Instruction Set Architecture (ISA)

Computer Science 104
Lecture 6
Admin

• Homework #1 Due Monday!
  – Solutions posted after late days “timeout”
  – TAs should give fast grading turnaround

• Reading: Chapter 2

• Midterm: Feb 15
  – Will post practice midterm + solutions soon
Last time…

- Who can remind us what we did last time?
Last time…

• Who can remind us what we did last time?
  – Finished up C
    Pointers: variables which hold address of other data
    Pretty low level programming
  – Debugging
    Scientific Method in action
    gdb
Now: Moving “downwards”

• C is pretty low level
  – But compiled into assembly…

• Now: moving down to lower levels:
  – How does software communicate with hardware?
  – What does assembly look like?
  – What variations can there be?
  – Advantages/disadvantages?
Instruction Set Architecture

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

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)

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
# Requirements for ISA

```c
#include <stdio.h>

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

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

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

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

• 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

![Diagram showing the usage of interface and its implementations]

© Andrew Hilton / Alvin R. Lebeck
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, store (st)

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

```
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>clear</td>
<td>0</td>
</tr>
<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>
</tbody>
</table>
```

9 instructions

**Memory**

```
<table>
<thead>
<tr>
<th></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 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: \(a \times b - (a + c \times b)\)

```
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:** \( a^*b - (a+c^*b) \)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>tmp1, tmp2</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>add tmp1, b</td>
<td>3, ?</td>
<td>a 4</td>
</tr>
<tr>
<td>mult tmp1, c</td>
<td>6, ?</td>
<td>b 3</td>
</tr>
<tr>
<td>add tmp1, a</td>
<td>10, ?</td>
<td>c 2</td>
</tr>
<tr>
<td>add tmp2, b</td>
<td>10, 3</td>
<td></td>
</tr>
<tr>
<td>mult tmp2, a</td>
<td>10, 12</td>
<td></td>
</tr>
<tr>
<td>sub tmp2, tmp1</td>
<td>10, 2</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: \( a * b - (a + c * b) \)

<table>
<thead>
<tr>
<th>Operation</th>
<th>tmp1</th>
<th>tmp2</th>
</tr>
</thead>
<tbody>
<tr>
<td>mult</td>
<td>6</td>
<td>?</td>
</tr>
<tr>
<td>add</td>
<td>10</td>
<td>?</td>
</tr>
<tr>
<td>mult</td>
<td>10</td>
<td>12</td>
</tr>
<tr>
<td>sub</td>
<td>10</td>
<td>2</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?

<table>
<thead>
<tr>
<th>Byte Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00110110</td>
</tr>
<tr>
<td>1</td>
<td>00001100</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{align*}
2^{n-4} & \leq r0 < 2^{n-1} \\
2^{n-1} & \leq r31 < 2^n
\end{align*}
\]
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)

<table>
<thead>
<tr>
<th></th>
<th>r1</th>
<th>r2</th>
</tr>
</thead>
<tbody>
<tr>
<td>mult</td>
<td>a,b,c</td>
<td>6,</td>
</tr>
<tr>
<td>add</td>
<td>a</td>
<td>10,</td>
</tr>
<tr>
<td>mult</td>
<td>a,b</td>
<td>10,</td>
</tr>
<tr>
<td>sub</td>
<td>r2, r1</td>
<td>10,</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:** $a \times b - (a + c \times b)$

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

**7 instructions**
Using Registers to Access Memory

• Registers can hold memory addresses

Given

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

Instructions

```c
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
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

• Everything is a number: instructions too!

• 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</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></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>Rs2/Opx</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>
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 2, Appendix B