CPU performance, depending on User CPU time

Remember: performance of a computer only determined by User CPU time.

$$\Rightarrow \text{User CPU performance} = \frac{1}{\text{User CPU Time}}$$

$$\rightarrow \text{User CPU time} = \text{CPU clock cycles} \times \text{cycle Time}$$

$$= \frac{\text{CPU clock cycles}}{\text{clock Rate}}$$

User CPU time = cycles x cycle Time

= cycles / clock Rate.

$$\text{cycle time} = \frac{1}{\text{Rate}}$$ → Determined by the hardware.

$$\text{Rate} = \frac{1}{\text{cycle time}}$$ → Determined by the hardware.

clock cycles = Instruction count x cycles per Instruction

→ cycles per Instruction: Determined by the Hardware.

Remember

$$\text{cycles} = IC_{total} \times CPI_{average}$$

→ Instruction count: Determined by - program
- ISA
- compiler.

Then weighted CPU Average:

$$CPI = \frac{\text{clock cycles}}{\text{Instruction count}} = \sum_{i=1}^{n} \left(CPI_i \times \frac{IC_i}{IC_{total}} \right)$$

or → $$\text{cycles} = \sum_{i=1}^{n} (CPI_i \times \text{Instruction count})$$ for each instruction type.

Building CPU by Pipelining

Pipelining → Parallelism + Overlapping

كل new cycle نجيب instruction جديدة

Pipelining Analogy :-

عشان احدد ال speedup = $\frac{\text{الوقت القديم}}{\text{وقت ال pipelining}}$

① for specific number of instructions :-

pipelining time = time for first instructio + final stage time × number of the rest of instructions.

② for Non-stop instructions :-

Time = time for first instruction + (n−1) × time of final stage

↳ $n = \infty$

So speedup = number of stages

↳ $\lim_{n \to \infty} = \frac{\text{Instruction time (not pipelined)} \times 2}{\text{Instruction time pipelined} + (n-1)\ \text{stage time}}$

↳ for balanced this will result with the number of stages.

↳ for not balanced, this will result with less

* لو كان ال stages ← مش كلهم نفس ال Clock period
بخليهم كلهم عند تنفيذ ياخدوا زمن اطول stage
← imbalanced stages.

Speedup

total time of pipelined is less than single-cycle. knowing that the time of one instruction may increase due to balancing & other things.

All stages balanced

→ all take same time.

Number of stages = speedup = $\frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Time between instructions}_{\text{pipelined}}}$

time between instructions: الوقت اللي بخلص فيه عشان ابدأ instruction جديدة

All stages are not balanced.

→ all takes same time except some stages.

speedup is less than number of stages.

* speedup is due to increasing throughput
* latency (time for each instruction) does not decrease

Conditions of Max speedup (= number of stages)

1. stages are balanced
2. infinite excution
3. No. Hazards

Scanned with CamScanner

Hazards

[اي مشكلة تمنعني اني اعمل Fetch لـ new instruction
عال cycle جديدة نسميها Hazard.

[1] Structure Hazards

two instructions trying to use the same resource

if happened, new instruction with the stage of hazard would have to stall for that cycle (pipeline bubble)
(can't fetch new instruction)

→ can be solved by
- adding new resources
- OR
- waiting.

→ it never happens in RISC-V pipeline design

[2] Data Hazards

Types:

[1] Read After Write (RAW)
↳ the problem is it reads before it writes
(reads the old value while should read the new written value)

[2] Write After Read (WAR)

[3] Write After write (WAW)

Type ① only happens in RISC-V

[1] first solution for Data Hazard:-

remember [Read After write] Hazard only Happens.

* Pipelined must be stalled
* two lines of bubbles (2 bubbles at the end)
↳ calculated in time.

maximum
↳ when there is two stages shift between read and write

↳ not 3 bubbles due to Internal forwarding.

[2] Second Solution for Data Hazard.

[Forwarding or Bypassing]

→ to use the result directly when it's computed.
→ distination stage should be later in time than the source stage.

* Can't always avoid stalls by forwarding.
→ if the wanted value is not yet computed when needed, but you can't forward back
So stall the forward.

Example:- Load-use Data Hazard

Note:- for Read After Write Hazard when you have (write instruction from R-type, you don't need to stall)

example:-

add F D E M W

F D E M W

you only forwarded.

add x3, x1, x2

sd x3, 24(x0)

* Code scheduling can be used to Avoid stalls

→ try to put load instructions first

→ then any arithemetic or logical operations.

→ then store instructions.

↓

avoid:

load

↓

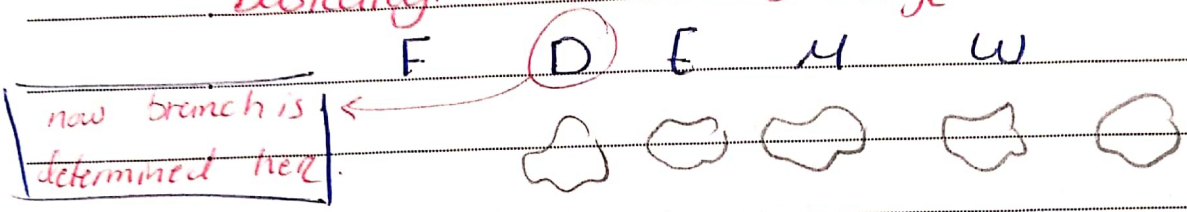
any instruction depends on it directly in the next line.

[3] Control Hazard

branch instruction does two things

* determine which address of instructions to follow (PC+4, target address)
  - if not taken
  - if taken

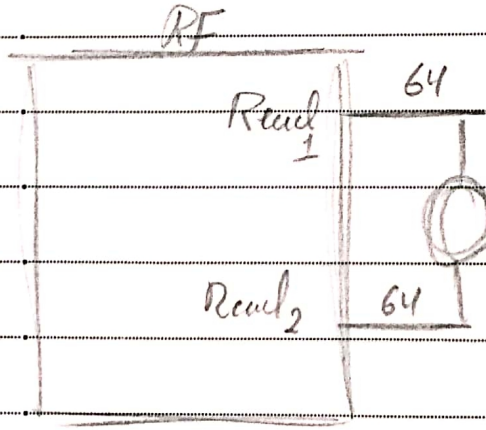
Basically work on ID stage



instruction ← F
excuted after branch.

Hard? ① compare registers and compute the target early in the pipeline.

(+ to add hardware to do it in ID stage



But then still need bubble

[1] First solution: Stall on Branch

wait until branch outcome determined before fetching next instruction

fetch PC+4 or fetch target

→ Determine to take or not to take at ID
→ calculate next fetch address in E
→ one bubble is needed

[2] Second Solution: Prediction

* principle: ① prediction correct: fetch directly
② " uncorrect: you lose a cycle with one bubble to [redacted]. fetch then

* principle: you either Predict to take (taken branch) or Predict not to take (not taken)

[Way 1] Static prediction: [branch behavior]

[Way 2] Dynamic prediction [behavior History]