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

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)

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

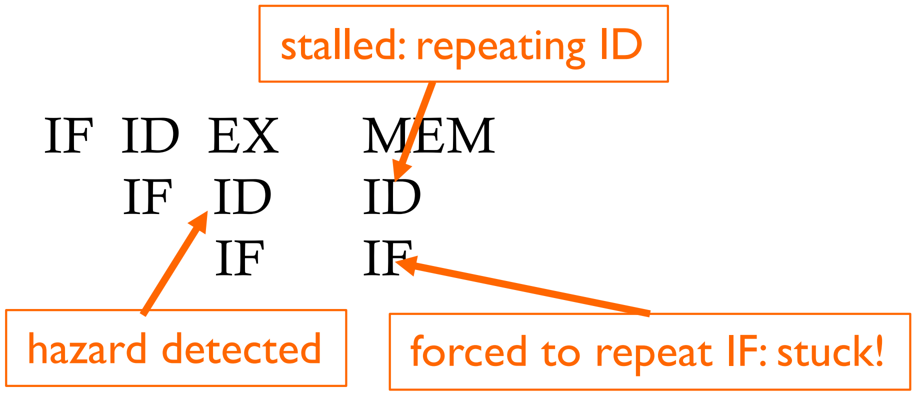
Outline

- 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

Dynamic Scheduling

- Key idea: determine which instructions are ready and allow independent instructions behind stall to proceed

```
lw      $t0, 20($s2)
addu    $t1, $t0, $t2
sub     $s4, $s4, $t3
slti    $t5, $s4, 20
```

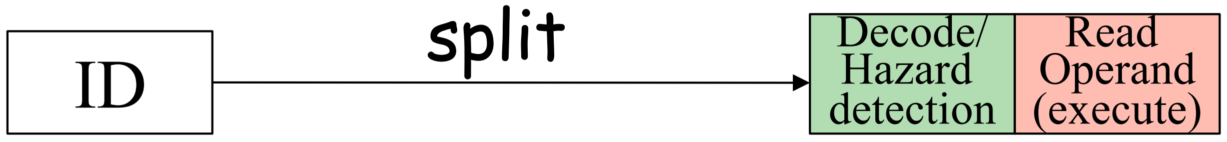


Better if we can start sub while addu is waiting for lw – How?

- Simple pipeline
  - Issue stage: ID (instruction decode)
    - Decoding / check hazards
    - Read operands

Dynamic Scheduling

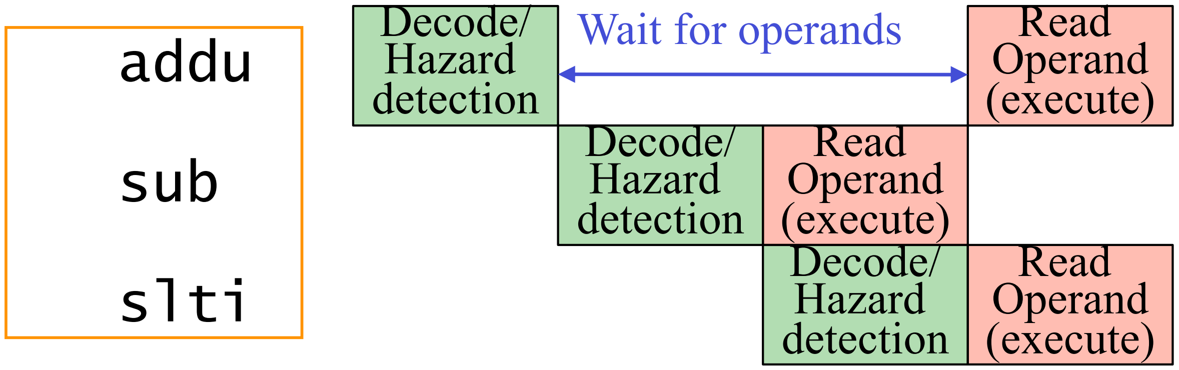
- Split the ID pipe stage of simple 5-stage pipeline into 2 stages:
  - *Issue* — decode instructions, check for hazards (no delay needed)
  - *Read operands* —wait until no data hazards, then read operands (execution can start immediately after operands reading)



- Enables *in-order issue*, *out-of-order execution* and allows *out-of-order completion*

Splitting Issue & Execution

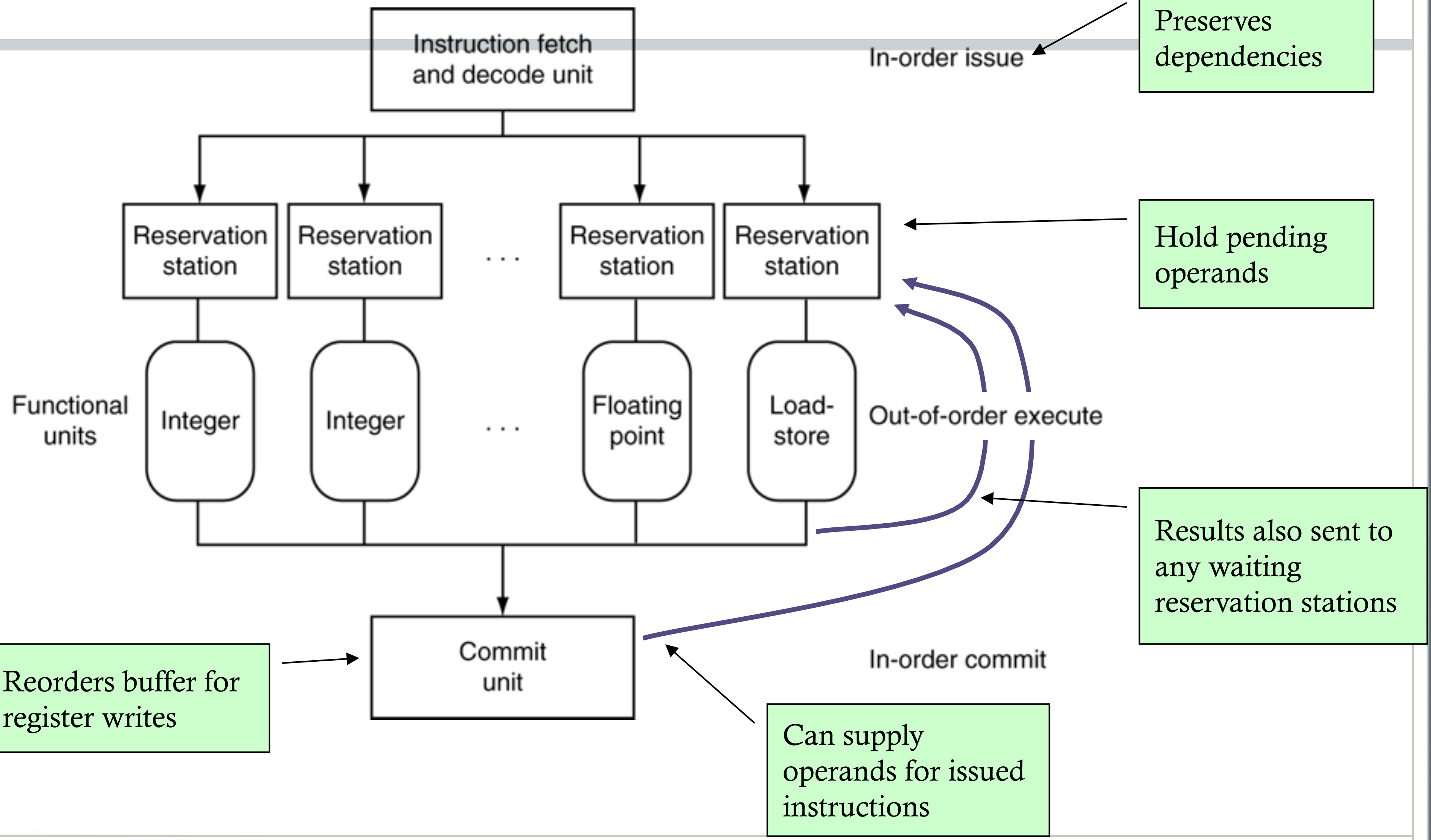
```
lw    $t0, 20($s2)
addu  $t1, $t0, $t2
sub   $s4, $s4, $t3
slti  $t5, $s4, 20
```



- Need a buffer for instructions which are waiting for the operand(s)

- Need to make sure to forward values to instructions waiting for them

Dynamically Scheduled CPU



Optional: Tomasulo Algorithm

- Hardware implementation of dynamic scheduling
  - First proposed by Tomasulo with IBM
  - Adopted by many processors including Intel x86
- http://en.wikipedia.org/wiki/Tomasulo_algorithm
- http://courses.cs.washington.edu/courses/csep548/06au/lectures/tomasulo.pdf

Why Dynamic Scheduling?

- Why not just let the compiler schedule code?
- Not all stalls are predicable
  - e.g., cache misses
- Can’t always schedule around branches
  - Branch outcome is dynamically determined
- Different implementations of an ISA have different latencies and hazards

Speculation for Greater ILP

- "Guess" what to do with an instruction
  - Start operation as soon as possible
  - Check whether guess was right
    - If so, complete the operation
    - If not, roll-back and do the right thing
- Common to static and dynamic multiple issue
- Examples
  - Speculate on branch outcome
    - Roll back if path taken is different
  - Speculate on load
    - Roll back if location is updated

Meltdown and Spectre

- Vulnerabilities in modern computers
  - Reported Jan. 2018
  - Affect a wide range of systems
    - Intel x86, IBM POWER, ARM
    - iOS, Linux, macOS, Windows
  - Attacking processors that support dynamic scheduling / speculated execution

Example

```
t = a+b
u = t+c
v = u+d
if v:
    w = kern_mem[address]
    # if we get here, fault
    x = w&0x100
    y = user_mem[x]
```

Original sequence

```
#speculative execution attempted
t, w_ = a+b, kern_mem[address]
u, x_ = t+c, w_&0x100
v, y_ = u+d, user_mem[x_]
if v:
    # fault
    w, x, y = w_, x_, y_ # we never get here
```

Possible shuffled sequence for an out-of-order, dual-issue, speculative processor

- If we think the branch might be taken

Example

- Feels safe
  - Either v is zero,
  - Or kernel access fault
- Race condition in reality
  - Memory accessed and cached before privilege checking completed

```
t, w_ = a+b, kern_mem[address]
u, x_ = t+c, w_&0x100
v, y_ = u+d, user_mem[x_]
if v:
    # fault
    w, x, y = w_, x_, y_ # we never get here
```

- Side-channel / timing attack
  - Check cache ➔ get to know which location has been fetched from memory for y_
  - ➔get the value of x_
  - ➔deduct one bit of w_, which is from kernel space!

References

- http://en.wikipedia.org/wiki/Meltdown_(security_vulnerability)
- http://en.wikipedia.org/wiki/Spectre_(security_vulnerability)
- http://meltdownattack.com/
- http://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html
- http://www.raspberrypi.org/blog/why-raspberry-pi-isnt-vulnerable-to-spectre-or-meltdown/

Does Advanced ILP Work?

The BIG Picture

- Helped a lot but now comes with diminishing returns
- Programs have real dependencies that limit ILP
- Some dependencies are hard to eliminate
  - e.g., pointer aliasing
- Some parallelism is hard to expose
  - Limited window size during instruction issue
- Memory delays and limited bandwidth
  - Hard to keep pipelines full
- Speculation can help if done well

Power Efficiency

- Complexity of dynamic scheduling and speculations requires power
- Multiple simpler cores may be better

Microprocessor	Year	Clock Rate	Pipeline Stages	Issue width	Out-of-order/ Speculation	Cores	Power
i486	1989	25MHz	5	1	No	1	5W
Pentium	1993	66MHz	5	2	No	1	10W
Pentium Pro	1997	200MHz	10	3	Yes	1	29W
P4 Willamette	2001	2000MHz	22	3	Yes	1	75W
P4 Prescott	2004	3600MHz	31	3	Yes	1	103W
Core	2006	2930MHz	14	4	Yes	2	75W
UltraSparc III	2003	1950MHz	14	4	No	1	90W
UltraSparc T1	2005	1200MHz	6	1	No	8	70W

Concluding Remarks

- ISA influences design of datapath and control
- Datapath and control influence design of ISA
- Pipelining improves instruction throughput using parallelism
  - More instructions completed per second
  - Latency for each instruction not reduced
- Hazards: structural, data, control
- Multiple issue and dynamic scheduling (ILP)
  - Dependencies limit achievable parallelism
  - Complexity leads to the power wall