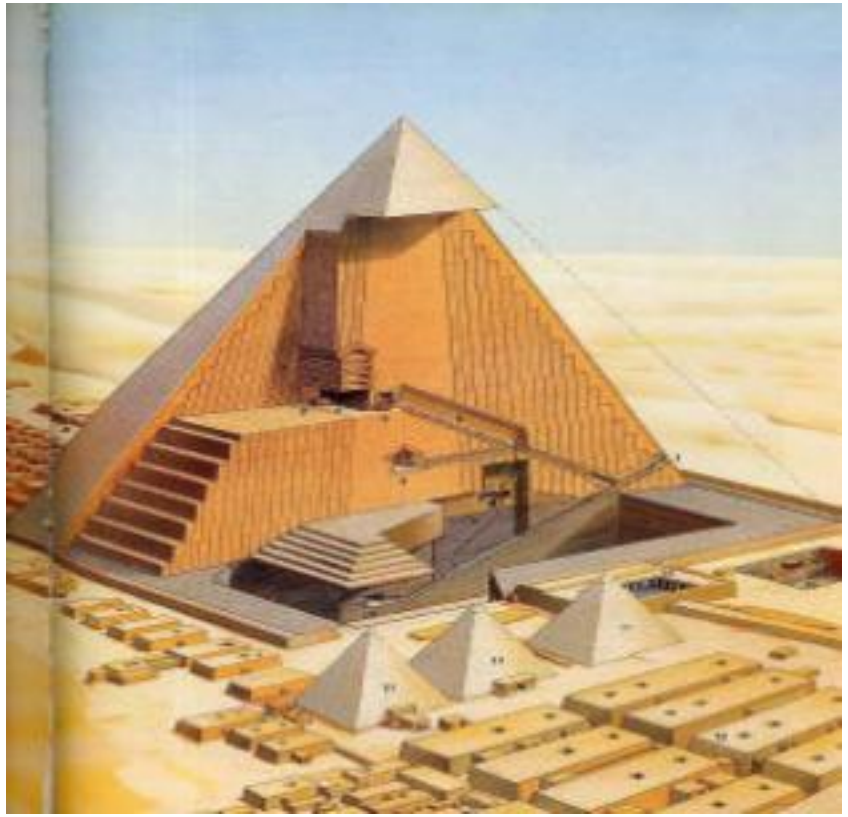
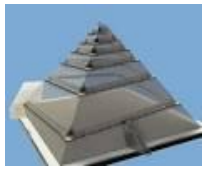
Chapter 2
Instructions: Language of the Computer





❖ The process of compiling and assembling

High-level language program (in C)

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

↓

C compiler

↓

Assembly language program (for MIPS)

```
swap:
    muli $2, $5,4
    add  $2, $4,$2
    lw   $15, 0($2)
    lw   $16, 4($2)
    sw   $16, 0($2)
    sw   $15, 4($2)
    jr   $31
```

↓

Assembler

↓

Binary machine language program (for MIPS)

```
00000000101000010000000000011000
00000000100011100001100000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```

Assembly Instruction
a symbolic representation of machine instructions

计算机科学与技术学院 College of Computer Science and Technology 中国·浙江·杭州

2.1 Introduction

- Language of the machine
  - Instructions
  - Instruction set
- *Computer* Designer goals
  - Find a language that makes it easy to build hardware and compiler.
  - Maximize performance
  - Minimize cost & energy
  - Clarity of its application
  - Simplicity: reduce design time
- Our chosen instruction set: RISC V

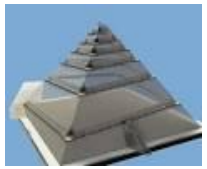
计算机科学与技术学院
College of Computer Science and Technology
中国·浙江·杭州

MIPS & ARM & RISC V

Category	RISC V	MIPS	ARM
Arithmetic	ADD x5, x6, x7	ADD $r1, $r2, $r3	ADD r1, r2, r3
	SUB x5, x6, x7	SUB $r1, $r2, $r3	SUB r1, r2, r3
Data Transfer	LW x5, 100(x6)	LW $r1, 100($r2)	LDR r1, [r2, #100]
	SW x5, 100(x6)	SW $r1, 100($r2)	SDR r1, [r2, #100]
	LH x5, 100(x6)	LH $r1, 100($r2)	LDRH r1, [r2, #100]
	SH x5, 100(x6)	SH $r1, 100($r2)	SDRH r1, [r2, #100]
Logical	AND x5, x6, x7	AND $r1, $r2, $r3	AND r1, r2, r3
	SLL x5, x6, x7	SLL $r1, $r2, 10	LSL r1, r2, #10
Conditional branch	BLT x5, x6, Lable	SLT $r1, $r2, $r3	CMP r1, r2
	BEQ x5, x6, 100	Beq $r1, $r2, Lable	BEQ Lable
	BNE x5, x6, 100	Bne $r1, $r2, Lable	
Unconditional branch	BEQ x0, x0, Lable	J Lable	B Label
	JAL x1, 100	JAL Lable	BL Label
	JALR x1, 100(x5)	JR $ra	Mov PC LR

Von Neumann' Computer

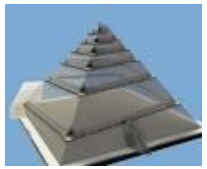
- Today's computers are built on 2 key principles: (Stored-program concept)
  - ①Instruction are represented as numbers.
  - ②Programs can be stored in memory to be read or written just like numbers.



Four Design Principles

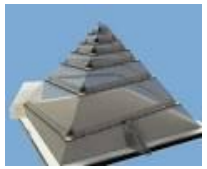
- 1. Simplicity favors regularity
- 2. Smaller is faster
- 3. Good design demands good compromises
- 4. Make the common case fast

计算机科学与技术学院
College of Computer Science and Technology
中国·浙江·杭州



- If you are asked to design the instruction set of computer, what will be the main elements ?

Contents of Chapter 2

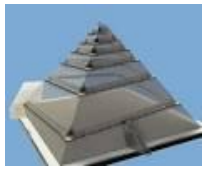


Instruction Set

- Important design principles when defining the instruction set architecture (ISA):
  - keep the hardware simple – the chip must only implement basic primitives and run fast
  - keep the instructions regular – simplifies the decoding/scheduling of instructions

Instruction characteristics

- ❖ Type of internal storage in processer （Stack/Accu/GP register)
- ❖ The number of the memory operand in the instruction (0 ~ 3)
- ❖ Operations in the instruction Set
- ❖ Type and Size of Operands
- ❖ Representation in the Computer
  - ➢ Encoding



Type of internal storage in processer

- Stack
- Accumulator
- General purpose register
  - Register-Memory
  - **Register-Register: load/store**

The number of the memory operand In the instruction

- Register-Register
  - Maximum number of operands allowed 3
  - Number of memory addresses is 0
- Register-memory
  - Maximum number of operands allowed 2
  - Number of memory addresses is 1
- Memory-memory
  - Maximum number of operands allowed 2 or 3
  - Number of memory addresses is 2 or 3

计算机科学与技术学院
College of Computer Science and Technology
中国·浙江·杭州

Variables difference

- C
  - Int i; char f;
- Instruction Set
  - Register
  - Memory address
    - Displacement
    - Immediate
  - Stack

2.2 Operations of the Computer Hardware

- **Every computer must be able to perform arithmetic:**
  - Only one operation per instruction
  - Exactly three variables(add a, b, c # a=b+c)
- **Design Principle 1**
  - ***Simplicity favors regularity(简单源自规整，指令包含3个操作数）***

- **Example(p65)** Compiling two C assignment statements into RISC-V
  - **C code:**

    a = b + c;

    d = a – e;

  - RISC-V code:

    add a, b, c

    sub d, a, e

计算机科学与技术学院 College of Computer Science and Technology 中国·浙江·杭州

❖ Example Compiling a complex C assignment

➢ C code:

f = (g + h) – (i + j);

◆ 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 operands
	subtract	sub a,b,c	a←b-c	Always three operands

计算机科学与技术学院 College of Computer Science and Technology 中国·浙江·杭州

2.3 Operands of the Computer Hardware

- ❖ **Register Operands**
- ❖ **Memory Operands**
- ❖ **Constant or Immediate Operand**

Register Operands

- **Arithmetic instructions operands must be registers or immediate**
  - 32 registers in RISC-V
  - 64 bits for each register in RISC-V
- **Design Principle 2**
  - ***Smaller is faster***（*越少越快，寄存器个数一般不超过32个*）
- **RISC-v register operand**
  - Size is 64 bits, which named ***doubleword*** （***we use 32 bits***）

计算机科学与技术学院 College of Computer Science and Technology 中国·浙江·杭州

RISC-V register conventions

Name	Register name	Usage	Preserved On call?
x0	0	The constant value 0	n.a.
x1(ra)	1	Return address(link register) (ra)	yes
x2(sp)	2	Stack pointer (sp)	yes
x3(gp)	3	Global pointer (gp)	yes
x4(tp)	4	Thread pointer (tp)	yes
x5-x7	5-7	Temporaries (t0~t2)	no
x8-x9	8-9	Saved (s0~s1)	yes
x10-x17	10-17	Arguments/results (a0~a7)	no
x18-x27	18-27	Saved (s2~s11)	yes
x28-x31	28-31	Temporaries (t3~t6)	no

MIPS Register Conventions

R0	$0	Constant 0
R1	$at	Reserved Temp.
R2	$v0	Return Values
R3	$v1	
R4	$a0	Procedure arguments
R5	$a1	
R6	$a2	
R7	$a3	
R8	$t0	Caller Save Temporaries: May be overwritten by called procedures
R9	$t1	
R10	$t2	
R11	$t3	
R12	$t4	
R13	$t5	
R14	$t6	
R15	$t7	

R16	$s0	Callee Save Temporaries: May not be overwritten by called pro-cedures
R17	$s1	
R18	$s2	
R19	$s3	
R20	$s4	
R21	$s5	
R22	$s6	
R23	$s7	
R24	$t8	Caller Save Temp
R25	$t9	
R26	$k0	Reserved for Operating Sys
R27	$k1	
R28	$gp	Global Pointer
R29	$sp	Stack Pointer
R30	$s8	Callee Save Temp
R31	$ra	Return Address

RISC-V operands

Name	Example	Comments
32 registers	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 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.

Example(p67) Compiling a C statement using registers，suppose f,g,h,i,j are assigned x19~x23

❖ C code

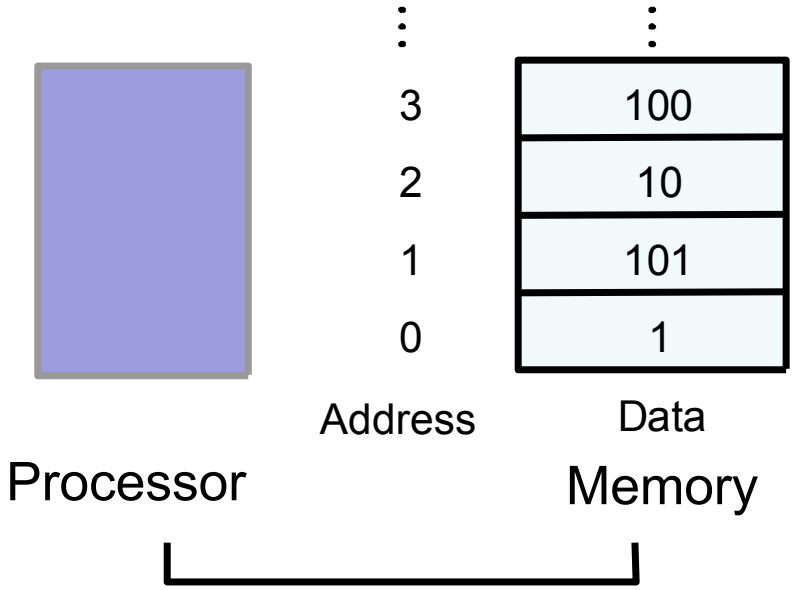
```
f = ( g + h ) – ( i + j ) ;
```

❖ RISC-V code

```
add   x5, x20, x21    // register x5 contains g + h
add   x6, x22, x23    // register x6 contains i + j
sub   x19, x5, x6     // f gets x5 – x6, which is ( g+h)–(i+j)
```


计算机科学与技术学院 College of Computer Science and Technology 中国·浙江·杭州

Memory Operands



- **Advantage**
  - Could save much more data
  - Save complex data structures
    - Arrays and structures
- **Data transfer instructions**
  - Load: from memory to register; load world/doubleword (lw/ld)
  - Store: from register to memory; store world/doubleword(sw/sd)
- **Memory addresses and contents at those locations**

计算机科学与技术学院 College of Computer Science and Technology 中国·浙江·杭州

Memory Operands

- 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(or x86) does not require words to be aligned in memory
  - Unlike some other ISAs (MIPS)

Endianness/byte order

- Big endian：
  - 数据的高字节存放在低地址；
  - 数据的低字节存放在高地址
  - PowerPC
- Little endian：
  - 数据的高字节存放在高地址；
  - 数据的低字节存放在低地址
  - RISC-V
- E.g.：32位机器上存放0x12345678，其大小端模式存储如下：

Memory Alignment: （faster data transfer）
each variable stores at a word address

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

正确

e			
Unused	D[2]	D[1]	D[0]
Unused	Unused	C[1]	C[0]
Unused	Unused	Unused	b
a			

错误

Unused	e		
e	D[2]	D[1]	D[0]

错误

因为内存一次只能读出4字节内存中的一行

这样布局，e变量不能一次读出

Data Transfer instruction

- The format of a load instruction:

destination register

source address

lw $t0, 8($t3) ; $t0 ← MEM[$t3 + 8]

any register

a constant that is added to the register in brackets

Could you explain "Sw $t0, 8($t3)" ?

Example(p69) Compiling with an operand in memory

C code:

g = h + A[8] ; // A is an array of 100 doublewords

(Assume: g ---- x20 h ---- x21

base address of A ---- x22)

RISC-V code:

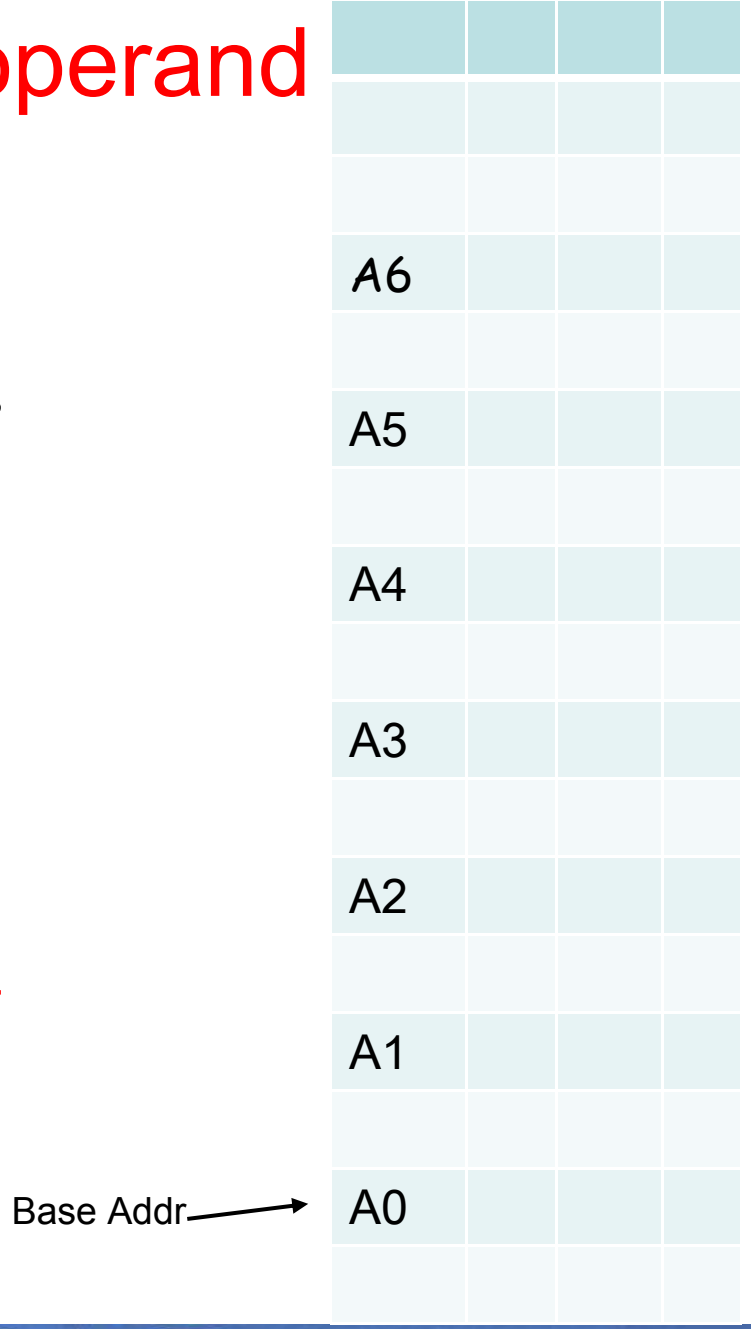
```
ld    x9, 64(x22)     // temporary reg x9 gets A[8]
add   x20, x21, x9    // g = h + A[8]
```

- Offset:

  the constant in a data transfer instruction → 64

- Base register:

  register added to form the address →64(x22)



计算机科学与技术学院 College of Computer Science and Technology 中国·浙江·杭州

Example(p71) Compiling using load and store

C code:

A[12] = h + A[8] ; // A is an array of 100 double words

(Assume: h ---- x21 base address of A ---- x22)

RISC-V code:

```
ld      x9 , 64(x22)      // temporary reg x9 gets A[8]
add     x9, x21, x9       // temporary reg x9 gets h + A[8]
sd      x9, 96(x22)       // stores  h + A[8]  back into A[12]
```


计算机科学与技术学院 College of Computer Science and Technology 中国·浙江·杭州

Discussion：How to represent?

$$g = h + A[i]$$

(Assume: g, h, *i* – x18, x19, *x20* base address of A – x22)

Example 2.6: g = h + A[i]

- Example 2.6 Compiling using a variable array index
  - C code:

```
g = h + A[i];        // A is an array of 100 doublewords
```

(Assume: g, h, *i* – x18, x19, ***x20*** base address of A – x22)

  - RISC-V code:

```
add   x5, x20, x20      # temp reg x5 = 2 * i
add   x5, x5, x5        # temp reg x5 = 4 * i
add   x5, x5, x5        # temp reg x5 = 8 * i
add   x5, x5, x22       # x5 = address of A[i] (8 * i + x22)
ld    x6, 0(x5)         # temp reg x6 = A[i]
add   x18, x19, x6      # g = h + A[i]
```

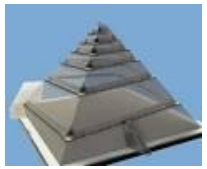

计算机科学与技术学院 College of Computer Science and Technology 中国·浙江·杭州

Registers vs. Memory

- Registers are faster to access than memory
- Operating on memory data requires loads and stores
  - More instructions to be executed
- Compiler must use registers for variables as much as possible
  - Spilling registers -- Putting less commonly used variables (or those needed later) into memory
  - Register optimization is important!

What if many variables ?

- Spilling registers:
  - Putting less commonly used variables (or those needed later) into memory.



Discussion： How to represent?

Constant

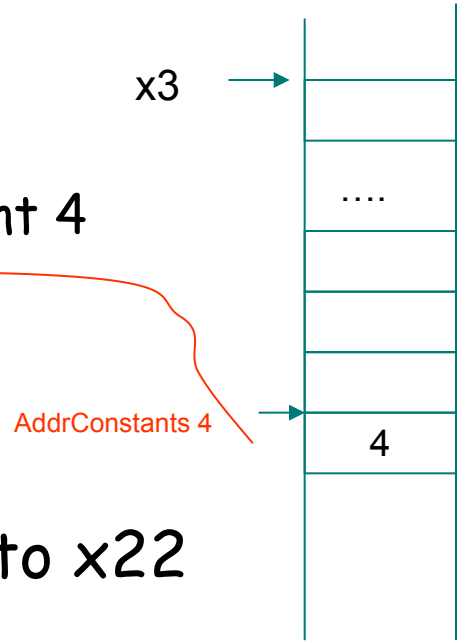
$$g = h + 55$$

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 x22
  - 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
```

功能测试程序生成常数方式非常累赘，可以用此方法代替。前提是要初始化

计算机科学与技术学院 College of Computer Science and Technology 中国·浙江·杭州

Immediate Operands

- Constant data specified in an instruction

```
addi x22, x22, 4
```

- **Design Principle 3:** Make the common case fast
  - Small constants are common
  - Immediate operand avoids a load instruction

- **Constant zero: a register x0**

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: $0 \ldots 2^n-1$

- **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?

计算机科学与技术学院
College of Computer Science and Technology
中国·浙江·杭州

Unsigned Binary Integers

- Given an n-bit number

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

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

2's-Complement Signed Integers

- Given an n-bit number

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

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

计算机科学与技术学院 College of Computer Science and Technology 中国·浙江·杭州

2's-Complement Signed Integers

- Bit 63 is sign bit in a 64-bits integer
  - 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

计算机科学与技术学院 College of Computer Science and Technology 中国·浙江·杭州

Signed Negation

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

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

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

- Example: negate 2
  - +2 = 0000 0000 … 0010_2
  - −2 = 1111 1111 … 1101_2 + 1
    = 1111 1111 … 1110_2

计算机科学与技术学院 College of Computer Science and Technology 中国·浙江·杭州

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

计算机科学与技术学院 College of Computer Science and Technology 中国·浙江·杭州

Other representations

- text characters
  - ASCII,GB2312,Unicode(UTF-8)
- Floating point numbers
  - numeric calculations
  - Different grades of precision
    - Single precision (IEEE)
    - Double precision (IEEE)
    - Quadruple precision
- Instructions in the Computer

2.5 representing Instructions in the computer

- 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 numbers 0 to 31

RISC-V instructions

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

Example(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

$0000\ 0001\ 0101\ 1010\ 0000\ 0100\ 1011\ 0011_{two} = 015A04B3_{16}$

计算机科学与技术学院 College of Computer Science and Technology 中国·浙江·杭州

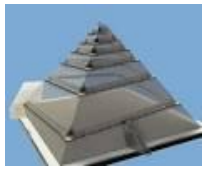
RISC-V R-format Instructions

- R-type or R-format

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

- *opcode* : basic operation and format of an instruction.
- *rd* : the register destination operand.
- *funct3*: an additional opcode field.
- *rs1*: the first register source operand.
- *rs2*: the second register source operand.
- *funct7*: an additional opcode field.

计算机科学与技术学院 College of Computer Science and Technology 中国·浙江·杭州



❖ Design Principle 4

- ➢ *Good design demands good compromises*

❖ All instructions in RISC-V have the same length

- ➢ Conflict: same length ←---→ single instruction

计算机科学与技术学院
College of Computer Science and Technology
中国·浙江·杭州

RISC-V I-format Instructions

❖ I-format

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
  - 2'-complement, sign extended

- ld x9, 64(x22)

64	22	011	9	0000011
12 bits	5 bits	3 bits	5 bits	7 bits

计算机科学与技术学院 College of Computer Science and Technology 中国·浙江·杭州

RISC-V S-format Instructions

- S-format

31–25	24–20	19–15	14–12	11–7	6–0
imm[11:0]	**rs2**	**rs1**	**funct3**	imm[4:0]	**opcode**
7bits	5bits	5bits	3bits	**5bits**	7bits

- 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
  - sd x9, 64(x22)

S_imm = {{20{inst[31]}}, Inst[31:25], inst[11:7]};

2	9	22	011	0	0100011
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

计算机科学与技术学院 College of Computer Science and Technology 中国·浙江·杭州

Summary of R-, I-, S-type instruction format

Instruction	Format	funct7	rs2	rs1	funct3	rd	opcode
add (Add)	R	0000000	reg	reg	000	reg	0110011
sub (Sub)	R	0100000	reg	reg		reg	0110011
Instruction	**Format**	**immediate**		**rs1**	**funct3**	**rd**	**opcode**
addi (Add Immediate)	I	constant		reg	000	reg	0010011
ld (Load doubleword)	I	address		reg	011	reg	0000011
Instruction	**Format**	**immed -iate**	**rs2**	**rs1**	**funct3**	**immed -iate**	**opcode**
sd (Store doubleword)	S	address	reg	reg	011	address	0100011

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    x5, 240(x10)    // temporary reg x5 gets A[30]
add   x5, x21, x5     // temporary reg x5 gets h + A[30]
addi  x5, x5, 1       // temporary reg x5 gets h + A[30] + 1
sd    x5, 240(x10)    // stores h + A[30] + 1 back into A[30]
```

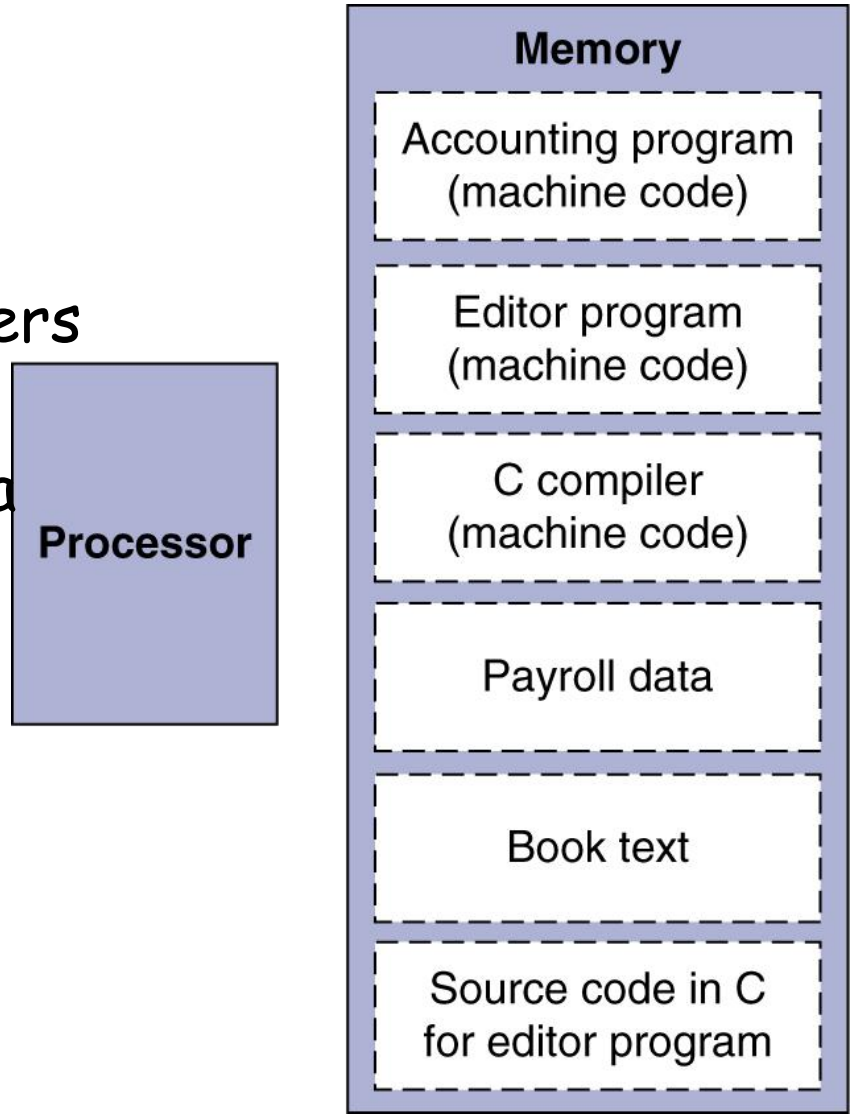
RISC-V machine language code: Decimal version

ld x9, 240(x10)	immediate		rs1	funct3	rd	opcode
	240		10	3	9	3
add x9, x21, x9	funct7	rs2	rs1	funct3	rd	opcode
	0	9	21	0	9	51
addi x9, x9, 1	immediate		rs1	funct3	rd	opcode
	1		9	0	9	19
sd x9, 240(x10)	im[11:5]	rs2	rs1	funct3	im[4:0]	opcode
	7	9	10	3	16	35

Stored-program

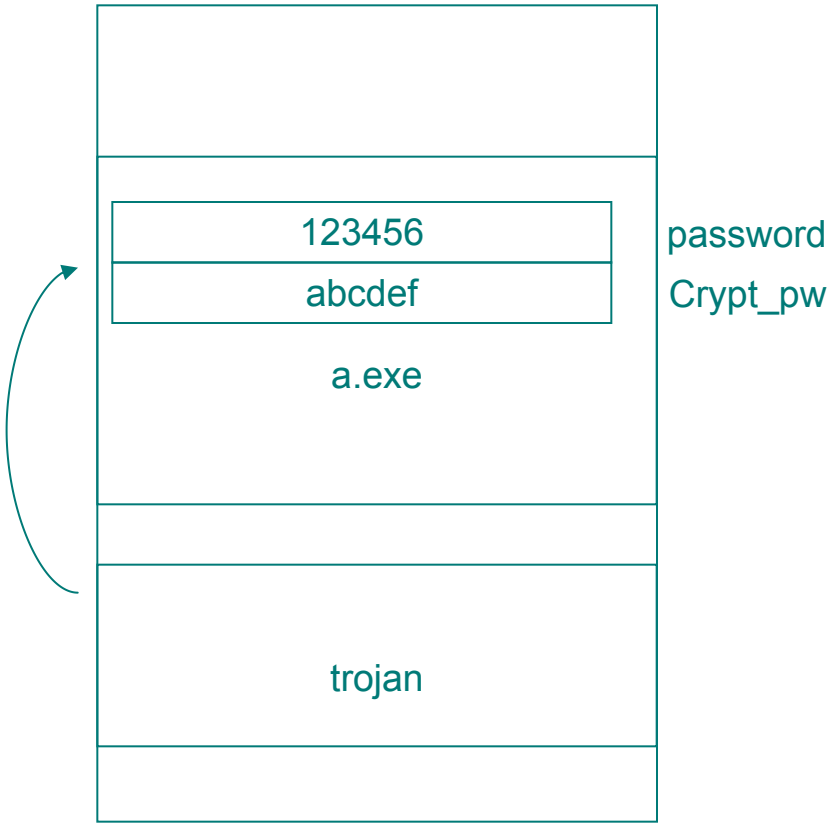
- Two key principles of today's computers
  - Instructions are represented as numbers
  - Programs can be stored in memory like numbers

- 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



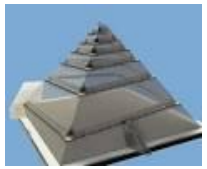
Trojan 最简单例子-密码窃取

```
main() {
    char password[256],
    crypt_pw[256];
    scanf("%s",password);
    crypt(password, crypt_pw);
    if ( strcmp(crypt_pw,
    "xxxxxx")!=0 )
        printf("password error\n");
    else
        ;
}
```



Input: 123456

Trojan read memory variable "password"



问题：如果这是一个执行程序拳击游戏，该行指令表示您挨打，血(t0)在减少(s2为负数)，当血小于等于0，KO。

用ultraedit将该二进制执行文件修改，让自己不损血，怎么改？

- 000000 | 01000 | 10010 | 01000 | 00000 | 100000
- op rs rt rd shamt funct
- add $t0, $t0 , $s2,

Memory addr: 0x40000008

00000001000100100100000000100000

```
main() {
int *p;
p=0x40000008;
*p+=2
}
```

000000 | 01000 | 10010 | 01000 | 00000 | 100010

sub $t0, $t0 , $s2

Demo: Edit the execute file and memory

- Change the program info(static)
  - Use hex edit tools
  - Eg:use ultraedit, change title
    - Leapftp -> leepftp

- Change the program info(run time)
  - Use memory edit tools
  - Eg:use 金山游侠, change data

Shell (脱壳)

一个执行程序，有判断是否为盗版，问如何crack

1. 查找判断之处，找到后看变量作用位置
2. 找到惩罚处，将该处的二进制代码改成noop

```
main()  {
      int expired=false;
      calculate expired;
      if (expired) {
          printf("软件过期了\n");
          exit(1);              -> ";"   ->   noop
      }
      else ..
}
```


计算机科学与技术学院 College of Computer Science and Technology 中国·浙江·杭州

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)

R-type Instructions	funct7	rs2	rs1	funct3	rd	opcode	Example
add (Add)	0000000	00011	00010	000	00001	0110011	add x1,x2,x3
sub (Sub)	010000	00011	00010	000	00001	0110011	sub x1,x2,x3
I-type Instructions	**immediate**		**rs1**	**funct3**	**rd**	**opcode**	**Example**
addi (Add Immediate)	001111101000		00010	000	00001	0010011	addi x1,x2, 1000
ld (Load doubleword)	001111101000		00010	011	00001	0000011	ld x1,1000 (x2)
S-type Instructions	**immed -iate**	**rs2**	**rs1**	**funct3**	**immed -iate**	**opcode**	**Example**
sd (Store doubleword)	0011111	00001	00010	011	01000	0100011	sd x1,1000(x2)

2.6 logical operations

- Instructions for bitwise manipulation
- Operating some bits within word or individual bit
- Useful for extracting and inserting groups of bits in a word

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

Shift Operations

- *Shift* operator
  - Move all the bits in a word to left or right, filling emptied bits with 0
  - Shifting left by ***i*** is same result as multiplying by 2^i

0000 0000 0000 0000 0000 0000 0000 1001 $(9)_{10}$

Shift left 4

0000 0000 0000 0000 0000 0000 1001 0000 $(9\times16=144)_{10}$

slli x11, x19, 4 // reg x11=reg x19 << 4 bit

Funct6	immediate	rs1	funct3	rd	opcode
0 Why?	4	19	1	11	19
6 bits	6 bits	5 bits	3 bits	5 bits	7 bits

计算机科学与技术学院 College of Computer Science and Technology 中国·浙江·杭州

And Operations

- *AND* operator
- Useful to mask bits in a word
  - Select some bits, clear others to 0
  - It is bit-by-bit (bitwise-AND)
    - Result=1 : both bits of the operands are 1

```
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

xori x10,x10,-1  // ==  NOT x10
```

x10	00000000	00000000	00000000	00000000	00000000	00000000	00001101	11 000000
x12	11111111	11111111	11111111	11111111	11111111	11111111	11111111	11111111
x9	11111111	11111111	11111111	11111111	11111111	11111111	11110010	00111111

计算机科学与技术学院 College of Computer Science and Technology 中国·浙江·杭州

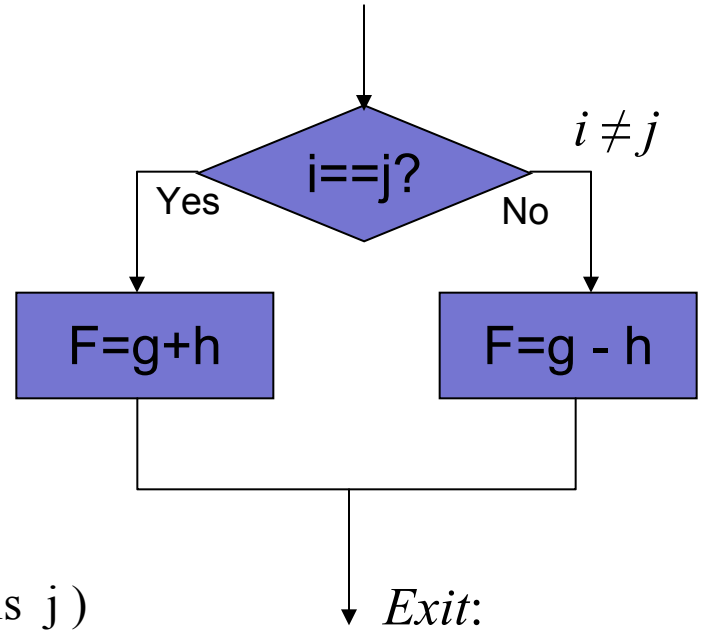
2.7 Instructions for making decisions

- Branch instructions
  - beq register1, register2, L1
  - bne register1, register2, L1
- Example Compiling an *if* statement to a branch

  (Assume: f ~ j ---- x19 ~ x23)
  - C code:

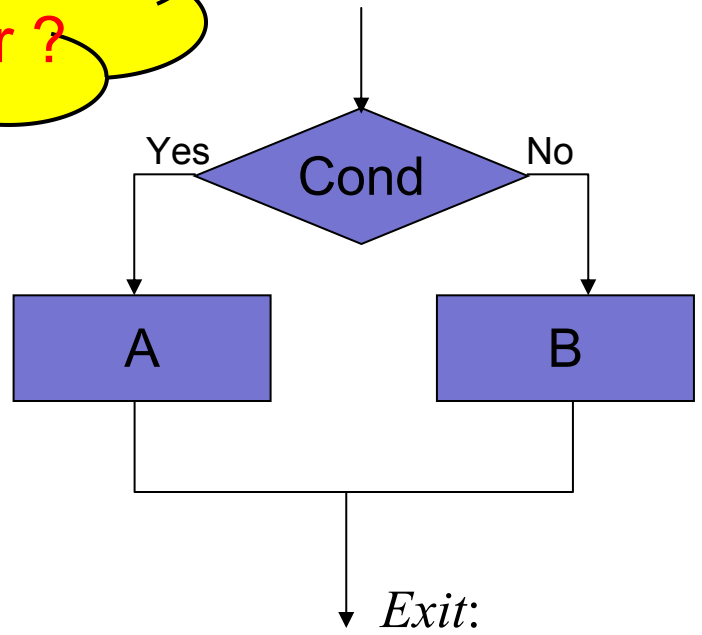
    if (i == j) f = g + h ; else f = g – h;
  - RISC-V assembly code:

```
        bne   x22, x23, ELSE   // go to ELSE if i != j
        add   x19, x20, x21    // f = g + h ( skipped if i not equals j )
        beq   x0, x0, EXIT     // as jump (unconditional branch)
ELSE:   sub   x19, x20, x21    // f = g - h ( skipped if i equals j )
EXIT:
```



Conditional branch

Which is better ?



C code1 :

```
if ( cond )  A ; else  B;
```

RISC-V assembly code:

```
        bne/beq/blt/bge    x22, x23, ELSE     // go to ELSE
            B                                 // not taken branch
        beq    x0, x0,  EXIT                  // as jump (unconditional branch)
ELSE:       A                                 // taken branch
EXIT:
```

C code 2:

```
if (! cond)  B  else  A
```

RISC-V assembly code:

```
        beq/bne/bge/blt    x22, x23, ELSE     // go to ELSE
            A                                 // not taken branch
        beq    x0, x0,  EXIT                  // as jump (unconditional branch)
ELSE:       B                                 // taken branch
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
  - C code: if (a > b) a += 1;
  - RISC V: assume a in x22, b in x23

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

Pseudo Instruction

- `BGT rs1, rs2, L1`
  - if (rs1 > rs2) goto L1
  - Blt rs2, rs1, L1
- `BLE rs1, rs2, L1`
  - if (rs1 <= rs2) goto L1
  - BGE rs2, rs1, L1

计算机科学与技术学院 College of Computer Science and Technology 中国·浙江·杭州

Compare operations

- Different compare operations required for both number types
  - **Signed integer**
    - slt： Set on less than
    - slti： Set on less than immediate
  - **Unsigned integer**
    - sltu： Set on less than
    - sltiu： Set on less than immediate

计算机科学与技术学院 College of Computer Science and Technology 中国·浙江·杭州

slt（Set on less than）instruction

- If the first reg. is less than second reg. then sets third reg to 1

  slt x2, x3, x4 # if x3 < x4 then x2=1 else x2=0

- Example: Compiling a less than test

  (Assume: a – x6 b – x7)

  - C code:

    if (a < b), goto Less

  - Use blt:

    blt x6, x7, Less

  - Use slt:

```
slt   x5, x6, x7      # x5 = 1 if x6 < x7 ( a < b)
bne   x5, x0, Less    # go to Less if x5 != 0 (that is, if a < b)
……
Less:
```


计算机科学与技术学院 College of Computer Science and Technology 中国·浙江·杭州

Example for Compare

- Register x2

  1111 1111 1111 1111 1111 1111 1111 1111

- Register x3

  0000 0000 0000 0000 0000 0000 0000 0001

- Compared Operations

  slt x1, x2, x3

  sltu x1, x2, x3

  used to generate 1

- Results

  x1 = 1 (-1 < 1)

  x1 = 0 ($4{,}294{,}967{,}295_{ten} > 1_{ten}$)

计算机科学与技术学院 College of Computer Science and Technology 中国·浙江·杭州

Bounds check Shortcut

- Reduce an index-out-of-bounds check
  - If (x20>=x11 or x20<0) goto IndexOutofBounds
  - RISC-V version:

    bgeu x20, x11, IndexOutofBounds
  - MIPS version:

    sltu $t0, $a1, $t2 ; x20 < x11

    beq $t0, $zero, IndexOutofBounds

计算机科学与技术学院 College of Computer Science and Technology 中国·浙江·杭州

Loop statements

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

  (Assume: i and k---- x22 and x23 base of save ---- x10)

  - C code:

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

  - RISC-V assembly code:

```
Loop:   slli   x28, x22, 3        // Temp reg x28 = i * 8
        add    x28, x10, x28      // Temp reg x28 = address of save[i]
        ld     x29, 0(x28)        // Temp x29 = save[i]
        bne    x29, x23, Exit     // go to Exit if save[i] != k

        addi   x22, x22, 1        // i += 1
        beq    x0, x0, Loop       // go to Loop
Exit:
```


计算机科学与技术学院 College of Computer Science and Technology 中国·浙江·杭州

Case/Switch

- used to select one of many alternatives
- Example

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

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

- 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 */
}
```

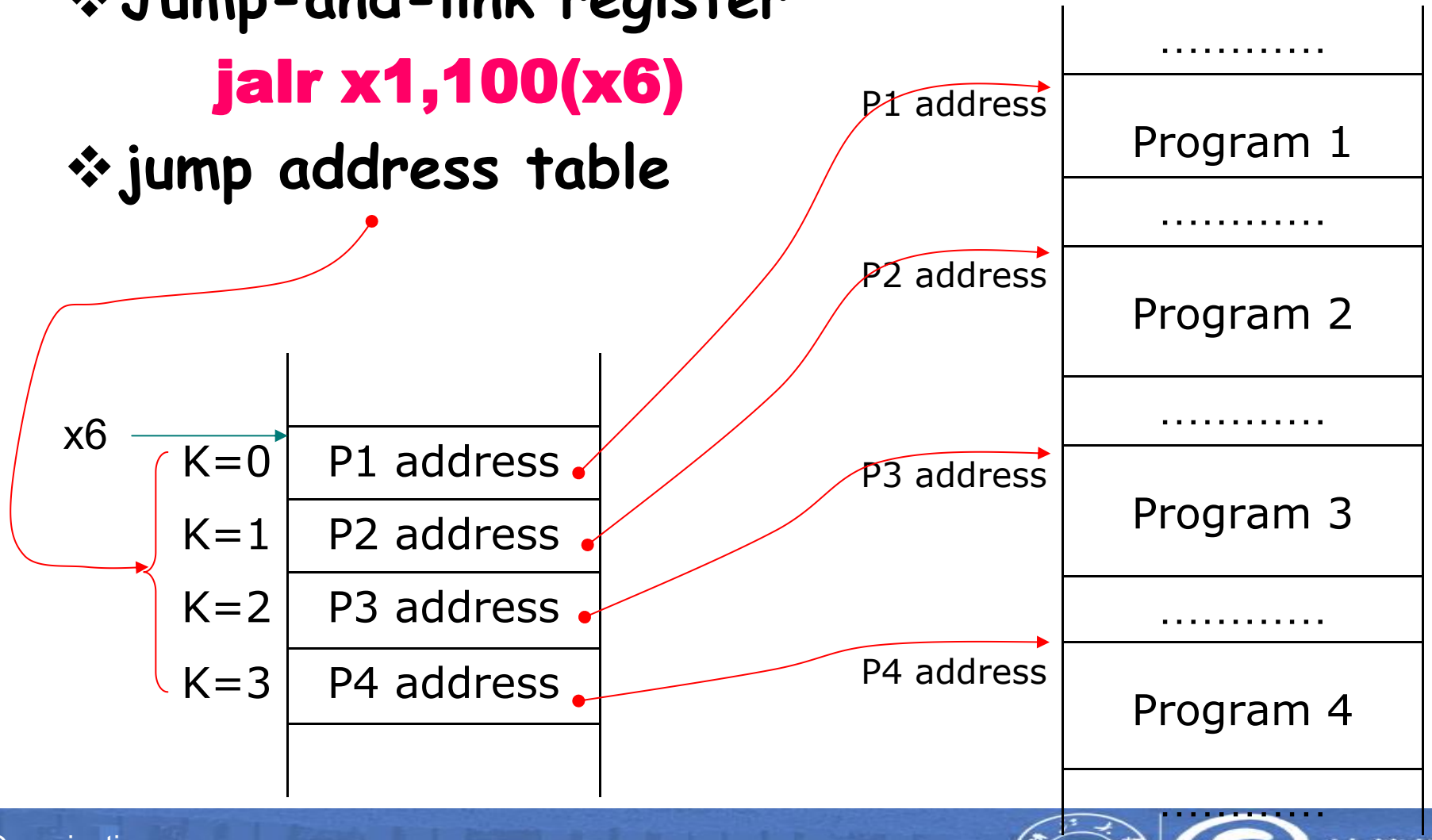

计算机科学与技术学院 College of Computer Science and Technology 中国·浙江·杭州

Jalr

- Jump-and-link register

  jalr x1,100(x6)

- jump address table



Case/Switch

- RISC-V assembly code:

```
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
add   x7, x7, x6           // x7 = address of JumpTable[k]
ld    x7, 0(x7)            // x7 gets JumpTable[k]
jalr  x1, 0(x7)            // jump entrance
Exit:
```

jump address table

L0:address

L1:address

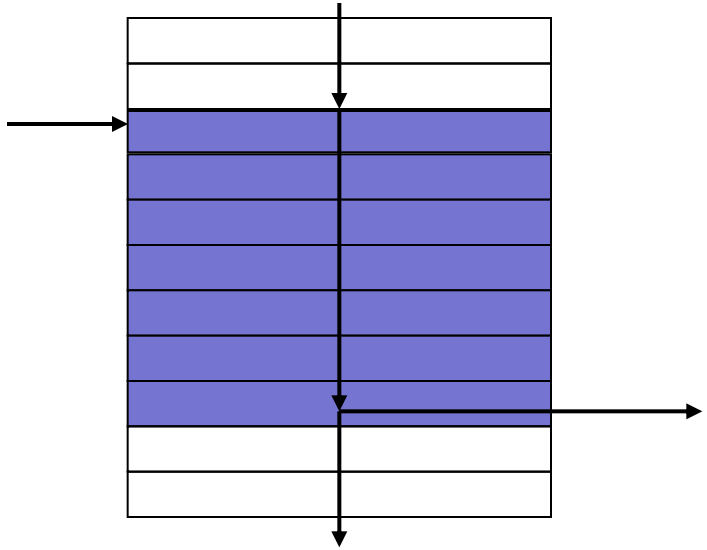
L2: address

L3:address

```
L0:  add   x20, x23, x24    // k = 0 so f gets i + j
     jalr  x0, 0(x1)        // end of this case so go to Exit
L1:  add   x20, x21, x22    // k = 1 so f gets g + h
     jalr  x0, 0(x1)        // end of this case so go to Exit
L2:  sub   x20, x21, x22    // k = 2 so f gets g - h
     jalr  x0, 0(x1)        // end of this case so go to Exit
L3:  sub   x20, x23, x24    // k = 3 so f gets i - j
     jalr  x0, 0(x1)        //end of switch statement
```

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 in Computer Hardware

- 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 steps
    1. Place Parameters in a place where the procedure can access them （in registers x10~x17）
    2. Transfer control to the procedure
    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 (address in x1)

Procedure Call Instructions

PC+4→x1

- Procedure call: jump and link

  `jal x1, ProcedureLabel`

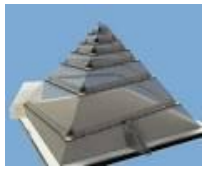
  - Address of following instruction put in x1
  - Jumps to target address
- Procedure return: jump and link register

  `jalr x0, 0(x1)`

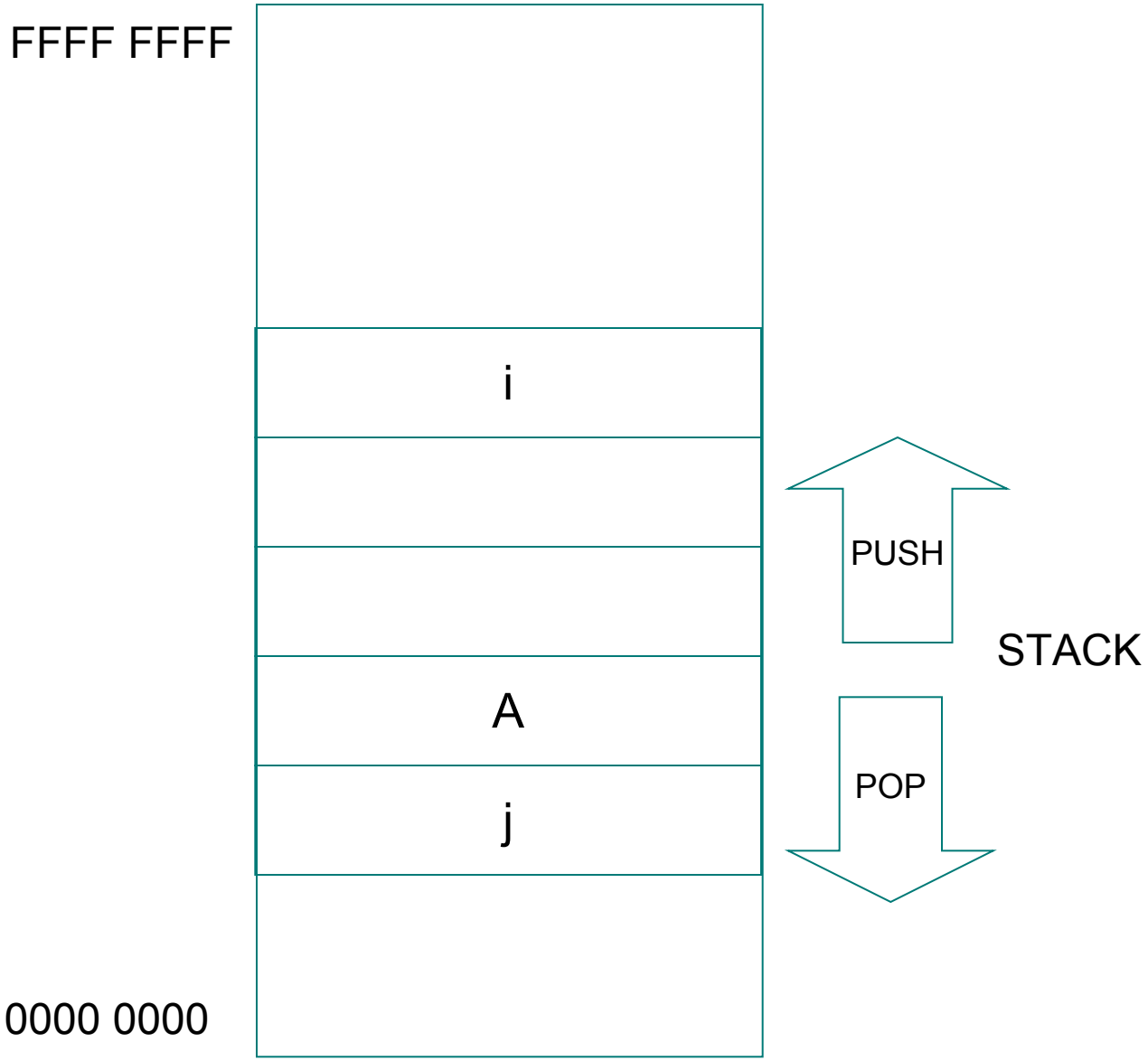
  - Like jal, but jumps to 0 + address in x1
  - Use x0 as rd (x0 cannot be changed)
  - Can also be used for computed jumps
    - e.g., for case/switch statements

Using More Registers

- Registers for procedure calling
  - x10~ x17: 8 argument registers to pass parameters or return values
  - x1: one return address register to return to origin point
- Stack: Ideal data structure for spilling registers
  - Push, pop
  - Stack pointer (sp): x2
- Stack grow from higher address to lower address
  - Push: sp= sp-8
  - Pop: sp = sp+8

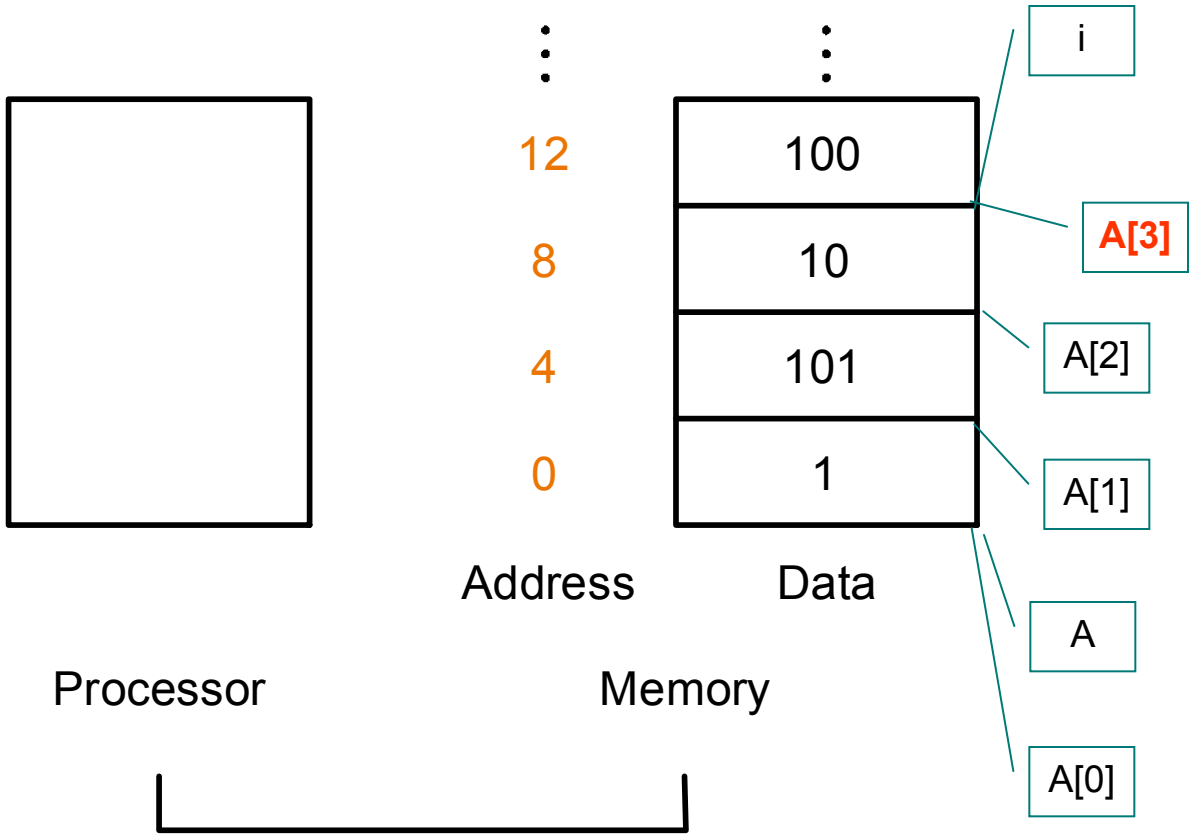


```
int i;
int A[3];
int  j;
```



```
int i,A[3], j;
...
j=A[3];
```

The result j=?



Leaf Procedure Example

- 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;
    }
```

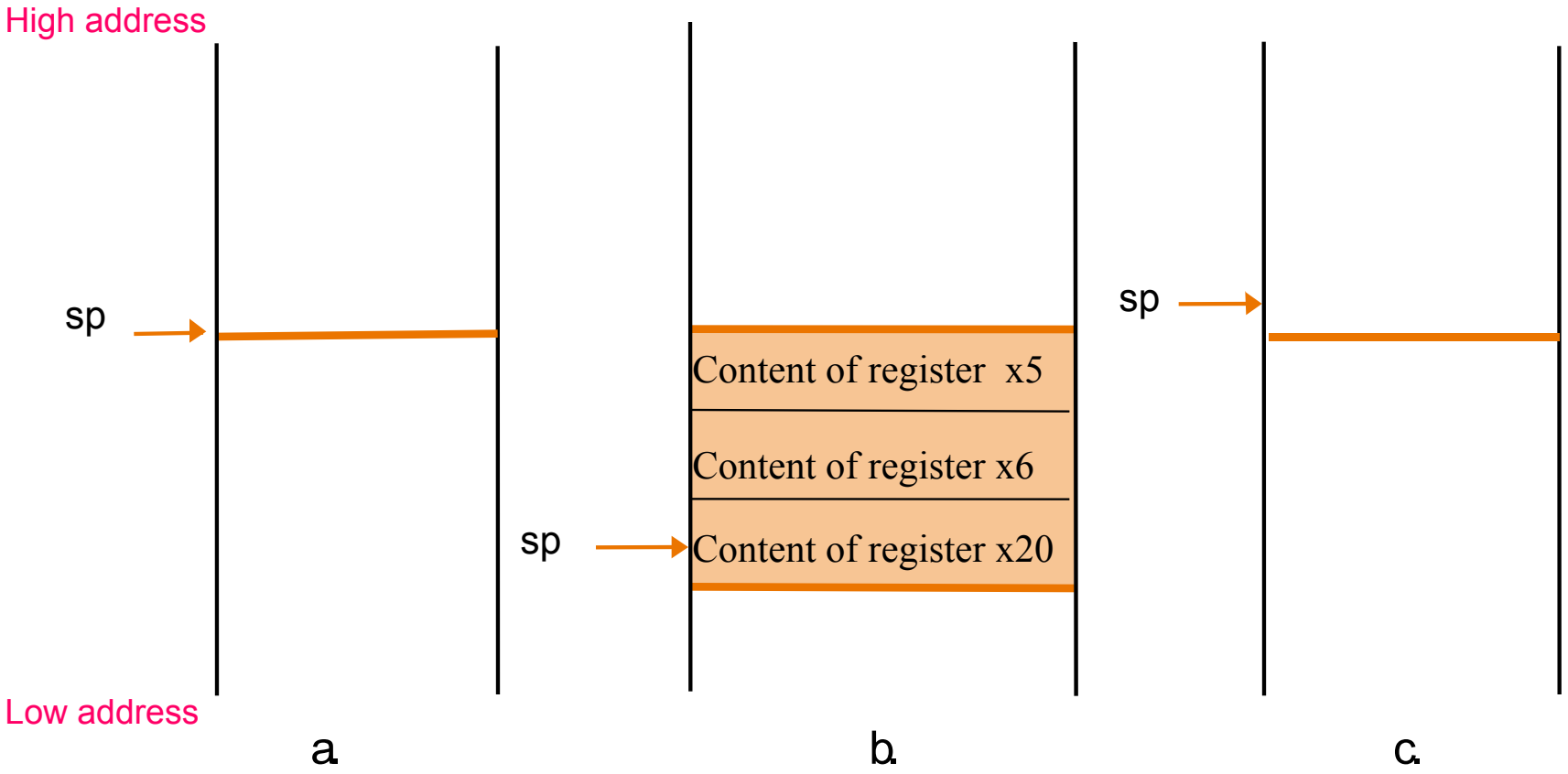
  - Arguments g, …, j in x10, …, x13
  - f in x20
  - temporaries x5, x6
  - Need to save x5, x6, x20 on stack

Leaf Procedure Example

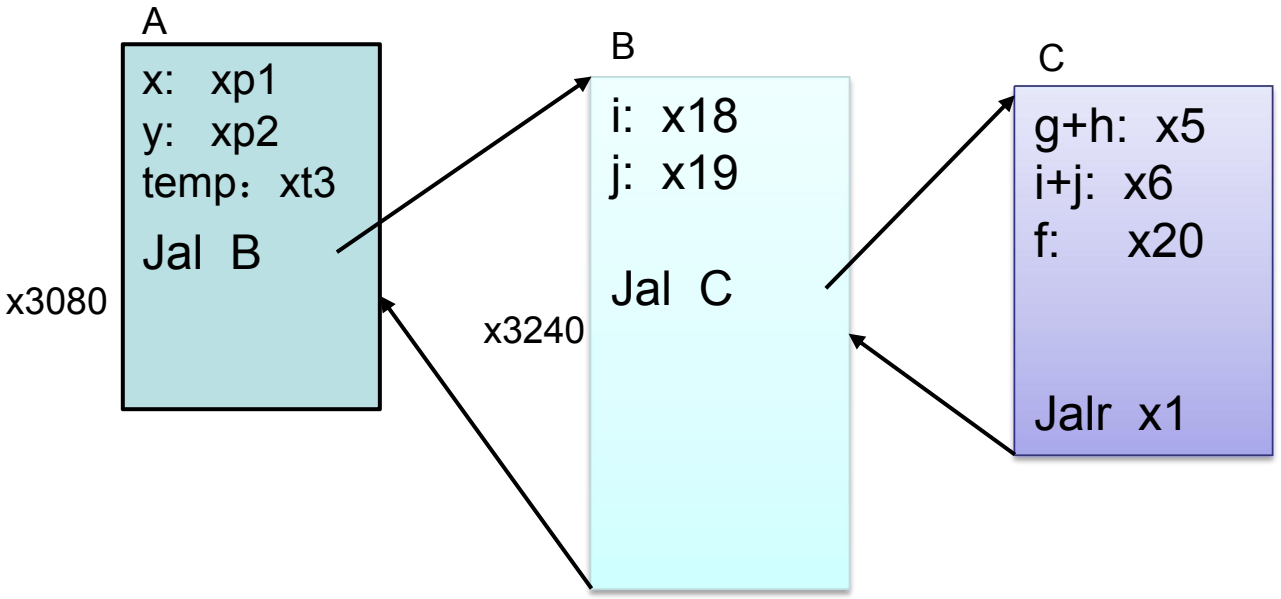
❖ RISC-V code:

```
leaf_example:
   addi sp,sp,-24
   sd   x5,16(sp)        // Save x5, x6, x20 on stack
   sd   x6,8(sp)
   sd   x20,0(sp)
   add  x5,x10,x11       // x5 = g + h
   add  x6,x12,x1        // x6 = i + j
   sub  x20,x5,x6        // f = x5 – x6
   addi x10,x20,0        //  copy f to return register
   ld   x20,0(sp)        //  Restore x5, x6, x20 from stack
   ld   x6,8(sp)
   ld   x5,16(sp)
   addi sp,sp,24
   jalr x0,0(x1)         Return to caller
```

Local Data on the Stack



How about non-leaf Procedures ?



Note: B is callee to A, but caller to C.

Non-Leaf Procedures

- Procedures that call other procedures
- For nested call, caller needs to save on the stack:
  - Its return address
  - Any arguments and temporaries needed after the call
- Restore from the stack after the call

Caller save: return address
arguments
important temporaries(**T** registers) that will be used after call

Callee save: any **S** registers used for local variables

计算机科学与技术学院
College of Computer Science and Technology
中国·浙江·杭州

Nested Procedure

- Example Compiling a recursive procedure that computes n!, suppose argument n is in x10, and results in x10

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

- RISC-V assembly code

```
fact:   addi   sp, sp, 16          // adjust stack for 2 items
        sd     x1, 8(sp)           // save the return address
        sd     x10, 0(sp)          // save the argument  n
        addi   x5, x10, -1         // x5 = n  -  1
        bge    x5, x0, L1          // if  n  >=  1, go to L1(else)
        addi   x10, x0, 1          // return 1 if n <1
        addi   sp, sp, 16          // Recover sp (Why not recover x1and x10 ?)
        jalr   x0, 0(x1)           // return to caller
```

```
L1:  addi   x10, x10, -1        // n >= 1: argument gets ( n - 1 )
     jal    x1, fact            // call fact with ( n - 1 )
     add    x6, x10, x0
     ld     x10, 0(sp)          // restore argument n
     ld     x1, 8(sp)           // restore the return address
     addi   sp, sp, 16          // adjust stack pointer to pop 2 items
     mul    x10, x10, x6        // return n*fact ( n - 1 )
     jalr   x0, 0(x1)           // return to the caller
```

- Why x10 is saved? Why x1 is saved?
- Preserved things across a procedure call
  - Saved registers, stack pointer register(sp),
  - return address register(x1), stack above the stack pointer
- Not preserved things across a procedure call
  - Temporary registers, argument registers(x10 ~ x17),
  - return value registers (x10 ~ x17), stack below the stack pointer

Register Usage

- x5 – x7, x28 – x31: temporary registers
  - Not preserved by the callee
- x8 – x9, x18 – x27: saved registers
  - If used, the callee saves and restores them

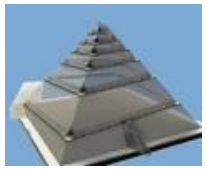
Leaf Procedure Example

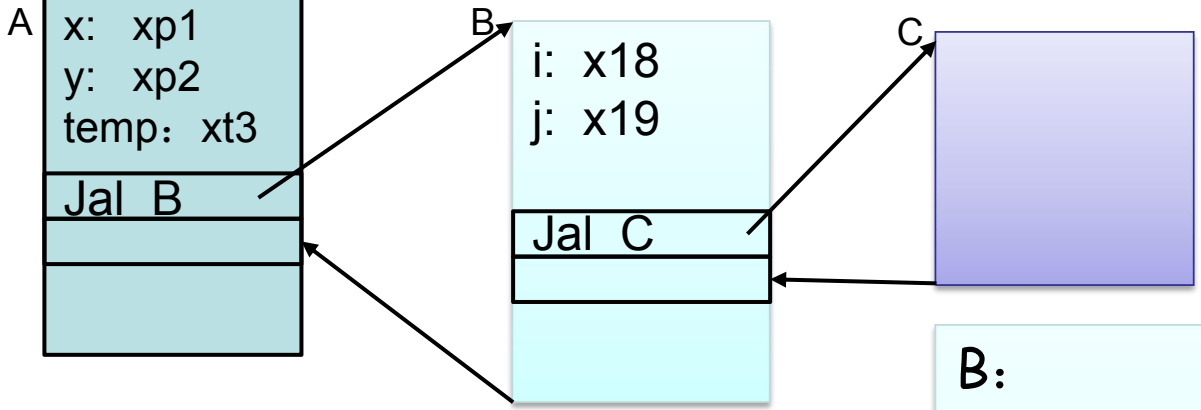
❖ RISC-V code:

```
leaf_example:
   addi sp,sp,-8
   sd   x5,16(sp)        // Save x5, x6, x20 on stack
   sd   x6,8(sp)
   sd   x20,8(sp)
   add  x5,x10,x11       // x5 = g + h
   add  x6,x12,x1        // x6 = i + j
   sub  x20,x5,x6        // f = x5 – x6
   addi x10,x20,0        //  copy f to return register
   ld   x20,0(sp)        //  Restore x5, x6, x20 from stack
   ld   x6,8(sp)
   ld   x5,16(sp)
   addi sp,sp,8
   jalr x0,0(x1)         Return to caller
```

Six steps of Function

1. Place Parameters in a place where the procedure can access them （in registers x10~x17） (before Jal in Caller)
2. Transfer control to the procedure (Jal)
3. Acquire the storage resources needed for the procedure, save the return address/parameters/callee save registers.
4. Perform the desired task
5. Place the result value in a place where the calling program can access it. Release the resources. Restored saved registers.
6. Return control to the point of origin (address in x1) (Jalr)
7. Use return value x17 (in caller)





Note: B is callee to A, but caller to C.

```
A:
move  x10,  xp1
move  x11,  xp2
move  x18,  xt3
Jal  x1, B
use  x17
move  xt3,  x18        ;restore xt3
```

caller save $t0 ~ $t6 if the content is important and not want to be modified.

```
B:
Addi  x2, x2, -40      ;apply space
Push  x1, x10, x11     ;caller save
Push  x18, x19,        ;callee save
Addi  x2, x2, …        ;space for local array
                        or structure
B-func1
Jal  x1, C             ; B-function
B-func2
Pop  x19, x18,
Pop  x11,  x10, x1
Addi  x2, x2, …        ;release space of local
Addi  x2, x2, +40      ;release ST space
Move  x17,  ?          ;return value
Jalr  x0, 0(x1)
```

Summerise of procedure (MIPS)

- The jal instruction is used to jump to the procedure and save the current PC+4 into the return address register X0
- Arguments are passed in $a0-$a3; return values in $v0-$v1
- Since the callee may over-write the caller's registers, relevant values may have to be copied into memory
- Each procedure may also require memory space for local variables – a stack is used to organize the memory needs for each procedure

Disadvantages of recursion

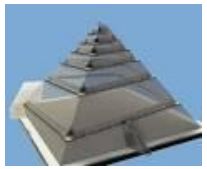
- Use too much resource，to protect the processor status，recursion may result in stack overflow.
- Need push and pop，takes a lot of memory space leading to inefficient usage of memory.
- How to avoid？ use loop instead of recursion （tail call）.

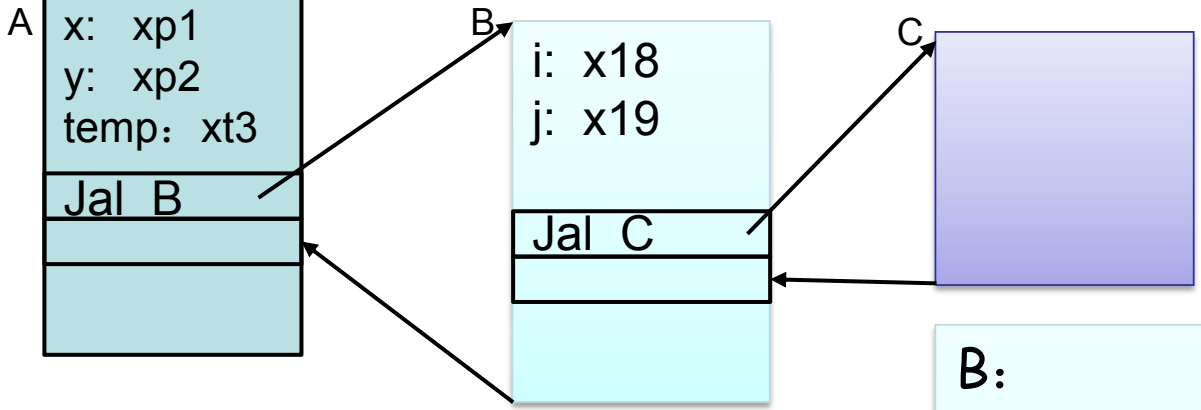
RISC-V register conventions

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

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





Note: B is callee to A, but caller to C.

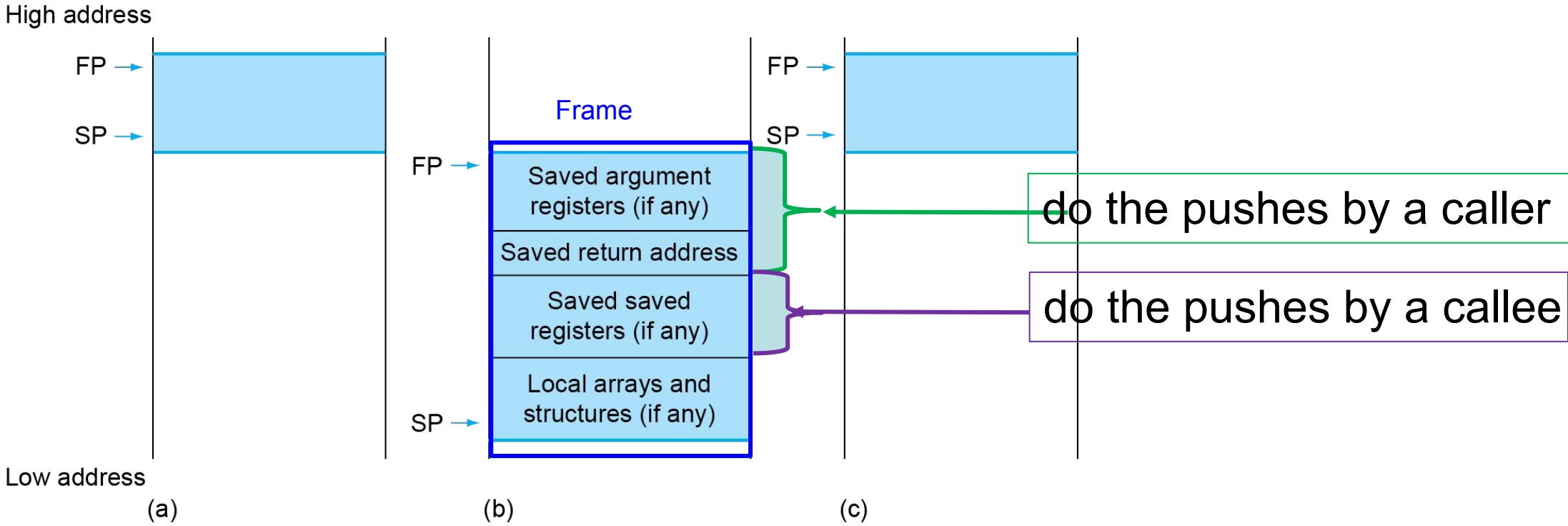
```
A:
move  x10,  xp1
move  x11,  xp2
move  x18,  xt3
Jal  x1, B
use  x17
move  xt3,  x18        ;restore xt3
```

caller save $t0 ~ $t6 if the content is important and not want to be modified.

```
B:
Addi  x2, x2, -40      ;apply space
Push  x1, x10, x11     ;caller save
Push  x18, x19,        ;callee save
Addi  x2, x2, …        ;space for local array
                        or structure
B-func1
Jal  x1, C             ; B-function
B-func2
Pop  x19, x18,
Pop  x11,  x10, x1
Addi  x2, x2, …        ;release space of local
Addi  x2, x2, +40      ;release ST space
Move  x17,  ?          ;return value
Jalr  x0, 0(x1)
```

- Storage class of C variables
  - *automatic*
  - *static*
- Procedure frame and frame pointer (x8 or fp)
  - The importance of fp
  - *automatic*
- Global pointer (x3 or gp)
  - *static*

Local Data on the Stack



- Local data allocated by callee
  - e.g., C automatic variables
- Procedure frame (activation record)
  - Used by some compilers to manage stack storage

The Concept : procedure Frame/activation record

- Procedure Frame: the segment of the stack containing a procedure's saved registers and local variables

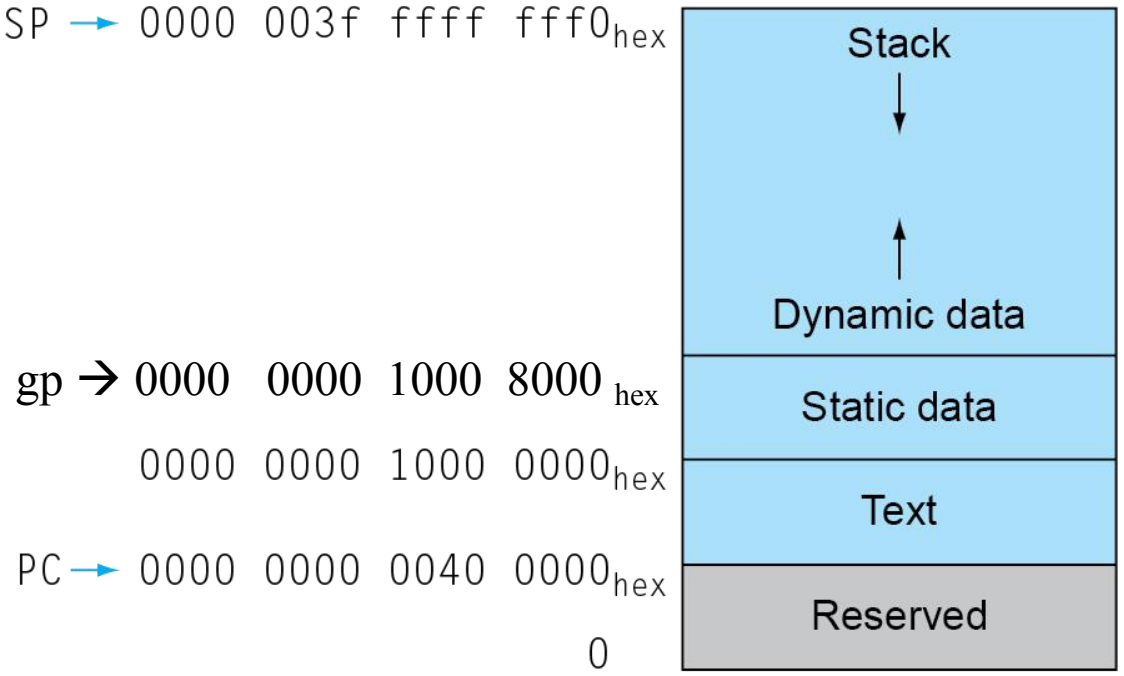
  fp (x8) -- start (bottom) sp(x2) -- end (top)

- Why fp
  - stable pointer for programmers to reference variables easily
- What's in the Frame:
  - saved argument registers
  - Saved return address [saved fp]
  - Local arrays and structures or variables

1. What fp points to in reality ?
2. What could you do if you have more parameters than 8 ?

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



TWO Bugs:

1） forget to free space

2) Free space too early

计算机科学与技术学院 College of Computer Science and Technology 中国·浙江·杭州

2.9 communication with people

- Byte-encoded character sets
  - ASCII: 128 characters
    - 95 graphic, 33 control
  - Latin-1: 256 characters
    - ASCII, +96 more graphic characters

- Unicode: 16-bit/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: Zero 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

- Three choices for representing a string
  - Place the length of the string in the first position
  - An accompanying variable has the length
  - A character in the last position to mark the end of a string

- Java uses the first choice
- C uses the third choice
  - Terminate a string with a byte whose value is 0 (null in ASCII)

String Copy Example

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

- C code: Null-terminated string 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      // address of y[ i ] in x5
         lbu   x6, 0(x5)         // x6 = y [ i ]
         add   x7, x19, x10      // address of x[ i ] in x7
         sb    x6, 0(x7)         // x[ i ] = y[ i ]
```

```
        beq   x6, x0,  L2          // if  y[ i ]  ==  0, go to L2
        addi  x19, x19, 1          //  i  =  i  +  1
        jal   x0,  L1              // go to L1
L2:     ld    x19, 0(sp)           // restore x19
        addi  sp, sp, 8            // pop 1 doubleword off stack
        jalr  x0,  0(x1)           // return
```

- Optimization for example
  - strcpy is a leaf procedure
  - Allocate i to a temporary register x28
- For a leaf procedure
  - The compiler exhausts all temporary registers
  - Then use the registers it must save

计算机科学与技术学院 College of Computer Science and Technology 中国·浙江·杭州

2.10 RISC-V Addressing for Wide Immediate & Addresses

- Most constants are small
  - 12-bit immediate is sufficient
- For the occasional 32-bit constant

```
lui rd, constant
```

imm[31:12]	rd	opcode	U-type format
20bits	5 bits	7 bits	

  - Copies 20-bit constant to bits [31:12] of rd
  - Extends bit 31 to bits [63:32]
  - Clears bits [11:0] of rd to 0

```
lui x19, 976  // 0x003D0
```

	31 … 12	11 … 7	6 … 0
Instruction	0000 0000 0011 1101 0000	10011	**011 0111**
Register	0000 0000 0011 1101 0000	00000	**0000000**

Filling zero

计算机科学与技术学院 College of Computer Science and Technology 中国·浙江·杭州

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

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

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

  The value of s3 afterward is:

0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0011 1101 0000	1001 0000 0000

- Note: Why does it need two steps?

计算机科学与技术学院 College of Computer Science and Technology 中国·浙江·杭州

Branch Addressing

- 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[31:0]

Inst[31] signextension	Inst[7]	Inst[30:25]	Inst[11:8]	1'B0

imm[31:0]

- PC-relative addressing
  - Target address = PC + branch offset

计算机科学与技术学院 College of Computer Science and Technology 中国·浙江·杭州

Jump Addressing

- Jump and link (jal) target uses 20-bit immediate for larger range
- UJ format: jal x0, 2000 //2000 = 0111 1101 0000

0	1111101000	0	00000000	00000	1101111
imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	opcode

inst[31:0]

Inst[31] signextension	Inst[19:12]	Inst[20]	Inst[30:21]	1'B0

imme[31:0]

- For 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

计算机科学与技术学院 College of Computer Science and Technology 中国·浙江·杭州

Show branch offset in machine language

- Example (p116) Show branch offset in machine language
  - C language (p94):
    - while (save[i]==k) i=i+1;

  - RISC-V assembler code :

```
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:
```


计算机科学与技术学院 College of Computer Science and Technology 中国·浙江·杭州

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	01100	1100011	01849663
	addi	80016	0000000	00001	10110	000	10110	0010011	001B0B13
	beq	80020	1111111	00000	00000	000	01101	1100011	FE0006E3
Exit:		80024	……						

-10 = 11110110

6 =0110

-20 = 80000 - 80020

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

计算机科学与技术学院 College of Computer Science and Technology 中国·浙江·杭州

- 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 (p117) Branching far away
  - Given a branch:

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

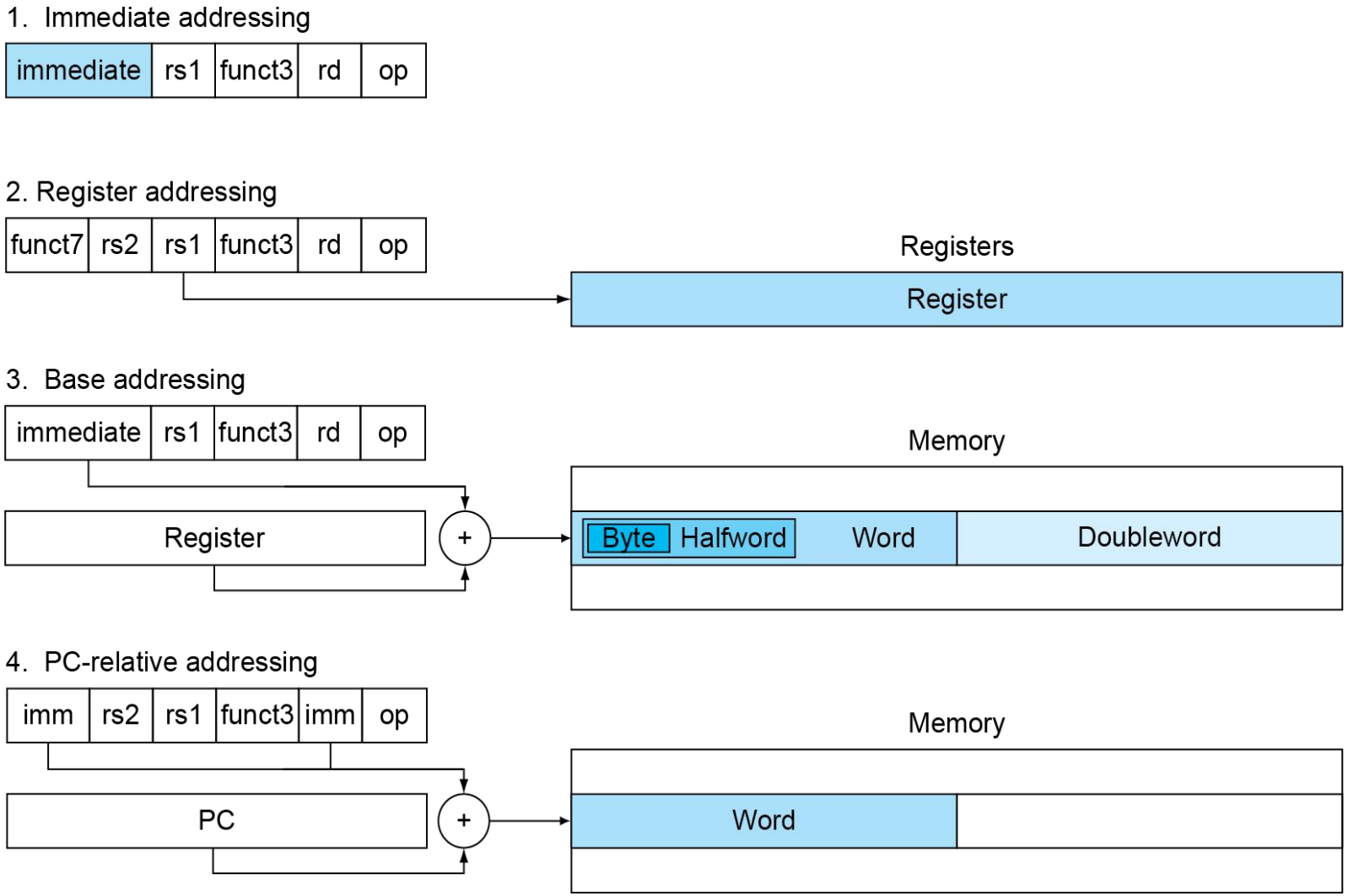
  - Rewrite it to offer a much greater branching distance:

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


计算机科学与技术学院 College of Computer Science and Technology 中国·浙江·杭州

RISC-V Addressing Summary

- Immediate addressing:
  addi x5,x6,4
- Register addressing:
  add x5,x6,x7
- Base addressing:
  ld x5,100(x6)
- PC-relative addressing:
  beq x5,x6,L1



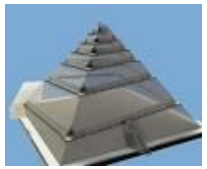
RISC-V operands

Name	Example	Comments
32 registers	$x0$-$x31$	Fast locations for data. In RISC-V, data must be in registers to perform arithmetic. Register x0 always equals 0.
2^{61} memory 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
$x0$(zero)	**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
$x5$-$x7$(t0-t2)	5-7	Temporaries	no
$x8$(s0/fp)	8	Saved/frame point	Yes
$x9$(s1)	9	Saved	Yes
$x10$-$x17$(a0-a7)	10-17	Arguments/results	no
$x18$-$x27$(s2-s11)	18-27	Saved	yes
$x28$-$x31$(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

RISC-V encoding summary

Name (Field Size)	Field 7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	Comments
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	immed[12,10:5]	rs2	rs1	funct3	immed[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

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	0000000
	or	0110011	110	0000000
	and	0110011	111	0000000
	lr.d	0110011	011	0001000
	sc.d	0110011	011	0001100

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.

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	1100011	000	n.a.
	bne	1100011	001	n.a.
	blt	1100011	100	n.a.
	bge	1100011	101	n.a.
	bltu	1100011	110	n.a.
	bgeu	1100011	111	n.a.
U-type	lui	0110111	n.a.	n.a.
UJ-type	jal	1101111	n.a.	n.a.

Decoding Machine Language

- Example (p120)
  - Machine instruction (0x00578833)

    0000 0000 0101 0111 1000 1000 0011 0011
  - Decoding
    - Determine the operation from opcode

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

funct7	rs2	rs1	funct3	rd	opcode
0000000	00101	01111	000	10000	0110011

      funct7 and funct3 are all 0 → **add instruction**
    - Determine other fields

      rs2: x5; rs1: x15; rd: x16
    - Show the assembly instruction

      add x16, x15, x5 (Note: add rd,rs1,rs2)

计算机科学与技术学院 College of Computer Science and Technology 中国·浙江·杭州

2.11 Parallelism and Instructions: Synchronization*

- cause
  - multiprocessors
  - task preemption
  - interrupt
- result
  - data race
  - resources race
  - Critical region
- solution
  - synchronization
    - mutual exclusion、semaphore …
  - hardware level
    - atomic exchange or atomic swap (instructions in RISC-V: lr.d and sc.d)

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

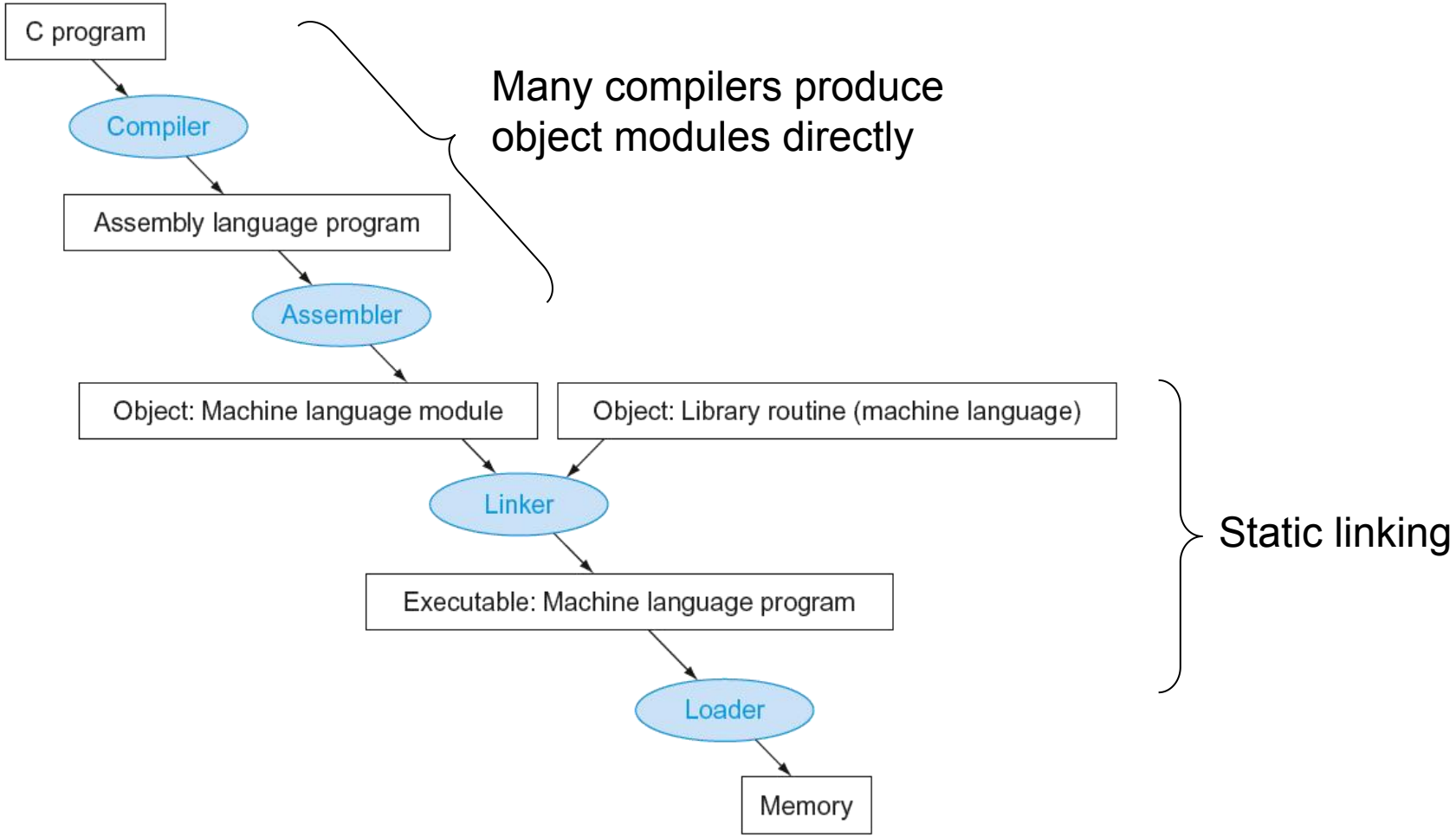
```
        addi x12,x0,1          // copy locked value
again:  lr.d x10,(x20)         // read lock
        bne  x10,x0,again      // check if it is 0 yet
        sc.d x11,(x20),x12     // attempt to store
        bne  x11,x0,again      // branch if fails
```

- Unlock:

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


计算机科学与技术学院 College of Computer Science and Technology 中国·浙江·杭州

2.12 Translating and Starting a Program



Start a C program in a file on disk to run

object file

- **Compiling**
  - C program → assembly language program
- **Assembling**
  - Assembly language program → machine language module
  - pseudoinstructions

    mv x10, x11 // register x10 gets register x11

    add x10, x11, x0
  - Symbol table
    - A table that matches name of lables to the addresses of the memory words that instructions occupy.
  - Producing an object file of UNIX (six distinct pieces)

计算机科学与技术学院 College of Computer Science and Technology 中国·浙江·杭州

Object file

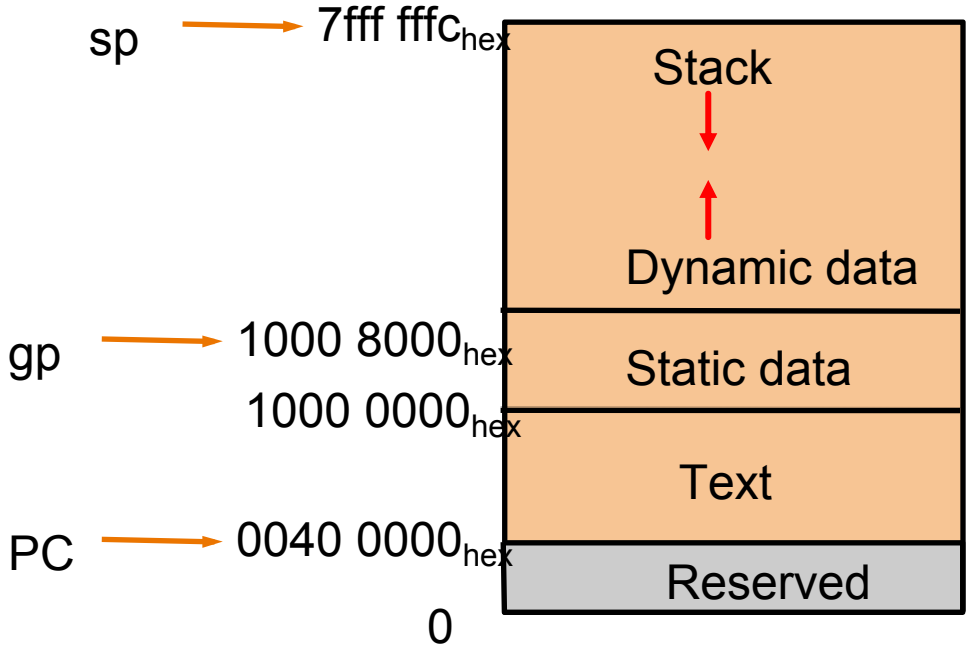
- object file header—size and position of the other pieces
- Text segment
- static data segment and dynamic data
- The relocation information ----identifies absolute addresses of instruction and data words when the program is loaded into memory
- symbol table
- Debugging information

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	**--**	

Linking Object modules

- Object modules(including library routine) → **executable program**
- 3 step 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**)
- Could leave location dependencies for fixing by a relocating loader
  - But with virtual memory, no need to do this
  - Program can be loaded into absolute location in virtual memory space

RISC-v memory allocation for program and data



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

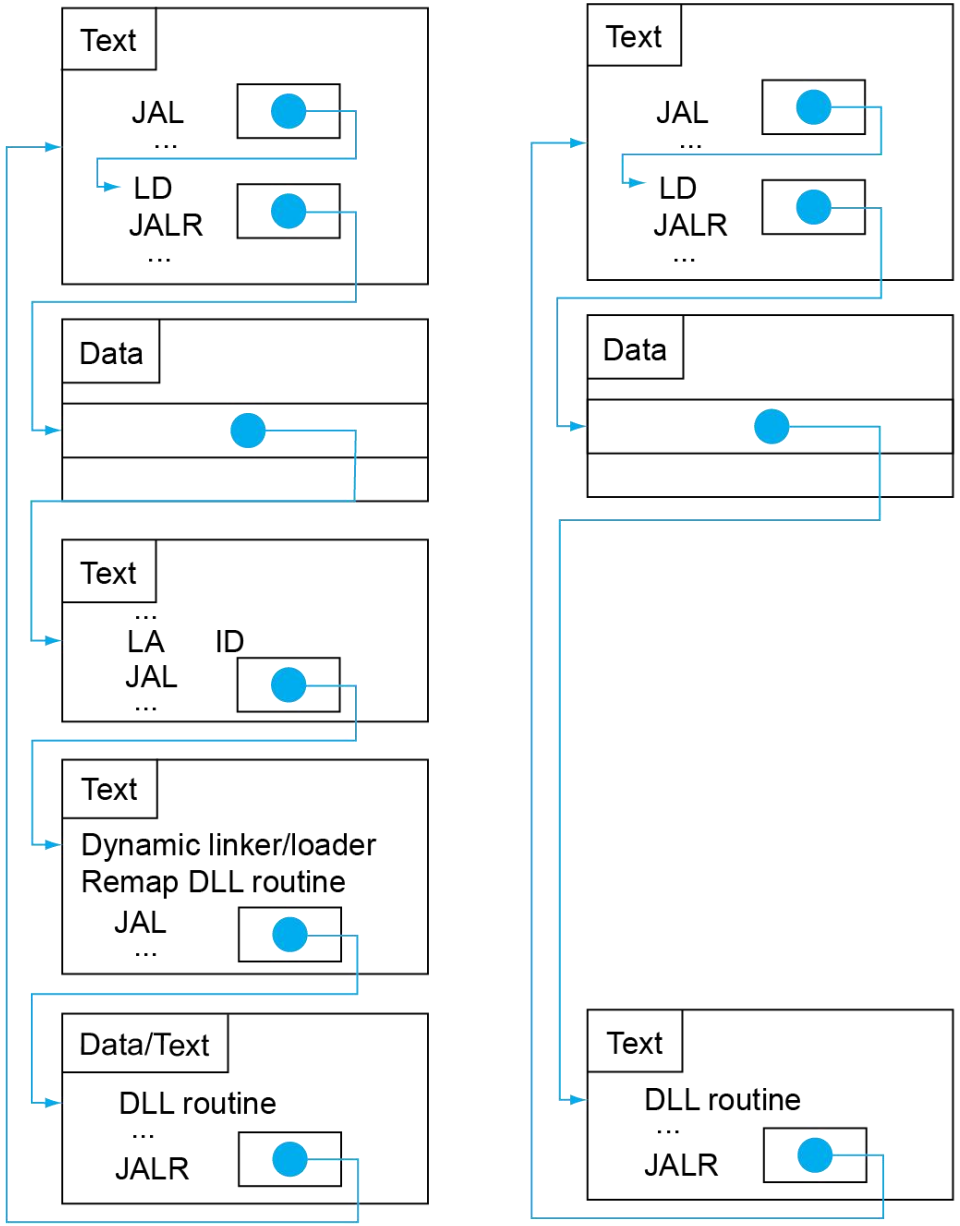
Lazy Linkage

Indirection table

Stub: Loads routine ID,
Jump to linker/loader

Linker/loader code

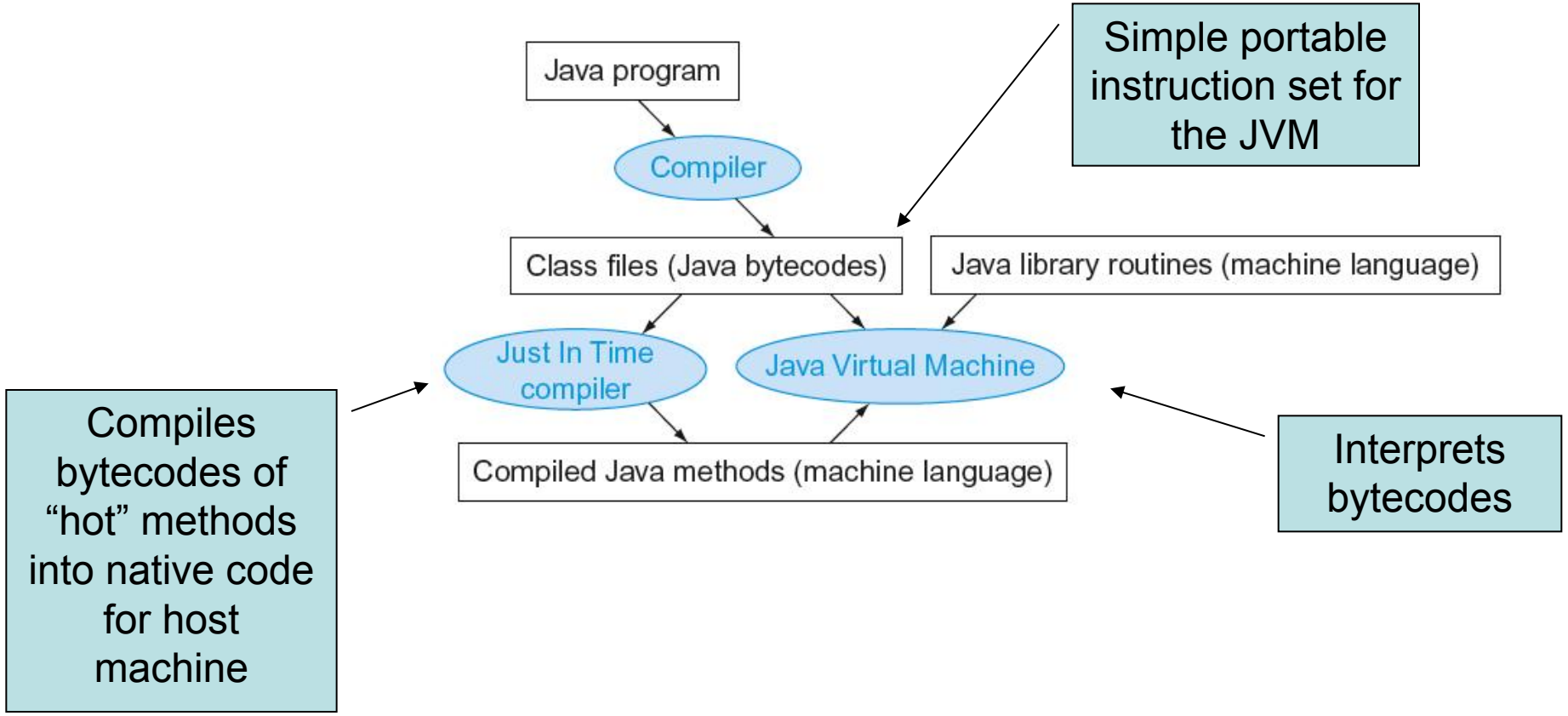
Dynamically
mapped code



(a) First call to DLL routine

(b) Subsequent calls to DLL routine

Starting Java Applications



执行文件与进程

- 执行文件
  - 在硬盘上，非执行态
  - 如病毒样本
- 进程
  - 转载到内存
  - 可以细分为多个可以并发执行的线程
  - 如激活态病毒
  - 如何看线程: 任务管理器

计算机任何动作都是程序设计的

- 病毒
  - 组成：感染能力（自我复制），隐藏，破坏能力
  - 一个简单的
    - 感染：看到一个执行程序，就把病毒程序拷贝到执行程序最后，程序启动时加一条无条件跳转指令，跳到病毒处，病毒程序执行完成跳回首地址后
    - 隐藏：把执行程序拆封为n个，运行时组装为一体
    - 破坏：随便
- 程序都需要消耗内存
  - System idle 也是程序，也消耗内存

EXE文件加密

- 输入密码才能执行该文件？
- 普通的方法，先读密码，验证后判断是否继续
  - 问题：简单的修改执行文件即可破解
- 一般方法：
  - 第一段代码是密码验证
  - 若通过，则解密后续代码
    - 解密后长度=加密前，简单
    - 随便压缩
- 高级方法：
  - 分段加密，前一段的中间结果作为后续的解密用的密钥

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 lon temp ;
    temp = v[ k ] ;
    v[ k ] = v[ k + 1 ] ;
    v[ k + 1 ] = temp ;
}
```


计算机科学与技术学院 College of Computer Science and Technology 中国·浙江·杭州

- Register allocation for *swap*

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

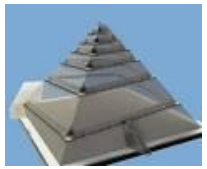
- *swap* is a leaf procedure, nothing to preserve
- RISC-V code for the procedure *swap*
  - **Procedure body**

```
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
```


计算机科学与技术学院 College of Computer Science and Technology 中国·浙江·杭州

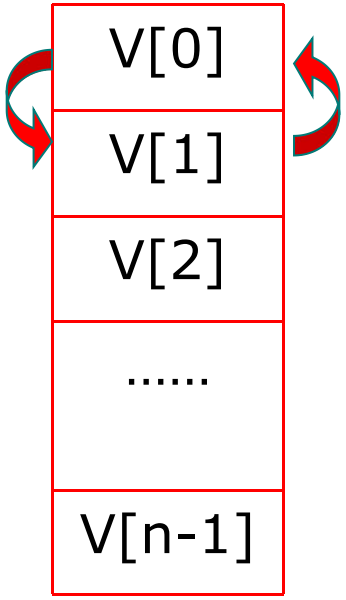


❖ Procedure *sort*

- 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 ) ;
    }
}
```

If V[0]> V[1]



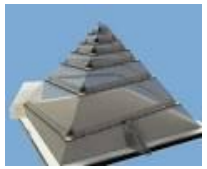
- **Register allocation** for *sort*

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

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

  x1 , x19, x20, x21, x22

计算机科学与技术学院 College of Computer Science and Technology 中国·浙江·杭州



➢ RISC V Code for the procedure *sort*

- **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
```

- **Code for Procedure body**
  - **Outer loop—first for loop**

    for (i = 0 ; i < n ; i += 1) {

Move parameters

```
mv   x21, x10        //  copy parameter x10 into x21
mv   x22, x11        //  copy parameter x11 into x22
```

Outer loop

```
        li  x19, 0          //   i = 0
for1tst:  bge  x19, x22, exit1      // go to exit1 if i >= n
        ……
```

（body of first for loop is second *for* loop）

```
        ……
exit2:  addi  x19, x19, 1        #  i = i + 1
        j     for1tst            #  jump to test of outer loop
exit1:
```


计算机科学与技术学院 College of Computer Science and Technology 中国·浙江·杭州

– **Inner loop-- second *for* loop is body of first *for* 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  j < 0
            slli  x5, x20, 3            // x5 = j * 8
            add   x5, x21, x5           //  x5 = the address of v[j]
            ld    x6, 0(x5)             //  x6 = v[j]
            ld    x7, 8(x5)             //  x7 = v[j + 1]
            blt   x6, x7, exit2         //  go to exit2 if v[j] < v[j+1]
```

……

（body of first *for* loop ）

……

```
            addi  x20, x20, -1          //  j = j - 1
            j     for2tst               //  jump to test of inner loop
exit2:
```

– **body of first *for* loop**

Pass parameters and call

```
mv  x10，x21        // first swap parameter v
mv  x11，x20        // second swap parameter j
```

Call function swap

```
jal   x1, swap
```

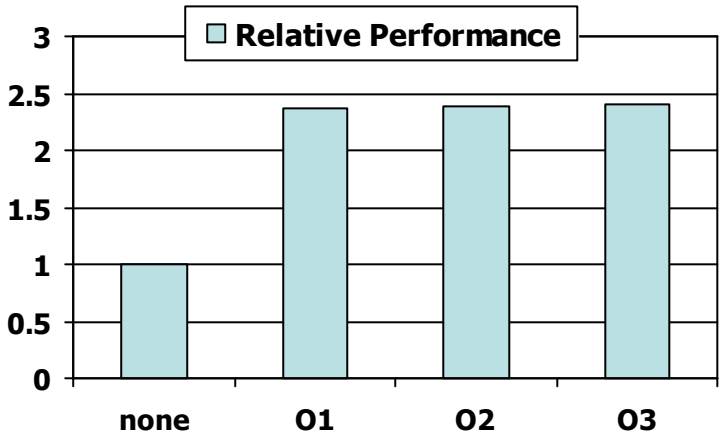
- Notice:
  1.Why are x10 and x11 saved?
  x10 is the base of the array v. x10 will be used repeatedly and might be(actually not here) changed by the procedure swap.
  x11 is the size of the array v. x11 will be used repeatedly and changed before the procedure swap is called.

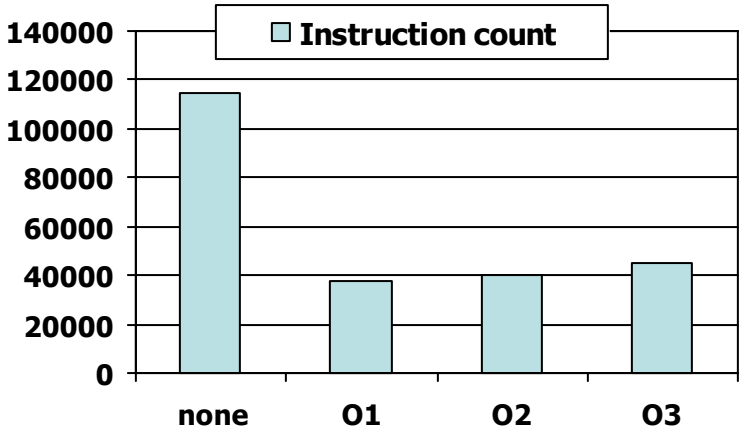
- 2.Why are they not pushed to stack?
  - Register variable is faster

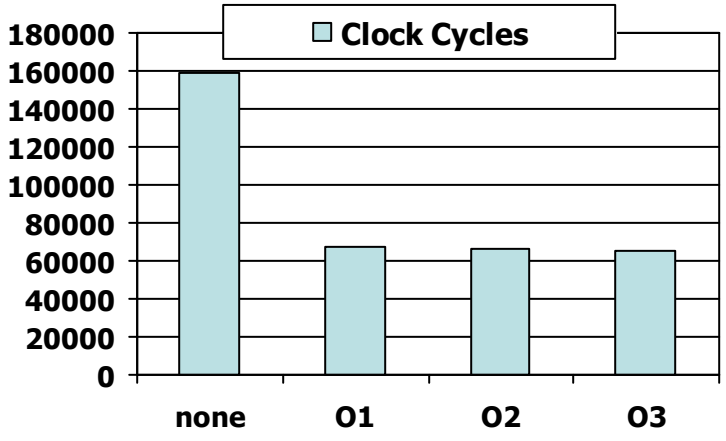
计算机科学与技术学院 College of Computer Science and Technology 中国·浙江·杭州

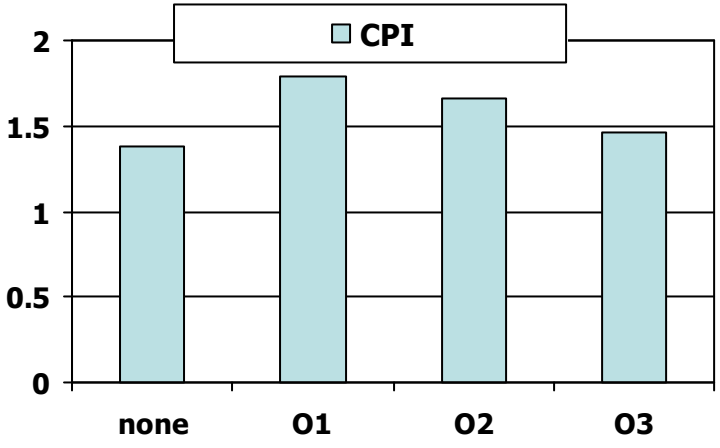
Effect of Compiler Optimization

Compiled with gcc for Pentium 4 under Linux

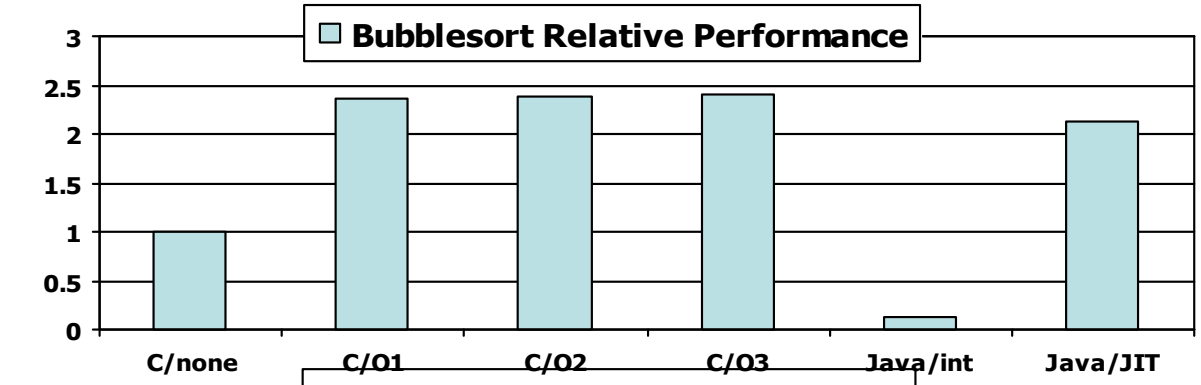


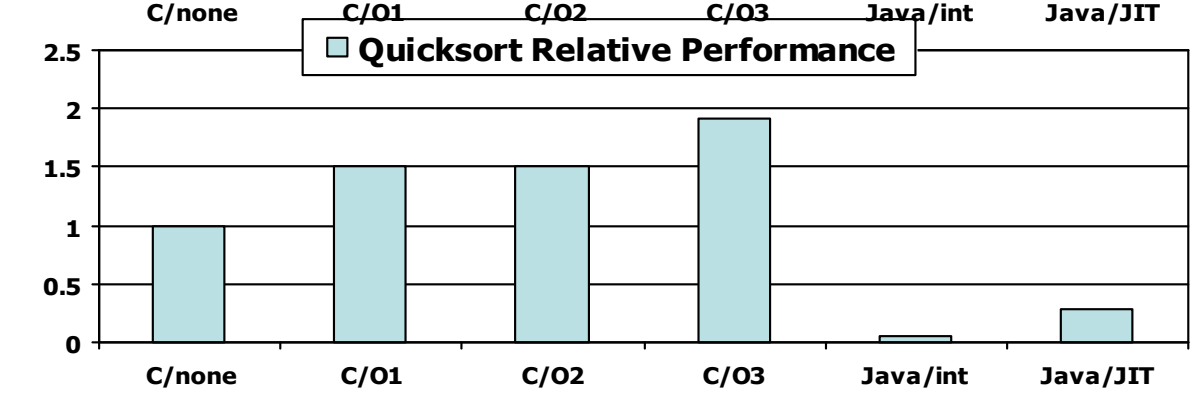


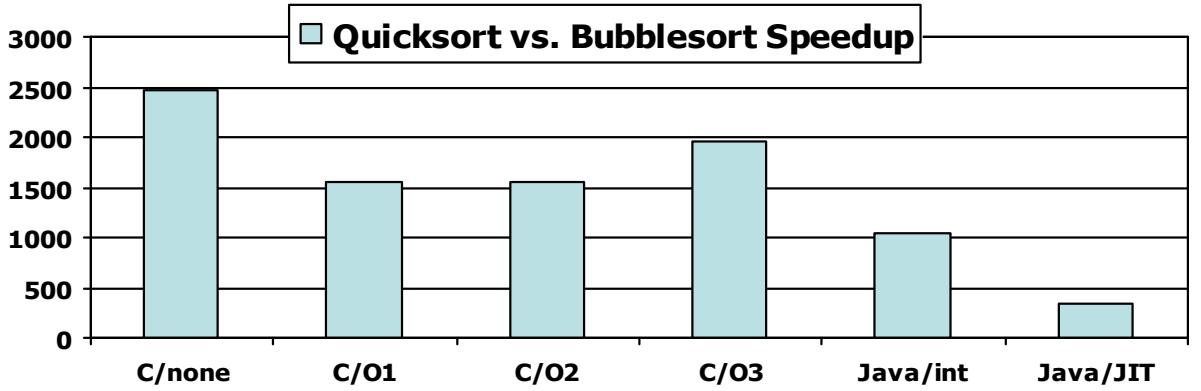




Effect of Language and Algorithm







Lessons Learnt

- Instruction count and CPI are not good performance indicators in isolation
- Compiler optimizations are sensitive to the algorithm
- Java/JIT compiled code is significantly faster than JVM interpreted
  - Comparable to optimized C in some cases
- Nothing can fix a dumb algorithm!

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;
}
```

```
   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
```

This code works as long as *size* is greater than 0.

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

```
   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
```

This code works as long as *size* is greater than 0.

Comparison of Array vs. Ptr

- 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

计算机科学与技术学院 College of Computer Science and Technology 中国·浙江·杭州

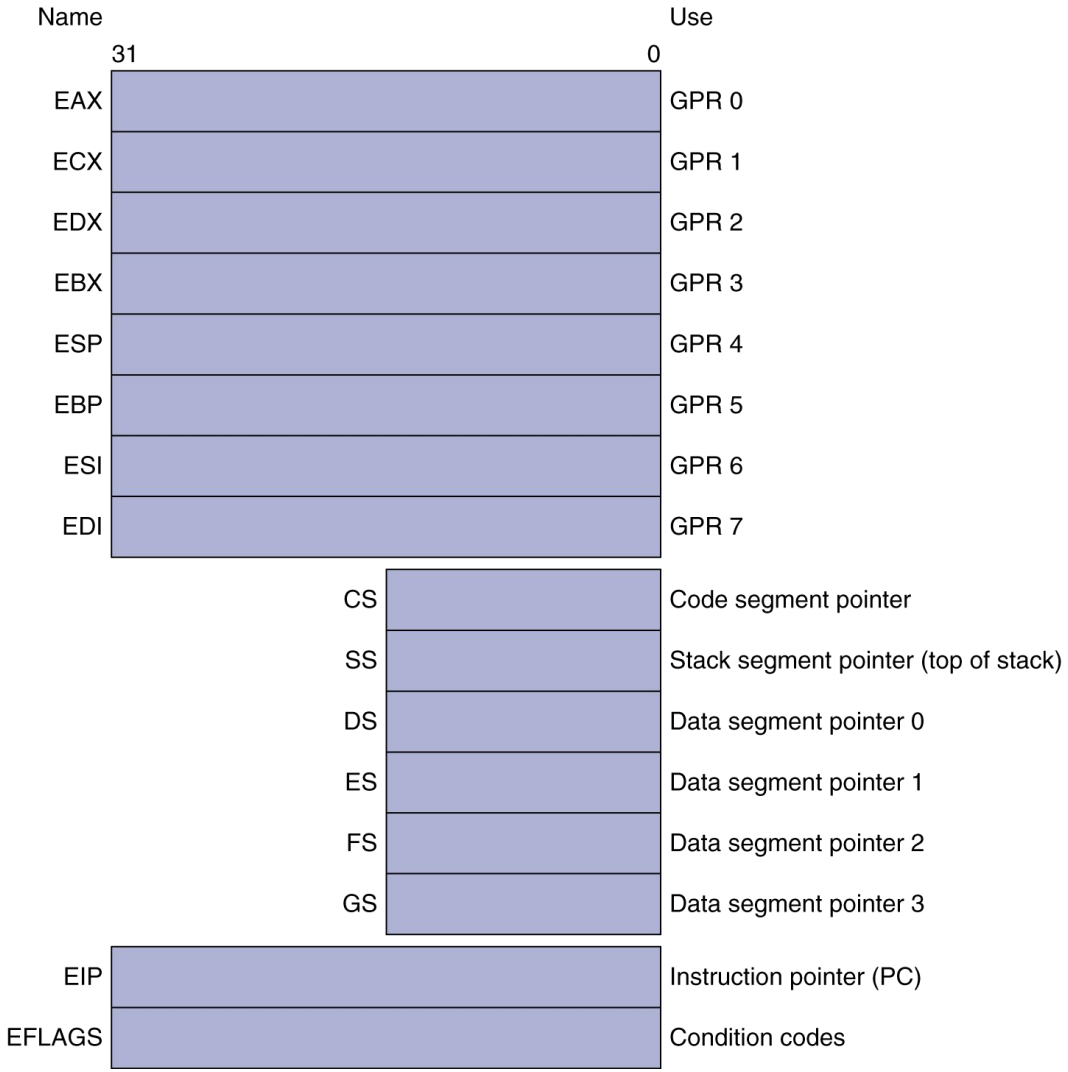
The Intel x86 ISA

- 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

a. JE EIP + displacement

4	4	8
JE	Condi-tion	Displacement

b. CALL

8	32
CALL	Offset

c. MOV EBX, [EDI + 45]

6	1	1	8	8
MOV	d	w	r/m Postbyte	Displacement

d. PUSH ESI

5	3
PUSH	Reg

e. ADD EAX, #6765

4	3	1	32
ADD	Reg	w	Immediate

f. TEST EDX, #42

7	1	8	32
TEST	w	Postbyte	Immediate

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

计算机科学与技术学院 College of Computer Science and Technology 中国·浙江·杭州

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

Summary

- **Two principles of stored-program computers**
  - *Use instructions as numbers*
  - *Use alterable memory for programs*

- **Four design principles**
  - *Simplicity favors regularity*
  - *Smaller is faster*
  - *Good design demands good compromises*
  - *Make the common case fast*

- **RISC –V Instruction Set**