Computer Architecture Review

Mark Greenstreet

CpSc 418 – Jan. 22, 2016

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.

Matrix multiply

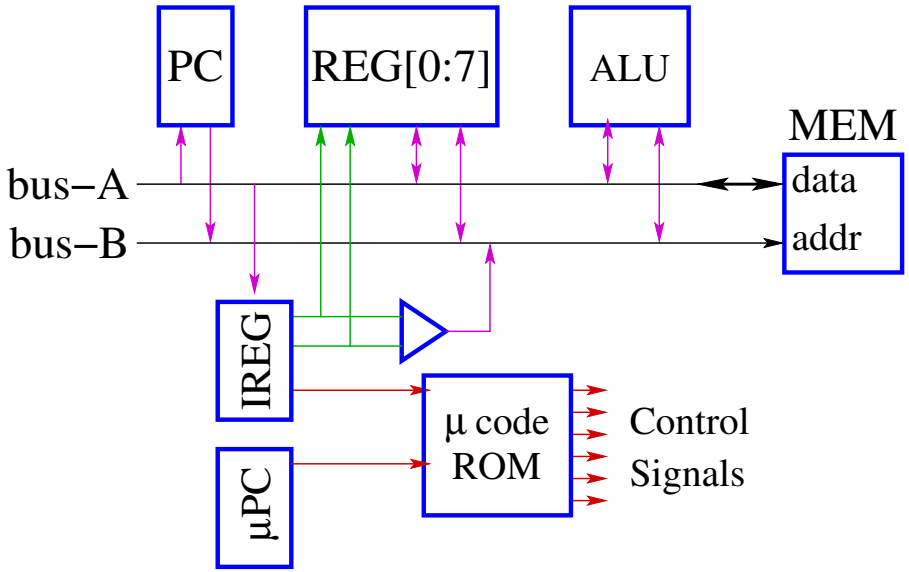
In C:

```
for(i = 0; i < n; i++) {
    for(j = 0; j < n; j++) {
        sum = 0.0;
        for(k = 0; k < n; k)
            sum += a[i,k]*b[k,j];
        c[i,j] = sum;
    }
}
```

Machine-level operations, just the inner-loop:

```
LOOP_TOP:
    $x ← Mem($aptr).double
    $aptr ← $aptr + 8
    $y ← Mem($bptr).double
    $bptr ← $bptr + $N8
    $z ← $x * $y
    $sum ← $sum + $z
    branch $aptr ≠ $atop, LOOP_TOP
```

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.

Matrix multiply on a microcoded machine

`$x ← Mem($aptr).double`: 5+ cycles:

1. Fetch instruction
2. Decode instruction
3. Fetch register, `$aptr`
4. Read memory (may take more than one cycle)
5. Write result into register $x

`$aptr ← $aptr + 8`: 5 cycles

1-3, 5: Like the load operation above.

4: An ALU operation instead of a memory operation.

`$z ← $x * $y`: 6 cycles

1-3, 5: Like the add operation above.

4: Assuming 2-cycle latency for floating point operations. See Table 2 in the *MIPS R10000* paper. Today's processors tend to have higher latencies, typically 3-5 cycles.

Matrix multiply on a microcoded machine (cont).

`branch $aptr ≠ $atop, LOOP_TOP`: Five cycles.

1-4 Like an ALU operation.

5 Update the program counter instead of a data register.

TOTAL: **36 cycles**.

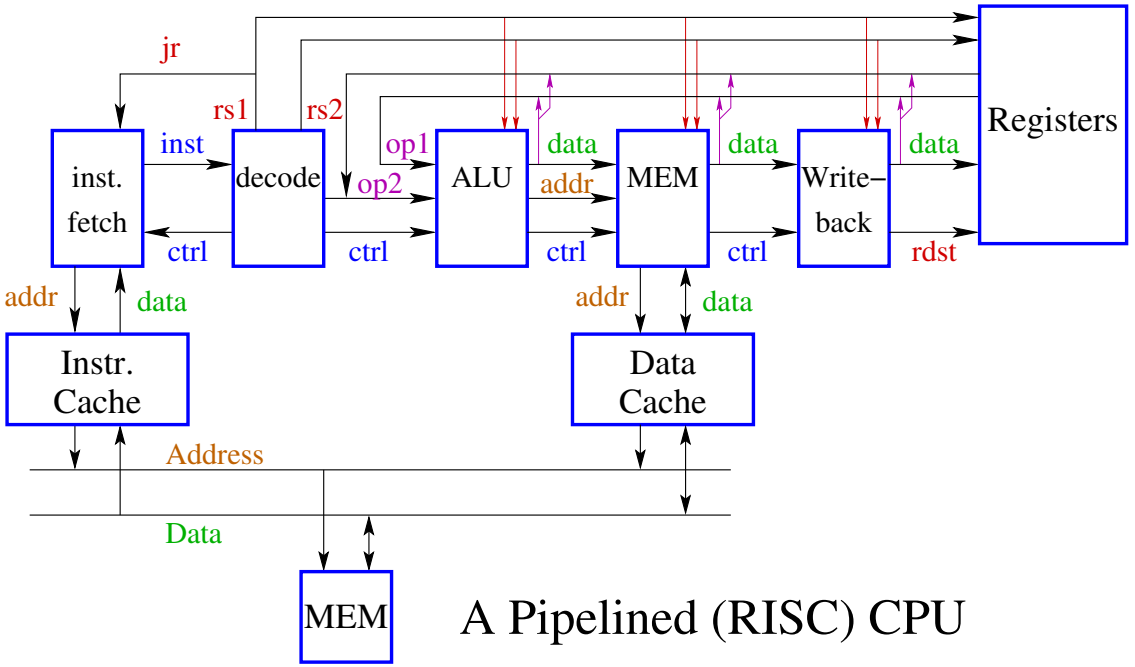
Microcode: summary

- 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 4, 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.

Data Hazards

- An instruction that reads register *I* may be issued before an earlier instruction that writes register *I* completes.
- Most RISC processors use **bypassing**
  - Register accesses are “seen” by later pipeline stages.
  - Stage *J* holds an instruction that will write register *I*,
    - Stage *J* signals the register file to ignore the access.
    - If stage *J* has the result, it provides it.
    - Otherwise, stage *J* tells the decode stage to stall.

Control Hazards

- A jump or branch will not take effect until it reaches pipeline stage where it can be executed.
- A few instructions may be fetched after a jump or a taken branch.
- Most RISC processors use **delay slots**
  - Branches are executed in the decode stage.
  - The instruction after the branch is always fetched.
  - Two choices:
    - Sqash that instruction if the branch is taken.
    - Execute it anyway – this is the “delay slot” approach.
    - Now, it’s the compiler’s problem.

Matrix multiplication on a pipelined machine

```
LOOP_TOP:
    $x ← Mem($aptr).double
    $aptr ← $aptr + 8
    $y ← Mem($bptr).double
    $bptr ← $bptr + $N8
    $z ← $x * $y
    branch $aptr ≠ $atop, LOOP_TOP
    $sum ← $sum + $z % delay slot
```

- We moved the `$sum ← $sum + $z` operation into the branch delay slot.
- This also prevented the floating point add from stalling while waiting for the floating point multiply to finish.
- The loop executes in **7 cycles** per iteration.
- But what if we have a modern processor with longer operation latencies?

Matrix multiplication on a pipelined machine

```
LOOP_TOP:
    $x ← Mem($aptr).double
    $aptr ← $aptr + 8
    $y ← Mem($bptr).double
    $bptr ← $bptr + $N8
    $z ← $x * $y
    branch $aptr ≠ $atop, LOOP_TOP
    $sum ← $sum + $z % delay slot
```

- But what if we have a modern processor with longer operation latencies?
  - E.g., an Intel Core i7 has a three cycle latency for floating point add, and a five-cycle latency for floating point multiplication.
  - It can issue a new multiply and add every cycle.

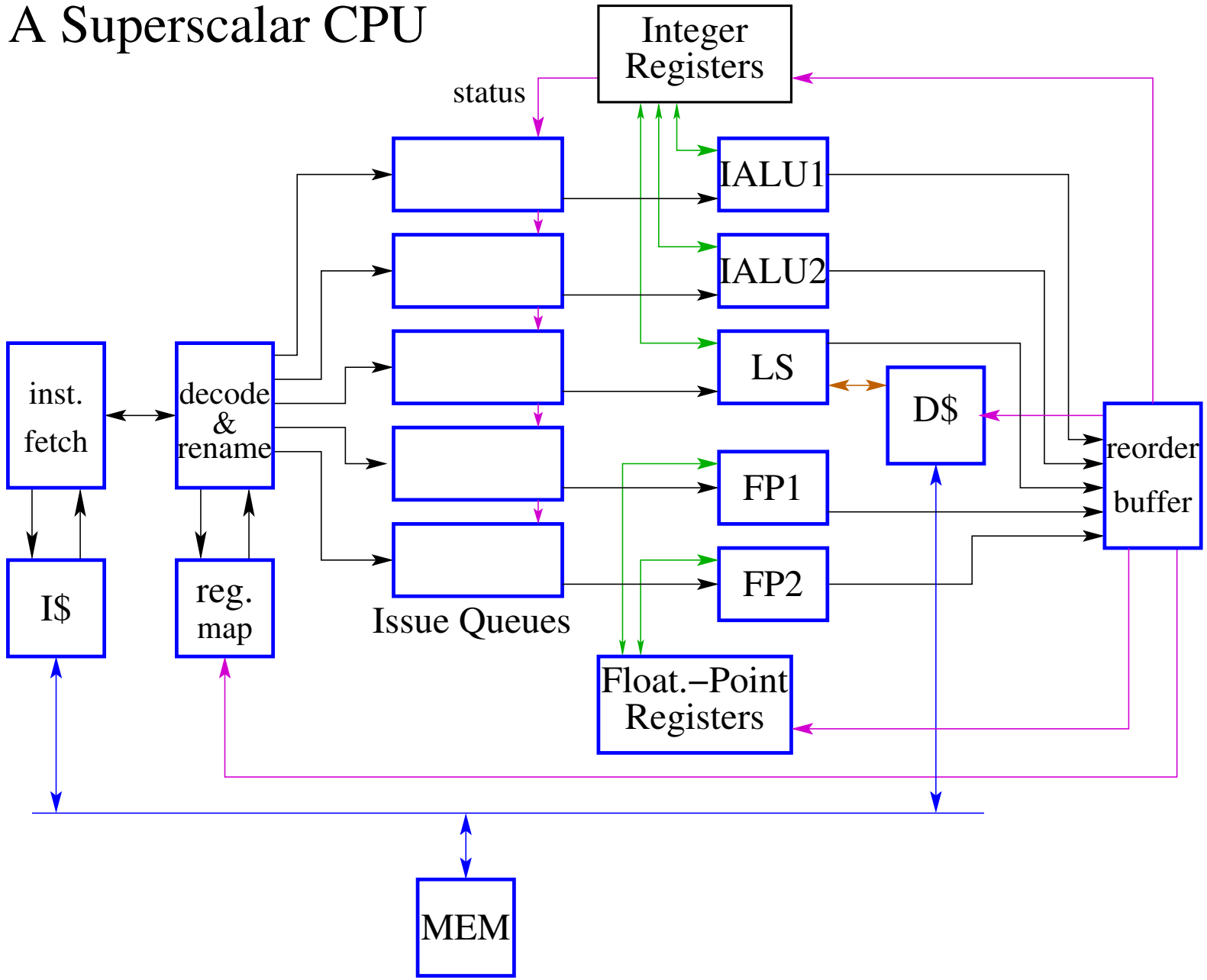
Matrix multiplication: execution

On the whiteboard

Comparison

- 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?

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.
- We'll take a closer look on Monday.

Preview

January 25: Superscalars and Simultaneous Multi-Threading
Reading: Pacheco, Chapter 2, Sections 2.1 and 2.2.

January 27: Shared-Memory Machines
Reading: Pacheco, Chapter 2, Section 2.3

January 29: Distributed-Memory Machines
Reading: Pacheco, Chapter 2, Sections 2.4 and 2.5.

February 1: Parallel Performance: Speed-up
Reading: Pacheco, Chapter 2, Section 2.6.

February 3: Parallel Performance: Modeling

February 5: Matrix Multiplication
Reading: Lin & Snyder, Chapter 5, pp. 125–133.

February 10: Midterm

Review

- How does a pipelined architecture execute instruction in parallel?
- What is a data hazard?
- What is a control hazard?
- What is bypassing?
- What is a delay slot?
- 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.