Unit 5: MIPS Arrays

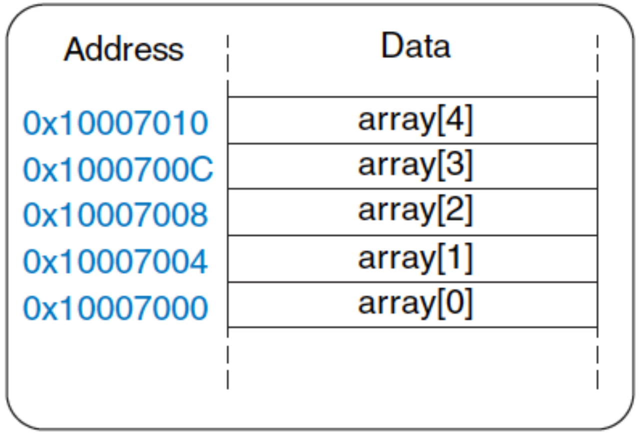
CSE 220: System Fundamental I

Stony Brook University

Joydeep Mitra

What is an Array?

- An array is a sequential collection of memory addresses
- Each array element has an *index*
- The no. of elements in an array is the *size* of the array



Five element array in memory

Array Indexing

- An array is stored in main memory starting at the *base address*
- The *base address* holds the value of array[0]
- How to access elements in an array?
  - Load base address of array into a register
    - ***lui*** followed by ***ori*** (***li*** pseudoinstruction) loads a 32-bit address into a register
    - ***la*** pseudoinstruction loads a 32-bit address from a label into a register
  - Use the **offset** to access subsequent elements. E.g.,
    - array[1] is stored at memory address = base address + 4 (assuming every element is 4 bytes long)

High-Level Code	MIPS Assembly Code

`int array[5];`	`# $s0 = base address of array`
	`lui $s0, 0x1000        # $s0 = 0x10000000`
	`ori $s0, $s0, 0x7000   # $s0 = 0x10007000`

`array[0] = array[0] * 8;`	`lw  $t1, 0($s0)        # $t1 = array[0]`
	`sll $t1, $t1, 3        # $t1 = $t1 << 3 = $t1 * 8`
	`sw  $t1, 0($s0)        # array[0] = $t1`

`array[1] = array[1] * 8;`	`lw  $t1, 4($s0)        # $t1 = array[1]`
	`sll $t1, $t1, 3        # $t1 = $t1 << 3 = $t1 * 8`
	`sw  $t1, 4($s0)        # array[1] = $t1`

N-Array Indexing

- Accessing elements from an array with many (N) elements requires a loop
- We iterate over the size of the array; each time calculating the address of the element appropriately

High-Level Code	MIPS Assembly Code

High-Level Code

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

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

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

MIPS Assembly Code

```
# $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

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]
  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:
```

Strings

- Strings are nothing but an array of characters
- Recall that every character has a unique ASCII encoding
  - S = 0x53 (83_{10}), a = 0x61 (97_{10}), A = 0x41 (65_{10})
  - Lower and upper-case English letters differ by 32_{10} (0x20).
- The NULL character (0x0) is used to indicate the end of a string.
- Recall the ASCII table earlier; we also have an extended ASCII table with 256-character encoding!

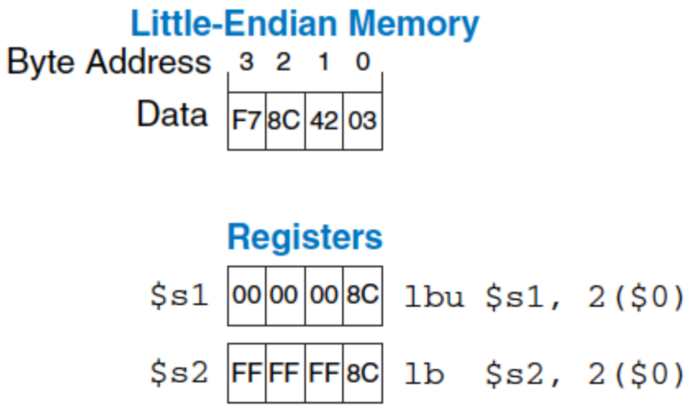
ASCII control characters		
00	NULL	(Null character)
01	SOH	(Start of Header)
02	STX	(Start of Text)
03	ETX	(End of Text)
04	EOT	(End of Trans.)
05	ENQ	(Enquiry)
06	ACK	(Acknowledgement)
07	BEL	(Bell)
08	BS	(Backspace)
09	HT	(Horizontal Tab)
10	LF	(Line feed)
11	VT	(Vertical Tab)
12	FF	(Form feed)
13	CR	(Carriage return)
14	SO	(Shift Out)
15	SI	(Shift In)
16	DLE	(Data link escape)
17	DC1	(Device control 1)
18	DC2	(Device control 2)
19	DC3	(Device control 3)
20	DC4	(Device control 4)
21	NAK	(Negative acknowl.)
22	SYN	(Synchronous idle)
23	ETB	(End of trans. block)
24	CAN	(Cancel)
25	EM	(End of medium)
26	SUB	(Substitute)
27	ESC	(Escape)
28	FS	(File separator)
29	GS	(Group separator)
30	RS	(Record separator)
31	US	(Unit separator)
127	DEL	(Delete)

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

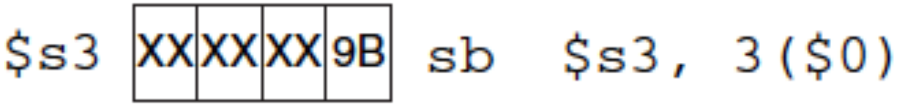
Extended ASCII characters							
128	Ç	160	á	192	└	224	Ó
129	ü	161	í	193	┴	225	ß
130	é	162	ó	194	┬	226	Ô
131	â	163	ú	195	├	227	Ò
132	ä	164	ñ	196	─	228	õ
133	à	165	Ñ	197	┼	229	Õ
134	å	166	ª	198	ã	230	µ
135	ç	167	º	199	Ã	231	þ
136	ê	168	¿	200	╚	232	Þ
137	ë	169	®	201	╔	233	Ú
138	è	170	¬	202	╩	234	Û
139	ï	171	½	203	╦	235	Ù
140	î	172	¼	204	╠	236	ý
141	ì	173	¡	205	═	237	Ý
142	Ä	174	«	206	╬	238	¯
143	Å	175	»	207	¤	239	´
144	É	176	░	208	ð	240	≡
145	æ	177	▒	209	Ð	241	±
146	Æ	178	▓	210	Ê	242	‗
147	ô	179	│	211	Ë	243	¾
148	ö	180	┤	212	È	244	¶
149	ò	181	Á	213	ı	245	§
150	û	182	Â	214	Í	246	÷
151	ù	183	À	215	Î	247	¸
152	ÿ	184	©	216	Ï	248	°
153	Ö	185	╣	217	┘	249	¨
154	Ü	186	║	218	┌	250	·
155	ø	187	╗	219	█	251	¹
156	£	188	╝	220	▄	252	³
157	Ø	189	¢	221	¦	253	²
158	×	190	¥	222	Ì	254	■
159	ƒ	191	┐	223	▀	255	nbsp

Loading/Storing Characters

- Recall that each character is a byte and each byte has a unique address
- *load byte* (**`lb`**) and *load byte unsigned* (**`lbu`**)
- When we load a byte what happens to the upper 24 bits in the destination register?
  - Fill the upper 24 bits with the 0s
    `lbu $s1, 2($0)`
  - Fill the upper 24 bits with the sign bit
    `lb $s2, 2($0)`



- Similarly, we can store bytes using **sb**



- Replaces 0xF7 with 0x9B in memory byte 3

Example

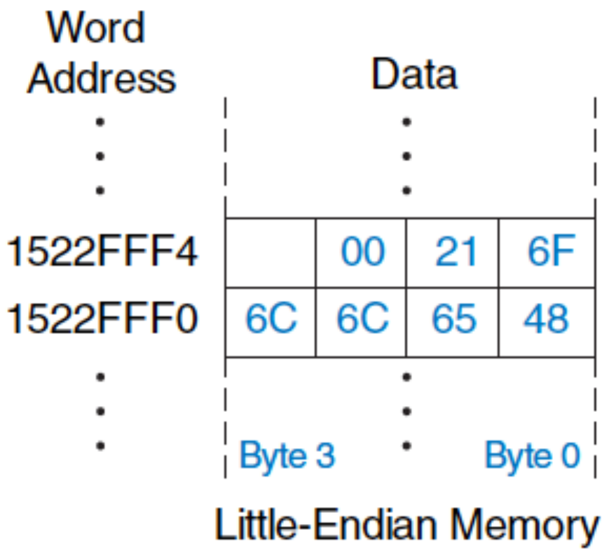
- Convert every character in a ten-character string from lowercase to upper case. Note that every character is 1 byte.

```
// high-level code

char chararray[10];
int i;
for (i = 0; i != 10; i = i + 1)
  chararray[i] = chararray[i] - 32;
```

```
# MIPS assembly code
# $s0 = base address of chararray, $s1 = i

         addi $s1, $0, 0      # i = 0
         addi $t0, $0, 10     # $t0 = 10
loop:    beq  $t0, $s1, done  # if i == 10, exit loop
         add  $t1, $s1, $s0   # $t1 = address of chararray[i]
         lb   $t2, 0($t1)     # $t2 = array[i]
         addi $t2, $t2, -32   # convert to upper case: $t2 = $t2 – 32
         sb   $t2, 0($t1)     # store new value in array:
                              # chararray[i] = $t2
         addi $s1, $s1, 1     # i = i+1
         j    loop            # repeat
done:
```

Variable Length Strings

- Strings may have variable length
- How do we know the end of string?
- MIPS uses the null character (0x00) to denote end of string
- E.g., string “hello” is stored as `0x48 65 6C 6C 6F 21 00` in memory



Declaring Arrays

- There are two to ways to declare arrays
  - Using name and size (in bytes)
    - E.g., `myArray : .space 20`
  - Using name and initial values
    - E.g., `myArray : .word 10 20 30 40 50`
      `myString : .asciiz “Hello!”`
- Non-character arrays do not terminate with null character; character arrays may or may not terminate with null characters (`.asciiz` vs `.ascii`).
- The programmer is responsible for tracking the size of an array

Memory Alignment

- Elements in an array need to be memory aligned; if not, you will see a memory alignment error
- 1-byte values can be read from or written to at any address
- 2-byte values can be accessed at even-numbered addresses
- 4-byte values can be accessed only at addresses that are multiples of 4
- Pay attention to these rules when working with arrays in MIPS.

```
.data
numbers:  .word 10 20 30

.text
main:
  la $s0, numbers   # load base address of array numbers into $s0
  lw $t0, 0($s0)    # $t0 = numbers[0]
  lw $t0, 1($s1)    # Memory alignment error! Word address should be multiple of 4
```

Integer Array Binary Search Example

```
.data
numbers: .word 10 20 30 40 50 60 70
.text
main:
    la $s0, numbers
    li $t1, 0           # $t1 = start = 0
    li $t2, 7           # $t2 = end = 7
    li $t0, 2           # $s0 = 2
    li $s1, 15          # $s1 = number = 20
    loop:
      bgt $t1, $t2, exit        # if start > end then exit loop
      add $t3, $t1,$t2          # $t3 = start + end
      div $t3, $t0              # set lo register to (start + end)/2
      mflo $t3                  # mid = $t3 = (start + end)/2
      sll $t4, $t3, 2           # i = $t4 = mid*4
      add $t4, $t4, $s0         # i = i + base address
      lw $t5, 0($t4)            # $t5 = numbers[i]
      beq $t5, $s1, found       # if numbers[i] = number then exit loop
      blt $s1, $t5, lt          # if number < numbers[i] then modify end
      addi $t1, $t3, 1          # if number > numbers[i] then start = mid + 1
      j default
      lt:
        addi $t2, $t3, -1       # if number < numbers[i] then end = mid - 1
      default:
        j loop
```

```
found:
    sll $t4, $t3, 2         # i = 4*mid
    add $t4, $t4, $s0       # $t4 = i = i + base address
    lw $a0, 0($t4)          # print numbers[i]
    li $v0, 1
    syscall
exit:
    li $v0, 10
       syscall
```

Reverse String Example

```
.data
str: .asciiz "Hello World!"

.text
main:
    li $s0, 0   # Forward pointer fp
    li $s1, -1  # Backward pointer bp
    la $s2, str
    loop:
        lb $t1, 0($s2)          # load a character in str
        beqz $t1, next          # Exit the loop if character is null character
        addi $s1, $s1, 1        # increment backward pointer by 1
        addi $s2, $s2, 1        # increment base address by 1
        j loop

    next:
        la $s2, str             # re-initialize $s2
    swap:
        bge $s0, $s1, end       # if fp >= bp then quit next loop
        add $t0, $s2, $s0       # $t0 is effective address + fp
        add $t1, $s2, $s1       # $t1 is effective address + bp
        lb $t3, 0($t0)          # $t3 = str[fp]
        lb $t4, 0($t1)          # $t4 = str[bp]
        sb $t3, 0($t1)          # str[bp] = $t3
        sb $t4, 0($t0)          # str[fp] = $t4
        addi $s0, $s0, 1        # increment fp by 1
        addi $s1, $s1, -1       # decrement bp by 1
        j swap
```

```
end:
    la $a0, str
    li $v0, 4
    syscall
```

Recap

- To access an element in the array we need to find the effective address
  `effective address = base address + elem_index*elem_size_in_bytes`
- `elem_index` starts at 0 and the `base address` denotes the address of the array
- The indexing operation in MIPS is *offset(register)*; the offset is always a number. E.g., `4($s0)`

Two Dimensional Arrays

- In higher-level languages 2D arrays are stored in either row-major form or column-major form
- MIPS stores all array elements sequentially; it doesn’t have a notion of row-major or column-major. We can use either!
- Consider a 2D array with 3 rows and 5 columns

```
a b c d e
f g h i j
k l m n o
```

- Row-major ordering will store elements in memory:

```
a b c d e f g h i j k l m n o
```

- Column-major ordering will store elements in memory:

```
a f k b g l c h m d i n e j o
```

Row-Major Ordering

- The effective address of an array at index [i][j] is

```
effective_addr = base_addr +
       i * size_of_a_row_in_bytes +
        j * elem_size_in_bytes

=> base_addr +
      i * num_columns * elem_size_in_bytes +
      j * elem_size_in_bytes

=> base_addr +
   elem_size_in_bytes * (i * num_columns + j)
```

```
a b c d e
f g h i j
k l m n o
```

Example

- Assume a string "HelloWorld" is stored in a 2D array. Access elements in row-major order.

```
.data
newline: .asciiz "\n"
space: .asciiz " "
arr: .word 1 2 3 4 5 6 7 8 9 10
rows: .word 5
cols: .word 2
```

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x0020000a	0x00000001	0x00000002	0x00000003	0x00000004	0x00000005	0x00000006	0x00000007
0x10010020	0x00000008	0x00000009	0x0000000a	0x00000005	0x00000002	0x73e56e79	0xa3cba32e	0x41d57ec0

Example

```
la $t0, arr
lw $t1, rows   # row count
lw $t2, cols   # column count
move $t3, $0   # initialize row counter

iter_arr_row:
 beq $t3, $t1, end_iter_row    # terminate row loop if row counter i reaches row limit
 move $t4, $0                  # initialize column counter
 iter_arr_col:
  beq $t4, $t2, end_iter_col   # terminate column loop if column counter j reaches column limit
  mul $t5, $t3, $t2            # i * column count
  add $t6, $t5, $t4            # (i * column count) + j
  sll $t6, $t6, 2              # 4 * (i * column count) + j
  add $t7, $t0, $t6            # base_addr + 4 * (i * column count) + j

  # print number at $t7
  lw $a0, 0($t7)
  li $v0, 1
  syscall
  #print space
  la $a0, space
  li $v0, 4
  syscall
  addi $t4, $t4, 1             # increment column counter j
  j iter_arr_col
 end_iter_col:
```

Example

```
end_iter_col:
 # print newline
 la $a0, newline
 li $v0, 4
 syscall

 addi $t3, $t3, 1      # increment row counter i
 j iter_arr_row
end_iter_row:
 li $v0, 10
 syscall
```

1 2
3 4
5 6
7 8
9 10

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x0020000a	0x00000001	0x00000002	0x00000003	0x00000004	0x00000005	0x00000006	0x00000007
0x10010020	0x00000008	0x00000009	0x0000000a	0x00000005	0x00000002	0x73e56e79	0xa3cba32e	0x41d57ec0

Column-Major Ordering

- Can you think of a similar formula for column major ordering?
- The effective address of an array at index [i][j] is

```
effective_addr = base_addr +
       j * size_of_a_col_in_bytes +
        i * elem_size_in_bytes

=> base_addr +
      j * num_rows * elem_size_in_bytes +
      j * elem_size_in_bytes

=> base_addr +
  elem_size_in_bytes * (j * num_rows + i)
```

a	b	c	d	e
f	g	h	i	j
k	l	m	n	o

Example

```
iter_arr_row:
 beq $t3, $t1, end_iter_row    # terminate row loop if row counter i reaches row limit
 move $t4, $0                  # initialize column counter j
 iter_arr_col:
  beq $t4, $t2, end_iter_col   # terminate column loop if column counter j reaches column limit
  mul $t5, $t4, $t1            # j * row count
  add $t6, $t5, $t3            # (j * row count) + i
  sll $t6, $t6, 2              # 4 * (j * row count) + i
  add $t7, $t0, $t6            # base_addr + 4 * (j * row count) + i

  # print number at $t7
  lw $a0, 0($t7)
  li $v0, 1
  syscall
  #print space
  la $a0, space
  li $v0, 4
  syscall
  addi $t4, $t4, 1             # increment column counter j
  j iter_arr_col
 end_iter_col:
  # print newline
  la $a0, newline
```

```
1 6
2 7
3 8
4 9
5 10
```

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x0020000a	0x00000001	0x00000002	0x00000003	0x00000004	0x00000005	0x00000006	0x00000007
0x10010020	0x00000008	0x00000009	0x0000000a	0x00000005	0x00000002	0x191d805c	0x1d8a29c2	0x07c4e922