Announcements

- Homework 1 will be posted shortly. Due in a week or so (see the assignment for the exact date)

Today:

- Finish up memory
- Control-flow (branches) in MIPS
  - if/then
  - loops
  - case/switch
- (maybe) Start: Array Indexing vs. Pointers
  - In particular pointer arithmetic
  - String representation
Quick Review

- Registers x Memory

\[ \text{lw } \$t0, 4(\$a0) \]

\$a0 is simply another name for register 4
\$t0 is another name for register ____ (green sheet)

What does \$a0 contain?

What will \$t0 contain after the instruction is executed? (address)

Upper/lower bytes in a register (lui example)
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.

```c
// Find the absolute value of a0
v0 = a0;
if (v0 < 0)
    v0 = -v0; // This might not be executed
v1 = v0 + v0;
```

- **Loops** cause some statements to be executed many times.

```c
// Sum the elements of a five-element array a0
v0 = 0;
t0 = 0;
while (t0 < 5) {
    v0 = v0 + a0[t0]; // These statements will
    t0++; // be executed five times
}
```
// Find the absolute value of a0
v0 = a0;
if (v0 < 0)
    v0 = -v0;
vl = v0 + v0;

// Sum the elements of a0
v0 = 0;
t0 = 0;
while (t0 < 5) {
    v0 = v0 + a0[t0];
    t0++;
}
MIPS control instructions

- MIPS’s control-flow instructions
  
  \[
  \begin{align*}
  j & \quad // \text{for unconditional jumps} \\
  \text{bne and beq} & \quad // \text{for conditional branches} \\
  \text{slt and slti} & \quad // \text{set if less than (w/o and w an immediate)}
  \end{align*}
  \]

- Now we’ll talk about
  - MIPS’s pseudo branches
  - if/else
  - case/switch
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, L1` // Branch if $t0 < $t1
- `ble $t0, $t1, L2` // Branch if $t0 <= $t1
- `bgt $t0, $t1, L3` // Branch if $t0 > $t1
- `bge $t0, $t1, L4` // Branch if $t0 >= $t1

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 using slt. For example, a branch-if-less-than instruction `blt $a0, $a1, Label` is translated into the following.

  ```
  slt $at, $a0, $a1    // $at = 1 if $a0 < $a1
  bne $at, $0, Label   // Branch if $at != 0
  ```

- This supports immediate branches, which are also pseudo-instructions. For example, `blti $a0, 5, Label` is translated into two instructions.

  ```
  slti $at, $a0, 5      // $at = 1 if $a0 < 5
  bne $at, $0, Label    // Branch if $a0 < 5
  ```

- 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 $at 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.

\[
\begin{align*}
v_0 &= a_0; \\
\text{if } (v_0 < 0) \\
&\quad v_0 = -v_0; \\
v_1 &= v_0 + v_0;
\end{align*}
\]

\[
\begin{align*}
\text{move } &v0 \text{ } a0 \\
\text{bge } &v0, 0, \text{Label} \\
\text{sub } &v0, 0, v0 \\
\text{Label: } &\text{add } v1, v0, v0
\end{align*}
\]

- Sometimes it’s easier to invert the original condition.
  - In this case, we changed “continue if \( v_0 < 0 \)” to “skip if \( v_0 \geq 0 \)”.
  - This saves a few instructions in the resulting assembly code.
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
addi $t0, $t0, 1       # i ++
slti $t1, $t0, 4        # $t1 = 1 if i < 4
bne $t1, $zero, Loop   # go to Loop if i < 4
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

```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.
Case/Switch Statement

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

```plaintext
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:

```plaintext
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

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

```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 != 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 (ls) 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)