CHAPTER

3

MIPS – 32 BITS

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHTN
TP. HO CHI MINH

cdio
4.0

KHOA CÔNG NGHỆ THÔNG TIN
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

fit@hcmus

REMIND

- ISA
- RISC vs CISC

PREREQUITES

- Install MARS software already

fit@hcmus

What will you learn?

- Basic of a MIPS program
- Registers
- Memory Organization
- Operations
- System calls
- Procedures

MIPS Instruction Set

- Large share of embedded core market

  Applications in consumer electronics, network/storage equipment, cameras, printers, …

- Typical of many modern ISAs

MIPS Principles

According to the RISC architecture with 4 principles:

- Simpler is more stable
- Smaller is faster
- Increase processing speed for frequent cases
- Design requires good compromise

MIPS Assembly Program

```
        .data           # data label declaration
label1:   <data type> <initial value>
label2:   <data type> <initial value>
…
        .text           # instructions follow this directive
        .globl  #global text label, can access from another file
        .globl  main        # This is the required global text label of
   the program
        …
main:                   # indicated start of code
…
                        #end of program
```

How to use MARS to code an assembly program

- Introduce how to use MARS tool
- Create a basic assembly program
- Observe the changing in the registers set and memory

Policy of use conventions for registers

Name	Register Number	Usage
$zero	0	The constant value 0
$v0	2	Result from callee
$v1	3	Return to caller
$a0	4	Argument to callee
$a1-$a3	5-7	From caller: caller saves
$t0-$t7	8-15	Temporaries: callee can clobber
$s0-$s7	16-23	Saved: callee can clobber
$t8-$t9	24-25	More temporaries (conditions)
$gp	28	Global pointer
$sp	29	Stack pointer
$fp	30	Frame pointer
$ra	31	Return address (caller saves)
$at	1	Reserved for assembler
$k0-$k1	26-27	Reserved for the OS kernel

Memory Organization

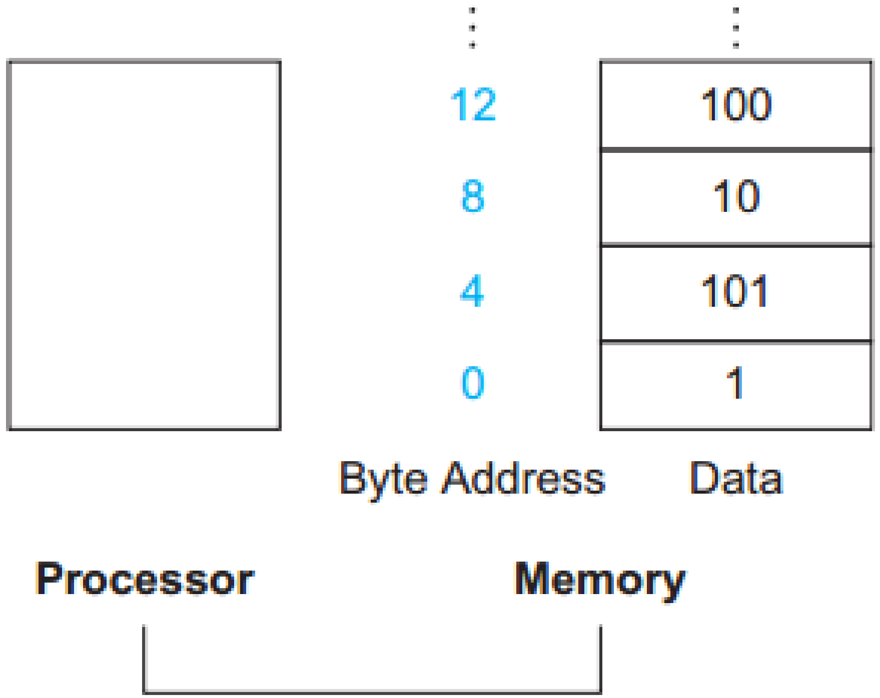
- Address, index
- Byte addressing
- Little/ Big Endian byte order
- MIPS addressing mode

Address/ Index

- The **address** is a value used to specify the location of a specific data element within a memory array
- The address of a word matches the address of one of the 4 bytes within the word and addresses of sequential words differ by 4

Byte addressing

- **Byte addressing** is the index points to a byte of memory
- MIPS uses byte addressing, with word addresses are multiples of 4
- To access a word in memory, the instruction must supply the memory address



Chap2, A. Patterson and J. L. Hennessy, ***Computer Organization and Design: The Hardware/Software Interface,5th ed, 2014*** Figure 2.3

Actual MIPS memory addresses and contents of memory for those words

Little/ Big Endian byte order

Bytes in a word can be numbered in two ways:

- Byte 0 at the leftmost (most significant) to the rightmost (least significant), called *big-endian* *(MIPS uses big-endian byte order)*
- The leftmost (most significant) to byte 0 at the rightmost (least significant), called *little-endian*

Big-endian (Bit 31 … Bit 0)

Byte 0	Byte 1	Byte 2	Byte 3
00	00	07	E4

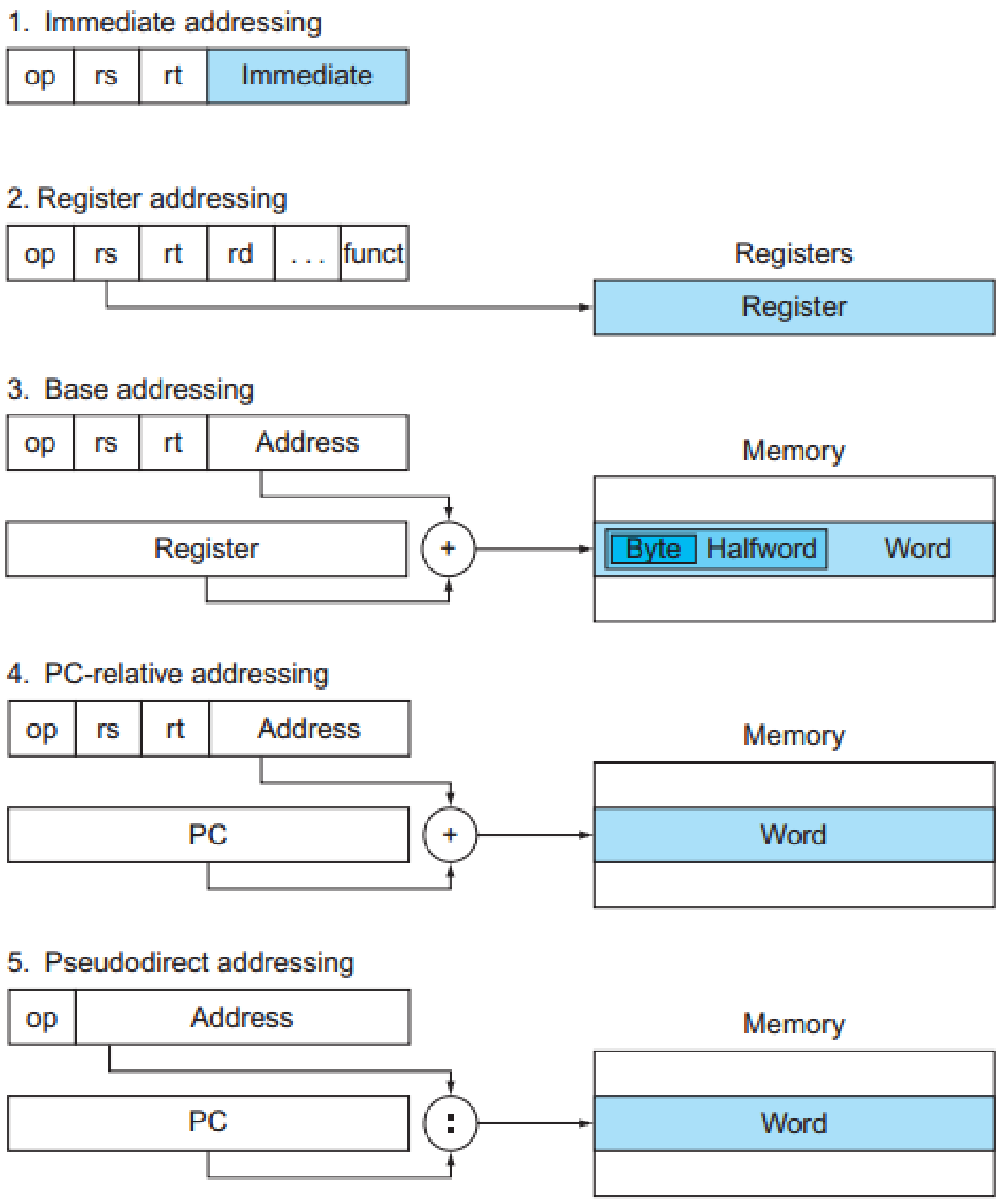
Little-endian (Bit 31 … Bit 0)

Byte 3	Byte 2	Byte 1	Byte 0
E4	07	00	00

MIPS addressing mode

MISPS supports the following addressing mode:

- Immediate addressing
- Register addressing
- Memory addressing
  - Base addressing
  - PC-relative addressing
  - Pseudo-direct addressing



fit@hcmus

MIPS Operations

- Arithmetic
- Logical
- Data transfer
- Branch (Unconditional/ conditional)

Arithmetic operations

Syntax (R-format): `op rd, rs, rt`

op: operation code

rd: destination register number

rs: the first source register number

rt: the second source register number/ an immediate

Arithmetic operations

C code:

A = B + C

MIPS code

```
add $s0, $s1, $s2
```

Compiler will associate the variables to the registers

Discussions

1. How to know if an operation compiled from C is a signed operation or not?

→ Compiler

2. Can an operand be used as both a source and a target?

→ Yes

3. Can constant data specify in an instruction?

→ Yes (use I-format)

addi $s0, $s1, 3 (addi = add immediate)
addi $s0, $s1, -3 (do not have subi)

Example: compute multiple operands

C code:

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

MIPS code: f→$s0, h →$s1 , i →$s2 , j →$s3

```
add    $t0, $s1, $s2    #  temp1 = g + h
add    $t1, $s3, $s4    #  temp2 = i + j
sub    $s0, $t0, $t1    #  f = temp1 - temp2
```

Example: assignment/ move between registers

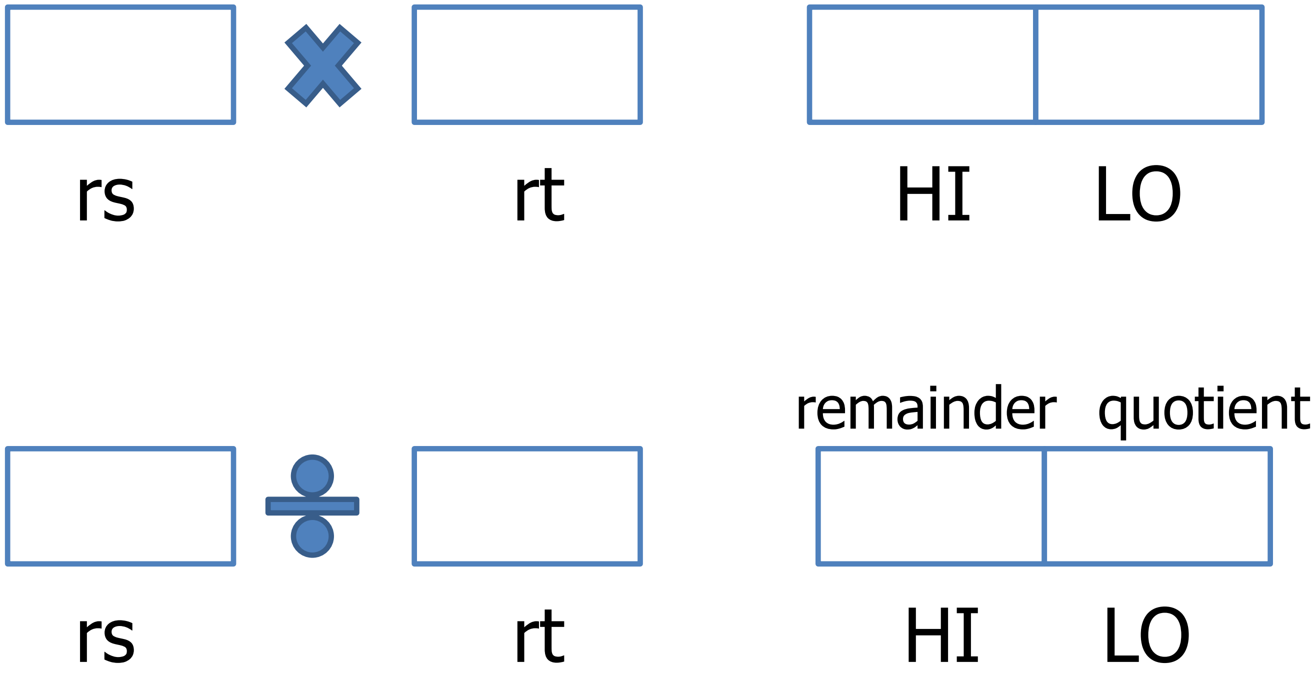
C code:

```
a = b
```

MIPS code:

```
add     $t1, $t0, $zero # a → $t1, b → $t0
        #$zero contains the constant value 0
```

Arithmetic operations: Multiply & Division



- Two 32-bit registers for product
  - HI: most-significant 32 bits
  - LO: least-significant 32-bits
- Use HI/LO registers for result
  - HI: 32-bit remainder
  - LO: 32-bit quotient
- Use mfhi, mflo instructions to access the result

Arithmetic operations: Multiplication

Syntax 1: `mult rs, rt / multu rs, rt`

Get the result:

```
mfhi rd     #test HI value to see if product overflow 32 bits

mflo rd
```

Syntax 2: `mul rd, rs, rt`

```
# Least significant 32 bits of product → rd
```

Arithmetic operations: Division

Syntax : `div rs, rt` / `divu rs, rt`

Get the result:

```
mfhi rd      #HI contains 32 bits of remainder
mflo rd      #LO contains 32 bits of quotient
```

No overflow or divide by 0 checking (software's job)

Arithmetic operations: Floating-point numbers

- Coprocessor1: the adjunct processor that extends the ISA
- Separate FP (Floating-point) registers:
  - Single-precision: \$f0 - \$31 (32 registers)
  - Use the paired registers to save double precision floating-point number
- FP instructions only operate on FP registers

Example: floating-point operands

Single-precision:

```
add.s    $f0, $f1, $f2
```

Double-precision:

```
add.d    $f0, $f1, $f2
```

Logical Operations

Instructions for bitwise manipulation

Operation	C	Java	MIPS
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bitwise AND	&	&	and, andi
Bitwise OR	|	|	or, ori
Bitwise NOT	~	~	nor

Useful for extracting and inserting groups of bits in a word

Logical Operations: Shift operations

Syntax (R-format): `op rd, rs, shamt`

op: operation code

rd: destination register number

rs: the source register number

shamt: How many positions to shift (< 32 bits)

Logical Operations: Shift operations

Shift left logical: shift left and fill with 0 bits

```
sll    $s0, $s1, 2    # $s1 shl 2 → $s0
```

Shift right logical: shift right and fill with 0 bits

```
srl    $s0, $s1, 2    # $s1 shr 2 → $s0
```

Shift right arithmetic: shift right and fill with bits which has a value equal to the MSB bit

```
sra    $s0, $s1, 2    # $s1 sar 2 → $s0
```

Logical Operations: Logical math operations

Syntax (R-format): `op rd, rs, rt`

op: operation code

rd: destination register number

rs: the first source register number

rt: the second source register/ immediate

Logical Operations: Logical math operations

MIPS does not support instructions for NOT, XOR, NAND

```
                              #a→$s1, b →$s2
and     $s0, $s1, $s2   # a AND b
ori     $s0, $s1, 2   # a OR i (a constant
number)
nor     $t0, $t1, $zero #NOT A = A NOR 0
```

Data transfer

- RAM access only allow load/ store instructions
- Load word has destination first, store has destination last

Data transfer operations

Syntax (I-format): `op rt,(constant/address)rs`

op: operation code

rs: base address

rt: destination/ source register number

Constant: $-2^{15} \rightarrow 2^{15} - 1$

Address: offset added to base address in rs (offset is always a multiple of 4)

Data transfer operations

```
lw    rt, offset(rs)   # load a word at the
address                $s0 + 12 from the memory
sw    rt, offset(rs)   # store a word to the
memory at              the location $s0 + 12
```

Example

- Suppose that A is an array of 100 words with the starting address (base address) contained in register $s0.
- The value of the variable g are stored in registers $s1 and $s2 respectively

C code: g = A[2];

MIPS code:

```
lw  $t0, 8($s0)   #load A[2] to the register $t0
add $s1, $t0,$zero   #save the value of A[2] to g
```

Example

C code: `A[12] = h + A[8];`

MIPS code:

```
lw  $t0, 32($s0) #load A[8] to the register $t0
add $t0, $s2,$t0 #temporary register $t0=h + A[8]
sw  $t0, 48 ($s0)#store h + A[8] back to A[12]
```

Discussions

- ☐ Load a byte xzzz zzzz from memory to a register in CPU. (Useful for ASCII)

```
lb $t0, g
```

What is the value of this register (x is the MSB of the byte) ?

→ Sing-extended: xxxx xxxx xxxx xxxx xxxx xxxx xzzz zzzz

- ☐ If you want the remaining bits from the right to have a zero value (unsigned number)

→ Load byte unsigned: `lbu $t0, g`

Discussions

Load/ store a half of word (useful for Unicode)

→ Load halft: `lh $t0, g` #load ½ word – the 2 right most bytes

→ Store half: `sh $t0, g` #store ½ word – the 2 right most bytes

Branch operations:

- Two kinds of branch instructions:
  - Conditional
  - Unconditional
- MIPS branch destination address = PC + (4 * offset)
  - Target address = PC + offset × 4
  - PC already plus 4 bytes by this time

Branch operations: Conditional branch

Syntax (I-format): oprs , rt , target address

rs: the first source register number

rt: the second source register number

Target Address: the address of the next instruction

```
beq    rs, rt, label        #if (opr1 == opr2) goto label

bne    rs, rt, label        #if (opr1 != opr2) goto label
```

Example

C Code:

```
if(a==b)
    i = 1;
else
    i = 0;
```

MIPS code:

```
#a → $t0, b → $t1, i → $t2
beq   $t0, $t1, Label
addi $t2, $zero, 0
Label
addi $t2, $zero, 1
```

Branch operations: Unconditional branch

Syntax (J-format): opTarget Address

rs: the first source register number

rt: the second source register number

Target Address: the address of the next instruction

```
j    label                    #goto label
```

Example

C Code:

```
Do{
      Task 1;
}while(a!=0)
Task 2
```

MIPS code:

```
#a → $t0
Loop:
Task 1
bne   $t0, $zero, Loop
j Task 2
```

Discussion

- Bigger/ smaller comparison ?

→ MIPS support slt instruction to make the solutions for not equal comparison

Syntax:

```
slt rd, rs, rt
```

Explain:

```
if (rs < rt)
  rd = 1
else
  rd = 0
```

fit@hcmus

Example

```
a < b
slt     $t0, $s0, $s1          # if (a < b) then $t0 = 1
bne   $t0, $0, Label           # if (a < b) then goto Label
Another Task                   # else then do something
Label
a > b
slt     $t0, $s1, $s0          # if (b < a) then $t0 = 1
bne   $t0, $0, Label           # if (b < a) then goto Label
Another Task                   # else then do something
Label
Another Task                   # else then do something
```

Example

a ≥ b

```
slt     $t0, $s0, $s1       # if (a < b) then $t0 = 1
beq   $t0, $0, Label        # if (a ≥ b) then goto Label
Another Task                # else then do something
Label
```

a ≤ b

```
slt     $t0, $s1, $s0       # if (b < a) then $t0 = 1
beq   $t0, $0, Label        # if (b ≥ a) then goto Label
Another Task                # else then do something
Label
```

Discussion

Compare with constant?

→ MIPS support slti instruction to make the solutions for not equal comparison

Syntax:

```
slti rd, rs, constant
```

Explain:

```
if (rs < constant)
  rd = 1
else
  rd = 0
```

Example: Switch...case in C

C code:

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

C code: Convert to if...else

```
if (k == 0)
    f = i + j;
else if (k == 1)
    f = g + h;
else if (k == 2)
    f = g - h;
```

f	g	h	i	j	k
\$s0	\$s1	\$s2	\$s3	\$s4	\$s5

Example: Switch..case in MIPS

```
bne $s5, $0, L1             # if (k != 0) then goto L1
Add $s0, $s3, $s4           # else (k == 0) then f = i + j
J Exit                      # end of case → Exit (break)
L1:
addi $t0, $s5, -1           # $t0 = k - 1
bne $t0, $0, L2             # if (k != 1) then goto L2
add $s0, $s1, $s2           # else (k == 1) then f = g+ h
J Exit                      # end of case → Exit (break)
L2:
addi $t0, $s5, -2           # $t0 = k - 2
bne $t0, $0, Exit           # if (k != 2) then goto Exit
sub $s0, $s1, $s2           # else (k == 2) then f = g - h
Exit:
```

cdio 4.0
fit@hcmus

System calls

- Through the system call (syscall) instruction to request a service from a small set of the operating system services
- System call code → $v0
- Arguments → $a0 - $a3 ($f12 for floating-point)
- Return value → $v0 ($f0 for floating-point)

fit@hcmus

System calls

Service	System call code	Arguments	Result
print_int	1	$a0 = integer	
print_float	2	$f12 = float	
print_double	3	$f12 = double	
print_string	4	$a0 = string	
read_int	5		integer (in $v0)
read_float	6		float (in $f0)
read_double	7		double (in $f0)
read_string	8	$a0 = buffer, $a1 = length	
sbrk	9	$a0 = amount	address (in $v0)
exit	10		
print_char	11	$a0 = char	
read_char	12		char (in $v0)
open	13	$a0 = filename (string), $a1 = flags, $a2 = mode	file descriptor (in $a0)
read	14	$a0 = file descriptor, $a1 = buffer, $a2 = length	num chars read (in $a0)
write	15	$a0 = file descriptor, $a1 = buffer, $a2 = length	num chars written (in $a0)
close	16	$a0 = file descriptor	
exit2	17	$a0 = result	

fit@hcmus

Example

```
addi   $v0, $0, 4  # $v0 = 0 + 4 = 4
                   # print string syscall
la  $a0, str       # $a0 = address(str)
syscall            # execute the system call
```

cdio