CS 237 Meeting 24 — 11/5/12

Announcements

1. TA applications.

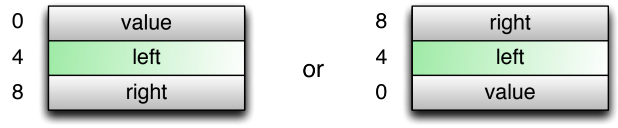
embly Language Programming with Pointers and Structures (conclusion)

1. Last class, we talked about how one might implement programs using pointers and structures in assembly language.

2. As an example, I suggested we look at the tree sort program I showed in C a week or so ago. That program depended on a struct type:

```
typedef struct intTree {
  int value;
  struct intTree *left;
  struct intTree *right;
}
```

3. I suggested that variables of this struct type would be allocated enough contiguous memory to hold all of the components and that the components would be held at fixed offsets from the lowest address in the memory used:



4. I explained that MARS provides a syscall that replaces C's malloc method:

- The syscall number is 9 (i.e. you place 9 in $v0).
- It expects the number of bytes you need in $a0.
- It returns the address of the first byte of the requested memory in $v0.

5. Using this, I sketched out code for the newTree function from the intTree program:

```
#    // Make a new, one-node tree containing a give value
#    intTree newTree( int value ) {       value in $t0
newTree:
move $t0,$a0
#     intTree result =
#           (intTree) malloc( sizeof( struct intTree ) );
li $v0,9
li $a0,12
syscall
#       returns pointer to new space in $v0

#     result->value = value;
sw $t0,0($v0)

#     result->left = result->right = NULL;
sw $0,4($v0)
sw $0,8($v0)
#
#     return result;
jr $ra
#    }
```

6. At the end of last class, I asked you all to spend a little time thinking about how to code one of the other two functions essential to the intTree type: insert and preOrder. Their C code is listed below:

```
// Visit and print the values of all of the nodes in root
void preOrder( intTree root ) {
  if ( root != NULL ) {
    preOrder( root->left );
    printf("%d ", root->value );
    preOrder( root->right );
  }
}
```

```
// Insert a node for the number value in the
// appropriate place in a tree
void insert( intTree *rootPtr, int value ) {
  if ( *rootPtr == NULL ) {
    *rootPtr = newTree( value );
  } else if ( value <= (*rootPtr)->value ) {
    insert( &(*rootPtr)->left, value );
  } else {
    insert( &(*rootPtr)->right, value );
  }
}
```

7. Once you have had time to think about it, we will work together on the board/computer to complete assembly language implementations of these functions.

8. Since class is over by now, here are versions of what the code should look like:

```
#    // Visit and print the values of all of the nodes in root
#    void preOrder( intTree root ) {
# root = s0                 saved at 4(sp)
# return address            saved at 0(sp)
preOrder:
#     if ( root != NULL )
beq $a0,$0,skipPreOrder

addi   $sp,$sp,-8
sw   $ra,0($sp)
sw   $s0,4($sp)
move  $s0,$a0

#        preOrder( root->left );
lw   $a0,4($s0)
jal   preOrder

#        printf("%d ", root->value );
li   $v0,1
lw   $a0,0($s0)
syscall

li   $v0,4
la   $a0,blankSpace
syscall

#        preOrder( root->right );
lw   $a0,8($s0)
jal   preOrder

  lw   $ra,0($sp)
lw   $s0,4($sp)
addi   $sp,$sp,8
#     }

skipPreOrder:
jr   $ra

#    }
```

and

```
#    // Insert a node for the number value in the
#    // appropriate place in a tree
#    void insert( intTree *rootPtr, int value ) {
#   return address saved at 0($sp)
#   rootPtr in $s0 saved at 4($s0)
#   value   in $a1
insert:
addi    $sp,$sp,-8
sw    $ra,0($sp)
sw    $s0,4($sp)
move   $s0,$a0

#      if ( *rootPtr == NULL ) {
```

```
lw     $t0,0($a0)
bne    $t0,$0,if1Skip

#        *rootPtr = newTree( value );
move   $a0,$a1
jal    newTree
sw    $v0,0($s0)
b    returnFromInsert
if1Skip:
#      } else if ( value <= (*rootPtr)->value ) {
lw    $t0,0($s0)
lw    $t0,0($t0)
slt    $t0,$t0,$a1,
bne    $t0,$0,if2Skip

#        insert( &(*rootPtr)->left, value );
lw    $a0,0($s0)
addi    $a0,$a0,4
jal    insert
b    returnFromInsert

#      } else {
if2Skip:
#        insert( &(*rootPtr)->right, value );
lw    $a0,0($s0)
addi    $a0,$a0,8
jal    insert

#      }
returnFromInsert:
lw    $ra, 0($sp)
lw    $s0,4($sp)
addi    $sp,$sp,8
jr    $ra
#    }
```

A Whole New Assembly Language

1. Our final project this semester will involve designing a digital circuit to implement a hypothetical machine language in the same way that the circuits presented recently in class and described in our text implement a subset of the MIPS machine language.

2. The machine language we will implement is a small subset of a hypothetical architecture Duane Bailey designed for 237 a few years ago. It is based on a real architecture known as ARM (which once stood for Acorn RISC Machine but now stands for Advanced RISC Machine).

3. Today, I will give you a brief introduction to the architecture we will be implementing. Duane named his machine "WARM" for Williams Academic RISC Machine. I was thinking of calling our subset of WARM LukeWARM, but Duane suggested TEPID. Out of respect for Duane (and because I could think of a corny interpretation for TEPID but not for LukeWARM), we will call our machine TEPID.

4. ARM[1], WARM, and TEPID all have a lot in common with MIPS.

   - They all are 32 bit machines. To simplify implementation, WARM and TEPID make the 32-bit word the fundamental unit of memory. Their memories are word addressable so that the first word is word 0 and the second is word 1 (rather than word 4 as in MIPS).
   - They are all RISC architectures. WARM and TEPID are in some sense extreme examples since they were stripped down to make for tractable final projects.
   - They are all load/store architectures. The arithmetic instructions of all of these machines can only read and write values in registers. Separate load and store instructions are used to move data between memory and registers.

5. So how do they differ? First there are a few simple differences.

[1]To tell you the truth, I don't really know the ARM architecture very well at this point. With a little luck, I will learn more and we will discuss it in more detail later in the semester.

- WARM and TEPID have a 24-bit limitation on memory addresses and therefore a 16 megabyte limit on the size of the memories they can access (Surprise! Logisim has a 16 megabyte limit on the memories it will simulate.)
- WARM and ARM only have 16 registers instead of 32.
  - They are named r0 through r15 (no dollar signs required).
  - r0 is not equal to 0. Instead, it is typically used for function return values.
  - r0 or r1 through r3 are typically used for function parameters.
  - r4 through r10 are typically used for local variables.
  - r13 is usually used as the stack pointer.
  - r14 is the link register (equivalent to $ra in MIPS).
  - r15 holds the program counter. This is very different from MIPS! Any instruction (including loads or adds) that sets r15 has the effect of taking a branch.

6. There are also several more interesting differences:

7. Conditional execution in WARM/TEPID depends on a set of bits know as condition codes or condition code registers. They record summary information about a recent ALU output:

   Z ZERO - The ALU output was zero.

   N NEGATIVE - The ALU output was negative (the high order bit was 1).

   C CARRY - There was a carry out of the high order digit as part of the last ALU operation.

   V OVERFLOW - The last operation involved an overflow (the carry out was different from the high order bit).

8. Every TEPID/WARM instruction begins with a 4 bit field that determines:

   (a) Whether the instruction should be executed based on the current values of the condition code bits.

   (b) Whether if the instruction executes, the condition code bits should be updated to reflect the result produced by the ALU during its execution.