

CS 61C: Great Ideas in Computer Architecture (Machine Structures)
Lecture 32: Pipeline Parallelism 3

Instructor: **Dan Garcia**
<http://inst.eecs.Berkeley.edu/~cs61c/sp13>


You Are Here!

Software


- Parallel Requests**
Assigned to computer
e.g., Search "Katz"
- Parallel Threads**
Assigned to core
e.g., Lookup, Ads
- Parallel Instructions**
>1 instruction @ one time
e.g., 5 pipelined instructions
- Parallel Data**
>1 data item @ one time
e.g., Add of 4 pairs of words
- Hardware descriptions**
All gates functioning in parallel at same time

Hardware

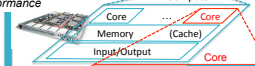
Warehouse Scale Computer



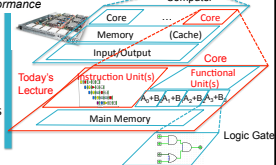
Smart Phone



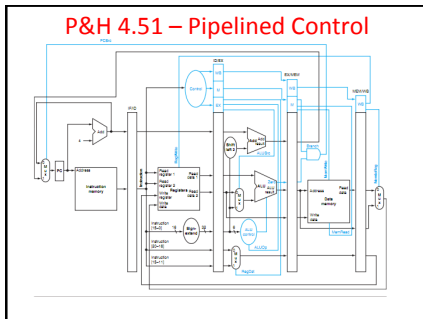
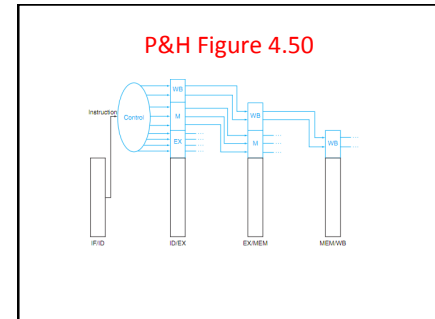
Computer



Today's Lecture



Harness Parallelism & Achieve High Performance



Hazards

Situations that prevent starting the next logical instruction in the next clock cycle

- Structural hazards**
 - Required resource is busy (e.g., roommate studying)
- Data hazard**
 - Need to wait for previous instruction to complete its data read/write (e.g., pair of socks in different loads)
- Control hazard**
 - Deciding on control action depends on previous instruction (e.g., how much detergent based on how clean prior load turns out)

Data Hazards

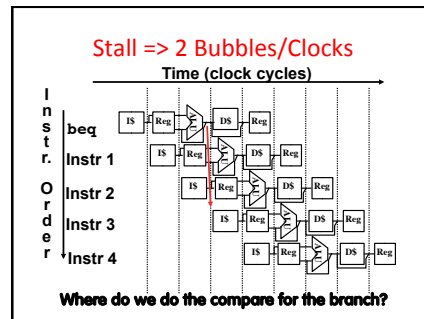
Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction
- C code for $A = B + E$; $C = B + F$;

	lw \$t1, 0(\$t0)	lw \$t1, 0(\$t0)
	lw (\$t2), 4(\$t0)	lw (\$t2), 4(\$t0)
stall	add \$t3, \$t1, (\$t2)	lw (\$t4), 8(\$t0)
	sw \$t3, 12(\$t0)	add \$t3, \$t1, (\$t2)
	lw (\$t4), 8(\$t0)	sw \$t3, 12(\$t0)
stall	add \$t5, \$t1, (\$t4)	add \$t5, \$t1, (\$t4)
	sw \$t5, 16(\$t0)	sw \$t5, 16(\$t0)
	13 cycles	11 cycles

3. Control Hazards

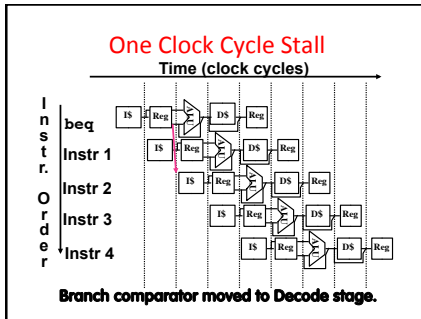
- Branch determines flow of control
 - Fetching next instruction depends on branch outcome
 - Pipeline can't always fetch correct instruction
 - Still working on ID stage of branch
- BEQ, BNE in MIPS pipeline
- Simple solution Option 1: **Stall** on every branch until have new PC value
 - Would add 2 bubbles/clock cycles for every Branch! (~ 20% of instructions executed)



Control Hazard: Branching

- Optimization #1:
 - Insert **special branch comparator** in Stage 2
 - As soon as instruction is decoded (Opcode identifies it as a branch), immediately make a decision and set the new value of the PC
 - Benefit: since branch is complete in Stage 2, only one unnecessary instruction is fetched, so only one no-op is needed
 - Side Note: means that branches are idle in Stages 3, 4 and 5

Question: What's an efficient way to implement the equality comparison?



Control Hazards: Branching

- Option 2: **Predict** outcome of a branch, fix up if guess wrong
 - Must cancel all instructions in pipeline that depended on guess that was wrong
 - This is called “**flushing**” the pipeline
- Simplest hardware if we predict that all branches are NOT taken
 - Why?

Control Hazards: Branching

- Option #3: Redefine branches
 - Old definition: if we take the branch, none of the instructions after the branch get executed by accident
 - New definition: whether or not we take the branch, the single instruction immediately following the branch gets executed (the **branch-delay slot**)
- Delayed Branch** means *we always execute inst after branch*
- This optimization is used with MIPS

Example: Nondelayed vs. Delayed Branch

<p>Nondelayed Branch</p> <pre> or \$8, \$9, \$10 add \$1, \$2, \$3 sub \$4, \$5, \$6 beq \$1, \$4, Exit xor \$10, \$1, \$11 </pre>	<p>Delayed Branch</p> <pre> add \$1, \$2, \$3 sub \$4, \$5, \$6 beq \$1, \$4, Exit or \$8, \$9, \$10 xor \$10, \$1, \$11 </pre>
---	--

Control Hazards: Branching

- Notes on **Branch-Delay Slot**
 - Worst-Case Scenario: put a no-op in the branch-delay slot
 - Better Case: place some instruction preceding the branch in the branch-delay slot—as long as the changed doesn't affect the logic of program
 - Re-ordering instructions is common way to speed up programs
 - Compiler usually finds such an instruction 50% of time
 - Jumps also have a delay slot ...

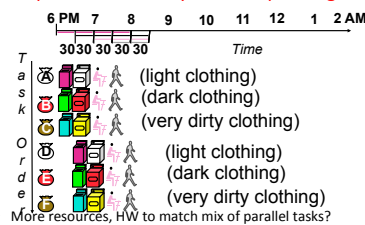
Greater Instruction-Level Parallelism (ILP)

- Deeper pipeline (5 => 10 => 15 stages)
 - Less work per stage => shorter clock cycle
- Multiple issue “superscalar”
 - Replicate pipeline stages => multiple pipelines
 - Start multiple instructions per clock cycle
 - CPI < 1, so use Instructions Per Cycle (IPC)
 - E.g., 4GHz 4-way multiple-issue
 - 16 BIPS, peak CPI = 0.25, peak IPC = 4
 - But dependencies reduce this in practice

Multiple Issue

- 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

Superscalar Laundry: Parallel per stage

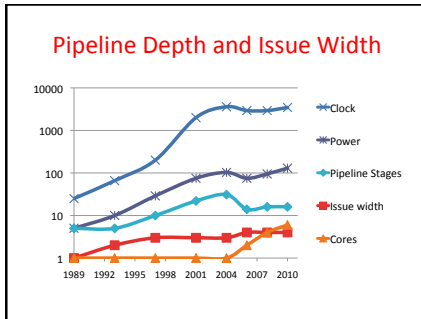


Pipeline Depth and Issue Width

- Intel Processors over Time

Microprocessor	Year	Clock Rate	Pipeline Stages	Issue width	Cores	Power
i486	1989	25 MHz	5	1	1	5W
Pentium	1993	66 MHz	5	2	1	10W
Pentium Pro	1997	200 MHz	10	3	1	29W
P4 Willamette	2001	2000 MHz	22	3	1	75W
P4 Prescott	2004	3600 MHz	31	3	1	103W
Core 2 Conroe	2006	2930 MHz	14	4	2	75W
Core 2 Yorkfield	2008	2930 MHz	16	4	4	95W
Core i7 Gulltown	2010	3460 MHz	16	4	6	130W

Chapter 4 – The Processor



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

Scheduling Static Multiple Issue

- Compiler must remove some/all hazards
 - Reorder instructions into issue packets
 - No dependencies **within** a packet
 - Possibly some dependencies between packets
 - Varies between ISAs; compiler must know!
 - Pad issue packet 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							
		IF	ID	EX	MEM	WB			
n	ALU/branch								
n + 4	Load/store								
n + 8	ALU/branch								
n + 12	Load/store								
n + 16	ALU/branch								
n + 20	Load/store								

Hazards in the Dual-Issue MIPS

- More instructions executing in parallel
 - 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

- 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:			1
			2
			3
			4

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 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
	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, \$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

Dynamic Multiple Issue

- “Superscalar” processors
- CPU decides whether to issue 0, 1, 2, ... each cycle
 - Avoiding structural and data hazards
- Avoids the need for compiler scheduling
 - Though it may still help
 - Code semantics ensured by the CPU

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
subu  $s4, $s4, $t3
slli  $t5, $s4, 20
```

 - Can start subu while addu is waiting for lw

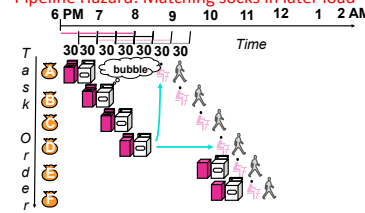
Why Do Dynamic Scheduling?

- Why not just let the compiler schedule code?
- Not all stalls are predictable
 - 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

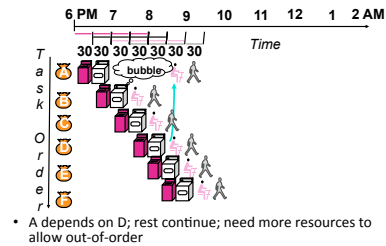
Speculation

- “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 (Branch Prediction)
 - Roll back if path taken is different
 - Speculate on load
 - Roll back if location is updated

Pipeline Hazard: Matching socks in later load



Out-of-Order Laundry: Don't Wait



Out Of Order Intel

- All use OOO since 2001

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	2600MHz	14	4	Yes	2	75W
Core 2 Yorkfield	2008	2900 MHz	16	4	Yes	4	95W
Core i7 Gulltown	2010	3400 MHz	16	4	Yes	6	130W

Does Multiple Issue Work?

The BIG Picture

- 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

“And in Conclusion..”

- Pipelining is an important form of ILP
- Challenge is (are?) hazards
 - Forwarding helps w/many data hazards
 - Delayed branch helps with control hazard in 5 stage pipeline
 - Load delay slot / interlock necessary
- More aggressive performance:
 - Longer pipelines
 - Superscalar
 - Out-of-order execution
 - Speculation