Announcements

- Homework #7 is due today
- Reading for this lecture SPIM Manual

SPIM (MIPS) Architecture

- 32 bit architecture
  - registers are 32 bits wide
  - memory addresses are 32 bits (2^{30} words each 4 bytes)
  - instructions are 32 bits wide
- Load/Store Architecture
  - can only load or store values to memory
    - I.e. there is not increment memory location.
    - no indirect addressing mode
- instructions use “three addresses”
  - destination, source1, source2
  - add t0, t1, t2
- has 32 registers
  - but one of them contains a zero

Syntax of SPIM Assembly

- Labels
  - arbitrary strings (letters, digits, _, or period) but must start with a non-number
  - separated by a colon from the rest of the statement.
  - the label main is special, it is the place where your program starts execution.
  - may be on a line by themselves.
- Instructions
  - have 1-4 letter pneumonic
- Operands
  - can be registers, immediate, or labels
  - registers have a $ in front of their names
- Comments
  - comment character is #
  - from the comment character to end of line is ignored.

Registers

- names indicates the purpose of a register
  - v0-v1 return value of a function
  - a0-a3 argument 1-4
  - t0-t9 temporary registers
  - s0-s7 saved temporary
  - gp pointer to global area
  - sp stack pointer
  - fp frame pointer
  - ra return address
  - k0-k1 reserved for OS kernel

Load & Store Instructions

- Load instruction has two operands
  - a destination registers
  - a source operand
- Source operand has four addressing modes
  - (reg) load from another register
  - imm a constant
  - imm (reg) immediate plus contents of register
  - symbol address of a label
  - symbol imm address of a label + imm
  - symbol imm (register) address of label + imm + register

Load & Store (cont.)

- Load instructions
  - li rd, <operand> load the values into register rd.
  - la rd, <operand>
  - lw rd, <operand> load the contents of the memory indicated by <operand> into the register rd.
  - lb rd, <operand> load byte into the register rd.
- Store instructions
  - sw rs, <operand> store rs into the memory location indicated by <operand>
- Load and Store should be aligned
  - if you load a word it should have 0's in the two low bits of the address.

Register Instructions

- move rd, rs
  - move contents of rs into rd
- add rd, rs1, <rs2 | imm>

  addu rd, rs1, <rs2 | imm>
  - rd gets the sum of rs1 and rs2
- sub rd, rs1, <rs2 | imm >

  subu rd, rs1, <rs2 | imm >
  - rd gets the unsigned results of rs1 - rs2
- mul rd, rs1, rs2
  - rd = rs1 * rs2
- div rd, rs1, rs2
  - rs = rs1 / rs2
- rem rd, rs1, rs2
  - rs = rs1 mod rs2

Logical Operations

- and rd, rs1, rs2

  and rd, rs1, imm
  - rd gets the bitwise and of rs1 and rs2
- or rd, rs1, rs2

  or rd, rs1, imm
  - rs gets the bitwise or of rs1 and rs2
- sll rd, rs, imm
  - logical shift left into rs of rd by imm bits
- srl rd, rs, imm
  - logical shift right into rs of rd by imm bits

Branch Instructions

- j <addr>
  - jump to the address indicated by <addr>
- jr reg
  - jump to the address stored in register reg
- jal reg
  - jump to the address stored in reg, but save PC in reg 31 (ra)
- beq rs1, rs2, <addr>
  - branch to <addr> if contents of rs1 equal contents of rs2
- bne rs1, rs2, <addr>
  - branch to <addr> if contents of rs1 do not equal contents of rs2

Branch Instructions (cont.)

- other comparison branch instructions
  - bge rs1, rs2, <addr> branch if (rs1 >= rs2)
  - bgt rs1, rs2, <addr> branch if (rs1 > rs2)
  - ble rs1, rs2, <addr> branch if (ts1 <= rs2)
  - blt rs1, rs2, <addr> branch if (ts1 < rs2)
- <addr>
  - is typically a label
  - branches are 16 bit PC relative
    - limited to 2^{15}-1 instructions forward or 2^{15} backwards

I/O (& exit) in SPIM

- Uses the syscall instruction
  - load appropriate code into $v0
- Output

Name	Code	Argument
Print_int	1	$a0 = integer
print_string	4	$a0 = address of start of string
exit	10	

- Input

Name	Code	Argument	Return
Read_string	8	$a0 = buf $a1 = len	
read_int	5		$v0

Segments

- Program and data are separated into two areas
- code segment
  - read only
- data segment
  - read and write
- stack segment
  - stores information about current procedures
    - return address
    - local variables
- This prevents changing the program by accident

Data Segment

- Use .data directive to start a data segment
- storage allocation

  .word n
  - one word of memory is allocated and loaded with n

  .byte x
  - one byte of memory is allocated and loaded with x

  .asciiz str
  - one zero (null) terminated string is allocated
  - str is a string in double quotes
    - can contain C style special characters (e.g. \n)

  .align n
  - align the next memory allocated to a n bytes boundary
    - good habit to place these after any .asciiz

  .space n
  - allocate (save) n bytes of space (usefull for arrays)

Sample Program

- Read two numbers and add them and print their sum

```
main:
    la $a0, prompt1
    li $v0, 4
    syscall

    li $v0, 5
    syscall

    move $t0, $v0

    la $a0, prompt2
    li $v0, 4
    syscall
```

Example (cont.)

```
li $v0, 5
syscall

move $t1, $v0

add $t2, $t0, $t1
la $a0, outputString
li $v0, 4
syscall
move $a0, $t2
li $v0, 1
syscall

li $v0, 10
syscall
```

Data Segment

```
        .data
prompt1:
        .asciiz             "Enter the first number: "
        .align 4
prompt2:
        .asciiz "Enter the second number: "
        .align 4
outputString:
        .asciiz "The sum is "
        .align 4
```