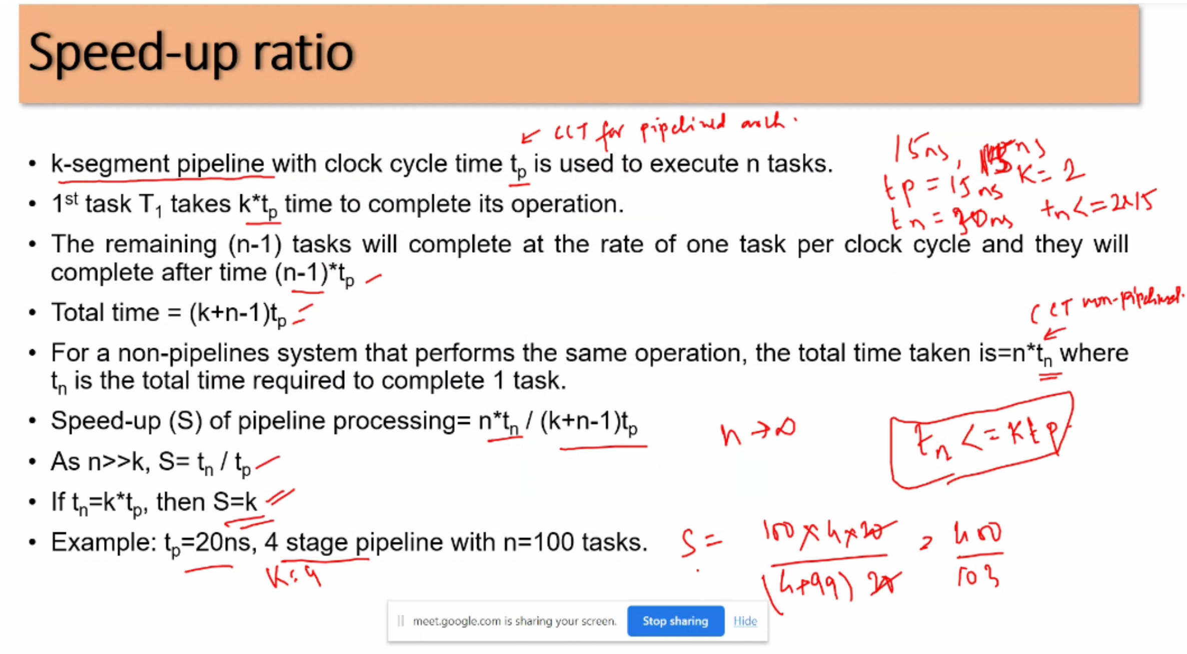


Speed-up ratio

- k-segment pipeline with clock cycle time t_p is used to execute n tasks. ← CCT for pipelined arch.
- 1st task T_1 takes $k*t_p$ time to complete its operation.
- The remaining (n-1) tasks will complete at the rate of one task per clock cycle and they will complete after time $(n-1)*t_p$
- Total time = $(k+n-1)t_p$
- For a non-pipelines system that performs the same operation, the total time taken is=$n*t_n$ where t_n is the total time required to complete 1 task.
- Speed-up (S) of pipeline processing= $n*t_n / (k+n-1)t_p$
- As n>>k, S= t_n / t_p
- If $t_n=k*t_p$, then S=k
- Example: t_p=20ns, 4 stage pipeline with n=100 tasks.

15ns, ~~15~~ns

$t_p = 15$ns $k=2$

$t_n = 30$ns $t_n <= 2\times 15$

CCT non-pipelined.

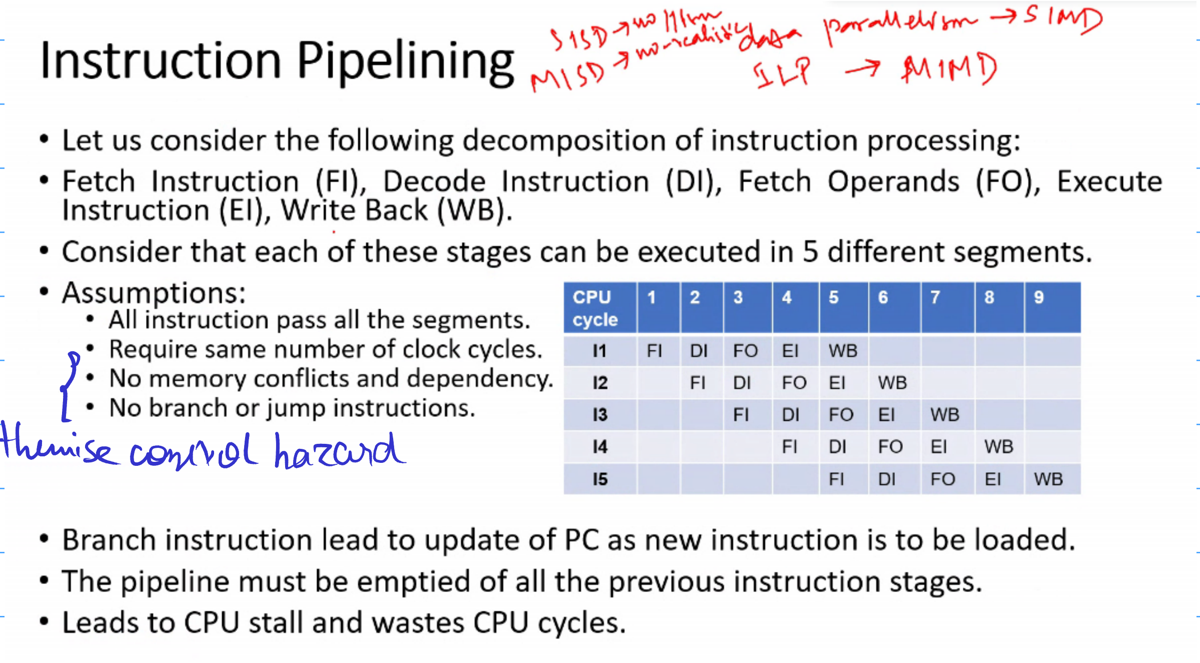
$n \to \infty$

$\boxed{t_n <= k t_p}$

$k=4$

$$S = \frac{100 \times 4 \times 20}{(4+99)\,20} = \frac{400}{103}$$

meet.google.com is sharing your screen. Stop sharing Hide



Instruction Pipelining

SISD → no ||lism

MISD → no-reality

data parallelism → SIMD

ILP → MIMD

- Let us consider the following decomposition of instruction processing:
- Fetch Instruction (FI), Decode Instruction (DI), Fetch Operands (FO), Execute Instruction (EI), Write Back (WB).
- Consider that each of these stages can be executed in 5 different segments.
- Assumptions:
  - All instruction pass all the segments.
  - Require same number of clock cycles.
  - No memory conflicts and dependency.
  - No branch or jump instructions.

otherwise control hazard

CPU cycle	1	2	3	4	5	6	7	8	9
I1	FI	DI	FO	EI	WB				
I2		FI	DI	FO	EI	WB			
I3			FI	DI	FO	EI	WB		
I4				FI	DI	FO	EI	WB	
I5					FI	DI	FO	EI	WB

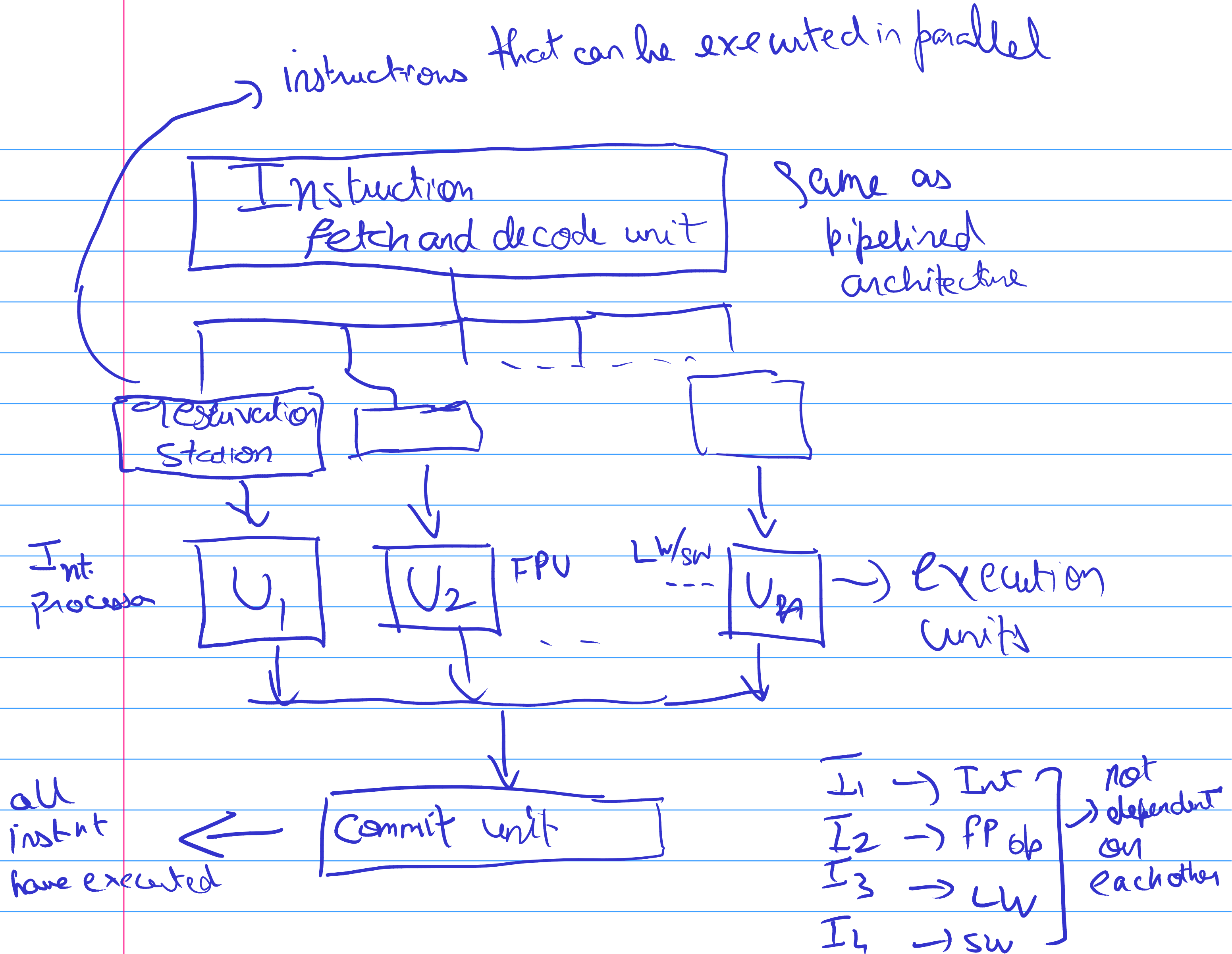
- Branch instruction lead to update of PC as new instruction is to be loaded.
- The pipeline must be emptied of all the previous instruction stages.
- Leads to CPU stall and wastes CPU cycles.

① Single cycle arch. $CPI_{avg} > 1$

② Multicycle arch. $CPI_{avg} > 1$

③ Pipelining $CPI_{avg} = 1$ (ideally)

④ Superscalar $CPI_{avg} < 1$


instructions that can be executed in parallel
Instruction fetch and decode unit
Same as pipelined architecture
reservation station
Int. Processor
U_1
U_2
FPU
LW/SW
U_n
execution units
Commit unit
all instrt have executed

I_1 → Int
I_2 → FP op
I_3 → LW
I_4 → SW

→ not dependent on eachother

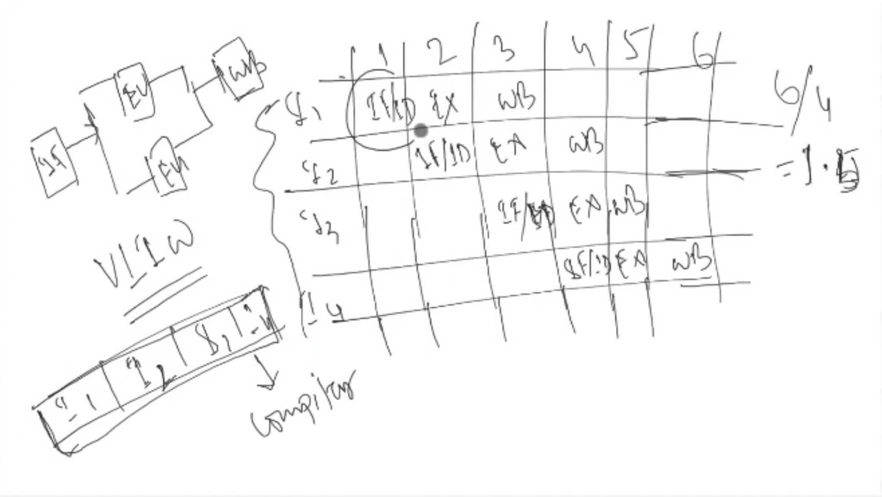
Reservation station fetches the operands
↳ instruction reserved for execution in that unit

if inst has to wait for operand to be generated it will do so in reservation station

once generated, it goes directly to reservation station, not main memory

How is CPI < 1 (PTO)?

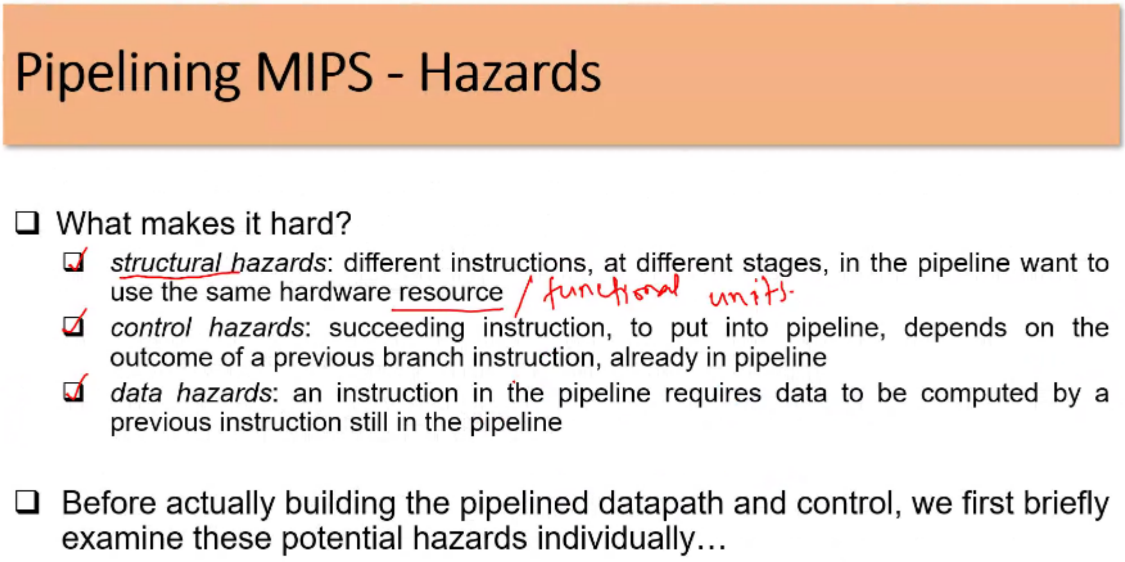
How does a single fetch help!?



You can fetch the Very Large Instruction Word

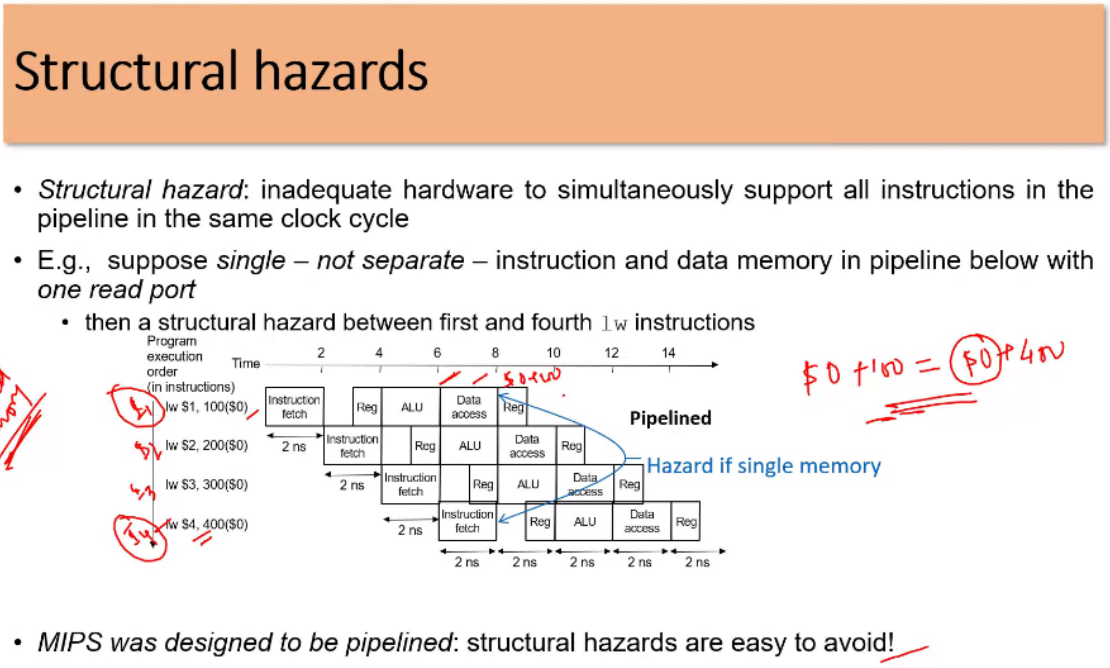
and then do all of them at once!

$CPI = 3/4 = 0.75 < 1$



Pipelining MIPS - Hazards

- ❑ What makes it hard?
  - ☑ *structural hazards*: different instructions, at different stages, in the pipeline want to use the same hardware resource / functional units.
  - ☑ *control hazards*: succeeding instruction, to put into pipeline, depends on the outcome of a previous branch instruction, already in pipeline
  - ☑ *data hazards*: an instruction in the pipeline requires data to be computed by a previous instruction still in the pipeline
- ❑ Before actually building the pipelined datapath and control, we first briefly examine these potential hazards individually…



Structural hazards

- *Structural hazard*: inadequate hardware to simultaneously support all instructions in the pipeline in the same clock cycle
- E.g., suppose *single – not separate* – instruction and data memory in pipeline below with *one read port*
  - then a structural hazard between first and fourth `lw` instructions

$0 + 100 = $\boxed{\$0}$ + 400

- *MIPS was designed to be pipelined*: structural hazards are easy to avoid!

(Just by using 2 fkin read ports)

Data Hazards

I1 lw $1, 12($2) → $1 is written

I2 add $3, $1, $5 → $1, $5 is read
$3 is written

I3 lw $1, 12($2) → $1 is written

I4 add $5, $1, $2 → $1, $2 is read and
$5 is written

① RAW → $1 read after write } true hazard

② WAR → $5 write after read } Pseudo/Fake hazard

③ WAW → $1 write after write

(that's why in superscalar arch we need independent instructions)

Data hazard resolution

① Static instruction scheduling → Compiler based (VLIW)

② Dynamic instruction scheduling → HW based (Tomasulo's algo) registers renaming +

WTF... just missed it next class ig

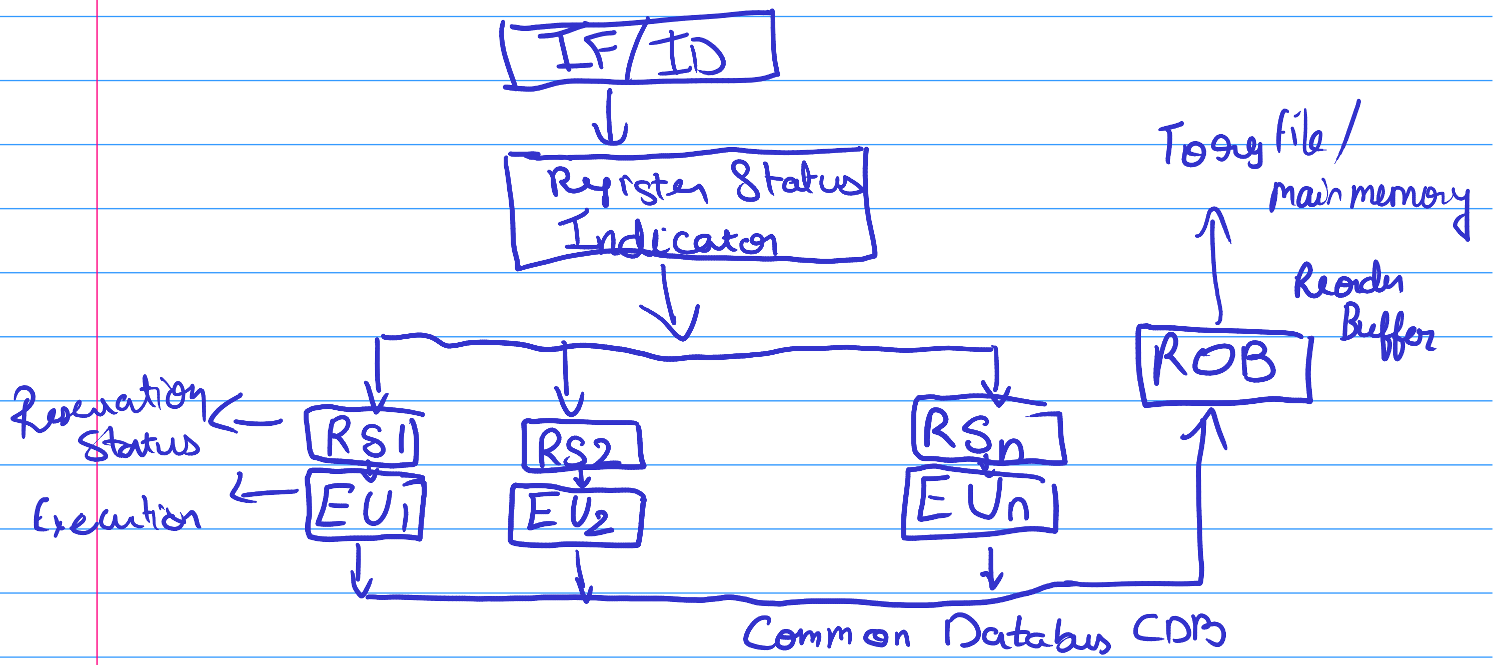
(3) Operand forwarding

RAW can only be avoided by a nop.

Next class

WAR]
WAW] — Pseudo dependency

Dynamic Scheduling



★ Registers are fetched one by one, and decoded to find the type of operation and source of operands

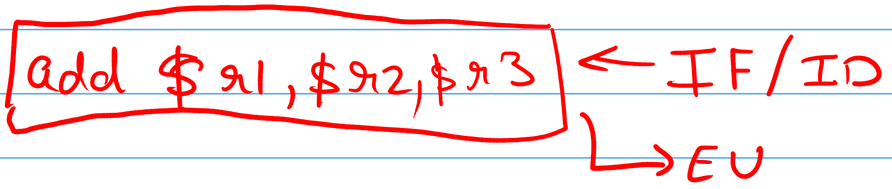
=> in seperate cycle

★ Register status indicator indicates whether the latest val of reg is in the reg file or currenty being computed by some execution unit and if the latter, it states the EV number.

If all the operands are avail., the operation proceeds in the alloted EU, else it waits in the reservation station of the alloted EU piping the CDB.

→ Every EU writes the results along with the unit number on to the CDB which is forwarded to all reservation stations, reg-file & main memory.

Step I



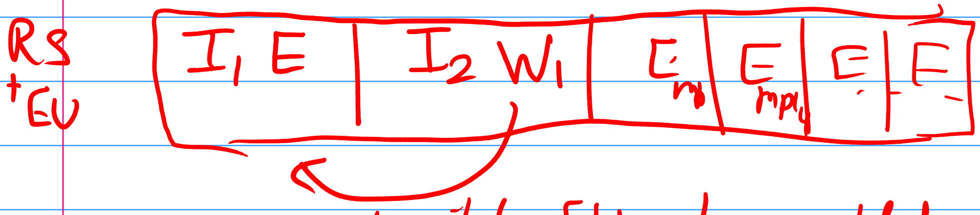
r_1	r_2	r_3	r_4	r_5	r_6	r_7	r_8	r_9	r_{10}
1	0	0	0	0	0	0	0	0	0

← Reg status

EU_1 ←	RS IE	Empty	Empty	- -

Step-2: sw $r1, 12($r4) (r_1 is not available)

r_1	r_2	r_3	...	r_{10}
1	0	0		0



wait for EU_1 to complete, pipes the common DBus add $r1, $r5, $r6

r_1	r_2	r_3	r_4	r5	r6	r7	r8	r9	r10
3	0	0	0	0	0	0	0	0	0

I_1Ex	I_2 W_1	I_3 Ex	- - —

Step-4 Sub $r7, $r1, $r8

1	2	3	4	5	6	7	8	9	10
3	0	0	0	0	0	4	0	0	0

I_1Ex	I_2 W_1	I_3Ex	I_4 W_3	Empty	. . . .

Step5 sw $r1, 16($r4)

1	2	3	4	5	6	7	8	9	10
3	0	0	0	0	0	4	0	0	0

I_1 Ex	$I_2$$W_1$	I_3 Ex	$I_4$$W_3$	I_5 W_3	Emp

I_6 add $r1, $r9, $r10

Latest value of $r1 ←	1	2	3	4	5	6	7	8	9	10
	6	0	0	0	0	0	4	0	0	0

I_1 Ex	I_2 W_1	I_3 Ex	I_4 W_8	I_5 W_3	I_6 Ex

I_1, I_3, I_4 are in execution state

I_2, I_4, I_5 are in wait state

true dependency ← RAW – operand forwarding
resolved through

WAR, WAW

```
E1 generates results ←  add  $r1, $r3, $r3  ] RAW
                        sw   (E1) , 12($r4) } WAR
                        add  $r1, $r5, $r6
                        sub  $r7, (E3), $r8
                        sw   (E3), 16($r4)
                        add  $r1, $r9, $r10
```

⇒ Forward the results directly we are not waiting for the write to actually occur!

register renaming ↓

Rename those registers waiting for the Execution unit that is going to generate the result.

Part of Tomasulo's algorithm }

WAR is resolved in the reorder buffer, he will explain next class!

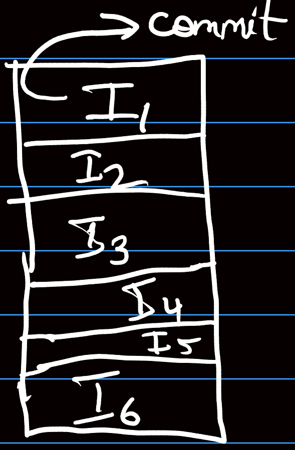
How to maintain the sequence of execution?

WAR

Reorder Buffer

* Writing the final value into the register is called a "commit"

\\ Read this part of lec again plox



* Lets say I_1 & I_6 finish at the same time

* They are committed

* rest aren't, however!

We read from EU, by bizzing the common Dbus to check if the previous instruction completes!

Value is written to reservation station & registerfile

In the previous example, I_2 will never read the value of EU_3

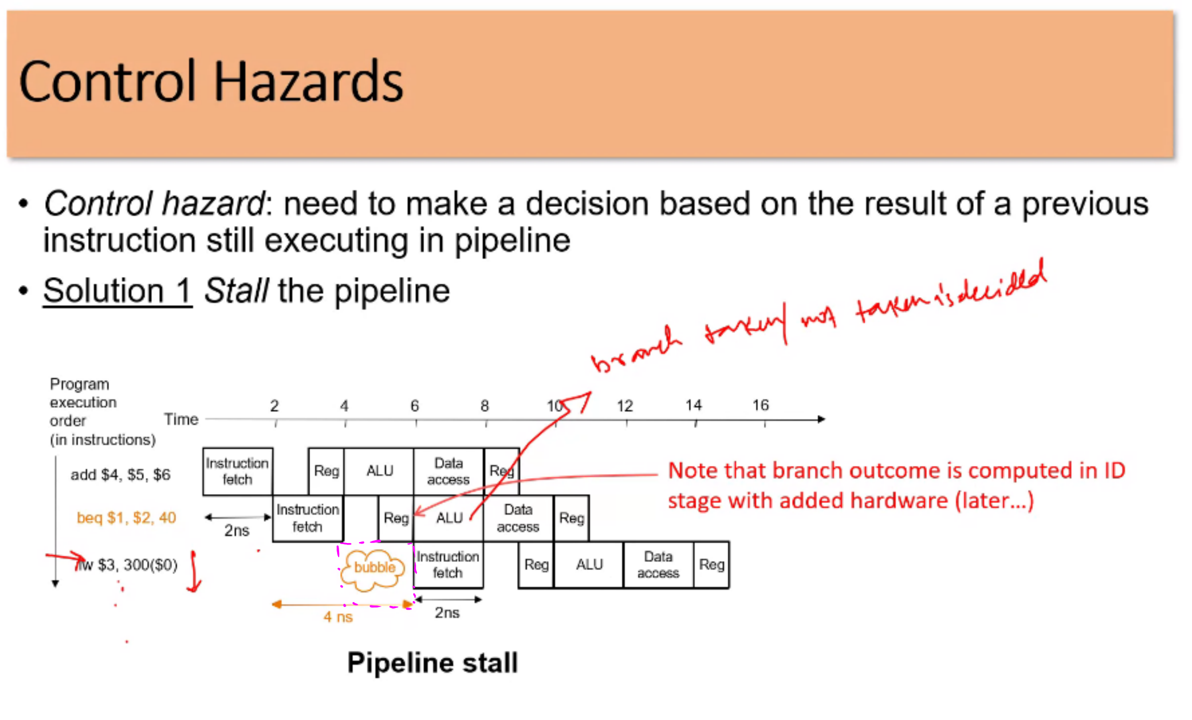
So even if I_3 completes first, I_2 will never read it.

⇒ WAR hazard also satisfied

Control Hazard

- ✯ Problem is not with unconditional branches [we already know where to go]
- ✯ Problem is with conditional jump.

Program execution order (in instrns)



Control Hazards

- *Control hazard*: need to make a decision based on the result of a previous instruction still executing in pipeline
- Solution 1 *Stall* the pipeline

Note that branch outcome is computed in ID stage with added hardware (later...)

Pipeline stall

① Stall

② Predict branch is not-taken (basically be pessimistic)

- ↳ exec the successor instr in sequence
- ↳ "squash" instruction in pipeline if branch is actually taken (remove from pipeline and also clean out regs)

[flush pipelines and clean out regs too]

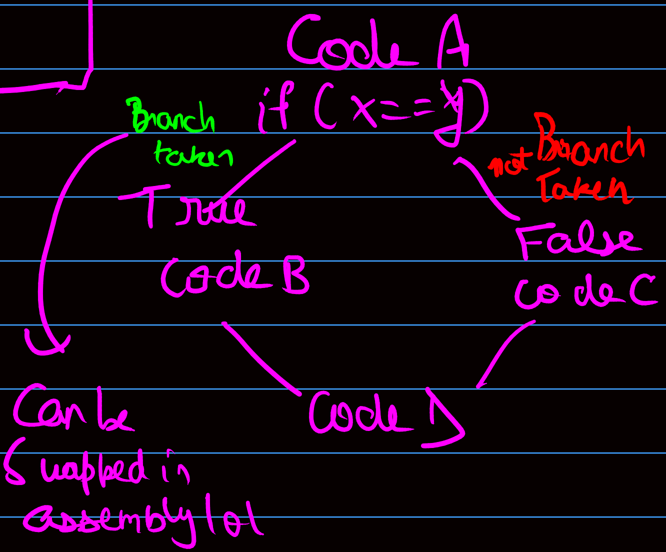
MIPS never has to revert as no instruction reaches exec stage by the time branch is determined

③ Predict branch address

④ Delayed branch

④ Code A

if (x==y) then
  code B
else
  code C
code D



I_1
I_2
I_3
I_4
} The number of instructions there in the pipeline, till the branch taken or not taken is decided, is called the delay slot

Which instructions should I execute in branch delay?
You have many options

Eg: MIPS has just 1 delay slot, so for simplicity lets take that.

We can put instructions that we want to execute after the branch

[shift previous instr'n before branch to branch delay slot]

Option ①

```
        add  $1, $2, $3
        beq  $4, $5, L1
true  [ ______________ ]  ← Some instruction
      → L1:
```

(this will go into branch delay)

if true, then it'll go waste, we need to flush & revert. So we just put that add instruction above it, since there's no dependency there

```
      beq $4, $5, L1
True  [ add $1, $2, $3 ]
      → L1:
```

This is done by compiler

But!

```
add $1, $2, $3  ]
beq $1, $6, L1  ]  → RAW
[ ______________ ]  → Now we can't do that
→ L1
```

Option ② We insert the first instruction of Branch is taken [TARGET ADDR]

Here also we need to check if we are changing the state in such a way that the behavior is affected if branch isn't taken

Option (3)

(3) [Fall through]

We take the instruction for branch not taken

⇒ We face a similar problem as option (2) if branch is taken.

We can choose either options (1), (2) or (3) for the branch delay slot.