# 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



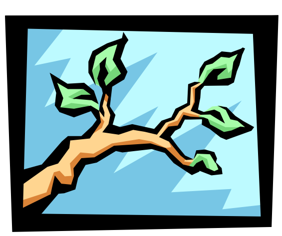
# Translating an if-then statement

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

```
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

```
// increase the magnitude of v0 by one
if (v0 < 0)                         bge  $v0, $0, E
    v0 --;                          sub  $v0, $v0, 1
                                    j    L
else
    v0 ++;                      E:  add  $v0, $v0, 1
v1 = v0;                        L:  move $v1, $v0
```

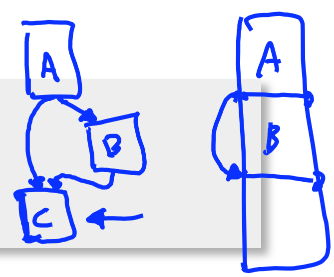
  - Drawing the control-flow graph can help you out.



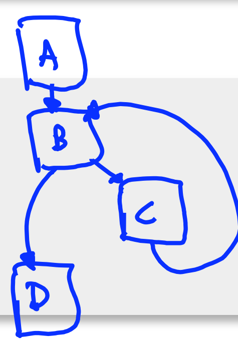
## Control-flow graphs

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

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




```
// 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

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

---

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

```
        add    $t0, $zero, $zero   # i is initialized to 0, $t0 = 0
Loop:   // stuff
        addi   $t0, $t0, 1         # i ++
        slti   $t1, $t0, 4         # $t1 = 1 if i < 4
        bne    $t1, $zero, Loop    # go to Loop if i < 4
```

## 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.

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



# 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.

```
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
- 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)