EEC 180 Lab 6 Report

RISC-V Multicycle Pipelined Processor

Shengmin Liu

Kushagra Tiwari

Section A02

University of California, Davis
Department of Electrical & Computer Engineering

EEC 180 **Winter 2024**

Laboratory Report Cover Sheet

Laboratory Exercise Number	Lab 6 - Final Project
Title of the Laboratory Exercise	Multicycle Pipeline Processor
Date	2024. 3. 13

Names of Team Members (if any)

1. Shengmin Liu
2. Kushagra Tiwari

Preparation (Pre-lab) Verification

TA Signature : [illegible]

Completion Verification

TA Signature: [illegible]

Lab Score:

Laboratory Grading Weightage

Preparation	20%
Design Quality & Correctness	50%
Report	30%

Introduction

In this project, we designed a multicycle pipelined processor based on the RISC-V instruction set (RV32M). It supports the following instructions:

- R-type: ADD, SUB, MUL, AND, OR, XOR
- I-Type: ADDI, SRLI, SLLI
- Memory: SW, LW
- Branch: BNE

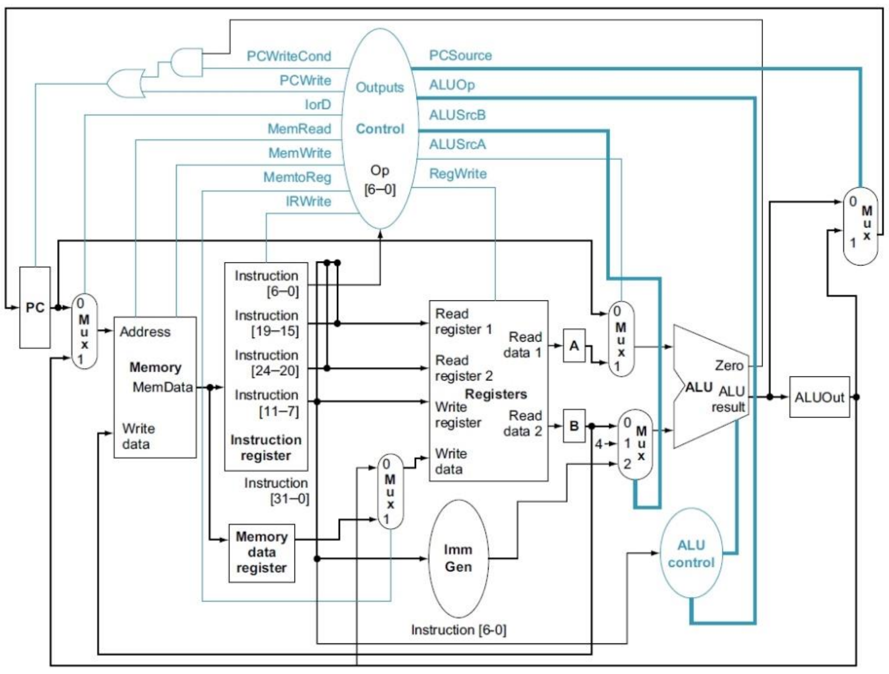
We started by making a multicycle processor that executes 1 instruction at a time through several cycles, then implemented the pipeline function that divides the execution of 1 instruction into 5 stages (fetch, decode, execution, memory access, write back) and executes 5 instructions in five different stage at the same time.

For the pipelined multicycle processors, there will be hazards (Read-After-Write, Load-Use and Branch hazards). They are detected in the instruction fetch stage, where stalls are inserted automatically by the hardware. To test our design, we wrote a simple matrix multiplier algorithm in RISC-V assembly using the Venus RISC-V simulator. We minimized hazards in the software side by rearranging instructions. By importing the machine code into our instruction memory, we were able to verify the functionality of the pipeline. The processor automatically detects the unavoidable hazards and inserts stalls.

Design:

High-level block diagram for multicycle processor:

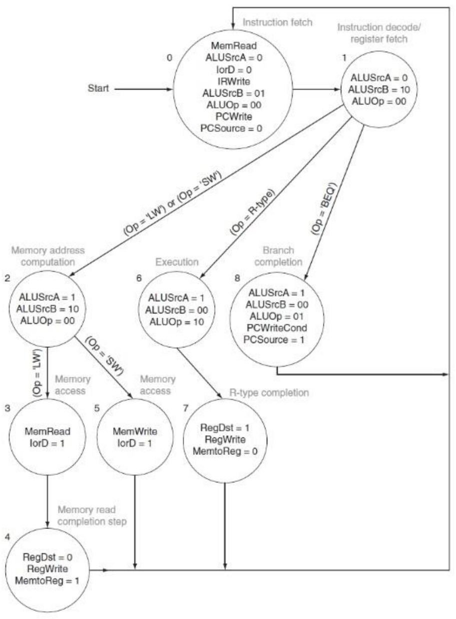
The design is based on the data path in the below figure[1] with some modifications to the PC data path.



[1] Patterson and Hennessy: Organization and Design RISC-V Edition: The Hardware Software Interface. Page 282.e5

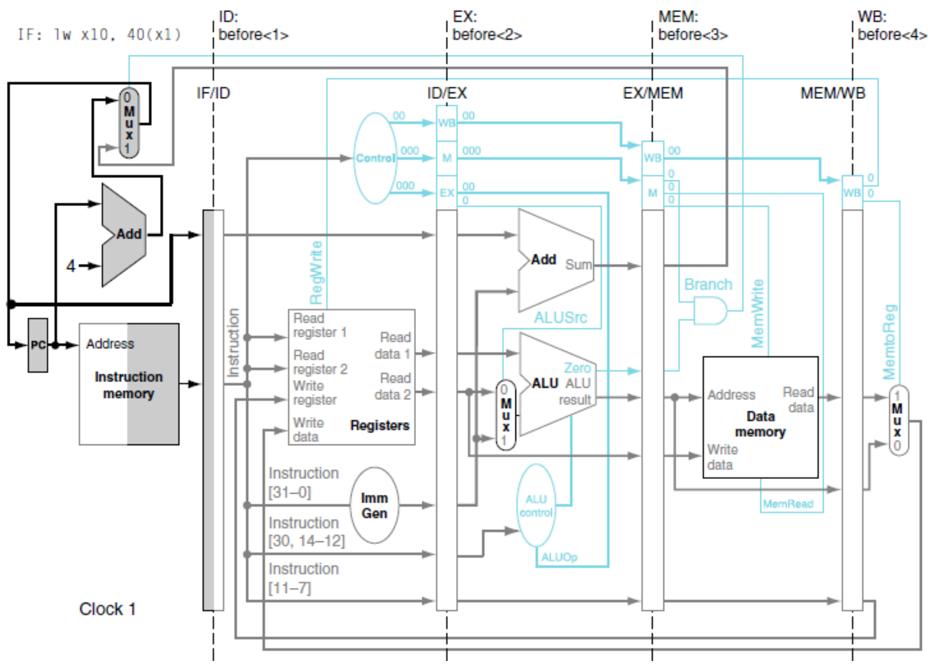
State Diagram (Multicycle non-pipelined processor):

The state diagram is based on the below state diagram[2] with modification to the PC-related signals. Since we are also including I-Type instructions in our processor, we also add another state S10 to the state diagram which can be accessed from state 2 if the opcode is equal to I-Type and this goes to state 7 for write-back the registers.


Instruction fetch
Instruction decode/ register fetch
Start
0
MemRead ALUSrcA = 0 IorD = 0 IRWrite ALUSrcB = 01 ALUOp = 00 PCWrite PCSource = 0
1
ALUSrcA = 0 ALUSrcB = 10 ALUOp = 00
(Op = 'LW') or (Op = 'SW')
(Op = R-type)
(Op = 'BEQ')
Memory address computation
2
ALUSrcA = 1 ALUSrcB = 10 ALUOp = 00
Execution
6
ALUSrcA = 1 ALUSrcB = 00 ALUOp = 10
Branch completion
8
ALUSrcA = 1 ALUSrcB = 00 ALUOp = 01 PCWriteCond PCSource = 1
(Op = 'LW')
(Op = 'SW')
Memory access
Memory access
R-type completion
3
MemRead IorD = 1
5
MemWrite IorD = 1
7
RegDst = 1 RegWrite MemtoReg = 0
Memory read completion step
4
RegDst = 0 RegWrite MemtoReg = 1

High-Level Block Diagram (Pipeline Processor):

The design is based on the data path in the below figure[3] with some modifications to the branch data path. Since the design lacks forwarding and branch prediction, it uses hardware detection of data and branch hazards to insert stalls (NOPs).

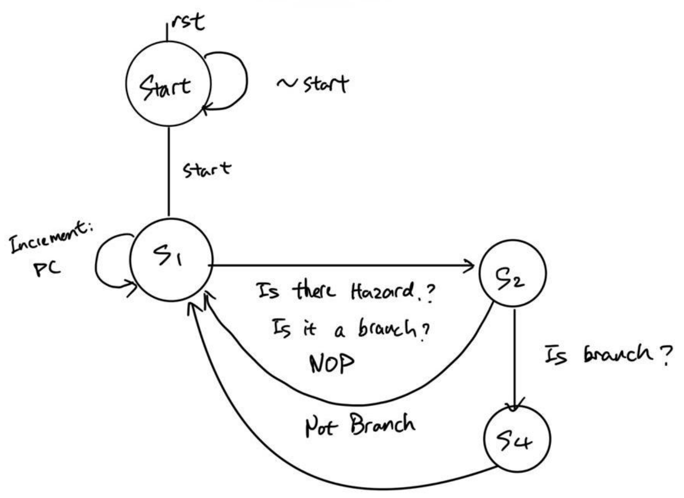


---

[2] Patterson and Hennessy: Organization and Design RISC-V Edition: The Hardware Software Interface. Page 282.e18

[3] Patterson and Hennessy: Organization and Design RISC-V Edition: The Hardware Software Interface. Page 365.e21

State Diagram (Pipeline Processor Hazard Detector):



- State S1 is the hazard or branch detection state. If a hazard is detected, the logic will freeze the program counter (PC) and pass a NOP to the IFID register. If the dependency is two cycles ahead, this NOP resolves the hazard, and the PC can be incremented. If a dependency is one cycle ahead or if there is a branch, the machine goes to state S2.
- State S2 inserts another NOP into the pipeline. This will happen when the dependency is only one cycle ahead in the pipeline. This is also done for branches.
- State S4 inserts a third NOP into the pipeline. It is needed only for branch instructions, since our state machine is in the IF stage, unlike Patterson and Hennessy's ID stage implementation. This results in an additional NOP needed to wait for the ALU to calculate the branch target.

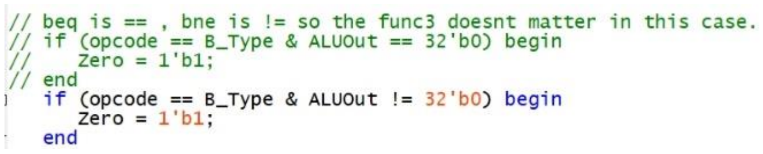
Hardware Description Language (HDL) Modules

Functional Modules:

- Control (pipelined): This module in the instruction decode stage takes the opcode as input and generates the 7 control signals needed for later-stage executions (2 for the execution stage, 3 for the memory access stage, 2 for the write-back stage).

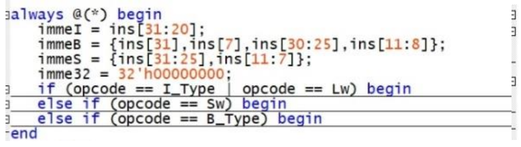
- ALU control (pipelined): This module in the execution stage reads the relevant fields in the instruction and generates the control signal for the ALU. These fields are the opcode, func3, and func7, which decide which arithmetic, shift, or logical operation the ALU needs to perform.

- ALU: This module takes the ALU mode, opcode, and two input words, and outputs the resultant word and the Zero signal. When the Zero signal is 0, it means the flow of the instruction should continue. When the Zero signal is 1, it means the flow of the instruction is changed by the branch instruction. Notice how this implements the BNE instruction control path.

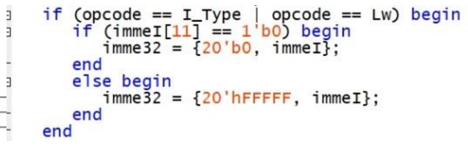


```
// beq is == , bne is != so the func3 doesnt matter in this case.
// if (opcode == B_Type & ALUOut == 32'b0) begin
//     Zero = 1'b1;
// end
   if (opcode == B_Type & ALUOut != 32'b0) begin
      Zero = 1'b1;
   end
```

- ❖ Immediate generator: This module in the instruction decode stage takes the instruction as input and decodes the 32-bit immediate number that may be used in the later stages. This process involves reading the 12-bit number from the appropriate field and sign-extending it. According to the ISA, the immediate field position is different for different types of instructions, we use the if statements (collapsed here) to make sure we read the correct immediate field.



```
always @(*) begin
    immeI = ins[31:20];
    immeB = {ins[31],ins[7],ins[30:25],ins[11:8]};
    immeS = {ins[31:25],ins[11:7]};
    imme32 = 32'h00000000;
    if (opcode == I_Type | opcode == Lw) begin
    else if (opcode == Sw) begin
    else if (opcode == B_Type) begin
end
```



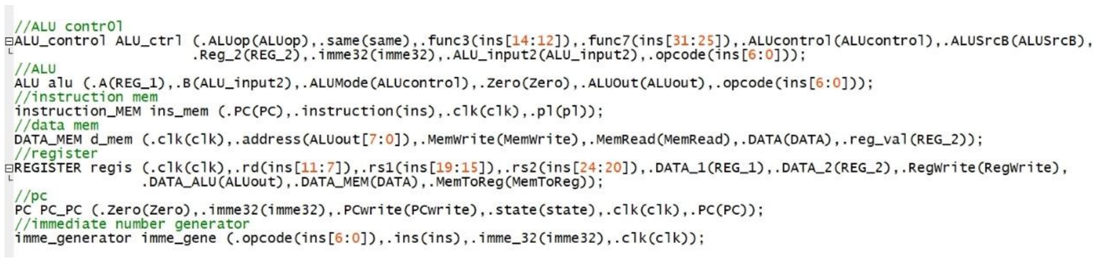
```
if (opcode == I_Type | opcode == Lw) begin
    if (immeI[11] == 1'b0) begin
        imme32 = {20'b0, immeI};
    end
    else begin
        imme32 = {20'hFFFFF, immeI};
    end
end
```

- ❖ Control (multicycle): This module implements the state machine of the multicycle processor. The control module takes the opcode whenever a new instruction is executed and generates the required signals for the execution of the instruction based on the current state in each cycle.

- ❖ ALU control (multicycle): This module reads the relevant fields in the instruction generates the control signal for the ALU and chooses the second input for the ALU. The fields for generating the ALU mode include ALU opcode, func3, and func7. The signals needed for generating second ALU inputs are the ALUSrcB, Register 2, and immediate value.

- ❖ PC (multicycle): This module controls the flow of the PC of the multicycle processor. It reads signals from the relevant modules and increments PC by needed values. The relevant signals are the Zero signals from the ALU module, the immediate value from the immediate generator, and the PCwrite signal from the multicycle control module.

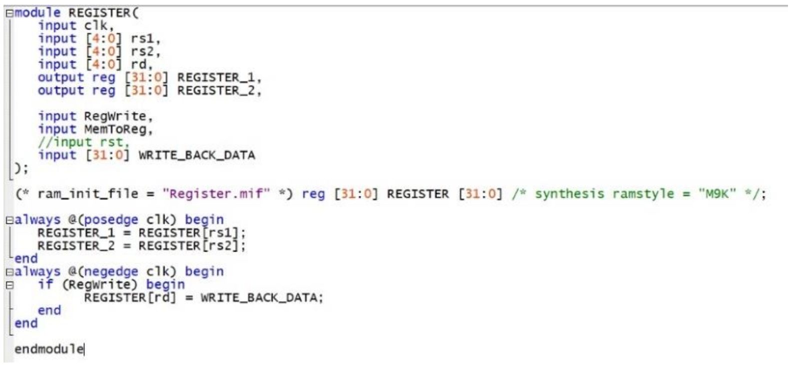
- ❖ Datapath (multicycle): This module implements the data path between different functional modules, memory files, and registers files.



```
//ALU contr0l
ALU_control ALU_ctrl (.ALUop(ALUop),.same(same),.func3(ins[14:12]),.func7(ins[31:25]),.ALUcontrol(ALUcontrol),.ALUSrcB(ALUSrcB),
                      .Reg_2(REG_2),.imme32(imme32),.ALU_input2(ALU_input2),.opcode(ins[6:0]));
//ALU
ALU alu (.A(REG_1),.B(ALU_input2),.ALUMode(ALUcontrol),.Zero(Zero),.ALUOut(ALUout),.opcode(ins[6:0]));
//instruction mem
instruction_MEM ins_mem (.PC(PC),.instruction(ins),.clk(clk),.pl(pl));
//data mem
DATA_MEM d_mem (.clk(clk),.address(ALUout[7:0]),.MemWrite(MemWrite),.MemRead(MemRead),.DATA(DATA),.reg_val(REG_2));
//register
REGISTER regis (.clk(clk),.rd(ins[11:7]),.rs1(ins[19:15]),.rs2(ins[24:20]),.DATA_1(REG_1),.DATA_2(REG_2),.RegWrite(RegWrite),
                .DATA_ALU(ALUout),.DATA_MEM(DATA),.MemToReg(MemToReg));
//pc
PC PC_PC (.Zero(Zero),.imme32(imme32),.PCwrite(PCwrite),.state(state),.clk(clk),.PC(PC));
//immediate number generator
imme_generator imme_gene (.opcode(ins[6:0]),.ins(ins),.imme_32(imme32),.clk(clk));
```

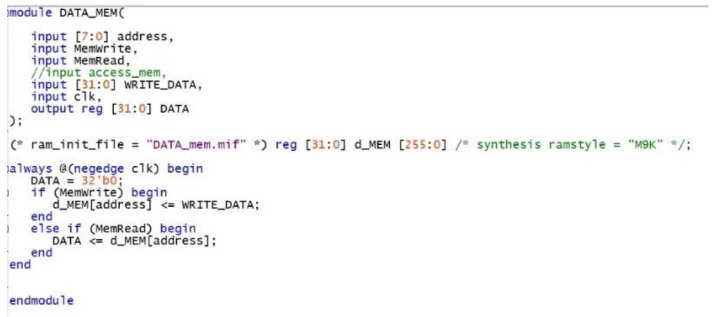
Memory and Register files:

❖ Register file: This is a 32-word by 32-bit memory that contains the registers used for execution. Register write operations happen on the negative edges of the clock cycle, and read operations happen on the positive edges. This prevents the structural hazard caused by reads and writes being scheduled in the same clock cycle due to the pipeline.



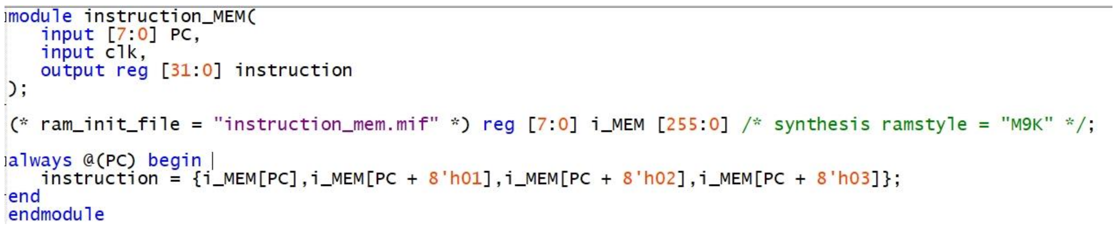
```
module REGISTER(
    input clk,
    input [4:0] rs1,
    input [4:0] rs2,
    input [4:0] rd,
    output reg [31:0] REGISTER_1,
    output reg [31:0] REGISTER_2,

    input RegWrite,
    input MemToReg,
    //input rst,
    input [31:0] WRITE_BACK_DATA
);

(* ram_init_file = "Register.mif" *) reg [31:0] REGISTER [31:0] /* synthesis ramstyle = "M9K" */;

always @(posedge clk) begin
    REGISTER_1 = REGISTER[rs1];
    REGISTER_2 = REGISTER[rs2];
end
always @(negedge clk) begin
    if (RegWrite) begin
            REGISTER[rd] = WRITE_BACK_DATA;
    end
end

endmodule
```

❖ Data Memory: This is a 256-word by 32-bit memory representing the data memory of a processor. On the negative edge of the clock, it updates the memory at the specified address if the MemWrite signal is asserted, or reads the data from some input address for memory if MemRead is asserted.



```
module DATA_MEM(

    input [7:0] address,
    input MemWrite,
    input MemRead,
    //input access_mem,
    input [31:0] WRITE_DATA,
    input clk,
    output reg [31:0] DATA
);

(* ram_init_file = "DATA_mem.mif" *) reg [31:0] d_MEM [255:0] /* synthesis ramstyle = "M9K" */;

always @(negedge clk) begin
    DATA = 32'b0;
    if (MemWrite) begin
        d_MEM[address] <= WRITE_DATA;
    end
    else if (MemRead) begin
        DATA <= d_MEM[address];
    end
end


endmodule
```

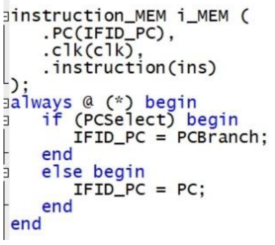
❖ Instruction Memory: This is a 256-word by 8-bit memory representing the instruction memory of a processor, indexed by the program counter. Since in this ISA, instructions are byte-addressable, the word size is 8-bits. The module concatenates the four bytes from the value of PC to PC + 3 to provide a complete 32-bit instruction to the pipeline.



```
module instruction_MEM(
    input [7:0] PC,
    input clk,
    output reg [31:0] instruction
);

(* ram_init_file = "instruction_mem.mif" *) reg [7:0] i_MEM [255:0] /* synthesis ramstyle = "M9K" */;

always @(PC) begin
    instruction = {i_MEM[PC],i_MEM[PC + 8'h01],i_MEM[PC + 8'h02],i_MEM[PC + 8'h03]};
end
endmodule
```

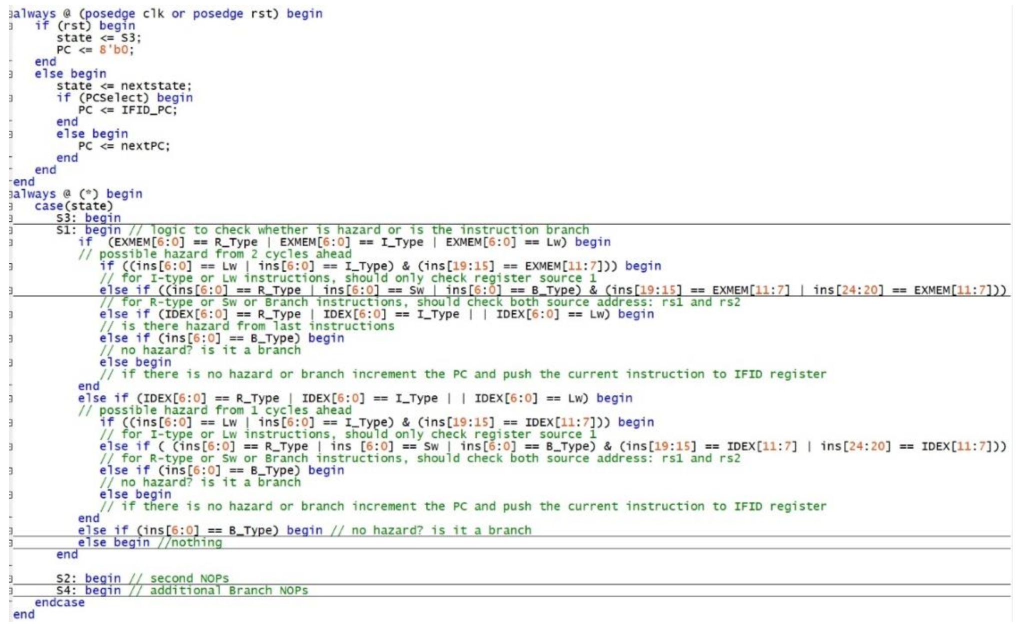
Datapath Modules for pipelined processor:
These are the five-stage modules of the RISC-V instructions.

- Instruction Fetch: This module instantiates the instruction memory to read instructions and uses a mux to decide which instruction to fetch.



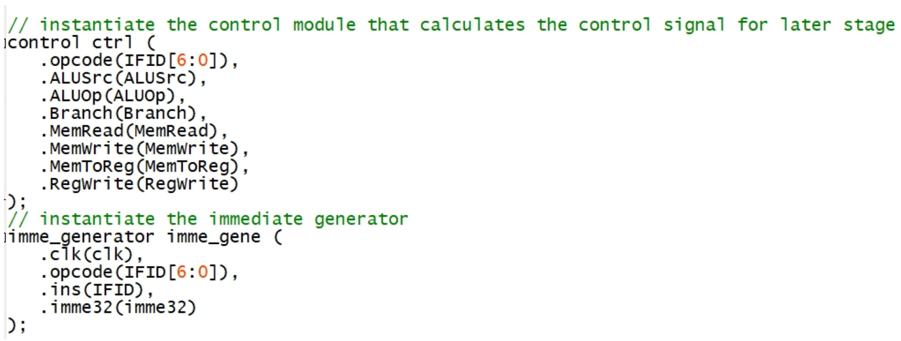
```
instruction_MEM i_MEM (
    .PC(IFID_PC),
    .clk(clk),
    .instruction(ins)
);
always @ (*) begin
    if (PCSelect) begin
        IFID_PC = PCBranch;
    end
    else begin
        IFID_PC = PC;
    end
end
```

The instruction fetch module also has the state machine for the program. In state S1, it will check with the previous 2 instructions to see if there are any data hazards with the current instructions. EXMEM hazards require 1 cycle of NOP, IDEX hazards require 2 cycles of NOPs, and Branch instructions require 3 cycles of NOPs. Based on the test conditions, the program will freeze the PC and put NOPs in the IFID register when there is a hazard or branches. Otherwise, the program will increment the PC by 4 and put the instruction to the IFID register.



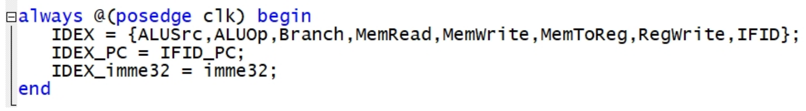
```
always @ (posedge clk or posedge rst) begin
    if (rst) begin
        state <= S3;
        PC <= 8'b0;
    end
    else begin
        state <= nextstate;
        if (PCSelect) begin
            PC <= IFID_PC;
        end
        else begin
            PC <= nextPC;
        end
    end
end
always @ (*) begin
    case(state)
        S3: begin
        S1: begin // logic to check whether is hazard or is the instruction branch
            if  (EXMEM[6:0] == R_Type | EXMEM[6:0] == I_Type | EXMEM[6:0] == Lw) begin
            // possible hazard from 2 cycles ahead
                if ((ins[6:0] == Lw | ins[6:0] == I_Type) & (ins[19:15] == EXMEM[11:7])) begin
                // for I-type or Lw instructions, should only check register source 1
                else if ((ins[6:0] == R_Type | ins[6:0] == Sw | ins[6:0] == B_Type) & (ins[19:15] == EXMEM[11:7] | ins[24:20] == EXMEM[11:7]))
                // for R-type or Sw or Branch instructions, should check both source address: rs1 and rs2
                else if (IDEX[6:0] == R_Type | IDEX[6:0] == I_Type | | IDEX[6:0] == Lw) begin
                // is there hazard from last instructions
                else if (ins[6:0] == B_Type) begin
                // no hazard? is it a branch
                else begin
                // if there is no hazard or branch increment the PC and push the current instruction to IFID register
            end
            else if (IDEX[6:0] == R_Type | IDEX[6:0] == I_Type | | IDEX[6:0] == Lw) begin
            // possible hazard from 1 cycles ahead
                if ((ins[6:0] == Lw | ins[6:0] == I_Type) & (ins[19:15] == IDEX[11:7])) begin
                // for I-type or Lw instructions, should only check register source 1
                else if ( (ins[6:0] == R_Type | ins [6:0] == Sw | ins[6:0] == B_Type) & (ins[19:15] == IDEX[11:7] | ins[24:20] == IDEX[11:7]))
                // for R-type or Sw or Branch instructions, should check both source address: rs1 and rs2
                else if (ins[6:0] == B_Type) begin
                // no hazard? is it a branch
                else begin
                // if there is no hazard or branch increment the PC and push the current instruction to IFID register
            end
            else if (ins[6:0] == B_Type) begin // no hazard? is it a branch
            else begin //nothing
        end

        S2: begin // second NOPs
        S4: begin // additional Branch NOPs
    endcase
end
```

- Instruction Decode:
  The instruction decode module instantiates the control module and imme_generator module.



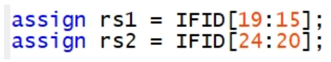
```
// instantiate the control module that calculates the control signal for later stage
control ctrl (
    .opcode(IFID[6:0]),
    .ALUSrc(ALUSrc),
    .ALUOp(ALUOp),
    .Branch(Branch),
    .MemRead(MemRead),
    .MemWrite(MemWrite),
    .MemToReg(MemToReg),
    .RegWrite(RegWrite)
);
// instantiate the immediate generator
imme_generator imme_gene (
    .clk(clk),
    .opcode(IFID[6:0]),
    .ins(IFID),
    .imme32(imme32)
);
```

This module will concatenate the 7 control signals with the instructions in the IFID register to become the IDEX register. It will also pass the IFID_PC and the immediate value to the execution stage as IDEX_PC and IDEX_imme32.



```
always @(posedge clk) begin
    IDEX = {ALUSrc,ALUOp,Branch,MemRead,MemWrite,MemToReg,RegWrite,IFID};
    IDEX_PC = IFID_PC;
    IDEX_imme32 = imme32;
end
```

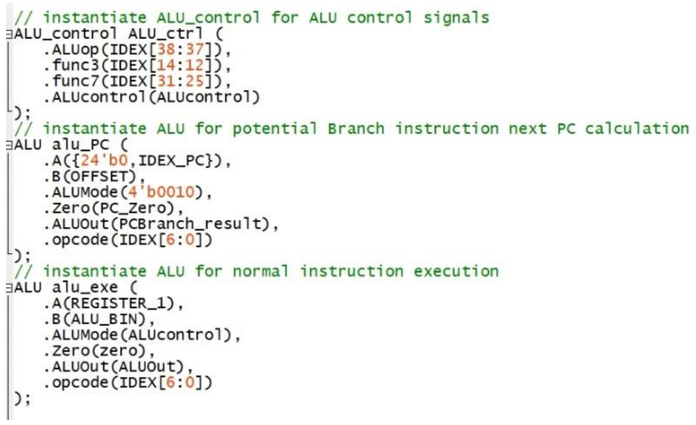
It also assigns values to 2 variables rs1 and rs2 to read data stored in the register file. The two values read from the register files will be stored in REGISTER_1 and REGISTER_2 and passed to the execution module.



```
assign rs1 = IFID[19:15];
assign rs2 = IFID[24:20];
```

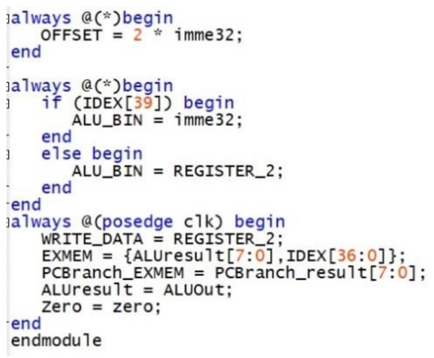
❖ Execution:

The execution module does the calculation for the next PC if there is a branch and executions for the instruction. It instantiates the ALUcontrol module, and two ALU modules (one for PC, one for execution).



```
// instantiate ALU_control for ALU control signals
ALU_control ALU_ctrl (
    .ALUop(IDEX[38:37]),
    .func3(IDEX[14:12]),
    .func7(IDEX[31:25]),
    .ALUcontrol(ALUcontrol)
);
// instantiate ALU for potential Branch instruction next PC calculation
ALU alu_PC (
    .A({24'b0,IDEX_PC}),
    .B(OFFSET),
    .ALUMode(4'b0010),
    .Zero(PC_Zero),
    .ALUOut(PCBranch_result),
    .opcode(IDEX[6:0])
);
// instantiate ALU for normal instruction execution
ALU alu_exe (
    .A(REGISTER_1),
    .B(ALU_BIN),
    .ALUMode(ALUcontrol),
    .Zero(zero),
    .ALUOut(ALUOut),
    .opcode(IDEX[6:0])
);
```

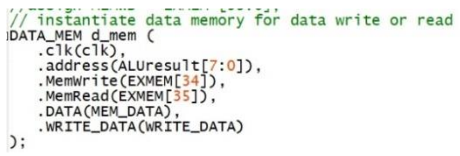
The module will calculate the offset for the branched PC calculation since RV32M uses half words for branches. There is also a MUX to give the correct signal for the execution alu module based on the control signal. It will pass the branched PC, register_2, execution ALU output, and the Zero signal to the data memory stage as the EXMEM registers.



```
always @(*)begin
    OFFSET = 2 * imme32;
end

always @(*)begin
    if (IDEX[39]) begin
        ALU_BIN = imme32;
    end
    else begin
        ALU_BIN = REGISTER_2;
    end
end
always @(posedge clk) begin
    WRITE_DATA = REGISTER_2;
    EXMEM = {ALUresult[7:0],IDEX[36:0]};
    PCBranch_EXMEM = PCBranch_result[7:0];
    ALUresult = ALUOut;
    Zero = zero;
end
endmodule
```

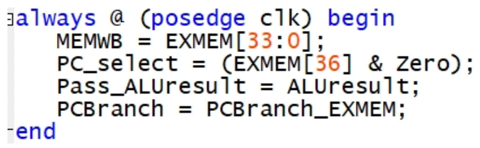
❖ Data Memory:

The data memory module instantiates the data memory file for reading and writing the data.



```
// instantiate data memory for data write or read
DATA_MEM d_mem (
    .clk(clk),
    .address(ALUresult[7:0]),
    .MemWrite(EXMEM[34]),
    .MemRead(EXMEM[35]),
    .DATA(MEM_DATA),
    .WRITE_DATA(WRITE_DATA)
);
```

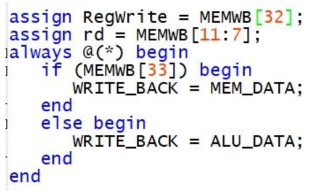
This module passes the MEMWB register and the ALU result from the execution stage to the write-back stage to write the data to the register and the PC-select signal and Branch result to the instruction fetch stage to decide the next PC.



```
always @ (posedge clk) begin
    MEMWB = EXMEM[33:0];
    PC_select = (EXMEM[36] & Zero);
    Pass_ALUresult = ALUresult;
    PCBranch = PCBranch_EXMEM;
end
```

- Write Back:

  This module sends the RegWrite signal, the write address, and the data to be written to the register file.



```
assign RegWrite = MEMWB[32];
assign rd = MEMWB[11:7];
always @(*) begin
    if (MEMWB[33]) begin
        WRITE_BACK = MEM_DATA;
    end
    else begin
        WRITE_BACK = ALU_DATA;
    end
end
```

Top Level for multicycle processor:

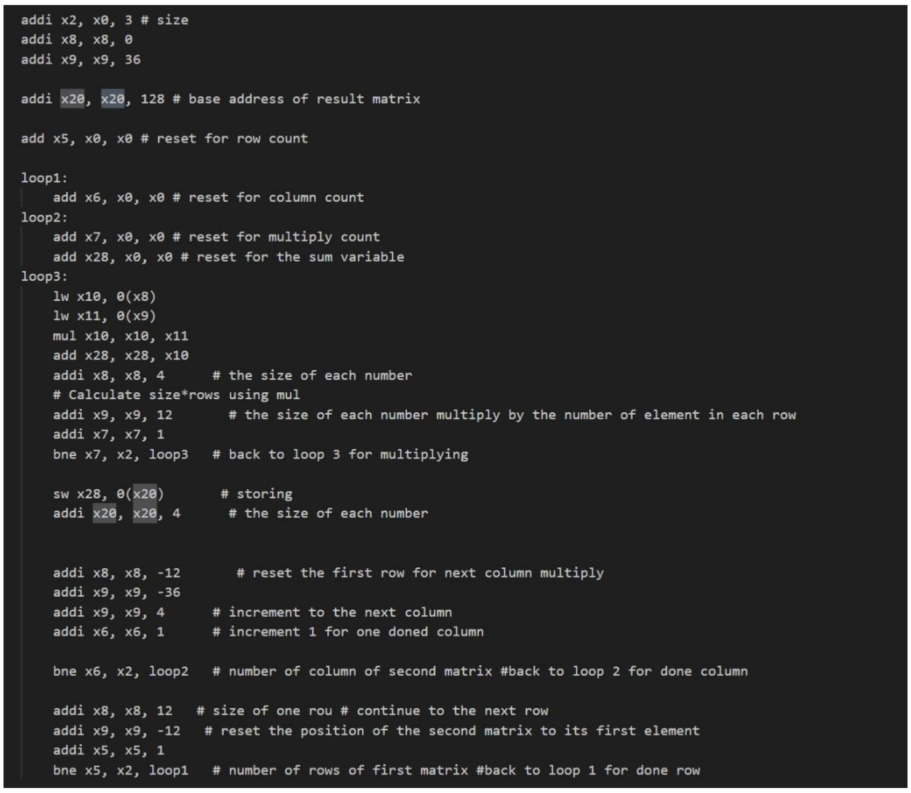
The top-level design instantiates the Control module and Datapath module.

Top Level for pipelined multicycle processor:

The top-level design instantiates the five pipelined data path modules and the register file modules. It takes CLK, RST, and START signals as inputs.

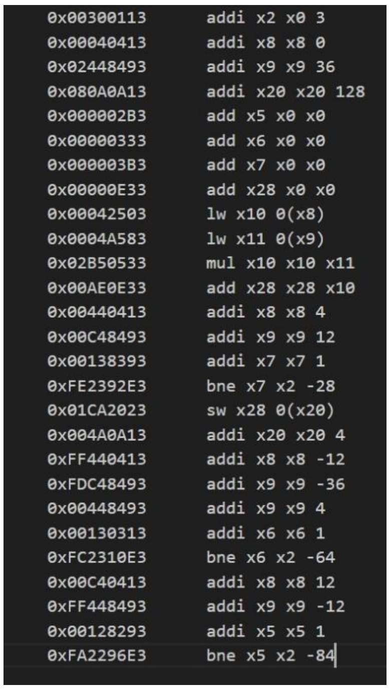
Testbench:

We tested the two processors with the following matrix multiplication algorithm. It assumes that the two matrix operands are stored in memory word addresses 0x00-0x08 and 0x9-0x11. The result is stored in memory addresses 0x20-0x28.



```
addi x2, x0, 3 # size
addi x8, x8, 0
addi x9, x9, 36

addi x20, x20, 128 # base address of result matrix

add x5, x0, x0 # reset for row count

loop1:
    add x6, x0, x0 # reset for column count
loop2:
    add x7, x0, x0 # reset for multiply count
    add x28, x0, x0 # reset for the sum variable
loop3:
    lw x10, 0(x8)
    lw x11, 0(x9)
    mul x10, x10, x11
    add x28, x28, x10
    addi x8, x8, 4      # the size of each number
    # Calculate size*rows using mul
    addi x9, x9, 12       # the size of each number multiply by the number of element in each row
    addi x7, x7, 1
    bne x7, x2, loop3   # back to loop 3 for multiplying

    sw x28, 0(x20)       # storing
    addi x20, x20, 4      # the size of each number


    addi x8, x8, -12       # reset the first row for next column multiply
    addi x9, x9, -36
    addi x9, x9, 4      # increment to the next column
    addi x6, x6, 1      # increment 1 for one doned column

    bne x6, x2, loop2   # number of column of second matrix #back to loop 2 for done column

    addi x8, x8, 12   # size of one rou # continue to the next row
    addi x9, x9, -12   # reset the position of the second matrix to its first element
    addi x5, x5, 1
    bne x5, x2, loop1   # number of rows of first matrix #back to loop 1 for done row
```

To run these instructions in Verilog, we first transform these instructions into RV32M machine code using the "View Assembly" feature in Venus. For the non-pipelined processor, we manually add NOPS in the software. A NOP in our design is just 0x00000000.



```
0x00300113    addi x2 x0 3
0x00040413    addi x8 x8 0
0x02448493    addi x9 x9 36
0x080A0A13    addi x20 x20 128
0x000002B3    add x5 x0 x0
0x00000333    add x6 x0 x0
0x000003B3    add x7 x0 x0
0x00000E33    add x28 x0 x0
0x00042503    lw x10 0(x8)
0x0004A583    lw x11 0(x9)
0x02B50533    mul x10 x10 x11
0x00AE0E33    add x28 x28 x10
0x00440413    addi x8 x8 4
0x00C48493    addi x9 x9 12
0x00138393    addi x7 x7 1
0xFE2392E3    bne x7 x2 -28
0x01CA2023    sw x28 0(x20)
0x004A0A13    addi x20 x20 4
0xFF440413    addi x8 x8 -12
0xFDC48493    addi x9 x9 -36
0x00448493    addi x9 x9 4
0x00130313    addi x6 x6 1
0xFC2310E3    bne x6 x2 -64
0x00C40413    addi x8 x8 12
0xFF448493    addi x9 x9 -12
0x00128293    addi x5 x5 1
0xFA2296E3    bne x5 x2 -84
```

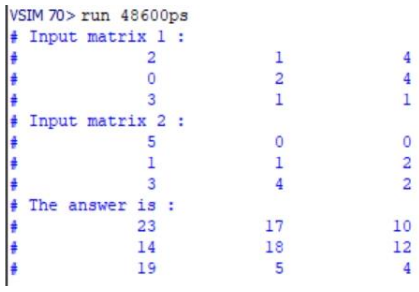
Secondly, we store and initialize all these instructions with the following format following RISC-V standards in the testbench. Unfortunately, for the purposes of this testbench, we were unable to get Modelsim to read the MIF file format, so we had to manually add instructions.

```
//0x00300113      addi x2 x0 3
   UUT.dp.ins_mem.i_MEM[0] = 8'h00;
   UUT.dp.ins_mem.i_MEM[1] = 8'h30;
   UUT.dp.ins_mem.i_MEM[2] = 8'h01;
   UUT.dp.ins_mem.i_MEM[3] = 8'h13;
```

The matrix operands are stored in data memory as shown:

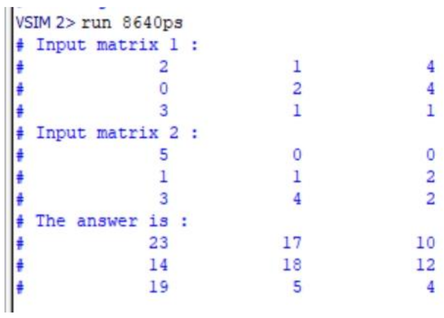
```
    // matrix 1
UUT.dp.d_mem.d_MEM[0]  = 32'h00000002;
UUT.dp.d_mem.d_MEM[4]  = 32'h00000001;
UUT.dp.d_mem.d_MEM[8]  = 32'h00000004;

UUT.dp.d_mem.d_MEM[12]  = 32'h00000000;
UUT.dp.d_mem.d_MEM[16]  = 32'h00000002;
UUT.dp.d_mem.d_MEM[20]  = 32'h00000004;

UUT.dp.d_mem.d_MEM[24]  = 32'h00000003;
UUT.dp.d_mem.d_MEM[28]  = 32'h00000001;
UUT.dp.d_mem.d_MEM[32]  = 32'h00000001;

// matrix 2
UUT.dp.d_mem.d_MEM[36]  = 32'h00000005;
UUT.dp.d_mem.d_MEM[40]  = 32'h00000000;
UUT.dp.d_mem.d_MEM[44]  = 32'h00000000;

UUT.dp.d_mem.d_MEM[48]  = 32'h00000001;
UUT.dp.d_mem.d_MEM[52]  = 32'h00000001;
UUT.dp.d_mem.d_MEM[56]  = 32'h00000002;

UUT.dp.d_mem.d_MEM[60]  = 32'h00000003;
UUT.dp.d_mem.d_MEM[64]  = 32'h00000004;
UUT.dp.d_mem.d_MEM[68]  = 32'h00000002;
```

Testbench Output for Multicycle Processor:



```
VSIM 70> run 48600ps
# Input matrix 1 :
#            2           1           4
#            0           2           4
#            3           1           1
# Input matrix 2 :
#            5           0           0
#            1           1           2
#            3           4           2
# The answer is :
#           23          17          10
#           14          18          12
#           19           5           4
```

Testbench Output for Pipeline Processor :



```
VSIM 2> run 8640ps
# Input matrix 1 :
#            2           1           4
#            0           2           4
#            3           1           1
# Input matrix 2 :
#            5           0           0
#            1           1           2
#            3           4           2
# The answer is :
#           23          17          10
#           14          18          12
#           19           5           4
```

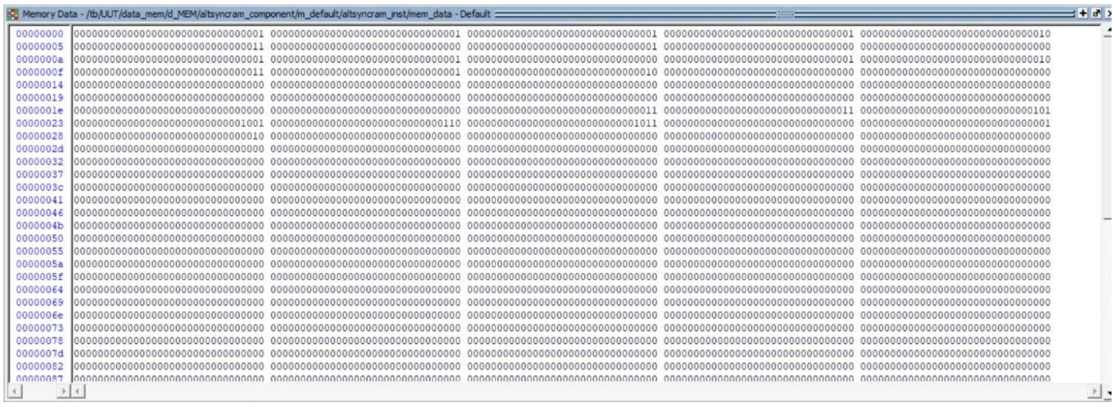
After testing many cases, we verified that both processors are fully functional.
Based on our trials, for a 3 by 3 matrix multiplication, the multicycle processor takes 2430 clock cycles to finish the calculation. The pipelined processor takes only 432 clock cycles to finish the same task, which is a huge speedup. Overall, the pipelined processor is 5.625 times faster than the multicycle processor. The clock period for both processors were kept the same, but if synthesized, we would be able to make the pipelined processor run at an even faster clock speed. Even conservative estimates of the total speedup would be at least 20x.

Additionally, we tested the non-optimized machine code against the code that we optimized for minimal hazards. Without hazard minimization, it took 650 clock cycles. Organizing the code, it took 432 clock cycles with a 33.5 percent increase.

Hardware Tests:

Unfortunately, we were struggling to synthesize our design, as Quartus would optimize our memory files by deleting them. The solution to this problem was to reimplement our memory with Quartus IPs and allow the In-System Memory Content Editor to access and edit the memories. This indicates to Quartus that the memory is mutable by the user, and so it doesn’t optimize it away. When we synthesized the design, only some multipliers got optimized away. This was expected as we were doing a 32x32 multiplication but keeping only the least significant 32 bits.

Before our hardware tests, we needed to re-testbench our design to ensure that the Quartus IPs were functioning identically to the register-based implementation. We used Quartus’ integration with Modelsim, which also allowed the program to initialize the memory for us, saving us the time of manually entering data. The IP-based memory had some mandatory additional latency, so we had to tweak our design to account for it. Eventually, our design passed the testbench. Below is a screenshot of our data memory module. Addresses 0x00-0x08 and 0x09-0x11 are the two matrix operands stored in row-major format. Addresses 0x20-0x28 store the outputs.

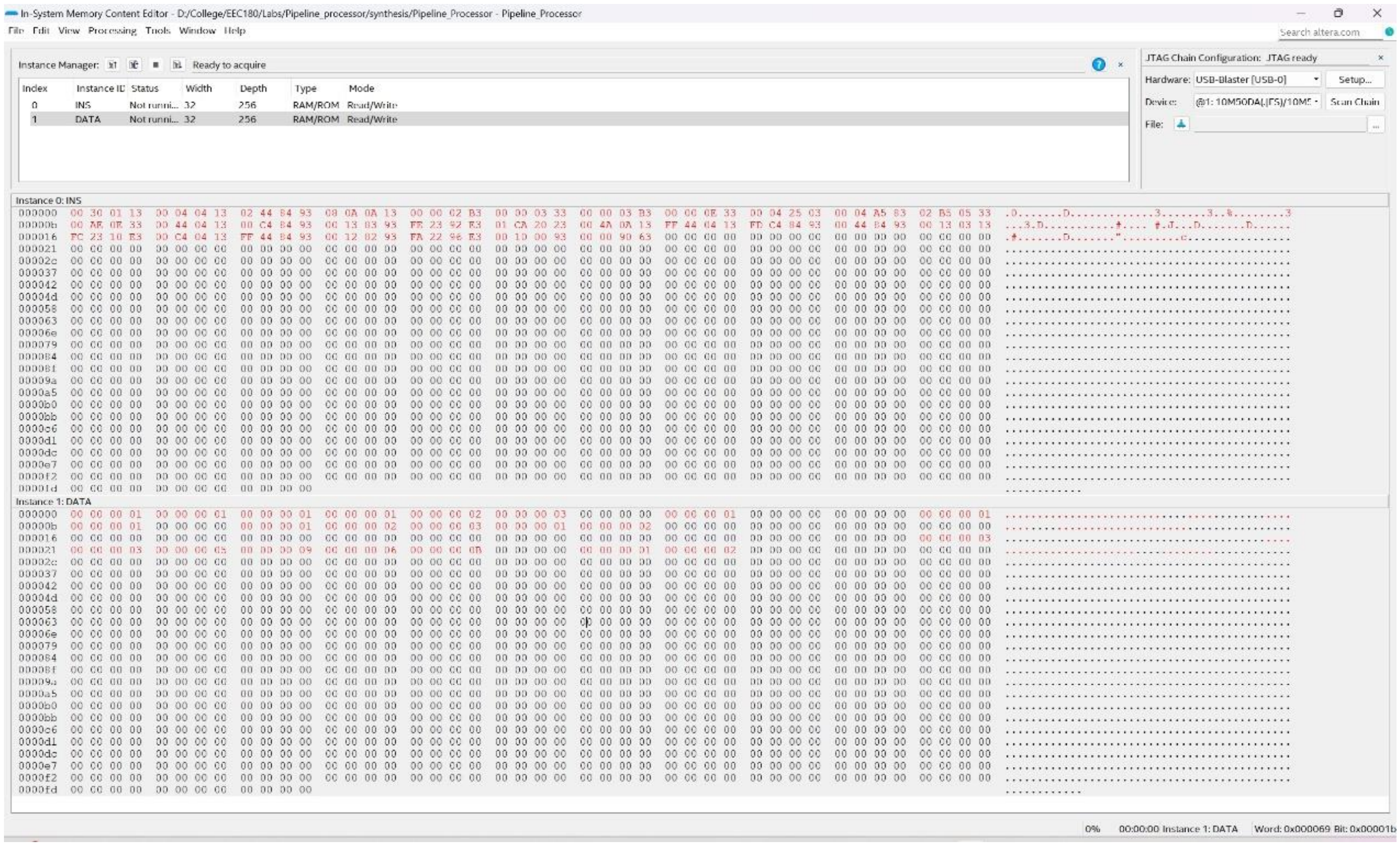


```
Memory Data - /tb/UUT/data_mem/d_MEM/altsyncram_component/m_default/altsyncram_inst/mem_data - Default
00000000 00000000000000000000000000000001 00000000000000000000000000000001 00000000000000000000000000000001 00000000000000000000000000000001 00000000000000000000000000000010
00000005 00000000000000000000000000000011 00000000000000000000000000000000 00000000000000000000000000000001 00000000000000000000000000000000 00000000000000000000000000000000
0000000a 00000000000000000000000000000001 00000000000000000000000000000001 00000000000000000000000000000000 00000000000000000000000000000001 00000000000000000000000000000010
0000000f 00000000000000000000000000000011 00000000000000000000000000000001 00000000000000000000000000000010 00000000000000000000000000000000 00000000000000000000000000000000
00000014 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000
00000019 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000
0000001e 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000011 00000000000000000000000000000011 00000000000000000000000000000101
00000023 00000000000000000000000000001001 00000000000000000000000000000110 00000000000000000000000000001011 00000000000000000000000000000000 00000000000000000000000000000001
00000028 00000000000000000000000000000010 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000
0000002d 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000
00000032 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000
00000037 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000
0000003c 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000
00000041 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000
00000046 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000
0000004b 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000
00000050 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000
00000055 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000
0000005a 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000
0000005f 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000
00000064 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000
00000069 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000
0000006e 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000
00000073 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000
00000078 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000
0000007d 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000
00000082 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000
00000087 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000
```

Below are the matrices we used for this test, and the resultant matrix:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 3 \\ 0 & 1 & 0 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 1 \\ 0 & 1 & 2 \\ 3 & 1 & 2 \end{bmatrix} = \begin{bmatrix} 3 & 3 & 5 \\ 9 & 6 & 11 \\ 0 & 1 & 2 \end{bmatrix}$$

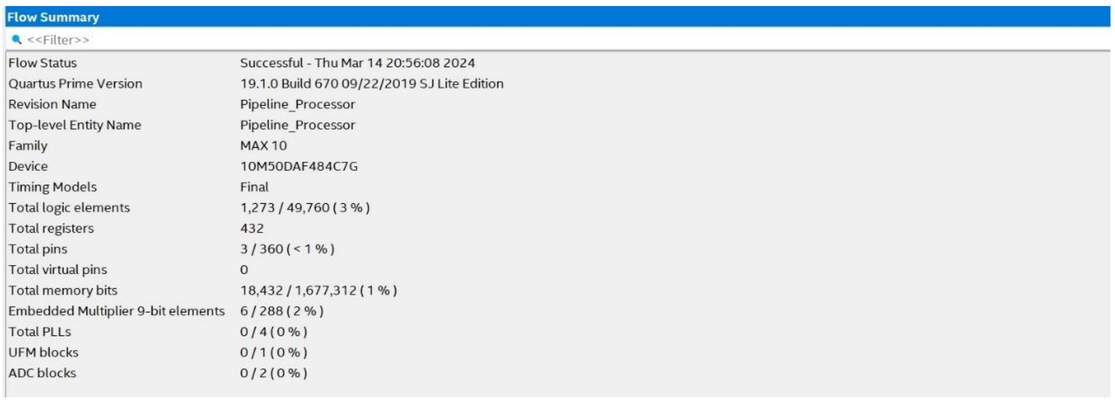
After running the testbench, we programmed the device onto the DE-10 lite board and launched In-System Memory Content Editor. Below is a screenshot. The top block is the instruction memory, which was pre-initialized with the algorithm. The bottom block is the data memory, which was pre-initialized with the matrix operands in addresses 0x00-0x11. After giving the start signal to the processor using a switch, we saw the addresses 0x20 to 0x28 update with the resultant matrix.



```
In-System Memory Content Editor - D:/College/EEC180/Labs/Pipeline_processor/synthesis/Pipeline_Processor - Pipeline_Processor

Instance Manager:   Ready to acquire
Index  Instance ID  Status        Width  Depth  Type     Mode
0      INS          Not runni...  32     256    RAM/ROM  Read/Write
1      DATA         Not runni...  32     256    RAM/ROM  Read/Write

JTAG Chain Configuration: JTAG ready
Hardware: USB-Blaster [USB-0]
Device: @1: 10M50DA(.|ES)/10M5
File:

Instance 0: INS
000000  00 30 01 13  00 04 04 13  02 44 84 93  08 0A 0A 13  00 00 02 B3  00 00 03 33  00 00 03 B3  00 00 0E 33  00 04 25 03  00 04 A5 83  02 B5 05 33  .0.......D.............3.......3..%........3
00000b  00 AE 0E 33  00 44 04 13  00 C4 84 93  00 13 03 93  FE 23 92 E3  01 CA 20 23  00 4A 0A 13  FF 44 04 13  FD C4 84 93  00 44 84 93  00 13 03 13  ...3.D...........#.... #.J...D.......D......
000016  FC 23 10 E3  00 C4 04 13  FF 44 84 93  00 12 82 93  FA 22 96 E3  00 10 00 93  00 00 90 63  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  .#.......D.......".........c................
000021  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ............................................
00002c  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ............................................
000037  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ............................................
000042  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ............................................
00004d  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ............................................
000058  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ............................................
000063  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ............................................
00006e  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ............................................
000079  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ............................................
000084  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ............................................
00008f  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ............................................
00009a  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ............................................
0000a5  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ............................................
0000b0  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ............................................
0000bb  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ............................................
0000c6  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ............................................
0000d1  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ............................................
0000dc  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ............................................
0000e7  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ............................................
0000f2  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ............................................
0000fd  00 00 00 00  00 00 00 00  00 00 00 00                                                                                                            ............
Instance 1: DATA
000000  00 00 00 01  00 00 00 01  00 00 00 01  00 00 00 01  00 00 00 02  00 00 00 03  00 00 00 00  00 00 00 01  00 00 00 00  00 00 00 00  00 00 00 01  ............................................
00000b  00 00 00 01  00 00 00 00  00 00 00 01  00 00 00 02  00 00 00 03  00 00 00 01  00 00 00 02  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ............................................
000016  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 03  ............................................
000021  00 00 00 03  00 00 00 05  00 00 00 09  00 00 00 06  00 00 00 0B  00 00 00 00  00 00 00 01  00 00 00 02  00 00 00 02  00 00 00 00  00 00 00 00  ............................................
00002c  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ............................................
000037  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ............................................
000042  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ............................................
00004d  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ............................................
000058  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ............................................
000063  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ............................................
00006e  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ............................................
000079  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ............................................
000084  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ............................................
00008f  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ............................................
00009a  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ............................................
0000a5  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ............................................
0000b0  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ............................................
0000bb  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ............................................
0000c6  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ............................................
0000d1  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ............................................
0000dc  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ............................................
0000e7  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ............................................
0000f2  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ............................................
0000fd  00 00 00 00  00 00 00 00  00 00 00 00                                                                                                            ............

0%   00:00:00 Instance 1: DATA   Word: 0x000069 Bit: 0x00001b
```

Resource Utilization:

The following is the resource used for the pipelined processor. It uses 1273 logic elements and 432 registers.



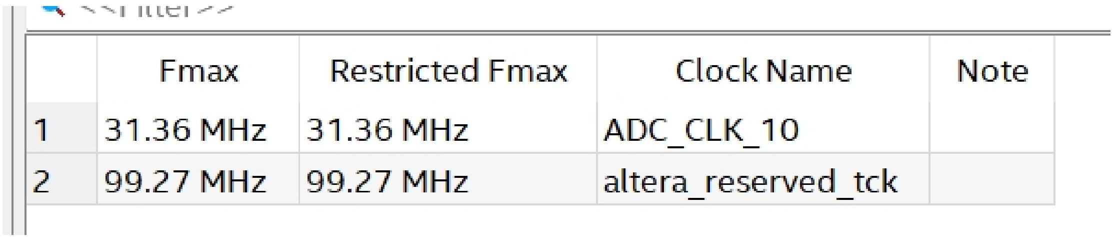
Flow Summary

<<Filter>>

Flow Status	Successful - Thu Mar 14 20:56:08 2024
Quartus Prime Version	19.1.0 Build 670 09/22/2019 SJ Lite Edition
Revision Name	Pipeline_Processor
Top-level Entity Name	Pipeline_Processor
Family	MAX 10
Device	10M50DAF484C7G
Timing Models	Final
Total logic elements	1,273 / 49,760 (3 %)
Total registers	432
Total pins	3 / 360 (< 1 %)
Total virtual pins	0
Total memory bits	18,432 / 1,677,312 (1 %)
Embedded Multiplier 9-bit elements	6 / 288 (2 %)
Total PLLs	0 / 4 (0 %)
UFM blocks	0 / 1 (0 %)
ADC blocks	0 / 2 (0 %)

Timing Analysis:

We used ADC_CLK_10 for our design, as the 50MHz clock was too fast for the setup and hold time requirements of our RTL critical path. Our maximum frequency for the processor was reported as 31.36 MHz. Given more time, we would have liked to use the on-device PLLs to test this maximum frequency.



	Fmax	Restricted Fmax	Clock Name	Note
1	31.36 MHz	31.36 MHz	ADC_CLK_10	
2	99.27 MHz	99.27 MHz	altera_reserved_tck	

Acknowledgements:

We would like to acknowledge Patterson and Henessy for the high-level architecture of the two processors. We would also like to thank Tyler Sheaves, our teaching assistant, for helping us set up the Quartus memory IPs to allow our design to be synthesized.