CSC 3210 Computer organization and programming

Chunlan Gao

Georgia State University

Attendance Number

6

Attention!

- Every time when you send me message through email please including your full name, your session number.

Last Class

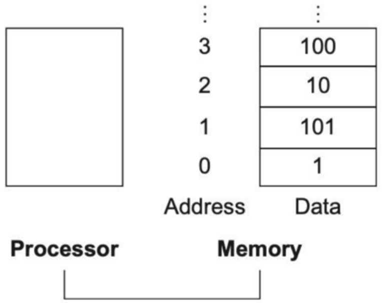
- 1. Instruction & Instruction set
- 2. Arithmetic Operations(add, sub, addi)
- 3. Registor operands
- 4. Memory Operands

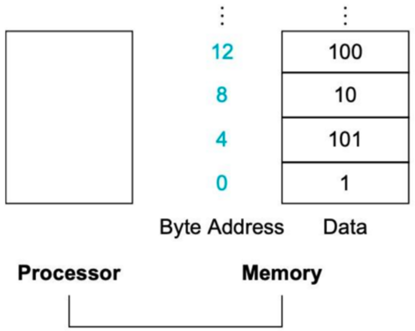
Memory Operands

- Memory is byte addressed : Each address identifies an 8-bit byte

- C code:

  `A[12] = h + A[8];`
  - h in x21, base address of A in x22
- Compiled RISC-V code:
  - Index 8 requires offset of 32
    - 4 bytes per word






```
lw      x9, 32(x22)  \\Temporary reg x9 gets A[8]
add     x9, x21, x9   \\Temporary reg x9 gets h + A[8]
sw      x9, 48(x22)    \\  Stores h + A[8] back into A[12]
```

Little Endian: Least-significant byte at least address of a word

instructions

Category	Instruction	Example	Meaning	Comments
Arithmetic	Add	`add x5, x6, x7`	`x5 = x6 + x7`	Three register operands; add
	Subtract	`sub x5, x6, x7`	`x5 = x6 - x7`	Three register operands; subtract
	Add immediate	`addi x5, x6, 20`	`x5 = x6 + 20`	Used to add constants
Data transfer	Load word	`lw x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Word from memory to register
	Load word, unsigned	`lwu x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Unsigned word from memory to register
	Store word	`sw x5, 40(x6)`	`Memory[x6 + 40] = x5`	Word from register to memory
	Load halfword	`lh x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Halfword from memory to register
	Load halfword, unsigned	`lhu x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Unsigned halfword from memory to register
	Store halfword	`sh x5, 40(x6)`	`Memory[x6 + 40] = x5`	Halfword from register to memory
	Load byte	`lb x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Byte from memory to register
	Load byte, unsigned	`lbu x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Byte unsigned from memory to register
	Store byte	`sb x5, 40(x6)`	`Memory[x6 + 40] = x5`	Byte from register to memory
	Load reserved	`lr.d x5, (x6)`	`x5 = Memory[x6]`	Load; 1st half of atomic swap
	Store conditional	`sc.d x7, x5, (x6)`	`Memory[x6] = x5; x7 = 0/1`	Store; 2nd half of atomic swap
	Load upper immediate	`lui x5, 0x12345`	`x5 = 0x12345000`	Loads 20-bit constant shifted left 12 bits
Logical	And	`and x5, x6, x7`	`x5 = x6 & x7`	Three reg. operands; bit-by-bit AND
	Inclusive or	`or x5, x6, x8`	`x5 = x6 | x8`	Three reg. operands; bit-by-bit OR
	Exclusive or	`xor x5, x6, x9`	`x5 = x6 ^ x9`	Three reg. operands; bit-by-bit XOR
	And immediate	`andi x5, x6, 20`	`x5 = x6 & 20`	Bit-by-bit AND reg. with constant
	Inclusive or immediate	`ori x5, x6, 20`	`x5 = x6 | 20`	Bit-by-bit OR reg. with constant
	Exclusive or immediate	`xori x5, x6, 20`	`x5 = x6 ^ 20`	Bit-by-bit XOR reg. with constant
Shift	Shift left logical	`sll x5, x6, x7`	`x5 = x6 << x7`	Shift left by register
	Shift right logical	`srl x5, x6, x7`	`x5 = x6 >> x7`	Shift right by register
	Shift right arithmetic	`sra x5, x6, x7`	`x5 = x6 >> x7`	Arithmetic shift right by register
	Shift left logical immediate	`slli x5, x6, 3`	`x5 = x6 << 3`	Shift left by immediate
	Shift right logical immediate	`srli x5, x6, 3`	`x5 = x6 >> 3`	Shift right by immediate
	Shift right arithmetic immediate	`srai x5, x6, 3`	`x5 = x6 >> 3`	Arithmetic shift right by immediate

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!

Data Representation: Numbering System

- Assembly language deals with **Data at the physical level**

  so you need to examine **registers** and **memory**

- **Binary** and **hexadecimal** numbers are commonly used to describe those contents (other systems used as well)

- **Need to learn how to translate from one format to another**

Table 1-2 Binary, Octal, Decimal, and Hexadecimal Digits.

System	Base	Possible Digits
Binary	2	0 1
Octal	8	0 1 2 3 4 5 6 7
Decimal	10	0 1 2 3 4 5 6 7 8 9
Hexadecimal	16	0 1 2 3 4 5 6 7 8 9 A B C D E F

Data Representation: Binary Numbers **(Integers)**

- Binary integers can be **signed** or **unsigned**.
  - **A signed** integer is **positive** or **negative**.
  - **An unsigned** integer is by default **positive**.

Data Representation: Binary Numbers **(Integers)**

- Base 2,
- Digits are 1 and 0
  - 1 = true
  - 0 = false
- **MSB** – most significant bit
- **LSB** – least significant bit

MSB LSB

1 0 1 1 0 0 1 0 1 0 0 1 1 1 0 0

15 0

Data Representation: Binary Numbers **(Integers)**

- Each digit (bit) is either 1 or 0
- Each bit represents **a power of 2:**

1	1	1	1	1	1	1	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

Every binary number

is a sum of powers of 2

Table 1-3 Binary Bit Position Values.

2^n	Decimal Value	2^n	Decimal Value
2^0	1	2^8	256
2^1	2	2^9	512
2^2	4	2^{10}	1024
2^3	8	2^{11}	2048
2^4	16	2^{12}	4096
2^5	32	2^{13}	8192
2^6	64	2^{14}	16384
2^7	128	2^{15}	32768

Unsigned Binary Integers **Binary to Decimal**

- In any number base, the value of i_{th} digit *d* is:
  - $d \times \text{base}^{I}$

Where i starts at 0 and increase from right to left

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

Example 10110_{two} represents :

$(1 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)$

$= 16 + 0 + 4 + 2 + 1$

$= 23_{\text{ten}}$

Data Representation: Translating **Unsigned Decimal to Binary**

- Repeatedly divide the decimal integer by 2.
- Each remainder is a binary digit in the translated value:

37 = 100101

Division	Quotient	Remainder
37 / 2	18	1
18 / 2	9	0
9 / 2	4	1
4 / 2	2	0
2 / 2	1	0
1 / 2	0	1