MICROPROCESSOR AND COMPUTER ARCHITECTURE

UNIT-2

Pipelining

feedback/corrections: vibha@pesu.pes.edu

VIBHA MASTI

PIPELINING

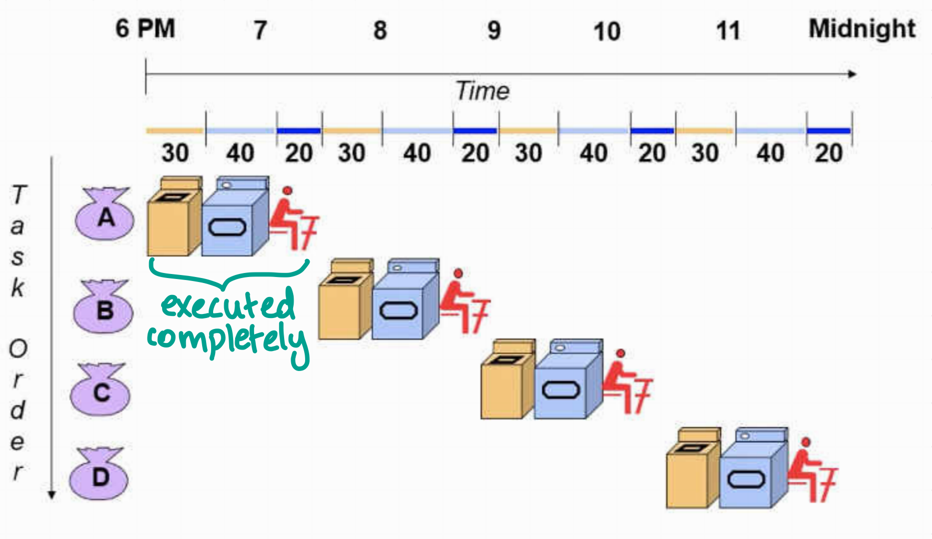
- Multiple instructions overlapped in execution
- Parallelism
- eg: assembly line for automobile manufacturing
- Each step: pipe stage/segment
- stages connected to form pipe
- Instructions enter from one end, progress through the stages, and exit through the other end

throughput

- automobile — how often a completed car exits assembly line
- computer — how often instruction exits pipeline
- time required to move instruction from stage i to stage i+1 in the pipeline is called processor cycle
- length of processor cycle determined by time required for the slowest pipe stage
- time per instruction =

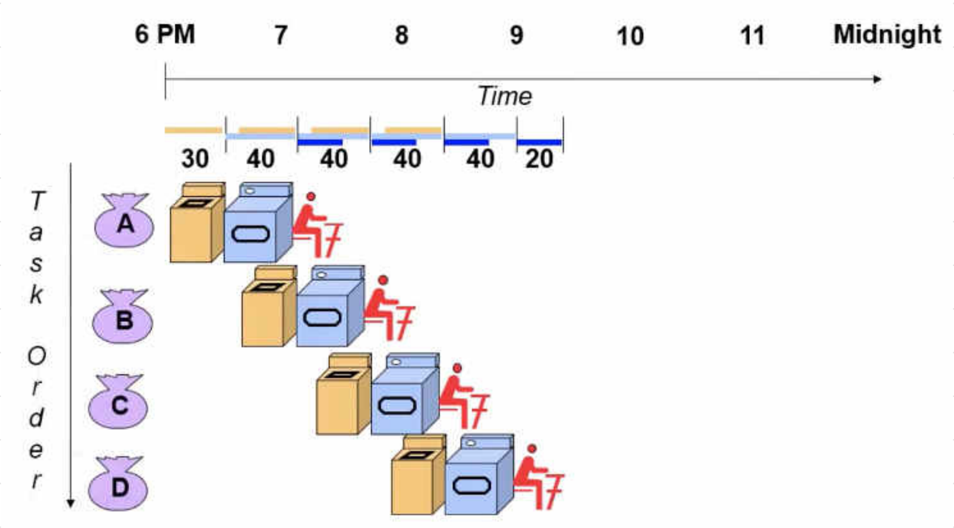
$$\boxed{\frac{\text{time per instruction on unpipelined machine}}{\text{no. of pipe stages}}}$$

SEQUENTIAL



executed completely

PIPELINED

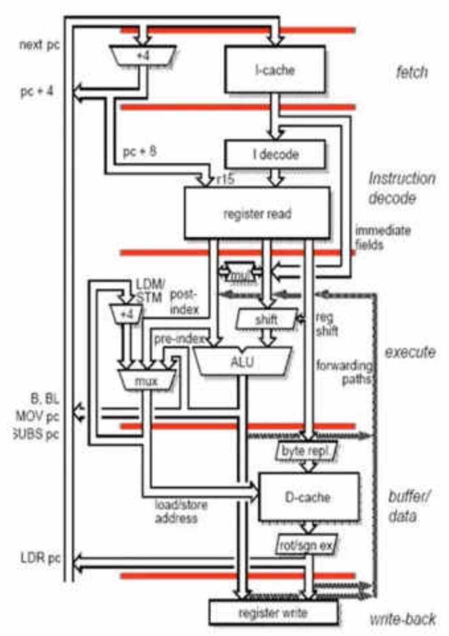


ARM ARCHITECTURE - 5 stage pipeline

- ARM9TDMI - 5 stages Harvard
- ARM7TDMI - 3 stages von Neumann

1. Fetch IF
2. Decode ID
3. Execute EX
4. Buffer/Data or Mem Access MEM
5. Write back WB

- 3 stage: fetch, decode, execute

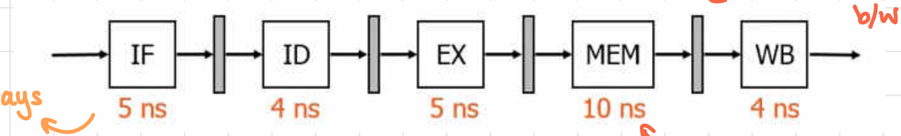


throughput

- Throughput = instructions completed per second (how often an instruction is completed) due to parallelism
- Latency = time taken to execute a single instruction in the pipeline

Q: Determine pipeline throughput and latency

Pipeline registers store values b/w cycles



delays

max delay

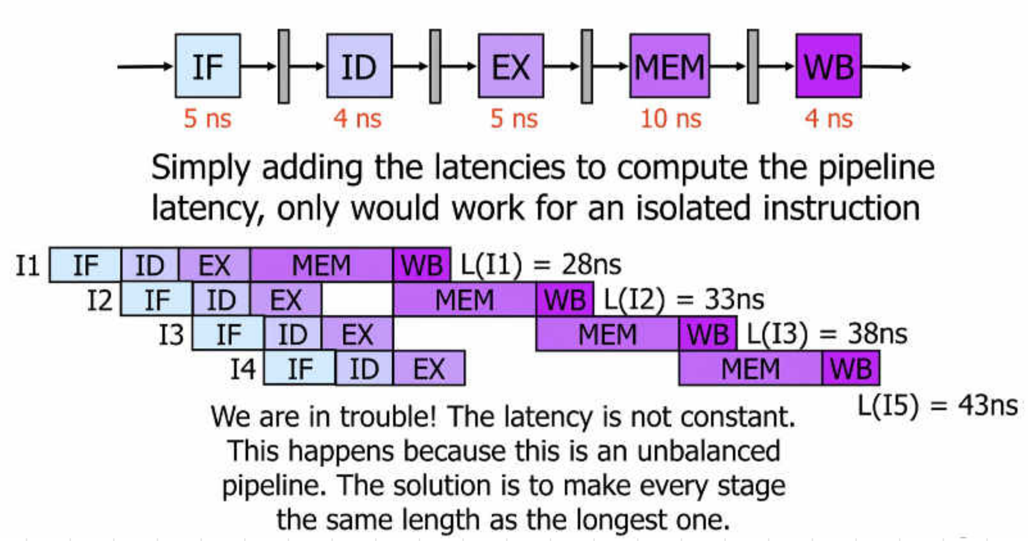
throughput = 1 instr / 10 ns (ignoring pipeline register overhead)

latency = 5+4+5+10+4 = 28 ns (for isolated instr)

Example

instr 1	IF	ID	EX	MEM	WB		
instr 2		IF	ID	EX	MEM	WB	
instr 3			IF	ID	EX	MEM	. . .
instr 4				IF	ID	EX	
instr 5					IF	ID	

- one instruction per clock cycle
- unbalanced pipeline: each instr takes diff times — latency of each instruction increases and time is wasted
- solution: balance by making all instr as long as slowest one

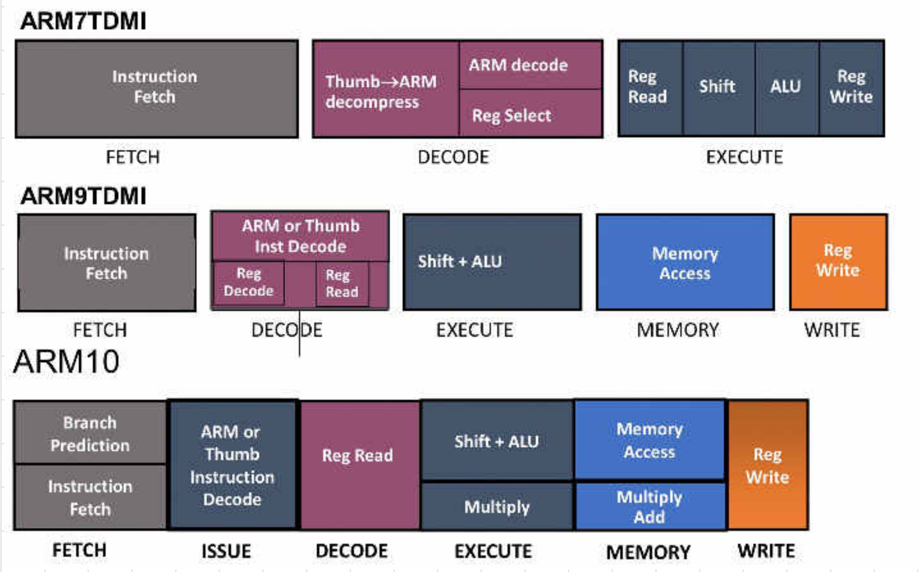

IF
ID
EX
MEM
WB
5 ns
4 ns
5 ns
10 ns
4 ns

Simply adding the latencies to compute the pipeline latency, only would work for an isolated instruction

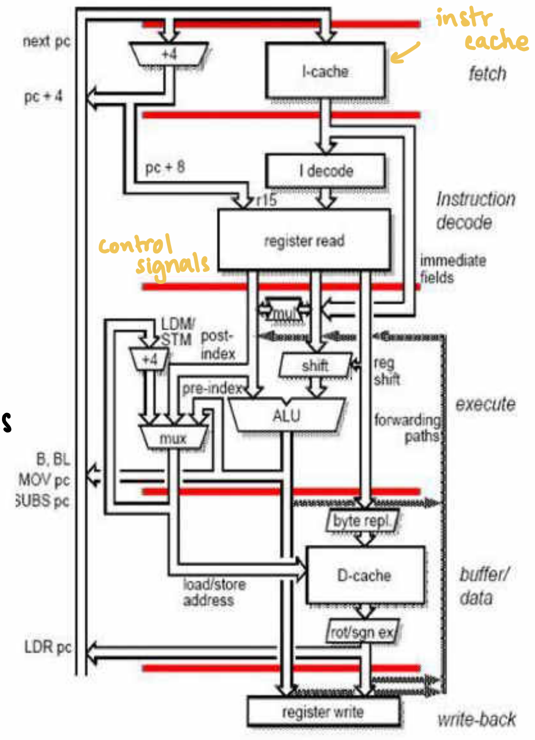
I1 IF ID EX MEM WB L(I1) = 28ns
I2 IF ID EX MEM WB L(I2) = 33ns
I3 IF ID EX MEM WB L(I3) = 38ns
I4 IF ID EX MEM WB
L(I5) = 43ns

We are in trouble! The latency is not constant. This happens because this is an unbalanced pipeline. The solution is to make every stage the same length as the longest one.

Stages of Pipeline



5-Stage ARM Processor



1. Fetch (IF)

- instr fetched from memory
- placed in pipeline
- PC updated to PC+4
- one clock cycle

2. Decode (ID)

- instr decoded
- reg operands read from reg files
- 3 operand read ports
- equality test on reg for possible branch
- sign extend offset
- one clock cycle
- branch : only fetch & decode

- add sign-extended offset to PC in case branch needs to happen (branch target address)
- if branch needs to happen, store back in PC reg
- fixed-field decoding

3. Execute (EX)

- effective address cycle
- ALU operations (barrel shifter, multipier, MUX)
- three possible functions

  (a) Memory Reference

  - ALU adds base reg and offset to calculate effective address LDR R0, [R1], #4

  (b) Reg-Reg ALU instruction

  - ALU performs operation specified by opcode on values read from reg file

  (c) Reg-Imm ALU instruction

  - ALU performs operation specified by opcode on first value read from reg file and sign extended immediate

← effective address compute

- in LOAD/STORE architecture, EA and execution cycles are combined to single clock cycle as no instr needs to simultaneously do both (LDR/STR vs ALU)

4. Buffer/Data or Memory Access (MEM)

- Data in memory is accessed (LDR/STR)
- Otherwise, ALU result buffered for one clock cycle
- LOAD: mem read using effective address
- STORE: mem writes from second reg to effective address

5. Write back (WB)

- Data written back to reg (result of instr)
- Data either from memory system (LOAD) or ALU

execution time

CPU Time = instruction count (IC) x clock cycle x CPI

$\downarrow$

cycles per instr

Pipeline Issues

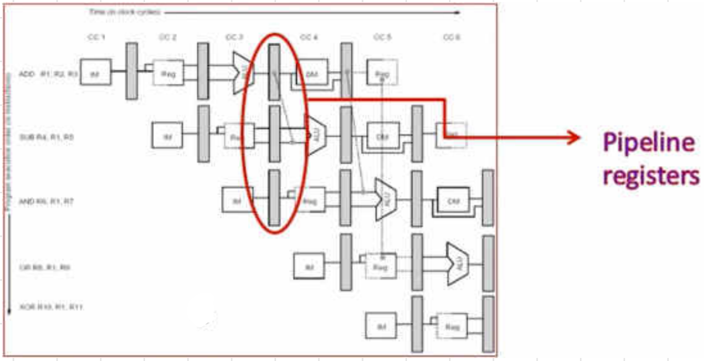
1. Pipeline imbalance
   - reduces performance as clock can run no faster than time needed for slowest pipeline stage
   - unbalanced pipeline

2. Pipeline latency
   - limits arise from imbalance among pipeline stages and pipelining overhead

3. Pipeline overhead
   - combination of pipeline register delay and clock skew

4. Pipeline registers delay
   - add setup time that triggers a write and propagation delay to the clock

5. Clock skew
   - max delay between when the clock arrives at any two registers (pipeline regs)

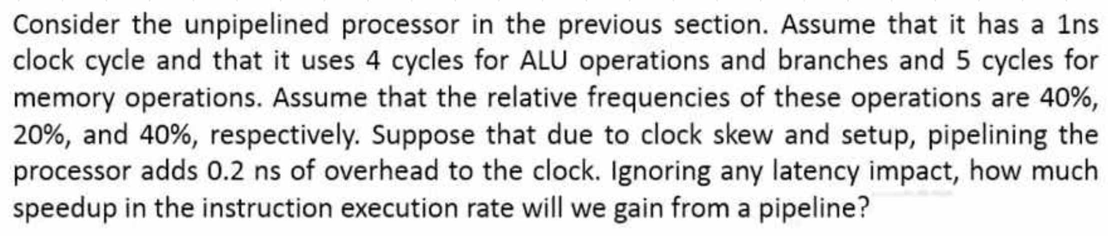


HAZARDS

- Dependency of instructions: instructions need to wait for execution before next dependent instruction can execute

- Ideal pipelines: no dependence

Q:



Consider the unpipelined processor in the previous section. Assume that it has a 1ns clock cycle and that it uses 4 cycles for ALU operations and branches and 5 cycles for memory operations. Assume that the relative frequencies of these operations are 40%, 20%, and 40%, respectively. Suppose that due to clock skew and setup, pipelining the processor adds 0.2 ns of overhead to the clock. Ignoring any latency impact, how much speedup in the instruction execution rate will we gain from a pipeline?

$$\text{speedup} = \frac{\text{avg instruction time unpipelined}}{\text{avg instruction time pipelined}}$$

$$\text{avg unpipelined} = (1\times4\times0.4) + (1\times4\times0.2) + (1\times5\times0.4)$$
$$= 1.6 + 0.8 + 2.0$$
$$= 4.4 \text{ ns}$$

$$\text{avg pipelined} = 1 + 0.2$$
$$= 1.2 \text{ ns}$$

$$\text{Speedup} = \frac{4.4}{1.2} = 3.67 \text{ times}$$

- # HAZARDS

- Situations that prevent next instruction in instruction stream from executing in its designated clock cycle
- Hazards reduce performance from ideal speedup
- Hazarde make pipeline stall - no new fetching during stall

1) Structural hazards

- arise from resource conflicts
- hardware unable to support all combinations of overlapping instructions

2) Data Hazards

- instruction depends on results of previous instruction
- exposed by overlapping of instruction

3) Control Hazards

- pipelining of branches / instructions that change PC

Performance of Pipelining With Stalls

- Causes pipeline performance to degrade from ideal performance

CASE 1: NO DEPTH

$$\text{Actual speedup} = \frac{\text{CPI unpipelined} \times \text{clock cycle unpipelined}}{\text{CPI pipelined} \times \text{clock cycle pipelined}}$$

$$= \frac{\text{CPI unpipelined}}{\text{CPI pipelined}} \times \frac{\text{clock } \cancel{\text{cycle}} \text{ unpipelined}}{\text{clock } \cancel{\text{cycle}} \text{ pipelined}}$$

assume equal

$$= \frac{\text{CPI unpipelined}}{\underset{\overset{\uparrow}{\text{ideal}}}{\textcircled{1}} + \text{pipeline stall clock cycles per instruction}}$$

$$\boxed{\text{speedup} = \frac{\text{CPI unpipelined}}{1 + \text{pipeline stall clock cycles per instruction}}}$$

CASE 2: DEPTH

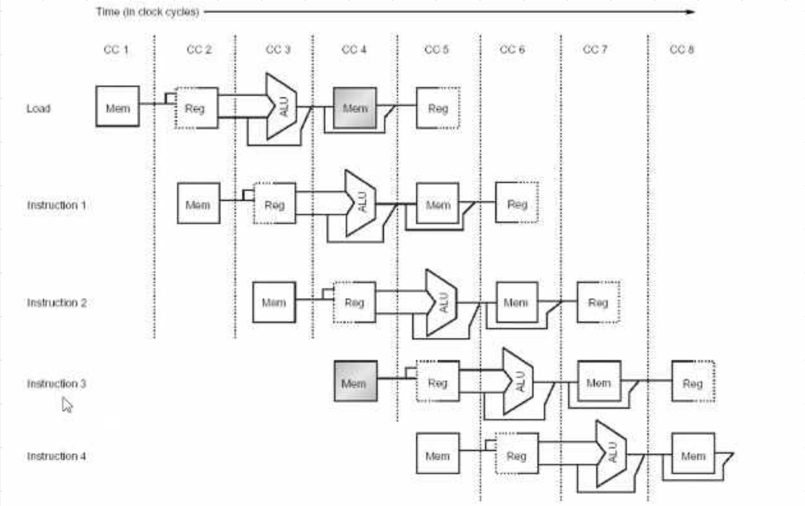
pipeline depth = no. of stages

$$\boxed{\text{speedup} = \frac{\text{pipeline depth}}{1 + \text{pipeline stall clock cycles per instruction}}}$$

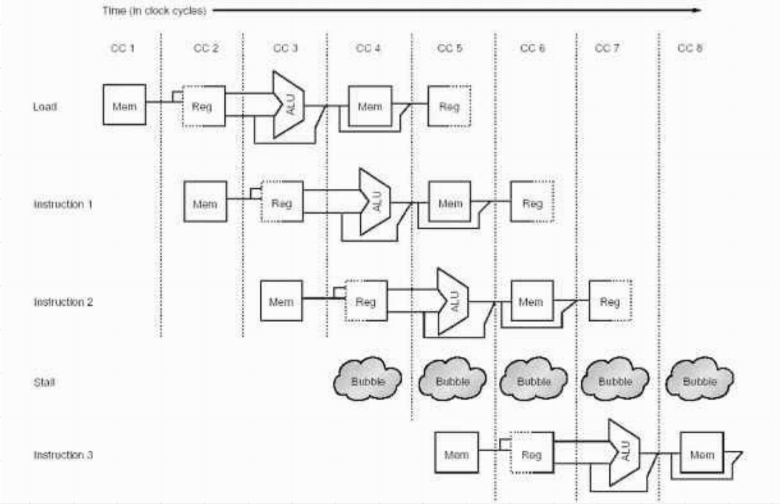
1. Structural Hazards

Hazards - Von Neumann Architecture

- same mem for instr, data



- stall added (pipeline bubble)



Eliminate Hazard

- Duplicate resource (I-Cache, D-cache)
- Stall pipeline (no-op)

2. Data Hazards

- Dependency: name & data dependency
- Results of prev instructions required for next instructions
- Pipeline registers

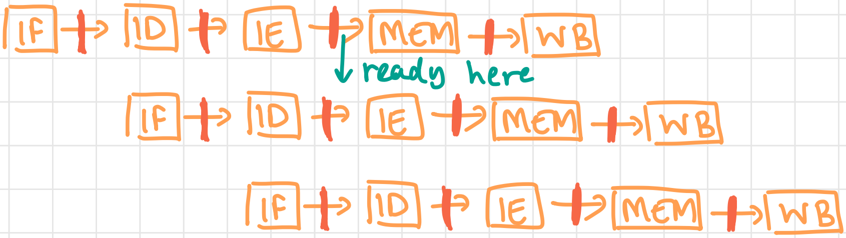
i) Read After Write (RAW) Hazard

- instr J reads before instr I writes it

```
ADD R1, R2, R3
SUB R4, R1, R5
AND R5, R6, R1
OR  R6, R7, R1
```

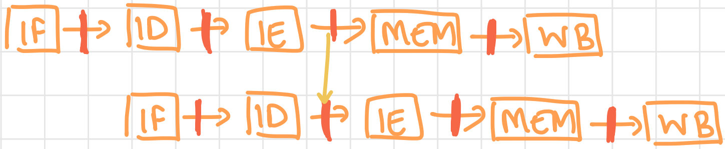
} data from R1 required

- aka data dependency



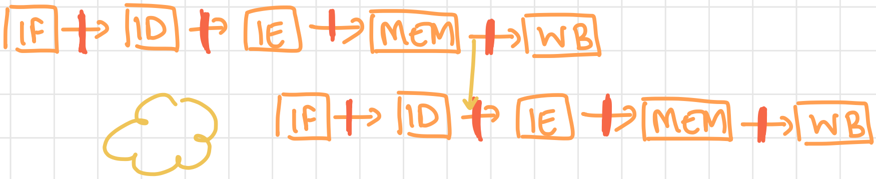
Solution

- stall pipeline (2 stalls for eg)
- data forwarding/operand forwarding
- ALU: no stalls if DF used



- stalled for only 1 cc

```
LW   R1, 0(R2)
SUB  R4, R1, R5
AND  R5, R6, R1
OR   R6, R7, R1
```



- stall needed

- No way to eliminate stalling

2) Write After Read (WAR) Hazard

- instr J writes before instr I reads it

```
ADD  R2, R1, R3
SUB  R1, R4, R5   ← supposed to be
                    overwritten
                    after reading
```

- no data dependency

- out of order execution : instr J ends up writing before instr I (permits OOO exec)

- aka antidependency or name dependency (name causes conflict, not result)

Solution

- use differenent register name

3) Write After Write (WAW) Hazard

- instr J writes before instr I writes it

```
ADD  R1, R2, R3
SUB  R1, R4, R5
MUL  R2, R1, R6   ←  can end up reading wrong data
```

- output dependency / named dependency

Q: a = b+c;
d = e-f;   (MIPS)

```
LW   R1, b        → 5 cc
LW   R2, c        → 6 cc
ADD  R0, R1, R2   → 8 cc
SW   R0, a        → 9 cc
LW   R4, e        → 10 cc
LW   R5, f        → 11 cc
SUB  R3, R4, R5   → 13 cc
SW   R3, d        → 14 cc
```

dependency

LW followed by ALU: 1 stall

Data Forwarding

- hardware solution

Compiler Rearrangement

- software solution - software scheduling
- reduce dependencies by changing order
- minimise penalty
- without changing logic

```
LW  R1, b
LW  R2, c
LW  R4, e
LW  R5, f
ADD R0, R1, R2
SW  R0, a
SUB R3, R4, R5
SW  R3, d
```

design tradeoff:
stalling vs
dummy instruction

FLUSHING PIPELINE

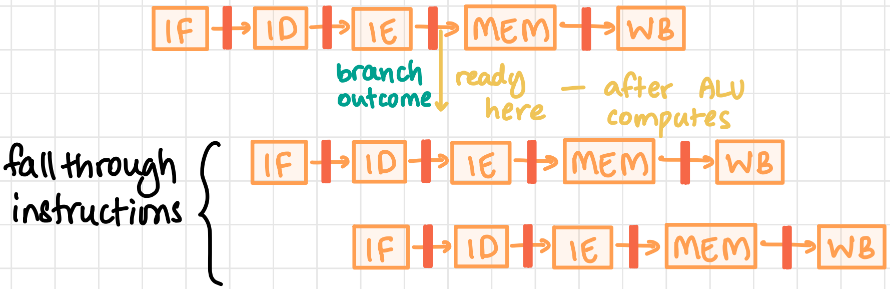
- Pumping in 0's to empty pipeline

3. Control Hazard

- control sequencers

```
       beq   r1, r3, 36
       add   r2, r3, r4
       and   r5, r6, r7

36     xor   r7, r8, r9
```

- PC → PC+ offset in branch instruction
  ↳ target address



- if branch condition is true, next fall through instr should be flushed from pipleline

- when decision made, 3 instr in pipeline

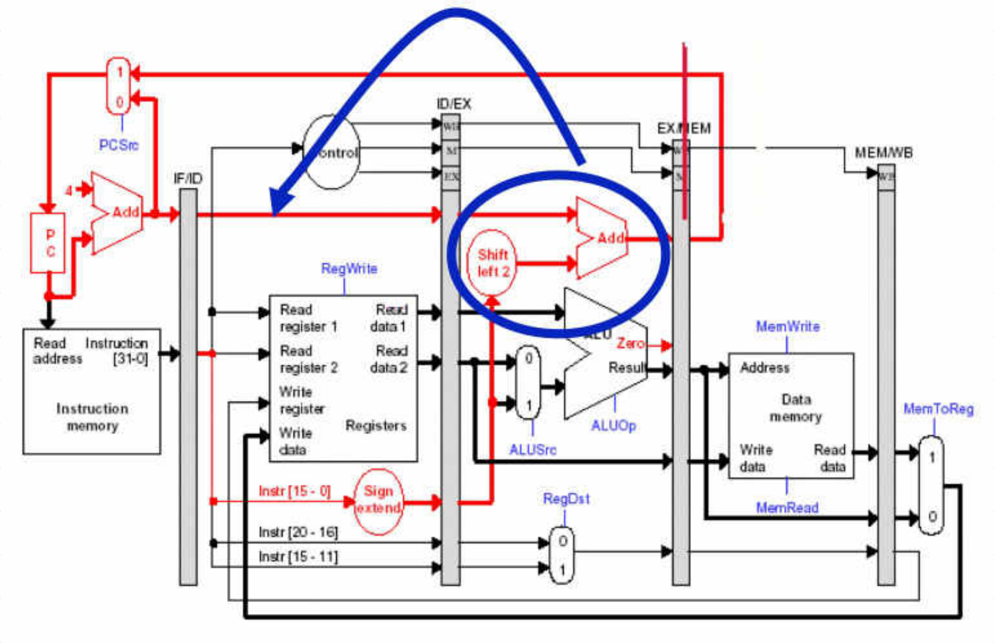
Solution

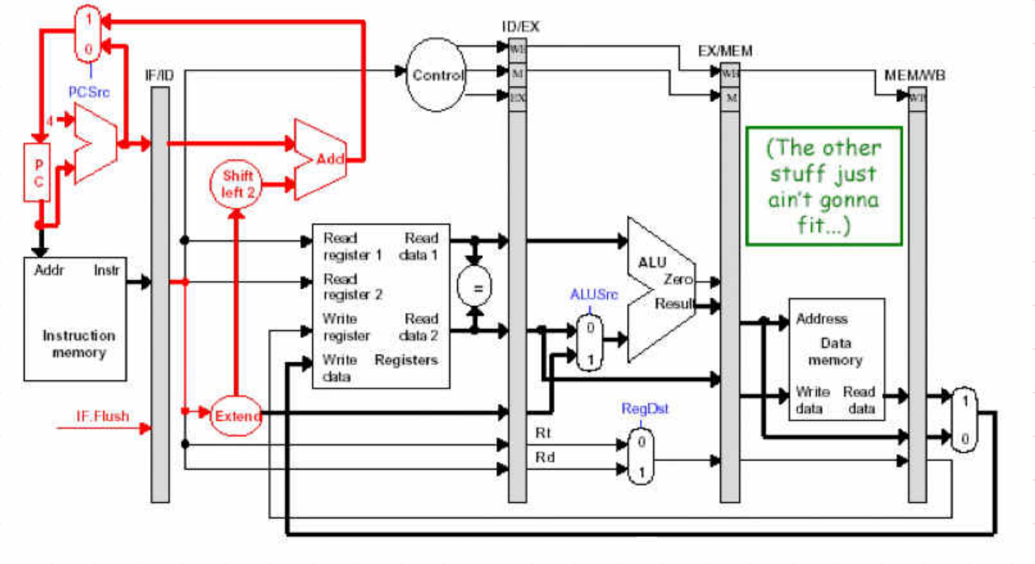
1. Stall pipeline until outcome of branch known
2. Zero tester circuit
   - move decision hardware to ID stage

- Branching done at ID stage — takes 2 cc

- Henceforth: branch – 2 cc

- Branch prediction : field in CS ; moment it is fetched, decide if it is branch (research topic)





(The other stuff just ain't gonna fit...)

Branch Prediction

1. Static branch prediction   compile time
2. Dynamic branch prediction   run time

Static branch Prediction

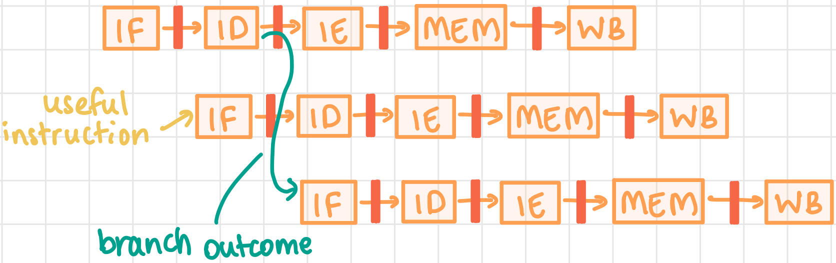
- compiler makes prediction ; ID stage
- 4 alternatives

1. Stall until branch direction is clear
   - do nothing until direction known

2. Predict branch not taken
   - untaken branch
   - compiler thinks not taken
   - penalty if taken

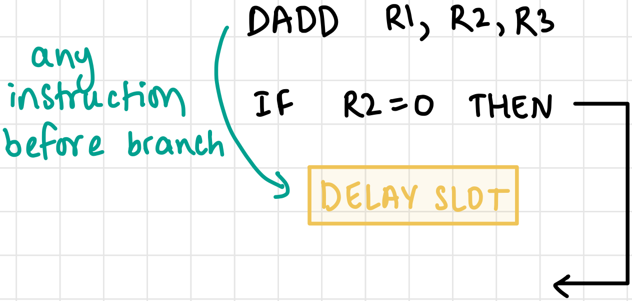
3. Predict branch taken
   - taken branch
   - penalty of 1cc if wrong (untaken)

4. Delay slot
   - insert another useful instruction right after branch
   - special instr (delay slot) that would otherwise also execute



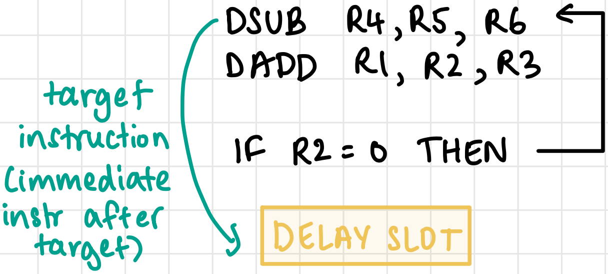
- three kinds:

(4a) From before



```
DADD  R1, R2, R3

IF  R2=0  THEN

DELAY SLOT
```

(4b) From target

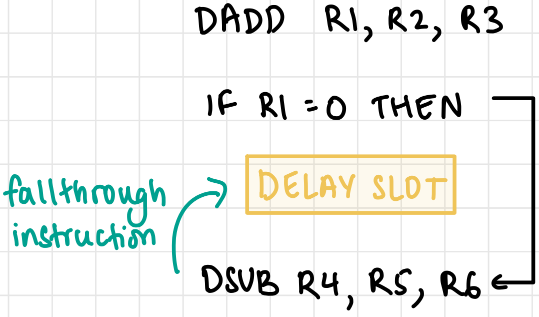


```
DSUB  R4, R5, R6
DADD  R1, R2, R3

IF  R2 = 0  THEN

DELAY SLOT
```

hardware can determine

high probability that branch condition is true and branch is taken

(4c) From fallthrough



```
DADD  R1, R2, R3

IF R1 =0 THEN

DELAY SLOT

DSUB R4, R5, R6
```

high probability of branch not taken

BRANCH HISTORY TABLE

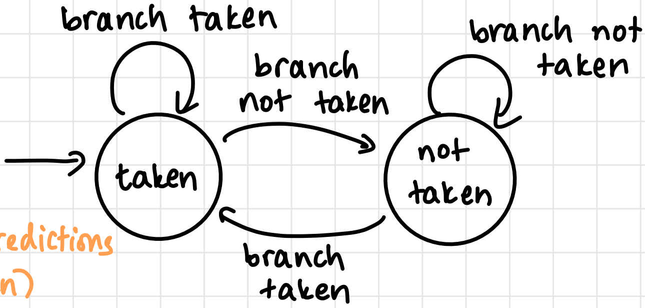
- some bits reserved to store branch history
- branch prediction buffer (BPB / BHT)
- at runtime, prediction made using BPB (table lookup)

dynamic branch Prediction

- based on branch prediction buffer, decisions made at runtime
- two kinds:
- IF stage : buffer

1. One-bit predictor

- 0 & 1 : 2 states
- not taken & taken



(mispredictions often)

no. of mispredictions depends on start stat

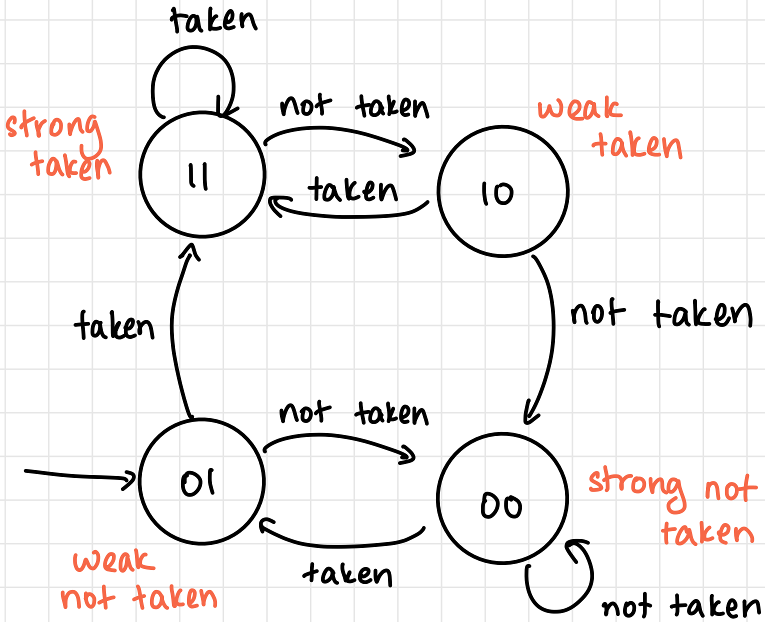
2. Two-bit predictor

conviction bits

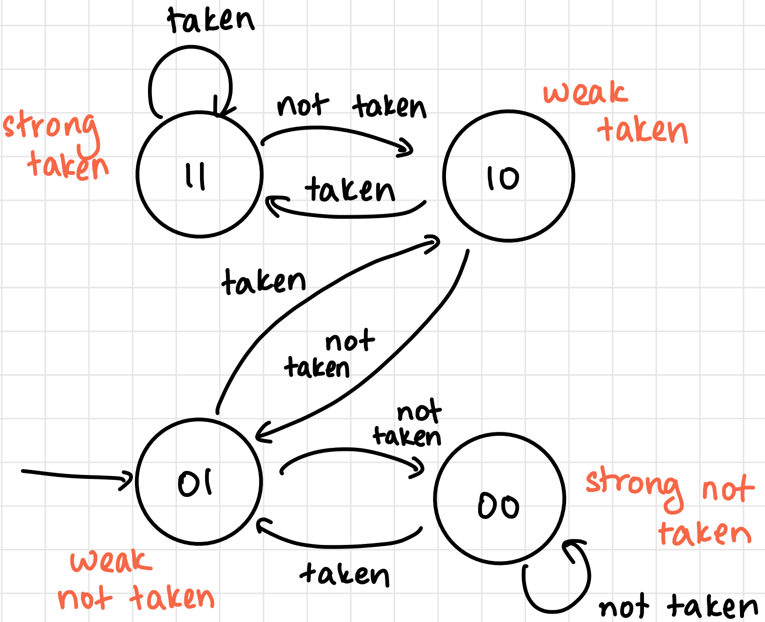
0	0	strong not taken
0	1	weak not taken
1	0	weak taken
1	1	strong taken

prediction bits

Variant 1



Variant 2



Q: Assume

(a) Pipeline contains 5 stages

(b) Each stage: 1 CC

How many CC in nonpipelined

```
      LDR R4, =A          @ A=400
loop  L1: LDR R1, [R4]
      LDR R2, [R4, #400]
      ADD R3, R1, R2
      STR R3, [R4]
      SUBS R4, R4, #4
      BNEZ R4, L1
```

$$\left(1 + 6 \times \frac{400}{4}\right) \times 5 = 3005 \text{ cc}$$

Q: Consider instr pipeline w 4 stages, stage delay = 8 ns, register delay = 0. Speedup =? 100 instr

Unpipelined = $8 \times 4 \times 100$
$= 3200$ ns

$k = 4$
$n = 100$
$t_c = 8$

Pipelined $= (k + n - 1) * t_c$
$= (4 + 99) \times 8 = 103 \times 8$
$= 824$ ns

$$\text{Speedup} = \frac{3200}{824} = 3.9$$

Q: Consider MIPS32 processor pipeline, data references = 42%, ideal CPI is 1.25 (ignoring mem structural hazard). How much faster is ideal machine without hazard vs with hazard?

$$\text{Speedup} = \frac{\text{ideal CPI} \times \text{Pipeline depth}}{\text{ideal CPI} + \text{stall cycles per instr}}$$

$$\text{speedup}_{\text{ideal}} = \frac{1.25 \times K}{1.25 + 0} = K$$

i-Cache & d-cache

$$\text{Speedup}_{\text{real}} = \frac{1.25 \times K}{1.25 + 0.42 \times 1} = \frac{1.25 \times K}{1.67}$$

$$\text{faster} = \frac{1.67}{1.25} = 1.336$$

Q: Calculate CC for execution of this segment on simple pipeline without data forwarding when result of branch instr (new PC) is available after WB stage. Show timing

L1: LDR, R1, [R4]	IF	ID	EX	M	W										
LDR R2,[R4, #400]		IF	ID	EX	M	W									
ADD R3, R1, R2			IF	ID	*	*	EX	M	W						
STR R3,[R4]				IF	*	*	ID	EX	*	M	W				
SUBS R4, R4, #4							IF	ID	*	EX	M	W			
BNEZ R4, L1								IF	*	ID	*	*	EX	M	W

Q: Same as above, with data forwarding

L1: LDR, R1, [R4]	IF	ID	EX	M	W						
LDR R2,[R4, #400]		IF	ID	EX	M	W					
ADD R3, R1, R2			IF	ID	*	EX	M	W			
STR R3,[R4]				IF	*	ID	EX	M	W		
SUBS R4, R4, #4						IF	ID	EX	M	W	
BNEZ R4, L1							IF	ID	EX	M	W

no of pipe stages

$$\text{speedup} = \frac{\text{pipeline depth}}{1 + \text{CPI Penalty}}$$

ideal CPI = 1

$$\text{speedup} = \frac{\text{pipeline depth}}{1 + \text{branch freq} * \text{branch penalty}}$$

$$\text{speedup} = \frac{\text{pipeline depth}}{1 + \% \cdot \text{branch} \cdot (\%_T \cdot \text{penalty}_T + \%_{NT} \cdot \text{penalty}_{NT})}$$