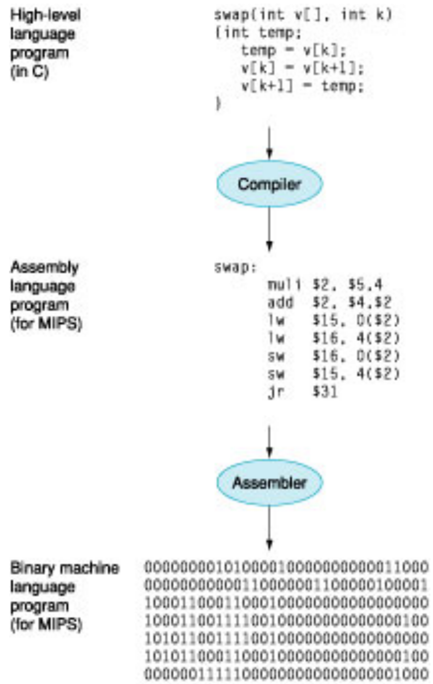
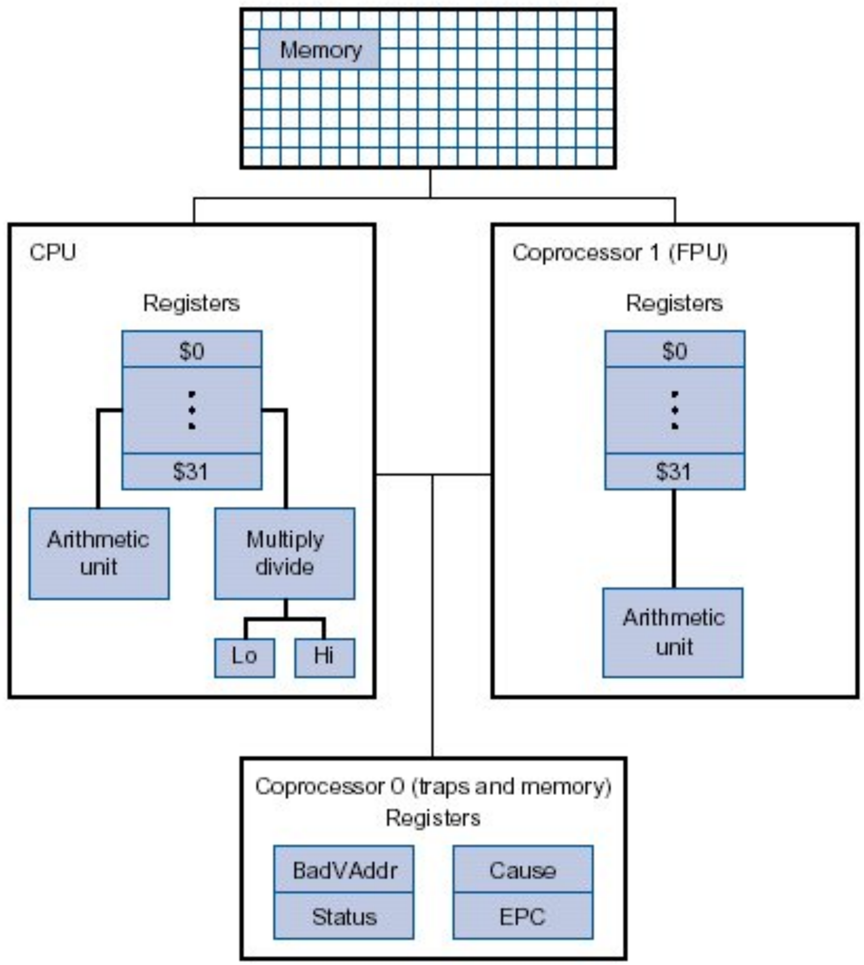
QtSPIM and MARS : MIPS Simulators


High-level language program (in C)
swap(int v[], int k)
{int temp;
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
}
Compiler
Assembly language program (for MIPS)
swap:
muli $2, $5,4
add $2, $4,$2
lw $15, 0($2)
lw $16, 4($2)
sw $16, 0($2)
sw $15, 4($2)
jr $31
Assembler
Binary machine language program (for MIPS)
00000000101000010000000000011000
00000000000110000001100000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000

Learning MIPS & SPIM

- MIPS assembly is a *low-level programming language*
- *The best way to learn any programming language is to write code*
- We will get you started by going through a few example programs and explaining the key concepts
- *Tip*: Start by copying existing programs and modifying them incrementally making sure you understand the behavior at each step
- *Tip*: The best way to understand and remember a construct or keyword is to *experiment with it in code*, not by reading about it

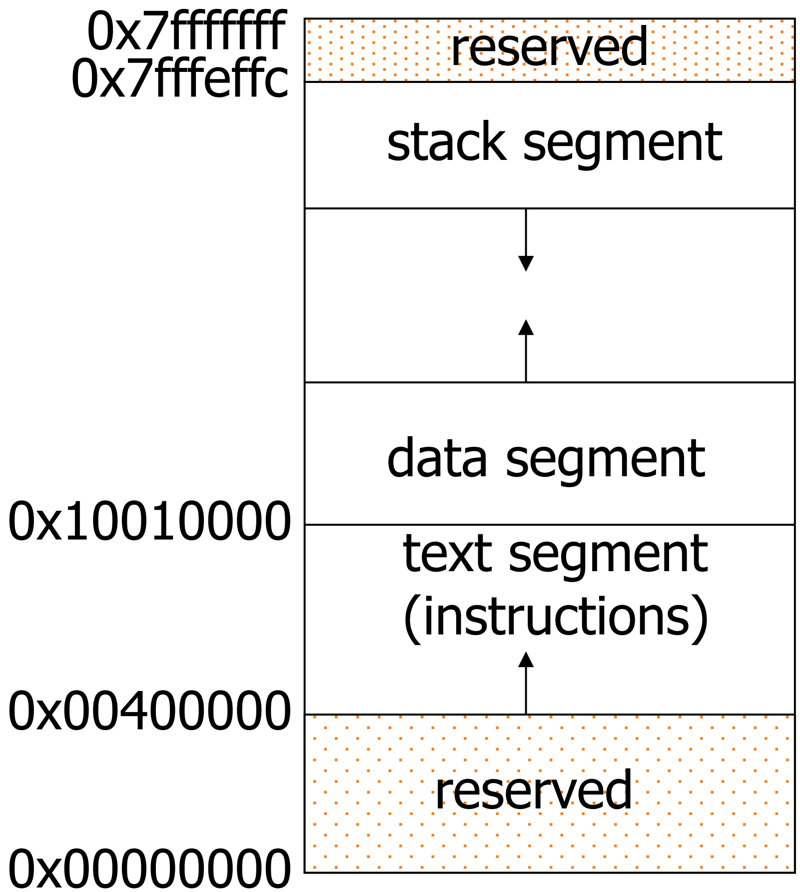

Memory
CPU
Registers
$0
$31
Arithmetic unit
Multiply divide
Lo
Hi
Coprocessor 1 (FPU)
Registers
$0
$31
Arithmetic unit
Coprocessor 0 (traps and memory)
Registers
BadVAddr
Cause
Status
EPC

MIPS Assembly Code Layout

- Typical Program Layout

```
        .text           #code section
        .globl main     #starting point: must be global
main:
        # user program code
        .data           #data section
         # user program data
```

MIPS Memory Usage as viewed in SPIM



MIPS Assembler Directives

- Top-level Directives:
  - **.text**
    - indicates that following items are stored in the user text segment, typically instructions
  - **.data**
    - indicates that following data items are stored in the data segment
  - **.globl** sym
    - declare that symbol sym is global and can be referenced from other files

MIPS Assembler Directives

- Common Data Definitions:
  - **.word** w1, …, wn
    - store n 32-bit quantities in successive memory words
  - **.half** h1, …, hn
    - store n 16-bit quantities in successive memory halfwords
  - **.byte** b1, …, bn
    - store n 8-bit quantities in successive memory bytes
  - **.ascii** str
    - store the string in memory but do not null-terminate it
      - strings are represented in double-quotes “str”
      - special characters, eg. \n, \t, follow C convention
  - **.asciiz** str
    - store the string in memory and null-terminate it

MIPS Assembler Directives

- Common Data Definitions:
  - **.float** f1, …, fn
    - store n floating point single precision numbers in successive memory locations
  - **.double** d1, …, dn
    - store n floating point double precision numbers in successive memory locations
  - **.space** n
    - reserves n successive bytes of space
  - **.align** n
    - align the next datum on a 2^n byte boundary.
    - For example, **.align 2** aligns next value on a word boundary.
    - **.align 0** turns off automatic alignment of **.half**, **.word**, etc. till next **.data** directive

MIPS: Software Conventions for Registers

0	zero	constant 0
1	at	reserved for assembler
2	v0	results from callee
3	v1	returned to caller
4	a0	arguments to callee
5	a1	from caller: caller saves
6	a2	
7	a3	
8	t0	temporary
. . .		
15	t7	

16	s0	callee saves
. . .		
23	s7	
24	t8	temporary (cont'd)
25	t9	
26	k0	reserved for OS kernel
27	k1	
28	gp	pointer to global area
29	sp	stack pointer
30	fp	frame pointer
31	ra	return Address
		caller saves

Pseudoinstructions

- **Pseudoinstructions** do not correspond to real **MIPS** instructions.
- Instead, the assembler, would translate **pseudoinstructions** to real instructions (one on more instructions).
- **Pseudoinstructions** not only make it easier to program, it can also add clarity to the program, by making the intention of the programmer more clear.

Pseudoinstructions

- Here's a list of useful pseudo-instructions.
- **mov $t0, $t1:** Copy contents of register t1 to register t0.
- **li $s0, immed:** Load immediate into to register **s0**.
  - The way this is translated depends on whether **immed** is 16 bits or 32 bits.
- **la $s0, addr:** Load address into to register **s0**.
- **lw $t0, address:** Load a word at address into register t0
- Similar pseudo-instructions exist for **sw**, etc.

Pseudoinstructions

- **Translating Some Pseudoinstructions**
- **mov $t0, $s0** addi $t0, $s0, 0
- **li $rs, small** addi $rs, $zero, small
- **li $rs, big** lui $rs, upper(big) ori $rs, $rs, lower(big)
- **la $rs, big** lui $rs, upper(big) ori $rs, $rs, lower(big)

- where **small** means a quantity that can be represented using 16 bits, and **big** means a 32 bit quantity. **upper(big)** is the upper 16 bits of a 32 bit quantity. **lower(big)** is the lower 16 bits of the 32 bit quantity.
- **upper(big)** and **lower(big)** are not real instructions. If you were to do the translation, you'd have to break it up yourself to figure out those quantities.

Pseudoinstructions

- As you look through the branch instructions, you see **beq** and **bne**, but not **bge** (branch on greater than or equal), **bgt** (branch on greater than), **ble** (branch on less than or equal), **blt** (branch on less than). There are no branch instructions for relational operators!

Pseudoinstructions

- Here's the table for translating pseudoinstructions.
- **bge $t0, $s0, LABEL** slt $at, $t0, $s0
beq $at, $zero, LABEL
- **bgt $t0, $s0, LABEL** slt $at, $s0, $t0
bne $at, $zero, LABEL
- **ble $t0, $s0, LABEL** slt $at, $s0, $t0
beq $at, $zero, LABEL
- **blt $t0, $s0, LABEL** slt $at, $t0, $s0
bne $at, $zero, LABEL

System Calls

- System Calls (syscall)
  - OS-like services
- Method
  - Load system call code into register $v0
  - Load arguments into registers $a0…$a3
  - call system with SPIM instruction `syscall`
  - After call, return value is in register $v0
- Frequently used system calls

Service	Code($v0)	Arg	Result
Print_int	1	$a1	
Print_string	4	$a0	
Read_int	5		$v0

System Call Codes

Service	Code (put in $v0)	Arguments	Result
print_int	1	$a0=integer	
print_float	2	$f12=float	
print_double	3	$f12=double	
print_string	4	$a0=addr. of string	
read_int	5		int in $v0
read_float	6		float in $f0
read_double	7		double in $f0
read_string	8	$a0=buffer, $a1=length	
sbrk	9	$a0=amount	addr in $v0
exit	10		

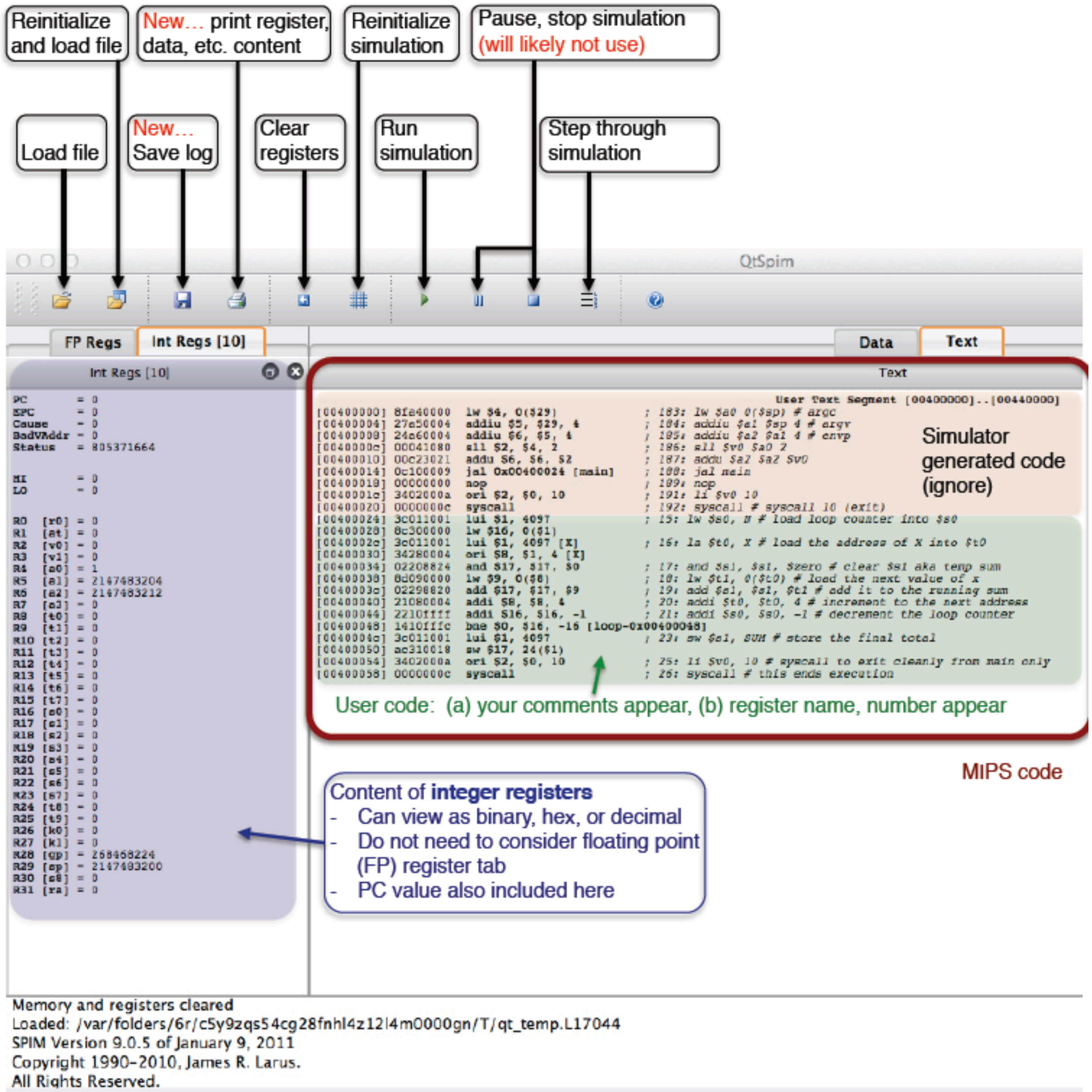
QtSPIM

- QtSpim is software that will help you to simulate the execution of MIPS assembly programs.
- It does a context and syntax check while loading an assembly program.
- In addition, it adds in necessary overhead instructions as needed, and updates register and memory content as each instruction is executed.
- Download the source from the SourceForge.org link at: http://pages.cs.wisc.edu/~larus/spim.html
- Alternatively, you can go directly to: http://sourceforge.net/projects/spimsimulator/files/
- Versions for Windows, Linux, and Macs are all available

QtSPIM

- QtSPIM window is divided into different sections:

1. The *Register tabs display the content of all registers.*
2. Buttons across the top are used to load and run a simulation
   - Functionality is described in Figure 2.
3. The *Text tab displays the MIPS instructions loaded into memory to be executed.*
   - From left-to-right, the memory address of an instruction, the contents of the address in hex, the actual MIPS instructions where register numbers are used, the MIPS assembly that you wrote, and any comments you made in your code are displayed.
4. The *Data tab displays memory addresses and their values in the data and stack segments* of the memory.
5. The *Information Console lists the actions performed by the simulator.*


Reinitialize and load file
New… print register, data, etc. content
Reinitialize simulation
Pause, stop simulation (will likely not use)
Load file
New… Save log
Clear registers
Run simulation
Step through simulation
QtSpim
FP Regs
Int Regs [10]
Data
Text
Simulator generated code (ignore)
User code: (a) your comments appear, (b) register name, number appear
MIPS code
Content of integer registers
- Can view as binary, hex, or decimal
- Do not need to consider floating point (FP) register tab
- PC value also included here
Memory and registers cleared
Loaded: /var/folders/6r/c5y9zqs54cg28fnhl4z12l4m0000gn/T/qt_temp.L17044
SPIM Version 9.0.5 of January 9, 2011
Copyright 1990-2010, James R. Larus.
All Rights Reserved.

QtSPIM Program Example

- A Simple Program

```
#sample example 'add two numbers'

.text                       # text section
.globl main                 # call main by SPIM

main:   la $t0, value       # load address 'value' into $t0
        lw $t1, 0($t0)      # load word 0(value) into $t1
        lw $t2, 4($t0)      # load word 4(value) into $t2
        add $t3, $t1, $t2   # add two numbers into $t3
        sw $t3, 8($t0)      # store word $t3 into 8($t0)

.data                       # data section
value:  .word 10, 20, 0     # data for addition
```

QtSPIM Example Program

```
## Program adds 10 and 11

    .text                       # text section
    .globl  main                # call main by SPIM

main:
    ori     $8,$0,0xA           # load “10" into register 8
    ori     $9,$0,0xB           # load “11" into register 9
    add     $10,$8,$9           # add registers 8 and 9, put result
                                # in register 10
```

QtSPIM Example Program: swap2memoryWords.asm

```
## Program to swap two memory words

   .data            # load data
   .word 7
   .word 3

   .text
   .globl main

main:
   lui $s0, 0x1001 # load data area start address 0x10010000
   lw  $s1, 0($s0)
   lw  $s2, 4($s0)
   sw  $s2, 0($s0)
   sw  $s1, 4($s0)
```

QtSPIM Example Program: procCallsProg2.asm♪

```
## Procedure call to swap two array words

    .text
        .globl  main
main:

        la      $a0, array
                addi    $a1, $0, 0
```

load para-meters for swap

```
        addi    $sp, $sp, -4
                sw      $ra, 0($sp)
```

save return address $ra in stack

jump and link to swap

```
        jal     swap
```

```
        lw      $ra, 0($sp)
        addi    $sp, $sp, 4
```

restore return address

```
        jr      $ra
```

jump to $ra

```
#       equivalent C code:♪
#          swap(int v[], int k)
#       {
#               int temp;
#               temp = v[k];
#               v[k] = v[k+1];
#               v[k+1] = temp;
#       }
# swap contents of elements $a1
# and $a1 + 1 of the array that
# starts at $a0
swap:   add     $t1, $a1, $a1
        add     $t1, $t1, $t1
        add     $t1, $a0, $t1
        lw      $t0, 0($t1)
        lw      $t2, 4($t1)
        sw      $t2, 0($t1)
        sw      $t0, 4($t1)
        jr      $ra

.data
array:  .word 5, 4, 3, 2, 1♪
♪
```

QtSPIM Example Program: systemCalls.asm♪

```
## Enter two integers in
## console window
## Sum is displayed
.text
.globl main

main:
   la $t0, value

   li $v0, 5
   syscall
   sw $v0, 0($t0)

   li $v0, 5
   syscall
   sw $v0, 4($t0)
```

system call code for read_int

result returned by call

```
   lw $t1, 0($t0)
   lw $t2, 4($t0)
   add $t3, $t1, $t2
   sw $t3, 8($t0)

   li $v0, 4
   la $a0, msg1
   syscall

   li $v0, 1
   move $a0, $t3
   syscall

   li  $v0, 10
   syscall

.data
value: .word 0, 0, 0
msg1:  .asciiz "Sum = "♪
```

system call code for print_string

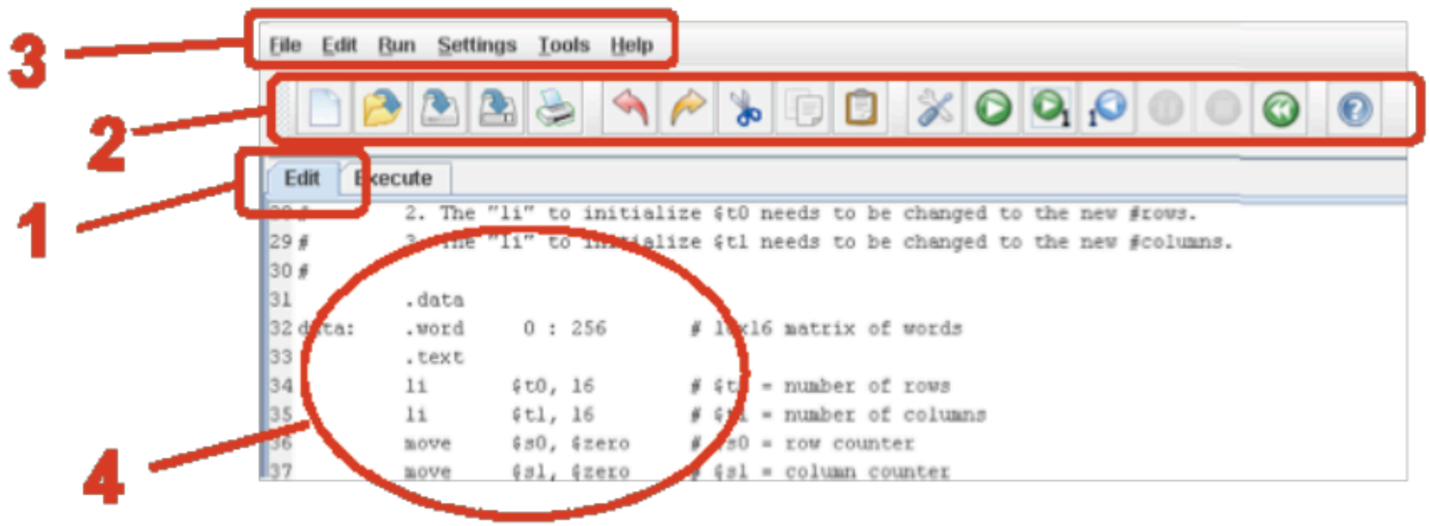
argument to print_string call

system call code for print_int

argument to print_int call

system call code for exit

MARS



1. Edit display is indicated by highlighted tab.
2, 3. Typical edit and execute operations are available through icons and menus, dimmed-out when unavailable or not applicable.
4. WYSIWYG editor for MIPS assembly language code.

Conclusion & More

- The code presented so far should get you started in writing your own MIPS assembly
- Remember the only way to master the MIPS assembly language – in fact, any computer language – is to *write lots and lots of code*
- For anyone aspiring to understand modern computer architecture *it is extremely important to master MIPS assembly as all modern computers (since the mid-80's) have been inspired by, if not based fully or partly on the MIPS instruction set architecture*