Computer organization

Lab2 Assembly language-MIPS(1)

data, load & store, arithmetic Instructions

2021 Spring term

wangw6@sustech.edu.cn

Machine Language & Assembly Language

- **Machine instruction** : a binary representation used for communication with a computer system.
- **Assembly instruction** : a symbolic representation of machine language
- **Assembler**: translate assembly codes into binary instructions

Data | Text

Text

```
                              User Text Segment [00400000]..[00440000]
[00400000] 8fa40000  lw $4, 0($29)             ; 183: lw $a0 0($sp) # argc
[00400004] 27a50004  addiu $5, $29, 4          ; 184: addiu $a1 $sp 4 # argv
[00400008] 24a60004  addiu $6, $5, 4           ; 185: addiu $a2 $a1 4 # envp
[0040000c] 00041080  sll $2, $4, 2             ; 186: sll $v0 $a0 2
[00400010] 00c23021  addu $6, $6, $2           ; 187: addu $a2 $a2 $v0
[00400014] 0c100009  jal 0x00400024 [main]     ; 188: jal main
```

Assembly Language based on MIPS

Data declaration

- Data declaration section starts with "**. data**".
- The declaration means a piece of memory is required to be allocated. The declaration usually includes lable (name of address on this meomory unit), size(optional), and initial value(optional).

Code definition

- Code definition starts with "**.text**", includes basic instructions, extended instructions, labels of the code(optional).
- At the end of the code, "exit" system service should be called.

comments

- Comments start from "**#**" till the end of current line

Data Types and Literals

In **MIPS32**

- Unit Conversion
  - **1 word = 32bit = 2*half word(2*16bit) = 4* byte(4*8bit)**
- Data Storage:
  - Instructions are **all 32 bits**(1 word)
  - A character requires **1 byte** of storage
- Literals :
  - Characters enclosed in single quotes. e.g. ‘C’
  - Strings enclosed in double quotes. E.g. “a String”
  - Numbers in code. e.g. 10

Q: what’s the size of Register in MIPS32?

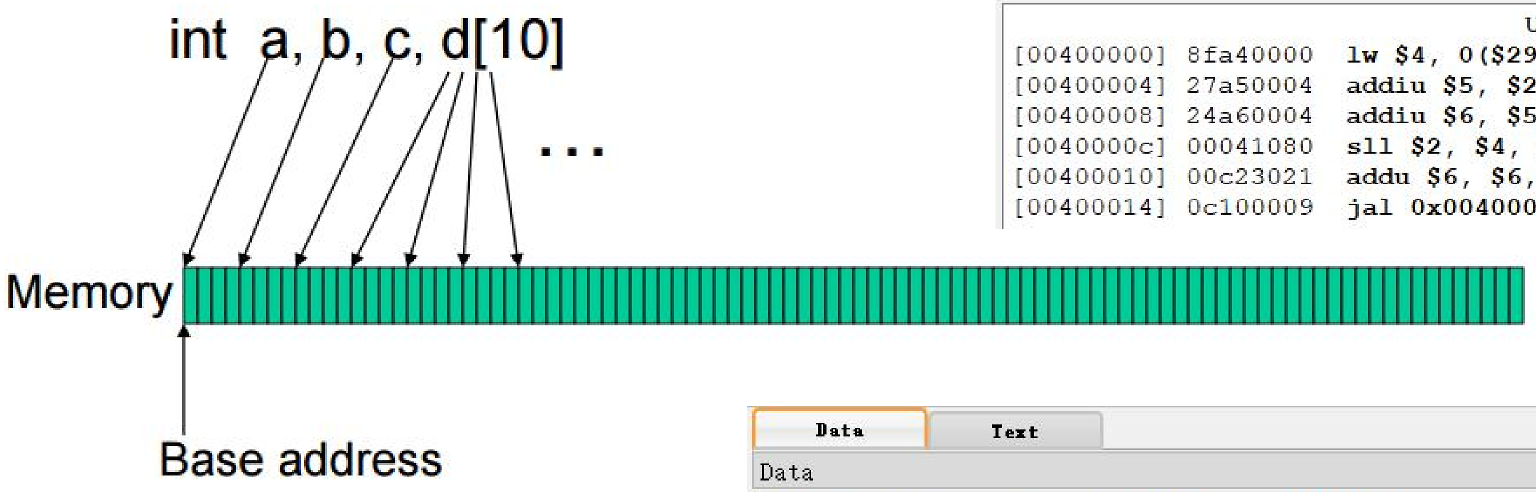
Data Declaration

```
name:        storage_type    value(s)
```

```
example

var1:       .word   3       # create a single integer:
                            #variable with initial value 3

array1:     .byte   'a','b' # create a 2-element character
                            # array with elements initialized:
                            # to  a  and  b

array2:     .space  40      # allocate 40 consecutive bytes,
                            # with storage uninitialized
                            # could be used as a 40-element
                            # character array, or a
                            # 10-element integer array;
                            # a comment should indicate it.

string1:    .asciiz "Print this.\n"        #declare a string
```

Memory

- The compiler organizes data in memory… it knows the location of every variable (saved in a table)… it can fill in the appropriate mem-address for load-store instructions




```
Data    Text
Text
                                   User Text Segment [00400000]..[00440000]
[00400000] 8fa40000  lw $4, 0($29)              ; 183: lw $a0 0($sp) # argc
[00400004] 27a50004  addiu $5, $29, 4           ; 184: addiu $a1 $sp 4 # argv
[00400008] 24a60004  addiu $6, $5, 4            ; 185: addiu $a2 $a1 4 # envp
[0040000c] 00041080  sll $2, $4, 2              ; 186: sll $v0 $a0 2
[00400010] 00c23021  addu $6, $6, $2            ; 187: addu $a2 $a2 $v0
[00400014] 0c100009  jal 0x00400024 [main]      ; 188: jal main
```

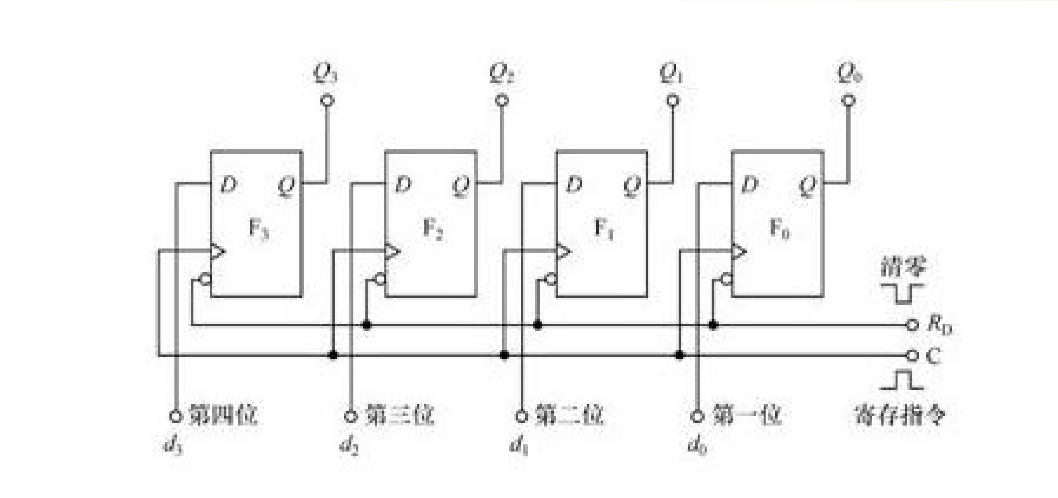
```
Data    Text
Data
User data segment [10000000]..[10040000]
[10000000]..[1000ffff]  00000000
[10010000]    6c6c6568  6e49206f  6e726574  000a7465    h e l l o   I n t e r n e t . .
[10010010]..[1003ffff]  00000000
```

Register (1)

- **Registers** are small storage areas used to store data in the CPU, which are used to temporarily store the data and results involved in the operation.
- All MIPS arithmetic instructions must operate on registers
- The **size of registers** in MIPS32 is **32 bits**

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0x10010000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x10010000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000063
$t1	9	0x00000000



Register (2)

REGISTER NAME, NUMBER, USE, CALL CONVENTION

NAME	NUMBER	USE	PRESERVED ACROSS A CALL?
$zero	0	The Constant Value 0	N.A.
$at	1	Assembler Temporary	No
$v0-$v1	2-3	Values for Function Results and Expression Evaluation	No
$a0-$a3	4-7	Arguments	No
$t0-$t7	8-15	Temporaries	No
$s0-$s7	16-23	Saved Temporaries	Yes
$t8-$t9	24-25	Temporaries	No
$k0-$k1	26-27	Reserved for OS Kernel	No
$gp	28	Global Pointer	Yes
$sp	29	Stack Pointer	Yes
$fp	30	Frame Pointer	Yes
$ra	31	Return Address	No

Load & Store

- In **MIPS**
  - access the **data in memory** could **only** be invoked by two types of instruction: **load** and **store**.
  - All the calculation are based on the data in **Registers**.

Name	Example	Comments
32 registers	`$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic, register `$zero` always equals 0, and register `$at` is reserved by the assembler to handle large constants.
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.

Load (Load to Register)

```
lw          register_destination, RAM_source
                                   # copy word (4 bytes) at
                                   # source_RAM location
                                   # to destination register.
                                   # load word -> lw

lb          register_destination, RAM_source
                                   # copy byte at source RAM
                                   # location to low-order byte of
                                    # destination register,
                                    # and sign -e.g. tend to
                                    # higher-order bytes
                                    # load byte -> lb

 li          register_destination, value
                                    #load immediate value into
                                    #destination register
                                    #load immediate --> li
```

Store (Store to Memory)

```
sw          register_source, RAM_destination
                                     #store word in source register
                                     # into RAM destination

sb          register_source, RAM_destination
                                     #store byte (low-order) in
                                     #source register into RAM
                                     #destination
```

Addressing on the Memory

- **Direct addressing**: such as “la”(load address), load the address into the register.

- **Indirect addressing**: using the content in register as address.

- **Baseline/ index addressing**: using the sum of baseline address and offset as address.

Direct Addressing

```
la          $t0, var1
```

- **Load the address which is labeled by "var1" into the Register "t0" .**
  - NOTIC : "var1" could be either a label of data or a MIPS instruction.

print string	4	$a0 = address of null-terminated string to print

```
.data
    str:     .asciiz "the answer = "
.text
        li $v0, 4                 # system call code for print_str
        la $a0, str               # address of string to print
        syscall                   # print the string
```

Indirect Addressing

```
lw          $t2, ($t0)
```

- **Load the word** from the memory unit whose address is in the register "t0" to the register "t2".

```
sw          $t2, ($t0)
```

- **Store the word** in register "t2" to the memory unit whose address is in the register "t0".

Baseline / Index Addressing:

```
lw          $t2, 4($t0)
```

- **Load the word** from the memory unit whose address is in the sum of the value in register "t0" and 4 to the register "t2".

```
sw          $t2, -12($t0)
```

- **Store the word** in register "t2" to the memory unit whose address is the sum of the value in the register "t0" and -12.

- Baselin/Index addressing is applied widely while process on the Array and stack.

Demo1

Edit Execute

lab2_ls.asm

```
.data
        str: .asciiz "A"
.text
main:
        la $a0,str
        li $v0,4
        syscall
        lb $t0,($a0)
        addi $t0,$t0,32
        sb $t0,str
        syscall
        li $v0,10
        syscall
```

Q: what's the output of the code?

```
.data
        arrayx: .word 11
.text
main:
        lw $a0,arrayx
        li $v0,1
        syscall

        li,$v0,10
        syscall
```

```
.data
        arrayx: .word 11
.text
main:
        la $t0,arrayx
        move $a0,$t0
        li $v0,1
        syscall

        li,$v0,10
        syscall
```

Arithmetic Instructions(1)

```
add     $t0,$t1,$t2    #  $t0 = $t1 + $t2;   add as signed
                       # (2's complement) integers
sub     $t2,$t3,$t4    #  $t2 = $t3 Ð $t4
addi    $t2,$t3, 5     #  $t2 = $t3 + 5;   "add immediate"
                       # (no sub immediate)
addu    $t1,$t6,$t7    #  $t1 = $t6 + $t7;
addu    $t1,$t6,5      #  $t1 = $t6 + 5;
                       #  add as unsigned integers
subu    $t1,$t6,$t7    #  $t1 = $t6 - $t7;
subu    $t1,$t6,5      #  $t1 = $t6 - 5
                       #  subtract as unsigned integers
```

Arithmetic Instructions(2)

```
mult    $t3,$t4     # multiply 32-bit quantities in $t3
                    # and $t4, and store 64-bit
                    # result in special registers Lo
                    # and Hi:  (Hi,Lo) = $t3 * $t4

div     $t5,$t6     # Lo = $t5 / $t6   (integer quotient)
                    # Hi = $t5 mod $t6   (remainder)
mfhi    $t0         # move quantity in special register Hi
                    # to $t0:   $t0 = Hi
mflo    $t1         # move quantity in special register Lo
                    # to $t1:   $t1 = Lo,  used to get at
                    # result of product or quotient
```

Demo2

```
.data
        str1: .asciiz "13/4  quotient is: "
        str2: .asciiz " , reminder is : "
.text
main:
        la $a0,str1
        li $v0,4
        syscall

        li $t0,13
        li $t1,4
        divu $t0,$t1
        mflo $a0
        li $v0,1
        syscall

        la $a0,str2
        li $v0,4
        syscall

        mfhi $a0
        li $v0,1
        syscall

        li $v0,10
        syscall
```

Q1:What's processing of "mflo" and "mfhi" ?

Q2: What's the usage of registers "hi" and "lo"?

Q3: What's the value of "str2" ?

Q4: Is there any instruction doing "move to" ?

Q5: What's the output of this demo?

Demo 3

answer the questions in the comments of following code

```
.data
	name:		.space  16		#malloc 16 byte , not initialize   #####  name value : 0x10010000
	mick:		.ascii "mick\n"		# malloc 4+1 = 5byte = 5 * asciic(byte)
	alice:		.asciiz "alice\n"	#####  what's the value of alice ?
	tony:		.asciiz "tony\n"	#####  what's the value of tony ?
	chen:		.asciiz "chen\n"

.text
main:
	la $t0,name		#using  name value which is an address, load this address to $t0

	la $t1,mick
	sw $t1,($t0)		#1,get value of $to, use it as the address of a piece of memory
	la $t1,alice
	sw $t1,4($t0)		#baseline : the content of  $t0 , offset :4
	la $t1,tony
	sw $t1,8($t0)
	la $t1,chen
	sw $t1,12($t0)

	li $v0,4
	lw $a0,0($t0)
	syscall			#what's the output while this syscall is done

	li $v0,10
	syscall
```

============in JAVA/C=========

```
int    i,j,k;                    //i,j,k are labels of address in memory
k = i+j;
```

============in MIPS============

1) i,j(memory) ----->>>>> $t0,$t1 (register)

load (target: Register): lb lw lh ld ; li , la.

2) add $t2,$t0,$t1

3) $t2 (register) ----->>>>> k(memory)

store (target: Memory) : sb sw sh sd.