Today’s Lecture

• Homework #2
• Midterm I Feb 12

Outline
• Review
• Assembly Programming

Reading
  Chapter 2 & 3, Appendix A, NiosII Soft Processor
Review: A Program

```cpp
#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**
- **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

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs</td>
<td>Rt</td>
<td>Rd</td>
<td>shamt</td>
<td>func</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

I-type: Register-Immediate

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</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>
</tr>
</tbody>
</table>

J-type: Jump / Call

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>target</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)
    Translates 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
Assembly Language (cont.)

• **Directives**: tell the assembler what to do...

• **Format** “.”<string> [arg1], [arg2] . . .

• **Examples**
  
  .data [address]  # start a data segment.
  # [optional beginning address]
  .text [address]  # start a code segment.
  .align n  # align segment on 2^n byte boundary.
  .ascii <string> # store a string in memory.
  .asciiz <string> # store a null terminated string in memory
  .word w1, w2, . . . , wn # store n words in memory.

---

A Simple Program

• **Add two numbers x & y together**

```assembly
.text  # declare text segment
.align 2  # align it on 4-byte boundary
main:  # label for main
  la $3, x  # load address of x into R3 (pseudo-inst)
  lw $4, 0($3)  # load value of x into R4
  la $3, y  # load address of y into R3 (pseudo-inst)
  lw $5, 0($3)  # load value of y into R5
  add $6, $4, $5  # compute x+y
  jr $31  # return to calling routine
.data  # declare data segment
.align 2  # align it on 4-byte boundary
x:.word 10  # initialize x to 10
y:.word 3  # initialize y to 3
```

---
The C / C++ code

```c
#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 ... :)
```

System Call Instruction

- System call is used to communicate with the operating system, and request services (memory allocation, I/O)
- Load system call code (value) into Register $v0
- Load arguments (if any) into registers $a0, $a1 or $f12 (for floating point).
- do: syscall
- Results returned in registers $v0 or $f0.
- On NiosII this is a trap instruction (but we are not going to use it)
Echo number and string

.text
main:
    li $v0, 5          # code to read an integer
    syscall          # do the read (invokes the OS)
    move $a0, $v0     # copy result from v0 to a0

    li $v0, 1          # code to print an integer
    syscall           # print the integer

    li $v0, 4          # code to print string
    la $a0, nln        # address of string (newline)
    syscall

.data
.align 2
name: .word 0, 0
nln: .asciiz "\n"

Echo Continued

    li $v0, 8          # code to read a string
    la $a0, name       # address of buffer (name)
    li $a1, 8          # size of buffer (8 bytes)
    syscall

    la $a0, name       # address of string to print
    li $v0, 4          # code to print a string
    syscall

    jr $31              # return
### Nios II ASM Input/Output

```
.text
.global main

main:   # start function
    mov r16, r31 # save a copy of r31
    movia r4, input_fmt # put address of input_format in r4 (arg1)
    call printf # invoke printf function to display string
    movia r4, scanf_str # put address of scanf format in r4
    movia r5, string # put address of string storage in r5 (arg2)
    call scanf # invoke scanf
    movia r4, output_fmt # put address of output format string into r4 (arg1)
    movia r5, string # put string address into r5 (arg2)
    call printf # invoke printf to display string
    mov r31, r16 # restore copy of r31
    ret # return to calling function

.data

input_fmt: .asciz "Please enter a string\n"
scanf_str: .asciz "%s"
string: .skip 256
output_fmt: .asciz "String is %s\n"
```

### 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 ",
nln:    .asciiz "\n"
```

© Alvin R. Lebeck
Nios II toupper

```assembly
.text  # a directive, begin the text segment
.global main  # a 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 uppercase
        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:  # another label
    .asciz "This is a string"
```

Review: Procedure Call and Return

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

main() {
    int x, y, same;
    x = 43;
    y = 2;
    same = equal(x, y);
    // other computation
}
```

PC $31

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x10000</td>
<td>addi $1, $0, 43</td>
</tr>
<tr>
<td>0x10004</td>
<td>addi $2, $0, 2</td>
</tr>
<tr>
<td>0x10008</td>
<td>jal 0x30408</td>
</tr>
<tr>
<td>0x30408</td>
<td>addi $3, $0, 0</td>
</tr>
<tr>
<td>0x3040c</td>
<td>bne $1, $2, 8</td>
</tr>
<tr>
<td>0x30410</td>
<td>addi $3, $0, 1</td>
</tr>
<tr>
<td>0x30414</td>
<td>jr $31</td>
</tr>
</tbody>
</table>
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
NiosII calling convention

Review: A Program

```cpp
#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;
}
```
### MIPS Register Naming Conventions

<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>v0 expression evaluation &amp;</td>
</tr>
<tr>
<td>3</td>
<td>v1 function results</td>
</tr>
<tr>
<td>4</td>
<td>a0 arguments</td>
</tr>
<tr>
<td>5</td>
<td>a1</td>
</tr>
<tr>
<td>6</td>
<td>a2</td>
</tr>
<tr>
<td>7</td>
<td>a3</td>
</tr>
<tr>
<td>8</td>
<td>t0 temporary: caller saves</td>
</tr>
<tr>
<td>9</td>
<td>t1</td>
</tr>
<tr>
<td>10</td>
<td>t2</td>
</tr>
<tr>
<td>11</td>
<td>t3</td>
</tr>
<tr>
<td>12</td>
<td>t4</td>
</tr>
<tr>
<td>13</td>
<td>t5</td>
</tr>
<tr>
<td>14</td>
<td>t6</td>
</tr>
<tr>
<td>15</td>
<td>t7</td>
</tr>
<tr>
<td>16</td>
<td>s0 callee saves</td>
</tr>
<tr>
<td>17</td>
<td>s1</td>
</tr>
<tr>
<td>18</td>
<td>s2</td>
</tr>
<tr>
<td>19</td>
<td>s3</td>
</tr>
<tr>
<td>20</td>
<td>s4</td>
</tr>
<tr>
<td>21</td>
<td>s5</td>
</tr>
<tr>
<td>22</td>
<td>s6</td>
</tr>
<tr>
<td>23</td>
<td>s7</td>
</tr>
<tr>
<td>24</td>
<td>t8 temporary (cont’d)</td>
</tr>
<tr>
<td>25</td>
<td>t9</td>
</tr>
<tr>
<td>26</td>
<td>k0 reserved for OS kernel</td>
</tr>
<tr>
<td>27</td>
<td>k1</td>
</tr>
<tr>
<td>28</td>
<td>gp Pointer to global area</td>
</tr>
<tr>
<td>29</td>
<td>sp Stack pointer</td>
</tr>
<tr>
<td>30</td>
<td>fp frame pointer</td>
</tr>
<tr>
<td>31</td>
<td>ra Return Address (HW)</td>
</tr>
</tbody>
</table>

### Nois II Register Naming Conventions

<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></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>temporary: caller saves</td>
</tr>
<tr>
<td>9</td>
<td>t0</td>
</tr>
<tr>
<td>10</td>
<td>t1</td>
</tr>
<tr>
<td>11</td>
<td>t2</td>
</tr>
<tr>
<td>12</td>
<td>t3</td>
</tr>
<tr>
<td>13</td>
<td>t4</td>
</tr>
<tr>
<td>14</td>
<td>t5</td>
</tr>
<tr>
<td>15</td>
<td>t6</td>
</tr>
<tr>
<td>16</td>
<td>callee saves</td>
</tr>
<tr>
<td>17</td>
<td>t7</td>
</tr>
<tr>
<td>18</td>
<td>t8</td>
</tr>
<tr>
<td>19</td>
<td>t9</td>
</tr>
<tr>
<td>20</td>
<td>et exception temp (OS)</td>
</tr>
<tr>
<td>21</td>
<td>bt breakpoint temp</td>
</tr>
<tr>
<td>22</td>
<td>gp pointer to global area</td>
</tr>
<tr>
<td>23</td>
<td>sp stack pointer</td>
</tr>
<tr>
<td>24</td>
<td>fp frame pointer</td>
</tr>
<tr>
<td>25</td>
<td>ea exception return address</td>
</tr>
<tr>
<td>26</td>
<td>ba break return address</td>
</tr>
<tr>
<td>27</td>
<td>ra Return Address (HW)</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**
  - Add the stack <frame size> - 4 to the address in $sp
    - `addiu $fp, $sp, <frame-size> - 4`
MIPS/GCC Procedure Calling Conventions (cont.)

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
  - Add the frame size to $sp.
    ```
    addiu $sp, $sp, <frame-size>
    ```
- Step-4: Return
  - Jump to the address in $ra.
    ```
    jr $ra
    ```

Example 2

```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 # \ Push the stack
    sw $ra, 36($sp) # \ Save return address
    sw $s3, 32($sp) # \
    sw $s2, 28($sp) # > Entry Housekeeping
    sw $s1, 24($sp) # / save registers on stack
    sw $s0, 20($sp) # /
    move $v0, $0 #/ initialize exit code to 0
    move $s1, $0 #\
    la $s0, list # \ Initialization
    la $s2, msg # /
    la $s3, list+36 #/
```

Example2 (cont.)

Main code segment

again:
  Begin main loop
  lw  $t6, 0($s0)  \\ Begin main loop
  addu $s1, $s1, $t6  #/ Actual "work"
  li $v0, 4  \\ SPIM I/O
  move $a0, $s2  # > Print a string
  syscall #/
  li $v0, 1  \\ > Print a number
  move $a0, $s1  # > Print a number
  syscall #/
  li $v0, 4  #\ Print a string (eol)
  la $a0, nln  # > Print a string (eol)
  syscall #/

  addu $s0, $s0, 4  # index update and
  bne $s0, $s3, again  # end of loop

Exit Code

move $v0, $0  \\ Exit Code
lw $s0, 20($sp)  \\
lw $s1, 24($sp)  \\ Closing Housekeeping
lw $s2, 28($sp)  \\ / restore registers
lw $s3, 32($sp)  \\ / load return address
addu $sp, 40  \\ / Pop the stack
jr $ra  \\ / exit(0) ;
.end main  # end of program

Data Segment

.data  # Start of data segment
list: .word 35, 16, 42, 19, 55, 91, 24, 61, 53
msg: .asciiz "The sum is 

lnl: .asciiz \"\n"
Example2 NiosII Code

# Example for CPS 104
# Program to add together list of 9 numbers
.text  # Code
.align 2
.global main

main:  # MAIN procedure Entrance
    subi    sp, sp, 40  #\ Push the stack
    stw    ra, 36(sp)  # \ Save return address
    stw    r19, 32(sp) # \
    stw    r18, 28(sp) # > Entry Housekeeping
    stw    r17, 24(sp) # / save registers on stack
    stw    r16, 20(sp) # /
    mov    r2, zero    #/ initialize exit code to 0
    mov    r17, zero    #\ Initialization
    movia r16, ll     #\
    movia r18, msg    # /
    movia r19, ll+36

again:  # Begin main loop
    ldw r8, 0(r16)     #\ Actual work
    add r17, r17, r8   #/ I/O
    mov r4, r18       # > put format address into arg1
    mov r5, r17       # put number into arg2
    call printf      #/
    addi r16, r16, 4  #\ index update and
    bne r16, r19, again #/ end of loop
Example2 (NiosII cont)

```plaintext
# Exit Code

mov r2, zero       #
ldw r16, 20(sp)    #
ldw r17, 24(sp)    #
ldw r18, 28(sp)    # Closing Housekeeping
ldw r19, 32(sp)    # / restore registers
ldw ra, 36(sp)     # / load return address
addi sp, sp, 40     # / Pop the stack
ret                     # / exit(0) ;

# Data Segment

.data                   # Start of data segment
ll:
.word   35, 16, 42, 19, 55, 91, 24, 61, 53
msg:
.asciz "The sum is %d\n"
.end main
```

Details of the MIPS & Nios II instruction sets

- Register zero always has the value zero
  - even if you try to write it
- Branch and jump 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:
  - logical immediates are zero extended to 32 bits
  - arithmetic immediates are sign extended to 32 bits
- lb and lh extend data as follows:
  - lbu, lhu 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

**Next Time**

- Recursion, Other Instruction Sets

**Reading**

- Ch. 3, Appendix A