ASSIGNMENT 8

1 Problem Statement

Implement a single-cycle 32-bit MIPS processor supporting a subset of instructions including:

- Memory reference instructions: `lw` (load word), `sw` (store word)
- Arithmetic-logical instructions: `add`, `sub`, `and`, `or`, `slt` (set-less-than)
- Control transfer instructions: `beq` (branch equal), `j` (jump)
- Instruction for supporting subroutine: `jal` (jump and link)
- Additional instruction implemented: `addi` (add immediate), `jr` (jump register)

The processor must execute one instruction per clock cycle (single-cycle implementation). All components of the datapath must be implemented as separate Verilog modules to maintain a modular design approach.

2 Instruction Format Summary

MIPS uses three instruction formats:

Format	Bit Fields	Example Instructions
R-type	op[31:26], rs[25:21], rt[20:16], rd[15:11], shamt[10:6], funct[5:0]	add, sub, and, or, slt, jr
I-type	op[31:26], rs[25:21], rt[20:16], immediate[15:0]	lw, sw, beq, addi
J-type	op[31:26], address[25:0]	j, jal

Table 1: MIPS Instruction Formats

2.1 OpCode and Function Field Summary

Instruction	Format	OpCode (hex)	Function (hex)
add	R-type	0x00	0x20
sub	R-type	0x00	0x22
and	R-type	0x00	0x24
or	R-type	0x00	0x25
slt	R-type	0x00	0x2A
jr	R-type	0x00	0x08 (custom)
lw	I-type	0x23	-
sw	I-type	0x2B	-
beq	I-type	0x04	-
addi	I-type	0x08	-
j	J-type	0x02	-
jal	J-type	0x03	-

Table 2: OpCode and Function Field Values

3 Datapath Design

The single-cycle MIPS processor executes each instruction in one clock cycle. The processor is designed with a modular approach, separating the datapath and control logic.

3.1 Overall Architecture

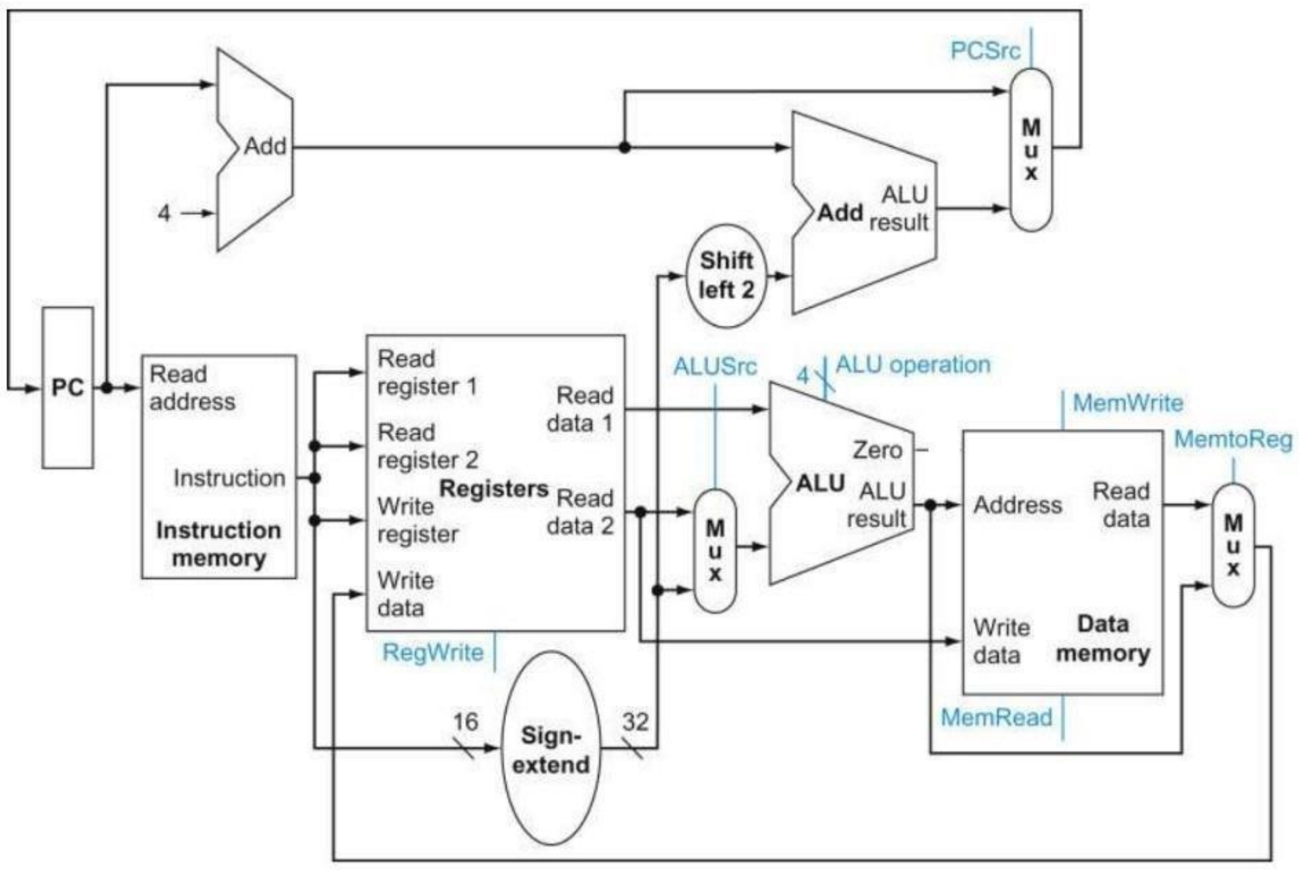


Figure 1: Single-Cycle MIPS Processor Datapath

3.2 Key Components

3.2.1 Program Counter (PC)

The PC module maintains the address of the current instruction. It has an enable input that allows updating the PC on each clock cycle when enabled. The PC is a 32-bit register with synchronous reset functionality.

3.2.2 Instruction Memory

The instruction memory module stores the program to be executed. It is implemented as a read-only memory with 256 32bit locations. In our implementation, we initialize it with a test program that demonstrates all the supported instructions.

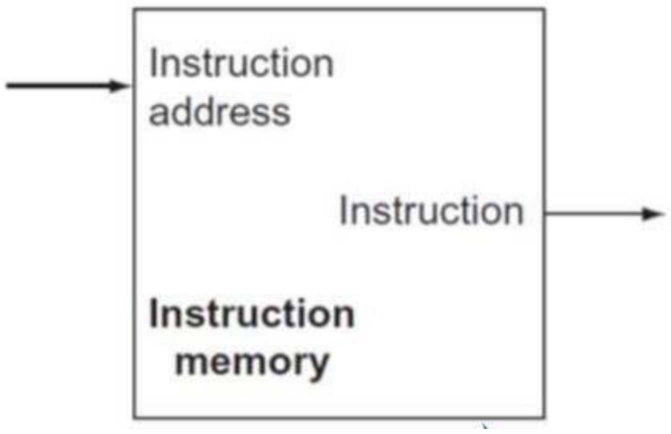


Figure 2: Instruction Memory

3.2.3 Register File

The register file consists of 32 general-purpose registers, each 32 bits wide. It has:

- Two read ports that allow reading two registers simultaneously
- One write port for writing data to a register
- Write enable signal to control when writing occurs
- Register $0 is hardwired to zero

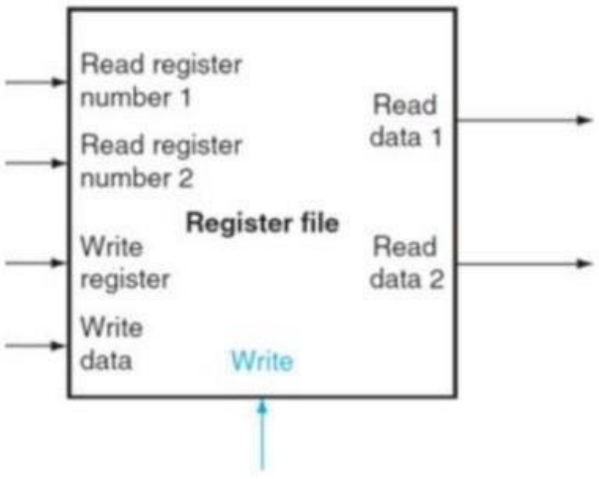


Figure 3: Register File

The implementation uses a decoder and multiplexers to select registers for reading and writing.

3.2.4 ALU (Arithmetic Logic Unit)

The ALU performs operations based on a 4-bit control signal:

- 0000: AND • 0001: OR
- 0010: ADD
- 0110: SUB
- 0111: SLT (set if less than)

The ALU also outputs a zero flag that indicates whether the result is zero (used for branch decisions).

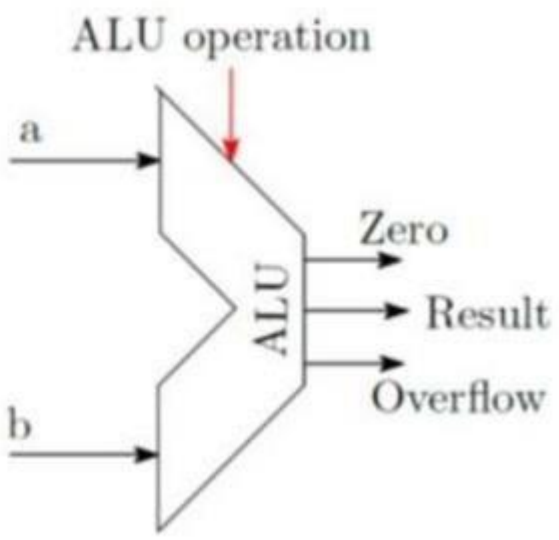


Figure4:ALU

3.2.5 Data Memory

The data memory has 256 32-bit word locations, with separate read and write enable signals. Memory addresses are wordaligned (divided by 4).

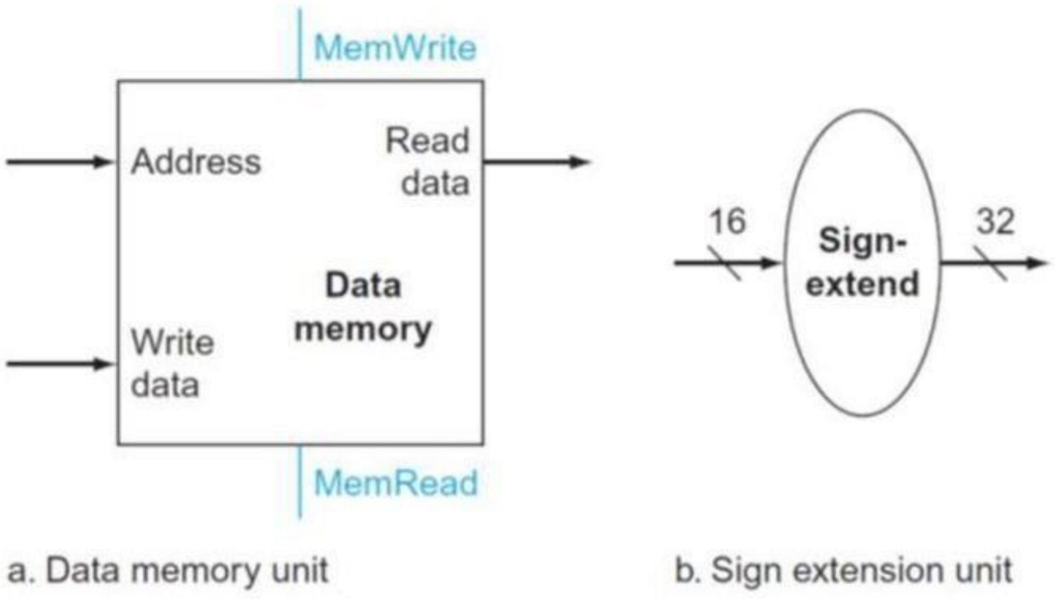


Figure 5: Data Memory

3.2.6 Sign Extender

Extends the 16-bit immediate values to 32 bits by replicating the most significant bit.

3.2.7 Shifters

Two types of shifters are used:

- Left shift by 2 bits (multiply by 4) for branch address calculation
- Left shift by 2 bits for jump address calculation

3.3 Data Path Operations

3.3.1 R-type Instructions (add, sub, and, or, slt)

1. Fetch instruction from instruction memory using PC
2. Read two source registers (rs, rt) from register file
3. Perform ALU operation based on function code
4. Write result to destination register (rd)
5. Increment PC by 4

3.3.2 Load Word (lw)

1. Fetch instruction from instruction memory
2. Read base register (rs) from register file
3. Sign-extend immediate field
4. Calculate memory address using ALU (base + o!set)
5. Read data from memory at the calculated address
6. Write data to destination register (rt)
7. Increment PC by 4

3.3.3 Store Word (sw)

1. Fetch instruction from instruction memory
2. Read base register (rs) and data register (rt) from register file
3. Sign-extend immediate field
4. Calculate memory address using ALU (base + o!set)
5. Write data to memory at the calculated address
6. Increment PC by 4

3.3.4 Branch Equal (beq)

1. Fetch instruction from instruction memory
2. Read two registers (rs, rt) from register file
3. Compare register values using ALU subtraction 4. Sign-extend and shift immediate field (multiply by 4)
5. If registers are equal (ALU zero flag is set):
    - Calculate target address (PC + 4 + shifted immediate)
    - Set PC to target address
6. Else: Increment PC by 4

3.3.5 Jump (j)

1. Fetch instruction from instruction memory
2. Shift jump target left by 2 bits
3. Combine upper 4 bits of (PC + 4) with shifted target
4. Set PC to jump address

3.3.6 Jump and Link (jal)

1. Fetch instruction from instruction memory
2. Calculate return address (PC + 4)
3. Write return address to register $31 (ra)
4. Shift jump target left by 2 bits
5. Combine upper 4 bits of (PC + 4) with shifted target
6. Set PC to jump address

3.3.7 Jump Register (jr)

1. Fetch instruction from instruction memory
2. Read register (rs) from register file
3. Set PC to the value in rs

3.3.8 Add Immediate (addi)

1. Fetch instruction from instruction memory
2. Read source register (rs) from register file
3. Sign-extend immediate field
4. Add register value and sign-extended immediate
5. Write result to destination register (rt)
6. Increment PC by 4

4 Control Unit Design

The control unit is divided into two parts:

1. Main Controller: Generates control signals based on the OpCode
2. ALU Controller: Generates ALU control signals based on the ALUOp from the main controller and the function field for R-type instructions

4.1 Main Controller

The main controller decodes the 6-bit OpCode and generates the following control signals:

Instr	RegDst	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALUOp	Jump	Link	JR
R-type	1	0	0	1	0	0	0	10	0	0	0
lw	0	1	1	1	1	0	0	00	0	0	0
sw	x	1	x	0	0	1	0	00	0	0	0
beq	x	0	x	0	0	0	1	01	0	0	0
j	x	x	x	0	0	0	0	xx	1	0	0
jal	x	x	x	1	0	0	0	xx	1	1	0
addi	0	1	0	1	0	0	0	00	0	0	0
jr	x	x	x	0	0	0	0	xx	0	0	1

Table 3: Control Signal Truth Table (Landscape Orientation)

- **RegDst**: Selects the destination register (rt or rd)
- **ALUSrc**: Selects the second ALU operand (rt or immediate)

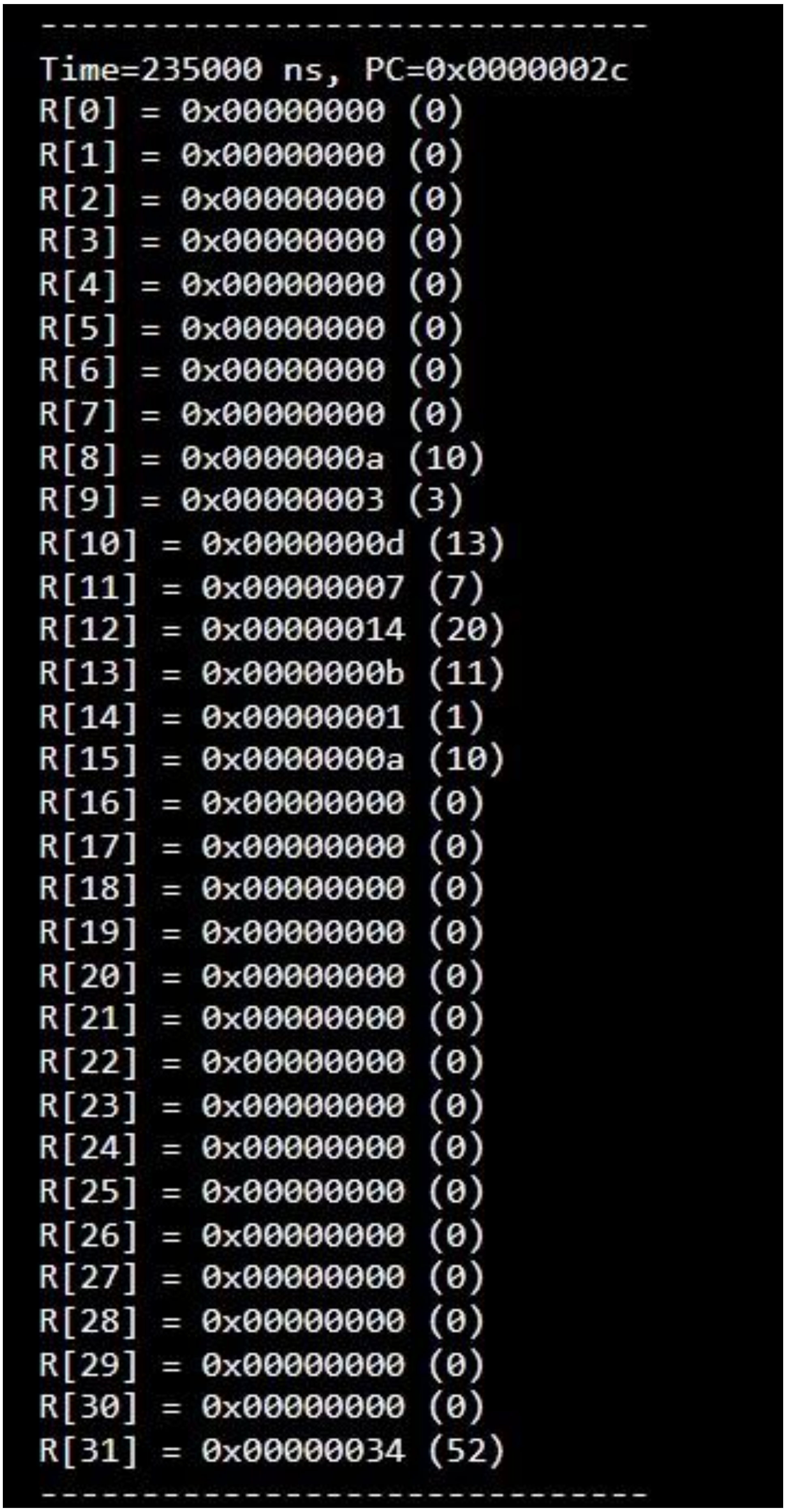
- **MemtoReg**: Selects the data to write to the register file (ALU result or memory data)
- **RegWrite**: Enables writing to the register file
- **MemRead**: Enables reading from data memory
- **MemWrite**: Enables writing to data memory
- **Branch**: Indicates a branch instruction
- **ALUOp**: Determines the ALU operation (00 for add, 01 for subtract, 10 for R-type)
- **Jump**: Indicates a jump instruction
- **Link**: Indicates to save PC+4 in $31 (for jal)
- **JR**: Indicates a jump register instruction

4.2 ALU Controller

The ALU controller takes the 2-bit ALUOp from the main controller and the 6-bit function field (for R-type instructions) to generate the 4-bit ALU control signal:

ALUOp	Function Field	ALU Control
00	x	0010 (ADD)
01	x	0110 (SUB)
10	100000 (add)	0010 (ADD)
10	100010 (sub)	0110 (SUB)
10	100100 (and)	0000 (AND)
10	100101 (or)	0001 (OR)
10	101010 (slt)	0111 (SLT)

Table 4: ALU Control Signal Generation



```
-------------------------------
Time=235000 ns, PC=0x0000002c
R[0] = 0x00000000 (0)
R[1] = 0x00000000 (0)
R[2] = 0x00000000 (0)
R[3] = 0x00000000 (0)
R[4] = 0x00000000 (0)
R[5] = 0x00000000 (0)
R[6] = 0x00000000 (0)
R[7] = 0x00000000 (0)
R[8] = 0x0000000a (10)
R[9] = 0x00000003 (3)
R[10] = 0x0000000d (13)
R[11] = 0x00000007 (7)
R[12] = 0x00000014 (20)
R[13] = 0x0000000b (11)
R[14] = 0x00000001 (1)
R[15] = 0x0000000a (10)
R[16] = 0x00000000 (0)
R[17] = 0x00000000 (0)
R[18] = 0x00000000 (0)
R[19] = 0x00000000 (0)
R[20] = 0x00000000 (0)
R[21] = 0x00000000 (0)
R[22] = 0x00000000 (0)
R[23] = 0x00000000 (0)
R[24] = 0x00000000 (0)
R[25] = 0x00000000 (0)
R[26] = 0x00000000 (0)
R[27] = 0x00000000 (0)
R[28] = 0x00000000 (0)
R[29] = 0x00000000 (0)
R[30] = 0x00000000 (0)
R[31] = 0x00000034 (52)
-------------------------------
```