Superscalar Architectures: Part 1

Multiple-Issue Processors

Lecture 3.1
August 23rd, 2017

Jae W. Lee (jaewlee@snu.ac.kr)
Computer Science and Engineering
Seoul National University

Download this lecture slides at https://goo.gl/rJPMQU

Slide credits: [COD5e] and [CA:AQA5e] slides from Elsevier Inc.

Outline

Reference: [COD5e] 4.10

- **Instruction-Level Parallelism**
- **Static Multiple Issue**
- **Dynamic Multiple Issue**
- **Summary**

Instruction-Level Parallelism (ILP)

- **Pipelining: executing multiple instructions in parallel**
  - Thus, increasing instruction-level parallelism (ILP)

- **To further increase ILP**
  - Deeper pipeline
    - Less work per stage ⇒ shorter clock cycle
  - Multiple issue
    - Replicate pipeline stages ⇒ multiple pipelines
    - Start multiple instructions per clock cycle
    - E.g., 4GHz 4-way multiple-issue
      - 16 BIPS, peak CPI (Cycles Per Inst.) = 0.25, peak IPC (Inst. Per Second) = 4
    - But dependencies reduce this in practice
  - We will explore multiple-issue designs in this lecture

Instruction-Level Parallelism (ILP)

- **Static multiple issue**
  - Compiler groups instructions to be issued together
  - Packages them into “issue slots”
  - Compiler detects and avoids hazards
- **Dynamic multiple issue**
  - CPU examines instruction stream and chooses instructions to issue each cycle
  - Compiler can help by reordering instructions
  - CPU resolves hazards using advanced techniques at runtime
- **Also known as “superscalar” processors**
  - That is, IPC > 1

Static Multiple Issue (1)

- **Compiler groups instructions into "issue packets"**
  - Group of instructions that can be issued on a single cycle
  - Determined by pipeline resources required

- **Very Long Instruction Word (VLIW)**
  - Think of an issue packet as a very long instruction
  - Specifies multiple concurrent operations

Static Multiple Issue (2)

- **Scheduling Static Multiple Issue**
  - Compiler must remove some/all hazards
    - Reorder instructions into issue packets
    - No dependencies with a packet
    - Possibly some dependencies between packets
      - Varies between ISAs; compiler must know!
    - Pad with nop (dummy instruction) if necessary

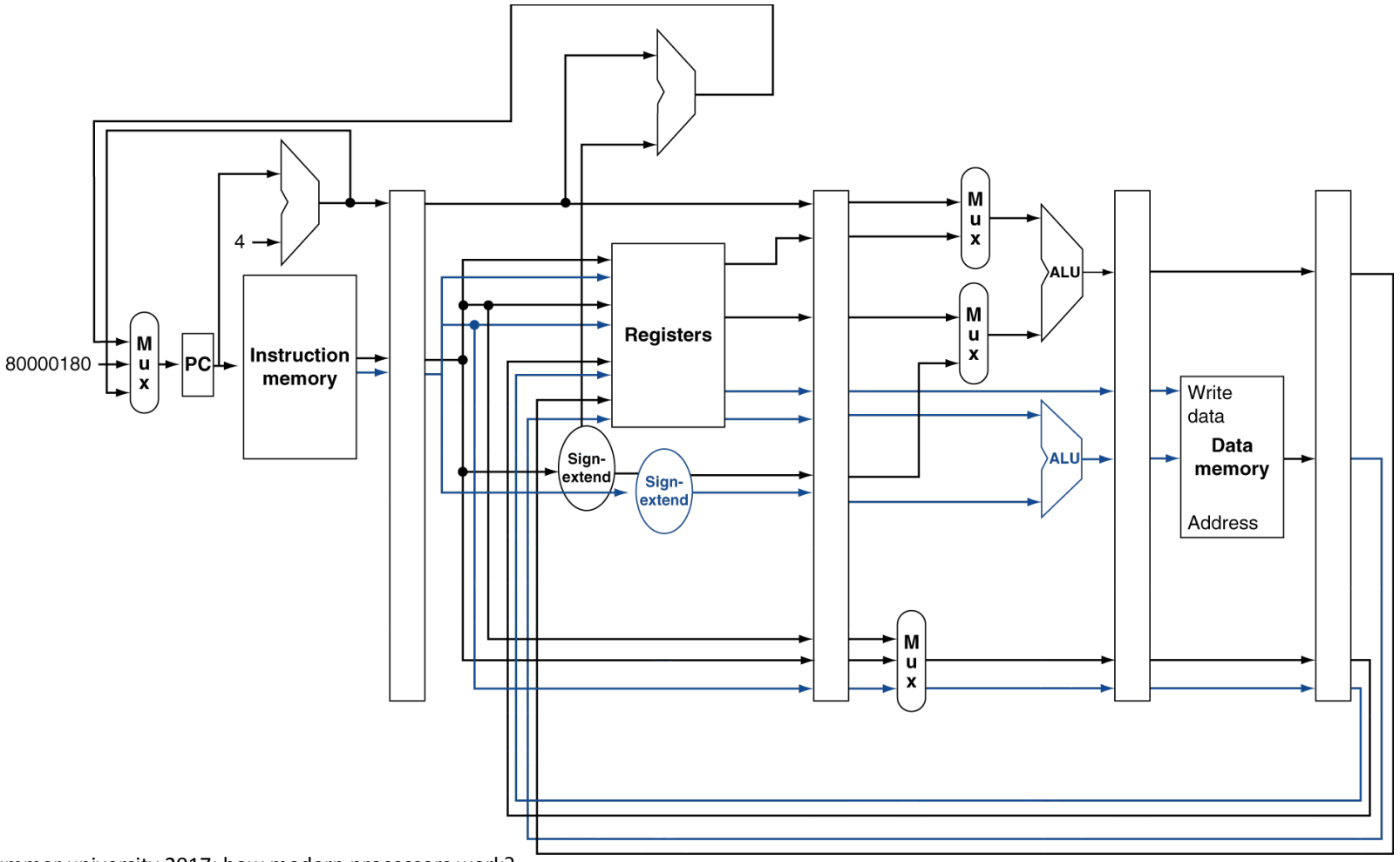
Static Multiple Issue (3)

- **Example: MIPS with static dual issue**
  - One ALU/branch instruction
  - One load/store instruction
  - 64-bit aligned
    - ALU/branch, then load/store
    - Pad an unused instruction 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

Static Multiple Issue (4)

- **Example: MIPS with static dual issue - Pipeline structure**



Static Multiple Issue (5)

- **Example: MIPS with static dual issue – Hazards**
  - More instructions executing in parallel
  - EX data hazard
    - Forwarding avoided stalls with single-issue
    - Now can't use ALU result in load/store in same packet
      - Split into two packets, effectively a stall

```
add   $t0, $s0, $s1
lw    $s2, 0($t0)
```

  - Load-use hazard
    - Still one cycle use latency, but now two instructions
  - More aggressive scheduling required

Static Multiple Issue (6)

- **Example: MIPS with static dual issue - Scheduling 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
```

	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)**

Static Multiple Issue (7)

- **Example: MIPS with static dual issue - Loop unrolling**
  - Replicate loop body to expose more parallelism
    - Reduces loop-control overhead
  - Use different registers per replication
    - Called “register renaming”
    - Avoid loop-carried “anti-dependencies”
      - Store followed by a load of the same register
      - Also known as “name dependence”: Reuse of a register name

Static Multiple Issue (8)

- **Example: MIPS with static dual issue – Loop unrolling example**

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

```
Loop: lw   $t0, 0($s1)
      addu $t0, $t0, $s2
      sw   $t0, 0($s1)
      lw   $t0, -4($s1)
      addu $t0, $t0, $s2
      sw   $t0, -4($s1)
      lw   $t0, -8($s1)
      addu $t0, $t0, $s2
      sw   $t0, -8($s1)
      lw   $t0, -12($s1)
      addu $t0, $t0, $s2
      sw   $t0, -12($s1)
      addi $s1, $s1, -16
      bne  $s1, $zero, Loop
```

Static Multiple Issue (9)

- **Example: MIPS with static dual issue – unrolled loop scheduling**

	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

Dynamic Multiple Issue (1)

- **CPU decides whether to issue 0, 1, 2, … instructions each cycle**
  - Avoiding structural and data hazards
- **Avoids the need for compiler scheduling**
  - Though it may still help
  - Code semantics preserved by the CPU (software-transparent)

Dynamic Multiple Issue (2)

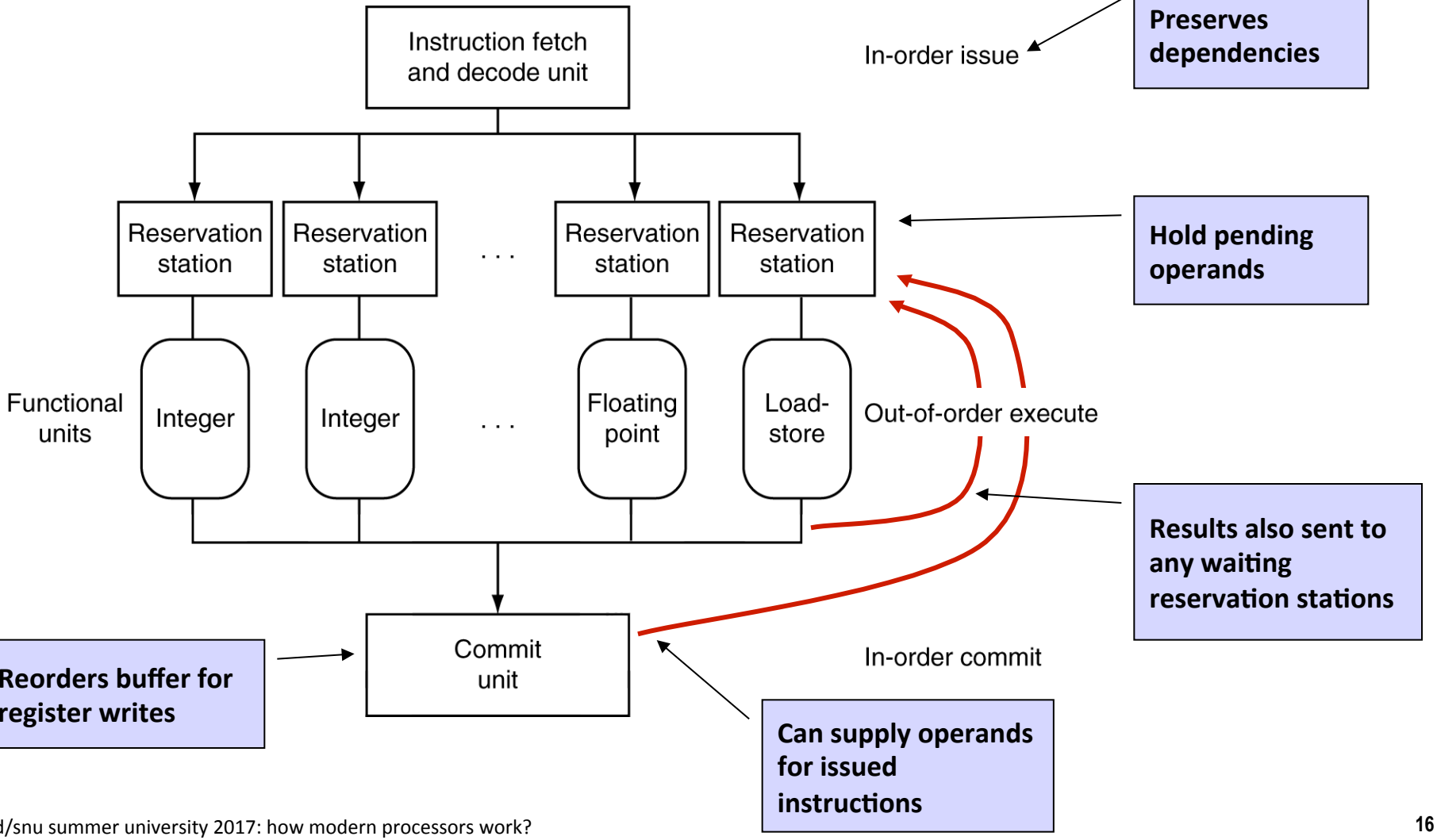
- **Dynamic Pipeline Scheduling**
  - Allow the CPU to execute instructions out of order to avoid stalls
    - But commit result to registers in order
- **Example**

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

  - Can start `sub` while `addu` is waiting for `lw`

Dynamic Multiple Issue (3)

- **Dynamically Scheduled CPU**



Dynamic Multiple Issue (4)

- **Register Renaming**
  - Reservation stations and reorder buffer effectively provide register renaming
  - On instruction issue to reservation station
    - If operand is available in register file or reorder buffer
      - Copied to reservation station
      - No longer required in the register; can be overwritten
    - If operand is not yet available
      - It will be provided to the reservation station by a function unit
      - Register update may not be required

Dynamic Multiple Issue (5)

- **Speculation**
  - Predict branch and continue issuing
    - Don’t commit until branch outcome determined
  - Load speculation
    - Avoid load and cache miss delay
      - Predict the effective address
      - Predict loaded value
      - Load before completing outstanding stores
      - Bypass stored values to load unit
    - Don’t commit load until speculation cleared

Dynamic Multiple Issue (6)

- **Why Do 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

Dynamic Multiple Issue (7)

- **Does Multiple Issue Work?**
  - Yes, but not as much as we'd like
  - 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

Summary

- **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