Programming in Assembly Language the *pRISC*-based Heterogeneous System

pRISC team

(Version 1.0)

This document was prepared with LaTeX 2_ε

Preamble

Instead of using as accelerator Intel's multi-core CPU, or the Nvidia's oximoronic General Purpose Graphic Processing Unit, GPGPU, or Google's Tensor Processing Unit, TPU, or Intel's Video processing Unit, VPU, or Nvidia's Data Processing Unit, DPU, let's see how to use pRISC Heterogeneous System to obtain and use a library of accelerated functions.

The system is described through a top view of the structure and a detailed view of the architectural resources. The second chapter describes the host and accelerator assembly language, and the third chapter is dedicated to accelerating computation through the use of libraries of functions defined and optimized in assembly language.

Contents

Chapter 1

Heterogeneous System

A heterogeneous computing system consists of a host computer and an accelerator. The host has an assembly language that contains, in addition to the established instructions of a RISC processor, a library of kernel functions that call on the hardware structure of the accelerator for efficient calculation of computationally intensive functions.

1.1 Heterogeneous Structure

The heterogeneity of computing systems became necessary when the distinction between complex and intensive computing was imposed. Intensive computing involves optimizing execution time, energy consumption and silicon area (price). For this reason, the segregation between the two computing resources (see Figure 1.1) of a heterogeneous system appeared:

- the host processor, as part of the HOST COMPUTER, for complex computing
- the ACCELERATOR, for intense computing, which works as a hardware accelerated library of functions.

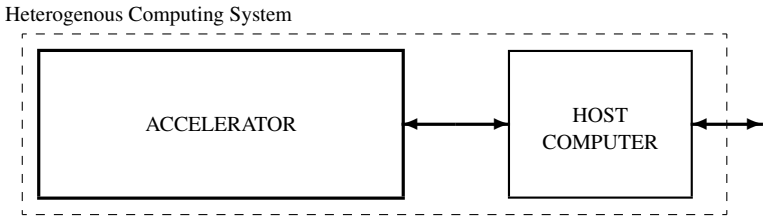

Figure 1.1: Heterogenous computing system.

The organization of the ACCELERATOR – a pRISC[1] parallel computing system – part of the heterogeneous computing system is represented in Figure 1.2, where:

[1] pRISC comes form *parallel RISC*, term coined by Jim Peek, the leader of the master students team who designed the first RISC chip under the supervision of David Patterson and Carlo Sequin (`https://www.computerhistory.org/revolution/digital-logic/12/286/1593`).

MAP : is a linear array of p cells each containing an execution unit and a local data memory (register file) of m scalars

DISTRIBUTE : is a log-depth pipelined tree distribution network use to distribute in each clock cycle a command from CONTROLLER to the cells in MAP

CONTROLLER : is a mono-core processing element with its program and data memory and IO system used to communicate with the HOST processor; it issues in each clock cycle a command to be executed by each active cells in MAP

SCAN/REDUCE : is a log-depth circuit performing scan and reduction functions.

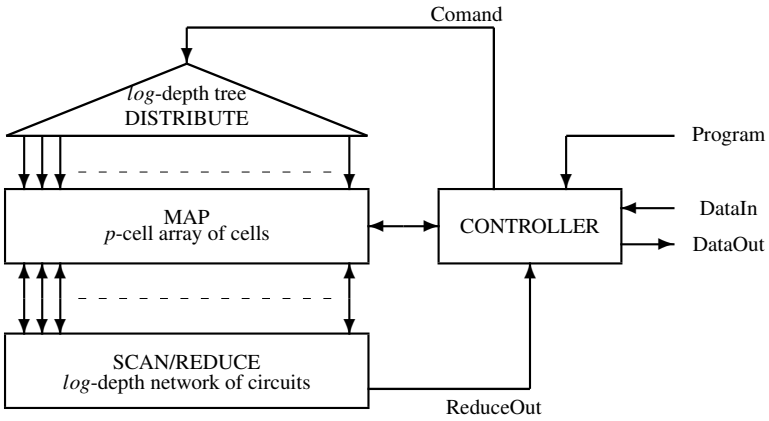

Figure 1.2: pRISC accelerator.

pRISC accelerator is connected to the HOST system on 3 channels:

- Program: used to transfer from the HOST's processor
  - the programs in assembly languages to be loaded in the program memory of CONTROLLER, usually at the initialization of the system
  - the sequence of operations (each having associated in the program memory of CONTROLLER a program in assembler) to be performed by ACCELERATOR under the control of HOST
  - the sequence of data transfer orders to manage the flow on the other two channels
- DataIn: is the way used to load data from the HOST's memory
- DataOut: is the way used to store data in the HOST's memory

1.2 Heterogeneous Architecture

The instructions executed in the heterogeneous system are grouped into three sets:

hISA : host's processor instruction set architecture, with two subsets:

mainISA : a standard set of RISC-type instructions

libraryISA : a library of specific functions executed by the accelerator

cISA : accelerator's controller instruction set architecture, with a standard set of RISC-type instructions

aISA : array's instruction set architecture oriented on vector operations.

Programming the heterogeneous system means to design and run two programs: one on HOST and another on ACCELERATOR.

1.2.1 Host Program

The file `0_hProgram` is part of the project installed on Vivado environment. It instals in accelerator the program(s) to be executed and starts running them.

```
/*************************************************************************
File name: 0_hProgram.sv
                    HOST PROGRAM
*************************************************************************/
            hVALUE(0,0); // rf[0] <= 0
            hPSEND(0,0); // send program from hDmem[0] to contrPM[0] and
                         // and run it from 0 in accelerator

    // here comes the code to be executed

            hHALT;
```

1.2.2 Accelerator Program

The file `0_aProgram` is part of the project installed on Vivado environment. It is installed when the system is initialized. The HOST's program does this by the first two instructions it run.

```
/*************************************************************************
File name: 0_aProgram.sv
                    ACCELERATOR PROGRAM
*************************************************************************/
            cPLOAD(0); ACTIVATE; // load program from 0; all cell active
            cNOP;      GETIO(1); // discard the I/O register of array
            cNOP;      IXLOAD;   // ACC <= [0, 1, ..., (p-1)]

    // here comes the code to be executed;
    // for example: `include "00_theKernel.sv"

    LB(32); cHALT;     NOP;
            cPRUN(0);  NOP;      // run from 0
```

The assembly program for accelerator has two columns: the first for the code executed by the comtroller, and the second for the code issued by the controller to be executed in the active p cells of the MAP array.

Chapter 2

Assembly Level

The architectural parameters are defined in the file `0_DEFINES.vh` whose first lines are as follows:

```
/*************************************************************
File name: 0_DEFINES.vh
*************************************************************/
`define n (32)       // internal word
`define p (16)       // number of cells in MAP
`define m (1024)     // memory size

...
```

2.1 Host Assembler

2.1.1 Host's ISA

The memory resources of the HOST computer:

```
reg [31:0]  hPmem[0:`m-1], HOST's program memory
reg [63:0]  hDmem[0:`m-1], HOST's data memory
```

For the simulation reasons we limit, for simple testes, the size of the memories to $m = 1024$.

The processor's variable are:

```
reg [`n-1:0]            rf[0:31] : register file
reg [$clog2(`m)-1:0]    pc       : program counter
reg                     cr       : carry bit
```

The two instruction formats are:

```
instr={ opCode[5:0] ,   // operation code
        dest[4:0]   ,   // destination address in register file
        left[4:0]   ,   // left operand address in register file
        right[4:0]  ,   // right operand address in register file
        nu[10:0]    }   // not used
        |
      { opCode[5:0] ,   // operation code
```

```
dest[4:0]   ,   // destination address in register file
left[4:0]   ,   // left operand address in register file
value[15:0] }   // signed integer value as right operand
```

Each assembly-level instruction is made by concatenating the operation micro-code (indicated on the first column of the following tables) with the micro-code that selects the operation mode (indicated on the first line in the following table). In use (see Table 2.1), `value[15:0]` is expanded to its signed 32-bit form, as follows:

```
val[31:0] = {{16{value[15]}}, value[15:0]}
```

Table 2.1: Host's Instruction Set Architecture.

OpCode	Commentaries	Assembly
CONTROL OPERATIONS		
nop	no operation: `pc <= pc+1`	hNOP
hjmp	relative jump: `pc <= pc+val`	hJMP(lb)
hjmpz	conditioned jump: `pc <= (rf[left]==0) ? pc+val : pc+1`	hJMPZ(l, lb)
hjmpnz	conditioned jump: `pc <= (rf[left]==0) ? pc+1 : pc+val`	hJMPNZ(l, lb)
hajmp	absolute jump: `pc <= pc + val`	hAJMP(val)
hhalt	halt: `pc <= pc`	hHALT
hpsend	send to Map a auto-delimited program	hPSEND(d, l)
hdget	get from Map a stream of data	hDGET(d, l, r)
hdsend	send to Map a stream of data	hDSEND(d, l, r)
hintwait	interrupt wait	hINTWAIT
hsttcc	start host's cycle counter	hSTARTCC
hstpcc	stop host's cycle counter	hSTOPCC
FUNCTIONAL OPERATIONS		
hadd	`{cr, rf[dest]} <= rf[left] + rf[right]`	hADD(d, l, r)
haddcr	`{cr, rf[dest]} <= rf[left] + rf[right] + cr`	hADDCR(d, l, r)
hsub	`{cr, rf[dest]} <= rf[left] - rf[right]`	hSUB(d, l, r)
hsubcr	`{cr, rf[dest]} <= rf[left] - rf[right] - cr`	hSUBCR(d, l, r)
hmult	`{cr, rf[dest]} <= {cr, rf[left] * rf[right]}`	hMULT(d, l, r)
hbwand	`{cr, rf[dest]} <= {cr, rf[left] & rf[right]}`	hAND(d, l, r)
hbwor	`{cr, rf[dest]} <= {cr, rf[left] | rf[right]}`	hOR(d, l, r)
hbwxor	`{cr, rf[dest]} <= {cr, rf[left]` $\wedge$ `rf[right]}`	hXOR(d, l, r)
haddv	`{cr, rf[dest]} <= rf[left] + val`	hADDV(d, l, val)
haddcrv	`{cr, rf[dest]} <= rf[left] + val + cr`	hADDCRV(d, l, val)
hsubv	`{cr, rf[dest]} <= rf[left] - val`	hSUBV(d, l, val)
hsubcrv	`{cr, rf[dest]} <= rf[left] - val - cr`	hSUBCRV(d, l, val)
hmultv	`{cr, rf[dest]} <= {cr, rf[left] * val}`	hMULTV(d, l, val)
hbwandv	`{cr, rf[dest]} <= {cr, rf[left] & val}`	hANDV(d, l, val)
hbworv	`{cr, rf[dest]} <= {cr, rf[left] | val}`	hORV(d, l, val)
hbwxorv	`{cr, rf[dest]} <= {cr, rf[left]` $\wedge$ `val}`	hXORV(d, l, val)
DATA TRANSFER INSTRUCTIONS		
hvalue	`{cr, rf[dest]} <= {cr, {16{val[15]}}, val}`	hVALUE(d, val)
hinsval	`{cr, rf[dest]} <= {cr, {rf[left][15:0], val}}`	hINSVAL(d, val)
hload	`{cr, rf[dest]} <= {cr, hDmem[rf[left]]}`	hLOAD(d, l)
hstore	`hDmem[rf[left]] <= rf[right]`	hSTORE(l, r)
hssend	`accelerator <= {desr,left,value} = scalar`	hSSEND(val)
hfsend	`accelerator <= value = function`	hFSEND(val)

2.1.2 HOST's Library of Function

The library of functions are defined using two mainISA functions:

- `hfsend`: to specify the function using a code sent to accelerator
- `hssend`: to send the associated values for a function if needed

In file `cgHOST_LIBRARY.sv` the way the functions are generated is presented.

Table 2.2: Host's Function Library Architecture (FLA) defined in file: `00_theKernel.sv`.

Assembly	**Commentaries**
hSTART	Start controller's cycle counter
hSTOP	Stop controller's cycle counter
hINTRQ	Interrupt request
hVGENX(addr)	`V(addr) <= IX`
hVGENN(addr,val)	`V(addr) <= [val ... val]`
hSQGENX(addr)	Generate square matrix `[V[addr]=IX ... V[addr+p-1]=IX+p-1]`
hSQGENN(addr)	Generate square matrix with `V[addr+i]= [i i ... i]`, for $i = 0,...,p-1$
hMAIN(addr,val)	Generate diagonal matrix starting with `V[addr]` with value `val`
hPRANDOM(addr)	Generate pseudo-random square matrix starting with `V[addr ]`
hMSEND(addr,size)	Send square matrix of `size` starting from `V[addr]`
hMGET(addr,size)	Get square matrix of `size` and load from `V[addr]`
hSQMADD(d,l,r)	Add square matrices: `M[d] <= M[l] + M[r]`
hSQMVMULT(d,l,r)	Multiply square matrices with vector: `V[d] <= M[l] * V[r]`
hSQMMULT(d,l,r)	Square matrices multiply: `M[d] <= M[l] * M[r]`
hSQMMAC(d,l,r)	Square matrices multiply and add: `M[d] <= M[d] + (M[l] * M[r])`
hTRANS(d,l)	Transpose square matrix: `M[d] <= trans(M[l])`

In Table 2.2, `M[v]` represents a $p \times p$ square matrix with the first vector `V[a]` and the last vector `V[a+p-1]`.

2.2 Accelerator Assembler

While the register file of the host processor is a two output port memory, in accelerator is used a register file with only one output port. The processing is accumulator based with a big sized register file.

The memory resources of the accelerator are:

```
reg [63:0]   contrPM[0:‘m-1] // controller’s program memory
reg [31:0]   contrDM[0:‘m-1] // controller’s data memory (register file)
reg [‘n-1:0] arrayDM[0:‘m-1] // each cell’s data memory (register file)
```

The variable seen by the instruction set are:

```
FOR CONTROLLER:
    reg [$clog2(‘m)-1:0] pc         // program counter
    reg [31:0]           acc        // controller’s accumuulator
    reg                  cr         // controller’s carry
    reg[$clog2(‘m)-1:0]  addrReg    // controller’s address register
    reg[31:0]            mem[0:m-1] // controller’s data memory
```

```
    val = {{8{value[23]}},value}    // used as operand
    addr = value[$clog2(‘m)-1:0]    // used as address

    coOp = redOut                   // co-operand: reduction network output
FOR MAP ARRAY IN EACH CELL:
    reg [31:0]          acc[0:p-1]      // array's accumuulator
    reg                 cr[0:p-1]       // array's carry
    reg[$clog2(m)-1:0]  addrReg[0:p-1]  // array's address register
    reg[n-1:0]          mem[0:m]        // array's data memory
    reg[$clog2(p)-1]    act[0:p-1]      // activate counter
    reg[n-1:0]          sr[0:p-1]       // serial register

    val = {{8{value[23]}},value}        // used as operand
    addr = value[$clog2(‘m)-1:0]        // used as address

    b[i] = (act == 0) ? 1 : 0

    coOp = acc                          // co-operand: controller's accumulator
```

2.2.1 Controller Assembly

Instruction Formats

The two instruction formats are:

```
    instr = {cOpcodeBinary[4:0] , // operation code for binary operations
             cMode[2:0]         , // second operand selection mode
             value[23:0]        } // signed integer value
             |
            {8'b11111_000       ,
             cOpcodeUnary[4:0]  , // operation code for unary operations
             value[18:0]        } // integer to parameterize operations
```

Binary Operations

Because the architecture is accumulator-based, the left operand is the accumulator, `acc`, and the second operand, for the binary operations (BOP), is established according to the following selection modes:

```
    imm : val
        acc <= acc BOP val
    dir : mem[addr]
        acc <= acc BOP mem[addr]
    rel : mem[addrReg+addr]
        acc <= acc BOP mem[addrReg+addr]
    rei : mem[addrReg+addr]; addrReg = addrReg+addr
        acc <= acc BOP mem[addrReg+addr];
    cim : coOp = redOut
        acc <= acc BOP redOut
```

Each binary assembly-level instruction is made by concatenating the 5-bit operation micro-code (indicated on the first column of Table 2.3 and described in the file `0_DEFINES.sv`) with the 3-bit micro-code that selects the operation mode (indicated on the first line in the same table).

Table 2.3: Instructions with operands from the controller's Instruction Set Architecture.

cOpcodeBinary\cMode	**imm**	**dir**	**rel**	**rei**	**cim**
cadd	cVADD(s)	cADD(s)	cRADD(s)	cRIADD(s)	CADD
caddc	cVADDC(s)	cADDC(s)	cRADDC(s)	cRIADDC(s)	CADDC
csub	cVSUB(s)	cSUB(s)	cRSUB(s)	cRISUB(s)	CSUB
crsub	cVRSUB(s)	cRSUB(s)	cRRSUB(s)	cRIRSUB(s)	CRSUB
csubc	cVSUBC(s)	cSUBC(s)	cRSUBC(s)	cRISUBC(s)	CSUBC
crsubc	cVRSUBC(s)	cRSUBC(s)	cRRSUBC(s)	cRIRSUBC(s)	CRSUBC
cmult	cVMULT(s)	cMULT(s)	cRMULT(s)	cRIMULT(s)	CMULT
cbwand	cVAND(s)	cAND(s)	cRAND(s)	cRIAND(s)	CAND
cbwor	cVOR(s)	cOR(s)	cROR(s)	cRIOR(s)	COR
cbwxor	cVXOR(s)	cXOR(s)	cRXOR(s)	cRIXOR(s)	CXOR
cload	cVLOAD(s)	cLOAD(s)	cRLOAD(s)	cRILOAD(s)	CLOAD
cstore		cSTORE(s)	cRSTORE(s)	cRISTORE(s)	CCSTORE

For example, the addition can be performed in 5 different modes, as follows:

- `cVADD(35)` means `acc <= acc + 35`
- `cADD(12)` means `acc <= acc + mem[12]`
- `cRADD(12)` means `acc <= acc + mem[addr + 12]`
- `cRIADD(12)` means `acc <= acc + mem[addr + 12]; addr <= addr + 12`
- `cCADD(12)` means `acc <= acc + redOut`

Unary Operations

The unary operations are defined for `instr[31:24] = 8'b11111_000 = {'contr, 'imm}`, by `instr[23:0]` (see **Instruction Formats** in 2.2.1).

Table 2.4: Sequencer's Instruction Set Architecture with no operand.

{cOpcodeUnary,value[2:0]}	**Commentaries**	**Assembly**
{cshift,4}	Shift right one bit position	cSHR
{cshift,5}	Shift right arithmetic one bit position	cASHR
{cshift,6}	Shift right one bit position with carry	cSHRC
{cshift,1}	Shift left one bit position	cSHL
{cshift,2}	Shift left one bit position with carry	cSHLC
{cshift,7}	Rotate right one bit position	cROTR
{cshift,3}	Rotate left one bit position	cROTL
{gshift,0}	1-position global right shift	cGRSHIFT
{gshift,1}	1-position global left shift	cGLSHIFT
{gshift,5}	Reduction output insert right	cRREDINS
{gshift,4}	Reduction output insert left	cLREDINS
{gshift,2}	1-position global right rotate	cGRROTATE

{cOpcodeUnary,value[2:0]}	Commentaries	Assembly
{gshift,3}	1-position global left rotate	cGLROTATE
nop	No operation	cNOP
jmp	Relative jump to label *s*	cJMP(s)
ajmp	Absolute jump to label *s*	cAJMP(s)
brz	Jump to label *s* if `acc=0`	cBRZ(s)
brnz	Jump to label *s* if `acc!=0`	cBRNZ(s)
brzdec	Jump to label *s* if `acc=0`; `acc<=acc-1`	cBRZDEC(s)
brnzdec	Jump to label *s* if `acc!=0`; `acc<=acc-1`	cBRNZDEC(s)
halt	`pc <= pc`	cHALT
cinsval	`acc <= {(acc << 8), value[7:0]}`	cINSVAL
crela	`addr <= acc`	cADDRLD
start	Start cycle counter	cSTART
stop	Stop cycle counter	cSTOP

2.2.2 Array Assembly

Instruction Formats

The two instruction formats are:

```
instr = {opcodeBinary[4:0]  , // operation code for binary operations
         mode[2:0]          , // second operand selection mode
         value[23:0]        } // signed integer value
         |
        {8'b11111_000       ,
         opcodeUnary[4:0]   , // operation code for unary operations
         value[18:0]        } // integer to parameterize operations
```

Because the architecture is accumulator-based, the left operand in each cell is the accumulator, `acc[i]`, and the second operand, for the binary operations (BOP), is established according to the following selection modes:

```
imm : val
    acc[i] <= acc[i] BOP val
dir : mem[i][addr]
    acc[i] <= acc[i] BOP mem[i][addr]
rel : mem[i][addrReg[i]+addr]
    acc[i] <= acc[i] BOP mem[i][addrReg[i]+addr]
rei : mem[i][addrReg[i]+addr]; addrReg[i] = addrReg[i]+addr
    acc[i] <= acc[i] BOP mem[i][addrReg[i]+addr]
cim : coOp = acc
    acc[i] <= acc[i] BOP acc
cdr : mem[i][coOp]
    acc[i] <= acc[i] BOP  mem[i][acc]
crl : mem[i][addrReg[i]+coOp]
    acc[i] <= acc[i] BOP  mem[i][addrReg[i]+acc]
cri : mem[i][addrReg[i]+coOp]; addrReg[i] = addrReg[i]+acc
    acc[i] <= acc[i] BOP  mem[i][addrReg[i]+acc]
```

Binary Operations

Each binary assembly-level instruction is made by concatenating the 5-bit operation micro-code (indicated on the first column of Table 2.5 and described in the file `0_DEFINES.sv`) with the 3-bit micro-code that selects the operation mode (indicated on the first line in the same table).

Table 2.5: Instructions with operands from the Instruction Set Architecture.

	imm	**dir**	**rel**	**rei**	**cim**	**cdr**	**crl**	**cri**
add	VADD(s)	ADD(s)	RADD(s)	RIADD(s)	CADD	CAADD	CRADD	CRIADD
addc	VADDC(s)	ADDC(s)	RADDC(s)	RIADDC(s)	CADDC	CAADDC	CRADDC	CRIADDC
sub	VSUB(s)	SUB(s)	RSUB(s)	RISUB(s)	CSUB	CASUB	CRSUB	CRISUB
rsub	VRSUB(s)	RSUB(s)	RRSUB(s)	RIRSUB(s)	CRSUB	CARSUB	CRRSUB	CIRRSUB
subc	VSUBC(s)	SUBC(s)	RSUBC(s)	RISUBC(s)	CSUBC	CASUBC	CRSUBC	CRISUBC
rsubc	VRSUBC(s)	RSUBC(s)	RRSUBC(s)	RIRSUBC(s)	CRSUBC	CARSUBC	CRRSUBC	CRIRSUBC
mult	VMULT(s)	MULT(s)	RMULT(s)	RIMULT(s)	CMULT	CAMULT	CRMULT	CRIMULT
bwand	VAND(s)	AND(s)	RAND(s)	RIAND(s)	CAND	CAAND	CRAND	CRIAND
bwor	VOR(s)	OR(s)	ROR(s)	RIOR(s)	COR	CAOR	CROR	CRIOR
bwxor	VXOR(s)	XOR(s)	RXOR(s)	RIXOR(s)	CXOR	CAXOR	CRXOR	CRIXOR
load	VLOAD(s)	LOAD(s)	RLOAD(s)	RILOAD(s)	CLOAD	CALOAD	CRLOAD	CRILOAD
store		STORE(s)	RSTORE(s)	RISTORE(s)		CSTORE	CRSTORE	CRISTORE
getio		GETIO(s)		RIGETIO(s)				
sendio		SENDIO(s)		RISENDIO(s)				
search	VSEARCH(s)				SEARCH			
csearch	VCSEARCH(s)				CSEARCH			
insert	VINSERT(s)				INSERT			

For example, the multiplication can be performed in 8 different modes, as follows:

- `VMULT(35)` means `acc[i] <= acc[i] * 35`, for `i = 0,1,...,p-1`
- `MULT(12)` means `acc[i] <= acc[i] * mem[i][12]`, for `i = 0,1,...,p-1`
- `RMULT(12)` means `acc[i] <= acc[i] * mem[i][addr[i] + 12]`, for `i = 0,1,...,p-1`
- `RIMULT(12)` means `acc[i] <= acc[i] * mem[i][addr[i] + 12]`; `addr[i] <= addr[i] + 12`, for `i = 0,1,...,p-1`
- `CMULT` means `acc[i] <= acc[i] * acc`, for `i = 0,1,...,p-1`
- `CAMULT` means `acc[i] <= acc[i] * mem[i][acc]`, for `i = 0,1,...,p-1`
- `CRMULT` means `acc[i] <= acc[i] * mem[i][addr[i] + acc]`, for `i = 0,1,...,p-1`
- `CRIMULT` means `acc[i] <= acc[i] * mem[i][addr[i] + acc]`; `addr[i] <= addr[i] + acc`, for `i = 0,1,...,p-1`

Unary Operations

The unary operations are defined for `instr[31:24] = 8'b11111_000` = {`contr, `imm}, by `instr[23:0]` (see **Instruction Formats** in 2.2.2).

Table 2.6: Map's Instruction Set Architecture with no operand.

{cOpcode, value[2:0]}	**Commentaries**	**Assembly**
{shift,4}	Shift right one bit position	SHR
{shift,5}	Shift right arithmetic one bit position	ASHR

{cOpcode, value[2:0]}	Commentaries	Assembly
{shift,6}	Shift right one bit position with carry	SHRC
{shift,1}	Shift left one bit position	SHL
{shift,2}	Shift left one bit position with carry	SHLC
{shift,7}	Rotate right one bit position	ROTR
{shift,3}	Rotate left one bit position	ROTL
ixload	`acc[i] <= i`	IXLOAD
getsr	`acc[i] <= serialReg[i]`	GETSR
sendsr	`serialReg[i] <= acc[i]`	SENDSR
{where,0}	`b[i] <= (b[i] & acc[i]=0) ? 1 : 0`	WHEREZERO
{where,1}	`b[i] <= (b[i] & carry[i]) ? 1 : 0`	WHERECARRY
{where,2}	`b[i] <= (b[i] & acc[i][31]=1) ? 1 : 0`	WHERENEGATIVE
{where,3}	`b[i] <= i=0 ? 0 : (b[i-1] ? 1 : 0)`	WHEREPREVACT
{where,4}	`b[i] <= (b[i] & first ? 1 : 0)`	WHEREFIRST
{where,5}	`b[i] <= (b[i] & next ? 1 : 0)`	WHERENEXT
{where,6}	`b[i] <= (b[i] & !(first | next)) ? 1 : 0`	WHEREPREV
{where,8}	`b[i] <= (b[i] & !(acc[i]=0)) ? 1 : 0`	WHERENZERO
{where,9}	`b[i] <= (b[i] & !carry[i]) ? 1 : 0`	WHERENCARRY
{where,10}	`b[i] <= (b[i] & !(acc[i][31]=1)) ? 1 : 0`	WHERENNEGATIVE
{where,11}	`b[i] <= i=0 ? 1 : (b[i-1] ? 0 : 1)`	WHERENPREVACT
{where,12}	`b[i] <= (b[i] & !first) ? 1 : 0`	WHERENFIRST
{where,13}	`b[i] <= (b[i] & !next) ? 1 : 0`	WHERENNEXT
{where,14}	`b[i] <= (b[i] & (first | next)) ? 1 : 0`	WHERENPREV
elsew	Reapply the last `where` with the negated condition	ELSEWHERE
back	Redo `b[i]` affected by the most recently active where	ENDWHERE
selsh	`b[i] <= i=0 ? 0 : b[i-1]`	SELSHIFT
allact	`b[i] <= 1`	ACTIVATE
{setred,0}	Set the reduction function on addition	REDADD
{setred,3}	Set the reduction function on minimum	REDMIN
{setred,2}	Set the reduction function on maximum	REDMAX
{setred,1}	Set the reduction function on logical OR	REDOR
delete		DELETE
cinsval	`acc[i] <= {(acc[i] << 8), value[7:0]}`	INSVAL
crela	`addr[i] <= acc[i]`	ADDRLD

2.3 Examples

2.3.1 The HOST as an array program launcher

Example 2.1 *If we only want to run a single program (for testing), then we will use the host only to load and run the program written for the accelerator. The file* `0_hProgram.sv` *will have the following minimal form:*

```
/*****************************************************************
File name: 0_hProgram.sv
                    HOST PROGRAM
*****************************************************************/
            hVALUE(0,0); // rf[0] <= 0
            hPSEND(0,0); // send program from 0 and run it from 0
// no host program
            hHALT;
```

The file `0_aProgram.sv` *will contain the program we want to run, with the prefix and suffix previously described. Let's exemplify it with the program that loads into the vector V[2] the index vector to which the constant 1 is added.*

```
/*****************************************************************
File name: 0_aProgram.sv
Description: load in V[2] the index vector incremented with 5
*****************************************************************/
            cPLOAD(0);   ACTIVATE; // load program from 0; all cell active
            cNOP;        GETIO(1); // discard the I/O register of array
            cNOP;        IXLOAD;   // ACC <= [0 1 ... (p-1)]
// the program
            cVLOAD(77); VADD(5);
            cSTORE(3);   STORE(2);
// end program
     LB(32); cHALT;      NOP;
            cPRUN(0);    NOP;      // run from 0
```

The simulation program, `0_simulation.sv`*, first use the previous two programs running* `0_hostCodeGenerator.sv` *and* `0_accCodeGenerator.sv` *generating the following code for the host in host's program memory:*

```
hPmem[0] = 10100000000000000000000000000000
hPmem[1] = 10000000000000000000000000000000
hPmem[2] = 00100100000000000000000000000000
hPmem[3] = xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

and the following code for the accelerator in host's data memory:

```
hDmem[0]     = 11111000000000000000000000000001111100001100000000000000000000
hDmem[1]     = 01100001000000000000000000000001000000000000000000000000000000
hDmem[2]     = 11111000010010000000000000000000000000000000000000000000000000
hDmem[3]     = 00000000000000000000000000001010100000000000000000000001001101
hDmem[4]     = 01001001000000000000000000000010010010010000000000000000000011
hDmem[5]     = 00000000000000000000000000000011111100000111000000000000000000
hDmem[6]     = 00000000000000000000000000000011111100001101000000000000000000
hDmem[7]     = xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

The simulator displays the two programs, the initializes the simulation generating the reset signal for 4 time units. The host sends the program from its data memory in the program memory of the controller using the instruction `hPSEND(0,0)`*. Because the program ends with the instruction* `cPRUN(0); NOP;` *its execution starts from the address 0 in controller's program memory.*

The result provided by the simulation's monitor running on the Vivado environment is:

```
t=0  pc=x cAcc=x  cMEM=[x, x, x, x, x, x, x, x]  ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]          B = [xxxxxxxxxxxxxxxx]
t=1  pc=0 cAcc=x  cMEM=[x, x, x, x, x, x, x, x]  ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]          B = [xxxxxxxxxxxxxxxx]
t=27 pc=1 cAcc=x  cMEM=[x, x, x, x, x, x, x, x]  ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]          B = [xxxxxxxxxxxxxxxx]
t=29 pc=2 cAcc=x  cMEM=[x, x, x, x, x, x, x, x]  ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]          B = [xxxxxxxxxxxxxxxx]
t=31 pc=3 cAcc=x  cMEM=[x, x, x, x, x, x, x, x]  ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]          B = [xxxxxxxxxxxxxxxx]
t=33 pc=4 cAcc=x  cMEM=[x, x, x, x, x, x, x, x]  ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]          B = [xxxxxxxxxxxxxxxx]
t=35 pc=5 cAcc=x  cMEM=[x, x, x, x, x, x, x, x]  ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]          B = [xxxxxxxxxxxxxxxx]
t=37 pc=5 cAcc=77 cMEM=[x, x, x, x, x, x, x, x]  ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]          B = [xxxxxxxxxxxxxxxx]
t=39 pc=5 cAcc=77 cMEM=[x, x, x, 77, x, x, x, x] ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]          B = [1111111111111111]
t=41 pc=0 cAcc=77 cMEM=[x, x, x, 77, x, x, x, x] ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]          B = [1111111111111111]
t=43 pc=0 cAcc=77 cMEM=[x, x, x, 77, x, x, x, x] ACC=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]     B = [1111111111111111]
t=45 pc=0 cAcc=77 cMEM=[x, x, x, 77, x, x, x, x] ACC=[5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20] B = [1111111111111111]
```

while the display provides, at the end of simulation:

```
vect[0] = [x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]
vect[1] = [x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]
vect[2] = [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
vect[3] = [x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]
```

Note*: in this example and in the followings, the results provided by the monitor and the display of the simulator program,* `0_simulator.sv` *are edited to illustrate only the significant behavior for the example run.*

◇

2.3.2 Vector operations

Example 2.2 *To raise the index vector to the fourth power and store it in* `V[1]`*, we write the following program:*

```
/*********************************************************************
File name: 0_aProgram.sv
Description: V[1] <= IX^4
*********************************************************************/
            cPLOAD(0);      ACTIVATE;
            cNOP;           GETIO(1);
            cNOP;           IXLOAD;
// the program
            cVLOAD(2);      STORE(0);
            cNOP;           NOP;        // because the lack of forwarding
    LB(13); cBRNZDEC(13);   MULT(0);
            cNOP;           STORE(1);
// end program
    LB(32); cHALT;          NOP;
            cPRUN(0);       NOP;
```

The result is:

```
t=0  cAcc=x  ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]
t=1  cAcc=x  ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]
t=11 cAcc=x  ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]
t=29 cAcc=x  ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]
t=41 cAcc=1  ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]
t=43 cAcc=1  ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]
t=45 cAcc=0  ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]
t=47 cAcc=-1 ACC=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
t=49 cAcc=-1 ACC=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
t=51 cAcc=-1 ACC=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
t=53 cAcc=-1 ACC=[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225]
t=55 cAcc=-1 ACC=[0, 1, 8, 27, 64, 125, 216, 343, 512, 729, 1000, 1331, 1728, 2197, 2744, 3375]
t=57 cAcc=-1 ACC=[0, 1, 16, 81, 256, 625, 1296, 2401, 4096, 6561, 10000, 14641, 20736, 28561, 38416, 50625]
```

while the display provides, at the end of simulation:

```
vect[0] = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
vect[1] = [0, 1, 16, 81, 256, 625, 1296, 2401, 4096, 6561, 10000, 14641, 20736, 28561, 38416, 50625]
vect[2] = [x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]
```

◇

2.3.3 Predicated operations

Example 2.3 *The effect of the predicated operation is illustrated by the following program which multiplies by 2 the even indexes and with 3 the odd indexes.*

```
/*************************************************************************
File name: 0_aProgram.sv
Description: acc[i] <= (i = 2j) ? 2i : 3i, for j = 0,1,...,p/2-1
*************************************************************************/
              cPLOAD(0);        ACTIVATE;
              cNOP;             GETIO(1);
              cNOP;             IXLOAD;
// the program
              cNOP;             VAND(1);
              cNOP;             WHEREZERO;
              cNOP;             IXLOAD;
              cNOP;             VMULT(2);
              cNOP;             ELSEWHERE;
              cNOP;             IXLOAD;
              cNOP;             VMULT(3);
              cNOP;             ENDWHERE;
              cNOP;             STORE(1);
// end program
     LB(32); cHALT;             NOP;
              cPRUN(0);         NOP;
```

The result is:

```
t=0  pc=x  ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]          B = [xxxxxxxxxxxxxxxx]
t=1  pc=0  ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]          B = [xxxxxxxxxxxxxxxx]
t=41 pc=1  ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]          B = [xxxxxxxxxxxxxxxx]
t=43 pc=2  ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]          B = [xxxxxxxxxxxxxxxx]
t=45 pc=3  ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]          B = [xxxxxxxxxxxxxxxx]
t=47 pc=4  ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]          B = [xxxxxxxxxxxxxxxx]
t=49 pc=5  ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]          B = [xxxxxxxxxxxxxxxx]
t=51 pc=6  ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]          B = [xxxxxxxxxxxxxxxx]
t=53 pc=7  ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]          B = [1111111111111111]
t=55 pc=8  ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]          B = [1111111111111111]
t=57 pc=9  ACC=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]    B = [1111111111111111]
t=59 pc=10 ACC=[0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1]          B = [1111111111111111]
t=61 pc=11 ACC=[0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1]          B = [1010101010101010]
t=63 pc=12 ACC=[0, 1, 2, 1, 4, 1, 6, 1, 8, 1, 10, 1, 12, 1, 14, 1]       B = [1010101010101010]
t=65 pc=12 ACC=[0, 1, 4, 1, 8, 1, 12, 1, 16, 1, 20, 1, 24, 1, 28, 1]     B = [1010101010101010]
t=67 pc=12 ACC=[0, 1, 4, 1, 8, 1, 12, 1, 16, 1, 20, 1, 24, 1, 28, 1]     B = [0101010101010101]
t=69 pc=0  ACC=[0, 1, 4, 3, 8, 5, 12, 7, 16, 9, 20, 11, 24, 13, 28, 15]  B = [0101010101010101]
t=71 pc=0  ACC=[0, 3, 4, 9, 8, 15, 12, 21, 16, 27, 20, 33, 24, 39, 28, 45] B = [0101010101010101]
t=73 pc=0  ACC=[0, 3, 4, 9, 8, 15, 12, 21, 16, 27, 20, 33, 24, 39, 28, 45] B = [1111111111111111]
```

while the display provides, at the end of simulation:

```
vect[0] = [x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]
vect[1] = [0, 3, 4, 9, 8, 15, 12, 21, 16, 27, 20, 33, 24, 39, 28, 45]
vect[2] = [x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]
```

◇

2.3.4 Reductions operations

Example 2.4 *The effect of the reduction network is illustrated by the following program which, starting from the index vector incremented with 1, loads the sum of the components, Boolean OR, the minimum and the maximum values into the controller accumulator one by one.*

```
/*************************************************************************
File name: 0_aProgram.sv
Description: ACC <= redADD(IX+1); redOR((IX+1); redMIN(IX+1);redMAX(IX+1)
*************************************************************************/
            cPLOAD(0);      ACTIVATE;
            cNOP;           GETIO(1);
            cNOP;           IXLOAD;
// the program
            cNOP;           VADD(1);
            cNOP;           REDADD;
            cNOP;           REDOR;
            cNOP;           REDMIN;
            cNOP;           REDMAX;
            cNOP;           NOP;
            cNOP;           NOP;
            cCLOAD;         NOP;
            cCLOAD;         NOP;
            cCLOAD;         NOP;
            cCLOAD;         NOP;
            cCLOAD;         NOP;
            cCLOAD;         NOP;
            cCLOAD;         NOP;
// end program
    LB(32); cHALT;          NOP;
            cPRUN(0);       NOP;
```

The result is:

```
t=1  pc=0  cAcc=x   ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]        B = [xxxxxxxxxxxxxxxx]
t=51 pc=1  cAcc=x   ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]        B = [xxxxxxxxxxxxxxxx]
t=53 pc=2  cAcc=x   ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]        B = [xxxxxxxxxxxxxxxx]
t=55 pc=3  cAcc=x   ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]        B = [xxxxxxxxxxxxxxxx]
t=57 pc=4  cAcc=x   ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]        B = [xxxxxxxxxxxxxxxx]
t=59 pc=5  cAcc=x   ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]        B = [xxxxxxxxxxxxxxxx]
t=61 pc=6  cAcc=x   ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]        B = [xxxxxxxxxxxxxxxx]
t=63 pc=7  cAcc=x   ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]        B = [1111111111111111]
t=65 pc=8  cAcc=x   ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]        B = [1111111111111111]
t=67 pc=9  cAcc=x   ACC=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]  B = [1111111111111111]
t=69 pc=10 cAcc=x   ACC=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16] B = [1111111111111111]
t=71 pc=11 cAcc=x   ACC=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16] B = [1111111111111111]
t=73 pc=12 cAcc=x   ACC=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16] B = [1111111111111111]
t=75 pc=13 cAcc=x   ACC=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16] B = [1111111111111111]
t=77 pc=14 cAcc=x   ACC=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16] B = [1111111111111111]
t=79 pc=15 cAcc=x   ACC=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16] B = [1111111111111111]
t=81 pc=16 cAcc=136 ACC=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16] B = [1111111111111111]
t=83 pc=17 cAcc=31  ACC=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16] B = [1111111111111111]
t=85 pc=17 cAcc=1   ACC=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16] B = [1111111111111111]
t=87 pc=17 cAcc=16  ACC=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16] B = [1111111111111111]
t=89 pc=0  cAcc=16  ACC=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16] B = [1111111111111111]
```

◇

Example 2.5 *Example 2.4 is repeated in the following program which uses a wait loop to take into account the latencies introduced by the distribution network and by the reduction network.*

```
/*************************************************************************
File name: 0_aProgram.sv
Description: ACC <= redADD(IX+1); redOR((IX+1); redMIN(IX+1);redMAX(IX+1)
*************************************************************************/
            cPLOAD(0);          ACTIVATE;
            cNOP;               GETIO(1);
            cNOP;               IXLOAD;
// the program
            cNOP;               VADD(1);
            cNOP;               REDADD;
            cNOP;               REDOR;
            cNOP;               REDMIN;
            cVLOAD($clog2(`p)); REDMAX;
    LB(5);  cBRNZDEC(5);        NOP;        // latency loop
            cCLOAD;             NOP;
            cCLOAD;             NOP;
            cCLOAD;             NOP;
            cCLOAD;             NOP;
// end program
    LB(32); cHALT;              NOP;
            cPRUN(0);           NOP;
```

The result is:

```
t=1  pc=0  cAcc=x   ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]       B = [xxxxxxxxxxxxxxxx]
t=43 pc=1  cAcc=x   ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]       B = [xxxxxxxxxxxxxxxx]
t=45 pc=2  cAcc=x   ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]       B = [xxxxxxxxxxxxxxxx]
t=47 pc=3  cAcc=x   ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]       B = [xxxxxxxxxxxxxxxx]
t=49 pc=4  cAcc=x   ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]       B = [xxxxxxxxxxxxxxxx]
t=51 pc=5  cAcc=x   ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]       B = [xxxxxxxxxxxxxxxx]
t=53 pc=6  cAcc=x   ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]       B = [xxxxxxxxxxxxxxxx]
t=55 pc=7  cAcc=x   ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]       B = [1111111111111111]
t=57 pc=8  cAcc=x   ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]       B = [1111111111111111]
t=59 pc=8  cAcc=x   ACC=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15] B = [1111111111111111]
t=61 pc=8  cAcc=4   ACC=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16] B = [1111111111111111]
t=63 pc=8  cAcc=3   ACC=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16] B = [1111111111111111]
t=65 pc=8  cAcc=2   ACC=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16] B = [1111111111111111]
t=67 pc=9  cAcc=1   ACC=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16] B = [1111111111111111]
t=69 pc=10 cAcc=0   ACC=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16] B = [1111111111111111]
t=71 pc=11 cAcc=-1  ACC=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16] B = [1111111111111111]
t=73 pc=12 cAcc=136 ACC=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16] B = [1111111111111111]
t=75 pc=13 cAcc=31  ACC=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16] B = [1111111111111111]
t=77 pc=13 cAcc=1   ACC=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16] B = [1111111111111111]
t=79 pc=13 cAcc=16  ACC=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16] B = [1111111111111111]
t=81 pc=0  cAcc=16  ACC=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16] B = [1111111111111111]
```

◇

2.3.5 Search-based operations

Example 2.6 *Test the effect of the instruction* `WHEREFIRST` *(select the first active cell),* `VSEARCH(val)` *(keep active all the cells where* `acc[i] = val`*) and* `VCSEARCH(val)` *(keep active all the cells where* `acc[i] = val` *if it is preceded by an active cell).*

```
/*****************************************************************
File name: 0_aProgram.sv
Description:
*****************************************************************/
            cPLOAD(0);   ACTIVATE;
            cNOP;        GETIO(1);
            cNOP;        IXLOAD;          // load index
// the program
            cVLOAD(8);   VADD(-5);        // subtract 5 form each cell
            cNOP;        WHERENCARRY;     // select where is no carry
            cNOP;        NOP;
            cNOP;        WHEREFIRST;      // keep selected the first
            cNOP;        VADD(20);        // add 20
            cNOP;        ACTIVATE;        // activate all cells
            cNOP;        NOP;
            cNOP;        VSEARCH(7);      // search where acc = 7
            cNOP;        VCSEARCH(8);     // select where the next is 8
            cNOP;        VADD(10);        // add 10 in the selected cell
// end program
    LB(32); cHALT;       NOP;
            cPRUN(0);    NOP;
```

The result is:

```
t=0  pc=x  cAcc=x ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]          B = [xxxxxxxxxxxxxxxx]
t=1  pc=0  cAcc=x ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]          B = [xxxxxxxxxxxxxxxx]
t=43 pc=1  cAcc=x ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]          B = [xxxxxxxxxxxxxxxx]
t=45 pc=2  cAcc=x ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]          B = [xxxxxxxxxxxxxxxx]
t=47 pc=3  cAcc=x ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]          B = [xxxxxxxxxxxxxxxx]
t=49 pc=4  cAcc=x ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]          B = [xxxxxxxxxxxxxxxx]
t=51 pc=5  cAcc=x ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]          B = [xxxxxxxxxxxxxxxx]
t=53 pc=6  cAcc=8 ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]          B = [xxxxxxxxxxxxxxxx]
t=55 pc=7  cAcc=8 ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]          B = [1111111111111111]
t=57 pc=8  cAcc=8 ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]          B = [1111111111111111]
t=59 pc=9  cAcc=8 ACC=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]    B = [1111111111111111]
t=61 pc=10 cAcc=8 ACC=[-5, -2, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]    B = [1111111111111111]
t=63 pc=11 cAcc=8 ACC=[-5, -2, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]    B = [0000011111111111]
t=65 pc=12 cAcc=8 ACC=[-5, -2, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]    B = [0000011111111111]
t=67 pc=13 cAcc=8 ACC=[-5, -2, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]    B = [0000010000000000]
t=69 pc=13 cAcc=8 ACC=[-5, -2, -3, -2, -1, 20, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]   B = [0000010000000000]
t=71 pc=13 cAcc=8 ACC=[-5, -2, -3, -2, -1, 20, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]   B = [1111111111111111]
t=73 pc=0  cAcc=8 ACC=[-5, -2, -3, -2, -1, 20, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]   B = [1111111111111111]
t=75 pc=0  cAcc=8 ACC=[-5, -2, -3, -2, -1, 20, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]   B = [0000000000001000]
t=77 pc=0  cAcc=8 ACC=[-5, -2, -3, -2, -1, 20, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]   B = [0000000000000100]
t=79 pc=0  cAcc=8 ACC=[-5, -2, -3, -2, -1, 20, 1, 2, 3, 4, 5, 6, 7, 18, 9, 10]  B = [0000000000000100]
```

◇

2.3.6 Shift register operations

Example 2.7 *Test the global shift and rotate instructions bi loading the serial register with the index vector.*

```
/*****************************************************************
File name: 0_aProgram.sv
```

```
Description:
**********************************************************************/
                cPLOAD(0);      ACTIVATE;
                cNOP;           GETIO(1);
                cNOP;           IXLOAD;
// the program
                cNOP;           SENDSR;
                cGRROTATE;      NOP;        // global right rotate
                cGRROTATE;      NOP;
                cGLROTATE;      NOP;        // global left rotate
                cGLROTATE;      NOP;
                cGRSHIFT;       NOP;        // global right shift
                cGRSHIFT;       NOP;
                cGLSHIFT;       NOP;        // global left shift
                cGLSHIFT;       NOP;
// end program
    LB(32);     cHALT;          NOP;
                cPRUN(0);       NOP;
```

The result is:

```
t=1  pc=0  ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]      SR=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]
t=41 pc=1  ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]      SR=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]
t=43 pc=2  ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]      SR=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]
t=45 pc=3  ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]      SR=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]
t=47 pc=4  ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]      SR=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]
t=49 pc=5  ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]      SR=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]
t=51 pc=6  ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]      SR=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]
t=53 pc=7  ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]      SR=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]
t=55 pc=8  ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]      SR=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]
t=57 pc=9  ACC=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15] SR=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]
t=59 pc=10 ACC=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15] SR=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
t=61 pc=11 ACC=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15] SR=[15, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
t=63 pc=12 ACC=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15] SR=[14, 15, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
t=65 pc=12 ACC=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15] SR=[15, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
t=67 pc=12 ACC=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15] SR=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
t=69 pc=0  ACC=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15] SR=[0, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
t=71 pc=0  ACC=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15] SR=[0, 0, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
t=73 pc=0  ACC=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15] SR=[0, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 0]
t=75 pc=0  ACC=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15] SR=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 0, 0]
```

◇

Example 2.8 *Accumulate the index vector with its value and its value globally shifted right three times.*

```
/**********************************************************************
File name: 0_aProgram.sv
Description:
**********************************************************************/
                cPLOAD(0);      ACTIVATE;
                cNOP;           GETIO(1);
                cNOP;           IXLOAD;
// the program
                cNOP;           SENDSR;
                cGRSHIFT;       SRADD;
                cGRSHIFT;       SRADD;
                cGRSHIFT;       SRADD;
                cGRSHIFT;       SRADD;
// end program
```

```
    LB(32); cHALT;          NOP;
            cPRUN(0);       NOP;
```

The result is:

```
t=0  pc=x ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]          SR=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]
t=1  pc=0 ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]          SR=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]
t=33 pc=1 ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]          SR=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]
t=35 pc=2 ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]          SR=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]
t=37 pc=3 ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]          SR=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]
t=39 pc=4 ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]          SR=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]
t=41 pc=5 ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]          SR=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]
t=43 pc=6 ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]          SR=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]
t=45 pc=7 ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]          SR=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]
t=47 pc=8 ACC=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]          SR=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]
t=49 pc=8 ACC=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]    SR=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]
t=51 pc=8 ACC=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]    SR=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
t=53 pc=0 ACC=[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30] SR=[0, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
t=55 pc=0 ACC=[0, 2, 5, 8, 11, 14, 17, 20, 23, 26, 29, 32, 35, 38, 41, 44] SR=[0, 0, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
t=57 pc=0 ACC=[0, 2, 5, 9, 13, 17, 21, 25, 29, 33, 37, 41, 45, 49, 53, 57] SR=[0, 0, 0, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
t=59 pc=0 ACC=[0, 2, 5, 9, 14, 19, 24, 29, 34, 39, 44, 49, 54, 59, 64, 69] SR=[0, 0, 0, 0, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

◇

Example 2.9 *Teste the insert and delete instructions performed on the content of the shift register (SR).*

```
/*****************************************************************
File name: 0_aProgram.sv
Description:
*****************************************************************/
            cPLOAD(0);      ACTIVATE;
            cNOP;           GETIO(1);
            cNOP;           IXLOAD;
// the program
            ccNOP;          SENDSR;     // load SR with index vector
            cNOP;           VSEARCH(8); // select the cell with acc[i]=8
            cNOP;           NOP;
            cNOP;           DELETE;     // delete RS in the selected cell
            cNOP;           DELETE;
            cVLOAD(55);     INSERT(66); // insert value 66
            cNOP;           CINSERT;    // insert acc=55
// end program
    LB(32); cHALT;          NOP;
            cPRUN(0);       NOP;
```

The result is:

```
t=0  pc=x  cAcc=x  ACC=[x, x, x, ...  x,  x,  x] SR=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]        B = [xxxxxxxxxxxxxxxx]
t=1  pc=0  cAcc=x  ACC=[x, x, x, ...  x,  x,  x] SR=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]        B = [xxxxxxxxxxxxxxxx]
t=37 pc=1  cAcc=x  ACC=[x, x, x, ...  x,  x,  x] SR=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]        B = [xxxxxxxxxxxxxxxx]
t=39 pc=2  cAcc=x  ACC=[x, x, x, ...  x,  x,  x] SR=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]        B = [xxxxxxxxxxxxxxxx]
t=41 pc=3  cAcc=x  ACC=[x, x, x, ...  x,  x,  x] SR=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]        B = [xxxxxxxxxxxxxxxx]
t=43 pc=4  cAcc=x  ACC=[x, x, x, ...  x,  x,  x] SR=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]        B = [xxxxxxxxxxxxxxxx]
t=45 pc=5  cAcc=x  ACC=[x, x, x, ...  x,  x,  x] SR=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]        B = [xxxxxxxxxxxxxxxx]
t=47 pc=6  cAcc=x  ACC=[x, x, x, ...  x,  x,  x] SR=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]        B = [xxxxxxxxxxxxxxxx]
t=49 pc=7  cAcc=x  ACC=[x, x, x, ...  x,  x,  x] SR=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]        B = [1111111111111111]
t=51 pc=8  cAcc=x  ACC=[x, x, x, ...  x,  x,  x] SR=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]        B = [1111111111111111]
t=53 pc=9  cAcc=x  ACC=[0, 1, 2, ... 13, 14, 15] SR=[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]        B = [1111111111111111]
t=55 pc=10 cAcc=x  ACC=[0, 1, 2, ... 13, 14, 15] SR=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]  B = [1111111111111111]
t=57 pc=10 cAcc=55 ACC=[0, 1, 2, ... 13, 14, 15] SR=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]  B = [0000000010000000]
t=61 pc=0  cAcc=55 ACC=[0, 1, 2, ... 13, 14, 15] SR=[0, 1, 2, 3, 4, 5, 6, 7, 9, 10, 11, 12, 13, 14, 15, 0]  B = [0000000010000000]
t=63 pc=0  cAcc=55 ACC=[0, 1, 2, ... 13, 14, 15] SR=[0, 1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14, 15, 0, 0]  B = [0000000010000000]
t=65 pc=0  cAcc=55 ACC=[0, 1, 2, ... 13, 14, 15] SR=[0, 1, 2, 3, 4, 5, 6, 7, 66, 10, 11, 12, 13, 14, 15, 0] B = [0000000010000000]
t=67 pc=0  cAcc=55 ACC=[0, 1, 2, ... 13, 14, 15] SR=[0, 1, 2, 3, 4, 5, 6, 7, 55, 66, 10, 11, 12, 13, 14, 15] B = [0000000010000000]
```

◇

Chapter 3

Library Level

The library of functions FLA defined in Section 2.1.2 in Table 2.2 is implemented in assembler in file `00_theKernel.sv`. When FLA is loaded in controller's program emory its execution halts on its first line. It is activated by programs written for the host computer: `0_hProgram.sv`.

3.1 Designing Library

The 16 functions of FLA are accessed through a jump located at the beginning of the program after the initial instruction `HALT`.

```
/*****************************************************************
File name: 00_theKernel.v
Description: LINEAR ALGEBRA KERNEL LIBRARY FUNCTIONS
*****************************************************************/
            cHALT;      NOP;
            cJMP(1);    NOP; // hSTART
            cJMP(2);    NOP; // hSTOP
            cJMP(3);    NOP; // hINTRQ
            cJMP(4);    NOP; // hSQGENX(d)
            cJMP(5);    NOP; // hSQGENN
            cJMP(6);    NOP; // hMSEND(addr-1,size)
            cJMP(7);    NOP; // hMGET(addr-1,size)
            cJMP(8);    NOP; // hMAIN(addr-1)
            cJMP(9);    NOP; // hSQADD(dest,left,right)
            cJMP(10);   NOP; // hVGENX(addr)
            cJMP(11);   NOP; // hVGENN(addr,value)
            cJMP(12);   NOP; // hSQMVMULT(matrix,vectror,dest)
            cJMP(13);   NOP; // hSQMMULT(dest,left,right)
            cJMP(14);   NOP; // hSQMMAC(dest,left,right)
            cJMP(15);   NOP; // hTRANS(dest,left)
            cJMP(16);   NOP; // hPRANDOM(dest)
//********** START COUNTER ***************************************
    LB(1);  cSTART;         VLOAD(22);
            cJMP(32);       NOP;
//********** STOP COUNTER ****************************************
    LB(2);  cSTOP;          VLOAD(33);
            cJMP(32);       NOP;
```

```
//********** INTERRUPT REQEST ************************************
    LB(3);   cSETINT;          VLOAD(44);
             cJMP(32);         NOP;
//********** SQUARE MATRIX X GENERATE ****************************
    LB(4);   cPARAM;           NOP;
             cNOP;             CLOAD;
             cVLOAD('p-1);     VSUB(1);
             cNOP;             ADDRLD;
             cNOP;             IXLOAD;
    LB(17);  cNOP;             RISTORE(1);
             cBRNZDEC(17);     VADD(1);
             cJMP(32);         NOP;
//********** SQUARE MATRIX N GENERATE ****************************
    LB(5);   cPARAM;           NOP;
             cNOP;             CLOAD;
             cVLOAD('p-1);     VSUB(1);
             cNOP;             ADDRLD;
             cNOP;             VLOAD(0);
    LB(18);  cNOP;             RISTORE(1);
             cBRNZDEC(18);     VADD(1);
             cJMP(32);         NOP;
//********** SEND MATRIX *****************************************
    LB(6);   cPARAM;           NOP;
             cPARAM;           CLOAD;
             cNOP;             ADDRLD;
             cNOP;             NOP;
             cNOP;             RISENDIO(0);
    LB(19);  cNOP;             NOP;
             cBRZDEC(32);      NOP;
             cNOP;             NOP;
             cNOP;             NOP;
             cNOP;             NOP;
             cDATAEXT('p/2);   NOP;
             cJMP(19);         RISENDIO(1);
//********** GET MATRIX ******************************************
    LB(7);   cPARAM;           GETIO('m-1);
             cVSUB(1);         NOP;
             cPARAM;           CLOAD;
             cVSUB(1);         ADDRLD;
             cNOP;             NOP;
             cNOP;             NOP;
    LB(20);  cDATAINS('p/2);   NOP;
             cNOP;             RIGETIO(1);
             cBRZDEC(32);      NOP;
             cNOP;             NOP;
             cNOP;             NOP;
             cNOP;             NOP;
             cJMP(20);         NOP;
//********** MAIN MATRIX GENERATE ********************************
    LB(8);   cPARAM;           NOP;
             cNOP;             CLOAD;
             cPARAM;           ADDRLD;
```

```
            cNOP;               IXLOAD;
            cNOP;               WHEREZERO;
            cNOP;               CLOAD;
            cNOP;               ELSEWHERE;
            cNOP;               VLOAD(0);
            cNOP;               ENDWHERE;
            cVLOAD('p-2);       SENDSR;
            cGRSHIFT;           RSTORE(0);
    LB(22); cNOP;               GETSR;
            cGRSHIFT;           RISTORE(1);
            cBRNZDEC(22);       NOP;
            cJMP(32);           NOP;
//********** ADD SQUARE MATRICES ************************************
    LB(9);  cPARAM;             NOP;
            cSTORE(3);          NOP;      // dest at mem[3]
            cPARAM;             NOP;
            cSTORE(4);          NOP;      // left at mem[4]
            cPARAM;             NOP;
            cSTORE(5);          NOP;      // right at mem[5]
            cSUB(4);            NOP;
            cSTORE(0);          NOP;      // right-left at mem[0]
            cLOAD(3);           NOP;
            cSUB(5);            NOP;
            cSTORE(1);          NOP;      // dest-right at mem[1]
            cLOAD(4);           NOP;
            cSUB(3);            NOP;
            cVADD(1);           NOP;
            cSTORE(2);          NOP;      // left-dest+1 at mem[2]
            cVLOAD('p);         NOP;
            cSTORE(6);          NOP;
    LB(23); cLOAD(4);           NOP;
            cADD(0);            CALOAD;
            cADD(1);            CAADD;
            cADD(2);            CSTORE;
            cSTORE(4);          NOP;
            cLOAD(6);           NOP;
            cVSUB(1);           NOP;
            cSTORE(6);          NOP;
            cBRNZ(23);          NOP;
            cJMP(32);           NOP;
//*********** INDEX VECTOR GENERATE *********************************
    LB(10); cPARAM;             IXLOAD;
            cJMP(32);           CSTORE;
//*********** N VECTOR GENERATE *************************************
    LB(11); cPARAM;             NOP;
            cPARAM;             CLOAD;
            cJMP(32);           CSTORE;
//*********** MATRIX-VECTOR MULTIPLY ********************************
    LB(12); cPARAM;                   NOP;      // dest address
            cSTORE(1);                NOP;      // mem[1] dest address
            cPARAM;                   NOP;      // matrix address
            cVADD('p);                REDADD;   // end matrix
```

```
            cNOP;                       CLOAD;
            cPARAM;                     ADDRLD;
            cNOP;                       CALOAD;
            cNOP;                       STORE(0);
            cVLOAD('p-1);               RILOAD(-1);
    LB(24); cLREDINS;                   MULT(0);
            cBRNZDEC(24);               RILOAD(-1);
            cVLOAD($clog2('p));         NOP;
    LB(27); cBRNZDEC(27);               NOP;
            cLOAD(1);                   GETSR;
            cJMP(32);                   CSTORE;
//********** MULTIPLY SQUARE MATRICES ********************************
    LB(13); cPARAM;                     REDADD;
            cVSUB(1);                   NOP;
            cSTORE(3);                  NOP;      // dest => 3
            cPARAM;                     NOP;
            cVADD('p-1);                NOP;
            cSTORE(0);                  NOP;      // left => 0
            cPARAM;                     CLOAD;
            cSTORE(2);                  ADDRLD; // right => 2
            cVLOAD('p);                 NOP;
            cSTORE(1);                  NOP;
            cLOAD(2);                   NOP;
            cVADD(1);                   CALOAD;
            cSTORE(2);                  STORE(0);
            cVLOAD('p-1);               RILOAD(0);
    LB(25); cLREDINS;                   MULT(0);
            cBRNZDEC(25);               RILOAD(-1);
            cVLOAD($clog2('p)-3);       NOP;
    LB(28); cBRNZDEC(28);               NOP;
            cLOAD(3);                   NOP;
            cVADD(1);                   NOP;
            cSTORE(3);                  GETSR;
            cLOAD(2);                   CSTORE;
            cVADD(1);                   CALOAD;
            cSTORE(2);                  NOP;
            cLOAD(0);                   STORE(0);
            cLOAD(1);                   CLOAD;
            cVSUB(1);                   NOP;
            cBRZ(32);                   ADDRLD;
            cSTORE(1);                  NOP;
            cVLOAD('p-1);               RILOAD(0);
            cJMP(25);                   NOP;
//********** MULTIPLY & ACCUMULATE SQUARE MATRICES ******************
    LB(14); cPARAM;                     REDADD;
            cVSUB(1);                   NOP;
            cSTORE(3);                  NOP;      // dest => 3
            cPARAM;                     NOP;
            cVADD('p-1);                NOP;
            cSTORE(0);                  NOP;      // left => 0
            cPARAM;                     CLOAD;
            cSTORE(2);                  ADDRLD; // right => 2
```

```
                cVLOAD(‘p);                NOP;
                cSTORE(1);                 NOP;
                cLOAD(2);                  NOP;
                cVADD(1);                  CALOAD;
                cSTORE(2);                 STORE(0);
                cVLOAD(‘p-1);              RILOAD(0);
    LB(26);     cLREDINS;                  MULT(0);
                cBRNZDEC(26);              RILOAD(-1);
                cVLOAD($clog2(‘p)-3);      NOP;
    LB(29);     cBRNZDEC(29);              NOP;
                cLOAD(3);                  NOP;
                cVADD(1);                  NOP;
                cSTORE(3);                 GETSR;
                NOP;                       CAADD;
                cLOAD(2);                  CSTORE;
                cVADD(1);                  CALOAD;
                cSTORE(2);                 NOP;
                cLOAD(0);                  STORE(0);
                cLOAD(1);                  CLOAD;
                cVSUB(1);                  NOP;
                cBRZ(32);                  ADDRLD;
                cSTORE(1);                 NOP;
                cVLOAD(‘p-1);              RILOAD(0);
                cJMP(26);                  NOP;
//********** TRANSPOSE ********************************************
    LB(15);     cPARAM;                    IXLOAD;
                cSTORE(0);                 SENDSR; // mem[0]=dest
                cPARAM;                    NOP;
                cSTORE(1);                 NOP;     // mem[1]= source
                cVADD(2*‘p);               NOP;
                cSTORE(2);                 NOP;     // mem[2]=temp
                cVLOAD(‘p+1);              NOP;     // counter
                cSTORE(3);                 NOP;
// READ ALL
    LB(33);     cLOAD(1);                  GETSR;
                cGRROTATE;                 CADD;
                cLOAD(3);                  ADDRLD;
                cVSUB(1);                  NOP;
                cBRZ(34);                  NOP;
                cSTORE(3);                 NOP;
                cLOAD(2);                  RLOAD(0);
                cVADD(1);                  CSTORE;
                cSTORE(2);                 NOP;
                cJMP(33);                  NOP;
// ROTATE ALL
    LB(34);     cVLOAD(‘p);                VLOAD(3*‘p+8);
                cSTORE(3);                 ADDRLD;
                cNOP;                      NOP;
    LB(35);     cVSUB(1);                  RILOAD(-1);
                cBRZ(37);                  SENDSR;
                cSTORE(3);                 NOP;
    LB(36);     cGLROTATE;                 GETSR;
```

```
            cBRNZDEC(36);           NOP;
            cLOAD(3);               NOP;
            cJMP(35);               RSTORE(0);
// STORE ALL
    LB(37); cVLOAD(`p-1);           IXLOAD;
            cSTORE(3);              SENDSR;
            cGRROTATE;              NOP;
    LB(38); cLOAD(0);               GETSR;
            cLOAD(3);               CADD;
            cVADD(2*`p+8);          ADDRLD;
            cGRROTATE;              CALOAD;
            cLOAD(3);               RSTORE(0);
            cBRZDEC(32);            NOP;
            cSTORE(3);              NOP;
            cJMP(38);               NOP;
//********** PSEUDO-RANDOM MATRIX ************************************
    LB(16); cPARAM;                 ACTIVATE;
            cNOP;                   CLOAD;
            cNOP;                   ADDRLD;
            cVLOAD(`p-1);           IXLOAD;
    LB(33); cNOP;                   VADD(29);
            cNOP;                   VMULT(98765);
            cNOP;                   SHR;
            cNOP;                   SHR;
            cNOP;                   SHR;
            cNOP;                   SHR;
            cNOP;                   SHR;
            cNOP;                   VAND(31);
            cNOP;                   CRSTORE;
            cBRNZDEC(33);           NOP;
            cJMP(32);               NOP;
```

3.2 Using Library

The FLA is inserted into the program run by the accelerator as follows:

```
/*****************************************************************
File name: 0_aProgram.sv
Description: load in V[2] the index vector incremented with 1
*****************************************************************/
            cPLOAD(0);  ACTIVATE; // load program from 0; all cell active
            cNOP;       GETIO(1); // discard the I/O register of array
            cNOP;       IXLOAD;   // ACC <= [0 1 ... (p-1)]
// the program
          `include "00_theKernel.sv"
// end program
    LB(32); cHALT;      NOP;
            cPRUN(0);   NOP;      // run from 0
```

Example 3.1 *The host program that loads a matrix of $p \times p$ pseudo-randomly generated elements into the accelerator is the following:*

```
/*************************************************************************
File name: 0_hProgram.sv
                        HOST PROGRAM
*************************************************************************/
    hVALUE(0,0);      // rf[0] <= 0
    hPSEND(0,0);      // send program from 0 and run it from 0
// the host program
    hSTART;
    hPRANDOM(8);      // generate at 8 a pseudo-random matrix
    hSTOP;
// end program
    hHALT;
```

The result:

```
vect[7]  = [x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]
vect[8]  = [21, 16, 30, 13, 27, 29, 24, 6, 21, 3, 17, 8, 21, 29, 16, 13]
vect[9]  = [8, 21, 22, 3, 24, 13, 6, 27, 8, 29, 30, 16, 8, 13, 21, 3]
vect[10] = [16, 8, 17, 29, 6, 3, 27, 24, 16, 13, 22, 21, 16, 3, 8, 29]
vect[11] = [21, 16, 30, 13, 27, 29, 24, 6, 21, 3, 17, 8, 21, 29, 16, 13]
vect[12] = [8, 21, 22, 3, 24, 13, 6, 27, 8, 29, 30, 16, 8, 13, 21, 3]
vect[13] = [16, 8, 17, 29, 6, 3, 27, 24, 16, 13, 22, 21, 16, 3, 8, 29]
vect[14] = [21, 16, 30, 13, 27, 29, 24, 6, 21, 3, 17, 8, 21, 29, 16, 13]
vect[15] = [8, 21, 22, 3, 24, 13, 6, 27, 8, 29, 30, 16, 8, 13, 21, 3]
vect[16] = [16, 8, 17, 29, 6, 3, 27, 24, 16, 13, 22, 21, 16, 3, 8, 29]
vect[17] = [21, 16, 30, 13, 27, 29, 24, 6, 21, 3, 17, 8, 21, 29, 16, 13]
vect[18] = [8, 21, 22, 3, 24, 13, 6, 27, 8, 29, 30, 16, 8, 13, 21, 3]
vect[19] = [16, 8, 17, 29, 6, 3, 27, 24, 16, 13, 22, 21, 16, 3, 8, 29]
vect[20] = [21, 16, 30, 13, 27, 29, 24, 6, 21, 3, 17, 8, 1, 29, 16, 13]
vect[21] = [8, 21, 22, 3, 24, 13, 6, 27, 8, 29, 30, 16, 0, 13, 1, 3]
vect[22] = [16, 8, 17, 29, 6, 3, 27, 24, 16, 13, 22, 1, 11, 3, 0, 9]
vect[23] = [1, 16, 30, 13, 27, 9, 24, 6, 21, 3, 17, 0, 14, 29, 11, 25]
vect[24] = [x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]

aCC=175
```

The execution time is 175 clock cycles (the start and stop of the clock cycle counter is included).
◇

Example 3.2 *The transpose operation for the square matrix of $p \times p$ elements generated starting with the vector 8 is described by the following program:*

```
/*************************************************************************
File name: 0_hProgram.sv
                        HOST PROGRAM
*************************************************************************/
    hVALUE(0,0);      // rf[0] <= 0
```

```
    hPSEND(0,0);      // send program from 0 and run in array from 0
// the host program
    hSQGENN(8);       // generate at 8 matrix N
    hSTART;
    hTRANS('p+8,8); // transpose from 8 to p+8
    hSTOP;
// end program
    hHALT;
```

The result:

```
vect[8]  = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
vect[9]  = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
vect[10] = [2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
vect[11] = [3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
vect[12] = [4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4]
vect[13] = [5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5]
vect[14] = [6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6]
vect[15] = [7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7]
vect[16] = [8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8]
vect[17] = [9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9]
vect[18] = [10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10]
vect[19] = [11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11]
vect[20] = [12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12]
vect[21] = [13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13]
vect[22] = [14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14]
vect[23] = [15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15]
vect[24] = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
vect[25] = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
vect[26] = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
vect[27] = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
vect[28] = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
vect[29] = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
vect[30] = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
vect[31] = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
vect[32] = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
vect[33] = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
vect[34] = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
vect[35] = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
vect[36] = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
vect[37] = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
vect[38] = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
vect[39] = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]

aCC=662
```

For each component of the transposed matrix a number of $652/256 = 2.52$ *clock cycles are used. The acceleration obtained is only in* $O(1)$*. As p increases, the number of cycles per element of the resulting matrix reduces, converging to 1.*

◇

Example 3.3 *Matrix-vector multiplication is exemplified with matrix obtained starting from the index vector, IX, and the a constant vector containing 3.*

```
/*********************************************************************
File name: 0_hProgram.sv
                    HOST PROGRAM
*********************************************************************/
    hVALUE(0,0);      // rf[0] <= 0
    hPSEND(0,0);      // send program from 0 and run in array from 0
// the host program
    hSQGENX(8);                    // generate matrix IX at 8
    hVGENN(3,`p+8);                // generate constant 3 vector at p+8
    hSTART;
    hSQMVMULT(`p+9, 8, `p+8);      // matrix-vector multiply
    hSTOP;
// end program
    hHALT;
```

The result:

```
vect[0]  = [3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
vect[1]  = [x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]
vect[2]  = [x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]
vect[3]  = [x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]
vect[4]  = [x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]
vect[5]  = [x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]
vect[6]  = [x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]
vect[7]  = [x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]
vect[8]  = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
vect[9]  = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]
vect[10] = [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17]
vect[11] = [3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18]
vect[12] = [4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
vect[13] = [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
vect[14] = [6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21]
vect[15] = [7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22]
vect[16] = [8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23]
vect[17] = [9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24]
vect[18] = [10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]
vect[19] = [11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26]
vect[20] = [12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27]
vect[21] = [13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28]
vect[22] = [14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29]
vect[23] = [15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30]
vect[24] = [3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
vect[25] = [360, 408, 456, 504, 552, 600, 648, 696, 744, 792, 840, 888, 936, 984, 1032, 1080]

aCC=59
```

The execution time per element of vector is $49/16 = 3.06$ *clock cycles. It corresponds to 16 multiplications and 15 additions. As p increases, the number of clock cycles per element of the resulting matrix reduces, converging to 2.*

◇

Example 3.4 *Matrix multiplication is exemplified using IX matrix generated from vector 8 and the diagonal matrix, containing 2 as non-zero elements, generated immediately after the first matrix. The result is computed immediately after the second matrix.*

```
/*************************************************************************
File name: 0_hProgram.sv
                    HOST PROGRAM
*************************************************************************/
    hVALUE(0,0); // rf[0] <= 0
    hPSEND(0,0); // send program from 0 and run in array from 0
// the host program
    hSQGENX(8);                  // to array: generate matrix X
    hMAIN(`p+8,2);               // to array: generate matrix UNIT
    hSTART;
    hSQMMULT(2*`p+8,`p+8,8);     // to array: multiply matrices
    hSTOP;
// end program
    hHALT;
```

The result:

```
vect[8] = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
vect[9] = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]
vect[10] = [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17]
vect[11] = [3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18]
vect[12] = [4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
vect[13] = [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
vect[14] = [6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21]
vect[15] = [7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22]
vect[16] = [8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23]
vect[17] = [9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24]
vect[18] = [10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]
vect[19] = [11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26]
vect[20] = [12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27]
vect[21] = [13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28]
vect[22] = [14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29]
vect[23] = [15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30]
vect[24] = [2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
vect[25] = [0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
vect[26] = [0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
vect[27] = [0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
vect[28] = [0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
vect[29] = [0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
vect[30] = [0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0]
vect[31] = [0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0]
vect[32] = [0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0]
vect[33] = [0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0]
vect[34] = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0]
vect[35] = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0]
vect[36] = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0]
vect[37] = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0]
vect[38] = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0]
vect[39] = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2]
vect[40] = [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30]
vect[41] = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32]
vect[42] = [4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34]
vect[43] = [6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36]
vect[44] = [8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38]
vect[45] = [10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40]
vect[46] = [12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42]
vect[47] = [14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44]
vect[48] = [16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46]
vect[49] = [18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48]
vect[50] = [20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50]
vect[51] = [22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52]
vect[52] = [24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54]
```

```
vect[53] = [26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56]
vect[54] = [28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58]
vect[55] = [30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60]

aCC=789
```

The execution time per element of the resulting matrix is $779/256 = 3.08$ *clock cycles. As p increases, the number of clock cycles per element of the resulting matrix reduces, converging to 2.*

◇

Example 3.5 *Matrix addition is exemplified using IX matrix generated from vector 0 and the diagonal matrix, containing 55 as non-zero elements, generated immediately after the first matrix. The result is computed at the end of the second matrix.*

```
/*************************************************************************
File name: 0_hProgram.sv
                    HOST PROGRAM
*************************************************************************/
    hVALUE(0,0); // rf[0] <= 0
    hPSEND(0,0); // send program from 0 and run in array from 0
// the host program
    hSQGENX(0);              // to array: generate matrix X
    hMAIN('p,55);            // to array: generate matrix UNIT
    hSTART;
    hSQMADD(2*'p, 0, 'p);    // to array: add matrices
    hSTOP;
// end program
    hHALT;
```

The result:

```
vect[0] = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
vect[1] = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]
vect[2] = [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17]
vect[3] = [3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18]
vect[4]  = [4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
vect[5]  = [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
vect[6]  = [6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21]
vect[7]  = [7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22]
vect[8]  = [8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23]
vect[9]  = [9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24]
vect[10] = [10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]
vect[11] = [11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26]
vect[12] = [12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27]
vect[13] = [13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28]
vect[14] = [14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29]
vect[15] = [15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30]
vect[16] = [55, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
vect[17] = [0, 55, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
vect[18] = [0, 0, 55, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
vect[19] = [0, 0, 0, 55, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
vect[20] = [0, 0, 0, 0, 55, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
vect[21] = [0, 0, 0, 0, 0, 55, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
vect[22] = [0, 0, 0, 0, 0, 0, 55, 0, 0, 0, 0, 0, 0, 0, 0, 0]
vect[23] = [0, 0, 0, 0, 0, 0, 0, 55, 0, 0, 0, 0, 0, 0, 0, 0]
vect[24] = [0, 0, 0, 0, 0, 0, 0, 0, 55, 0, 0, 0, 0, 0, 0, 0]
vect[25] = [0, 0, 0, 0, 0, 0, 0, 0, 0, 55, 0, 0, 0, 0, 0, 0]
vect[26] = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 55, 0, 0, 0, 0, 0]
vect[27] = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 55, 0, 0, 0, 0]
```

```
vect[28] = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 55, 0, 0, 0]
vect[29] = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 55, 0, 0]
vect[30] = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 55, 0]
vect[31] = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 55]
vect[32] = [55, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
vect[33] = [1, 57, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]
vect[34] = [2, 3, 59, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17]
vect[35] = [3, 4, 5, 61, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18]
vect[36] = [4, 5, 6, 7, 63, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
vect[37] = [5, 6, 7, 8, 9, 65, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
vect[38] = [6, 7, 8, 9, 10, 11, 67, 13, 14, 15, 16, 17, 18, 19, 20, 21]
vect[39] = [7, 8, 9, 10, 11, 12, 13, 69, 15, 16, 17, 18, 19, 20, 21, 22]
vect[40] = [8, 9, 10, 11, 12, 13, 14, 15, 71, 17, 18, 19, 20, 21, 22, 23]
vect[41] = [9, 10, 11, 12, 13, 14, 15, 16, 17, 73, 19, 20, 21, 22, 23, 24]
vect[42] = [10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 75, 21, 22, 23, 24, 25]
vect[43] = [11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 77, 23, 24, 25, 26]
vect[44] = [12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 79, 25, 26, 27]
vect[45] = [13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 81, 27, 28]
vect[46] = [14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 83, 29]
vect[47] = [15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 85]
```

◇

Example 3.6 *Matrix multiplication and accumulation (MACC) is illustration using IX matrix and unit diagonal matrix. They al multiplied and then the second multiplication is added with the previous multiplication.*

```
/*************************************************************************
File name: 0_hProgram.sv
                        HOST PROGRAM
*************************************************************************/
    hVALUE(0,0); // rf[0] <= 0
    hPSEND(0,0); // send program from 0 and run in array from 0
// the host program
    hSQGENX(8);                     // generate matrix X
    hMAIN(`p+8,1);                  // generate matrix UNIT
    hSQMMULT(2*`p+8,`p+8,8);        // multiply matrices
    hSTART;
    hSQMMAC(2*`p+8,`p+8,8);         // multiply & accumulate matrices
    hSTOP;
// end program
    hHALT;
```

The result:

```
vect[8]  = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
vect[9]  = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]
vect[10] = [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17]
vect[11] = [3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18]
vect[12] = [4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
vect[13] = [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
vect[14] = [6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21]
vect[15] = [7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22]
vect[16] = [8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23]
vect[17] = [9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24]
vect[18] = [10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]
vect[19] = [11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26]
vect[20] = [12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27]
vect[21] = [13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28]
vect[22] = [14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29]
```

```
vect[23] = [15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30]
vect[24] = [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
vect[25] = [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
vect[26] = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
vect[27] = [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
vect[28] = [0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
vect[29] = [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
vect[30] = [0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
vect[31] = [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0]
vect[32] = [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
vect[33] = [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
vect[34] = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
vect[35] = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0]
vect[36] = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0]
vect[37] = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0]
vect[38] = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0]
vect[39] = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
vect[40] = [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30]
vect[41] = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32]
vect[42] = [4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34]
vect[43] = [6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36]
vect[44] = [8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38]
vect[45] = [10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40]
vect[46] = [12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42]
vect[47] = [14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44]
vect[48] = [16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46]
vect[49] = [18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48]
vect[50] = [20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50]
vect[51] = [22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52]
vect[52] = [24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54]
vect[53] = [26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56]
vect[54] = [28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58]
vect[55] = [30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60]

aCC=821
```

◇

Example 3.7 *Matrix transfer is tested by generating, from* `vect[0]`*, the IX matrix which is sent to the host's memory and then is get back in accelerator starting with* `vect[20]`.

```
/*****************************************************************
File name: 0_hProgram.sv
                    HOST PROGRAM
*****************************************************************/
    hVALUE(0,0);     // rf[0] <= 0
    hPSEND(0,0);     // send program from 0 and run in array from 0
// the host program
    hSQGENX(0);      // to array: generate matrix X
    hSTART;          // to array: start cycles counter
    hMSEND(0, 16);   // to array: send matrix
    hVALUE(1,127);   // for host: end address in host
    hVALUE(0,0);     // for host: start address in host
    hDGET(0,0,1);    // for host: get data in host

    hMGET(20, 16);   // to array: get matrix
    hVALUE(1,127);   // for host: end address in host
    hVALUE(0,0);     // for host: start address in host
    hDSEND(0,0,1);   // for host: send data from host
    hSTOP;           // to array: stop cycles coounter
```

```
// end program
    hHALT;
```

The result:

```
vect[0]  = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
vect[1]  = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]
vect[2]  = [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17]
vect[3]  = [3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18]
vect[4]  = [4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
vect[5]  = [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
vect[6]  = [6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21]
vect[7]  = [7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22]
vect[8]  = [8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23]
vect[9]  = [9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24]
vect[10] = [10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]
vect[11] = [11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26]
vect[12] = [12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27]
vect[13] = [13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28]
vect[14] = [14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29]
vect[15] = [15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30]
vect[16] = [x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]
vect[17] = [x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]
vect[18] = [x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]
vect[19] = [x, x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]
vect[20] = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
vect[21] = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]
vect[22] = [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17]
vect[23] = [3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18]
vect[24] = [4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
vect[25] = [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
vect[26] = [6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21]
vect[27] = [7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22]
vect[28] = [8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23]
vect[29] = [9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24]
vect[30] = [10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]
vect[31] = [11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26]
vect[32] = [12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27]
vect[33] = [13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28]
vect[34] = [14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29]
vect[35] = [15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30]

aCC=503
```

The transfer of 256 scalars back and forth is performed in 493 clock cycles, i.e., $493/512 = 0.96$ *clock cycles.*

◇

Bibliography

[1] Mihaela Maliţa, Gheorghe Ştefan, Dominique Thiébaut: "Not Multi-, but Many-Core: Designing Integral Parallel Architectures for Embedded Computation", *International Workshop on Advanced Low Power Systems held in conjunction with 21st International Conference on Supercomputing* Seattle, WA, USA, June 17, 2007.

[2] Mihaela Maliţa, George Vlăduţ Popescu, Gheorghe M. Ştefan, "Heterogenous Computing System for Deep Learning", in Witold Pedrycz, Shyi-Chen (Eds.): *Deep Learning: Concepts and Architectures*, Springer International Publishing, pp 287-319.

[3] Mihaela Maliţa, George Vlăduţ Popescu, Gheorghe M. Ştefan, "Heterogenous Computing for Markov Models in Big Data", *CSCI 2019 International Conference*.

[4] Gheorghe Ştefan, Anand Sheel, Bogdan Mîţu, Tom Thomson, Dan Tomescu: "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, August 20 to 22, 2006.

[Online]. Available:

`https://youtu.be/HMLT4EpKBAw` at 35:00

[5] Gheorghe 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 Circuits, Systems, Communications and Computers*, 2014, pp. 582–597.

[Online]. Available:

`http://www.inase.org/library/2014/santorini/bypaper/COMPUTERS/COMPUTERS2-42.pdf`