Assembly Language Programming
High-level vs. Assembly language

Consider the following statements

1. \( a = x + y - z \)
2. if \( x > y \)
   then \( x := x + y \)
   else \( x := x - y \)

HLL (High Level Language) programs are machine independent. They are easy to learn, easy to use, and convenient for managing complex tasks.

Assembly language programs are machine specific. It is the language that the processor “directly” understands.
Memory can be viewed as a bookshelf

View registers as spaces on your table
Understanding Assembly Language

Let us begin with data representation. How to represent

- Signed integers
- Fractions
- Alphanumeric characters
- Floating point numbers
- Pictures?

Memory

<table>
<thead>
<tr>
<th>01001011</th>
</tr>
</thead>
<tbody>
<tr>
<td>11011010</td>
</tr>
<tr>
<td>10011000</td>
</tr>
</tbody>
</table>

Can you read the contents of these memory cells?
Visualizing instruction execution

(The main concept is register-transfer operation.)

A register is a fast storage within the CPU

\[
a = x + y - z
\]

- load x into r1
- load y into r2
- load z into r0
- \( r3 \leftarrow r1 + r2 \)
- \( r0 \leftarrow r3 - r0 \)
- store r0 into a
Assembly language instructions for a hypothetical machine (not MIPS)

Load x, r1
Load y, r2
Load z, r0
Add r3, r1, r2
Sub r0, r3, r0
Store r0, a

Each processor has a different set of registers, and different assembly language instructions. The assembly language instructions of Intel Pentium and MIPS are completely different.

Motorola 68000 has 16 registers r0-r15
MIPS has 32 registers r0-r31
Pentium has 8 general purpose & 6 segment registers.
Binary or Machine Language program

Both program and data are represented using only 0’s and 1’s inside a computer. Here is a sample:

These are instruction formats. Each instruction has a specific format.
Can we distinguish program from data?

Program

Data

Both are bit strings.
Indistinguishable.

MEMORY

Normally, the programmer has to tell the machine (or use some convention) to specify the address of the first instruction. Incorrect specification will lead to errors, and the program is most likely to crash.
**Bits, bytes, words**

**Bit:** 0, 1

**Byte:** string of 8 bits. Each byte has an address.

**Word:** one or more bytes (usually 2 or 4 or 8).

```
0  01010000
1  11110000
2  00000000
3  11111111
4  00001111
5  10111011
6  00111100
7  00000111
```
Byte order in a word

Big Endian order [byte 0, byte 1, byte 2, byte 3]

Little Endian order [byte 3, byte 2, byte 1, byte 0]
Registers vs. memory

Data can be stored in registers or memory locations. Memory access is slower (takes approximately 50 ns) than register access (takes approximately 1 ns or less).

To increase the speed of computation it pays to keep the variables in registers as long as possible. However, due to technology limitations, the number of registers is quite limited (typically 8-64).

MIPS registers

MIPS has 32 registers r0-r31. The conventional use of these registers is as follows:
<table>
<thead>
<tr>
<th>register</th>
<th>assembly name</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>$zero</td>
<td>Always 0</td>
</tr>
<tr>
<td>r1</td>
<td>$at</td>
<td>Reserved for assembler</td>
</tr>
<tr>
<td>r2-r3</td>
<td>$v0-$v1</td>
<td>Stores results</td>
</tr>
<tr>
<td>r4-r7</td>
<td>$a0-$a3</td>
<td>Stores arguments</td>
</tr>
<tr>
<td>r8-r15</td>
<td>$t0-$t7</td>
<td>Temporaries, not saved</td>
</tr>
<tr>
<td>r16-r23</td>
<td>$s0-$s7</td>
<td>Contents saved for later use</td>
</tr>
<tr>
<td>r24-r25</td>
<td>$t8-$t9</td>
<td>More temporaries, not saved</td>
</tr>
<tr>
<td>r26-r27</td>
<td>$k0-$k1</td>
<td>Reserved by operating system</td>
</tr>
<tr>
<td>r28</td>
<td>$gp</td>
<td>Global pointer</td>
</tr>
<tr>
<td>r29</td>
<td>$sp</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>r30</td>
<td>$fp</td>
<td>Frame pointer</td>
</tr>
<tr>
<td>r31</td>
<td>$ra</td>
<td>Return address</td>
</tr>
</tbody>
</table>
Example assembly language programs

Example 1

\[ f = g + h - i \]

Assume that \( f, g, h, i \) are assigned to \( s0, s1, s2, s3 \)

\[
\begin{align*}
\text{add} & \quad t0, \quad s1, \quad s2 \quad \# \text{ register } t0 \text{ contains } g + h \\
\text{sub} & \quad s0, \quad t0, \quad s3 \quad \# f = g + h - i
\end{align*}
\]

Example 2

\[ g = h + A[8] \]

Assume that \( g, h \) are in \( s1, s2 \). \( A \) is an array of words, the elements are stored in consecutive locations of the memory. The base address is stored in \( s3 \).

\[
\begin{align*}
\text{l}w & \quad t0, \quad 32(s3) \quad \# t0 \text{ gets } A[8], \quad 32= 4 \times 8 \\
\text{add} & \quad s1, \quad s2, \quad t0 \quad \# g = h + A[8]
\end{align*}
\]
**Machine language representations**

Instruction “add” belongs to the **R-type format**.

```
<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shift amt</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>
```

```
add $s1, $s2, $t0 will be coded as
```

```
<table>
<thead>
<tr>
<th>0</th>
<th>18</th>
<th>8</th>
<th>17</th>
<th>0</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>
```

The function field is an extension of the opcode, and they together determine the operation.

Note that “sub” has a similar format.