Lecture 6: Assembly Programs

• Today’s topics:
  ▪ Procedures
  ▪ Examples
  ▪ Large constants
  ▪ The compilation process
  ▪ A full example
Procedures

- Local variables, AR, $fp, $sp
- Scratchpad and saves/restores, $fp
- Arguments and returns
- jal and $ra
Procedures

• Each procedure (function, subroutine) maintains a scratchpad of register values – when another procedure is called (the callee), the new procedure takes over the scratchpad – values may have to be saved so we can safely return to the caller

  ▪ parameters (arguments) are placed where the callee can see them
  ▪ control is transferred to the callee
  ▪ acquire storage resources for callee
  ▪ execute the procedure
  ▪ place result value where caller can access it
  ▪ return control to caller
Jump-and-Link

- A special register (storage not part of the register file) maintains the address of the instruction currently being executed – this is the *program counter* (PC)

- The procedure call is executed by invoking the jump-and-link (jal) instruction – the current PC (actually, PC+4) is saved in the register $ra and we jump to the procedure’s address (the PC is accordingly set to this address)

  \[
  \text{jal} \quad \text{NewProcedureAddress}
  \]

- Since jal may over-write a relevant value in $ra, it must be saved somewhere (in memory?) before invoking the jal instruction

- How do we return control back to the caller after completing the callee procedure?
The register scratchpad for a procedure seems volatile – it seems to disappear every time we switch procedures – a procedure’s values are therefore backed up in memory on a stack.

Stack grows this way

Proc A’s values
Proc B’s values
Proc C’s values
...

High address
Low address

Proc A

call Proc B
...
call Proc C
...
return
return

return
Saves and Restores
Storage Management on a Call/Return

• A new procedure must create space for all its variables on the stack

• Before/after executing the jal, the caller/callee must save relevant values in $s0-$s7, $a0-$a3, $ra, temps into the stack space

• Arguments are copied into $a0-$a3; the jal is executed

• After the callee creates stack space, it updates the value of $sp

• Once the callee finishes, it copies the return value into $v0, frees up stack space, and $sp is incremented

• On return, the caller/callee brings in stack values, ra, temps into registers

• The responsibility for copies between stack and registers may fall upon either the caller or the callee
int leaf_example (int g, int h, int i, int j) {
    int f;
    f = (g + h) – (i + j);
    return f;
}

leaf_example:
    addi $sp, $sp, -12
    sw $t1, 8($sp)
    sw $t0, 4($sp)
    sw $s0, 0($sp)
    add $t0, $a0, $a1
    add $t1, $a2, $a3
    sub $s0, $t0, $t1
    add $v0, $s0, $zero
    lw $s0, 0($sp)
    lw $t0, 4($sp)
    lw $t1, 8($sp)
    addi $sp, $sp, 12
    jr $ra

Notes:
In this example, the callee took care of saving the registers it needs.

The caller took care of saving its $ra and $a0-$a3.

Could have avoided using the stack altogether.
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
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: `x.c`
- Compiler
- Assembly language program: `x.s`
- Assembler
- Object: machine language module: `x.o`
- Assembler
- Object: library routine (machine language): `x.a, x.so`
- Linker
- Executable: machine language program: `a.out`
- Linker
- Loader
- Memory
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
Full Example – Sort in C (pg. 133)

- Allocate registers to program variables
- Produce code for the program body
- Preserve registers across procedure invocations

```c
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;
}
```
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
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)

```assembly
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:
```

```c
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)
    bgt       $t3, $t4, exit2
    … body of inner loop …
    addi       $s1, $s1, -1
    j          loopbody2
```

```c
for (i=0; i<n; i+=1) {
    for (j=i-1; j>=0 && v[j] > v[j+1]; j-=1) {
        swap (v,j);
    }
}
```
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)
mov      $s2, $a0
mov      $s3, $a1
...
mov      $a0, $s2   # the inner loop body starts here
mov      $a1, $s1
jal       swap
...
exit1:   lw        $s0, 0($sp)
...      addi     $sp, $sp, 20
jr        $ra
Title

• Bullet