CS 61C:
Great Ideas in Computer Architecture
Instruction Level Parallelism: Multiple Instruction Issue

Instructors:
Krste Asanovic, Randy H. Katz
http://inst.eecs.Berkeley.edu/~cs61c/fa12

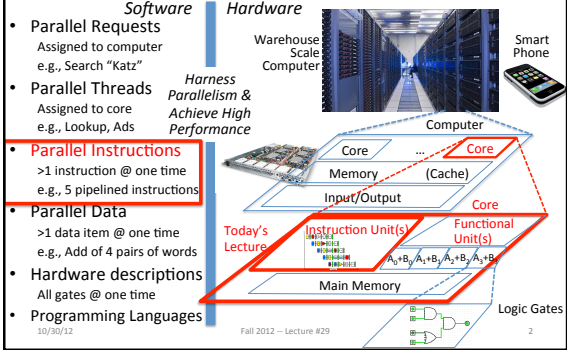
You Are Here!

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 @ one time
- Programming Languages

Hardware

Harness Parallelism & Achieve High Performance



Review

The BIG Picture

- Pipelining improves performance by increasing instruction throughput: exploits ILP
  - Executes multiple instructions in parallel
  - Each instruction has the same latency
- Subject to hazards
  - Structure, data, control
- Stalls reduce performance
  - But are required to get correct results
- Compiler can arrange code to avoid hazards and stalls
  - Requires knowledge of the pipeline structure

Agenda

- Higher Level ILP
- Administrivia
- Dynamic Scheduling
- And, in Conclusion, …

Agenda

- Higher Level ILP
- Administrivia
- Dynamic Scheduling
- And, in Conclusion, …

Greater Instruction-Level Parallelism (ILP)

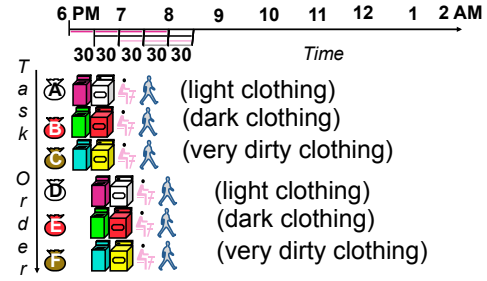
1. Deeper pipeline (5 => 10 => 15 stages)
   – Less work per stage ⇒ shorter clock cycle
2. Multiple issue *superscalar*
   – Replicate pipeline stages ⇒ multiple pipelines
   – Start multiple instructions per clock cycle

- CPI < 1, so can use Instructions Per Cycle (IPC)
  – E.g., 4 GHz 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



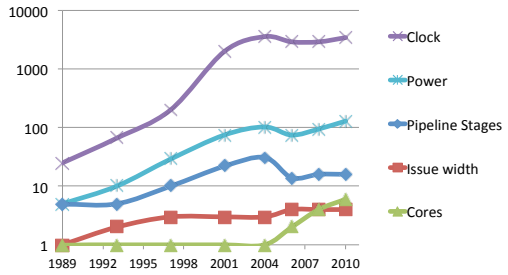
- More resources, HW to match mix of parallel tasks?

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 Gulftown	2010	3460 MHz	16	4	6	130W

Pipeline Depth and Issue Width



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
  - Called VLIW for Very Long Instruction Word

Scheduling Static Multiple Issue

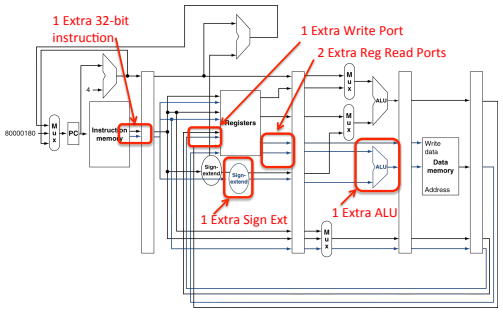
- Compiler must remove some/all hazards
  - Reorder instructions into issue packets
  - *No* dependencies with a packet
  - Possibly some dependencies between packets
    - Varies between ISAs; compiler must know!
  - Pad with nop if necessary

MIPS with Static Dual Issue

- Dual-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

Datapath with Static Dual Issue



Hazards in the Dual-Issue MIPS

- More instructions executing in parallel
- EX data hazard
  - Forwarding avoided stalls with single-issue
  - Now can't use ALU result for 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:	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 (vs. 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 but no real dependency between instructions

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, $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 more registers and bigger code

Agenda

- Higher Level ILP
- Administrivia
- Dynamic Scheduling
- And, in Conclusion, …

Administrivia

Agenda

- Higher Level ILP
- Administrivia
- Dynamic Scheduling
- And, in Conclusion, …

Dynamic Multiple Issue

- "Superscalar" processors
- CPU decides whether to issue 0, 1, 2, … instructions each cycle
  - Avoiding structural and data hazards
- Avoids 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
slti  $t5, $s4, 20
```

  - Can start `subu` while `addu` is waiting for `lw`
- Especially if cache misses, can execute many instructions

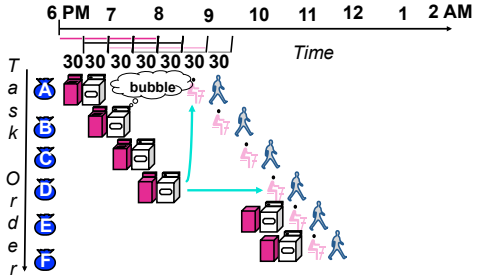
Why Do Dynamic Scheduling?

- 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

Speculation

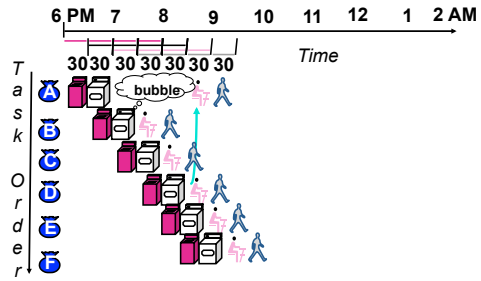
- "Guess" what to do with an instruction
  - Start operation as soon as possible
  - Check whether guess was right
    - If so, complete the operation
    - If not, roll-back and do the right thing
- Examples
  - Speculate on branch outcome (Branch Prediction)
    - Roll back if path taken is different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

Pipeline Hazard: Matching socks in later load



- A depends on D; stall since folder tied up;

Out-of-Order Laundry: Don't Wait



- A depends on D; rest continue; need more resources to allow out-of-order

Not a simple linear pipeline

- 3 major units operating in parallel
- 1 Instruction fetch and issue unit
  - Issues instructions *in program order*
- Many parallel functional (execution) units
  - Each functional unit has input buffers called Reservation Stations
  - Holds operands and records the operation
  - Can execute instructions *out-of-program-order (OOO)*
- 1 Commit unit
  - Gets results from functional unit and saves in buffers called Reorder Buffer
  - Stores results once branch resolved so OK to execute
  - Commits results *in program order*

Out-of-Order Execution (1/2)

- Basically, unroll loops in hardware

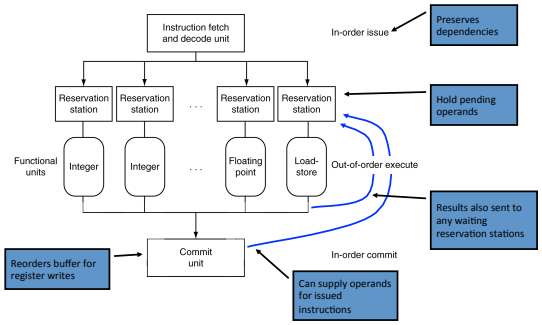
1. Fetch instructions in program order (≤4/clock)
2. Predict branches as taken/untaken
3. To avoid hazards on registers, *rename registers* using a set of internal registers (~80 registers)
4. Collection of renamed instructions might execute in a *window* (~60 instructions)
5. Execute instructions with ready operands in 1 of multiple *functional units* (ALUs, FPUs, Ld/St)

Out-of-Order Execution (2/2)

- Basically, unroll loops in hardware

6. Buffer results of executed instructions until predicted branches are resolved in *reorder buffer*
7. If predicted branch correctly, *commit* results in program order
8. If predicted branch incorrectly, discard all dependent results and start with correct PC

Dynamically Scheduled CPU



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	2930MHz	14	4	Yes	2	75W
Core 2 Yorkfield	2008	2930 MHz	16	4	Yes	4	95W
Core i7 Gulftown	2010	3460 MHz	16	4	Yes	6	130W

"And in Conclusion, …"

- Big Ideas of Instruction Level Parallelism
- Pipelining, Hazards, and Stalls
- Forwarding, Speculation to overcome Hazards
- Multiple issue to increase performance
  - IPC instead of CPI
- Dynamic Execution: Superscalar in-order issue, branch prediction, register renaming, out-of-order execution, in-order commit
  - "unroll loops in HW", hide cache misses