ICS 233
Computer Architecture & Assembly Language

MIPS PROCESSOR INSTRUCTION SET

ICS 233
Computer Architecture & Assembly Language

Lecture 10

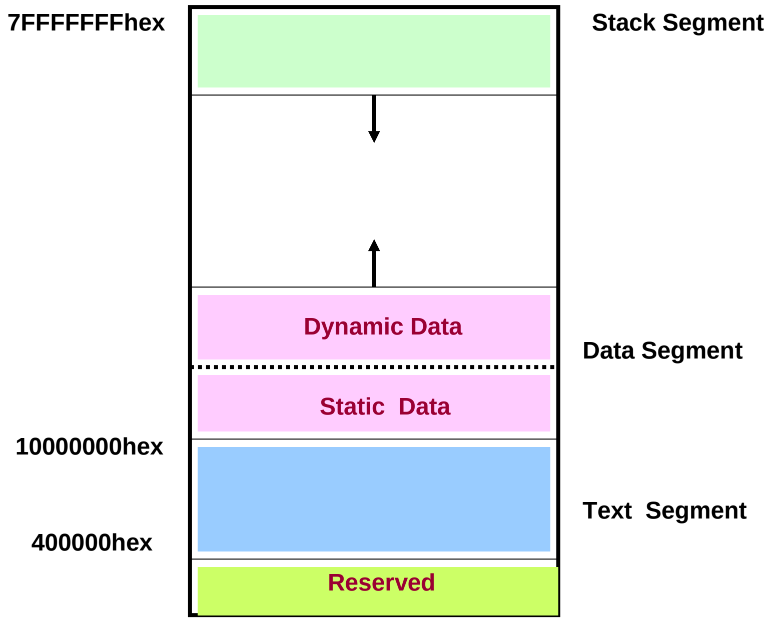
Lecture Outline

- SPIM MIPS Simulator
- Assembly Language statements
- System Calls
- Assembler Pseudo-instructions

Memory Usage

- **Systems based on MIPS processors typically divide memory into three parts :**
  - **Text Segment**
  - **Data Segment**
  - **Stack Segment**
- **Text segment** is the first part of the memory near the bottom of the address space starting at address 400000hex, which holds the program's instructions.
- **Data segment** which is second part of the memory above the text segment which is further divided into two parts :
  - **Static data** starting at address 10000000hex contains objects whose size is known to the compiler and whose lifetime – i.e., the interval during which a program can access them – is the program's entire execution.
  - **Dynamic Data** which is immediately above static data. This data as its name implies, is allocated by the program as it executes.
- **Stack Segment** is the third part of the memory which resides at the top of the virtual address space starting at address 7FFFFFFF hex.
  - Like dynamic data, the maximum size of a program's stack is not known in advance.
  - As the program pushes values onto the stack, the operating system expands the stack segment down towards the data segment

MEMORY LAYOUT



SPIM - MIPS SIMULATOR

- SPIM is a software simulator that runs programs written for MIPS R2000/R3000 processors
- SPIM's name is just MIPS spelled backwards
- SPIM can read and immediately execute assembly language files.
- SPIM is a self-contained system for running MIPS programs.
- It contains a debugger and provides a few operating system-like services.

SPIM - MIPS SIMULATOR

- SPIM comes in multiple versions

spim

- It is a command line-driven program and requires only an alphanumeric terminal to display it.
- It operates like most programs of this type : type a line of text,i.e., command, hit the return key and spim executes the command

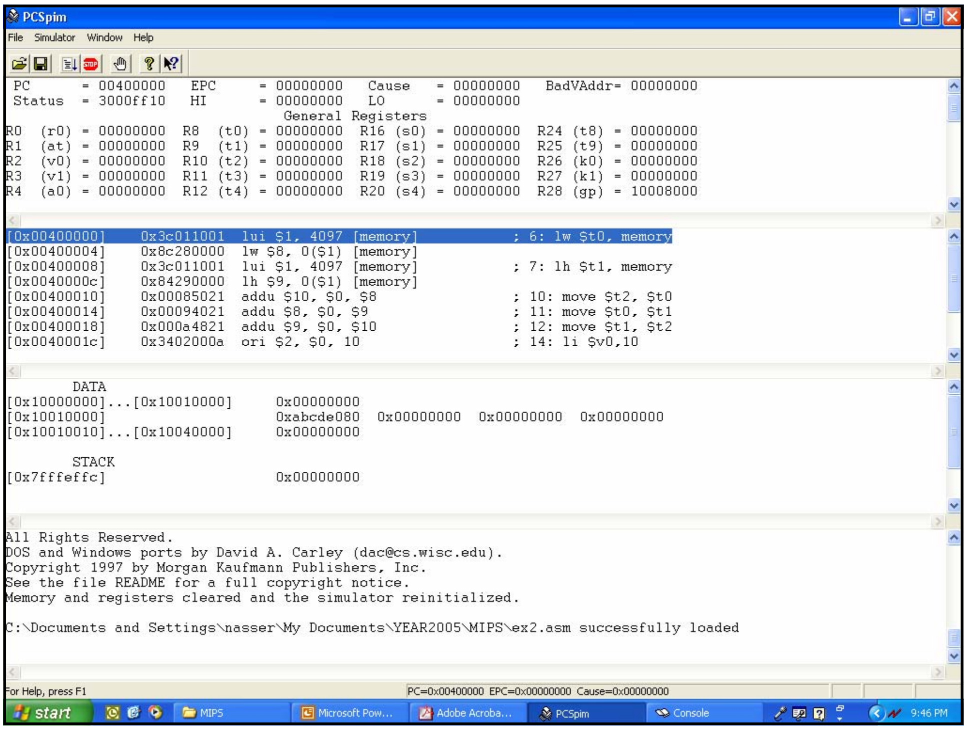
xspim

- It runs in the X-window environment of the UNIX system.
- It is a much easier program to learn and use because its commands are always visible on the screen and because it continually displays the machine's register

PCspim

- It is compatible with Microsoft Windows 3.1, Windows 95/XP and Windows NT

- The UNIX, Windows, and DOS versions of SPIM are available through www.mkp.com/cod2e.htm



Assembly Language Statements

- **Three types of statements in assembly language**
  - Typically, one statement should appear on a line

1. **Executable Instructions**
   - Generate machine code for the processor to execute at runtime
   - Instructions tell the processor what to do
2. **Pseudo-Instructions and Macros**
   - Translated by the assembler into real instructions
   - Simplify the programmer task
3. **Assembler Directives**
   - Provide information to the assembler while translating a program
   - Used to define segments, allocate memory variables, etc.
   - Non-executable: directives are not part of the instruction set

Instructions

- **Assembly language instructions have the format:**

```
[label:]   mnemonic   [operands]   [#comment]
```

- **Label: (optional)**
  - Marks the address of a memory location, must have a colon
  - Typically appear in data and text segments
- **Mnemonic**
  - Identifies the operation (e.g. **add**, **sub**, etc.)
- **Operands**
  - Specify the data required by the operation
  - Operands can be registers, memory variables, or constants
  - Most instructions have three operands

```
L1:   addiu $t0, $t0, 1        #increment $t0
```

Comments

- **Comments are very important!**
  - Explain the program's purpose
  - When it was written, revised, and by whom
  - Explain data used in the program, input, and output
  - Explain instruction sequences and algorithms used
  - Comments are also required at the beginning of every procedure
    - Indicate input parameters and results of a procedure
    - Describe what the procedure does
- **Single-line comment**
  - Begins with a hash symbol # and terminates at end of line

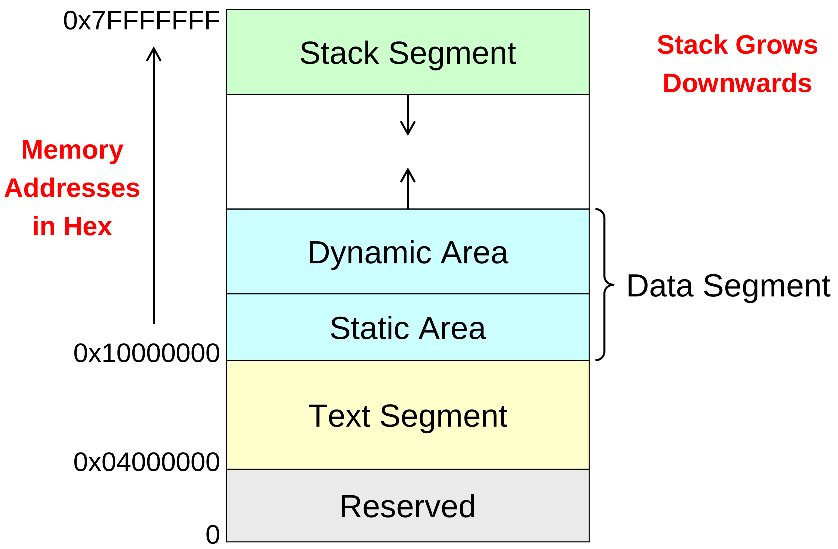
Program Template

```
# Title:                        Filename:
# Author:                       Date:
# Description:
# Input:
# Output:
################# Data segment###################
.data
 . . .
################# Code segmen###################
.text
.globl main
main:                           # main program entry
 . . .
li $v0, 10                      # Exit program
syscall
```

.DATA, .TEXT, & .GLOBL Directives

- **.DATA** directive
  - Defines the data segment of a program containing data
  - The program's variables should be defined under this directive
  - Assembler will allocate and initialize the storage of variables
- **.TEXT** directive
  - Defines the code segment of a program containing instructions
- **.GLOBL** directive
  - Declares a symbol as global
  - Global symbols can be referenced from other files
  - We use this directive to declare *main* procedure of a program

Layout of a Program in Memory



SPIM Assembler Segment Directives

Name	Arguments	Description
.text	*addr*	**Defines the Text Segment** (Code Segment) The items following this statement are to be assembled into the text segment. By default, begin at the next available address in the text segment. If the optional argument addr is present, then begin at addr. In SPIM, the only items that can be assembled into the text segment are instructions
.data	*addr*	**Defines the Data Segment** The items following this statement are to be assembled into the segment. By default, begin at the next available address in the data segment. If the optional argument addr is present, then begin at addr.
.ktext	*addr*	**Defines the Kernel Text Segment** Like the Text segment, but used by the Operating System
.kdata	*addr*	**Defines the Kernel Data Segment** Like the Data segment, but used by the Operating System

SPIM Assembler Linker Directives

Name	Arguments	Description
.extern	*sym size*	**Declare as global the label *sym*, and declare that it is size bytes in length (this information can be used by the assembler)**
.globl	*sym*	**Declares as global the label *sym***

Data Definition Statement

- Sets aside storage in memory for a variable
- May optionally assign a name (label) to the data
- Syntax:

  [*name:*] *directive* *initializer* [, *initializer*] . . .

```
var1: .WORD    10
```

- All initializers become binary data in memory

Data Directives

- **.BYTE** Directive
  - Stores the list of values as 8-bit bytes
- **.HALF** Directive
  - Stores the list as 16-bit values aligned on half-word boundary
- **.WORD** Directive
  - Stores the list as 32-bit values aligned on a word boundary
- **.FLOAT** Directive
  - Stores the listed values as single-precision floating point
- **.DOUBLE** Directive
  - Stores the listed values as double-precision floating point

String Directives

- **.ASCII** Directive
  - Allocates a sequence of bytes for an ASCII string
- **.ASCIIZ** Directive
  - Same as **.ASCII** directive, but adds a NULL char at end of string
  - Strings are null-terminated, as in the C programming language
- **.SPACE** Directive
  - Allocates space of *n* uninitialized bytes in the data segment

Examples of Data Definitions

```
.DATA
var1:  .BYTE     'A', 'E', 127, -1, '\n'
var2:  .HALF     -10, 0xffff
var3:  .WORD     0x12345678
var4:  .FLOAT    12.3, -0.1
var5:  .DOUBLE   1.5e-10
str1:  .ASCII    "A String\n"
str2:  .ASCIIZ   "NULL Terminated String"
array: .SPACE    100
```

SPIM Assembler Data Directives

Name	Arguments	Description
.ascii	*str*	Assemble the given string in memory. Do not null-terminate.
.asciiz	*str*	.Assemble the given string in memory. Do null-terminate.
.byte	*byte1 ……………byteN*	Assemble the given bytes (8-bit integers)
.half	*half1 ……………halfN*	Assemble the given halfwords (16-bit integers)
.word	*word1 ……………wordN*	Assemble the given words (32-bit integers)
.space	*size*	Allocate size bytes of space in the current segment. In SPIM, this is only permitted in the data segment.
.align	*n*	Align the next item on the next 2^n byte boundary. .align 0 turns off automatic alignment.


```
# Program to compute N1 x N2
                  (signed numbers)
        .text
        .globl  main
main:
        lw $t0, N1
        lw $t1, N2
        mult  $t0, $t1
        mflo $t2
        mfhi $t3
        sw  $t2, PRDL
        sw  $t3,PRDH

        li $v0,10
        syscall              # exit

        .data
N1:     .word 0xFFFFFFFF
N2:     .word 0x0000000F
PRDL:   .word 0x00000000
PRDH:   .word 0x00000000
```

```
# Program to compute N1 x N2
                  (unsigned numbers)
        .text
        .globl    main
main:
        lw $t0, N1
        lw $t1, N2
        multu  $t0, $t1
        mflo $t2
        mfhi $t3
        sw  $t2, PRDL
        sw  $t3,PRDH

        li $v0,10
        syscall              # exit

        .data
N1:     .word 0xFFFFFFFF
N2:     .word 0x0000000F
PRDL:   .word 0x00000000
PRDH:   .word 0x00000000
```

```
# Program to compute N1 / N2
            (signed numbers)
        .text
        .globl   main
main:
        lw $t0, N1
        lw $t1, N2
        div  $t0, $t1
        mflo $t2
        mfhi $t3
        sw  $t2, QUOT
        sw  $t3, REM

        li $v0,10
        syscall             # exit

        .data
N1:     .word 0xFFFFFFFF
N2:     .word 0x0000000F
QUOT:   .word 0x00000000
REM:    .word 0x00000000
```

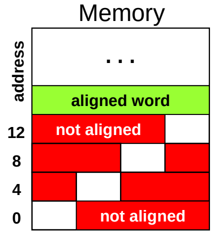
```
# Program to compute N1 / N2
            (unsigned numbers)
        .text
        .globl   main
main:
        lw $t0, N1
        lw $t1, N2
        divu  $t0, $t1
        mflo $t2
        mfhi $t3
        sw  $t2, QUOT
        sw  $t3, REM

        li $v0,10
        syscall             # exit

        .data
N1:     .word 0xFFFFFFFF
N2:     .word 0x0000000F
QUOT:   .word 0x00000000
REM:    .word 0x00000000
```


Memory Alignment

- **Memory is viewed as an array of bytes with addresses**
  - Byte Addressing: address points to a byte in memory
- **Words occupy 4 consecutive bytes in memory**
  - MIPS instructions and integers occupy 4 bytes
- **Alignment: address is a multiple of size**
  - Word address should be a multiple of 4
    - Least significant 2 bits of address should be 00
  - Halfword address should be a multiple of 2
- **.ALIGN n directive**
  - Aligns the next data definition on a 2^n byte boundary



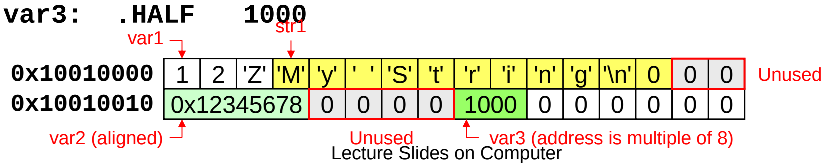
Symbol Table

- **Assembler builds a symbol table for labels (variables)**
  - Assembler computes the address of each label in data segment
- **Example**

```
.DATA
var1:   .BYTE   1, 2,'Z'
str1:   .ASCIIZ "My String\n“
.ALIGN  2
var2:   .WORD   0x12345678
.ALIGN  3
var3:   .HALF   1000
```

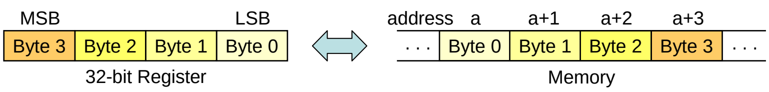
Symbol Table

Label	Address
var1	0x10010000
str1	0x10010003
var2	0x10010010
var3	0x10010018

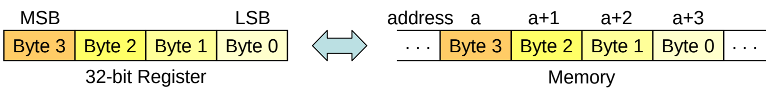


Byte Ordering and Endianness

- Processors can order bytes within a word in two ways
- Little Endian Byte Ordering
  - Memory address = Address of **least significant byte**
  - Example: Intel IA-32, Alpha



- Big Endian Byte Ordering
  - Memory address = Address of **most significant byte**
  - Example: SPARC, PA-RISC



- MIPS can operate with both byte orderings

SPIM - MIPS SIMULATOR

❑ System Calls

- **SPIM provides a small set of operating-system like services through the system call (syscall) instruction.**
- **System calls are used to invoke services to perform system Input and Output operations.**
- **To request a service, a program loads the system call code into register $v0 and arguments into registers $a0-$a3 (or $f12 for floating-point values).**
- **System calls that return values put their results in register $v0 (or $f0 for floating-point results)**
- **When a program reads or writes, its I/O appears in a separate window, called the console, which pops up when needed.**

Service	System Call Code	Arguments	Operation
print_int	1	$a0 = integer	Passes an integer in $a0 as argument and displays it on the console
print_float	2	$f12 = float	Passes single precision floating point number in $f12 as argument and displays it on the console
print_double	3	$f12 = double	Passes double precision floating point number in $f12 as argument and displays it on the console
print_string	4	$a0 = string	Passes a pointer to a null-terminated string in $a0 as argument and displays it on the console
read_int	5		Reads an integer from the console and returns it in $v0
read_float	6		Reads a single floating point number from the console and returns it in $f0
read_double	7		Reads a double floating point number from the console and returns it in $f0
read_string	8	$a0 = buffer, $a1 = length	Reads up to length-1 characters from the console into a buffer (address in $a0) and terminates the string with a null byte
sbrk	9	$a0=amount	Returns a pointer to a block of memory in $v0
exit	10		exits from program

System Calls

- **Programs do input/output through system calls**
- **MIPS provides a special `syscall` instruction**
  - To obtain services from the operating system
  - Many services are provided in the SPIM and MARS simulators
- **Using the `syscall` system services**
  - Load the service number in register $v0
  - Load argument values, if any, in registers $a0, $a1, etc.
  - Issue the `syscall` instruction
  - Retrieve return values, if any, from result registers

Syscall Services

Service	$v0	Arguments / Result
Print Integer	1	$a0 = integer value to print
Print Float	2	$f12 = float value to print
Print Double	3	$f12 = double value to print
Print String	4	$a0 = address of null-terminated string
Read Integer	5	$v0 = integer read
Read Float	6	$f0 = float read
Read Double	7	$f0 = double read
Read String	8	$a0 = address of input buffer $a1 = maximum number of characters to read
Exit Program	10	
Print Char	11	$a0 = character to print
Read Char	12	$a0 = character read

Supported by MARS

Reading and Printing an Integer

```
################# Code segment#################
.text
.globl main
main:                           # main program entry
  li    $v0, 5                  # Read integer
  syscall                       # $v0 = value read

  move  $a0, $v0                # $a0 = value to print
  li    $v0, 1                  # Print integer
  syscall

  li    $v0, 10                 # Exit program
  syscall
```


Reading and Printing a String

```
################# Data segment#################
.data
  str: .space  10             # array of 10 bytes
################# Code segment#################
.text
.globl main
main:                         # main program entry
  la    $a0, str              # $a0 = address of str
  li    $a1, 10               # $a1 = max string length
  li    $v0, 8                # read string
  syscall
  li    $v0, 4                # Print string str
  syscall
  li    $v0, 10               # Exit program
  syscall
```

Program 1: Sum of Three Integers

```
# Sum of three integers
#
# Objective: Computes the sum of three integers.
#      Input: Requests three numbers.
#     Output: Outputs the sum.
################### Data segment ###################
.data
prompt:  .asciiz     "Please enter three numbers: \n"
sum_msg: .asciiz     "The sum is: "
################### Code segment ###################
.text
.globl main
main:
      la    $a0,prompt       # display prompt string
      li    $v0,4
      syscall
      li    $v0,5            # read 1st integer into $t0
      syscall
      move  $t0,$v0
```


Sum of Three Integers – Continued

```
      li    $v0,5            # read 2nd integer into $t1
      syscall
      move  $t1,$v0

      li    $v0,5            # read 3rd integer into $t2
      syscall
      move  $t2,$v0

      addu  $t0,$t0,$t1      # accumulate the sum
      addu  $t0,$t0,$t2

      la    $a0,sum_msg      # write sum message
      li    $v0,4
      syscall

      move  $a0,$t0          # output sum
      li    $v0,1
      syscall

      li    $v0,10           # exit
      syscall
```

Program 2: Case Conversion

```
# Objective: Convert lowercase letters to uppercase
#     Input: Requests a character string from the user.
#    Output: Prints the input string in uppercase.
################### Data segment #####################
.data
name_prompt:.asciiz       "Please type your name: "
out_msg:    .asciiz       "Your name in capitals is: "
in_name:    .space 31     # space for input string
################### Code segment #####################
.text
.globl main
main:
      la    $a0,name_prompt     # print prompt string
      li    $v0,4
      syscall
       la   $a0,in_name    # read the input string
      li    $a1,31         # at most 30 chars + 1 null char
      li    $v0,8
      syscall
```


Case Conversion – Continued

```
       la    $a0,out_msg       # write output message
      li    $v0,4
      syscall
      la    $t0,in_name
loop:
      lb    $t1,($t0)
      beqz  $t1,exit_loop     # if NULL, we are done
      blt   $t1,'a',no_change
      bgt   $t1,'z',no_change
      addiu $t1,$t1,-32       # convert to uppercase: 'A'-'a'=-32
no_change:
      sb    $t1,($t0)
      addiu $t0,$t0,1         # increment pointer
      j     loop
exit_loop:
      la    $a0,in_name       # output converted string
      li    $v0,4
      syscall
      li    $v0,10            # exit
      syscall
```

SPIM - MIPS SIMULATOR

❑ **Assembler Pseudoinstructions**

- **SPIM provides assembler pseudoinstructions which are not real instructions of the MIPS processor**
- **SPIM translates assembler pseudoinstructions into one to three MIPS instructions.**

❖ Use MIPS simulator, SPIM available at http://www.cs.wisc.edu/~larus/spim.html

Assembler Pseudoinstructions

❑ **li (load immediate register with a value)**

- **Instruction Mnemonic :**

  li rd, const ;where rd is a register,
  ; const is a value

- **Meaning :**

  rd ← const

- **Example :**

  i) li $v0, 4 ; $v0← 4

  translated to ori $2, $0, 4

  ii) li $t0, 0xABCDEF90

  translated to lui $at, 0xABCD
  ori $t0, $at, 0xEF90

Assembler Pseudoinstructions

- **la (load register with address)**

- **Instruction Mnemonic :**

  la rd, addr ;where rd is a register,
  ; addr is the label of the memory location

- **Meaning :**

  rd ← address of the location having the label addr

- **Example :**

  la $v0, mem-addr ; $v0← address of mem_addr

  translated to **lui $at, mem_addr_upper16bits**
  ori $v0, $at, mem_addr_lower16bits

Assembler Pseudoinstructions

- **move**

  - Moves data between registers directly

- **Instruction Mnemonic :**

  move rd, rs ;where rs, rd are registers,

- **Meaning :**

  rd ← rs

- **Example :**

  move $a0, $t0 ; $a0← $t0

  translated to **addu $4, $0, $8**
  same as or $4, $0, $8

Assembler Pseudoinstructions

- **abs**

  - gets absolute value

- **Instruction Mnemonic :**

  abs rd, rs ;where rs, rd are registers,

- **Meaning :**

  rd ← | rs |

- **Example :**

  abs $a0, $t0 ; $a0← | $t0 |

  translated to add $a0, $0, $t0
  bgez $t0, skip
  sub $a0, $0, $t0
  skip:

Assembler Pseudoinstructions

- **not (logical not)**

- **Instruction Mnemonic :**

  not rd, rs ;where rs, rd are registers,

- **Meaning :**

  rd ← not rs

- **Example :**

  not $a0, $t0 ; $a0← not $t0

  translated to nor $a0, $t0, $0

Assembler Pseudoinstructions

- ❑ **neg** (negate)
- ➢ **Instruction Mnemonic :**
  neg rd, rs ;where rs, rd are registers
- ➢ **Meaning :**
  rd ← - rs
- ➢ **Example :**
  neg $a0, $t0 ; $a0← - $t0
  translated to sub $a0, $0, $t0

- ❑ **negu** (negate unsigned)
- ➢ **Instruction Mnemonic :**
  negu rd, rs ;where rs, rd are registers
- ➢ **Meaning :**
  rd ← - rs
- ➢ **Example :**
  negu $a0, $t0 ; $a0← - $t0
  translated to subu $a0, $0, $t0

Assembler Pseudoinstructions

- ❑ **rem** (remainder)
- ➢ **Instruction Mnemonic :**
  rem rd, rs, rt ;where rs, rt, rd are registers,
- ➢ **Meaning :**
  rd ← remainder of rs/rt
- ➢ **Example :**
  rem $t0, $t1, $t2 ; $t0← rem ($t1 / $t2)

- ❑ **remu** (remainder unsigned)
- ➢ **Instruction Mnemonic :**
  remu rd, rs, rt ;where rs, rt, rd are registers,
- ➢ **Meaning :**
  rd ← remainder of rs/rt
- ➢ **Example :**
  remu $t0, $t1, $t2 ; $t0← rem ($t1 / $t2)

Assembler Pseudoinstructions

- **rol** (rotate left)
  - **Instruction Mnemonic :**
    rol rd, rs, const ;where rs, rd are registers,
  - **Meaning :**
    rd ← rotate rs left const bits
  - **Example :**
    rol $t0, $t1, 4
    ; rotate contents of $t1 left by 4 bits and store the result in $t0

- **ror** (rotate right)
  - **Instruction Mnemonic :**
    ror rd, rs, const ;where rs, rd are registers,
  - **Meaning :**
    rd ← rotate rs right const bits
  - **Example :**
    ror $t0, $t1, 3
    ; rotate contents of $t1 right by 3 bits and store the result in $t0

Assembler Pseudoinstructions

- **Question : Expand the following pseudo-instruction**
  rol $t1, $t0, 1

```
srl  $at, $t0, 31
sll  $t1, $t0, 1
or   $t1, $t1, $at
```

- **Question : Expand the following pseudo-instruction**
  rol $t1, $t0, 4

```
srl  $at, $t0, 28
sll  $t1, $t0, 4
or   $t1, $t1, $at
```

Assembler Pseudoinstructions

- **Question : Expand the following pseudo-instruction**
  ror $t1, $t0, 1

  sll $at, $t0, 31
  srl $t1, $t0, 1
  or $t1, $t1, $at

- **Question : Expand the following pseudo-instruction**
  ror $t1, $t0, 4

  sll $at, $t0, 28
  srl $t1, $t0, 4
  or $t1, $t1, $at

Pseudo-Instructions

- Introduced by assembler as if they were real instructions
  - To facilitate assembly language programming
  - Assembler reserves $at = $1 for its own use
  - `$at` is called the assembler temporary register

Pseudo-Instructions	Conversion to Real Instructions
`move $s1, $s2`	`addu Ss1, $s2, $zero`
`not $s1, $s2`	`nor $s1, $s2, $s2`
`li $s1, 0xabcd`	`ori $s1, $zero, 0xabcd`
`li $s1, 0xabcd1234`	`lui $s1, 0xabcd` `ori $s1, $s1, 0x1234`
`sgt $s1, $s2, $s3`	`slt $s1, $s3, $s2`
`blt $s1, $s2, label`	`slt $at, $s1, $s2` `bne $at, $zero, label`

#Example : Swap values in registers $s0 and $s1

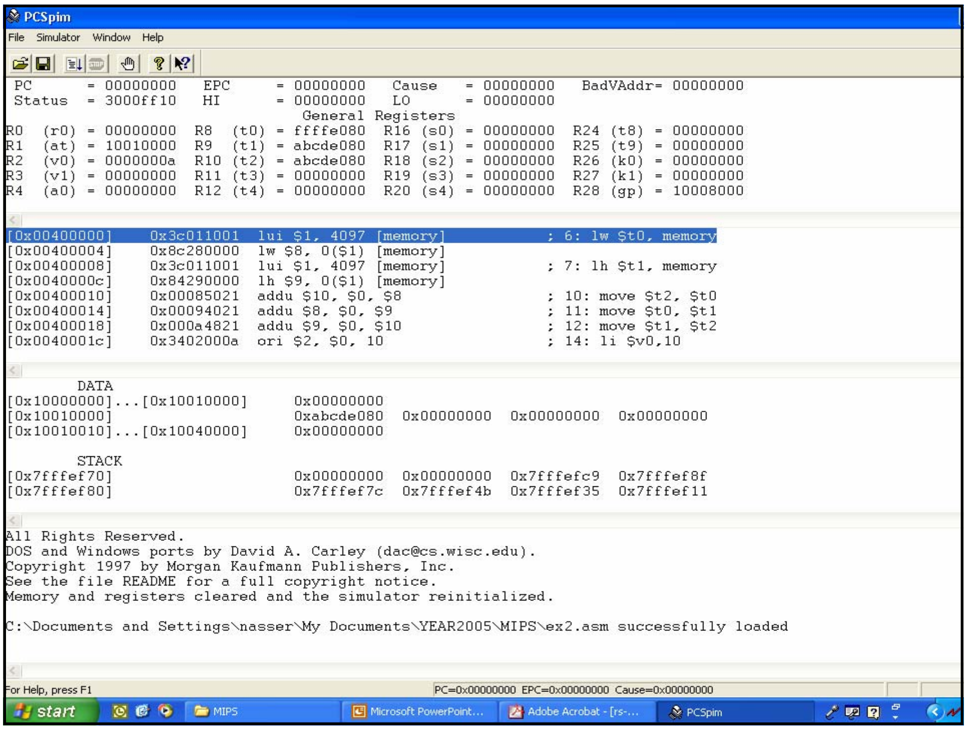
```
        .text
        .globl  main
main:

        lw  $s0, val1
        lw  $s1, val2

        # swap values $s0 and $s1
        move $s2, $s0
        move $s0, $s1
        move $s1, $s2

        li $v0,10
        syscall             # exit

        .data
val1:   .word 0xABCDEF98
val2:   .word 0x76543210
```

Assembler Pseudoinstructions

- **Question :** **Expand the following pseudo instruction to MIPS instruction**
  and $t0, $t0, 0xABCDEF98

translated to MIPS instruction

```
lui  $at, 0xABCD
ori  $at, 0xEF98
and $t0, $t0, $at
```

- **Question :** **Swap (exchange) the contents of register $s0 and $s1 without using memory accesses and without using temporary registers.**

```
xor  $s0, $s0, $s1
xor  $s1, $s0, $s1
xor  $s0, $s0, $s1
```



```
## load.asm - example demonstrating load instructions
##
##        t0 - holds word from memory location mem_addr
##        t1 - holds half word from memory location mem_addr
##        t2 - holds byte from memory location mem_addr
##        t3 - holds half word without sign extension from memory location mem_addr
##        t4 - holds byte without sign extension from memory location mem_addr
##
##        syscall used - print interger (call code 1)
##        syscall used - print string (call code 4)
##
#################################################
#                                               #
#                  text segment                 #
#                                               #
#################################################
        .text
        .globl main
main:
        lw  $t0,mem_addr     # load word into  $t0
        lh  $t1,mem_addr     # load half word into  $t1
        lb  $t2,mem_addr     # load byte into  $t2
        lhu  $t3,mem_addr    # load halfword unsigned into  $t3
        lbu  $t4,mem_addr    # load byte unsigned  into  $t4
```

```
la $a0, message1    # $a0 with message1 address
li $v0, 4
syscall
move $a0, $t0       # $a0 with data from $t0
li $v0, 1
syscall
la $a0,endl         # system call to print
li $v0,4            # out a newline
syscall

la $a0, message2    # $a0 with message2 address
li $v0, 4
syscall
move $a0, $t1       # $a0 with data from $t1
li $v0, 1
syscall
la $a0,endl         # system call to print
li $v0,4            # out a newline
syscall
```



```
la $a0, message3    # $a0 with message3 address
li $v0, 4
syscall
move $a0, $t2       # $a0 with data from $t2
li $v0, 1
syscall
la $a0,endl         # system call to print
li $v0,4            # out a newline
syscall

la $a0, message4    # $a0 with message4 address
li $v0, 4
syscall
move $a0, $t3       # $a0 with data from $t3
li $v0, 1
syscall
la $a0,endl         # system call to print
li $v0,4            # out a newline
syscall
```

```
            la $a0, message5    # $a0 with message5 address
            li $v0, 4
            syscall
            move $a0, $t4       # $a0 with data from $t4
            li $v0, 1
            syscall
            la $a0,endl         # system call to print
            li $v0,4            # out a newline
            syscall

            li $v0,10
            syscall             # exit
##################################################
#                                                #
#                    data segment                #
#                                                #
##################################################
                     .data
mem_addr:            .word  0x456789AB
message1:            .asciiz "load word : "
message2:            .asciiz "load halfword : "
message3:            .asciiz "load byte : "
message4:            .asciiz "load halfword unsigned : "
message5:            .asciiz "load byte unsigned : "
endl:                .asciiz  "\n"
```


Assembler Pseudoinstructions

- **seq (set equal)**
  - **Instruction Mnemonic :**
    seq rd, rs, rt ;where rs, rt, rd are registers,
  - **Meaning :**
    if (rs == rt) then rd = 1 else rd = 0
  - **Example :**
    seq $s1, $s2, $s3 ; if ($s2 == $s3) then $s1=1 else $s1=0

- **sne (set not equal)**
  - **Instruction Mnemonic :**
    sne rd, rs, rt ;where rs, rt, rd are registers,
  - **Meaning :**
    if (rs != rt) then rd = 1 else rd = 0
  - **Example :**
    sne $s1, $s2, $s3 ; if ($s2 != $s3) then $s1=1 else $s1=0

Assembler Pseudoinstructions

- **sgt (greater than)**

➢ **Instruction Mnemonic :**

sgt rd, rs, rt ;where rs, rt, rd are registers,

➢ **Meaning :**

if (rs > rt) then rd = 1 else rd = 0

➢ **Example :**

sgt $s1, $s2, $s3 ; if ($s2 > $s3) then $s1=1 else $s1=0

- **sge (greater than or equal)**

➢ **Instruction Mnemonic :**

sge rd, rs, rt ;where rs, rt, rd are registers,

➢ **Meaning :**

if (rs >= rt) then rd = 1 else rd = 0

➢ **Example :**

sge $s1, $s2, $s3 ; if ($s2 >= $s3) then $s1=1 else $s1=0

Assembler Pseudoinstructions

- **sgtu (greater than unsigned)**

➢ **Instruction Mnemonic :**

sgtu rd, rs, rt ;where rs, rt, rd are registers,

➢ **Meaning :**

if (rs > rt) then rd = 1 else rd = 0

➢ **Example :**

sgtu $s1, $s2, $s3 ; if ($s2 > $s3) then $s1=1 else $s1=0

- **sgeu (greater than or equal unsigned)**

➢ **Instruction Mnemonic :**

sgeu rd, rs, rt ;where rs, rt, rd are registers,

➢ **Meaning :**

if (rs >= rt) then rd = 1 else rd = 0

➢ **Example :**

sgeu $s1, $s2, $s3 ; if ($s2 >= $s3) then $s1=1 else $s1=0

Assembler Pseudoinstructions

- **sle** (less than or equal)
  - **Instruction Mnemonic :**
    sle rd, rs, rt ;where rs, rt, rd are registers,
  - **Meaning :**
    if (rs <= rt) then rd = 1 else rd = 0
  - **Example :**
    sle $s1, $s2, $s3 ; if ($s2 <= $s3) then $s1=1 else $s1=0

- **sleu** (less than or equal unsigned)
  - **Instruction Mnemonic :**
    sleu rd, rs, rt ;where rs, rt, rd are registers,
  - **Meaning :**
    if (rs <= rt) then rd = 1 else rd = 0
  - **Example :**
    sleu $s1, $s2, $s3 ; if ($s2 <= $s3) then $s1=1 else $s1=0

Assembler Pseudoinstructions

- **bgt** (branch on greater than)
  - **Instruction Mnemonic :**
    bgt rd, rs, addr ;where rs, rd are registers,
    ; addr is the label of the target location
  - **Meaning :**
    if (rd > rs) then branch to location addr
    i.e., goto PC + 4 + const*4 (i.e., PC = Updated PC + offset)
  - **Example :**
    bgt $s1, $s2, up ; if ($s1 > $s2) goto target location up

---

- **bge** (branch on greater than or equal)
  - **Instruction Mnemonic :**
    bge rd, rs, addr ;where rs, rd are registers,
    ; addr is the label of the target location
  - **Meaning :**
    if (rd >= rs) then branch to location addr
    i.e., goto PC + 4 + const*4 (i.e., PC = Updated PC + offset)
  - **Example :**
    bge $s1, $s2, loop ; if ($s1 >= $s2) goto target location loop

Assembler Pseudoinstructions

- **bgtu** (branch on greater than unsigned)
  - **Instruction Mnemonic :**
    bgtu rd, rs, addr ;where rs, rd are registers,
    ; addr is the label of the target location
  - **Meaning :**
    if (rd > rs) then branch to location addr
    i.e., goto PC + 4 + const*4 (i.e., PC = Updated PC + offset)
  - **Example :**
    bgtu $s1, $s2, up ; if ($s1 > $s2) goto target location up

---

- **bgeu** (branch on greater than or equal unsigned)
  - **Instruction Mnemonic :**
    bgeu rd, rs, addr ;where rs, rd are registers,
    ; addr is the label of the target location
  - **Meaning :**
    if (rd >= rs) then branch to location addr
    i.e., goto PC + 4 + const*4 (i.e., PC = Updated PC + offset)
  - **Example :**
    bgeu $s1, $s2, loop ; if ($s1 >= $s2) goto target location loop

Assembler Pseudoinstructions

- **blt** (branch on less than)
  - **Instruction Mnemonic :**
    blt rd, rs, addr ;where rs, rd are registers,
    ; addr is the label of the target location
  - **Meaning :**
    if (rd < rs) then branch to location addr
    i.e., goto PC + 4 + const*4 (i.e., PC = Updated PC + offset)
  - **Example :**
    blt $s1, $s2, up ; if ($s1 < $s2) goto target location up

---

- **ble** (branch on less than or equal)
  - **Instruction Mnemonic :**
    ble rd, rs, addr ;where rs, rd are registers,
    ; addr is the label of the target location
  - **Meaning :**
    if (rd <= rs) then branch to location addr
    i.e., goto PC + 4 + const*4 (i.e., PC = Updated PC + offset)
  - **Example :**
    ble $s1, $s2, loop ; if ($s1 <= $s2) goto target location loop

Assembler Pseudoinstructions

- **bltu** (branch on less than unsigned)
  - **Instruction Mnemonic :**
    bltu rd, rs, addr ;where rs, rd are registers,
    ; addr is the label of the target location
  - **Meaning :**
    if (rd < rs) then branch to location addr
    i.e., goto PC + 4 + const*4 (i.e., PC = Updated PC + offset)
  - **Example :**
    bltu $s1, $s2, up ; if ($s1 < $s2) goto target location up

---

- **bleu** (branch on less than or equal unsigned)
  - **Instruction Mnemonic :**
    bleu rd, rs, addr ;where rs, rd are registers,
    ; addr is the label of the target location
  - **Meaning :**
    if (rd <= rs) then branch to location addr
    i.e., goto PC + 4 + const*4 (i.e., PC = Updated PC + offset)
  - **Example :**
    bleu $s1, $s2, loop ; if ($s1 <= $s2) goto target location loop

Assembler Pseudoinstructions

- **beqz** (branch on equal to zero)
  - **Instruction Mnemonic :**
    beqz rd, addr ;where rd is a register,
    ;addr is the label of the target location
  - **Meaning :**
    if (rd == 0) then branch to location addr
    i.e., goto PC + 4 + const*4 (i.e., PC = Updated PC + offset)
  - **Example :**
    beqz $s1, up ; if ($s1 == 0) goto target location up

- **bnez** (branch on not equal to zero)
  - **Instruction Mnemonic :**
    bnez rd, rs, addr ;where rd is a register,
    ;addr is the label of the target location
  - **Meaning :**
    if (rd != 0) then branch to location addr
    i.e., goto PC + 4 + const*4 (i.e., PC = Updated PC + offset)
  - **Example :**
    bnez $s1, loop ; if ($s1 != 0) goto target location loop

Assembler Pseudoinstructions

- ❑ **mul (multiply registers signed without overflow)**
- **Instruction Mnemonic :**
  mul rd, rs, rt ;where rs, rt, rd are registers
- **Meaning :**
  rd = rs * rt ; 32-bit signed product in rd,
  ; overflow undetected
- **Example :**
  mul $s1, $s2,$s3 ; $s1 ← $s2 * $s3

- ❑ **mulo (multiply registers signed with overflow)**
- **Instruction Mnemonic :**
  mulo rd, rs, rt ;where rs, rt, rd are registers
- **Meaning :**
  rd = rs * rt ; 32-bit signed product in rd
  ; overflow detected
- **Example :**
  mulo $s1, $s2,$s3 ; $s1 ← $s2 * $s3

Assembler Pseudoinstructions

- ❑ **mulou (multiply registers unsigned with overflow)**
- **Instruction Mnemonic :**
  mulou rd, rs, rt ;where rs, rt, rd are registers
- **Meaning :**
  rd = rs * rt ; 32-bit unsigned product in rd
  ; overflow detected
- **Example :**
  mulou $s1, $s2,$s3 ; $s1 ← $s2 * $s3

Assembler Pseudoinstructions

- **div (signed divide registers with overflow)**
  - **Instruction Mnemonic :**
    div rd, rs, rt ;where rs, rt, rd are registers
  - **Meaning :**
    rd = rs / rt ; signed quotient in rd
  - **Example :**
    div $s1,$s2, $s3 ; $s1 ← $s2 / $s3

- **divu (unsigned divide registers without overflow)**
  - **Instruction Mnemonic :**
    divu rd, rs, rt ;where rs, rt, rd are registers
  - **Meaning :**
    rd = rs / rt ; unsigned quotient in rd
  - **Example :**
    divu $s1,$s2, $s3 ; $s1 ← $s2 / $s3

 # Example : Read N1 & N2 from keyboard, multiply N1 & N2 and display the product on the console

```
            .text
            .globl   main
main:
            la $a0,prompt1          # print prompt1 on terminal
            li $v0,4
            syscall
            li $v0,5                # syscall 5 reads an integer
            syscall
            move $t1,$v0            # $t1 holds first number N1
            la $a0,prompt2          # print prompt2 on terminal
            li $v0,4
            syscall
            li $v0,5                # syscall 5 reads an integer
            syscall
            move $t2,$v0            # $t2 holds second number N2
```

```
            mul  $t0, $t1,$t2
            sw  $t0, PRD32
            la $a0,promptr          # print promptr on terminal
            li $v0,4
            syscall

            move $a0,$t0            # display result
            li $v0,1
            syscall
            la $a0,endl             # print newline on terminal
            li $v0,4
            syscall
            li $v0,10
            syscall                 # exit

            .data
PRD32:      .space 4
prompt1:    .asciiz "Enter first number N1 = "
prompt2:    .asciiz "Enter second number N2 = "
promptr:    .asciiz "Product = “
endl:       .asciiz  “\n”
```