Today’s class

- Instruction set architecture
Instruction Set Design

- The instruction set architecture (ISA) level is between the high-level languages and the hardware
- When new hardware architecture comes along …
  - Need to maintain backward compatibility
  - Can add new features to exploit new hardware capabilities
ISA Level

FORTRAN 90 program

FORTRAN 90 program compiled to ISA program

C program

C program compiled to ISA program

ISA level

ISA program executed by microprogram or hardware

Hardware

Software

 Hardware
Properties of the ISA Level

- In principle, the ISA level is defined by how the machine appears to a machine language programmer.
- In modern times, this is refined to say that ISA-level code is what a compiler outputs.
- Compiler writer needs to know:
  - Memory model
  - What registers are available
  - What data types are available
  - What instructions are available
Memory Models

- Consider 8-byte (64-bit) words
- Do words have to be “word-aligned” (start at a word address)?

(a) Aligned 8-byte word at address 8
(b) Nonaligned 8-byte word at address 12
Instruction and Data Spaces

- Most machines have a single linear address space at the ISA level
- However, a few machines have separate address spaces for instructions and data
  - Effectively doubles memory!
  - All writes go to data space, protecting instructions from being overwritten
Registers

- **Special purpose**
  - Program counter
  - Stack pointer
  - Program status word

- **General purpose**
  - Hold key local variables and intermediate results of calculations
  - RISC machines usually have at least 32
  - Trend in new CPU designs is to have even more
Instructions

- LOAD and STORE instructions – move data between memory and registers
- MOVE instructions – move data among the registers
- Arithmetic instructions
- Boolean instructions
- Comparing data items and branching on the results
Pentium 4

- Three operating modes
  - Real mode – all features added since the 8088 are turned off (effectively behaves like a simple 8088; if a program does something wrong whole machine crashes)
  - Virtual 8086 mode – makes it possible to run old 8088 programs in a protected way (if a program crashes, the OS is notified instead of the machine crashing)
  - Protected mode – actually behaves like a Pentium 4!
Pentium 4

- Address space
  - 16,384 segments
  - Each segment has $2^{32}$ addresses (4 GB)
  - Most operating systems, including Unix and Windows, support only one segment
  - Every byte has its own address
  - Words are 32 bits long, and low-endian
Pentium 4

- **Registers**
  - EAX is main arithmetic register
  - EBX holds pointers
  - ECX plays a role in looping
  - EDX used in multiplication and division
  - ESI – source string pointer
  - EDI – destination string pointer
  - EBP – base of the current stack frame
  - ESP – stack pointer
  - EIP – instruction pointer
Pentium 4

- Data types
  - Two’s complement integers
  - Unsigned integers
  - Binary coded decimal numbers
  - IEEE 754 floating point numbers

<table>
<thead>
<tr>
<th>Type</th>
<th>1 Bit</th>
<th>8 Bits</th>
<th>16 Bits</th>
<th>32 Bits</th>
<th>64 Bits</th>
<th>128 Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Signed integer</td>
<td>×</td>
<td></td>
<td></td>
<td>×</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unsigned integer</td>
<td>×</td>
<td></td>
<td>×</td>
<td>×</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary coded decimal integer</td>
<td>×</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Floating point</td>
<td></td>
<td></td>
<td></td>
<td>×</td>
<td>×</td>
<td></td>
</tr>
</tbody>
</table>

Friday, November 16, 2007

Computer Architecture I - Class 11
Instruction Formats

(a) 

(b) 

(c) 

(d)
Design Criteria

- Short instructions are better than long ones
- Sufficient room in the instruction format to express all the operations desired
- Number of bits in the address field
Examples

- Consider an \((n + k)\) bit instruction that has a \(k\)-bit opcode and a single \(n\)-bit address
  - Allows for \(2^k\) different operations
  - Allows for \(2^n\) addressable memory cells

- Could alternatively have:
  - \(k-1\) bit opcode and \(n+1\) bit address
    - Half as many instructions but twice the addressable memory
  - \(k+1\) bit opcode and \(n-1\) bit address
    - Twice as many instructions but half the addressable memory
Expanding Opcodes

- An instruction with a 4-bit opcode and three 4-bit address fields.
Expanding Opcodes

- Suppose the designers need:
  - 15 three-address instructions
  - 14 two-address instructions
  - 31 one-address instructions
  - 15 no-address instructions

- Use opcodes 0-14 as the three-address instructions

- Interpret opcode 15 differently
Expanding Opcodes

- 16 bits
  - 4-bit opcode
  - 15 3-address instructions
  - 12-bit opcode
  - 14 2-address instructions
  - 16-bit opcode
  - 16 0-address instructions

- 8-bit opcode
  - 1111 0000 yyy yyyy
  - 1111 0001 yyy yyy
  - 1111 0010 yyy yyy
  - 1111 1011 yyy yyy
  - 1111 1100 yyy yyy
  - 1111 1101 yyy yyy

Bit number:
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Pentium 4 Instruction Formats

- Highly complex and irregular!
- Up to 6 variable-length fields, 5 of which are optional

```
<table>
<thead>
<tr>
<th>Bytes</th>
<th>0 - 5</th>
<th>1 - 2</th>
<th>0 - 1</th>
<th>0 - 1</th>
<th>0 - 4</th>
<th>0 - 4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PREFIX</td>
<td>OPCODE</td>
<td>MODE</td>
<td>SIB</td>
<td>DISPLACEMENT</td>
<td>IMMEDIATE</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Bits</th>
<th>6</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>INSTRUCTION</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Which operand is source?
- Byte/word

```
<table>
<thead>
<tr>
<th>Bits</th>
<th>2</th>
<th>3</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SCALE</td>
<td>INDEX</td>
<td>BASE</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Bits</th>
<th>MOD</th>
<th>REG</th>
<th>R/M</th>
</tr>
</thead>
</table>
```
Addressing Modes

- How to specify where an operand for an instruction is located
- How the bits of an address field in an instruction are interpreted
Immediate Addressing

- Address part of the instruction contains the actual operand itself
- An immediate instruction for loading 4 into register 1
  
  | MOV | R1 | 4 |
  |
- Does not require an extra memory reference to fetch the operand
Direct Addressing

- Instruction specifies full address of the operand
- Instruction will always access the same memory location
- Useful for accessing global variables
Register Addressing

- Specifies a register for the operand
- Most common addressing mode on computers
Register Indirect Addressing

- Register contains the address in memory of the operand (the register acts as a pointer)
- The following example computes the sum of the elements of an array

```
MOV R1,#0 ; accumulate the sum in R1, initially 0
MOV R2,#A ; R2 = address of the array A
MOV R3,#A+4096 ; R3 = address of the first word beyond A
LOOP: ADD R1,(R2) ; register indirect through R2 to get operand
ADD R2,#4 ; increment R2 by one word (4 bytes)
CMP R2,R3 ; are we done yet?
BLT LOOP ; if R2 < R3, we are not done, so continue
```
Indexed Addressing

- Contents of a register plus a constant offset form the address of the operand
- Consider the following program for computing the OR of corresponding products in two arrays

```assembly
MOV R1,#0 ; accumulate the OR in R1, initially 0
MOV R2,#0 ; R2 = index, i, of current product: A[i] AND B[i]
MOV R3,#4096 ; R3 = first index value not to use
LOOP:
    MOV R4,A(R2)
    AND R4,B(R2)
    OR R1,R4 ; OR all the Boolean products into R1
    ADD R2,#4 ; i = i + 4 (step in units of 1 word = 4 bytes)
    CMP R2,R3 ; are we done yet?
    BLT LOOP ; if R2 < R3, we are not done, so continue
```

Friday, November 16, 2007
Reverse Polish Notation

- Infix form: \( a + b \)
- Postfix form: \( a \ b \ + \)
- Any expression can be expressed without parentheses:
  \( (a + b) \times c = a \ b \ + \ c \times \)
- Convenient for evaluating formulas using stacks on a computer
Reverse Polish Notation

- Some examples of infix expressions and their reverse Polish notation equivalents.

<table>
<thead>
<tr>
<th>Infix</th>
<th>Reverse Polish notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>A + B × C</td>
<td>A B C × +</td>
</tr>
<tr>
<td>A × B + C</td>
<td>A B × C +</td>
</tr>
<tr>
<td>A × B + C × D</td>
<td>A B × C D × +</td>
</tr>
<tr>
<td>(A + B) / (C – D)</td>
<td>A B + C D – /</td>
</tr>
<tr>
<td>A × B / C</td>
<td>A B × C /</td>
</tr>
<tr>
<td>((A + B) × C + D) / (E + F + G)</td>
<td>A B + C × D + E F + G + /</td>
</tr>
</tbody>
</table>
Evaluation of RPN Formulas

- Evaluation of $\frac{8 + 2 \times 5}{1 + 3 \times 2 - 4}$
- RPN form is $8 \ 2 \ 5 \ x \ + \ 1 \ 3 \ 2 \ x \ + \ 4 \ - \ /$

<table>
<thead>
<tr>
<th>Step</th>
<th>Remaining string</th>
<th>Instruction</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$8 \ 2 \ 5 \ x \ + \ 1 \ 3 \ 2 \ x \ + \ 4 \ - \ /$</td>
<td>BIPUSH 8</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>$2 \ 5 \ x \ + \ 1 \ 3 \ 2 \ x \ + \ 4 \ - \ /$</td>
<td>BIPUSH 2</td>
<td>8, 2</td>
</tr>
<tr>
<td>3</td>
<td>$5 \ x \ + \ 1 \ 3 \ 2 \ x \ + \ 4 \ - \ /$</td>
<td>BIPUSH 5</td>
<td>8, 2, 5</td>
</tr>
<tr>
<td>4</td>
<td>$x \ + \ 1 \ 3 \ 2 \ x \ + \ 4 \ - \ /$</td>
<td>IMUL</td>
<td>8, 10</td>
</tr>
<tr>
<td>5</td>
<td>$+ \ 1 \ 3 \ 2 \ x \ + \ 4 \ - \ /$</td>
<td>IADD</td>
<td>18</td>
</tr>
<tr>
<td>6</td>
<td>$1 \ 3 \ 2 \ x \ + \ 4 \ - \ /$</td>
<td>BIPUSH 1</td>
<td>18, 1</td>
</tr>
<tr>
<td>7</td>
<td>$3 \ 2 \ x \ + \ 4 \ - \ /$</td>
<td>BIPUSH 3</td>
<td>18, 1, 3</td>
</tr>
<tr>
<td>8</td>
<td>$2 \ x \ + \ 4 \ - \ /$</td>
<td>BIPUSH 2</td>
<td>18, 1, 3, 2</td>
</tr>
<tr>
<td>9</td>
<td>$x \ + \ 4 \ - \ /$</td>
<td>IMUL</td>
<td>18, 1, 6</td>
</tr>
<tr>
<td>10</td>
<td>$+ \ 4 \ - \ /$</td>
<td>IADD</td>
<td>18, 7</td>
</tr>
<tr>
<td>11</td>
<td>$4 \ - \ /$</td>
<td>BIPUSH 4</td>
<td>18, 7, 4</td>
</tr>
<tr>
<td>12</td>
<td>$- \ /$</td>
<td>ISUB</td>
<td>18, 3</td>
</tr>
<tr>
<td>13</td>
<td>$/$</td>
<td>IDIV</td>
<td>6</td>
</tr>
</tbody>
</table>
Pentium 4 Addressing Modes

- Highly irregular
- Depends on which operating mode you are in (16-bit or 32-bit)
- Not all modes apply to all instructions
- Not all registers can be used in all modes
- Compiler writer’s nightmare!
# Pentium 4 Addressing Modes

<table>
<thead>
<tr>
<th>R/M</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>MOD</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>M[EAX]</td>
<td>M[EAX + OFFSET8]</td>
<td>M[EAX + OFFSET32]</td>
<td>EAX or AL</td>
</tr>
<tr>
<td>100</td>
<td>SIB</td>
<td>SIB with OFFSET8</td>
<td>SIB with OFFSET32</td>
<td>ESP or AH</td>
</tr>
<tr>
<td>101</td>
<td>Direct</td>
<td>M[EBP + OFFSET8]</td>
<td>M[EBP + OFFSET32]</td>
<td>EBP or CH</td>
</tr>
</tbody>
</table>
Data Movement Instructions

- Copying data is the most fundamental of all operations
- Four possibilities:
  - Register to register
  - Register to memory
  - Memory to register
  - Memory to memory
- May have one general instruction to cover all cases, or separate instructions for each case
- Need to indicate how much data to copy
Dyadic Operations

- Combine two operands to produce a result
- Arithmetic instructions
- Boolean instructions
Monadic Operations

- Have one operand and produce one result
- Shift and rotate instructions
- CLR, INC, NEG
Comparisons and Conditional Branches

- Common operation is a comparison to 0
- Use bits in the PSW, such as C, N, Z
Other Instructions

- Procedure calls
- Loop control
Programmed Input/Output

- Single input instruction and single output instruction
- Selects an I/O device and a single character is transferred
- CPU must execute an explicit sequence of instructions for each and every character read or written
Programmed Input/Output

- A simple terminal might have four 1-byte registers
- Registers might be part of computer’s memory address space
Direct Memory Access

- DMA gets rid of most of the interrupts of programmed I/O
# Pentium 4 Instructions

<table>
<thead>
<tr>
<th><strong>Moves</strong></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV DST, SRC</td>
<td>Move SRC to DST</td>
</tr>
<tr>
<td>PUSH SRC</td>
<td>Push SRC onto the stack</td>
</tr>
<tr>
<td>POP DST</td>
<td>Pop a word from the stack to DST</td>
</tr>
<tr>
<td>XCHG DS1, DS2</td>
<td>Exchange DS1 and DS2</td>
</tr>
<tr>
<td>LEA DST, SRC</td>
<td>Load effective addr of SRC into DST</td>
</tr>
<tr>
<td>CMOVcc DST, SRC</td>
<td>Conditional move</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Arithmetic</strong></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD DST, SRC</td>
<td>Add SRC to DST</td>
</tr>
<tr>
<td>SUB DST, SRC</td>
<td>Subtract SRC from DST</td>
</tr>
<tr>
<td>MUL SRC</td>
<td>Multiply EAX by SRC (unsigned)</td>
</tr>
<tr>
<td>IMUL SRC</td>
<td>Multiply EAX by SRC (signed)</td>
</tr>
<tr>
<td>DIV SRC</td>
<td>Divide EDX:EAX by SRC (unsigned)</td>
</tr>
<tr>
<td>IDIV SRC</td>
<td>Divide EDX:EAX by SRC (signed)</td>
</tr>
<tr>
<td>ADC DST, SRC</td>
<td>Add SRC to DST, then add carry bit</td>
</tr>
<tr>
<td>SBB DST, SRC</td>
<td>Subtract SRC &amp; carry from DST</td>
</tr>
<tr>
<td>INC DST</td>
<td>Add 1 to DST</td>
</tr>
<tr>
<td>DEC DST</td>
<td>Subtract 1 from DST</td>
</tr>
<tr>
<td>NEG DST</td>
<td>Negate DST (subtract it from 0)</td>
</tr>
</tbody>
</table>
## Pentium 4 Instructions

### Binary coded decimal

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAA</td>
<td>Decimal adjust</td>
</tr>
<tr>
<td>DAS</td>
<td>Decimal adjust for subtraction</td>
</tr>
<tr>
<td>AAA</td>
<td>ASCII adjust for addition</td>
</tr>
<tr>
<td>AAS</td>
<td>ASCII adjust for subtraction</td>
</tr>
<tr>
<td>AAM</td>
<td>ASCII adjust for multiplication</td>
</tr>
<tr>
<td>AAD</td>
<td>ASCII adjust for division</td>
</tr>
</tbody>
</table>

### Boolean

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND DST,SRC</td>
<td>Boolean AND SRC into DST</td>
</tr>
<tr>
<td>OR DST,SRC</td>
<td>Boolean OR SRC into DST</td>
</tr>
<tr>
<td>XOR DST,SRC</td>
<td>Boolean Exclusive OR SRC to DST</td>
</tr>
<tr>
<td>NOT DST</td>
<td>Replace DST with 1’s complement</td>
</tr>
</tbody>
</table>

### Shift/rotate

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAL/SAR DST,#</td>
<td>Shift DST left/right # bits</td>
</tr>
<tr>
<td>SHL/SHR DST,#</td>
<td>Logical shift DST left/right # bits</td>
</tr>
<tr>
<td>ROL/ROR DST,#</td>
<td>Rotate DST left/right # bits</td>
</tr>
<tr>
<td>RCL/RCR DST,#</td>
<td>Rotate DST through carry # bits</td>
</tr>
</tbody>
</table>
### Pentium 4 Instructions

#### Test/compare
- **TEST SRC1, SRC2**: Boolean AND operands, set flags
- **CMP SRC1, SRC2**: Set flags based on SRC1 - SRC2

#### Transfer of control
- **JMP ADDR**: Jump to ADDR
- **Jxx ADDR**: Conditional jumps based on flags
- **CALL ADDR**: Call procedure at ADDR
- **RET**: Return from procedure
- **IRET**: Return from interrupt
- **LOOPxx**: Loop until condition met
- **INT n**: Initiate a software interrupt
- **INTO**: Interrupt if overflow bit is set

#### Strings
- **LODS**: Load string
- **STOS**: Store string
- **MOVS**: Move string
- **CMPS**: Compare two strings
- **SCAS**: Scan Strings
## Pentium 4 Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STC</td>
<td>Set carry bit in EFLAGS register</td>
</tr>
<tr>
<td>CLC</td>
<td>Clear carry bit in EFLAGS register</td>
</tr>
<tr>
<td>CMC</td>
<td>Complement carry bit in EFLAGS</td>
</tr>
<tr>
<td>STD</td>
<td>Set direction bit in EFLAGS register</td>
</tr>
<tr>
<td>CLD</td>
<td>Clear direction bit in EFLAGS reg</td>
</tr>
<tr>
<td>STI</td>
<td>Set interrupt bit in EFLAGS register</td>
</tr>
<tr>
<td>CLI</td>
<td>Clear interrupt bit in EFLAGS reg</td>
</tr>
<tr>
<td>PUSHFD</td>
<td>Push EFLAGS register onto stack</td>
</tr>
<tr>
<td>POPFD</td>
<td>Pop EFLAGS register from stack</td>
</tr>
<tr>
<td>LAHF</td>
<td>Load AH from EFLAGS register</td>
</tr>
<tr>
<td>SAHF</td>
<td>Store AH in EFLAGS register</td>
</tr>
</tbody>
</table>
# Pentium 4 Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SWAP DST</td>
<td>Change endianness of DST</td>
</tr>
<tr>
<td>CWQ</td>
<td>Extend EAX to EDX:EAX for division</td>
</tr>
<tr>
<td>CWDE</td>
<td>Extend 16-bit number in AX to EAX</td>
</tr>
<tr>
<td>ENTER SIZE,LV</td>
<td>Create stack frame with SIZE bytes</td>
</tr>
<tr>
<td>LEAVE</td>
<td>Undo stack frame built by ENTER</td>
</tr>
<tr>
<td>NOP</td>
<td>No operation</td>
</tr>
<tr>
<td>HLT</td>
<td>Halt</td>
</tr>
<tr>
<td>IN AL,PORT</td>
<td>Input a byte from PORT to AL</td>
</tr>
<tr>
<td>OUT PORT,AL</td>
<td>Output a byte from AL to PORT</td>
</tr>
<tr>
<td>WAIT</td>
<td>Wait for an interrupt</td>
</tr>
</tbody>
</table>

- **SRC** = source
- **DST** = destination
- **#** = shift/rotate count
- **LV** = # locals
Flow of Control

- Most statements execute sequentially
- Branches alter statement flow
- Procedures also alter statement flow, but unlike branches, control is returned to the statement after the procedure call once the procedure is finished
- Recursive procedures are procedures that call themselves
The Towers of Hanoi

- An ancient problem with a simple recursive solution
- In a certain monastery in Hanoi there are 3 gold pegs
- Around the first peg are 64 gold disks, each disk slightly smaller in diameter than the disk below it
- The monks have to move the disks to the third peg, one disk at a time, but at no time may a larger disk sit on top of a smaller one
- It is said that when they finish the world will come to an end
5-Disk Initial Configuration
Solution Outline

- First move \( n-1 \) disks from peg 1 to peg 2
- Now move 1 disk from peg 1 to peg 3
- Then move the \( n-1 \) disks from peg 2 to peg 3
Solution

This procedure moves \( n \) disks from peg \( I \)

```java
public void towers(int n, int i, int j) {
    int k;

    if (n == 1)
        System.out.println("Move a disk from " + i + " to " + j);
    else {
        k = 6 - i - j;
        towers(n - 1, i, k);
        towers(1, i, j);
        towers(n - 1, k, j);
    }
}
```
The Stack

- Need a stack to store the parameters and local variables for each invocation of a recursive procedure.
- Do not want to confuse values of these variables in different invocations.
The Stack
Procedure Calls

A called from main program

A returns to main program

(a) Calling procedure

(b) Called procedure
Coroutines

A called from main program

RESUME B
RESUME A
RESUME B
RESUME A
RESUME B
RESUME A

A returns to main program

(a)

(b)
Interrupts

- Disk interrupt priority 4 held pending
- RS232 ISR finishes disk interrupt occurs
- Printer interrupt priority 2
- RS232 interrupt priority 5

Time:
- User program
- Printer ISR
- RS232 ISR
- Disk ISR
- Printer ISR
- User program

Stack:
- User
- Printer
- User
- Printer