# Computer Hardware Engineering (IS1200)
# Computer Organization and Components (IS1500)

Spring 2019

Lecture 10: Pipelined Processors

Elias Castegren

Postdoc, KTH Royal Institute of Technology

Slides by David Broman

Slides version 1.1



## Course Structure

**Module 1:** C and Assembly Programming

LE1 → LE2 → LE3 → LE4 → EX1 LAB1 S1 LAB2

**Module 4:** Processor Design

LE9 → LE10 → EX4 S2 LAB4

**Module 2:** I/O Systems

LE5 → LE6 → EX2 LAB3

**Module 5:** Memory Hierarchy

LE11 → EX5 S3

**Module 3:** Logic Design (IS1500 only) PROJ START

LE7 → LE8 → EX3 LD-LAB

**Module 6:** Parallel Processors and Programs

LE12 → LE13 → EX6 S4

Proj. Expo LE14

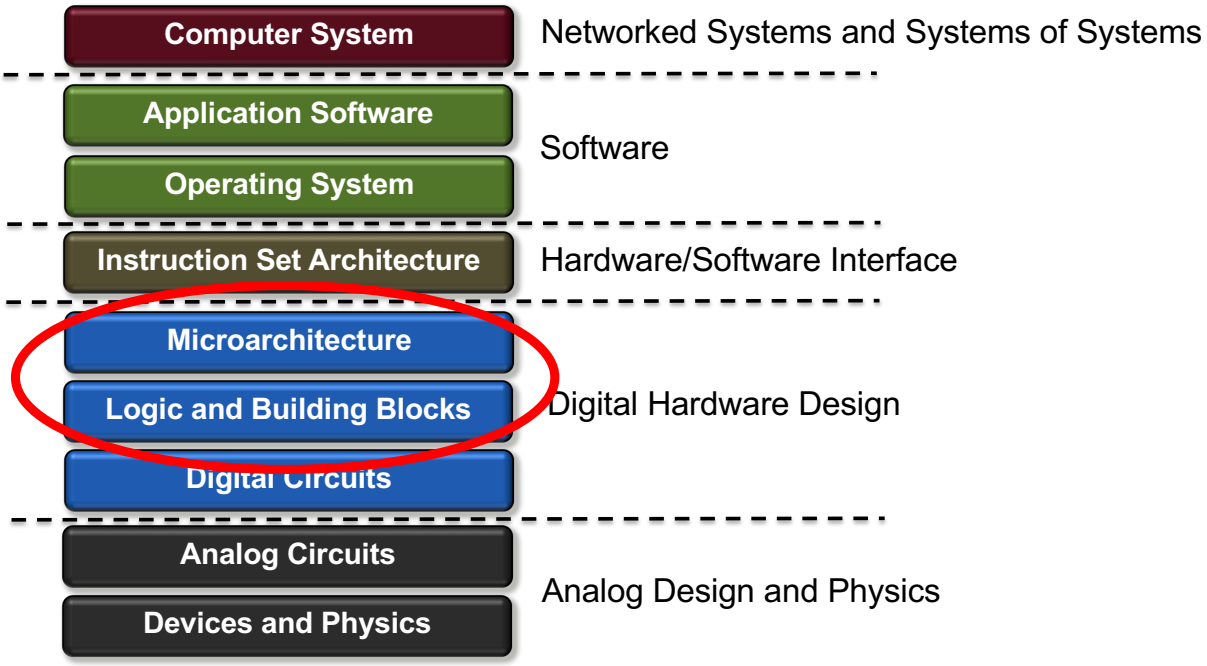
David Broman
dbro@kth.se

**Part I** Pipeline and Datapath

**Part II** Data and Control Hazards

**Part III** ARM and x86

# Abstractions in Computer Systems








# Agenda

**Part I**

**Pipeline and Datapath**

**Part II**

**Data and Control Hazards**


**Part III**

**ARM and x86**

by Raysonho @ Open Grid Scheduler / Grid Engine - Own work. Licensed under Creative Commons Zero, Public Domain

# Part I

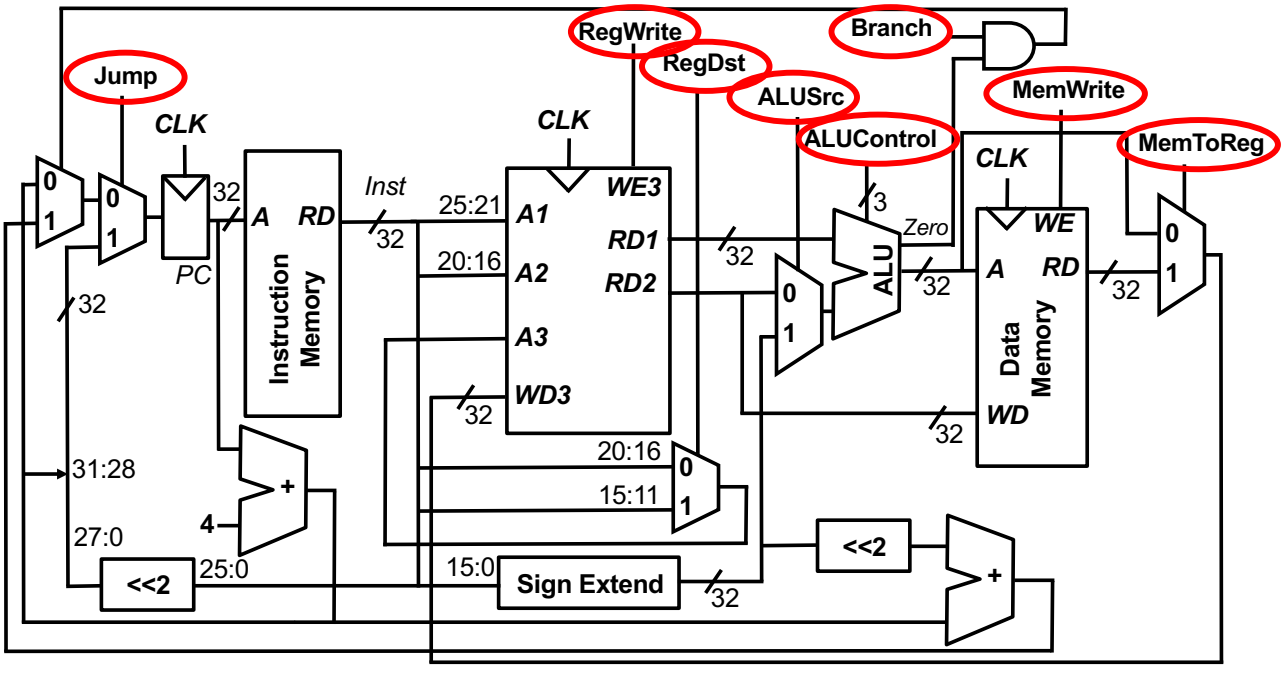
# Pipeline and Datapath

Acknowledgement: The structure and several of the good examples are derived from the book "Digital Design and Computer Architecture" (2013) by D. M. Harris and S. L. Harris.

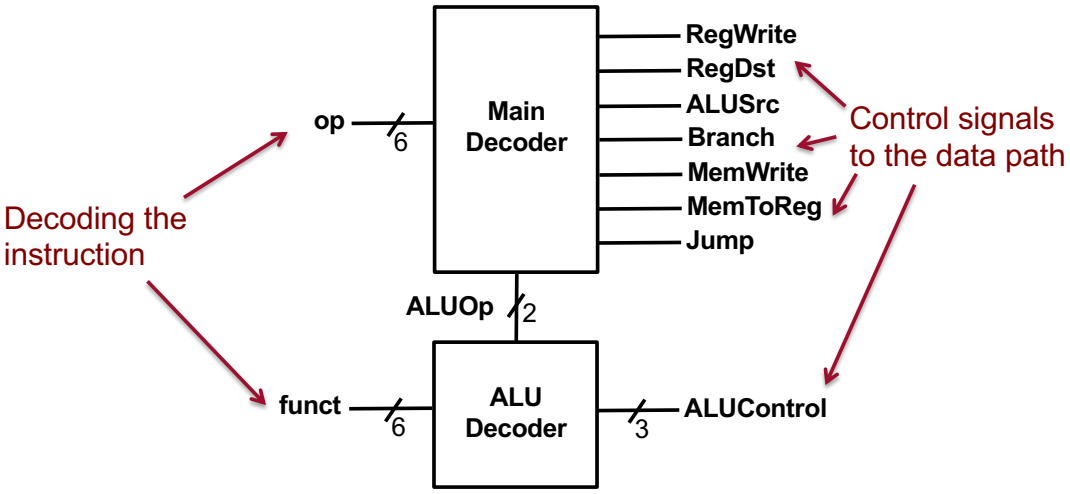




## Data Path (Revisited)



# Control Unit (Revisited)








# Performance Analysis (Revisited)

$$\text{Execution time (in seconds)} = \#\text{ instructions} \times \frac{\text{clock cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{clock cycle}}$$

Number of instructions in a program (# = number of)

Average **cycles per instruction (CPI)**

Seconds per cycle = **clock period $T_C$.**

Determined by programmer or the compiler or both.

Determined by the micro-architecture implementation.

Determined by the critical path in the logic.

For the single-cycle processor, each instruction takes one clock cycle. That is, CPI = 1.

The main problem with the single-cycle processor design (last lecture) is the **long critical path.**

**Solution: Pipelining**

# Parallelism and Pipelining (1/6)
# Definitions

**Processing System**: A system that takes input and produces outputs.

**Token**: An input that is processed by the processing system and results in an output.

**Latency**: The time it takes for the system to process one token.

**Throughput**: The number of tokens that can be processed per time unit.





# Parallelism and Pipelining (2/6)
# Sequential Processing

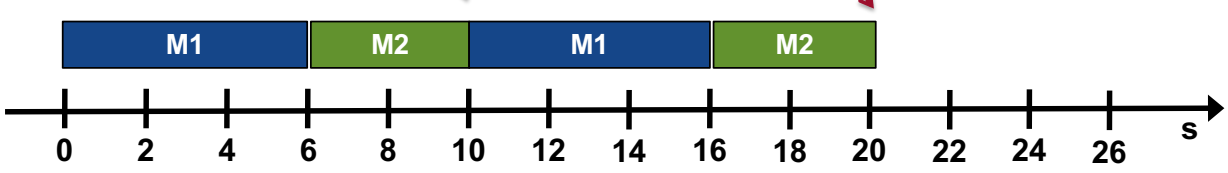
**Example**: Assume we have a Christmas card factory with two machines (M1 and M2).

**M1:** Prints out the card (takes 6s)
**M2:** Puts on a stamp (takes 4s)

**Approach 1.** Process tokens **sequentially**.
In this case a token is a card.

The **latency** is 6 + 4 = 10s

The **throughput** is 1/10 = 0.1 tokens per second or 6 tokens per minute.



# Parallelism and Pipelining (3/6)
# Parallel Processing (Spatial Parallelism)

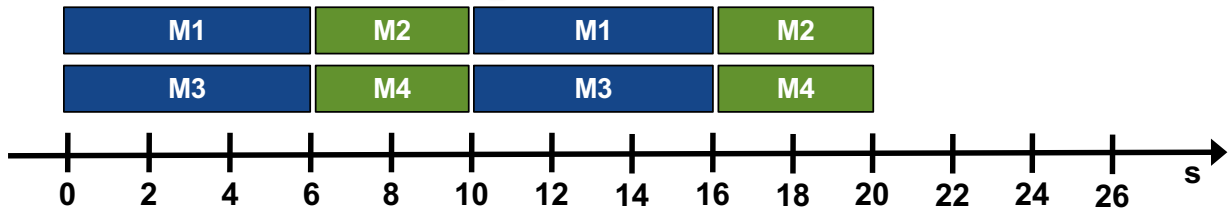
**Example**: Assume we have a Christmas card factory with four machines.

**M1:** Prints out the card (takes 6s)
**M2:** Puts on a stamp (takes 4s)
**M3:** Prints out the card (takes 6s)
**M4:** Puts on a stamp (takes 4s)

**Approach 2.** Process tokens **in parallel** using more machines.

The **latency** is 6 + 4 = 10s

The **throughput** is 2 * 1/10 = 0.2 tokens per second or 12 tokens per minute.








# Parallelism and Pipelining (4/6)
# Pipelining (Temporal Parallelism)

**Example**: Assume we have a Christmas card factory with two machines.

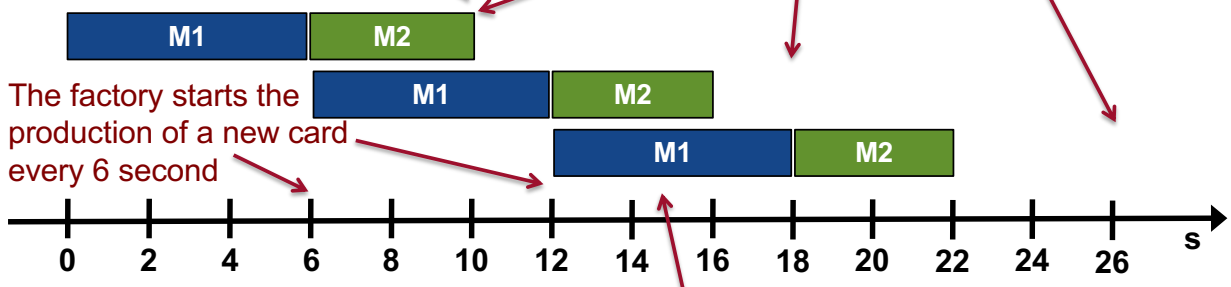
**M1:** Prints out the card (takes 6s)
**M2:** Puts on a stamp (takes 4s)

**Approach 3.** Process tokens by **pipelining** using only two machines.

The **latency** is still 6 + 4 = 10s

The **throughput** is 1/6 (on average) = 0.1666… tokens per second or 10 tokens per minute.

The factory starts the production of a new card every 6 second




Note that the throughput is limited by the slowest machine.

# Parallelism and Pipelining (5/6) Summary

**Approach 1.** Process tokens sequentially using *two* machines

Latency: 10s
Throughput: 6 tokens/min

We improve throughput, but not latency

**Approach 2.** Process tokens in parallel using *four* machines

Latency: 10s
Throughput: 12 tokens/min

Parallelism in approach 2 adds extra machines, but pipelining in approach 3 does not require extra hardware.

**Approach 3.** Process tokens by **pipelining** using only *two* machines.

Latency: 10s
Throughput: 10 tokens/min

Again: throughput improvements are limited by the slowest machine (in this case M1)






# Parallelism and Pipelining (6/6) Performance Analysis for Pipelining

**Idea:** We introduce a pipeline in the processor

How does this affect the execution time?

$$\text{Execution time (in seconds)} = \#\text{ instructions} \times \frac{\text{clock cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{clock cycle}}$$

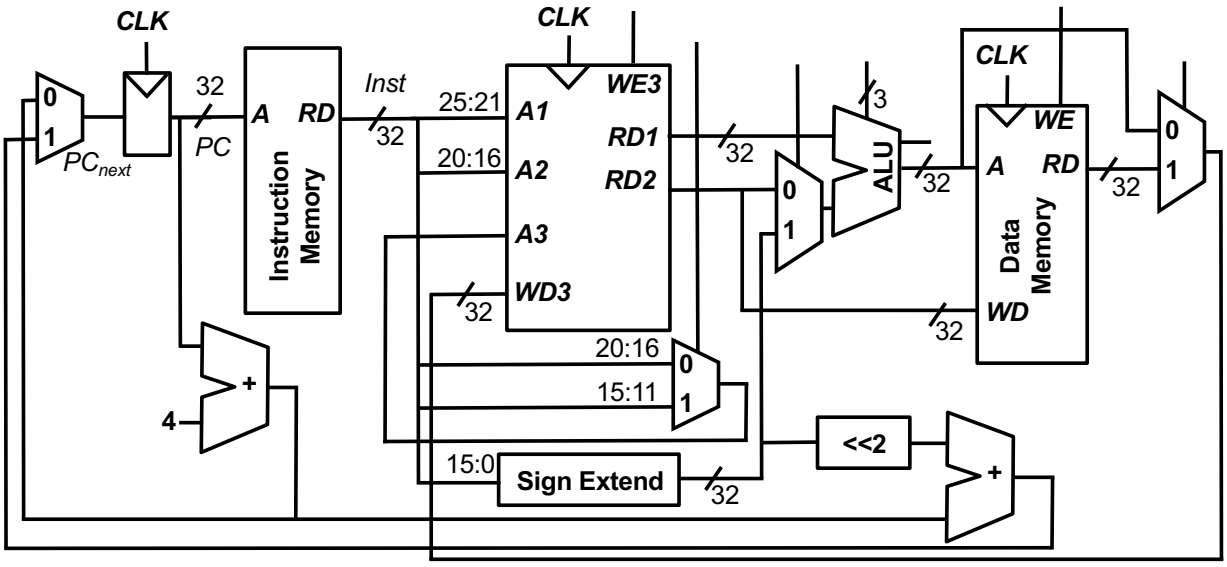
Pipelining does not change the number of instructions

Pipelining will not improve the CPI (actually, it will be slightly worse)

Pipelining will improve the cycle period (make the critical path shorter)

# Towards a Pipelined Datapath (1/8)

Recall the single-cycle data path (the logic for the `j` and `beq` instructions is hidden)

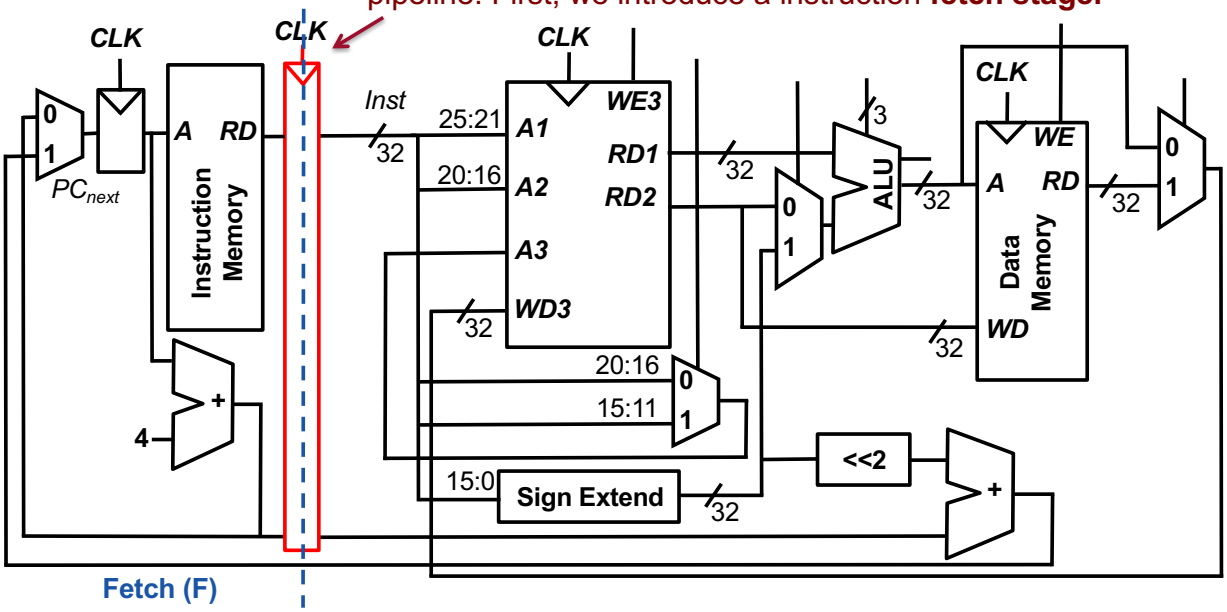






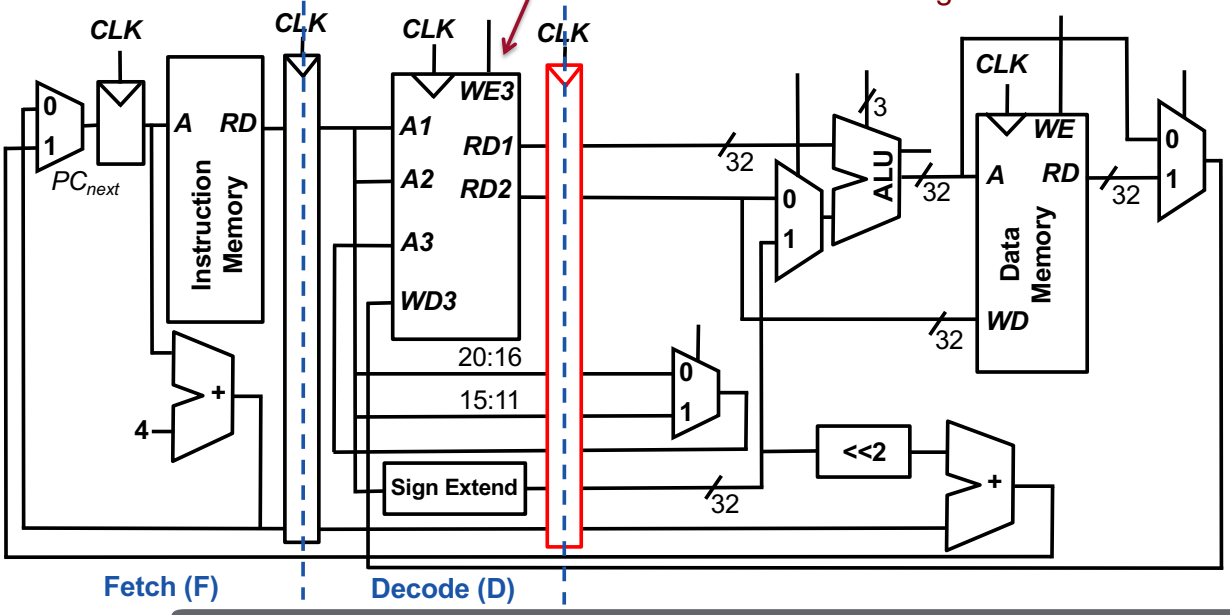

# Towards a Pipelined Datapath (2/8) Fetch Stage

A register splits the datapath into stages, forming a pipeline. First, we introduce a instruction **fetch stage.**



# Towards a Pipelined Datapath (3/8)
## Decode Stage

A **decode stage** decodes an instruction and reads out values from the register file.

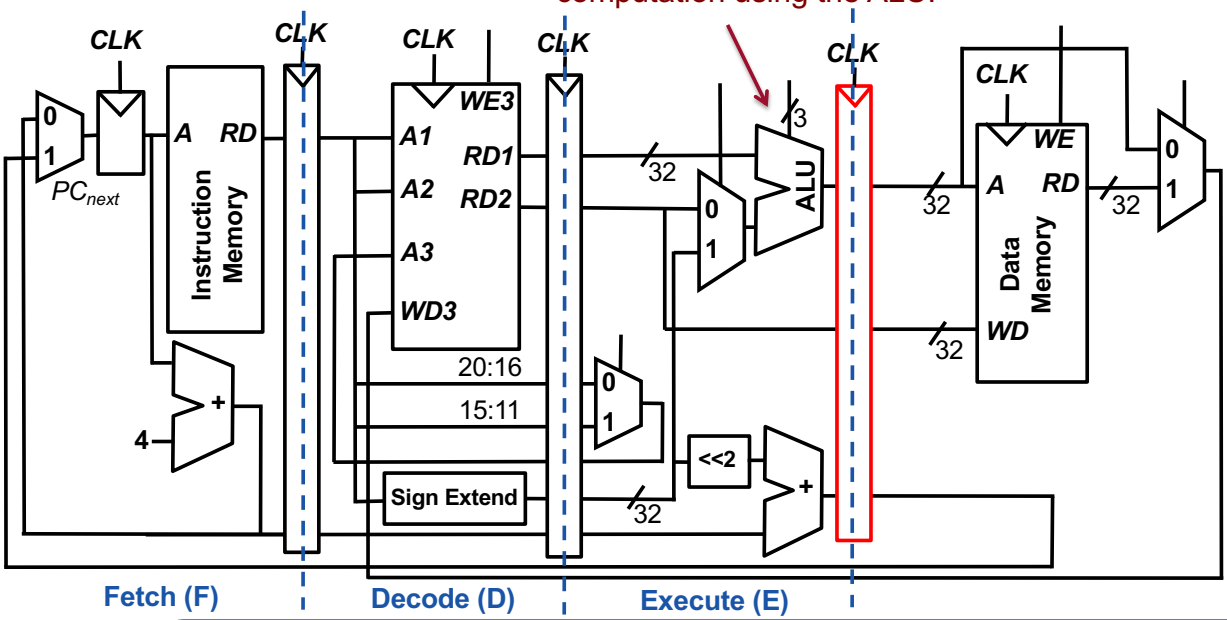






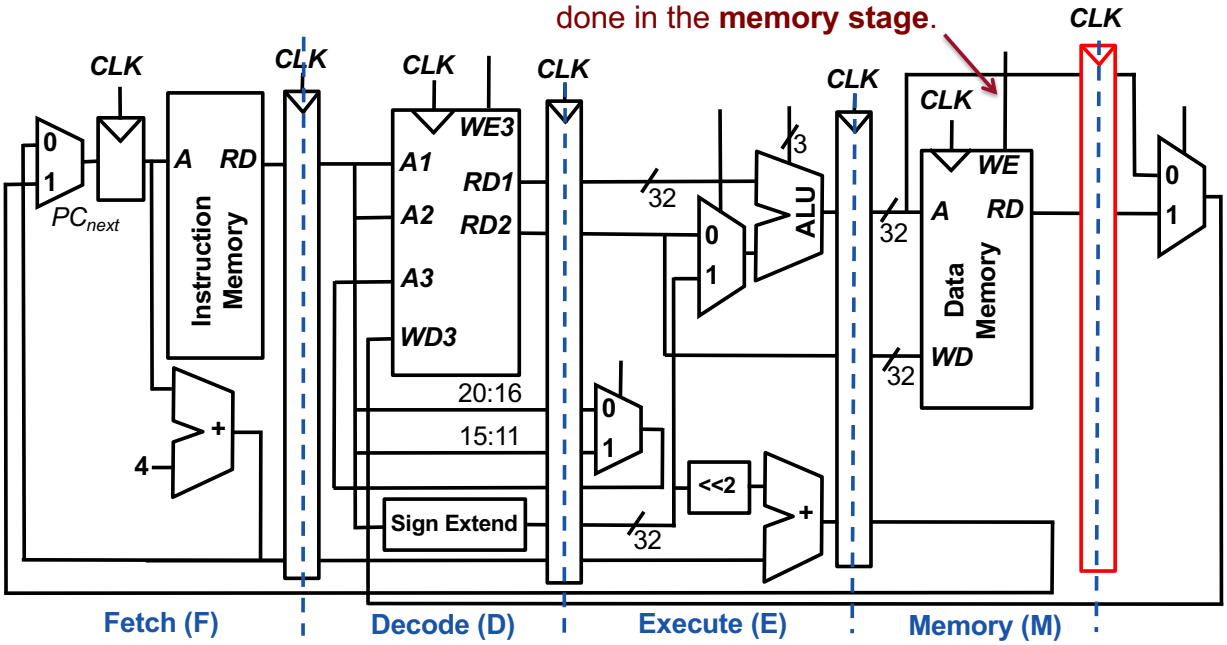

# Towards a Pipelined Datapath (4/8)
## Execute Stage

An **execute stage** performs the computation using the ALU.



# Towards a Pipelined Datapath (5/8)
## Memory Stage

Reading and writing to memory is done in the **memory stage**.



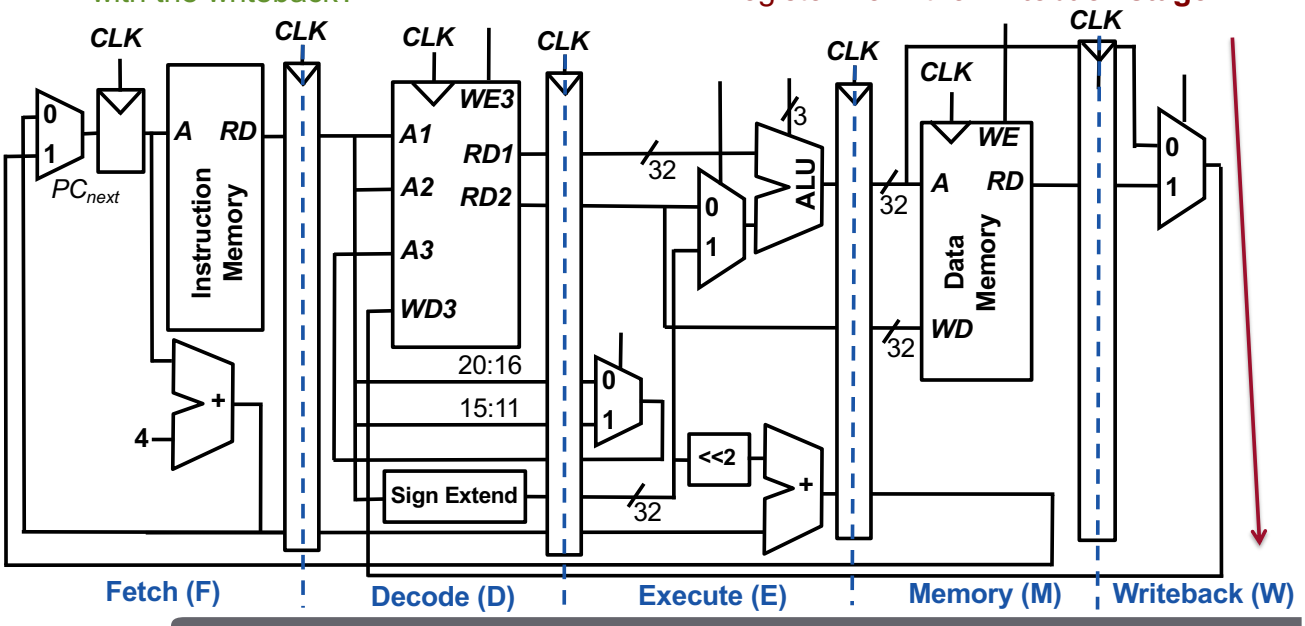





# Towards a Pipelined Datapath (6/8)
## Writeback Stage

Can you see a problem with the writeback?

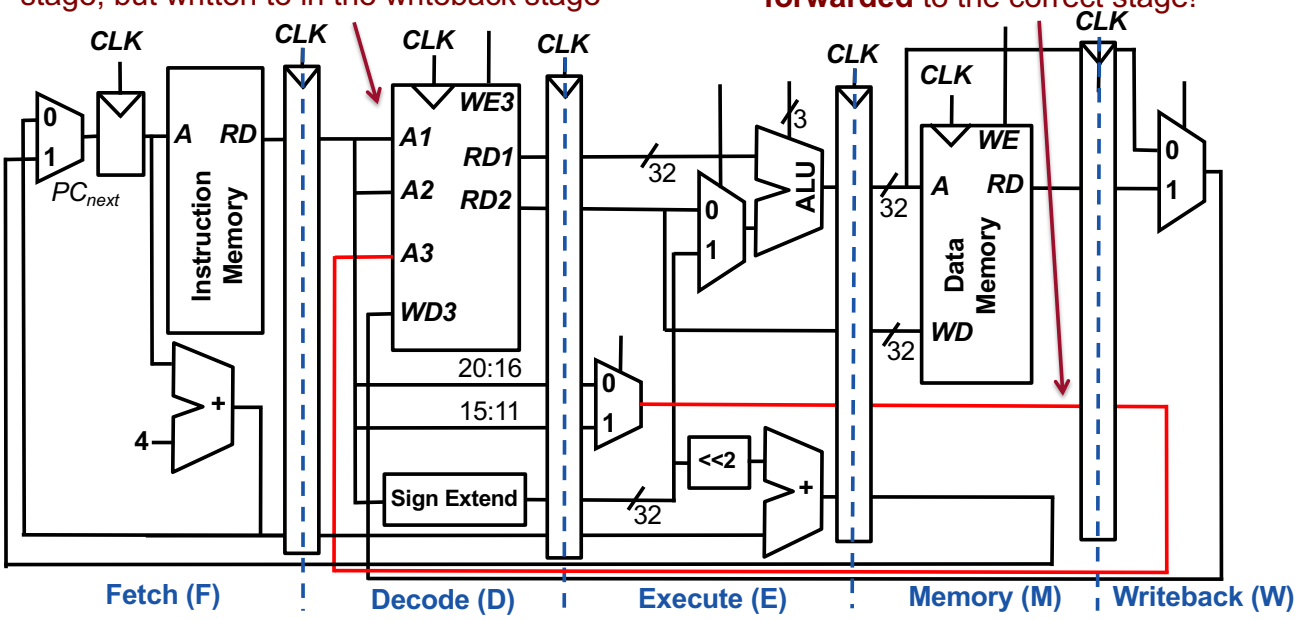
The results are written back to the register file in the **writeback stage.**



# Towards a Pipelined Datapath (7/8) Writeback Stage

Note that the register file is read in the decode stage, but written to in the writeback stage

The address must be **forwarded** to the correct stage!



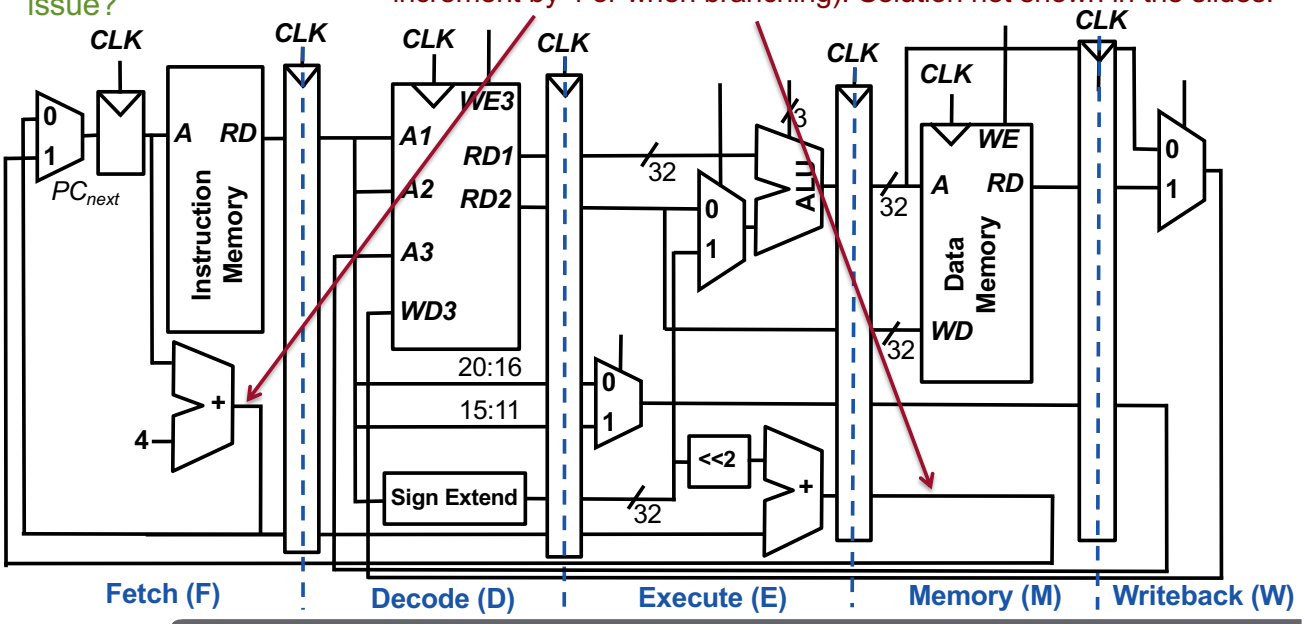





# Towards a Pipelined Datapath (8/8) Another issue

Can you see another issue?

The program counter can be updated in the wrong stage (PC increment by 4 or when branching). Solution not shown in the slides.



# Part II

# Data and Control Hazards


Acknowledgement: The structure and several of the good examples are derived from the book "Digital Design and Computer Architecture" (2013) by D. M. Harris and S. L. Harris.



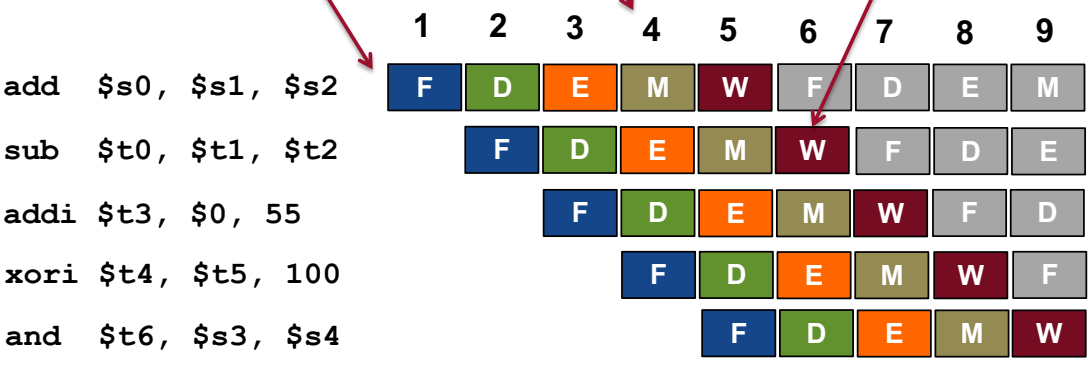


## A Five-Stage Pipeline

In each cycle, a new instruction is fetched, but it takes 5 cycles to complete the instruction.

In each cycle all stages are handling different instructions in parallel.

**Example**. In cycle 6, the result of the `sub` instruction is written back to register `$t0`.




We can fill the pipeline because there are no dependencies between instructions

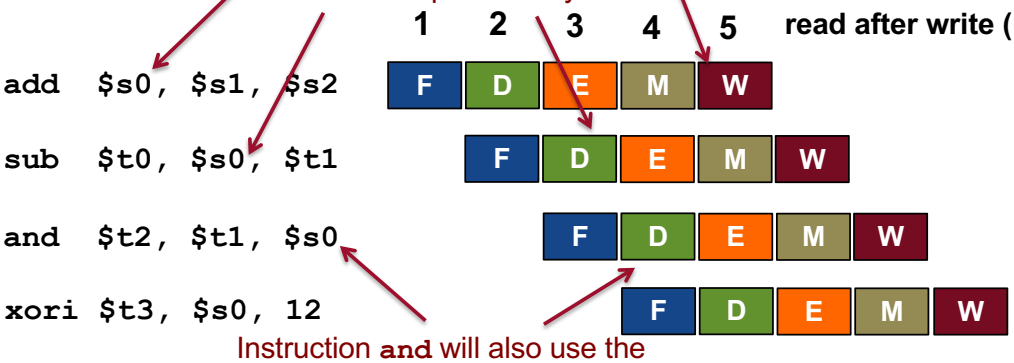
**Exercise**: What is the ALU doing in cycle 5?

**Answer**: Adding together values 0 and 55

# Data Hazards (1/4)
# Read after Write (RAW)

The `add` instruction writes back the value $s0 in cycle 5

But $s0 is used in the decode phase in cycle 3.




Instruction `and` will also use the wrong value for $s0.

A **data hazard** occurs when an instruction reads a register that has not yet been written to.

This kind of data hazard is called **read after write (RAW)** hazard.

**Exercise**: For MIPS, will instruction `xori` result in a hazard? Stand for yes, sleep for no.

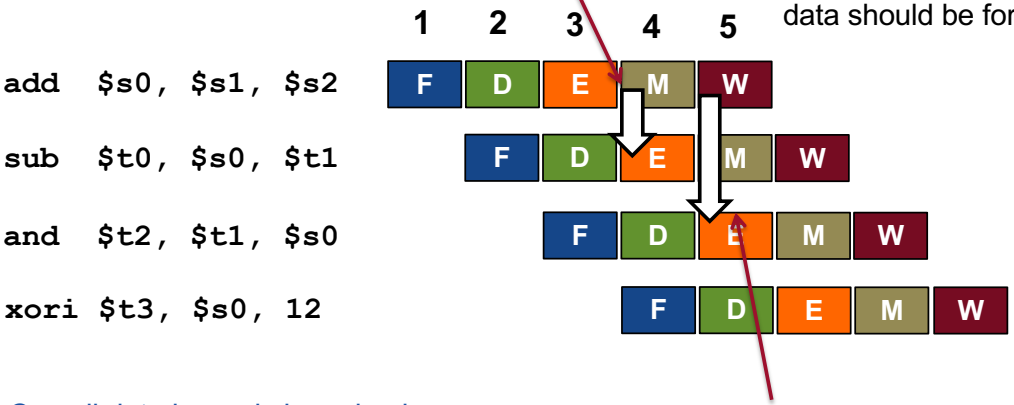
**Answer:** No. `xori` is OK for MIPS, because it writes on the first part of the cycle and reads on the second part.





# Data Hazards (2/4)
# Solution 1: Forwarding

The result from the execute stage for `add` can be **forwarded** (also called bypassing) to the execute stage for `sub`.




Hazard detection is implemented using a **hazard detection unit** that gives control signals to the datapath if data should be forwarded.

Can all data hazards be solved using forwarding?

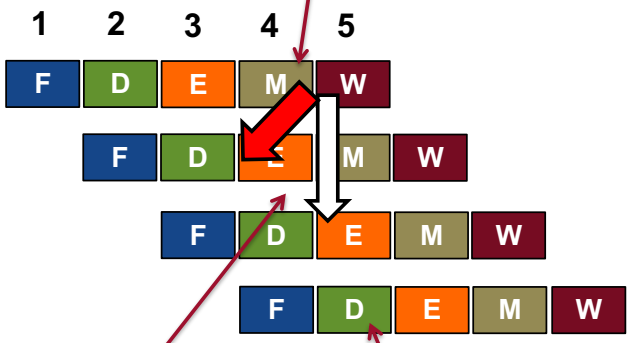
The `and` instruction's hazard is solved by forwarding as well.

# Data Hazards (3/4)
# Solution 1: Forwarding (partially)

**Exercise**: Which of the instructions `sub`, `and`, and `xori` have data hazards? Which can be solved using forwarding?

**Answer:**
Hazards: `sub` and `and`
Can use forwarding: `and`

```
lw  $s0, 20($s2)
sub  $t0, $s0, $t1
and  $t2, $t1, $s0
xori $t3, $s0, 12
```




The `sub` instruction cannot be solved using forwarding because the memory access is available at the end of cycle 4, but is needed in the beginning of cycle 4.

The `and` instruction memory result can be forwarded after the memory stage to execution.

`xori` can read the data from the write stage (writes in first part of cycle, reads in second part)



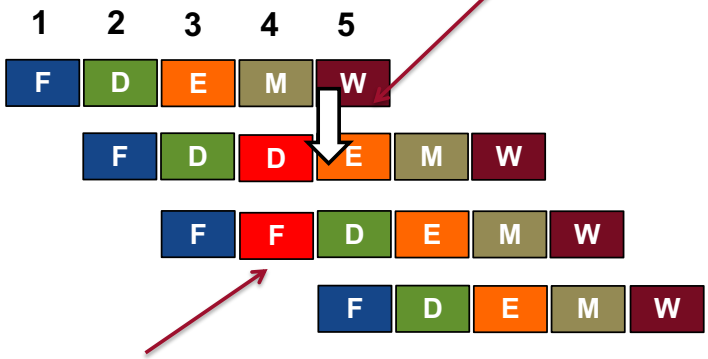


# Data Hazards (4/4)
# Solution 2: Stalling

Solution when forwarding does not work: **stalling**

After stalling, the result can be forwarded to the execute stage.

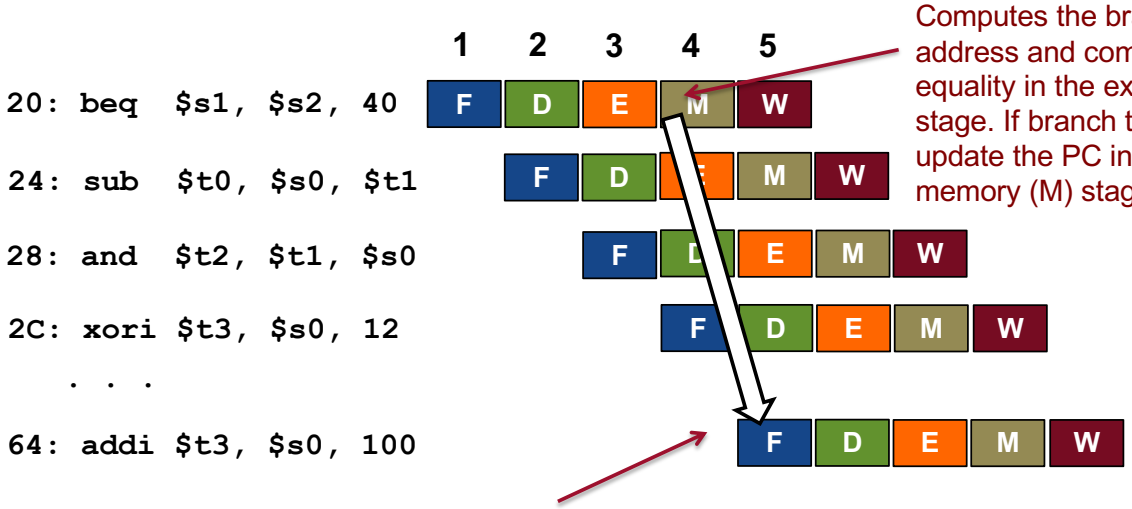
```
lw  $s0, 20($s2)
sub  $t0, $s0, $t1
and  $t2, $t1, $s0
xori $t3, $s0, 12
```




We need to stall the pipeline. Stages are repeated and the fetch of `xori` is delayed.

Stalling results in more than one cycle per instruction. The unused stage is called a **bubble**.

# Control Hazards (1/5)
# Assume Branch Not Taken




Computes the branch target address and compares for equality in the execute (E) stage. If branch taken, update the PC in the memory (M) stage.

If the branch is taken, we need to flush the pipeline. We have a **branch misprediction penalty** of 3 cycles.

**Can we improve this?**



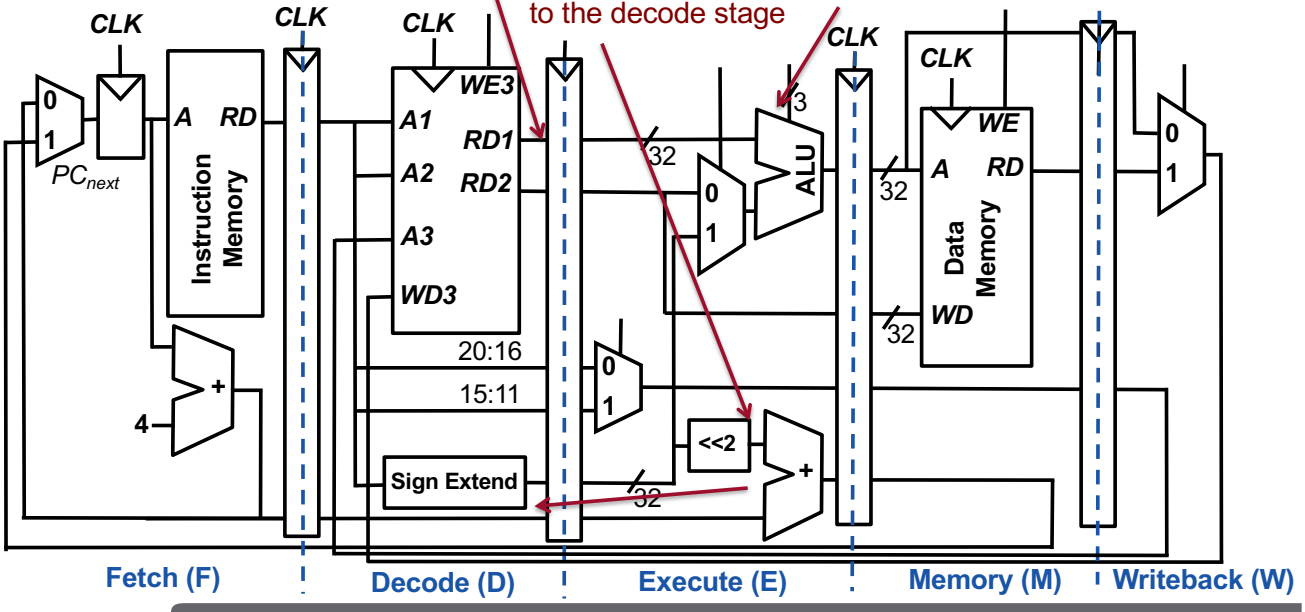


# Control Hazards (2/5)
# Improving the Pipeline

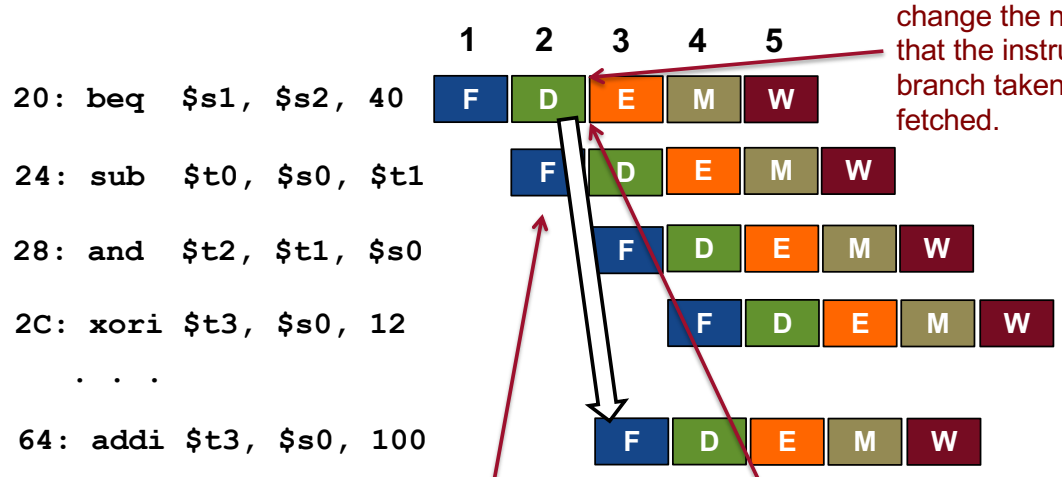
Add an equality comparison for `beq` in the decode phase (not shown here)

Move the branch address calculation to the decode stage

Right now, branch comparison is done in the execute stage



# Control Hazards (3/5)
# Assume Branch Not Taken




The decode phase can change the next PC, so that the instruction at the branch taken address is fetched.

Branch misprediction penalty is now reduced to 1 cycle.

Note that we may now introduce another data hazard (if operands are not available in the decode stage).

Can be solved with forwarding or stalling





# Control Hazards (4/5)
# Deeper Pipelines

Why do we sometimes want more stages than 5?

The critical path can be shorter with less logic in the slowest stage.

The processor can have higher clock frequency.

For instance, Intel's Core 2 duo has more than 10 pipeline stages.

Why not always have more pipeline stages?

Adds hardware (registers)

The branch mispredication penalty increases!

# Control Hazards (4/5)
# Deeper Pipelines

How can we handle deep pipelines, and minimize misprediction?

## Static Branch Predictors

- Statically (at compile time) determine if a branch is taken or not. For instance, predict branch not taken.

## Dynamic Branch Predictors

- Dynamically (at runtime) predict if a branch will be taken or note.
- Operates in the fetch state.
- Maintains a table, called the **branch target buffer**, that contains hundreds or thousands of executed branch instructions, their destinations, and information if the branches were taken or not.





# Part III

# ARM and x86

# Armv7

The most popular ISA for **embedded devices**
(9 billon devices in 2011, growth 2 billion a year).


More complex addressing modes than MIPS (can do shift and add of addresses in registers in one instruction)

Condition results are saved in special flags: negative, zero, carry, overflow.

16 registers, each 32-bit (integers)

Instruction size 32-bit (Thumb-mode, 16-bits encoding).

Conditional execution of instructions, depending on condition code.

Example1: ARM Cortex-A8, a processor at 1GHz, 14-stage pipeline, with branch predictor.
Example 2: A6 by Apple, manufactured by Samsung, used in IPhone 5.





# x86

Standard in laptops, PCs, and in the cloud

CISC –instructions are more powerful than for ARM and MIPS, but requires more complex hardware

x86 architecture has evolved over the last 35 years,

There are 16, 32, and 64 bits variants.

8 general purpose registers (eax, ebx, ecx, edx, esp, ebp, esi, edi).


Variable length of instruction encoding (between 1 and 15 bytes)

Arithmetic operations allow destination operand to be in memory.

Major manufacturers are Intel and AMD.

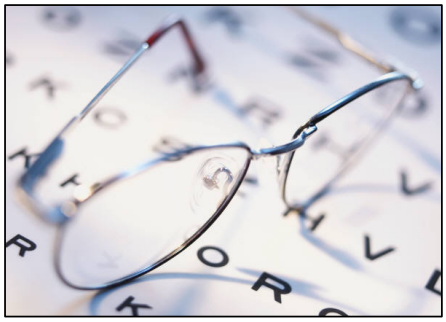
# You know the drill…

**…keep calm**





# Reading Guidelines



**Module 4: Processor Design**

Lecture 9: ALU and Single-Cycled Processors
- H&H Chapters 5.2.4, 7.1-7.3.

Lecture 10: Pipelined processors
- H&H Chapters 7.5, 7.8.1-7.8.2, 7.9

**Reading Guidelines**
See the course webpage for more information.

# Summary

**Some key take away points:**


- Pipelining is a **temporal** way of achieving parallelism
- **Pipelining processors** improve performance by reducing the clock period (shorter critical path)
- Pipelining introduces pipeline hazards. There are two main kind of hazards: **data hazards** and **control hazards**.
- Data hazards are solved by forwarding or stalling
- Control hazards are solved by flushing the pipeline and improved by **branch prediction**.

**Thanks for listening!**