DESIGN AND IMPLEMENTATION OF A CUSTOM PROCESSOR OPTIMIZED FOR IMAGE PROCESSING

EN3030
Circuits and Systems Design

GROUP MEMBERS

150001C : G.Abarajithan
150172A : T.T.Fonseka
150689N : W.M.R.R.Wickramasinghe
150707V : C.Wimalasuriya

Abstract

In this document we describe our unique and efficient approach of implementing a processor optimized for matrix manipulation on FPGA. We present our elegant and highly user-friendly ISA which consists only 16 instructions that take only 2 clock cycles for execution. Our ISA, unlike anything before, is highly flexible with features such as: over 8 types of jump, ability to copy from one register to many at a time, ability to increment, decrement and reset upto 8 registers at once.

We describe our exclusive architectural design that has special modules traversing a matrix, shift registers for error-free convolution and special loop registers all implemented using less than 1000 logic elements. We also describe the design and implementation of a python-based compiler and a simulator we developed for remote algorithm development.

Finally we demonstrate our results: downsampling by any integer upto 16, upsampling, edge-detection, Gaussian smoothing and application of custom filters to 512 x 512 size images. All image processing algorithms were implemented with zero SSD error using only 30 lines (30 bytes) of assembly code. We also demonstrate the implementation of generic algorithms such as prime finding, fibnocci series and pascal triangle generation.

REPOSITORIES

github.com/BlazeCode2/ABRUTECH_processor_automatic

github.com/BlazeCode2/ABRUTECH_processor_manual

github.com/BlazeCode2/ABRUTECH_cache

github.com/Jester-2-6/ABRUTECH_graphic_equalizer

Contents

1 Introduction **5**

2 Instruction Set Architecture (ISA) **7**
2.1 Design Goals and Features . . . 7
2.1.1 Minimizing the number of instructions in ISA . . . 7
2.1.2 Minimizing the number of lines of a program . . . 8
2.1.3 Reducing the number of total number of clock cycles needed for the program to run . . . 8
2.1.4 Matrix Manipulation . . . 9
2.1.5 Linear Decomposition of 2D Algorithms . . . 9
2.2 Registers and Datapath . . . 11
2.3 The Instruction Set . . . 13
2.3.1 COPY . . . 13
2.3.2 INCR, DECR, RSET . . . 15
2.3.3 JUMP . . . 16
2.3.4 LOAD . . . 17
2.3.5 STAC . . . 17
2.3.6 ADD, SUBT . . . 18
2.3.7 MUL, DIV . . . 18
2.3.8 LODK . . . 19
2.3.9 LADD . . . 19
2.4 ISA: Summery . . . 20

3 Mathematical Justification of Algorithm Design **22**
3.0.1 Aliasing effect and the required 3-db cut off frequency for downsampling . 22
3.0.2 Low pass characteristics of the averaging filter with length k . . . 23
3.0.3 The 3-db frequency of discrete approximation of Gaussian kernel . . . 24

4 Algorithms **27**
4.1 Downsampling: Squeeze Averaging Algorithm . . . 28
4.2 Snake Averaging Algorithm . . . 31
4.3 Gaussian Smoothing . . . 33
4.3.1 Problem with the Traditional Algorithms . . . 33
4.3.2 Our solution . . . 33
4.4 Edge Detection Algorithms . . . 35
4.5 Upsampling Algorithms . . . 37
4.5.1 Nearest Neighbor Interpolation . . . 37
4.5.2 Bilinear interpolation . . . 38
4.6 Custom Filter . . . 39
4.7 Prime Finding Algorithm . . . 40

5 Architecture **43**
5.1 State Diagram . . . 43
5.2 Micro Instructions . . . 44
5.3 Complete Architectural Diagram . . . 46
5.4 System . . . 47
5.4.1 Top level module . . . 47
5.4.2 Processor . . . 48
5.4.3 DRAM and IRAM . . . 49
5.4.4 Memory router . . . 51
5.4.5 Clock control . . . 52
5.5 Special Modules . . . 53
5.5.1 ADR Maker . . . 53
5.5.2 Automatic Controller . . . 54
5.5.3 State Machine . . . 55
5.6 Controllers: Muxes, Demuxes and Decoders . . . 56
5.6.1 AWM(A-Bus write mux) . . . 56
5.6.2 ACI Decoder . . . 56
5.6.3 INC, DEC, RST Decoder . . . 57
5.6.4 JMP Mux . . . 57
5.6.5 PRM Decoder . . . 58
5.6.6 OPR Decoder . . . 58
5.7 Registers . . . 59
5.7.1 Shift Registers (G0, G1, G2) . . . 59
5.7.2 Looping Registers (K0, K1) . . . 60
5.7.3 Data registers(MDDR/MIDR) . . . 60
5.7.4 AW registers(AWT/AWG) . . . 61
5.7.5 AR registers(ART/ARG) . . . 61
5.7.6 PC . . . 61
5.7.7 ALU((Arithmetic and Logic Unit)) . . . 62

6 Timing Diagrams of Instructions **63**

7 Hardware Debugging Features **73**

8 Problems Faced & Solutions **75**

9 Compiler and Simulator **76**
9.1 Workflow of the Programmer or Architect . . . 76
9.2 How the Compiler and Simulator Work . . . 80
9.2.1 Parsing the ISA . . . 80
9.2.2 Making the define file . . . 80
9.2.3 Compiling . . . 81
9.3 Simulation . . . 81

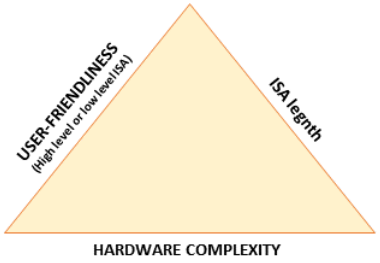
10 Communications **82**
10.1 Overview . . . 82
10.2 Transmitter . . . 83
10.3 Receiver . . . 83
10.4 Data Retriever . . . 83
10.5 Tx Modifier / Cropper . . . 84
10.6 Data Writer . . . 85
10.7 Serial Link . . . 85

10.8 MATLAB script 85

11 Results **87**
11.1 Downsample by factor 2 87
11.2 Downsample by factor 3 88
11.3 Downsample by factor 5 88
11.4 Bilinear Upsampling 89
11.5 Custom Filtering 89
11.6 Edge Detection 90
11.7 Gaussian Smoothing 90
11.8 Prime finding 91
11.9 Pascal Triangle 91
11.10Fibbonacci Series 91

1 Introduction

The major challenge in designing a processor is the trade-off between size of ISA, hardware complexity and user friendliness. ABRUTECH is a unique custom processor highly optimized to manipulate matrices while preserving the functionalities of a generic processor. It has been designed to strike the delicate balance in the above tradeoffs. While having only 16 instructions, the ISA of ABRUTECH is crafted to be simple, yet highly powerful and is implemented using only 1000 logic elements.



This is demonstrated in the results section, with sample programs that are only 30-40 bytes long but are able to downsample and upsample any 512x512 image by any integer, detect edges, find prime numbers and Fibonacci numbers. . . etc.

ABRUTECH works with an 8-bit wide, 262144-bit (512x512) deep data memory and an 8-bit wide 256-bit deep instruction memory, both of which can be loaded through UART. The system was coded in Verilog HDL using Intel Quartus II Prime and implemented successfully on an Altera de2-115 development board.

While being an 8-bit processor, (bus sizes and most register sizes being 8-bit), ABRUTECH features a 12-bit ALU and accumulator, which allows it to process calculations with intermediate steps that give results up to 4096, without causing an overflow error. The ALU is also designed to perform round-off divisions (unlike the typical floor division), to improve accuracy.

Our processor also features a special module called Address Maker, which (optionally) allows the programmer to navigate a 512 x 512 matrix either row wise or column wise, without the need of a complex algorithm. This helps in implementing image processing algorithms such as downsampling, nearest neighbor upsampling, upsampling by bilinear interpolation.

Another special feature in ABRUTECH is a bank of shift registers, which help to perform linear convolution operations several times faster. Together with Address Maker, this allows the

programmer to perform 2D convolution with a linearly decomposable kernel, such as Gaussian smoothing or edge detection, without losing a row and a column of data in the process, as with the traditional algorithms.

The qualities of a generic processor are also preserved, which is presented in the section 'Preservation of Genericity'. Algorithms to calculate the Fibonacci sequence and to find prime numbers less than 256 have been implemented and presented with results.

Results section of the report portrays the results of each of these algorithms. Implementation of each of these algorithms resulted in a sum of squared difference (SSD) error of zero, which shows the accuracy of our FPGA implementation.

We also built a corresponding compiler program, which scans the excel sheet where ISA is specified and translates the algorithm written in human language to an array of binary values, which are then sent to the instruction memory through UART. The compiler identifies syntax errors, which allows the programmer to write assembly code with ease, using our ISA.

In addition to the compiler, we also built a simulator software for ABRUTECH. The simulator can run the algorithm like the processor and show the values of registers and memory at each step, helping us debug an algorithm fast and remotely, without repeatedly loading it into the processor.

The system itself is implemented with hardware debugging features, such as the ability to run the processor either at 1 Hz clock frequency, 10 MHz clock frequency or through a manual clock provided by a push button. We are able to see the currently fetched instruction and currently retrieved data on 7 segment displays and LED bulbs.

Our processor is also free of major hardware vulnerabilities, such as spectre and meltdown since we did not have the time to implement the branch prediction and speculative execution modules.

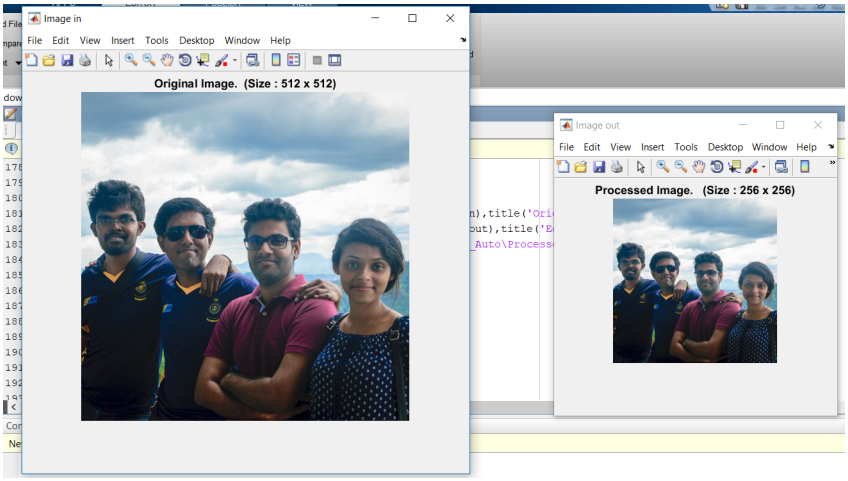


Figure 1.1: Downsample by any integer upto 16, upsample, detect-edges, apply custom filters using only about 30 lines of code

2 Instruction Set Architecture (ISA)

The ISA of ABRUTECH was designed with the objective of striking a delicate balance between different goals with mutual trade-offs. With only 16 instructions each of which execute in only 2.2 clocks cycles on average, our ISA and the complier allow the programmer to write programs quickly that take only 1.8 bytes of memory per instruction on average.

FEATURES

- Number of Instructions = 16
- Number of Microinstructions = 35
- Average number of clock cycles per instruction (Excluding Fetch cycle) = 2.2
- Average number of clock cycles per instruction (Including Fetch cycle) = 3.87
- Average memory used per instruction (with operand and parameter) = 1.8 bytes

Figure 2.1: Key Features

Our ISA was designed to incorporate advantageous features from both RISC and CISC instruction sets. As in RISC, each operation is designed to perform a specific task, especially the load and store operations are maintained strictly separate. However, unlike RISC, not all instructions are of equal length, but are either 1, 2 or 4 bytes long. Certain instructions are encoded, resulting in high code density. This allows building smaller programs that use the limited instruction memory efficiently while also allowing faster execution. However, this is balanced with maintaining moderate hardware level complexity in our system architecture.

2.1 Design Goals and Features

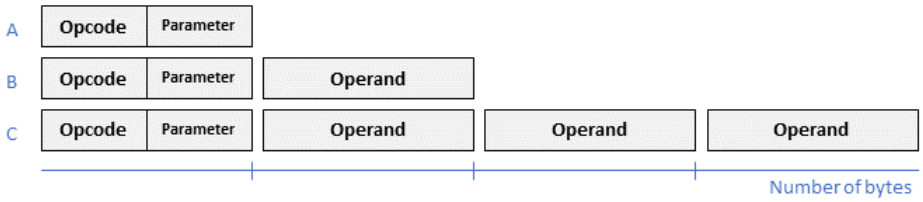
2.1.1 Minimizing the number of instructions in ISA

Inspired from RISC, we decided to have replace groups of instructions with single instructions that take parameters and operands to decide what should be done. With only 16 instructions,

Figure 2.2: Code examples to portray the unique ISA design

Sample code given to the compiler. Each line corresponds to one byte in memory		**Output from Compiler**
`LOAD : FROM_MAT`	`# This is a comment`	81
`RSET`		128
`[all]`		255
`COPY`	`# Can write to many`	112
`[AC -> MDDR, K0, G0]`	`# registers at once`	22
`INCR`	`# Can increment many`	160
`[K0, AC, ART, ARG]`	`# registers at once`	102
`$loop1 STAC : TO_MAT`	`# Loops names are replaced`	98
`JUMP : NZ_ART`	`# with line numbers`	150
`[loop1]`	`# by the compiler`	7

Figure 2.3: Three types of instructions, based on number of operands



the structure of our instructions are as shown.

Figure 2.4: Instruction types and examples

Type of Instruction	Length (bytes)	Example	Example binary
A	1	Without Parameter:	
		• TOGL • NOOP	1111 0000 0001 0000
		With Parameter:	
		• LOAD : FROM_MAT • STAC : TO_MAT	0101 0001 0110 0010
B	2	Without Parameter:	
		• COPY [AC -> MDDR, G0, K0] • INCR [ART, AC, K0]	0111 0000 000 10110 1010 0000 0110 0010
		With Parameter:	
		• JUMP : Z_AC [210]	1001 0010 1101 0010
C	4	• LADD : TO_AR [262142]	0100 0111 0000 0011 1111 1111 1111 1110

2.1.2 Minimizing the number of lines of a program

We believe, an advanced ISA should enable the programmer to program intuitively and do things faster with shorter code. For example, COPY instruction in our ISA is used to copy data from any register to any or to all registers at a single step, eliminating the need for multiple MOVE instructions. Similarly, INCR (increment), DECR (decrement), RSET (reset) instructions can take multiple register names as operands, which are bundled within one 8-bit word as operand after compilation.

- COPY [K0 ← MDDR] : Copies MDDR into K0
- COPY [all ← AC] : Copies AC into all registers at once
- INCR [K0, AC, ART] : Increment all these registers at once
- JUMP: Z_AC [11] : Jump to 11 if AC is zero

2.1.3 Reducing the number of total number of clock cycles needed for the program to run

The compiler encodes multiple operands (names of many registers that can be incremented at once) into one operands 8-bit integer, which is decoded by a demultiplexer module called OPR

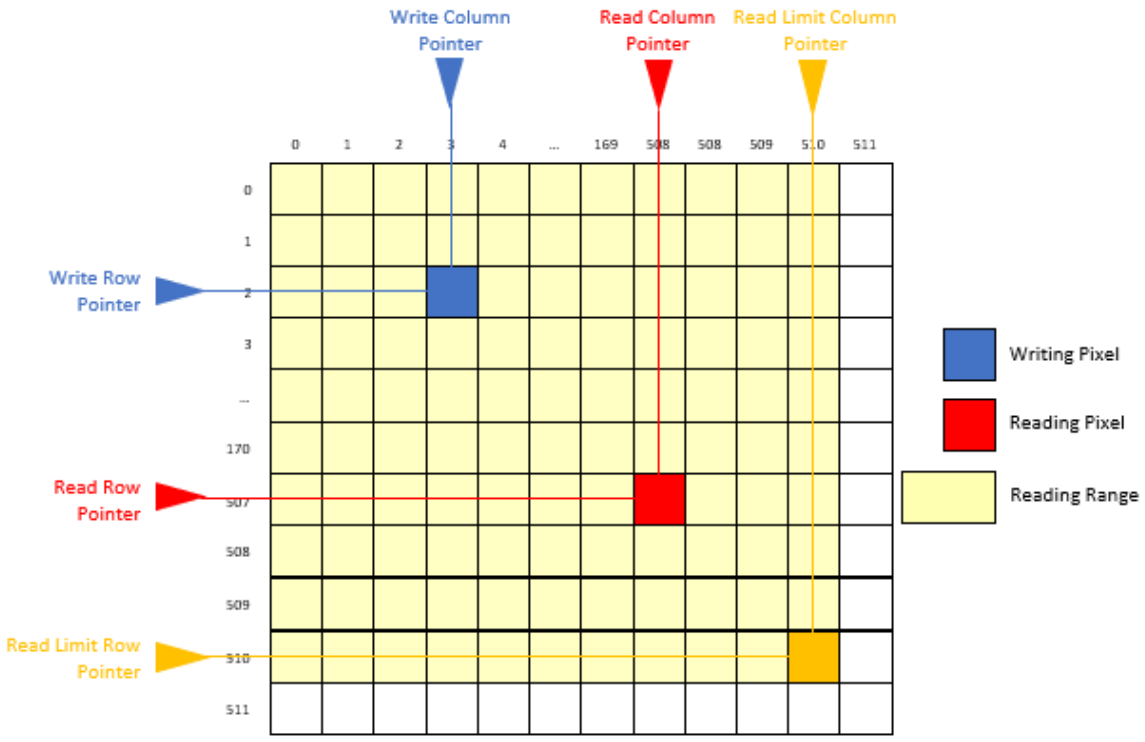
(Operand Router) in our architecture. Similarly, the opcode (eg: JUMP) and the parameter (eg: Z_AC, that defines what should be checked before jump) are encoded into one 8-bit opcode which is decoded by another demultiplexer called PRM (Parameter Router).
This allows the programmer to write code faster, in an intuitive way which actually executes inside the processor at a single clock cycle.

2.1.4 Matrix Manipulation

Our processor is designed with Matrix manipulation in mind. The ISA allows the programmer to optionally navigate a 262,144-words deep memory as a 512 x 512 matrix, both row-wise and column-wise, while detecting overflows at any pre-set point.

The address bus of the memory consists of 18 bits. Of this, the first 9 bits correspond to the column number and the last 9 bits correspond to the row number. By adjusting these two 9-bit halves of the address bus, the programmer is able to navigate to any pixel and to perform read, write operations along the rows (by incrementing last 9 bits) and along the columns (by incrementing first 9 bits).

Figure 2.5: Horizontal and Vertical Pointers for Matrix Manipulation



We introduce the concept of two address pointers as shown in the above figure: the read_pointer and the write_pointer, which point to the address where data can be read from and written to, at the moment. This allows the programmer to read from one location and write into another location, while incrementing the row-column halves of read and write pointers independently.

2.1.5 Linear Decomposition of 2D Algorithms

Many 2D image processing operations such as downsampling, upsampling and convolution by important kernels (gaussian kernel, edge detection kernels) can be decomposed into linear operations. To exploit this and build shorter and faster code, our ISA also allows the programmer

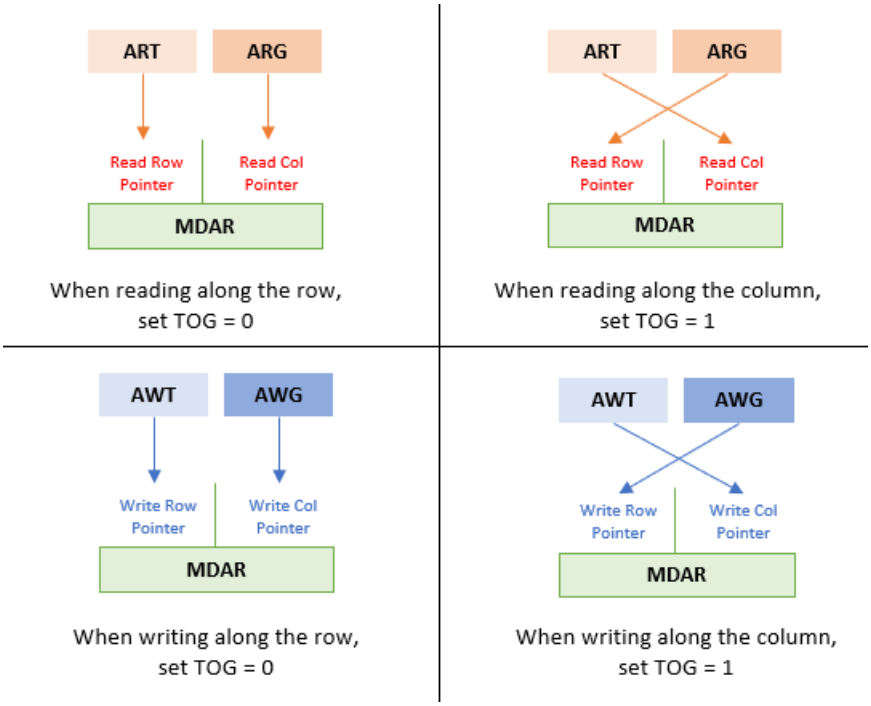
to do the same operation both row-wise and then column-wise, without code replication. We introduce the concept of row-column toggling to allow the programmer perform this.

This is achieved by allowing the programmer to change the order in which two 9 but registers are connected to the Address register MDAR, based on a toggle register. The toggle register (ZT) placed inside the address maker determines the row vs column navigation. The following table explains this design concisely.

Toggle	When Reading	When Writing	
ZT = 0	MDAR = {ARG, ART}	MDAR = {AWG, AWT}	Navigating along the column
ZT = 1	MDAR = {ART, ARG}	MDAR = {AWT, AWG}	Navigating along the row

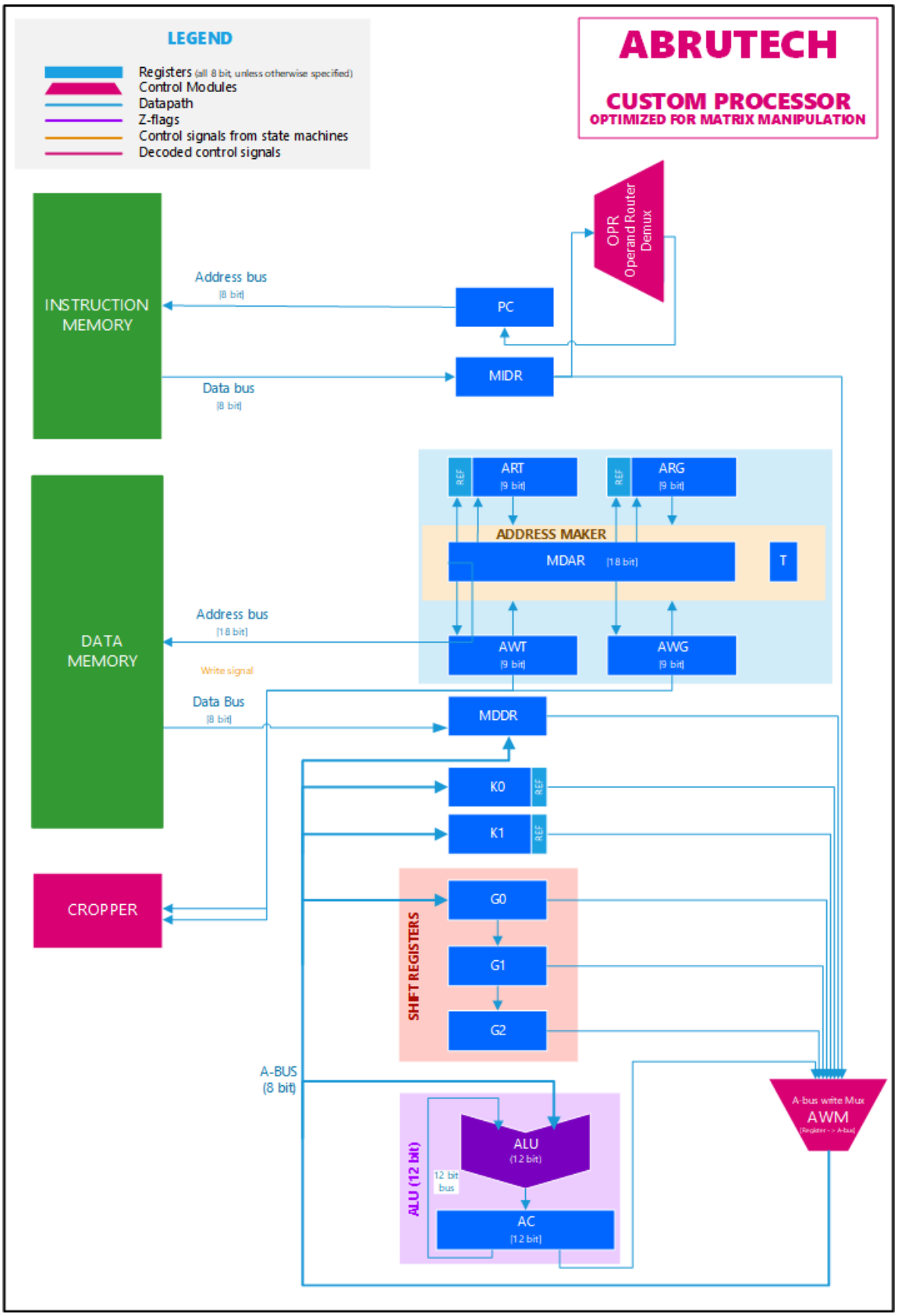
In the program, when reading, the programmer can keep incrementing a 9-bit register ARG and when that overflows, increment another 9-bit register ART. When performing this with TOG = 0, ART and ARG are connected to the column (first half) and row (second half) of address respectively. This allows the programmer to read along the row. Once done, the programmer can loop to the top of the code and set TOG = 1. Now, ART and ARG get connected to row (second half) and column (first half) of address. Now the same code executes, while performing the operation along the columns instead.

Figure 2.6: How Z_TOGL flag changes the address composition



2.2 Registers and Datapath

REGISTERS (13)						
REGISTER		SIZE	IDR	A	NAME	DESCRIPTION
INSTRUCTION	PC	8			Program Counter	Connected to the address bus. Ins-memory sends data to data bus based on this address
	STATE	8			State	Stores current state (micro instruction)
	PARAM	8			Paramteter	Stores parameter of last operand
	MIDR	8		W	Memory Instruction Data Register	Ins.Data - Data (opcode/ operand) from i_mem comes into here
DATA	MDAR	18	*		Memory Data Address register	Address Register for d_mem
	MDDR	8		RW	Memory Data Data register	Data reg for d_mem
	ART	9	*		Constant Reading Address	Constant half of address to be read
	ARG	9	*		Changing Reading Address	Changing half of address to be read
	AWT	9	*		Constant Writing Address	Constant half of address to be written
	AWG	9	*		Changing Writing Address	Changing half of address to be written
CALC	AC	12	*	RW	Accumulator	Stores result of calculations. **NOTE:** 12 bit allows to add upto 16 8-bit values without overflow error
	K0	8	*	RW	K0 - Loop register	Loop Registers (Can be used as GPR) When writing for the first time, stores that value the the main register as well as in a hidden reference register. When writing furthur, if current value = reference, raises the zero flag. Register operation can be reset by RSET
	K1		*	RW	K1 - Loop register	
	G0	8		RW	G0 - Shift register	Can write only to G0 from A bus. But all three can be read to A bus. When writing to G0, values shift as: G2 <- G1 <- G0
	G1			W	G2 - Shift register	
	G2			W	G3 - Shift register	
FLAG	Z	1				Zero Flag (if AC = 0)
	ZT	1				Manual flag, Togglable register 1: Read along row: MDAR = A_T,A_G 0: Read along col: MDAR = A_G,A_T
	ZRG	1				!(ARG = 0): still reading by row
	ZRT	1				!(ART = 0): still reading by col
	ZK0	1				Zero if K0 = K0_REF
	ZK1	1				Zero if K1 = K1_REF


LEGEND
Registers (all 8 bit, unless otherwise specified)
Control Modules
Datapath
Z-flags
Control signals from state machines
Decoded control signals
ABRUTECH
CUSTOM PROCESSOR
OPTIMIZED FOR MATRIX MANIPULATION
INSTRUCTION MEMORY
Address bus
[8 bit]
Data bus
[8 bit]
PC
MIDR
OPR
Operand Router
Demux
DATA MEMORY
Address bus
[18 bit]
Write signal
Data Bus
[8 bit]
REF
ART
[9 bit]
ARG
[9 bit]
ADDRESS MAKER
MDAR [18 bit]
T
AWT
[9 bit]
AWG
[9 bit]
MDDR
K0
K1
CROPPER
SHIFT REGISTERS
G0
G1
G2
A-BUS
(8 bit)
ALU (12 bit)
ALU
(12 bit)
12 bit
bus
AC
[12 bit]
A-bus write Mux
AWM

2.3 The Instruction Set

2.3.1 COPY

COPY instruction copies an 8-bit data from any one of the registers connected to A-bus into any number of registers connected in the A-bus at once, as shown. Details about the FROM, TO registers are encoded into a single 8-bit operand and decoded using OPR. This instruction was designed as an elegant alternative to numerous move statements in the traditional style: MOV_AC_MDDR, MOV_MDDR_AC, MOV_AC_K1...etc. As a result, it reduces the number of lines in a program (hence reduce instruction memory usage) and number of instructions in the ISA at once.

- Type : B
- Number of bytes : 2
- Possible parameters : none
- No. of clock cycles : 2
- Syntax and examples :

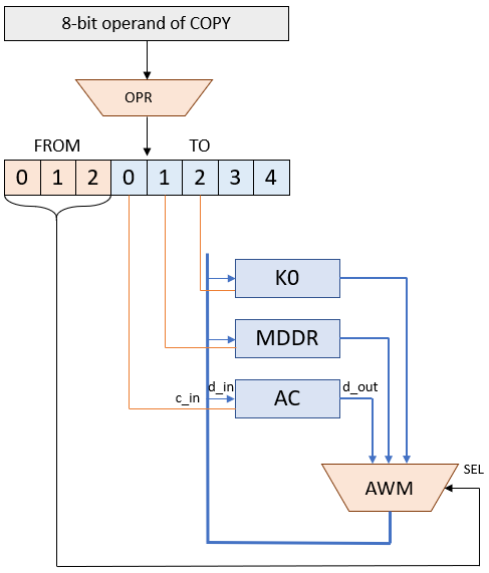
```
COPY
    [from_register -> list_of_to_registers]
COPY
    [list_of_to_registers <- from_register]

COPY
    [K0 -> all]
COPY
    [AC -> K0, K1, MDDR]
COPY
    [K0, K1, MDDR <- AC]
```

The compiler parses the line and converts it into two bytes of instruction data as follows. The first byte is the opcode (112) and to generate the next byte, the registers are coded as:

FROM Registers

The first 3 bits of the 8 bit operand of COPY consists of the key number of the FROM register. This is the possible set of registers that can write data into the A-bus. The key values given to these registers (as below) are directly routed to the 3 SEL bits of A-bus-Write-Multiplexor (AWM) though Operand Router (OPR).



0-AC
1-MDDR
2-K0
3-K1
4-G0
5-G1
6-G2
7-MIDR

TO Registers

The last 5 bits of the 8 bit operand of COPY consists of the key number of the TO register. This is the possible set of registers that can read data from the A-bus. The key values given to these registers (as below) sent to the c_in (data in control) pins of these 5 registers though Operand Router (OPR). This way, we can write into any of the 5 registers at once.

0-AC
1-MDDR
2-K0
3-K1
4-G0

COPY on Loop Registers

K0 and K1 are special type of loop registers (as described in the module section. When some value is COPY-ied into them for the first time in a program, the same value is also written into their respective REF registers. Afterwards, whenever a new value is COPY-ied, the value of loop registers will change, but their REF will stay constant. The Z flags of these registers turn HIGH when the value stored in the register becomes equal to the REF value.

To change the value of a REF register in the middle of a program, the programmer needs to RSET a loop register, which sets both its value and its REF value to zero and when a new value is COPY-ied afterwards, it gets written into both the loop register and its REF register

```
1   LODK
2       [5]
3   COPY
4       [AC -> K0]
5   RSET
6       [AC]
7   COPY
8       [AC -> K0]
9   $loop    INCR
10                [K0]
11  JUMP: NZ_K0
12      [loop]
```

(a) Counting K0 up from 0 to 5

```
1   RSET
2       [AC, K0]
3   COPY
4       [AC -> K0]
5   LODK
6       [5]
7   COPY
8       [AC -> K0]
9   $loop    DECR
10                [K0]
11  JUMP: NZ_K0
12      [loop]
```

(b) Counting K0 down from 5 to 0

COPY on Shift Registers

G0, G1 and G2 are shift registers. When a value is COPY-ied into G0, the value in G0 automatically shifts (moves) into G1 and that in G1 moves into G2, with G2 value being discarded. This is highly useful when performing linear convolution (into which most major 2D convolutions can be decomposed).

2.3.2 INCR, DECR, RSET

These three instructions are used to increment, decrement or reset any combination of the following 8 registers at once. These were designed as alternatives to INC_AC, INC_K0... instructions in the traditional ISAs and they help to reduce the number of lines of programs and the number of instructions in the ISA.

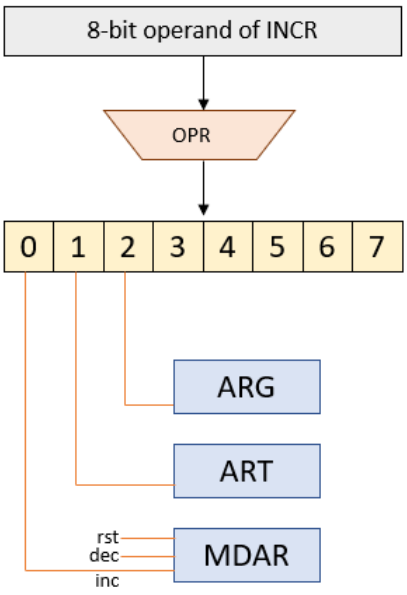
- Type : B
- Number of bytes : 2
- Possible parameters : none
- No. of clock cycles : 2
- Syntax and examples :

```
INCR
    [list_of_registers]
DECR
    [list_of_registers]
RSET
    [list_of_registers]
```

```
INC
    [all]
DECR
    [AC, K0, K1]
RSET
    [K0, K1, AC]
```

The compiler parses the list of operands and converts it into a single 8-bit number as follows:

0-MDAR
1-ART
2-ARG
3-AWT
4-AWG
5-AC
6-K0
7-K1



RSET operation on Loop registers (K0,K1) makes both their register values and their REF values zero and returns the register to the initial state. When a value is COPY-ied into one of them after RSET, that value will be written to both the register and the respective REF register.

2.3.3 JUMP

JUMP instruction is used for branching and looping. It takes one of the following 8 parameters within the latter 4 bits of its 8-bit opcode and allowing the programmer to execute 8 types of jump. The operand that follows is the address within the instruction memory (8 bits) to which jump should be made.

- Type : B
- Number of bytes : 2
- Possible parameters : 9
- No. of clock cycles : 2
- Syntax and examples :

```
JUMP: parameter
    [address]

JUMP: Z_AC        # If AC = 0
    [100]
JUMP: NZ_AC       # If AC != 0
    [100]
JUMP: jump        # Unconditional Jump
    [100]
```

Parameters List:

1-J
2-Z_AC
3-NZ_AC
4-Z_TOG
5-NZ_ARG
6-NZ_ART
7-NZ_K0
8-NZ_K1

The 4 bit parameter and the 8 bit operand of Jump are both routed to the Jump Selector (JMP) module by Parameter Router (PRM) and Operand Router (OPR) respectively. Jump Selector decided whether or not to make the jump, based on the parameter and the 6 different flag signals from 6 registers. The decision is given as the output of JMP module into the c_in (data in control) pin of PC. The OPR routes the address (operand) into the d_in (data_input) of PC register. Hence, if JMP decides to jump, the value in operand will be written into PC.

A Trick for Non-Straightforward Jumps

```
61      JUMP: NZ_K0
62            [65]
63      JUMP: jump
64            [100]
65
```

One could notice that there is no obvious way for the programmer to make a jump if K0 is zero. This is because, the parameters are designed for the most common jump types. However, the programmer can use a clever trick such as the following one to jump using such conditions. Here, when K0 is not zero, the first jump on line 61 is activated, sending the control to line 65, continuing with the rest of the program (as if nothing happened). But when K0 is zero, the jump on line 61 fails and the control goes to the next line (line 63) where an unconditional jump awaits. It sends the control to line 100. The net effect of this trick is, a JUMP is made to line 100 when K0 is zero.

2.3.4 LOAD

- Type : B
- Number of bytes : 2
- Possible parameters : 2
- No. of clock cycles : 3
- Syntax and examples :

```
LOAD: parameter
    [address]

LOAD: FROM_ADR    # Treats memory as single array
    [100]
LOAD: FROM_MAT    # Treats memory as a matrix
[100]
```

Loads data from data memory into MDDR. Takes one of two parameters. With parameter: FROM_ADR, it simply loads the value that corresponds to the current address in MDAR. This is useful for the generic purposes, when we want to treat the memory as a single long array, when we increment the address one by one from 0 to 262143 (without matrix navigation) and read values.

With parameter: FROM_MAT, generates an address from the Matrix read registers: ART and ARG according to TOGL (see 2.6), fills MDAR register and then loads the value corresponding to that address. This is useful for the image processing purposes, since the memory is treated as a 512 x 512 square matrix, which can be navigated row-wise and column-wise by independently manipulating ART, ARG and TOGL registers.

2.3.5 STAC

- Type : B
- Number of bytes : 2
- Possible parameters : 2
- No. of clock cycles : 1
- Syntax and examples :

```
STAC: parameter
    [address]

STAC: TO_ADR# Treats memory as single array
    [100]
STAC: TO_MAT# Treats memory as a matrix
    [100]
```

Stores data into data memory from AC. Takes one of two parameters. With parameter: TO_ADR, it simply stores the value at the location that corresponds to the current address in MDAR. This is useful for the generic purposes, when we want to treat the memory as a single long array, when we increment the address one by one from 0 to 262143 (without matrix navigation) and write values.

With parameter: TO_MAT it, generates an address from the Matrix write registers: AWT and AWG according to TOGL (see 2.6), fills MDAR register and then stores the value to location corresponding to that address. This is useful for the image processing purposes, since the memory is treated as a 512 x 512 square matrix, which can be navigated row-wise and column-wise by independently manipulating AWT, AWG and TOGL registers.

2.3.6 ADD, SUBT

Adds or subtracts the value in the register from AC and stores result into AC. The name of the register is taken as the parameter. The parameter router (PRM) routes the last 3 bits of the 4-bit parameter into A-bus-Write-Mux (AWM), which releases the value of the appropriate register into A-bus.

- Type : A
- Number of bytes : 1
- Possible parameters : 8
- No. of clock cycles : 1
- Syntax and examples :

```
ADD: register_name
SUBT: register_name

ADD: MDDR
SUBT: K0
```

SUBT performs absolute subtraction. The output of SUBT is always positive (absolute value) of the result of subtraction. For example, $5 - 3 = 2$ and $3 - 5 = 2$. The ALU was designed this way for the implantation of edge detection algorithms. If not, an edge detection operation such as 255-232 will result in an overflow (when represented in our unsigned 8 bits), which is inappropriate for such operation.

2.3.7 MUL, DIV

Multiplies or divides the value in AC by the given "constant" (Note: not by a register's value). It was designed this way so that the multiplicand or the divisor are not stored in the registers, but stored in the instruction memory.

- Type : B
- Number of bytes : 2
- Possible parameters : none
- No. of clock cycles : 2
- Syntax and examples :

```
MUL
    [multiplicand]
DIV
    [divisor]

MUL
    [3]
DIV
    [5]
```

The resulting value of division is rounded-off to nearest integer (unlike truncation in traditional architectures). This is to make sure no errors are obtained due to truncation (or flooring, which usually happens in ALUs) during division. Such division is performed using the following formula:

$$\textbf{rounded}\left(\frac{a}{b}\right) = \left\lfloor \frac{a}{b} + \left\lfloor \frac{b}{2} \right\rfloor \right\rfloor$$

2.3.8 LODK

Loads the operand into AC. Used to copy a necessary constant into the registers.

- Type : B
- Number of bytes : 2
- Possible parameters : none
- No. of clock cycles : 2
- Syntax and examples :

```
LODK
    [constant_to_be_loaded]

LODK
    [255]
```

2.3.9 LADD

Loads an 18 bit address from 3 consecutive 8-bit operands into the registers.

- Type : C
- Number of bytes : 4
- Possible parameters : 4
- No. of clock cycles : 7
- Syntax and examples :

```
LODK
    [constant_to_be_loaded]

LODK
    [255]
```

Parameters:

1. TO_MDAR

Loads 18-bit address into 18-bit register MDAR directly

2. TO_AR

Loads 18-bit address into 9-bit registers ARG and ART, as specified by Z_TOG (see part 5 in Design Goals and Features). Used when it is necessary to move the read pointer to an arbitrary location.

3. TO_AW

Loads 18-bit address into 9-bit registers AWG and AWT, as specified by Z_TOG (see part 5 in Design Goals and Features). Used when it is necessary to move the write pointer to an arbitrary location.

4. TO_AR_REF

Loads 18-bit address into the reference registers of ARG and ART. Initially, these reference registers are set to zero. Then, the flags Z_ARG and Z_ART turn high when ARG and ART are zero, respectively. However, after changing reference registers, Z_ARG and Z_ART become high when ART = ART_REF and ARG = ARG_REF respectively. This is used to set an arbitrary reading range in the matrix (yellow region in figure 2.5).

2.4 ISA: Summery

	OPCODE	Op	BIN	PARAMETERS	EXAMPLE	DESCRIPTION	CLK
GENERAL	END	-	0		END	Make processor idle Given at the end of the program	1
	NOOP	-	16		NOOP	No operation	1
	COPY	RW	112		COPY [K0 -> AC, G0]	Copy value from any register to one, many or all registers in A bus **NOTE:** Copying to G0 shifts the register banks Copying to loop registers would write into their reference on the first time only	2
	INCR	I	160		INCR [AC, MDDR, K0]	Increment one, many or all incrementable registers by one	2
	DECR	I	176		DECR [AC, MDDR, K0]	Decrement one, many or all decrementable registers by one	2
	RSET	I	128		RSET [AC, MDDR, K0]	Reset one, many or all decrementable registers **NOTE:** Resetting the Loop registers will reset their refereces also	2
	LADD	3A	64	6-TO_MDAR 7-TO_AR 8-TO_AW 9-TO_AR_REF	LADD: TO_AR_REF [510]	Fill 18 bit address from 3 operands Param: 6- Fill address to MDAR 7- Fill to ART, ARG in order based on tog bit 8- Fill to AWT, AWG in order based on tog bit 9- Fill to ART, ARG reference registers in order based on tog bit	4
	LODK	K	48		LODK [255]	Load AC from const	2
	TOGL	-	240		TOGL	Toggle the ZT register to interchange row vs column wise operations	1
MEMORY	LOAD	-	80	0-FROM_ADR 1-FROM_MAT	LOAD: FROM_ADR	Load MDDR from memory **PARAMETERS:** 0 - Create read address from MDAR directly 1 - Create read address from matrix registers	1
	STAC	-	96	0-TO_ADR 2-TO_MAT	STAC: TO_MAT	Store value in AC to memory **PARAMETERS:** 0 - Create write address from MDAR directly 2 - Create write address from matrix registers	1

BRANCHING & LOOPING	JUMP	A	144	1-J 2-Z_AC 3-NZ_AC 4-Z_TOG 5-NZ_ARG 6-NZ_ART 7-NZ_K0 8-NZ_K1	JUMP: Z_AC [210]	Jump to instruction address **PARAMTERS:** 1 - Uncondtional jump. 2 - If (AC = 0) 3 - If (AC != 0) 4 - If (Toggle reg = 1) 5 - If (ARG != ARG_REF) 6 - If (ART != ARG_REF) 7 - If (K0 != K0_REF) 8 - If (K1 != K1_REF) Unconditional jump can be used to create other types of jumps	2
ARITHMATIC	ADD	RR	192	0-AC 1-MDDR 2-K0 3-K1 4-G0 5-G1 6-G2 7-MIDR	ADD : MDDR	Add given register to AC **PARAMTERS:** Register names	1
	SUBT	RR	208	0-AC 1-MDDR 2-K0 3-K1 4-G0 5-G1 6-G2 7-MIDR	SUBT : MDDR	Subtract register from AC and return the absolute value **PARAMTERS:** Register names	1
	DIV	K	224		DIV [20]	Divide AC by given value **NOTE:** Result is rounded off to the nearest integer (not truncated)	2
	MUL	K	32		MUL [20]	Multiply AC by given value	2

3 Mathematical Justification of Algorithm Design

Downsampling is a simple operation in digital signal processing. However, as we show in 3.0.1, the aliasing effect arises if the image is downsampled improperly. To avoid aliasing, the image is generally convolved with a kernel with low pass characteristics before downsampling. Hence, there are many algorithms that can be used for aliasing-free downsampling. However, when a kernel with an improper cut off frequency is used, the output image would either lack some major, required frequency component or would have unwanted frequency components which lead to aliasing. Therefore, the choice of the suitable kernel for smoothing is paramount in a precise downsampling process.

In this section, we present the mathematical argument of how the averaging filter of length k provides a remarkably suitable anti-aliasing effect for downsampling by any integer factor of k. We first derive the required cut off frequency to avoid aliasing when downsampling by a factor of k and then show that this filter provides the accurate 3 dB frequency for any value of k. We also show that the more commonly used 0.25 [1 2 1] filter, which is regarded as an approximation to Gaussian filter is only suitable for downsampling by a factor of three and is highly inaccurate for any other downsampling factor.

In addition to the low pass characteristics, this filter is easily implementable in a custom processor, since we simply need to add the consecutive values and divide, eliminating the need to scale individual vales or remember past and future values.

3.0.1 Aliasing effect and the required 3-db cut off frequency for downsampling

Consider a one dimensional signal $i_{in}[n]$. Downsampling by an integer factor of k is a process that scales the input variable as in the following expression:

$$i_{out}[n] = i_{in}[kn] \qquad ; \quad k \in \mathbb{Z}^+$$

In Fourier domain, the frequency spectrum expands in frequency axis by a factor of k as a result of downsampling:

$$i_{in}[n] \xrightarrow{DTFT} I_{in}(\omega)$$

$$i_{out}[n] = i_{in}[kn] \xrightarrow{DTFT} I_{out}(\Omega) = \frac{1}{k} I_{in}\left[\frac{\omega}{k}\right]$$

Hence, the frequencies present the range of $\omega \in \left[-\frac{\pi}{k}, \frac{\pi}{k}\right]$ will be expanded to take the full range of $\omega \in [\pi, \pi]$. During this process, frequencies of the range $\omega \in \left[-\frac{\pi}{k}, \pi\right]$ wrap around, causing aliasing. Hence, these frequencies need to be removed by a lowpass filter of cutoff frequency $\omega_c = \frac{\pi}{k}$.

$$\boxed{\text{Required } \omega_c = \frac{\pi}{k}}$$

3.0.2 Low pass characteristics of the averaging filter with length k

Downsampling by averaging with a filter length k is similar to first passing the signal through a a filter with impulse response $a[n]$ and then applying the downsampling operation.

$$a[n] = \begin{cases} \frac{1}{k}, & \text{if} \quad n \in [0, k-1] \\ 0 & \text{otherwise} \end{cases}$$

The frequency characteristics of this rectangular impulse response can be analyzed by DTFT as:

$$a[n] \xrightarrow{DTFT} A(\omega)$$

$$\begin{aligned} X(\omega) &= \sum_{n=-\infty}^{+\infty} h[n] \cdot e^{-j\omega n} \\ &= \sum_{n=0}^{k-1} \frac{1}{k} \cdot e^{-j\omega n} \\ &= \frac{1}{k} \cdot e^{-j\omega\left(\frac{k-1}{2}\right)} \cdot \frac{sin\left(\frac{k\omega}{2}\right)}{sin\left(\frac{\omega}{2}\right)} \end{aligned}$$

$$|X(\omega)| = \frac{1}{k} \cdot \frac{sin\left(\frac{k\omega}{2}\right)}{sin\left(\frac{\omega}{2}\right)}$$

Using this, we derive the 3-db cut off frequencies for k = 2,3,4 onwards:

$$|X(\omega_c)| = \frac{1}{\sqrt{2}}|X(\omega)| \implies \omega_c = 0.5\pi \quad \text{for k} = 2$$

k = downsampling factor	Required ω_c	ω_c from the averaging filter of length k
2	0.5000 π	0.5000 π
3	0.3300 π	0.3110 π
4	0.2500 π	0.2280π
5	0.2000 π	0.1803π
6	0.1667 π	0.1495 π
10	0.1000 π	0.0890 π
15	0.0667π	0.0592π

From the above table, it can be seen that the anti-aliasing low pass characteristics of the averaging filter of length k is remarkably suited for downsampling by a factor of k. It should also be noted that filter is in fact precisely suitable for a factor of 2.

3.0.3 The 3-db frequency of discrete approximation of Gaussian kernel

In image processing applications, it is a common practice to use the following 2D kernel or its corresponding decomposed 1D kernel for smoothing and removing high frequency components of the image.

$$h_{2D} = \frac{1}{16}\begin{bmatrix}1 & 2 & 1\\ 2 & 4 & 2\\ 1 & 2 & 1\end{bmatrix} \qquad h_{1D}[n] = \left\{\frac{1}{4}, \frac{1}{2}, \frac{1}{4}\right\}$$

Standard deviation of the Gaussian function approximated by the kernel

This kernel is considered as the discrete approximation to a Gaussian kernel. To find the cut off frequency of that Gaussian kernel, we first find the variance (standard deviation) of the closest Gaussian kernel. For this, we minimize the error between a general continuous Gaussian distribution $g_\sigma(t)$ with zero mean and the given kernel.

$$g_\sigma(t) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\Big(-\frac{t^2}{2\sigma^2}\Big)$$

$$\mathcal{E}(\sigma) = \sqrt{\Big(h[-1] - g_\sigma(-1)\Big)^2 + \Big(h[0] - g_\sigma(0)\Big)^2 + \Big(h[1] - g_\sigma(1)\Big)^2}$$

$$\frac{d}{d\sigma}\mathcal{E}(\sigma) = 0 \quad \Longrightarrow \quad \sigma = 0.8133$$

$$\frac{d^2}{d\sigma^2}\mathcal{E}(0.8133) > 0 \implies \text{local minum at } \sigma = 0.8133$$

Hence, the the closest standard deviation of the Gaussian function approximated by smoothing kernel is 0.8133.

Low pass characteristics of the Gaussian kernel

The Fourier transform of a continuous Gaussian pulse is also a Gaussian pulse with a standard deviation in frequency domain inversely proportional to that in time domain.

$$g_\sigma(t) \xrightarrow{\mathscr{F}} G_\sigma(f)$$

$$\text{where} \quad G_\sigma(f) = \exp\Big(-2\pi^2 f^2 \sigma^2\Big)$$

$$\text{with} \quad s^2 = \frac{1}{4\pi^2\sigma^2} \quad \text{and} \quad A = \sqrt{2\pi s^2}$$

To find the 3-db cut off frequency of this Gaussian pulse, we find the frequency where the power drops to half of the $G_s(0)$

$$\begin{aligned} G_s(f) &= A\frac{1}{\sqrt{2\pi s^2}} \exp\Big(-\frac{f^2}{2s^2}\Big) \\ &= A \cdot g_s(f) \end{aligned}$$

$$
\begin{aligned}
\frac{1}{\sqrt{2}} G_s(0) &= G_s(f_c) \\
\frac{1}{\sqrt{2}} &= exp\Big(- \frac{f^2}{2s^2} \Big) \\
f_c &= \sqrt{\ln 2} \cdot s \\
&= 0.6516 \qquad for \quad \sigma = 0.8133 \\
\omega_c &= 0.3258\pi \qquad for \quad \sigma = 0.8133
\end{aligned}
$$

Hence the 3-db cutoff frequency of the Gaussian function that was approximated by the kernel in 3.0.3 is 0.3258π. As per the required cut off frequency derived in 3.0.1, this is only suitable for downsampling by a factor of 3, not by 2. When downsampling by 2 after smoothing with this kernel, it tends to smooth too much, attenuating some frequencies that are actually required, causing a slight error.

Low pass characteristics of the kernel without approximating to Gaussian

The above method of approximating a discrete kernel to a continuous Gaussian may not be rigorous since it introduces a slight error of its own when approximating. Hence, in this section, we analyze the low cut off frequency of the discrete kernel itself by taking the discrete-time-fourier-transform (DTFT) of the standard smoothing kernel.

$$
h[n] = \Big\{ \frac{1}{4}, \frac{1}{2}, \frac{1}{4} \Big\}
$$

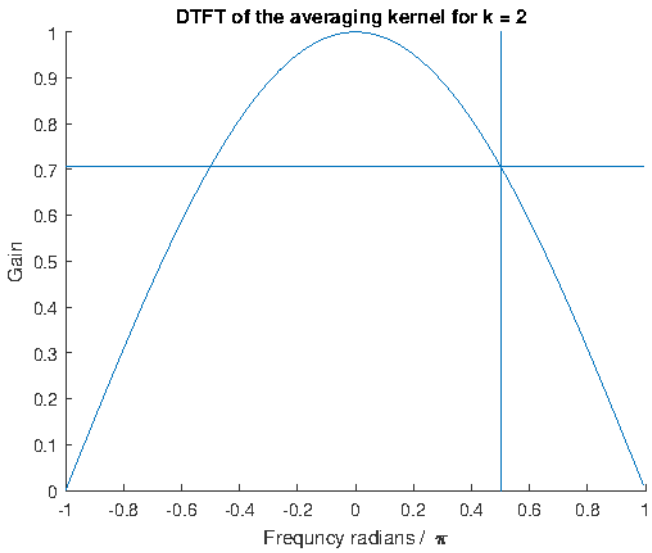
$$
h[n] \xrightarrow{DTFT} X(\omega)
$$

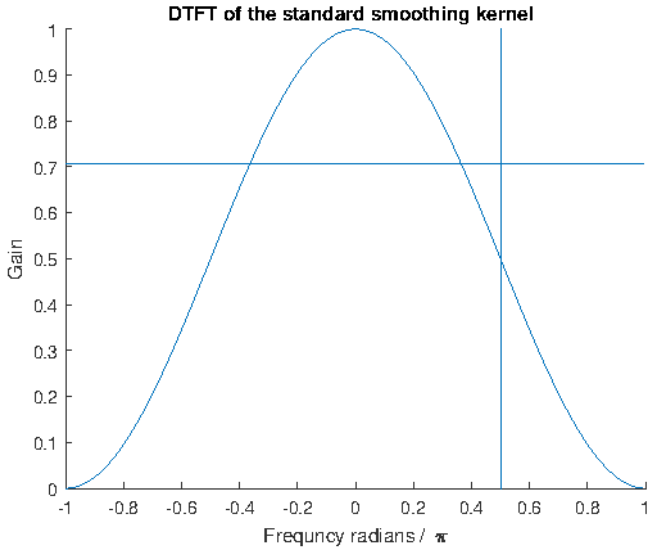
$$
\begin{aligned}
X(\omega) &= \sum_{n=-\infty}^{+\infty} h[n] \cdot e^{-j\omega n} \\
&= \frac{1}{4} e^{j\omega} + \frac{1}{2} + \frac{1}{4} e^{-j\omega} \\
&= \frac{1}{2} \Big(1 + cos(\omega) \Big) \quad \text{for} \quad \omega \in [-\pi, \pi]
\end{aligned}
$$

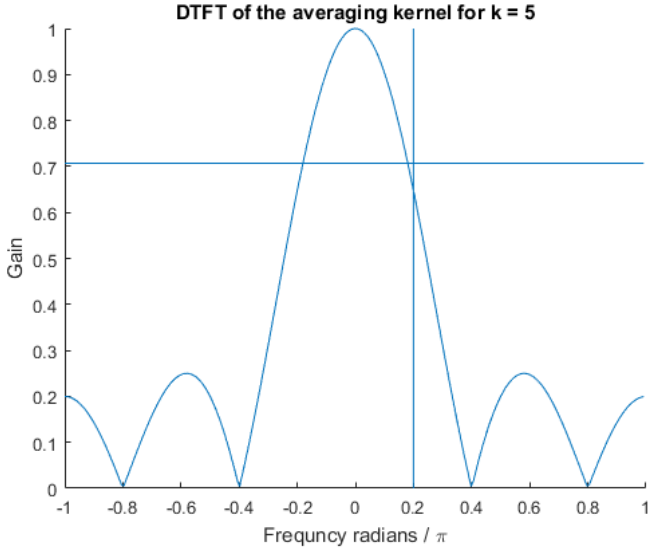
Then we proceed to derive the 3-db cutoff frequency of this kernel.

$$
X(\omega_c) = \frac{1}{\sqrt{2}} X(\omega) \implies \omega_c = 0.3641\pi
$$

Once again, we see that this kernel is suitable for downsampling by an integer of 3 only.


DTFT of the averaging kernel for k = 2
Gain
Frequncy radians / π

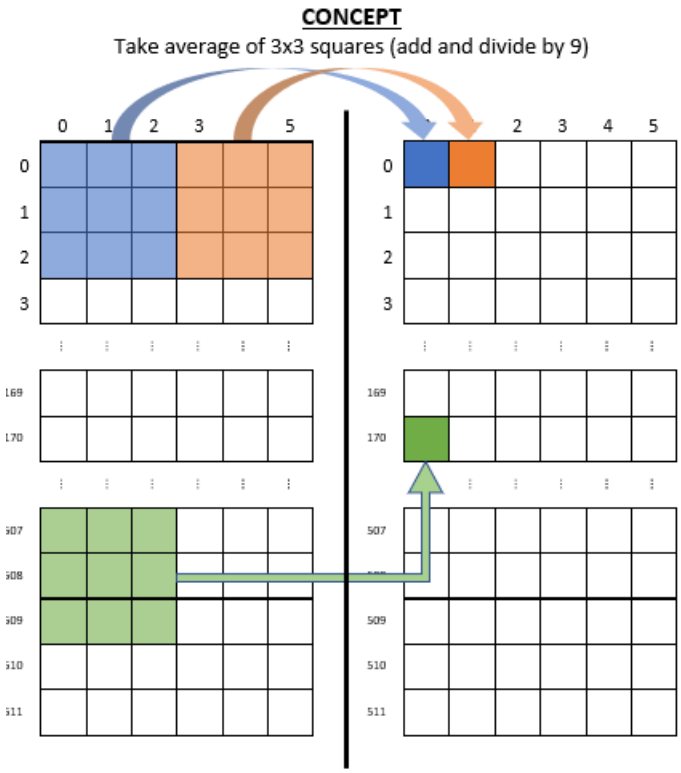

DTFT of the standard smoothing kernel
Gain
Frequncy radians / π


DTFT of the averaging kernel for k = 5
Gain
Frequncy radians / π

4 Algorithms

It was mathematically justified that the averaging algorithm is far superior to Gaussian smoothing algorithm in terms of preventing aliasing as well as in terms of reduced computational complexity.

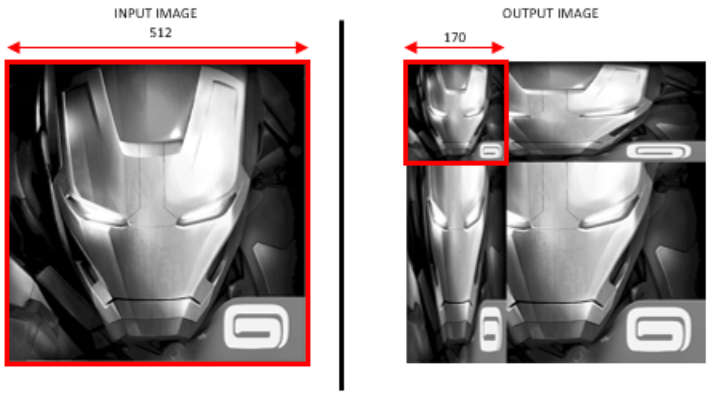
The averaging algorithm works in the following fashion. When downsampling by 3, from the input image, a square of 9 pixels are taken, and their average is stored into the first pixel. Then the next (non-overlapping) 3x3 square is chosen and its average is stored into the second pixel. This way, 172 x 172 pixels are populated. The last 2 rows and columns are omitted since, 512 is not properly divisible by 3 (remainder is 2). Note that we do not lose any necessary information in this method, unlike in traditional algorithms.



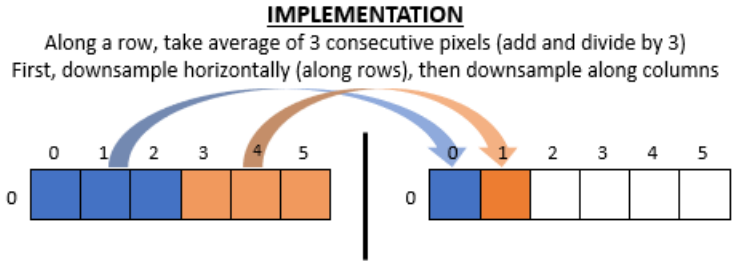
Two ways of implementing the averaging algorithm are discussed below, with their advantages and disadvantages. The Snake algorithm is better for downsampling by 2 and 3 and the Squeeze algorithm is better for integers from 4 to 15.

4.1 Downsampling: Squeeze Averaging Algorithm

Figure 4.1: Downsampling by 3 using Squeeze Algorithm



The squeeze algorithm (as we named it) produces the above output when downsampling by a factor of 3. The key intuition in this approach is 2D averaging operation is decomposable into linear averaging. That is, averaging every 3x3 square is mathematically identical to linearly averaging 3 pixels along the rows and then doing the same along the columns.



This approach makes use of our TOGL operation (see: 2.6) to avoid code replication, reducing the program size by half.

Disadvantages

However, the main disadvantage of this algorithm is the time taken (number of clock cycles) to execute it. From the output image, it can be observed that unwanted data has been written (STAC'ed) into the upper left and lower right regions. Also, since we read by rows and then by columns, each pixel is LOAD'ed twice, which is ineffective, given LOAD takes 3 clock cycles to execute. To eliminate these disadvantages, the Snake Algorithm was invented and used for downsampling by 2.

Advantages

The main advantage of this algorithm is the number of values it needs to store inside the processor during each iteration. Since we perform linear averaging here, to downsample by 3, we only need to add 3 numbers in the AC register, not 9. This means, this algorithm can be used to downsample by any integer upto 16 (since our AC is 12-bit and hence allows adding upto 16 8-bit numbers without overflow).

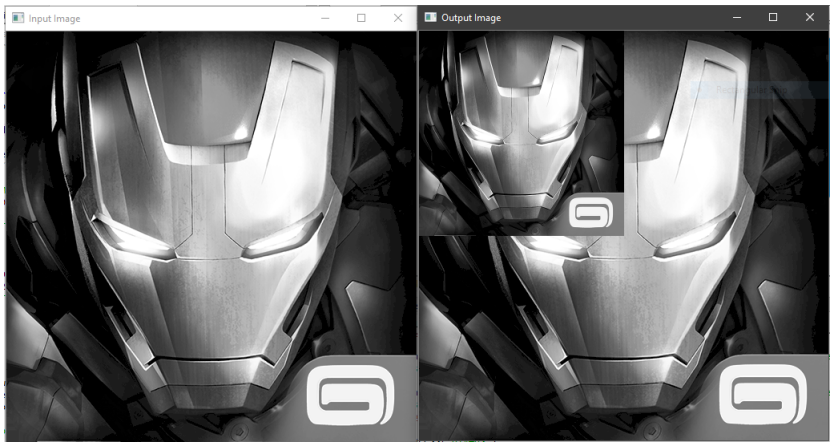
Program: Downsampling by 2 using Squeeze Algorithm

```
1   # Squeeze Downsampling by 2
2   # 29 lines(bytes)
3   $part2          RSET
4                       [all]
5                   COPY
6                       [AC -> all]
7                   TOGL
8       $rloop      RSET
9                       [AWG]
10                  $ploop  LOAD: FROM_MAT
11                          COPY
12                              [MDDR -> AC]
13                          INCR
14                              [ARG]
15                          LOAD: FROM_MAT
16                          ADD : MDDR
17                          DIV
18                              [2]
19                          STAC: TO_MAT
20                          INCR
21                              [ARG, AWG]
22                  JUMP: NZ_ARG
23                      [ploop]
24                  INCR
25                      [ART, AWT]
26      JUMP: NZ_ART
27          [rloop]
28  JUMP: Z_TOG
29      [part2]
30  LADD: TO_AW
31      [130815]
```

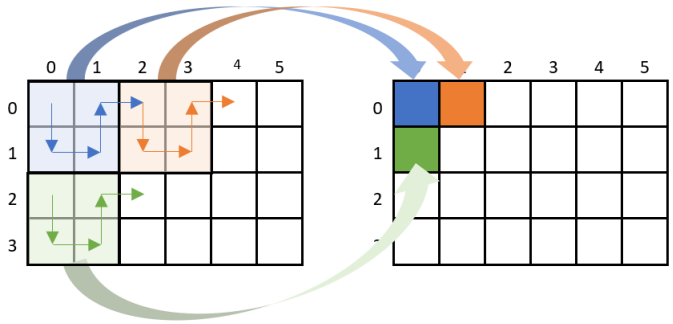
Program: Downsampling by Any Integer using Squeeze Algorithm

```
1   # Squeeze Downsampling ANY integer upto 16
2   # 40 lines(bytes)
3   RSET                                    # Downsample by any integer
4       [all]                               # Take f = downsampling factor
5   LADD : TO_AR_REF                        # and set the following values
6       [510,510]                           # [ 512 % f, 512 % f]
7
8
9   LODK
10      [15]                                # [f]
11  COPY
12      [AC -> K0]
13  $part2  TOGL
14          RSET
15              [AWG,AWT,ARG,ART]
16          $loop           RSET
17                              [AC]
18                          COPY
19                              [AC -> K0]
20                          $add    LOAD : FROM_MAT
21                                  ADD  : MDDR
22                                  INCR
23                                      [ARG, K0]
24                                  JUMP : NZ_K0
25                                      [add]
26                          DIV
27                              [15]        # [f]
28                          STAC : TO_MAT
29                          INCR
30                              [AWG]
31                  JUMP: NZ_ARG
32                      [loop]
33                  INCR
34                      [ART, AWT]
35                  RSET
36                      [AWG]
37          JUMP: NZ_ART
38              [loop]
39  JUMP: Z_TOG
40      [part2]
41  LADD: TO_AW
42      [34,34]                             # [ floor(512 / f), floor(512 /
        f)]
43
44
```

4.2 Snake Averaging Algorithm



The snake algorithm (as we named it) produces above output when downsampling by 2. It is implemented as shown in the following figure.



The read pointer (see: 2.5) starts at pixel (0,0), in the first iteration, it moves along the blue arrows (like a snake), reading the values from 4 pixels in the blue square and ends up on pixel (0,2). The average of those 4 pixels is written to (0,0). On second iteration, the pixels moves like a snake again, along orange arrows. After finishing the 256 big squares along the first row, the pointer moves to the pixel at (2,0) and follows the green arrows. This way, row by row, a 256 x 256 image is filled (overwritten) as output.

Disadvantages

The major disadvantage of this algorithm is the number of values it needs to store inside the processor during each iteration. Since we perform 2D averaging here, to downsample by 3, we need to add 9 numbers in the AC register. This means, this algorithm cannot be used to downsample by any integer above 3 (since our AC allows adding upto only 16 8-bit numbers). Downsampling by 4 requires adding 16 integers, which might overflow the AC.

Advantages

The advantage of this algorithm is the speed. Each pixel is read ONLY once and only the necessary region is overwritten, which minimizes time.

Program: Snake Algorithm

```
1   # Snake Downsampling by 2
2   # 32 lines(bytes)
3   RSET
4       [all]
5   $row                RSET
6                           [AWG]
7       $pixel          LOAD: FROM_MAT
8                       COPY
9                           [MDDR -> AC]
10                      INCR
11                          [ART]
12                      LOAD: FROM_MAT
13                      ADD : MDDR
14                      INCR
15                          [ARG]
16                      LOAD: FROM_MAT
17                      ADD : MDDR
18                      DECR
19                          [ART]
20                      LOAD: FROM_MAT
21                      ADD : MDDR
22                      DIV
23                          [4]
24                      STAC: TO_MAT
25                      INCR
26                          [ARG, AWG]
27              JUMP: NZ_ARG
28                  [pixel]
29              INCR
30                  [ART, AWT]
31              INCR
32                  [ART]
33  JUMP: NZ_ART
34      [row]
```

4.3 Gaussian Smoothing

We do not use Gaussian Smoothing to downsample an image (see: Mathematical Justification of Algorithm Design). However, we have implemented a shift register bank to make convolution operations efficient with our processor. This is demonstrated by our Gaussian Smoothing algorithm.

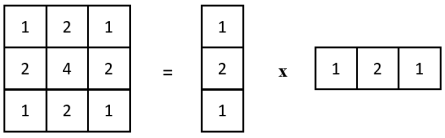
4.3.1 Problem with the Traditional Algorithms

In the traditional algorithms, there exists a problem of overwriting. Convolution (say by a kernel of size 3) requires the original values of previous and next pixels to compute the output of current pixel. This means, after overwriting the first pixel, to compute fr the second pixel, we would need the initial value of first pixel, which is lost by overwriting.

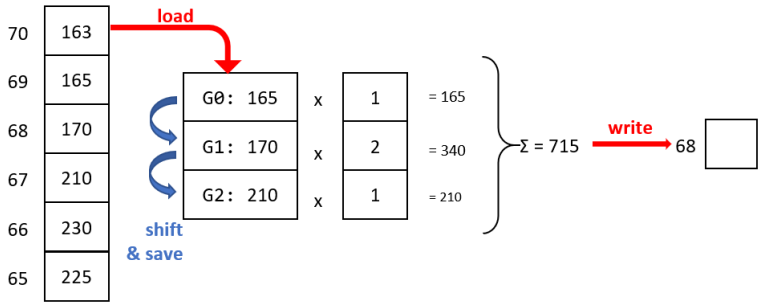
To avoid this issue, one can use two separate memories for input and output (hence no overwriting). This is very inefficient when considering hardware-wise. Instead, in the traditional algorithms, the average of a 3x3 square is calculated and put into the top right corner, to overcome this issue. However, this approach has the undesirable effect of shifting the image to the left and top by 1 pixel (i.e. The output is a shifted and smoothed version of input, one row and one column are lost).

4.3.2 Our solution

In our processor, by using a shift register bank, the original values of 3 consecutive pixels are stored right within the processor. This allows us the programmer to execute a far superior algorithm, that is several times faster. Our algorithm does not suffer either the overwriting problem or the missing rows problem encountered by traditional algorithms.



Since the discrete gaussian kernel is decomposable into simple linear kernels, convolution by the 3x3 kernel can be achieved by first convolving the image along the columns by the vertical 3x1 kernel and then convolving the result along the row by the horizontal 1x3 kernel.



This method can be done fast and without code replication, using the shift registers, address maker and toggle functionality (see 2.6) available in our architecture. As shown above, these features allow us to execute only one load and one store operation per one step of convolution, rather than requiring 3 read operations and 3 address shift operations if implemented otherwise.

Program: Gaussian Smoothing

```
1   # Gaussian Smoothing
2   # 48 lines(bytes)
3   RSET
4       [all]
5   $tLoop TOGL
6           $rLoop  LOAD : FROM_MAT
7                   COPY
8                       [MDDR -> G0]
9                   COPY
10                      [MDDR -> G0]
11                  INCR
12                      [ARG]
13                  LOAD : FROM_MAT
14                  COPY
15                      [MDDR -> G0,AC]
16                  ADD : G1
17                  ADD : G1
18                  ADD : G2
19                  DIV
20                      [4]
21                  INCR
22                      [AWG, ARG]
23                  $pLoop  LOAD : FROM_MAT
24                          COPY
25                              [MDDR -> G0, AC]
26                          ADD : G1
27                          ADD : G1
28                          ADD : G2
29                          DIV
30                              [4]
31                          STAC :TO_MAT
32                          INCR
33                              [AWG,ARG]
34                  JUMP: NZ_ARG
35                      [pLoop]
36                  COPY
37                      [G0 -> AC]
38                  ADD : G0
39                  ADD : G1
40                  DIV
41                      [4]
42                  STAC :TO_MAT
43                  INCR
44                      [AWG, AWT, ART]
45          JUMP: NZ_ART
46              [rLoop]
47  JUMP: Z_TOG
48      [tLoop]
49  DECR
50      [AWG, AWT]
```

4.4 Edge Detection Algorithms

Prewitt edge detection kernel, which is decomposable into a two linear kernels as below is used for our edge detection algorithm. The pixels are loaded into the shift registers and the next pixel's value is subtracted from the last pixel's value to get and the absolute value of the output (see: Instruction: SUBT) is stored into the current pixel location.

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix} \mathbf{x} \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}$$

Vertical edges and horizontal edges are detected independently (by loading two different programs) and the resulting images are combined in MATLAB to produce the final output.

Program: Vertical Edge Detection Algorithm

```
1   # Vertical Edge Detection
2   # 35 lines(bytes)
3   RSET
4       [all]
5           $row      LOAD: FROM_MAT
6                     COPY
7                         [MDDR -> G0]
8                     COPY
9                         [MDDR -> G0]
10                    INCR
11                        [ARG]
12                    LOAD : FROM_MAT
13                    COPY
14                        [MDDR -> AC, G0]
15                    SUBT : G2
16                    STAC : TO_MAT
17                    INCR
18                        [AWG, ARG]
19                    $pixel  LOAD: FROM_MAT
20                            COPY
21                                [MDDR -> AC, G0]
22                            SUBT : G2
23                            STAC : TO_MAT
24                            INCR
25                                [AWG, ARG]
26                    JUMP: NZ_ARG
27                        [pixel]
28                    COPY
29                        [G0 -> AC]
30                    SUBT : G2
31                    STAC : TO_MAT
32                    INCR
33                        [AWG, AWT, ART]
34          JUMP: NZ_ART
35                [row]
36  DECR
37      [AWG,AWT]
```

Program: Horizontal Edge Detection Algorithm

```
1   # Horizontal Edge Detection
2   # 36 lines(bytes)
3   TOGL
4   RSET
5       [all]
6           $row     LOAD: FROM_MAT
7                    COPY
8                        [MDDR -> G0]
9                    COPY
10                       [MDDR -> G0]
11                   INCR
12                       [ARG]
13                   LOAD : FROM_MAT
14                   COPY
15                       [MDDR -> AC, G0]
16                   SUBT : G2
17                   STAC : TO_MAT
18                   INCR
19                       [AWG, ARG]
20                   $pixel   LOAD: FROM_MAT
21                            COPY
22                                [MDDR -> AC, G0]
23                            SUBT : G2
24                            STAC : TO_MAT
25                            INCR
26                                [AWG, ARG]
27                   JUMP: NZ_ARG
28                       [pixel]
29                   COPY
30                       [G0 -> AC]
31                   SUBT : G2
32                   STAC : TO_MAT
33                   INCR
34                       [AWG, AWT, ART]
35           JUMP: NZ_ART
36               [row]
37  DECR
38      [AWG,AWT]
```

4.5 Upsampling Algorithms

Two types of upsampling operations (resizing a 256x256 image into 512x512) have been implemented using two different interpolation techniques.

4.5.1 Nearest Neighbor Interpolation

Nearest neighbor interpolation is the simplest (zeroth order) of upsampling algorithms. First the read pointer is moved to the end of the input image (255,255) and write pointer is moved to the end of the output image (511,511). Then the pointer moves backwards, picks up a pixel and writing it to both (511,511) and (510,510). In next iteration, (254,254) is written into (509,509) and (508,508). This way, the pointers move along the rows until whole image is covered.

Program: Nearest Neighbor Upsampling Algorithm

```
# Upsampling by 2
# Nearest Neighbour Interpolation
# 38 lines(bytes)
RSET
    [all]
$part2      TOGL
            RSET
                [all]
            LODK
                [255]
            COPY
                [AC -> K0]
            $row    RSET
                        [AC]
                    COPY
                        [AC -> K0]
                    $countLoop  INCR
                                    [K0,ARG]
                                JUMP : NZ_K0
                                    [countLoop]
                    INCR
                        [ARG]
                    $spacingPixel   DECR
                                        [ARG, AWG]
                                    LOAD: FROM_MAT
                                    COPY
                                        [MDDR -> AC]
                                    STAC :TO_MAT
                                    DECR
                                        [AWG]
                                    STAC :TO_MAT
                                    JUMP: NZ_ARG
                                        [spacingPixel]
                    INCR
                        [AWT, ART]
                    JUMP: NZ_ART
                        [row]
            RSET
                [all]
            JUMP : Z_TOG
                [part2]
```

4.5.2 Bilinear interpolation

Bilinear interpolation is a first order interpolation algorithm. To perform this, in our algorithm, we first perform nearest neighbor upsampling. Next, we start at (0,0) and traverse the image along rows, averaging two pixels at a time and writing the output into one of them. This way, we linearly interpolate along rows. Then, using our TOGL function, we do the same along the columns without code replication.

Bilinear interpolation can be done with our processor, without performing Nearest neighbor in the first place. The two operations have to be combined within one loop. Due to lack of time, we could not implement that kind of algorithms.

Program: Bilinear Upsampling Algorithm

```
# Upsampling by 2
# Nearest Neighbour Interpolation
# 58 lines(bytes)
 $part2      TOGL
             RSET
                 [all]
             LODK
                 [255]
             COPY
                 [AC -> K0]
             $row     RSET
                          [AC]
                      COPY
                          [AC -> K0]
                      $countLoop  INCR
                                      [K0, ARG]
                                  JUMP : NZ_K0
                                      [countLoop]
                      INCR
                          [ARG]
                      $spacingPixel   DECR
                                          [ARG, AWG]              # Now
                                          AWT = 0   AWG = 511; ART = 0
                                          ARG = 255
                                      LOAD: FROM_MAT
                                      COPY
                                          [MDDR -> AC]
                                      STAC :TO_MAT
                                      DECR
                                          [AWG]
                                      STAC :TO_MAT
                                      JUMP: NZ_ARG
                                          [spacingPixel]
                      INCR
                          [AWT, ART]                              # AWT,
                          ART = 1
                      JUMP: NZ_ART
                          [row]
             RSET
                 [all]
             $intpl  INCR
                          [ARG]
                      LOAD : FROM_MAT
                      COPY
                          [MDDR -> AC]
                      INCR
                          [ARG,AWG]
                      LOAD: FROM_MAT
                      ADD: MDDR
                      DIV
                          [2]
                      STAC : TO_MAT
                      INCR
                          [AWG]
                      JUMP : NZ_ARG
                        [intpl]
                 INCR
                     [ART, AWT]
                 JUMP: NZ_ART
                     [intpl]
             JUMP : Z_TOG
                 [part2]
DECR
[AWT,AWG]
```

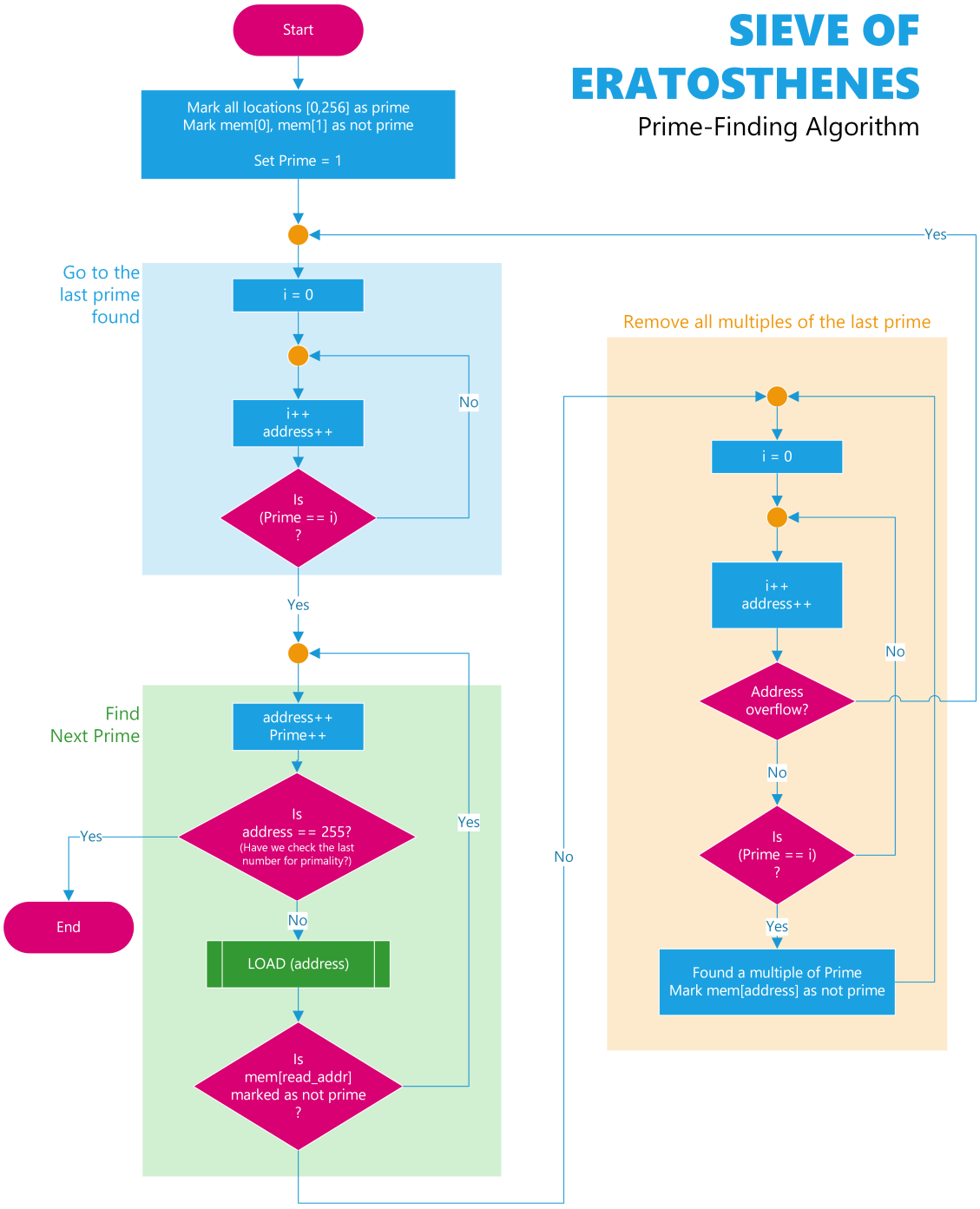
4.6 Custom Filter

We also implemented a multi-layer image processing algorithm to apply a custom image filter (one similar to Instagram filters) to an RGB image. Here, we produce an output image with 100% R, 75 %B and 10 %B to apply a filter. To run different codes on different layers of image, we pass a zero through the shift register bank and shift it every time a layer is processed. This way, we identify the current layer and apply the appropriate operation.

```
# Custom Filter (41 lines-bytes)
RSET
    [all]
INCR
    [AC]
COPY
    [AC-> G0]
$loop   LOAD: FROM_MAT
        COPY
            [G0->AC]
        JUMP: Z_AC
            [no_red]
        $no_red COPY
                    [G1->AC]
        JUMP: Z_AC
            [no_blue]
                        COPY
                            [MDDR -> AC]
                        DIV
                            [4]
                        MUL
                            [3]
                        STAC: TO_MAT
        $no_blue COPY
                    [G2->AC]
        JUMP: Z_AC
            [no_green]
                        COPY
                            [MDDR -> AC]
                        DIV
                            [10]
                        STAC: TO_MAT
        $no_green  INCR
                        [AWG,ARG]
        JUMP: NZ_ARG
            [loop]
INCR
    [AWT,ART]
JUMP: NZ_ART
    [loop]
DECR
    [AWT,AWG]
```

4.7 Prime Finding Algorithm

To prove that our processor, which is highly optimized for matrix manipulation, can do generic tasks, the 2000 year old prime finding algorithm: Sieve of Eratosthenes was implemented. Coding this algorithm in our assembly language was the hardest task in the project and took about 4 days. The algorithm consists of JUMPS that go in and out of multiple loops, which actually work and successfully end after finding the first 255 prime numbers and filling them to the memory.



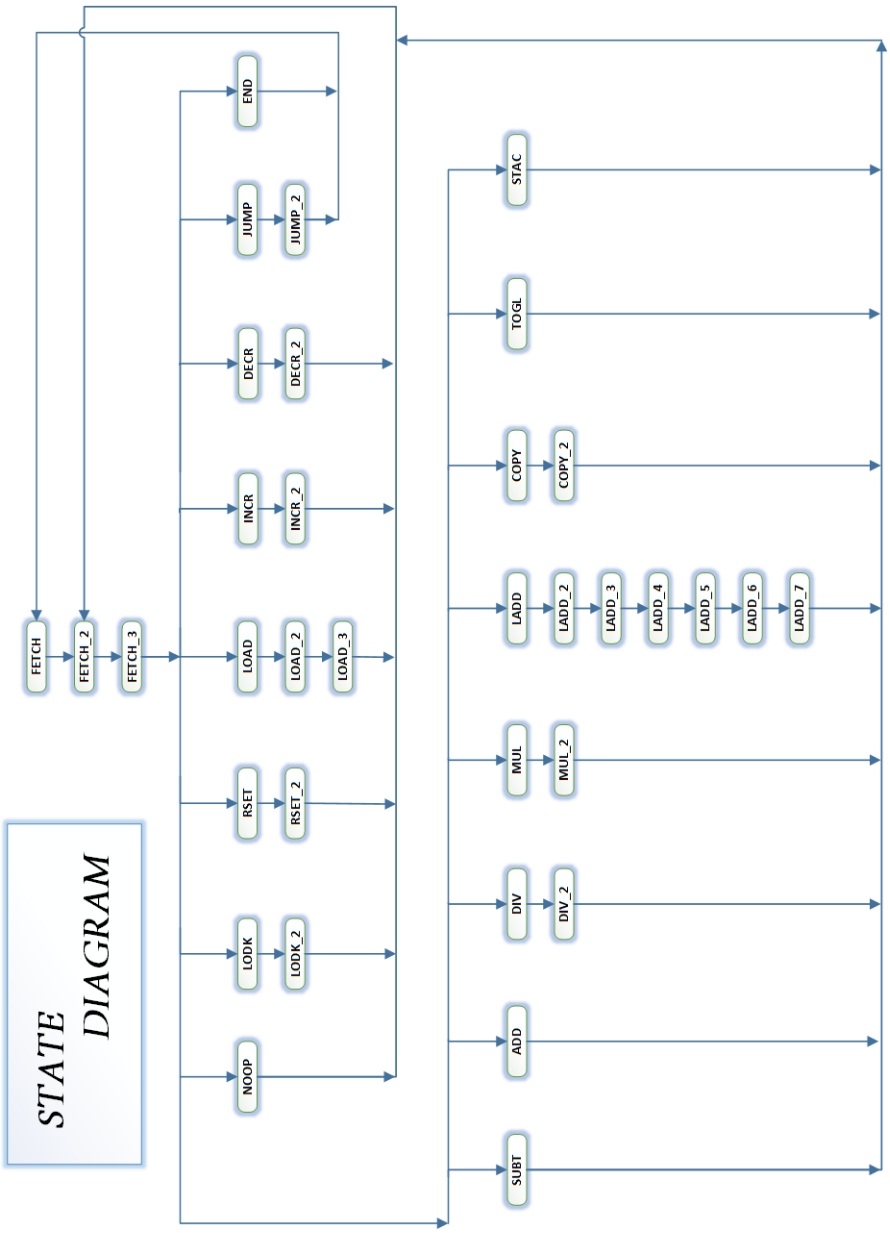
Program: Prime Finding Algorithm

```
# Prime Finding
# Eratosthenes Sieve
# 110 lines(bytes)
RSET
    [all]
TOGL
DECR
    [ART, AWT]
INCR
    [AC]
STAC : TO_MAT
INCR
    [AWG]
STAC : TO_MAT      # [0], [1] are set to 1
COPY
    [AC-> G0, K0] # G0,K0 = 1
RSET
    [AWG]
$gotoG0 RSET            ###################### GOTO CURRENT PRIME
            [AC]
        COPY
            [AC->K1]
        $checkP COPY
                    [K1 -> AC]
                INCR
                    [AC]
                INCR
                    [ARG, AWG]
                COPY
                    [AC -> K1]
                SUBT : G0
        JUMP: NZ_AC
            [checkP]     #------------------- ARG = AWG = G0 =
            (current prime)
        $nextP  INCR     ###################### Find Next Prime
                    [ARG, AWG]
                COPY
                    [G0 -> AC]
                INCR
                    [AC]
                COPY
                    [AC -> G0]
                LODK
                    [255]
                SUBT: G0
JUMP: Z_AC
    [end]
                LOAD: FROM_MAT
                COPY
                    [MDDR -> AC]
        JUMP: NZ_AC
            [nextP]     #---------------------- ARG = AWG = G0 = (next
            prime)
        $nxtMul RSET
                    [AC]
                COPY
                    [AC -> K1]
                $chkMul COPY
                            [K1-> AC]
                        INCR
                            [AC, ARG, AWG]
                    JUMP: NZ_ARG
                        [nD]
        JUMP: J
            [gotoG0]
                    $nD COPY
                            [AC->K1]
                        SUBT: G0
```

```
                    JUMP: NZ_AC
                        [chkMul]
                    INCR
                        [AC]
                    STAC : TO_MAT
            JUMP: J
                [nxtMul]
$end    CLAC
COPY
    [AC->all]
RSET
    [all]
DECR
    [ART]
LODK
    [255]
COPY
    [AC->K0]
RSET
    [AC]
INCR
    [AC]
COPY
    [AC->K1]
$count  INCR
            [ARG]
        LOAD :FROM_MAT
        COPY
            [MDDR->AC]
        JUMP: NZ_AC
            [nP]
                COPY
                    [K1->AC]
                STAC: TO_MAT
                INCR
                    [AWG]
    $nP COPY
            [K1->AC]
        INCR
            [AC]
        COPY
            [AC->K1]
        SUBT: K0
JUMP: NZ_AC
    [count]
LADD: TO_AW
    [511,511]
```

5 Architecture

5.1 State Diagram



5.2 Micro Instructions

OPCODE	u-ins	STEPS	CONTOL SIGNALS	N
END	END	STATE = IDLE done signal		0
NOOP	NOOP	STATE = FETCH2		16
FETCH	FETCH	read wait		1
	FETCH_2	MIDR <- M PC <- PC + 1	MEM = 001 PCI = 1	2
	FETCH_3	STATE <- {MIDR[7:4], 4'd0} PARAM <- MIDR[3:0]		3
LODK	LODK	MIDR <- M PC <- PC + 1	MEM = 001 PCI = 1	48
	LODK_2	AC <- MIDR STATE = FETCH2	AWM = 11, ALU = 1	49
LADD	LADD	MIDR <- M PC <- PC + 1	MEM = 001 PCI = 1	64
	LADD_2	MDAR[7:0] <- MIDR	ADR = 3	65
	LADD_3	MIDR <- M PC <- PC + 1	MEM = 001 PCI = 1	66
	LADD_4	MDAR[15:8] <- MIDR	ADR = 4	67
	LADD_5	MIDR <- M PC <- PC + 1	MEM = 001 PCI = 1	68
	LADD_6	MDAR[17:16] <- MIDR PC <- PC + 1	ADR = 5	69
	LADD_7	ADR <- PARAM STATE = FETCH2	PCI = 1	70
LOAD	LOAD	MDAR <- Read Matrix or MDAR read	ADR = PARAM MEM = 000	80
	LOAD_2	read wait	MEM = 000	81
	LOAD_3	MDDR <- M STATE = FETCH2 MDDR <- AC	MEM = 010 ACI = 1, AWM = 0	82
STAC	STAC	MDAR <- Write Matrix write STATE = FETCH2	ADR = PARAM MEM = 100	96
COPY	COPY	MIDR <- M PC <- PC + 1	MEM = 000 PCI = 1	112
	COPY_2	A_write_mux_en <- MIDR [7:5] ACI <- MIDR [4:0] STATE = FETCH2	OPR = 1 ALU = 1	113
RSET	RSET	MIDR <- M PC <- PC + 1	MEM = 000 PCI = 1	128
	RSET_2	RST <- MIDR STATE = FETCH2	OPR = 5	129
JUMP	JUMP	MIDR <- M PC <- PC + 1	MEM = 000 PCI = 1	144
	JUMP_2	JMP <- {1,PARAM[2:0]} if(J): PC <- MIDR else: none STATE = FETCH1	JMP = {1,PARAM[2:0]} OPR = 4	145
INCR	INCR	MIDR <- M PC <- PC + 1	MEM = 000 PCI = 1	160

	INCR_2	inc_ctrl_signals <- MIDR STATE = FETCH2	OPR = 3	161
DECR	DECR	MIDR <- M PC <- PC + 1	MEM = 000 PCI = 1	176
	DECR_2	dec_ctrl_signals <- MIDR STATE = FETCH2	OPR = 6	177
ADD	ADD	MIDR <- M PC <- PC + 1	MEM = 000 PCI = 1	192
SUBT	SUBT	A_write_ctrls <- PARAM ALU_add STATE = FETCH2	MEM = 000	208
DIV	DIV	MIDR <- M PC <- PC + 1	MEM = 000 PCI = 1	224
	DIV_2	A_bus <- MIDR ALU_div STATE = FETCH2	AWM = MIDR ALU = div	225
MUL	MUL	MIDR <- M PC <- PC + 1 A_bus <- MIDR	MEM = 000 PCI = 1 AWM = MIDR	32
	MUL_2	ALU_mul STATE = FETCH2	ALU = mul	33
TOGL	TOGL	TOG STATE = FETCH2	TOG = 1	240

Begin Signal

STATE MACHINE
Control Signal Generator

End Signal

INSTRUCTION MEMORY

Address bus
[8 bit]

Data bus
[8 bit]

PC

MIDR

OPR
Operand Router
Demux

PRM
Param Router
Demux

JMP
Jump Decider
Mux

INC
Increment
Decoder

DEC
Decrement
Decoder

RST
Reset
Decoder

ACI
Write to
Register
Decoder
[A bus -> Register]

REF ART [9 bit]

REF ARG [9 bit]

ADDRESS MAKER

MDAR [18 bit]

T

DATA MEMORY

Address bus
[18 bit]

Write signal

Data Bus
[8 bit]

AWT
[9 bit]

AWG
[9 bit]

MDDR

K0 REF

K1 REF

CROPPER

SHIFT REGISTERS

G0

G1

G2

A-BUS
(8 bit)

A-bus write Mux
AWM
[Register -> A-bus]

ALU (12 bit)

ALU
(12 bit)

12 bit
bus

AC
[12 bit]

LEGEND

- Registers (all 8 bit, unless otherwise specified)
- Control Modules
- Datapath
- Z-flags
- Control signals from state machines
- Decoded control signals

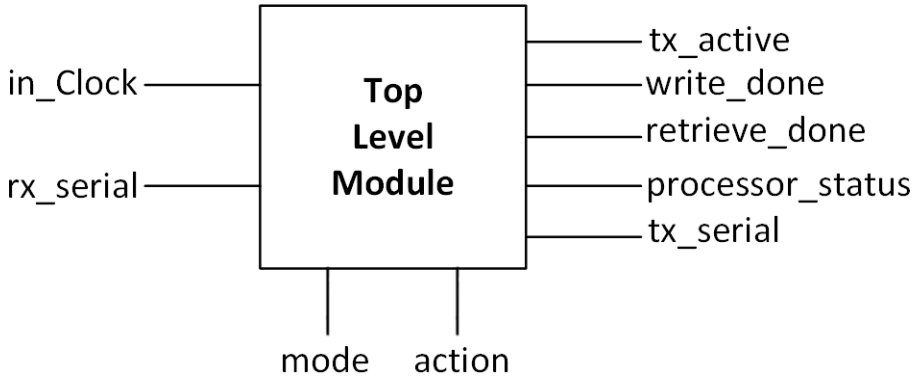
ABRUTECH

CUSTOM PROCESSOR

OPTIMIZED FOR MATRIX MANIPULATION

5.4 System

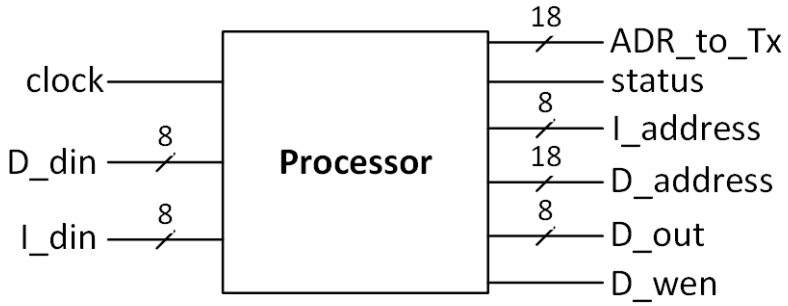
5.4.1 Top level module



This module holds all of the other modules of the project. A clock of 50MHz is fed in through **in_Clock** port and the data transmission lines from computer to the project are connected to **rx_serial** and **tx_serial** ports. It has indication LEDs attached to write_done – to indicate memory storing completion, **tx_active** – to indicate an active transmission, **retreive_done** – to indicate memory transmission completion and **processor_status** – to indicate whether the processor is busy or not.
The user controls the project in 4 modes of operation which are cycled using a single push button through mode pin.In each mode a different action can be performed using the push button connected to action pin.

Mode name	Description of the mode	Action button function
IDLE mode	Does nothing. Used for debugging	Cycles between IRAM and DRAM
RX mode	Can receive and store incoming data through serial to IRAM or DRAM	Cycles between IRAM and DRAM
PROCESSOR mode	Can use the processor to start processing in different clocks	Start process
TX mode	Used to initiate transmission of data back to the outside via serial	Start transmission

5.4.2 Processor

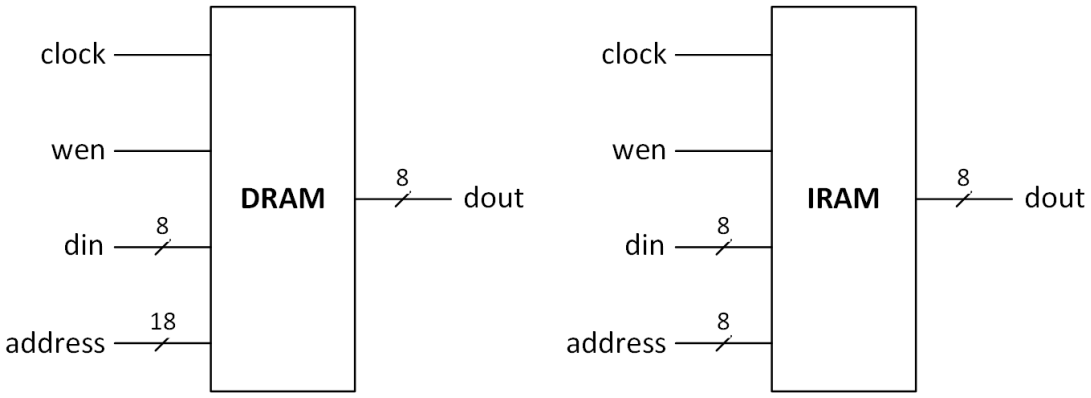


The processor module in our project is comparable with a real-world CPU chip. It houses all the processing related components and the modules outside it are simply support units. The control structures for doing the work after all the instructions and data are loaded in to the proper locations, and the unit is properly clocked, are contained within it and it just needs to be told when to start.

About the input ports of the module, first comes the **instruction memory data input** and **data memory data input** lines. Each of these are 8-bits wide and supplies the processor module with instructions and data needed for its operation. There is the **clock** input that clocks the whole thing, optimized for running at 50 MHz, but also operable at 1 Hz or hand clocked for demonstration and debugging purposes. Last input is the enable signal, which tells the processor all the data is in place, and commands to initiate the processing sequence.

Most obvious among the output ports are the 18-bit wide **transmission end address** and **data memory address buses**. Transmission end address bus goes into the communications system, more specifically the data retriever module, and commands it to 'crop' the outbound matrix data at this address. How the cropping process works is thoroughly explained in the communications section. Data memory address is basically the output of the ADR Maker module, which indicates which **address of data memory** is to be read from or written to. An identical but 8-bit wide line outputs the **instruction memory address**, again signaling which location of the instruction memory is to be read. Then there are the **data memory output** line, again 8-bit wide and its **write enable** control signal, which give out the data to be written to the data memory and controls when to write that data, respectively.

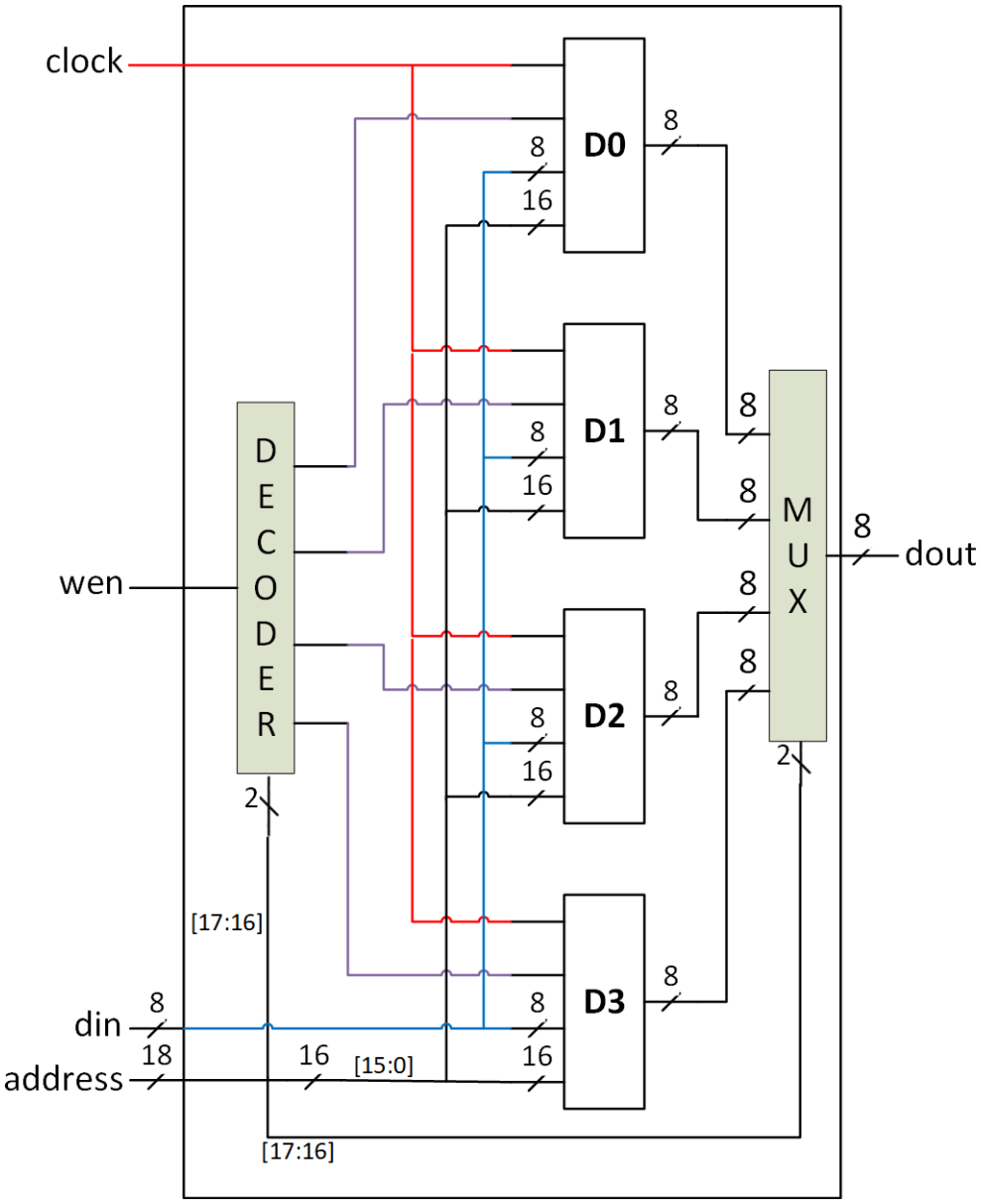
5.4.3 DRAM and IRAM



To create the DRAM and IRAM we have used the IP modules in QUARTUS II. These RAMs have a **din** port to feed data into the RAM module and **address** port to give the address and the **dout** bus to take the data out corresponding to the address given. When write enable pin **wen** is set to 1, data in the **din** bus is written into the location specified by the value in the **address** port. We have made the RAM modules negative edge sensitive by giving them an inverted clock pulse of the pulse used for other modules.

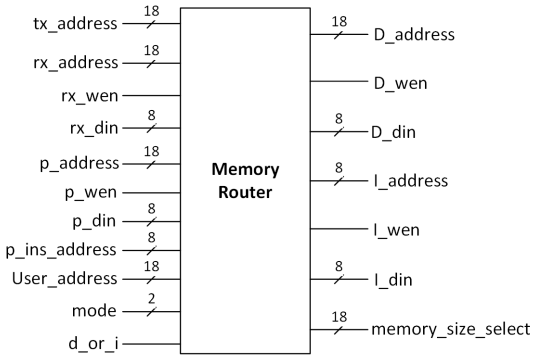
We created the IRAM (Instruction Random Access Memory) with a width of 8-bits and a depth of 256 locations.

The IP ram module of QUARTUS II was limited to a maximum depth of 65536 locations. Therefore, we had to combine 4 such RAMs (each of depth 65536 of 8-bit wide) to create the DRAM (Data Random Access Memory) which has 262144 locations of 8-bit wide in order to store 512 x 512 image. We have used a 4-way decoder and a multiplexer in order to combine them as seen in the wiring diagram below.



Inside the DRAM module, the 18-bit **address** is split in two, wiring most significant 2 bits of it to the selection ports of the decoder and the multiplexer while the least significant 16 bits are connected directly to the address bus of each sub RAM module (D0, D1, D2, D3). So the correct RAM output is given out through the multiplexer. **wen** of the DRAM is connected to the **enable** port of the decoder. So when **wen** is low, the decoder will give 0 to the write enable port of each sub RAM. When **wen** is high, the decoder will give 1 to the write enable port of the sub RAM, selected by the most significant 2 bits of the address writing only to that particular sub RAM.

5.4.4 Memory router



We have a single IRAM and a DRAM in our project. But that same ram needs to be used by other modules according to the mode of operation.In the IDLE mode, the rams are not used other than by the user for debugging purposes. In the Rx mode the IRAM/DRAM should be handed over to writer module. In PROCESSOR mode, both RAMS are used by the processor and in the Tx mode, only the DRAM is used for the transmission process. dout of IRAM is directly connected to "processor" module while the dout of DRAM is directly connected to both "processor" and "UART Tx" modules. The write enable, din and the address of IRAM and DRAM are connected to **I_wen, I_din, I_address** and **D_wen, D_din, D_address** ports respectively. So, in this module, above IRAM/DRAM ports are routed to different modules to use according to the mode of operation selected using **mode** port, and the ram selected using **d_or_i** port

Modules using dram in different modes of operation

	IDLE (00)	Rx (01)	PROCESS (10)	Tx (11)
Wen	-	Data_writer	Processor	-
Din	-	Data_writer	Processor	-
Address	switches	Data_writer	Processor	Data_retreiver
Dout	-	-	Processor	Data_retreiver

Modules using iram in different modes of operation

	IDLE (00)	Rx (01)	PROCESS (10)	Tx (11)
Wen	-	Data_writer	-	-
Din	-	Data_writer	-	-
Address	switches	Data_writer	Processor	-
Dout	-	-	Processor	-

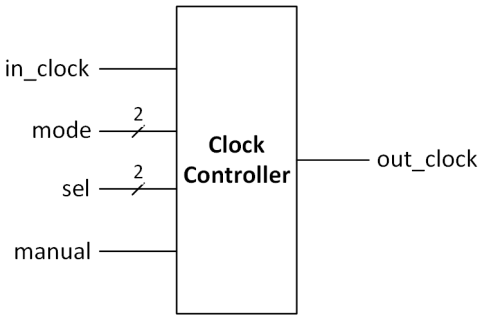
In "IDLE mode", user is able to view the data stored in the RAMs on Seven Segment Displays using the switches connected to **user_address** port to change address. So the address bus of IRAMDRAM is routed to **user_address** port inside.

We use the uart Rx to load data and instructions to DRAM and IRAM respectively. So in "Rx mode", the "data_writer" module must be able to access IRAM or DRAM according to the value in **d_or_i** port (DRAM : 0 , IRAM : 1) and according to which ram it writes, the address up to which it should be written is sent to "data_writer" module through **memory_size_select** port (255 if IRAM selected. Else 262143). So in this mode the ram ports are routed to **rx_din, rx_address, rx_wen** so that the "data_writer" module can access the selected RAM.

In "Processor mode", the "processor" needs access to wen, address, din of the DRAM and address of the IRAM. So the address bus of IRAM is routed to **p_ins_address** while the din,address and wen of DRAM is routed to **p_din, p_wen** and **p_address**. In "Tx mode" we only transmit the data in DRAM. So, the address bus of DRAM is routed to the port **tx_address** so that the "UART Tx" module can access it.

5.4.5 Clock control

We have used this module in our top level module in order to generate different clock signals from **out_clock port**, using the clock of 50 MHz fed in through the **in_clock port**. The mode of operation of the system is given into this module through the **mode** port and in the "Processor mode" (mode = 10) it can output clock signals of,

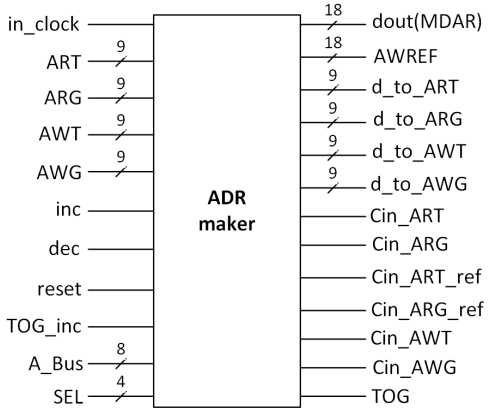


1. 10 MHz (sel = 00)
2. 1 Hz (sel = 01)
3. Manual clock (sel = 10)
4. 25 MHz (sel = 11)

In any other mode, this will always give out a clock of 10 MHz as Transmission and Receiving processes need a clock of 10 MHz. In "Processor mode" the clock of choice can be changed using the **sel** port. When using the manual clock mode, the manual clock pulse can be given through the **manual** port using a push button.

5.5 Special Modules

5.5.1 ADR Maker

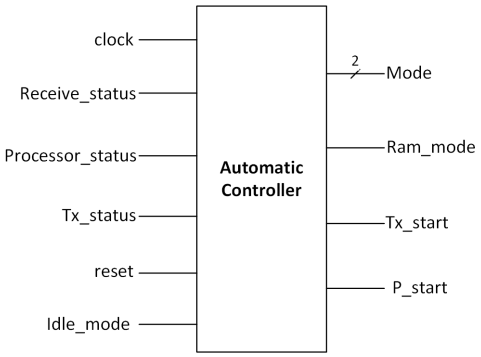


PC register (Program counter) also a positive clock sensitive register is the register holding the address of the next instruction or the operand in the IRAM. It is an 8-bit wide register starting from 0000 0000 and is incremented by the "state machine" via **inc** pin. Its stored value is accessed by the IRAM address bus through **dout**. The value in PC can also be overwritten by the value in **din** by making **cin** pin high, which is used when jumping.

An 18-bit dout port connects to the input of MDAR register that indicates which address of the DRAM will be written to or read from.**AWREF** goes to the communication controller to be used when data is transmitted from the memory to denote what is the end address for valid data range. This is discussed further in the communications section. There are 4 more 9-bit ports to output data to the special registers and their write control signals. The ref lines write the current input of the ART or ARG to a special reference location of the register itself and is used to specify looping limits. Finally, **TOG** outputs the state of the T flag to be used by the jump controller to govern loops.

There are 5 ways that the module can create an address to be written to MDAR. It can load a predefined address from 3 eight-bit RAM locations using the LADD instruction. Or the address can be ART concatenated with ARG or ARG concatenated with ART or AWG concatenated with AWT or AWT concatenated with AWG. The latter 4 methods are used for matrix manipulation, reading or writing, row-wise or column-wise. It also has the capability to load an address from the memory using LADD, break down its contents into two parts and load them into the pairs of special registers. This module is what has enabled us to manipulate the 18-bit address using only an 8-bit wide memory. The special registers themselves can be incremented independently, allowing the linear 262,144 location memory to be treated as a 512 X 512 square shaped memory, essentially representing a square matrix, or more importantly, an image. ADR maker can hold two separate address sequences, one for reading data and other for writing data. For the image down sampling algorithm, each cycle increments the read address by two and write address by one. A normal processor would need about 10-20 instructions per loop or hardcoded addresses to accomplish this but ADR maker cuts it down to 2 instructions. Additionally, the T flag which can be manually toggled via TOGL instruction provides a way to run the same program twice with address format changed, for example, running the down sampling twice but once for rows and once for columns.

5.5.2 Automatic Controller



This processor was later modified to operate automatically when data were fed. In the Automatic Mode, the mode selection task (IDLE, Rx, PROCESS, Tx) is done automatically by this module called **'Automatic Controller'**. This module functions in the negative edge of clock and its input, and output ports are described below.

Receive_status: Keeps track whether the receiver is busy or not.
Processor_status: Keeps track whether the processor is busy or not.
Tx_status: Keeps track whether the transmitter is busy or not.
Reset: Reset the entire process.
Idle_mode: This port is connected to key3. Whenever this is pressed during the LOAD_DAT state, the state will change to IDLE so that it can browse the data stored in RAMs using switches.
Mode: Mode of operation (IDLE (00), Rx (01), PROCESS (10), Tx (11)).
Ram_mode: States whether the RAM is IRAM (1) or DRAM (0).
Tx_start: Gives the transmit start signal.
P_start: Gives the processor enable signal.
This module basically acts as a state machine which switches between the following states.

1. **LOAD_INS**: Initial state which loads instructions to the IRAM in the Rx mode. When the IRAM is full the state is changed to the LOAD_DAT state.

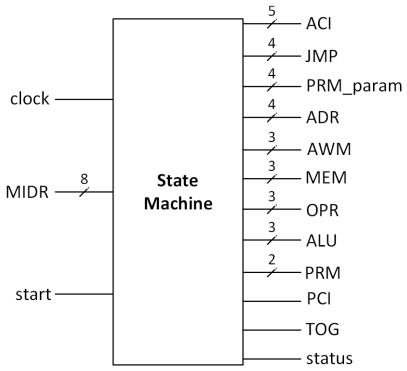
2. **LOAD_DAT**: Loads data to the DRAM in the Rx mode. In this state,
   • Whenever the reset button is pressed, the state will change to LOAD_INS.
   • Whenever the idle button is pressed, the state will change to IDLE so that it can browse the data stored in RAMs using switches.
   • If the DRAM is full, the state will change to PROCESS state.

3. **IDLE**: In this state the user has access to browse the data in both IRAM and DRAM.

4. **PROCESS**: Whenever the processing is finished it will change the state to TRANSMIT.

5. **TRANSMIT**: In this state the transmission of the processed data occurs. When the transmission is done the state will change to LOAD_DAT state.

5.5.3 State Machine



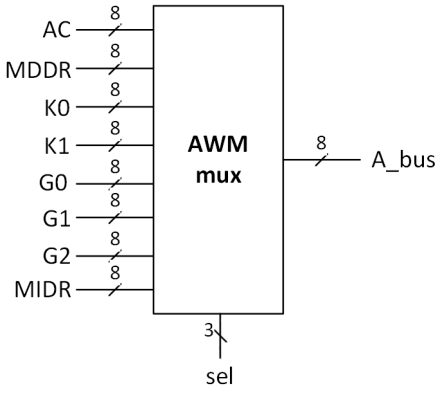
All the control signals are given by the State machine. This has three inputs,

Clock
MIDR: Inputs the instruction to the state machine
Start: Escape from the END state and start processing
The output control signals are given as follows,
ACI: Gives write enable signals to AC, MDDR, K0, K1, G0 modules.
JMP: Gives the selection signal to the JMP Mux.
PRM_param: The least significant 4 bits of the instruction are passed to the PRM module.
PRM: Gives selection signal to PRM module.
ADR: Gives the selection signal to the ADR Maker.
AWM: Gives the selection signal to the AWM Mux.
MEM: Gives the write enable signals to each DRAM, MDDR, MIDR as a bundle of 3-bits.
OPR: Gives the selection signal to OPR Mux.
ALU: Gives the Operation selection bits to ALU.
PCI: Gives the increment signal to PC.
TOG: Toggle signal which is given to the ADR maker.
Status: State whether the Processor is busy or not.

PCI	OPR	PRM	ACI	AWM	INC	DEC	RST	MEM			ALU	ADR	JMP	TOG
								DM write	MDDR-M-ci	MIDR-M-ci				
Increment PC	0-none 1-ACI-AWM 2-AWM 3-INC 4-PC 5-RST 6-DEC	0-none 1-ADR 2-JMP 3-ADD/SUB	0-AC 1-MDDR 2-K0 3-K1 4-G0	0-AC 1-MDDR 2-K0 3-K1 4-G0 5-G1 6-G2 7-MIDR	0-MDAR 1-ART 2-ARG 3-AWT 4-AWG 5-AC 6-K0 7-K1	0-MDAR 1-ART 2-ARG 3-AWT 4-AWG 5-AC 6-K0 7-K1	0-MDAR 1-ART 2-ARG 3-AWT 4-AWG 5-AC 6-K0 7-K1				0-none 1-add 2-sub 3-div 4-mul	0-none 1-R matrix 2-W matrix 3-A-> last 8 4-A-> midd 8 5-A-> firs 2 6-To MDAR 7-To AR 8-To AW 9-To ARREF	0-none 1-JUMP 2-JMPZ 3-JPNZ 4-JZT 5-JNRG 6-JNRT 7-JNK0 8-JNK1	Tog

5.6 Controllers: Muxes, Demuxes and Decoders

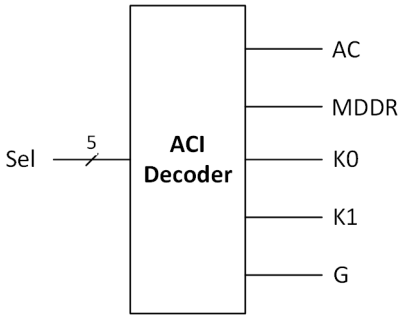
5.6.1 AWM(A-Bus write mux)



The AWM mux module is used to choose and rout one of the outputs of many registers into the **A_bus** of 8-bit width. The selection is done via the **sel** port, with the combinations as follows,

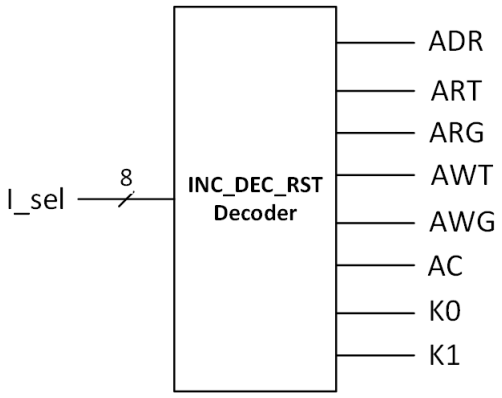
AWM selection	
Selection bit pattern given	Register occupying the A-bus
0 = 000	AC
1 = 001	MDDR
2 = 010	K0
3 = 011	K1
4 = 100	G0
5 = 101	G1
6 = 110	G2
7 = 111	MIDR

5.6.2 ACI Decoder



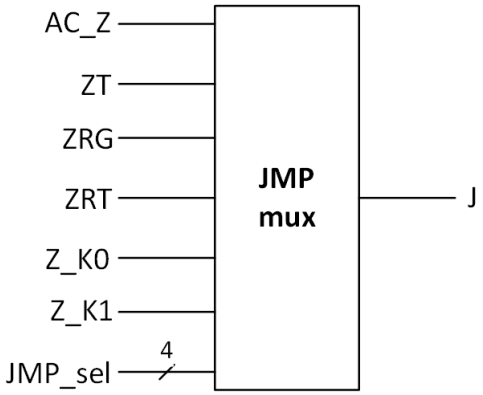
ACI Decoder module controls which registers are written from the A bus. The incoming signal **'SEL'**is decoded by this and the 'Cin' of the writable registers **AC, MDDR, K0, K1** and **G**. It is not clock sensitive and changes its output whenever its input changes.

5.6.3 INC, DEC, RST Decoder



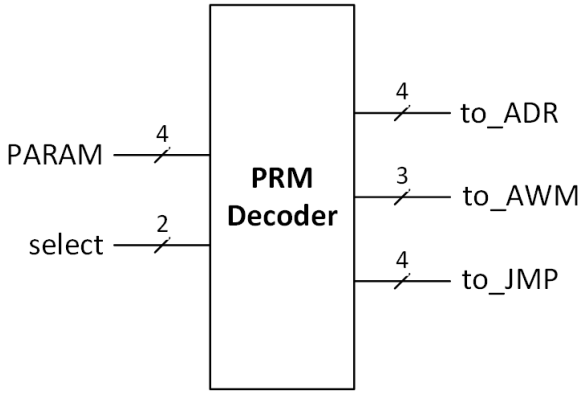
This definition is instantiated as 3 separate modules inc decoder, **dec** decoder and **rst** decoder. It takes an 8-bit selection input and gives out control signals (to increment, decrement or reset) to our main registers **ADR, AWT, AWG, ART, AWG, K0, K1** and **AC**. The module is not clock sensitive and changes output whenever input is changed.

5.6.4 JMP Mux



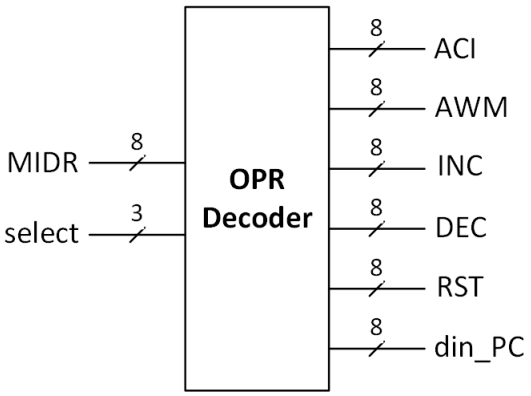
Jumping is one of the most important tasks in the processor. Since our processor is customized for matrix manipulation, there are several uncommon jumping procedures. These jumping procedures include watching for the zero flag of different registers and when they do indicate zero, command the program counter to perform the jump. JMP_mux achieves this using a 4-bit selection line, which tells the module to which flag to monitor, and when the monitored flag goes high (or low as depending on the jump mode), the **jump** signal that connects to the program counter goes high, indicating it to take the jump.

5.6.5 PRM Decoder



Parameterized instructions are a feature we adapted from high end architectures like RISC V and x86. This allows a lower number of opcodes with different options, narrowing down the semantic gap between the human programmer and assembly code. The part that handles the parameter parts of the opcodes is the Parameter Mux or PRM. It accepts the parameter input from both the state machine state output and the parameter part of the opcode itself. State machine state directs the parameter given in the opcode to its destination, which can be the **Jump Mux**, **ADR maker** or the **ALU**. It can select the flag used in jumps, change the address source (MIDR or matrix mode) or direct specific ALU states. The module is not clock sensitive and changes output upon input change.

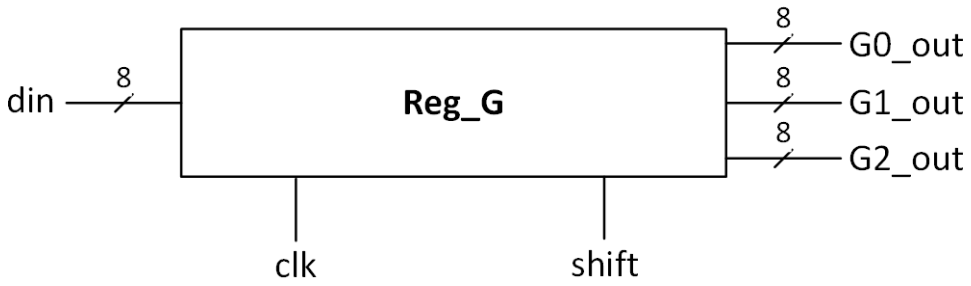
5.6.6 OPR Decoder



OPR stands for operand path router. This module is used to route the operand part of two-part instructions. The inputs are the 8-bit operand line and the 4-bit selection line, commanded by the state machine. As guided by the selection line, this module routes the operand to **ACI, AWM, INC, DEC, RST** and **Din_PC**, most of which are decoders themselves. The module works on the current inputs, without needing a clock.

5.7 Registers

5.7.1 Shift Registers (G0, G1, G2)

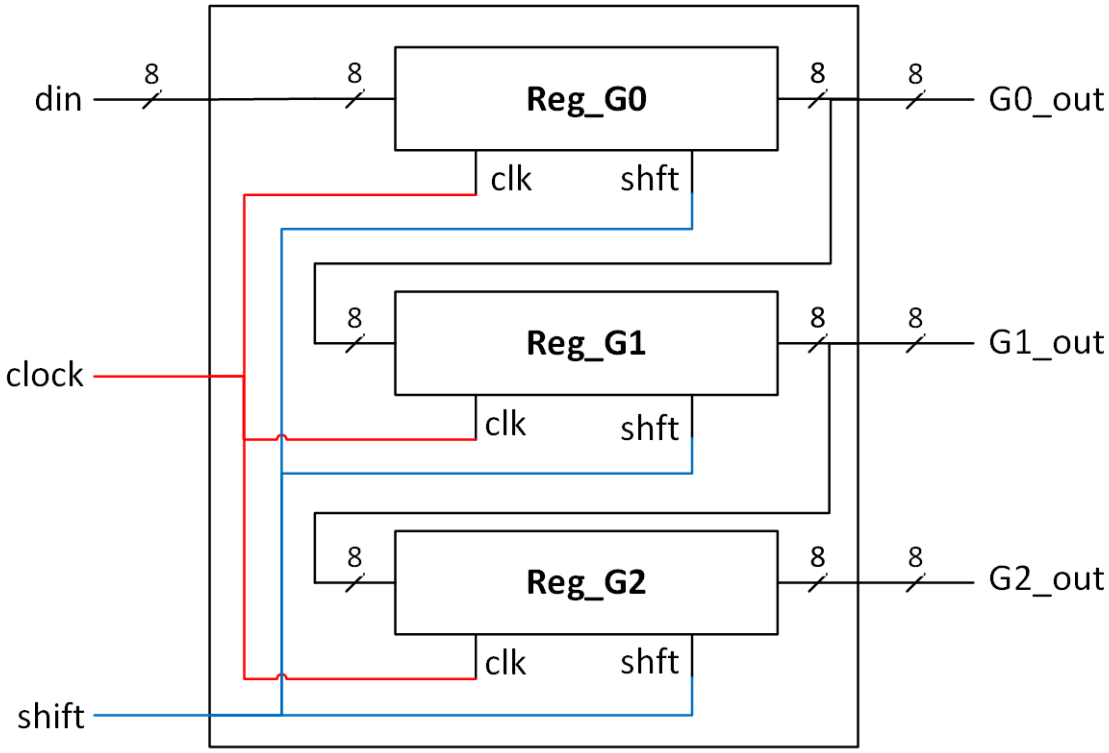


This is a single module containing 3 registers (G0, G1, G2) inside which are used for shifting purposes. The data stored in each of the G registers inside can be accessed by the 3 output busses **G0_out, G1_out, G2_out**.

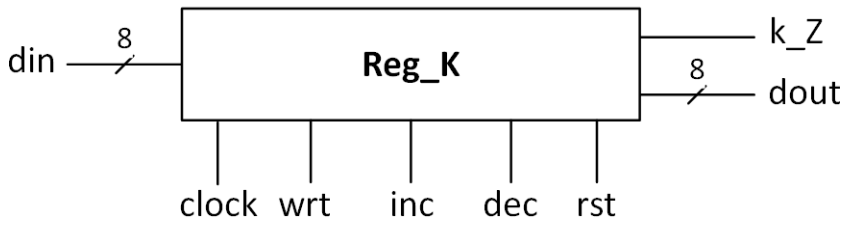
This module is positive clock sensitive. New data can be fed into this using the **din** port. When the **shift** pin is set high, at each positive edge of the clock the data get shifted to the next register as follows.

din => G0 => G1 =>G2

Inside this module, the G registers are arranged as follows



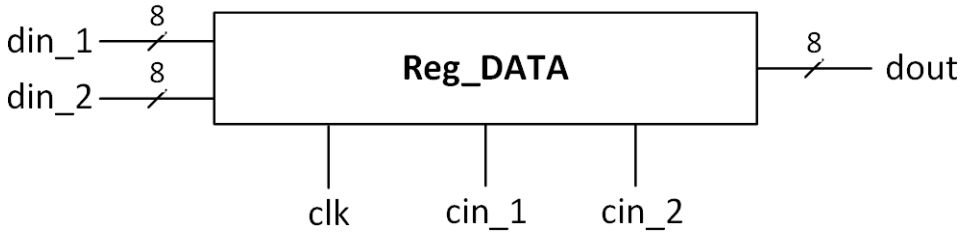
5.7.2 Looping Registers (K0, K1)



In our processor we have used two special purpose L registers named K0 & K1 which are optimized for looping (can also be used as general purpose registers). This register has the functionality to store a reference value which is used in looping process. When writing for the first time to the register it stores that value in the K register as well as the reference. When writing further, if current value equals the reference, it raises the **k_Z** flag.

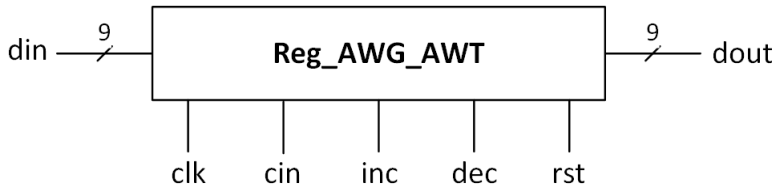
Writing to the K register is done by making **wrt** pin high; which is connected to ACI decoder. This register can be incremented, decremented, and reset using the respective control pins **inc, dec** & **rst**. These pins are connected to the outputs of INC, DEC and RST decoders. Moreover, the din input bus is directly connected to A-bus and the dout bus is connected to an input of AWM mux.

5.7.3 Data registers(MDDR/MIDR)



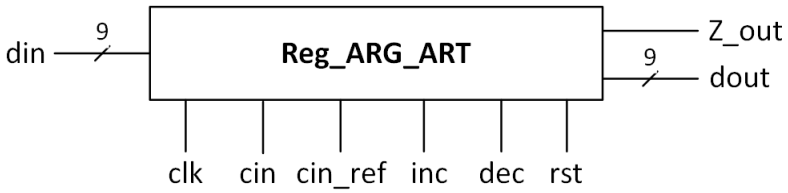
We instantiated the registers MDDR (Memory Data Register) and MIDR (Memory Instruction Data Register) using the same module "reg_DATA". This module is positive clock sensitive. MDDR is taking in data 2 ways, one from dout of DRAM and the other from the A-bus which are connected directly to **din_1** and **din_2** respectively. When the control pins **cin_1** or **cin_2** are set to 1, the value in din_1 or din_2 is written into the register. The stored value can be accessed through **d_out** port. **d_out** of the MDDR register is connected to an input of the "AWM decoder".

5.7.4 AW registers(AWT/AWG)



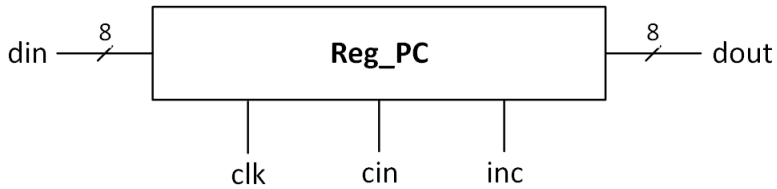
These positive clock sensitive AW registers are used for matrix manipulation when writing to memory by the processor. We have instantiated 2 registers AWT and AWG each 9-bit wide. The value stored in these registers can be incremented, decremented and reset by the control pins **inc, dec, rst**. And new data can be written into this from **din** port by making **cin** high. Data stored in this can be accessed through **dout** port. By combining these 2 registers to form an 18-bit address, user can surf through memory locations as in a matrix.

5.7.5 AR registers(ART/ARG)



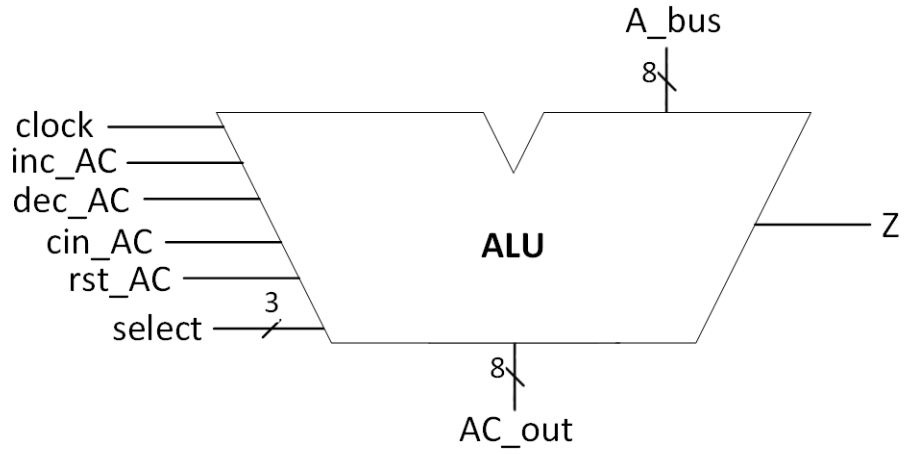
This AR register has all the features of AW registers; and it also has some additional features. This module contains an additional reference registers where the input data can be written to it by making the **cin_ref** port high. And also, whenever the values of the two internal registers are equal, the **Z_out** flag will be set to high.

5.7.6 PC



PC register (Program counter) also a positive clock sensitive register is the register holding the address of the next instruction or the operand in the IRAM. It is an 8-bit wide register starting from 0000 0000 and is incremented by the "state machine" via **inc** pin. Its stored value is accessed by the IRAM address bus through **dout**. The value in PC can also be overwritten by the value in **din** by making **cin** pin high, which is used when jumping.

5.7.7 ALU((Arithmetic and Logic Unit))



This is the module that does all the arithmetic operations to data inside the processor. It is positive clock sensitive and is connected to A-bus through **A_bus** port. The **AC** register is located inside the ALU module and its value is always given out through **AC_out** port. The value stored inside the 12bit wide AC register can be incremented, decremented and reset using the control pins **inc_AC, dec_AC**, and **rst_AC** input pins respectively. In order to write into AC from A-bus, the **cin_AC** pin can be made 1.

Selection (select)	Operation	Description
000	None	No operation
001	AC <= AC + A-bus	Data in A-bus is added to AC and stored back in AC
010	AC <= | (AC - A-bus) |	Data in A-bus is subtracted from AC and stored back in AC
011	AC <= AC/A-bus	Data in AC is divided by A-bus value and rounded off answer is stored back in AC
100	AC <= AC * A-bus	Data in AC is multiplied by A-bus value and rounded off answer is stored back in AC

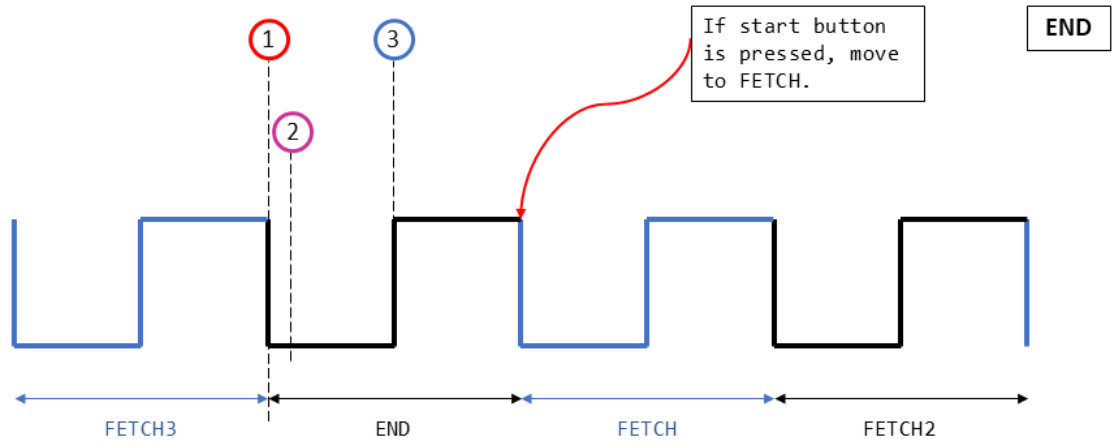
Whenever the value of AC becomes 0, the flag **Z** is set high. The arithmetic functions can be controlled using the combination given through **select** port.
We have made the AC register 12 bit wide in order to avoid overflowing problems when adding.

6 Timing Diagrams of Instructions

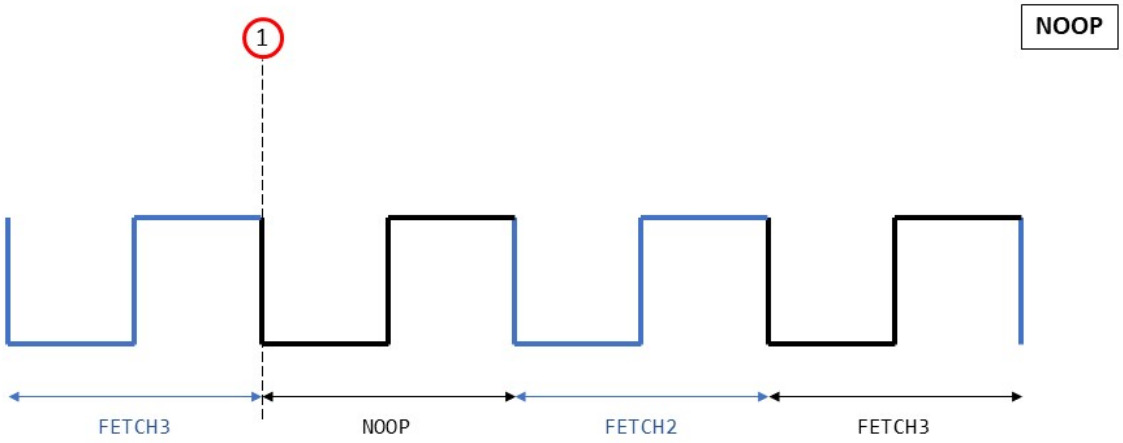
Our processor is designed such that the control signals are issued by the state machine at negative clock edge and the modules respond at positive clock edge. By experimentation with hand-driven clock, we found the following behavior with the RAM module and fine-tuned our timing to reduce the number of clock cycles required:

MDAR is updated at positive edge, RAM accepts the address in the following negative edge. In the next negative edge, data is released to the data bus. MDDR has been designed to accept the data in the following positive edge. Hence reading takes 2 clock cycles.

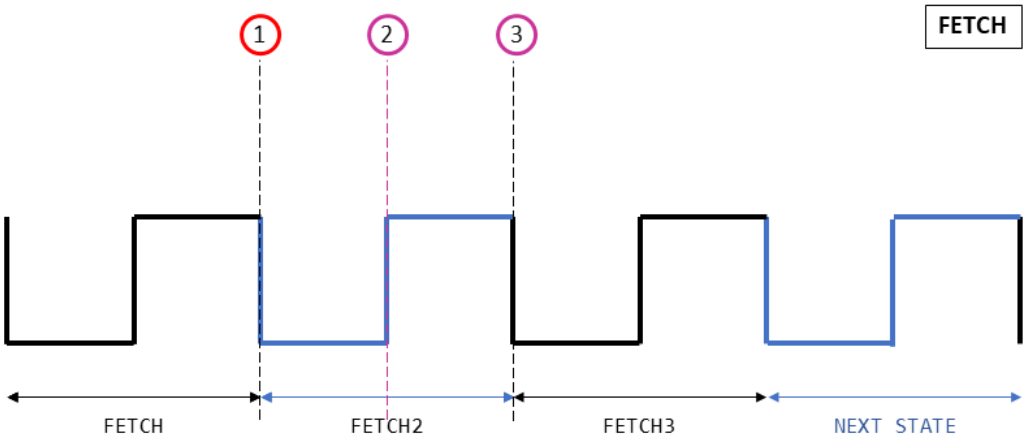
MDAR is updated at negative edge, RAM accepts the address in the following negative edge and immediately writes to the respective location. In the next negative edge, data is released to the data bus. Hence writing take only 1 clock cycle.



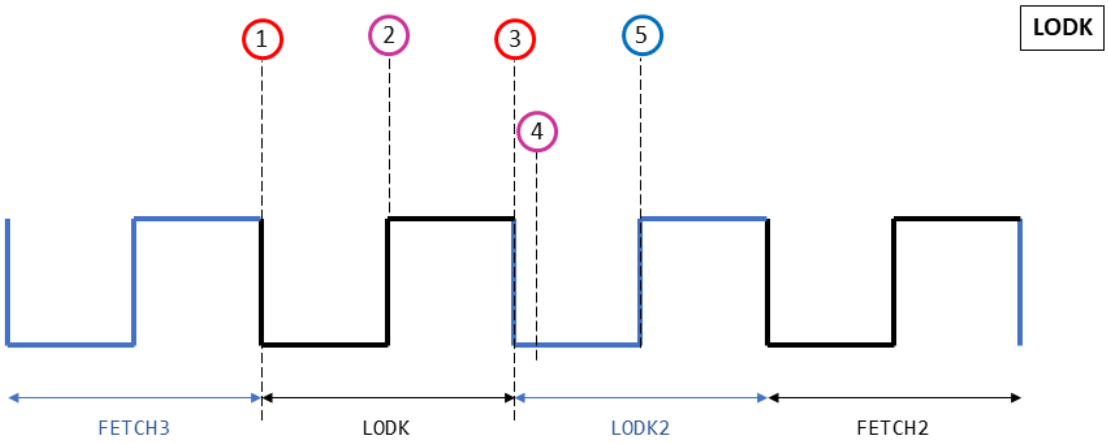
1) Control signal is given to JMP mux.
   Control signal to OPR is given to map MIDR to PC
2) Write enable signal to PC is given.
   MIDR is routed to PC through OPR.
3) PC is updated (with 00000000).



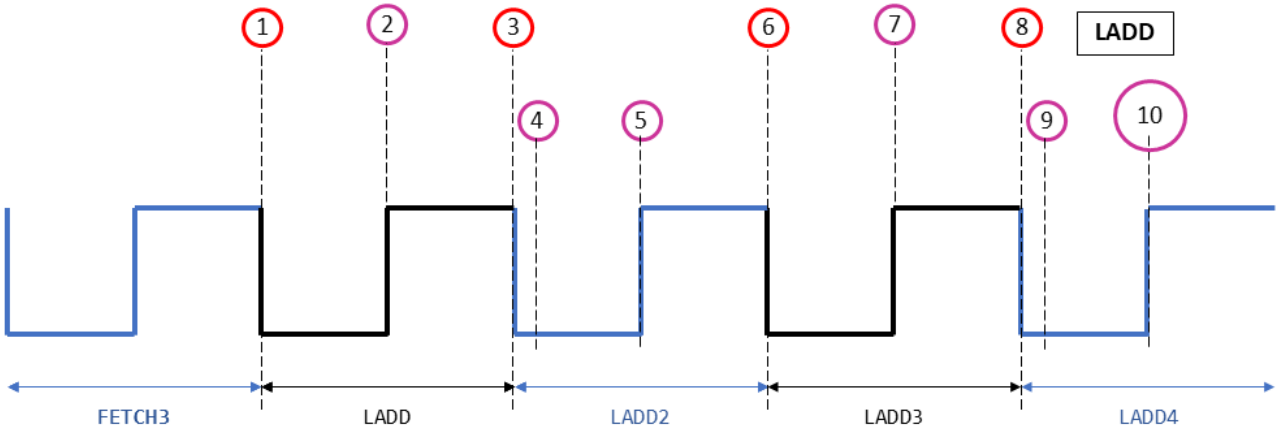
1) Making all control signals zero



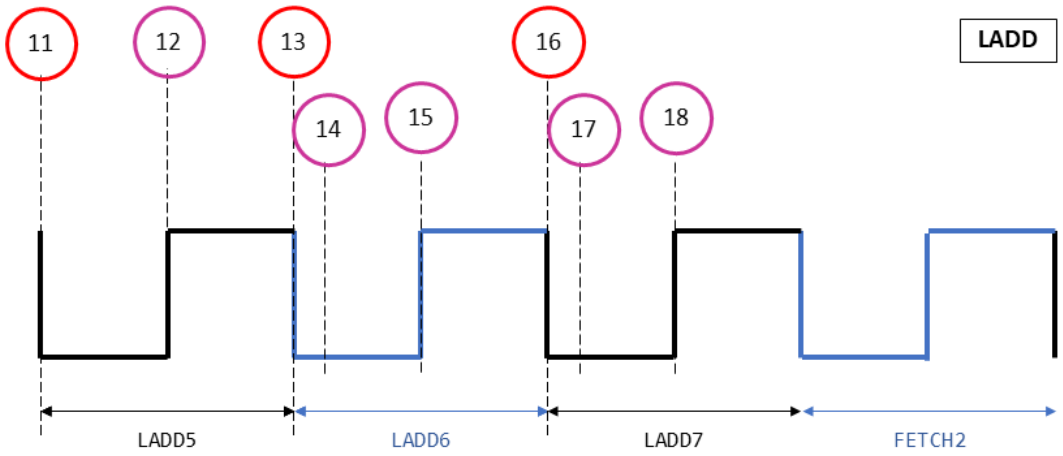
1) Control signal, PC increment is given.
   Control signal, MIDR write enable is given.
2) MIDR is updated.
   PC is incremented.
3) Decide next state according to instruction in MIDR.
   Parameter is routed to PRM input.



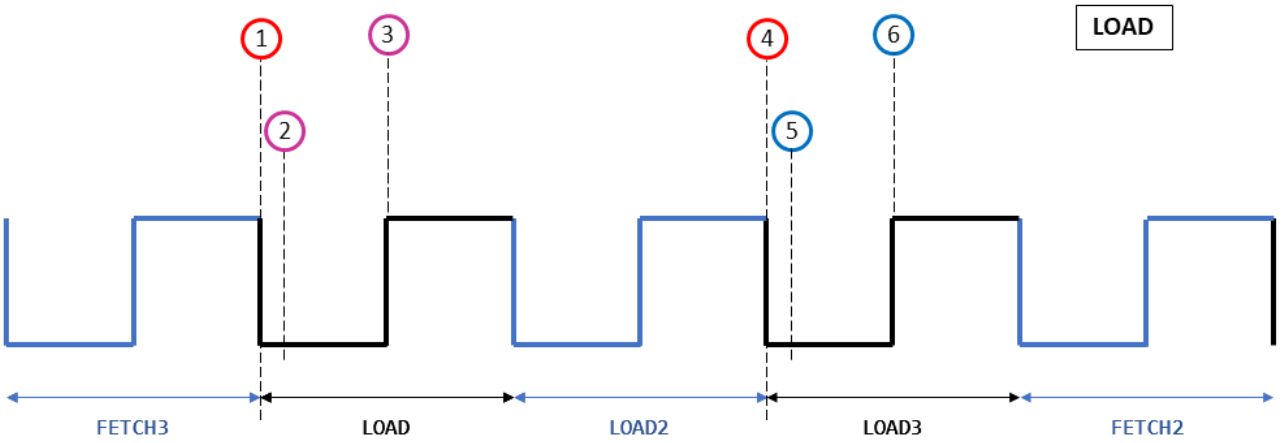
1) Control signal, PC increment is given.
   Control signal, MIDR write enable is given.
2) MIDR updates with operand.
   PC incremented.
3) Control signal to AWM is given to load MIDR to A-bus.
   Control signal to ACI is given to write from bus to AC.
4) A-bus is updated with MIDR.
5) Data is written to AC from A-Bus.



1) Control signal, PC increment is given.
   Control signal, MIDR write enable is given.
2) MIDR updates with first operand.(000000XX – Most significant 2 bits of the address)
   PC incremented.
3) Control signal is given to AWM to update A-bus from MIDR.
   Control signal to ADR is given to update most significant two bits of Temp_MDAR.
4) A-bus is updated with MIDR.
5) Temp_MDAR's most significant two bits are updated.
6) Control signal, PC increment is given.
   Control signal, MIDR write enable is given.
7) MIDR updates with second operand.(XXXXXXXX – Mid 8 bits of the address)
   PC incremented.
8) Control signal is given to AWM to update A-bus from MIDR.
   Control signal to ADR is given to update mid 8 bits of Temp_MDAR.
9) A-bus is updated with MIDR.
10) Temp_MDAR's mid 8 bits are updated.

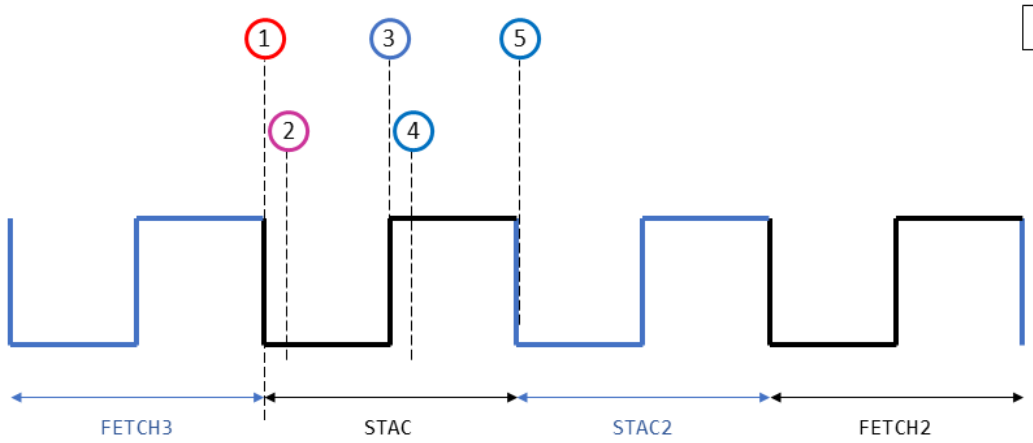


11) Control signal, PC increment is given.
    Control signal, MIDR write enable is given.
12) MIDR updates with third operand.(XXXXXXXX – least significant 8 bits of the address)
    PC incremented.
13) Control signal is given to AWM to update A-bus from MIDR.
    Control signal to ADR is given to update least significant 8 bits of Temp_MDAR.
14) A-bus is updated with MIDR.
15) Temp_MDAR's least significant 8 bits are updated.
16) Control signal is given to PRM to pass the parameter to ADR.
17) Parameter is routed to ADR.
18) ADR updates the desired registers(MDAR,AR,AW,AR_REF) according to the parameter.



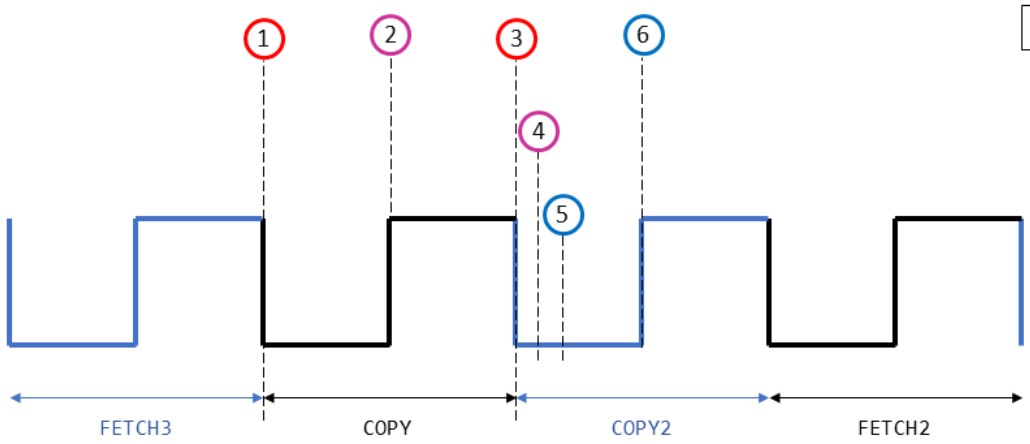
1) Control signal is given to PRM to update MDAR according to parameter.
2) Parameter is routed to ADR.
3) MDAR gets updated according to parameter.
4) Write enable pin of MDDR is set to 1 to update MDDR from DRAM output value.
5) Data from DRAM is available at MDDR din.
6) MDDR is updated with the value requested from DRAM.

STAC



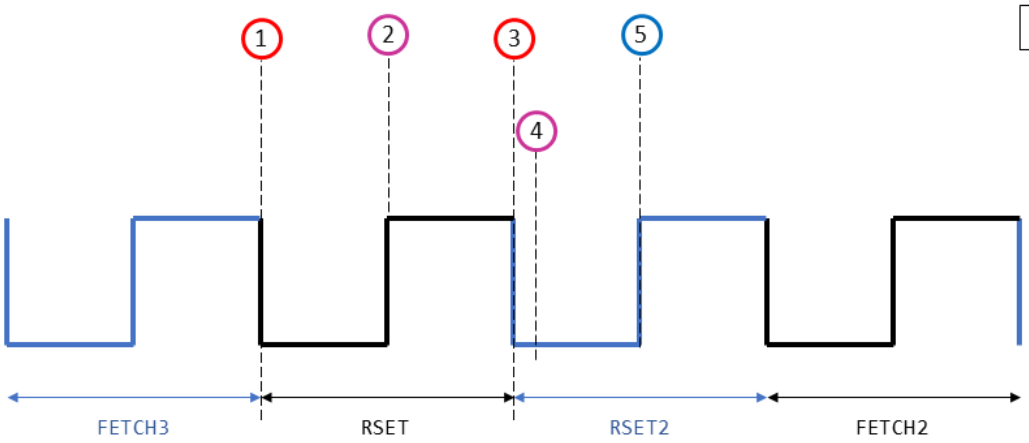
1) Control signal, to ACI is given to write from A-bus to MDDR.
   Control signal, DRAM write enable is set to high.
   Control signal is given to PRM to rout the parameter to ADR.
2) Cin pin of MDDR is set to 1.
   Parameter is routed to ADR through PRM.
3) MDDR gets updated with the value of A-bus(AC).
   MDAR gets updated according to parameter.
4) Din bus of MDDR is stabilized.
   Address bus of MDDR is stabilized.
5) MDDR value is written to DRAM in the location in MDAR.

COPY



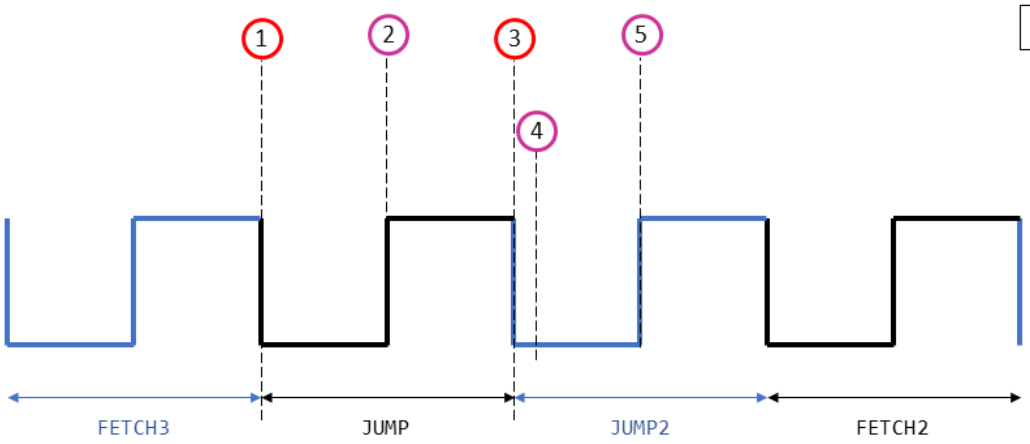
1) Control signal, PC increment is given.
   Control signal, MIDR write enable is given.
2) MIDR updates with operand.
   PC incremented.
3) Control signal to OPR is given to split the operand into AWM & ACI selection bits.
4) Operand is being split as AWM & ACI selection bits by OPR.
5) A-bus is updated with the corresponding register data & the write enable signals for the desired registers are given.
6) Data is written to the desired registers from A-Bus.

RSET



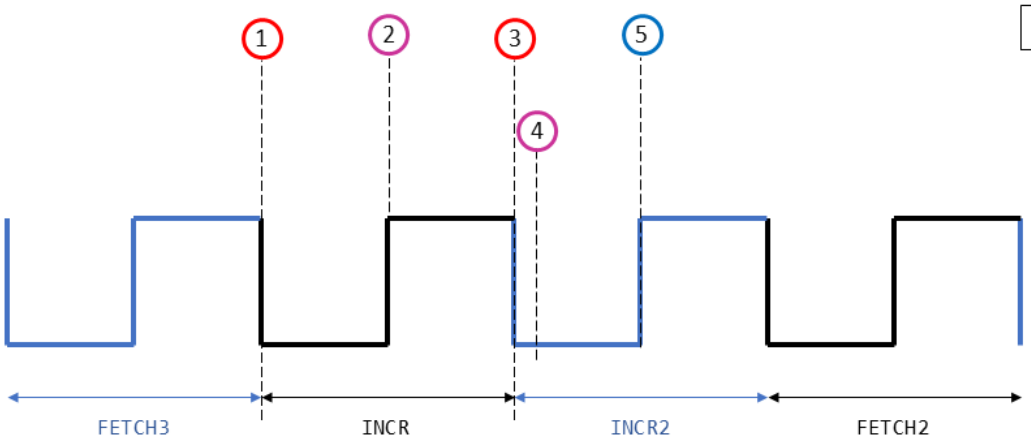
1) Control signal, PC increment is given.
   Control signal, MIDR write enable is given.
2) MIDR updates with operand.
   PC incremented.
3) Control signal to OPR is given to rout the operand to RST decoder.
4) Reset pins of desired registers are set to 1.
5) Desired registers gets reset.

JUMP



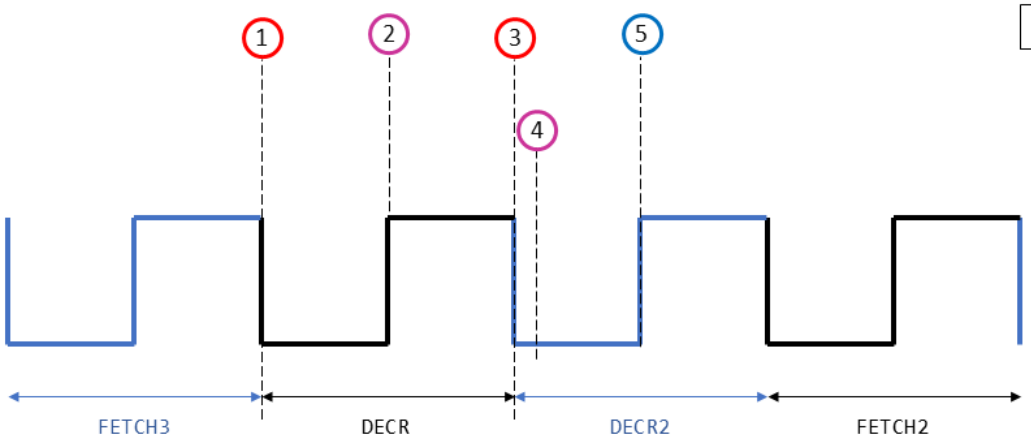
1) Control signal, PC increment is given.
   Control signal, MIDR write enable is given.
2) MIDR updates with operand.
   PC incremented.
3) Control signal to PRM is given to rout the parameter to JMP Mux.
   Control signal is given to OPR to rout MIDR to PC.
4) MIDR is routed to PC.
   Parameter is routed to JMP mux.
5) PC write enable (output of JMP mux) is executed accordingly.

INCR



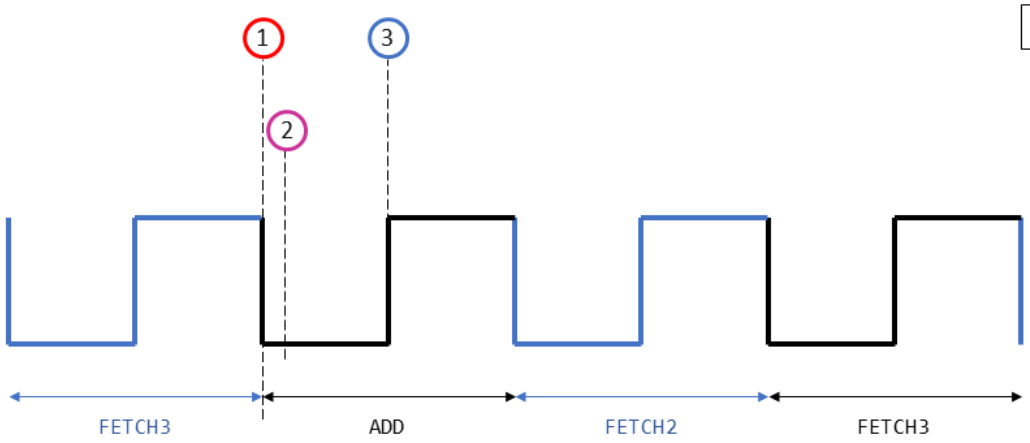
1) Control signal, PC increment is given.
   Control signal, MIDR write enable is given.
2) MIDR updates with operand.
   PC incremented.
3) Control signal to OPR is given to rout the operand to INC decoder.
4) Increment pins of desired registers are set to 1.
5) Desired registers gets incremented.

DECR



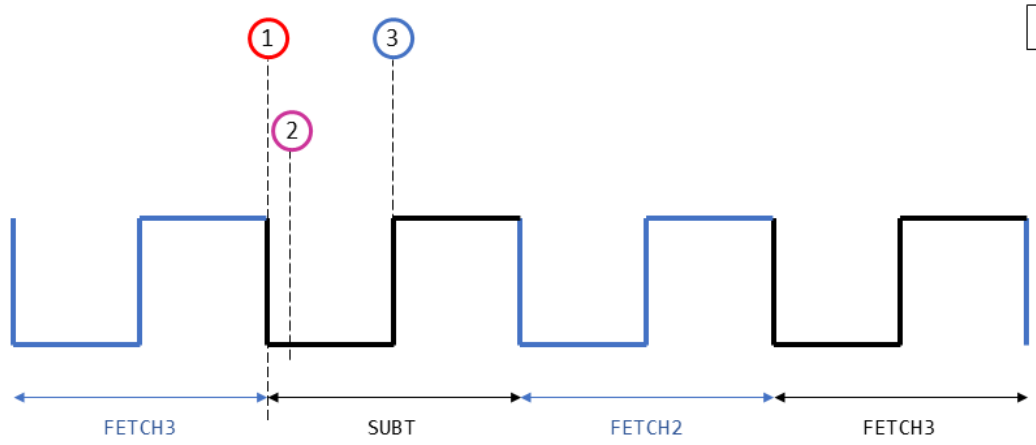
1) Control signal, PC increment is given.
   Control signal, MIDR write enable is given.
2) MIDR updates with operand.
   PC incremented.
3) Control signal to OPR is given to rout the operand to DEC decoder.
4) Decrement pins of desired registers are set to 1.
5) Desired registers gets decremented.

ADD

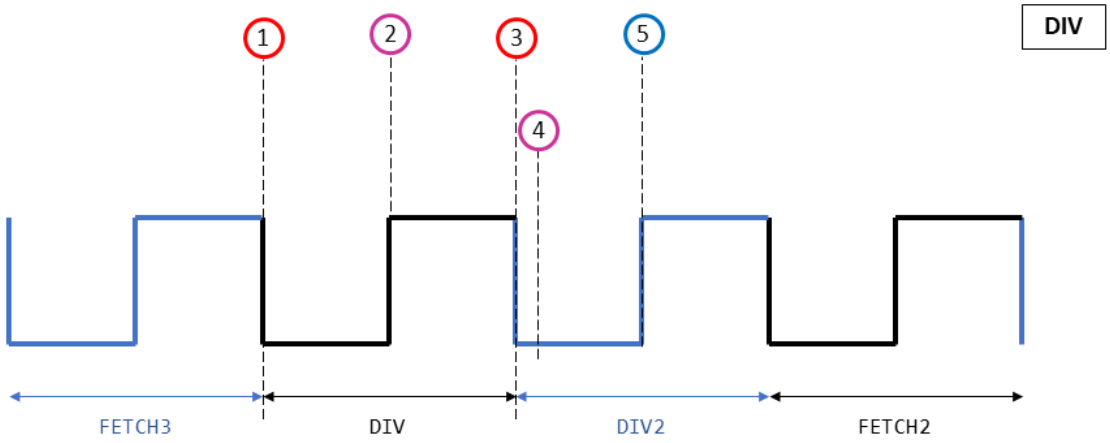


1) Control signal, ALU-ADD is given.
   Control signal to PRM is given to rout parameter to AWM.
2) A-bus gets updated with the desired register value which needs to be added.
3) AC gets updated as, AC <- AC + A-bus

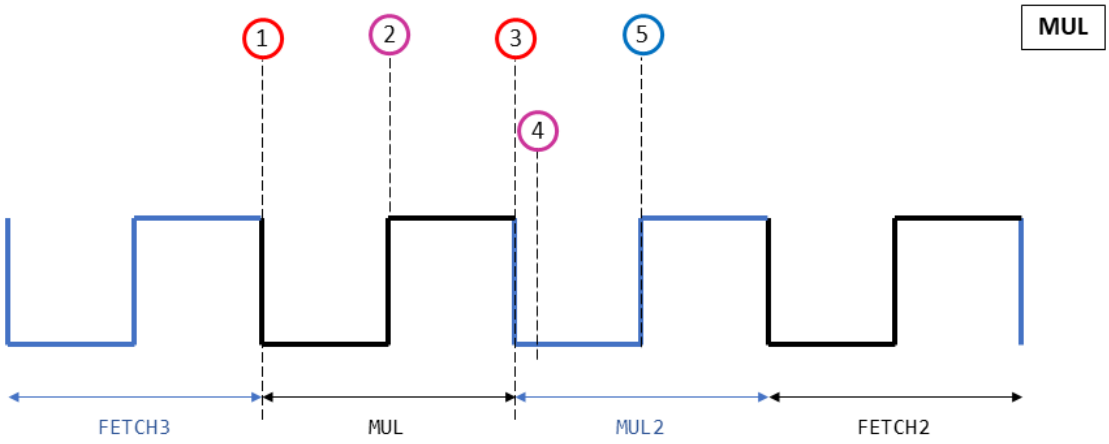
SUBT



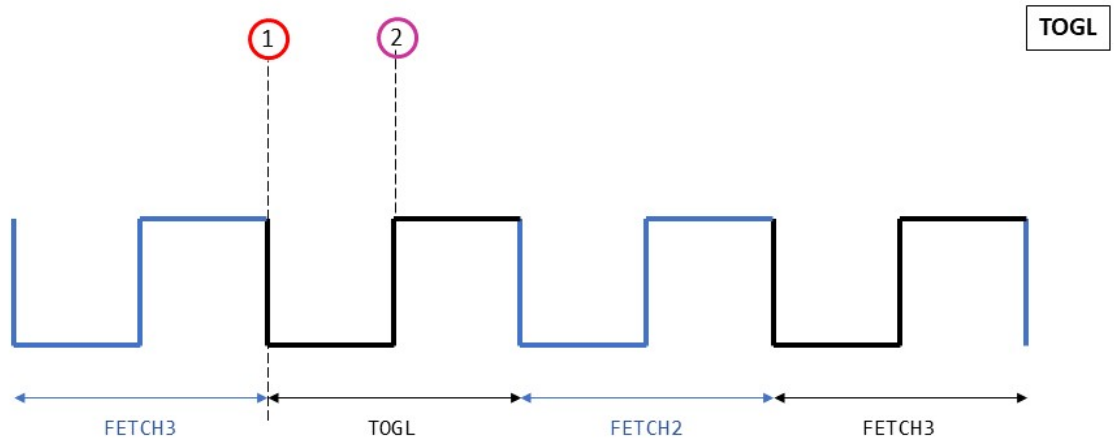
1) Control signal, ALU-SUBT is given.
   Control signal to PRM is given to rout parameter to AWM.
2) A-bus gets updated with the desired register value which needs to be added.
3) AC gets updated as, AC <- AC - A-bus



1) Control signal, PC increment is given.
   Control signal, MIDR write enable is given.
2) MIDR updates with operand.
   PC incremented.
3) Control signal, ALU-DIV is given.
   Control signal given to AWM to update A-bus with MIDR.
4) A-bus gets updated with MIDR.
5) AC gets updated as, AC <- AC / A-bus



1) Control signal, PC increment is given.
   Control signal, MIDR write enable is given.
2) MIDR updates with operand.
   PC incremented.
3) Control signal, ALU-MUL is given.
   Control signal given to AWM to update A-bus with MIDR.
4) A-bus gets updated with MIDR.
5) AC gets updated as, AC <- AC * A-bus



1) Control signal TOG is given.
2) Toggle register is flipped.

7 Hardware Debugging Features

There are 4 modes of operation (Controlled by [Key3]) for our design which are,

- 0: **IDLE**
- 1: **Rx mode**
- 2: **Processor mode**
- 3: **Tx mode**

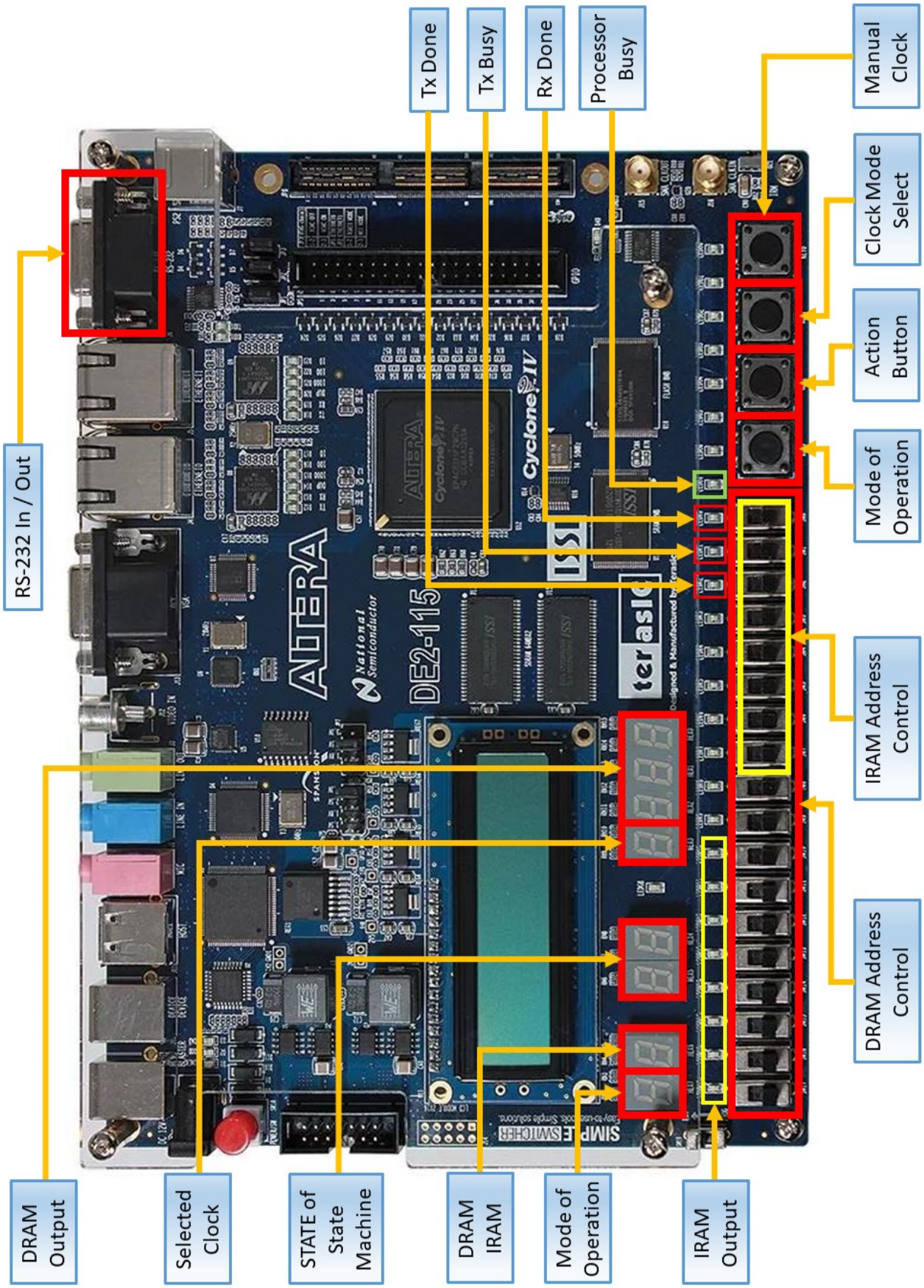
We have used some features of the DE2-115 development board in order to debug the design.

• In the IDLE mode and Rx mode we can view the Data / Instructions stored in the DRAM and IRAM by changing their address using the 18 switches.
• 2 Seven Segment Displays (SSDs) were used to display the current STATE of the "state machine" inside the processor.
• 3 Seven Segment Displays (SSDs) were used to display the data output of the DRAM for current address given to it.
• 8 LEDs were used to view the current output of the IRAM for current address given to it.
In order to debug the processor, we have implemented a way to run the processor in 4 different clock values,

- 0 : **10 MHz** (default)
- 1 : **1 Hz** - We are able to monitor state changes per second
- 2 : **Manual clock** – We can monitor state changes manually using a push button
- 3 : **25 MHz**

	Mode of Operation button (Key3)	Action button (Key2)	Clock Mode button (Key1)	Manual Clock button (Key0)
IDLE (0)	Cycles through modes	Change between IRAM and DRAM	10 MHz (won't change)	-
Rx mode (1)	Cycles through modes	Change between IRAM and DRAM	10 MHz (won't change)	-
Processor (2)	Cycles through modes	Start processing	Cycles between 10 MHz,1 Hz, manual clock, 25MHz	In "manual clock" mode , clock can be given manually
Tx mode (3)	Cycles through modes	Start transmission	10 MHz (won't change)	-

Board Layout



8 Problems Faced & Solutions

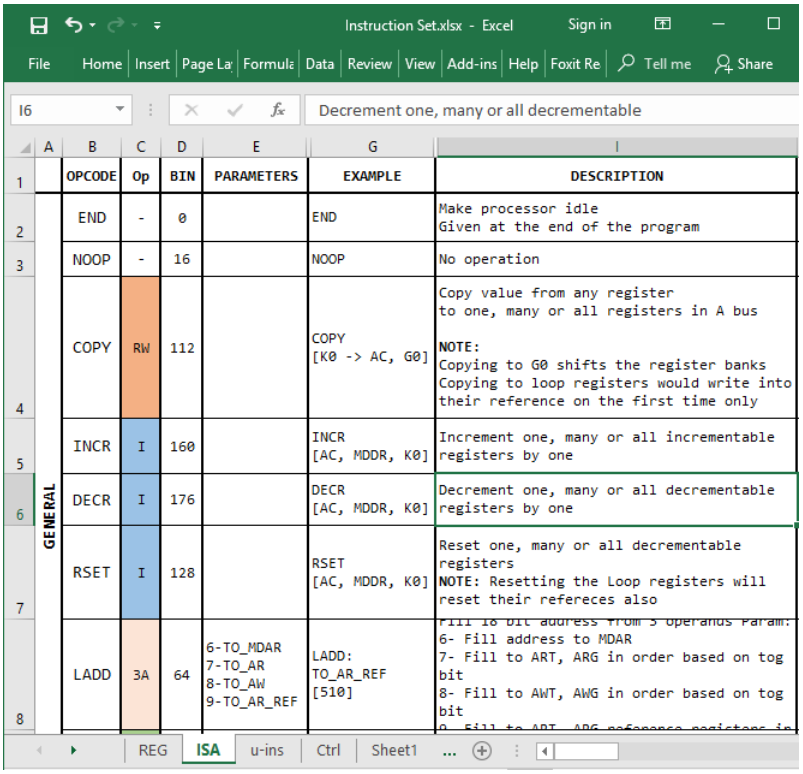
- Inability to manipulate 18-bit addresses on 8-bit buses. - We designed a unique architecture with a combination of 8, 9 and 12 bit registers to overcome this
- IP core not directly supporting 512 x 512 locations. - We designed a special memory module using four 256 x 256 RAMs
- Difficulty in developing algorithms when groups members are in different places - Hence we created the simulator for remote development.
- High clock speeds causing glitches. - We reduced the speed to 25 MHz
- Signed number operation problems. - We designed the ALU to perform absolute value calculations
- Number overflow and underflow. - We used a 12 bit ALU and AC to make sure 16 numbers can be added without overflow
- Simulation files causing conflicts with project files. - Took a long time to figure out. Afterwards, we did all of simulation and debugging directly on hardware.
- Lack of proper tutorials. - Hence we write this detailed report.
- Board hardware issues (rusted switches).
- UART cable connection issue.
- Long compile times. - Tested on smaller files and combined them together.
- Quartus causing computer crashes.
- Inability to simulate some IP core modules. - Hence we embraced hardware debugging.

9 Compiler and Simulator

ABRUTECH is supported by our custom-built robust compiler and a simulator, that allow the programmer to write algorithms with ease and to quickly eliminate syntax & runtime errors. Both are built using python with over 800 lines of code with the aid of python libraries such as Pandas, Numpy and OpenCV.

9.1 Workflow of the Programmer or Architect

Figure 9.1: Modifying the ISA on the excel sheet (optional)



		OPCODE	Op	BIN	PARAMETERS	EXAMPLE	DESCRIPTION
2	GENERAL	END	-	0		END	Make processor idle Given at the end of the program
3		NOOP	-	16		NOOP	No operation
4		COPY	RW	112		COPY [K0 -> AC, G0]	Copy value from any register to one, many or all registers in A bus **NOTE:** Copying to G0 shifts the register banks Copying to loop registers would write into their reference on the first time only
5		INCR	I	160		INCR [AC, MDDR, K0]	Increment one, many or all incrementable registers by one
6		DECR	I	176		DECR [AC, MDDR, K0]	Decrement one, many or all decrementable registers by one
7		RSET	I	128		RSET [AC, MDDR, K0]	Reset one, many or all decrementable registers **NOTE:** Resetting the Loop registers will reset their refereces also
8		LADD	3A	64	6-TO_MDAR 7-TO_AR 8-TO_AW 9-TO_AR_REF	LADD: TO_AR_REF [510]	Fill 16 bit address from 3 operands Param: 6- Fill address to MDAR 7- Fill to ART, ARG in order based on tog bit 8- Fill to AWT, AWG in order based on tog bit

The architect can adjust the ISA on the excel sheet by adding new instructions, removing old ones, changing the binary code for an instruction...etc.

Figure 9.2: Starting the python based Compiler & Simulaor

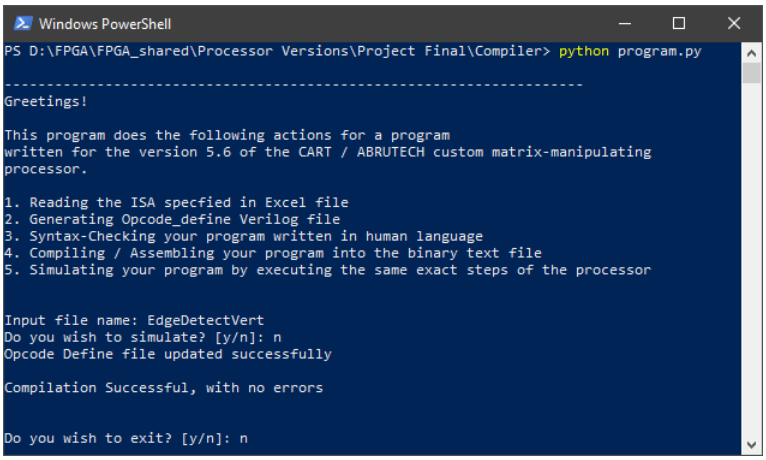
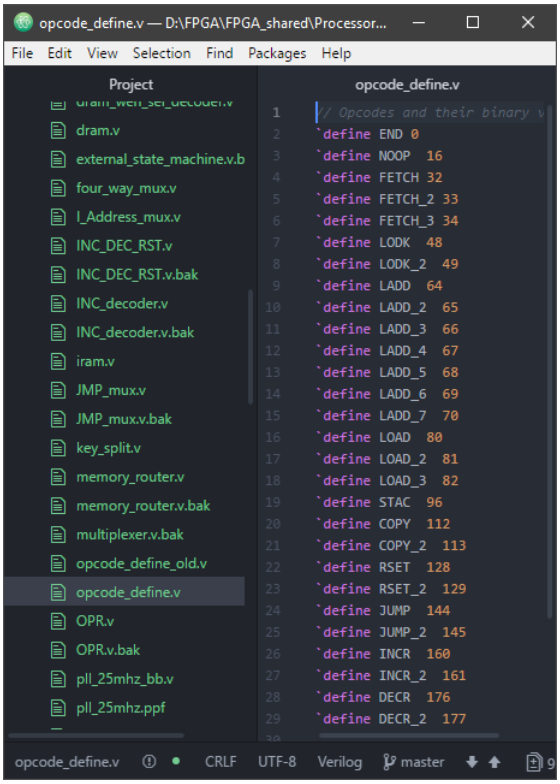
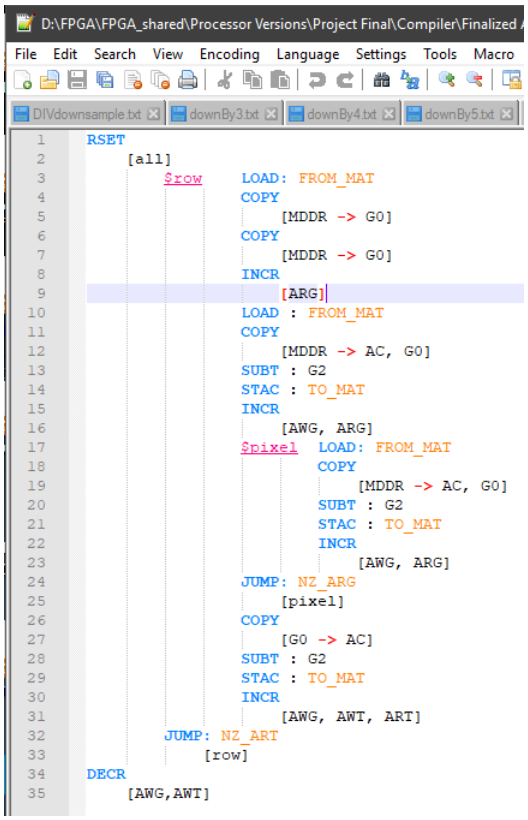


Figure 9.3: Auto-Generated 'opcode_define.v' Verilog file



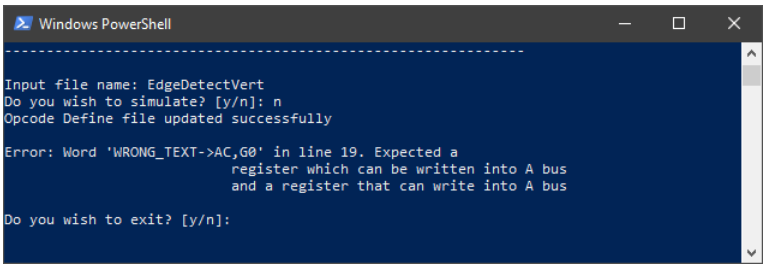
Our python script parses the excel sheet and automatically generates the above verilog file: 'opcode_define.v' which maps micro instruction (state names) to state numbers using the verilog preprocessor directive 'define. This file is 'include-ed in each verilog file. It makes sure we do not have to rewrite the entire state machine code when making minor changes in ISA.

Figure 9.4: Writing a program as .txt file (Edge Detection Program)



The program is written using human understandable language, with indentations, loop reference names and comments, into a text file

Figure 9.5: Debugging Using Syntax Checker



The compiler understands the human readable syntax and produces an array of integers as output. Syntax errors are reported with line number, word and description of the problem, making debugging easy. The programmer can recompile after correcting the reported syntax errors.

Figure 9.6: Program compiled into an array of integers

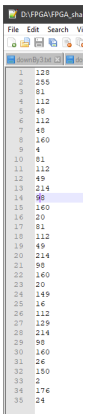
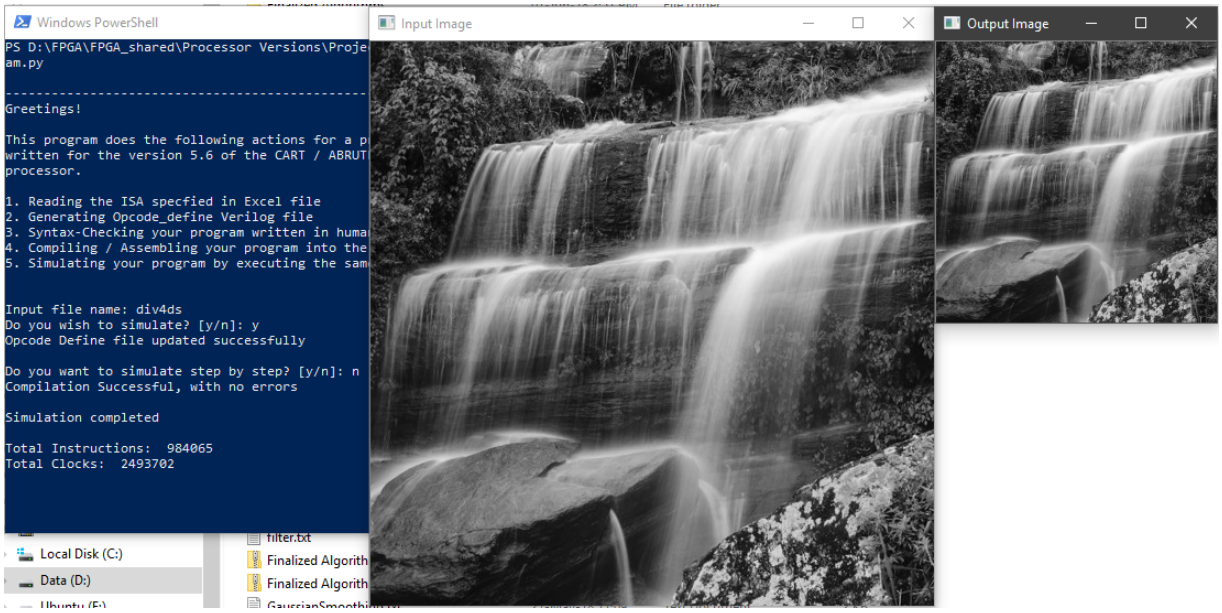
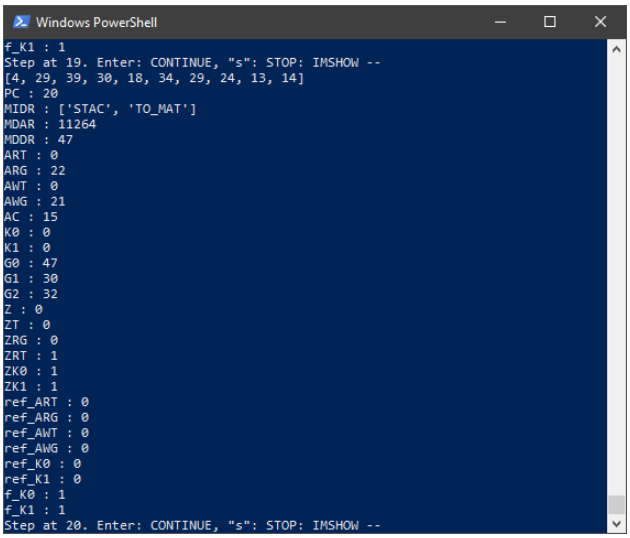


Figure 9.7: Full Simulation Mode: Simulated Output (Downsampling Program)



Optionally, the program is loaded into the simulator, which runs the program through the exact steps of the processor and outputs the state of all registers, memory, input and output images, the number of clock cycles required for the program...etc at the end of simulation. This is to identify the runtime errors in the program.

Figure 9.8: Step by step simulation mode, observing simulated register and memory values



If the expected result is not obtained in simulation, the program can be simulated step by step, or it can be paused at predefined breakpoints to visualize the memory and register status in intermediate steps.

The program text can be corrected to eliminate bugs, and recompiled and re-simulated iteratively until expected outcome is obtained in simulation.

Finally the list of integers produced by the compiler can be loaded into the processor using MATLAB through UART and it would produce the exact same result as produced by the simulator.

9.2 How the Compiler and Simulator Work

The main program: *program.py* calls different functions from different custom built python-modules available as independent python files.

9.2.1 Parsing the ISA

The first step is to parse the Excel sheet where the ISA is defined as a large table. The *read_ isa.py* module uses the pandas library to parse the excel sheet into a dataframe and then into a python-dictionary.

9.2.2 Making the define file

make_ define.py modules creates an empty .v file and writes into it with verilog syntax, filling it line by line with the instruction name and the respective opcode binary.

In the state machine verilog code, we use the opcode-names such as 'COPY, 'LOAD...etc. and we include the opcode_define.v in that file. The 'define preprocessor directive replaces these names with respective opcode-numbers when compiling the code.

This means, we do not need to change the state machine verilog file every time we update the ISA.

9.2.3 Compiling

To compile a program written in our syntax, *compile.py* module (about 300 lines long) is used. It first calls the parsing module and obtains the latest ISA as a python-dictionary. It then checks for the availability of the text file and starts reading it line by line.

In each line, spaces and tabs are removed and all words are converted to uppercase. This means, our language is not case sensitive and not sensitive to the presence of whitespace. Also, all comments (anything that follows a # sign) are removed.

Based on the previous line, it is determined that whether the current line is an opcode or operand. Then the compiler matches the words in the line with different dictionaries to produce the output.

If there is no error, each line is converted into an integer that is written into a file named: *binary.txt*. We did not bother using a .hex file or a binary format, since having the output as 10-based integers help us to quickly check if the compiler is working correctly.

If anything unexpected is encountered, the compiler throws an exception, printing the current line number, current word and the error description and then it terminates. The programmer can correct the error and compile again.

9.3 Simulation

The simulator of our processor is a unique and powerful tool we built for easy debugging and remote development of algorithms.

```
92 def TOGL():
93     if(reg['ZT'] == 0):
94         reg['ZT'] = 1
95     else:
96         reg['ZT'] = 0
97     INPC()
98
99 def LOAD():
100    if(param == 'FROM_ADR'):
101        pass
102    elif(param == 'FROM_MAT'):
103        if(reg['ZT'] == 0):
104            reg['MDAR'] = reg['ARG']*512 + reg['ART']
105        else:
106            reg['MDAR'] = reg['ART']*512 + reg['ARG']
107    else:
108        print("ERROR. Parameter mismatch in LOAD")
109
110    reg['MDDR'] = d_mem[int(reg['MDAR'])]
111    INPC()
```

Inside the 400-lines long *processor.py* module, a dictionary is defined with names of registers their initial values as key-value pairs.

A set of python functions are defined to do the exact task of the processor, by following the exact same steps, while counting the simulated clock cycles. They modify the values of the dictionary to simulate the changing values of registers in the processor.

The simulator first calls the compiler module and obtains the program as a python-list of strings that is free of syntax errors. Then it iterates through the list, calling the respective functions (mapped using a dictionary and python's function-handler objects).

The output image is shown using OpenCV library.

10 | Communications

10.1 Overview

One of the most important external parts of the project is the communication system. Even if the processor is very accurately designed, if the communication system is weak, it cannot function properly. Therefore, we had to design an efficient way to allow the processor to communicate with a computer. The system mainly has 3 vital components 1. Communication controller system on FPGA 2. UART over RS232 link 3. MATLAB program running on a PC The communication controller is built around a modified version of our semester 4 FPGA project to build a simple UART link. Two separate modules are in charge of handling the Rx and Tx lines, respectively deserializing and serializing data in the standard UART protocol specifications.

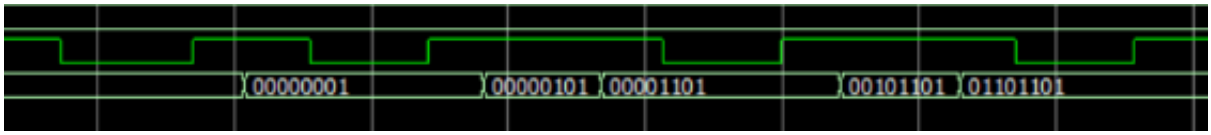


Figure 10.1: Typical UART data unit

Given above is a typical UART data unit, 8 bits of data between a start bit and a stop bit. The serial wire is maintained at logic 1 at idle. When a byte is transmitted, first the line is pulled down for one-bit period, indicating the start. Then the data is transmitted, LSB first. After data byte is sent, the line is held high for one-bit period, signaling byte end. Our unit operates at a master clock of 10MHz and a baud rate of 115200, allocating 87 clocks per bit period. We selected this speed because higher clock speeds and baud rates caused certain errors in the data streams. These Tx and Rx units have a special "tick" line that remains high for one clock cycle after a successful transmission/reception. These tick lines are then used by the controller to properly guide the data to and from the Rx/Tx modules. The data lines from the Rx/Tx modules are routed to DRAM and IRAM data buses by the controller as necessary. When there is a data transmission or reception, DRAM/IRAM address buses are in control of the communication controller, properly advancing through the addresses and putting the data in correct order, whether it is writing to the RAM or reading from the RAM.

10.2 Transmitter

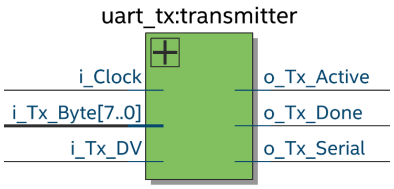


Figure 10.2: UART Transmitter

The Uart tx module takes an 8-bit data unit as input (i Tx Byte). After receiving the send command on i Tx DV, which is simply pulling the line high for a clock cycle, the module serializes the data as shown in the above diagram and sends them through o Tx Serial. Once the transmission is done, the o Tx Done signal goes high for a clock cycle, indicating the controller that it is ready for the next byte. O Tx Active shows when the serial line is in use. I clock is the 10 MHz clock input.

10.3 Receiver

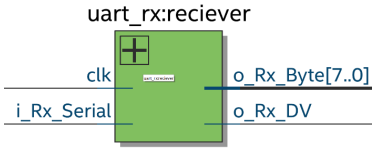


Figure 10.3: UART Receiver

Uart rx module takes a serialized data input (i Rx Serial) and outputs it to an 8-bit wide data bus(o Rx Byte). Whenever a new data set is ready, o Rx DV tells the controller to read this data from the bus. This module also takes a 10 MHz clock input from clk.

10.4 Data Retriever

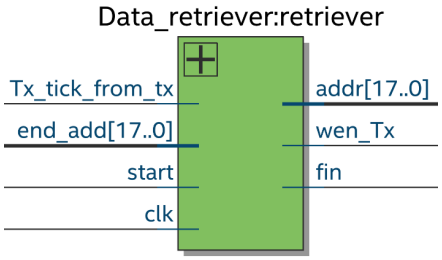


Figure 10.4: Data Retriever

Data retriever module is the outbound communication controller. It takes the end address for the data and sets it as the limit for the data transmission. When the system is in transmit mode, the DRAM address is already assigned to the addr bus of this module and the DRAM data to the data input of the uart tx. It starts from address zero, puts the wen Tx line high for a clock

cycle, which connects to the DV line of the uart tx. Uart Tx transmits the data from the first memory location and drives the Tx Done high, which is connected to Tx tick of this module, commanding the address to increment by one and the wen Tx to go high again. This cycles through all the locations of the DRAM. But if the end address is smaller than the full DRAM depth, there is a separate module inside that prevents the data out of range from transmitting.

10.5 Tx Modifier / Cropper

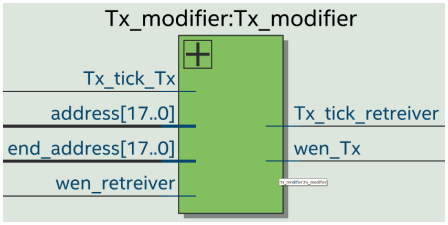


Figure 10.5: Cropper

Tx modifier is what safeguards the communication system from transmitting out of range data. It essentially 'crops' the image by assigning a valid address range to the DRAM, derived from the end address

	A	B	C	D	E
1	0				
2					
3			end		
4					
5					

Figure 10.6: Valid Address range

Let's assume the above grid is the DRAM, with 25 locations. We give the end address as 12, which is square C3, then the image we need extracting lies in the green squares and the rest is junk data. What the tx modifier does is splitting the RAM address into two parts, first part signifying the row an second part signifying the column. These parts are then compared with the row and column parts of the end address, which in this example are C and 3. Whenever the address is out of the range, the tick line and tx DV lines are 'hijacked' by this module, automatically incrementing the tick and holding the DV low until a valid address comes in.

10.6 Data Writer

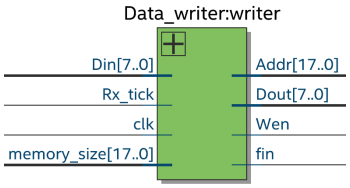


Figure 10.7: Data Writer

Data writer is the inbound communications controller. It starts with a memory size that indicates when to stop receiving. The data bus of the uart rx is directly connected to the Din of this module. When the system is in receiving mode, both DRAM address and data buses are assigned to this module. Starting from memory location zero, at every receiving 'tick' of the rx, it puts the data on the memory data bus and when the data is ready, gives the write signal to the ram and after the data is written, increments the address by one and writes the next incoming byte. When the end address is reached, it stops writing and signals the reception end via the fin line.

10.7 Serial Link

The Rx/Tx serial lines from the communication controllers are routed to the inbuilt RS232 interface of the ALTERA DE2 115 board. This interface connects to a USB virtual serial port on a PC through a special cable which has a USB – UART converter built-in. Windows PCs can listen in to this interface through a special driver CP2103 which converts the USB link to a classic serial port. The cable has RX and TX lines as well as the support lines for other Serial functions such as DTR and CTS. These connections are managed by the serial converter itself and holds no significance to us. When the data is properly placed on RX and TX lines, transfer happens automatically.

10.8 MATLAB script

The USB virtual serial port is handled on the PC by a MATLAB script. This script has the capability to,

1. Transmit/receive a monochrome image
2. Split a color image into RGB layers and transmit/receive layers separately
3. Load a hex code file to be sent to the instruction memory

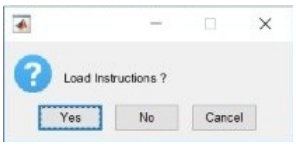


Figure 10.8: Load instruction option

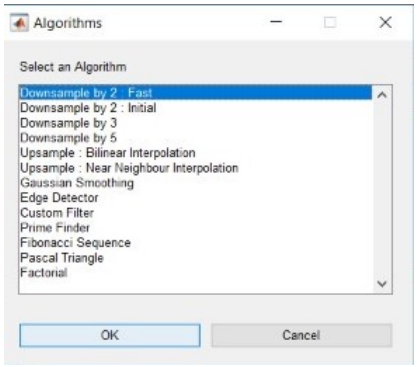


Figure 10.9: instruction Algorithm select

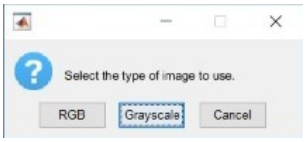


Figure 10.10: RGB/Monochrome mode selection

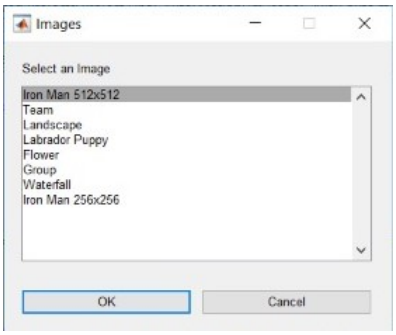
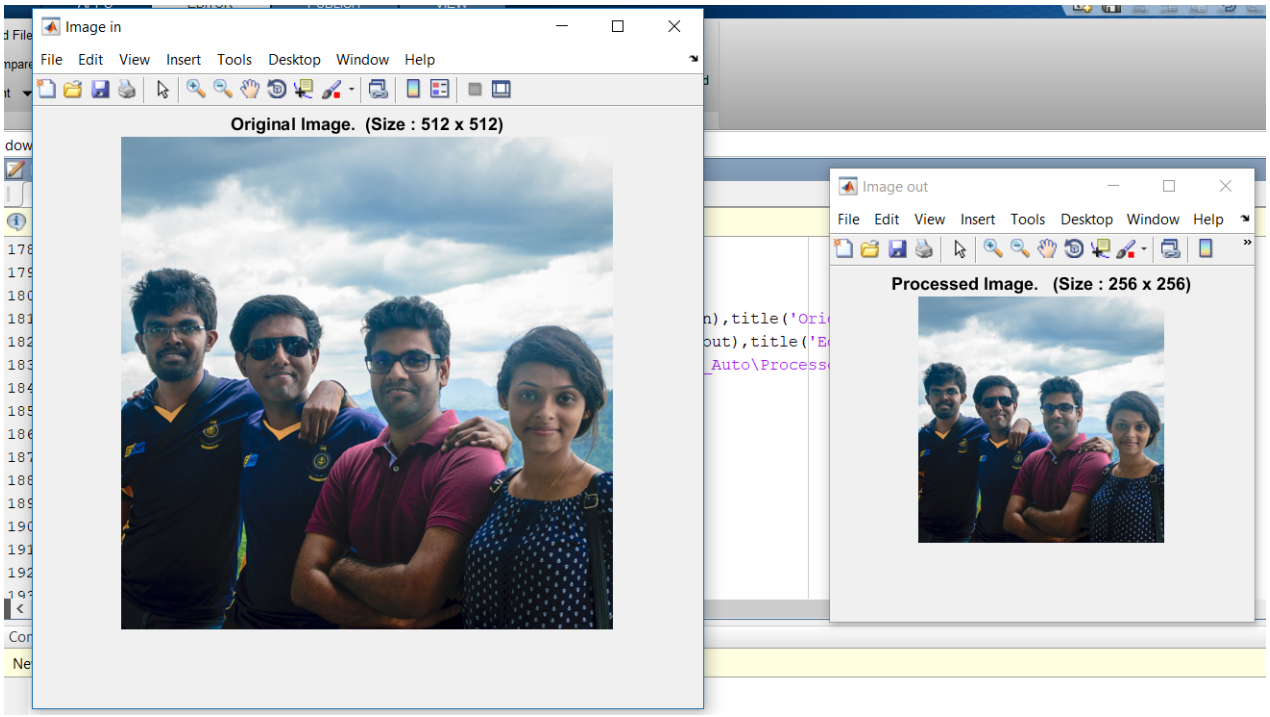


Figure 10.11: Image selection dialog

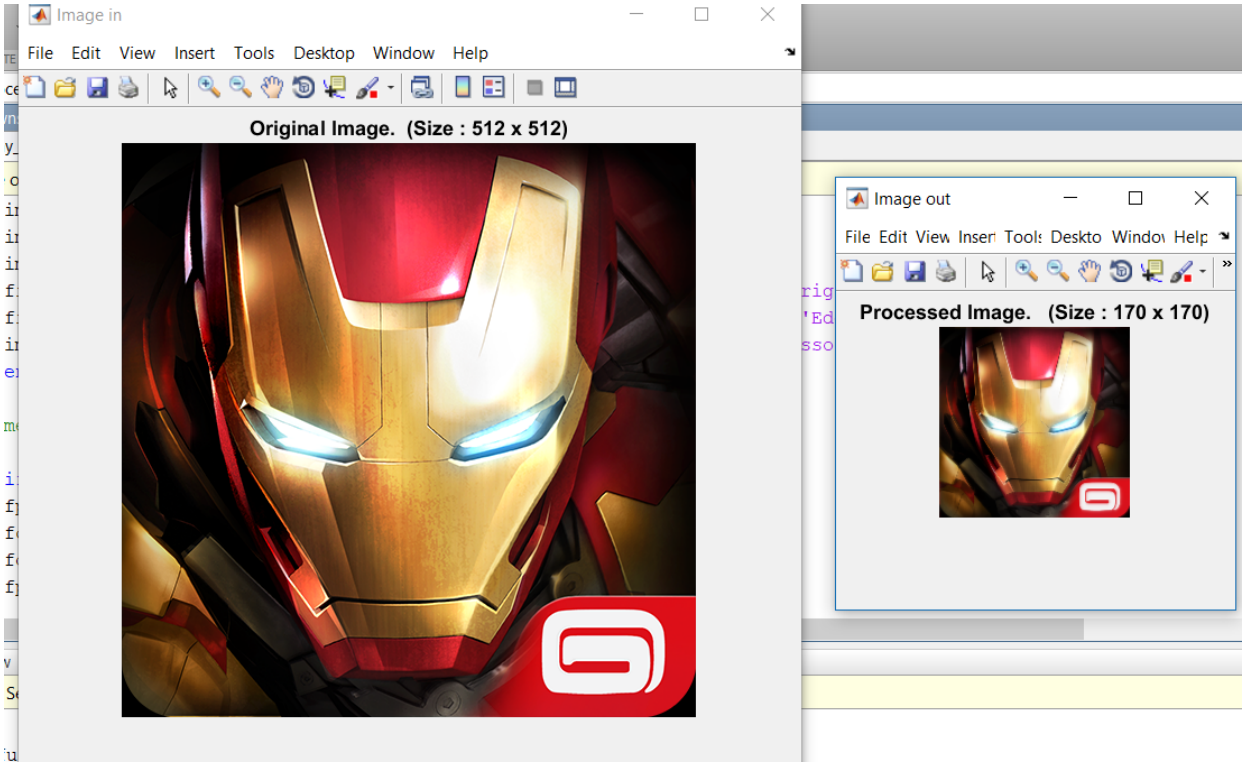
And also, since our processor is capable of generic programs, the script is easily customizable to handle any kind of required transmission/reception. The source code for the script can be found in the appendix.
The physical controls for the communication controllers are wired to the push buttons of the DE2 115. These switches include transmission start, receive ready and mode select controls.

11 | Results

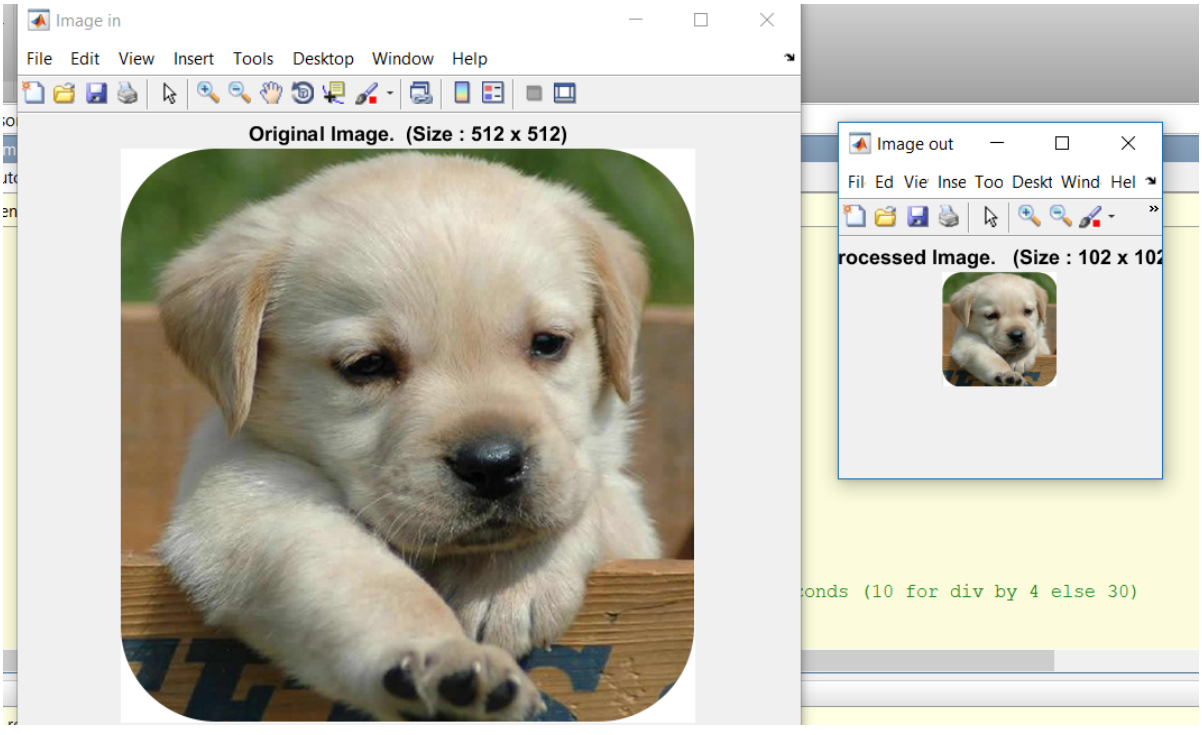
11.1 Downsample by factor 2



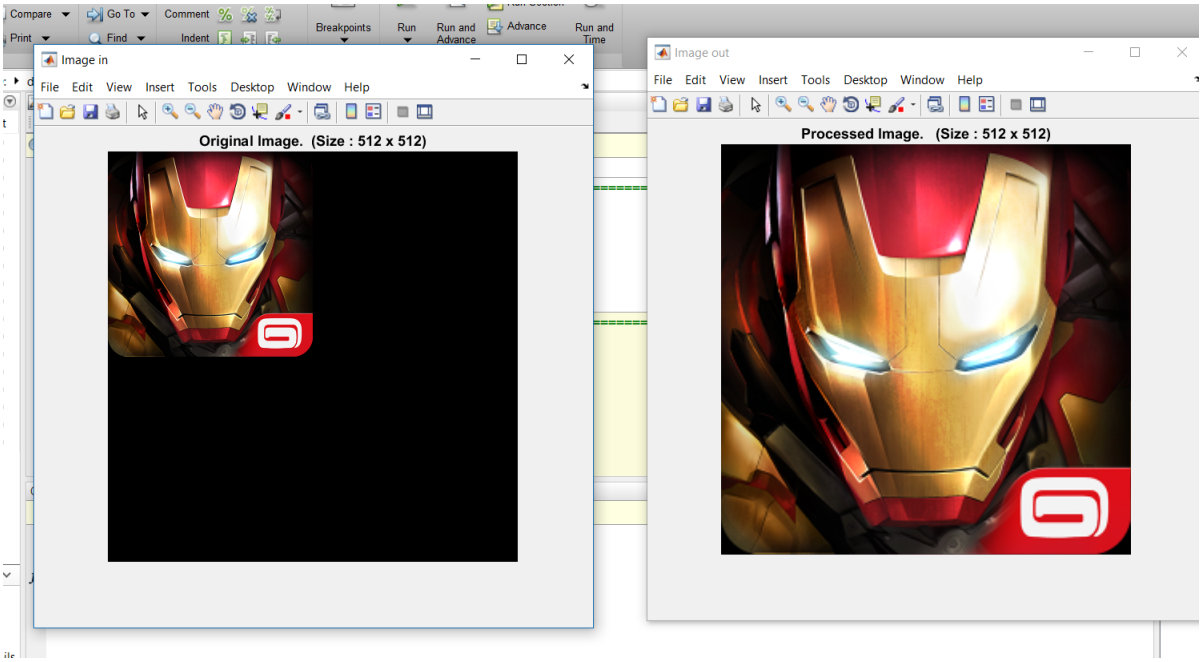
11.2 Downsample by factor 3



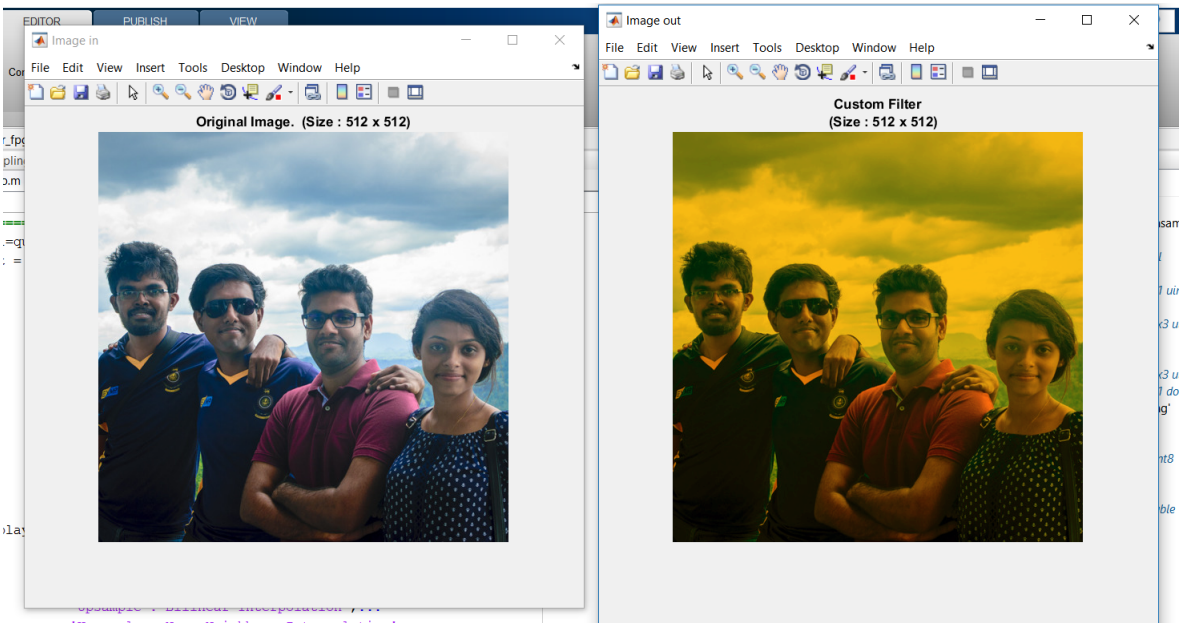
11.3 Downsample by factor 5



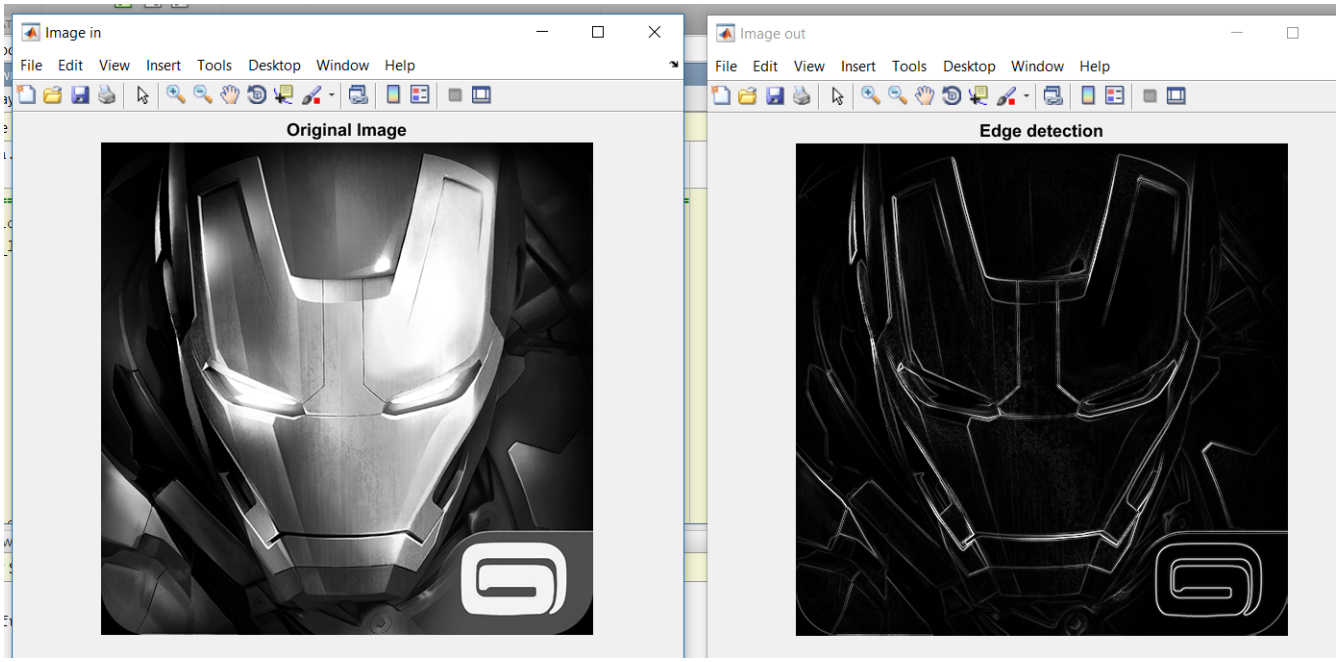
11.4 Bilinear Upsampling



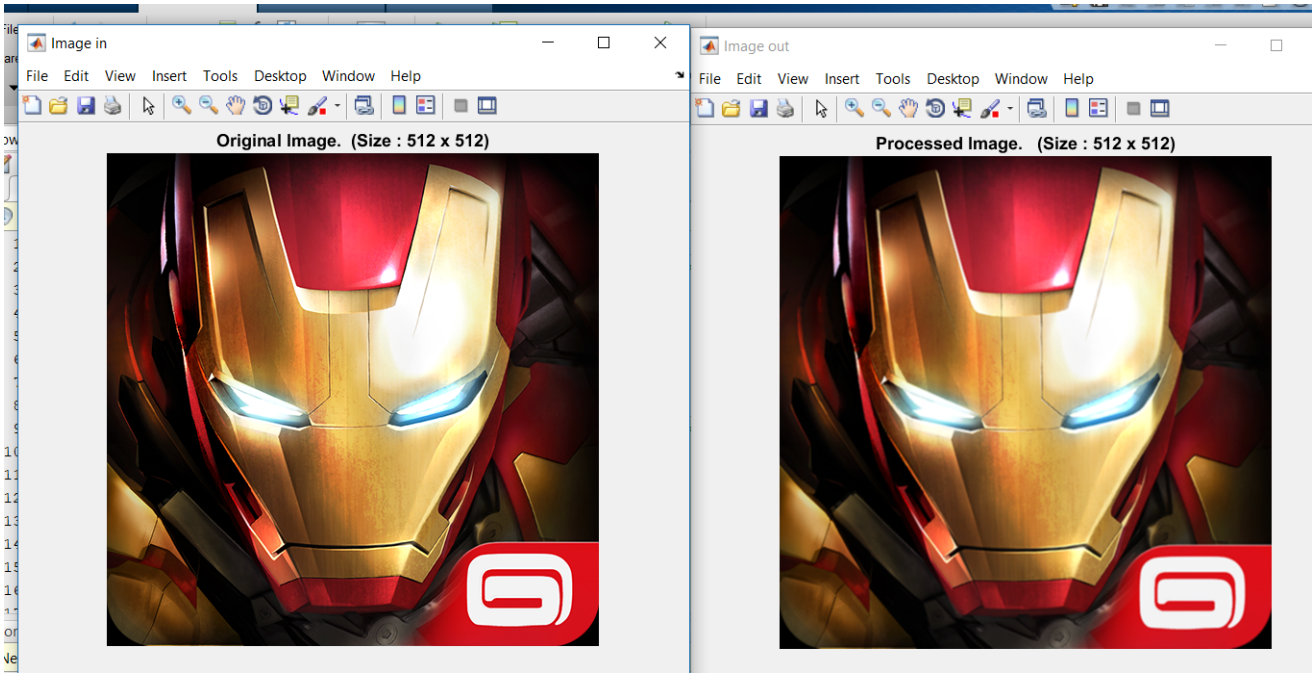
11.5 Custom Filtering



11.6 Edge Detection



11.7 Gaussian Smoothing



11.8 Prime finding

primes

6x9 double

1	2	3	4	5	6	7	8	9
2	3	5	7	11	13	17	19	23
29	31	37	41	43	47	53	59	61
67	71	73	79	83	89	97	101	103
107	109	113	127	131	137	139	149	151
157	163	167	173	179	181	191	193	197
199	211	223	227	229	233	239	241	251

11.9 Pascal Triangle

```
Received Data.

Pascal Triangle

           1
         1 1
        1 2 1
       1 3 3 1
      1 4 6 4 1
     1 5 10 10 5 1
    1 6 15 20 15 6 1
   1 7 21 35 35 21 7 1
  1 8 28 56 70 56 28 8 1
 1 9 36 84 126 126 84 36 9 1
1 10 45 120 210 252 210 120 45 10 1
fx >>
```

11.10 Fibbonacci Series

Fibonacci Series

File Edit View Insert Tools

Memory Location	Data
0	1
1	2
2	3
3	5
4	8
5	13
6	21
7	34
8	55
9	89
10	144
11	233