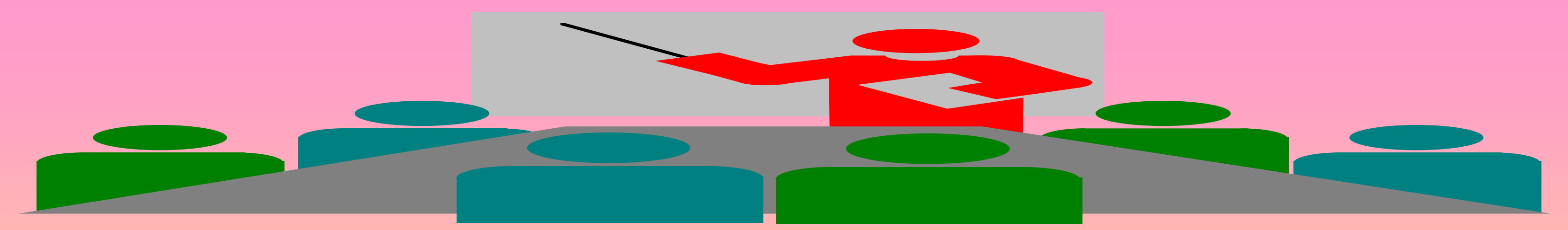
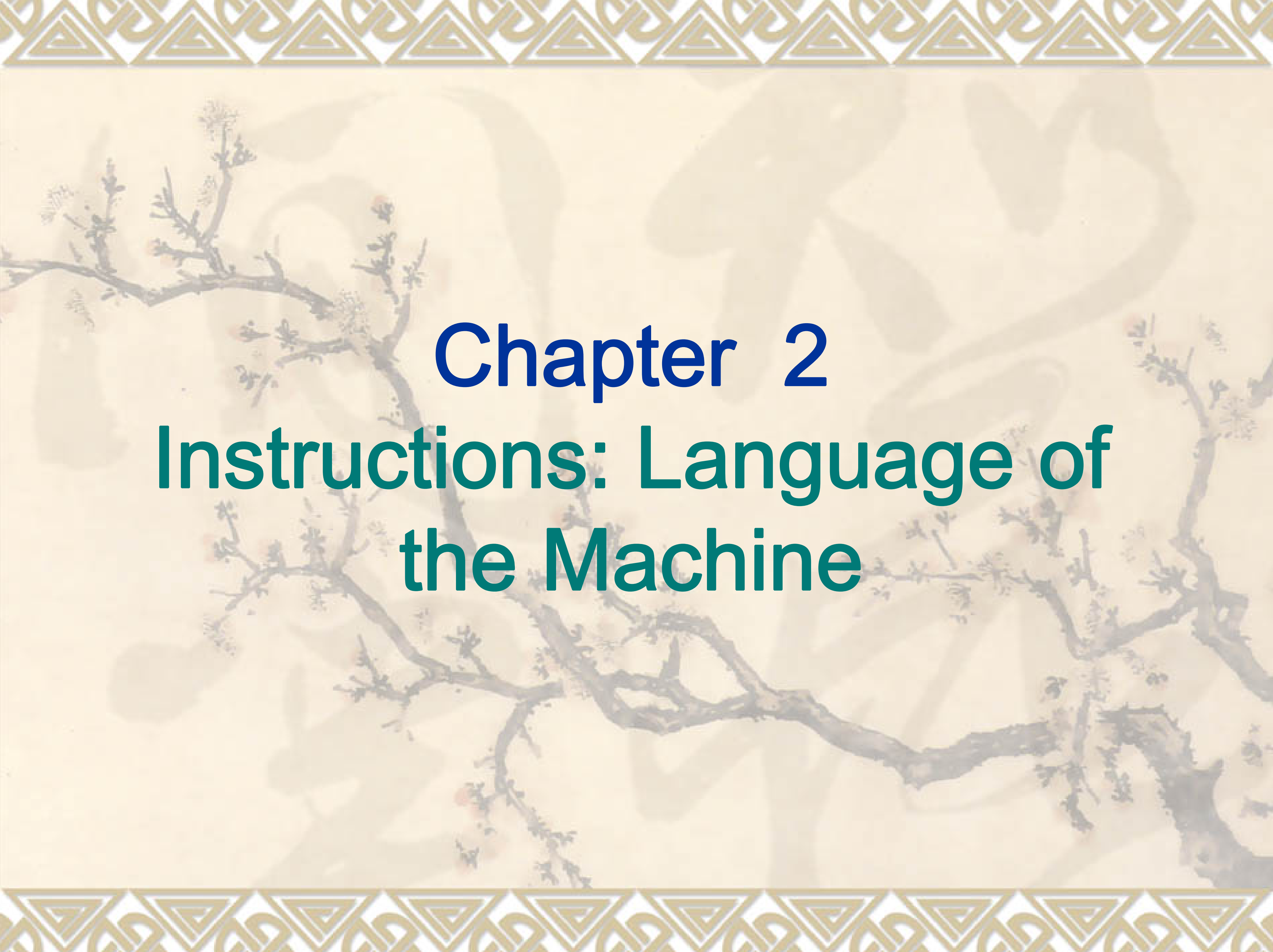


Computer Organization & Design

The Hardware/Software Interface

施青松

http://zjsqs.hzs.cn Email: zjsqs@zju.edu.cn


Chapter 2
Instructions: Language of the Machine

Contents of Chapter 2

- 2.1 Introduction
- 2.2 Operations of the Computer Hardware
- 2.3 Operands of the Computer Hardware
- 2.4 Representing Instructions in the Computer
- 2.5 Logical Operation
- 2.6 Instructions for Making Decisions
- 2.7 Supporting Procedures in Computer Hardware
- 2.8 Communicating with People
- 2.9 MIPS Addressing for 32 Bit Immediates and Addresses

- 2.10 Translanting and starting a Program
- 2.11 *How Compilers Optimize
- 2.12 *How Compilers Work
- 2.13 A C Sort Example to Put It All together
- 2.14 *Implementing an Object-Oriented Language
- 2.15 Arrays Versus Pointers
- 2.16 Real Stuff: IA-32 Instructions
- 2.17 Fallacies and Pitfalls
- 2.18 Concluding Remarks
- 2.19 Historical Perspective and Further Reading

2.1 Introduction

- Language of the machine
  - Instructions
  - Instruction set
- Design goals
  - Maximize performance
  - Minimize cost
  - Reduce design time
- Our chosen instruction set: MIPS
  - Similar to other ones developed since the 1980's
  - Used by NEC, Nintendo, Silicon Graphics, Sony

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***
- Example 2.1 Compiling two simple C statements
  - C code:

```
a = b + c;
d = a – e;
```

- MIPS code:

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

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
```

MPIS assembly language

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

2.3 Operands of the Computer Hardware

- **Register Operands**
  - Arithmetic instructions operands must be registers
  - Difference between the variables of a programming
    - Registers is limited number of
      - 32 registers in MIPS
      - 32 bits for each register in MIPS
- **Design Principle 2**
  - ***Smaller is faster***
- MIPS convention for registers
  - $s0, $s1, … for registers corresponding to variables in C and Java
  - $t0, $t1, … for temporary registers for compiler

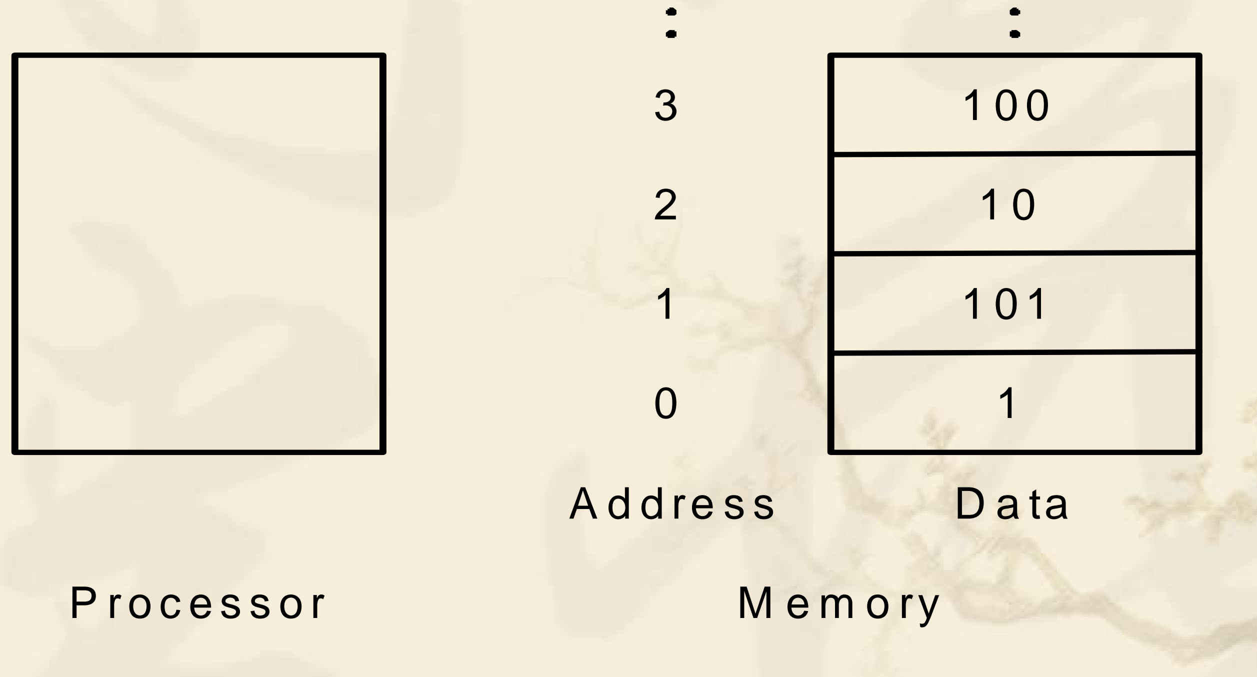
- Example 2.3 Compiling a C statement using registers
  - C code

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

  - MIPS code

```
add   $t0, $s1, $s2    # $t0 contains g + h
add   $t1, $s3, $s4    # $t1 contains i + j
sub   $s0, $t0, $t1    # f gets $t0 - $t1
```

- **Memory operands**
  - 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   Compiling with an operand in memory

C code:

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

(Assume: g ---- $s1   h ---- $s2   base address of A ---- **$s3**)

MIPS code:

```
lw    $t0, 8($s3)      #  temporary reg $t0 gets A[8]
add   $s1, $s2, $ t0    # g = h + A[8]
```

- Offset: the constant in a data transfer instruction →8(**$s3**)
- Base register: the register added to form the address

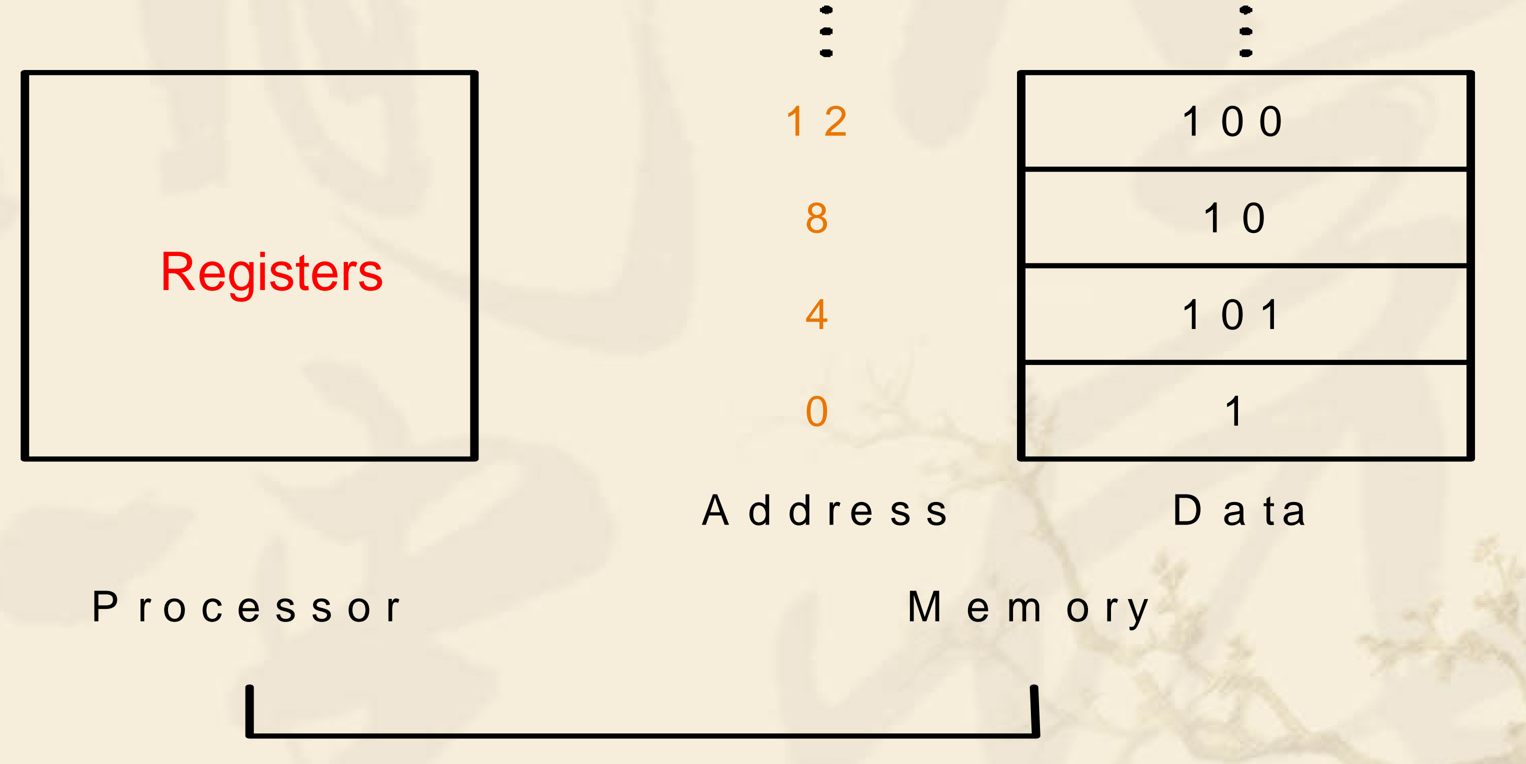
- Byte addressing →8($s3)
- Alignment restriction
  - Addresses of words are multiples of 4 in MIPS

Software/hardware interface

Compiler allocates data structures to memory

Compiler associating variables with registers

Actual MIPS memory addresses and contents



The offset to be added to $s3 in Example 2.4 must be 4×8

Example 2.5 Compiling using load and store

C code:

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

(Assume: h ---- $s2   base address of A ---- $s3)

- MIPS code:

```
lw     $t0 , 32($s3)     #  temporary reg $t0 gets A[8]
add    $t0, $s2, $ t0    #  temporary reg $t0 gets h + A[8]
sw     $t0, 48($s3)      # stores  h + A[8]  back into A[12]
```

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.

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 $s3
  - Assuming AddrConstants 4 is address pointer of constant 4

```
lw $t0, AddrConstant4($s1)   # $t0=constant 4
add $s3, $s3, $t0            #$s3=$s3+$t0($t0==4)
```


- Immediate: Other method for adding constant 4 to $s3
  - Avoids the load instruction
  - Offer versions of the instruction

```
addi   $s3, $s3, 4          #$s3= $s3+ 4
```

Design Principle 3

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

  Constant operands occur frequently
  it is very common
  Loading them from memory is very slow

MIPS operands p59

Name	Example	Comments
32 register	$s0,$s1…… $t0, $t1…….	Fast locations for data. In MIPS, data must be in registers to perform arithmetic.
2^{30} memory words	Memory[0], Memory[4] , …… , Menory[4294967292]	Accessed only by data transfer instructions in MIPS. MIPS uses byte addr. , so sequential word addr. Differ by 4. Memory holds data structures, arrays, and spilled registers.

MIPS assembly language

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

2.4 Representing Instructions in the Computer ----Instruction Format

- All information in computer consists of binary bits
- Mapping registers into numbers
  - Map registers **$s0 to $s7** onto registers **16 to 23**
  - Map registers **$t0 to $t7** onto registers **8 to 15**
- Example 2.7 Translating assembly into machine instruction
  - MIPS code

```
add   $t0, $s1, $s2
```

Sometime use Hex!

  - Decimal version of machine code

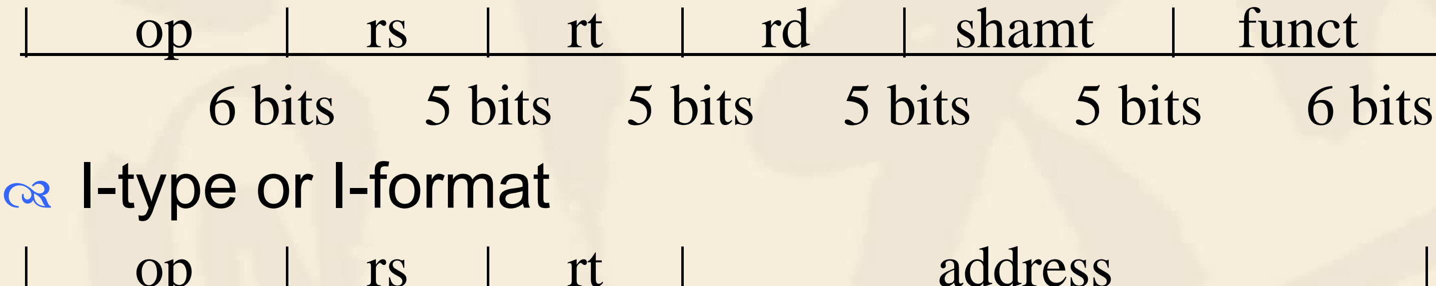
0	17	18	8	0	32

  - Binary version of machine code

000000	10001	10010	01000	00000	100000
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

MIPS fields (format)

p63



Must bear in mind !

Region: $\pm 2^{15}$

- *op*: basic operation of the instruction, traditionally called the opcode.
- *rs*: the first register source operand.
- *rt*: the second register source operand.
- *rd*: the register destination operand.
- *shamt*: shift amount.
- *funct*: function,this field selects the specific variant of the operation in the op field.

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

- Example 2.8 Translating assembly into machine instruction

  C code:

  A[300] = h + A[300] ;

  (Assume: h ---- $s2 base address of A ---- $t1)

  - MIPS assembly code:

```
lw   $t0, 1200($t1)   # temporary reg $t0 gets A[300]
add  $t0, $s2, $t0    # temporary reg $t0 gets h + A[300]
sw   $t0, 1200($t1)   # stores h + A[300] back into A[300]
```

  - MIPS machine language code:
    - Decimal version

	op	rs	rt	rd	address/ shamt	funct
lw	35	9	8	1200		
add	0	18	8	8	0	32
sw	43	9	8	1200		

- Binary version

lw	100011	01001	01000	*0000 0100 1011 0000		
add	000000	10010	01000	01000	00000	100000
sw	101011	01001	01000	0000 0100 1011 0000		

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

Stored-program concept

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

MIPS operands, assembly and machine language

Name	Example	Comments
32 register	$s0,$s1……,$s7 $t0, $t1…….,$t7	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. Registers $s0-$s7 map to 16-23 and $t0-$t7 map to 8-15.
2^30 memory words	Memory[0], Memory[4] , …… , Menory[4294967292]	Accessed only by data transfer instructions in MIPS. MIPS uses byte addr., so sequential word addr. Differ by 4. 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**

2.5 Logical Operation

- Operating some bits within word or individual bit

<table>
<tr><th>Logic operations</th><th>C operators</th><th>Java operators</th><th>MIPS instructions</th></tr>
<tr><td>Shift left</td><td>〈〈</td><td>〈〈</td><td>sll</td></tr>
<tr><td>Shift right</td><td>〉〉</td><td>〉〉〉</td><td>srl</td></tr>
<tr><td>Bit-by-bit AND</td><td>&</td><td>&</td><td>And, andi</td></tr>
<tr><td>Bit-by-bit OP</td><td>|</td><td>|</td><td>Or, ori</td></tr>
<tr><td>Bit-by-bit NOT</td><td>~</td><td>~</td><td>nor</td></tr>
</table>

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

```
sll $t2, $s0, 4   #reg $t2=reg $s0<<4 bit
```

op	rs	rt	rd	shamt	funct
0	0	16	10	4	0

AND operator

- It is bit-by-bit
  - Result=1 : both bits of the operands are 1

$t2:

0000 0000 0000 0000 0000 1101 0000 0000

$t1:

0000 0000 0000 0000 0011 1100 0000 0000

and $t0, $t1, $t2 #reg $t0=reg $t1 & reg $t2

Result:

0000 0000 0000 0000 0000 *11*00 0000 0000

OR operator

- It is bit-by-bit
  - Result=1 : ***either*** bits of the operands is 1

$t2:

0000 0000 0000 0000 0000 1101 0000 0000

$t1:

0000 0000 0000 0000 0011 1100 0000 0000

```
or $t0, $t1, $t2          #reg $t0=reg $t1 | reg $t2
```

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

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

Result:

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

MIPS operands

Name	Example	Comments
32 register	$s0,$s1……,$s7 $t0, $t1…….,$t7	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. Registers $s0-$s7 map to 16-23 and $t0-$t7 map to 8-15.
2^{30} memory words	Memory[0], Memory[4] , …… , Menory[4294967292]	Accessed only by data transfer instructions in MIPS. MIPS uses byte addr., so sequential word addr. Differ by 4. Memory holds data structures,arrays, and spilled registers.

MIPS assembly language

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**	+constants; overflow detected
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
logical	and	and $s1,$s2,$s3	$s1=$s2 & $s3	**three reg. operands;bit-by-bit AND**
	or	or $s1,$s2,$s3	$s1=$s2 | $s3	**three reg. operands;bit-by-bit OR**
	nor	nor $s1,$s2,$s3	$s1=~($s2 | $s3)	**three reg. operands;bit-by-bit NOR**
	and immediate	addi $s1,$s2,100	$s1=$s2 & 100	Bit-by-bit AND reg with constant
	or immediate	ori $s1,$s2,100	$s1=$s2 | 100	Bit-by-bit OR reg with constant
	shift left logical	sll $s1,$s2,10	$s1=$s2 << 10	Shift left by constant
	shift right logical	srl $s1,$s2,10	$s1=$s2 >> 10	Shift right by constant

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 ---- $s0 ~ $s4)
  - C code:

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

  - MIPS assembly code:

```
        beq   $s3, $s4, L1    # go to L1 if i equals j
        add   $s0, $s1, $s2   # f = g + h ( skipped if i equals j )
L1:     sub   $s0, $s0, $s3   # f = f - i ( always executed )
```

Example 2.10

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

(Assume: f ~ j ---- $s0 ~ $s4)

- C code:

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

i↑j

i==j?

F=g+h

F=g - h

Exit:

- MIPS assembly code:

```
        bne   $s3, $s4, Else   # go to Else if i ≠ j
        add   $s0, $s1, $s2    # f = g + h ( Executed if i == j if)
        j     Exit             # go to Exit
Else:   sub   $s0, $s1, $s2    # 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 ---- $s1 ~ $s4   base of A[i] ---- $s5)

  - C code:

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

  - MIPS assembly code:

```
Loop:   add   $t1, $s3, $s3      # temp reg $t1 = 2 * i
        add   $t1, $t1, $t1      # temp reg $t1 = 4 * i
        add   $t1, $t1, $s5      # $t1 = address of A[i]
        lw    $t0, 0($t1)        # temp reg $t0 = A[i]
        add   $s1, $s1, $t0      # g = g + A[i]
        add   $s3, $s3, $s4      # i = i + j
        bne   $s3, $s2, Loop     # go to Loop if i != h
```

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

(Assume: i ~ k---- $s3 ~ $s5 base of save ---- $s6)

- C code:

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

- MIPS assembly code:

```
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:
```

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 $t0, $s3, $s4 # $t0=1 if $s3 < $s4

- Example 2.13 Compiling a less than test

  (Assume: a ---- $s0 b ---- $s1)

  - C language:

    if (a < b), goto Less

  - MIPS assembly code:

```
slt   $t0, $s0, $s1        # $t0 gets 1  if  $s0  < $s1   ( a < b)
bne   $t0, $zero, Less  # go to Less  if  $t0  !=  0  (that is,  if  a  <  b )
```

Hold out Case/Switch

- used to select one of many alternatives
- Example 2.14

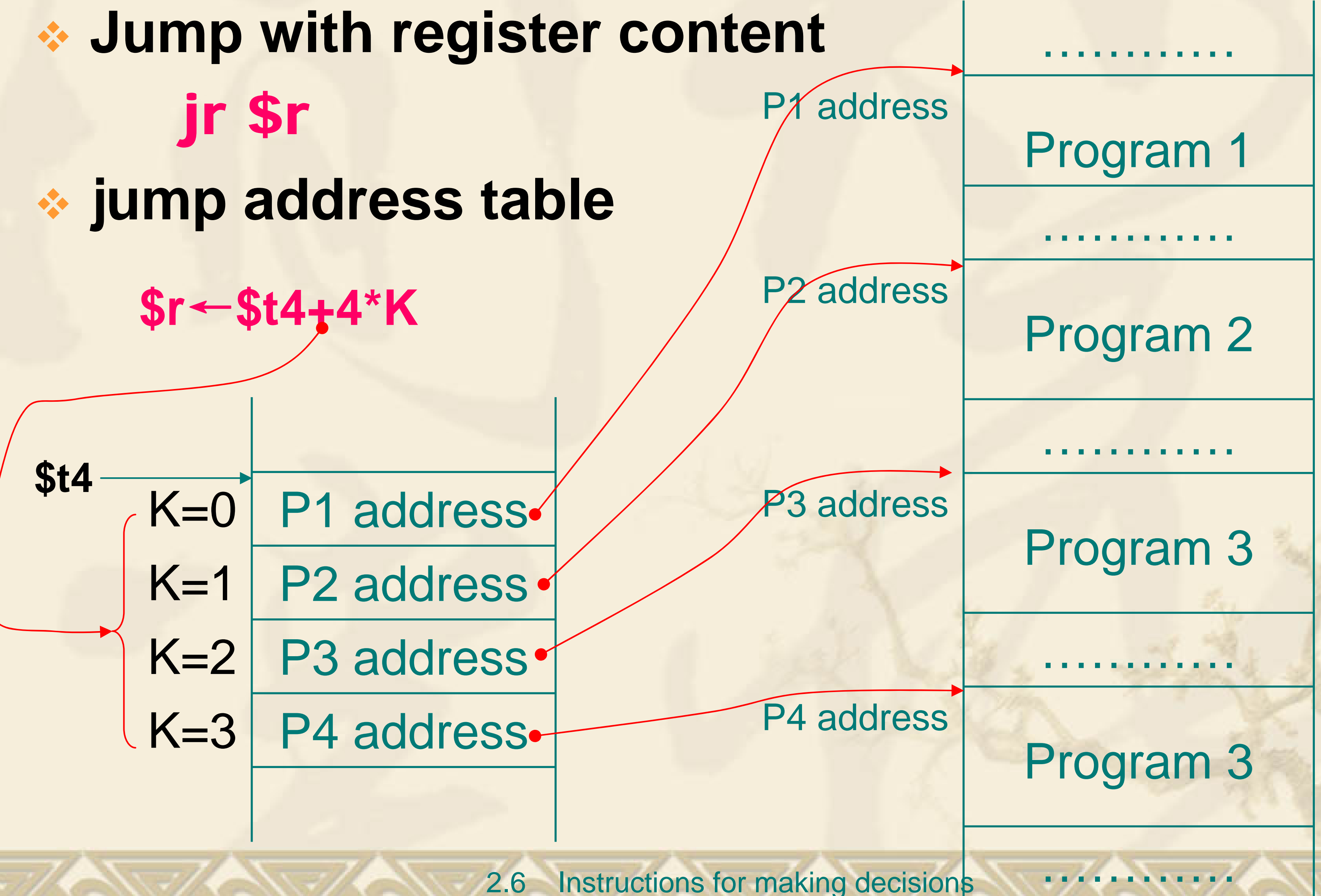
Compiling a switch using ***jump address table***

(Assume: f ~ k ---- $s0 ~ $s5 $t2 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



MIPS assembly code:

```
slt   $t3, $s5, $zero      # test if  k  <  0
bne   $t3, $zero, Exit     # if  k  <  0,  go to Exit
slt   $t3, $s5, $t2        # test if  k  <  4
beq   $t3, $zero, Exit     # if  k  >=  4,  go to Exit
add   $t1, $s5, $s5        # temp reg $t1  =  2  *  k (0<=k<=3)
add   $t1, $t1, $t1        # temp reg $t1  =  4  *  k
add   $t1, $t1, $t4        # $t1  =  address of JumpTable[k]
lw    $t0, 0($t1)          # temp reg $t0  =  JumpTable[k]
jr    $t0                  # jump based on register $t0
```

jump address table

$t1= $t4+4 * k:

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
```

- Important conception-----***basic block***
  - ***A sequence of instructions without branches (except possibly at end) and without branch target or branch lables (except possibly at the beginning)***

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
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

- Registers for procedure calling
  - $a0 ~ $a3: four argument registers to pass parameters
  - $v0 ~ $v1: two value registers to return values
  - $ra: one return address register to return to origin point
- Instruction for procedures: **jal** (jump-and-link)

  Caller **jal ProcedureAddress**

  Callee **jr $ra**

  PC+4→$ra

- Using more registers
  - Stack: ideal data structure for spilling registers
    - Push, pop
    - Stack pointer ($sp)

Special registers

Example 2.15 Compiling a leaf procedure

(Assume: g ~ j ---- $a0 ~ $a3   f ---- $s0)

- C code:

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

High address

($t1)
($t0)
($s0)

$sp(-12) →

Low address

Save value

Return value

- MIPS assembly code:

```
sub   $sp, $sp,12      # adjust stack to make room for 3 items
sw    $t1, 8($sp)      #These three instructions save three
sw    $t0, 4($sp)      # register $t1,$t0,$s0
sw    $s0, 0($sp)      # Let's consider why it need to be done.
```

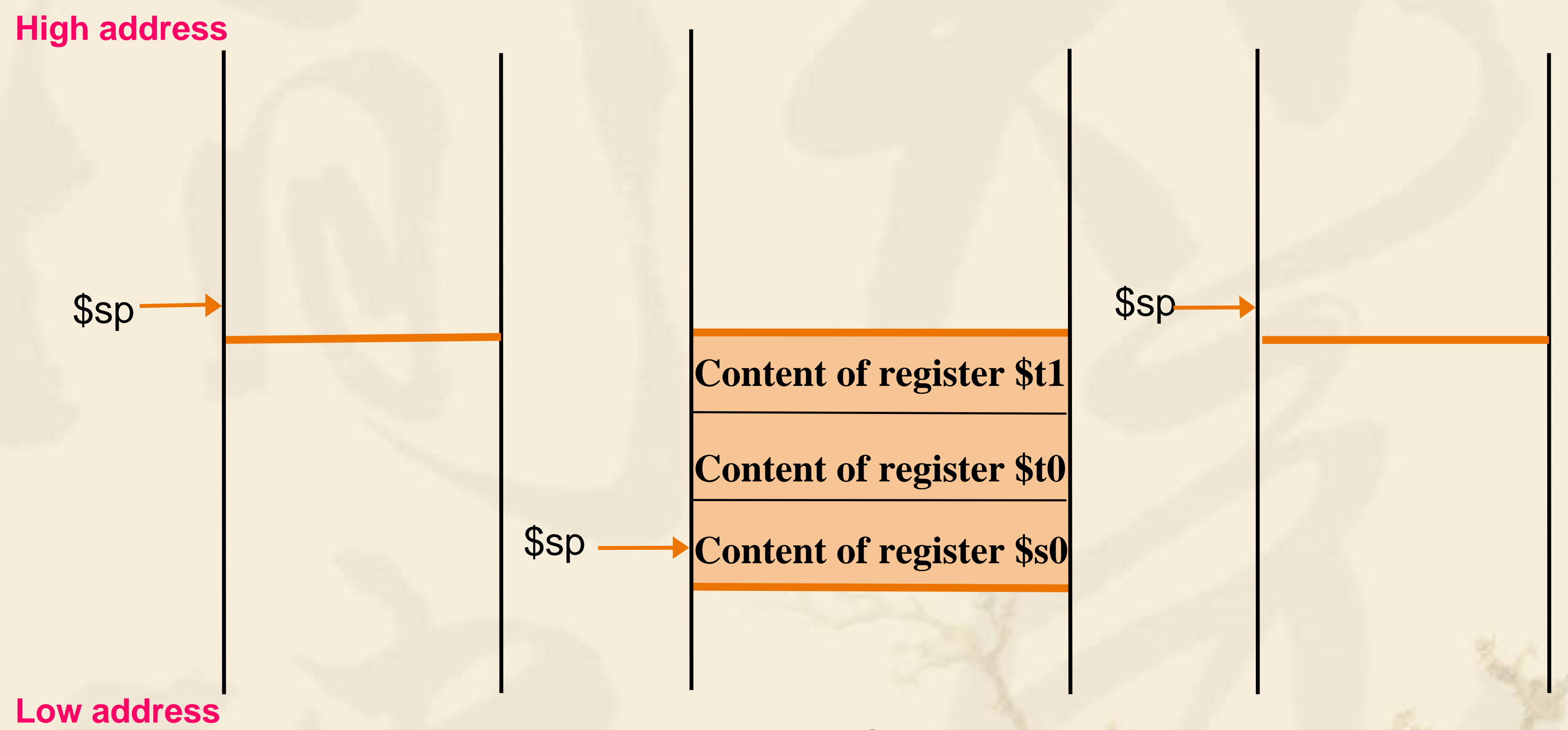
```
add   $t0, $a0, $a1      # register $t0 contains  g + h
add   $t1, $a2, $a3      # register $t1 contains  i + j
sub   $s0, $t0, $t1      # f = $t0 - $t1, which is ( g + h ) – ( i + j )

add   $v0, $s0, $zero    # returns f ( $v0 = $s0 + 0)

lw    $s0, 0($sp)        # restore register $s0 for caller
lw    $t0, 4($sp)        # restore register $t0 for caller
lw    $t1, 8($sp)        # restore register $t1 for caller
add   $sp, $sp, 12       # adjust stack to delete 3 items
jr    $ra                # jump back to calling routine
```

- But maybe some of the three are not used by the caller.So,this way might be inefficient.
  - Two classes of registers
    - $t0 ~ $t9: 10 temporary registers , not preserved
    - $s0 ~ $s7: 8 saved registers, must be preserved

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 $a0~$a3 or $t0~$t9
  - Callee: pushes $ra (return address) and $s0~$s7

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 ) ) ;
}
```

- MIPS assembly code

```
fact:   sub    $sp, $sp, 8               # adjust stack for 2 items
        sw     $ra, 4($sp)               # save the return address
        sw     $a0, 0($sp)               # save the argument  n
        slt    $t0, $a0, 1               # test for  n  <  1
        beq    $t0, $zero, L1            # if  n  >=  1, go to L1(else)
        add    $v0, $zero, 1             # return if n <1
        add    $sp, $sp, 8     # Recover $sp (Why not recover $ra and $a0 ?)
        jr     $ra                       # return to after jal
```

```
L1: sub   $a0, $a0, 1        # n >= 1: argument gets ( n - 1 )
    jal   fact               # call fact with ( n - 1 )
    lw    $a0, 0($sp)        # return from jal: restore argument n
    lw    $ra, 4($sp)        # restore the return address
    add   $sp, $sp, 8        # adjust stack pointer to pop 2 items
    mul   $v0, $a0, $v0      # return n*fact ( n - 1 )
    jr    $ra                # return to the caller
```

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

  Saved registers($s0 ~ $s7), stack pointer register($sp), return address register($ra), stack **above** the stack pointer
- Not preserved things across a procedure call

  Temporary registers($t0 ~ $t9), argument registers($a0 ~ $a3), return value registers($v0 ~ $v1), stack **below** the stack pointer

Stack allocation before, during and after procedure call

High address

$fp

$sp

$fp

Saved argument registers (if any)

Saved return address

Saved saved registers (if any)

Local arrays and structures (if any)

$sp

$fp

$sp

Low address

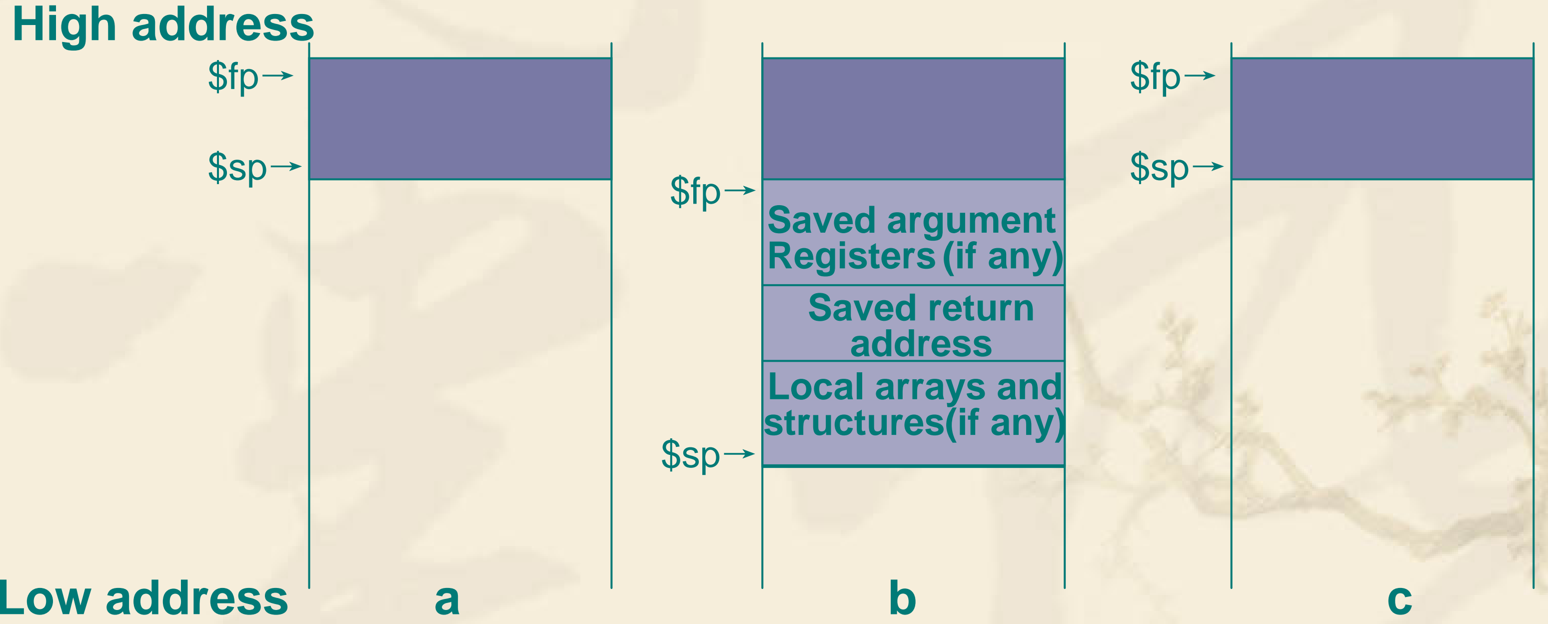
a.

b.

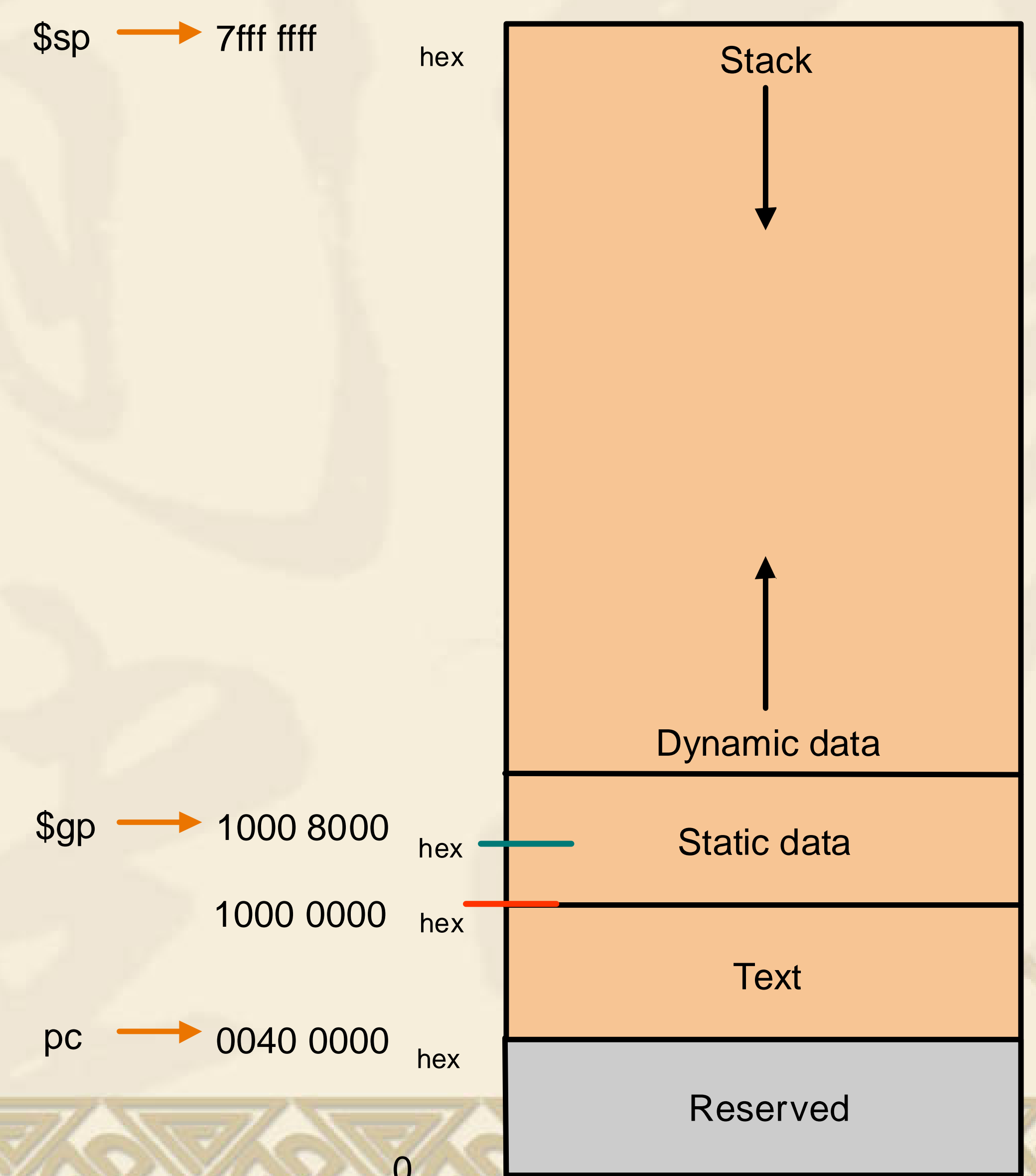
c.

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

- 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



Allocating Space for New Data on the Heap



MIPS operands

Name	Example	Comment
32 registers	$s0-$s7,$t0-$t9. $zero,$a0-$a3,$v0-$v1,	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. $gp(28) is the global pointer, $sp(29) is the stack pointer, $fp(30) is the frame pointer, and $ra(31) is the return address.
2^{30} memory words	Memory[0] Memory[4]…… Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers, such as those saved on procedure calls.

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

MIPS assembly language p89

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
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
logical	and	and $s1,$s2,$s3	$s1=$s2 & $s3	**three reg. operands;bit-by-bit AND**
	or	or $s1,$s2,$s3	$s1=$s2 | $s3	**three reg. operands;bit-by-bit OR**
	nor	nor $s1,$s2,$s3	$s1=~($s2 | $s3)	**three reg. operands;bit-by-bit NOR**
	and immediate	addi $s1,$s2,100	$s1=$s2 & 100	Bit-by-bit AND reg with constant
	or immediate	ori $s1,$s2,100	$s1=$s2 | 100	Bit-by-bit OR reg with constant
	shift left logical	sll $s1,$s2,10	$s1=$s2 << 10	Shift left by constant
	shift right logical	srl $s1,$s2,10	$s1=$s2 >> 10	Shift right by constant
Conditional branch	branch on equal	beq $s1,$s2,L	If($s1==$s2) go to L	Equal test and branch
	branch not eaqal	bne $s1,$s2,L	If($s1!=$s2) go to L	Not equal test and branch
	set on less than	slt $s1,$s2,$s3	If($s2<$s3) $s1=1 Else $s1=0	Compare less than;used with beq, bne
	set on less than immediate	slt $s1,$s2100	If($s2<100) $s1=1 Else $s1=0	Compare less than immediate; used with beq, bne
Uncondition-al jump	jump	j L	go to L	Jump to target address
	jump register	jr $ra	go to $ra	For procedure return
	jump and link	jal L	$ra=PC+4; go to L	For procedure 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

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

Instruction

001111	00000	01000	0000 0000 1111 1111

Register

Filling with "0"

0000 0000 1111 1111	0000 0000 0000 0000

- 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:

	Address							
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:
```

- **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

1. Immediate addressing

op	rs	rt	Immediate

2. Register addressing

op	rs	rt	rd	. . .	funct

Registers

Register

3. Base addressing

op	rs	rt	Address

Register

Memory

Byte Halfword Word

4. PC-relative addressing

op	rs	rt	Address

PC

Memory

Word

5. Pseudodirect addressing

op	Address

PC

Memory

Word

- 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**

    - Determine other fields

rs: $a1; rt: $t7; rd: $s0 **P88**

    - Show the assembly instruction

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

Summary of MIPS architecture in Chapter 2 **P105**

- MIPS instruction format

Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer,branch,imm. format
J-format	op	target address					Jump instruction format

- MIPS operands

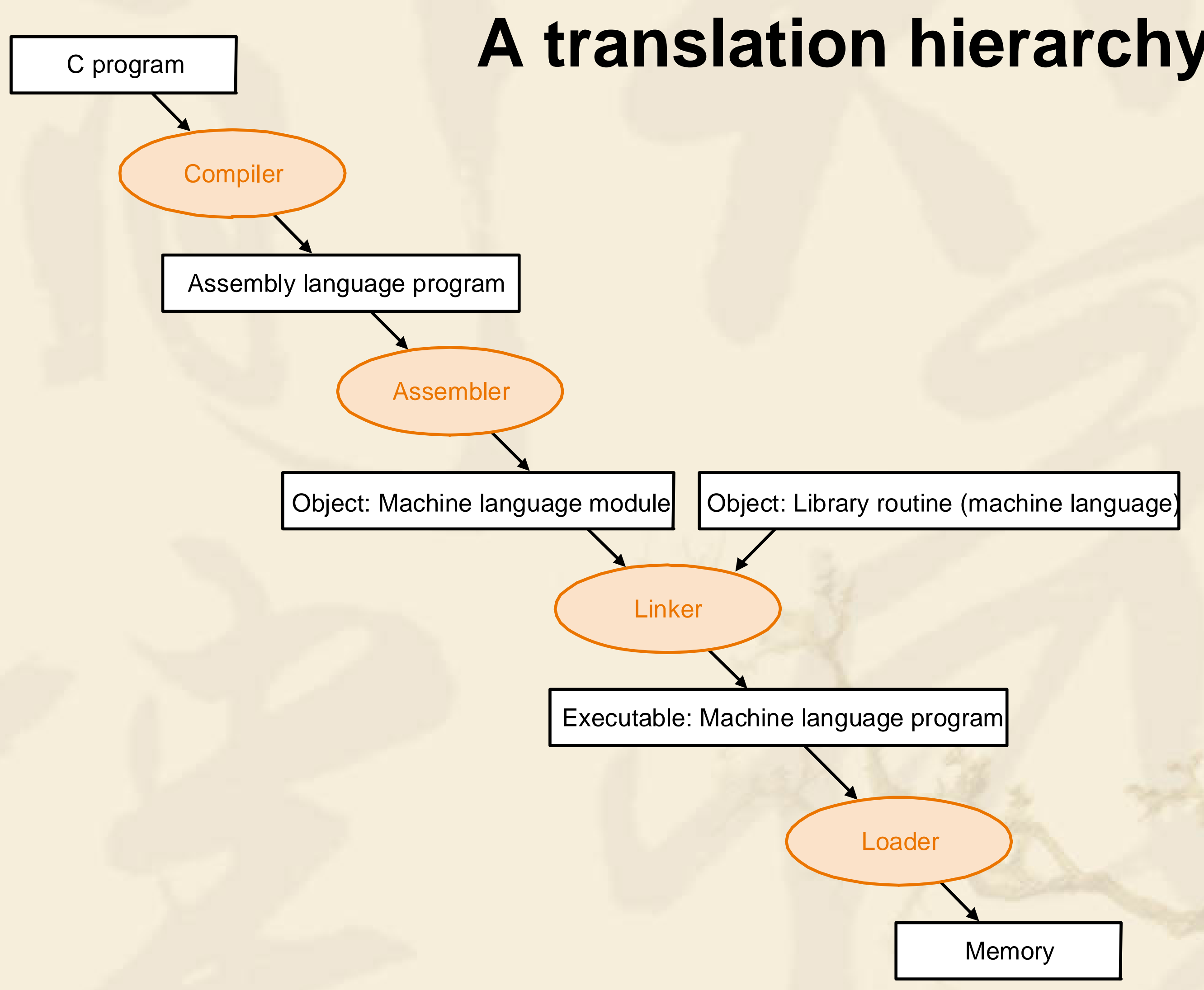
Name	Example
32 registers	$s0~$s7, $t0~$t9,$zero, $a0~$a3, $v0~$v1, $gp $fp, $gp, $ra, $at
Mem words	Memory[0], Memory[4], . . . , Memory[4294967292]

- 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
  - branch on equal beq $s1, $s2, 25(Why not beqi?)
  - 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

2.10 Translanting and starting a Program

A translation hierarchy



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 St0 gets register $t1

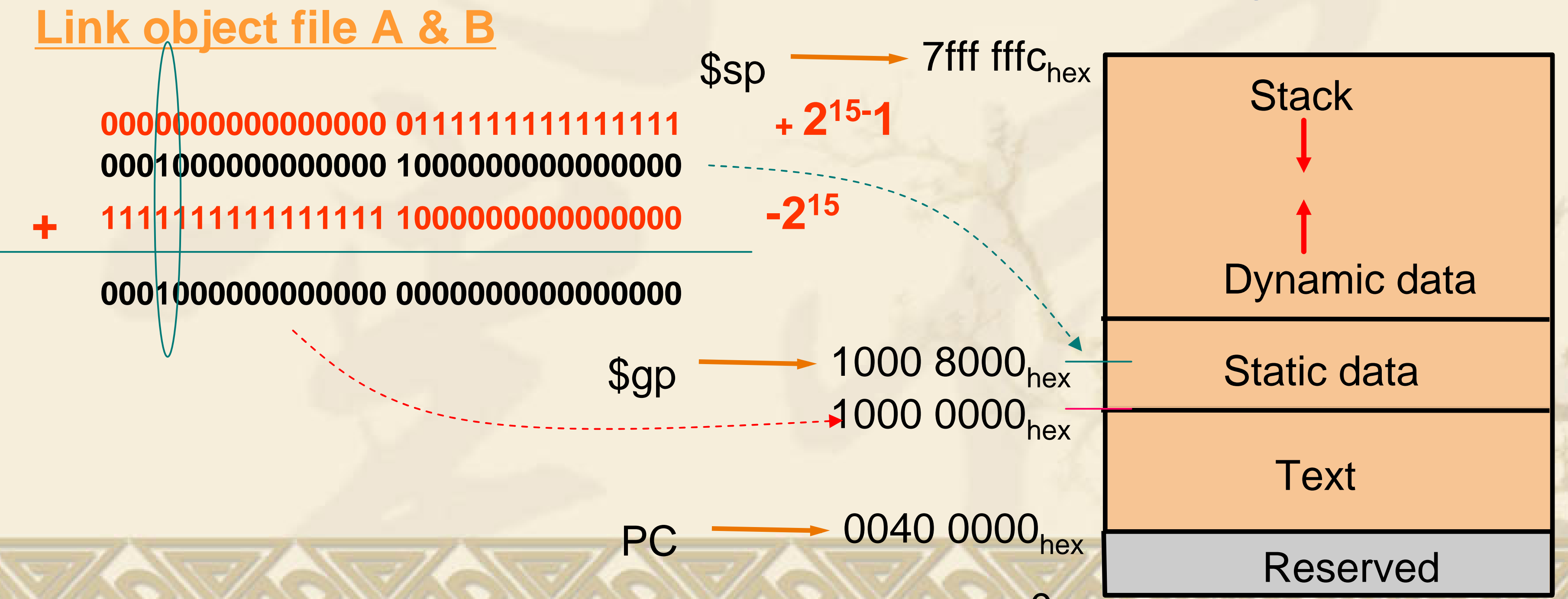
    add $t0,$zero, $t1 # register St0 gets 0+register $t1
  - Symbol table
    - A table that matches name of lables to the addresses of the memory words that instructions occupy.
  - Object file of UNIX (six distinct pieces)
    - object file header—size and position of the other pieces
    - Text segment
    - 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

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 ) ;
      }
}
```

If V[0]> V[1]

V[0]
V[1]
V[2]
······
V[n-1]

- **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:
```

The Full Procedure

- **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)
```

- 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;
}
```

- 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)

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

```
        move   $t0, $a0             #  p = the start address of the array[]
        sll    $t1, $t1, 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.

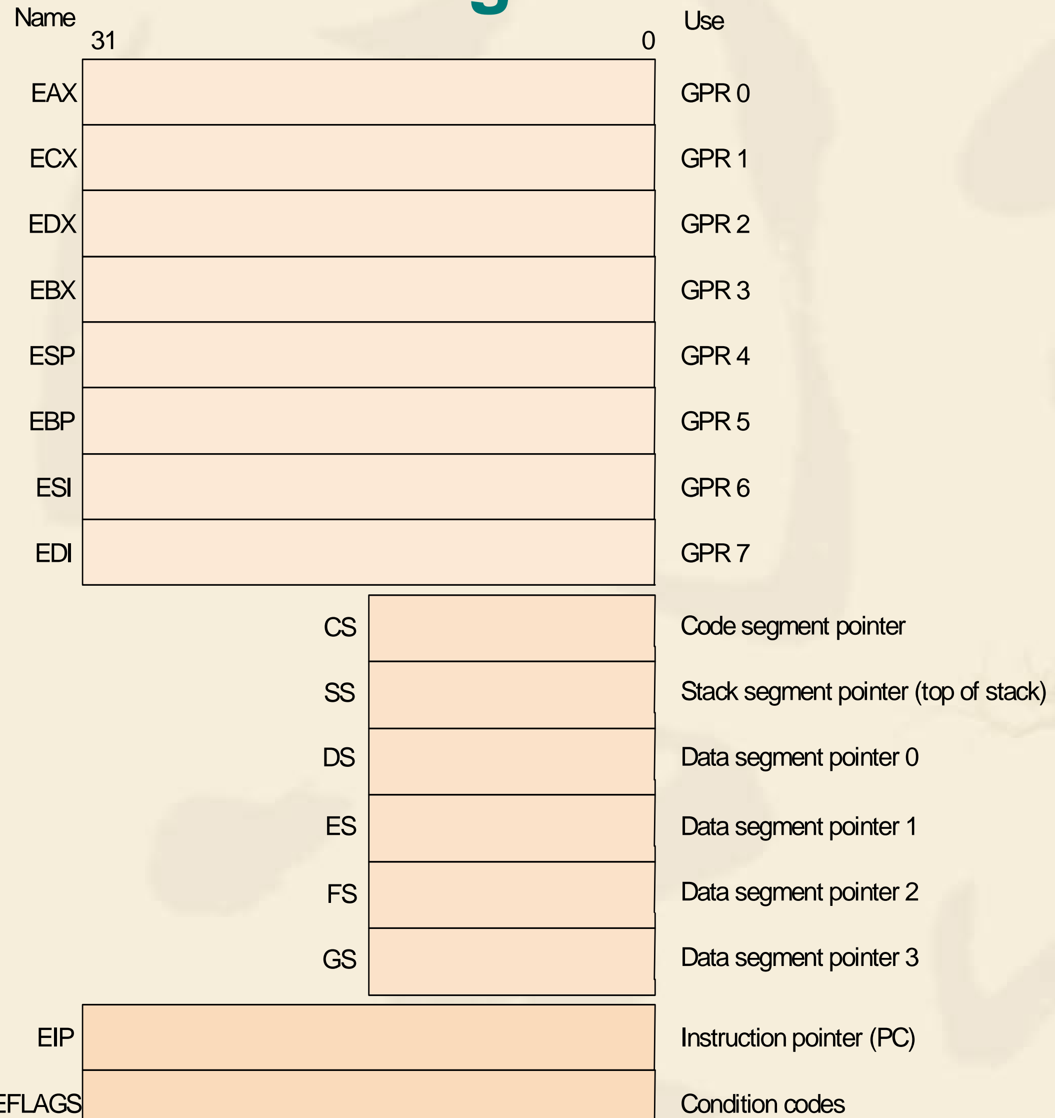
2.16 Real Stuff: IA-32 Instructions

- The Intel IA-32
  - 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	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

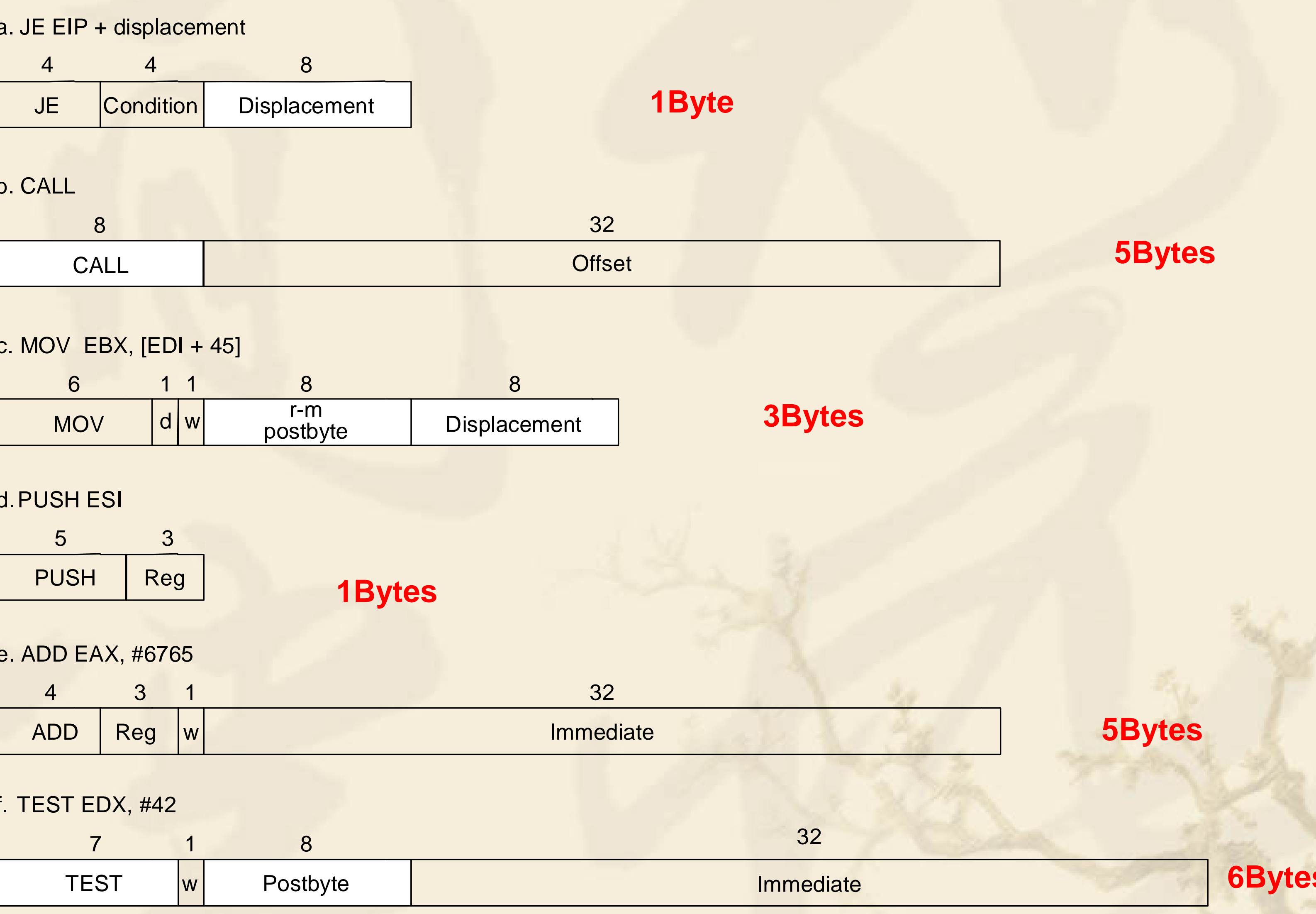
Instruction types for ALU & data transfer

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



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

- 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