

VLIW Processors

CS/COE 1541 (Fall 2020)
Wonsun Ahn

Limits of 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
- 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 increases 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

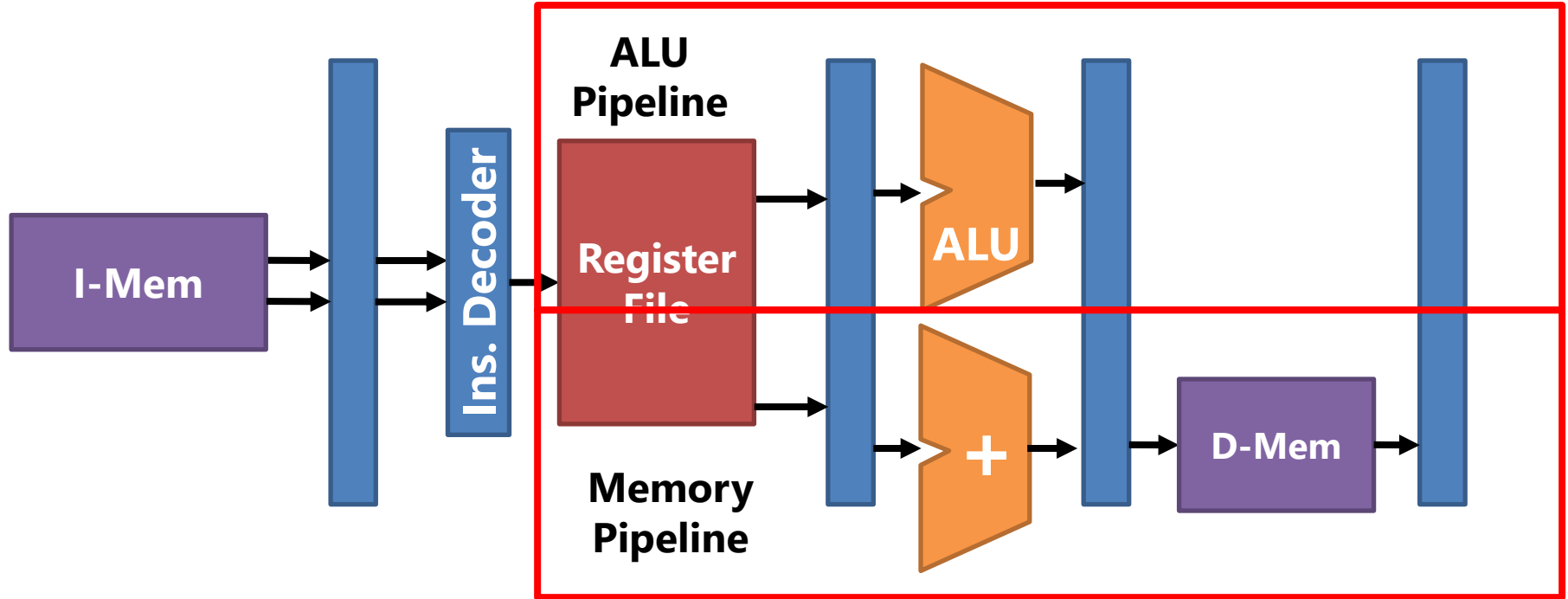
- How about if we **fetch two** instructions each cycle?
 - Maybe, fetch one ALU instruction and one load/store instruction

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

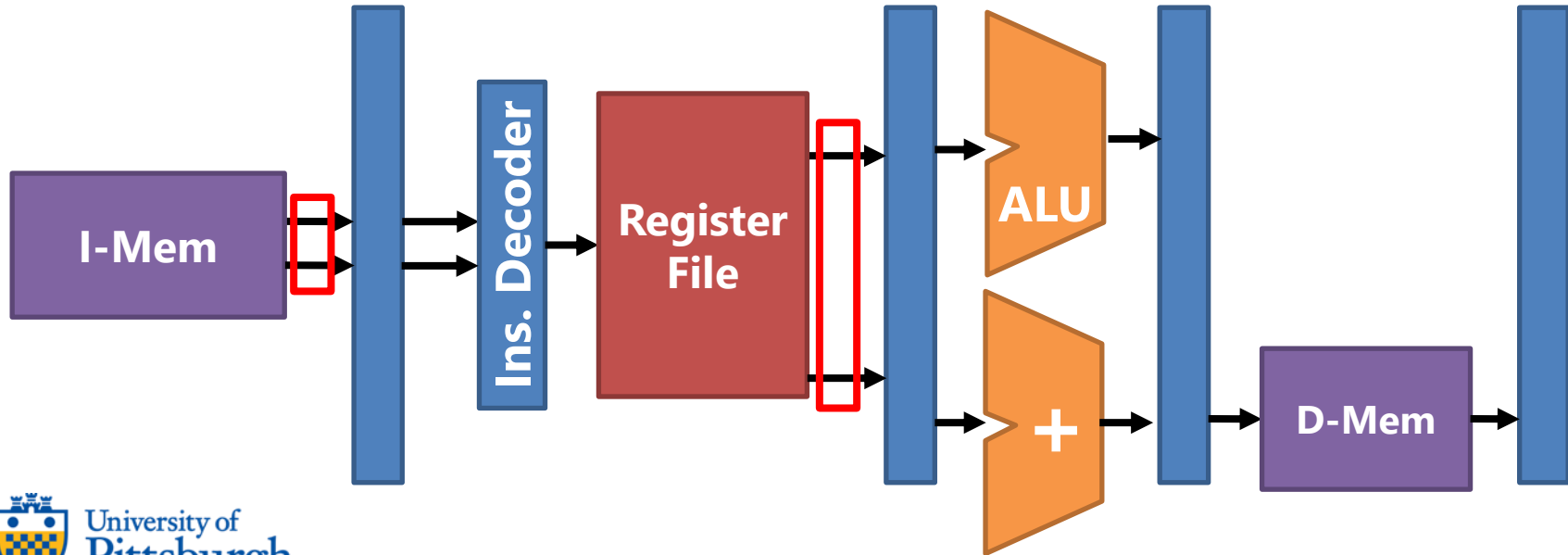
- One pipeline for ALU/Branches and one for loads and stores



- Now new opportunities for structural hazards abound!

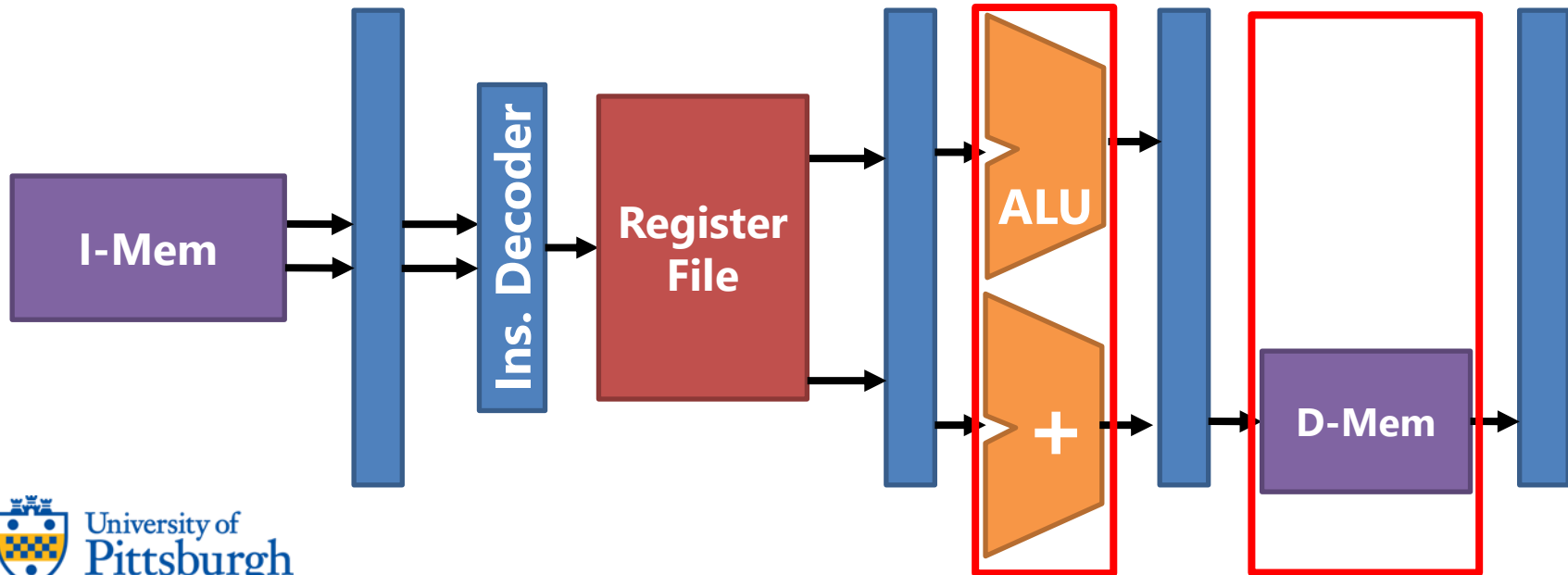
Structural Hazard in Storage Locations

- 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 writeback to register file at WB stage
→ 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!

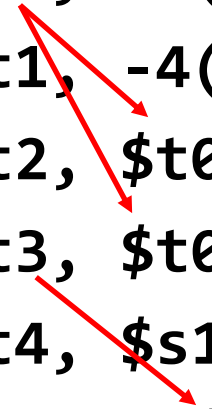
```
lw    $t0, 0($s1)
lw    $t1, -4($s1)
addi  $t2, $t0, -8
add   $t3, $t0, $s1
add   $t4, $s1, $s1
sw    $t5, 8($t3)
sw    $t6, 4($s1)
```

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, \$t0, -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?

- Have two full ALUs, have MEM units at both pipelines ...
 - That way, we can avoid those structural hazards in the first place
 - But that means more transistors and importantly more power
- Most processors have specialized pipelines for different instructions
 - Integer ALU pipeline, FP ALU pipeline, Load/Store pipeline, ...
 - With smart scheduling, can still achieve high utilization
- Who does the scheduling? Well, we talked about this already ...
 - Compiler: Static scheduling
 - Processor: Dynamic scheduling

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 static 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)`

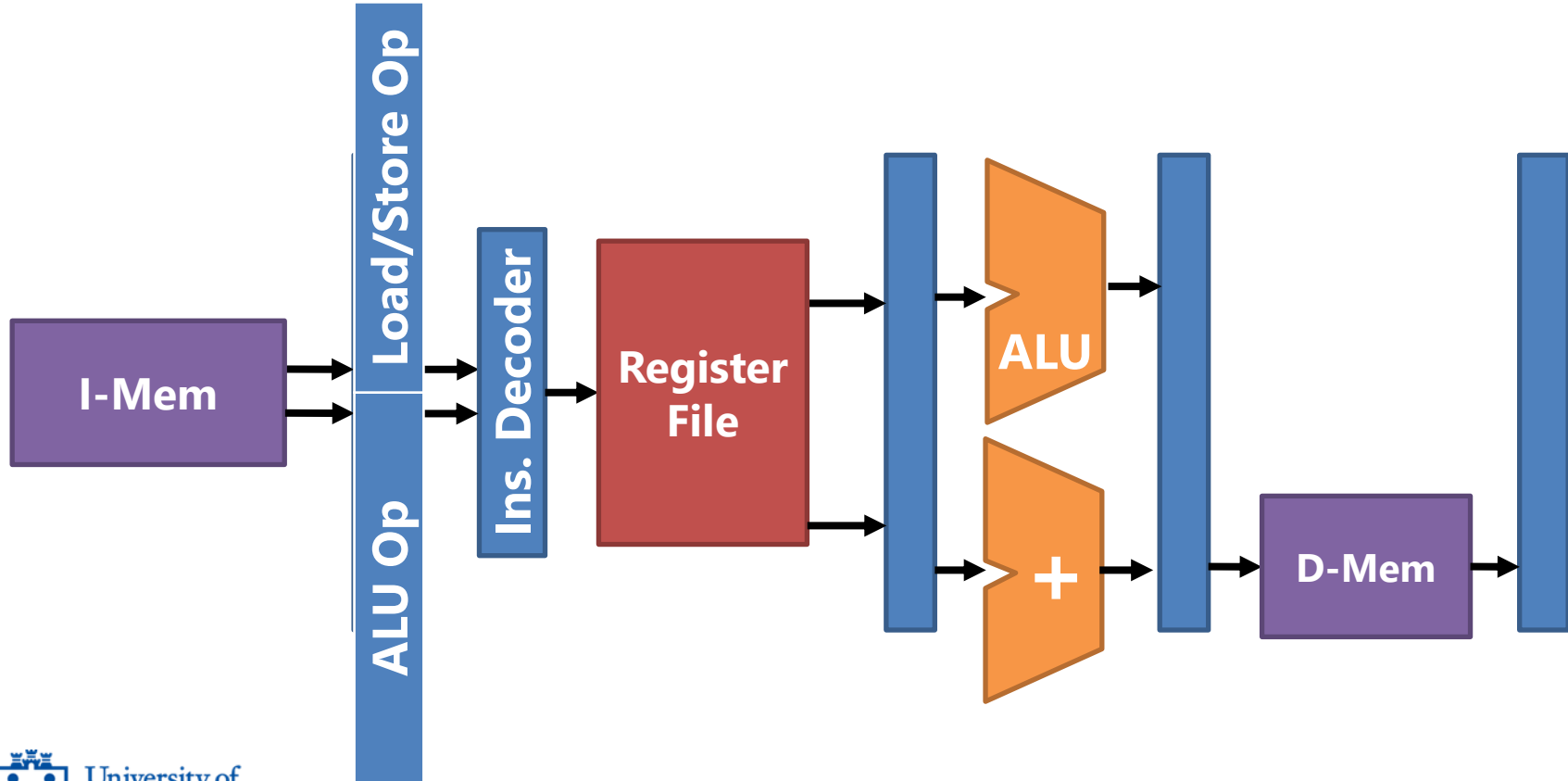
- Or another example could be:

`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, \$t0, -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		lw t0
2	addi t2	lw t1
3	add t3	
4	add t4	sw t5
5		sw t6

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

VLIW instruction encoding (optimized)

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

lw \$t0, 0(\$s1)

addi \$t2, \$t0, -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		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 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 suffer data hazard stalls
 - 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 instructions 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 VLIW binary to new 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
 - You can even cache the translated binary for later use

Success of VLIW depends on the Compiler

- Up to the compiler to create schedule with minimal nops
- Compiler tries to use all the tricks in the book
 - For data hazards: fill in nops by reordering instructions
 - For control hazards: use predicates to remove branches
- **Loops** are particularly challenging to the compiler. Why?
 - For data hazards, reordering is limited to within the loop
 - For tight loops, may not be enough instructions to fill nops
 - For control hazards, predicates can't be used on loop branches

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
```

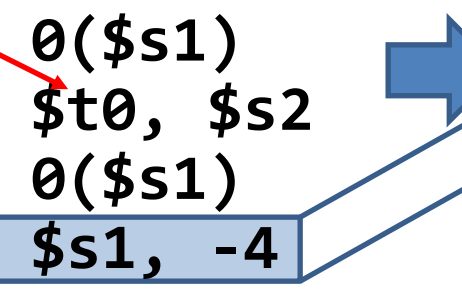
- It's basically iterating over an array adding **\$s2** to each element

Compiler Scheduling of a Loop

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

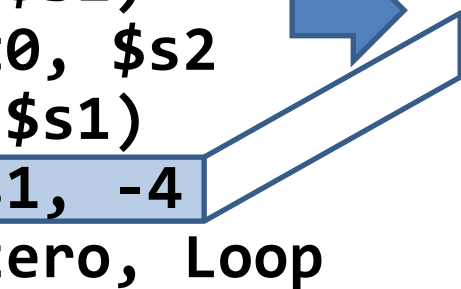
Loop:

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

- We can't compress the code further due to data hazards
- So 4 cycles for each loop iteration?
 - No! We need to now consider the control hazards.

Compiler Scheduling of a Loop

- The actual code that gets generated is the following:

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

- Why the extra **nops** at the end?
 - Processor needs **nops** to munch on while **bne** resolves at EX stage
 - That adds two extra cycles before **addi** can be fetched
 - So 6 cycles per loop iteration. That's even worse than IPC = 1!

Compiler Scheduling of a Loop

- The actual code that gets generated is the following:

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

- Can't you use predicates to get rid of that loop back branch?
 - No. Predicates are a way of avoiding branches altogether.
 - Useful in an if-then-else context where branching can be skipped.
 - But for a loop, you **need to jump** to re-execute the loop body.

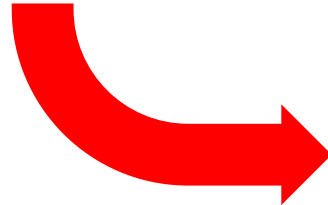
Loop unrolling

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 in loop to hide data hazard bubbles
 - Less frequent loop back branches causing control hazard bubbles

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    $t0, 4($s1)
sw    $t1, 0($s1)
bne   $s1, $zero, Loop
```

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

Let's try unrolling our example code

- 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    $t0, 4($s1)
sw    $t1, 0($s1)
bne   $s1, $zero, Loop
```

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

- Now we spend 7 cycles for 2 iterations of the loop
 - Much better than the previous 6 cycles for 1 iteration!

How far to unroll?

- The previous example doubled the code in the loop. Of course we can unroll 3X, 4X, 8X... what are the tradeoffs?
- **Space vs. time** is the big one.
 - But memory today is big, network connections are fast... is this so much of a problem?
 - Well.....
- **Caching** is the big bottleneck these days.
 - The bigger the code is, **the less of it will fit in the cache.**
 - This is bad, as we'll see.