Computer Organization &
Assembly Language Programming

CSE 2312
Lecture 11 Addressing

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

- **Instruction Architecture Level**
  - Lay between the microarchitecture level and operating system level

- **Significance**
  - ISA is the interface between the software and the hardware.
  - It is not good to have the hardware directly execute programs written in C, C++, Java, or some other high-level language: 1) the performance advantage of compiling over interpreting would then be lost. 2) most computers have to be able to execute programs written in multiple languages, not just one.

- **One Design Principle**
  - Let programs in various high-level languages to be translated to a common intermediate form—the ISA level
  - Build hardware to be able to execute ISA-level programs directly.

- **Theory vs. Reality**
  - Theory: satisfy both compiler writer and hardware engineer
  - Practice: backward compatible
Reviewing (2): ISA Level

• **Relationship**
  - The ISA level define the interface between the compilers and the hardware.
  - It is the language that both of them have to understand.
  - The relationship among the compilers, the ISA level, and the hardware
Reviewing (3): What make a good ISA?

• Implementation
  – A good ISA should define a set of instructions that can be implemented efficiently in current and future technologies, resulting in cost-effective designs over several generations.
  – A poor design is more difficult to implement and may require many more gates to implement a processor and more memory for executing programs. It also may run slower because the ISA obscures opportunities to overlap operations, requiring much more sophisticated designs to achieve equivalent performance.

• Target for Compiler Code
  – A good ISA should provide a clean target for compiled code.
  – Regularity and completeness of a range of choices are important traits that are not always present in an ISA.
  – Since the ISA is the interface between the hardware and the software, it should make the hardware designers happy (easy to implement efficiently) and make the software designers happy (easy to generate good code for).
Reviewing (4): Properties of the ISA

• Definition
  – The ISA level is defined by how the machine appears to a machine language programmer.
  – Also defined as: ISA level code is what a compiler outputs

• To produce ISA level code, the compiler writer has to know
  – What the memory model is
  – What registers there are
  – What data types and instructions are available
  – ……

• Two Mode
  – Kernel mode is intended to run the OS and allows all instructions to be executed
  – User mode is intended to run application programs and does not permit certain sensitive instructions (manipulate cache directly)
Reviewing (5): Memory and Register

• **Memory Models**
  - All memory requests are serialized. Each one is completed before the next one is issued
  - Execute a SYNC instruction to block issuing new memory operations until all previous ones have completed
  - Intermediate: hardware automatically blocks the issuing of certain memory references but not others

• **Registers**
  - Some of them visible in the ISA level
  - If visible in the ISA level, also visible in the microarchitecture level
  - Control execution of the program
  - Hold temporary results

• **Program Status Word (PSW)**
  - Holds bits that are needed by the CPU
  - Condition Codes: N (negative), Z (zeros), V (overflow), P (even parity)
Reviewing (6): Pentium 4 ISA

- **Operation Mode**
  - Real Mode: like a simple 8088
  - Virtual mode: run old 8088 programs in a protected mode
  - Protected Mode

- **Protected Mode**
  - Level 0: corresponds to kernel mode, used by OS
  - Level 3: used by user program
  - Level 1 and 2, rarely used
Reviewing (7): Design Principles for Instruction Formats

• **Shorter is better**
  – All things being equal, shorter instructions are better
  – Shorter instruction is important for memory bandwidth

\[
\text{Suppose the bandwidth of an instruction cache is} \ t \ \text{bps and the average instruction length is} \ r \ \text{bits, the cache can deliver at most} \ \frac{t}{r} \ \text{instructions}
\]

• **Sufficient Room**
  – To express all the operations desired;
  – \(2^n\) operations at most with \(n\) bit instructions

• **Number bit in an address field**
  – To gain a finer memory resolution, one must pay the price of longer addresses and thus longer instructions
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
Immediate Addressing

An Immediate addressing:

MOV | R1 | 4

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.
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

- **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.
Example: Register Indirect Addressing

Register Indirect Addressing: a generic assembly program for computing the sum of the elements of an array.

- **Register Indirect Addressing**
  - The loop itself does not contain any memory addresses

```
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 (1)

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

```assembly
MOVE R1,#0 ; accumulate the OR in R1, initially 0
MOVE R2,#0 ; R2 = index, i, of current product: \( A[i] \) AND \( B[i] \)
MOVE R3,#4096 ; R3 = first index value not to use
LOOP:
    MOVE R4,A(R2) ; R4 = \( A[i] \)
    AND R4,B(R2) ; R4 = \( A[i] \) AND \( B[i] \)
    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
```

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

A possible representation of **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\textsuperscript{st} loop: the memory word addressed is $A_0 = 124300$
  - 2\textsuperscript{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
Based-Indexed Addressing

LOOP:   MOV R4,(R2+R5)
        AND R4,(R2+R6)

• 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
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: $3+5\times7$ | $35\times7+$
  – For example: $3\times5+7$ | $35\times7+$

• **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?
Train Stack

Each railroad car represents one symbol in the formula to be converted from infix to reverse Polish notation.

- **Rules**
  - Cars containing variables always go directly to CA and never to TX
  - Cars containing others must inquire about the contents of the nearest car on the TX line before entering the switch
### Decision Table

<table>
<thead>
<tr>
<th>Most recently arrived car on the Texas line</th>
<th>Car at the switch</th>
<th>\</th>
<th>+</th>
<th>-</th>
<th>x</th>
<th>/</th>
<th>( )</th>
</tr>
</thead>
<tbody>
<tr>
<td>\top</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>+</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>-</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>x</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>/</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>(</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

Decision table used by the infix-to-reverse Polish notation algorithm

- **Numbers mean:**
  - 1: the cars at switch head to TX
  - 2: the most recent car on the TX line turns and goes to CA
  - 3: both the car at the switch and the most recent car on the TX line are diverted and disappear (i.e., both are deleted).
  - 4: stop. the symbols now in CA represent the reverse Polish notation formula when read from left to right.
  - 5: stop. An error has occurred. The original formula was not correctly balanced.
### Examples

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

\[ (8+2*5)/(1+3*2-4) \]

<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>

Use of a stack to evaluate a reverse Polish notation formula.
Opcodes and Addressing Modes

1. OPCODE 0 DEST SRC1 SRC2

2. OPCODE 1 DEST SRC1 OFFSET

3. OPCODE OFFSET

A simple design for the instruction formats of a three-address machine.

Bits 8 3 5 4 3 5 4

<table>
<thead>
<tr>
<th>OPCODE</th>
<th>MODE</th>
<th>REG</th>
<th>OFFSET</th>
<th>MODE</th>
<th>REG</th>
<th>OFFSET</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(Optional 32-bit direct address or offset)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(Optional 32-bit direct address or offset)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A simple design for the instruction formats of a two-address machine.
The Pentium 4 Addressing Modes (1)

The Pentium 4 32-bit addressing modes. M[\(x\)] is the memory word at \(x\).

Operand is read from the memory address contained in the EAX register.
The Pentium 4 Addressing Modes (2)

EBP: point to the base of the stack frame containing the local variables and arrays
Exercise

• Convert to reverse polish notation
  – \((A-B)\cdot(C+D)+E\)
    
  – \(AB-CD+\cdot E+\)

• Convert to reverse polish notation
  – \((A-B)\cdot(((C-D\cdot E)/F)/G)\cdot H\)
    
  – \(AB-CDE\cdot-F/G/\cdot H\cdot\)
Exercise

• *Covert to infix*
  – AB-C+D*

  – (A-B+C)*D

• *Covert to infix*
  – ABCDE+**/

  – A/(B*(C*(D+E)))