

# VLIW Processors

CS/COE 1541 (Fall 2020)

Wonsun Ahn

# Limits on Deep Pipelining

- Ideally,  $\text{CycleTime}_{\text{Pipelined}} = \text{CycleTime}_{\text{SingleCycle}} / \text{Number of Stages}$ 
  - In theory, can indefinitely improve performance with more stages
- Limitation 1: **Cycle time** does not improve indefinitely
  - With shorter stages, delay due to latches become significant
  - With many stages, hard to keep stage lengths perfectly balanced
  - Manufacturing variability exacerbates the stage length unbalance
- Limitation 2: **CPI** tends to increase with deep pipelines
  - Penalty due to branch misprediction increases
  - Stalls due to data hazards cause more bubbles
- Limitation 3: **Power consumption** increases with deep pipelines
  - Wires for data forwarding increase quadratically with depth
- Is there another way to improve performance?

# What if we improve CPI?

- Remember the three components of performance?

$$\frac{\text{instructions}}{\text{program}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{cycle}}$$

- Pipelining focused on seconds / cycle, or cycle time
- Can we improve cycles / instruction, or CPI?
  - But the best we can get is CPI = 1, right?
  - How can an instruction be executed in less than a cycle?

# Wide Issue Processors

---

# From CPI to IPC

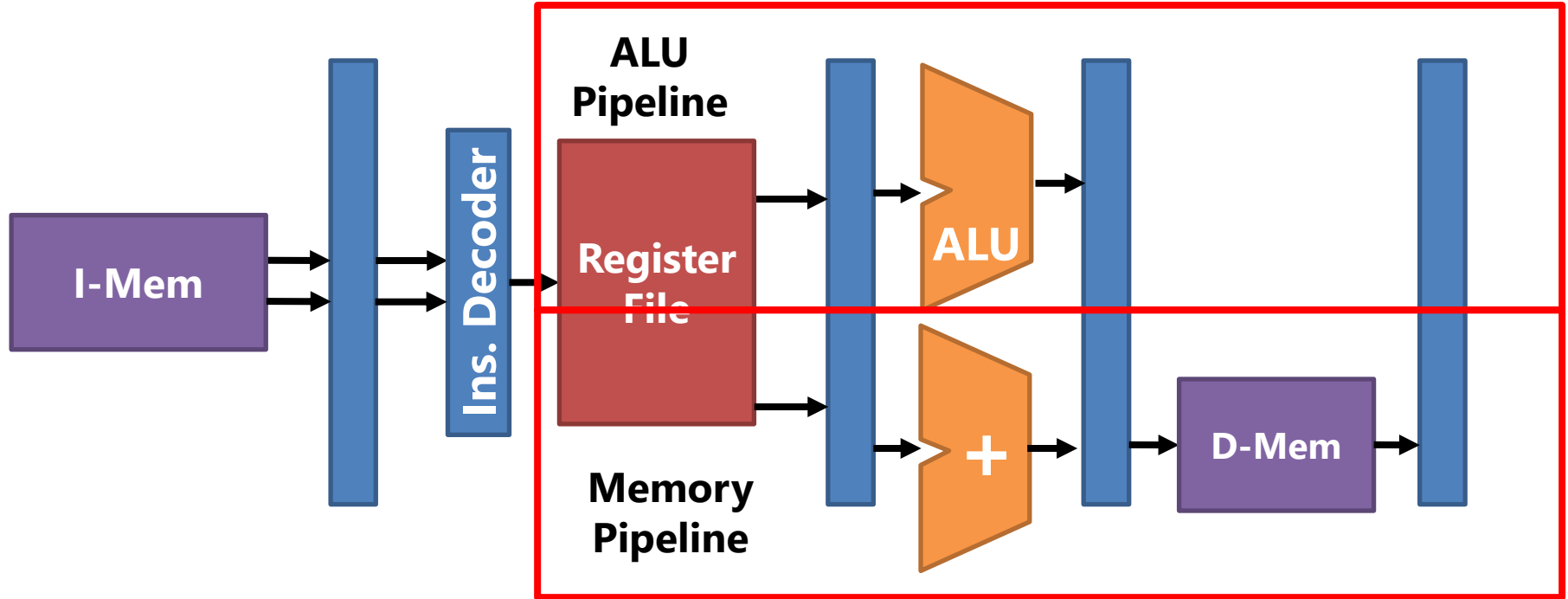
- How about if we **fetch two** instructions each cycle?
  - Maybe, fetch one ALU instruction and one load/store instruction

| Instruction type          | Pipe stages |    |    |     |     |     |     |    |
|---------------------------|-------------|----|----|-----|-----|-----|-----|----|
| ALU or branch instruction | IF          | ID | EX | MEM | WB  |     |     |    |
| Load or store instruction | IF          | ID | EX | MEM | WB  |     |     |    |
| ALU or branch instruction |             | IF | ID | EX  | MEM | WB  |     |    |
| Load or store instruction |             | IF | ID | EX  | MEM | WB  |     |    |
| ALU or branch instruction |             |    | IF | ID  | EX  | MEM | WB  |    |
| Load or store instruction |             |    | IF | ID  | EX  | MEM | WB  |    |
| ALU or branch instruction |             |    |    | IF  | ID  | EX  | MEM | WB |
| Load or store instruction |             |    |    | IF  | ID  | EX  | MEM | WB |

- Then, **IPC (Instructions per Cycle) = 2**
  - And by extension,  $CPI = 1 / IPC = 0.5$  !
- **Wide-issue** processors can execute multiple instructions per cycle

# Pipeline design for previous example

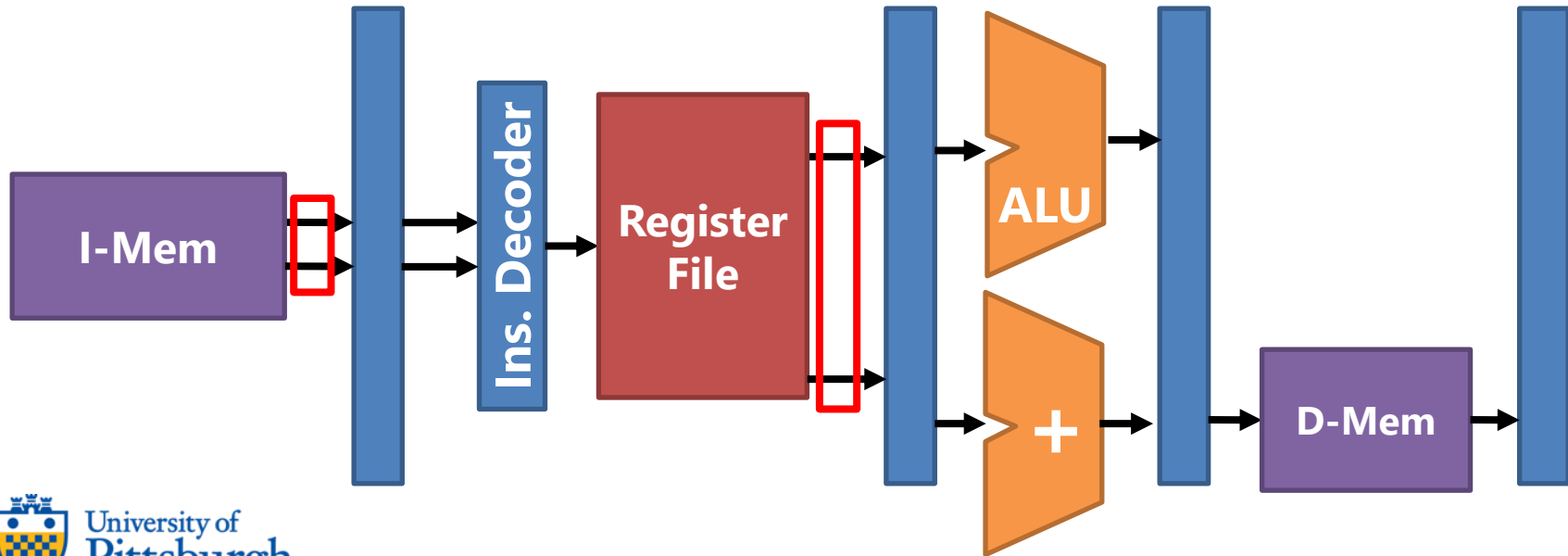
- One pipeline for ALU/Branches and one for loads and stores



- This introduces new structural hazards that we didn't have before!

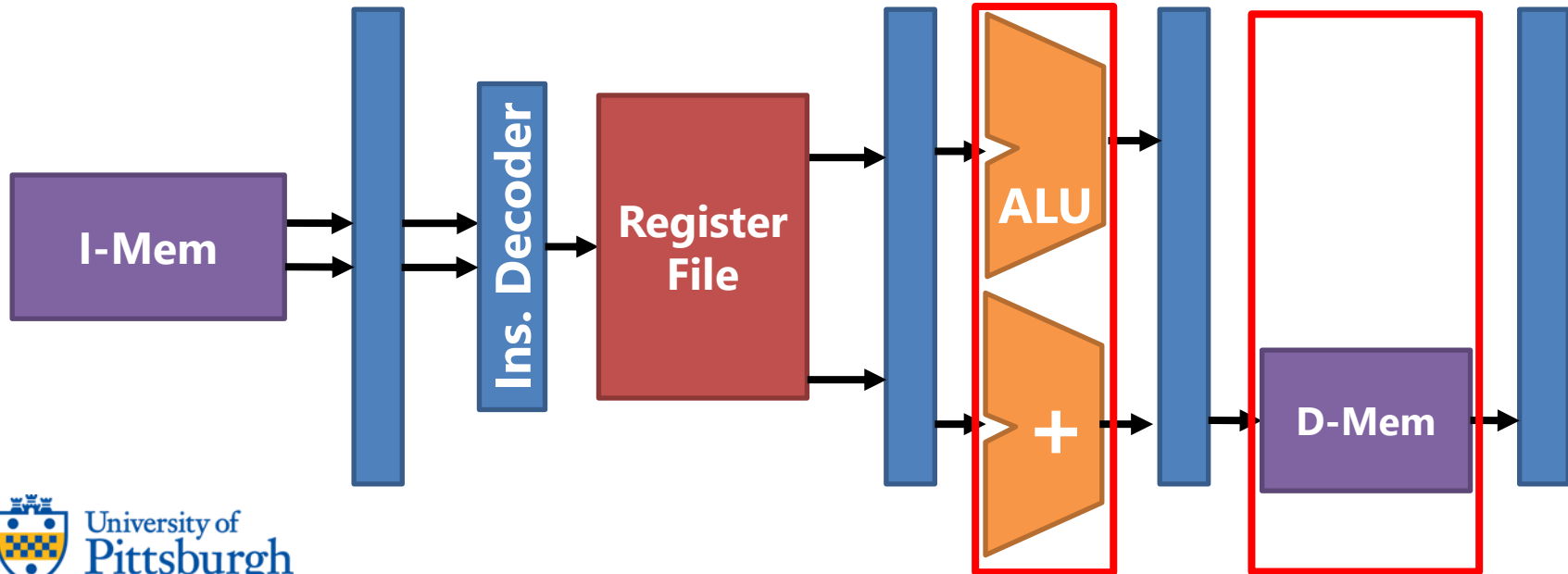
# Structural Hazard in Storage Locations

- Two instructions must be fetched from instruction memory  
→ Add extra read ports to the instruction memory
- Two ALUs must read from the register file at the same time  
→ Add extra read ports to the register file
- Two instructions must write to register file at WB stage (not shown)  
→ Add extra write ports to the register file



# Structural Hazard in Functional Units

- Structural hazard on EX units
  - Top ALU can handle all arithmetic (  $+$ ,  $-$ ,  $*$ ,  $/$  )
  - Bottom ALU can only handle  $+$ , needed for address calculation
- Structural hazard on MEM unit
  - Top ALU pipeline does not have a MEM unit to access memory





# Structural Hazard in Functional Units

- Code on the left will result in a timeline on the right
  - If it were not for the bubbles, we could have finished in 4 cycles!

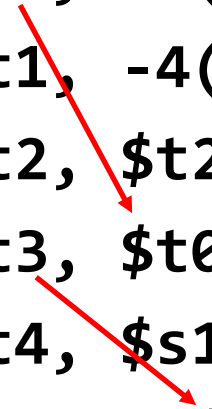
```
lw    $t0, 0($s1)
lw    $t1, -4($s1)
addi  $t2, $t2, -8
add   $t3, $t0, $s1
add   $t4, $s1, $s1
sw    $t5, 8($t3)
sw    $t6, 4($s1)
```

| CC | ALU Pipeline | Mem Pipeline |
|----|--------------|--------------|
| 1  |              | lw t0        |
| 2  | addi t2      | lw t1        |
| 3  | add t3       |              |
| 4  | add t4       | sw t5        |
| 5  |              | sw t6        |

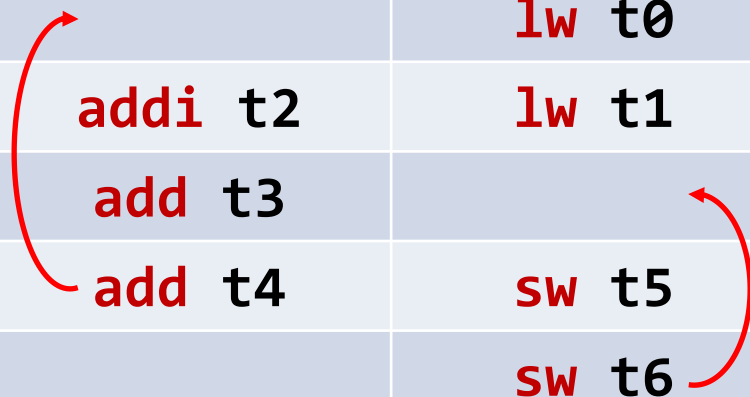
# Structural Hazard Solution: Reordering

- Of course we can come up with a better schedule
  - While still adhering to the data dependencies

**lw** \$t0, 0(\$s1)  
**lw** \$t1, -4(\$s1)  
**addi** \$t2, \$t2, -8  
**add** \$t3, \$t0, \$s1  
**add** \$t4, \$s1, \$s1  
**sw** \$t5, 8(\$t3)  
**sw** \$t6, 4(\$s1)



| CC | ALU Pipeline   | Mem Pipeline |
|----|----------------|--------------|
| 1  |                | <b>lw</b> t0 |
| 2  | <b>addi</b> t2 | <b>lw</b> t1 |
| 3  | <b>add</b> t3  |              |
| 4  | <b>add</b> t4  | <b>sw</b> t5 |
| 5  |                | <b>sw</b> t6 |



# Why not just duplicate all resources?

- Why not have two full ALUs, have MEM units at both pipelines?
  - That way, we can avoid those structural hazards in the first place
  - But that leads to **low utilization**
    - ALU/Branch type instructions will not use the MEM unit
    - Load/Store instructions will not need the full ALU
- Most processors have specialized pipelines for different instructions
  - Integer ALU pipeline, FP ALU pipeline, Load/Store pipeline, ...
  - With **scheduling**, can achieve **high utilization** *and* **performance**
- Who does the scheduling? Well, we talked about this already:
  - Static scheduling → Compiler
  - Dynamic scheduling → Processor

# VLIW vs. Superscalar

- There are two types of wide-issue processors
- If the compiler does **static scheduling**, the processor is called:
  - **VLIW (Very Long Instruction Word)** processor
  - This is what we will learn this chapter
- If the processor does **dynamic scheduling**, the processor is called:
  - **Superscalar** processor
  - This is what we will learn next chapter

# VLIW Processors

---

# VLIW Processor Overview

- What does **Very Long Instruction Word** mean anyway?
  - It means one instruction is *very* long!
  - Why? Because it contains multiple operations in one instruction
- A (64 bits long) VLIW instruction for our example architecture:

**ALU/Branch Operation (32 bits)**

**Load/Store Operation (32 bits)**

- An example instruction could be:

`addi $t2, $t0, -8`

`lw $t1, -4($s1)`

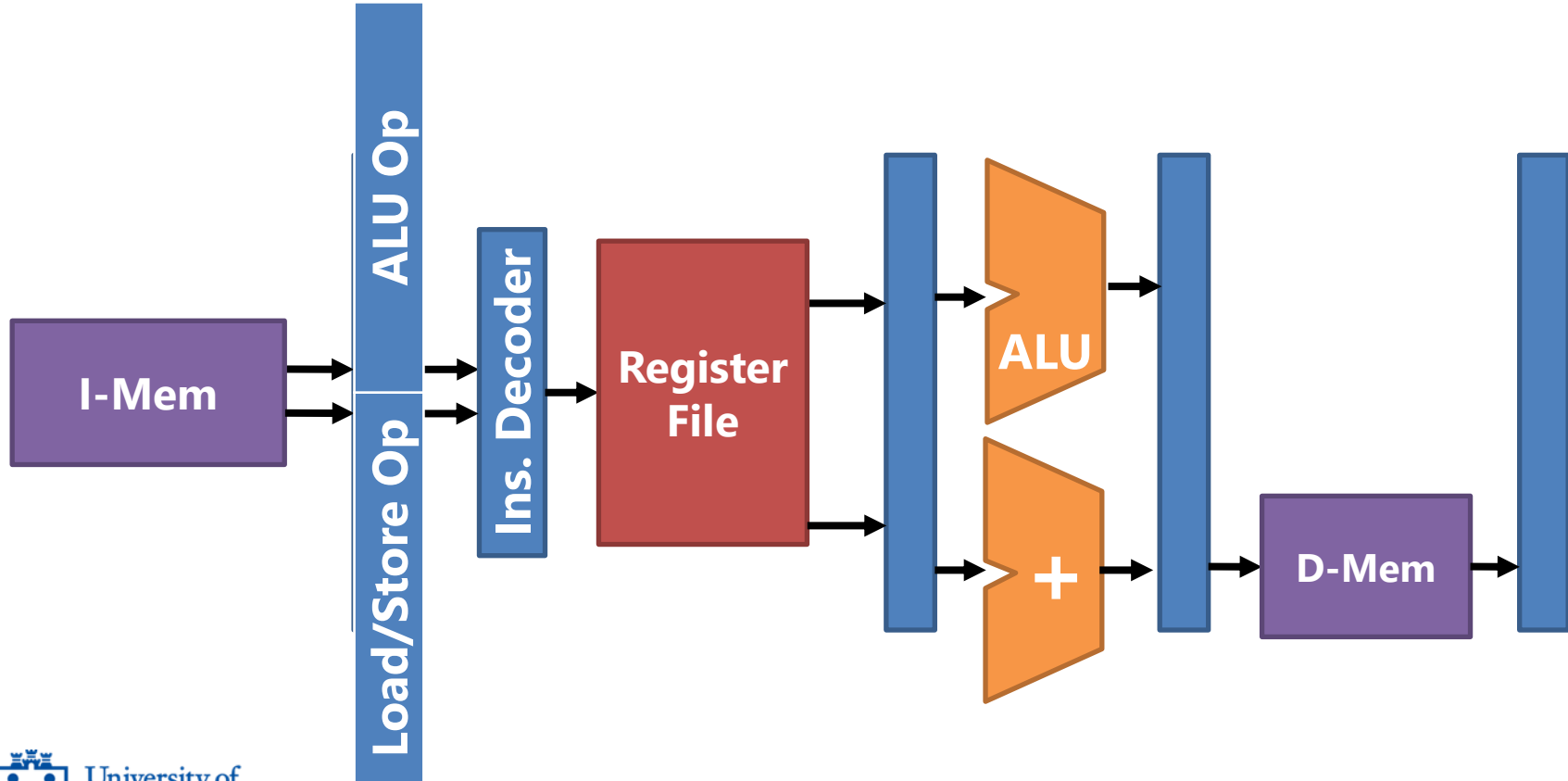
- Or another example could be:

`nop`

`lw $t1, -4($s1)`

# A VLIW instruction is one instruction

- For all purposes, a VLIW instruction acts like one instruction
  - It moves as a unit through the pipeline



# VLIW instruction encoding for example

**nop**

**lw** \$t0, 0(\$s1)

**addi** \$t2, \$t2, -8

**lw** \$t1, -4(\$s1)

**add** \$t3, \$t0, \$s1

**nop**

**add** \$t4, \$s1, \$s1

**sw** \$t5, 8(\$t3)

**nop**

**sw** \$t6, 4(\$s1)

| Inst | ALU Op         | Load/Store Op |
|------|----------------|---------------|
| 1    | <b>nop</b>     | <b>lw</b> t0  |
| 2    | <b>addi</b> t2 | <b>lw</b> t1  |
| 3    | <b>add</b> t3  | <b>nop</b>    |
| 4    | <b>add</b> t4  | <b>sw</b> t5  |
| 5    | <b>nop</b>     | <b>sw</b> t6  |

- Each square is an instruction.  
(There are 5 instructions.)
- Nops are inserted by the compiler.



# VLIW instruction encoding (after reordering)

**add** \$t4, \$s1, \$s1  
**lw** \$t0, 0(\$s1)

**addi** \$t2, \$t2, -8  
**lw** \$t1, -4(\$s1)

**add** \$t3, \$t0, \$s1  
**sw** \$t6, 4(\$s1)

**nop**  
**sw** \$t5, 8(\$t3)

| Inst | ALU Op         | Load/Store Op |
|------|----------------|---------------|
| 1    | <b>add</b> t4  | <b>lw</b> t0  |
| 2    | <b>addi</b> t2 | <b>lw</b> t1  |
| 3    | <b>add</b> t3  | <b>sw</b> t6  |
| 4    | <b>nop</b>     | <b>sw</b> t5  |

- Same program with 4 instructions!

# VLIW Architectures are (Very) Power Efficient

- All scheduling is done by the compiler offline
- No need for the Hazard Detection Unit
  - Nops have already been inserted by the compiler
- No need for a dynamic scheduler
  - Which can be even more power hungry than the HDU
- Will still need the Forwarding Unit
  - Unless you are willing to insert a lot of data hazard nops
  - But VLIW processors aren't typically deeply pipelined  
(They get performance out of wide execution, not frequency)

# VLIW Software is not very Portable

- Q: Wouldn't this tie the ISA to a particular processor design?
  - One that is 2-wide and has an ALU unit and a Load/Store unit?
  - What if future processors are wider or have different designs?
- Drawback of VLIW: ties binary to a particular processor
  - Code must be recompiled repeatedly for future processors
  - Not suitable for releasing general purpose software
  - Reason VLIW is most often used for embedded software  
(Because embedded software is not expected to be portable)
- Is there a way to get around this problem?

# Future Proofing VLIW Software

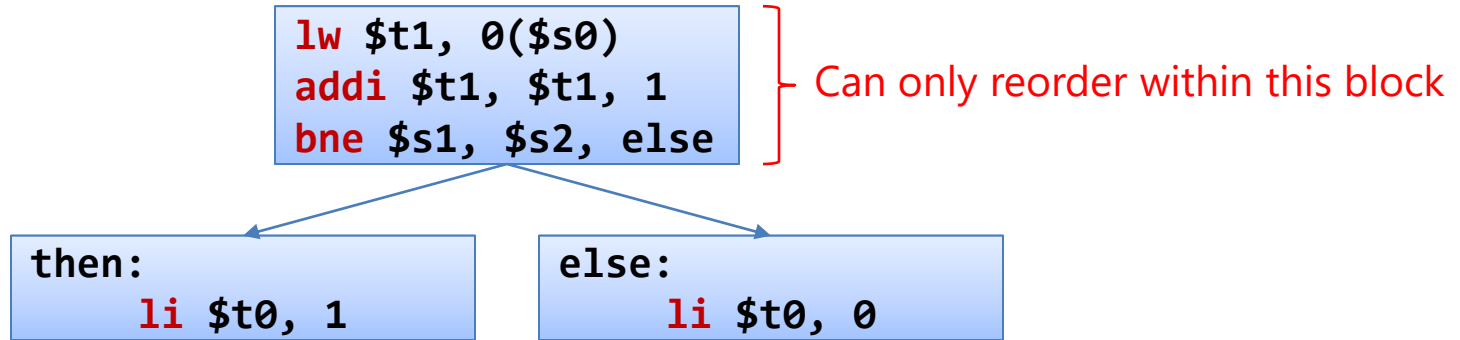
- There are mainly two ways VLIW software can become portable
  1. Create bundles, not instructions
    - A bundle is a group of operations that can execute together
    - Wider processors can fetch multiple bundles in one cycle
      - A “stop” bit tells processor to stop fetching the next bundle
    - Intel Itanium EPIC(Explicitly parallel instruction computing)
  2. Binary translation
    - Have firmware translate binary to new VLIW ISA on the fly
    - Doesn't this go against the power efficiency of VLIWs?
      - Yes, but if SW runs for long time, one-time translation is nothing
    - Transmeta Crusoe converted x86 to VLIW so it runs x86 apps!

# Success of VLIW depends on the Compiler

- Up to the compiler to create schedule with minimal nops
  - Use reordering to fill nops with useful operations
- All the challenges of static scheduling apply here too
  - Compiler must know about the pipeline (already discussed this)
  - Length of MEM stage is hard to predict (due to Memory Wall)
  - Data dependencies are hard to figure out by the compiler
- But these challenges become especially acute for VLIW
  - For 4-wide VLIW, need to find 4 operations to fill “one” bubble!
  - Operations in one instruction must be data independent
    - Data forwarding will not work within one instruction!

# Predicates Help in Compiler Scheduling

- Use of predicates can be a big help in finding useful operations
  - Control dependencies (as well as data deps) prevent reordering



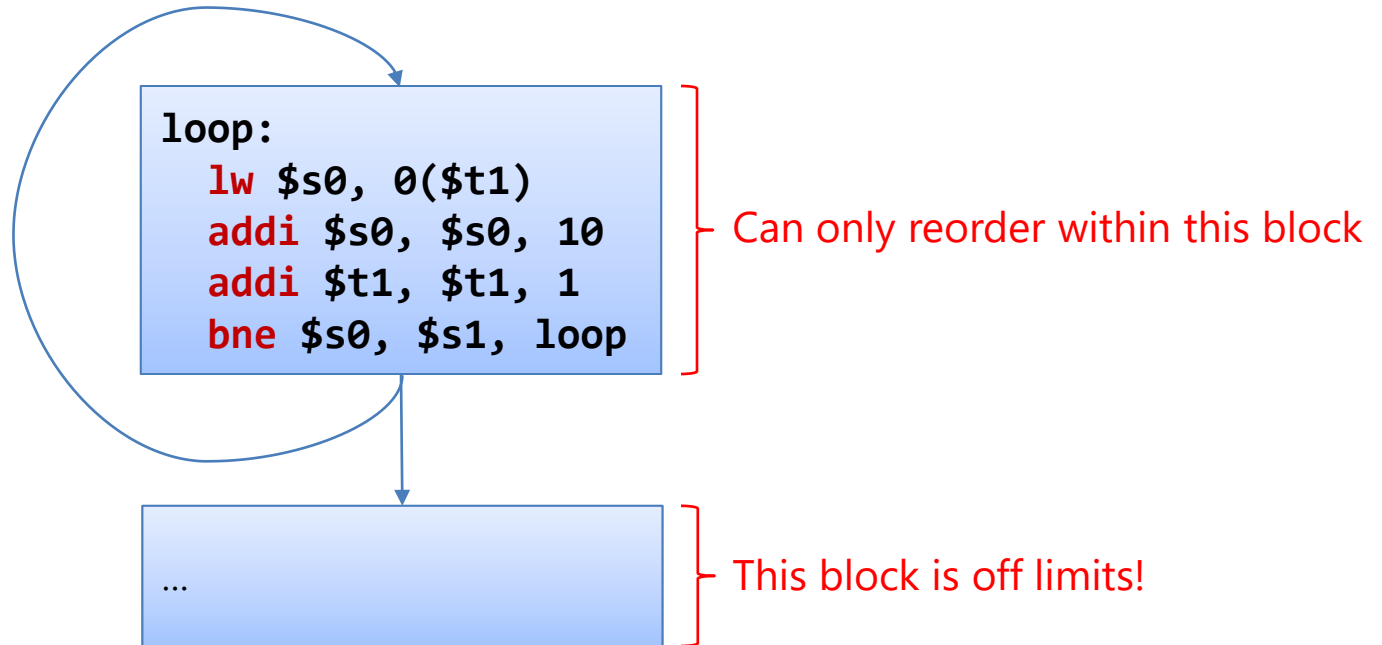
- Predicates can convert if-then-else code into straight-line code

```
pne $p1, $s1, $s2
lw $t1, 0($s0)
li.p $t0, 1, !$p1
li.p $t0, 0, $p1
addi $t1, $t1, 1
```

Can reorder within a larger window!

# But Loops are Particularly Challenging

- **Loops** are particularly challenging to the compiler. Why?
  - Scheduling is limited to within the loop
  - For tight loops, not much compiler can do with a handful of insts



# Compiler Scheduling of a Loop

- Suppose we had this example loop (in MIPS):

**Loop:**

```
lw    $t0, 0($s1)           // $t0 = $s1[0]
add   $t0, $t0, $s2          // $t0 = $t0 + $s2
sw    $t0, 0($s1)           // $s1[0] = $t0
addi  $s1, $s1, -4           // $s1-- decrement pointer
bne   $s1, $zero, Loop       // branch if $s1 != 0
```

- It's basically iterating over an array adding **\$s2** to each element

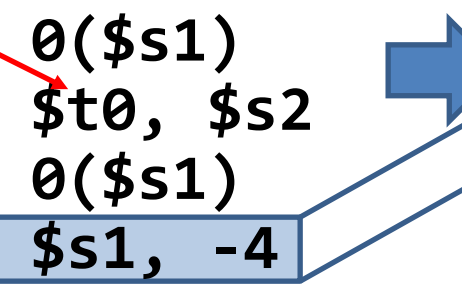


# Compiler Scheduling of a Loop

- Let's first reschedule to hide the use-after-load hazard

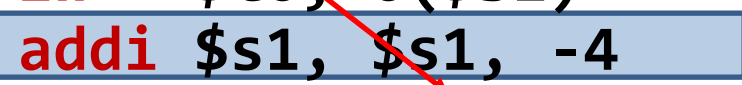
Loop:

```
lw    $t0, 0($s1)
add   $t0, $t0, $s2
sw    $t0, 0($s1)
addi  $s1, $s1, -4
bne   $s1, $zero, Loop
```



Loop:

```
lw    $t0, 0($s1)
addi  $s1, $s1, -4
add   $t0, $t0, $s2
sw    $t0, 4($s1)
bne   $s1, $zero, Loop
```



- Now the dependence on **\$t0** is further away
- Note we broke a WAR (Write-After-Read) dependence between:
  - sw** \$t0, 0(\$s1)
  - addi** \$s1, \$s1, -4
- But we compensated by changing the **sw** offset to 4:
  - sw** \$t0, 4(\$s1)

# Compiler Scheduling of a Loop

- Below is the VLIW representation of the rescheduled MIPS code:

Loop:

```
lw    $t0, 0($s1)
addi  $s1, $s1, -4
add   $t0, $t0, $s2
sw    $t0, 4($s1)
bne   $s1, $zero, Loop
```

|       | ALU/Branch Op               | Load/Store Op           |
|-------|-----------------------------|-------------------------|
| Loop: | <b>addi</b> \$s1, \$s1, -4  | <b>lw</b> \$t0, 0(\$s1) |
|       | <b>nop</b>                  | <b>nop</b>              |
|       | <b>add</b> \$t0, \$t0, \$s2 | <b>nop</b>              |
|       | <b>bne</b> \$s1, \$0, Loop  | <b>sw</b> \$t0, 4(\$s1) |

- We can't fill any further nops due to data hazards
- In terms of MIPS instructions,  $IPC = 5 / 4 = 1.25$ 
  - Ideally, IPC can reach 2 so we are not doing very well here
- Is there a way compiler can expand the "window" for scheduling?
  - Idea: use *multiple iterations* of the loop for scheduling!

# Loop unrolling

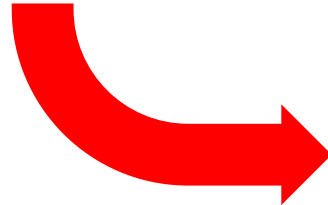
---

# What is Loop Unrolling?

- **Loop unrolling** : a compiler technique to enlarge loop body
  - By duplicating loop body for an X number of iterations

```
for(i = 0; i < 100; i++)  
    a[i] = b[i] + c[i];
```

**Original loop**



**Unrolled loop (2X)**

```
for(i = 0; i < 100; i += 2){  
    a[i] = b[i] + c[i];  
    a[i+1] = b[i+1] + c[i+1];  
}
```

- What does this buy us?
  - More instructions inside loop to reorder and hide bubbles
  - And less instructions to execute as a whole
    - Less frequent loop branches
    - Two i++ are merged into one i += 2

# Let's try unrolling our example code

- If you unroll the left loop 2X, you get the right loop:

Loop:

```
lw    $t0, 0($s1)
addi  $s1, $s1, -4
add   $t0, $t0, $s2
sw    $t0, 4($s1)
bne   $s1, $zero, Loop
```



Loop:

```
lw    $t0, 0($s1)
lw    $t1, -4($s1)
addi  $s1, $s1, -8
add   $t0, $t0, $s2
add   $t1, $t1, $s2
sw    $t0, 8($s1)
sw    $t1, 4($s1)
bne   $s1, $zero, Loop
```

- Most instructions are duplicated but using **\$t1** instead of **\$t0**
- Duplicates of **addi \$s1, \$s1, -4** are merged into one instruction:
  - **addi \$s1, \$s1, -8**

# Let's try unrolling our example code

- 2X Unrolled loop converted to VLIW:

Loop:

```
lw    $t0, 0($s1)
lw    $t1, -4($s1)
addi  $s1, $s1, -8
add   $t0, $t0, $s2
add   $t1, $t1, $s2
sw    $t0, 8($s1)
sw    $t1, 4($s1)
bne   $s1, $zero, Loop
```

|       | ALU/Branch Op               | Load/Store Op            |
|-------|-----------------------------|--------------------------|
| Loop: | <b>nop</b>                  | <b>lw</b> \$t0, 0(\$s1)  |
|       | <b>addi</b> \$s1, \$s1, -8  | <b>lw</b> \$t1, -4(\$s1) |
|       | <b>add</b> \$t0, \$t0, \$s2 | <b>nop</b>               |
|       | <b>add</b> \$t1, \$t1, \$s2 | <b>sw</b> \$t0, 8(\$s1)  |
|       | <b>bne</b> \$s1, \$0, Loop  | <b>sw</b> \$t1, 4(\$s1)  |

- Now we spend 5 cycles for 2 iterations of the loop
  - So,  $5 / 2 = 2.5$  cycles per iteration
  - Much better than the previous 4 cycles for 1 iteration!

# Let's try unrolling our example code 4X

- 4X Unrolled loop converted to VLIW:

|       | ALU/Branch Op               | Load/Store Op            | Inst |
|-------|-----------------------------|--------------------------|------|
| Loop: | <b>addi</b> \$s1, \$s1, -16 | <b>lw</b> \$t0, 0(\$s1)  | 1    |
|       | <b>nop</b>                  | <b>lw</b> \$t1, 12(\$s1) | 2    |
|       | <b>add</b> \$t0, \$t0, \$s2 | <b>lw</b> \$t2, 8(\$s1)  | 3    |
|       | <b>add</b> \$t1, \$t1, \$s2 | <b>lw</b> \$t3, 4(\$s1)  | 4    |
|       | <b>add</b> \$t2, \$t1, \$s2 | <b>sw</b> \$t0, 16(\$s1) | 5    |
|       | <b>add</b> \$t3, \$t1, \$s2 | <b>sw</b> \$t1, 12(\$s1) | 6    |
|       | <b>nop</b>                  | <b>sw</b> \$t2, 8(\$s1)  | 7    |
|       | <b>bne</b> \$s1, \$0, Loop  | <b>sw</b> \$t3, 4(\$s1)  | 8    |

- Now we spend 8 cycles for 4 iterations of the loop
  - So,  $8 / 4 = 2$  cycles per iteration
  - Even better 2.5 cycles per iteration for 2X unrolling

# What happens when you unroll 8X?

- 8X Unrolled loop converted to VLIW:

|       | ALU/Branch Op               | Load/Store Op            | Inst |
|-------|-----------------------------|--------------------------|------|
| Loop: | <b>addi</b> \$s1, \$s1, -32 | <b>lw</b> \$t0, 0(\$s1)  | 1    |
|       | <b>nop</b>                  | <b>lw</b> \$t1, 28(\$s1) | 2    |
|       | ...                         | ...                      | ...  |
|       | <b>add</b> \$t1, \$t1, \$s2 | <b>lw</b> \$t7, 4(\$s1)  | 8    |
|       | <b>add</b> \$t2, \$t1, \$s2 | <b>sw</b> \$t0, 32(\$s1) | 9    |
|       | <b>add</b> \$t3, \$t1, \$s2 | <b>sw</b> \$t1, 28(\$s1) | 10   |
|       | ...                         | ...                      | ...  |
|       | <b>bne</b> \$s1, \$0, Loop  | <b>sw</b> \$t7, 4(\$s1)  | 16   |

- Now we spend 16 cycles for 8 iterations of the loop
  - So,  $16 / 8 = 2$  cycles per iteration (no improvement over 4X)
  - 2 is minimum because you need one **lw** and one **sw** per iteration



# When should the compiler stop unrolling?

- Obviously when there is no longer a benefit as we saw just now
  - But there are other constraints that can prevent unrolling
1. Limitation in number of registers
    - More unrolling uses more registers \$t0, \$t1, \$t2, ...
    - For this reason, VLIW ISAs have many more registers than MIPS
      - Intel Itanium has 256 registers!
  2. Limitation in code space
    - More unrolling means more code bloat
    - Embedded processors don't have lots of code memory
    - Matters even for general purpose processors because of caching  
(Code that overflows i-cache can lead to lots of cache misses)

# List Scheduling

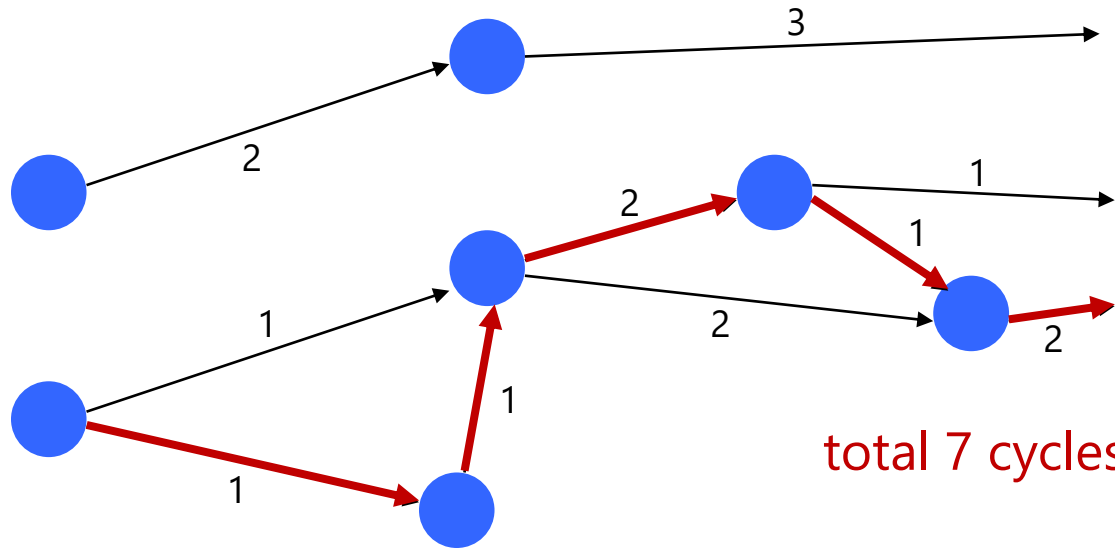
---

# How does the compiler schedule instructions?

- Compiler will first expand the **instruction window** that it looks at
  - Instruction window: block of code without branches
  - Compiler uses predication and loop unrolling
- Once compiler has a sizable window, it will construct the schedule
- A popular scheduling algorithm is ***list scheduling***
  - Idea: list instructions in some order of priority and schedule
  - Instructions on the **critical path** should be prioritized
- List scheduling can be used with any statically scheduled processor
  - Simple single-issue statically scheduled processor (not just VLIW)
  - GPUs, which are also statically scheduled

# Critical Path in Code

- At below is a data dependence graph for a code with 7 instructions
  - Nodes are instructions
  - Arrows are dependencies annotated with required delay
- Q: How long is the critical path in this code?

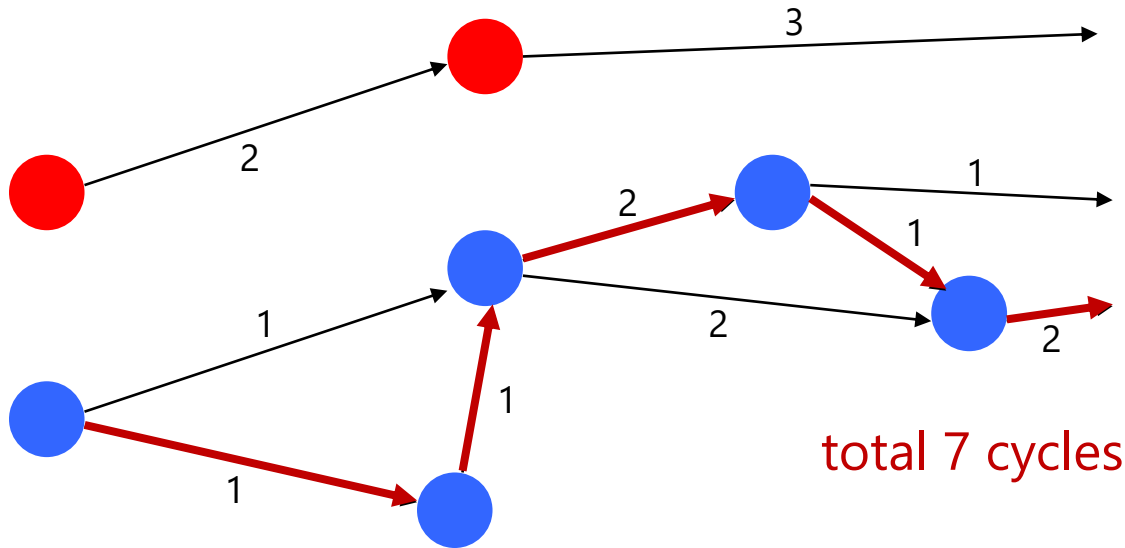


total 7 cycles

That means, at minimum, this code will take 7 cycles, period. Regardless of how wide your processor is or how well you do your scheduling.

# Instruction Level Parallelism (ILP)

- The 7 cycles is achievable only through **instruction level parallelism**
  - That is, parallel execution of instructions
  - The nodes marked in red can execute in parallel with blue nodes
- This tell us that this code is where a VLIW processor can shine



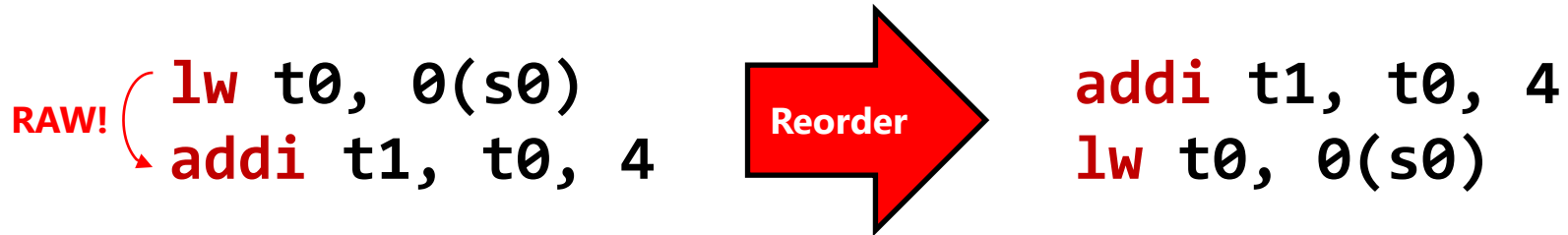
total 7 cycles

# Maximizing Instruction Level Parallelism (ILP)

- The more ILP code has, the more VLIW will shine
- So before list scheduling, compiler maximizes ILP in code
  - What constrains ILP? Data dependencies!
  - Some data dependencies can be removed by the compiler
- There are 3 types of data dependencies actually:
  - **RAW (Read-After-Write)**: cannot be removed
  - **WAR (Write-After-Read)**: can be removed
  - **WAW (Write-After-Write)**: can also be removed
- How about Read-After-Read? Not a data dependency.

# Read-After-Write (RAW) Dependency

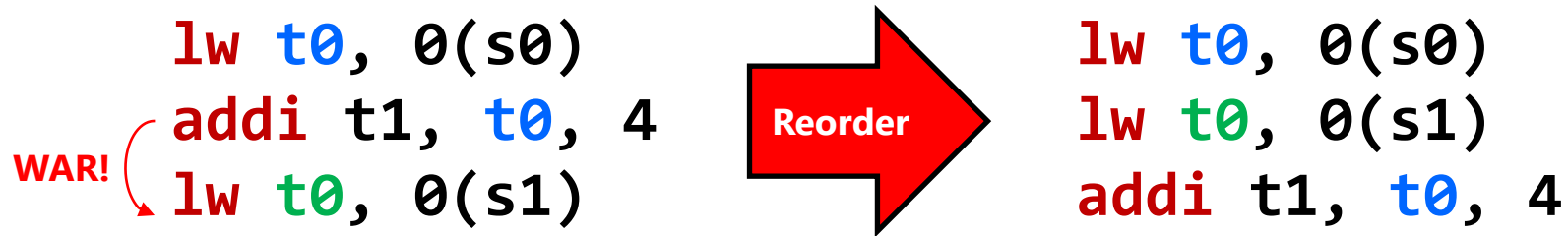
- RAW dependencies are also called **true dependencies**
  - In the sense that other dependencies are not “real” dependencies
- Suppose we reorder this snippet of code:



- The code is incorrect because now **t1** has a wrong value
  - No amount of compiler tinkering will allow this reordering
  - Value must be loaded into **t0** before being used to compute **t1**

# Write-After-Read (WAR) Dependency

- WAR dependencies are also called ***anti-dependencies***
  - In the sense that they are the opposite of true dependencies
- Suppose we reorder this snippet of code:



- The code is again incorrect because `t1` has the wrong value
  - `addi` should not read `t0` produced by `lw t0, 0(s1)`
- Q: Is there a way for `addi` to *not use* that value?
  - `t0` and `t0` contain different values. Why use the same register?
  - Just **rename** register `t0` to some other register!



# Removing WAR with SSA

- **Static Single Assignment:** Renaming registers with different values
  - A register is assigned a value only a single time (never reused)
- Reordering after converting to SSA form:

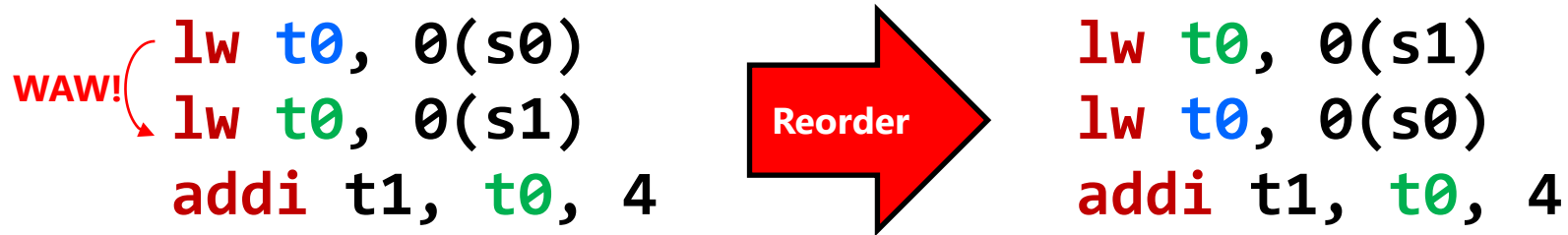


- Note how destination registers always use a new register
  - Yes, if you do this, you will need lots of registers
  - But, no more WAR dependencies!

# Write-After-Write (WAW) Dependency

- WAW dependencies are also called **false dependencies**
  - In the sense that they are not real dependencies

- Suppose we reorder this snippet of code:



- The code is again incorrect because **t1** has the wrong value
  - **addi** should not read **t0** produced by **lw t0, 0(s0)**
- Q: Is there a way for **addi** to *not use* that value?
  - Again, **rename** register **t0** to some other register!

# Removing WAW with SSA

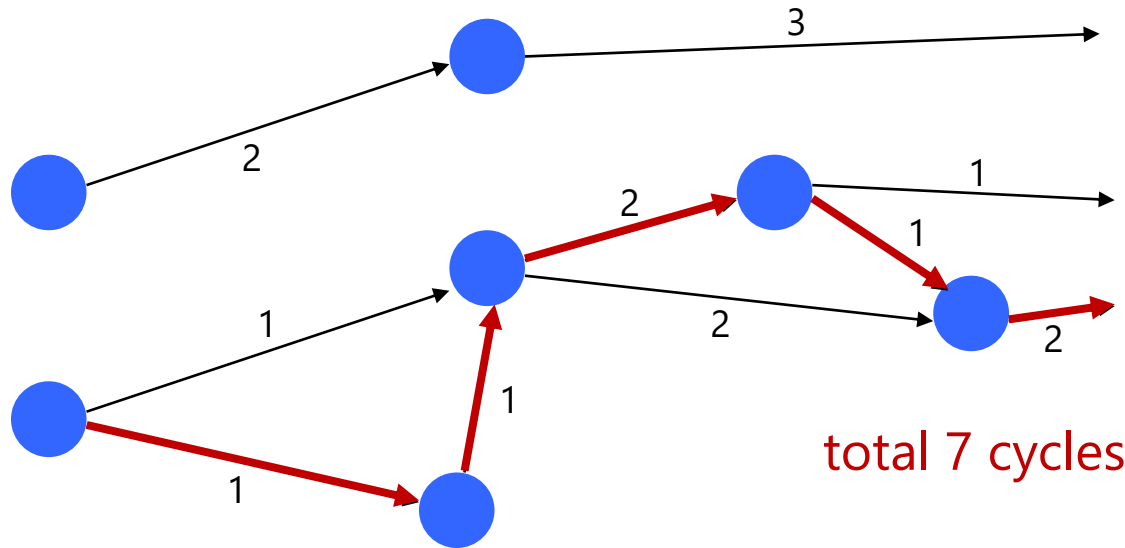
- Again, Static Single Assignment (SSA) to the rescue!
  - SSA removes both WAR and WAW dependencies
- Reordering after converting to SSA form:



- SSA form is now the norm in all mature compilers
  - Clang / LLVM ("Apple" Compiler)
  - GCC (GNU C Compiler)
  - Java Hotspot / OpenJDK Compiler
  - Chrome JavaScript Compiler

# List Scheduling

- Back to our original example.
- The critical path length is 7 cycles but that is not always achievable
  - If processor is not wide enough for the available parallelism
  - If compiler does a bad job at scheduling instructions
- List scheduling **guarantees** that the compiler does an optimal job



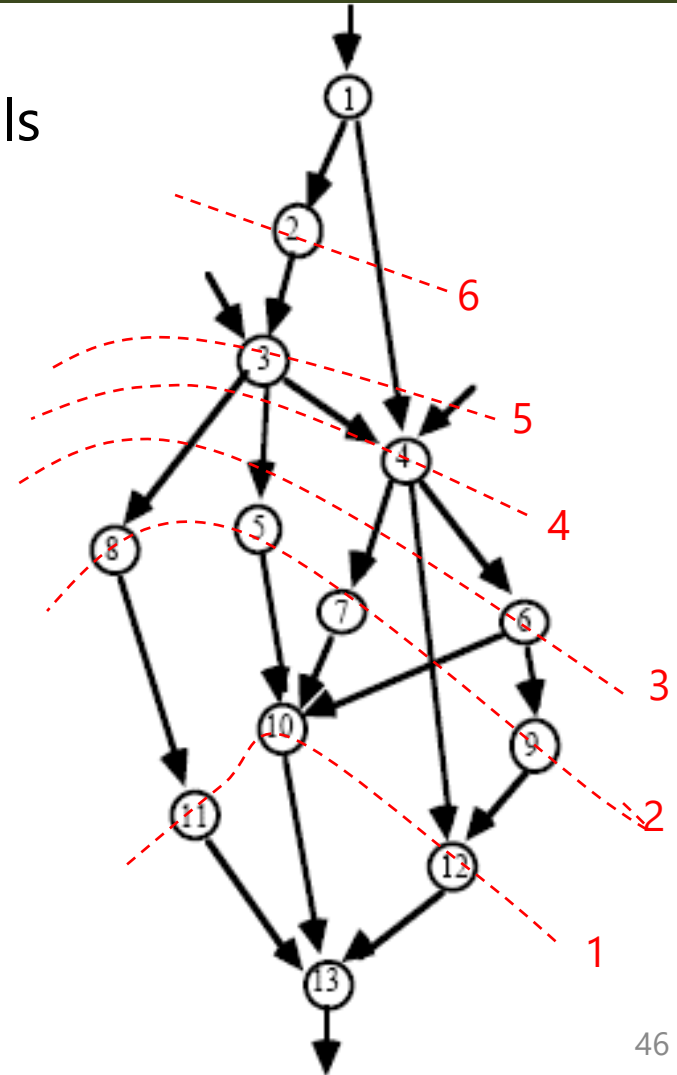
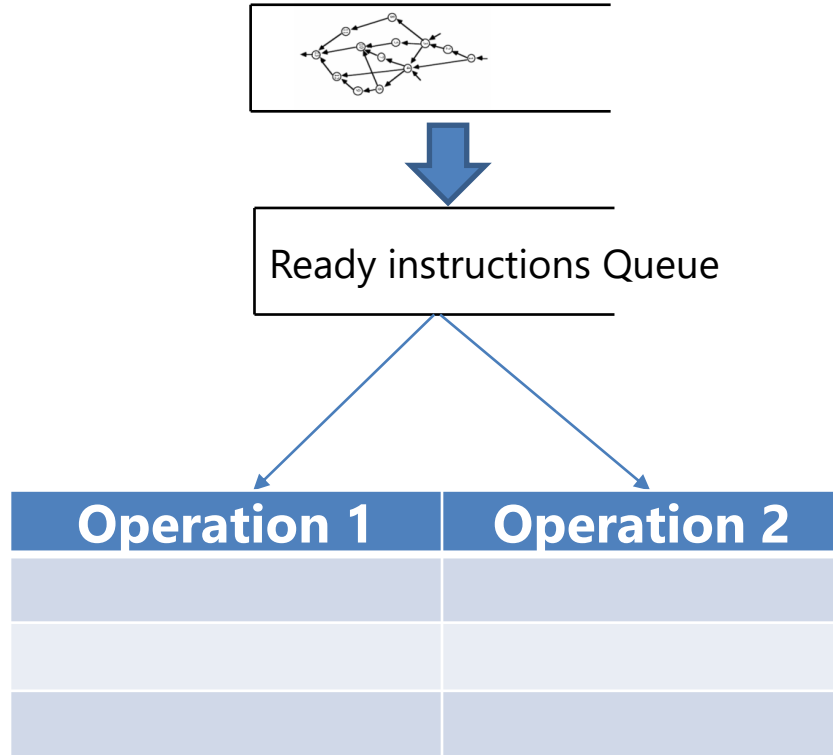
Q: Can the graph be cyclic?

# List Scheduling Algorithm

- Idea: Prioritize instructions on the critical path
- Steps:
  1. Create a **data dependence graph**
  2. Assign a **priority** to each node (instruction)
    - Priority = **critical path** length starting from that node
  3. Schedule nodes one by one starting from **ready** instructions
    - Ready = all dependencies have been fulfilled  
(Initially, only roots of dependency chains are ready)
    - When there are multiple nodes that are ready  
→ Choose the node with the highest priority

# List Scheduling Example

- Assume all edges have a delay of 1
  - Red dashed lines indicate priority levels



# List Scheduling Example

- This will result in the following schedule:

| Operation 1 | Operation 2 |
|-------------|-------------|
| 1           |             |
| 2           |             |
| 3           |             |
| 4           | 5           |
| 6           | 7           |
| 8           | 9           |
| 10          | 11          |
| 12          |             |
| 13          |             |

- 9 cycles. We couldn't achieve 7 cycles!
  - But could've if we had a wider processor

