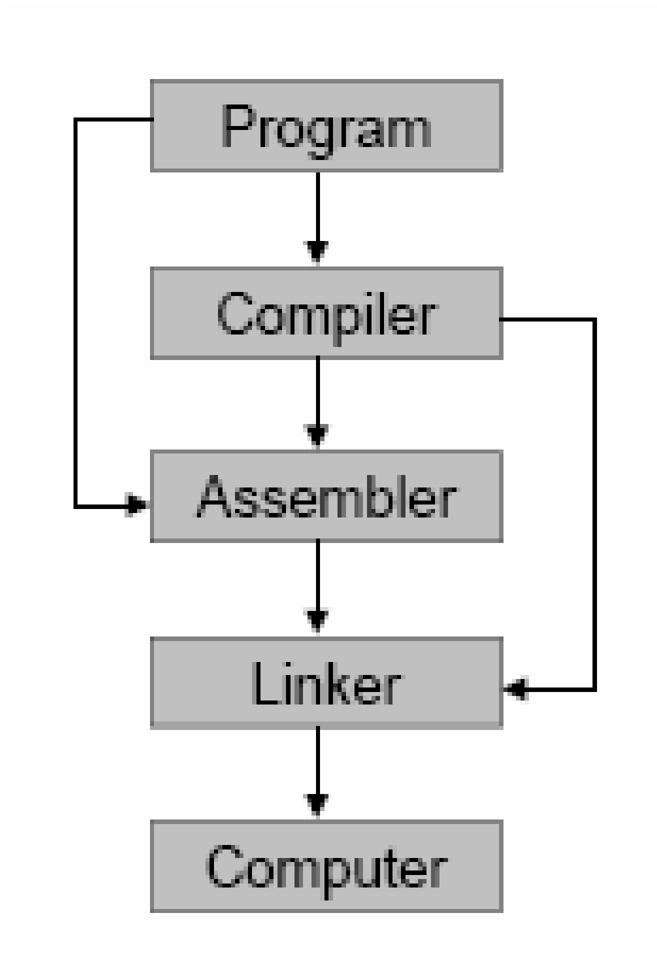
Tutorial 8:
Assembly Language

- Overview of assembler
- Writing an assembly program
- Assembly language instructions
- Demo of debugging

Hierarchy



- Program written in high-level language

- Compiler converts program to machine code

- Assembler converts assembly code to machine code

- Linker combines files from one project into a single executable file

- Computer executes the machine code

Why do we need assemblers?

```
00100111101111011111111111100000
10101111101111110000000000010100
10101111101001000000000000100000
10101111101001010000000000100100
10101111101000000000000000011000
10101111101000000000000000011100
10001111101011100000000000011100
10001111101110000000000000011000
00000001110011100000000000011001
00100101110010000000000000000001
00101001000000010000000001100101
10101111101010000000000000011100
00000000000000000111100000010010
00000011000011111100100000100001
00010100001000001111111111110111
10101111101110010000000000011000
00111100000001000001000000000000
10001111101001010000000000011000
00001100000100000000000011101100
00100100100001000000010000110000
10001111101111110000000000010100
00100111101111010000000000100000
00000011111000000000000000001000
00000000000000000001000000100001
```

FIGURE A.2 MIPS machine language code for a routine to compute and print the sum of the squares of integers between 0 and 100.

(From Patterson & Hennessy)

```
addiu  $29, $29, -32
sw     $31, 20($29)
sw     $4, 32($29)
sw     $5, 36($29)
sw     $0, 24($29)
sw     $0, 28($29)
lw     $14, 28($29)
lw     $24, 24($29)
multu  $14, $14
addiu  $8, $14, 1
slti   $1, $8, 101
sw     $8, 28($29)
mflo   $15
addu   $25, $24, $15
bne    $1, $0, -9
sw     $25, 24($29)
lui    $4, 4096
lw     $5, 24($29)
jal    1048812
addiu  $4, $4, 1072
lw     $31, 20($29)
addiu  $29, $29, 32
jr     $31
move   $2, $0
```

FIGURE A.1.3 The same routine written in assembly language.

(From Patterson & Hennessy)

SPIM

Can download at:

`http://www.cs.wisc.edu/~larus/spim.html`

Can download documentation at same site, including Appendix A of textbook, which is a reference for SPIM

SPIM

- Code starts with the **`.text`** directive
- **`.globl main`** directive: says "main" is global; so can be used from other files
- **`main`** label
  - gives the start of your program
  - your main program calls your procedures
- Data starts with **`.data`** directive
- **`#`** used to comment out rest of line
  - comments are very important!
  - should comment every line of code, if you want to understand it later...

Labels

- Can start any line with a label
- The label is then used elsewhere in the program, where it will contain the memory address of that line.
- For example, you can use labels to access data:

Data with Labels

```
          .data

Label1:   .word 42, 36       #32-bit quantities

Label2:   .byte 12, 7        #8-bit quantities

Label3:   .asciiz "hi\n"     #NULL-terminated
                             #ASCII

          .align 2           #Align to next 2^n byte

Label4:   .word 12           #32-bit quantity
```

Data with labels (continued)

So, if data segment starts at 0x1000, we get:

```
Label1:     .word 42, 36       #32-bit quantities
Label2:     .byte 12, 7        #8-bit quantities
```

```
Label1:     0x1000     0x2a
            0x1001     0x00
            0x1002     0x00
            0x1003     0x00
            0x1004     0x24
            0x1005     0x00
            0x1006     0x00
            0x1007     0x00
Label2:     0x1008     0x0c
            0x1009     0x07
```

Data with labels (continued)

```
Label3:    .asciiz "hi\n"    #NULL-terminated
                             #ASCII
           .align 2          #Align
Label4:    .word 12          #32-bit quantity
```

```
Label3:    0x100a    0x68
           0x100b    0x69
           0x100c    0x0a
           0x100d    0x00
           0x100e    (skipped over)
           0x100f    (skipped over)
Label4:    0x1010    0x0c
           0x1011    0x00
           0x1012    0x00
           0x1013    0x00
```

Labels with Code

- Use labels in your program for entry points for procedures, branches, and loops
- For each procedure, first label is usually name of procedure
- Can then have labels with the procedure name and a number, counting by 10's

Labels with code - example

```
count:    li   $15, 12      #start count at 12
count10:  move $4, $15      #move count to $4
          li   $2, 1        #code for print int
          syscall           #print count

          addi $15, -1      #decrement count
          bne  $15, $0, count10
                            #if not zero, keep
                            #going
count20:  jr   $31          #done.  So, return!
```

Register Conventions

- R0: zero constant
- R1: “at” reserved for assembler
- R2: “v0” expression evaluation
- R3: “v1” function results
- R4-R7: “a0..a3” arguments
- R8-R15, R24-R25: “t0..t7, t8-t9” temporary registers
- R16-R23: “s0..s7” secure (protected) registers
- R26-R27: “k0-k1” reserved for OS kernel
- R28: “gp” pointer to global area
- R29: “sp” stack pointer
- R30: “fp” frame pointer
- R31: “ra” Return Address

Some I/O Functions (Syscall)

Code	Service	Arguments	Notes
1	Print int	$4	
4	Print string	$4	(address)
5	Read int		Integer in $2
8	Read string	$4=buffer $5=length	
10	exit		

Syscall Print

- Printing something:
  - Load information (address for string, value for integer) into argument register ($4):
    - li $4, 42
  - Load desired system call code into $2
    - li $2, 1
  - Execute system call
    - syscall

Syscall Read

- Reading Something:
  - Load desired system call code into $2
    - li $2, 5
  - Execute system call
    - syscall
  - Value is now stored in $2
  - Should be moved from there before next syscall

Syscall Exit

- Exiting
  - Load desired system call code into $2
    - li $2, 10
  - Execute system call
    - syscall

Arithmetic and Logic

($8, $9, and $10 could be any register, e.g. $15)

- **`add $8, $9, $10`**
  - put sum of $9 and $10 into $8
- **`sub $8, $9, $10`**
  - subtract $10 from $9 and put result in $8
- **`and $8, $9, $10`**
  - "and" $9 with $10 and put result in $8
- **`or $8, $9, $10`**
  - "or" $9 with $10 and put result in $8

Arithmetic and Logic

Immediate versions (using a constant, N)

- **`addi $8, $9, N`**
  - put sum of $9 and N into $8
- **`subi $8, $9, N`**
  - subtract N from $9 and put result in $8
- **`andi $8, $9, N`**
  - "and" $9 with N and put result in $8
- **`ori $8, $9, N`**
  - "or" $9 with N and put result in $8

(Note: N can only have 16 bits max)

Arithmetic and Logic

- **`sll $8, $9, N`**
  - Set $8 to $9, shifted left by N bits (shift left logical)
- **`srl $8, $9, N`**
  - Set $8 to $9, shifted right by N bits (shift right logical)
- **`negu $8, $9`**
  - Set $8 to negative $9 (negate, no overflow)

Some Branch instructions

- **`b label`**
  - branch to label
- **`beq $9, $10, label`**
  - If $9 equals $10, branch to label (Branch if equal)
- **`bne $9, $10, Label`**
  - If $9 and $10 different, branch to label (Branch if not equal)
- **`blt $9, $10, Label`**
  - Branch if $9 less than $10 (Branch if less than)
- **`bgt $9, $10, Label`**
  - Branch if $10 greater than $10 (Branch if greater than)

Jump Instructions

Used to jump to a new location

- j label
  - Jump to instruction at label
- jal label
  - Jump to instruction at label, saving return address in register $31
- jr Register
  - Jump to the address given in register (usually $31)

Some comparison instructions

- **`slt $8, $9, $10`**
  - Set $8 to 1 if register $9 is less than $10, and to 0 otherwise (set if less than)
- **`sgt $8, $9, $10`**
  - Set $8 to 1 if register $9 is greater than $10, and to 0 otherwise (set if greater than)

Load/Store

- **`li $7, N`**
  - Load number N into register $7 (load immediate)
- **`la $8, Address`**
  - Load memory address into $8 (load address)
- **`lw $9, 0($8)`**
  - Load 32-bit word at memory address given by register $8 into register $9 (load word)
- **`move $7, $9`**
  - Move contents of register $9 into $7
- **`sw $9, 0($8)`**
  - Store contents of register $9 at the memory location given by register $8.

Indirect Addressing

- Often used with loads and stores to memory
  - lw $15, 4($sp) loads the word at memory address $sp + 4 into register $15
  - sw $15, 4($sp) loads the word in register $15 into memory address $sp + 4
- Number outside bracket is a constant, and is added to contents of register inside bracket to get a memory location

Subroutine Calls

- When your subroutine is called, it needs to save any registers that it uses (with the exception of arguments that will be returned)
- When your subroutine finishes, it must restore the registers it used
- Registers are stored on the stack

Saving Registers – example

```
s1:  addi $29, $29, -32  # Allocate space on
                       # stack for 8 registers
     sw $2, 0($29)     # Save $2 onto stack
     sw $4, 4($29)     # Save $4 onto stack
     sw $5, 8($29)     # Save $5 onto stack
     sw $11, 12($29)# Save $11 onto stack
     sw $15, 16($29)# Save $15 onto stack
     sw $16, 20($29)# Save $16 onto stack
     sw $17, 24($29)# Save $17 onto stack
     sw $18, 28($29)# Save $18 onto stack
```

(subroutine can then use these registers)

Restoring Registers – example

(At the end of the subroutine, must restore the registers!)

```
lw $2, 0($29)  # Load $2 from stack
lw $4, 4($29)  # Load $4 from stack
lw $5, 8($29)  # Load $5 from stack
lw $11, 12($29)# Load $11 from stack
lw $15, 16($29)# Load $15 from stack
lw $16, 20($29)# Load $16 from stack
lw $17, 24($29)# Load $17 from stack
lw $18, 28($29)# Load $18 from stack
addi $29, $29, 32  # Restore stack pointer
```

Demo of SPIM

- Edit a simple program, count.s
  - use any text editor, e.g. LCC
- Load the program into SPIM
- Run the program
- Simple debugging