# PROJECT – PIPELINE DESIGN OF MIPS32 WITH VERILOG

-HARSHIT RAGHUVANSHI(22JE0395)

# A Quick Look at MIPS32

- MIPS32 registers:
  a) 32, 32-bit general purpose registers (GPRs), *R0* to *R31*.
     - Register *R0* contains a constant 0; cannot be written.
  b) A special-purpose 32-bit program counter (*PC*).
     - Points to the next instruction in memory to be fetched and executed.
- No flag registers (zero, carry, sign, etc.).
- Very few addressing modes (register, immediate, register indexed, etc.)
  – Only load and store instructions can access memory.
- We assume memory word size is 32 bits (*word addressable*).

# The MIPS32 Instruction Subset Being Considered

- Load and Store Instructions

```
LW   R2,124(R8)     // R2 = Mem[R8+124]
SW   R5,-10(R25)    // Mem[R25-10] = R5
```

- Arithmetic and Logic Instructions (only register operands)

```
ADD   R1,R2,R3      // R1 = R2 + R3
ADD   R1,R2,R0      // R1 = R2 + 0
SUB   R12,R10,R8    // R12 = R10 - R8
AND   R20,R1,R5     // R20 = R1 & R5
OR    R11,R5,R6     // R11 = R5 | R6
MUL   R5,R6,R7      // R5 = R6 * R7
SLT   R5,R11,R12    // If R11 < R12, R5=1; else R5=0
```

4

- Arithmetic and Logic Instructions (immediate operand)

```
ADDI R1,R2,25      // R1 = R2 + 25
SUBI R5,R1,150     // R5 = R1 - 150
SLTI R2,R10,10     // If R10<10, R2=1; else R2=0
```

- Branch Instructions

```
BEQZ R1,Loop       // Branch to Loop if R1=0
BNEQZ R5,Label     // Branch to Label if R5!=0
```

- Jump Instruction

```
J    Loop          // Branch to Loop unconditionally
```

- Miscellaneous Instruction

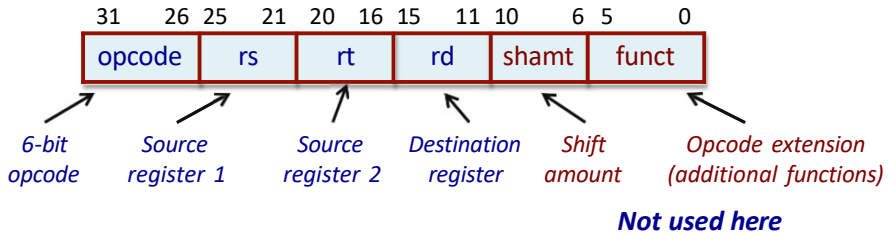
```
HLT                // Halt execution
```

# MIPS Instruction Encoding

- All MIPS32 instructions can be classified into three groups in terms of instruction encoding.
  - R-type (Register), I-type (Immediate), and J-type (Jump).
  - In an instruction encoding, the 32 bits of the instruction are divided into several fields of fixed widths.
  - All instructions may not use all the fields.
- Since the relative positions of some of the fields are same across instructions, instruction decoding becomes very simple.

## *(a) R-type Instruction Encoding*

- Here an instruction can use up to three register operands.
  - Two source and one destination.
- In addition, for shift instructions, the number of bits to shift can also be specified (*we are not considering such instructions here*).



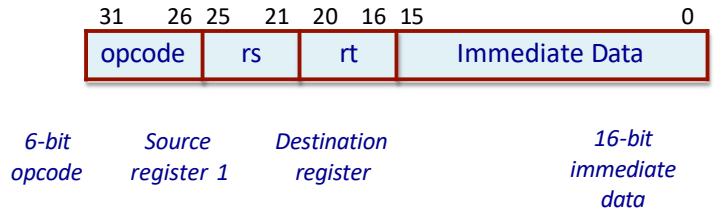

- R-type instructions considered with opcode:

| Instruction | opcode |
|---|---|
| ADD | 000000 |
| SUB | 000001 |
| AND | 000010 |
| OR | 000011 |
| SLT | 000100 |
| MUL | 000101 |
| HLT | 111111 |

```
SUB    R5,R12,R25

000001 01100 11001 00101 00000 000000
 SUB    R12   R25    R5

= 05992800 (in hex)
```

## *(b) I-type Instruction Encoding*

- Contains a 16-bit immediate data field.
- Supports one source and one destination register.




- I-type instructions considered with opcode:

| Instruction | opcode |
|---|---|
| LW | 001000 |
| SW | 001001 |
| ADDI | 001010 |
| SUBI | 001011 |
| SLTI | 001100 |
| BNEQZ | 001101 |
| BEQZ | 001110 |

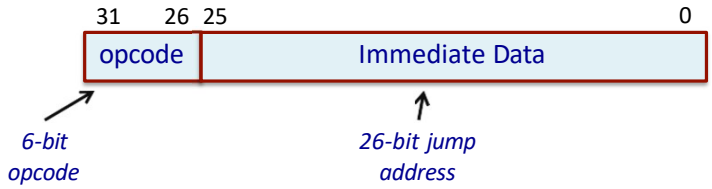
```
LW    R20,84(R9)

001000 01001 10100 0000000001010100
  LW     R9    R20        offset

= 21340054 (in hex)
```

```
BEQZ    R25,Label

001110 11001 00000  yyyyyyyyyyyyyyyy
 BEQZ   R25  Unused        offset

= 3b20YYYY (in hex)
```

1

## *(c) J-type Instruction Encoding*

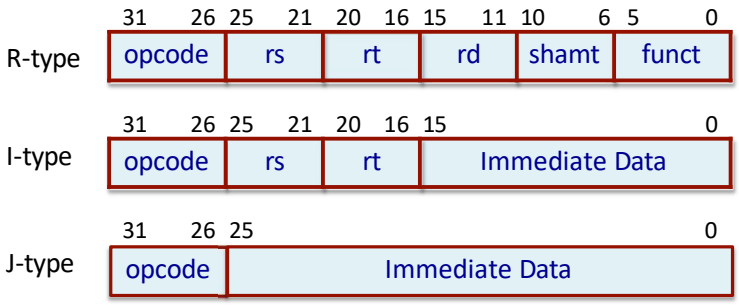
- Contains a 26-bit jump address field.
  - Extended to 28 bits by padding two 0's on the right.




| Instruction | opcode |
|---|---|
| J | 010000 |

Not Implemented

## A Quick View




- Some instructions require two register operands *rs* & *rt* as input, while some require only *rs*.
- Gets known only after instruction is decoded.
- While decoding is going on, we can prefetch the registers in parallel.
  - May or may not be required later.

- Similarly, the 16-bit and 26-bit immediate data are retrieved and sign-extended to 32-bits in case they are required later.

# Addressing Modes in MIPS32

- Register addressing *ADD R1,R2,R3*
- Immediate addressing *ADDI R1,R2, 200*
- Base addressing *LW R5, 150(R7)*
  - Content of a register is added to a "base" value to get the operand address.
- PC relative addressing *BEQZ R3, Label*
  - 16-bit offset is added to PC to get the target address.
- Pseudo-direct addressing *J Label*
  - 26-bit offset is added to PC to get the target address.

# MIPS32 Instruction Cycle

- We divide the instruction execution cycle into five steps:
  a) IF : Instruction Fetch
  b) ID : Instruction Decode / Register Fetch
  c) EX : Execution / Effective Address Calculation
  d) MEM : Memory Access / Branch Completion
  e) WB : Register Write-back
- We now show the generic micro-instructions carries out in the various steps.

# (a) IF : Instruction Fetch

- Here the instruction pointed to by *PC* is fetched from memory, and also the next value of *PC* is computed.
  - Every MIPS32 instruction is of 32 bits.
  - Every memory word is of 32 bits and has a unique address.
  - For a branch instruction, new value of the *PC* may be the target address. So *PC* is not updated in this stage; new value is stored in a register *NPC*.

**IF:**

IR ← Mem [PC];
NPC ← PC + 1 ;

For byte addressable memory, PC has to be incremented by 4.

# (b) ID : Instruction Decode

- The instruction already fetched in *IR* is decoded.
  - *Opcode* is 6-bits (bits 31:26).
  - First source operand *rs* (bits 25:21), second source operand *rt* (bits 20:16).
  - 16-bit immediate data (bits 15:0).
  - 26-bit immediate data (bits 25:0).
- Decoding is done in parallel with reading the register operands *rs* and *rt*.
  - Possible because these fields are in a fixed location in the instruction format.
- In a similar way, the immediate data are sign-extended.

**ID:**

A ← Reg [rs];
B ← Reg [rt];
Imm ← (IR15)16 ## IR15..0 // sign extend 16-bit immediate field
Imm1 ← (IR25)6 ## IR25..0 // sign extend 26-bit immediate field

***A, B, Imm, Imm1 are temporary registers.***

## (c) EX: Execution / Effective Address Computation

- In this step, the ALU is used to perform some calculation.
  - The exact operation depends on the instruction that is already decoded.
  - The ALU operates on operands that have been already made ready in the previous cycle.
    - A, B, Imm, etc.
- We show the micro-instructions corresponding to the type of instruction.

| | |
|---|---|
| Memory Reference:<br>ALUOut ← A + Imm; | Example: LW R3, 100(R8) |
| Register-Register ALU Instruction:<br>ALUOut ← A func B; | Example: SUB R2, R5, R12 |
| Register-Immediate ALU Instruction:<br>ALUOut ← A func Imm; | Example: SUBI R2, R5, 524 |
| Branch:<br>ALUOut ← NPC + Imm ;<br>cond ← (A op 0); | Example: BEQZ R2, Label<br>[op is ==] |

# (d) MEM: Memory Access / Branch Completion

- The only instructions that make use of this step are loads, stores, and branches.
  - The load and store instructions access the memory.
  - The branch instruction updates *PC* depending upon the outcome of the branch condition.

Load instruction:

PC ← NPC;

LMD ← Mem [ALUOut];

Store instruction:

PC ← NPC;

Mem [ALUOut] ← B;

Branch instruction:

if (cond) PC ← ALUOut;

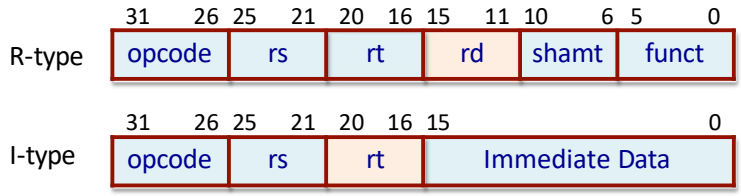
else PC ← NPC;

Other instructions:

PC ← NPC;

# (e) WB: Register Write Back

- In this step, the result is written back into the register file.
  - Result may come from the ALU.
  - Result may come from the memory system (viz. a LOAD instruction).
- The position of the destination register in the instruction word depends on the instruction → *already known after decoding has been done*.



Register-Register ALU Instruction:

Reg [rd] ← ALUOut;

Register-Immediate ALU Instruction:

Reg [rt] ← ALUOut;

Load Instruction:

Reg [rt] ← LMD;

## ADD R2, R5, R10

| | |
|---|---|
| IF | IR ← Mem [PC];<br>NPC ← PC + 1 ; |
| ID | ✌ ← Reg [rs];<br>👌 ← Reg [rt]; |
| EX | ALUOut ← A + B; |
| MEM | PC ← NPC; |
| WB | Reg [rd] ← ALUOut; |

## ADDI R2, R5, 150

| | |
|---|---|
| IF | IR ← Mem [PC];<br>NPC ← PC + 1; |
| ID | A ← Reg [rs];<br>Imm ← $(IR_{15})^{16}$ ## $IR_{15..0}$ |
| EX | ALUOut ← A + Imm; |
| MEM | PC ← NPC; |
| WB | Reg [rt] ← ALUOut; |

## LW R2, 200 (R6)

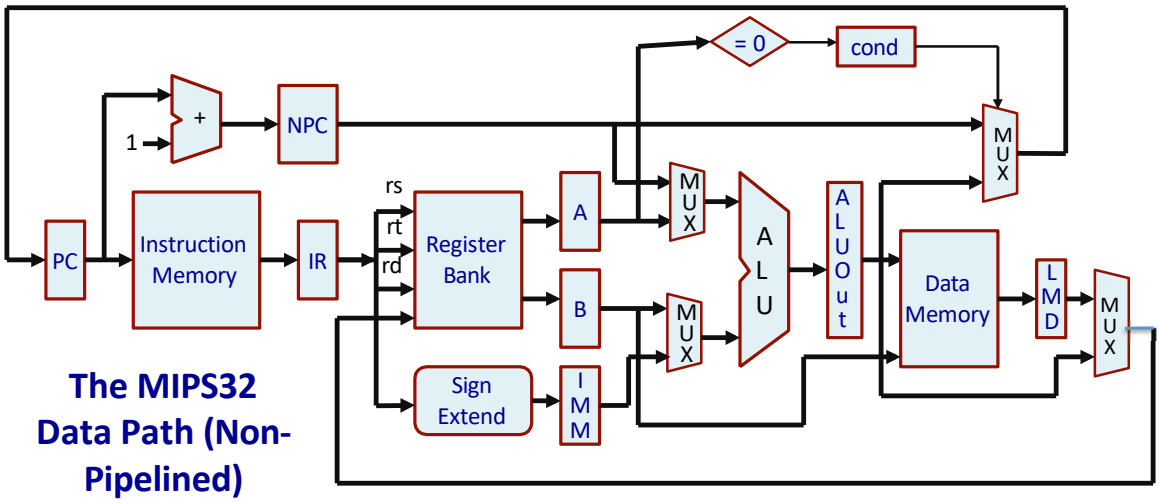
| | |
|---|---|
| IF | IR ← Mem [PC];<br>NPC ← PC + 1; |
| ID | A ← Reg [rs];<br>Imm ← $(IR_{15})^{16}$ ## $IR_{15..0}$ |
| EX | ALUOut ← A + Imm; |
| MEM | PC ← NPC;<br>LMD ← Mem [ALUOut]; |
| WB | Reg [rt] ← LMD; |

## SW R3, 25 (R10)

| | |
|---|---|
| IF | IR ← Mem [PC];<br>NPC ← PC + 1; |
| ID | ✌ ← Reg [rs];<br>👌 ← Reg [rt];<br>Imm ← $(IR_{15})^{16}$ ## $IR_{15..0}$ |
| EX | ALUOut ← A + Imm; |
| MEM | PC ← NPC;<br>Mem [ALUOut] ← B; |
| WB | - |

## BEQZ R3, Label

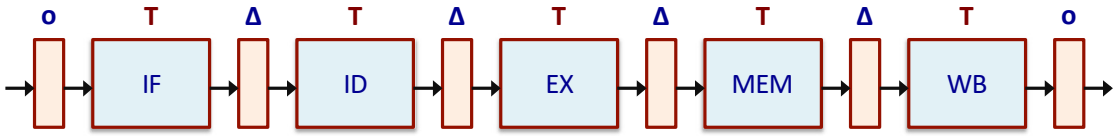
| | |
|---|---|
| **IF** | IR ← Mem [PC];<br>NPC ← PC + 1; |
| **ID** | A ← Reg [rs];<br>Imm ← $(IR_{15})^{16}$ ## $IR_{15..0}$ |
| **EX** | ALUOut ← NPC + Imm;<br>cond ← (A == 0); |
| **MEM** | PC ← NPC;<br>if (cond) PC ← ALUOut; |
| **WB** | - |




**The MIPS32 Data Path (Non-Pipelined)**

# Introduction

- Basic requirements for pipelining the MIPS32 data path:
  - We should be able to start a new instruction every clock cycle.
  - Each of the five steps mentioned before (IF, ID, EX, MEM and WB) becomes a pipeline stage.
  - Each stage must finish its execution within one clock cycle.




***Clock Cycles***

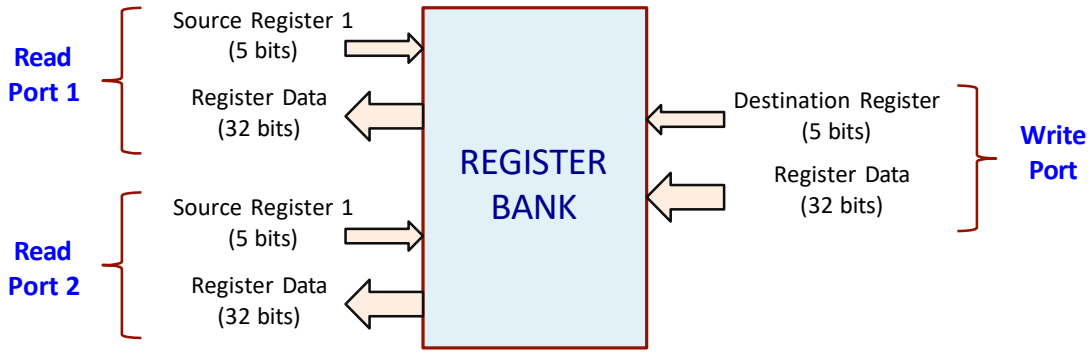
| Instruction | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| ***i*** | IF | ID | EX | MEM | WB | | | |
| ***i + 1*** | | IF | ID | EX | MEM | WB | | |
| ***i + 2*** | | | IF | ID | EX | MEM | WB | |
| ***i + 3*** | | | | IF | ID | EX | MEM | WB |

*Instr-i finishes*

*Instr-(i+1) finishes*

*Instr-(i+2) finishes*

*Instr-(i+3) finishes*

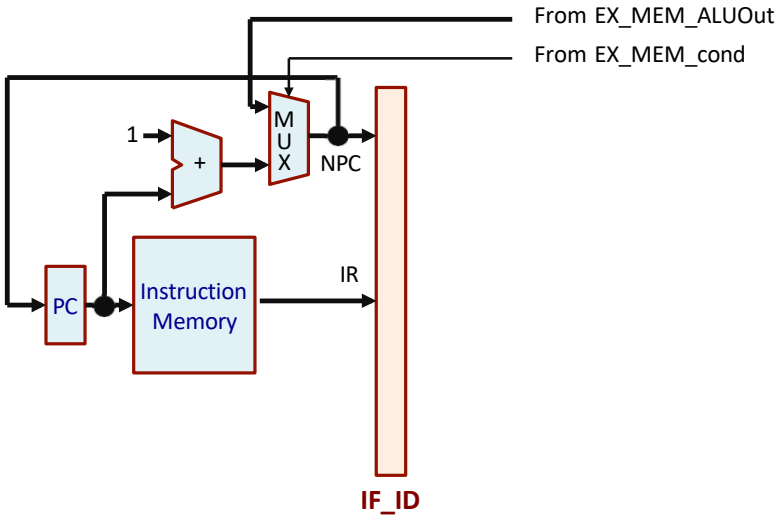



# Micro-operations for Pipelined MIPS32

- Convention used:
  - Most of the temporary registers required in the data path are included as part of the inter-stage latches.
  - **IF_ID**: denotes the latch stage between the IF and ID stages.
  - **ID_EX**: denotes the latch stage between the ID and EX stages.
  - **EX_MEM**: denotes the latch stage between the EX and MEM stages.
  - **MEM_WB**: denotes the latch stage between the MEM and WB stages.
- Example:
  - *ID_EX_A* means register *A* that is implemented as part of the *ID_EX* latch stage.

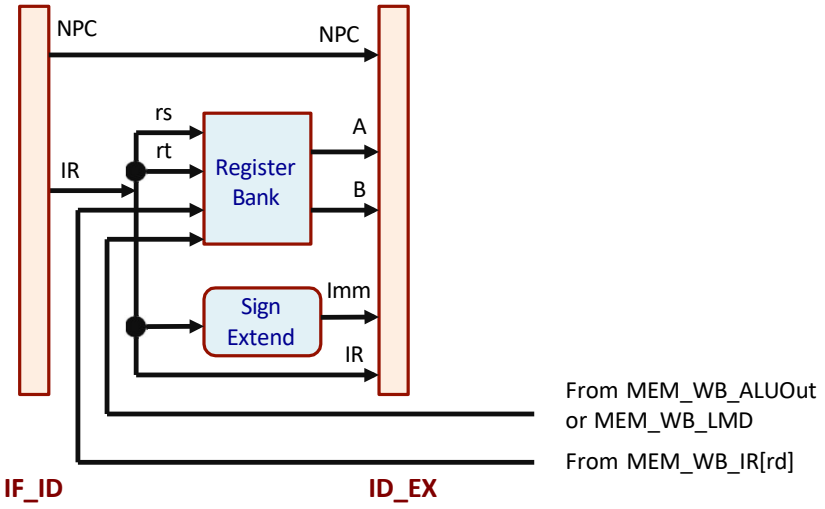
## (a) Micro-operations for Pipeline Stage IF

```
IF_ID_IR          ←  Mem [PC];
IF_ID_NPC,PC  ←  ( if ((EX_MEM_IR[opcode] == branch) & EX_MEM_cond)
                        { EX_MEM_ALUOut}
                    else  {PC + 1} );
```



## (b) Micro-operations for Pipeline Stage ID

ID_EX_A ← Reg [IF_ID_IR [rs]];
ID_EX_B ← Reg [IF_ID_IR [rt]];
ID_EX_NPC ← IF_ID_NPC;
ID_EX_IR ← IF_ID_IR;
ID_EX_Imm ← sign-extend ($IF\_ID\_IR_{15..0}$);



## (c) Micro-operations for Pipeline Stage EX

```
EX_MEM_IR        ← ID_EX_IR;
EX_MEM_ALUOut    ← ID_EX_A  func  ID_EX_B;
```

**R-R ALU**

```
EX_MEM_IR        ← ID_EX_IR;
EX_MEM_ALUOut    ← ID_EX_A  func  ID_EX_Imm;
```

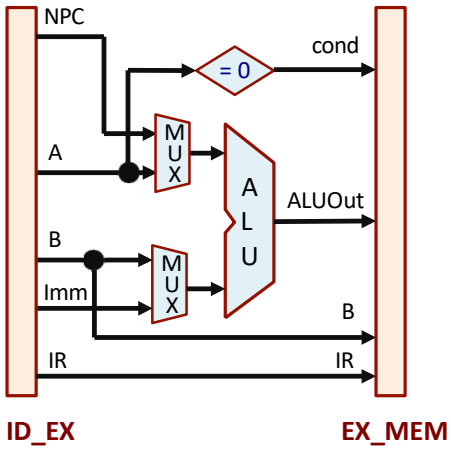
**R-M ALU**

```
EX_MEM_ALUOut    ← ID_EX_NPC +
                   ID_EX_Imm;
EX_MEM_cond      ← (ID_EX_A == 0);
```

**BRANCH**

```
EX_MEM_IR        ← ID_EX_IR;
EX_MEM_ALUOut    ← ID_EX_A  + ID_EX_Imm;
EX_MEM_B         ← ID_EX_B;
```

**LOAD / STORE**



# (d) Micro-operations for Pipeline Stage MEM

```
MEM_WB_IR ← EX_MEM_IR;
MEM_WB_ALUOut ← EX_MEM_ALUOut;
```

**ALU**

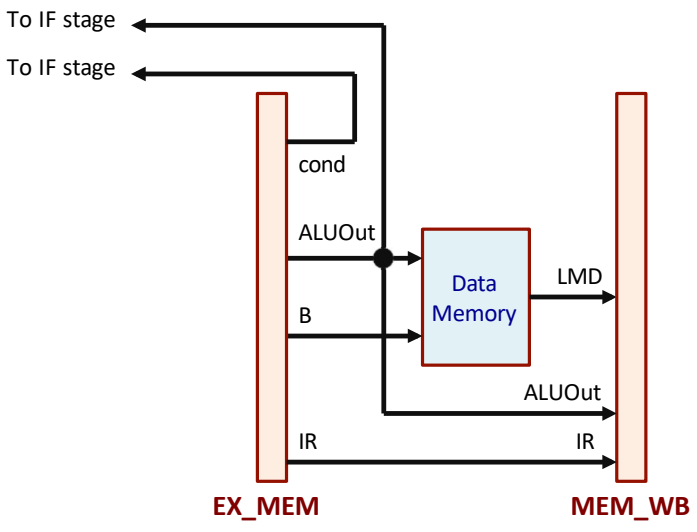
```
MEM_WB_IR ← EX_MEM_IR;
MEM_WB_LMD ← Mem [EX_MEM_ALUOut];
```

**LOAD**

```
MEM_WB_IR ← EX_MEM_IR;
Mem [EX_MEM_ALUOut] ← EX_MEM_B;
```

**STORE**

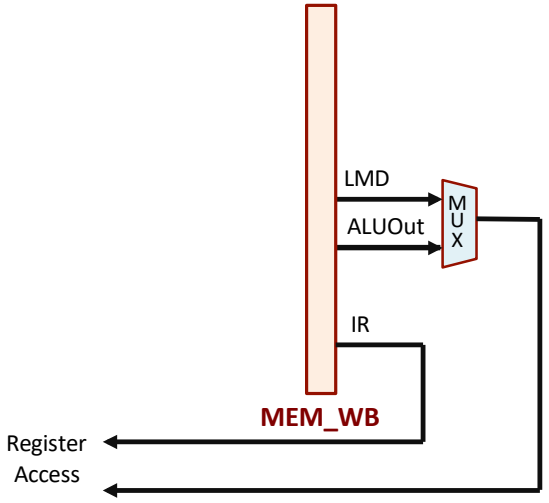





Hardware Modeling Using Verilog

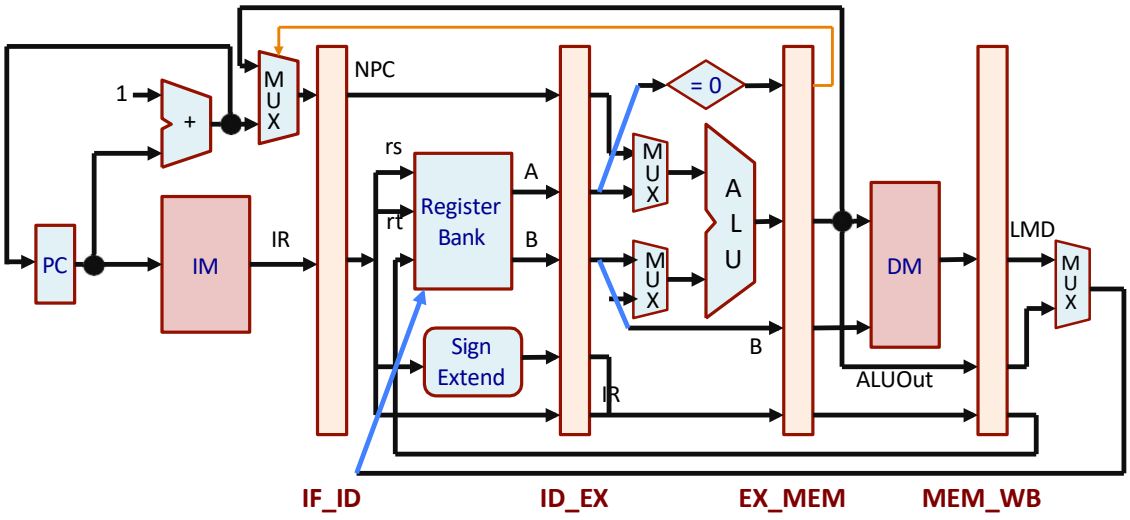
# (e) Micro-operations for Pipeline Stage WB

| | |
|---|---|
| Reg [MEM_WB_IR [rd]] ← MEM_WB_ ALUOut; | **R-R ALU** |
| Reg [MEM_WB_IR [rt]] ← MEM_WB_ ALUOut; | **R-M ALU** |
| Reg [MEM_WB_IR [rt]] ← MEM_WB_ LMD; | **LOAD** |



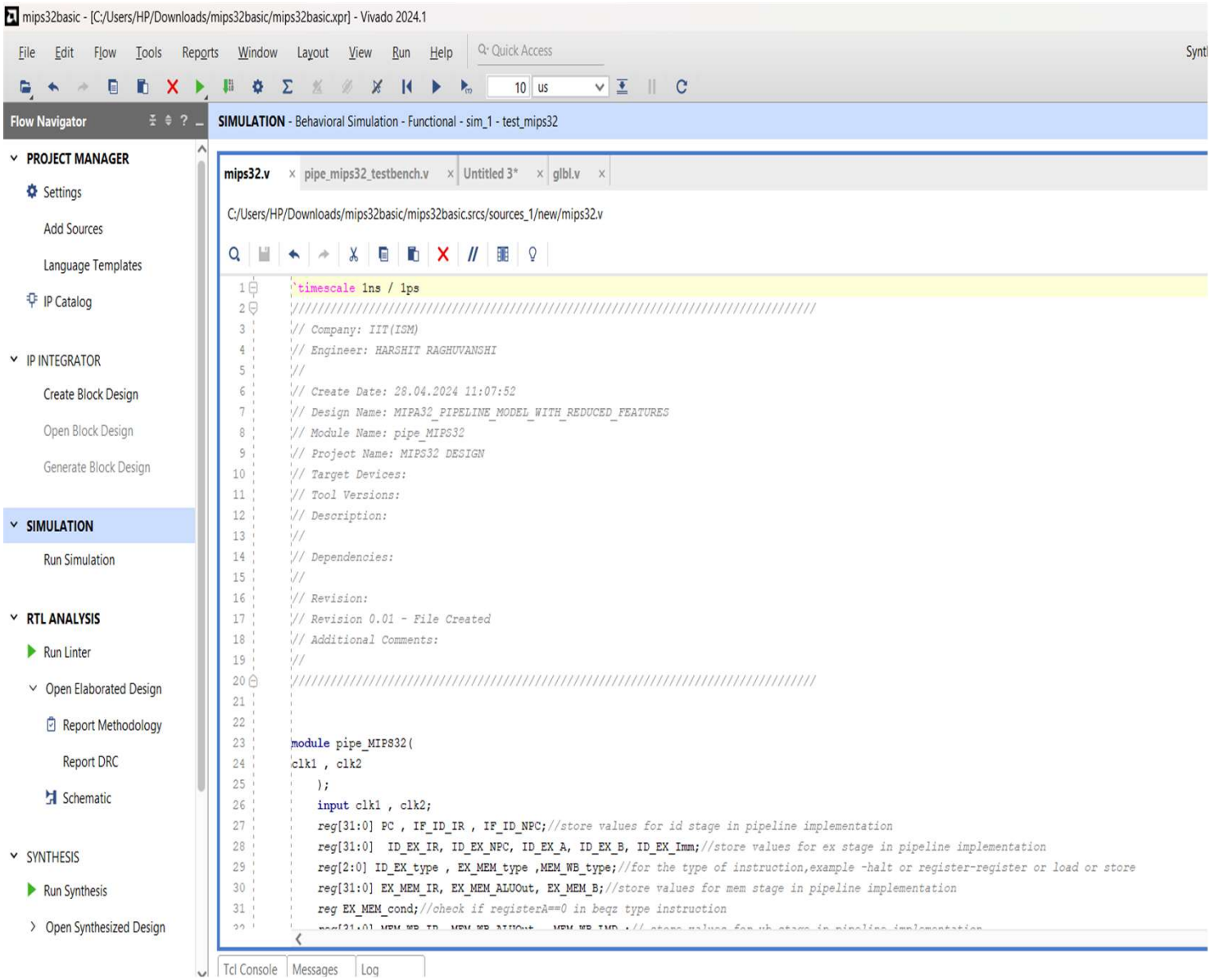
# PUTTING IT ALL TOGETHER :: MIPS32 PIPELINE






Hardware Modeling Using Verilog

# VERILOG MODELING OF THE PROCESSOR -FOLLOWING IS THE VERILOG CODE FOR MODELLING THE PROCESSOR



mips32basic - [C:/Users/HP/Downloads/mips32basic/mips32basic.xpr] - Vivado 2024.1

File Edit Flow Tools Reports Window Layout View Run Help Quick Access

Flow Navigator

**PROJECT MANAGER**
- Settings
- Add Sources
- Language Templates
- IP Catalog

**IP INTEGRATOR**
- Create Block Design
- Open Block Design
- Generate Block Design

**SIMULATION**
- Run Simulation

**RTL ANALYSIS**
- Run Linter
- Open Elaborated Design
  - Report Methodology
  - Report DRC
  - Schematic

**SYNTHESIS**
- Run Synthesis
- Open Synthesized Design

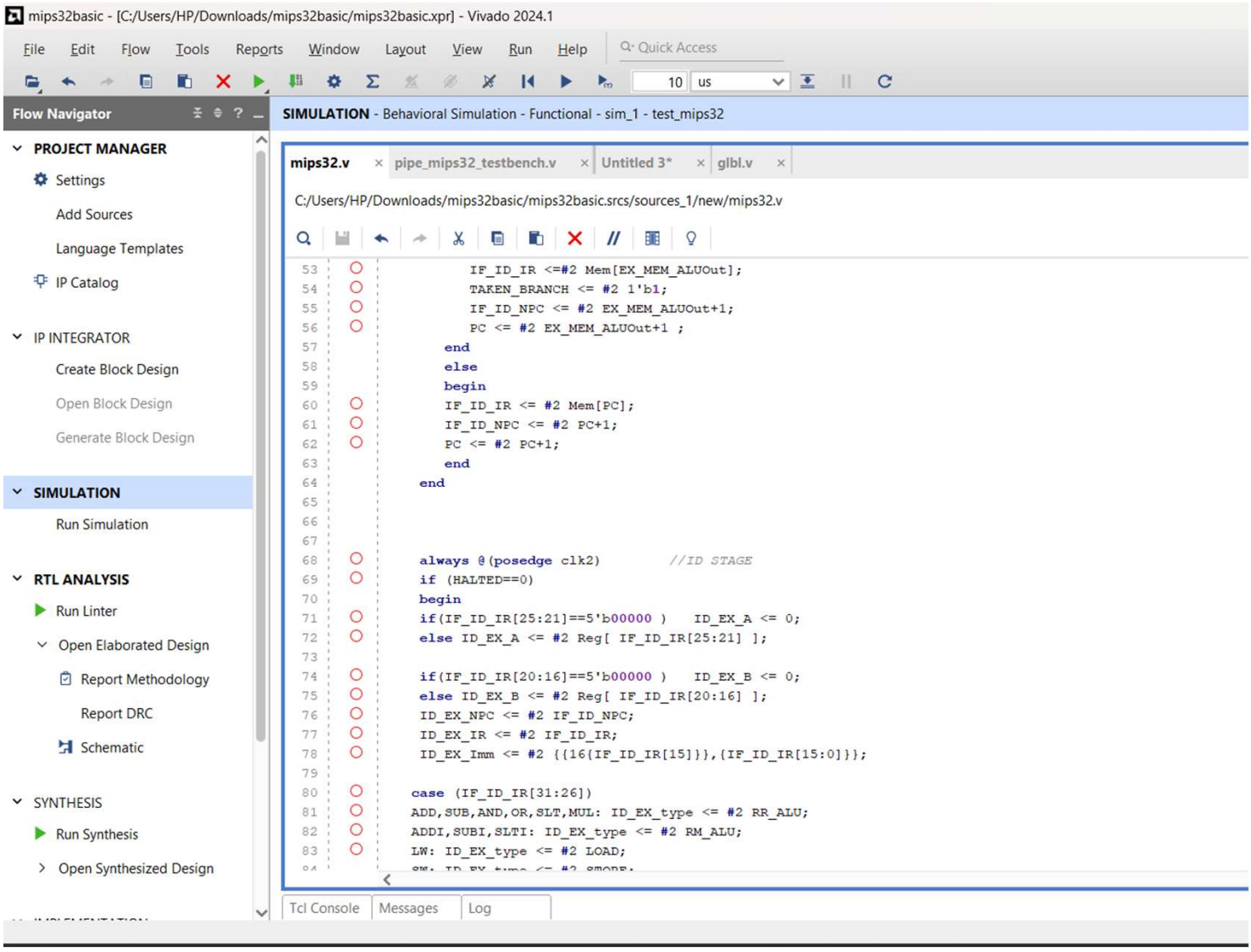
**SIMULATION** - Behavioral Simulation - Functional - sim_1 - test_mips32

mips32.v | pipe_mips32_testbench.v | Untitled 3* | glbl.v

C:/Users/HP/Downloads/mips32basic/mips32basic.srcs/sources_1/new/mips32.v

```
31      reg EX_MEM_cond;//check if registerA==0 in beqz type instruction
32      reg[31:0] MEM_WB_IR, MEM_WB_ALUOut , MEM_WB_LMD ;// store values for wb stage in pipeline implem
33
34      reg[31:0] Reg[0:31];
35      reg [31:0] Mem[0:1023];
36      parameter ADD=6'b000000, SUB=6'b000001, AND=6'b000010,OR=6'b000011, SLT=6'b000100,
37      MUL=6'b000101, HLT=6'b111111, LW=6'b001000,
38      SW=6'b001001, ADDI=6'b001010, SUBI=6'b001011,SLTI=6'b001100,
39      BNEQZ=6'b001101, BEQZ=6'b001110;
40      parameter RR_ALU=3'b000, RM_ALU=3'b001, LOAD=3'b010, STORE=3'b011,
41                BRANCH=3'b100, HALT=3'b101;
42      reg HALTED;
43      // Set after HLT instruction is completed (in WB stage)
44      reg TAKEN_BRANCH;
45      // Required to disable instructions after branch
46
47
48      always @(posedge clk1)          //IF STAGE
49      if (HALTED==0)
50      begin
51          if(((EX_MEM_IR==BEQZ)&&(EX_MEM_cond == 1))||((EX_MEM_IR==BNEQZ)&&(EX_MEM_cond==0)))
52          begin
53             IF_ID_IR <=#2 Mem[EX_MEM_ALUOut];
54             TAKEN_BRANCH <= #2 1'b1;
55             IF_ID_NPC <= #2 EX_MEM_ALUOut+1;
56             PC <= #2 EX_MEM_ALUOut+1 ;
57          end
58          else
59          begin
60          IF_ID_IR <= #2 Mem[PC];
61          IF_ID_NPC <= #2 PC+1;
```

Tcl Console | Messages | Log


mips32basic - [C:/Users/HP/Downloads/mips32basic/mips32basic.xpr] - Vivado 2024.1
File Edit Flow Tools Reports Window Layout View Run Help
Quick Access
10 us
Flow Navigator
SIMULATION - Behavioral Simulation - Functional - sim_1 - test_mips32
PROJECT MANAGER
Settings
Add Sources
Language Templates
IP Catalog
IP INTEGRATOR
Create Block Design
Open Block Design
Generate Block Design
SIMULATION
Run Simulation
RTL ANALYSIS
Run Linter
Open Elaborated Design
Report Methodology
Report DRC
Schematic
SYNTHESIS
Run Synthesis
Open Synthesized Design
mips32.v
pipe_mips32_testbench.v
Untitled 3*
glbl.v
C:/Users/HP/Downloads/mips32basic/mips32basic.srcs/sources_1/new/mips32.v
53 IF_ID_IR <=#2 Mem[EX_MEM_ALUOut];
54 TAKEN_BRANCH <= #2 1'b1;
55 IF_ID_NPC <= #2 EX_MEM_ALUOut+1;
56 PC <= #2 EX_MEM_ALUOut+1 ;
57 end
58 else
59 begin
60 IF_ID_IR <= #2 Mem[PC];
61 IF_ID_NPC <= #2 PC+1;
62 PC <= #2 PC+1;
63 end
64 end
65
66
67
68 always @(posedge clk2) //ID STAGE
69 if (HALTED==0)
70 begin
71 if(IF_ID_IR[25:21]==5'b00000 ) ID_EX_A <= 0;
72 else ID_EX_A <= #2 Reg[ IF_ID_IR[25:21] ];
73
74 if(IF_ID_IR[20:16]==5'b00000 ) ID_EX_B <= 0;
75 else ID_EX_B <= #2 Reg[ IF_ID_IR[20:16] ];
76 ID_EX_NPC <= #2 IF_ID_NPC;
77 ID_EX_IR <= #2 IF_ID_IR;
78 ID_EX_Imm <= #2 {{16{IF_ID_IR[15]}},{IF_ID_IR[15:0]}};
79
80 case (IF_ID_IR[31:26])
81 ADD,SUB,AND,OR,SLT,MUL: ID_EX_type <= #2 RR_ALU;
82 ADDI,SUBI,SLTI: ID_EX_type <= #2 RM_ALU;
83 LW: ID_EX_type <= #2 LOAD;
Tcl Console
Messages
Log

mips32basic - [C:/Users/HP/Downloads/mips32basic/mips32basic.xpr] - Vivado 2024.1

File Edit Flow Tools Reports Window Layout View Run Help Quick Access

Flow Navigator

**PROJECT MANAGER**
- Settings
- Add Sources
- Language Templates
- IP Catalog

**IP INTEGRATOR**
- Create Block Design
- Open Block Design
- Generate Block Design

**SIMULATION**
- Run Simulation

**RTL ANALYSIS**
- Run Linter
- Open Elaborated Design
  - Report Methodology
  - Report DRC
  - Schematic

**SYNTHESIS**
- Run Synthesis
- Open Synthesized Design

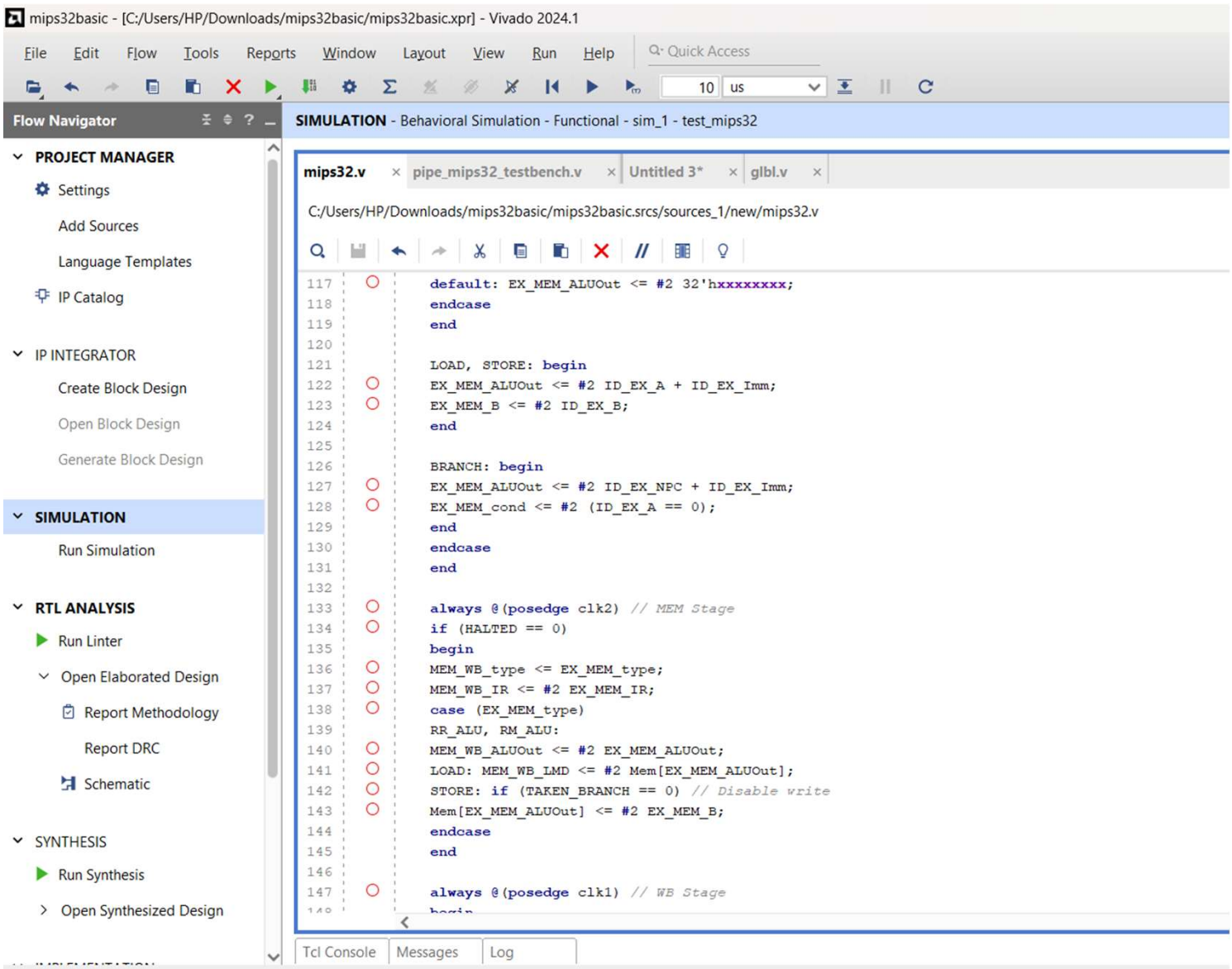
**SIMULATION** - Behavioral Simulation - Functional - sim_1 - test_mips32

mips32.v | pipe_mips32_testbench.v | Untitled 3* | glbl.v

C:/Users/HP/Downloads/mips32basic/mips32basic.srcs/sources_1/new/mips32.v

```
BNEQZ,BEQZ: ID_EX_type <= #2 BRANCH;
HLT: ID_EX_type <= #2 HALT;
default: ID_EX_type <= #2 HALT; //also halt if op code is invalid
endcase
end


always @(posedge clk1) // EX Stage
if (HALTED == 0)
begin
EX_MEM_type <= #2 ID_EX_type;
EX_MEM_IR <= #2 ID_EX_IR;
TAKEN_BRANCH <= #2 0;
case (ID_EX_type)

RR_ALU: begin
case (ID_EX_IR[31:26]) // "opcode"
ADD: EX_MEM_ALUOut <= #2 ID_EX_A + ID_EX_B;
SUB: EX_MEM_ALUOut <= #2 ID_EX_A - ID_EX_B;
AND: EX_MEM_ALUOut <= #2 ID_EX_A & ID_EX_B;
OR: EX_MEM_ALUOut <= #2 ID_EX_A | ID_EX_B;
SLT: EX_MEM_ALUOut <= #2 ID_EX_A < ID_EX_B;
MUL: EX_MEM_ALUOut <= #2 ID_EX_A * ID_EX_B;
default: EX_MEM_ALUOut <= #2 32'hxxxxxxxx;
endcase
end

RM_ALU: begin
case (ID_EX_IR[31:26]) // "opcode"
ADDI: EX_MEM_ALUOut <= #2 ID_EX_A + ID_EX_Imm;
SUBI: EX_MEM_ALUOut <= #2 ID_EX_A - ID_EX_Imm;
```

Tcl Console | Messages | Log


mips32basic - [C:/Users/HP/Downloads/mips32basic/mips32basic.xpr] - Vivado 2024.1
File Edit Flow Tools Reports Window Layout View Run Help
Quick Access
10 us
Flow Navigator
SIMULATION - Behavioral Simulation - Functional - sim_1 - test_mips32
PROJECT MANAGER
Settings
Add Sources
Language Templates
IP Catalog
IP INTEGRATOR
Create Block Design
Open Block Design
Generate Block Design
SIMULATION
Run Simulation
RTL ANALYSIS
Run Linter
Open Elaborated Design
Report Methodology
Report DRC
Schematic
SYNTHESIS
Run Synthesis
Open Synthesized Design
mips32.v
pipe_mips32_testbench.v
Untitled 3*
glbl.v
C:/Users/HP/Downloads/mips32basic/mips32basic.srcs/sources_1/new/mips32.v
117 default: EX_MEM_ALUOut <= #2 32'hxxxxxxxx;
118 endcase
119 end
120
121 LOAD, STORE: begin
122 EX_MEM_ALUOut <= #2 ID_EX_A + ID_EX_Imm;
123 EX_MEM_B <= #2 ID_EX_B;
124 end
125
126 BRANCH: begin
127 EX_MEM_ALUOut <= #2 ID_EX_NPC + ID_EX_Imm;
128 EX_MEM_cond <= #2 (ID_EX_A == 0);
129 end
130 endcase
131 end
132
133 always @(posedge clk2) // MEM Stage
134 if (HALTED == 0)
135 begin
136 MEM_WB_type <= EX_MEM_type;
137 MEM_WB_IR <= #2 EX_MEM_IR;
138 case (EX_MEM_type)
139 RR_ALU, RM_ALU:
140 MEM_WB_ALUOut <= #2 EX_MEM_ALUOut;
141 LOAD: MEM_WB_LMD <= #2 Mem[EX_MEM_ALUOut];
142 STORE: if (TAKEN_BRANCH == 0) // Disable write
143 Mem[EX_MEM_ALUOut] <= #2 EX_MEM_B;
144 endcase
145 end
146
147 always @(posedge clk1) // WB Stage
Tcl Console
Messages
Log

mips32basic - [C:/Users/HP/Downloads/mips32basic/mips32basic.xpr] - Vivado 2024.1

File Edit Flow Tools Reports Window Layout View Run Help

Flow Navigator

PROJECT MANAGER
- Settings
- Add Sources
- Language Templates
- IP Catalog

IP INTEGRATOR
- Create Block Design
- Open Block Design
- Generate Block Design

SIMULATION
- Run Simulation

RTL ANALYSIS
- Run Linter
- Open Elaborated Design
  - Report Methodology
  - Report DRC
  - Schematic

SYNTHESIS
- Run Synthesis
- Open Synthesized Design

SIMULATION - Behavioral Simulation - Functional - sim_1 - test_mips32

mips32.v | pipe_mips32_testbench.v | Untitled 3* | glbl.v

C:/Users/HP/Downloads/mips32basic/mips32basic.srcs/sources_1/new/mips32.v

```
    end

    always @(posedge clk2) // MEM Stage
    if (HALTED == 0)
    begin
    MEM_WB_type <= EX_MEM_type;
    MEM_WB_IR <= #2 EX_MEM_IR;
    case (EX_MEM_type)
    RR_ALU, RM_ALU:
    MEM_WB_ALUOut <= #2 EX_MEM_ALUOut;
    LOAD: MEM_WB_LMD <= #2 Mem[EX_MEM_ALUOut];
    STORE: if (TAKEN_BRANCH == 0) // Disable write
    Mem[EX_MEM_ALUOut] <= #2 EX_MEM_B;
    endcase
    end

    always @(posedge clk1) // WB Stage
    begin
    if (TAKEN_BRANCH == 0) // Disable write if branch taken
    case (MEM_WB_type)
    RR_ALU: Reg[MEM_WB_IR[15:11]] <= #2 MEM_WB_ALUOut; // "rd"
    RM_ALU: Reg[MEM_WB_IR[20:16]] <= #2 MEM_WB_ALUOut; // "rt"
    LOAD: Reg[MEM_WB_IR[20:16]] <= #2 MEM_WB_LMD; // "rt"
    HALT: HALTED <= #2 1'b1;
    endcase
    end



endmodule
```

Tcl Console | Messages | Log

POSSIBLE REASONS FOR FAILURE OF PIPELINE MODEL-

- ***Pipeline Hazards*** are situations that prevent the next instruction in the instruction stream from executing in its designated clock cycle
- Hazards reduce the performance from the ideal speedup gained by pipelining
- Three types of hazards
  - ***Structural hazards***
    - Arise from resource conflicts when the hardware can't support all possible combinations of overlapping instructions
  - ***Data hazards***
    - Arise when an instruction depends on the results of a previous instruction in a way that is exposed by overlapping of instruction in pipeline
  - ***Control hazards***
    - Arise from the pipelining of branches and other instructions that change the PC (Program Counter)

In the test benches we write for our model, we happen to encounter data hazards the most .
The cure to data hazards can be dummy instruction , due to which the data we want to use gets ready before execution of the next statement. The dummy instructions can be R5=R5||R5 , which gives R5.

We write two test benches to test our model which are given below---

- Test Bench 1-Add three numbers 10, 20 and 30 stored in processor registers.
- The steps:
  - Initialize register R1 with 10.
  - Initialize register R2 with 20.
  - Initialize register R3 with 30.
  - Add the three numbers and store the sum in R4.

| Assembly Language Program | Machine Code (in Binary) |
|---|---|
| ADDI R1,R0,10 | 01010 00000 00001 0000000000001010 |
| ADDI R2,R0,20 | 001010 00000 00010 0000000000010100 |
| ADDI R3,R0,25 | 001010 00000 00011 0000000000011001 |
| ADD R4,R1,R2 | 000000 00001 00010 00100 00000 000000 |
| ADD R5,R4,R3 | 000000 00100 00011 00101 00000 000000 |
| HLT | 111111 00000 00000 00000 00000 000000 |

For ease , we convert the following codes into hexadecimel and store them in mem[0] to mem[8], dummy instructions are also entered in between so that wrong data fetching does not happen.

# Verilog code for testbench1-

```verilog
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:IIT(ISM)
// Engineer: Harshit Raghuvanshi
//
// Create Date:
// Design Name:
// Module Name: pipe_mips32_testbench
// Project Name: testbench1_mips32
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
`include "pipe_MIPS32.v"
module test_mips32;
reg clk1, clk2;
integer k;
pipe_MIPS32 mips (clk1, clk2);
initial
begin
clk1 = 0; clk2 = 0;
repeat (20) // Generating two-phase clock
begin
#5 clk1 = 1; #5 clk1 = 0;
#5 clk2 = 1; #5 clk2 = 0;
end
end


initial
begin
for (k=0; k<31; k=k+1)
mips.Reg[k] = k;
mips.Mem[0] = 32'h2801000a; // ADDI R1,R0,10
mips.Mem[1] = 32'h28020014; // ADDI R2,R0,20
mips.Mem[2] = 32'h28030019; // ADDI R3,R0,25
```

File Edit Selection View Go Run Terminal Help

EXPLORER

OPEN EDITORS 2 unsaved

- Welcome
- pipe_MIPS32.v v...
- test_mips32.v ve...

VSC

- verilog
  - build
  - pipe_MIPS32.v
  - test_mips32.v

OUTLINE

TIMELINE

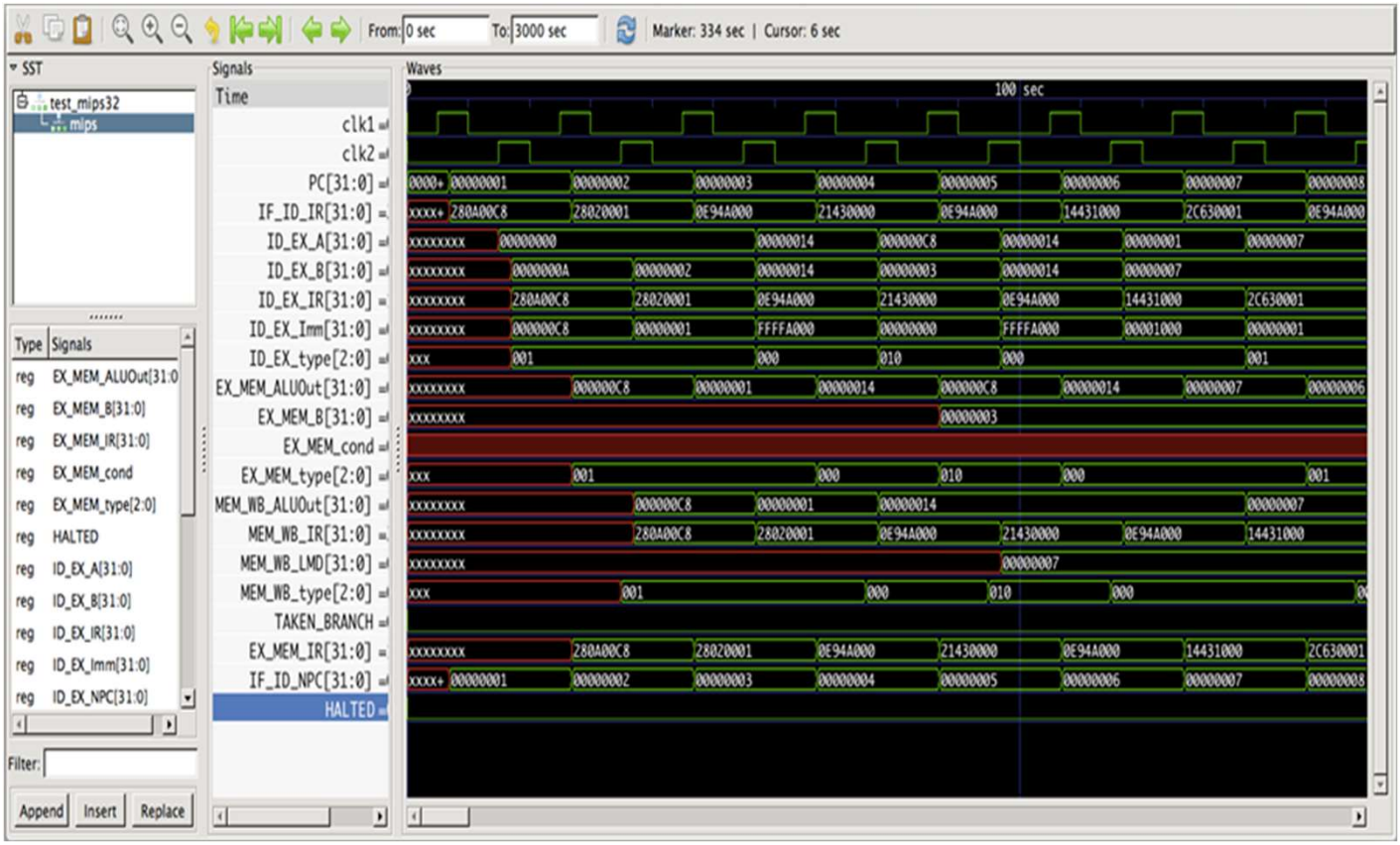
Welcome | pipe_MIPS32.v | test_mips32.v

verilog > test_mips32.v

```
`include "pipe_MIPS32.v"

initial
begin
for (k=0; k<31; k=k+1)
mips.Reg[k] = k;
mips.Mem[0] = 32'h2801000a; // ADDI R1,R0,10
mips.Mem[1] = 32'h28020014; // ADDI R2,R0,20
mips.Mem[2] = 32'h28030019; // ADDI R3,R0,25
mips.Mem[3] = 32'h0ce77800; // OR R7,R7,R7 -- dummy instr.
mips.Mem[4] = 32'h0ce77800; // OR R7,R7,R7 -- dummy instr.
mips.Mem[5] = 32'h00222000; // ADD R4,R1,R2
mips.Mem[6] = 32'h0ce77800; // OR R7,R7,R7 -- dummy instr.
mips.Mem[7] = 32'h00832800; // ADD R5,R4,R3
mips.Mem[8] = 32'hfc000000; // HLT


mips.HALTED = 0;
mips.PC = 0;
mips.TAKEN_BRANCH = 0;
#280
for (k=0; k<6; k=k+1)
$display ("R%1d - %2d", k, mips.Reg[k]);
end
initial
begin
$dumpfile ("mips.vcd");
$dumpvars (0, test_mips32);
#300 $finish;
end
endmodule
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
```

⊗0 ⚠0 0

# Simulation of test bench 1 on gtkwave --



Testbench 2-

Load a word stored in memory location 120, add 45 to it, and store the result in memory location 121.

- The steps:
- – Initialize register R1 with the memory address 120.
- – Load the contents of memory location 120 into register R2.
- – Add 45 to register R2.
- – Store the result in memory location 121

| Assembly Language Program | Machine Code (in Binary) |
|---|---|
| ADDI R1,R0,120 | 001010 00000 00001 0000000001111000 |
| LW R2,0(R1) | 001000 00001 00010 0000000000000000 |
| ADDI R2,R2,45 | 001010 00010 00010 0000000000101101 |
| SW R2,1(R1) | 001001 00010 00001 0000000000000001 |
| HLT | 111111 00000 00000 00000 00000 000000 |

Verilog code for testbench2-

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company: IIT(ISM)
// Engineer: Harshit Raghuvanshi
//
// Create Date:
// Design Name:
// Module Name: pipe_mips32_testbench
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
module test_mips32;
 reg clk1, clk2;
 integer k;
 pipe_MIPS32 mips (clk1, clk2);
 initial
 begin
 clk1 = 0; clk2 = 0;
 repeat (50) // Generating two-phase clock
 begin
 #5 clk1 = 1; #5 clk1 = 0;
 #5 clk2 = 1; #5 clk2 = 0;
 end
 end


initial
 begin
 for (k=0; k<31; k++)
 mips.Reg[k] = k;
 mips.Mem[0] = 32'h28010078; // ADDI R1,R0,120
 mips.Mem[1] = 32'h0c631800; // OR R3,R3,R3 -- dummy instr.
 mips.Mem[2] = 32'h20220000; // LW R2,0(R1)
 mips.Mem[3] = 32'h0c631800; // OR R3,R3,R3 -- dummy instr.
```

EXPLORER

OPEN EDITORS 2 unsaved

- Welcome
- pipe_MIPS32.v v...
- test_mips32.v ve...

VSC

- verilog
  - build
  - pipe_MIPS32.v
  - test_mips32.v

Welcome | pipe_MIPS32.v | test_mips32.v

verilog > test_mips32.v

```
module test_mips32;
 end


initial
 begin
 for (k=0; k<31; k++)
 mips.Reg[k] = k;
 mips.Mem[0] = 32'h28010078; // ADDI R1,R0,120
 mips.Mem[1] = 32'h0c631800; // OR R3,R3,R3 -- dummy instr.
 mips.Mem[2] = 32'h20220000; // LW R2,0(R1)
 mips.Mem[3] = 32'h0c631800; // OR R3,R3,R3 -- dummy instr.
 mips.Mem[4] = 32'h2842002d; // ADDI R2,R2,45
 mips.Mem[5] = 32'h0c631800; // OR R3,R3,R3 -- dummy instr.
 mips.Mem[6] = 32'h24220001; // SW R2,1(R1)
 mips.Mem[7] = 32'hfc000000; // HLT
 mips.Mem[120] = 85;
mips.PC = 0;
 mips.HALTED = 0;
 mips.TAKEN_BRANCH = 0;
 #500 $display ("Mem[120]: %4d \nMem[121]: %4d",
 mips.Mem[120], mips.Mem[121]);
 end
 initial
 begin
 $dumpfile ("mips.vcd");
 $dumpvars (0, test_mips32);
 #600 $finish;
 end
endmodule
```

OUTLINE

TIMELINE

Result of simulation on gtkwave-