

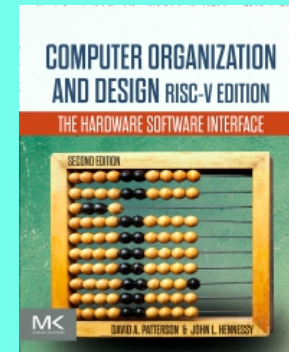
# CSC 411

Computer Organization (Spring 2022)  
Lecture 5b: RISC-V Memory Organization

Prof. Marco Alvarez, University of Rhode Island

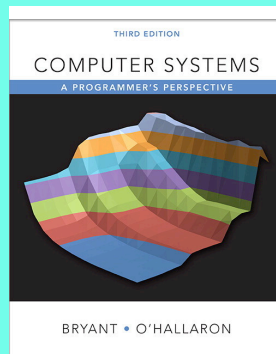
## Disclaimer

Some of the following slides are adapted from:  
Computer Organization and Design (Patterson and Hennessy)  
The Hardware/Software Interface



## Disclaimer

Some of the following slides are copied from:  
Computer Systems (Bryant and O'Hallaron)  
A Programmer's Perspective



## Byte-Oriented Memory Organization



### ■ Programs refer to data by address

- Conceptually, envision it as a very large array of bytes
  - In reality, it's not, but can think of it that way
- An address is like an index into that array
  - and, a pointer variable stores an address

### ■ Note: system provides private address spaces to each "process"

- Think of a process as a program being executed
- So, a program can clobber its own data, but not that of others

## Machine Words

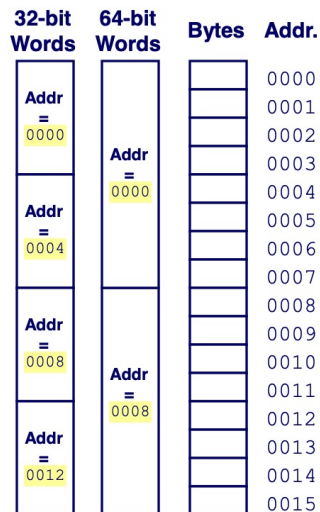
- Any given computer has a “Word Size”
  - Nominal size of integer-valued data
    - and of addresses
  - Until recently, most machines used 32 bits (4 bytes) as word size
    - Limits addresses to 4GB ( $2^{32}$  bytes)
  - Increasingly, machines have 64-bit word size
    - Potentially, could have 18 EB (exabytes) of addressable memory
    - That's  $18.4 \times 10^{18}$
  - Machines still support multiple data formats
    - Fractions or multiples of word size
    - Always integral number of bytes

## Example Data Representations

C Data Type	Typical 32-bit	Typical 64-bit	x86-64
char	1	1	1
short	2	2	2
int	4	4	4
long	4	8	8
float	4	4	4
double	8	8	8
pointer	4	8	8

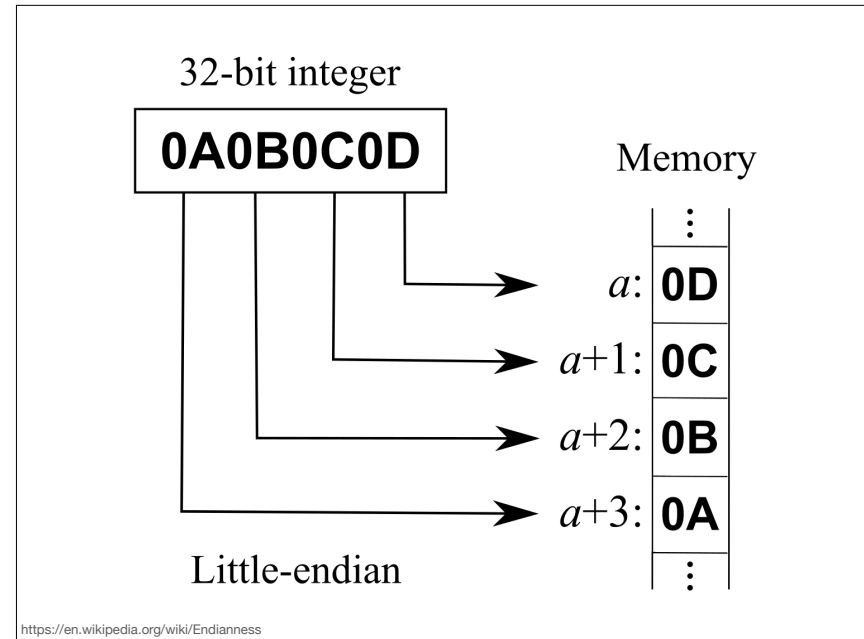
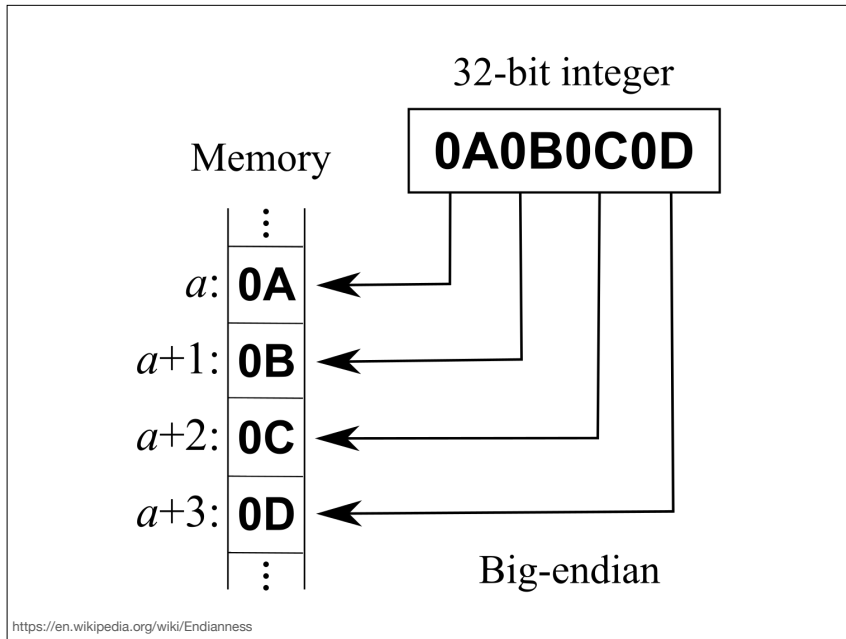
## Word-Oriented Memory Organization

- Addresses Specify Byte Locations
  - Address of first byte in word
  - Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)



## Byte Ordering

- So, how are the bytes within a multi-byte word ordered in memory?
- Conventions
  - Big Endian: Sun (Oracle SPARC), PPC Mac, *Internet*
    - Least significant byte has highest address
  - Little Endian: *x86*, ARM processors running Android, iOS, and Linux
    - Least significant byte has lowest address



Carnegie Mellon

## Byte Ordering Example

- Example**
  - Variable x has 4-byte value of 0x01234567
  - Address given by &x is 0x100

**Big Endian**

0x100	0x101	0x102	0x103
01	23	45	67

**Little Endian**

0x100	0x101	0x102	0x103
67	45	23	01

Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Carnegie Mellon

## Representing Integers

Decimal: 15213  
Binary: 0011 1011 0110 1101  
Hex: 3 B 6 D

**int A = 15213;**

**long int C = 15213;**

**int B = -15213;**

Two's complement representation

Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

## Examining Data Representations

### Code to Print Byte Representation of Data

- Casting pointer to unsigned char \* allows treatment as a byte array

```
typedef unsigned char *pointer;

void show_bytes(pointer start, size_t len){
    size_t i;
    for (i = 0; i < len; i++){
        printf("%p\t0x%.2x\n", start+i, start[i]);
        printf("\n");
    }
}
```

#### Printf directives:

%p: Print pointer  
%x: Print Hexadecimal

## show\_bytes Execution Example

```
int a = 15213;
printf("int a = 15213;\n");
show_bytes((pointer) &a, sizeof(int));
```

### Result (Linux x86-64):

```
int a = 15213;
0x7fffb7f71dbc    6d
0x7fffb7f71dbd    3b
0x7fffb7f71dbe    00
0x7fffb7f71dbf    00
```

## Representing Strings

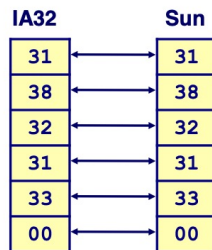
### Strings in C

- Represented by array of characters
- Each character encoded in ASCII format
  - Standard 7-bit encoding of character set
  - Character "0" has code 0x30
    - Digit  $i$  has code  $0x30+i$
- String should be null-terminated
  - Final character = 0

### Compatibility

- Byte ordering not an issue

```
char S[6] = "18213";
```



ASCII control characters				ASCII printable characters				Extended ASCII characters			
00	NULL	(Null character)		32	space	64	@	96	.		
01	SOH	(Start of Header)		33	!	65	A	97	a		
02	STX	(Start of Text)		34	"	66	B	98	b		
03	ETX	(End of Text)		35	#	67	C	99	c		
04	EOT	(End of Trans.)		36	\$	68	D	100	d		
05	ENQ	(Enquiry)		37	%	69	E	101	e		
06	ACK	(Acknowledgement)		38	&	70	F	102	f		
07	BEL	(Bell)		39	'	71	G	103	g		
08	BS	(Backspace)		40	(	72	H	104	h		
09	HT	(Horizontal Tab)		41	)	73	I	105	i		
10	LF	(Line feed)		42	*	74	J	106	j		
11	VT	(Vertical Tab)		43	+	75	K	107	k		
12	FF	(Form feed)		44	,	76	L	108	l		
13	CR	(Carriage return)		45	-	77	M	109	m		
14	SO	(Shift Out)		46	.	78	N	110	n		
15	SI	(Shift In)		47	/	79	O	111	o		
16	DLE	(Data link escape)		48	0	80	P	112	p		
17	DC1	(Device control 1)		49	1	81	Q	113	q		
18	DC2	(Device control 2)		50	2	82	R	114	r		
19	DC3	(Device control 3)		51	3	83	S	115	s		
20	DC4	(Device control 4)		52	4	84	T	116	t		
21	NAK	(Negative acknowl.)		53	5	85	U	117	u		
22	SYN	(Synchronous idle)		54	6	86	V	118	v		
23	ETB	(End of trans. block)		55	7	87	W	119	w		
24	CAN	(Cancel)		56	8	88	X	120	x		
25	EM	(End of medium)		57	9	89	Y	121	y		
26	SUB	(Substitute)		58	:	90	Z	122	z		
27	ESC	(Escape)		59	;	91	[	123	{		
28	FS	(File separator)		60	<	92	\	124			
29	GS	(Group separator)		61	=	93	]	125	}		
30	RS	(Record separator)		62	>	94	^	126	~		
31	US	(Unit separator)		63	?	95	_				
127	DEL	(Delete)									
128	Ç			160	à			192	À		
129	ù			161	á			193	Á		
130	é			162	â			194	Â		
131	â			163	ã			195	Ã		
132	ä			164	ä			196	Ä		
133	å			165	Å			197	Å		
134	ä			166	ä			198	ä		
135	ç			167	ç			199	Ç		
136	è			168	è			200	È		
137	é			169	É			201	É		
138	ê			170	Ê			202	Ê		
139	ï			171	Ï			203	Ï		
140	ì			172	Ì			204	Ì		
141	í			173	Í			205	Í		
142	Ä			174	«			206	»		
143	Å			175	»			207	«		
144	É			176				208			
145	Ê			177				209			
146	Ë			178				210			
147				179				211			
148				180				212			
149				181				213			
150				182				214			
151				183				215			
152				184				216			
153				185				217			
154				186				218			
155				187				219			
156				188				220			
157				189				221			
158				190				222			
159				191				223			
								224			
								225			
								226			
								227			
								228			
								229			
								230			
								231			
								232			
								233			
								234			
								235			
								236			
								237			
								238			
								239			
								240			
								241			
								242			
								243			
								244			
								245			
								246			
								247			
								248			
								249			
								250			
								251			
								252			
								253			
								254			
								255			

# Memory operands

## RISC-V interpreter

### RISC-V Interpreter

Input your RISC-V code here:

```
1 lui t0, 0x0
2 lui t1, 0x7ffff
3
4
5
6
7
8
9
10
11
12
13
14
15
16
```

Reset Step Run CPU: 32 Hz

The most recent instructions will be shown here when stepping.

#### Features

- Reset to load the code, Step one instruction, or Run all instructions
- Set a breakpoint by clicking on the line number (only for Run)
- View registers on the right, memory on the bottom of this page

#### Supported instructions

- Arithmetic: **ADD**, **ADDI**, **SUB**
- Logical: **AND**, **ANDI**, **OR**, **ORI**, **XOR**, **XORI**
- Shift: **SLT**, **SLTI**, **SLTU**, **SLTIU**
- Memory: **LB**, **SB**, **LBU**, **LD**, **ST**
- Jumps: **JAL**, **JALR**
- Branches: **BEQ**, **BNE**, **BLT**, **BLTU**, **BGE**, **BGEU**

RISC-V Reference: riscv-spec-v2.2.pdf

Init Value	Register	Decimal	Hex	Binary
0	x0 (zero)	0	0x00000000	0x00000000000000000000000000000000
0	x1 (ra)	0	0x00000000	0x00000000000000000000000000000000
0	x2 (sp)	0	0x00000000	0x00000000000000000000000000000000
0	x3 (gp)	0	0x00000000	0x00000000000000000000000000000000
0	x4 (tp)	0	0x00000000	0x00000000000000000000000000000000
0	x5 (t0)	0	0x00000000	0x00000000000000000000000000000000
0	x6 (t1)	0	0x00000000	0x00000000000000000000000000000000
0	x7 (t2)	0	0x00000000	0x00000000000000000000000000000000
0	x8 (t3)	0	0x00000000	0x00000000000000000000000000000000
0	x9 (t4)	0	0x00000000	0x00000000000000000000000000000000
0	x10 (a0)	0	0x00000000	0x00000000000000000000000000000000
0	x11 (a1)	0	0x00000000	0x00000000000000000000000000000000
0	x12 (a2)	0	0x00000000	0x00000000000000000000000000000000
0	x13 (a3)	0	0x00000000	0x00000000000000000000000000000000
0	x14 (a4)	0	0x00000000	0x00000000000000000000000000000000
0	x15 (a5)	0	0x00000000	0x00000000000000000000000000000000
0	x16 (a6)	0	0x00000000	0x00000000000000000000000000000000
0	x17 (a7)	0	0x00000000	0x00000000000000000000000000000000
0	x18 (a8)	0	0x00000000	0x00000000000000000000000000000000
0	x19 (a9)	0	0x00000000	0x00000000000000000000000000000000
0	x20 (a10)	0	0x00000000	0x00000000000000000000000000000000
0	x21 (a11)	0	0x00000000	0x00000000000000000000000000000000
0	x22 (a12)	0	0x00000000	0x00000000000000000000000000000000
0	x23 (a13)	0	0x00000000	0x00000000000000000000000000000000
0	x24 (a14)	0	0x00000000	0x00000000000000000000000000000000
0	x25 (a15)	0	0x00000000	0x00000000000000000000000000000000
0	x26 (a16)	0	0x00000000	0x00000000000000000000000000000000
0	x27 (a17)	0	0x00000000	0x00000000000000000000000000000000
0	x28 (a18)	0	0x00000000	0x00000000000000000000000000000000
0	x29 (a19)	0	0x00000000	0x00000000000000000000000000000000
0	x30 (a20)	0	0x00000000	0x00000000000000000000000000000000
0	x31 (a21)	0	0x00000000	0x00000000000000000000000000000000

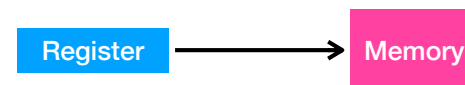
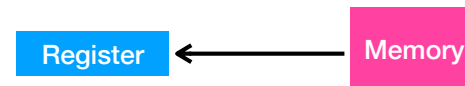
<https://www.cs.cornell.edu/courses/cs3410/2019sp/riscv/interpreter/>

## Memory operands

- Main memory used for composite data
  - arrays, structures, dynamic data
- To apply arithmetic operations:
  - **load** values from memory into registers
  - **store** result from register to memory
- Memory is byte addressed
  - each address identifies an 8-bit byte
- RISC-V is **Little Endian**
  - least-significant byte at least address of a word
- RISC-V does not require words to be aligned in memory
  - unlike some other ISAs

## Fetching and storing data

- Values can be fetched from memory
  - using a **load** instruction
- Values can be stored in memory
  - using a **store** instruction



## Load instruction

- Load doubleword from memory

destination register      source address  
**ld**    x9,    64(x22)

- destination can be any register
- source address uses a constant (**offset**) added to the register in parentheses (**base**)

## Store instruction

- Store doubleword into memory

source register      destination address  
**sd**    x9,    96(x22)

- source can be any register
- destination address uses a constant (**offset**) added to the register in parentheses (**base**)

## Example with memory operand

- C code

```
g = h + A[8];
```

- Compiled RISC-V code

- index 8 requires offset of 64 (8 bytes)

```
# assume g in x21, h in x22  
# assume base address of A in x23  
ld        x21, 64(x23)  
add       x21, x21, x22
```

## Example with memory operand

- C code

```
A[12] = h + A[8];
```

- Compiled RISC-V code

- try yourself

```
# assume h in x21  
# assume base address of A in x22  
ld        x9, 64(x22)  
add       x9, x21, x9  
sd        x9, 96(x22)
```

## Registers vs memory

- Operating on memory data requires **loads** and **stores**
  - consider memory latency and additional instructions to be executed
- **Registers** are a fast read/write memory right on the CPU that can hold values
- Compiler must use registers for variables as much as possible
  - only spill to memory for less frequently used variables
  - **register optimization** is important!