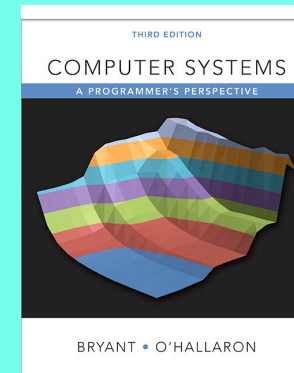# CSC 411

Computer Organization (Spring 2023)
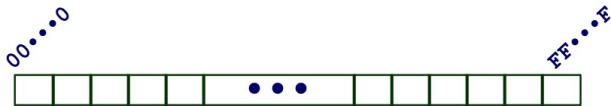**Lecture 6: RISC-V Memory Organization**

Prof. Marco Alvarez, University of Rhode Island

Disclaimer

Some of the following slides are copied from:

Computer Systems (Bryant and O'Hallaron)

A Programmer's Perspective

THIRD EDITION
COMPUTER SYSTEMS
A PROGRAMMER'S PERSPECTIVE

BRYANT • O'HALLARON

Byte-Oriented Memory Organization

- **Programs refer to data by address**
  - Imagine all of RAM as an enormous array of bytes
  - An address is an index into that array
    - A pointer variable stores an address
- **System provides a private *address space* to each "process"**
  - A process is an instance of a program, being executed
  - An address space is one of those enormous arrays of bytes
  - Each program can see only its own code and data within its enormous array
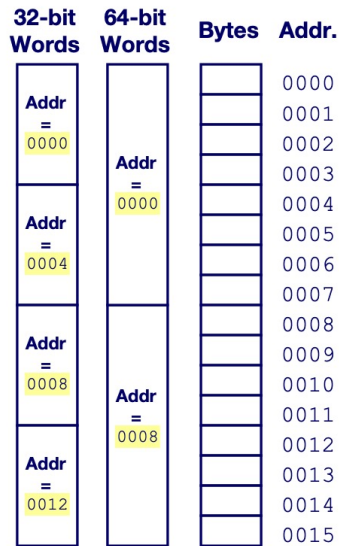 - We'll come back to this later ("virtual memory" classes)

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 X 10^{18}

  - Machines still support multiple data formats
    - Fractions or multiples of word size
    - Always integral number of bytes

Addresses *Always* Specify Byte Locations

- Address of a word is address of the first byte in the word
- Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)
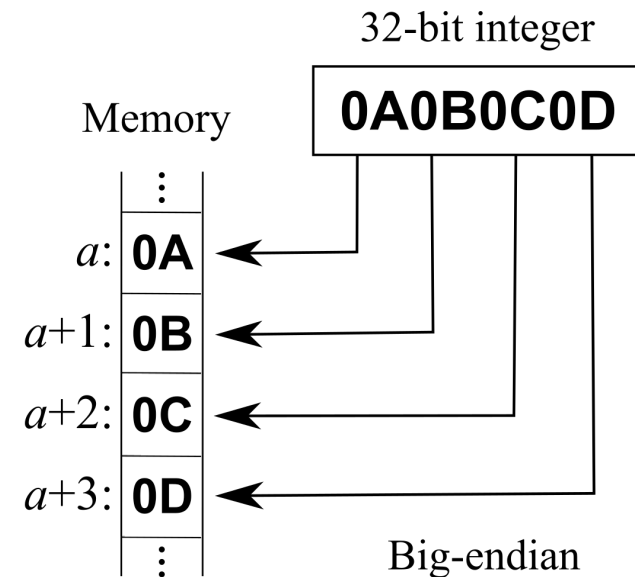
32-bit Words	64-bit Words	Bytes	Addr.
Addr = 0000	Addr = 0000		0000
			0001
			0002
			0003
Addr = 0004			0004
			0005
			0006
			0007
Addr = 0008	Addr = 0008		0008
			0009
			0010
			0011
Addr = 0012			0012
			0013
			0014
			0015

Example Data Representations

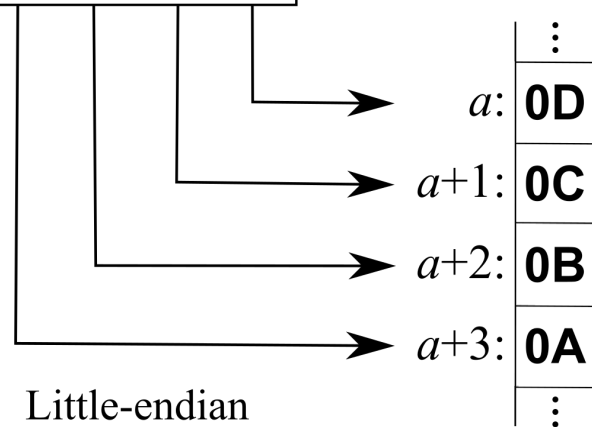| C Data Type	Typical 32-bit	Typical 64-bit	x86-64
char	1	1	1
short	2	2	2
int	4	4	4
long	4	8	8
float	4	4	4
double	8	8	8
pointer	4	8	8

Byte Ordering

- So, how are the bytes within a multi-byte word ordered in memory?
- Conventions
  - Big Endian: Sun, PPC Mac, *network packet headers*
    - Least significant byte has highest address
  - Little Endian: *x86*, ARM processors running Android, iOS, and Windows
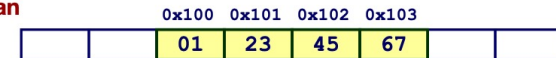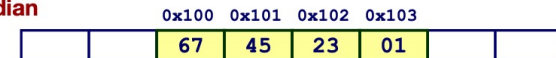   - Least significant byte has lowest address

---

32-bit integer

Memory

0A0B0C0D

a: **0A**

$a+1$: **0B**

$a+2$: **0C**

$a+3$: **0D**

Big-endian

Slide 1 (top-left)

32-bit integer

0A0B0C0D

Memory

Little-endian

a:	**0D**
$a+1$:	**0C**
$a+2$:	**0B**
$a+3$:	**0A**

Slide 2 (top-right)

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		

Slide 3 (bottom-left)

Representing Integers

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

`int A = 15213;`

Increasing addresses

IA32, x86-64	Sun
6D	00
3B	00
00	3B
00	6D

`long int C = 15213;`

IA32	x86-64	Sun
6D	6D	00
3B	3B	00
00	00	3B
00	00	6D
	00	
	00	
	00	
	00	

`int B = -15213;`
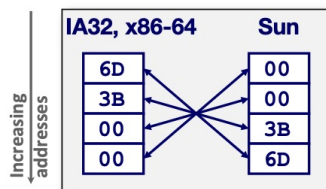
IA32, x86-64	Sun
93	FF
C4	FF
FF	C4
FF	93
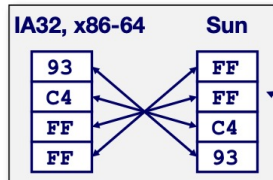
Two's complement representation

Slide 4 (bottom-right)

Examining Data Representations

- **Code to Print Byte Representation of Data**
  - Casting pointer to unsigned char * allows treatment as a byte array

```c
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 Pointers

```
int B = -15213;
int *P = &B;
```

Sun	IA32	x86-64
EF	AC	3C
FF	28	1B
FB	F5	FE
2C	FF	82
		FD
		7F
		00
		00

Different compilers & machines assign different locations to objects

Even get different results each time run program

Representing Strings

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

- **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

IA32	Sun
31	31
38	38
32	32
31	31
33	33
00	00

ASCII control characters			ASCII printable characters			Extended ASCII characters				
00	NULL	(Null character)	32	space	64	@	96	`	128	Ç	160	á	192	└	224	Ó
01	SOH	(Start of Header)	33	!	65	A	97	a	129	ü	161	í	193	┴	225	ß
02	STX	(Start of Text)	34	"	66	B	98	b	130	é	162	ó	194	┬	226	Ô
03	ETX	(End of Text)	35	#	67	C	99	c	131	â	163	ú	195	├	227	Ò
04	EOT	(End of Trans.)	36	$	68	D	100	d	132	ä	164	ñ	196	─	228	õ
05	ENQ	(Enquiry)	37	%	69	E	101	e	133	à	165	Ñ	197	┼	229	Õ
06	ACK	(Acknowledgement)	38	&	70	F	102	f	134	å	166	ª	198	ã	230	µ
07	BEL	(Bell)	39	'	71	G	103	g	135	ç	167	º	199	Ã	231	þ
08	BS	(Backspace)	40	(	72	H	104	h	136	ê	168	¿	200	╚	232	Þ
09	HT	(Horizontal Tab)	41	)	73	I	105	i	137	ë	169	®	201	╔	233	Ú
10	LF	(Line feed)	42	*	74	J	106	j	138	è	170	¬	202	╩	234	Û
11	VT	(Vertical Tab)	43	+	75	K	107	k	139	ï	171	½	203	╦	235	Ù
12	FF	(Form feed)	44	,	76	L	108	l	140	î	172	¼	204	╠	236	ý
13	CR	(Carriage return)	45	-	77	M	109	m	141	ì	173	¡	205	═	237	Ý
14	SO	(Shift Out)	46	.	78	N	110	n	142	Ä	174	«	206	╬	238	¯
15	SI	(Shift In)	47	/	79	O	111	o	143	Å	175	»	207	¤	239	´
16	DLE	(Data link escape)	48	0	80	P	112	p	144	É	176	░	208	ð	240	≡
17	DC1	(Device control 1)	49	1	81	Q	113	q	145	æ	177	▒	209	Ð	241	±
18	DC2	(Device control 2)	50	2	82	R	114	r	146	Æ	178	▓	210	Ê	242	‗
19	DC3	(Device control 3)	51	3	83	S	115	s	147	ô	179	│	211	Ë	243	¾
20	DC4	(Device control 4)	52	4	84	T	116	t	148	ö	180	┤	212	È	244	¶
21	NAK	(Negative acknowl.)	53	5	85	U	117	u	149	ò	181	Á	213	ı	245	§
22	SYN	(Synchronous idle)	54	6	86	V	118	v	150	û	182	Â	214	Í	246	÷
23	ETB	(End of trans. block)	55	7	87	W	119	w	151	ù	183	À	215	Î	247	¸
24	CAN	(Cancel)	56	8	88	X	120	x	152	ÿ	184	©	216	Ï	248	°
25	EM	(End of medium)	57	9	89	Y	121	y	153	Ö	185	╣	217	┘	249	¨
26	SUB	(Substitute)	58	:	90	Z	122	z	154	Ü	186	║	218	┌	250	·
27	ESC	(Escape)	59	;	91	[	123	{	155	ø	187	╗	219	█	251	¹
28	FS	(File separator)	60	<	92	\	124			156	£	188	╝	220	▄	252	³
29	GS	(Group separator)	61	=	93	]	125	}	157	Ø	189	¢	221	▌	253	²
30	RS	(Record separator)	62	>	94	^	126	~	158	×	190	¥	222	▐	254	■
31	US	(Unit separator)	63	?	95	_			159	ƒ	191	┐	223	▀	255	nbsp
127	DEL	(Delete)														

https://computersciencewiki.org/index.php/ASCII

Disclaimer

Some of the following slides are adapted from:

Computer Organization and Design (Patterson and Hennessy)

The Hardware/Software Interface

COMPUTER ORGANIZATION AND DESIGN RISC-V EDITION
THE HARDWARE SOFTWARE INTERFACE
SECOND EDITION
DAVID A. PATTERSON & JOHN L. HENNESSY
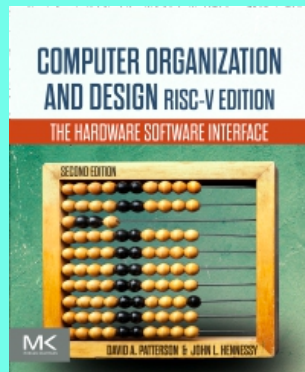
Memory operands

Data transfer instructions

‣ Main memory used for composite data

- arrays, structures, dynamic data

‣ Register <-> Memory instructions

- **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**

- RISC-V and x86 do not require words to be aligned in memory

  - alignment restriction forces **words** to start at addresses that are multiples of the word size

Fetching and storing data

‣ Values can be fetched from memory

- using a **load** instruction
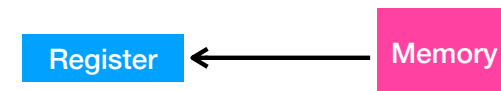
Register	←	Memory

‣ Values can be stored in memory

- using a **store** instruction
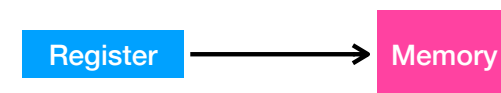
Register	→	Memory

Load instructions

‣ Load **word** from memory

destination register   source address

```
lw   x9,  32(x22)
```

- destination can be any register

- source address uses a constant (**offset**) added to the register value (**base**)

Store instruction

‣ Store **word** into memory

source register   destination address

```
sw   x9,  48(x22)
```

- source can be any register

- destination address uses a constant (**offset**) added to the register value (**base**)

Example with memory operand

‣ C code

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

‣ Compiled RISC-V code

- index 8 requires offset of 32 (4 bytes)

```
# assume g in x21, h in x22
# assume base address of A in x23
lw      x21, 32(x23)
add     x21, x21, x22
```

Example with memory operand

‣ C code

```
A[7] = h + A[5];
```

‣ Compiled RISC-V code

- try yourself (assume h in x21 and base address of A in x22)

```
lw      x9, 20(x22)
add     x9, x21, x9
sw      x9, 28(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!