VLIW Processors

CS 1541
Wonsun Ahn

University of Pittsburgh

Limits on Deep Pipelining

- Ideally, $\text{CycleTime}_{\text{Pipelined}} = \text{CycleTime}_{\text{SingleCycle}}$ / Number of Stages
  - In theory, can indefinitely improve performance with more stages

- Limitation 1: **Cycle time** does not improve indefinitely
  - Latch delay + unbalanced stages (manufacturing variability)
- Limitation 2: **CPI** tends to increase with deep pipelines
  - Penalty due to branch misprediction increases
  - Stalls due to data hazards cause more bubbles

- Is there another way to improve performance?

University of Pittsburgh

What if we improve CPI?

- Remember the three components of performance?

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

- 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

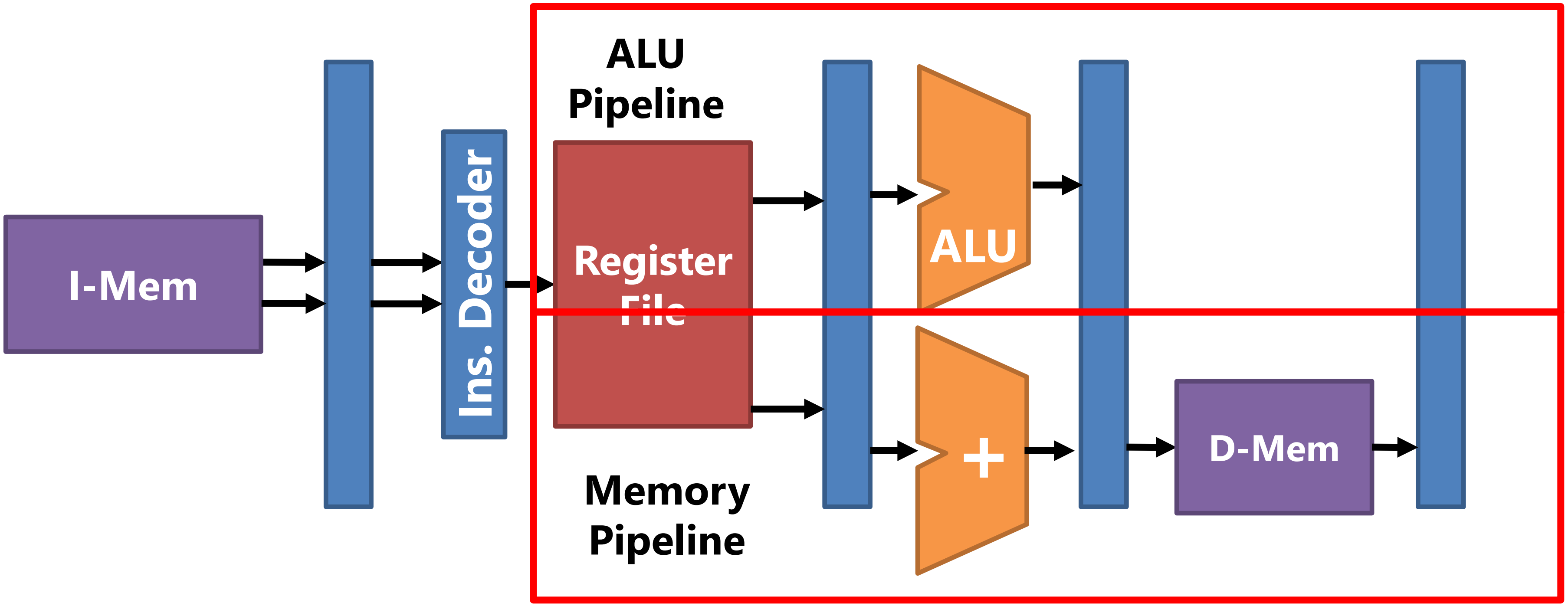
- How about if we execute **two** instructions each cycle?
  - Maybe, fetch one ALU instruction and one load/store instruction

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

2X speedup with mere addition of one + unit?

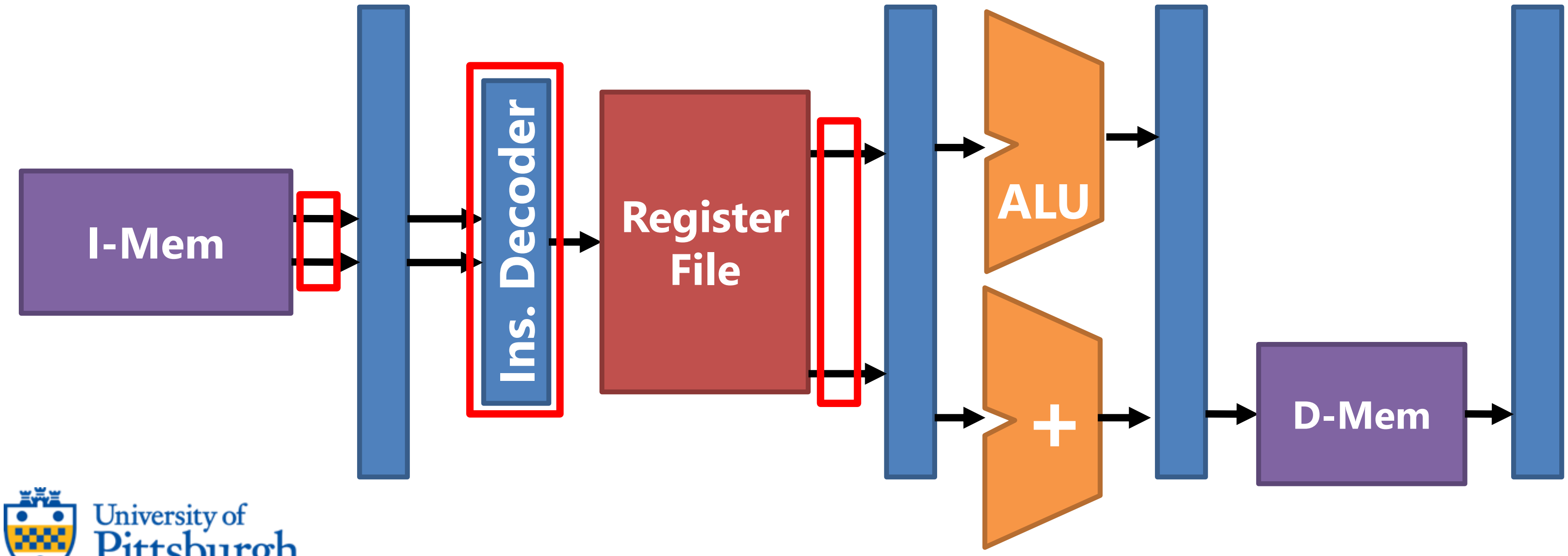
- One pipeline for ALU/Branches and one for loads and stores



- Causes contention on shared resources and structural hazards!

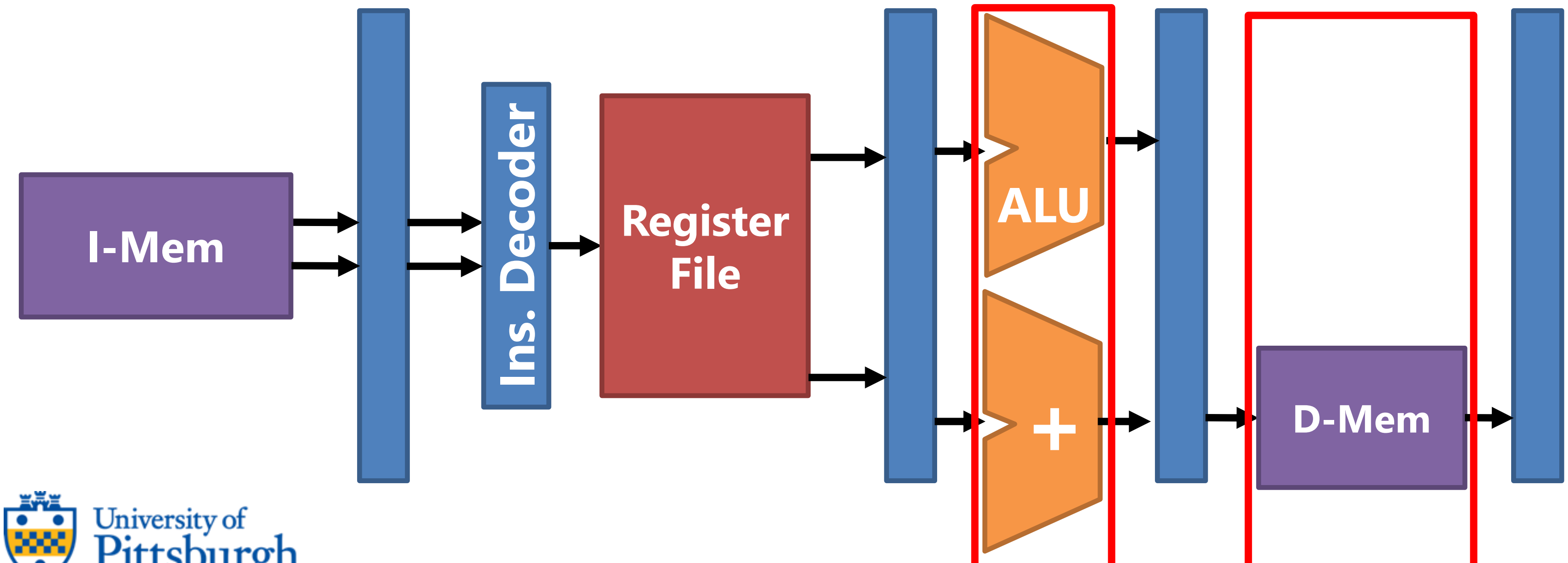
Extra investments needed to avoid structural hazard

- At max, every clock cycle in this 2-wide processor:
  - 2 instructions fetched from I-MEM → 1 extra read port needed
  - 2 instructions decoded → Extra wide instruction decoder needed
  - 3 reads from register file → 1 extra read port needed
  - 2 writes to register file → 1 extra write port needed



And fundamental structural hazards still remain

- 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
  - ALU pipeline does not have a MEM unit to access memory



Structural Hazard in Functional Units: Example

- 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

University of Pittsburgh

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		lw t0
2	addi t2	lw t1
3	add t3	
4	add t4	sw t5
5		sw 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:

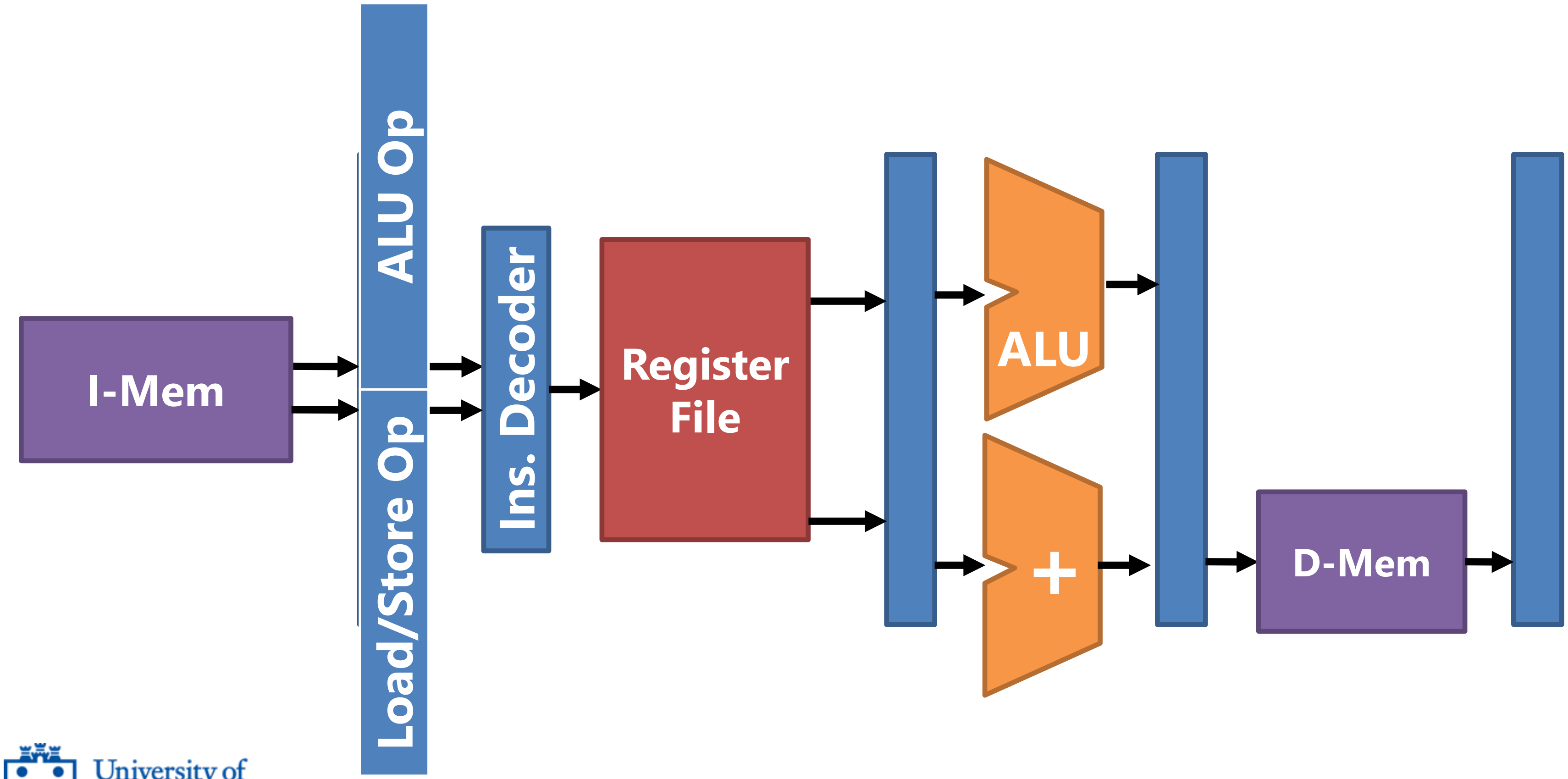
`addi $t2, $t0, -8`	`lw $t1, -4($s1)`

- 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	nop	lw t0
2	addi t2	lw t1
3	add t3	nop
4	add t4	sw t5
5	nop	sw t6

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

University of Pittsburgh

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	add t4	lw t0
2	addi t2	lw t1
3	add t3	sw t6
4	nop	sw 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 are inserted by the compiler when necessary

- No need for a dynamic scheduler
  - Which can be even more power hungry than the HDU

- Even no need for the Forwarding Unit
  - If compiler is good enough and fill all bubbles with instructions
  - Or, may have cheap compiler-controlled forwarding in ISA

Challenges of VLIW

- All the challenges of static scheduling apply here **X 2**
- Review: what were the limitations?
  - Compiler must make **assumptions about the pipeline**
    → ISA now becomes much more than instruction set + registers
    → ISA restricts modification of pipeline in future generations
  - Compiler must do **scheduling without runtime information**
    → Length of MEM stage is hard to predict (due to Memory Wall)
    → Data dependencies are hard (must do pointer analysis)
- These limitations are exacerbated with VLIW

Not Portable due to Assumptions About Pipeline

- VLIW ties ISA to a particular processor design
  - One that is 2-wide and has an ALU op and a Load/Store op
  - What if future processors are wider or contain different ops?

- 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 any way to get around this problem?

Making VLIW Software Portable

- There are mainly two ways VLIW software can become portable

1. Allow CPU to exploit parallelism according to capability
    - Analogy: multithreaded software does not specify number of cores
        - SW: Makes parallelism explicit by coding using threads
        - CPU: Exploits parallelism to the extent it has number of cores
    - Portable VLIW: ISA does not specify number of ops in instruction
        - SW: Makes parallelism explicit by using bundles
            - Bundle: a group of ops that can execute together
            - Wider processors fetch several bundles to form one instruction
            - A "stop bit" tells processor to stop fetching the next bundle
        - Intel Itanium EPIC(Explicitly Parallel Instruction Computing)
            - A general-purpose ISA that uses bundles

Making VLIW Software Portable

- There are mainly two ways VLIW software can become portable

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
        - Translation can be cached in file system for next run
    - Transmeta processors convert x86 to an ultra low-power VLIW

- Other examples of binary translation
    - Apple Rosetta converts x86 to ARM ISA for M1 or M2 chips
    - NVIDIA GPUs convert PTX (Parallel Thread Execution) to SASS
        - SASS (Stream ASSembly) is different for every generation of GPU

Scheduling without Runtime Information

- Up to the compiler to create schedule with minimal nops
  - Use reordering to fill nops with useful operations

- All the challenges of static scheduling remain
  - Length of MEM stage is hard to predict (due to Memory Wall)
  - Data dependencies are hard to figure out (due to pointer analysis)

- And 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 (Obviously because they are executing on the same cycle)
  - Operations must also be control independent

Predicates Help in Compiler Scheduling

- Predicates can enlarge "instruction window" for scheduling
  - Reordering cannot happen across control dependencies

```
lw $t1, 0($s0)
addi $t1, $t1, 1
bne $s1, $s2, else
```

Can only reorder within this block

```
then:
  lw $t2, 1($s0)
```

```
else:
  lw $t2, 2($s0)
```

  - Predicates can convert if-then-else code into one big block

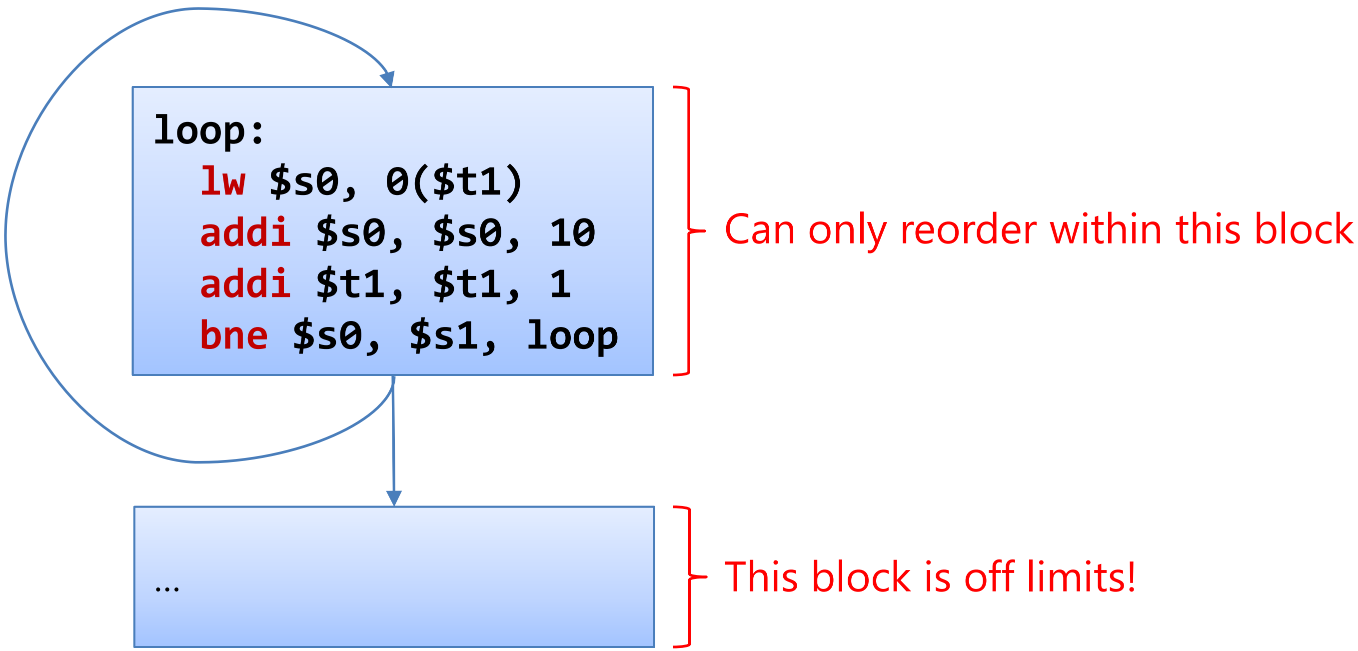
```
lw $t1, 0($s0)
pne $p1, $s1, $s2
lw $t2, 1($s0), !$p1
addi $t1, $t1, 1
lw $t2, 2($s0), $p1
```

Can reorder within a larger window!

University of Pittsburgh

But Predicates Cannot Remove Loopback Branches

- **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



Let's use this loop as a running example

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

Is equivalent to

```
for (p = &a[0]; p < &a[100]; p++)
  *p = *p + x;
```

Translated to MIPS

```
// Map p → $s1, x → $s2, and &a[100] → $s3
Loop:
  lw   $t0, 0($s1)          // $t0 = *p
  add  $t0, $t0, $s2        // $t0 = $t0 + x
  sw   $t0, 0($s1)          // *p = $t0
  addi $s1, $s1, 4          // p++
  blt  $s1, $s3, Loop       // loopback if p < &a[100]
```

This is the loop schedule on VLIW as-is

- This is the best the compiler can do with current order:

```
Loop:
  lw   $t0, 0($s1)
  add  $t0, $t0, $s2
  sw   $t0, 0($s1)
  addi $s1, $s1, 4
  blt  $s1, $s3, Loop
```

	ALU/Branch Op	Load/Store Op
Loop:	nop	lw $t0, 0($s1)
	nop	nop
	add $t0, $t0, $s2	nop
	addi $s1, $s1, 4	sw $t0, 0($s1)
	blt $s1, $s3, Loop	nop

- **lw** and **add** separated by 2 cycles → due to use-after-load on **$t0**
- **add** and **sw** separated by 1 cycle → due to dependence on **$t0**
- **addi** and **sw** on same cycle → **sw** uses old value of **$s1**
- **IPC = 5 / 5 = 1**. On a 2-wide VLIW. We need to better!
  - But chain of dependencies prevents any reordering... or does it?

Dependencies can be broken (with proper patch-up)

- We broke the WAR (Write-After-Read) dependence on **$s1!**

```
Loop:
  lw   $t0, 0($s1)
  addi $s1, $s1, 4
  add  $t0, $t0, $s2
  sw   $t0, -4($s1)
  blt  $s1, $s3, Loop
```

	ALU/Branch Op	Load/Store Op
Loop:	addi $s1, $s1, 4	lw $t0, 0($s1)
	nop	nop
	add $t0, $t0, $s2	nop
	blt $s1, $s3, Loop	sw $t0, -4($s1)

- Now, **sw** uses the new value of **$s1** produced by **addi**
- The compiler compensates by changing the **sw** offset by -4:
  - `sw $t0, 0($s1)` → `sw $t0, -4($s1)`
- Now, **IPC = 5 / 4 = 1.25**. Can we do even better?

Expand scheduling window using *multiple iterations* of the loop!

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] = a[i] + x;
```

Original loop

Unrolled loop (2X)

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

- What does this buy us?
    - And **expanded scheduling window** 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

1. Unroll the loop

```
Loop:
  lw   $t0, 0($s1)
  addi $s1, $s1, 4
  add  $t0, $t0, $s2
  sw   $t0, -4($s1)
  blt  $s1, $s3, Loop
```

Unroll 2X

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

  lw   $t1, 0($s1)
  addi $s1, $s1, 4
  add  $t1, $t1, $s2
  sw   $t1, -4($s1)

  blt  $s1, $s3, Loop
```

- Instructions are duplicated but using **`$t1`** instead of **`$t0`**
- This is intentional to minimize false dependencies during reordering

2. Interleave iterations to space out dependencies

```
Loop:
  lw   $t0, 0($s1)
  lw   $t1, 4($s1)

  addi $s1, $s1, 4
  addi $s1, $s1, 4

  add  $t0, $t0, $s2
  add  $t1, $t1, $s2

  sw   $t0, -8($s1)
  sw   $t1, -4($s1)

  blt  $s1, $s3, Loop
```

Reorder!

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

  lw   $t1, 0($s1)
  addi $s1, $s1, 4
  add  $t1, $t1, $s2
  sw   $t1, -4($s1)

  blt  $s1, $s3, Loop
```

- Interleaving iterations spaces out dependencies by 2X (unroll factor)

3. Merge induction variable increment

```
Loop:
  lw   $t0, 0($s1)
  lw   $t1, 4($s1)

  addi $s1, $s1, 4
  addi $s1, $s1, 4

  add  $t0, $t0, $s2
  add  $t1, $t1, $s2

  sw   $t0, -8($s1)
  sw   $t1, -4($s1)

  blt  $s1, $s3, Loop
```

Merge

```
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)

  blt  $s1, $s3, Loop
```

- Two **addi $s1, $s1, 4** are merged into **addi $s1, $s1, 8**

4. Schedule unrolled loop onto 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)
  blt  $s1, $s3, Loop
```

	ALU/Branch Op	Load/Store Op
Loop:	nop	lw $t0, 0($s1)
	addi $s1, $s1, 8	lw $t1, 4($s1)
	add $t0, $t0, $s2	nop
	add $t1, $t1, $s2	sw $t0, -8($s1)
	blt $s1, $s3, Loop	sw $t1, -4($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!

Unrolling loop 4X buys us even more speedup!

- 4X Unrolled loop converted to VLIW:

	ALU/Branch Op	Load/Store Op	Inst
Loop:	addi $s1, $s1, 16	lw $t0, 0($s1)	1
	nop	lw $t1, -12($s1)	2
	add $t0, $t0, $s2	lw $t2, -8($s1)	3
	add $t1, $t1, $s2	lw $t3, -4($s1)	4
	add $t2, $t1, $s2	sw $t0, -16($s1)	5
	add $t3, $t1, $s2	sw $t1, -12($s1)	6
	nop	sw $t2, -8($s1)	7
	blt $s1, $s3, Loop	sw $t3, -4($s1)	8

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

University of Pittsburgh

Unrolling beyond that won't buys us anything

- 8X Unrolled loop converted to VLIW:

	ALU/Branch Op	Load/Store Op	Inst
`Loop:`	`addi $s1, $s1, 32`	`lw $t0, 0($s1)`	1
	`nop`	`lw $t1, -28($s1)`	2
	…	…	…
	`add $t1, $t1, $s2`	`lw $t7, -4($s1)`	8
	`add $t2, $t1, $s2`	`sw $t0, -32($s1)`	9
	`add $t3, $t1, $s2`	`sw $t1, -28($s1)`	10
	…	…	…
	`blt $s1, $s3, Loop`	`sw $t7, -4($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

University of Pittsburgh

When should the compiler stop unrolling?

- When dependencies are already sufficiently spaced far apart
- There are constraints that can prevent unrolling even before that

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

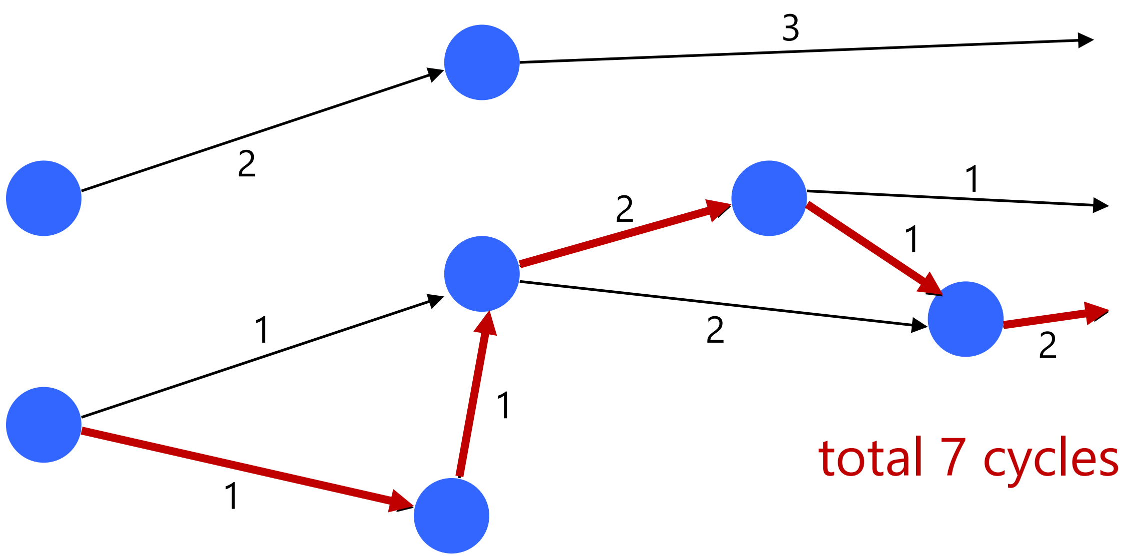
- Once compiler has a sizable window, it will construct the schedule

- 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 are also statically scheduled using list scheduling

Critical Path in Code

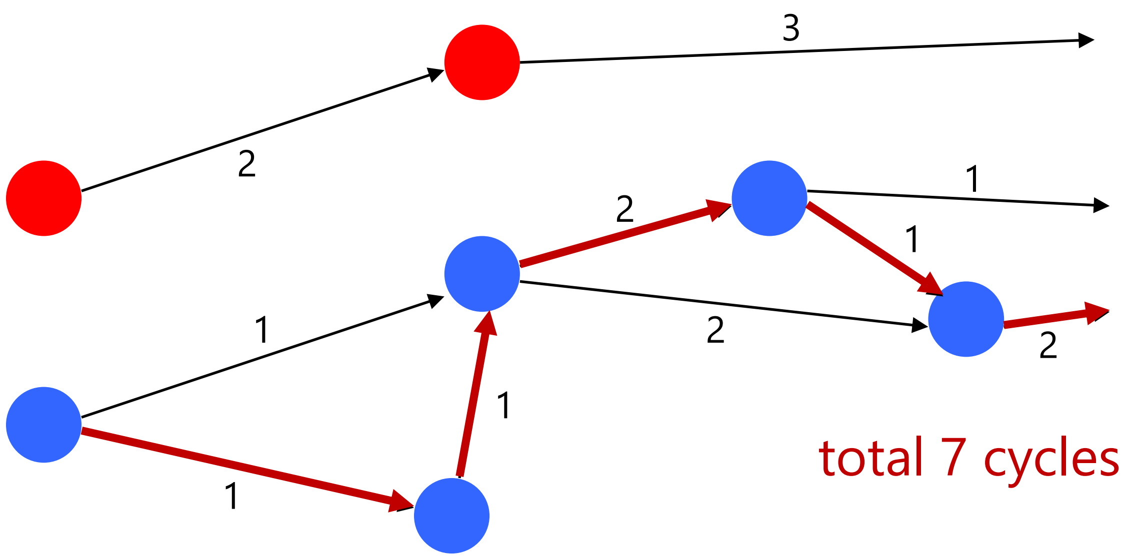
- At below is a data dependence graph for a code with 7 instructions
  - Nodes are instructions
  - Arrows are data dependencies annotated with required delay
- Q: How long is the critical path in this code?



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 thanks to **instruction level parallelism**
  - Property of graph that allows 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



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.

University of Pittsburgh

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:

RAW!

```
lw t0, 0(s0)
addi t1, t0, 4
```

Reorder

```
addi t1, t0, 4
lw t0, 0(s0)
```

- The code is incorrect because now **`t1`** has a wrong value
  - Value in **`t0`** can be read only after written
  - No amount of compiler tinkering will allow this reordering

University of Pittsburgh

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:

```
lw t0, 0(s0)
addi t1, t0, 4
lw t0, 0(s1)
```

WAR!

Reorder

```
lw t0, 0(s0)
lw t0, 0(s1)
addi t1, t0, 4
```

- 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!

University of Pittsburgh

Removing WAR with SSA

- ***Static Single Assignment***:
  - Renaming registers when a different value is stored into it
  - A register is assigned a value only a single time (never reused)

- Reordering after converting to SSA form:

NO WAR!

```
lw t0, 0(s0)
addi t1, t0, 4
lw t2, 0(s1)
```

Reorder

```
lw t0, 0(s0)
lw t2, 0(s1)
addi t1, t0, 4
```

- 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!

University of Pittsburgh

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:

WAW!

```
lw t0, 0(s0)
lw t0, 0(s1)
addi t1, t0, 4
```

Reorder

```
lw t0, 0(s1)
lw t0, 0(s0)
addi t1, t0, 4
```

- 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!

University of Pittsburgh

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:

NO WAW!

```
lw t0, 0(s0)
lw t1, 0(s1)
addi t2, t1, 4
```

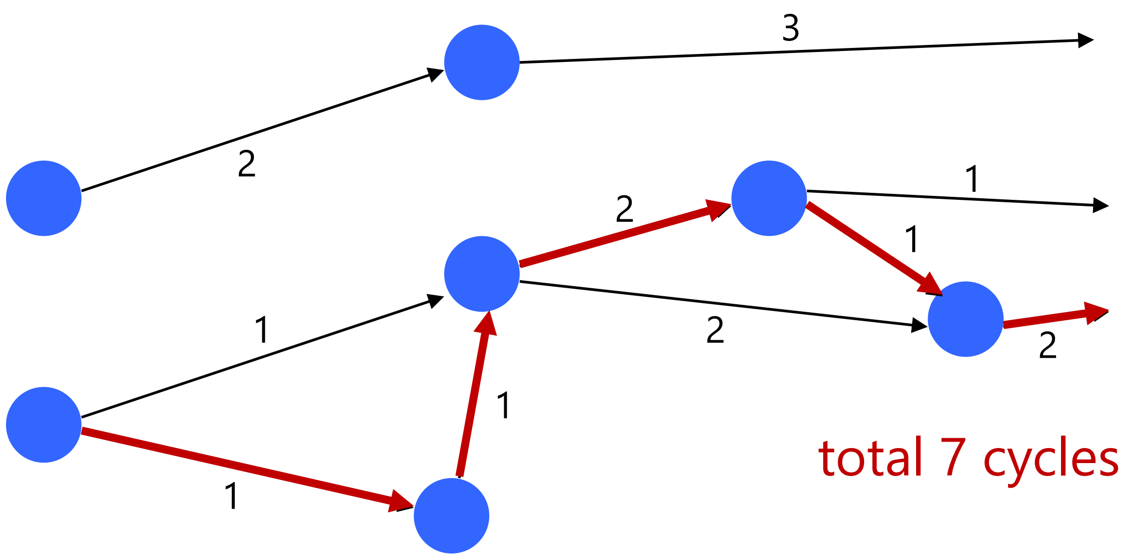
Reorder

```
lw t1, 0(s1)
lw t0, 0(s0)
addi t2, t1, 4
```

- 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

Back to List Scheduling

- With SSA, the only data dependences that remain are RAW ones.
- 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 compiler is within 2X of optimal



Q: Can the graph be cyclic?

List Scheduling is a Greedy Algorithm

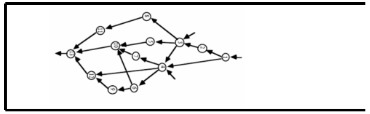
- Idea: Greedily prioritize instructions on the critical path

- 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

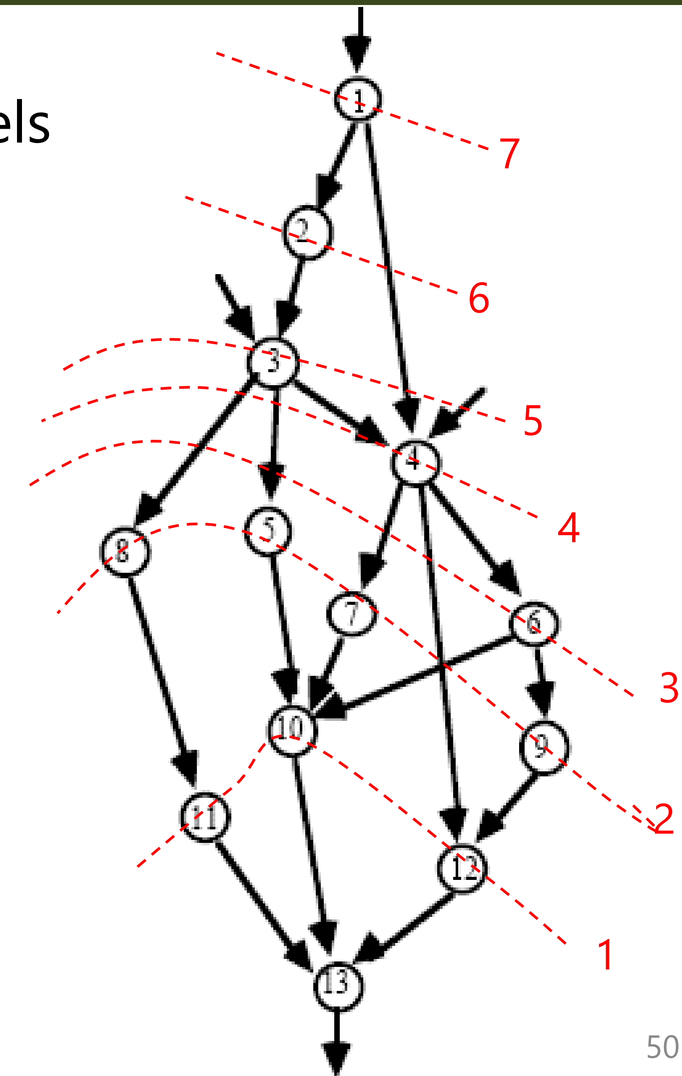
- Assume all edges have a delay of 1
  - Red dashed lines indicate priority levels

n insertions +
n removals.
If using priority heap
= O(nlogn)



Ready instructions List

Operation 1	Operation 2

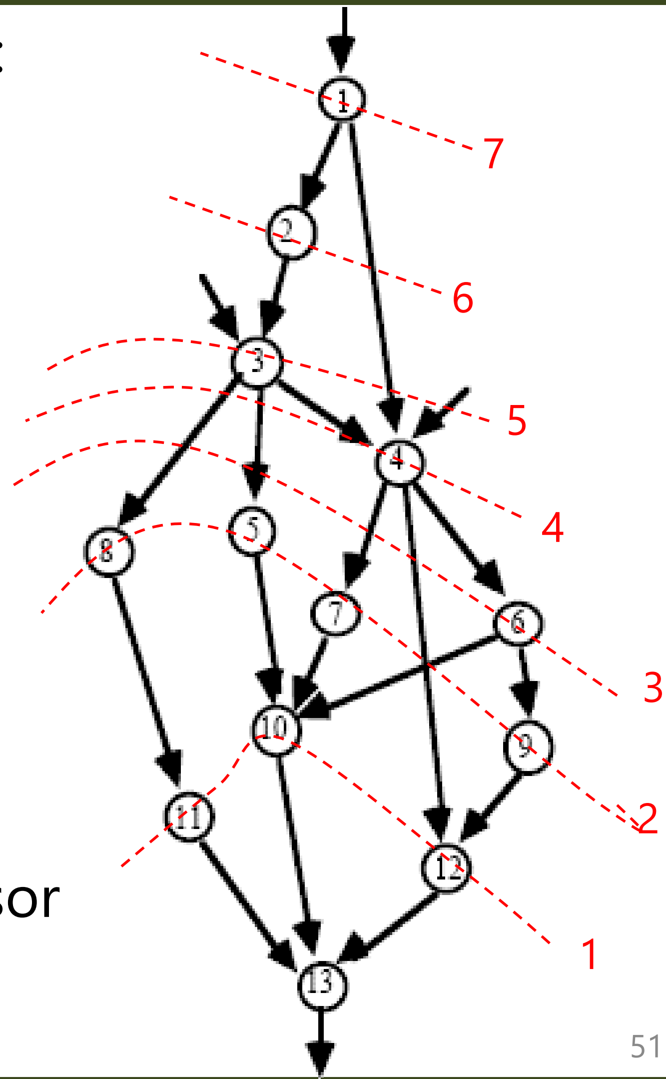


List Scheduling Example

- 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



University of Pittsburgh

List Scheduling is not optimal

- Optimal scheduling is an **NP-Complete** problem:
  - John L. Hennessy and Thomas Gross. "Postpass Code Optimization of Pipeline Constraints", ACM Trans. Program. Lang. Syst., July 1983.
- List scheduling achieves worst-case **2 – 1/n** of optimal schedule
  - Where **n** is the width of the processor
  - R. L. Graham. "Bounds on multiprocessing timing anomalies", SIAM Journal of Applied Mathematics, 1969.

- More elaborate algorithms exist to approach optimal
  - Using constraint solvers or machine learning

- Still, list scheduling using critical path lengths is most widely used

- GCC uses other heuristics in addition to critical path length:
  - https://github.com/gcc-mirror/gcc/blob/master/gcc/haifa-sched.cc