Lec-38

MIPS32 Instruction cycle

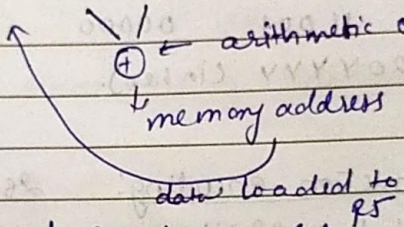
Instruction execution cycle — 5 stages

(i) instruction fetch (IF): fetching instruction from memory

(ii) ID: Instruction Decode/ Register fetch: decode opcode & find what instruction it is.

(iii) EX: Execution/ Effective Address calculation: we execute instruction or for some instruction, we have to compute effective address size.

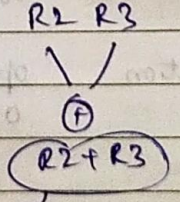
eg LD R5, 56(R12)




(iv) MEM: memory access/ Branch completion — read or write

(v) WB: register write-back

eg ADD R1 R2 R3




Now this result of R2+R3 register values needs to be stored in R1.
This happens in Register write back stage.

(in these steps, there are various steps → they are micro-operations.

PC – Program counter

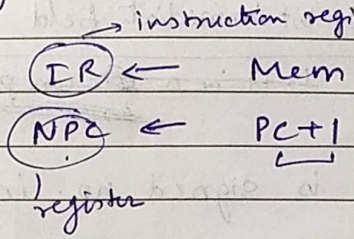
(a) Instruction fetch (IF):

Here the instruction pointed to by PC is fetched from memory & also next value of PC is computed.

* PC always contains the address of next instruction to be executed.

– 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.

eg: IF:




we update PC but do not store it back in PC because for jump instruction, the target address can be something else. (every byte of memory has unique address)

* for byte addressable memory, PC has to be incremented by 4 bytes (32 bits).

(b) Instruction Decode (ID):

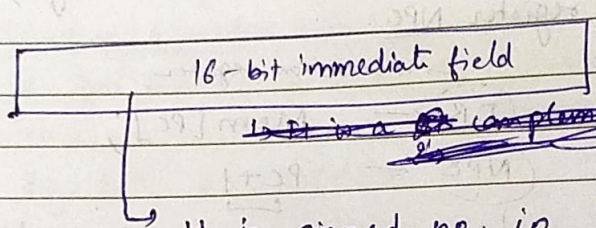
• instruction already fetched in IR is decoded.
– opcode is 6-bits (bits 31:26)
– first source operand rs (bits 25:21), second source operand rt (bits 20:16).
– 16 bit immediate data (bits 15:0)
– 26-bit immediate data (bits 25:0)

• 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.

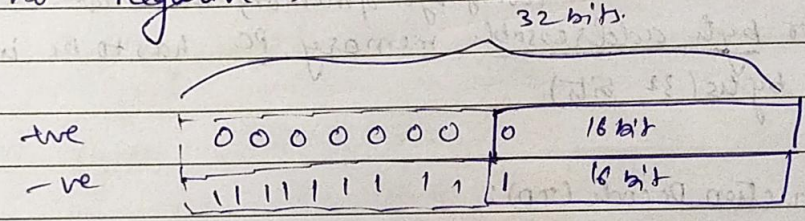
A, B, Imm, Imm1 → temp. registers.

ID: A ← Reg [rs]
B ← Reg [rt]
Imm ← $(IR_{15})^{16}$ ## $IR_{15...0}$ // sign extension of 16-bit immediate field.
Imm1 ← $(IR_{25})^{6}$ ## $IR_{25...0}$ // sign extension of 26-bit immediate field.

What is actually happening




if no. positive → starts with 0.
if no. negative → starts with 1.




if no. is +ve → pad zeroes to the left

(c) EX: Execution / Effective address computation

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

eg LW R3, 100(R8). (memory reference):

ALUout ← A + Imm

eg: SUB R2, R5, R12

ALUout ← A func B (reg-reg ALU instruct).

(Register - Immediate ALU instructions)

eg: SUBI R2, R5, #24.

ALUout ← A func Imm

(Branch)

eg BEQZ R2, label

ALUout ← NPC + Imm; (op is ==)

Cond ← (A op 0);

LMD → load memory data
↳ it is just a temporary register.

(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 instructions update PC depending upon the outcome of the branch condition.

Load instruc
PC ← NPC;
LMD ← Mem[ALUout];

Store instruction
PC ← NPC;
Mem[ALUout] ← B;

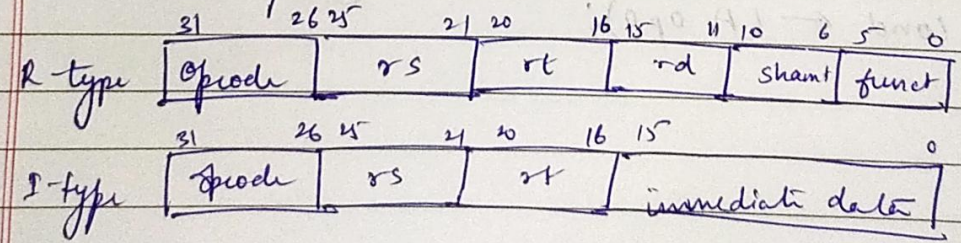
Branch instruction :
if (cond) PC ← ALUout;
else PC ← NPC;

Others instructions
PC ← NPC

(e) WB : Register Writ Back

→ in this step, result → written back to register.
(Result may come from ALU)
(Result may come from the memory system (via LOAD instruction).

→ the position of destination register in the instruction word depends on 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 ;

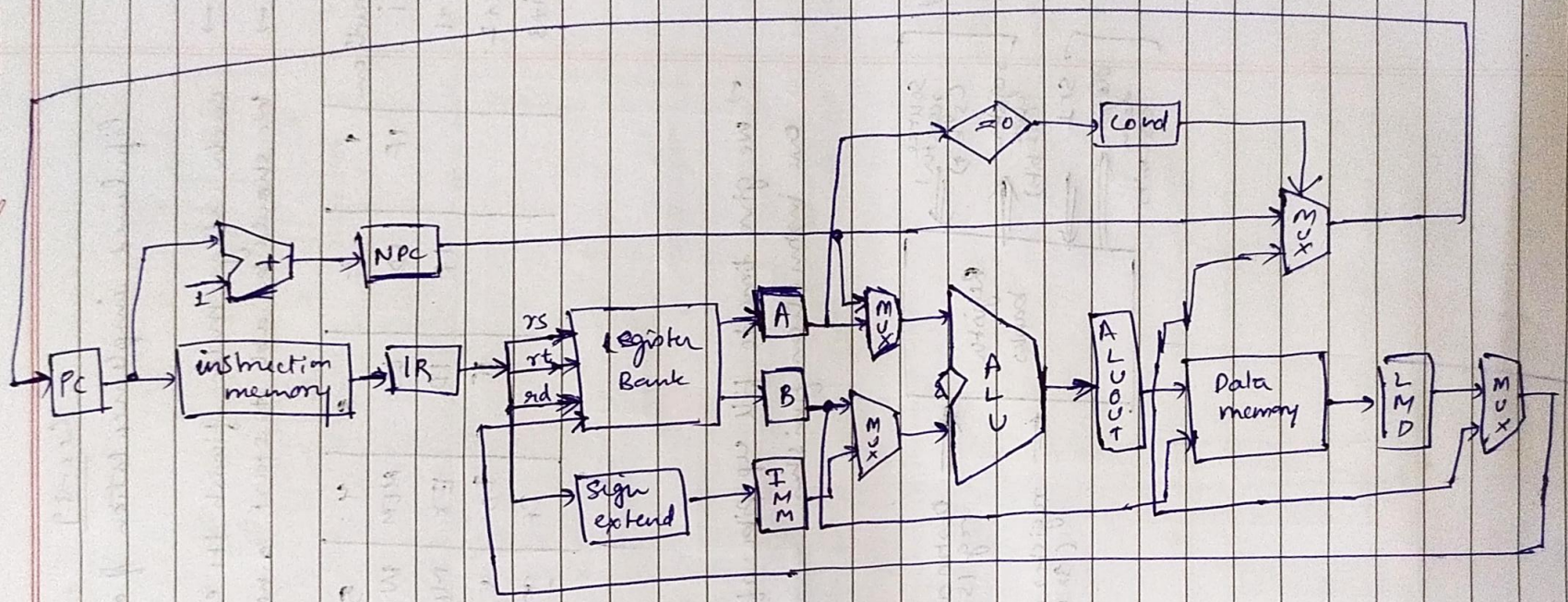
Complete steps of micro operations

eg: ADD R2, R5, R10

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

eg: 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 ; |




MIPS 32

Non-pipelined datapath

Lec-39

Pipelined implementation of a processor (3).

→ each stage must finish its execution every clock cycle

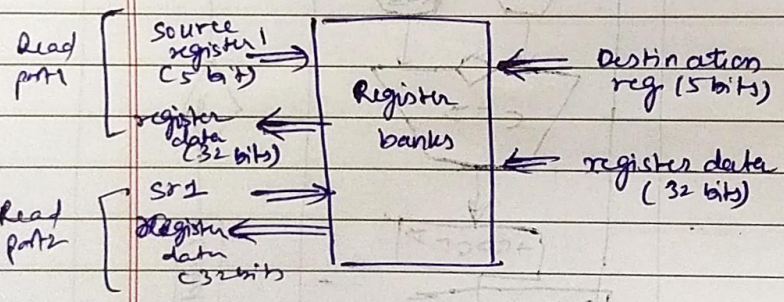
→ We should be able to start a new instruction every clock cycle

| 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 |

instruction i finishes at 5

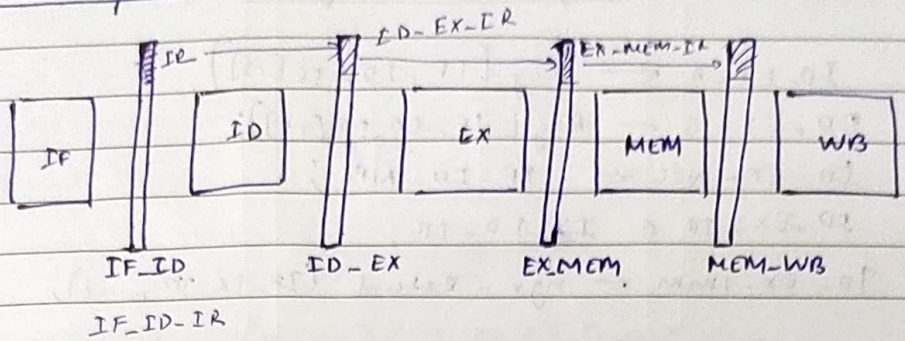
instruction i+1 finishes at 6

→ We give time to fill up pipe. after that every clock cycle one instruction finishes.



classmate Date Page

Some naming conventions



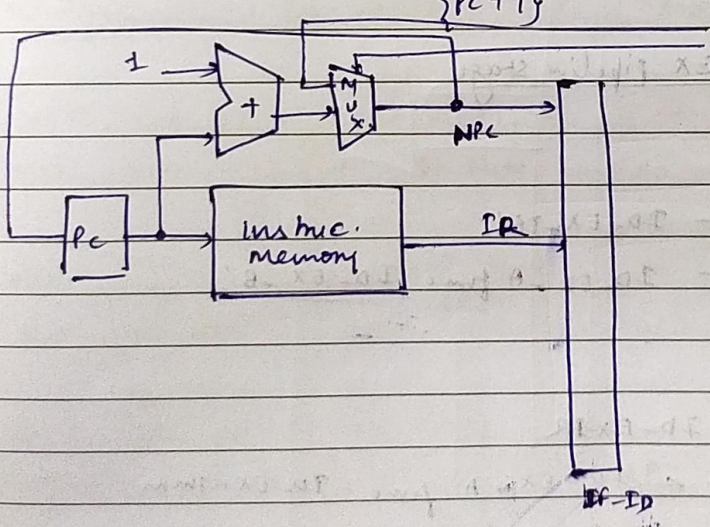

→ most of the temporary regs. required in data path are included as part of inter stage latches.

eg IR ← Mem [PC]

(a) PIPELINE STAGE - IR

IF_ID_IR ← Mem [PC];

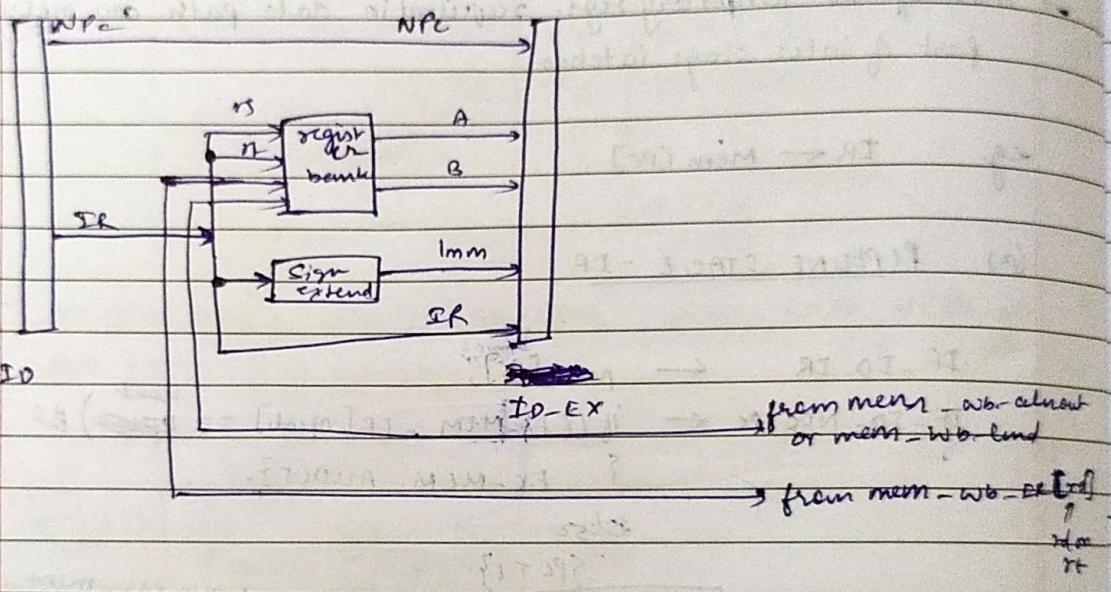
IF_ID_NPC; PC ← if ((EX_MEM_IR[opcode] == ~~opcode~~ branch) && EX_MEM_cond)
{ EX_MEM_ALUOUT };
else
{PC + 1}




IF

(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 op. for EX pipeline stage

→ R-R ALU

EX_MEM_IR ← ID_EX_IR;
EX_MEM_ALUOut ← ID_EX_A func ID_EX_B;

→ R_M ALU

EX_MEM_IR ← ID_EX_IR
EX_MEM_ALUOut ← ID_EX_A func ID_EX_Imm

→ Load/Store :

EX-MEM-IR ← ID-EX-IR

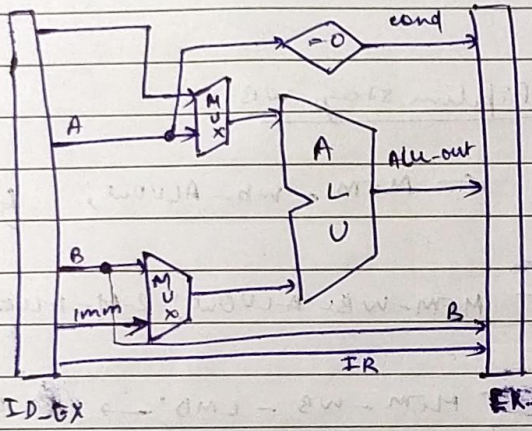
EX-MEM-ALUout ← ID-EX-A + ID-EX-Imm

EX-MEM-B ← ID-EX-B;

→ Branch :

EX-MEM-ALUout ← ID-EX-NPC + ID-EX-Imm

EX-MEM-cond ← (ID-EX-A == 0);

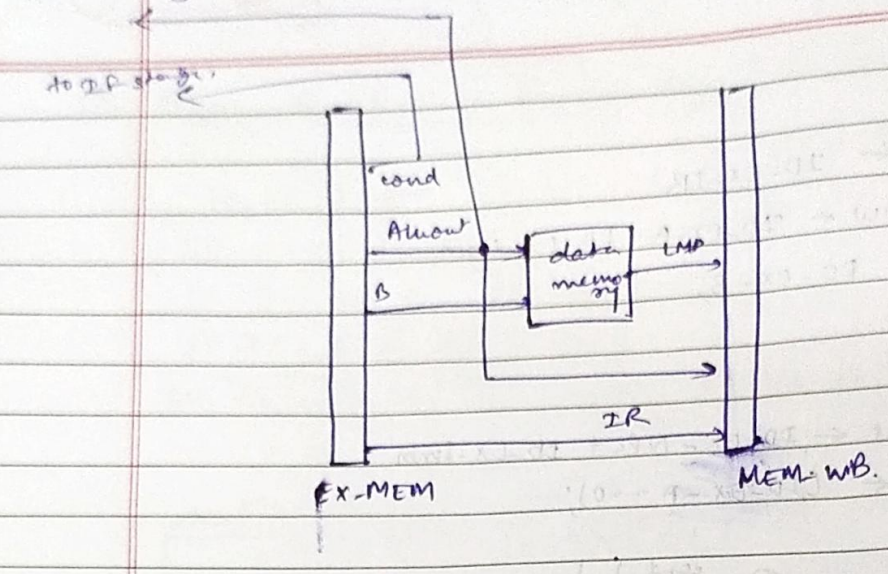



(d) Micro operations for pipeline stage MEM

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-ALU-out] ; } Load

MEM-WB-IR ← EX-MEM-IR
MEM[EX-MEM-ALU-out] ← EX-MEMB ; } Store

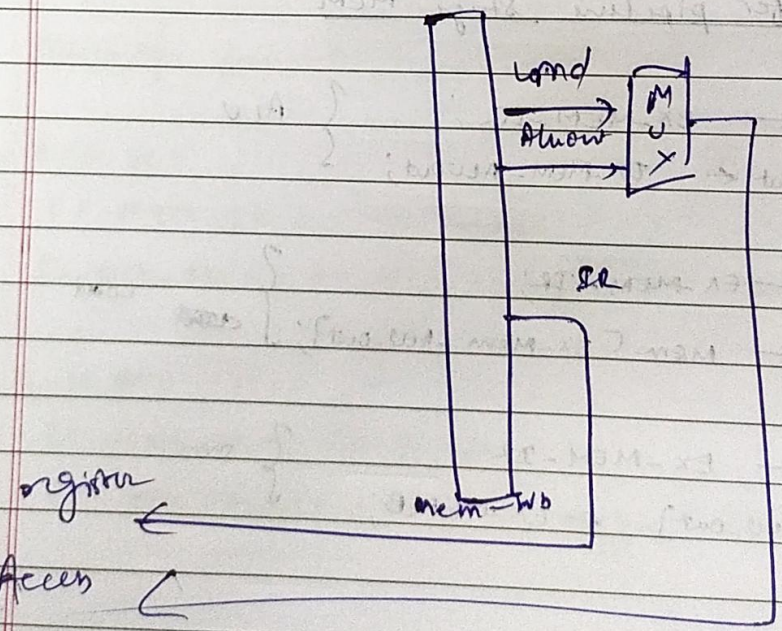



(c) Micro operations for Pipeline stage WB

Reg [MEM-WB-IR[rd]] ← MEM-WB-ALUOut; for R-RALU

Reg [MEM-WB-IR[rt]] ← MEM-WB-ALUOut; R-M-ALU

Reg [MEM-WB-IR[rt]] ← MEM-WB-LMD; → LOAD



After combining all → MIPS 32

HAZARDS in pipelines:

(a) Structural hazard: two instructions which are already there in the pipeline, they are in two different stages, but they are trying to use same hardware resource

(b) Data Hazard → arises due to instruction dependency

IF ID EX MEM WB

IF ID ... ID

R1 not ready yet

ADD R1 R2 R3 R1 = R2 + R3

SUB R5, R1, R3

LW R10, 25(R3)

ADD R12, R10, R6

Lec-41

## Verilog modeling of the processor - Part 1

→ Behavioural verilog code to implement pipeline.

→ Two special 1-bit variables are used:

→ HALTED : Set after a halt instruction executes & reached WB stage.

→ TAKEN_BRANCH : Set after a decision to take a branch is known. required to disable the instruction that have already entered the pipeline from making any state changes.

### How pipeline works?

| | | | | | | |
|---|---|---|---|---|---|---|
| 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 |

{ ≡≡≡ HLT

Suppose we have a sequence of instructions to be executed and after that we have HLT instruction. the processor is assumed to stop now.

Suppose instruction i+1 is HLT instruction. There are i, i-1, i-2th instruction earlier as well.

The i+1 HLT instruction is decoded during ID stage*.

During the ID stage of (i+1)th instruction, instruction i is in EX stage. So we can not stop the processor right there because previous instructions ie i, i-1, i-2 are not completed. So we wait till WB stage of i+1 stage. for that HALTED is used.

as soon as HLT instruction is decoded, we set the HALTED to 1.

and before executing any further instructions, the flag HALTED is checked if it is equal to 1. therefore i+1 and i+2 will not be carried out