CSE378 - Lecture 4

- Announcements
  - HW1 out

- Today:
  - Finish-up control-flow
    - if/then
    - loops
    - case/switch
  - Array Indexing vs. Pointers
    - In particular pointer arithmetic
    - String representation

Control-flow Example

- Let's write a program to count how many bits are set in a 32-bit word.

```c
int count = 0;
for (int i = 0 ; i < 32 ; i ++) {
   int bit = input & 1;
   if (bit != 0) {
      count ++;
   }
   input = input >> 1;
}
```

```
.text
main:

        li      $a0, 0x1234             ## input = 0x1234
        li      $t0, 0                  ## int count = 0;
        li      $t1, 0                  ## for (int i = 0

main_loop:
        bge     $t1, 32, main_exit      ## exit loop if i >= 32

        andi    $t2, $a0, 1             ## bit = input & 1
        beq     $t2, $0, main_skip      ## skip if bit == 0

        addi    $t0, $t0, 1             ## count ++

main_skip:
        srl     $a0, $a0, 1             ## input = input >> 1
        add     $t1, $t1, 1             ## i ++

        j       main_loop

main_exit:
        jr      $ra
```

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

```
// increase the magnitude of v0 by one
if (v0 < 0)
    v0 --;

else
    v0 ++;
v1 = v0;
```

```
        bge   $v0, $0, E
        sub   $v0, $v0, 1
        j     L

E:      add   $v0, $v0, 1
L:      move  $v1, $v0
```

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

Case/Switch Statement

- Many high-level languages support multi-way branches, e.g.

```
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:

```
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.

Case/Switch Statement

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

- 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>
<tr><td>32</td><td>space</td><td>48</td><td>0</td><td>64</td><td>@</td><td>80</td><td>P</td><td>96</td><td>`</td><td>112</td><td>p</td></tr>
<tr><td>33</td><td>!</td><td>49</td><td>1</td><td>65</td><td>A</td><td>81</td><td>Q</td><td>97</td><td>a</td><td>113</td><td>q</td></tr>
<tr><td>34</td><td>”</td><td>50</td><td>2</td><td>66</td><td>B</td><td>82</td><td>R</td><td>98</td><td>b</td><td>114</td><td>r</td></tr>
<tr><td>35</td><td>#</td><td>51</td><td>3</td><td>67</td><td>C</td><td>83</td><td>S</td><td>99</td><td>c</td><td>115</td><td>s</td></tr>
<tr><td>36</td><td>$</td><td>52</td><td>4</td><td>68</td><td>D</td><td>84</td><td>T</td><td>100</td><td>d</td><td>116</td><td>t</td></tr>
<tr><td>37</td><td>%</td><td>53</td><td>5</td><td>69</td><td>E</td><td>85</td><td>U</td><td>101</td><td>e</td><td>117</td><td>u</td></tr>
<tr><td>38</td><td>&</td><td>54</td><td>6</td><td>70</td><td>F</td><td>86</td><td>V</td><td>102</td><td>f</td><td>118</td><td>v</td></tr>
<tr><td>39</td><td>’</td><td>55</td><td>7</td><td>71</td><td>G</td><td>87</td><td>W</td><td>103</td><td>g</td><td>119</td><td>w</td></tr>
<tr><td>40</td><td>(</td><td>56</td><td>8</td><td>72</td><td>H</td><td>88</td><td>X</td><td>104</td><td>h</td><td>120</td><td>x</td></tr>
<tr><td>41</td><td>)</td><td>57</td><td>9</td><td>73</td><td>I</td><td>89</td><td>Y</td><td>105</td><td>I</td><td>121</td><td>y</td></tr>
<tr><td>42</td><td>*</td><td>58</td><td>:</td><td>74</td><td>J</td><td>90</td><td>Z</td><td>106</td><td>j</td><td>122</td><td>z</td></tr>
<tr><td>43</td><td>+</td><td>59</td><td>;</td><td>75</td><td>K</td><td>91</td><td>[</td><td>107</td><td>k</td><td>123</td><td>{</td></tr>
<tr><td>44</td><td>,</td><td>60</td><td><</td><td>76</td><td>L</td><td>92</td><td>\</td><td>108</td><td>l</td><td>124</td><td>|</td></tr>
<tr><td>45</td><td>-</td><td>61</td><td>=</td><td>77</td><td>M</td><td>93</td><td>]</td><td>109</td><td>m</td><td>125</td><td>}</td></tr>
<tr><td>46</td><td>.</td><td>62</td><td>></td><td>78</td><td>N</td><td>94</td><td>^</td><td>110</td><td>n</td><td>126</td><td>~</td></tr>
<tr><td>47</td><td>/</td><td>63</td><td>?</td><td>79</td><td>O</td><td>95</td><td>_</td><td>111</td><td>o</td><td>127</td><td>del</td></tr>
</table>

Null-terminated Strings

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

72	97	114	114	121	32	80	111	116	116	101	114	0
H	a	r	r	y		P	o	t	t	e	r	\0

- 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?

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

Array Indexing Implementation of strlen

```
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.
    - (But if you have an aggressive optimizing compiler, you may see the same assembly code for both versions!)

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

```
int strlen(char *string) {
    int len = 0;
    while (*string != 0) {
        string ++;
        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
- In C, 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…

```
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)