Computer Organization & Assembly Language Programming

CSE 2312
Lecture 12 Instruction Types

Junzhou Huang, Ph.D.
Department of Computer Science and Engineering
Reviewing (1): Addressing

• **Addressing**
  – How to specify where the operands are for an instruction?

• **Addressing Mode**
  – How the bits of an address field are interpreted to find the operands in the instruction?

• **Different Modes**
  – Immediate Addressing
  – Direct Addressing
  – Register Addressing
  – Register Indirect Addressing
  – Indexed Addressing
  – Based-Indexed Addressing
  – Stacked Addressing
Reviewing (2): Immediate Addressing

An immediate instruction for loading 4 into register 1.

- **What**
  - The address part of an instruction actually contain the operand itself instead of an address or others of the operand. Hence, the operand is immediately available for use

- **Advantage**
  - Do not require an extra memory reference to fetch the operand

- **Disadvantage**
  - Only a constant can be supplied with this addressing mode
  - The number of values is limited by the size of the field
Reviewing (2): Direct and Register Addressing

• **Direct addressing**
  – Specify an operand in memory by its full address
  – The instruction will always access exactly the same memory location.
  – While the value can change, the location can not
  – Only be used to access the global variable whose address is known at compile time

• **Register Addressing**
  – Similar to direct addressing, but specify a register instead of a memory location
  – The most common one modern computers due to the important role of the register in the modern computers
Register Indirect Addressing

- Specified operand comes from or go to memory, but its address is contained in a register. The address is called a pointer in this mode.
- Advantage: 1) can reference memory without having full address in the instruction; 2) use different memory words on different executions of the instruction.
Reviewing (4): Indexed Addressing (1)

A generic assembly program for computing the OR of \( A_i \) and \( B_i \) for two 1024-element arrays

- **Indexed Addressing**
  - Addressing memory by giving a register plus a constant offset
  - **Above**: a register R2, a constant offset (the address of A)
Reviewing (5): Indexed Addressing (2)

A possible representation of $\text{MOV R4, A(R2)}$.

- **Indexed Addressing**
  - **Above**: a register R2, a constant offset (the address of A),
  - Suppose the constant offset is 124300;
  - Suppose R2 is 4;
  - $1^{\text{st}}$ loop: the memory word addressed is $A_0=124300$
  - $2^{\text{nd}}$ loop: the memory word addressed is $A_1=124304$
  - Here: the offset is the memory pointer;
  - Here: the value in the register is a small integer that is incremented during the calculation
Reviewing (6): Based-Indexed Addressing

• **Based-Indexed Addressing**
  – Memory address is computed by adding up two registers plus an optional offset
  – One of register is the base and another is the index
  – Above: outside of the loop, put the address of A in R5 and the address of B in R6

```
LOOP:    MOV R4,(R2+R5)
         AND R4,(R2+R6)
```
Reviewing (7): Stack Addressing

- **Different Notation for** $x + y$
  - Infix: operator + between the operands x and y
  - Postfix: operator after operands $xy^+$

- **Reverse Polish Notation**
  - For example: $3 + 5 \times 7$ | $357^+ +$
  - For example: $3 \times 5 + 7$ | $35^7^+ +$

- **Advantages**
  - Any formula can be expressed without parentheses
  - It is convenient for evaluating formulas on computers with stacks.
  - Infix operators have precedence, which is arbitrary and undesirable.
  - For example, we know that $a \times b + c$ means $(a \times b) + C$ and not $a \times (b + c)$ because multiplication has been arbitrarily defined to have precedence over addition. But how computers know?
Instruction Types

- **Different Instructions**
  - **Data Movement Instructions**: copy data from one place to another place, =
  - **Dyadic Instructions**: combine two operands to produce a result, +, -
  - **Monadic Instructions**: have one operand and produce a result
  - **Instructions for Comparisons and Conditional Branches**: test some condition and branch to a particular memory address if the condition is met
  - **Procedure Call Instructions**: call subroutine
  - **Loop Control Instructions**: execute a group of instruction a fixed number of times
  - **Input/Output Instructions**:
    1) programmed I/O;
    2) Interrupt-driven I/O;
    3) DMA I/O
Dyadic Instruction

- **Dyadic Instructions**
  - Combine two operands to produce a result, +, -

- **A AND B**
  - The result of this operation is that the unwanted bits are all changed into zeros—that is, masked out

- **A OR B**
  - While the AND tends to remove 1s, because there are never more 1s in the result than in either of the operands. The OR operation tends to insert 1s,

```
10110111 10111100 11011011 10001011 A
11111111 11111111 11111111 00000000 B (mask)
10110111 10111100 1101101100000000 AANDB
00000000 00000000 00000000 01010111 C
10110111 10111100 11011011 01010111 (A AND B) OR C
```
Monadic Instructions (1)

- **Monadic Instructions**
  - Have one operand and produce a result

- **Shift**
  - Operations in which the bits are moved to the left or right, with bits shifted off the end of the word being lost.
  - An important use is multiplication and division by powers of 2.
  - Example: left shift k bits: the original number multiplied by $2^k$.
  - Example: right shift k bits: the original number divided by $2^k$.
  - Example: $18 \cdot n = 16 \cdot n + 2 \cdot n$; Then, a move, two shift and an addition instead of the multiplication.

```
00000000 00000000 00000000 01110011 A
00000000 00000000 00000000 00011100 A shifted right 2 bits
11000000 00000000 00000000 00011100 A rotated right 2 bits
```
Exercise

- **00001100 = 12**
  - Left shift 1 bit:
    - 00011000 = 24  = 12 * 2
  - Left shift 2 bit:
    - 00110000 = 48  = 12 * 2^2
  - Right shift 1 bit:
    - 00000110 = 6    = 12 / 2
  - Right shift 2 bit:
    - 00000011 = 3    = 12 / 2^2
Exercise

• How many shift and addition operations we need, if we do not want to use the multiple and division operations?
  – 65*n
  – 65*n = 64*n + n  one shift and one addition

• How many shift and addition operations we need, if we do not want to use the multiple and division operations?
  – 3*n/4
  – (n+2*n)/4  two shift and one addition
Monadic Instructions (2)

- **Rotate**
  - Shifts in which bits pushed off one end reappear on the other end.
  - Useful for packing and unpacking bit sequence from words
  - For example, test all bits in a word. Restore the original word after test all bits

- **Signed Number Representations**
  - One's complement negative number: BITWISE NOT applied to it — the "complement" of its positive counterpart.
  - Example: one's complement of 00101011 (43) becomes 11010100 (−43).
  - Two's complement negative number: The bit pattern which is one greater (in an unsigned sense) than the one's complement of the positive value.
  - Example: An easier method to get the negation of a number in two's complement is as follows:

<table>
<thead>
<tr>
<th>Example 1</th>
<th>Example 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0101001</td>
<td>0101100</td>
</tr>
<tr>
<td>1010111</td>
<td>1010100</td>
</tr>
</tbody>
</table>

1. Starting from the right, find the first '1'
2. Invert all of the bits to the left of that one
Exercise

• One's complement negative number
  – 1111110
  – -1

• One's complement negative number
  – 1111001
  – -6
Exercise

• Negation of a number in two's complement is as follows:
  – 00010101
  – 11101011

• Negation of a number in two's complement is as follows:
  – 00001000
  – 11111000
Monadic Instructions (3)

• Shift Signed Number
  – Left shifting one’s complement negative numbers does not multiply by 2.
  – Right shifting does simulate division correctly, however.

Right Shift are often performed with signed extension

Shifting negative number
Loop Control

- **Loop**
  - Execute a group of instructions a fixed number of times
  - A counter that is increased or decreased by some constant once each time through the loop. The counter is also tested once each time through the loop. If a certain condition holds, the loop is terminated.
  - Test-at-the-end looping has the property that the loop will always be executed at least once, even if \( n \) is less than or equal to 0.

```
(a)  i = 1;
L1:  first-statement;
    .
    .
    last-statement;
    i = i + 1;
    if (i < n) goto L1;

(b)  i = 1;
L1:  if (i > n) goto L2;
    first-statement;
    .
    .
    last-statement
    i = i + 1;
    goto L1;

L2:  
```

Test-at-the-end loop.

Test-at-the-beginning loop.
Input/Output (1)

- **Programming I/O**
  - The simplest possible I/O method, which is commonly used in low-end microprocessors, for example, in embedded systems or in systems that must respond quickly to external changes (real-time systems).
  - These CPUs usually have a single input instruction and a single output instruction.
  - Each of these instructions selects one of the I/O devices.
  - A single character is transferred between a fixed register in CPU and the selected I/O device.
  - The processor must execute an explicit sequence of instructions for each and every character read or written.
Input/Output (2)

**DMA I/O**

- To write a block of 32 bytes from memory address 100 to a terminal (device 4),
- CPU writes 32, 100, 4 and 1 (WRITE) into four DMA registers
- DMA controller makes a bus request to read byte 100 from the memory, the same way the CPU would read from the memory.
- DMA controller then makes an I/O request to device 4 to write the byte to it.
- DMA controller increments its address register by 1 and decrements its count register by 1.
- If the count register is still greater than 0, another byte is read from memory and then written to the device.

DMA always has a higher bus priority than the CPU.
The Pentium 4 Instructions (1)

A selection of the Pentium 4 integer instructions.

<table>
<thead>
<tr>
<th>Moves</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>Arithmetic</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>
## The Pentium 4 Instructions (2)

### 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>
The Pentium 4 Instructions (3)

A selection of the Pentium 4 integer instructions.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEST SRC1,SRC2</td>
<td>Boolean AND operands, set flags</td>
</tr>
<tr>
<td>CMP SRC1,SRC2</td>
<td>Set flags based on SRC1 - SRC2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Transfer of Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMP ADDR</td>
</tr>
<tr>
<td>Jxx ADDR</td>
</tr>
<tr>
<td>CALL ADDR</td>
</tr>
<tr>
<td>RET</td>
</tr>
<tr>
<td>IRET</td>
</tr>
<tr>
<td>LOOPxx</td>
</tr>
<tr>
<td>INT n</td>
</tr>
<tr>
<td>INTO</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Strings</th>
</tr>
</thead>
<tbody>
<tr>
<td>LODS</td>
</tr>
<tr>
<td>STOS</td>
</tr>
<tr>
<td>MOVS</td>
</tr>
<tr>
<td>CMPS</td>
</tr>
<tr>
<td>SCAS</td>
</tr>
</tbody>
</table>
## The Pentium 4 Instructions (4)

### Condition codes

<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>

### Miscellaneous

<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

A selection of the Pentium 4 integer instructions.