Simulation Manual
for
Configurable MapReduce Accelerator

(2.0)

Gheorghe M. Ştefan

The emergence of the hybrid computation domain is an incipient process. Roughly speaking it is about a system containing two parts: a standard computing engine, used as host and to run the complex part of the code[1], and an accelerator, for running the intense part of the code[2]. While for the host there are few consecrated solutions ("from the shelf" mono- or multi-core processors), for the position of the accelerator part compete few solutions. Some of them have a considerable advance: various GPUs (such as Nvidia, AMD's ATI), MICs[3] (such as Intel's Xeon Phi, Adapteva's Epiphany), or FPGA implemented circuits. GPU solutions are limited because the architecture is biased due to the graphic functionality legacy, while MIC processors are limited because of their *ad hoc* structured organization. The FPGA solutions look the most promising because of their flexibility. The flexibility is used to provide well fitted solutions and in the same time it helps in the prototyping process when the final target is an ASIC implemented hybrid system.

The only drawback in using FPGAs is the requirement of circuit design abilities in defining and implementing the circuit used as accelerator or a very well optimized "silicon compiler" which takes the code and generate the circuit. A good compromise is to use a predefined framework for the FPGA design as a ***configurable programmable parallel system***. In the following, a configurable Map-Reduce programmable structure [11] is considered as a generic accelerator engine.

In the second section the structure of the simulated system is described. The assembly language is described in the third section. The fourth section contains examples. The fifth section develops a library of functions. The last section is reserved for upgrades expected as outcomes of the evaluation process.

[1] A code is said complex if its size is in the same range with its execution time.

[2] A code is said intense if its size is much smaller than its execution time.

[3] Many Integrated Core

Contents

1 Functional Electronics

The evolution of electronics tends naturally toward the emergence of systems where circuits interleave with information in order to achieve high functional capabilities. The action of Moore's Low provides big sized circuits, but there is not a "Moore Low" for the functional complexity. Structures get big but only if they remain simple, characterized by repetitive patterns. The complexity comes only if the flexible informational structures can be inserted in the big pattern-based physical structures.

Indeed, it is easy to design, verify, implement and test silicon chips with billions of transistors, but only if the description of these circuits are kept in reasonable limits. If the structure is big & complex (the description of the circuit has the size in the same magnitude order with the size of the circuits), then it is impossible to provide a verifiable design and a credible test procedure for it.

In this context, Functional Electronics is the emergent domain of the functionally big & complex systems built by tightly interleaving pattern-based big circuits with complex information. Thus:

Circuit & Information = Functional Electronics

Because circuits a naturally parallel engines, *Functional Electronics* is equivalent in the commercial space with

Parallel Embedded Systems.

The accelerator described in the following is a typical product of *Functional Electronics* with applications in the *Parallel Embedded System* domain. As circuit it is based on a *N*-order digital system with a scan super (global) loop, a reduction super (global) loop and a controlled super (global) loop [12]. As computation engine, it is based on the synergy between Stephen Kleene's mathematical model of computation and John Backus's Functional Programming Systems [11].

The physical implementation of the accelerator provides, in 28 nm technology, for less than 10 Watt:

- 2 32-bit TOPS
- 1 32-bit TFLOPS for hi-intense flop applications
- 2 16-bit TFLOPS for hi-intense flop applications

The degree of parallelism depends on the application. For linear algebra domain it tends to be more than 90%. For molecular dynamics it is already proved to be over 75%. In the Artificial Neural Network domain the performance of our programmable solution is similar to those of ASIC's.

Part I
SIMULATOR

2 The General Description of the Hybrid System

The structure of the hybrid system we consider (see Figure 1) consists of:

- HOST SYSTEM: a general purpose computing system with Harvard architecture
- ACCELERATOR: a parallel engine

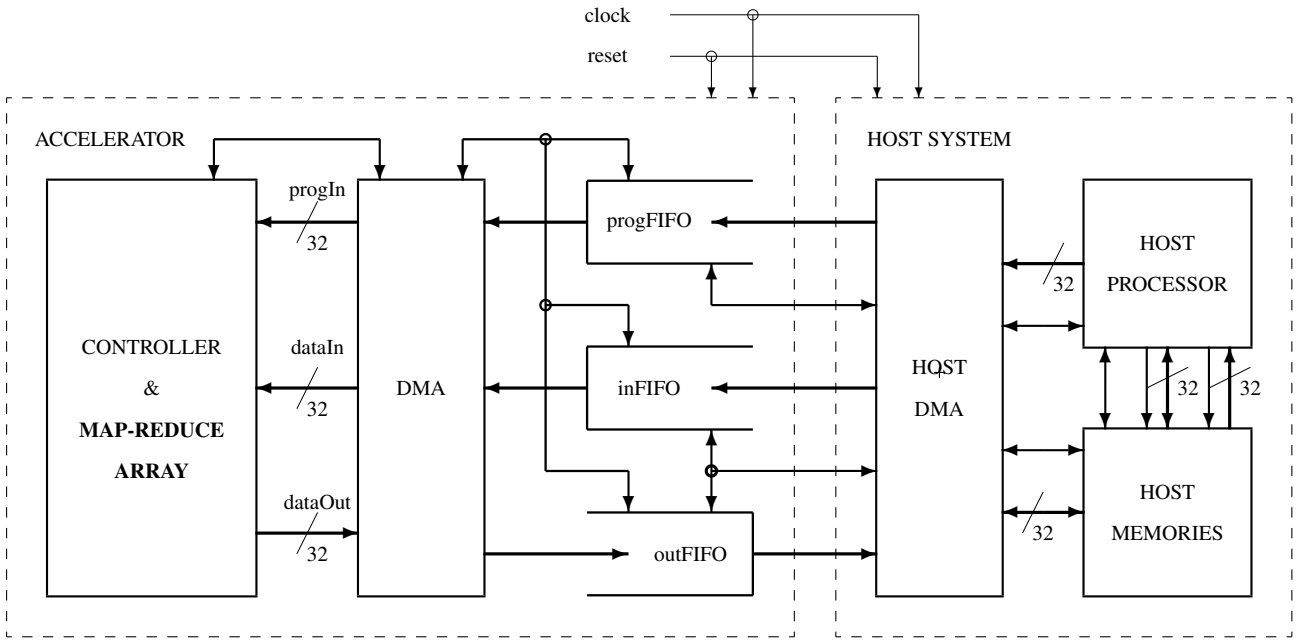

Figure 1: The hybrid system: HOST SYSTEM & ACCELERATOR.

The HOST SYSTEM 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.

2.1 Host System

2.2 Host System's Structure

The host system (see Figure 1) consists of:

- HOST PROCESSOR, a simple, general purpose RISC processor with Harvard architecture.
- HOST DMA, an interface with ACCELERATOR having two counter-extended automata:
  - one to control the program and commands transfer from HOST PROCESSOR system to ACCELERATOR
  - another to manage data transfer between ACCELERATOR and data section of MEMORY
- MEMORY, the memory system containing two memories, one for programs and another for data.

2.3 Host System's Architectural Image

The storage resources of the host system are:

- host program memory (part of MEMORY):
  `reg [31:0] hostProgMem[0:1023]`
- host data memory (part of MEMORY):
  `reg [31:0] hostDataMem[0:1023]`
- register file:
  `reg [31:0] rf[0:31]`
- program counter:
  `reg [31:0] pc`
- cycles counter:
  `reg [31:0] hostCounter`

2.4 Accelerator

2.4.1 Accelerator's Structure

The accelerator (see Figure 1) consists of:

- DMA: Direct Memory Access controller which receives programs and commands from Host, through progFIFO, or manages data transfers between MEMORY and ACCELERATOR; it consists of two counter-extended finite automata:
  - one for managing program and commands transfer from HOST to the ACCELERATOR
  - another to manage data transfer between ACCELERATOR and MEMORY and to send messages to HOST
- progFIFO: used to transfer the program which run on ACCELERATOR and to trigger the functions (programs) programmed on ACCELERATOR
- inFIFO: used to receive data from the External Memory
- outFIFO: used to
  - send back to MEMORY the result of computation
  - send requests of data from ACCELERATOR to the external memory MEMORY
  - to send simple messages to HOST (such as end of running the requested function)
- CONTROLLER & MAP-REDUCE ARRAY: is in fact the accelerator.

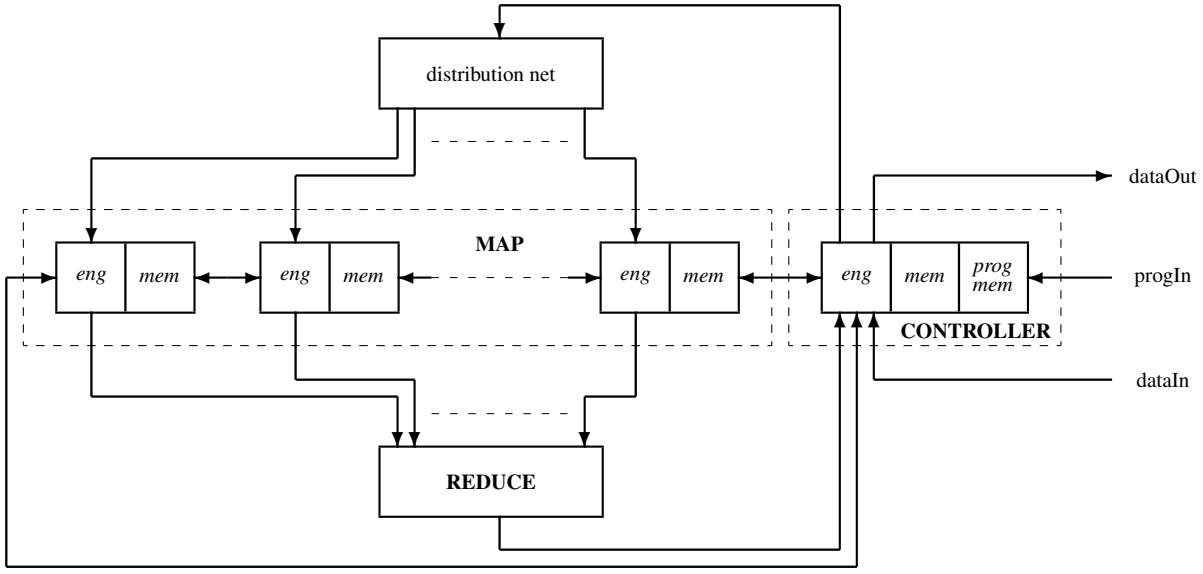

Figure 2: The functional organization of the accelerator's core.

The computational part of the accelerator (see Figure 2) performs functions dealing with scalar or vectors and consists of a four parts:

- CONTROLLER: performing functions defined on scalars with values in scalars; it has a Hardware RISC architecture with its program memory (*prog mem*), data memory (*mem*) and execution unit (*eng*)
- distribution net: is a log-depth pipe-lined network used to distribute instruction, data and address form CONTROLLER to the cells of the MAP section

- MAP section: performing functions defined on vectors with values in vectors; it is a linear array of cells each with its own data memory and execution unit similar with those of the controller
- REDUCTION network: performing functions defined on vectors with values in scalars; it is a *log*-depth circuit.

The parameters used to configure the ACCELERATOR are the following:

```
parameter
    n = 32   , // word size
    x = 10   , // index size -> 2^x = 1024 cells
    v = 11   , // vector memory address size -> 2048 1024-component vectors
    s = 9    , // scalar memory address size -> 512 32-bit scalars
    p = 8    , // program memory address size -> 256 pairs of instructions
    c = 8    , // value size in instruction
    a = 5      // (size of activation counter -> 32 embedded WHEREs)
```

for an engine characterized by:

- 32-bit word
- 1024-cell array
- 2048-word local memory in each cell, which translates in a Vector Memory of 2048 vectors of 1024 32-bit scalar each
- 512-word Data Memory in CONTROLLER
- 256-instruction Program Memory in CONTROLLER
- 8-bit immediate in instruction for both, array and controller.

2.4.2 Accelerator's Architectural Image

The user's image of the accelerator system is presented in Figure 3. It consists of the memory resources accessible at the level of the assembly language. There are two levels of storage in the system we simulate:

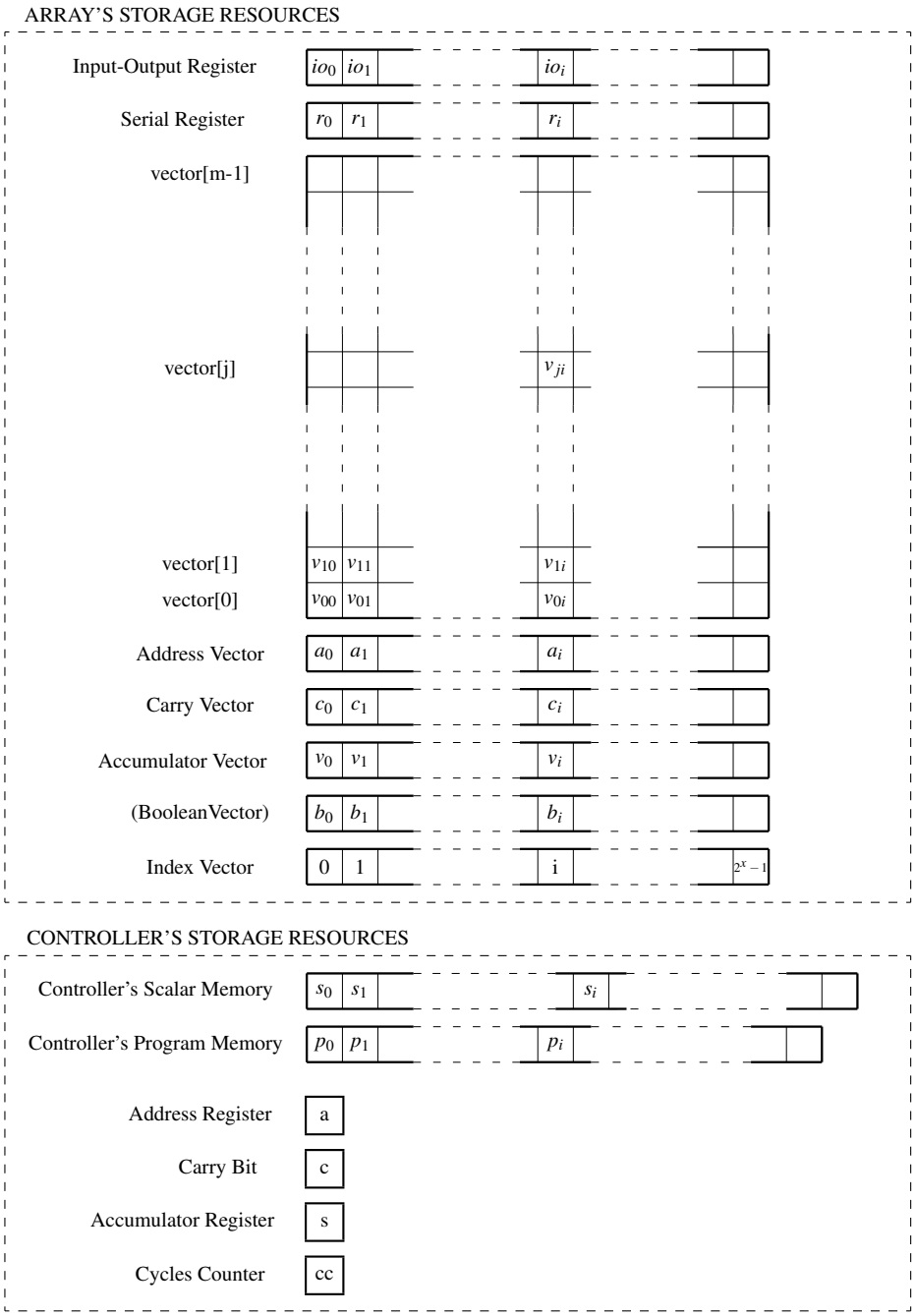

Figure 3: The users view of the architecture.

- Controller's Memory resources are:
  - Accumulator Register: is a 32-bit register in the accumulator-based execution unit; it provides one of the operand and and stores the result of the unary and binary operations performed by the execution unit
    `reg [n-1:0] acc`

- Carry Bit: is a 1-bit register whose content is actualized at each arithmetic operation (shifts are arithmetic operations)
  `reg cr`
- Scalar Memory: is the data memory of the controller; it provides, by the rule,, the second operand for binary operations.
  `reg [n-1:0] mem[0:(1<<s)-1]`
- Address Register: is a register used to form the address for Scalar Memory when relative addressing mode is used; its content is added with the immediate value provided by controller's instruction
  `reg [s-1:0] addr`
- Programm Memory: contains at each location a pair of instructions, one for CONTROLLER and another for MAP-REDUCE array; it is loaded under the control of DMA unit
  `reg [31:0] progMem[0:(1<<p)-1]`

- Array's Memory resources are:
  - Boolean Vector: is a $p = 2^x$ one-bit words vector; if $b_i = 1$ the $cell_i$ is active, i.e., the instruction received from controller is executed, else, if $b_i = 0$ the $cell_i$ is inactive
    `reg boolVect[0:(1<<x)-1]`
  - Accumulator vector: is a p n-bit words vector distributed along the p cells of the MAP section; its components are used as accumulators in the execution units of each cell
    `reg [n-1:]  accVect[0:(1<<x)-1]`
  - Carry Vector: is a p one-bit words vector distributed along the p cells of the MAP section; its content is updated at each arithmetic and shift operation
    `reg crVect[0:(1<<x)-1]`
  - Address Vector: is a vector distributed along the p cells of the MAP section; it is used for relative addressing the data memory of each cell
    `reg [v-1:0] addrVect[0:(1<<x)-1]`
  - Vector Memory: contains $m = 2^v$ p-component vectors
    `reg [n-1:0] vectMem[0:(1<<x)-1][0:(1<<v)-1]`
    as follow
    - vector[0]: `reg [n-1:0] vectMem[0:(1<<x)-1][0]`
    - vector[1]: `reg [n-1:0] vectMem[0:(1<<x)-1][1]`
    - ...
    - vector[i]: `reg [n-1:0] vectMem[0:(1<<x)-1][j]`
    - ...
    - vector[p-1]: `reg [n-1:0] vectMem[0:(1<<x)-1][p-1]`
  - Serial Register: is a serial-parallel register distributed along the MAP's cells; each of the p cells contains a n-bit parallel register serially connected in the previous and in the next cell
    `reg [n-1:0] serialReg[0:(1<<x)-1]`
  - Index Vector: is a constant vector used to index the p cells of the MAP section
    `reg [x-1:0] ixVect[0:(1<<x)-1]`
  - Input-Output Register: is used to insert `inData` or to extract `outData` (see Figure 1) in/from array of cells
    `reg [n-1:0] ioReg [0:(1<<x)-1]`

There are the following five operation modes in the storage space just described:

1. ***vector to scalar mode***: is performed in REDUCTION section starting from `accVect` and providing a value in `acc` or back to the MAP section.
   Important note: the REDUCTION unit is a *log*-depth circuit with a latency $\lambda(p) = 1 + 0.5log_2 p$. Therefore, any scalar generated at the output of the REDUCTION unit is valid with a λ cycles delay, i.e., between the instruction which set the content of `accVect` submitted to a reduction operation and the instruction which uses the result of the reduction operation whatever λ instructions must be inserted; if nothing to do, then no operation instructions are welcome.

2. ***scalar-scalar to scalar mode***: is performed in CONTROLLER between `acc` and `mem[i]` or immediate value contained in instruction or `coOperand` with result in `acc`; `coOperand` is the scalar value received, with λ cycles latency, through REDUCTION unit from MAP section

3. ***vector-scalar to vector***: is performed in MAP section between `accVect` and immediate value contained in instruction or `coOperand` with result in `accVect`; `coOperand` is the scalar value received from CONTROLLER or, with λ cycles latency, from the REDUCTION unit

4. ***vector-vector to vector mode***: is performed in MAP section between `accVect` and `vectMem[j]`

5. ***vector to vector mode***: is performed in MAP section on `accVect`

3 The Assembly Languages

3.1 System Initialization

3.1.1 External memory & its initialization

The external data memory (part of MEMORY) is managed, in the actual system, by the host engine. For the simulation purpose this memory is positioned in the module `host` as `reg[31:0] hostDataMem[0:1023]` in `hostMemories.v`.

The assembler will load the program for ACCELERATOR starting from the address in `hostDataMem`. The external memory could be initialized with data, if needed, by loading the file `initialData.txt`. The file is defined in hexa and has the form:

```
@yyy

yyyyyyyy
yyyyyyyy
...
yyyyyyyy
```

where `y` is a hexa symbol. The file starts with `yyy` which represents the starting address in `extMem`. This address must be carefully attributed in concordance with the size of the program loaded starting form the address 0.

3.1.2 System initialization

To start the system the user must write two programs, one for HOST and another for ACCELERATOR.

The host's program must be stored in the file `00_hostProgram`, and the accelerator's program must be stored in the file `00_theProgram`. The simulator will assemble first the program for accelerator then for the host.

Example 3.1 *The content of the file* `00_hostProgram.v`

```
/*****************************************************************************
File: 00_hostProgram.v
    load the program in CONTROLLER's program memory
    initialize external data memory with data from array
        RUN(storeMatrix, from, to, columns, lines)
        RUN(storeVector, from, to, columns, lines)
    then run the selected program
    halt
*****************************************************************************/
                                    // LOAD THE PROGRAM
        hTRANS(0);                  // initialize transfer to zero from zero
        hWAITINT;                   // wait initialization end
        hWAITACC;                   // signals to the host the end of program load
                                    // INITIALIZE DATA IN THE EXTERNAL MEMORY
        RUN(`MSTORE, 4, 16, 9, 9);  // store a 9x9 matrix in host's data memory
        RUN(`MSTORE, 8, 0, 9, 1);   // store a 9-component vector in host's data memory

        //`include "01_hostTest.v"
        //`include "02_storeVector.v"
        `include "04_libraryHostTest.v"

        hHALT;
/*****************************************************************************/
```

The program transfers the program for ACCELERATOR from data memory of HOST intro the program memory of CONTROLLER. Then stores in the host's data memory a matrix and a vector. The file `04_libraryHosttest.v` *contains the application program which runs on the data stored in the data memory of HOST.*

The content of the file `00_theProgram.v`*:*

```
/*******************************************************************************
The file: 00_theProgram.v
    initialize a 16x16 matrix in vector memory starting with vect[4],
    then run the selected program, if any, from the included code
*******************************************************************************/
            cPLOAD;         NOP;

            cNOP;           ACTIVATE;
            cVLOAD(16);     VLOAD(3);
            cNOP;           ADDRLD;
            cNOP;           VLOAD(255);
    LB(0);  cVSUB(1);       VADD(1);
            cBRNZ(0);       RISTORE(1);
            cTRUN(7);       NOP;
            cHALT;          NOP;

    `include "04_libraryTest.v"

            cPRUN(0);       NOP;
/*******************************************************************************/
```

◇

3.2 Host's Assembly Language

3.2.1 Instruction Format for Host ISA

There are two forms for the host's instructions:

```
inst[31:0] = {0, opCode[4:0], dest[4:0], left[4:0], hvalue[15:0]} |
             {1, opCode[4:0], dest[4:0], left[4:0], right[4:0], 11'b0};
```

where:

- `inst[31]` is used to select as right operand the content of the register file or the immediate value, as follows: `rigthOp = inst[31] ? rf[right] : {{16*{value[15]}}, hvalue[15:0]}`
- `opCode[4:0]` selects the operation executed by the host processor
- `left[4:0]` selects the left operand from the register file
- `right[4:0]` selects the right operand from the register file when `inst[31] = 1`
- `dest[4:0]` selects the destination in the register file
- `hvalue[15:0]` represents the immediate value used, when `inst[31] = 0`, as constant or as address.

3.2.2 Data Move Instructions

move :
```
hMOVE(destReg, leftReg) :
rf[destReg] <= rf[leftReg]
```

load immediate :
```
hVAL(destReg, hvalue) :
rf[destReg] <= {{16{hvalue[15]}}, hvalue[15:0]}
```

load immediate unsigned :
```
hUVAL(destReg, hvalue) :
rf[destReg] <= {{16'b0}, hvalue[15:0]}
```

insert immediate : used to generate, in conjunction with `hVAL`, a 32-bit value
```
hINSVAL(destReg, leftReg, hvalue) :
rf[destReg] <= {rf[leftReg][15:0], hvalue}
```

compose : used to compose a 32-bit value from two 16-bit values
```
hCOMPOSE(destReg, leftReg, rightReg) :
rf[destReg] <= {rf[leftReg][15:0], rf[rightReg][15:0]}
```

immediate store :
```
hISTORE(leftReg, hvalue) :
dataMem[hvalue] <= rf[leftReg]
```

direct store :
```
hSTORE(leftReg, rightReg) :
dataMem[rf[rightReg]] <= rf[leftReg]
```

immediate read : read command, provide in the next clock cycle the content addressed in the host's data memory
```
hIGET(hvalue) :
dataMemOut <= dataMem[rf[hvalue]]
```

direct read : read command, provide in the next clock cycle the content addressed in the host's data memory
```
hGET(rightReg) :
dataMemOut <= dataMem[rf[rigthReg]]
```

load : load in the destination register the content provided by `hIGET` or `hGET`
```
hLOAD(destReg) :
rf[destReg] <= dataMemOut
```

program transfer : transfers the program stored in the data memory, starting from the address `hvalue`, to the program memory of CONTROLLER starting from the address 0
```
hTRANS(hvalue)
```

data push : push {{16{hvalue[15]}}, hvalue[15:0]} in the program FIFO
```
hVPUSH(hvalue)
```

3.2.3 Arithmetic & Logic Instructions

add :

```
hADD(destReg, leftReg, rightReg) :
rf[destReg] <= rf[leftReg] + rf[rightReg]
```

immediate add :

```
hVADD(destReg, leftReg, hvalue) :
rf[destReg] <= rf[leftReg] + {{16{hvalue[15]}}, hvalue[15:0]}
```

sub :

```
hSUB(destReg, leftReg, rightReg) :
rf[destReg] <= rf[leftReg] - rf[rightReg]
```

immediate sub :

```
hVSUB(destReg, leftReg, hvalue) :
rf[destReg] <= rf[leftReg] - {{16{hvalue[15]}}, hvalue[15:0]}
```

carry : returns the value of the carry in destination register

```
hADDCR(destReg, leftReg, rightReg) :
rf[destReg] <= {{31{1'b0}}, (rf[leftReg] + rf[rightReg])[32]}
```

immediate carry : returns the value of the carry, provided by immediate add, in destination register

```
hVADDCR(destReg, leftReg, hvalue) :
rf[destReg] <= {{31{1'b0}}, (rf[leftReg] + {{16{value[15]}}, hvalue[15:0]})[32]}
```

borrow : returns the value of the borrow in destination register

```
hSUBCR(destReg, leftReg, rightReg) :
rf[destReg] <= {{31{1'b0}}, (rf[leftReg] - rf[rightReg])[32]}
```

immediate borrow : returns the value of the borrow, provided by immediate subtract, in destination register

```
hVSUBCR(destReg, leftReg, hvalue) :
rf[destReg] <= {{31{1'b0}}, (rf[Reg] - {{16{hvalue[15]}}, hvalue[15:0]})[32]}
```

multiply :

```
hMULT(destReg, leftReg, rightReg) :
rf[destReg] <= rf[leftReg] * rf[rightReg]
```

immediate multiply :

```
hVMULT(destReg, leftReg, hvalue) :
rf[destReg] <= rf[leftReg] * {{16{hvalue[15]}}, hvalue[15:0]}
```

right shift :

```
hRSH(destReg, leftReg) :
rf[destReg] <= rf[leftReg] >> 1
```

arithmetic right shift :

```
hARSH(destReg, leftReg) :
rf[destReg] <= {rf[leftReg][31], rf[leftReg][31:1]}
```

extended right shift :

```
hERSH(destReg, leftReg, rightReg) :
rf[destReg] = {rf[rightReg][0], rf[leftReg][31:1]}
```

extended left shift :

```
hELSH(destReg, leftReg, rightReg) :
rf[destReg] = {rf[leftReg][30:0], rigthReg[31]}
```

bitwise and :

```
hAND(destReg, leftReg, rightReg) :
rf[destReg] <= rf[leftReg] & rf[rightReg]
```

immediate bitwise and :

```
hVAND(destReg, leftReg, hvalue) :
rf[destReg] <= rf[leftReg] & {{16{hvalue[15]}}, hvalue[15:0]}
```

bitwise or :

```
hOR(destReg, leftReg, rightReg) :
rf[destReg] <= rf[leftReg] | rf[rightReg]
```

immediate bitwise or :

```
hVOR(destReg, leftReg, hvalue) :
rf[destReg] <= rf[leftReg] | {{16{hvalue[15]}}, hvalue[15:0]}
```

bitwise xor :

```
hXOR(destReg, leftReg, rightReg) :
rf[destReg] <= rf[leftReg] ^rf[right]
```

immediate bitwise xor :

```
hVXOR(destReg, leftReg, hvalue) :
rf[destReg] <= rf[leftReg] ^{{16{hvalue[15]}}, hvalue[15:0]}
```

bitwise not :

```
NOT(destReg, leftReg) :
rf[destReg] <=  rf[leftReg]
```

3.2.4 Control Instructions

no operation :
```
hNOP : pc <= pc + 1
```

halt :
```
hHALT : pc <= pc
```

relative jump : `label` is used by the assembler program to compute `hvalue`, the signed integer to be added to `pc`
```
hRJMP(label) :  pc <= pc + {{16{hvalue[15]}}, hvalue}
```

branch if zero : `label` is used by the assembler program to compute `hvalue`, the signed integer to be added to `pc` if the condition is fulfilled
```
hBRZ(leftReg, label) :  pc <= (rf[leftReg] = 0) ?  pc + {{16{hvalue[15]}}, hvalue} :  pc + 1
```

branch if not zero : `label` is used by the assembler program to compute `hvalue`, the signed integer to be added to `pc` if the condition is not fulfilled
```
hBRZ(leftReg, label) :  pc <= !(rf[leftReg] = 0) ?  pc + {{16{hvalue[15]}}, hvalue} :  pc + 1
```

immediate jump :
```
hIJMP(hvalue) :  pc = hvalue
```

labeled jump : `label` is used by the assembler program to compute `hvalue`, the signed integer to be added to `pc`
```
hJMP(label) :  pc = hvalue
```

immediate call : unconditioned jump to `hvalue` and save the value of the next `pc` in `rf[destReg]` for `RET` instruction
```
hICALL(destReg, hvalue) :  pc <= pc + hvalue; rf[destReg] <= pc + 1
```

call : unconditioned jump to `hvalue` (computed by the assembler program using the labeling mechanism) and save the value of the next `pc` in `rf[destReg]` for `RET` instruction
```
hCALL(destReg, label) :  pc <= pc + hvalue; rf[destReg] <= pc + 1
```

return from subroutine : the content of `rf[rightreg]` is used to fetch the next instruction
```
hRET(rightReg) :  nextPc = rf[rightreg]
```

wait for accelerator : wait for the accelerator to enter in the halt state
```
hWAITACC : pc <= (accelerator is halt) ?  pc + 1 :  pc
```

wait for interface : wait for HOST DMA to enter in the idle state
```
hWAITINT : pc <= (host dma is idle) ?  pc + 1 :  pc
```

function run : ACCELERATOR receives the command to run the function `label`
```
hRUN(label)
```

parameterized function run : generates five instructions as follows
```
RUN(label, param1, param2, param3, param4):
hRUN(label);
hPUSH(param1);
hPUSH(param2);
hPUSH(param3);
hPUSH(param4);
```

start cycles counter :
```
hSTART
```

stop cycles counter :
```
hSTOP
```

3.3 Accelerator's Assembly Language

3.3.1 Instruction Format for Accelerator

In program memory of CONTROLLER (*prog mem* in Figure 2) are stored pair of 16-bit instructions. One for CONTROLLER and another for ARRAY. The instruction format is:

```
mraInstr[31:0] = {contrInstr[7:0]   , // CONTROLLER's instruction
                  cScalar[7:0]      , // CONTROLLER's immediate value
                  arrayInstr[7:0]   , // ARRAY's instruction
                  aScalar[7:0]      } // ARRAY's immediate value
```

The assembly language provides a sequence of lines each containing an instruction for CONTROLLER (with the prefix `c`) and another for ARRAY. For jumps and branches, some of the line are labeled by `LB(n)`, where *n* is a positive integer.

Example 3.2 *The content of the file containing a program for ACCELERATOR is:*

```
/*****************************************************************************
THE PROGRAM:
    initialize a 16x16 matrix in vector memory starting with vect[4],
    then run the program "matrixVectorMultiply"
*****************************************************************************/
            cPLOAD;         NOP;

            cNOP;           ACTIVATE;
            cVLOAD(16);     VLOAD(3);
            cNOP;           ADDRLD;
            cNOP;           VLOAD(255);
    LB(0);  cVSUB(1);       VADD(1);
            cBRNZ(0);       RISTORE(1);
            cTRUN(7);       NOP;
            cHALT;          NOP;

    `include "05_matrixVectorMultiply.v"

            cPRUN(0);       NOP;
/*****************************************************************************/
```

The left column is the code for CONTROLLER, while the right column is the code for ARRAY.
◇

3.3.2 Transfer Instructions

size send : send to the DMA unit `mem[cScalar][x-1:0]`, as the size of the vector to be transferred
`cLSIZE(cScalar): dma.size <= mem[cScalar][x-1:0]`

address send send to the DMA unit mem[cScalar][28:0], as the address in the external memory where starts the transfer]:
`cLADDR(cScalar): dma.addr <= mem[cScalar[s-1]][28:0]`

run transfer :
`cTRUN(1)`: starts load with the address and size already sent to the DMA unit
`cTRUN(2)`: starts store with the address and size already sent to the DMA unit
`cTRUN(7)`: "accelerator enter in idle state" sent to HOST; `hwaitacc` : the instruction on HOST waiting the accelerator to enter in idle state

starts program load : starts the program load from the address 0 in the program memory of CONTROLLER
`cPLOAD`

run program : from address `cScalar`]:
`cPRUN(cScalar): pc <= cScalar`

pop from progFIFO : loads in CONTROLLER's accumulator the parameter sent by HOST through progFIFO
`cPOPFIFO : acc <= progIn`

3.3.3 Load instructions

The subset of load instructions are used to load, from various storage resources inside the accelerator, *n*-bit words in the accumulator of controller, `acc`, or in the accumulators of each active cell, `acc[i]`.

index load :

```
IXLOAD:  acc[i] <= i
```

immediate load :

```
cVLOAD(cScalar):  acc    <= {(n-c){cScalar[c-1]}}, cScalar}
VLOAD(aScalar):   acc[i] <= {(n-c){aScalar[c-1]}}, aScalar}
```

absolute load :

```
cLOAD(cScalar):   acc    <= mem[cScalar]
LOAD(aScalar):    acc[i] <= vectMem[i][aScalar]
```

relative load :

```
cRLOAD(cScalar):  acc    <= mem[addr + cScalar]
RLOAD(aScalar):   acc[i] <= vectMem[i][addrVect[i] + aScalar]
```

relative load & increment :

```
cRILOAD(cScalar):  acc         <= mem[addr + cScalar
                   addr        <= addr + cScalar
RILOAD(aScalar):   acc[i]      <= vectMem[i][addrVect[i] + aScalar]
                   addrVect[i] <= addrVect[i] + aScalar
```

co-operand load :

```
cCLOAD(0):  acc    <= reductionAdd
cCLOAD(1):  acc    <= reductionMin
cCLOAD(2):  acc    <= reductionMax
cCLOAD(3):  acc    <= reductionFlag
cCLOAD(4):  acc    <= serialReg[0]
cCLOAD(5):  acc    <= serialReg[(1<<x)-1]
CLOAD:      acc[i] <= acc
CALOAD:     acc[i] <= vectMem[i][acc]
CRLOAD:     acc[i] <= vectMem[i][addrVect[i] + acc]
```

serial register load :

```
SRLOAD:  acc[i] <= serialReg[i]
```

input-output register load :

```
IOLOAD:  acc[i] <= ioReg[i]
```

3.3.4 Store instructions

The subset of store instructions are used to store, into various storage resources inside the accelerator, *n*-bit words from the accumulator of controller, `acc`, or from the accumulators of each active cell, `acc[i]`.

absolute store :

```
cSTORE(cScalar):  mem[cScalar]        <= acc
STORE(aScalar):   vectMem[i][aScalar] <= acc[i]
```

relative store :

```
cRSTORE(cScalar): mem[addr + cScalar]               <= acc
RSTORE(aScalar):  vectMem[i][addrVect[i] + aScalar] <= acc[i]
```

relative store & increment :

```
cRISTORE(cScalar): mem[addr + cScalar]               <= acc
                   addr                              <= addr + cScalar
RISTORE(aScalar):  vectMem[i][addrVect[i] + aScalar] <= acc[i]
                   addrVect[i]                       <= addrVect[i] + aScalar
```

co-operand store : the output of the REDUCTION section is considered with a latency of (1+x/2) cycles

```
cCRSTORE(0): mem[addr + reductionAdd]      <= acc (reduction applied to acc[i]
cCRSTORE(1): mem[addr + reductionMin]      <= acc (reduction applied to acc[i]
cCRSTORE(2): mem[addr + reductionMax]      <= acc (reduction applied to acc[i]
cCRSTORE(3): mem[addr + reductionFlag]     <= acc (reduction applied to acc[i]
CSTORE:      vectMem[i][acc]               <= acc[i]
CRSTORE:     vectMem[i][addrVect[i] + acc] <= acc[i]
```

serial register store :

```
SRSTORE:  serialReg[i] <= acc[i]
```

input-output register store :

```
IOSTORE:  ioReg[i] <= acc[i]
```

3.3.5 Address register load instructions

These instructions are used to instantiate the value of the address register in controller, `addr`, and in each cell of the array, `addrVect[i]`. The address register is used to provide differentiations in the address apace of each local data memory distributed in array at the cells level.

address register takes the value from accumulator :

```
cADDRLD:  addr        <= acc
ADDRLD:   addrVect[i] <= acc[i]
```

address registers in array take the value from controller's accumulator :

```
CADDRLD   addrVect[i] <= acc
```

3.3.6 Two-operand n-bit integer instructions

The pattern for the two-operand instruction is presented using the function ADD (addition). Each of the two-operand instruction has the following 12 forms (5 for Controller and 7 for Array) according to the way the second operand is selected. (For the sake of simplicity, in the following, `acc[i]` stands for `accVect[i]` and `cr[i]` stands for `crVect[i]`.)

immediate add :

```
cVADD(cScalar):  {carry, acc}       <= acc + {(n-8){cScalar[7]}}, cScalar}
VADD(aScalar):   {carry[i], acc[i]} <= acc[i] + {(n-8){aScalar[7]}}, aScalar}
```

absolute add :

```
cADD(cScalar):   {carry, acc}       <= acc + mem[cScalar]
ADD(aScalar):    {carry[i], acc[i]} <= acc[i] + vectMem[i][aScalar]
```

relative add :

```
cRADD(cScalar):  {carry, acc}       <= acc + mem[addr + cScalar]
RADD(aScalar):   {carry[i], acc[i]} <= acc[i] + vectMem[i][addrVect[i] + aScalar]
```

relative add & increment :

```
cRIADD(cScalar): {carry, acc}       <= acc + mem[addr + cScalar]
                 addr               <= addr + cScalar
RIADD(aScalar):  {carry[i], acc[i]} <= acc[i] + vectMem[i][addrVect[i] + aScalar]
                 addrVect[i]        <= addrVect[i] + aScalar
```

co-operand add :

```
cCADD(0): {carry, acc} <= acc + reductionAdd(applied to acc[i] (1+x/2) cycles before)
cCADD(1): {carry, acc} <= acc + reductionMin(applied to acc[i] (1+x/2) cycles before)
cCADD(2): {carry, acc} <= acc + reductionMax(applied to acc[i] (1+x/2) cycles before)
cCADD(3): {carry, acc} <= acc + reductionFlag(applied to acc[i] (1+x/2) cycles before)
cCADD(4): {carry, acc} <= acc + serialReg[0]
cCADD(5): {carry, acc} <= acc + serialReg[(1<<x)-1]
CADD:     {carry[i], acc[i]} <= acc[i] + acc
CAADD:    {carry[i], acc[i]} <= acc[i] + vectMem[i][acc]
CRADD:    {carry[i], acc[i]} <= acc[i] + vectMem[i][addrVect[i] + acc]
```

relative to co-operand add[4] :

```
cCRADD(0): {carry, acc} <= acc + reductionAdd(applied to acc[i] (1+x/2) cycles before)
cCRADD(1): {carry, acc} <= acc + reductionMin(applied to acc[i] (1+x/2) cycles before)
cCRADD(2): {carry, acc} <= acc + reductionMax(applied to acc[i] (1+x/2) cycles before)
cCRADD(3): {carry, acc} <= acc + reductionFlag(applied to acc[i] (1+x/2) cycles before)
```

For the following mnemonics, the previously described 12 instructions forms are the same:

```
ADDC    - add with carry: {carry, acc} <= acc + op + carry
SUB     - subtract:       {carry, acc} <= acc - op
RSUB    - reverse SUB:    {carry, acc} <= op - acc
SUBC    - SUB with carry: {carry, acc} <= acc - op - carry
RSUBC   - reverse SUBC:   {carry, acc} <= op - acc - carry
DIV     - division:                acc <= acc / op
```

```
RDIV    - reverse DIV:             acc <= op / acc
MULT    - multiplication:          acc <= acc * op
AND     - bitwise and:             acc <= acc & op
OR      - bitwise or:              acc <= acc | op
XOR     - bitwise xor:             acc <= acc ^ op
COMPARE - compare:        {carry, acc} <= (acc - op)&(10...0)|{0, acc};
```

Thus, instead of the mnemonic `ADD` in one of the previous 12 instruction descriptions, one of the previous 12 can be used. For example: `VADDC`, instead of `VADD`. Thus, 12×13 instructions are already described.

3.3.7 Floating point instructions

The floating point set of instructions use as co-operand only the local memory content addressed by the immediate value from the instruction: `mem[cScalar]` for controller and `vectMem[i][aScalar]` for each array's cell. The execution times for float operations are:

- float add: 3 cycles for the following sequence of instructions (exemplified for controller):

```
cFADD(28);  NOP; // fadd(acc, mem[28])
cMADD;      NOP;
cAPACK;     NOP;
```

- float multiplication: 2 cycles for the following sequence of instructions (exemplified for controller):

```
cMULT(28);  NOP; // fmult(acc, mem[28])
cMPACK;     NOP;
```

- float division: 26 cycles for the following sequence of instructions (exemplified for controller):

```
cFDIV(28);  NOP; // fdiv(acc, mem[28])
cMDIV;      NOP; // executed in 24 cycles
cDPACK;     NOP;
```

first step float addition: unpack & align :

```
cFADD(cScalar): performs fadd(acc, mem[cScalar])
FADD(aScalar):  performs fadd(acc[i], vectMem[i][aScalar])
```

second step float addition: add :

```
cMADD: one-cycle operation on mantissa
MADD:  one-cycle operation on mantissa
```

third step float addition: pack back :

```
cAPACK:  acc    <= fadd(acc, mem[cScalar])
APACK:   acc[i] <= fadd(acc[i], vectMem[i][aScalar])
```

first step float multiplication :

```
cFMULT(cScalar):  performs fmult(acc, mem[cScalar])
FMULT(aScalar):   performs fadd(acc[i], vectMem[i][aScalar])
```

second step float multiplication :

```
cMPACK:  acc    <= fmult(acc, mem[cScalar])
MPACK:   acc[i] <= fmult(acc[i], vectMem[i][aScalar])
```

first step float division :

```
cFDIV(cScalar):  performs fdiv(acc, mem[cScalar])
FDIV(aScalar):   performs fdiv(acc[i], vectMem[i][aScalar])
```

second step float division :

```
cMDIV: 24-cycle operation on mantissa
MDIV:  24-cycle operation on mantissa
```

third step float division :

```
cDPACK:  acc    <= fdiv(acc, mem[cScalar])
DPACK:   acc[i] <= fdiv(acc[i], vectMem[i][aScalar])
```

3.3.8 Shift instructions

shift right one bit position :

```
cSHRIGHT(cScalar):  {cr, acc}       <= {acc[0], acc >> cScalar}
SHRIGHT(aScalar):   {cr[i], acc[i]} <= {acc[i][0], acc[i] >> aScalar}
```

shift right one bit position with carry :

```
cSHRIGHTC:  {cr, acc}       <= {acc[0], cr, acc[n-1:1]}
SHRIGHTC:   {cr[i], acc[i]} <= {acc[i][0], cr[i], acc[i][n-1:1]}
```

shift right arithmetic one bit position :

```
cSHARIGHT:  {cr, acc}       <= {acc[0], acc[n-1], acc[n-1:1]}
SHARIGHT:   {cr[i], acc[i]} <= {acc[i][0], acc[i][n-1], acc[i][n-1:1]}
```

rotate right one bit position :

```
cRROT(cScalar):  acc    <= {acc[cScalar-1:0], acc[n-1:cScalar]}
RROT(aScalar):   acc[i] <= {acc[aScalar-1:0][0], acc[i][n-1:aScalar]}
```

insert `value` **on the least positions** :

```
cINSVAL:  acc    <= {acc[(n-c):0], cScalar}
INSVAL:   acc[i] <= {acc[i][(n-c):0], aScalar}
```

3.3.9 Send controller's operand as co-operand for array

Are executed only by the controller. Are used to send as co-operand for the array the operand of the controller. Must be used in conjunction with an instruction which in array requests the co-operand.

send as co-operand to array `mem[cScalar]` :

```
cSEND(cScalar):     opVect[k] = mem[cScalar]
```

send as co-operand to array `mem[addr + cScalar]` :

```
cRSEND(cScalar):    opVect[k] = mem[addr + cScalar]
```

send as co-operand to array `mem[addr + cScalar]` **and update** `addr` :

```
cRISEND(cScalar):   opVect[k] = mem[addr + cScalar]
                    addr <= addr + cScalar
```

send as co-operand to array the output of the reduction unit selected with `cScalar[1:0]` :

```
cCSEND(0): opVect[k] = reductionAdd
cCSEND(1): opVect[k] = reductionMin
cCSEND(2): opVect[k] = reductionMax
cCSEND(3): opVect[k] = reductionFlg
cCSEND(4): opVect[k] = serialReg[0]
cCSEND(5): opVect[k] = serialReg[(1<<x)-1]
```

This subset of instructions is used in conjunction with the instructions CXXX, where XXX is one of the two-operand instructions previously defined.

Example 3.3 *Let's show how to use this subset of instructions in conjunction with the instruction executed in ARRAY.*

```
/*****************************************************************************
Each accumulator in ARRAY is summed with the content of location 5 from scalar memory of
CONTROLLER
*****************************************************************************/
    cSEND(5);    CADD;    // {cr[i], acc[i]} <= acc[i] + mem[5]
/*****************************************************************************/
```

◇

3.3.10 Sequential control instructions

no operation :

```
cNOP:  acc    <= acc
```

unconditioned jump to the instruction labeled with `LB(cScalar)` :

```
cJMP(cScalar):     pc <= pc + valueComputedByAssembler
```

branch if `acc` **is zero to the instruction labeled with** `LB(cScalar)` :

```
cBRZ(cScalar):     pc <= (acc = 0) ? pc + valueComputedByAssembler : pc + 1
```

branch if `acc` **is not zero to the instruction labeled with** `LB(cScalar)` :

```
cBRNZ(cScalar):    pc <= (acc = 0) ? pc + 1 : pc + valueComputedByAssembler
```

branch if `acc` **is zero to the instruction labeled with** `LB(cScalar)` **& decrement** :

```
cBRZDEC(cScalar):  pc  <= (acc = 0) ? pc + valueComputedByAssembler : pc + 1
                   acc <= acc - 1
```

branch if `acc` **is not zero to the instruction labeled with** `LB(cScalar)` **& decrement** :

```
cBRNZDEC(cScalar): pc  <= (acc = 0) ? pc + 1 : pc + valueComputedByAssembler
                   acc <= acc - 1
```

branch if `acc+1` **is zero to the instruction labeled with** `LB(cScalar)` **& increment** :

```
cBRZINC(cScalar):  pc  <= (acc+1 = 0) ? pc + valueComputedByAssembler : pc + 1
                   acc <= acc + 1
```

branch if `acc+1` **is not zero to the instruction labeled with** `LB(cScalar)` **& increment** :

```
cBRNZINC(cScalar): pc  <= (acc+1 = 0) ? pc + 1 : pc + valueComputedByAssembler
                   acc <= acc + 1
```

branch if `acc` **is negative to the instruction labeled with** `LB(cScalar)` **& increment** :

```
cBRSGN(cScalar):  pc  <= (acc[n-1] = 1) ? pc + valueComputedByAssembler : pc + 1
```

branch if `acc` **is positive to the instruction labeled with** `LB(cScalar)` **& increment** :

```
cBRNSGN(cScalar): pc  <= (acc[n-1] = 1) ? pc + 1 : pc + valueComputedByAssembler
```

skip next instruction if `acc = mem[cScalar]` :

```
cSKIPEQ(cScalar):     pc <= (acc = mem[cScalar]) ? pc + 2 : pc + 1
```

skip next instruction if `acc != mem[cScalar]` :

```
cSKIPNEQ(cScalar):    pc <= (acc = mem[cScalar]) ? pc + 1 : pc + 2
```

wait for IO system to end the current transfer :

```
cIOWAIT: pc <= idleState ? pc : pc + 1;
```

halt :

```
cHALT: pc  <= pc
```

3.3.11 Spatial control instructions

The instructions from this subset are used to select the active cells.
The cell `i` is active if `boolVect[i] = 1`, where: `boolVect[i] = (actVect[i] == 0)`.

no operation :

```
NOP:   acc[i] <= acc[i]
```

activate all cells :

```
ACTIVATE: actVect[i] <= 0
```

keep active where zero :

```
WHEREZERO: actVect[i] <= (boolVect[i]&(condVect[i][0])) ?  actVect[i] : actVect[i]+1
```

keep active where carry :

```
WHERECARRY: actVect[i] <= (boolVect[i]&(condVect[i][1])) ?  actVect[i] : actVect[i]+1
```

keep active where first :

```
WHEREFIRST: actVect[i] <= (boolVect[i]&(condVect[i][2])) ?  actVect[i] : actVect[i]+1
```

keep active where next :

```
WHERENEXT: actVect[i] <= (boolVect[i]&(condVect[i][3])) ?  actVect[i] : actVect[i]+1
```

keep active where not zero :

```
WHERENZERO: actVect[i] <= (boolVect[i]&!(condVect[i][0])) ?  actVect[i] : actVect[i]+1
```

keep active where not carry :

```
WHERENCARRY: actVect[i] <= (boolVect[i]&!(condVect[i][1])) ?  actVect[i] : actVect[i]+1
```

keep active where not first :

```
WHERENFIRST: actVect[i] <= (boolVect[i]&!(condVect[i][2])) ?  actVect[i] : actVect[i]+1
```

keep active where not next :

```
WHERENNEXT: actVect[i] <= (boolVect[i]&!(condVect[i][3])) ?  actVect[i] : actVect[i]+1
```

activate the cells inactivated by the last `where` :

```
ELSEWHERE: actVect[i] <= (actVect[i]==0) ? 1 : ((actVect[i]==1) ? 0 : actVect[i])
```

restore `actVect` **before the corresponding** `where` :

```
ENDWHERE: actVect[i] <= (actVect[i] == 0) ? actVect[i] : (actVect[i] - 1)
```

add to the active cells the cells where `accVect[i] == acc` :

```
ACTWHERE: actVect[i] <= (!boolVect[i] && (accVect[i] == acc)) ? 0 : actVect[i]
```

save the active cells going back to the previous selection pattern :

```
SAVEACT: actVect[i] <= actVect[i] - 1
```

restore the activation pattern saved by the last `saveact` :

```
RESTACT: actVect[i] <= actVect[i] + 1
```

3.3.12 Global shift instructions

global rotate with one position :

```
GROTATE: acc[i] <= acc[(i+1)%(1<<x)]
```

global right shift with one position :

```
GRSHIFT:  accVect[i] <= (i==0) ? 0 : accVect[i-1]
```

global left shift with one position :

```
GLSHIFT: acc[i] <= (i < (1<<x)-1) ? acc[i+1] : 0
```

3.3.13 Global search/insert/delete instructions

search for co-operand in all cells :

```
SRCALL: boolVect[i] <= (acc[i] == acc) ? 1'b1 : 1'b0
```

search for value `aScalar` **in all cells** :

```
VSRCALL(aScalar): boolVect[i] <= (acc[i] == aScalar) ? 1'b1 : 1'b0
```

search for co-operand :

```
SEARCH: boolVect[i] <= (boolVect[i] & (acc[i] == acc)) ? 1'b1 : 1'b0
```

search for value `aScalar` :

```
VSEARCH(aScalar): boolVect[i] <= (boolVect[i] & (acc[i] == aScalar)) ? 1'b1 : 1'b0
```

conditioned search for co-operand :

```
CSEARCH: boolVect[i] <= (i==0) ? 0 : ((acc[i] == acc) & boolVect[i-1]) ? 1 : 0
```

conditioned search for value `aScalar` :

```
VCSEARCH(aScalar): boolVect[i] <= (i==0) ? 0 : ((acc[i]==aScalar)&boolVect[i-1]) ? 1 : 0
```

insert value `aScalar` **in the first active position** :

```
INSERT(aScalar):
acc[i] <= (firstVect[i]) ? aScalar : ((nextVect[i]) ? acc[i-1] : acc[i])
```

insert co-operand in the first active position :

```
CINSERT: acc[i] <= (firstVect[i]) ? acc : ((nextVect[i]) ? acc[i-1] : acc[i])
```

delete the first active accumulator :

```
DELETE: acc[i] <= (firstVect[i] | nextVect[i]) ?
                  ((i == ((1<<x)-1)) ? 0 : acc[i+1]) : acc[i]
```

move selections of active cells one position right :

```
SELSHIFT: boolVect[i] <= (i == 0) ? 1'b0 : boolVect[i-1]
```

3.3.14 Serial register instructions

push right `cScalar` **in serial register** :

```
cVPUSHR(cScalar):
serialReg[i] <= (i < (1<<x)-1) ? serialReg[i+1] : {(n-c){cScalar[c-1]}, cScalar}
```

push right `mem[cScalar]` **in serial register** :

```
cPUSHR(cScalar): serialReg[i] <= (i < (1<<x)-1) ? serialReg[i+1] : mem[cScalar]
```

push right `mem[cScalar[s-1:0] + addr]` **in serial register** :

```
cRPUSHR(cScalar): serialReg[i] <= (i < (1<<x)-1) ? serialReg[i+1] : mem[cScalar + addr]
```

push right `mem[cScalar[s-1:0] + addr]` **in serial register & update** `addr` :

```
cRIPUSHR(cScalar): serialReg[i] <= (i < (1<<x)-1) ? serialReg[i+1] : mem[cScalar + addr]
                   addr <= cScalar + addr
```

push right in the serial register the co-operand selected by `cScalar[1:0]` :

```
cCPUSHR(0): serialReg[i] <= (i < (1<<x)-1) ? serialReg[i+1] : reductionAdd
cCPUSHR(1): serialReg[i] <= (i < (1<<x)-1) ? serialReg[i+1] : reductionMin
cCPUSHR(2): serialReg[i] <= (i < (1<<x)-1) ? serialReg[i+1] : reductionMax
cCPUSHR(3): serialReg[i] <= (i < (1<<x)-1) ? serialReg[i+1] : reductionFlag
```

push left `cScalar` **in serial register** :

```
cVPUSHL(cScalar):
serialReg[i] <= (i == 0) ?  {(n-c){cScalar[c-1]}, cScalar} : serialReg[i-1]
```

push left `mem[cScalar]` **in serial register** :

```
cPUSHL(cScalar): serialReg[i] <= (i == 0) ? mem[cScalar] : serialReg[i-1]
```

push left `mem[cScalar[s-1:0] + addr]` **in serial register** :

```
cRPUSHL(cScalar): serialReg[i] <= (i == 0) ?  mem[cScalar + addr] : serialReg[i-1]
```

push left `mem[cScalar[s-1:0] + addr]` **in serial register & update** `addr` :

```
cRIPUSHL(cScalar): serialReg[i] <= (i == 0) ? mem[cScalar + addr] : serialReg[i-1]
                   addr <= cScalar + addr
```

push left in the serial register the co-operand selected by `cScalar[1:0]` :

```
cCPUSHL(0): serialReg[i] <= (i == 0) ? reductionAdd  : serialReg[i-1]
cCPUSHL(1): serialReg[i] <= (i == 0) ? reductionMin  : serialReg[i-1]
cCPUSHL(2): serialReg[i] <= (i == 0) ? reductionMax  : serialReg[i-1]
cCPUSHL(3): serialReg[i] <= (i == 0) ? reductionFlag : serialReg[i-1]
```

shift left one position the serial register :

```
SRLEFT: serialReg[i] <= (i < (1<<x)-1) ? 0 :  serialReg[i+1]
```

4 How to Use the Assembler

In order to evaluate what can be the maximum performance of our architecture, the assembly language must be used. Therefore, the initial stage of evaluation must be done in assembly language. (The next stage, beyond the scope of our approach, is to provide an efficient compiler from a high level language to the machine language.) We provide few simple example of using the previously described assembly language. The behavioral description of the generic structure is simulated on **Vivado Design Suite 2017.4** provided by Xilinx.

For simulation reasons, the engine is kept small. While the HOST's resources can not be configured, the ACCELERATOR's resources are defined by the content of `00_parameters.v` file as follows:

```
/*****************************************************************************
The list of ARRAY's parameters
*****************************************************************************/
    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
                c = 8   , // value size in instruction
                a = 5     // size of activation counter
/*****************************************************************************/
```

In order to run a simulation two programs must be written: (1) the program running on HOST and (2) the program running on CONTROLLER. The generic forms of these two programs are:

```
/*****************************************************************************
HOST PROGRAM:
    initialize from the external data memory the CONTROLLER's program
    then run "theHostProgram"
    halt
*****************************************************************************/
        hTRANS(0);  // initialize transfer to zero from zero
        hWAITINT;   // wait initialization end
        hWAITACC;   // wait the end of the CONTROLLER's program
                    // theHostProgram
        ...
        hRUN(`NAME);
        ...
                    // end of theHostProgram
        hHALT;
/*****************************************************************************/
```

```
/*****************************************************************************
CONTROLLER PROGRAM:
*****************************************************************************/
            `define NAME 5

            cPLOAD;       NOP;    // starts the program loading process in CONTROLLER
            ...           ...
LB(`NAME);  ...           ...     // starts the program for the function NAME
            ...           ...
            cPRUN(0);     NOP;    // ends the program loading and starts the run from 0
/*****************************************************************************/
```

The two programs are assembled at the beginning of the simulation and are loaded in the HOST's program memory and data memory. The CONTROLLER's program is assembled first, so as the actual address in in the CONTROLLER's programm memory associated to the label `NAME` is available for assembling the HOST's program.

4.1 How to Program the ACCELERATOR

The following examples are presented in order to show how the main features of the MapReduce engine, with the generic architecture, works. The following classes of operations are exemplified:

1. data transfer operations
2. vector operations
3. reduction operations

For the programs running only on ACCELERATOR, the HOST runs only the program which loads the program ACCELERATOR's program memory. Therefore, the HOST's program, `00_hostProgram.v` must be commented to look as follows:

```
/*********************************************************************************
File name: 00_hostProgram.v
HOST PROGRAM:
    initialize from the external data memory the CONTROLLER's program
    halt
*********************************************************************************/
        hTRANS(0);   // initialize transfer to zero from zero
        hWAITINT;    // wait initialization end
        hWAITACC;    // wait the end of the CONTROLLER's program
        hHALT;
/*********************************************************************************/
```

The program for CONTROLLER, `00_theProgram.v`, for the following examples of programs, must be commented to look as follows:

```
/*********************************************************************************
File name:  00_theProgram.v
THE PROGRAM:
*********************************************************************************/
            cPLOAD;          NOP;
            cSTART;          NOP;        // start cycle counter

    `include "04_examples.v"

            cSTOP;           NOP;        // stop cycle counter
            cHALT;           NOP;
            cPRUN(0);        NOP;
/*********************************************************************************/
```

In the file `04_examples.v` the program must be selected by an appropriate comment mechanism.

4.1.1 Data transfer programs

The following data transfer operations are possible in this generic version of the accelerator:

store : vector store, which requests the following parameters:

- size: the number of n-bit scalars of vector
- address: the starting address in the external memory

load : vector load, which requests the following parameters:

- size: the number of n-bit scalars of vector
- address: the starting address in the external memory

Once a parameter of a certain type is received by the DMA unit it is maintained until a new value is sent to the DMA unit for the same type of parameter. For example, once the vector size is established for the first data transfer, we don't need to re-send the size for the next transfers if the size remains the same.

Example 4.1 *Store vector requests two parameters: the size of vector and the starting address in the external memory, because the program "knows" from where, in the vector memory, to take the vector. The parameters of the transfer are loaded in two locations of the controller's memory in order to be used in more complex programs, where eventually they can be submitted to some computations. In or example the size of vector is stored in* `mem[1]` *and the initial address in external memory is stored in* `mem[2]`*. These values are used in the instructions* `cLSIZE` *and* `cLADDR` *pointed by the locations in* `mem` *where they are stored: 1, respectively 2.*

Important note: *the instruction* `cIOWAIT` *cannot follow in the next cycle the instruction* `TRUN`*. At least on cycle delay must be introduced before starting to wait the end of transfer. Any kind of processing instruction can be executed meantime. In our example we inserted a* `cNOP` *instructions.*

```
/*****************************************************************************
TEST PROGRAM FOR: store vector
File name: 04_examples.v (appropriately commented)
The program:
    - loads 10 in {\tt mem[1]} and 14 in {\tt mem[2]}
    - stores the first mem[1] components of the index vector in the external memory
     starting from the address mem[2]
*****************************************************************************/
//*
            cVLOAD(10); NOP;         // size = 10
            cSTORE(1);  NOP;         // save size at mem[1]
            cVLOAD(14); ACTIVATE;    // addr = 14 ; activate all cells
            cSTORE(2);  IXLOAD;      // save addr at mem[2]; load index in all cells
            cLADDR(2);  NOP;         // send addr to DMA
            cLSIZE(1);  IOSTORE;     // send size to DMA; load io register with acc
            cTRUN(2);   NOP;         // start in DMA the store operation
            cNOP;       NOP;         // wait to start before wait to end
            cIOWAIT;    NOP;         // wait the transfer end
//*//***************************************************************************/
```

The program ends with the stream `0, 1, ... 9` *loaded in the external memory starting from the address 14.*
◇

Example 4.2 *Load vector requests two parameters: the size of vector and the starting address in the external memory, because the program "knows" where to load the vector in the vector memory. The parameters of the transfer are loaded in two locations of the controller's memory in order to be used in more complex programs, where eventually they can be submitted to some computations. In or example the size of vector is stored in* `mem[1]` *and the initial address in external memory is stored in* `mem[2]`*. These values are used in the instructions* `cLSIZE` *and* `cLADDR` *pointed by the locations in* `mem` *where they are stored: 1, respectively 2.*

Important note: *the instruction* `cIOWAIT` *cannot follow in the next cycle the instruction* `TRUN`*. At least on cycle delay must be introduced before starting to wait the end of transfer. Any kind of processing instruction can be executed meantime. In our example we inserted a* `cNOP` *instructions.*

```
/*****************************************************************************
TEST PROGRAM FOR: load vector
File name: 04_examples.v (appropriately commented)
*****************************************************************************/
//*
            cVLOAD(10); NOP;         // size = 10
            cSTORE(1);  NOP;         // save size at mem[1]
            cVLOAD(13); ACTIVATE;    // addr = 13 ; activate all cells
            cSTORE(2);  IXLOAD;      // save addr at mem[2]; load index in all cells
            cLADDR(2);  NOP;         // send addr to DMA
            cLSIZE(1);  NOP;         // send size to DMA; load io register with acc
            cTRUN(1);   NOP;         // start in DMA the load operation
            cNOP;       NOP;         // wait to start before wait to end
            cIOWAIT;    IOLOAD;      // wait the transfer end and load in acc[i]
//*//***************************************************************************/
```

The program ends with the transferred vector in accumulator vector.

This program can be tested maintained active in `04_examples.v` *the store program which stores in the data memory of HOST a vector of data.*

◇

4.1.2 Simple Vector & Reduction Programs

Example 4.3 *The program which provide in the controller's accumulator,* `acc`*, the sum of indexes loaded in the accumulators of each cell,* `accVect[i]`*, is:*

```
/*****************************************************************************
File name: 04_examples.v (appropriately commented)
    - activate all cells
    - acc[i] <= i, for i = 0,1,...,15
    - acc <= acc[0]+acc[1]+...+acc[15]
    - only three latency steps are inserted because x = 4 (lambda = 2 + x)
*****************************************************************************/
//*
    cNOP;       ACTIVATE; // activate all cells
    cNOP;       IXLOAD;   // load the index of each cell in accumulator
    cNOP;       NOP;      // latency step
    cNOP;       NOP;      // latency step
    cNOP;       NOP;      // latency step
    cNOP;       NOP;      // latency step
    cNOP;       NOP;      // latency step
    cNOP;       NOP;      // latency step (for x=4)
    //cNOP;     NOP;      // latency step
    //cNOP;     NOP;      // latency step (for x=6)
    //cNOP;     NOP;      // latency step
    //cNOP;     NOP;      // latency step (for x=8)
    cCLOAD(0);  NOP;      // acc <= sum of indexes
//*//***************************************************************************/
```

Appropriately commented means `//*` *instead of* `/*` *before the first line of code.*
The assembled code, provided by the simulator, is:

```
progMem[0]  = 00000000000000001111111100000000
progMem[1]  = 00000000000000001110111100000000
progMem[2]  = 00110111000000000000000000000000
progMem[3]  = 01101111000000000000000000000000
progMem[4]  = 00000000000000000000000000000000
progMem[5]  = 00000000000000000000000000000000
progMem[6]  = 00000000000000000000000000000000
progMem[7]  = 00000000000000000000000000000000
progMem[8]  = 00000000000000000000000000000000
progMem[9]  = 00000000000000000000000000000000
progMem[10] = 00000000000000000110010000000000
progMem[11] = 00000000000000000111111100000000
progMem[12] = 00000000000000000000011100000000
progMem[13] = 00000000000000001111011100000000
```

The result of simulation is:

```
t=0 pc=  x  a=x      a[0]=x a[1]=x a[2]=x ... a[14]=x  a[15]=x  b=xxxxxxxxxxxxxxxx cc=x
t=0 pc=255  a=x      a[0]=x a[1]=x a[2]=x ... a[14]=x  a[15]=x  b=xxxxxxxxxxxxxxxx cc=0
t=1 pc=  0  a=x      a[0]=x a[1]=x a[2]=x ... a[14]=x  a[15]=x  b=xxxxxxxxxxxxxxxx cc=0
t=2 pc=  1  a=x      a[0]=x a[1]=x a[2]=x ... a[14]=x  a[15]=x  b=1111111111111111 cc=0
t=3 pc=  2  a=x      a[0]=0 a[1]=1 a[2]=2 ... a[14]=14 a[15]=15 b=1111111111111111 cc=1
t=4 pc=  3  a=x      a[0]=0 a[1]=1 a[2]=2 ... a[14]=14 a[15]=15 b=1111111111111111 cc=2
t=5 pc=  4  a=x      a[0]=0 a[1]=1 a[2]=2 ... a[14]=14 a[15]=15 b=1111111111111111 cc=3
```

```
t=6 pc=  5  a=x      a[0]=0 a[1]=1 a[2]=2 ... a[14]=14 a[15]=15 b=1111111111111111 cc=4
t=4 pc=  6  a=x      a[0]=0 a[1]=1 a[2]=2 ... a[14]=14 a[15]=15 b=1111111111111111 cc=5
t=5 pc=  7  a=x      a[0]=0 a[1]=1 a[2]=2 ... a[14]=14 a[15]=15 b=1111111111111111 cc=6
t=6 pc=  8  a=x      a[0]=0 a[1]=1 a[2]=2 ... a[14]=14 a[15]=15 b=1111111111111111 cc=7
t=7 pc=  9  a=120    a[0]=0 a[1]=1 a[2]=2 ... a[14]=14 a[15]=15 b=1111111111111111 cc=8
t=8 pc=  10 a=120    a[0]=0 a[1]=1 a[2]=2 ... a[14]=14 a[15]=15 b=1111111111111111 cc=9
t=9 pc=  11 a=120    a[0]=0 a[1]=1 a[2]=2 ... a[14]=14 a[15]=15 b=1111111111111111 cc=9
```

In the initial cycle (`t=0`*) the system reset resets the cycle counter,* `cc = 0`*. In the first cycle (*`t=1`*) the program activate all the cells of the array (the Boolean vector is filled up with 1s). The operation is validated in the next cycle when* `b <= 11...1`*. Then the accumulator in each cell takes the value of* `index`*. During three cycles the reduction network computes the sum of indexes. Then in* `t=7` *the controller's accumulator is loaded with the sum of indexes, i.e., the sum of the numbers* `acc = 0 + 1 + 2 + ... + 15 = 120`*, because we instantiated for our simulation an array with 16 cells. The cycle counter stops in the next cycle on the value 6, which means: the program performed the task in* `cc - 1 = 5` *cycles (the instruction which stops the counter is also counted).*

◇

Example 4.4 *The program which stores at* `mem[24]` *the inner product of the index vector with itself is:*

```
/*****************************************************************************
File name: 04_examples.v (appropriately commented)
    - activate all cells
    - acc[i] <= i for i = 0,1,...,15
    - memVect[i][4] <= acc[i] for i = 0,1,...,15
    - acc[i] <= acc[i] x vectMem[i][4]
    - acc <= acc[0]+acc[1]+...+acc[15]
    - mem[24] <= acc = innerProduct(index, index)
*****************************************************************************/
//*
    cNOP;        ACTIVATE; // activate all cells
    cNOP;        IXLOAD;   // acc[i] <= index
    cNOP;        STORE(4); // memVect[i][4] <= acc[i], for all i
    cNOP;        MULT(4);  // acc[i] <= acc[i] * memVect[i][4]
    cNOP;        NOP;      // latency step
    cNOP;        NOP;      // latency step
    cNOP;        NOP;      // latency step
    cNOP;        NOP;      // latency step
    cNOP;        NOP;      // latency step
    cNOP;        NOP;      // latency step x=4
    //cNOP;      NOP;      // latency step
    //cNOP;      NOP;      // latency step x=6
    //cNOP;      NOP;      // latency step
    //cNOP;      NOP;      // latency step x=8
    cCLOAD(0);   NOP;      // acc <= reductionAdd(acc[i])
    cSTORE(24); NOP;       // mem[24] <= acc
//*//============================================================================
```

The simulation provides, a little edited to fit in page, the following results:

```
t=1   pc= 0 a=x     a[0]=x a[1]=x a[2]=x ... a[14]=x   a[15]=x   b=xx...x cc=0
t=2   pc= 1 a=x     a[0]=x a[1]=x a[2]=x ... a[14]=x   a[15]=x   b=11...1 cc=0
t=3   pc= 2 a=x     a[0]=0 a[1]=1 a[2]=2 ... a[14]=14  a[15]=15  b=11...1 cc=1
t=4   pc= 3 a=x     a[0]=0 a[1]=1 a[2]=2 ... a[14]=14  a[15]=15  b=11...1 cc=2
t=5   pc= 4 a=x     a[0]=0 a[1]=1 a[2]=4 ... a[14]=196 a[15]=225 b=11...1 cc=3
t=6   pc= 5 a=x     a[0]=0 a[1]=1 a[2]=4 ... a[14]=196 a[15]=225 b=11...1 cc=4
t=7   pc= 6 a=x     a[0]=0 a[1]=1 a[2]=4 ... a[14]=196 a[15]=225 b=11...1 cc=5
t=8   pc= 7 a=x     a[0]=0 a[1]=1 a[2]=4 ... a[14]=196 a[15]=225 b=11...1 cc=6
t=9   pc= 5 a=x     a[0]=0 a[1]=1 a[2]=4 ... a[14]=196 a[15]=225 b=11...1 cc=7
t=10  pc= 6 a=x     a[0]=0 a[1]=1 a[2]=4 ... a[14]=196 a[15]=225 b=11...1 cc=8
t=11  pc= 7 a=x     a[0]=0 a[1]=1 a[2]=4 ... a[14]=196 a[15]=225 b=11...1 cc=9
t=12  pc= 8 a=1240  a[0]=0 a[1]=1 a[2]=4 ... a[14]=196 a[15]=225 b=11...1 cc=10
t=13  pc= 9 a=1240  a[0]=0 a[1]=1 a[2]=4 ... a[14]=196 a[15]=225 b=11...1 cc=11
t=14  pc=10 a=1240  a[0]=0 a[1]=1 a[2]=4 ... a[14]=196 a[15]=225 b=11...1 cc=12

...
mem[23]       = x
mem[24]       = 1240
mem[25]       = x
...
```

In cycle 9 *the controllers accumulator is loaded with the value of the inner product and in the next cycle its content is stored in the local scalar memory.*

◇

Example 4.5 *The program which provide in* `acc` *the number of components of the index vector bigger than 5 and smaller than 15 is:*

```
/*****************************************************************************
File name: 04_examples.v (appropriately commented)
    - activate all cells
    - acc[i} <= i
    - keep active cells where (acc[i] >= 5)
    - keep active cells where (acc[i] < 15)
    - acc[i] <= 1 only in all active cells
    - acc <= acc[0]+acc[1]+...+acc[15] only for the active cells
*****************************************************************************/
//*
    cNOP;       ACTIVATE;       // activate all cells
    cNOP;       IXLOAD;         // acc[i] <= index
    cNOP;       VSUB(5);        // {cr, acc[i]} <= acc[i] - 5
    cNOP;       WHERENCARRY;    // where cr=1 remain active
    cNOP;       VSUB(10);       // {{cr, acc[i]} <= acc[i] - (15 - 5)
    cNOP;       WHERECARRY;     // where cr=0 remain active
    cNOP;       VLOAD(1);
    cNOP;       ENDWHERE;       // reactivate where the second WHERE acted
    cNOP;       ENDWHERE;       // reactivate where the first WHERE acted
    cNOP;       NOP;            // latency step
    cNOP;       NOP;            // latency step
    cNOP;       NOP;            // latency step
    cNOP;       NOP;            // latency step x=4
    //cNOP;     NOP;            // latency step
    //cNOP;     NOP;            // latency step x=6
    //cNOP;     NOP;            // latency step
    //cNOP;     NOP;            // latency step x=8
    cCLOAD(0);  NOP;            // acc <= number of active cells
//*/
//============================================================================
```

The simulation provides:

```
t=1   pc= 0 a=x   a[0]=x          ... a[6]=x          a[7]=x   b=xxxxxxxxxxxxxxxx
t=2   pc= 1 a=x   a[0]=x          ... a[6]=x          a[7]=x   b=1111111111111111
t=3   pc= 2 a=x   a[0]=0          ... a[6]=14         a[7]=15  b=1111111111111111
t=4   pc= 3 a=x   a[0]=4294967291 ... a[14]=9         a[15]=10 b=1111111111111111
t=5   pc= 4 a=x   a[0]=4294967291 ... a[14]=9         a[15]=10 b=0000011111111111
t=6   pc= 5 a=x   a[0]=4294967291 ... a[14]=4294967295 a[15]=0  b=0000011111111111
t=7   pc= 6 a=x   a[0]=4294967291 ... a[14]=4294967295 a[15]=0  b=0000011111111110
t=8   pc= 7 a=x   a[0]=4294967291 ... a[14]=1         a[15]=0  b=0000011111111110
t=9   pc= 8 a=x   a[0]=4294967291 ... a[14]=1         a[15]=0  b=0000011111111111
t=10  pc= 9 a=x   a[0]=4294967291 ... a[14]=1         a[15]=0  b=1111111111111111
t=11  pc=10 a=x   a[0]=4294967291 ... a[14]=1         a[15]=0  b=1111111111111111
t=12  pc=11 a=x   a[0]=4294967291 ... a[14]=1         a[15]=0  b=1111111111111111
t=13  pc=12 a=x   a[0]=4294967291 ... a[14]=1         a[15]=0  b=1111111111111111
t=14  pc=13 a=x   a[0]=4294967291 ... a[14]=1         a[15]=0  b=1111111111111111
t=15  pc=14 a=10 a[0]=4294967291 ... a[14]=1         a[15]=0  b=1111111111111111
```

Indeed, the index vector contains 10 components bigger then 5 and smaller than 15.
◇

Example 4.6 *Load index in cell's accumulators and do $l = 9$ times: divide by 2 (integer operation) and increment with 99 each accumulator. The program is:*

```
/*****************************************************************************
File name: 04_examples.v (appropriately commented)
    - activate all cells
    - acc <= 8; initialize the loop counter with l-1
    - acc[i} <= i; load index
    - do (acc+1) times
          acc[i] <= acc[i]/2
          acc[i] <= acc[i] + 99
*****************************************************************************/
//*
            cNOP;           ACTIVATE;
            cVLOAD(8);      IXLOAD;
    LB(1);  cNOP;           SHRIGHT;
            cBRNZDEC(1);    VADD(99); // branch if acc=0 and acc<=acc-1
//*///=========================================================================
```

The simulation provides:

```
t=1  pc=0 a=x          a[0]=x   a[1]=x   a[2]=x   ... a[14]=x   a[15]=x
t=3  pc=2 a=8          a[0]=0   a[1]=1   a[2]=2   ... a[14]=14  a[15]=15
t=4  pc=3 a=8          a[0]=0   a[1]=0   a[2]=1   ... a[14]=7   a[15]=7
t=5  pc=2 a=7          a[0]=99  a[1]=99  a[2]=100 ... a[14]=106 a[15]=106
t=6  pc=3 a=7          a[0]=49  a[1]=49  a[2]=50  ... a[14]=53  a[15]=53
t=7  pc=2 a=6          a[0]=148 a[1]=148 a[2]=149 ... a[14]=152 a[15]=152
t=8  pc=3 a=6          a[0]=74  a[1]=74  a[2]=74  ... a[14]=76  a[15]=76
t=9  pc=2 a=5          a[0]=173 a[1]=173 a[2]=173 ... a[14]=175 a[15]=175
t=10 pc=3 a=5          a[0]=86  a[1]=86  a[2]=86  ... a[14]=87  a[15]=87
t=11 pc=2 a=4          a[0]=185 a[1]=185 a[2]=185 ... a[14]=186 a[15]=186
t=12 pc=3 a=4          a[0]=92  a[1]=92  a[2]=92  ... a[14]=93  a[15]=93
t=13 pc=2 a=3          a[0]=191 a[1]=191 a[2]=191 ... a[14]=192 a[15]=192
t=14 pc=3 a=3          a[0]=95  a[1]=95  a[2]=95  ... a[14]=96  a[15]=96
t=15 pc=2 a=2          a[0]=194 a[1]=194 a[2]=194 ... a[14]=195 a[15]=195
t=16 pc=3 a=2          a[0]=97  a[1]=97  a[2]=97  ... a[14]=97  a[15]=97
t=17 pc=2 a=1          a[0]=196 a[1]=196 a[2]=196 ... a[14]=196 a[15]=196
t=18 pc=3 a=1          a[0]=98  a[1]=98  a[2]=98  ... a[14]=98  a[15]=98
t=19 pc=2 a=0          a[0]=197 a[1]=197 a[2]=197 ... a[14]=197 a[15]=197
t=20 pc=3 a=0          a[0]=98  a[1]=98  a[2]=98  ... a[14]=98  a[15]=98
t=21 pc=4 a=4294967295 a[0]=197 a[1]=197 a[2]=197 ... a[14]=197 a[15]=197
```

The initial value of `accVect` *is* {`0, 1, ..., 14, 15`}. *After 8 execution of the two cycles loop it becomes:* {`197, 197, ..., 197, 197`}.

◇

Example 4.7 *Add in* `accVect` *index with the sum of all indexes. The program is:*

```
/*****************************************************************************
File name: 04_examples.v (appropriately commented)
    - activate all cells
    - acc[i} <= i; load index
    - acc <= acc[0]+acc[1]+...acc[15]
    - acc[i] <= acc[i] + acc
*****************************************************************************/
/*
        cNOP;         ACTIVATE; // ccEnable <= 1; boolVect <= 11...1
        cNOP;         IXLOAD;   // acc[i] <= i
        cNOP;         NOP;      // latency step 1
        cNOP;         NOP;      // latency step 2
        cNOP;         NOP;      // latency step 3
        cNOP;         NOP;      // latency step 4
        cNOP;         NOP;      // latency step 5
        cNOP;         NOP;      // latency step 6
        cCLOAD(0);    NOP;      // acc <= acc[0] + acc[1] + ... acc[15]
        cNOP;         CADD;     // acc[i] <= acc[i] + acc
//*///============================================================================
```

The result of simulation is:

```
t=1  pc=0  a=x   a[0]=x   a[1]=x   a[2]=x   ... a[14]=x   a[15]=x   cc=0
t=2  pc=1  a=x   a[0]=x   a[1]=x   a[2]=x   ... a[14]=x   a[15]=x   cc=0
t=3  pc=2  a=x   a[0]=0   a[1]=1   a[2]=2   ... a[14]=14  a[15]=15  cc=1
t=4  pc=3  a=x   a[0]=0   a[1]=1   a[2]=2   ... a[14]=14  a[15]=15  cc=2
t=5  pc=4  a=x   a[0]=0   a[1]=1   a[2]=2   ... a[14]=14  a[15]=15  cc=3
t=6  pc=5  a=x   a[0]=0   a[1]=1   a[2]=2   ... a[14]=14  a[15]=15  cc=4
t=7  pc=6  a=x   a[0]=0   a[1]=1   a[2]=2   ... a[14]=14  a[15]=15  cc=5
t=8  pc=7  a=x   a[0]=0   a[1]=1   a[2]=2   ... a[14]=14  a[15]=15  cc=6
t=9  pc=8  a=x   a[0]=0   a[1]=1   a[2]=2   ... a[14]=14  a[15]=15  cc=7
t=10 pc=9  a=120 a[0]=0   a[1]=1   a[2]=2   ... a[14]=14  a[15]=15  cc=8
t=11 pc=10 a=120 a[0]=120 a[1]=121 a[2]=122 ... a[14]=134 a[15]=135 cc=9
t=12 pc=11 a=120 a[0]=120 a[1]=121 a[2]=122 ... a[14]=134 a[15]=135 cc=10
```

In 6 cycles is performed a computation consisting of 29 additions. In the general case, for a number of p cells, the execution time is $3+(1+0.5\times log\,p)=4+0.5\times log\,p$, *where* $1+0.5\times log\,p$ *is the latency of the reduction net.*

Therefore, $2p-1$ *are performed in* $4+0.5\times log\,p$ *cycles by an engine with p cells. The acceleration belongs to*

$$O(\frac{p}{log\,p})$$

which is normal for a computation involving communication.

◇

5 How to Build a Library of Functions

In this section is presented an example of how to build a library of function on ACCELERATOR. The functions are accessed and used by HOST. The functions are defined on data stored in the data memory of HOST and the results are sent back to the same memory.

Therefore, the HOST's program, `00_hostProgram.v` must be commented to look as follows:

```
/*****************************************************************************
File name: 00_hostProgram.v
HOST PROGRAM:
    initialize from the external data memory the CONTROLLER's program
    halt
*****************************************************************************/
                                    // LOAD THE PROGRAM
        hTRANS(0);                  // initialize transfer to zero from zero
        hWAITINT;                   // wait initialization end
        hWAITACC;
                                    // INITIALIZE DATA IN THE EXTERNAL MEMORY
        RUN(`MSTORE, 4, 16, 9, 9);
        RUN(`MSTORE, 8, 0, 9, 1);

        `include "04_libraryHostTest.v"

        hHALT;
/*****************************************************************************/
```

The program for CONTROLLER, `00_theProgram.v`, for the following examples of programs, must be commented to look as follows:

```
/*****************************************************************************
File name:  00_theProgram.v
THE PROGRAM:
    initialize a 16x16 matrix in vector memory starting with vect[4],
    then run the selected program
*****************************************************************************/
*****************************************************************************/
            cPLOAD;         NOP;
            cSTART;         NOP;        // start cycle counter
            cNOP;           ACTIVATE;
            cVLOAD(16);     VLOAD(3);
            cNOP;           ADDRLD;
            cNOP;           VLOAD(255);
    LB(0);  cVSUB(1);       VADD(1);
            cBRNZ(0);       RISTORE(1);
            cTRUN(7);       NOP;
            cHALT;          NOP;

    `include "04_libraryTest.v"

            cSTOP;          NOP;        // stop cycle counter
            cHALT;          NOP;
            cPRUN(0);       NOP;
/*****************************************************************************/
```

The program included in `00_hostProgram.v` is:

```
/*****************************************************************************
File name: 04_libraryHostTest.v
        RUN(loadMatrix, to, from, columns, lines)
        RUN(loadVector, to, from, columns, lines)
        RUN(matrixVectorMultiply, matrix, lines, vector, destination)
        RUN(storeVector, from, to, columns, lines)
*****************************************************************************/
    hSTART;                             // start cycles counter
    RUN(`MLOAD, 21, 16, 9, 9);          // load matrix
    RUN(`MLOAD, 33, 0, 9, 1);           // load vector
    RUN(`MVMULT, 21, 9, 33, 34);        // multiply
    RUN(`MSTORE, 34, 118, 9, 1);        // store vector
    hRUN(`EOP);                         // request 'end of program' flag
    hWAITACC;                           // wait for 'end of program'
    hSTOP;                              // stop cycles counter
    hHALT;                              // halt host
/*****************************************************************************/
```

The program included in `00_theProgram.v` is:

```
/*****************************************************************************
File name: 04_libraryTest.v
    Test library:
    MSTORE:(storeMatrix)(from, to, columns, lines)
    MLOAD:(loadMatrix)(to, from, columns, lines)
    MVMULT:(matrixVectorMultiply)(matrix, lines, vector, destination)
    EOP: end of program
*****************************************************************************/
                                                    // FUNCTION LABELS
                    `define MSTORE  1               // matrix store
                    `define MLOAD   2               // matrix load
                    `define MVMULT  3               // matrix-vector multiply
                    `define EOP     9               // end of program

                                                    // JUMP/BRANCH LABELS
                    `define MS1     4
                    `define MS2     5
                    `define ML      6
                    `define MVM1    7
                    `define MVM2    8
                    `define EOP     9
                                                    // MATRIX STORE
    LB(`MSTORE);    cPOPFIFO;       NOP;            // receive the source address in array
                    cNOP;           CADDRLD;        // addr[i] <= first line address
                    cPOPFIFO;       RILOAD(0);      // receive the destination address
                    cSTORE(2);      NOP;            // save address at mem[2]
                    cPOPFIFO;       NOP;            // receive the size of vector
                    cSTORE(1);      NOP;            // save size at mem[1]
                    cPOPFIFO;       IOSTORE;        // receive number of lines
                    cSTORE(3);      NOP;
                                                    // main loop
    LB(`MS1);       cLADDR(2);      NOP;            // send addr to DMA
                    cLSIZE(1);      NOP;            // send size to DMA
                    cTRUN(2);       NOP;            // start the send operation in DMA
                    cLOAD(3);       NOP;
                    cVSUB(1);       NOP;
                    cBRZ(`MS2);     NOP;
```

```
                    cSTORE(3);      NOP;
                    cLOAD(2);       RILOAD(1);
                    cADD(1);        NOP;
                    cSTORE(2);      NOP;
                    cIOWAIT;        NOP;
                    cJMP('MS1);     IOSTORE;
    LB('MS2);       cIOWAIT;        NOP;
                    cHALT;          NOP;
                                            // MATRIX LOAD
    LB('MLOAD);     cPOPFIFO;       NOP;    // receive the destination address
                    cNOP;           CLOAD;  //
                    cPOPFIFO;       VSUB(1); // receive the source address
                    cSTORE(2);      ADDRLD;
                    cPOPFIFO;       NOP;    // receive the size of vector
                    cSTORE(1);      NOP;    // save size at mem[1]
                    cPOPFIFO;       NOP;    // receive number of lines
                    cSTORE(3);      NOP;    // save lines in mem[3]
                                            // main loop
    LB('ML);        cLADDR(2);      NOP;    // send addr to DMA
                    cLSIZE(1);      NOP;    // send size to DMA
                    cTRUN(1);       NOP;    // start the load operation in DMA
                    cLOAD(2);       NOP;
                    cADD(1);        NOP;
                    cSTORE(2);      NOP;
                    cIOWAIT;        NOP;    // wait the transfer end
                    cLOAD(3);       NOP;
                    cVSUB(1);       IOLOAD;
                    cSTORE(3);      RISTORE(1);
                    cBRNZ('ML);     NOP;
                    cHALT;          NOP;
                                            // MATRIX-VECTOR MULTIPLY
    LB('MVMULT);    cPOPFIFO;       NOP;    // receive the matrix address in array
                    cNOP;           CLOAD;  //
                    cPOPFIFO;       NOP;    // receive the number of lines
                    cSTORE(3);      CADD;   //
                    cNOP;           ADDRLD; // save number of lines at mem[3]
                    cNOP;           IXLOAD; //
                    cNOP;           CSUB;
                    cPOPFIFO;       WHERECARRY; // receive vector address
                    cNOP;           CALOAD; //
                    cPOPFIFO;       STORE(0); // mem[0][i] <= vector
                    cSTORE(2);      NOP;    // receive the result address
                    cLOAD(3);       NOP;
                                            // Multiplication
                    cNOP;           RILOAD(255);//
                    cVSUB(1);       MULT(0);
    LB('MVM1);      cCPUSHL(0);     RILOAD(255);
                    cBRNZDEC('MVM1);MULT(0);
                    cVLOAD(x/2-2);  NOP;    //
    LB('MVM2);      cBRNZDEC('MVM2);NOP;    // latency loop
                    cLOAD(2);       SRLOAD; // load result in acc[i]
                    cNOP;           CSTORE;
                    cHALT;          NOP;
                                            // END OF PROGRAM
    LB('EOP);       cTRUN(7);       NOP;
                    cHALT;          NOP;
/*****************************************************************************/
```

References

[1] Krste Asanovic, *et al.*, *The landscape of parallel computing research: A view from Berkeley*, 2006.
See at: `www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf`

[2] John Backus, Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM* 21, 8 (August) 1978. 613-641.

[3] Călin Bîră, R. Hobincu, Lucian Petrică, "OPINCAA: A Light-Weight and Flexible Programming Environment For Parallel SIMD Accelerators" *Romanian Journal of Information Science and Technology*, Volume 16, Numbers 4, 2013, 336-350

[4] Stephen Kleene, General recursive functions of natural numbers. *Mathematische Annalen* 112, 5, 1936. 727-742.

[5] V. Kumar, A. Grama, A. Gupta, G. Karypis, *Introduction to Parallel Computing. Design and Analysis of Algorithms*, The Benjamin/Cummings Pub. Comp., Inc., 1994.

[6] Mihaela Maliţa, Gheorghe M. Ştefan, Dominique Thiébaut, Not Multi-, but Many-Core: Designing Integral Parallel Architectures for Embedded Computation. *ACM SIGARCH Computer Architecture News*, Vol. 35, No. 5, December 2007. 32-39.

[7] Mihaela Maliţa, and Gheorghe M. Ştefan, Backus language for functional nano-devices. *CAS 2011, vol. 2*, 331-334.

[8] Gheorghe M. Ştefan, *et al.*, The CA1024: A fully programmable system-on-chip for cost-effective HDTV media processing. *Hot Chips: A Symposium on High Performance Chips*. Memorial Auditorium, Stanford University.

[9] Gheorghe M. Ştefan, One-chip TeraArchitecture. *Proceedings of the 8th Applications and Principles of Information Science Conference*. Okinawa, Japan, 2009.
See at: `www.dropbox.com/s/5oqncu71t7zf8es/teraArchitecture.pdf?dl=`

[10] Gheorghe M. Ştefan, Integral parallel architecture in system-on-chip designs. *The 6th International Workshop on Unique Chips and Systems*, Atlanta, GA, USA, December 4, 2010, pp. 23-26.

[11] Gheorghe M. Ştefan, Mihaela Maliţa, Can One-Chip Parallel Computing Be Liberated From Ad Hoc Solutions? A Computation Model Based Approach and Its Implementation, *18th Inter. Conf. on Ciruits, Systems, Communications and Computers*, Santorini, July 17-21, 2014, 582-597.
See at: `www.dropbox.com/s/rtzzs1d06526jzj/COMPUTERS2-42.pdf?dl=0`

[12] Gheorghe Stefan, *Loops & Complexity in DIGITAL SYSTEMS.* Lecture Notes on Digital Design in Giga-Gate/Chip Era, (work in endless progress) 2016 version.
See at: `www.dropbox.com/s/neooi2cca5y8lxa/0-BOOK.pdf?dl=0`