FACULTY OF ENGINEERING & TECHNOLOGY

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

ENCS4370 - Computer Architecture

Project #2: Multi-Cycle

Prepared by:

Marah Hamarsheh	**No:**1220281	**Section:** 3
Layan Salem	**No:** 1221026	**Section:** 3

Instructor: Aziz Qaroush

Date: 10-6-2025

Team Member Name	Team Member ID	Contributions
Marah Hamarsheh	1220281	1/2
Layan Salem	1221026	1/2
-	-	We work together

Processor Implementation (Tick One)	
Single Cycle	
Multi Cycle	✓
Pipelined	

In case of pipeline implementation (Tick the correct answer)		
	Implemented in RTL Code?	Verified and Correctly Worked?
Data hazards detection	-	-
Control hazards detection	-	-
Structural hazards detection	-	-
Forwarding	-	-
Stalling	-	-

Tick the correct answer						
Instruction	Did you implement this instruction in the RTL?		Did you write the verification code for this instruction?		Did the instruction Work perfectly when it has been verified?	
	Yes	No	Yes	No	Yes	No
OR Rd, Rs, Rt	Yes		Yes		50%	
ADD Rd, Rs, Rt	Yes		Yes		50%	
SUB Rd, Rs, Rt	Yes		Yes		50%	
CMP Rd, Rs, Rt	Yes		Yes		50%	
ORI Rd, Rs, Imm	Yes		Yes		50%	
ADDI Rd, Rs, Imm	Yes		Yes		50%	
LW Rd, Imm (Rs)	Yes		Yes		50%	
LDW Rd, Imm (Rs)	Yes		Yes		50%	

SDW Rd, Imm (Rs)	**Yes**		**Yes**		**50%**	
BZ Rs, Label	**Yes**		**Yes**		**50%**	
BGZ Rs, Label	**Yes**		**Yes**		**50%**	
BLZ Rs, Label	**Yes**		**Yes**		**50%**	
JR Rs	**Yes**		**Yes**		**50%**	
J Label	**Yes**		**Yes**		**50%**	
CALL Label	**Yes**		**Yes**		**50%**	
Tick one of the following						
My processor can execute only test programs consisting of one instruction only						**50%**
My processor can execute complete programs (A simulation screenshot must be provided as evidence)						**50%**

We tested the instructions, but it didn't work completely because of datapath had an issue that we didn't know how to print the ALU result, but it wrote to the Register correctly, and we noticed that the control unit changed correctly with other things.

The processor operates correctly as a multi-cycle architecture, executing instructions through five stages (IF, ID, EX, MEM, WB) that advance synchronously with each clock cycle, maintaining proper data flow and state preservation between stages.

Abstract:

This project aims to design and verify a simple 32-bit multi-cycle RISC processor using Verilog. Every instruction is carried out by the processor over a number of clock cycles, with separate phases for fetch, decode, execute, memory, and write-back. It supports a custom ISA with ALU, memory, and control flow instructions. There are 16 general-purpose registers in the architecture, with R14 as the return address and R15 as the program counter. Signals are produced by the control unit to control the Datapath at each cycle.

Table of Contents:

Table of Figures:

Datapath Components:

Program Counter (PC):

Input to the PC comes via a multiplexer that chooses between jump register (JR) values from Rs, branch target addresses computed from PC plus a sign-extended immediate, or PC+1 for sequential execution. In addition to forwarding the current instruction address to the instruction memory, the PC also calculates branch targets by adding the current PC to the sign-extended immediate value. And stores the return address for the CALL instruction. The next sequential instruction address is created by adding 1 to the current PC value using the PC Increment Adder (+1). Additionally, this PC+1 value is kept as the return address for the CALL instruction, which is written to register R14, and used by the multiplexer in the PC update logic.

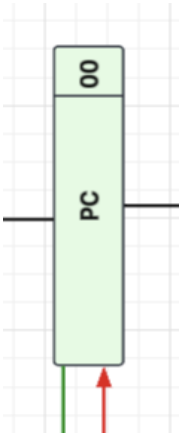


Figure 1: PS Block

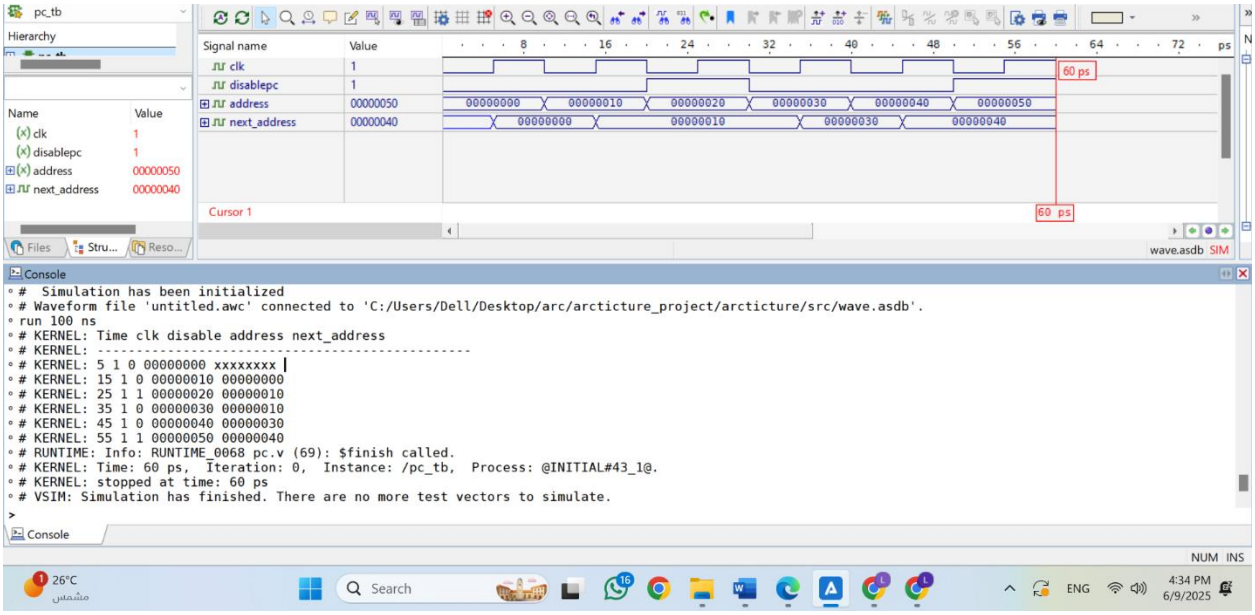


Figure 2: PC Testbench Result

Instruction Memory:

This component outputs the corresponding 32-bit instruction word after receiving the PC address as input. The instruction is then separated into its fields: the 14-bit immediate value spans bits [13:0], the first source register Rs is found in bits [21:18], the second source register Rt is found in bits [17:14], the destination register Rd occupies bits [25:22], and the 6-bit opcode is found in bits [31:26]. The next steps are then carried out using these extracted fields.

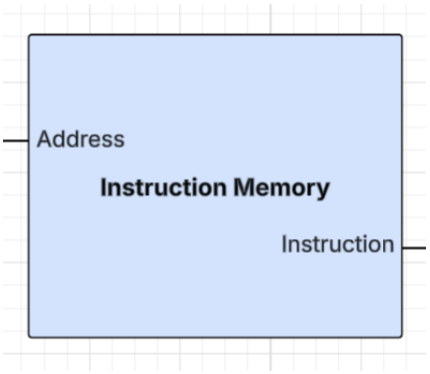


Figure 3: Instruction Memory Block

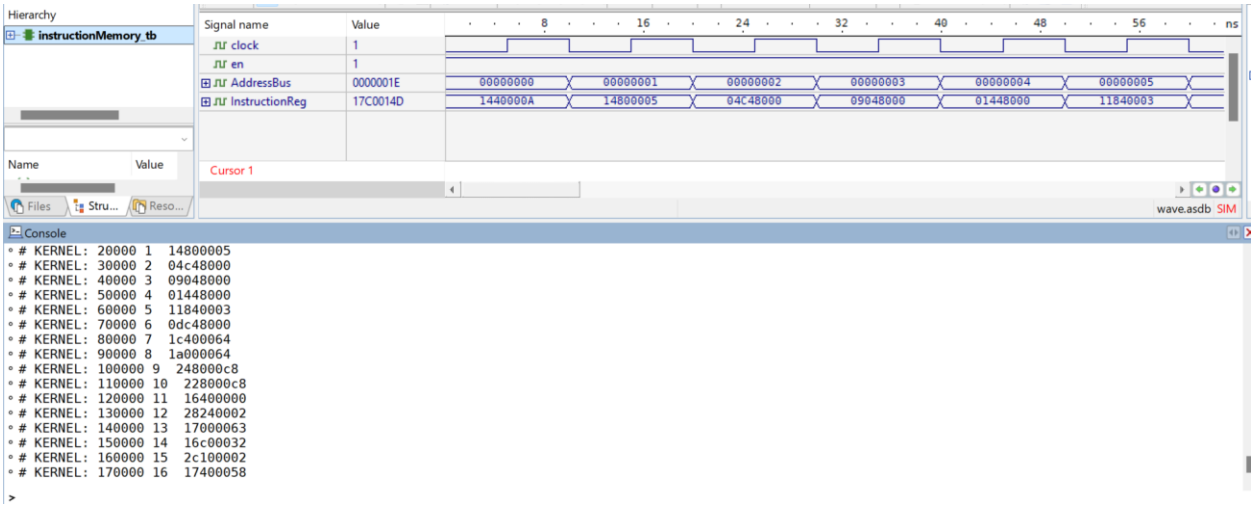


Figure 4: Instruction Memory Testbench Result

Register File:

The register file generates the data held in the two 4-bit addresses, Rs and Rt, that were taken from the instruction. Depending on the kind of instruction, a multiplexer then decides whether the second operand should originate from Rt or Rd, or Rd+1. Both Rs and Rt values are used as operands in OR, ADD, SUB, and CMP instructions. On the other hand, the immediate value takes the role of the Rt operand in ORI, ADDI, LW, and SW instructions, which only require the Rs data. Meanwhile, Rd and Rd+1 take the role of the Rt operand in the SDW instruction for the first and second cycles. And in the destination register selection is handled by a different multiplexer, which selects between Rd and the fixed register R14, which is where CALL instructions store the return address. For LDW instruction, the destination multiplexer alternates between Rd (first cycle) and Rd+1 (second cycle) to access consecutive registers. The control unit ensures Rd contains an even register number, throwing an exception if odd.

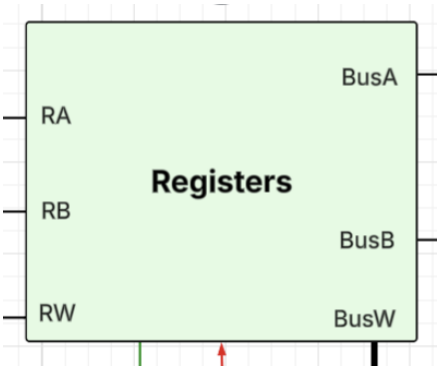


Figure 5: Register File Block

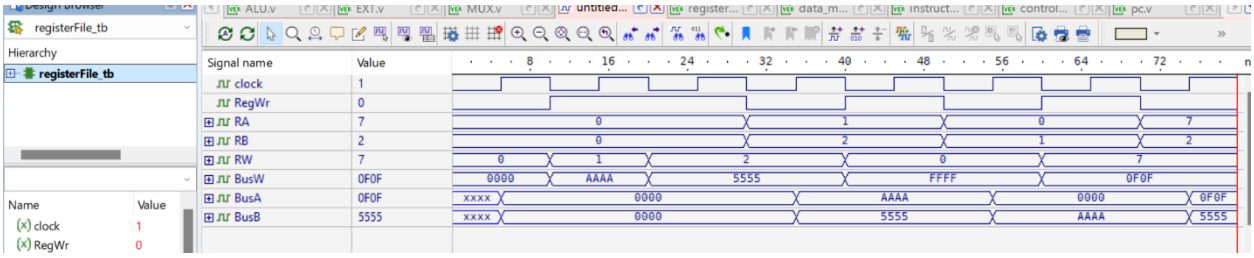
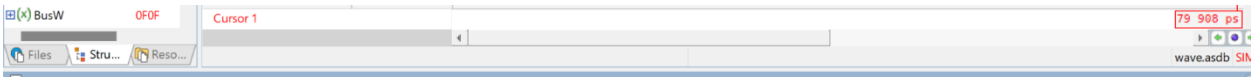


Figure 6: Register File Testbench Result



Sign/Zero Extension:

In order to process the 14-bit immediate field [13:0], the Sign/Zero Extension Unit expands it to a complete 32-bit value. For ORI instruction, it performs a zero extension by adding the upper 18 bits with zeros. For ALU and memory-related instructions, including ADDI, LW, SW, LDW, SDW, J, CLL, and branch operations, it performs sign extension by replicating the sign bit (bit 13) through bits [31:14]. The resulting 32-bit

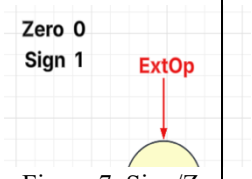


Figure 7: Sign/Zero Extension Block

value is sent to an ALU input multiplexer, where the ALUSrc control signal determines whether to use this extended immediate, the data from the Rt register, or, in the case of the second cycle of LDW/SDW, the immediate value incremented by one.

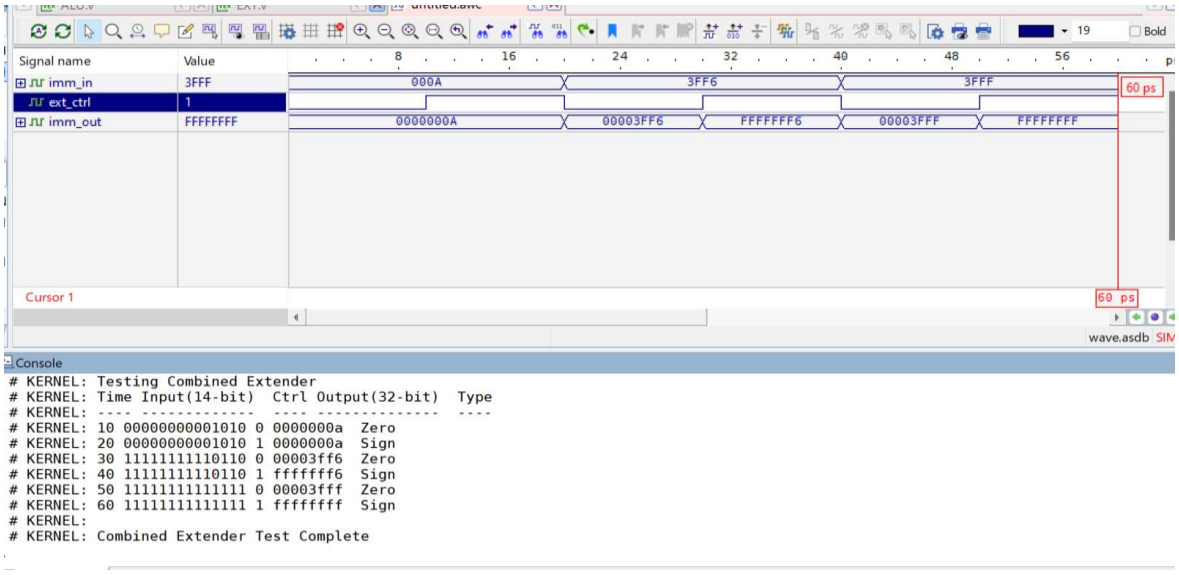


Figure 8: Sign/Zero Extension Testbench Result

ALU:

The ALUSrc control signal is used by the ALU Input 2 Multiplexer to choose the second operand for the ALU. It picks the sign- or zero-extended immediate value for ORI, ADDI, LW, and SW instructions, the immediate value + 1 for LDW/SDW instructions, 0 for BZ, BGZ, and BLZ instructions, and the Rt register value for OR, ADD, SUB, and CMP instructions. This multiplexer is necessary for differentiating between register-to-immediate and register-to-register operations. After receiving two 32-bit inputs from its input multiplexers, the ALU uses ALUOp control signals to carry out the necessary operation. It generates a zero, Negative, Positive flags, and a 32-bit result. For memory instructions like LDW/SDW, LW, and SW, it calculates the effective address. For CMP, it outputs 0, -1, or +1 based on the operand comparison.

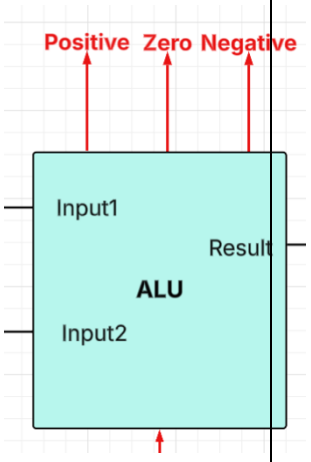


Figure 9: ALU Block

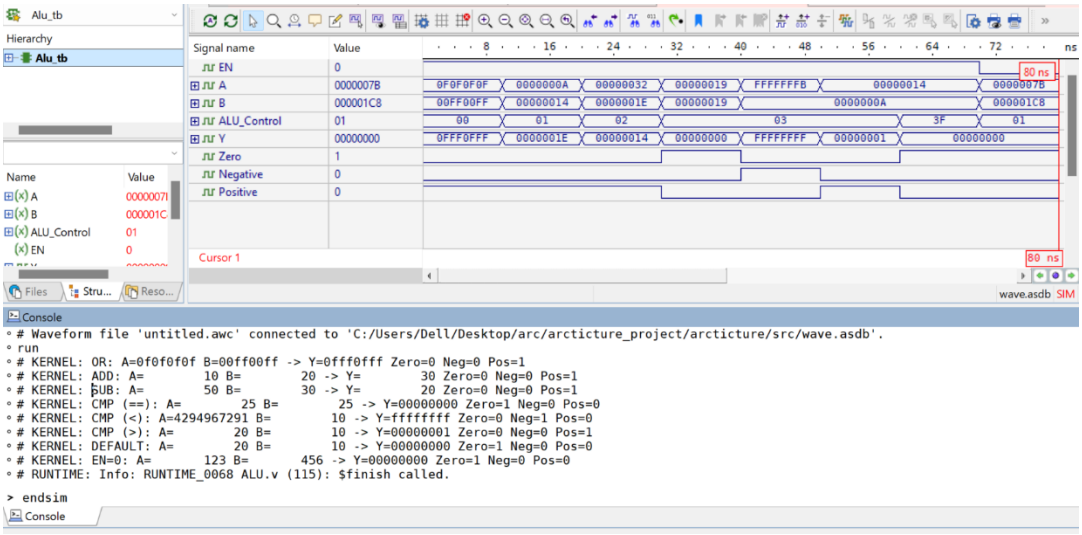


Figure 10: ALU Testbench Result

Data Memory:

The address of the data memory is obtained from the ALU result (effective address), the ldw_sdw for LDW/SDW instruction, with the MemRead/MemWrite control signals, determines the operation. It sends data to the write-back stage for load instructions LW and LDW. The memory data input is used to receive data from a register to store instructions SW and SDW.

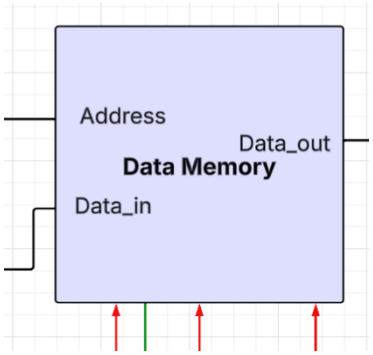


Figure 11: Data Memory Block

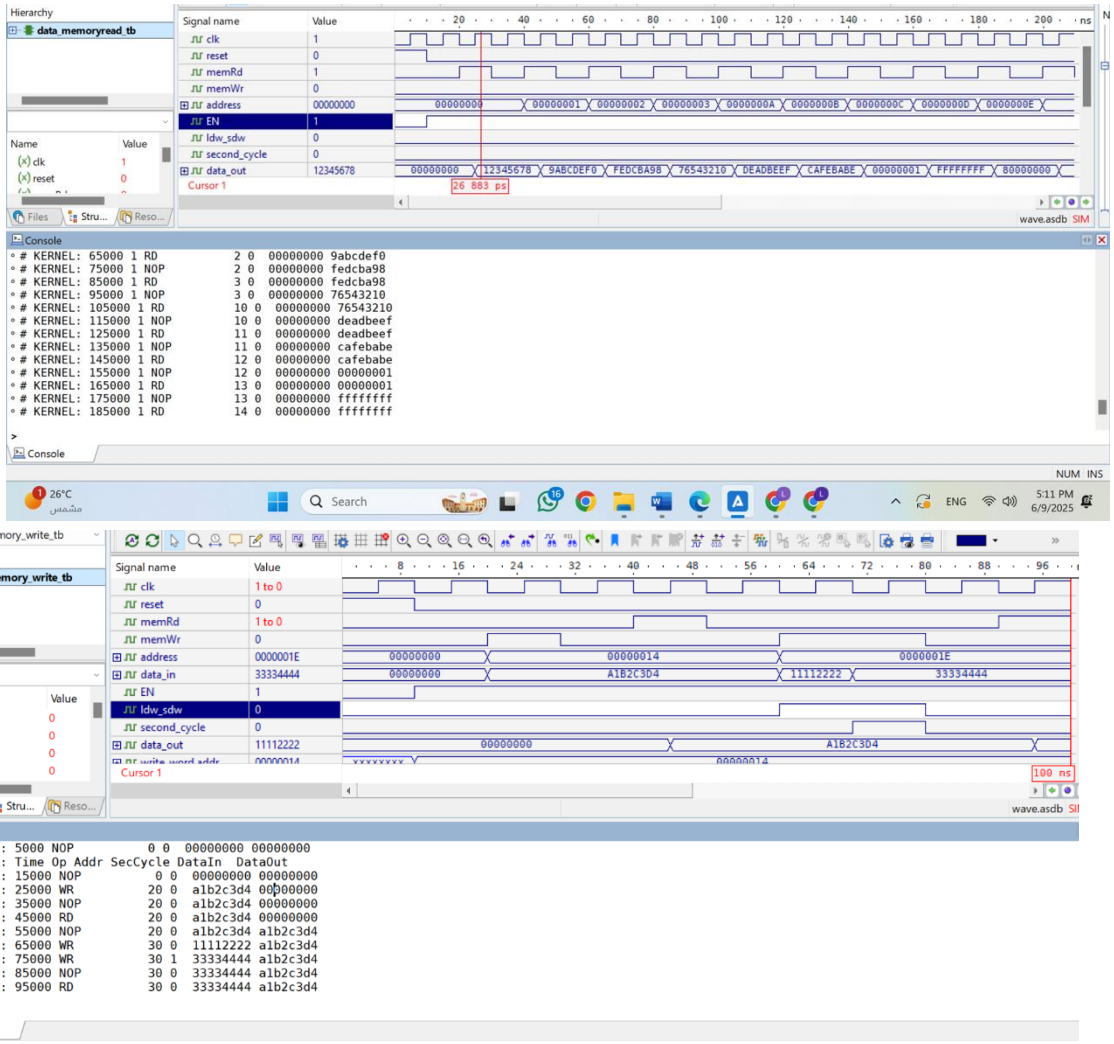


Figure 13: Data Memory Testbench Write on Memory Result

Write-Back Data Multiplexer:

This multiplexer uses the WBdata control signal to choose which data should be written to the target register. It selects the ALU result for instructions that are arithmetic or logical, such as OR, ADD, ORI, and ADDI. For load instructions LW and LDW, it chooses the output of the memory data. For CALL instructions, it determines the PC+1 value to store as the return address. The selected data is written to the register specified by the destination register multiplexer.

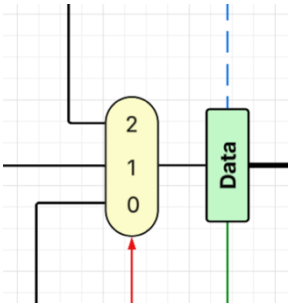


Figure 14: Write-Back Data Multiplexer Block

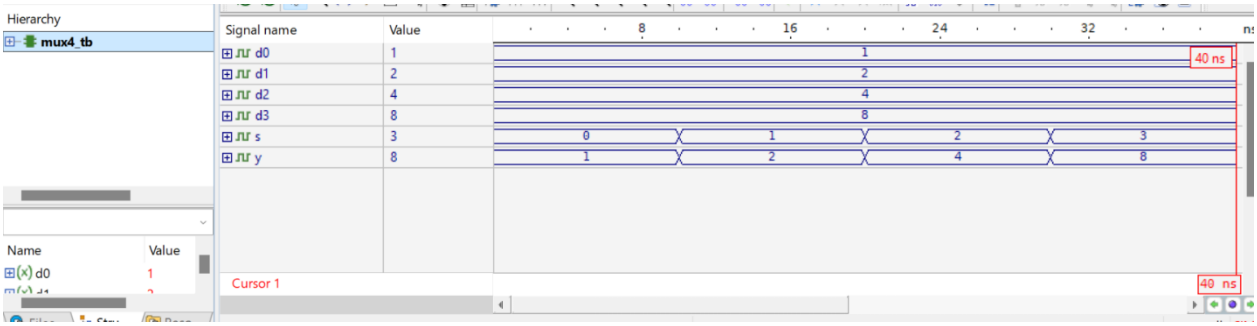


Figure 15: Testbench for Mux4-2 in Datapath Result

Control Unit:

The control unit generates the required control signals for Datapath operation by decoding the 6-bit opcode. Depending on the kind of instruction, it identifies the **ALUOp** signal and enables **RegWr** to control register file updates. **RegDst** is used in destination register selection to select the Rd field, Rd+1 during the second LDW cycle, or R14 for CALL return addresses. **ReadRd** controls whether the second source comes from the Rt for OR, ADD, SUB, and CMP instructions,

Figure 16: Control Unit Block

Rd for SW and SWD instructions, or Rd+1 for SWD second cycle instruction. While **ALUSrc** chooses between the Rt register data, immediate value, or immediate+1 value. The **ExtOp** signal controls whether the 14-bit immediate is extended to zero (for logical operations) or to sign (for arithmetic and memory operations). **MemRd/MemWr** signals are used to control memory operations for load and store instructions. **WBdata** controls the selection of write-back data, selecting either PC+1 for CALL instructions, memory data for load operations (LW, LDW), or ALU results for logical and arithmetic operations (like OR, ORI, ADD, ADDI, and CMP). **PCSrc** selects between PC+1, PC+immediate for branch targets, or jump register values by evaluating the zero, negative, and positive flags from ALU results, as well as the opcode. The **ldw_sdw** signal enables double-word memory operations (64-bit accesses) for LDW and SDW instructions. When active (ldw_sdw = 1), the memory module performs two sequential 32-bit accesses controlled by second_cycle, either reading/writing two consecutive memory locations. This ensures correct handling of 64-bit data, where the first cycle accesses the base address and the second cycle accesses the next word (address + 1). If ldw_sdw = 0, only single-word (32-bit) operations (LW/SW) are executed. The **Exception_Flag** signal is raised when the control unit detects illegal instruction behavior, such as misaligned LDW/SDW (odd-numbered register access). Lastly, **DisablePC** manages the process stall mechanism, which stops fresh instructions from being fetched during the second LDW/SDW operation cycle to finish the double-word transfers.

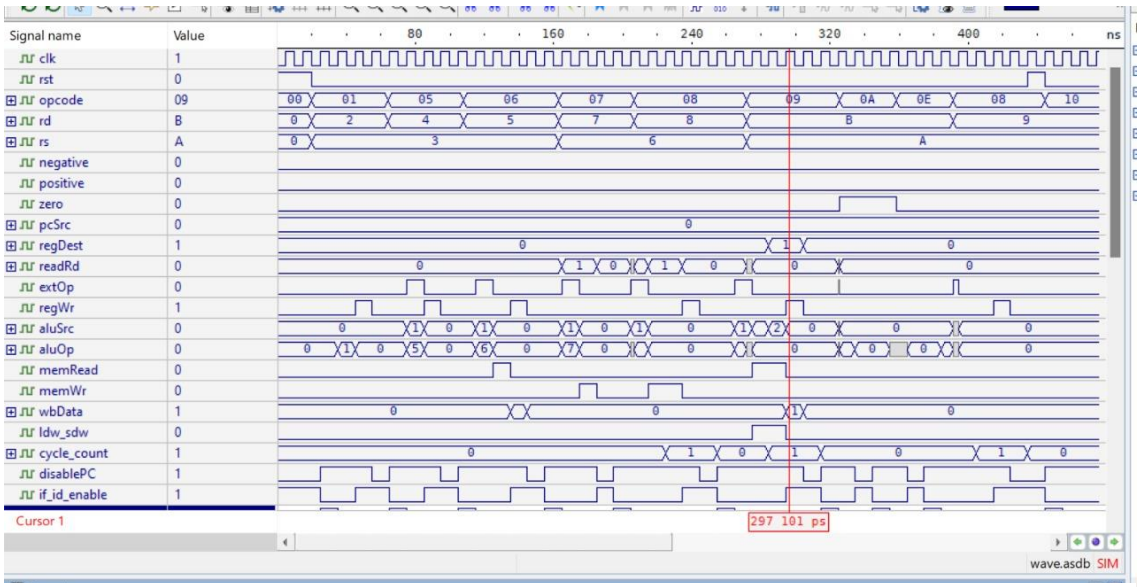


Figure17 : Control Unit Testbench Result

Datapath:

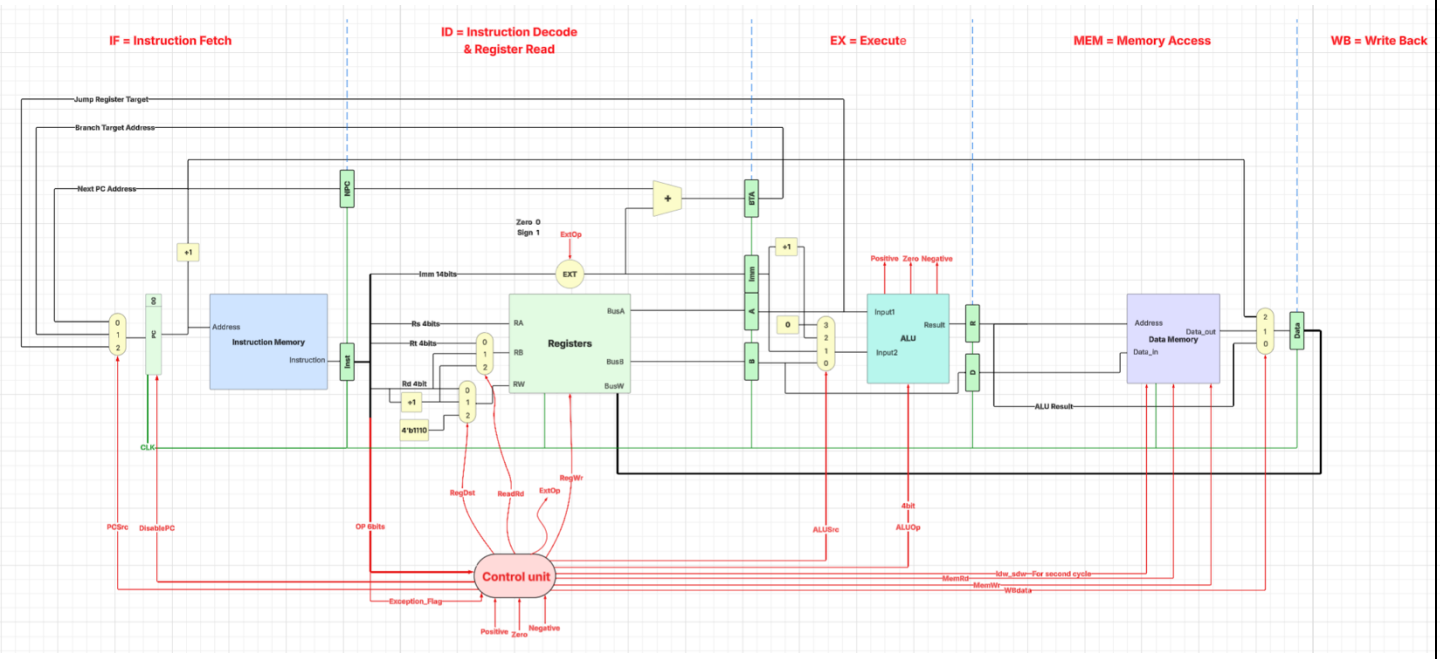


Figure 18: Datapath for Multi-Cycle

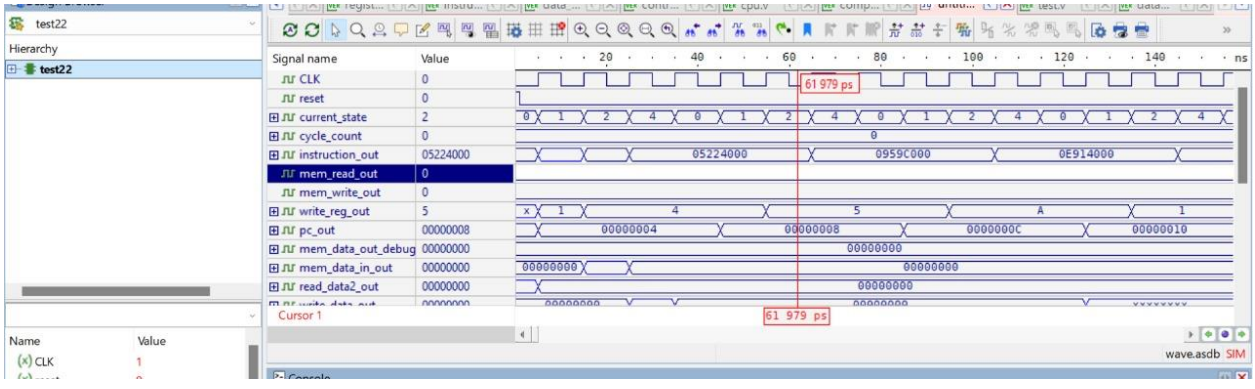


Figure 19: Testbench for Datapath

We test every component and verify it is working, but the datapath has an issue that the alu result doesn’t print right, but it writes to the Register correctly.

RTL:

OR Rd, Rs, Rt:

Fetch Instruction: Instruction ← MEM [PC]

Fetch Operands: data1 ← Reg (Rs) , data2 ← Reg (Rt)

Execute Operation: ALU_Result ← data1 | data2

Write ALU Result: Reg (Rd) ← ALU_Result

Next PC Address: PC ← PC + 1

ADD Rd, Rs, Rt:

Fetch Instruction: Instruction ← MEM [PC]

Fetch Operands: data1 ← Reg (Rs) , data2 ← Reg (Rt)

Execute Operation: ALU_Result ← data1 + data2

Write ALU Result: Reg (Rd) ← ALU_Result

Next PC Address: PC ← PC + 1

SUB Rd, Rs, Rt:

Fetch Instruction: Instruction ← MEM [PC]

Fetch Operands: data1 ← Reg (Rs) , data2 ← Reg (Rt)

Execute Operation: ALU_Result ← data1 - data2

Write ALU Result: Reg (Rd) ← ALU_Result

Next PC Address: PC ← PC + 1

CMP Rd, Rs, Rt:

Fetch Instruction: Instruction ← MEM [PC]

Fetch Operands: data1 ← Reg (Rs) , data2 ← Reg (Rt)

Execute Operation: ALU_Result ← if data1 == data2 → Reg(Rd) ← 0

else if data1 < data2 → Reg(Rd) ← -1

else → Reg(Rd) ← 1

Write ALU Result: Reg (Rd) ← ALU_Result

Next PC Address: PC ← PC + 1

ORI Rd, Rs, Imm:

Fetch Instruction: Instruction ← MEM [PC]

Fetch Operands: data ← Reg (Rs) , Immediate ← Zero_Extend (Imm)

Execute Operation: ALU_Result ← data | Immediate

Write ALU Result: Reg (Rd) ← ALU_Result

Next PC Address: PC ← PC + 1

ADDI Rd, Rs, Imm:

Fetch Instruction: Instruction ← MEM [PC]

Fetch Operands: data ← Reg (Rs) , Immediate ← Sign_Extend (Imm)

Execute Operation: ALU_Result ← data + Immediate

Write ALU Result: Reg (Rd) ← ALU_Result

Next PC Address: PC ← PC + 1

LW Rd , Imm(Rs):

Fetch Instruction: Instruction ← MEM [PC]

Fetch Register: Base← Reg (Rs)

Execute Operation: ALU_Result ← Base + Sign-Extend (Imm)

Memory Access: data← MEM[ALU_Result]

Write Register Rd: Reg (Rd) ← data

Next PC Address: PC ← PC + 1

SW Rd , Imm(Rs):

Fetch Instruction: Instruction ← MEM [PC]

Fetch Register: Base← Reg (Rs), data← Reg(Rd)

Execute Operation: ALU_Result ← Base + Sign-Extend (Imm)

Write Register Rd: MEM[ALU_Result] ← data

Next PC Address: PC ← PC + 1

LDW Rd, Imm (Rs):

Cycle one:

Fetch Instruction: Instruction ← MEM[PC]

Fetch Register: Base ← Reg(Rs)

Execute Operation: ALU_Result ← Base + Sign-Extend(Imm)

Memory Access: data ← MEM[ALU_Result]

Write Register: Reg(Rd) ← data

Next PC Address: PC (no increment - hold current PC)

Cycle two:

Fetch Instruction: No new fetch (hold current instruction)

Fetch Register: Base ← Reg(Rs)

Execute Operation: ALU_Result ← Base + Sign-Extend(Imm) + 1

Memory Access: data ← MEM[ALU_Result]

Write Register: Reg(Rd+1) ← data

Next PC Address: PC ← PC + 1

SDW Rd , Imm(Rs):

Cycle one:

Fetch Instruction: Instruction ← MEM[PC]

Fetch Register: Base ← Reg(Rs), data ← Reg(Rd)

Execute Operation: ALU_Result ← Base + Sign-Extend(Imm)

Memory Write: MEM[ALU_Result] ← data

Next PC Address: PC (no increment - hold current PC)

Cycle two:

Fetch Instruction: No new fetch (hold current instruction)

Fetch Register: Base ← Reg(Rs), data ← Reg(Rd+1)

Execute Operation: ALU_Result ← Base + Sign-Extend(Imm) + 1

Memory Write: MEM[ALU_Result] ← data

Next PC Address: PC ← PC + 1

BZ Rs, Label:

Fetch Instruction: Instruction ← MEM[PC]

Fetch Register: data ← Reg(Rs)

Execute Operation: Imm ← Sign_Extend (Imm)

Write Register Rd: Zero ← (data == 0)

Next PC Address: if Zero: PC ← PC + Imm

else: PC ← PC + 1

BGZ Rs, Label:

Fetch Instruction: Instruction ← MEM[PC]

Fetch Register: data ← Reg(Rs)

Execute Operation: Imm ← Sign_Extend (Imm)

Write Register Rd: Positive ← (data > 0)

Next PC Address: if Positive: PC ← PC + Imm

else: PC ← PC + 1

BLZ Rs, Label:

Fetch Instruction: Instruction ← MEM[PC]

Fetch Register: data ← Reg(Rs)

Execute Operation: Imm ← Sign_Extend (Imm)

Write Register Rd: Negative ← (data < 0)

Next PC Address: if Negative : PC ← PC + imm

else : PC ← PC + 1

JR Rs:

Fetch Instruction: Instruction: ← MEM [PC]

Calculate Target Address: target ← reg(R(s))

Jump: PC ← target

J Label:

Fetch Instruction: Instruction: ← MEM [PC]

Calculate Target Address: target ← sign_extend(Imm)

Jump: PC ← PC + target

CLL Label:

Fetch Instruction: Instruction: ← MEM [PC]

Calculate Target Address: target ← sign_extend(Imm)

Save Return Address: Reg(R14) ← PC + 1

Jump: PC ← PC + target

Control Unit Truth Table:

*We couldn't take one screenshot because the table is too big

opcode						cycle	negative	positive	zero	RegDst		ReadRd		RegWr	ExtOp	ALUSrc	
bit 6	bit 5	bit 4	bit 3	bit 2	bit 1					bit 2	bit 1	bit 2	bit 1	bit 1	bit 1	bit 2	bit 1
0	0	0	0	0	0	X	X	X	X	0	0	0	0	1	X	0	0
0	0	0	0	0	1	X	X	X	X	0	0	0	0	1	X	0	0
0	0	0	0	1	0	X	X	X	X	0	0	0	0	1	X	0	0
0	0	0	0	1	1	X	X	X	X	0	0	0	0	1	X	0	0
0	0	0	1	0	0	X	X	X	X	0	0	X	X	1	0	0	1
0	0	0	1	0	1	X	X	X	X	0	0	X	X	1	1	0	1
0	0	0	1	1	0	X	X	X	X	0	0	X	X	1	1	0	1
0	0	0	1	1	1	X	X	X	X	X	X	0	1	0	1	0	1
0	0	1	0	0	0	0	X	X	X	0	0	X	X	1	1	0	1
0	0	1	0	0	0	1	X	X	X	0	1	X	X	1	1	1	0
0	0	1	0	0	1	0	X	X	X	X	X	0	1	0	1	0	1
0	0	1	0	0	1	1	X	X	X	X	X	1	0	0	1	1	0
0	0	1	0	1	0	X	X	X	X	X	X	X	X	0	1	X	X
0	0	1	0	1	1	X	X	X	X	X	X	X	X	0	1	X	X
0	0	1	1	0	0	X	X	X	X	X	X	X	X	0	1	X	X
0	0	1	1	0	1	X	X	X	X	X	X	X	X	0	X	X	X
0	0	1	1	1	0	X	X	X	X	X	X	X	X	0	1	X	X
0	0	1	1	1	1	X	X	X	X	1	0	X	X	1	1	1	1

ALUOp				MemRd	MemWr	WBdata		DisablePC	ldw_sdw
bit 4	bit 3	bit 2	bit 1	bit 1	bit 1	bit 2	bit 1	bit 1	bit 1
0	0	0	0	X	X	0	0	0	0
0	0	0	1	X	X	0	0	0	0
0	0	1	0	X	X	0	0	0	0
0	0	1	1	X	X	0	0	0	0
0	1	0	0	X	X	0	0	0	0
0	1	0	1	X	X	0	0	0	0
0	1	1	0	1	0	0	1	0	0
0	1	1	1	0	1	X	X	0	0
1	0	0	0	1	0	0	1	1	1
1	0	0	0	1	0	0	1	0	0
1	0	0	1	0	1	X	X	1	1
1	0	0	1	0	1	X	X	0	0
1	0	1	0	X	X	X	X	0	0
1	0	1	1	X	X	X	X	0	0
1	1	0	0	X	X	X	X	0	0
1	1	0	1	X	X	X	X	0	0
1	1	1	0	X	X	X	X	0	0
1	1	1	1	X	X	1	0	0	0

Figure 20: Control Unit Truth Table 1

opcode						cycle	negative	positive	zero	PCSrc	
bit 6	bit 5	bit 4	bit 3	bit 2	bit 1					bit 2	bit 1
0	0	0	0	0	0	X	X	X	X	0	0
0	0	0	0	0	1	X	X	X	X	0	0
0	0	0	0	1	0	X	X	X	X	0	0
0	0	0	0	1	1	X	X	X	X	0	0
0	0	0	1	0	0	X	X	X	X	0	0
0	0	0	1	0	1	X	X	X	X	0	0
0	0	0	1	1	0	X	X	X	X	0	0
0	0	0	1	1	1	X	X	X	X	0	0
0	0	1	0	0	0	0	X	X	X	X	X
0	0	1	0	0	0	1	X	X	X	0	0
0	0	1	0	0	1	0	X	X	X	X	X
0	0	1	0	0	1	1	X	X	X	0	0
0	0	1	0	1	0	X	X	X	0	0	0
0	0	1	0	1	0	X	0	0	1	0	1
0	0	1	0	1	1	X	X	0	X	0	0
0	0	1	0	1	1	X	0	1	0	0	1
0	0	1	1	0	0	X	0	X	X	0	0
0	0	1	1	0	0	X	1	0	0	0	1
0	0	1	1	0	1	X	X	X	X	1	0
0	0	1	1	1	0	X	X	X	X	0	1
0	0	1	1	1	1	X	X	X	X	0	1

Figure 22: Control Unit Truth Table 2

opcode						cycle	Odd Rd	Odd Rs	negative	positive	zero	Exception
bit 6	bit 5	bit 4	bit 3	bit 2	bit 1							bit 1
0	0	0	0	0	0	X	X	X	X	X	X	0
0	0	0	0	0	1	X	X	X	X	X	X	0
0	0	0	0	1	0	X	X	X	X	X	X	0
0	0	0	0	1	1	X	X	X	X	X	X	0
0	0	0	1	0	0	X	X	X	X	X	X	0
0	0	0	1	0	1	X	X	X	X	X	X	0
0	0	0	1	1	0	X	X	X	X	X	X	0
0	0	0	1	1	1	X	X	X	X	X	X	0
0	0	1	0	0	0	0	0	X	X	X	X	0
0	0	1	0	0	0	0	1	X	X	X	X	1
0	0	1	0	0	0	1	X	X	X	X	X	0
0	0	1	0	0	1	0	X	0	X	X	X	0
0	0	1	0	0	1	0	X	1	X	X	X	1
0	0	1	0	0	1	1	X	X	X	X	X	0
0	0	1	0	1	0	X	X	X	X	X	X	0
0	0	1	0	1	1	X	X	X	X	X	X	0
0	0	1	1	0	0	X	X	X	X	X	X	0
0	0	1	1	0	1	X	X	X	X	X	X	0
0	0	1	1	1	0	X	X	X	X	X	X	0
0	0	1	1	1	1	X	X	X	X	X	X	0

Figure 21: Control Unit Truth Table 3

*Each table has the important input for a specific flag, and the rest of the input will be Don't Care.

K-map and Boolean Equations:

*Bits 5 and 6 in the opcode are always 0:

RegDst bit 2:

F = (O3 & O2 & O1)

Figure 23: K-map RegDst bit 2

RegDst bit 1:

F = (O4 & ~O3 & C)

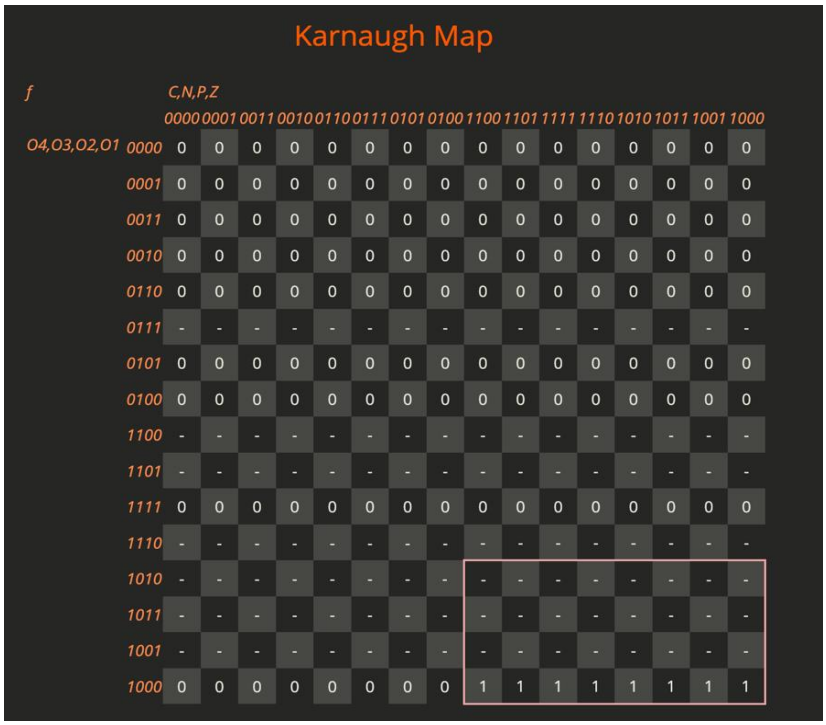


Figure 24: K-map RegDst bit 1

ReadRd bit 2:

F = (O4 & C)

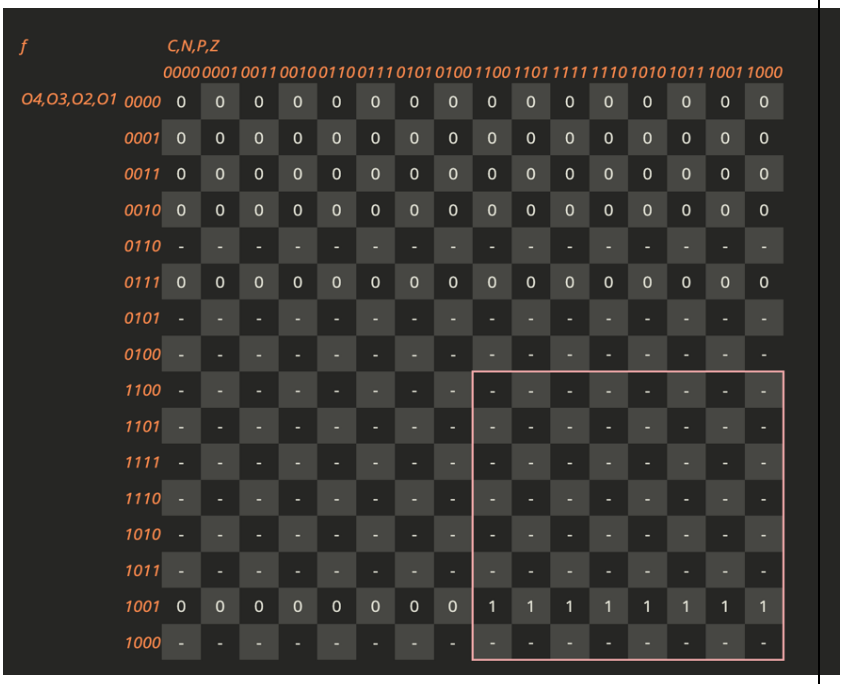


Figure 25: K-map ReadRd bit 2

ReadRd bit 1:

F = (O3) | (O4 & ~C)

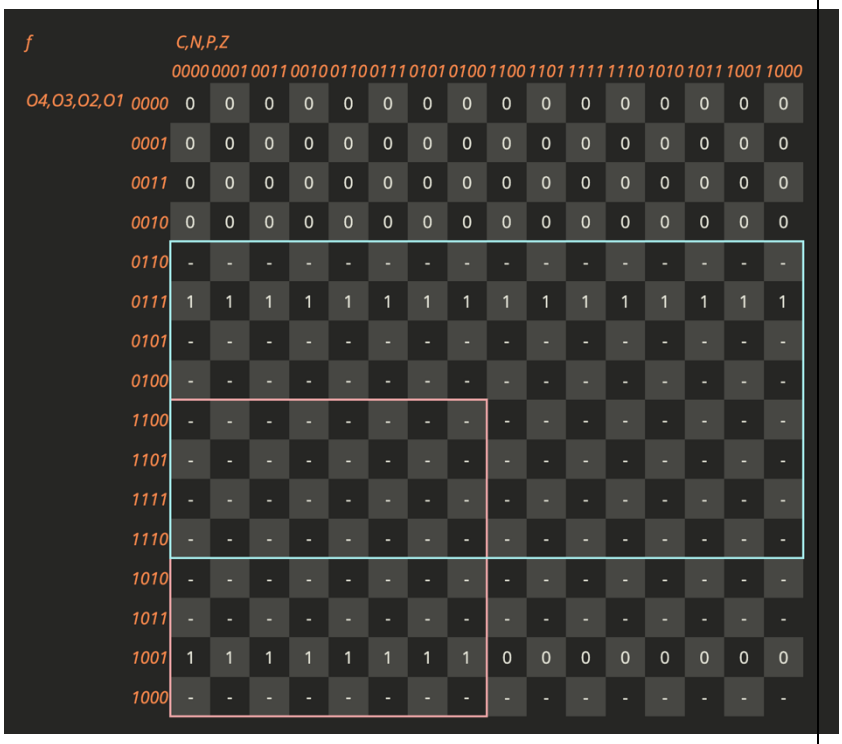


Figure 26: K-map ReadRd bit 1

RegWr:

F = (~O4 & ~O3) | (~O4 & ~O2) | (~O4 & ~O1) | (~O3 & ~O2 & ~O1) | (O4 & O3 & O2 & O1)

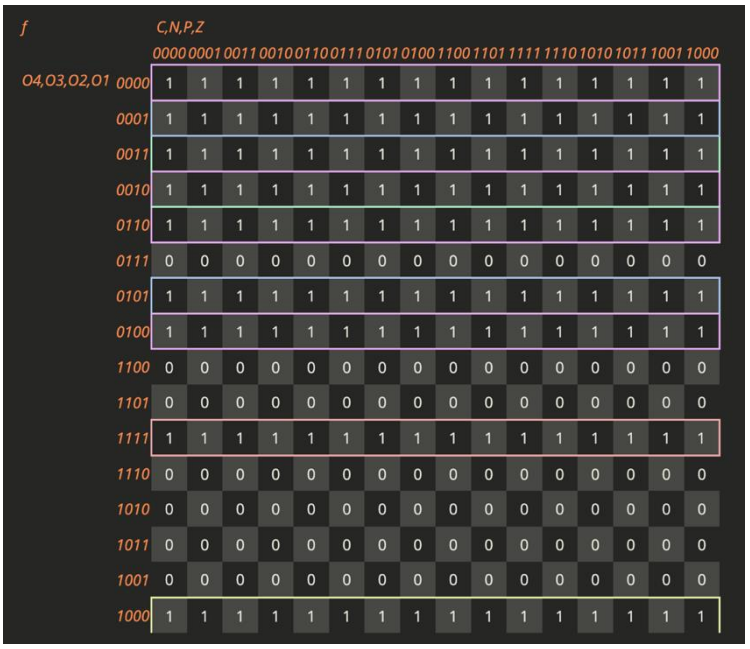


Figure 27: K-map RegWr

ExtOp:

F = (O1) | (O2) | (O4)

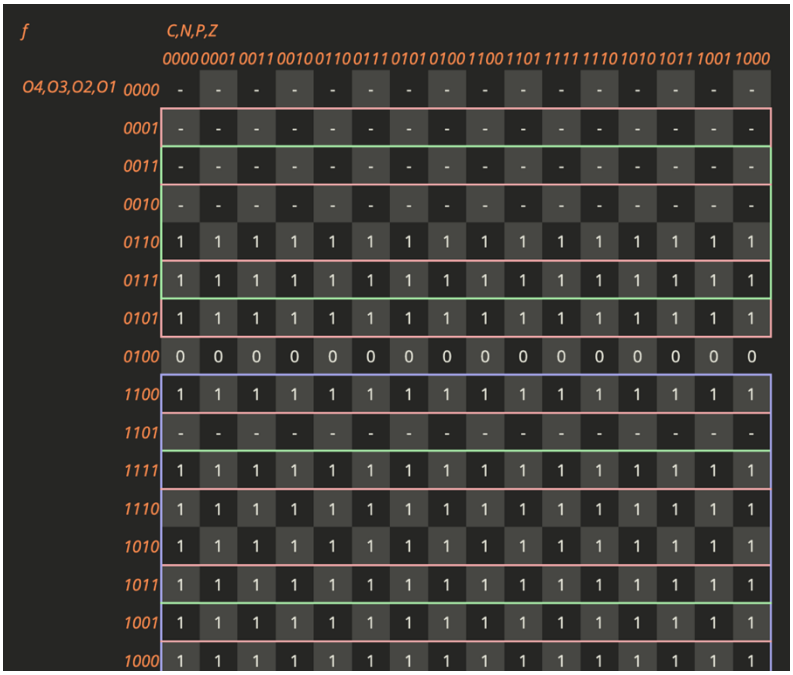


Figure 28: K-map ExtOp

ALUSrc bit 2:

F = (O4 & C) | (O4 & O2)

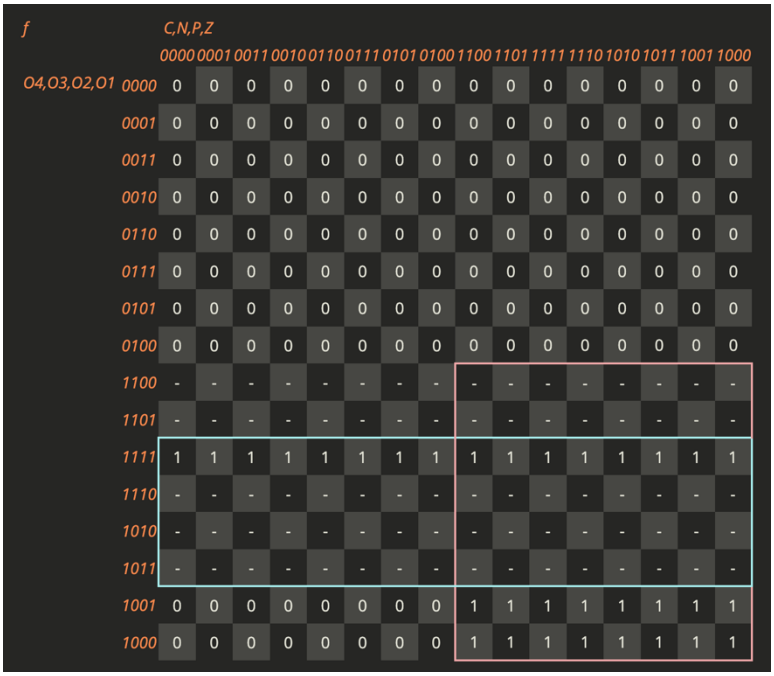


Figure 29: K-map ALUSrc bit 2

ALUSrc bit 1:

F = (O3) | (O4 & ~C)

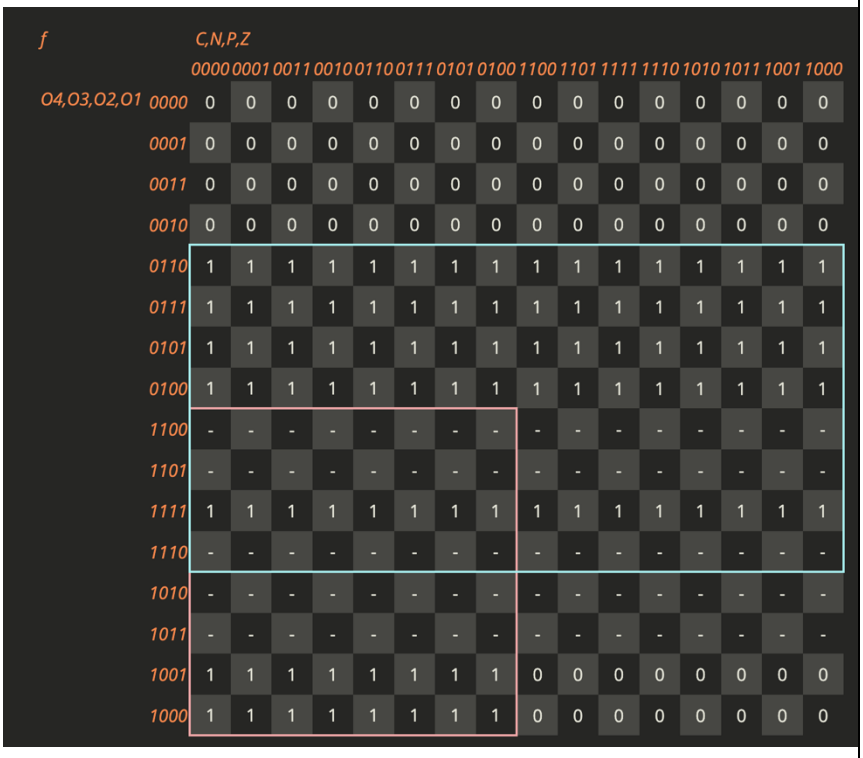


Figure 30: K-map ALUSrc bit 1

ALUOp:

We can make it equal the last 4 bits from the opcode.

MemRd:

F = (~O1)

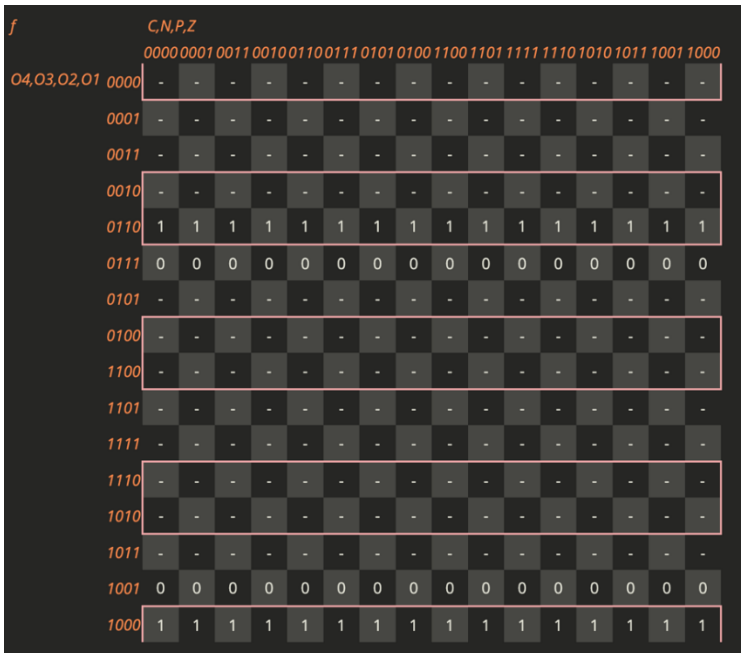


Figure 31: K-map MemRd

MemWr:

F = (O1)

Figure 32: K-map MemWr

WBdata bit 2:

F = (O3 & O2 & O1)

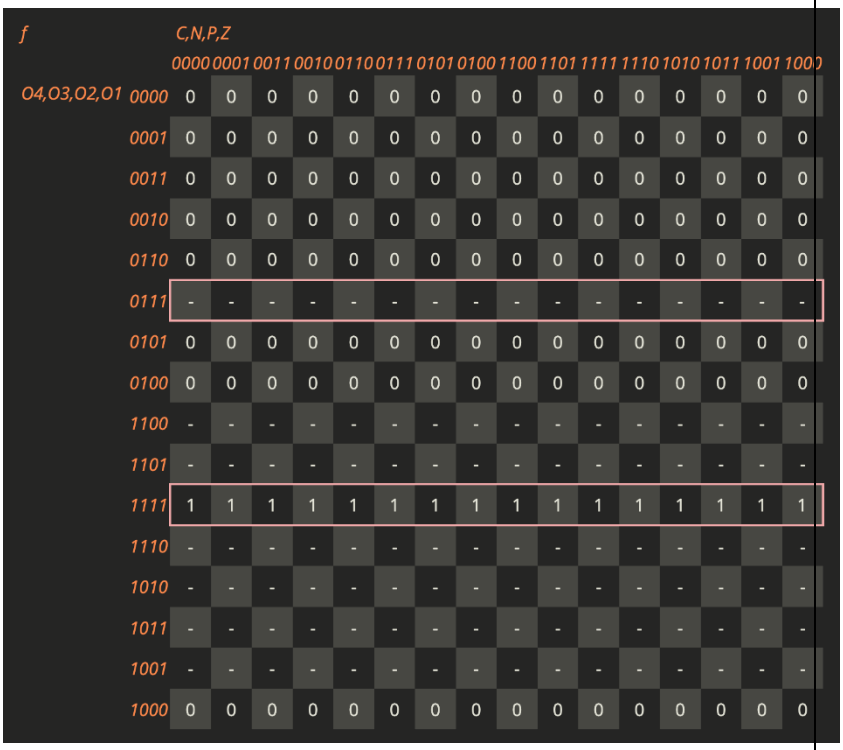


Figure 33: K-map WBdata bit 2

WBdata bit 1:

F = (~O4 & O3 & O2) | (O4 & ~O3)

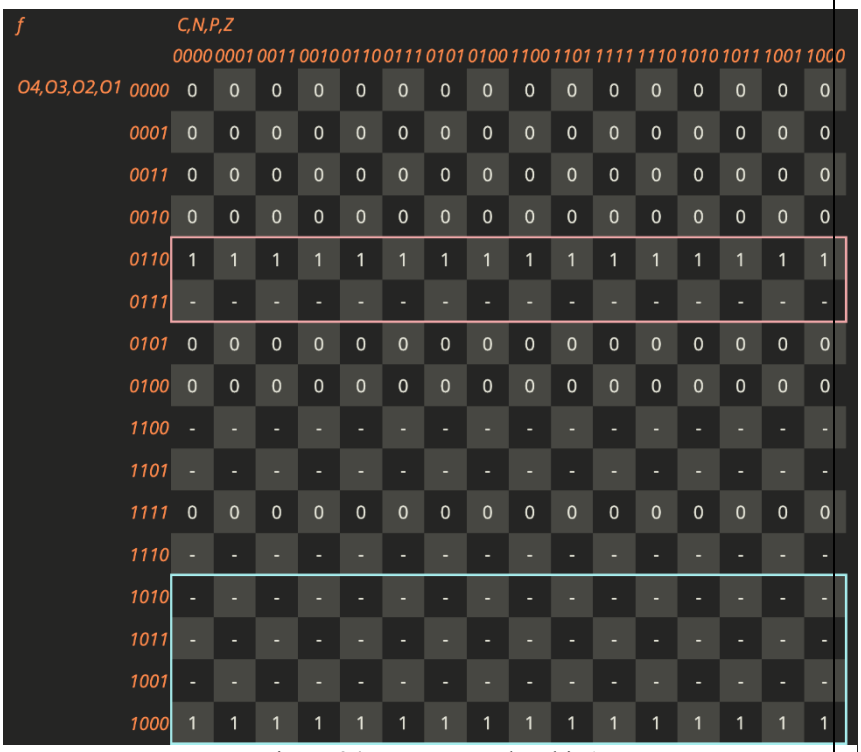
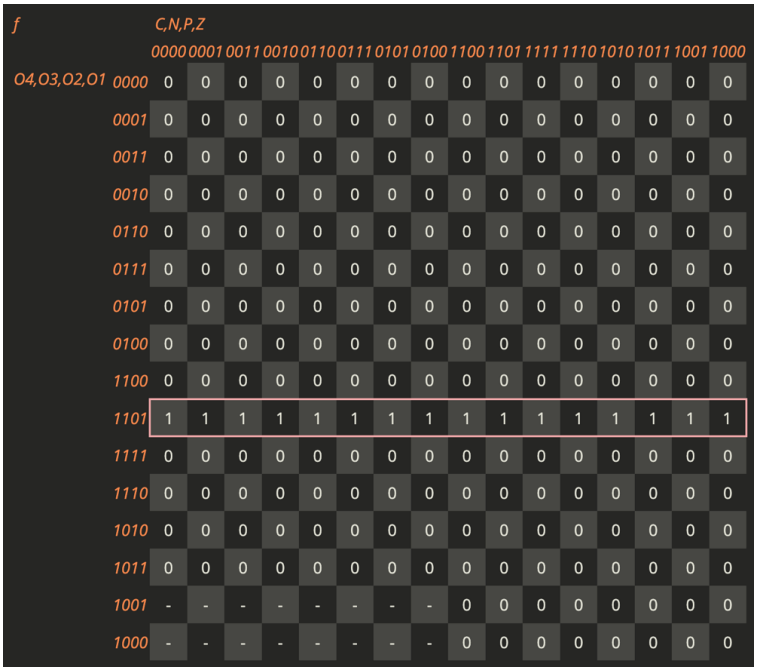


Figure 34: K-map WBdata bit 1

PCSrc bit 2:

F = (O4 & O3 & ~O2 & O1)

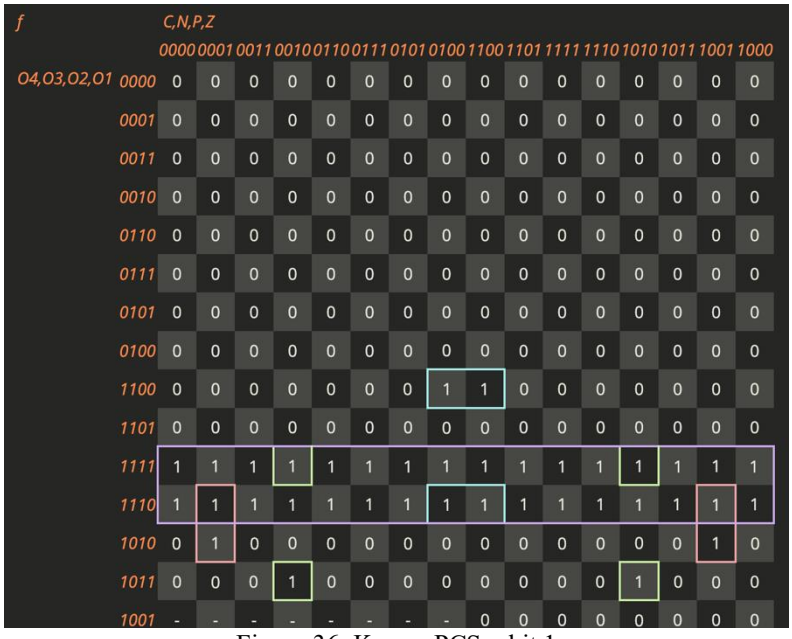


f	C,N,P,Z	0000	0001	0011	0010	0110	0111	0101	0100	1100	1101	1111	1110	1010	1011	1001	1000
O4,O3,O2,O1	0000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0001	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0011	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0010	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0110	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0111	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0101	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0100	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1100	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1101	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	1111	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1110	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1010	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1011	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1001	-	-	-	-	-	-	-	-	0	0	0	0	0	0	0	0
	1000	-	-	-	-	-	-	-	-	0	0	0	0	0	0	0	0

Figure 35: K-map PCSrc bit 2

PCSrc bit 1:

F = (O4 & O2 & ~O1 & ~N & ~P & Z) | (O4 & O2 & O1 & ~N & P & ~Z) | (O4 & O3 & ~O1 & N & ~P & ~Z) | (O4 & O3 & O2)

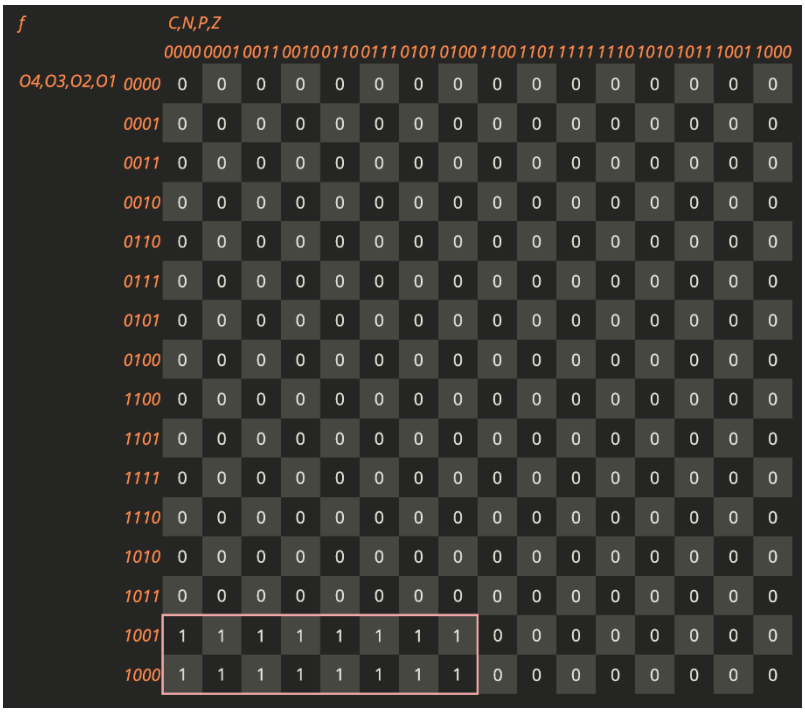


f	C,N,P,Z	0000	0001	0011	0010	0110	0111	0101	0100	1100	1101	1111	1110	1010	1011	1001	1000
O4,O3,O2,O1	0000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0001	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0011	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0010	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0110	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0111	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0101	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0100	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1100	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0
	1101	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1111	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	1110	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	1010	0	1	0	0	0	0	0	0	0	0	0	0	0	1	1	0
	1011	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0
	1001	-	-	-	-	-	-	-	-	0	0	0	0	0	0	0	0

Figure 36: K-map PCSrc bit 1

DisablePC:

F = (O4 & ~O3 & ~O2 & ~C)

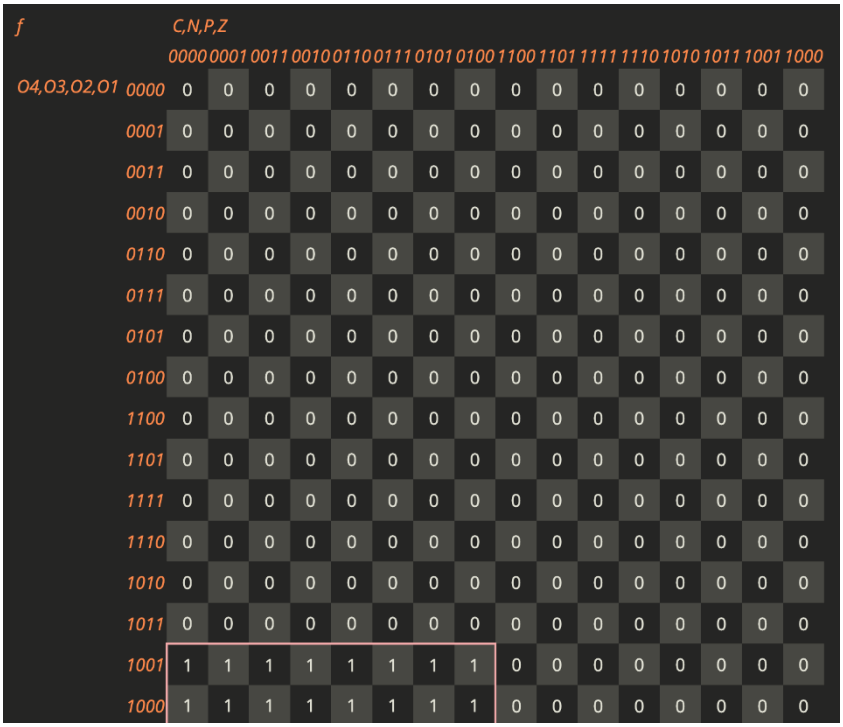


f	C,N,P,Z	0000	0001	0011	0010	0110	0111	0101	0100	1100	1101	1111	1110	1010	1011	1001	1000
O4,O3,O2,O1	0000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0001	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0011	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0010	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0110	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0111	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0101	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0100	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1100	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1101	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1111	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1110	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1010	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1011	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1001	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
	1000	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0

Figure 37: K-map DisablePC

ldw_sdw:

F = (O4 & ~O3 & ~O2 & ~C)



f	C,N,P,Z	0000	0001	0011	0010	0110	0111	0101	0100	1100	1101	1111	1110	1010	1011	1001	1000
O4,O3,O2,O1	0000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0001	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0011	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0010	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0110	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0111	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0101	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0100	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1100	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1101	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1111	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1110	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1010	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1011	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1001	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
	1000	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0

Figure 38: K-map ldw_sdw

Exception:

F = (O4 & ~O3 & ~O2 & ~O1 & ~C & ORd) | (O4 & ~O3 & ~O2 & O1 & ~C & ORs)

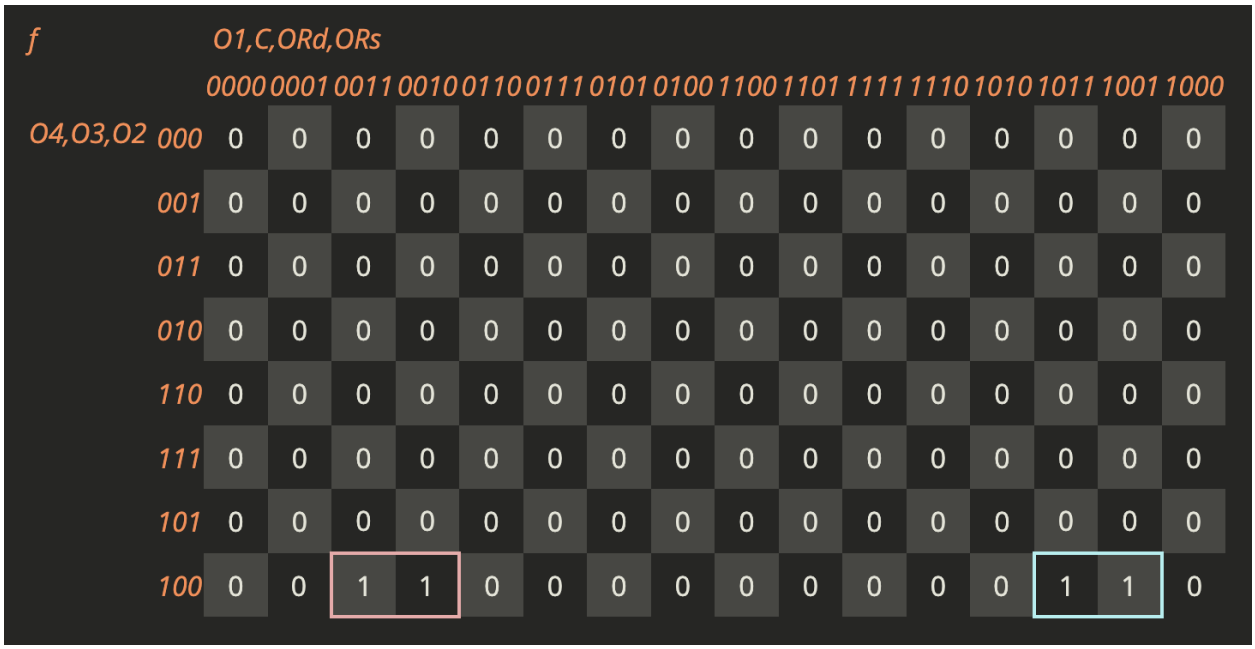


Figure 39: K-map Exception

State Diagram:

This state diagram represents a 6-state finite state machine that controls a multi-cycle RISC processor implementation. The processor cycles through FETCH, DECODE, EXECUTE, MEMORY, and WRITEBACK states for normal instruction execution, with each state enabling specific pipeline registers and control signals. The DECODE state performs exception detection for invalid opcodes and improper register usage in LDW/SDW instructions, transitioning to an EXCEPTION state when violations occur. Multi-cycle operations like LDW and SDW require two memory cycles, implemented through a self-loop in the MEMORY state with a cycle counter tracking progress. Different instruction types follow distinct paths through the state machine: ALU operations bypass the MEMORY state, branches and jumps return directly to FETCH after execution, while memory operations proceed through all states. The design ensures proper pipeline control by selectively enabling stage registers (if_id_enable, id_ex_enable, ex_mem_enable, mem_wb_enable) and managing PC updates through the disablePC signal, creating a robust control mechanism that handles the processor's instruction set architecture requirements, including exception handling and multi-cycle memory operations.

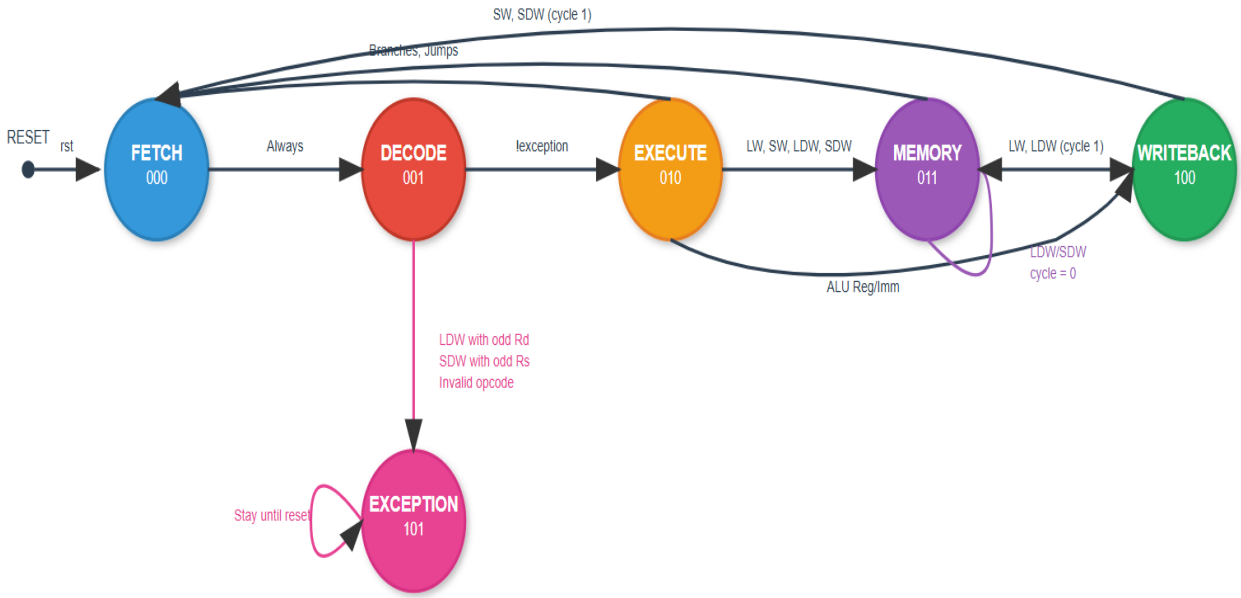


Figure 40: State Diagram

- **States:**
  - FETCH (000): Instruction fetch
  - DECODE (001): Instruction decode
  - EXECUTE (010): ALU operation
  - MEMORY (011): Memory access
  - WRITEBACK (100): Register write
  - EXCEPTION (101): Exception handling

- **Instruction Types:**
  - ALU Reg: OR, ADD, SUB, CMP
  - ALU Imm: ORI, ADDIâ€¢
  - Memory: LW, SW, LDW, SDW
  - Branch: BZ, BGZ, BLZ
  - Jump: JR, J, CALL

- **Multi-cycle Operations:**
  - LDW/SDW requires 2 memory cycles
  - The cycle counter tracks progress
  - Exception on odd register numbers

- **Control Signals by State:**
- **FETCH:**
  - if_id_enable = 1
  - PC increment enabled

- **DECODE:**
  - id_ex_enable = 1
  - disablePC = 1
  - Exception Detection

- **EXECUTE:**
  - ex_mem_enable = 1
  - ALU operation setup
  - Address calculation

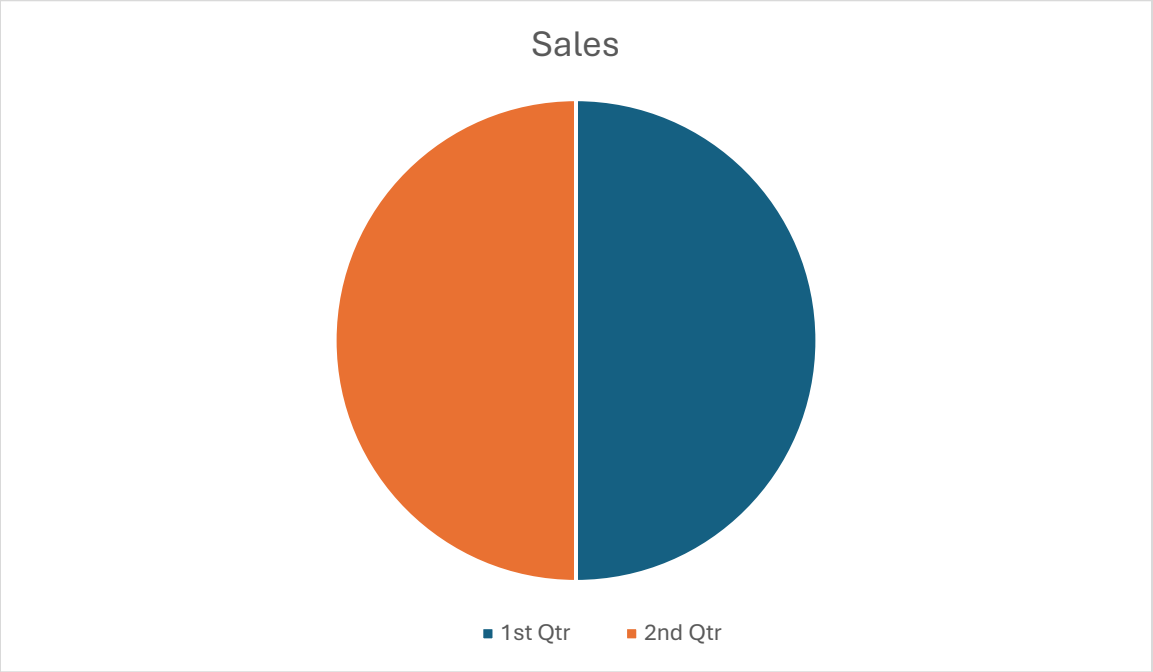
- **MEMORY:**
  - mem_wb_enable = 1
  - memRead/memWr control
  - Multi-cycle handling

- **WRITEBACK:**
  - regWr = 1
  - Register destination control
  - Branch/Jump PC control

Conclusion:

In this project, a 32-bit multi-cycle RISC processor is designed and implemented using Verilog HDL. A comprehensive set of 16 operations, including arithmetic, logic, memory access, and control flow, is supported by the processor. The processor divides instruction execution into five separate stages (fetch, decode, execute, memory access, and write-back) to optimize hardware resource usage through the adoption of a multi-cycle architecture. For effective data routing, the Datapath combines essential parts such as a program counter, instruction memory, a 16-register file, an ALU, data memory, and a multiplexer system. The control unit provides 13 control signals. Multi-cycle memory operations (LDW/SDW) are handled by special mechanisms such as DisablePC, which also detects illegal instructions. Additionally, the processor includes flag generation (zero, negative, positive) to enable precise conditional branching.

Teamwork:



We work on everything together in the code and report, and in the Datapath Design.