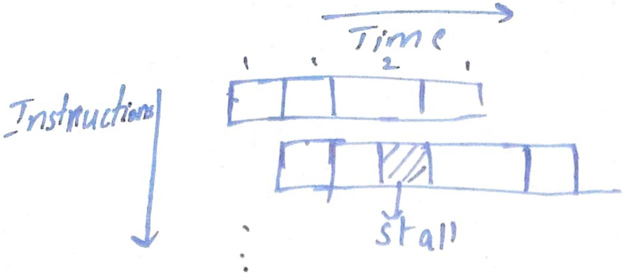
Pipelining

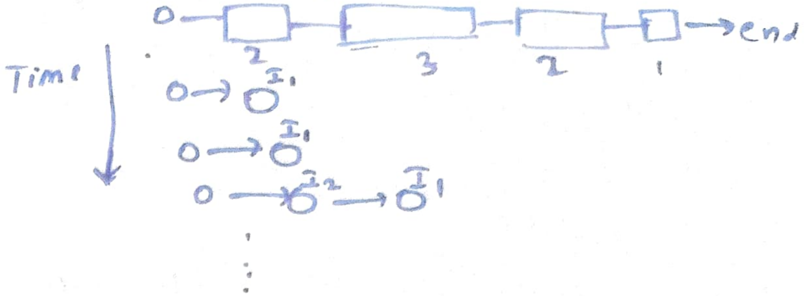
→ Overall speed is as good as the speed of the slowest unit

→ Non-uniformity is bad ~~a~~ and exceptions are bad.

Pipeline Timing Diagram



Pipelining ~~in Abstract~~ Diagram (Alterna



~~Pipe~~ Speedup :- ~~λt~~ $\dfrac{\text{Time}_{\text{without piplelining}}}{\text{Time}_{\text{with pipelining}}}$

as N (No. of instructions) → ∞ Speedup → $\dfrac{\text{Total time for an instruction}}{\text{Time for the longest ~~cycle~~ step}}$

Ideal Speedup (All steps take equal time t)

for N units (instructions),

$$\text{Speedup} = \frac{N \times (\overset{\text{no. of steps in pipeline}}{K} t)}{Kt + (N-1)t} = \frac{NK}{N+K-1}$$

i.e $\boxed{\lim_{N\to\infty} \text{speedup} = K}$ ↳ ideal speedup

↳ all stages must be of equal time

↳ Enough hardware

↳ Large units (instruction)

Latency :- Time taken for a single instruction to execute

Throughput :- Rate at which instructions complete

Pipelining in MIPS

Stage1:- Instruction fetch, PC+=4 (all inst's) [IF]

Stage 2:- Instruction decode, [Reg read, Branch tgt computation] (all inst's) [ID]

Stage 3:- Execute, { ALU operation (reg-reg) / Memory address computation (lw, sw) / Branch condition computation (beq) } [EX]

Stage 4:- Memory, [Memory access (lw, sw)] [MEM]

Stage 5:- Write Back [Reg write back (reg-reg) / Reg write back (lw)] [WB]

Hazards in a pipeline

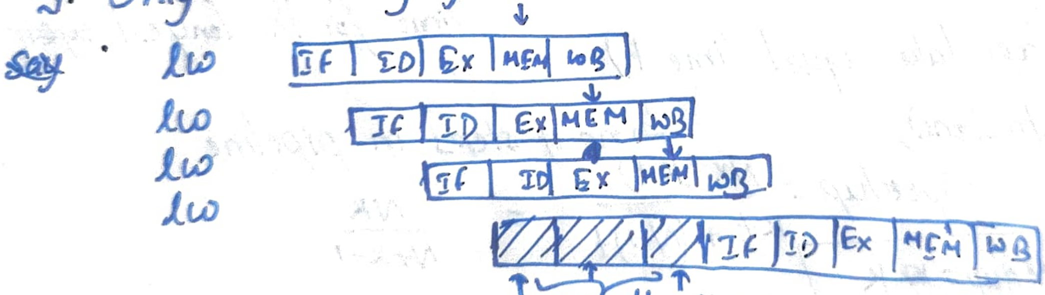
- Structural Hazard
- Data Hazard
- Control Hazard

Structural Hazard → Insufficient hardware

Eg:- In stage 1 and 2 → seperate hardware required for PC+=4, branch tgt computat

↳ Can be solved using Stalling

Eg:- Only 1 memory for instructions and data



3stalls (since MEM and IF use same hardware)

Throughput decreases

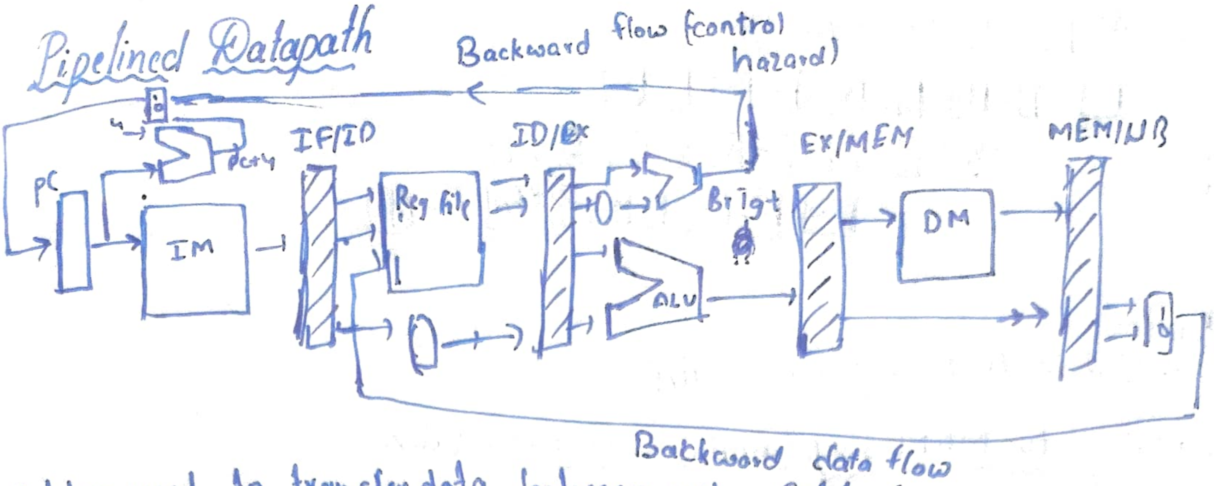
- CPI in ideal pipeline is 1
- CPI in pipeline with stalls = 1 + f_{stalls}

Structural Hazard in Register file

Stage 2 → Read and stage 5 → Write

This is fixed by writing on rising edge (first half) reading on falling edge (second half)

Pipelined Datapath


Backward flow (control hazard)
4
Dury
PC
IM
IF/ID
Reg file
ID/EX
Br tgt
ALU
EX/MEM
DM
MEM/WB
Backward data flow (data hazard)

Latches used to transfer data between ~~cycl~~ stages.

Data Hazards

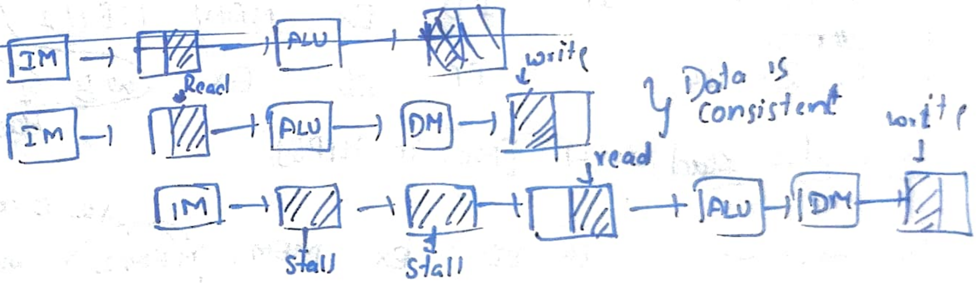
Eg:
add $t1 $t2 $t3 — written in cycle 5 (WB)
add $t4 $t1 $t5 — ~~used~~ read in cycle (1+2) (start, ID)
↳ old value used

- Stalling can solve
- But, can be improved by doing the reads after writes

~~add $t1 $t2 $t3~~

add R1 R2 R3

add R4 R1 R5



- Data forwarding

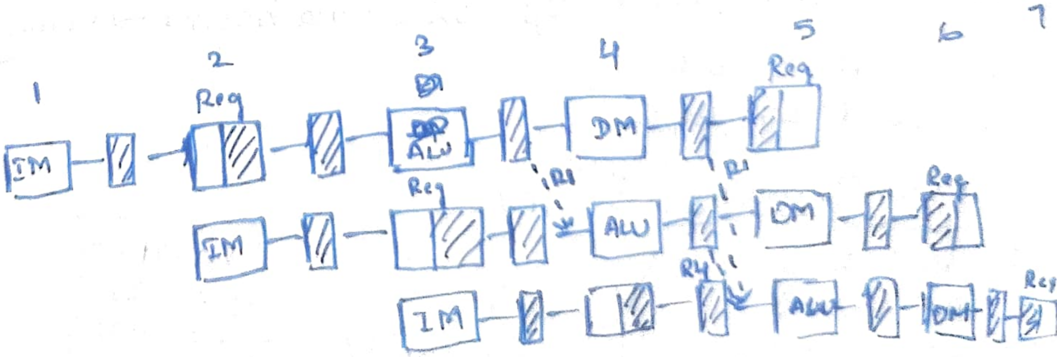
Eg:
add R1, R2, R3
sub R4, R1, R5
and R6, R1, R7
or R8, R1, R9

- Data forwarding

add R1, R2, R3

sub R4, R1, R5

and R6, R1, R4



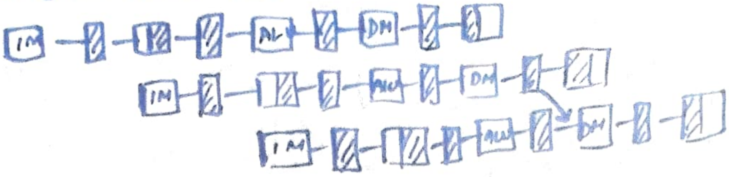
At 4, data of R1 is forwarded to ALU from EX-MEM Latch (

At 5, data of R1 is forwarded to ALU from ~~DM DD-WB Latch~~ MEM-WB Latch (

data of R4 is forwarded to ALU from Ex-MEM Latch

Data forwarding to MEM stage

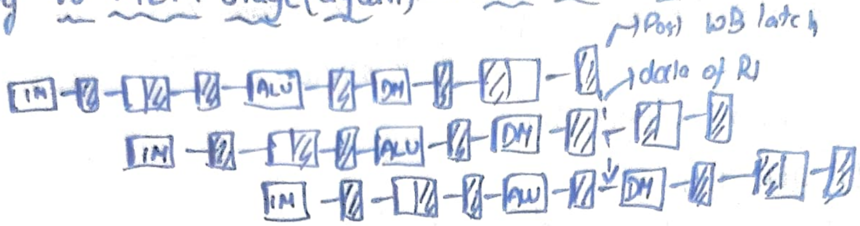
```
ADD  R5,R2,R3
LW   R1,0(R4)
SW   4(R6),R1
```



What if
SW 4(R1), R2
↳ needed in ALU → requires 1 stall?

Data forwarding to MEM stage(again) → Post WB latch

```
add  R1, R2, R3
lw   R5, 0(R4)
sw   4(R6), R1
```



Data Hazards Classification

→ Read after Write (RAW): data forwarding used
→ Write after Write (WAW):- when writes happen in diff stages [Not in MIPS]

LW R1 O(R2)	IF	ID	Ex	MEM1	MEM2	WB
~~ADD R1 R1~~						
ADD R1 R3 R4		IF	ID	EX	WB	

↓ Hazard

→ Write after Read (rare) [Not in MIPS]

sw R1, 0(R2)	IF	ID	EX	MEM1	MEM2	WB
add R2, R4, R3		IF	ID	EX	WB	

→ R2 is read here (in second half)
R2 is written here (in first half)

~~If one more~~ ∴ wrong R2 value is ~~used~~ read by sw

Data forwarding can't always solve data hazard → stalling required

Eg:-
```
LW   R1  O(R2)
add  R4 ,R1, R3
```

Code Scheduling { a=b+c
d=e+f }

```
lw   r1, b
lw   r2, c
add  r4, r1, r2      ↳ stall
sw   a, r4                      → naive compiler
lw   r10, e
lw   r11, f
add  r12, r10, r11   ↳ stall
sw   d, r12
```

```
lw   r1, b
lw   r2, c
lw   r10, e
lw   r11 f
add  r4, r1, r2      ↳ Clever compiler
sw   a, r4              no stalls
add  r12, r10, r11
sw   d, r12
```

Control Hazards

Eg:- add R1, R2, R3
beq R4, R5, LBL
sub R6, R7, R8
lw R6 0(R7) ↳ need not be executed

Stalling fixes (2 cycles are stalled)
↳ bad efficiency

→ Techniques to reduce branch penalty

↳ 2-Stage branch completion

- Extra comparator in stage-2 [for branch condition]
  ↳ Needs to be completed in halfcycle [since regs are read in second half]
  ↳ Can cause data hazards [forward to ID stage]
  ↳ can cause additional stall for beq

add	R1,R2,R3	IF	ID	EX	MEM	WB			
beq	R4,R5,LBL		IF	ID	EX	MEM	WB		
sub	R6,R7,R8			\\\	IF	ID	EX	MEM	WB

~~Stall CR~~ Stall for beq due to 2-stage

add	R1, R2, R3	IF	ID	EX	MEM	WB		
beq	R1, R7, LBL		IF	▨	ID	EX	MEM	WB

R1 data needs to be forwarded

↳ Assume branch not taken

• control is run assuming branch is not taken, if branch is taken, the instruction is cancelled out

add	r1	r2 r3	IF	ID	EX	MEM	WB				
beq	r4	r5 LBL		IF	ID	EX	MEM	WB			
sub	r6	r7 r8			IF	ID	EX	MEM	WB		
lw	r6	0(r7)				IF	ID	EX	MEM	WB	

PC is adjusted here in first half

Read here in second half

If branch taken, needs to be cancelled out

Techniques to reduce control hazards

- Branch Prediction

  Predicting if branch taken or not taken, based on ~~previous~~ whether branch was taken last time

  → Single bit predictor and 2-bit predictor [remembers 2 instances]

- Delayed branches

  ↳ Instruction after branch will be executed, even if branch is taken
  ↓
  in branch-delay slot → filled by compiler

Eg:

```
add R1 R2 R3
beq R4 R5 LBL
| sub R6 R7 R8 |
```

Filling the Branch delay-slot

From before branch

```
and r1, r2, r3
sll r5, r4, 2
or  r2, r3, r4
beq r2, r10, LBL
[          ]
```

From branch fall through

```
and  r1, r2, r3
or   r2, r3, r4
beq  r2, r10, lbl
[          ]
addi r5, r2, 4
sub  r6, r2, r7
  :
LBL:
addi r2, r5, 4
add  r6, r5, r2
```

From branch target

```
and  r1, r2, r3
or   r2, r3, r4
beq  r2, r10, lbl
[          ]
addi r5, r2, 4
sub  r6, r2, r5
  :
LBL:
addi R7, r5, 4
add  r6, r5, r7
```

→ If there is no instruction fill in delay slot → fill nop

Pipeline Control

Single cycle control lines

ALUSrc, ALUOp → Ex
MemRd, Mem Wr → MEM
RegWr, RegDst, Mem2Reg → WB
branch → Ex

The required control lines are carried forward by latches i.e lot in by ID-EX latch (all ctrl lines) and less by MEM-WB latch (only MEM-WB ctrl lines)

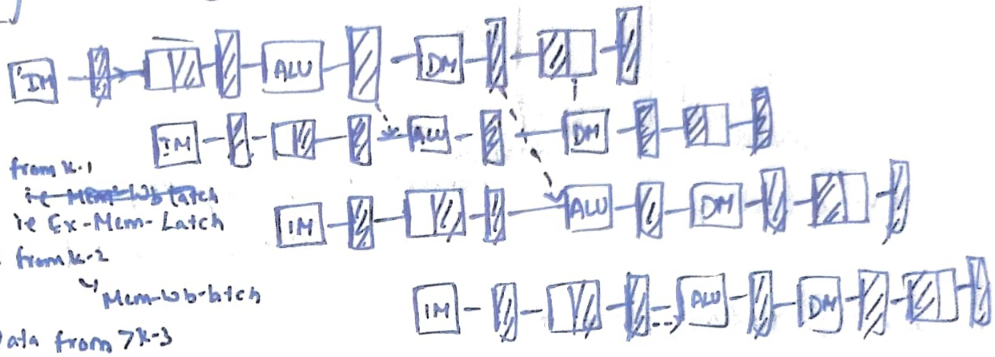
Control unit in latch is combinatorial

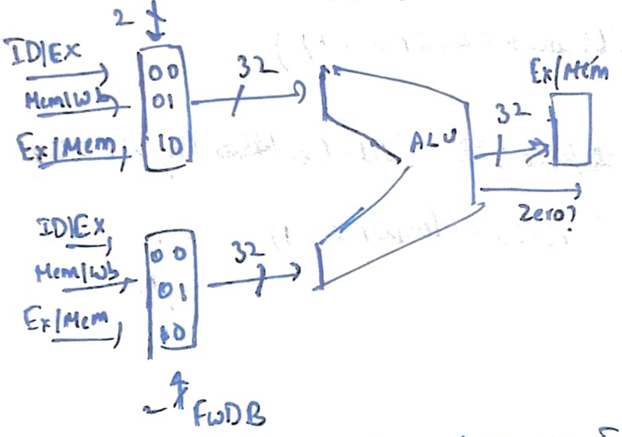
→ Single cycle control → combinational circuit
→ Multi cycle control → State machine
→ Pipeline control → micro-code (Turing machine)
↳ code executed by a simpler processor within main processor (to generate control lines)

Pipeline Control for Data forward to Ex

Data producing instn
add r1, r2, r3

Dependent inst^s
- Sub r4, r1, r5 ↳ Data from k-1 ~~ie Mem-Wb latch~~ ie Ex-Mem-Latch
- and r6, r1, r2 ↳ Data from k-2 ↳ Mem-wb-latch
- or r8, r1, r9 ↳ Data from 7k-3 ↳ ID-EX latch





Logic for FwDA and FwDB takes place in ID stage, but fwDing is in Ex stage

PseudoCode (For FwDA/FwDB)

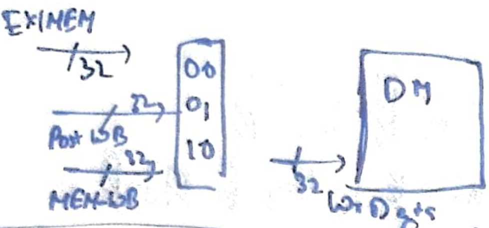
```
// Case 1 : no fwding
  FwdA = 00
// Case 2 : fwd from k-2
  if ((IF/ID.Rs|Rt) == ID/Ex.Rd) && (ID/Ex.RegWr == 1) && (ID/Ex.MemRd == 0)) :
     (FwdA | FwdB) = 01
// Case 3  Fwd from K-1
  if ((IF/ID.(Rs|Rt) == Ex/MEM.Rd) && (            ) && (          )) :
     (FwdA / FwdB) = 10
```

order of cases is imp, because of exceptions

add r1, r2, r3
sub r1, r4, r5
slt r6, r1, r0

Forwarding to Mem :

	CC1	CC2	CC3	CC4	CC5
OR R1, R4, R5	MEM	WB			
Sll R1, R2, 3	Ex	MEN	WB		
add R1, R2, R3	ID	Ex	MEM	WB	
Sw R1, 4(R20)	If	ID	Ex	MEM	WB



Fowding from Reg-Reg
From lw → Rd becomes Rt

```
// Case 1 :- no fwding
FwdMem = 00
// Case 2: fwding from k-2
if( IF/ID.Rt == Ex/MEM.Rd) && (Ex/MEM.RegWr == 1) &&
    (Ex/Mem.MemRd == 0) && (Ctl unit.MemWr == 1))
    FwdMem = 01
IF (IF/ID.Rt == ID/Ex.Rd) && (Id/Ex.Reg Wr == 1)
    && (ID/Ex.MemRd == 0) && (Ctl unit.MemWr == 1)
    Fwd Mem = 10
```

Pipeline Control for Stalling

Only case :- Dependent reg-reg after lw

```
// Case 1 - dependent Rs
if ((IF/ID.Rs == ID/Ex.Rt) && (ID/Ex.MemRd == 1))
    Stall
// Case 2 - dependent Rt
if ((IF/ID.Rt == ID/Ex.Rt) && (ID/Ex.MemRd == 1)).
    Stall
```

Stall

lw r1 0(r4)	IF	ID	Ex	MEM	WB		
add r3 r1 r4		IF	ID (nop)	Ex (nop)	MEM(nop)	WB (nop)	
add r3 r1 r4			IF	ID	Ex	MEM	WB

∴ To perform a stall,

when ID of add is run, the ctrl sgnls are all set to 0, performing nop (especially MemWr, MemRd, Reg ~~Rd~~ Wr)

and PC must not be changed (since IF stage should repeat)

and disable writing of IF/ID latch (why??)

PCWr = 0
IF/ID·Wr = 0
ID/Ex latch = 0

Stalling Logic for Control Hazard (3-stage branch condition)

```
if (CtlUnit. branch = =1)
    IF/ID latch=0     // 1st nop follows branch
if (ID/Ex.branch ==1)
    IF/ID latch =0    // 2nd nop follow branch
```

Here nop follows branch, whereas previously nop is run and next instn is run

Exceptions in Pipelines

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8
sll r2, r2, 3	IF	ID	EX	MEM	WB			
lw r4, 4(r5)		IF	ID	EX	MEM	WB		
add r1, r2, r3			IF	ID	EX	MEM	WB	
sw r4, 4(r20)				IF	ID	EX	MEM	WB

misalignment

detected here

This pipeline should be flushed and sll needs to be exec

∴ not flush MEM/WB latch

PC value of exception → stored in EPC [PC-8 in this case, if sw PC not overwritten]

Cause register → approp value is loaded

PC = 0x8000 0180 → exception handler

Exceptions

1) Multiple exceptions → first exception needs to considered

2) Out-of-order completion
roll back reqd

mul R1, R2 R4	IF	ID	MUL1	MUL2	MUL3	MUL4	MUL5	WB
add R4, R5, R6		IF	ID	EX	MEN		WB	

Ovf Excep

Completed

3) Partially Changed Machine State → String copy in intel
→ halfway error (due to other instn)