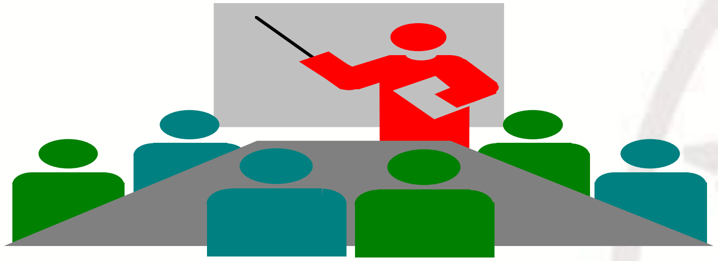


计算机组成与设计

Computer Organization & Design

The Hardware/Software Interface

Chapter 2

Instructions： Language of the Machine

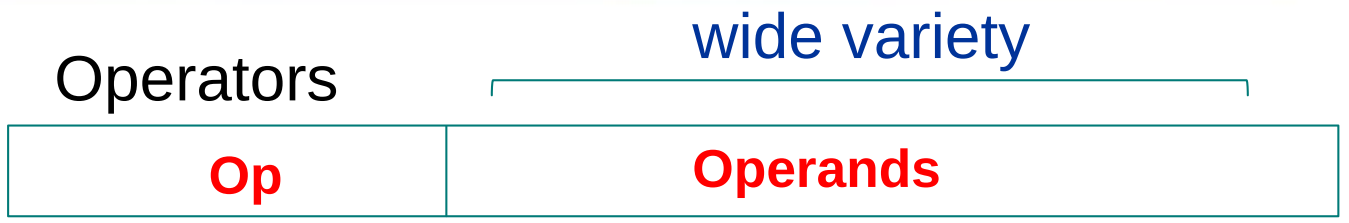
施青松
Asso. Prof. Shi Qingsong
College of Computer Science and Technology, Zhejiang University
zjsqs@zju.edu.cn

Contents of Chapter 2

Introduction

- **Language of the machine**
  - Instructions → Statement
  - Instruction set → Syntax
  - Assembler → Grammar
- **Design goals**
  - Maximize performance
  - Minimize cost
  - Reduce design time
- **Chosen instruction set: RISC-V**
  - Developed at UC Berkeley starting in 2010
  - MIPS as a relative case

Instruction characteristics



- 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

Type of internal storage in processer

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

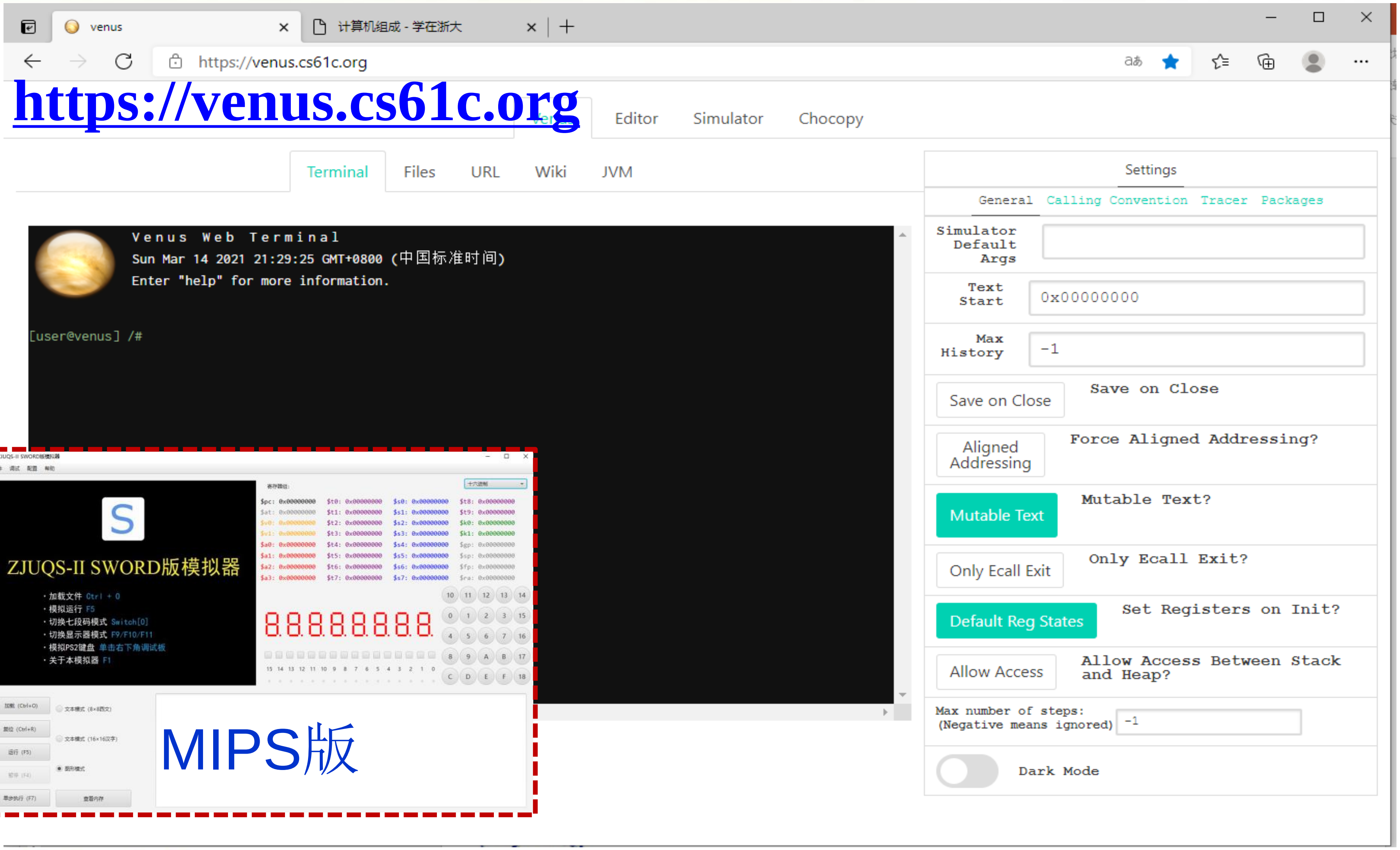
The number of the memory operand In the instruction

- **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

Variables difference

- **C**
  - Int   char   f
- **Instruction Set**
  - Regsister
  - Memory address
    - Displacement
    - Immediate
  - Stack

RISV-V机器指令和程序模拟系统

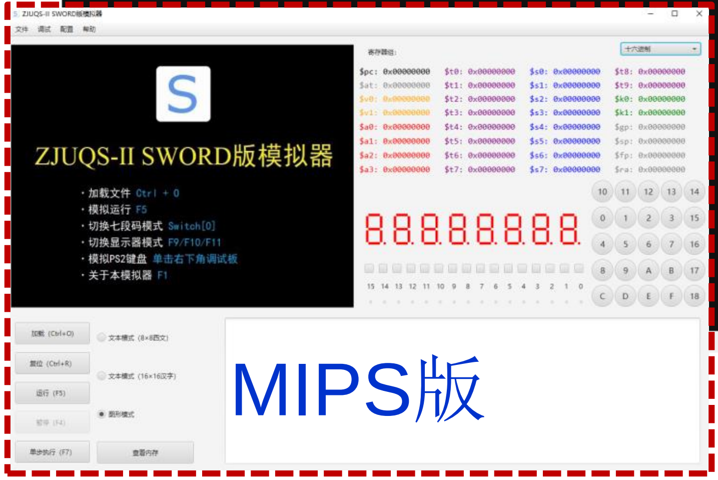


https://venus.cs61c.org

```
Venus Web Terminal
Sun Mar 14 2021 21:29:25 GMT+0800 (中国标准时间)
Enter "help" for more information.

[user@venus] /#
```

Simulator Default Args:
Text Start: 0x00000000
Max History: -1
Save on Close — Save on Close
Aligned Addressing — Force Aligned Addressing?
Mutable Text — Mutable Text?
Only Ecall Exit — Only Ecall Exit?
Default Reg States — Set Registers on Init?
Allow Access — Allow Access Between Stack and Heap?
Max number of steps: (Negative means ignored) -1
Dark Mode



```
$pc: 0x00000000  $t0: 0x00000000  $s0: 0x00000000  $t8: 0x00000000
$at: 0x00000000  $t1: 0x00000000  $s1: 0x00000000  $t9: 0x00000000
$v0: 0x00000000  $t2: 0x00000000  $s2: 0x00000000  $k0: 0x00000000
$v1: 0x00000000  $t3: 0x00000000  $s3: 0x00000000  $k1: 0x00000000
$a0: 0x00000000  $t4: 0x00000000  $s4: 0x00000000  $gp: 0x00000000
$a1: 0x00000000  $t5: 0x00000000  $s5: 0x00000000  $sp: 0x00000000
$a2: 0x00000000  $t6: 0x00000000  $s6: 0x00000000  $fp: 0x00000000
$a3: 0x00000000  $t7: 0x00000000  $s7: 0x00000000  $ra: 0x00000000
```

MIPS版

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

指令包含3个操作数

- Design Principle 1
  - Simplicity favors regularity （*简单源自规整*）
- Example 2.1

P65: **Compiling two simple C statements**

  - C code:

a = b + c;
d = a – e;

RISCV code:

```
add  a, b, c
sub  d, a, e
```

Arithmetic

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

```
f = ( g + h ) – ( i + j );
```

  - MIPS 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

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

2.3 Operands of the Computer Hardware

- **Three Category**
  - Register Operands、Memory Operands、Constant or Immediate Operand
- **Register Operands**
  - Arithmetic instructions operands must be registers
  - Difference between the variables of a programming
    - Use for frequently accessed data
      - 64-bit data is called a “double-word”
      - 32-bit data is called a “word”
  - Registers is limited number of
    - RISC-V has a 32 × 64-bit register file
      - general purpose registers x0 to x31(?)
- **Design Principle 2**
  - ***Smaller is faster***

(*越小越快*：寄存器个数一般不超过32个)

RISC-V register conventions

- $s0, $s1, … for registers corresponding to variables in C and Java
- $t0, $t1, … for temporary registers for compiler

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)	

Operate with Register Operands

- **Example 2.3**

  P67： Compiling a C statement using registers

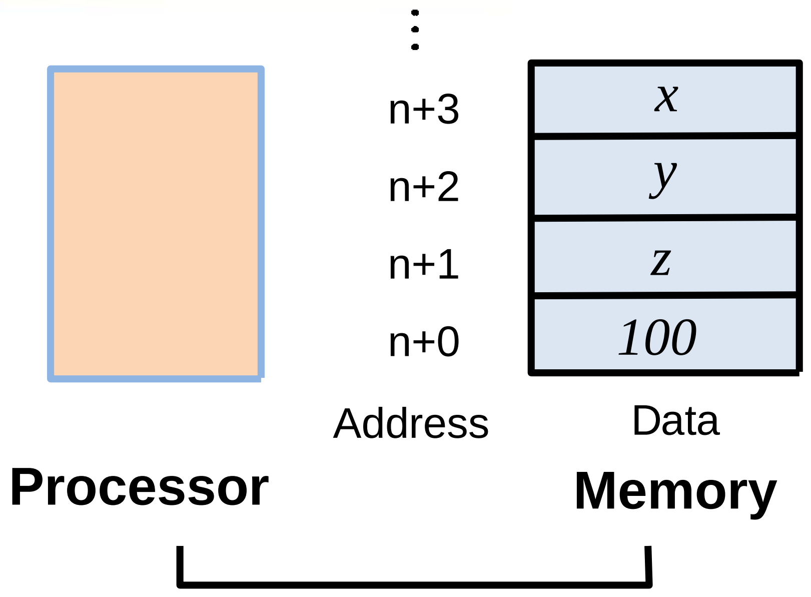
  - C code

    ```
    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 )
    ```

Memory operands



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

- **Example 2.4 P69:**

C code: **Compiling with an operand in memory**

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

(Assume: g ---- x20 h ---- x21 base address of A ---- **x22**)

MIPS 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
  - Base register: the register added to form the address →64(**x22**)
- **Byte/Half-word/word/Dword addressing**
  - →8(x22) /16(x22)/32(x22)/ 64(x22)
- **Alignment restriction**
  - RISC-V and x86 do not, but MIPS does

→32( )

- Endianness/byte order

Memory Alignment

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

正确

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

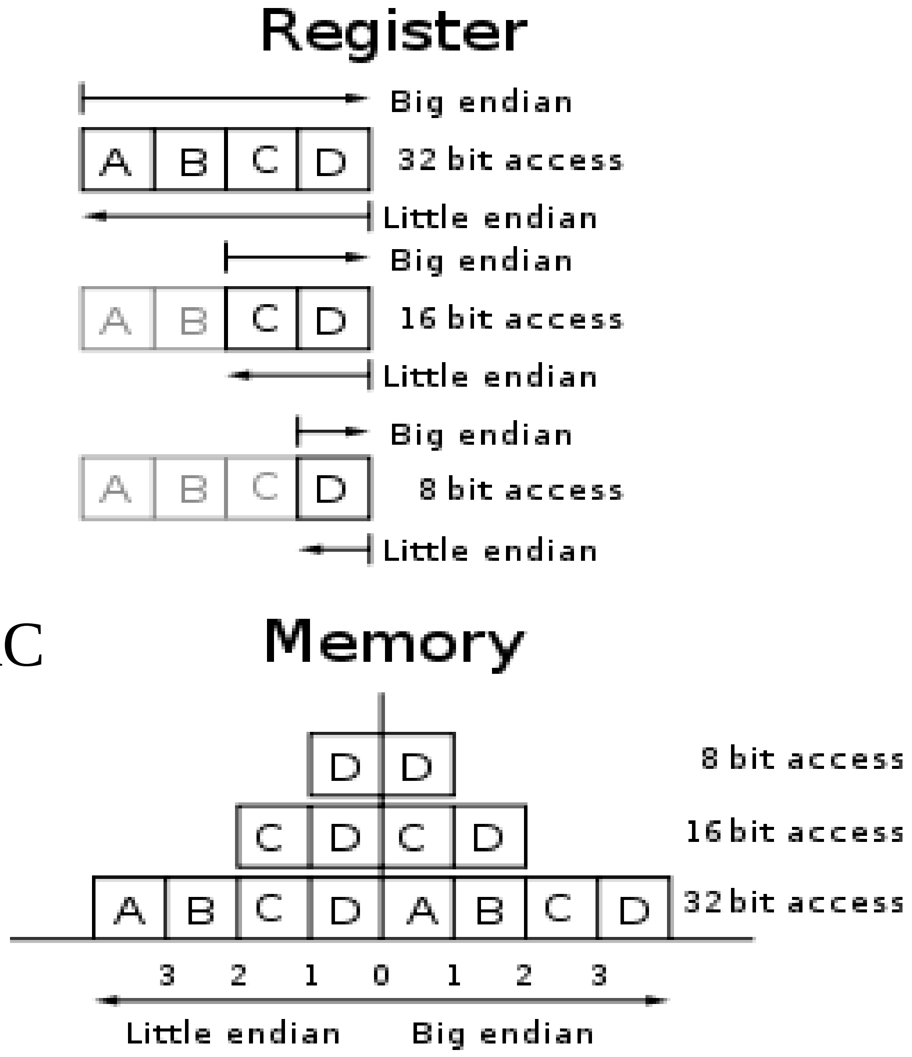
错误

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

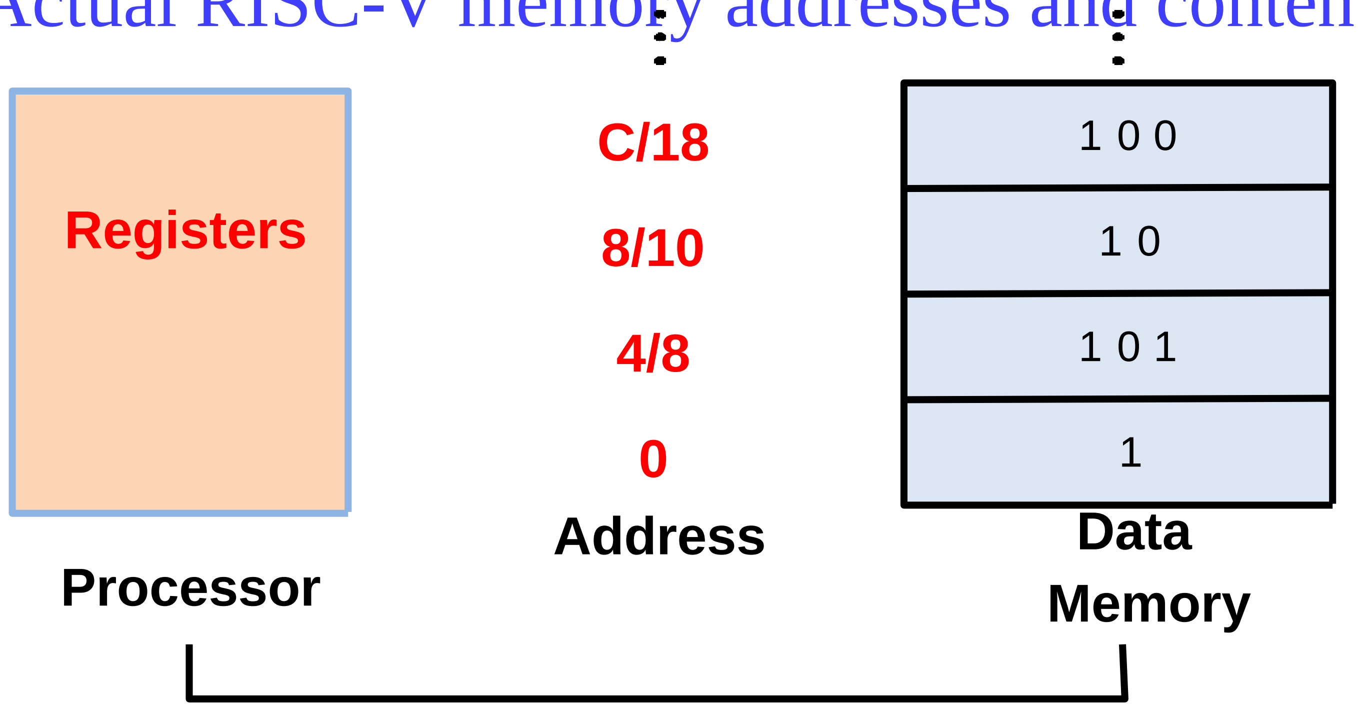
- **Big end: Leftmost**
  - PowerPC
  - 01 02 = 258
- **Little end: Rightmost**
  - RISC-V
  - 01 02 = 513
- **Bi-endian**
  - MIPS, ARM , Alpha, SPARC



Software/hardware interface

Compiler allocates data structures to memory
Compiler associating variables with registers

Actual RISC-V memory addresses and contents



The offset to be added to x22 in Example 2.4 must be 8×8

Example 2.5

- C code: P71：Compiling using load and store

  A[12] = h + A[8] ;   // A is an array of 100 words
  (Assume: h ---- x21   base address of A ---- x22)

- MIPS code:

```
ld     x9 , 64($s3)       #  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]
```

Registers vs. Memory

- **Registers are faster to access than memory**
- **Operating on memory data requires loads and stores**
  - More instructions to be executed
- **Compiler must use registers for variables as much as possible**
  - Only spill to memory for less frequently used variables
  - Register optimization is important!

Discussion： How to represent?

g = h + A[*i*]

(Assume: g, h, ***i*** -- s1, s2, ***s4*** base address of A -- s3)

A[i] that is 32bit Word

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 words

    (Assume: g, h, ***i*** -- s1, s2, ***s4*** base address of A -- s3)
  - MIPS code:

```
add   t1, s4, s4        #  temp reg t1 = 2 * i
add   t1, t1, t1        #  temp reg t1 = 4 * i
add   t1, t1, s3        #  t1 = address of A[i] (4 * i + s3)
lw    t0 , 0(t1)        #  temp reg t0 = A[i]
add   s1, s2, t0        #  g = h + A[i]
```

- **Spilling registers**

  Putting less commonly used variables(or those needed later) into memory

Discussion：How to represent?

Constant

g = h + 55

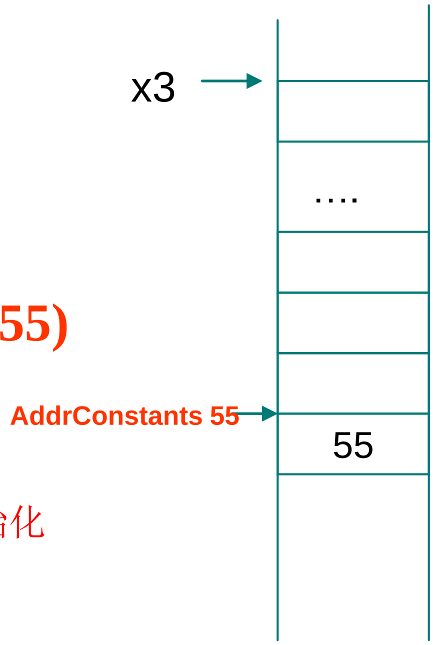
Many time a program
will use a constant in an operation

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 55 to register x22
  - Assuming $\text{AddrConstants}_{55}$ is address pointer of constant 55

Prestored in memory

```
ld    x9, AddrConstant55(x3) # x9 =constant 55
add  x22, x22, x9                         x22x22+x(x9==55)
```

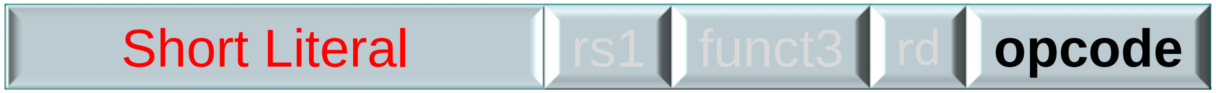


功能测试程序生成常数方式非常累赘，可以用此方法代替。前提是要初始化

Immediate Operands

- **Prestored with in Instruction**
  - Avoids the load instruction
  - **Immediate:** Other method for adding constant 55 to x22
    - Offer versions of the instruction

      addi x22, x22, **55** #$x22= x22 + 55
  - Short Literal



- **Design Principle 3**
  - ***Make the common case fast*: (why?)**

    Constant operands occur frequently

    it is very common

    Loading them from memory is very slow

*x*0 == zero

Brief summary

MIPS 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 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.

MIPS 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

2.4 Representing Instructions in the Computer --Instruction Format

- **All information in computer consists of binary bits**
  - Instructions are encoded in binary
    - Called machine code
- **Mapping registers into numbers(index)**
  - 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!

Register operands format Code

- Example 2.7 P81：

  Translating assembly into machine instruction

- MIPS code

  add x9, x20, x21

- Decimal version of machine code

7bits	5bits	5bits	3bits	5bits	7bits
0	21	20	0	9	**51**

- Binary version of machine code

Sometime use Hex!

7bits	5bits	5bits	3bits	5bits	7bits
0000000	10101	10100	000	01001	**0110011**

RISC-V fields (format)

Imm Region: ±2^{12}

Name	Fields						Comments
Field size	7bits	5bits	5bits	3bits	5bits	7bits	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

- **op:** *basic operation of the instruction, traditionally called the opcode.*
- **rd:** *destination register number.*
- **funct3:** *3-bit function code (additional opcode).*
- **rs1:** *the first register source operand.*
- **rs2:** *the second register source operand.*
- **funct7** *7-bit function code (additional opcode).*

Must bear in mind!

- Design Principle 3
  - *Good design demands good compromises*

- **All instructions in MIPS have the same length**
  - Conflict: same length ← → single instruction

RISC-V I-format Instructions

ld x9, 64(x22)

immediate[11:0]	rs1	funct3	rd	opcode
12bits	5bits	3bits	3bits	7bits

- **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
- **All instructions in RISC-V have the same length**
  - Conflict: same length ← → single instruction

RISC-V S-format Instructions

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

sd x9, 64(x22)

- 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 2.8 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]
```

- Decimal version RISC-V machine language code

	funct7	rs2	rs1	funct3	rd	opcode
ld	240/imm11-0		10	3	9	3
add	0	9	21	0	9	31
addi	1		9	0	9	19
sd	7/imm11-5	9	10	3	16/imm4-0	35
	7bits	5bits	5bits	3bits	3bits	7bits

- Binary version

	funct7	rs2	rs1	funct3	rd	opcode
ld	000_1111_0000		01010	011	01001	0000011
add	0000000	01001	10101	000	01001	0011111
addi	0000_0000_0001		01001	000	01001	0010011
sd	0000111	01001	01010	011	10000	0100011
	7bits	5bits	5bits	3bits	3bits	7bits

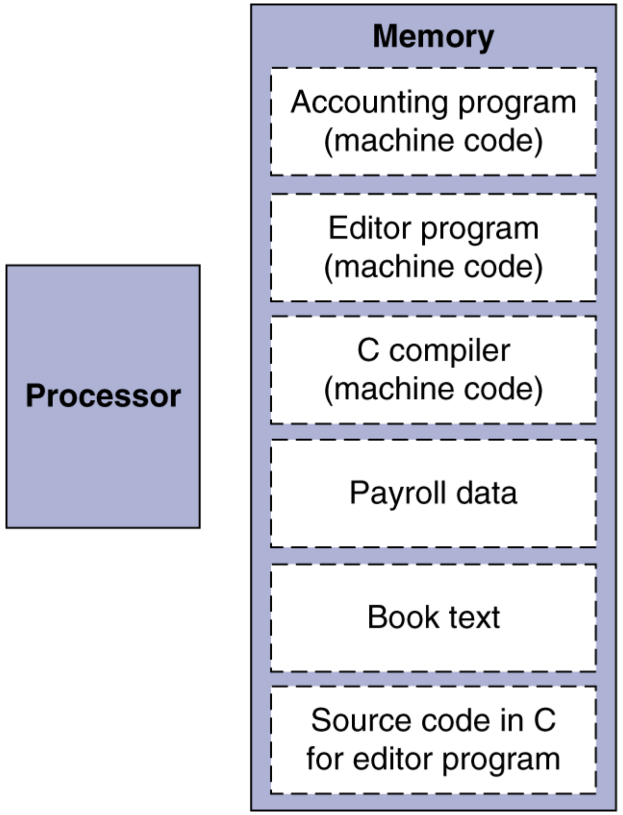
Note the only difference of the first and last instructions!

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

Store-program

Stored-program concept

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

Summary encoding

- 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	63	1	2	3	8	35	sd x1, 1000(x2)

MIPS VS RISC-V

MIPS

Field size	6bits	5bits	5bits	5bits	5bits	6bits	All MIPS instruction 32 bits
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
i-format	op	rs	rt	Imm/Word address			Data transfer ,branch format
J-format	op	target address (word)					Uncondition-al jump

RISC-V

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

MIPS operands, assembly and machine language

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 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
Arithmetic	**add**	add $s1,$s2,$s3	**$s1=$s2 + $s3**	**Three register operands**
	subtract	sub $s1,$s2,$s3	**$s1=$s2 - $s3**	**Three register operands**
	Add immediate	addi $s1,$s2,100	$s1=$s2+100	Used to add constants
Data transfer	load word	lw $1, 100($s2)	$s1=Memory[$s2+100]	Data from memory to register
	store word	sw $s1, 100($s2)	Memory[$s2+100]=$s1	Data from register to memory

Name	Format	Example						Comment
add	R	0	18	19	17	0	32	add $s1, $s2, $s3
sub	R	0	18	19	17	0	34	sub $s1, $s2, $s3
addi	I	8	18	17	100			addi $s1,$s2,100
lw	I	35	18	17	100			lw $s1, 100($s2)
sw	I	43	18	17	100			sw $s1, 100($s2)
Field size		6bits	5bits	5bits	5bits	5bits	6bits	All MIPS instruction 32 bits
R-format	**R**	**op**	**rs**	**rt**	**rd**	**shamt**	**funct**	**Arithmetic instruction format**
I-format	**I**	**op**	**rs**	**rt**	**address**			**Data transfer ,branch format**

请同学课后改为**RISC-V**体系构架加深指令构架理解
同时了解**MIPS**指令构架

2.5 Logical Operation Self-learning

- Operating some bits within word or individual bit

Logic operations	C operators	Java operators	RISCV instructions
Logic 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***

没有nor、not指令，是否违反布尔代数原则？

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 $\longleftarrow$

0000 0000 0000 0000 0000 0000 1001 0000 $(9\times 16=144)_{10}$

```
sll x11, x19, 4          // reg x11=reg x19 << 4 bit
```

funct6	immediate	rs1	funct3	rd	opcode
0	4	19	1	11	19

AND operator

- It is bit-by-bit (bitwise-AND)
  - Result=1 : both bits of the operands are 1

x2:

0000 0000 0000 0000 0000 1101 0000 0000

x1:

0000 0000 0000 0000 0011 1100 0000 0000

and x0, x1, x2 # reg x0 =reg x1 & reg x2

Result:

0000 0000 0000 0000 0000 *1100* 0000 0000

OR operator

- It is bit-by-bit(bitwise-OR)
  - Result=1 : ***either*** bits of the operands is 1

x2:
0000 0000 0000 0000 0000 1101 0000 0000
x1:
0000 0000 0000 0000 0011 1100 0000 0000

```
or x0, x1, x2     # reg x0 =reg x1 | reg x2
```

Result:
0000 0000 0000 0000 0011 *11*01 0000 0000

- ***NOR* operator**
  - NOT(A OR B)
    - A NOR 0 =NOT(A OR 0)=***NOT***(A)

$t1:

0000 0000 0000 0000 0011 1100 0000 0000

$t3:

0000 0000 0000 0000 0000 0000 0000 0000

RISC-V 没有not指令

```
nor $t0, $t1, $t3          #reg $t0=~(reg $t1 | reg $t3)
```

Result:

1111 1111 1111 1111 11*00 00*11 1111 1111

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 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.

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

2.6 Instructions for making decisions

- **Branch instructions**
  - beq register1, register2, L1
  - bne register1, register2, L1
- **Example 2.9** **Compiling an *if* statement to a branch**

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

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

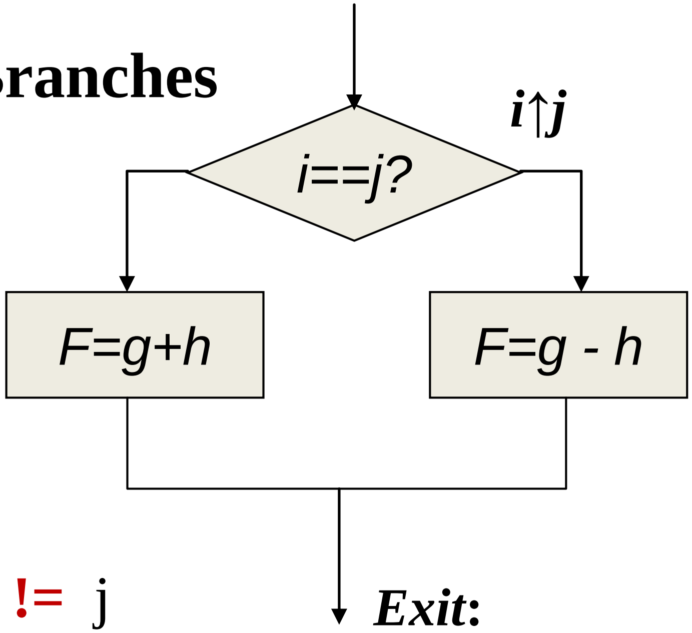
  - RISC-V assembly code:

```
          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 )
```

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
```

Supports 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  $t0, 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
```

Supports 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:
```

Most popular Compare Operation

-- set on less than : *slt*

- **set on less than -slt**
  - If the first reg. is less than second reg. then sets third reg to 1

    slt x5, x19, x20 # x5=1 if x19 < x20
- **Example 2.13 Compiling a less than test**

  (Assume: a -- s0 b -- s1)
  - C language:

    if (a < b), goto Less
  - MIPS assembly code:

```
    slt    x5, x8, x9       #x5 = 1  if  x8  < x9   ( a < b)
    bne    x5, zero, Less  # go to Less  if  x5  !=  0 (that is,  if  a  <  b)
    ……
Less:
```

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)

C code:

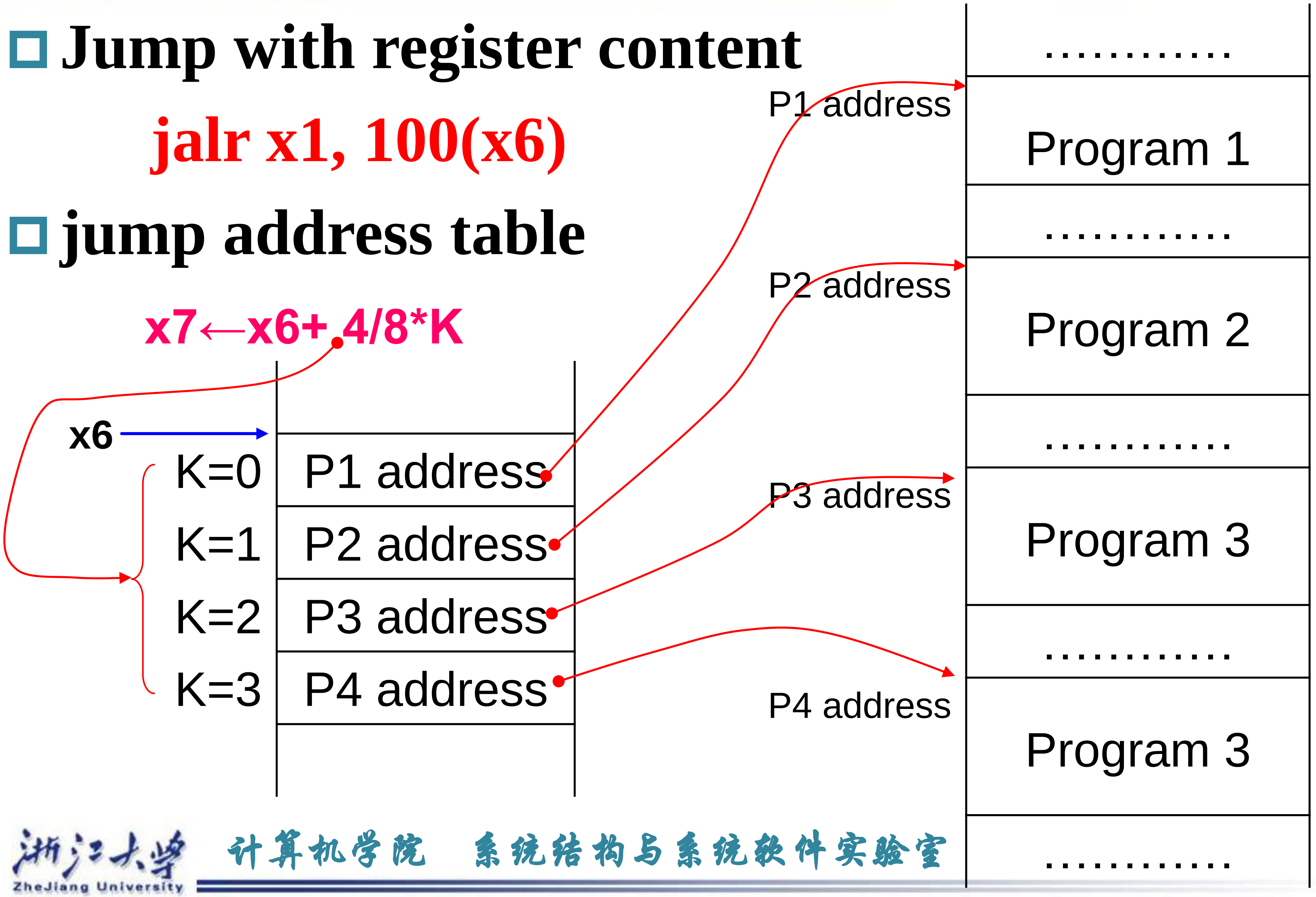
```
switch ( k )  {
        case  0 :   f  =  i  +  j ;  break ;    /*  k  =  0  */
        case  1 :   f  =  g +  h ;  break ;   /*  k  =  1  */
        case  2 :   f  =  g  -  h ;  break ;   /*  k  =  2  */
        case  3 :   f  =  i  -  j ;  break ;    /*  k  =  3  */
}
```

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:

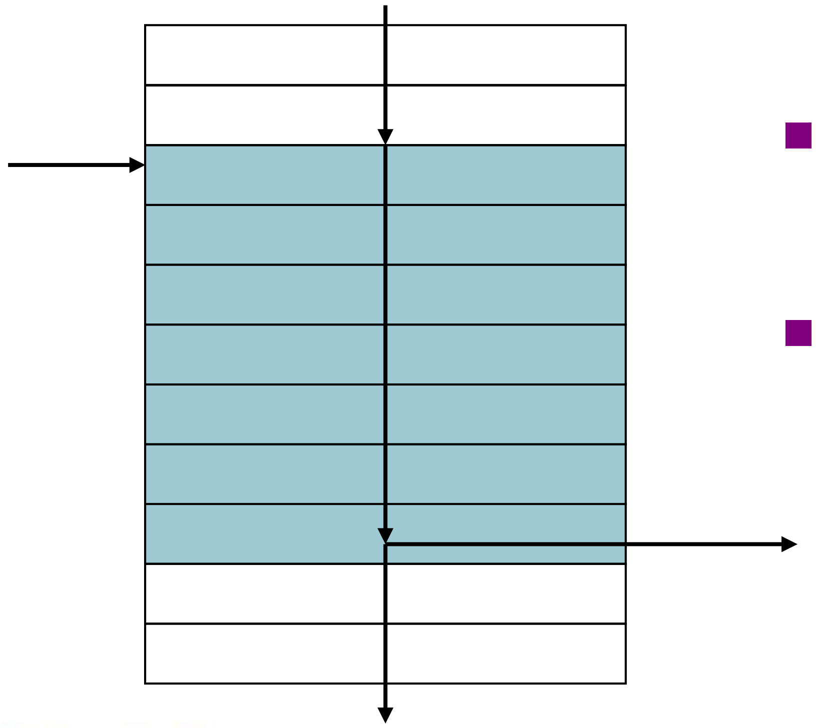
Memory1
L0:address
L1:address
L2: address
L3:address

```
L0:  add   $s0, $s3, $s4      # k  =  0  so  f  gets  i  +  j
     j     Exit               # end of this case so go to Exit
L1:  add   $s0, $s1, $s2      # k  =  1  so  f  gets  g  +  h
     j     Exit               # end of this case so go to Exit
L2:  sub   $s0, $s1, $s2      # k  =  2  so  f  gets  g  -  h
     j     Exit               # end of this case so go to Exit
L3:  sub   $s0, $s3, $s4      # k  =  3  so  f  gets  i  -  j
           Exit:              # 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**/***branch lables*** (except at beginning)



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

2.7 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 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

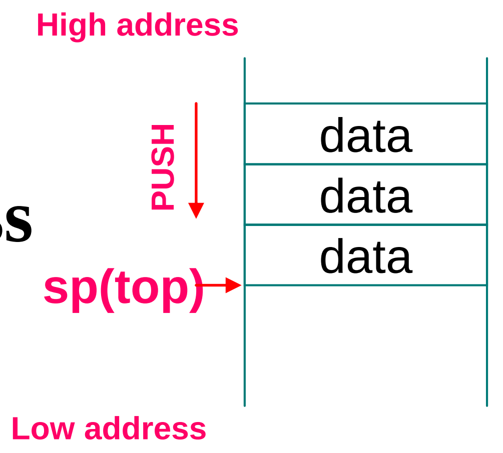
Using More Registers

- **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 - 4

```
addi sp,sp,-8
sd …,8(sp)
```

  - Pop: sp = sp + 4

```
ld     …,8(sp)
addi sp,sp,8
```

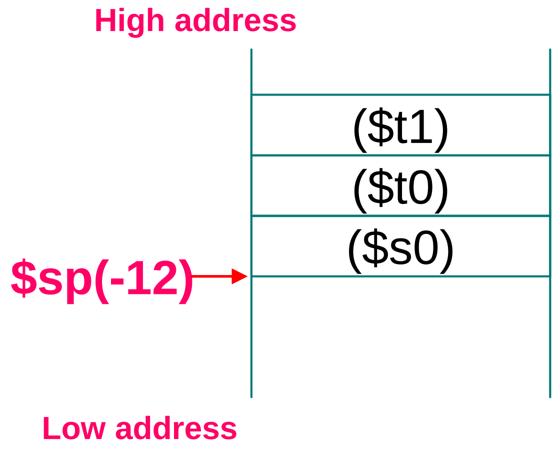


Example 2.15 Compiling a leaf procedure

(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.
```

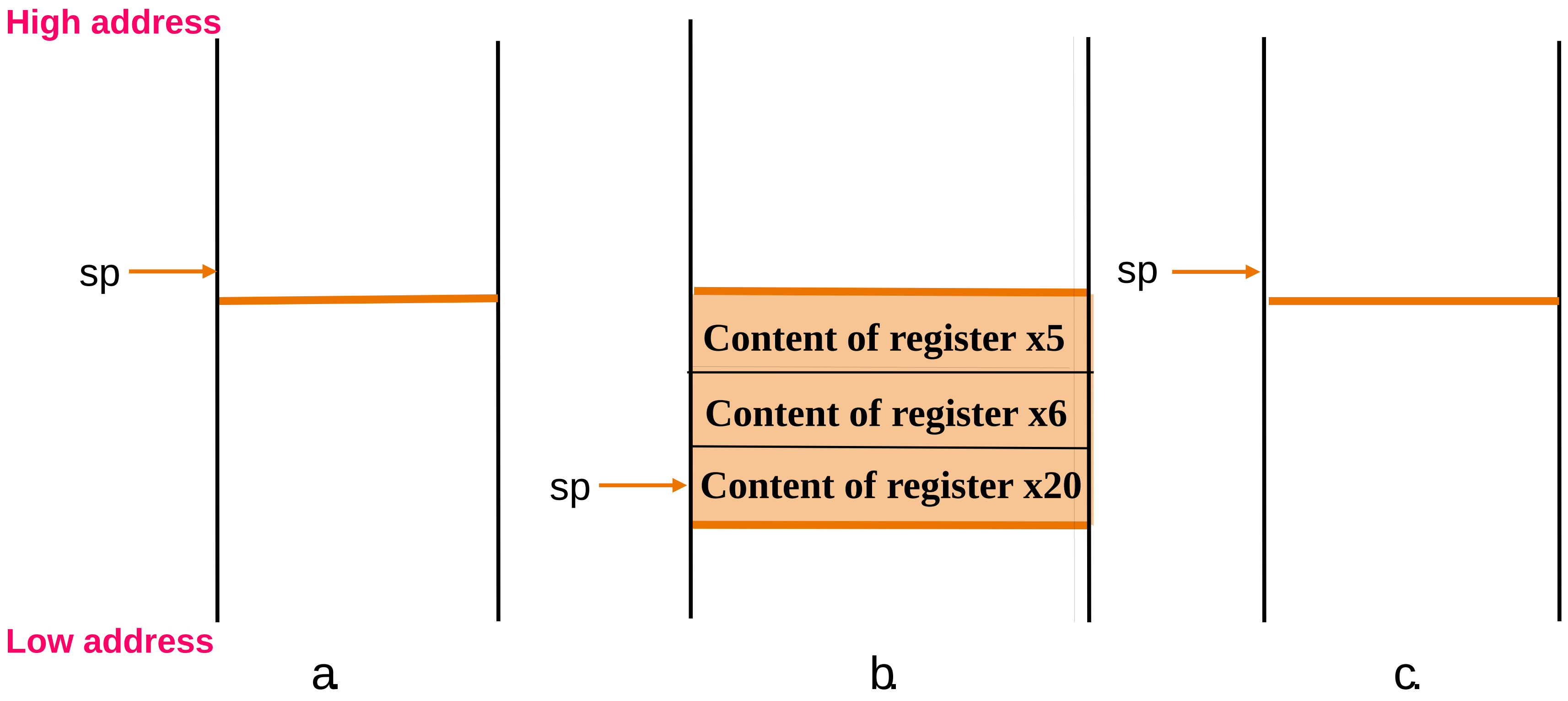
```
add  x5,x10,x11          # register x5contains   g  +  h
add  x6,x12,x1           # register x6 contains   i  +  j
sub  x20,x5,x6           # f  =  x5-  x6, which is ( g  +  h ) – ( i  +  j )

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

```
      ld    x20, 0(sp)        # restore register x20 for caller
      ld    x6,  8(sp)        # restore register x6 for caller
POP   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

  - MIPS 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-Continue

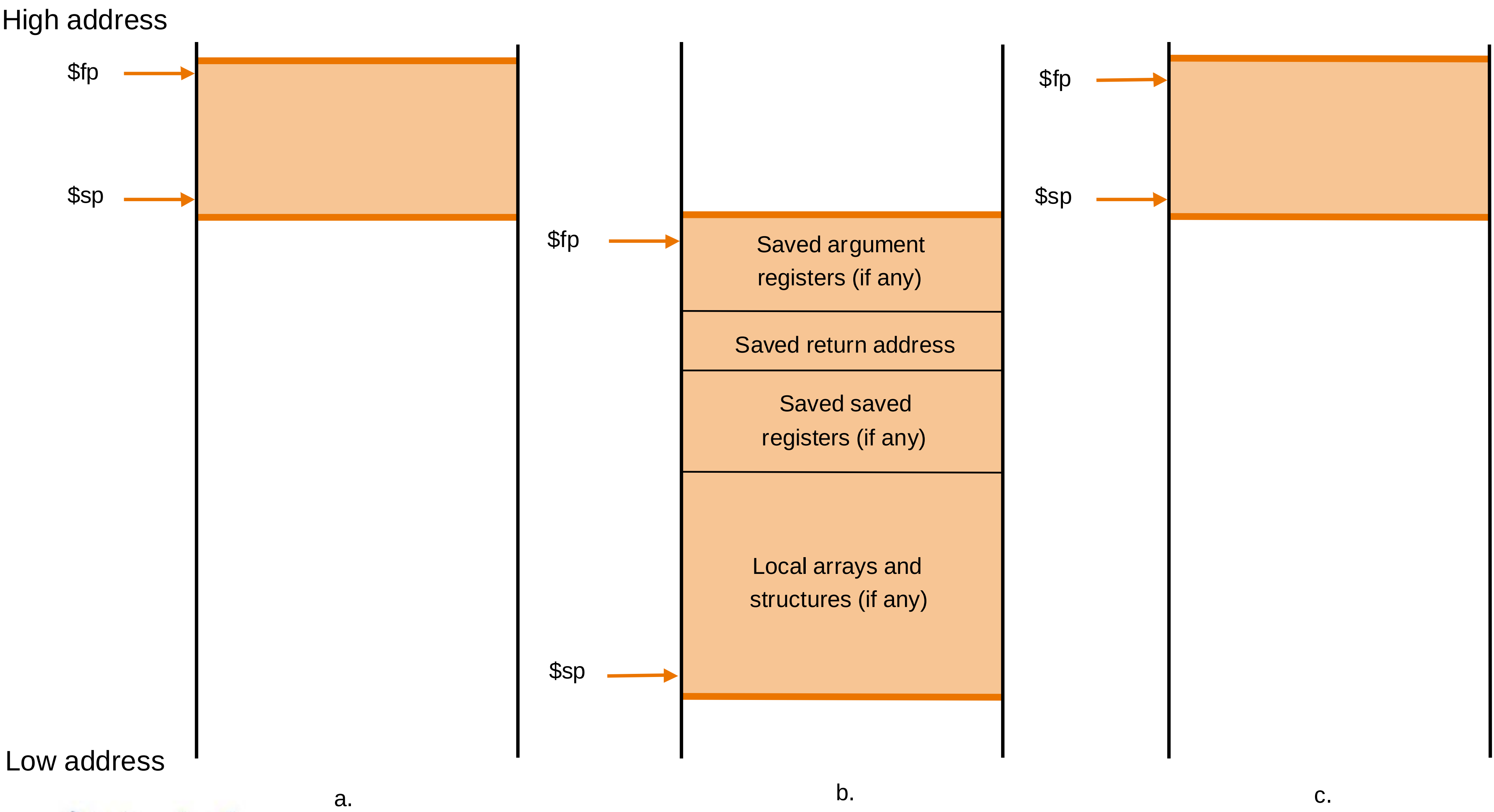
```
L1:  addi   a0,  a0, -1          # n  >=  1: argument gets ( n  -  1 )
     jal    ra, fact             # call fact with ( n  -  1)
     add    t1, a0, zero         #move result of fact(n - 1) to x6(t1)
     ld     a0, 0($sp)           # return from jal: restore argument n
     ld     ra, 8($sp)           # 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**

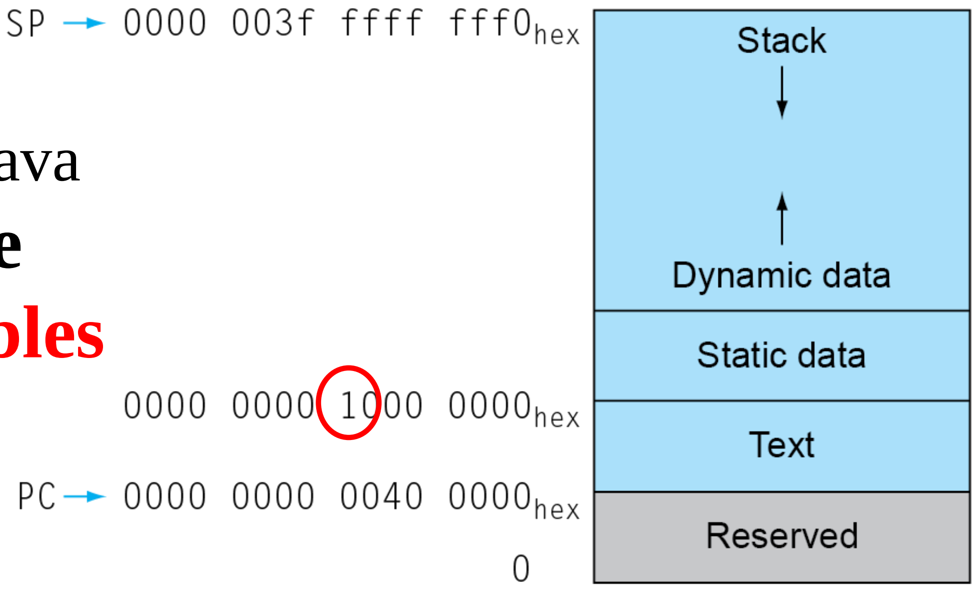
  Temporary registers(t0 ~ t7), argument registers(a0 ~ a7),
  return value registers(a0 ~ a7), stack below the stack pointer

Stack allocation before, during and after procedure call


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

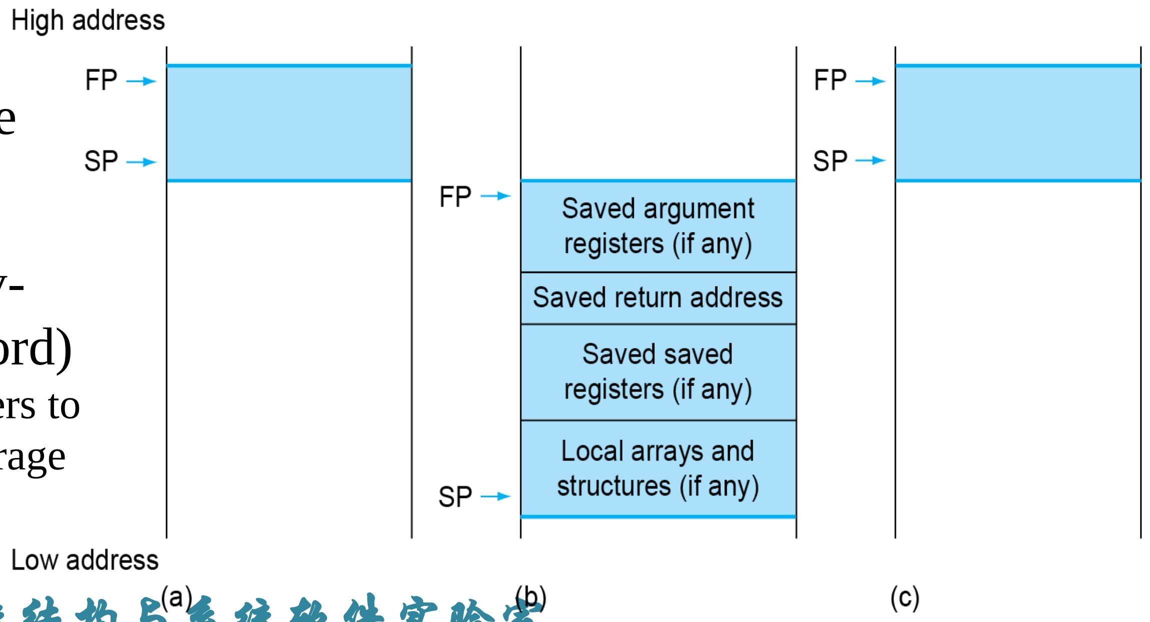
Memory Layout

- **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 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

MIPS machine language

Name	Format	Example						Comment
add	R	0	18	19	17	0	32	add $s1, $s2, $s3
sub	R	0	18	19	17	0	34	sub $s1, $s2, $s3
lw	I	35	18	17	100			lw $s1, 100($s2)
sw	I	43	18	17	100			sw $s1, 100($s2)
and	R	0	18	19	17	0	36	and $s1, $s2, $s3
or	R	0	18	19	17	0	37	or $s1, $s2, $s3
nor	R	0	18	19	17	0	39	nor $s1, $s2, $s3
addi	I	12	18	17	100			addi $s1, $s2,100
ori	I	13	18	17	100			ori $s1, $s2,100
sll	R	0	0	18	17	10	0	sll $s1, $s2,10
srl	R	0	0	18	17	10	2	srl $s1, $s2,10
beq	I	4	17	18	25			beq $s1, $s2,100
bne	I	5	17	18	25			bne $s1, $s2,100
slt	R	0	18	19	17	0	42	slt $s1, $s2,$s3
j	J	2	2500					j 10000(see section 2.9)
jr	R	0	31	0	0	0	8	j Sra
jal	J	3	2500					jar 10000(see section 2.9)
Field size		6bits	5bits	5bits	5bits	5bits	6bits	All MIPS instruction 32 bits
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
i-format	I	op	rs	rt	address			Data transfer ,branch format

请同学课后改为**RISC-V**体系构架加深指令构架理解

同时了解**MIPS**指令构架

2.8 Communicating with People Beyond Numbers

- **ASCII** **(American Standard Code for Information Interchange)**
- **Instructions for moving bytes in MIPS**
  - Load byte (lb): lb $t0, 0($sp) # read byte from source
  - Store byte (sb): sb $t0, 0($sp) # write byte to destination
- **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
- **C uses the third choice**
  - Terminate a string with a byte whose value is 0 (null in ASCII)

Example 2.17 Compiling a string copy procedure

(Assume: base addresses for x and y -- $a0 and $a1   i -- $s0)

- C code： X→Y

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

- MIPS assembly code:

```
strcpy:   sub   $sp, $sp, 4             # adjust stack for 1 more item
          sw    $s0, 0($sp)             # save $s0
          add   $s0, $zero, $zero       # i = 0 + 0
L1:       add   $t1, $a1, $s0           # address of y[ i ] in $t1
          lb    $t2, 0($t1)             # $t2 = y [ i ]
          add   $t3, $a0, $s0           # address of x[ i ] in $t3
          sb    $t2, 0($t3)             # x[ i ] = y[ i ]
```

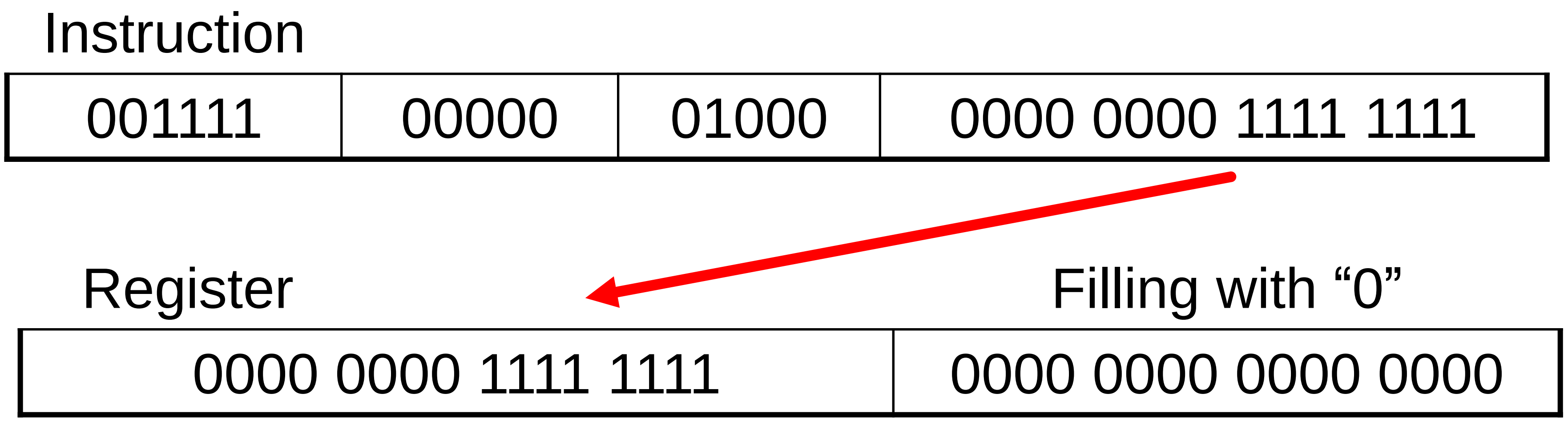
```
add    $s0, $s0, 1          # i = i + 1
bne    $t2, $zero, L1       # if y[ i ] != 0, go to L1
lw     $s0, 0($sp)          # y[ i ] = = 0: end of string;
                            # restore old $s0
add    $sp, $sp, 4          # pop 1 word off stack
jr     $ra                  # return
```

- **Optimization for example 2.17**
  - strcpy is a leaf procedure
  - Allocate i to a temporary register $t0
- **For a leaf procedure**
  - The compiler exhausts all temporary registers
  - Then use the registers it must save

2.9 MIPS Addressing for 32-Bit Immediate and Addresses

32-Bit Immediate addressing

- most constants is short and fit into 16-bit field
- Set upper 16 bits of a constants in a register with *load upper immediate* (lui)
- lui $t0 , 255



- **Example 2.19 Loading a 32-bit constant**
  - The 32-bit constant:

    0000 0000 0011 1101 0000 1001 0000 0000 $(61*16^4 + 2304=4000000)_{10}$
  - MIPS code:

```
        lui    $s0, 61              #  61 decimal  =  0000 0000 0011 1101 binary
(The value of $s0 afterward is: 0000 0000 0011 1101   0000 0000 0000 0000)

        addi  $s0, $s0, 2304  #  2304 decimal  = 0000 1001 0000 0000 binary
(The value of $s0 afterward is: 0000 0000 0011 1101   0000 1001 0000 0000)
```

- **Note： Why does it need two steps?**
- **The reserved register $at for the assembler**

Addressing in branches and jumps

- For jumps: J-format
  - Example:

```
j      10000              #  go to location  10000
|     2     |                  10000                          |
    6 bits                     26 bits
```

  - Pseudo-direct addressing

26 bits of the instruction concatenated with the upper 4 bits of PC

- For branches:
  - Example:

```
 bne   $s0, $s1, Exit  #  go to Exit  if  $s0  !=  $s1
|     5     |    16    |    17    |          Exit           |
    6 bits     5 bits     5 bits           16 bits
```

  - PC-relative addressing

PC = (PC + 4) + Branch address

Example 2.20 Show branch offset in machine language

- C language:

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

MIPS assembler code in Example 2.12:

```
Loop:   add   $t1, $s3, $s3      # temp reg $t1 = 2 * i
        add   $t1, $t1, $t1      # temp reg $t1 = 4 * i
        add   $t1, $t1, $s6      # $t1 = address of save[i]
        lw    $t0, 0($t1)        # temp reg $t0 = save[i]
        bne   $t0, $s5, Exit     # go to Exit if save[i] != k
---------------------------
        add   $s3, $s3, $s4      # i = i + j
        j     Loop               # go to Loop
Exit:
```

- Assembled instructions and their addresses:

add	80000	0	19	19	9	0	32	Loop:
add	80004	0	9	9	9	0	32	
add	80008	0	9	22	9	0	32	
Lw	80012	35	9	8	0			
bne	*80016*	5	8	21	2 (8)			
add	80020	0	19	20	19	0	32	
j	80024	2	20000 (80000)					
	80028	. . .						Exit:

80028 – 80020 =8

PC+4+offset=80028

- Modification:
  - All MIPS instructions are 4 bytes long
  - PC-relative addressing refers to the number of words
  - The address field at 80016 above should be 2 instead of 8

- **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 Branching far away**
  - Given a branch:

```
        beq   $s0, $s1, L1
```

  - Rewrite it to offer a much greater branching distance:

```
        bne    $s0, $s1, L2
        j        L1
L2:
```

Summary of MIPS architecture in Ch. 2

RISC-V Instruction Format

Namec	Fields						Co
Field size	7bits	5bits	5bits	3bits	5bits	7bits	All MIPS i
R-type	funct7	rs2	rs1	funct3	rd	**opcode**	Arithmetic
I-type	immediate[11:0]		rs1	funct3	rd	**opcode**	Loads & im
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	**opcode**	
SB-type	*imm[12,10:5]*	rs2	rs1	funct3	*imm[4:1,11]*	**opcode**	Condition
UJ-type	*immediate[20,10:1,11,19:12]*				rd	**opcode**	Unconditi
U-type	immediate[31:12]				rd	**opcode**	Upper im

RISC-V Operands & Conventions

Name	Example
32 registers	$zero, ra, sp, gp, tp, t0-t6, s0~s11, a0-
Mem words	Memory[0], Memory[8], Memory[10], . . . , Memory[18,446,744,073,70

Name	Register no.	Usage	Pr
*x*0(**zero**)	**0**	**The constant value 0**	
*x*1(**ra**)	1	Return address(link register)	
*x*2(**sp**)	2	Stack pointer	
*x*3(gp)	3	Global pointer	
*x*4(tp)	4	Thread pointer	
*x*5-*x*7(t0-t2)	5-7	Temporaries	

MIPS assembly language

- Arithmetic

add	add $s1, $s2, $s3
subtract	sub $s1, $s2, $s3
add immediate	addi $s1, $s2, -3 (Note:subi does not exist)

- Data transfer

load word	lw	$s1, 100($s2)
store word	sw	$s1, 100($s2)
load byte	lb	$s1, 100($s2)
store byte	sb	$s1, 100($s2)
load upper immediate	lui	$s1, 100

- Conditional branch

Why not *beqi*?

branch on equal	beq	$s1, $s2, 25
branch on not equal	bne	$s1, $s2, 25
set on less than	slt	$s1, $s2, $s3
set on less than immediate	slti	$s1, $s2, 100

- Unconditional jump

jump	j	2500
jump register	jr	$ra
jump and link	jal	2500

MIPS addressing mode summary

- Register addressing:

  add $s0,$s0,$s0

- Base or displacement addressing:

  lw $s1,0($s0)

- Immediate addressing:

  addi $s0,$s0,4

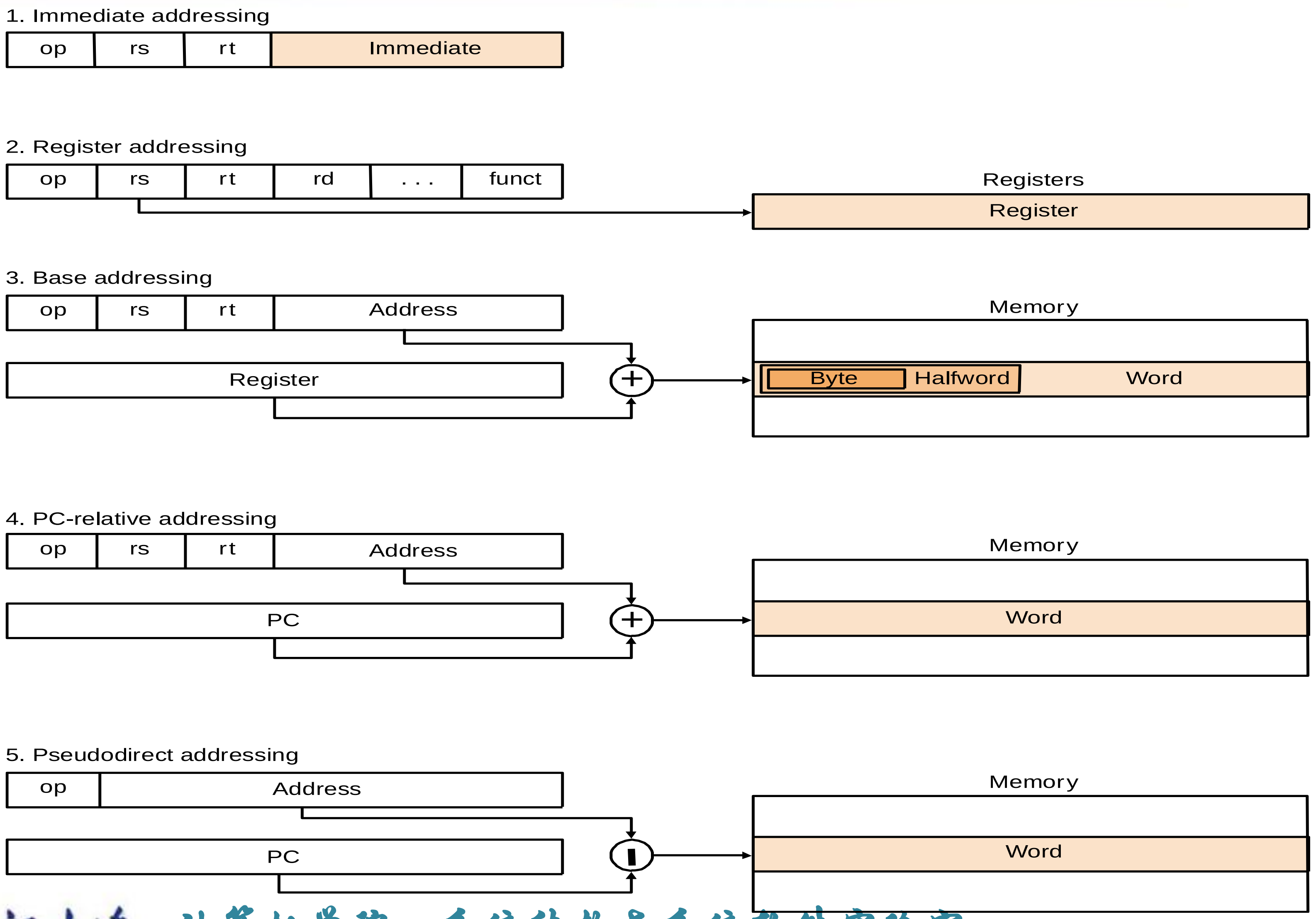
- PC-relative addressing:

  beq $s0,$s1,L1

- Pseudodirect addressing:

  j Address1

Five MIPS addressing modes



Example 2.22 Decoding machine code

- **Machine instruction**

(Bits: 31 28 26 5 2 0)

0000 0000 1010 1111 1000 0000 0010 0000

- **Decoding**
  - Determine the operation from opcode

    op: 000000 → **R-format instruction**

op	rs	rt	rd	shamt	funct
000000	00101	01111	10000	00000	100000

funct: 100000 → **add instruction**

p135-P136

  - Determine other fields

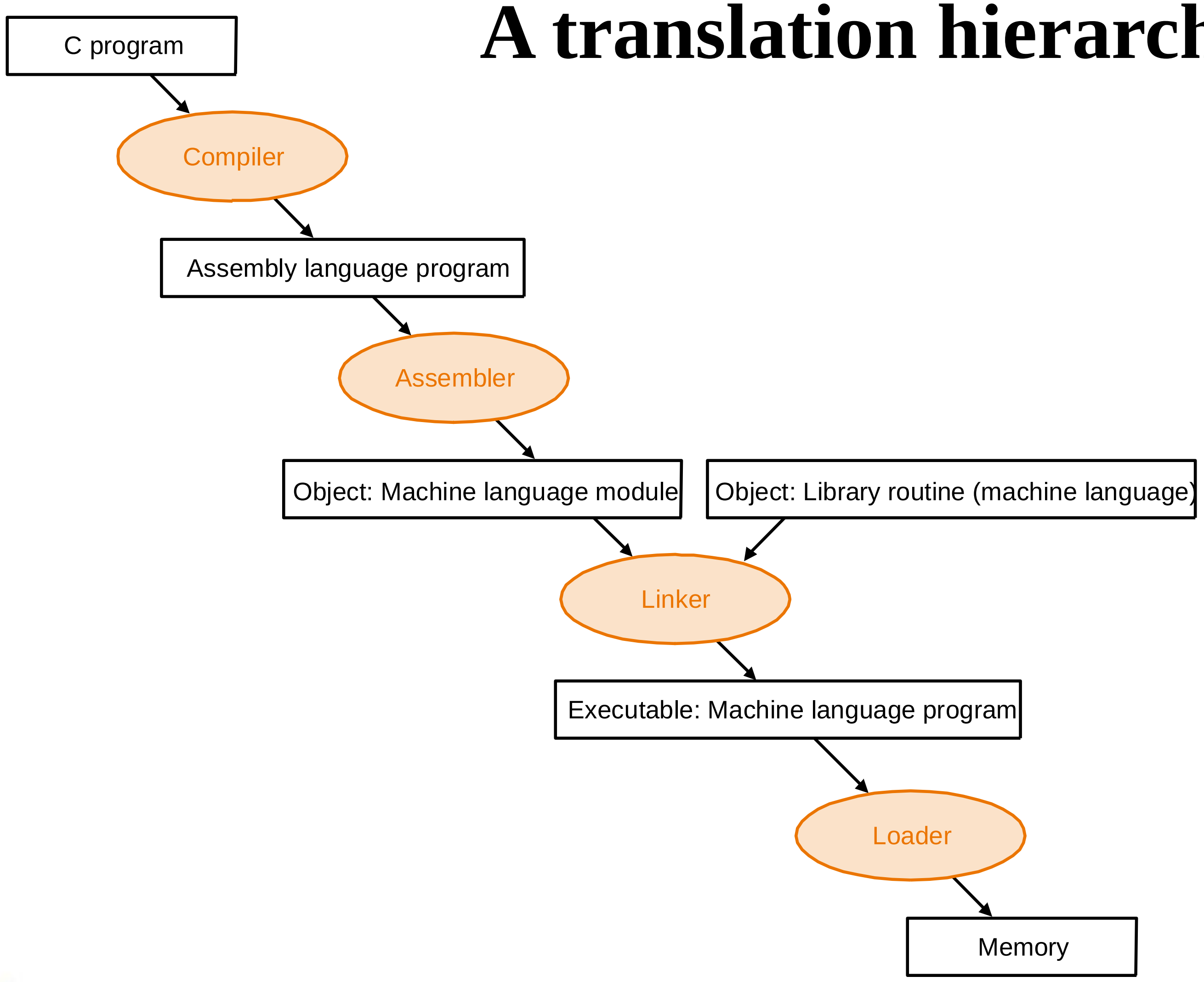
    rs: $a1; rt: $t7; rd: $s0

  - Show the assembly instruction

    add $s0, $a1, $t7 (Note: add rd,rs,rt)

2.10 Translanting and Starting a Program

A translation hierarchy


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

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

    move \$t0,\$t1 # register \$t0 gets register \$t1

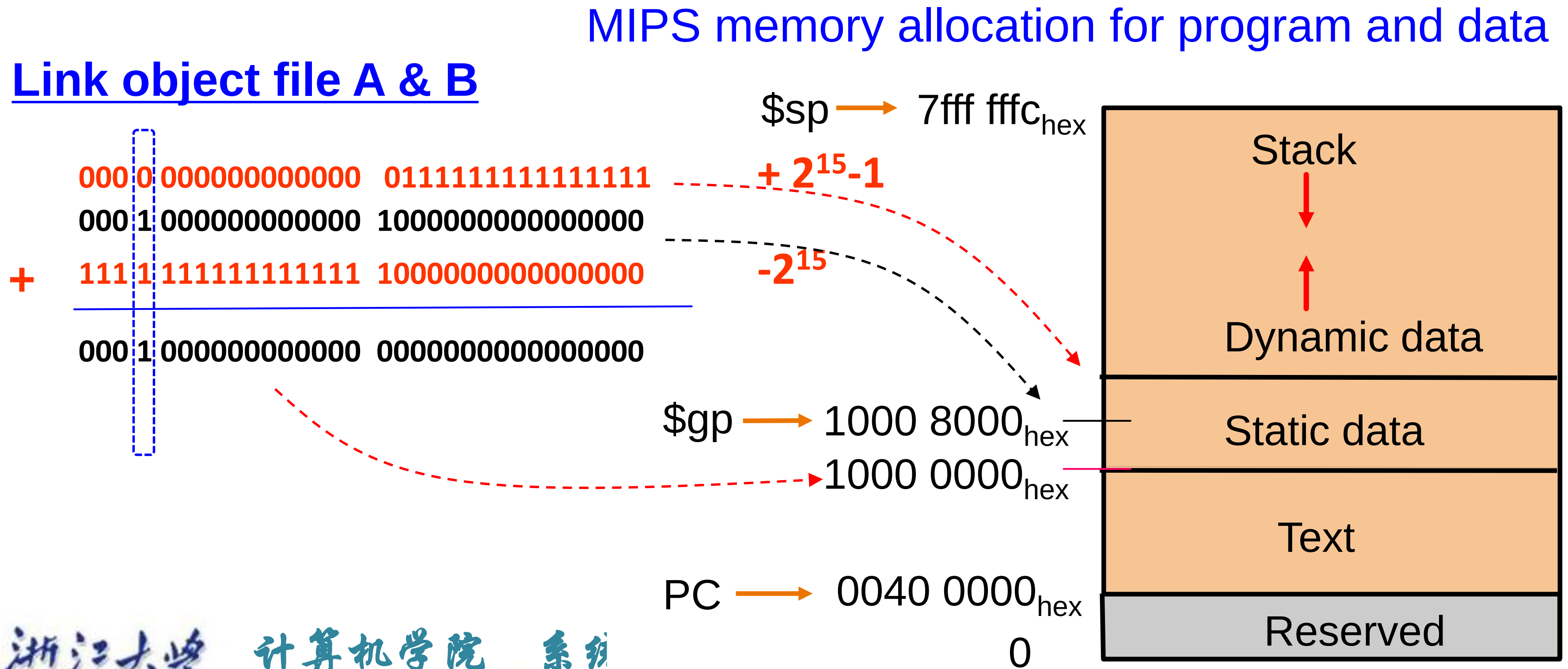
    add \$t0,\$zero, \$t1 # register \$t0 gets 0+register \$t1
  - Symbol table
    - A table that matches name of **lables** to the addresses of the memory words that instructions occupy.
- **Object file of UNIX (six distinct pieces)**
  - object file header--size and position of the other pieces
  - Text 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	lw $a0, 0($gp)	
	4	jal 0	
	……	--	
Data segment	0	(X)	
	……	……	
Relocation information	Address	Instruction type	Dependency
	0	lw	X
	4	jal	B
Symbol table	label	Address	
	X	--	
	B	--	

Linking

- 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**)



Loading

- Determine size of text and data segments
- Create an address space large enough
- Copy instructions and data from executable file to memory
- Copy parameters (if any) to the main program onto the stack
- Initialize registers and set $sp to the first free location
- Jump to a start-up routine

[Link object file A & B]

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

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

- Register allocation for *swap*

  v ---- $a0 k ---- $a1 temp ---- $t0

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

```
swap:   sll  $t1, $a1, 2          #  $t1  =  k  *  4
        add $t1, $a0, $t1         #  $t1  =  v  +  ( k  *  4 )
                                  #  $t1 has the address of  v[ k ]
        lw     $t0, 0($t1)        #  $t0 ← v[ k ]
        lw     $t2, 4($t1)        #  $t2 ← v[ k + 1 ]

        sw     $t2, 0($t1)        #  v[k+1]) → v[ k ]
        sw     $t0, 4($t1)        #  v[k] → v[ k + 1 ]
```

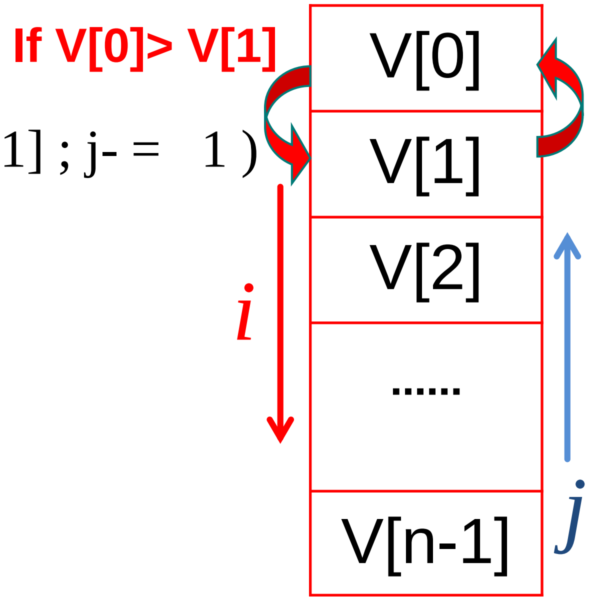
  - **Procedure return**

```
        jr     $ra                #  return to calling routine
```

Procedure *sort*

- C code

```
sort ( int   v[  ] ,   int   n )
{      int   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 -- $a0   n -- $a1   i -- $s0   j -- $s1

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

  $ra , $s0, $s1, $s2, $s3

Code for the procedure *sort*

- **Saving registers**

```
sort:   addi  $sp, $sp, -20     # make room on stack for 5 registers
        sw    $ra, 16($sp)      # save $ra on stack
        sw    $s3, 12($sp)      # save $s3 on stack
        sw    $s2,  8($sp)      # save $s2 on stack
        sw    $s1,  4($sp)      # save $s1 on stack
        sw    $s0,  0($sp)      # save $s0 on stack
```

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

```
exit1:  lw    $s0,  0($sp)      # restore $s0 from stack
        lw    $s1,  4($sp)      # restore $s1 from stack
        lw    $s2,  8($sp)      # restore $s2 from stack
        lw    $s3, 12($sp)      # restore $s3 from stack
        lw    $ra, 16($sp)      # restore $ra from stack
        addi  $sp, $sp, 20      # restore stack pointer
```

- **Procedure return**

```
        jr    $ra               # return to calling routine
```

◻Code for Procedure body

- Outer loop—first for loop

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

Move parameters

```
          move  $s2, $a0          #  $s2 ← $a0 ($a0: base address)
          move  $s3, $a1          #  $s3 ← $a1 ($a1: array size)
```

Outer loop

```
          move  $s0, $zero         #  $s0 ← $zero ( i = 0)
for1tst:  slt   $t0, $s0, $s3      # test  if $s0 >= $s3 (i >= n)
           beq   $t0, $zero, exit1 # go to exit1 if $s0 >= $s3  (i >= n)
          ………………
```

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

```
          ………………
exit2:  addi  $s0, $s0, 1          #  i = i + 1
          j     for1tst            #   jump to test of outer loop
exit1:
```

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  $s1, $s0, -1            # j = i - 1
for2tst:  slti  $t0, $s1, 0             # test  if j < 0
          bne   $t0, $zero, exit2       # go to exit2 if  j < 0
          sll   $t1, $s1, 2             #   $t1 = j * 4
          add   $t2, $s2, $t1           #  $t2 = the address of v[j]
          lw    $t3, 0($t2)             #  $t3 = v[j]
          lw    $t4, 4($t2)             #  $t4 = v[j + 1]
          slt   $t0, $t4, $t3           #  test  if  v[j+1]>=v[j]
          beq   $t0, $zero, exit2       #  go to exit2 if  v[j+1]>=v[j]

          ………………

          （body of first for loop ）

          ………………

          addi  $s1, $s1, -1            #    j = j - 1
          j     for2tst                 # jump to test of inner loop
exit2:
```

- **body of first *for* loop**

Pass parameters and call

```
move  $a0， $s2      #  $a0←$s2 ($s2 : base address of the array )
move  $a1， $s1      #  $a1←$s1 ($a1← j)
```

Call function swap(int v[],int k)

```
jal   swap              # ($a0 might be changed in swap)
```

The Full Procedure

- **Notice:**

1.Why are $a0 and $a1 saved?

$a0 is the base of the array v. $a0 will be used repeatedly and might be(actually not here) changed by the procedure swap.

$a1 is the size of the array v. $a1 will be used repeatedly and changed before the procedure swap is called.

2.Why are they not pushed to stack?

- Register variable is faster

2.15 Arrays versus Pointers

- **Two C procedures**
  - Array version

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

  - Pointer version

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

Pointers

- **Assembly code for clear-1 procedure (array version)**

```
        move  $t0, $zero          #  i = 0
loop1:  sll   $t1, $t0, 2         #  $t1 = i * 4
        add   $t2, $a0, $t1       #  $t2 = address of array[ i ]
        sw    $zero, 0($t2)       #  array[ i ] = 0
        addi  $t0, $t0, 1         #  i = i +1
        slt   $t3, $t0, $a1       #  test if  i  <  size
        bne   $t3, $zero, loop1   # if ( i  <  size ) go to  loop1
        jr    $ra
```

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

In this case， the loop will be executed once even though the value of the size parameter is invalid. Actually， size > 0 must be checked at first.

Pointers

- **Assembly code for clear-2 procedure (pointer version)**

```
        move   $t0, $a0            #  p  =  the start address of the array[]
        sll    $t1, $a1, 2         #   $t1 = size * 4
        add    $t2, $a0, $t1       #   $t2 = &array[size](address of array[size] )
loop2:  sw     $zero, 0($t0)       #  Memory[ p ]  =  0
        addi   $t0, $t0, 4         #  p  =  p  +  4
        slt    $t3, $t0, $t2       #  $t3  =  (p  <  &array[size] )
        bne    $t3, $zero, loop2   # if ( p  <  &array[size] )  go to  loop2
        jr     $ra
```

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

- **Compare the two versions**
  - Array version has the “multiply” and add inside loop
  - Pointer version reduces instructions/iteration from 6 to 4
- **For modern compliers, both ways are the same**

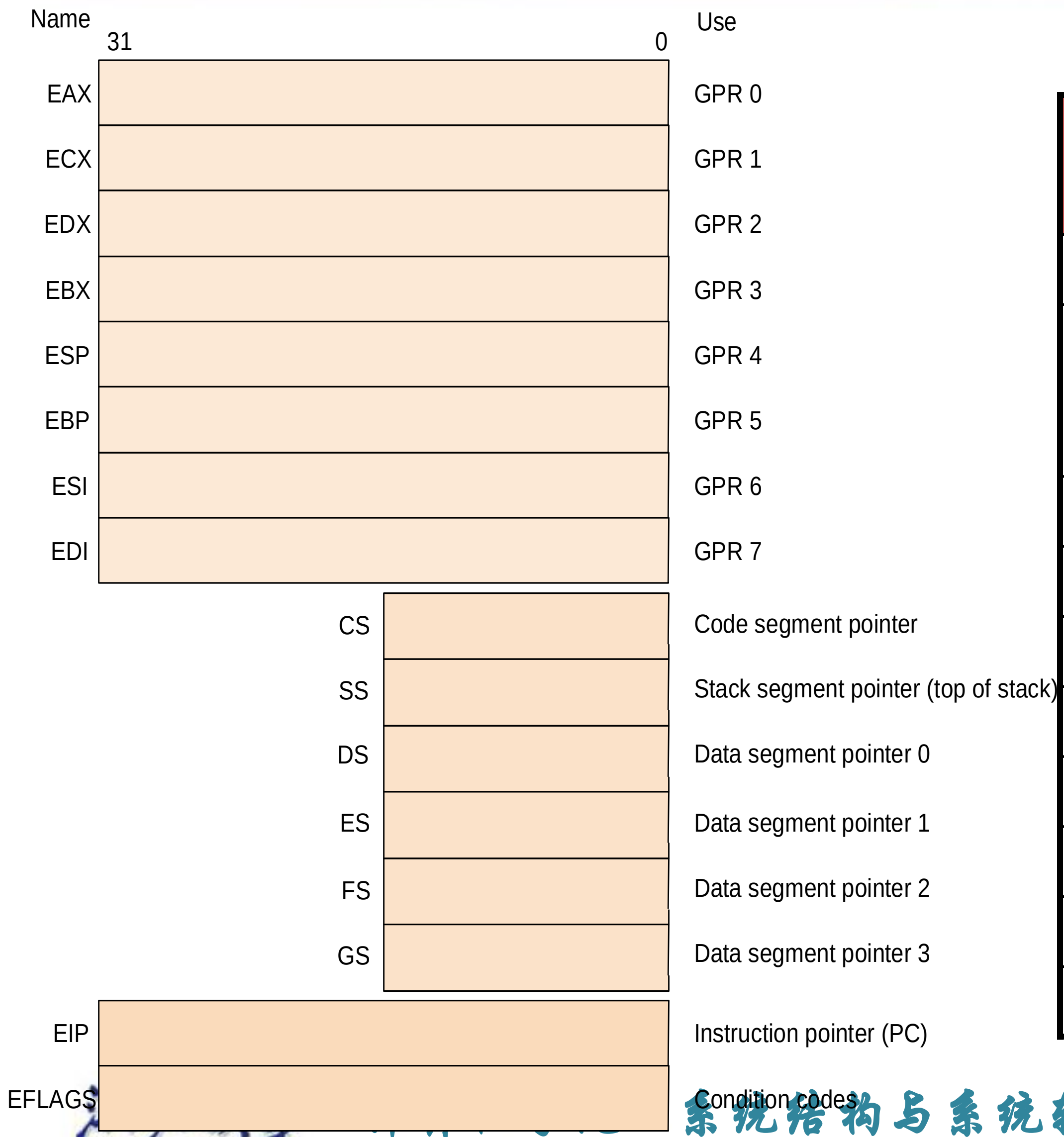
2.16 Real Stuff: IA-32 Instructions

- **The Intel IA-32** **Reading: *2.15~2.20 for 5th Edition**
  - 1978 intel 8086
    - 16-bit architecture
    - Is not considered a general-purpose register
  - 1980 intel 8087 floating-point coprocessor
  - 1982 80286 extended the 8086 architecture by
    - Increasing Address Space to 24 bits
    - Manipulate the protection model
  - 1985 80386 extended the 80286 architecture
    - 32-bit architecture with 32-bit registers
    - 32-bit address space
    - Add paging support in addition to segmented addressing
    - Nearly a general-purpose register machine ?

- 1989~95 Higher performance
  - 80486 in 1989
  - Pentium in 1992
  - Pentium Pro in 1995
- 1997 Expand Pentium and Pentium Pro with MMX
- 1999 Expand Pentium with SSE(SIMD) as Pentium III
  - 8 separate registers ,double their width to 128 bits
  - Add a single-precision floating-point data type
  - 4 32-bit floating-point operations can be performed in parallel
  - Cache prefetch instructions
- 2001 Intel Pentium 4
- 2003 A company other than Intel enhanced the IA-32 architecture
  - AMD
  - Executing all IA-32 instructions with 64-bit Address space & data
- **2004 Intel capitulates and embraces AMD64**

- **80x86 registers and data addressing modes**
  - 80386 extended all 16-bit registers but segment ones to 32 bits
  - GPR (general-purpose register)
  - Addressing modes
    - Register indirect
    - Based mode with 8- or 32-bit displacement
    - Base plus scaled index
    - Base plus scaled index with 8- or 32-bit displacement
- **80x86 integer operations**
  - Data movement instructions
  - Arithmetic and logic instructions
  - Control flow
  - String instructions

IA-32 Register and Data Addressing Modes


Name
31
0
Use
EAX GPR 0
ECX GPR 1
EDX GPR 2
EBX GPR 3
ESP GPR 4
EBP GPR 5
ESI GPR 6
EDI GPR 7
CS Code segment pointer
SS Stack segment pointer (top of stack)
DS Data segment pointer 0
ES Data segment pointer 1
FS Data segment pointer 2
GS Data segment pointer 3
EIP Instruction pointer (PC)
EFLAGS Condition codes

Name	Register no.	Usage
$zero	0	The constant value 0
$v0-$v1	2-3	Values for results and expression evaluation
$a0-$a3	4-7	Arguments
$t0-$t7	8-15	Temporaries
$s0-$s7	16-23	Saved
$t8-$t9	24-25	More temporaries
$gp	28	Global pointer
$sp	29	Stack pointer
$fp	30	Framer pointer
$ra	31	Return address

Instruction types for ALU & data transfer

Source/destination operand type	Second source operand
Register	Register
Register	Immediate
Register	Memory
Memory	Register
Memory	Immediate

Some typical IA-32 Integer Operations

Instruction	Function
JE name	If equal (CC) EIP = name}; EIP – 128 ≤ name < EIP + 128
JMP name	{EIP = NAME};
CALL name	SP = SP – 4; M[SP] = EIP + 5; EIP = name;
MOVW EBX,[EDI + 45]	EBX = M [EDI + 45]
PUSH ESI	SP = SP – 4; M[SP] = ESI
POP EDI	EDI = M[SP]; SP = SP + 4
ADD EAX,#6765	EAX = EAX + 6765
TEST EDX,#42	Set condition codea (flags) with EDX & 42
MOVSL	M[EDI] = M[ESI]; EDI = EDI + 4; ESI = ESI + 4

Typical 80x86 instruction format

a. JE EIP + displacement

4	4	8
JE	Condition	Displacement

1Byte

b. CALL

8	32
CALL	Offset

5Bytes

c. MOV EBX, [EDI + 45]

6	1	1	8	8
MOV	d	w	r-m postbyte	Displacement

3Bytes

d. PUSH ESI

5	3
PUSH	Reg

1Bytes

e. ADD EAX, #6765

4	3	1	32
ADD	Reg	w	Immediate

5Bytes

f. TEST EDX, #42

7	1	8	32
TEST	w	Postbyte	Immediate

6Bytes

2.18 Concluding Remarks

- **Two principles of stored-program computers**
  - Use instructions as numbers
  - Use alterable memory for programs
- **Four design principles**
  - ***Simplicity favors regularity***
  - ***Smaller is faster***
  - ***Make the common case fast***
  - ***Good design demands good compromises***
- **MIPS instruction set**

2.19 History of Instruction Set Development

- **Accumulator Architectures**
  - Only 1 register for arithmetic instructions: *accumulator*
  - Memory-based operand-addressing mode
- **Example 2.23** **Compiling C code to accumulator instructions**
  - C code: A = B + C ;
  - Accumulator instructions:

```
load   AddressB     # Acc = Memory[AddressB], or Acc  =  B
add    AddressC     # Acc = Acc + Memory[AddressC], or Acc = B + C
store  AddressA     # Memory[AddressA] = Acc, or A = B + C
```

- **Extended Accumulator Architectures**
  - Intel *i86*

General-Purpose Register Architectures

- Register-memory architecture
  - 80386
  - IBM 360
- Load-store or register-register architecture
  - CDC 6600
  - MIPS
- DEC’s VAX architecture
  - Allow any combination of registers and memory operands
  - Memory-memory architecture

Example 2.24 Compiling C code to memory-memory instructions

- C code: A = B + C;
- instructions:

```
add      AddressA,  AddressB,  AddressC
```

Compact Code and Stack Architectures

- Variable-length instructions
  - To match the varying operand specifications
  - To minimize code size
- Stack model of execution
  - All registers are abandoned, so the instructions are short.
  - Push, pop

Example 2.25 Compiling C code to stack instructions

- C code: A = B + C;
- Stack instructions:

```
push    AddressC    # Top = Top+4; Stack[Top]=Memory[AddressC]
push    AddressB    # Top = Top+4; Stack[Top]=Memory[AddressB]
add                 # Stack[Top-4]= Stack[Top]+Stack[Top-4];Top=Top-4;
pop      AddressA   # Memoryp[AddressA]=Stack[Top]; Top=Top-4;
```

- **High-Level-Language Computer Architecture**
  - Goal: hardware more like programming languages
  - Finally failed
- **Reduced Instruction Set Computer Architecture (RISC)**
  - Fixed instruction lengths
  - Load-store instruction sets
  - Limited addressing modes
  - Limited operations
  - MIPS, Sun SPARC, Hewlett-Packard PA-RISC
  - IBM PowerPC, DEC Alpha

⦿END