CSCE 312: Computer Organization

David Kebo Hougninou

Machine Language
What is Machine Language?

A machine language is an agreed-upon formalism, designed to code low-level programs as series of machine instructions.

Machine instructions can:
1. Perform arithmetic and logic operations in the processor
2. Load values from memory
3. Store values to memory
4. Move values from one register to another
5. Test Boolean conditions etc.

A compiler generates machine code based on:
1. Rules of the programming language
2. The instruction set of the target machine.

Why knowing machine language?

1. A must if you will write compilers or operating Systems for a living
2. Complete control over a system’s resources: Performance!
3. Direct access to hardware, critical for security
4. Understanding processor and memory function
5. Transparent Execution (WYSIWYG). Easy to Debug

“machine language makes it possible to manipulate hardware directly, address critical issues concerning performance and also provide access to special instructions for processors. Uses of machine language include coding device drivers, real-time systems, low-level embedded systems, boot codes, reverse engineering and more” -Techopedia
Memory

Memory is a collection of hardware devices that store data and instructions in a computer.

A programmer views memory as a continuous array of cells of some fixed width, also called words or locations, each having a unique address.

![Von Neumann Architecture](image1)

- Shared address space for data and code

![Harvard Architecture](image2)

- Separate address space for data and code

Processor

The processor core includes registers, an Arithmetic Logic Unit and controllers.

The processor performs:
1. Arithmetic and logic operations
2. Memory access operations
3. Branching operations

The operands of these operations are binary values.
The results of CPU operations are stored in registers or main memory.
The register is a high-speed local memory and is the fundamental storage area in the processor.

The features of the register are:
• Proximity to the processor
• Manipulate data and instructions quickly.
• Reduce the use of memory access commands to speed up program execution

The limitations of the registers are:
• Size in the order of bits
• Finite number of registers

Most registers are general purpose and can store any type of information:
- data – e.g. timer value, constants
- address – e.g. ASCII table, stack

Some are reserved for specific purpose:
- program counter (r15 in ARM)
- program status register (CPSR in ARM)
Memory size vs. memory access

The larger the memory, the longer its address and therefore the wider (and slower) the decoder circuit.

2 to 4 decoder for a 4 Block Memory

4 to 16 decoder for a 16 Block Memory

Compilation

machine language

while (n < 100) {
    sum += arr[i];
    n++
}
How to read Mnemonics

Instructions:

<table>
<thead>
<tr>
<th></th>
<th>1011</th>
<th>000011</th>
<th>000010</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>R3</td>
<td>R2</td>
<td></td>
</tr>
</tbody>
</table>

Interpretation 1:
• The symbolic form add R₃ R₂ doesn’t really exist
• It is just a convenient mnemonic that can be used to present machine language instructions to humans.

Interpretation 2:
• Allow humans to write symbolic machine language instructions, using assembly language
• Use an assembler program to translate the symbolic code into binary form.
Instruction classes

Instructions can be broadly separated into three basic classes:
1. Data Movement
   - Memory load/store
   - Register Transfers
2. Data Operation
   - Arithmetic
   - Logical
   - Comparison and test
3. Flow Control
   - Branch
   - Conditional execution

Terminologies

**Instruction set architecture:** The parts of a processor design that one needs to understand or write assembly/machine code.
Intel: x86, IA32, Itanium, x86-64
ARM: Used in almost all mobile phones

**Microarchitecture:** Implementation of the architecture.
e.g.: cache sizes and core frequency.

**Machine Code:** The byte-level programs that a processor executes

**Assembly Code:** A text representation of machine code
Assembly/Machine Code View

Programmer-Visible State
- **PC**: Program counter
- Address of next instruction
- **Register file**
- Heavily used program data
- **Condition codes**
  - Store status information about most recent arithmetic or logical operation.
  - Used for conditional branching.

Memory
- Byte addressable array
- Code and user data
- Stack to support procedures

From C to Object Code

Code in files `p1.c` `p2.c`

Compile with command: `gcc -Og p1.c p2.c -o p`

Use basic optimizations (`-Og`) [New to recent versions of GCC]
Machine Instruction Example

*C Code*  
Store value \( t \) where designated by \( \text{dest} \)

*Assembly*  
Move 8-byte value to memory  
Operands:  
\( t: \) Register \( \%rax \)  
\( \text{dest}: \) Register \( \%rbx \)  
\( \*\text{dest}: \) Memory \( M[\%rbx] \)

*Object Code*  
3-byte instruction  
Stored at address 0x40059e

Compiling Into Assembly (text)

*C Code (sum.c)*

```c
long plus(long x, long y);
void sumstore(long x, long y, long *dest)
{
    long t = plus(x, y);
    *dest = t;
}
```

*Generated x86-64 Assembly*

```assembly
sumstore:
    pushq  %rbx
    movq  %rdx, %rbx
    call  plus
    movq  %rax, (%rbx)
popq  %rbx
ret
```

`gcc -Og -S sum.c` the *Produces file sum.s*
Assembly to Object Code (Binary)

Code for sumstore

0x0400595:
0x53
0x48
0x89
0xd3
0xe8
0xf2
0xff
0xff
0xff
0x48
0x89
0x03
0x5b
0xc3

Assembler

- Translates .s into .o
- Binary encoding of each instruction

Linker

- Resolves references between files
- Combines with static run-time libraries
e.g., code for malloc, printf
- Linking occurs when program begins execution

Disassembling Object Code to Assemble

0000000000400595 <sumstore>:
400595: 53  push %rbx
400596: 48 89 d3  mov %rdx,%rbx
400599: e8 f2 ff ff ff  callq 400590 <plus>
40059e: 48 89 03  mov %rax,(%rbx)
4005a1: 5b  pop %rbx
4005a2: c3  retq

Disassembler

objdump -d sum

- Useful tool for examining object code
- Analyzes bit pattern of series of instructions
- Produces approximate rendition of assembly code
- Can be run on either a.out (complete executable) or .o file
IA 32 registers

- An IA32 CPU has 08 registers of 32-bit each.
- These registers store integer data as well as pointers.
- The register names all begin with %e.
- The first six registers are general-purpose registers.
- The last two registers (%ebp and %esp) contain pointers.
- The low-order 2 bytes of the first four registers can be independently read or written by the byte operation instructions.

IA32 Registers

| %eax | %ax | %ah | %al |
| %ecx | %cx | %ch | %cl |
| %edx | %dx | %dh | %dl |
| %ebx | %bx | %bh | %bl |
| %esi | %si |
| %edi | %di |
| %esp | %sp |
| %ebp | %bp |

16-bit virtual registers (backwards compatibility)

Origin (mostly obsolete)

- accumulate
- counter
- data
- base
- source
- index
- destination
- index
- stack
- pointer
- base
- pointer
### x86-64 Integer Registers

<table>
<thead>
<tr>
<th>64</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>%eax</td>
</tr>
<tr>
<td>%rbx</td>
<td>%ebx</td>
</tr>
<tr>
<td>%rcx</td>
<td>%ecx</td>
</tr>
<tr>
<td>%rdx</td>
<td>%edx</td>
</tr>
<tr>
<td>%rsi</td>
<td>%esi</td>
</tr>
<tr>
<td>%rdi</td>
<td>%edi</td>
</tr>
<tr>
<td>%r8</td>
<td>%r8d</td>
</tr>
<tr>
<td>%r9</td>
<td>%r9d</td>
</tr>
<tr>
<td>%r10</td>
<td>%r10d</td>
</tr>
<tr>
<td>%r11</td>
<td>%r11d</td>
</tr>
<tr>
<td>%r12</td>
<td>%r12d</td>
</tr>
<tr>
<td>%r13</td>
<td>%r13d</td>
</tr>
<tr>
<td>%r14</td>
<td>%r14d</td>
</tr>
<tr>
<td>%r15</td>
<td>%r15d</td>
</tr>
</tbody>
</table>

Can reference low-order 4 bytes (also low-order 1 & 2 bytes)

### x86-64 Integer Registers (another view)

- %rax: Return value
- %rbx: Callee saved
- %rcx: 4th argument
- %rdx: 3rd argument
- %rsi: 2nd argument
- %rdi: 1st argument
- %r8: 15th argument
- %r9: 16th argument
- %r10: 17th argument
- %r11: 18th argument
- %r12: 19th argument
- %r13: Unused for C
- %r14: Callee saved
- %r15: Callee saved
Instruction classes

Instructions can be broadly separated into three basic classes:

1. Data Movement
   - Memory load/store
   - Register Transfers

2. Data Operation
   - Arithmetic
   - Logical
   - Comparison and test

3. Flow Control
   - Branch
   - Conditional execution

Data movement

An x86 CPU can address up to $2^{32}$ Bytes of memory.

The data movement instructions are:

- mov
- push
- pop
Address Computation

Here are some practical examples:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8(%rdx)</td>
<td>0x000 + 0x8</td>
<td>0x0008</td>
</tr>
<tr>
<td>(%rdx,%rcx)</td>
<td>0x000 + 0x100</td>
<td>0x0100</td>
</tr>
<tr>
<td>(%rdx,%rcx,4)</td>
<td>0x000 + (4*0x100)</td>
<td>0x0400</td>
</tr>
<tr>
<td>0x80(,%rdx,2)</td>
<td>(2*0x000) + 0x80</td>
<td>0x1e080</td>
</tr>
</tbody>
</table>

The mov instruction

The `mov` instruction copies the data referred in the first operand to the location referred in the second operand.

What types of mov operations can you make?
- register-to-register: yes 😊
- memory-to-register: yes 😊
- memory-to-memory: no 😞

Syntax:
```
mov <reg>, <reg>
mov <reg>, <mem>
mov <mem>, <reg>
mov <imm>, <reg>
mov <imm>, <mem>
```

Examples:
```
mov %ebx, %eax /*copy the value in EBX into EAX*/
movb $5, var(,1) /*store the value 5 into the byte at location var*/
```
mov instruction: example

The mov instruction moves a source to destination

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mov (%ebx), %eax</code></td>
<td>Load 4 bytes from the memory address in EBX into EAX.</td>
</tr>
<tr>
<td><code>mov %ebx, var(,1)</code></td>
<td>Move the contents of EBX into the 4 bytes at memory address var. (var is a 32-bit constant)</td>
</tr>
<tr>
<td><code>mov -4(%esi), %eax</code></td>
<td>Move 4 bytes at memory address ESI + (-4) into EAX.</td>
</tr>
<tr>
<td><code>mov %cl, (%esi,%eax,1)</code></td>
<td>Move the contents of CL into the byte at address ESI+EAX.</td>
</tr>
<tr>
<td><code>mov (%esi,%ebx,4), %edx</code></td>
<td>Move the 4 bytes of data at memory address ESI+4*EBX into EDX.</td>
</tr>
</tbody>
</table>

The push instruction

The push instruction places its operand on the top of the hardware stack in memory.

push first decrements the stack pointer (ESP) by 4, then places its operand into the contents of the 32-bit location at address (%esp).

push decrements the ESP because the x86 stack grows down.

Note: A stack grows from high addresses to lower addresses.

<table>
<thead>
<tr>
<th>Syntax:</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>push &lt;reg32&gt;</code></td>
<td>Push a register on the stack</td>
</tr>
<tr>
<td><code>push &lt;mem&gt;</code></td>
<td>Push a memory address on the stack</td>
</tr>
<tr>
<td><code>push &lt;con32&gt;</code></td>
<td>Push a constant on the stack</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Examples:</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>push %eax /* push eax on the stack */</code></td>
<td>Push the value of the eax register onto the stack</td>
</tr>
<tr>
<td><code>push var(,1) /* push the 4 bytes at address var onto the stack */</code></td>
<td>Push the 4 bytes at the memory address var onto the stack</td>
</tr>
</tbody>
</table>
The pop instruction

The **pop** instruction removes the 4-byte data element from the top of the hardware stack into the specified operand (register or memory location).

It first moves the 4 bytes located at memory location (%esp) into the specified register or memory location, and then increments ESP by 4.

<table>
<thead>
<tr>
<th>Syntax:</th>
<th>pop &lt;reg32&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>pop &lt;mem&gt;</td>
</tr>
</tbody>
</table>

| Examples:     | pop %edi /*pop the top element of the stack into EDI*/ |
|              | pop (%ebx) /*pop the top element of the stack into memory at the four bytes starting at location EBX.*/ |

Data movement

Moving Data

```plaintext
movq Source, Dest:
```

Operand Types

**Immediate**: Constant integer data

- Example: $0x400, $-533
- Like C constant, but prefixed with ‘$’

**Register**: One of 16 integer registers

- Example: %rax, %r13
- But %rsp reserved for special use

**Memory**: 8 consecutive bytes of memory at address given by register

- Example: (%rax)
- Various other “address modes”