Pipeline Implementation of RISC architecture processors: MIPS32

MIPS 32.

a) 32- general purpose registers R0 to R31 (32-bit each)
b) Program Counter register PC -32-bit.
No flag registers & Only few addressing modes
every word is memory addressable and 32-bit word size.

Below are the instructions considered.

Load & store instructions

```
LW    R2,124(R8)      // R2 = Mem[R8+R4]
SW    R5,-10(R25)     // Mem[R5-10] = R5
```

Arithematic & logic instructions

```
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
```

SLT => Set Less Than Instruction

```
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.
```

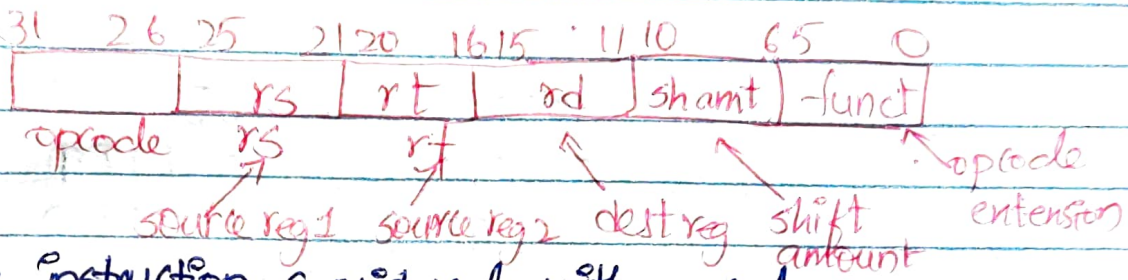
MIPS 32 instruction encoding : 3 types

R-type (Register type)

I-type (Immediate type)

J-type (Jump type)

R-type:



R-type instruction considered with opcode

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

example for opcode instruction with 32-bit registers

SUB R5, R12, R25

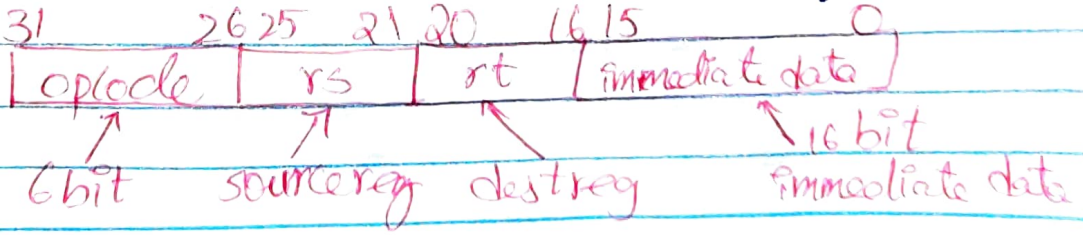
000001 01100, 11001, 00101, 00000 000000

= 05992800 (in hex).

↳ Broke above instruction into hexadecimal.

I-type: Contains a 16-bit immediate data field.

has one source & one destination field.



I-type instructions with opcode

Instruction	Opcode	
LW	001000	//Load
SW	001001	//Store
ADDI	001010	
SUBI	001011	
SLTI	001100	
BNEQZ	001101	
BEQZ	001110	

example for immediate instruction

LW R20, 84(R9)

001000 01001, 10100,0000000101010100 [84 in binary]

= 8B400054 (in hexadecimal) missing fields

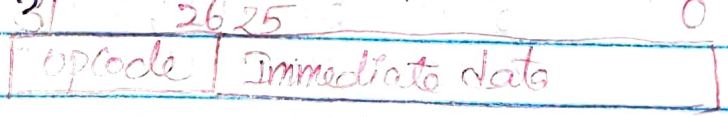
example2:

BEQZ R25, label

001110 11001, 00000, YYYYYYYYYYYYYYYY

= 3B20YYYY (in hexadecimal)

J-type: Contains 26-bit jump address



Instruction	Opcode
J	010000

** Jump instruction is not implemented for RISC processor

Instruction cycle: time taken to execute one complete instruction

MIPS 32 instruction cycle :- 5 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

a) Instruction Fetch:

PC contains next instruction address in memory
→ Every MIPS 32 instruction is 32 bit
→ Every memory word is 32 bits and unique

IF: IR ← Mem[PC]; // instruction register
NPC ← PC+; // New PC (Prg Counter)

b) Instruction Decode:

Instruction Fetched in IR is decoded in this stage
Opcode :- 6 bits (31:26)
first source operand rs (25:21), second source operand rt (20:16); 16-bit immediate data (15:0); 26-bit immediate data (25:0)

*Decoding is done in parallel with reading register operands

rs & rt ID: A ← Reg[rs]
B ← Reg[rt]
Imm ← $(IR)_{15}^{16}$ ## $IR_{15..0}$
Imm1 ← $(IR)_{25}^{6}$ ## $IR_{25..0}$

16-bit immediate field is signed number in 2's complement
sign extension is to extend 16 bit to 32-bit
easy in 2's complement as it is to pad sign bit to the left (MSBs) until 32-bits

c) Execution/Effective address allocation

ALU for some calculations

ALU inputs are from stage (b) (A, B, Imm, etc)

examples:

Memory reference: ALUOut ← A + Imm

ex: LW R3, 100(R8)

Reg-Reg ALU : ALUOut ← A func B

ex: SUB R2, R5, R12

Reg-Imm ALU : ALUOut ← A func Imm

ex : SUBI R2, R5, 524

Branch : ALUOut = NPC + Imm; Cond ← (A op 0);

ex: BEQZ R2, Label

d) Memory Access/Branch Completion.

Only load, stores, use this step

Load & Store access memory

Branch Updates PC depending upon outcome

Load instruction:

PC ← NPC;

LMD ← Mem[ALUout];

store instruction:

PC ← NPC;

Mem[ALUOut] ← B;

Branch instruction:

if (cond) PC ← ALUout;

else PC ← NPC;

Others:

PC ← NPC;

e) Write Back to Register (WB)

result is written back to register

↳ may come from ALU

↳ may come from memory system via LOAD instru

position of destination register varies wrt type of instruction (R-type / I-type)

for Reg-Reg

Reg[rd] ← ALUOut

Reg-Imm

Reg[rt] ← ALUOut

Load-Instr

Reg[rt] ← LMD;

Basic example for instruction execution step-by-step

ex: ADD R2, R5, R10 // Register Instruction example

IF: IR ← Mem[PC];

NPC ← PC+1;

ID: A ← Reg[RS]

B ← Reg[rt]

EX: ALUOut ← A+B;

Mem: PC ← NPC;

WB: Reg[rd] ← ALUOut;

// immediate instruction example

ex: ADDI R2, R5, 150

IF: IR ← Mem[PC];

NPC ← PC+1;

ID: A ← Reg[rs];

Imm ← $(IR)_{15}^{16}$ ## $IR_{15..0}$

Ex: ALUOut ← A + Imm;

Mem: PC ← NPC;

WB : Reg[rt] ← ALUOut ;

// Load instruction example.

ex: LW R2, 200(R6)

IF : IR ← Mem[PC];
NPC ← PC+1;

ID : A ← Reg[rs]
Imm ← $(IR_{15})^{16}$ ## $IR_{15...0}$

EX : ALUOut ← A + Imm

Mem : PC ← NPC;
LMD ← Mem[ALUOut]

WB : Reg[rt] ← LMD;

// Store Instruction example

ex: SW R3, 25(R10)

IF : IR ← Mem[PC];
NPC ← PC+1;

ID : A ← Reg[rs] ; B ← Reg[rt];
Imm ← $(IR_{15})^{16}$ ## $IR_{15...0}$

EX : ALUOUT = A+Imm;

Mem : PC ← NPC;
Mem[ALUOut] ← B;

WB : —

ex: BEQZ R3, Label. // Branch instruction example

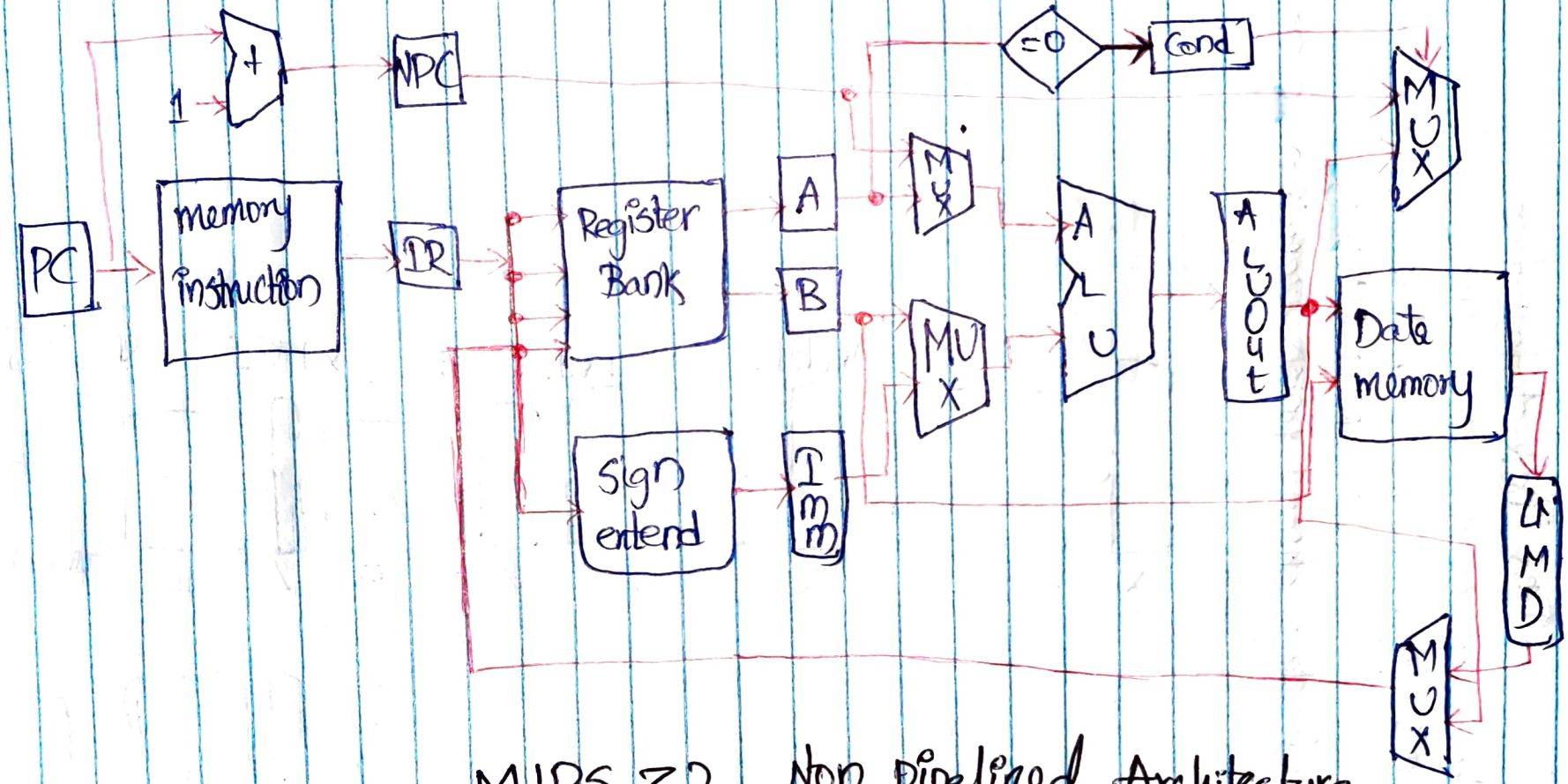
IF : IR ← Mem[PC];
NPC ← PC+1

ID : A ← Reg[rs]
Imm ← $(IR_{15})^{16}$ ... ## .. $IR_{15...0}$

EX : ALUOut ← NPC + Imm
Cond ← (A == 0)

Mem : PC ← NPC; if (Cond) PC ← ALUOut

WB : —

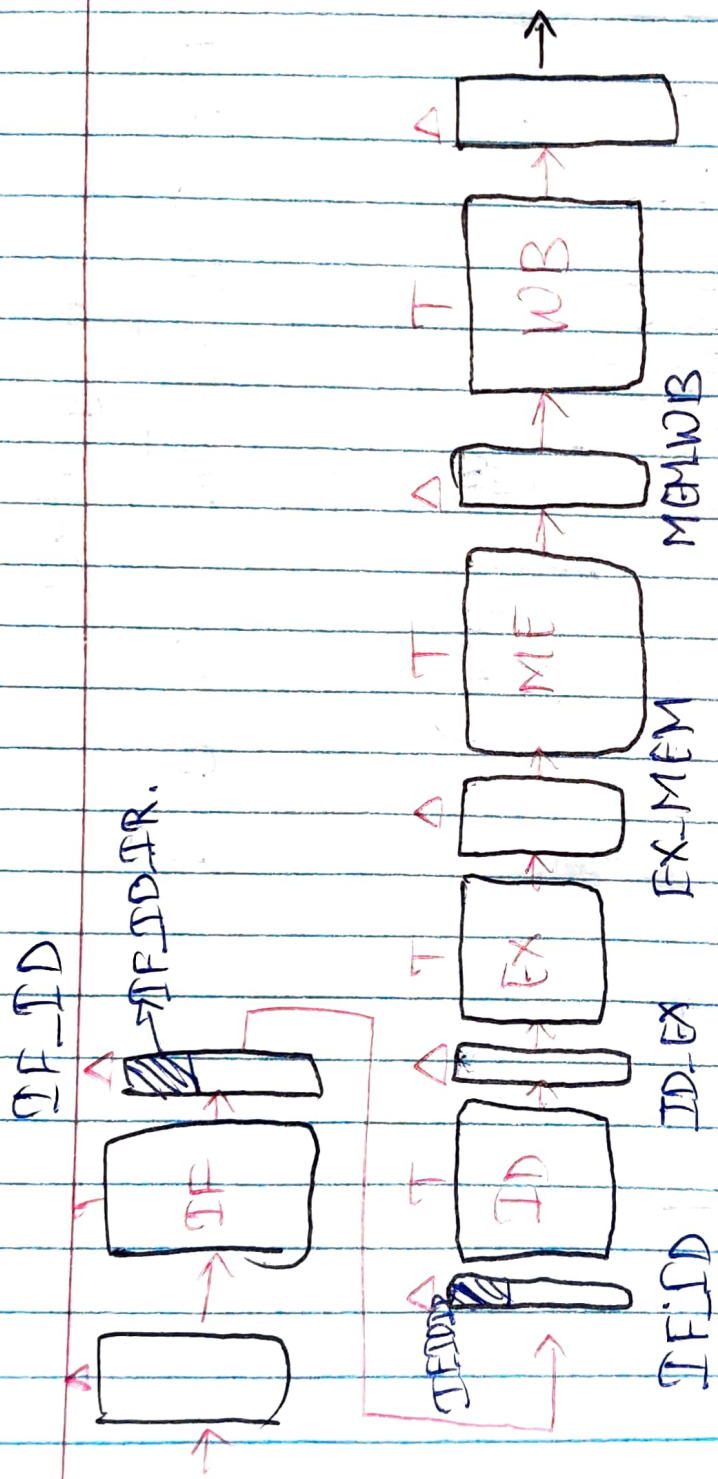


MIPS 32 Non pipelined Architecture

(DATA PATH)

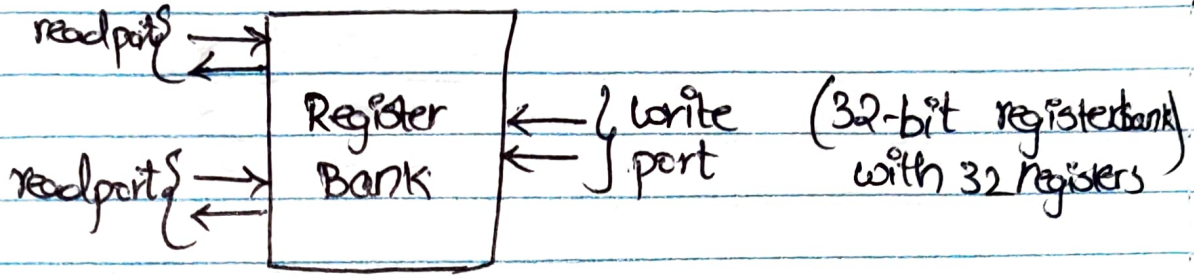
Pipelined architecture:

→ New instructions every clock cycle
→ Each stage mentioned before becomes pipeline stage (IF, ID, EX, MEM, WB)
→ Each stage should finish execution in one clock cycle.



clock cycles of pipelined architecture

Inst	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



Micro operations of MIPS 32 architecture.

IF_ID : Latch between IF & ID

ID_EX : Latch between ID & EX

EX_MEM : Latch between EX & MEM

MEM_WB: Latch between MEM & WB

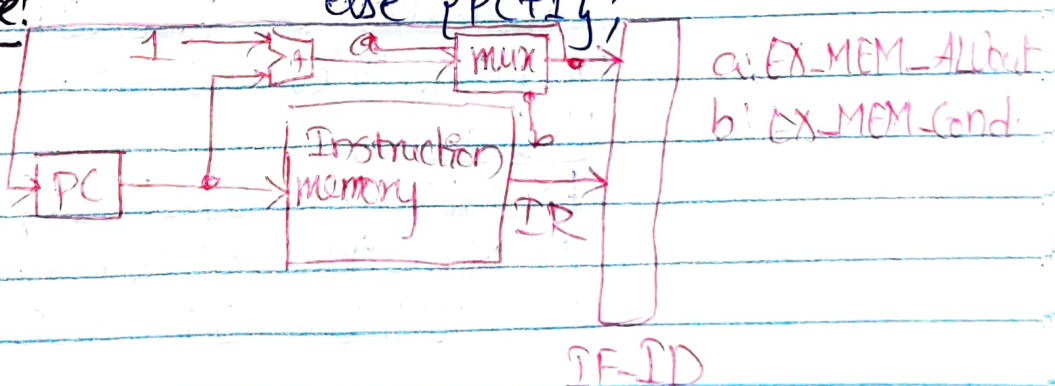
Now using above naming conventions we name the processor micro functions as below:

a) Instruction Fetch:

IF_ID_IR ← Mem[PC];

IF_ID_NPC, PC ← (if ((EX_MEM.IR[OPCODE] == branch) & (EX_MEM.cond)) : {EX_MEM.ALUout} else {PC+1};

hardware:



b) Instruction Decode stage:

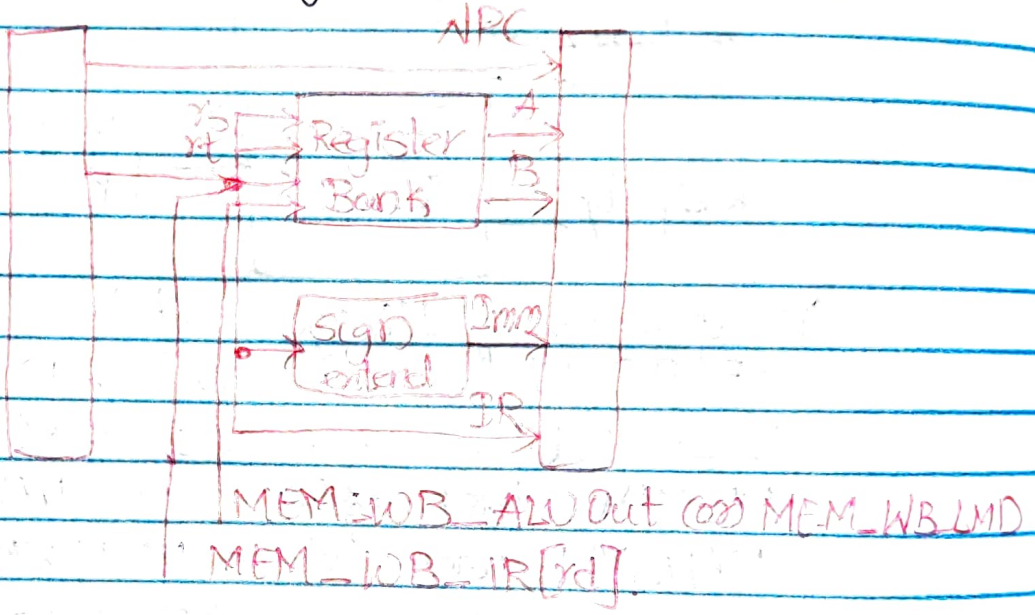
ID_EX_A ← Reg [IF_ID_IR[rs]]

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) Execution stage

RR ALU

EX_MEM_IR ← ID_EX_IR;

EX_MEM_ALUOut ← ID_EX_A func ID_EX_B

R-I ALU

EX_MEM_IR ← ID_EX_IR;

EX_MEM_ALUOut ← ID_EX_A func ID_EX_Imm

LD/SW

EX_MEM_IR ← ID_EX_IR;

EX_MEM_ALUOut ← ID_EX_A + ID_EX_Imm;

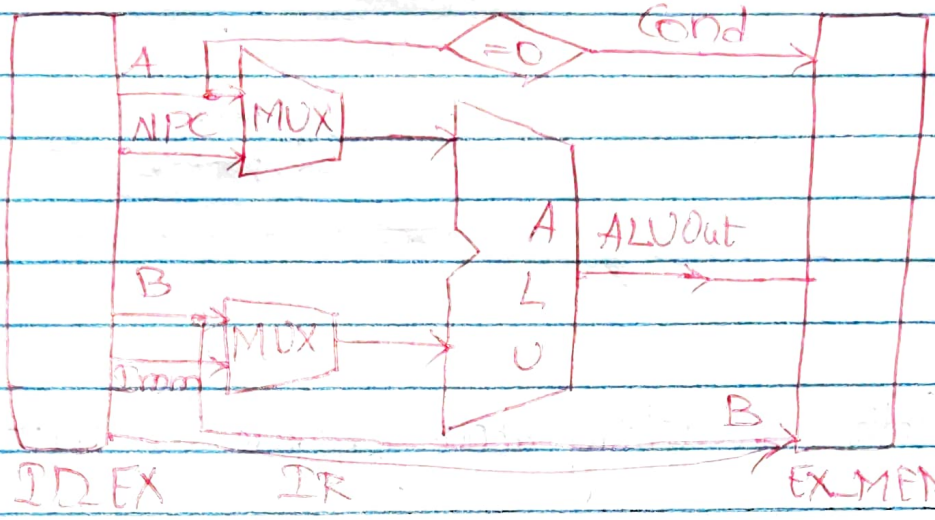
EX_MEM_B ← ID_EX_B;

~~EX_MEM (ALU~~

Branch:

EX_MEM_ALUOut ← ID_EX_NPC+ID_EX_Imm;

EX_MEM_Cond ← (ID_EX_A == 0);

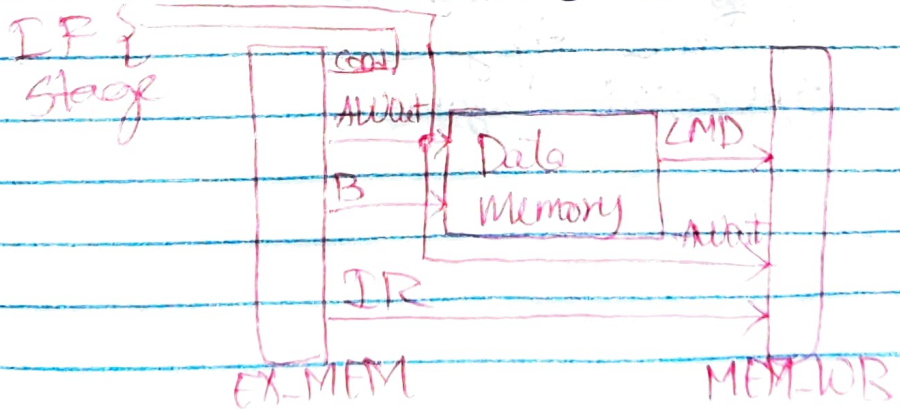


d) Memory Access stage:

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

MEM_WB_IR ← EX_MEM_IR;
MEM[EX_MEM_ALUOut] ← EX_MEM_B; } store.

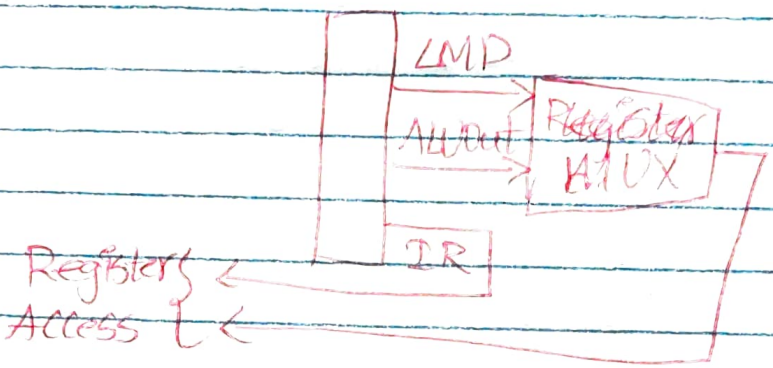


e) Register Write Back Stage:

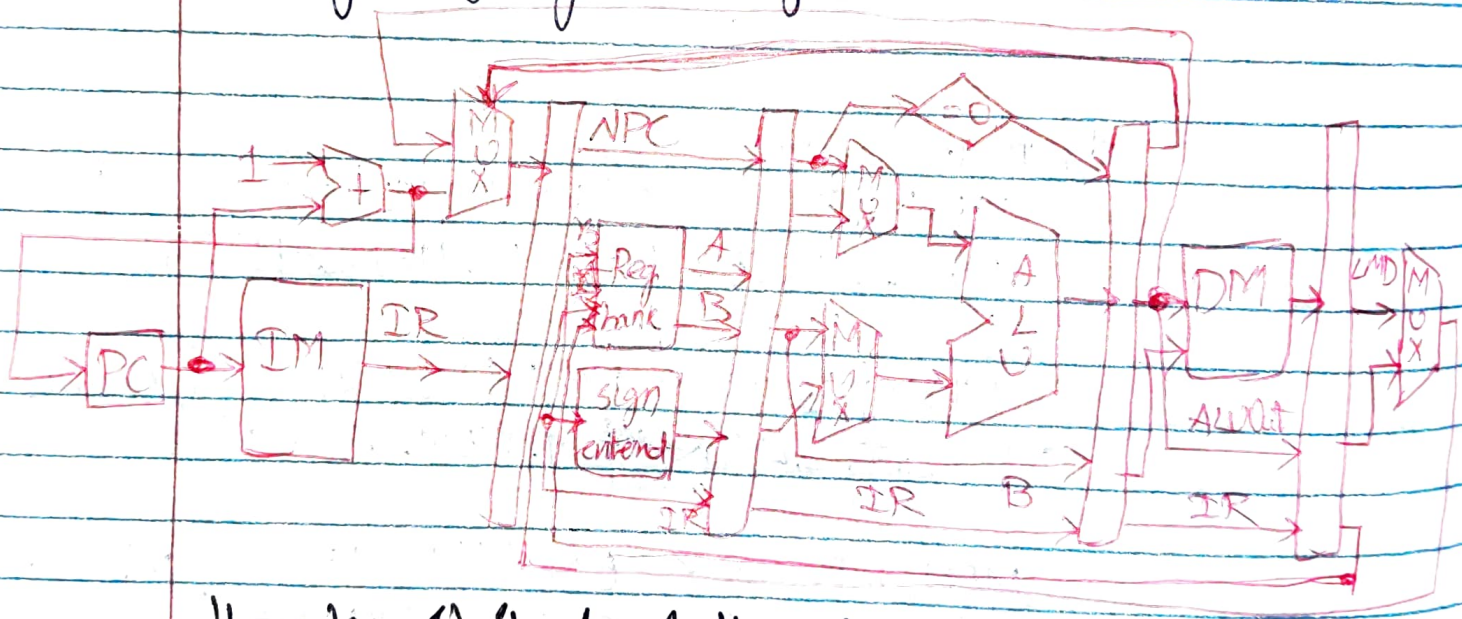
Reg[MEM_WB_IR[rd]] ← MEM_WB_ALUOut; R-R

Reg[MEM_WB_IR[rt]] ← MEM_WB_ALUOut; R-I

Reg[MEM_WB_IR[rt]] ← MEM_WB_LMD; Load.



Putting everything above together we have as below.



Hazards: 1) Structural Hazards
2) Data Hazards.
3) Control Hazards.

Verilog modelling of MIPS 32 processor.

When a halt instruction is encountered we need to set a flag to stop other instructions from entering pipeline. This is Halted flipflop.

When a branch instruction is encountered we need to consider branching for which we need a flip flop to discard that instruction. This flipflop is called ~~Branch~~ taken_Branch.

```
module pipe_mips32(clk1, clk2);
    input clk1, clk2;
    reg [31:0] PC, IF_ID_IR, IF_ID_NPC;
    reg [31:0] ID_EX_IR, ID_EX_NPC, ID_EX_A, ID_EX_B,
               ID_EX_Imm;
    reg [2:0] ID_EX_type, EX_MEM_type, MEM_WB_type;
    reg [31:0] EX_MEM_IR, EX_MEM_ALUOut, EX_MEM_B;
    reg        EX_MEM_Cond;
    reg [31:0] MEM_WB_IR, MEM_WB_ALUOut, MEM_WB_LMD;
    reg [31:0] Reg[0:31]      // register bank
    reg [31:0] Mem[0:1023]    // memory 1024x32

    parameter ADD = 6'b000000, SUB = 6'b000001,
    AND = 6'b000010, OR = 6'b000011, SLT = 6'b000100,
    MUL = 6'b000101, HLT = 6'b111111, LW = 6'b001000,
    SW = 6'b001001, ADDI = 6'b001010, SUBI = 6'b001011,
    SLTI = 6'b001100, BNEQ = 6'b001101, BEQZ = 6'b001110;

    parameter RR_ALU = 3'b000, RM_ALU = 3'b001, LOAD = 3'b010,
      STORE = 3'b011, BRANCH = 3'b100, HALT = 3'b101;
```

```
reg HALTED;
reg TAKEN_BRANCH;

//IF stage
always@(posedge CLK1)
    if(HALTED==0)
    begin
        if(((EX_MEM_IR[31:26]== BEQZ) && (EX_MEM_cond
            ==1)) || ((EX_MEM_IR[31:26]==BNEQZ) &&
                (EX_MEM_cond==0)))
        begin
            IF_ID_IR<=#2 Mem[EX_MEM_ALUOut];
            TAKEN_BRANCH=#2 1'b1;
            IF_ID_NPC<= #2 EX_MEM_ALUOUT;
            PC <= #2 EX_MEM_ALUOUT;
        end
    else
    begin
        IF_ID_IR <=#2 Mem[PC];
        IF_ID_NPC <=#2 PC+1;
        PC <= #2 PC+1;
    end
end

//ID stage
always@(posedge CLK2)
    if(HALTED==0)
    begin
```

```
If(IF_ID_IR[25:21] == 5'b00000)
        ID_EX_A <= 0;
else
    ID_EX_A <= #2 Reg [IF_ID_IR[25:21]];
if (IF_ID_IR[20:16] == 5'b00000)
        ID_EX_B = 0;
else
    ID_EX_B <= #2 Reg [IF_ID_IR[20:16]];

ID_EX_NPC <= #2 IF_ID_NPC;
ID_EX_IR <= #2 IF_ID_IR;
ID_EX_Imm <= #2 {{16{IF_IR_IR[15]}},IF_ID_IR
                [15:0]};
Case (IF_ID_IR[31:26])
    ADD,SUB,AND,OR,SLT,MUL: ID_EX_type<=#2 RR_ALU;
    ADI,SUBI,SLTI: ID_EX_type <= #2 RM_ALU;
    LW:  ID_EX_type <= #2 LOAD;
    SW:  ID_EX_type <= #2 STORE;
    BEQZ,BNEQZ: ID_EX_type <= #2 BRANCH;
    HLT: ID_EX_type <= #2 HALT;
    Default: ID_EX_type <= #2 HALT;
endCase
end

// EX Stage
always @(posedge clk1)
  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])
           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_B * ID_EX_B;
          default: EX_MEM_ALUOut <= #2 32'hxxxxxxxx;
         endcase
        end
 RM_ALU: begin
          Case (ID_EX_IR[31:26])
          ADDI: EX_MEM_ALUOut <= #2 ID_EX_A + ID_EX_Imm;
         SUBI: EX_MEM_ALUOut <= #2 ID_EX_A - ID_EX_Imm;
         SLTI: EX_MEM_ALUOut <= #2 ID_EX_A < ID_EX_Imm;
        default: EX_MEM_ALUOut <= #2 32'hxxxxxxxx;
        endcase
       end
LOAD, STORE: begin
          EX_MEM_ALUOut <= #2 ID_EX_A + ID_EX_Imm;
          EX_MEM_B <= #2 ID_EX_B;
         end
```

```
        BRANCH: begin
                    EX_MEM_ALUOut <= #2 ID_EX_NPC+ID_EX_Imm;
                    EX_MEM_Cond <= #2 (ID_EX_A==0)
                end
    endcase
end
//MEM stage
always @(posedge clk2)
  If (HALTED ==0)
     begin
     MEM_WB_Type <= EX_MEM_type;
     MEM_WB_IR   <= 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)
                        Mem[EX_MEM_ALUOut] <= #2 EX_MEM_B;
     endcase
end

//WB Stage
always @(posedge clk1)
  begin
     if (TAKEN_BRANCH==0)
       case (MEM_WB_type)
          RR_ALU: Reg[MEM_WB_IR[15:11]] <= #2 MEM_WB_ALUOut;
          RM_ALU: Reg[MEM_WB_IR[20:16]] <= #2 MEM_WB_ALUOut;
          LOAD : Reg[MEM_WB_IR[20:16]] <= #2 MEM_WB_LMD;
```

```
        HALT : HALTED <= #2 1'b1;
    endcase
  end
endmodule
```

Using the above module described, execute programs to test functionality : Examples as below: (think & extend further)

ex:1: ADD three numbers and store in processor registers ie set R1=10; R2=20, R3=30 & store sum in R4;

```
module mips32test;
    reg clk1, clk2;
    integer k;
    pipe_mips32 mips(.clk1(clk1), .clk2(clk2));
    initial begin
        clk1=0; clk2=0;
        repeat(20)
          #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;
            // Load all memory locations
            // with the hexcode of the
            // program to add three numbers
            // and store results. Add dummy
            // instructions to avoid data hazards.
```

```
        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, mips32test);
      #300 $finish;
      end
  endmodule
```

ex2: load a word stored in memory location 120, add 45 to it and store result in memory location 121

ex3 Calculate fatorial of Number with loop instruction.

** The testbench inputs are the Hexcodes of each program. Remaining parts are the same vary the Hexcodes and observe the outputs with different examples.

As a whole, the high level programming languages are converted to lower levels and fed to processors and they are infact testbenches.

x