Computer Architecture Review

Mark Greenstreet

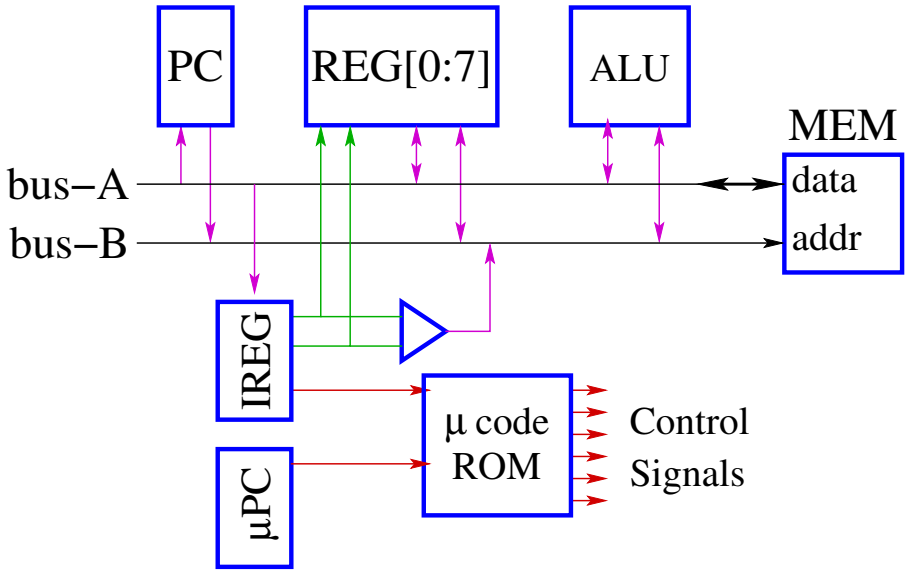
CpSc 418 – Jan. 23, 2017

- A microcoded machine
- A pipelined machine: RISC
- Let's write some code
- Superscalars and the memory bottleneck

Objectives

- Review classical, sequential architectures
  - a simple microcoded, machine
  - a pipelined, one-instruction per clock cycle machine
- Pipelining **is** parallel execution
  - the machine is supposed to appear (nearly) sequential
  - introduce the ideas of hazards and dependencies.

Microcoded machines



A simple, microcoded machine

- The microcode (μcode) ROM specifies the sequence of operations necessary to carry out an instruction.
- For simplicity, I'm assuming that the op-code bits of the instruction form the most significant bits of the μcode ROM address, and that the value of the micro-PC (μPC) form the lower half of the address.

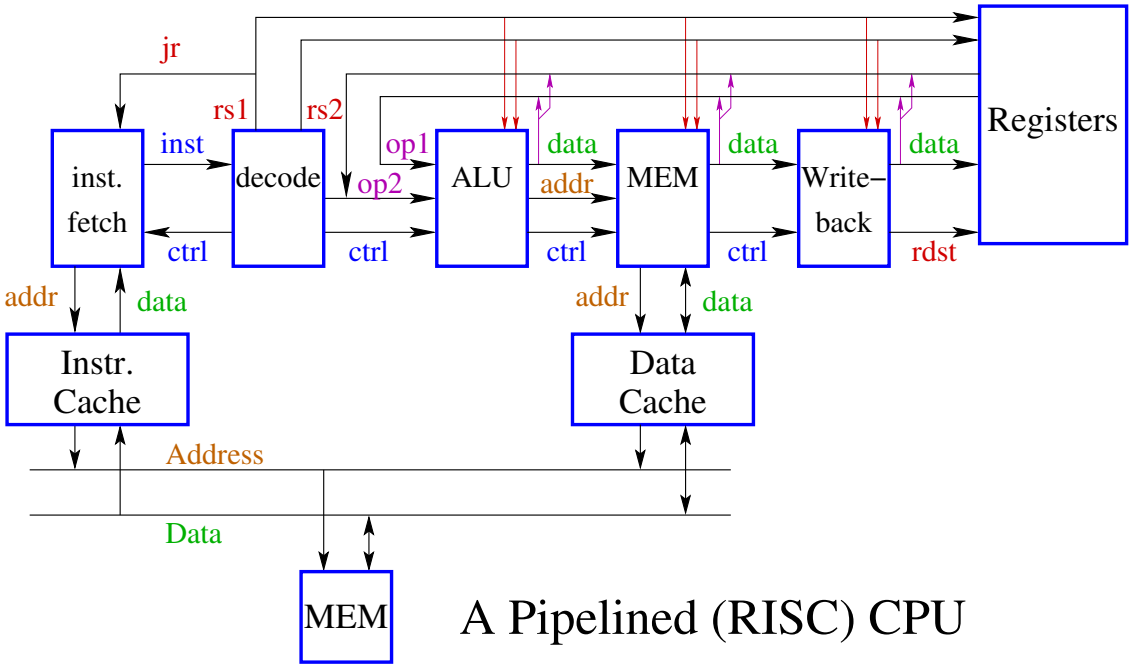
Microcode: summary

- Separates hardware from instruction set.
  - Different hardware can run the same software.
  - Enabled IBM to sell machines with a wide range of performance that were all compatible
    - I.e. IBM built an empire and made a fortune on the IBM 360 and its successors.
    - Intel has done the same with the x86.
- But, as implemented on slide 3, it's very sequential.

```
while(true) {
    fetch an instruction;
    perform the instruction
}
```

- Instruction fetch is "overhead"
  - Motivates coming up with complicated instructions that perform lots of operations per instruction fetch.
  - But these are hard for compilers to use.
  - Can we do better?

Pipelined instruction execution



A Pipelined (RISC) CPU

- Successive instructions in each stage
- When instruction `i` in `ifetch`, instruction `i-1` in decode, ...
- Allows throughput of one instruction per cycle.
- Favors simple instructions that execute on a single pass through the pipeline.
  - This is known as RISC: “Reduced Instruction Set Computer”
  - A modern x86 is CISC on the outside, but RISC on the inside.

What about Dependencies?

- Multiple-instructions are in the pipeline at the same time.
- An instruction starts before all of its predecessors have completed.
- Data hazards occur if
  - an instruction can read a different value than would have been read with a sequential execution of instructions,
  - or if a register or memory location is left holding a different value than it would have had in a sequential execution.
- Control hazards occurs if
  - an instruction is executed that would not have been executed in a sequential execution.
  - This is because the instruction “depends” on a jump or branch that hasn’t finished in time.

Handling Hazards

- Bypass: If an instruction has a result that a later instruction needs, the earlier instruction can provide that result directly without waiting to go through the register file.
- Move common operations early:
  - Decide branches in decode stage
  - ALU operations in the stage after decode
  - Memory reads take longer, but they happen less often.
- Let the compiler deal with it
- If nothing else helps, stall.

Break for Live Coding

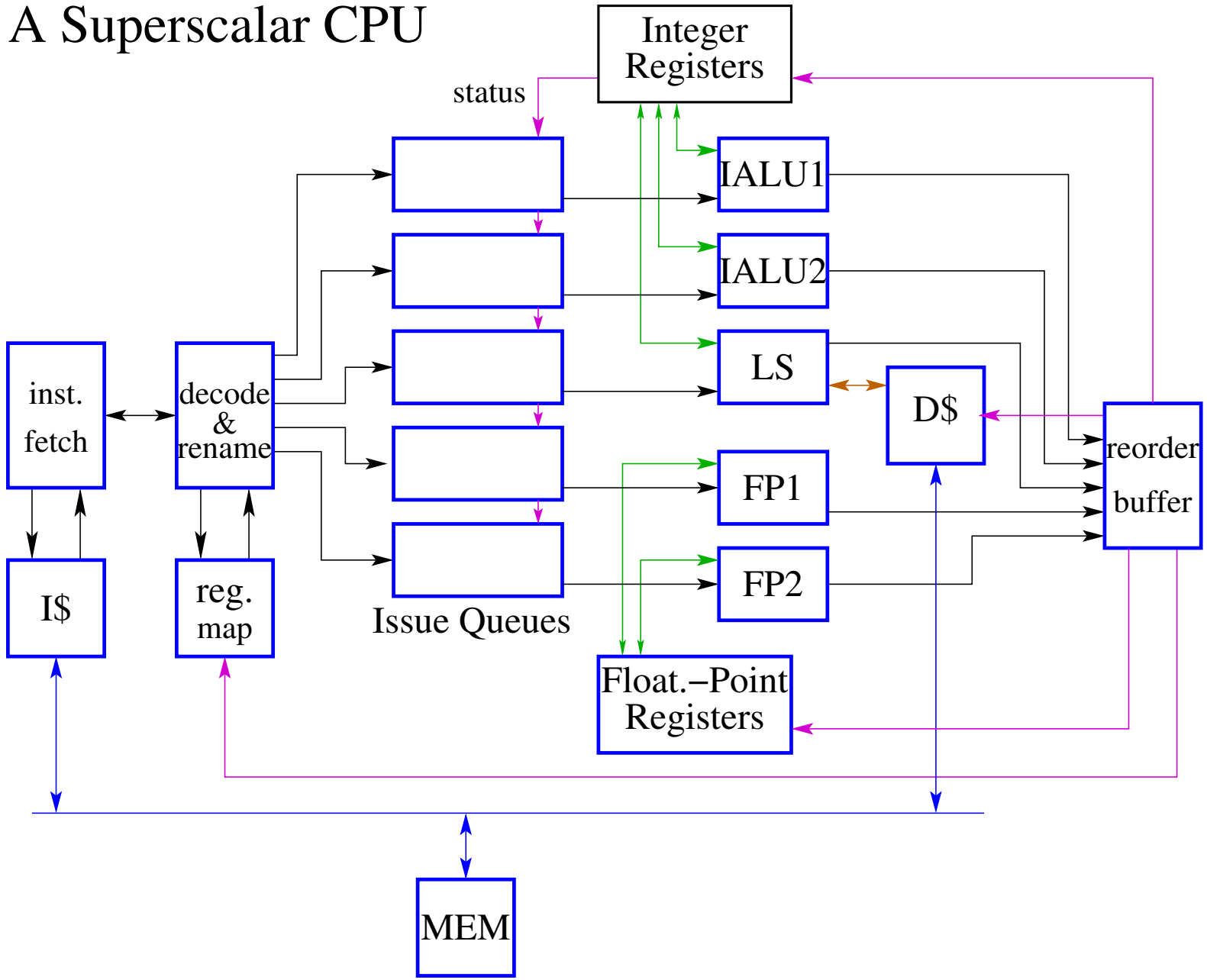
Back to Architecture

- the microcoded machine takes 5+ clock-cycles per instruction.
- the RISC machine takes 1 clock-cycle per instruction – in the best case:
  - There can be stalls due to cache misses,
  - unfilled delay slots, or
  - multi-cycle operations.
- Can we break the one-cycle-per instruction barrier?

The Memory Bottleneck

- A CPU core can execute roughly one instruction per clock-cycle.
  - With a 3GHz clock, that's roughly 0.3ns per instruction.
- Main memory accesses take 60-200ns (or longer)
  - That's 200-600 instructions per main memory access.
- Why?
  - CPUs designed for speed.
  - Memory designed for capacity:
    - fast memories are small
    - large memories are slow

Superscalar Processors



Superscalar Execution

- Fetch several, W, instructions each cycle.
- Decode them in parallel, and send them to issue queues for the appropriate functional unit.
- But what about dependencies?
  - We need to make sure that data and control dependencies are properly observed.
  - Code should execute on a superscalar as if it were executing on sequential, one-instruction-at-a-time machine.
  - Data dependencies can be handled by "**register renaming**" – this uses register indices to dynamically create the dependency graph as the program runs.
  - Control dependencies can be handled by "**branch speculation**" – guess the branch outcome, and rollback if wrong.
- The opportunity to execute instructions in parallel is called **Instruction Level Parallelism**, ILP.

What superscalars are good at

- Scientific computing:
  - often successive loop iterations are independent
  - the superscalar **pipelines** the loop
  - Perform memory reads for loop i, while doing multiplications for loop i-2, while doing additions for loop i-4, while storing the results for loop i-5.
- Commercial computing (databases, webservers, . . .)
  - often have large data sets and high cache miss rates.
  - the superscalar can find executable instructions after a cache miss.
  - if it encounters more misses, the CPU benefits from **pipelined** memory accesses.
- Burning lots of power
  - many operations in a superscalar require hardware that grows quadratically with W.
  - basically, all instructions in a batch of W have to compare there register indices with all of the other ones.

Superscalar Reality

- Most general purpose CPUs (x86, Arm, Power, SPARC) are superscalar.
- Register renaming works **very** well:
- Branch prediction is also very good, often $> 90\%$ accuracy.
  - But, data dependent branches can cause very poor performance.
- Superscalar designs make multi-threading possible
  - The features for executing multiple instruction in parallel work well for mixing instructions from several threads or processes – this is called “multithreading” (or “hyperthreading”, if you’re from Intel).
  - In practice, superscalars are often *better at multithreading* than they are at extracting ILP from a sequential program.

Preview

January 25: Shared-Memory Machines	
Reading:	Pacheco, Chapter 2, Section 2.3
January 27: Distributed-Memory Machines	
Reading:	Pacheco, Chapter 2, Sections 2.4 and 2.5.
Mini Assignments	Mini 4 goes out.
January 30: Parallel Performance: Speed-up	
Reading:	Pacheco, Chapter 2, Section 2.6.
Homework:	HW 2 earlybird (11:59pm). HW 3 goes out.
February 1: Parallel Performance: Overheads	
Homework:	HW 2 due (11:59pm).
February 3: Parallel Performance: Models	
Mini Assignments	Mini 3 due (10am)
February 6: Parallel Performance: Wrap Up	
January 8–February 15: Parallel Sorting	
Homework (Feb. 15):	HW 3 earlybird (11:59pm), HW 4 goes out.
February 17: Map-Reduce	
Homework:	HW 3 due (11:59pm).
February 27: TBD	
March 1: Midterm	

Review

- How does a pipelined architecture execute instruction in parallel?
- What are hazards?
- What are dependencies?
- What is multithreading.
- For further reading on RISC:
  "Instruction Sets and Beyond: Computers, Complexity, and Controversy"
  R.P. Colwell, *et al.*, *IEEE Computer*, vol. 18, no. 3,
  - You can download the paper for free if your machine is on the UBC network.
  - If you are off-campus, you can use the library's proxy.