MIPS & NiosII Assembly Language

Computer Science 104
Lecture 6

Today’s Lecture
• Homework #2
• Midterm I Feb 16 (in class open book open notes)

Outline
• Review
• Assembly Programming
Reading
Chapter 2, Appendix B, NiosII Soft Processor

Review: A Program
#include <iostream.h>
main()
{
    int *a = new int[100];
    int *p = a;
    int k;
    for (k = 0; k < 100; k++)
    {
        *p = k;
        p++;
    }
    cout << “entry 3 = “ << a[3] << endl;
}

Review: Stored Program Computer
• Instructions: a fixed set of built-in operations
• Instructions and data are stored in the (same) computer memory.
• Fetch Execute Cycle
  while (!done)
      fetch instruction
      execute instruction

Review: What Must be Specified?
• Instruction Format
  ➢ how do we tell what operation to perform?
• Location of operands and result
  ➢ where other than memory?
  ➢ how many explicit operands?
  ➢ how are memory operands located?
  ➢ which can or cannot be in memory?
• Data type and Size
• Operations
  ➢ what are supported
• Successor instruction
  ➢ jumps, conditions, branches
  • fetch-decode-execute is implicit!

Review: MIPS ISA Categories
• Arithmetic
  ➢ add, sub, mul, etc
• Logical
  ➢ AND, OR, SHIFT
• Data Transfer
  ➢ load, store
  ➢ MIPS is LOAD/STORE architecture
• Conditional Branch
  ➢ implement if, for, while... statements
• Unconditional Jump
  ➢ support method invocation (procedure calls)
Review: MIPS Instruction Formats

R-type: Register-Register

| 31 | 26 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Op | Rs | Rt | Rd | shamt | funct |

I-type: Register-Immediate

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs</td>
<td>Rt</td>
<td>immediate</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

J-type: Jump / Call

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>target</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Terminology

Op = opcode
Rs, Rt, Rd = register specifier

Assembler and Assembly Language

• Machine language is a sequence of binary words.
• Assembly language is a text representation for machine language plus extras that make assembly language programming easier (more readable too!).

MIPS Assembly Language

• One instruction per line.
• Numbers are base-10 integers or Hex w/ leading 0x.
• Identifiers: alphanumeric, _, string starting in a letter or _
• Labels: identifiers starting at the beginning of a line followed by “:”
• Comments: everything following # till end-of-line.
• Instruction format: Space and “,” separated fields.
  > [Label:] <op> reg1, [reg2], [reg3]     [# comment]
  > [Label:] <op>  reg1, offset(reg2)       [# comment]
  > .Directive [arg1], [arg2], . . .

Assembly Language (cont.)

• Pseudo-instructions: extend the instruction set for convenience
• Examples
  > move $2, $4 # $2 = $4, (copy $4 to $2)
  Translate to:
  add $2, $4, $0
  > li $8, 40 # $8 = 40, (load 40 into $8)
  addi $8, $0, 40
  > sd $4, 0($29) # mem[$29] = $4; Mem[$29+4] = $5
  sw $4, 0 ($29)
  sw $5, 4($29)
  > la $4, 0x1000056c # Load address $4 = <address>
  lui $4, 0x1000
  ori $4, $4, 0x056c

• Directives: tell the assembler what to do...
• Format “."<string> [arg1], [arg2] ...
The C / C++ code

#include <iostream.h>

int main ( )
{
    int i;
    int sum = 0;
    for(i=0; i <= 100; i++)
    sum = sum + i*i ;
    cout << "The answer is " << sum < endl;
}

Let's write the assembly … :)
Example2

Task: sum together the integers stored in memory

```
.text  # Code
.align 2  # align on word boundary
.globl main # declare main
main:
# MAIN procedure Entrance
# fill in what goes here
.data                   # Start of data segment
list:   .word   35, 16, 42, 19, 55, 91, 24, 61, 53
msg:    .asciiz "The sum is 
nn:
.asciiz "\n"
```

Nios II toupper

```
.text  # $ directive, begin the text segment
.global main # $ directive, declare main as a global variable
main:   # $ a label, it provides a name for a memory location
# convert the string at str toupper
movi r9, 96
movia r8, str
loop:
    ldb r10, 0(r8)  # load character
    beq r10, zero, done  # if null (0) done
    blt r10, r9, isupper  # if < 96 already upper
    subi r10, r10, 32  # subtract 32 to make it uppercase
    stb r10, 0(r8)  # write the character back to memory
    isupper:
        addi r8, r8, 1  # increment pointer into string
    br loop  # go back to loop
done:
    ret  # return from main
.data  # $ a directive, begin the data segment
str:
    .asciiz "This is a string"
```

Review: Procedure Call and Return

```
int equal(int a1, int a2) {
    int tsame;
    tsame = 0;
    if (a1 == a2)
        tsame = 1;
    return(tsame);
}
main()
```

Procedure Call GAP

ISA Level
• call and return instructions
C++ Level
• Local Name Scope
  • change tsame to same
• Recursion
• Arguments and Return Value (functions)
Assembly Level
• Must bridge gap between HLL and ISA
• Supporting Local Names
• Passing Arguments (arbitrary number?)

Supporting Procedures

• What data structure?

Procedure Call (Stack) Frame

• Procedures use a frame in the stack to:
  • Hold values passed to procedures as arguments.
  • Save registers that a procedure may modify, but which the procedure’s caller does not want changed.
  • To provide space for local variables.
    (variables with local scope)
  • To evaluate complex expressions.
**Call-Return Linkage: Stack Frames**

- **FP**
  - Arguments
  -callee saves
  -local variables at fixed offset from FP
- **Local Variables**
- **Dynamic area**

Grows and shrinks during expression evaluation

**FP**

- **SP**

**NiosII calling convention**

- **High Mem**
  - Higher address:
    - arguments
    - registers
- **Low Mem**
  - Lower address:
    - stack frame
    - space for stack temporary

**MIPS Register Naming Conventions**

- **Stack**
- **Data**
- **Text**

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>zero constant 0</td>
</tr>
<tr>
<td>1</td>
<td>at reserved for assembler</td>
</tr>
<tr>
<td>2</td>
<td>return value [low 32 bits]</td>
</tr>
<tr>
<td>3</td>
<td>return value [next 32 bits]</td>
</tr>
<tr>
<td>4</td>
<td>arguments</td>
</tr>
<tr>
<td>5</td>
<td>temporary: caller saves</td>
</tr>
</tbody>
</table>

**Nois II Register Naming Conventions**

- **Stack**
- **Data**
- **Text**

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>zero constant 0</td>
</tr>
<tr>
<td>1</td>
<td>at reserved for assembler</td>
</tr>
<tr>
<td>2</td>
<td>return value [low 32 bits]</td>
</tr>
<tr>
<td>3</td>
<td>return value [next 32 bits]</td>
</tr>
<tr>
<td>4</td>
<td>arguments</td>
</tr>
<tr>
<td>5</td>
<td>temporary: caller saves</td>
</tr>
</tbody>
</table>

**MIPS/GCC Procedure Calling Conventions**

**Calling Procedure**

- **Step-1: Setup the arguments:**
  - The first four arguments (arg0-arg3) are passed in registers $a0$-$a3$
  - Remaining arguments are pushed onto the stack (in reverse order arg5 is at the top of the stack).

- **Step-2: Save caller-saved registers**
  - Save registers $t0$-$t9$ if they contain live values at the call site.

- **Step-3: Execute a jal instruction.**
MIPS/GCC Procedure Calling Conventions (cont.)

Called Routine

- Step-1: Establish stack frame.
  - Subtract the frame size from the stack pointer.
  ```
  subiu $sp, $sp, <frame-size>
  ```
  - Typically, minimum frame size is 32 bytes (8 words).
- Step-2: Save callee saved registers in the frame.
  - Register $fp is always saved.
  - Register $ra is saved if routine makes a call.
  - Registers $s0-$s7 are saved if they are used.
- Step-3: Establish Frame pointer
  ```
  addiu $fp, $sp, <frame-size> - 4
  ```

On return from a call

- Step-1: Put returned values in registers $v0, $v1.
  (if values are returned)
- Step-2: Restore callee-saved registers.
  - Restore $fp and other saved registers. $ra, $s0 - $s7
- Step-3: Pop the stack
  ```
  addiu $sp, $sp, <frame-size>
  ```
- Step-4: Return
  ```
  jr $ra
  ```

Example2

```c
# Example for CPS 104
# Program to add together list of 9 numbers.
.text
.align 2
.globl main
main:
# MAIN procedure Entrance
subu $sp, 40
#
sw $ra, 36($sp) # Save return address
sw $s2, 28($sp) # > Entry Housekeeping
sw $s1, 24($sp) # / save registers on stack
sw $s0, 20($sp) # /
move $v0, $0
#
la $s0, list # Initialization
la $s2, msg # /
la $s3, list+36 #/
main:
# MAIN procedure Entrance
subu $sp, 40
#
sw $ra, 36($sp) # Save return address
sw $s2, 28($sp) # > Entry Housekeeping
sw $s1, 24($sp) # / save registers on stack
sw $s0, 20($sp) # /
move $v0, $0
#
la $s0, list # Initialization
la $s2, msg # /
la $s3, list+36 #/
main:
# MAIN procedure Entrance
subu $sp, 40
#
sw $ra, 36($sp) # Save return address
sw $s2, 28($sp) # > Entry Housekeeping
sw $s1, 24($sp) # / save registers on stack
sw $s0, 20($sp) # /
move $v0, $0
#
la $s0, list # Initialization
la $s2, msg # /
la $s3, list+36 #/
main:
# MAIN procedure Entrance
subu $sp, 40
#
sw $ra, 36($sp) # Save return address
sw $s2, 28($sp) # > Entry Housekeeping
sw $s1, 24($sp) # / save registers on stack
sw $s0, 20($sp) # /
move $v0, $0
#
la $s0, list # Initialization
la $s2, msg # /
la $s3, list+36 #/
main:
# MAIN procedure Entrance
subu $sp, 40
#
sw $ra, 36($sp) # Save return address
sw $s2, 28($sp) # > Entry Housekeeping
sw $s1, 24($sp) # / save registers on stack
sw $s0, 20($sp) # /
move $v0, $0
#
la $s0, list # Initialization
la $s2, msg # /
la $s3, list+36 #/
main:
# MAIN procedure Entrance
subu $sp, 40
#
sw $ra, 36($sp) # Save return address
sw $s2, 28($sp) # > Entry Housekeeping
sw $s1, 24($sp) # / save registers on stack
sw $s0, 20($sp) # /
move $v0, $0
#
la $s0, list # Initialization
la $s2, msg # /
la $s3, list+36 #/
main:
# MAIN procedure Entrance
subu $sp, 40
#
sw $ra, 36($sp) # Save return address
sw $s2, 28($sp) # > Entry Housekeeping
sw $s1, 24($sp) # / save registers on stack
sw $s0, 20($sp) # /
move $v0, $0
#
la $s0, list # Initialization
la $s2, msg # /
la $s3, list+36 #/
```
Details of the MIPS & Nios II instruction sets

- Register zero always has the value zero
- Even if you try to write it
- Branch (al) and jal instructions put the return address PC+4 into the link register
- All instructions change all 32 bits of the destination register ( lui, lb, lh) and read all 32 bits of sources (add, sub, and, or, ...)
- Immediate arithmetic and logical instructions are extended as follows:
  > Arithmetic immediates are zero extended to 32 bits
  > Logical immediates are sign extended to 32 bits
- Ib and lh extend data as follows:
  > lb, lh are zero extended
  > lb, lh are sign extended

Miscellaneous MIPS Instructions

- break A breakpoint trap occurs, transfers control to exception handler
- syscall A system trap occurs, transfers control to exception handler
- coprocessor instrs Support for floating point
- TLB instructions Support for virtual memory: discussed later
- restore from exception Restores previous interrupt mask & kernel/user mode bits into status register
- load word left/right Supports unaligned word loads
- store word left/right Supports unaligned word stores

Summary

- Assembler Translates Assembly to Machine code
- Pseudo Instructions
- System Call
- Procedure Calls
- Recursion, Other Instruction Sets

Next Time

Reading
- Ch. 2, Appendix B