CSC 3210 Computer organization and programming

Chunlan Gao

GeorgiaState University

Attendance Number

7

Unsigned Binary Integers **Binary to Decimal**

Example 10110_{two} represents :

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

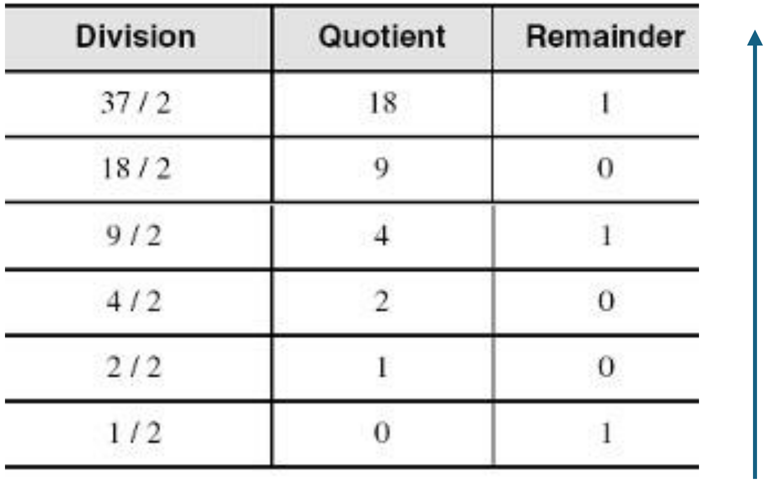
$$= 16 + 0 + 4 + 2 + 1$$

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

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

Data Representation: Binary **Addition**

- Starting with the **LSB**, add each pair of digits, **include the carry if present**.

carry:					1				
	0	0	0	0	0	1	0	0	(4)
+	0	0	0	0	0	1	1	1	(7)
	0	0	0	0	1	0	1	1	(11)
bit position:	7	6	5	4	3	2	1	0	

In RISC-V: Word

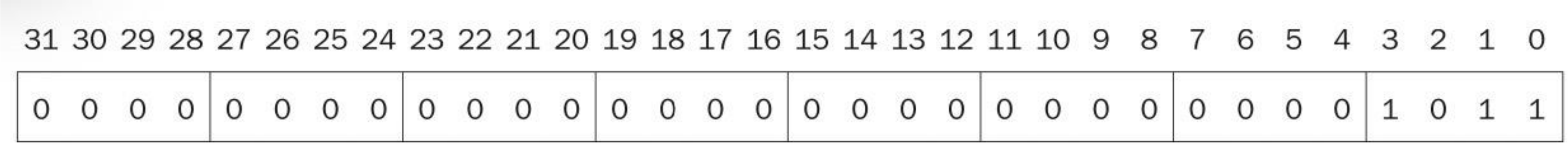
Bit: 1 or 0

Byte： 8 bits

Word： 4 Byte （32 bits）

Doubleword： 2 word, 8 byte, 64 bits

The placement of the number 1011$_{two}$:



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1

(32 bits wide)

The RISC-V word is 32 bits long, so it can represent 2^{32} different 32-bit patterns and represent the numbers from 0 to 2^{32}-1.

Negative Numbers

Sometimes we also use negative numbers.

1. Most obvious solution: add a separate sign, which we call *sign and magnitude*

shortcomings:

- ➢Where to put the sign bit
- ➢adder for sign and magnitude may need an extra step to set the sign
- ➢a separate sign bit means has both a negative and a positive zero

Solution: no better alternative

- Final solution:
- pick the representation that made the hardware simple: leading 0s mean positive, and leading 1s mean negative.

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
  - $1111\ 1111 \ldots 1111\ 1100_2$
    $= -1\times2^{31} + 1\times2^{30} + \ldots + 1\times2^2 + 0\times2^1 + 0\times2^0$
    $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$
- Using 64 bits: −9,223,372,036,854,775,808
  to 9,223,372,036,854,775,807

2s-Complement Signed Integers

- Bit 63 is signed bit
  - 1 for negative numbers
  - 0 for non-negative numbers
- –(–2^{n-1}) can’t be represented
- Non-negative numbers have the same unsigned and 2s-complement representation
- Some specific numbers
  - 0: 0000 0000 … 0000
  - –1: 1111 1111 … 1111
  - Most-negative: 1000 0000 … 0000
  - Most-positive: 0111 1111 … 1111

Signed Negation

- Example: negate +2

$$x + \overline{x} = 1111\ldots111_2 = -1$$

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

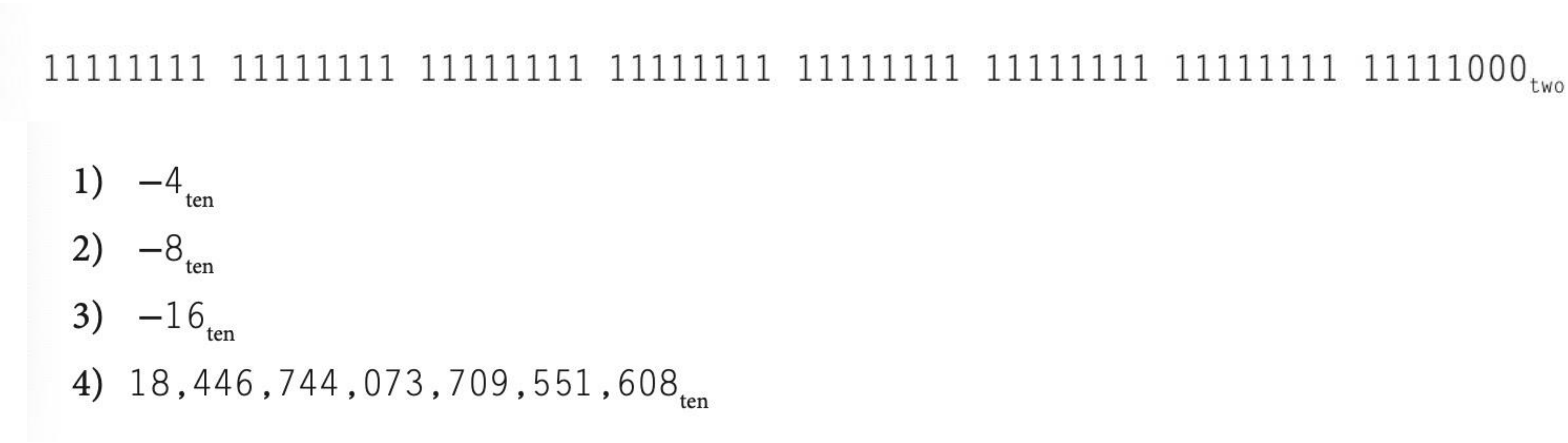
$+2 = 0000\ 0000\ \ldots\ 0010_{\text{two}}$

$-2 = 1111\ 1111\ \ldots\ 1101_{\text{two}} + 1$

$= 1111\ 1111\ \ldots\ 1110_{\text{two}}$

Question !

- What is the decimal value of this 64-bit two's complement number?



$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111000_{two}$

1) -4_{ten}
2) -8_{ten}
3) -16_{ten}
4) $18{,}446{,}744{,}073{,}709{,}551{,}608_{ten}$

Sign Extension

- Representing a number using more bits
  - Preserve the numeric value
- Replicate the sign bit to the left
  - c.f. unsigned values: extend with 0s
- Examples: 8-bit to 16-bit
  - +2: 0000 0010 =>
    0000 0000 0000 0010
  - –2: 1111 1110 =>
  - 1111 1111 1111 1110

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

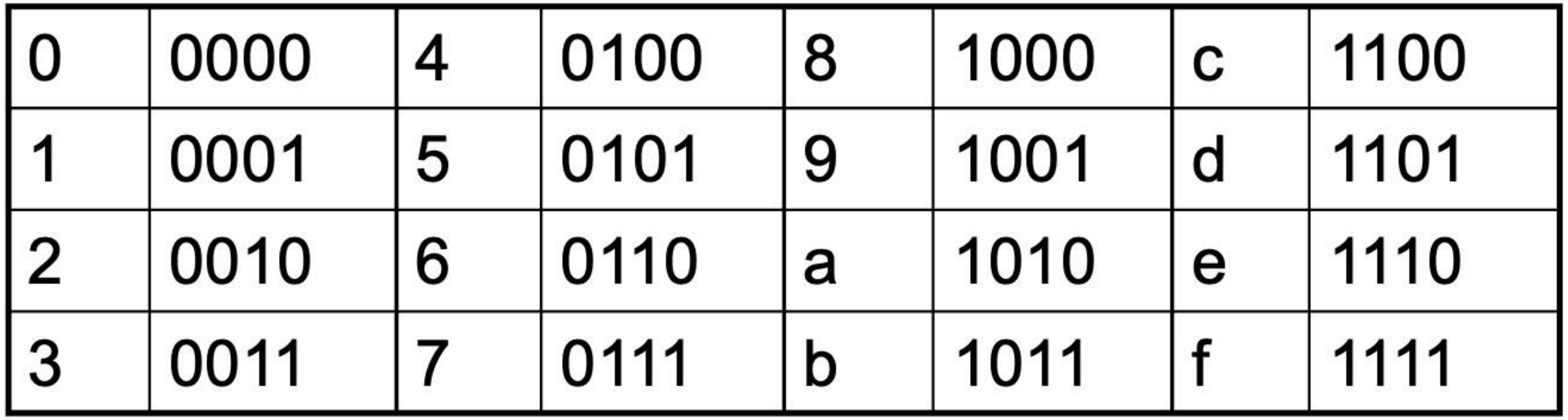
2.5 Representing Instructions

- Instructions are encoded in binary
  - Called machine code

- RISC-V instructions
  - Encoded as 32-bit instruction words
  - Small number of formats encoding operation code (opcode), register numbers, …
  - Regularity!

Hexadecimal

- Base 16
  - Compact representation of bit strings
  - 4 bits per hex digit



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: eca8 6420
  - 1110 1100 1010 1000 0110 0100 0010 0000

RISC-V R-format Instructions

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

- Instruction fields
  - opcode: operation code
  - rd: destination register number
  - funct3: 3-bit function code (additional opcode)
  - rs1: the first source register number
  - rs2: the second source register number
  - funct7: 7-bit function code (additional opcode)

R-format Example

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

- `add x9,x20,x21`

0	21	20	0	9	51

0000000	10101	10100	000	01001	0110011

0000 0001 0101 1010 0000 0100 1011 0011$_{\text{two}}$ =
015A04B3$_{16}$

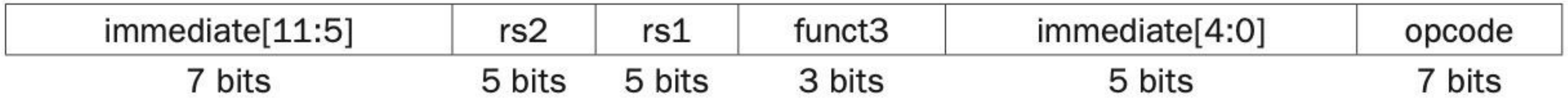
RISC-V I-format Instructions

immediate	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits

- Immediate arithmetic and load instructions
  - rs1: source or base address register number
  - immediate: constant operand, or offset added to base address
    - 2s-complement, sign extended
- *Design Principle 3:* Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible

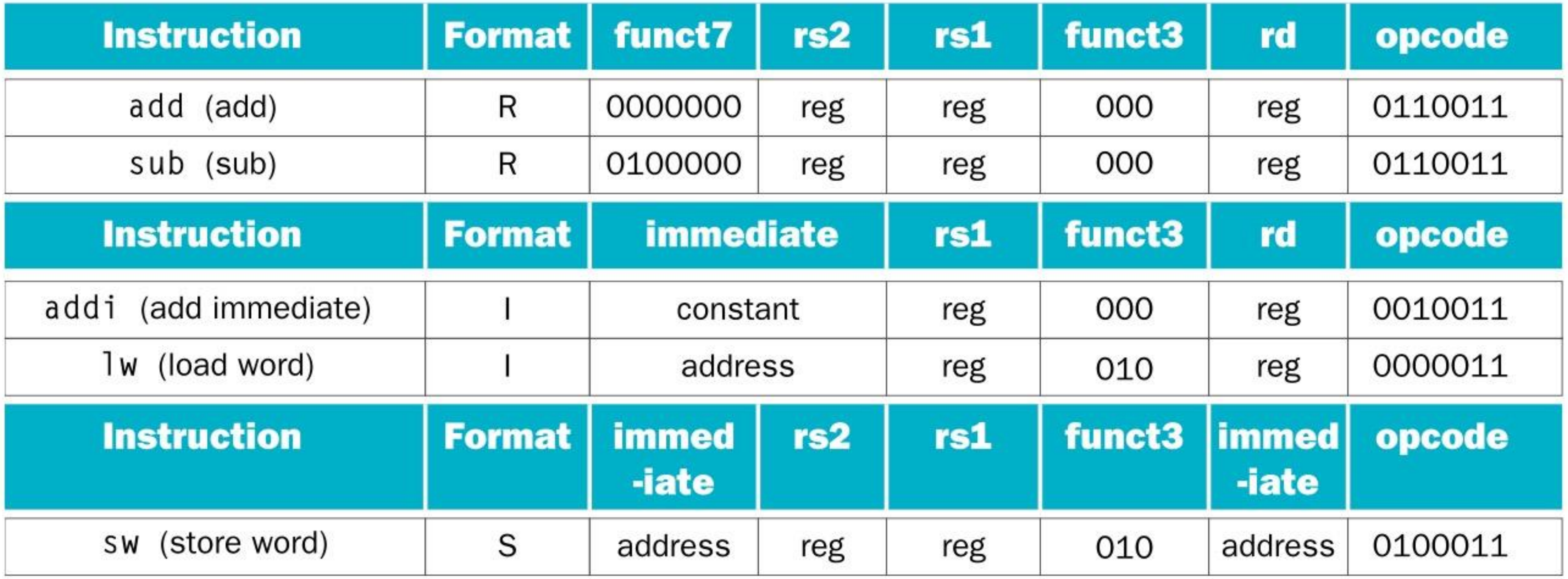
RISC-V S-format Instructions

- Different immediate format for store instructions
  - rs1: base address register number
  - rs2: source operand register number
  - immediate: offset added to base address
    - Split so that rs1 and rs2 fields always in the same place



immediate[11:5]	rs2	rs1	funct3	immediate[4:0]	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

Compare



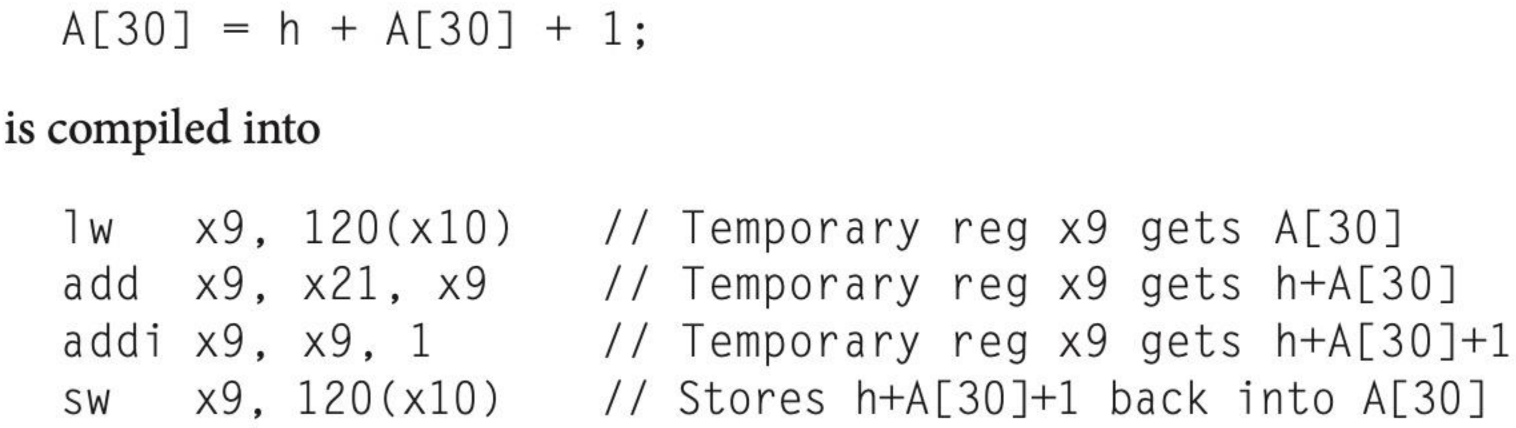
Instruction	Format	funct7	rs2	rs1	funct3	rd	opcode
add (add)	R	0000000	reg	reg	000	reg	0110011
sub (sub)	R	0100000	reg	reg	000	reg	0110011

Instruction	Format	immediate	rs1	funct3	rd	opcode
addi (add immediate)	I	constant	reg	000	reg	0010011
lw (load word)	I	address	reg	010	reg	0000011

Instruction	Format	immed -iate	rs2	rs1	funct3	immed -iate	opcode
sw (store word)	S	address	reg	reg	010	address	0100011

FIGURE 2.5 RISC-V instruction encoding. In the table above, "reg" means a register number between 0 and 31 and "address" means a 12-bit address or constant. The funct3 and funct7 fields act as additional opcode fields.

Example:



```
A[30] = h + A[30] + 1;
```

is compiled into

```
lw   x9, 120(x10)    // Temporary reg x9 gets A[30]
add  x9, x21, x9     // Temporary reg x9 gets h+A[30]
addi x9, x9, 1       // Temporary reg x9 gets h+A[30]+1
sw   x9, 120(x10)    // Stores h+A[30]+1 back into A[30]
```

immediate	rs1	funct3	rd	opcode
120	10	2	9	3

funct7	rs2	rs1	funct3	rd	opcode
0	9	21	0	9	51

immediate	rs1	funct3	rd	opcode
1	9	0	9	19

immediate[11:5]	rs2	rs1	funct3	immediate[4:0]	opcode
3	9	10	2	24	35

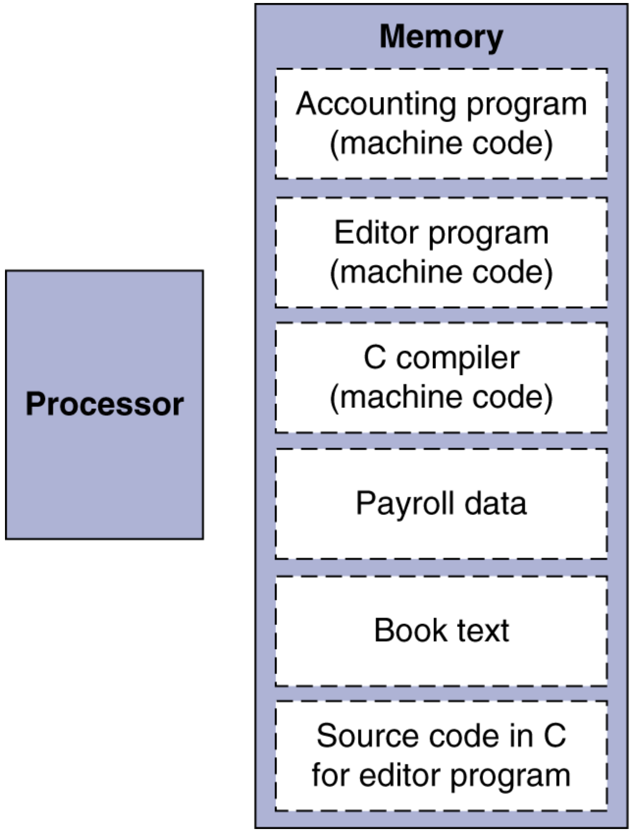
immediate	rs1	funct3	rd	opcode
000011110000	01010	010	01001	0000011

funct7	rs2	rs1	funct3	rd	opcode
0000000	01001	10101	000	01001	0110011

immediate	rs1	funct3	rd	opcode
000000000001	01001	000	01001	0010011

immediate[11:5]	rs2	rs1	funct3	immediate[4:0]	opcode
0000011	01001	01010	010	11000	0100011

Stored Program Computers


Memory
Accounting program (machine code)
Editor program (machine code)
C compiler (machine code)
Payroll data
Book text
Source code in C for editor program
Processor

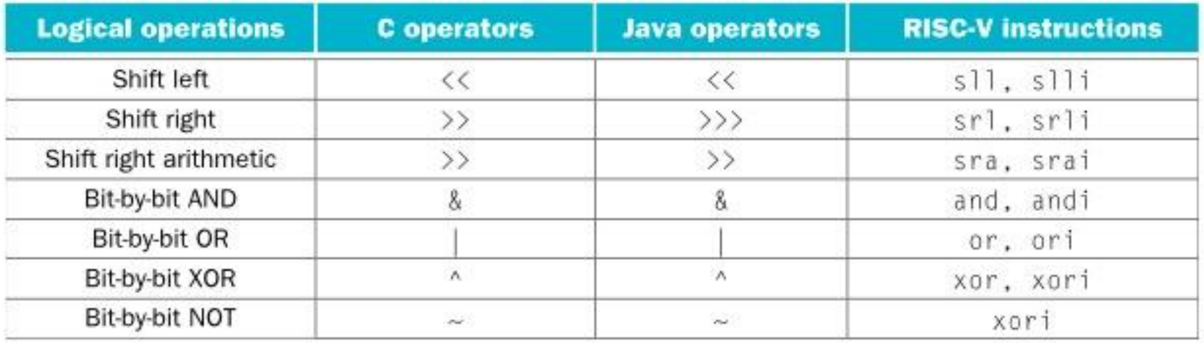
- Instructions represented in binary, just like data

Instructions and data stored in memory

- Programs can operate on programs
  - e.g., compilers, linkers, …
- Binary compatibility allows compiled programs to work on different computers
  - ❖Standardized ISAs

Logical Operations

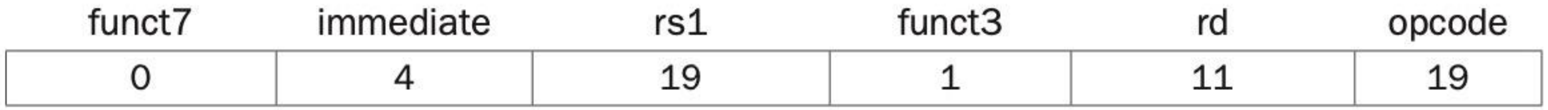
- Instructions for bitwise manipulation



Logical operations	C operators	Java operators	RISC-V instructions
Shift left	<<	<<	`sll`, `slli`
Shift right	>>	>>>	`srl`, `srli`
Shift right arithmetic	>>	>>	`sra`, `srai`
Bit-by-bit AND	&	&	`and`, `andi`
Bit-by-bit OR	|	|	`or`, `ori`
Bit-by-bit XOR	^	^	`xor`, `xori`
Bit-by-bit NOT	~	~	`xori`

FIGURE 2.8 C and Java logical operators and their corresponding RISC-V instructions. One way to implement NOT is to use XOR with one operand being all ones (FFFF FFFF FFFF FFFF$_{\text{hex}}$).

Shift Operations(I type)



funct7	immediate	rs1	funct3	rd	opcode
0	4	19	1	11	19

immed: how many positions to shift

Shift left logical:

- Shift left and fill with 0 bits
- `slli` by *i* bits multiplies by 2^i

Shift right logical:

- Shift right and fill with 0 bits
- `srli` by *i* bits divides by 2^i (unsigned only)

Example

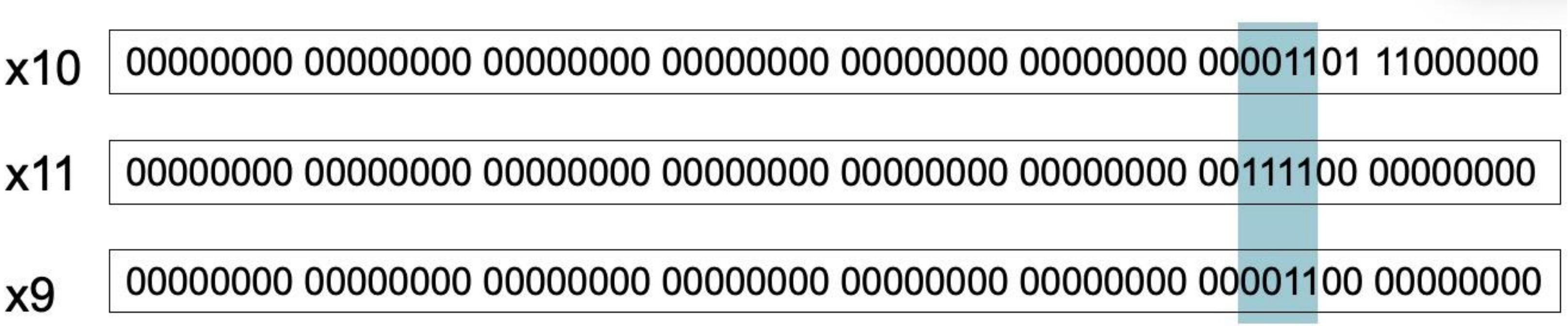
slli x11, x19, 4

if register x19 contained
00000000 00000000 00000000 00001111_{two} = 15_{ten}

and the instruction to shift left by 4 was executed, the new value would be:
00000000 00000000 00000000 11110000two = 240_{ten}

AND Operations

- Useful to mask bits in a word
  - Select some bits, clear others to 0b
  - `and x9,x10,x11`

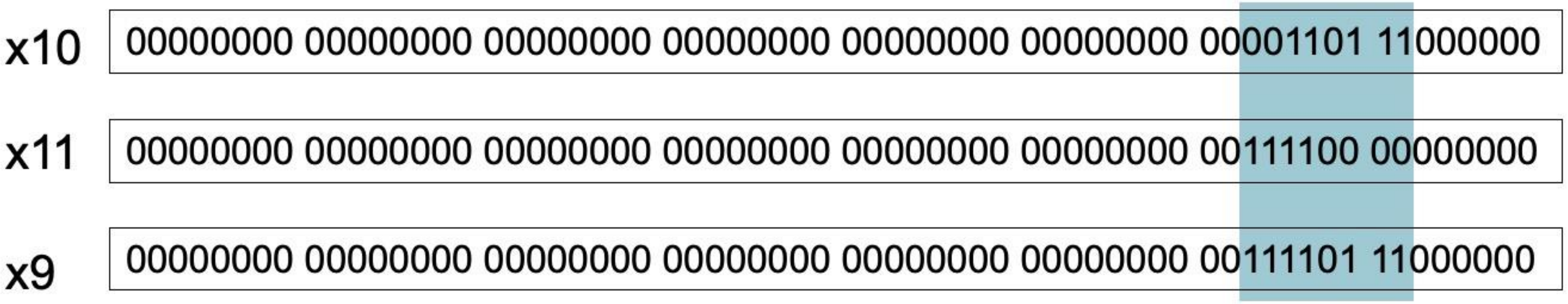


x10	00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000
x11	00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000
x9	00000000 00000000 00000000 00000000 00000000 00000000 00001100 00000000

OR Operations

- Useful to include bits in a word
  - Set some bits to 1, leave others unchanged

```
or x9,x10,x11
```



x10 00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000

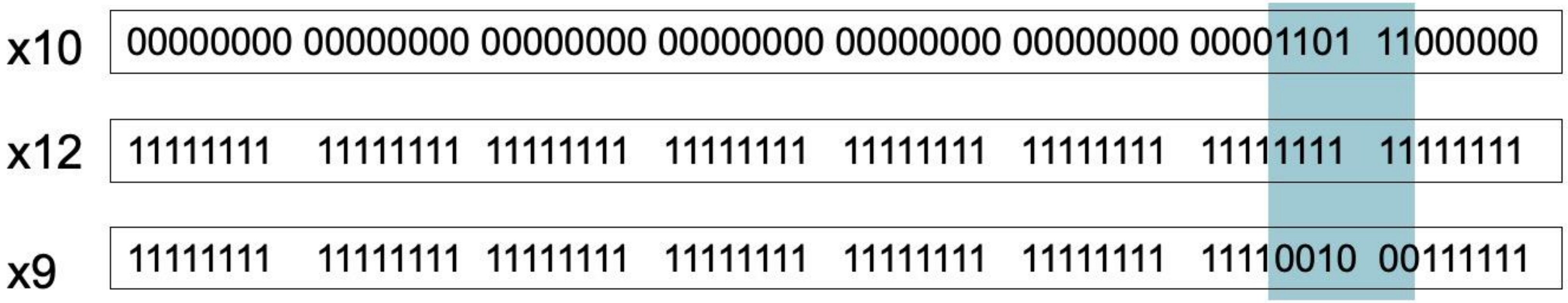
x11 00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000

x9 00000000 00000000 00000000 00000000 00000000 00000000 00111101 11000000

XOR Operations

- Differencing operation
  - Set some bits to 1, leave others unchanged

```
xor x9,x10,x12  // NOT operation
```



x10	00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000
x12	11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111
x9	11111111 11111111 11111111 11111111 11111111 11111111 11110010 00111111

Instructions for Making Decisions

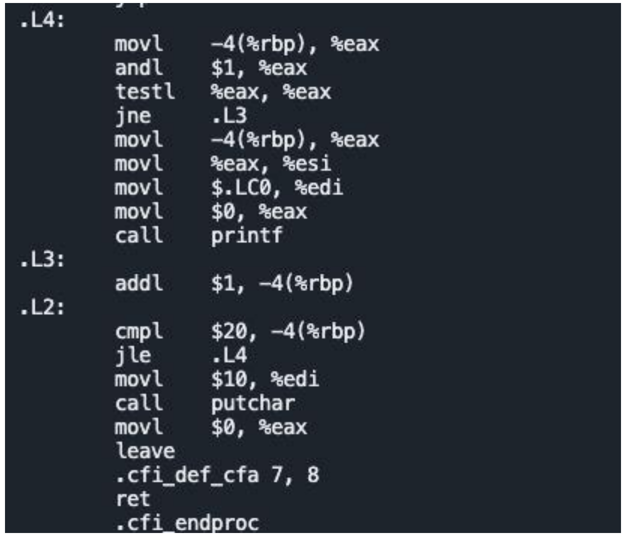
- What distinguishes a computer from a simple calculator is its ability to make decisions. Based on the input data and the values created during computation, different instructions execute.

Conditional Operations

- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially

- `beq rs1, rs2, L1`
  - if (rs1 == rs2) branch to instruction labeled L1

- `bne rs1, rs2, L2`
  - if (rs1 != rs2) branch to instruction labeled L2



```
.L4:
        movl    -4(%rbp), %eax
        andl    $1, %eax
        testl   %eax, %eax
        jne     .L3
        movl    -4(%rbp), %eax
        movl    %eax, %esi
        movl    $.LC0, %edi
        movl    $0, %eax
        call    printf
.L3:
        addl    $1, -4(%rbp)
.L2:
        cmpl    $20, -4(%rbp)
        jle     .L4
        movl    $10, %edi
        call    putchar
        movl    $0, %eax
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
```

Compiling if-then-else into Conditional Branches

- C code:

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

five variables f through j correspond to the five registers x19 through x23

- Compiled RISC-V code:

```
      bne x22, x23, Else
      add x19, x20, x21
      beq x0,x0,Exit // unconditional
Else: sub x19, x20, x21
Exit:
```

Compiling Loop Statements

- C code:

```
while (save[i] == k) i += 1;
```

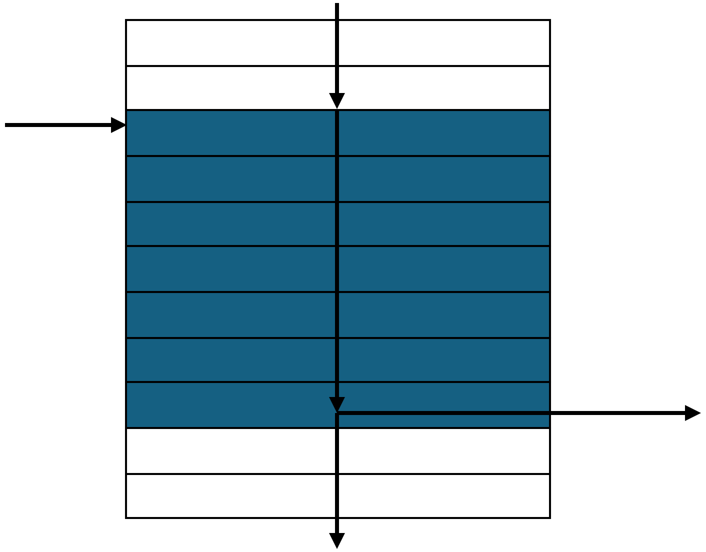
  - i in x22, k in x24, address of save in x25

- Compiled RISC-V code:

```
Loop: slli x10, x22, 3
      add  x10, x10, x25
      ld   x9, 0(x10)
      bne  x9, x24, Exit
       addi x22, x22, 1
      beq  x0, x0, Loop
Exit: …
```

Basic Blocks

- A basic block is a sequence of instructions with
  - No embedded branches (except at end)
  - No branch targets (except at beginning)



A compiler identifies basic blocks for optimization.
An advanced processor can accelerate execution of basic blocks.

More Conditional Operations

- `blt rs1, rs2, L1`
  - if (rs1 < rs2) branch to instruction labeled L1
- `bge rs1, rs2, L1`
  - if (rs1 >= rs2) branch to instruction labeled L1
- Example
  - if (a > b) a += 1;
  - a in x22, b in x23

```
      bge  x23, x22, Exit      // branch if b >= a
      addi x22, x22, 1
Exit:
```

Signed vs. Unsigned

- Signed comparison: blt, bge
- Unsigned comparison: bltu, bgeu
- Example
  - x22 = 1111 1111 1111 1111 1111 1111 1111 1111
  - x23 = 0000 0000 0000 0000 0000 0000 0000 0001
  - `x22 < x23 // signed`
    - –1 < +1
  - `x22 > x23 // unsigned`
    - +4,294,967,295 > +1