

# The Processor

⇒ [Processor] architecture

The processor has two personalities depending on where you look!

## ⇒ The “ active part ”

🔑 🖊 Follow instructions

🖊 Perform requested operations

👁 In earlier machines, the instruction set, addressing modes, register and memory organization closely matched the hardware.

**Instruction set architecture (ISA)**, often preserved across generations of hardware, remains useful as a specification for writing machine programs (directly or via compiler agents).

## ⇒ Programmer interface

What functions are available for use

## ⇒ Implementation details

How parts/functions are setup/organized

The internal makeup.

# Processor Implementation The Datapath

⇒ Load-store

⇒ Microarchitecture

👁 Collection of datapaths, storage, and functional units hidden from programmers.

In a simplified scenario, the processor deals with instructions and data from different threads of many programs in memory encoded in a logical bit-stream from processing viewpoint.

At least one datapath for core instructions (typically default 2's comp operands, int in C); there may be more for specialized operands and ops.

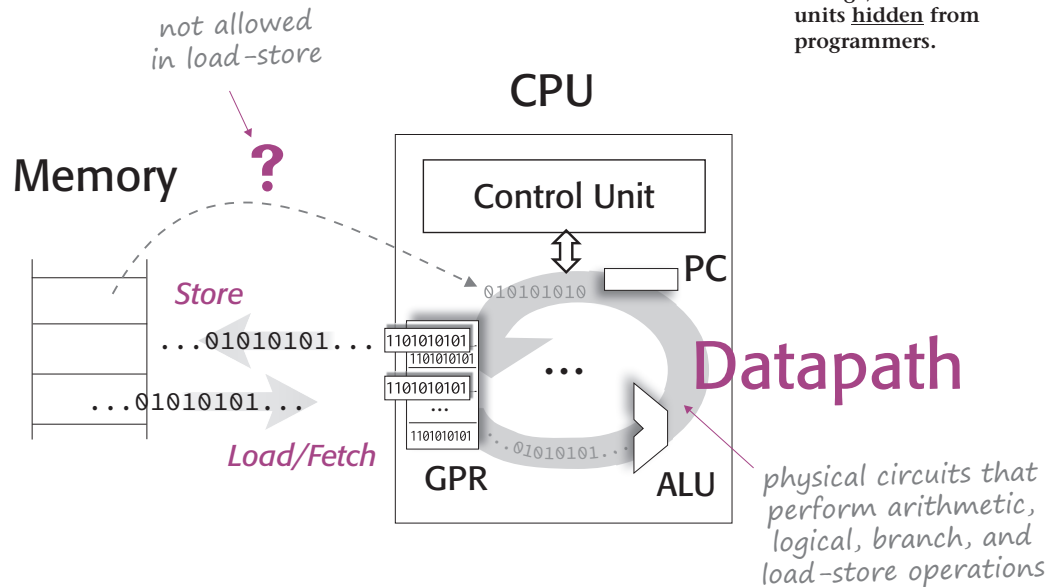
Goal, generally, to minimize instruction time (**latency**) and maximize processing rates (**throughput**).



Opcodes indicate operations to perform, how to obtain operands, and which variants to use, i.e., the **datapaths**.

## Quiz

Give examples from MIPS.



# Instruction Implementation Datapath Resources

- ⇒ Architectural regs
- ⇒ Register file

## Example: Load Word

**Recall Specification (ISA)**

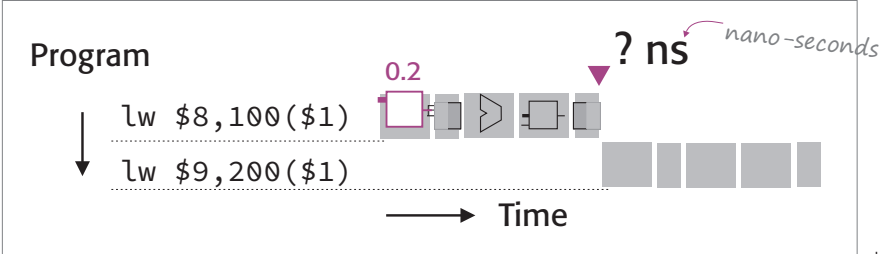
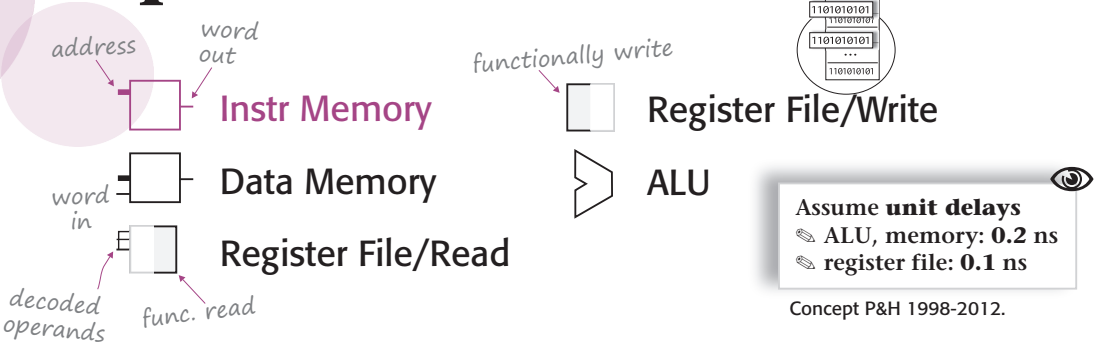
- Fetch instr word from instr memory
- Load base addr from \$1
- Sum base addr and constant passed in instr
- Fetch data word using sum as addr
- Store data word in \$8/\$9

**Required Functions/Resources**

- Read an Instr Memory
- Read access a Register File
- Perform op in ALU
- Read a Data Memory
- Write access Register File

Load word has to go through specified functions in sequence regardless of how a datapath is designed.

**Exercise**  
Specify a different behavior, determine the sequence of functions and suggest functional units to implement.



# Instruction Overlapping

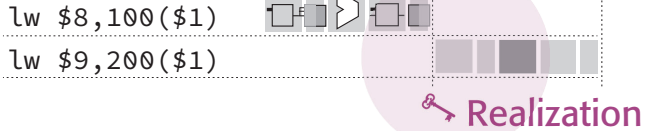


May identify an **execution stage** characterized by use of a main **datapath resource**; for example, an ALU stage.

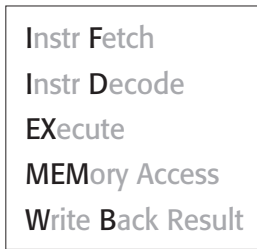


In each stage, most of the datapath is unused.

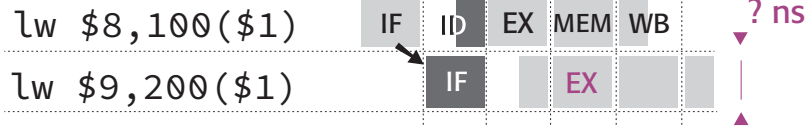
## Sequential execution



## Overlapped execution



Classic MIPS Stages



In MIPS, Load-word is the **most expensive**, requiring all major resources sequentially.

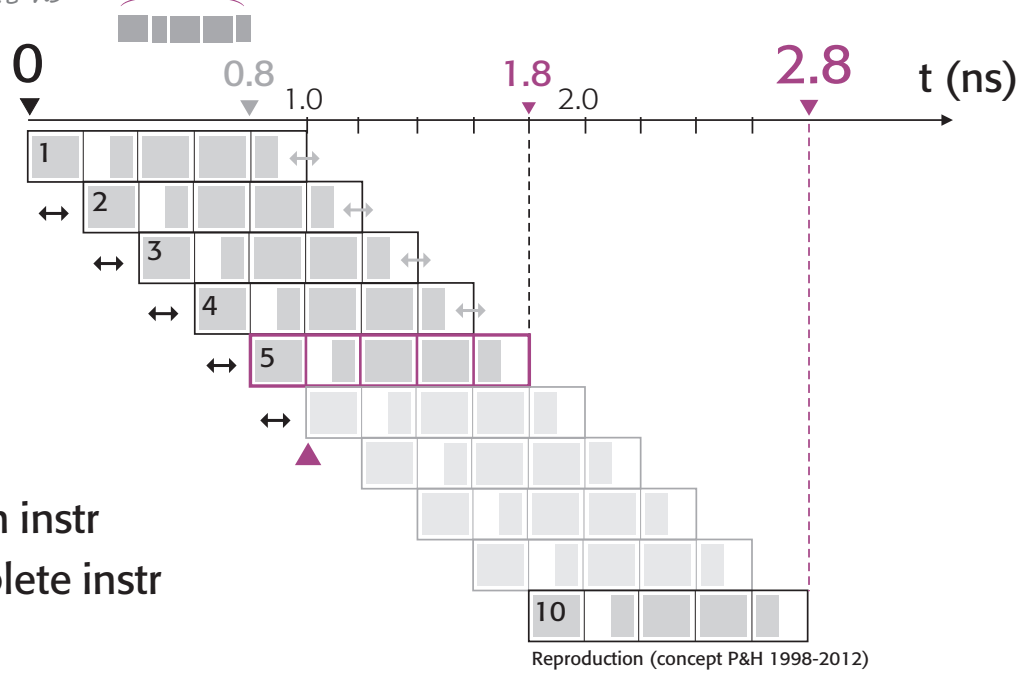
## How much better?

1.6 ns sequential,  
1.2 ns overlapped,  
a 33% improve-  
ment.

SIMPLIFIED

# Processor Implementation Pipelining Load-Word

on a sequential  
datapath, may start  
one every 0.8 ns



## Compare

- Time between instr
- Time to complete instr
- Speedup = ?

# Pipelining Load-Word Performance

- ⇒ Instr throughput
- ⇒ Instr latency
- ⇒ Pipeline paradox
- ⇒ Ideal speedup

Time between instr  
(time to start or *issue* nth instr)

$n$	non-pipelined	pipelined	Speedup
3	$.8 \times 3$	$.2 \times 3$	$\frac{.8 \times 3}{.2 \times 3} = 4$
10	?	?	4

Reproduction (concept P&H 1998-2012)

*long run, should complete  
instrs as fast as can issue  
them!*

## Quiz

What is the formula  
for the pipelined case?  
*Answer later slide.*



Seems attractive to add  
stages to raise speedup  
ceiling/headroom. (Keep  
an eye on this idea).

Time to complete  
(total exec time)

$n$	non-pipelined	pipelined	Speedup
3	$.8 \times 3$	1.4	$\frac{2.4}{1.4} \approx 1.71$
10	$.8 \times 10$	2.8	$\frac{8.0}{2.8} \approx 2.86$
1000	$.8 \times 1000$	$.8 + 1000 \times .2$	$\frac{800}{200.8} \approx 3.98$
100,000	?	?	$\approx 4.0$

Reproduction (concept P&H 1998-2012)

# Review: Logic Design Clock Signal



A computer is a **finite state machine** (FSM), a math model (tool) to capture automatic logic.

An FSM transitions between different **states** in response to inputs (assume **Moore** style).

A physical FSM based on digital **sequential logic** directs reads and writes of stored bits (encoding **state**) in response to inputs, resulting in controlled moves between states.

[https://www.tutorialspoint.com/automata\\_theory/moore\\_and\\_mealy\\_machines.htm](https://www.tutorialspoint.com/automata_theory/moore_and_mealy_machines.htm)

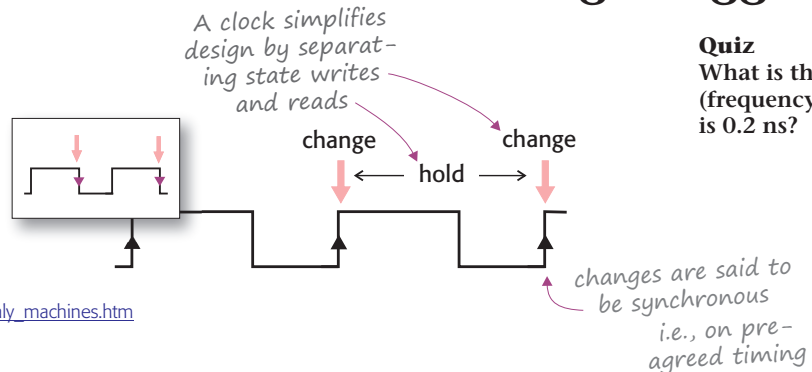
Recall, physical devices in a computer naturally encode and transform a binary state (bits).

⇒ **Sequential logic**

⇒ **Edge triggered**

**Quiz**

What is the clock speed (frequency) if the cycle is 0.2 ns?



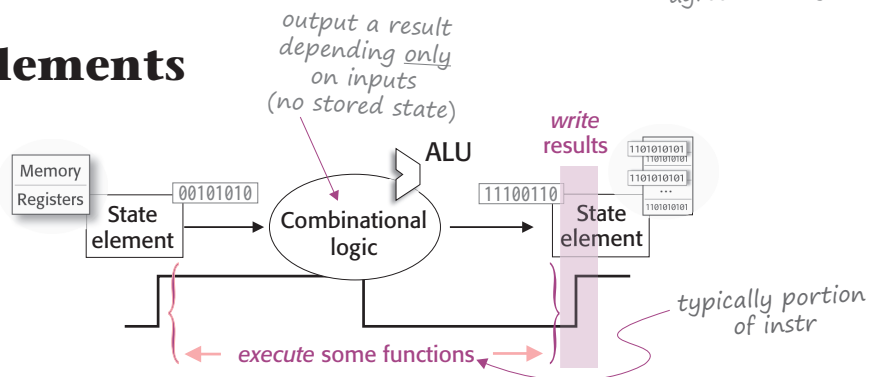
stored bits

## State elements

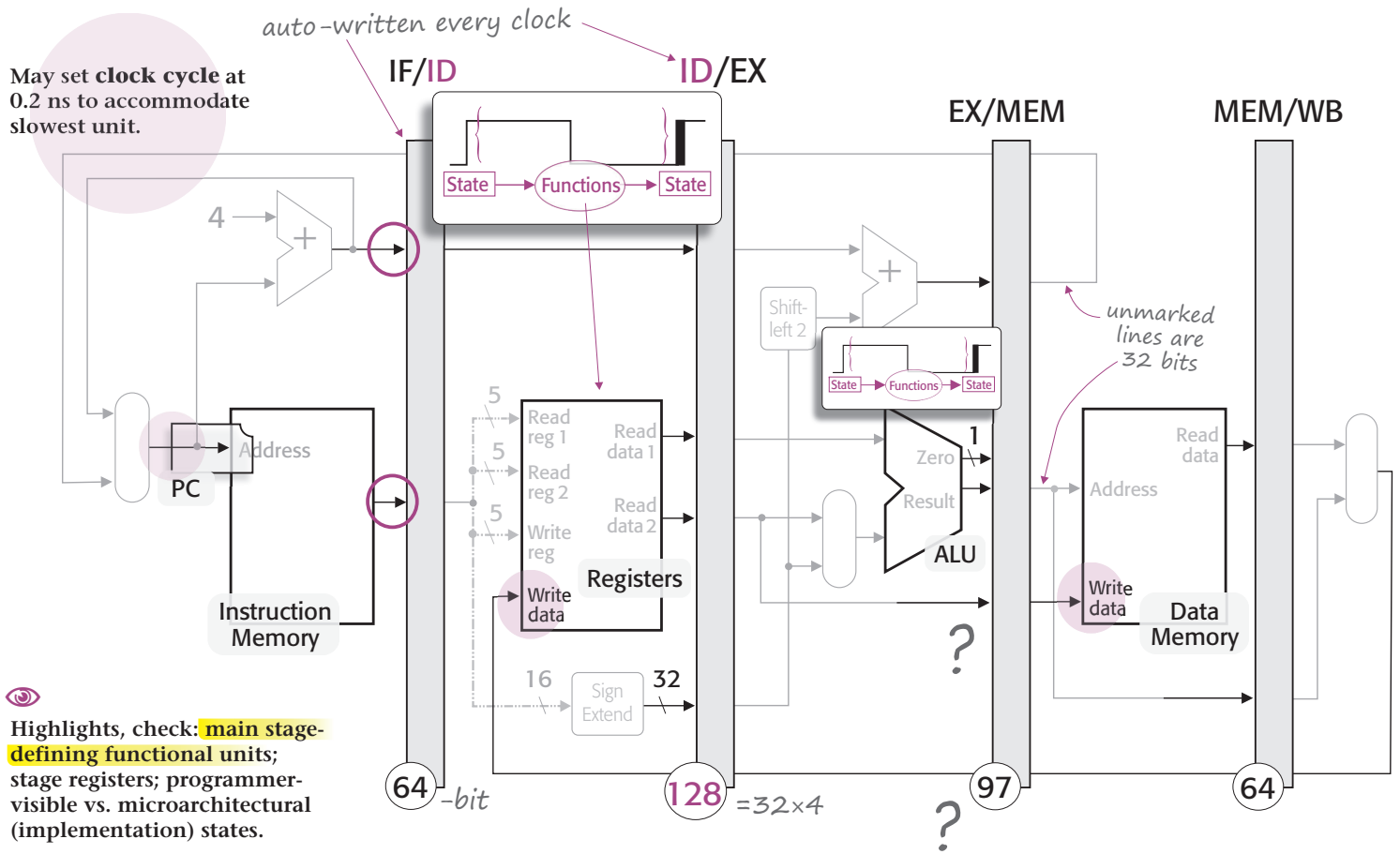
A fixed clock hides (masks) continuous, unpredictable physical timing of devices, units, and paths.

**Quiz**

What about input change during read part of clock cycle?



# An Execution Pipeline



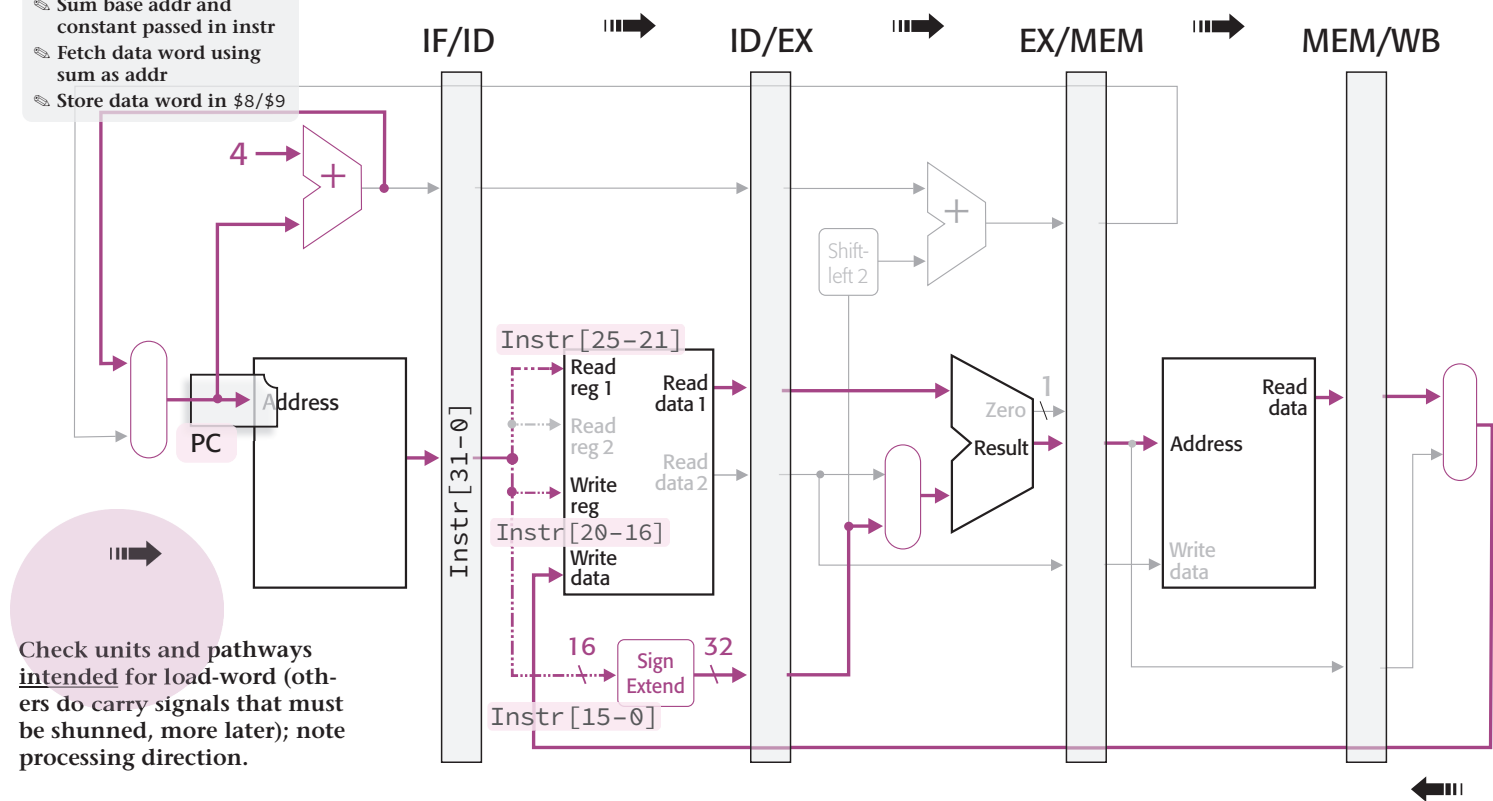
Highlights, check: **main stage-defining functional units; stage registers; programmer-visible vs. microarchitectural (implementation) states.**



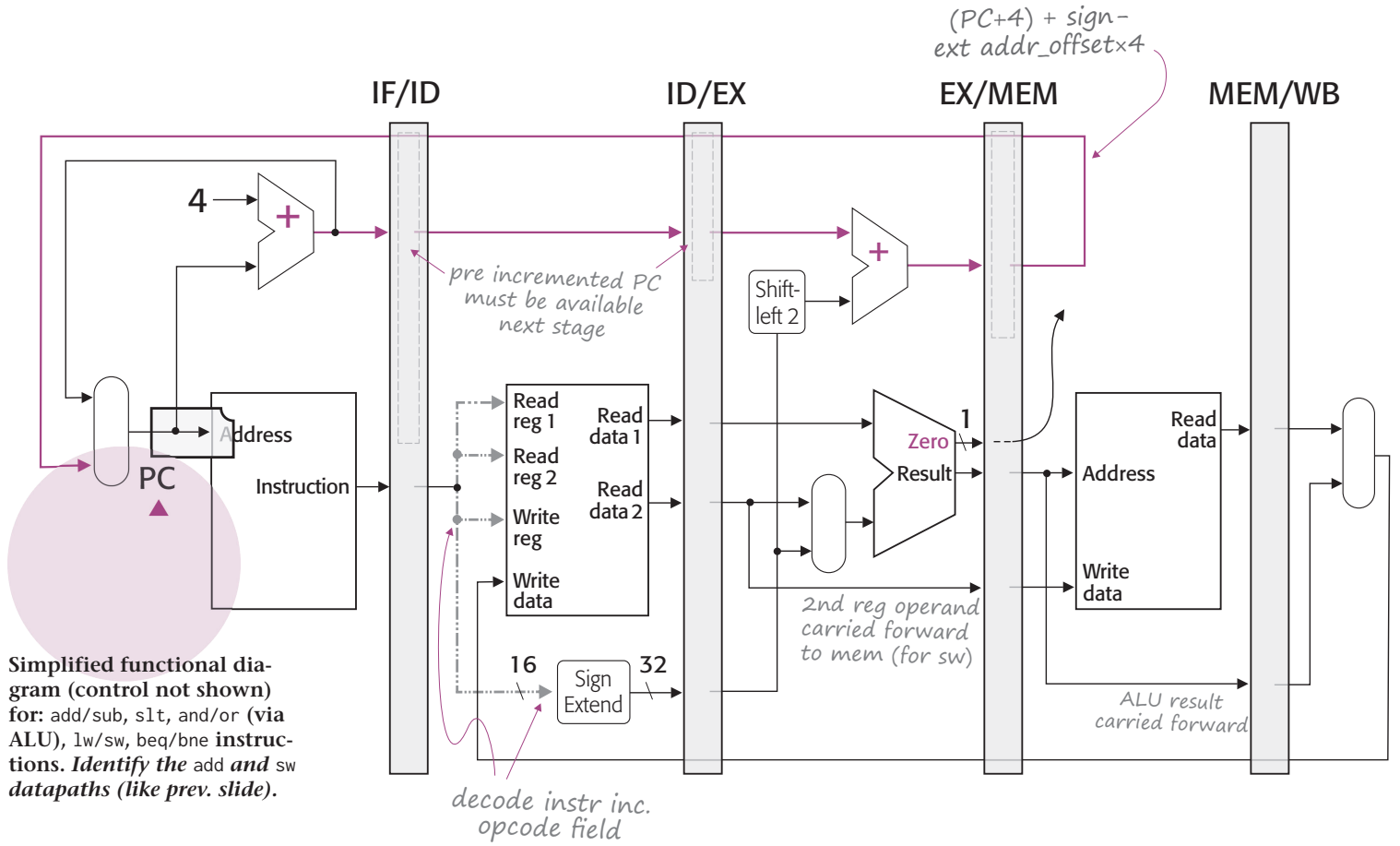
# Pipelining Load-Word Load Datapath

- Fetch instr word from instr memory
- Load base addr from \$1
- Sum base addr and constant passed in instr
- Fetch data word using sum as addr
- Store data word in \$8/\$9

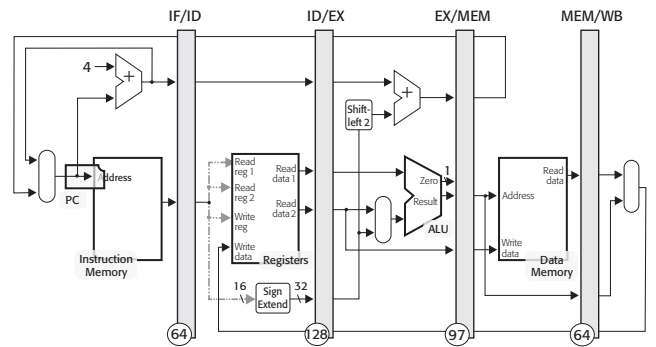
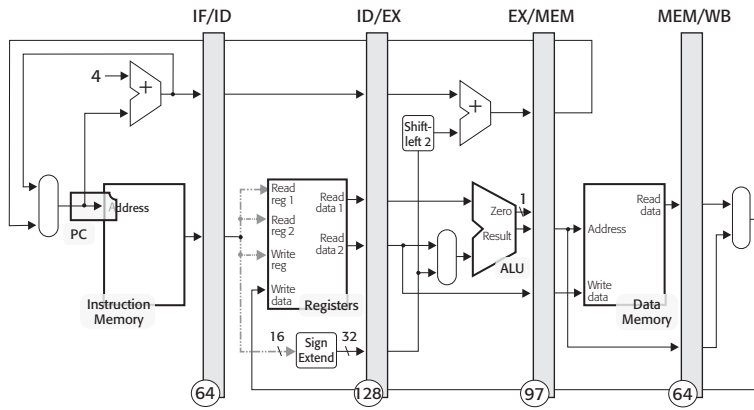
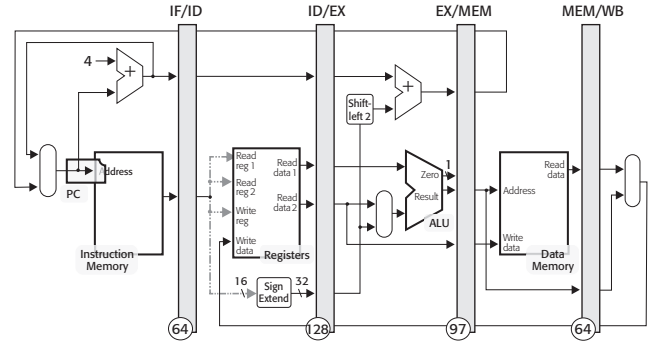
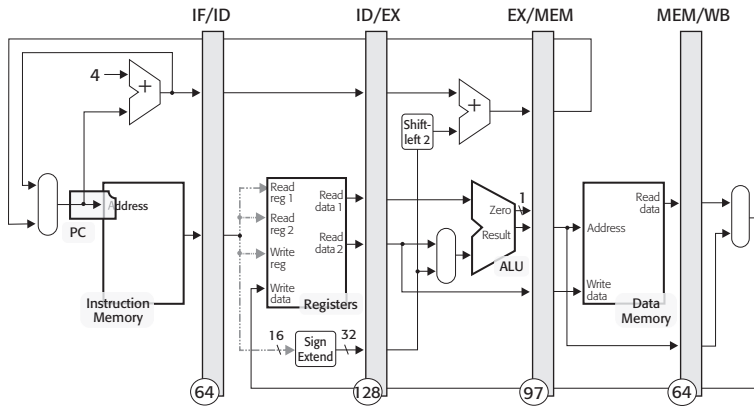
INSTR SPECS



# A Simple MIPS Datapath



# Practice Sheet



# Pipeline Hazards

⇒ [Pipeline] Conflicts

Realized speedup will be less than the ideal: 1) stages may not be perfectly balanced, and, more crucially, 2) one instr/cycle issue may not always be possible.

⇒ **Prevent instr issue every cycle**

⇒ **Types: depending on cause**

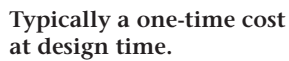
 **Structural: functional unit conflict**

 **Data: operand dependence**

 **Control: decision dependance**

Pipelining load word example:  
Speedup = time to complete instr (non-pipelined) / time to complete instr (pipelined). Note in pipelined case, 1st instr goes through m-1 stages after which one instr will complete per cycle, i.e., n instrs should complete in  $0.8 + n \times 0.2$ , therefore 10 instrs complete after  $0.8 + 10 \times 0.2 = 2.8$  ns in  $4+n = 4+10 = 14$  cycles if cycle set at 0.2 ns.

## ⇒ [Instr] Orthogonality



SIMPLIFIED

# Data Hazards

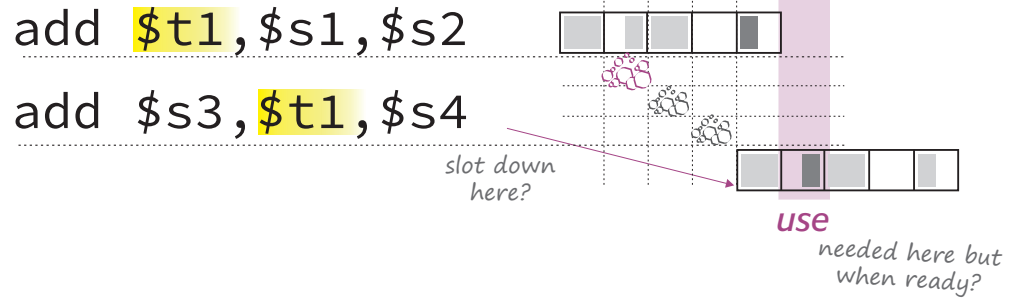
⇒ Stall cycle (bubble)



Even a casual look shows the 2nd add can't start the following cycle (why? study figure).

## ⇒ Example

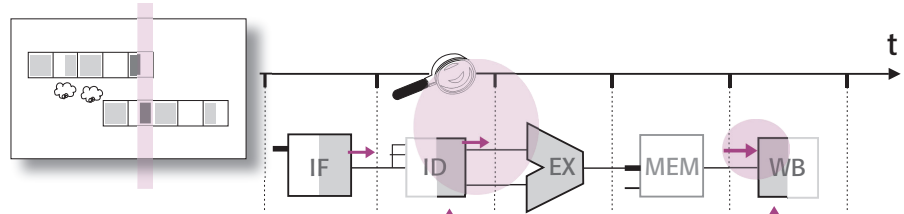
The data hazard seems to cause 3 delay cycles more than a **hazard-free** sequence, each a missed opportunity for more instruction-level parallelism.



## ⇒ Pipeline representation: add

### Exercise

The **read-after-write** data hazard will actually cause less than 3c delay in MIPS. Why? *Hint: lookup MIPS-2000 actual 5-stage timing.*



# Resolving Pipeline Hazards Forward Results

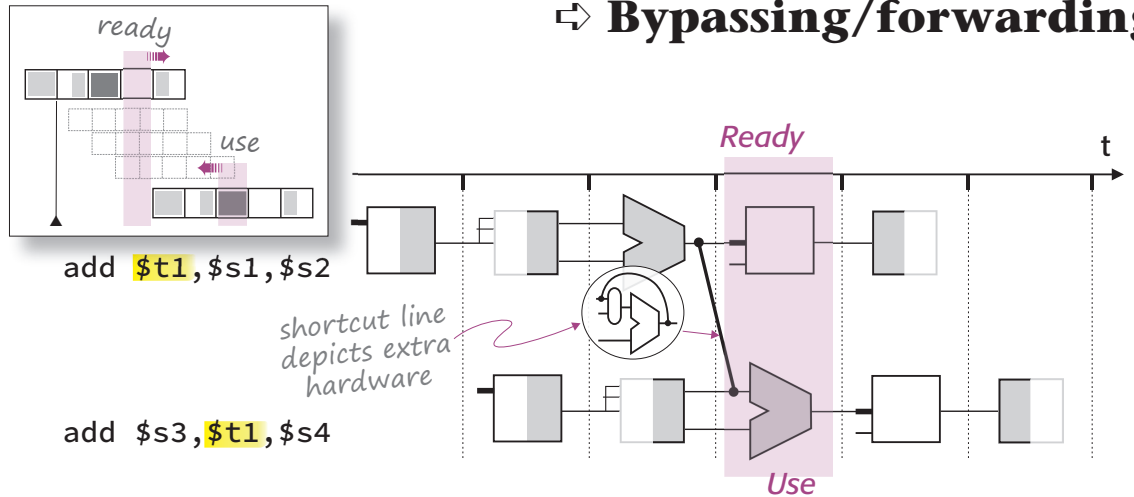
⇒ Bypassing/forwarding

Actual *ready* and *use* stages may be aligned if extra hardware were available to get ALU result early.

Additional paths, MUX and control, mostly; note ALU result must be recorded somewhere for next stage.

## Exercise

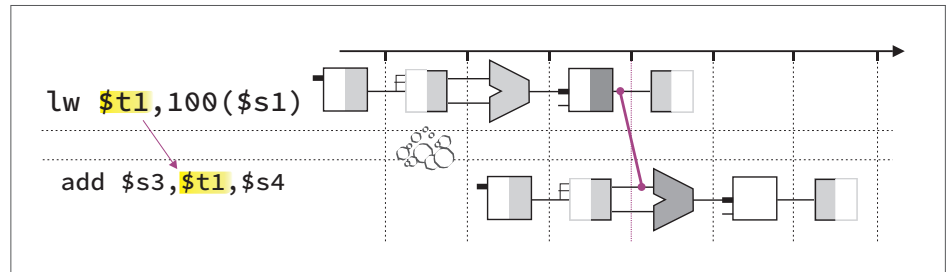
Construct a **write-after-read** example from these instructions. (Hint: tricky, play with *\$t1* position.)



**Forwarding** can't solve all data dependance problems.

## Exercise

Draw the diagram for the **load-use** hazard without forwarding. How many bubbles?



# Resolving Pipeline Hazards Reorder Instructions

🔑 ⇔ **Cycles-per-instruction (CPI)**

High-level code compiled into a “sensible” (symbolic) machine code based on register allocation in figures.

Assume high-level variables in memory (fig) starting at address loaded in \$1

**Exercise**  
Suggest a better schedule hazard-wise. (Ans. bottom corner.)

\$t3  
\$t1 \$t2  
A = B + E;  
\$t5 \$t4  
C = B - F;

byte offset	
\$1 ← 0	B
4	E
8	F
12	A
16	C

```
lw    $t1, 0($1)    # fetch B
lw    $t2, 4($1)    # fetch E
add   $t3, $t1,$t2  # A←B+E
sw    $t3, 12($1)   # store A
lw    $t4, 8($1)    # fetch F
sub   $t5, $t1,$t4  # C←B-F
sw    $t5, 16($1)   # store C
```

Reproduction (concept P&H 1998-2012)

## Exercise

1. Draw a pipeline diagram, determine number of cycles to complete sequence in each case
  - (a) Without forwarding or reordering (worst case). Hint: 5 data hazards.
  - (b) With forwarding alone (hardware-only solution)
  - (c) With both forwarding and reordering (pipeline-aware compiler)
2. What's the **CPI** for ideal **hazard-free** pipeline?
3. What is the **speedup** in each case? (CPI case/CPI hazard-free)

? ▶

A better schedule: the independent loads of vars (B, E, F), next the arithmetic, then the stores.

**Quiz**  
What's the CPI in a perfect pipeline in the long run?



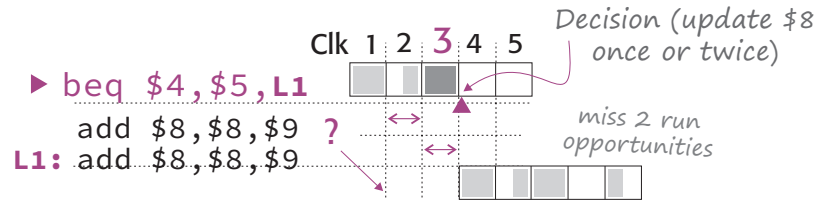
# Control Hazards

⇒ Delay slot

Change \$8 depending on a condition; pipeline does not know which add should it issue next until the condition resolves.

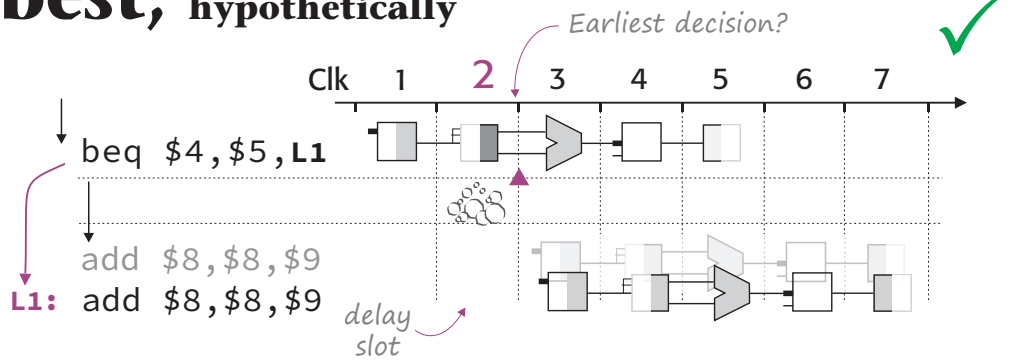
Test difference \$4-\$5 (an ALU op) for zero, so must wait until cycle 3 ends to decide if branch taken (PC over-written via *EX/MEM*, stage 3 results latch, just in time for a fetch in cycle 4).

## Stall on branch



Assume test-equal hardware in ID stage/2 to allow branching a stage earlier (perhaps xor register operands to pre-determine if equal).

## At best, hypothetically



One **delay slot** still seems unavoidable even in simplest pipelining scenarios; it gets worse with complex instrs and longer pipelines.

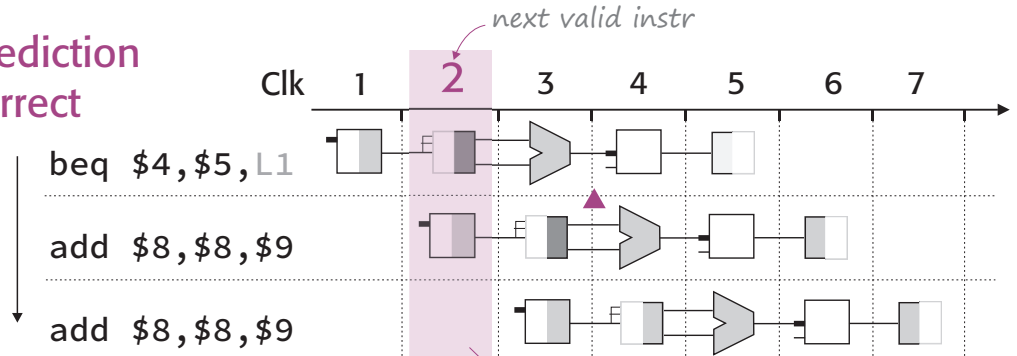
# Resolving Pipeline Hazards Branch Prediction

Hypothetical scenario, more changes needed to the ID stage + can bypass the obvious data hazard on \$8.

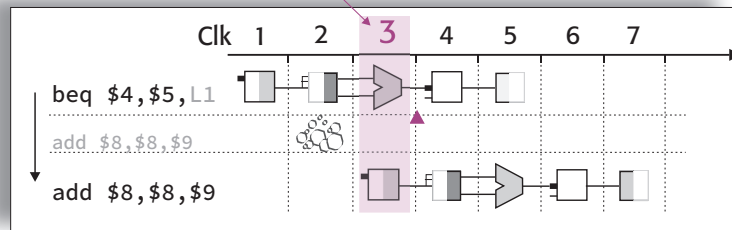
At least some of the time the pipeline will operate without a bubble (better than always having bubbles with no prediction).

## Branch always untaken (why reasonable?)

Prediction  
correct



Prediction  
wrong



# Better Prediction

⇒ Basic block

Example **Basic block** depicting a typical loop. Refer to Glossary for definition.

⇒ **Common looping pattern**  
Typical run:  $n$  taken to 1 not-taken

```
inner: add $t1,$zero,$zero
      add $t3,$a1,$t1
      lw  $t3,0($t3)
      bne $t3,$t4,skip
      addi $s0,$s0,1
skip:  addi $t1,$t1,4
      bne $t1,$a3,inner
```

Reproduction (concept P&H 1998-2012)

Diagram illustrating a common looping pattern (Basic block) with a back branch. The code shows a loop labeled **inner:** that branches to **skip:** if not equal (**bne**). The **skip:** block increments **\$t1** and branches back to **inner:** if not equal (**bne**). A blue arrow indicates the back branch from **skip:** to **inner:**. A red 'x' marks the **bne \$t3,\$t4,skip** instruction, and a green checkmark marks the **bne \$t1,\$a3,inner** instruction. The condition for the back branch is  $\$t1 < (\$a3 \text{ addr} > 0)$ .

Basic blocks offer opportunities to overcome pipeline hazards.

⇒ **Prediction**  
Back branch always taken

# Resolving Control Hazards

Simple **prediction** (guessing) improves pipeline performance by removing bubbles in some cases; better guessing should improve even more.

## ⇒ Branch prediction

✎ Static (fixed), i.e., same, prediction

✎ Dynamic (changing) prediction

Reordering instrs to avoid data and control hazards is a form of compiler-assisted resolution (e.g., transparently fill as many branch delay slots as possible).

## ⇒ Compiler-assisted resolution

**Quiz**  
What is the drawback of the compiler assisted approach for a pipeline with more stages?

## ⇒ Speculative execution (later)

# SIMPLIFIED Control

A control unit must output, in each clock cycle, the required signals in response to an op-code to correctly: a) pick paths, b) read/write state, or c) select ALU functions.

Recall, physical devices in a computer naturally encode and transform a binary state (bits).

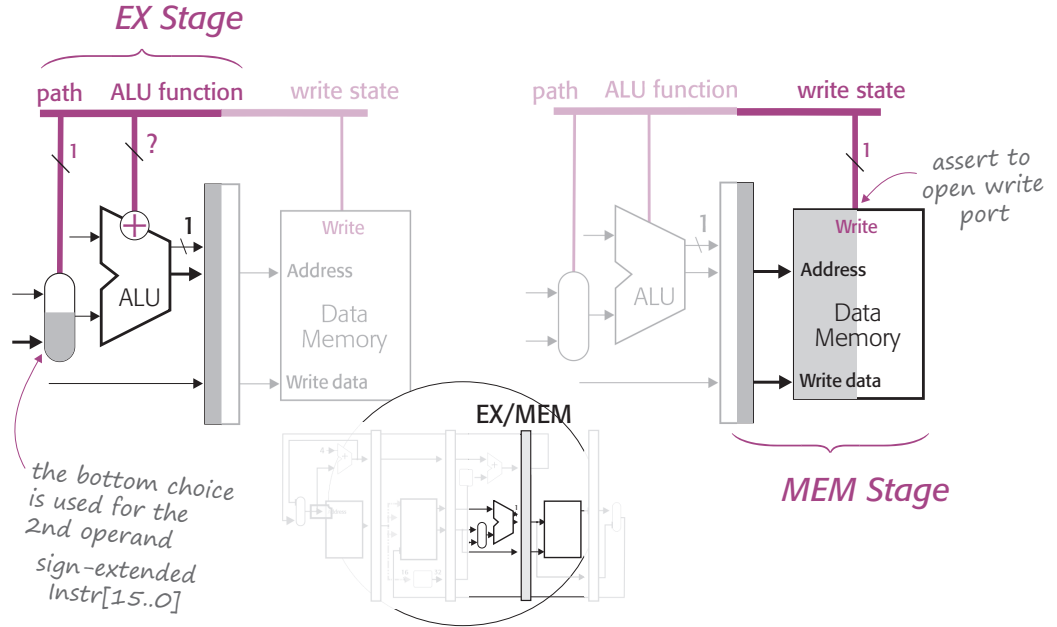
- ⇒ Machine state
- ⇒ Program state
- ⇒ Control word

In a scenario of two successive cycles/stages, the first one performs an ALU operation (add); the following writes a word in memory based on ALU output from the previous cycle.

**Quiz**  
Determine the *ALU func* control bits (how many, source)?  
Guess the instruction.

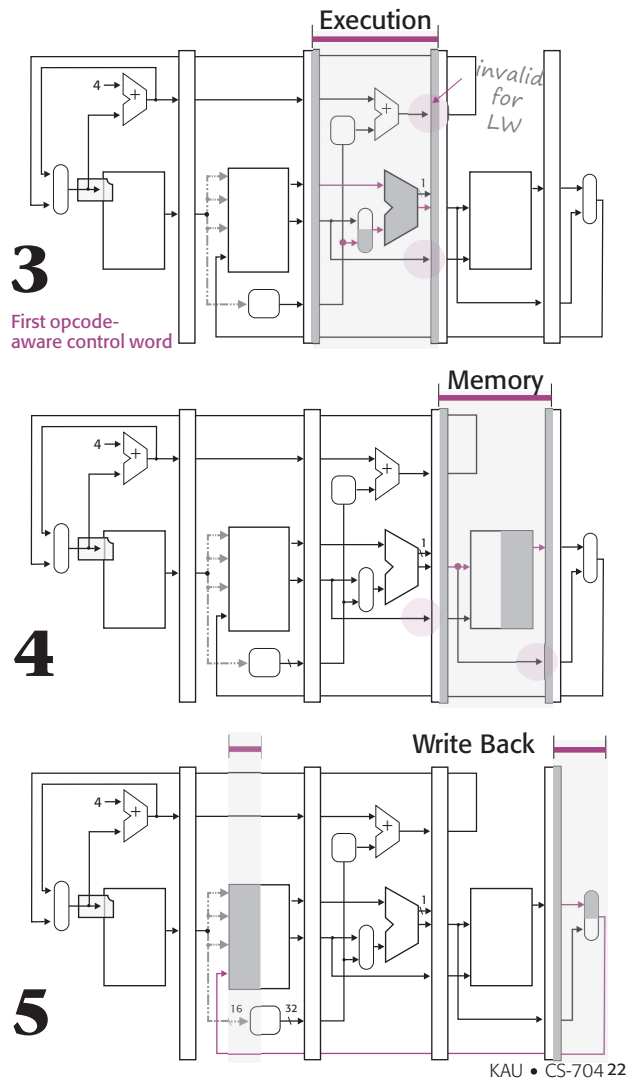
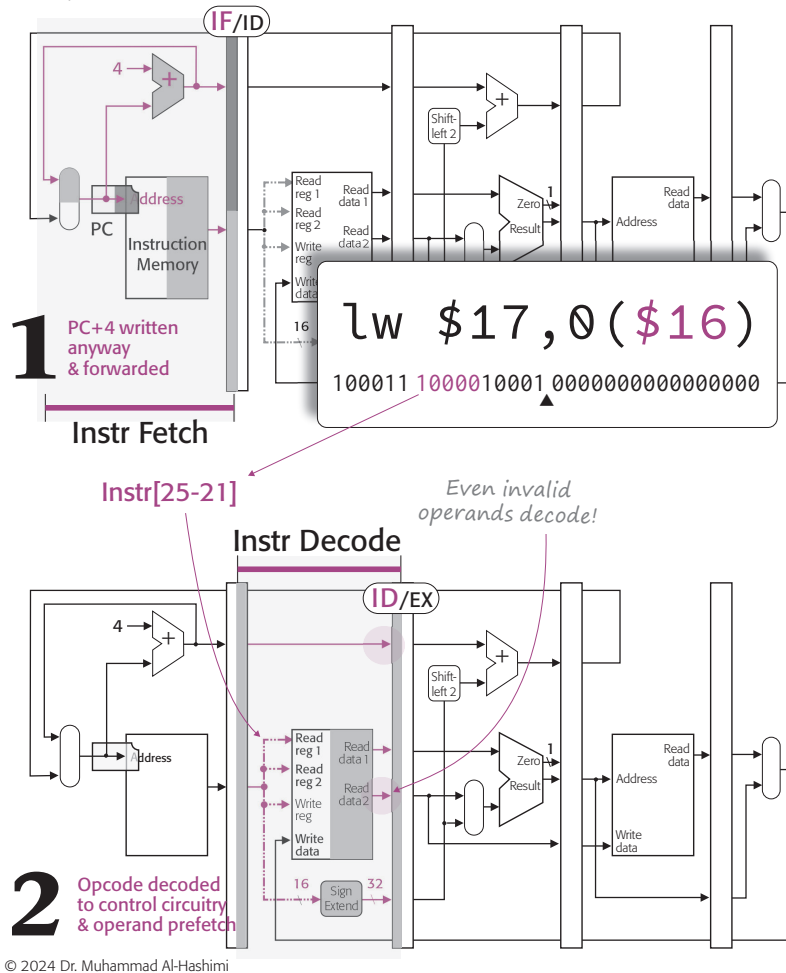
Electronic signals will propagate in all circuits and invalid bits may occur everywhere in a datapath.

A **control word** ensures desired functions will compute correctly, and only wanted results are recorded at the end of each cycle in *some* registers or memory.



# How It Works

Reproduction (concept P&H 1998-2012)



# Control How It Works

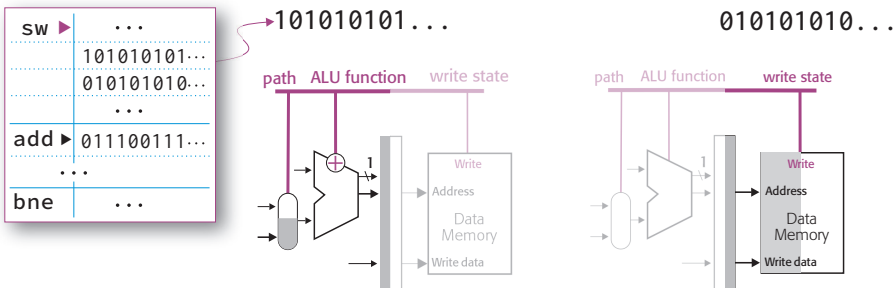
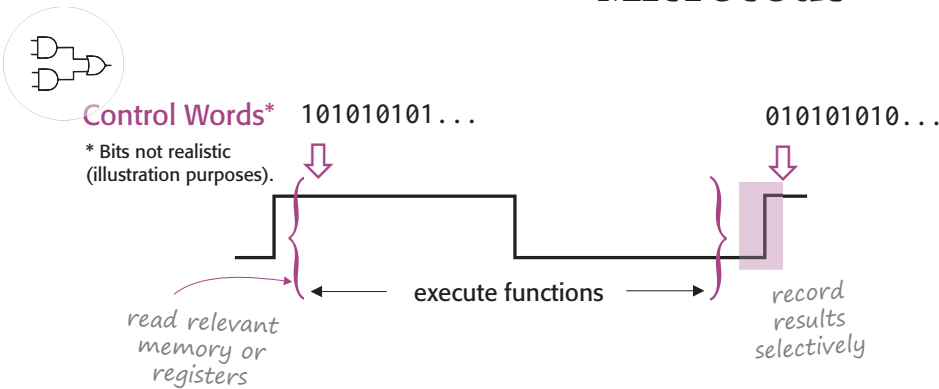
## ⇔ Microcode

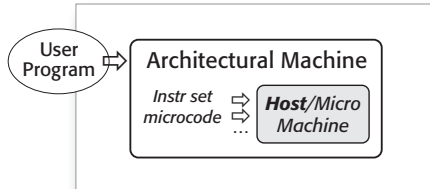
The control unit must output a suitable **control word** in each clock cycle.

Sequence of control words needed to exec an instruction may be generated on-the-fly by fixed logic (**hard-wired**), or stored in a memory (**microcode**).

**Quiz**  
Suggest situations where one approach to control may be preferable to the other.

Note relatively cheap to add/modify instructions in the microcode.





# Control Summary



Realistically, a multi-stage control must output for all active stages, in every cycle, control words for different instructions in different execution stages + handle forwarding, stalls, exceptions, and flow predictions.

## ⇒ Control function, a model

An embedded [programmable] micro machine to host (i.e., run) an instr set



The primary duty of the micro-architectural state is maintaining the **program state** as contractually agreed with users at the ISA level.

## ⇒ Microarchitectural state

Pre-agreed timing relative to a fixed clock + a control that ensures correct bits at inputs and selects valid bits to record leads to a manageable physical machine to implement an abstract FSM.

## ⇒ Control + clocking

Simplified (relatively), reliable machine

*A fixed clock hides (masks) continuous, unpredictable physical timings ...* Recall



# Processor Instructions Design to Pipeline

## Quiz

What is **interlocking** in context of pipelining? (Look up the name MIPS.)

## Exercise

Explain the impact of each feature on a pipeline, give examples from the original MIPS.

“Instr sets can either simplify or make life harder for pipeline designers”

 P&H

*Same applies to compiler writers*

⇒ **Fixed length**

Operands in the same places make **pre-fetch** possible, that is, to decode independent of opcode.

⇒ **Regular** (e.g., few, similar encodings)

⇒ **Load-store**

⇒ **Memory operand alignment**

## Exercise

Discuss the costs of pipelining instructions relative to sequential execution.

⇒ **Single final write back**

# Technical Essay Assignment






See essay descriptions in *Technical Essay Assignments* conversation for this semester in the course group.

Check the RISC-V links in the Reading File (under Background).

Check the case study links in the Reading File (under Background) for REQUIRED ARM material.

Check the group for specifications.

## Essay 1: pick one from 1-5

-  Discuss how well the classic MIPS fits RISC criteria by Colwell et al.
-  Discuss how well MIPS fits Wulf's principles
-  Compare ISA: classic MIPS to RISC-V
-  Compare dynamic scheduling/exec schemes
-  ARM pipelines vs. classic 5-stage MIPS

## Expectations

Contribute small essay in course group






# Concept Review

# Pipeline Performance

Performance ultimately depends on how well **conflicts** (=hazards) are handled.

⇒ **Conflicts due to dependencies arise while executing instruction streams causing delays**

⇒ **Conflict types**

-  Datapath resource
-  Program dataflow
-  Program decision

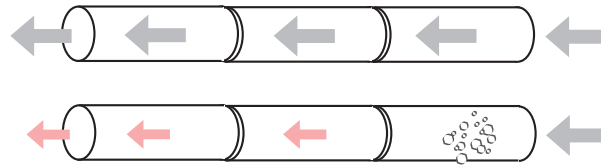


# Concept Review Pipelined Execution

## ⇒ Misprediction

A logical "pipeline" is created by overlapping instr exec

Flow is even in a perfect execution pipeline, like a real one, each pipe contributes equally to the flow.



Stall cycles, like bubbles in a real pipeline, reduce the throughput

More to actual latency than nominal number of stages; more to come.

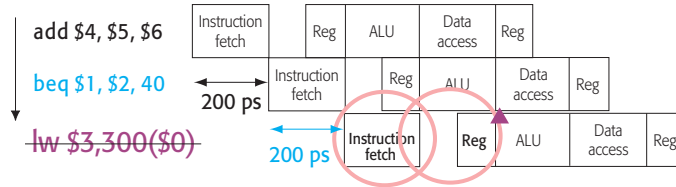
## ⇒ Instr latency, ideally

On top of wasted cycles on wrong instructions, a pipeline must be *flushed*, e.g., state rolled back.

## ⇒ Misprediction cost

### Exercise

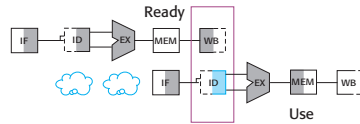
Discuss the misprediction cost in long pipelines and complex decision scenarios.



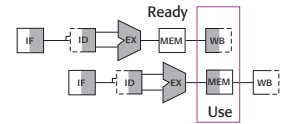
# Resolving Hazards Exercise

```

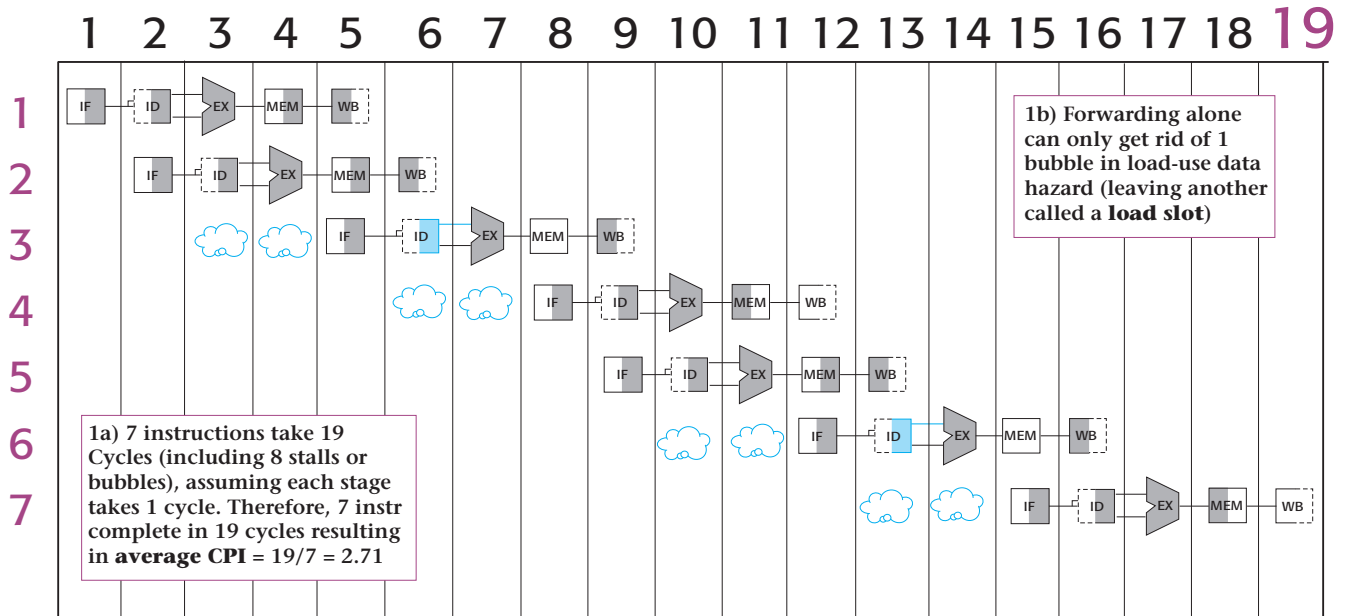
1 lw    $t1, 0($t0)
2 lw    $t2, 4($t0)
3 add   $t3, $t1, $t2
4 sw    $t3, 12($t0)
5 lw    $t4, 8($t0)
6 add   $t5, $t1, $t4
7 sw    $t5, 16($t0)
    
```



Use stage aligned with or after ready stage (as in this case, why?), never before!



Original—note **write-after-write** hazard



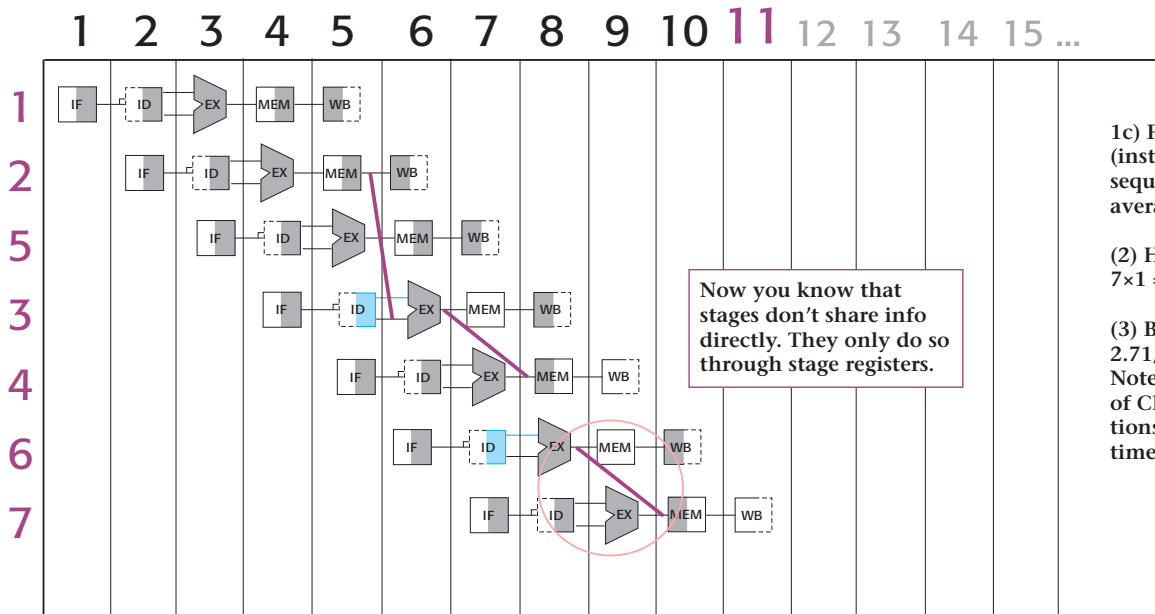
# Resolving Hazards Exercise

```
1 lw    $t1, 0($t0)
2 lw    $t2, 4($t0)
5 lw    $t4, 8($t0)
3 add   $t3, $t1, $t2
4 sw    $t3, 12($t0)
6 add   $t5, $t1, $t4
7 sw    $t5, 16($t0)
```

Re-ordered with forwarding



Execution timings and CPI here are ideal. We will see why CPI must be measured carefully when we factor in memory.



1c) From diagram: 11 c (instead of 19 in original sequence), therefore, the average **CPI** =  $11/7 = 1.57$

(2) Hazard-free formula:  $4 + 7 \times 1 = 11c$  (ideally).

(3) Best case **speedup** =  $2.71/1.57 = 1.73$  (+73%). Note the speedup is the ratio of CPI since same instructions using a common cycle time (more later).