Roadmap

- Resolving hazards
  - Data hazards: forwarding
  - Stalling the pipeline
  - Control hazards: prediction
- Advanced ILP techniques
  - Static approaches
    - Reordering, multiple issue, loop unrolling, …
  - Dynamic scheduling and speculation
- Concluding remarks

Instruction-Level Parallelism (ILP)

- Pipelining: executing multiple instructions in parallel
- More approaches to increase ILP
  - Deeper pipeline
    - Less work per stage ⇒ shorter clock cycle
    - Potential speedup (number of pipeline stages) increases
  - Multiple issue
    - Add more hardware to build multiple pipelines
    - Start multiple instructions per clock cycle
    - Dependencies enforce restrictions
  - Reordering instructions to cover pipeline stalls
    - Dependencies enforce restrictions

Data Dependences

- Flow dependences - read after write

```
x := 4 + a;
…
y := x + 1;
```

- Anti-dependences - write after read

```
y := x + 1;
…
x := 4 + a;
```

- Output dependences - write after write

```
x := 4 + a;
…
x := y + 1;
```

- The latter two can be eliminated by renaming
  - No value flows (also called name dependences)
  - Flow dependences are also called true dependences

Basic rule: If there is data dependence between two instructions, we must keep the execution order.

Roadmap

- Resolving hazards
  - Data hazards: forwarding
  - Stalling the pipeline
  - Control hazards: prediction
- Advanced ILP techniques
  - Static approaches ←
    - Reordering, multiple issue, loop unrolling, …
  - Dynamic scheduling and speculation
- Concluding remarks

Code Reordering to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction

```
I0: lw  $t1, 0($t0)
I1: lw  $t2, 4($t0)
I2: add $t3, $t1, $t2
I3: sw  $t3, 12($t0)
I4: lw  $t4, 8($t0)
I5: add $t5, $t1, $t4
I6: sw  $t5, 16($t0)
```

(reg, I_producer, I_consumer)

  - First: identify true dependences (reordering constraints)
    - ($t1, I0, I2), ($t1, I0, I5), ($t2, I1, I2), ($t3, I2, I3), ($t4, I4, I5), ($t5, I5, I6)

Code Reordering to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction

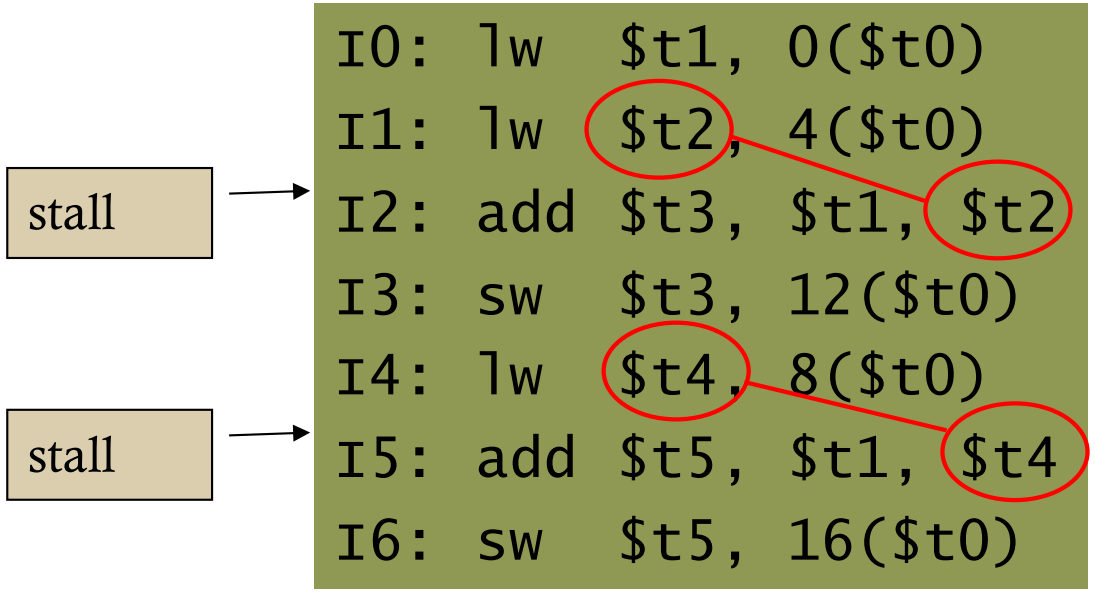
($t1, I0, I2), ($t2, I1, I2),
($t3, I2, I3), ($t4, I4, I5),
($t5, I5, I6), ($t1, I0, I5)

```
I0: lw  $t1, 0($t0)
I1: lw  $t2, 4($t0)
I2: add $t3, $t1, $t2
I3: sw  $t3, 12($t0)
I4: lw  $t4, 8($t0)
I5: add $t5, $t1, $t4
I6: sw  $t5, 16($t0)
```

  - First: identify true dependences
  - Then: identify load-use hazards (assume forwarding will take care of all other data hazards)
    - E.g. sw needs additional support to accept forwarded $rt

Code Reordering to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction

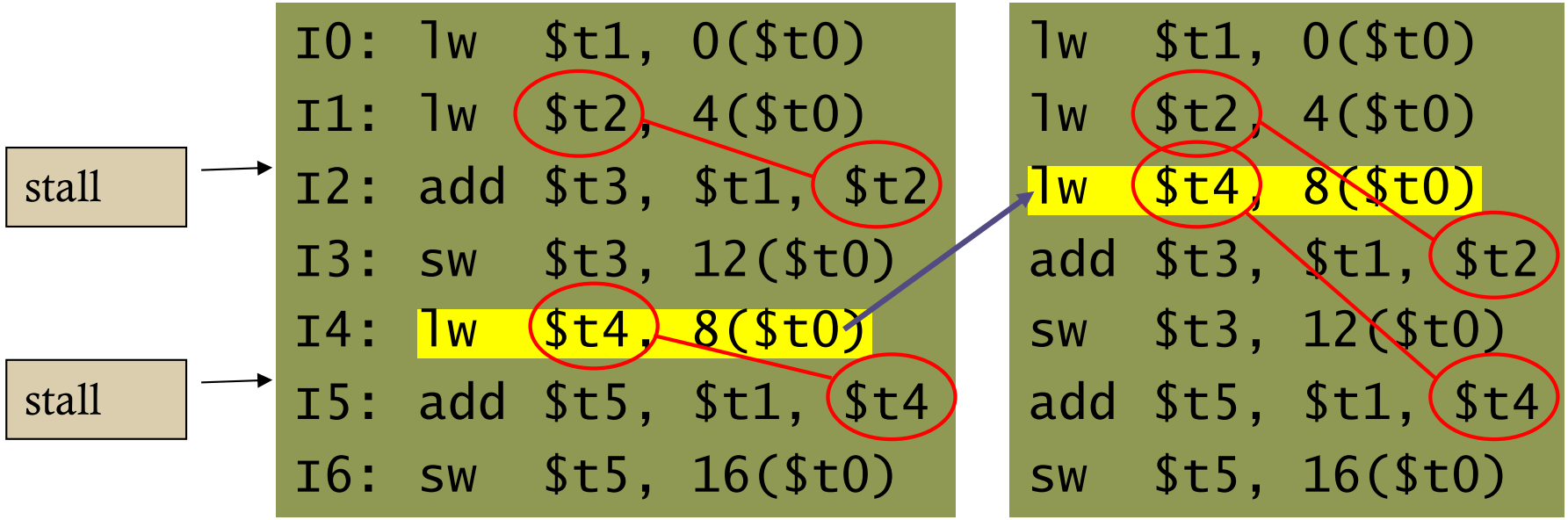


($t1, I0, I2), ($t2, I1, I2),
($t3, I2, I3), ($t4, I4, I5),
($t5, I5, I6), ($t1, I0, I5)

13 cycles to complete: 9 cycles to start last sw; 4 more cycles to finish

Code Reordering to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction



13 cycles to complete: 9 cycles to start last sw; 4 more cycles to finish

11 cycles to complete: start one instruction @ every cycle

Multiple Issue

- Replicate hardware so that we can start multiple instructions in pipeline at each cycle
  - Hardware overhead
  - More instructions executed in parallel ➔ more complicated hazard detection / solution
- The issue step can be implemented statically or dynamically
  - Static multiple issue: compiler groups instructions to be issued together
    - Compiler detects and avoids hazards
  - Dynamic multiple issue
    - CPU examines instruction stream and chooses instructions to issue for each cycle
    - CPU resolves hazards using advanced techniques at runtime

Static Multiple Issue

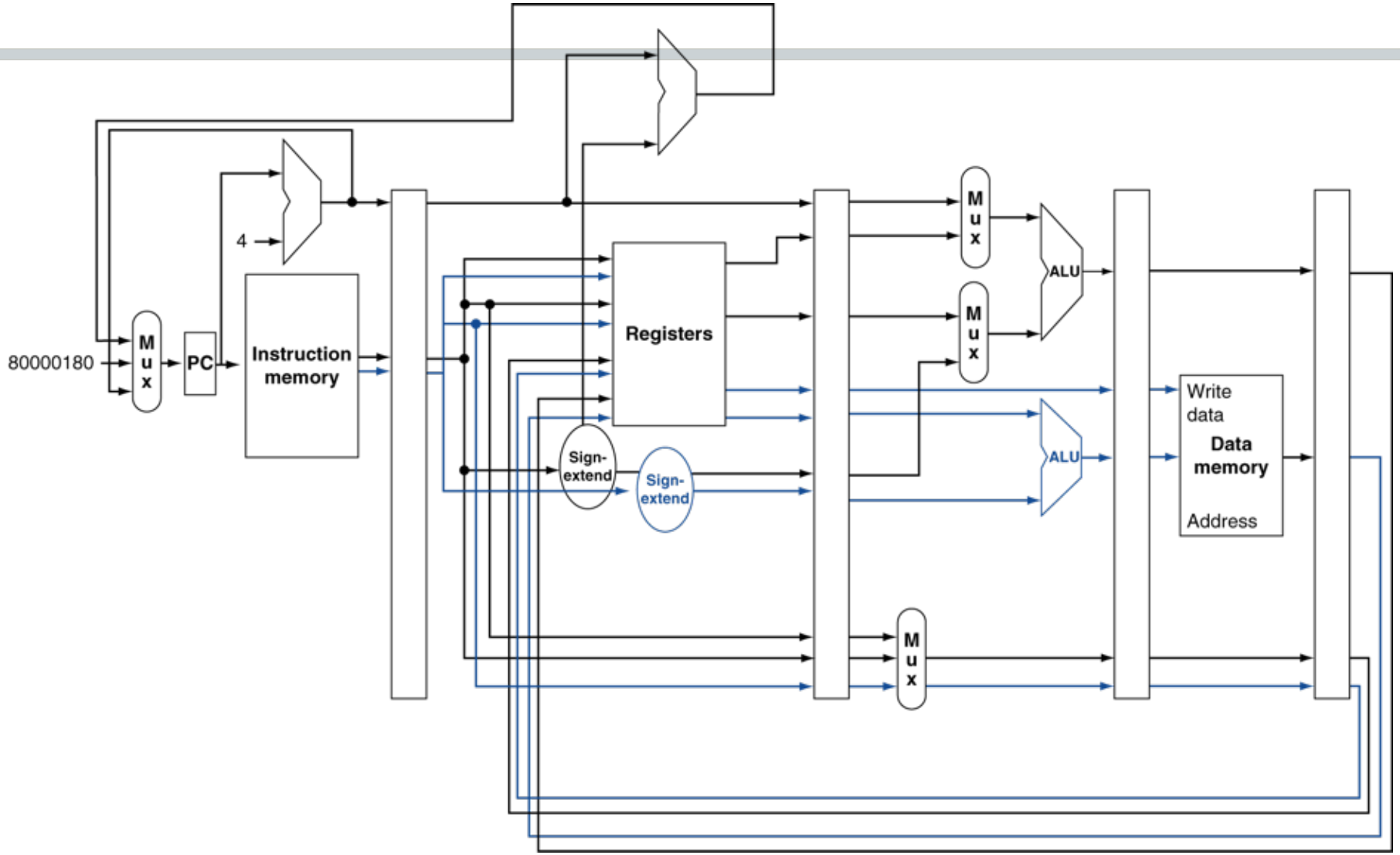
- Compiler groups instructions into issue packets
  - Fixed number of slots
    - Pad with nop if necessary
  - Fill in as many slots as possible
    - Reordering instructions when needed
- Constraints
  - The size of the "candidate window"
    - ILP within basic blocks is limited
    - Loop-level parallelism: parallelism among different loop iterations
  - Must preserve original program meaning
    - Dependences prevent reordering arbitrarily

MIPS with Static Dual Issue

- Two-issue packets example
  - One ALU/branch instruction
  - One load/store instruction
  - 64-bit aligned
    - ALU/branch, then load/store
    - Pad an unused instruction slot with nop

Address	Instruction type	Pipeline Stages						
n	ALU/branch	IF	ID	EX	MEM	WB		
n + 4	Load/store	IF	ID	EX	MEM	WB		
n + 8	ALU/branch		IF	ID	EX	MEM	WB	
n + 12	Load/store		IF	ID	EX	MEM	WB	
n + 16	ALU/branch			IF	ID	EX	MEM	WB
n + 20	Load/store			IF	ID	EX	MEM	WB

MIPS with Static Dual Issue



Scheduling Example

- Schedule this for dual-issue MIPS

```
Loop: lw   $t0, 0($s1)       # $t0=array element
      addu $t0, $t0, $s2     # add scalar in $s2
      sw   $t0, 0($s1)       # store result
      addi $s1, $s1,-4       # decrement pointer
      bne  $s1, $zero, Loop  # branch $s1!=0
```

	ALU/branch	Load/store	cycle
Loop:	nop	lw $t0, 0($s1)	1
	addi $s1, $s1,-4	nop	2
	addu $t0, $t0, $s2	nop	3
	bne $s1, $zero, Loop	sw $t0, 4($s1)	4

- IPC = 5/4 = 1.25 (c.f. peak IPC = 2)

Roadmap

- Resolving hazards
  - Data hazards: forwarding
  - Stalling the pipeline
  - Control hazards: prediction
- Advanced ILP techniques
  - Static approaches
    - Reordering, multiple issue, loop unrolling, …
  - Dynamic scheduling and speculation
- Concluding remarks

Loop Unrolling

- Replicate loop body and adjust the increment of loop variable and loop body
  - Use different registers per replication - "register renaming"
- Effects
  - Loop overhead is reduced
  - Larger loop body: more ILP/ scheduling freedom
  - Register usage within the loop body increased

```
do i=1 to n by 1
    a[i] = a[i]*s
end
```

⇒

```
do i=1 to n by 2
    a[i] = a[i]*s
    a[i+1] = a[i+1]*s
end
```

Additional code needed at the end if n is not even

Loop Unrolling Example

- Original loop

```
Loop: lw   $t0, 0($s1)       # $t0=array element
      addu $t0, $t0, $s2     # add scalar in $s2
      sw   $t0, 0($s1)       # store result
      addi $s1, $s1,-4       # decrement pointer
      bne  $s1, $zero, Loop  # branch $s1!=0
```

- 1: make four copies of the loop body

Loop Unrolling Example

```
Loop:   lw   $t0, 0($s1)       # $t0=array element
        addu $t0, $t0, $s2     # add scalar in $s2
        sw   $t0, 0($s1)       # store result
        addi $s1, $s1,-4       # decrement pointer
        bne  $s1, $zero, Loop # branch $s1!=0

        lw   $t0, 0($s1)       # $t0=array element
        addu $t0, $t0, $s2     # add scalar in $s2
        sw   $t0, 0($s1)       # store result
        addi $s1, $s1,-4       # decrement pointer
        bne  $s1, $zero, Loop # branch $s1!=0

        lw   $t0, 0($s1)       # $t0=array element
        addu $t0, $t0, $s2     # add scalar in $s2
        sw   $t0, 0($s1)       # store result
        addi $s1, $s1,-4       # decrement pointer
        bne  $s1, $zero, Loop # branch $s1!=0

        lw   $t0, 0($s1)       # $t0=array element
        addu $t0, $t0, $s2     # add scalar in $s2
        sw   $t0, 0($s1)       # store result
        addi $s1, $s1,-4       # decrement pointer
        bne  $s1, $zero, Loop # branch $s1!=0
```

- 2: drop unnecessary updating and checking of loop variable – only need to keep one checking/updating

Loop Unrolling Example

```
Loop:   lw   $t0, 0($s1)     # $t0=array element
        addu $t0, $t0, $s2   # add scalar in $s2
        sw   $t0, 0($s1)     # store result

        lw   $t0, 0($s1)     # $t0=array element
        addu $t0, $t0, $s2   # add scalar in $s2
        sw   $t0, 0($s1)     # store result

        lw   $t0, 0($s1)     # $t0=array element
        addu $t0, $t0, $s2   # add scalar in $s2
        sw   $t0, 0($s1)     # store result

        lw   $t0, 0($s1)     # $t0=array element
        addu $t0, $t0, $s2   # add scalar in $s2
        sw   $t0, 0($s1)     # store result
        addi $s1, $s1,-16    # decrement pointer
        bne  $s1, $zero, Loop # branch $s1!=0
```

- 3: offsets needs to be adjusted for different iterations

Loop Unrolling Example

```
Loop:   lw   $t0, 0($s1)      # $t0=array element
        addu $t0, $t0, $s2    # add scalar in $s2
        sw   $t0, 0($s1)      # store result

        lw   $t0, -4($s1)     # $t0=array element
        addu $t0, $t0, $s2    # add scalar in $s2
        sw   $t0, -4($s1)     # store result

        lw   $t0, -8($s1)     # $t0=array element
        addu $t0, $t0, $s2    # add scalar in $s2
        sw   $t0, -8($s1)     # store result

        lw   $t0, -12($s1)    # $t0=array element
        addu $t0, $t0, $s2    # add scalar in $s2
        sw   $t0, -12($s1)    # store result
        addi $s1, $s1,-16     # decrement pointer
        bne  $s1, $zero, Loop # branch $s1!=0
```

Dependences?

- 4: rename registers to remove anti- and output dependences

Loop Unrolling Example

```
Loop:   lw   $t0, 0($s1)      # $t0=array element
        addu $t0, $t0, $s2    # add scalar in $s2
        sw   $t0, 0($s1)      # store result

        lw   $t1, -4($s1)     # $t1=array element
        addu $t1, $t1,0 $s2   # add scalar in $s2
        sw   $t1, -4($s1)     # store result

        lw   $t2, -8($s1)     # $t2=array element
        addu $t2, $t2, $s2    # add scalar in $s2
        sw   $t2, -8($s1)     # store result

        lw   $t3, -12($s1)    # $t3=array element
        addu $t3, $t3, $s2    # add scalar in $s2
        sw   $t3, -12($s1)    # store result
        addi $s1, $s1,-16     # decrement pointer
        bne  $s1, $zero, Loop # branch $s1!=0
```

- 5: ready to schedule!

Scheduling Unrolled Loop

	ALU/branch	Load/store	cycle
Loop:	addi $s1, $s1,-16	lw $t0, 0($s1)	1
	nop	lw $t1, 12($s1)	2
	addu $t0, $t0, $s2	lw $t2, 8($s1)	3
	addu $t1, $t1, $s2	lw $t3, 4($s1)	4
	addu $t2, $t2, $s2	sw $t0, 16($s1)	5
	addu $t3, $t4, $s2	sw $t1, 12($s1)	6
	nop	sw $t2, 8($s1)	7
	bne $s1, $zero, Loop	sw $t3, 4($s1)	8

- IPC = 14/8 = 1.75
  - Closer to 2, but at cost of registers and code size