ECE260: Fundamentals of Computer Engineering

Accessing and Addressing Memory

James Moscola
Dept. of Engineering & Computer Science
York College of Pennsylvania

American Standard Code for Information Interchange (ASCII)

- Common character encoding standard
- Available in 7-bit and extended 8-bit
  - 7-bit version encodes 27 (127) characters
  - 8-bit version encodes 28 (256) characters
- Not so great for languages based on non-English alphabets
- Unicode has replaced ASCII in many contexts
  - Backwards compatible with ASCII (UTF-8)
  - UTF-16 commonly used

Binary	Character	Binary	Character	Binary	Character	Binary	Character
00000000	NUL	00100000	SP	01000000	@	01100000	`
00000001	SOH	00100001	!	01000001	A	01100001	a
00000010	STX	00100010	"	01000010	B	01100010	b
00000011	ETX	00100011	#	01000011	C	01100011	c
00000100	EOT	00100100	$	01000100	D	01100100	d
00000101	ENQ	00100101	%	01000101	E	01100101	e
00000110	ACK	00100110	&	01000110	F	01100110	f
00000111	BEL	00100111	'	01000111	G	01100111	g
00001000	BS	00101000	(	01001000	H	01101000	h
00001001	HT	00101001	)	01001001	I	01101001	i
00001010	LF	00101010	*	01001010	J	01101010	j
00001011	VT	00101011	+	01001011	K	01101011	k
00001100	FF	00101100	,	01001100	L	01101100	l
00001101	CR	00101101	-	01001101	M	01101101	m
00001110	SO	00101110	.	01001110	N	01101110	n
00001111	SI	00101111	/	01001111	O	01101111	o
00010000	DLE	00110000	0	01010000	P	01110000	p
00010001	DC1	00110001	1	01010001	Q	01110001	q
00010010	DC2	00110010	2	01010010	R	01110010	r
00010011	DC3	00110011	3	01010011	S	01110011	s
00010100	DC4	00110100	4	01010100	T	01110100	t
00010101	NAK	00110101	5	01010101	U	01110101	u
00010110	SYN	00110110	6	01010110	V	01110110	v
00010111	ETB	00110111	7	01010111	W	01110111	w
00011000	CAN	00111000	8	01011000	X	01111000	x
00011001	EM	00111001	9	01011001	Y	01111001	y
00011010	SUB	00111010	:	01011010	Z	01111010	z
00011011	ESC	00111011	;	01011011	[	01111011	{
00011100	FS	00111100	<	01011100	\	01111100	|
00011101	GS	00111101	=	01011101	]	01111101	}
00011110	RS	00111110	>	01011110	^	01111110	~
00011111	US	00111111	?	01011111	_	01111111	DEL

Storing Numbers: ASCII vs. UTF-16 vs. Binary

- ASCII and Unicode are great for storing text strings … *not* so great for storing numeric data
- Example: storing the number 1,000,000,000 in 3 different formats

Storing as ASCII characters

```
Requires 10 8-bit ASCII characters
Total space requirement:  10 chars * 8 bits/char = 80 bits
```

Storing as UTF-16 characters

```
Requires 10 16-bit UTF-16 characters
Total space requirement:  10 chars * 16 bits/char = 160 bits
```

Storing as binary

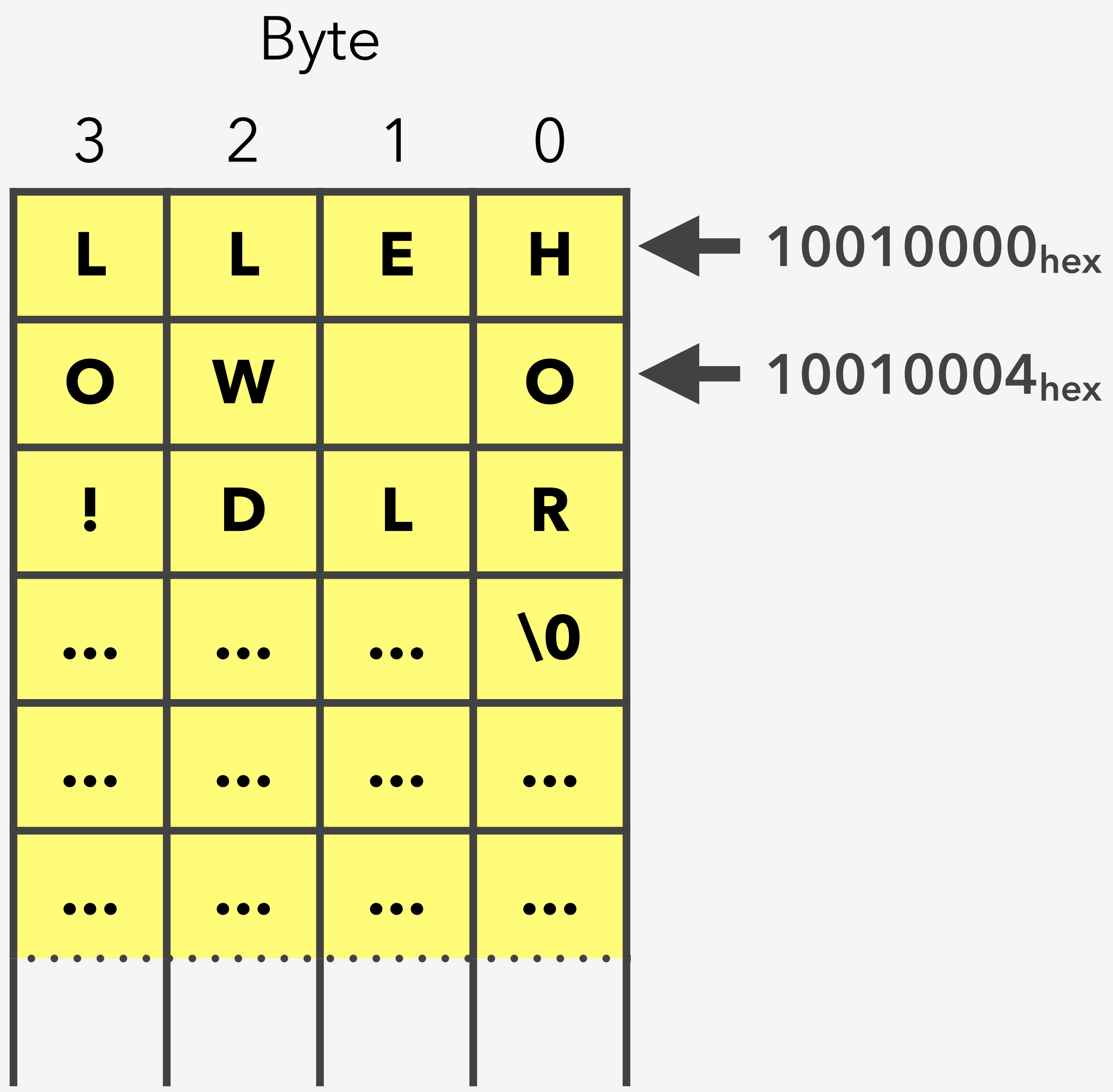
```
Requires a single 32-bit word
Total space requirement:  32 bits
```

Getting Byte Data from Memory

- Strings are commonly stored as 8-bit ASCII character sequences
- Oftentimes only need 8 bits to store your numeric data
- Different architectures offer different methods for reading a byte from memory
  - Hard way
    - Read entire word from memory, then use shifting and bitmasks to extract the desired byte
  - Easier way
    - Use a special instruction for reading a byte directly from memory

Getting a Byte from Memory (the hard way)

- Hard way
  - Read entire word from memory
  - Shift to move desired byte into least significant byte
  - Use bitmask to clear unwanted bytes



- Example: get byte 2 from memory word 10010004_{hex}
  - Assume \$s0 contains base address of "HELLO WORLD!"

```
lw   $t0, 4($s0)        # load entire word from memory into register $t0
srl  $t0, $t0, 16       # move desired bits [23:16] into position [7:0] of register $t0
andi $t0, $t0, 0x00FF   # AND with bitmask to keep bottom 8 bits, clear top 8 bits
  # register $t0 now has the 8-bit ASCII value for "W"  (i.e. 0x00000057)
```

Getting a Byte from Memory (the MIPS way)

- MIPS provides instructions to load and store bytes
  - I-type instruction with 16-bit immediate offset
  - Load byte from memory address ($s7 + 1)

Load byte with sign extension

```
lb  $t0, 1($s7)
```

Load byte without sign extension

```
lbu $t0, 1($s7)
```

  - Store byte into memory address ($s7 + 13)

Store bottom 8 bits of $t0

```
sb $t0, 13($s7)
```

Byte

3	2	1	0	
L	L	E	H	← 10010000_{hex}
O	W		O	← 10010004_{hex}
!	D	L	R	
…	…	…	\0	
…	…	…	…	
…	…	…	…	

NOTE: When loading bytes, memory address need *NOT* be a multiple of 4! Address can be ANY value: Examples: 0x00, 0x01, 0x02, 0x03, …

- Example: Load "W", write it back
  - Assume $s0 contains base address of "HELLO WORLD!"

```
lbu   $t0, 6($s0)       # load "W" from memory into register $t0
sb    $t0, 6($s0)       # overwrite memory with value
```

Example: Copying a C String

- Recall C strings end with a null terminator character '\0'
- Example C code – copy string y to string x

```
void strcpy (char x[], char y[]) {
  int i = 0;
  while ((x[i] = y[i]) != '\0') {
    i += 1;
  }
}
```

Note that the while condition both assigns x[i] AND compares the value to '\0'

- Assume the following:
  - Base address for x is passed to procedure in register $a0
  - Base address for y is passed to procedure in register $a1
  - Local variable i is stored in register $s0

Example: Copying a C String (continued)

- First example of strcpy directly mimics C code

```
strcpy:
    addi $sp, $sp, -4      # adjust $sp to make room on stack to save 1 word
    sw   $s0, 0($sp)       # As CALLEE: save $s0 for parent
    add  $s0, $zero, $zero # initialize i = 0

L1: add  $t1, $s0, $a1     # add offset to base address of y and store in $t1 (BYTE ADDRESSED!!!)
    lbu  $t2, 0($t1)       # $t2 = y[i] (i.e. load y[i] into $t2)
    add  $t3, $s0, $a0     # add offset to base address of x and store in $t3 (BYTE ADDRESSED!!!)
    sb   $t2, 0($t3)       # x[i] = y[i] (i.e. write to new location … do the copy)

    beq  $t2, $zero, L2    # exit loop if y[i] == 0 (i.e. it is the NULL terminator character)
    addi $s0, $s0, 1       # i = i + 1 (i.e. increment i)
    j    L1                # next iteration of loop

L2: lw   $s0, 0($sp)       # As CALLEE: restore saved $s0 for parent
    addi $sp, $sp, 4       # pop 1 item from stack
    jr   $ra               # return to CALLER
```

Example: Copying a C String (continued … better)

- Better example of strcpy – eliminate the $s0 register so there is no need to save it on stack

```
strcpy:
    addi $sp, $sp, -4        # adjust $sp to make room on stack to save 1 word
    sw   $s0, 0($sp)         # As CALLEE: save $s0 for parent
    add  $t0, $zero, $zero # initialize i = 0 in $t0 (THIS IS A LEAF PROCEDURE!!)

L1: add  $t1, $t0, $a1      # add offset to base address of y and store in $t1 (BYTE ADDRESSED!!!)
    lbu  $t2, 0($t1)        # $t2 = y[i] (i.e. load y[i] into $t2)
    add  $t3, $t0, $a0      # add offset to base address of x and store in $t3 (BYTE ADDRESSED!!!)
    sb   $t2, 0($t3)        # x[i] = y[i] (i.e. write to new location … do the copy)

    beq  $t2, $zero, L2     # exit loop if y[i] == 0 (i.e. it is the NULL terminator character)
    addi $t0, $t0, 1        # i = i + 1 (i.e. increment i)
    j    L1                 # next iteration of loop

L2: lw   $s0, 0($sp)        # As CALLEE: restore saved $s0 for parent
    addi $sp, $sp, 4        # pop 1 item from stack
    jr   $ra                # return to CALLER
```

Example: Copying a C String (continued … best?)

- Best[?] example of strcpy – eliminate i completely and just do pointer arithmetic

```
strcpy:
    addi $sp, $sp, -4          # adjust $sp to make room on stack to save 1 word
    sw   $s0, 0($sp)           # As CALLEE: save $s0 for parent
    add  $t0, $zero, $zero # initialize i = 0 in $t0 (THIS IS A LEAF PROCEDURE!!)

L1: add  $t1, $t0, $a1         # add offset to base address of y and store in $t1 (BYTE ADDRESSED!!!)
    lbu  $t2, 0($a1)           # $t2 = y[i] (i.e. load y[i] into $t2)
    add  $t3, $s0, $a0         # add offset to base address of x and store in $t3 (BYTE ADDRESSED!!!)
    sb   $t2, 0($a0)           # x[i] = y[i] (i.e. write to new location … do the copy)

    beq  $t2, $zero, L2        # exit loop if y[i] == 0 (i.e. it is the NULL terminator character)
    addi $t0, $t0, 1           # i = i + 1 (i.e. increment i)
    addi $a0, $a0, 1           # increment x array pointer by 1 byte
    addi $a1, $a1, 1           # increment y array pointer by 1 byte
    j    L1                    # next iteration of loop
L2: lw   $s0, 0($sp)           # As CALLEE: restore saved $s0 for parent
    addi $sp, $sp, 4           # pop 1 item from stack
    jr   $ra                   # return to CALLER
```

Example: Copying a C String (continued … best?)

- Best[?] example of strcpy – cleaned up

```
strcpy:
L1: lbu  $t2, 0($a1)       # $t2 = y[i] (i.e. load y[i] into $t2)
    sb   $t2, 0($a0)       # x[i] = y[i] (i.e. write to new location … do the copy)
    beq  $t2, $zero, L2    # exit loop if y[i] == 0 (i.e. it is the NULL terminator character)
    addi $a1, $a1, 1       # increment y array pointer by 1 byte
    addi $a0, $a0, 1       # increment x array pointer by 1 byte
    j    L1                # next iteration of loop
L2:
    jr   $ra               # return to CALLER
```

Getting a Halfwords from Memory (the MIPS way)

- MIPS also provides instructions to load and store <u>halfwords</u>
  - I-type instruction with 16-bit immediate offset
  - Load halfword from memory address ($s7 + 2)

Load halfword with sign extension

```
lh  $t0, 2($s7)
```

Load halfword without sign extension

```
lhu $t0, 2($s7)
```

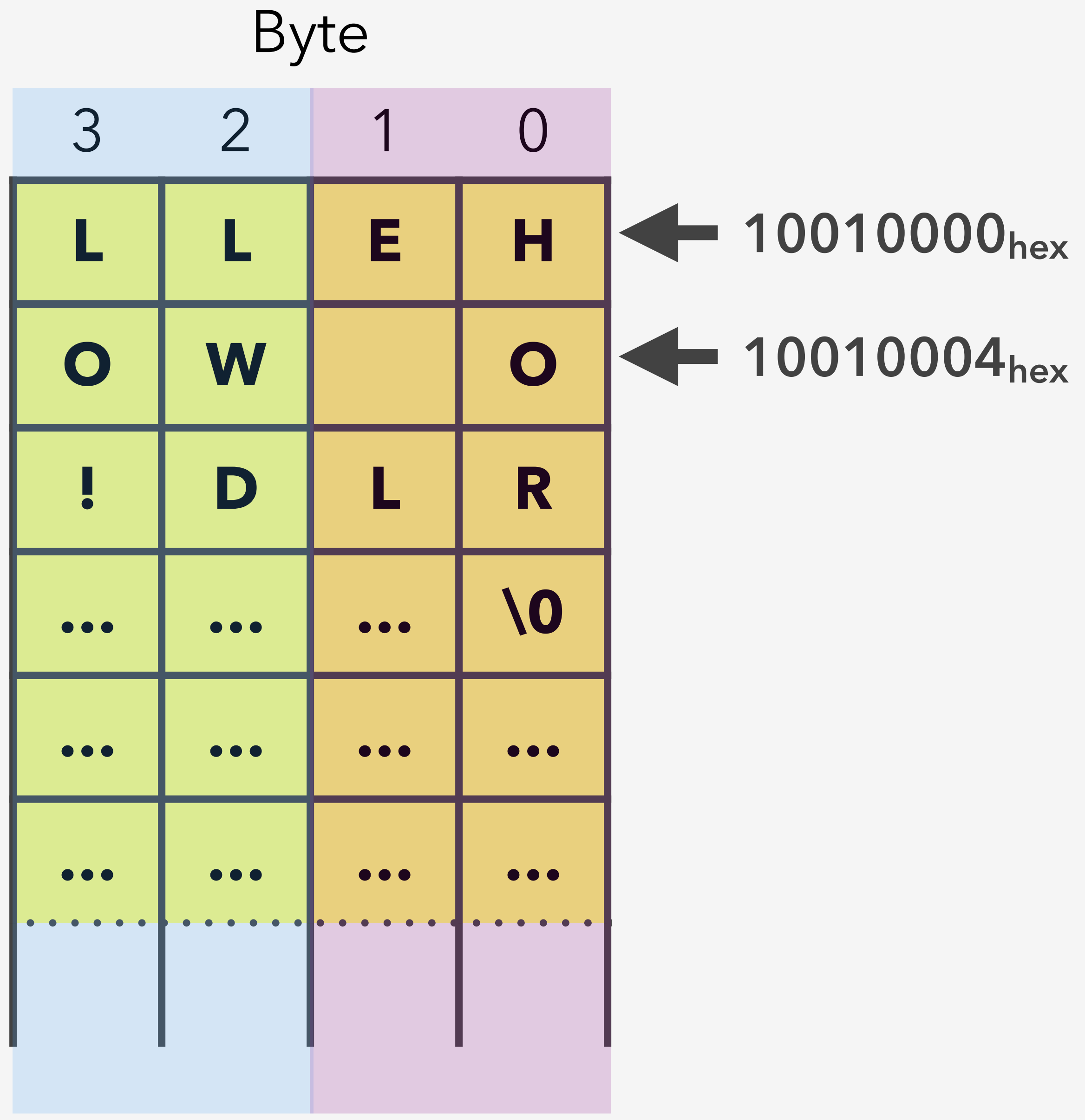
  - Store halfword into memory address ($s7 + 14)

Store bottom 16 bits of $t0

```
sh $t0, 14($s7)
```

- Example: Load “D!”, write it back
  - Assume $s0 contains base address of “HELLO WORLD!”

```
lhu  $t0, 10($s0)      # load “D!” from memory into register $t0
sh   $t0, 10($s0)      # overwrite memory with value
```



NOTE: When loading halfwords, memory address must be multiple of 2! Examples: 0x00, 0x02, 0x04, 0x06, ...