Arrays and Loops

Based on slides by Jared Moore

Looping through array

Looping through array is very common operation

You already know how to write loop in assembly – use same pattern here

Important to consider whether loop variable is *index* or *offset* into array – these can be different if array elements are larger than one byte

High-Level Code

```
int i;
int array[1000];

for (i = 0; i < 1000; i = i + 1)

  array[i] = array[i] * 8;
```

MIPS Assembly Code

```
# initialization code
  lui  $s0, 0x23B8
  ori  $s0, $s0, 0xF000
  addi $s1, $0, 0
  addi $t2, $0, 1000

loop:

done:
```

High-Level Code

```
int i;
int array[1000];

for (i = 0; i < 1000; i = i + 1)

  array[i] = array[i] * 8;
```

MIPS Assembly Code

```
# initialization code
  lui  $s0, 0x23B8
  ori  $s0, $s0, 0xF000
  addi $s1, $0, 0
  addi $t2, $0, 1000

loop:
  slt  $t0, $s1, $t2
  beq  $t0, $0, done
  sll  $t0, $s1, 2
  add  $t0, $t0, $s0
  lw   $t1, 0($t0)
  sll  $t1, $t1, 3
  sw   $t1, 0($t0)
  addi $s1, $s1, 1
  j    loop
done:
```

High-Level Code	MIPS Assembly Code

`int i;`
`int array[1000];`
	`# $s0 = array base address, $s1 = i`
	`# initialization code`
	`  lui  $s0, 0x23B8          # $s0 = 0x23B80000`
	`  ori  $s0, $s0, 0xF000    # $s0 = 0x23B8F000`
	`  addi $s1, $0, 0          # i = 0`
	`  addi $t2, $0, 1000       # $t2 = 1000`

`for (i = 0; i < 1000; i = i + 1)`	`loop:`
	`  slt  $t0, $s1, $t2       # i < 1000?`
	`  beq  $t0, $0, done       # if not, then done`
	`  sll  $t0, $s1, 2         # $t0 = i*4 (byte offset)`
	`  add  $t0, $t0, $s0       # address of array[i]`

`  array[i] = array[i] * 8;`	`  lw   $t1, 0($t0)         # $t1 = array[i]`
	`  sll  $t1, $t1, 3         # $t1 = array[i] * 8`
	`  sw   $t1, 0($t0)         # array[i] = array[i] * 8`
	`  addi $s1, $s1, 1         # i = i + 1`
	`  j    loop                # repeat`
	`done:`

Working with arrays

MIPS does not allow directly working with (adding, subtracting, etc.) values in memory

Get used to pattern of

- Load to register
- Do stuff
- Store to memory

Architectures with this property, like MIPS, referred to as load-store architectures

Don’t forget about size of entries

All examples so far have considered integer arrays

Integers generally stored as 32-bits – one word in MIPS

That is why offsets have been multiples of 4

Character-based data generally works with one byte at a time

Character Mapping

Numbers in [-128, 127] can be stored in a byte.

(8-bits)

Fewer than 256 characters on an English keyboard

But 8-bits is a byte, and memory is word addressable!

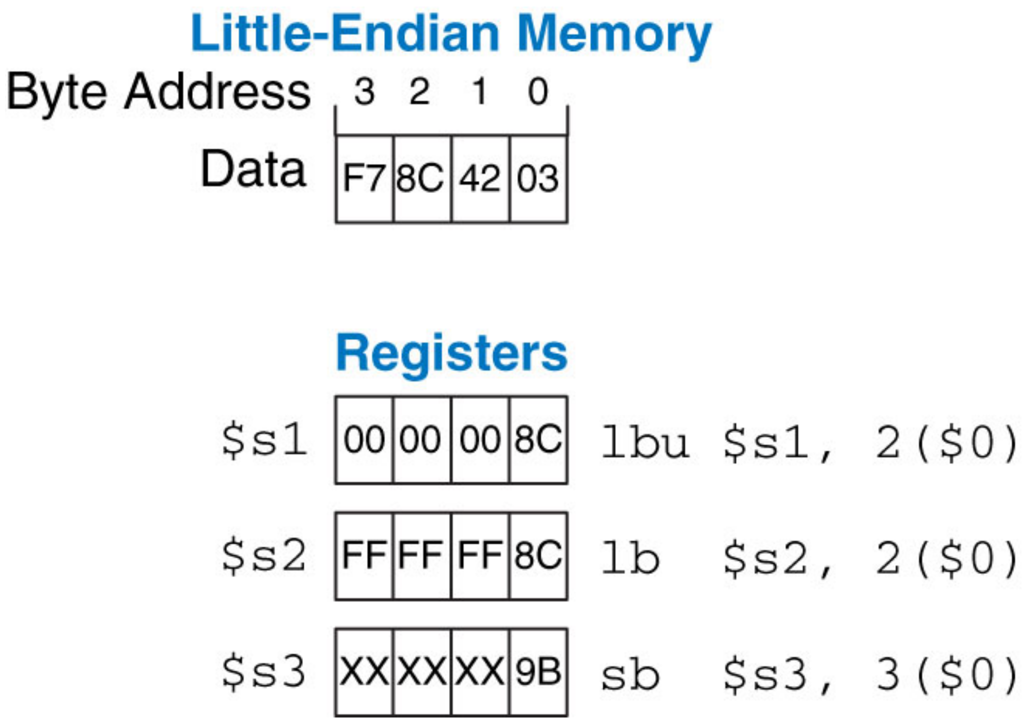
Must remember to use lbu, lb, and sb (load byte unsigned, load byte, and store byte)

ASCII

American Standard Code for Information Interchange

Each text character assigned a unique byte value.

#	Char	#	Char	#	Char	#	Char	#	Char	#	Char
20	space	30	0	40	@	50	P	60	`	70	p
21	!	31	1	41	A	51	Q	61	a	71	q
22	"	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	$	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	'	37	7	47	G	57	W	67	g	77	w
28	(	38	8	48	H	58	X	68	h	78	x
29	)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
2C	,	3C	<	4C	L	5C	\	6C	l	7C	|
2D	-	3D	=	4D	M	5D	]	6D	m	7D	}
2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
2F	/	3F	?	4F	O	5F	_	6F	o		


Little-Endian Memory
Byte Address 3 2 1 0
Data F7 8C 42 03
Registers
$s1 00 00 00 8C lbu $s1, 2($0)
$s2 FF FF FF 8C lb $s2, 2($0)
$s3 XX XX XX 9B sb $s3, 3($0)

	0	1	2	3	4	5
0x10007000	31	35	33	53	49	43

```
char src[6];
char dst[6];

for (int i = 5; i >= 0; i--)
   dst[5 - i] = src[i]
```

```
# This example should use something called the data segment,
# but we'll get to that later.

lui $s0, 0x1000
ori $s0, 0x7000  # s0 holds the base address of the source
lui $s1, 0x1000
ori $s1, 0x70A0  # s1 holds the base address of the destination
```

```
char src[6];
char dst[6];

for (int i = 5; i >= 0; i --)
   dst[5 - i] = src[i]
```

```
# This example should use something called the data segment,
# but we'll get to that later.

lui $s0, 0x1000
ori $s0, 0x7000  # s0 holds the base address of the source
lui $s1, 0x1000
ori $s1, 0x70A0  # s1 holds the base address of the destination

addi $t0, $0, 5  # int i = 5
addi $s2, $0, 5  # constant 5
for:  # for(i = 5; i >= 0; i--)
      blt $t0, $0, done

      ... actual work ...

      addi $t0, $t0, -1  # i--
      j for
done:
```

```
char src[6];
char dst[6];

for (int i = 5; i >= 0; i --)
  dst[5 - i] = src[i]
```

```
# This example should use something called the data segment,
# but we'll get to that later.

lui $s0, 0x1000
ori $s0, 0x7000  # s0 holds the base address of the source
lui $s1, 0x1000
ori $s1, 0x70A0  # s1 holds the base address of the destination

addi $t0, $0, 5  # int i = 5
addi $s2, $0, 5  # constant 5
for:  # for(i = 5; i >= 0; i--)
      blt $t0, $0, done

      add $t2, $s0, $t0  # address of current letter
      lbu $t3, 0($t2)  # value of current letter

      sub $t1, $s2, $t0  # offset of destination (5 - i)
      add $t1, $s1, $t1  # address of destination (offset + base)
      sb $t3, 0($t1)  # put current letter into destination array

      addi $t0, $t0, -1  # i--
      j for
done:
```

```
char src[6];
char dst[6];

for (int i = 5; i >= 0; i --)
  dst[5 - i] = src[i]
```

	0	1	2	3	4	5
src	31	35	33	53	49	43

0x10007004	49	43	X	X
0x10007000	31	35	33	53

What implicit assumption did we make when drawing the diagram of memory?

0x10007004	X	X	X	X
0x100070A0	X	X	X	X
…	…			
0x10007004	49	43	X	X
0x10007000	31	35	33	53

```
$t0 = 5  # i
$s0 = 0x10007000
$s1 = 0x100070A0
$s2 = 5  # constant
```

```
add $t2, $s0, $t0
lbu $t3, 0($t2)

sub $t1, $s2, $t0
add $t1, $s1, $t1

sb $t3, 0($t1)
```

0x10007004	X	X	X	X
0x100070A0	X	X	X	X
…		…		
0x10007004	49	43	X	X
0x10007000	31	35	33	53

```
$t0 = 5  # i
$s0 = 0x10007000
$s1 = 0x100070A0
$s2 = 5  # constant
```

```
add $t2, $s0, $t0          $t2 = 0x10007005
lbu $t3, 0($t2)            $t3 = 43

sub $t1, $s2, $t0          $t1 = 0
add $t1, $s1, $t1          $t1 = 0x100070A0

sb $t3, 0($t1)
```

0x10007004	X	X	X	X
0x100070A0	43	X	X	X
…		…		
0x10007004	49	43	X	X
0x10007000	31	35	33	53

```
$t0 = 5  # i
$s0 = 0x10007000
$s1 = 0x100070A0
$s2 = 5  # constant
```

```
add $t2, $s0, $t0          $t2 = 0x10007005
lbu $t3, 0($t2)            $t3 = 43

sub $t1, $s2, $t0          $t1 = 0
add $t1, $s1, $t1          $t1 = 0x100070A0

sb $t3, 0($t1)
```

Summary

1. Arrays allow us to store data sequentially in memory
2. Important to distinguish between addresses and values of array entries
   1. Use offsets to calculate address relative to the zeroth element
   2. Use `lw`/`sw` to interact with entries
3. Arrays are commonly used with loops, so familiarize yourself with looping through an array
4. Be careful to consider size of array entries