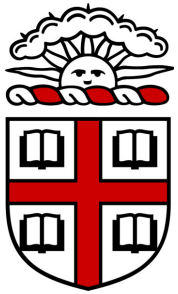
EN164: Design of Computing Systems
Topic 06: VLIW Processor Design

Professor Sherief Reda
http://scale.engin.brown.edu
Electrical Sciences and Computer Engineering
School of Engineering
Brown University
Spring 2012



[material from Patterson & Hennessy, 4th ed]

Instruction Level Parallelism

- Pipelining: executing multiple instructions in parallel
- To increase ILP
  - Deeper pipeline: more multiple instructions
    - Less work per stage ⇒ shorter clock cycle
  - Multiple issue
    - Replicate pipeline stages ⇒ multiple pipelines
    - Start multiple instructions per clock cycle
    - CPI < 1, or Instructions Per Cycle (IPC) > 1
    - E.g., 4GHz 4-way multiple-issue
      - 16 BIPS, peak CPI = 0.25, peak IPC = 4
    - But dependencies reduce this in practice

Where can performance be improved?

$$\text{CPU Time} = \text{CPU Clock Cycles} \times \text{Clock Cycle Time}$$

$$= \frac{\text{Instructions}}{\text{Program}} \times \boxed{\frac{\text{Clock cycles}}{\text{Instruction}}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

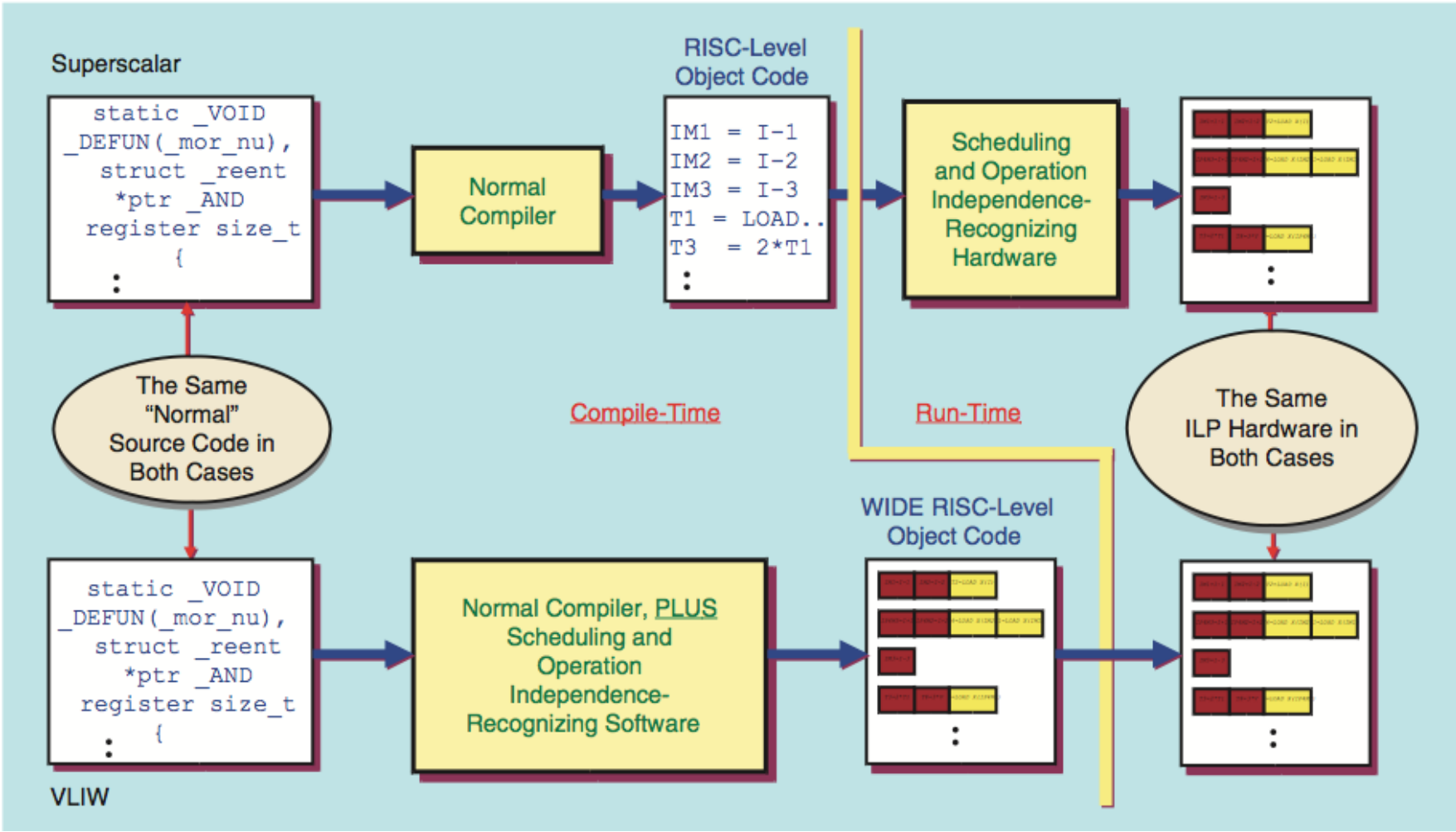
optimization goal for multiple-issue processors

- Pipeline CPI = Ideal pipeline CPI + Structural Stalls + Data Hazard Stalls + Control Stalls

Processor design for multiple issue

- VLIW:
  - Compiler groups instructions to be issued together Very Large Instruction Words (VLIW)
  - Packages them statically into “issue slots”
  - Compiler detects and avoids hazards
- Superscalar:
  - 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
  - Can be dynamic or static

Difference between superscalar and VLIW



[from Fisher *et al.*]

Static multiple issue

- Compiler groups instructions into “issue packets”
  - Group of instructions that can be issued on a single cycle
  - Determined by pipeline resources required
- Think of an issue packet as a very long instruction
  - Specifies multiple concurrent operations
  - ⇒ Very Long Instruction Word (VLIW)

Scheduling static multiple issue

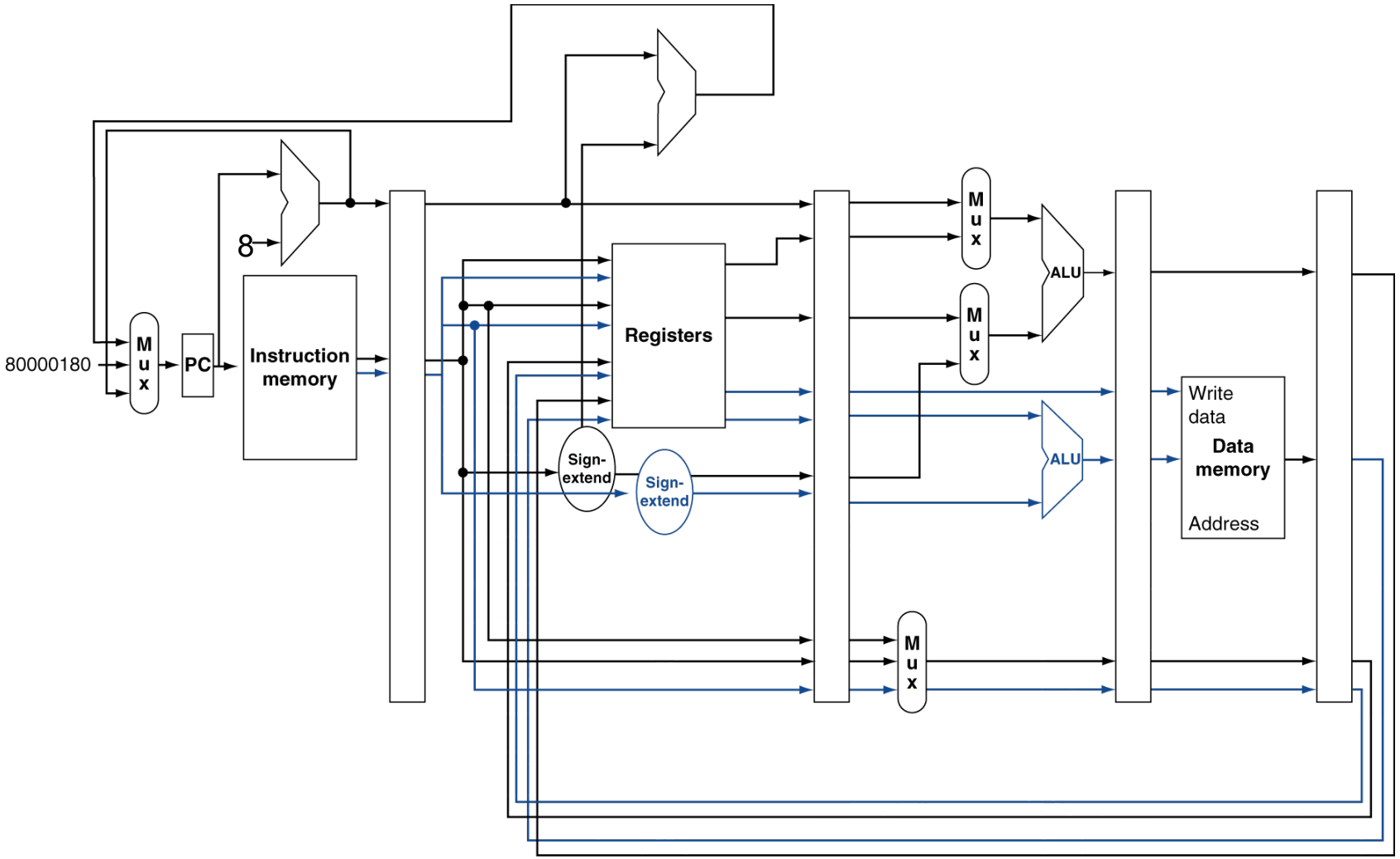
- Compiler must remove some/all hazards
  - Reorder instructions into issue packets
  - No dependencies with a packet
  - Possibly some dependencies between packets
    - Varies between ISAs; compiler must know!
  - Pad with nop if necessary

MIPS with static dual issue

- Two-issue packets
  - 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

Pipeline design for multiple issue



Hazards in dual-issue MIPS

- 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
    - ```
      add  $t0, $s0, $s1
      load $s2, 0($t0)
      ```
    - Split into two packets, effectively a stall
- Load-use hazard
  - Still one cycle use latency, but now two instructions
- More aggressive scheduling required
      ```

Scheduling example for dual-issue MIPS

- ***Schedule*** this code for dual-issue MIPS

```
Loop: lw   $t0, 0($s1)       # $t0=array element
      add  $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
	nop	nop	2
	add $t0, $t0, $s2	nop	3
	addi $s1, $s1,-4	sw $t0, 0($s1)	4
	bne $s1, $zero, Loop	nop	5

- IPC = 5/5 = 1 (c.f. peak dual-issue IPC = 2 and single-issue IPC = 5/6 = 0.83 for single-issue pipeline)

Limits to ILP: data dependencies

- Data dependencies determine:
  - Order which results should be computed
  - Possibility of hazards
  - Degree of freedom in scheduling instructions

  => limit to ILP

- Data/Name dependency hazards:
  - Read After Write (RAW)
  - Write After Read (WAR)
  - Write After Write (WAW)

Data dependency: RAW

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

```
add     $s2, $s1, $s0
sub     $s4, $s2, $s3
```

```
sw      $s2, 0($t0)
….
lw      $s1, 100($t1)
```

- A true data dependency because values are transmitted between the instructions
- Dependency is clear when registers are involved – less obvious when memory is involved. Alias analysis is required for memory

Name dependency (antidependence): WAR

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

```
add     $s4, $s2, $s0
…..
sub     $s2, $s1, $s3
```

```
lw      $t2, 0($s2)
….
lw      $s2, 4($t0)
```

- Just a name dependency – no values being transmitted
- Dependency can be removed by renaming registers (either by compiler or HW)

Name dependency (output dependency): WAW

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

```
add     $s2, $s1, $s0
....
sub     $s2, $t2, $t3
```

- Just a name dependency – no values being transmitted
- Dependency can be removed by renaming registers (either by compiler or HW)

Impact of branches on data flow

- Data flow: actual flow of data values among instructions that produce results and those that consume them
  - branches make flow dynamic, determine which instruction is supplier of data
- Example:

```
add     $s2,$s1,$s0
beq     $s2, $t2, L
sub     $s2,$s3,$s4
L:  …
or      $s6,$s2,$s5
```

- `or` depends on `add` or `sub`? Must preserve data flow on execution.
- Willing to execute instructions that should not have been executed, thereby violating the control dependences, if can do so without affecting correctness of the program

Re-schedule example for dual-issue MIPS

- ***Re-Schedule*** this code for dual-issue MIPS

```
Loop: lw   $t0, 0($s1)       # $t0=array element
      add  $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
	add $t0, $t0, $s2	nop	3
	bne $s1, $zero, Loop	sw $t0, 4($s1)	4

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

Exposing ILP using 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
    - Aka “name dependence”
      - Reuse of a register name

Loop unrolling example

	ALU/branch	Load/store	cycle
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, $t2, $s2	sw $t0, 16($s1)	5
	add $t3, $t3, $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

Limits to loop unrolling

1. Decrease in amount of overhead amortized with each extra unrolling
2. Growth in code size (might not fit in instruction memory cache)
3. Register pressure: loop unrolling increase demands on registers and they are few of them to begin with.

Summary of VLIW architectures

- Advantages:
  - Simplified HW for management of hazards and scheduling (important for power and cost)
  - Works well in data-intensive applications with little control
- Disadvantages:
  - Some hazards can't be resolved during compile time
  - Poor portability and backward compatibility

- Found a niche in embedded market (e.g., DSPs)