

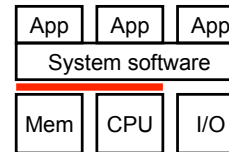
CIS 371

Computer Organization and Design

Unit 1: Instruction Set Architectures

Slides developed by Milo Martin & Amir Roth at the University of Pennsylvania with sources that included University of Wisconsin slides by Mark Hill, Guri Sohi, Jim Smith, and David Wood.

Instruction Set Architecture (ISA)



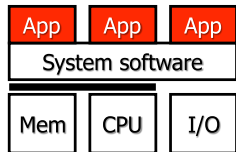
- What is an ISA?
 - A functional contract
- All ISAs similar in high-level ways
 - But many design choices in details
 - Two “philosophies”: CISC/RISC
 - Difference is blurring
- Good ISA...
 - Enables high-performance
 - At least doesn't get in the way
- Compatibility is a powerful force
 - Tricks: binary translation, μ ISAs

Readings

- Readings
 - Introduction
 - P&H, Chapter 1
 - ISAs
 - P&H, Chapter 2

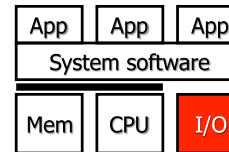
Recall from CIS240...

240 Review: Applications



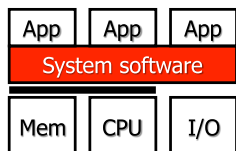
- **Applications** (Firefox, iTunes, Skype, Word, Google)
 - Run on hardware ... but how?

240 Review: I/O



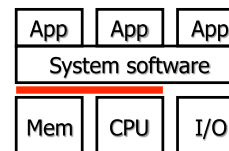
- Apps interact with us & each other via **I/O (input/output)**
 - With us: display, sound, keyboard, mouse, touch-screen, camera
 - With each other: disk, network (wired or wireless)
 - Most I/O proper is analog-digital and domain of EE
 - I/O devices present rest of computer a digital interface (1s and 0s)

240 Review: OS



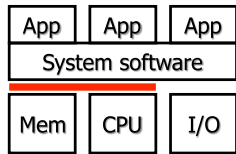
- I/O (& other services) provided by **OS (operating system)**
 - A super-app with privileged access to all hardware
 - Abstracts away a lot of the nastiness of hardware
 - Virtualizes hardware to isolate programs from one another
 - Each application is oblivious to presence of others
 - Simplifies programming, makes system more robust and secure
 - Privilege is key to this
 - Commons OSes are Windows, Linux, MacOS

240 Review: ISA



- App/OS are software ... execute on hardware
- HW/SW interface is **ISA (instruction set architecture)**
 - A **"contract"** between SW and HW
 - Encourages compatibility, allows SW/HW to evolve independently
 - **Functional definition** of HW storage locations & operations
 - Storage locations: registers, memory
 - Operations: add, multiply, branch, load, store, etc.
 - **Precise description** of how to invoke & access them
 - Instructions (bit-patterns hardware interprets as commands)

240 Review: LC4 ISA



- **LC4**: a toy ISA you know
 - 16-bit ISA (what does this mean?)
 - 16-bit insns
 - 8 registers (integer)
 - ~30 different insns
 - Simple OS support
- **Assembly language**
 - Human-readable ISA representation

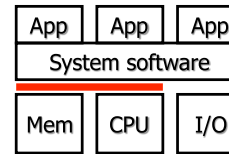
```

.DATA
array .BLKW #100
sum .FILL #0
.CODE
.FALIGN
array_sum
    CONST R5, #0
    LEA R1, array
    LEA R2, sum
array_sum_loop
    LDR R3, R1, #0
    LDR R4, R2, #0
    ADD R4, R3, R4
    STR R4, R2, #0
    ADD R1, R1, #1
    ADD R5, R5, #1
    CMPI R5, #100
    BRn array_sum_loop
    
```

CIS 371 (Martin): Instruction Set Architectures

9

371 Preview: A Real ISA



- **MIPS**: example of real ISA
 - 32/64-bit operations
 - 32-bit insns
 - 64 registers
 - 32 integer, 32 floating point
 - ~100 different insns
 - Full OS support

Example code is MIPS, but all ISAs are similar at some level

```

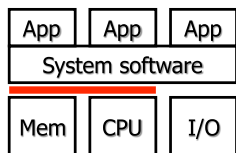
.data
array: .space 100
sum: .word 0
.text

array_sum:
    li $5, 0
    la $1, array
    la $2, sum
array_sum_loop:
    lw $3, 0($1)
    lw $4, 0($2)
    add $4, $3, $4
    sw $4, 0($2)
    addi $1, $1, 1
    addi $5, $5, 1
    li $6, 100
    blt $5, $6, array_sum_loop
    
```

CIS 371 (Martin): Instruction Set Architectures

10

240 Review: Program Compilation



```

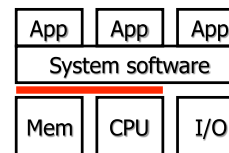
int array[100], sum;
void array_sum() {
    for (int i=0; i<100;i++) {
        sum += array[i];
    }
}
    
```

- **Program** written in a “high-level” programming language
 - C, C++, Java, C#
 - Hierarchical, structured control: loops, functions, conditionals
 - Hierarchical, structured data: scalars, arrays, pointers, structures
- **Compiler**: translates program to **assembly**
 - Parsing and straight-forward translation
 - Compiler also optimizes
 - Compiler itself another application ... who compiled compiler?

CIS 371 (Martin): Instruction Set Architectures

11

240 Review: Assembly Language



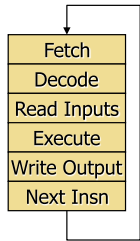
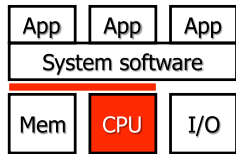
- **Assembly language**
 - Human-readable representation
- **Machine language**
 - Machine-readable representation
 - 1s and 0s (often displayed in “hex”)
- **Assembler**
 - Translates assembly to machine

Machine code	Assembly code
x9A00	CONST R5, #0
x9200	CONST R1, array
xD320	HICONST R1, array
x9464	CONST R2, sum
xD520	HICONST R2, sum
x6640	LDR R3, R1, #0
x6880	LDR R4, R2, #0
x18C4	ADD R4, R3, R4
x7880	STR R4, R2, #0
x1261	ADD R1, R1, #1
x1BA1	ADD R5, R5, #1
x2B64	CMPI R5, #100
x03F8	BRn array_sum_loop

CIS 371 (Martin): Instruction Set Architectures

12

240 Review: Insn Execution Model

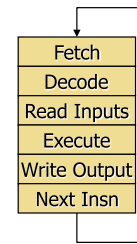


Instruction → Insn

- The computer is just finite state machine
 - **Registers** (few of them, but fast)
 - **Memory** (lots of memory, but slower)
 - **Program counter** (next insn to execute)
 - Sometimes called “instruction pointer”
- A computer executes **instructions**
 - **Fetches** next instruction from memory
 - **Decodes** it (figure out what it does)
 - **Reads** its **inputs** (registers & memory)
 - **Executes** it (adds, multiply, etc.)
 - **Write** its **outputs** (registers & memory)
 - **Next insn** (adjust the program counter)
- **Program is just “data in memory”**
 - Makes computers programmable (“universal”)

What is an ISA?

The Sequential Model



- Basic structure of all modern ISAs
 - Often called VonNeuman, but in ENIAC before
- **Program order**: total order on dynamic insns
 - Order and **named storage** define computation
- Convenient feature: **program counter (PC)**
 - Insn itself stored in memory at location pointed to by PC
 - Next PC is next insn unless insn says otherwise
- Processor logically executes loop at left
- **Atomic**: insn finishes before next insn starts
 - Implementations can break this constraint physically
 - But must maintain illusion to preserve correctness

What Is An ISA?

- **ISA (instruction set architecture)**
 - A well-defined hardware/software interface
 - The **“contract”** between software and hardware
 - **Functional definition** of storage locations & operations
 - Storage locations: registers, memory
 - Operations: add, multiply, branch, load, store, etc
 - **Precise description** of how to invoke & access them
- Not in the “contract”: non-functional aspects
 - How operations are implemented
 - Which operations are fast and which are slow and when
 - Which operations take more power and which take less
- Instructions
 - Bit-patterns hardware interprets as commands
 - Instruction → Insn (instruction is too long to write in slides)

A Language Analogy for ISAs

- Communication
 - Person-to-person → software-to-hardware
- Similar structure
 - Narrative → program
 - Sentence → insn
 - Verb → operation (add, multiply, load, branch)
 - Noun → data item (immediate, register value, memory value)
 - Adjective → addressing mode
- Many different languages, many different ISAs
 - Similar basic structure, details differ (sometimes greatly)
- Key differences between languages and ISAs
 - Languages evolve organically, many ambiguities, inconsistencies
 - ISAs are explicitly engineered and extended, unambiguous

LC4 vs Real ISAs

- LC4 has the basic features of a real-world ISAs
 - ± LC4 lacks a good bit of realism
 - Address size is only 16 bits
 - Only one data type (16-bit signed integer)
 - Little support for system software, none for multiprocessing (later)
- Many real-world ISAs to choose from:
 - **Intel x86**
 - **MIPS (used throughout in book)**
 - ARM
 - PowerPC
 - SPARC
 - Intel's Itanium
 - Historical: IBM 370, VAX, Alpha, PA-RISC, 68k, ...

ISA Design Goals

What Makes a Good ISA?

- **Programmability**
 - Easy to express programs efficiently?
- **Performance/Implementability**
 - Easy to design **high-performance implementations**?
 - More recently
 - 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, ...

Programmability

- Easy to express programs efficiently?
 - For whom?
- Before 1985: **human**
 - Compilers were terrible, most code was hand-assembled
 - Want high-level coarse-grain instructions
 - As similar to high-level language as possible
- After 1985: **compiler**
 - Optimizing compilers generate much better code than you or I
 - Want low-level fine-grain instructions
 - Compiler can't tell if two high-level idioms match exactly or not
- More on this later in this set of slides...

Performance, Performance, Performance

- How long does it take for a program to execute?
 - Three factors
- 1. How many insns must execute to complete program?
 - **Instructions per program** during execution
 - "Dynamic insn count" (not number of "static" insns in program)
- 2. How quickly does the processor "cycle"?
 - **Clock frequency** (cycles per second) 1 gigahertz (Ghz)
 - or expressed as reciprocal, **Clock period** nanosecond (ns)
 - Worst-case delay through circuit for a particular design
- 3. How many *cycles* does each instruction take to execute?
 - **Cycles per Instruction** (CPI) or reciprocal, **Insn per Cycle** (IPC)

$$\text{Execution time} = (\text{instructions/program}) * (\text{seconds/cycle}) * (\text{cycles/instruction})$$

Maximizing Performance

$$\text{Execution time} = (\text{instructions/program}) * (\text{seconds/cycle}) * (\text{cycles/instruction})$$
$$(1 \text{ billion instructions}) * (1 \text{ ns per cycle}) * (1 \text{ cycle per insn}) = 1 \text{ second}$$

- Instructions per program:
 - Determined by program, compiler, instruction set architecture (ISA)
- Cycles per instruction: "CPI"
 - Typical range today: 2 to 0.5
 - Determined by program, compiler, ISA, micro-architecture
- Seconds per cycle: "clock period"
 - Typical range today: 2ns to 0.25ns
 - Reciprocal is frequency: 0.5 Ghz to 4 Ghz (1 Htz = 1 cycle per sec)
 - Determined by micro-architecture, technology parameters
- For minimum execution time, minimize each term
 - Difficult: **often pull against one another**

Example: Instruction Granularity

$$\text{Execution time} = (\text{instructions/program}) * (\text{seconds/cycle}) * (\text{cycles/instruction})$$

- **CISC** (Complex Instruction Set Computing) **ISAs**
 - Big heavyweight instructions (lots of work per instruction)
 - + Low "insns/program"
 - Higher "cycles/insn" and "seconds/cycle"
 - We have the technology to get around this problem
- **RISC** (Reduced Instruction Set Computer) **ISAs**
 - Minimalist approach to an ISA: simple insns only
 - + Low "cycles/insn" and "seconds/cycle"
 - Higher "insn/program", but hopefully not as much
 - Rely on compiler optimizations

Compiler Optimizations

- Primarily goal: reduce instruction count
 - Eliminate redundant computation, keep more things in registers
 - + Registers are faster, fewer loads/stores
 - An ISA can make this difficult by having too few registers
- But also...
 - Reduce branches and jumps (later)
 - Reduce cache misses (later)
 - Reduce dependences between nearby insns (later)
 - An ISA can make this difficult by having implicit dependences
- How effective are these?
 - + Can give 4X performance over unoptimized code
 - Collective wisdom of 40 years (“Proebsting’s Law”): 4% per year
 - Funny but ... shouldn’t leave 4X performance on the table

ISA Code Example

Compiler Optimization Example (LC4)

```
;; temp = *first
LDR R7, R5, #2 ; R7=first
LDR R4, R7, #0
STR R4, R5, #-1
;; *first = *second
LDR R3, R5, #3 ; R3=second
LDR R2, R3, #0
LDR R7, R5, #2 ; redundant
STR R2, R7, #0
;; *second = temp
LDR R4, R5, #-1
LDR R3, R5, #3 ; redundant
STR R4, R3, #0
```

```
;; temp = *first
LDR R7, R5, #2
LDR R4, R7, #0
STR R4, R5, #-1 ; unneeded
;; *first = *second
LDR R3, R5, #3
LDR R2, R3, #0
STR R2, R7, #0
;; *second = temp
LDR R4, R5, #-1 ; unneeded
STR R4, R3, #0
```

- Left: **common sub-expression elimination**
 - Remove calculations whose results are already in some register
- Right: **register allocation**
 - Keep temporary in register across statements, avoid stack spill/fill

Array Sum Loop: LC4

```
.DATA
array .BLKW #100
sum .FILL #0
.CODE
.FALIGN
array_sum
CONST R5, #0
LEA R1, array
LEA R2, sum
L1
LDR R3, R1, #0
LDR R4, R2, #0
ADD R4, R3, R4
STR R4, R2, #0
ADD R1, R1, #1
ADD R5, R5, #1
CMPI R5, #100
BRn L1
```

```
int array[100];
int sum;
void array_sum() {
    for (int i=0; i<100;i++)
    {
        sum += array[i];
    }
}
```

Array Sum Loop: LC4 → MIPS

<pre>.DATA array .BLKW #100 sum .FILL #0 .CODE .FALIGN array_sum CONST R5, #0 LEA R1, array LEA R2, sum L1 LDR R3, R1, #0 LDR R4, R2, #0 ADD R4, R3, R4 STR R4, R2, #0 ADD R1, R1, #1 ADD R5, R5, #1 CMPI R5, #100 BRn L1</pre>	<pre>.data array: .space 100 sum: .word 0 .text array_sum: li \$5, 0 la \$1, array la \$2, sum L1: lw \$3, 0(\$1) lw \$4, 0(\$2) add \$4, \$3, \$4 sw \$4, 0(\$2) addi \$1, \$1, 1 addi \$5, \$5, 1 li \$6, 100 blt \$5, \$6, L1</pre>	<p>MIPS (right) similar to LC4</p> <p>Syntactic differences: register names begin with \$ immediates are un-prefixed</p> <p>Only simple addressing modes syntax: displacement(reg)</p> <p>Left-most register is generally destination register</p>
---	--	--

CIS 371 (Martin): Instruction Set Architectures 29

Array Sum Loop: LC4 → x86

<pre>.DATA array .BLKW #100 sum .FILL #0 .CODE .FALIGN array_sum CONST R5, #0 LEA R1, array LEA R2, sum L1 LDR R3, R1, #0 LDR R4, R2, #0 ADD R4, R3, R4 STR R4, R2, #0 ADD R1, R1, #1 ADD R5, R5, #1 CMPI R5, #100 BRn L1</pre>	<pre>.LFE2 .comm array,400,32 .comm sum,4,4 .globl array_sum array_sum: movl \$0, -4(%rbp) .L1: movl -4(%rbp), %eax movl array(,%eax,4), %edx movl sum(%rip), %eax addl %edx, %eax movl %eax, sum(%rip) addl \$1, -4(%rbp) cmpl \$99,-4(%rbp) jle .L1</pre>	<p>x86 (right) is different</p> <p>Syntactic differences: register names begin with % immediates begin with \$</p> <p>%rbp is base (frame) pointer</p> <p>Many addressing modes</p>
---	---	---

CIS 371 (Martin): Instruction Set Architectures 30

x86 Operand Model

<pre>.LFE2 .comm array,400,32 .comm sum,4,4 .globl array_sum array_sum: movl \$0, -4(%rbp) .L1: movl -4(%rbp), %eax movl array(,%eax,4), %edx movl sum(%rip), %eax addl %edx, %eax movl %eax, sum(%rip) addl \$1, -4(%rbp) cmpl \$99,-4(%rbp) jle .L1</pre>	<ul style="list-style-type: none"> x86 uses explicit accumulators <ul style="list-style-type: none"> Both register and memory Distinguished by addressing mode <p>Two operand insns (right-most is typically source & destination)</p> <p>Register accumulator: %eax = %eax + %edx</p> <p>"L" insn suffix and "%e..." reg. prefix mean "32-bit value"</p> <p>Memory accumulator: Memory[%rbp-4] = Memory[%rbp-4] + 1</p>
---	--

CIS 371 (Martin): Instruction Set Architectures 31

Array Sum Loop: x86 → Optimized x86

<pre>.LFE2 .comm array,400,32 .comm sum,4,4 .globl array_sum array_sum: movl \$0, -4(%rbp) .L1: movl -4(%rbp), %eax movl array(,%eax,4), %edx movl sum(%rip), %eax addl %edx, %eax movl %eax, sum(%rip) addl \$1, -4(%rbp) cmpl \$99,-4(%rbp) jle .L1</pre>	<pre>.LFE2 .comm array,400,32 .comm sum,4,4 .globl array_sum array_sum: movl sum(%rip), %edx xorl %eax, %eax .L1: addl array(%rax), %edx addq \$4, %rax cmpq \$400, %rax jne .L1</pre>
---	--

CIS 371 (Martin): Instruction Set Architectures 32

Aspects of ISAs

LC4/MIPS/x86 Length and Encoding

- LC4: 2-byte insns, 3 formats

0-reg	Op(4)	Offset(12)			
1-reg	Op(4)	R(3)	Offset(9)		
2-reg	Op(4)	R(3)	R(3)	Offset(6)	
3-reg	Op(4)	R(3)	R(3)	U(3)	R(3)

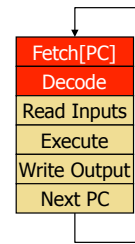
- MIPS: 4-byte insns, 3 formats

R-type	Op(6)	Rs(5)	Rt(5)	Rd(5)	Sh(5)	Func(6)
I-type	Op(6)	Rs(5)	Rt(5)	Immed(16)		
J-type	Op(6)	Target(26)				

- x86: 1–16 byte insns, many formats

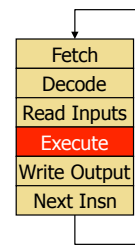
Prefix*(1-4)	Op	OpExt*	ModRM*	SIB*	Disp*(1-4)	Imm*(1-4)
--------------	----	--------	--------	------	------------	-----------

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 can do increment in one 8-bit instruction
 - Complex fetch (where does next instruction begin?)
 - Compromise: two lengths
 - E.g., MIPS16 or ARM's Thumb
- Encoding**
 - A few simple encodings simplify decoder
 - x86 decoder one nasty piece of logic

Operations and Datatypes

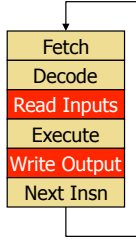


- Datatypes**
 - Software: attribute of data
 - Hardware: attribute of operation, data is just 0/1's
- All processors support
 - Integer arithmetic/logic (8/16/32/64-bit)
 - IEEE754 floating-point arithmetic (32/64-bit)
- More recently, most processors support
 - "Packed-integer" insns, e.g., MMX
 - "Packed-floating point" insns, e.g., SSE/SSE2
 - For multimedia, more about these later
- Other, infrequently supported, data types
 - Decimal, other fixed-point arithmetic

LC4/MIPS/x86 Operations and Datatypes

- LC4
 - 16-bit integer: add, and, not, sub, mul, div, or, xor, shifts
 - No floating-point
- MIPS
 - 32(64) bit integer: add, sub, mul, div, shift, rotate, and, or, not, xor
 - 32(64) bit floating-point: add, sub, mul, div
- x86
 - 32(64) bit integer: add, sub, mul, div, shift, rotate, and, or, not, xor
 - 80-bit floating-point: add, sub, mul, div, sqrt
 - 64-bit packed integer (MMX): padd, pmul...
 - 64(128)-bit packed floating-point (SSE/2): padd, pmul...

Where Does Data Live?

- 
- **Registers**
 - Named directly in instructions
 - “short term memory”
 - Faster than memory, quite handy
 - **Memory**
 - Fundamental storage space
 - “longer term memory”
 - **Immediates**
 - Values spelled out as bits in instructions
 - Input only

How Many Registers?

- Registers faster than memory, have as many as possible?
 - **No**
- One reason registers are faster: there are **fewer of them**
 - Small is fast (hardware truism)
- Another: they are **directly addressed** (no address calc)
 - More registers, means more bits per register in instruction
 - Thus, fewer registers per instruction or larger instructions
- **Not everything can be put in registers**
 - Structures, arrays, anything pointed-to
 - Although compilers are getting better at putting more things in
- More registers means **more saving/restoring**
 - Across function calls, traps, and context switches
- Trend: more registers: 8 (x86) → 32 (MIPS) → 128 (IA64)
 - 64-bit x86 has 16 64-bit integer and 16 128-bit FP registers

LC4/MIPS/x86 Registers

- LC4
 - 8 16-bit integer registers
 - No floating-point registers
- MIPS
 - 32 32-bit integer registers (\$0 hardwired to 0)
 - 32 32-bit floating-point registers (or 16 64-bit registers)
- x86
 - 8 8/16/32-bit integer registers (not general purpose)
 - No floating-point registers!
- 64-bit x86
 - 16 64-bit integer registers
 - 16 128-bit floating-point registers

How Much Memory? Address Size

- What does “64-bit” in a 64-bit ISA mean?
 - **Each program can address (i.e., use) 2^{64} bytes**
 - 64 is the **virtual address (VA) size**
 - Alternative (wrong) definition: width of arithmetic operations
- Most critical, inescapable ISA design decision
 - Too small? Will limit the lifetime of ISA
 - May require nasty hacks to overcome (E.g., x86 segments)
- x86 evolution:
 - 4-bit (4004), 8-bit (8008), 16-bit (8086), 24-bit (80286),
 - 32-bit + protected memory (80386)
 - 64-bit (AMD’s Opteron & Intel’s Pentium4)
- All ISAs moving to 64 bits (if not already there)

LC4/MIPS/x86 Memory Size

- LC4
 - 16-bit (2^{16} 16-bit words) x 2 (split data and instruction memory)
- MIPS
 - 32-bit
 - 64-bit
- x86
 - 8086: 16-bit
 - 80286: 24-bit
 - 80386: 32-bit
 - AMD Opteron/Athlon64, Intel’s newer Pentium4, Core 2: 64-bit

How Are Memory Locations Specified?

- Registers are specified **directly**
 - Register names are short, can be encoded in instructions
 - Some instructions implicitly read/write certain registers
- How are addresses specified?
 - Addresses are as big or bigger than insns
 - **Addressing mode**: how are insn bits converted to addresses?
 - Think about: what high-level idiom addressing mode captures

Memory Addressing

- **Addressing mode**: way of specifying address
 - Used in memory-memory or load/store instructions in register ISA
- Examples
 - **Displacement**: $R1 = \text{mem}[R2 + \text{immed}]$
 - **Index-base**: $R1 = \text{mem}[R2 + R3]$
 - **Memory-indirect**: $R1 = \text{mem}[\text{mem}[R2]]$
 - **Auto-increment**: $R1 = \text{mem}[R2]$, $R2 = R2 + 1$
 - **Auto-indexing**: $R1 = \text{mem}[R2 + \text{immed}]$, $R2 = R2 + \text{immed}$
 - **Scaled**: $R1 = \text{mem}[R2 + R3 * \text{immed1} + \text{immed2}]$
 - **PC-relative**: $R1 = \text{mem}[\text{PC} + \text{imm}]$
- What high-level program idioms are these used for?
- What implementation impact? What impact on insn count?

LC4/MIPS/x86 Addressing Modes

- LC4
 - **Displacement:** R1+offset (6-bit)
 - MIPS
 - **Displacement:** R1+offset (16-bit)
 - Experiments showed this covered 80% of accesses on VAX
- | | | | | |
|--------|-------|-------|-------|-----------|
| I-type | Op(6) | Rs(5) | Rt(5) | Immed(16) |
|--------|-------|-------|-------|-----------|
- x86 (MOV instructions)
 - **Absolute:** zero + offset (8/16/32-bit)
 - **Displacement:** R1+offset (8/16/32-bit)
 - **Indexed:** R1+R2
 - **Scaled:** R1 + (R2*Scale) + offset (8/16/32-bit) Scale = 1, 2, 4, 8
 - **PC-relative:** PC + offset (32-bit)

x86 Addressing Modes

```

.LFE2
.comm array,400,32
.comm sum,4,4

.globl array_sum
array_sum:
movl $0, -4(%rbp)

.L1:
movl -4(%rbp), %eax
movl array(%eax,4), %edx
movl sum(%rip), %eax
addl %edx, %eax
movl %eax, sum(%rip)
addl $1, -4(%rbp)
cmpl $99,-4(%rbp)
jle .L1
    
```

Note: "mov" can be load, store, or reg-to-reg move

Two More Addressing Issues

- **Access alignment:** address % size == 0?
 - Aligned: `load-word @XXXX00`, `load-half @XXXXX0`
 - Unaligned: `load-word @XXXX10`, `load-half @XXXXX1`
 - Question: what to do with unaligned accesses (uncommon case)?
 - Support in hardware? Makes all accesses slow
 - Trap to software routine? Possibility
 - Use regular instructions
 - Load, shift, load, shift, and
 - **MIPS? ISA support:** unaligned access using two instructions


```
lwl @XXXX10; lwr @XXXX10
```
- **Endian-ness:** arrangement of bytes in a word
 - Big-endian: sensible order (e.g., MIPS, PowerPC)
 - A 4-byte integer: "00000000 00000000 00000010 00000011" is 515
 - Little-endian: reverse order (e.g., x86)
 - A 4-byte integer: "00000011 00000010 00000000 00000000" is 515
 - Why little endian? To be different? To be annoying? Nobody knows

How Many Explicit Register Operands

- **Operand model:** how many explicit operands
 - **3:** general-purpose
 - `add R1,R2,R3` means: $R1 = R2 + R3$ **(MIPS uses this)**
 - **2:** multiple explicit accumulators (output doubles as input)
 - `add R1,R2` means: $R1 = R1 + R2$ **(x86 uses this)**
 - **1:** one implicit accumulator
 - `add R1` means: $ACC = ACC + [R1]$
 - **4+:** useful only in special situations
 - Fused multiply & accumulate instruction
- Why have fewer?
 - Primarily code density (size of each instruction in program binary)

Operand Model: Register or Memory?

- “Load/store” architectures
 - Memory access instructions (loads and stores) are distinct
 - Separate addition, subtraction, divide, etc. operations
 - Examples: MIPS, ARM, SPARC, PowerPC
- Alternative: mixed operand model (x86, VAX)
 - Operand can be from register **or** memory
 - x86 example: `addl 100, 4(%eax)`
 - 1. Loads from memory location `[4 + %eax]`
 - 2. Adds “100” to that value
 - 3. Stores to memory location `[4 + %eax]`
 - Would requires three instructions in MIPS, for example.

LC4/MIPS/x86 Operand Models

- LC4
 - Integer: 8 general-purpose registers, load-store
 - Floating-point: none
- MIPS
 - Integer/floating-point: 32 general-purpose registers, load-store
- x86
 - Integer (8 registers) reg-reg, reg-mem, mem-reg, but no mem-mem
 - Floating point: stack (why x86 floating-point lagged for years)
 - SSE introduced 16 general purpose floating-point registers
 - Note: integer `push`, `pop` for managing software stack
 - Note: also reg-mem and mem-mem string functions in hardware
- x86-64
 - Integer/floating-point: 16 registers

x86 Operand Model: Accumulators

```
.LFE2
.comm array,400,32
.comm sum,4,4

.globl array_sum
array_sum:
    movl $0, -4(%rbp)

.L1:
    movl -4(%rbp), %eax
    movl array(,%eax,4), %edx
    movl sum(%rip), %eax
    addl %edx, %eax
    movl %eax, sum(%rip)
    addl $1, -4(%rbp)
    cmpl $99,-4(%rbp)
    jle .L1
```

- x86 uses explicit accumulators
 - Both register and memory
 - Distinguished by addressing mode

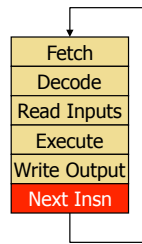
Register accumulator: `%eax = %eax + %edx`

Memory accumulator:
`Memory[%rbp-4] = Memory[%rbp-4] + 1`

Operand Model & Compiler Optimizations

- How do operand model & addressing mode affect compiler?
- Again, what does a compiler try to do?
 - Reduce insn count, reduce load/store count (important), schedule
- What features enable or limit these?
 - + (Many) general-purpose registers let you reduce stack accesses
 - Implicit operands clobber values
 - `addl %edx, %eax` destroys initial value in `%eax`
 - Requires additional insns to preserve if needed
 - Implicit operands also restrict scheduling
 - Classic example, condition codes (flags)
 - Result: you want a general-purpose register load-store ISA (MIPS)

Control Transfers



- Default next-PC is PC + sizeof(current insn)
- Branches and jumps can change that
 - Otherwise dynamic program == static program
- **Computing targets:** where to jump to
 - For all branches and jumps
 - PC-relative: for branches and jumps with function
 - Absolute: for function calls
 - Register indirect: for returns, switches & dynamic calls
- **Testing conditions:** whether to jump at all
 - For (conditional) branches only

Control Transfers I: Computing Targets

- The issues
 - How far (statically) do you need to jump?
 - Not far within procedure, further from one procedure to another
 - Do you need to jump to a different place each time?
- **PC-relative**
 - Position-independent within procedure
 - Used for branches and jumps within a procedure
- **Absolute**
 - Position independent outside procedure
 - Used for procedure calls
- **Indirect** (target found in register)
 - Needed for jumping to dynamic targets
 - Used for **returns**, dynamic procedure calls, *switch* statements

Control Transfers II: Testing Conditions

- **Compare and branch insns**
 - `branch-less-than R1,10,target`
 - + Fewer instructions
 - Two ALUs: one for condition, one for target address
 - Less room for target in insn
 - Extra latency
- **Implicit condition codes or "flags" (x86, LC4)**
 - `cmp R1,10 // sets "negative" flag`
 - `branch-neg target`
 - + More room for target in insn, condition codes often set "for free"
 - + Branch insn simple and fast
 - Implicit dependence is tricky
- **Condition registers, separate branch insns (MIPS)**
 - `set-less-than R2,R1,10`
 - `branch-not-equal-zero R2,target`
 - ± A compromise

LC4, MIPS, x86 Control Transfers

- LC4
 - 9-bit offset PC-relative branches (condition codes)
 - 11-bit offset PC-relative jumps
 - 11-bit absolute 16-byte aligned calls
- MIPS
 - 16-bit offset PC-relative conditional branches
 - **Uses register for condition**
 - Compare 2 regs: `beq`, `bne` or reg to 0: `bgtz`, `bgez`, `bltz`, `blez`
 - + Don't need adder for these, cover 80% of cases
 - Explicit condition registers: `slt`, `sltu`, `slti`, `sltiu`, etc.
 - 26-bit target absolute jumps and calls
- x86
 - 8-bit offset PC-relative branches
 - **Uses condition codes**
 - Explicit compare instructions (and others) to set condition codes

ISAs Also Include Support For...

- Function calling conventions
 - Which registers are saved across calls, how parameters are passed
- Operating systems & memory protection
 - Privileged mode
 - System call (TRAP)
 - Exceptions & interrupts
 - Interacting with I/O devices
- Multiprocessor support
 - "Atomic" operations for synchronization
- Data-level parallelism
 - Pack many values into a wide register
 - Intel's SSE2: four 32-bit float-point values into 128-bit register
 - Define parallel operations (four "adds" in one cycle)

RISC and CISC

- **RISC**: reduced-instruction set computer
 - Coined by Patterson in early 80's
 - RISC-I (Patterson), MIPS (Hennessy), IBM 801 (Cocke)
 - Examples: PowerPC, ARM, SPARC, Alpha, PA-RISC
- **CISC**: complex-instruction set computer
 - Term didn't exist before "RISC"
 - Examples: x86, VAX, Motorola 68000, etc.
- Philosophical war (one of several) started in mid 1980's
 - RISC "won" the technology battles
 - CISC won the high-end commercial war (1990s to today)
 - Compatibility a stronger force than anyone (but Intel) thought
 - RISC won the embedded computing war

The RISC vs. CISC Debate

The Context

- Pre 1980
 - Bad compilers (so assembly written by hand)
 - Complex, high-level ISAs (easier to write assembly)
 - Slow multi-chip micro-programmed implementations
 - Vicious feedback loop
- Around 1982
 - Moore's Law makes single-chip microprocessor possible...
 - **...but only for small, simple ISAs**
 - Performance advantage of this "integration" was compelling
 - Compilers had to get involved in a big way
- **RISC manifesto**: create ISAs that...
 - **Simplify single-chip implementation**
 - **Facilitate optimizing compilation**

Role of Compilers

- Who is generating assembly code?
- Humans like high-level "CISC" ISAs (close to prog. langs)
 - + Can "concretize" ("drill down"): move down a layer
 - + Can "abstract" ("see patterns"): move up a layer
 - Can deal with few things at a time → like things at a high level
- Computers (compilers) like low-level "RISC" ISAs
 - + Can deal with many things at a time → can do things at any level
 - + Can "concretize": 1-to-many lookup functions (databases)
 - Difficulties with abstraction: many-to-1 lookup functions (AI)
 - Translation should move strictly "down" levels
- Stranger than fiction
 - People once thought computers would execute prog. lang. directly

Early 1980s: The Tipping Point

- Moore's Law makes single-chip microprocessor possible...
 - ...but only for small, simple ISAs
- Performance advantage of "integration" was compelling
- **RISC manifesto**: create ISAs that...
 - Simplify implementation
 - Facilitate optimizing compilation
 - Some guiding principles ("tenets")
 - Single cycle execution/hard-wired control
 - Fixed instruction length, format
 - Lots of registers, load-store architecture
- No equivalent "CISC manifesto"

The RISC Design Tenets

- **Single-cycle execution**
 - CISC: many multicycle operations
- **Hardwired (simple) control**
 - CISC: "microcode" for multi-cycle operations
- **Load/store architecture**
 - CISC: register-memory and memory-memory
- **Few memory addressing modes**
 - CISC: many modes
- **Fixed-length instruction format**
 - CISC: many formats and lengths
- **Reliance on compiler optimizations**
 - CISC: hand assemble to get good performance
- **Many registers** (compilers are better at using them)
 - CISC: few registers

CISCs and RISCs

- The CISCs: x86, VAX (**V**irtual **A**ddress **eX**tension to PDP-11)
 - Variable length instructions: 1-321 bytes!!!
 - 14 registers + PC + stack-pointer + condition codes
 - Data sizes: 8, 16, 32, 64, 128 bit, decimal, string
 - Memory-memory instructions for all data sizes
 - Special insns: `crc`, `insque`, `polyf`, and a cast of hundreds
 - x86: "Difficult to explain and impossible to love"
- The RISCs: MIPS, PA-RISC, SPARC, PowerPC, Alpha, ARM
 - 32-bit instructions
 - 32 integer registers, 32 floating point registers, load-store
 - 64-bit virtual address space
 - Few addressing modes
 - Why so many basically similar ISAs? Everyone wanted their own

The Debate

- RISC argument
 - CISC is fundamentally handicapped
 - For a given technology, RISC implementation will be better (faster)
 - Current technology enables single-chip RISC
 - When it enables single-chip CISC, RISC will be pipelined
 - When it enables pipelined CISC, RISC will have caches
 - When it enables CISC with caches, RISC will have next thing...
- CISC rebuttal
 - CISC flaws not fundamental, can be fixed with more transistors
 - Moore's Law will narrow the RISC/CISC gap (true)
 - Good pipeline: RISC = 100K transistors, CISC = 300K
 - By 1995: 2M+ transistors had evened playing field
 - Software costs dominate, **compatibility** is paramount

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 survives
- **Backward compatibility**
 - New processors supporting old programs
 - Can't drop features (**caution in adding new ISA features**)
 - Or, 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 of features
 - Add ISA hints by overloading no-ops (example: x86's PAUSE)
 - New firmware/software on old processors to emulate new insns

Intel's Compatibility Trick: RISC Inside

- 1993: Intel wanted "out-of-order execution" in Pentium Pro
 - Hard to do with a coarse grain ISA like x86
- Solution? Translate x86 to RISC micro-ops (**μops**) in hardware

```
push $eax
becomes (we think, uops are proprietary)
store $eax, -4($esp)
addi $esp, $esp, -4
```

 - + Processor maintains **x86 ISA externally for compatibility**
 - + But executes **RISC μISA internally for implementability**
 - Given translator, x86 almost as easy to implement as RISC
 - Intel implemented "out-of-order" before any RISC company
 - "out-of-order" also helps x86 more (because ISA limits compiler)
 - Also used by other x86 implementations (AMD)
- Different **μops** for different designs
 - **Not part of the ISA specification**, not publically disclosed

Potential Micro-op Scheme

- Most instructions are a **single** micro-op
 - Add, xor, compare, branch, etc.
 - Loads example: `mov -4(%rax), %ebx`
 - Stores example: `mov %ebx, -4(%rax)`
- Each memory access adds a micro-op
 - "addl -4(%rax), %ebx" is two micro-ops (load, add)
 - "addl %ebx, -4(%rax)" is three micro-ops (load, add, store)
- Function call (CALL) – 4 uops
 - Get program counter, store program counter to stack, adjust stack pointer, unconditional jump to function start
- Return from function (RET) – 3 uops
 - Adjust stack pointer, load return address from stack, jump register
- Again, just a basic idea, micro-ops are specific to each chip

Translation and Virtual ISAs

- New compatibility interface: ISA + translation software
 - **Binary-translation**: transform static image, run native
 - **Emulation**: unmodified image, interpret each dynamic insn
 - Typically optimized with just-in-time (JIT) compilation
 - Examples: FX!32 (x86 on Alpha), Rosetta (PowerPC on x86)
 - Performance overheads reasonable (many recent advances)
- **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 first Itanium supported x86 code
 - x86 processor (comparable to Pentium) on chip
 - How PlayStation2 supported PlayStation games
 - Used PlayStation processor for I/O chip & **emulation**

Current Winner (Revenue): CISC

- x86 was first 16-bit microprocessor by ~2 years
 - IBM put it into its PCs because there was no competing choice
 - Rest is historical inertia and "financial feedback"
 - x86 is most difficult ISA to implement and do it fast but...
 - Because Intel sells the most **non-embedded** processors...
 - It has the most money...
 - Which it uses to hire more and better engineers...
 - Which it uses to maintain competitive performance ...
 - **And given competitive performance, compatibility wins...**
 - So Intel sells the most **non-embedded** processors...
 - AMD as a competitor keeps pressure on x86 performance
- Moore's law has helped Intel in a big way
 - Most engineering problems can be solved with more transistors

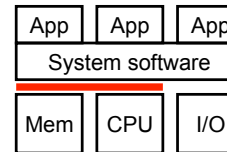
Current Winner (Volume): RISC

- ARM (Acorn RISC Machine → Advanced RISC Machine)
 - First ARM chip in mid-1980s (from Acorn Computer Ltd).
 - 3 billion units sold in 2009 (>60% of all 32/64-bit CPUs)
 - Low-power and **embedded** devices (phones, for example)
 - Significance of embedded? ISA Compatibility less powerful force
- 32-bit RISC ISA
 - 16 registers, PC is one of them
 - Many addressing modes, e.g., auto increment
 - Condition codes, each instruction can be conditional
- Multiple implementations
 - X-scale (design was DEC's, bought by Intel, sold to Marvel)
 - Others: Freescale (was Motorola), Texas Instruments, STMicroelectronics, Samsung, Sharp, Philips, etc.

Redux: Are ISAs Important?

- Does “quality” of ISA actually matter?
 - Not for performance (mostly)
 - Mostly comes as a design complexity issue
 - Insn/program: everything is compiled, compilers are good
 - Cycles/insn and seconds/cycle: μ ISA, many other tricks
 - What about power efficiency? *Maybe*
 - ARMs are most power efficient today...
 - ...but Intel is moving x86 that way (e.g, Intel’s Atom)
 - **Open question: can x86 be as power efficient as ARM?**
- Does “nastiness” of ISA matter?
 - Mostly no, only compiler writers and hardware designers see it
- Even compatibility is not what it used to be
 - Software emulation
 - **Open question: will “ARM compatibility” be the next x86?**

Summary



- What is an ISA?
 - A functional contract
- All ISAs similar in high-level ways
 - But many design choices in details
 - Two “philosophies”: CISC/RISC
 - Difference is blurring
- Good ISA...
 - Enables high-performance
 - At least doesn’t get in the way
- Compatibility is a powerful force
 - Tricks: binary translation, μ ISAs