14 September 2004

- First Exam next Tuesday 21 September
- Programming Questions?
- All of Chapter 3 and Appendix A are relevant and helpful to your programming assignments.
- More programming issues…

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:
  jr $ra
```


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:
  jr      $ra
```


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:
  jr      $ra
```


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:
  jr      $ra
```

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)

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

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

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