Lecture 7: Examples, MARS

• Today’s topics:
  ▪ More examples
  ▪ MARS intro
Saving Conventions

• Caller saved: Temp registers $t0-$t9 (the callee won’t bother saving these, so save them if you care), $ra (it’s about to get over-written), $a0-$a3 (so you can put in new arguments)

• Callee saved: $s0-$s7 (these typically contain “valuable” data)

• Read the Notes on the class webpage on this topic!
Example 2 (pg. 101)

```c
int fact (int n)
{
    if (n < 1) return (1);
    else return (n * fact(n-1));
}
```

Notes:
The caller saves $a0 and $ra in its stack space.
Temp register $t0 is never saved.
Dealing with Characters

• Instructions are also provided to deal with byte-sized and half-word quantities: lb (load-byte), sb, lh, sh

• These data types are most useful when dealing with characters, pixel values, etc.

• C employs ASCII formats to represent characters – each character is represented with 8 bits and a string ends in the null character (corresponding to the 8-bit number 0); A is 65, a is 97
Example 3 (pg. 108)

Convert to assembly:
void strcpy (char x[], char y[])
{
    int i;
    i=0;
    while ((x[i] = y[i]) != `'\0'`)
        i += 1;
}

Notes:
Temp registers not saved.
Large Constants

• Immediate instructions can only specify 16-bit constants

• The lui instruction is used to store a 16-bit constant into the upper 16 bits of a register… combine this with an OR instruction to specify a 32-bit constant

• The destination PC-address in a conditional branch is specified as a 16-bit constant, relative to the current PC

• A jump (j) instruction can specify a 26-bit constant; if more bits are required, the jump-register (jr) instruction is used
Starting a Program

C Program → Compiler → Assembly language program → Assembler → Object: machine language module

Object: library routine (machine language) → Linker → Executable: machine language program → Loader → Memory

x.c → x.s → x.o → x.a, x.so → a.out
Role of Assembler

• Convert pseudo-instructions into actual hardware instructions – pseudo-instrs make it easier to program in assembly – examples: “move”, “blt”, 32-bit immediate operands, etc.

• Convert assembly instrs into machine instrs – a separate object file (x.o) is created for each C file (x.c) – compute the actual values for instruction labels – maintain info on external references and debugging information
Role of Linker

• Stitches different object files into a single executable
  ▪ patch internal and external references
  ▪ determine addresses of data and instruction labels
  ▪ organize code and data modules in memory

• Some libraries (DLLs) are dynamically linked – the executable points to dummy routines – these dummy routines call the dynamic linker-loader so they can update the executable to jump to the correct routine
void sort (int v[], int n)
{
    int i, j;
    for (i=0; i<n; i+=1) {
        for (j=i-1; j>=0 && v[j] > v[j+1]; j-=1) {
            swap (v,j);
        }
    }
}

void swap (int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}

• Allocate registers to program variables
• Produce code for the program body
• Preserve registers across procedure invocations
The swap Procedure

- Register allocation: $a0 and $a1 for the two arguments, $t0 for the temp variable – no need for saves and restores as we’re not using $s0-$s7 and this is a leaf procedure (won’t need to re-use $a0 and $a1)

```assembly
swap:
    sll     $t1, $a1, 2
    add    $t1, $a0, $t1
    lw     $t0, 0($t1)
    lw     $t2, 4($t1)
    sw     $t2, 0($t1)
    sw     $t0, 4($t1)
    jr      $ra
```

```c
void swap (int v[], int k) {
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```
The sort Procedure

- Register allocation: arguments v and n use $a0 and $a1, i and j use $s0 and $s1; must save $a0 and $a1 before calling the leaf procedure

- The outer for loop looks like this: (note the use of pseudo-instrs)

move $s0, $zero          # initialize the loop
loopbody1: bge $s0, $a1, exit1  # will eventually use slt and beq
    ... body of inner loop ...
addi $s0, $s0, 1
j loopbody1

exit1:

for (i=0; i<n; i+=1) {
    for (j=i-1; j>=0 && v[j] > v[j+1]; j-=1) {
        swap (v,j);
    }
}


The sort Procedure

• The inner for loop looks like this:

```
addi $s1, $s0, -1          # initialize the loop
loopbody2: blt $s1, $zero, exit2   # will eventually use slt and beq
        sll $t1,  $s1, 2
        add $t2, $a0, $t1
        lw $t3, 0($t2)
        lw $t4, 4($t2)
        ble $t3, $t4, exit2

        ... body of inner loop ...
addi $s1, $s1, -1
j            loopbody2

exit2:
```
Saves and Restores

- Since we repeatedly call “swap” with $a0 and $a1, we begin “sort” by copying its arguments into $s2 and $s3 – must update the rest of the code in “sort” to use $s2 and $s3 instead of $a0 and $a1

- Must save $ra at the start of “sort” because it will get over-written when we call “swap”

- Must also save $s0-$s3 so we don’t overwrite something that belongs to the procedure that called “sort”
Saves and Restores

sort:  addi $sp, $sp, -20
sw  $ra, 16($sp)
sw  $s3, 12($sp)
sw  $s2, 8($sp)
sw  $s1, 4($sp)
sw  $s0, 0($sp)
move  $s2, $a0
move  $s3, $a1
...
move  $a0, $s2  # the inner loop body starts here
move  $a1, $s1
jal  swap
...
exit1: lw  $s0, 0($sp)
...
addi  $sp, $sp, 20
jr  $ra
MARS

• MARS is a simulator that reads in an assembly program and models its behavior on a MIPS processor

• Note that a “MIPS add instruction” will eventually be converted to an add instruction for the host computer’s architecture – this translation happens under the hood

• To simplify the programmer’s task, it accepts pseudo-instructions, large constants, constants in decimal/hex formats, labels, etc.

• The simulator allows us to inspect register/memory values to confirm that our program is behaving correctly
MARS Intro

- Directives, labels, global pointers, system calls
MARS Intro
MARS Intro

- Read the google doc on the class webpage for details!
Example Print Routine

.data
    str:  .asciiz "the answer is "
.text
    li    $v0, 4               # load immediate; 4 is the code for print_string
    la    $a0, str            # the print_string syscall expects the string
    syscall # MARS will now invoke syscall-4
    li    $v0, 1              # syscall-1 corresponds to print_int
    li    $a0, 5              # print_int expects the integer as its argument
    syscall # MARS will now invoke syscall-1
Example

- Write an assembly program to prompt the user for two numbers and print the sum of the two numbers
Example

.data
  str1: .asciiz "Enter 2 numbers:"
  str2: .asciiz "The sum is 

.text
  li    $v0, 4
  la    $a0, str1
  syscall
  li    $v0, 5
  syscall
  add    $t0, $v0, $zero
  li    $v0, 5
  syscall
  add    $t1, $v0, $zero
  li    $v0, 4
  la    $a0, str2
  syscall
  li    $v0, 1
  add    $a0, $t1, $t0
  syscall
IA-32 Instruction Set

• Intel’s IA-32 instruction set has evolved over 20 years – old features are preserved for software compatibility

• Numerous complex instructions – complicates hardware design (Complex Instruction Set Computer – CISC)

• Instructions have different sizes, operands can be in registers or memory, only 8 general-purpose registers, one of the operands is over-written

• RISC instructions are more amenable to high performance (clock speed and parallelism) – modern Intel processors convert IA-32 instructions into simpler micro-operations
Endian-ness

Two major formats for transferring values between registers and memory

Memory: low address 45 7b 87 7f high address

Little-endian register: the first byte read goes in the low end of the register
  Register: 7f 87 7b 45
  Most-significant bit  Least-significant bit  (x86)

Big-endian register: the first byte read goes in the big end of the register
  Register: 45 7b 87 7f
  Most-significant bit  Least-significant bit  (MIPS, IBM)