**Instruction Set Architecture (ISA)**

**CPS 104**

**Lecture 4**

---

**Administrivia**
- Homework #1 Due Wednesday
- My Thursday office hours moved to 11am to noon.
- Yang Liu (yliu@cs.duke.edu)
  - First point of contact for homework
  - Office hours are: Tuesday & Wed 3pm to 4pm 005 North
- UTAs
  - Aaron Carlson (alc28@duke.edu) (Tuesday 4:15 to 5:15pm)
  - Kenneth Leiter (kwli2@duke.edu)
  - Anthony Bishopric (ahb5@duke.edu)

---

**Today’s Lecture**
- Operations provided by the machine
- Outline
- Review
- From high level to instructions
- Types of Instruction Sets
- Reading
  - Chapter 2

---

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

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

On 32-bit machine, `p` is 32-bits
Review: C pointers

typedef struct node {
    int me;
    struct node *next;
} node;

main()
{
    node *list;
    list = (node *) malloc(10*sizeof(node));
    list->me = 1;
    list->next = NULL;
}

C Input/Output

• Printf for output, Scanf for input.
• printf("<format string>", variables..);
  
  int x;
  printf("The value of %x is \%x\n", x);
  %x output, unsigned integer in hex
  %d output, integer in decimal
  %s output, string (variable is pointer to string)
  %f output, float

• Scanf is similar except it reads values into variables,
  so it needs "address of" where to put value
  int x;
  scanf("%x\n", &x);

Review: Bitwise Operators

• & is AND, | is OR
• >> is shift right, << is shift left,
  xpos = 0x134c & 0x000ff
  ypos = (0x134c & 0x0ff00) >> 8
  button = (0x134c & 0x30000) >> 16

  button y x
  0x134c = 01 1010 0011 0100 1100
  0x000ff = 00 1111 1111 0000 0000
  0x0a300 = 00 1010 0011 0000 0000

Instruction Set Architecture

Levels of Representation

High Level Language Program
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;

Compiler
  hw $15, 0($2)
  sw $16, 0($2)

Assembly Language Program
  hw $16, 4($2)
  sw $15, 4($2)

Machine Language Program

Control Signal Specification
  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

#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;
}

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: 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?
- Parallelism

ISA Metrics
- Aesthetics:
  - Regularity (Orthogonality)
  - Completeness
- 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 add Ra Rb Rc Ra ← mem[Rb] + Ra
load Ra Rb Ra ← mem[Rb]
store Ra Rb mem[Rb] ← Ra

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

Accumulator

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

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

Memory

<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>
<tr>
<td>9 instructions</td>
<td></td>
</tr>
</tbody>
</table>

Accumulator
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 c
push b
mult
add
sub
9 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?

2-address ISA

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

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
```

3-address General Purpose Register ISA

- Instruction set: Three explicit operands, ZERO implicit
  add, sub, mult, div...
  add, sub, mult, div...

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

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

3-address ISA

- Instruction set: Three explicit operands, ZERO implicit
  add, sub, mult, div...
  add, sub, mult, div...

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

```
tmp1, tmp2
mult tmp1, b, c 6, ?
add tmp1, tmp1, a 10, 3
mult tmp2, a 10, 12
sub tmp2, tmp1 10, 2
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: \( a*b - (a+c*b) \) (assume in memory)

```
rd r1, c 2, 7, ?
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
```
**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]
  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 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+1]
- Autodecrement M[Ri - 1]
- 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, MC68100, AMD29000, i960, i860
  PARisc, POWERPC, DEC Alpha, Clipper,
  CDC 6600, CDC 7600, Cray-1, Cray-2, Cray-3
  ```

**Example: MIPS**

- **Register-Register**
  ```
  Op 31 26 25 21 20 16 15 11 10 6 5 0
  Rs1 Rs2 Rd Opx
  ```

- **Register-Immediate**
  ```
  Op 31 26 25 21 20 16 15 0
  Rs1 Rd immediate
  ```

- **Branch**
  ```
  Op 31 26 25 21 20 16 15 0
  Rs1 Rs2/Opx immediate
  ```

- **Jump / Call**
  ```
  Op 31 26 25 0
  target
  ```

**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 A