Lab 10 – MIPS32

Typy danych

- 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. <u>e.g.</u> 4
  - characters enclosed in single quotes. <u>e.g.</u> 'b'
  - Strings enclosed in double quotes. <u>e.g.</u> "A string"

Rejestry

- 32 general-purpose registers
- register preceded by $ in assembly language instruction
  two formats for addressing:
  - using register number e.g. $0 through $31
  - using equivalent names e.g. $t1, $sp
- special registers Lo and Hi used to store result of multiplication and division
  - not directly addressable; contents accessed with special instruction mfhi ("move from Hi") and mflo ("move from Lo")
- stack grows from high memory to low memory

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

Struktura programu

- just plain text file with data declarations, program code (name of file should end in suffix .s to be used with SPIM simulator)
- data declaration section followed by program code section

Data Declarations

- placed in section of program identified with assembler directive **.data**
- declares variable names used in program; storage allocated in main memory (RAM)

Code

- placed in section of text identified with assembler directive **.text**
- contains program code (instructions)
- starting point for code e.g.ecution given label **main:**
- ending point of main code should use exit system call (see below under System Calls)

Comments

- anything following # on a line

```
.data # variable declarations follow this line
        # ...
.text # instructions follow this line

main: # indicates start of code
        # (first instruction to execute)
       # ...
 # End of program, leave a blank line afterwards
```

Instrukcje ładowania/zapisu

- RAM access only allowed with load and store instructions
- all other instructions use register operands

<u>load:</u>

- lw register_destination, RAM_source
  - #copy word (4 bytes) at source RAM location to destination register.
- lb register_destination, RAM_source
  - #copy byte at source RAM location to low-order byte of destination register, # and sign-e.g.tend to higher-order bytes

<u>store word:</u>

- sw register_source, RAM_destination #store word in source register into RAM destination
- sb register_source, RAM_destination
  - #store byte (low-order) in source register into RAM destination

<u>load immediate:</u>

- li register_destination, value #load immediate value into destination register

Tryby adresowania

<u>load address:</u>

- la $t0, var1
  copy RAM address of var1 (presumably a label defined in the program) into register $t0

<u>indirect addressing:</u>

- lw $t2, ($t0) load word at RAM address contained in $t0 into $t2
- sw $t2, ($t0) store word in register $t2 into RAM at address contained in $t0

<u>based or indexed addressing:</u>

- lw $t2, 4($t0) load word at RAM address ($t0+4) into register $t2
- "4" gives offset from address in register $t0
- sw $t2, -12($t0) store word in register $t2 into RAM at address ($t0 - 12)
- negative offsets are fine

Instrukcje arytmetyczne

- 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 - $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

Instrukcje sterujące

<u>Branches</u>

- comparison for conditional branches is built into 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

<u>Jumps</u>

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

<u>Subroutine Calls</u>

- 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)

Wywołania systemowe

Service	Code in $v0	Arguments	Results
print_int	1	$a0 = integer to be printed	
print_float	2	$f12 = float to be printed	
print_double	3	$f12 = double to be printed	
print_string	4	$a0 = address of string in memory	
read_int	5		integer returned in $v0
read_float	6		float returned in $v0
read_double	7		double returned in $v0
read_string	8	$a0 = memory address of string input buffer $a1 = length of string buffer (n)	
sbrk	9	$a0 = amount	address in $v0
exit	10		