Homework #1

Question?
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

Compiler

Assembly Language Program

Assembler

Machine Language Program

Machine Interpretation

Control Signal Specification

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)

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
#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, . . .
  How is it specified?

• 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, . . .
  How is it specified?

Other Aspects

• Successor instruction
  How is it specified?

• Conditions
  How are they determined?

• Encoding
  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:
0 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 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

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

- 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

<table>
<thead>
<tr>
<th>time</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>tmp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
Stack Machines

- 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:** $a \times b - (a + c \times b)$

<table>
<thead>
<tr>
<th>Operation</th>
<th>tmp1, tmp2</th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add tmp1, b</code></td>
<td>3, ?</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>mult tmp1, c</code></td>
<td>6, ?</td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td><code>add tmp1, a</code></td>
<td>10, ?</td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td><code>add tmp2, b</code></td>
<td>10, 3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>mult tmp2, a</code></td>
<td>10, 12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>sub tmp2, tmp1</code></td>
<td>10, 2</td>
<td></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`  \( a \leftarrow b + c \)

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

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

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?

![Diagram showing byte address and data with registers r0 and r31]
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:** \( a \times b - (a + c \times b) \)

```
<table>
<thead>
<tr>
<th></th>
<th>r1</th>
<th>r2</th>
</tr>
</thead>
<tbody>
<tr>
<td>mult r1, b, c</td>
<td>6,</td>
<td>?</td>
</tr>
<tr>
<td>add r1, r1, a</td>
<td>10,</td>
<td>?</td>
</tr>
<tr>
<td>mult r2, a, b</td>
<td>10,</td>
<td>12</td>
</tr>
<tr>
<td>sub r2, r2, r1</td>
<td>10,</td>
<td>2</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: \(a \cdot b - (a \cdot c \cdot b)\)

- \(\text{ld } r1, c\)
  - 2, ?, ?
- \(\text{ld } r2, b\)
  - 2, 3, ?
- \(\text{mult } r1, r1, r2\)
  - 6, 3, ?
- \(\text{ld } r3, a\)
  - 6, 3, 4
- \(\text{add } r1, r1, r3\)
  - 10, 3, 4
- \(\text{mult } r2, r2, r3\)
  - 10, 12, 4
- \(\text{sub } r3, r2, r1\)
  - 10, 12, 2

7 instructions
Using Registers to Access Memory

- Registers can hold memory addresses

Given

```c
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
  - "not all Instruction sets include all modes"
Kinds of Addressing Modes

- Register direct \( R_i \)
- Immediate (literal) \( v \)
- Direct (absolute) \( M[v] \)
- Register indirect \( M[R_i] \)
- Base+Displacement \( M[R_i + v] \)
- Base+Index \( M[R_i + R_j] \)
- Scaled Index \( M[R_i + R_j \times d + v] \)
- Autoincrement \( M[R_i++] \)
- Autodecrement \( M[R_i - -] \)
- Memory Indirect \( M[M[R_i]] \)
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, AMD2900, 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 20</th>
<th>16</th>
<th>15</th>
<th>11 10</th>
<th>6 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>Opx</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Register-Immediate

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21 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>immediate</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Branch

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21 20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs1</td>
<td>Rs2/Opx</td>
<td>immediate</td>
<td></td>
<td></td>
<td></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>target</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