Basic Program Structure

CSE 361S

Assembly Standard Form

• Assembly language is made up of statements and directives.
• Statements translate directly into machine instructions:
  – e.g., addl $01,%eax
• Standard form is as follows:
  – label opcode operands comment

Label

• As assembly process proceeds, assembler keeps track of the current “position” (i.e., address) where the current instruction is located. The ‘label’ is a symbolic name for this position. The programmer is “naming” the current position “label” by his/her action.
• This label then can be used elsewhere, say as the target of a branch.
  bne foo /* branch to foo if ZERO is not set */

Directives

• Directives are instructions not to the machine, but to the assembler itself.
• Sometimes they are called pseudo-ops, because they are not true op codes (i.e., machine instructions).
  .equiv MAX_CNT, 12
  makes the string MAX_CNT the textual equivalent of 12

Sections

.text
  tells the assembler we are in the text (code) section

other legal sections include:
  .rodata read only data
  .data initialized data
  .bss uninitialized data

Why sections, especially .bss?

• The name “bss” comes from the IBM 704 assembly language instruction “block storage start” in the late 1950s.
• Distinct sections are important, esp. in embedded computing, when the assembler or linker needs to do different things with different types of information, e.g., in volatile or non-volatile memory.
• The alternate form for sections is as follows:

```
.section <name>
```

where `<name>` is one of .text, .rodata, .data, or .bss

• Labels come in handy in data sections too, e.g.,

```
.section .data
i: .int 0
str: .string "this is an example string"
```

• Note: `.` is a special label that means “right here”, so expressions like `-4` means 4 addresses before the current position and `+.8` means 8 locations ahead of the current position.

---

C Program Structure

• Classically separated into two sets of files:

```c
#include <stdio>
#include "bar.h"

int var = 1; // global variables
int x;

int main(...) {
  int var = 2; // local variables
  // code goes here */
  x = var; // what is value of x after this stmt?
}
```

• Labels come in handy in data sections too, e.g.,

```
.section .data
i: .int 0
str: .string "this is an example string"
```

• Note: `.` is a special label that means “right here”, so expressions like `-4` means 4 addresses before the current position and `+.8` means 8 locations ahead of the current position.

---

Basic Operations

• Data movement

  - Assembly:

  ```assembly
  movl b, a /* move 4 bytes */
  movb b, a /* move 1 byte */
  ```

  - Source (b) can be {register, memory, immediate}
  - Destination (a) can be {r, m}

  ```c
  a = (char) b;
  ```

  - Type conversion is performed if a and b are not of the same type, which is not true in assembly language
Addition

• Assembly:
  addl b, a
  (r, m, i), (r, m)

• C:
  a = a + b;

a += b;

Bit-wise Logical Operations

AND –

• Assembly:
  andl b, a
  (r, m, i), (r, m)

• C:
  a = a & b;

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>a &amp; b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

• e.g.,
  \[ \begin{align*}
  10011010 & \quad & 01010111 \\
  \& 00010010 & \quad & 00000011 \\
  \end{align*} \]

• Common uses: masking bits, mod $2^n$ (e.g., mod 4 via 00000011)

Element-wise Logical Operations

• Not available in assembly language, only in C

  && is \hspace{1em} AND

  || is \hspace{1em} OR

  ! is \hspace{1em} NOT

• For each operand, all bits equal 0 makes the operand FALSE, any non-zero value in any bit makes the operand TRUE.

• Operator returns 0 for FALSE and 1 for TRUE

Evaluation Order

• C expressions are evaluated left-to-right, only what is needed to determine result

  e.g., x && 1/x won’t divide by zero