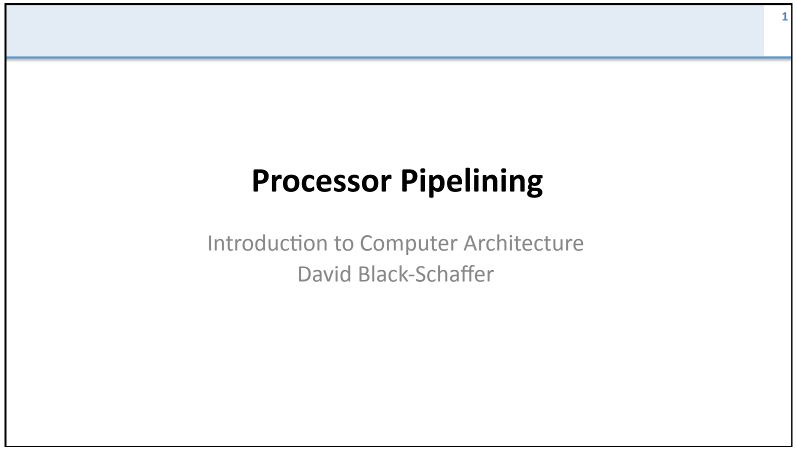
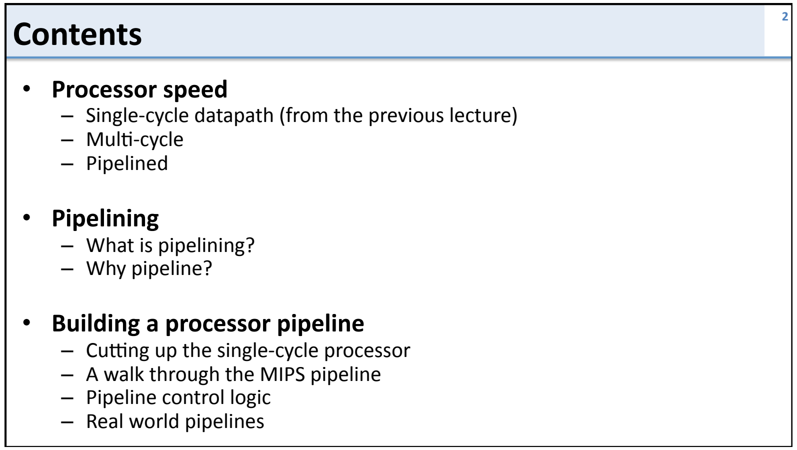


1

Processor Pipelining

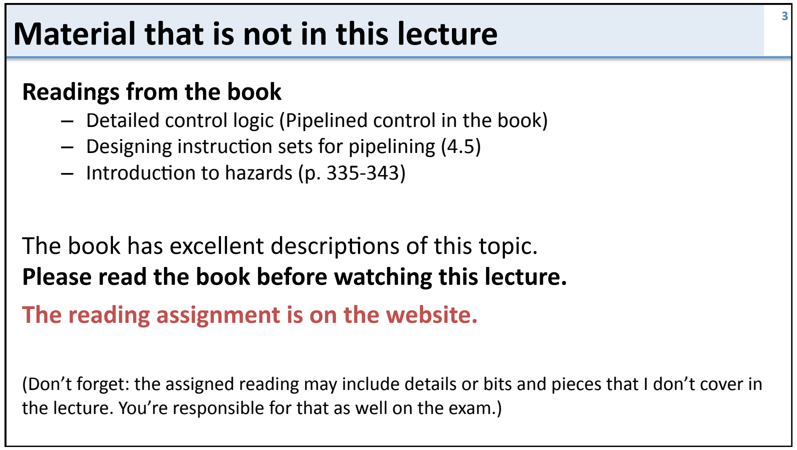
Introduction to Computer Architecture
David Black-Schaffer



2

Contents

- **Processor speed**
  - Single-cycle datapath (from the previous lecture)
  - Multi-cycle
  - Pipelined
- **Pipelining**
  - What is pipelining?
  - Why pipeline?
- **Building a processor pipeline**
  - Cutting up the single-cycle processor
  - A walk through the MIPS pipeline
  - Pipeline control logic
  - Real world pipelines



3

Material that is not in this lecture

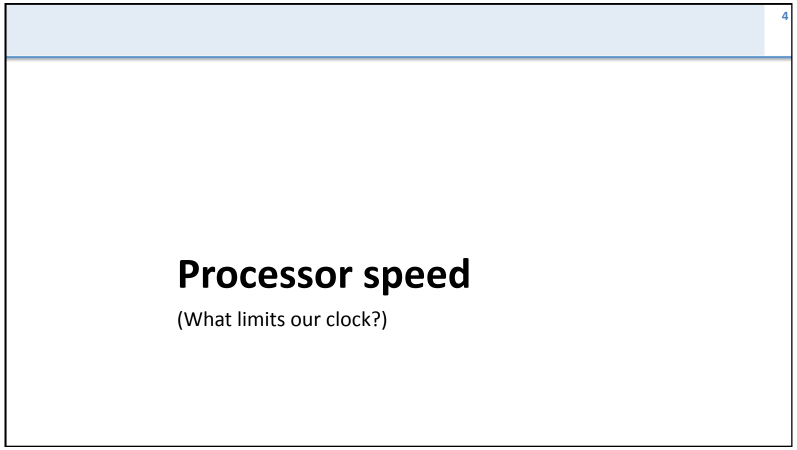
Readings from the book

- Detailed control logic (Pipelined control in the book)
- Designing instruction sets for pipelining (4.5)
- Introduction to hazards (p. 335-343)

The book has excellent descriptions of this topic.
Please read the book before watching this lecture.

The reading assignment is on the website.

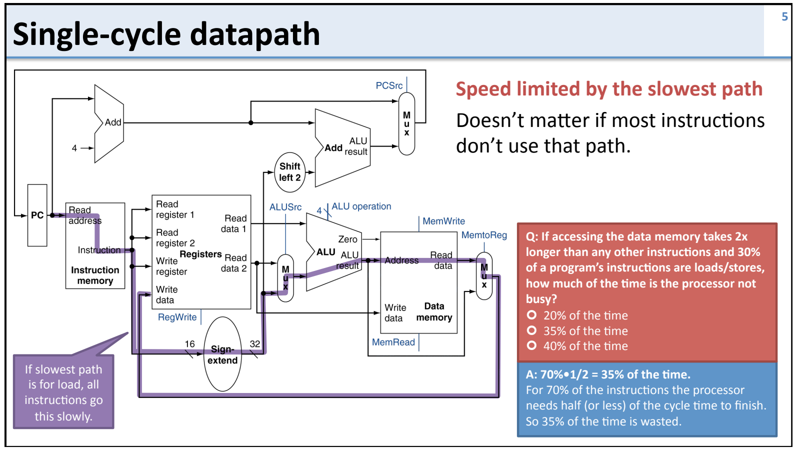
(Don’t forget: the assigned reading may include details or bits and pieces that I don’t cover in the lecture. You’re responsible for that as well on the exam.)



4

Processor speed

(What limits our clock?)



Single-cycle datapath

Speed limited by the slowest path

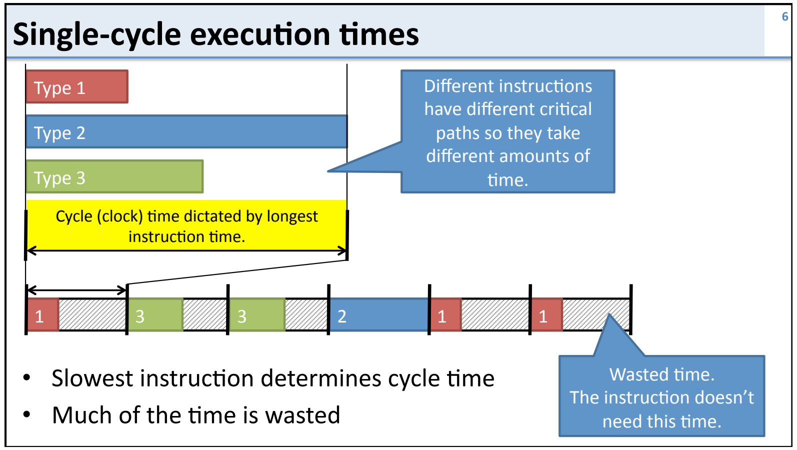
Doesn't matter if most instructions don't use that path.

If slowest path is for load, all instructions go this slowly.

Q: If accessing the data memory takes 2x longer than any other instructions and 30% of a program's instructions are loads/stores, how much of the time is the processor not busy?

○ 20% of the time
○ 35% of the time
○ 40% of the time

A: 70%•1/2 = 35% of the time.
For 70% of the instructions the processor needs half (or less) of the cycle time to finish. So 35% of the time is wasted.



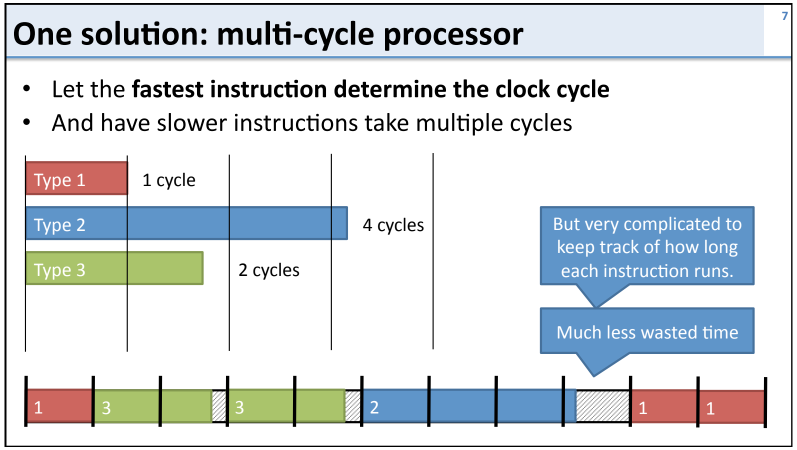
Single-cycle execution times

Different instructions have different critical paths so they take different amounts of time.

Cycle (clock) time dictated by longest instruction time.

- Slowest instruction determines cycle time
- Much of the time is wasted

Wasted time. The instruction doesn't need this time.

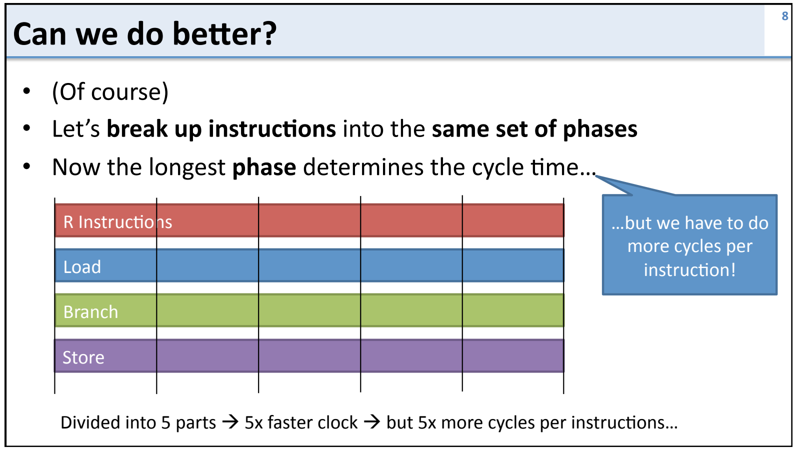


One solution: multi-cycle processor

- Let the **fastest instruction determine the clock cycle**
- And have slower instructions take multiple cycles

But very complicated to keep track of how long each instruction runs.

Much less wasted time

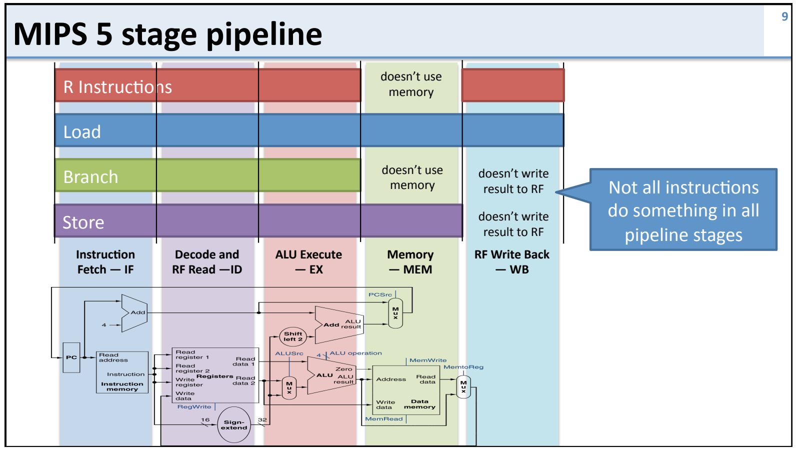


Can we do better?

- (Of course)
- Let's **break up instructions** into the **same set of phases**
- Now the longest **phase** determines the cycle time…

…but we have to do more cycles per instruction!

Divided into 5 parts → 5x faster clock → but 5x more cycles per instructions…

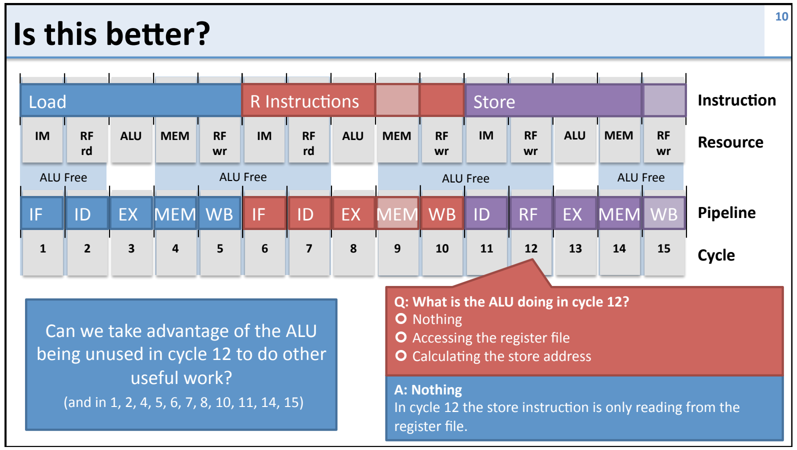


MIPS 5 stage pipeline

9

Instruction Fetch — IF	Decode and RF Read —ID	ALU Execute — EX	Memory — MEM	RF Write Back — WB
R Instructions			doesn’t use memory	
Load				
Branch			doesn’t use memory	doesn’t write result to RF
Store				doesn’t write result to RF

Not all instructions do something in all pipeline stages



Is this better?

10

Load					R Instructions					Store					Instruction
IM	RF rd	ALU	MEM	RF wr	IM	RF rd	ALU	MEM	RF wr	IM	RF wr	ALU	MEM	RF wr	Resource
ALU Free			ALU Free					ALU Free					ALU Free		
IF	ID	EX	MEM	WB	IF	ID	EX	MEM	WB	ID	RF	EX	MEM	WB	Pipeline
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Cycle

Can we take advantage of the ALU being unused in cycle 12 to do other useful work?

(and in 1, 2, 4, 5, 6, 7, 8, 10, 11, 14, 15)

Q: What is the ALU doing in cycle 12?

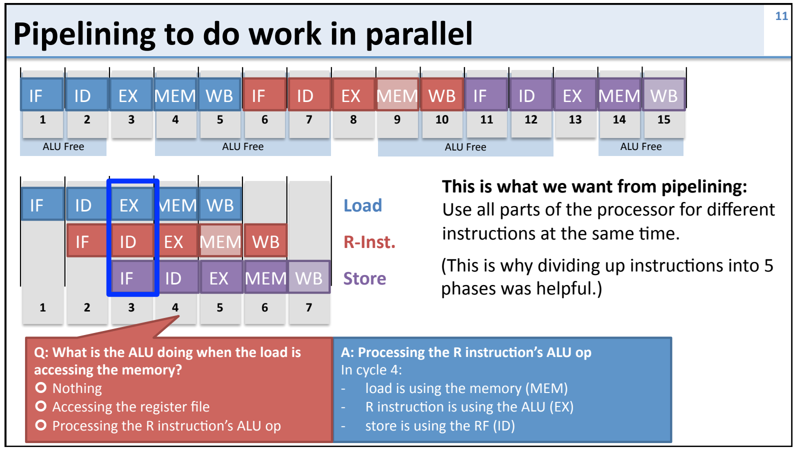
○ Nothing

○ Accessing the register file

○ Calculating the store address

A: Nothing

In cycle 12 the store instruction is only reading from the register file.



Pipelining to do work in parallel

11

IF	ID	EX	MEM	WB	IF	ID	EX	MEM	WB	IF	ID	EX	MEM	WB
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
ALU Free			ALU Free					ALU Free					ALU Free	

IF	ID	EX	MEM	WB			Load
	IF	ID	EX	MEM	WB		R-Inst.
		IF	ID	EX	MEM	WB	Store
1	2	3	4	5	6	7	

This is what we want from pipelining: Use all parts of the processor for different instructions at the same time.

(This is why dividing up instructions into 5 phases was helpful.)

Q: What is the ALU doing when the load is accessing the memory?

○ Nothing

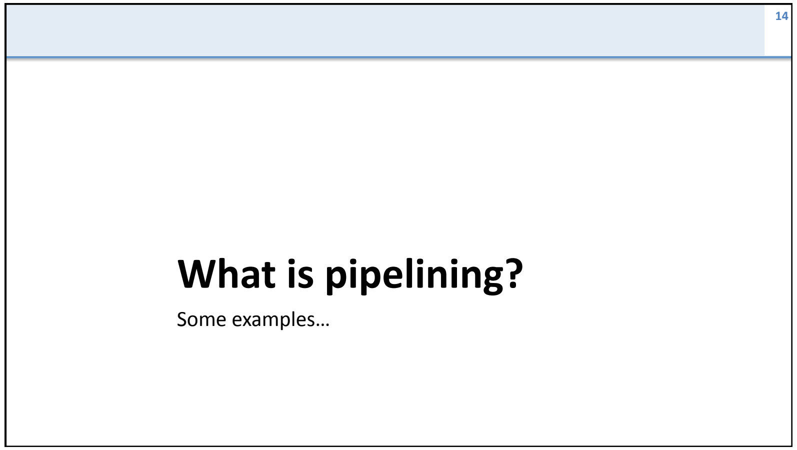
○ Accessing the register file

○ Processing the R instruction’s ALU op

A: Processing the R instruction’s ALU op

In cycle 4:

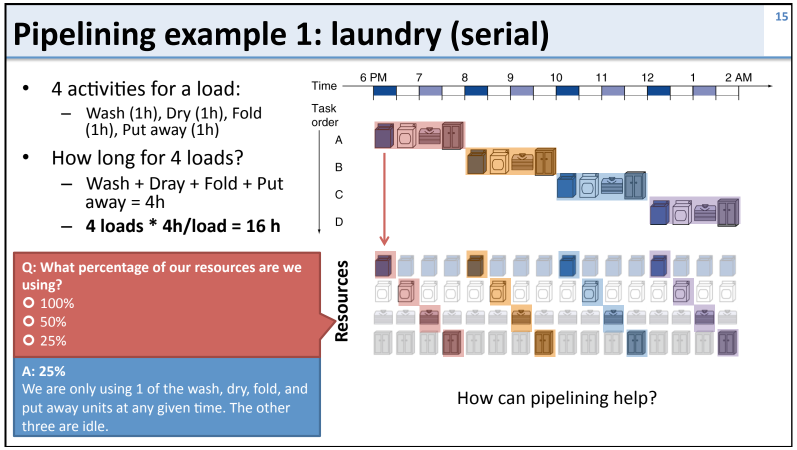
- load is using the memory (MEM)
- R instruction is using the ALU (EX)
- store is using the RF (ID)



14

What is pipelining?

Some examples…



Pipelining example 1: laundry (serial)

15

- 4 activities for a load:
  - Wash (1h), Dry (1h), Fold (1h), Put away (1h)
- How long for 4 loads?
  - Wash + Dray + Fold + Put away = 4h
  - **4 loads * 4h/load = 16 h**

Q: What percentage of our resources are we using?

○ 100%

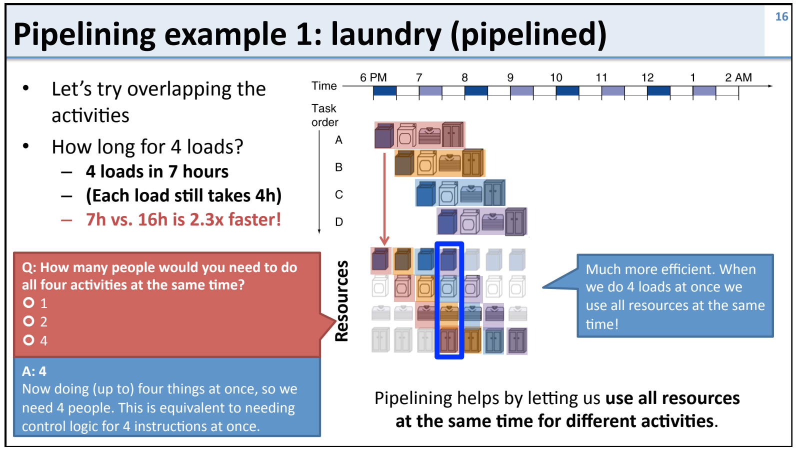
○ 50%

○ 25%

A: 25%

We are only using 1 of the wash, dry, fold, and put away units at any given time. The other three are idle.

How can pipelining help?



Pipelining example 1: laundry (pipelined)

16

- Let's try overlapping the activities
- How long for 4 loads?
  - **4 loads in 7 hours**
  - **(Each load still takes 4h)**
  - **7h vs. 16h is 2.3x faster!**

Q: How many people would you need to do all four activities at the same time?

○ 1

○ 2

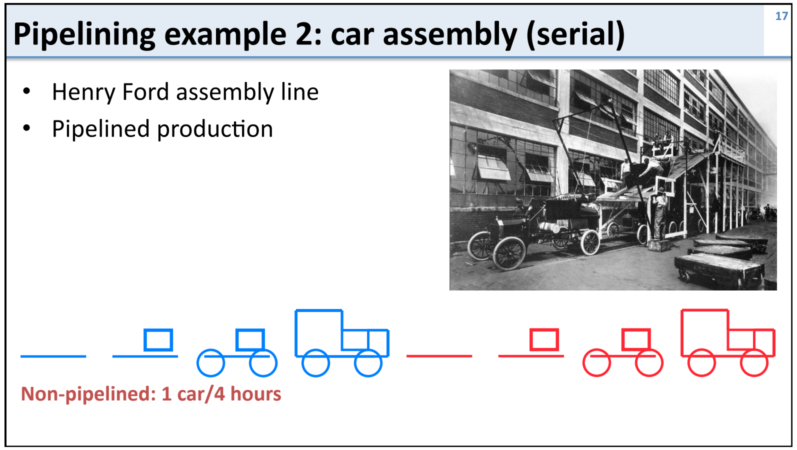
○ 4

A: 4

Now doing (up to) four things at once, so we need 4 people. This is equivalent to needing control logic for 4 instructions at once.

Much more efficient. When we do 4 loads at once we use all resources at the same time!

Pipelining helps by letting us **use all resources at the same time for different activities**.

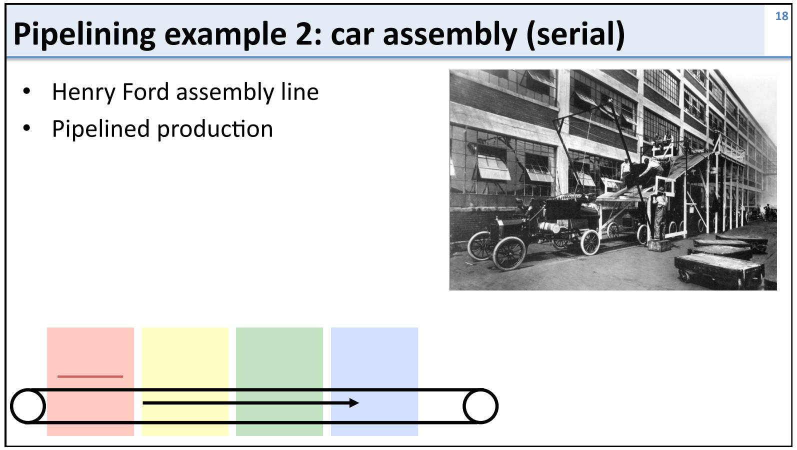


Pipelining example 2: car assembly (serial)

17

- Henry Ford assembly line
- Pipelined production

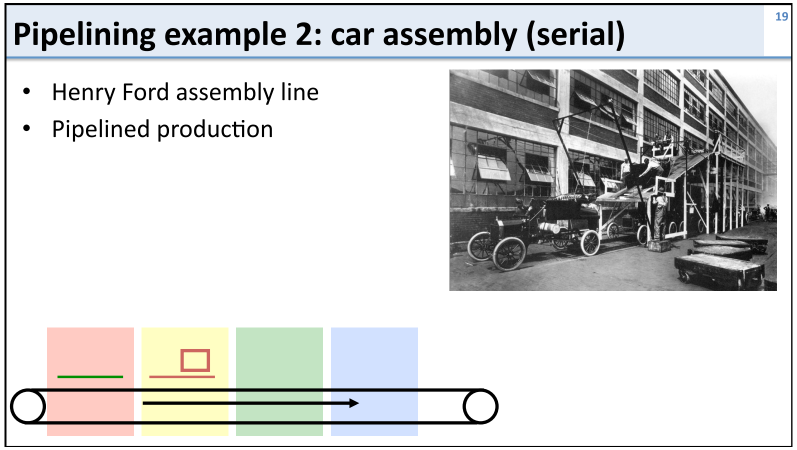
Non-pipelined: 1 car/4 hours



Pipelining example 2: car assembly (serial)

18

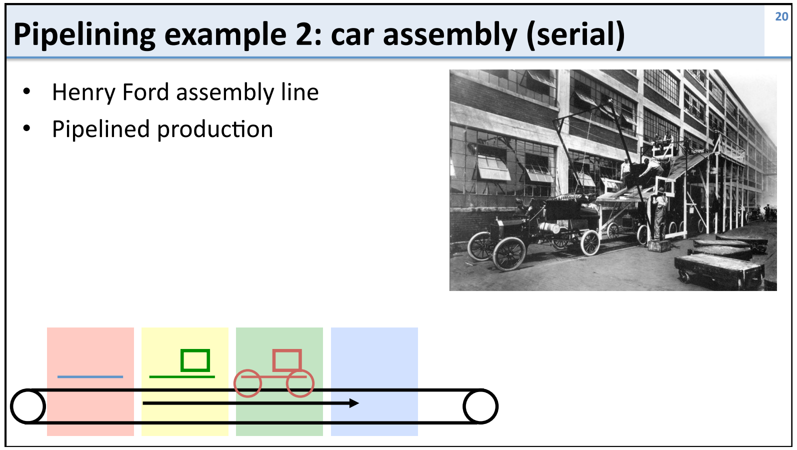
- Henry Ford assembly line
- Pipelined production



Pipelining example 2: car assembly (serial)

19

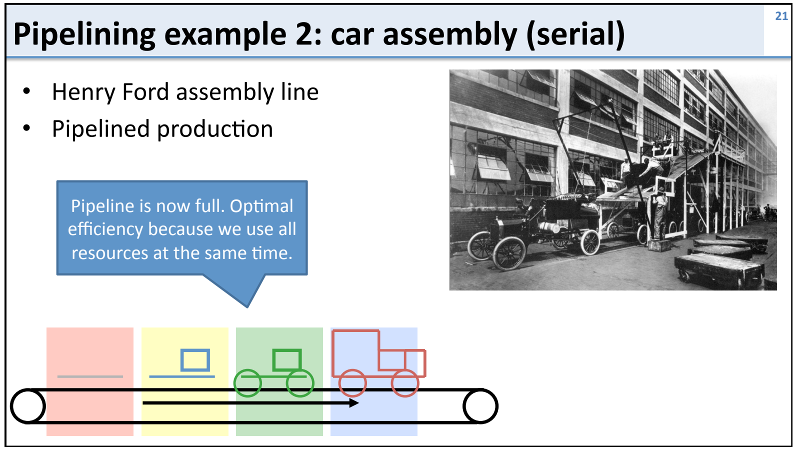
- Henry Ford assembly line
- Pipelined production



Pipelining example 2: car assembly (serial)

20

- Henry Ford assembly line
- Pipelined production

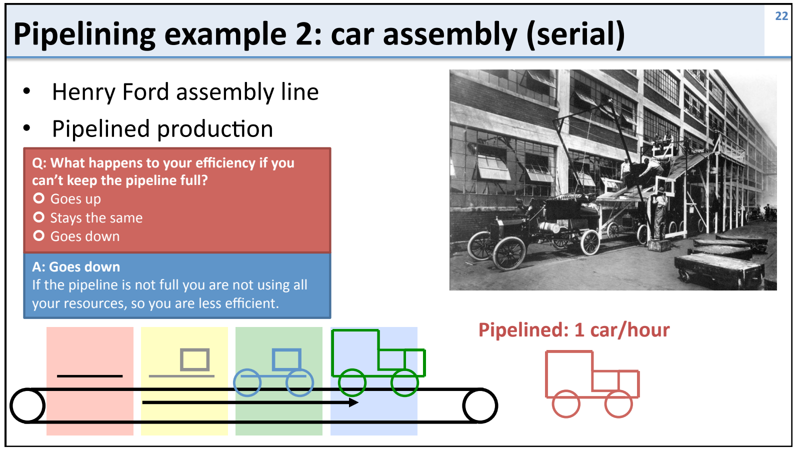


Pipelining example 2: car assembly (serial)

21

- Henry Ford assembly line
- Pipelined production

Pipeline is now full. Optimal efficiency because we use all resources at the same time.



Pipelining example 2: car assembly (serial)

22

- Henry Ford assembly line
- Pipelined production

Q: What happens to your efficiency if you can't keep the pipeline full?

○ Goes up
○ Stays the same
○ Goes down

A: Goes down

If the pipeline is not full you are not using all your resources, so you are less efficient.

Pipelined: 1 car/hour

Why pipeline?

(Hint: performance)

Why pipeline?

- If we can keep the pipeline full we get better **throughput** (per time)
  - Laundry: 1 load of laundry/hour
  - Car: 1 car/hour
  - MIPS: 1 instruction/cycle
- But, we have the same **latency** (total time per)
  - Laundry: 4 hours for each load of laundry
  - Car: 4 hours for each car
  - MIPS: 5 cycles for each instruction
- Pipelining is faster because we **use all resources at the same time**
  - Laundry: Washer, dryer, folding, and closet
  - Car: Base assembly, engine assembly, wheel assembly, cab assembly
  - MIPS: Instruction fetch, register read, ALU, memory, and register write
- But, it only works if we **keep the pipeline full**!
  - Empty slots mean unused resources (this is the hard part in reality)

Pipelining performance in processors

- Look at a program of three load instructions
- Each takes 800ps (0.8ns)
- But if we pipeline and overlap so **we use all resources in parallel** we can finish faster

Q: What is the *throughput* speedup due to this 5-stage pipeling?

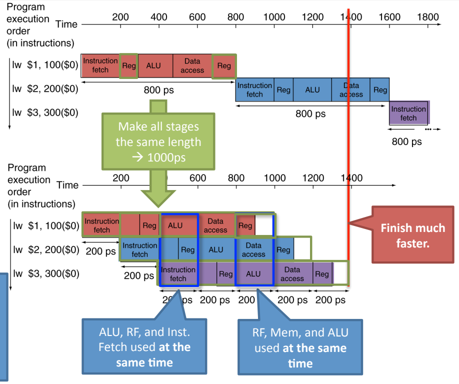
○ 1.7x

○ 4x

○ 5x

A: 4x

With the pipeline the **throughput** is one instruction every 200ps vs. 800ps without it. However, we had to increased the **latency** to 1000ps per instruction to balance the 5 pipeline stages. The **absolute speedup** for these three particular instructions is 1.7x (1400ps/2400ps).



Make all stages the same length → 1000ps

Finish much faster.

ALU, RF, and Inst. Fetch used **at the same time**

RF, Mem, and ALU used **at the same time**

How much faster?

- **Pipeline speedup**
  - If all the stages are the same length (e.g., balanced)

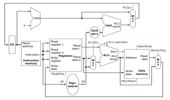
$$\text{Time per finished unit pipelined} = \frac{\text{Time per finished unit non-pipeline}}{\text{Number of pipeline stages}}$$

- **Example: Pipelined**
  - Time per laundry load = 4h/4 stages = **1 load every 1h** (throughput)
  - Time per car = 4h/4 stages = **1 car every 1h** (throughput)
- **But**
  - Time for **per** laundry load is **still 4h** (latency)
  - Time for **per** car is **still 4h** (latency)
- Pipelining **only helps when the pipeline is full**: not when it is filling
  - Speedup for 4 loads of laundry was only 2.3x, not 4x

Why not a zillion stages?

29

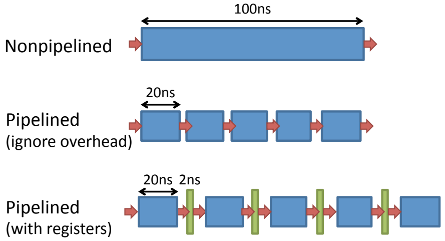
- Ideally we get an N*x* speedup for a pipeline with N stages
- Why not use a zillion stages to get a zillion *x* speedup?
- Two problems:
  - Most things **can't be broken down into infinitely small chunks**
    - Think about the processor we built:
    - How much can we chop up the ALU? or the RF?
    - Practical limit to logic design
  - There is an **overhead for every stage**
    - We need to store the state (which instruction) for each stage
    - This requires a register, and it takes some time



Pipeline registers and overhead

30

- Each pipeline stage is **combinational logic** (think: ALU, sign extension)
- Need to store the **state** for each stage (think: which instruction)
- Need **pipeline registers** between each stage to store the instruction for the stage



Output: 1 per 100ns

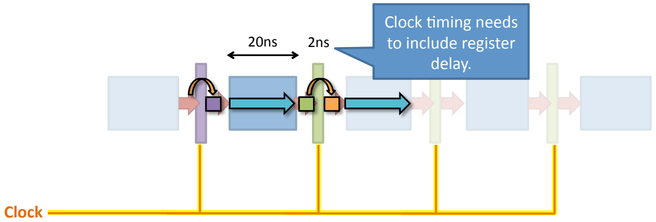
Output: 1 per 20ns (5x faster)

Output: 1 per 22ns (4.5x faster)
10% overhead from pipeline registers

Pipeline clocking

31

- Clock speed determined by **register** → **stage** → **register**
  - **Clock** moves data into **first register**
  - Data goes through the **stage** (combinational: think an adder)
  - Data needs to be at the **next register** in time for the **next clock**



Clock timing needs to include register delay.

Ideal pipelines and reality

32

- **Not all stages are the same length** (not balanced)
  - E.g., RF read may be longer than ALU operation
  - Forces the clock to be the slowest stage, which may not be 1/n
- There **overhead for long pipelines**
  - Hard to chop up the work
  - Pipeline registers take up time
- Hard to keep the pipeline full
  (We'll see more of this in the next lecture)

36

Building a processor pipeline

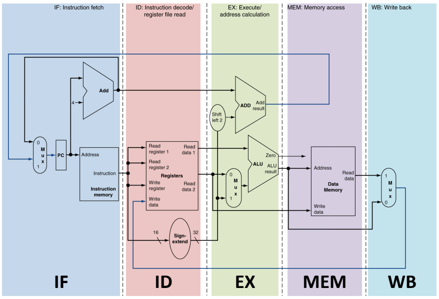
Cutting up the single-cycle processor

How should we divide up the MIPS instructions?

37

(You've already seen it…)

Note: these are not to scale in terms of time.



1. IF: Instruction fetch from memory
2. ID: Instruction decode and register read
3. EX: Execute operation or calculate address
4. MEM: Access memory
5. WB: Write result back to register

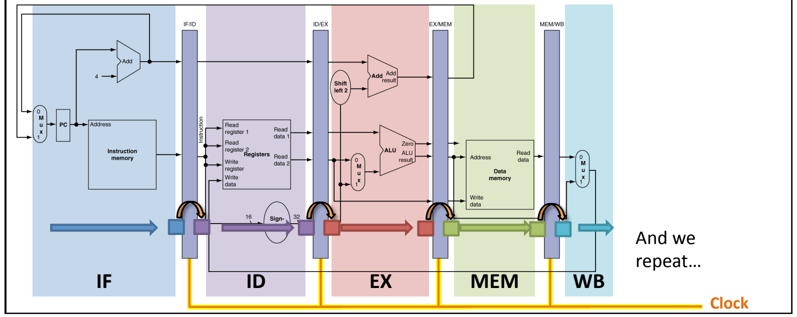
Q: What is missing from this picture?
- ○ Balanced stages
- ○ Pipeline registers
- ○ Write back for the RF

A: Pipeline registers
We need them to store the state (instruction and results) between stages.

Add the pipeline registers

38

- Registers hold the information produced between stages
- Move that data to the next stage on the clock



And we repeat…

Performance benefits of pipelined MIPS

39

- **Single-cycle design:**
  - Clock set for slowest instruction: **800ps clock time**
- **Pipelined design:**
  - Clock set for slowest stage: **200ps**
- Note that some instructions don't use some stages
  - Need control to make sure the stages do the right thing for the right instruction

Instr	IF Instruction Fetch	ID Decode & RF Read	EX Execute	MEM Access Memory	WB Write back to RF	Total time
lw	200ps	100ps	200ps	200ps	100ps	800ps
sw	200ps	100ps	200ps	200ps		700ps
R-format	200ps	100ps	200ps		100ps	600ps
beq	200ps	100ps	200ps			500ps

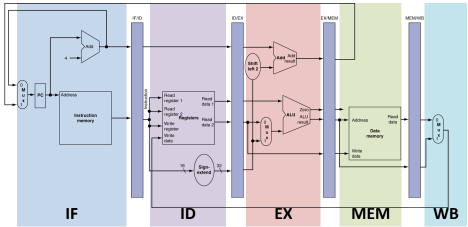
43

A walk through the MIPS pipeline

Pipeline walkthrough: load

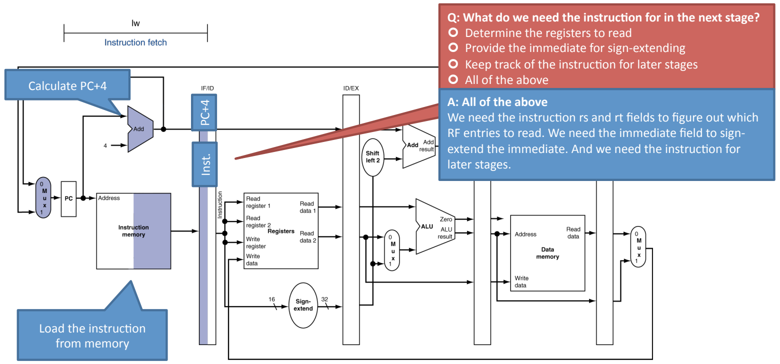
44

- Let's see how a load instruction goes through the pipeline
- Key points:
  - **What happens** in each stage? (combinational)
  - **What is stored** in each pipeline register? (state)



IF for load

45



Calculate PC+4

Load the instruction from memory

Q: What do we need the instruction for in the next stage?

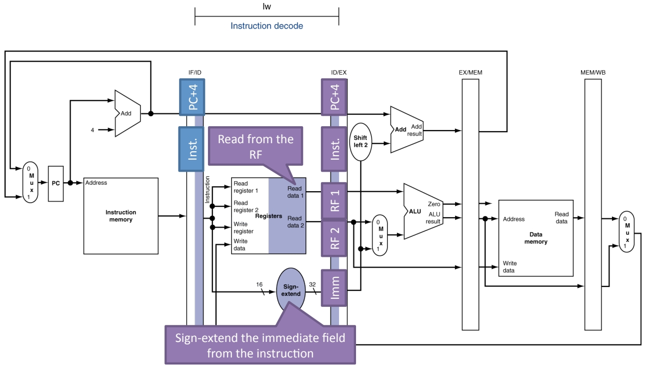
- ○ Determine the registers to read
- ○ Provide the immediate for sign-extending
- ○ Keep track of the instruction for later stages
- ○ All of the above

A: All of the above

We need the instruction rs and rt fields to figure out which RF entries to read. We need the immediate field to sign-extend the immediate. And we need the instruction for later stages.

ID for load

46

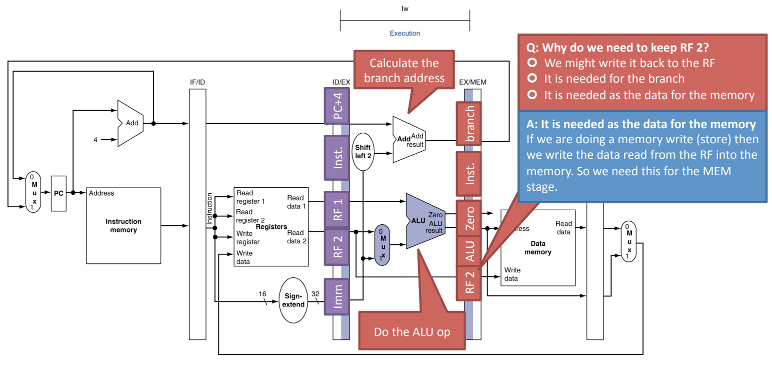


Read from the RF

Sign-extend the immediate field from the instruction

EX for load

47



Calculate the branch address

Do the ALU op

Q: Why do we need to keep RF 2?

○ We might write it back to the RF

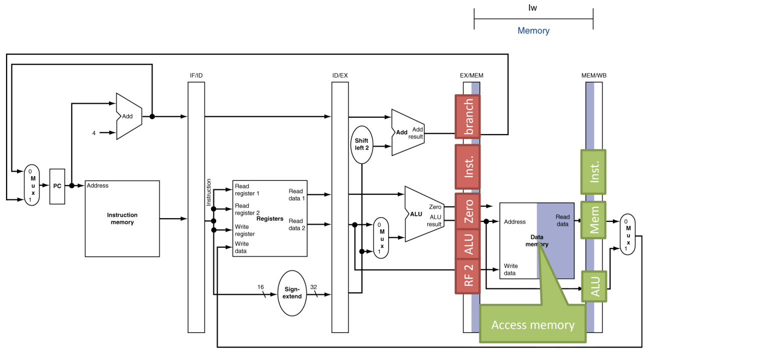
○ It is needed for the branch

○ It is needed as the data for the memory

A: It is needed as the data for the memory
If we are doing a memory write (store) then we write the data read from the RF into the memory. So we need this for the MEM stage.

MEM for load

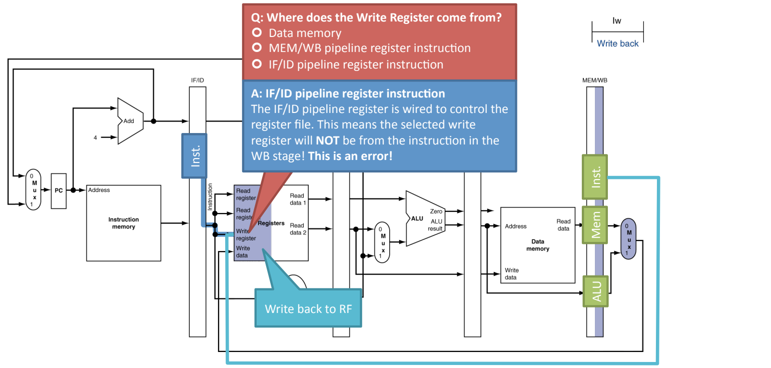
48



Access memory

WB for load

49



Q: Where does the Write Register come from?

○ Data memory

○ MEM/WB pipeline register instruction

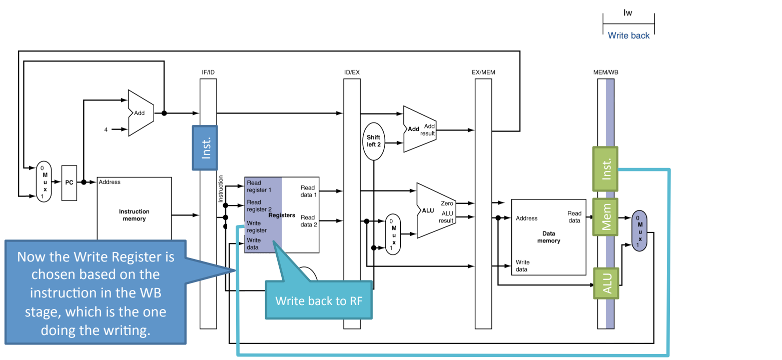
○ IF/ID pipeline register instruction

A: IF/ID pipeline register instruction
The IF/ID pipeline register is wired to control the register file. This means the selected write register will **NOT** be from the instruction in the WB stage! **This is an error!**

Write back to RF

Fixing the WB stage

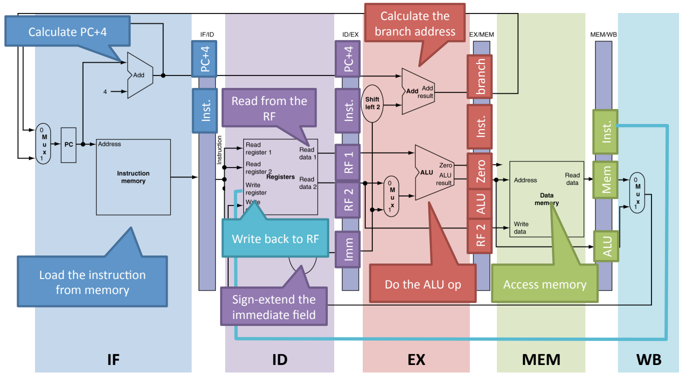
50



Now the Write Register is chosen based on the instruction in the WB stage, which is the one doing the writing.

Write back to RF

The MIPS pipeline

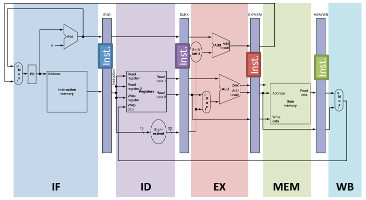


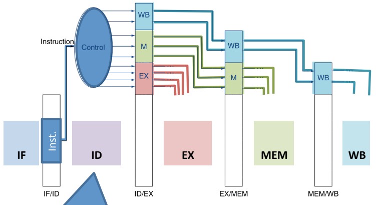
Pipeline control logic

(How do we decode instructions in a pipeline?)

Pipeline control

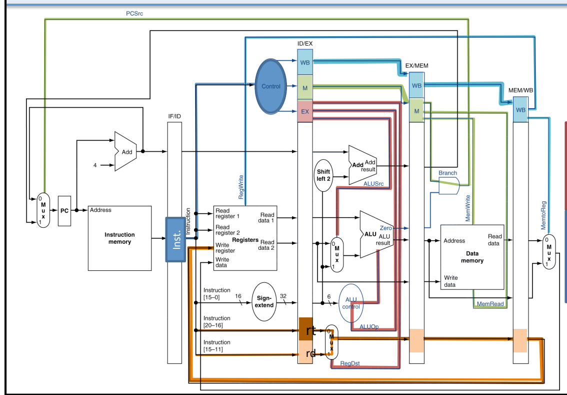
- Do we really need the instruction in every pipeline stage?
- No, we only need some bits for each stage





This is why it is called **Decode**: we decode the instruction into control signals for the pipeline.

Pipeline control in detail



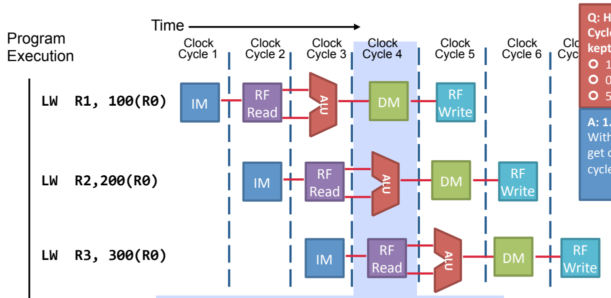
Q: Where does the Write Register come from?

- ○ The MEM/WB control bits (top)
- ○ Instruction in the IF/ID register
- ○ Data in the MEM/WB register

A: Data in the MEM/WB register
Instruction bits 20-16 or 11-15 are sent through to the MEM/WB register and used to determine the register to write to.

Flow of instructions through the pipeline

57



Q: How many Instructions Per Cycle (IPC) would we get if we kept doing load instructions?

○ 1.0
○ 0.2 (one every 5 cycles)
○ 5.0

A: 1.0
With the pipeline full, we will get one instruction out every cycle, or an IPC of 1.0.

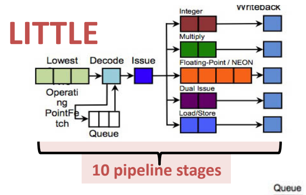
In cycle 4 we have 3 instructions "in-flight":
Inst 1 is accessing the data memory (MEM)
Inst 2 is using the ALU (EX)
Inst 3 is access the register file (ID)

60

Real world pipelines

We saw this earlier

61



Q: Which one is going to run at a faster clock frequency?

○ Little
○ Big
○ Same

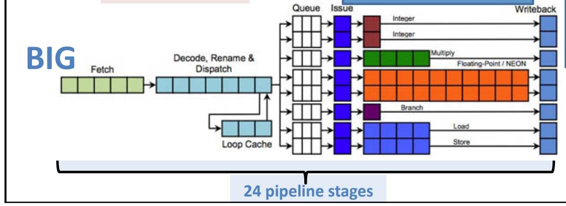
A: Big
The big processor has a longer pipeline, which means each stage will be shorter, so a higher clock frequency.

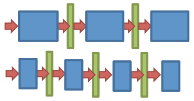
Q: Which pipeline will waste more time on pipeline registers?

○ Little
○ Big
○ Same

A: Big
Running at a higher frequency means that a larger percentage of the time will be spent in pipeline registers.

Equally important, because there are so many more stages, there will be more registers, which use more power and area.

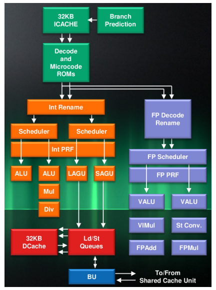


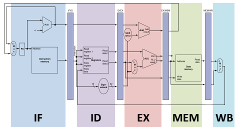


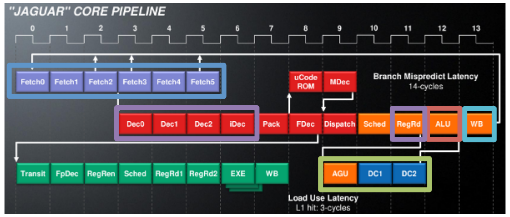
From ARM

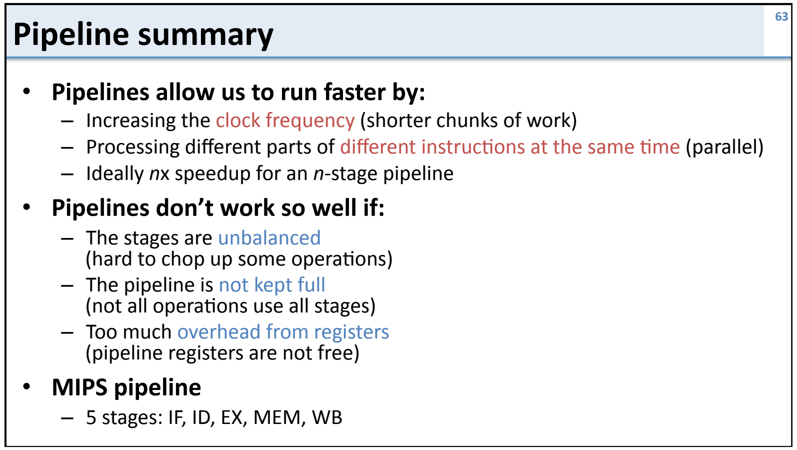
What is AMD doing?

62





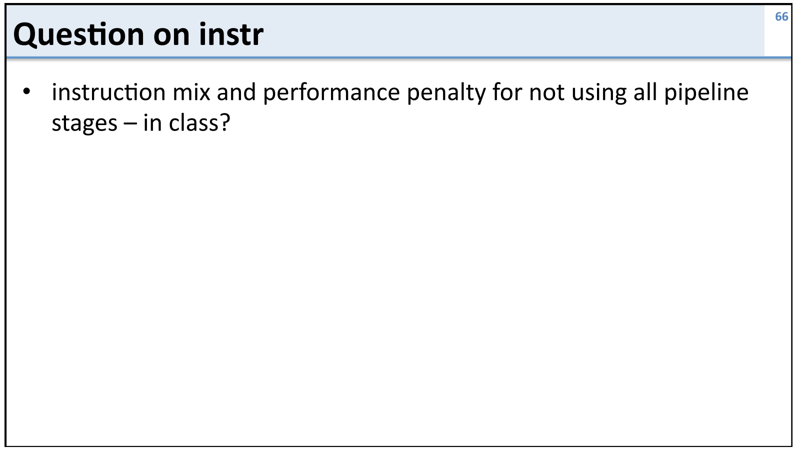




63

Pipeline summary

- **Pipelines allow us to run faster by:**
  - Increasing the clock frequency (shorter chunks of work)
  - Processing different parts of different instructions at the same time (parallel)
  - Ideally *n*x speedup for an *n*-stage pipeline
- **Pipelines don't work so well if:**
  - The stages are unbalanced
    (hard to chop up some operations)
  - The pipeline is not kept full
    (not all operations use all stages)
  - Too much overhead from registers
    (pipeline registers are not free)
- **MIPS pipeline**
  - 5 stages: IF, ID, EX, MEM, WB



66

Question on instr

- instruction mix and performance penalty for not using all pipeline stages – in class?