Instruction Set Architecture (ISA)

CPS 104
Lecture 4

Administrivia

• Homework #1 Due Today
• Homework #2 Assigned
  ➢ Due Feb 4
  ➢ Part 1 in class
  ➢ Part 2 midnight (submit)
Today's Lecture

• Operations provided by the machine

Outline
• Review
• From high level to instructions
• Types of Instruction Sets

Reading
Chapter 3

Review: Computer Memory

• Memory is a large linear array of bytes.
  ➢ Each byte has a unique address (location).
  ➢ Byte of data at address 0x100, and 0x101
• Most computers support byte (8-bit) addressing.
• Data may have to be aligned on word (4 byte) or double word (8 byte) boundary.
  ➢ int is 4 bytes
  ➢ double precision floating point is 8 bytes
• 32-bit v.s. 64-bit addresses
  ➢ we will assume 32-bit for rest of course, unless otherwise stated
Review: A Simple Program’s Memory Layout

```c
... int result;
main()
{
    int *x;
    ...
    result = x + result;
    ...
}
mem[0x208] = mem[0x400] + mem[0x208]
```

Review: Pointers

- A pointer is a memory location that contains the address of another memory location
- "address of" operator &
  ➢ don’t confuse with reference operator, or bitwise AND operator (later today)

Given

```c
int x; int *p;
p = &x;
```

Then

```
*p = 2; and x = 2; produce the same result
```

On 32-bit machine, p is 32-bits

```
x 0x26cf0
p 0x26d00
```
Review: Bitwise Operators

- `&` is AND, `|` is OR
- `>>` is shift right, `<<` is shift left,

\[
x_{\text{pos}} = 0x1a34c \& 0x000ff
y_{\text{pos}} = (0x1a34c \& 0x0ff00) >> 8
button = (0x1a34c \& 0x30000) >> 16
\]

<table>
<thead>
<tr>
<th>button</th>
<th>y</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1a34c = 01</td>
<td>1010</td>
<td>0011</td>
</tr>
<tr>
<td>0x000ff = 00</td>
<td>1111</td>
<td>1111</td>
</tr>
<tr>
<td>0x0a300 = 00</td>
<td>1010</td>
<td>0011</td>
</tr>
</tbody>
</table>

Instruction Set Architecture

- Application
- Operating System
- Compiler
- Firmware
- CPU
- Memory
- I/O system
- Digital Design
- Circuit Design

Software

Interface Between HW and SW

Instruction Set Architecture, Memory, I/O

Hardware
Levels of Representation

- High Level Language Program
- Assembly Language Program
- Machine Language Program
- Control Signal Specification

Compiler
Assembler
Machine Interpretation

```
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

Computer Architecture?

... the attributes of a [computing] system as seen by the programmer, i.e. the conceptual structure and functional behavior, as distinct from the organization of the data flows and controls the logic design, and the physical implementation.

Amdahl, Blaaw, and Brooks, 1964
Towards Evaluation of ISA and Organization

What primitive operations do we need?
(i.e., What should be implemented in hardware?)

Requirements for ISA

```cpp
#include <iostream.h>

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

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

    cout << "entry 3 = " << a[3] << endl;
}
```
Design Space of ISA

Five Primary Dimensions
- Operations: add, sub, mul, ...
- Number of explicit operands: (0, 1, 2, 3)
- Operand Storage: Where besides memory?
- Memory Address: How is memory location specified?
- Type & Size of Operands: byte, int, float, vector, ...

Other Aspects
- Successor instruction: How is it specified?
- Conditions: How are they determined?
- Encodings: Fixed or variable? Wide?
- Parallelism

Interface Design

A good interface:
- Lasts through many implementations (portability, compatibility)
- Is used in many different ways (generality)
- Provides convenient functionality to higher levels
- Permits an efficient implementation at lower levels
ISA Metrics

- Aesthetics:
- Regularity (Orthogonality)
  - No special registers, few special cases, all operand modes available with any data type or instruction type
- Primitives not solutions
- Completeness
  - Support for a wide range of operations and target applications
- Streamlined
  - Resource needs easily determined
- Ease of compilation (programming?)
- Ease of implementation
- Scalability

Basic ISA Classes

Accumulator:

- 1 address: add A \( \text{acc} \leftarrow \text{acc} + \text{mem}[A] \)
- 1+x address: addx A \( \text{acc} \leftarrow \text{acc} + \text{mem}[A + x] \)

Stack:

- 0 address: add \( \text{tos} \leftarrow \text{tos} + \text{next} \) (JAVA VM)

General Purpose Register:

- 2 address: add A B \( A \leftarrow A + B \)
- 3 address: add A B C \( A \leftarrow B + C \)

Load/Store:

- 3 address: add Ra Rb Rc \( Ra \leftarrow Rb + Rc \)
  - load Ra Rb \( Ra \leftarrow \text{mem}[Rb] \)
  - store Ra Rb \( \text{mem}[Rb] \leftarrow Ra \)
Accumulator

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

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

```

<table>
<thead>
<tr>
<th>clear</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>add c</td>
<td>2</td>
</tr>
<tr>
<td>mult b</td>
<td>6</td>
</tr>
<tr>
<td>add a</td>
<td>10</td>
</tr>
<tr>
<td>st tmp</td>
<td>10</td>
</tr>
<tr>
<td>clear</td>
<td>0</td>
</tr>
<tr>
<td>add a</td>
<td>4</td>
</tr>
<tr>
<td>mult b</td>
<td>12</td>
</tr>
<tr>
<td>sub tmp</td>
<td>2</td>
</tr>
<tr>
<td><strong>9 instructions</strong></td>
<td></td>
</tr>
</tbody>
</table>
```

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:** \( a \cdot b - (a + c \cdot b) \)

```

<table>
<thead>
<tr>
<th>push a</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>push b</td>
<td></td>
</tr>
<tr>
<td>mult</td>
<td></td>
</tr>
<tr>
<td>push a</td>
<td></td>
</tr>
<tr>
<td>push c</td>
<td></td>
</tr>
<tr>
<td>push b</td>
<td></td>
</tr>
<tr>
<td>mult</td>
<td></td>
</tr>
<tr>
<td>add</td>
<td></td>
</tr>
<tr>
<td>sub</td>
<td></td>
</tr>
<tr>
<td><strong>9 instructions</strong></td>
<td></td>
</tr>
</tbody>
</table>
```
2-address ISA

- **Instruction set**: Two explicit operands, one implicit
  - `add, sub, mult, div, ...`
  - one source operand is also destination
  - `add a,b` \( \Rightarrow a \leftarrow a + b \)

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

<table>
<thead>
<tr>
<th></th>
<th>\text{tmp1, tmp2}</th>
<th>Memory</th>
<th>\text{tmp1, tmp2}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{add tmp1, b}</td>
<td>3, ?</td>
<td>a</td>
<td>4</td>
</tr>
<tr>
<td>\text{mult tmp1, c}</td>
<td>6, ?</td>
<td>b</td>
<td>3</td>
</tr>
<tr>
<td>\text{add tmp1, a}</td>
<td>10, ?</td>
<td>c</td>
<td>2</td>
</tr>
<tr>
<td>\text{add tmp2, b}</td>
<td>10, 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\text{mult tmp2, a}</td>
<td>10, 12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\text{sub tmp2, tmp1}</td>
<td>10, 2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6 instructions

3-address ISA

- **Instruction set**: Three explicit operands, ZERO implicit
  - `add, sub, mult, div, ...`
  - `add a,b,c` \( \Rightarrow a \leftarrow b + c \)

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

<table>
<thead>
<tr>
<th></th>
<th>\text{tmp1, tmp2}</th>
<th>Memory</th>
<th>\text{tmp1, tmp2}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{mult tmp1, b, c}</td>
<td>6, ?</td>
<td>a</td>
<td>4</td>
</tr>
<tr>
<td>\text{add tmp1, tmp1, a}</td>
<td>10, ?</td>
<td>b</td>
<td>3</td>
</tr>
<tr>
<td>\text{mult tmp2, a, b}</td>
<td>10, 12</td>
<td>c</td>
<td>2</td>
</tr>
<tr>
<td>\text{sub tmp2, tmp2, tmp1}</td>
<td>10, 2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4 instructions
Adding Registers to an ISA

- A place to hold values that can be named within the instruction
- Like memory, but much smaller
  - 32-128 locations
- How many bits to specify a register?

3-address General Purpose Register ISA

- Instruction set: Three explicit operands, ZERO implicit
  - add, sub, mult, div, ...
  - add a,b,c  a <- b + c

Example: a*b - (a+c*b) (assume all in registers)

<table>
<thead>
<tr>
<th></th>
<th>r1, r2</th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>mult</td>
<td>r1, b, c</td>
<td>6, ?</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>r1, r1, a</td>
<td>10, ?</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>mult</td>
<td>r2, a, b</td>
<td>10, 12</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>sub</td>
<td>r2, r2, r1</td>
<td>10, 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4 instructions</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
LOAD / STORE ISA

• Instruction set:
  add, sub, mult, div, … only on operands in registers
  ld, st, to move data from and to memory, only way to access memory

Example: $a \times b - (a + c \times b)$ (assume in memory)

```
ld r1, c 2, ?, ?  
ld r2, b 2, 3, ?  
mult r1, r1, r2 6, 3, ?  
lr 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

Using Registers to Access Memory

• Registers can hold memory addresses

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

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

```
x  0x26c0  
p  0x26d0
```

• Many different ways to address operands
  ➢ not all Instruction sets include all modes
Kinds of 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 \( M[M[Ri]] \)

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

- 32-bit fixed format instruction (3 formats)
- 32 64-bit GPR (R0 contains zero)
- 3-address, reg-reg arithmetic instruction
- Single address mode for load/store: base + displacement
  - no indirection

see: SPARC, MIPS, MC88100, AMD29000, i960, i860
PARisc, POWERPC, DEC Alpha, Clipper,
CDC 6600, CDC 7600, Cray-1, Cray-2, Cray-3

Example: MIPS

Register-Register

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs1</td>
<td>Rs2</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Opx</td>
</tr>
</tbody>
</table>

Register-Immediate

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs1</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>immediate</td>
</tr>
</tbody>
</table>

Branch

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs1</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>immediate</td>
</tr>
</tbody>
</table>

Jump / Call

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td></td>
<td></td>
<td>target</td>
</tr>
</tbody>
</table>

© Alvin R. Lebeck
Evolution of Instruction Sets

• Major advances in computer architecture are typically associated with landmark instruction set designs
  - Ex: Stack vs GPR (System 360)

• Design decisions must take into account:
  - technology
  - machine organization
  - programming languages
  - compiler technology
  - operating systems

• And they in turn influence these
Summary

• Instruction Set Architecture is bridge between Software and the Processor (CPU)
• Many different possibilities
  ➢ accumulator
  ➢ stack
  ➢ GPR
  ➢ LD/ST

Next Time
• MIPS Instruction Set

Reading
• Chapter 3