University of Victoria

16 Bit CPU Project

ECE 449 – Computer Architecture

Brett Dionello	V01026046
Ayman Al Rubaey	V009800780

Sunday, April 6, 2025

Abstract

This report presents the design and implementation of a 16-bit RISC-like CPU. The project involves developing a pipelined processor with separate stages for instruction fetch, decode, execute, memory access, and write-back. The implemented CPU supports various instruction formats, including arithmetic, logical, memory, and branch operations. The report outlines the architecture, data hazard handling, control logic, and synthesis results. Simulation and synthesis results are provided to validate functionality and performance.

Table of Contents

Introduction

Design Specifications

The objective of this project is to design and implement a 16-bit RISC CPU with a five-stage pipeline. The CPU supports arithmetic, logical, load/store, and branch instructions. The design includes hazard handling mechanisms and an efficient control unit to ensure correct execution.

Key Ideas

- Pipelining: The CPU follows a five-stage pipeline (Fetch, Decode, Execute, Memory, Write-back).
- Instruction Set: Supports essential arithmetic, branch, and memory operations.
- Hazard Handling: Implements stalling techniques
- VHDL Implementation: Synthesizable design verified through simulation.
- Hardware realization: Run the VHDL design on the basys3 development board
    - Bootloader
    - Debug console

Instruction set architecture (ISA)

It is required that the CPU implements the ISA in Appendix C

Objective

This project aims to design, implement, and test a 16-bit RISC CPU with pipelined execution. The CPU must support a defined instruction set while handling control and data hazards efficiently.

Body – Design

The simplified CPU block Diagram can be seen in Figure 1: Full system diagram (A larger image is available in appendix B). The main components of the system are visible, except for the following omissions: The hazard unit and some additional signal multiplexing which have been omitted for clarity, the memory management unit is not shown and is depicted as two separate memory components, instruction and data memory, the overflow flags which were added later, and all signals required for the debug console. The CPU design was heavily influenced by the reference text “Computer Organization & Design” by J. L. Hennessy and D. A. Patterson [1], which provided a detailed design of a MIPS CPU.

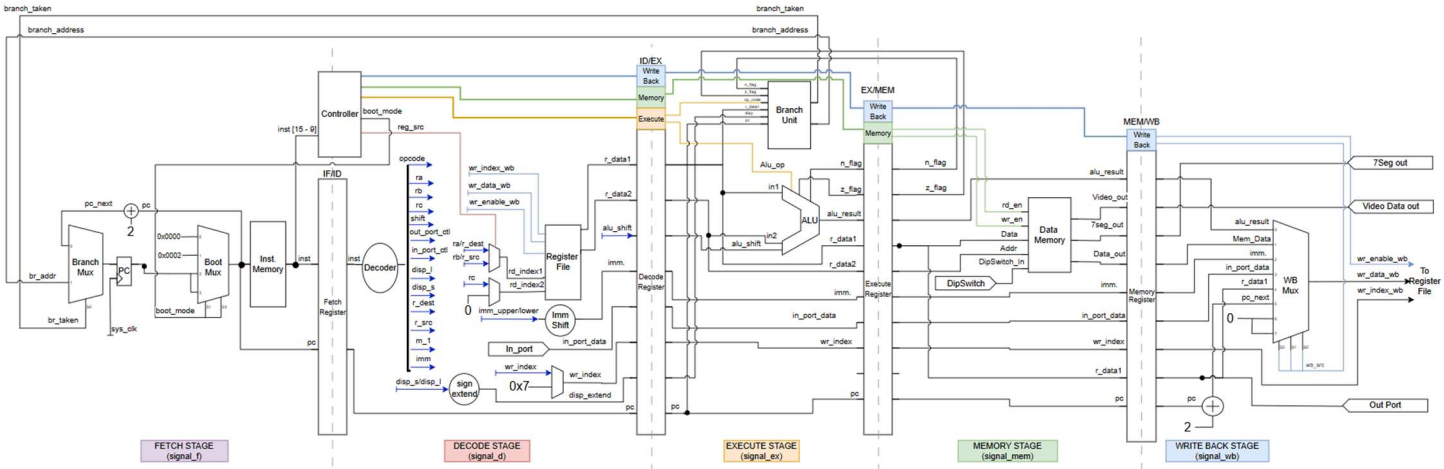


Figure 1: Full system diagram

Fetch stage

The fetch stage can be seen in Figure 2: Fetch stage diagram and is the first stage of the pipeline, it includes signals from the execute stage branch unit. Note: the controller uses the instruction opcode from the fetch stage to maintain pipeline synchronization since the controller is a clocked unit and adds 1 clock cycle delay, effectively acting as a state register.

Components:

- Program counter
- Instruction memory – Part of the memory management unit
- Multiplexors
    - Branch mux – Responsible for switching between the branch address and PC next
    - Boot mux - Responsible for setting the initial PC value to one of the bootloader modes
        - 0x0000 for Execute mode
        - 0x0002 for Load mode

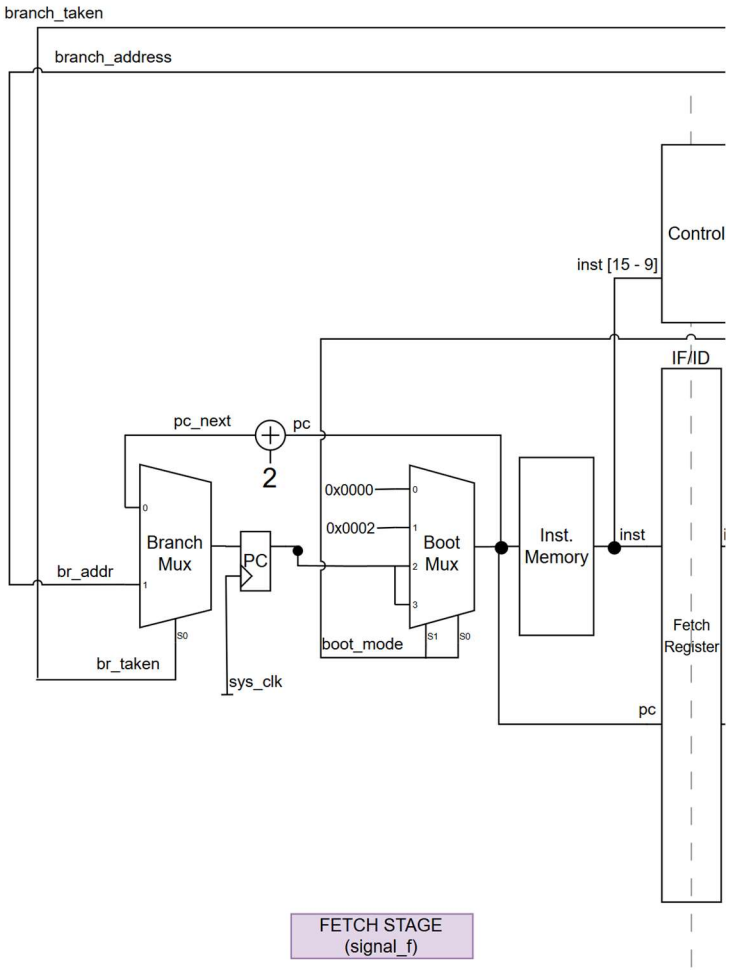


Figure 2: Fetch stage diagram

Decode stage

The decode stage is the second stage of the pipeline and can be seen in Figure 3: Decode stage diagram. Note that the hazard unit and many data signal multiplexors are not shown in the figure.

Components:

- Controller
  - Uses the opcode to update control state
  - Asserts/De-asserts all control signals for entire CPU
  - Handles reset states
    - Reset-Execute
    - Reset-Load
- Decoder
  - Breaks the instruction into its data components
  - Implements the ISA specification
- Register file
  - Working registers 0-7 for CPU instruction operations
  - Asynchronous read operations

        - Read signals from decode stage
    - Falling edge write operations
        - Write signals from writeback stage
- Sign extender
    - Sign-extend
    - 2's complement
    - Shift left by 1
- Multiplexors – Various multiplexors exist in the decode stage to select data signals from the decoder to the correct destinations
    - Write Index MUX – selects between register 7, r_dest (load and mov) and ra based on opcode
    - Read index 1 – selects between r_src, (load and mov), r_dest (store), rb, ra and register 7 based on opcode
    - Read index 2 – selects between r_src (store), rc and 0 based on opcode
- Immediate shift – Shifts the 8 bit immediate value according to the opcode and outputs a 16 bit result to send down the pipeline for writeback.
- In_port – Takes the value from the external in_port constraint and passes it down the pipeline for writeback.

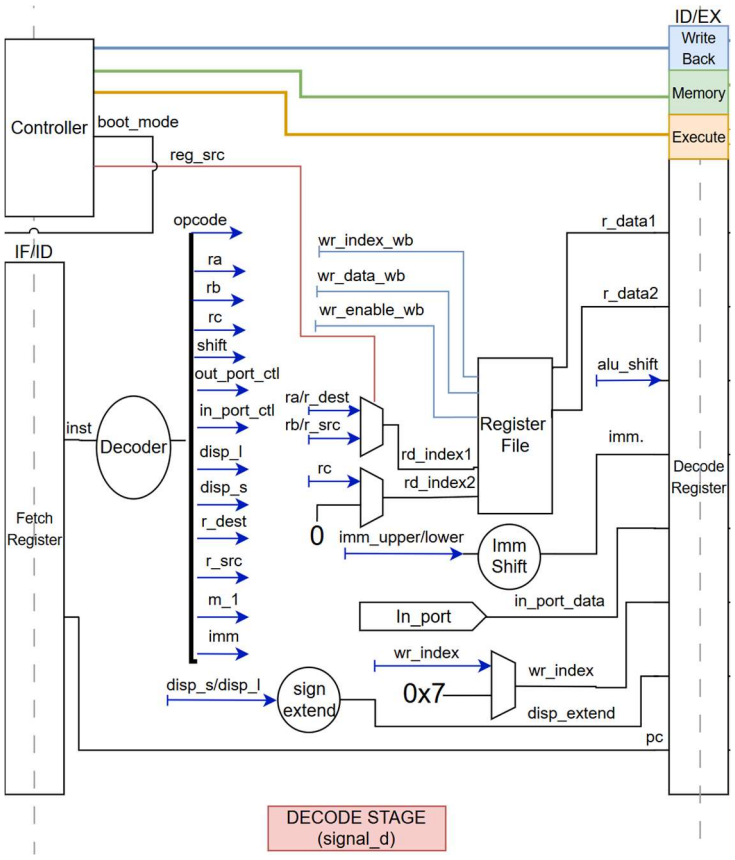


Figure 3: Decode stage diagram

Hazard Detection

The hazard detection unit is implemented in the decode stage and handles branch conditions and data hazards.

Data Hazards

Stalling

All data hazards are handled with stalling, the hazard condition is detected by comparing signals in the execute stage with those in the decode stage. The first check is to see if the instruction in the execute stage has asserted a control signal for memory read, or register write operations. A memory read indicates the possibility of a “read after load” hazard, and a register write operation indicates the possibility of a “read after write” or a “write after write” hazard. If one of these control signals is asserted, the source registers of the currently decoding instruction are compared to the destination of the executing instruction, if there is a match then a hazard is detected, and the fetch and decode stages must be stalled. The “stall_pipeline_low” control signal is an active low signal connected to the enable pin of the fetch and decode pipeline registers. Note that the stall condition also requires the control signals to be set to zero so that NOP are injected into the pipeline for the duration of the stall. Stalls are set to 2 clock cycles

Branching hazards

In a pipelined CPU branching involves data hazards and branch hazards. For data hazards, the branch operation must not be executed until the branch condition has been updated, therefore a stall may be necessary in between the SUB and TEST instructions. In this CPU design all branches are predicted as “not taken” meaning that the instructions following a branch instruction are fetched and decoded normally and must be flushed from the pipeline if the branch is taken.

Flushing

When a branch is taken, the fetch and decode stages are flushed, meaning all data and control signals must be cleared. The “flush_f_reg” and “flush_d_reg” signals are connected to the reset pins of the fetch and decode pipeline registers, as well as the controller which must de-assert all control signals to a NOP state. A flush takes 1 clock cycle.

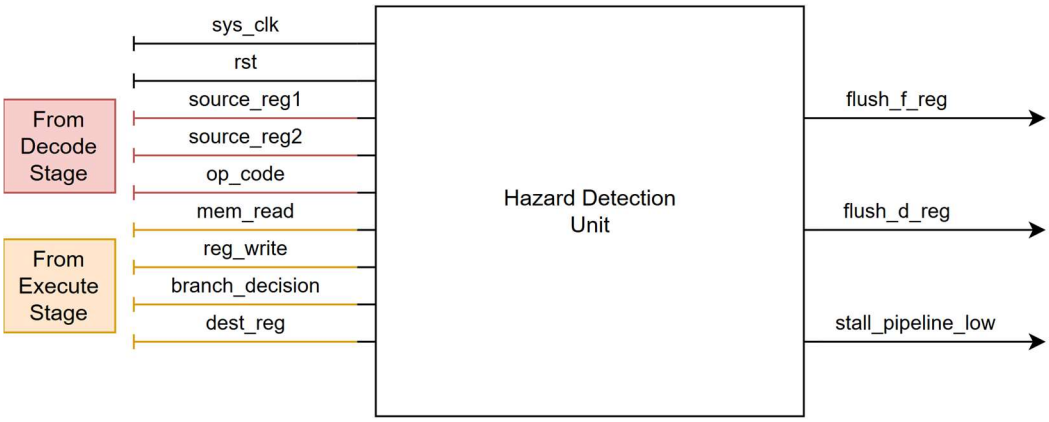


Figure 4: Hazard Detection Unit

Execute stage

The execute stage is the third stage and can be seen in Figure 5: Execute stage diagram, where the main components can be seen

Components:

- ALU – Responsible for all arithmetic operations for format A and TEST instruction for flags
- Branch unit – calculates the branch address and determines if the branch condition has been met
    - Uses ALU flags from previous TEST instruction
    - Takes OPCODE as input to determine case

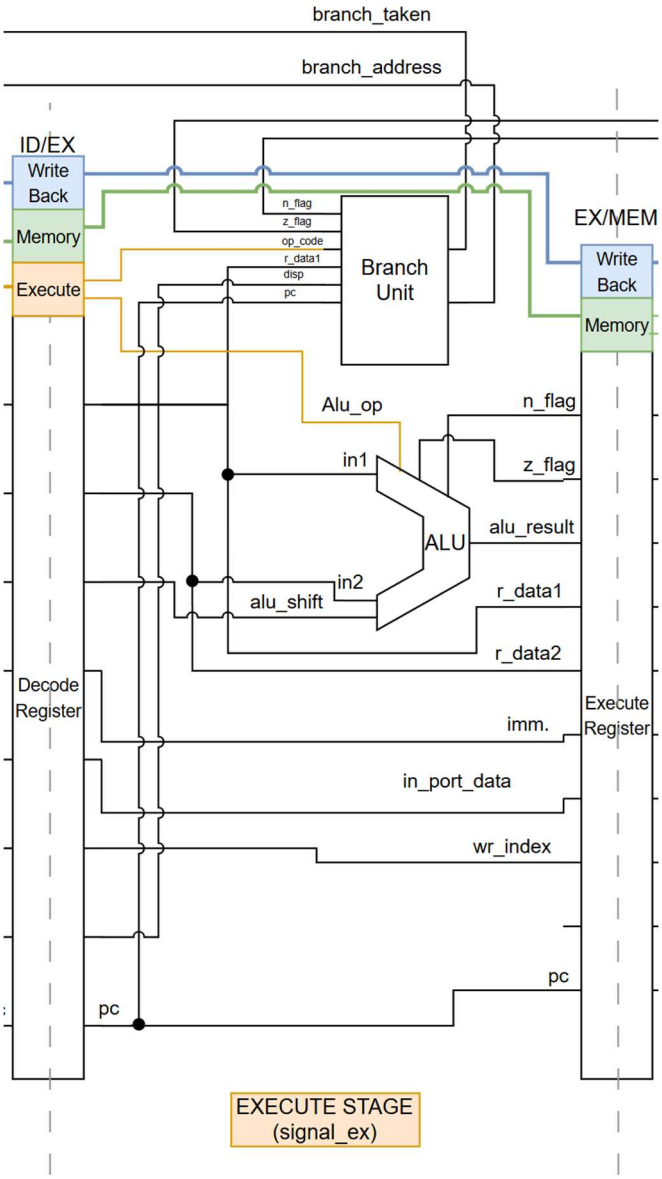


Figure 5: Execute stage diagram

Memory Stage

The memory stage is the fourth stage and can be seen in Figure 6: Memory stage diagram, where the data memory unit can be seen.

Data memory – performs load and store memory operations

- Memory Mapped IO
    - LOAD from memory address 0XFFF0 will use the external constraint signal connected to the dipswitch instead of reading from RAM
    - STORE to memory address 0XFFF2 will write to the external constraint signal connected to the seven-segment display component instead of writing to RAM
    - STORE to memory address in range 0XFC00 to 0XFDFF will write to the external constraint signal connected to the video memory component instead of writing to RAM

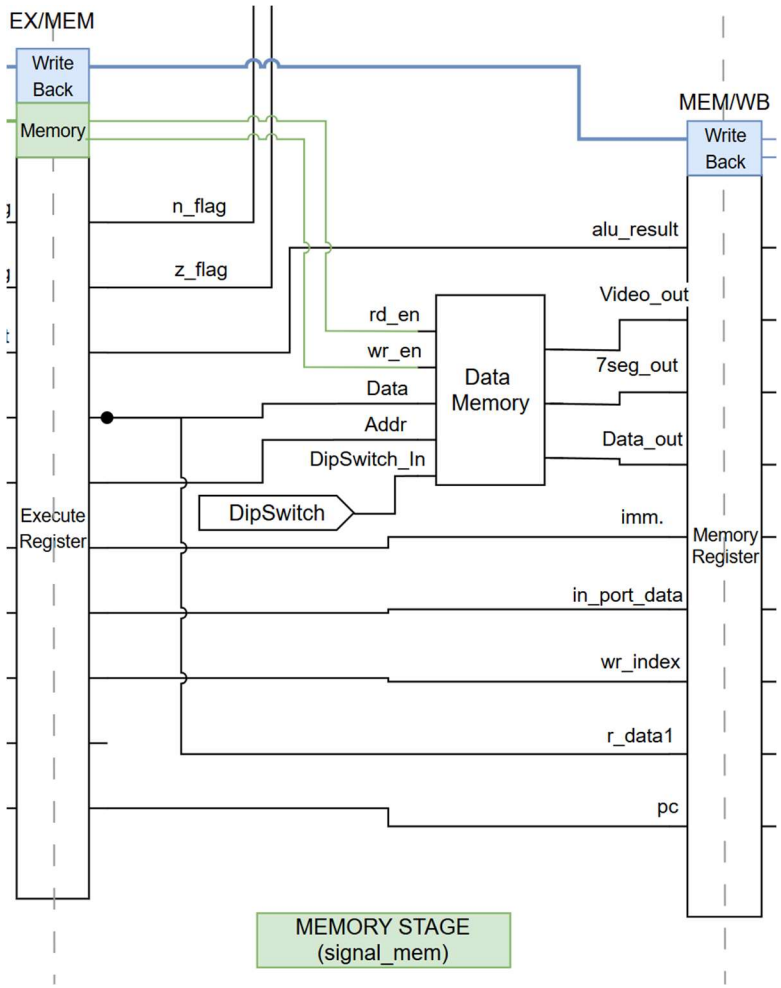


Figure 6: Memory stage diagram

Write-back Stage

The write-back stage is the 5th and final stage of the pipeline and can be seen in Figure 7: Write-back stage diagram. The write-back stage is essentially composed of a multiplexor for the writeback data and other external output values. Note: the immediate concatenation unit is not depicted in the diagram. The immediate concatenation unit is responsible for ensuring that register 7 is written to correctly for all "load immediate" instructions.

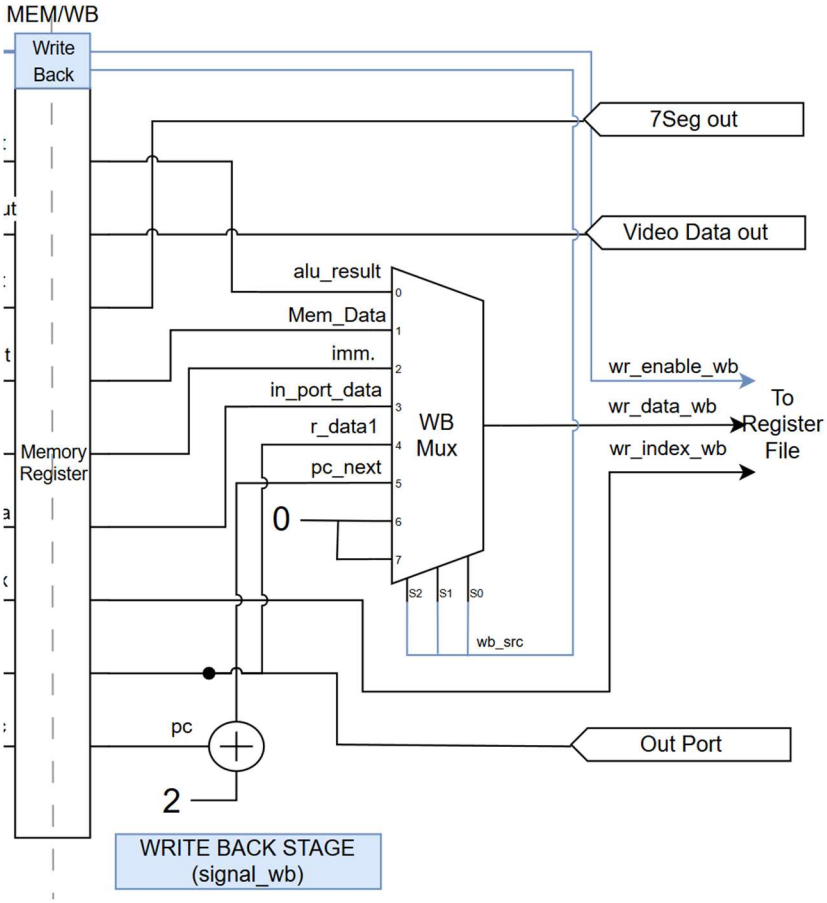


Figure 7: Write-back stage diagram

Memory Manager

The memory depicted in the block diagram is a simplified model that shows separate instruction and data memory, which improves the clarity of the diagram but hides the actual implementation. The actual structure of the memory management unit is shown in Figure 8: Memory manager. The ROM and RAM are implemented using Xilinx XPM memory modules.

Memory Manager functional requirements:

- Byte addressable (CPU Specification)
  - 1024 bytes means 1024 address values
- Word Aligned (CPU Specification)
  - Only even-valued addresses are valid
- Instruction memory from 0x0000 to 0x03FF is in ROM

- Instruction memory from 0x0400 to 0x04FF is in RAM
- Memory mapped I/O
    - 0XFFF0 input from Dipswitch
    - 0XFFF2 output to seven-segment display
    - 0XFC00 to 0xFDFF output to video memory (VGA display)
- ROM initialized with bootloader code
    - Bootloader loads “user code” into RAM
    - Bootloader creates boot vectors at top of RAM which will vector to the user code

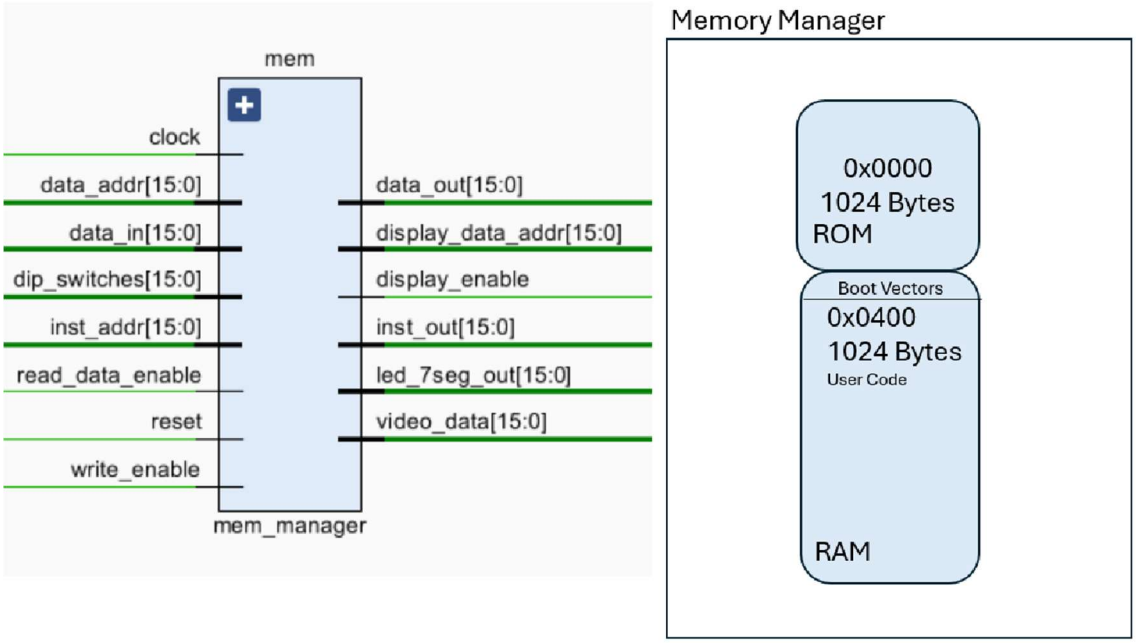


Figure 8: Memory manager

Controller

The controller is implemented as a Moore type finite state machine.

Inputs:

- External
    - Stm_sys_clk – from STM board
    - Rst_execute – from basys3 button
    - Rst_load – from basys3 button
- Internal
    - Opcode – From data path

Outputs:

- All control signals for CPU data path
    - Implemented as VHDL Record types (like C struct)

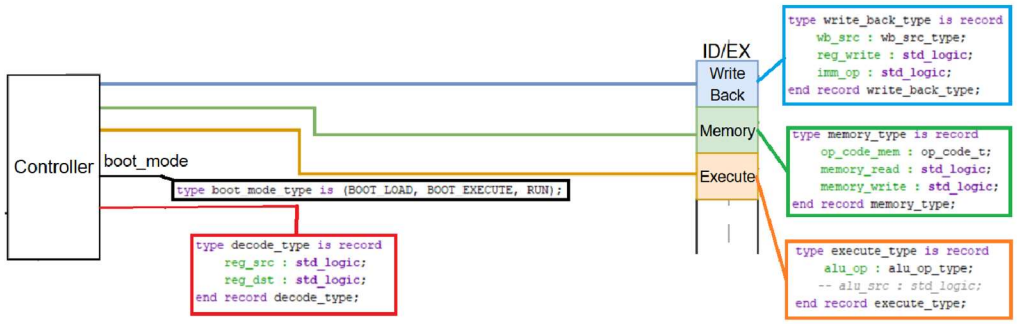


Figure 9: Controller output signals

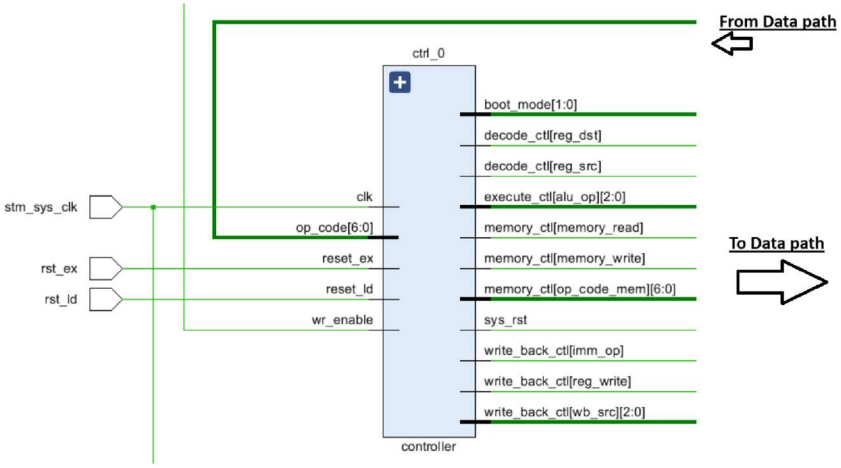


Figure 10: Controller schematic

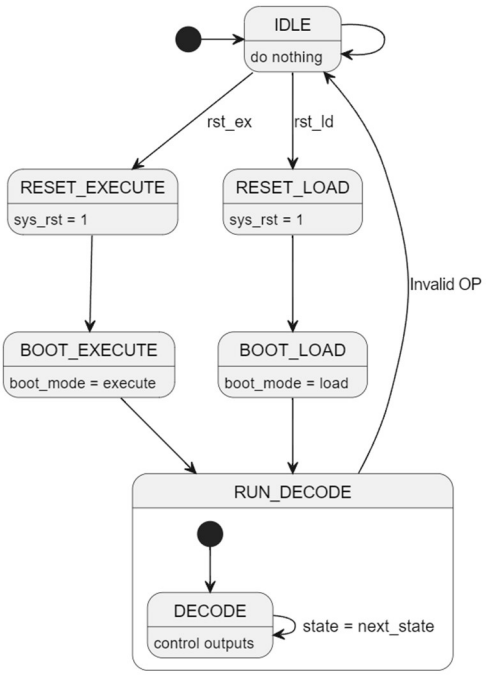


Figure 11: Controller state diagram

Results

Synthesis Report

The full synthesis report can be viewed in appendix D, the slice logic table is presented here in Figure 12: Synth report slice logic.

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	2276	0	20800	10.94
LUT as Logic	2020	0	20800	9.71
LUT as Memory	256	0	9600	2.67
LUT as Distributed RAM	256	0		
LUT as Shift Register	0	0		
Slice Registers	714	0	41600	1.72
Register as Flip Flop	714	0	41600	1.72
Register as Latch	0	0	41600	0.00
F7 Muxes	278	0	16300	1.71
F8 Muxes	8	0	8150	0.10

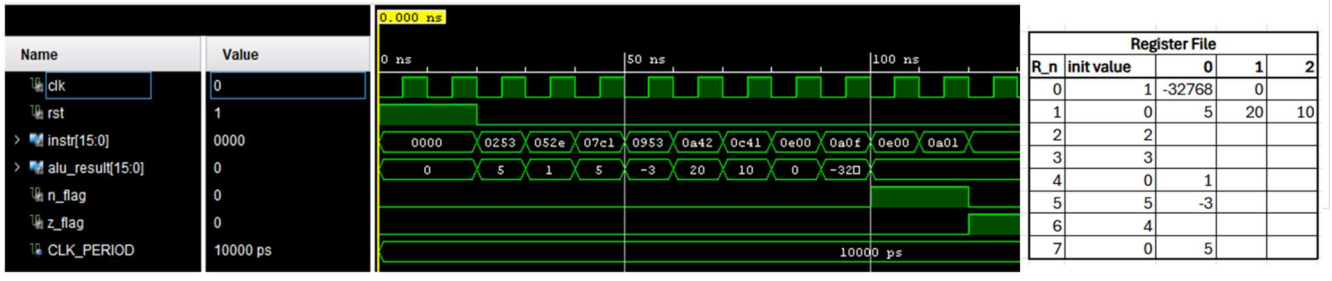
Figure 12: Synth report slice logic

Format A

Format A is essentially all arithmetic operations except for the IN, OUT and RETURN instructions. The format A test results are shown in Figure 13: Format A test results, which demonstrates the correct output for all format A arithmetic operations when compared to the expected results. The large table at the top of figure lists the test code that was run and tabulates the test results and the expected output, the wave diagram shows the simulation results, and the register file table to the right of the figure shows the expected register values for each instruction step.

Format A Test

Test:1 - Format A					Binary			Decimal			Expected		Testbench		
Instruction #	Code	Function	OpCode	Format	ra	rb	rc	ra	rb	rc	Result	Flag (z/n)	Result	Flag	Pass?
0	ADD R1, R2, R3	R1 = R2 + R3	0000001	A1	001	010	011	1	2	3	5	x	5		y
1	SUB R4, R5, R6	R4 = R5 - R6	0000010	A1	100	101	110	4	5	6	1	x	1		y
2	MUL R7, R0, R1	R7 = R0 * R1	0000011	A1	111	000	001	7	0	1	5	x	5		y
3	NAND R5, R2, R3	R5 = R2 NAND R3	0000100	A1	101	010	011	5	2	3	-3	x	-3		y
4	SHL R1, #2	R1 = R1 << 2	0000101	A2	001	00	0010	1	0	2	20	x	20		y
5	SHR R1, #1	R1 = R1 >> 1	0000110	A2	001	00	0001	1	0	1	10	x	10		y
6	TEST R0	Check if R0 is zero or negative	0000111	A3	000	000	000	0	0	0	x	0 0	x	0 0	y
7	SHL R0, #15	R0 = R0 << 15	0000101	A2	000	00	1111	0	0	15	-32768	x x	-32768	x x	y
8	TEST R0	Check if R0 is zero or negative	0000111	A3	000	000	000	0	0	0	x	0 1	x	0 1	y
9	SHL R0, #1	R0 = R0 << 1	0000101	A2	000	00	0001	0	0	1	1	x x	1	x x	y
10	TEST R0	Check if R0 is zero or negative	0000111	A3	000	000	000	0	0	0	x	1 0	x	1 0	y



Register File				
R_n	init value	0	1	2
0	1	-32768	0	
1	0	5	20	10
2	2			
3	3			
4	0	1		
5	5	-3		
6	4			
7	0	5		

Figure 13: Format A test results

Format B

Format B includes all branching instructions which are define by the ISA.

In Figure 14: Branching test results, the branch taken signal and the pipeline flush signals can be clearly seen, as well as noting that the program counter is updated to the branch address while the control signals are set to NOP during the flush state. The flush state takes 1 clock cycle.

In Figure 15: Format B test-1 results, data hazards are tested to confirm that a branch condition will be updated correctly even when data hazards exist, ensuring that a TEST instruction produces flags that are representative of the values that are expected to be tested. The assembly test code is shown to the left of the figure, with table showing the expected arithmetic results and correctly updated register values. In the wave image note the stall signal is asserted when a data hazard is detected, and the program counter values are held for additional clock cycles while the value is being written back to the register.

In Figure 16: Format B test-2 results, the assembly test code is shown to the left of the figure, the BR.SUB, BRR.z, BRR and RETURN instructions are tested, the wave image shows the register values which match the expected results displayed in the table above it.

```
branch_hazard: process ( branch_decision, rst )
begin
    if(rst='1') then
        flush_f_reg <= '0';
        flush_d_reg <= '0';
    elsif (branch_decision = '1') then
        --flush := '1';
        flush_f_reg <= '1';
        flush_d_reg <= '1';
    else
        --flush := '0';
        flush_f_reg <= '0';
        flush_d_reg <= '0';
    end if;
end process;
```

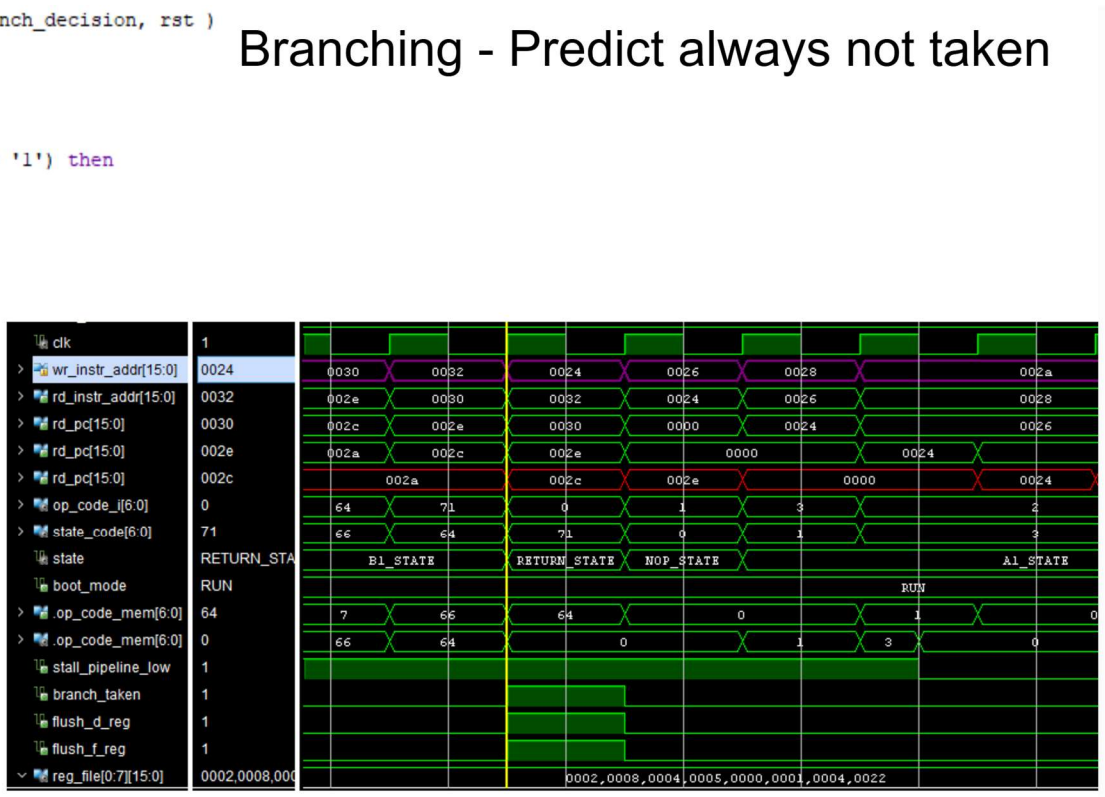


Figure 14: Branching test results

Format B Test 1 - Data Hazards

```
ORG  0x0210
.CODE
        IN R0 ; 02  ;
        IN R1 ; 03  ;
        IN R2 ; 01
        IN R3 ; 05  ;
        ADD R1, R1, R2
        SUB R2, R1, R0
        SUB R1, R3, R2

END
```

rReg Index	Value n=0
0	2
1	3
2	1
3	5

4 3

2

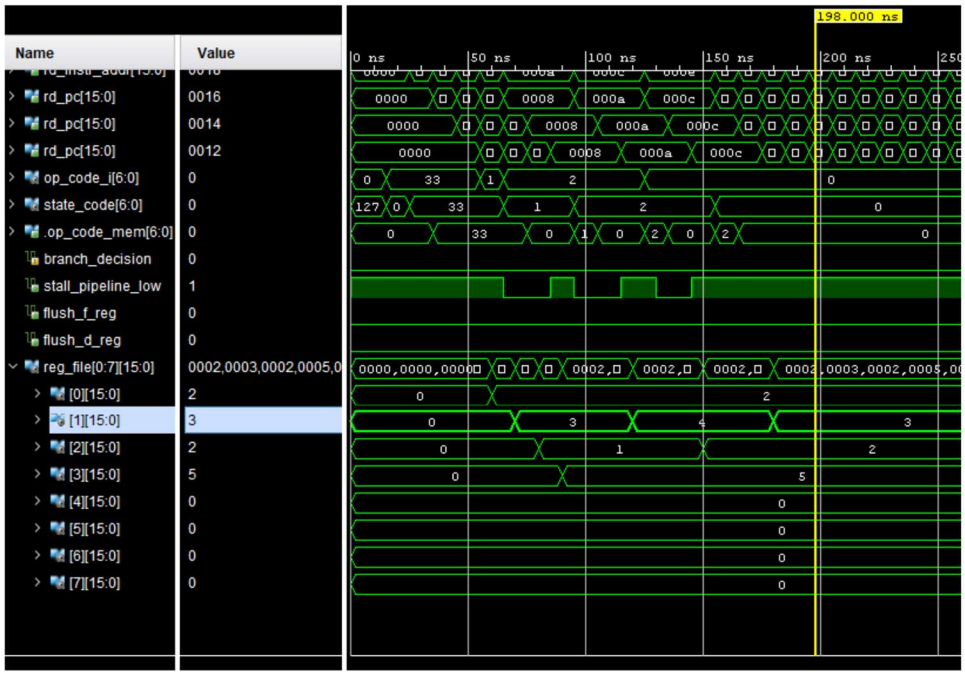


Figure 15: Format B test-1 results

Format B Test 2 - Branching

hex	
4200	IN R0; 02
4240	IN R1; 03
4280	IN R2; 01
42C0	IN R3; 05
4300	IN R4; 0
4340	IN R5; 01
4380	IN R6; 05
43C0	IN R7; 00
8D0A	BR.SUB R4,10
8000	BRR 0
028D	ADD R2, R1, R5
0642	MUL R1, R0, R2
05B5	SUB R6, R6, R5
0F80	TEST R6
8402	BRR.z 2
81FB	BRR -5
8E00	RETURN

Register Init

rReg Index	Value n=0	1	2	3	4	5
0	2	2	2	2	2	2
1	3	8	18	38	78	158
2	1	4	9	19	39	79
3	5	5	5	5	5	5
4	0	0	0	0	0	0
5	1	1	1	1	1	1
6	5	4	3	2	1	0
7	0	10	10	10	10	10

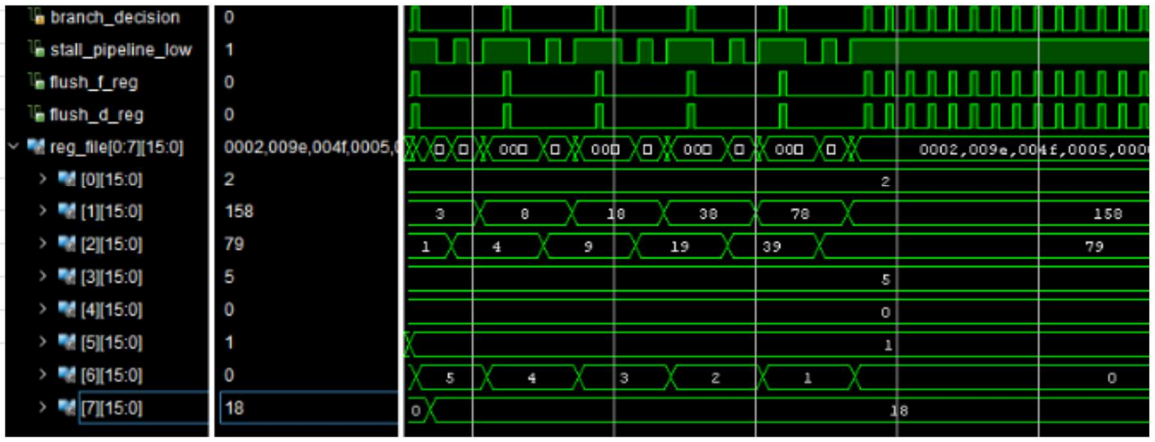


Figure 16: Format B test-2 results

Format L

Format L contains all memory operations such as LOAD, STORE, LOADIMM and MOV. In Figure 17: Format L test results can be seen, the assembly test code is shown on the left of the figure and includes all format L instructions as well as some arithmetic operations to test a “read after load” hazard condition which occurs when an operation using a register value (ADD in this case) is executed directly after a LOAD instruction that loads the register value that is needed. The wave image shows the register values with R5 highlighted. The output matches the expected values tabulated in the table above.

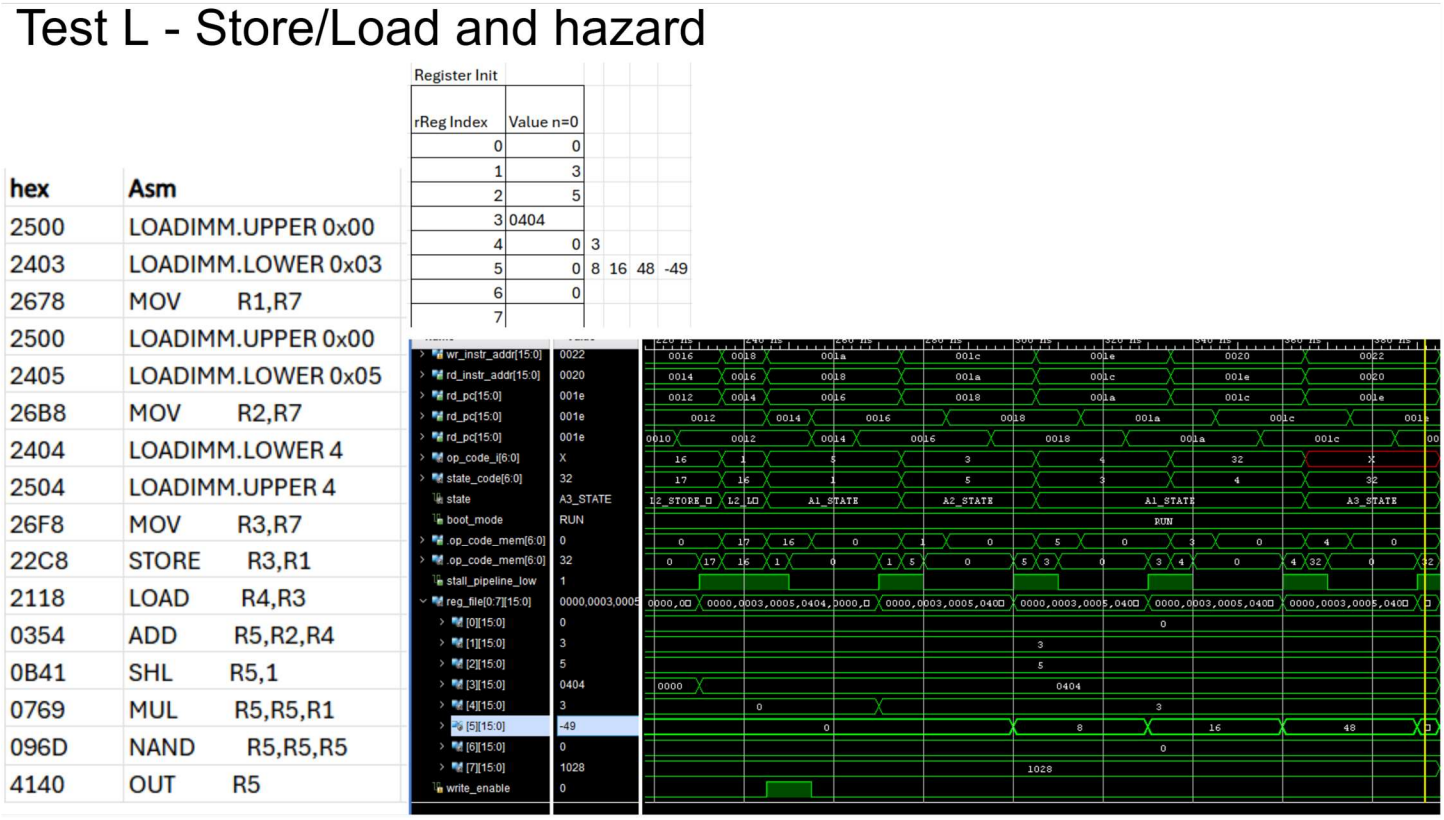


Figure 17: Format L test results

Final Test

The final test requires all instruction formats to be working correctly, as well as data hazards and branch hazards being handled, additionally an overflow flag was added. The overflow flag will set the ALU result to 0 and assert the overflow flag which will be stored in the register file with the result. Figure 18: Final test results shows the final test results where the assembly code can be seen on the left-hand side, the expected results on the right-hand side, and the simulation output in the wave figure at the bottom of the figure. Note that value in register 4 (HEX) matches the expected results (DEC), and the overflow condition occurs when the output exceeds 32,767.

```
0410 - 25FF main:          loadimm.upper DipSwitches.hi
0412 - 24F0                loadimm.lower DipSwitches.lo
0414 - 21B8                load          r6,r7
0416 - 257F                loadimm.upper DipSwitchMask.hi
0418 - 24FF                loadimm.lower DipSwitchMask.lo
041A - 09B7                nand          r6,r6,r7
041C - 09B6                nand          r6,r6,r6
041E - 2500                loadimm.upper 0x00
0420 - 2401                loadimm.lower 0x01
0422 - 2738                mov           r4,r7
0424 - 26F8                mov           r3,r7
0426 - 0F80                test          r6
0428 - 840D                brr.z         Done
042A - 05B3                sub           r6,r6,r3
042C - 0F80                test          r6
042E - 840A                brr.z         Done
0430 - 2500                loadimm.upper 0x00
0432 - 2402                loadimm.lower 0x02
0434 - 2778                mov           r5,r7
0436 - 0725 loop:          mul           r4,r4,r5
0438 - 036B                add           r5,r5,r3
043A - 05B3                sub           r6,r6,r3
043C - 0F80                test          r6
043E - 8402                brr.z         Done
0440 - 81FB                brr           loop
0442 - 25FF Done:          loadimm.upper LedDisplay.hi
0444 - 24F2                loadimm.lower LedDisplay.lo
0446 - 23E0                store         r7,r4
0448 - 81FD                brr           Done
```

Final Test - Factorial with overflow

Register Init												
rReg Index	Value n=0											
0	0											
1	0											
2	0											
3	0								OVERFLOW			
4	0	1	2	6	24	120	720	5040	40320			
5	0	2	3	4	5	6	7	8	9	10		
6	9	9	8	7	6	5	4	3	2	1	0	
7	0											

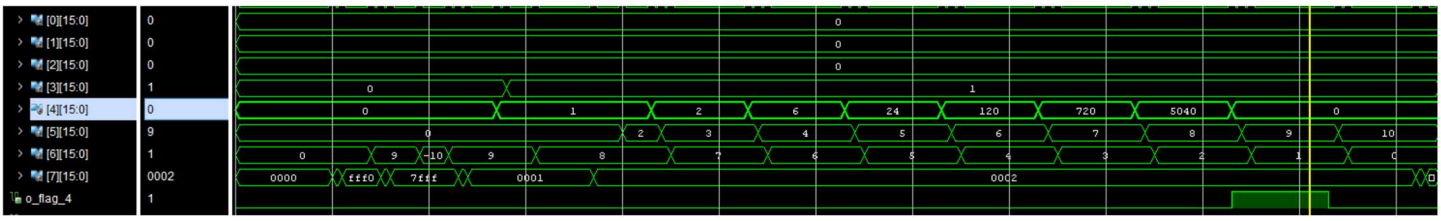


Figure 18: Final test results

Discussion

Design Limitations

Data forwarding was not implemented, instead we implemented stalling technique for hazard control. Forwarding would make the CPU more efficient, so that further work could be to implement forwarding instead of stalling for data hazard control. Overall, Format A, B and L instructions were all implemented and worked correctly.

Challenges and Solutions

Pipeline Synchronization: Ensuring that data flowed correctly through all five pipeline stages without misalignment was a major challenge. This was addressed by using structured signal declarations and well-documented inner stage signal name suffixes. For example, “signal_name_x” where x is replaced by the pipeline stage that the signal corresponds to such as f, d, ex, mem and wb.

Data Hazards: A major challenge was managing read-after-write (RAW) hazards, where an instruction attempts to read a register before the previous instruction writes to it. We addressed this by introducing stalling logic in the decode stage. When a RAW hazard is detected, the pipeline is stalled for one or more cycles until the write-back completes.

Branch Hazards: Early versions of the CPU did not flush incorrect instructions on taken branches, leading to incorrect results. A simple pipeline flush mechanism was added to the decode stage to address this issue.

Result Analysis

The synthesized CPU meets timing constraints and operates as expected under test conditions. We also tested the CPU with bootloader and debug display over VGA, and it operates as expected.

Conclusion

In conclusion, we successfully implemented a 16-bit pipelined CPU. The design achieved functional correctness, demonstrating effective instruction execution and hazard management. Future improvements include optimizing pipeline efficiency and expanding the instruction set.

References

[1] J. L. Hennessy and D. A. Patterson, Computer Organization & Design: The hardware/software Interface, San Francisco: Morgan Kaufmann Publishers, Inc., 2014.

Appendices

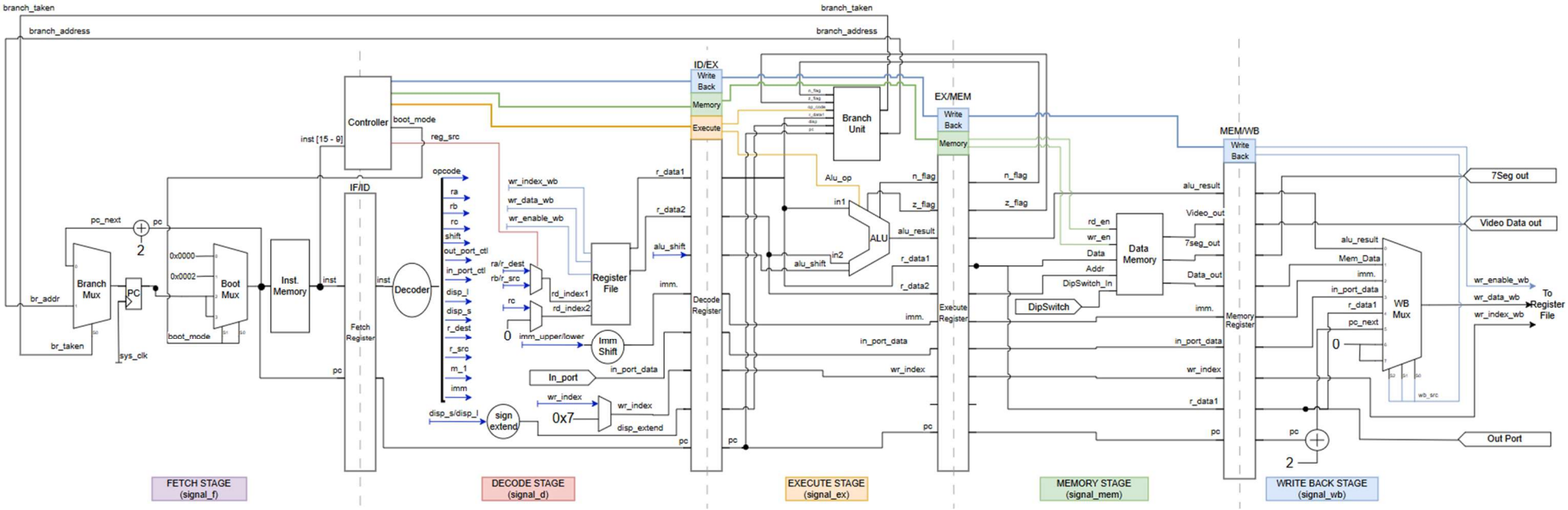
Appendix A: Code

Project repo: https://github.com/bdionello/16bit-CPU-basys3

Branch: main_boot

Directory: sub_modules

Appendix B: Full system block diagram



Appendix C: ISA

Mne-monic	Op-code	Function	Type	Syntax
NOP	0	Nothing	A0	*NOP*
ADD	1	R[ra] ← R[rb] + R[rc];	A1	*ADD ra,rab,rc*
SUB	2	R[ra] ← R[rb] – R[rc];	A1	*SUB ra,rb,rc*
MUL	3	R[ra] ← R[rb] × R[rc];	A1	*MUL ra,rb,rc*
NAND	4	R[ra] ← R[ra] NAND R[rb]:	A1	*NAND ra,rb,rc*
SHL	5	n:=c1<3…0>): (n|0) → (R[ra]<15…0>←R[ra]<15-n…0>#(n@0)): (n=0) →() ;	A2	*SHL ra#n*
SHR	6	n:=c1<3…0>): (n|0) → (R[ra]<15…0>←(n@0)#R[ra]<15…n>): (n=0) →() ;	A2	*SHR ra#n*
TEST	7	(R[ra) = 0) →Z ← 1; else →Z ← 0: (R[ra] < 0) →N ← 1; else →N ← 0;	A3	*TEST ra*
OUT	32	OUT.PORT ← R[ra];	A3	*OUT ra*
IN	33	R[ra] ← IN.PORT;	A3	*IN ra*
BRR	64	PC ← PC'+2*disp.l {sign extended 2's complement}	B1	*BRR +disp.l*
BRR.N	65	(N=1) → PC ← PC+2*disp.l {sign extended 2's complement}; (N=0) → PC ← PC+2 { 2's complement};	B1	*BRR.N +disp.l*
BRR.Z	66	(Z=1) → PC ← PC+2*disp.l {sign extended 2's complement}; (Z=0) → PC ← PC+2 { 2's complement};	B1	*BRR.Z +disp.l*
BR	67	PC ← R[ra]+2*disp.s {sign extended 2's complement}	B2	*BR ra+disp.s*
BR.N	68	(N=1) → PC ← R[ra] {word aligned}+2*disp.s {sign extended 2's complement}; (N=0) → PC ← PC+2 { 2's complement};	B2	*BR.N ra+disp.s*
BR.Z	69	(Z=1) → PC ← R[ra] {word aligned}+2*disp.s {sign extended 2's complement}; (Z=0) → PC ← PC+2 { 2's complement};	B2	*BR.Z ra+disp.s*
BR.SUB	70	r7 ← PC + 2; PC ← R[ra] {word aligned}+2*disp.s {sign extended 2's complement};	B2	*BR.SUB ra+disp.s*
RETURN	71	PC ← r7;	A0	*RETURN*
LOAD	16	R[r.dest] ← M[R[r.src]];	L2	*LOAD r.dest, r.src* *LOAD r.drst, @r.src*
STORE	17	M[R[r.dest]] ← R[r.src];	L2	*STORE r.dest, r.src* *STORE @r.dest, r.src*
LOADIMM	18	(m.l=1) →R7<15…8>← imm: (m.l=0) →R7<7…0>← imm;	L1	*LOADIMM.upper #n* *LOADIMM.lower #n*

Figure 19: Instruction set description

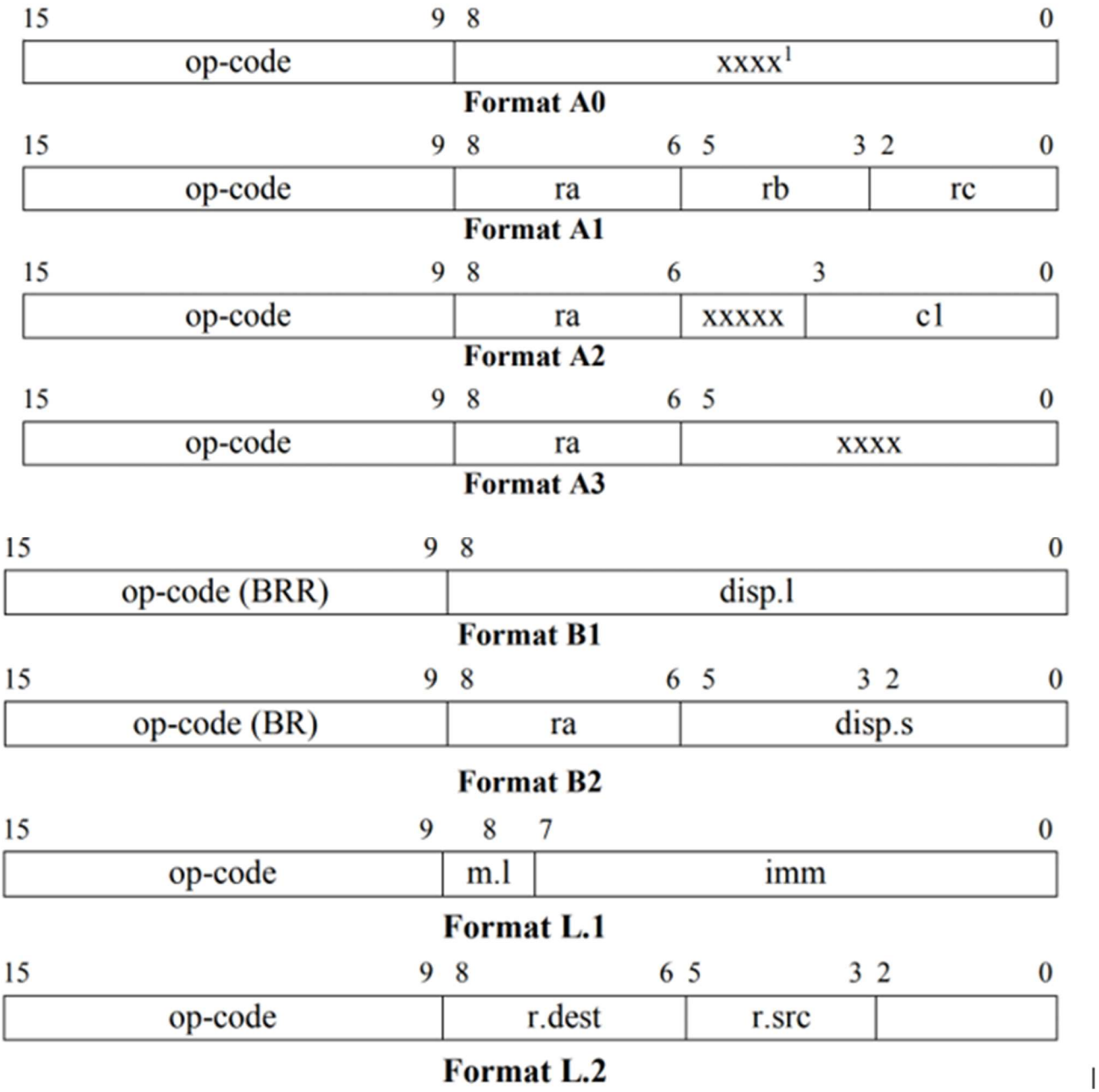


Figure 20: Instruction format types

Appendix D: Synthesis Report


```
------------------------------------------------------------------------------------------------------------
| Tool Version : Vivado v.2017.4 (win64) Build 2086221 Fri Dec 15 20:55:39 MST 2017
| Date         : Mon Apr  7 13:29:33 2025
| Host         : DESKTOP-BJ3E32D running 64-bit major release  (build 9200)
| Command      : report_utilization -file cpu_top_utilization_synth.rpt -pb cpu_top_utilization_synth.pb
| Design       : cpu_top
| Device       : 7a35tcpg236-1
| Design State : Synthesized
------------------------------------------------------------------------------------------------------------
```

Utilization Design Information

Table of Contents

1. Slice Logic

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	2276	0	20800	10.94
LUT as Logic	2020	0	20800	9.71
LUT as Memory	256	0	9600	2.67
LUT as Distributed RAM	256	0		
LUT as Shift Register	0	0		
Slice Registers	714	0	41600	1.72
Register as Flip Flop	714	0	41600	1.72
Register as Latch	0	0	41600	0.00
F7 Muxes	278	0	16300	1.71
F8 Muxes	8	0	8150	0.10

* Warning! The Final LUT count, after physical optimizations and full implementation, is typically lower. Run opt_design after synthesis, if not already completed, for a more realistic count.

1.1 Summary of Registers by Type

Total	Clock Enable	Synchronous	Asynchronous
0	_	-	-
0	_	-	Set
0	_	-	Reset
0	_	Set	-
0	_	Reset	-
0	Yes	-	-
0	Yes	-	Set
168	Yes	-	Reset
15	Yes	Set	-
531	Yes	Reset	-

2. Memory

Site Type	Used	Fixed	Available	Util%
Block RAM Tile	0.5	0	50	1.00
RAMB36/FIFO*	0	0	50	0.00
RAMB18	1	0	100	1.00
RAMB18E1 only	1			

* Note: Each Block RAM Tile only has one FIFO logic available and therefore can accommodate only one FIFO36E1 or one FIFO18E1. However, if a FIFO18E1 occupies a Block RAM Tile, that tile can still accommodate a RAMB18E1

3. DSP

Site Type	Used	Fixed	Available	Util%
DSPs	1	0	90	1.11
DSP48E1 only	1			

4. IO and GT Specific

Site Type	Used	Fixed	Available	Util%
Bonded IOB	72	0	106	67.92
Bonded IPADs	0	0	10	0.00
Bonded OPADs	0	0	4	0.00
PHY_CONTROL	0	0	5	0.00
PHASER_REF	0	0	5	0.00
OUT_FIFO	0	0	20	0.00
IN_FIFO	0	0	20	0.00
IDELAYCTRL	0	0	5	0.00
IBUFDS	0	0	104	0.00
GTPE2_CHANNEL	0	0	2	0.00
PHASER_OUT/PHASER_OUT_PHY	0	0	20	0.00
PHASER_IN/PHASER_IN_PHY	0	0	20	0.00
IDELAYE2/IDELAYE2_FINEDELAY	0	0	250	0.00
IBUFDS_GTE2	0	0	2	0.00
ILOGIC	0	0	106	0.00
OLOGIC	0	0	106	0.00

5. Clocking

Site Type	Used	Fixed	Available	Util%
BUFGCTRL	3	0	32	9.38
BUFIO	0	0	20	0.00
MMCME2_ADV	0	0	5	0.00
PLLE2_ADV	0	0	5	0.00
BUFMRCE	0	0	10	0.00
BUFHCE	0	0	72	0.00
BUFR	0	0	20	0.00

6. Specific Feature

Site Type	Used	Fixed	Available	Util%
BSCANE2	0	0	4	0.00
CAPTUREE2	0	0	1	0.00
DNA_PORT	0	0	1	0.00
EFUSE_USR	0	0	1	0.00
FRAME_ECCE2	0	0	1	0.00
ICAPE2	0	0	2	0.00
PCIE_2_1	0	0	1	0.00
STARTUPE2	0	0	1	0.00
XADC	0	0	1	0.00

7. Primitives

Ref Name	Used	Functional Category
LUT6	948	LUT
FDRE	531	Flop & Latch
LUT5	514	LUT
LUT4	443	LUT
LUT3	300	LUT
MUXF7	278	MuxFx
LUT2	265	LUT
RAMD64E	256	Distributed Memory
FDCE	168	Flop & Latch
CARRY4	60	CarryLogic
OBUF	42	IO
IBUF	30	IO
FDSE	15	Flop & Latch
LUT1	11	LUT
MUXF8	8	MuxFx
BUFG	3	Clock
RAMB18E1	1	Block Memory
DSP48E1	1	Block Arithmetic

8. Black Boxes

Ref Name	Used

9. Instantiated Netlists

Ref Name	Used