CS356 Unit 4

Intro to x86 Instruction Set
Why Learn Assembly

• To understand something of the limitation of the HW we are running on
• Helpful to understand performance
• To utilize certain HW options that high-level languages don't allow (e.g. operating systems, utilizing special HW features, etc.)
• To understand possible security vulnerabilities or exploits
• Can help debugging
Compilation Process

• Demo of assembler
  – $ g++ -Og -c -S file1.cpp

• Demo of hexdump
  – $ g++ -Og -c file1.cpp
  – $ hexdump -C file1.o | more

• Demo of objdump/disassembler
  – $ g++ -Og -c file1.cpp
  – $ objdump -d file1.o

```
void abs(int x, int* res)
{
  if(x < 0)
    *res = -x;
  else
    *res = x;
}
```

Disassembly of section .text:

```
0000000000000000 <_Z3absiPi>:
  0: 85 ff  test  %edi,%edi
  2: 79 05  jns   9 <_Z3absiPi+0x9>
  4: f7 df  neg   %edi
  6: 89 3e  mov   %edi,(%rsi)
  8: c3    retq
  9: 89 3e  mov   %edi,(%rsi)
 b:  c3    retq
```

Original Code

Notice how each instruction is turned into binary (shown in hex)
Where Does It Live

- Match (1-Processor / 2-Memory / 3-Disk Drive) where each item resides:
  - Source Code (.c/.java) = 3
  - Running Program Code = 2
  - Global Variables = 2
  - Compiled Executable (Before It Executes) = 3
  - Current Instruction Being Executed = 1
  - Local Variables = 2
BASIC COMPUTER ORGANIZATION
Processor

- Performs the same 3-step process over and over again
  - Fetch an instruction from memory
  - Decode the instruction
    - Is it an ADD, SUB, etc.?
  - Execute the instruction
    - Perform the specified operation
- This process is known as the Instruction Cycle
Processor

- 3 Primary Components inside a processor
  - ALU
  - Registers
  - Control Circuitry
- Connects to memory and I/O via **address**, **data**, and **control** buses (**bus** = group of wires)
Arithmetic and Logic Unit (ALU)

- Digital circuit that performs arithmetic operations like addition and subtraction along with logical operations (AND, OR, etc.)
Registers

- Recall memory is **SLOW** compared to a processor
- Registers provide **fast, temporary** storage locations within the processor
General Purpose Registers

- Registers available to software instructions for use by the programmer/compiler
- Programmer/compiler is in charge of using these registers as inputs (source locations) and outputs (destination locations)
What if we didn’t have registers?

- Example w/o registers: \( F = (X+Y) - (X*Y) \)
  - Requires an ADD instruction, MULtiply instruction, and SUBtract Instruction
  - w/o registers
    - ADD: Load X and Y from memory, store result to memory
    - MUL: Load X and Y again from mem., store result to memory
    - SUB: Load results from ADD and MUL and store result to memory
    - 9 memory accesses
What if we have registers?

- Example w/ registers: \( F = (X+Y) - (X*Y) \)
  - Load \( X \) and \( Y \) into registers
  - ADD: \( R0 + R1 \) and store result in \( R2 \)
  - MUL: \( R0 * R1 \) and store result in \( R3 \)
  - SUB: \( R2 - R3 \) and store result in \( R4 \)
  - Store \( R4 \) back to memory
  - 3 total memory access
Other Registers

• Some bookkeeping information is needed to make the processor operate correctly
• Example: Program Counter/Instruction Pointer (PC/IP) Reg.
  – Recall that the processor must fetch instructions from memory before decoding and executing them
  – PC/IP register holds the address of the next instruction to fetch
Fetching an Instruction

- To fetch an instruction
  - PC/IP contains the address of the instruction
  - The value in the PC/IP is placed on the address bus and the memory is told to read
  - The PC/IP is incremented, and the process is repeated for the next instruction
Fetching an Instruction

- To fetch an instruction
  - PC/IP contains the address of the instruction
  - The value in the PC/IP is placed on the address bus and the memory is told to read
  - The PC/IP is incremented, and the process is repeated for the next instruction
Control Circuitry

• Control circuitry is used to decode the instruction and then generate the necessary signals to complete its execution
• Controls the ALU
• Selects Registers to be used as source and destination locations
Control Circuitry

• Assume 0x0201 is machine code for an ADD instruction of \( R2 = R0 + R1 \)
• Control Logic will...
  – select the registers (R0 and R1)
  – tell the ALU to add
  – select the destination register (R2)
Summary

• Registers are used for fast, temporary storage in the processor
  – Data (usually) must be moved into registers
• The PC or IP register stores the address of the next instruction to be executed
  – Maintains the current execution location in the program
UNDERSTANDING MEMORY
Memory and Addresses

• Set of cells that each store a group of bits
  – Usually, 1 byte (8 bits) per cell

• Unique address (number) assigned to each cell
  – Used to reference the value in that location

• **Data** and **instructions** are both stored in memory and are always represented as a string of 1’s and 0’s

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>11010010</td>
</tr>
<tr>
<td>1</td>
<td>01001011</td>
</tr>
<tr>
<td>2</td>
<td>10010000</td>
</tr>
<tr>
<td>3</td>
<td>11101000</td>
</tr>
<tr>
<td>4</td>
<td>01101000</td>
</tr>
<tr>
<td>5</td>
<td>11010001</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>FFFF</td>
<td>00001011</td>
</tr>
</tbody>
</table>

Data Inputs/Outputs

Address Inputs

Memory Device
Reads & Writes

- Memories perform 2 operations:
  - **Read**: retrieves data value in a particular location (specified using the address)
  - **Write**: changes data in a location to a new value

- To perform these operations a set of **address**, **data**, and **control** wires are used to talk to the memory:
  - **Note**: A group of wires/signals is referred to as a ‘bus’
  - **Thus**, we say that memories have an **address**, **data**, and **control bus**.
Memory vs. I/O Access

- Processor performs reads and writes to communicate with memory and I/O devices
  - I/O devices have memory locations that contain data that the processor can access
  - All memory locations (be it RAM or I/O) have unique addresses which are used to identify them
  - The assignment of memory addresses is known as the physical memory map

Video Interface

- FE may signify a white dot at a particular location

Processor

Memory

Keyboard Interface

- 'a' = 61 hex in ASCII
Address Space Size and View

• Most computers are *byte-addressable*
  – Each unique address corresponds to 1-byte of memory (so we can access char variables)

• Address width determines max amount of memory
  – Every byte of data has a unique address
  – 32-bit addresses => 4 GB address space
  – 36-bit address bus => 64 GB address space
Data Bus & Data Sizes

- Moore's Law meant we could build systems with more transistors
- More transistors meant greater bit-widths
  - Just like more physical space allows for wider roads/freeways, more transistors allowed us to move to 16-, 32- and 64-bit circuitry inside the processor
- To support smaller variable sizes (char = 1-byte) we still need to access only 1-byte of memory per access, but to support int and long ints we want to access 4- or 8-byte chunks of memory per access
- Thus the data bus (highway connecting the processor and memory) has been getting wider (i.e. 64-bits)
  - The processor can use 8-, 16-, 32- or all 64-bits of the bus (lanes of the highway) in a single access based on the size of data that is needed

<table>
<thead>
<tr>
<th>Processor</th>
<th>Data Bus Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel 8088</td>
<td>8-bit</td>
</tr>
<tr>
<td>Intel 8086</td>
<td>16-bit</td>
</tr>
<tr>
<td>Intel 80386</td>
<td>32-bit</td>
</tr>
<tr>
<td>Intel Pentium</td>
<td>64-bit</td>
</tr>
</tbody>
</table>
## Intel Architectures

<table>
<thead>
<tr>
<th>Processor</th>
<th>Year</th>
<th>Address Size</th>
<th>Data Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>8086</td>
<td>1978</td>
<td>20</td>
<td>16</td>
</tr>
<tr>
<td>80286</td>
<td>1982</td>
<td>24</td>
<td>16</td>
</tr>
<tr>
<td>80386/486</td>
<td>’85/’89</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>Pentium</td>
<td>1993</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>Pentium 4</td>
<td>2000</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>Core 2 Duo</td>
<td>2006</td>
<td>36</td>
<td>64</td>
</tr>
<tr>
<td>Core i7 (Haswell)</td>
<td>2013</td>
<td>39</td>
<td>64</td>
</tr>
</tbody>
</table>
x86-64 Data Sizes

Integer
- 4 Sizes Defined
  - Byte (B)
    - 8-bits
  - Word (W)
    - 16-bits = 2 bytes
  - Double word (L)
    - 32-bits = 4 bytes
  - Quad word (Q)
    - 64-bits = 8 bytes

Floating Point
- 3 Sizes Defined
  - Single (S)
    - 32-bits = 4 bytes
  - Double (D)
    - 64-bits = 8 bytes
    - (For a 32-bit data bus, a double would be accessed from memory in 2 reads)

In x86-64, instructions generally specify what size data to access from memory and then operate upon.
x86-64 Memory Organization

- Because each byte of memory has its own address we can picture memory as one column of bytes (Fig. 2)
- But, 64-bit logical data bus allows us to access up to 8-bytes of data at a time
- We will usually show memory arranged in rows of 4-bytes (Fig. 3) or 8-bytes
  - Still with separate addresses for each byte

```
int x, y = 5; z = 8;
x = y + z;
```

![Logical Byte-Oriented View of Mem.](Fig. 2)

![Logical DWord-Oriented View](Fig. 3)
Memory & Word Size

- To refer to a chunk of memory we must provide:
  - The starting address
  - The size: B, W, D, L
- There are rules for valid starting addresses
  - A valid starting address must be a multiple of the data size
  - Words (2-byte chunks) must start on an even (divisible by 2) address
  - Double words (4-byte chunks) must start on an address that is a multiple of (divisible by) 4
  - Quad words (8-byte chunks) must start on an address that is a multiple of (divisible by) 8
**Endian-ness**

- **Endian-ness** refers to the two alternate methods of ordering the **bytes** in a larger unit (word, DWORD, etc.)
  - Big-Endian
    - PPC, Sparc
    - **MS byte** is put at the starting address
  - Little-Endian
    - used by Intel processors / original PCI bus
    - **LS byte** is put at the starting address
- Some processors (like ARM) and busses can be configured for either big- or little-endian

The DWORD value:

```
0 x 1 2 3 4 5 6 7 8
```

can be stored differently

<table>
<thead>
<tr>
<th></th>
<th>Big-Endian</th>
<th></th>
<th>Little-Endian</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>12</td>
<td>0x00</td>
<td>78</td>
</tr>
<tr>
<td>0x01</td>
<td>34</td>
<td>0x01</td>
<td>56</td>
</tr>
<tr>
<td>0x02</td>
<td>56</td>
<td>0x02</td>
<td>34</td>
</tr>
<tr>
<td>0x03</td>
<td>78</td>
<td>0x03</td>
<td>12</td>
</tr>
</tbody>
</table>
Big-endian vs. Little-endian

- **Big-endian**
  - makes sense if you view your memory as starting at the top-left and addresses increasing as you go down

- **Little-endian**
  - makes sense if you view your memory as starting at the bottom-right and addresses increasing as you go up
Big-endian vs. Little-endian

- Issues arise when transferring data between different systems
  - Byte-wise copy of data from big-endian system to little-endian system
  - Major issue in networks (little-endian computer => big-endian computer) and even within a single computer (System memory => I/O device)

Addresses increasing downward:

Big-Endian

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>11345678</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000004</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000008</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000010</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000014</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Addresses increasing upward:

Little-Endian

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>78563412</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000014</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000010</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000008</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000004</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Byte 0: 1 2 3 4 5 6 7 8
Byte 1: 7 8 5 6 3 4 1 2

DWORD @ 0 in big-endian system is now different than DWORD @ 0 in little-endian system

Intel is LITTLE-ENDIAN

Copy byte 0 to byte 0, byte 1 to byte 1, etc.

DWORD @ addr. 0
Summary

• The processor communicates with all other components in the processor via reads/writes using unique addresses for each component
• Memory can be accessed in different size chunks (byte, word, dword, quad word)
• Alignment rules: data of size \( n \) should start on an address that is a multiple of size \( n \)
  – dword should start on multiples of 4
  – Size 8 should start on an address that is a multiple of 4
• x86 uses little-endian
  – The start address of a word (or dword or qword) refers to the LS-byte
X86-64 ASSEMBLY
x86-64 Instruction Classes

• Data Transfer (mov instruction)
  – Moves data between processor & memory (loads and saves variables between processor and memory)
  – One operand must be a processor register (can't move data from one memory location to another)
  – Specifies size via a suffix on the instruction (movb, movw, movl, movq)

• ALU Operations
  – One operand must be a processor register
  – Size and operation specified by instruction (addl, orq, andb, subw)

• Control / Program Flow
  – Unconditional/Conditional Branch (cmpq, jmp, je, jne, jl, jge)
  – Subroutine Calls (call, ret)

• Privileged / System Instructions
  – Instructions that can only be used by OS or other “supervisor” software (e.g. int to access certain OS capabilities, etc.)
Operand Locations

• Source operands must be in one of the following 3 locations:
  – A register value (e.g. %rax)
  – A value in a memory location (e.g. value at address 0x0200e8)
  – A constant stored in the instruction itself (known as an ‘immediate’ value)
    [e.g. ADDI $1,D0]
  – The $ indicates the constant/immediate

• Destination operands must be
  – A register
  – A memory location (specified by its address)
Intel x86 Register Set

• 8-bit processors in late 1970s
  – 4 registers for integer data: A, B, C, D
  – 4 registers for address/pointers: SP (stack pointer), BP (base pointer), SI (source index), DI (dest. index)

• 16-bit processors extended registers to 16-bits but continued to support 8-bit access
  – Use prefix/suffix to indicate size: AL referenced the lower 8-bits of register A, AH referenced the high 8-bits, AX referenced the full 16-bit value

• 32-/64-bit processors (see next slide)
Intel (IA-32/64) Architectures

General Purpose Registers

<table>
<thead>
<tr>
<th>63</th>
<th>31</th>
<th>15</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAX</td>
<td>EAX</td>
<td>AX</td>
<td>AL</td>
<td></td>
</tr>
<tr>
<td>RBX</td>
<td>EBX</td>
<td>BX</td>
<td>BL</td>
<td></td>
</tr>
<tr>
<td>RCX</td>
<td>ECX</td>
<td>CX</td>
<td>CL</td>
<td></td>
</tr>
<tr>
<td>RDX</td>
<td>EDX</td>
<td>DX</td>
<td>DL</td>
<td></td>
</tr>
<tr>
<td>RSP</td>
<td>ESP</td>
<td>SP</td>
<td>Stack Pointer</td>
<td></td>
</tr>
<tr>
<td>RBP</td>
<td>EBP</td>
<td>EBP</td>
<td>Base “Frame” Ptr.</td>
<td></td>
</tr>
<tr>
<td>RSI</td>
<td>ESI</td>
<td>SI</td>
<td>Source Index</td>
<td></td>
</tr>
<tr>
<td>RDI</td>
<td>EDI</td>
<td>DI</td>
<td>Dest. Index</td>
<td></td>
</tr>
</tbody>
</table>

Special Purpose Registers

<table>
<thead>
<tr>
<th>Pointer/Index Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>RIP</td>
</tr>
<tr>
<td>EIP (Instruction Pointer)</td>
</tr>
</tbody>
</table>

Status Register

| EFLAGS |

<table>
<thead>
<tr>
<th>Pointer/Index Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>R8</td>
</tr>
<tr>
<td>R8D</td>
</tr>
<tr>
<td>R8W</td>
</tr>
<tr>
<td>R8B</td>
</tr>
<tr>
<td>R9</td>
</tr>
<tr>
<td>R9D</td>
</tr>
<tr>
<td>R9W</td>
</tr>
<tr>
<td>R9B</td>
</tr>
<tr>
<td>R15</td>
</tr>
<tr>
<td>R15D</td>
</tr>
<tr>
<td>R15W</td>
</tr>
<tr>
<td>R15B</td>
</tr>
</tbody>
</table>
DATA TRANSFER INSTRUCTIONS
mov Instruction & Data Size

- **Moves data between memory and processor register**
- Always provide the **LS-Byte address (little-endian)** of the desired data
- Size is explicitly defined by the instruction suffix (‘mov[bwlq]’) used
- Recall: Start address **should** be divisible by size of access

(Assume start address = A)

<table>
<thead>
<tr>
<th>Processor Register</th>
<th>Memory / RAM</th>
<th>Byte operations only access the 1-byte at the specified address</th>
<th>Word operations access the 2-bytes starting at the specified address</th>
<th>Word operations access the 4-bytes starting at the specified address</th>
<th>Word operations access the 8-bytes starting at the specified address</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>7</td>
<td><strong>movb</strong></td>
<td><strong>movw</strong></td>
<td><strong>movl</strong></td>
<td><strong>movq</strong></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td><strong>movb</strong> leaves upper bits unaffected</td>
<td><strong>movw</strong> leaves upper bits unaffected</td>
<td><strong>movl</strong> leaves upper bits unaffected</td>
<td><strong>movq</strong> leaves upper bits unaffected</td>
</tr>
<tr>
<td>63</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000 0000</td>
<td><strong>Double Word</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td><strong>movl</strong> zeros the upper bits</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>63</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Mem/Register Transfer Examples

- mov[b,w,l,q] src, dst

- Initial Conditions:
  - movl 0x204, %eax
  - movw 0x202, %ax
  - movb 0x207, %al
  - movq 0x200, %rax

- movb %al, 0x4e5
- movl %eax, 0x4e0

Treat these instructions as a sequence where one affects the next.
Immediate Examples

- `movl $0xfe1234, %eax`
- `movw $0xaa55, %ax`
- `movb $20, %al`
- `movq $-1, %rax`
- `movabsq $0x123456789ab, %rax`
- `movq $-1, 0x4e0`

**Rules:**
- Immediate operands must be source operands.
- Indicate with `$` and can be specified in decimal (default) or hex (start with 0x).
- `movq` can only support a 32-bit immediate (and will then sign-extend that value to fill the upper 32-bits).
- Use `movabsq` for a full 64-bit immediate value.
Move Variations

- There are several variations when the destination of a `mov` instruction is a register
  - This only applies when the destination is a register
- Normal `mov` **does not affect upper portions** of registers (with exception of `movl`)
- `movzxy` will **zero-extend** the upper portion
  - `movzbw` (move a byte from the source but zero-extend it to a word in the dest. register)
  - `movzbxw`, `movzbl`, `movzbq`, `movzwl`, `movzwl`, `movzwlq`
- `movsxy` will **sign-extend** the upper portion
  - `movsbw` (move a byte from the source but sign-extend it to a word in the dest. register)
  - `movsbl`, `movsbl`, `movsbq`, `movswl`, `movswq`, `movswlq`
Zero/Signed Move Variations

- Initial Conditions:
  - movslq 0x200, %rax
  - movzwl 0x202, %eax
  - movsbw 0x201, %ax
  - movsbl 0x206, %eax
  - movzbpq %dl, %rax

Treat these instructions as a sequence where one affects the next.
Why So Many Oddities & Variations

• The x86 instruction set has been around for nearly 40 years and each new processor has had to maintain backward compatibility (support the old instruction set) while adding new functionality

• If you wore one clothing article from each decade you'd look funny too and have a lot of oddities
Summary

• To access different size portions of a register requires different names in x86 (e.g. AL, AX, EAX, RAX)

• Moving to a register may involve zero- or sign-extending since registers are 64-bits
  – Long (dword) operations always 0-extend the upper 32-bits

• Moving to memory never involves zero- or sign-extending since it memory is broken into finer granularities
ADDRESSING MODES
What Are Addressing Modes

• Recall an operand must be:
  – A register value (e.g. %rax)
  – A value in a memory location
  – An immediate

• To access a memory location we must supply an address
  – However, there can be many ways to compute an address, each useful in particular contexts [e.g. accessing an array element, a[i] vs. object member, obj.member]

• The ways to specify the operand location are known as addressing modes
# Common x86-64 Addressing Modes

<table>
<thead>
<tr>
<th>Name</th>
<th>Form</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>$\text{imm}$</td>
<td><code>movl \$-500,\%rax</code></td>
<td>$R[\text{rax}] = \text{imm}$.</td>
</tr>
<tr>
<td>Register</td>
<td>$\text{r}_a$</td>
<td><code>movl \%rdx,\%rax</code></td>
<td>$R[\text{rax}] = R[\text{rdx}]$.</td>
</tr>
<tr>
<td>Indirect Addressing</td>
<td>$(\text{r}_a)$</td>
<td><code>movl (\%rdx),\%rax</code></td>
<td>$R[\text{rax}] = M[R[\text{r}_a]]$.</td>
</tr>
<tr>
<td>Base w/ Displacement</td>
<td>imm$(\text{r}_b)$</td>
<td><code>movl 40(\%rdx),\%rax</code></td>
<td>$R[\text{rax}] = M[R[\text{r}_b]+40]$.</td>
</tr>
<tr>
<td>Scaled Index</td>
<td>$(\text{r}_b,\text{r}_i,\text{s}⁺)$</td>
<td><code>movl (\%rdx,\%rcx,4),\%rax</code></td>
<td>$R[\text{rax}] = M[R[\text{r}_b]+R[\text{r}_i]*\text{s}]$.</td>
</tr>
<tr>
<td>Scaled Index w/ Displacement</td>
<td>imm$(\text{r}_b,\text{r}_i,\text{s}⁺)$</td>
<td><code>movl 80(\%rdx,\%rcx,2),\%rax</code></td>
<td>$R[\text{rax}] = M[80+R[\text{r}_b]+R[\text{r}_i]*\text{s}]$.</td>
</tr>
</tbody>
</table>

†Known as the scale factor and can be \{1,2,4, or 8\}

Imm = Constant, $R[x]$ = Content of register $x$, $M[\text{addr}]$ = Content of memory @ addr.

*Purple values = effective address (EA) = Actual address used to get the operand*
Register Mode

- Specifies the contents of a register as the operand

Both operands in this example are using Register Mode.

- Initial val. of `%rdx` = `ffff` _f_f_f_f
- `%rdx` = `0000_0000_1234_5678`

Intruc: `movq %rax, %rdx`

Processor:

- `rax`:
  - `0000_0000_1234_5678`
- `rbx`:
  - `0000_0000_0000_0200`
- `rcx`:
  - `0000_0000_0000_0002`
- `rdx`:
  - `0000_0000_1234_5678`

Memory / RAM:

- `cc55 aa33` @ `0x00208`
- `7654 3210` @ `0x00204`
- `fedc ba98` @ `0x00200`
Immediate Mode

- Specifies the constant stored in the instruction as the operand.
- Immediate is indicated with '$' and can be specified in hex or decimal.

**Example Instruction:**

```
movw $5, %dx
```

**Memory / RAM:**

```
<table>
<thead>
<tr>
<th></th>
<th>0x00208</th>
<th>0x00204</th>
</tr>
</thead>
<tbody>
<tr>
<td>cc55 aa33</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7654 3210</td>
<td>0x00204</td>
<td></td>
</tr>
<tr>
<td>fedc ba98</td>
<td>0x00200</td>
<td></td>
</tr>
</tbody>
</table>
```

**Processor State:**

- Initial val. of %rdx = ffff ffff ffff ffff
- Initial val. of %rdx = ffff ffff ffff 0005
Direct Addressing Mode

- Specifies a constant memory address where the true operand is located
- Address can be specified in decimal or hex

Intruc: `movb 0x20a, %dl`

Processor:
- `rax`: 0000 0000 1234 5678
- `rbx`: 0000 0000 0000 0200
- `rcx`: 0000 0000 0000 0002
- `rdx`: ffff ffff ffff fff5

Initial val. of `%rdx` = ffff ffff ffff ffff

Source is using Direct Addressing mode

Memory / RAM:
- `cc55 aa33`
- `7654 3210`
- `fedc ba98`
Indirect Addressing Mode

- Specifies a register whose value will be used as the effective address in memory where the true operand is located
  - Similar to dereferencing a pointer
- Parentheses indicate indirect addressing mode

Intruc: \texttt{movl} (%rbx), %edx

Processor:

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>rax</td>
<td>0000 0000 1234 5678</td>
</tr>
<tr>
<td>rbx</td>
<td>0000 0000 0000 0200</td>
</tr>
<tr>
<td>rcx</td>
<td>0000 0000 0000 0002</td>
</tr>
</tbody>
</table>

Memory / RAM:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00208</td>
<td>cc55 aa33</td>
</tr>
<tr>
<td>0x00204</td>
<td>7654 3210</td>
</tr>
<tr>
<td>0x00200</td>
<td>fedc ba98</td>
</tr>
</tbody>
</table>

Initial val. of %rdx = ffff ffff ffff ffff

EA = fedc ba98
Base/Indirect with Displacement Addressing Mode

- Form: \( d(\%\text{reg}) \)
- Adds a constant displacement to the value in a register and uses the sum as the effective address of the actual operand in memory

**Source is using Base with Displacement Addressing mode**

- Instruction: `movw 8(\%rbx), %dx`
  - Initial val. of \%rdx = ffff ffff ffff ffff
  - \%rdx = ffff ffff ffff aa33
Base/Indirect with Displacement Example

- Useful for access members of a struct or object

```c
struct mystruct {
    int x;
    int y;
};
struct mystruct data[3];

int main()
{
    for(i=0; i<3; i++){
        data[i].x = 1;
        data[i].y = 2;
    }
}
```

**Memory / RAM**

```
data[2].y     0000 0002  0x00214
data[2].x     0000 0001  0x00210
data[1].y     0000 0002  0x0020c
data[1].x     0000 0001  0x00208
data[0].y     0000 0002  0x00204
data[0].x     0000 0001  0x00200
```

**Assembly**

```
movq $0x0200,%rbx
loop 3 times {
    movl $1, (%rbx)
    movl $2, 4(%rbx)
    addq $8, %rbx
}
```
Scaled Index Addressing Mode

• Form: (%reg1,%reg2,s) [s = 1, 2, 4, or 8]
• Uses the result of %reg1 + %reg2*s as the effective address of the actual operand in memory

Initial val. of %rdx = ffffffff
rdx 0000 0000 cc55 aa33

movl (%rbx,%rcx,4), %edx

Source is using Scaled Index Addressing mode
Scaled Index Addressing Mode Example

- Useful for accessing array elements

```c
int data[6];

int main()
{
    for(int i=0; i<6; i++){
        data[i] = i;
        // *(startAddr+4*i) = i;
    }
}  
```

**C Code**

```
movq $0x0200,%rbx
movl $0, %rcx
Loop 6 times {
    movl %rcx, (%rbx,%rcx,4)
    addl $1, %rcx
}
```

**Assembly**

---

Array of:
- chars/bytes => Use s=1
- shorts/words => Use s=2
- ints/floats/dwords => Use s=4
- long longs/doubles/qwords => Use s=8

Memory / RAM
```
data[5]    0000 0005  0x00214
data[4]    0000 0004  0x00210
data[3]    0000 0003  0x0020c
data[2]    0000 0002  0x00208
data[1]    0000 0001  0x00204
data[0]    0000 0000  0x00200
```
Scaled Index w/ Displacement Addressing Mode

- Form: \( d(\%\text{reg1}, \%\text{reg2}, s) \) [s = 1, 2, 4, or 8]
- Uses the result of \( d + \%\text{reg1} + \%\text{reg2} \times s \) as the effective address of the actual operand in memory
Addressing Mode Exercises

- `movq (%rbx), %rax`
- `movl -4(%rbx), %eax`
- `movb (%rbx,%rcx), %al`
- `movw (%rbx,%rcx,2), %ax`
- `movsbl -16(%rbx,%rcx,4), %eax`
- `movw %cx, 0xe0(%rbx,%rcx,2)`
### Addressing Mode Examples

<table>
<thead>
<tr>
<th></th>
<th>Instruction</th>
<th>Memory Address</th>
<th>%eax</th>
<th>%ecx</th>
<th>%edx</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>movl $0x7000,%eax</td>
<td>0x0000 7000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>movl $2,%ecx</td>
<td>0x0000 0002</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>movb (%eax),%dl</td>
<td>0x0000 001d</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>movb %dl,9(%eax)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>movw (%eax,%ecx),%dx</td>
<td>0x0000 1a1b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>movw %dx,6(%eax,%ecx,2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Main Memory

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1A</td>
<td>1B</td>
<td>1D</td>
<td>00</td>
<td></td>
<td>7008</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td></td>
<td>7004</td>
</tr>
<tr>
<td>1A</td>
<td>1B</td>
<td>1C</td>
<td>1D</td>
<td></td>
<td>7000</td>
</tr>
</tbody>
</table>
Instruction Limits on Addressing Modes

- To make the HW faster and simpler, there are restrictions on the combination of addressing modes
  - Aids overlapping the execution of multiple instructions
- Primary restriction is both operands cannot be memory locations
  - `movl 2000, (%eax)` is not allowed since both source and destination are in memory
  - To move mem->mem use two move instructions with a register as the intermediate storage location
- Legal move combinations:
  - Imm -> Reg
  - Imm -> Mem
  - Reg -> Reg
  - Mem -> Reg
  - Reg -> Mem
Summary

• Addressing modes provide variations for how to specify the location of an operand
• EA = Effective Address
  – Computed address used to access memory
ARITHMETIC INSTRUCTIONS
ALU Instruction(s)

- Performs arithmetic/logic operation on the given size of data
- Restriction: Both operands cannot be memory
- Format
  - `add[b,w,l,q] src2, src1/dst`
  - Example 1: `addq %rbx, %rax` ( `%rax += %rbx` )
  - Example 2: `subq %rbx, %rax` ( `%rax -= %rbx` )
Arithmetic/Logic Operations

• **Initial Conditions**

  - addl $0x12300, %eax
  - addq %rdx, %rax
  - andw 0x200, %ax
  - orb 0x203, %al
  - subw $14, %ax
  - addl $0x12345, 0x204

**Processor Registers**

<table>
<thead>
<tr>
<th>Memory / RAM</th>
<th>Processor Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>7654 3210</td>
<td>ffff ffff 1234 5678</td>
</tr>
<tr>
<td>0f0f ff00</td>
<td>rdx</td>
</tr>
<tr>
<td>0x00200</td>
<td>0x00204</td>
</tr>
<tr>
<td>0x00204</td>
<td>rax</td>
</tr>
<tr>
<td>0000 0000</td>
<td>ffff ffff de69 23cd</td>
</tr>
<tr>
<td>cc33 aa55</td>
<td>rax</td>
</tr>
<tr>
<td>rax</td>
<td>ffff ffff de69 2300</td>
</tr>
<tr>
<td>rax</td>
<td>ffff ffff de69 230f</td>
</tr>
<tr>
<td>rax</td>
<td>ffff ffff de69 2301</td>
</tr>
<tr>
<td>rax</td>
<td>7655 5555</td>
</tr>
<tr>
<td>0f0f ff00</td>
<td>0x00204</td>
</tr>
<tr>
<td>0x00200</td>
<td>0x00200</td>
</tr>
</tbody>
</table>

**Rules:**
- addl, subl, etc. zero out the upper 32-bits
- addq, subq, etc. can only support a 32-bit immediate (and will then sign-extend that value to fill the upper 32-bits)
## Arithmetic and Logic Instructions

<table>
<thead>
<tr>
<th>C operator</th>
<th>Assembly</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>add[b,w,l,q] src1,src2/dst</td>
<td>src2/dst += src1</td>
</tr>
<tr>
<td>-</td>
<td>sub[b,w,l,q] src1,src2/dst</td>
<td>src2/dst -= src1</td>
</tr>
<tr>
<td>&amp;</td>
<td>and[b,w,l,q] src1,src2/dst</td>
<td>src2/dst &amp; src1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>or[b,w,l,q] src1,src2/dst</td>
</tr>
<tr>
<td>^</td>
<td>xor[b,w,l,q] src1,src2/dst</td>
<td>src2/dst ^= src1</td>
</tr>
<tr>
<td>~</td>
<td>not[b,w,l,q] src/dst</td>
<td>src/dst = ~src/dst</td>
</tr>
<tr>
<td>-</td>
<td>neg[b,w,l,q] src/dst</td>
<td>src/dst = (~src/dst) + 1</td>
</tr>
<tr>
<td>++</td>
<td>inc[b,w,l,q] src/dst</td>
<td>src/dst += 1</td>
</tr>
<tr>
<td>--</td>
<td>dec[b,w,l,q] src/dst</td>
<td>src/dst -= 1</td>
</tr>
<tr>
<td>* (signed)</td>
<td>imul[b,w,l,q] src1,src2/dst</td>
<td>src2/dst *= src1</td>
</tr>
<tr>
<td>&lt; (signed)</td>
<td>sal cnt, src/dst</td>
<td>src/dst = src/dst &lt;&lt; cnt</td>
</tr>
<tr>
<td>&gt; (signed)</td>
<td>shl cnt, src/dst</td>
<td>src/dst = src/dst &lt;&lt; cnt</td>
</tr>
<tr>
<td>&gt;= (signed)</td>
<td>sar cnt, src/dst</td>
<td>src/dst = src/dst &gt;&gt; cnt</td>
</tr>
<tr>
<td>&lt;= (signed)</td>
<td>shr cnt, src/dst</td>
<td>src/dst = src/dst &gt;&gt; cnt</td>
</tr>
<tr>
<td>==, &lt;, &gt;, &lt;&gt;, &lt;=, =&gt;, !=</td>
<td>cmp[b,w,l,q] src1, src2</td>
<td>cmp performs: src2 - src1</td>
</tr>
<tr>
<td>(src2 ? src1)</td>
<td>test[b,w,l,q] src1, src2</td>
<td>test performs: src1 &amp; src2</td>
</tr>
</tbody>
</table>
**lea Instruction**

- Recall the exotic addressing modes supported by x86

<table>
<thead>
<tr>
<th>Scaled Index w/ Displacement</th>
<th>imm(r_b, r_i, s)</th>
<th>movl 80(%rdx,%rcx,2),%rax</th>
<th>R[rax] = M[80 + R[r_b]+R[r_i]*s]</th>
</tr>
</thead>
</table>

- The hardware has to support the calculation of the effective address (i.e. 2 adds + 1 mul [by 2,4,or 8])
- Meanwhile normal add and mul instructions can only do 1 operation at a time
- Idea: Create an instruction that can use the address calculation hardware but for normal arithmetic ops
- lea = Load Effective Address
  - lea 80(%rdx,%rcx,2),$rax; // $rax=80+%rdx+2*%rcx
  - Computes the "address" and just puts it in the destination (doesn't load anything from memory)
leal Examples

• Initial Conditions

- leal (%edx,%ecx),%eax
- leaq -8(%rbx),%rax
- leaq 12(%rdx,%rcx,2),%rax

Processor Registers:

- rdx 0000 0000 1234 4000
- rbx ffff ffff ff00 0300
- rax 0000 0000 1234 4020
- rax ffff ffff ff00 02f8
- rax 0000 0089 1234 404c

Rules:
- leal zeroes out the upper 32-bits
### Optimization with lea

**Original Code**

```c
// x = %edi
int f1(int x)
{
    return 9*x+1;
}
```

**Unoptimized Output**

```assembly
f1:
    movl %edi,%eax       # tmp=x
    sal 3, %eax          # tmp *= 8
    addl %edi,%eax       # tmp += x
    addl $1, %eax        # tmp += 1
    ret
```

**Optimized With lea Instruction**

```assembly
f1:
    leal 1(%edi,%edi,8),%eax
    ret
```

**x86 Convention:** The return value of a function is expected in %eax / %rax
## mov and add/sub Examples

<table>
<thead>
<tr>
<th>Instruction</th>
<th>M[0x7000]</th>
<th>M[0x7004]</th>
<th>%rax</th>
</tr>
</thead>
<tbody>
<tr>
<td>movl 0x26CE071B, 0x7000</td>
<td>5A13 F87C</td>
<td>2933 ABC0</td>
<td>0000 0000 0000 0000</td>
</tr>
<tr>
<td>movl $0x26CE071B, 0x7000</td>
<td>26CE 071B</td>
<td>2933 ABC0</td>
<td>0000 0000 0000 0000</td>
</tr>
<tr>
<td>movsbw 0x7002,%ax</td>
<td>26CE 071B</td>
<td>2933 ABC0</td>
<td>0000 0000 0000 ffce</td>
</tr>
<tr>
<td>movzwq 0x7004,%rax</td>
<td>26CE 071B</td>
<td>2933 ABC0</td>
<td>0000 0000 0000 abc0</td>
</tr>
<tr>
<td>movw $0xFE44,0x7006</td>
<td>26CE 071B</td>
<td>FF4E ABC0</td>
<td>0000 0000 0000 abc0</td>
</tr>
<tr>
<td>addl 0x7000,%eax</td>
<td>26CE 071B</td>
<td>FF4E ABC0</td>
<td>0000 0000 26CE B2DB</td>
</tr>
<tr>
<td>subb %eax,0x7007</td>
<td>26CE 071B</td>
<td>244E ABC0</td>
<td>0000 0000 26CE B2DB</td>
</tr>
</tbody>
</table>
Compiler Example 1

```
// data = %edi
// val  = %esi
// i    = %edx
int f1(int data[], int* val, int i)
{
    int sum = *val;
    sum += data[i];
    return sum;
}
```

```
f1:
    movl (%esi), %eax
    addl (%edi,%edx,4), %eax
    ret
```

x86 Convention: The return value of a function is expected in %eax / %rax
struct Data {
    char c;
    int d;
};

// ptr = %edi
// x    = %esi
int f1(struct Data* ptr, int x) {
    ptr->c++;
    ptr->d -= x;
}

f1:
    addb $1, (%edi)
    subl %esi, 4(%edi)
    ret

x86 Convention: The return value of a function is expected in %eax / %rax
Compiler output

ASSEMBLY TRANSLATION EXAMPLE
Translation to Assembly

• We will now see some C code and its assembly translation

• A few things to remember:
  – Data variables live in memory
  – Data must be brought into registers before being processed
  – You often need an address/pointer in a register to load/store data to/from memory

• Generally, you will need 4 steps to translate C to assembly:
  – Setup a pointer in a register
  – Load data from memory to a register (mov)
  – Process data (add, sub, and, or, shift, etc.)
  – Store data back to memory (mov)
Translating HLL to Assembly

- Variables are simply locations in memory
  - A variable name really translates to an address in assembly

<table>
<thead>
<tr>
<th>C operator</th>
<th>Assembly</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>int x,y,z; [...] [z = x + y;]</td>
<td>movl $0x10000004,%ecx \noml (%ecx), %eax \addl 4(%ecx), %eax \movl %eax, 8(%ecx)</td>
<td>Assume x @ 0x10000004 &amp; y @ 0x10000008 &amp; z @ 0x1000000C</td>
</tr>
<tr>
<td>char a[100]; [...] [a[1]--;]</td>
<td>movl $0x10000000c,%ecx \decb 1(%ecx)</td>
<td>Assume array ‘a’ starts @ 0x1000000C</td>
</tr>
</tbody>
</table>

- Purple = Pointer init
- Blue = Read data from mem.
- Red = ALU op
- Green = Write data to mem.
### Translating HLL to Assembly

<table>
<thead>
<tr>
<th>C operator</th>
<th>Assembly</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>int dat[4], x;</td>
<td>movl $0x10000010,%ecx</td>
<td>Assume dat @ 0x10000010 &amp; x @ 0x10000020</td>
</tr>
<tr>
<td>...</td>
<td>movl (%ecx), %eax</td>
<td>• Purple = Pointer init</td>
</tr>
<tr>
<td>x = dat[0];</td>
<td>movl %eax, 16(%ecx)</td>
<td>• Blue = Read data from mem.</td>
</tr>
<tr>
<td>x += dat[1];</td>
<td>movl 16(%ecx), %eax</td>
<td>• Red = ALU op</td>
</tr>
<tr>
<td></td>
<td>addl 4(%ecx), %eax</td>
<td>• Green = Write data to mem.</td>
</tr>
<tr>
<td></td>
<td>movl %eax, 16(%ecx)</td>
<td></td>
</tr>
<tr>
<td>unsigned int y; short z;</td>
<td>movl $0x10000010,%ecx</td>
<td>Assume y @ 0x10000010 &amp; z @ 0x10000014</td>
</tr>
<tr>
<td>y = y / 4;</td>
<td>movl (%ecx), %eax</td>
<td></td>
</tr>
<tr>
<td>z = z &lt;&lt; 3;</td>
<td>shrl 2, %eax</td>
<td></td>
</tr>
<tr>
<td></td>
<td>movl %eax, (%ecx)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>movw 4(%ecx), %ax</td>
<td></td>
</tr>
<tr>
<td></td>
<td>salw 3, %ax</td>
<td></td>
</tr>
<tr>
<td></td>
<td>movw %ax, 4(%ecx)</td>
<td></td>
</tr>
</tbody>
</table>
How instruction sets differ

INSTRUCTION SET ARCHITECTURE
Instruction Set Architecture (ISA)

• Defines the software interface of the processor and memory system
• Instruction set is the vocabulary the HW can understand and the SW is composed with
• 2 approaches
  – CISC = Complex instruction set computer
    • Large, rich vocabulary
    • More work per instruction but slower HW
  – RISC = Reduced instruction set computer
    • Small, basic, but sufficient vocabulary
    • Less work per instruction but faster HW
Components of an ISA

• Data and Address Size
  – 8-, 16-, 32-, 64-bit

• Which instructions does the processor support
  – SUBtract instruc. vs. NEGate + ADD instruc.

• Registers accessible to the instructions
  – How many and expected usage

• Addressing Modes
  – How instructions can specify location of data operands

• Length and format of instructions
  – How is the operation and operands represented with 1’s and 0’s
General Instruction Format Issues

• Different instruction sets specify these differently
  – 3 operand instruction set (ARM, PPC)
    • Similar to example on previous page
    • Format: ADD DST, SRC1, SRC2  (DST = SRC1 + SRC2)
  – 2 operand instructions (Intel)
    • Second operand doubles as source and destination
    • Format: ADD SRC1, S2/D  (S2/D = SRC1 + S2/D)
  – 1 operand instructions (Old Intel FP, Low-End Embedded)
    • Implicit operand to every instruction usually known as the Accumulator (or ACC) register
    • Format: ADD SRC1  (ACC = ACC + SRC1)
General Instruction Format Issues

- Consider the pros and cons of each format when performing the set of operations
  - \( F = X + Y - Z \)
  - \( G = A + B \)
- Simple embedded computers often use single operand format
  - Smaller data size (8-bit or 16-bit machines) means limited instruc. size
- Modern, high performance processors use 2- and 3-operand formats

<table>
<thead>
<tr>
<th>Single-Operand</th>
<th>Two-Operand</th>
<th>Three-Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD X</td>
<td>MOVE F,X</td>
<td>ADD F,X,Y</td>
</tr>
<tr>
<td>ADD Y</td>
<td>ADD F,Y</td>
<td>SUB F,F,Z</td>
</tr>
<tr>
<td>SUB Z</td>
<td>SUB F,Z</td>
<td>ADD G,A,B</td>
</tr>
<tr>
<td>STORE F</td>
<td>MOVE G,A</td>
<td></td>
</tr>
<tr>
<td>LOAD A</td>
<td>ADD G,B</td>
<td></td>
</tr>
<tr>
<td>ADD B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STORE G</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(+) Smaller size to encode each instruction  
(-) Higher instruction count to load and store ACC value  
Compromise of two extremes  
(+ ) More natural program style  
(+ ) Smaller instruction count  
(- ) Larger size to encode each instruction
Instruction Format

• Load/Store architecture
  – Load (read) data values from memory into a register
  – Perform operations on registers
  – Store (write) data values back to memory
  – Different load/store instructions for different operand sizes (i.e. byte, half, word)