ECE 250 / CS 250
Computer Architecture

Instruction Set Architecture

Benjamin Lee

Slides based on those from Alvin Lebeck, Daniel Sorin, Andrew Hilton, Amir Roth, Gershon Kedem

Admin

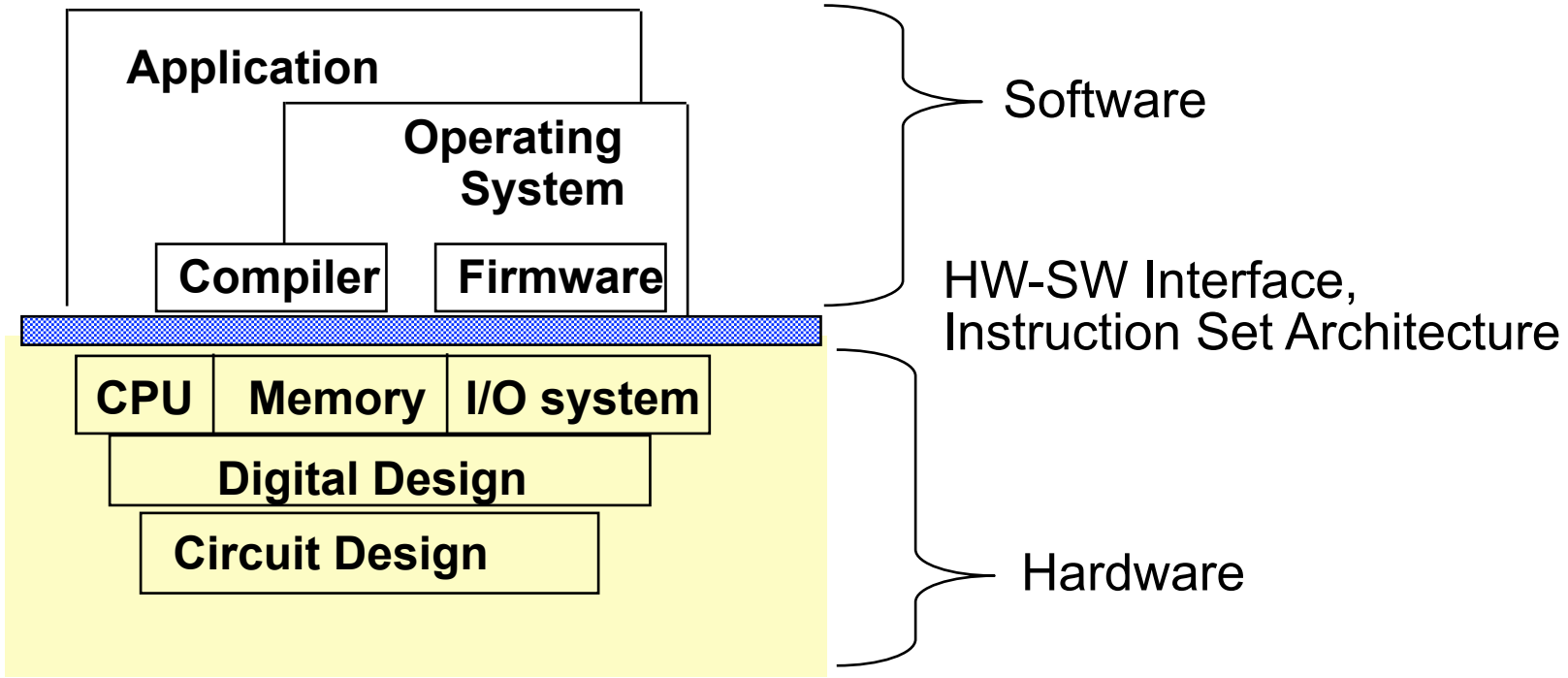
- Homework #1
  - Due this Thursday, Jan 30 @ 11:55pm
- Today's Lecture
  - Operations provided by the machine (the CPU)
  - From high level to instructions
  - Types of Instruction Sets
- Reading
  - Ch 1 -- Introduction, data representations, memory
  - Ch 2 – Instruction sets and assembly programming
  - An overview of SPIM and the MIPS32 instruction set @ http://spimsimulator.sourceforge.net/HP_AppA.pdf

Review: Memory & Bit Operations

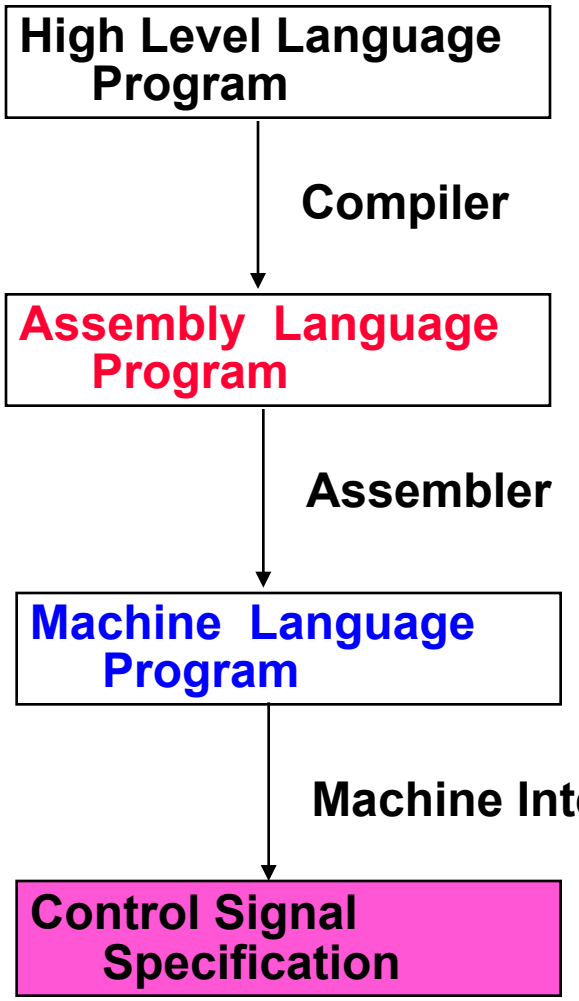
- Computer memory is linear array of bytes
- Pointer is memory location that contains address of another memory location
- Data representations
  - char 1 byte
  - int 4 bytes
  - float 4 bytes
  - double 8 bytes
- Bitwise operations (&, |, ^, ~, <<, >>)

- Code examples are linked to course web page (docs)

Instruction Set Architecture



Levels of Representation




```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw $15,   0($2)
lw $16,   4($2)
sw$16,   0($2)
sw$15,   4($2)
```

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

Transistors turning on and off

What is an ISA?

- Instruction Set Architecture (ISA)
  - The "contract" between software and hardware
  - If software does X, hardware promises to do Y
    - **Functional definition** of operations, modes, storage locations supported by hardware
    - **Precise description** of how software invokes, accesses them
  - Strictly speaking, ISA is the architecture, i.e., the interface between the hardware and the software
  - Less strictly speaking, when people talk about architecture, they're also talking about how the the architecture is implemented

Requirements for ISA

```
#include <stdio.h>

main()
{
  int *a = new int[100];
  int *p = a;
  int k;

  for (k = 0; k < 100; k++)
    {
      *p = k;
      p++;
    }

  printf("entry 3 = %d\n", a[3]);
}
```

What primitive operations do we need?

What should be implemented in hardware?

How Would You Design an ISA?

- What interface should hardware present to software?
  - Types of instructions?
  - Instruction representation?
  - Change in control flow?
  - View of storage? Where do variables live?
  - Does the hardware help to support function/method calls?
  - If so, how?
  - Should the hardware support other features that are specific to certain HLLs (e.g., garbage collection for Java)?

Microarchitecture

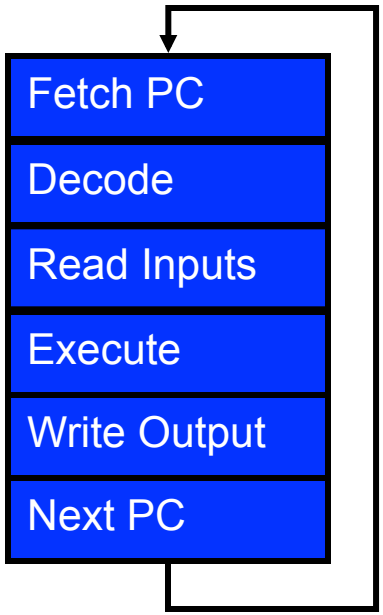
- ISA specifies what hardware does, not how it does it
  - No guarantees regarding…
    - How operations are implemented
    - Which operations are fast and which are slow
    - Which operations take more power and which take less
  - These issues are determined by the microarchitecture
    - Microarchitecture = how hardware implements architecture
    - Any number of microarchitectures can implement the same architecture
    - Pentium and Pentium 4 are almost the same architecture, but are very different microarchitectures

Aspects of ISAs

- We will discuss the following aspects of ISAs
  1. The Von Neumann (pronounced NOY-muhn) model
     - Stored-program computer
     - Implicit structure of all modern ISAs
  2. Format
     - Length and encoding
  3. Operations
  4. Operands
     - Where are operands stored and how to address them?
  5. Datatypes
  6. Control

- Example with MIPS instruction set

(1) The Sequential (Von Neumann) Model



- Implicit model of all modern ISAs
  - Often called Von Neumann, but in ENIAC before
- Program counter (PC)
  - Defines total order of dynamic instructions
    - Next PC is PC++ unless insn says otherwise
  - Instruction order, named storage define computation
    - Value flows from insn X to Y via storage A…
    - X names A as output, Y names A as input…
    - Insn Y after X
  - Processor implicitly executes loop at left
  - Instruction execution is assumed atomic
  - Instruction X finishes before insn X+1 starts

(2) Instruction Format

- Length
  1. Fixed length
     - 8, 16, 32 or 64 bits (depends on architecture)
     + Simple implementation: compute next PC using only this PC
     – Code density: 32 or 64 bits for a NOP (no operation)?
  2. Variable length
     – Complex implementation
     + Code density
  3. Compromise: two lengths
     - Example: $MIPS_{16}$
- Encoding
  - A few simple encodings simplify decoder implementation

(3) Operations

- Operation type encoded in instruction opcode
- Many types of operations
  - Integer arithmetic: add, sub, mul, div, mod/rem (signed/unsigned)
  - FP arithmetic: add, sub, mul, div, sqrt
  - Integer logical: and, or, xor, not, sll, srl, sra
- What other operations might be useful?
- More operation types == better ISA??
- DEC VAX computer had **many** operation types
  - Example: instruction for polynomial evaluation
  - But many of them were rarely/never used. Why?

(4) Operations Act on Operands

- If you're going to add, you need at least 3 operands
  - Two source operands, one destination operand
  - Note: operands don't have to be unique (e.g., A = B + A)

- Question #1: Where do operands come from?

- Question #2: How are they specified?

Basic ISA Classes

Accumulator:

1 address	add A	acc ← acc + mem[A]
1+x address	addx A	acc ← acc + mem[A + x]

Stack:

0 address	add	tos ← tos + next

General Purpose Register:

2 address	add A B	A ← A + B
3 address	add A B C	A ← B + C

Load/Store:

3 address	add Ra Rb Rc	Ra ← Rb + Rc
	load Ra Rb	Ra ← mem[Rb]
	store Ra Rb	mem[Rb] ← Ra

Accumulator

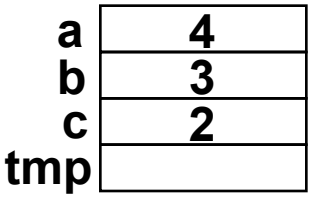
- Instruction set: Accumulator is implicit operand
  - one explicit operand
  - add, sub, mult, div, . . .
  - clear, store (st)

Example: a*b - (a+c*b)

clear	**0**
add c	**2**
mult b	**6**
add a	**10**
st tmp	**10**
clear	**0**
add a	**4**
mult b	**12**
sub tmp	**2**

9 instructions

Memory



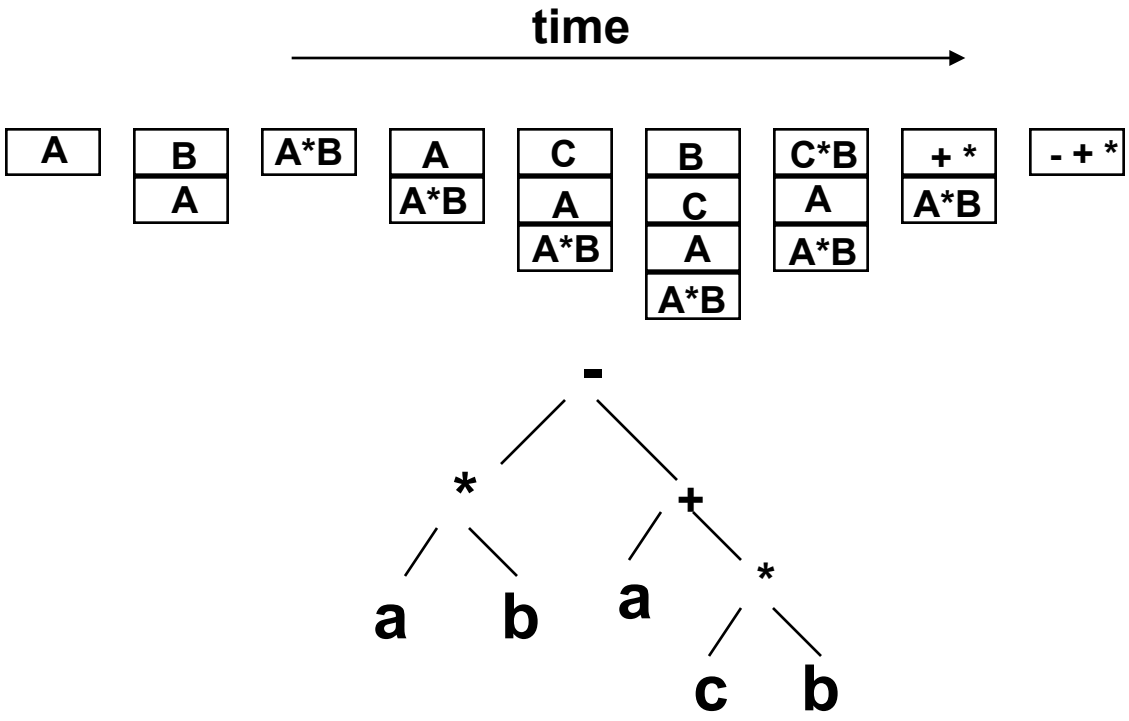
Stack Instruction Set Architecture

- Instruction set

add, sub, mult, div	Top of stack (TOS) and TOS+1 are implicit
push A, pop A	TOS is implicit operand, one explicit operand

Example: a*b - (a+c*b)

push a
push b
mult
push a
push c
push b
mult
add
sub
9 instructions



2-address ISA

- Instruction set: two explicit operands, one implicit
  - add, sub, mult, div, …
  - one source operand is also destination
    - add a,b a <- a + b

Example: a*b - (a+c*b)

	tmp1, tmp2
add tmp1, b	3, ?
mult tmp1, c	6, ?
add tmp1, a	10, ?
add tmp2, b	10, 3
mult tmp2, a	10, 12
sub tmp2, tmp1	10, 2
6 instructions	

Memory

a	**4**
b	**3**
c	**2**
tmp1	
tmp2	

3-address ISA

- Instruction set: Three explicit operands

  add, sub, mult, div, …

  add a,b,c a <- b + c

Example: a*b - (a+c*b)

	tmp1, tmp2
mult tmp1, b, c	**6, ?**
add tmp1, tmp1, a	**10, ?**
mult tmp2, a, b	**10, 12**
sub tmp2, tmp2, tmp1	**10, 2**

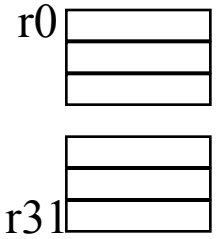
4 instructions

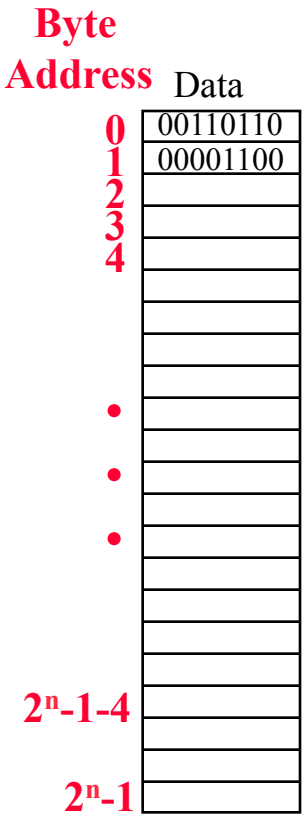
a	**4**
b	**3**
c	**2**
tmp1	
tmp2	

Adding Registers to an ISA

- Registers hold values.
- Registers are named within the instruction

- Like memory, but much smaller
  - 32-128 locations
- How many bits to specify a register?





3-address General Pupose Register ISA

- Instruction set: Three explicit operands

  add, sub, mult, div, …

  add a,b,c a <- b + c

Example: a*b - (a+c*b)

	r1,	r2
mult r1, b, c	6,	?
add r1, r1, a	10,	?
mult r2, a, b	10,	12
sub r2, r2, r1	10,	2

4 instructions

a	4
b	3
c	2

LOAD / STORE ISA

- Instruction set:

  add, sub, mult, div — only for operands in registers

  ld, st, — to move data from/to mem

Example: a*b - (a+c*b)

	r1,	r2,	r3
ld r1, c	2,	?,	?
ld r2, b	2,	3,	?
mult r1, r1, r2	6,	3,	?
ld r3, a	6,	3,	4
add r1, r1, r3	10,	3,	4
mult r2, r2, r3	10,	12,	4
sub r3, r2, r1	10,	12,	2

7 instructions

a	4
b	3
c	2

Using Registers for Pointer Access

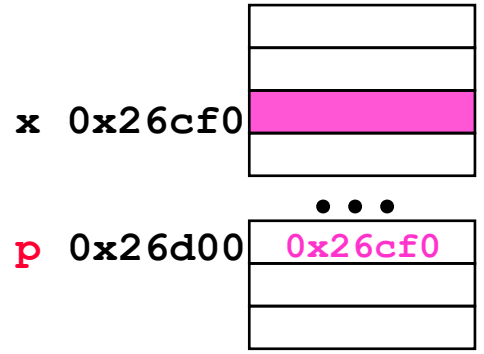
- Registers can hold memory addresses

Given

```
int x; int *p;
p = &x;
*p = *p + 8;
```

Instructions

```
ld r1, p          // r1 <- mem[p]
ld r2, r1         // r2 <- mem[r1]
add r2, r2, 0x8   // increment x by 8
st r1, r2         // mem[r1] <- r2
```



- Many different ways to address operands

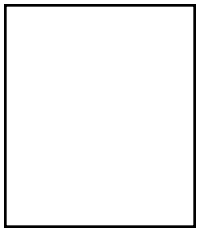
Operand Addressing Modes

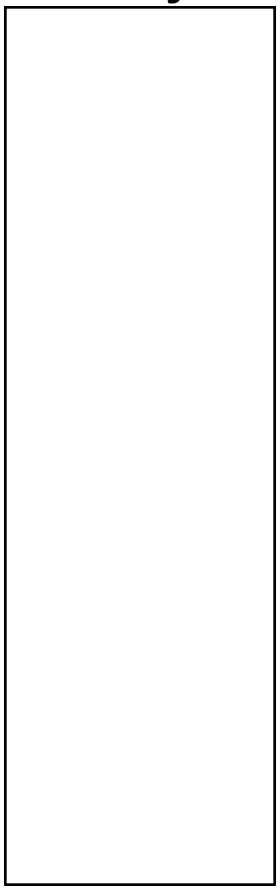
- Register direct Ri
- Immediate (literal) v
- Direct (absolute) M[v]

- Register indirect M[Ri]
- Base+Displacement M[Ri + v]
- Base+Index M[Ri + Rj]
- Scaled Index M[Ri + Rj*d + v]
- Autoincrement M[Ri++]
- Autodecrement M[Ri - -]

- Memory Indirect M[M[Ri]]

Instruction Fields





Making Instructions Machine Readable

- So far, still too abstract
  - add r1, r2, r3
- Need to specify instructions in machine readable form
  - Bunch of Bits
- Instructions are bits with well defined fields
  - Like a floating point number has different fields
- Instruction Format
  - establishes a mapping from "instruction" to binary values
  - which bit positions correspond to which parts of the instruction (operation, operands, etc.)

A Typical RISC Instruction Set

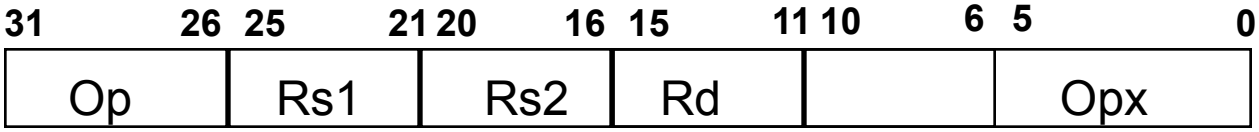
- RISC – reduced instruction set computer

- 3-address, reg-reg arithmetic instruction
- 32-bit fixed format instruction (3 formats)

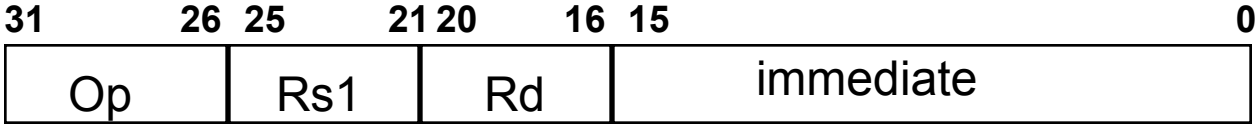
- 32 64-bit general-purpose registers (R0 contains zero)
- Single address mode for load/store
  - base + displacement

Example: MIPS

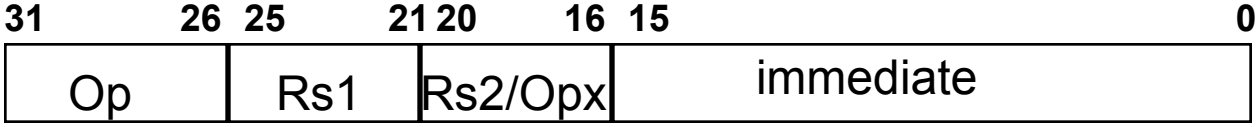
Register-Register



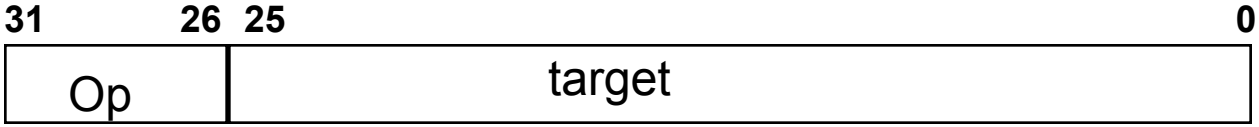
Register-Immediate



Branch



Jump / Call



Summary

- Instruction Set Architecture is bridge between Software and the Processor (CPU)
- Many different possibilities
  - Accumulator
  - Stack
  - General-purpose registers
  - Load-store

Reading

- An overview of SPIM and the MIPS32 instruction set @ http://spimsimulator.sourceforge.net/HP_AppA.pdf