Today’s lecture

- Last lecture we started talking about control flow in MIPS (branches)
- Finish up control-flow (branches) in MIPS
  - if/then
  - loops
  - case/switch
- Array Indexing vs. Pointers
  - In particular pointer arithmetic
  - String representation

Slides adapted from Josep Torrellas, Craig Zilles, and Howard Huang

Translating an if-then statement

- We can use branch instructions to translate if-then statements into MIPS assembly code.

```assembly
v0 = a0;
if (v0 < 0)
v0 = -v0;
v1 = v0 + v0;
move $v0 $a0
bge $v0, $0, Label
sub $v0, 0, $v0
Label: add $v1, $v0, $v0
```

- Sometimes it’s easier to invert the original condition.
  - In this case, we changed “continue if v0 < 0” to “skip if v0 >= 0”.
  - This saves a few instructions in the resulting assembly code.
Translating an if-then-else statements

- If there is an else clause, it is the target of the conditional branch
  - And the then clause needs a jump over the else clause

```assembly
// increase the magnitude of v0 by one
if (v0 < 0)
    v0 --;
else
    v0 ++;
v1 = v0;
```

- Drawing the control-flow graph can help you out.

Control-flow graphs

- It can be useful to draw control-flow graphs when writing loops and conditionals in assembly:

```assembly
// Find the absolute value of *a0
v0 = *a0;
if (v0 < 0)
    v0 = -v0;
v1 = v0 + v0;
```

```assembly
// Sum the elements of a0
v0 = 0;
t0 = 0;
while (t0 < 5) {
    v0 = v0 + a0[t0];
t0++;
}
What does this code do?

label: sub $a0, $a0, 1
    bne $a0, $zero, label

Loops

for (i = 0; i < 4; i++) {
    // stuff
}

add $t0, $zero, $zero # i is initialized to 0, $t0 = 0

Loop: j Loop # goto Loop

addi $t0, $t0, 1 # i ++
slti $t1, $t0, 4 # $t1 = 1 if i < 4
bne $t1, $zero, Loop # go to Loop if i < 4
Many high-level languages support multi-way branches, e.g.

```c
switch (two_bits) {
    case 0: break;
    case 1: /* fall through */
    case 2: count ++; break;
    case 3: count += 2; break;
}
```

We could just translate the code to if, thens, and elses:

```c
if ((two_bits == 1) || (two_bits == 2)) {
    count ++;
} else if (two_bits == 3) {
    count += 2;
}
```

This isn't very efficient if there are many, many cases.

Alternatively, we can:
1. Create an array of jump targets
2. Load the entry indexed by the variable two_bits
3. Jump to that address using the jump register, or `jr`, instruction
Representing strings

- A C-style string is represented by an array of bytes.
  - Elements are one-byte ASCII codes for each character.
  - A 0 value marks the end of the array.

<table>
<thead>
<tr>
<th>32</th>
<th>space</th>
<th>48</th>
<th>@</th>
<th>80</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>33</td>
<td>!</td>
<td>49</td>
<td>1</td>
<td>81</td>
<td>q</td>
</tr>
<tr>
<td>34</td>
<td>&quot;</td>
<td>50</td>
<td>2</td>
<td>82</td>
<td>r</td>
</tr>
<tr>
<td>35</td>
<td>#</td>
<td>51</td>
<td>3</td>
<td>83</td>
<td>s</td>
</tr>
<tr>
<td>36</td>
<td>$</td>
<td>52</td>
<td>4</td>
<td>84</td>
<td>t</td>
</tr>
<tr>
<td>37</td>
<td>%</td>
<td>53</td>
<td>5</td>
<td>85</td>
<td>u</td>
</tr>
<tr>
<td>38</td>
<td>&amp;</td>
<td>54</td>
<td>6</td>
<td>86</td>
<td>v</td>
</tr>
<tr>
<td>39</td>
<td>'</td>
<td>55</td>
<td>7</td>
<td>87</td>
<td>w</td>
</tr>
<tr>
<td>40</td>
<td>(</td>
<td>56</td>
<td>8</td>
<td>88</td>
<td>x</td>
</tr>
<tr>
<td>41</td>
<td>)</td>
<td>57</td>
<td>9</td>
<td>89</td>
<td>y</td>
</tr>
<tr>
<td>42</td>
<td>*</td>
<td>58</td>
<td>:</td>
<td>90</td>
<td>z</td>
</tr>
<tr>
<td>43</td>
<td>+</td>
<td>59</td>
<td>;</td>
<td>91</td>
<td>[</td>
</tr>
<tr>
<td>44</td>
<td>,</td>
<td>60</td>
<td>&lt;=</td>
<td>92</td>
<td>\</td>
</tr>
<tr>
<td>45</td>
<td>-</td>
<td>61</td>
<td>=</td>
<td>93</td>
<td>]</td>
</tr>
<tr>
<td>46</td>
<td>.</td>
<td>62</td>
<td>&gt;</td>
<td>94</td>
<td>^</td>
</tr>
<tr>
<td>47</td>
<td>/</td>
<td>63</td>
<td>?</td>
<td>95</td>
<td>_</td>
</tr>
</tbody>
</table>

Null-terminated Strings

- For example, “Harry Potter” can be stored as a 13-byte array.

<table>
<thead>
<tr>
<th>72</th>
<th>97</th>
<th>114</th>
<th>114</th>
<th>121</th>
<th>32</th>
<th>80</th>
<th>111</th>
<th>116</th>
<th>116</th>
<th>101</th>
<th>114</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>a</td>
<td>r</td>
<td>r</td>
<td>y</td>
<td>P</td>
<td>o</td>
<td>t</td>
<td>t</td>
<td>e</td>
<td>r</td>
<td>\</td>
<td>0</td>
</tr>
</tbody>
</table>

- Since strings can vary in length, we put a 0, or null, at the end of the string.
  - This is called a null-terminated string

- Computing string length
  - We’ll look at two ways.
What does this C code do?

```c
int foo(char *s) {
    int L = 0;
    while (*s++) {
        ++L;
    }
    return L;
}
```

Array Indexing Implementation of strlen

```c
int strlen(char *string) {
    int len = 0;
    while (string[len] != 0) {
        len ++;
    }
    return len;
}
```
Pointers & Pointer Arithmetic

- Many programmers have a vague understanding of pointers
  — Looking at assembly code is useful for their comprehension.

```c
int strlen(char *string) {
    int len = 0;
    while (string[len] != 0) {
        len ++;
    }
    return len;
}
```

What is a Pointer?

- A pointer is an address.
- Two pointers that point to the same thing hold the same address.
- Dereferencing a pointer means loading from the pointer’s address.
- A pointer has a type; the type tells us what kind of load to do
  — Use load byte (lb) for char *
  — Use load half (lh) for short *
  — Use load word (lw) for int *
  — Use load single precision floating point (l.s) for float *
- Pointer arithmetic is often used with pointers to arrays
  — Incrementing a pointer (i.e., ++) makes it point to the next element
  — The amount added to the point depends on the type of pointer
    • pointer = pointer + sizeof(pointer’s type)
      ‣ 1 for char *, 4 for int *, 4 for float *, 8 for double *
What is really going on here...

```c
int strlen(char *string) {
    int len = 0;
    while (*string != 0) {
        string ++;
        len ++;
    }

    return len;
}
```

Pointers Summary

- Pointers are just addresses!!
  - “Pointees” are locations in memory
- Pointer arithmetic updates the address held by the pointer
  - “string ++” points to the next element in an array
  - Pointers are typed so address is incremented by sizeof(pointee)