Chapter 2

Instructions: Language of the Computer

指令：计算机的语言

目录

- **机器语言——指令集**
  - **操作码、操作数（寄存器、内存地址、小常量）**
- 数据的二进制表示
  - 无符号数、有符号数（二进制补码）
- 指令的二进制表示
  - 算术逻辑、访存、控制、函数调用
- 其他表示
  - 字符串、常量、数组与指针
  - 例子
- 并行与同步指令
- 程序的编译与运行

What is language

- A language is a structured system of communication.
  - the structure：grammar
  - the free components：vocabulary

C/C++语言

- Language of C/C++
  - 算术逻辑运算符
    - 算术：+， -, *, /
    - 逻辑：&&, ||, !
    - 关系：>, <, ==, >=, <=, !=
  - 控制语句
    - 控制语句：if-else, for, while-do, do-while
    - 过程调用语句：f()
  - 数据读取
    - x = a

```
int fact(int n)
{
        if (n <= 1 )
                return 1;
        else
                return n * fact(n-1);
}
```

机器语言

- 汇编语言——机器语言的符号化
  - 算术逻辑指令
    - arithmetic
    - logical
  - 内存访问指令
    - data transfer
  - 控制指令
    - branch（conditional branch）
    - jump（unconditional） ...

Instruction class	RISC-V examples	HLL correspondence	Frequency	
			Integer	Fl. Pt.
Arithmetic	add, sub, addi	Operations in assignment statements	16%	48%
Data transfer	ld, sd, lw, sw, lh, sh, lb, sb, lui	References to data structures in memory	35%	36%
Logical	and, or, xor, sll, srl, sra	Operations in assignment statements	12%	4%
Branch	beq, bne, blt, bge, bltu, bgeu	*If* statements; loops	34%	8%
Jump	jal, jalr	Procedure calls & returns; *switch* statements	2%	0%

图2.41

- 在线编译工具（含arm、riscv、x86...)
  - godbolt.org

机器语言：Instruction Set

- The set of instructions of a computer (api接口?)
- Different computers have different instruction sets
  - But with many aspects in common
  - add x2, x3, x4   vs add x2, **x3**
- Early computers had very simple instruction sets
  - Simplified implementation
- Many modern computers also have simple instruction sets

The RISC-V Instruction Set

- Used as the example throughout the book
- Developed at UC Berkeley as open ISA
- Now managed by the RISC-V Foundation (riscv.org)
- Typical of many modern ISAs
  - See RISC-V Reference Data tear-out card
- Similar ISAs have a large share of embedded core market
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …
- Riscv tools, such as simulator
  - https://github.com/riscv-software-src

指令集设计的考虑

- 机器对外提供的API接口的集合
  - 既要便于组装大型、复杂软件
    - 方便编译器翻译
  - 又要便于硬件实现
    - 性能、能效、硬件成本
- 比如
  - 机器通常提供：add, sub, mul, div等指令
  - 需要提供sin、cos、exp等指令么?
  - 需要relu, sigmoid等指令吗?

Arithmetic Operations 算术操作

- Add and subtract, three operands
  - Two sources and one destination
  - Ex: **add a, b, c** // a gets b + c
- All arithmetic operations have this form
- Design Principle 1: Simplicity favours regularity
  - **Regularity** makes implementation simpler
  - **Simplicity** enables higher performance at lower cost
  - 规整 -> 简单 -> 易于理解、记忆、掌握 -> 决策
  - 复杂 -> 失控?

Arithmetic Example

- C code:

```
f = (g + h) - (i + j);
```

- Compiled RISC-V code (ignore operands):

```
add t0, g, h   // temp t0 = g + h
add t1, i, j   // temp t1 = i + j
sub f, t0, t1  // f = t0 - t1
```

为什么需要3条指令，而不是一条?

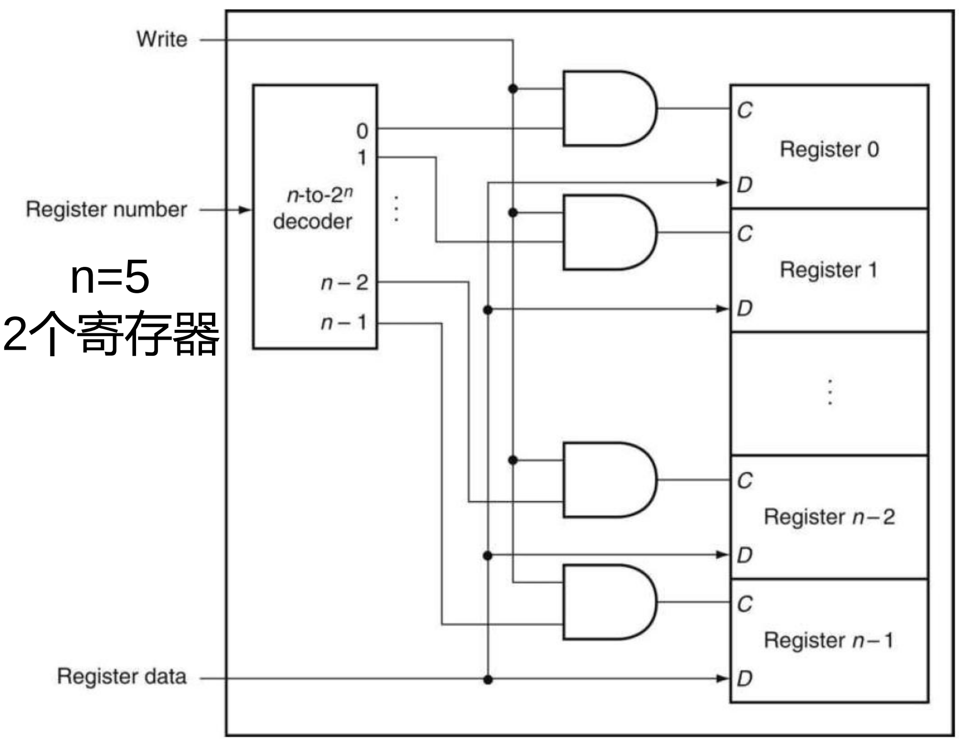
Register Operands 寄存器操作数

- Arithmetic instructions use register operands
- RISC-V has a 32 × 32-bit register file
  - Use for frequently accessed data
  - 32-bit data is called a “word”
    - 32 x 32-bit general purpose registers x0 to x31
  - 64-bit data is called a “doubleword”
  - 什么是寄存器?
  - 32个? 64位? 都是2的幂
- Design Principle 2: Smaller is faster
  - c.f. main memory: millions of locations

寄存器与触发器

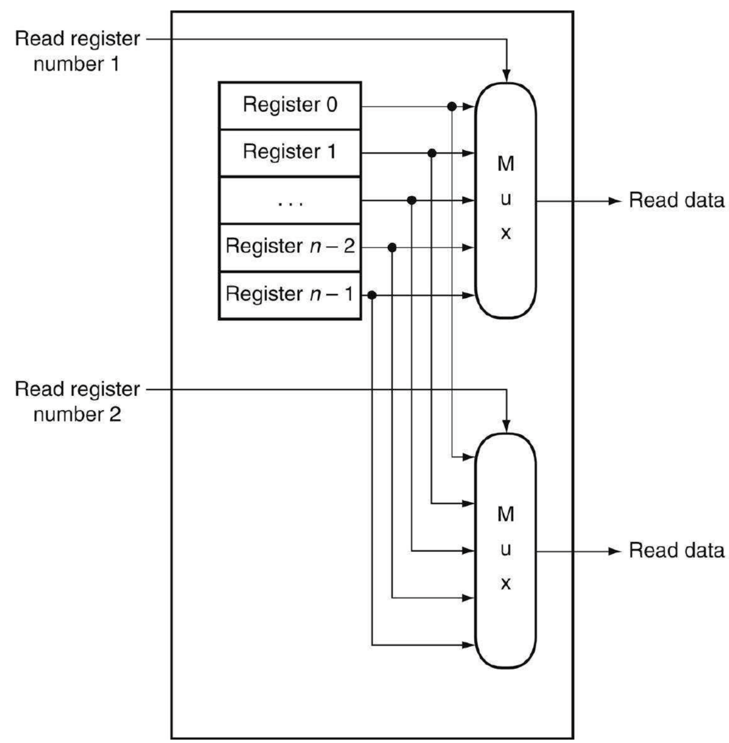
- D触发器、译码器、多路选择器

单一写端口



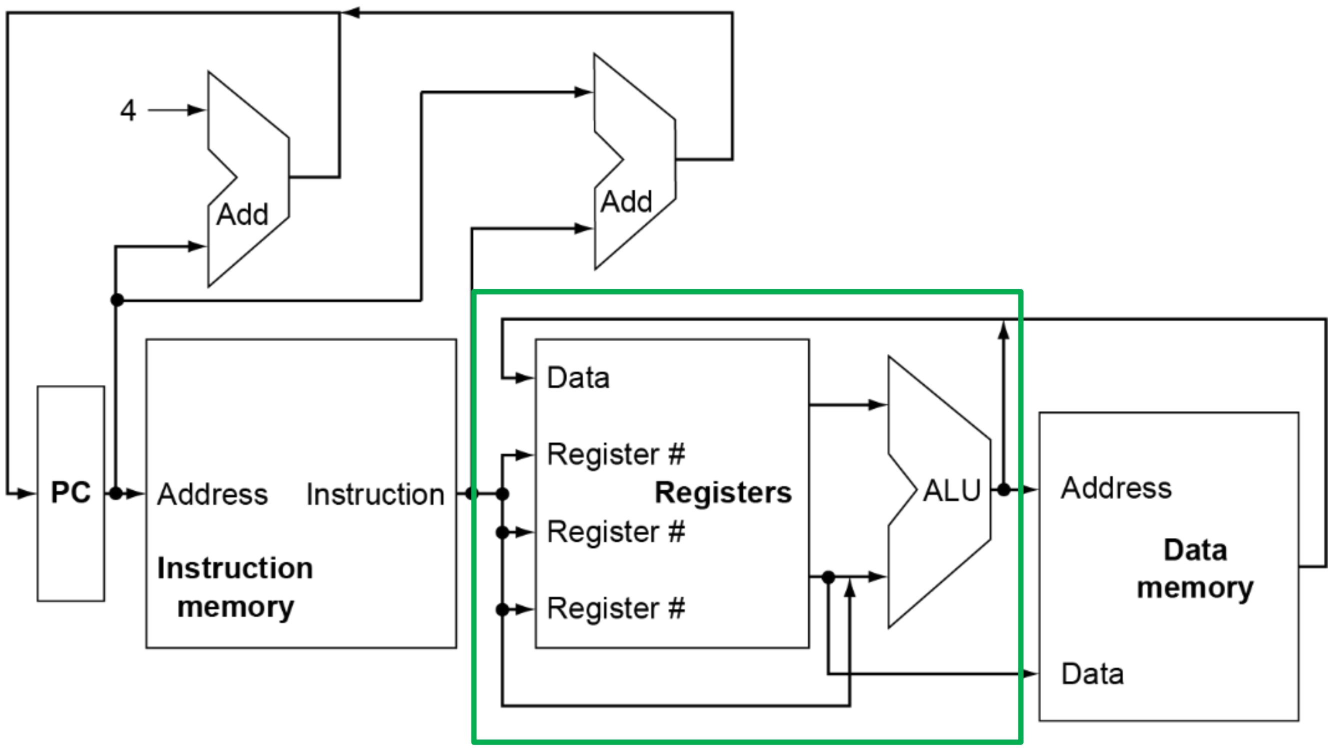
n=5
32个寄存器

两个读端口



寄存器在CPU中的位置

- 寄存器是最快的存储设备
  - 是不是越多越好?
  - 如果不够用，怎么办?



RISC-V Registers

- x0: 常量 0
- x1: 返回地址
- x2: 栈指针
- x3: global pointer
- x4: thread pointer
- x5 – x7, x28 – x31: 临时变量
- x8: frame pointer
- x9, x18 – x27: saved registers
- x10 – x11: 函数参数/返回值
- x12 – x17: 函数参数

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5–7	t0–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller

Register Operand Example 寄存器操作数

- C code:

```
f = (g + h) - (i + j);
```

  - f, …, j in x19, x20, …, x23

- Compiled RISC-V code:

```
add x5, x20, x21
add x6, x22, x23
sub x19, x5, x6
```

No-Op(空指令）

- A No-op is an instruction that does nothing...
  - Why? You may need to replace code later: No-ops can fill space, align data, and perform other options
- By ***convention*** RISC-V has a specific no-op instruction...
  - **add x0 x0 x0**
- Why?
  - **Writes to x0 are always ignored**...
    RISC-V uses that a lot as we will see in the **jump-and-link** operations (函数调用)
  - Making a "standard" no-op improves the disassembler and can potentially improve the processor
    - Special case the particular conventional no-op.

Memory Operands 内存操作数

- Main memory used for composite data
  - Arrays, structures, dynamic data
    - 能不能也放寄存器?
- Convention to apply arithmetic operations (not that in x86)
  - Load values from memory into **registers**
  - Compute the arithmetic by reading/writting **registers**
  - Store result from **register** to memory
- Memory is byte addressed
  - Each address identifies an 8-bit byte
  - 什么是内存泄漏?
    - 用**地址**表示的内存资源，即使以后不再使用，也没能及时回收，可用的**内存地址**越来越少，直到耗尽

大端模式 vs 小端模式

- 内存以字节为单位寻址，但是操作数可能是多字节的
- RISC-V is Little Endian
  - Least-significant byte at least address of a word
  - x86, ARM processors running Android, iOS, and Windows, RISC-V

- Big Endian
  - Most-significant byte at least address
  - Big Endian: Sun, PPC Mac, Internet

Byte Ordering Example 字节顺序的例子

- **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		

- 需要特别留意的场合
  - 以长类型(int)写入该值，以短类型（short/char）读出该值时，如何区分高位和低位

如何查看机器的字节顺序

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

```
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");
}
```

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

Memory Operand Example 内存操作数

- C code:

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

  - h in x21, base address of A in x22
  - A数组中每个元素占一个双字（即8个字节）

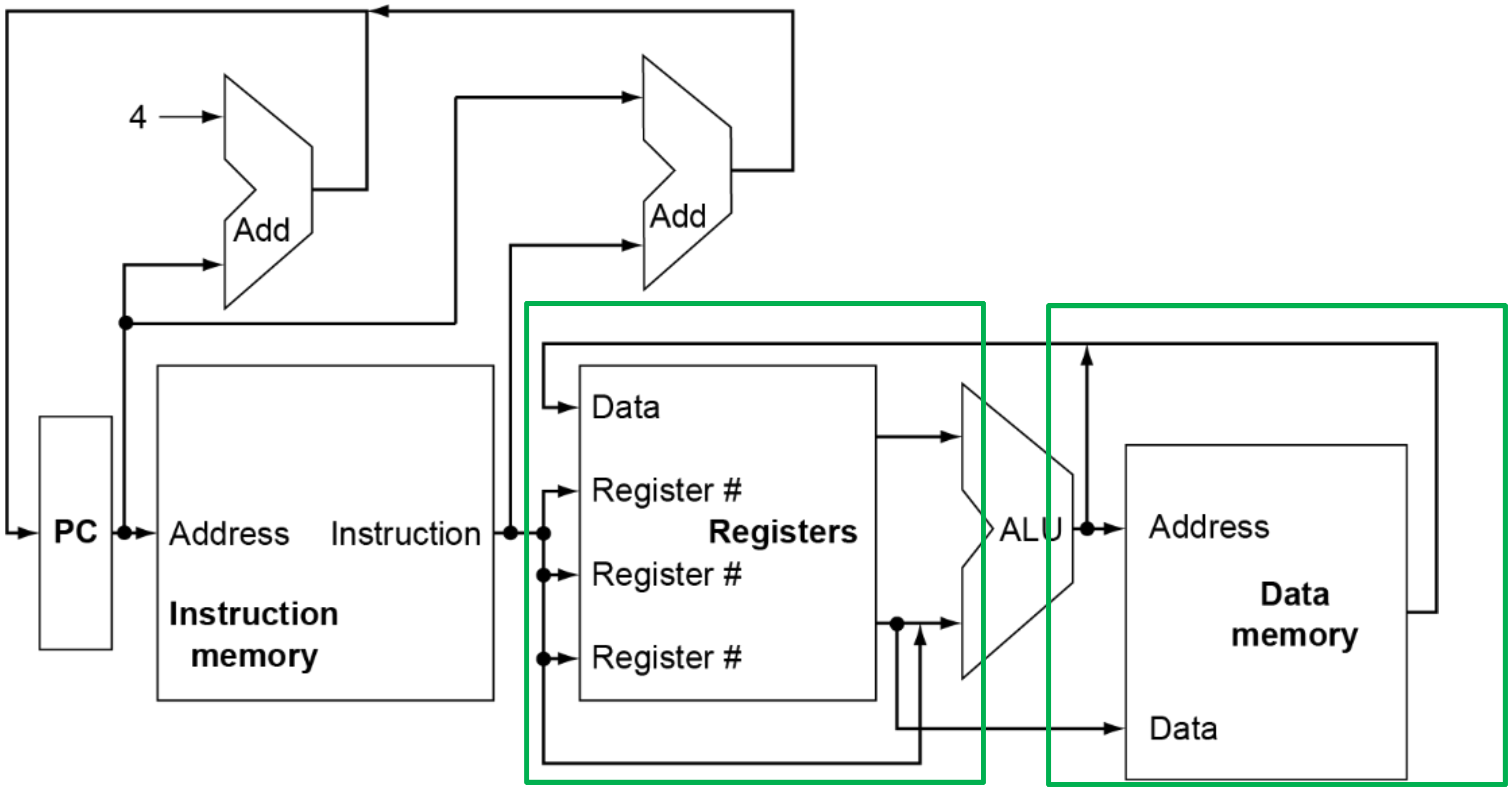
- Compiled RISC-V code:
  - composite data stored in memory
  - Index 8 requires offset of 64
  - ld & sd 指令
    - 8 bytes per doubleword

```
ld      x9, 64(x22)   ==》 为什么不是 8(x22)?
add     x9, x21, x9
sd      x9, 96(x22)
```

Registers vs. Memory 寄存器 vs 内存

- Registers are faster to access than memory
  - 200 X speedup
  - 10,000 X power efficiency
- Operating on memory data requires loads and stores
  - More instructions to be executed, much slower
- Compiler must use registers for variables as much as possible
  - 编译器负责C/C++变量映射到寄存器或内存的分配方案
  - Only spill to memory for less frequently used variables
  - Register optimization is important!
    - what is register optimization?

CPU一瞥



4
Add
Add
PC
Address
Instruction
Instruction memory
Data
Register #
Registers
Register #
Register #
ALU
Address
Data memory
Data

Immediate Operands 立即数操作数

- Constant data specified in an instruction 指令内编码的常量
  - `addi x22, x22, 4`
  - `the most popular instruction in most programs`
- No need of subi instruction
  - Just use a negative constant
    - `addi $s1, $s1, -4  <==> subi $s1, $s1, 4`
- 回忆：常量0
- Make the common case fast
    - Small constants are common
      - more than 50% arithmetic insts in SPEC CPU2006
    - Immediate avoids a load instruction, better than
      - ~~load x9， addr_of_consant4(x3)~~
      - ~~add x22, x22, x9~~

The Constant Zero

- RISC-V register x0 is the constant 0
  - Cannot be overwritten
- Useful for common operations
  - E.g., negate a value in a register by sub zero by the value
  - `sub x9, x0, x8`
- Make the common case fast

Summary小结

- Instruction set as APIs of a CPU
- Operations 运算符:
  - API operators：add, sub;
    - like insert(), size(), sort() in C/C++
  - how many?
- Operands 操作数:
  - API arguments: 寄存器、内存、常量
    - like 函数参数 in C/C++
  - how many?
- compared to C/C++ APIs
  - correspondance
  - why different choice between C and assembly?

数据与指令的二进制表示

- The 0/1 represenation of APIs
  - operators: instruction
  - operands: data

目录

- **机器语言——指令集**
  - **操作码、操作数（寄存器、内存地址、小常量)**
- **数据的二进制表示**
  - **无符号数、有符号数（二进制补码）**
- 指令的二进制表示
  - 算术逻辑、访存、控制、函数调用
- 其他表示
  - 字符串、常量、数组与指针
  - 例子
- 并行与同步指令
- 程序的编译与运行

Unsigned Binary Integers 无符号二进制整数

- Given an n-bit number

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

- Range: 0 to $+2^n - 1$
  - 何时会溢出?
- Example
  - $0000\ 0000\ \ldots\ 0000\ 1011_2$
    $= 0 + \ldots + \mathbf{1{\times}2^3} + 0{\times}2^2 + \mathbf{1{\times}2^1} + \mathbf{1{\times}2^0}$
    $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$

怎么表示负数?

- sign and magnitude（原码表示法）
  - 最高位专门用来表示+/-符号，其余参照无符号
    - 0：正数
    - 1：负数
  - 缺点
    - 算术运算时比较慢（需要额外步骤来设置符号位）
    - +0和-0
- 2s-Complement Signed Integers（补码表示法）
    - 正数、负数的范围不对等，但是速度快，且减法可以用加法器实现
    - 只需要检测最高位就可以判断正负

2s-Complement Signed Integers

- Given an n-bit number

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

- Range: -2^{n-1} to $+2^{n-1} - 1$
- Example：32位
  - $1111\ 1111\ \ldots\ 1111\ 1100_2$
    $= -1\times2^{31} + 1\times2^{30} + \ldots + 1\times2^2 + 0\times2^1 + 0\times2^0$
    $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$

2s-Complement Signed Integers

- Bit 31 is sign bit
  - 1 for negative numbers
  - 0 for non-negative numbers
  - 最大正数：$2^{n-1}-1$
  - 最小负数：-2^{n-1}
  - imbalanced maxium 负数和正数
- 补码表示非负数时，跟无符号表示一样
- Some specific numbers
  - 0: 0000 0000 … 0000
  - –1: 1111 1111 … 1111
  - Most-negative: 1000 0000 … 0000
  - Most-positive: 0111 1111 … 1111

Mapping Signed ↔ Unsigned

同一个bit pattern，表示unsigned或signed的差值是多少？

How to prove？
符号位的变化..

Bits	Signed	Unsigned
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	-8	8
1001	-7	9
1010	-6	10
1011	-5	11
1100	-4	12
1101	-3	13
1110	-2	14
1111	-1	15

=

+/- 16

补码的相反数

- Complement and add 1 （按位取反，+1）
  - Complement means 1 → 0, 0 → 1
  - 证明正确性

$$x + \overline{x} = 1111...111_2 = -1$$

$$\overline{x} + 1 = -x$$

- Example: negate +2
  - $+2 = 0000\ 0000\ \ldots\ 0010_{two}$
  - $-2 = 1111\ 1111\ \ldots\ 1101_{two} + 1$
    $= 1111\ 1111\ \ldots\ 1110_{two}$

Sign Extension符号扩展

- Representing a number using more bits
  - Preserve the numeric value
- 通过复制符号位来扩展——保持值不变
  - c.f. unsigned values: extend with 0s
  - 如何证明正确性?
- Examples: 4-bit to 8-bit
  - +2: 0010 => 0000 0010
  - –2: 1110 => 1111 1110

- In RISC-V instruction set
  - `lb`: sign-extend loaded byte
  - `lbu`: zero-extend loaded byte

说明：从内存加载到寄存器（32位）时，寄存器总是要填满4个字节

Hexadecimal 十六进制

- Base 16
  - Compact representation of bit strings 二进制序列的紧凑表示法
  - 4 bits per hex digit 十六进制中的1位可以表示二进制中的4位

0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

- Example: 0xeca8 6420
  - 1110 1100 1010 1000 0110 0100 0010 0000
  - 在指令中区分16进制和寄存器：add x10, x10, 0x10

- C++里面怎么识别16进制表示?

Exercise 1

```
addi x11, x0, 0x3f5   #将常量0x3f5存入x11
sw x11, 0(x5)         #将x11中的值存储到内存地址(x5)
lb x12, 1(x5)         #从内存地址(x5)+1读取一个byte
```

What’s the value in x12?

A. 0x5
B. 0xf
C. 0x3
D. 0xffffffff

考点:

小端模式、符号扩展

Exercise 1

```
addi x11, x0, 0x3f5   #将常量0x3f5存入x11
sw x11, 0(x5)         #将x11中的值存储到内存地址(x5)
lb x12, 1(x5)         #从内存地址(x5)+1读取一个byte
```

What’s the value in x12?

A. 0x5
B. 0xf
C. 0x3
D. 0xffffffff

考点:

小端模式、符号扩展

Exercise 2

addi x11, x0, 0x80f5 #将常量0x80f5存入x11
sw x11, 0(x5) #将x11的值存储到内存地址(x5)
lb x12, 1(x5)   #从内存地址(x5)+1读取一个byte

What's the value in x12?

A. 0x80
B. 0xf5
C. 0x0
D. 0xffff ff80

Exercise 2

addi x11, x0, 0x80f5 #将常量0x80f5存入x11
sw x11, 0(x5) #将x11的值存储到内存地址(x5)
lb x12, 1(x5)   #从内存地址(x5)+1读取一个byte

What's the value in x12?

A. 0x80
B. 0xf5
C. 0x0
D. 0xffff ff80