Πανεπιστήμιο Ιωαννίνων

Ειδικά Θέματα Αρχιτεκτονικής και Προγραμματισμού Μικροεπεξεργαστών

Εργαστήριο

Διδάσκων: Βαρτζιώτης Φώτιος

Τμήμα Πληροφορικής και Τηλεπικοινωνιών

Slides based on Dr. Sumanta Guha's work

Introduction

- What is QtSpim?
  - a *simulator* that runs *assembly programs* for *MIPS* R2000/R3000 RISC computers
- Resources
  - http://spimsimulator.sourceforge.net/
  - Computer Organization & Design: The Hardware/Software Interface", by Patterson and Hennessy: Chapter 3 and Appendix A.9-10
- What does QtSpim do?
  - *reads* MIPS assembly language files and *translates* to machine language
  - *executes* the machine language instructions
  - shows contents of *registers* and *memory*
  - works as a *debugger* (supports *break-points* and *single-stepping*)
  - provides basic *OS-like services*, like simple I/O

Learning MIPS & QtSpim

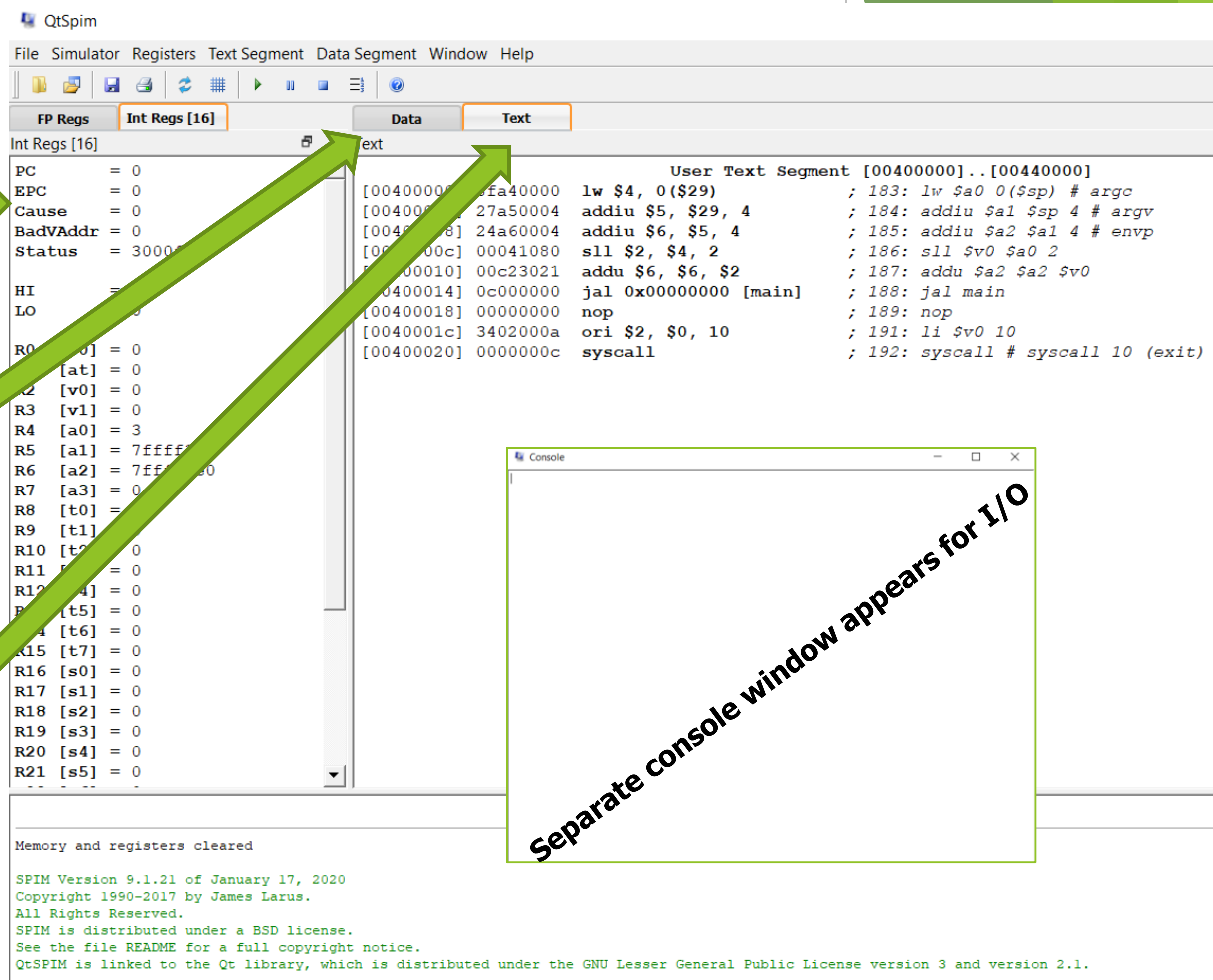
- MIPS assembly is a *low-level programming language*
- *The best way to learn any programming language is from live code*
- We will get you started by going through a few example programs and explaining the key concepts
- *We will not try to teach you the syntax line-by-line*: pick up what you need from the book and on-line tutorials
- *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

QtSpim Installation

- Windows Installation
  - download the .msi file from https://sourceforge.net/projects/spimsimulator/files/ and save it on your machine. For your convenience a copy is kept locally at the class website
  - Double click to install..

QtSpim Windows Interface

- Registers window
  - shows the values of all registers in the MIPS CPU and FPU
- Data segment window
  - shows the data loaded into the program's memory and the data of the program's stack
- Text segment window
  - shows assembly instructions & corresponding machine code
- Messages window
  - shows :QtSpim messages



Using :QtSpim

- Loading source file
  - Use *File* -> *Open* menu
- Simulation

Important!!

  - *Simulator* -> *Go* : run loaded program
    - Click the OK button in the Run Parameters pop-up window if the starting address value is “0x00400000”
  - *Simulator* -> *Break* : stop execution
  - *Simulator* -> *Clear Registers* and *Reinitialize* : clean-up before new run

Using QtSpim

- *Simulator* -> *Reload* : load file again after editing
- *Simulator* -> *Single Step* or *Multiple Step* : stepping to debug
- *Simulator* -> *Breakpoints* : set breakpoints

- Notes:
  - text segment window of QtSpim shows assembly and corresponding machine code
    - *pseudo-instructions each expand to more than one machine instruction*
  - if *Delayed Branches* is checked in *Simulator* -> *Settings (tab MIPS)…* then `statementx` will execute before control jumps to `L1` in following code - to avoid insert `nop` before `statementx`:

```
    jal L1           <-------nop
    statementx
    …
L1: …
```

QtSpim Example Program: add2numbersProg1.asm

```
## 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
```

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 Assembler Directives

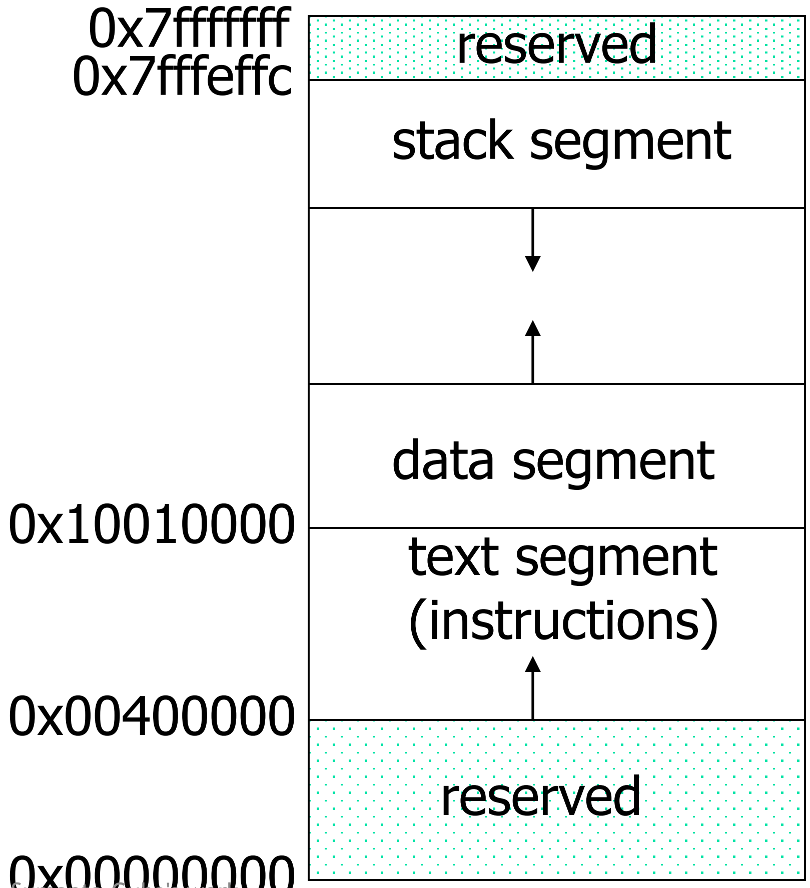
- Top-level Directives:
  - **.text**
    - indicates that following items are stored in the user text segment, typically instructions
  - **.data**
    - indicates that following data items are stored in the data segment
  - **.globl** sym
    - declare that symbol sym is global and can be referenced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

QtSpim Example Program: add2numbersProg2.asm

```
# Program adds 10 and 20

    .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        # load data integers. Default data
                              # start address 0x10010000(= value)
```

Parse the machine code for these two instructions!

MIPS Memory Usage as viewed in QtSpim



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

QtSpim Example Program: storeWords.asm

```
## Program shows memory storage and access (big vs. little endian)
            .data
here:       .word 0xabc89725, 100
            .byte 0, 1, 2, 3
            .asciiz "Sample text"
there:      .space 6
            .byte 85
            .align 2
            .byte 32

            .text
            .globl main
main:
           la $t0, here
           lbu $t1, 0($t0)
           lbu $t2, 1($t0)
           lw  $t3, 0($t0)
           sw  $t3, 36($t0)
           sb  $t3, 41($t0)
```

SPIM's memory storage depends on the underlying machine: Intel 80x86 processors are **little-endian**!

Word placement in memory is exactly same in big or little endian – a copy is placed.

Byte placement in memory depends on if it is big or little endian. In big-endian bytes in a Word are counted from the byte 0 at the left (most significant) to byte 3 at the right (least significant); in little-endian it is the other way around.

***Word access* (lw, sw)** is exactly same in big or little endian – it is a copy from register to a memory word or vice versa.

Byte access depends on if it is big or little endian, because bytes are counted 0 to 3 from left to right in big-endian and counted 0 to 3 from right to left in little-endian.

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: branchJump.asm

```
## Nonsense program to show address calculations for
## branch and jump instructions

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

# Nonsense code
# Load in SPIM to see the address calculations
main:
    j label
    add $0, $0, $0
    beq $8, $9, label
    add $0, $0, $0
    add $0, $0, $0
    add $0, $0, $0
    add $0, $0, $0
label:
    add $0, $0, $0
```

QtSpim Example Program: procCallsProg2.asm

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

```
        .text
        .globl  main
main:
```

load para-meters for swap

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

save return address $ra in stack

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

jump and link to swap

```
        jal     swap
```

restore return address

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

jump to $ra

```
        jr      $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
```

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: caller saves
. . .		(callee can clobber)
15	t7	

16	s0	callee saves
. . .		(caller can clobber)
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 (HW):
caller saves		

QtSpim System Calls

- System Calls (syscall)
  - OS-like services
- Method
  - load system call code into register \$v0 (see following table for codes)
  - load arguments into registers \$a0, …, \$a3
  - call system with QtSpim instruction `syscall`
  - after call return value is in register \$v0, or \$f0 for floating point results

SPIM 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 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

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*
- To help those with high-level programming language (e.g., C) experience, in the remaining slides we show how to synthesize various high-level constructs in assembly…

Synthesizing Control Statements (if, if-else)

```
if ( condition ) {
      statements
}
```

```
    beqz  $t0, if_end_label
    # MIPS code for the
    # if-statements.
if_end_label:
```

```
if ( condition ) {
      if-statements
} else {
      else-statements
}
```

```
    beqz    $t0, if_else_label
    # MIPS code for the
    # if-statements.
    j if_end_label
if_else_label:
    # MIPS code for the
    # else-statements
if_end_label:
```

Synthesizing Control Statements (while)

```
while ( condition ) {
        statements
}
```

```
while_start_label:
    # MIPS code for the condition expression
    beqz    $t0, while_end_label
    # MIPS code for the while-statements.
    j       while_start_label
while_end_label:
```

Synthesizing Control Statements (do-while)

```
do {
    statements
} while ( condition );
```

```
do_start_label:
     # MIPS code for the do-statements.
do_cond_label:
     # MIPS code for the condition expr:
     beqz    $t0, do_end_label
     j do_start_label
do_end_label:
```

Synthesizing Control Statements (for)

for (init ; condition ; incr) {
statements
}

```
     # MIPS code for the init expression.
for_start_label:
     # MIPS code for the condition expression
     beqz    $t0, for_end_label
     # MIPS code for the for-statements.
for_incr_label:
     # MIPS code for the incr expression.
     j for_start_label
for_end_label:
```

Synthesizing Control Statements (switch)

```
switch ( expr ) {
   case const1:
     statement1
   case const2:
     statement2
   ...
   case constN:
     statementN
   default:
     default-statement
}
```

```
# MIPS code to compute expr.
# Assume that this leaves the
# value in  $t0
beq     $t0, const1, switch_label_1
beq     $t0, const2, switch_label_2
...
beq     $t0, constN, switch_label_N
# If there is a default, then add
b       switch_default
# Otherwise, add following lineinstead:
b       switch_end_label
```

Synthesizing Control Statements (switch), cont.

```
switch_label_1:
    # MIPS code to compute statement1.
switch_label_2:
    # MIPS code to compute statement2.
   ...
switch_label_N:
    # MIPS code to compute statementN.
   # If there's a default:
switch_default:
    # MIPS code to compute default-statement.
switch_end_label:
```

Array Address Calculation

Address calculation in assembler:

address of A [n] = address of A [0] + (n* sizeof (element of A))

```
# $t0 = address of start of A.
# $t1 = n.
mul $t2, $t1, 4      # compute offset from
                     # the start of the array
                   # assuming sizeof(element)=4
add $t2, $t0, $t2    # add the offset to the
                     # address of A [0].
                     # now $t2 = &A [n].

sw  $t3, ($t2)       # A [n] = whatever is in $t3.
lw  $t3, ($t2)       # $t3 = A [n].
```

Short-Cut Expression Evaluation (and)

cond1 && cond2

```
    # MIPS code to compute cond1.
    # Assume that this leaves the value in $t0.
    # If $t0 is zero, we're finished
    # (and the result is FALSE).
    beqz $t0, and_end

    # MIPS code to compute cond2.
    # Assume that this leaves the value in $t0.
and_end:
```

Short-Cut Expression Evaluation (or)

cond1 || cond2

```
    # MIPS code to compute cond1.
    # Assume that this leaves the value in $t0.
    # If $t0 is not zero, we're finished
    # (and the result is TRUE).
     bnez $t0, or_end

     # MIPS code to compute cond2.
     # Assume that this leaves the value in $t0.
or_end:
```