Memory Operations

Caiwen Ding

Department of Computer Science and Engineering

University of Connecticut

CSE3666: Introduction to Computer Architecture

Outline

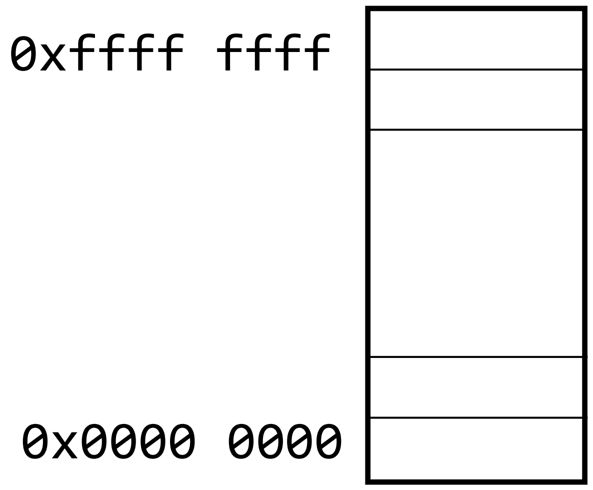
- Memory
- Load/store instructions
  - Move data between registers and memory
- Data of other types
  - Words, halfwords, and bytes
  - ASCII strings
- Address alignment
- Endianness

Reading: Sections 2.3.

References: Reference card in the book.

Memory

- Memory is an array of bytes
- Each byte is numbered. The number is the address
- Each address identifies a byte
  - If a data item is larger than one byte, its address is the first byte in memory
- A 32-bit address space supports 4 GiB
  - A 64-bit address space supports 16 EiB (exbibytes)



Kibibytes (KiB) vs kilobytes (KB)

- We always mean KiB, MiB, GiB

Decimal term	Abbreviation	Value	Binary term	Abbreviation	Value	% Larger
kilobyte	KB	10^3	kibibyte	KiB	2^{10}	2%
megabyte	MB	10^6	mebibyte	MiB	2^{20}	5%
gigabyte	GB	10^9	gibibyte	GiB	2^{30}	7%
terabyte	TB	10^{12}	tebibyte	TiB	2^{40}	10%
petabyte	PB	10^{15}	pebibyte	PiB	2^{50}	13%
exabyte	EB	10^{18}	exbibyte	EiB	2^{60}	15%
zettabyte	ZB	10^{21}	zebibyte	ZiB	2^{70}	18%
yottabyte	YB	10^{24}	yobibyte	YiB	2^{80}	21%

A video on Kilobyte or Kibibyte?
https://www.youtube.com/watch?v=ZRQVPcgf5yE

Using data in memory

- Many ISAs like RISC-V cannot compute on data in memory directly
  - Must load data into a register first

- Two kinds of instructions to exchange data between registers and memory
  - Load : memory to register
  - Store : register to memory

- Need to know the address to read/write memory
  - You need an address to save/fetch items

Variables defined in your program

```
        .align 2
# a word with initial value 3
x:      .word  3

# two words with initial values
y:      .word  4, 5



// in C
int    x = 3;
int    y[2] = {4, 5};
```

How do you get the address of a variable in a register?

	Address	Value
	0x00FE 901C	
	0x00FE 9018	
	0x00FE 9014	5
y	0x00FE 9010	4
x	0x00FE 900C	3
	0x00FE 9008	
	0x00FE 9004	
	0x00FE 9000	

How to get the address of a variable in a register?

- Basically, we need to load a 32-bit constant in a register

- We can also use a pseudoinstruction LA
  - It is converted into a couple of real instructions
  - We will learn the real instructions later

```
la    s1, x
```

a label in assembly
or
a variable name

Load/Store instructions

```
# load a word from mem into rd
# Reg[rd] = Mem[Reg[rs1] + offset]

lw   rd,  offset(rs1)

# save a word to mem
# Mem[Reg[rs1] + offset] = Reg[rs2]
sw   rs2, offset(rs1)
```

- Load/store words
- Offset is also called displacement
- The effective address is the value in register plus the offset

address = Reg[rs1] + offset

Effective address is the calculated memory address

Write 0(rs1) even if the offset is 0
Assembler may support "(s1)" as a pseudoinstruction

Example

- Each row in the table is a byte
- Assume a's address is in s1. Write RISC-V instructions to do

```
int a, b;

b = a
```

Var name	Address	Value
	0x00FE 9007	
	0x00FE 9006	
	0x00FE 9005	
b	0x00FE 9004	
	0x00FE 9003	
	0x00FE 9002	
	0x00FE 9001	
a	0x00FE 9000	

Answer

- Each row in the table is a byte
- Assume a's address is in s1. Write RISC-V instructions to do

```
b = a;
```

```
lw  t0,0(s1)
sw  t0,4(s1)
```

Var name	Address	Value
	0x00FE 9007	
	0x00FE 9006	
	0x00FE 9005	
b	0x00FE 9004	
	0x00FE 9003	
	0x00FE 9002	
	0x00FE 9001	
a	0x00FE 9000	

Array

Suppose word array A starts from 0x9000 (stored in s1).

How do we read/write A[0], a[1], etc.?

Given a valid index i, how do we access A[i]?

C	Address	Value
	…	
A[7]	0x901C	700
A[6]	0x9018	600
A[5]	0x9014	500
A[4]	0x9010	400
A[3]	0x900C	300
A[2]	0x9008	200
A[1]	0x9004	100
A[0]	0x9000	0

4 bytes in A[1]

Address	Value
0x9007	0x00
0x9006	0x00
0x9005	0x00
0x9004	0x64

Memory Example

Variable	Register
`h`	`s2`
`A's addr`	`s3`

C code:

```
A[20] = h + A[5];
```

A is a word array.

Memory Example

C code:

```
A[20] = h + A[5];
```

Variable	Register
h	s2
A's addr	s3

A is a word array.

```
# RISC-V code
lw   t0, 20(s3)       # load A[5]
add  t1, t0, s2
sw   t1, 80(s3)       # save to A[20]
```

offset

base register

Example: Clearing an array

```
// assume a's address is in s1
for (i = 0; i < 8; i = i + 1)
        a[i] = 0;
```

How do we do this ???

Address	Value
0x9024	
0x9020	
0x901C	a[7]
0x9018	a[6]
0x9014	a[5]
0x9010	a[4]
0x900C	a[3]
0x9008	a[2]
0x9004	a[1]
0x9000	a[0]

Clearing an array - v1

- Array indexing involves
  - Multiplying index by element size
  - Adding to array base address

```
for (i = 0; i < 8; i = i + 1)
      a[i] = 0;

      i = 0
      goto test
loop:
      Compute 4*i
      Add to base address (in s1)
      Write to the address
      Increment i
test: If (i < 8) goto loop
```

Address	Value
0x9024	
0x9020	
0x901C	a[7]
0x9018	a[6]
0x9014	a[5]
0x9010	a[4]
0x900C	a[3]
0x9008	a[2]
0x9004	a[1]
0x9000	a[0]

Example: array copying

C code:

```
for (i = 0; i < 100; i ++)
    B[i] = A[i];
```

Variable	Register
`i`	`s1`
`A's addr`	`s2`
`B's addr`	`s3`

A and B are word arrays.

```
for (i = 0; i < 100; i ++) {
      t = A[i];    # how do we do this ???
      B[i] = t;
 }
```

Array copying - v1

Variable	Register
i	s1
A's addr	s2
B's addr	s3

```
# copy array. array version

for (i = 0; i < 100; i ++)
      B[i] = A[i];

# RISC-V code
      li    s4, 100
      li    s1, 0
      beq   x0, x0, test # we know s1 < s4
loop:
      slli  t0, s1, 2    # t0 = i * 4
      add   t2, t0, s2   # compute addr of A[i]
      lw    t1, 0(t2)
      add   t3, t0, s3   # compute addr of B[i]
      sw    t1, 0(t3)
      addi  s1, s1, 1
test: bne   s1, s4, loop # 7 instructions in the loop
```

Address alignment

- Alignment: Data item's address is a multiple of its size
  - Address of words is a multiple of 4
  - Address of half words is a multiple of 2
- Data addresses do not have to be aligned in RISC-V, but misalignment will cause poor performance
  - The addresses must be aligned in this course!

```
# align the address of next variable to 2^2 = 4
       .align 2
```

You want to sit with you family when you fly!

Byte order

How is a word stored in memory?

```
# x1 is 0x01020304
sw   x1, 0x100(x0)
```

Which byte goes to address 0x100?

Memory Address	Value
`0x0000 0103`	
`0x0000 0102`	
`0x0000 0101`	
`0x0000 0100`	

Endianness

```
#  x1 is 0x01020304
sw    x1, 0x100(x0)
```

Big-endian: The highest byte goes to the lowest memory address.

Memory Address	Value
0x0000 0103	04
0x0000 0102	03
0x0000 0101	02
0x0000 0100	01

Little-endian: The lowest byte goes to the lowest memory address.

Memory Address	Value
0x0000 0103	01
0x0000 0102	02
0x0000 0101	03
0x0000 0100	04

RISC-V uses little endian.

Question

What are the bits in t0 after the following instruction?

```
    lw    t0, 0x200(x0)
```

A. 0x3265 81AC

B. 0xAC81 6532

C. 0xCA18 5623

D. 0x6532 AC81

E. None of the above

Memory Address	Value
0x0000 0203	0x32
0x0000 0202	0x65
0x0000 0201	0x81
0x0000 0200	0xAC

Data of other sizes

- RISC-V supports data of other sizes
  - Each type can be signed or unsigned

Number of bits	Name	C types (typical)
8 bits	byte	char
16 bits	half word	short int
32 bits	word	int, long int

```
# load signed (sign extended) byte/halfword
lb/lh    rd, offset(rs1)

# load unsigned (0 extended) byte/halfword
lbu/lhu  rd, offset(rs1)

# Store the lowest byte/halfword
sb/sh    rs2, offset(rs1)
```

Load/store instructions

addr is the same for all load/store instructions

addr: `offset(rs1)`

Data size	Load signed	Load unsigned	Store
`Word (32 bits)`	`lw rd,addr`		`sw rs2,addr`
`Half word (16 bits)`	`lh rd,addr`	`lhu rd,addr`	`sh rs2,addr`
`Byte (8 bits)`	`lb rd,addr`	`lbu rd,addr`	`sb rs2,addr`

Why is there no 'lwu' here?

Strings in our programs

```
# We will only deal with ASCII strings in this course
s:      .string       "CSE3666"     # or use .asciz

# print a string (terminated by null)
la      a0, s
addi    a7, x0, 4
ecall

// in C
char    s[10] = "CSE3666";
```

What is the value in a0 after la?

	Address	Value
	0x00FE 9017	0
	0x00FE 9016	54
	0x00FE 9015	54
	0x00FE 9014	54
	0x00FE 9003	51
	0x00FE 9002	69
	0x00FE 9001	83
s	0x00FE 9000	67

Example: string copy

Copy string s to d.

Use pointers.

Variable	Register
`s's addr`	`a1`
`d's addr`	`a0`
`c`	`t0`

```
// C code
char c;
do {
        c = *s;
        *d = c;
        s += 1;
        d += 1;
} while (c);
```

*s means s[0]
*d means d[0]

Example: string copy answer

Copy string s to d.

Variable	Register
s's addr	a1
d's addr	a0
c	t0

```c
// C code
char c;
do {
      c = *s;
      *d = c;
      s += 1;
      d += 1;
} while (c);
```

```
# RISC-V

loop:
      lb    t0, 0(a1)
      sb    t0, 0(a0)
      addi  a1, a1, 1
      addi  a0, a0, 1
      bne   t0, x0, loop
```

Question

What is the value in t0 after the following instruction?

```
lb   t0, 0x201(x0)
```

A. `0x0000 00AC`
B. `0x0000 0081`
C. `0xFFFF FFAC`
D. `0xFFFF FF81`
E. None of the above

Memory Address	Value
`0x0000 0203`	`0x32`
`0x0000 0202`	`0x65`
`0x0000 0201`	`0x81`
`0x0000 0200`	`0xAC`

Registers vs. Memory

- Registers are faster to access than memory
- Operating on memory data requires loads and stores
  - More instructions to be executed
- Compiler must use registers for variables as much as possible
  - Only spill to memory for less frequently used variables
  - Register optimization is important!

We need to know where data are stored when coding!

Pitfalls

- A word has four bytes
  - LW loads four bytes
    - There are four bytes in a word! They are located at sequential addresses
  - Sequential word addresses are incremented by 4!

- Sequential half words/bytes are NOT incremented by 4
  - Pay attention to the size
  - Sequential bytes do have sequential addresses

- Offset is a 12-bit 2's completed number, sign extended to 32 bits
  - If offset is too large, add offset with instructions

- Byte order matters

Summary of memory

- Memory is byte addressed
  - Each address identifies an 8-bit byte
  - A 32-bit address space support 4 GiB memory
- RV32I supports byte (8 bits), half-word (16 bits), and word (32 bits)
- Words and half-words should be aligned in memory
  - They must be aligned in this course
  - Although they do not have to in real processors, misalignment leads to poor performance
- Endianness affects the order of bytes when data are converted from/to bytes
  - RISC-V is little endian

Further thinking and reading

- How do you find out the endianness of a processor?

- Byte order is very important
  - Unicode BOM (byte order mark), U+FEFF
    - Search the Internet and find out how the mark is represented in UTF-16 (BE), UTF-16(LE), UTF-32(BE), and UTF-32(LE)

Find out what load/store instructions do

- Ask the following questions for load instructions
  - What is the address?
  - What are the bytes/is the byte the memory module finds at the address?
  - If there are multiple bytes, how should you put them together?
  - If necessary, how do you extend the byte(s) to 32 bits?

- Ask the following questions for store instructions
  - What is the address?
  - How many bytes are going to be stored in the address?
  - What is the order of bytes in the memory?

Question

What are the bits in t0 after the following instructions?

```
lh   t0, 0x200(x0)
```

A. 0x0000 81AC
B. 0x0000 AC81
C. 0xFFFF 81AC
D. 0xFFFF AC81
E. None of the above

Memory Address	Value
0x0000 0203	0x32
0x0000 0202	0x65
0x0000 0201	0x81
0x0000 0200	0xAC