CSC 411

Computer Organization (Spring 2024)
Lecture 11: RISC-V load/store

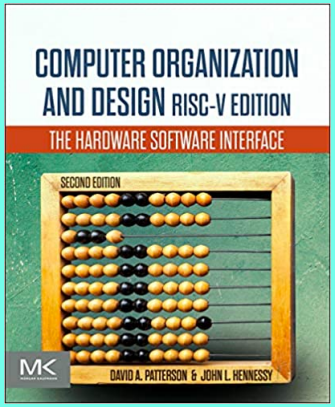
Prof. Marco Alvarez, University of Rhode Island

Disclaimer

Some figures and slides are adapted from:

Computer Organization and Design (Patterson and Hennessy)

The Hardware/Software Interface



So far ...

- Addition / subtraction

```
add rd, rs1, rs2
sub rd, rs1, rs2
```

- Add immediate

```
addi rd, rs1, imm
```

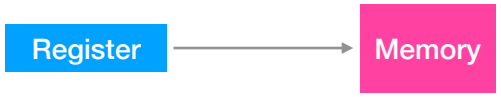
Data transfer instructions

- RV32I is a load-store architecture
  - only **load** and **store** instructions access memory
  - arithmetic instructions only operate on CPU registers
- 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 strongly encouraged)
    - alignment restriction forces **words** to start at addresses that are multiples of the word size

Loading and storing data

- Values can be loaded from memory

- Values can be stored in memory



Loading words

- The **lw** instruction loads a 32-bit value from memory into rd

destination register — base register

```
lw   rd,   imm(rs1)
```

offset

`destination register gets memory word from address (base + offset)`

Storing words

- The **sw** instruction stores a 32-bit value from rs2 to memory

source register — base register

```
sw   rs2,   imm(rs1)
```

offset

`memory at address (base + offset) gets the value in source register`

Practice

- C code
  - assume all are `int` variables

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

- RISC-V code
  - index 8 requires offset of 32 (4 bytes)
  - assume g in x1, h in x2, and base address of A in x3

```
lw   x1, 32(x3)
add  x1, x1, x2
```

Practice

- C code
  - assume all are `int` variables

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

- RISC-V code
  - assume h in x1 and base address of A in x2
  - can use an additional register x3

Loading and storing bytes

- Instructions for transferring individual bytes from/to memory
- Use same format as **`lw`** and **`sw`**
  - load byte: **`lb`**
    - loads a 8-bit value from memory, then **sign-extends** to 32-bits before storing in rd
  - load unsigned byte: **`lbu`**
    - loads a 8-bit value from memory but then **zero-extends** to 32-bits before storing in rd
  - store byte: **`sb`**
    - stores 8-bit value from the **low bits** of register rs2 to memory

Practice

- What are the resulting values in x3 and x4 respectively?
  - express the values using 32-bits (in hexadecimal notation)

```
addi   x1, x0, 0x80AB
sw     x1, 0(x2)
lb     x3, 1(x2)
lbu    x4, 0(x2)
```

Practice

- Translate the following C code into RISC-V
  - can use an additional register x3

```
// assume x, y are integer pointers
// stored in x1 and x2 respectively
*x = *y;
```

ASCII representation

ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter
32	space	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	DEL

Registers vs memory

- Operating on memory data requires **loads** and **stores**
  - consider memory latency
  - **registers** are a fast read/write memory right on the CPU that can hold values
  - memory is byte-addressable but `lw` and `sw` can transfer entire words
- Compiler must use registers for variables as much as possible
  - only spill to memory for less frequently used variables
  - **register optimization** is important!