CMPE 110: Computer Architecture

Week 2

ISA II

Jishen Zhao (http://users.soe.ucsc.edu/~jzhao/)

[Adapted in part from Jose Renau, Mary Jane Irwin, Joe Devietti, Onur Mutlu, and others]

Reminder

- Homework 1 will be posted on course Google site by 4:30pm today
  - Performance and ISA
  - Due by midnight Oct. 17

- Midterm 1
  - Questions reduced to four
  - Cover everything up until last Friday class

Q1	12	
Q2	12	
Q3	18	
Q4	18	
Total	60	

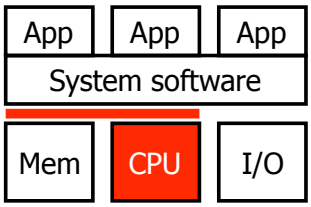
This exam is closed book and closed notes. Personal calculators (four-function calculators only) *are* allowed. Show your work on the attached sheets (front and back) and insert your answer in the space(s) provided. **Please provide details on how you reach a result.** Ask for extra paper sheets if necessary.

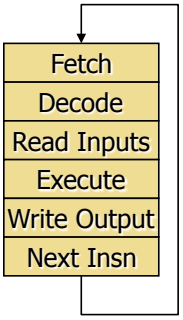
You have 70 minutes to complete the exam. This exam is worth 60 points. This exam counts for 15% of your course grade.

Reminder

- No office hour on Wednesday (Oct. 5)
- Xin's TA office hour announced
  - Time: Thu 9 - 10 am
  - Office: BE-312B

Review: ISA Overview





- What is an ISA?
  - An ISA includes a specification of the set of opcodes and the native commands implemented by a particular processor. <Wikipedia>
- Program execution model
  - **High-level programming language** (C/C++/Java/C#/…)
  - **Assembly language**
    - Human-readable representation
  - **Machine language**
    - 1s and 0s (often displayed in "hex")
- Instruction execution model

Review: An ADD Documentation

ADD[S] : Addition

ADD will add two values.

Operand 1 is a register, operand 2 can be a register, shifted register, or an immediate value (which may be shifted).

If the S bit is set (***ADDS***), the N and Z flags are set according to the result, and the C and V flags are set as follows: **C** if the result generated a carry (unsigned overflow); **V** if the result generated a signed overflow.

ADD is useful for basic addition. Use ADC to perform addition with the Carry flag considered.

Syntax

```
ADD<suffix>  <dest>, <op 1>, <op 2>
```

Function

```
dest = op_1 + op_2
```

Technical

The instruction bit pattern is as follows:

31 - 28	27	26	25	24 - 21	20	19 - 16	15 - 12	11 - 0
condition	0	0	I	0 1 0 0	S	op_1	dest	op_2/shift

Note: If the I bit is *zero*, and bits 4 and 7 are both *one* (with bits 5,6 zero), the instruction is UMULL, not ADD.

Review: ISA vs. Performance

- Latency = seconds / program =
  - (insns / program) * (cycles / insn) * (seconds / cycle)
  - Insns / program: **insn count**
    - Impacted by program, compiler, ISA
  - Cycles / insn: **CPI**
    - Impacted by program, compiler, ISA, micro-arch
  - Seconds / cycle: clock period (Hz)
    - Impacted by micro-arch, technology

- How does ISA affect insn count and CPI?
  - Example: CISC vs. RISC

CISC vs. RISC

- **CISC** (Complex Instruction Set Computing) **ISAs**
  - Examples: Intel/AMD x86, IBM System/360, Intel 8051
  - Big heavyweight instructions (lots of work per instruction)
  + Low "insns/program"
  – Higher "cycles/insn" and "seconds/cycle"

- **RISC** (Reduced Instruction Set Computer) **ISAs**
  - Examples: MIPS, ARM, Alpha
  - Simple instructions only: minimalist approach to an ISA
  + Low "cycles/insn" and "seconds/cycle"
  – Higher "insn/program", but hopefully not as much

Today: ISA / Pipelining

- ISA design goals
  - Programmability
  - Performance/implementability
  - Compatibility
- Aspects of ISAs
- Pipelining

ISA Design Goals

What Makes a Good ISA?

- **Programmability**
  - Easy to express programs efficiently?
- **Performance/Implementability**
  - Easy to design high-performance implementations?
  - Easy to design low-power implementations?
  - Easy to design low-cost implementations?
- **Compatibility**
  - Easy to maintain as languages, programs, and technology evolve?
  - x86 (IA32) generations: 8086, 286, 386, 486, Pentium, PentiumII, PentiumIII, Pentium4, Core2, Core i7, …

Compatibility

- In many domains, ISA must remain compatible
  - IBM's 360/370 (the *first* "ISA family")
  - Another example: Intel's x86 and Microsoft Windows
    - x86 one of the worst designed ISAs EVER, but it survives
- **Backward compatibility**
  - New processors supporting old programs
    - **Hard to drop features**
    - Update software/OS to emulate dropped features (slow)
- **Forward (upward) compatibility**
  - Old processors supporting new programs
    - Include a "CPU ID" so the software can test for features
    - Add ISA hints by overloading no-ops (example: x86's PAUSE)
    - New firmware/software on old processors to emulate new insn

Translation and Virtual ISAs

- New compatibility interface: ISA + translation software
  - **Binary-translation** (static): transform static image, run native
  - **Emulation** (dynamic binary-translation):
    unmodified image, interpret each dynamic insn
    - Typically optimized with just-in-time (JIT) compilation
  - Examples: DEC FX!32 (x86 on Alpha), Rosetta (PowerPC on x86)
  - Performance overheads reasonable (many advances over the years)

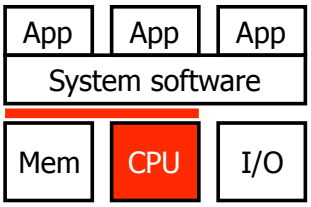
- **Virtual ISAs**: designed for translation, not direct execution
  - Target for high-level compiler (one per language)
  - Source for low-level translator (one per ISA)
  - Goals: Portability (abstract hardware nastiness), flexibility over time
  - Examples: Java Bytecodes, C# CLR (Common Language Runtime), NVIDIA's "PTX"

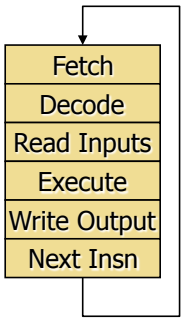
Ultimate Compatibility Trick

- Support old ISA by…
  - …having a simple processor for that ISA somewhere in the system
  - How did PlayStation2 support PlayStation1 games?
    - Used PlayStation processor for I/O chip **& emulation**

What we learned

- What is ISA?
- Execution model:
  - Compilation
  - Assembly & machine language
- Instruction execution model
  - Registers, memory, PC
  - Instruction execution
- ISA design goals
  - Programmability
  - Performance/implementability
  - Compatibility

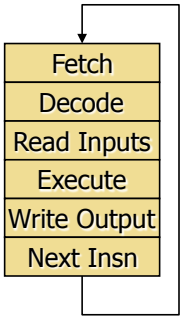




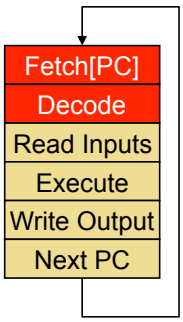
Aspects of ISAs

Aspects of ISAs

- Instruction length and format
  - Instruction encoding
- Where does data live?
  - Addressing modes
- Control transfers
  - How to find the next instruction
  - Branch
  - Jump



Length and Format

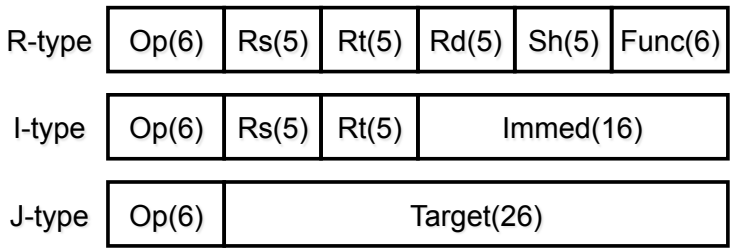


- **Length**
  - Fixed length
    - Most common is 32 bits
    - \+ Simple implementation (next PC often just PC+4)
    - – Code density: 32 bits to increment a register by 1
  - Variable length
    - \+ Code density
      - x86 averages 3 bytes (ranges from 1 to 16)
    - – Complex fetch (where does next instruction begin?)
  - Compromise: two lengths
    - E.g., MIPS16 or ARM's Thumb (16 bits)
- **Encoding**
  - A few simple encodings simplify decoder
  - Machine code (1s and 0s) <-> assembly

Example Instruction Encodings

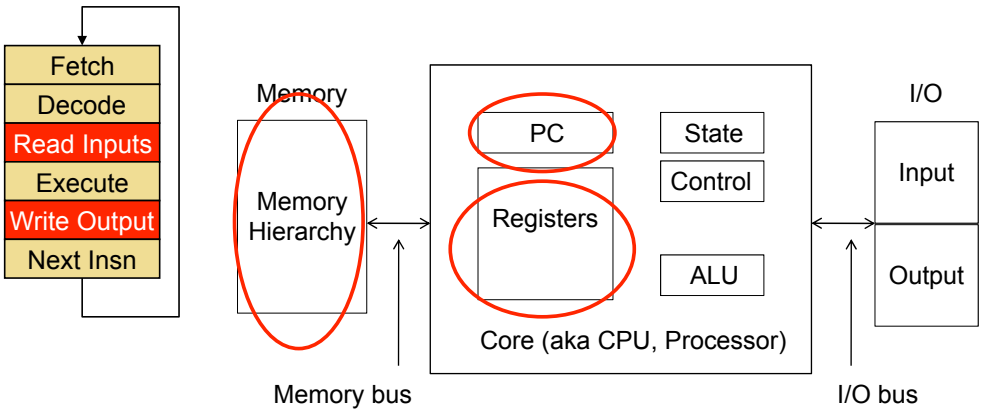
- MIPS
  - Fixed length
  - 32-bits, 3 formats, simple encoding

```
add R1,R2,R3
```

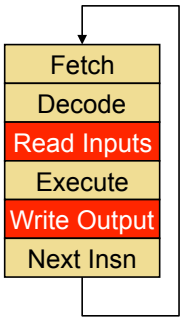


- x86
  - Variable length encoding (1 to 15 bytes)

Where Does Data Live?



Where Does Data Live?



- **Registers (e.g., R0, R1, F0)**
  - "short term memory"
  - Faster than memory, quite handy
  - Named directly in instructions

```
ADD R1,R2,R3
```

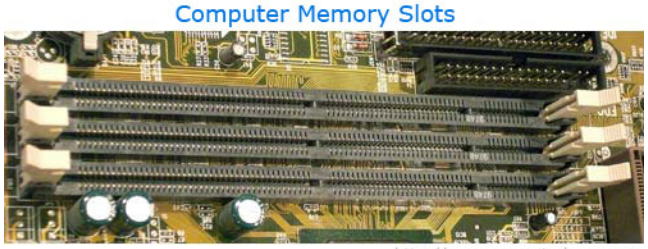
- **Memory (e.g., (R3), #20(R5))**
  - "longer term memory"
  - Accessed via "addressing modes"
    - Address to read or write calculated by instruction

```
ADD R1,R2,(R3)
```

- "Immediates" (e.g., #36, #7)
  - Values spelled out as bits in instructions
  - Input only

Memory Addressing

Computer Memory Slots



http://www.computerhope.com