IA-32 Instruction Set Architecture

CS 365 Lecture 4
Prof. Yih Huang

General-Purpose Registers

<table>
<thead>
<tr>
<th>Assembly Name</th>
<th>Reg #</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>000</td>
</tr>
<tr>
<td>EBX</td>
<td>001</td>
</tr>
<tr>
<td>ECX</td>
<td>010</td>
</tr>
<tr>
<td>EDX</td>
<td>011</td>
</tr>
<tr>
<td>ESP</td>
<td>100</td>
</tr>
<tr>
<td>EDP</td>
<td>101</td>
</tr>
<tr>
<td>ESI</td>
<td>110</td>
</tr>
<tr>
<td>EDI</td>
<td>111</td>
</tr>
</tbody>
</table>
We also have BX, BH, BL, CX, CH CL, DX, CH, CL.

SP, BP, SI, DI are lower-halves of the other 4 registers.

When operating on 16-bit data, the 7 register numbers (000 – 111) refers to AX, BX, CX, DX, SP, BP, SI and DI.

When operating on 8-bit data, the 7 register numbers (000 – 111) refers to AL, CL, DL, BL, AH, CL, DH and BL.

Data width is specified by the opcode.
Instruction Format

<table>
<thead>
<tr>
<th>Opcode</th>
<th>ModR/M</th>
<th>SIB</th>
<th>displacement</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 or 2 bytes</td>
<td>1 byte,</td>
<td>1 byte,</td>
<td>1,2 or 4 bytes</td>
<td>1,2 or 4 bytes</td>
</tr>
<tr>
<td>If required</td>
<td>If required</td>
<td>If required</td>
<td>If required</td>
<td></td>
</tr>
</tbody>
</table>

- Opcode: determine the action
- ModR/M: Addressing modes register/memory
- SIB: Scale-Index-Base
- Not all fields are present in all instr.
- If present, must be in the above order

ModR/M

<table>
<thead>
<tr>
<th>Mod</th>
<th>Reg #</th>
<th>R/M</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 bits</td>
<td>3 bits</td>
<td>3 bits</td>
</tr>
</tbody>
</table>

- **Mod=00,**
  - First operand a register, specified by Reg #
  - Second operand in memory; address stored in a register numbered by R/M.
    - That is, Memory[Reg[R/M]]
  - Exceptions:
    - R/M=100 (SP): SIB needed
    - R/M=101 (BP): disp32 needed
- **Mod=01**, same as Mod 00 with 8-bit displacement.
  - Second operand: Memory[disp8+Reg[R/M]].
  - Exception: SIB needed when R/M=100

- **Mod=10**, same as Mod 01 with 32-bit displacement

- **Mod=11**
  - Second operand is also a register, numbered by R/M.

- Do not confuse displacement width with data width.
  - Data width is specified by the opcode.
  - For example, the use of disp8 does not imply 8-bit data.

  *For some opcodes, the reg# is used as an extension of the opcode.*
SIB

<table>
<thead>
<tr>
<th>Scale</th>
<th>Index</th>
<th>Base</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 bits</td>
<td>3 bits</td>
<td>3 bits</td>
</tr>
</tbody>
</table>

- Specify how a memory address is calculated
- Address = Reg[base] + Reg[Index] * 2^{scale}
- Exceptions:
  - SP cannot be an index, and
  - BP cannot be a base.

Example: Add Instructions

- The first operand is the destination.
  - Can be register or memory
- The second operand is the source
  - Can be register or memory
- The two operands cannot be both memory.
- Action: dest += source
<table>
<thead>
<tr>
<th>04</th>
<th>immd8</th>
<th>AL += immd8</th>
</tr>
</thead>
<tbody>
<tr>
<td>05</td>
<td>immd32</td>
<td>EAX += immd32</td>
</tr>
<tr>
<td>00</td>
<td>modRM</td>
<td>Rm8 += r8</td>
</tr>
<tr>
<td>01</td>
<td>modRM</td>
<td>Rm32 += r32</td>
</tr>
<tr>
<td>03</td>
<td>modRM</td>
<td>r32 += rm32</td>
</tr>
<tr>
<td>80</td>
<td>11000</td>
<td>immd8</td>
</tr>
<tr>
<td>81</td>
<td>11000</td>
<td>immd32</td>
</tr>
</tbody>
</table>

**Even Longer Varieties?**
**Multiplication**

- **Action:** EDX:EAX $\rightarrow$ EAX $\times$ Rm32
- Notice that the multiplier is fixed. It must be EAX.
- The multiplicand can be register or memory.

---

**Special Purpose Instructions**

- Decimal arithmetic
- Strings
- MMX
- SIMD (single instruction multiple data)
MIPS versus IA32

- Fixed instruction formats of MIPS
  - Simple decoding logic
  - Waste of memory space
  - Limited addressing modes
- Variable length formats of IA32
  - Difficult to decode; sequential decoding
  - Compact machine codes
  - Accommodate versatile addressing modes

- Large pool of general purpose registers in MIPS.
  - No special considerations for particular opcodes/registers; everything is born equal.
    - Well, there are exceptions. Can you name one?
  - Simplify programming and program optimizations
  - Good for compilations
Small pool of registers in IA32

- Small amount of data stored inside CPU
  - Recall that moving data between CPU and memory is slow, compared to pure register operations.
  - Usually lead to inefficient code
- Many registers serve special purposes; making programmer/compiler’s job difficult
  - Again could lead to inefficient code

Operand architecture of MIPS

- Uses three register operands
- All data must be (explicitly) moved into registers before the CPU and manipulate them.
- Results have to be explicitly stored back to memory.
- Creates longer machine codes but reflects the reality.
Operand architecture of IA32

- One or two operands
- Operands in some instructions are fixed and implied
  - Compact code but lack flexibilities
  - Makes code optimizations difficult
- One operand can be memory
  - No explicit load/stores; compact code
  - Data are moved in/out of CPU anyway; no gain in performance

IA32 has to be backward compatible with previous 8/16 bit architectures.

- This contributes to its complexities, many of which unnecessarily so
- However, Intel gets to keep its software and customer base. BIG PLUS.
- Intel commands huge resources to push improvements.
- The result is IA32 chips are generally on par with other modern ISAs.
MIPS representation a new generation of computer architectures.

– Called Reduced Instruction Set Computer (RISC)
– No corpses to carry; clean designs
– Everything is purposely kept simple.
– In theory, this shortens design cycles and produces efficient implementations.
– In reality, you need people and money to compete with Intel. Very difficult.