RECONFIGURABLE COMPUTING

(work in progress)

Gheorghe M. Ştefan

– 2023 –

This document was prepared with LaTeX 2_ε

Introduction

Anyone can build a fast CPU. The trick is to build a fast system.

Seymour Cray (1925-1996)

The concept of reconfigurable computing, RC, has existed since the 1960s, when Gerald Estrin's paper [Estrin '60] proposed the concept of a computer made of a standard processor and an array of "reconfigurable" hardware. The main processor would control the behavior of the reconfigurable hardware.

RC emerges as a solution to two main shortcomings suffered by the computing systems that originated in the 1940s:

Turing tariff : Turing-based universal computer could perform any function, but not necessarily efficiently. The flexibility of Turing-based computers can still be used for complex computation, while for intense computation acceleration solutions will have to be found.

von Neumann bottleneck : the abstract model of computation disseminated by John von Neumann, in his 1945 report, provides a solution with a *small and complex* ***processor*** connected to a *simple and big* ***memory*** through a communication channel which isolates data and programs from the engine which work on data according to programs. The *Harvard abstract model*, issued in the same time, relaxes a little the limitation but the main problem remains: data are isolated from the processing engine.

The emergence of the SRAM-based field-programmable gate array (FPGA) in the 1980s boosted Reconfigurable Computing as a research and engineering field.

The book is structured in eight chapters organized in three parts followed by three appendixes.

First part : **PRELIMINARIES**, introduces the theoretical and historical context in which reconfigurable computation appears and will develop.

- **First chapter** : ***History***, answers, from a historical perspective, the question: why reconfigurable computing?
- **Second chapter** ***Why do we need reconfigurable computation?***, describes the current spectrum of possibilities offered by computer science.
- **Third chapter** : ***System-level Organization for Reconfigurable Computation***, provide a general description of a reconfigurable computing system.
- **Fourth chapter** : ***Mathematical Models of Computation***, is a short review of the main mathematical models of computation involved in defining what the reconfigurable computation could be.

Second part : **DIGITAL HIERARCHY**, is a review of the hardware resources we have to use circuits as a computability model.

Fifth chapter : ***Digital System Hierarchy***, describes the circuits that can be used to design the accelerator part of the reconfigurable computing system. It starts with combinational circuits and ends with mono-core programmable computing systems.

Sixth chapter : ***Cellular System Hierarchy***, deals with the hierarchy of cellular systems starting with cellular automata and describes various systems improved functionally by adding global loops.

Seventh chapter : ***Recursive Hierarchy***, introduces an abstract model for parallel computing based on the Stephen Kleene's partial recursive functions model.

Third part : *RECONFIGURABLE SYSTEMS*, provides the main techniques involved in supporting the implementation of RC systems.

Eight chapter : ***Optimizing Reconfigurable Systems***, deals with the optimization of the way the code must be written iin order to optimize the hardware of the accelerator.

Ninth chapter : ***Optimizing Pseudo-reconfigurable Systems***, provides an efficient solution for designing the functionality of the accelerator.

APPENDIXES **Composition: the only independent rule in Kleene's model** : the proof that out of three rules in the partial recursive model, only the first, the composition, is independent, the other two can be expressed as a composition of specific compositions.

How to instantiate DSP48E1 : provide the Verilog code used to instantiate the DSP48E1 module.

ConnexArray^{TM} simulator : provides the full description of the simulator for the generic ConnexArray^{TM} used as accelerator for the pseudo-reconfigurable version proposed in the last capitol of this book.

What do we expect in the post-Moore era? Improvements in computing performance will come from technologies at the "Top" of the computing hierarchy, not from those at the basic technological level (transistors). Thus we will assist to the reversing of the historical trend.

In the hardware field, processor simplification and domain specialization are expected to add performance in the near future.

It is important to specify what means processor *simplification*! We will define carefully what means *simple* (see Subsection 2.4.2).

And in terms of specialization, we will highlight several levels at which it can be applied. From this perspective, we will distinguish between reconfigurable and pseudo-reconfigurable computation.

Contents

Contents (detailed)

Part I

PRELIMINARIES

Chapter 1

History

The history of computation consists of few independent threads, starting in Antiquity. The history starts with an imaginary thread, a conceptual thread and a factual thread. Initially, the concepts and the objects evolved independently. At their mature stage, stimulated by the sad event of World War II, the conceptual evolution interferes with the physical implementation and the IT era begins. Application-driven history gradually emerges around 1971 when the conceptual approach reaches a maturity that slows down the theoretical approaches. In parallel with these threads, along the history, has been manifested and it still manifests also an imaginary thread. One of the main driving force in any domain is the human will and imagination. Therefore, we cannot ignore an ever developing imaginary history of the computing technology and its applications.

1.1 Imaginary history

At the beginning is always an image, a dream.

1.1.1 Antiquity

Hephaestus & Vulcan

Greek god Hephaestus is the god of technology, blacksmiths, craftsmen and artisans. Hephaestus made a bronze giant called *Talos* that would patrol around the island and throw rocks at enemy ships.

A roman counterpart of Hephaestus is Vulcan: made slave-girls of gold for himself.

Pygmalion

Pygmalion was a mythical character who, in search of perfection, sculpted in ivory the image of a perfect woman with whom he later fell in love and the goddess Aphrodite gave life to the statue.

We are always dealing with the human being's aspiration to correct the imperfections of nature through artifacts. Pygmalion seems to be a transhumanist *avant la lettre*.

1.1.2 Middle Ages

There were several stories and legends in the Middle Ages that involved the creation of artificial beings or creatures. These stories often reflected the beliefs and fears of medieval society regarding the power of human beings to create and control life.

Golem

Golem (Prague, ∼1500) is a metaphor for a brainless entity who serves man under controlled conditions but is hostile to him under others. The earliest known written account of how to create a golem can be found in Jewish Tradition (1165-1230).

Artificial animals and creatures at the court of Emperor Frederick II

There were also stories of artificial animals and creatures, such as the legendary mechanical eagle of Holy Roman Emperor Frederick II (1194-1250), which was said to be able to fly and to emit various sounds and cries.

Brazen Head

Another famous example is the story of the Brazen Head, a mechanical or artificial head that was said to be able to answer any question put to it. According to legend, the head was created by the medieval scholar and philosopher Roger Bacon (1220-1292), who was said to have used his knowledge of natural philosophy to imbue the head with the power of speech and reason.

Homunculus

Another example is the legend of the Homunculus, a tiny human-like creature that was said to be created by alchemists through the use of special substances and rituals. The Homunculus was believed to possess magical powers and to be able to perform various tasks, including the transmutation of metals and the creation of life. Paracelsus (1493–1541) is credited with the first mention of Homunculus in *De homunculis* (c. 1529–1532), and *De natura rerum* (1537)

Overall, these stories and legends reflected the fascination and curiosity of medieval society with the idea of artificial life and the power of human beings to create and control it.

1.1.3 Modernity

Frankenstein‘s Creature

Mary Shelley (1797-1851, wife of the poet Percy Shelley and daughter of Mary Wollstonecraft a founding figure of feminism) published in 1818 the novel em Frankenstein about a brilliant but unorthodox scientist, Dr. Victor Frankenstein, who rejects the artificial man he created; Creature escapes and later swears revenge.

Offenbach’s Olympia

Jacques Offenbach (1819-1880) in his *The Tales of Hoffmann* opera finished in 1880 introduced the character *Olympia*, a mechanical or an animatronical doll.

Karel Capek’s Robota

Robot (*robota* in Russian) is coined by Karel Capek 1920 R.U.R. is a 1920 science fiction play by the Czech writer Karel Čapek. R.U.R. stands for *Rossumovi Univerzální Roboti* (Rossum’s Universal Robots). The English phrase ”Rossum’s Universal Robots” has been used as a subtitle.

Fritz Lang's Metropolis

In 1927, German film maker Fritz Lang made the science fiction film Metropolis. The script contains the construction of a robot that acquires perfect human appearance and behavior. The artificial product has the ability to disrupt the behavior of the masses. It's far beyond what was imagined for Capek's robot.

1.1.4 Contemporary

Scary Science Fiction (SF) scenarios about Artificial Intelligence (AI) [Tegmark '17].

Max Tegmark: Life 1.0 referring to biological origins, Life 2.0 referring to cultural developments in humanity, and Life 3.0 referring to the technological age of humans.

We must make distinctions between the three main human brain behaviors: Spiritually – Imaginary – Rationally. AI refers mainly to the third.

1.2 Conceptual history

1.2.1 Binary Arithmetic to the Chinese

In *Discourse on the Natural Theology of the Chinese*, Gottfried Wilhelm von Leibniz (1646-1716) mentioned that the 64 hexagrams of em I Ching ($\sim$1000 BC) represent the binary arithmetic used a few thousand years ago in China.

1.2.2 Epimenides of Crete

Karl Jaspers (1883–1969) introduced the concept of an Axial Age in his book *The Origin and Goal of History*, published in 1949. During this period new ways of thinking emerged in Persia, India, China, Greece and Roman Empire, in a singular synchronous development, without any effective direct cultural contact between all of the Eurasian cultures. Jaspers emphasized prominent thinkers from this period who had a profound influence on future sciences, philosophies and religions.

In this Axial Age, around 7th or 6th century BC, Epimenides of Cnossos (Crete) was a semi-mythical Greek seer and philosopher-poet which started the conceptual development leading to the contemporary computer science. In one day he uttered a sentence which troubled the inquisitive minds from everywhere for the next two and half millennia:

"Cretans, always liars."

The sentence is equivalent with *"I lie"*, and is undecidable: its truth value can not be decided.

1.2.3 Liar's paradox in Middle Ages

In the Middle Ages, the paradox was studied and commented on by many philosophers and theologians, who tried to resolve the contradiction it presents.

Boethius

Boethius is one of the earliest recorded commentaries on the paradox in the Middle Ages. He is a Roman philosopher who lived in the 6th century CE. In his work "Consolation of Philosophy," Boethius discusses the paradox and argues that it arises from a confusion of terms and concepts.

Peter Abelard

In the 12th century, the paradox was further discussed by the French philosopher and theologian Peter Abelard, who used it to criticize the doctrine of divine omnipotence. Abelard argued that the paradox shows that there are limits to what even an omnipotent God can do, since he cannot make a statement that is both true and false at the same time.

William of Ockham

In the 14th century, the English logician William of Ockham used the paradox to argue against the idea of universal propositions. Ockham argued that the paradox shows that there are no universal propositions that can be true or false in all cases, since there are some statements that cannot be consistently evaluated as true or false.

Overall, the paradox of the liar was a topic of interest and debate among medieval philosophers and theologians, who used it to explore the limits of language, logic, and the nature of truth.

1.2.4 Gottfried Wilhelm von Leibniz

Binary representation

In 1703, Leibniz published in the *Mémories de l'Académie Royale des Sciences* his essay "Explication de l'arithmétique binaire, qui se sert des seules caractères 0 & 1; avec des remarques sur son utilité, et sur ce qu'elle donne de sens des anciennes figures chinoises de Fohy" where he explains how to perform addition, subtraction, multiplication and division using the binary representation for numbers.

Calculus ratiocinator

The *Calculus ratiocinator* is a a concept introduced by Leibniz related to *characteristica universalis*, an universal conceptual language. This concept could be related to both the hardware and software aspects of the modern digital computer.

1.2.5 George Boole

In 1847 George Boole (1815-1864) published *Mathematical Analysis of Logic* and in 1854 *An Investigation into the Laws of Thought, on which are Founded the Mathematical Theories of Logic and Probabilities* which underpins what we now call *Boolean algebra*, a successful attempt to formalize Aristotelian logic. It is thus made available to innovators an instrument that will be used to substantiate the science of calculus as a ***decision*** tool in the first place, and only through a second approach as a calculation tool. Indeed, computation is mainly about deciding. Numerical computation comes only as a consequence. At first it was the true/false alternative, and only then the 0/1 alternative.

1.2.6 1900-1928: David Hilbert

David Hilbert (1862-1943) one of the most influential and universal mathematicians of the 19th and early 20th centuries.

At the International Congress of Mathematicians held in Bologna, Hilbert revisited to the second of the 23 problems posed in his 1900 paper *Mathematische Probleme* [Hilbert 1900], asking [Isaacson '85]:

1. Was its set of rules complete, so that any statement could be proved (or disproved) using only the rules of the system?

2. Was it consistent, so that no statement could be proved true and also proved false?
3. Was there some procedure that could determine whether a particular statement was provable, rather than allowing the possibility that some statements (such as enduring math riddles like Fermat's last theorem, Goldbach's conjecture, or the Collatz conjecture) were destined to remain in undecidable limbo?

Hilbert thought that the answer to the first two questions was yes, making the third one moot [Isaacson '85].

In mathematics and computer science, the *Entscheidungsproblem* (German for "decision problem") is a challenge posed by David Hilbert and Wilhelm Ackermann in 1928 [Hilbert & Ackermann '28]. By the completeness theorem of first-order logic, a statement is universally valid if and only if it can be deduced from the axioms, so the Entscheidungsproblem can also be viewed as asking for an algorithm to decide whether a given statement is provable from the axioms using the rules of logic.

As late as 1930, Hilbert believed that there would be no such thing as an unsolvable problem.

The *Entscheidungsproblem* is related to Hilbert's tenth problem (from Hilbert's address of 1900 to the International Congress of Mathematicians in Paris [Hilbert 1900]), which asks for an algorithm to decide whether Diophantine equations have a solution. The non-existence of such an algorithm, established by Yuri Matiyasevich in 1970, also implies a negative answer to the *Entscheidungsproblem*.

Hilbert's address of 1900 to the International Congress of Mathematicians in Paris is perhaps the most influential speech ever given to mathematicians, given by a mathematician, or given about mathematics. In it, Hilbert outlined 23 major mathematical problems to be studied in the coming century.

1.2.7 1931: Kurt Gödel

Kurt Friedrich Gödel (1906-1978) The *logician* Gödel published his two incompleteness theorems in 1931 when he was 25 years old, one year after finishing his doctorate at the University of Vienna. The first incompleteness theorem states that for any self-consistent recursive axiomatic system powerful enough to describe the arithmetic of the natural numbers (for example Peano arithmetic), there are true propositions about the naturals that cannot be proved from the axioms. To prove this theorem, Gödel developed a technique now known as Gödel numbering, which codes formal expressions as natural numbers.

The Austrian-born logician Kurt Gödel polished off **the first two** Hilbert's questions with unexpected answers: no and no. In his "incompleteness theorem", he showed that there existed statements that could be neither proved nor disproved.

1.2.8 1936: Church – Kleene – Post – Turing

What a synchronicity! Indeed, the logician Gödel's approach triggered four *mathematicians* to provide independently mathematical versions to the logical challenge raised by the *Entscheidungsproblem* (the third of Hilbert's questions).

Alonzo Church

Alonzo Church (1903-1995): The lambda calculus emerged in his 1936 paper showing the unsolvability of the *Entscheidungsproblem*. This result preceded Alan Turing's work on the halting problem, which also demonstrated the existence of a problem unsolvable by mechanical means. Church and Turing then showed that the lambda calculus and the Turing machine used in Turing's halting problem were equivalent in capabilities, and subsequently demonstrated a variety of alternative "mechanical processes for computation". This resulted in the Church–Turing thesis.

The lambda calculus influenced the design of the LISP programming language and functional programming languages in general.

Stephen Kleene

Stephen Cole Kleene (1909-1994) is best known as a founder of the branch of mathematical logic known as recursion theory, which subsequently helped to provide the foundations of theoretical computer science.

Emil Post

Emil Leon Post (1897-1957) developed in 1936, independently of Alan Turing, a mathematical model of computation that was essentially equivalent to the Turing machine model. This model is sometimes called "Post's machine" or a Post–Turing machine.

Alan Turing

"When the great Cambridge math professor Max Newman taught Turing about Hilbert's questions, the way he expressed the *Entscheidungsproblem* was this: Is there a "mechanical process" that can be used to determine whether a particular logical statement is provable?" [Isaacson '85]

Alan Mathison Turing (1912-1954) in 1936 published his paper "On Computable Numbers, with an Application to the *Entscheidungsproblem*". It was published in the Proceedings of the London Mathematical Society journal in two parts, the first on 30 November and the second on 23 December. In this paper, Turing reformulated Kurt Gödel's 1931 results on the limits of proof and computation, replacing Gödel's universal arithmetic-based formal language with the formal and simple hypothetical devices that became known as Turing machines. The *Entscheidungsproblem* (decision problem) was originally posed by German mathematician David Hilbert in 1928. Turing proved that his "universal computing machine" would be capable of performing any conceivable mathematical computation if it were representable as an algorithm. He went on to prove that there was no solution to the decision problem by first showing that the halting problem for Turing machines is undecidable: It is not possible to decide algorithmically whether a Turing machine will ever halt.

1.2.9 1940s: abstract models of computation

The transition from a mathematical model of computation to a realizable physical structure was enabled by the abstract models of computers. Purely mathematical models contain descriptions that assume concepts that have no physical counterpart, such as infinity. For this reason abstract models were necessary.

1943: Neural nets

Warren S. McCulloch, Walter H. Pitts introduced the neural network model for computation [McCulloch '43].

1944: Harvard abstract model

The term originated from the Harvard Mark I, or IBM Automatic Sequence Controlled Calculator (ASCC), an electromechanical computer, which stored instructions on punched tape and data in electromechanical counters.

1945: von Neumann abstract model

John von Neumann wrote up a description titled *First Draft of a Report on the EDVAC* [Neumann '45] based on the work of Eckert and Mauchly. It was unfinished when his colleague Herman Goldstine circulated it, and bore only von Neumann's name (to the consternation of Eckert and Mauchly).

1.3 Factual history

1.3.1 Antikythera mechanism

The Antikythera mechanism is believed to be designed to predict eclipses. It has been designed and constructed by Greeks and is dated to about 200 BC to 80 BC. It is a clockwork mechanism composed of more than 30 engaged bronze gears.

1.3.2 Hero of Alexandria

Hero of Alexandria (c.10-c.70) was the first to build a vending machine; when a coin was introduced via a slot on the top of the machine, a set amount of holy water was dispensed.

Hero described the construction of the aeolipile (a version of which is known as Hero's engine) which was a rocket-like reaction engine and the first-recorded steam engine

1.3.3 Gerbert of Aurillac

In 996 A.D., Gerbert of Aurillac (Pope Sylvester II from 999) (946-1003) invented the first weight-driven mechanical pendulum clock at a monastery in Magdeburg in Germany. The clock's mechanism would ring bells at regular intervals throughout the day to call his fellow monks to prayer.

Gerbert took the idea of the abacus calculator from a Spanish Arab. But the calculations with his abacus were extremely difficult, because the people of his day used only Roman numerals.

1.3.4 Wilhelm Schickard

Johannes Kepler, claimed that the drawings of a calculating clock, predating the public release of Pascal's calculator by twenty years, had been discovered in two unknown letters written by Wilhelm Schickard (1592-1635) to him in 1623 and 1624.

1.3.5 Blaise Pascal

Pascaline: Blaise Pascal (1623-1662) was led to develop a calculator by the laborious arithmetical calculations required by his father's work as the supervisor of taxes in Rouen. He designed the machine to add and subtract two numbers directly and to perform multiplication and division through repeated addition or subtraction.

1.3.6 Gottfried Wilhelm von Leibniz

In *Machina arithmetica in qua non additio tantum et subtractio sed et multiplicatio nullo, diviso vero paene nullo animi labore peragantur*, written in 1685, Gottfried Wilhelm (von) Leibniz (1646-1716) described an arithmetic machine he had invented that was made by linking two separate machines, one to perform additions/subtractions and one for multiplications/divisions.

1.3.7 Joseph Marie Charles *dit* Jacquard

The Joseph Jacquard (1752-1834) Loom is a mechanical loom that uses pasteboard cards with punched holes, each card corresponding to one row of the design. Multiple rows of holes are punched in the cards and the many cards that compose the design of the textile are strung together in order.

1.3.8 Charles Babbage

Difference engine

Charles Babbage (1791-1871) began in 1822 with what he called the difference engine, made to compute values of polynomial functions. It was created to calculate a series of values automatically. By using the method of finite differences, it was possible to avoid the need for multiplication and division.

Analytical Engine

The Analytical Engine marks the transition from mechanised arithmetic to fully-fledged general purpose computation. It is largely on it that Babbage's standing as computer pioneer rests.

The major innovation was that the Analytical Engine was to be programmed using punched cards: the Engine was intended to use loops of Jacquard's punched cards to control a mechanical calculator, which could use as input the results of preceding computations.[157][158] The machine was also intended to employ several features subsequently used in modern computers, including sequential control, branching and looping. It would have been the first mechanical device to be, in principle, Turing-complete.

1.3.9 Ada Byron, Countess of Lovelace

Augusta Ada King, Countess of Lovelace (née Byron; 1815–1852) chiefly known for her work on Charles Babbage's proposed mechanical general-purpose computer, the Analytical Engine. She was the first to recognise that the machine had applications beyond pure calculation, and published the first algorithm intended to be carried out by such a machine. As a result, she is sometimes regarded as the first to recognise the full potential of a "computing machine" and one of the first computer programmers.

1.3.10 Herman Hollerith

Herman Hollerith (1860-1929) developed an electromechanical tabulating machine for punched cards to assist in summarizing information and, later, in accounting. His invention of the punched card tabulating machine, patented in 1889, marks the beginning of the era of semiautomatic data processing systems, and his concept dominated that landscape for nearly a century. He was the founder of the Tabulating Machine Company that was amalgamated (via stock acquisition) in 1911 with three other companies to form a fifth company, the Computing-Tabulating-Recording Company, which was renamed IBM in 1924. Hollerith is regarded as one of the seminal figures in the development of data processing.

1.3.11 Claude Shannon & Thomas Flowers

Implementing electro-mechanically Boolean functions

Claude Elwood Shannon (1916-2001) known as "the father of information theory". Shannon is noted for having founded information theory with a landmark paper, *A Mathematical Theory of Communication*, that he published in 1948.

He is also well known for founding digital circuit design theory in 1937, when -— as a 21-year-old master's degree student at the Massachusetts Institute of Technology (MIT) -— he wrote his thesis demonstrating that electrical applications of Boolean algebra could construct any logical numerical relationship.

Implementing electronically Boolean functions

Thomas Harold Flowers (1905-1998). From 1935 onward, he explored the use of electronics for telephone exchanges and by 1939, he was convinced that an all-electronic system was possible. A background in switching electronics would prove crucial for his computer designs [Copeland '06].

1.4 Merged history

The triad *Math & Logic – War – Technology* (Ethos – Pathos – Logos) provides the context of the emergence of the Information Technology (IT) era.

WWII made the turbulent transition toward IT industry.

1.4.1 Colossus

Colossus was a set of computers developed by British code-breakers in the years 1943–1945 to help in the cryptanalysis of the Lorenz cipher. Colossus used thermionic valves (vacuum tubes) to perform Boolean and counting operations. Colossus is thus regarded as the world's first programmable, electronic, digital computer, although it was programmed by switches and plugs and not by a stored program.

A Colossus computer was thus not a fully Turing complete machine. The notion of a computer as a general purpose machine — that is, as more than a calculator devoted to solving difficult but specific problems — did not become prominent until after World War II.

1.4.2 ENIAC – EDVAC

ENIAC

Electronic Numerical Integrator and Computer was the first electronic general-purpose computer. It was Turing-complete, digital and able to solve "a large class of numerical problems" through reprogramming.

ENIAC was completed in 1945 and first put to work for practical purposes on December 10, 1945. ENIAC was designed by *John Mauchly* and *J. Presper Eckert* of the University of Pennsylvania, U.S.

By the end of its operation in 1956, ENIAC contained 20,000 vacuum tubes; 7,200 crystal diodes; 1,500 relays; 70,000 resistors; 10,000 capacitors; and approximately 5,000,000 hand-soldered joints. It weighed more than 27 t, was roughly $2.4m \times 0.9m \times 30m$ in size, occupied $167m^2$ and consumed $150kW$ of electricity.

EDVAC

Electronic Discrete Variable Automatic Computer: unlike its predecessor, the ENIAC, it was binary rather than decimal, and was designed to be a stored-program computer. Functionally, EDVAC was a binary serial computer with automatic addition, subtraction, multiplication, programmed division and automatic checking with an ultrasonic serial memory[1] capacity of 1,000 34-bit words. EDVAC's average addition time was 864 microseconds and its average multiplication time was 2,900 microseconds.

ENIAC inventors John Mauchly and J. Presper Eckert proposed EDVAC's construction in August 1944, and design work for EDVAC commenced before ENIAC was fully operational. The design would implement a number of important architectural and logical improvements conceived during the ENIAC's construction and would incorporate a high-speed serial-access memory. Like the ENIAC, the EDVAC was built for the U.S. Army's Ballistics Research Laboratory at the Aberdeen Proving Ground by the University of Pennsylvania's Moore School of Electrical Engineering. Eckert and Mauchly and the other ENIAC designers were joined by John von Neumann in a consulting role; von Neumann summarized and discussed logical design developments in the 1945 *First Draft of a Report on the EDVAC*.

1.4.3 Princeton computer

The IAS machine was the first electronic computer to be built at the Institute for Advanced Study (IAS) in Princeton, New Jersey. It is sometimes called the von Neumann machine, since the paper describing its design was edited by John von Neumann, a mathematics professor at both Princeton University and IAS. The computer was built from late 1945 until 1951 under his direction.

The IAS machine was a binary computer with a 40-bit word, storing two 20-bit instructions in each word. The memory was 1,024 words (5.1 kilobytes). Negative numbers were represented in "two's complement" format. It had two general-purpose registers available: the Accumulator (AC) and Multiplier/Quotient (MQ). It used 1,700 vacuum tubes. The memory was originally designed for about 2,300 RCA Selectron vacuum tubes.

It weighed about 1,000 pounds (450 kg).[11]

It was an asynchronous machine, meaning that there was no central clock regulating the timing of the instructions. One instruction started executing when the previous one finished. The addition time was 62 microseconds and the multiplication time was 713 microseconds.

1.4.4 IBM entered the scene

The IBM 701 Electronic Data Processing Machine, known as the Defense Calculator while in development, was IBM's first commercial scientific computer, which was announced to the public on April 29, 1952. It was designed by Nathaniel Rochester and based on the IAS machine at Princeton.

1.4.5 John Backus: first involvement

John Warner Backus (1924-2007): directed the team that invented and implemented FORTRAN, the first widely used high-level programming language, and was the inventor of the Backus–Naur form (BNF), a widely used notation to define formal language syntax.

1.4.6 Computer architecture

Brooks went on to help develop the IBM System/360 (now called the IBM zSeries) line of computers, in which "architecture" became a noun defining "what the user needs to know".

In [Amdahl '64] [Blaauw '64] the concept of computer architecture (low level machine model) is introduced to allow independent evolution for the two different aspects of computer design, which have different rate of evolution: software and hardware; thus, there are now on the market few stable and successful architectures, such as x86, ARM, PowerPC.

1.4.7 John Backus: second involvement

He later did research into the function-level programming paradigm, presenting his findings in his influential 1977 Turing Award lecture "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs"

1.4.8 Parallel computing enter the scene on the back door

While for mono-core computing there are the following stages [**?**]:

- **1936 – mathematical computational models**: four equivalent models are published [Turing '36] [Church '36] [Kleene '36] [Post '36] (all reprinted in [Davis '04]), out of which the *Turing Machine* offered the most expressive and technologically appropriate suggestion for future developments leading eventually to the mono-core, sequential computing

- **1944-45 – abstract machine models**: the MARK 1 computer, built by IBM for Harvard University, consecrated the *Harvard abstract model*, while the von Neumann's report [Neumann '45] introduced the *von Neumann abstract model*; these two concepts backed the *RAM* (random access machine) abstract model used to evaluate algorithms for sequential machines
- **1952 – manufacturing in quantity**: IBM launched *IBM 701*, the first large-scale electronic computer
- **late 1953 – high-level programming language**: John W. Backus submitted a proposal to his superiors at IBM to develop a more practical alternative to assembly language for programming their IBM 704 mainframe computer; a draft specification for "The IBM Mathematical Formula Translating System" was completed by November 1954; the first manual for FORTRAN appeared in October 1956; with the first FORTRAN compiler delivered in April 1957.
- **1964 – computer architecture**: in [Blaauw '64] the concept of *computer architecture* (low level machine model) is introduced to allow the independent development for the two different aspects of computer design which have different rate of evolution: software and hardware; thus, there are now on the market few stable and successful architectures, such as x86, ARM, ….

for parallel computing we are faced with a completely distorted evolution; let us see its first stages:

- **1962 – manufacturing in quantity**: the first symmetrical MIMD engine is introduced on the computer market by Burroughs
- **1965 – architectural issues**: Edsger W. Dijkstra formulates in [Dijkstra '65] the first concerns about specific parallel programming issues
- **1974-76 – abstract machine models**: proposals of the first abstract models (bit vector models in [Pratt '74] and PRAM models in [Fortune '78], [Goldschlager '82]) start to come in after almost two decades of non-systematic experiments (started in the late 1950) and the too early market production
- **? – mathematical computation model**: no one yet really considered it, regrettably confused with abstract machine models, although it is there waiting for us (see Kleene's mathematical model for computation [Kleene '36]).

1.4.9 RISC

The term RISC (Reduction Instruction Set Computer) was coined by David Patterson. It means processors with an architecture characterized by:

load-store mechanism : divides instructions into two categories: ALU operations between registers, and memory access as simple load and store between memory and registers instead complex multi-indirected memory access modes

one-word instructions : instructions are coded in on word; even when an immediate value is involved, it is taken into account that in most cases small values are involved that can be encoded with a small number of bits making it unnecessary to add an additional word to specify the value.

one-cycle execution : using a Harvard abstract model, a load-store mechanism and one-word instructions, it is possible to design a processor which execute each instruction in one clock cycle

only most frequent instructions : because the statistics compiled on large program databases showed an uneven distribution of the use of the instructions in the established ISAs, it was decided to keep in the ISA only the frequently used instructions, provided that the omitted ones could be made by a sequence of those maintained

which will lead to:

> *The goal of any instruction format should be: 1. simple decode, 2. simple decode, and 3. simple decode. Any attempts at improved code density at the expense of CPU performance should be ridiculed at every opportunity.* [Weaver '09]

The resulting reductions in complexity and size have increased the number of registers, increased clock frequency and reduced power consumption.

The first implementations:

IBM 801 : based on a statistical study launched in the mid-1970s that highlighted the need to increase the number of registries and the possibility of removing from current ISAs a significant number of complex instructions that compilers "ignored".

Berkeley RISC : when David Patterson was sent in 1979 on a sabbatical from University of California, Berkeley to help DEC to improve the VAX microcode, he discovered that if the microcode was removed, the programs would run faster. The microcode was responsible for interpreting the complex instructions. Then, removing the complex instructions from ISA becomes a solution for improving processor's performance.

MIPS : stands for Microprocessor without Interlocked Pipeline Stages, a project which came from a graduate course of John L. Hennessy At Stanfort University; it produced a functioning system in 1983.

Since 2010 a new ***open source*** ISA, RISC-V, has been under development at the University of California, Berkeley.

1.4.10 FPGA & Adaptive Computer Acceleration Platform

1985 is considered the year of the birth of FPGA (Field-Programmable Gate Array) technology with the founding of Xilinx, although in 1983 the founding of Altera led to the first forms of this technology.

ACAP (Adaptive Computer Acceleration Platform) is the technology that naturally emerges from the FPGA approach by the fact that users of the last decades have expressed preferences that have outlined specific structures that can be implemented as standardizable IPs.

> *An ACAP is a heterogeneous, hardware adaptable platform that is built from the ground up to be fully software programmable. An ACAP is fundamentally different from any multi-core architecture in that it provides hardware programmability but the developer does not have to understand any of the hardware detail.* [Banerjee '19]

1.5 User-driven evolution: Computation as General-Purpose Technology

1.5.1 Microsoft's Surface

There comes a time when users begin to define and produce computing equipment for their own use. A significant example is Microsoft's Surface series of touchscreen-based personal computers, tablets and interactive whiteboards.

Microsoft first announced Surface at an event on June 18, 2012, as the first major initiative by Microsoft to integrate its Windows operating system ***with its own hardware***, and is the first PC designed and distributed solely by Microsoft.

1.5.2 Google's Tensor Processing Unit

In 2015, a step forward is taken: an end-user begins to define and produce ICs for his own use. Google began producing and using Tensor Processing Unit (TPU) as an AI accelerator ASIC developed specifically for neural network machine learning, particularly using Google's own TensorFlow software.

From 2018, the circuits from the TPU family are also made available to other users. We are witnessing a mechanism by which the new technology is developed by an end-user and then made available to other users. The development of general purpose machines (processors, computers) that are made available to different users is replaced by a process in a reverse way, users of devices dedicated to a particular field promote products that are disseminated as general purpose products. We can also exemplify by GPUs used as GPGPU. Note the oxymoronic formulation: General-Purpose Graphic Processing Unit.

1.5.3 Apple's M1

M1 = (8-core ARM CPU + GPU + Neural Engine + ... + Cache) + DRAM: a first example of Accelerator-Level Parallelism.

> *"So the physical RAM modules are still separate entities, but they are sitting on the same green substrate as the processor. ... Apple calls its approach a "Unified Memory Architecture" (UMA)."*

1.5.4 Tesla's Artificial Intelligence & Autopilot

FSD Chip Build AI inference chips to run our Full Self-Driving software, considering every small architectural and micro-architectural improvement while squeezing maximum silicon performance-per-watt.

1.5.5 Hadoop & Big-Data

Hadoop is an open source processing system that manages distributed data processing and storage for ***Big Data*** applications for scalable clusters. It manages an ecosystem of Big Data applications that are used to support advanced data mining and machine learning.

1.5.6 The Next Target: Artificial General Intelligence

AGI may be the ability of computers to solve problems in a way that human beings do, using intuition and common sense in addition to formal skills.

Current AI techniques can be considered "narrow AI" or "weak AI" because they refer to well-defined areas of competence, areas in which they currently exceed human performance.

AGI is a goal that is not only difficult to achieve, but, first of all, very difficult to define.

Artificial General Intelligence (AGI)

1.6 Application-driven history

A significant turning point came in 1971, when Intel launched the first successful silicon memory (1103) and the first one-chip microprocessor (4004). In the same year, e-mail (`@Mail`) and the wireless network appeared. It is the moment when the evolution of the field of computing begins to be more and more marked by applications oriented towards the big market. The computer and its applications, until then oriented towards government institutions, universities or corporate space, are beginning to be oriented towards the consumer market. The main consequence will be, from that moment, the evolution under the pressure of the criteria imposed by the market.

Is it a coincidence that one last important theoretical issue – *NP-completeness* – is being addressed this year? An era of theoretical research seems to be coming to an end, and an era of applied developments is beginning.

It is worth mentioning some of the stages completed in the last half century [Garfinkel '15]:

1972: HP-35 pocket calculator destroyed the market for the slide rules. HP-35 because it had 35 keys. Used to run at 200 KHz programs no longer then 768 instructions.

1973: first cell phone call in April 3, on Sixth Avenue in New-York City between Fifty-Third and Fifty Four Streets.

1973: Alto the first personal computer developed by Xerox equipped with a graphic-user interface.

1975: *Adventure* the first text-based simulation used as a game.

1983: 3-D printing is an additive manufacturing technology which fabricate objects in the field starting from raw materials.

1983: first laptop comes preloaded with a rudimentary word processor and a basic spreadsheet program. It comes on the market under the specifications of RadioShack® TRS-80 Model 100 equiped with an 8-bit Intel 80C85 microprocessor. The operating system, written almost entirely by Bill Gates, was loaded in a 32 KB of ROM.

1983: MIDI computer music interface helped to put music creation into the hands of more users to generate great music without a professional performer because the computer played the music. (MIDI stands for Musical Instrument Digital Interface.)

1984: text-to-speech technology commoditized by DEC by its standalone appliance DECtalk

1984: virtual reality a term coined by Jaron Lanier as being the outcome of running programs written in Virtual Programming Language on specific hardware.

1984: Verilog is a Hardware Description Language used by designers to describe, simulate, and synthesize digital systems.

1985: desktop publishing allowed anybody to generate high quality documents with a tight control on fonts and graphics.

1988: CD-ROM stands for Compact Disc - Read-Only Memory; it is used to store music, video and software.

1989: www which stands for world wide web, transformed internet connection into a dominating technology connecting virtually every person on the planet.

1990: GPS which stands for Global Positioning System, is a consumer navigation system based on old radio waves technologies.

1992: Boston Dynamics a robotics company maker of biped and quadruped robots capable of traversing rough landscapes.

1992: First mass-market web browser called Mosaic changed the way the www is accessed from a professional procedure to an easy way which does not imply technical expertise.

1993: Apple Newton electronic organizer; handwriting recognizer based on Arcon RISC MAchine (ARM) a low-power computationally powerful controller.

1995: E-Commerce becomes possible as soon as the most important ingredient - security - has been implemented. Netscape *Secure Socket Layer* allowed consumers to securely send credit card numbers over the Internet.

1997: Deep-Blue beat world chess campion Garry Kasparov

1997: E-Ink electronic paper display (EPD) is a reflective display that is visible in direct sunlight.

1998: Google based on an algorithm to rank organizes the pages on WWW. The algorithm takes into account the number of links the quality of pages. A page is important if it is pointed from a big numbers of important pages. In 2006 *Google* becomes a verb.

2001: Wikipedia is the result of a mass-collaborative effort of organizing knowledge as a continuous process which may containing errors, mistakes, biased attitude, but being open to self-correcting mechanisms, till the end it is able to provide a very useful image of the current stage of knowledge.

2004: Facebook is a (too) free communication platform. Provides a solution to the desire to connect with and learn about other people.

2007: iPhone invented by Apple puts together telephony, messaging, internet access, music, color screen, touch-based interface. The main big things associated with iPhone: specialized programs called *apps*.

2008: Blockchain a collection of transactions – *blocks* – managed in the most possible secure mode thus allowing the development of the *criptocoin* environment and many other distributed applications.

2022: chatbot the chat robot *chatGPT* where GPT stands for ***generative pre-trained transformer*** .

1.7 Programming paradigms

Programming languages:

- low level languages
  - machine languages: uses the instructions' numeric values of instructions directly
  - assembly languages: generate executable machine code from assembly code where for each statement there is a machine instruction; uses mnemonic codes to refer to machine code instructions
- high level languages
  - imperative languages: generate explicit statements about how the machine state changes
    - FORTRAN: scientific applications (1953-1957)
    - ALGOL (1958)
    - COBOL: business applications (1959)
    - Basic
    - Pascal
    - C
    - ...
  - declarative languages: describe what a computation should perform, without specifying detailed state changes

    - functional languages: use evaluation of mathematical functions instead of explicit state change (Lisp (1958), Clojure (2007))
    - logic languages: involves explicit mathematical logic for programming (Prolog)
  - multi-paradigm programming languages: programming languages that supports more than one programming paradigm.
    - Python: supports multiple programming paradigms, including procedural, object-oriented, and functional programming
    - ...

- library of functions
  - BLAS
  - Eigen
  - Tensorflow
  - ONNX (Open Neural Network Exchange) is an open format built to represent machine learning models. ONNX defines a common set of operators - the building blocks of machine learning and deep learning models
  - ...

1.8 The Qubit

Quantum computing is a type of computation based on the collective properties of quantum states, such as superposition, interference, and entanglement, to perform computation. The elementary devices that perform quantum computations are qubits organized in what known as quantum computers.

The quantum computation performs the computation using a network of *quantum logic gates*. A quantum gate is a complex linear-algebraic generalization of boolean circuits.

In 2001, a team of IBM scientists factored the number 15 with a quantum computer that had 7 qubits. In 2019, IBM has launched *Q System One*, the first circuit-based commercial quantum computer.

Chapter 2

Why do we need reconfigurable computation?

In order to understand why we need reconfigurable computing, we will first have to briefly review the current state of the possibilities offered by hardware technologies.

2.1 Mono-core approach

The mono-core approach dominates the first decades of computer science and technology. It is based on the mathematical model proposed by Turin and Post, and on the ***abstract model*** promoted by von Neumann and the computer structure built at Harvard University.

There are two trends in the mono-core period: (1) increasing the capacity and speed of memories and (2) increasing the performance of processors.

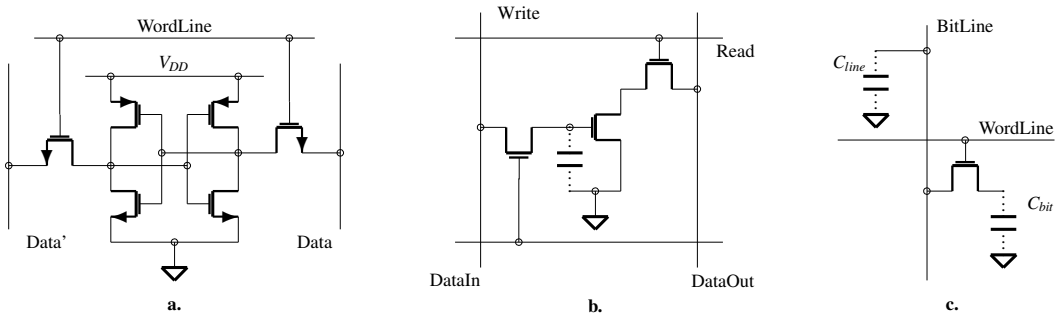

Figure 2.1: From static to dynamic memory cell. **a** The static 6-transistor cell. **b.** The dynamic 3-transistor cell. **c.** The dynamic one-transistor cell.

Today we have a few giga-bits per chip memories. The storage devices have evolved from the magnetic to silicon support, so as in the early 1970s, the first 1Kbit/chip silicon memory appeared: the Intel's 1103 chip (see Figure 2.2). Current density was possible because the one-bit storage cell evolved from the static version (see Figure 2.1.a) to the dynamic cell. First to a 3-transistor cell (Figure 2.1.b), then to the one transistor cell (Figure 2.1.c).

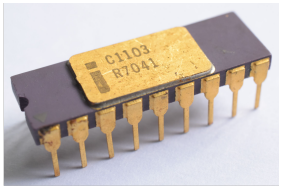

Figure 2.2: Intel's 1103 dynamic memory chip released in 1971.

In half a century, from 1Kb/chip (see Figure 2.2) we get, through the action of Moore's law, to $\sim$1 Gb/chip (see Figure 2.3). One million times more bits per chip!

Figure 2.3: HP - DDR4 - 16 GB - SO-DIMM 260-pin.

If in the case of memories we are dealing with a purely quantitative evolution, in the case of processors there have been several conceptual leaps. The first one was the transition form multi-chip processors to mono-chip processors. It happens gradually with Intel's 4004, continues with 8008 and 8080 but matures with x86 architecture. In parallel, Motorola, starting with its 6800 8-bit engine, is promoting 68000 series processors; with a much more elegant architecture, but supported by a less efficient management than the one promoted by Intel.

Then, the most important transition is from CISC (Complex Instruction Set computer) to RISC (Reduced Instruction Set Computer) processors.

2.1.1 CISC

Turing's UTM $\Rightarrow$ CISC: a computing engine working with a processor which interprets the code. A CISC processor is a microprogrammed machine which translates instructions in sequences of microinstructions. This mechanism allows the designed to define very complex instructions but cannot "persuade" the compiler to use them frequently.

2.1.2 RISC

No state UTM $\Rightarrow$ RISC [Ştefan '06]: which, instead interpreting the code executes it. Main consequences: smaller, simpler, faster, memory saver, energy aware processor. It has been proven that the

intermediation offered by microprogramming is not useful for general computing. But be careful! It may prove useful for specialized niche processors.

2.1.3 Memory wall

Originally theorized in 1994 by Wulf and McKee [Wulf '95], *memory wall* concept revolves around the idea that computer processing units (CPUs) are advancing at a fast enough pace that will leave memory (DRAM) stagnant.

CPU speed increased at an average rate of 55% per year from 1986 to 2000, whereas RAM speed increased by just 10% per year.

A partial solutions available to combat the problem of memory wall is the use of cached data. But this solution doesn't work efficiently for intense computing.

2.2 Multi-core approach

Multi-core computer = *ad hoc* construct

Backed by the concept of multi-threading.

2.3 Many-core approach

Many-core engine = application oriented accelerator

The oximoronic *General-purpose application-oriented processing unit*.

Example: GPGPU which stands for *"General-Purpose Graphics Processing Unit"*.

Must be backed by a mathematical model. What would you say about the Kleene's model [Kleene '36]? See a possible approach in [Stefan '14].

2.3.1 Heterogenous approach

Circuits represent a mathematical computing model for computation.

We are faced with the hierarchical distinctions:

- communication
  - between memory and the computation engine
  - between the components (cells) of the computation engine
- computation
  - complex computation
  - intense computation

2.3.2 Adaptive Compute Acceleration Platform (ACAP)

2.3.3 Accelerator-Level Parallelism

2.4 Complexity vs. Size

2.4.1 Circuit size vs. circuit complexity

The huge *size* of the actual circuits implemented on a single chip imposes a more precise distinction between *simple* circuits and *complex* circuits. When we can integrated on a single chip more than 10^9

components, the size of the circuits becomes less important than their complexity. Unfortunately we don't make a clear distinction between *size* and *complexity*. We say usually: "the complexity of a computation is given by the size of memory and by the CPU time". But, if we have to design a circuit of 100 million transistors it is very important to distinguish between a circuit having an uniform structure and a randomly structured ones [**?**].

As a consequence we must distinguish more carefully the concept of *size* by the concept of *complexity*.

Definition 2.1 *The* **size** *of a digital circuit, $S_{digital\ circuit}$, is given by the dimension of the physical resources used to implement it.*

◇

Definition 2.2 *The* **algorithmic complexity** *of a digital circuit, simply the* **complexity**, *$C_{digital\ circuit}$, has the magnitude order given by the minimal number of symbols needed to express its definition.*

◇

Definition 2.2 is inspired by Gregory Chaitin's definition for the algorithmic complexity of a string of symbols [Chaitin '77]. The *algorithmic complexity* of a string is related to the dimension of the smallest program that generates it. Our $C_{digital\ circuit}$ can be associated to the shortest unambiguous circuit description in a certain HDL (in the most of cases it is about a behavioral description).

Definition 2.3 *A* **simple circuit** *is a circuit having the complexity much smaller than its size:*

$$C_{simple\ circuit} << S_{simple\ circuit}.$$

Usually the complexity of a simple circuit is constant: $C_{simple\ circuit} \in O(1)$.

◇

Definition 2.4 *A* **complex circuit** *is a circuit having the complexity in the same magnitude order with its size:*

$$C_{complex\ circuit} \sim S_{complex\ circuit}$$

.

◇

2.4.2 Complex computation vs. intense computation

Similar with the distinction between complexity and size of circuits, we define the complex computation and the intense computation.

Definition 2.5 *Complex computation is described by a program whose size, expressed in number of lines, is in the same magnitude order with the associated execution time, expressed in number of execution cycles.*

◇

Definition 2.6 *Intense computation is described by a program whose size, expressed in number of lines, is much smaller than the associated execution time, expressed in number of execution cycles.*

◇

2.5 Communication vs. computation

One of the main issue in computer science once the parallel, accelerated computation entered the scene is the relation between the computation time and the communication between the main memory and the computing device. There are two distinct possibilities:

1. the computation is I/O bounded: the accelerated computation is performed faster than the data transfer because it involves a big amoun of data to be transferred

2. the computation is not I/O bounded: the transfer time is much smaller than the accelerated computational time.

A typical example for the first case is the inner product of two n-component vectors performed by a SIMD machine. The accelerated computation is performed in $O(log\,n)$ time, while the transfer is performed in $O(n)$ time. For the second case, the matrix multiplication is a good example.

2.6 Host & Accelerator

Once the distinction between complex and intense computation becomes meaningful for the overall performance of a computing system, the dichotomy between the host engine, responsible for the complex computation, and the accelerator, performing the intense computation, will start to dominate the implementations of the high performance computing systems.

While the host engine remains a conventional computing system, the accelerator is implemented is a various forms. The main implementations takes *off-the-shelf* devices such as the oxymoronic GPGPU.

2.7 High-level language vs. library of functions

The programming environment for the host remains the conventional one: based on the most popular programming languages, such as C, C++, Python. For the accelerator there are two solutions:

1. to compile from the popular programming languages toward the unconventional structures of the accelerators

2. to see the accelerator as a hardware implemented library of functions.

2.8 Turing tariff

The term *Turing tariff* seems to be introduced by Paul Kelly [Edwards '21], professor at Imperial College. Roughly speaking, it is about the fact that a Turing-based computer can calculate any function, but not always efficiently.

The main Turing Tariffs are [Kelly '20]:

1. Fetch-execute is the original Turing tariff because of the *von Neumann bottleneck* between memory and processor which originates in the Turing's model with tape-head-automaton.

2. FPGAs pay Turing tariffs in the reconfigurable fabric use if we pass toward FPGA-implemented circuits highly intense computations.

3. Registers are a Turing Tariff because if we know the program's dataflow, we can use wires and latches to pass data from functional unit to functional unit

4. Memory, because if we can stream data from where it's produced to where it's processed, maybe we can reduce the use of RAM and the effect of the von Neumann bottleneck.
5. Cache, which is useless and area, energy and time consuming, because the intense computation is very predictable in term of data and program flow allowing us to use controlled buffers instead.
6. Floating-point arithmetic can be avoided if we know the dynamic range of expected values; thus we can avoid the wide range of values offered by the floating point representation.

The price for these tariffs are:

1. Fetch-execute, decode
2. Registers, forwarding
3. Dynamic instruction scheduling, cracking, packing, renaming
4. Cache tags
5. Cache blocks
6. Cache coherency
7. Prefetching
8. Branch prediction
9. Speculative execution
10. Address translation
11. Store-to-load forwarding, write combining, address decoding, ECC, DRAM refresh
12. Mis-provisioning: unused bandwidth, unusable FLOPs, under-used accelerators

They manifest in unuseful complexity of the computer organization. A lot of parasitic functions, unrelated with the main targets, where added to our computing machine to attenuate the effects of Turing tariffs.

The solutions currently proposed for reducing Turing tariffs are:

1. SIMD: amortise fetch-execute over a vector or matrix of operands
2. VLIW, EPIC (Explicitly parallel instruction computing), register rotation
3. Macro-instructions: FMA an extension to the 128 and 256-bit Streaming SIMD Extensions instructions in the x86 microprocessor instruction set to perform *fused multiply–add* operations), crypto, conflict-detect, custom ISAs
4. Streaming dataflow: FPGAs, CGRAs (Coarse Grain Reconfigurable Architectures)
5. Systolic arrays
6. Circuit switching instead of packet switching in communication networks
7. DMA
8. Predication
9. Long cache lines

10. Non-temporal loads/stores, explicit prefetch instructions
11. Scratchpads (small & fast local memories)
12. Multi-threading
13. Message passing

But, unfortunately, not all of these solutions paid back in a satisfactory manner.
We ca conclude with Paul H. J. Kelly:

1. Parallelism is (usually) easy – locality is hard
2. Don't spend your whole holiday carrying your skis uphill
3. Domain-specific compiler architecture is not about analysis! It is all about designing representations, and doing the right thing at the right level
4. When there's no more room at the bottom, all efficient computers will be domain-specific
5. Design of efficient algorithms will be about designing efficient domain-specific architectures
6. All compilers will have a place-and-route phase

Chapter 3

System-level organization

3.1 Defining reconfigurable computing

Reconfigurable computing develops in the context of the heterogenous accelerated computing paradigm and involves the use of reconfigurable devices, such as field programmable gate arrays (FPGAs), for computing purposes.

3.2 Taxonomy

3.2.1 Heterogenous accelerated computing

A reconfigurable system typically consists of one or more processors, one or more reconfigurable fabrics, and one or more memories. Reconfigurable systems are often classified according to the degree of coupling between the reconfigurable fabric, used as accelerator, and the host engine which is a PROCESSOR [Compton '02] [Todman '06] [Cardoso '10]. The framework used to develop reconfigurable computing is represented in Figure 3.1, where:

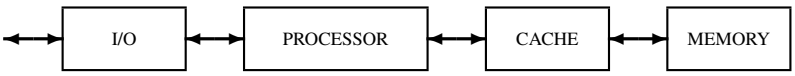

Figure 3.1: The framework used to define the reconfigurable computing by adding, in various configuration, an accelerator for the critical function supposed to be computed by HOST.

- PROCESSOR: is a mono/multi-core computing engine (for example, ARM Cortex-A9 on the Xilinx Zynq-7000)
- I/O: represents the set of input-output devices
- CACHE: is the Level 2 or Level 3 cache memory of the system
- MEMROY: is the main memory (usually a SDRAM) of the computing system.

There are various ways to attach an accelerator in this framework.

Stand-alone accelerator is a complex system designed to accelerate complex and intense functions. The computation time for a task sent through the I/O subsystem is enough big to compensate the communication overhead.

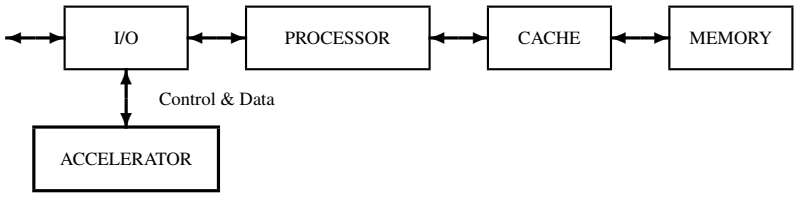

Figure 3.2: The reconfigurable accelerator is a stand-alone computing system *serial* connected with PROCESSOR.

Attached processing unit is an accelerator with a similar degree of complexity and size as the host engine. The communication cost with the rest of the system is lover.

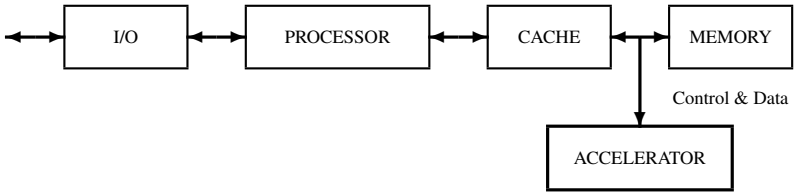

Figure 3.3: The reconfigurable accelerator is a processing unit *parallel* connected with PROCESSOR.

Co-processor which transfers data directly to/from the main memory because performs function defined on big data sequences.

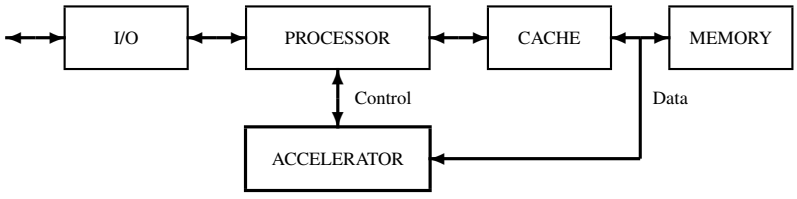

Figure 3.4: The reconfigurable accelerator is a *loop* connected co-processor.

Tightly coupled co-processor with data and control managed directly by the host engine.

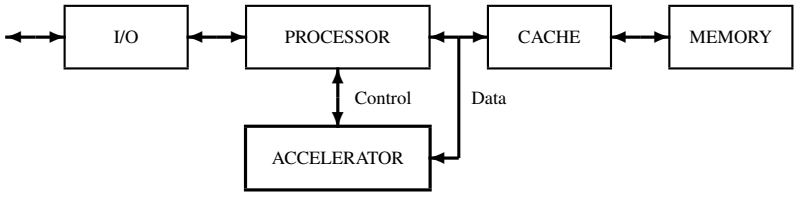

Figure 3.5: The reconfigurable accelerator is a *loop* connected co-processor

Flexible coupled accelerator which uses for big sequences the direct connection to the external memory, while for fast and small sized data transfer the internal memory resources (Cache and OCM (On Chip memory)) are accessed.

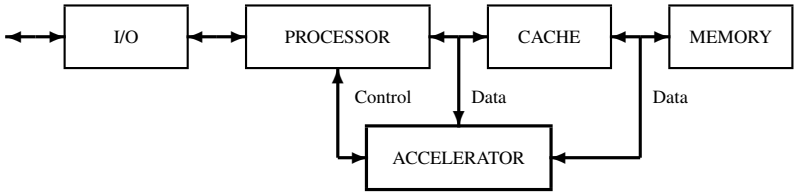

Figure 3.6:

Accelerator embedded in processor is a processor

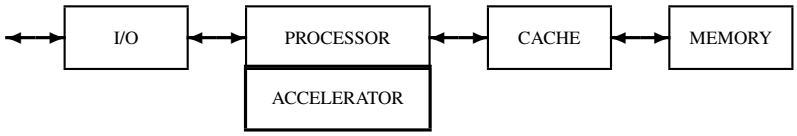

Figure 3.7:

Processor embedded in accelerator is defined in [Todman '06] as a "processor embedded in a reconfigurable fabric".

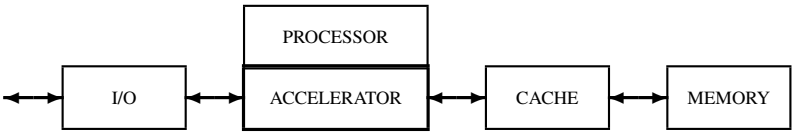

Figure 3.8:

3.2.2 Reconfigurable accelerators

Reconfigurable computing means a heterogenous system with a FPGA-based reconfigurable accelerator. There are two main types of reconfigurable accelerators:

- the reconfigurable version, when in FPGA is implemented a specific circuit for each function to be accelerated
- the pseudo-reconfigurable version, when in FPGS is implemented a parameterized and configurable programmable system

In the first case for each new function to be accelerated in a program running on PROCESSOR (see previous subsection) a new circuit must be loaded in FPGA. Unlike this solution, the second version requires a single load in FPGA for more than one accelerated function.

Chapter 4

Mathematical Models of Computation

4.1 Circuits

Uniform/nonuniform circuits Boolean circuits are one of the prime examples of so-called non-uniform models of computation in the sense that inputs of different lengths are processed by different circuits, in contrast with uniform models such as Turing machines where the same computational device is used for all possible input lengths. An individual computational problem is thus associated with a particular *family* of Boolean circuits $C_1, C_2, \ldots$ where each C_n is the circuit handling inputs of n bits.

Logic circuits are directed acyclic graphs (DAG) in which all vertices except input vertices carry the labels of gates. Input vertices carry the labels of Boolean variables, variables assuming values over the set $B = \{0,1\}$.

The circuit size, $S_{circ}(n)$ and depth, $D_{circ}(n)$ of most Boolean functions $f : B^n \to B$ on n variables are in $O(n^2$ and $o(n)$ respectively.

What means: a function is minimised? The characteristic Boolean vector is lossless compressible.

4.1.1 Combinational circuits

Theorem 4.1 *Any Boolean function $f : \{0,1\}^n \to \{0,1\}$ can be computed using a logic circuit.*

◇

Proof 4.1 *Let be the function $f(x_0, \ldots, x_{n-1})$, with $x_i \in \{0,1\}$ for $i = 0, \ldots, n-1$. We can write:*

$$f(x_0, \ldots, x_{n-1}) = x_0 g(x_1, \ldots, x_{n-1}) + x_0' h(x_1, \ldots, x_{n-1})$$

where:

$$g(x_1, \ldots, x_{n-1}) = f(1, x_1, \ldots, x_{n-1})$$

$$h(x_1, \ldots, x_{n-1}) = f(0, x_1, \ldots, x_{n-1})$$

The functions g and h are computed similarly using for each two $(n-2)$-input functions. An so on, until the $(n-n)$-input constant functions 0 and 1 are reached.

◇

Theorem 4.2 *Any Boolean function $f : \{0,1\}^n \to \{0,1\}$ is computed by a logic circuit with the size $\in O(2^n)$.*

◇

Proof 4.2 *The function:*

$$f(x_0,\ldots,x_{n-1}) = x_0 g(x_1,\ldots,x_{n-1}) + x_0' h(x_1,\ldots,x_{n-1})$$

is implemented by an elementary multiplexor, eMUX, and the functions g and fg. The functions g and h are implemented similarly, each using an eMUX and $(n-2)$-input functions. And so on until the last layer of $2^n/2$ eMUXs used to select between 1s and 0s. The resulting circuit is a binary tree of 2^n-1 eMUXs.

◇

The actual circuits, because the 1s and 0s on leafs of the tree of eMUXs generate the "collapse" of the eMUXs to connections to 0, 1, x_n, or x_n', are smaller.

Example 4.1 *Discrete Fourier Transform (DFT) implemented using Fast Fourier Transform* [Cooley '65] *has a $O(N \log N)$ sized circuit for N-sample input.*

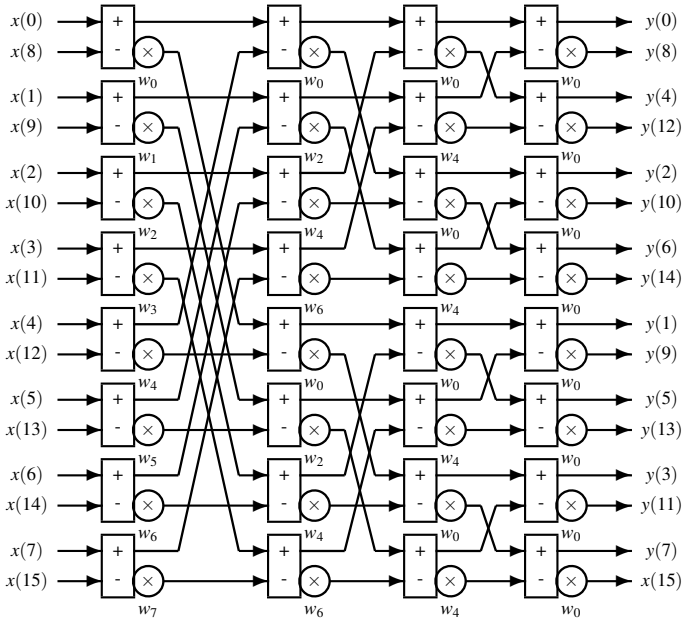

Figure 4.1: 16-input FFT circuit.

If the input is the time sequence $x_1, x_2, \ldots, x_N$ and the frequency sequence is $y_1, y_2, \ldots, y_N$, then:

$$y(i) = \sum_{n=0}^{N-1} x(n) \cdot e^{-j(\frac{2\pi}{N})nk}$$

while for Inverse Discrete Fourier Transform:

$$x(i) = \frac{1}{N} \sum_{n=0}^{N-1} y(n) \cdot e^{j(\frac{2\pi}{N})nk}$$

for $i = 0, 1, \ldots, N-1$. The execution time is in $O(N^2)$.

If we take a look at those FFT and IFFT equation it looks like very similar but two differences:

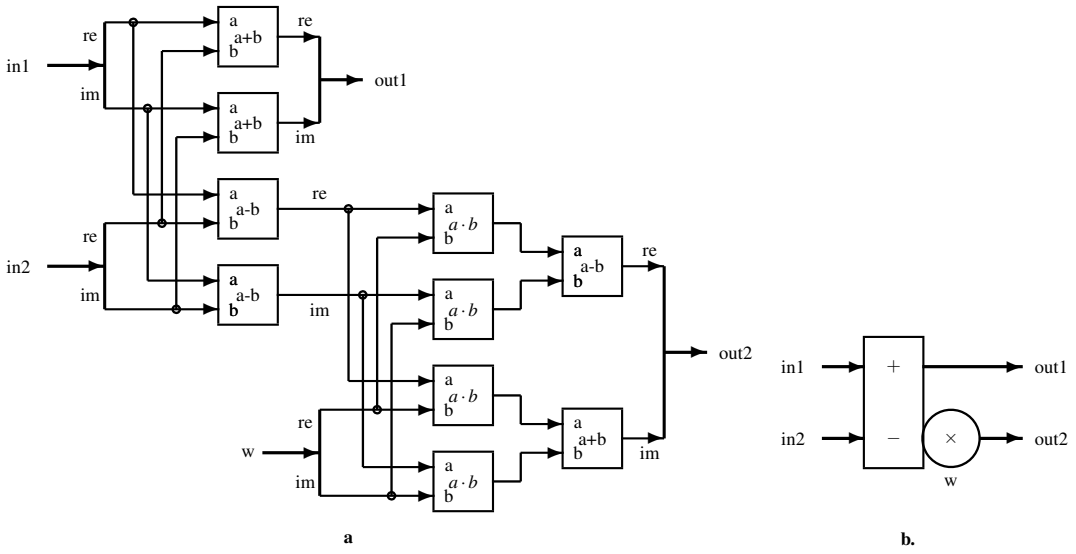

Figure 4.2: Combinational elementary cell for FFT: the Radix-2 Butterfly.

- Divided by N
- sign of power of exponential

Since the differences are small the FFT circuit can be used for IFFT by:

- swapping the imaginary part of each input with its real part (replace the twiddle factor with its complex conjugate) at the input of the FFT circuit
- swapping the imaginary part of each output with the real counterpart
- dividing by N the output (because usually N is a power of 2, the operation is performed in binary with a shift)

Because of the periodicity in the sequence of the values of the twiddle factors

$$e^{-j(\frac{2\pi}{N})nk}$$

instead of multiplying a vector with a matrix in time belonging to $O(N^2)$ the computation is accelerated using the circuit exemplified for $N = 16$ in Figure 4.1, where the elementary cell is the circuit represented in Figure 4.2, and the size is in $O(nlog\,n)$. The propagation time is in $O(log\,n)$.

The structure of the cell represented in Figure 4.2 results from the form of twiddle factors, which according to the famous Euler's formula:

$$e^{j\theta} = cos\theta + jsin\theta$$

◇

4.1.2 Pipelined circuits

In order to accelerate to computation performed by a deep circuit pipeline registers are used. The main price for this improvement is one cycle latency introduced by each pipeline level.

Example 4.2 *The example of the FFT circuit can be revisited using the pipelined cells represented in Figure 4.3.*

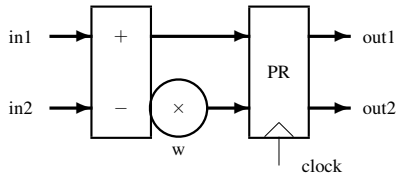

Figure 4.3: Sequential elementary cell for FFT, eSFFT.

The pipeline register, PR, associated with each cell introduces a log_2N cycles latency but increase the clock frequency of the system almost log_2N times.

◇

4.1.3 Iterative circuits

For a small solution we have another approach: an iterative circuit. For a deep multi-layer circuit on each layer with identical circuits connected differently we can use only one layer and a loop which connects appropriately the output back to the inputs.

Example 4.3 *Revisiting once again the FFT example, let's consider the circuit represented in Figure 4.4.*

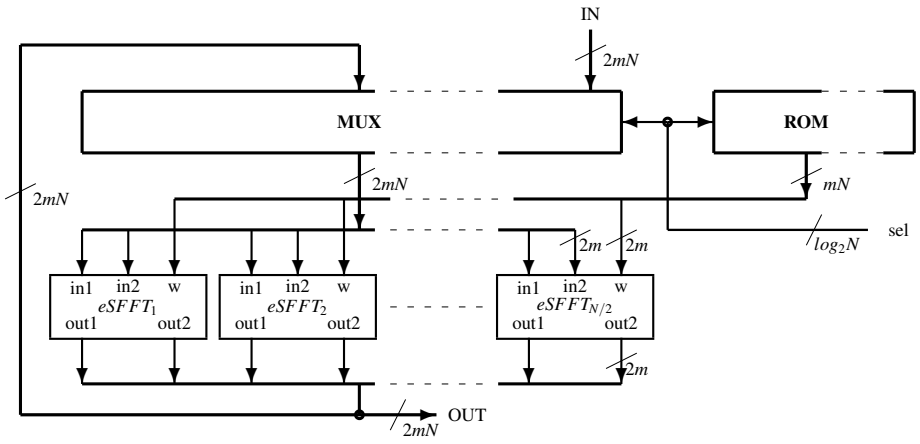

Figure 4.4: The iterative version of the FFT circuit.

For N inputs, the number of the cells eSFFT is $N/2$. Instead of another $-1+log_2N$ layers each containing $N/2$ cell, the circuit has the multiplexer MUX. Because each input value is a complex number with the real part and the imaginary part each represented on m bits, the input IN has $2mN$ bits. The same for the output OUT. The output of ROM is of mN bits because in each stage of the computation we need only $N/2$ complex numbers.

◇

4.1.4 Controlled circuits

The FFT computation, used as example until now, is a computation which end in a predictable time. It halts independent of the way the computation evolves. A lot of computations fall in this category. But not all. There are computational processes whose evolution is guided takeing into account a condition tested during the process. Therefore, we must consider circuits evolving under the control of an additional loop.

To the *big & simple* circuit used to perform an intense function is added a *small & complex* – usually a finite automaton (FA) or a counter extended FA (CFA) – circuit to coordinate the elements of the big and simple circuit in order to optimize its use.

Anothe characteristics of a controlled circuit used in accelerating the computation is the iterative aspect, i.e., the computation stops when a condition is fulfilled.

Example 4.4 *Clustering is an intense computation described by a WHILE loop. Therefore, it supposes a circuit, because of intensity and a control because the WHILE loop end when the result fulfills a certain condition. The algorithm is the following:*

k-MEANS CLUSTERING ALGORITHM
$X = [x_1,\ldots,x_n]$: vector of the coordinates x of each point
$Y = [y_1,\ldots,y_n]$: vector of the coordinates y of each point
$K = [k_1,\ldots,k_n]$: vector of cluster's name associated to each point, initialized with 0 in each position
$x = [x^1,\ldots,x^k]$: vector of the coordinates x of each center, initialized randomly
$y = [y^1,\ldots,y^k]$ vector of the coordinates y of each center, initialized randomly

WHILE $(K! == K')$ // test the end of process
    $K \Leftarrow K'$

    FOR $(i = 1; i \leq k; i = i+1)$
        $D' \Leftarrow (X - x^i)^2$
        $D' \Leftarrow D' + (Y - y^i)^2$

        WHERE $(D' < D)$
            $D \Leftarrow D'$
            $K' \Leftarrow i$

    FOR $(i = 1; i \leq k; i = i+1)$

        WHERE $(K' = i)$
            $x^i \Leftarrow redAdd(X)/redAdd(B)$
            $y^i \Leftarrow redAdd(Y)/redAdd(B)$

The circuit represented in Figure 4.5 computes the k-Means Clustering of a N points distributed in a two-dimension space. The memory RAM is loaded in N steps with a pair of m-bit coordinates $\{x_i, y_i\}$ for $i = 1,\ldots,N$, in each location, while the RAMK memory is loaded randomly with (log_2k)-bit numbers representing the initial, random, distribution of the centers allocated to each point. The $2m$-bit registers, R_i are initialized randomly with the pairs of coordinates $\{x^i, y^i\}$ for $i = 1,\ldots,k$, for the clustering centers. The k modules D compute in each clock cycle, c_i for $i = 1,\ldots N$, the square of the distance between the

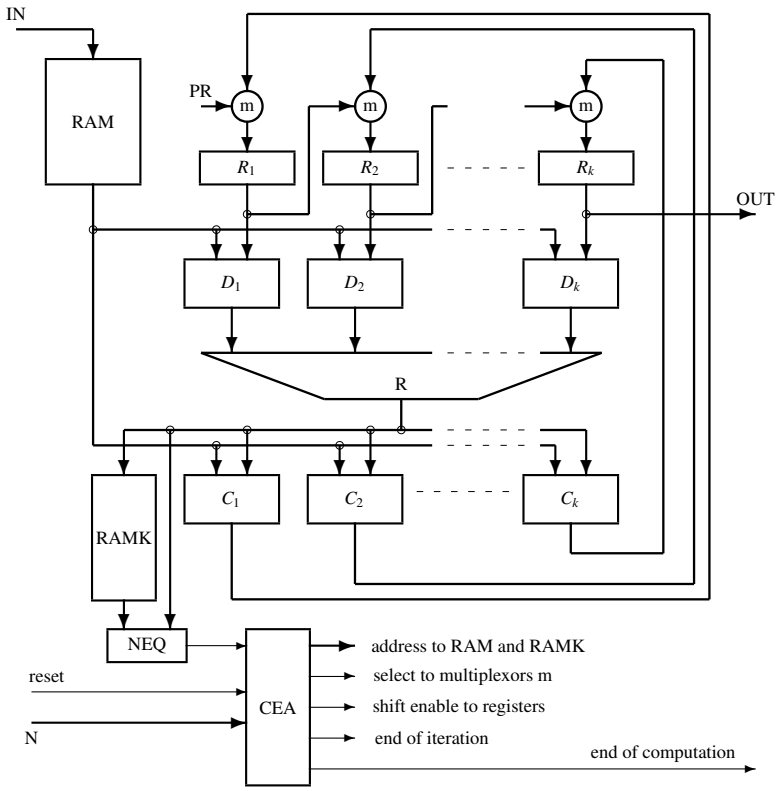

Figure 4.5:

point p_i and each center c^j for $j = 1,\ldots,k$:

$$d_i^j = (x_i - x^j)^2 + (y_i - y^j)^2$$

The reduction circuit R provides the index j of the smallest squared distance. The index j is compared with the index stored at the location i in RAMK and then is loaded at the location i. If the index is equal with the value store at i in RAMK, then the output of NEQ circuit is unchanged, else it becomes 1. In each cycle only one circuit C_j is activated by accumulating in two registers the current coordinates x_i and y_i and by incrementing counter initialized, at each iteration, on zero. Each iteration ends in all C_j for $j = 1,\ldots,k$ by computing to their outputs the quotient of the divisions between the sum of coordinated associated to the cluster j and the number accumulated in the local counter. These two numbers are then loaded in R_j. If the output of NEQ is 1, then another iteration is computed, else the registers R_j are shifted out as result.

Because in real applications $k << N$ we decided to design a circuit with the size of the computational resources in $O(k)$, and the execution time in $O(N)$.

$\diamond$

4.2 Turing/Post Model

Definition 4.1 *Turing Machine (TM) has three main components (see Figure 4.6):*

- *a* **finite automaton** *(FA): a small but complex structure*
- *an "infinite"* **tape** *with locations from where in each cycle a symbol is read and substituted: a big but simple structure*
- *an access* **head** *which accesses in each cycle a cell from the tape and can be moved one position to the left or to the right: a big but simple structure*

and is defined by:

$$TM = (S, A, \Sigma, \#, s_0, f)$$

where:

S**:** *the finite set of states of the FA*

A**:** *the finite alphabet of symbols used to generate the content in the locations of a tape with an "infinite" number of locations*

$\Sigma \subset S$**:** *the subset of finite final states of the FA*

$\# \in A$**:** *the delimiter symbol used in the left end and right end of the string of symbols stored on the tape*

$s_0 \in S$**:** *the initial state of FA*

$f : (S \setminus \Sigma) \times A \to S \times A \times \{nop, left, right\}$**:** *is the transition function of TM which in each cycle according to the state of FA and the symbol accessed on the tape generate the next state of FA, write a symbol on the tape of TM and sends a command to the access head to move one position to left or right or to stay in the same position.*

$\diamond$

The use of the term "infinite" means: the needed size of the band is not known, in the general case, at the beginning of the computation.

Initially, FA of TM is in state s_0, having on the tape a finite string of symbols $\ldots\#, x_1, x_2, \ldots, x_n\#\ldots$, with $x_i \in A$, for $i = 1, \ldots, n$, and the had pointing to the left #. Only if FA is defined for a ***computable function***, then, after a finite number of cycles, FA "halts" in a final state $s_f \in \Sigma$, with the head pointing y_1 form the tape content of form $\ldots, \#, y_1, y_2, \ldots, y_m, \#, \ldots$ with $y_i \in A$, for $i = 1, \ldots, m$.

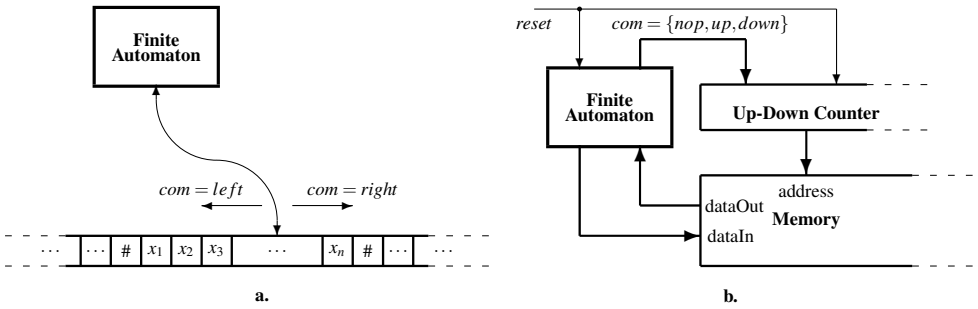

Figure 4.6: **Turing Machine**. **a.** The original representation. **b.** The current representation with the head implemented as an "infinite" up/down counter and the tape as an "infinite" random-access memory.

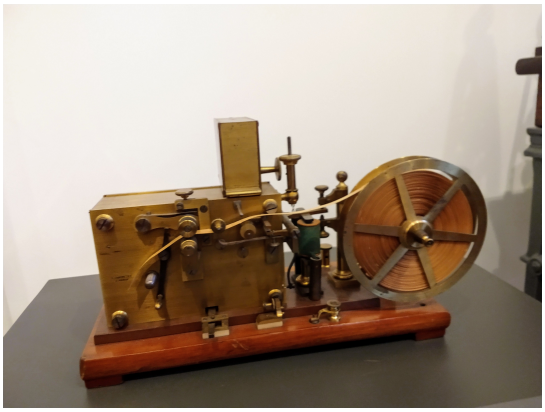

Figure 4.7: The electro-mechanic telegraph with a paper tape which inspired Turing to define the tape of its machine.

Definition 4.2 Universal TM *(UTM) is a TM with the band divided in two sections as follows:*

$$\ldots, \#, p_1, p_2, \ldots, p_n, \$, d_1, d_2, \ldots, d_m, \#, \ldots$$

where:

- $e(M) = \#, p_1, p_2, \ldots, p_n, \$$ *is a description of the behavior of the automaton* FA_k *of any TM* T*, i.e., it is the* ***program*** *describing the function* f
- $T = \$, d_1, d_2, \ldots, d_m, \#$ *is the content,* T*, of the band associated to* T*, i.e., is* ***data*** *used by the program.*

while FA of UTM is able to apply the description of $e(M)$ *on* T*.*

◇

UTM is the mathematical model of a computer formed by an engine (FA and head) and a memory as large as necessary ("infinite" band).

4.2.1 The Halting Problem

A function $F(x)$ is called computable if it can be computed by a TM, i.e., its **finite** description as $e(M)$ of T stops working on the tape containing T.

Definition 4.3 *Given a TM $M(T)$ working on a tape content T, the* **Halting Problem** *(HP) is to find a MT $H(\langle e(M),T\rangle)$ so as:*

$$H(\langle e(M),T\rangle) = 1 \textbf{ if } M(T) \textit{ halts}$$

$$H(\langle e(M),T\rangle) = 0 \textbf{ if } M(T) \textit{ runs forever}$$

where: $e(M)$ is the symbolic description of the M's FA.

$\diamond$

In his seminal paper [Turing '36] Alan Turing proves that H doesn't exists. In the following we will use a proof provided by John Casti [Casti '92].

Theorem 4.3 *The function $H(\langle e(M),T\rangle)$ is uncomputable for any $M(T)$.* $\diamond$

Proof Assume that the TM H exists for *any* encoded machine description and for *any* input tape. We will define an *effective* TM G such that for any TM F, G halts with the tape content $e(F)$ if $H(\langle e(F),e(F)\rangle) = 0$ and runs forever if $H(\langle e(F),e(F)\rangle) = 1$. G is an effective machine because it involves the function H and we assumed that this function is computable.

Now consider the computation $H(\langle e(G),e(G)\rangle)$ (G halts or not, running on its own description).

If $H(\langle e(G),e(G)\rangle) = 1$, **then** the computation of $G(e(G))$ *halts*, **but** starting from the G's definition $G(e(G))$ the computation halts only **if** $H(\langle e(G),e(G)\rangle) = 0$. Therefore, **if** $H(\langle e(G),e(G)\rangle) = 1$, **then** $H(\langle e(G),e(G)\rangle) \neq 1$.

If $H(\langle e(G),e(G)\rangle) = 0$, **then** the computation of $G(e(G))$ *runs forever*, **but** starting from the G's definition $G(e(G))$ the computation runs forever only **if** $H(\langle e(G),e(G)\rangle) = 1$. Therefore, **if** $H(\langle e(G),e(G)\rangle) = 0$, **then** $H(\langle e(G),e(G)\rangle) \neq 0$.

The application of function H to the machine G and its description generates a contradiction. Because H is defined to work for any machine description and for any input tape, we must *conclude* that the initial assumption is not correct and H is not computable.

$\diamond$

The price for structural simplicity is the limited domain of the computable. See also the minimalization rule in Kleene's model.

Let us remember the Theorem 2.1 that proves that circuits compute *all* the functions. UTM is limited because it does not compute at least HP. But the advantage of UTM is that the computation has a finite description instead of the circuits that are huge and complex. Circuits are *complex* while the algorithms for TMs are *simple*. *But, the price for the simplicity is the incompleteness*.

4.3 Church Model

Definition 4.4 *A λ-expression is defined as a stream of symbols belonging to a finite set V to which three special symbols – λ, (,) – adds. It form obeys to the following four rules:*

1. *$x \in V$ is a λ-expression*
2. *if M is a λ-expression and x a variable, then λxM is a λ-expression, where: λx is the* bound variable part*, while M is the* body
3. *if both F and A are λ-expressions, then (FA) is λ-expression, where: F is the* operator *part and A is the* operand *part*

4. *any expression constructed using the above three rules is a λ-expression.*

◇

Definition 4.5 Reduction Rule*: If we have a λ-expression (λxMA), with the operator λxM and the operand A, then it will be replaced with the body of the operator, M, in which all the free occurrences of x will be substituted with the operand A.*

◇

Example 4.5 *Let us take the following simple example of λ-expression:*

$$(\lambda f((f3)+(f4))\lambda x(x\times x))$$

where: the operator *is $\lambda f((f3)+(f4))$ with the* variable *f and* body *$((f3)+(f4))$, and the* operand *is $\lambda x(x\times x)$. The reduction follows in tree steps:*

$$(\lambda f((f3)+(f4))\lambda x(x\times x)) \rightarrow ((\lambda x(x\times x)3)+(\lambda x(x\times x)4) \rightarrow ((3\times 3)+(4\times 4)) \rightarrow 25$$

◇

4.3.1 The Halting Problem

Example 4.6 *Let us take the following λ-expression:*

$$(\lambda x.xx)(\lambda x.xx)$$

which means to use the operator as operand, i.e., to apply the function to its description. By reduction we obtain:

$$(\lambda x.xx)(\lambda x.xx) \rightarrow (\lambda x.xx)(\lambda x.xx) \rightarrow \ldots \rightarrow (\lambda x.xx)(\lambda x.xx) \rightarrow \ldots$$

as an unending process.

◇

Again, the self-reference leads to uncomputability.

4.4 Kleene Model

Definition 4.6 *Let be the positive integers $x,y,i \in \mathbf{N}$ and the sequence $X = \langle x_0, x_1, \ldots, x_{n-1}\rangle \in \mathbf{N}^n$. Any partial recursive function $f : \mathbf{N}^n \rightarrow \mathbf{N}$ can be computed, according to [Kleene '36], using three **initial functions**:*

- *$ZERO(x) = 0$: the variable x takes the value* **zero**
- *$INC(x) = x+1$:* **increment***s the variable $x \in \mathbf{N}$*
- *$SEL(i,X) = x_i$: i* **select***s the value of x_i from the sequence of positive integers X*

*and the application of the following three **rules**:*

- **Composition:** *$f(X) = g(h_1(X), \ldots, h_p(X))$, where: $f : \mathbf{N}^n \rightarrow \mathbf{N}$ is a total function if $g : \mathbf{N}^p \rightarrow \mathbf{N}$ and $h_i : \mathbf{N}^n \rightarrow \mathbf{N}$, for $i = 1, \ldots p$, are total functions*
- **Primitive recursion:** *$f(X,y) = g(X, f(X,(y-1)))$, with $f(X,0) = h(X)$ where:*
  $f : \mathbf{N}^{n+1} \rightarrow \mathbf{N}$ is a total function if $g : \mathbf{N}^{n+1} \rightarrow \mathbf{N}$ and $h : \mathbf{N}^n \rightarrow \mathbf{N}$ are total functions.

- **Minimization:** $f(x) = \mu y[g(x,y) = 0]$, *which means: the value of the function* $f : \mathbf{N} \to \mathbf{N}$ *is the smallest* y, ***if any***, *for which the function* $g : \mathbf{N}^2 \to \mathbf{N}$ *takes the value* $g(x,y) = 0$.

◇

Example 4.7 *Given the function* $f(x) = x/3$ *it is computed by searching for the value of* y *for which* $g(x) = 3y - x = 0$, *i.e.:*

$$f(x) = \mu y[g(x,y) = 0]$$

If $x = 9$, *then in in 4 steps starting form* $y = 0$ *we reach the result* $f(9) = 3$. *But if* $x = 10$ *we enter in a never-ending loop.*

◇

4.4.1 The Halting Problem

The non-computability surfaces in the Kleene's approach occasioned by the minimization applied to a function for which the smallest y for which $g(x,y) = 0$ is never reached. The first to rules act building the solution, while the third rule, minimisation, searches for the solution, and when you search you must be prepared to find forever, that is, to find nothing.

4.4.2 The Circuit Implementation of Partial Recursive Functions

For recursive functions there are very promising circuit implementations (for details see Appendix A).

The circuit for the composition rule

For the composition rule there is a direct circuit implementation as a cellular serial-parallel expansion. In Figure A.1, a two level system is associated with composition:

- map level: performs in a synchronic parallel manner the functions $h_1(X)$
- reduction level: performs in a diachronic parallel manner related to the map level the function $g(h_1(X), \ldots, h_p(X))$

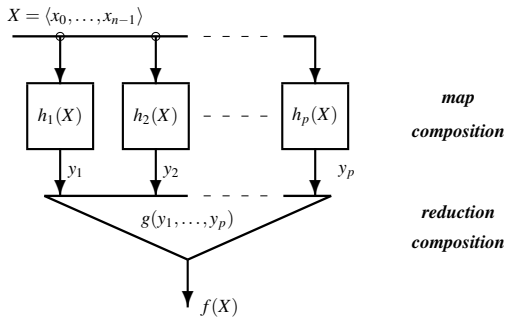

Figure 4.8: **The circuit version of composition.** It is a two-layer construct: the parallel expanded *map* layer serially connected with the *log*-dept *reduction* layer.

The circuit for the primitive recursion rule

In Figure 4.9a, the primitive recursion mechanism is implemented. In the MOP (Multiple Output Pipeline) section are computed iteratively the functions $f(X,y)$ for $y = 0,1,\ldots,i,\ldots$, each cell using the result generated by the previous cell (except for the first cell).

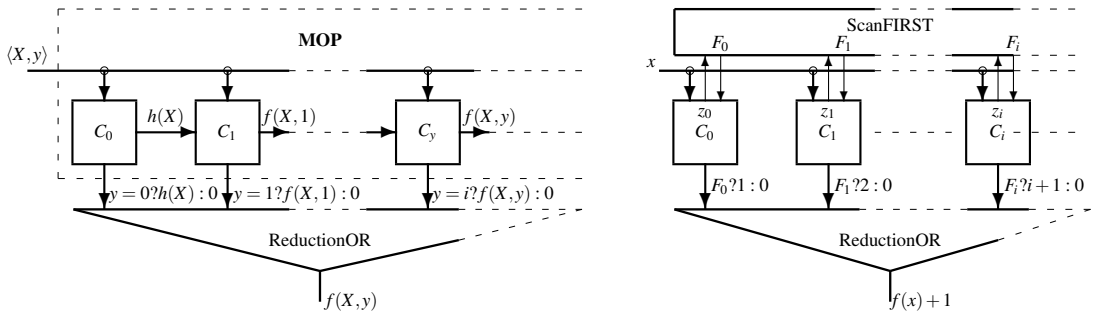

Figure 4.9: **Primitive recursion and minimization are multiple applications of specific compositions. a.** The circuit version of primitive recursive rule. **b.** The circuit version of minimization rule.

The reduction level computes the reduction OR function from the outputs provided by each cell. Only one cell will generate a value of the function, all the others will send to the reduction net zero.

The circuit for the minimisation rule

In Figure 4.9b, the minimisation mechanism is implemented. The circuit scanFIRST receives a Boolean vector and sends back another Boolean vector in which only the first occurrence of 1 is maintained. The output z of each cell takes the value of the predicate $y[g(x,y) == 0$. Only the unique cell for which $F_i = 1$, if any, sends to the reductionOR net the value $i+1$ representing $f(x)+1$.

Important note: the representations form Figure 4.9 are associated to a mathematical model. For real circuits the structures are limited to a number of cells. Additional loops allow the expansion of the functionality quantitatively.

Chapter 5

Designing Reconfigurable Systems

Reconfigurable computing is a computer paradigm combining the flexibility of software with the high performance of hardware implemented with the very flexible, high speed solution offered by the field-programmable gate arrays (FPGAs). The principal advantage when compared to using ordinary processor-based mechanism is the ability to make changes to the data-path itself in addition to the control flow. The main difference from application-specific integrated circuits (ASICs) is the possibility to adapt the hardware used to accelerate computationally intense functions during runtime by instantiating totally or partially a new circuit on the reconfigurable fabric.

5.1 High Level Synthesis

High-level synthesis, HLS, is a new step in abstraction enabling the designer to focus on larger structural questions rather than registers and cycle-level operations. Instead a designer describes the behavior of the circuit in a program, and the HLS tool creates the detailed RTL code. Currently in use tools start with C/C++ as the input language.

Fundamentally, a HLS tool does a lot of things automatically that a HDL designer does manually:

- analyzes and uses the concurrency in the algorithm described in C/C++
- includes pipeline registers as necessary to achieve a maximum (desired) clock frequency
- generates control logic that directs the data path.
- provides the design for interfaces to connect to the rest of the system.
- distributes data in registers or BRAMs to optimize resource usage and bandwidth.
- distributes computation effort to combinational logic elements to achieve ant efficient implementation.

The Vivado HLS design environment requires the following inputs:

- the function specified in C, C++, or SystemC
- the design testbench that calls the function and verifies its correctness by checking the results.
- the target FPGA device
- the desired clock period

- directives guiding the implementation process

We make the following assumptions about the input function specification, which generally adheres to the guidelines of the Vivado HLS tool:

- No dynamic memory allocation (no operators like malloc(), free(), new, and delete())
- Limited use of pointers-to-pointers (e.g., may not appear at the interface)
- System calls are not supported (e.g., abort(), exit(), printf(), etc. They can be used in the code, e.g., in the testbench, but they are ignored (removed) during synthesis.
- Limited use of other standard libraries (e.g., common math.h functions are supported, but uncommon ones are not)
- Limited use of function pointers and virtual functions in C++ classes (function calls must be compile-time determined by the compiler).
- No recursive function calls.
- The interface must be precisely defined.

Vivado HLS generates the following outputs:

- Synthesizable Verilog and VHDL
- RTL simulations based on the design testbench
- Static analysis of performance and resource usage
- Metadata at the boundaries of a design, making it easier to integrate into a system.

5.1.1 Organization

The first important decision about the RTL code is the organization of the circuit which is a compromise between the physical resources (the size of the circuit) and performance (expressed in clock frequency and latency).

Example 5.1 *Let's consider [Kastner '20] a simple yet common hardware function: the finite impulse response (FIR) filter. An FIR performs a convolution on an input sequence with a fixed set of coefficients.*

The file `fir8.c` describe a 8-tap FIR. It can be used as input for Vivado HLS.

```
/*****************************************************************
File name:  fir8.c
Circuit name:
Description: Code for a 8-tap FIR filter.
*****************************************************************/
    #define NUM_TAPS 8
    void fir(int input, int *output, int taps[NUM_TAPS]) {
        static int delay_line[NUM_TAPS] = {};
        int result = 0;
        for(int i = NUM_TAPS - 1; i > 0; i--) {
        delay_line[i] = delay_line[i - 1];
        }
```

```
    delay line[0] = input;
    for (int i = 0; i < NUM_TAPS; i++) {
        result += delay line[i] * taps[i];
    }
    *output = result;
}
```

Instead of outputting assembly code for a standard processor (such as Intel or ARM) , the HLS compiler geneate an RTL hardware description.

One possible circuit will execute the function sequentially like a mono-core processor. It is represented in Figure 5.1. The Read-Only Memory, ROM, contains the tap constants. The block SEQ generate de sequence of commands for selecting, with the output i, at the multiplier's input the appropriate register D_i and the corresponding value tap_i from ROM. At each sampling cycle:

- the output register R_{out} is enabled
- the accumulator register R_{acc} is cleared
- the input register R_{in} and the delay registers D_I, for $i = 1,\ldots,7$, are enabled.

The only register which switches in each clock cycle is R_{acc}. The signal *sync* generated by SEQ is used as output signal *get* to synchronize the filter with the system which includes it, as *reset* for the accumulator register R_{acc}, and as *enable* for all the other registers. Thus, for each sample of the input signal, the system uses 8 clock cycles to compute the output sample of the filtered signal. The clock frequency is:

$$f_{clock} = \frac{1}{max((t_{SEQ} + t_{ROM}), t_{prop\_reg}) + t_{MULT} + t_{ADDER} + t_{reg\_set\_up}}$$

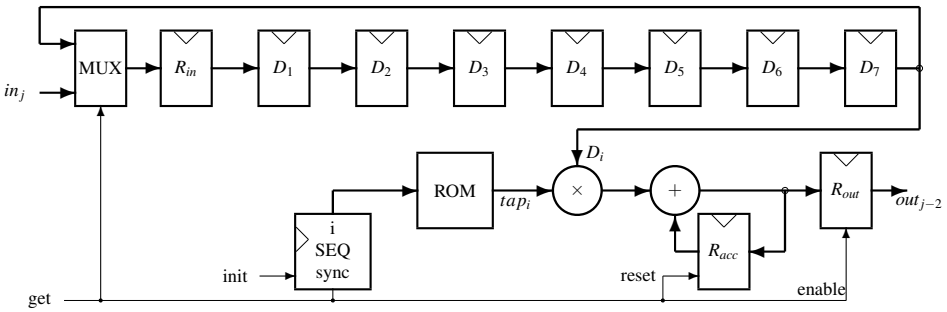

Figure 5.1:

The sampling frequency is $f_{sampling} = 1/f_{clock}$, i.e., the index i runs $8\times$ faster than the index j. The resulting performance is not far from the performance provided by a programmed solution.

Another possible solution is presented in Figure 5.2 where the structure is simpler but larger. The speed is similar. The system works at the sampling frequency computed as follows:

$$f_{clock} = f_{sampling} = \frac{1}{t_{prop\_reg} + t_{MULT} + 7 \times t_{ADDER} + t_{reg\_set\_up}}$$

A certain increase in frequency is obtained compared to the previous situation, because the multiplexer and the ROM are eliminated, thus the access of the operands to the inputs of the multiplier is accelerated.

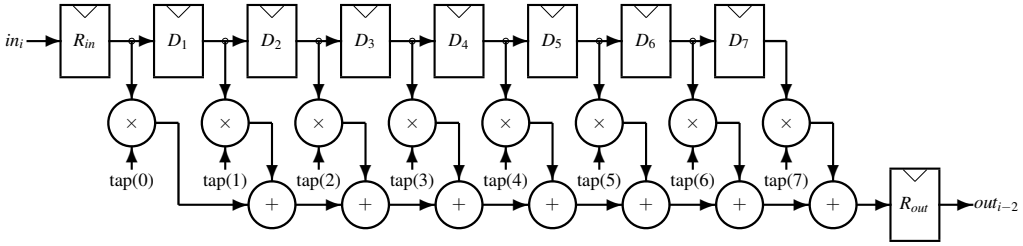

Figure 5.2:

In practice, designs must include specific tradeoffs between a sequential architecture (like the one shown in the Figure 5.1) and parallel architectures (like the solutions shown in the Figures 5.2, 5.3, 5.4), in order to achieve the best performance. In Vivado HLS, these improvements are mainly controlled by the user, using tool options and code annotations, such as `#pragma` directive.

How can be "determined" the compiler to generate RTL code for the structure presented in Figure 5.2? By adding additional directives such as:

`#pragma HLS unroll` : to unroll the first `for` by generating the behavior of a hardware delay line

`#pragma HLS pipeline` : to unroll the second `for` by generating a pipeline execution of the addition

in the previous code is provided the following code:

```
/*************************************************************************
File name:
Circuit name:
Description:
*************************************************************************/
    #include "block_fir.h"
    void block_fir(int input[256], int output[256], int taps[NUM_TAPS],
    int delay_line[NUM_TAPS]) {
        int i, j;
        for (j = 0; j < 256; j++) {
            int result = 0;
            for(int i = NUM_TAPS - 1; i > 0; i--) {
                #pragma HLS unroll
                delay_line[i] = delay_line[i - 1];
            }
            delay_line[0] = input[j];
            for (i = 0; i < NUM_TAPS; i++) {
                #pragma HLS pipeline
                result += delay_line[i] * taps[i];
            }
            output[j] = result;
        }
    }
```

◇

5.1.2 Processing Rate

There are two main limitations in accelerating the processing rate, i.e., the clock frequency used to trigger the circuit inferred by the compileer from the high-level language code. The most important limit arises from:

- resource limitations
- recurrences or feedback loops in a design.

Managing resource limitations

Example 5.2 *Let's revisit the previous example. A third solution for the FIR filter comes only with a rearrangement of the circuits.*

In Figure 5.3, we use the same physical resources so interconnected that the frequency of the clock can be increased to the following value:

$$f_{clock} = f_{sampling} = \frac{1}{t_{prop\_reg} + t_{MULT} + 3 \times t_{ADDER} + t_{reg\_set\_up}}$$

This solution comes directly by applying Spira's theorem [**?**], according to which a degenerate tree with depth n can be transformed into a perfectly balanced tree of depth $log_2 n$.

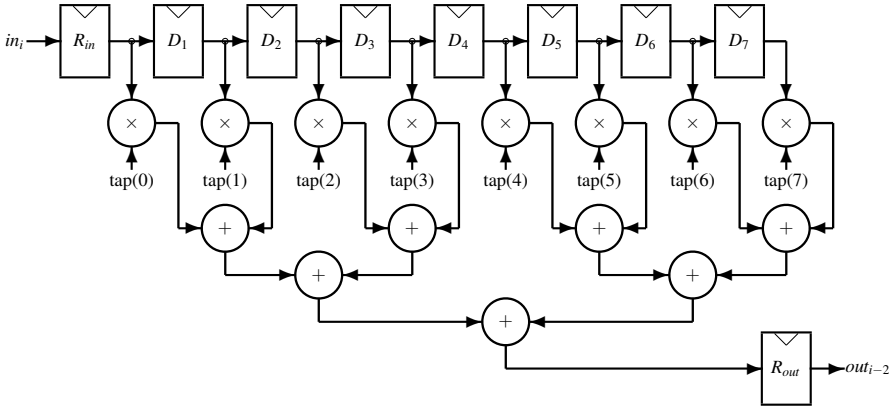

Figure 5.3:

If we accept additinal 3 clock cycle latency the fourth solution (see Figure 5.4) provide a faster version of the FIR filter. For this version the clock frequency is:

$$f_{clock} = f_{sampling} = \frac{1}{t_{prop\_reg} + t_{MULT} + t_{reg\_set\_up}}$$

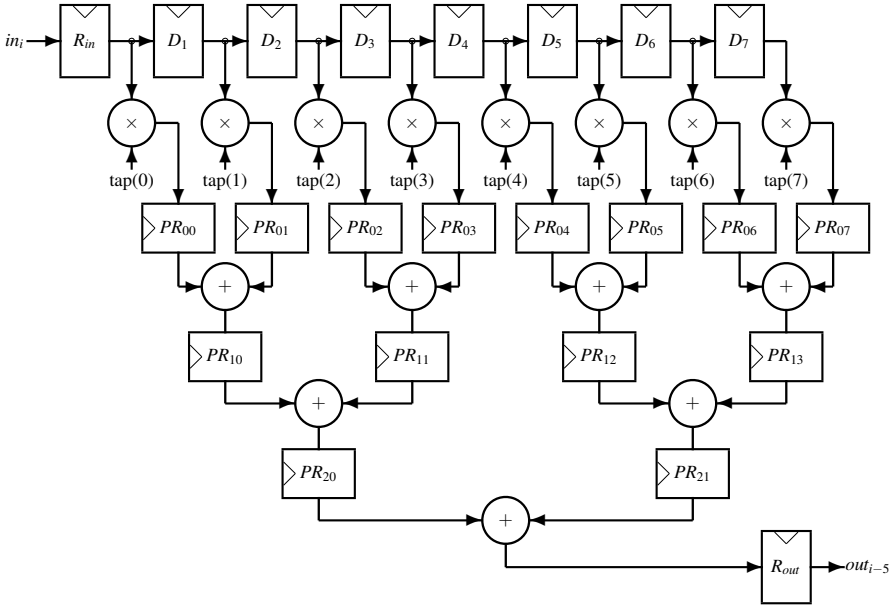

Figure 5.4:

If we need a faster version, a pipeline level can be introduced in the structure of the multiplier. The multiplier has two levels: first level is implemented with carry-save adders and the second is a ripple-carry adder. Between these two levels a pipeline-register will allow to almost double the frequency of the clock signal. Then the clock frequency of the system becomes:

$$f_{clock} = f_{sampling} = \frac{1}{t_{prop\_reg} + t_{ADDER} + t_{reg\_set\_up}}$$

This last improvement comes with an additional clock cycle latency. Therefore the output is out_{i-6}.

◇

Managing recurrences

The good news about loop is its their "ability" to add new features. But any good news is accompanied by its own bad news. In this case is about the limiting of the degree of parallelism allowed in a system with a just added loop. It is mainly about the necessity to **stop** sometimes the input stream of data in order to decide, inspecting an output, how to continue the computation. The input data waits for data arriving from an output a number of clock cycles related with the system latency. To do something special the system must be allowed to accomplish certain internal processes.

Both, data parallelism and time parallelism are possible because when the data arrive the system "knows" what to do with them. But sometimes the function to be applied on certain input data is decided by processing previously received data. If the decision process is to complex, then new data can not be processed even if the circuits to do it are there.

Example 5.3 *Let be the system performing the following function:*

```
procedure cond_acc(a,b, cond);
    out = 0;
    end = 0;
    loop    if (cond = 1)   out = out + (a + b);
                else        out = out + (a - b);
    until   (end = 1) // the loop is unending
endprocedure
```

For each pair of input data the function is decided according to a condition input. The Verilog code describing an associated circuit is:

```
/*****************************************************************
File name:      cond_acc0.v
Circuit name:   Conditioned Accumulator
Description:    the behavioral description of the conditioned accumulator
*****************************************************************/
module cond_acc0(  output  reg [15:0]  out,
                   input       [15:0]  a, b,
                   input               cond, reset, clock);
    always @(posedge clock) if (reset)      out <= 0;
                            else if (cond)  out <= out + (a + b);
                                 else       out <= out + (a - b);
endmodule
```

In order to increase the speed of the circuit a pipeline register is added with the penalty of $\lambda = 1$. *Results:*

```
/*****************************************************************
File name:      cond_acc1.v
Circuit name:   Pipelined Conditioned Accumulator
Description:    the behavioral description for the circuit
*****************************************************************/
module cond_acc1(  output  reg [15:0]  out,
                    input       [15:0]  a, b,
                    input               cond, reset, clock);
    reg[15:0]  pipe;
    always @(posedge clock)
    if (reset)  begin   out <= 0;
                        pipe <= 0;
                end
    else begin  if (cond)   pipe <= a + b;
                 else       pipe <= a - b;
                out <= out + pipe;
           end
endmodule
```

Now let us close a loop in the first version of the system (without pipeline register). The condition input takes the value of the sign of the output. The loop is: `cond = out[15]`. *The function performed*

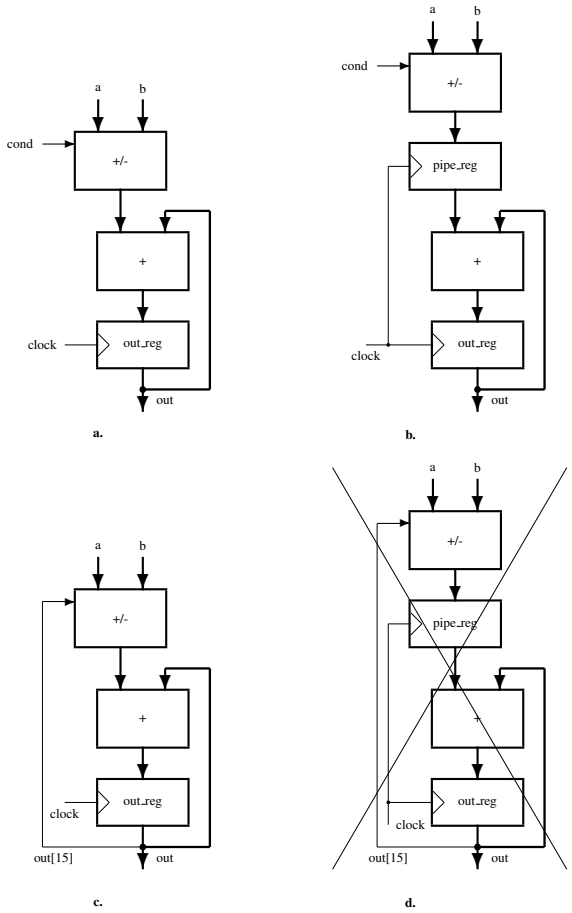

Figure 5.5: **Data dependency when a loop is closed in a pipelined structure.** **a.** The non-pipelined version. **b.** The pipelined version. **c.** Adding a loop to the non-pipelined version. **d.** To the pipelined version the loop can not be added without supplementary precautions because *data dependency* change the overall behavior. The selection between add and sub, performed by the looped signal comes too late.

on each pair of input data in each clock cycle is determined by the sign of the output resulted from the computation performed with the previously received pairs of data. The resulting system is called `addapt_acc`.

```
/*************************************************************************
File name:      adapt_acc0.v
Circuit name:   Adaptive Accumulator
Description:    the structural description of the adaptive accumuulator
*************************************************************************/
 module adapt_acc0(output  [15:0]  out,
                   input   [15:0]  a, b,
                   input           reset, clock);
 cond_acc0  cont_acc0(  .out     (out),
                        .a       (a),
```

```
                    .b      (b),
                    .cond   (out[15]),     // the loop
                    .reset  (reset),
                    .clock  (clock));
endmodule
```

Figure 5.5a represents the first implementation of the `cond_acc` *circuit, characterized by a low clock frequency because both the* adder *and the* adder/subtracter *contribute to limiting the clock frequency:*

$$f_{clock} = \frac{1}{t_{+/-} + t_{+} + t_{reg}}$$

Figure 5.5b represents the pipelined version of the same circuit working faster because only one from adder *and the* adder/subtracter *contributes to limiting the clock frequency:*

$$f_{clock} = \frac{1}{max(t_{+/-}, t_{+}) + t_{reg}}$$

A small price is paid by $\lambda = 1$.

The 1-bit loop closed from the output `out[15]` *to* `cond` *input (see Figure 5.5c) allows the circuit* to decide itself *if the sum or the difference is accumulated. Its speed is identical with the initial, no-loop, circuit.*

Figure 5.5d warns us against the expected damages of closing a loop in a pipelined system. Because of the latency the "decision comes" to late and the functionality is altered. $\diamond$

In the system from Figure 5.5a the degree of parallelism is 1, and in Figure 5.5b the system has the degree of parallelism 2, because of the pipeline execution. When we closed the loop we where obliged to renounce to the bigger degree of parallelism because of the latency associated with the pipe. We have a new functionality – the circuit decides itself regarding the function executed in each clock cycle – but we must pay the price of reducing the speed of the system.

According to the algorithm the function performed by the block `+/-` *depends on data* received in the previous clock cycles. Indeed, the sign of the number stored in the output register depends on the data stream applied on the inputs of the system. We call this effect **data dependency**. It is responsible for limiting the degree of parallelism in digital circuits.

The circuit from Figure 5.5d is not a solution for our problem because the condition `cond` comes to late. It corresponds to the operation executed on the input stream excepting the most recently received pair of data. The condition comes too late, with a delay equal with the latency introduced by the pipeline execution.

How can we avoid the speed limitation imposed by a new loop introduced in a pipelined execution? It is possible, but we must pay a price enlarging the structure of the circuit.

If the circuit does not know what to do, addition or subtract in our previous example, then in it will be compute both in the first stage of pipeline and will delay the decision for the next stage so compensating the latency. We use the same example to be more clear.

Example 5.4 *The pipelined version of the circuit* `addapt_acc` *is provided by the following Verilog code:*

```
/*****************************************************************
File name:      adapt_acc1.v
```

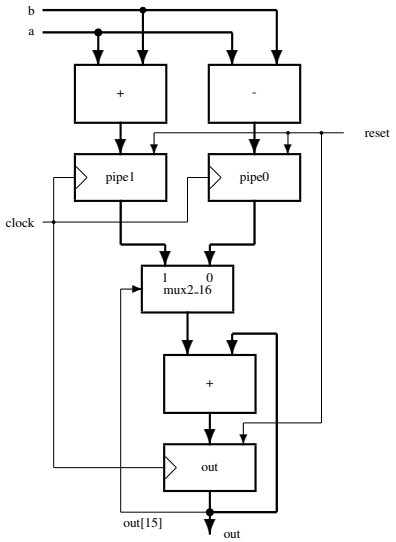

Figure 5.6: **The speculating solution to avoid data dependency.** In order to delay the moment of decision both addition and subtract are computed on the first stage of pipeline. Speculating means instead to decide what to do, addition or subtract, we decide what to consider after doing both.

```
Circuit name:    Pipelined Adaptive Accumulator
Description:     the behavioral description circuit
*************************************************************************/
module adapt_acc1(output  reg [15:0]  out,
                  input       [15:0]  a, b,
                  input               reset, clock);
reg    [15:0]  pipe1, pipe0;
always @(posedge clock)
   if (reset)  begin   out <= 0;
                       pipe1 <= 0;
                       pipe0 <= 0; end
    else   begin   pipe1 <= a + b;
                   pipe0 <= a - b;
                   if (out[15])     out <= out + pipe1;
                    else            out <= out + pipe0; end
endmodule
```

The execution time for this circuit is limited by the following clock frequency:

$$f_{clock} = \frac{1}{max(t_+, t_-, (t_+ + t_{mux})) + t_{reg}} \simeq \frac{1}{t_- + t_{reg}}$$

The resulting frequency is very near to the frequency for the pipeline version of the circuit designed in the previous example.

Roughly speaking, the price for the speed is: an adder & two registers & a multiplexer *(see for comparing Figure 5.5c and Figure 5.6). Sometimes it deserves!* ◇

The procedure applied to design `addapr_acc1` involves the multiplication of the physical resources. We *speculated*, computing on the first level of pipe both the sum and the difference of the input values. On the second state of pipe the multiplexer is used to select the appropriate value to be added to `out`.

We call this kind of computation *speculative evaluation* or simply **speculation**. It is used to accelerate complex (i.e., "under the sign" of a loop) computation. The price to be paid is an increased size of the circuit.

5.1.3 Coding style issues

An important key question we should ask ourself is:

Is this code the best way to capture the algorithm?

Sometimes, this way to put the question is misleading. The goal of our design is not only the highest performance of the resulting circuit, but how maintainable and modifiable is the code we write. Therefore, it is not only about a stylistic preference, because coding style can sometimes limit the hardware organization that a HLS tool can generate from a our critical piece of code.

Unfortunately, the first bad news is: there are a multitude of criteria for the coding style when HLS tools are used. It is difficult to make a winning compromise between:

- the clarity of the code
- the maintainability of the code
- the ease of modifying it
- the achievement of the pursued target as controllable as possible

all this being accompanied too often by modest knowledge in the field of digital circuits.

The second bad news: there is no theory about how the compromise can be resolved. First, because the behavior of the tools we use is not well documented. But we cannot neglect the fact that HLS users come from professional environments that extend between programming and digital design. The habits acquired at the two extremes are practically incompatible. Consequently, it is difficult to speak of a style. We will have to practice solving problems through reconfigurable computing for a while before we can talk about a specific style.

The first good news: we have a solution, well grounded in theory, to speed up intense computing. The natural parallelism offered by the circuits is a hope.

The second god news: it is very promising the fact not that the entire process of the Vivado HLS tool is constantly being improved with each new release.

5.2 Examples

This section is based on examples investigated in [Kastner '20] and [Crockett '15].

5.2.1 FIR

```
/*****************************************************************
File name: fir.cpp
Circuit name:
Description:   11 tap FIR filter.
```

```
*********************************************************************/
    #include "fir.h"

    void fir(data_t *y, data_t x) {
        coef_t c[N] = {53, 0, -91, 0, 313, 500, 313, 0, -91, 0, 53};
        static
            data_t shift_reg[N];
        acc_t acc;
        int i;

        acc = 0;
        Shift_Accum_Loop:
        for (i=N-1; i>=0; i--) {
            if (i == 0) {
                acc += x * c[0];
                shift_reg[0] = x;
            } else {
                shift_reg[i] = shift_reg[i-1];
                acc += shift_reg[i] * c[i];
            }
        }
        *y = acc;
    }
```

```
/*********************************************************************
File name: fir.h
Circuit name:
Description:   11 tap FIR filter.
*********************************************************************/
    #define N 11

    typedef int coef_t;
    typedef int data_t;
    typedef int acc_t;

    void fir(data_t *y, data_t x);
```

This description for the FIR function does not provide an efficient implementation. It is too sequential, and supposes a big amount of control logic.

Preliminaries about speed

The speed is expressed in clock frequency. It is imposed using `create_clock tcl` command. For example, `create_clock -period5` targets a clock period of 5 ns.

Lower target clock frequency give more leeway space for the Vivado HLS tool to combine multiple dependent operations in a single cycle. This process is called *operation chaining*. Once you have the first results for your design, you can vary the clock frequency and optimize experimentally your design.

Because Vivado HLS deals with clock frequency estimates, you must include some margin to account for the fact that there is some error in the time estimate. The goal of this margin is to en-

sure that the generated RTL code can be placed and routed. This margin can be controlled using the `set_clock_uncertainty` TCL command.

Clock period & circuit structure

The main operation in our application is multiply and accumulate (MACC). Multiplication and addition are performed on FPGA support depending on the technology node used to produce the FPGA. If we target the clock period, T_{clock}, having in mind the values for execution times associated to these two operations, t_{mult} and t_{add}, then we will be able to provide an important information for the synthesis tool to decide about the structure of the circuit. Some typical situations can be highlighted:

$\mathbf{T_{clock} \geq t_{mult} + t_{add}}$: MACC will implemented with a RTL code which performs it in one clock cycle, and the synthesis tool will generate accordingly the RTL code

$\mathbf{t_{mult} + t_{add} \geq T_{clock} > t_{mult}}$: MACC must be performed in two clock cycles, and the synthesis tool will generate accordingly the RTL code, considering the fact that $t_{mult} > t_{add}$

$\mathbf{t_{mult} \geq T_{clock} > t_{add}}$: MACC must be performed in at least three clock cycles, depending on the value of

$$k = \lceil t_{mult}/t_{add} \rceil$$

$\mathbf{t_{add} \geq T_{clock}}$: MACC must be performed in more than k cycles.

In order to find an optimal solution you change the target clock period and observe the differences in the performance and structure's size. Unfortunately, no one is able to provide the rule to pick the optimal frequency. We don't have an *"Art of codding"*. It is good or bad?

Code improvement

The **if** statement in the **for** loop is executed for each value of i. The main consequence is a complex control structure. Let us remove this **if** from the loop. The `Shift_Accum_Looop` will become:

```
Shift_Accum_Loop:
for (i=N-1; i>0; i--) {
    shift_reg[i] = shift_reg[i-1];
    acc += shift_reg[i] * c[i];
}

acc += x * c[0];
shift reg[0] = x;
```

Loop splitting

In order to indicate more clearly to the compiler the component of the circuit we intend to design the **for** loop must split in two distinct loops: *tapped delay line* (TDL) and *multiply accumulate* (MACC) as follows:

```
TDL:
for (i=N-1; i>0; i--) {
    shift_reg[i] = shift_reg[i-1];
}
shift_reg[0] = x;

acc = 0;
MACC:
for (i=N-1; i>0; i--) {
    acc += shift_reg[i] * c[i];
}
```

This makes the optimization process easier. Instead of optimizing a relatively complex loop, the compiler will have to independently optimize two simpler loops.

But beware: the reverse process can also be valid. Sometimes combining two simple loops into a more complex one can be compiled more efficiently. This fact depends very much on the real problem we are solving.

Tip: you always have the disposition to try as much variety of tricks as possible through which the code is written so that the physical structure generated corresponds to the requirements imposed by the person who formulated the problem.

Loop unrolling

The Vivado HLS tool synthesizes **for** loops by default in a sequential manner. The tool implements the circuit for one execution of the statements. The circuit executes the computation in N cycles per input sample (as in circuit represented in Figure 5.1). This creates an area efficient structure, but it limits the possibility to exploit parallelism that is present across loop iterations. By loop unrolling we replicate the body of the loop by a number of times.

Let us take the TDL loop and unroll it manually so as to execute two shifts at a time in the delay register.

```
TDL:
for (i=N-1; i>1; i=i-2) {
    shift_reg[i] = shift_reg[i-1];
    shift_reg[i-1] = shift_reg[i-2];
}
if (==1) {
    shift_reg[1] = shift_reg[0];
}
shift_reg[0] = x;
```

The **if** statement is for the case when N is an odd number.

The same effect is obtained using the **unroll** directive by inserting right after the **for** loop header the directive:

```
#pragma HLS unroll factor=2
```

For the MACC loop the unrolling by a factor of 4 is the following:

```
acc = 0;
MAC:
for (i=N-1; i>=3; i-=4) {
    acc +=  shift_reg[i] * c[i] +
            shift_reg[i-1] * c[i-1] +
            shift_reg[i-2] * c[i-2] +
            shift_reg[i-3] * c[i-3];
}
for (; i>=0; i--) {
    acc += shift_reg[i] * c[i];
}
```

The same effect is obtained using the **unroll** directive by inserting right after the **for** loop header the directive:

```
#pragma HLS unroll factor=4
```

If the optional argument `skip_exit_check` is specified in that directive, the tool will not add the final **for** loop. This is useful when you know that the loop will never require the final partial iterations.

The for loop is completely unrolled when no factor argument is specified.

Loop pipelining

Because, by default the Vivado HLS tool implements —bf for loops sequentially, loop pipelining must be specified explicitly when considered. Related with the pipelining execution we must define two types of latency.

Definition 5.1 ***Iteration latency*** *is the number of clock cycle that it takes to perform one iteration in a* **for** *loop.*

◇

Definition 5.2 for *loop latency* *is the number of clock cycle that it takes to perform all the iterations in a* **for** *loop.*

◇

Definition 5.3 ***Loop initialization interval****, II, is the number of clock cycles until the next iteration of the* **for** *loop can start.*

◇

A possible scenario for MACC operation consist of three operations: read (constant and value, in one cycle), multiply (constant with value, in two cycles), add (in one cycle). Results: $iteration\_latency = 4$. Because each iteration is independent, at each clock cycle one new iteration can start in a pipelined fashion. Therefore, for pipeline execution with $N = 11$ and $II = 1$ results $for\_loop\_latency = 14$ instead of $for\_loop\_latency = 44$ provided by the sequential execution.

The directive used for pipelining execution with loop initialization interval 1 is:

```
#pragma HLS pipeline II=2
```

Select hardware resources

If we intend to map an operation to a hardware resource of the FPGA a specific pragma is available. For example, the execution of the code `a = b + c` can be distributed to a DSP48 writing:

```
#pragma HLS RESOURCE variable=a core=AddSub_DSP
```

Arbitrary precision data types

Vivado HLS provides arbitrary precision data types for signed and unsigned of any bitwidth:

Unsigned : `ap_uint<width>`

Signed : `ap_int<width>`

For example: `ap_int<12>` for data provided by a 12-bit ADC.

To use these arbitrary precision data types we must use C++ and include the file `ap_int.h`, i.e., add the code **#include**`"ap_int.h"` in your project and use a filename ending in `.cpp`.

Example 5.5 *For the operation:* $c = a \times b$, *if* `ap_int<x> a, ap_int<y> b, ap_int<z> c)` *then* $z = x + y$.

◇

Concluding about FIR example

We succeeded to implement a pipelined version of the solution presented in Figure 5.1. Until the solution in Figure 5.4 we have a few more steps to go.

How Vivado HLS works

Example 5.6 *Let us use as input for Vivado HLS the FIR filter defined by* `fir.cpp` *and* `fir.h`. *The following benchmark must be added.*

```
/*************************************************************************
File name: fir_test.cpp
Circuit name:
Description:   11 tap FIR filter benchmarck.
*************************************************************************/
    #include <iostream>
    #include "fir.h"

    using namespace std;

    int main(data_t *y, data_t x){
        coef_t c[N] = {53, 37, -91, 38, 313, 500, 313, 39};
        data_t shift_reg[N] = {5, 5, 5, 5, 5, 5, 5, 5};
        data_t x = 1;

        data_t  hw_result, sw_result;
        int error = 0;

        // Generate the expected result
        sw_result = 0;
        Shift_Accum_Loop:
        for (i = N - 1; i >= 0; i--) {
            if (i == 0) {
                sw_result += x * c[0];
                shift_reg[0] = x;
            } else {
                shift_reg[i] = shift_reg[i - 1];
```

```
            sw_result += shift_reg[i] * c[i];
        }
    }
    #ifdef HW_COSIM
        // Run the Vivado HLS fir
        fir(hw_result, x);
    #endif

    #ifdef HW_COSIM
        // Check result of HLS vs. expected
        if (hw_result != sw_result) {
            error_count++;
        }
    #else
        cout << sw_result;
    #endif

    #ifdef HW_COSIM
        if (error_count)
            cout << "TEST FAIL: " << error_count
            << "Results do not match!" << endl;
        else
            cout << "Test passed!" << endl;
    #endif
    return error_count;
}
```

To the file `fir.cpp` will be improved removing the **if** statement from the **for** loop and by splitting the `Shift_Accum_Loop` in two lops havin the label names `TDL` and `MAC`. The code in `fir.cpp` becomes:

```
/*************************************************************************
File name: fir.cpp
Circuit name:
Description:
*************************************************************************/
    void fir(data_t *y, data_t x) {
        coef_t c[N] = {53, 37, -91, 38, 313, 500, 313, 39};
        static data_t shift_reg[N] = {5, 5, 5, 5, 5, 5, 5, 5};
        acc_t acc;

        TDL:
        for (int i = N - 1; i > 0; i--)
        {
                shift_reg[i] = shift_reg[i - 1];
        }
        shift_reg[0] = x;

        acc = 0;
        MAC:
        for (int i = N - 1; i > 0; i--)
```

```
        {
                acc += shift_reg[i] * c[i];
        }
        acc += x * c[0];

        *y = acc;//result
    }
```

```
/************************************************************************
File name: fir.h
Circuit name:
Description:
************************************************************************/
    #ifndef __FIR_H__
    #define __FIR_H__

    #include <cmath>
    using namespace std;

    // Compare TB vs HW C-model and/or RTL
    #define HW_COSIM

    #define N 8

    typedef short coef_t;
    typedef short data_t;
    typedef int acc_t;

    void fir(data_t *y, data_t x);

    #endif
```

Clock cycle	LUTs	FFs	DSPs	16-bit REGs	Latency
1.75 ns	53	406	8	24	10
2 ns	52	264	8	16	7
3 ns	52	248	8	15	7
4 ns	51	182	8	11	5
8 ns	52	147	8	9	2
10 ns	51	114	8	7	1

Table 5.1: The experiments done with the Vivaro HLS tool for designing the FIR filter.

In the `fir.h` file we set the design for 8 stages in the filter, by `#define N 8`, and for 16-bit numbers. With the Vivado HLS tool we used the flowing directives:

% HLS UNROLL for TDL loop and MAC loop

% HLS PIPELINE for MAC loop.

The design is implemented for different clock cycles, from 1.75*ns* to 10*ns*. The results are summarized in Table 5.1.

The speed is controlled by the number of pipeline registers used in design. At the lowest speed only the registers from the shift register are involved in design. We are confident in continuing the implementation with the 2*ns* version because the estimated clock cycle is 1.82*ns* with an uncertainty of 0.25*ns*. With the $iteration\_latency = 7$ and $II = 1$, we obtain $loop\_latency = 14$.

◇

5.2.2 Matrix-Vector Multiplication

The square matrix *M* of $SIZE \times SIZE$ is multiplied with the *V_In* vector of size *SIZE*. The result is the vector *V_Out*.

```
#define SIZE 8
typedef int BaseType;

void matrix_vector(BaseType M[SIZE][SIZE],
                   BaseType V_In[SIZE], BaseType V_Out[SIZE]) {
    BaseType i, j;
    Data loop:
    for (i=0; i< SIZE; i++) {
        BaseType sum = 0;
        dot product loop:
        for (j=0; j<SIZE; j++) {
            sum += V_In[j] * M[i][j];
        }
        V_Out[i] = sum;
    }
}
```

`BaseType` is mapped as a **float**.

This code, without directives, will be synthesized as a sequential circuit with one multiplier and one adder.

Parallelism

The inner loop can be parallelized explicitly with the following code:

```
#define SIZE 8
typedef int BaseType;

void matrix vector(BaseType M[SIZE][SIZE],
                   BaseType V In[SIZE], BaseType V Out[SIZE]) {
    BaseType i, j;
    data loop:
    for (i=0; i<SIZE; i++) {
        V_Out[i] = V_In[0] * M[i][0] +
                   V_In[1] * M[i][1] +
                   V_In[2] * M[i][2] +
```

```
                    V_In[3] * M[i][3] +
                    V_In[4] * M[i][4] +
                    V_In[5] * M[i][5] +
                    V_In[6] * M[i][6] +
                    V_In[7] * M[i][7];
    }
}
```

or by using #pragma HLS unroll. The circuit associated to the inner loop becomes a 4-level binary tree. The first level computes 8 multiplications, the second 4 additions, the third 2 additions and the last one addition. Assuming that multiplication has a latency of 3 clock cycle and the addition is executed in 1 clock cycle, the overall latency is 6.

Array partitioning

Global configuration of array partitioning is possible based on the `config_array_partition` project option. Big arrays can be partitioned using the **array_partition** directive. The directive **array_partition** splits each element of an array into its own register, resulting in a flip-flop based implementation.

```
#define SIZE 8
typedef int BaseType;

void matrix_vector(BaseType M[SIZE][SIZE],
                   BaseType V_In[SIZE], BaseType V_Out[SIZE]) {
    #pragma HLS array partition variable=M dim=2 complete
    #pragma HLS array partition variable=V_In complete
    BaseType i, j;
    data loop:
    for (i=0; i<SIZE; i++) {
        #pragma HLS pipeline II=1
        BaseType sum = 0;
        dot product loop:
        for (j=0; j<SIZE; j++) {
            sum += V_In[j] * M[i][j];
        }
        V_Out[i] = sum;
    }
}
```

The Vivado HLS tool provides the fastest solution starting from the previous code. Because the iteration latency is 6 (3 for multiplication and 3 for the addition tree) and the initialization interval is 1, the loop latency of the inner loop is 13.

Another form of array partitioning is to use **array_partition** `cyclic`.

Example 5.7 *Let be* `X = [1, 2, 3, 4, 5, 6, 7, 8, 9]` *Applying:*

array_partition `variable=x factor=2 cyclic`

results two arrays which are [1, 3, 5, 7, 9] and [2, 4, 6, 8].

◇

An intermediate solution is provided by the following code where the inner loop of matrix-vector multiply manually unrolled by a factor of two.

```
#define SIZE 8
typedef int BaseType;

void matrix_vector(BaseType M[SIZE][SIZE],
                   BaseType V_In[SIZE], BaseType V_Out[SIZE]) {
    #pragma HLS array_partition variable=M dim=2 cyclic factor=2
    #pragma HLS array_partition variable=V_In cyclic factor=2
    BaseType i, j;
    data_loop:
    for (i = 0; i < SIZE; i++) {
        BaseType sum = 0;
        dot_product_loop:
        for (j = 0; j < SIZE; j+=2) {
            #pragma HLS pipeline II=1
            sum += V_In[j] * M[i][j];
            sum += V_In[j+1] * M[i][j+1];
        }
        V_Out[i] = sum;
    }
}
```

5.2.3 FFT

5.2.4 SpMV

5.2.5 Matrix Multiplication

HLS version

```
#include "matrix_mult.h"

void matrix_mult(
      mat_a a[IN_A_ROWS][IN_A_COLS],
      mat_b b[IN_B_ROWS][IN_B_COLS],
      mat_prod prod[IN_A_ROWS][IN_B_COLS])
{
  // Iterate over the rows of the A matrix
   Row: for(int i = 0; i < IN_A_ROWS; i++) {
      // Iterate over the columns of the B matrix
      Col: for(int j = 0; j < IN_B_COLS; j++) {
         prod[i][j] = 0;
         // Do the inner product of a row of A and col of B
         Product: for(int k = 0; k < IN_B_ROWS; k++) {
            prod[i][j] += a[i][k] * b[k][j];
         }
      }
   }
```

```
}
```

```
#ifndef __MATRIXMUL_H__
#define __MATRIXMUL_H__

#include <cmath>
using namespace std;

// Compare TB vs HW C-model and/or RTL
#define HW_COSIM

#define IN_A_ROWS 5
#define IN_A_COLS 5
#define IN_B_ROWS 5
#define IN_B_COLS 5

typedef char mat_a;
typedef char mat_b;
typedef short mat_prod;

// Prototype of top level function for C-synthesis
void matrix_mult(
      mat_a a[IN_A_ROWS][IN_A_COLS],
      mat_b b[IN_B_ROWS][IN_B_COLS],
      mat_prod prod[IN_A_ROWS][IN_B_COLS]);

#endif // __MATRIXMUL_H__ not defined
```

The first solution provided by the Vivado HLS tool is implemented using 41 LUTs, 49 FFs and one DSP. The design performs the computation in 687 cycles ($\sim$ 27 cycles per element of the result matrix. The implementation is obviously sequential and the execution time provided by the resulting circuit is in the same range with the execution time provided by running the C program.

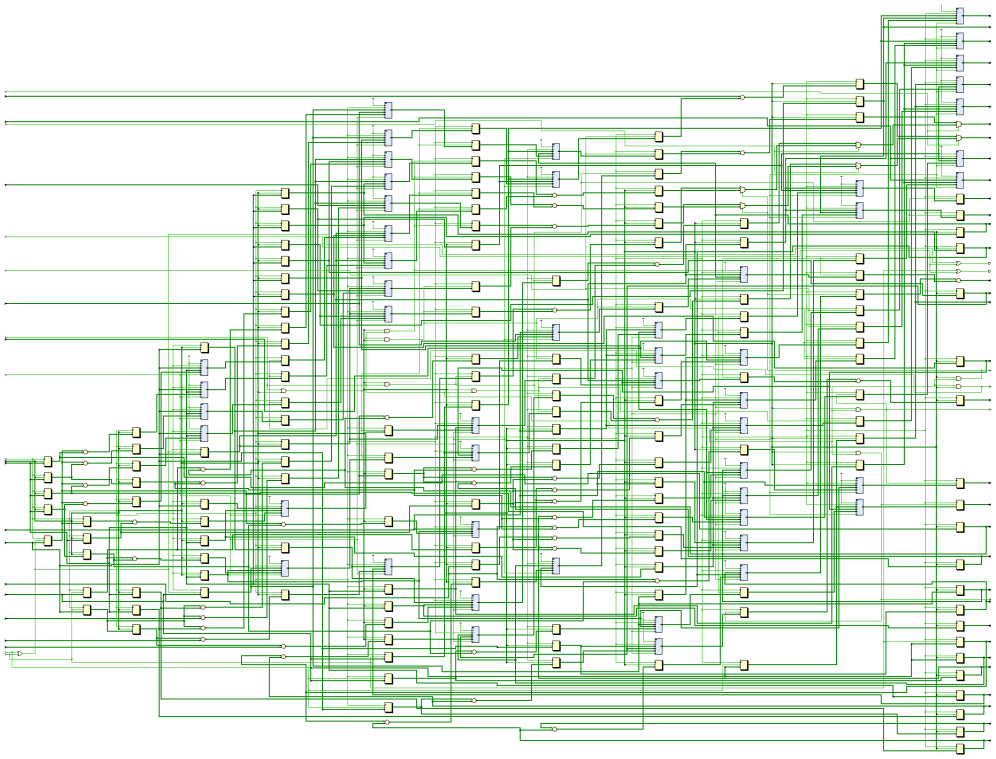

Figure 5.7: The HLS version of the circuit for 5×5 8-bit matrix multiplication.

The fifth solution: 635 LUTs, 665 FFs, and 122 DSPs. The solution is provided in 23 cycles.

Hand codded version

The hand codded version of the matrix multiplication, designed for 16-bit numbers, is presented in order to compare it with the HLS version for 8-bit numbers. If an experienced circuit designer writes the code it looks as follows:

```
/*************************************************************************
File name: mmm.v
Circuit name: Matrix multiplier
Description: Multiplies 5x5 matrices. The matrix A is received line by
             line, while the matrix B is inlined to the input b. The
             result matrix C is generated line by line with latency 4
*************************************************************************/
module  mmm(input       [399:0] b   , // Matrix B inlined
            input       [79:0]  a   , // Lines of matrix A
            output      [159:0] c   , // Lines of matrix C
            input               clk ); // my matrix multiplier

    wire    [79:0]  col[0:4];
    wire    [31:0]  s[0:4]  ;
```

```
    // Compose the line of the matrix C
    assign c = {s[0],s[1],s[2],s[3],s[4]}   ;
    // Select the columne of the matrix B
    assign
    col[0] = {b[399:384], b[319:304], b[239:224], b[159:144], b[79:64]} ,
    col[1] = {b[383:368], b[303:288], b[223:208], b[143:128], b[63:48]} ,
    col[2] = {b[367:352], b[287:272], b[207:192], b[127:112], b[47:32]} ,
    col[3] = {b[351:336], b[271:256], b[191:176], b[111:96] , b[31:16]} ,
    col[4] = {b[335:320], b[255:240], b[175:160], b[95:80]  , b[15:0]}  ;

    genvar j    ;
    generate    begin: M
        for(j=0; j<5; j=j+1)
            mmmm   m(  .b  (col[j] ),
                       .a  (a      ),
                       .c  (s[j]   ),
                       .clk(clk    ));
    end
    endgenerate
endmodule
```

```
/*********************************************************************
File name:
Circuit name:
Description:
*********************************************************************/
module mmmm(input       [79:0]  b   , // Column of matrix B
            input       [79:0]  a   , // Line of matrix A
            output  reg [31:0]  c   , // Element of line in matrix C
            input               clk ); // my matrix multiplier module

    wire    [15:0]  B[0:4]  ;
    wire    [15:0]  A[0:4]  ;

    reg     [31:0]  C[0:4]  ;
    reg     [31:0]  S0[0:2] ;
    reg     [31:0]  S1[0:1] ;

    assign {A[0], A[1], A[2], A[3],A[4]} = a    ;
    assign {B[0], B[1], B[2], B[3],B[4]} = b    ;

    integer i   ;

    always @(posedge clk) for(i=0; i<5; i=i+1)
                          C[i] <= A[i] * B[i] ;

    always @(posedge clk)   begin   S0[0]   <= C[0] + C[1]  ;
                                    S0[1]   <= C[2] + C[3]  ;
                                    S0[2]   <= C[4]         ;
                                    S1[0]   <= S0[0] + S0[1];
```

```
                                    S1[1]    <= S0[2]          ;
                                    c        <= S1[0] + S1[1];
                                end
endmodule
```

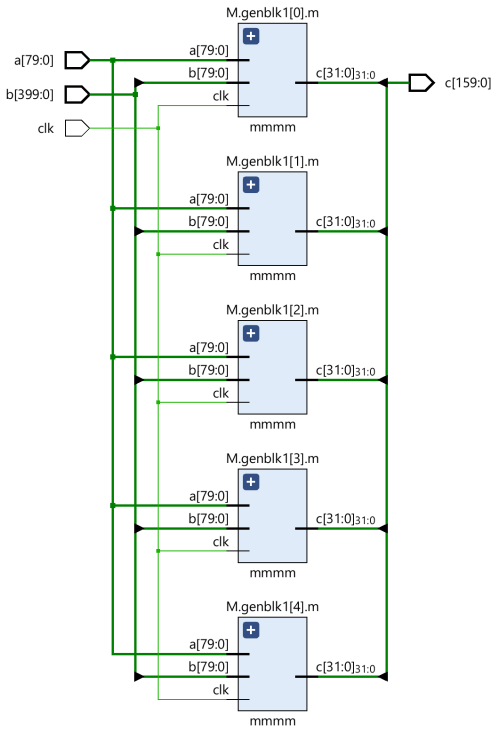

Figure 5.8: The top module for the hand codded version of the circuit for 5×5 16-bit matrix multiplication. It consists of 5 slices, one for each element of the line in the result matrix.

The synthesis tool report the use of 30 DSPs, and the latency is 9 clock cycles. In the first 5 cycles the lines of the matrix `A` are received to the input of the circuit. The first line of the result is provided to the output `c` triggered by the fourth clock cycle.

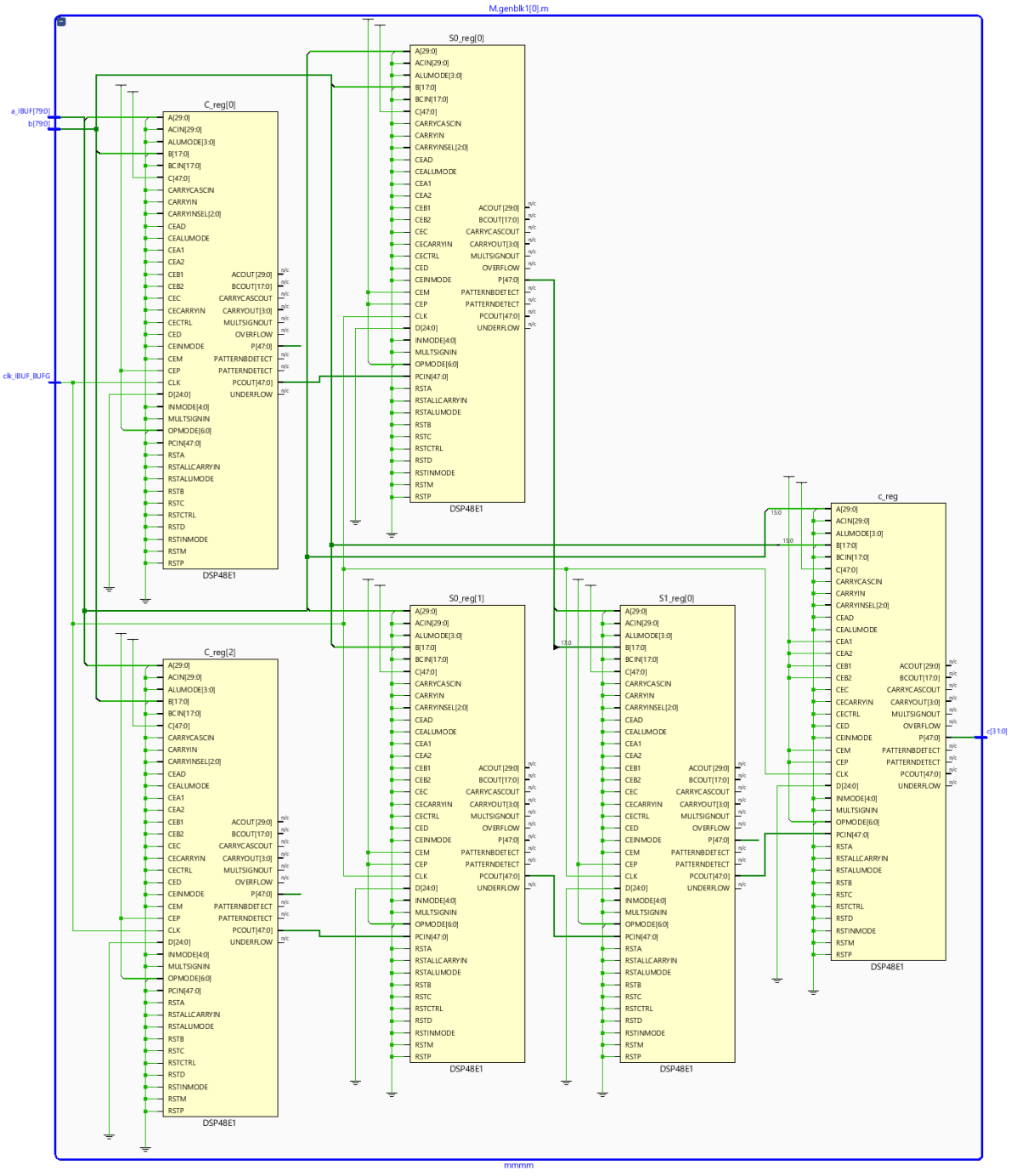

Figure 5.9: The structure of the slice used in Figure 5.8 for matrix multiplication.

5.2.6 Sorting

5.3 Programming Reconfigurable Systems

How looks a program

Part II

DIGITAL HIERARCHY

Chapter 6

Digital System Hierarchy

These ambiguities, redundancies and deficiencies remind us of those which doctor Franz Kuhn attributes to a certain Chinese encyclopedia entitled 'Celestial Empire of benevolent Knowledge'. In its remote pages it is written that the animals are divided into: (a) belonging to the emperor, (b) embalmed, (c) tame, (d) sucking pigs, (e) sirens, (f) fabulous, (g) stray dogs, (h) ***included in the present classification****, (i) frenzied, (j) innumerable, (k) drawn with a very fine camel-hair brush, (l) et cetera, (m) having just broken the water pitcher, (n) that from a long way off look like flies.* [Borges '52]

Jorge Luis Borges

In the fictitious taxonomy of animals, described by the writer Jorge Luis Borges in his 1942 essay "*The Analytical Language of John Wilkins*", the animals are divided into 14 strange categories. Meantime, the criteria have been modified many times because the image about the animal domain has changed under the pressure of a deeper understanding and a diversified use. The same process allows us to propose a new taxonomy in the digital world.

Thus, instead of classifying digital circuits in only two category:

1. combinational circuits: are digital networks of logic gates with no backward connections; the output of these systems depend, with a delay due to the propagation time associated to each gate, exclusively by the current binary values applied on the inputs
2. sequential circuits: are digital networks that include memory elements that determine an output response that depends on the time and binary configuration applied to the input

we are in the position to propose a more nuanced taxonomy based on the maximum number of included loops closed inside the digital system. The category of sequential circuits is detailed resulting the following loop-based taxonomy [**?**]:

zero-order digital systems (0-OS) : containing only combinational structures organized in a number of layers with no-loop closed from one layer back to one of the previous one, see Figure 6.1a

first-order digital systems (1-OS) : besides pure combinational sub-structures it contains memory circuits configured by closing one loop over a small and simple network of gates (see Figure 6.1b); the main storage elements are: latches, clocked latches, master-slave flip-flops, random-access memories, registers.

second-order digital systems (2-OS) : typically represented by finite automata (finite-state machines), see Figure 6.1c; other state machines, like counters, constitute a class of simple automata involved in various applications

third-order digital systems (3-OS) : typically represented by processor systems built by connecting in a loop a simple functional automaton with a complex control automaton, see Figure 6.1d

fourth-order digital systems (4-OS) : typically represented by computing systems based on the von Neumann abstract machine model (a data & program memory loop-connected with a processor), see Figure 6.1e

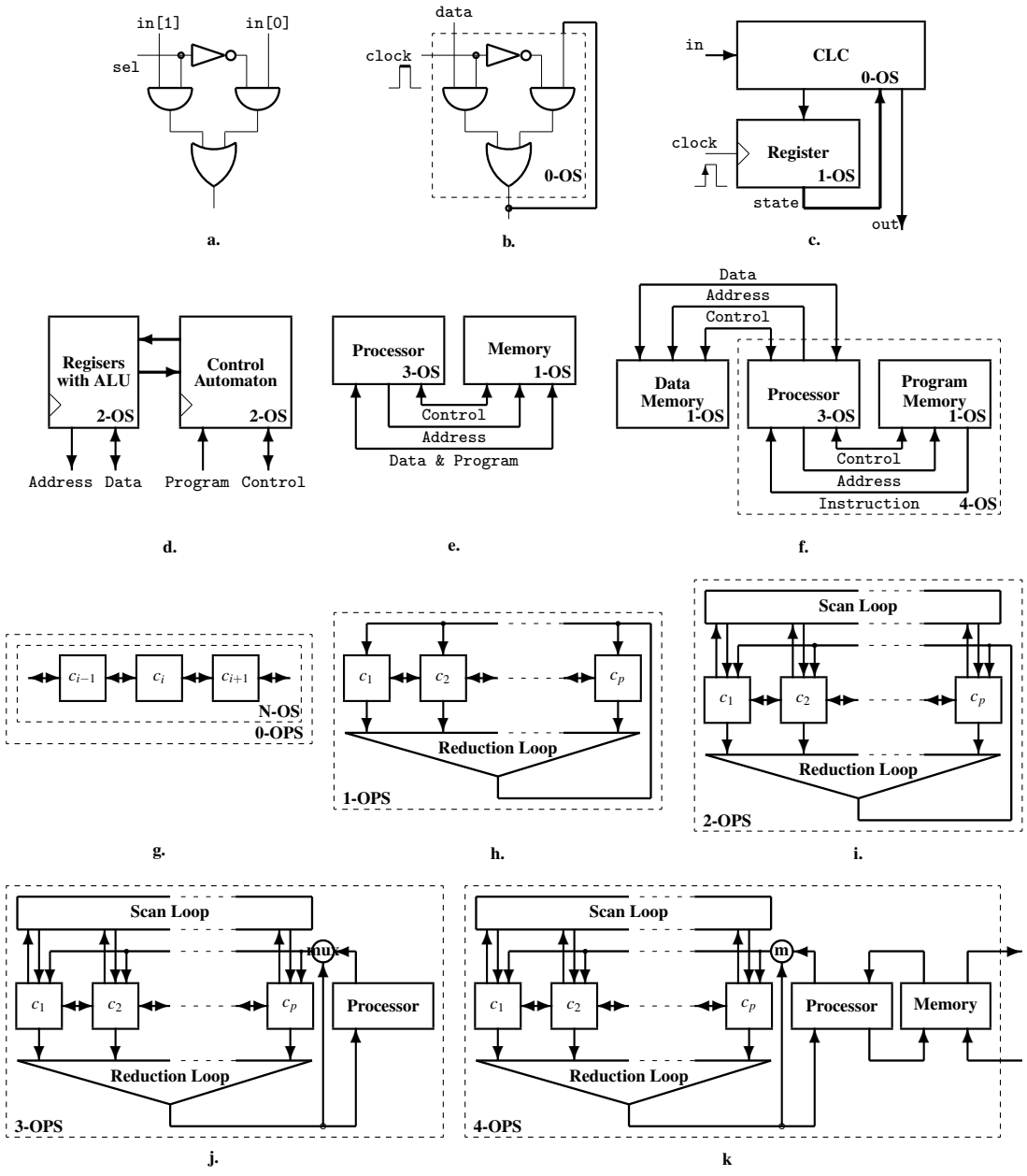

Figure 6.1: **The loop-based featuring mechanism. a.** Zero-order system (0-OS): a zero-loop combinational circuit. **b.** First-order system (1st-OS): the one-loop memory circuit. **c.** Second-order system (2nd-OS): the two-loop finite automaton circuit. **d.** Third-order system (3rd-OS): the processor as a three-loop system. **e.** Fourth-order system (4th-OS): the von Neumann abstract model for a computer as a four-loop system. **f.** Fifth-order system (5th-OS): the Harvard abstract model for a computer as a five-loop system. **g.** N-th order system (N-OS): the cellular automaton as a $O(N)$-loop system. **h.** The first global loop closed over a simple N-OS: the reduction loop. **i.** The second global loop closed over a simple N-OS: the scan loop. **j.** The third global loop closed over a simple N-OS: the scan loop. **k.** The fourth global loop closed over a simple N-OS: the scan loop.

fifth-order digital systems (5-OS) : typically represented by computing systems based on the Harvard abstract machine model (a computer with program memory only loop-connected with a data memory), see Figure 6.1f

...

n-order digital systems (N-OS) : typically represented by linear cellular automaton, see Figure 6.1g.

Once we reach the N-OS level, how do we move forward? We will close loops over N-OSs. Results:

zero-order parallel systems (0-OPS) : cellular automaton (N-OS), see Figure 6.1g

first-order parallel systems (1-OPS) : cellular automaton with direct reduce loop, see Figure 6.1h

second-order parallel systems (2-OPS) : cellular automaton with scan loop which includes the reduce loop, see Figure 6.1i

third-order parallel systems (3-OPS) : cellular automaton with scan loop with the reduce loop closed through a sequencer, see Figure 6.1j

third-order parallel systems (4-OPS) : is a Map-Scan-Reduce parallel computing system, see Figure 6.1k.

And further? In 'Celestial Empire of benevolent Knowledge' [Borges '52] I identified and emphasized a recursive item. Let us do something similar. Therefore, the proposal is a recursive growing on the frame of a 3-OPS:

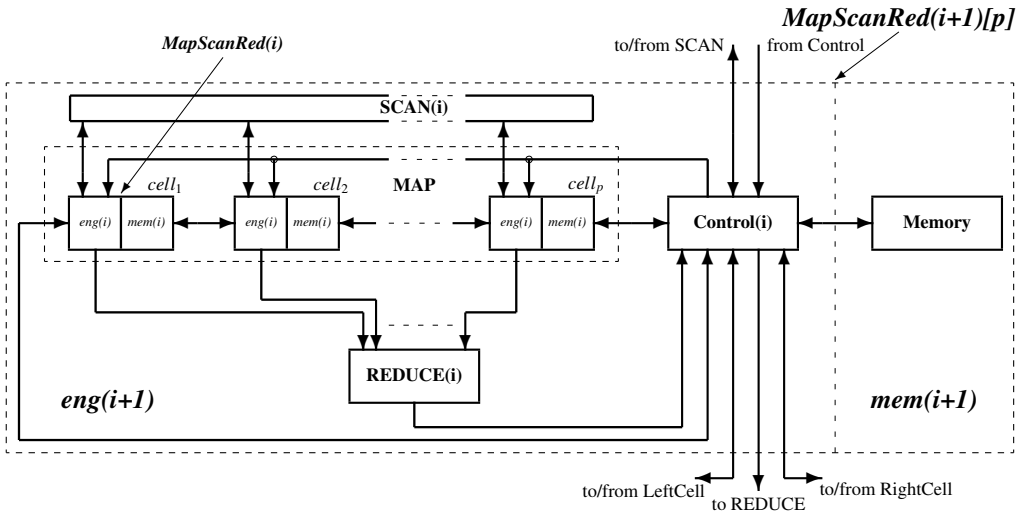

Figure 6.2: ***MapScanReduce recursive abstract model* for parallel computation.** In the lowest level in the hierarchy, $eng(0)$ is a scalar execution engine with $mem(0)$ a small static RAM.

zero-order recursive systems (0-ORS) : zero-order parallel systems designed as a 4-OPSs with a scalar execution unit and a data memory per cell

first-order recursive systems (1-ORS) : first-order parallel systems designed as a 4-OPSs with a 0-ORS as cell

...

n-order recursive systems (N-ORS) : n-order parallel systems designed as a 4-OPSs with a (N-1)-ORS as cell

6.1 Combinational Circuits: Zero-order Digital Systems

6.1.1 Behavioral vs. structural

DCD

```
/*****************************************************************
File name:      dec.v
Circuit name:   Decoder
Description:    behavioral description of a n-input decoder
*****************************************************************/
module dec #(parameter inDim = n)( input  [inDim - 1:0]         sel,
                                   output [(1 << inDim) - 1:0] out);
   assign out = 1 << sel;
endmodule
```

```
/*****************************************************************
File name:      dec.v
Circuit name:   Decoder
Description:    structural description of a 3-input decoder
*****************************************************************/
module dec3( output [7:0] out,
             input  [2:0] in );

    wire in0, nin0, in1, nin1, in2, nin2;

    not not00(nin0, in[0]);
    not not01(in0, nin0)  ;
    not not10(nin1, in[1]);
    not not11(in1, nin1)  ;
    not not20(nin2, in[2]);
    not not21(in2, nin2)  ;
    and and0(out[0], nin2, nin1, nin0); // output 0
    and and1(out[1], nin2, nin1, in0 ); // output 1
    and and2(out[2], nin2, in1,  nin0); // output 2
    and and3(out[3], nin2, in1,  in0 ); // output 3
    and and4(out[4], in2,  nin1, nin0); // output 4
    and and5(out[5], in2,  nin1, in0 ); // output 5
    and and6(out[6], in2,  in1,  nin0); // output 6
    and and7(out[7], in2,  in1,  in0 ); // output 7
endmodule
```

MUX

```
/*****************************************************************
File name:      mux.v
Circuit name:   Multiplexor
```

```
Description:    behavioral description for a n-selection inputs
                multiplexor
*************************************************************************/
 module mux #(parameter inDim = n)
        (input [inDim-1:0]       sel, // selection inputs
         input [(1<<inDim)-1:0] in , // selected inputs
         output                 out);
  assign out = in[sel];
endmodule
```

DMUX

```
/*************************************************************************
File name:      dmux.v
Circuit name:   Demultiplexor
Description:    behavioral description for a n-input demultiplexor
*************************************************************************/
 module dmux #(parameter inDim = n)(input  [inDim - 1:0]        sel    ,
                                    input                       enable,
                                    output [(1 << inDim) - 1:0] out    );
   assign out = enable << sel;
 endmodule
```

PE

```
/*************************************************************************
File name:      priority_encoder.v
Circuit name:   Priority Encider
Description:    behavioral description for a m-output priority encoder
*************************************************************************/
module priority_encoder #(parameter m = 4)
            (   input       [(1<<m)-1:0]    in      ,
                input                       enable  ,
                output  reg [m-1:0]         out     ,
                output  reg                 zero    );
    integer i;
    always @(*) if (enable) begin   out = 0;
                                    for(i=(1'b1 << m)-1; i>=0; i=i-1)
                                        if ((out == 0) && in[i]) out = i;
                                    if (in == 0)    zero = 1;
                                        else        zero = 0;
                            end
                    else    begin   out = 0;
                                    zero = 0;
                            end
endmodule
```

COMP

```
/*****************************************************************
File name:      comp.v
Circuit name:   Comparator
Description:    behavioral description for a n-bit words comparator
*****************************************************************/
module comp #(parameter n = 256)(output          lt_out   ,
                                                 eq_out   ,
                                                 gt_out   ,
                                 input   [n-1:0] a        ,
                                                 b        ,
                                 input           lt_in    ,
                                                 eq_in    ,
                                                 gt_in    );

    assign lt_out = lt_in | eq_in & (a < b);
    assign eq_out = eq_in & (a == b)       ;
    assign gt_out = gt_in | eq_in & (a > b);

endmodule
```

PREFIX

```
/*****************************************************************
File name:      addPrefixes.v
Circuit name:   ADD Prefixes
Description:    behavioral description for n-input add prefixes circuit
*****************************************************************/
 module prefixe #(`include "parameter.v")(  output  [0:m*n-1]   out ,
                                            input   [0:m*n-1]   in  );
    genvar i;
    generate    begin
        assign out[0:m-1] = in[0:m-1];
        for (i=1; i<n; i=i+1) begin
            assign out[i*m:(i+1)*m-1] =
                    in[i*m:(i+1)*m-1] + out[(i-1)*m:i*m-1] ;
        end
    end
    endgenerate
 endmodule
```

CSA

Carry-Save Adders

Four-input *n*-bit adders have two solutions:

Four-input *n*-bit adder with ripple-carry adders are implemented as a two layers system

Four-input n-bit adder with carry-save adders uses two carry-save addesr

```
/*****************************************************************
File name:      eCSA.v
Circuit name:   Elementary Carry-Save Adder
Description:    A m-bit inputs carry-save adder
*****************************************************************/
module eCSA #(parameter n=16)(  input   [n-1:0] in0, in1, in2   ,
                                output  [n:0]   sOut, cOut      );
    wire    [n-1:0] out ;
    wire    [n-1:0] cr  ;

    genvar  i   ;
    generate for (i=0; i<n; i=i+1) begin: S
        fa adder(in0[i], in1[i], in2[i], out[i], cr[i]);
        end
    endgenerate

    assign sOut = {1'b0, out}   ;
    assign cOut = {cr, 1'b0}    ;
endmodule

module fa(      input   in1, in2, cIn   ,
           output  out, cOut        );

    assign out  = in1  ^ in2  ^ cIn                 ;
    assign cOut = in1  & in2  | cIn  & (in1  ^ in2) ;
endmodule
```

```
/*****************************************************************
File name:      add4.v
Circuit name:   Four-input adder
Description:    A m-bit 4-inputs adder based on carry-save adders
*****************************************************************/
module add4 #(parameter n=16)(  output  [n+1:0] out                 ,
                                input   [n-1:0] in0, in1, in2, in3  );

    wire    [n:0]   sOut1, cOut1;
    wire    [n+1:0] sOut2, cOut2;

    eCSA    eCSA1(  .in0(in0    ),
                    .in1(in1    ),
                    .in2(in2    ),
                    .sOut(sOut1 ),
                    .cOut(cOut1 ));

    eCSA    #(.n(n+1))  eCSA2(  .in0(sOut1  ),
                                .in1(cOut1  ),
                                .in2({1'b0, in3}    ),
                                .sOut(sOut2 ),
```

```
                                    .cOut(cOut2 ));

    assign out = sOut2 + cOut2;
endmodule
```

6.1.2 Recursive descriptions

DCD

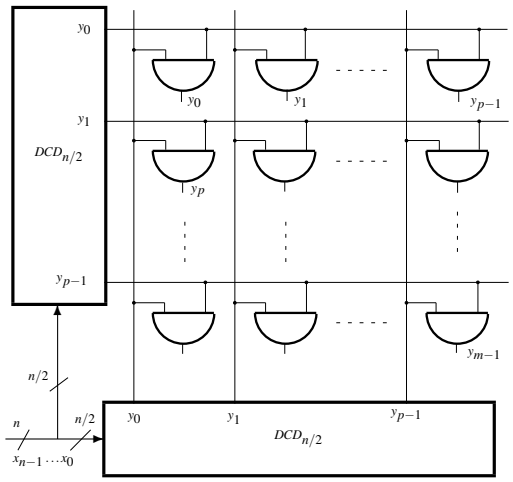

Figure 6.3: **The recursive definition of n-inputs decoder (DCD_n).**

```
/*****************************************************************
File name:      dec.v
Circuit name:   Decoder
Description:    recursive description of a n-input decoder
*****************************************************************/
module dec #(parameter n = 4)(  input  [n-1:0]       sel,
                                output [(1<<n)-1:0] out);

    wire    [(1<<n/2)-1:0]  out0;
    wire    [(1<<n/2)-1:0]  out1;

    genvar  i,j ;

    generate
    if (n == 1) eDec eDec(  .sel    (sel        ),
                            .out    (out[1:0]   ));
     else   begin   dec #(.n(n/2))  dec0(   .sel(sel[n/2-1:0]   ),
                                            .out(out0           )),
                                    dec1(   .sel(sel[n-1:n/2]   ),
```

```
                                                .out(out1              ));
                        for(i=0; i<(1<<n/2); i=i+1) begin
                            for(j=0; j<(1<<n/2); j=j+1) begin: outAND
                                and AND(out[(1<<n/2)*j+i], out1[j], out0[i]);
                            end
                        end
            end
    endgenerate
endmodule

// Elementary decoder
module eDec(input           sel ,
            output  [1:0]   out );

    not bufNot(out[0], sel      );
    not outNot(out[1], out[0]   );

endmodule
```

MUX

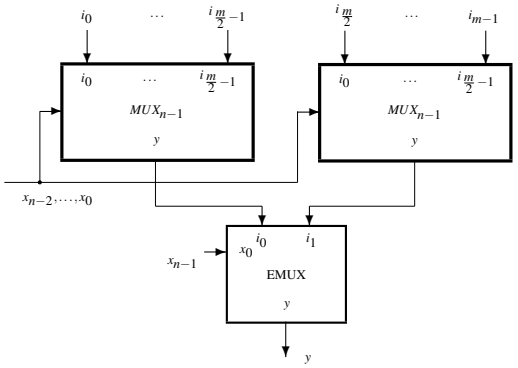

Figure 6.4: **The recursive definition of MUX_n.**

DMUX

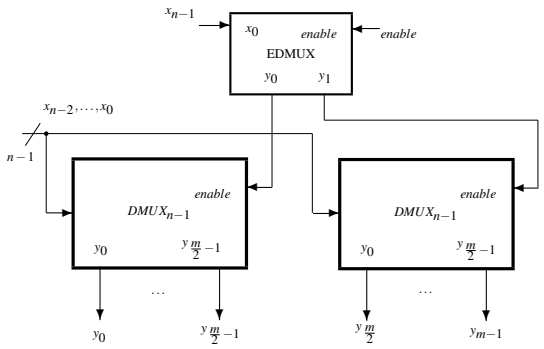

Figure 6.5: **The recursive definition of $DMUX_n$.**

PE

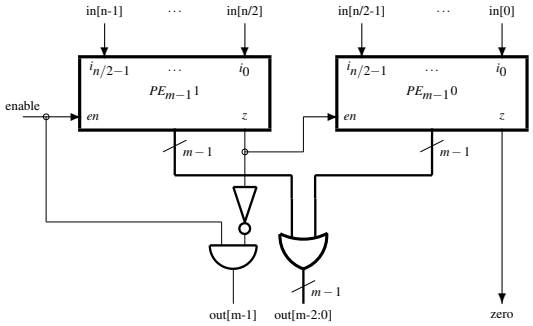

Figure 6.6: **The recursive definition of PE_m.**

```
/*************************************************************************
File name:      priority_encoder.v
Circuit name:   Priority Encider
Description:    behavioral description for a m-output priority encoder
*************************************************************************/
module priority_encoder #(parameter m = 4)
        (   input   [(1<<m)-1:0]    in      ,
            input                   enable  ,
            output  [m-1:0]         out     ,
            output                  zero    );
    wire    [m-2:0] out0    ;
    wire    [m-2:0] out1    ;
    wire            zero1   ;
```

```
    generate
    if (m == 2) begin
                    assign zero     = enable & ~(|in)                    ;
                    assign out[1]   = enable & (in[3] | in[2])           ;
                    assign out[0]   = enable & (in[3] | ~in[2] & in[1]);
                end
     else    begin    priority_encoder #(.m(m-1))
                        pe0(     .in     (in[(1<<(m-1))-1:0]     ),
                                 .enable (zero1                  ),
                                 .out    (out0                   ),
                                 .zero   (zero                   )),
                        pe1(     .in     (in[(1<<m)-1:1<<(m-1)]  ),
                                 .enable (enable                 ),
                                 .out    (out1                   ),
                                 .zero   (zero1                  ));
                    assign out = {~zero1 & enable, (out0 | out1)};
            end
    endgenerate
endmodule
```

COMP

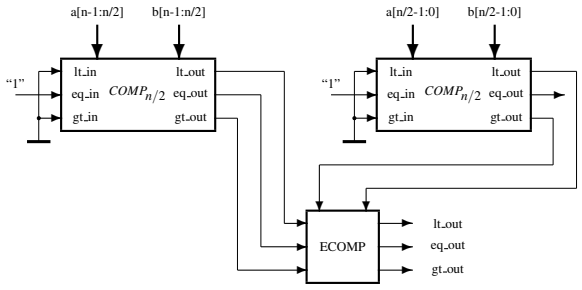

Figure 6.7: **The optimal *n*-bit comparator.** Applying the *divide et impera* principle a $COMP_n$ is built using two $COMP_{n/2}$ and an *ECOMP*. Results a *log*-depth circuit with the size in $O(n)$.

```
/*****************************************************************
File name:      comp.v
Circuit name:   Comparator
Description:    behavioral description for a n-bit words comparator
*****************************************************************/
module comp #(parameter n = 8)
        (   output          lt_out, eq_out, gt_out,
            input   [n-1:0] a, b,
            input           lt_in, eq_in, gt_in);
```

```
    wire    lt_out0, eq_out0, gt_out0, lt_out1, eq_out1, gt_out1    ;

    generate
    if (n == 1) e_comp e_comp(  lt_out, eq_out, gt_out,
                                a, b, lt_in, eq_in, gt_in);
        else    begin   comp #(.n(n/2)) comp0(  .lt_out (lt_out0    ),
                                                .eq_out (eq_out0    ),
                                                .gt_out (gt_out0    ),
                                                .a      (a[n/2-1:0] ),
                                                .b      (b[n/2-1:0] ),
                                                .lt_in  (lt_in      ),
                                                .eq_in  (eq_in      ),
                                                .gt_in  (gt_in      )),
                                        comp1(  .lt_out (lt_out1    ),
                                                .eq_out (eq_out1    ),
                                                .gt_out (gt_out1    ),
                                                .a      (a[n-1:n/2] ),
                                                .b      (b[n-1:n/2] ),
                                                .lt_in  (lt_in      ),
                                                .eq_in  (eq_in      ),
                                                .gt_in  (gt_in      ));
                        e_comp dut( .lt_out (lt_out ),
                                    .eq_out (eq_out ),
                                    .gt_out (gt_out ),
                                    .a      (gt_out0),
                                    .b      (lt_out0),
                                    .lt_in  (lt_out1),
                                    .eq_in  (eq_out1),
                                    .gt_in  (gt_out1));
                end
    endgenerate
endmodule

module e_comp(  output lt_out, eq_out, gt_out,
                input a, b, lt_in, eq_in, gt_in);

    assign  lt_out = lt_in | eq_in & ~a & b,
            eq_out = eq_in & ~(a ^ b),
            gt_out = gt_in | eq_in & a & ~b;
endmodule
```

SORT

```
/*****************************************************************
File name:      parameters.v
Circuit name:   it is not a circuit
Description:    defines the two parameters used in the sorter's definition
*****************************************************************/
 parameter  n = 16, // number of inputs
```

```
m = 8    // number of bits per input
```

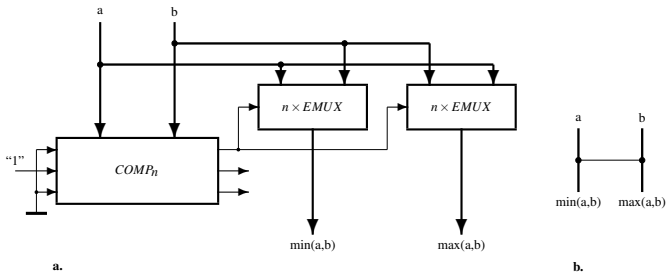

Figure 6.8: **The elementary sorter. a.** The internal structure of an elementary sorter. The output `lt_out` of the comparator is used to select the input values to output in the received order (if `lt_out = 1`) or in the crossed order (if `lt_out = 0`). **b.** The logic symbol of an elementary sorter.

```
/*********************************************************************
File name:      sorter.v
Circuit name:   Sorter Network
Description:    recursive definition for a sorter n m-bit numbers
*********************************************************************/
module sorter #(`include "0_parameters.v")
        (   output  [m*n-1:0]   out ,
            input   [m*n-1:0]   in  );

    wire    [m*n/2-1:0] out0;
    wire    [m*n/2-1:0] out1;

    generate
    if (n == 2)
        eSorter eSorter(.out0   (out[m-1:0]     ),
                        .out1   (out[2*m-1:m]   ),
                        .in0    (in[m-1:0]      ),
                        .in1    (in[2*m-1:m]    ));
        else    begin
                    sorter #(.n(n/2))   sorter0(.out(out0           ),
                                                .in (in[m*n/2-1:0]  )),
                                        sorter1(.out(out1           ),
                                                .in (in[m*n-1:m*n/2]));
                    merger  #(.n(n))    merger( .out(out            ),
                                                .in ({out1, out0}   ));
                end
    endgenerate
endmodule
```

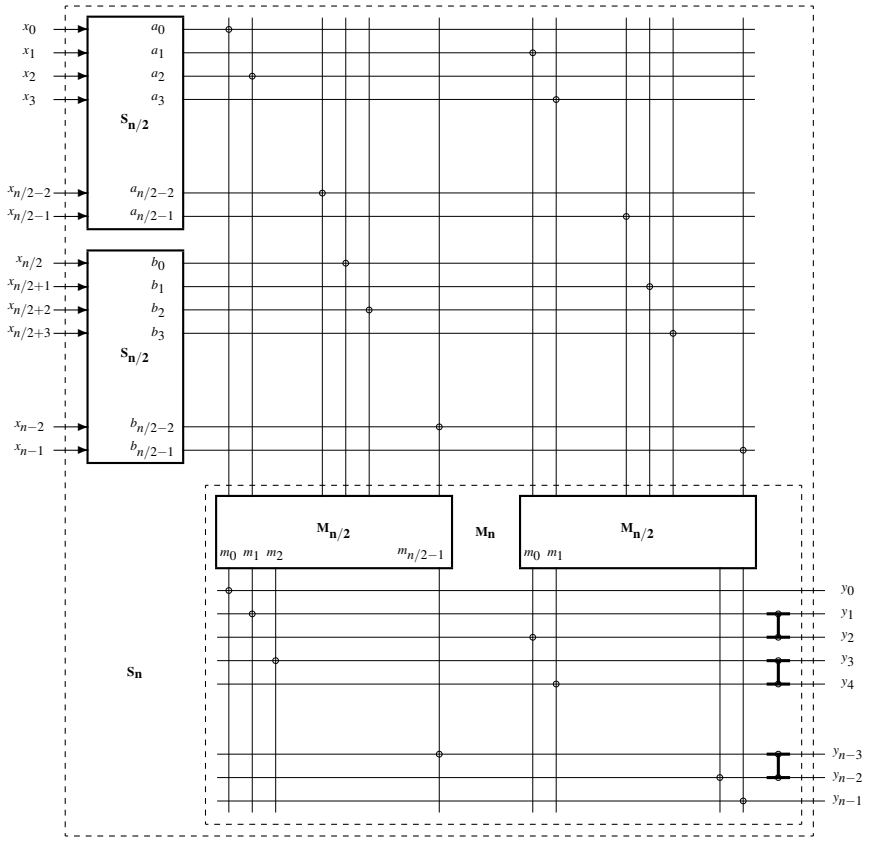

Figure 6.9: **Batcher's sorter.** The n-input sorter, S_n, is defined by a double-recursive construct: "$S_n = 2 \times S_{n/2} + M_n$", where the merger M_n consists of "$M_n = 2 \times M_{n/2} + (n/2-1)S_2$".

```
/*************************************************************************
File name:      eSorter.v
Circuit name:   Elementary Sorter
Description:    behavioral description of an elementary sorter for m-bit
                numbers
*************************************************************************/
module eSorter #(`include "0_parameters.v")
        (   output  [m-1:0] out0,
            output  [m-1:0] out1,
            input   [m-1:0] in0 ,
            input   [m-1:0] in1 );

    assign out0 = (in0 > in1) ? in1 : in0   ;
    assign out1 = (in0 > in1) ? in0 : in1   ;
endmodule
```

```
/*************************************************************************
File name:      merger.v
Circuit name:   Merger Network
Description:    recursive definition of a merger network for n m-bit
                numbers
*************************************************************************/
module merger #(`include "0_parameters.v")( output  [m*n-1:0]   out ,
                                            input   [m*n-1:0]   in  );
    wire    [m*n/4-1:0] even0   ;
    wire    [m*n/4-1:0] odd0    ;
    wire    [m*n/4-1:0] even1   ;
    wire    [m*n/4-1:0] odd1    ;
    wire    [m*n/2-1:0] out0    ;
    wire    [m*n/2-1:0] out1    ;

    genvar i;
    generate
    if (n == 2) eSorter eSorter(.out0   (out[m-1:0]     ),
                                .out1   (out[2*m-1:m]   ),
                                .in0    (in[m-1:0]      ),
                                .in1    (in[2*m-1:m]    ));
     else   begin
            for (i=0; i<n/4; i=i+1) begin : oddEven
                assign  even0[(i+1)*m-1:i*m] =
                            in[2*i*m+m-1:2*i*m]                 ;
                assign  even1[(i+1)*m-1:i*m] =
                            in[m*n/2+2*i*m+m-1:m*n/2+2*i*m]      ;
                assign  odd0[(i+1)*m-1:i*m] =
                            in[2*i*m+2*m-1:2*i*m+m]              ;
                assign  odd1[(i+1)*m-1:i*m] =
                            in[m*n/2+2*i*m+2*m-1:m*n/2+2*i*m+m]  ;
            end
            merger #(.n(n/2))   merger0(.out(out0           ),
                                        .in ({even1, even0} )),
                                merger1(.out(out1           ),
                                        .in ({odd1, odd0}   ));
            for (i=1; i<n/2; i=i+1) begin : elSort
                eSorter eSorter(.out0(out[(2*i-1)*m+m-1:(2*i-1)*m]  ),
                                .out1(out[2*i*m+m-1:2*i*m]          ),
                                .in0 (out0[i*m+m-1:i*m]             ),
                                .in1 (out1[i*m-1:(i-1)*m]           ));
            end
            assign out[m-1:0]           = out0[m-1:0]               ;
            assign out[m*n-1:m*(n-1)]   = out1[m*n/2-1:m*(n/2-1)]   ;
            end
    endgenerate
endmodule
```

REDUCE

```
/*************************************************************************
File name:      defines.v
Circuit name:   no circuit
Description:    defines the parameters for a n m-bit words reduction add
*************************************************************************/
    `define n 8             // number of inputs
    `define m 8             // input size
```

```
/*************************************************************************
File name:      reduce.v
Circuit name:   Reduce
Description:    recursive description for a n m-bit words reduction add
*************************************************************************/
`include "defines.v"

module reduce #(parameter N = `n)(  output  [`m-1:0]     out ,
                                    input   [N*`m-1:0]   in  );

    wire    [(N*`m/2)-1:0]  leftIn  ;
    wire    [(N*`m/2)-1:0]  rightIn ;
    wire    [`m-1:0]        lOut    ;
    wire    [`m-1:0]        rOut    ;

    genvar  i   ;
    generate
        if(N == 2)  assign out = in[2*`m-1:`m] + in[`m-1:0] ;
         else   begin   assign leftIn   = in[N*`m-1:N*`m/2];
                        assign rightIn  = in[(N*`m/2)-1:0];
                        reduce #(.N(N/2))   lr(lOut, leftIn ),
                                            rr(rOut, rightIn);
                        assign  out = lOut + rOut   ;
                end

    endgenerate
endmodule
```

PREFIX

A n-input prefix circuit, PXn, is built using elementary prefix circuits, ePx, defined by:

$$y_0 = x_0$$

$$y_1 = x_0 \circ y_1$$

where $\circ$ is an associative and commutative function. The recursive optimal structure is described, for $n = 8$, in Figure 6.11.

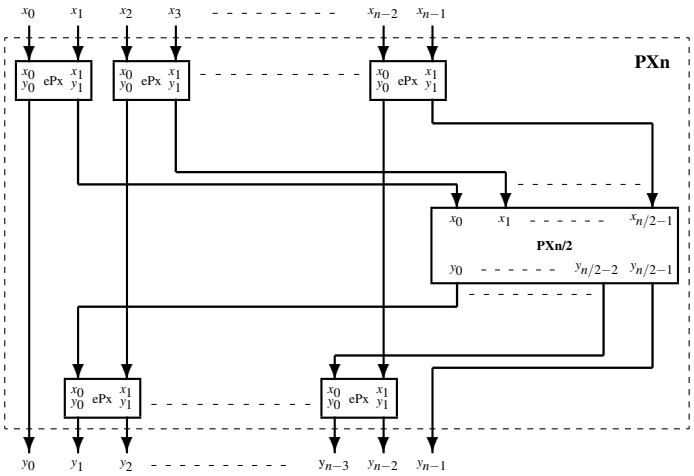

Figure 6.10: The recursive prefix network.

```
/*****************************************************************
File name:      parameter.v
Circuit name:   no circuit
Description:    defines the parameters and functions of the prefix network
*****************************************************************/
    parameter   m = 8,
                n = 16,
                func = 2'b00

    /*  func = 00: add
        func = 01: and
        func = 10: or
        func = 11: xor
    */
```

```
/*****************************************************************
File name:      prefix.v
Circuit name:   Elementary Sorter
Description:    recursive description of a n m-bit inputs prefix network
*****************************************************************/
module prefix #(`include "parameter.v")(output  [0:m*n-1]   out ,
                                        input   [0:m*n-1]   in  );

    wire    [m-1:0]         even[0:(n/2)-1]     ;
    wire    [m-1:0]         odd[0:(n/2)-1]      ;
    wire    [0:m*(n/2)-1]   hin                 ;
    wire    [0:m*(n/2)-1]   hout                ;
```

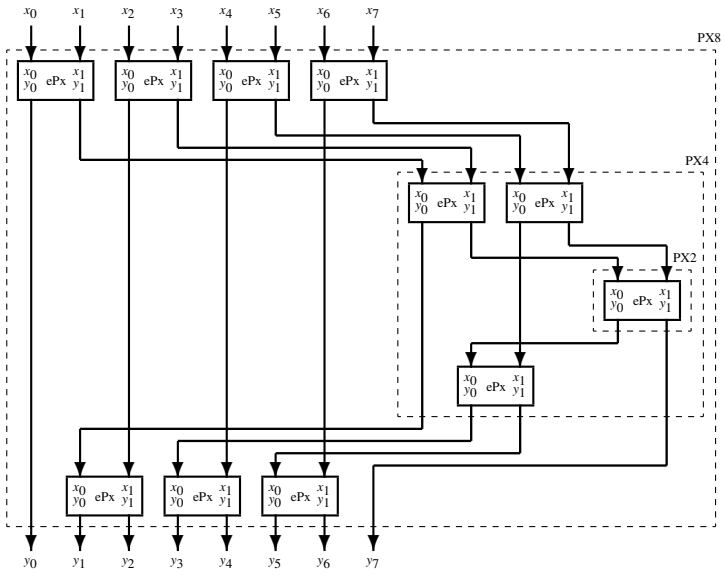

Figure 6.11: The recursive version of a 8-input prefix network.

```
genvar i     ;
generate
if (n == 2) ePrefix ePrefix(.out(out[0:2*m-1]),
                             .in (in[0:2*m-1] ));
 else
    begin
        for(i=0; i<n/2; i=i+1)  begin: ePxIn
            ePrefix ePrefix(     .out({even[i], odd[i]}       ),
                                 .in (in[2*i*m:2*(i+1)*m-1]   ));
            end
        for(i=0; i<n/2; i=i+1)  begin
            assign hin[m*i:m*(i+1)-1] = odd[i]  ;
        end
        prefix #(.n(n/2))    hPrefix(.out(hout),
                                     .in (hin ));
        assign  out[0:m-1] = even[0]     ;
        for(i=0; i<(n/2)-1; i=i+1)  begin: ePxOout
            ePrefix ePrefix(.out({  out[m*(2*i+1):m*(2*i+2)-1],
                                    out[m*(2*i+2):m*(2*i+3)-1]} ),
                             .in ({  hout[i*m:(i+1)*m-1],
                                     even[i+1]}                  ));
        end
        assign out[m*(n-1):m*n-1] = hout[m*((n/2)-1):m*(n/2)-1] ;
    end
endgenerate
```

```
endmodule

module  ePrefix #(`include "parameter.v")
            (   output  reg [0:2*m-1]   out ,
                input       [0:2*m-1]   in  );

    always @(*) case(func)
                    2'b00: out = {in[0:m-1], (in[0:m-1] + in[m:2*m-1])} ;
                    2'b01: out = {in[0:m-1], (in[0:m-1] & in[m:2*m-1])} ;
                    2'b10: out = {in[0:m-1], (in[0:m-1] | in[m:2*m-1])} ;
                    2'b11: out = {in[0:m-1], (in[0:m-1] ^ in[m:2*m-1])} ;
                endcase
endmodule
```

An application: FIRST circuit which receives

```
/*********************************************************************
File name:      parameter.v
Circuit name:   no circuit
Description:    defines the parameters FIRST network
*********************************************************************/
    parameter   m = 1,
                n = 16,
                func = 2'b10

    /*  func = 00: add
        func = 01: and
        func = 10: or
        func = 11: xor  */
```

```
/*********************************************************************
File name:      first.v
Circuit name:   FIRST
Description:    defines the  FIRST network
*********************************************************************/
module first #(`include "parameter.v")( output  [n-1:0] out ,
                                        input   [n-1:0] in  );
    wire    [n-1:0] pxOut   ;

    prefix prefix(  .out(pxOut),
                    .in (in));

    assign  out = pxOut & ~(pxOut >> 1) ;
endmodule
```

6.2 Memory Circuits: First-order Digital Systems

6.2.1 Random-Access Memories

Asynchronous RAM are "out of fashion" because of the timing difficulties. Instead, we use synchronous RAMs which solve the timing issues at the IP level.

SRAM (synchronous RAM)

```
/*************************************************************************
File name:      defines.v
Circuit name:   no circuit
Description:    defines the parameters for a 2^n m-bits words SRAM
*************************************************************************/
    `define n 8             // address size
    `define m 8             // data input/output size
```

```
/*************************************************************************
File name:      SRAM.v
Circuit name:   Synchronous RAM
Description:    defines a 2^n m-bits words Synchronous RAM
*************************************************************************/
`include "defines.v"

module SRAM(    input   [`m-1:0]    din ,
                input   [`n-1:0]    addr,
                output  [`m-1:0]    dout,
                input               we  ,
                input               clk );

    reg [`m-1:0]  mem[0:(1<<`n)-1];

    always  @(posedge clk)  if (we) mem[addr] <= din    ;

    assign  dout = mem[addr]    ;
endmodule
```

PSRAM (pipelined SRAM)

```
/*************************************************************************
File name:      defines.v
Circuit name:   no circuit
Description:    defines the parameters for a 2^n m-bits words PSRAM
*************************************************************************/
    `define n 8             // address size
    `define m 8             // data input/output size
```

```
/*****************************************************************
File name:      PSRAM.v
Circuit name:   First version of a pipelined synchronous RAM
Description:    defines a 2^n m-bits words Pipelined synchronous RAM
*****************************************************************/
`include "defines.v"

module PSRAMV1( input        [`m-1:0]    din ,
                input        [`n-1:0]    addr,
                output reg   [`m-1:0]    dout,
                input                    we  ,
                input                    clk );

    reg [`m-1:0]  mem[0:(1<<`n)-1];

    always @(posedge clk) if (we)   dout <= din         ;
                          else      dout <= mem[addr]   ;

    always @(posedge clk) if (we )  mem[addr] <= din;
endmodule
```

```
/*****************************************************************
File name:      PSRAM.v
Circuit name:   Second version of a pipelined synchronous RAM
Description:    defines a 2^n m-bits words Pipelined synchronous RAM
*****************************************************************/
`include "defines.v"

module PSRAMV2( input        [`m-1:0]    din ,
                input        [`n-1:0]    addr,
                output reg   [`m-1:0]    dout,
                input                    we  ,
                input                    clk );

    reg [`m-1:0]  mem[0:(1<<`n)-1];

    always @(posedge clk)   dout <= mem[addr]   ;

    always @(posedge clk) if (we )  mem[addr] <= din;
endmodule
```

BRAM (Block SRAM in FPGA)

The BRAM is a dual-port RAM module which can be instantiated into the FPGA to provide on-chip memory for a relatively large set of data. Two types of BRAM memories are available: of 18k or 36k bits. The dual-port implementation of these BRAMs allows for same-clock-cycle access to different locations.

6.2.2 Register Files

```
/*************************************************************************
File name:      defines.v
Circuit name:   no circuit
Description:    defines the parameters for a 32 32-bits register file
*************************************************************************/
    `define n 5     // address size
    `define m 32    // data input/output size
```

```
/*************************************************************************
File name:      registerFile.v
Circuit name:   Register file
Description:    Two-output port register file
*************************************************************************/
module registerFile(output  [`m-1:0]    left_operand    ,
                    output  [`m-1:0]    right_operand   ,
                    input   [`m-1:0]    result          ,
                    input   [`n-1:0]    left_addr       ,
                    input   [`n-1:0]    right_addr      ,
                    input   [`n-1:0]    dest_addr       ,
                    input               write_enable    ,
                    input               clock           );

    reg [`m-1:0]  file[0:(1<<`n)-1];

    assign  left_operand    = file[left_addr]   ,
            right_operand   = file[right_addr]  ;

    always @(posedge clock) if (write_enable) file[dest_addr] <= result;
endmodule
```

6.2.3 Pipelining

REDUCE

```
/*************************************************************************
File name:      defines.v
Circuit name:   no circuit
Description:    defines the parameters for the pipelined version of
                a n m-bit words reduction add
*************************************************************************/
    `define n 8             // number of inputs
    `define m 8             // input size
```

```
/*************************************************************************
File name:      pipelinedRed.v
Circuit name:   pipelined reduction add
Description:    defines the pipelined version of the reduction add circuit
*************************************************************************/
`include "defines.v"

module pipelinedRed #(parameter N = `n)
        (   output  reg[`m-1:0]     out     ,
            input       [N*`m-1:0]  in      ,
            input                   clock   );

    wire    [(N*`m/2)-1:0]  leftIn  ;
    wire    [(N*`m/2)-1:0]  rightIn ;
    wire    [`m-1:0]        lOut    ;
    wire    [`m-1:0]        rOut    ;

    genvar  i   ;
    generate
        if(N == 2)      always @(posedge clock)
                            out <= in[2*`m-1:`m] + in[`m-1:0]   ;
         else   begin   assign leftIn   = in[N*`m-1:N*`m/2];
                        assign rightIn  = in[(N*`m/2)-1:0];
                        reduce #(.N(N/2))   lr(lOut, leftIn, clock  ),
                                            rr(rOut, rightIn, clock );
                        always @(posedge clock)
                            out <= lOut + rOut  ;
                end

    endgenerate
endmodule
```

PERMUTE

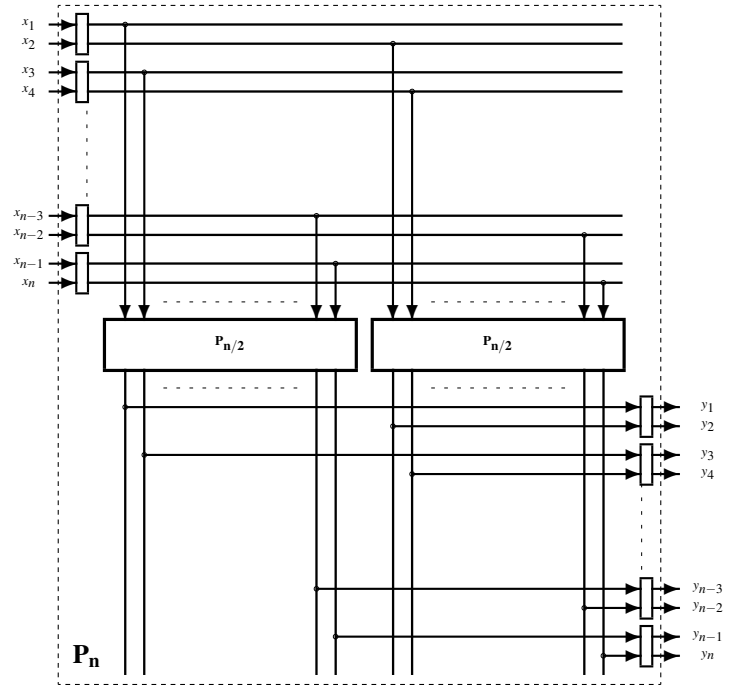

Figure 6.12: The recursive definition of the n-input permutation network, $\mathbf{P_n}$, introduced by Václav E. Beneš.

```
/*****************************************************************
File name:      defines.v
Circuit name:   no circuit
Description:    defines the  parameters
*****************************************************************/
    `define n 8
    `define m 8
    `define d 13 // m+2*log_2(n)-1
```

```
/*****************************************************************
File name:      permute.v
Modules:        permute.v
                ePermute.v
Circuit name:
Description:    defines the  PERMUTE network
*****************************************************************/
`include "defines.v"
```

```
module permute #(parameter N = `n)( output  [0:N*`d-1]  out     ,
                                    input   [0:N*`d-1]  in      ,
                                    input               clock   );


    wire    [`d-1:0]        even[0:N/2-1]   ;
    wire    [`d-1:0]        odd[0:N/2-1]    ;
    wire    [0:(N/2)*`d-1]  leftIn          ;
    wire    [0:(N/2)*`d-1]  rightIn         ;
    wire    [0:(N/2)*`d-1]  leftOut         ;
    wire    [0:(N/2)*`d-1]  rightOut        ;

    genvar i    ;
    generate
     if (N == 2) ePermute eP(   .out    (out[0:2*`d-1]  ),
                                .in     (in[0:2*`d-1]   ),
                                .clock  (clock          ));
      else
        begin
            for(i=0; i<N/2; i=i+1)  begin: ePin
                ePermute ePe(.out   ({even[i], odd[i]}      ),
                             .in    (in[2*i*`d:2*(i+1)*`d-1]),
                             .clock (clock                  ));
            end
            for(i=0; i<N/2; i=i+1)  begin
                assign leftIn[i*`d:(i+1)*`d-1] = even[i]    ;
                assign rightIn[i*`d:(i+1)*`d-1] = odd[i]    ;
            end
            permute #(.N(N/2))  permL(  .out    (leftOut    ),
                                        .in     (leftIn     ),
                                        .clock  (clock      )),
                                permR(  .out    (rightOut   ),
                                        .in     (rightIn    ),
                                        .clock  (clock      ));
            for(i=0; i<N/2; i=i+1)  begin: ePout
                ePermute ePe(.out   (  {out[2*i*`d:(2*i+1)*`d-1],
                                        out[(2*i+1)*`d:2*(i+1)*`d-1]}),
                             .in    (  {leftOut[i*`d:(i+1)*`d-1],
                                        rightOut[i*`d:(i+1)*`d-1]}   ),
                             .clock (clock                          ));
            end
        end
    endgenerate
endmodule

module ePermute
        (   output  reg [0:2*`d-1]  out     ,
            input       [0:2*`d-1]  in      ,
            input                   clock   );

    wire    [`m-1:0]    data0;
    wire    [`m-1:0]    data1;
```

```
    wire    [`d-`m-1:0] dest0;
    wire    [`d-`m-1:0] dest1;

    assign  {dest0, data0} = in[0:`d-1]     ;
    assign  {dest1, data1} = in[`d:2*`d-1]  ;

    always @(posedge clock)
        case({dest0[0], dest1[0]})
            2'b00: out <= {(dest0 >> 1), data0, (dest1 >> 1), data1};
            2'b01: out <= {(dest0 >> 1), data0, (dest1 >> 1), data1};
            2'b10: out <= {(dest0 >> 1), data1, (dest1 >> 1), data0};
            2'b11: out <= {(dest0 >> 1), data0, (dest1 >> 1), data1};
        endcase
endmodule
```

SYSTOLIC MATRIX-VECTOR MULTIPLIER

When a very intense computational function is requested for an Application Specific Integrated Circuit (ASIC) systolic systems represent an appropriate solution. In a systolic system data are inserted and/or extracted rhythmically in/from a uniform modular structure. H. T. Kung and Charles E. Leiserson published the first paper describing a systolic system in 1978 [Kung '79] (however, the first machine known to use a a systolic approach was the Colossus Mark II in 1944). The following example of systolic system is taken from this paper.

Let us design the circuit which multiplies a band matrix with a vector as follows:

$$\begin{pmatrix} a_{11} & a_{12} & 0 & 0 & 0 & \cdots \\ a_{21} & a_{22} & a_{23} & 0 & 0 & \cdots \\ a_{31} & a_{32} & a_{33} & a_{34} & 0 & \cdots \\ 0 & a_{42} & a_{43} & a_{44} & a_{45} & \cdots \\ 0 & 0 & \ddots & \ddots & \ddots & \ddots \\ 0 & 0 & 0 & \ddots & \ddots & \ddots \end{pmatrix} \times \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ \vdots \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ \vdots \end{pmatrix}$$

The main operation executed for matrix-vector operations is *multiply and accumulate* (MACC):

$$Z = A \times B + C$$

for which a specific combinational module is designed. Interleaving MACCs with memory circuits is provided a structure able to compute and to control the flow of data in the same time. The systolic vector-matrix multiplier is represented in Figure 6.13.

The systolic module is represented in Figure 6.13a, where a combinational multiplier ($M = A \times B$) is serially connected with an combinational adder ($M + C$). The result of MACC operation is latched in the output latch which latches besides the result of the computation, the two input value A and B. The latch is transparent on the high level of the clock. It is used to buffer intermediary results and to control the data propagation through the system.

The system is configured using pairs of modules to generate a master-slave structures, where one module receives `ck` and another `ck'`. The resulting structure is a non-transparent one ready to be used in a pipelined connection.

For a band matrix having the width 4, two non-transparent structures are used (see Figure 6.13c). Data is inserted in each phase of the clock (correlate data insertion with the phase of clock represented in Figure 6.13b) as follows:

The result of the computation is generated sequentially to the output y_i of the circuit from Figure 6.13c, as follows:

$$y_1 = a_{11}x_1 + a_{12}x_2$$
$$y_2 = a_{21}x_1 + a_{22}x_2 + a_{23}x_3$$
$$y_3 = a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + a_{34}x_4$$

$y_4 = a_{42}x_2 + a_{43}x_3 + a_{44}x_4 + a_{45}x_5$
$y_5 = \ldots$
$\ldots$

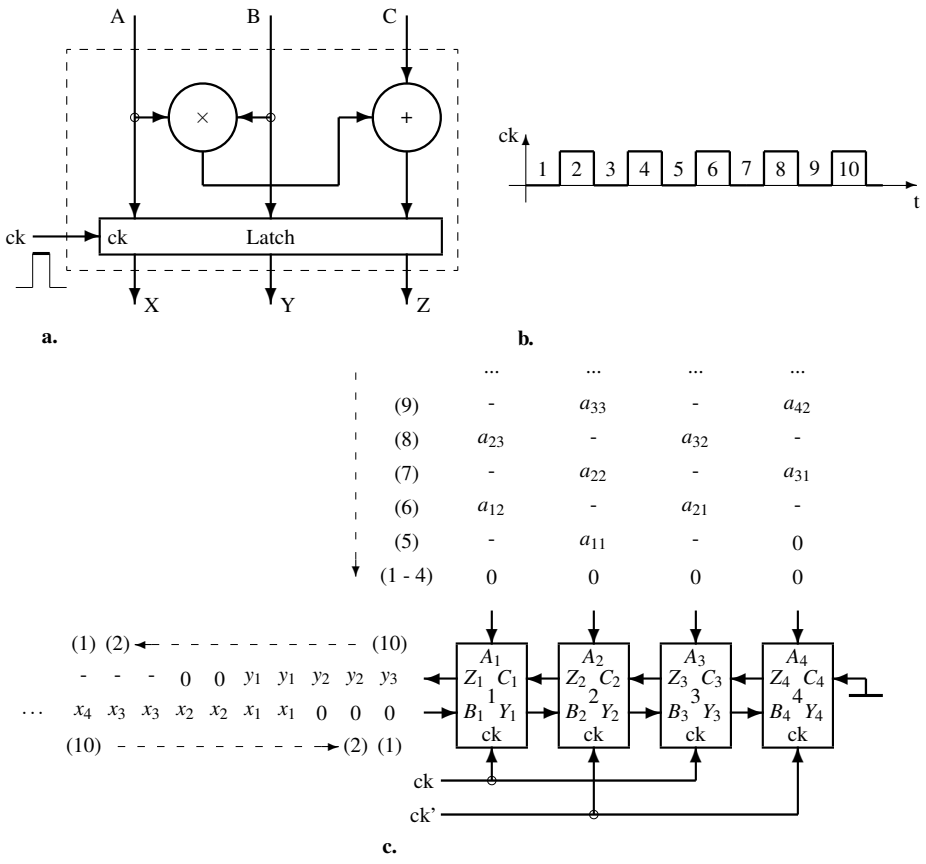

Figure 6.13: **Systolic vector-matrix multiplier. a.** The module. **b.** The clock signal with indexed half periods. **c.** How the modular structure is fed with the data in each half period of the clock signal.

6.3 Automata Circuits: Second-order Digital Systems

6.3.1 Function-oriented automata: the simple automata

RALU (Register file with ALU)

```
/*****************************************************************
File name:        defines.v
Circuit name:     no circuit
Description:      defines the  parameters for RALU
```

```
*****************************************************************/
    `define n 5     // addres size
    `define m 32    // data size
```

```
/*****************************************************************
File name:      RALU.v
Circuit name:   Register file with ALU
Description:    defines RALU
*****************************************************************/
`include "defines.v"

module RALU(    output  [`m-1:0]    left_out    ,
                output  [`m-1:0]    right_out   ,
                output              carryOut    ,
                input               load        ,
                input   [`n-1:0]    left_addr   ,
                input   [`n-1:0]        right_addr  ,
                input   [`n-1:0]        dest_addr   ,
                input               write_enable,
                input   [`m-1:0]    in          ,
                input               carryIn     ,
                input   [2:0]       func        ,
                input               clock       );

    wire    [`m-1:0]    aluOut      ;
    wire    [`m-1:0]    muxOut      ;

    registerFile rf(.left_operand    (left_out      ),
                    .right_operand   (right_out     ),
                    .result          (aluOut        ),
                    .left_addr       (left_addr     ),
                    .right_addr      (right_addr    ),
                    .dest_addr       (dest_addr     ),
                    .write_enable    (write_enable  ),
                    .clock           (clock         ));

    mux inMux(  .out(muxOut),
                .in0(left_out),
                .in1(in),
                .sel(load));

    ALU alu(.carryIn    (carryIn    ),
            .func       (func       ),
            .left       (muxOut     ),
            .right      (right_out  ),
            .carryOut   (carryOut   ),
            .out        (aluOut     ));
endmodule

module registerFile(output  [`m-1:0]    left_operand    ,
```

```
                    output  [`m-1:0]    right_operand    ,
                    input   [`m-1:0]    result           ,
                    input   [`n-1:0]    left_addr        ,
                    input   [`n-1:0]    right_addr       ,
                    input   [`n-1:0]    dest_addr        ,
                    input               write_enable     ,
                    input               clock            );

    reg [`m-1:0]  file[0:(1<<`n)-1];

    assign  left_operand    = file[left_addr]   ,
            right_operand   = file[right_addr]  ;

    always @(posedge clock) if (write_enable) file[dest_addr] <= result;
endmodule

module mux(output   [`m-1:0]    out,
           input    [`m-1:0]    in0,
           input    [`m-1:0]    in1,
           input                sel);

    assign out = sel ? in1 : in0;

endmodule

module  ALU (   input                   carryIn ,
                input       [2:0]       func    ,
                input       [`m-1:0]    left    ,
                input       [`m-1:0]    right   ,
                output  reg             carryOut,
                output  reg [`m-1:0]    out     );

    always @(*)
     case(func)
        3'b000: {carryOut, out} = left + right + carryIn                ;
        3'b001: {carryOut, out} = left - right - carryIn                ;
        3'b010: {carryOut, out} = {1'b0, left & right   }               ;
        3'b011: {carryOut, out} = {1'b0, left | right   }               ;
        3'b100: {carryOut, out} = {1'b0, left ^ right   }               ;
        3'b101: {carryOut, out} = {1'b0, left           }               ;
        3'b110: {carryOut, out} = {left[0], left  >> 1  }               ;
        3'b111: {carryOut, out} = {left[0], left[`m-1], left[`m-1:1]}   ;
        default:{carryOut, out} = {1'b0, left           }               ;
     endcase
endmodule
```

Informational structure & information are differentiated the content of the register file.

Definition 6.1 *The informational structure is a structure with a syntactic order.*

◇

Example 6.1 *The content of the register file in a RALU is an informational structure, because it is a Boolean matrix with each line interpreted as an operand for ALU. In the context of the system containing RALU the content of the*

register file has no any functional meaning. The system acts on the content of the register file, but the content does not conttribute to the functionality of the system.

◇

Definition 6.2 *Information is an informational structure which* **acts** *according to ita meaning.*

◇

DSP (Digital Signal Processing module in FPGA)

The synthesis of RALU for $n = 5$ and $m = 32$ provided a use of 194 LUTs. And let's keep in mind that we only had a small number of functions and among them there was no multiplication. If in our application are involved hundreds of RALU the physical resources of a medium size FPGA are exhausted. Fortunately, the FPGA producers provide specialized block implemented in ASIC technology. One of these is DSP48E1 deployed in large numbers by Xilinx on its FPGAs. In Figure 6.14, the main resources of the DSP module are represented. For details consult [Xilinx '18]. The code used to instantiate the DSP module is in Appendix B.

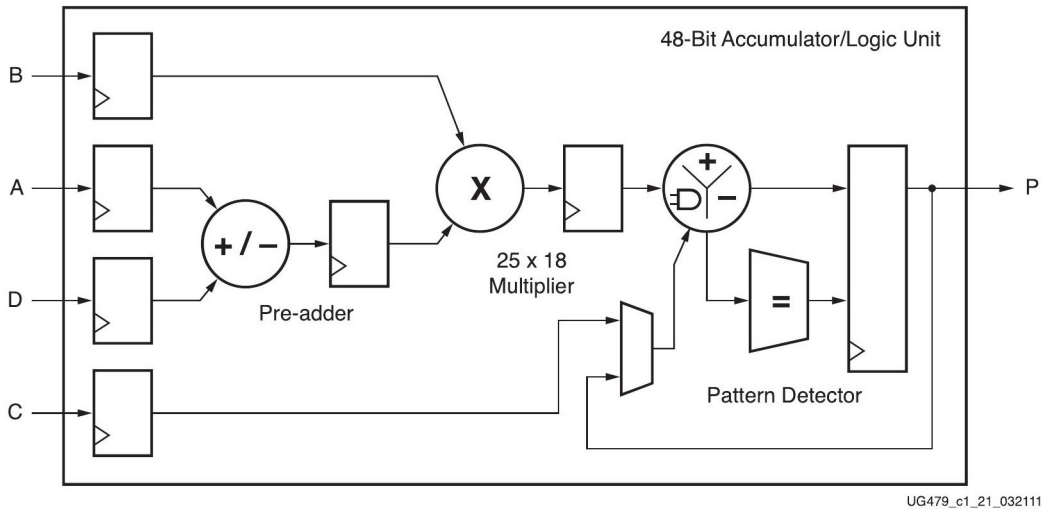

Figure 6.14: **Basic DSP48E1 Slice Functionality** [Xilinx '18].

6.3.2 Finite automata: the complex automata

Finite automata, FA, are automata with a constant number of states whatever long is the input or output sequence of symbols belonging to a finite set. Constant sequence means the sequence's length is not a function of the state set size. Therefore, "finite automata" does't means that we differentiate them from "infinite automata".

FA are complex automata, i.e., their description has a size depending on the state set size.

There are three main categories of FA:

- generator automata
- recognizer automata
- control automata

Generators

A FA can be used to generate sequences belonging to regular (type 3) languages.

Example 6.2 *Let be the language $L_1 = \{0^n 1^m | n,m > 0\}$. The circuit which generate any sequence belonging to L_1, specified by the pair (m,n), is defined by a Verilog module with the following connections list:*

```
/*********************************************************************
FA's output set is:
    2'b00: interpreted as 0
    2'b01: interpreted as 1
    2'b1x: interpreted as e, the empty symbol
Values m and n are limited to 1023.
*********************************************************************/
module genFA(   output  [1:0]   out         ,
                input   [9:0]   mSize, nSize,
                input           reset, clock);
// here comes the description
endmodule
```

The actual structure consists of two presetable down counters serially connected with a FA (the overall system remains in the 2-OS category).

◇

Recognizers

A FA can be used to recognize sequences belonging to regular (type 3) languages.

Example 6.3 *Let be the language $L_2 = \{a^n bc^m | n,m > 0\}$. The circuit which generate any sequence belonging to L is defined by a Verilog module with the following connections list:*

```
/*********************************************************************
FA's input set is:
    2'b00: interpreted as a
    2'b01: interpreted as b
    2'b10: interpreted as c
    2'b11: interpreted as e, the empty symbol
FA's output set is:
    2'b00: interpreted as idle
    2'b01: interpreted as running
    2'b10: interpreted as the string belongs to L2
    2'b11: interpreted as the string does no belongs to L2
*********************************************************************/
module genFA(   output  [1:0]   out         ,
                input   [1:0]   in          ,
                input           reset, clock);
// here comes the description
endmodule
```

No matter how long the input sequence (even "infinite") it is, it is recognized with this automaton with a finite number of states.

◇

Control automata

We must make the distinction between *command automata* and *control automata*. The last are used to introduce a new loop in a system: the control loop which generate commands according to the evolutionn in the commanded sub-system.

Definition 6.3 *Control Automaton, CA, is an initial FA defined by:*

$$CA = (S, X, Y, S_0, f, g, s_0)$$

where:

- S*: the state set*
- $X = \{F, S_0\}$*: the input set structured as a Cartesian product of*
  - $F = \langle f_0, f_1, \ldots, f_{p-1} \rangle$*: the set of p-component Boolean vectors*
  - S_0*: the set of initial states*
- Y*: the set of outputs*
- $f : S \times X \to S$*: the state transition function which, in each state, takes into account only one component of the vector F*
- $g : S \times X \to Y$*: the output transition function which, in each state, takes into account only one component of the vector F*
- $s_0 \in S_0$*: the initial state.*

*The reset signal leads the automaton in the state s_0. The automaton generates sequences of commands which evolve in the context offered by the **flags** represented in the Boolean vector F.*

◇

Example 6.4 *Working with a CA we will make the following remark:* the most part of the sequence generated is organized in a linear sequence. *Therefore, the commands associated to the linear sequences can be stored in ROM at the successive addresses, i.e., the next address for a Read-Only Memory, ROM, can be obtained incrementing the current address stored in the register R.*

Results the structure represented in Figure 6.15. What is specific for this structure is an increment circuit connected to the output of the state register and a small combinational circuit that trans-codes the bits S_1 and S_0. There are the following transition modes coded by M_1, M_0:

- `inc`, codded by $S_1S_0 = 11$: the next address for ROM results by incrementing the current value of R_n, i.e., the address to the ROM's input
- `jmp`, codded by $S_1, S_0 = 10$: the next address for ROM is given by the content of the field `jmp` from the output of ROM
- `cjmp`, codded by $S_1, S_0 = 1T'$: **if** the value of the selected flag, T (the output of MUXT), is 1, **then** the next address for ROM is given by the content of the one field `jmp` from the output of ROM, **else** the next address for ROM results by incrementing the current address
- `nextOp`, codded by $S_1, S_0 = 01$: the next address for ROM is selected by $nMUX_4$ from the initialization input `op`
- `reset`, codded by $S_1, S_0 = 00$: the next address for ROM is selected by $nMUX_4$ from the `init` input

The only complex circuit in the CROM is the trans-coder TC. The overall complexity of the system is given by the content of ROM.

The output of ROM can be seen as a *microinstruction* defined as follows:

```
<microinstruction>::= <setLabel> <Command> <Mod> <Test> <useLabel>;
<comm>::= <to be defined when use>;
<mod>::=  jmp | cjmp | init | inc ;
<test>::= <to be defined when use>;
<setLabel>::= setLabel(<number>);
<useLabel>::= useLabel(<number>);
<number>::= 0 | 1 | ... | 9 | <number><number>;
```

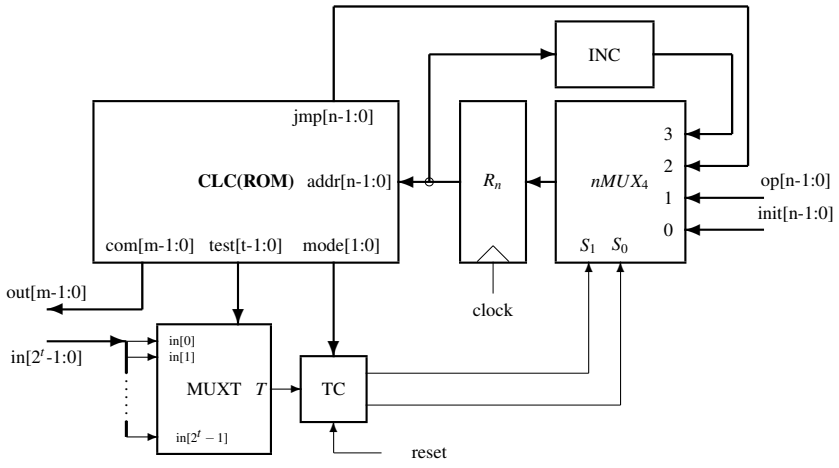

Figure 6.15: **The simplest Controller with ROM (CROM).** The Moore form of control automaton is optimized using an incremented circuit (INC) to compute the most frequent next address for ROM.

Let us call CROM this version of CA (**C**ontroller with **ROM**).

◇

6.4 Processing Circuits: Third-order Digital Systems

6.4.1 Counter extended automata (CEA)

Let us try to solve the problem of recognizing the sequence of symbols of form 0^n1^n for $n > 0$ using a finite automaton. It is possible, but we must design for each n an automaton with a number of states in $O(n)$. A simpler, efficient solution is to use a counter extended automaton, CEA.

Definition 6.4 *A CEA is made of a* small & complex *finite initial automaton, FA, and an up/down counter, UDC, which is a* big & simple *automaton. FA issues in each clock cycle a command belonging to the set* {`reset, nop, up, down`} *to UDC, and receives back the predicate* `zero` *which is 0 if the value of counter is different form zero, and is 1 if the counter is zero (see Figure 6.16). Formally:*

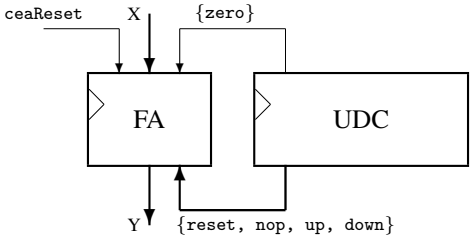

Figure 6.16: Counter Extended Automaton (CEA): a Finite Automaton (FA) loop coupled with an Up-Down Counter (UDC).

$$CEA = (X, Y, S, f, S_0)$$

$$FA = (X \times \{zero\}), (Y, \{reset, nop, up, down\}), Q, g, Q_0)$$
$$UDC = (\{reset, nop, up, down\}, \{zero\}, N, h, N_0)$$

where:

X *is the input set of CEA*

Y *is the output set of CEA*

S = $(Q \times N)$ *is the set of states of CEA*

$S_0 = \{Q_0, N_0\}$ *is the initial state of CEA*

Q *is the set of state of FA*

$Q_0 \in Q$ *is the initial state of FA*

N *is the set of states of UDC*

$N_0 = 0$ *is the initial state of UDC*

$h : (N \times \{reset, nop, up, down\}) \rightarrow (N \times \{zero\})$ *is the transition of UDC*

$g : (Q \times \{X \times \{zero\}) \rightarrow (Q \times Y \times \{reset, nop, up, down\})$ *is the transition of FA*

f *is the state transition of of CEA*

◇

Example 6.5 *The system used to recognize the language* $L = \{0^n 1^n | n > 0\}$ *is a CEA whose inital FA is described by the following pseudo-code:*

```
/*************************************************************************
FA's list of states:
    init:     inital state
    recZero:  state receiving zeroes
    recOne:   state receiving ones
    finalNo:  final state when the sequence does not belongs to L
    finalYes: final state when the sequence belongs to L

State register: state

Input set: {in, {zero}} = {{e,0,1}, {zero}}
*************************************************************************/
    if (ceaReset)                    {state, com} <= {init, reset}    ;
     else    case(state)
                 init:       case(in)
                                 e:  {state, com} <= {state, nop}     ;
                                 0:  {state, com} <= {recZero, up}    ;
                                 1:  {state, com} <= {finalNo, nop}   ;
                             endcase
                 recZero:    case(in)
                                 e:  {state, com} <= {finalNo, nop}   ;
                                 0:  {state, com} <= {state, up}      ;
                                 1:  {state, com} <= {recOne, down}   ;
                             endcase
                 recOne:     case(in)
                                 e:  {state, com} <= zero ?
                                                     {finalYes, nop} :
                                                     {finalNo, nop}   ;
                                 0:  {state, com} <= {finalNo, nop}   ;
```

```
                        1:  {state, com} <= {state, down}    ;
                    endcase
        finalNo:            {state, com} <= {state, nop}     ;
        finalYes:           {state, com} <= {state, nop}     ;
    endcase
```

◇

6.4.2 Push-down automata

The first example of loop coupled automata uses a finite automaton and a functional automaton: the stack (LIFO memory). A finite complex structure is interconnected with an "infinite" but simple structure. The simple and the complex are thus perfectly segregated. This approach has the role of minimizing the size of the random part. More, this loop affects the *magnitude order* of the randomness, instead of the previous examples (*Arithmetic & Logic Automaton*) in which the size of randomness is reduced only by a constant. The proposed structure is a well known system having many theoretical and practical applications: the *push-down automaton*.

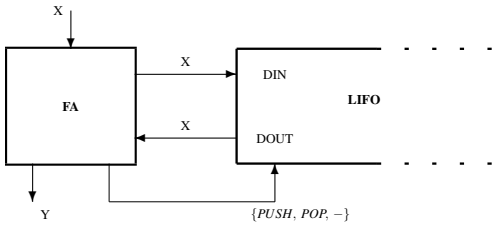

Figure 6.17: **The push-down automaton (*PDA*).** A finite (random) automaton loop-coupled with an "infinite" stack (a simple automaton) is an enhanced toll for dealing with formal languages.

Definition 6.5 *The* push-down automaton*, PDA, (see Figure 6.17) built by a finite automaton loop connected with a push-down stack (LIFO), is defined by the six-tuple:*

$$PDA = (X \times X', Y \times Y' \times X, Q, f, g, z_0)$$

where:

X *: is the finite alphabet of the machine; the input string is in* X^*

X' *: is the finite alphabet of the stack,* $X' = X' \cup \{z_0\}$

Y *: is the finite output set of the machine*

Y' *: is the set of commands issued by the finite automaton toward LIFO,* $\{PUSH, POP, -\}$

Q *: is the finite set of the automaton states (i.e.,* $|Q| \neq h(max\ l(s))$*, where* $s \in X^*$ *is received on the input of the machine)*

f *: is the state transition function of the* machine

$$f : X \times X' \times Q \rightarrow Q \times X \times Y'$$

(i.e., depending on the received symbol, by the value of the top of stack (TOS) and by the automaton's state, the automaton switches in a new state, a new value can be sent to the stack and the stack receives a new command (PUSH, POP or NOP))

g *: is the output transition function of the* automaton - $g : Q \rightarrow Y$

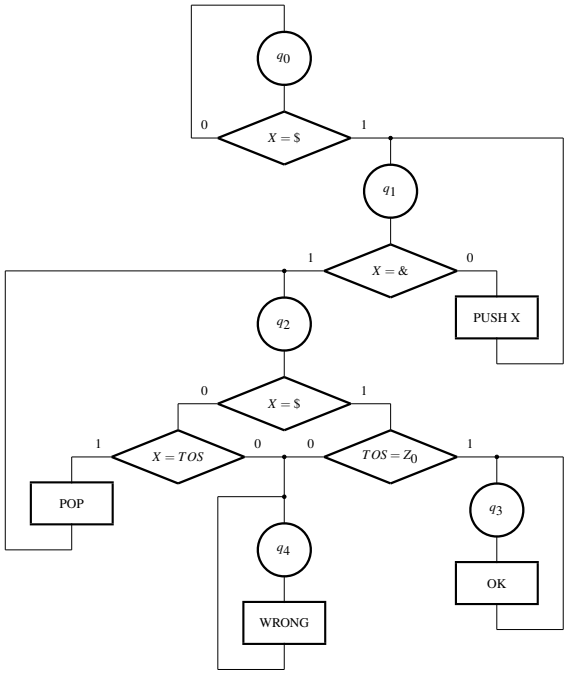

Figure 6.18: **Defining the behavior of a PDA.** The algorithm detecting the antisymmetrical sequences of symbols.

z_0 *: is the initial value of TOS.* $\diamond$

Example 6.6 *The problem to be solved is designing a machine that recognizes strings having the form* $\$x\&y\$$*, where* $\$, \& \in X$ *and* $x, y \in X^*$*, X being a finite alphabet and y is the antisymmetric version of x.*

The solution is to use a PDA with f and g described by the flow-chart given in Figure 6.18. Results a five state, initial (in q_0*) automaton, each state having the following meaning and role:*

q_0 *: is the initial state in which the machine is waiting for the first* \$

q_1 *: in this state the received symbols are pushed into the stack, excepting* & *that switches the automaton in the next state*

q_2 *: in this state, each received symbol is compared with TOS, that is poped on, while the received symbol is not* \$*; when the input is* \$ *and* $TOS = z_0$ *the automaton switches in* q_3*, else, if the received symbols do not correspond with the successive value of the TOS or the final value of TOS differs from* z_0*, the automaton switches in* q_4

q_3 *: if the automaton is in this state the received string was recognized as a well formed string*

q_4 *: if the automaton is in this state the received string was wrong.* $\diamond$

The reader can try to solve the problem using only an automaton. For a given X set, especially for a small set, the solution is possible and small, but the LOOP PLA of the resulting automaton will be a circuit with the size and the form depending by the dimension and by the content of the set X. If only one symbol is added or at least is changed, then the entire design process must be restarted from scratch. The automaton imposes a solution in which the simple, recursive part of the solution is mixed up with the random part, thus all the system has a very large apparent complexity. The automaton must store in the state space what PDA stores in stack. You imagine how huge become the state set in a such crazy solution. Both, the size and the complexity of the solution become unacceptable.

The solution with PDA, just presented, does not depend by the content and by the dimension of the set X. In this solution the simple is well segregated from the complex. The simple part is the "infinite" stack and the complex part is a small, five-state finite automaton.

6.4.3 Processor

Processor & information: generic definition

Definition 6.6 *The generic structure of a processor is defined by the pair RALU + CROM loop connected. CROM issues in each clock cycle a command for RALU and computes its next state according to its inputs: the flags generated selected from F or the operation received on* `op`*. The generic functionality of the processor is defined by the Instruction Set Architecture, ISA, specified by the set S_O of the initial states of CROM. Each element of ISA is associated with a sequence of states generated by CROM.*

◇

There are two informational structure associated to the processor:

- the content of the register file (which is a RAM): the data on which the operations controlled by CROM are applied
- the content of ROM: a set of sequence of micro-instructions defining ISA

Both are arrays of bits. The main difference between these to arrays is given by the fact that *the content of ROM defines the action on the content of RAM*. ROM's content is active, while RAM's content is passive. At the level of processor only the ROM's content has a well defined meaning, while the content of RAM has no meaning for the processor's functionality. Therefore, we will consider RAM's content as ***data***, while the ROM's content will be considered as ***information***.

Elementary processor

Definition 6.7 *The elementary processor, EP, is a processor with $S_0 = \{s_0\}$.*

◇

EP is a functional structure controlled by a strict initial automaton (see Example 4.5).

CISC vs. RISC

The way the CROM, as *control unit*, in a processor is implemented classifies the processors in two main categories.

Definition 6.8 *A Complex Instruction Set Computer, CISC, processor associates for at least one element in ISA a sequence of state in CROM, while a Reduced Instruction Set Computer processor associates for each element in ISA only one state in CROM, i.e., a CISC processor* ***interprets*** *ISA, while a RISC processor* ***executes*** *ISA.*

◇

Accumulator-based processor

The way RALU is implemented, as the *execution unit* of a processor, allows us to design few versions of processors.

Definition 6.9 *An* accumulator-based *processor is equipped with an execution unit which use a special register, named* **accumulator***, as the left operand and destination for arithmetic and logic operations.*

◇

A RISC version of an accumulator-based processor is considered in the following example.

Example 6.7 **ISA** *of an accumulator-based processor is defined in the folder* `0_definitions.v`.

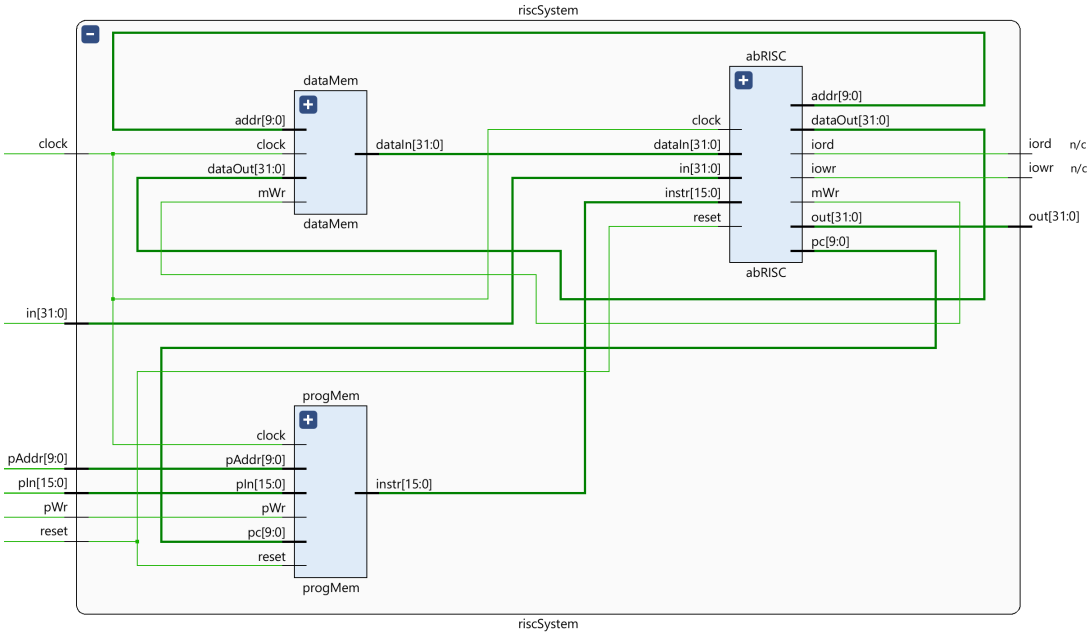

Figure 6.19: System.

```
/*****************************************************************************
File name: 0_definitions.v

        Accumulator-Based Processor's Instruction Set Architecture

                                    // REGISTERS
    reg [15:0]  instr       ; // instruction register
    reg [31:0]  acc         ; // accumultaor
    reg [7:0]   addr        ; // address register in register file
    reg [31:0]  rf[0:255]   ; // register file
    reg [15:0]  pc          ; // program counter
    reg [15:0]  maddr       ; // memory address register
    reg [31:0]  min         ; // data in register
    reg [31:0]  mout        ; // data out register
    reg [31:0]  in          ; // input register
    reg [31:0]  out         ; // output register
    reg         int         ; // interrupt input

                                    // OUTPUTS
    reg         inta        ; // interrupt acknowledge
    reg         readCom     ; // read command to memory
    reg         writeCom    ; // write commmand to memory
    reg         sendCom     ; // read command to input-output
    reg         getCom      ; // write commmand to input-output
```

```
                                // INPUTS
    reg         int         ; // interrupt input
    reg         ioReady     ; // input data is valid
    reg         memready    ; // memory output is valid

    instr[15:0] = {operandSel[2:0], operation[4:0], value[7:0]}
*****************************************************************************/

                                // SECOND OPERAND SELECTION
    `define imm     3'b000    // op = {{24{val[7]}}, val}
    `define dir     3'b001    // op = rf[{2'b0, val}]
    `define rel     3'b010    // op = rf[addr + {{2{val[7]}}, val}]
    `define ire     3'b011    // op = rf[addr + {{2{val[7]}}, val}]
                              // addr <= {{2{val[7]}}, val}
    `define mem     3'b100    // op = min
    `define ext     3'b101    // op = in
    //`define   sck     3'b110   // op = utos in stack mode
    //`define ctl       3'b111   // no right operand operation

                                // CONTROL INSTRUTIONS
    `define jmp     5'b11000 // relative jump: pc <= pc + val
    `define ajmp    5'b11001 // absolute jump: pc <= acc
    `define brz     5'b11010 // pc <= acc == 0 ? pc + val : pc + 1
    `define brnz    5'b11011 // pc <= acc != 0 ? pc + val : pc + 1
    `define ie      5'b11100 // interrupt desable
    `define de      5'b11101 // interrupt enable

                                // ARITHMETIC & LOGIC INSTRUCTIONS
    `define add     5'b00000 // {cr, acc} <= acc + op
    `define addc    5'b00001 // {cr, acc} <= acc + op + cr
    `define sub     5'b00010 // {cr, acc} <= acc - op
    `define rsub    5'b00011 // {cr, acc} <= op - acc
    `define subc    5'b00100 // {cr, acc} <= acc - op -cr
    `define rsubc   5'b00101 // {cr, acc} <= op - acc - cr
    `define mult    5'b00110 // {cr, acc} <= {1'b0, acc[15:0] * op[15:0]}
    `define shiftc  5'b00111 // {cr, acc} <= {acc[0], cr, acc[31:1]}
    `define shift   5'b01000 // {cr, acc} <= {acc[0], 1'b0, acc[31:1]}
    `define ashift  5'b01001 // {cr, acc} <= {acc[0], acc[31], acc[31:1]}
    `define bwand   5'b01010 // acc <= acc & op
    `define bwor    5'b01011 // acc <= acc | op
    `define bwxor   5'b01100 // acc <= acc ^ op

                                // DATA MOVE INSTRUCTIONS
    `define load    5'b10000 // acc <= op
    `define store   5'b10001 // op <=  acc, when apply
    `define iosend  5'b10010 // out <= acc
    `define ioget   5'b10011 // acc <= in
    `define write   5'b10100 // maddr <= acc; mout <= op
    `define read    5'b10101 // maddr <= acc
    `define addr    5'b10110 // addr <= val
    `define insert  5'b10111 // acc <= {acc << 8, val}
```

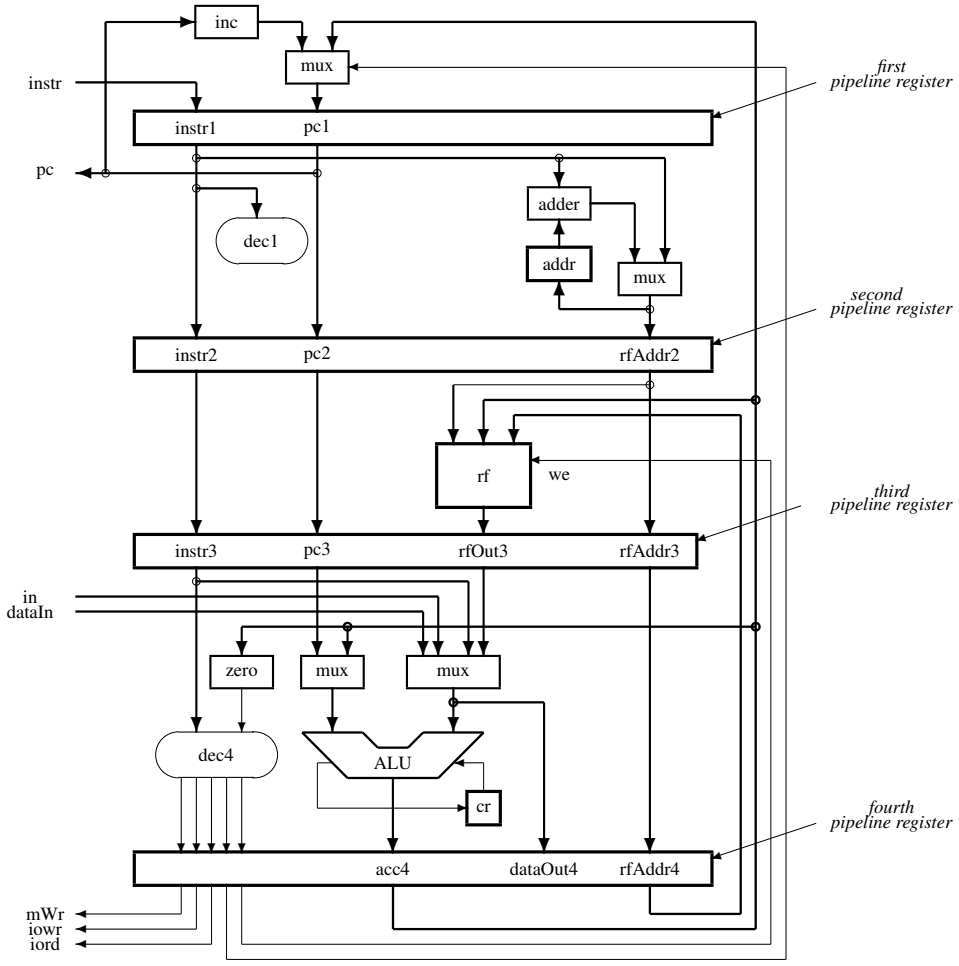

Figure 6.20: The four pipeline stages of asRISC processor.

```
/*************************************************************************
File name:   abRISC.v
Circuit name:
Description:
*************************************************************************/
    `include "0_definitions.v"
module abRISC(  input           reset   ,
                input           clock   ,
                input   [9:0]   pAddr   ,
                input   [15:0]  pIn     ,
                input           pWr     ,
                input   [31:0]  in      ,
                output          iord    ,
```

```
            output  [31:0]  out      ,
            output           iowr     );

    wire    [15:0]  instr1   ;
    wire    [9:0]   pc1      ;

    wire    [15:0]  instr2   ;
    wire    [9:0]   pc2      ;
    wire    [7:0]   rfAddr2  ;

    wire    [15:0]  instr3   ;
    wire    [9:0]   pc3      ;
    wire    [7:0]   rfAddr3  ;
    wire    [31:0]  rfOut3   ;

    wire            jmp4     ;
    wire    [31:0]  acc4     ;
    wire            we4      ;
    wire            mWr4     ;
    wire    [7:0]   rfAddr4  ;
    wire    [31:0]  dataOut4 ;

    wire    [31:0]  dataIn5  ;

    firstStage  first( pAddr   ,
                       pIn     ,
                       pWr     ,
                       jmp4    ,
                       acc4    ,
                       instr1  ,
                       pc1     ,
                       reset   ,
                       clock   );

    secondStage second( instr1  ,
                        pc1     ,
                        instr2  ,
                        pc2     ,
                        rfAddr2 ,
                        reset   ,
                        clock   );

    thirdStage  third( instr2  ,
                       pc2     ,
                       rfAddr2 ,
                       acc4    ,
                       we4     ,
                       rfAddr4 ,
                       instr3  ,
                       pc3     ,
                       rfAddr3 ,
                       rfOut3  ,
                       reset   ,
```

```
                        clock    );

    fourthStage  fourth( in       ,
                         instr3   ,
                         pc3      ,
                         rfAddr3  ,
                         rfOut3   ,
                         dataIn5  ,

                         jmp4     ,
                         acc4     ,
                         we4      ,
                         mWr4     ,
                         rfAddr4  ,
                         dataOut4 ,
                         iord     ,
                         iowr     ,
                         reset    ,
                         clock    );

    lastStage    last(   acc4     ,
                         dataOut4 ,
                         mWr4     ,
                         dataIn5  ,
                         out      ,
                         clock    );
endmodule
```

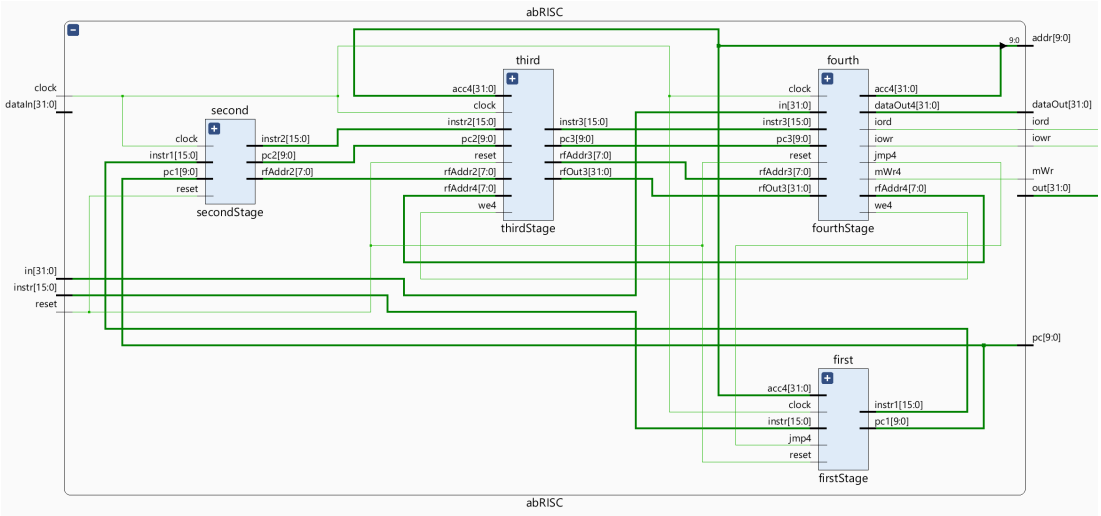

Figure 6.21: Accumulator-based RISC.

```
/*************************************************************************
File name:  firstStage.v
Circuit name:
Description:
*************************************************************************/
module firstStage(  input       [9:0]   pAddr   ,
                    input       [15:0]  pIn     ,
                    input               pWr     ,
                    input               jmp4    ,
                    input       [31:0]  acc4    ,
                    output  reg [15:0]  instr1  ,
                    output  reg [9:0]   pc1     ,
                    input               reset   ,
                    input               clock   );

    reg [15:0]  pMem[0:1023]    ;

    always @(posedge clock)
        if (reset)  begin               instr1      <= 0    ;
                                        pc1         <= 0    ;
                            if (pWr)    pMem[pAddr] <=  pIn ;
                    end
         else       begin   instr1      <= pMem[pc1]                    ;
                            pc1         <= jmp4 ? acc4[9:0] : pc1 + 1   ;
                    end
endmodule
```

```
/*************************************************************************
File name:  secondStage.v
Circuit name:
Description:
*************************************************************************/
    `include "0_definitions.v"

module secondStage( input       [15:0]  instr1  ,
                    input       [9:0]   pc1     ,
                    output  reg [15:0]  instr2  ,
                    output  reg [9:0]   pc2     ,
                    output  reg [7:0]   rfAddr2 ,
                    input               reset   ,
                    input               clock   );

    reg [7:0]   addr;

    always @(posedge clock)
        if (reset)  begin   instr2  <= 0        ;
                            pc2     <= 0        ;
                            rfAddr2 <= 0        ;
                    end
         else       begin   instr2  <= instr1   ;
```

```
                            pc2     <= pc1      ;
                            case(instr1[15:13])
                                `dir: rfAddr2 <= instr1[7:0]        ;
                                `rel: rfAddr2 <= addr + instr1[7:0] ;
                                `ire: rfAddr2 <= addr + instr1[7:0] ;
                            default : rfAddr2 <= rfAddr2            ;
                            endcase
                            addr <= (instr1[15:13] == `ire) ?
                                        addr + instr1[7:0] :
                                    ((instr1[12:8] == `addr)) ?
                                        instr1[7:0] : addr          ;
                        end

endmodule
```

```
/*********************************************************************
File name:  thirdStage.v
Circuit name:
Description:
*********************************************************************/
    `include "0_definitions.v"
module thirdStage(  input       [15:0]  instr2  ,
                    input       [9:0]   pc2     ,
                    input       [7:0]   rfAddr2 ,
                    input       [31:0]  acc4    ,
                    input               we4     ,
                    input       [7:0]   rfAddr4 ,
                    output  reg [15:0]  instr3  ,
                    output  reg [9:0]   pc3     ,
                    output  reg [7:0]   rfAddr3 ,
                    output  reg [31:0]  rfOut3  ,
                    input               reset   ,
                    input               clock   );

    reg [31:0]  rf[0:255]   ;

    always @(posedge clock)
        if (reset)  begin   instr3      <= 0            ;
                            pc3         <= 0            ;
                            rfAddr3     <= 0            ;
                            rfOut3      <= 0            ;
                    end
         else       begin   instr3      <= instr2       ;
                            pc3         <= pc2          ;
                            rfAddr3     <= rfAddr2      ;
                            rfOut3      <= rf[rfAddr2]  ;
                            if (we4) rf[rfAddr4] <= acc4;
                    end
endmodule
```

```
/*************************************************************************
File name:  fourthStage.v
Circuit name:
Description:
*************************************************************************/
    `include "0_definitions.v"
module fourthStage( input       [31:0]  in      ,
                    input       [15:0]  instr3  ,
                    input       [9:0]   pc3     ,
                    input       [7:0]   rfAddr3 ,
                    input       [31:0]  rfOut3  ,
                    input       [31:0]  dataIn  ,
                    output  reg         jmp4    ,
                    output  reg [31:0]  acc4    ,
                    output  reg         we4     ,
                    output  reg         mWr4    ,
                    output  reg [7:0]   rfAddr4 ,
                    output  reg [31:0]  dataOut4,
                    output  reg         iord    ,
                    output  reg         iowr    ,
                    input               reset   ,
                    input               clock   );

    reg         cr  ;

    reg [31:0]  op  ;

    always @(*) case(instr3[15:13])
                    `imm: op = {{24{instr3[7]}}, instr3[7:0]}   ;
                    `dir: op = rfOut3                           ;
                    `rel: op = rfOut3                           ;
                    `ire: op = rfOut3                           ;
                    `mem: op = dataIn                           ;
                    `ext: op = in                               ;
                default: op = 32'b0                             ;
                endcase

    always @(posedge clock)
     if (reset) begin   rfAddr4     <= 0;
                        acc4        <= 0;
                        jmp4        <= 0;
                        we4         <= 0;
                        mWr4        <= 0;
                        cr          <= 0;
                end
      else begin
            rfAddr4     <= rfAddr3  ;
            dataOut4    <= op       ;
            case(instr3[12:8])
                `add    : {cr, acc4} <= {acc4 + op}                 ;
                `addc   : {cr, acc4} <= {acc4 + op + cr}            ;
                `sub    : {cr, acc4} <= {acc4 - op}                 ;
```

```
                `rsub   : {cr, acc4} <= {op - acc4}                    ;
                `subc   : {cr, acc4} <= {acc4 - op - cr}               ;
                `rsubc  : {cr, acc4} <= {op - acc4 - cr}               ;
                `mult   : {cr, acc4} <= {cr, acc4[15:0] * op[15:0]}    ;
                `shiftc : {cr, acc4} <= {acc4[0], cr, acc4[31:1]}      ;
                `shift  : {cr, acc4} <= {acc4[0], 1'b0, acc4[31:1]}    ;
                `ashift : {cr, acc4} <= {acc4[0], acc4[31], acc4[31:1]} ;
                `bwand  : {cr, acc4} <= {cr, acc4 & op}                ;
                `bwor   : {cr, acc4} <= {cr, acc4 | op}                ;
                `bwxor  : {cr, acc4} <= {cr, acc4 ^ op}                ;
                `load   : {cr, acc4} <= {cr, op}                       ;
                `read   : {cr, acc4} <= {cr, op}                       ;
                `ioget  : {cr, acc4} <= {cr, op}                       ;
                `insert : {cr, acc4} <= {cr, acc4[23:0], op[7:0]}      ;
                `jmp    : {cr, acc4} <= {cr, (pc3 + op)}               ;
                `ajmp   : {cr, acc4} <= {cr, acc4}                     ;
                `brz    : {cr, acc4} <= {cr, ((acc4 == 0) ?
                                        pc3 + op : pc3 + 1)}           ;
                `brnz   : {cr, acc4} <= {cr, ((acc4 == 0) ?
                                        pc3 + 1 : pc3 + op)}           ;
            default     : {cr, acc4} <= {cr, acc4}                     ;
            endcase
            case(instr3[12:8])
                `jmp : jmp4 <= 1'b1                    ;
                `ajmp: jmp4 <= 1'b1                    ;
                `brz : jmp4 <= (acc4 == 0) ? 1'b1 : 1'b0;
                `brnz: jmp4 <= (acc4 == 0) ? 1'b0 : 1'b1;
            default  : jmp4 <= 1'b0                    ;
            endcase
            we4         <= (instr3[12:8] == `store) ? 1'b1 : 1'b0   ;
            iord        <= (instr3[12:8] == `ioget) ? 1'b1 : 1'b0   ;
            iowr        <= (instr3[12:8] == `iosend) ? 1'b1 : 1'b0  ;
            mWr4        <= (instr3[12:8] == `write) ? 1'b1 : 1'b0   ;
        end
endmodule
```

```
/*****************************************************************
File name:  lastStage.v
Circuit name:
Description:
*****************************************************************/
    `include "0_definitions.v"
module lastStage(   input       [31:0]  acc4    ,
                    input       [31:0]  dataOut4,
                    input               mWr4    ,
                    output  reg [31:0]  dataIn5 ,
                    output  reg [31:0]  out     ,
                    input               clock   );

    reg [31:0]  dMem[0:1023]    ;
```

```
    always @(posedge clock)
        begin                   dataIn5          <= dMem[acc4[9:0]]  ;
                                out              <= acc4             ;
                if (mWr4)       dMem[acc4[9:0]] <= dataOut4         ;
        end
endmodule
```

◇

The simulation environment for abRISC consists of the module `1_simulator` which contains a code generator `2_codeGenerator`

```
/*********************************************************************
File name:  simulator.v
Circuit name:
Description:
*********************************************************************/
    `include "0_definitions.v"

module simulator;

    reg             reset   ;
    reg             clock   ;
    reg     [9:0]   pAddr   ;
    reg     [15:0]  pIn     ;
    reg             pWr     ;
    reg     [31:0]  in      ;
    wire            iord    ;
    wire    [31:0]  out     ;
    wire            iowr    ;

    initial begin               clock = 0       ;
                forever #1      clock = ~clock  ;
        end

    riscSystem riscSystem(  reset   ,
                            clock   ,
                            pAddr   ,
                            pIn     ,
                            pWr     ,
                            in      ,
                            iord    ,
                            out     ,
                            iowr    );

    // BINARY CODE GENERATOR
    `include "2_codeGenerator.v"

    integer i;
```

```
    //SIMULATION
    initial begin
        reset   = 1 ;
        pAddr   = 0 ;
        pIn     = 0 ;
        pWr     = 0 ;
        in      = 0 ;
        for (i=0; i<16; i=i+1)
         $display("memory[%0d] \t =%b", i, riscSystem.progMem.pMem[i]);
        #4      reset = 0   ;
        #100    $finish ;
        end

//*      // MONITOR FOR PROGRAM LAOD & CONTROLLER
    initial begin
    $monitor
    ("t=%0d pc=%d  instr=%b acc=%d pc3=%d op=%d rf=[%0d, %0d, %0d,
    %0d, %0d, %0d, %0d]",
        $time,
        riscSystem.pc,
        riscSystem.instr,
        riscSystem.addr,
        riscSystem.abRISC.pc3,
        riscSystem.abRISC.fourth.op,
        riscSystem.abRISC.third.rf[0],
        riscSystem.abRISC.third.rf[1],
        riscSystem.abRISC.third.rf[2],
        riscSystem.abRISC.third.rf[3],
        riscSystem.abRISC.third.rf[4],
        riscSystem.abRISC.third.rf[5],
        riscSystem.abRISC.third.rf[6],
        riscSystem.abRISC.third.rf[7]);
    end
endmodule
```

```
/*********************************************************************
File name:  2_codeGenerator.v
Circuit name:
Description:
*********************************************************************/
    `include "0_definitions.v"
// CODE GENERATOR
    reg [4:0]   opCode          ;
    reg [2:0]   operand         ;
    reg [7:0]   value           ;
    reg [9:0]   addrCounter     ;
    reg [9:0]   labelTab[0:1023];

    task endLine;
```

```
    begin
        riscSystem.progMem.pMem[addrCounter] =
            {operand, opCode, value}     ;
        addrCounter = addrCounter + 1    ;
    end
endtask

// in first pass associates 'counter' with 'labelIndex' in labelTab
task LB ;
    input [7:0] labelIndex;

    labelTab[labelIndex] = addrCounter;
endtask
// uses the content of labelTab in the second pass
task cULB;
    input [7:0] labelIndex;

    value = labelTab[labelIndex] - addrCounter - 2'b10;
endtask

task NOP;
    begin   opCode  = `add  ;
            operand = `imm  ;
            value   = 8'b0  ;
            endLine         ;
    end
endtask

`include "3_cgCONTROL.v"          // control instructions
`include "3_cgADD.v"              // addition
//`include "cgADDC.v"         // addition with carry
//`include "cgSUB.v"          // subtract
//`include "cgSUBC.v"         // subtract with carry
//`include "cgRVSUB.v"        // reverse subtract
//`include "cgRVSUBC.v"       // reverse subtract with carry
//`include "cgMULT.v"         // multiplication
//`include "cgSHIFT.v"        // shift
//`include "cgLOAD.v"         // load accumulator
//`include "cgSTORE.v"        // store accumulator
//`include "cgAND.v"          // bit-wise AND
//`include "cgOR.v"           // bit-wise OR
//`include "cgXOR.v"          // bit-wise XOR

// RUNNING
initial begin   addrCounter = 0;
                `include "00_program.v" // first pass
                addrCounter = 0;
                `include "00_program.v" // second pass
        end
```

```
/*********************************************************************
File name:  3_cgADD.v
Circuit name:
Description:
*********************************************************************/

    `include "0_definitions.v"
/*  imm = 3'b000 immediate value: op = {{24{scalar[7]}}, value}
    dir = 3'b001 absolute: op = mem[value]
    rel = 3'b010 relative: op = mem[addr + value]
    ire = 3'b011 relative & inc: op = mem[addr + value]
                 addr <= addr + value
    mem = 3'b100 op = dataIn
    ext = 3'b101 op = in
*/

    task VADD; // value add: acc <= acc + op
        input [7:0] scalar;

        begin   opCode  = `add  ;
                operand = `imm  ;
                value   = scalar;
                endLine         ;
        end
    endtask

    task ADD; // absolute add: acc[i] <= acc[i] + op
        input [7:0] scalar;

        begin   opCode  = `add  ;
                operand = `dir  ;
                value   = scalar;
                endLine         ;
        end
    endtask

    task RADD; // relative add: acc[i] <= acc[i] + op
        input [7:0] scalar;

        begin   opCode  = `add  ;
                operand = `rel  ;
                value   = scalar;
                endLine         ;
        end
    endtask

    task RIADD; // relative add: acc[i] <= acc[i] + op
                // and increment: addr <= addr + value
        input [7:0] scalar;

        begin   opCode  = `add  ;
                operand = `ire  ;
```

```
                value   = scalar;
                endLine         ;
        end
    endtask

    task MADD;  // memory add: acc[i] <= acc[i] + dataIn

        begin   opCode  = `add  ;
                operand = `mem  ;
                value   = 8'b0  ;
                endLine         ;
        end
    endtask

    task EADD;  // input add: acc[i] <= acc[i] + in

        begin   opCode  = `add  ;
                operand = `ext  ;
                value   = 8'b0  ;
                endLine         ;
        end
    endtask
```

```
/*************************************************************************
File name:  3_cgCONTROL.v
Circuit name:
Description:
*************************************************************************/
    `include "0_definitions.v"

    task JMP;
        input [7:0] label   ;

        begin   opCode  = `jmp  ;
                operand = `imm  ;
                cULB(label)     ;
                endLine         ;
        end
    endtask

    task AJMP;
        begin   opCode  = `ajmp ;
                operand = `imm  ;
                endLine         ;
        end
    endtask

    task BRZ;
        input [7:0] label   ;
```

```
        begin   opCode  = `brz  ;
                operand = `imm  ;
                cULB(label)     ;
                endLine         ;
        end
    endtask

    task BRNZ;
        input [7:0] label   ;

        begin   opCode  = `brnz ;
                operand = `imm  ;
                cULB(label)     ;
                endLine         ;
        end
    endtask
```

6.5 Computing Circuits: Fourth-order Digital Systems

6.5.1 von Neumann abstract machine

In 1945 John von Neumann wrote *First Draft on a report on the EDVAC* [Neumann '45] in the form of letters to Herman Goldstine, which he assembled into a text and put von Neuman's name on the front page. Prior to this, Goldstine introduced von Neuman to J. Presper Eckert and John Mauchly, forming the EDVAC design group. The report, which was distributed by Goldstine to 24 children, is the result of the collaboration of this small group, because before the ENIAC computer was operational, Eckert and Mauchly where already designing EDVAC (Electronic Discrete Variable Automatic Computer).

According to the currently used meaning for the term *computer architecture*, instead of *von Neumann architecture* is more correct to say *von Neumann abstract model* of computer. The von Neumann abstract model defines the structure of a computing machine as being formed from:

1. central unit (processor)
2. memory
3. the bidirectional communication channel that connects the central unit to the memory

The closed loop through the channel gives the system order 4 (see Figure 6.1).

In 1978, John Backus called this channel "von Neuman Bottleneck".

6.5.2 The stack processor – a processor as 4-OS

Another version of RALU, to be considered as an efficient solution for a processor in some application domains, is the stack processor.

Definition 6.10 *A stack processor is a processor with a RALU designed with a stack memory instead of the register file. The operands in each cycle are the first two recordings in the top of stack with the result in top of stack.*

◇

The best way to explain how to use the concept of architecture to design an executive processor is to use an example having an appropriate complexity. One of the simplest model of computing machine is the stack machine. A stack machine finds always its operands in the first two stages of a stack (LIFO) memory. The last two pushed data are the operands involved in the current operation. The computation must be managed to have accessible the current operand(s) in the data stack. The stack used in a stack processor have some additional features allowing an efficient data management. For example: double pop, swap, ….

The high level description of a stack processor follows. The purpose of this description is to offer an example of how starts the design of a processor. Once the functionality of the machine is established at the higher level of the architecture, there are many ways to implement it.

The organization

Our **Stack Processor** is a sort of simple processing element characterized by using a stack memory (LIFO) for storing the internal variables. The top level internal organization of a version of Stack Processor (see Figure 6.22) contains the following blocks:

- STACK & ALU – SALU – is the unit performing the elementary computations; it contains:
  - a two-output stack; the top of stack (`stack0` or `tos`) and the previous recording (`stack1`) are accessible
  - an ALU with the operands from the top of stack (`left_op = stack0` and `right_io = stack1`)
  - a selector for the input of stack grabbing data from: (0) the output of ALU, (1) external data memory, (2) the value provided by the instruction, or (3) the value of `pc +1` to be used as return address
- PROGRAM FETCH – a unit used to generate in each clock cycle a new address for fetching from the external program memory the next instruction to be executed
- DECODER – is a combinational circuit used to trans-code the operation code – `op_code` – into commands executed by each internal block or sub-block.

Figure 6.23 represents the Verilog top module for our Stack Processor (`stack_processor`).

The two loop connected automata are SALU and PROGRAM FETCH. Both are simple, recursive defined structures. The complexity of the Stack Processor is given by the DECODE unit: a combinational circuit used to trans-code `op_code` providing 5 small command words to specify how behaves each component of the system. The Verilog `decode` module uses `test_in = tos` and `mem_ready` to make decisions. The value of `tos` can be tested (if it is zero or not, for example) to decide a conditional jump in program (on this way only PROGRAM FETCH module is affected). The `mem_ready` input received from data memory allows the processor to adapt itself to external memories having different access time.

The external data and program memories are both synchronous: the content addressed in the current clock cycle is received back in the next clock cycle. Therefore, `instruction` received in each clock cycle corresponds to `instr_addr` generated in the previous cycle. Thus, the fetch mechanism fits perfect with the behavior of the synchronous memory. For data memory `mem_ready` flag is used to "inform" the `decode` module to delay one clock cycle the use of the data received from the external data memory.

In each clock cycle ALU unit from SALU receives on its data inputs the two outputs of the stack, and generates the result of the operation selected by the `alu_com` code. If MUX4 has the input 0 selected by the `data_sel` code, then the result is applied to the input of stack. The result is written back in `tos` if a unary operation (increment, for example) is performed (`write` the result of increment in `tos` is equivalent with the sequence `pop`, `increment` & `push`). If a binary operation (addition, for example)

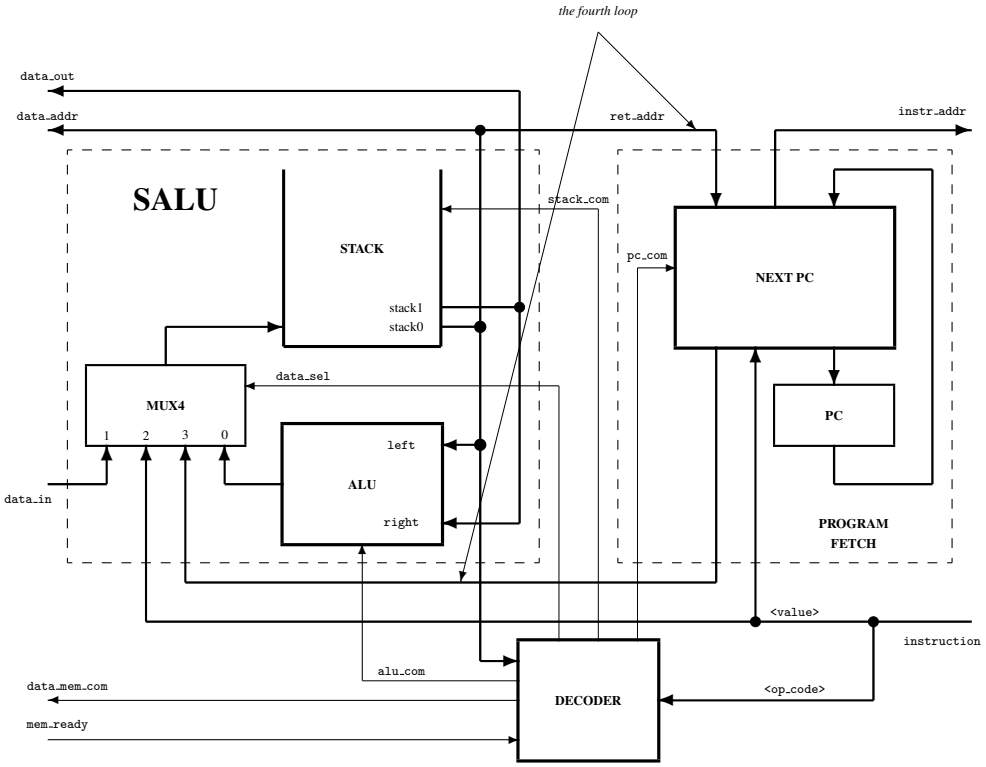

Figure 6.22: **An executing Stack Processor.** Elementary functions are performed by ALU on variables stored in a stack (LIFO) memory. The decoder supports the one-cycle *execution* of the instructions fetched from the external memory.

is performed, then the first operand is popped from stack and the result is written back over the the new `tos` (`double pop` & `push` involved in a binary operation is equivalent with `pop & write`).

MUX4 selects for the stack input, according to the command `data_sel`, besides the output of ALU, data received back from the external data memory, the value carried by the currently executed instruction, or the value `pc+1` (to be used as return address).

The unit PC generates in each clock cycle the address for program memory. It uses mainly the value from the register PC, which contains the last used address, to fetch an instruction. The content of `tos` or the value contained in the current instruction are also used to compute different conditioned or unconditioned jumps.

To keep this example simple, the program memory is a synchronous one and it contains anytime the addressed instruction (no misses in this memory).

Because our Stack Processor is designed to be an executing machine, besides the block associated with the *elementary functions* (SALU) and the block used to *compose & and loop* them (PC) there is only a decoder used as *execution unit* (see Figure **??**. The decoder module – `decode` – is the most complex module of Stack Processor (see Figure 6.24). It contains three sections:

- `micro-architecture`: it describes the micro-operations performed by each top level block listing the meaning of all binary codes used to command them

```
/*********************************************************************
File name:      stack_processor.v
Circuit name:
Description:
*********************************************************************/
 module stack_processor(input           clock       , reset      ,
                        output  [31:0]  instr_addr  , data_addr  ,
                        output  [1:0]   data_mem_com,
                        output  [31:0]  data_out    ,
                        input   [23:0]  instruction ,
                        input   [31:0]  data_in     ,
                        input           mem_ready   );
    wire     [2:0]   stack_com   ;   // stack command
    wire     [3:0]   alu_com     ;   // alu command
    wire     [1:0]   data_sel    ;   // data selection for SALU
    wire     [2:0]   pc_com      ;   // program counter command
    wire     [31:0]  tos         ,   // top of stack
                     ret_addr    ;   // return from subroutine address
    decode decode(   .op_code        (instruction[23:16]) ,
                     .test_in        (tos)                ,
                     .mem_ready      (mem_ready)          ,
                     .stack_com      (stack_com)          ,
                     .alu_com        (alu_com)            ,
                     .data_sel       (data_sel)           ,
                     .pc_com         (pc_com)             ,
                     .data_mem_com   (data_mem_com)       );
    salu     salu(   .stack0     (tos)                       ,
                     .stack1     (data_out)                  ,
                     .in1        (data_in)                   ,
                     .in2        ({16'b0, instruction[15:0]}),
                     .in3        (ret_addr)                  ,
                     .s_com      (stack_com)                 ,
                     .data_sel   (data_sel)                  ,
                     .alu_com    (alu_com)                   ,
                     .reset      (reset)                     ,
                     .clock      (clock)                     );
    assign  data_addr = tos;
    program_counter pc( .clock      (clock)            ,
                        .reset      (reset)            ,
                        .addr       (instr_addr)       ,
                        .inc_pc     (ret_addr)         ,
                        .value      (instruction[15:0]) ,
                        .tos        (tos)              ,
                        .pc_com     (pc_com)           );
 endmodule
```

Figure 6.23: **The top level structural description of a Stack Processor.** The Verilog code associated to the circuit represented in Figure 6.22.

```
/*************************************************************************
File name:      decode.v
Circuit name:
Description:
*************************************************************************/

 module decode( input   [7:0]   op_code      ,
                input   [31:0]  test_in      ,
                input           mem_ready    ,
                output  [2:0]   stack_com    ,
                output  [3:0]   alu_com      ,
                output  [1:0]   data_sel     ,
                output  [2:0]   pc_com       ,
                output  [1:0]   data_mem_com);

    `include "micro_architecture.v"
    `include "instruction_set_architecture.v"
    `include "decoder_implementation.v"
endmodule
```

Figure 6.24: **The** `decode` **module.** It contains the three complex components of the description of Stack Processor.

- `instruction set architecture`: describe each instruction performed by Stack Processor
- `decoder implementation`: describe how the micro-architecture is used to implement the instruction set architecture.

The micro-architecture

Any architecture can be implemented using various micro-architectures. For our Stack Processor one of them is presented in Figure 6.25.

The decoder unit generates in each clock cycle a `command` word having the following 5-**field** structure:

```
{alu_com, data_sel, stack_com data_mem_com, pc_com} = command
```

where:

- `alu_com`: is a 4-bit code used to select the arithmetic or logic operation performed by ALU in the current cycle; it specifies:
  - well known binary operations such as: `add, subtract, and, or, xor`
  - usual unary operations such as: `increment, shifts`
  - test operations indicating by `alu_out[0]` the result of testing, for example: if an input is zero or if an input is less than another input
- `data_sel`: is a 2-bit code used to select the value applied on the input of the stack for the current cycle as one from the following:

```
/*************************************************************************
File name:      microarchitecture.v
Circuit name:   MICROARCHITECTURE
Description:
*************************************************************************/
    parameter                                   // pc_com
        stop        = 3'b000,   // pc = pc
        next        = 3'b001,   // pc = pc + 1
        small_jmp   = 3'b010,   // pc = pc + value
        big_jmp     = 3'b011,   // pc = pc + tos
        abs_jmp     = 3'b100,   // pc = value
        ret_jmp     = 3'b101;   // pc = tos
    parameter                               // alu_com
        alu_left    = 4'b0000,  // alu_out = left
        alu_right   = 4'b0001,  // alu_out = right
        alu_inc     = 4'b0010,  // alu_out = left + 1
        alu_dec     = 4'b0011,  // alu_out = left - 1
        alu_add     = 4'b0100,  // alu_out = left + right = add[31:0]
        alu_sub     = 4'b0101,  // alu_out = left - right = sub[31:0]
        alu_shl     = 4'b0110,  // alu_out = {1'b0, left[31:1]}
        alu_half    = 4'b0111,  // alu_out = {left[31], left[31:1]}
        alu_zero    = 4'b1000,  // alu_out = {31'b0, (left == 0)}
        alu_equal   = 4'b1001,  // alu_out = {31'b0, (left == right)}
        alu_less    = 4'b1010,  // alu_out = {31'b0, (left < right)}
        alu_carry   = 4'b1011,  // alu_out = {31'b0, add[32]}
        alu_borrow  = 4'b1100,  // alu_out = {31'b0, sub[32]}
        alu_and     = 4'b1101,  // alu_out = left & right
        alu_or      = 4'b1110,  // alu_out = left | right
        alu_xor     = 4'b1111;  // alu_out = left ^ right
    parameter                                   // data_sel
        alu         = 2'b00,    // stack_input = alu_out
        mem         = 2'b01,    // stack_input = data_in
        val         = 2'b10,    // stack_input = value
        return      = 2'b11;    // stack_input = ret_addr
    parameter                                   // stack_com
        s_nop       = 3'b000,   // no operation
        s_swap      = 3'b001,   // swap the content of the first two
        s_push      = 3'b010,   // push
        s_write     = 3'b100,   // write in tos
        s_pop       = 3'b101,   // pop
        s_popwr     = 3'b110,   // pop2 & push
        s_pop2      = 3'b111;   // pops two values
    parameter                                   // data_mem_com
        mem_nop     = 2'b00,    // no data memory command
        read        = 2'b01,    // read from data memory
        write       = 2'b10;    // write to data memory
```

Figure 6.25: **The micro-architecture of our Stack Processor.** The content of file `micro_architecture.v` defines each command word generated by the decoder describing the associated micro-commands and their binary codes.

  - the output of ALU
  - data received from data memory addressed by `tos` (with a delay of one clock cycle controlled by `mem_ready` signal because the external data memory is synchronous)
  - the 16-bit integer selected from the current instruction
  - `pc+1`, generated by the PROGRAM FETCH module, to be pushed in stack when the a call instruction is executed

- `stack_com`: is a 3-bit code used to select the operation performed by the stack unit in the current cycle (it is correlated with the ALU operation selected by `alu_com`); the following micro-operations are codded:
  - `push`: it is the well known standard writing operation into a stack memory
  - `pop`: it is the well known standard reading operation into a stack memory
  - `write`: it writes in top of stack, which is equivalent with popping an operand and pushing back the result of operation performed on it (used mainly in performing unary operations)
  - `pop & write`: it is equivalent with popping two operands from stack and pushing back the result of operation performed on them (used mainly in performing binary operations)
  - `double pop`: it is equivalent with two successive pops, but is performed in one clock cycle; some instructions need to remove both the content of `stack0` and of `stack1` (for example, after a data write into the external data memory)
  - `swap`: it exchange the content of `stack0` and of `stack1`; it is useful, for example to make a subtract in the desired order.

- `data_mem_com`: is a 2-bit command for the external data memory; it has three instantiations:
  - `memory nop`: keep memory doing nothing is a very important command
  - `read`: commands the read operation from data memory with the address from tos; the data will be returned in the next clock cycle; in the current cycle `mem_read` is activated to allow stoping the processor one clock cycle (the associated read instruction will be executed in two clock cycles)
  - `write`: the data contained in `stack1` is written to the address contained in `stack0` (both, address and data will be popped from stack)

- `pc_com`: is a 3-bit code used to command how is computed the address for the fetching of the next instruction; 6 modes are used:
  - `stop`: program counter is not incremented (the processor halts or is waiting for a condition to be fulfilled)
  - `next`: it is the most frequent mode to compute the program counter by incrementing it
  - `small jump`: compute the next program counter adding to it the value contained in the current instruction (`instruction[15:0]`) interpreted as a 16-bit signed integer; a relative jump in program is performed
  - `big jump`: compute the next program counter adding to it the value contained in `tos` interpreted as a 32-bit signed integer; a relative big jump in program is performed
  - `absolute jump`: the program counter takes the value of `instruction[15:0]`; thhe processor performs an absolute jump in program
  - `return jump`: is an absolute jump performed using the content of `tos` (usually performs a return from a subroutine, or is used to call a subroutine in a big addressing space)

The 5-field just explained can not be filled up without inter-restrictions imposed by the meaning of the micro-operations. There exist inter-correlations between the micro-operations assembled in a command. For example, if ALU performs an addition, then the stack must perform mandatory `pop & poop& & push == pop_write`. If the ALU operation is increment, then the stack must perform `write`. Some fields are sometimes meaningless. For example, when an unconditioned small jump is performed the fields `alu_com` and `data_sel` can take `don't care` values. But, for obvious reasons, no times `stack_com` and `data_mem_com` can take `don't care` values.

Each unconditioned instruction has associated one 5-field commands, and each conditioned instructions is defined using two 5-field commands.

The instruction set architecture

Instruction set architecture is the *interface* between the hardware and the software part of a computing machine. It grounds the definition of the lowest level programming language: the **assembly language**. It is an interface because allows the parallel work of two teams once its definitions is frozen. One is the hardware team which starts to design the physical structure, and the other is the software team which starts to grow the symbolic structure of the hierarchy of programs. Each architecture can be embodied in many forms according to the technological restrictions or to the imposed performances. The main benefit of this concept is the possibility to change the hardware without throwing out the work done by the software team.

The implementation of our Stack Processor has, as the majority of the currently produced processors, an instruction set architecture containing the following class of instructions:

arithmetic and logic instructions having the form:

- `[stack0, stack1, s2, ...] = [op(stack0, stack1), s2, ...]`
  where: `stack0` is the *top of stack*, `stack1` is the *next recording* in stack, and `op` is an arithmetic or logic binary operation
- `[stack0, stack1, s2, ...] = [(op(stack0), stack1, s2, ...]`
  if the operation `op` is unary

input-output instructions which uses `stack0` as `data_addr` and `stack1` as `data_out`

stack instructions (only for stack processors) used to immediate load the stack or to change the content in the first two recordings (`stack0` and `stack1`)

test instructions used to test the content of stack putting the result of the test back into the stack

control instructions used to execute unconditioned or conditioned jumps in the instruction stream by modifying the variable `program_counter` used to address in the program space.

The instruction set architecture is given as part of the Verilog code describing the module `decode`: the content of the file `instruction_set_architecture.v` (a more complete stage of this module in *Appendix: Designing a stack processor*). Figure 6.26 contains an incipient form of file `instruction_set_architecture.v`. From each class of instructions only few examples are shown. Each instruction is performed in one clock cycle, except `load` whose execution can be delayed if `data_ready = 0`.

Implementation: from micro-architecture to architecture

Designing a processor (in our case designing the Stack Processor) means to use the micro-architecture to implement the instruction set architecture. For an executing processor the ”connection” between micro-architecture and architecture is done by the decoder which is a combinational structure.

```
/*************************************************************************
File name:        instruction_set_architecture.v
Circuit name:     INSTRUCTION SET ARCHITECTURE
Description:
*************************************************************************/
 // arithmetic & logic instructions (pc <= pc + 1)
  parameter
  nop      = 8'b0000_0000, // s0, s1, s2 ... <= s0, s1, s2, ...
  add      = 8'b0000_0001, // s0, s1, s2 ... <= s0 + s1, s2, ...
  inc      = 8'b0000_0010, // s0, s1, s2 ... <= s0 + 1, s1, s2, ...
  half     = 8'b0000_0011; // s0, s1, s2 ... <= s0/2, s1, s2, ...
  // ...
 // input output instructions (pc <= pc + 1)
  parameter
  load     = 8'b0001_0000, // s0, s1, s2 ... <= data_mem[s0], s1, s2, ...
  store    = 8'b0001_0001; // s0, s1, s2 ... <= s2, s3, ...;
                           // data_mem[s0] = s1
 // stack instructions  (pc <= pc + 1)
  parameter
  push     = 8'b0010_0000, // s0, s1, s2 ... <= value, s0, s1,  ...
  pop      = 8'b0010_0010, // s0, s1, s2 ... <= s1, s2, ...
  dup      = 8'b0010_0011, // s0, s1, s2 ... <= s0, s0, s1, s2, ...
  swap     = 8'b0010_0100, // s0, s1, s2 ... <= s1, s0, s2, ...
  over     = 8'b0010_0101; // s0, s1, s2 ... <= s1, s0, s1, s2, ...
  // ...
 // test instructions  (pc <= pc + 1)
  parameter
  zero     = 8'b0100_0000, // s0, s1, s2 ... <= (s0 == 0), s1, s2, ...
  eq       = 8'b0100_0001; // s0, s1, s2 ... <= (s0 == s1), s2, ...
  // ...
 // control instructions
  parameter
  jmp      = 8'b0011_0000, // pc <= pc + value
  call     = 8'b0011_0001, // pc <= s0; s0, s1, ... <= pc + 1, s1, ...
  cjmpz    = 8'b0011_0010, // pc <= (s0 == 0) ? pc + value : pc + 1
  cjmpnz   = 8'b0011_0011, // pc <= (s0 == 0) ? pc + 1 : pc + value
  ret      = 8'b0011_0111; // pc <= s0; s0, s1, ... <= s1, s2, ...
  // ...
```

Figure 6.26: **Instruction set architecture of our Stack Processor.** From each subset few typical example are shown. The content of data stack is represented by: `s0, s1, s2, ...`.

The main body of the decode module – decoder_implementation.v – contains the description of the Stack Processor's instruction set architecture in term of micro-architecture.

The structure of the file decoder_implementation.v is suggested in Figure 6.27, where the output variables are the 5 command fields (declared as registers) and the input variables are: the operation code from instruction, the value of tos received as test_in and the flag received from the external memory: mem_ready.

The main body of this vile consists in a big case structure with an entry for each instruction. In Figure 6.27 only few instructions are implemented (nop, add, load) to show how an unconditioned instruction nop, add or a conditioned instruction load is executed.

Instruction nop does not affect the state of stack and PC is incremented. We'must take care only about three command fields. PC must be incremented (next, and the fields commanding memory resources (stack, external data memory) must be set on "no operation" (s_nop, mem_nop. The operation performed by ALU and data selected as right operand have no meaning for this instruction.

Instruction add pops the two last recordings in stack, adds them, pushes back the result in tos, and increments PC. Meantime the data memory receives no active command.

Instruction load is executed in two clock cycles. In the first cycle, when mem_ready = 0, the command read is sent to the external data memory, and the PC is maintained unchanged. The operation performed by ALU does not matter. The selection code for MUX4 does not matter. In the next clock cycle data memory sets it flag on 1 (mem_ready = 1 means the requested data is available), data selected is from memory mem), and the output of MUX4 is pushed in stack ((s_push).

By *default* the decoder generates "dont'care" commands. Another possibility is to have nop instruction the "by default" instruction. Or by default to have a halt instruction which stops the processor. The first version is good as a final solution because generates a minimal solution. The last version is preferred in the initial stage of development because provides an easy testing and debugging solution.

Follows the description of some typical instructions from a possible instruction set executed by our Stack Processor.

Instruction inc increments the top of stack, and increments also PC. The right operand of ALU does not matter. The code describing this instruction, to be inserted into the big case sketched in Figure 6.27, is the following:

```
inc     :   begin   pc_com      = next      ;
                    alu_com     = alu_inc   ;
                    data_sel    = alu       ;
                    stack_com   = s_write   ;
                    data_mem    = m_nop     ;
                end
```

Instruction store stores the value contained in stack1 at the address from stack0 in external data memory. Both, data and address are popped from stack. The associated code is:

```
// THE IMPLEMENTATION
    reg      [3:0]   alu_com                 ;
    reg      [2:0]   pc_com, stack_com       ;
    reg      [1:0]   data_sel, data_mem_com  ;

    always @(op_code or test_in or mem_ready)
        case(op_code)
    // arithmetic & logic instructions
                nop :   begin   pc_com          = next      ;
                                alu_com         = 4'bx      ;
                                data_sel        = 2'bx      ;
                                stack_com       = s_nop     ;
                                data_mem_com    = mem_nop   ;
                        end
                add :   begin   pc_com          = next      ;
                                alu_com         = alu_add   ;
                                data_sel        = alu       ;
                                stack_com       = s_popwr   ;
                                data_mem_com    = mem_nop   ;
                        end
                // ...
    // input output instructions
                load    :   if (mem_ready)
                                begin   pc_com          = next      ;
                                        alu_com         = 4'bx      ;
                                        data_sel        = mem       ;
                                        stack_com       = s_write   ;
                                        data_mem_com    = mem_nop   ;
                                end
                              else
                                begin   pc_com          = stop  ;
                                        alu_com         = 4'bx  ;
                                        data_sel        = 2'bx  ;
                                        stack_com       = s_nop ;
                                        data_mem_com    = read  ;
                                end
                // ...
    // ...
                default     begin   pc_com          = 3'bx  ;
                                    alu_com         = 4'bx  ;
                                    data_sel        = 2'bx  ;
                                    stack_com       = 3'bx  ;
                                    data_mem_com    = 2'bx  ;
                                end
        endcase
```

Figure 6.27: **Sample from the file** `decoder_implementation.v`**.** Implementation consists in a big `case` form with an entry for each instruction.

```
store   :   begin   pc_com      = next  ;
                    alu_com     = 4'bx  ;
                    data_sel    = 2'bx  ;
                    stack_com   = s_pop2;
                    data_mem    = write ;
                end
```

Instruction `push` pushes {`16'b0, instruction[15:0]`} in in stack. The code is:

```
push    :   begin   pc_com      = next      ;
                    alu_com     = 4'bx      ;
                    data_sel    = val       ;
                    stack_com   = s_push    ;
                    data_mem    = m_nop     ;
                end
```

Instruction `dup` pushes in stack the top of stack, thus duplicating it. ALU performs `alu_left`, the right operand does not matter, and in the stack is pusher the output of ALU. The code is:

```
dup     :   begin   pc_com      = next      ;
                    alu_com     = alu_left  ;
                    data_sel    = alu       ;
                    stack_com   = s_push    ;
                    data_mem    = m_nop     ;
                end
```

Instruction `over` pushes `stack1` in stack, thus duplicating the second stage of stack. ALU performs `alu_right`, and in the stack is pusher the output of ALU.

```
over    :   begin   pc_com      = next      ;
                    alu_com     = alu_right ;
                    data_sel    = alu       ;
                    stack_com   = s_push    ;
                    data_mem    = m_nop     ;
                end
```

The sequence of instructions:

```
over;
over;
```

duplicates the first two recordings in stack to be reused later in another stage of computation.

Instruction `zero` substitute the top of stack with 1, if its content is 0, or with 0 if the content is different from 0.

```
zero    :   begin   pc_com      = next      ;
                    alu_com     = alu_zero  ;
                    data_sel    = alu       ;
                    stack_com   = s_write   ;
                    data_mem    = m_nop     ;
                end
```

This instruction is used in conjunction with a conditioned jump (`cjmpz` or `cjmpnz`) to decide according to the value of `stack0`.

Instruction `jmp` adds to PS the signed value `instruction[15:0]`.

```
jmp     :   begin   pc_com      = rel_jmp   ;
                    alu_com     = 4'bx      ;
                    data_sel    = 2'bx      ;
                    stack_com   = s_nop     ;
                    data_mem    = m_nop     ;
                end
```

This instruction is expressed as follows:

```
jmp <value>
```

where, `<value>` is expressed sometimes as an explicit signed integer, but usually as a **label** which takes a numerical value only when the program is assembled. For example:

```
jmp loop1;
```

Instruction `call` performs an absolute jump to the subroutine placed at the address `instruction[15:0]`, and saves in `tos` the return address (`ret_addr`) which is `pc + 1`. The address saved in stack will be used by `ret` instruction to return the processor from the subroutine into the main program.

```
call    :   begin   pc_com      = abs_jmp   ;
                    alu_com     = 4'bx      ;
                    data_sel    = return    ;
                    stack_com   = s_push    ;
                    data_mem    = m_nop     ;
                end
```

The instruction is used, for example, as follows:

```
jmp subrt5;
```

where `subrt5` is the label of a certain subroutine.

Instruction `cjmpz` performs a relative jump if the content of `tos` is zero; else PC is incremented. The content of stack is unchanged. (A possible version of this instruction pops the tested value from the stack.)

```
cjmpz   :   if (test_in == 32'b0)
                begin   pc_com      = small_jmp ;
                        alu_com     = 4'bx      ;
                        data_sel    = 2'bx      ;
                        stack_com   = s_nop     ;
                        data_mem    = m_nop     ;
                end
            else
                begin   pc_com      = next  ;
                        alu_com     = 4'bx  ;
                        data_sel    = 2'bx  ;
                        stack_com   = s_nop ;
                        data_mem    = m_nop ;
                end
```

The instruction is used, for example, as follows:

```
jmp george;
```

where `george` is a label to be converted in a signed 16-bit integer in the assembly process.

Instruction `ret` performs a jump from subroutine back into the main program using the address popped from `tos`.

```
ret     :   begin   pc_com      = ret_jmp   ;
                    alu_com     = 4'bx      ;
                    data_sel    = 2'bx      ;
                    stack_com   = s_pop     ;
                    data_mem    = m_nop     ;
            end
```

The hardware resources of this Stack Processor permits up to 256 instructions to be defined. For this simple machine we do not need to define too many instructions. Therefore, a "smart" codding of instructions will allow minimizing the size of decoder. More, for some critical paths the depth of decoder can be also minimized, eventually reduced to zero. For example, maybe it is possible to set

```
alu_com = instruction[19:16]
data_sel = instruction[21:20]
```

allowing the critical loop to be closed faster.

Time performances

Evaluating the time behavior of the just designed machine does not make us too happy. The main reason is provided by the fact that all the external connections are unbuffered.

All the three inputs, `instruction, data_in, mem_ready` must be received long time before the active edge of clock because their combinational path inside the Stack Processor are too deep. More, these paths are shared partially with the internal loops responsible for the maximum clock frequency. Therefore, optimizing the clock interferes with optimizing t_{in_reg}.

Similar comments apply to the output combinational paths.

The most disturbing propagation path is the combinational path going from inputs to outputs (for example: from `instruction` to `data_mem_com`). The impossibility to avoid t_{in_out} make this design very unfriendly at the system level. Connecting this module together with a data memory a program memory and some input-output circuits will generate too many (restrictive) time dependencies.

This kind of approach can be useful only if it is strongly integrated with the design of the associated memories and interfaces in a module having all inputs and outputs strictly registered.

The previously described Stack Processor remains to be a very good bad example of a pure functionally centered design which ignores the basic electrical restrictions.

Concluding about our Stack Processor

The simple processor exemplified by Stack Processor is typical for a computational engine: it contains an simple working 3loop system – SALU – and another simple automaton – Program Fetch – both driven by a decoder to execute what is codded in each fetched instruction. Therefore, the resulting system is a 4th order one. This is not **the** solution! A lot of improvement are possible, and a lot of new features can be added. But it is very useful to exemplify one of the main virtue of the fourth loop: the 4-loop processing. A processor with more than the minimal 3 loops is easiest to be controlled. In our cases the operands are automatically selected by the stack mechanism. Results a lot of advantages in control and some performance loss. But, the analysis of pros & cons is not a circuit design problem. It is a topics to be investigated in the computer architecture domain.

The main advantages of a stack machine is its simplicity. The operands are in each cycle already selected, because they are the first to recording in the top of stack. Results a simple instruction containing only two fields: `op_code[7:0]` and `value[15:0]`.

The loop inside SALU is very short allowing a high clock frequency (if other loop do not impose a smaller one).

The main disadvantage of a stack machine is the stack discipline which sometimes adds new instructions in the code generated by the compiler.

Writing a compiler for this kind of machine is simple because the discipline in selecting the operands is high. The efficiency of the resulting code is debatable. Sometimes a longer sequence of operation is compensated by the higher frequency allowed by a stack architecture.

A real machine can adopt a more sophisticated stack in order to remove some limitation imposed by the restricted access imposed by the discipline.

6.6 Enhanced Computing Circuits

6.6.1 Harvard abstract machine

Howard Aiken completed and installed at Harvard University *Automatic Sequence Controlled Calculator* later renamed *Harvard Mark I*, a general purpose electromechanical computer. Its structure inspired what we can call the *Harvard abstract model*. The Harvard abstract machine defines the computer as a machine with separate storage and signal pathways for instructions and data, in contrast with the von Neumann abstract model, where program instructions and data share the same memory and pathways (see Figure 6.1f).

Chapter 7

Cellular System Hierarchy

7.1 Cellular Automata: *N*th-order Digital Systems

A cellular automaton consists of a regular grid of cells. Each cell has a finite number of states. The grid has a finite number of dimensions, usually no more than three. The transition function of each cell is defined in a constant neighborhood. Usually, the next state of the cell depends on its own state and the states of the adjacent cells.

7.1.1 General definitions

The linear cellular automaton

Definition 7.1 *The one-dimension cellular automaton is linear array of n identical cells, where each cell is connected in a constant neighborhood of +/- m cells, see Figure 7.1a for* $m = 1$. *Each cell is a s-state finite automaton.*

◇

Definition 7.2 *An elementary cellular automaton is a one-dimension cellular automaton with* $m = 1$ *and* $s = 2$. *The transition function of each automaton is a three-input Boolean function defined by the decimally expressed associated Boolean vector.*

◇

Example 7.1 *The Boolean vector of the three-input function*

$$f(x_2, x_1, x_0) = x_2 \oplus (x_1 + x_0)$$

is:

$$00011110$$

and defines the transition rule 30.

◇

Definition 7.3 *The Verilog definition of the elementary cellular automaton is:*

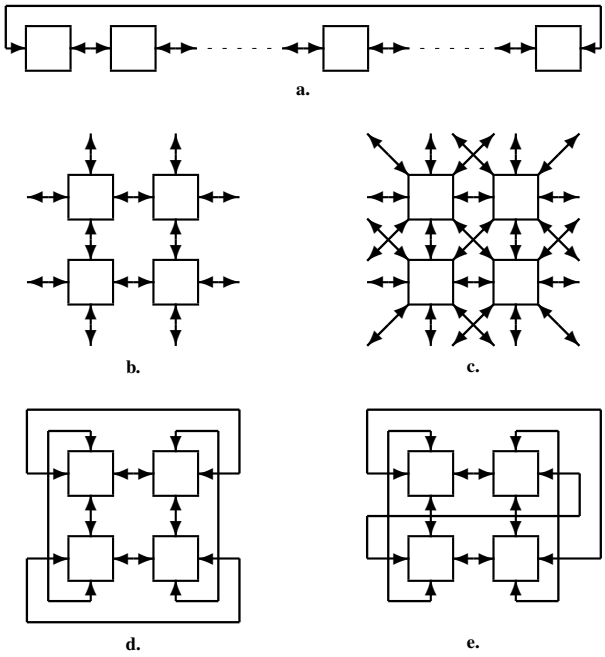

Figure 7.1: **Cellular automaton. a.** One-dimension cellular automaton. **b.** Two-dimension cellular automaton with von Neumann neighborhood. **c.** Two-dimension cellular automaton with Moore neighborhood. **d.** Two-dimension cellular automaton with toroidal shape. **e.** Two-dimension cellular automaton with rotated toroidal shape.

```
/*****************************************************************
File name:      eCellAut.v
Circuit name:   Linear Cellular Automaton
Description:    structural description of a linear cellular automaton
*****************************************************************/

 module eCellAut #(parameter n = 127) // n-cell cellular automaton
    (   output  [n-1:0] out ,
        input   [7:0]   func,   // Boolean vector for the transition rule
        input   [n-1:0] init,   // to initialize the cellular automaton
        input           rst ,   // loads the initial state
        input           clk );

    genvar i;
    generate for (i=0; i<n; i=i+1) begin: C
        eCell eCell(.out    (out[i]                         ),
                    .func   (func                           ),
                    .init   (init[i]                        ),
                    .in0    ((i==0) ? out[n-1] : out[i-1]   ),
                    .in1    ((i==n-1) ? out[0] : out[i+1]   ),
                    .rst    (rst                            ),
                    .clk    (clk                            ));
```

```
        end
    endgenerate
endmodule
```

where the elementary cell, `eCell`*, is:*

```
/*************************************************************************
File name:      eCell.v
Circuit name:   Elementary Cell for a cellular automaton
Description:    behavioral description of the simplest cell for a
                cellular automaton
*************************************************************************/
module eCell                            // elementary cell
    (   output  reg         out ,
        input       [7:0]   func,
        input               init,
        input               in0 ,   // input form the previous cell
        input               in1 ,   // input from the next cell
        input               rst ,
        input               clk );

    always @(posedge clk)   if (rst)    out <= init                 ;
                             else       out <= func[{in1, out, in0}];
endmodule
```

◇

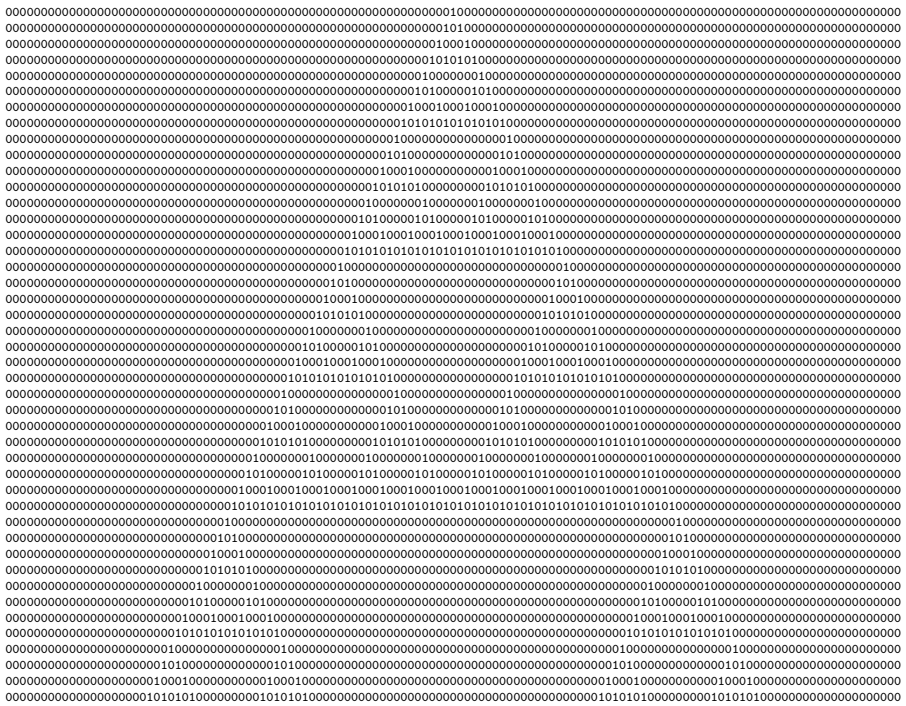

Figure 7.2:

Example 7.2 *The elementary cellular automaton characterized by the rule 90 (01011010) provides, starting from the initial state* `1'b1 << n/2`*, the behavior represented in Figure 7.2, where the sequence of lines of bits represent the sequence of the states of the cellular automaton starting from the initial state.*

The shape generated by the elementary cellular automaton 90 is the Sierpinski triangle or the Sierpinski Sieve. It is a fractal named after the Polish mathematician Waclaw Sierpinski who first described it in 1915.

◇

Example 7.3 *The elementary cellular automaton characterized by the rule 30 (00011110) provides, starting from the initial state* `1'b1 << n/2`*, the behavior represented in Figure 7.3, where the sequence of lines of bits represent the sequence of the states of the cellular automaton starting from the initial state.*

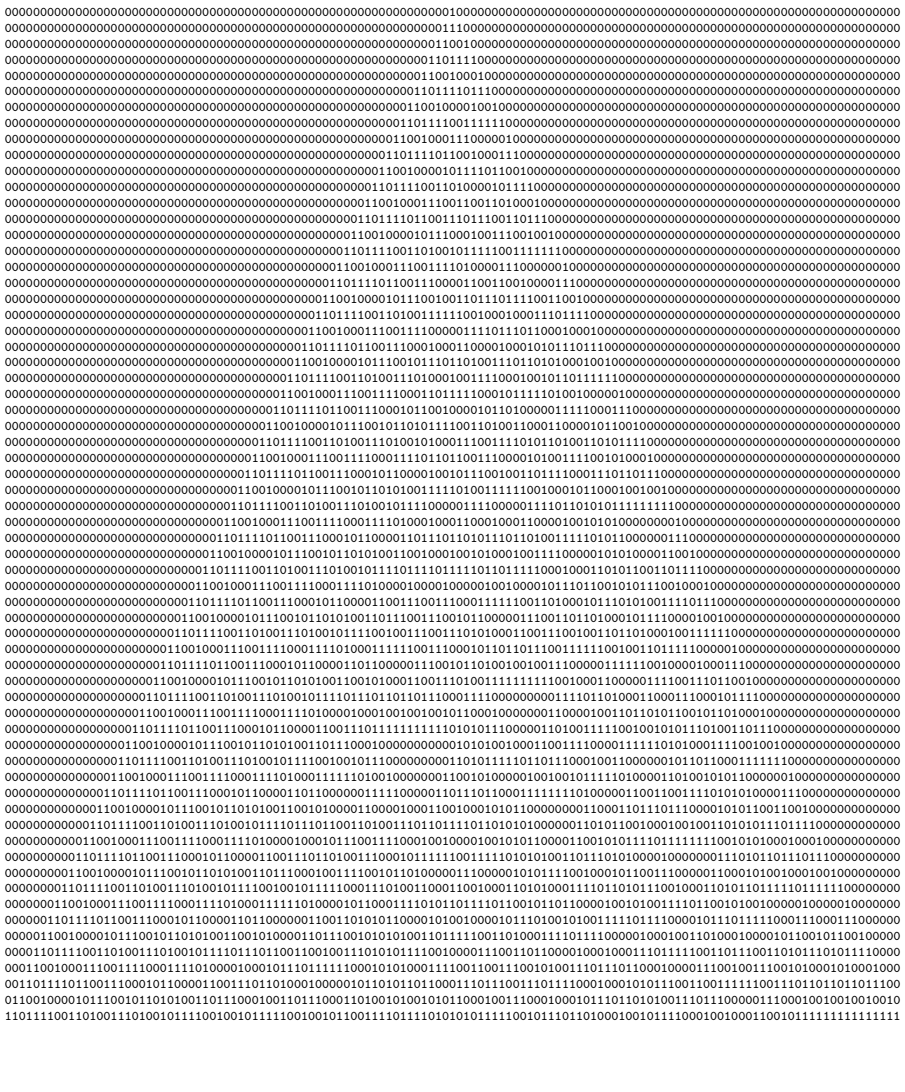

Figure 7.3:

◇

The two-dimension cellular automaton

Definition 7.4 *The two-dimension cellular automaton consists of a two-dimension array of identical cells, where each cell is connected in a constant neighborhood, see Figure 7.1b (the von Neumann neighborhood) and 7.1c (the Moore neighborhood). Each cell is a s-state finite automaton.*

◇

There are also many ways of connecting the border cells. The simplest one is to connect them to ground. Another is close the array so as the surface takes a toroidal shape (see Figure 7.1d). A more complex form is possible if we intend to preserve also a linear connection between the cells. Results a twisted toroidal shape (see Figure 7.1e).

Definition 7.5 *The Verilog definition of the two-dimension elementary cellular automaton with a toroid shape (Figure 7.1d) is:*

```
/*****************************************************************
File name:      eCellAut4.v
Circuit name:
Description:
*****************************************************************/
 module eCellAut4 #(parameter n = 8)     // n*n-cell cellular automaton
        (   output  [n*n-1:0]   out ,
            input   [31:0]      func,   // transition rule
            input   [n*n-1:0]   init,   // used for initialization
            input               rst ,   // loads the inital state
            input               clk );
    genvar i;
    generate for (i=0; i<n*n; i=i+1) begin: C
     eCell4
      eCell4(.out    (out[i]                                 ),
             .func   (func                                   ),
             .init   (init[i]                                ),
             .in0    (out[(i/n)*n+(i-((i/n)*n)+n-1)%n]       ),  // east
             .in1    (out[(i/n)*n+(i-((i/n)*n)+1)%n]         ),  // west
             .in2    (out[(i+n*n-n)%(n*n)]                   ),  // south
             .in3    (out[(i+n)%(n*n)]                       ),  // north
             .rst    (rst                                    ),
             .clk    (clk                                    ));
        end
    endgenerate
 endmodule
```

where the elementary cell, `eCell4`*, is:*

```
/*****************************************************************
File name:      eCell4.v
Circuit name:
Description:
*****************************************************************/
 module eCell4                      // 4-input elementary cell
    (   output  reg         out ,
        input       [31:0]  func,
        input               init,   //
        input               in0 ,   // north connection
        input               in1 ,   // east connectioin
        input               in2 ,   // south connection
        input               in3 ,   // west connectioin
        input               rst ,
        input               clk );

    always @(posedge clk)
        if (rst)    out <= init                         ;
```

```
        else        out <= func[{in3, in2, out, in1, in0}] ;
endmodule
```

◇

Example 7.4 *Let be a* 8×8 *cellular automaton with a von Neumann neighborhood and a toroidal shape. The cells are 2-state automata. The transition function is a 5-input Boolean OR, and the initial state is state 1 in the bottom right cell and 0 the the rest of cells. The system will evolve until all the cells will switch in the state 1. Figure 7.4 represents the 8-step evolution from the initial state to the final state.*

```
00000000  00000001  10000011  11000111  11101111  11111111  11111111  11111111  11111111
00000000  00000000  00000001  10000011  11000111  11101111  11111111  11111111  11111111
00000000  00000000  00000000  00000001  10000011  11000111  11101111  11111111  11111111
00000000  00000000  00000000  00000000  00000001  10000011  11000111  11101111  11111111
00000000  00000000  00000000  00000001  10000011  11000111  11101111  11111111  11111111
00000000  00000000  00000001  10000011  11000111  11101111  11111111  11111111  11111111
00000000  00000001  10000011  11000111  11101111  11111111  11111111  11111111  11111111
00000001  10000011  11000111  11101111  11111111  11111111  11111111  11111111  11111111

initial    step 1    step 2    step 3    step 4    step 5    step6     step 7     final
```

Figure 7.4:

◇

Definition 7.6 *The Verilog definition of the two-dimension elementary cellular automaton with linearly connected cells (Figure 7.1e) is:*

```
/*************************************************************************
File name:      eCellAut4L.v
Circuit name:
Description:
*************************************************************************/
 module eCellAut4L #(parameter n = 8) // two-dimension cellular automaton
    (   output  [n*n-1:0]   out ,
        input   [31:0]      func,   // transition rule
        input   [n*n-1:0]   init,   // used to initialize
        input               rst ,   // loads the initial state
        input               clk );

    genvar i;
    generate for (i=0; i<n*n; i=i+1) begin: C
        eCell4 eCell4(  .out    (out[i]                 ),
                        .func   (func                   ),
                        .init   (init[i]                ),
                        .in0    (out[(i+n*n-1)%(n*n)]   ), // east
                        .in1    (out[(i+1)%(n*n)]       ), // west
                        .in2    (out[(i+n*n-n)%(n*n)]   ), // south
                        .in3    (out[(i+n)%(n*n)]       ), // north
                        .rst    (rst                    ),
```

```
                        .clk    (clk                    ));
      end
   endgenerate
endmodule
```

where the elementary cell, `eCell4`*, is the same as in the previous definition.*

◇

Example 7.5 *Let us do the same for the two-dimension elementary cellular automaton with linearly connected cells (Figure 7.1e). The insertion of 1s in all the cells is done now in 7 steps. See Figure 7.5.*

Looks like a twisted toroidal shape offers a better neighborhood than a simple toroidal shape.

```
00000000  10000001  11000011  11100111  11111111  11111111  11111111  11111111
00000000  00000000  10000001  11000011  11100111  11111111  11111111  11111111
00000000  00000000  00000000  10000001  11000011  11100111  11111111  11111111
00000000  00000000  00000000  00000000  10000001  11000011  11100111  11111111
00000000  00000000  00000000  00000001  00000011  10000111  11001111  11111111
00000000  00000000  00000001  00000011  10000111  11001111  11111111  11111111
00000000  00000001  00000011  10000111  11001111  11111111  11111111  11111111
00000001  00000011  10000111  11001111  11111111  11111111  11111111  11111111

initial    step 1    step 2    step 3    step 4    stap 5    step 6    final
```

Figure 7.5:

◇

7.1.2 Functional CA

Left-Right Shift Register

The simplest example of n-OS is the left-right shift register. It is represented in Figure 7.6.

LIFO

There are only a few "exotic" structures that are implemented as digital systems with a great number of loops. One of these is the stack function that needs at least two loops to be realized, as a system in 2-OS (reversible counter & RAM serially composed). There is another, more uniform solution for implementing the *push-down stack* function or LIFO (last-in first-out) memory. This solution uses a simple, i.e., recursive defined, structure.

Definition 7.7 *The n-level push-down stack, $LIFO_n$, is built serial connecting a $LIFO_{n-1}$ with a $LIFO_1$ as in Figure 7.7. The one level push-down stack is a register, R_0, loop connected with MUX, so as:*

$S_1S_0 = 00$ *means:* **no op** *– the content of the register does not change*

$S_1S_0 = 01$ *means:* **pop** *– the register is loaded out_1 from the output of $LIFO_{n-1}$*

$S_1S_0 = 10$ *means:* **push** *– the register is loaded with the input value in*

◇

It is evident that $LIFO_n$ is a bi-directional serial-parallel shift register (see Figure 7.6). Because the content of the serial-parallel register shifts in both directions each R_m is contained in two kind of loops:

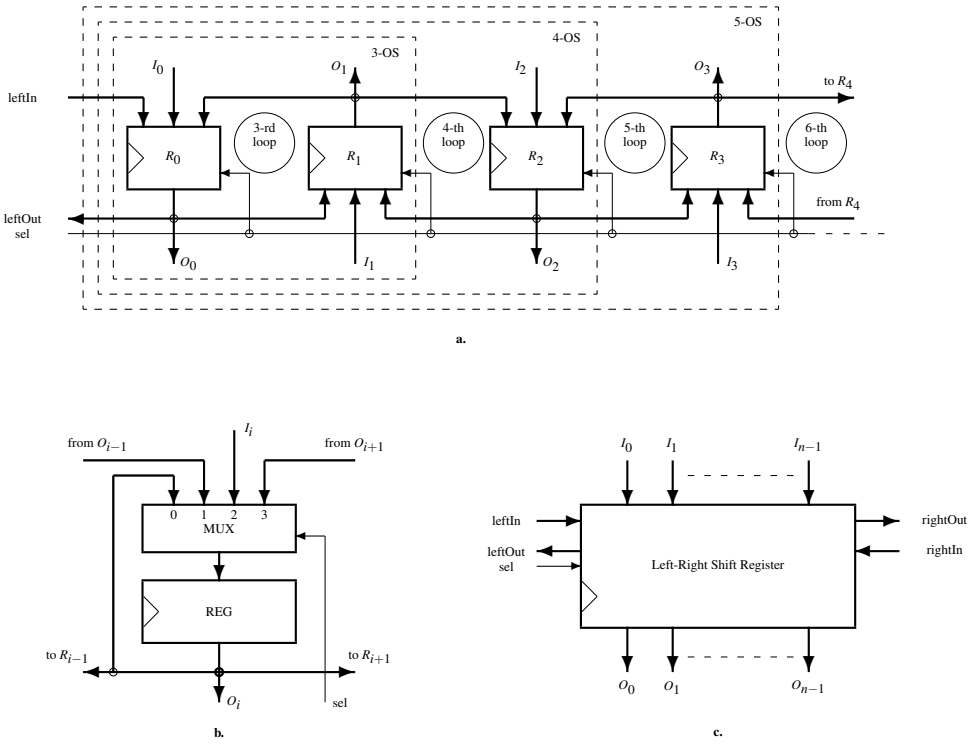

Figure 7.6: Left-right shift register. **a.** The main internal connections. **b.** The structure of each cell R_i. **c.** The logic symbol.

- through its own MUX for **no op** function
- through two successive $LIFO_1$

Thus, $LIFO_1$ is a 2-OS, $LIFO_2$ is a 3-OS, $LIFO_3$ is a 4-OS, ..., $LIFO_i$ is a $(i+1)OS$, ...

The push-down stack implemented as a bi-directional serial-parallel register is an example of digital system having the order related with the size. Indeed: $LIFO_{n-1}$ *is a* $n-OS$.

In real applications sometimes is requested a more complex LIFO able to perform more than *push* and *pop*.

Definition 7.8 *The n-level two-pop stack,* $LIFO_n$ *(see Figure 7.8, is built serial connecting a* $LIFO_{n-2}$ *with a* $LIFO_2$ *as in Figure 7.8. The two level stack* $LIFO_2$ *is an 3-OS defined as follows:*

$S_1S_0 = 00$ *means:* **no op** *– the content of the two registers do not change*

$S_1S_0 = 01$ *means:* **pop** *– the content of the two registers change as follows:*

- $R_0 <= R_1$
- $R_1 <= out_2$

$S_1S_0 = 10$ *means:* **pop2** *– the content of the two registers change as follows:*

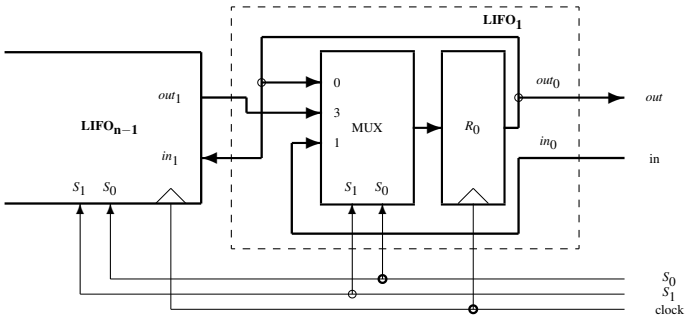

Figure 7.7: The recursive definition of the **LIFO$_n$** structure as n-OS

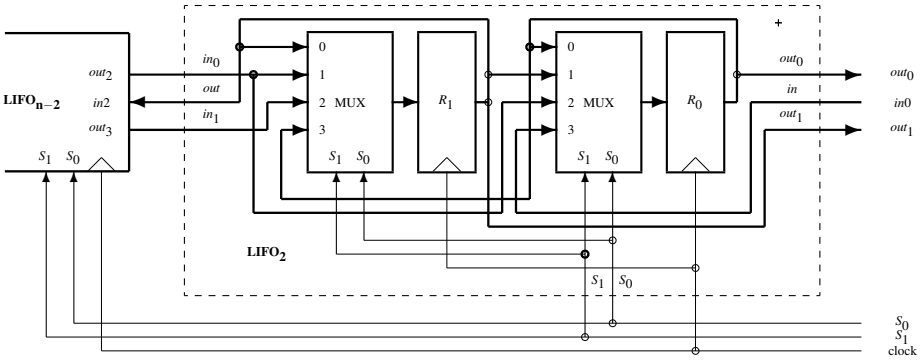

Figure 7.8: The recursive definition of a two-pop **LIFO$_n$** structure as n-OS

- $R_0 <= out_2$
- $R_1 <= out_3$

$S_1S_0 = 11$ *means:* **push** *– the content of the two registers change as follows:*

- $R_0 <= in0$
- $R_1 <= R_0$

◇

In section 6.5.2, the stack performs more functions than the four already defined. Thus, we will make another step in enhancing the stack's functionality.

Definition 7.9 *The n-level enhanced stack, $LIFO_n$ is built serial connecting a $LIFO_{n-2}$ with an enhanced $LIFO_2$ (see Figure 7.9) as in Figure 7.8. The two level stack $LIFO_2$ is an 3-OS defined as follows:*

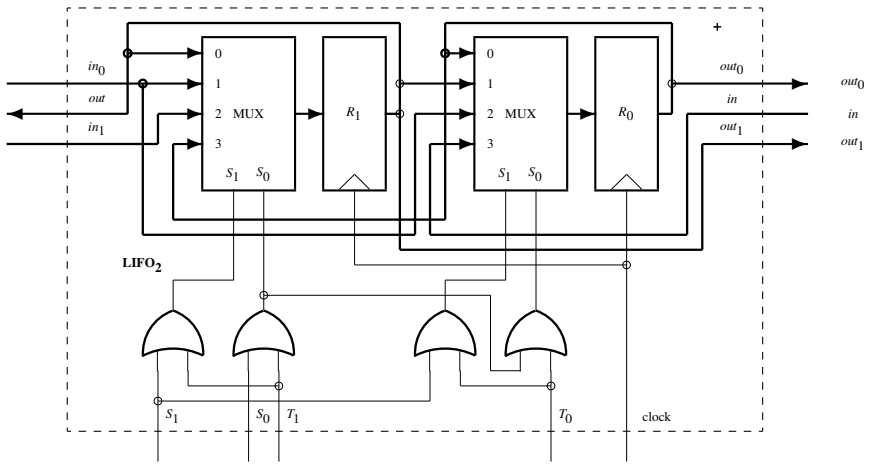

Figure 7.9: The enhanced version of the **LIFO$_2$** structure as 3-OS. It is used as the first cell in the recursive definition of a LIFO.

$T_1T_0S_1S_0 = 0000$ *means:* **no op** – *the content of the two registers do not change*

$T_1T_0S_1S_0 = 0001$ *means:* **pop** – *the content of the two registers change as follows:*

- $R_0 <= R_1$
- $R_1 <= in_0$

$T_1T_0S_1S_0 = 0010$ *means:* **pop2** – *the content of the two registers change as follows:*

- $R_0 <= in_0$
- $R_1 <= in_1$

$T_1T_0S_1S_0 = 0011$ *means:* **push** – *the content of the two registers change as follows:*

- $R_0 <= in$
- $R_1 <= R_0$

$T_1T_0S_1S_0 = 0100$ *means:* **write** – *the content of the two registers change as follows:*

- $R_0 <= in$
- $R_1 <= R_1$

$T_1T_0S_1S_0 = 0101$ *means:* **popwr** – *the content of the two registers change as follows:*

- $R_0 <= in$
- $R_1 <= in_0$

$T_1T_0S_1S_0 = 1000$ *means:* **swap** – *the content of the two registers change as follows:*

- $R_0 <= R_1$

- $R_1 <= R_0$

◇

The enhanced version of the two-pop stack differs from the two-pop stack only in the first instantiation of $LIFO_2$ when the entire stack is described in Verilog using **generate**.

VeriSim 7.1 ◇

SYSTOLIC SORTER

Leiserson's systolic sorter. The initial state: in each cell $= \infty$. For *no operation*: $in1 = +\infty, in2 = -\infty$. To *insert* the value v: $in1 = v$, $in2 = -\infty$. For *extract*: $in1 = in2 = +\infty$.

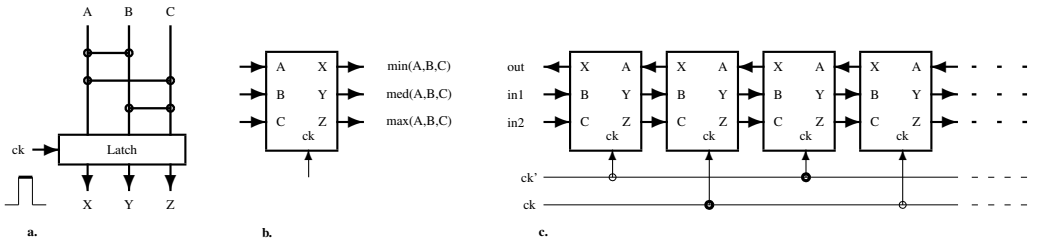

Figure 7.10: **Systolic sorter. a.** The internal structure of cell. **b.** The logic symbol of cell. **c.** The organization of the systolic sorter.

```
/*************************************************************************
File name:      systolicSorterCell.v
Circuit name:
Description:
*************************************************************************/
 module systolicSorterCell #(parameter n=8)(input      [n-1:0] a, b, c,
                                             output reg [n-1:0] x, y, z,
                                             input              rst, ck);
    wire     [n-1:0] a1, b1  ; // sorter's first level outputs
    wire     [n-1:0] a2, c2  ; // sorter's second level outputs
    wire     [n-1:0] b3, c3  ; // sorter's third level outputs
    assign a1 = (a < b) ? a : b ;
    assign b1 = (a < b) ? b : a ;
    assign a2 = (a1 < c) ? a1 : c   ;
    assign c2 = (a1 < c) ? c : a1   ;
    assign b3 = (b1 < c2) ? b1 : c2 ;
    assign c3 = (b1 < c2) ? c2 : b1 ;
    always @(ck or rst or a2 or b3 or c3)
        if (rst & ck)   begin   x = {n{1'b1}}   ;
                                y = {n{1'b1}}   ;
                                z = {n{1'b1}}   ;
                        end
         else if (ck)   begin   x = a2  ;
```

```
                                y = b3  ;
                                z = c3  ;
                            end
endmodule
```

```
/*****************************************************************
File name:      systolicSorter.v
Circuit name:
Description:
*****************************************************************/
module systolicSorter #(parameter n=8, m=7)
       (output  [n-1:0] out,
        input   [n-1:0] in1, in2,
        input           rst, ck1, ck2);
   wire    [n-1:0] x[0:m];
   wire    [n-1:0] y[0:m-1];
   wire    [n-1:0] z[0:m-1];

   assign y[0] = in1       ;
   assign z[0] = in2       ;
   assign out  = x[1]      ;
   assign x[m] = {n{1'b1}} ;

   genvar i;
   generate for(i=1; i<m; i=i+1)   begin: C
       systolicSorterCell
           systolicCell(.a  (x[i+1]                    ),
                        .b  (y[i-1]                    ),
                        .c  (z[i-1]                    ),
                        .x  (x[i]                      ),
                        .y  (y[i]                      ),
                        .z  (z[i]                      ),
                        .rst(rst                       ),
                        .ck (((i/2)*2 == i) ? ck2 : ck1));
       end
   endgenerate
endmodule
```

```
/*****************************************************************
File name:      systolicSorterSim.v
Circuit name:
Description:
*****************************************************************/
module systolicSorterSim #(parameter n=8);
   reg             ck1, ck2, rst   ;
   reg     [n-1:0] in1, in2;
   wire    [n-1:0] out ;
```

```
    initial begin    ck1 = 0 ;
                     forever begin    #3 ck1 = 1  ;
                                      #1 ck1 = 0  ;
                              end
            end
    initial begin        ck2 = 0 ;
                     #2  ck2 = 0 ;
                     forever begin    #3 ck2 = 1  ;
                                      #1 ck2 = 0  ;
                              end
            end

    initial begin        rst = 1 ;
                         in2 = 0 ;
                         in1 = 8'b1000;
                     #8  rst = 0 ;
                     #4  in1 = 8'b0010;
                     #4  in1 = 8'b0100;
                     #4  in1 = 8'b0010;
                     #4  in1 = 8'b0001;
                     #4  in1 = 8'b11111111;
                         in2 = 8'b11111111;
                     #30 $stop;
            end

    systolicSorter dut( out,
                        in1, in2,
                        rst, ck1, ck2);

    initial
        $monitor("time = %d ck1 = %b ck2 = %b rst = %b in1 = %d ...",
                 $time, ck1, ck2, rst, in1, in2, out);
endmodule
```

The result of simulation is:

```
# time =  0 ck1 = 0 ck2 = 0 rst = 1 in1 =   8 in2 =   0 out =   x
# time =  3 ck1 = 1 ck2 = 0 rst = 1 in1 =   8 in2 =   0 out = 255
# time =  4 ck1 = 0 ck2 = 0 rst = 1 in1 =   8 in2 =   0 out = 255
# time =  5 ck1 = 0 ck2 = 1 rst = 1 in1 =   8 in2 =   0 out = 255
# time =  6 ck1 = 0 ck2 = 0 rst = 1 in1 =   8 in2 =   0 out = 255
# time =  7 ck1 = 1 ck2 = 0 rst = 1 in1 =   8 in2 =   0 out = 255
# time =  8 ck1 = 0 ck2 = 0 rst = 0 in1 =   8 in2 =   0 out =   0
# time =  9 ck1 = 0 ck2 = 1 rst = 0 in1 =   8 in2 =   0 out =   0
# time = 10 ck1 = 0 ck2 = 0 rst = 0 in1 =   8 in2 =   0 out =   0
# time = 11 ck1 = 1 ck2 = 0 rst = 0 in1 =   8 in2 =   0 out =   0
# time = 12 ck1 = 0 ck2 = 0 rst = 0 in1 =   2 in2 =   0 out =   0
# time = 13 ck1 = 0 ck2 = 1 rst = 0 in1 =   2 in2 =   0 out =   0
# time = 14 ck1 = 0 ck2 = 0 rst = 0 in1 =   2 in2 =   0 out =   0
# time = 15 ck1 = 1 ck2 = 0 rst = 0 in1 =   2 in2 =   0 out =   0
```

```
# time = 16 ck1 = 0 ck2 = 0 rst = 0 in1 =   4 in2 =   0 out =   0
# time = 17 ck1 = 0 ck2 = 1 rst = 0 in1 =   4 in2 =   0 out =   0
# time = 18 ck1 = 0 ck2 = 0 rst = 0 in1 =   4 in2 =   0 out =   0
# time = 19 ck1 = 1 ck2 = 0 rst = 0 in1 =   4 in2 =   0 out =   0
# time = 20 ck1 = 0 ck2 = 0 rst = 0 in1 =   2 in2 =   0 out =   0
# time = 21 ck1 = 0 ck2 = 1 rst = 0 in1 =   2 in2 =   0 out =   0
# time = 22 ck1 = 0 ck2 = 0 rst = 0 in1 =   2 in2 =   0 out =   0
# time = 23 ck1 = 1 ck2 = 0 rst = 0 in1 =   2 in2 =   0 out =   0
# time = 24 ck1 = 0 ck2 = 0 rst = 0 in1 =   1 in2 =   0 out =   0
# time = 25 ck1 = 0 ck2 = 1 rst = 0 in1 =   1 in2 =   0 out =   0
# time = 26 ck1 = 0 ck2 = 0 rst = 0 in1 =   1 in2 =   0 out =   0
# time = 27 ck1 = 1 ck2 = 0 rst = 0 in1 =   1 in2 =   0 out =   0
# time = 28 ck1 = 0 ck2 = 0 rst = 0 in1 = 255 in2 = 255 out =   0
# time = 29 ck1 = 0 ck2 = 1 rst = 0 in1 = 255 in2 = 255 out =   0
# time = 30 ck1 = 0 ck2 = 0 rst = 0 in1 = 255 in2 = 255 out =   0
# time = 31 ck1 = 1 ck2 = 0 rst = 0 in1 = 255 in2 = 255 out =   1
# time = 32 ck1 = 0 ck2 = 0 rst = 0 in1 = 255 in2 = 255 out =   1
# time = 33 ck1 = 0 ck2 = 1 rst = 0 in1 = 255 in2 = 255 out =   1
# time = 34 ck1 = 0 ck2 = 0 rst = 0 in1 = 255 in2 = 255 out =   1
# time = 35 ck1 = 1 ck2 = 0 rst = 0 in1 = 255 in2 = 255 out =   2
# time = 36 ck1 = 0 ck2 = 0 rst = 0 in1 = 255 in2 = 255 out =   2
# time = 37 ck1 = 0 ck2 = 1 rst = 0 in1 = 255 in2 = 255 out =   2
# time = 38 ck1 = 0 ck2 = 0 rst = 0 in1 = 255 in2 = 255 out =   2
# time = 39 ck1 = 1 ck2 = 0 rst = 0 in1 = 255 in2 = 255 out =   2
# time = 40 ck1 = 0 ck2 = 0 rst = 0 in1 = 255 in2 = 255 out =   2
# time = 41 ck1 = 0 ck2 = 1 rst = 0 in1 = 255 in2 = 255 out =   2
# time = 42 ck1 = 0 ck2 = 0 rst = 0 in1 = 255 in2 = 255 out =   2
# time = 43 ck1 = 1 ck2 = 0 rst = 0 in1 = 255 in2 = 255 out =   4
# time = 44 ck1 = 0 ck2 = 0 rst = 0 in1 = 255 in2 = 255 out =   4
# time = 45 ck1 = 0 ck2 = 1 rst = 0 in1 = 255 in2 = 255 out =   4
# time = 46 ck1 = 0 ck2 = 0 rst = 0 in1 = 255 in2 = 255 out =   4
# time = 47 ck1 = 1 ck2 = 0 rst = 0 in1 = 255 in2 = 255 out =   8
# time = 48 ck1 = 0 ck2 = 0 rst = 0 in1 = 255 in2 = 255 out =   8
# time = 49 ck1 = 0 ck2 = 1 rst = 0 in1 = 255 in2 = 255 out =   8
# time = 50 ck1 = 0 ck2 = 0 rst = 0 in1 = 255 in2 = 255 out =   8
# time = 51 ck1 = 1 ck2 = 0 rst = 0 in1 = 255 in2 = 255 out = 255
# time = 52 ck1 = 0 ck2 = 0 rst = 0 in1 = 255 in2 = 255 out = 255
# time = 53 ck1 = 0 ck2 = 1 rst = 0 in1 = 255 in2 = 255 out = 255
# time = 54 ck1 = 0 ck2 = 0 rst = 0 in1 = 255 in2 = 255 out = 255
# time = 55 ck1 = 1 ck2 = 0 rst = 0 in1 = 255 in2 = 255 out = 255
# time = 56 ck1 = 0 ck2 = 0 rst = 0 in1 = 255 in2 = 255 out = 255
# time = 57 ck1 = 0 ck2 = 1 rst = 0 in1 = 255 in2 = 255 out = 255
```

7.2 The First Global Loop: Generic ConnexArray^{TM}

A first attempt to close a loop over a simple cellular automaton is presented in [**?**]. The effect of a global loop on the behavior of a cellular automaton is presented in [Matita '13]. In [Gheolbanoiu '14] the attempt from [**?**] is finalized as an actual circuit.

In 1936 Stephen Kleene defined [Kleene '36] the concept of *partial recursive function* as the general

framework for computing any function of form:

$$f : \{0,1\}^n \rightarrow \{0,1\}$$

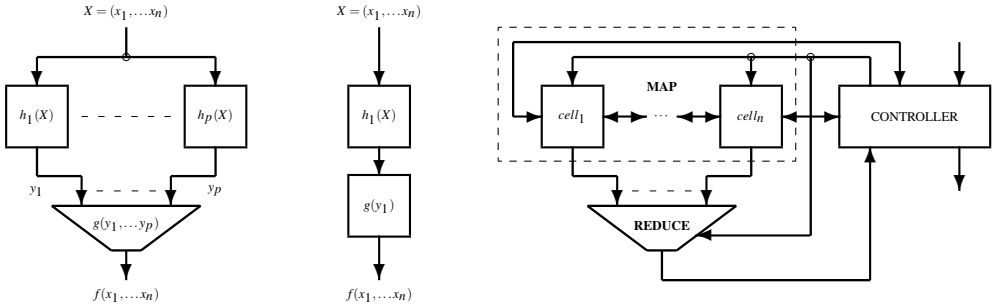

Figure 7.11: **From the mathematical model to the abstract machine model. a.** The structure associated to the composition rule in Kleene's model. **b.** The limit case for $p = 1$. It provides the pipelined connection which can be generalized for serially connected p cells. **c.** The abstract machine model for parallel computing. The MAP section consists of the parallel connected cells, the REDUCE section stands for the function g, while the serial connections between cells in MAP section provided by the serial pipelined connection for the limit case of $p = 1$.

In [**?**] is proved that from the three basic rules proposed by Kleene only the first, the composition rule, is independent. Therefore, computation could be defined as repeated application of the composition having the form:

$$f(x_1,\ldots x_n) = g(h_1(x_1,\ldots x_n),\ldots h_p(x_1,\ldots x_n))$$

In Figure 7.11a the two-level circuit version of the composition rule is represented. Each function h_i is computed by a module on the first level, while the function g reduces the resulting vector to a scalar. In Figure 7.11b, the limit case for $p = 1$ is represented. The repeated application of a composition requests the additional structures represented in Figure 7.11c. In [Stefan '14] the transition from Figure 7.11a and Figure 7.11b to Figure 7.11c is described.

The two-direction connections between the cells provide the $p = n$ levels of loops which gives the order n to the system, while the global loop is closed through the CONTROLLER.

The simplest version of the engine is behaviorally described in the next subsection as the **Generic *ConnexArray*^{TM}** system. Some temporal aspects are not catched in the following description because we will be focused only on the functional aspects. A structural description takes into account at least the pipelines used to optimize the clock frequency. Also, some aspects related with data transfer are treated in this behavioral description ignoring the timing issues.

7.2.1 The behavioral description of Generic ConnexArray^{TM}

The cell used in the generic n-order array with the first global loop contains a data memory for the local data and a simple accumulator-based engine.

The cell's structure is presented in Figure 7.12a, while the controller's structure is presented in Figure 7.12b.

The behavioral description uses the storage resources detailed in the file `ConnexArray.v` represented in Figure 7.14, where:

vectorial resources describes the resources distributed in array (for the contribution of each cell see Figure 7.13a)

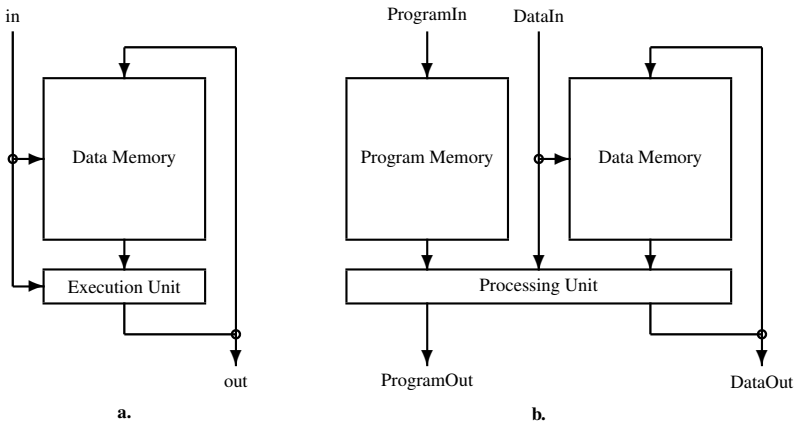

Figure 7.12: **The components of the abstract machine model. a.** The cell's structure. **b.** The controller's structure.

ixv : index vector used to associate the index i, from 0 to $2^x - 1$, to each cell

boolv : Boolean vector used to enable the execution in i-th cell; if `boolv[i]` is 1, then the cell is active, else the instruction received in the current cycle is ignored (substituted with `nop`)

accv : is the scalar vector containing the accumulator registers of the cells; the execution unit is accumulator based, thus `accv[i]` is the accumulator of the cell i

crv : is the Boolean vector containing the carry registers of each cell; `crv[i]` stores the carry bit generated in cell i by the last arithmetic operation

vmem : is the vector memory distributed along the cells; each cell, $cell_i$, stores in its local memory the i-th components of all the 2^v vectors

addrv : used to specify a locally computed address in each cell

control resources describes the resources involved in the sequential control (see Figure 7.13b)

pc : p-bit program counter

ir : 32-bit instruction register

progMem : the program memory organize in 2^p 32-bit words

scalar resources describes the resources involved in the scalar computation (see Figure 7.13b)

acc : controller's accumulator

cr : the carry flip-flop

addr : the address register used to compute the address for controller's data memory

mem : controller's data memory

The instruction read from the program memory in each clock cycle is of the following form:

```
instruction =
{arrayInstr, contrInstr} =
```

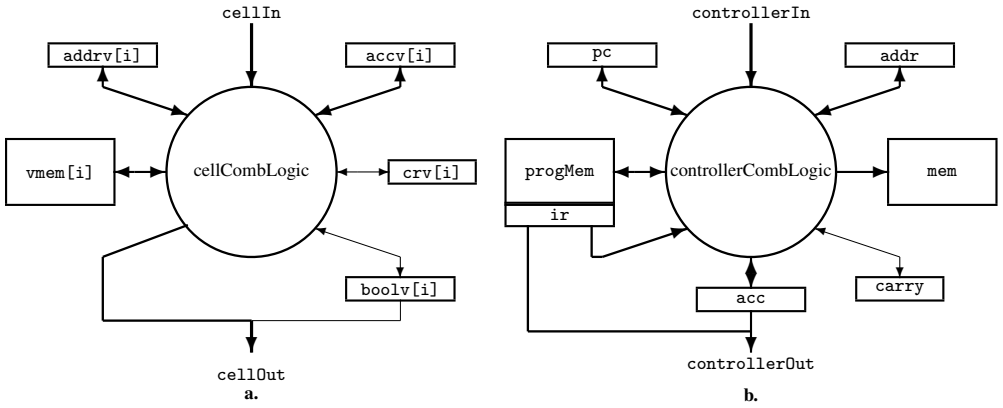

Figure 7.13: **The resources used in the behavioral description. a.** Cell's internal state support. **b.** Controller's internal state support.

```
{aOpCode[4:0], // operation code for the array
 aOpr[2:0]   , // selection operand for array
 aval[7:0]   , // immediate value for array
 cOpCode[4:0], // operation code for controller
 cOpr[2:0]   , // selection operand for controller
 cval[7:0]   } // immediate value for controller
```

The input received by each cell (see Figure 7.12a and Figure 7.13a) is:

```
in = {instruction[15:0], data[n-1:0], address[v-1:0]}
```

while the output is:

```
out = {boolv[i], (boolv[i] ?  accv[i][n-1:0] :  0)}
```

From the program memory, in each cycle is read a pair of instructions: one (`progMem[nextPc][15:0]`) for the use of controller and another (`progMem[nextPc][31:16]`) to be executed in each active cell (where `boolv[i] = 1`). The structure of the two instructions is detailed also in Figure 7.14.

```
/*************************************************************************
File name:      ConnexArray.v
Circuit name:   Generic Connex Array
Description:    behavioral description for simulation; the content of data
                memory and program memory are generated in the simulation
                environment
*************************************************************************/
module ConnexArray #(`include "parameters.v")(input reset, clock);
// control resources
    reg [p-1:0] pc                              ; // program counter
    reg [31:0]  ir                              ; // instruction register
    reg [31:0]  progMem[0:(1<<p)-1]             ; // program memory
// scalar resources
    reg [n-1:0] acc                             ; // scalar accumulator
    reg         cr                              ; // scalar carry
    reg [s-1:0] addr                            ; // scalar address
    reg [n-1:0] mem[0:(1<<s)-1]                 ; // scalar memory
// vector resources
    reg         bool[0:(1<<x)-1]                ; // Boolean vector
    reg [n-1:0] accv[0:(1<<x)-1]                ; // accumulator vector
    reg         crv[0:(1<<x)-1]                 ; // carry vector
    reg [v-1:0] addrv[0:(1<<x)-1]               ; // address vector
    reg [n-1:0] vmem[0:(1<<x)-1][0:(1<<v)-1]; // vector memory
// structure of the instructions for array and for controller
    wire    [4:0] aOpCode       ; // operation code for the array
    wire    [2:0] aOpr          ; // selection operand for array
    wire    [7:0] aval          ; // immediate value for array
    wire    [4:0] cOpCode       ; // operation code for controller
    wire    [2:0] cOpr          ; // selection operand for controller
    wire    [7:0] cval          ; // immediate value for controller
    assign aOpCode  = ir[31:27] ;
    assign aOpr     = ir[26:24] ;
    assign aval     = ir[23:16] ;
    assign cOpCode  = ir[15:11] ;
    assign cOpr     = ir[10:8]  ;
    assign cval     = ir[7:0]   ;
// behavior of the system
    integer i   ;
    `include "programControl.v"
    `include "cOperandSel.v"
    `include "cDataOperations.v"
    `include "spatialControl.v"
    `include "aOperandSel.v"
    `include "aDataOperations.v"
    `include "vectorTransfer.v"
endmodule
```

Figure 7.14: Generic Connex Array described in the file `ConnexArray.v`.

The Instruction Set Architecture

```
/*****************************************************************************
File name:      parameters.v
Description:    defines Instruction Set Architecture for Generic Connex Array
*****************************************************************************/
    parameter   n = 32   , // word size
                x = 4    , // index size
                v = 8    , // vector memory address size
                s = 8    , // scalar memory address size
                p = 8    , // program memory address size
/*****************************************************************************
opCode: selects the right operand. The architecture is accumulator based
*****************************************************************************/
    val = 3'b000, // immediate value: {24{scalar[7]}}, scalar}
    mab = 3'b001, // absolute: mem[scalar]
    mrl = 3'b010, // relative: mem[addr+scalar]
    mri = 3'b011, // relative & increment: mem[addr+scalar]; addr <= addr+scalar
    cop = 3'b100, // co-operand
    ctl = 3'b111, // control operations
/*****************************************************************************
Instruction Set Architecture
*****************************************************************************/
    add         = 5'b00000, // {cr, acc} <= acc + op;
    addc        = 5'b00001, // {cr, acc} <= acc + op + cr;
    sub         = 5'b00010, // {cr, acc} <= acc - op;
    rsub        = 5'b00011, // {cr, acc} <= operand - acc;
    subc        = 5'b00100, // {cr, acc} <= acc - op - cr;
    rsubc       = 5'b00101, // {cr, acc} <= op - acc - cr;
    mult        = 5'b00110, // acc <= acc * op;
    load        = 5'b00111, // acc <= op;
    store       = 5'b01000, // op <= acc;
    bwand       = 5'b01001, // acc <= acc & op;
    bwor        = 5'b01010, // acc <= acc | op;
    bwxor       = 5'b01011, // acc <= acc ^ op;
    insval      = 5'b01100, // acc <= {acc[23:0], scalar}
    shrightc    = 5'b01101, // {cr, acc} <= {acc[0], cr, acc[n-1:1]}
    shright     = 5'b01110, // {cr, acc} <= {acc[0], 1'b0, acc[n-1:1]}
    sharight    = 5'b01111, // acc <= {acc[n-1], acc[n-1:1]}
                            // ONLY FOR CONTROLLER
    jmp         = 5'b10000, // pc <= pc + scalar;
    brz         = 5'b10001, // pc <= acc=0 ? pc + scalar : pc + 1;
    brnz        = 5'b10010, // pc <= acc=0 ? pc + 1 : pc + scalar;
    brzdec      = 5'b10011, // pc <= acc=0 ? pc + scalar : pc + 1; acc <= acc - 1
    brnzdec     = 5'b10100, // pc <= acc=0 ? pc + 1 : pc + scalar; acc <= acc - 1
                            // ONLY FOR ARRAY
    where       = 5'b10000, // bool <= condv[operand] ? 1 : 0;
    elsew       = 5'b10001, // bool <= ~boolVect;
    endwhere    = 5'b10010, // bool <= 1;
    ixload      = 5'b10011, // acc <= i
    gshift      = 5'b11001, // acc[i] <= acc[i+/-1];
    vload       = 5'b11010, // accv[i] <= mem[addr + i]
    vstore      = 5'b11011  // mem[addr + i] <= accv[i]
```

Figure 7.15: Instruction Set Architecture described in the file `parameters.v`.

The arithmetic-logic operations performed in each cell and in the controller are very similar. Only

the sequential control in controller and the spatial control in the array of cells differentiate the instruction set architecture (ISA) (see Figure 7.15) which describes the functions of the controller and of the cells. The instructions are specified by three fields (see Figure 7.14 for the structure of the instruction): one for operation (`opCode`), one for the right operand (`operand`), because the left operand is always the accumulator, and the last for an 8-bit immediate value.

Program Control Section

The program control section of the controller works as it is described in Figure 7.16:

```
/*********************************************************************
File name:      programControl.v
Description:    the code manages the value of the program counter (pc)
*********************************************************************/
    reg [p-1:0] nextPc  ;

    always @(*) if (cOpr == ctl)
     case(cOpCode)
       jmp     : nextPc = pc + cval[p-1:0]                              ;
       brz     : nextPc = (acc == 0) ? (pc + cval[p-1:0]) : (pc + 1'b1);
       brnz    : nextPc = (acc == 0) ? (pc + 1'b1) : (pc + cval[p-1:0]);
       brzdec  : nextPc = (acc == 0) ? (pc + cval[p-1:0]) : (pc + 1'b1);
       brnzdec : nextPc = (acc == 0) ? (pc + 1'b1) : (pc + cval[p-1:0]);
       default : nextPc = pc + cval[p-1:0]                              ;
     endcase
     else        nextPc = pc + 1'b1                                     ;

    always @(posedge clock) if (reset)  begin  pc  <= {p{1'b1}}        ;
                                               ir  <= 0                ;
                                        end
                            else        begin  pc  <= nextPc           ;
                                               ir  <= progMem[nextPc]  ;
                                        end
```

Figure 7.16: The file `programControl.v` describes the control function for Controller.

Operand selection in controller

Data operations in controller

Data operations in controller is performed using as operands the accumulator, `acc`, and data selected by `contrOperand`, as described in Figure 7.18.

```
/*********************************************************************
File name:      cOperandSel.v
Description:    the code selects the right operand for Controller
*********************************************************************/
    reg [n-1:0] op  ;

    always @(*)
        case(cOpr)  // selects the right operand for controller
            mrl:                op = mem[addr + cval[s-1:0]]          ;
            mri:                op = mem[addr + cval[s-1:0]]          ;
            val:                op = {{(n-8){cval[7]}}, cval}         ;
            cop: case(cval[1:0])
                    2'b00:  begin
                                op = accv[0]                          ;
                            for (i=1; i<(1<<x); i=i+1)
                                op = op + accv[i]                     ;
                            end
                    2'b01:  begin
                            op = accv[0]                              ;
                            for (i=1; i<(1<<x); i=i+1)
                                op = (op < accv[i]) ? accv[i] : op    ;
                            end
                    2'b10:  begin
                                op = {{(n-1){1'b0}}, bool[0]}         ;
                            for (i=1; i<(1<<x); i=i+1)
                                op = {{(n-1){1'b0}}, op[0] | bool[i]};
                            end
                    default:    op = 0                                ;
                endcase
            default:            op = mem[cval[s-1:0]]                 ;
        endcase
```

Figure 7.17: The code used to select the right operand in Controller: `cOperandSel.v`

Spatial control in array

```
/*********************************************************************
File name:      spatialControl.v
Description:    describes the spatial control in Array
*********************************************************************/
    reg [3:0]   condv[0:(1<<x)-1]   ;

    always @(*) for (i=0; i<(1<<x); i=i+1)
        condv[i] = {!crv[i], (accv[i] !== 0), crv[i], (accv[i]  == 0)};

    always @(posedge clock) for (i=0; i<(1<<x); i=i+1)
        case(aOpCode)
            where   : bool[i] <= (condv[i][aval[1:0]]) ? 1'b1 : 1'b0;
            elsew   : bool[i] <= ~bool[i]                           ;
            endwhere: bool[i] <= 1'b1                               ;
        endcase
```

Figure 7.19: File `spatialControl.v` which describes the spatial control functions in Array.

Operand selection in the array's cells

Operand selection in the array's cells is described by the code from Figure 7.20:

```
/*********************************************************************
File name:      aOpSelection.v
Description:    describes the right operand selection for Array
*********************************************************************/
    reg [n-1:0] opv[0:(1<<x)-1]     ;

    always @(*) for (i=0; i<(1<<x); i=i+1)
        case(aOpr) // selects the right operand in each cell
            mrl:     opv[i] = vmem[i][addrv[i] + aval[v-1:0]]   ;
            mri:     opv[i] = vmem[i][addrv[i] + aval[v-1:0]]   ;
            val:     opv[i] = {{(n-8){aval[7]}}, aval}          ;
            cop:     opv[i] = acc                               ;
            default: opv[i] = vmem[i][aval[v-1:0]]              ;
        endcase
```

Figure 7.20: The file `aOpSelection.v` describes the right operand selection for Array.

Data operations in the array's cells

Data operations in the array's cells is performed using as operands the accumulator, `accVect[i]`, and data selected by `arrayOperand`, as shown in Figure 7.21.

Vector transfer

instructions are used to exchange data between the vector memory distributed in the array of cell and the controller's memory. The transfer is strided with stride given by `contrScalar`. The definition is in Figure 7.22

7.2.2 Assembler Programming the Generic ConnexArray^{TM}

Each line of program must contain code for both instructions: the instruction issued for the array and the instruction performed by the controller. For conditioned or unconditioned relative jumps in program some lines are labeled; `LB(i)` denote the label *i*. The use of the label is indicated by by the value used by control instructions (example: `cJMP(2)`).

Example 7.6 *The program which compute in the accumulator of the controller the inner product of the index vector* `ix` *with itself is:*

```
cNOP;       ENDWHERE;     // activate all cells
cNOP;       IXLOAD;       // accVect[i] <= ixVect[i]
cNOP;       IXMULT;       // accVect[i] <= accVect[i] * ixVect[i]
cRSLOAD;    NOP;          // load acc with the reduction sum
cHALT;      NOP;
```

The content of program memory is:

```
programMemory[0] = 10010111000000000000000000000000
programMemory[1] = 10011000000000000000000000000000
programMemory[2] = 01000001000000000000000000000000
programMemory[3] = 00110001000000000000000000000000
programMemory[4] = 00000000000000000111110000000000
programMemory[5] = 00000000000000001000011100000000
```

The result of simulation:

```
t=0  reset=1 pc=x   acc=   x ACC = [x, x, x, ... x]   b = [xxxxxxxxxxxxxxxx]
t=1  reset=1 pc=255 acc=   x ACC = [x, x, x, ... x]   b = [xxxxxxxxxxxxxxxx]
t=4  reset=0 pc=255 acc=   x ACC = [x, x, x, ... x]   b = [xxxxxxxxxxxxxxxx]
t=5  reset=0 pc=0   acc=   x ACC = [x, x, x, ... x]   b = [xxxxxxxxxxxxxxxx]
t=7  reset=0 pc=1   acc=   x ACC = [x, x, x, ... x]   b = [1111111111111111]
t=9  reset=0 pc=2   acc=   x ACC = [0, 1, 2, ... 15]  b = [1111111111111111]
t=11 reset=0 pc=3   acc=   x ACC = [0, 1, 2, ... 15]  b = [1111111111111111]
t=13 reset=0 pc=4   acc=   x ACC = [0, 1, 4, ... 225] b = [1111111111111111]
t=15 reset=0 pc=5   acc=1240 ACC = [0, 1, 4, ... 225] b = [1111111111111111]
```

◇

Example 7.7 *The program which load the accumulator the index in each cell, stores it incremented in 12 successive addresses starting from the address 2, than add in accumulator the stored values. The program is:*

```
        cVLOAD(12);     ENDWHERE;     // acc <= 12; activate all cells
        cNOP;           VLOAD(2);     // accVect[i] <= 2
        cNOP;           ADDRLD;       // addrVect[i] <= accVect[i]
        cNOP;           IXLOAD;       // accVect[i] <= index
LB(1);  cNOP;           RISTORE(1);   // vectMem[i][addrVect + 1] <= accVect[i];
                                      // addrVect <= addrVect + 1
        cBRNZDEC(1);    VADD(1);      // if (acc !== 0) branch to LB(1); acc<=acc-1;
                                      // accVect[i] <= accVect[i] + 1;
        cVLOAD(13);     VLOAD(2);     // acc <= 13; accVect[i] <= 2
        cNOP;           ADDRLD;       // addrVect[i] <= accVect[i]
        cNOP;           VLOAD(0);     // accVect[i] <= 0
LB(2);  cBRNZDEC(2);    RIADD(1);     // if (acc !== 0) branch to LB(2); acc<=acc-1;
                                      // accVect[i] <=
                                      // accVect[i] + vectMem[i][addrVect + 1];
                                      // addrVect <= addrVect + 1
        cHALT;          NOP;          // halt
```

The result of simulation is:

```
t=97 ACC= [78,91,104,117,130,143,156,169,182,195,208,221,234,247,260,273]

vect[4] = 1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
vect[5] = 2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17
```

```
vect[6]  =  3  4   5   6   7   8   9   10  11  12  13  14  15  16  17  18
vect[7]  =  4  5   6   7   8   9   10  11  12  13  14  15  16  17  18  19
vect[8]  =  5  6   7   8   9   10  11  12  13  14  15  16  17  18  19  20
vect[9]  =  6  7   8   9   10  11  12  13  14  15  16  17  18  19  20  21
vect[10] =  7  8   9   10  11  12  13  14  15  16  17  18  19  20  21  22
vect[11] =  8  9   10  11  12  13  14  15  16  17  18  19  20  21  22  23
vect[12] =  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24
vect[13] = 10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25
vect[14] = 11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26
vect[15] = 12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27
```

◇

7.3 The Second Global Loop: Search Oriented Generic ConnexArray^{TM}

The global loop closed in the previous section sends back to the array the same data for each cell. A new feature is added when another loop sends back to the array specific, differentiated information for each cell. Let us start with a very simple function associated to this second global loop: it takes the Boolean vector `boolVect[0:(1<<x)-1]` distributed along the array of cells and sends back two Boolean vectors:

- `firstVect[0:(1<<x)-1]`: with 1 only on the position of the first occurrence of 1 in `boolVect`; it is used to indicate the first active cell
- `nextVect[0:(1<<x)-1]`: with 1 in all the positions next to the 1 in the `firstVect` Boolean vector; it is used to indicate all the cells next to the first active cell

There are 5 additional instructions supported by this second global loop. In the file `parameters.v` the following 5 lines are added:

```
...
    search  = 5'b10100, // b[i] <= (acc[i] == op) ? 1 : 0
    csearch = 5'b10101, // b[i] <= (acc[i] == op) && b[i-1] ? 1 : 0
    insert  = 5'b10110, // acc[first] <= op; acc[next] <= acc[i-1]
    delete  = 5'b10111, // acc[first || next] <= acc[i+1]
    read    = 5'b11000, // b[i] <= b[i-1]
...
```

In the file `ConnexArray.v` the following line is added:

```
...
    `include "searchOperations.v"
...
```

where the `searchOperations.v` file is shown in Figure 7.23

The instruction `search` identifies all the positions in array where the accumulator has a certain value, while the `csearch` supports the search of a certain string of values.

Example 7.8 *The program which load the index in each* `acc[i]`*, identifies the occurrence of the stream* `<1 2 3>` *in* `accVect` *and adds in* `acc` *the next four numbers:*

```
cNOP;         ENDWHERE; // set active all cells
cVLOAD(1);    IXLOAD;   // acc = 1; load index in each cell
cVLOAD(2);    CSEARCH;  // acc = 2; search 'acc' in each acc[i]
cVLOAD(3);    CSEARCH;  // acc = 3; search 'acc' after each active cell
cNOP;         CSEARCH;  // search 'acc' after each active cell
cNOP;         READ;     // boolVect >> 1
cCLOAD(0);    READ;     // acc = reduceSum; boolVect >> 1
cCADD(0);     READ;     // acc = acc + reduceSum; boolVect >> 1
cCADD(0);     READ;     // acc = acc + reduceSum; boolVect >> 1
cCADD(0);     NOP;      // acc = acc + reduceSum
cHALT;        NOP;      // halt
```

```
pc=0  acc=  x ACC = [x,x,x,x,x,x,x,x,x,x,x, x, x, x, x, x ] b = [xxxxxxxxxxxxxxxx]
pc=1  acc=  x ACC = [x,x,x,x,x,x,x,x,x,x,x, x, x, x, x, x ] b = [1111111111111111]
pc=2  acc=  1 ACC = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15] b = [1111111111111111]
pc=3  acc=  2 ACC = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15] b = [0100000000000000]
pc=4  acc=  3 ACC = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15] b = [0010000000000000]
pc=5  acc=  3 ACC = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15] b = [0001000000000000]
pc=6  acc=  3 ACC = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15] b = [0000100000000000]
pc=7  acc=  4 ACC = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15] b = [0000010000000000]
pc=8  acc=  9 ACC = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15] b = [0000001000000000]
pc=9  acc= 15 ACC = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15] b = [0000000100000000]
pc=10 acc= 22 ACC = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15] b = [0000000100000000]
```

◇

Example 7.9 *The program which load the index in each* `acc[i]`*, identifies the occurrence(s) of the stream* `<0 1>` *in* `accVect` *and insert in the vector* `accVect` *the sequence* `<13 15>`*.*

```
cNOP;  ENDWHERE;     // set active all cells
cNOP;  IXLOAD;       // load index in each cell
cNOP;  VSEARCH(1);   // search '0' in each acc[i]
cNOP;  VCSEARCH(2);  // search '1' after each active cell
cNOP;  READ;         // boolVect >> 1
cNOP;  INSERT(13);   // insert '13' in the first active cell
cNOP;  INSERT(15);   // insert '15' in the first active cell
cHALT; NOP;          // halt
```

◇

The second global loop adds specific features which support search applications, sparse matrix/vector operations, ...

```
/*****************************************************************
File name:      cDataOperations.v
Description:    the code describes the data operations in Controller
*****************************************************************/
    always @(posedge clock)
        case(cOpCode)
            add     : {cr, acc} <= acc + op                         ;
            addc    : {cr, acc} <= acc + op + cr                    ;
            sub     : {cr, acc} <= acc - op                         ;
            rsub    : {cr, acc} <= op - acc                         ;
            subc    : {cr, acc} <= acc - op - cr                    ;
            rsubc   : {cr, acc} <= op - acc - cr                    ;
            mult    : {cr, acc} <= {cr, acc * op}                   ;
            load    : {cr, acc} <= {cr, op}                         ;
            store   : case(cOpr)
                        mab : mem[cval[s-1:0]]              <= acc  ;
                        mri : mem[cval[s-1:0] + addr]       <= acc  ;
                        mri : begin mem[cval[s-1:0] + addr] <= acc  ;
                                    addr <= cval[s-1:0] + addr      ;
                              end
                        val : addr <= acc[s-1:0]                    ;
                        default: addr <= acc[s-1:0]                 ;
                      endcase
            bwand   : {cr, acc} <= {cr, acc & op}                   ;
            bwor    : {cr, acc} <= {cr, acc | op}                   ;
            bwxor   : {cr, acc} <= {cr, acc ^ op}                   ;
            insval  : {cr, acc} <= {cr, acc[23:0], op[7:0]}         ;
            shrightc: {cr, acc} <= {acc[0], cr, acc[n-1:1]}         ;
            shright : {cr, acc} <= {acc[0], 1'b0, acc[n-1:1]}       ;
            sharight: {cr, acc} <= {acc[0], acc[n-1], acc[n-1:1]}   ;
            brzdec  : {cr, acc} <= acc - 1'b1                       ;
            brnzdec : {cr, acc} <= acc - 1'b1                       ;
        endcase
```

Figure 7.18: The file describes the data operations performed in Controller: `cDataOperations.v`

```
/*************************************************************************
File name:      aDataOperations.v
Description:    describes the data operations in Array
*************************************************************************/
    always @(posedge clock) for (i=0; i<(1<<x); i=i+1)
    if (bool[i]) begin
     if (aOpr == mri)  addrv[i] <= addrv[i] + aval[v-1:0]                    ;
     case(aOpCode)
        add     : {crv[i], accv[i]} <= accv[i] + opv[i]                      ;
        addc    : {crv[i], accv[i]} <= accv[i] + opv[i] + crv[i]             ;
        sub     : {crv[i], accv[i]} <= accv[i] - opv[i]                      ;
        rsub    : {crv[i], accv[i]} <= opv[i] - accv[i]                      ;
        subc    : {crv[i], accv[i]} <= accv[i] - opv[i] - crv[i]             ;
        rsubc   : {crv[i], accv[i]} <= opv[i] - accv[i] - crv[i]             ;
        mult    : {crv[i], accv[i]} <= {crv[i], accv[i] * opv[i]}            ;
        load    : {crv[i], accv[i]} <= {crv[i], opv[i]}                      ;
        store   : case(aOpr)
                    mab : vmem[i][aval[v-1:0]] <= accv[i]                    ;
                    mrl : vmem[i][aval[v-1:0] + addrv[i]] <= accv[i]         ;
                    mri : vmem[i][aval[v-1:0] + addrv[i]] <= accv[i]         ;
                    val : addrv[i] <= accv[i][v-1:0]                         ;
                    default addrv[i] <= addrv[i]                             ;
                  endcase
        bwand   : {crv[i], accv[i]} <= {crv[i], accv[i] & opv[i]}            ;
        bwor    : {crv[i], accv[i]} <= {crv[i], accv[i] | opv[i]}            ;
        bwxor   : {crv[i], accv[i]} <= {crv[i], accv[i] ^ opv[i]}            ;
        insval  : {crv[i], accv[i]} <= {crv[i], accv[i][23:0], opv[i][7:0]}  ;
        gshift  : accv[i] <= opv[i][0] ? (i == ((1<<x)-1) ? 0 : accv[i+1]) :
                                         (i == 0 ? 0 : accv[i-1])            ;
        shrightc: {crv[i], accv[i]} <= {accv[i][0], crv[i], accv[i][n-1:1]}
;
        shright : {crv[i], accv[i]} <= {accv[i][0], 1'b0, accv[i][n-1:1]}    ;
        sharight: {crv[i], accv[i]} <=
                                      {accv[i][0], accv[i][n-1], accv[i][n-1:1]};
        vload   : vmem[i][accv[i]] <= mem[acc + i*cval]                      ;
        ixload  : {crv[i], accv[i]} <= i                                     ;
     endcase
    end
```

Figure 7.21: The file `aDataOperations.v` describes data operations in the array of cells.

```
/*************************************************************************
File name:      vectorTransfer.v
Description:    describes the vector transfer operations
*************************************************************************/
    always @(posedge clock) for (i=0; i<(1'b1<<x); i=i+1) begin
            if (aOpCode == vload)   accv[i]        <= mem[addr + i];
            if (aOpCode == vstore)  mem[addr + i]  <= accv[i]      ;
        end
```

Figure 7.22: The file `vectorTransfer.v` describes the vector transfer operations between array of cells and Controller's data memory.

```
/*********************************************************************
File name:      searchOperations.v
Description:    describes the search operations in array
*********************************************************************/
    reg px[0:(1<<x)-1]      ;
    reg first[0:(1<<x)-1]   ;
    reg next[0:(1<<x)-1]    ;
// The scan loop
    always @(*) for (i=0; i<(1<<x); i=i+1) begin
            px[i]       = (i == 0) ? bool[0] : (bool[i] | px[i-1])  ;
            first[i]    = (i == 0) ?   px[0] : (px[i] & ~px[i-1])   ;
            next[i]     = (i == 0) ?    1'b0 : px[i-1]              ;
        end

    always @(posedge clock) for (i=0; i<(1<<x); i=i+1)
        case(aOpCode)
            search  : bool[i] <= (accv[i] == opv[i]) ? 1'b1 : 1'b0  ;
            csearch : bool[i] <= (i == 0) ? 1'b0 :
                        (((accv[i] == opv[i]) & bool[i-1]) ? 1'b1 :
                        1'b0)                                       ;
            read    : bool[i] <= (i==0) ? 0 : bool[i-1]             ;
            insert  : accv[i] <= first[i] ? opv[i] : (next[i] ?
                        accv[i-1] : accv[i])                        ;
            delete  : accv[i] <= (i == (1<<x)-1) ? 0 :
                        ((first[i] | next[i]) ? accv[i+1] : accv[i]);
        endcase
```

Figure 7.23: File `searchOperations.v` describes the search operations in the array of cells.

Chapter 8

Recursive Hierarchy

The process of adding new functionality to a cellular system is recursive. The hierarchy can no longer be based on additional loops, and increasing the size beyond a certain limit produces systems that are difficult to control. For these reasons we are obliged to adopt recursive structuring. The resulting structural organization will allow a hierarchically distributed control based on a hierarchy of function libraries.

The cellular structures presented in the previous chapter lend themselves to a recursive hierarchical structuring that leads to a hierarchy of memories that attenuates "von Neumann Bottleneck". By associating the hierarchically distributed memory modules with local execution / processing units, we further reduce what is more recently designated by "Turing tariff".

The recursive hierarchy that we will introduce in this chapter naturally highlights the HOST / ACCELERATOR duality imposed by the segregation of intense computation from the complex one. At the top of the hierarchy will always be HOST the computer that will "see" everything that floated underneath like an ACCELERATOR.

8.1 Integrating ConnexArray^{TM} as Accelerator in a Computing System

Integrating the generic version of ConnexArray^{TM} as accelerator means to add interfaces and mechanisms to transfer data in and out to/form the memories defined in ConnexArray. The program memory of the Controller must be loaded and the data memory of controller and the vector memory distributed along the cells must communicate with the external system memory. Besides these, the host processor must be able to activate the functions of the accelerator and to receive the minimal information back in the form of an interrupt.

In Figure 8.1 is shown how ConnexArray^{TM} is used to design an accelerator for a general purpose computing system. The interface must support the following communication facilities:

- program load: the program memory, `progMem` (see Figure 7.13b), of the controller is loaded with the program(s); the interface signals are:
  - `fromHost[31:0]`: receives programs (stream of instructions) and calls (functions and, if needed, parameters)
  - `write` and `ready`: dialog signals
- data transfer: the data memory of the controller, `mem` (see Figure 7.13b), and the memory distributed along the cells, `vectorMem[i]` (see Figure 7.13a), exchange data with the system memory of the host; the interface signals are:

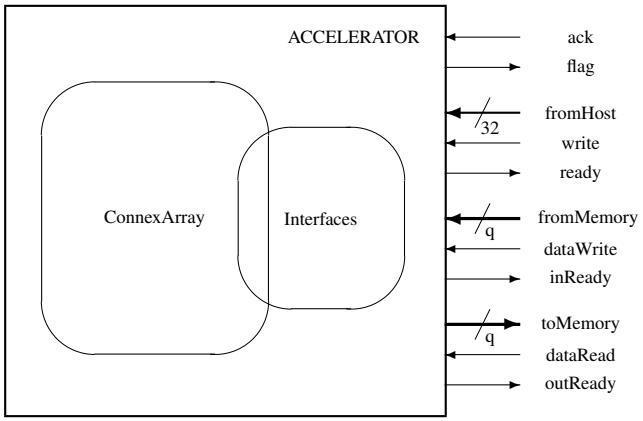

Figure 8.1: Integrating ConnexArray^{TM} with a host means to add an interface for data, programs and commends.

  - `fromMemory[q-1:0]`: receives data form the system memory; it is recommended $q = m \times n$ with m as big as possible to attenuate the effect of the "von Neumann Bottleneck" (usually, the range is $m = 4 \div 12$)
  - `dataWrite` and `inReady`: dialog signals
  - `toMemory[q-1:0]`: sends data to the system memory
  - `dataRead` and `outReady`: dialog signals

- control: the host processor calls the functions to be accelerated by starting the run of programs loaded in controller's program memory. The command is transferred through `fromHost` port. The synchronization is done using the signals:
  - `flag`: is the flag sent back by the accelerator, if needed, to notify the end of a process
  - `ack`: acknowledges the receiving of the `flag` signal

8.2 ACCELERATOR as a Recursive Structured Parallel Engine

The generalized form of ConnexArray is presented as a recursive structure of the type shown in Figure 8.2, where:

- MAP: a finite linear array of p cells. Each cell executes, according to its internal state, the command (instruction, function) issued by the Control unit. It consists from a data (and program) memory, *mem*, and a computing engine, *eng*.
- REDUCE(i): a log-depth tree network of functional units used to reduce a n-dimension data structure to a $(n-1)$-dimension data structure.
- SCAN(i): a log-depth 2-dimension network of functional units which takes from MAP a vector and sends back as the result, for example, a permutation or a prefix computation.
- Control(i): is a mono/multi-core computing unit, with its own data and program memory, which issues commands for the MAP array and receives back from the array a result of the reduction.

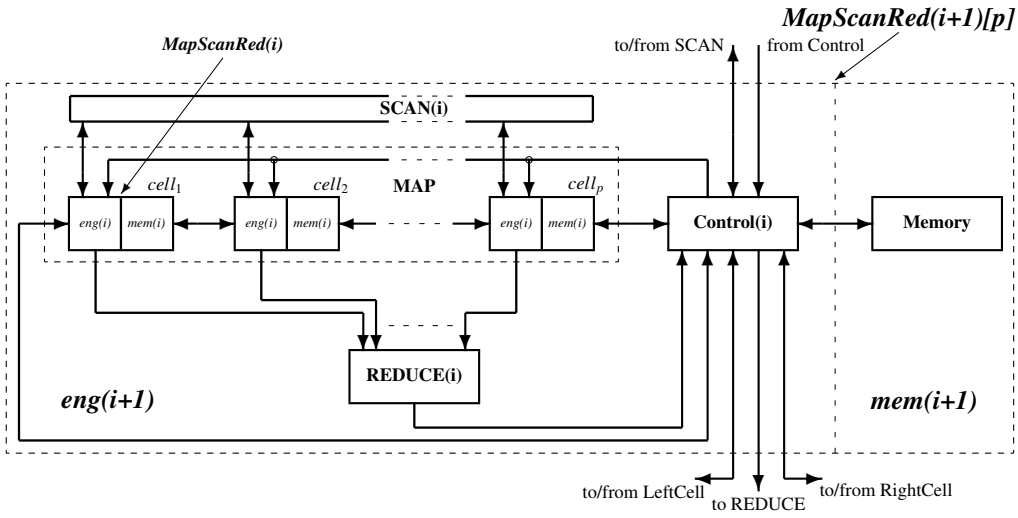

Figure 8.2: ***MapScanReduce recursive abstract model* for parallel computation.**

- Memory: stores data (and programs) for the entire system. The transfer between Memory and the local memories $mem(i)$, distributed in the array, is supervised by the Control unit.

For **MapScanReduce(1)[p]**:

- $eng(0)$ is an execution unit for 8, 16, or 32 bits words (floating-point operations are performed as a sequence of integer operations); each execute, according to its internal state, the instructions issued by the controller in each cycle
- $mem(0)$ is a local data memory of 1 to 16 KB
- **Control(0)** is a computing engine ($eng(0) + mem(0) + programMemory$) which fetches from its *programMemory* a pair of instructions, one for the controller and on to be issued to MAP array
- **REDUCE(0)** is a log-depth circuit performing ADD, MIN, MAX
- **SCAN(0)** is a log-depth circuit performing prefix and permute functions
- **Memory** is the external memory containing data and programs

Example 8.1 *The recursive hierarchy is exemplified for a 2-level implementation. On the second level $p = 4$, while on the first level $p = 256$. According to our notation we have a* ***MapScanReduce(2)[4][256]****. The system is represented in Figure 8.3, where:*

- **Control(1)** *is an ARM processor*
- $eng(1)$ *is ConnexArray*^{TM} *with 256 cells, i.e.* **MapScanReduce(1)[256]**
- $mem(1)$ *is a SDRAM of 1GB*

The lowest level is programmed in assembly language. It provides for the second level a **kernel library** *of functions used in programs written in a high level language to develop a library of functions. Thus the program running on the ARM processor sees the accelerator as a hardware implemented library of functions.*

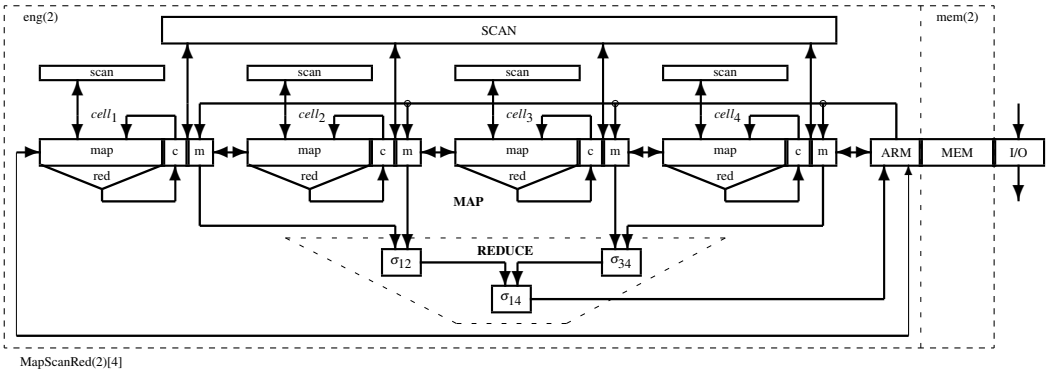

Figure 8.3: ***MapScanRed(2)[4]*** with ***MapScanRed(1)[256]***, i.e., ***MapScanReduce(2)[4][256]***.

A mix solution is possible, because in a certain function library there are functions that do not benefit from a parallel implementation. Therefore, part of the library is kept in its original form as an optimized program to run on the host processor. For example, the inner product (scalar product) of two vectors does not benefit substantially from a parallel implementation because the transfer time is in the same order of magnitude as the sequential execution time.

◇

8.3 Programming Recursive Structured Parallel Engine

The structural hierarchy corresponds to a functional hierarchy that is directly reflected in the structure of the programs at each level. For $i > 0$ the **Control(i)** computer is programmed in a high level language (Python, C ++) which calls the accelerated functions on the lower level. The functions on the lower level are organized in a function library. For **Control(0)**, programming is done in the assembler, as is done to accelerate any function library designed for any current processor.

```
def functionName(paramter1, ..., parameterN)
    # here comes statement 1
    # ...
    # here comes statement M
```

Example 8.2 *Be the simplest version of the recursive hierarchy:* ***MapScanReduce(2)(1)(128)****. A part of a small & simple linear algebra kernel function library for the use of a Python program running on HOST is defined as follows:*

```
# loads at 'dest' a number of 'size' p-scalar lines sent by HOST
def loadMatrix(dest, size)
    # send to ACC. the pointer to the load program
    # send to ACC. the location of the first line
    # send to ACC. number of lines
```

```
# sends from 'dest' a number of 'size' p-scalar lines to HOST
def storeMatrix(source, size)
    # send to ACC. the pointer to the store program
    # send to ACC. the location of the first line
    # send to ACC. number of lines

# multiply in 'dest' the matrices located at 'left' and 'right'
def matrixMult(dest, left, right)
    # send to ACC. the pointer to the multiply program
    # send to ACC. the location of the first line of destination
    # send to ACC. the location of the first line of left operand
    # send to ACC. the location of the first line of right operand

# add in 'dest' the matrices located at 'left' and 'right'
def matrixAdd(dest, left, right)
    # send to ACC. the pointer to the add program
    # send to ACC. the location of the first line of destination
    # send to ACC. the location of the first line of left operand
    # send to ACC. the location of the first line of right operand

# add to 'dest' the matrices located at 'left' and 'right'
def matrixMACC(dest, left, right)
    # send to ACC. the pointer to the MACC program
    # send to ACC. the location of the first line of destination
    # send to ACC. the location of the first line of left operand
    # send to ACC. the location of the first line of right operand
```

◇

On the zero level of the hierarchy, the programs associated with the 4 previously defined functions are run. These programs are written in assembly language to maximize performance. On any of the higher levels, a high-level language can be used that accesses the lower level as a library of functions implemented in the hardware.

Example 8.3 *The program which multiplies two* 128×128 *matrices stored in* **Memory** *with the result sent back in the same memory is:*

```
loadMatrix(8, 128)
loadMatrix(136, 128)
matrixMult(264, 8, 136)
storeMatrix(264, 128)
```

Because the transfer between **Memory** *and ACCELERATOR is done through some FIFO type memories, the program does not require additional synchronization mechanisms. The program run by HOST will "take care" to send the two matrices to the ACCELERATOR and to receive from it, when the operation has been completed, the result that it will be stored in the* **Memory**. *For the same reason, transfer commands can be inserted into the program written in Python before or after the previous sequence of commands. It is preferable that the matrix loading commands be earlier and the result unloading*

command is good to follow the sequence of commands sent to the accelerator. But, as well, the transfer controls can be grouped all three before or after the accelerator function sequence.

◇

Example 8.4 *Another form to use functions is to define the action of a sequence of functions using lambda expressions, as follows:*

```
(lambda dest, left, right, size:
    loadMatrix(left, size)
    loadMatrix(right, size)
    matrixMult(dest, left, right)
    storeMatrix(dest, size)       )(264, 8, 136)
```

◇

The use of lambda form

Part III

RECONFIGURABLE SYSTEMS

Chapter 9

Designing Pseudo-Reconfigurable Systems

Unlike pure reconfigurability, pseudo-reconfigurability involves the instantiation of a ***configurable & programmable*** accelerator in FPGA **only** at the beginning of a program. It is configurable to be adapted to the data structures and to the specifics of the functions to be accelerated. It is also programmable to run functions used to speed up the running of the program which calls the accelerator.

Why pseudo-reconfigurability? Because:

- the current compilers from the high level languages to HDL are far from being efficient
- the overhead introduced by the loading mechanism of a new circuit in FPGA is sometimes to big

Pros for pseudo-reconfigurability :

- the overhead of loading FPGA during the run of the program is avoided
- the structure of the accelerator is used for a family of functions which sometimes can be thought as a kernel of a library of functions

Cons for pseudo-reconfigurability :

- a programmable structure cannot achieve the performance of a circuit
- it is not always possible to optimize a programmable structure for the set of functions required to accelerate a program

The programmable structure that best approximates a circuit is a parallel computing structure. We decided to use, in adaptable configurations, the MapScanReduce structure described in the last three sections of Chapter 6.

9.1 The Pseudo-Reconfigurable Computing System

The structure of the Pseudo-Reconfigurable Computing system (see Figure 9.1) consists of:

- HOST SYSTEM: a general purpose computing system with Harvard abstract model (architecture)

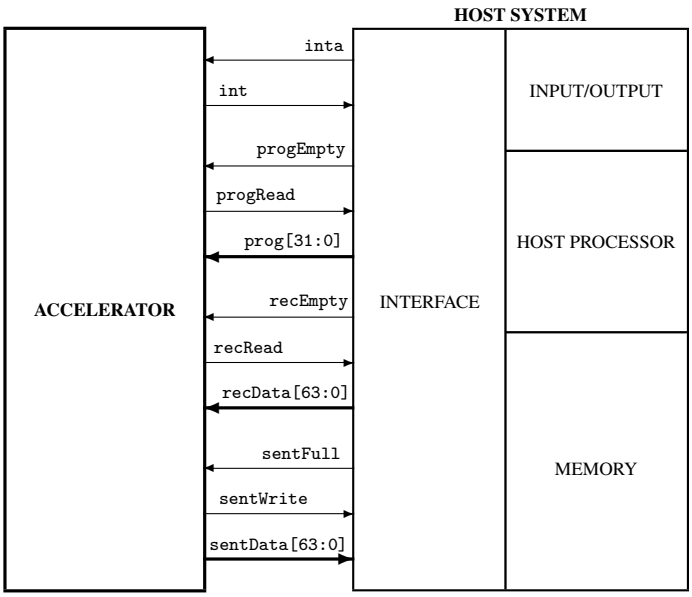

Figure 9.1: The hybrid system: HOST SYSTEM & ACCELERATOR.

- ACCELERATOR: a MapScanReduce parallel engine

The HOST SYSTEM (HOST PROCESSOR + MEMORY + INPUT/OUTPUT + INTERFACE) is supposed to run a ***complex*** part of the program stored in its program memory, part of MEMORY, while the ***intense*** part of the program runs in ACCELERATOR. The intense part of the program and the associated data, stored in the data section of MEMORY, are transferred between HOST SYSTEM and ACCELERATOR in the following steps:

1. in the program to run are identified the functions to be accelerated
2. the MapScanReduce model is configured according to the data structure and the functions to be accelerated
3. using the parameterized ISA the programs associated to the functions to be accelerated are written
4. the configured RTL design is loaded in FPGA
5. the programs are loaded using the programming port (`progEmpty, progRead, prog[31:0]`)
6. run the program on HOST PROCESSOR and the function to be accelerated are called on the programming port while the associated data is transferred through the data ports

9.2 Kernel Library Concept

The generic ConnexArray^{TM} can be used as the accelerator part of the proposed hybrid system. Because the design is parameterizable and programmable it can be used as a meaningful version for a pseudo-reconfigurable computing system.

The functional definition as an accelerator is given by a set of functions implemented as programs written in assembly language. The data structures for which these programs are defined have a size given by the limits that the hardware structure imposes. Because real applications always involve a more extensive data structure, this set of functions will be considered as the kernel library on the basis of which a library can be developed through host programs written in a high-level language.

9.2.1 Linear Algebra Kernel Library

The assembler must generate two codes:

- the code associated to the kernel function
- the code necessary to call the functions from the kernel

Example 9.1 *A small and simple kernel library for linear algebra with the following functions, defined for an accelerator with N cells:*

`GET(X,Y)`: *loads in the accelerator, starting with line X a matrix of Y N-scalar lines received on the* `recData[63:0]` *path*

`SEND(X,Y)`: *send via* `sendData[63: 0]` *an array of Y N-scalar lines stored in the accelerator starting with line X*

`SQMULT(X,Y,Z)`: *multiply two $N \times N$ matrices stored starting with the locations Y and Z, with the result in a matrix stored starting with the location X*

`SQMACC(X,Y,Z)`: *multiply two $N \times N$ matrices stored starting with the locations Y and Z, with the result added with the matrix stored starting with the location X*

`START`: *start the cycle counter*

`STOP`: *stop the cycle counter*

`INTRQ`: *interrupt request*

`SQADD(X,Y,Z)`: *add two $N \times N$ matrices stored starting with the locations Y and Z, with the result in a matrix stored starting with the location X*

The two sections of code generated for using this library are described in tab `00_theProgram.v`. In the first section appears the program that is loaded at system initialization and in the second section is sent to the accelerator the program that uses the kernel function library. The instruction `cPLOAD` received on the `prog[31:0]` starts the program loading process. The instruction `cPRUN(0)` stops the program load and start its execution from the address 0.

The file `00_the Kernel.v` starts with the instruction `cHALT` on the first line, because once loaded, the kernel library waits to be called. The second section contains a sequence of functions sent by the host processor correlated with the data transfer managed on the two data paths: `recData[63:0]`, `sentData[63:0]`. Each function exits with a jump to the label `LB(32)`.

(**Exercise**: add a new function to be used by the host processor for accessing the value of the cycle counter.)

The following code is generated for a 16-cell accelerator.

```
/*************************************************************************
File name: 00_theProgram.v
Description: the code for the accelerator sent on the prog[31:0] path
*************************************************************************/
// First section: THE PROGRAM USED TO INITIALIZE THE ACCELERATOR
            cPLOAD;          ACTIVATE;    // activate all cells
            cNOP;            IOLOAD;      // reset the input/ouput register
            cNOP;            REDNOP;      // inactivate the reduction net
            `include "00_theKernel.v"
    LB(32); cHALT;           NOP;         // halts the accelerator
            cPRUN(0);        NOP;         // stop program initialization
                                          // and run it from 0
// Second section: EXAMPLE OF PROGRAM SENT BY THE HOST PROCESSOR
            START;                        // start the cycle counter
            GETM(16,16);                  // get the first matrix
            GETM(32,16);                  // get the second matrix
            SQMULT(48,16,32);             // multiply
            SQMACC(48,16,32);             // multiply and accumulate
            SQADD(64,48,16);              // add
            SENDM(64,16);                 // send the result
            INTRQ;                        // send the interrupt
            STOP;                         // stop the cycle counter
```

The program in the second section

Each function is accessed in two steps. The 8 functions are labeled from LB (1) to LB (8), and in the program memory locations starting from address 4 we write jumps to the previous labels. The host launches each function with the instructions cPRUN (4), ..., cPRUN (11).

```
/*************************************************************************
File name: 00_theKernel.v
Description: LINEAR ALGEBRA KERNEL LIBRARY FUNCTIONS
*************************************************************************/
            cHALT;           NOP;
            cJMP(1);         NOP; // GET(addr, lines)          cPRUN(4);
            cJMP(2);         NOP; // SEND(addr, lines)         cPRUN(5);
            cJMP(3);         NOP; // SQMULT(dest, left, right) cPRUN(6);
            cJMP(4);         NOP; // SQMACC(dest, left, right) cPRUN(7);
            cJMP(5);         NOP; // START                     cPRUN(8);
            cJMP(6);         NOP; // STOP                      cPRUN(9);
            cJMP(7);         NOP; // INTRQ                     cPRUN(10);
            cJMP(8);         NOP; // SQADD(dest, left, right)  cPRUN(11);
/********** GET MATRIX ***************************************************/
    LB(1);  cPARAM;          NOP;
            cPARAM;          NOP;
            cVSUB(1);        CLOAD;
            cSTORE(0);       VSUB(1);
            cNOP;            ADDRLD;
    LB(15); cVLOAD(8);       NOP;
            cGETV;           NOP;
            cLOAD(0);        IOLOAD;
```

```
                cBRZDEC(32);      RISTORE(1);
                cSTORE(0);        NOP;
                cJMP(15);         NOP;
/********** SEND MATRIX **********************************************/
    LB(2);      cPARAM;           NOP;
                cPARAM;           NOP;
                cVADD(1);         CLOAD;
                cSTORE(0);        VSUB(1);
                cNOP;             ADDRLD;
    LB(16);     cNOP;             RILOAD(1);
                cLOAD(0);         IOSTORE;
                cBRZDEC(32);      NOP;
                cSTORE(0);        NOP;
                cNOP;             NOP;
                cNOP;             NOP;      // x=4
                cVLOAD(7);        NOP;
                cSENDV;           NOP;
                cJMP(16);         NOP;
/********** MULTIPLY SQUARE MATRICES ***********************************/
    LB(3);      cPARAM;           REDADD;
                cVSUB(1);         NOP;
                cSTORE(3);        NOP;      // dest => 3
                cPARAM;           NOP;
                cSTORE(0);        NOP;      // left => 0
                cPARAM;           CLOAD;
                cSTORE(2);        ADDRLD;   // right => 2
                cVLOAD(15);       NOP;
                cSTORE(1);        NOP;
                cLOAD(2);         NOP;
                cNOP;             NOP;
                cVADD(1);         CALOAD;
                cSTORE(2);        STORE(0);
                cVLOAD(16);       RLOAD(0);
    LB(17);     cREDINS;          MULT(0);
                cBRNZDEC(17);     RILOAD(1);
                cLOAD(3);         NOP;
                cVADD(1);         NOP;
                cSTORE(3);        SRLOAD;
                cLOAD(2);         CSTORE;   // CADD;
                cVADD(1);         NOP;
                cSTORE(2);        CALOAD;
                cLOAD(0);         STORE(0);
                cNOP;             NOP;
                cLOAD(1);         CLOAD;
                cBRZDEC(32) ;     ADDRLD;
                cSTORE(1);        NOP;
                cVLOAD(16);       RLOAD(0);
                cJMP(17);         NOP;
/**** MULTIPLY AND ACCUMULATE MATRICES ***********************************/
    LB(4);      cPARAM;           REDADD;
                cVSUB(1);         NOP;
                cSTORE(3);        NOP;      // dest => 3
                cPARAM;           NOP;
```

```
            cSTORE(0);      NOP;      // left => 0
            cPARAM;         CLOAD;
            cSTORE(2);      ADDRLD;   // right => 2
            cVLOAD(15);     NOP;
            cSTORE(1);      NOP;
            cLOAD(2);       NOP;
            cNOP;           NOP;
            cVADD(1);       CALOAD;
            cSTORE(2);      STORE(0);
            cVLOAD(16);     RLOAD(0);
    LB(18); cREDINS;        MULT(0);
            cBRNZDEC(18);   RILOAD(1);
            cLOAD(3);       NOP;
            cVADD(1);       NOP;
            cNOP;           SRLOAD;
            cSTORE(3);      CAADD;
            cLOAD(2);       CSTORE;
            cVADD(1);       NOP;
            cSTORE(2);      CALOAD;
            cLOAD(0);       STORE(0);
            cNOP;           NOP;
            cLOAD(1);       CLOAD;
            cBRZDEC(32) ;   ADDRLD;
            cSTORE(1);      NOP;
            cVLOAD(16);     RLOAD(0);
            cJMP(18);       NOP;
/********** START COUNTER ***********************************************/
    LB(5);  cSTART;         NOP;
            cJMP(32);       NOP;
/********** STOP COUNTER ************************************************/
    LB(6);  cSTOP;          NOP;
            cJMP(32);       NOP;
/********** INTERRUPT REQEST ********************************************/
    LB(7);  cSETINT;        NOP;
            cJMP(32);       NOP;
/********** ADD SQUARE MATRICES *****************************************/
    LB(8);  cPARAM;         NOP;
            cSTORE(3);      NOP;      // dest
            cPARAM;         NOP;
            cSTORE(4);      NOP;      // left
            cPARAM;         NOP;
            cSTORE(5);      NOP;      // right
            cSUB(4);        NOP;
            cSTORE(0);      NOP;      // right - left
            cLOAD(3);       NOP;
            cSUB(5);        NOP;
            cSTORE(1);      NOP;      // dest - right
            cLOAD(4);       NOP;
            cSUB(3);        NOP;
            cVADD(1);       NOP;
            cSTORE(2);      NOP;      // left - dest + 1
            cVLOAD(16);     NOP;
            cSTORE(6);      NOP;
```

```
    LB(19); cLOAD(4);       NOP;
            cADD(0);        NOP;
            cADD(1);        CALOAD;
            cADD(2);        CAADD;
            cSTORE(4);      CSTORE;
            cLOAD(6);       NOP;
            cVSUB(1);       NOP;
            cSTORE(6);      NOP;
            cBRNZ(19);      NOP;
            cNOP;           NOP;
            cNOP;           NOP;
            cNOP;           NOP;
            cNOP;           NOP;
            cNOP;           NOP;
            cNOP;           NOP;
            cJMP(32);       NOP;
```

The file `cgTRANSFER.v` included in `02_codeGenerator` (see Appendix C) must be completed with the following tasks:

```
/*************************************************************************
File name: LINEAR ALGEBRA KERNEL LIBRARY FUNCTIONS
Description:
*************************************************************************/
task GETM; // get matrix: get, starting at 'firstVect', 'lines' lines
    input[31:0] firstVect   ;
    input[31:0] lines       ;
    begin
    {aOpCode, aOperand, aScalar, cOpCode, cOperand, cScalar} =
    {16'b0, prun, ctl, 8'b0000_0100}                                    ;
    endLine                                                             ;
    {aOpCode, aOperand, aScalar, cOpCode, cOperand, cScalar} = firstVect;
    endLine                 ;
    {aOpCode, aOperand, aScalar, cOpCode, cOperand, cScalar} = lines    ;
    endLine                 ;
    end
endtask

task SENDM; // send matrix: send, starting at 'firstVect', 'lines' lines
    input[31:0] firstVect   ;
    input[31:0] lines       ;
    begin
    {aOpCode, aOperand, aScalar, cOpCode, cOperand, cScalar} =
    {16'b0, prun, ctl, 8'b0000_0101}                                    ;
    endLine                                                             ;
    {aOpCode, aOperand, aScalar, cOpCode, cOperand, cScalar} = firstVect;
    endLine                 ;
    {aOpCode, aOperand, aScalar, cOpCode, cOperand, cScalar} = lines    ;
    endLine                 ;
    end
```

```
endtask

task SQADD; // square matrix add
    input[31:0] dest    ;
    input[31:0] first   ;
    input[31:0] second  ;
    begin
    {aOpCode, aOperand, aScalar, cOpCode, cOperand, cScalar} =
    {16'b0, prun, ctl, 8'b0000_1011}                                    ;
    endLine                                                             ;
    {aOpCode, aOperand, aScalar, cOpCode, cOperand, cScalar} = dest;
    endLine                 ;
    {aOpCode, aOperand, aScalar, cOpCode, cOperand, cScalar} = first    ;
    endLine                 ;
    {aOpCode, aOperand, aScalar, cOpCode, cOperand, cScalar} = second   ;
    endLine                 ;
    end
endtask

task SQMULT; // square matrix multuply
    input[31:0] dest    ;
    input[31:0] first   ;
    input[31:0] second  ;
    begin
    {aOpCode, aOperand, aScalar, cOpCode, cOperand, cScalar} =
    {16'b0, prun, ctl, 8'b0000_0110}                                    ;
    endLine                                                             ;
    {aOpCode, aOperand, aScalar, cOpCode, cOperand, cScalar} = dest;
    endLine                 ;
    {aOpCode, aOperand, aScalar, cOpCode, cOperand, cScalar} = first    ;
    endLine                 ;
    {aOpCode, aOperand, aScalar, cOpCode, cOperand, cScalar} = second   ;
    endLine                 ;
    end
endtask

task SQMACC; // square matrix multuply & accumulate
    input[31:0] dest    ;
    input[31:0] first   ;
    input[31:0] second  ;
    begin
    {aOpCode, aOperand, aScalar, cOpCode, cOperand, cScalar} =
    {16'b0, prun, ctl, 8'b0000_0111}                                    ;
    endLine                                                             ;
    {aOpCode, aOperand, aScalar, cOpCode, cOperand, cScalar} = dest;
    endLine                 ;
    {aOpCode, aOperand, aScalar, cOpCode, cOperand, cScalar} = first    ;
    endLine                 ;
    {aOpCode, aOperand, aScalar, cOpCode, cOperand, cScalar} = second   ;
    endLine                 ;
    end
endtask
```

```
task START; // start cycle counter
    begin
    {aOpCode, aOperand, aScalar, cOpCode, cOperand, cScalar} =
    {16'b0, prun, ctl, 8'b0000_1000}                                    ;
    endLine                                                             ;
    end
endtask

task STOP; // stop cycle counter
    begin
    {aOpCode, aOperand, aScalar, cOpCode, cOperand, cScalar} =
    {16'b0, prun, ctl, 8'b0000_1001}                                    ;
    endLine                                                             ;
    end
endtask

task INTRQ; // interrupt request
    begin
    {aOpCode, aOperand, aScalar, cOpCode, cOperand, cScalar} =
    {16'b0, prun, ctl, 8'b0000_1010}                                    ;
    endLine                                                             ;
    end
endtask
```

Each task generate a first line associated to:

```
cPRUN(starting_address); NOP;
```

followed, if needed by lines containing 32-bit parameters.

Thus, to call the matrix multiplication function, `SQMULT(0, 32, 64)`, on the `prog[31:0]` path are inserted three 32-bit words as follows:

```
000_00000_00000000_111_11110_00000110
00000000_00000000_00000000_00000000
00000000_00000000_00000000_00100000
00000000_00000000_00000000_01000000
```

The ACCELERATOR‘s controller will “wake up” from `halt` state, when at the output of programm FIFO the instruction `prun` is detected, and will execute the program starting with the instruction stored at the location 6 in the controller‘s program memory. At the location 6 is a jump to the code which perform the matrix multiplication. The program ends in the halt state testing if a new call waits at the output of program FIFO. Meantime, from the program FIFO the three parameters used by the function are downloaded.

◇

APPENDIXES

Appendix A

Composition: the only independent rule in Kleene's model

Kleene's model looks like a good candidate for a mathematical model for parallel computing as the Turing's model was for the mono-core computation. In this respect, the composition seems to be a natural embodiment of a many-core abstract model for the parallel computing engine. The following conjecture has a big chance to become a theorem.

Conjecture A.1 *The composition rule, implemented as a two level structure (see Figure A.1):*

- *the* **map** *level: a linear array of circuits, one for each $h_i(X)$ function*
- *the* **reduce** *level: a log-depth tree-like network of circuits for $g(h_1(X),\ldots,h_p(X))$*

where the functions $h_i(X)$ and the function $g(h_1(X),\ldots,h_p(X))$ are initial functions or hierarchic compositions of initial functions, computes any functions $f:\mathbb{N}^n\to\mathbb{N}$.

◇

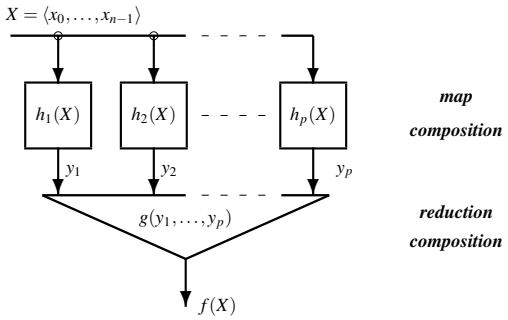

Figure A.1: **The circuit version of composition**. It is a two-layer construct: the parallel expanded *map* layer serially connected with the *reduction* layer.

Two kinds of parallelism are emphasized by the composition rule:

- a n-degree of synchronic parallelism between the computation performed in the circuits of the map level
- a 2-degree of diachronic (pipeline) parallelism between the computation on the map level and the computation of the reduction level

If the reduce circuit is detailed so as its function is performed as a composition, as follows:

$$g(y_1,\ldots,y_p) = g(g(y_1,y_2),g(y_3,y_4),\ldots)$$

where g is an associative and commutative function, then, occurs a new map level with a $n/2$-degree of synchronic parallelism followed by a reduce level with $p/2$ inputs. The process continues until a the binary form of the function g is reached. Therefore, the overall degree of parallelism is theoretically: $\delta = (2p-1)/(1+log_2 p)$.

In the next sections will be proved that, for the other two rules, specific compositions can be used, so as we are in the position to conclude that the computation model proposed by Kleene leads to implementations involving only compositions for which the previous theorem applies.

A.1 Preliminary Definitions

Definition A.1 *The reduction-less composition or* map composition*, MC, is the particular composition* $f:\mathbb{N}^n \to \mathbb{N}^p$ *where:*

$$f(X) = f(x_0,\ldots x_{n-1}) = \langle h_1(X),\ldots h_p(X)\rangle = \langle y_1,\ldots y_p\rangle$$

$h_i:\mathbb{N}^n \to \mathbb{N}$*, and* $g(y_1,\ldots,y_p) = \langle y_1,\ldots,y_p\rangle$ *is the identity function, for* $i = 1,\ldots,p$*.*

◇

Definition A.2 *The map-less composition or* reduction composition*, RC, is the particular composition* $f:\mathbb{N}^n \to \mathbb{N}$ *where:*

$$f(X) = f(x_0,\ldots x_{n-1}) = g(x_1,\ldots x_p)$$

with $y_i = h_i(X) = SEL(i-1,X) = x_{i-1}$*, for* $i = 1,\ldots,p$ *and* $n = p$*.*

◇

According to the previous two definitions, the composition rule can be considered as having a ***map-reduce*** structure (Figure A.1), where a MC is serially connected with a RC. The two functional level can have associated the physical implementation with the h_i functions and the g function embodied in various forms, starting from combinational circuits and reaching the complexity and competence of a processor, even a computer.

Definition A.3 *The function* $C_i:\mathbb{N}^{i\times n+m} \to \mathbb{N}^{(i+1)\times n}$ *is a MC defined as:*

$$C_i(\mathbf{Y},Z_i) = \langle Y, P_i(X_i,Z_i)\rangle = \langle X_1,\ldots,X_i,P_i(X_i,Z_i)\rangle$$

where:

- $\mathbf{Y} = \langle X_1,\ldots,X_i\rangle$*, the first argument, is a sequence of sequences with* $X_j \in \mathbb{N}^n$*, for* $j = 1,\ldots,i$
- Z_i*, the second argument, is a sequence of m scalars*
- $h_j(Y) = SEL(j,Y) = X_j$ *for* $j = 1,2,\ldots,i$
- $h_{i+1}(Y) = P_i(SEL(i,Y),Z_i) = P_i(X_i,Z_i)$.

◇

The C function adds, at the end of the sequence X, a new element computed using as arguments the last element of X and and the second argument of the function, Z_i.

Definition A.4 *Multiple application of C_i (see Figure A.2a), starting from $C_1(\langle X\rangle, Z_1) = \langle X, P_1(X,Z_1)\rangle$, where the arguments are $\langle X\rangle$, a one component sequence of n scalars, and Z_1 the first component of the sequence of sequences $\mathbf{Z} = \langle Z_1, Z_2, \ldots\rangle$, defines the* multi-output pipeline*, MOP:*

$$MOP(X,\mathbf{Z}) = \langle X, P_1(X,Z_1), P_2(P_1,Z_2), \ldots, P_k(P_{k-1}(\ldots,(P_1(X,Z_1)\ldots)), Z_k), \ldots\rangle$$

The resulting structure, with one sequence, X and $\mathbf{Z}$, as arguments and as many as necessary computed values, $P_k(P_{k-1}(\ldots,(P_1(X,Z_1)\ldots)), Z_k)$, is represented in Figure A.2b.

◇

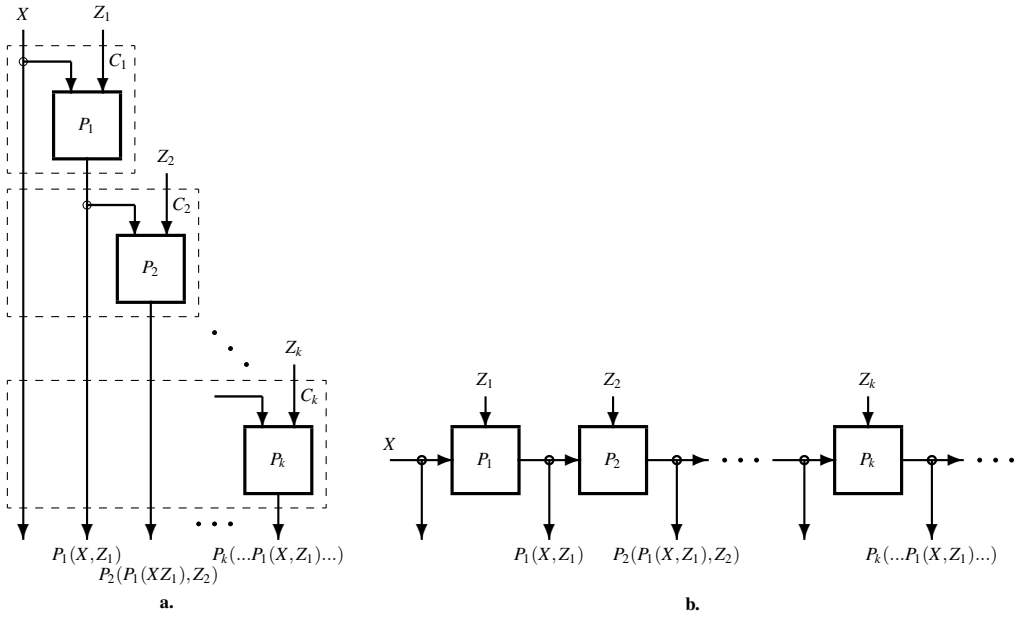

Figure A.2: **The multi-output pipeline structure (MOP)**. **a**. The explicit application of C_i. **b**. The resulting MOP circuit structure.

The function $MOP(X,\mathbf{Z})$ is a total function if the functions P_i are total functions, since it is computed using only the repeated application of the composition C_i. For the theoretical model, k is not limited to a specific value. By defining this sequence of MCs, a left to right serial connection between cells is added to the general form of the MC structure. Similarly, a right to left connection can be defined.

Definition A.5 *The RC function $redOR : \mathbb{N}^n \to \mathbb{N}$ is*

$$redOR(X) = x_0|x_1|\ldots|x_i|\ldots$$

where: $X = \langle x_0, x_1, \ldots, x_i, \ldots\rangle$, and $|$ denote the bitwise OR logical function.

◇

Definition A.6 *The MC function scanFIRST* : $\{0,1\}^n \rightarrow \{0,1\}^n$ *is:*

$$scanFIRST(B) = \langle B_0, |B_1\& \sim |B_0, \ldots |B_i\& \sim |B_{i-1} \ldots \rangle$$

where:

- $B = \langle b_0, b_1, \ldots, b_i, \ldots \rangle$ *is a Boolean sequence with an unspecified size*
- $B_i = \langle b_0, b_1, \ldots, b_i \rangle$ *is a finite Boolean sequence (the first* $i+1$ *elements of B)*
- $|$ *applied to sequence of Booleans it returns the OR functions applied to the components of the sequence*
- & *is the logic operator AND*
- $\sim$ *is the logic negation*

◇

The *scanFIRST(B)* function identifies the first occurrence of 1 in a the Boolean sequence B. This function is based on the prefixes of the logic function OR.

A.2 Primitive Recursion Computed as a Sequence of Compositions

Theorem A.1 *The primitive recursive rule is reducible to repeated applications of specific compositions.*

◇

Proof A.1 The primitive recursion rule could be applied using its iteratively expanded form:

$$f(x,y) = g(x, f(x,y-1)) = \ldots = \underbrace{g(x, g(x, g(x, \ldots g}_{(y-1)\ times}(x, f(x,1)) \ldots))) =$$

$$= \underbrace{g(x, g(x, g(x, \ldots g}_{y\ times}(x, f(x,0)) \ldots))) = \underbrace{g(x, g(x, g(x, \ldots g}_{y\ times}(x, h(x)) \ldots)))$$

Let be, in Figure A.3, the specific instantiation of the *MOP* function (see Definition A.4). It computes iteratively, starting in the first stage with the function $f(x,0) = h(x)$, the values $f(x,i)$ for $i = 0,1,....$ In each stage the predicate $i = y$ is computed. The functions P_i, for $i = 1,2,\ldots$, takes form P_{i-1} the value of $f(X,i-1)$ and computes $f(X,i)$. The *redOR* function takes from the *MOP* function its arguments as $y = i\ ?f(X,i) : 0$ for $i = 0,1,2,\ldots$. Because for only one i the predicate $y = i$ takes the value 1, the function *redOR* returns the value of $f(X,y)$.

Thus, for primitive recursion we need to compose two compositions, *MOP* and *redOR*.

◇

Figure A.3 presents the circuit version of the function obtained by composing a specific *MOP* function with the *redOR* function. The two stage computation just described, as a structure indefinitely extensible to the right, is a theoretical model because the index i takes values no matter how large, similar with the "infinite" tape of Turing Machine. But, it is very important that the algorithmic complexity of the description is in $O(1)$, because the functions P_i, *MOP* and *redOR* have constant size descriptions.

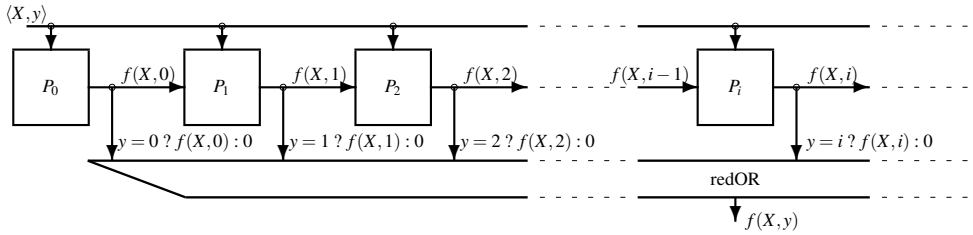

Figure A.3: **The *MOP* & *redOR* circuit version for the partial recursive rule**.

A.3 Minimization Computed as a Sequence of Compositions

Theorem A.2 *The minimization (least-search) rule is reducible to repeated applications of specific compositions.*

◇

Proof A.2 The minimization (least-search) rule computes the value of $f(x)$ as the smallest y, **if any**, for which $g(x,y) = 0$.

Let be, a specific structure from Figure A.4. Each cell on the map level computes the pair $\langle predicate, value \rangle$:

$$G_i(x, \phi_i) = \langle (g(x,i) = 0), (\phi_i \ ? \ (i+1) : 0) \rangle$$

The predicate is sent to the scan circuit, while the value to the reduction circuit. The *scanFIRST* loop points to the first cell, if any, which provided the predicate $(g(x,i) = 0) = 1$. The reduction level computes *redOR* selecting to the output the value $i+1$ for $\phi_i = 1$, if any. If for $x = a$ the output of the reduction is 0, then the function is not defined for $x = a$, else the output takes the value $f(x)+1$, because the value 0 is reserved to indicate that the function is not defined for the value x applied on the input.

◇

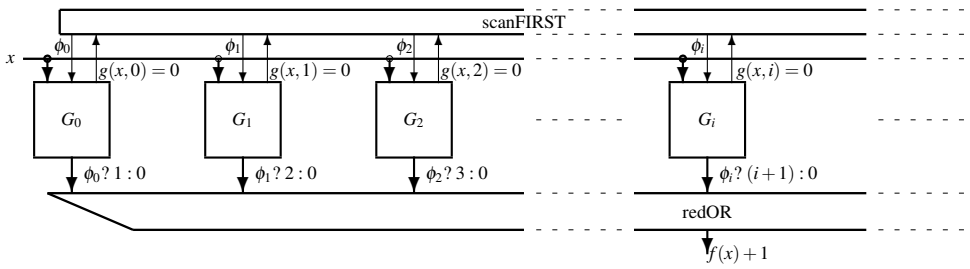

Figure A.4: **The circuit structure for the minimization rule**.

The computation just described is only a theoretical model, because the index i has an indefinitely large value. But, the size of the algorithmic description remains $O(1)$.

A.4 Partial Recursion Means Composition Only

Kleene's approach defines, besides the composition rule, the other two rules, ordinary (primitive) recursion and minimization (least-search), only for providing the means for classifying the recursive functions (to emphasize in the class of recursive functions the partial recursive functions and primitive recursive functions). Therefor, to define the computation the next corollary makes the necessary and sufficient delimitation.

Corollary A.1 *Any computation defined in Definition 4.6 can be done, according to Theorem A.1 and Theorem A.2, using the initial functions and the repeated application of the composition rule.*
◇

The primitive recursion is differentiated from the partial recursion by the main fact that it does not require the scan loop. It is only a straight forward sequence of compositions. For the partial recursive rule, the additional loop interferes with the sequence of compositions in order to validate intermediate results. Thus, an actual abstract model based on the Kleene's computational model must provide also a sequencing mechanism. This means at least to provide programmability at the cells level and an external control.

Corollary A.1 can be used to define an abstract model for parallel computation. The resulting structure is able to support the hybrid solution for advanced parallel computation. A Church inspired lambda architecture [Church '36] can be defined for the complex control, while a Kleene inspired architecture could be used for solving problems raised by the intense computation. The operating systems problems can remain to be handled by Turing/Post [Turing '36] [Post '36] inspired engines.

Appendix B

How to instantiate DSP48E1

```
DSP48E1 #(
// Feature Control Attributes: Data Path Selection
      .A_INPUT   (),   // Selects A input source, "DIRECT" (A port) or "CASCADE" (ACIN port)
      .B_INPUT   (),   // Selects B input source, "DIRECT" (B port) or "CASCADE" (BCIN port)
      .USE_DPORT(),   // Select D port usage (TRUE or FALSE)
      .USE_MULT (),   // Select multiplier usage ("MULTIPLY", "DYNAMIC", or "NONE")
// Pattern Detector Attributes: Pattern Detection Configuration
      .AUTORESET_PATDET      (),   // "NO_RESET", "RESET_MATCH", "RESET_NOT_MATCH"
      .MASK                  (),   // 48-bit mask value for pattern detect (1=ignore)
      .PATTERN               (),   // 48-bit pattern match for pattern detect
      .SEL_MASK              (),   // "C", "MASK", "ROUNDING_MODE1", "ROUNDING_MODE2"
      .SEL_PATTERN           (),   // Select pattern value ("PATTERN" or "C")
      .USE_PATTERN_DETECT    (),   // Enable pattern detect ("PATDET" or "NO_PATDET")
// Register Control Attributes: Pipeline Register Configuration
      .ACASCREG      (), // Number of pipeline stages between A/ACIN and ACOUT (0, 1 or 2)
      .ADREG         (), // Number of pipeline stages for pre-adder (0 or 1)
      .ALUMODEREG    (), // Number of pipeline stages for ALUMODE (0 or 1)
      .AREG          (), // Number of pipeline stages for A (0, 1 or 2)
      .BCASCREG      (), // Number of pipeline stages between B/BCIN and BCOUT (0, 1 or 2)
      .BREG          (), // Number of pipeline stages for B (0, 1 or 2)
      .CARRYINREG    (), // Number of pipeline stages for CARRYIN (0 or 1)
      .CARRYINSELREG (), // Number of pipeline stages for CARRYINSEL (0 or 1)
      .CREG          (), // Number of pipeline stages for C (0 or 1)
      .DREG          (), // Number of pipeline stages for D (0 or 1)
      .INMODEREG     (), // Number of pipeline stages for INMODE (0 or 1)
      .MREG          (), // Number of multiplier pipeline stages (0 or 1)
      .OPMODEREG     (), // Number of pipeline stages for OPMODE (0 or 1)
      .PREG          (), // Number of pipeline stages for P (0 or 1)
      .USE_SIMD      ()  // SIMD selection ("ONE48", "TWO24", "FOUR12")
   )
   DSP48E1_inst (
// Cascade: 30-bit (each) output: Cascade Ports
      .ACOUT         (),  // 30-bit output: A port cascade output
      .BCOUT         (),  // 18-bit output: B port cascade output
      .CARRYCASCOUT  (),  // 1-bit output: Cascade carry output
      .MULTSIGNOUT   (),  // 1-bit output: Multiplier sign cascade output
      .PCOUT         (),  // 48-bit output: Cascade output
// Control: 1-bit (each) output: Control Inputs/Status Bits
      .OVERFLOW          (),  // 1-bit output: Overflow in add/acc output
      .PATTERNBDETECT    (),  // 1-bit output: Pattern bar detect output
      .PATTERNDETECT     (),  // 1-bit output: Pattern detect output
      .UNDERFLOW         (),  // 1-bit output: Underflow in add/acc output
// Data: 4-bit (each) output: Data Ports
      .CARRYOUT (),  // 4-bit output: Carry output
      .P        (),  // 48-bit output: Primary data output
// Cascade: 30-bit (each) input: Cascade Ports
      .ACIN          (),  // 30-bit input: A cascade data input
      .BCIN          (),  // 18-bit input: B cascade input
      .CARRYCASCIN   (),  // 1-bit input: Cascade carry input
      .MULTSIGNIN    (),  // 1-bit input: Multiplier sign input
```

```
      .PCIN          (),  // 48-bit input: P cascade input
// Control: 4-bit (each) input: Control Inputs/Status Bits
      .ALUMODE       (),  // 4-bit input: ALU control input
      .CARRYINSEL    (),  // 3-bit input: Carry select input
      .CLK           (),  // 1-bit input: Clock input
      .INMODE        (),  // 5-bit input: INMODE control input 10111
      .OPMODE        (),  // 7-bit input: Operation mode input
// Data: 30-bit (each) input: Data Ports
      .A         (),  // 30-bit input: A data input
      .B         (),  // 18-bit input: B data input
      .C         (),  // 48-bit input: C data input
      .CARRYIN   (),  // 1-bit  input: Carry input signal
      .D         (),  // 25-bit input: D data input
// Reset/Clock Enable: 1-bit (each) input: Reset/Clock Enable Inputs
      .CEA1          (), // 1-bit input: Clock enable input for 1st stage AREG
      .CEA2          (), // 1-bit input: Clock enable input for 2nd stage AREG
      .CEAD          (), // 1-bit input: Clock enable input for ADREG
      .CEALUMODE     (), // 1-bit input: Clock enable input for ALUMODE
      .CEB1          (), // 1-bit input: Clock enable input for 1st stage BREG
      .CEB2          (), // 1-bit input: Clock enable input for 2nd stage BREG
      .CEC           (), // 1-bit input: Clock enable input for CREG
      .CECARRYIN     (), // 1-bit input: Clock enable input for CARRYINREG
      .CECTRL        (), // 1-bit input: Clock enable input for OPMODEREG and CARRYINSELREG
      .CED           (), // 1-bit input: Clock enable input for DREG
      .CEINMODE      (), // 1-bit input: Clock enable input for INMODEREG
      .CEM           (), // 1-bit input: Clock enable input for MREG
      .CEP           (), // 1-bit input: Clock enable input for PREG
      .RSTA          (), // 1-bit input: Reset input for AREG
      .RSTALLCARRYIN(), // 1-bit input: Reset input for CARRYINREG
      .RSTALUMODE    (), // 1-bit input: Reset input for ALUMODEREG
      .RSTB          (), // 1-bit input: Reset input for BREG
      .RSTC          (), // 1-bit input: Reset input for CREG
      .RSTCTRL       (), // 1-bit input: Reset input for OPMODEREG and CARRYINSELREG
      .RSTD          (), // 1-bit input: Reset input for DREG and ADREG
      .RSTINMODE     (), // 1-bit input: Reset input for INMODEREG
      .RSTM          (), // 1-bit input: Reset input for MREG
      .RSTP          ()  // 1-bit input: Reset input for PREG
   );
```

Appendix C

ConnexArray^{TM} Simulator

C.1 Top Module: `simulator.v`

```
/*********************************************************************************************
File name:      simulator.v
Description:    Simulator for the module ConnexArray.v
*********************************************************************************************/
module simulator #(`include "parameters.v");
    reg     reset, clock;
    integer j;

    initial begin               clock = 0       ;
                forever #1  clock = ~clock  ;
            end

    ConnexArray dut(reset   ,
                    clock   );

    initial begin   $readmemh("initialData.txt", dut.mem); end

    // ASSEMBLER
    `include "codeGenerator.v"      // accelerator's assembler

    //SIMULATION
    initial begin
            reset = 1   ;
            for (j=0; j<16; j=j+1)
            $display("programMemory[%0d] \t = %b ", j, dut.progMem[j]);
        #4  reset = 0   ;
        #190 begin
            // DISPLAY VECTORS OF THE ARRAY
            for (j=0; j<8; j=j+1)
            $display("vect[%0d] \t = %d \t %0d \t %0d \t %0d \t %0d \t %0d \t %0d \t %0d \t
                    %0d \t %0d \t %0d \t %0d \t %0d \t %0d \t %0d \t %0d \t %0d \t %0d",
                    j,  dut.vmem[0][j],  dut.vmem[1][j],
                        dut.vmem[2][j],  dut.vmem[3][j],
                        dut.vmem[4][j],  dut.vmem[5][j],
                        dut.vmem[6][j],  dut.vmem[7][j],
                        dut.vmem[8][j],  dut.vmem[9][j],
                        dut.vmem[10][j], dut.vmem[11][j],
                        dut.vmem[12][j], dut.vmem[13][j],
                        dut.vmem[14][j], dut.vmem[15][j]);
            // DISPLAY THE SCALAR MEMORY
            for (j=0; j<32; j=j+1)
                    $display("mem[%0d] \t = %0d ", j, dut.mem[j]);
            end
        #2  $finish ;
    end

    // MONITOR FOR GENERAL TEST
```

```
    initial begin
        $monitor("t=%0d   pc=%0d   ir=%b acc=%d   addr=%0d ACC=[%d, %0d, %0d, %0d, %0d,
                  %0d, %0d, %0d, %0d, %0d, %0d, %0d, %0d, %0d, %0d, %0d] ADDR=[%0d, %0d,
                  %0d, %0d, %0d, %0d, %0d, %0d, %0d, %0d, %0d, %0d, %0d, %0d, %0d, %0d]
        b=[%0d%0d%0d%0d%0d%0d%0d%0d%0d%0d%0d%0d%0d%0d%0d%0d]",
            $time,
            // CONTROLLER
            dut.pc,      // program counter
            dut.ir,      // instruction register
            dut.acc,     // accumulator register
            dut.addr,    // address register
            // MAP-REDUCE ARRAY
                                            // accumulator vector
            dut.accv[0],   dut.accv[1],   dut.accv[2],   dut.accv[3],
            dut.accv[4],   dut.accv[5],   dut.accv[6],   dut.accv[7],
            dut.accv[8],   dut.accv[9],   dut.accv[10],  dut.accv[11],
            dut.accv[12],  dut.accv[13],  dut.accv[14],  dut.accv[15],
                                            // address vector
            dut.addrv[0],  dut.addrv[1],  dut.addrv[2],  dut.addrv[3],
            dut.addrv[4],  dut.addrv[5],  dut.addrv[6],  dut.addrv[7],
            dut.addrv[8],  dut.addrv[9],  dut.addrv[10], dut.addrv[11],
            dut.addrv[12], dut.addrv[13], dut.addrv[14], dut.addrv[15],
                                            // Boolean vector
            dut.bool[0],   dut.bool[1],   dut.bool[2],   dut.bool[3],
            dut.bool[4],   dut.bool[5],   dut.bool[6],   dut.bool[7],
            dut.bool[8],   dut.bool[9],   dut.bool[10],  dut.bool[11],
            dut.bool[12],  dut.bool[13],  dut.bool[14],  dut.bool[15]
            );
    end
endmodule
```

C.2 Code generator

```
/*********************************************************************************************
File name:      codeGenerator.v
Description:    assembler for Connex Array
*********************************************************************************************/
    reg [4:0]   aOpCode             ;
    reg [2:0]   aOperand            ;
    reg [7:0]   aScalar             ;
    reg [4:0]   cOpCode             ;
    reg [2:0]   cOperand            ;
    reg [7:0]   cScalar             ;
    reg [p-1:0] deltaAddr           ;
    reg [p-1:0] addrCounter         ;
    reg [p-1:0] labelTab[0:(1<<p)-1];
    task endLine;
        begin
            dut.progMem[addrCounter] = {aOpCode ,
                                        aOperand,
                                        aScalar ,
                                        cOpCode ,
                                        cOperand,
                                        cScalar }   ;
            addrCounter = addrCounter + 1           ;
        end
    endtask
    // sets labelTab in the first pass loading 'counter' with 'labelIndex'
    task LB ;
        input [4:0] labelIndex;
        labelTab[labelIndex] = addrCounter;
    endtask
    // uses the content of labelTab in the second pass
    task cULB;
        input [4:0] labelIndex;
        begin   deltaAddr   = labelTab[labelIndex] - addrCounter;
                cScalar     = deltaAddr[7:0]                    ;
```

```
        end
    endtask
    `include "cgCONTROL.v"        // control instructions for controller
    `include "cgADD.v"            // addition
    `include "cgADDC.v"           // addition with carry
    `include "cgSUB.v"            // subtract
    `include "cgSUBC.v"           // subtract with carry
    `include "cgRVSUB.v"          // reverse subtract
    `include "cgRVSUBC.v"         // reverse subtract with carry
    `include "cgMULT.v"           // multiplication
    `include "cgSHIFT.v"          // shift
    `include "cgLOAD.v"           // load accumulator
    `include "cgSTORE.v"          // store accumulator
    `include "cgAND.v"            // bit-wise AND
    `include "cgOR.v"             // bit-wise OR
    `include "cgXOR.v"            // bit-wise XOR
    `include "cgARRAYcONTR.v"     // array control instructions
    `include "cgGLOBAL.v"         // global operations
    `include "cgTRANSFER.v"       // io transfer operations
    `include "cgSEARCH.v"         // search functions (ONLY FOR THE SEARCH VERSION)
    // RUNNING
    initial begin    addrCounter = 0;
                     `include "program.v" // first pass
                     addrCounter = 0;
                     `include "program.v" // second pass
            end
```

The line

```
    `include "cgSEARCH.v"         // search functions (ONLY FOR THE SEARCH VERSION)
```

is added only for the search version.

C.2.1 Assembly Functions

For each instruction, the following files contain tasks which generate the binary form of the instructions used to write programs in assembly language.

Add functions

```
/*****************************************************************************************
File name:       cgADD.v
*****************************************************************************************/
// in ARRAY
    task VADD; // value add:
               // acc[i] <= acc[i] + {(n-8){aScalar[7]}}, aScalar}
        input [7:0] value;
        begin   aOpCode     = add   ;
                aOperand    = val   ;
                aScalar     = value ;
                endLine             ;
        end
    endtask

    task ADD; // absolute add
              // acc[i] <= acc[i] + vectMem[i][aScalar[v-1:0]]
        input [7:0] value;
        begin   aOpCode     = add   ;
                aOperand    = mab   ;
                aScalar     = value ;
                endLine             ;
        end
    endtask

    task RADD; // relative add:
```

```
            // acc[i]<=acc[i]+vectMem[i][addrVect[i]+aScalar[v-1:0]]
        input [7:0] value;
        begin   aOpCode     = add   ;
                aOperand    = mrl   ;
                aScalar     = value ;
                endLine             ;
        end
    endtask
    task RIADD; // relative add and increment:
                // acc[i]<=acc[i]+vectMem[i][addrVect[i]+aScalar[v-1:0]]
                // addrVect[i] <= addrVect[i] + aScalar[v-1:0]
        input [7:0] value;
        begin   aOpCode     = add   ;
                aOperand    = mri   ;
                aScalar     = value ;
                endLine             ;
        end
    endtask
    task CADD; // co-operand add:
               // acc[i] <= acc[i] + acc
        begin   aOpCode     = add   ;
                aOperand    = cop   ;
                aScalar     = 8'b0  ;
                endLine             ;
        end
    endtask

// in CONTROLLER
    task cVADD; // value add:
                // acc <= acc + {(n-8){cScalar[7]}}, cScalar}
        input [7:0] value;
        begin   cOpCode     = add   ;
                cOperand    = val   ;
                cScalar     = value ;
        end
    endtask

    task cADD; // immediate add:
               // acc <= acc + mem[cScalar[s-1:0]]
        input [7:0] value;
        begin   cOpCode     = add   ;
                cOperand    = mab   ;
                cScalar     = value ;
        end
    endtask

    task cRADD; // relative add:
                // acc <= acc + mem[addr + cScalar[s-1:0]]
        input [7:0] value;
        begin   cOpCode     = add   ;
                cOperand    = mrl   ;
                cScalar     = value ;
        end
    endtask

    task cRIADD; // relative add:
                 // acc <= acc + mem[addr + cScalar[s-1:0]]
        input [7:0] value;
        begin   cOpCode     = add   ;
                cOperand    = mri   ;
                cScalar     = value ;
        end
    endtask

    task cCADD;// acc <= acc + mem[coOp]
               //  scalar[1:0] = 00: coOp = reduction add
               //  scalar[1:0] = 01: coOp = reduction min
               //  scalar[1:0] = 10: coOp = reduction max
               //  scalar[1:0] = 11: coOp = reduction flag
        input [7:0] value;
        begin   cOpCode     = add   ;
                cOperand    = cop   ;
```

```
                cScalar     = value ;
        end
    endtask
```

Add with carry functions

```
/*****************************************************************************************
File name:      cgADDC.v
*****************************************************************************************/
// in ARRAY
    task VADDC;
        input [7:0] value;
        begin   aOpCode     = addc  ;
                aOperand    = val   ;
                aScalar     = value ;
                endLine             ;
        end
    endtask

    task ADDC;
        input [7:0] value;
        begin   aOpCode     = addc  ;
                aOperand    = mab   ;
                aScalar     = value ;
                endLine             ;
        end
    endtask

    task RADDC;
        input [7:0] value;
        begin   aOpCode     = addc  ;
                aOperand    = mrl   ;
                aScalar     = value ;
                endLine             ;
        end
    endtask

    task RIADDC;
        input [7:0] value;
        begin   aOpCode     = addc  ;
                aOperand    = mri   ;
                aScalar     = value ;
                endLine             ;
        end
    endtask

    task CADDC;
        begin   aOpCode     = addc  ;
                aOperand    = cop   ;
                aScalar     = 8'b0  ;
                endLine             ;
        end
    endtask

// in CONTROLLER
    task cVADDC;
        input [7:0] value;
        begin   cOpCode     = addc  ;
                cOperand    = val   ;
                cScalar     = value ;
        end
    endtask

    task cADDC;
        input [7:0] value;
        begin   cOpCode     = addc  ;
                cOperand    = mab   ;
                cScalar     = value ;
```

```
        end
    endtask

    task cRADDC;
        input [7:0] value;
        begin   cOpCode     = addc  ;
                cOperand    = mrl   ;
                cScalar     = value ;
        end
    endtask

    task cRIADDC;
        input [7:0] value;
        begin   cOpCode     = addc  ;
                cOperand    = mri   ;
                cScalar     = value ;
        end
    endtask

    task cCADDC;
        input [7:0] value;
        begin   cOpCode     = addc  ;
                cOperand    = cop   ;
                cScalar     = value ;
        end
    endtask
```

Bitwise AND functions

```
/*****************************************************************************************
File name:      cgAND.v
*****************************************************************************************/
// in ARRAY
    task VAND;
        input [7:0] value;
        begin   aOpCode     = bwand ;
                aOperand    = val   ;
                aScalar     = value ;
                endLine             ;
        end
    endtask

    task AND;
        input [7:0] value;
        begin   aOpCode     = bwand ;
                aOperand    = mab   ;
                aScalar     = value ;
                endLine             ;
        end
    endtask
    task RAND;
        input [7:0] value;

        begin   aOpCode     = bwand ;
                aOperand    = mrl   ;
                aScalar     = value ;
                endLine             ;
        end
    endtask

    task RIAND;
        input [7:0] value;
        begin   aOpCode     = bwand ;
                aOperand    = mri   ;
                aScalar     = value ;
                endLine             ;
        end
    endtask
```

```
    task CAND;
        begin   aOpCode     = bwand ;
                aOperand    = cop   ;
                aScalar     = 8'b0  ;
                endLine             ;
        end
    endtask

// in CONTROLLER
    task cVAND;
        input [7:0] value;
        begin   cOpCode     = bwand ;
                cOperand    = val   ;
                cScalar     = value ;
        end
    endtask

    task cAND;
        input [7:0] value;
        begin   cOpCode     = bwand ;
                cOperand    = mab   ;
                cScalar     = value ;
        end
    endtask

    task cRAND;
        input [7:0] value;
        begin   cOpCode     = bwand ;
                cOperand    = mrl   ;
                cScalar     = value ;
        end
    endtask

    task cRIAND;
        input [7:0] value;
        begin   cOpCode     = bwand ;
                cOperand    = mri   ;
                cScalar     = value ;
        end
    endtask

    task cCAND;
        input [7:0] value;
        begin   cOpCode     = bwand ;
                cOperand    = cop   ;
                cScalar     = value ;
        end
    endtask
```

Array Control functions

```
/*****************************************************************************************
File name:      cgARRAYcONTROL.v
*****************************************************************************************/
    task WHEREZERO; // where acc[i] = 0
        begin   aOpCode     = where ;
                aOperand    = val   ;
                aScalar     = 8'b0  ;
                endLine             ;
        end
    endtask

    task WHERECARRY; // where carry
        begin   aOpCode     = where ;
                aOperand    = val   ;
                aScalar     = 8'b1  ;
                endLine             ;
```

```
    end
endtask

task WHERENZERO; // where acc[i] != 0
    begin   aOpCode     = where ;
            aOperand    = val   ;
            aScalar     = 8'b100;
            endLine             ;
    end
endtask

task WHERENCARRY; // where not carry
    begin   aOpCode     = where ;
            aOperand    = val   ;
            aScalar     = 8'b101;
            endLine             ;
    end
endtask

task ELSEWHERE; // else where
    begin   aOpCode     = elsew ;
            aOperand    = val   ;
            aScalar     = 8'b0  ;
            endLine             ;
    end
endtask

task ENDWHERE;
    begin   aOpCode     = endwhere  ;
            aOperand    = ctl       ;
            aScalar     = 8'b0      ;
            endLine                 ;
    end
endtask

task NOP;
    begin   aOpCode     = add   ;
            aOperand    = val   ;
            aScalar     = 8'b0  ;
            endLine             ;
    end
endtask
```

Controller's control functions

```
/*****************************************************************************************
File name:      cgCONTROL.v
*****************************************************************************************/
    task cJMP;
        input [5:0] label   ;
        begin   cOpCode     = jmp   ;
                cOperand    = ctl   ;
                cULB(label)         ;
        end
    endtask

    task cBRZ;
        input [5:0] label   ;
        begin   cOpCode     = brz   ;
                cOperand    = ctl   ;
                cULB(label)         ;
        end
    endtask

    task cBRNZ;
        input [5:0] label   ;
        begin   cOpCode     = brnz  ;
                cOperand    = ctl   ;
```

```
                cULB(label)         ;
        end
    endtask

    task cBRZDEC;
        input [5:0] label   ;
        begin   cOpCode     = brzdec;
                cOperand    = ctl   ;
                cULB(label)         ;
        end
    endtask

    task cBRNZDEC;
        input [5:0] label   ;
        begin   cOpCode     = brnzdec   ;
                cOperand    = ctl       ;
                cULB(label)             ;
        end
    endtask

    task cHALT;
        begin   cOpCode     = jmp   ;
                cOperand    = ctl   ;
                cScalar     = 8'b0  ;
        end
    endtask

    task cNOP;
        begin   cOpCode     = add   ;
                cOperand    = val   ;
                cScalar     = 8'b0  ;
        end
    endtask
```

Global functions

```
/*****************************************************************************************
File name:      cgGLOBAL.v
*****************************************************************************************/
// SHIFTS
    task GRSHIFT; // global right shift with one position
        begin   aOpCode     = gshift;
                aOperand    = val   ;
                aScalar     = 8'b0  ;
                endLine             ;
        end
    endtask

    task GLSHIFT; // global left shift with one position
        begin   aOpCode     = gshift;
                aOperand    = val   ;
                aScalar     = 8'b1  ;
                endLine             ;
        end
    endtask
```

Load functions

```
/*****************************************************************************************
File name:      cgLOAD.v
*****************************************************************************************/
    // in ARRAY
    task VLOAD;
```

```
    input [7:0] value
    begin   aOpCode     = load  ;
            aOperand    = val   ;
            aScalar     = value ;
            endLine             ;
    end
endtask

task LOAD;
    input [7:0] value;
    begin   aOpCode     = load  ;
            aOperand    = mab   ;
            aScalar     = value ;
            endLine             ;
    end
endtask

task RLOAD;
    input [7:0] value;
    begin   aOpCode     = load  ;
            aOperand    = mrl   ;
            aScalar     = value ;
            endLine             ;
    end
endtask

task RILOAD;
    input [7:0] value;
    begin   aOpCode     = load  ;
            aOperand    = mri   ;
            aScalar     = value ;
            endLine             ;
    end
endtask

task CLOAD;
    begin   aOpCode     = load  ;
            aOperand    = cop   ;
            aScalar     = 8'b0  ;
            endLine             ;
    end
endtask

task IXLOAD;
    begin   aOpCode     = ixload;
            aOperand    = val   ;
            aScalar     = 8'b0  ;
            endLine             ;
    end
endtask

// in CONTROLLER
task cVLOAD;
    input [7:0] value;
    begin   cOpCode     = load  ;
            cOperand    = val   ;
            cScalar     = value ;
    end
endtask

task cLOAD;
    input [7:0] value;
    begin   cOpCode     = load  ;
            cOperand    = mab   ;
            cScalar     = value ;
    end
endtask

task cRLOAD;
    input [7:0] value;
    begin   cOpCode     = load  ;
            cOperand    = mrl   ;
```

```
                cScalar     = value ;
        end
    endtask

    task cRILOAD;
        input [7:0] value;
        begin   cOpCode     = load  ;
                cOperand    = mri   ;
                cScalar     = value ;
        end
    endtask

    task cCLOAD; // scalar[1:0] = 00: reduction add
                 // scalar[1:0] = 01: reduction min
                 // scalar[1:0] = 10: reduction max
                 // scalar[1:0] = 11: reduction flag
        input [7:0] value;
        begin   cOpCode     = load  ;
                cOperand    = cop   ;
                cScalar     = value ;
        end
    endtask
```

Multiplication functions

```
/*****************************************************************************************
File name:      cgMULT.v
*****************************************************************************************/
// in ARRAY
    task VMULT;
        input [7:0] value;
        begin   aOpCode     = mult  ;
                aOperand    = val   ;
                aScalar     = value ;
                endLine             ;
        end
    endtask

    task MULT;
        input [7:0] value;
        begin   aOpCode     = mult  ;
                aOperand    = mab   ;
                aScalar     = value ;
                endLine             ;
        end
    endtask

    task RMULT;
        input [7:0] value;
        begin   aOpCode     = mult  ;
                aOperand    = mrl   ;
                aScalar     = value ;
                endLine             ;
        end
    endtask

    task RIMULT;
        input [7:0] value;
        begin   aOpCode     = mult  ;
                aOperand    = mri   ;
                aScalar     = value ;
                endLine             ;
        end
    endtask

    task CMULT;
        begin   aOpCode     = mult  ;
                aOperand    = cop   ;
```

```
                aScalar     = 8'b0  ;
                endLine             ;
        end
    endtask

// in CONTROLLER
    task cVMULT;
        input [7:0] value;
        begin   cOpCode     = mult  ;
                cOperand    = val   ;
                cScalar     = value ;
        end
    endtask

    task cMULT;
        input [7:0] value;
        begin   cOpCode     = mult  ;
                cOperand    = mab   ;
                cScalar     = value ;
        end
    endtask

    task cRMULT;
        input [7:0] value;
        begin   cOpCode     = mult  ;
                cOperand    = mrl   ;
                cScalar     = value ;
        end
    endtask

    task cRIMULT;
        input [7:0] value;
        begin   cOpCode     = mult  ;
                cOperand    = mri   ;
                cScalar     = value ;
        end
    endtask

    task cCMULT;
        input [7:0] value;
        begin   cOpCode     = mult  ;
                cOperand    = cop   ;
                cScalar     = value ;
        end
    endtask
```

Bitwise OR functions

```
/*****************************************************************************************
File name:      cgOR.v
*****************************************************************************************/
// in ARRAY
    task VOR;
        input [7:0] value;
        begin   aOpCode     = bwor  ;
                aOperand    = val   ;
                aScalar     = value ;
                endLine             ;
        end
    endtask

    task OR;
        input [7:0] value;
        begin   aOpCode     = bwor  ;
                aOperand    = mab   ;
                aScalar     = value ;
                endLine             ;
        end
```

```
    endtask

    task ROR;
        input [7:0] value;
        begin   aOpCode     = bwor  ;
                aOperand    = mrl   ;
                aScalar     = value ;
                endLine             ;
        end
    endtask

    task RIOR;
        input [7:0] value;
        begin   aOpCode     = bwor  ;
                aOperand    = mri   ;
                aScalar     = value ;
                endLine             ;
        end
    endtask

    task COR;
        begin   aOpCode     = bwor  ;
                aOperand    = cop   ;
                aScalar     = 8'b0  ;
                endLine             ;
        end
    endtask

// in CONTROLLER
    task cVOR;
        input [7:0] value;
        begin   cOpCode     = bwor  ;
                cOperand    = val   ;
                cScalar     = value ;
        end
    endtask

    task cOR;
        input [7:0] value;
        begin   cOpCode     = bwor  ;
                cOperand    = mab   ;
                cScalar     = value ;
        end
    endtask

    task cROR;
        input [7:0] value;
        begin   cOpCode     = bwor  ;
                cOperand    = mrl   ;
                cScalar     = value ;
        end
    endtask

    task cRIOR;
        input [7:0] value;
        begin   cOpCode     = bwor  ;
                cOperand    = mri   ;
                cScalar     = value ;
        end
    endtask

    task cCOR;
        input [7:0] value;
        begin   cOpCode     = bwor  ;
                cOperand    = cop   ;
                cScalar     = value ;
        end
    endtask
```

Reverse subtract functions

```
/*****************************************************************************************
File name:      cgRVSUB.v
*****************************************************************************************/
// in ARRAY
    task VRVSUB;
        input [7:0] value;
        begin   aOpCode     = rsub  ;
                aOperand    = val   ;
                aScalar     = value ;
                endLine             ;
        end
    endtask

    task RVSUB;
        input [7:0] value;
        begin   aOpCode     = rsub  ;
                aOperand    = mab   ;
                aScalar     = value ;
                endLine             ;
        end
    endtask

    task RRVSUB;
        input [7:0] value;
        begin   aOpCode     = rsub  ;
                aOperand    = mrl   ;
                aScalar     = value ;
                endLine             ;
        end
    endtask

    task RIRVSUB;
        input [7:0] value;
        begin   aOpCode     = rsub  ;
                aOperand    = mri   ;
                aScalar     = value ;
                endLine             ;
        end
    endtask

    task CRVSUB;
        begin   aOpCode     = rsub  ;
                aOperand    = cop   ;
                aScalar     = 8'b0  ;
                endLine             ;
        end
    endtask

// in CONTROLLER
    task cVRVSUB;
        input [7:0] value;
        begin   cOpCode     = rsub  ;
                cOperand    = val   ;
                cScalar     = value ;
        end
    endtask

    task cRVSUB;
        input [7:0] value;
        begin   cOpCode     = rsub  ;
                cOperand    = mab   ;
                cScalar     = value ;
        end
    endtask

    task cRRVSUB;
        input [7:0] value;
        begin   cOpCode     = rsub  ;
                cOperand    = mrl   ;
                cScalar     = value ;
```

```
        end
    endtask

    task cRIRVSUB;
        input [7:0] value;
        begin   cOpCode     = rsub  ;
                cOperand    = mri   ;
                cScalar     = value ;
        end
    endtask

    task cCRVSUB;
        input [7:0] value;
        begin   cOpCode     = rsub  ;
                cOperand    = cop   ;
                cScalar     = value ;
        end
    endtask
```

Reverse subtract with carry functions

```
/*****************************************************************************************
File name:      cgRVSUBC.v
*****************************************************************************************/
// in ARRAY
    task VRVSUBC;
        input [7:0] value;
        begin   aOpCode     = rsubc ;
                aOperand    = val   ;
                aScalar     = value ;
                endLine             ;
        end
    endtask

    task RVSUBC;
        input [7:0] value;
        begin   aOpCode     = rsubc ;
                aOperand    = mab   ;
                aScalar     = value ;
                endLine             ;
        end
    endtask

    task RRVSUBC;
        input [7:0] value;
        begin   aOpCode     = rsubc ;
                aOperand    = mrl   ;
                aScalar     = value ;
                endLine             ;
        end
    endtask

    task RIRVSUBC;
        input [7:0] value;
        begin   aOpCode     = rsubc ;
                aOperand    = mri   ;
                aScalar     = value ;
                endLine             ;
        end
    endtask

    task CRVSUBC;
        begin   aOpCode     = rsubc ;
                aOperand    = cop   ;
                aScalar     = 8'b0  ;
                endLine             ;
        end
    endtask
```

```
// in CONTROLLER
    task cVRVSUBC;
        input [7:0] value;
        begin   cOpCode     = rsubc ;
                cOperand    = val   ;
                cScalar     = value ;
        end
    endtask

    task cRVSUBC;
        input [7:0] value;
        begin   cOpCode     = rsubc ;
                cOperand    = mab   ;
                cScalar     = value ;
        end
    endtask

    task cRRVSUBC;
        input [7:0] value;
        begin   cOpCode     = rsubc ;
                cOperand    = mrl   ;
                cScalar     = value ;
        end
    endtask

    task cRIRVSUBC;
        input [7:0] value;
        begin   cOpCode     = rsubc ;
                cOperand    = mri   ;
                cScalar     = value ;
        end
    endtask

    task cCRVSUBC;
        input [7:0] value;
        begin   cOpCode     = rsubc ;
                cOperand    = cop   ;
                cScalar     = value ;
        end
    endtask
```

Search functions

This file is used only for the search version of the system.

```
/*****************************************************************************************
File name:      cgSEARCH.v
*****************************************************************************************/
// SEARCH
    task SEARCH; // search co-operand
        begin   aOpCode     = search;
                aOperand    = cop   ;
                aScalar     = 8'b0  ;
                endLine             ;
        end
    endtask

    task VSEARCH; // search value
        input [7:0] value;
        begin   aOpCode     = search;
                aOperand    = val   ;
                aScalar     = value ;
                endLine             ;
        end
    endtask

    task CSEARCH; // search co-operand
```

```
        begin   aOpCode     = csearch   ;
                aOperand    = cop       ;
                aScalar     = 8'b0      ;
                endLine                 ;
        end
    endtask

    task VCSEARCH; // search value
        input [7:0] value;
        begin   aOpCode     = csearch;
                aOperand    = val   ;
                aScalar     = value ;
                endLine             ;
        end
    endtask

// INSTERT
    task INSERT; // insert value in the first active position
        input [7:0] value;
        begin   aOpCode     = insert;
                aOperand    = val   ;
                aScalar     = value ;
                endLine             ;
        end
    endtask

    task  CINSERT; // insert co-operand in the first active position
        begin   aOpCode     = insert;
                aOperand    = cop   ;
                aScalar     = 8'b0  ;
                endLine             ;
        end
    endtask

// DELETE
    task  DELETE; // delete the first active position
        begin   aOpCode     = delete;
                aOperand    = val   ;
                aScalar     = 8'b0  ;
                endLine             ;
        end
    endtask
// READ
    task  READ; // shift right one position Boolean vector
        begin   aOpCode     = read  ;
                aOperand    = val   ;
                aScalar     = 8'b0  ;
                endLine             ;
        end
    endtask
```

Shift functions

```
/*****************************************************************************************
File name:      cgSHIFT.v
*****************************************************************************************/
// in ARRAY
    task SHRIGHTC; // shift right one bit position with carry
        begin   aOpCode     = shrightc  ;
                aOperand    = val       ;
                aScalar     = 8'b0      ;
                endLine                 ;
        end
    endtask

    task SHRIGHT; // shift right value  positions
        begin   aOpCode     = shright   ;
                aOperand    = val       ;
```

```
            aScalar     = 8'b0      ;
            endLine                 ;
    end
endtask

task SHARIGHT; // shift right arithmetic one bit position
    begin   aOpCode     = sharight  ;
            aOperand    = val       ;
            aScalar     = 8'b0      ;
            endLine                 ;
    end
endtask

task INSVAL; // insert value on the least positions
    input [7:0] value;
    begin   aOpCode     = insval;
            aOperand    = val   ;
            aScalar     = value ;
            endLine             ;
    end
endtask

// in CONTROLLER
task cSHRIGHTC; // shift right one bit position with carry
    begin   cOpCode     = shrightc  ;
            cOperand    = val       ;
            cScalar     = 8'b0      ;
    end
endtask

task cSHRIGHT; // shift right value  positions
    begin   cOpCode     = shright   ;
            cOperand    = val       ;
            cScalar     = 8'b0      ;
    end
endtask

task cSHARIGHT; // shift right arithmetic one bit position
    begin   cOpCode     = sharight  ;
            cOperand    = val       ;
            cScalar     = 8'b0      ;
    end
endtask

task cINSVAL; // insert value on the least positions
    input [7:0] value;
    begin   cOpCode     = insval;
            cOperand    = val   ;
            cScalar     = value ;
    end
endtask
```

Store functions

```
/*****************************************************************************************
File name:      cgSTORE.v
*****************************************************************************************/
// in ARRAY
    task ADDRLD; // addr[i] <= acc[i]
        begin   aOpCode     = store ;
                aOperand    = val   ;
                aScalar     = 8'b0  ;
                endLine             ;
        end
    endtask

    task STORE; // store acc[i] at arrayScalar
        input [7:0] value;
```

```
        begin   aOpCode     = store ;
                aOperand    = mab   ;
                aScalar     = value ;
                endLine             ;
        end
    endtask

    task RSTORE; // store acc[i] at addr[i] + arrayScalar
        input [7:0] value;
        begin   aOpCode     = store ;
                aOperand    = mrl   ;
                aScalar     = value ;
                endLine             ;
        end
    endtask

    task RISTORE; // store acc[i] at addr[i] + arrayScalar
                  // addr[i] <= addr[i] + contrScalar
        input [7:0] value;
        begin   aOpCode     = store ;
                aOperand    = mri   ;
                aScalar     = value ;
                endLine             ;
        end
    endtask

// in CONTROLLER
    task cADDRLD; // addr <= acc
        begin   cOpCode     = store ;
                cOperand    = val   ;
                cScalar     = 8'b0  ;
        end
    endtask

    task cSTORE; // store acc at contrScalar
        input [7:0] value;
        begin   cOpCode     = store ;
                cOperand    = mab   ;
                cScalar     = value ;
        end
    endtask

    task cRSTORE; // store acc at addr + contrScalar
        input [7:0] value;
        begin   cOpCode     = store ;
                cOperand    = mrl   ;
                cScalar     = value ;
        end
    endtask

    task cRISTORE; // store acc at addr + contrScalar
                   // addr <= addr + contrScalar
        input [7:0] value;
        begin   cOpCode     = store ;
                cOperand    = mri   ;
                cScalar     = value ;
        end
    endtask
```

Subtract functions

```
/*****************************************************************************************
File name:      cgSUB.v
*****************************************************************************************/
// in ARRAY
    task VSUB; // value sub:
               // acc[i] <= acc[i] - {(n-8){aScalar[7]}}, aScalar}
        input [7:0] value;
```

```
        begin   aOpCode     = sub   ;
                aOperand    = val   ;
                aScalar     = value ;
                endLine             ;
        end
    endtask

    task SUB; // absolute sub
              // acc[i] <= acc[i] - vectMem[i][aScalar[v-1:0]]
        input [7:0] value;
        begin   aOpCode     = sub   ;
                aOperand    = mab   ;
                aScalar     = value ;
                endLine             ;
        end
    endtask

    task RSUB; // relative sub:
               // acc[i]<=acc[i]-vectMem[i][addrVect[i]+aScalar[v-1:0]]
        input [7:0] value;
        begin   aOpCode     = sub   ;
                aOperand    = mrl   ;
                aScalar     = value ;
                endLine             ;
        end
    endtask

    task RISUB; // relative sub:
                // acc[i]<=acc[i]-vectMem[i][addrVect[i]+aScalar[v-1:0]]
                // addrVect[i] <= addrVect[i] + aScalar[v-1:0]
        input [7:0] value;
        begin   aOpCode     = sub   ;
                aOperand    = mri   ;
                aScalar     = value ;
                endLine             ;
        end
    endtask

    task CSUB; // co-operand sub:
               // acc[i] <= acc[i] - acc
        begin   aOpCode     = sub       ;
                aOperand    = cop       ;
                aScalar     = 8'b0  ;
                endLine                 ;
        end
    endtask

// in CONTROLLER
    task cVSUB; // value sub:
                // acc <= acc - {(n-8){cScalar[7]}}, cScalar}
        input [7:0] value;
        begin   cOpCode     = sub   ;
                cOperand    = val   ;
                cScalar     = value ;
        end
    endtask

    task cSUB; // immediate sub:
               // acc <= acc - mem[cScalar[s-1:0]]
        input [7:0] value;
        begin   cOpCode     = sub   ;
                cOperand    = mab   ;
                cScalar     = value ;
        end
    endtask

    task cRSUB; // relative sub:
                // acc <= acc - mem[addr + cScalar[s-1:0]]
        input [7:0] value;
        begin   cOpCode     = sub   ;
                cOperand    = mrl   ;
                cScalar     = value ;
```

```
        end
    endtask

    task cRISUB; // relative sub:
                 // acc <= acc - mem[addr + cScalar[s-1:0]]
                 // addr <= addr + cScalar[s-1:0]
        input [7:0] value;
        begin   cOpCode     = sub   ;
                cOperand    = mri   ;
                cScalar     = value ;
        end
    endtask

    task cCSUB;
        input [7:0] value;
        begin   cOpCode     = sub   ;
                cOperand    = cop   ;
                cScalar     = value ;
        end
    endtask
```

Subtract with carry functions

```
/*****************************************************************************************
File name:      cgSUBC.v
*****************************************************************************************/
// in ARRAY
    task VSUBC;
        input [7:0] value;
        begin   aOpCode     = subc  ;
                aOperand    = val   ;
                aScalar     = value ;
                endLine             ;
        end
    endtask

    task SUBC;
        input [7:0] value;
        begin   aOpCode     = subc  ;
                aOperand    = mab   ;
                aScalar     = value ;
                endLine             ;
        end
    endtask

    task RSUBC;
        input [7:0] value;
        begin   aOpCode     = subc  ;
                aOperand    = mrl   ;
                aScalar     = value ;
                endLine             ;
        end
    endtask

    task RISUBC;
        input [7:0] value;
        begin   aOpCode     = subc  ;
                aOperand    = mri   ;
                aScalar     = value ;
                endLine             ;
        end
    endtask

    task CSUBC;
        begin   aOpCode     = subc  ;
                aOperand    = cop   ;
                aScalar     = 8'b0  ;
                endLine             ;
```

```
        end
    endtask

// in CONTROLLER
    task cVSUBC;
        input [7:0] value;
        begin   cOpCode     = subc  ;
                cOperand    = val   ;
                cScalar     = value ;
        end
    endtask

    task cSUBC;
        input [7:0] value;
        begin   cOpCode     = subc  ;
                cOperand    = mab   ;
                cScalar     = value ;
        end
    endtask

    task cRSUBC;
        input [7:0] value;
        begin   cOpCode     = subc  ;
                cOperand    = mrl   ;
                cScalar     = value ;
        end
    endtask

    task cRISUBC;
        input [7:0] value;
        begin   cOpCode     = subc  ;
                cOperand    = mri   ;
                cScalar     = value ;
        end
    endtask

    task cCSUBC;
        input [7:0] value;
        begin   cOpCode     = subc  ;
                cOperand    = cop   ;
                cScalar     = value ;
        end
    endtask
```

Transfer functions

```
/*****************************************************************************************
File name:      cgTRANSFER.v
*****************************************************************************************/
// in CONTROLLER & ARRAY
    task VGET;
        begin   aOpCode     = vload ;
                aOperand    = val   ;
                aScalar     = 8'b0  ;
                endLine             ;
        end
    endtask

    task VSEND;
        begin   aOpCode     = vstore;
                aOperand    = val   ;
                aScalar     = 8'b0  ;
                endLine             ;
        end
    endtask
```

Bitwise exclusive OR functions

```
/*****************************************************************************************
File name:      cgXOR.v
*****************************************************************************************/
// in ARRAY
    task VXOR;
        input [7:0] value;
        begin   aOpCode     = bwxor ;
                aOperand    = val   ;
                aScalar     = value ;
                endLine             ;
        end
    endtask

    task XOR;
        input [7:0] value;
        begin   aOpCode     = bwxor ;
                aOperand    = mab   ;
                aScalar     = value ;
                endLine             ;
        end
    endtask

    task RXOR;
        input [7:0] value;
        begin   aOpCode     = bwxor ;
                aOperand    = mrl   ;
                aScalar     = value ;
                endLine             ;
        end
    endtask

    task RIXOR;
        input [7:0] value;
        begin   aOpCode     = bwxor ;
                aOperand    = mri   ;
                aScalar     = value ;
                endLine             ;
        end
    endtask

    task CXOR;
        begin   aOpCode     = bwxor ;
                aOperand    = cop   ;
                aScalar     = 8'b0  ;
                endLine             ;
        end
    endtask

// in CONTROLLER
    task cVXOR;
        input [7:0] value;
        begin   cOpCode     = bwxor ;
                cOperand    = val   ;
                cScalar     = value ;
        end
    endtask

    task cXOR;
        input [7:0] value;
        begin   cOpCode     = bwxor ;
                cOperand    = mab   ;
                cScalar     = value ;
        end
    endtask

    task cRXOR;
        input [7:0] value;
        begin   cOpCode     = bwxor ;
                cOperand    = mrl   ;
                cScalar     = value ;
```

```
    end
endtask

task cRIXOR;
    input [7:0] value;
    begin   cOpCode     = bwxor ;
            cOperand    = mri   ;
            cScalar     = value ;
    end
endtask

task cCXOR;
    input [7:0] value;
    begin   cOpCode     = bwxor ;
            cOperand    = cop   ;
            cScalar     = value ;
    end
endtask
```

Bibliography

[Amdahl '64] Gene Amdahl, Gerrit Blaauw, and Fred Brooks (1964) Architecture of the IBM System, *IBM Journal of Research and Development* **8**(2) pp. 87-101

[Banerjee '19] Niloy Banerjee (2019) A Versal ACAP is Significantly Different Than a Regular FPGA or SoC, *BISinfotech*, January 11.

[Batcher '68] K. E. Batcher (1968) Sorting networks and their applications, *AFIPS Conference Proceedings: 1968 Spring Joint Computer Conference, Atlantic City, NJ, USA, 30 April - 2 May*, pp. 307-314.

[Borges '52] Jorge Luis Borges (1952) El idioma analítico de John Wilkins (The Analytical Language of John Wilkins), *Otras Inquisiciones (1937–1952)*, Sur, 1952.

[Blaauw '64] Gerrit Anne Blaauw and Fred P. Brooks, Jr. (1964) The structure of SYSTEM/360, Part I: Outline of the logical structure, *IBM Systems Journal* **3**(2):119-135.

[Blakeslee '79] T. R. Blakeslee (1979) *Digital Design with Standard MSI and LSI*, John Wiley & Sons.

[Booth '67] T. L. Booth (1967) *Sequential Machines and Automata Theory*, John Wiley & Sons, Inc.

[Cardoso '10] Joao M. P. Cardoso, Pedro C. Diniz, Markus Weinhardt, (2010) Compiling for Reconfigurable Computing: A Survey, *ACM Computing Surveys*, **42**(4):13:1-13:65.

[Casti '92] John L. Casti (1992) *Reality Rules: II. Picturing the World in Mathematics - The Frontier*, John Wiley & Sons, Inc.

[Chaitin '77] Gregory Chaitin (1977) Algorithmic Information Theory, *IBM J. Res. Develop.*, **21**:350-359.

[Chomsky '56] Noam Chomsky (1956) Three Models for the Description of Languages, *IRE Trans. Inf. Theory*, **2**(3):113-124.

[Chomsky '59] Noam Chomsky (1959) On Certain Formal Properties of Grammars, *Information and Control*, **2**(2):137-167.

[Chomsky '63] Noam Chomsky (1963) Formal Properties of Grammars, *Handbook of Mathematical Psychology*, Wiley, New-York.

[Church '36] Alonzo Church (1936) An unsolvable problem of elementary number theory. *The American Journal of Mathematics* **58**(2):345-363.

[Clare '72] C. Clare (1972) *Designing Logic Systems Using State Machines*, Mc Graw-Hill, Inc.

[Compton '02] K. Compton, S. Hauck (2002) Reconfigurable computing: a survey of systems and software,*ACM Comput. Surv.*, **34**(2):171–210.

[Cooley '65] James W. Cooley and John W. Tukey (1965) An algorithm for the machine calculation of complex Fourier series, *Mathemetics of Computation* **19**(90):297-301.

[Copeland '06] Jack B. Copeland, ed. (2006) *Colossus: The Secrets of Bletchley Park's Codebreaking Computers* Oxford University Press.

[Crockett '15] Louise H. Crockett, Ross A. Elliot, Martin A. Enderwitz, David Northcote (2015) *The Zinq Book Tutorials for Zibo and ZedBoard*, Dept. of Electronic and Electrical engineering, University of Strathclyde, Glasgow, Scotland, UK.

[Davis '04] M. Davis (2004) *The Undecidable. Basic Papers on Undecidable Propositions, Unsolvable Problems and Computable Functions*. Dover Publications, Inc., Mineola, New-York.

[Dijkstra '65] E. W. Dijkstra (1965) Co-operating sequential processes. *Programming Languages* Academic Press, New York, pp. 43–112. Reprinted from: Technical Report EWD-123, Technological University, Eindhoven, the Netherlands.

[Edwards '21] Chris Edwards (2021) Moore's Law: What Comes Next?, *Communications of the ACM*, **64**(2):12-14.

[Einspruch '91] N. G. Einspruch, J. L. Hilbert (1991) *Application Specific Integrated Circuits (ASIC) Technology*, Academic Press, Inc.

[Ercegovac '04] Miloš D. Ercegovac, Tomás Lang (2004) *Digital Arithmetic*, Morgan Kaufman.

[Estrin '60] G. Estrin (1960) Organization of Computer Systems – The Fixed Plus Variable Structure Computer, *Proc. Western Joint Computer Conf., Western Joint Computer Conference*, New York, pp. 33–40.

[Fortune '78] S. Fortune, and J. C. Wyllie (1978) Parallelism in random access machines, *Conference Record of the Tenth Annual ACM Symposium on Theory of Computing*, pp. 114–118.

[Garfinkel '15] S. L. Garfinkel, R. H. Grunspan (2018) *The Computer Book*, Sterling.

[Gheolbanoiu '14] Alexandru Gheolbanoiu, Dan Mocanu, Radu Hobincu, Lucian Petrica (2014) Cellular Automaton pRNG with a Global Loop for Non-Uniform Rule Control, *18th International Conference on Ciruits, Systems, Communications and Computers (CSCC 2014)*, Santorini Island, Greece, July 17-21, vol. II, pp. 415-420. At: `http://www.euroment.org/library/2014/santorini/bypaper/COMPUTERS/COMPUTERS2-15.pdf`

[Goldschlager '82] L. M. Goldschlager (1982) A universal interconnection pattern for parallel computers, *Journal of the ACM* **29**(4):1073-1086.

[Greaves '08] D. J. Greaves (2008) Metastability Theory, University of Cambridge, Computer Laboratory. At: `https://www.cl.cam.ac.uk/teaching/1011/SysOnChip/slides/sp3socparts/zhp6e41885b8.html`

[Hennie '68] F. C. Hennie (1968) *Finite-State Models for Logical Machine*, John Wiley & Sons, Inc.

[Hilbert 1900] David Hilbert (1902) Mathematical problems, *Bull. Amer. Math. Soc*. **8**(10):437–479.

[Hilbert & Ackermann '28] David Hilbert, Wilhelm Ackermann (1928) *Grundzüge der theoretischen Logik* (Principles of Mathematical Logic). Springer-Verlag.

[Hillis '85] W. D. Hillis (1985) *The Connection Machine*, The MIT Press, Cambridge, Mass.

[Isaacson '85] W. Isaacson (2014) *The Innovators: How a Group of Hackers, Geniuses, and Geeks Created the Digital Revolution*, Simon & Schuster.

[Kaeslin '01] Hubert Kaeslin (2008) *Digital Integrated Circuit Design*, Cambridge Univ. Press.

[Kastner '20] Ryan Kastner, Janarbek Matai, Stephen Neuendorffer (2020) *Parallel Programming for FPGAs*, Kastner Research Group.

[Keeth '01] Brent Keeth, R. Jacob Baker (2001) *DRAM Circuit Design. A Tutorial*, IEEE Press.

[Kelly '20] Paul H. J. Kelly (2020) "Turing Tariff" Reduction: architectures, compilers and languages to break the universality barrier. At: `https://www.doc.ic.ac.uk/~phjk/Presentations/2020-06-24-DoCLunch-PaulKelly-TuringTaxV04.pdf`

[Kleene '36] Stephen Kleene (1936) General recursive functions of natural numbers, *Mathematische Annalen* **112**(5):727-742.

[Knuth '73] D. E. Knuth (1973) *The Art of Programming. Sorting and Searching*, Addison-Wesley.

[Kung '79] H. T. Kung, C. E. Leiserson (1979) Algorithms for VLSI processor arrays, in [Mead '79].

[Ladner '80] R. E. Ladner, M. J. Fischer (1980) Parallel prefix computation, *Journal of the Association for Computing Machinery*, **27**(4):831-838.

[Matita '13] Mihaela Maliţa, Gheorghe M. Ştefan (2013) Control Global Loops in Self-Organizing Systems, *ROMJIST*, **16**(2–3):177-191. At: `http://www.imt.ro/romjist/Volum16/Number16\_2/pdf/05-Malita-Stefan2.pdf`

[McCulloch '43] Warren S. McCulloch, Walter H. Pitts (1943) A Logical Calculus of the Ideas Immansnt in Nervous Activity, *Bulletin of Mathematical Biiophysics*, **5**:115-133.

[Omondi '94] Amos R. Omondi (1994) *Computer Arithmetic. Algorithm, Architecture and Implementation*, Prentice Hall.

[Palnitkar '96] Samir Palnitkar (1996) *Verilog HDL. AGuide to Digital Design and Synthesis*, SunSoft Press.

[Parberry 87] Ian Parberry (1987) *Parallel Complexity Theory*. Research Notes in Theoretical Computer science. Pitman Publishing, London.

[Parberry 94] Ian Parberry (1994) *Circuit Complexity and Neural Networks*, The MIT Press.

[Patterson '05] David A. Patterson, John L.Hennessy (2005) *Computer Organization & Design. The Hardware/-Software Interface*, Third Edition, Morgan Kaufmann.

[Post '36] E. Post (1936) Finite combinatory processes. Formulation I, *The Journal of Symbolic Logic*, **1**(3):103-105.

[Pratt '74] V. R. Pratt, M. O. Rabin, and L. J. Stockmeyer (1974) A characterization of the power of vector machines. *Proceedings of STOC'1974*, pp. 122–134.

[Prince '99] Betty Prince (1999) *High Performance Memories. New architecture DRAMs and SRAMs evolution and function*, John Wiley & Sons.

[Rafiquzzaman '05] Mohamed Rafiquzzaman (2005) *Fundamentals of Digital Logic and Microcomputer Design*, Fifth Edition, Wiley – Interscience.

[Sharma '97] Ashok K. Sharma (1997) *Semiconductor Memories. Techology, Testing, and Reliability*, Wiley – Interscience.

[Sharma '03] Ashok K. Sharma (2003) *Advanced Smiconductor Memories. Architectures, Designs, and Applications*, Whiley-Interscience.

[Sperberg-McQueen '04] C. M. Sperberg-McQueen (2004) *Notes on finite state automata with counters*. At: `https://www.w3.org/XML/2004/05/msm-cfa.html`

[1] P. M. Spira (1971) On time-Hardware Complexity Tradeoff for Boolean Functions, *Preceedings of Fourth Hawaii International Symposium on System Sciences*, pp. 525-527.

[Stefan '98] Gheorghe Ştefan (1998) "Looking for the Lost Noise", *CAS '98 Proceedings*, pp. 579-582. At: `http://arh.pub.ro/gstefan/CAS98.pdf`

[Ştefan '06] Gheorghe Ştefan (2006) A Universal Turing Machine with Zero Internal States, *Romanian Journal of Information Science and Technology*, **9**(3):227-243.

[Stefan '14] Gheorghe M. Stefan, Mihaela Malita (2014) Can One-Chip Parallel Computing Be Liberated From Ad Hoc Solutions? A Computation Model Based Approach and Its Implementation, *18th International Conference on Circuits, Systems, Communications and Computers (CSCC 2014)*, pp. 582-597. At: `http://www.inase.org/library/2014/santorini/COMPUTERS1.pdf`

[Stefan '22] Gheorghe. M. Ştefan (2022) *Loops & Complexity in DIGITAL SYSTEMS. Lecture Notes on Digital Design in Giga-Gate/Chip Era*, 2022 version. At: `http://users.dcae.pub.ro/~gstefan/2ndLevel/teachingMaterials/0-BOOK.pdf`

[Tegmark '17] Max Tegmark (2017) *LIFE 3.0. Being Human in the Age of Artificial Intelligence*, Alfred A. Knopf.

[Todman '06] T.J. Todman, G.A. Constantinides, S.J.E. Wilton, O. Mencer, W. Luk and P.Y.K. Cheung (2006) Reconfigurable computing: architectures and design methods, in Bashir M. Al-Hashimi (ed.), *System-on-Chip: Next Generation Electronics*, IET Digital Library, pp. 193-207.

[Turing '36] Alan Turing (1936) and (1937), On computable numbers with an application to the Eintscheidungsproblem, *Proceedings of the London Mathematical Society*, **42**(1):230–256 and a correction in **43**(6):544–546.

[Uyemura '02] John P. Uyemura (2002) *CMOS Logic Circuit Design*, Kluver Academic Publishers.

[Neumann '45] J. von Neumann, (1945) First draft of a report on the EDVAC, reprinted in *IEEE Annals of the History of Computing* **15**(4):27-75, 1993.

[Ward '90] S. A. Ward, R. H. Halstead (1990) *Computation Structures*, The MIT Press, McGraw-Hill Book Company.

[Waksman '68] Abraham Waksman (1968) A permutation network, *Journal of the ACM* **15**(1):159-163.

[Weaver '09] Vincent Weaver, Sally McKee (2009) Code Density Concerns for New Architectures, *2009 IEEE International Conference on Computer Design*, pp. 459-464.

[Wulf '95] Wm. A. Wulf, Sally A. McKee (1995) Hitting the memory wall: implications of the obvious, *ACM SIGARCH Computer Architecture News*, **23**(1):20-24.

[Xilinx '18] *** (2018) *7 Series DSP48E1 Slice. User Guide*, UG479 (v1.10) March 27.