

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 $\text{CPI} = 1$, right?
 - How can an instruction be executed in less than a cycle?

Wide Issue Processors

From CPI to IPC

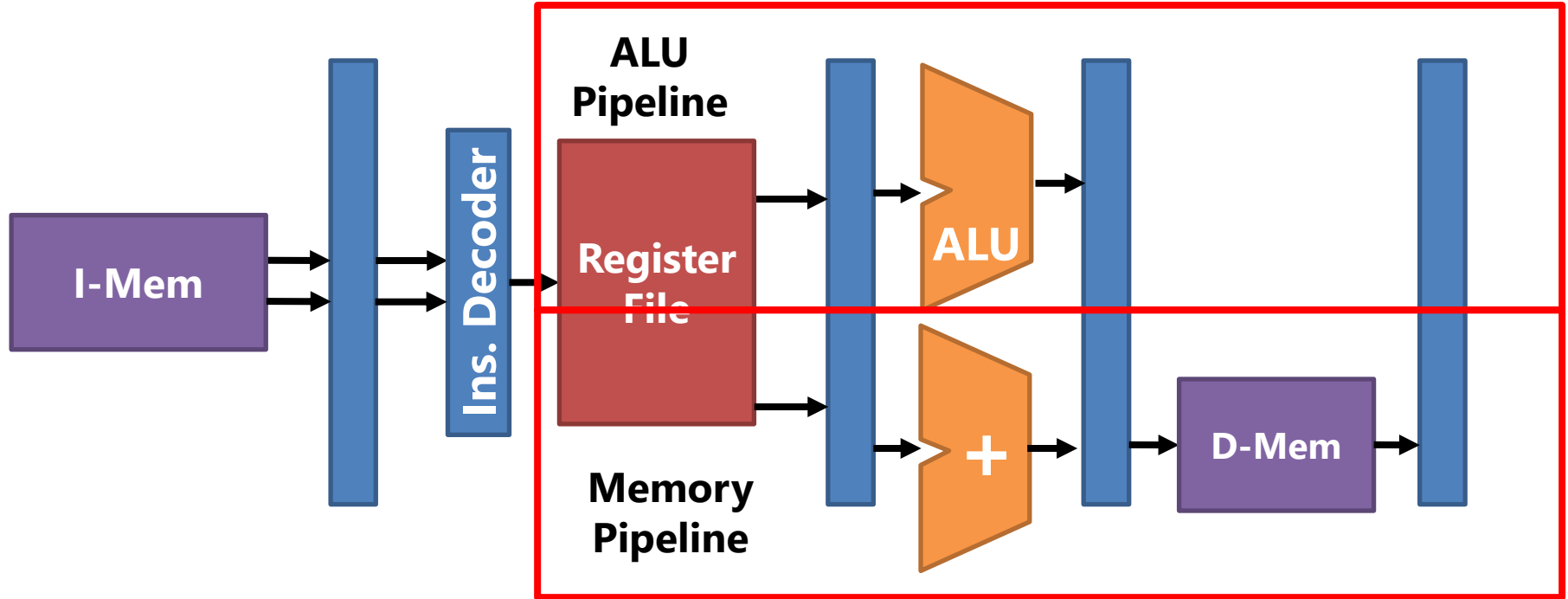
- How about if we **fetch two** instructions each cycle?
 - Maybe, fetch one ALU instruction and one load/store instruction

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

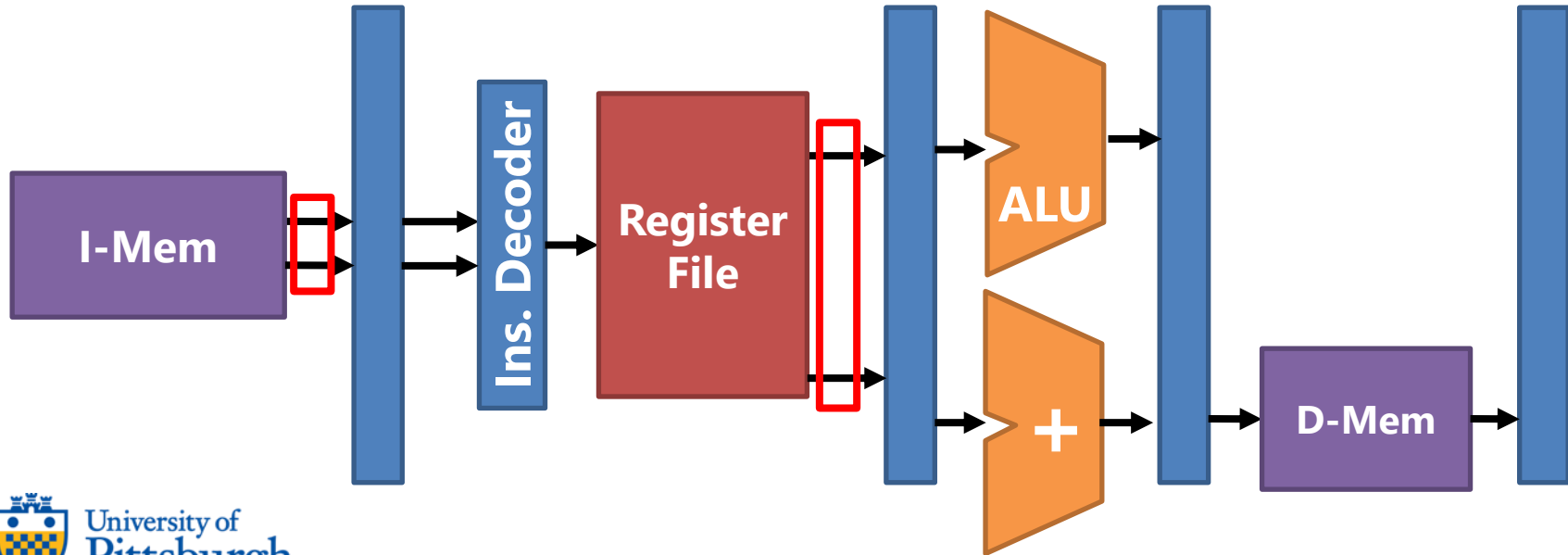
- One pipeline for ALU/Branches and one for loads and stores



- Now new opportunities for structural hazards abound!

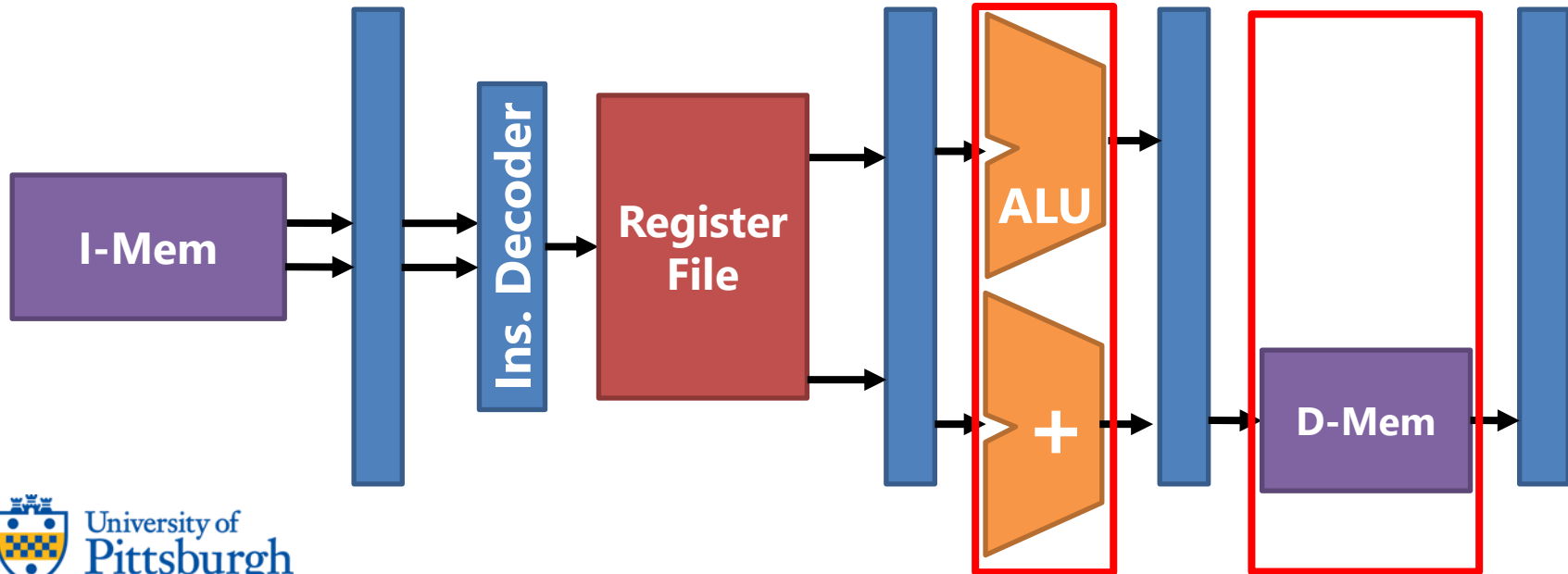
Structural Hazard in Storage Locations

- 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 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)`

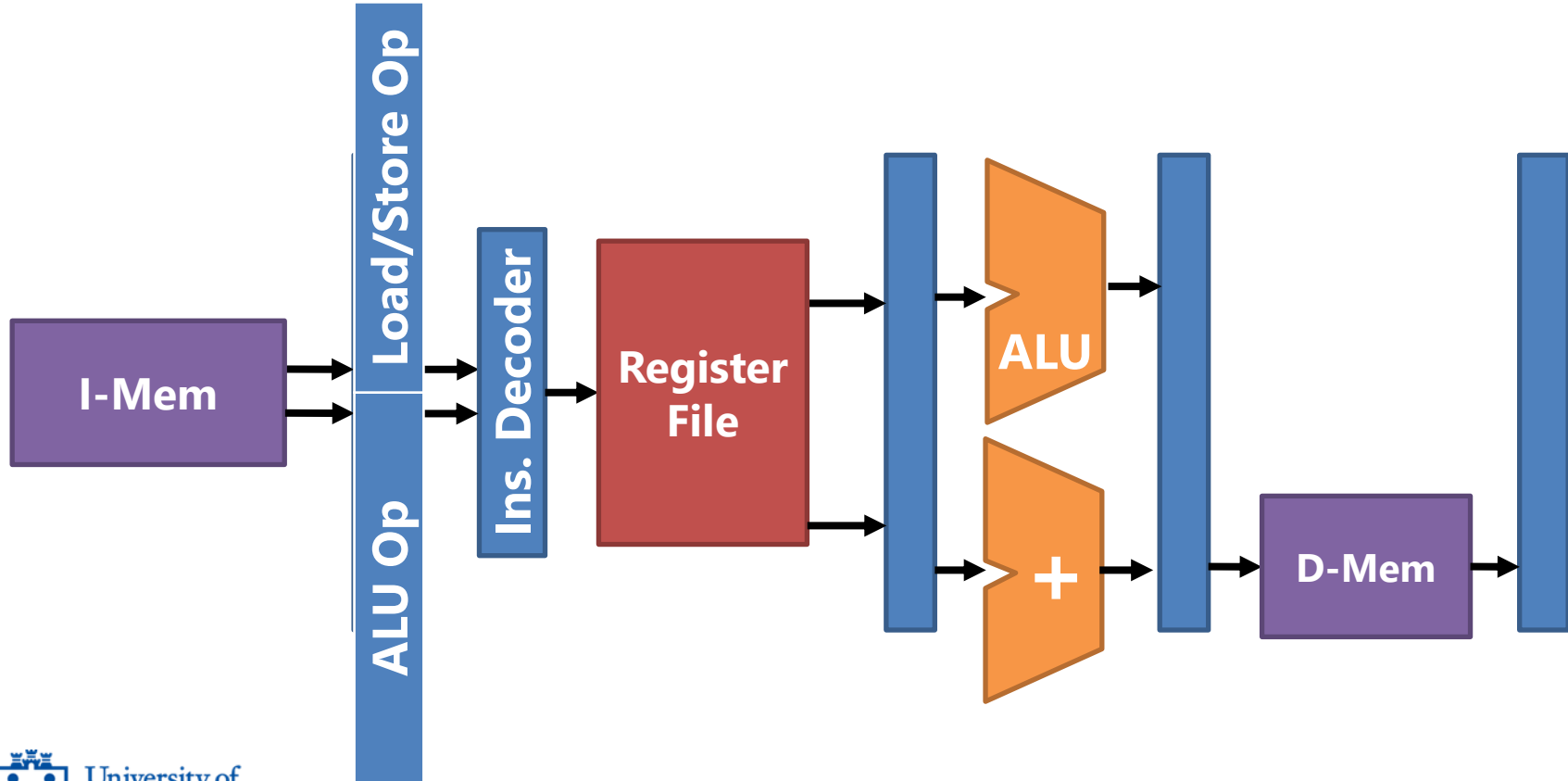
- Or another example could be:

`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, \$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 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 can run 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 instructions
- Use of predicates can be a big help in finding useful instructions
 - Control dependencies (as well as data deps) prevent reordering
 - A big block of straight-line code is ideal for reordering
 - Predicates can convert if-then-else code into straight-line code
 - Then-branch and else-branch can even be in one VLIW instruction!
- **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

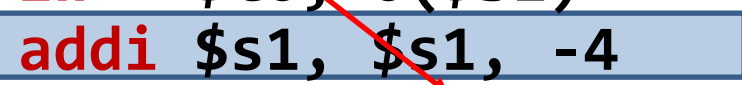
Loop:

```
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:	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 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!

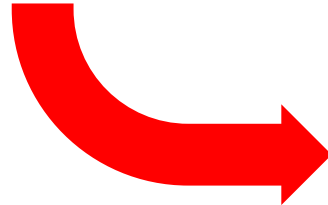
Loop unrolling

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
 - But 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:	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)
	bne \$s1, \$0, 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!

Let's try unrolling our example code 4X

- 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
	bne \$s1, \$0, 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 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:	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

	bne \$s1, \$0, 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

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)