Orange Coast College
Business Division
Computer Science Department

CS 116- Computer Architecture

Instructions: Part the Last
Brief review

• Different Types Instructions
  – R-Type
  – I-Type
  – J-Type

• Assembly Instructions
  – Arithmetic
  – Memory
  – Branching
Today

• Support for procedures
  – Assembler and Hardware
• Shifting
• Data other than numbers
• And Last, But Certainly Not Least…
  Actually programming in assembly!
Procedures

- AKA Subroutines
- Reusable chunks of code
- Used to structure programs
  - Easier to read
  - Easier to understand
- Leave no trace
Properties of Procedures

• Have the following properties:
  – Parameters - the input to the procedure
  – Code Block - the actual instructions of the procedure
  – Return value - the result of the procedure
  – Separate from the current program location

• Often need:
  – Storage for local variables
How do we call a procedure?

1) Place parameters someplace procedure can access them
2) Transfer execution to the procedure location
3) Acquire local storage resources
4) Execute the code
5) Place the result someplace the original code can access
6) Return control to the original code
How does the hardware help?

• Well, registers are fast...
  – MIPS reserves $a0-$a3 for passing parameters
  – $v0 and $v1 are reserved for return values
  – $ra is reserved for the address to return control to
And what about instructions?

- Because the procedure's code is not close by
  - Use a jump instruction
- MIPS provides a specialized jump instructions
  - Jump and link instruction
    - \texttt{jal ProcedureAddress}
    - Link means store the address we are coming from in \$ra
      - Called the return address
  - Jump register instruction
    - \texttt{jr $ra}
    - We can do this at the end of a procedure to return
A short example

Proc1: add $t0, $a0, $a1
mul $t0, $0, $a3
sub $t0, $a4, $t0
add $v0, $t0, $zero
jr $ra

Main:  li $a0, 12
li $a1, 42
li $a3, 123
li $a4, 10120
jal Proc1
(do some more instructions)
Great, but...

• What happens if:
  – More parameters than 4?
  – Need to call another procedure?
    • Destroy $ra as well as parameters ($a0-$a4)
  – Need storage for more than 8 temporaries?

• Answer
  – We need non-register memory to store information
Ahh, the Stack

• Recall, the stack is used for local storage
  – LIFO structure - Last In First Out

• Operations
  – Push values on
    • Adds data to stack
  – Pop top the value off
    • Removes data from the stack

• Keep in mind
  – Stack grows downward in memory
Making it work

• Need to know
  – Where the top of the stack is
    • MIPS: Stack Pointer $sp
  – How to put data on stack
    • Use $sp as the offset register and use
      – Store word instruction
      – Load word instruction
    • Use arithmetic to allocate/deallocate space
      – Add/subtract to/from $sp to adjust stack
The Stack

• Push example
  \[\text{subi } \$sp, \$sp, 12\]
  \[\text{sw } \$a0, 8(\$sp)\]
  \[\text{sw } \$a1, 4(\$sp)\]
  \[\text{sw } \$t0, 0(\$sp)\]

• Pop example
  \[\text{lw } \$t0, 0(\$sp)\]
  \[\text{lw } \$a1, 4(\$sp)\]
  \[\text{lw } \$a0, 8(\$sp)\]
  \[\text{addi } \$sp, \$sp, 12\]
Using the stack

• Push parameters on the stack
• Push registers that need to be saved
  – $s0..$s7
  – NOT $t0..$t7
• Allocate local temporary storage
  – Subtract appropriate amount from $sp
• When calling another procedure
  – Save $ra
But if stack changes...

- How do we access local arrays, etc?
  - Use a register to keep track of where they are located

- Frame Pointer
  - Register $fp
  - Set to value of $sp when procedure starts
  - Change $sp as much as we want, but always have a reference point
    - Handled by the program, not by the processor
Stack Frame

- $sp$
- $fp$
- $fp$
- $sp$

- Saved arg Regs
- Saved $ra$
- Saved regs
- Local storage

- $fp$
- $sp$
Shifting

• Operations that move bits
  – Shift Left
    • Move all the bits in register left a certain amount
  – Shift Right
    • Move all the bits in the register right a certain amount

• Examples
  – Shift Left  \texttt{sll \$t0, \$t1, 3}
    \[
    \begin{array}{cccc}
    0 & 0 & 0 & 1 \\
    \end{array}
    \quad \begin{array}{cccc}
    1 & 1 & 0 & 0 \\
    \end{array}
    \]
  – Shift Right  \texttt{srl \$t0, \$t1, 2}
    \[
    \begin{array}{cccc}
    1 & 1 & 0 & 0 \\
    \end{array}
    \quad \begin{array}{cccc}
    0 & 0 & 1 & 1 \\
    \end{array}
    \]
Shifting

• Notice
  – Shift left by 1 is like multiplying by 2
  – Shift right by 2 is like dividing by 2

• If bits are shifted past the 32\textsuperscript{nd} (or 0\textsuperscript{th}) bit
  – They are just shifted away, to be lost forever
Rotating

• Similar to shifting, except
  – Bits shifted off the end are wrapped around to the other side
  – Example: rotate left 3

\[
\begin{array}{c|c|c|c|c|c}
0 & 0 & 1 & 1 & 0 \\
\end{array}
\quad 1\quad 0\quad 0\quad 0\quad 1
\]
More than numbers

- Computers not calculators
  - Work with not just numbers
  - But also text
- Deal with characters
  - Encoded with ASCII
  - Combined into sequences of strings
- Use the load/store byte instruction to get data
  - `lb $t0, 1($s0)`
  - `sb $t0, 1($s0)`
- Use load address to get the address of data
  - `la $t0, DataLabel (Address)`
String representations

- Reserve the first position to indicate length
  - Can only store 255 characters in the string
  - Pascal

- Use an external variable
  - Requires more memory
  - Java

- Use a special character to terminate the string
  - C uses the 0 (zero) character
    - Null terminated strings
Finally, we are ready to code

- **Syntax**
  - One instruction per line
  - `#` starts a comment
    - Continues until the end of the line
  - **Identifiers**
    - Alphanumeric, `_' and `.`
    - Cannot start with a number
  - **Labels**
    - Identifiers at the start of a line followed by `:'
More Language

• Syntax
  – Strings enclosed in “ ”
    • \n    A new line
    • \t    A tab character
    • \”     A quotation mark
  – Numbers
    • Base 10
    • Hexadecimal
      – 0xABCD
Directives

- Special commands for the assembler
- Usually starts with '.'
  - Example:
    
    .asciiz "The sum from 0..10 is %d\n"
    
  - Is equivalent to:

    .byte 84, 104, 101, 32, 115, 117, 109, 32
    .byte 102, 114, 111, 109, 32, 48, 32, 46
    .byte 46, 32, 49, 48, 32, 105, 115, 32
    .byte 37, 100, 10, 0

  - Note:

    - 10 is ASCII code for line feed
    - 0 is ASCII code for the NULL character
Directives

• Some example directives:

  .asciiz str    # Store null-terminated string
  .byte b1, ... , bn  # Store in successive bytes
  .data <addr>  # Store in data segment
  .float fl1, ... , fn  # Store floating-pt. single precision
  .globl sym  # sym can be referenced from other files
  .space n    # Allocate n-bytes of space in data seg.
  .text <addr>  # Put text items in text segment
  .word w1, ... , wn  # Full word in successive memory
I/O

• Need to be able to read and write data
  – For the user

• SPIM, this is accomplished with syscall

• In general, uses interrupt
  – Stalls processor state
  – Tells the processor to execute a procedure associated with the interrupt.
  – In PCs, this is what an IRQ is
Using syscall (interrupts)

- Put a given constant into register $v0
- Fill in the set of argument registers
- Execute the syscall instruction
- The result will be in register $v0

- Look on page A-49 of the book for available service codes
Example syscall

.data
var: asciiiz "This is a string\n"

.text
.globl main
main: li $v0, 4
la $a0, var
syscall