浙江大學
ZHEJIANG UNIVERSITY

系统结构与网络安全研究所

# 计算机组成与设计

# Computer Organization & Design

## The Hardware/Software Interface

# Chapter 2

# Instructions: Language of the Machine

林 芃
Lin Peng
penglin@zju.edu.cn

# The process of compiling

## High-level programming language

- Notations more closer to the natural language ex. Y = A + B
- The compiler translates into assembly language
- Advantages over assembly language
  - Think in a more natural language
  - Programs can be independent of hardware

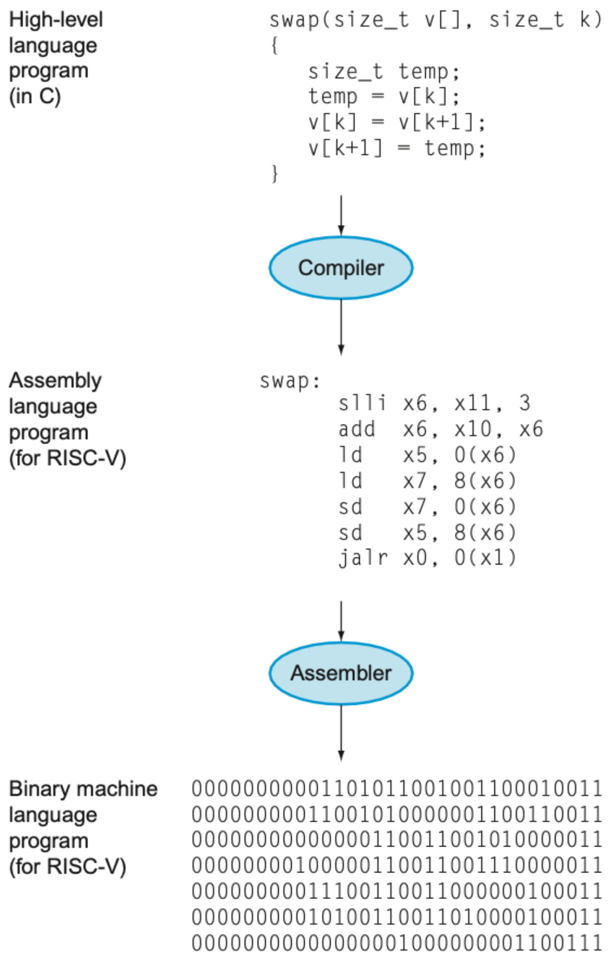
## Assembly language

- Symbolic notations ex. add Y, A, B
- The assembler translates them into machine instruction

## Machine language

- Computers only understands electrical signals on/off
- Binary numbers express machine instructions

  ex. 1000110010100000 means to add two numbers


High-level language program (in C)
Compiler
Assembly language program (for RISC-V)
Assembler
Binary machine language program (for RISC-V)

```
swap(size_t v[], size_t k)
{
   size_t temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

```
swap:
      slli x6, x11, 3
      add  x6, x10, x6
      ld   x5, 0(x6)
      ld   x7, 8(x6)
      sd   x7, 0(x6)
      sd   x5, 8(x6)
      jalr x0, 0(x1)
```

```
00000000001101011001001100010011
00000000011001010000001100110011
00000000000000110011001010000011
00000000100000110011001110000011
00000000011100110011000000100011
00000000010100110011010000100011
00000000000000001000000001100111
```

# Outline

- Introduction
- Operations of the computer hardware
- Operands of the computer hardware
- Signed and a numbers
- Representing instructions in the computer
- Logical operations
- Instructions for making decision
- Supporting procedures in computer hardware
- Instruction addressing

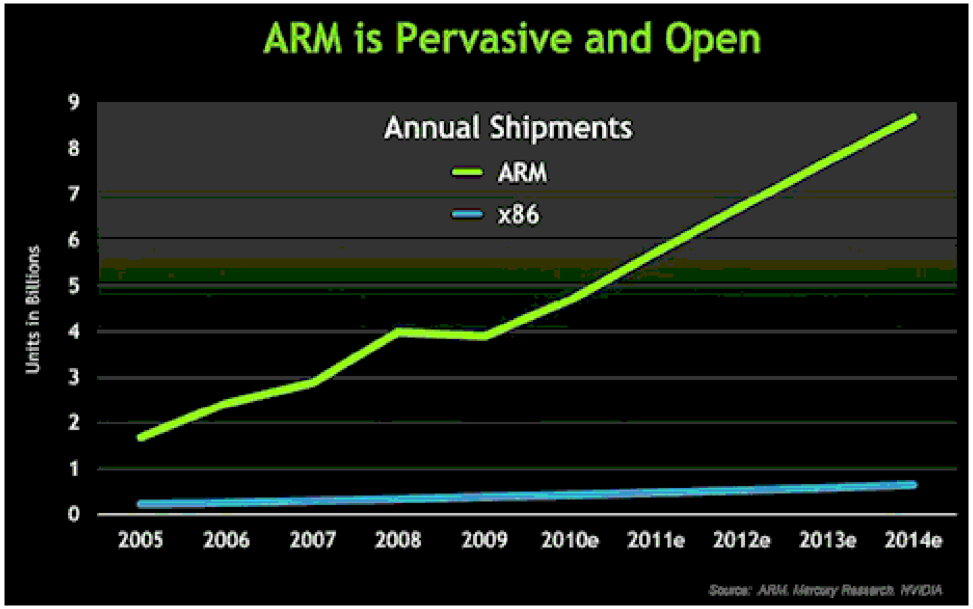
## 2.1 Introduction

- **Language of the machine**
  - Instructions → Word
  - Instruction set → Vocabulary
- **Design goals**
  - Maximize performance
  - Minimize cost
  - Reduce design time
- **Our chosen instruction set: RISC-V**

# Instruction Set

- Different computers have different instruction sets
- But with many aspects in common
- Early computers had very simple instruction sets
- Simplified implementation
- Many modern computers also have simple instruction sets

RISC vs CISC



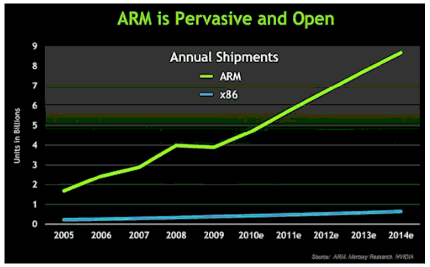
ZHEJIANG UNIVERSITY

## CISC vs RISC

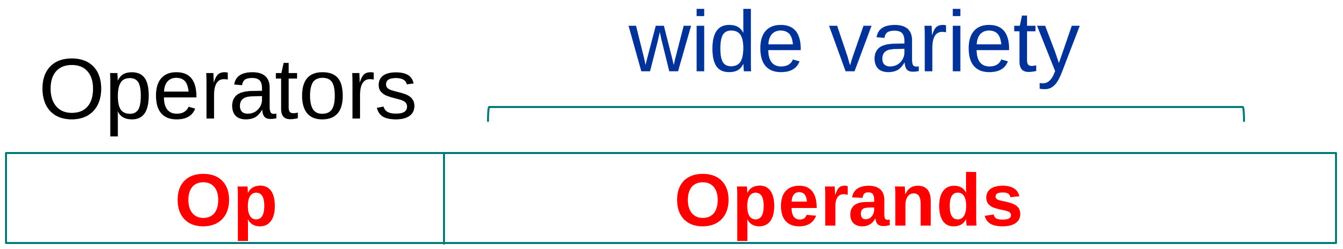
| **CISC**<br>complex instruction set computer | **RISC**<br>reduce instruction set computer |
|---|---|
| Emphasis on hardware (>200 instructions) | Emphasis on software (<100 instructions) |
| Multiple instruction sizes and formats | Instructions of same size with few formats |
| Less registers | Uses more registers |
| More addressing modes | Fewer addressing modes (Load/Store) |
| Extensive use of microprogramming | Complexity in compiler |
| Instructions take a varying amount of cycle time | Instructions take one cycle time |
| Pipelining is difficult | Pipelining is easy |

## The RISC-V Instruction Set

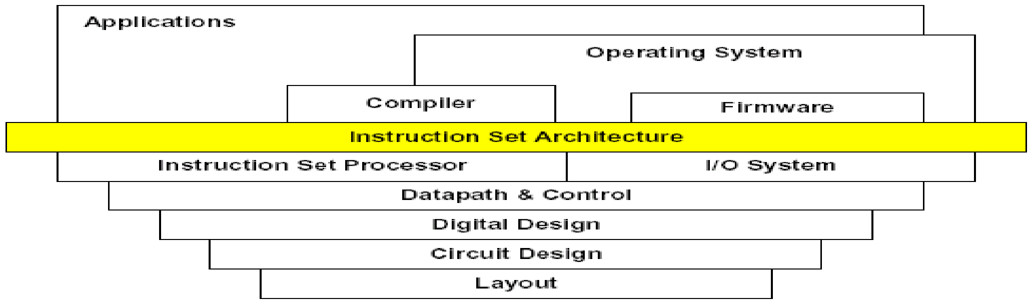
- Developed at UC Berkeley as open ISA (since 2010)
- Now managed by the RISC-V Foundation (riscv.org)
- Typical modern ISAs
  - See RISC-V Reference Data tear-out card
- Similar ISAs have a large share of embedded core market
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …



## Instruction formats



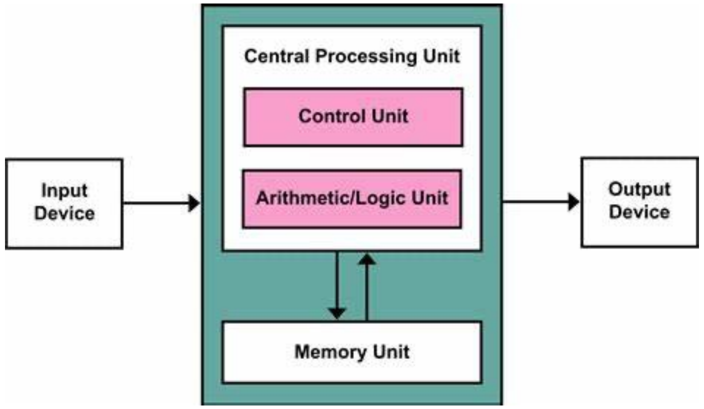
- Type of internal storage in processer
- The number of the memory operand in the instruction
- Operations in the instruction Set
- Type and Size of Operands
- Representing Instructions in the Computer
  - Encoding



# Stored-program concept

- Today's computers are built on 2 key principles: (Stored-program concept)
  - ①Instruction are represented as numbers.
  - ②Programs can be stored in memory to be read or written just like numbers.

Von Neumann' Computer



## 2.2 Arithmetic Operations

- **Every computer should perform arithmetic**
  - Only one operation per instruction
  - Add and subtract, three operands
    (Two sources and one destination)
    add a, b, c // a gets b + c
- **All arithmetic operations have this form**
- **Design Principle 1:** *Simplicity favors regularity*
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

# Arithmetic Example

- **Example 2.2 Compiling a complex C statement**
  - C code:

    f = ( g + h ) – ( i + j );

  - Compiled RISC-V code:

```
add  t0, g, h        # temporary variable t0 contains g + h
add  t1, i, j        # temporary variable t1 contains i + j
sub  f, t0, t1       # f gets t0 – t1
```

**RISC-V assembly language**

| Category | Instruction | Example | Meaning | Comments |
|---|---|---|---|---|
| Arithmetic | add | add a,b,c | a←b+c | Always three operand |
| | subtract | sub a,b,c | a←b-c | Always three operand |

## 2.3 Operands

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

# RISC-V Registers

- Lumped Model
  - C only
  - RC model

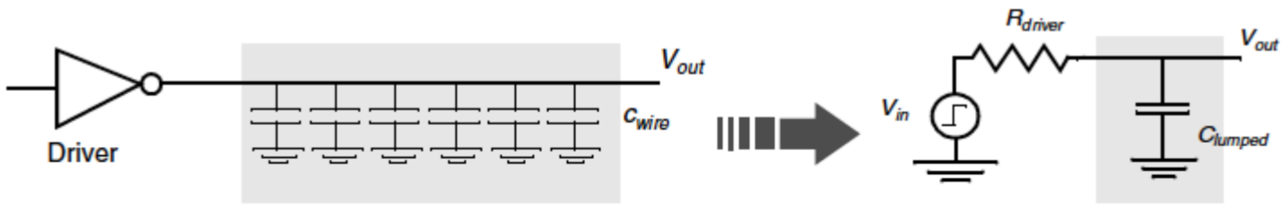


**Figure 4.11** Distributed versus lumped capacitance model of wire. $C_{lumped} = L \times c_{wire}$, with $L$ the length of the wire and $c_{wire}$ the capacitance per unit length. The driver is modeled as a voltage source and a source resistance $R_{driver}$.

$$C_{lumped}\frac{dV_{out}}{dt} + \frac{V_{out} - V_{in}}{R_{driver}} = 0$$

$$\tau = R_{driver} \times C_{lumped},$$

$$V_{out}(t) = (1 - e^{-t/\tau})\ V$$

$$t_{50\%} = 0.69 \times 10\ \text{K}\Omega \times 11\ \text{pF} = 76\ \text{nsec}$$
$$t_{90\%} = 2.2 \times 10\ \text{K}\Omega \times 11\ \text{pF} = 242\ \text{nsec}$$

# RISC-V Registers

| Name | Register Name | Usage | Preserved On Call? |
|---|---|---|---|
| x0 | 0 | The constant value 0 | n.a. |
| x1(ra) | 1 | Return address(link register) | yes |
| x2(sp) | 2 | Stack pointer | yes |
| x3(gp) | 3 | Global pointer | yes |
| x4(tp) | 4 | Thread pointer | yes |
| x5-x7 | 5-7 | Temporaries | no |
| x8-x9 | 8-9 | Saved | yes |
| x10-x17 | 10-17 | Arguments/results | no |
| x18-x27 | 18-27 | Saved | yes |
| x28-x31 | 28-31 | Temporaries | no |

# Register Operand Example

- **C code:**

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

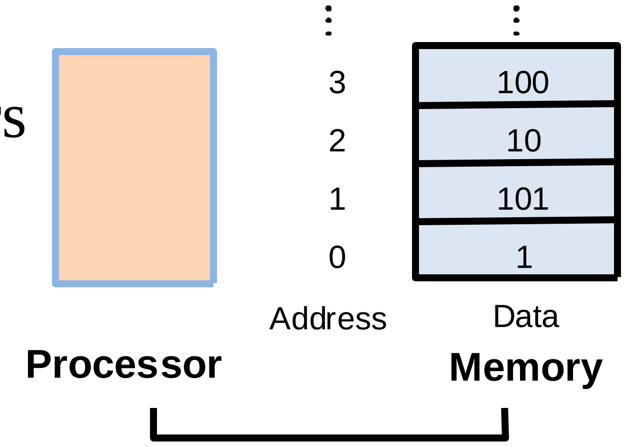
  - f, …, j in x19, x20, …, x23

- **Compiled RISC-V code:**

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

# Memory Operands

- **Main memory used for composite data**
  - Arrays, structures, dynamic data
- **To apply arithmetic operations**
  - Load values from memory into registers
  - Store result from register to memory
- **Memory is byte addressed**
  - Each address identifies an 8-bit byte
- **RISC-V is Little Endian**
  - Least-significant byte at least address of a word
  - *c.f.* Big Endian: most-significant byte at least address
- **RISC-V does not require words to be aligned in memory**
  - Unlike some other ISAs



# Memory Alignment

```
struct {
        int a;
        char b;
        char c[2];
        char d[3]
        float e;
}
```

Correct

| | | | |
|---|---|---|---|
| e | | | |
| d[1] | d[2] | No use | No use |
| b | c[0] | c[1] | d[0] |
| a | | | |

Wrong

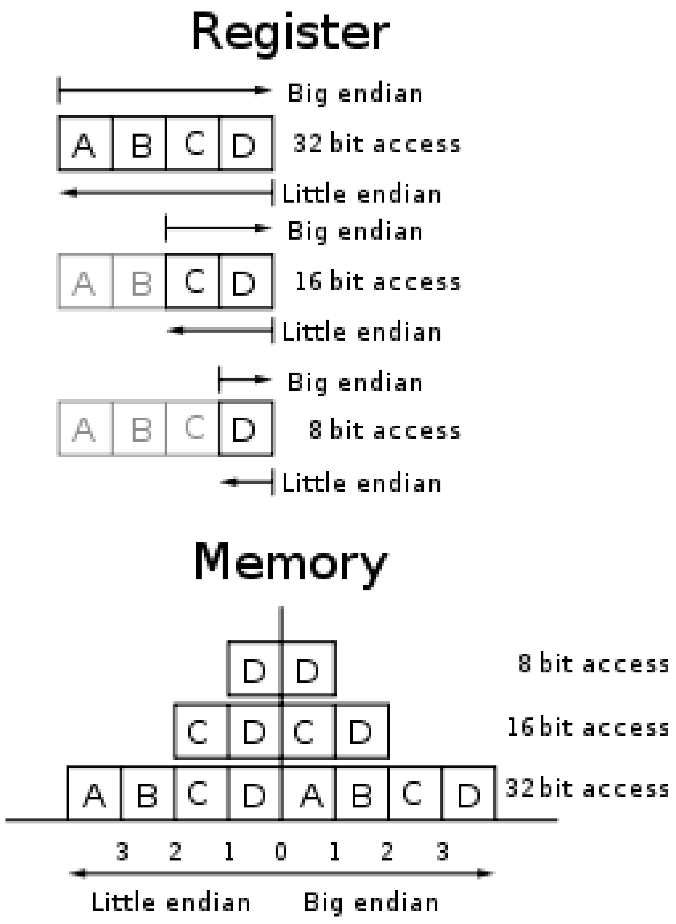
| | | | |
|---|---|---|---|
| e | | No use | No use |
| d[1] | d[2] | e | |
| b | c[0] | c[1] | d[0] |
| a | | | |

因为一次只能读出4字节内存中的一行

这样布局，e变量不能一次读出

# Endianness/byte order

- **Big end: Leftmost**
  - PowerPC
  - 01 02 = 513
- **Little end: Rightmost**
  - RISC-V
  - 01 02 = 258
- **Bi-endian**
  - MIPS, ARM , Alpha, SPARC



# Memory Operand Example

- **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 64
    - 8 bytes per doubleword
  - Offset: the constant in a data transfer instruction
  - Base register:  the register added to form the address

```
ld          x9, 64(x22)
add         x9, x21, x9
sd          x9, 96(x22)
```

## Register 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!

## Discussion：How to represent?

### Constant

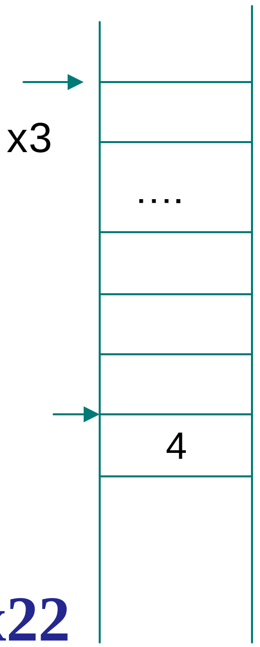
g = h + 55

Many time a program
will use a constant in an operation

# Constant or immediate Operands

- **Many time a program will use a constant in an operation**
  - Incrementing index to point to next element of array
  - Add the constant 4 to register x9
  - Assuming AddrConstants 4 is address pointer of constant 4

```
ld      x9, AddrConstant4(x3)    // x9=constant 4
add     x22, x22, x9
```



- **Immediate: Other method for adding constant 4 to x22**
  - Avoids the load instruction
  - Offer versions of the instruction

    **addi   x22, x22, 4    // x22= x22+ 4**
  - Constant zero: a register x0

- **Design Principle 3: Make common case fast**

# Brief summary

## RISC-V operands

| Name | Example | Comments |
|---|---|---|
| 32 register | x0~x31 | Fast locations for data. In RISC-V, data must be in registers to perform arithmetic. Register x0 always equals 0. |
| $2^{61}$ memory (double) words | Memory[0], Memory[8] , …… , Memory[18446744073709551608] | Accessed only by data transfer instructions. RISC-V uses byte addresses, so sequential doubleword accesses differ by 8. Memory holds data structures, arrays, and spilled registers. |

## RISC-V assembly language

| Category | Instruction | Example | Meaning | Comments |
|---|---|---|---|---|
| **Arithmetic** | add | add x5,x6,x7 | x5=x6 + x7 | Add two source register operands |
| | subtract | sub x5,x6,x7 | x5=x6 - x7 | First source register subtracts second one |
| | add immediate | addi x5,x6,20 | x5=x6+20 | Used to add constants |
| **Data transfer** | load doubleword | ld x5, 40(x6) | x5=Memory[x6+40] | doubleword from memory to register |
| | store doubleword | sd x5, 40(x6) | Memory[x6+40]=x5 | doubleword from register to memory |

# 2.4 Signed and unsigned numbers

- Bits are just bits (no inherent meaning): conventions define relationship between bits and numbers

- Binary numbers (base 2)
  0000 0001 0010 0011 0100 0101 0110 0111 1000 1001...
  decimal: 0...$2^n$-1

- Of course it gets more complicated:
  - numbers are finite (overflow)
  - fractions and real numbers
  - negative numbers

- How do we represent negative numbers?
  - which bit patterns will represent which numbers?

# Unsigned Binary 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: 0 to $+2^n - 1$
- Example
  - $0000\ 0000 \ldots 0000\ 1011_2$
    $= 0 + \ldots + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$
    $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$
- Using 64 bits: 0 to +18,446,774,073,709,551,615

# 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 sign 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

- **Complement and add 1**
  - Complement means 1 → 0, 0 → 1

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

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

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

# Sign Extension

- 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

符号位扩展 对符号位进行复制

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

# 2.5 Representing Instructions

- All information in computer consists of binary bits
- Instructions are encoded in binary
  - Called machine code
- Mapping registers into numbers
  - map registers x0 to x31 onto registers 0 to 31
- RISC-V instructions
  - Encoded as 32-bit instruction words
  - Small number of formats encoding operation code (opcode), register numbers, …
  - Regularity

# Example: Translating Assembly Code

- **(p81) Translating assembly into machine instruction**

RISC-V code

```
add    x9, x20, x21
```

Decimal version of machine code

| | | | | | |
|---|---|---|---|---|---|
| 0 | 21 | 20 | 0 | 9 | 51 |

Binary version of machine code

| | | | | | |
|---|---|---|---|---|---|
| 0000000 | 10101 | 10100 | 000 | 01001 | 0110011 |
| 7 bits | 5 bits | 5 bits | 3 bits | 5 bits | 7 bits |

# 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)
- **Design Principle 3**
  - ***Good design demands good compromises***
- **All instructions in RISC-V have the same length**
  - Conflict: same length ←--→ single instruction

# 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
- **Example：ld x9, 64(x22)**
  - 22 (x22) is placed in rs1;
  - 64 is placed in immediate
  - 9 (x9) is placed in rd

# RISC-V S-Format Instructions

| imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | opcode |
|---|---|---|---|---|---|
| 7 bits | 5 bits | 5 bits | 3 bits | 5 bits | 7 bits |

- **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

- **Example:** sd x9, 64(x22)
  - 22 (x22) is placed rs1;
  - 64 is placed immediate
  - 9 (x9) is placed rs2

# RISC-V instruction encoding

| Name | Format | Example | | | | | | Comment |
|---|---|---|---|---|---|---|---|---|
| **add** | R | 0 | 3 | 2 | 0 | 1 | 51 | add x1, x2, x3 |
| **sub** | R | 32 | 3 | 2 | 0 | 1 | 51 | sub x1, x2, x3 |
| **addi** | I | 1000 | | 2 | 0 | 1 | 19 | addi x1,x2,1000 |
| **ld** | I | 1000 | | 2 | 3 | 1 | 3 | ld x1, 1000(x2) |
| **sd** | S | 31 | 1 | 2 | 3 | 8 | 35 | sd x1, 1000(x2) |

# Example

## Example(p85) Translating assembly into machine instruction

### C code:

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

( Assume: h ---- x21        base address of A ---- x10 )

- RISC-V assembly code:

```
ld    x9, 240(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
sd    x9, 240(x10)        // stores h + A[30] + 1  back into A[30]
```

- RISC-V machine language code:
  - Decimal version

| | immediate | rs1 | funct3 | rd | opcode |
|---|---|---|---|---|---|
| ld | 240 | 10 | 3 | 9 | 3 |

| add | funct7 | rs2 | rs1 | funct3 | rd | opcode |
|---|---|---|---|---|---|---|
| | 0 | 9 | 21 | 0 | 9 | 51 |

| addi | immediate | rs1 | funct3 | rd | opcode |
|---|---|---|---|---|---|
| | 1 | 9 | 0 | 9 | 19 |

| sd | im[11:5] | rs2 | rs1 | funct3 | im[4:0] | opcode |
|---|---|---|---|---|---|---|
| | 7 | 9 | 10 | 3 | 16 | 35 |

- **Two key principles of today's computers**
  - Instructions are represented as numbers
  - Programs can be stored in memory like numbers

Stored-program

# RISC-V fields (format)

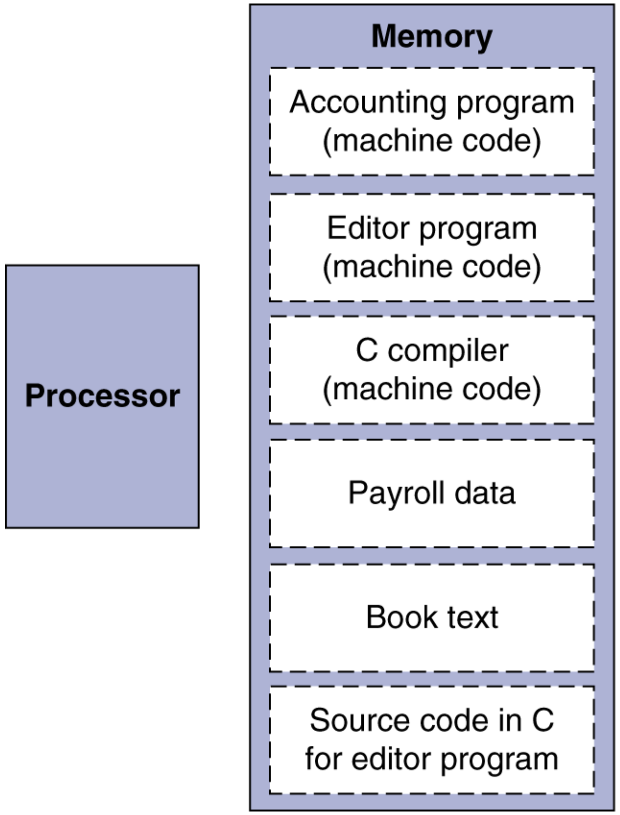
Imm Region: $\pm 2^{11}$

| Name | Fields | | | | | | Comments |
|---|---|---|---|---|---|---|---|
| **Field size** | 31 *7bits* 25 | 24 *5bits* 20 | 19 *5bits* 15 | 14 *3bits* 12 | 11 *5bits* 7 | 6 *7bits* 0 | All RISC-V instruction 32 bits |
| R-type | **funct7** | **rs2** | **rs1** | **funct3** | **rd** | **opcode** | Arithmetic instruction format |
| I-type | immediate[11:0] | | **rs1** | **funct3** | **rd** | **opcode** | Loads & immediate arithmetic |
| S-type | immed[11:5] | rs2 | **rs1** | **funct3** | immed[4:0] | **opcode** | Stores |
| SB-type | *imm[12,10:5]* | rs2 | **rs1** | **funct3** | *imm[4:1,11]* | **opcode** | Conditional branch format |
| UJ-type | *immediate[20,10:1,11,19:12]* | | | | **rd** | **opcode** | Unconditional jump format |
| U-type | immediate[31:12] | | | | **rd** | **opcode** | Upper immediate format |

**Must bear in mind !**

# Stored Program Computer

The BIG Picture



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

# 2.6 Logical Operations

- Instructions for bitwise manipulation

| Operation | C | Java | RISC-V |
|---|---|---|---|
| Shift left | << | << | `slli` |
| Shift right | >> | >>> | `srli` |
| Bit-by-bit AND | & | & | `and, andi` |
| Bit-by-bit OR | \| | \| | `or, ori` |
| Bit-by-bit XOR | ^ | ^ | `xor, xori` |
| Bit-by-bit NOT | ~ | ~ | |

- Useful for extracting and inserting groups of bits in a word

# Shift Operations

| funct6 | immed | rs1 | funct3 | rd | opcode |
|---|---|---|---|---|---|
| 6 bits | 6 bits | 5 bits | 3 bits | 5 bits | 7 bits |

- **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)

# AND Operations

- **Useful to mask bits in a word**
  - Select some bits, clear others to 0

```
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

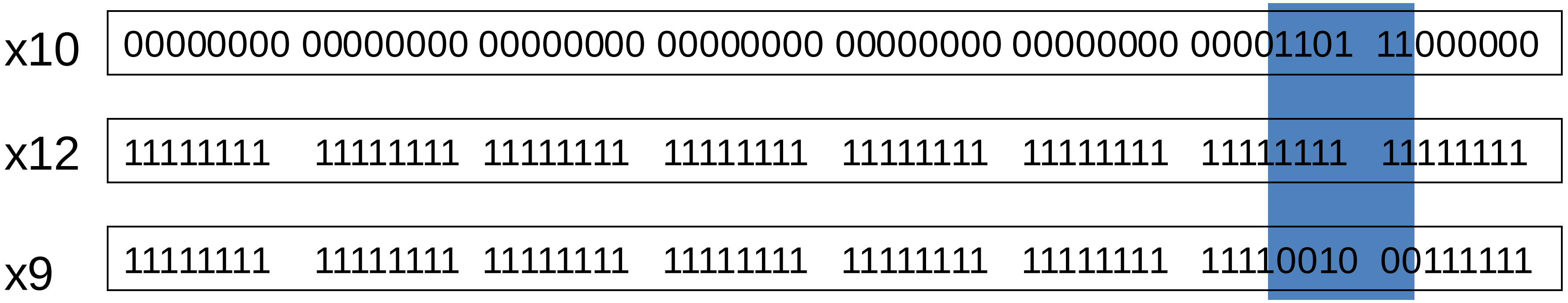
```
or x9,x10,x11
```

| | |
|---|---|
| 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**
  - Mark different bits between two words

```
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
```

# RISC-V operands

| Name | Example | Comments |
|---|---|---|
| 32 registers | *x*0-*x*31 | Fast locations for data. In RISC-V, data must be in registers to perform arithmetic. Register x0 always equals 0. |
| $2^{61}$ memory (double) words | Memory[0], Memory[8], …, Memory[18,446,744,073,709,551,608]] | Accessed only by data transfer instructions. RISC-V uses byte addresses, so sequential double word accesses differ by 8. Memory holds data structures, arrays, and spilled registers. |

| Category | Instruction | Example | Meaning | Comments |
|---|---|---|---|---|
| **Logical** | and | and x5, x6, 3 | x5=x6 & 3 | Arithmetic shift right by register |
| | inclusive or | or x5,x6,x7 | x5=x6 \| x7 | Bit-by-bit OR |
| | exclusive or | xor x5,x6,x7 | x5=x6 ^ x7 | 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 |

# 2.7 Instructions for making decisions

- **Branch instructions**
  - 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, L1**

  // if (rs1 != rs2) branch to instruction labeled L1

## Example Compiling an if statement

- **Example 2.9**

  **( Assume: f ~ j ---- x19 ~ x23 )**

  - C code:

```
             if ( i  = =  j )   goto  L1 ;
             f  =  g  +  h ;
L1:          f  =  f  -  i ;
```

  - RISC-V assembly code:

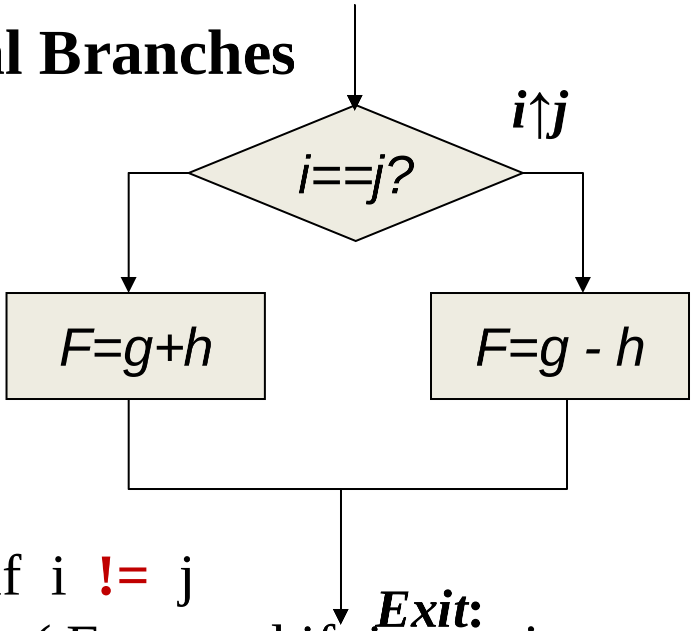
```
             beq     x21,  x22,  L1          # go to L1 if  i  equals  j
             add     x19,  x20,  x21         # f  =  g + h  ( skipped if  i  equals
j )
L1:          sub     x19,  x19,  x22         # f  =  f  -  i  ( always executed )
```

# Compiling if-then-else

- **Example 2.10**

  **Compiling *if-then-else* into Conditional Branches**

  **( Assume: f ~ j  ---- x19 ~ x23 )**



- C code:

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

- RISCV assembly code:

```
      bne     x22, x23, Else    # go to Else if  i  !=  j
      add     x19, x20, x21     # f  =  g  +  h  ( Executed if  i  = =  j
if)
      beq    x0, x0,  EXIT      # go to Exit
Else:  sub    x19, x20, x21     # f  =  g  -  h  ( Executed if i ≠ j
else)
Exit:                           # the first instruction of the next C
       …… statement
```

ZHEJIANG UNIVERSITY 浙江大学

系统结构与网络安全研究所

# Compiling LOOPs

- **Example 2.11 Compiling a loop with variable array index**

  **( Assume: g ~ j ----x19 ~ x23      base of A[i] ---- x25)**

  - C code:

```
Loop:    g  =  g  +  A[i] ;        // A is an array of 100 words
         i  =  i  +  j ;
         if ( i  !=  h )   goto  Loop ;
```

  - RISC-V assembly code:

```
Loop:    slli    x10, x22, 3          # temp reg x10 =  8  *  i
         add     x10, x10, x25        # x10  =  address of A[i]
         ld      x19, 0(x10)          # temp reg x19  =  A[i]
         add     x20, x20, x19        # g  =  g  +  A[i]
         add     x22, x22, x23        # i  =  i  +  j
         bne     x22, x21, Loop       # go to Loop  if  i  !=  h
```

# Compiling while

- **Example 2.12** **Compiling a *while* loop**

  **( Assume: i ~ k---- x22 and x24      base of save ---- x25)**

  - C code:

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

  - RISCV assembly code:

```
 Loop:   slli    x10, x22, 3          # temp reg $t1  =  8  *  i
         add     x10, x10, x25        # x10  =  address of save[i]
         ld      x9, 0(x10)           # x9 gets save[i]
         bne     x9, x24, Exit        # go to Exit  if  save[i]  !=  k
         addi    x22, x22, 1          # i  +=   1
         beq     x0, x0, Loop         # go to Loop
Exit:
```

## 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

# Hold out Case/Switch

- **Used to select one of many alternatives**
- **Example 2.14**

**Compiling a switch using *jump address* table**

**( Assume: f ~ k --x20 ~ x25        x5 contains 4/8)**

C code:

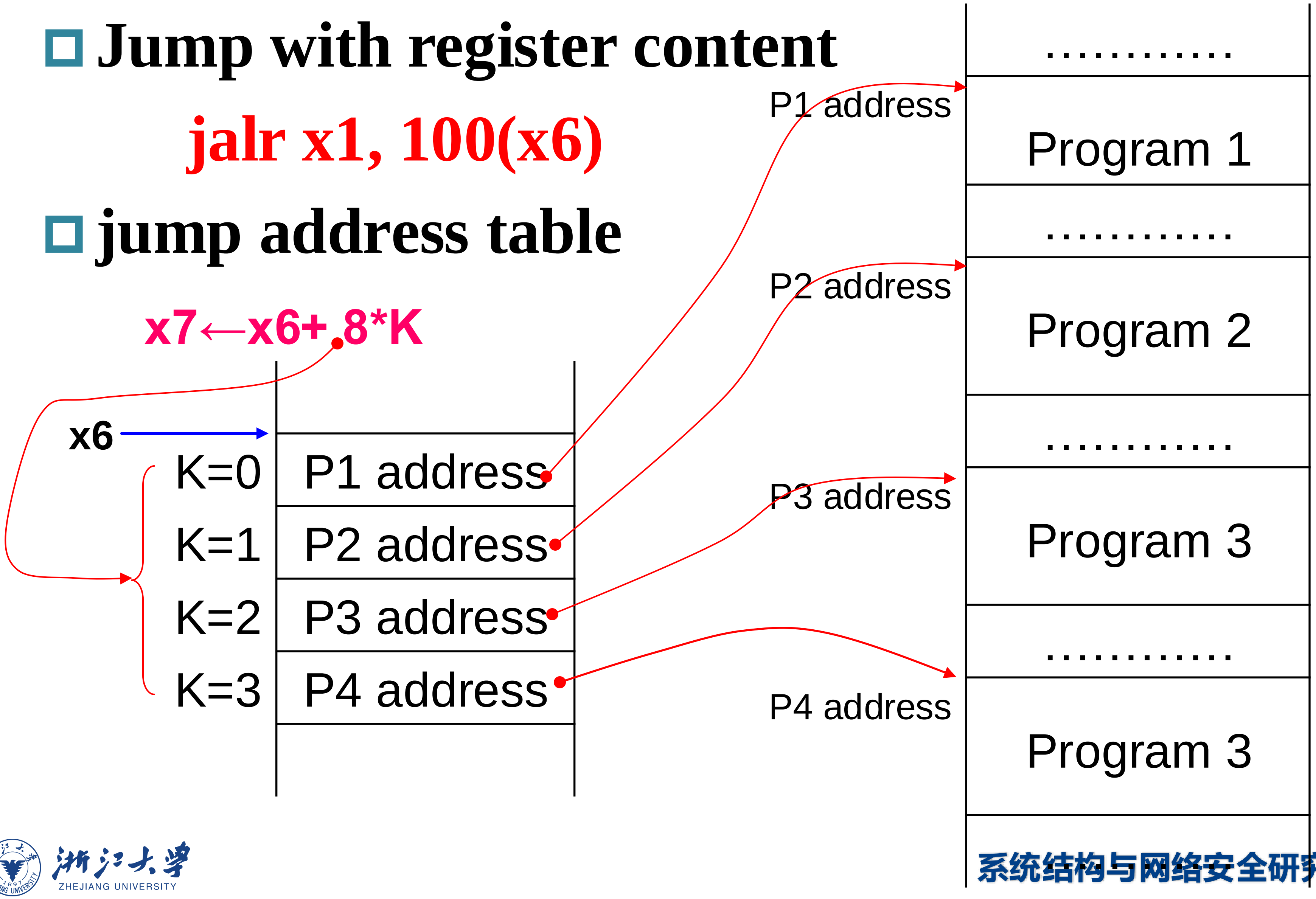
```
switch ( k )  {
        case  0 :   f  =  i  +  j ;  break ;   /*  k  =  0  */
        case  1 :   f  =  g +  h ;  break ;  /*  k  =  1  */
        case  2 :   f  =  g  -  h ;  break ;  /*  k  =  2  */
        case  3 :   f  =  i  -  j ;  break ;   /*  k  =  3  */
}
```

# Jump register & jump address table

- Jump with register content

  **jalr x1, 100(x6)**

- jump address table



# RISC-V assembly code:

Boundary

```
blt    x25, x0, Exit        # test if  k  <  0
bge    x25, x5, Exit        # if  k  >=  4,  go to Exit
slli   x7, x25, 3           # temp reg x7  =  8  *  k  (0<=k<=3)
add    x7, x7, x6           # x7  =  address of JumpTable[k]
ld     x7, 0(x7)            # temp reg x7 gets JumpTable[k]
jalr   x1, 0(x7)            # jump based on register x7(entrance)
```

Exit:

*jump address table*

x7= x6 +8 * k:

Memory1

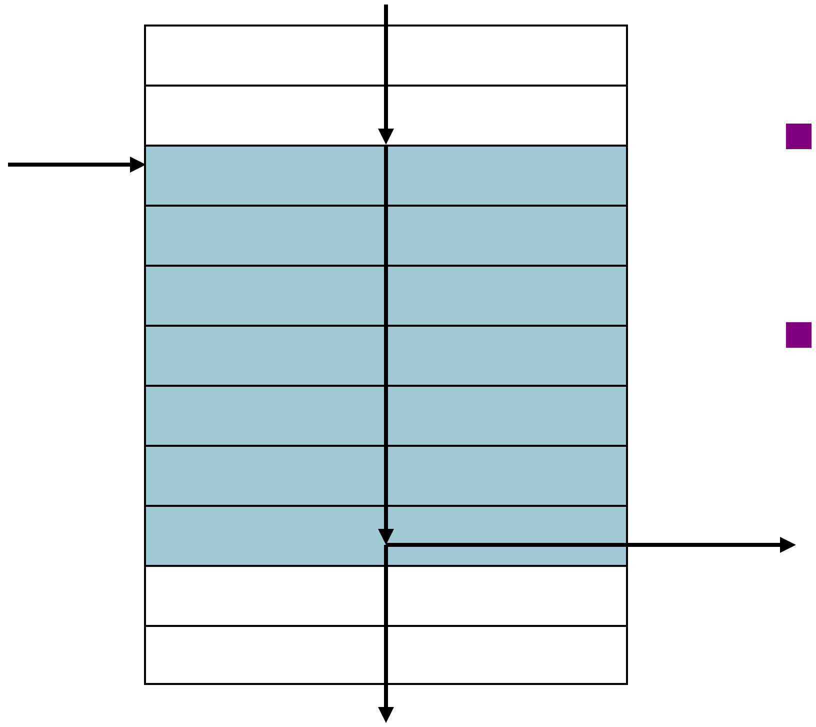
```
L0:address
L1:address
L2: address
L3:address
```

```
L0:   add    $s0, $s3, $s4       # k  =  0  so  f  gets  i  +  j
      jalr   x0, 0(x1)           # end of this case so go to Exit
L1:   add    $s0, $s1, $s2       # k  =  1  so  f  gets  g  +  h
      jalr   x0, 0(x1)           # end of this case so go to Exit
L2:   sub    $s0, $s1, $s2       # k  =  2  so  f  gets  g  -  h
      jalr   x0, 0(x1)           # end of this case so go to Exit
L3:   sub    $s0, $s3, $s4       # k  =  3  so  f  gets  i  -  j
      jalr   x0, 0(x1)           # end of  switch statement
```

Memory2

# Important conception--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

## 2.8 Supporting Procedures

- **Procedure/function be used to structure programs**
  - A stored subroutine that performs a specific task based on the parameters with which it is provided
    - easier to understand, allow code to be reused
  - **Six step**

1. Place Parameters in a place where the procedure can access them
2. Transfer control to the procedure： jump to
3. Acquire the storage resources needed for the procedure
4. Perform the desired task
5. Place the result value in a place where the calling program can access it
6. Return control to the point of origin

## Procedure Call Instructions

- **Instruction for procedures: jal ( jump-and-link )**

  **Caller jal x1, ProcedureAddress**

  - Address of following instruction put in x1
  - Jumps to target address

PC+4 → ra

- **Procedure return: jump and link register**

  **Callee jalr x0, 0(x1)**

  - Like jal, but jumps to 0 + address in x1
  - Use x0 as rd (x0 cannot be changed)
  - Can also be used for computed jumps
    - e.g., for case/switch statements

Special registers

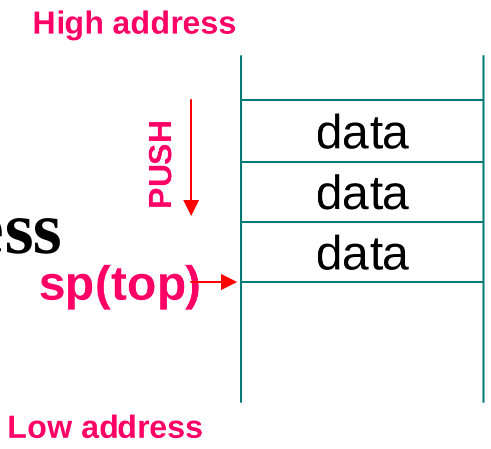
# Using More Registers

- **More Registers for procedure calling**
  - a0 ~ a7(x10-x17): eight argument registers to pass parameters & return values
  - ra/x1： one return address register to return to origin point
- **Stack**
  - ideal data structure for spilling registers
    - Push, pop
    - Stack pointer **( sp** )
- **Stack grow from higher address to lower address**
  - Push: sp= sp - 8

```
addi sp,sp,-8
sd …,0(sp)
```

  - Pop:  sp = sp + 8

```
ld      …,0(sp)
addi sp,sp,8
```

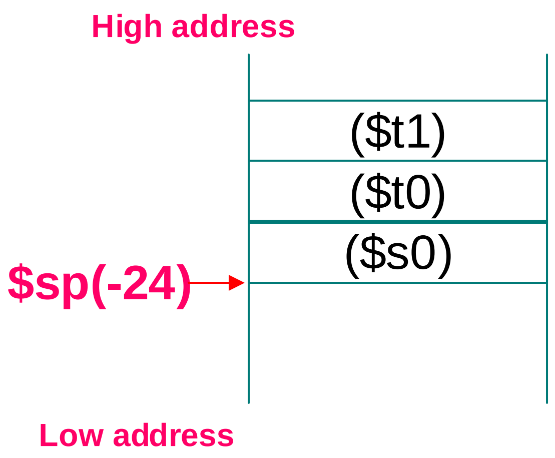


## Example 2.15 Compiling a leaf procedure

**( Assume: g, …, j in x10, …, x13 and f in x20)**

- C code:

```
long long int   leaf_example (
    long long int g, long long int h,
    long long int i, long long int j){
        long long int f;
        f = (g + h) - (i + j);
        return f;
    }
```



- RISC-V assembly code:

```
addi   sp, sp, -24        # adjust stack to make room for 3 items
sd     x5, 16(sp)         #These three instructions save three
sd     x6,  8(sp)         # register x5,x6,x20
sd     x20, 0(sp)         #  Let's consider why it need to be done.
```

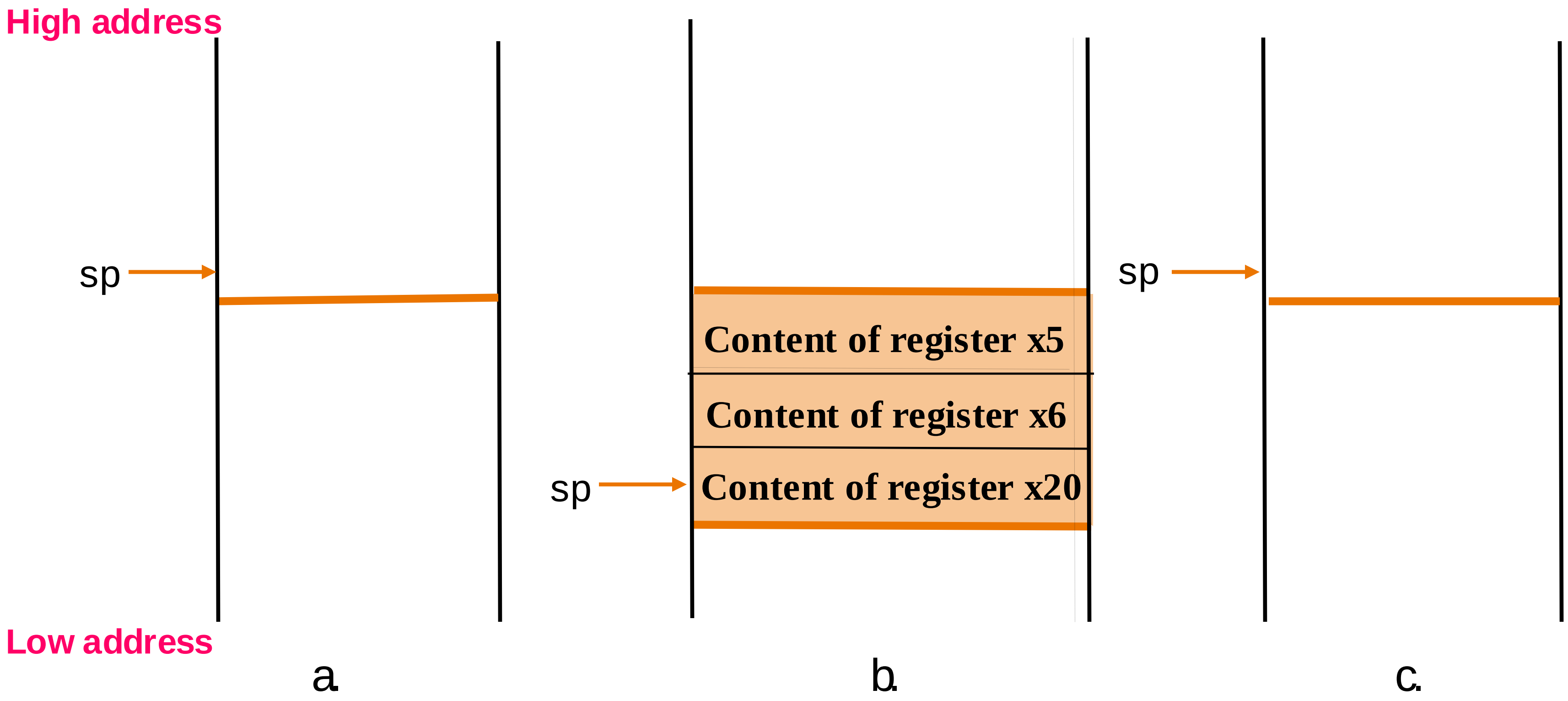
```
add  x5,x10,x11         # register x5contains  g + h
add  x6,x12,x1          # register x6 contains  i + j
sub  x20,x5,x6          # f = x5- x6, which is ( g + h ) – ( i + j )

addi x10,x20,0          # copy f to return register ( x10 = x20 + 0)
```

POP

```
ld    x20, 0(sp)        # restore register x20 for caller
ld    x6,  8(sp)        # restore register x6 for caller
ld    x5, 16(sp)        # restore register x5 for caller
addi sp, sp, +24        # adjust stack to delete 3 items
jalr x0,0(x1)           # jump back to calling routine
```

- **But maybe some of the three are not used by *the caller***
  - So, this way might be inefficient to save x5, x6, x20 on stack
  - Two classes of registers
    - t0 ~ t6: 7 temporary registers, by the callee not preserved
    - s0 ~ s11: 12 saved registers, must be preserved If used

# The values of the stack pointer and stack before, during and after procedure call in Example 2.15



- **Conflict over the use of register both**
  - **Push all the registers to stack**
    - Caller: pushes a0~a7 or t0~t6
    - Callee: pushes ra (return address) and s0~s11

# Nested Procedures

- **Example 2.16 Compiling a recursive procedure ( Assume: n -- a0 )**
  - C code for n!

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

Argument n in a0
Result in a0

  - RISC-V assembly code

```
fact:  addi    sp, sp, -16              # adjust stack for 2 items
       sd      ra, 8(sp)                # save the return address： x1
       sd      a0, 0(sp)                # save the argument  n: x10
       addi    t0, a0, -1               # x5 = n  -  1
       bge     t0, zero, L1             # if  n  >=  1, go to L1(else)
       addi    a0, zero, 1              # return 1 if n <1
       addi    sp, sp, 16           # Recover sp (Why not recover x1and x10 ?)
       jalr    zero, 0(ra)              # return to caller
```

# Nested Procedures

Argument n in a0
Result in a0

- RISC-V assembly code

```
fact:  addi   sp, sp, -16          # adjust stack for 2 items
       sd     ra, 8(sp)            # save the return address:  x1
       sd     a0, 0(sp)            # save the argument  n: x10
       addi   t0, a0, -1           # x5 = n  -  1
       bge    t0, zero, L1         # if  n  >=  1, go to L1(else)
       addi   a0, zero, 1          # return 1 if n <1
       addi   sp, sp, 16           # Recover sp (Why not recover x1and x10 ?)
       jalr   zero, 0(ra)          # return to caller
L1:    addi   a0,  a0, -1          # n  >=  1: argument gets ( n  -  1 )
       jal    ra, fact             # call fact with ( n  -  1)
------------------------------------------------------------------------------
       add    t1, a0, zero         #move result of fact(n - 1) to x6(t1)
       ld     a0, 0(sp)            # return from jal: restore argument n
       ld     ra, 8(sp)            # restore the return address
       add    sp, sp, 16           # adjust stack pointer to pop 2 items
       mul    a0, a0, t1           # return  n*fact ( n  -  1 )
       jalr   zero, 0(ra)          # return to the  caller
```

## Nested Procedures-Continue

```
L1:  addi   a0,  a0, -1             # n  >=  1: argument gets ( n  -  1 )
     jal    ra, fact                # call fact with ( n  -  1)
     add    t1, a0, zero            #move result of fact(n - 1) to x6(t1)
     ld     a0, 0(sp)               # return from jal: restore argument n
     ld     ra, 8(sp)               # restore the return address
     add    sp, sp, 16              # adjust stack pointer to pop 2 items
     mul    a0, a0, t1              # return  n*fact ( n  -  1 )
     jalr   zero, 0(ra)             # return to the  caller
```

- **Why a0 is saved?     Why ra is saved?**
- **Preserved things across a procedure call**

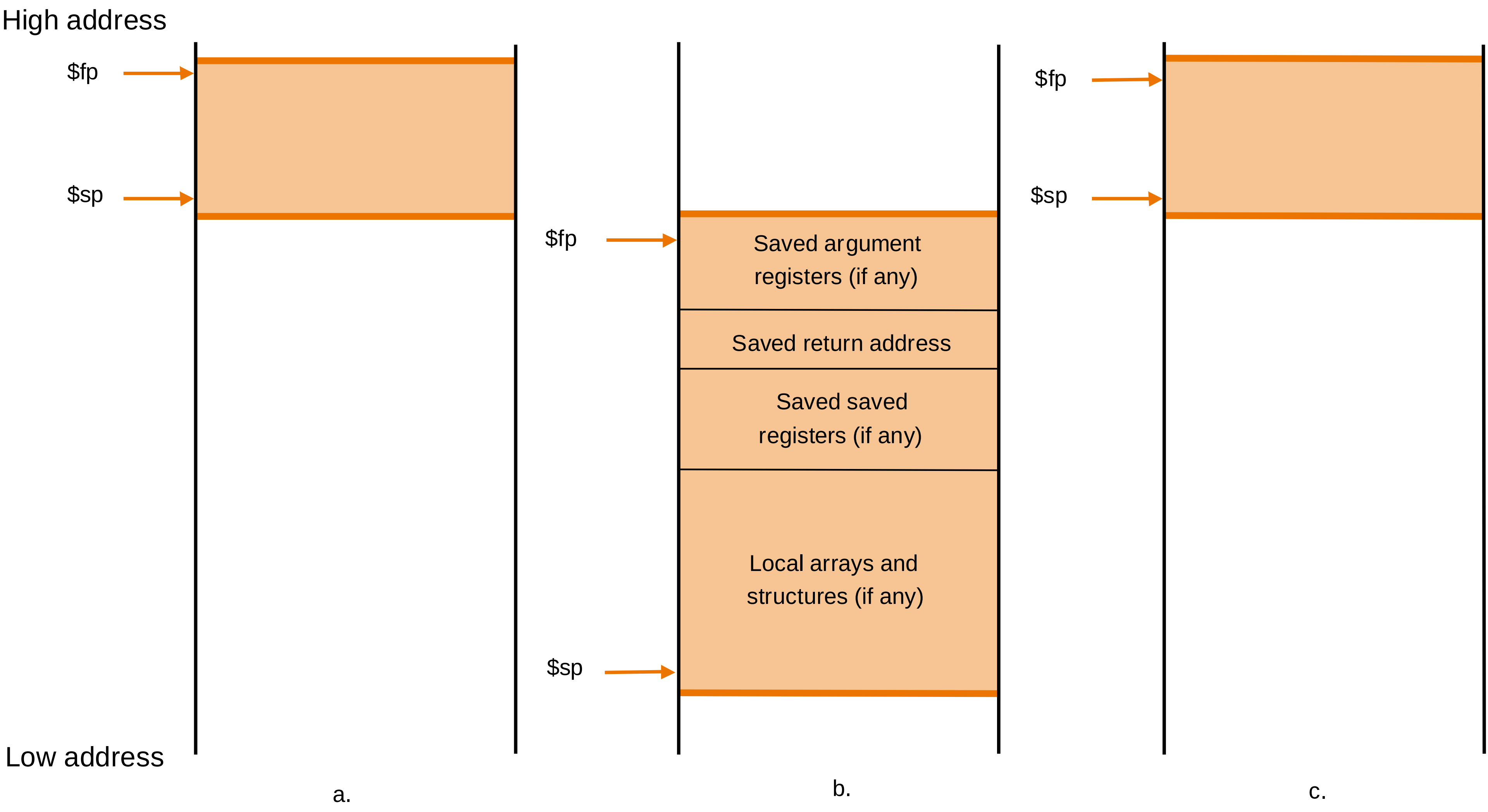
  Saved registers( s0 ~ s11), stack pointer register( $sp ),
  return address register( ra/x1 ), stack above the stack pointer
- **Not preserved things across a procedure call**

  Temporary registers( t0 ~ t7 ), argument registers( a0 ~ a7),
  return value registers( a0 ~ a7), stack below the stack pointer

## What is and what is not preserved across a procedure call

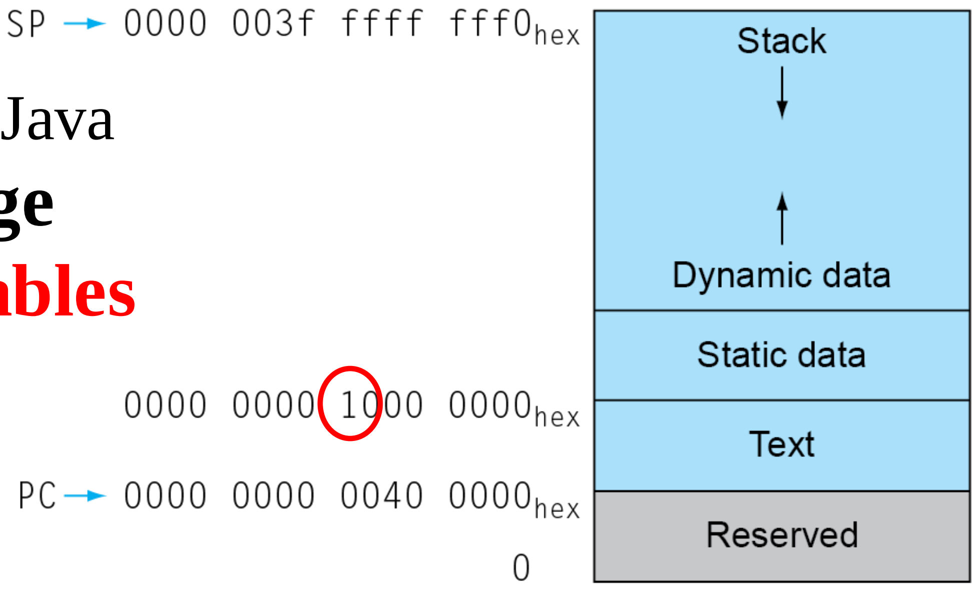
| Preserved | Not preserved |
|---|---|
| Saved registers: x8-x9, x18-x27 | Temporary registers: x5-x7, x28-x31 |
| Stack pointer register: x2(sp) | Argument/result registers: x10-x17 |
| Frame pointer: x8(fp) | |
| Return address: x1(ra) | |
| Stack above the stack pointer | Stack below the stack pointer |

# Stack allocation before, during and after procedure call


High address
$fp
$sp
$fp
Saved argument registers (if any)
Saved return address
Saved saved registers (if any)
Local arrays and structures (if any)
$sp
$fp
$sp
Low address
a.
b.
c.

# Memory Layout

- **Text: program code**
- **Static data: global variables**
  - e.g., static variables in C, constant arrays and strings
  - x3 (global pointer) initialized to address allowing ±offsets into this segment
- **Dynamic data: heap**
  - E.g., malloc in C, new in Java
- **Stack: automatic storage**
- **Storage class of C variables**
  - *automatic*
  - *static*

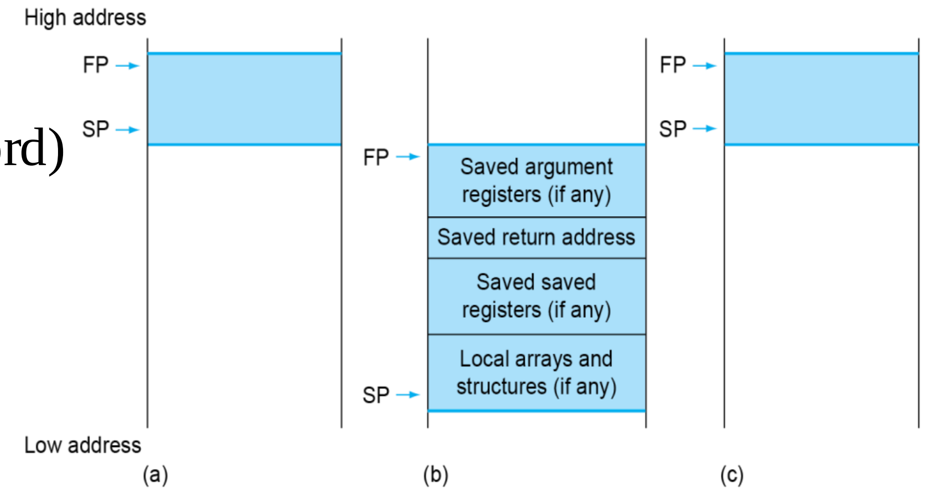


## Local Data on the Stack

- **Allocating Space for New Data on the Stack**
  - Procedure frame/activation record
    - The segment of stack containing a procedure’s saved registers and local variables
  - Frame pointer
    - A value denoting the location of saved register and local variables for a given procedure
    - Local data allocated by callee

      e.g., C automatic variables
    - Procedure frame (activation record)

      Used by some compilers to manage stack storage


High address
FP
SP
FP
Saved argument registers (if any)
Saved return address
Saved saved registers (if any)
Local arrays and structures (if any)
SP
FP
SP
Low address
(a)
(b)
(c)

# RISC-V operands

| Name | Example | Comments |
|---|---|---|
| 32 registers | *x*0-*x*31 | Fast locations for data. In RISC-V, data must be in registers to perform arithmetic. Register x0 always equals 0. |
| $2^{61}$ memory double words | Memory[0], Memory[8], …, Memory[18,446,744,073,709,551,608]] | Accessed only by data transfer instructions. RISC-V uses byte addresses, so sequential double word accesses differ by 8. Memory holds data structures, arrays, and spilled registers. |

| Name | Register no. | Usage | Preserved on call |
|---|---|---|---|
| ***x*0(zero)** | **0** | **The constant value 0** | n.a. |
| *x*1(**ra**) | 1 | Return address(link register) | yes |
| *x*2(**sp**) | 2 | Stack pointer | yes |
| *x*3(gp) | 3 | Global pointer | yes |
| *x*4(tp) | 4 | Thread pointer | yes |
| *x*5-*x*7(t0-t2) | 5-7 | Temporaries | no |
| *x*8(s0/fp) | 8 | Saved/frame point | Yes |
| *x*9(s1) | 9 | Saved | Yes |
| *x*10-*x*17(a0-a7) | 10-17 | Arguments/results | no |
| *x*18-*x*27(s2-s11) | 18-27 | Saved | yes |
| *x*28-*x*31(t3-t6) | 28-31 | Temporaries | No |
| PC | - | Auipc(Add Upper Immediate to PC) | Yes |

# RISC-V assembly language

| Category | Instruction | Example | Meaning | Comments |
|---|---|---|---|---|
| Arithmetic | add | add x5,x6,x7 | x5=x6 + x7 | Add two source register operands |
| | subtract | sub x5,x6,x7 | x5=x6 - x7 | First source register subtracts second one |
| | add immediate | addi x5,x6,20 | x5=x6+20 | Used to add constants |
| Data transfer | load doubleword | ld x5, 40(x6) | x5=Memory[x6+40] | doubleword from memory to register |
| | store doubleword | sd x5, 40(x6) | Memory[x6+40]=x5 | doubleword from register to memory |
| | 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 |
| Data transfer | 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 word, unsigned | lbu x5, 40(x6) | x5=Memory[x6+40] | Unsigned byte 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-bits constant shifted left 12 bits |

# RISC-V assembly language

| Category | Instruction | Example | Meaning | Comments |
|---|---|---|---|---|
| Logical | and | and x5, x6, 3 | x5=x6 & 3 | Arithmetic shift right by register |
| | inclusive or | or x5,x6,x7 | x5=x6 \| x7 | Bit-by-bit OR |
| | exclusive or | xor x5,x6,x7 | x5=x6 ^ x7 | 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 | 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 |
| Conditional branch | branch if equal | beq x5, x6, 100 | if(x5 == x6) go to PC+100 | PC-relative branch if registers equal |
| | branch if not equal | bne x5, x6, 100 | if(x5 != x6) go to PC+100 | PC-relative branch if registers not equal |
| | branch if less than | blt x5, x6, 100 | if(x5 < x6) go to PC+100 | PC-relative branch if registers less |
| | branch if greater or equal | bge x5, x6, 100 | if(x5 >= x6) go to PC+100 | PC-relative branch if registers greater or equal |
| | branch if less, unsigned | bltu x5, x6, 100 | if(x5 >= x6) go to PC+100 | PC-relative branch if registers less, unsigned |
| | branch if greater or equal, unsigned | bgeu x5, x6, 100 | if(x5 >= x6) go to PC+100 | PC-relative branch if registers greater or equal, unsigned |
| Unconditional branch | jump and link | jal x1, 100 | x1 = PC + 4; go to PC+100 | PC-relative procedure call |
| | jump and link register | jalr x1, 100(x5) | x1 = PC + 4; go to x5+100 | procedure return; indirect call |

## 2.9 Communicating with People
## Character Data

- **Byte-encoded character sets**
  - ASCII ( American Standard Code for Information Interchange )
    - 128 characters： 95 graphic, 33 control
  - Latin-1: 256 characters
    - ASCII, +96 more graphic characters
- **Unicode: 32-bit character set**
  - Used in Java, C++ wide characters, …
  - Most of the world’s alphabets, plus symbols
  - UTF-8, UTF-16: variable-length encodings

# Byte/Halfword/Word Operations

- RISC-V byte/halfword/word load/store
  - Load byte/halfword/word: Sign extend to 64 bits in rd
    - `lb rd, offset(rs1)`
    - `lh rd, offset(rs1)`
    - `lw rd, offset(rs1)`
  - Load byte/halfword/word unsigned: 0 extend to 64 bits in rd
    - `lbu rd, offset(rs1)`
    - `lhu rd, offset(rs1)`
    - `lwu rd, offset(rs1)`
  - Store byte/halfword/word: Store rightmost 8/16/32 bits
    - `sb rs2, offset(rs1)`
    - `sh rs2, offset(rs1)`
    - `sw rs2, offset(rs1)`

## String Copy Example

- **Example 2.17 Compiling a string copy procedure**

  **( Assume: base addresses for i -- x19, x's base --x10, y's base --x11 )**

  - C code: Y→X

```
void   strcpy ( char   x[ ] ,   char   y[ ] )
{     size_t   i ;
      i = 0 ;
      while ( ( x[ i ] = y[ i ] ) != '\ 0' )      /* copy and test byte */
              i += 1 ;
}
```

  - RISC-V assembly code:

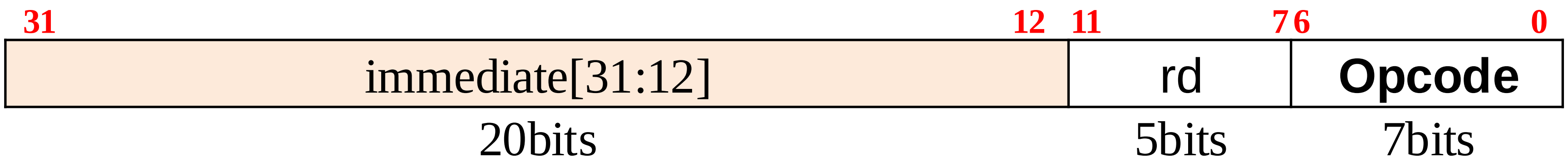
```
strcpy:   addi  sp, sp, -8           # adjust stack for 1 doubleword
          sd    x19, 0(sp)           # save x19
          add   x19, x0, x0          # i = 0
L1:       add   x5, x19, x11         # x5 = address of y[ i]
          lbu   x6, 0(x5)            # x6 = y [ i ]
          add   x7, x19, x10         # x7 = address of x[ i ]
          sb    x6, 0(x7)            # x[ i ] = y[ i ]
```

```
        beq    x6, x0,  L2          # if y[i] == 0 then exit
        addi   x19, x19, 1          # i  =  i  +  1
        jal    x0, L1               # next iteration of loop
L2:     ld     x19, 0(sp)           # restore saved old s3
        addi   sp, sp, 8            # pop 1 double word from stack
        jalr   zero  0(x1)          # return
```

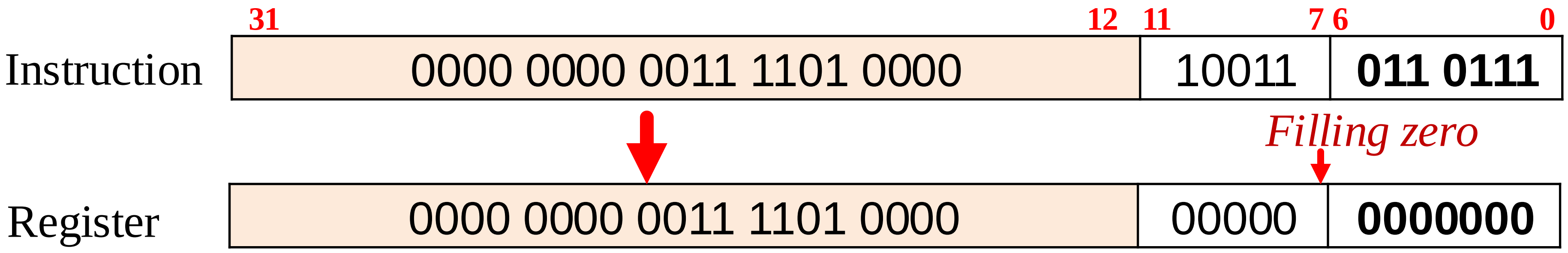
- **Optimization for example 2.17**
  - strcpy is a leaf procedure
  - Allocate  i  to a temporary register s3/x18
- **For a leaf procedure**
  - The compiler exhausts all temporary registers
  - Then use the registers it must save

## 2.10 Addressing for 32-Bit Immediate and Addresses

- **Wide Bit Immediate addressing**
  - most constants is short and fit into 12-bit field
  - Set upper 20 bits of a constants in a register with *load upper immediate* (lui rd, constant)
- instruction format (U-type)



- lui x19, 976 # 0x003D0



# 32-bit Constants

## Example 2.19 **Loading a 32-bit constant**

- The 32-bit constant:

**0000 0000 0011 1101 0000** *1001 0000 0000* $(976*16^3 + 2304=4000000)_{10}$

- RISC V code:

lui s3, 976 # 976 decimal = 0000 0000 0011 1101 0000 binary

| | | | |
|---|---|---|---|
| 0000 0000 0000 0000 | 0000 0000 0000 0000 | 0000 0000 0011 1101 0000 | 0000 0000 0000 |

(The value of s3 afterward is: 0000 0000 0011 1101 0000 0000 0000 0000)

addi s3, s3, 2304 # 2304 decimal = 1001 0000 0000 binary

| | | | |
|---|---|---|---|
| 1111 1111 1111 1111 | 1111 1111 1111 1111 | 1111 1111 1111 1111 1111 | 1001 0000 0000 |

The value of s3 afterward is: 实际补码值2304 become -1792 / sign extended

| | | | | |
|---|---|---|---|---|
| 0000 0000 0000 0000 | 0000 0000 0000 0000 | 0000 0000 0011 1100 1111 | 1001 0000 0000 | ✗ |

S3 + 2304 become S3 - 1792 ⬆

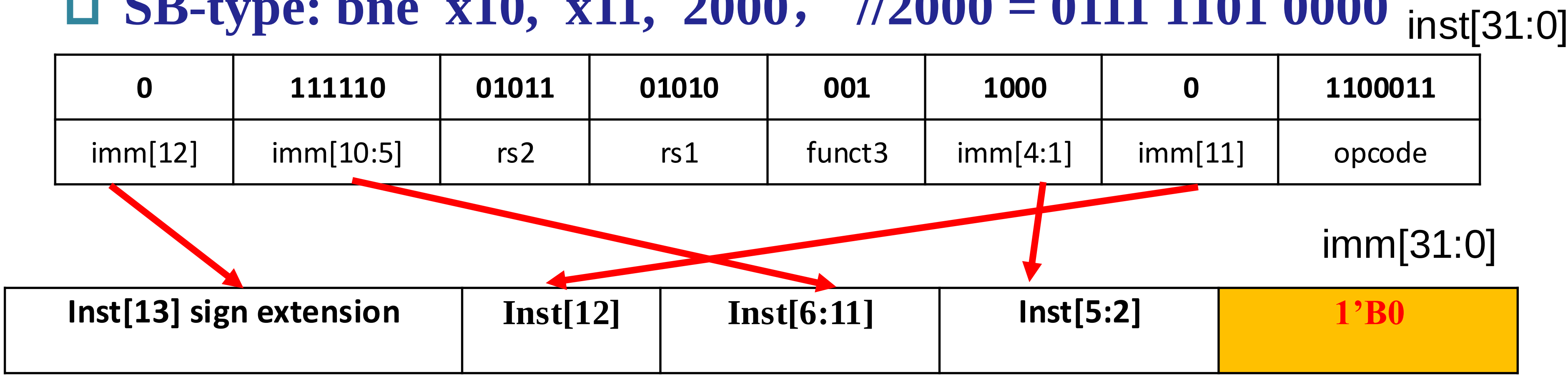
| | | | | |
|---|---|---|---|---|
| 0000 0000 0000 0000 | 0000 0000 0000 0000 | 0000 0000 0011 1101 0000 | 1001 0000 0000 | √ |

结果 + 4096（2^12）可以修正这个问题，所以就是bit 12 + 1

**这样结果就是两个立即数的拼接！**

# Branch Addressing

- **Addressing in branches**
  - Branch instructions specify
    - Opcode, two registers, target address
  - Most branch targets are near branch
    - Forward or backward
- **SB-type: bne x10, x11, 2000， //2000 = 0111 1101 0000**



| 0 | 111110 | 01011 | 01010 | 001 | 1000 | 0 | 1100011 |
|---|---|---|---|---|---|---|---|
| imm[12] | imm[10:5] | rs2 | rs1 | funct3 | imm[4:1] | imm[11] | opcode |

| | | | | |
|---|---|---|---|---|
| **Inst[13] sign extension** | **Inst[12]** | **Inst[6:11]** | **Inst[5:2]** | **1’B0** |

  - PC-relative addressing

    Target address = PC + Branch offset

    = PC + immediate × 2

## Jump Addressing

- **Jump and link (jal)**
  - target uses 20-bit immediate for larger range
- **UJ format:**

UJ = {{11{inst[31]}}, inst [31], inst[19:12], inst[20], inst[30:21],1'b0};

| 31 1bit | 10bits | 1bit | 8bits 12 | 11 7 | 6 0 |
|---|---|---|---|---|---|
| *inn[20]* | *imm[10:1]* | *imm[11]* | *imm[19:12]* | **rd** | **Opcode** |
| | ***20bit*** | | | 5bit | 7bit |

- **jal x0, 2000** $\#\ 2000_{10} = (0\ 00000000\ 0\ 111\ 1101\ 0000)_2$

| 31 1bit | 10bits | 1bit | 8bits 12 | 11 7 | 6 0 |
|---|---|---|---|---|---|
| 0 | ***1111101000*** | **0** | ***00000000*** | **00000** | **1101111** |
| | ***20bit*** | | | 5bit | 7bit |

- **For more long jumps：** eg, to 32-bit absolute address
  - lui: load address[31:12] to temp register
  - jalr: add address[11:0] and jump to target

## Show branch offset in machine language

- **Example 2.20 P116/p94**
  - C language:

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

RISC-V assembler code in Example 2.12:

```
Loop:  slli   x10,  x22, 3        # temp reg x10  =  8  *  i
       add    x10, x10, x25       # x10  =  address of save[i]
       ld     x9, 0(x10)          # temp reg x9  =  save[i]
       bne    x9, x24, Exit       # go to Exit  if  save[i]  !=  k
-----------------------------
       addi   x22, x22, 1         #  i  =  i  +  1
       beq    x0, x0, Loop        # go to Loop
Exit:
```

# Instructions Addressing and their Offset

instructions Code with Binary

| | Address | fun7 | rs2 | rs1 | fun3 | rd/offset | OP | Hex |
|---|---|---|---|---|---|---|---|---|
| Loop: slli | 80000 | 0000000 | 00011 | 10110 | 001 | 01010 | 0010011 | 003B1513 |
| add | 80004 | 0000000 | 11001 | 01010 | 000 | 01010 | 0110011 | 01950533 |
| ld | 80008 | 0000000 | 00000 | 01010 | 011 | 01001 | 0000011 | 00053483 |
| bne | ***80012*** | 0000000 | 11000 | 01001 | 001 | 0110***0*** | 1100011 | 01849663 |
| addi | 80016 | 0000000 | 00001 | 10110 | 000 | 10110 | 0010011 | 001B0B13 |
| beq | 80020 | 1111111 | 00000 | 00000 | 000 | 0110***1*** | 1100011 | FE0006E3 |
| Exit: | 80024 | …… | | | | | | |

-10

-20 = 80000 - 80020

6 =0110

PC + offset ： 12 = 80024 - 80012

- Modification:
  - All RISC-V instructions are 4 bytes long
  - PC-relative addressing refers to the number of halfwords
    - The address field at 80012 above should be 6 instead of 12

## While branch target is far away

- **Inserts an unconditional jump to target**
  - **Invert** the **condition** so that the branch decides whether to skip the jump
- **Example 2.21 p117: Branching far away**
  - Given a branch:

```
        beq   x10, x0, L1
```

  - Rewrite it to offer a much greater branching distance:

```
        bne   x10, x0, L2
         jal   x0,      L1
  L2:
```

# Summary of RISC-V architecture in Ch. 2

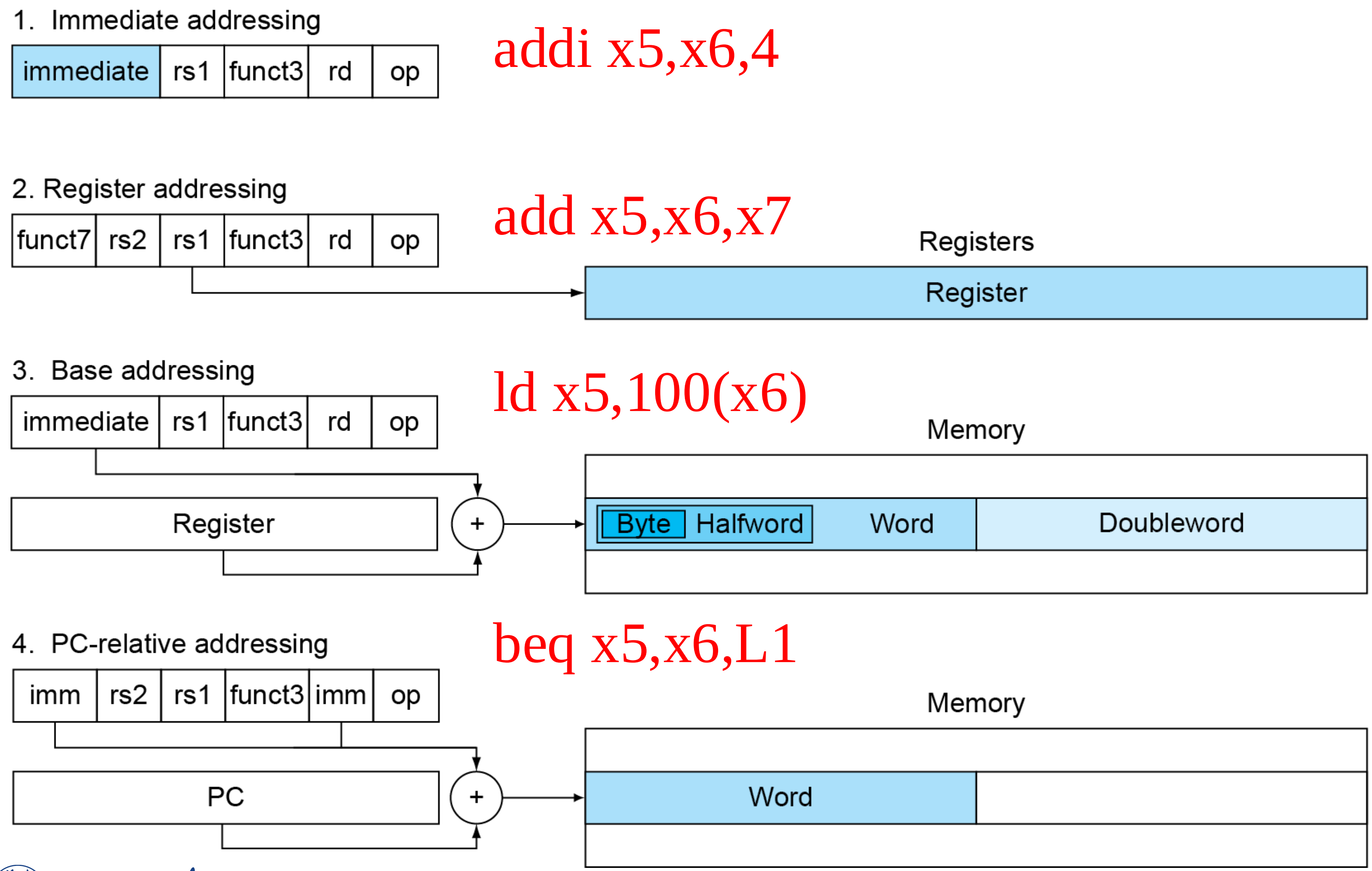
## RISC-V Instruction Format and Their Operands

| Name | Fields | | | | | | Comments |
|---|---|---|---|---|---|---|---|
| **Field size** | 31 *7bits* 25 | 24 *5bits* 20 | 19 *5bits* 15 | 14 *3bits* 12 | 11 *5bits* 7 | 6 *7bits* 0 | All RISC-V instruction 32 b |
| R-type | **funct7** | **rs2** | **rs1** | **funct3** | **rd** | **opcode** | Arithmetic instruction format |
| I-type | immediate[11:0] | | **rs1** | **funct3** | **rd** | **opcode** | Loads & immediate arithmetic |
| S-type | immed[11:5] | rs2 | **rs1** | **funct3** | immed[4:0*]* | **opcode** | Stores |
| SB-type | *imm[12,10:5]* | rs2 | **rs1** | **funct3** | *imm[4:1,11]* | **opcode** | Conditional branch format |
| UJ-type | *immediate[20,10:1,11,19:12]* | | | | **rd** | **opcode** | Unconditional jump format |
| U-type | immediate[31:12] | | | | **rd** | **opcode** | Upper immediate format |

| Name | Operands |
|---|---|
| **32 registers** | $zero, ra, sp, gp, tp, t0-t6, s0~s11, a0~a7 |
| **Mem words** | Memory[0], Memory[8], Memory[10], . . . , Memory[18,446,744,073,709,551,608] |

| | | | |
|---|---|---|---|
| ***x*0(zero)** | **0** | **The constant value 0** | n.a. |
| *x*1(**ra**) | 1 | Return address(link register) | yes |
| *x*2(**sp**) | 2 | Stack pointer | yes |
| *x*3(gp) | 3 | Global pointer | yes |
| *x*4(tp) | 4 | Thread pointer | yes |
| *x*5-*x*7(t0-t2) | 5-7 | Temporaries | no |
| *x*8(s0/fp) | 8 | Saved/frame point | Yes |
| *x*9(s1) | 9 | Saved | Yes |
| *x*10-*x*17(a0-a7) | 10-17 | Arguments/results | no |
| *x*18-*x*27(s2-s11) | 18-27 | Saved | yes |
| *x*28-*x*31(t3-t6) | 28-31 | Temporaries | No |
| PC | - | Auipc(Add Upper Immediate to PC) | |

# RISC-V Addressing Summary



addi x5,x6,4

add x5,x6,x7

ld x5,100(x6)

beq x5,x6,L1

# RISC-V Disassembly

- **Example 2.22 P120: Decoding machine code**
  - **Machine instruction(0x00578833)**

*( Bits: 31 25 24 20 7 2 0 )*

0000000 00101 01111 000 10000 0110011

**Decoding**

- Determine the operation from opcode

  opcode: 0110011→ **R-type arithmetic instruction**

| funct7 | rs2 | rs1 | funct3 | rd | opcode |
|---|---|---|---|---|---|
| 000 0000 | 00101 | 01111 | 000 | 10000 | **0110011** |

**p107-P119**

funct7 and funct3 are all 0 → **add instruction**

- Determine other fields

  **rs2: x5/t0; rs1: x15/a5; rd: x16/a7**

- Show the assembly instruction：

  **add a7, a5, t0** (Note: add rd,rs,rt)

# Summary of RISC-V instruction encoding

| Format | Instruction | Opcode | Funct3 | Funct6/7 |
|---|---|---|---|---|
| R-type | add | 0110011 | 000 | 0000000 |
| | sub | 0110011 | 000 | 0100000 |
| | sll | 0110011 | 001 | 0000000 |
| | xor | 0110011 | 100 | 0000000 |
| | srl | 0110011 | 101 | 0000000 |
| | sra | 0110011 | 101 | 0100000 |
| | or | 0110011 | 110 | 0000000 |
| | and | 0110011 | 111 | 0000000 |
| | lr.d | 0110011 | 011 | 0001000 |
| | sc.d | 0110011 | 011 | 0001100 |

## Summary of RISC-V instruction encoding

| Format | Instruction | Opcode | Funct3 | Funct6/7 |
|---|---|---|---|---|
| I-type | lb | 0000011 | 000 | n.a. |
| | lh | 0000011 | 001 | n.a. |
| | lw | 0000011 | 010 | n.a. |
| | ld | 0000011 | 011 | n.a. |
| | lbu | 0000011 | 100 | n.a. |
| | lhu | 0000011 | 101 | n.a. |
| | lwu | 0000011 | 110 | n.a. |
| | addi | 0010011 | 000 | n.a. |
| | slli | 0010011 | 001 | 000000 |
| | xori | 0010011 | 100 | n.a. |
| | srli | 0010011 | 101 | 000000 |
| | srai | 0010011 | 101 | 010000 |
| | ori | 0010011 | 110 | n.a. |
| | andi | 0010011 | 111 | n.a. |
| | jalr | 1100111 | 000 | n.a. |

# Summary of RISC-V instruction encoding

| Format | Instruction | Opcode | Funct3 | Funct6/7 |
|---|---|---|---|---|
| S-type | sb | 0100011 | 000 | n.a. |
| | sh | 0100011 | 001 | n.a. |
| | sw | 0100011 | 010 | n.a. |
| | sd | 0100011 | 111 | n.a. |
| SB-type | beq | 1100111 | 000 | n.a. |
| | bne | 1100111 | 001 | n.a. |
| | blt | 1100111 | 100 | n.a. |
| | bge | 1100111 | 101 | n.a. |
| | bltu | 1100111 | 110 | n.a. |
| | bgeu | 1100111 | 111 | n.a. |
| U-type | lui | 0110111 | n.a. | n.a. |
| UJ-type | jal | 1101111 | n.a. | n.a. |

# 2.11 Synchronization in RISC-V

- **Two processors sharing an area of memory**
  - P1 writes, then P2 reads
  - Data race if P1 and P2 don't synchronize
    - Result depends of order of accesses
- **Hardware support required**
  - Atomic read/write memory operation
  - No other access to the location allowed between the read and write
- **Could be a single instruction**
  - E.g., atomic swap of register ↔ memory
  - Or an atomic pair of instructions

# Synchronization in RISC-V

- **Load reserved: `lr.d rd,(rs1)`**
  - Load from address in rs1 to rd
  - Place reservation on memory address
- **Store conditional: `sc.d rd,(rs1),rs2`**
  - Store from rs2 to address in rs1
  - Succeeds if location not changed since the `lr.d`
    - Returns 0 in rd
  - Fails if location is changed
    - Returns non-zero value in rd

# Synchronization in RISC-V

- Example 1: atomic swap (to test/set lock variable)

```
again: lr.d x10,(x20)
       sc.d x11,(x20),x23 // X11 = status
       bne  x11,x0,again  // branch if store failed
       addi x23,x10,0     // X23 = loaded value
```

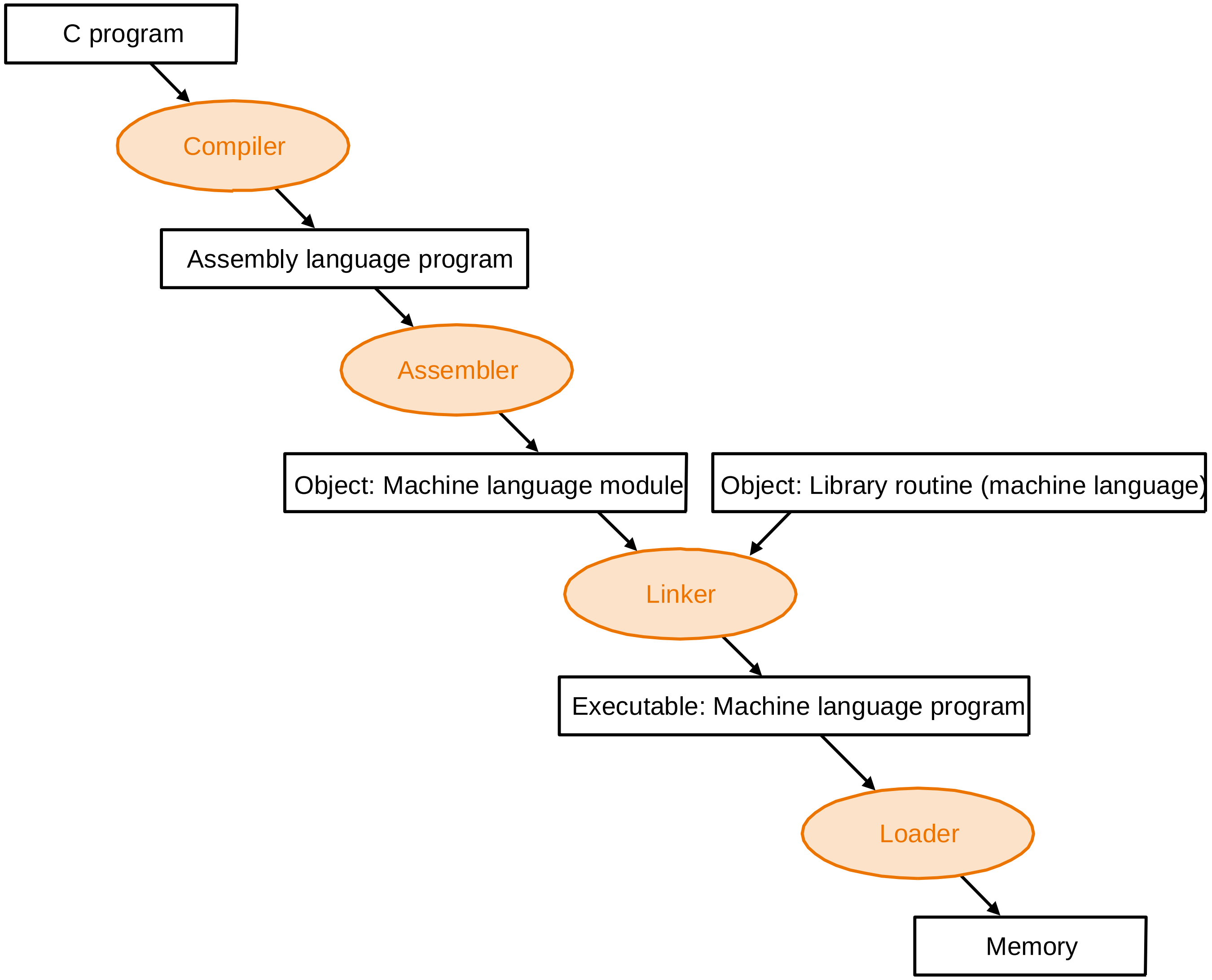
- Example 2: lock

```
       addi x12,x0,1         // copy locked value
again: lr.d x10,(x20)        // read lock
       bne  x10,x0,again     // check if it is 0 yet
       sc.d x11,(x20),x12    // attempt to store
       bne  x11,x0,again     // branch if fails
```

- Unlock:

```
       sd   x0,0(x20)        // free lock
```

# 2.12 Translating and starting a program


C program
Compiler
Assembly language program
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
Loader
Memory

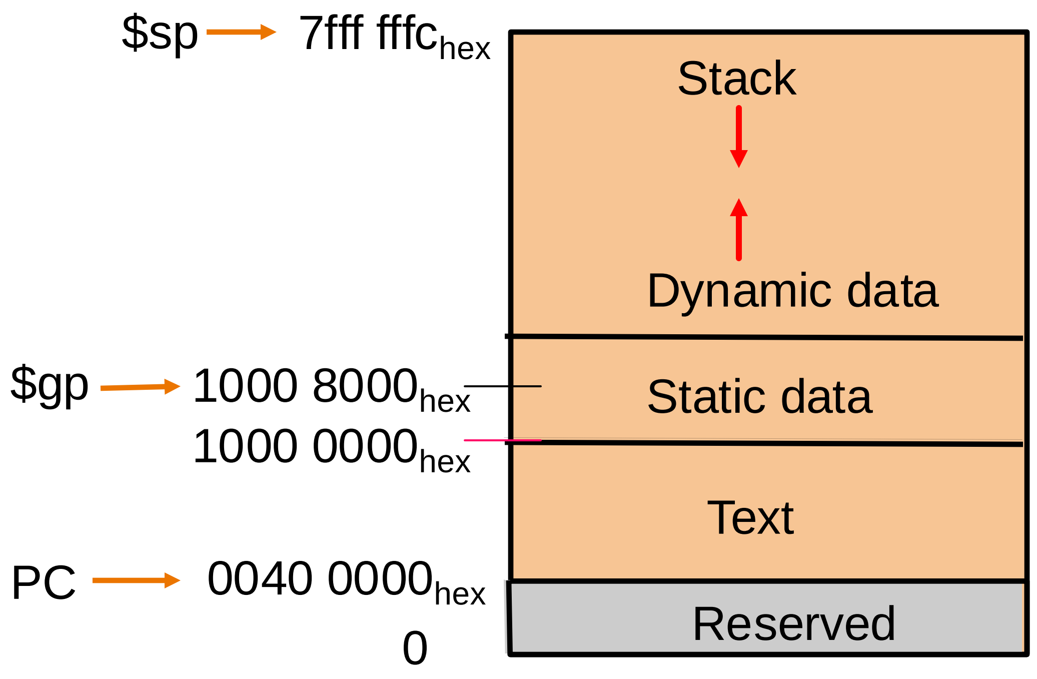
# Producing an Object Module

- Assembler (or compiler) translates program into machine instructions
- Provides information for building a complete program from the pieces
  - Header: described contents of object module
  - Text segment: translated instructions
  - Static data segment: data allocated for the life of the program
  - Relocation info: for contents that depend on absolute location of loaded program
  - Symbol table: global definitions and external refs
  - Debug info: for associating with source code

| Object file header | | | |
|---|---|---|---|
| | Name | Procedure A | |
| | Text size | $100_{hex}$ | |
| | Data size | $20_{hex}$ | |
| Text segment | Address | instruction | |
| | 0 | ld x10, 0(gp) | |
| | 4 | jal x1, 0 | |
| | …… | -- | |
| Data segment | 0 | (X) | |
| | …… | …… | |
| Relocation information | Address | Instruction type | Dependency |
| | 0 | ld | X |
| | 4 | jal | B |
| Symbol table | label | Address | |
| | X | -- | |
| | B | -- | |

# Link

- Object modules(including library routine) → **executable program**
- 3 steps of Link
  - Place code and data modules symbolically in memory
  - Determine the addresses of data and instruction labels
  - Patch both the internal and external references (**Address of invoke**)



# Loading a Program

- Load from image file on disk into memory
  1. Read header to determine segment sizes
  2. Create virtual address space
  3. Copy text and initialized data into memory
     - Or set page table entries so they can be faulted in
  4. Set up arguments on stack
  5. Initialize registers (including sp, fp, gp)
  6. Jump to startup routine
     - Copies arguments to x10, … and calls main
     - When main returns, do exit syscall

## Dynamic Linking

- **Only link/load library procedure when it is called**
  - Requires procedure code to be relocatable
  - Avoids image bloat caused by static linking of all (transitively) referenced libraries
  - Automatically picks up new library versions

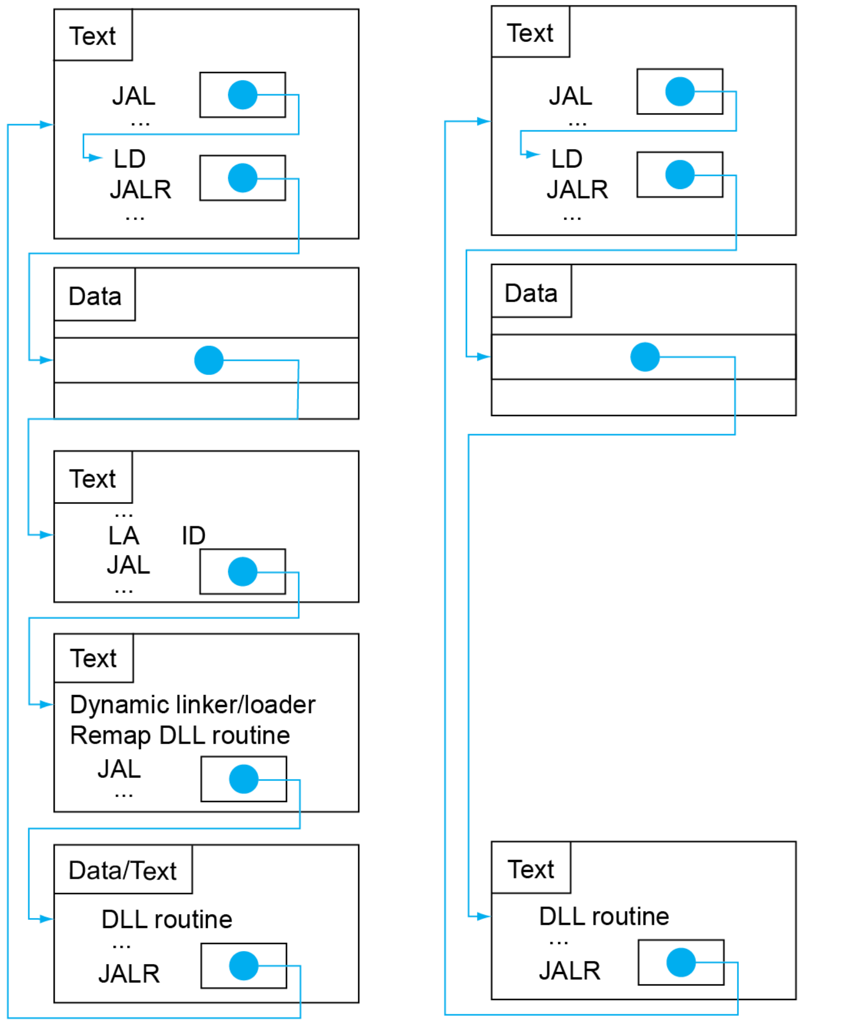
# Lazy Linkage

Indirection table

Stub: Loads routine ID,
Jump to linker/loader

Linker/loader code

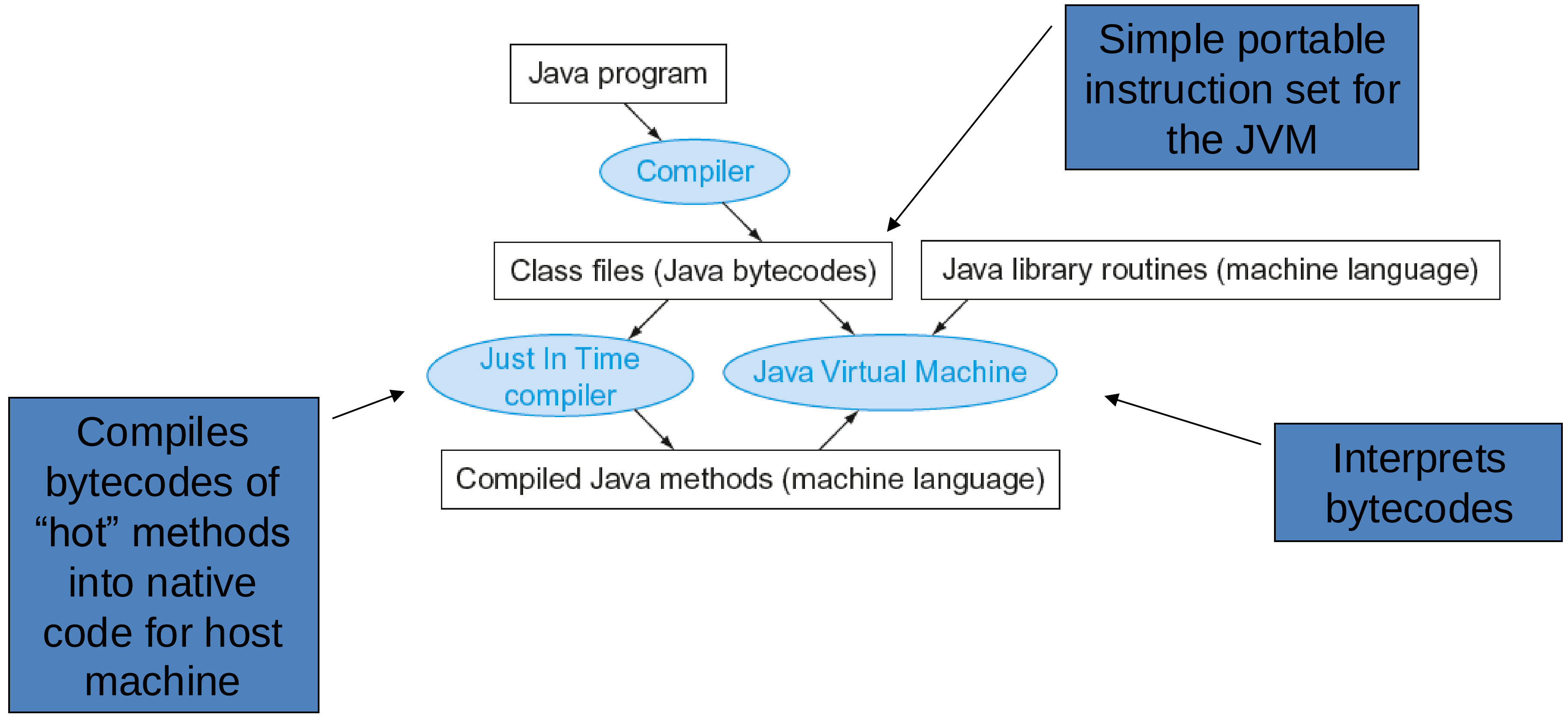
Dynamically
mapped code



(a) First call to DLL routine

(b) Subsequent calls to DLL routine

# Starting Java Applications


Java program
Compiler
Class files (Java bytecodes)
Java library routines (machine language)
Just In Time compiler
Java Virtual Machine
Compiled Java methods (machine language)
Simple portable instruction set for the JVM
Compiles bytecodes of "hot" methods into native code for host machine
Interprets bytecodes

## 2.13 A C Sort Example To Put it All Together

- **Three general steps for translating C procedures**
  - Allocate registers to program variables
  - Produce code for the body of the procedures
  - Preserve registers across the procedures invocation
- **Procedure *swap***
  - C code

```
void   swap ( long  long    v[  ] ,    size_t  k )
{
        long  long   temp ;
        temp  =  v[ k ] ;
        v[ k ] =  v[ k + 1 ] ;
        v[ k + 1 ]  =  temp ;
}
```

# The Procedure Swap

- Register allocation for *swap*

  v ---- x10    k ---- x11    temp ---- x5

- *swap* is a leaf procedure, nothing to preserve
- RISC-V code for the procedure *swap*
  - **Procedure body**

```
swap:   slli   x6,  x11, 3        //  x6 = k * 8
        add    x6,  x10, x6       //  x6 = v + ( k * 8 )
        ld     x5, 0(x6)          //  x5 ← v[ k ]
        ld     x7, 8(x6)          //  x7 ← v[ k + 1 ]
        sd     x7, 0(x6)          //  v[k+1] → v[ k ]
        sd     x5, 8(x6)          //  v[k] → v[ k + 1 ]
```

  - **Procedure return**

```
       jalr   x0,   0(x1)         //   return to calling routine
```

# The Sort Procedure in C

- **Procedure *sort***
  - C code

```
void  sort (long  long    v[  ] ,    size_t    n )
{
        size_t    i ,  j ;
        for ( i  =  0 ; i  <  n ; i + =  1 ) {
             for ( j  =  i  -  1 ; j  >=  0  &&  v[j]  >  v[j+1] ; j -=  1 )
                    swap ( v ,  j ) ;
        }
}
```

  - **Register allocation** for *sort*

    v ---- x10      n ---- x11      i ---- x19      j ---- x20

  - **Passing parameters** in *sort*
  - **Preserving registers** in *sort*

    x1 ,  x19, x20, x21, x22

# The Outer Loop

- Skeleton of outer loop:
  - for (i = 0; i <n; i += 1) {

```
   li    x19,0        // i = 0
for1tst:
   bge   x19,x11,exit1   // go to exit1 if x19 ≥ x11 (i≥n)

   (body of outer for-loop)

   addi x19,x19,1          // i += 1
   j     for1tst           // branch to test of outer loop
exit1:
```

# The Inner Loop

- **Skeleton of inner loop:**
  - for (j = i − 1; j >= 0 && v[j] > v[j + 1]; j − = 1) {

```
     addi x20,x19,-1    // j = i –1
for2tst:
   blt  x20,x0,exit2  // go to exit2 if x20 < 0 (j < 0)
   slli x5,x20,3      // reg x5 = j * 8
   add  x5,x10,x5     // reg x5 = v + (j * 8)
   ld   x6,0(x5)      // reg x6 = v[j]
   ld   x7,8(x5)      // reg x7 = v[j + 1]
   ble  x6,x7,exit2   // go to exit2 if x6 ≤ x7
   mv   x21, x10      // copy parameter x10 into x21
   mv   x22, x11      // copy parameter x11 into x22
   mv   x10, x21      // first swap parameter is v
   mv   x11, x20      // second swap parameter is j
     jal  x1,swap       // call swap
     addi x20,x20,-1    // j –= 1
     j    for2tst       // branch to test of inner loop
 exit2:
//see p139 on textbook for entire code, place of MV is different, why?
```

## Preserving Registers

- Saving registers

```
sort:   addi  sp,  sp, -40      // make room on stack for 5 registers
        sd    x1, 32(sp)        // save return address on stack
        sd    x22, 24(sp)       // save x22 on stack
        sd    x21, 16(sp)       // save x21 on stack
        sd    x20,  8(sp)       // save x20 on stack
        sd    x19,  0(sp)       // save x19 on stack
```

- Procedure body{Outer loop {Inner loop} }
- Restoring registers

```
exit1:  ld    x19,  0(sp)       // restore x19 from stack
        ld    x20,  8(sp)       // restore x20 from stack
        ld    x21,  16(sp)      // restore x21 from stack
        ld    x22, 24(sp)       // restore x22 from stack
        ld    x1, 32(sp)        // restore return address from stack
        addi  sp, sp, 40        // restore stack pointer
```

- Procedure return

```
        jalr  x0,  0(x0)        // return to calling routine
```

## 2.14 Arrays versus Pointers

- **Array indexing involves**
  - Multiplying index by element size
  - Adding to array base address
- **Pointers correspond directly to memory addresses**
  - Can avoid indexing complexity

# Example: Clearing an Array

```
clear1(int array[], int size) {
  int i;
  for (i = 0; i < size; i += 1)
    array[i] = 0;
}
```

```
clear2(int *array, int size) {
  int *p;
  for (p = &array[0]; p < &array[size];
       p = p + 1)
    *p = 0;
}
```

```
   li    x5,0          // i = 0
loop1:
   slli x6,x5,3        // x6 = i * 8
   add  x7,x10,x6      // x7 = address
                       // of array[i]
   sd   x0,0(x7)       // array[i] = 0
   addi x5,x5,1        // i = i + 1
   blt  x5,x11,loop1   // if (i<size)
                       // go to loop1
```

```
   mv x5,x10           // p = address
                       // of array[0]
   slli x6,x11,3       // x6 = size * 8
   add x7,x10,x6       // x7 = address
                       // of array[size]
loop2:
   sd x0,0(x5)         // Memory[p] = 0
   addi x5,x5,8        // p = p + 8
   bltu x5,x7,loop2
                       // if (p<&array[size])
                       // go to loop2
```

# Comparison of Array vs Pointers

- **Multiply "strength reduced" to shift**
- **Array version requires shift to be inside loop**
  - Part of index calculation for incremented i
  - c.f. incrementing pointer
- **Compiler can achieve same effect as manual use of pointers**
  - Induction variable elimination
  - Better to make program clearer and safer

# 2.16 Real Stuff: MIPS Instructions

- **MIPS: commercial predecessor to RISC-V**
- **Similar basic set of instructions**
  - 32-bit instructions
  - 32 general purpose registers, register 0 is always 0
  - 32 floating-point registers
  - Memory accessed only by load/store instructions
    - Consistent use of addressing modes for all data sizes
- **Different conditional branches**
  - For <, <=, >, >=
  - RISC-V: blt, bge, bltu, bgeu
  - MIPS: slt, sltu (set less than, result is 0 or 1)
    - Then use beq, bne to complete the branch

# Instruction Encoding

**Register-register**

| | 31 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 |
|---|---|---|---|---|---|---|
| RISC-V | funct7(7) | rs2(5) | rs1(5) | funct3(3) | rd(5) | opcode(7) |

| | 31 26 | 25 21 | 20 16 | 15 11 | 10 6 | 5 0 |
|---|---|---|---|---|---|---|
| MIPS | Op(6) | Rs1(5) | Rs2(5) | Rd(5) | Const(5) | Opx(6) |

**Load**

| | 31 20 | 19 15 | 14 12 | 11 7 | 6 0 |
|---|---|---|---|---|---|
| RISC-V | immediate(12) | rs1(5) | funct3(3) | rd(5) | opcode(7) |

| | 31 26 | 25 21 | 20 16 | 15 0 |
|---|---|---|---|---|
| MIPS | Op(6) | Rs1(5) | Rs2(5) | Const(16) |

**Store**

| | 31 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 |
|---|---|---|---|---|---|---|
| RISC-V | immediate(7) | rs2(5) | rs1(5) | funct3(3) | immediate(5) | opcode(7) |

| | 31 26 | 25 21 | 20 16 | 15 0 |
|---|---|---|---|---|
| MIPS | Op(6) | Rs1(5) | Rs2(5) | Const(16) |

**Branch**

| | 31 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 |
|---|---|---|---|---|---|---|
| RISC-V | immediate(7) | rs2(5) | rs1(5) | funct3(3) | immediate(5) | opcode(7) |

| | 31 26 | 25 21 | 20 16 | 15 0 |
|---|---|---|---|---|
| MIPS | Op(6) | Rs1(5) | Opx/Rs2(5) | Const(16) |

## 2.17 Real Stuff: The Intel x86 ISA

- **Evolution with backward compatibility**
  - 8080 (1974): 8-bit microprocessor
    - Accumulator, plus 3 index-register pairs
  - 8086 (1978): 16-bit extension to 8080
    - Complex instruction set (CISC)
  - 8087 (1980): floating-point coprocessor
    - Adds FP instructions and register stack
  - 80286 (1982): 24-bit addresses, MMU
    - Segmented memory mapping and protection
  - 80386 (1985): 32-bit extension (now IA-32)
    - Additional addressing modes and operations
    - Paged memory mapping as well as segments

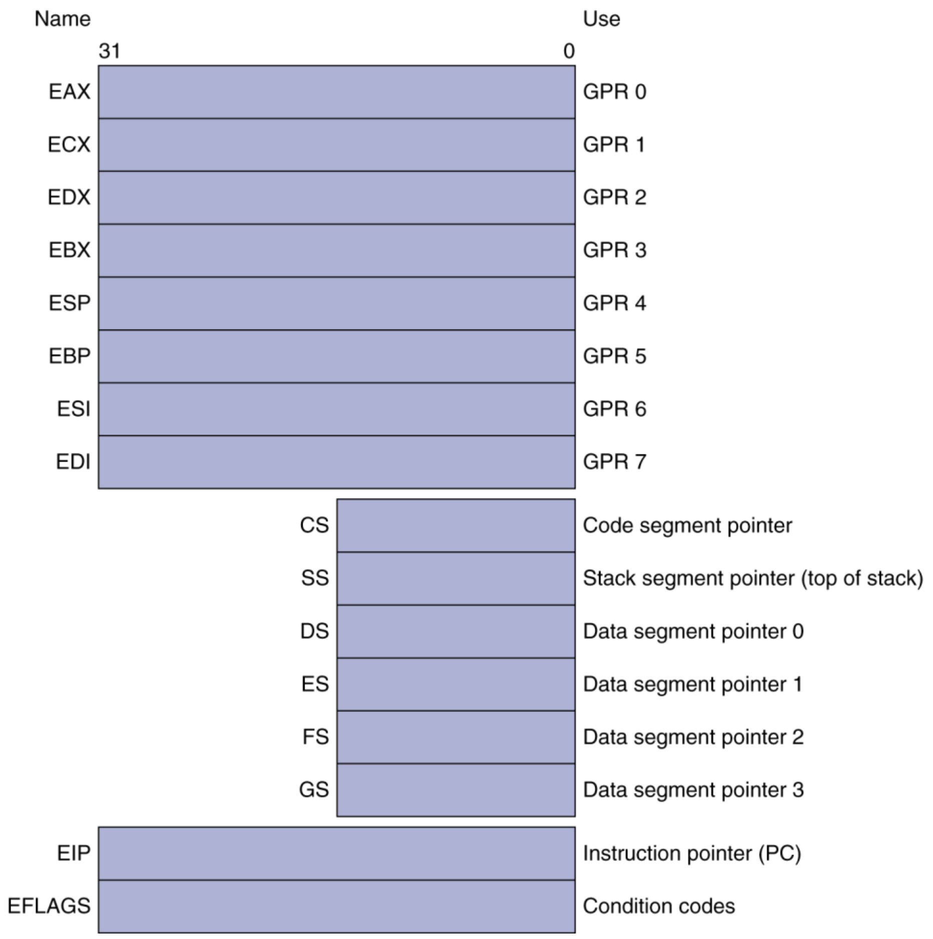
# The Intel x86 ISA

- **Further evolution…**
  - i486 (1989): pipelined, on-chip caches and FPU
    - Compatible competitors: AMD, Cyrix, …
  - Pentium (1993): superscalar, 64-bit datapath
    - Later versions added MMX (Multi-Media eXtension) instructions
    - The infamous FDIV bug
  - Pentium Pro (1995), Pentium II (1997)
    - New microarchitecture (see Colwell, *The Pentium Chronicles*)
  - Pentium III (1999)
    - Added SSE (Streaming SIMD Extensions) and associated registers
  - Pentium 4 (2001)
    - New microarchitecture
    - Added SSE2 instructions

# The Intel x86 ISA

- **And further…**
  - AMD64 (2003): extended architecture to 64 bits
  - EM64T – Extended Memory 64 Technology (2004)
    - AMD64 adopted by Intel (with refinements)
    - Added SSE3 instructions
  - Intel Core (2006)
    - Added SSE4 instructions, virtual machine support
  - AMD64 (announced 2007): SSE5 instructions
    - Intel declined to follow, instead…
  - Advanced Vector Extension (announced 2008)
    - Longer SSE registers, more instructions
- **If Intel didn't extend with compatibility, its competitors would!**
  - Technical elegance ≠ market success

# Basic x86 Registers



# Basic x86 Addressing Modes

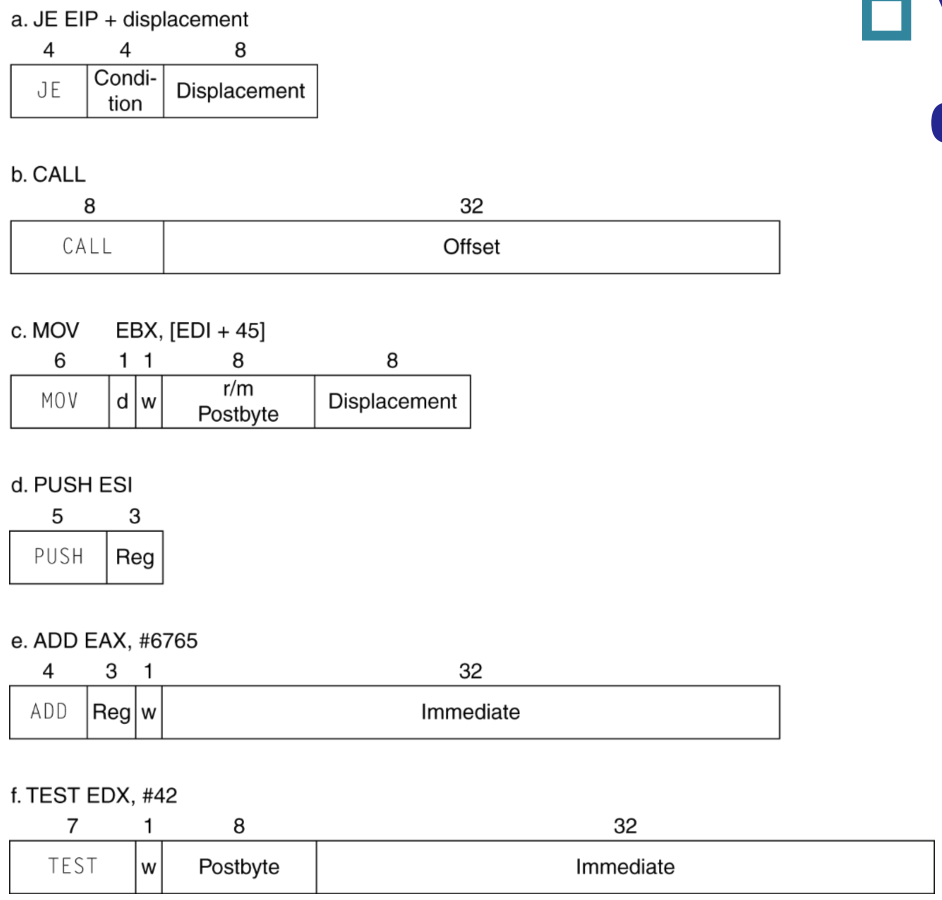
## Two operands per instruction

| Source/dest operand | Second source operand |
|---|---|
| Register | Register |
| Register | Immediate |
| Register | Memory |
| Memory | Register |
| Memory | Immediate |

## Memory addressing modes

- Address in register
- Address = $R_{base}$ + displacement
- Address = $R_{base} + 2^{scale} \times R_{index}$ (scale = 0, 1, 2, or 3)
- Address = $R_{base} + 2^{scale} \times R_{index}$ + displacement

# x86 Instruction Encoding



- **Variable length encoding**
  - Postfix bytes specify addressing mode
  - Prefix bytes modify operation
    - Operand length, repetition, locking, …

## Implementing IA-32

- **Complex instruction set makes implementation difficult**
  - Hardware translates instructions to simpler microoperations
    - Simple instructions: 1–1
    - Complex instructions: 1–many
  - Microengine similar to RISC
  - Market share makes this economically viable
- **Comparable performance to RISC**
  - Compilers avoid complex instructions

# 2.18 Other RISC-V Instructions

- **Base integer instructions (RV64I)**
  - Those previously described, plus
  - auipc rd, immed  // rd = (imm<<12) + pc
    - follow by jalr (adds 12-bit immed) for long jump
  - slt, sltu, slti, sltui: set less than (like MIPS)
  - addw, subw, addiw: 32-bit add/sub
  - sllw, srlw, srlw, slliw, srliw, sraiw: 32-bit shift
- **32-bit variant: RV32I**
  - registers are 32-bits wide, 32-bit operations

# Instruction Set Extensions

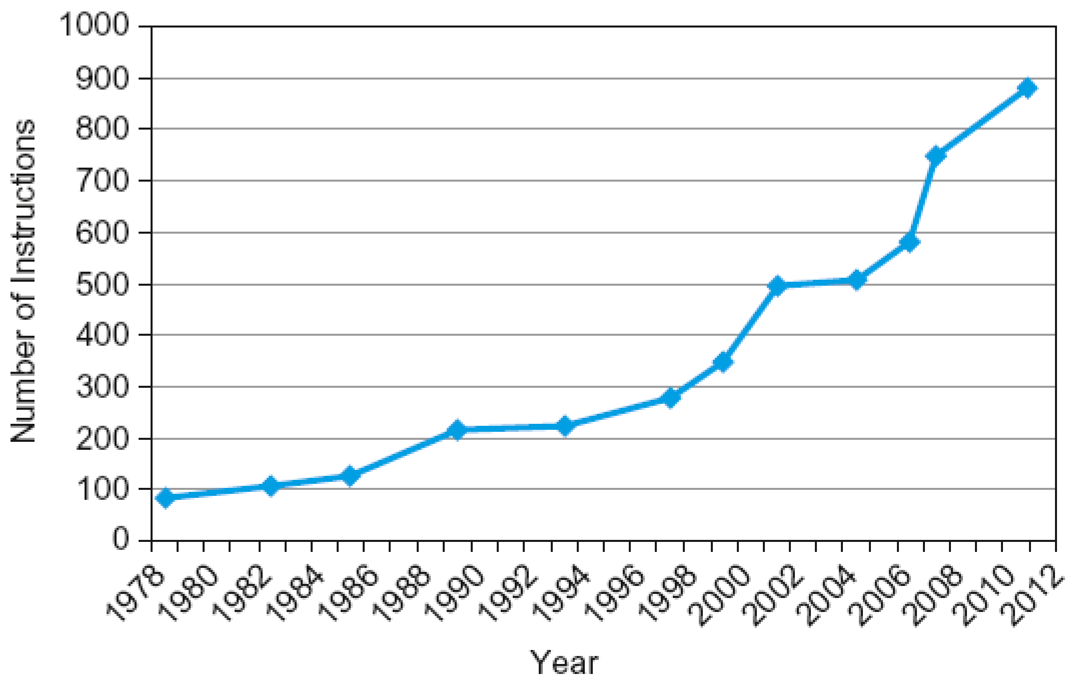
- **M: integer multiply, divide, remainder**
- **A: atomic memory operations**
- **F: single-precision floating point**
- **D: double-precision floating point**
- **C: compressed instructions**
  - 16-bit encoding for frequently used instructions

# Fallacies

- **Powerful instruction ⇒ higher performance**
  - Fewer instructions required
  - But complex instructions are hard to implement
    - May slow down all instructions, including simple ones
  - Compilers are good at making fast code from simple instructions
- **Use assembly code for high performance**
  - But modern compilers are better at dealing with modern processors
  - More lines of code ⇒ more errors and less productivity

# Fallacies

- Backward compatibility ⇒ instruction set doesn’t change
  - But they do accrete more instructions



x86 instruction set

# Pitfalls

- **Sequential words are not at sequential addresses**
  - Increment by 4, not by 1!
- **Keeping a pointer to an automatic variable after procedure returns**
  - e.g., passing pointer back via an argument
  - Pointer becomes invalid when stack popped

# Summary

- **Design principles**
  1. Simplicity favors regularity
  2. Smaller is faster
  3. Good design demands good compromises
- **Make the common case fast**
- **Layers of software/hardware**
  - Compiler, assembler, hardware
- **RISC-V: typical of RISC ISAs**
  - c.f. x86

## Homework

- 2.4, 2.8, 2.12, 2.14, 2.17, 2.22, 2.24, 2.29

END