February 20

- Fixing the holes where the rain gets in ...
- Feel free to take the day off if you understand this stuff

Programming Language Issues (JAVA vs. C, Pointers vs. Arrays)

Representation issues (numbers, characters, addresses, bytes, endians)

Vocabulary (prefixes, MIPS)

Programming details (registers, pseudo instructions, align, functions)

What does that assembly language do? (on the test, problem 3.2)

Big Picture Questions (syscall? OS? Who/What/Where?)

Just enough C

For our purposes C is almost identical to JAVA except:

C has “functions”, JAVA has “methods”. function == method without “class”.
A global method.

C has “pointers” explicitly. JAVA has them but hides them under the covers.

C pointers

```
int i;      // simple integer variable
int a[10];  // array of integers
int *p;     // pointer to integer(s)
```

`*(expression)` is content of address computed by expression.

`a[k] == *(a+k)`

`a` is a constant of type "**`int *`**"

a[k] = a[k+1] EQUIV *(a+k) = *(a+k+1)

Legal uses of C Pointers

```
p = &i;      // & means address of
p = a;       // no need for & on a
p = &a[5];   // address of 5th element of a
*p           // value of location pointed by p
*p = 1;      // change value of that location
*(p+1) = 1; // change value of next location
p[1] = 1;    // exactly the same as above
p++;         // step pointer to the next element
```

So what happens when

```
p = &i;
```

What is value of p[0]? What is value of p[1]?

C Pointers vs. object size

Does “p++” really add 1 to the pointer?

NO! It adds 4.

Why 4?

```
char *q;
...
q++; // really does add 1
```

Clear123

```
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
  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:
```

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 i^{th} object add i*sizeof(object)
- More details:
  - int R[5] ✍ R is int* constant address of 20 bytes
  - R[i] ✍ *(R+i)
  - int *p = &R[3] ✍ p = (R+3) (p points 12 bytes after R)

Representations

You need to know your powers of 2!

2^0	1
2^1	2
2^2	4
2^3	8
2^4	16
2^5	32
2^6	64
2^7	128
2^8	256
2^9	512
2^10	1k=1024
2^20	1M
2^30	1G

Big Constants

The MIPS architecture only allows immediate constants to be 16 bits

So how do we get bigger constants?

lui sets the upper 16 bits from the 16 bit immediate field
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?

hi = BIG / 64k (e.g. 4,000,000 / 64k == 61)
lo = BIG % 64k (e.g. 4,000,000 % 64k == 2304)

```
lui     $t0, hi
ori     $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

Facts about bytes

- A byte is 8 bits.
- Usually the smallest addressable unit of storage.
- What means addressable?
  - something you can load or store directly
- Why sb AND sw?
  - sb only changes 1 byte in memory and has no alignment constraints
  - sw changes 4 consecutive bytes ONLY on 4 byte boundaries
  - could implement sw with multiple sb instructions
  - could implement sb using lw, logical operations, and sw

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

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    $t0, 1($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

```
foo:  .word 0            # foo is a 32 bit int
      li    $t0, 1       # t0 = 1
      la    $t1, foo     # t1 = &foo
      sb    $t0, 2($t1) # stores a byte at foo+2
```

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 == 0x00010000 == 64k

Big Endian ✍ t2 == 0x00000100 == 256

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    $t0, 3($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:

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

Little Endian ✍ t2 == 0x01000000 == 16M

Big Endian ✍ t2 == 0x00000001 == 1

Endians?

Consider the following code

```
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:

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

Little Endian ✍ t2 == 0x79726147

Big Endian ✍ t2 == 0x47617279

On BOTH machines:

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

ASCII Chart

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