Ayush Patel
Moses Velez
ECE 469 : Lab 8

Lab 8: Pipeline CPU Design

Objective:
The objective of this lab was to design a dual speed-power mode Pipelined processor called EDU16. Additionally demonstrate the throughput of the Pipeline vs Non-Pipeline Design.

Design Process:
The CPU was designed with 5 pipeline stages: Fetch, Decode and Read, Execute, and Write Back. Along with these 5 stages the CPU contains a datapath that encomponses the Finite State Machines for each of the pipeline stages, ALU, Register File, Memory System, and Inter-State Registers (IR and PC registers to pass instruction and PC down to next stage, MAR and MDR). We started off with the following CPU design provided and enhanced it perform design a better and more efficient Datapath.

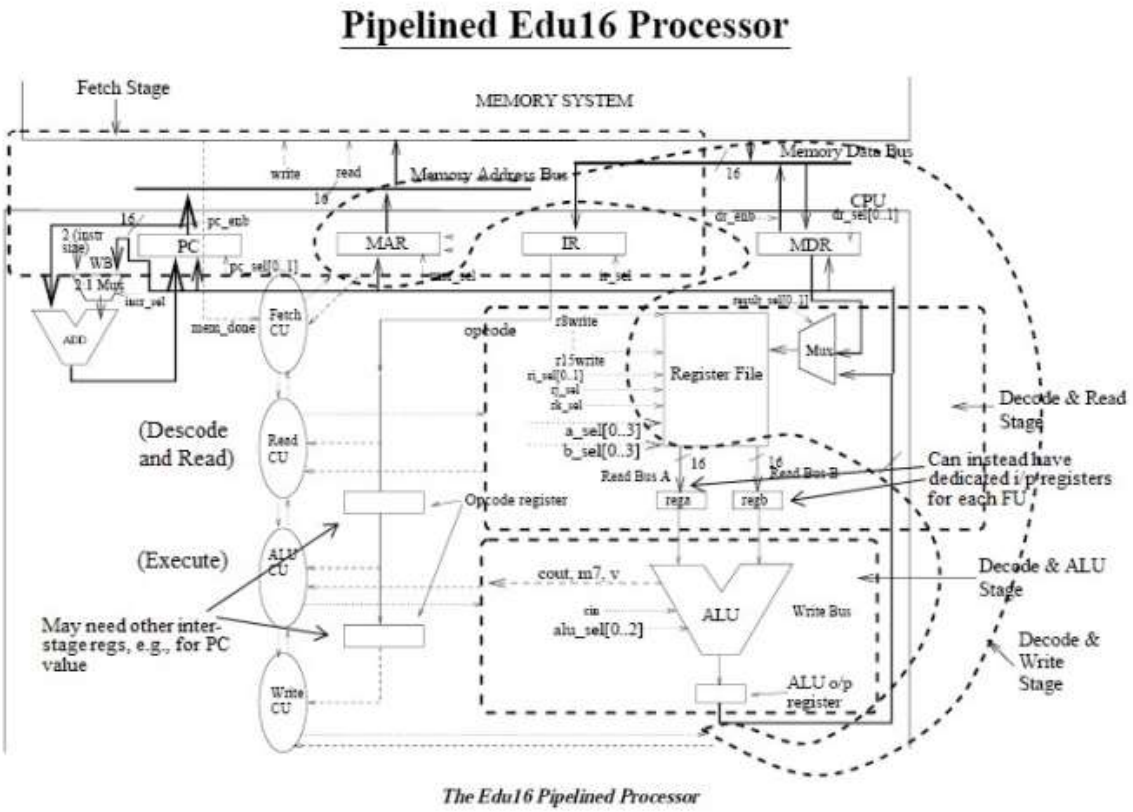


Figure 1: Pipeline Design Provided

Ayush Patel
Moses Velez
ECE 469 : Lab 8

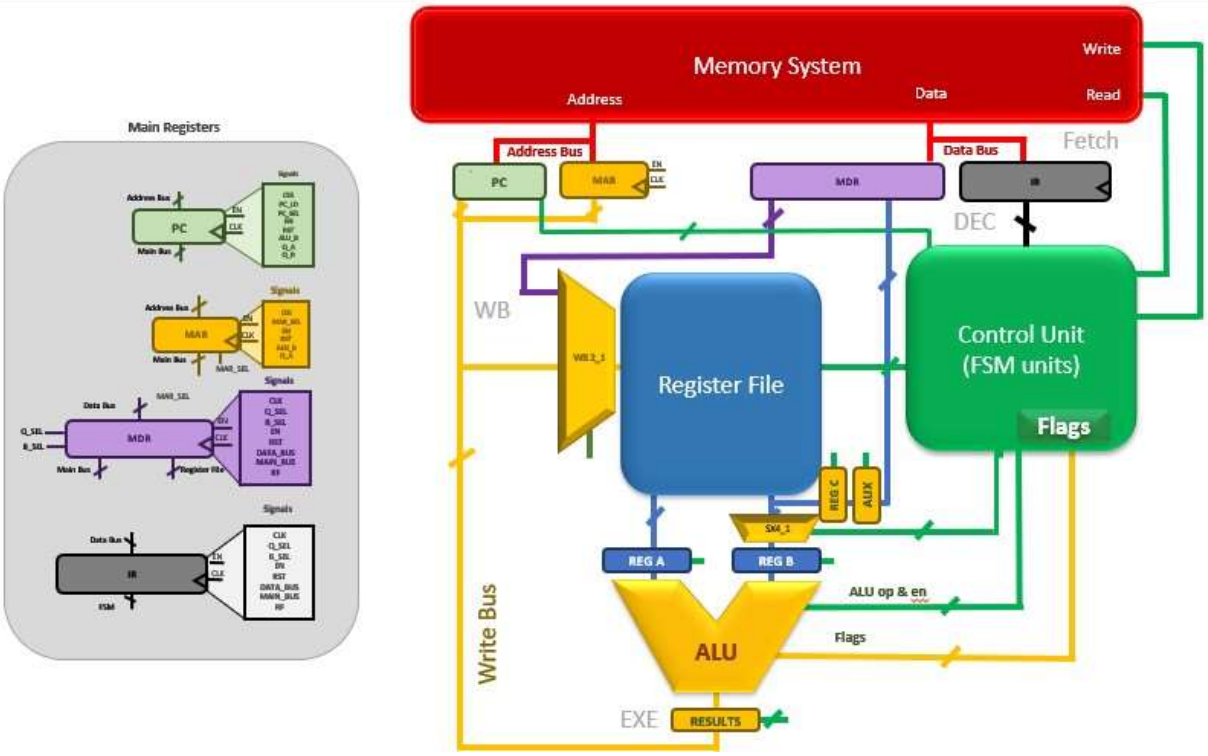
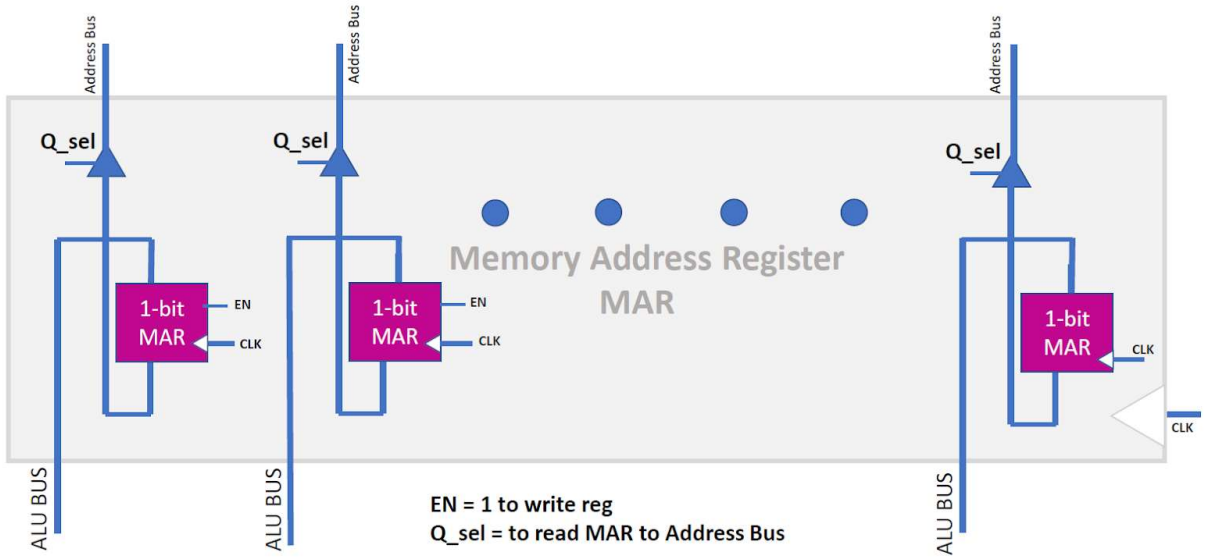
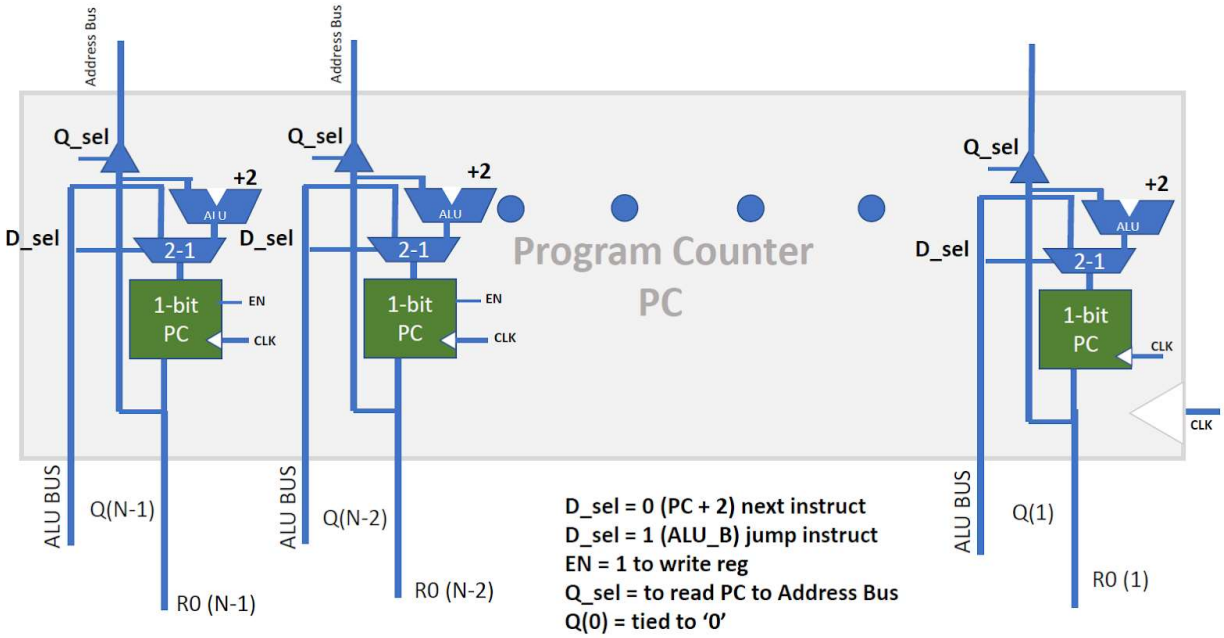


Figure 2: Datapath Redesigned + Registers

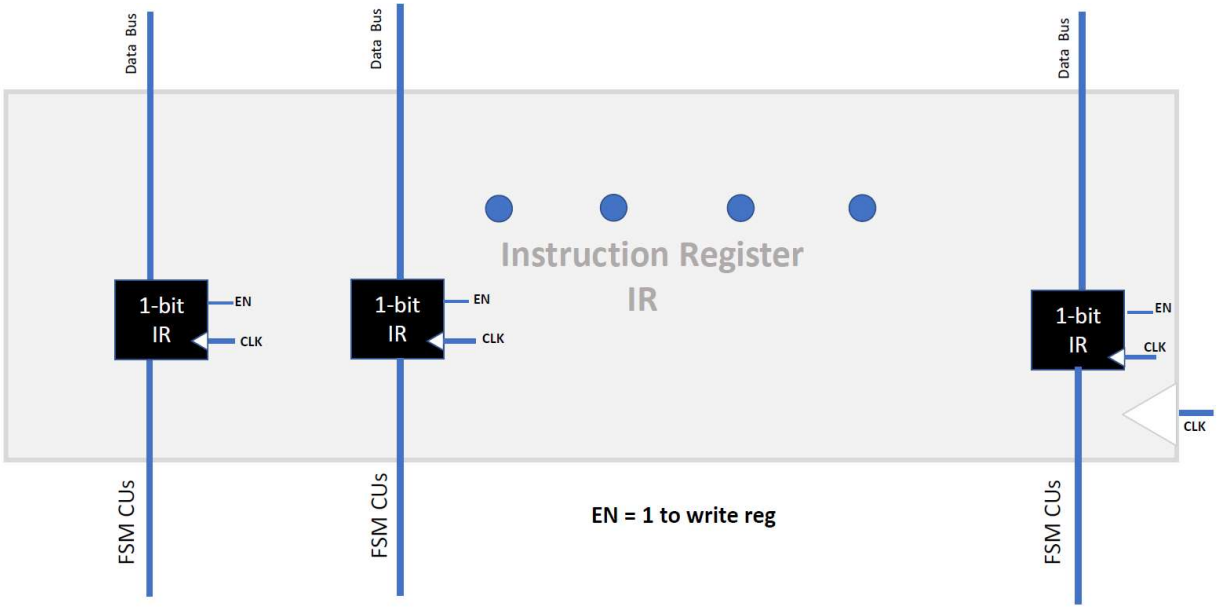
The Re-Designed datapath consists of a Control Unit that holds all the FSMs the different stages of Pipeline. There are also 4 registers what are shared by the entire CPU(PC, MAR, MDR, and IR). IR and PC registers shown in the picture hold the most recent information about CPU process while MDR and MAR are memory data and memory address registers used to Load and Store instruction and data from the Memory System. The following pictures show how the MAR, PC, IR, and MDR registers were designed and how they are controlled.

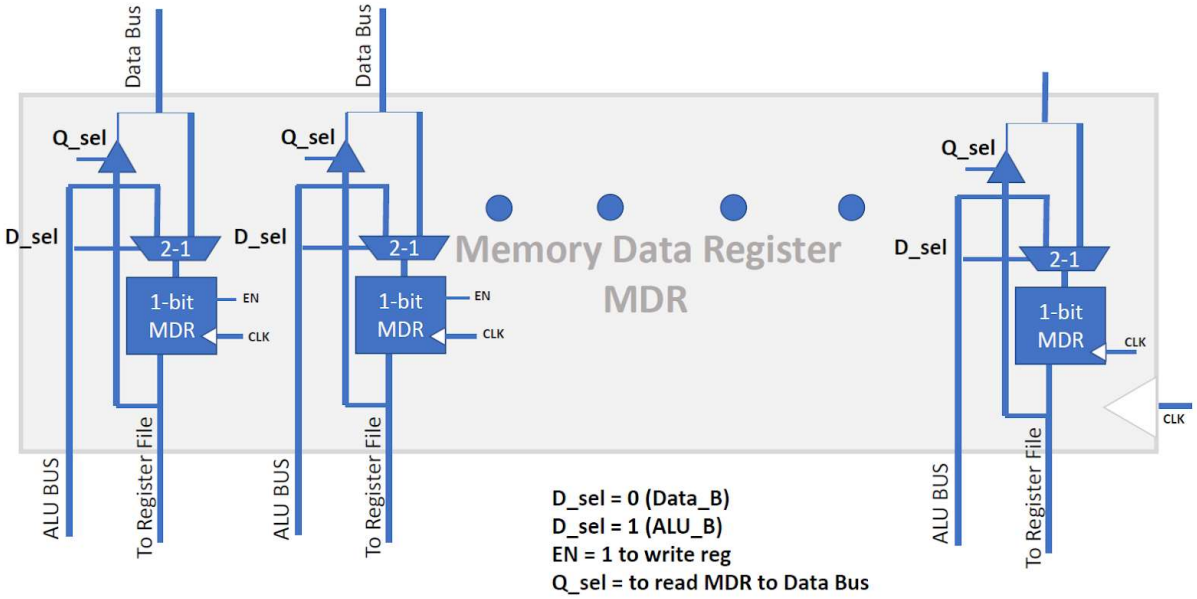
Ayush Patel
Moses Velez
ECE 469 : Lab 8





Ayush Patel
Moses Velez
ECE 469 : Lab 8

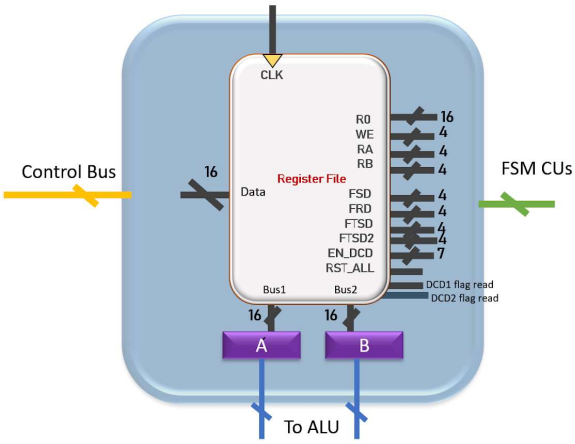




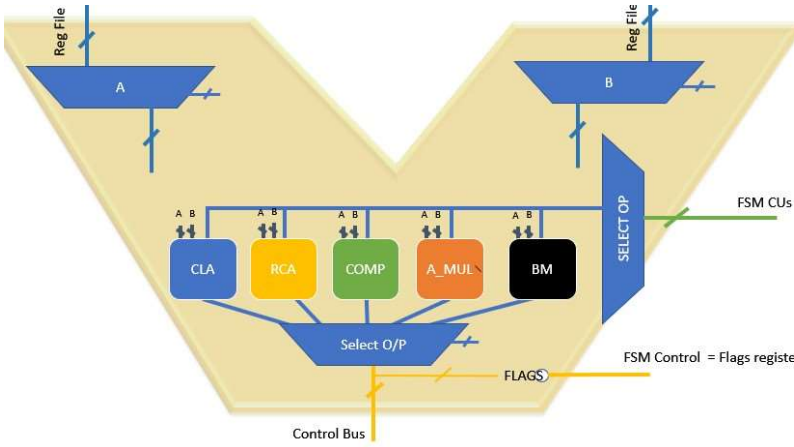
Ayush Patel
Moses Velez
ECE 469 : Lab 8

Below is some detailed description of the different components of redesigned Datapath provided above.

1. **Register File:**
   Register File holds 16 registers (R0 - R15) with R0 holding the PC value for incrementing and updating the PC counter while R1-R15 are used for data storage. A 16-bit 2x1 MUX is used to load data into the Register File that takes in the Write Bus(coming from the ALU) and MDR Bus(carrying data from Memory to MDR to Register File). Additionally there are 4 more registers used on the outside of a Register File to control what data goes to ALU (REG A/B) and MDR (REG C, AUX) when can that data be sent to the respective component using a control signal.



2. **ALU:**
   The ALU contains 4 types of FUs -- Fast Adder (CLA), Slow Adder (RCA), Fast Multiplier (Array Mult) and Slow Multiplier (Booth Mult). Having 4 FUs ranging from fast to slow allows us to operate the CPU in Fast-High Power and Slow-Low Power modes. When Using Fast-High Power mode only Fast Adder and Multipliers are used for all the Operations going to ALU (CLA and Array Mult) and similarly when running in Slow-Low Power mode only Slow Adder and Multiplier is used (RCA and Booth). Additionally there is the ALU is also capable of setting Flags for Compare operations (>, <, =, negative, and zero check). These Flags are used when performing Branch instructions.



3. **Control Unit:**
   The Control Unit holds all the Finite State Machines for the Pipeline stages (Master, Memory, Fetch, Decode, Execute, and Write Back). A detailed design connections are provided in the picture below followed by a detailed FSM design for each of the Stages/Units required for Pipelining CPU.

Ayush Patel
Moses Velez
ECE 469 : Lab 8

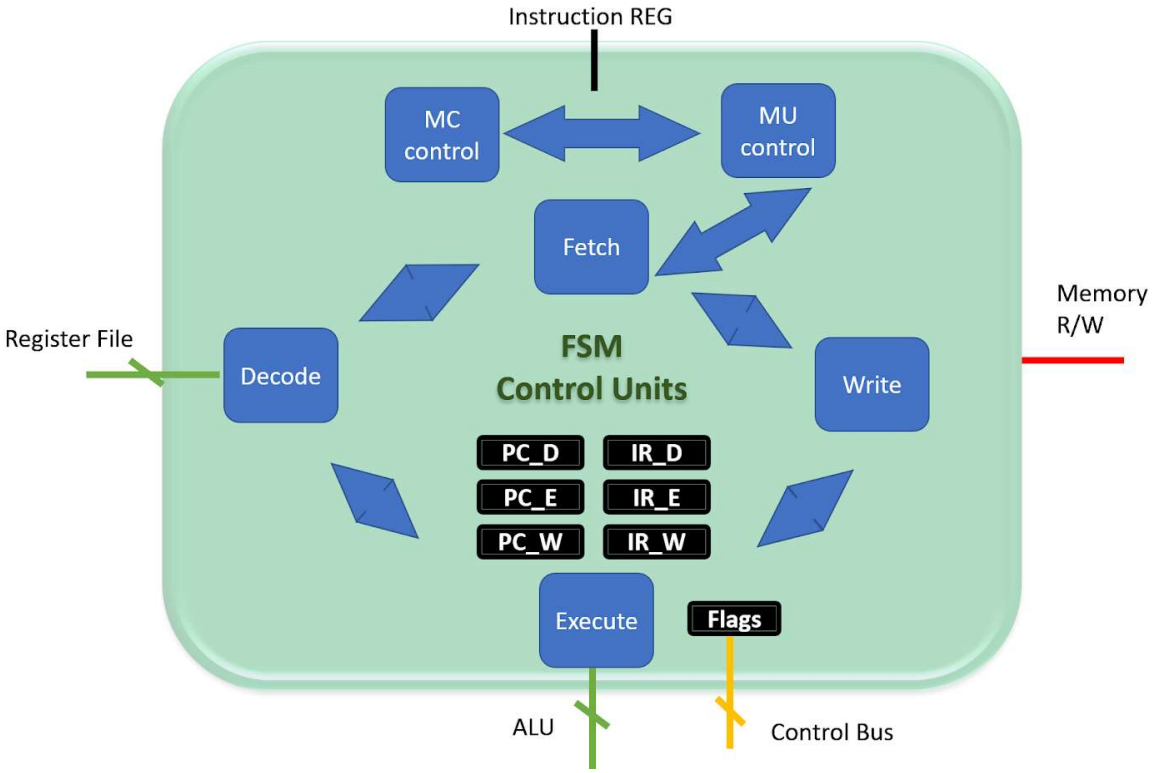


Figure: Control Unit Top Level View + FSM Controls+ Inter-Unit Regs

Master Control unit

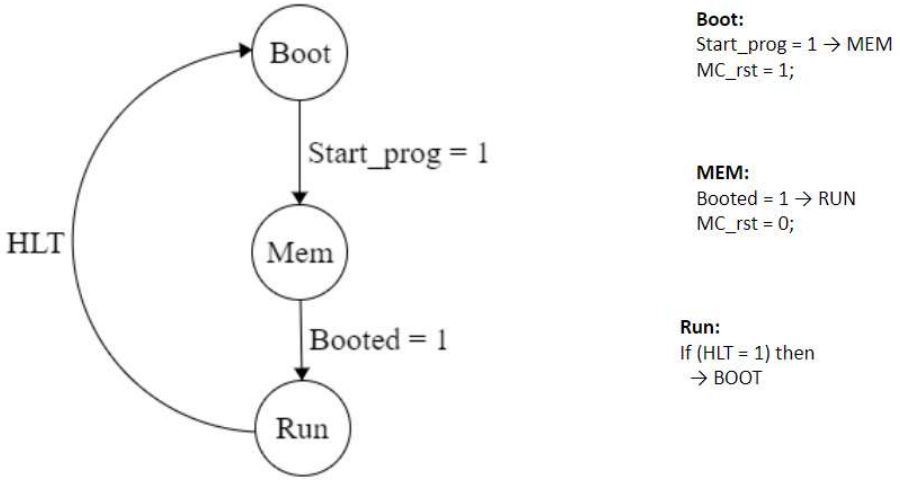


Figure: Master Control Unit

Ayush Patel
Moses Velez
ECE 469 : Lab 8

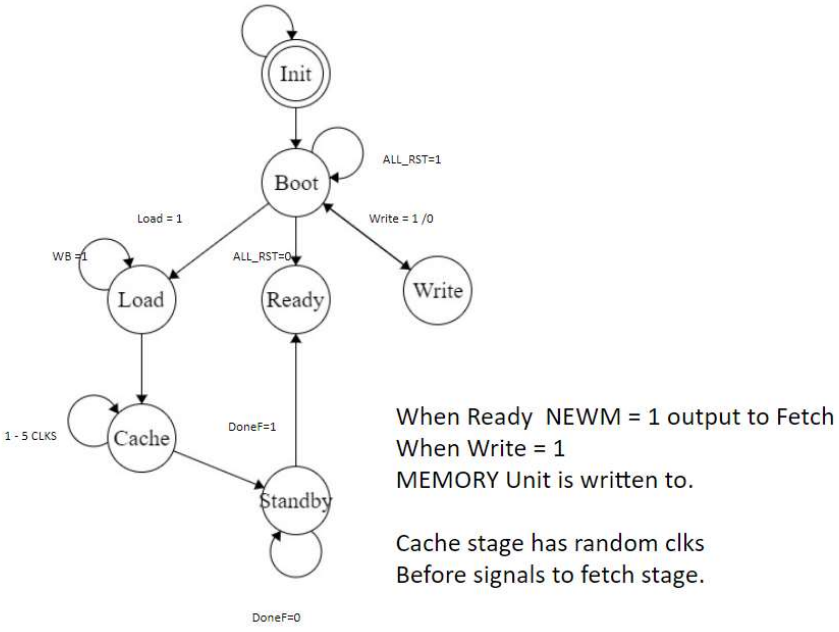


Figure: Memory Unit

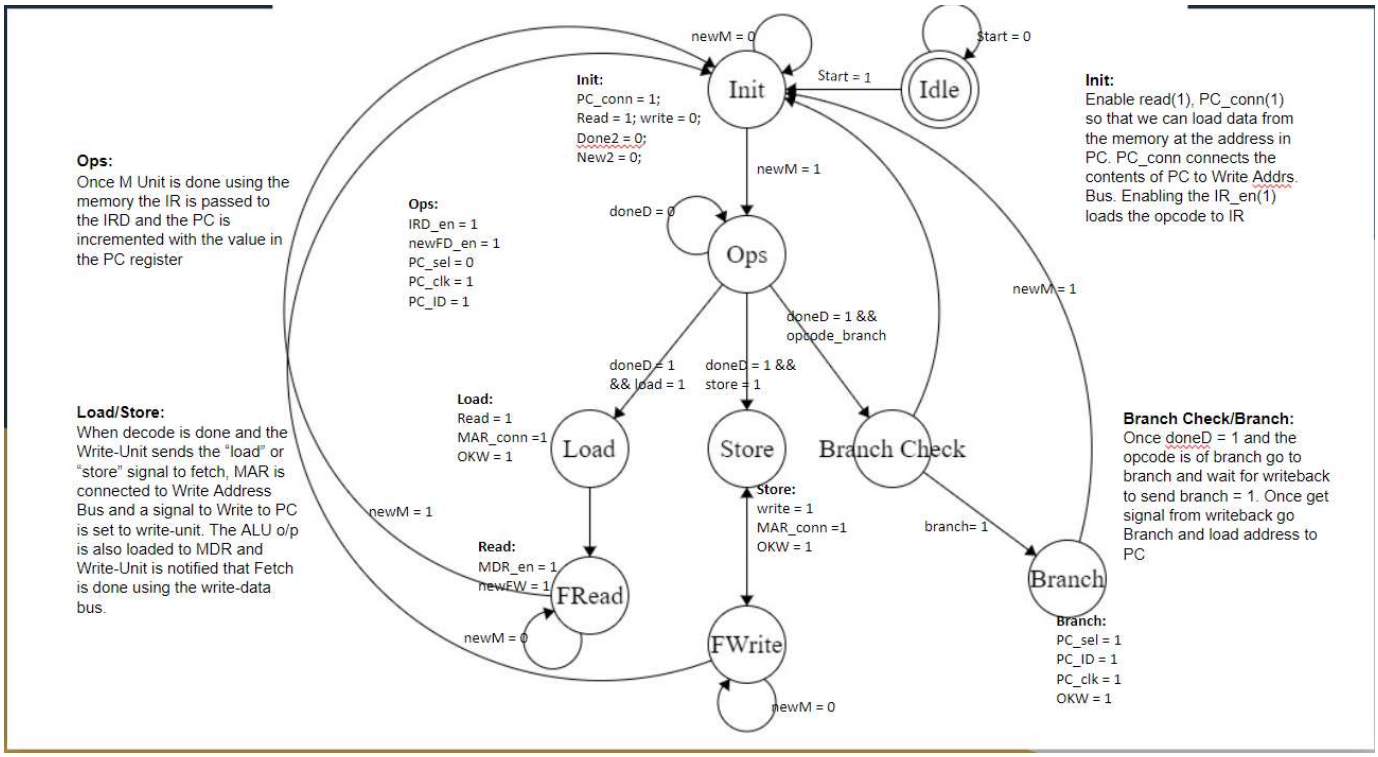


Figure: Fetch Unit

Ayush Patel
Moses Velez
ECE 469 : Lab 8

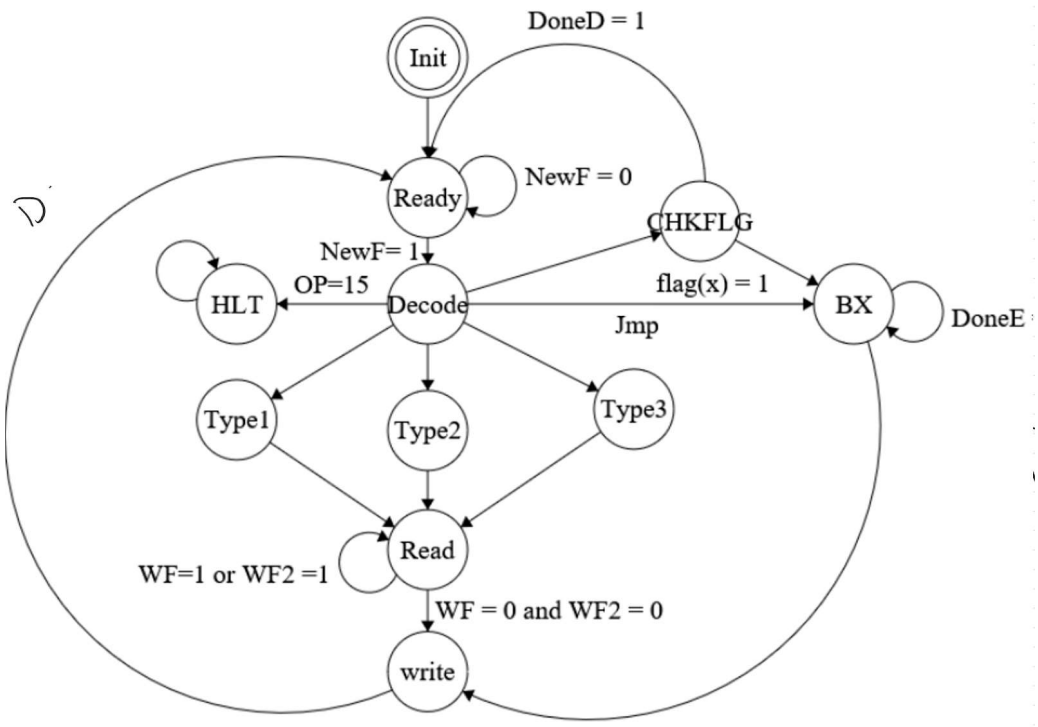


Figure: Decode and Read Unit

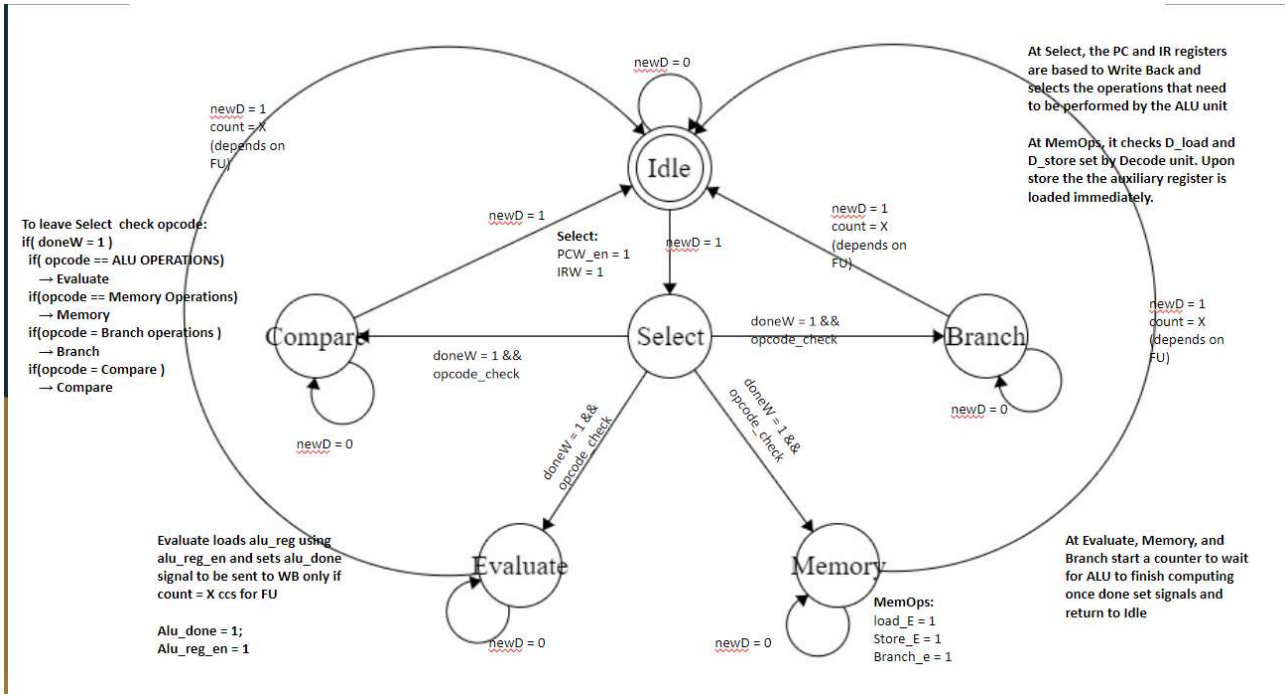


Figure: Execute Unit

Ayush Patel
Moses Velez
ECE 469 : Lab 8

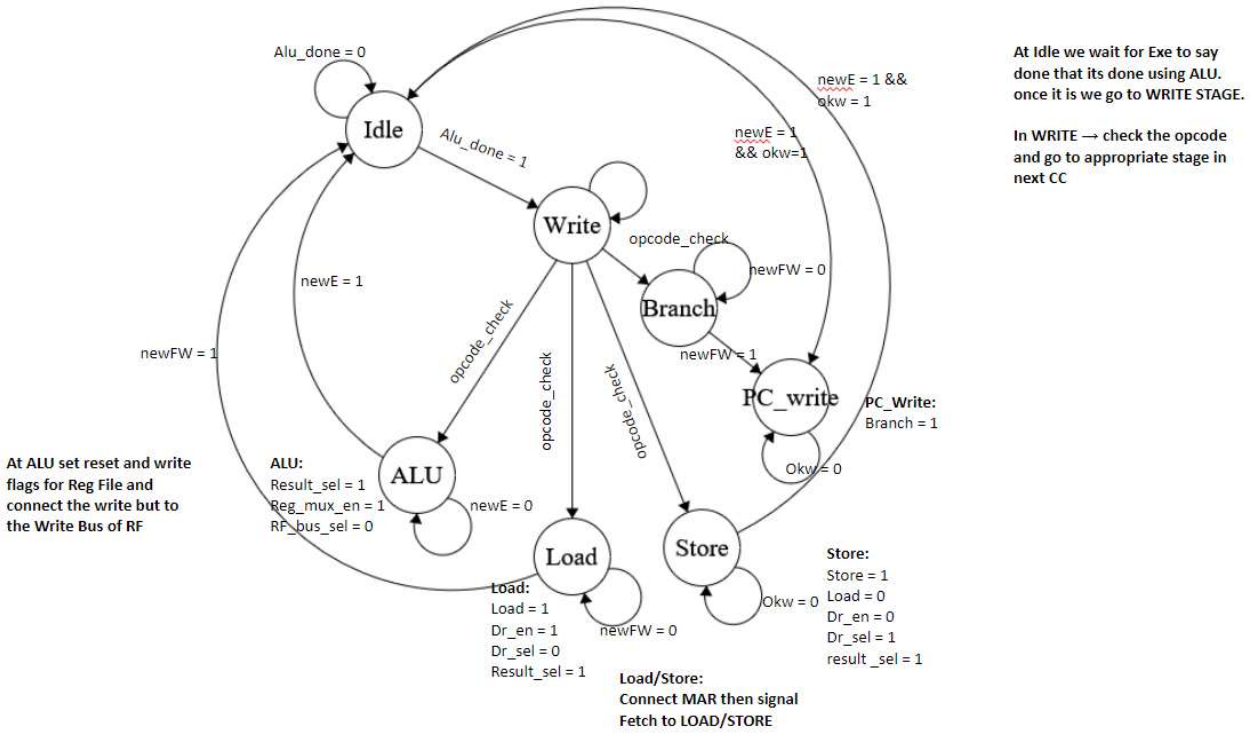


Figure: Write Back Unit

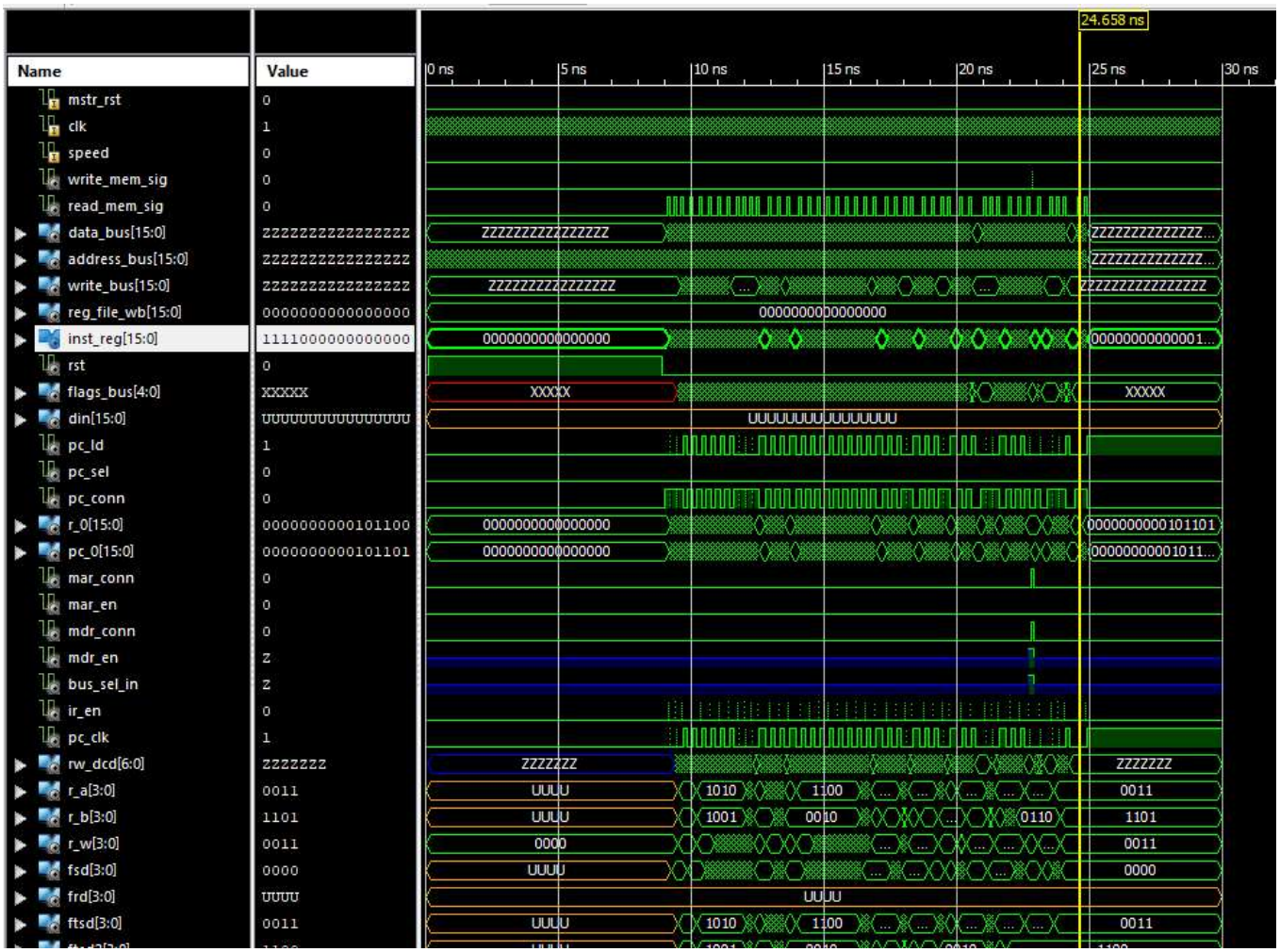
Test and Simulation:

Minimum Possible Clock Period:

Minimum Possible clock Period for our Machine was close to **50ps**. We determine this value by trial and error analysis of the provided program with smaller **t-clk < 100 ps.** The initial assumption was 100ps period because RCA takes about 90ps to finish and we decided give it some extra time. With **100ps we the Matrix Program provided finished in about 40ns or 40,000ps.** However with a period of **50ps we were able to finish the program in about 25ns or 25,000ps**. The picture below is of the timing diagram of the Matrix-program with a period of 50ps on a Fast-High Power Processor. We can see that the program reaches the end at about 25ns (highlighted inst_reg signal)

Ayush Patel
Moses Velez
ECE 469 : Lab 8

Figure: Matrix Program with Clock Period of 50ps



Ayush Patel
Moses Velez
ECE 469 : Lab 8

Non-Pipeline Delay for Instructions

The Non-Pipelined delay for each Non-HALT instruction is provided in the table below. For non specific instructions such as Add Immediate, Branches, Load and Store we used CLA. The clock period for each of the instruction was 100 ps.

Instruction	Non-Pipeline Delay
ADD_F (CLA)	**1 ns**
ADD_S (RCA)	**1.8 ns**
ADD_I (ADD Immediate)	**1 ns**
MULT_F (Array Mult)	**2.5 ns**
MULT_S (Booth Mult)	**3.5 ns**
Jmp (Branch Unconditional)	**1.2 ns**
BGT (Branch Greater Than)	**1.2 ns**
BLT (Branch Less Than)	**1.2 ns**
BEZ (Branch Equal To)	**1.2 ns**
BZ (Branch Equal to Zero)	**1.2 ns**
BN (Branch Negative #)	**1.2 ns**
LD (Load)	**1.3 ns**
LDi (Load Immediate)	**1 ns**
STR (Store)	**1.4 ns**

As we can see from the above table most of the generic instruction that used the same adder (CLA) ended up with about the same time that is approximately 1 ns. The Array Multiplier took about 2.5 ns with 100 ps period while the Booth Multiplier takes approximately 3.5 ns to complete computation and storing into register file. Similarly Load and Store instruction takes just over 1 ns to finish since they will be engaging the ALU and the MDR and/or MAR registers along with the Memory System which has a default delay of about 20 ps. A sample of the Assembly code used to do this testing is provided in the Appendix.

Ayush Patel
Moses Velez
ECE 469 : Lab 8

Fast-High Power Mode:

Below are couple screenshots of two programs we ran on Fast-High Power Mode(First is Matrix Program provide and second is a custom program we wrote for testing) with CLA and Array Multiplier. The overall delays are also listed under the picture.

Figure: Fast-High Power CPU Design + Matrix Program

Using the Fast-High Power mode with 100 ps clock period we found the Tcc completion time to be about **40 ns or 40,000 ps for Matrix Multiplication Program.** This equated to be about **1 second in real time**.

Ayush Patel
Moses Velez
ECE 469 : Lab 8

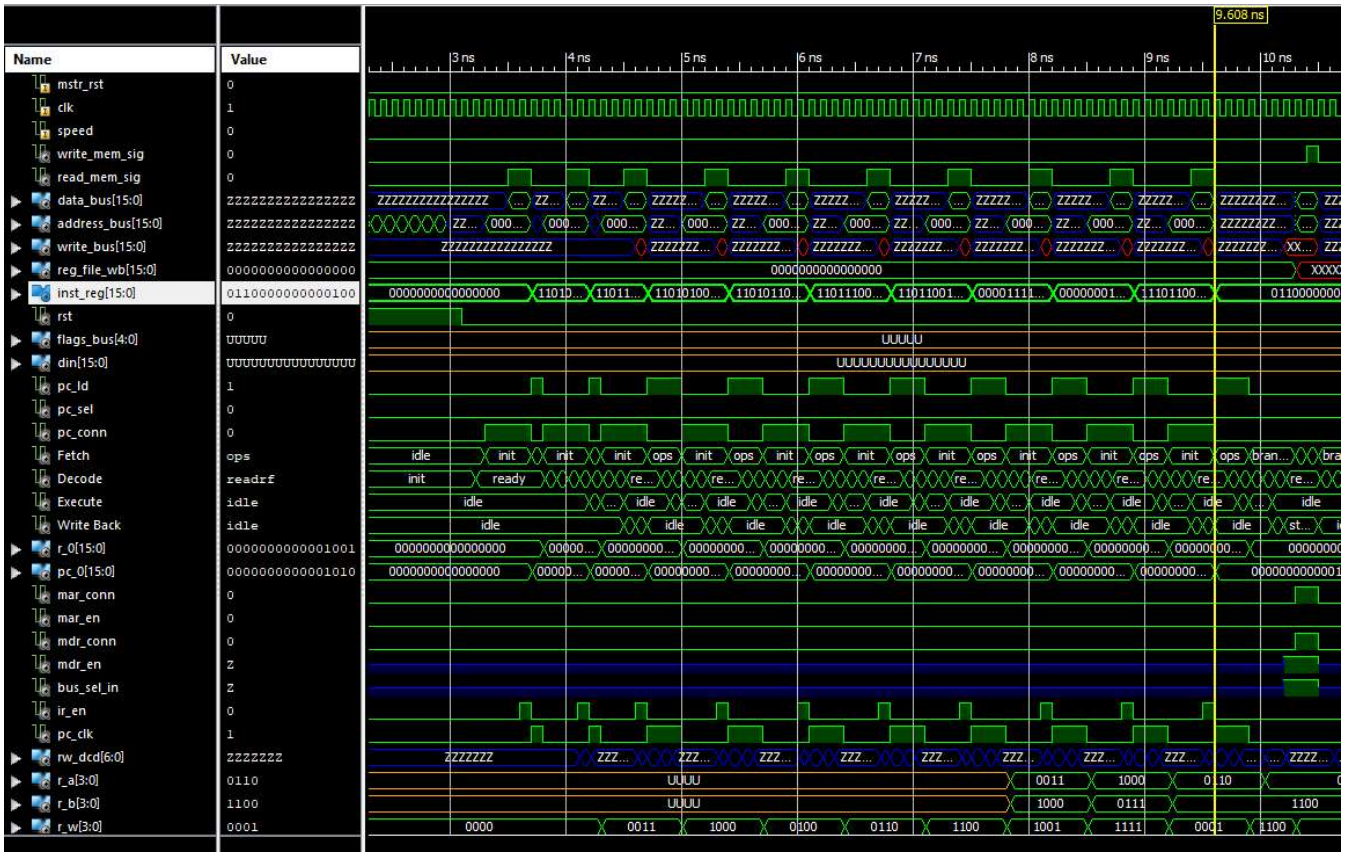


Figure: Fast-High Power CPU Design + Test Program

The above picture is of the of a custom program we wrote to test each individual instruction (Adds, Multipliers, Branch, Load and Store). This program had about 10 instructions and was able to finish about **10 ns or 10,000 ps.** This equated to about **0.5 seconds in real time.**

Ayush Patel
Moses Velez
ECE 469 : Lab 8

Slow-Low Power Mode:

Below are the screenshots the same two programs as mentioned above but now running in Slow-Low Power mode with RCA and Booth Multiplier.

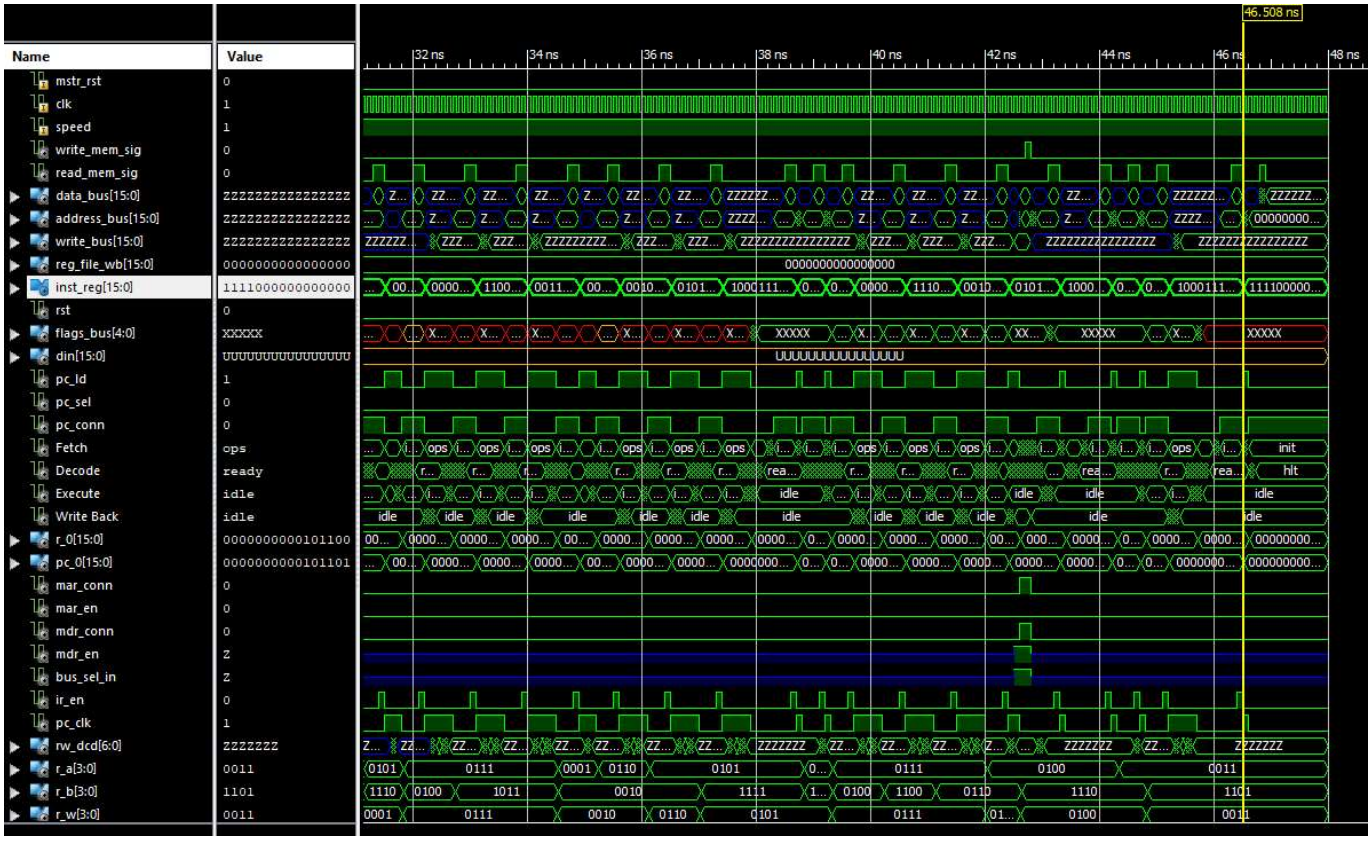


Figure: Slow-Low Power CPU Design + Matrix Program

The above picture of the Matrix Program provided. This program in Slow-Low Power mode takes about 46 ns or 46,000 ps to finish from start to end. This equates to about 1.5 seconds in real time. This shows that using RCA and Booth slows down the processor and save power.

Ayush Patel
Moses Velez
ECE 469 : Lab 8

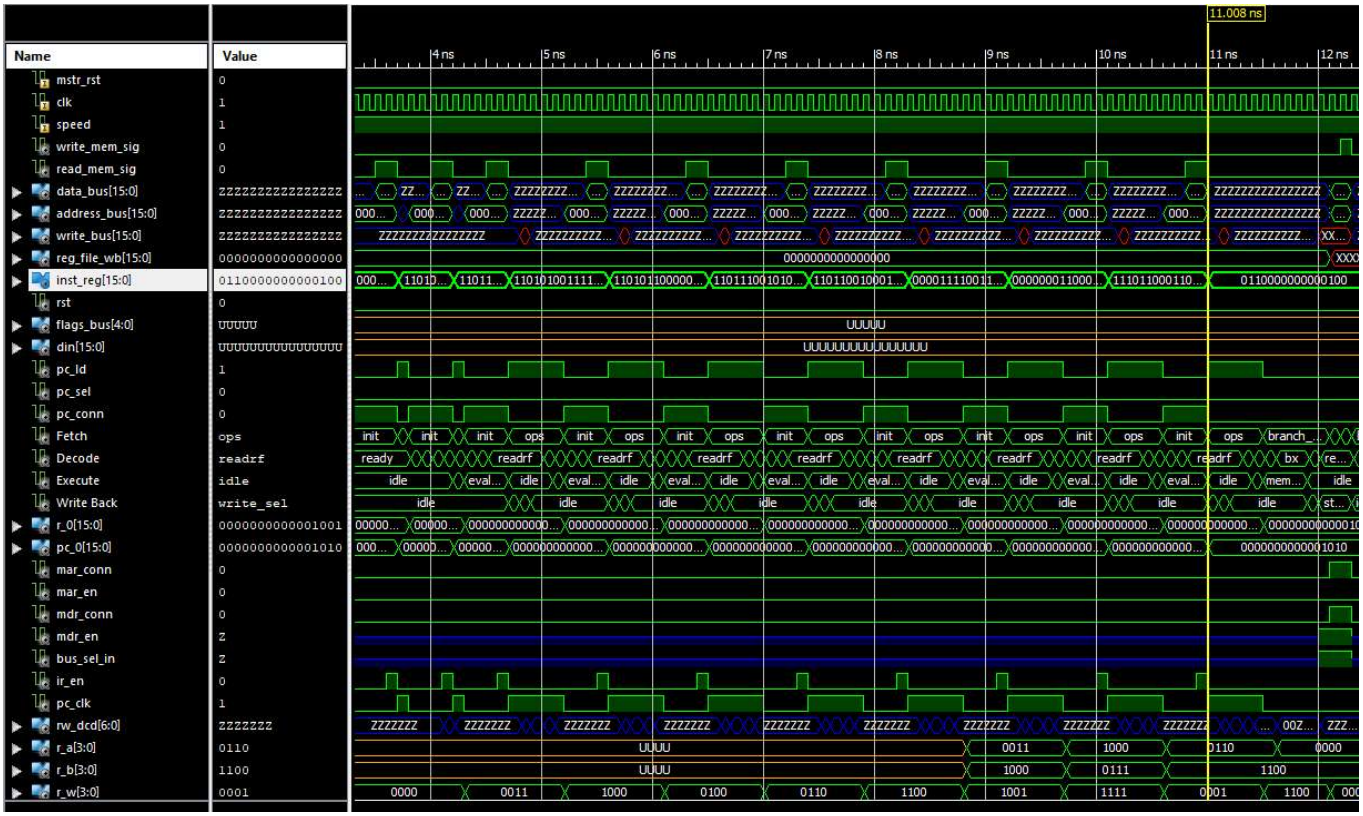


Figure: Slow-Low Power CPU Design + Test Program

The above picture is of the Test Program and it takes about 11 ns or 11,000 ps to finish from start to finish which equated to about 0.5 seconds in real time. This shows that the program runs slower and for lower time using RCA and Booth multiplier compare to the results we saw in Fast-High Power mode

Ayush Patel
Moses Velez
ECE 469 : Lab 8

Appendix:

Sample of Assembly and Machine Program used for Non-Pipeline testing

ARRAY MULTIPLIER in Assembly

```
MULT_F R1 R0 R2 #
MULT_F R2 R0 R3 #
MULT_F R3 R0 R4 #
MULT_F R4 R0 R5 #
MULT_F R5 R0 R6 #
MULT_F R6 R0 R7 #
MULT_F R7 R0 R8 #
MULT_F R8 R0 R9 #
MULT_F R9 R0 R10 #
MULT_F R10 R0 R11 #
HALT #
```

ARRAY MULTIPLIER in Machine

```
0011000100000010
0011001000000011
0011001100000100
0011010000000101
0011010100000110
0011011000000111
0011011100001000
0011100000001001
0011100100001010
0011101000001011
1111000000000000
```

Ayush Patel
Moses Velez
ECE 469 : Lab 8

Matrix Program Used for Analyzing -- Assembly + Machine

MATRIX PROGRAM in Assembly

```
LOAD_I R10   48
ADD_F  R10  R10  R9

LOAD_I R11   0
LOAD_I R12   0

LOAD R13  R10   -3
LOAD R14  R10   -2
LOAD R15  R10   -1

MUL_F  R2  R13  R15      #R2 = n * I    (FAST)
ADD_F R11 R10 R2         #R11 = R10 + n*I       (FAST)

ADD_F R11 R11 R2         #R11 = R11 + n*I       (FAST)
MUL_S  R2  R14  R15      #R2 = m * I    (SLOW)

ADD_S R12 R11 R2         #R12 = R11 + m*I       (SLOW)
ADD_S R12 R12 R2         #R12 = R12 + m*I       (SLOW)

LOAD_I R1  0             # A[i][k]
LOAD_I R2  0             # B[k][j] and can be used for storing A[i][k]*B[k][j]

LOAD_I R3  0             # R3: i = 0,  the 1st  For loop
L1: LOAD_I R4  0         # R4: j = 0,  the 2nd For loop
L2: LOAD_I R5  0         # R5: k = 0, the 3nd For loop

L3: MUL_F  R7  R3  R15   # R7 = i * (row size of A)
    ADD_F  R7  R7  R5    # R7 = R7 + k = i*(row size of A) + k
    ADD_F  R7  R7  R10   # R7 = Address of A[i][k] in the Mem[]
    LOAD   R1   R7  0    # R1 = A[i][k]

    MUL_F  R7  R5  R14   # R7 = k * (row size of B)
    ADD_F  R7  R7  R4    # R7 = k * (row size of B) + j
    ADD_F  R7  R7  R11   # R7 = address of B[k][j] in the Mem[]
    LOAD   R2  R7  0     # R2 = B[k][j]
```

Ayush Patel
Moses Velez
ECE 469 : Lab 8

```
MUL_F  R2  R1 R2          # R2 = A[i][k] * B[k][j]
ADD_F  R6  R6 R2          # Sum = Sum + A[i][k]*B[k][j]

ADD_I  R5  R5  1          # R5 = R5 + 1 (k = k+1)
COMP   R5  R15            # compare R5 and R15  (k and l)
BRANCH_LT   -12           # branch to L3 if R5 < l: offset = 42 - 66 = -24 (byte-addressable)

 # compute C[i][j] address
 MUL_F  R7  R3 R14        # R7 = i * (row size of C)
 ADD_F  R7  R7 R4         # R7 = R7 + j = i*(row size of C) + j
 ADD_F R7  R7  R12        # R7 = Address of C[i][j] in the Mem[]
 STORE  R6  R7  0         # R6-> R7 : C[i][j] = Sum

 ADD_I R4  R4  1          # j = j + 1
 COMP R4 R14              # compare R4, R14 --> j index)
 BRANCH_LT   -21          # branch to L2 if R4 < m: offset = 38 - 80 = -42 (byte-addressable)

 ADD_I R3 R3 1            # i = i + 1
 COMP R3 R13              #compare R3 and R13-->i index)
 BRANCH_LT   -25          # branch to L1 if R3 < n: offset = 36 - 86 = -50 (byte-addressable)

 HALT                     # end program
```

Ayush Patel
Moses Velez
ECE 469 : Lab 8

Machine Code for the Matrix Program:

1101100111111111
0000100110010000
1101101000110000
0000101010101001
1101101100000000
1101110000000000
1100110110101101
1100111010101110
1100111110101111
0011001011011111
0000101110100010
0000101110110010
0100001011101111
0001110010110010
0001110011000010
1101000100000000
1101001000000000
1101001100000000
1101010000000000
1101010100000000
1101011000000000
0011011100111111
0000011101110101
0000011101111010
1100000101110000
0011011101011110
0000011101110100
0000011101111011
1100001001110000
0011001000010010
0000011001100010
0010010101010001
0101010111110000
1000111111110100
0011011100111110
0000011101110100
0000011101111100
1110011001110000
0010010001000001
0101010011100000

Ayush Patel
Moses Velez
ECE 469 : Lab 8

1000111111101011
0010001100110001
0101001111010000
1000111111100111
1111000000000000
0000000000000100
0000000000000100
0000000000000100
0000000000010100
0000000000101000
0000000000111100
0000000001010000
1111111111110110
1111111111110110
0000000000000000
0000000000110111
1111111111101001
1111111110010110
0000000000011010
0000000000000111
1111111110001111
0000000001101101
0000000011100010
0000000000001000
0000000001101010
0000000000110100
0000000000110001
1111111111011101
0000000010001100
1111111111100110
1111111111111101
0000000000000000
0000000000001000
0000000000000101
0000000000011101
0000000000001011
0000000000001101
0000000000110111
1111111111111100
0000000010000110
==========