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

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
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 0x26c00
*p 0x26d00

Review: Bitwise Operators

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

\[ \text{xpos} = \text{0x1a34c} \& \text{0x000ff} \]
\[ \text{ypos} = (\text{0x1a34c} \& \text{0x0ff00}) \gg 8 \]
\[ \text{button} = (\text{0x1a34c} \& \text{0x30000}) \gg 16 \]

\[ \text{button} = \text{y} \times \text{x} \]
\[ \text{0x1a34c} = 01 1010 0011 0100 1100 \]
\[ \text{0x000ff} = 00 1111 1111 0000 0000 \]
\[ \text{0x0a300} = 00 1010 0110 0000 0000 \]

Levels of Representation

- High Level Language Program
  - Compiler
  - Assembly Language Program
    - Assembler
  - Machine Language Program
    - Machine Interpretation

Transistors turning on and off

Towards Evaluation of ISA and Organization

software

hardware

Software

Hardware

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

Requirements for ISA

```c
#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 it 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?

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 acc ← acc + mem[A]
1+X address
addx A acc ← acc + mem[A + X]

Stack:

9 address
add tos ← tos + next (JAVA VM)

General Purpose Register:

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

Load/Store:

3 address
load A B Ra Ra ← mem[Rb]
store A B Ra mem[Rb] ← Ra

Accumulator

- Instruction set: Accumulator is implicit operand
  - one explicit operand
  - add, sub, mul, . . .

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>clear</td>
<td>0</td>
</tr>
<tr>
<td>add a</td>
<td>0</td>
</tr>
<tr>
<td>add b</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>add b</td>
<td>12</td>
</tr>
<tr>
<td>sub tmp</td>
<td>2</td>
</tr>
</tbody>
</table>

9 instructions

Stack Instruction Set Architecture

- Instruction set: add, sub, mult, . . . Top of stack (TOS) and TOS+1 are implicit
- TOSS is implicit operand, one explicit operand

Example: \( a \cdot b - (a+c\cdot b) \)
### 2-address ISA

- **Instruction set:** Two explicit operands, one implicit
- `add`, `sub`, `mult`, `div`, ...
- One source operand is also destination

**Example:**

\[
a \cdot b - (a + c \cdot b)
\]

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add tmp1, b</code></td>
<td><code>b</code></td>
</tr>
<tr>
<td><code>mult tmp1, c</code></td>
<td><code>6, 7</code></td>
</tr>
<tr>
<td><code>add tmp1, a</code></td>
<td><code>10, 7</code></td>
</tr>
<tr>
<td><code>mult tmp2, b</code></td>
<td><code>10, 3</code></td>
</tr>
<tr>
<td><code>sub tmp2, tmp1</code></td>
<td><code>10, 2</code></td>
</tr>
</tbody>
</table>

6 instructions

### 3-address ISA

- **Instruction set:** Three explicit operands, ZERO implicit
- `add`, `sub`, `mult`, `div`, ...
  - `add a, b, c` → `a = b + c`

**Example:**

\[
a \cdot b - (a + c \cdot b)
\]

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mult tmp1, b, c</code></td>
<td><code>6, 7</code></td>
</tr>
<tr>
<td><code>add tmp1, tmp1, a</code></td>
<td><code>10, 7</code></td>
</tr>
<tr>
<td><code>mult tmp2, a, b</code></td>
<td><code>10, 12</code></td>
</tr>
<tr>
<td><code>sub tmp2, tmp1</code></td>
<td><code>10, 2</code></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?**

- `r0` to `r31`
  - `2^{31}` locations
  - 31 bits

### 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:**

\[
a \cdot b - (a + c \cdot b)
\]

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mult r1, b, c</code></td>
<td><code>6, 7</code></td>
</tr>
<tr>
<td><code>add r1, r1, a</code></td>
<td><code>10, 7</code></td>
</tr>
<tr>
<td><code>mult r2, a, b</code></td>
<td><code>10, 12</code></td>
</tr>
<tr>
<td><code>sub r2, r2, r1</code></td>
<td><code>10, 2</code></td>
</tr>
</tbody>
</table>

4 instructions

### 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:**

\[
a \cdot b - (a + c \cdot b)
\]

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ld r1, c</code></td>
<td><code>2, 7, 7</code></td>
</tr>
<tr>
<td><code>ld r2, b</code></td>
<td><code>2, 3, 7</code></td>
</tr>
<tr>
<td><code>mult r1, r1, r2</code></td>
<td><code>6, 3, 7</code></td>
</tr>
<tr>
<td><code>ld r3, a</code></td>
<td><code>6, 3, 4</code></td>
</tr>
<tr>
<td><code>add r1, r1, r3</code></td>
<td><code>10, 3, 4</code></td>
</tr>
<tr>
<td><code>mult r2, r2, r3</code></td>
<td><code>10, 12, 4</code></td>
</tr>
<tr>
<td><code>sub r2, r2, r1</code></td>
<td><code>10, 12, 2</code></td>
</tr>
</tbody>
</table>

7 instructions

### Using Registers to Access Memory

- Registers can hold memory addresses

**Given**

\[
\begin{align*}
\text{int } x & \text{; int* } p; \\
\text{p} & \text{ = & x; }
\end{align*}
\]

**Instructions**

\[
\begin{align*}
\text{ld r1, p} & \text{; } r1 \leftarrow \text{mem[p]}
\text{ld r2, r1} & \text{; } r2 \leftarrow r1 \\
\text{add r2, r2, 0x8} & \text{; incremental by 8}
\text{st r1, r2} & \text{; mem[r1] <- r2}
\end{align*}
\]

**x 0x26c9f0**

**p 0x26d00**

- 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]
- Autodecrement M[Ri -]
- Memory Indirect M[M[Ri]]

Making Instructions MachineReadable

- 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

Example: MIPS

<table>
<thead>
<tr>
<th>Format</th>
<th>Op</th>
<th>Rs1</th>
<th>Rd</th>
<th>Rs2</th>
<th>Immediate</th>
<th>Opx</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register-Register</td>
<td>31</td>
<td>26</td>
<td>25</td>
<td>21</td>
<td>20</td>
<td>16</td>
</tr>
<tr>
<td>Register-Immediate</td>
<td>31</td>
<td>26</td>
<td>25</td>
<td>21</td>
<td>20</td>
<td>16</td>
</tr>
<tr>
<td>Branch</td>
<td>31</td>
<td>26</td>
<td>25</td>
<td>21</td>
<td>20</td>
<td>16</td>
</tr>
<tr>
<td>Jump / Call</td>
<td>31</td>
<td>26</td>
<td>25</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Evolution of Instruction Sets

Single Accumulator (EDSAC 1950)
Accumulator + Index Registers
(Manchester Mark I, IBM 700 series 1953)
Separation of Programming Model from Implementation
High-level Language Based
(B5000 1963)
Concept of a Family
(IBM 360 1964)
General Purpose Register Machines
Complex Instruction Sets
(Vax, Intel 432 1977-80)
Load/Store Architecture
(CDC 6600, Cray 1 1963-76)
RISC
(Mips, Sparc, 88000, IBM RS6000, . . . 1987)
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