Getting started with SPIM or QtSPIM

The first step for using SPIM or QtSPIM MIPS simulator program is to download and install the program on your own machine or use a machine on campus that already has the program loaded. Computers in the Linux lab and computers in the Digital lab have QtSPIM installed. Those in the KRH MSWindows labs are suppose to have it loaded (needs to be confirmed).

In this document, mentioning SPIM will refer to both SPIM and QtSPIM unless specifically noted.

Writing a MIPS assembly program

The SPIM program contains a MIPS assembler and simulator. It does not have an integrated text editor. Thus the first step in creating a MIPS program for SPIM is to write the program using a programming oriented editor (i.e. a plain text editor, not MS Word or other word processor).

Here is an example MIPS assembly program Note that I expect programs written for this class, including assembly programs, to have a header as shown below.

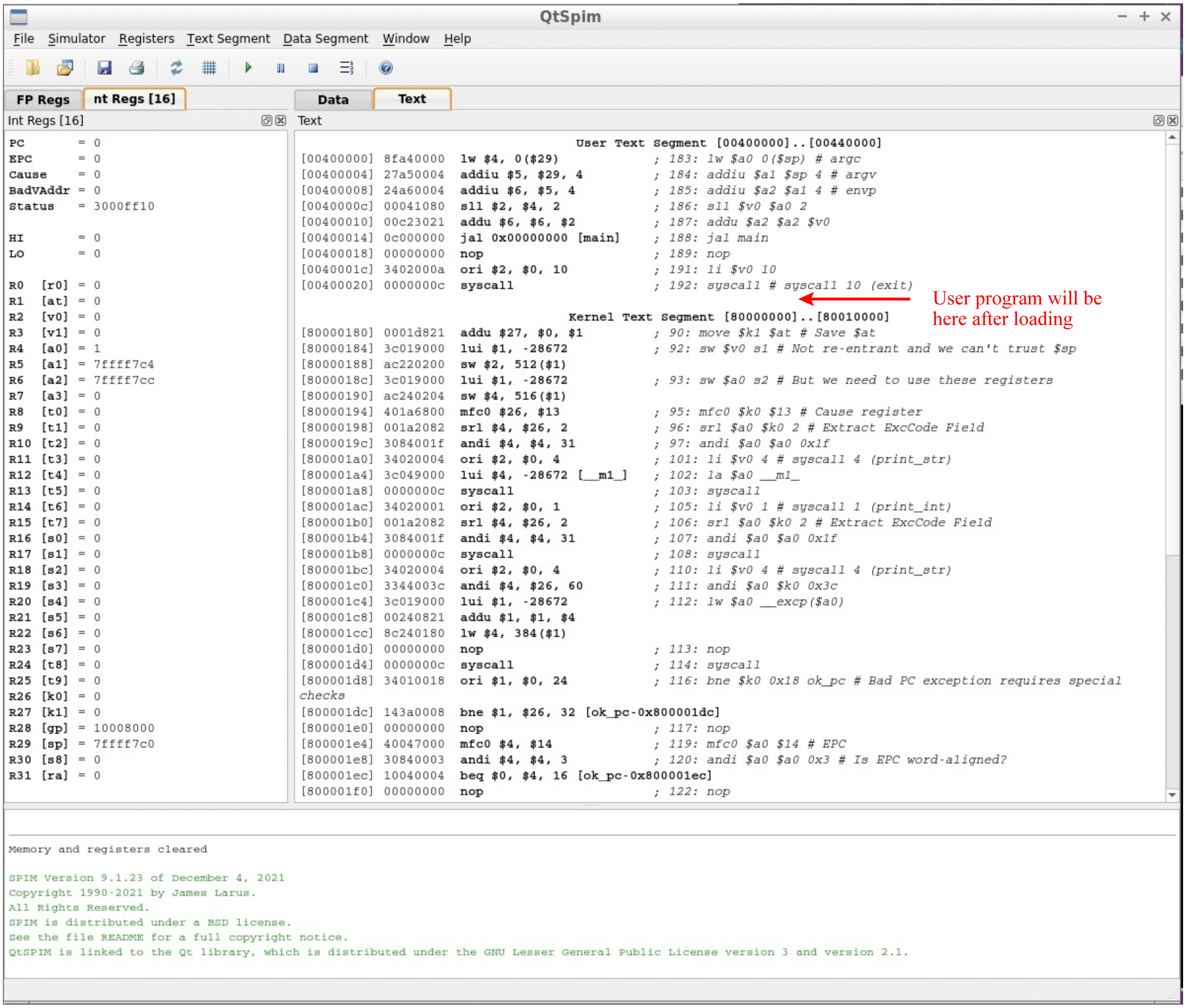
The C code for the program below is:

```
for (i=0; i<x; i++)
        array[i] = i;
```

```
########################################################################
#         A simple MIPS demo program
#     Filename:    mipsdemo1.s
#     Author:      L.Aamodt
#     Version:     1/23/22
#     Processor:   MIPS
#     Notes:         for execution using the SPIM simulator
########################################################################

        .data
arrayD: .space 100              # 100 bytes (25 words) reserved for arrayD
varX:   .word  6                # varX contains the desired loop count

        .text
                                # t0 is index variable i
                                # t1 is a temporary, frequently changing
                                # t2 is loop count & # of words put in array
                                # t3 contains the address of arrayD
                                # t4 is address of X
main:   ori     $t0, $0, 0      # set t0 to zero
        la      $t3, arrayD     # get address of arrayD
        la      $t4, varX       # place the address of X into register t4
        lw      $t2, 0($t4)     # get the loop count from varX

loop1:  slt     $t1, $t0, $t2   # check to see if i is in range
        beq     $t1, $0, exit
        sll     $t1, $t0, 2     # calculate byte index  i x 4
        add     $t1, $t1, $t3   # calculate actual word address
        sw      $t0, 0($t1)     # store index value in array
        addi    $t0, $t0, 1     # i++
        j       loop1
exit:   addi    $v0, $0, 10     # terminate the program with system call #10
        syscall
```

Note that in the example above an array and a word size variable are defined to show how it is done.

QtSPIM window before a program is loaded (see following page for a more readable copy)



To load an assembly program click File on the tool bar and select file. The program will be loaded right after the syscall in the user text segment.

Then click the single step icon on the tool bar or the run icon (arrow).

Note that integer register contents are displayed by default at the left.
On the lower part of the tool bar clicking Data will display the portion of memory where variables are stored. Memory addresses are shown but variable names are not. Look in the text section and registers to figure out memory addresses for the variables.

QtSpim

File Simulator Registers Text Segment Data Segment Window Help

FP Regs | nt Regs [16] | Data | Text

Int Regs [16]

```
PC       = 0
EPC      = 0
Cause    = 0
BadVAddr = 0
Status   = 3000ff10

HI       = 0
LO       = 0

R0  [r0] = 0
R1  [at] = 0
R2  [v0] = 0
R3  [v1] = 0
R4  [a0] = 1
R5  [a1] = 7ffff7c4
R6  [a2] = 7ffff7cc
R7  [a3] = 0
R8  [t0] = 0
R9  [t1] = 0
R10 [t2] = 0
R11 [t3] = 0
R12 [t4] = 0
R13 [t5] = 0
R14 [t6] = 0
R15 [t7] = 0
R16 [s0] = 0
R17 [s1] = 0
R18 [s2] = 0
R19 [s3] = 0
R20 [s4] = 0
R21 [s5] = 0
R22 [s6] = 0
R23 [s7] = 0
R24 [t8] = 0
R25 [t9] = 0
R26 [k0] = 0
R27 [k1] = 0
R28 [gp] = 10008000
R29 [sp] = 7ffff7c0
R30 [s8] = 0
R31 [ra] = 0
```

Text

```
                            User Text Segment [00400000]..[00440000]
[00400000] 8fa40000 lw $4, 0($29)              ; 183: lw $a0 0($sp) # argc
[00400004] 27a50004 addiu $5, $29, 4           ; 184: addiu $a1 $sp 4 # argv
[00400008] 24a60004 addiu $6, $5, 4            ; 185: addiu $a2 $a1 4 # envp
[0040000c] 00041080 sll $2, $4, 2              ; 186: sll $v0 $a0 2
[00400010] 00c23021 addu $6, $6, $2            ; 187: addu $a2 $a2 $v0
[00400014] 0c000000 jal 0x00000000 [main]      ; 188: jal main
[00400018] 00000000 nop                        ; 189: nop
[0040001c] 3402000a ori $2, $0, 10             ; 191: li $v0 10
[00400020] 0000000c syscall                    ; 192: syscall # syscall 10 (exit)

                            Kernel Text Segment [80000000]..[80010000]
[80000180] 0001d821 addu $27, $0, $1           ; 90: move $k1 $at # Save $at
[80000184] 3c019000 lui $1, -28672             ; 92: sw $v0 s1 # Not re-entrant and we can't trust $sp
[80000188] ac220200 sw $2, 512($1)
[8000018c] 3c019000 lui $1, -28672             ; 93: sw $a0 s2 # But we need to use these registers
[80000190] ac240204 sw $4, 516($1)
[80000194] 401a6800 mfc0 $26, $13              ; 95: mfc0 $k0 $13 # Cause register
[80000198] 001a2082 srl $4, $26, 2             ; 96: srl $a0 $k0 2 # Extract ExcCode Field
[8000019c] 3084001f andi $4, $4, 31            ; 97: andi $a0 $a0 0x1f
[800001a0] 34020004 ori $2, $0, 4              ; 101: li $v0 4 # syscall 4 (print_str)
[800001a4] 3c049000 lui $4, -28672 [__m1_]     ; 102: la $a0 __m1_
[800001a8] 0000000c syscall                    ; 103: syscall
[800001ac] 34020001 ori $2, $0, 1              ; 105: li $v0 1 # syscall 1 (print_int)
[800001b0] 001a2082 srl $4, $26, 2             ; 106: srl $a0 $k0 2 # Extract ExcCode Field
[800001b4] 3084001f andi $4, $4, 31            ; 107: andi $a0 $a0 0x1f
[800001b8] 0000000c syscall                    ; 108: syscall
[800001bc] 34020004 ori $2, $0, 4              ; 110: li $v0 4 # syscall 4 (print_str)
[800001c0] 3344003c andi $4, $26, 60           ; 111: andi $a0 $k0 0x3c
[800001c4] 3c019000 lui $1, -28672             ; 112: lw $a0 __excp($a0)
[800001c8] 00240821 addu $1, $1, $4
[800001cc] 8c240180 lw $4, 384($1)
[800001d0] 00000000 nop                        ; 113: nop
[800001d4] 0000000c syscall                    ; 114: syscall
[800001d8] 34010018 ori $1, $0, 24             ; 116: bne $k0 0x18 ok_pc # Bad PC exception requires special
checks
[800001dc] 143a0008 bne $1, $26, 32 [ok_pc-0x800001dc]
[800001e0] 00000000 nop                        ; 117: nop
[800001e4] 40047000 mfc0 $4, $14               ; 119: mfc0 $a0 $14 # EPC
[800001e8] 30840003 andi $4, $4, 3             ; 120: andi $a0 $a0 0x3 # Is EPC word-aligned?
[800001ec] 10040004 beq $0, $4, 16 [ok_pc-0x800001ec]
[800001f0] 00000000 nop                        ; 122: nop
```

```
Memory and registers cleared

SPIM Version 9.1.23 of December 4, 2021
Copyright 1990-2021 by James Larus.
All Rights Reserved.
SPIM is distributed under a BSD license.
See the file README for a full copyright notice.
QtSPIM is linked to the Qt library, which is distributed under the GNU Lesser General Public License version 3 and version 2.1.
```

Assembler Syntax

Comments in assembler files begin with a sharp sign (#). Everything from the sharp sign to the end of the line is ignored.

Identifiers are a sequence of alphanumeric characters, underbars (_), and dots (.) that do not begin with a number. Instruction opcodes are reserved words that *cannot* be used as identifiers. Labels are declared by putting them at the beginning of a line followed by a colon, for example:

```
      .data
item: .word 1
      .text
      .globl main       # Must be global
main: lw          $t0, item
```

Numbers are base 10 by default. If they are preceded by *0x*, they are interpreted as hexadecimal. Hence, 256 and 0x100 denote the same value.

Strings are enclosed in doublequotes ("). Special characters in strings follow the C convention:

- newline `\n`
- tab `\t`
- quote `\"`

SPIM supports a subset of the MIPS assembler directives:

`.align n`	Align the next datum on a 2^n byte boundary. For example, `.align 2` aligns the next value on a word boundary. `.align 0` turns off automatic alignment of `.half`, `.word`, `.float`, and `.double` directives until the next `.data` or `.kdata` directive.
`.ascii str`	Store the string *str* in memory, but do not null-terminate it.

.asciiz str	Store the string *str* in memory and null-terminate it.
.byte b1,..., bn	Store the *n* values in successive bytes of memory.
.data <addr>	Subsequent items are stored in the data segment. If the optional argument *addr* is present, subsequent items are stored starting at address *addr*.
.double d1, ..., dn	Store the *n* floating-point double precision numbers in successive memory locations.
.extern sym size	Declare that the datum stored at *sym* is *size* bytes large and is a global label. This directive enables the assembler to store the datum in a portion of the data segment that is efficiently accessed via register $gp.
.float f1,..., fn	Store the *n* floating-point single precision numbers in successive memory locations.
.globl sym	Declare that label *sym* is global and can be referenced from other files.
.half h1, ..., hn	Store the *n* 16-bit quantities in successive memory halfwords.
.kdata <addr>	Subsequent data items are stored in the kernel data segment. If the optional argument *addr* is present, subsequent items are stored starting at address *addr*.
.ktext <addr>	Subsequent items are put in the kernel text segment. In SPIM, these items may only be instructions or words (see the .word directive below). If the optional argument *addr* is present, subsequent items are stored starting at address *addr*.
.set noat and .set at	The first directive prevents SPIM from complaining about subsequent instructions that use register $at. The second directive reenables the warning. Since pseudoinstructions expand into code that uses register $at, programmers must be very careful about leaving values in this register.
.space n	Allocate *n* bytes of space in the current segment (which must be the data segment in SPIM).

`.text <addr>`	Subsequent items are put in the user text segment. In SPIM, these items may only be instructions or words (see the `.word` directive below). If the optional argument *addr* is present, subsequent items are stored starting at address *addr*.
`.word w1,..., wn`	Store the *n* 32-bit quantities in successive memory words.

SPIM does not distinguish various parts of the data segment (`.data`, `.rdata`, and `.sdata`).

The SPIM simulator provides some simple Input and Output routines that can be called using syscall.

Service	System call code	Arguments	Result
`print_int`	1	`$a0` = integer	
`print_float`	2	`$f12` = float	
`print_double`	3	`$f12` = double	
`print_string`	4	`$a0` = string	
`read_int`	5		integer (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	address (in `$v0`)
`exit`	10		
`print_char`	11	`$a0` = char	
`read_char`	12		char (in `$a0`)
`open`	13	`$a0` = filename (string), `$a1` = flags, `$a2` = mode	file descriptor (in `$a0`)
`read`	14	`$a0` = file descriptor, `$a1` = buffer, `$a2` = length	num chars read (in `$a0`)
`write`	15	`$a0` = file descriptor, `$a1` = buffer, `$a2` = length	num chars written (in `$a0`)
`close`	16	`$a0` = file descriptor	
`exit2`	17	`$a0` = result	

FIGURE A.9.1 System services.

System Calls

SPIM provides a small set of operating-system-like services through the system call (syscall) instruction. To request a service, a program loads the system call code (see Figure A.9.1) 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). For example, the following code prints "the answer = 5":

```
      .data
  str:
      .asciiz  "the answer = "
      .text

      li       $v0, 4   # system call code for print_str
      la       $a0, str # address of string to print
      syscall           # print the string

      li       $v0, 1   # system call code for print_int
      li       $a0, 5   # integer to print
      syscall           # print it
```

The print_int system call is passed an integer and prints it on the console. print_float prints a single floating-point number; print_double prints a double precision number; and print_string is passed a pointer to a null-terminated string, which it writes to the console.

The system calls read_int, read_float, and read_double read an entire line of input up to and including the newline. Characters following the number are ignored. read_string has the same semantics as the UNIX library routine fgets. It reads up to $n - 1$ characters into a buffer and terminates the string with a null byte. If fewer than $n - 1$ characters are on the current line, read_string reads up to and including the newline and again null-terminates the string.

Warning: Programs that use these syscalls to read from the terminal should not use memory-mapped I/O (see Section A.8).

sbrk returns a pointer to a block of memory containing n additional bytes. exit stops the program SPIM is running. exit2 terminates the SPIM program, and the argument to exit2 becomes the value returned when the SPIM simulator itself terminates.

print_char and read_char write and read a single character. open, read, write, and close are the standard UNIX library calls.

The SPIM assembler info above comes from Appendix A of the Patterson and Hennessey computer organization text where the SPIM information is written by the author of SPIM, James Larus.