Computer Organization and Design
The Hardware/Software Interface

Chapter 2 - Introductions: Language of the Computer

Dr. Feng Li
fli@sdu.edu.cn
https://funglee.github.io

Contents of Chapter 2

- 2.1 Introduction
- 2.2 Operations of the Computer Hardware
- 2.3 Operands of the Computer Hardware
- 2.4 Signed and unsigned Number
- 2.5 Representing Instructions
- 2.6 Logical Operations
- 2.7 Instructions for making decisions
- 2.8 Supporting Procedures
- ……

2.1 Introduction

- Language of the machine
  - Instructions
  - Instruction set
- Different computers have similar languages
- Design goals
  - Maximize performance
  - Minimize cost
  - Reduce design time

2.2 Operations of the Computer Hardware

- **Every computer must be able to perform arithmetic:**
  - **add a, b, c # a=b+c**
  - **Exactly three variables**
- **Design Principle 1**
  - ***Simplicity favors regularity***
  - ***Keeping hardware simple: hardware for a variable number of operands is more complicated than the one for a fixed number***
- **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
```

- **Arithmetic instruction operands must be registers**
  - **32 bits for each register in MIPS**
  - **32 registers in MIPS**
  - **A word in MIPS consists 32 bits**
- **Design Principle 2**
  - ***Smaller is faster***
  - **A very large number of registers may increase the clock cycle time simply since it takes electronic signals longer when they must travel farther**

Registers

- The 32 MIPS registers are partitioned as follows:
  - Register 0 : $zero always stores the constant 0
  - Regs 2-3 : $v0, $v1 return values of a procedure
  - Regs 4-7 : $a0-$a3 input arguments to a procedure
  - Regs 8-15 : $t0-$t7 temporaries
  - Regs 16-23: $s0-$s7 variables
  - Regs 24-25: $t8-$t9 more temporaries
  - Reg 28 : $gp global pointer
  - Reg 29 : $sp stack pointer
  - Reg 30 : $fp frame pointer
  - Reg 31 : $ra return address

• Example 2.3 Compiling a C statement using registers

➢ C code

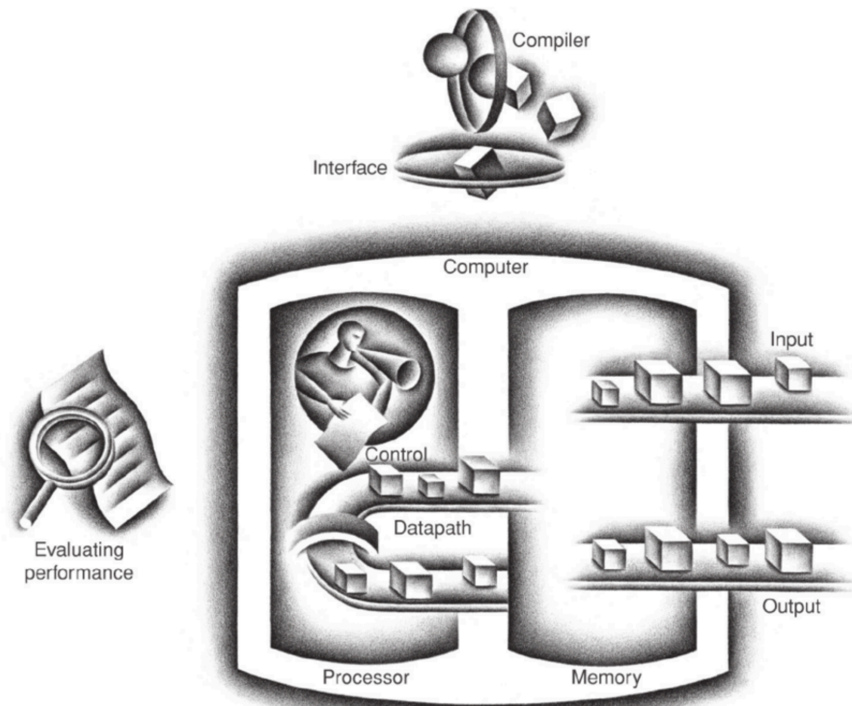
```
f = ( g + h ) – ( i + j ) ;
$s0   $s1   $s2   $s3   $s4
```

➢ MIPS code

```
add  $t0, $s1, $s2    # $t0 contains g + h
add  $t1, $s3, $s4    # $t1 contains i + j
sub  $s0, $t0, $t1    # $s0=$t0 – $t1
```

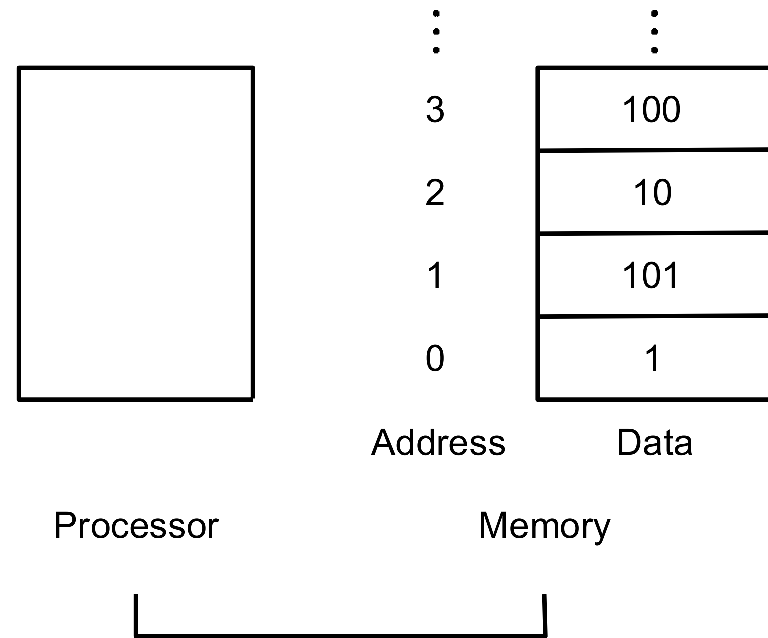
Memory operands

- The number of registers is limited and each register contains only one data elements, what if we need complex date structure which contains much more data?



Memory addresses and contents at those locations

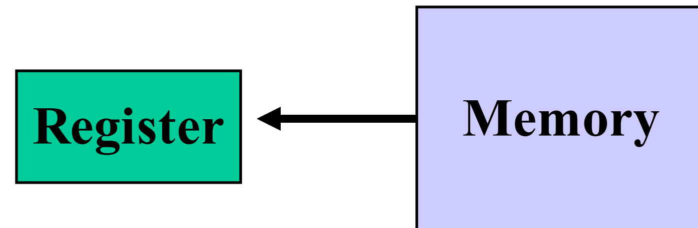
- Memory is a large, single-dimensional array, with the address acting as the index to that array, starting from 0
- Data transfer instructions
  - Load Word (lw): from memory to register
  - Store Word (sw): from register to memory



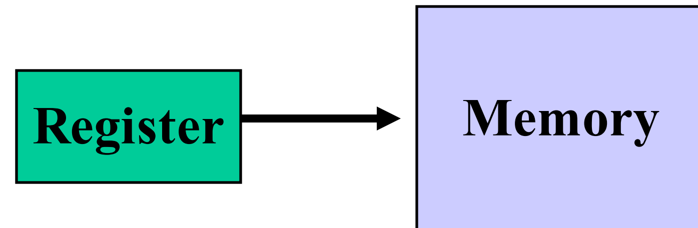
Memory Operands

- Values must be fetched from memory before (add and sub) instructions can operate on them

Load word
lw $t0, memory-address



Store word
sw $t0, memory-address



How is memory-address determined?

Example 2.4 Compiling with an operand in memory

Assume: g -- s1, h -- s2, base address of A -- s3

- **C code:**

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

- **MIPS code:**

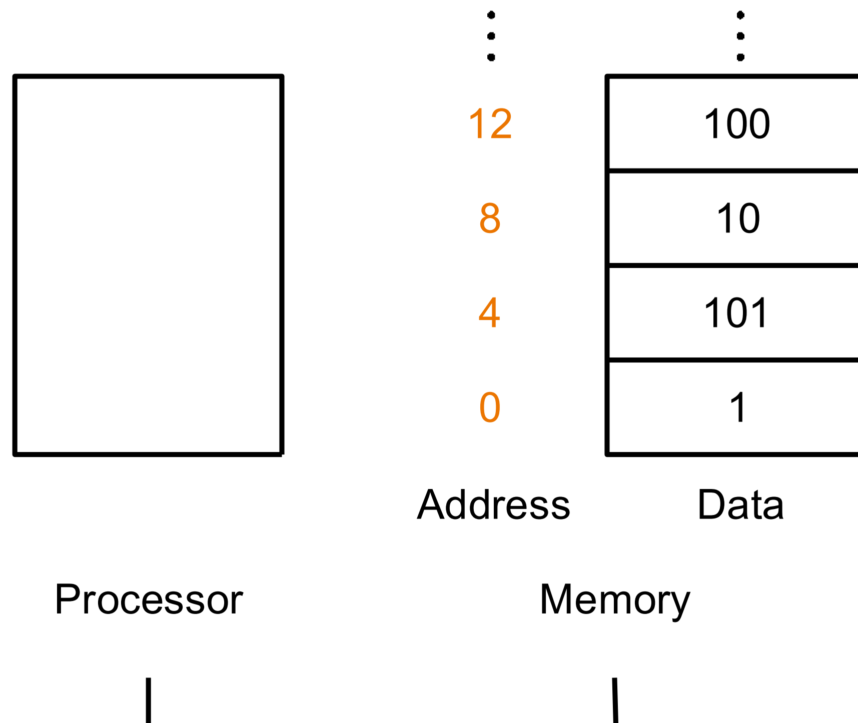
```
lw    $t0 , 8($s3)     # temporary reg $t0 gets A[8]
add   $s1, $s2, $t0    # g = h + A[8]
```

- **Offset: the constant in a data transfer instruction**
- **Base register: the register added to form the address**

- **Byte addressing**
- **Alignment restriction**
  - **Addresses of words are multiples of 4 in MIPS**

Actual memory addresses and contents

- **Alignment restriction**
  - **In MIPS, words must start at addresses that are multiples of 4**



Example 2.5 Compiling using load and store

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

- C code:

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

- 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]
```

Hardware/Software Interface

- What if a program has more variables than a computer has registers?
  - Spilling registers: The compiler tries keep the most frequently used variables in registers, while placing the rest in memory using load/store instructions.
- Principle
  - Registers are faster than memory
  - The number of registers is less
  - Accessing registers consumes less energy

Constant or immediate operands

- **Example**
  - **Add the constant 4 to register $s3**
  - **MIPS code (assuming $s1+ AddrConstant4 is the memory address of the constant 4):**

    lw $t0, AddrConstant($s1) #assume $s1+AddrConstant denotes the address of the memory unit storing constant 4

    add $s3, $s3, $t0
  - **MIPS code (addi: add immediate)**

    addi $s3, $s3, 4
- **Design Principle 3: Make the common case fast**
  - **Since zero can be used frequently to simply the instruction set by offering useful variations, MIPS dedicates a register $zero to be hard-wired to the value zero**
  - **The move operation is just an add instruction where one operand is zero**

Signed and unsigned number

- **Numbers are processed in computer hardware as a series of high and low electronic signals; therefore, they are represented by base 2 numbers (so-called "binary numbers")**
  - **Bits are the "atom" of computing.**

    Binary number (base 2): 0000 0001 0010 0011 0100 0101 0110 …
    Decimal number (base 10): 0...2^n-1
  - **Of course it gets more complicated:**
    - **numbers are finite (overflow)**
    - **fractions and real numbers**
    - **negative numbers**
  - **How do we represent negative numbers?**
    i.e., which bit patterns will represent which numbers?

Numbers and their representation

- **Number systems**
  - **Radix based systems are dominating**

    decimal, octal, binary,…

$0 \leq b \leq K$

$$(N)_k = (A_{n-1}A_{n-2}A_{n-3}\ldots A_1A_0 \bullet A_{-1}A_{-2}A\ldots A_{-m+1}A_{-m})_k$$

Most Significant Bit

$$(N)_K = (\sum_{i=m}^{n-1} b_i \bullet k^i)_k$$

Least Significant Bit

  - ***b*: value of the digit, *k*: radix, *n*: digits left of radix point, *m*: digits right of radix point**
  - **Alternatives, e.g. Roman numbers (or Letter)**
- **Decimal (k=10) -- used by humans**
- **Binary (k=2) -- used by computers**

Recap – Numeric Representations

- **Decimal** $35_{10} = 3 \times 10^1 + 5 \times 10^0$
- **Binary** $00100011_2 = 1 \times 2^5 + 1 \times 2^1 + 1 \times 2^0$
- **Hexadecimal (compact representation)**
  0x 23 or $23_{hex} = 2 \times 16^1 + 3 \times 16^0$

0-15 (decimal) → 0-9, a-f (hex)

Dec	Binary	Hex	Dec	Binary	Hex	Dec	Binary	Hex	Dec	Binary	Hex
0	0000	00	4	0100	04	8	1000	08	12	1100	0c
1	0001	01	5	0101	05	9	1001	09	13	1101	0d
2	0010	02	6	0110	06	10	1010	0a	14	1110	0e
3	0011	03	7	0111	07	11	1011	0b	15	1111	0f

Numbers and their representation

- Representation
  - ASCII - text characters
    - Easy read and write of numbers
    - Complex arithmetic (character wise)
  - Binary number
    - Natural form for computers
    - Requires formatting routines for I/O

Number types

- Integer numbers, unsigned
  - Address calculations
  - Numbers that can only be positive
- Signed numbers
  - Positive
  - Negative
- Floating point numbers
  - numeric calculations
  - Different grades of precision
    - Singe precision (IEEE)
    - Double precision (IEEE)
    - Quadruple precision

Signed number representation

- **First idea:**

  Positive and negative numbers

  - **Take one bit (e.g. 31) as the sign bit**

- **1's complement**
  - **0 0000000 = 0 positive zero!**
  - **1 1111111 = 0 negative zero!**
  - **Each comparison to 0 requires two steps**

Signed number representation

- A common wisdom may suggest that a number can be divided into *sign* and *magnitude*. But is it a good idea?
  - ➢ It is not obvious where to put the sign bit.
  - ➢ An extra step is necessitated.
  - ➢ Both positive zero and negative zero exist, which may resulting in problems for inattentive programmers.
- A smarter solution: *tow's complement* representation

$$
\begin{aligned}
0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} &= 0_{ten} \\
0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} &= 1_{ten} \\
0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} &= 2_{ten} \\
\ldots \qquad &\qquad \ldots \\
0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} &= 2{,}147{,}483{,}645_{ten} \\
0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} &= 2{,}147{,}483{,}646_{ten} \\
0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} &= 2{,}147{,}483{,}647_{ten} \\
1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} &= -2{,}147{,}483{,}648_{ten} \\
1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} &= -2{,}147{,}483{,}647_{ten} \\
1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} &= -2{,}147{,}483{,}646_{ten} \\
\ldots \qquad &\qquad \ldots \\
1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} &= -3_{ten} \\
1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} &= -2_{ten} \\
1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} &= -1_{ten}
\end{aligned}
$$

- **All negative numbers have a 1 in the most significant bit (so-called "sign bit")**
- **Binary to decimal conversion**
  $(x31 \times -2^{31}) + (x30 \times 2^{30}) + (x29 \times 2^{29}) + \cdots + (x1 \times 2^{1}) + (x0 + 2^{0})$
- **Overflow happens when the left most retained bit of the binary bit pattern is not the same as the infinite number of digits to the left (i.e., the sign is incorrect)**
- **Signed versus unsigned applies to loads as well**
  - **Sign extension**

Binary to decimal conversion

- **Binary number**

1111 1111 1111 1111 1111 1111 1111 1100_{two}

- **Decimal number**

$(1\times -2^{31})+(1\times 2^{30}) +(1\times 2^{29})+\ldots+(1\times 2^{2})+(0\times 2^{1})+(0\times 2^{0})$

$=-4_{ten}$

Negation shortcut

- A quick way to negate a two's complement binary number
  - Invert every 0 (resp. 1) to 1 (resp. 0)
  - Add one to the result
- Example
  - Negate 2_{ten}

2_{ten} = $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two}$

$$
\begin{array}{rr}
 & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} \\
+ & 1_{two} \\
\hline
= & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two}
\end{array}
$$

Sign extension shortcut

- **Convert a binary number represented in n bits to a number represented with more than n bits**
  - **Take the most significant bit from the smaller quantity and replicate it to fill the new bits of the larger quantity**
- **Example: 16-bit to 32-bit**
  - **16-bit: 2_{ten} = 0000 0000 0000 0010_{two}**
  - **32-bit: 2_{ten} = 0000 0000 0000 0000 0000 0000 0000 0010_{two}**
  - **16-bit: -2_{ten} = 1111 1111 1111 1110_{two}**
  - **32-bit: -2_{ten} = 1111 1111 1111 1111 1111 1111 1111 1110_{two}**
- **Load byte (lb)**
  - **A byte is treated as a signed number and thus can be sign-extends to fill the 24 left-most bits of the register**
- **Load byte unsigned (lbu)**

Elaboration

- **Two's complement**
  - **The unsigned sum of an n-bit number and its n-bit negative is 2^n**
  - **The negation or complement of a number x is 2^n-x**
- **One's complement**
  - **Inverting each bit from 0 to 1 and from 1 to 0**
  - **The negation or complement of a number x is 2^n-x-1**

Representing instructions in the computer

- Instructions are kept in computers as a series of high and low electronic signals; therefore, they consist of binary bits
- Mapping registers into numbers
  - $s0 to $s7 map onto register 16 to 23
  - $t0 to $t7 map onto register 8 to 15
- All instructions are 32-bit wide.
- Example 2.7 Translating assembly into machine instruction
  - MIPS code

    add $t0, $s1, $s2
  - Binary version of machine code

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

  - Decimal version of machine code

0	17	18	8	0	32

  - Hexadecimal?

R (Register) type instruction

- All operands are in registers.
- E.g. add $t0, $s1, $s2 # $t0=$s1+$s2
  - Decimal version of machine code

0	17 ($s1)	18 ($s2)	8 ($t0)	0	32

  - Binary version of machine code

000000	10001	10010	01000	00000	100000
op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

  - Op: Basic operation of the instration opcode
  - Rs: The first register source.
  - Rt: The second register source operand.
  - Rd: The register destination operand.
  - Shamt: Shift amount.
  - Funct: Function code, selects the specific variant of the operation
- How about substruction operation?
  - sub $t1, $t2, $t3 # $t1=$t2-$t3
  - funct = 34_{ten}

I (Immediate) type instruction

- Load: from memory to register;   load word (lw)
  - ➢ lw $t0, 1200($t1)   # $t0=A[300]=memory($t1+1200)
  - ➢ Decimal version of machine code

35	9	8	1200

- ➢ Binary version of machine code

100011	01001	01000	0000 0100 1011 0000
op	rs	rt	address
6 bits	5 bits	5 bits	16 bits

- Store: from register to memory; store word(sw)
  - ➢ sw $3, 32($4)   # memory（$4+32）=$3

Keep them simple and similar

Instruction	Format	op	rs	rt	rd	shamt	funct	address
add	R	0	reg	reg	reg	0	32_{ten}	n.a.
sub (subtract)	R	0	reg	reg	reg	0	34_{ten}	n.a.
add immediate	I	8_{ten}	reg	reg	n.a.	n.a.	n.a.	constant
lw (load word)	I	35_{ten}	reg	reg	n.a.	n.a.	n.a.	address
sw (store word)	I	43_{ten}	reg	reg	n.a.	n.a.	n.a.	address

FIGURE 2.5 MIPS instruction encoding. In the table above, "reg" means a register number between 0 and 31, "address" means a 16-bit address, and "n.a." (not applicable) means this field does not appear in this format. Note that add and sub instructions have the same value in the op field; the hardware uses the funct field to decide the variant of the operation: add (32) or subtract (34).

Translating assembly into machine instruction

- **C language:**
  - A[300]=h+A[300]
- **MIPS ASM**
  - lw $t0, 1200($t1) # $t0=A[300]
  - add $t0, $s2, $t0 # $t0=$s2+A[300]
  - sw $t0, 1200($t1) # A[300]= $t0
- **Machine language**

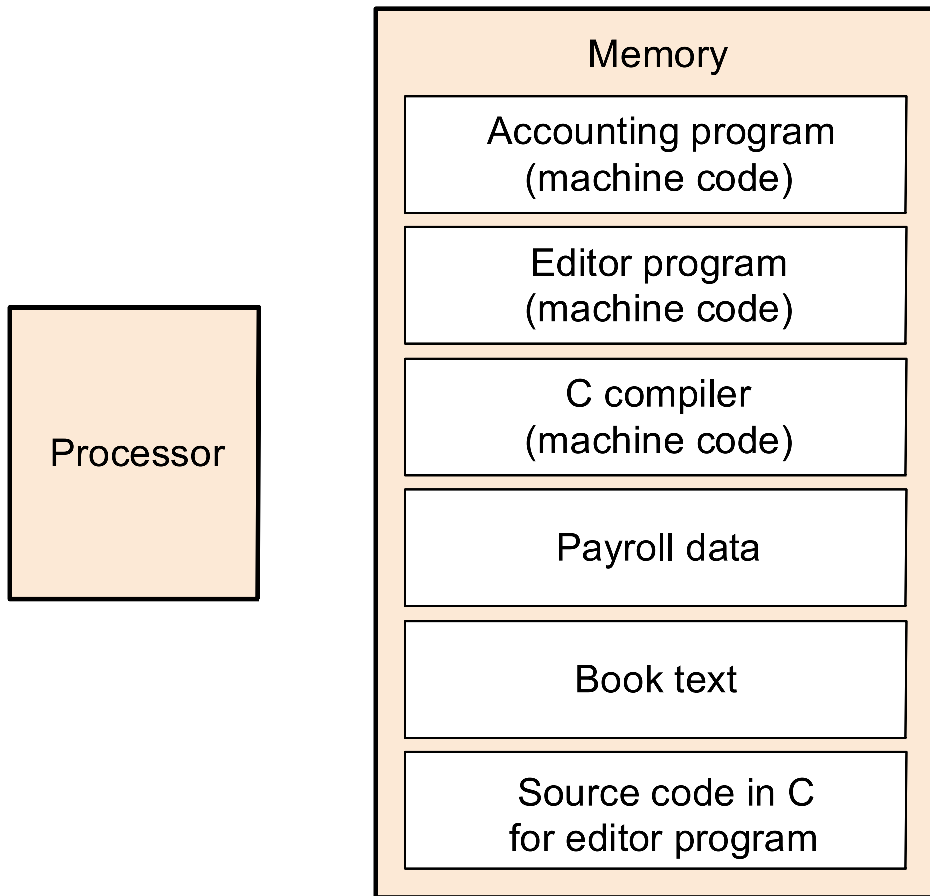
op	rs	rt	rd	Address/ shamt	funct
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		

$t0	8
$t1	9
$s2	18

Two key principles of today's computers

- **Instructions are represented as numbers**
- **Programs can be stored in memory to be read or written just like number**
- **The principles are the stored program concept**

Stored-program concept



Stored programs allow a computer that performs accounting to become, in the blink of an eye, a computer that helps an author write a book. The switch happens simply by loading memory with programs and data and then telling the computer to begin executing at a given location in memory. Treating instructions in the same way as data greatly simplifies both the memory hardware and the software of computer systems. Specifically, the memory technology needed for data can also be used for programs, and programs like compilers, for instance, can translate code written in a notation far more convenient for humans into code that the computer can understand.

2.6 logical operations

Logical operations	C operators	Java operators	MIPS instructions
Shift left	<<	<<	sll
Shift right	>>	>>	srl
Bit by bit AND	&	&	and, andi
Bit by bit OR	|	|	or, ori
Bit by bit NOT	~	~	nor

logical operations --- sll/srl

Shift left logical

e.g. \$s0: 0000 0000 0000 0000 0000 0000 0000 1001$_{two}$=9$_{ten}$

- **sll \$t2, \$s0, 4**

  \$t2: 0000 0000 0000 0000 0000 0000 1001 0000$_{two}$=144$_{ten}$

	op	rs	rt	rd	shamt	funct
Field size(bits)	6	5	5	5	5	6
Value	0	0	16	10	4	0
			\$s0	\$t2		

Logical operations --- and

$t2: 0000 0000 0000 0000 0000 1101 1100 0000

$t1: 0000 0000 0000 0000 0011 1100 0000 0000

and $t0, $t1, $t2

$t0: 0000 0000 0000 0000 0000 1100 0000 0000

or $t0, $t1, $t2

$t0: 0000 0000 0000 0000 0011 1101 1100 0000

Logical operations --- nor

- NOR (Not OR)
  - A NOR 0 = NOT(A OR 0) = NOT(A)
  - nor $t0, $t1, $t3   # reg $t0 = ~(reg $t1 | reg $t3)
- Immediate version
  - andi
  - ori

Instructions for making decisions

- **Conditional branches**
  - **beq (branch if equal)**

    beq register1, register2, L1 # if register1 = register2 GOTO L1
  - **bne (branch if not equal)**

    bne register1, register2, L1 # if register1 ≠ register2 GOTO L1
- **Jump instruction**
  - **j L1**

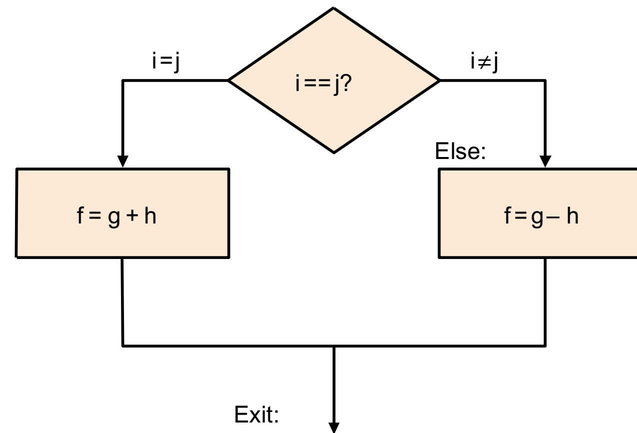
label

- Label
  - L1
  - Else
  - Exit
- Label = memory address
- j Exit
  - goto Exit
  - Move MemoryAddress(exit) to PC
  - EX. Assume: Exit=1000 then PC=1000

Example 2.9 Compiling an *if* statement to a branch

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

- C code:
  - if (i = = j)
    - f = g + h ;
  - else
    - f = g - h ;



- MIPS assembly code:

```
bne    $s3, $s4, Else        # go to else if  i ≠ j
add    $s0, $s1, $s2         #f = g + h  ( skipped if  i ≠  j )
j Exit                       # go to exit
Else: sub  $s0, $s1, $s2     # f  = h - h ( skipped if i = j )
Exit :
```

Compiling a while loop in C

- C code:

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

- Assume: i ~ k ---- $s3 ~ $s5

  base of int array save[] --- $s6

- MIPS assembly code:

```
Loop:  sll    $t1, $s3, 2
       add    $t1,$t1,$s6
       lw     $t0,0($t1)
       bne    $t0,$s5,Exit
       addi   $s3,$s3,1
       j      Loop
 Exit:
```

Basic block

- A sequence of instructions without branches (except possibly at the end) and without branch targets or branch labels (except possibly at the beginning)

Set on less than

- **Signed integer**
  - **slt: set on less than *signed register***
    - **E.g. slt $t0, $s3, $s4 # $t0=1, if $s3<$s4**
 # no change, else
  - **slti: set on less than *signed immediate***
    - **E.g. slti $t0, $s2, -10 # $t0=1, if $s2<-10**
 # no change, else
- **unsigned integer**
  - **sltu: set on less than *unsigned register***
  - **sltiu: set on less than *immediate unsigned***
    - **E.g. sltiu $t0, $s2, 10 # $t0=1, if $s2<10**
 # no change, else

Set on less than

- **Example**
  - **$s0: 1111 1111 1111 1111 1111 1111 1111 1111**
  - **$s1: 0000 0000 0000 0000 0000 0000 0000 0001**
  - **What are the values of registers after these two instructions?**
    - **slt $t0, $s0, $s1**
    - **sltu $t1, $s0, $s1**

	signed	unsigned
$s0	-1	4, 294, 967, 295
$s1	1	1

Bound check shortcut

- Index out of bounds
  - int a[10];
  - a[12]=3;
- If $s1>=$t2, goto the label IndexOutOfBounds
- slt or sltu?
- Answer
  - Assume that $t2 has the value of array length
  - sltu $t0, $s1, $t2   # $t0=0 if $s1>=$t2 or $s1<0

    beq $t0, $zero, IndexOutOfBounds

Case/switch statement

- A trivial solution is to turn the switch statement into a chain of *if-then-else* statements
- Another choice is to encode the alternatives as a table of addresses of alternative instruction sequences (i.e., jump address table or jump table)
  - An array of words containing address that correspond to labels in the code
  - Load the appropriate entry from the jump table to a register
  - Jump register instruction: jr

Supporting Procedures in Computer Hardware

- A procedure (or function) is one tool programmers use to structure programs, both to make them easier to understand and to allow code to be reused. It is one way to implement abstraction in software.
- Six steps for execution of the procedure

  1.Main program to place parameters in 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. Return result value to main program

  6. Return control to the point of origin

- Registers for procedure calling
  - $a0 ~ $a3: four argument registers to pass parameters
  - $v0, v1: return results
  - $ra: one return address register to return to the point of origin

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

  jal   ProcedureAddress

  1). Next instruction address (PC+4) is saved to $ra
  2). Procedure Address is moved to PC

- return control to the caller using
  jr $ra   # return

- **Caller**
- **Callee–Called procedure**
- **Caller-callee**
  - **Caller puts the parameters in $a0-$a3**
  - **jal ProcedureAddress**
  - **Callee (procedure) performs the calculation**
  - **Callee put the results into $v0-$v1**
  - **jr $ra**

Procedure example

- Compiling a C procedure
  - int leaf_example(int g, int h, int i, int j)

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

```
leaf_example:
addi $sp, $sp, -12
sw $t1, 8($sp)
sw $t0, 4($sp)
sw $s0, 0($sp)

add $t0, $a0,$a1
add $t1, $a2, $a3
sub $s0, $t0, $t1
add $v0,$s0, $zero
```

```
lw $s0, 0($sp)
lw $t0, 4($sp)
lw $t1, 8($sp)
addi $sp, $sp, 12
jr $ra
```

Using more registers

- More than 4 arguments
- More than 2 return value

- Spill registers to memory
- Stack
  - ideal data structure for spilling registers
- Stack Operation:
  - Push, pop ;
- Stack pointer: sp (register 29)
- Grow
  - from higher addresses to lower addresses

temporary registers

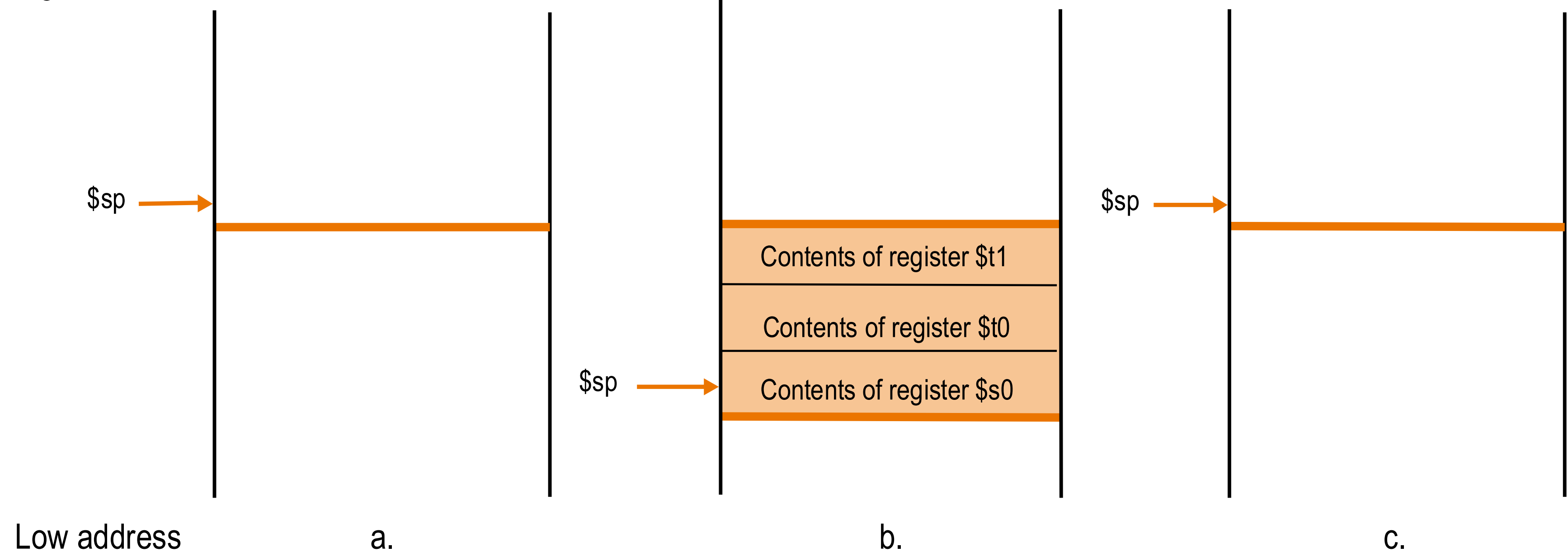
- Two groups of registers
  - $t0-$t9: ten temporary registers , not preserved by the callee
  - $s0-$s9 : 8 saved registers, must be preserved on a procedure call

leaf_example:
addi $sp, $sp, -4
~~sw $t1, 8($sp)~~
~~sw $t0, 4($sp)~~
sw $s0, 0($sp)

add $t0, $a0,$a1
add $t1, $a2, $a3
sub $s0, $t0, $t1
add $v0,$s0, $zero

lw $s0, 0($sp)
~~lw $t0, 4($sp)~~
~~lw $t1, 8($sp)~~
addi $sp, $sp, 4
jr $ra

The values of the stack pointer and stack before, during and after procedure call in the example



Nested Procedures

- Leaf procedure: procedures that do not call others
- Nested Procedure: call other procedures
  - Recursive procedure: invoke "clones" of themselves
    - Example 2.16 Compiling a recursive procedure (Assume: n ---- a0)

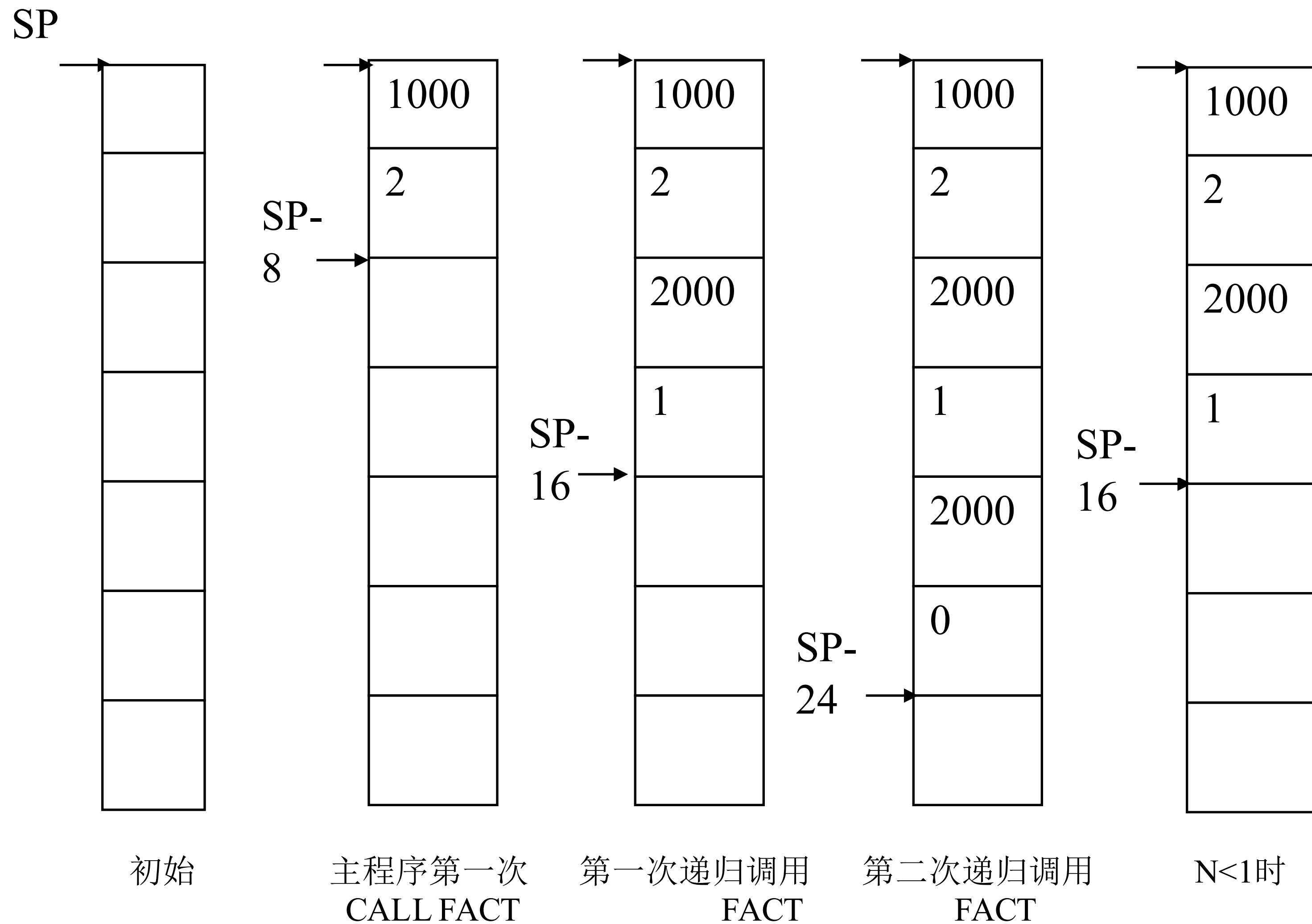
C code:

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

main:

```
  addi  $a0, $zero, 2
  jal  fact
  jr $ra  #return
```

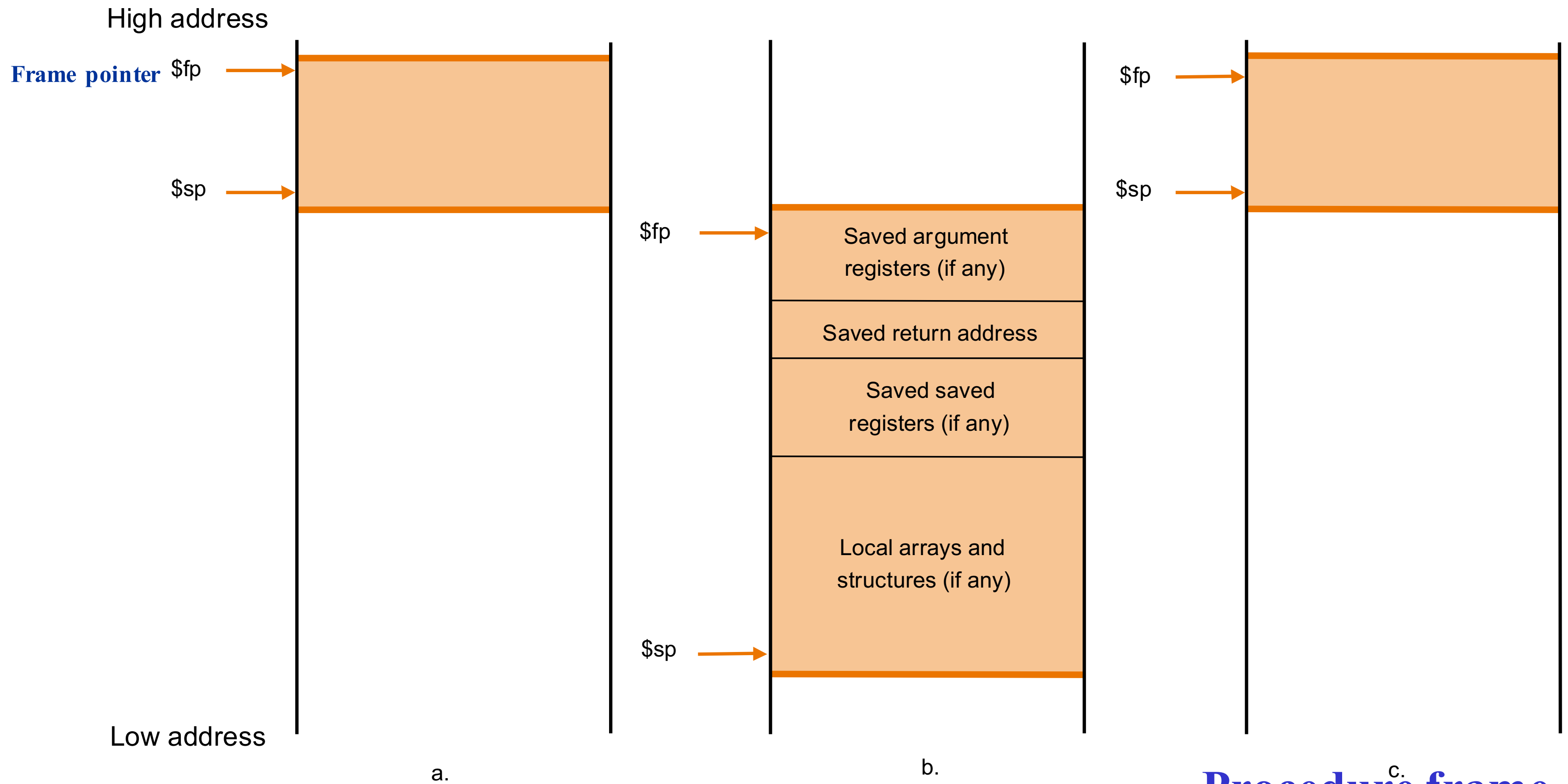
```
fact: addi    $sp, $sp, -8          #adjust stack for 2 items
      sw      $ra, 4($sp)           #save the return address
      sw      $a0, 0($sp)           #save the argument  n
      slti    $t0, $a0, 1           # test for n<1
      beq     $t0, $zero, L1        # if  n  >=  1, go to L1
      addi    $v0, $zero, 1         # return 1
      addi    $sp, $sp, 8
      jr      $ra                   #return to caller
L1:   addi    $a0, $a0, -1          #n>=1, argument gets(n-1)
      jal     fact
      lw      $a0, 0($sp)           #restore from jal: restore argument n
      lw      $ra, 4($sp)           #restore the  return address
      addi    $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
```


SP
SP-
8
1000
2
1000
2
2000
1
SP-
16
1000
2
2000
1
2000
0
SP-
24
1000
2
2000
1
SP-
16
初始
主程序第一次
CALL FACT
第一次递归调用
FACT
第二次递归调用
FACT
N<1时

Allocating space for new data on the stack

- The segment of the stack containing a procedure’s saved registers and local variables is called a procedure frame or activation record
- A frame pointer ($fp) is used to point to the first word of the frame of a procedure
- A frame pointer offers a stable base register within a procedure for local memory-reference.

Stack allocation before, during and after procedure call

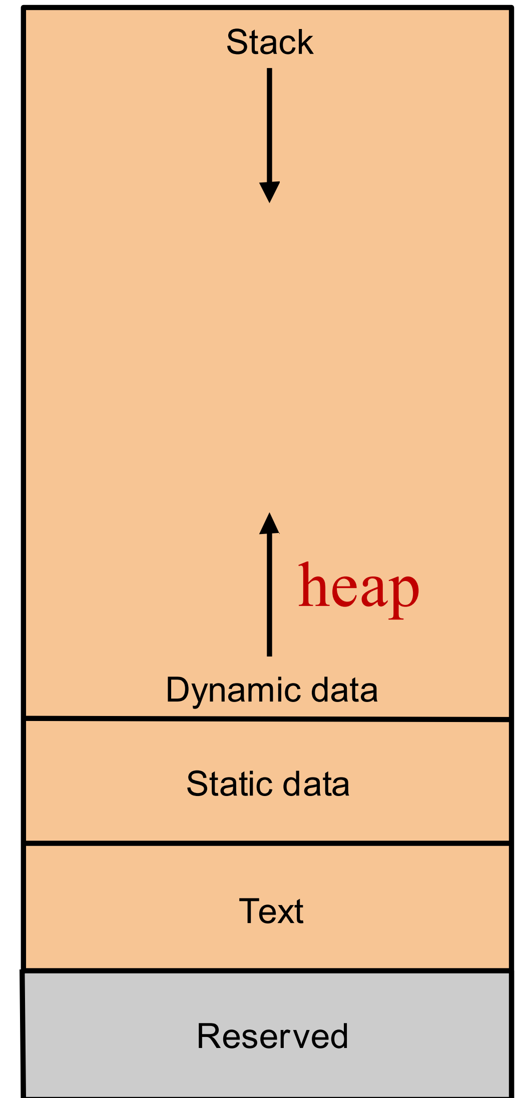


Procedure frame /activation record

MIPS memory allocation for program and data

- **Static data segment**
  - Global variables, static variables, constants
- **Stack**
- **Heap**
  - Data structures tend to grow and shrink during their lifetime
  - Stack and heap grow towards each other
- **Text segment**
  - Program codes

$sp → 7fff ffff hex



$gp → 1000 8000 hex

1000 0000 hex

pc → 0040 0000 hex

0

Segment fault（segfault）

- Running out memeory
- Out of bound of memory
- non-existent address
- a buffer overflow
- using uninitialized pointers
- …

Communicating with people

- Most computer today offer 8-bit bytes to represent characters
- ASCII (American Standard Code for Information Interchanges) is the most commonly used representation

ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter
32	space	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	DEL

FIGURE 2.15 ASCII representation of characters. Note that upper- and lowercase letters differ by exactly 32; this observation can lead to shortcuts in checking or changing upper- and lowercase. Values not shown include formatting characters. For example, 8 represents a backspace, 9 represents a tab character, and 13 a carriage return. Another useful value is 0 for null, the value the programming language C uses to mark the end of a string. This information is also found in Column 3 of the MIPS Reference Data Card at the front of this book.

- **Instructions for moving bytes**
  - **Load byte (lb): load a byte from memory, and place it in the rightmost 8 bits of a register**
    - **lb $t0, 0($sp) # read byte from source**
  - **Store byte (sb): take a byte from the rightmost 8 bits of a register and writes it to memory**
    - **sb $t0, 0($sp) # write byte to destination**
  - **load byte unsigned (lbu)**
    - **lbu $t0, 0($sp) #read byte from source**
  - **Load halfword (lh)**
    - **lh $t0, 0($sp)**
  - **Load halfword unsigned (lhu)**
    - **lhu $t0, 0($sp)**

String copy

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

(Assume: base addresses for x and y ---- r0 and r1 i --- r4)

- **C code:**

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

MIPS assembly code:

```
strcpy: addi  $sp, $sp, -4          # adjust stack
        sw    $s0, 0($sp)           # save $s0
        add   $s0, $zero, $zero     #i  =  0  +  0
L1:     add   $t1, $s0, $a1         # address of y[ i ] in $t1
        lbu   $t2, 0($t1)           #$t2=y [i]
        add   $t3, $s0, $a0         #address of x[ i ] in $t3
        sb    $t2, 0($t3)           # x[ i ]  =  y[ i ]
        beq   $t2, $zero, L2        # if  y[ i ]  0=  0, go to L2
        addi  $s0, $s0, 1           # i  =  i  +  1
        j     L1

L2:     lw    $s0, 0($sp)           # y[ i ]  = =  0; end of string;
                                    #restore old $s0
        addi  $sp, $sp, 4           #pop 1 word off stack
        jr    $ra                   # return
```

Optimization for example 3.17

- strcpy is a leaf procedure
- Allocate i to a temporary register

- For a leaf procedure
  - The compiler exhausts all temporary registers
  - Then use the registers it must save

MIPS Addressing

- **32-bit Immediate operands**
- **Addressing in Branches and Jumps**
- **Showing Branch offset in Machine Language**
- **MIPS addressing mode summary**

Load a 32-bit constant

- When a constant cannot fit the corresponding field (16-bit) in instructions, we have to load it to a register
- Load 32-bit constant to $s0
  - 0000 0000 0011 1101 0000 1001 0000 0000
  - lui $s0,0x003D #$s0=0000 0000 0011 1101 0000 0000 0000 0000
  - ori $s0, $s0, 0x0900 #$s0=0000 0000 0011 1101 0000 1001 0000 0000

- lui $t0, 255

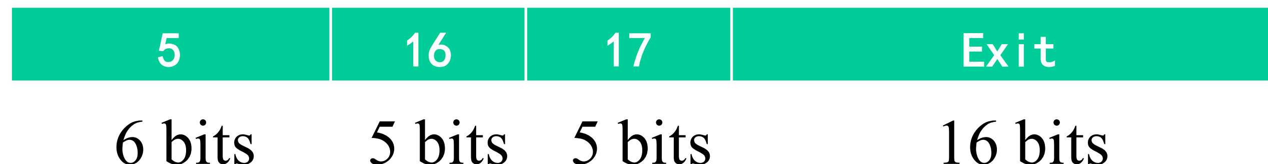
op	rs	rt	constant
001111	0000	01000	0000 0000 1111 1111

Addressing in branches and jumps

- **J-type instruction**
  - **j 10000 # go to location 10000**

6 bits 26 bits

- **Conditional branch instruction**
  - **bne $s0, $s1, Exit # go to Exit if $s0 != $s1**



6 bits 5 bits 5 bits 16 bits

No program could be bigger than 2^{16} !!!

- **Program counter = Register + Branch address**
- **PC-relative addressing**
  - **Almost all conditional branches go to locations less than 2^{16} words**
  - **PC is used as the base register**
  - **Since PC is increased by 4 early to point to the next instruction, MIPS address is actually relative to the address of the following instruction (PC+4) as opposed to the current instruction (PC)**
- **Jump-and-link and Jump instructions**
  - **The invoked procedures are usually far away from the branches**
  - **J-type instruction is employed**

Showing Branch offset in Machine Language

```
Loop: sll   $t1, $s3, 2
      add   $t1, $t1, $s6
      lw    $t0, 0($t1)
      bne   $t0, $s5, Exit
      addi  $s3, $s3,1
      j     Loop
Exit:
```

address	Op	rs	rt			
80000	0	0	19	9	2	0
80004	0	9	22	9	0	32
80008	35	9	8	0		
80012	5	8	21	2 (addr)		
80016	8	19	19	1		
80020	2	20000				
80024	…					

PC-relative addressing:

PC=PC+addr*4 #PC=80016

Branching Far Away

- beq $s0, $s1, L1
- What if the address L1 is far away?
  - bne $s0, $s1, L
  - j L1
  - L2:

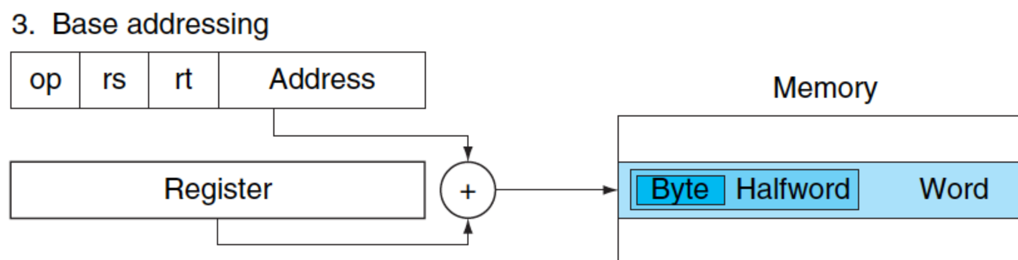
MIPS addressing mode summary

- **Immediate addressing**
  - **The operand is a constant within the instruction**

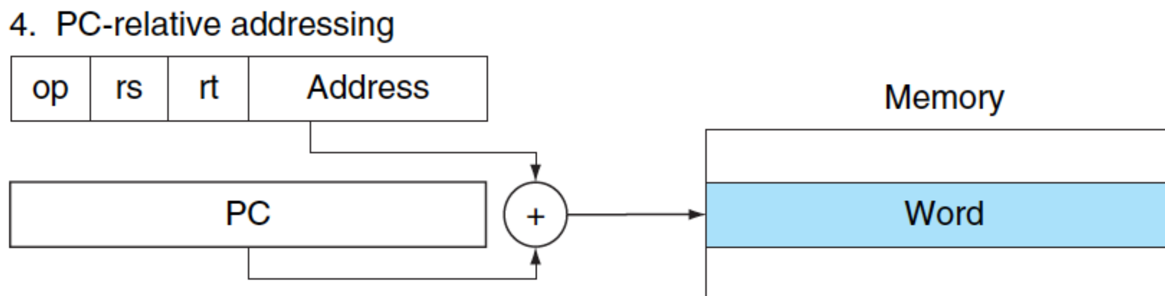
- **Register addressing**
  - **The operand is a register**

MIPS addressing mode summary

- Base or displacement addressing
  - The operand is at the memory location whose address is the sum of a register and a constant in the instruction

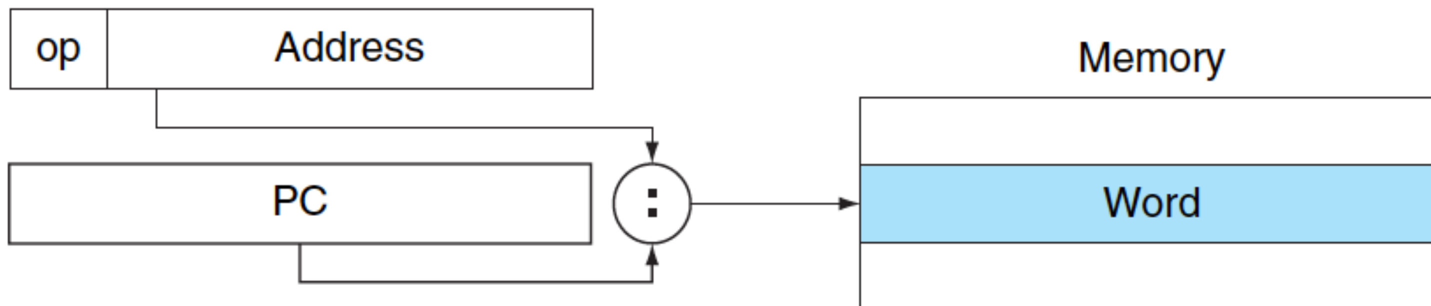


- PC-relative address
  - The branch address is the sum of the PC and a constant in the instruction



MIPS addressing mode summary

- **Pseudodirect addressing**
  - **The jump address is the 26 bits of the instruction concatenated with the upper bits of the PC**



Decoding Machine Code

Machine instruction: 0x00af8020

0000 0000 1010 1111 1000 0000 0010 0000

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

add $s0, $a1, $t7

Parallelism and Instructions Synchronization

- When there exist cooperation between tasks, their parallel execution is difficult. We need to synchronize them to avoid the risk of data race
- Basic operation: lock and unlock
  - Lock and unlock can be used straightforwardly to create regions where only a single processor can operate (so-called "mutual exclusion")

Build a basic synchronization primitive

- Atomic exchange
  - The value 0 indicates that the lock is free, while 1 implies that the lock is unavailable
  - A processor tries to set the lock by doing an exchange of 1, which is in a register, with the memory address corresponding to the lock
- Atomic swap
  - A pair of instructions in which the second one returns a value showing whether the pair of instructions was executed as if the pair were atomic
  - Load linked (ll) and store conditional (sc)

```
try: add $t0,$zero,$s4      ;copy exchange value
     ll  $t1,0($s1)         ;load linked
     sc  $t0,0($s1)         ;store conditional
     beq $t0,$zero,try      ;branch store fails
     add $s4,$zero,$t1      ;put load value in $s4
```

Translating and Starting a Program

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

Assembler

- Assembly language is an interface to high-level software
- Pseudoinstruction
  - A common variation of assembly language instructions often treated as if it was an instruction in its own right
  - move $t0, $t1 # register $t0 gets register $t1
  - add $t0, $zero, $t1 #register $t0 gets 0+register $t1
  - More can be found at https://en.wikibooks.org/wiki/MIPS_Assembly/Pseudoinstructions
- The main task is to translate assembly language into machine code (i.e., an object file)
  - 6 pieces of the object file for Unix systems: object file header, text segment, static data segment, relocation information, symbol table, debugging information

Linker

- Each procedure is compiled and assembled independently, and a linker (or link editor) "stitches" them together.
  - Place code and data modules symbolically in memory
  - Determine the addresses of data and instruction labels
  - Patch both the internal and external references
- The linker produces an executable file that can be run on a computer

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	–	
Object file header			
	Name	Procedure B	
	Text size	200_{hex}	
	Data size	30_{hex}	
Text segment	Address	Instruction	
	0	sw $a1, 0($gp)	
	4	jal 0	
	...	...	
Data segment	0	(Y)	
	...	...	
Relocation information	Address	Instruction type	Dependency
	0	sw	Y
	4	jal	A
Symbol table	Label	Address	
	Y	–	
	A	–	

Executable file header		
	Text size	300_{hex}
	Data size	50_{hex}
Text segment	Address	Instruction
	$0040\ 0000_{hex}$	lw $a0, 8000_{hex}($gp)
	$0040\ 0004_{hex}$	jal $40\ 0100_{hex}$
	...	...
	$0040\ 0100_{hex}$	sw $a1, 8020_{hex}($gp)
	$0040\ 0104_{hex}$	jal $40\ 0000_{hex}$
	...	...
Data segment	Address	
	$1000\ 0000_{hex}$	(X)
	...	...
	$1000\ 0020_{hex}$	(Y)
	...	...

- $1000\ 8000_{hex} + (-0000\ 8000_{hex}) = 1000\ 0000_{hex}$
- $-0000\ 8000hex = -0000\ 0000\ 0000\ 0000\ 1000\ 0000\ 0000\ 0000_{two}$

  $= 1111\ 1111\ 1111\ 1111\ 0111\ 1111\ 1111\ 1111_{two} + 1_{two}$

  $= 1111\ 1111\ 1111\ 1111\ 1000\ 0000\ 0000\ 0000_{two}$

- $1000\ 8000_{hex} + (-0000\ 7fe0_{hex}) = 1000\ 0020_{hex}$
- $-0000\ 7fe0_{hex} = -0000\ 0000\ 0000\ 0000\ 0111\ 1111\ 1110\ 0000_{two}$

  $= 1111\ 1111\ 1111\ 1111\ 1000\ 0000\ 0001\ 1111_{two} + 1_{two}$

  $= 1111\ 1111\ 1111\ 1111\ 1000\ 0000\ 0010\ 0000_{two}$

Loader

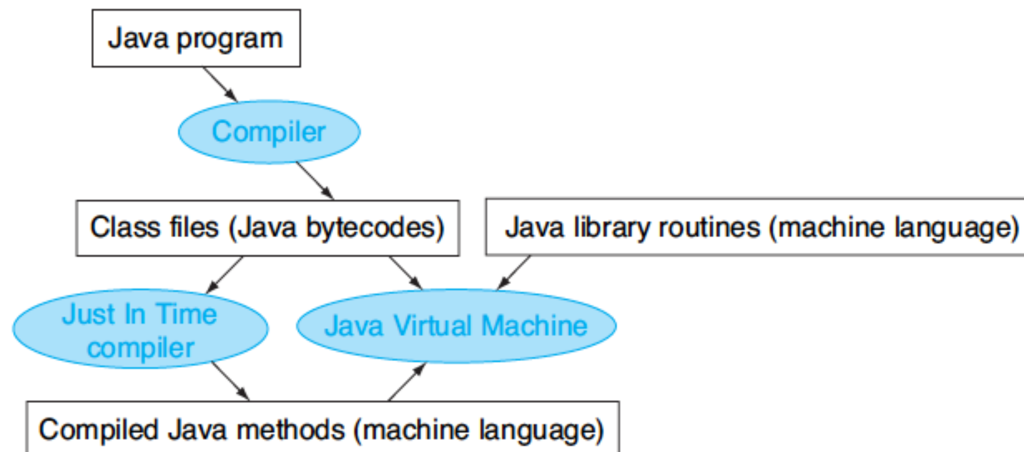
- Loader is the part of an operating system that is responsible for loading programs, one of the essential stages in the process of starting a program
- It means loader is a program that places programs into memory and prepares them for execution.
- Loading
  - Determine size of text and data segments
  - 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

Dynamically linked libraries

- **Linking Libraries before the program is run**
  - **The library routines become part of the executable code.**
  - **It loads all routines in the library that are called anywhere in the executable, even if those calls are not executed**
- **Dynamically linked libraries (DLLs)**
  - **Library routines that are linked to a program during execution**
  - **DLLs require extra space for the information needed for dynamic linking, but do not require that whole libraries be copied or linked**

Starting a java program

- A java program is first compiled to **Java bytecode** instructions, which are very close to java language such that the compilation is trivial
- A software interpreter, so-called **Java Virtual Machine (JVM)**, can execute Java bytecodes. JVM also links to desired routines in Java libraries while the program is running
  - High portability but low performance
- To preserve portability and improve execution speed, **Just In Time compilers (JIT)** is designed for optimization purpose
  - They profile the running program to find the “hot” method and then compile them into the native instruction.
  - The compiled portion is saved for the next time the program is run.



A C Sort Example To Put it All Together

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

```
void swap ( 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**

Code for the procedure swap

Procedure body

```
swap: sll   $t1, $a1, 2          # reg $t1 = k * 4
      add   $t1, $a0, $t1        # reg $t1 = v + (k * 4)
                                 # reg $t1 has the address of v[k]
      lw    $t0, 0($t1)          # reg $t0 (temp) = v[k]
      lw    $t2, 4($t1)          # reg $t2 = v[k + 1]
                                 # refers to next element of v
      sw    $t2, 0($t1)          # v[k] = reg $t2
      sw    $t0, 4($t1)          # v[k+1] = reg $t0 (temp)
```

Procedure return

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

Arrays versus Pointers

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

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

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

```
       move  $t0,$a0         # p = & array[0]
       sll   $t1,$a1,2       # $t1 = size * 4
       add   $t2,$a0,$t1     # $t2 = &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 () go to loop2
```

Summary

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

Fallacies and pitfalls

- Fallacies
  - More powerful instructions mean higher performance
  - Write in assembly language to obtain the highest performance
  - The importance of commercial binary compatibility means successful instruction sets don’t change
- Pitfalls
  - Forgetting that sequential word addresses in machines with byte addressing do not differ by one
  - Using a pointer to an automatic variable outside its defining procedure

Further reading

- 2.13 Sort algorithm in MIPS
- 2.15 Compiling C and interpreting Java
- 2.16 Real Stuff: ARM instructions
- 2.17 Real Stuff: x86 instructions

END