CS 237 Meeting 11 — 10/1/12

Announcements

1. None?

Accessing Memory in MIPS Assembly Language

1. At the end of the last class, we examined a simple program showing how the swap method previously shown in C, could be implemented in assembly language. The code for the example is shown in Figure **??**.

   This program illustrates many things you need to know to access memory in a MIPS assembly program.

   - While a program is running, MIPS memory is divided into several segments:

     text This is where the program's code resides. It is pretty big.

     data This is where global variables and string and other constants reside. It is pretty small.

     Dynamic data This is where the function call stack and heap (the place Java puts things when you say "new") live. It is huge.

   - We have previously only used the .data and .text directives at the beginning and end of our programs. Now, we will use them whenever we need to switch between telling the assembler we are describing code or variables/constants that should be stored in the data segment.

     For example, since we want the variable's x and y declared in this program's main function to appear in the data segment rather than on the stack (although local variable's usually would be on the stack), we have to switch back and forth between the text and data segments.

   - This example also illustrates the use of the .word directive which enables us to allocate and initialize a word of memory. We will see there is another directive used to allocate large blocks of memory (as for an array).

```
.text
      j main
#void swap( int *x, int *y ) {  x in $a0,  y in $a1
#  int temp;    temp in $t0
swap:
#  temp = *x;
      lw $t0,($a0)

#  * x = * y;
      lw $t1,($a1)
      sw $t1,($a0)

#  y = * temp;
      sw $t0,($a1)
#}
      jr   $ra
#
#int main( int argc, char * argv[] ) {   argc in $a0,  argv in $a1
#
.data
#  int a = 10;
a:    .word      10
#  int b = 100;
b:    .word      100

.text
#  swap( a, b );
main:      la      $a0,a
      la      $a1,b
      jal      swap
#
#  printf( "a = %d, b = %d\n", a, b );
      li      $v0,4
      la      $a0,aMess
      syscall
      li      $v0,1
      lw      $a0,a
      syscall
# SOME RELATIVELY BORING CODE IS MISSING HERE

.data
aMess:      .asciiz "a = "
bMess:       .asciiz " b = "
newLine: .asciiz "\n"
#}
```

- Before the main method, we associate the labels a and b with these variables. This makes it easy to access them directly. When we want to print their values, we just use their names in lw instructions.

2. This program also shows how to reference a memory location through a pointer rather than a simple name.

   - In the main method, we pass pointers to a and b to swap using the la (load address) instruction.
   - In swap, we have to load these addresses into register and then combine the value in the register with an immediate offset of 0. The notation "($a0)" in the load and store instruction in swap indicate that the address of the memory work to be accessed is in $a0.

Using Arrays in MIPS Assembly Language

1. The load and store instructions are particularly important for array access.
2. As a simple example of this, the program shown in Figure **??** shows how to step through the elements of an array of chars (a string), printing one character per line of output.
   - The address of the string to print is found in the 0th element of the array of pointers to strings passed as argv.
   - Since each character takes one byte, we can access the ith character by adding i to the address of the start of the string.
   - Since we are working with characters, we use lb.
   - The loop ends when it hits a 0 in the array.
3. Things get a bit more interesting when the elements of an array take more than a single byte of memory. In this case, to find an element's address in memory we need to multiply the index value by the element size and then add it to the address of the 0th element.

```
.text
#// Print a string one character per line
#void printDown( char arg[] ) {   &arg in $a0 must be moved to $t0
printDown:
      move $t0,$a0
#  int p;     use $t1 for p

#  for ( p = 0; arg[p]; p++ ) {
      li      $t1,0
printLoopStart:
      add      $a0,$t1,$t0       #use a0 to prep for printf
      lb      $a0,($a0)
      beq      $a0,$0,printLoopEnd

#    printf( "%c\n", arg[p] );
      li      $v0,11
      syscall
      li      $v0,4
      la      $a0,newLine
      syscall

      addi      $t1,$t1,1
      j      printLoopStart
printLoopEnd:
      jr      $ra
#  }
#}
#// This program takes a single argument and prints its text
#// in a vertical column on the screen
#int main( int argc, char * argv[] ) {    argv in $a1
main:
#  printDown( argv[0] );      # print 0th arg for MARS
      lw      $a0,($a1)
      jal      printDown
#}
      li      $v0,10
      syscall

.data
newLine: .asciiz "\n"
```

Figure 2: A MIPS program that prints its argument vertically

- We can often use the shift left logical instruction to do the multiplication since elements sizes are often powers of 2.
- The code for the function used to search for a value in an array of integers in the member.c program from last class provides a nice example of such accesses. Its code is shown in Figure **??**.

4. The code for address calculation for array accesses gets time consuming enough that as programmers it is worth looking for simple shortcuts when producing such code. For example, in the loop from bubble sort that makes a pass through all the elements, it would be silly to calculate the addresses of the two elements being compared separately since we know they will be 4 apart. A way to avoid this is shown in Figure **??**.

Useful Combinational Circuits

1. All of the digital circuits we have seen in class and in lab are examples of combinational circuits — circuits whose output only depend on the current state of the circuit's input.
2. There are a few such circuits that serve as building blocks in many more complicated digital circuits. Our goal today is to talk about a few examples.

Decoders

1. A decode takes an n-bit input and has 2^n outputs. Each output is numbered going from 0 to $2^n - 1$. All of the output are zero except the one whose number equals the binary number encoded by the 1s and 0s on the n input lines.
2. If you think about each of a decoder's output lines separately you will realize that the truth tables for all the lines share an important property. The functions associated with these output lines are 1 for exactly one combination of input values. That is, in sum-of-products form, each output is described by a single product of inputs and their negations. Better yet, for every possible mix of inputs and their negations there is one output of the decoder that corresponds to that combination.

```
#// Determine whether the parameter value can be found in values
#int inList( int val ) {     val in $a0
inList:
#  int p;          use $t0 for p
#
#  for ( p = 0; p < valc; p++ ) {
      li  $t0,0
inListLoopStart:
      slt $t1,$t0,$s0
      beq $t1,$0,inListLoopEnd

#    if ( values[p] == val ) {
      move $t1,$t0
      sll  $t1,$t1,2
      la   $t2,values
      add  $t2,$t1,$t2
      lw   $t1,($t2)
      bne  $t1,$a0,inListIfSkip
#      return TRUE;
      li $v0,1
      jr $ra
#    }
inListIfSkip:
#  }
      addi $t0,$t0,1
      j inListLoopStart
inListLoopEnd:
#  return FALSE;
      li $v0,0
      jr $ra
#}
```

Figure 3: An array search function

```
#  // BUBBLE SORT SWEEP!
#
#  int outOfOrder;     Keep in $s3

#    outOfOrder = FALSE;
li $s3,0

#    for ( p = 0; p < valc - 1; p++ ) {
li $s2,0
sweepLoopStart:
add $t0,$s0,-1
slt $t0,$s2,$t0
beq $t0,$0,sweepLoopEnd

#      if ( values[p] > values[p+1] ) {
move $t0,$s2
sll  $t0,$t0,2
la   $t1,values
add  $t1,$t1,$t0
lw   $t0,($t1)
lw   $t2,4($t1)
slt  $t0,$t2,$t0
beq  $t0,$0,sweepIfJoin

# outOfOrder = TRUE;
li $s3,1

# swap( &values[p], &values[p+1] );
move  $a0,$t1
addi  $a1,$t1,4
jal   swap
#      }
sweepIfJoin:
addi  $s2,$s2,1
j sweepLoopStart
sweepLoopEnd:
```

Figure 4: Main loop of bubble sort