

CS39001 COMPUTER ORGANIZATION AND ARCHITECTURE LAB

MIPS (MIPS)^R Basics

CSE, IIT Kharagpur

The MIPS Architecture

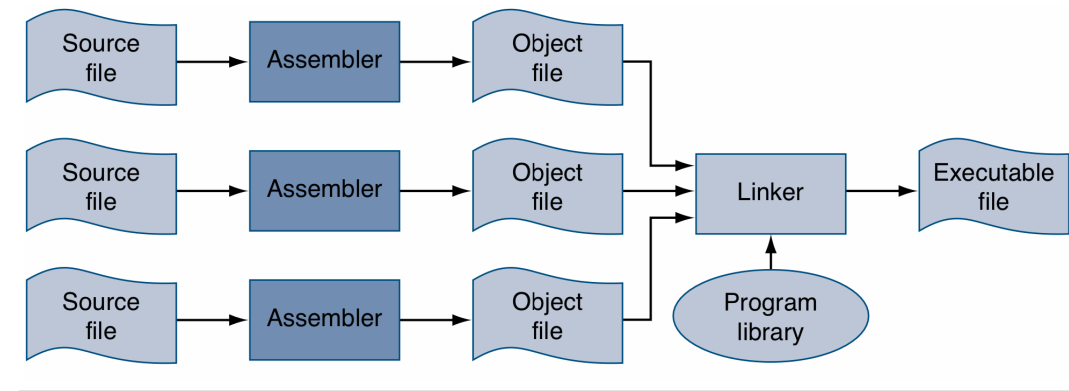
- Initiated at Stanford.
- MIPS is dominant in embedded applications. The first processor was R2000, which was developed by **MIPS Computer Systems, Inc.**
- The MIPS Instruction Set Architecture has evolved from MIPS 1 to MIPS V. (Now we have RISC-V)
- In the late 1990s, MIPS was based on two basic architectures:
 - MIPS32 for 32 bit architectures
 - MIPS64 for 64 bit architectures

SPIM

- SPIM is a simulator for the MIPS R2000/R3000 implementations.
- These are 32 bit ISA.
- *pages.cs.wisc.edu/~larus/spim.html*
- MIPS is based on load and store architecture
- From now on, we shall refer to MIPS32 as MIPS
- We shall use the QTSPIM simulator

Assembly Language

- Computer instructions can be represented as sequences of bits.
- Machine Language is the lowest level of representing a program.
- It is understood by a machine, as the sequence of 0s and 1s in the machine language instructions represent atomic operations.
- However, understanding for a human is difficult.
- So, assembly language is a more readable language (and is hence more high level), but usually is very closely related with the machine language.
- The assembly language is converted to the machine language by an assembler.
- SPIM is an assembler for the MIPS32 ISA.



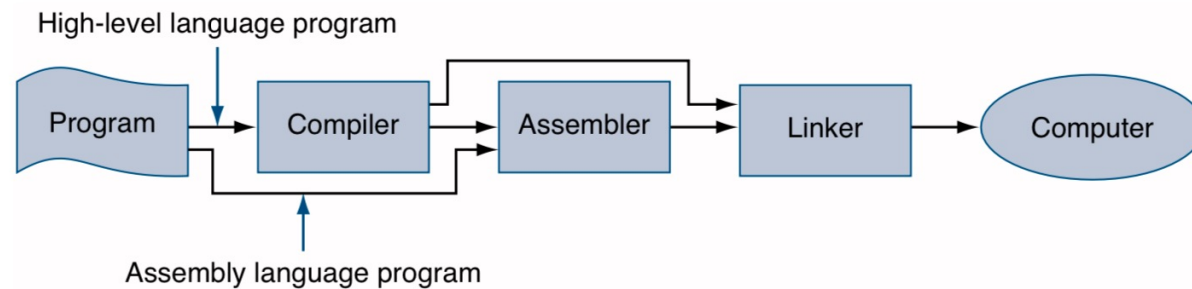
Assembly Language

```
#include <stdio.h>

int
main (int argc, char *argv[])
{
    int i;
    int sum = 0;

    for (i = 0; i <= 100; i = i + 1) sum = sum + i * i;
    printf ("The sum from 0 .. 100 is %d\n", sum);
}
```

FIGURE A.1.5 The routine written in the C programming language.



```
.text
.align    2
.globl    main

main:
    subu    $sp, $sp, 32
    sw      $ra, 20($sp)
    sd      $a0, 32($sp)
    sw      $0, 24($sp)
    sw      $0, 28($sp)

loop:
    lw      $t6, 28($sp)
    mul     $t7, $t6, $t6
    lw      $t8, 24($sp)
    addu    $t9, $t8, $t7
    sw      $t9, 24($sp)
    addu    $t0, $t6, 1
    sw      $t0, 28($sp)
    ble     $t0, 100, loop
    la      $a0, str
    lw      $a1, 24($sp)
    jal     printf
    move    $v0, $0
    lw      $ra, 20($sp)
    addu    $sp, $sp, 32
    jr      $ra

.data
.align    0

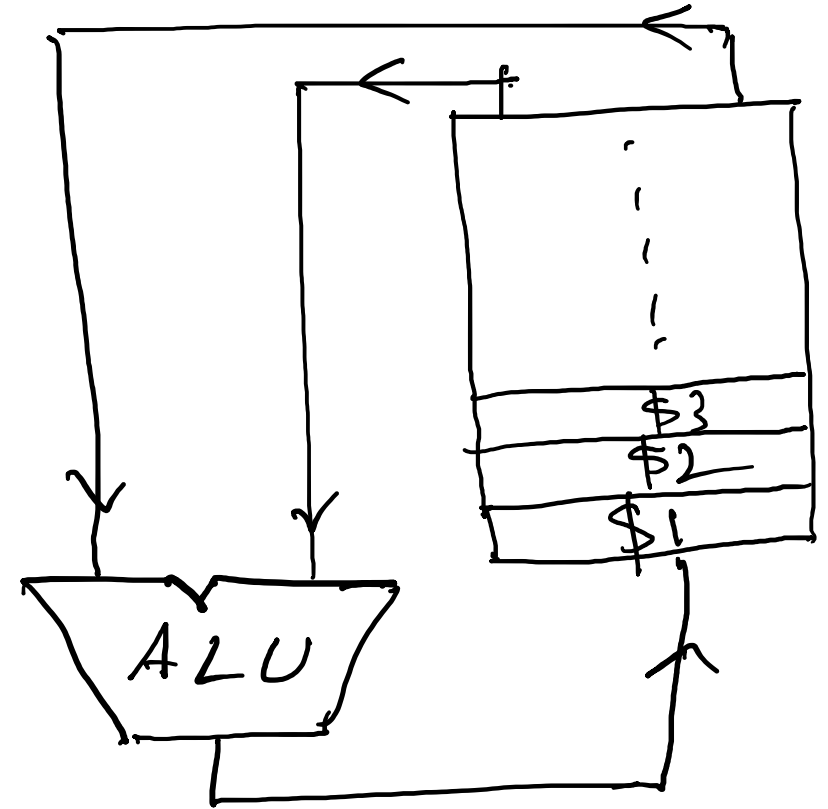
str:
.asciiz    "The sum from 0 .. 100 is %d\n"
```

Example

- The MIPS machine language instruction for adding the contents of registers 20 and 17 and placing the result in register 16 is the integer 0x02918020.
- The MIPS assembly instruction for the same operation is: add \$16, \$20, \$17, which is much more readable.
- If you want to change, say the destination register to \$12, it is easy!

Registers in MIPS

- Has 32 general purpose registers:
 - \$0, \$1, ..., \$31
 - \$0 and \$31 are reserved for specific purposes
- A Program Counter (PC)
- 2 Special Purpose registers
- All the registers are 32 bit wide.



MIPS General Purpose Registers

| Symbolic Name | Number | Usage |
|---------------|-----------------|----------------------------------|
| zero | 0 | Constant 0. |
| at | 1 | Reserved for the assembler. |
| v0 - v1 | 2 - 3 | Result Registers. |
| a0 - a3 | 4 - 7 | Argument Registers 1 \dots 4. |
| t0 - t9 | 8 - 15, 24 - 25 | Temporary Registers 0 \dots 9. |
| s0 - s7 | 16 - 23 | Saved Registers 0 \dots 7. |
| k0 - k1 | 26 - 27 | Kernel Registers 0 \dots 1. |
| gp | 28 | Global Data Pointer. |
| sp | 29 | Stack Pointer. |
| fp | 30 | Frame Pointer. |
| ra | 31 | Return Address. |

Register Usage Convention

- With time, there are some conventions developed for the usage of registers.
 - these are not hardware requirements.
- Registers \$v0 and \$v1 are used to return results from a procedure.
- \$a0 to \$a3 are used to pass the first four arguments.
- The remaining arguments are passed via the stack.
 - these registers are not preserved across procedure calls
 - the called procedures can freely modify the contents of these registers

Register Usage Convention

- \$t0 to \$t9 are temporary registers that need not be preserved across a procedure call.
 - assumed to be saved by the caller.
- \$s0 to \$s7 are called callee-saved registers that should be preserved across procedure calls.

Register Usage Convention

- Register \$sp is the stack pointer
- The register \$fp is the frame pointer
- The register \$ra is used to store the return address.
- The register \$gp points to the memory area that holds constants and global variables.
- The register \$at is reserved for the assembler:
 - the assembler uses it to convert the pseudo-instructions to processor instructions

Addressing Modes

- MIPS is a load/store architecture.
- There is a single memory addressing mode:

$\text{disp}(\text{Rx}),$

where disp is a signed, 16-bit immediate value which is the displacement.

The actual address is computed as:

$\text{disp} + \text{contents of base register Rx}$

Example

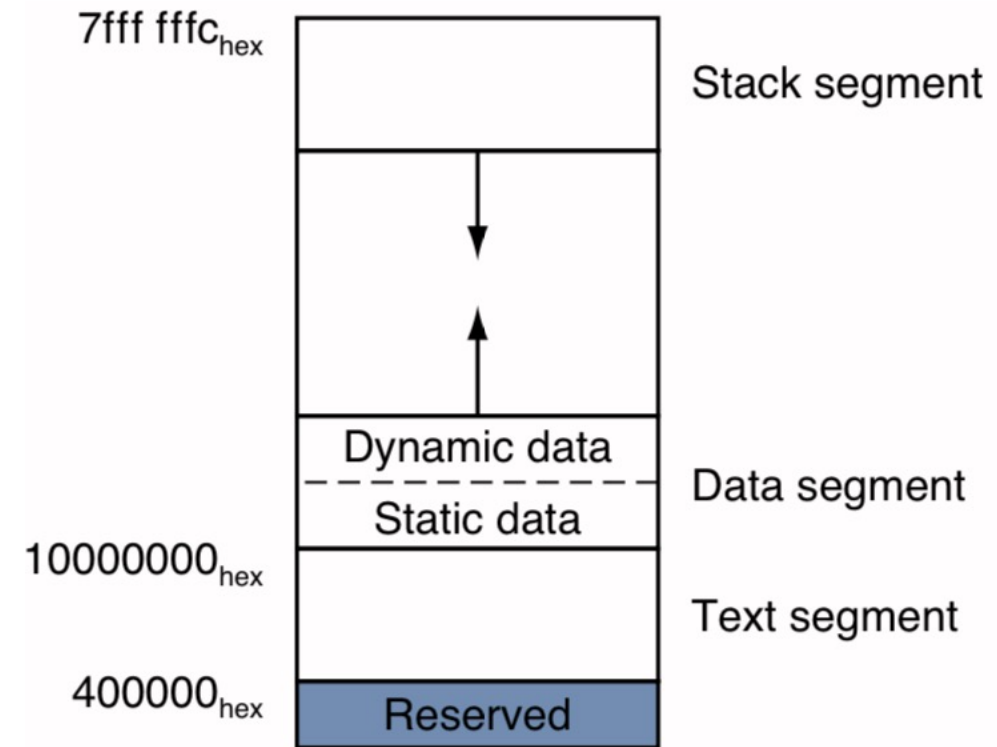
- If \$a0 points to an array that contains 4 byte elements, the first element is specified as:

0(\$a0)

- The next element is specified as 4(\$a0)

Memory Usage

- MIPS has a conventional memory layout.
- A Program's address space has three parts: code, data and stack.
- The text segment, which stores instructions, is placed at the bottom of the user address space at 0x4000000.
- The data segment is placed above the text segment and starts at 0x10000000: divided into static and dynamic areas.
 - the dynamic areas grows as memory is allocated to dynamic data structures.
- The stack segment is placed at the end of the user address space at 0x7FFFFFFF.
 - it grows downward towards the lower memory address



Assembly Language Statements

- Three types:
 - Executables: consists of an opcode for short. Causes the assemblers to generate machine language instructions.
 - Pseudo instructions: Not directly supported by the processor. Assembler supports them by generating one or more processor instructions.
 - assembler directives: Not executables; do not generate any machine language instructions.

Format

[label:] mnemonic [operands] [#comments]

label used to associate them with the address of a memory location

Mnemonic: opcode of instructions, like add, sub.

Operands: The data that is to be manipulated by the statements.

Ex: add \$t0,\$t1,\$t2

System Calls

- SPIM supports I/O through system call (syscall) instruction.
- Eight of these calls are for I/O of four basic data types: string, integer, float, double.
 - character I/O missing, have to tackle using that for strings.
- To invoke a service, the system call service code should be placed in \$a0 and \$a1 registers (use \$f12 for floating point values).
- Any value returned by a system call is placed in \$v0 (\$f0 are floating point values).

System Calls

| Service | System call Code (in \$v0) | Arguments | Result |
|--------------|----------------------------------|---|--|
| print_int | 1 | \$a0=integer | Integer in \$v0 Float in \$f0 Double in \$f0 |
| print_float | 2 | \$f12=float | |
| Print_double | 3 | \$f12=double | |
| Print_string | 4 | \$a0=string address | |
| Read_int | 5 | | |
| Read_float | 6 | | |
| Read_double | 7 | | |
| Read_string | 8 | \$a0=buffer address \$a1=buffer size | |
| Sbrk | 9 | | Address in \$v0 |
| exit | 10 | | |

Program Template

```
#####data segment#####  
    .data  
prompt:  
    .ascii "Enter your name" #prompt is a string variable  
in_name:  
    .space 31 #allocate n bytes of uninitialized space (assembler directive)  
#####Code Segment#####  
    .text  
.globl main  
main:  
    la $a0, prompt #Prints to prompt the user to enter  
    li $v0, 4  
    syscall  
    la $a0,in_name #Reads string in_name  
    li $a1,31 #limits input string length to 30 characters. String is null-terminated  
    li $v0,8  
    syscall
```

Where is a Program???

- Write a SPIM program to read two integers, and add them.

Program 1

```
#####Data Segment#####
```

```
prompt:
```

```
    .ascii "Enter two numbers: "
```

```
sum_msg:
```

```
    .ascii "The sum is: "
```

```
newline:
```

```
    .ascii "\n"
```

```
#####Code Segment#####
```

```
    .text
```

```
    .globl main
```

Program 1

main:

| | |
|----------------|--|
| la \$a0,prompt | #loads \$a0 with the address of “prompt” |
| li \$v0,4 | #prints the string |
| syscall | |

| | |
|-----------------|--------------------------|
| li \$v0,5 | #reads first integer |
| syscall | |
| move \$t0, \$v0 | #result returned in \$v0 |

| | |
|-----------------|--------------------------|
| li \$v0, 5 | #reads second integer |
| syscall | |
| move \$t1, \$v0 | #result returned in \$v0 |

Program 1

```
addu $t0, $t0, $t1
```

```
la $a0,sum_msg
```

```
li $v0,4
```

```
syscall
```

```
move $a0,$t0
```

```
li $v0,1      #prints the integer sum
```

```
syscall
```

```
li $v0,10     #exit
```

```
syscall
```

Program 2

- Write a SPIM program to compute the sum of individual digits of a number with maximum 10 digits.
 - Input: 12345
 - Sum: 15

Data Segment

```
#####Data Segment#####
```

```
.data
```

```
number_prompt:
```

```
    .ascii "Please enter a number (<11 digits): "
```

```
out_msg:
```

```
    .ascii "\n The sum of individual digits is: "
```

```
newline:
```

```
    .ascii "\n"
```

```
number:
```

```
    .space 11
```

Code Segment

```
#####Code Segment#####
```

```
.text
```

```
.globl main
```

```
main:
```

```
la $a0,number_prompt
```

```
li $v0,4
```

```
syscall
```

```
la $a0,number
```

```
li $a1,11
```

```
li $v0,8
```

```
syscall
```

```
la $a0,out_msg
```

```
li $v0,4
```

```
syscall
```

Code Segment

```
la $t0, number      #pointer to number
li $t2,0             #initialize sum=0
```

Loop:

```
lb $t1,($t0)         #load a byte from the memory address pointer by $t0
beq $t1,0xA,exit_loop #Check if it is linefeed
beqz $t1,exit_loop    #Check if it is a null character
and $t1,$t1,0x0F      #Mask the upper four bits, to obtain the decimal
value
addu $t2,$t2,$t1      #add and accumulate
addu $t0,$t0,1         #Move the pointer by one byte
b loop
```

Code Segment

exit_loop:

move \$a0,\$t2 #output sum

li \$v0,1

syscall

la \$a0,newline #output newline

li \$v0,4

syscall

exit:

li \$v0,10

syscall

Assignments

1. Modify the adddigits program so that it accepts a string which consists of both digits and non-digit characters. The program should however print the sum of only the digits, while ignoring the non-digit characters.

For example: if the string input is, 12ABC?3,
the sum is 6

Assignments

2. Write a SPIM program to encode the digits as shown below:

input digit: 0 1 2 3 4 5 6 7 8 9

output digit: 4 6 9 5 0 3 1 8 7 2

Your program should accept a string consisting of both digits and non-digits. The encoded string should be displayed in which only the digits are affected. Ask the user, whether he/she wants to terminate the program. If the response is “y” or “Y”, terminate the program; otherwise request for another input string.

Note that the above encoding has the property that if the encoding is applied twice, one gets the original string. Use this property to verify the program for ten arbitrary inputs.

ACK

- Many slides are from Prof. Debdeep's deck