Control flow in high-level languages

- The instructions in a program usually execute one after another, but it’s often necessary to alter the normal control flow.
- Conditional statements execute only if some test expression is true.

```
// Find the absolute value of a0
v0 = v0 + v0; // This might not be executed
if (v0 < 0) { v0 = v0 + v0[16]; }  
```

- Loops cause some statements to be executed many times.

```
// Sum the elements of a five-element array a0
v0 = 0;
while (v0 < 5) {  
v0 = v0 + v0[10]; // These statements will be executed five times
}
```

MIPS control instructions

- MIPS’s control flow instructions

```
add $s0, $a0, $a1  
```

- Now we’ll talk about
  - MIPS’s pseudo branches
  - Jumps
  - Case/switch

Pseudo-branches

- The MIPS processor only supports two branch instructions, beq and bne, but to simplify your life the assembler provides the following other branches:

```
blt $t0, $t1, $t2  
```

- There are also immediate versions of these branches, where the second source is a constant instead of a register.

- Later this quarter we’ll see how supporting just beq and bne simplifies the processor design.
**Implementing pseudo-branches**

- Most pseudo-branches are implemented with a single branch-iffalse-than instruction (e.g., `blt $a0, $a1`), translated into the following:

```plaintext
   slti $s0, $a0, $a1; // Set = 1 if $a0 < $a1
   beq $s0, $0, Label; // Branch if $s0 = 0
```

- This supports immediate branches, which are also pseudo-instructions. For example, `ble $s0, $a0` is translated into two instructions:

```plaintext
   slti $s0, $a0, 5; // Set = 1 if $a0 < 5
   bne $s0, $0, Label; // Branch if $s0 != 0
```

- All of the pseudo-branches need a register to save the result of `slt`, even though it's not needed afterwards.
  - MIPS assemblers use register `$1`, or `$at`, for temporary storage. You should be careful in using `slt` in your own programs, as it may be overwritten by assembler-generated code.

---

**Translating an if-then statement**

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

```plaintext
   v0 = a0;
   move $v0, $a0
   v0 = v0;
   move $v0, $a0
   j v0
```

- Sometimes it's easier to invert the original condition.
  - In this case, we changed "continue if (v) < 0" to "skip if v = 0".
  - This saves a few instructions in the resulting assembly code.

```plaintext
   move $v0, $a0
   blt $v0, $0, Label1
   j v0
   Label0:
   j Label2
   Label1:
   move $a1, $v0
   Label2:
```

---

**What does this code do?**

```plaintext
    label:  
    sub $s0, $a0, 1  
    bne $s0, $zero, label  

    max ($j, $a3)
```

---

**Control-flow Example**

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

```plaintext
    # Program to count the number of set bits in a 32-bit word
    # Input: $a0
    # Output: $s0

    data:
    # Constants
    count:  
    label:  
    loop:  
    max:  
    next:

    # Initializing
    # Input = $a0
    lo = low($a0)
    hi = high($a0)
    label:  
    loop:  
    max:  
    next:

    # Loop body
    if (hi < 0)
    loop:  
    max:  
    next:

    # Main loop
    for (i = 0; i < 32; i++)
    loop:  
    max:  
    next:

    # Core loop
    if ($s0 < 0)
    loop:  
    max:  
    next:

    # Add the carry bit
    if ($s0 < 0)
    loop:  
    max:  
    next:

    # Exit
    jr $ra
```

---

**Translating an if-then-else statement**

- If there is an `else` clause, it is the target of the conditional branch.
  - The else clause needs a jump over the `if` clause.

```plaintext
    if ($v0 < 0)
    goto Loop
    # Increase the magnitude of $v0 by one
    move $v1, $v0
    add $v1, $v1, $v0
    # Add the carry bit
    if ($v0 < 0)
    goto Loop
    # Drawing the control-flow graph can help you out.
```

---

**Loops**

```plaintext
    # goto Loop
    Loop:
    j Loop
```

---

**What does this code do?**

```plaintext
    label:  
    sub $s0, $a0, 1  
    bne $s0, $zero, label  

    max ($j, $a3)
```
Case/Switch Statement

- 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 == 3)) {  //
    count ++;
} else if (two_bits == 3)
    count = 2;
```

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

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>Character</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>65</td>
</tr>
<tr>
<td>B</td>
<td>66</td>
</tr>
<tr>
<td>C</td>
<td>67</td>
</tr>
<tr>
<td>D</td>
<td>68</td>
</tr>
<tr>
<td>E</td>
<td>69</td>
</tr>
<tr>
<td>F</td>
<td>70</td>
</tr>
<tr>
<td>G</td>
<td>71</td>
</tr>
<tr>
<td>H</td>
<td>72</td>
</tr>
<tr>
<td>I</td>
<td>73</td>
</tr>
<tr>
<td>J</td>
<td>74</td>
</tr>
<tr>
<td>K</td>
<td>75</td>
</tr>
<tr>
<td>L</td>
<td>76</td>
</tr>
<tr>
<td>M</td>
<td>77</td>
</tr>
<tr>
<td>N</td>
<td>78</td>
</tr>
<tr>
<td>O</td>
<td>79</td>
</tr>
<tr>
<td>P</td>
<td>80</td>
</tr>
<tr>
<td>Q</td>
<td>81</td>
</tr>
<tr>
<td>R</td>
<td>82</td>
</tr>
<tr>
<td>S</td>
<td>83</td>
</tr>
<tr>
<td>T</td>
<td>84</td>
</tr>
<tr>
<td>U</td>
<td>85</td>
</tr>
<tr>
<td>V</td>
<td>86</td>
</tr>
<tr>
<td>W</td>
<td>87</td>
</tr>
<tr>
<td>X</td>
<td>88</td>
</tr>
<tr>
<td>Y</td>
<td>89</td>
</tr>
<tr>
<td>Z</td>
<td>90</td>
</tr>
</tbody>
</table>

Null-terminated Strings

- For example, “Harry Potter” can be stored as a 15-byte array.

<table>
<thead>
<tr>
<th>Character</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>72</td>
</tr>
<tr>
<td>a</td>
<td>97</td>
</tr>
<tr>
<td>r</td>
<td>114</td>
</tr>
<tr>
<td>y</td>
<td>114</td>
</tr>
<tr>
<td>P</td>
<td>121</td>
</tr>
<tr>
<td>o</td>
<td>120</td>
</tr>
<tr>
<td>r</td>
<td>116</td>
</tr>
<tr>
<td>y</td>
<td>116</td>
</tr>
<tr>
<td>t</td>
<td>101</td>
</tr>
<tr>
<td>e</td>
<td>114</td>
</tr>
<tr>
<td>r</td>
<td>116</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++ != 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!)

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

```c
int strlen(char *string) {
    int len = 0;
    while (string[len] != 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 (b) for char *
  - Use load half (h) for short *
  - Use load word (w) for int *
  - Use load single precision floating point (f) 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)