# CPU Pipelining

CS/COE 1541 (Fall 2020) Wonsun Ahn



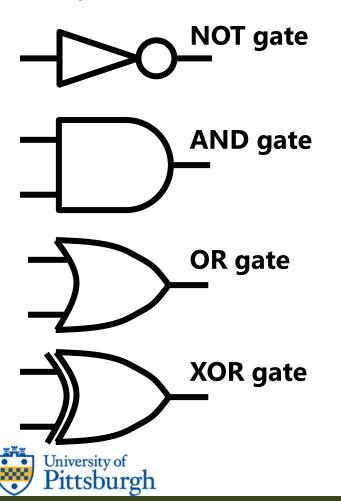
# Clocking Review

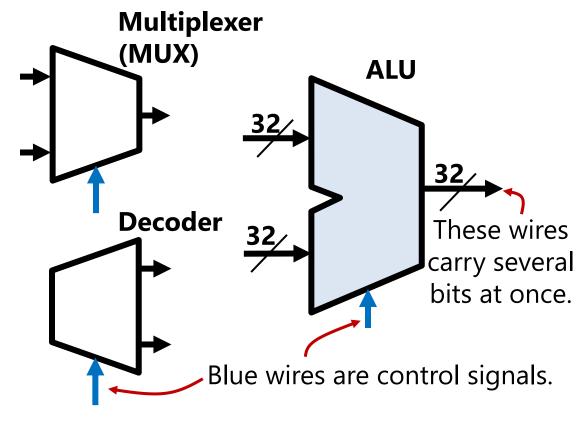
Stuff you learned in CS 447



#### Logic components

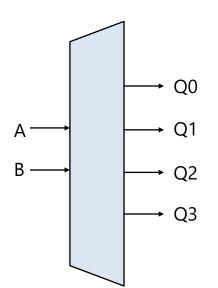
Do you remember what all these do?





#### Uses of a Decoder

- Translates a set of input signals to a bunch of output signals.
  - E.g. a binary decoder:



#### **Truth Table for Decoder**

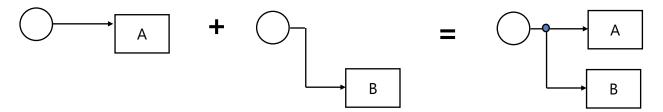
| Α | В | Q0 | Q1 | Q2 | Q3 |
|---|---|----|----|----|----|
| 0 | 0 | 1  | 0  | 0  | 0  |
| 0 | 1 | 0  | 1  | 0  | 0  |
| 1 | 0 | 0  | 0  | 1  | 0  |
| 1 | 1 | 0  | 0  | 0  | 1  |

You can come up with any truth table and make a decoder for it!

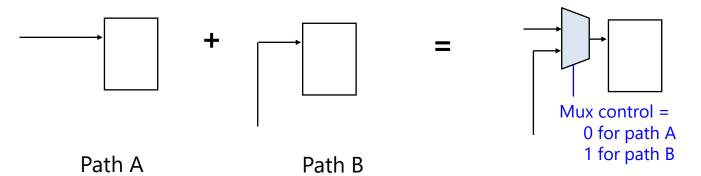


## Uses of a Multiplexer

No problem in fanning out one signal to two points



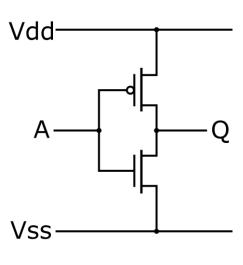
- Cannot connect two signals to one point
  - Must use a multiplexer to select between the two



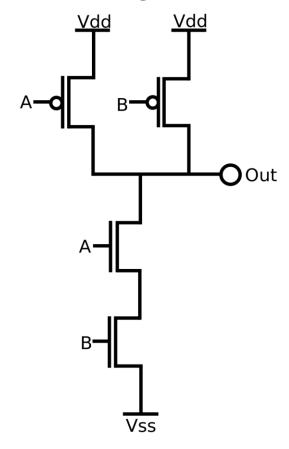


## Gates are made of transistors (of course)

NOT gate



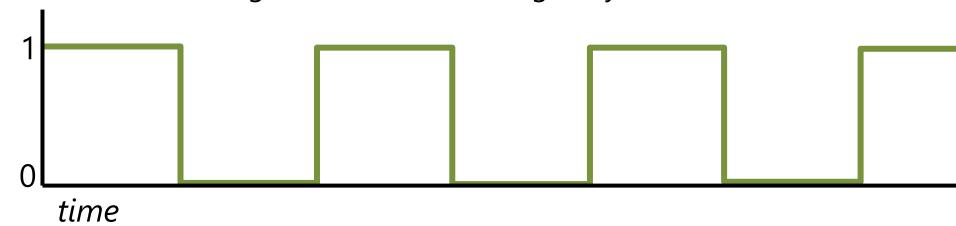
NAND gate





#### The clock signal

• The clock is a signal that alternates regularly between 0 and 1:



- Bits are latched on to registers and flip-flops on rising edges
- In between rising edges, bits propagate through the logic circuit
  - o Composed of ALUs, muxes, decoders, etc.
  - o Propagation delay: amount of time it takes from input to output



#### Critical Path

- Critical path: path in a circuit that has longest propagation delay
   Determines the overall clock speed.
  - A D Q D Q OUT

    Select

    Select
  - The ALU and the multiplexer both have a 5 ns delay
- How fast can we clock this circuit?
  - $\circ$  Is it 1 / 5 ns (5 × 10<sup>-9</sup>s) = 200 MHz?
  - $\circ$  Or is it 1 / 10 ns (10 × 10<sup>-9</sup>s) = 100 MHz?  $\checkmark$



## MIPS Review

Stuff you learned in CS 447



#### The MIPS ISA - Registers

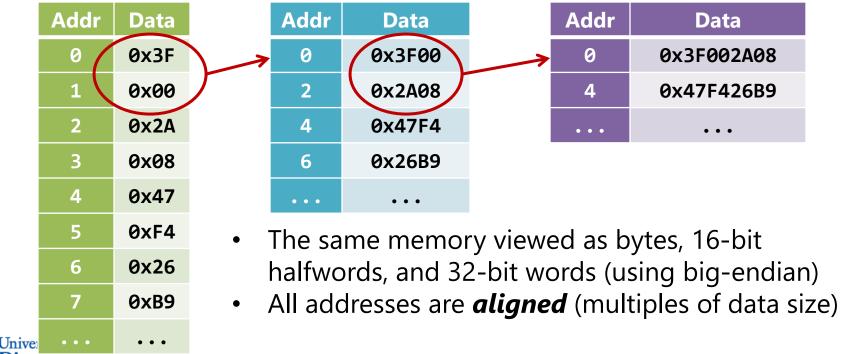
- MIPS has 32 32-bit registers, with the following usage conventions
  - o But really, all are general purpose registers (nothing special about them)

| Name      | Register number | Usage  |  |
|-----------|-----------------|--|--|
| \$zero    | 0               | the constant value 0 (can't be written)      |  |
| \$at      | 1               | assembler temporary                          |  |
| \$v0-\$v1 | 2-3             | values for results and expression evaluation |  |
| \$a0-\$a3 | 4-7             | function arguments                           |  |
| \$t0-\$t7 | 8-15            | unsaved temporaries                          |  |
| \$s0-\$s7 | 16-23           | saved temporaries (like program variables)   |  |
| \$t8-\$t9 | 24-25           | more unsaved temporaries                     |  |
| \$k0-\$k1 | 26-27           | reserved for OS kernel                       |  |
| \$gp      | 28              | global pointer                               |  |
| \$sp      | 29              | stack pointer                                |  |
| \$fp      | 30              | frame pointer                                |  |
| \$ra      | 31              | return address                               |  |



#### The MIPS ISA - Memory

- MIPS is a RISC (reduced instruction set computer) architecture
- It is also a *load-store* architecture
  - All memory accesses performed by load and store instructions
- Memory is a giant array of 2<sup>32</sup> bytes



#### The MIPS ISA - Memory

• Loads move data *from* memory *into* the registers.

0x0000BEEF

0x00000004

Registers

This is the address, and it means "the value of \$s4 + 8."

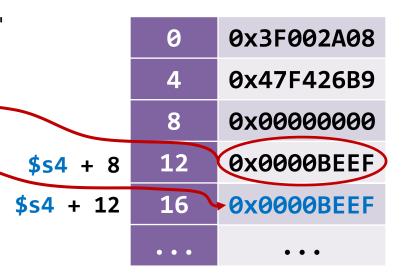
lw

SW

• Stores move data *from* the registers *into* memory.

sw (\$t0), 12(\$s4)

\$t0 is the SOURCE!



Memory



**S4** 

#### The MIPS ISA – Flow control

• Jump and branch instructions change the flow of execution.

- **j** : jumps *unconditionally*
- jumps to \_top

```
li $s0, 10
—loop:

# ....
addi $s0, $s0, -1
bne $s0, $zero, _loop
jr $ra
```

```
bne : jumps conditionally
If $s0 != $zero, jumps to _loop
If $s0 == $zero, continues to jr $ra
```

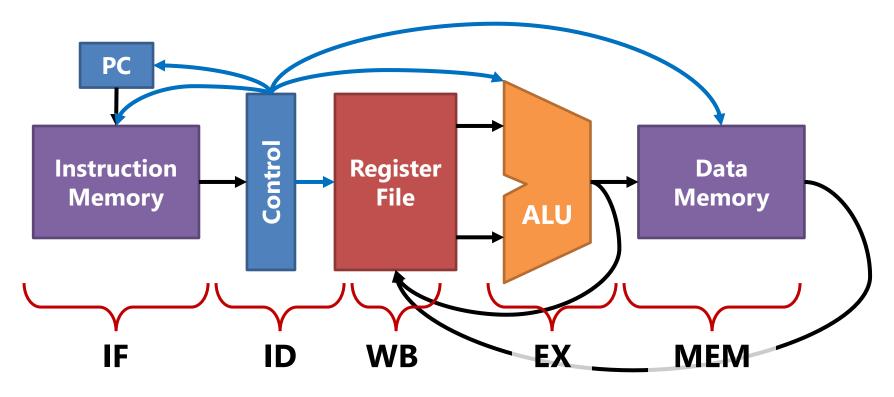


#### Phases of instruction execution

- In most architectures, there are five phases:
  - **1. IF** (Instruction Fetch) get next instruction from memory
  - 2. ID (Instruction Decode) figure out what instruction it is
  - **3. EX** (Execute ALU) do any arithmetic
  - **4. MEM** (Memory) read or write data from/to memory
  - 5. WB (Register Writeback) write any results to the registers
- Sometimes these phases are chopped into smaller stages



## A simple single-cycle implementation



• An instruction goes through IF/ID/EX/MEM/WB in one cycle



## "Minimal MIPS"



#### It's a "subset" of MIPS

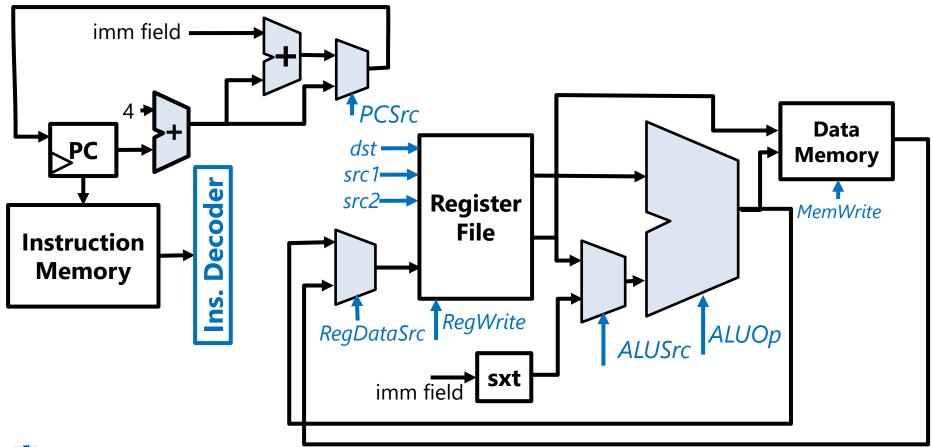
- For pedagogical (teaching) purposes
- Contains only a minimal number of instructions:
  - o lw, sw, add, sub, and, or, slt, beq, and j
  - Other instructions in MIPS are variations on these anyway
- Let's review the Minimal MIPS CPU focusing on the control signals
  - o Again, these control signals are decoded from the instruction



## The Minimal MIPS single-cycle CPU

University of **Pittsburgh** 

A more detailed view of the 5-phase implementation

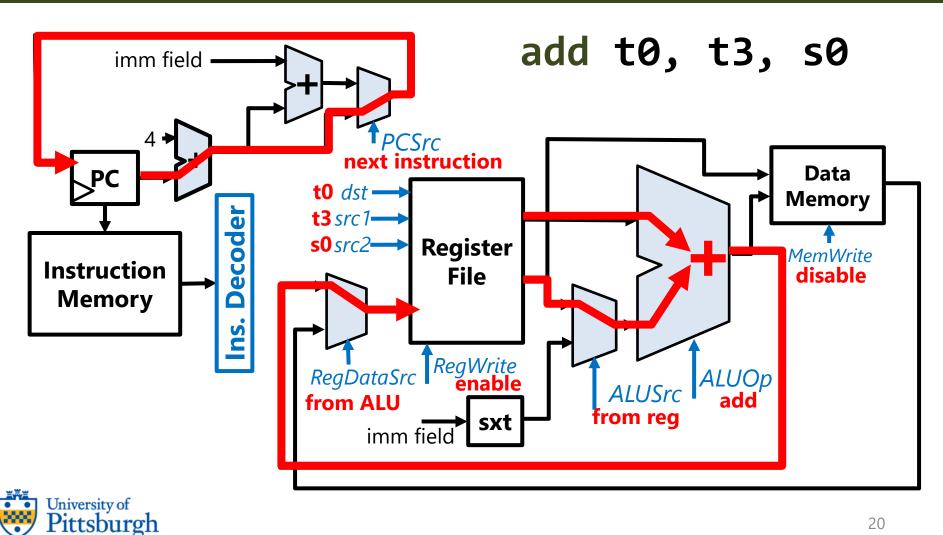


#### Control signals

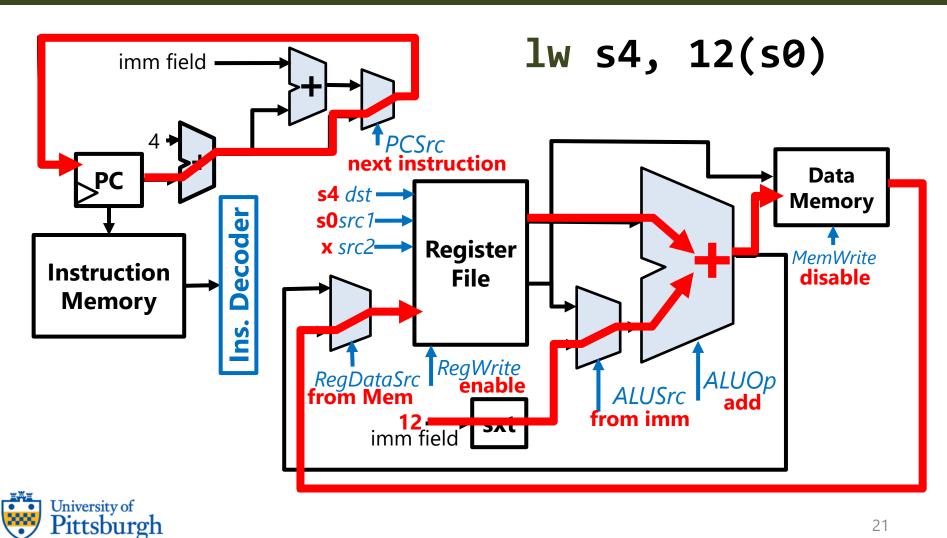
- Registers
  - RegDataSrc: controls source of a register write (ALU / memory)
  - RegWrite: enables a write to the register file
  - o src1, src2, dst: the register number for each respective operand
- ALU
  - ALUSrc: whether second operand of ALU is a register / immediate
  - ALUOp: controls what the ALU will do (add, sub, and, or etc)
- Memory
  - MemWrite: enables a write to data memory
- PC
  - PCSrc: controls source of next PC (PC + 4 / PC + 4 + imm)
- → All these signals are decoded from the instruction!



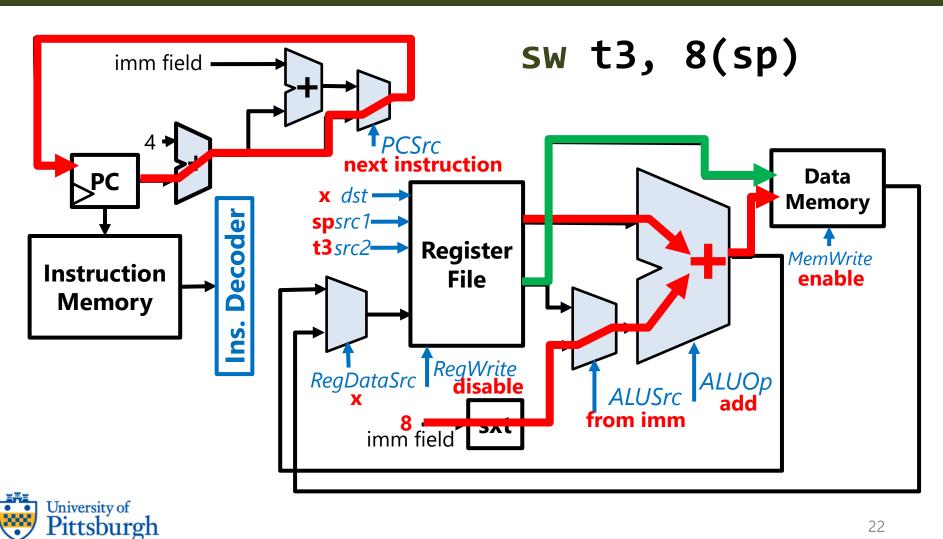
#### How an add/sub/and/or/slt work



#### How an **Iw** works

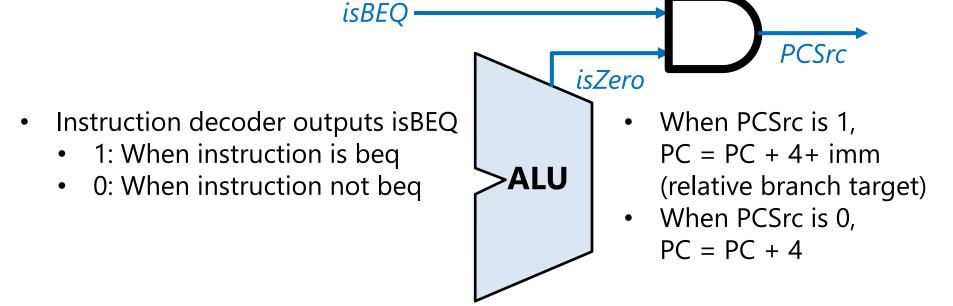


#### How an **sw** works



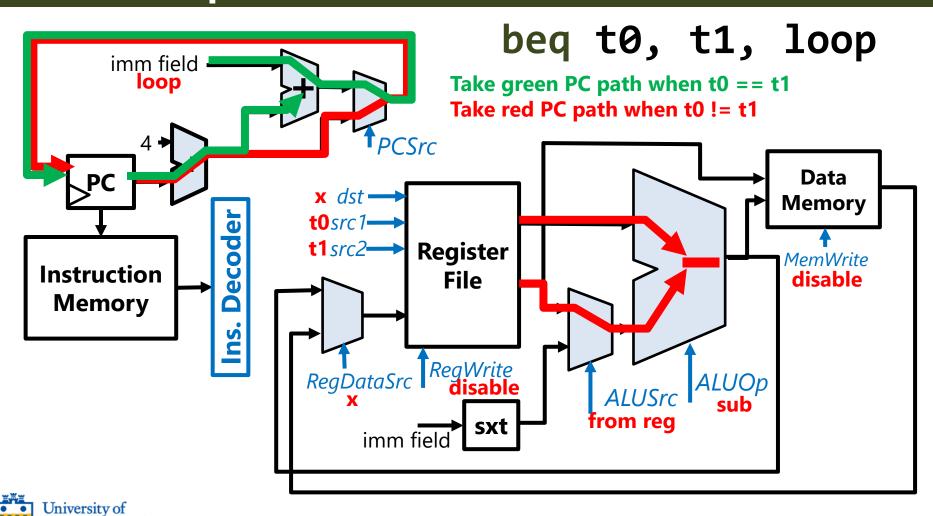
#### What about **beq**?

- Compares numbers by subtracting and see if result is 0
  - If result is 0, we set PCSrc to use the branch target.
  - Otherwise, we set PCSrc to PC + 4.



## How a **beq** works

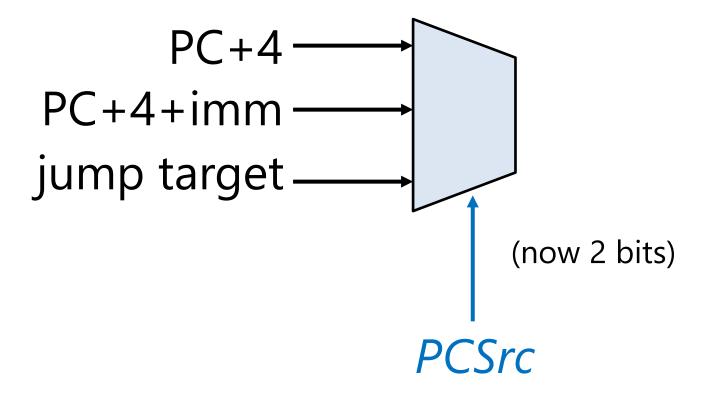
Pittsburgh



#### What about j?

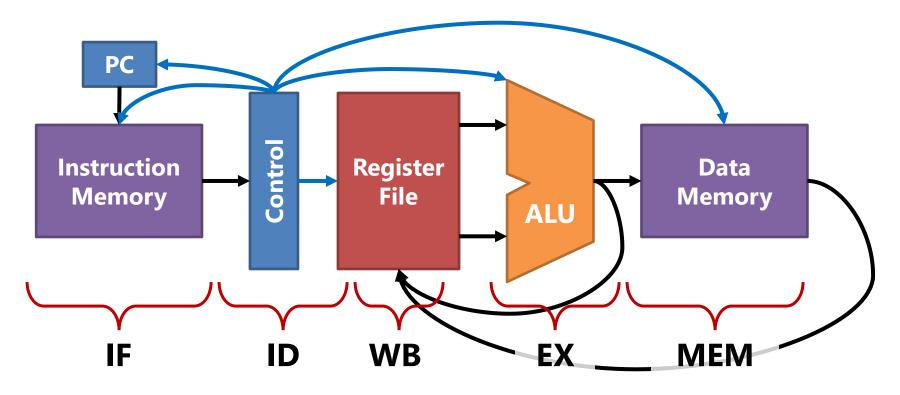
We have to add another input to the PCSrc mux.

j top





## A Single-cycle Implementation is not Optimal



- Why? Since the *longest* critical path must be chosen for cycle time
  - And there is a wide variation among different instructions



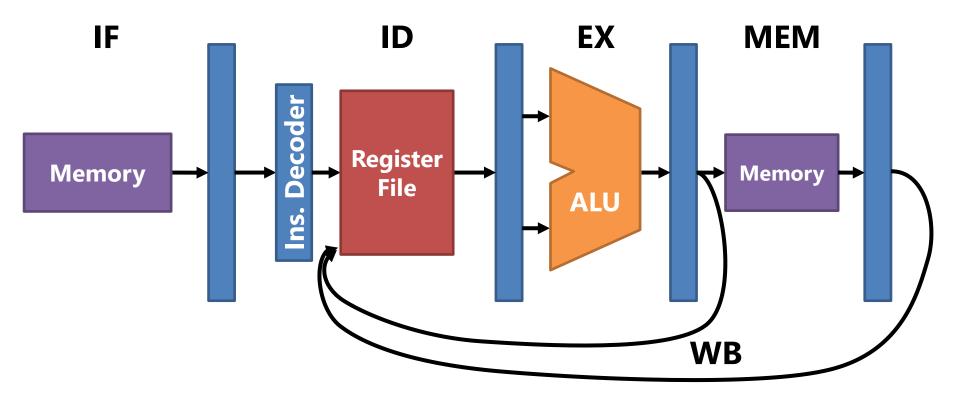
## A Single-cycle Implementation is not Optimal

- In our CPU, the **lw** instruction has the longest critical path
  - Must go through all 5 stages: IF/ID/EX/MEM/WB
  - Whereas add goes through just 4 stages: IF/ID/EX/WB
- If each phase takes 1 ns each, cycle time must be 5 ns:
  - Because it needs to be able to handle lw, which takes 5 ns
  - o **add** also takes 5 ns when it could have been done in 4 ns
- Q) If **Iw** is 1% and **add** is 99% of instruction mix, what is the average instruction execution time?
- A) Still 5 ns! Even if **Iw** is only 1% of instructions!



## A Multi-cycle Implementation

• It takes one cycle for each phase through the use of internal latches





#### A Multi-cycle Implementation is Faster!

- Now each instruction takes different number of cycles to complete
  - Iw takes 5 cycles: IF/ID/EX/MEM/WB
  - o add takes 4 cycles: IF/ID/EX/WB
- If each phase takes 1 ns as before:
  - Iw takes 5 ns and add takes 4 ns
- Q) If **lw** is 1% and **add** is 99% of instruction mix, what is the average instruction execution time?
- A) 0.01 \* 5 ns + 0.99 \* 4 ns = 4.01 ns (25% faster than single cycle)
- \* Caveat: delay due to the added latches not shown, but net win

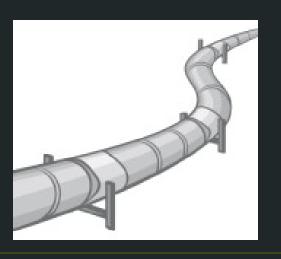


#### And we can do even better!

- Did you notice?
  - When an instruction is on a particular phase (e.g. IF) ...
  - o ... other phases (ID/EX/MEM/WB) are not doing any work!
- Our CPU is getting chronically underutilized!
  - o If CPU is a factory, 80% (4/5) of the workers are idling!
- Car factories create an assembly line to solve this problem
  - No need to wait until a car is finished before starting on next one
  - Our CPU is going to use a *pipeline* (similar concept)



# Pipelining Basics





## Improving Washer / Dryer / Closet Utilization

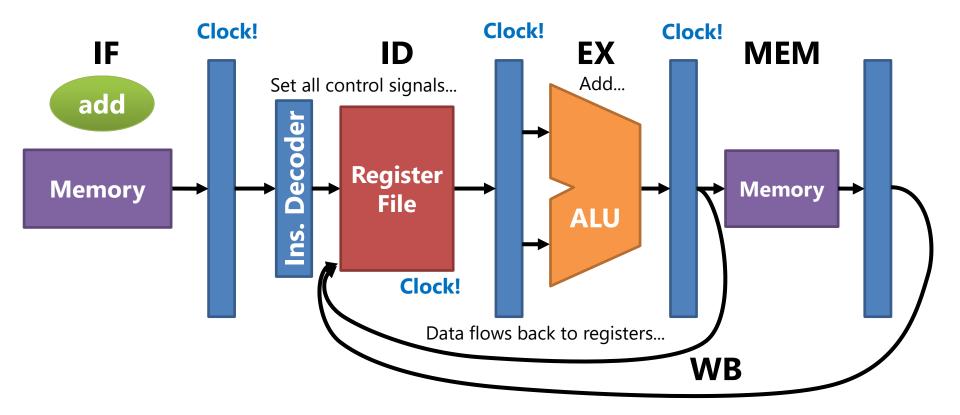
- If you work on loads of laundry one by one, you only get ~33% utilization
  If you form an "assembly line", you achieve ~100% utilization!





#### Multi-cycle instruction execution

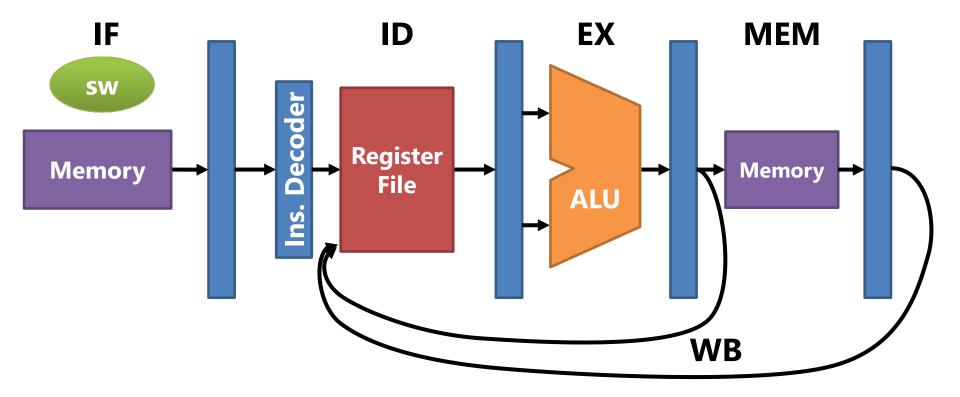
• Let's watch how an instruction flows through the datapath.





#### Pipelined instruction execution

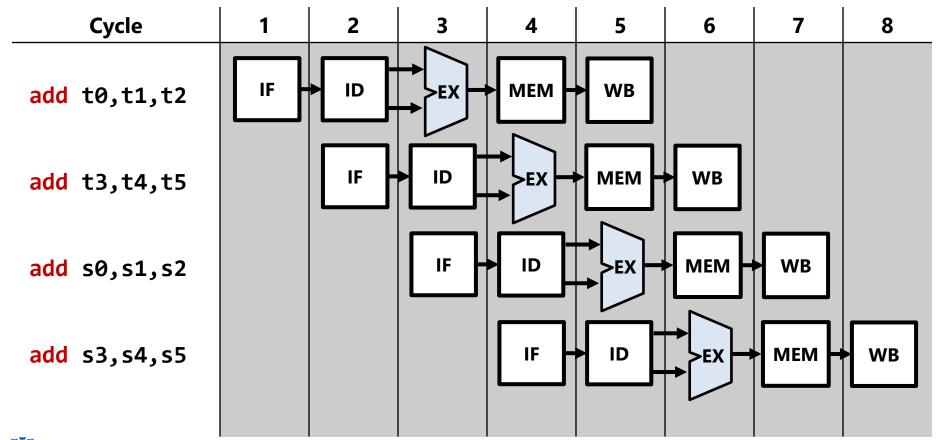
Pipelining allows one instruction to be fetched each cycle!





#### Pipelining Timeline

• This type of parallelism is called *pipelined parallelism*.





#### A Pipelined Implementation is even Faster!

- Again each instruction takes different number of cycles to complete
  - Iw takes 5 cycles: IF/ID/EX/MEM/WB
  - o add takes 4 cycles: IF/ID/EX/WB
- If each stage takes 1 ns each:
  - Iw takes 5 ns and add takes 4 ns
- Q) The average instruction execution time (given 100 instructions)?
- A) (99 ns + 5 ns) / 100 = 1.04 ns
  - Assuming last instruction is a lw (a 5-cycle instruction)
  - A ~5X speed up from single cycle!



# Pipelined vs. Multi-cycle vs. Single-cycle

What happened to the three components of performance?

$$\frac{\text{instructions}}{\text{program}}$$
 X  $\frac{\text{cycles}}{\text{instructions}}$  X  $\frac{\text{seconds}}{\text{cycle}}$ 

| Architecture | Instructions | СРІ | Cycle Time<br>(1/F) |
|--------------|--------------|-----|---------------------|
| Single-cycle | Same         | 1   | 5 ns                |
| Multi-cycle  | Same         | 4~5 | 1 ns                |
| Pipelined    | Same         | 1   | 1 ns                |

- Compared to single-cycle, pipelining improves clock cycle time
  - Or in other words CPU clock frequency
  - The deeper the pipeline, the higher the frequency will be

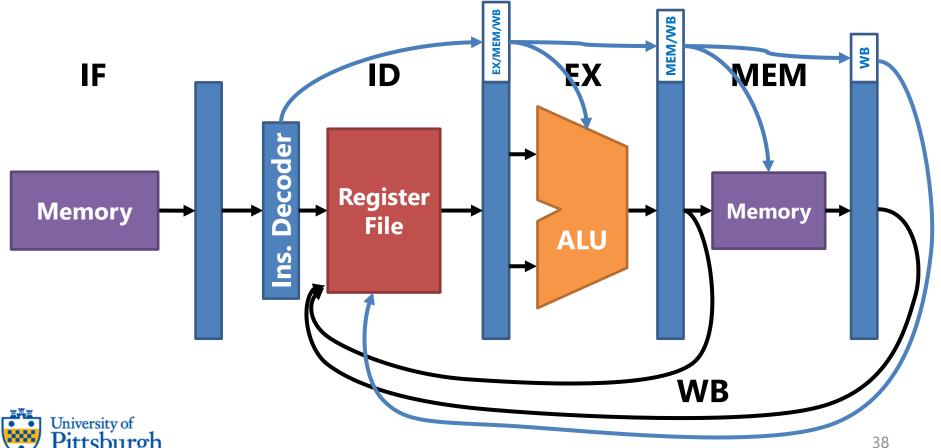
\* Caveat: latch delay and unbalanced stages can increase cycle time



## How about the control signals?

A new instruction is decoded at every cycle!

• Control signals must be passed along with the data at each stage



# Pipeline Hazards



### Pipeline Hazards

- For pipelined CPUs, we said CPI is practically 1
  - But that depends entirely on having the pipeline filled
  - o In real life, there are *hazards* that prevent 100% utilization

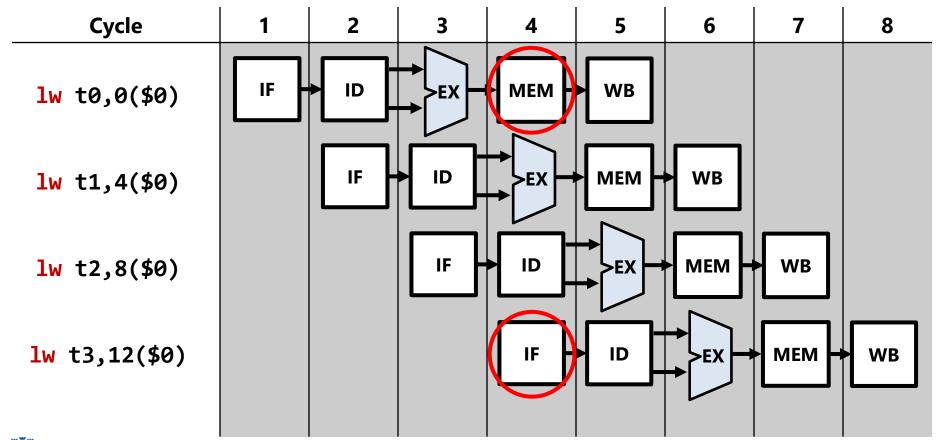
#### • Pipeline Hazard

- When the next instruction cannot execute in the following cycle
- o Hazards introduce **bubbles** (delays) into the pipeline timeline
- Architects have some tricks up their sleeves to avoid hazards
- But first let's briefly talk about the three types of hazards:
   Structural hazard, Data hazard, Control Hazard



### Structural Hazards

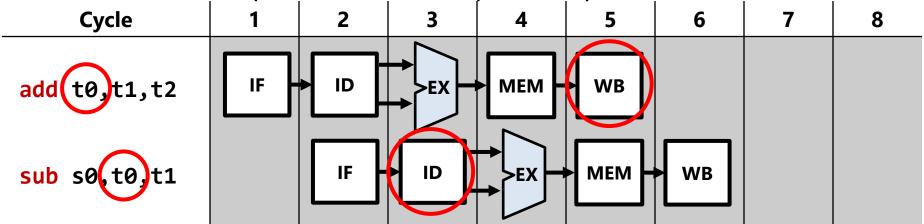
Two instructions need to use the same hardware at the same time.



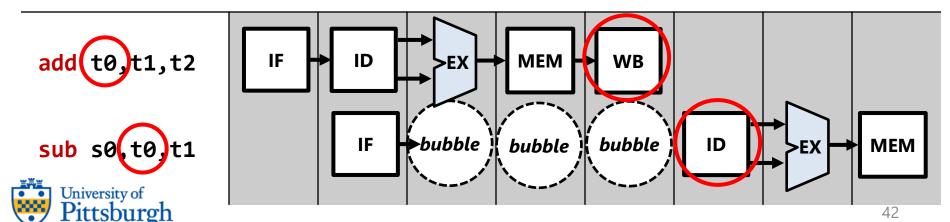


#### Data Hazards

An instruction depends on the output of a previous one.

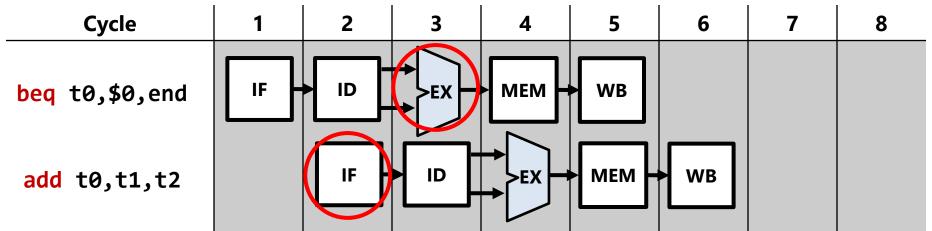


• sub must wait until add's WB phase is over before doing its ID phase

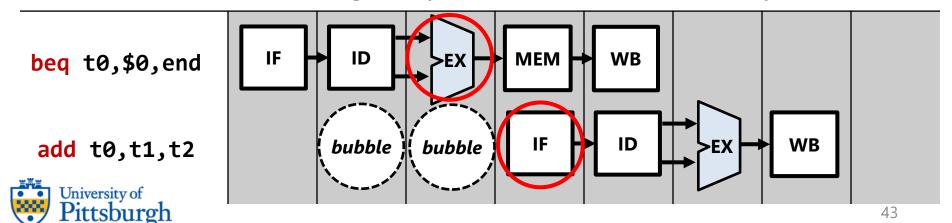


### **Control Hazards**

You don't know the outcome of a conditional branch.



• add must wait until beq's EX phase is over before its IF phase



## Dealing with Hazards

- Pipeline must be controlled so that hazards don't cause malfunction
- Who is in charge of that? You have a choice.
  - 1. Compiler can avoid hazards by inserting nops
    - Insert a nop where compiler thinks a hazard would happen
  - 2. CPU can internally avoid hazards using a *hazard detection unit* 
    - If structural/data hazard, pipeline stalled until resolved
    - If control hazard, pipeline *flushed* of wrong path instructions



## Compiler avoiding a data hazard

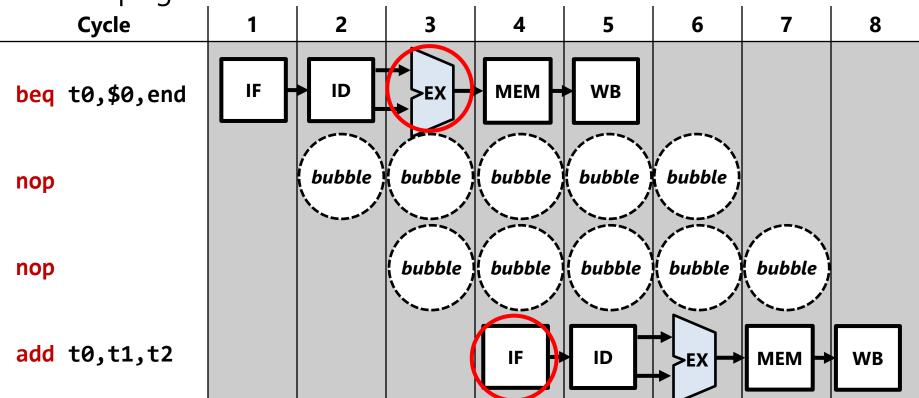
University of

 The nops flow through the pipeline not doing any work Cycle 8 add(t0,)t1,t2 IF ID MEM WB >EX bubble bubble bubble bubble bubble nop bubble bubble bubble bubble nop nop bubble bubble bubble ) bubble bubble IF ID >EX **MEM** sub s0, t0, t1

45

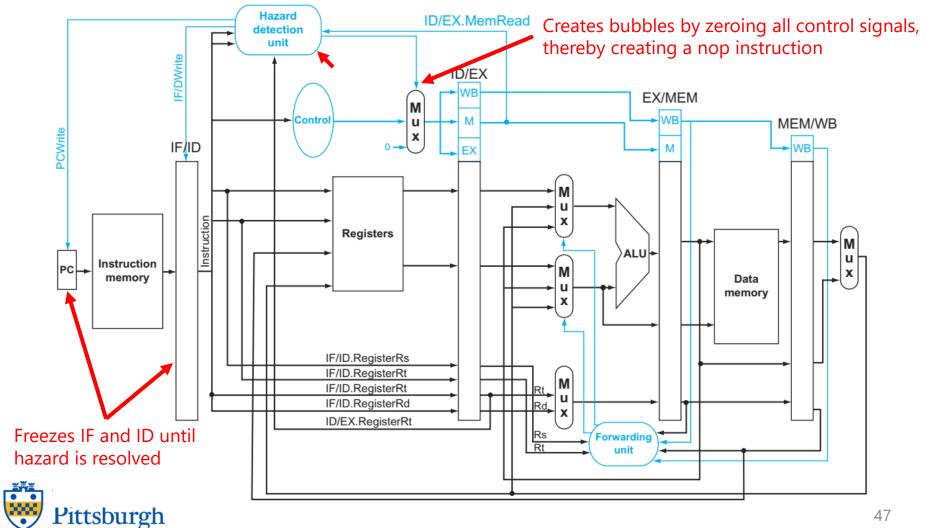
## Compiler avoiding a control hazard

The nops give time for condition to resolve before instruction fetch



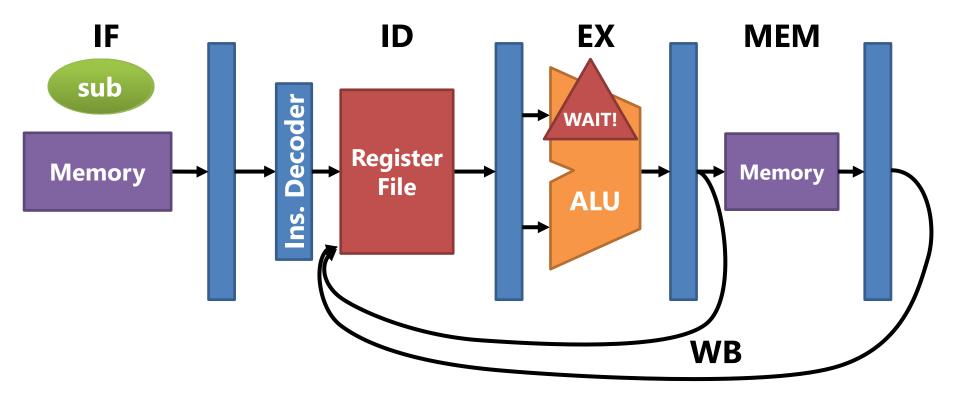


### Hazard Detection Unit



## Hazard Detection Unit avoiding a data hazard

• Suppose we have an **add** that depends on an **lw**.





### Structural / Data Hazards cause stalls

- If HDU detects a structural or data hazard, it does the following:
  - It stops fetching instructions (doesn't update the PC).
  - o It stops clocking the pipeline registers for the stalled stages.
  - The stages after the stalled instructions are filled with nops.
    - Change control signals to 0 using the mux!
  - o In this way, all following instructions will be stalled
- When structural or data hazard is resolved
  - HDU resumes instruction fetching and clocking of stalled stages
- But what about control hazards?
  - Instructions in wrong path are already in pipeline!
  - Need to *flush* these instructions



## Control Hazard Example

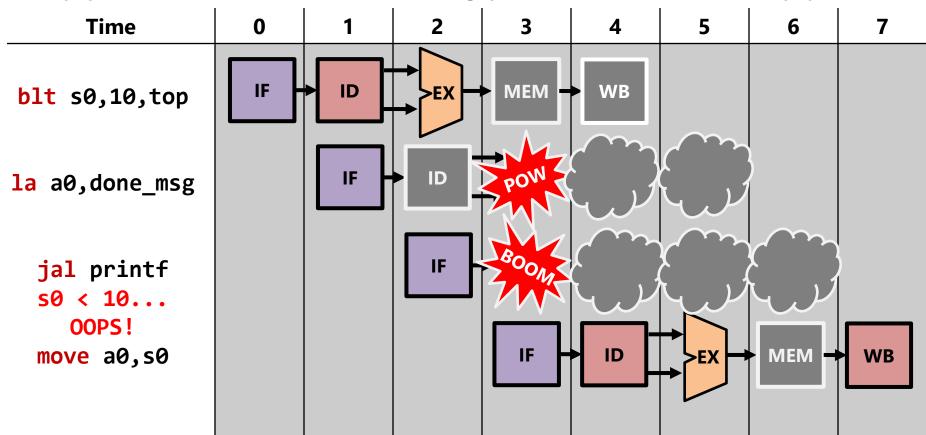
• Supposed we had this for loop followed by printf("done"):

```
for(s0 = 0 .. 10)
                                li
                                      s0, 0
    print(s0);
                            top:
                                move a0, s0
printf("done");
                                jal
                                      print
                                addi s0, s0, 1
                                blt s0, 10, top
  By the time s0, 10
  are compared at blt
  EX stage, the CPU
                                      a0, done msg
                                la
  would have already
                                      printf
                                jal
  fetched la and jal!
```



### What's a flush?

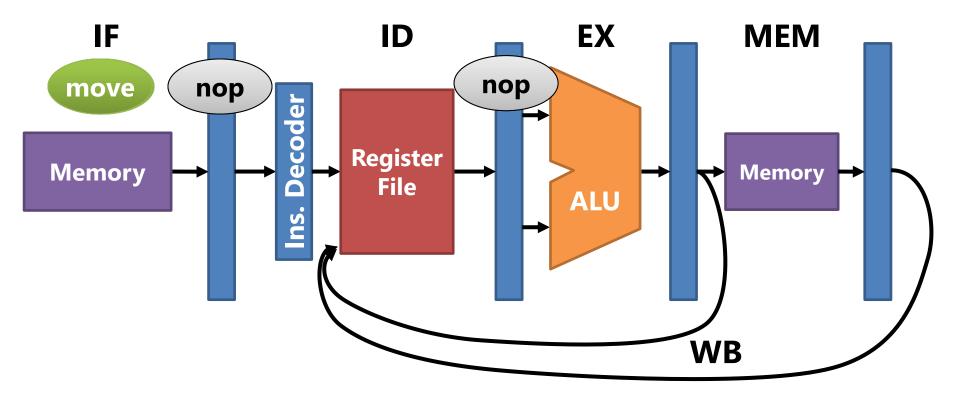
• A pipeline flush removes all wrong path instructions from pipeline





### Hazard Detection Unit avoiding a control hazard

• Let's watch the previous example.





### Control Hazards cause flushes

- If a control hazard is detected due to a branch instruction:
  - Any "newer" instructions (those already in the pipeline) are transformed into nops.
  - Any "older" instructions (those that came BEFORE the branch) are left alone to finish executing as normal.



# Performance penalty of pipeline stalls

• Remember the three components of performance:

$$\frac{\text{instructions}}{\text{program}}$$
 X  $\frac{\text{cycles}}{\text{instructions}}$  X  $\frac{\text{seconds}}{\text{cycle}}$ 

| Architecture           | Instructions | СРІ | Cycle Time (1/F) |
|------------------------|--------------|-----|------------------|
| Single-cycle           | Same         | 1   | 5 ns             |
| Ideal 5-stage pipeline | Same         | 1   | 1 ns             |
| Pipeline w/ stalls     | Same         | 2   | 1 ns             |

- Pipelining increases *clock frequency* proportionate to depth
- But stalls increase **CPI** (cycles per instruction)
  - $\circ$  If stalls prevent new instructions from being fetched half the time, the CPU will have a CPI of 2  $\rightarrow$  Only 2.5X speed up (instead of 5X)
- We'd like to avoid this penalty if possible!



## Compiler nops vs. CPU Hazard Detection Unit

- Limitations of compiler nops
  - Compiler must make assumptions about processor design
    - But processor design is not part of ISA
    - What if processor pipeline design changes in next CPU?
  - Length of MEM stage is very hard to predict by the compiler
    - Until now we assumed MEM takes a uniform one cycle
    - But remember what we said about the Memory Wall?
    - MEM isn't uniform really and sometimes hundreds of cycles
- But compiler nops is very energy-efficient
  - Hazard Detection Unit can be power hungry
    - A lot of long wires controlling remote parts of the CPU
    - Adds to the Power Wall problem
  - Compiler scheduling via nops removes need for HDU

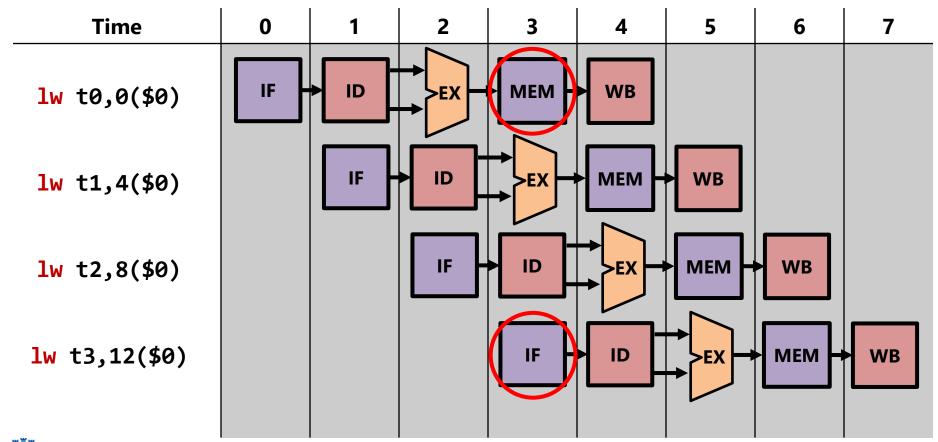


# Solving Structural Hazards



### Structural Hazard on Memory

Two instructions need to use the same hardware at the same time.





### What could we do??

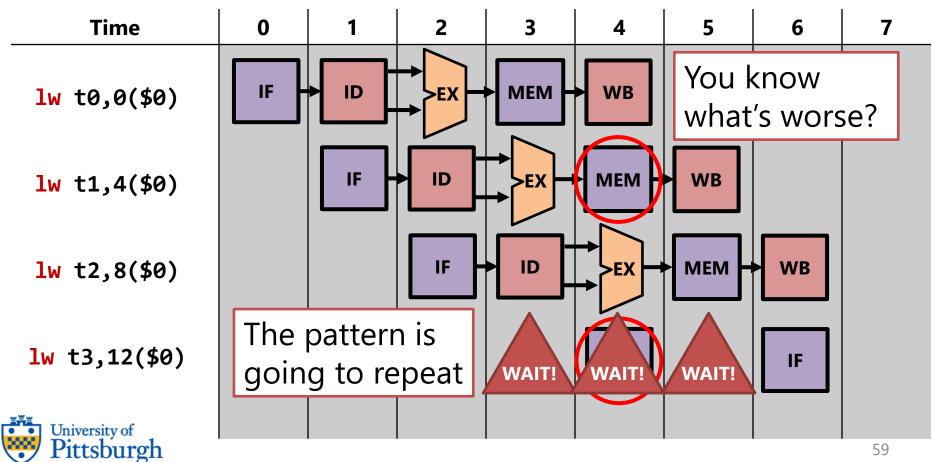
Two people need to use **one** sink at the same time
Well, in this case, it's memory but same idea





### We can do something similar!

• One option is to **wait** (a.k.a. **stall**).



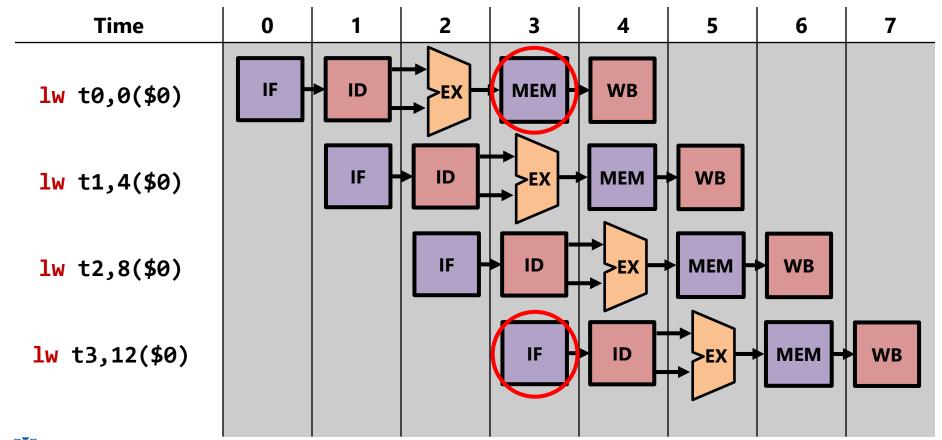
### Or we could throw in more hardware!

- For less commonly used CPU resources, stalling can work fine
- But memory (and some other things) is used **CONSTANTLY**
- How do the bathrooms solve this problem?
  - o Throw in lots of sinks!
  - o In other words, throw more hardware at the problem!
- Memory's a resource with a lot of *contention* 
  - So have two memories, one for instructions, and one for data!
  - Not literally but CPUs have separate instruction and data caches



### Structural Hazard removed with two Memories

With separate i-cache and d-cache, MEM and IF can work in parallel

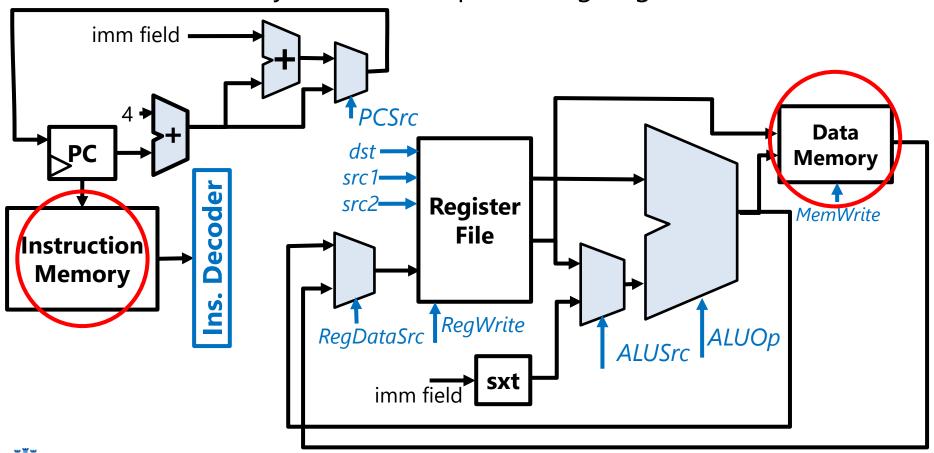




### Structural Hazard removed with two Memories

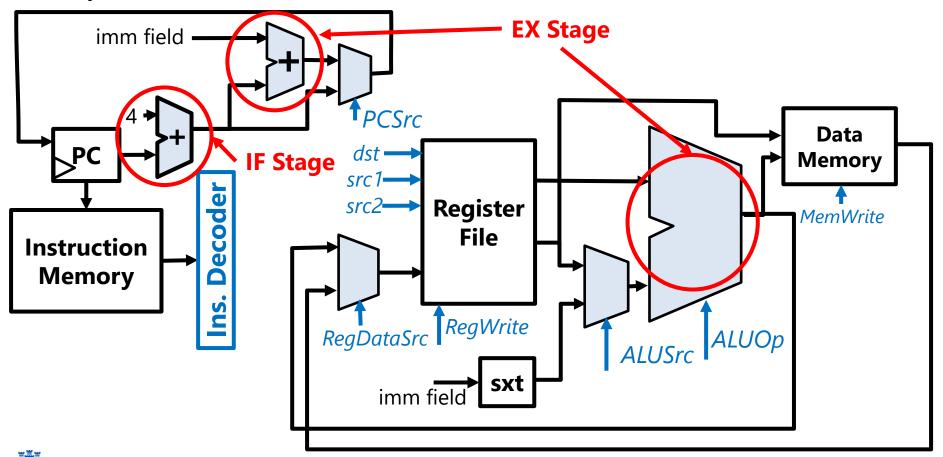
• But is that the only hardware duplication going on here?

University of



### Structural Hazards removed with Multiple Adders

• Why do we need 3 adders? To avoid stalls due to contention on ALU!



University of **Pittsburgh** 

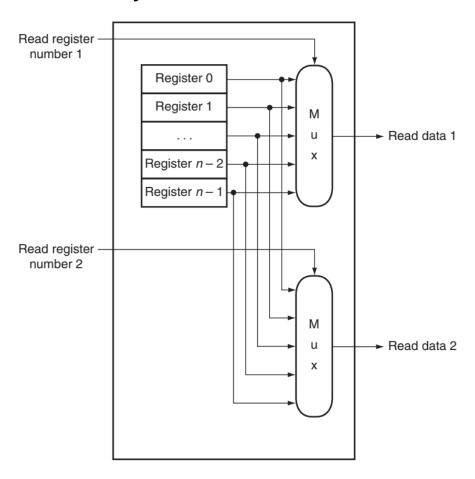
# Solving Structural Hazards

- There are mainly two ways to throw more hardware at the problem
- 1. Duplicate contentious resource
  - One memory cannot sustain MEM + IF stage at same cycle
    - → Duplicate into one instruction memory, one data memory
  - One ALU cannot sustain IF + EX stage at same cycle
    - → Duplicate into one ALU and two simple adders
- 2. Add ports to a single shared (memory) resource
  - o **Port**: Circuitry that allows either read or write access to memory
  - o If current number of ports cannot sustain rate of access per cycle
    - → Add more ports to memory structure for simultaneous access



## Two Register Read Ports

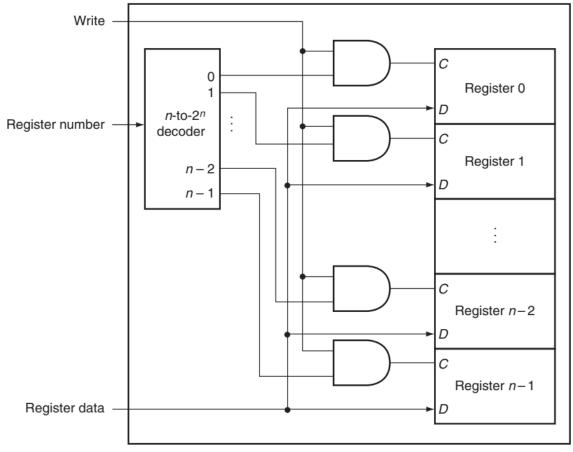
• By adding more MUXes, you can add even more read ports





## One Register Write Port

• By adding more decoders, you can add more write ports





### But who would need more register ports?

- With two read ports and one write port
  - Enough to sustain one ID and one WB stage per cycle
  - Enough to sustain CPI = 1 (or in other words IPC = 1)
- But what if we want an IPC > 1?
  - More than one instruction per cycle! (a.k.a superscalar processor)
  - Must sustain more than one ID / WB stage per cycle
  - Need more register read ports and write ports!
  - Not only registers, memory would need more ports too!
  - o Like everything else, this consumes lots of power
- We'll talk more about this when we discuss superscalars

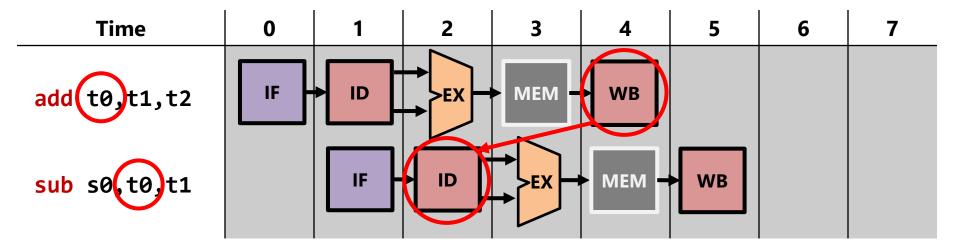


# Solving Data Hazards



#### Data Hazards

• An instruction depends on the output of a previous one.

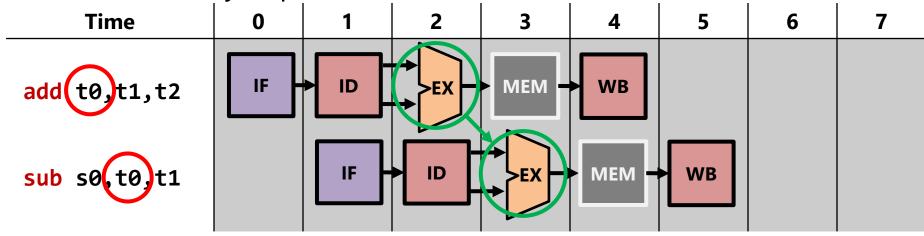


- When does add finish computing its sum?
- Well then... why not just use the sum when we need it?



## Solution 1: Data Forwarding

- Since we've pipelined control signals, we can check if instructions in the pipeline depend on each other (see if registers match).
- If we detect any dependencies, we can *forward* the needed data.

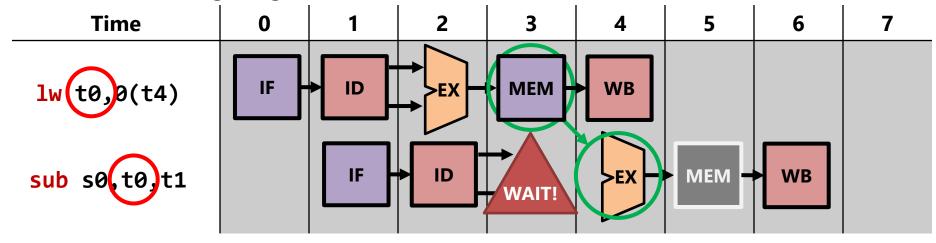


- This handles one kind of data forwarding...
- Where else can data come from and be written into registers?
  - Memory!



## Data Forwarding from Memory

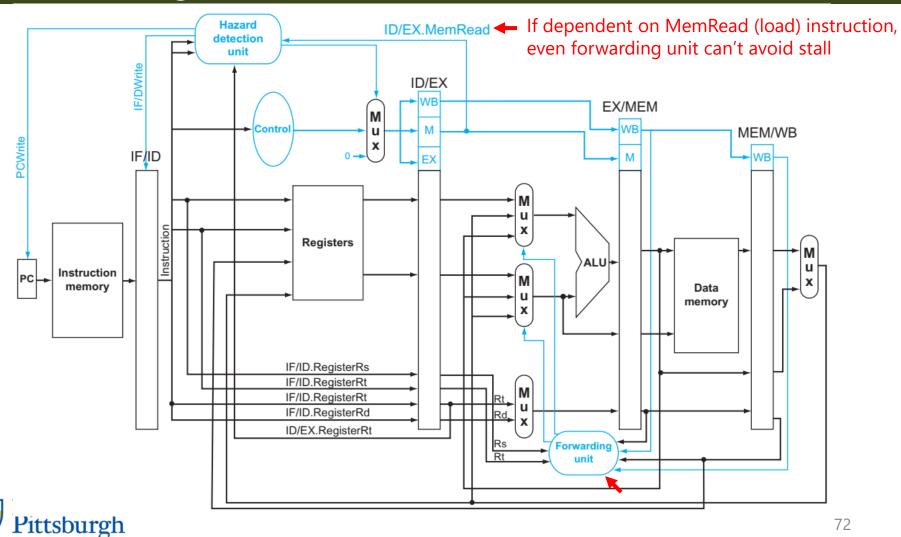
- Well memory accesses happen a cycle later...
- What are we going to have to do?



This kind of stall is unavoidable in our current pipeline



## Forwarding Unit and Use-after-load-hazard



### Forwarding Unit

- Just like the HDU, the Forwarding Unit is power hungry
- Number of forwarding wires  $\propto$  (pipeline stages)<sup>2</sup>
  - Why the quadratic relationship?
  - o Per pipeline stage, N stages after it from which data is forwarded
    - In previous picture, see number of inputs to MUX before ALU!
  - o And there are N stages to which data must be forwarded
    - In previous picture, only one EX stage is shown, but if there are multiple stages, need MUXes in all those stages
- Deep pipelining has diminishing returns on power investment
  - Cycle time improves by a factor of N
  - Power consumption increases by a factor of N<sup>2</sup> (or more)
  - Not the only problem with deep pipelining that we will see



# Solution 2: Avoid stalls by reordering

- Let's say the following is your morning routine (2 hours total)
  - Have laundry running in washing machine (30 minutes)
  - Have laundry running in dryer (30 minutes)
  - 3. Have some tea boiling in the pot (30 minutes)
  - 4. Drink tea (30 minutes)
- Can you make this shorter? Yes! (1 hour total)
  - 1. Have washing machine running and 3. Tea boiling (30 minutes)
  - 2. Have dryer running and 4. Drink tea (30 minutes)
- How? By simply by **reordering** our actions
  - $\circ$  Steps 1  $\rightarrow$  2 and 3  $\rightarrow$  4 have data dependencies
  - Other steps can be freely reordered with each other

















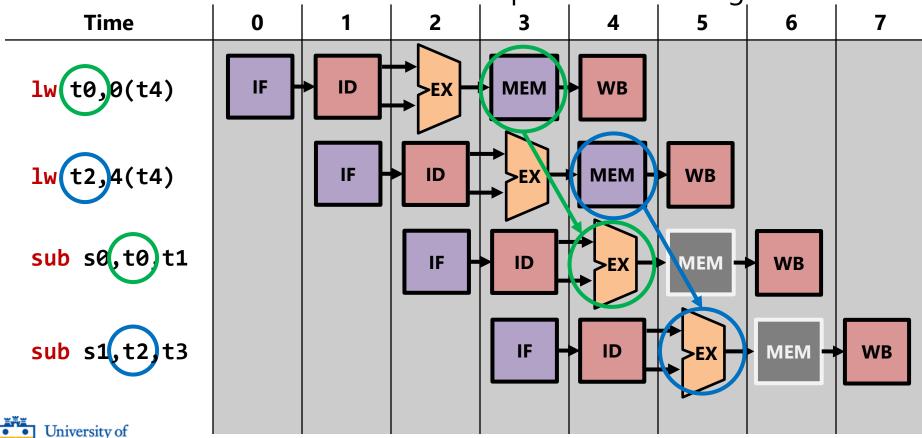
### Data Hazard removed through Compiler Reordering

 If the compiler has knowledge of how the pipeline works, it can reorder instructions to let loads complete before using their data.

| redider instructions to let loads complete belore using their data. |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
| Time  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| lw(t0,0(t4)   |   |   |   |   |   |   |   |   |
| sub s0, t0) t1  |   |   |   |   |   |   |   |   |
| lw(t2,4(t4)   |   |   |   |   |   |   |   |   |
| sub s1,t2)t3  |   |   |   |   |   |   |   |   |
| University of   |   |   |   |   |   |   |   |   |

### Data Hazard removed through Compiler Reordering

• If the **compiler** has knowledge of how the pipeline works, it can **reorder** instructions to let loads complete before using their data.



### Limits of Static Scheduling

- Reordering done by the compiler is called *static scheduling*
- Static scheduling is a powerful tool but is in some ways limited
  - Again, compiler must make assumptions about pipeline
  - Length of MEM stage is very hard to predict by the compiler
    - Remember the Memory Wall?
  - Data dependencies are hard to figure out by a compiler
    - When data is in registers, trivial to figure out
    - When data is in memory locations, more difficult. Given:

```
1w(t0), 0(t4)
```

**lw** t2,4(t4) But what if 8(t0) and 4(t4) are the same addresses?

This involves pointer analysis, a notoriously difficult analysis!



# Dynamic scheduling is another option

- **Dynamic scheduling** is scheduling done by the CPU
- It doesn't have the limitations of static scheduling
  - It doesn't have to predict memory latency
    - It can adapt as things unfold
  - o It's easy to figure out data dependencies, even memory ones
    - At runtime, addresses of 8(t0) and 4(t4) are easily calculated
- But at runtime it uses lots of power for the data analysis
  - o ... which again causes problems with the **Power Wall**
  - But more on this later



# Solving Control Hazards



### Loops

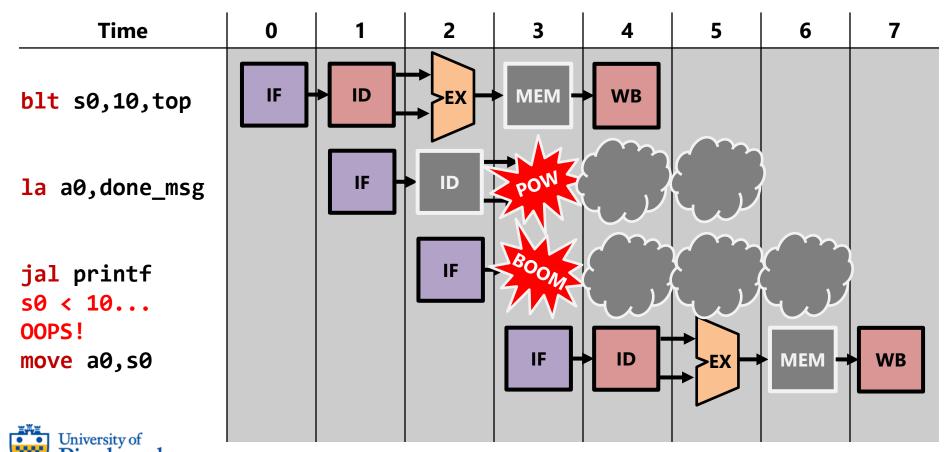
• Loops happen all the time in programs.

```
for(s0 = 0 .. 10)
                                       50, 0
                                 li
    print(s0);
                             top:
                                 move a0, s0
printf("done");
                                 jal
                                       print
                                 addi s0, s0, 1
 How often does this
                                 blt s0, 10, top
 blt instruction go to
top? How often does
                                       a0, done msg
                                 la
it go to the following
                                       printf
                                 jal
   la instruction?
```



### Pipeline Flushes at Every Loop Iteration

• The pipeline must be **flushed** every time the code loops back!



### Performance Impact from Control Hazards

- **Frequency** of flushes ∝ frequency of branches
  - If we have a tight loop, branches happen every few instructions
  - Typically, branches account for 15~20% of all instructions
- **Penalty** from one flush ∝ depth of pipeline
  - Number of flushed instructions == distance from IF to MEM
  - What if there are 4 ID stages and 3 EX stages? Penalty == 7!
  - Current architectures can have more than 20 stages!
- May spend more time just flushing instructions than doing work!



### Performance Impact from Control Hazards

- CPI = CPI<sub>nch</sub> +  $\alpha * \pi * K$ 
  - CPI<sub>nch</sub>: CPI with no control hazard
  - $\circ \alpha$ : fraction of branch instructions in the instruction mix
  - $\circ$   $\pi$ : probability a branch is actually taken
  - K: penalty per pipeline flush

**Example:** If 20% of instructions are branches and the probability that a branch is taken is 50%, and pipeline flush penalty 7 cycles, then:

$$CPI = CPI_{nch} + 0.2 * 0.5 * 7 = CPI_{nch} + 0.7$$
 cycles per instruction

• Is there a way to minimize the performance impact?

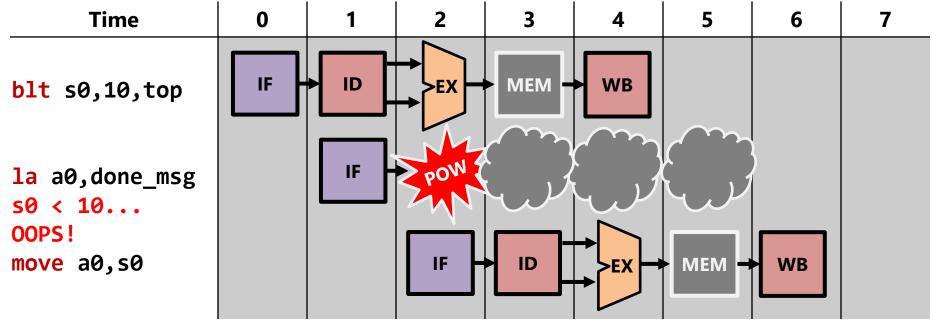


# Solution 1: MORE SINKS! (a.k.a. hardware)



### Do we reeeally need to compare at EX stage?

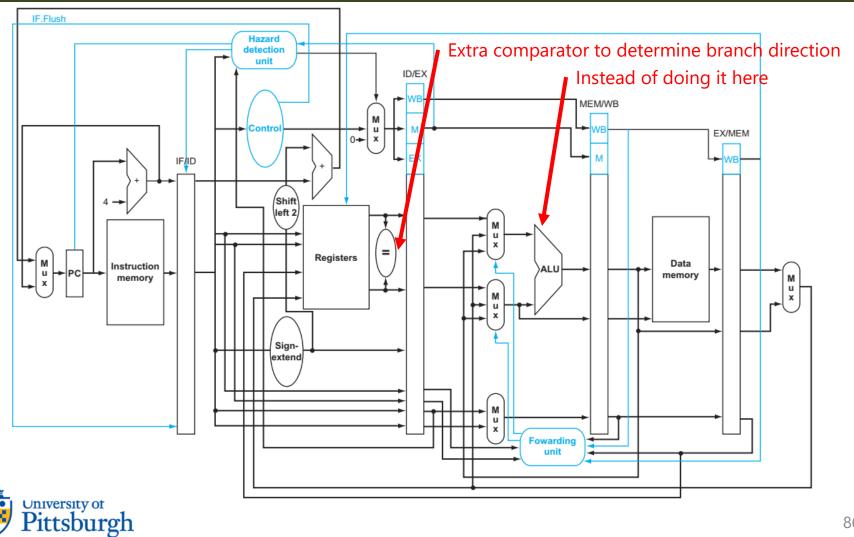
What if branch comparison was done at the ID stage, not EX stage?



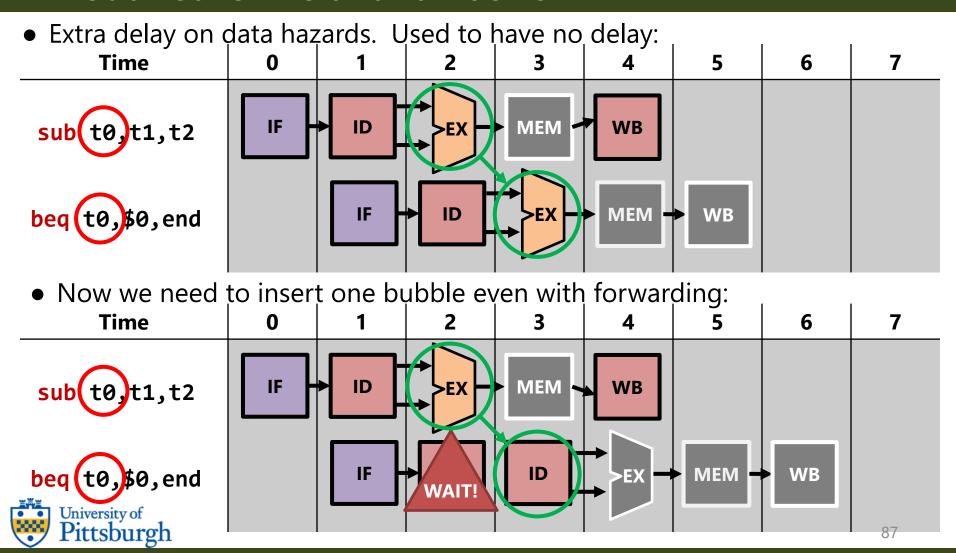
- Reduced penalty from 2 cycles → 1 cycle!
- But of course that means we need a comparator at the ID stage



# Solution 1: MORE SINKS! (a.k.a. hardware)

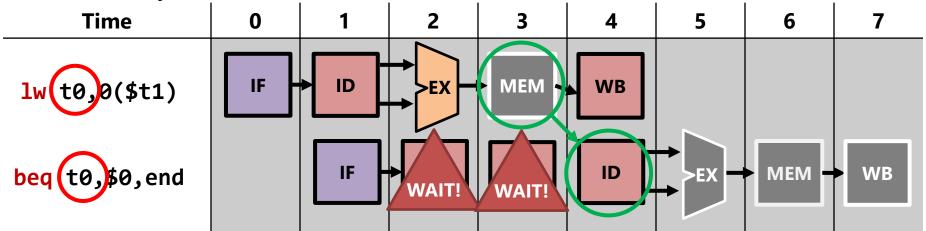


#### Not all sunshine and rainbows



#### Not all sunshine and rainbows

• Extra delay on data forwarded from **lw** also:



- Now we must insert two bubbles instead of one!
- Not to mention we must now add more forwarding paths:
  - From  $\mathbf{EX} \rightarrow \mathbf{ID}$
  - From MEM → ID
- Doing things in more stages means more forwarding paths.



# Branch Prediction



### Solution 2: Branch Prediction

- Comparator at ID stage is not satisfactory
- What if ...
  - We were able to **predict** the branch outcome?
  - o But were able to do that without reading register values?



- What would that get us?
  - 1. No extra data hazard bubble due to **EX**  $\rightarrow$  **ID** forwarding
    - We are not reading register values, remember?
  - 2. No extra data forwarding wires
  - 3. While still removing bubbles from control hazards!



# Types of Branch Prediction

#### • Static Branch Prediction

- Predicting branch behavior based on code analysis
- Done by the compiler that hints at next fetch target through ISA
- Not used nowadays due to inaccuracy of compiler predictions

#### • Dynamic Branch Prediction

- Predicting branch behavior during program execution
- o Typically using hardware that tracks history information
- o Premise: history repeats itself



### Dynamic Branch Prediction

- We have been doing a form of branch prediction all along!
  - We assumed that all branches will be not taken
- Two simple policies:
  - Predict not taken: continue fetching PC + 4, flush if taken Pros: Can start fetching the next instruction immediately
  - Predict taken: fetch branch target as soon as ID, flush if not taken Pros: 67% of branches are taken, on average (due to loops)
- What if we use past history as a guide?
  - Branches not taken in the past are likely not taken in the future (e.g. branches to error handling code)
  - Branches taken in the past are likely taken in the future (e.g. branch back to the top of the loop)

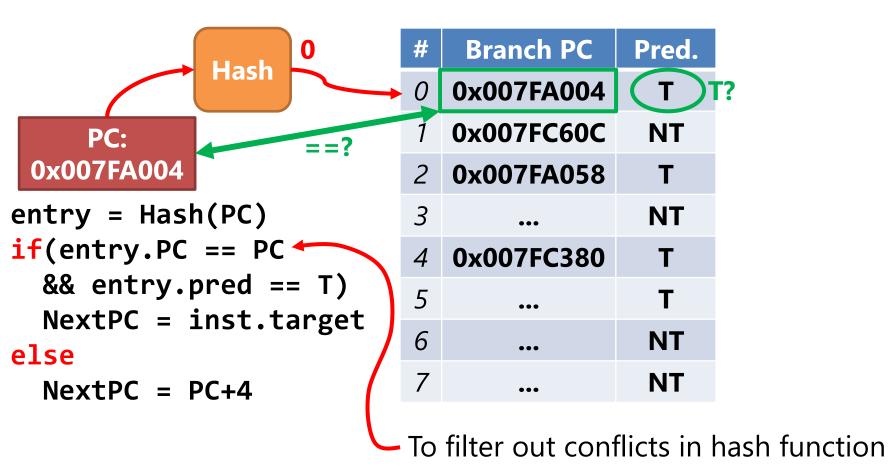


# The Branch History Table (BHT)

- BHT stores Taken (**T**) or Not Take (**NT**) history info for each branch
  - If branch was taken most recently, T is recorded
  - o If branch was not taken most recently, NT is recorded
- BHT is indexed using PC (Program Counter)
  - Each branch has a unique PC, so a unique entry in the BHT
- BHT, being hardware, is limited in capacity
  - Cannot have a huge table with all PCs possible in a program
  - o Besides not every PC address contains a branch
  - Best to use hash table to map branch PCs to (limited) entries



### The Branch History Table (BHT)





### Limitations of Branch History Table (BHT)

- Ideally, we would like a prediction at the IF stage
  - So that correct instruction is immediately fetched in next cycle
  - But this is hard to do with only a BHT
- It's possible to have the BHT at the IF stage
  - All the information needed is the PC (which is available at IF)
- But must still wait until ID to decode branch target, if branch taken
  - Even if BHT gives us a T or NT prediction at the IF stage
  - If NT: no need to wait (branch target is irrelevant)
     But if T: need to wait until target decode to jump to target
- That introduces a bubble for taken branches

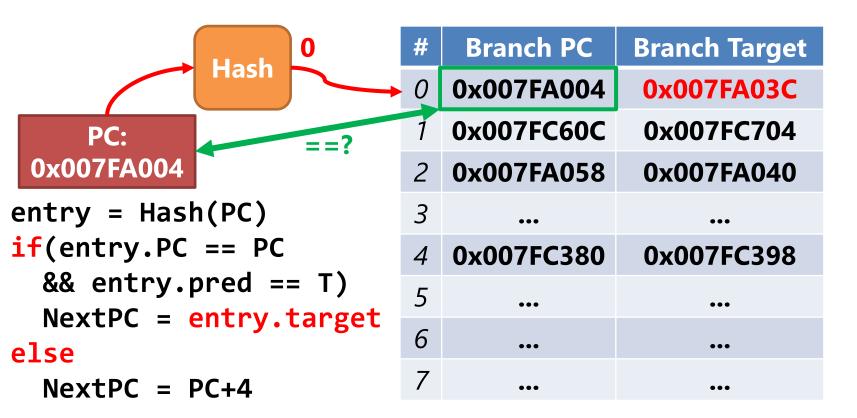


### The Branch Target Buffer (BTB)

- BTB stores branch target for each branch
- BTB is also indexed using PC of branch using a hash table
- BTB allows full prediction to happen on the IF stage
  - No need to wait until ID stage for branch target to be decoded

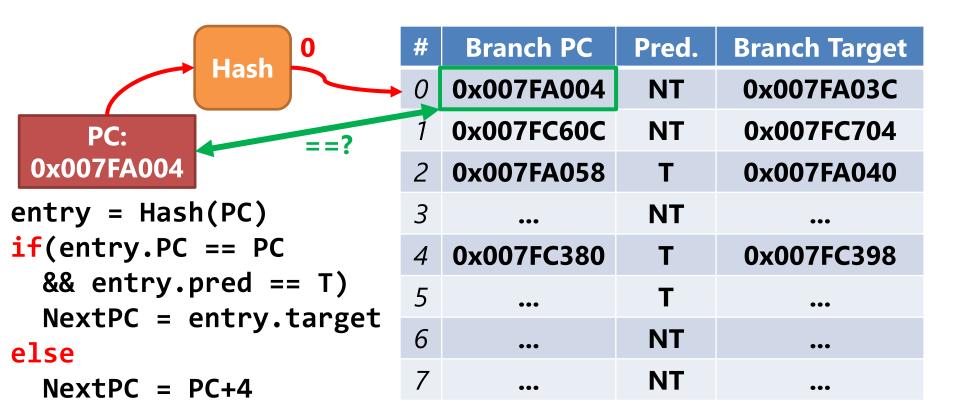


## The Branch Target Buffer (BTB)





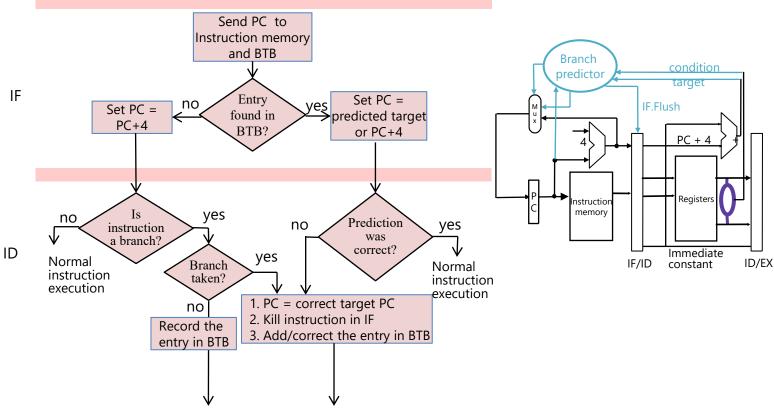
#### **Combined Branch Predictor**





#### **Branch Prediction Decision Tree**

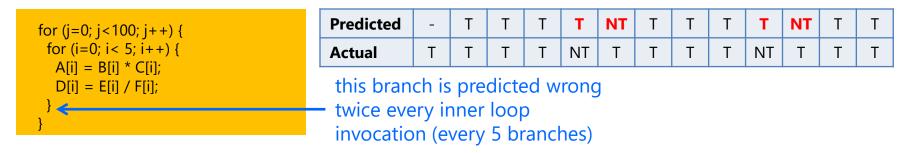
Assuming that branch condition and target are resolved in ID stage





#### Limitations of 1-bit BHT Predictor

- Is 1-bit (T / NT) enough history to make a good decision?
- Take a look at this example:

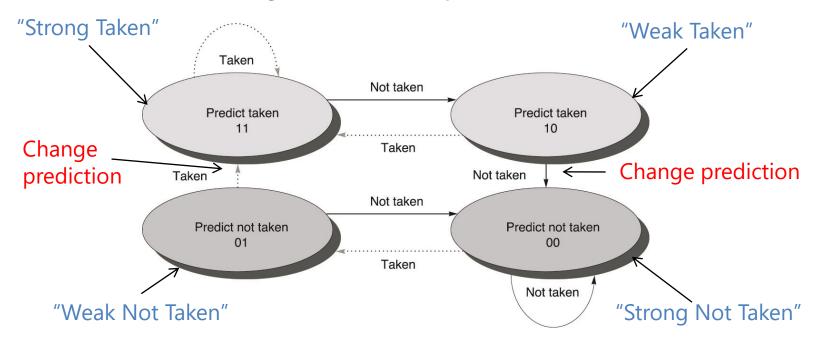


- It would have been better to stay with T than flip back and forth!
- Idea behind the 2-bit predictor: create some hysteresis
  - So that predictions don't flip immediately



### 2-bit BHT Predictor

State transition diagram of 2-bit predictor:

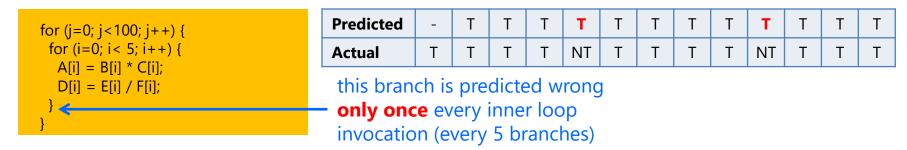


Requires two consecutive mis-predictions to flip direction!



#### 2-bit BHT Predictor

- How well does the 2-bit predictor do with our previous example?
- Our previous example:

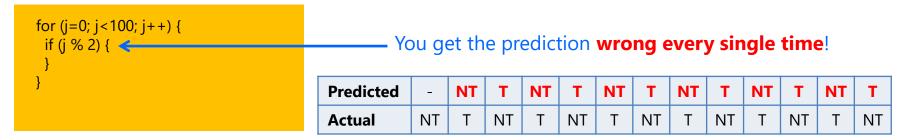


- Does it help beyond 2 bits? (e.g. 3-bit predictor, or 4-bit predictor)
  - o Empirically, no. 2 bits already cover loop which is most common.
  - 2 bits + large BHT gets you ~93% accuracy
- We need other tricks to improve accuracy!



# Correlating Predictors

Sometimes you need to know more than the PC of your branch



- o For a 1-bit predictor, but a 2-bit predictor doesn't do well either
- Should base the prediction also on the history of that branch!
- This is called local branch history (since it's the same branch)
- Knowing the result of other branches in your history also helps

```
If (j == 0) {

...

If (j!= 0) {

Knowing result of a previous different branch in your history helps in predicting this branch!
```

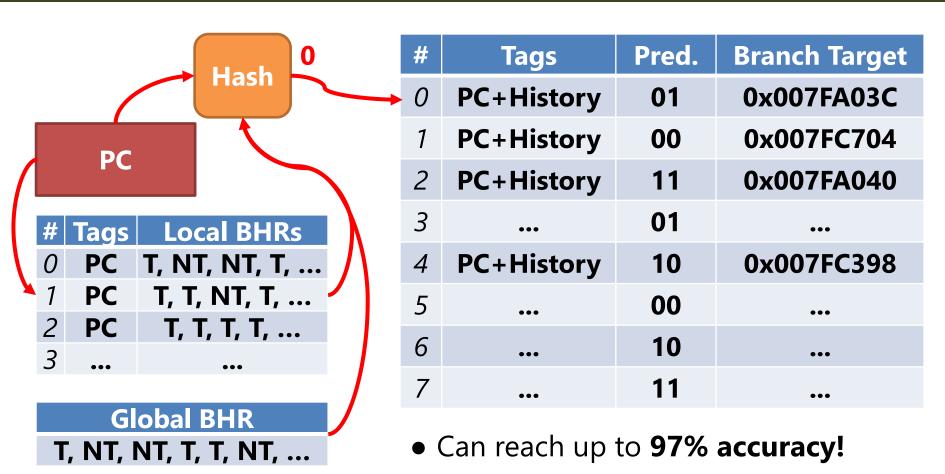
This is called **global branch history** (since it's a different branch)

# Correlating Predictors

- Idea: have multiple entries per branch depending on history
  - Local branch history + Global branch history
  - An entry with matching history gives more precise prediction!
- Now, instead of indexing into BHT by branch PC only
  - Use hash of PC + Local branch history + Global branch history
- History is stored in register called Branch History Shift Register (BHR)
  - o T/NT bit is shifted on to BHR whenever branch is encountered
  - One Global BHR (there is just one global history)
  - Multiple Local BHRs (local histories for each branch)



# Correlating Predictors





### How about jr \$ra?

- jr \$ra: Jump return to address stored in \$ra
  - When a function is called, the caller stores return address to \$ra
     (jal funcAddr stores PC of next instruction to \$ra)
  - When a function returns, jr \$ra jumps to return address in \$ra
- Why is this a problem?
  - Unlike other branches, branch target is not an immediate value!
     (This is called an indirect branch)
  - Target can change for same jr depending on who caller is
  - Makes life difficult for BTB which relies on target being constant
- Target of **jr** is predicted using the **Return Stack Buffer** 
  - Not the Branch Target Buffer (BTB)



#### The Return Stack Buffer

 Since functions return to where they were called every time, it makes sense to cache the return addresses (in a stack)

When we encounter 40CC00 4AB33C jal someFunc the jal, push the 46280C 4AB340 beq v0, \$0, blah return address. 4AB108 When we encounter 4AB340 the jr \$ra, pop the someFunc: return address. Easy! 000000 000000 jr \$ra 000000 On misprediction or stack overflow, empty stack 000000

Not a problem since this is for prediction anyway

