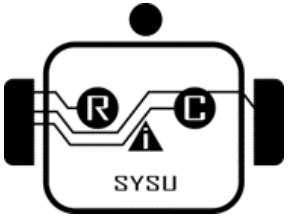
中山大學
SUN YAT-SEN UNIVERSITY



计算机组成原理

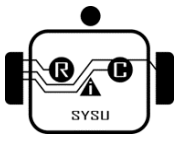
第二章：指令：计算机的语言

中山大学计算机学院
陈刚

2022年秋季

本讲内容

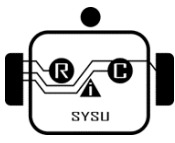
- MIPS指令集
  - 基本指令和指令类型
  - 程序的机器级表示
    - MIPS指令系统介绍
    - MIPS指令系统举例
    - 程序的调用
    - 汇编和仿真



Today

- Assembly programming
  - structure of an assembly program
  - assembler directives
  - data and text segments
  - allocating space for data
- MIPS assembler: MARS
  - development environment
- A few coding examples
  - self-study

Reading: Ch. 2.10



What is an Assembler?

- A program for writing programs
- Machine Language:
  - 1's and 0's loaded into memory.
    (Did anybody ever really do that?)
- Assembly Language:

Front panel of a classic PDP8e. The toggle switches were used to enter machine language.

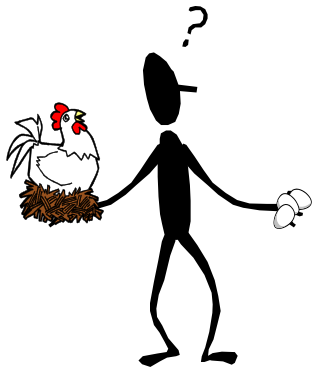
```
    .globl main
main:
    subu $sp, $sp, 24
    sw   $ra, 16($sp)
    li   $a0, 18
    li   $a1, 12
    li   $a2, 6
    jal  tak
    move $a0, $v0
```

→ *ASM* →

```
01101101
11000110
00101111
10110001
.....
```

STREAM of bits to be loaded into memory

?



Symbolic SOURCE text file

ASSEMBLER Translator program

Binary Machine Language

Assembler:

1. A Symbolic LANGUAGE for representing strings of bits
2. A PROGRAM for translating Assembly Source to binary

Assembly Source Language

An Assembly SOURCE FILE contains, in symbolic text, values of successive bytes to be loaded into memory... e.g.

```
.data 0x10000000
.byte 1, 2, 3, 4
.byte 5, 6, 7, 8
.word 1, 2, 3, 4
.asciiz "Comp 411"
```

Specifies “current” address, i.e., start of data
Four byte values
Another four byte values
Four word values (each is 4 bytes)
A zero-terminated ASCII string

Resulting memory dump:

```
[0x10000000]    0x04030201  0x08070605  0x00000001  0x00000002
[0x10000010]    0x00000003  0x00000004  0x706d6f43  0x31313420
```

Notice the byte ordering. Above is “XXX-endian” (The least significant byte of a word or half-word has the lowest address)

Assembler Syntax

- Assembler DIRECTIVES = Keywords prefixed with '.'
  - Control the placement and interpretation of bytes in memory

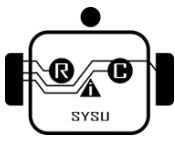
.data <addr>	Subsequent items are considered data
.text <addr>	Subsequent items are considered instructions
.align N	Skip to next address multiple of 2^N

  - Allocate Storage

.byte b_1, b_2, …, b_n	Store a sequence of bytes (8-bits)
.half h_1, h_2, …, h_n	Store a sequence of half-words (16-bits)
.word w_1, w_2, …, w_n	Store a sequence of words (32-bits)
.ascii "string"	Stores a sequence of ASCII encoded bytes
.asciiz "string"	Stores a zero-terminated string
.space n	Allocates n successive bytes

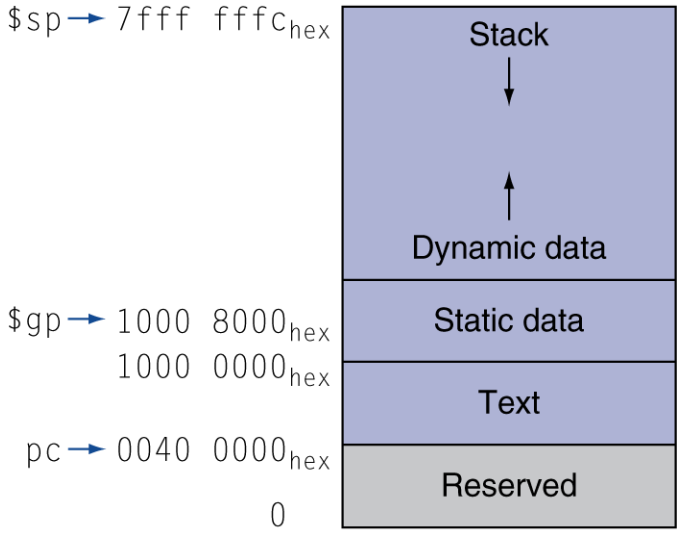
  - Define scope

.globl sym	Declares symbol to be visible to other files
.extern sym size	Sets size of symbol defined in another file (Also makes it directly addressable)



Memory Layout(recap)

- Text: program code
- Static data: global variables
  - e.g., static variables in C, constant arrays and strings
  - $gp initialized to address allowing ±offsets into this segment
- Dynamic data: heap
  - E.g., malloc in C, new in Java
- Stack: automatic storage



More Assembler Syntax

- Assembler COMMENTS
  - All text following a '#' (sharp) to the end of the line is ignored
- Assembler LABELS
  - Labels are symbols that represent memory addresses
    - labels take on the values of the address where they are declared
    - labels can be for data as well as for instructions
  - Syntax: <start_of_line><label><colon>

```
.data 0x80000000
item:   .word 1                 # a data word
.text 0x00010000
start:  add     $3, $4, $2      # an instruction label
        sll     $3, $3, 8
        andi    $3, $3, 0xff
        beq     ..., ..., start
```

Even More Assembler Syntax

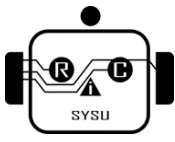
- Assembler PREDEFINED SYMBOLS
  - Register names and aliases

    $0-$31, $zero, $v0-$v1, $a0-$a3, $t0-$t9, $s0-$s7, $at, $k0-$k1, $gp, $sp, $fp, $ra

- Assembler MNEMONICS
  - Symbolic representations of individual instructions

    add, addu, addi, addiu, sub, subu, and, andi, or, ori, xor, xori, nor, lui, sll, sllv, sra, srav, srl, srlv, div, divu, mult, multu, mfhi, mflo, mthi, mtlo, slt, sltu, slti, sltiu, beq, bgez, bgezal, bgtz, blez, bltzal, bltz, bne, j, jal, jalr, jr, lb, lbu, lh, lhu, lw, lwl, lwr, sb, sh, sw, swl, swr, rfe

    - not allimplemented in all MIPS versions
  - *Pseudo-instructions* (mnemonics that are not instructions)
    - abs, mul, mulo, mulou, neg, negu, not, rem, remu, rol, ror, li, seq, sge, sgeu, sgt, sgtu, sle, sleu, sne, b, beqz, bge, bgeu, bgt, bgtu, ble, bleu, blt, bltu, bnez, la, ld, ulh, ulhu, ulw, sd, ush, usw, move,syscall, break, nop
    - not real MIPS instructions; broken down by assembler into real ones



An Aside: Pseudoinstructions

MIPS has relatively few instructions, however, it is possible to "fake" new instructions by taking advantage of special ISA properties (i.e. %0 is always zero, clever use of immediate values)

Examples:

move	$d,$s	becomes	addi	$d,$s,0
neg	$d,$s	becomes	sub	$d,$0,$s
negu	$d,$s	becomes	subu	$d,$0,$s
not	$d,$s	becomes	nor	$d,$s,$0
subiu	$d,$s,imm16	becomes	addiu	$d,$s,-imm16
b	label	becomes	beq	$0,$0,label
sge	$d,$s,$t	becomes	slt	$d,$t,$s
nop		becomes	sll	$0,$0,0

Which, BTW, assembles to 0x00000000

A Simple Programming Task

- Add the numbers 0 to 4 …
  - 10 = 0 + 1 + 2 + 3 + 4
- Program in "C":

```
int i, sum;

main() {
     sum = 0;
     for (i=0; i<5; i++)
          sum = sum + i;
}
```

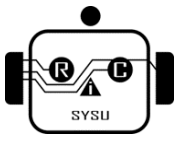
- Now let's code it in ASSEMBLY

First Step: Variable Allocation

- Two integer variables (by default 32 bits in MIPS)

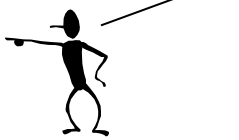
```
.data 0x80000000
.globl sum
.globl i
sum:    .space 4
i:      .space 4
```

- Thing to note:
  - ".data" assembler directive places the following words into the data segment
  - ".globl" directives make the "sum" and "i" variables visible to all other assembly modules (in other files)
  - ".space" directives allocate 4 bytes for each variable
    - in contrast to ".word", ".space" does not initialize the variables



Actual "Code"

- Next we write ASSEMBLY code using instr mnemonics



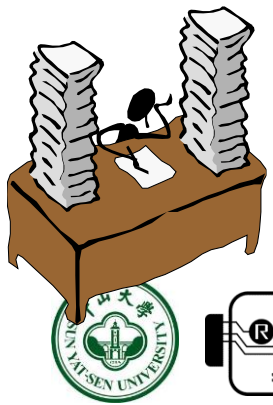
A common convention, which originated with the 'C' programming language, is for the entry point (starting location) of a program to named "main".

```
.text 0x10000000
.globl main
main:
      add   $8,$0,$0       # sum = 0
      add   $9,$0,$0       # for (i = 0; ...
loop:
      addu  $8,$8,$9       # sum = sum + i;
      addi  $9,$9,1        # for (...; ...; i++
      slti  $10,$9,5       # for (...; i<5;
      bne   $10,$0,loop
end:
```

Bookkeeping:

1) Register $8 is allocated as the "sum" variable
2) Register $9 is allocated as the "i" variable

We will talk about how to exit a program later



MARS

- MIPS Assembler and Runtime Simulator (MARS)
  - Java application
  - Runs on all platforms
  - Links on class website
  - Download it now!