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

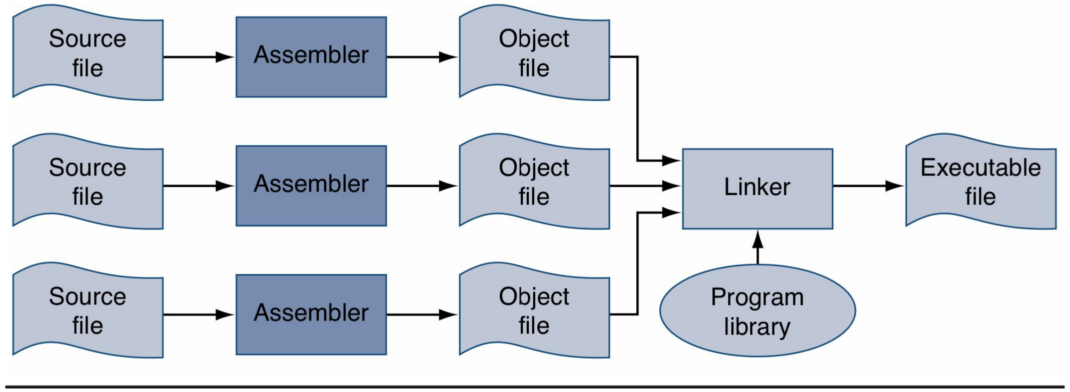
Acknowledgement: Some slides are from Prof. Debdeep's deck

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.



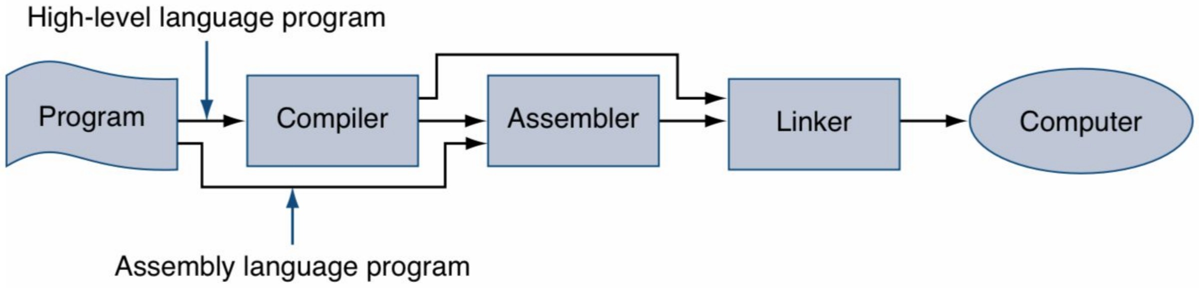
Assembly Language

```c
#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 ··· 4.
t0 - t9	8 - 15, 24 - 25	Temporary Registers 0 ··· 9.
s0 - s7	16 - 23	Saved Registers 0 ··· 7.
k0 - k1	26 - 27	Kernel Registers 0 ··· 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 regiesters 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:

disp(Rx),

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

The actual address is computed as:

disp + contents of base register Rx

Example

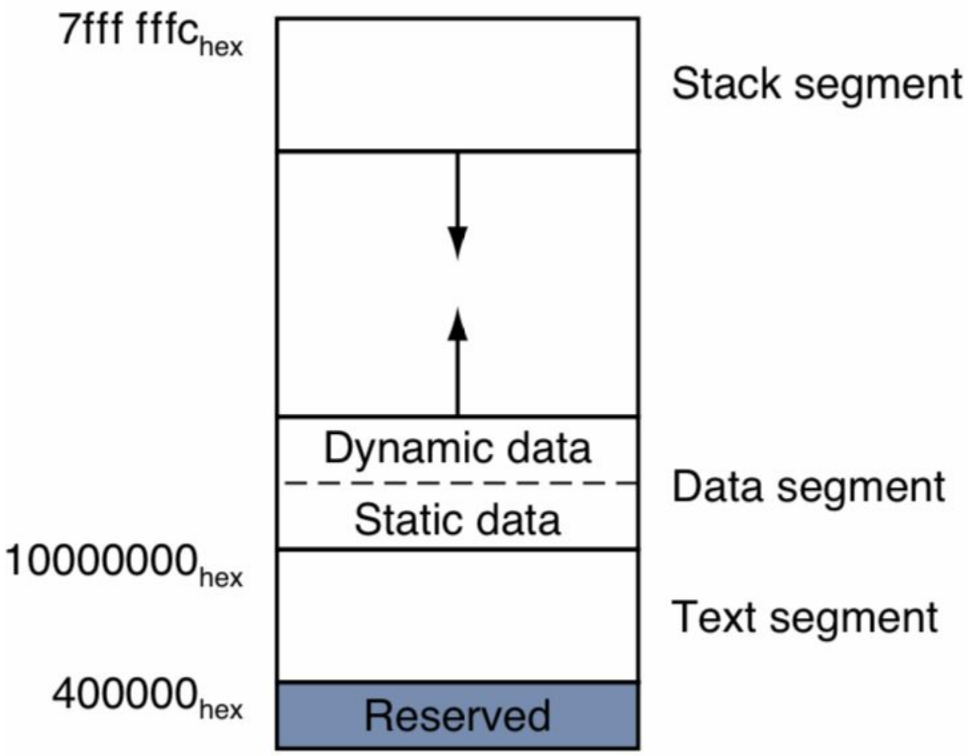
- If $a0 points to an array that contains 4 byte elements, the first element is specified as:

  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	
print_float	2	$f12=float	
Print_double	3	$f12=double	
Print_string	4	$a0=string address	
Read_int	5		Integer in $v0
Read_float	6		Float in $f0
Read_double	7		Double in $f0
Read_string	8	$a0=buffer address $a1=buffer size	
Sbrk	9		Address in $v0
exit	10		

Program Template

```
##########data segment##############
     .data
prompt:
      .asciiz “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:
	.asciiz  "Enter two numbers: "
sum_msg:
	.asciiz "The sum is: "
newline:
	.asciiz "\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:
   .asciiz "Please enter a number (<11 digits): "
out_msg:
   .asciiz "\n The sum of individual digits is: "
newline:
   .asciiz "\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

Assigments

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.

Data types and Registers

Data types:

- Instructions are all 32 bits
- byte(8 bits), halfword (2 bytes), word (4 bytes)
- a character requires 1 byte of storage
- an integer requires 1 word (4 bytes) of storage

Literals:

- numbers entered as is. e.g. 4
- characters enclosed in single quotes. e.g. 'b'
- strings enclosed in double quotes. e.g. "A string"

1. Reg Lo and Hi used for storing MUL/DIV results
2. Content accessed with mflo/mfhi instrs

Register Number	Alternative Name	Description
0	zero	the value 0
1	$at	(**a**ssembler **t**emporary) reserved by the assembler
2-3	$v0 - $v1	(**v**alues) from expression evaluation and function results
4-7	$a0 - $a3	(**a**rguments) First four parameters for subroutine. Not preserved across procedure calls
8-15	$t0 - $t7	(**t**emporaries) Caller saved if needed. Subroutines can use w/out saving. Not preserved across procedure calls
16-23	$s0 - $s7	(**s**aved values) - Callee saved. A subroutine using one of these must save original and restore it before exiting. Preserved across procedure calls
24-25	$t8 - $t9	(**t**emporaries) Caller saved if needed. Subroutines can use w/out saving. These are in addition to $t0 - $t7 above. Not preserved across procedure calls.
26-27	$k0 - $k1	reserved for use by the interrupt/trap handler
28	$gp	**g**lobal **p**ointer. Points to the middle of the 64K block of memory in the static data segment.
29	$sp	**s**tack **p**ointer Points to last location on the stack.
30	$s8/$fp	**s**aved value / **f**rame **p**ointer Preserved across procedure calls
31	$ra	**r**eturn **a**ddress

Data Declarations

format for declarations:

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

- create storage for variable of specified type with given name and specified value
- value(s) usually gives initial value(s); for storage type .space, gives number of spaces to be allocated

Note: labels always followed by colon (:)

```
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 which!
```

MIPS ISA: Arithmetic instructions

Instruction	Example	Meaning	Comments
add	`add $1,$2,$3`	\$1=\$2+\$3	
subtract	`sub $1,$2,$3`	\$1=\$2-\$3	
add immediate	`addi $1,$2,100`	\$1=\$2+100	"Immediate" means a constant number
add unsigned	`addu $1,$2,$3`	\$1=\$2+\$3	Values are treated as unsigned integers, not two's complement integers
subtract unsigned	`subu $1,$2,$3`	\$1=\$2-\$3	Values are treated as unsigned integers, not two's complement integers

MIPS ISA: Arithmetic instructions

add immediate unsigned	`addiu $1,$2,100`	\$1=\$2+100	Values are treated as unsigned integers, not two's complement integers
Multiply (without overflow)	`mul $1,$2,$3`	\$1=\$2*\$3	Result is only 32 bits!
Multiply	`mult $2,$3`	\$hi,\$low=\$2*\$3	Upper 32 bits stored in special register `hi` Lower 32 bits stored in special register `lo`
Divide	`div $2,$3`	\$hi,\$low=\$2/\$3	Remainder stored in special register `hi` Quotient stored in special register`lo`

Arithmetic ….

- most use 3 operands
- all operands are registers; no RAM or indirect addressing
- operand size is word (4 bytes)

```
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;   add as unsigned integers
subu    $t1,$t6,$t7     #  $t1 = $t6 + $t7;   subtract as unsigned integers

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

move    $t2,$t3 #  $t2 = $t3
```

MIPS ISA: Logical instructions

Instruction	Example	Meaning	Comments
and	`and $1,$2,$3`	\$1=\$2&\$3	Bitwise AND
or	`or $1,$2,$3`	\$1=\$2|\$3	Bitwise OR
and immediate	`andi $1,$2,100`	\$1=\$2&100	Bitwise AND with immediate value
or immediate	`or $1,$2,100`	\$1=\$2|100	Bitwise OR with immediate value
shift left logical	`sll $1,$2,10`	\$1=\$2<<10	Shift left by constant number of bits
shift right logical	`srl $1,$2,10`	\$1=\$2>>10	Shift right by constant number of bits

Data Transfer instructions

load word	`lw $1,100($2)`	$1=Memory[$2+100]	Copy from memory to register
store word	`sw $1,100($2)`	Memory[$2+100]=$1	Copy from register to memory
load upper immediate	`lui $1,100`	$1=100x2^16	Load constant into upper 16 bits. Lower 16 bits are set to zero.
load address	`la $1,label`	$1=Address of label	*Pseudo-instruction* (provided by assembler, not processor!) Loads computed address of label (not its contents) into register
load immediate	`li $1,100`	$1=100	*Pseudo-instruction* (provided by assembler, not processor!) Loads immediate value into register

Data Transfer instructions

move from hi	`mfhi $2`	\$2=hi	Copy from special register `hi` to general register
move from lo	`mflo $2`	\$2=lo	Copy from special register `lo` to general register
move	`move $1,$2`	\$1=\$2	*Pseudo-instruction* (provided by assembler, not processor!) Copy from register to register.

Data Transfer (immediate/offset based)

```
example:
        .data
var1:   .word   23              # declare storage for var1; initial value is 23

        .text
__start:
        lw      $t0, var1               # load contents of RAM location into register $t0:  $t0 = var1
        li      $t1, 5          #  $t1 = 5   ("load immediate")
        sw      $t1, var1               # store contents of register $t1 into RAM:  var1 = $t1
        done
```

```
example

                .data
array1:         .space  12              #  declare 12 bytes of storage to hold array of 3 integers
                .text
__start:        la      $t0, array1             #  load base address of array into register $t0
                li      $t1, 5          #  $t1 = 5   ("load immediate")
                sw $t1, ($t0)           #  first array element set to 5; indirect addressing
                li $t1, 13              #   $t1 = 13
                sw $t1, 4($t0)          #  second array element set to 13
                li $t1, -7              #   $t1 = -7
                sw $t1, 8($t0)          #  third array element set to -7
```

Flow control / Branching

branch on equal	`beq $1,$2,100`	if($1==$2) go to PC+4+100	Test if registers are equal
branch on not equal	`bne $1,$2,100`	if($1!=$2) go to PC+4+100	Test if registers are not equal
branch on greater than	`bgt $1,$2,100`	if($1>$2) go to PC+4+100	*Pseduo-instruction*
branch on greater than or equal	`bge $1,$2,100`	if($1>=$2) go to PC+4+100	*Pseduo-instruction*
branch on less than	`blt $1,$2,100`	if($1<$2) go to PC+4+100	*Pseduo-instruction*
branch on less than or equal	`ble $1,$2,100`	if($1<=$2) go to PC+4+100	*Pseduo-instruction*

```
b       target            #  unconditional branch to program label target
beq     $t0,$t1,target    #  branch to target if  $t0 = $t1
blt     $t0,$t1,target    #  branch to target if  $t0 < $t1
ble     $t0,$t1,target    #  branch to target if  $t0 <= $t1
bgt     $t0,$t1,target    #  branch to target if  $t0 > $t1
bge     $t0,$t1,target    #  branch to target if  $t0 >= $t1
bne     $t0,$t1,target    #  branch to target if  $t0 <> $t1
```

Comparison

Instruction	Example	Meaning	Comments
set on less than	`slt $1,$2,$3`	if($2<$3)$1=1; else $1=0	Test if less than. If true, set $1 to 1. Otherwise, set $1 to 0.
set on less than immediate	`slti $1,$2,100`	if($2<100)$1=1; else $1=0	Test if less than. If true, set $1 to 1. Otherwise, set $1 to 0.

Unconditional Jump

Instruction	Example	Meaning	Comments
jump	`j 1000`	go to address 1000	Jump to target address
jump register	`jr $1`	go to address stored in $1	For switch, procedure return
jump and link	`jal 1000`	$ra=PC+4; go to address 1000	Use when making procedure call. This saves the return address in $ra

```
j       target  #  unconditional jump to program label target
jr      $t3             #  jump to address contained in $t3 ("jump register")
```

Flow control : Subroutine Call

subroutine call: "jump and link" instruction

```
jal     sub_label       #  "jump and link"
```

- copy program counter (return address) to register $ra (return address register)
- jump to program statement at sub_label

subroutine return: "jump register" instruction

```
jr      $ra     #  "jump register"
```

- jump to return address in $ra (stored by jal instruction)

- ❖ With `jal’, the return address (PC+4) is automatically stored in `ra’
- ❖ Inside any active call, `jal’ always places return address in `ra’ register and hence will overwrite previous value
- ❖ If subroutine will call other subroutines, or is recursive, return address should be explicitly copied from $ra onto stack by caller routine
- ❖ More on this later

Assembler Directives

Directive	Result
`.word w1, ..., wn`	Store *n* 32-bit values in successive memory words
`.half h1, ..., hn`	Store *n* 16-bit values in successive memory words
`.byte b1, ..., bn`	Store *n* 8-bit values in successive memory words
`.ascii str`	Store the ASCII string str in memory. Strings are in double-quotes, i.e. "Computer Science"
`.asciiz str`	Store the ASCII string str in memory and null-terminate it Strings are in double-quotes, i.e. "Computer Science"
`.space n`	Leave an empty *n*-byte region of memory for later use
`.align n`	Align the next datum on a 2^n byte boundary. For example, .align 2 aligns the next value on a word boundary

Syscall examples

- **print_string** expects the address to start a null-terminated character string.
- The directive **.asciiz** creates a null-terminated character string.
- **read_int, read_float** and **read_double** read an entire line of input up to and including the newline character.
- **read_string** reads up to n-1 characters into a buffer and terminates the string with a null character.
- If fewer than n-1 characters are in the current line, it reads up to and including the newline and terminates the string with a null character.
- **sbrk** returns the address to a block of memory containing n additional bytes.
- **exit** stops a program from running.

```
e.g.   Print out integer value contained in register $t2

          li      $v0, 1               # load appropriate system call code into register $v0;
                                       # code for printing integer is 1
          move    $a0, $t2             # move integer to be printed into $a0:  $a0 = $t2
          syscall                      # call operating system to perform operation


e.g.   Read integer value, store in RAM location with label int_value (presumably declared in data section)

          li      $v0, 5               # load appropriate system call code into register $v0;
                                       # code for reading integer is 5
          syscall                      # call operating system to perform operation
          sw      $v0, int_value       # value read from keyboard returned in register $v0;
                                       # store this in desired location
```

Syscall examples

e.g. Print out string (useful for prompts)

```
                .data
string1         .asciiz "Print this.\n"       # declaration for string variable,
                                              # .asciiz directive makes string null terminated

                .text
main:           li      $v0, 4                # load appropriate system call code into register $v0;
                                              # code for printing string is 4
                la      $a0, string1          # load address of string to be printed into $a0
                syscall                       # call operating system to perform print operation
```

e.g. To indicate end of program, use **exit** system call; thus last lines of program should be:

```
                li      $v0, 10         # system call code for exit = 10
                syscall                       # call operating sys
```

Assignment 1 (7th Aug)

- Your program takes as input an integer x
- It computes e^x using Taylor series expansion

$$\sum_{n=0}^{\infty} \frac{x^n}{n!} = \frac{x^0}{0!} + \frac{x^1}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \cdots$$
$$= 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \frac{x^5}{120} + \cdots.$$

- DO not use any subroutine call
- Compute the series until the intermediate sum does not change in two successive iterations and print the final result