MIPS32 Processor

Aseem A Choudhary
Roll No: 2210095
Department of Electronics and Communication Engineering
Indian Institute of Technology, Kharagpur

Contents

1 32-bit RISC-like Processor Specification

Registers	32 32-bit general-purpose registers R0..R31, organized as a register bank with two read ports and one write port.
Memory	Memory is byte addressable with a 32-bit memory address. All operations are on 32-bit data, and all loads and stores occur from memory addresses that are multiples of 4.
Program Counter	A 32-bit program counter (PC).
Addressing Modes	Register addressing, Immediate addressing, Base addressing for accessing memory (with any of the registers used as base register), PC relative addressing for branch.

2 Instruction Set

Type	Instruction	Format	Example
4*Arithmetic & Logic	ADD	R1, R2, R3	R1 = R2 + R3
	SUB	R1, R2, R3	R1 = R2 - R3
	XOR	R1, R2, R3	R1 = R2 ^ R3
	SLT	R1, R2, R3	R1 = R2 $\ll$ R3[0]
1*Immediate Addressing	XORI	R1, #1	R1 = R1 $\ll$ 1
2*Load & Store	LW	R1, 10(R2)	R1 = Mem[R2 + 10]
	SW	R1, -2(R3)	Mem[R3 - 2] = R1
3*Branch	BR	#10	PC = PC + 10
	BNE	R1, R2, #-10	PC = PC - 10 if (R1 $\neq$ R2)
	JR	R1	PC = Reg[R1]

3 Register Bank

Register	Code	Register	Code	Register	Code
R0	00000	R11	01011	R22	10110
R1	00001	R12	01100	R23	10111
R2	00010	R13	01101	R24	11000
R3	00011	R14	01110	R25	11001
R4	00100	R15	01111	R26	11010
R5	00101	R16	10000	R27	11011
R6	00110	R17	10001	R28	11100
R7	00111	R18	10010	R29	11101
R8	01000	R19	10011	R30	11110
R9	01001	R20	10100	R31	11111
R10	01010	R21	10101		

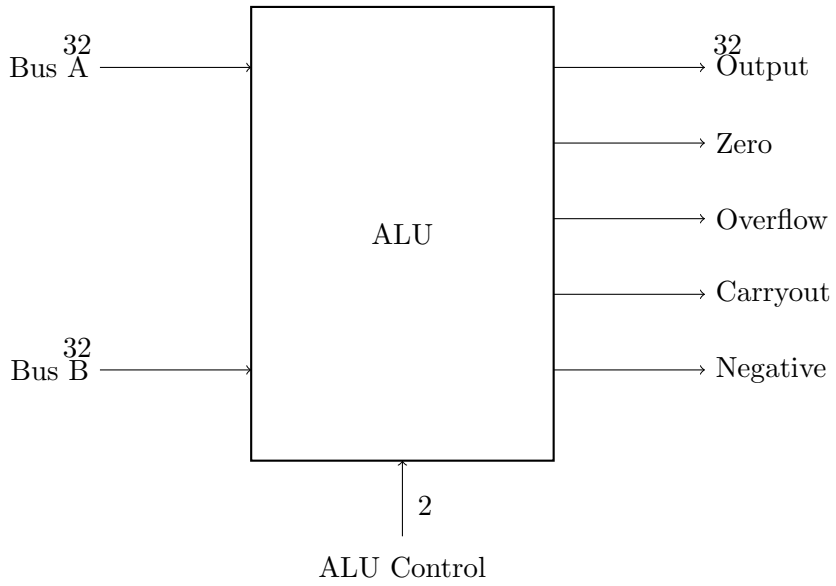
4 ALU Module Description

Inputs/Outputs

Input/Output	Description
`operand1 (32 bits)`	First operand
`operand2 (32 bits)`	Second operand
`mode (4 bits)`	Mode selector for the operation
`out (32 bits)`	Result of the operation

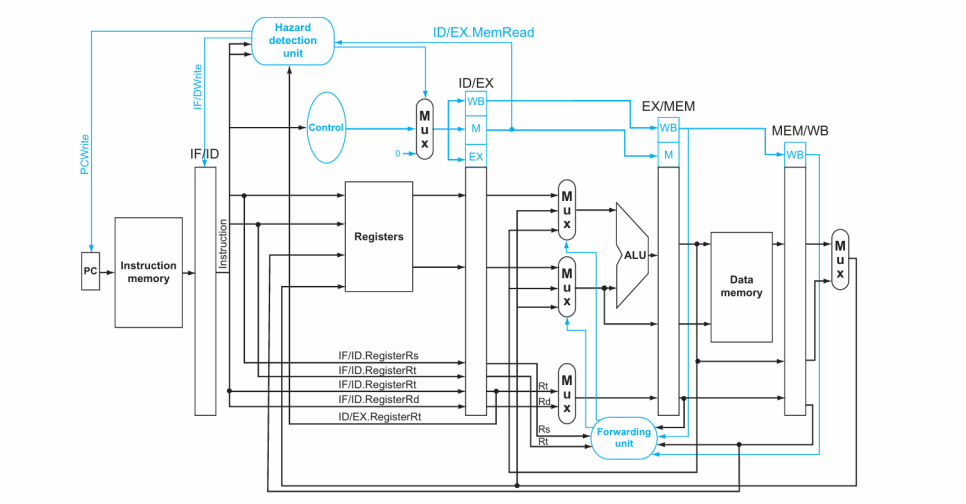
Mode	Operation
0000	Addition of `operand1` and `operand2`
0001	Subtraction of `operand2` from `operand1`
0100	Bitwise XOR of `operand1` and `operand2`
1010	Addition of `operand1` and `operand2` with `operand2` zero-extended to 32 bits

5 Instructions and Function Op-code



ALUOp	Function	ALUControl	ALU Operation	Instruction
11	xxxxxx	01	XOR	XORI
00	xxxxxx	00	Add	LW - SW
01	xxxxxx	10	Sub	BNE
10	100000	00	Add	R-type: ADD
10	100010	10	Sub	R-type: SUB
10	101010	11	SLT	R-type: SLT

6 Data Path



data path and control signals

7 Instruction Formats

7.1 R-Type Instructions

Field	Bits	Description
Opcode	31-26	00 0000
RS	25-21	Source Register 1
RT	20-16	Source Register 2
RD	15-11	Destination Register
ignored	10-6	doesn't care
Func	5-0	Function

Examples

Instruction	Binary Representation
ADD R1, R2, R3	000000 00010 00011 00001 00000 000001
SLT R5, R5, R7	000000 00101 00111 00101 00000 000111

7.2 I-Type Instructions

Field	Bits	Description
Opcode	31-26	opcode for different instructions
RS	25-21	Source Register
RT	20-16	Destination Register
Immediate Data	15-0	16-bit Immediate Data

Examples

Instruction	Binary Representation
LW R2, 10(R6)	100001 00110 00010 0000000000001010
SW R2, -2(R11)	100010 01011 00010 1111111111111110
BZ R8, -75	110011 01000 01000 1111111111011011

7.3 J-Type Instructions

Field	Bits	Description
Opcode	31-26	opcode for different instructions
Immediate Data	25-0	26-bit Imm. Data

Examples

Instruction	Binary Representation
BR #10	110000 00000000000000000001010

8 Control Signals

Instruction	RegDst	ALUSrc	Mem_Reg	RegWrite	MemRead	MemWrite	Branch	ALUOp1	ALUOp0	SignZero
R-type	1	0	0	1	0	0	0	1	0	0
LW	0	1	1	1	1	0	0	0	0	0
SW	0	1	0	0	0	1	0	0	0	0
BNE	0	0	0	0	0	0	1	0	1	0
XORI	0	1	0	1	0	0	0	1	1	1
J	0	0	0	0	0	0	0	0	0	0

```
module Control(RegDst,ALUSrc,MemtoReg,RegWrite,MemRead,MemWrite,
Branch,ALUOp,Jump, SignZero, Opcode
);

output RegDst, ALUSrc, MemtoReg, RegWrite, MemRead, MemWrite, Branch, J
output [1:0] ALUOp;
input [5:0] Opcode;

reg RegDst, ALUSrc, MemtoReg, RegWrite, MemRead, MemWrite, Branch, Jump
reg [1:0] ALUOp;

always @(*)
casex (Opcode)
  6'b000000 : begin // R-type
    RegDst = 1; ALUSrc = 0; MemtoReg = 0; RegWrite = 1;
```

```
    MemRead = 0; MemWrite = 0; Branch = 0;
    ALUOp = 2'b10; Jump = 0; SignZero = 0;
  end
  6'b100011 : begin // LW
    RegDst = 0; ALUSrc = 1; MemtoReg = 1; RegWrite = 1;
    MemRead = 1; MemWrite = 0; Branch = 0;
    ALUOp = 2'b00; Jump = 0; SignZero = 0;
  end
  6'b101011 : begin // SW
    RegDst = 1'bx; ALUSrc = 1; MemtoReg = 1'bx; RegWrite = 0;
    MemRead = 0; MemWrite = 1; Branch = 0;
    ALUOp = 2'b00; Jump = 0; SignZero = 0;
  end
  6'b000101 : begin // BNE
    RegDst = 0; ALUSrc = 0; MemtoReg = 0; RegWrite = 0;
    MemRead = 0; MemWrite = 0; Branch = 1;
    ALUOp = 2'b01; Jump = 0; SignZero = 0;
  end
  6'b001110 : begin // XORI
    RegDst = 0; ALUSrc = 1; MemtoReg = 0; RegWrite = 1;
    MemRead = 0; MemWrite = 0; Branch = 0;
    ALUOp = 2'b11; Jump = 0; SignZero = 1;
  end
  6'b000010 : begin // Jump
    RegDst = 0; ALUSrc = 0; MemtoReg = 0; RegWrite = 0;
    MemRead = 0; MemWrite = 0; Branch = 0;
    ALUOp = 2'b00; Jump = 1; SignZero = 0;
  end
  default : begin
    RegDst = 0; ALUSrc = 0; MemtoReg = 0; RegWrite = 0;
    MemRead = 0; MemWrite = 0; Branch = 0;
    ALUOp = 2'b10; Jump = 0; SignZero = 0;
  end
endcase

endmodule
```

9 Hazards:

9.1 Forwarding Unit Design

Condition	Variable	Truth Table / Equation
MEM_RegWrite == 1 MEM_WriteRegister != 0 MEM_WriteRegister == EX_rs	a b c	3*x = a & b & c
WB_RegWrite == 1 WB_WriteRegister != 0 WB_WriteRegister == EX_rs	d e f	3*y = d & e & f
ForwardA[1] = x ForwardA[0] = ∼x & y		
ForwardB is similar to ForwardA, replacing EX_rs with EX_rt		

Forwarding Logic Explanation

We designed for only for add and sub hazards for lw and sw we need to implement path from MEM/WB to EX/MEM

- **ForwardA Signal Interpretation**
  - `10`: Data is forwarded from the **EX/MEM** pipeline stage.
  - `01`: Data is forwarded from the **MEM/WB** pipeline stage.
  - `00`: No forwarding; data is sourced normally from the **ID/EX** pipeline register.
- **ForwardB Signal**
  - Logic for `ForwardB` mirrors that of `ForwardA`, with the source register `EX_rs` replaced by `EX_rt`.
- This forwarding mechanism effectively resolves **data hazards** by bypassing the regular data path and providing the most recent operand values directly from intermediate pipeline stages.

9.2 Stall Control Unit

Condition	Variable
EX_MemRead == 1	a
EX_rt == ID_rs	b
EX_rt == ID_rt	c
Opcode != 6'b001110 (XORI)	e
Opcode != 6'b100011 (LW)	f
Condition = a AND (b OR (c AND e AND f))	

Data hazards requiring a one-cycle stall occur when the destination register (`EX_rt`) of the current memory-read instruction matches the source register (`ID_rs` or `ID_rt`) of the instruction currently in the Instruction Decode (ID) stage.

An important exception applies to `XORI` and `LW` instructions, where `ID_rt` is used as the **destination register** rather than a source. Therefore, `ID_rt` is not considered for hazard detection in these cases.

*Truth Table

Condition	PC_WriteEn	IFID_WriteEn	Stall_flush
1	0	0	1
0	1	1	0

9.3 Flush Control Unit

If a `load` instruction is immediately followed by a conditional `branch` that depends on the load result, two stall cycles will be needed, as the result from the `load` appears at the end of the `MEM` stage but is needed at the beginning of the `ID` stage for branch evaluation.

The flush unit is responsible for ensuring correct pipeline behavior in the presence of control hazards, particularly due to branch instructions, jump register (‘jr‘), or unconditional jumps. When a control hazard is detected, certain instructions in the pipeline must be invalidated or discarded to maintain program correctness.

9.3.1 Discard Instruction Logic

The `Discard_Instr` module generates two flush signals: `ID_flush` and `IF_flush`. These signals are asserted based on the occurrence of jump (`jump`), branch not equal (`bne`), or jump register (`jr`) instructions. Specifically:

- `IF_flush` is asserted when any of `jump`, `bne`, or `jr` are high.
- `ID_flush` is asserted when either `bne` or `jr` are high.

9.3.2 Flush Control Block

The `flush_block` module clears or passes control signals from the ID stage to the next pipeline stage, based on the `flush` signal. If `flush` is high, the output control signals (`ID_RegDst`, `ID_ALUSrc`, etc.) are forced to zero, effectively nullifying the instruction in the pipeline. Otherwise, control signals are propagated normally.

This is achieved by ANDing each control signal with the negated flush signal (`notflush`). As a result, when a flush is triggered due to a control hazard, the control signals entering the Execute stage are zeroed out, thereby transforming the instruction into a no-operation (NOP).

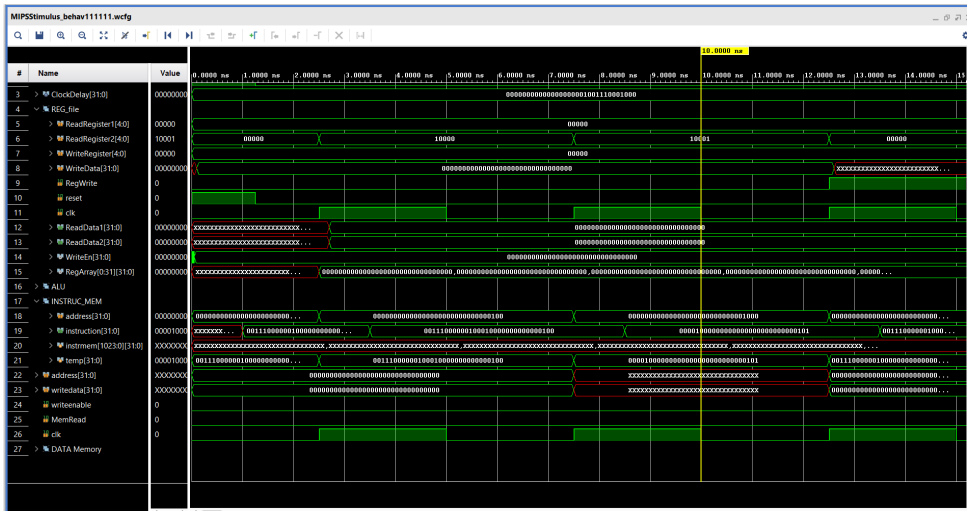
10 Instruction Sequence for Hazard Testing

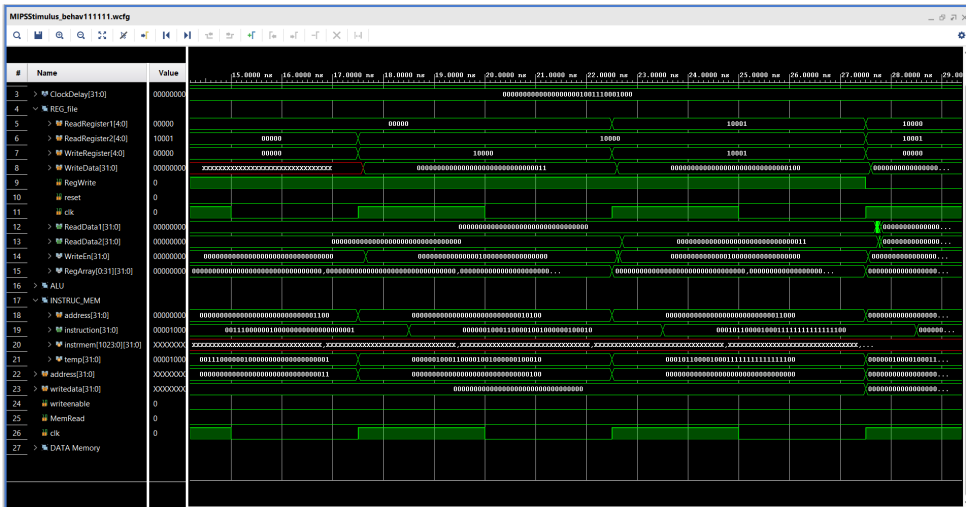
The following MIPS instruction sequence is used in the testbench to observe and verify various pipeline hazards such as data hazards, control hazards, and the effectiveness of forwarding and stalling mechanisms.

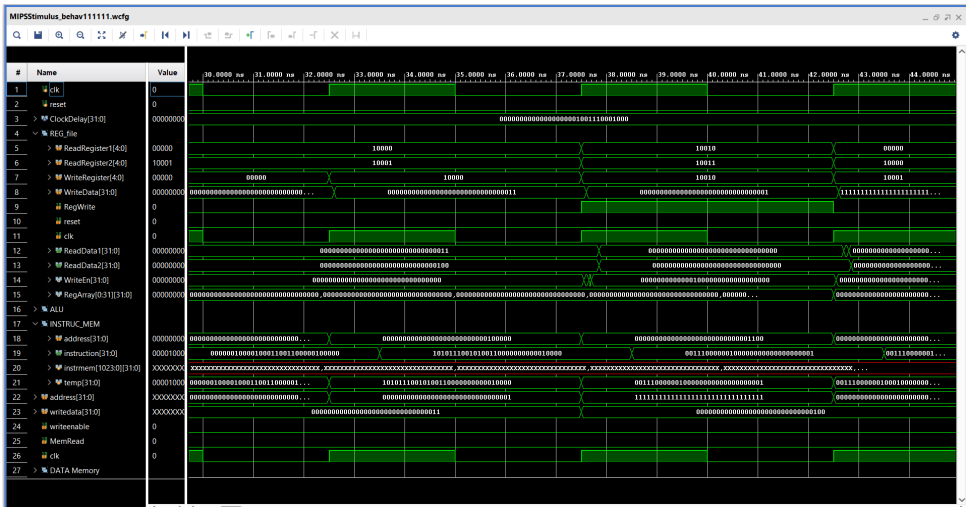
```
start:   xori $s0, $zero, 0x0003
         xori $s1, $zero, 0x0004
         j next1              # → Control hazard

next2:   xori $s0, $zero, 0x0001
         xori $s1, $zero, 0x0001

next1:   sub  $s2, $s1, $s0     # → Data hazard (forwarding)
         bne  $s0, $s1, next2   # → Data and Control hazard (forwarding)
         add  $s3, $s0, $s1
         sw   $s3, 16($s2)      # → Data hazard (forwarding)
         lw   $s4, 16($s2)
         slt  $s5, $s0, $s4     # → Data hazard (stalling and forwarding)
         lw   $s3, 16($s2)
         xori $s3, $s2, 0x0001  # → No data hazard (stalling)
         xori $s5, $s5, 0x0001
         jr   $s5               # → Data and Control hazard (forwarding)
```

Hazard Annotations:

- **Control Hazards:** Introduced by the `j` and `bne` instructions; handled via flushing.
- **Data Hazards:** Various types of RAW (Read After Write) hazards handled using forwarding logic or pipeline stalling.
- **Stalling:** Required when forwarding is not sufficient, e.g., after certain `lw` instructions.
- **Forwarding:** Applied where possible to reduce stalls and improve performance.







GitHub Repository

You can find the MIPS32 project here:

`https://github.com/aseem4772/MIPS32`