The Hardware/Software Interface

CSE351 Winter 2011

Module 5: Instruction Set Architectures

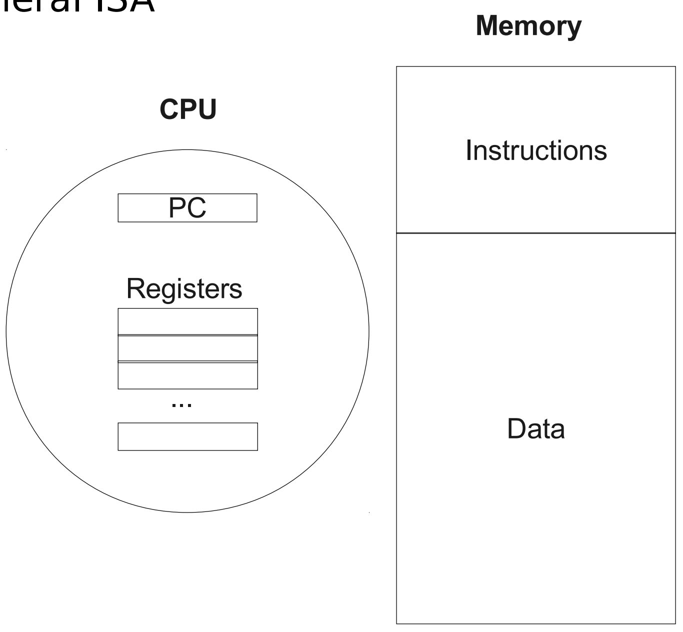
Today Topics: Instruction Set Architectures

- **ISA Goals**
- **ISA Design Decisions**
- **x86 ISA overview**

Preliminaries 1

- **We're going to talk very generally**
  - What does an ISA look like?
  - What design decisions must be made?
  - What are the factors affecting those decisions?
- **We'll talk about specifics of the Intel x86 architecture in more detail later**

The General ISA



General ISA Design Decisions

- **Instructions**
  - What instructions are available? What do they do?
  - How are then encoded?
- **Registers**
  - How many registers are there?
  - How wide are they?
- **Memory**
  - How do you specify a memory location?

Preliminaries 2

- **The goal of the CPU is to execute programs quickly**
- **The time required to execute a program depends on:**
  - The program (as written in C, for instance)
  - The compiler: what set of assembler instructions it translates the C program into
  - The ISA: what set of instructions it made available to the compiler
  - The hardware implementation: how much time it takes to execute an instruction
- **There is a complicated interaction among these**

CISC vs. RISC

- **CISC: Complicated Instruction Set Processor**
  RISC: Reduced Instruction Set Processor

- **CISC's have complicated instructions**
  - Each one does a lot, so is slow to execute, but...
  - A smart compiler can generate machine code that requires only a relatively small number of instruction executions

- **RISC's have small, regular instruction sets**
  - Each instruction does only something very simple, so is fast, but...
  - A relatively large number of instruction executions are required to complete the program

A trivial CISC vs. RISC example

- **x = x + A[y+2]**

- **CISC**
  - addl 8(r1, r2, 4), r3
  - r1: pointer to A
    r2: y
    r3: x
  - meaning: add the 32-bits at address (r1) + 4*(r2) + 8 to r3

- **RISC**

  -
    ```
    sll    r2, 2, r4    # r4 = y*4
    addi   r4, 8, r4    # r4 += 8
    load   0(r4), r4    # r4 = A[y+2]
    add    r4, r3, r3   # add A[y+2] to x
    ```

- **Which is faster?**

CISC vs. RISC: Which is faster?

- **The answer isn't obvious**
  - It depends on what C code programmers write, what machine code compilers can generate, and how fast hardware that implements the ISA can be
- **Extensive analysis of these factors indicates that the RISC wins**
- **Why?**
  - A side effect of being able to execute some complicated instructions is that simple instructions on the CISC execute more slowly than on the RISC
    - Simple instructions are common; complicated ones are rare
  - Complicated instructions interfere with parallelizing the execution of the instruction stream

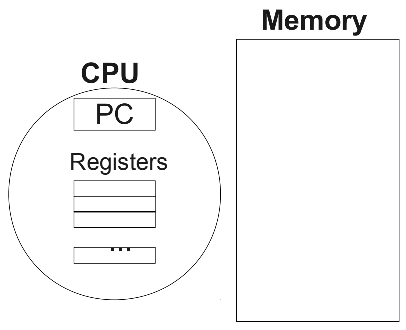
Preliminaries 3

- **This tradeoff wasn't understood when the x86 architecture was designed**
  - The prevailing wisdom was that the more that could be done in hardware, the faster the machine
- **So, the x86 is a CISC**
- **Can't change the ISA because too many installed programs rely on it**
  - If buying a new computer required that you re-purchase Microsoft Word, you might not buy a new computer
- **(What Intel did: the hardware compiles the x86 CISC program into a RISC program "on the fly," and the hardware implements the (hidden) RISC ISA.)**

The Upshot

- **The book talks a lot about the particulars of the x86 ISA**
  - The x86 is hugely important, of course
  - But, the details aren't that important to the typical programmer
- **The book then simplifies to the y86 architecture**
  - The y86 is basically a RISC subset of the x86

The x86 ISA



- **Main abstractions**
  - Instructions
    - Executed by the CPU
  - Data
    - Memory and registers

x86 ISA Part 1
Instructions

Instructions

- **There are three kinds of instructions**
  - Data transformation
    - Examples: and, or, shift, add, subtract, multiply, divide, …
  - Data copy
    - Example: move
  - Conditionals
    - Example: most jumps, some moves

Aside: An unconditional jump is simply an assignment to the PC. So, it's either a data copy or perhaps a data transformation instruction. The key functionality we need is an operation whose outcome depends on a test – a conditional.

Conditional Control Flow

- Two things are required:
  - Evaluate a condition
    - Example: compare [R1] with [R2]
  - Either branch or continue execution sequentially depending on the outcome of the condition
    - Examples: equal, not equal, less than, less than or equal, not less than (greater than or equal), …

Evaluating the Condition

- The result of the condition evaluation must be put somewhere
  - We could use a register, but registers are valuable
    - And on the x86 there are only 8 of them!
      - (And on the x86 there are only 6 of them!)
  - Instead, we use a special register, the condition code
    - The condition code is a bit mask: overflow, sign, carry, parity, and zero bits
- These bits are available when performing arithmetic operations, and it's cheap to save them to the condition code, so the x86 does
  - C code fragment: `x = x + y; if ( x != 0 ) ...`
  - Rather than

```
add r3,r4
cmp r4,$0
bz  skip
```

need only

```
add r3,r4
bz  skip
```

Evaluating the Condition (cont.)

- As well as being set as a side-effect of arithmetic instructions, there is a set of compare instructions whose only action is to set the condition code
  - An arithmetic operation writes some register
  - Registers are valuable!
  - The `cmp` instruction compares and sets the condition code bits, but doesn't alter any registers
- Lessons for later:
  - Registers are valuable!
  - The x86 has only 8 of them!

Encoding Branches

- The y86 (and z86) architectures encode branches using 32-bit absolute addresses
  - There is a 1 byte opcode
  - There is a 4 byte absolute address
- There are two problems with that:
  - The instruction takes a lot of bytes
  - You need to know the absolute address of the branch at assembly time
    - Why might it be impossible to know the absolute address at assembly time?

Encoding Branches (cont.)

- Both problems can be (mostly) solved using PC relative addressing
  - Instead of giving a 32-bit absolute address, give an 8- or 16-bit offset from the current PC
    - 8 bits: can branch between -127 and +128 bytes from current PC
    - 16 bits: can branch between -32,768 and +32,767 bytes away
  - Most branches are within those ranges
- Note that we don't need to know where the code will be loaded in memory at assembly time
  - The OS will set the PC to the first instruction of the program, before starting it
  - PC relative encoded branches will all work, no matter where the code is loaded

PC Relative Addressing

```
0x100     cmp  r2, r3     0x1000
0x102     je   0x70       0x1002
0x104     …               0x1004
…         …               …
0x172     add  r3, r4     0x1072
```

- PC relative branches are relocatable
- Absolute branches are not

Conditionals and Control Flow

- **A test / conditional branch is sufficient to implement most control flow constructs offered in higher level languages**
  - if (condition) then {...} else {…}
  - while(condition) {…}
  - do {…} while (condition)
  - for (initialization; condition;) {...}
- **(Unconditional branches implemented some related control flow constructs**
  - break, continue)

Compiling Loops

C/Java code

```
while ( sum != 0 ) {
   <loop body>
}
```

Machine code

```
loopTop:    cmp   r3, $0
            be    loopDone
                 <loop body code>
            jmp  loopTop
loopDone:
```

- How to compile other loops should be clear to you
  - The only slightly tricky part is to be sure where the conditional branch occurs: top or bottom of the loop
- Q: How is `for(i=0; i<100; i++)` implemented?
- Q: How are `break` and `continue` implemented?

The switch statement

```
switch (class) {
  case 0: <some code>
          break;
  case 1: <some code>
          break;
  case 4: <some code>
          break;
…
  default: <some code>
           break;
}
```

- At first glance, switch doesn't conform to our notion of “either take the branch or not”
  - It's not a binary decision, it's n-ary
- switch can be re-written as an if-then-else
  - transforms it into n binary decisions
- there is sometimes an optimized implementation available to the compiler
  - jump tables

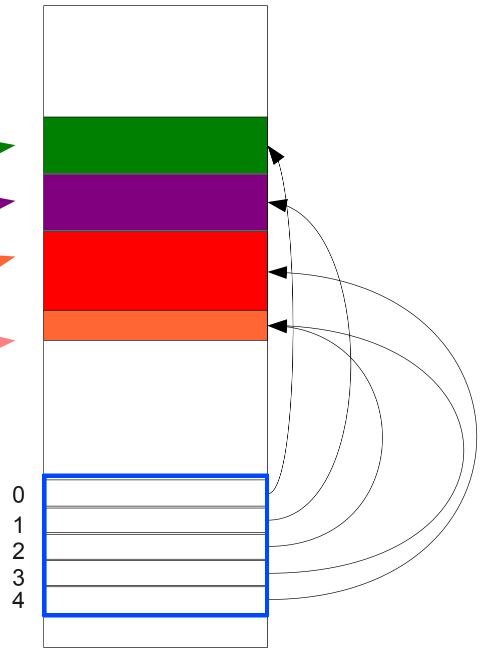
switch / jump tables

```
switch (class) {
  case 0: <some code>
          break;
  case 1: <some code>
          break;
  case 4: <some code>
          break;
…
  default: <some code>
           break;
}
```

- A jump table is an array of addresses
  - Each address points to an instruction
- In the case of switch, the jump table entries point to the sections of code for the case's
- To implement the switch:
  - use the selector (class, in the example) as an index into the jump table
  - load the 32-bit address from the jump table into a register, say r3
  - jmp r3
    - This is an unconditional jump
    - The PC is assigned the contents of r3

jump table picture

```
switch (class) {
  case 0: <some code>
          break;
  case 1: <some code>
          break;
  case 4: <some code>
          break;
  default: <some code>
           break;
}
```



switch / jump table review

```
switch (switch) {
  case 0:      <some code>
               break;
  case 1:      <some code>
               break;
  case 52000: <some code>
               break;
  default:     <some code>
               break;
}
```

Why is the compiler unlikely to implement this as a jump table?

switch aside

```
int x, y, z, class;
…
switch (class) {
  case x:     <some code>
              break;
  case y:     <some code>
              break;
  case z: <some code>
              break;
  default:    <some code>
              break;
}
```

Why is this not legal (in most languages)?

x86 ISA Part 2
Data

x86 Data

- Programmer controlled data is held in registers and memory
- The ISA says registers are 32-bits wide
- The ISA says that memory:
  - is byte addressable
  - allows transfers of 1, 2, 4 bytes into / out of registers
- The “names” for data are defined by the ISA, not the programmer
  - registers 0 – 7
  - memory locations 0, 1, 2, …

Types

- There is some notion of type defined by the ISA
  - byte vs. word
  - integer vs. float
- However, the type is <u>not</u> associated with the data
  - It's associated with the operation being performed on the data
  - `add  8(r2), r3` vs. `fadd 8(r2), f3`
- There is no notion of type checking
  - What might type checking mean at the hardware level?
  - Why would you not implement that?

Addressing memory

- Instructions that use memory have to specify an address
- Embedding addresses into instructions has two drawbacks
  - Instructions are big: an address is 32 bits
  - You have to know the address at assembly time
    - You don't if the code wants to 'new' up an object, say
    - You don't if you're not sure just where the code will be loaded into memory
    - You don't if only part of the code is compiled at a time
      - In C, it's routine to compile just one file of a program that is composed of dozens of files
      - The compiler sees only the code in that file, not the whole program, so cannot decide where in memory code or data will be located
      - Resolving this particular issue is the job of the linker. We'll come back to it later.

Addressing Memory (continued)

- The most general memory addressing scheme is to indirect using a register
  - The instruction names a register that holds the address
    - mrmovl (r2), r3
  - An arbitrary (and arbitrarily long) sequence of instructions can be used to compute the address
- That works, but you end up needing either
  - a lot of registers, each pointing at a variable currently used frequently, or
  - a single register but a lot of instructions (re)computing addresses you need frequently

Addressing Memory (cont.)

- Base-displacement addressing provides more flexibility
  - mrmovl $8(r2), r3
  - effective address is 8 + R[r2]
- Example use of base-displacement addressing: arrays
  - Array access with an index known at compile time
    - A[2]
    - r2 points at array A; the offset of element 2 is 8

Array Addressing

- It isn't that common to know the array index at compile time
  - A[j] is more common than A[2]
- The x86 supports this too, meaning it provides a way to create the required effective address without using extra registers or extra instructions
  - (r2, r5, 4)
    - r2 points to A
    - r5 holds j (you needed that in a register anyway)
    - 4 is the size of each element of the array, in bytes
  - effective address is R[r2] + R[r5]*4
- This is very CISC...
- x86 even supports A[j+2]
  - 8(r2,r5,4)

Back to base-displacement addressing...

- Simple (RISC-y) base-displacement addressing is useful beyond arrays
- Distinct variables that the compiler knows it has allocated contiguously can be addressed using a single register
  - int i, j, k;
    - r2 points to i; $4(r2) is j; $8(r2) is k
  - Note that the compiler knows the offsets of the variables at compile time
    - Note that it isn't essential that it know the value of the “base” (r2) – it can generate instructions to set that up at run time
  - Note that only a single base register allows access to a large number of distinct variables

C structs

- C provides a user-defined, structured data type: the struct
- struct {
    int accountNumber;
    int balance; // in pennies
  } account;
  - That actually creates a variable, named account, of the struct type, but there's a way to define the type and then declare many instances of it as well
- Note that if I have a pointer to account in r2, say:
  - 0(r2) points to account.accountNumber
  - 4(r2) points to account.balance
- This idea is how the C++ compiler generates instructions to access the instance variables of objects (C++ has classes...)

C: User defined types

- C's facility for defining types is really just an aliasing facility
  - typedef unsigned char byte;
    - You can now type 'byte' anywhere 'unsigned char' would have made sense

  -
    ```
    typedef  struct {
       int accountNumber;
       int balance;  // in pennies
    } AccountType;
    AccountType accounts[200];
    ```
    - Define a type (AccountType),, then create an array of elements of that type
- Type equivalence in C is by name
  - So, an element of array accounts is not the same type as variable 'account' on the previous slide (even though the struct definitions are identical)
    - accounts[0] is of type 'AccountType'
    - account is of type 'struct <anonymous>'
- When types are compatible, struct assignment is defined (as bit copy)
  - accounts[0] = accounts[10]; // is legal
  - This is "shallow copy", in 142/3 terminology

x86 ISA Part III
ISA Abstractions v. HLL Abstractions

Overview

- You know a lot about Java
    - You also know a bit about C, and how much of it is a lot like Java
- You know the essentials about what hardware does
- Let's look at the programming abstractions Java/C provide and compare with what the hardware provides
    - Anything not provided by the hardware must be being provided by software, e.g., the compiler

HLL vs. HW

- Let's divide the discussion into three parts:
    - things in C I don't understand (e.g., because they're not legal in Java)
    - statements, operators, and control flow
    - data / variables

```
#include <stdio.h>

int N = 16;

int fib(int n) {
  int result;
  if ( n == 0 ) result = 0;
  else if ( n == 1 ) result = 1;
  else result = fib(n-1) + fib(n-2);
  return result;
}

int main(int argc, char* argv[]) {
  int result = fib(N);
  printf("Fibonnaci[%d] = %d\n", N, result);
  return 0;
}
```