October 1

- Programming Questions?
- All of Chapter 3 and Appendix A are relevant and helpful to your programming assignments.
- More programming issues...

void clear1(int array[], int size) {
    for(int i=0; i<size; i++)
        array[i] = 0;
}

void clear2(int *array, int size) {
    for(int *p = &array[0]; p < &array[size]; p++)
        *p = 0;
}

void clear3(int *array, int size) {
    int *arrayend = array + size;
    while(array < arrayend) *array++ = 0;
}
clear1

void clear1(int array[], int size) {
    for(int i=0; i<size; i++)
        array[i] = 0;
}

move $t0,$zero # i = 0
for1:
    slt $t1, $t0, $a1 # break if i >= size
    beq $t1, $zero, for2 # break if i >= size
    add $t1, $t0, $t0 # t1 = i*2
    add $t1, $t1, $t1 # t1 = i*4
    add $t1, $a0, $t1 # t1 = &array[i]
    sw $zero, 0($t1) # *t1 = 0
    addi $t0, $t0, 1 # i++
    j for1
for2:

10/1/2001 Comp 120 Fall 2001 3

clear2

void clear2(int *array, int size) {
    for(int *p = &array[0]; p < &array[size]; p++)
        *p = 0;
}

move $t0, $a0 # p = array
for1:
    add $t1, $a1, $a1 # t1 = 2*size
    add $t1, $t1, $t1 # t1 = 4*size
    add $t1, $a0, $t1 # t1 = &array[size]
    slt $t2, $t0, $t1
    beq $t2, $zero, for2 # break if p >= t1
    sw $zero, 0($t0) # *p = 0;
    addi $t0, $t0, 4 # p++
    j for1
for2:
clear2 (slightly smarter)

void clear2(int *array, int size) {
    for(int *p = &array[0]; p < &array[size]; p++)
        *p = 0;
}

move $t0, $a0 # p = array
add $t1, $a1, $a1 # t1 = 2*size
add $t1, $t1, $t1 # t1 = 4*size
add $t1, $a0, $t1 # t1 = &array[size]
for1:
    slt $t2, $t0, $t1
    beq $t2, $zero, for2 # break if p >= t1
    sw $zero, 0($t0) # *p = 0;
    addi $t0, $t0, 4 # p++
    j for1
for2:


clear3

void clear3(int *array, int size) {
    int *arrayend = array + size;
    while(array < arrayend) *array++ = 0;
}

add $t1, $a1, $a1 # t1 = 2*size
add $t1, $t1, $t1 # t1 = 4*size
add $t1, $a0, $t1 # t1 = &array[size]
for1:
    slt $t2, $a0, $t1
    beq $t2, $zero, for2 # break if array >= t1
    sw $zero, 0($a0) # *array = 0;
    addi $a0, $a0, 4 # array++
    j for1
for2:
Pointer summary

- In the “C” world and in the “machine” world:
  - a pointer is just the address of an object in memory
  - size of pointer is fixed regardless of size of object
  - to get to the next object increment by the objects size in bytes
  - to get the the \textit{i}^{th} object add \textit{i} \times \text{sizeof}(object)

- More details:
  - int \texttt{R[5]} \Rightarrow \texttt{R} is int* constant address of 20 bytes
  - \texttt{R[i]} \Rightarrow \texttt{*(R+i)}
  - int \texttt{*p = \&R[3]} \Rightarrow \texttt{p = (R+3)} (p points 12 bytes after R)

Big Constants

The MIPS architecture only allows immediate constants to be 16 bits

So how do we get bigger constants?

\texttt{lui} sets the upper 16 bits from the 16 bit immediate field
\texttt{ori} will “or” into the lower 16 bits from the immediate field

How to break your BIG number into the required two 16 bit chunks?

\texttt{hi = BIG / 64k} \quad (\text{e.g. } 4,000,000 / 64k = 61)
\texttt{lo = BIG \% 64k} \quad (\text{e.g. } 4,000,000 \% 64k = 2304)
\texttt{lui} \quad \texttt{\$t0, hi}
\texttt{ori} \quad \texttt{\$t0, \$t0, lo}
Pointer Size vs. Addressable Space

- Pointers ARE addresses
- Number of unique addresses for N bits is 2^N

- With addresses that are 32 bits long you can address 4G bytes
- With addresses that are 13 bits long you can address 8k bytes
  - that’s 2k words

Endians?

Consider the following code

```
foo: .word 0           # foo is a 32 bit int
    li $t0, 1       # t0 = 1
    la $t1, foo     # t1 = &foo
    sb $t0, 0($t1)  # stores a byte at foo
```

What is the value of the WORD at foo? In other words what is the value of register t2 after:

```
    lw $t2, 0($t1)    # what is in t2?
```

Little Endian ≠ t2 == 0x00000001 == 1
Big Endian ≠ t2 == 0x01000000 == 16M
Consider the following code

```
foo: .word 0         # foo is a 32 bit int
li  $t0, 1          # t0 = 1
la  $t1, foo        # t1 = &foo
sb  $t0, 1($t1)     # stores a byte at foo+1
```

What is the value of the WORD at foo? In other words what is the value of register t2 after:

```
lw  $t2, 0($t1)     # what is in t2?
```

**Little Endian** $t2 == 0x00000100 == 256

**Big Endian** $t2 == 0x00010000 == 64k
Endians?

Consider the following code:

```assembly
foo: .word 0  # foo is a 32 bit int
    li $t0, 1  # t0 = 1
    la $t1, foo  # t1 = &foo
    sb $t0, 3($t1) # stores a byte at foo+3
```

What is the value of the WORD at foo? In other words, what is the value of register $t2 after:

```assembly
lw $t2, 0($t1)  # what is in t2?
```

Little Endian ➔ $t2 == 0x01000000 == 16M
Big Endian ➔ $t2 == 0x00000001 == 1

---

Endians?

Consider the following code:

```assembly
foo: .ascii "Gary"  # foo takes 4 bytes, 32 bits
    la $t1, foo  # t1 = &foo
```

What is the value of the WORD at foo? In other words, what is the value of register $t2 after:

```assembly
lw $t2, 0($t1)  # what is in t2?
```

Little Endian ➔ $t2 == 0x79726147
Big Endian ➔ $t2 == 0x47617279

On BOTH machines:

```assembly
lb $t3, 0($t1)  # t3 == ‘G’
```
ASCII Chart

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NUL</td>
<td>SOH</td>
<td>STX</td>
<td>ETX</td>
<td>EOT</td>
<td>ENQ</td>
<td>ACK</td>
<td>BEL</td>
<td>BS</td>
<td>HT</td>
<td>LF</td>
<td>VT</td>
<td>FF</td>
<td>CR</td>
<td>SO</td>
</tr>
<tr>
<td>1</td>
<td>DLE</td>
<td>DC1</td>
<td>DC2</td>
<td>DC3</td>
<td>DC4</td>
<td>NAK</td>
<td>SYN</td>
<td>ETB</td>
<td>CAN</td>
<td>EM</td>
<td>SUB</td>
<td>ESC</td>
<td>FS</td>
<td>GS</td>
<td>RS</td>
</tr>
<tr>
<td>2</td>
<td>SP</td>
<td>!</td>
<td>&quot;</td>
<td>#</td>
<td>$</td>
<td>%</td>
<td>&amp;</td>
<td>'</td>
<td>(</td>
<td>)</td>
<td>*</td>
<td>+</td>
<td>,</td>
<td>-</td>
<td>.</td>
</tr>
<tr>
<td>3</td>
<td>/</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>:</td>
<td>;</td>
<td>&lt;</td>
<td>=</td>
</tr>
<tr>
<td>4</td>
<td>&gt;</td>
<td>?</td>
<td>@</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
</tr>
<tr>
<td>5</td>
<td>M</td>
<td>N</td>
<td>O</td>
<td>P</td>
<td>Q</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>U</td>
<td>V</td>
<td>W</td>
<td>X</td>
<td>Y</td>
<td>Z</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>{</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>^</td>
<td>_</td>
<td>`</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td>f</td>
<td>g</td>
<td>h</td>
<td>i</td>
<td>j</td>
<td>k</td>
<td>l</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>m</td>
<td>n</td>
<td>o</td>
<td>p</td>
<td>q</td>
<td>r</td>
<td>s</td>
<td>t</td>
<td>u</td>
<td>v</td>
<td>w</td>
<td>x</td>
<td>y</td>
<td>z</td>
<td></td>
</tr>
</tbody>
</table>

Hex

- Numbers in hex are commonly preceded by 0x
  - 0x1 == 1, 0x10 == 16, etc.
- Hex is cool because each digit corresponds to 4 bits
  
  - 0x0 == 0000b, 0x1 == 0001b, 0x2 == 0010b, 0x3 == 0011b
  - 0x4 == 0100b, 0x5 == 0101b, 0x6 == 0110b, 0x7 == 0111b
  - 0x8 == 1000b, 0x9 == 1001b, 0xA == 1010b, 0xB == 1011b
  - 0xC == 1100b, 0xD == 1101b, 0xE == 1110b, 0xF == 1111b

So hex to binary is EASY!

- 0x2A == 00101010b
- 0x8001 == 1000000000000001b
- Binary to hex is easy too!
- 1000011110100000b == 0x87A0
Choosing Registers

• Arguments in $a0-3
• Results in $v0-1
• In a “leaf” function
  – use $t0-7 for everything and you won’t have to save and restore anything
  – if you need more then save $s0-7, use them, and then restore at end
• In a “non-leaf” function
  – use $t0-7 for temps that don’t need to be saved across calls
  – use $s0-7 for variables that you need across calls
• Always save $s0-7, $ra, $sp if you modify them.
• Use memory pointed to by $sp to save and restore registers, allocate arrays, etc.

pseudo instructions

• They aren’t REAL instructions
• You can think of them as
  – shorthand
  – macros
  – inline functions
  – syntactic sugar
• They are supposed to make your life easier and your programs easier to read
  – move $t1,$t0 ≜ add $t1, $t0, $zero
  – la $t0,foo ≜ lui $t0, UPPER16(foo)
    ori $t0, LOWER16(foo)
  – li $t0, 23 ≜ addi $t0, $zero, 23
  – li $t0, 0x2300AB ≜ lui $t0, 0x23
    ori $t0, 0x00AB