CS 61C: Great Ideas in Computer Architecture (Machine Structures) Instruction Level Parallelism

Instructors:
Randy H. Katz
David A. Patterson
http://inst.eecs.Berkeley.edu/~cs61c/fa10

Review

- Pipeline challenge is 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 (10 to 15 stages)
  - Superscalar (2 to 4 instructions at a time)
  - Out-of-order execution (go past the stall)
  - Speculation (branch prediction, speculative execution)

Agenda

- Review
- Dynamic Scheduling
- Example AMD Barcelona
- Administrivia
- Big Picture: Types of Parallelism
- Peer Instruction
- Summary

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

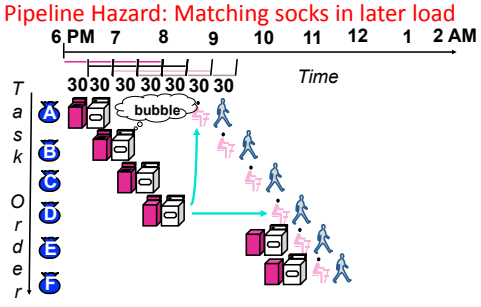
Why Do Dynamic Scheduling?

- 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

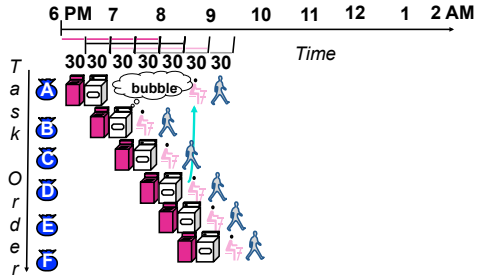
- "Guess" what to do with an instruction
  - Start operation as soon as possible
  - Check whether guess was right
    - If so, complete the operation
    - If not, roll-back and do the right thing
- Common to static and dynamic multiple issue
- Examples
  - Speculate on branch outcome (Branch Prediction)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

Pipeline Hazard: Matching socks in later load



- 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

Out-of-Order Execution (1/2)

- Basically, unroll loops in hardware

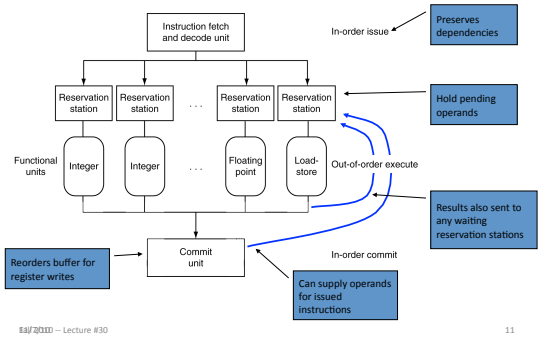
1. Fetch instructions in program order (≤4/clock)
2. Predict branches as taken/untaken
3. To avoid hazards on registers, *rename registers* using a set of internal registers (~80 registers)
4. Collection of renamed instructions might execute in a *window* (~60 instructions)
5. Execute instructions with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

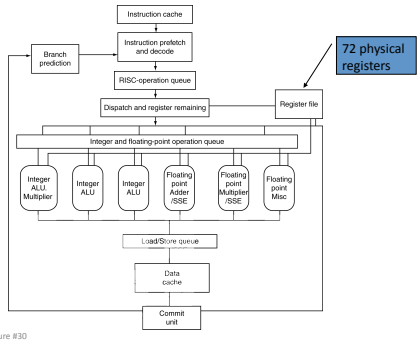


Out Of Order Intel

- 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

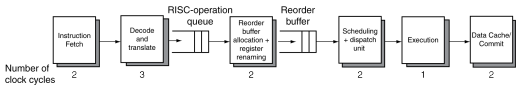
AMD Opteron X4 Microarchitecture



§4.11 Real Stuff: The AMD Opteron X4 (Barcelona) Pipeline

AMD Opteron X4 Pipeline Flow

- For integer operations



  - 12 stages (Floating Point is 17 stages)
  - Up to 106 RISC-ops in progress
- Intel Nehalem is 16 stages for integer operations, details not revealed, but likely similar to above
  - Intel calls RISC operations "Micro operations" or "μops"

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

Administrivia

- As of today, made 1 pass over all Big Ideas in Computer Architecture
- Following lectures go into more depth on topics you've already seen while you work on projects
  - 2 lectures in more depth on Caches
  - 1 in more depth on C storage management
  - 1 on Protection, Traps
  - 2 on Virtual Memory, TLB, Virtual Machines
  - 1 on Economics of Cloud Computing
  - 1.5 on Anatomy of a Modern Microprocessor (Sandy Bridge, latest microarchitecture from Intel)

Administrivia

- Project 3: TLP+DLP+Cache Opt (Due 11/13)
- Project 4: Single Cycle Processor in Logicsim
  - Due Part 2 due Saturday 11/27
  - Face-to-Face : Signup for 15m time slot 11/30, 12/2
- Extra Credit: Fastest Project 3 (due 11/29)
- Final Review: Mon Dec 6, 3hrs, afternoon (TBD)
- Final: Mon Dec 13 8AM-11AM (TBD)
  - Like midterm: T/F, M/C, short answers
  - Whole Course: readings, lectures, projects, labs, hmwks
  - Emphasize 2nd half of 61C + midterm mistakes

Administrivia

- What classes should I take (now)?
- Take classes from great teachers! (teacher > class)
  - Distinguished Teaching Award (very hard to get)
    - http://teaching.berkeley.edu/dta-dept.html
  - HKN Course evaluations (≥6 is very good)
    - http://hkn.eecs.berkeley.edu/student/CourseSurvey/
  - EECS web site has plan for year (up in late spring)
    - http://www.eecs.berkeley.edu/Scheduling/CS/schedule-draft.html
- If have choice of multiple great teachers
  - CS152 Computer Architecture and Engineering
  - CS162 Operating Systems and Systems Programming
  - CS169 Software Engineering (for SaaS with Fox)
  - CS194 Engineering Parallel Software
  - CS267 Applications of Parallel Computers

Big Picture on Parallelism

- 2 types of parallelism in *applications*

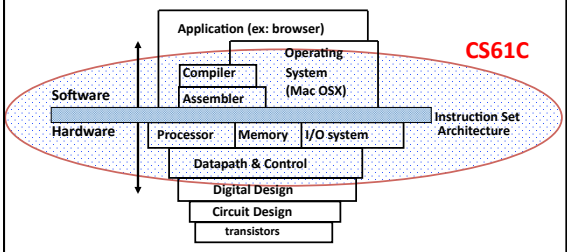
1. *Data-Level Parallelism (DLP)*: arises because there are many data items that can be operated on at the same time
2. *Task-Level Parallelism (TLP)*: arises because tasks of work are created that can operate largely in parallel

Big Picture on Parallelism

- Hardware can exploit app Data LP and Task LP in 4 ways

1. *Instruction Level Parallelism*: Hardware exploits application DLP using ideas like pipelining and speculative execution
2. *SIMD architectures* exploit app DLP by applying a single instruction to a collection of data in parallel
3. *Thread-Level Parallelism* exploits either app DLP or TLP in a tightly-coupled hardware model that allows for interaction among parallel threads
4. *Request-Level Parallelism* exploits parallelism among largely decoupled tasks and is specified by the programmer of the operating system

Machine Structures: Lecture 1



Peer Instruction

Instr LP, SIMD, Thread LP, Request LP are examples of

- **Parallelism *above* the Instruction Set Architecture**
- **Parallelism explicitly *at* the level of the ISA**
- **Parallelism *below* the level of the ISA**

	Inst. LP	SIMD	Thr. LP	Req. LP
A) (red)	=	=	=	∧
B) (orange)	∨	=	=	∧
C) (green)	=	=	∧	∧
D) (yellow)	∨	=	∧	∧
E) (burgundy)	=	∧	∧	∧
F) (blue)	∧	∧	∧	∧

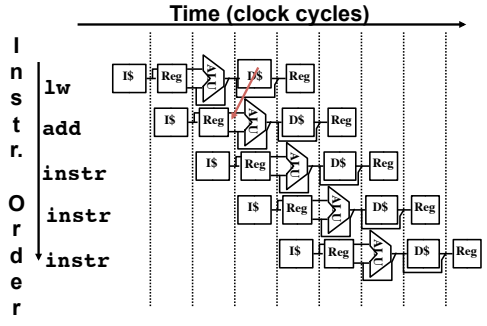
Peer Instruction: Stall, Forward, OK?

For each code sequence, chose whether
I. It must stall
II. It can avoid stalls using only forwarding
III. It can execute without stalling or forwarding

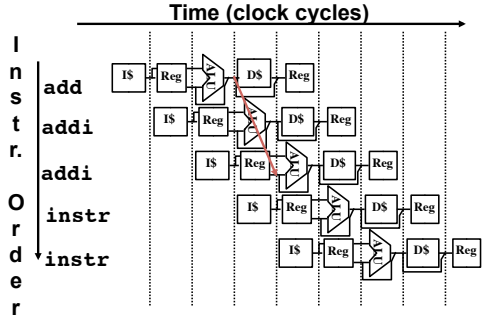
```
1:                  2:                  3: addi $t1,$t0,#1
lw   $t0,0($t0)     add $t1,$t0,$t0        addi $t2,$t0,#2
add $t1,$t0,$t0     addi $t2,$t0,#5        addi $t3,$t0,#2
                    addi $t4,$t1,#5        addi $t3,$t0,#4
                                           addi $t5,$t1,#5
```

A) All I (stall)
B) 1 I, 2 I, 3 II
C) 1 I, 2 I, 3 III
D) 1 I, 2 II, 3 III
E) 1 I, 2 II, 3 II
F) 1 II, 2 II, 3 III
G) All II (must forward)
H) All III (no stall, no fwd)

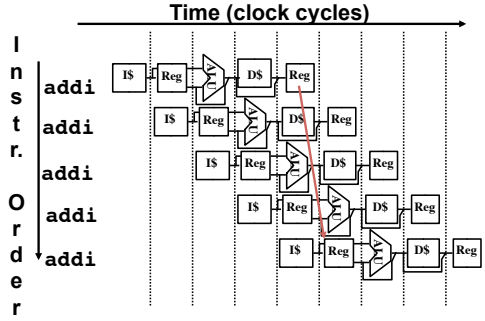
Sequence 1



Sequence 2



Sequence 3



Peer Instruction

Not all instructions are active in every stage of the 5-stage pipeline. Ignoring the effects of hazards, which of the following are true?

1. Allowing jumps, branches, and ALU instructions to take fewer stages than the 5 required by the load instruction will increase pipeline performance for most programs.
2. Trying to allow some instructions to take fewer cycles does not help, since the throughput is determined by the clock cycle; the number of pipe stages per instruction affects latency, not throughput.
3. You cannot make ALU instructions take fewer cycles because of the write back of the result, but branches and jumps can take fewer cycles, so there is some opportunity for improvement.
4. Instead of trying to make instructions take fewer cycles, we should explore making the pipeline longer, so that instructions take more cycles, but the cycles are shorter. This could improve performance.

A) All false
B) 1 F, 2 F, 3 F, 4 T
C) 1 F, 2 F, 3 T, 4 F
D) 1 F, 2 T, 3 F, 4 F
E) 1 T, 2 F, 3 F, 4 F
F) 1 F, 2 T, 3 F, 4 T
G) 1 F, 2 T, 3 T, 4 T
H) All true

Peer Instruction

State if following techniques are associated primarily with a software- or hardware-based approach to exploiting ILP (in some cases, the answer may be both): Superscalar, Out-of-Order execution, Speculation, Register Renaming

	Super-scalar	Out of Order	Specu-lation	Register Renaming
A) (red)	HW	HW	HW	HW
B) (orange)	SW	SW	SW	SW
C) (green)	Both	Both	Both	Both
D) (yellow)	HW	HW	Both	Both
E) (burgundy)	HW	HW	HW	Both
F) (blue)	HW	HW	HW	SW

Peer Instruction

I. Thanks to pipelining, I have reduced the time it took me to wash my one shirt.

II. Longer pipelines are always a win (since less work per stage & a faster clock).

A)(red) I is True and II is True
B)(orange) I is False and II is True
C)(green) I is True and II is False
D)(yellow) I is False and II is False

"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