CS 237 Meeting 23 — 11/2/12

Announcements

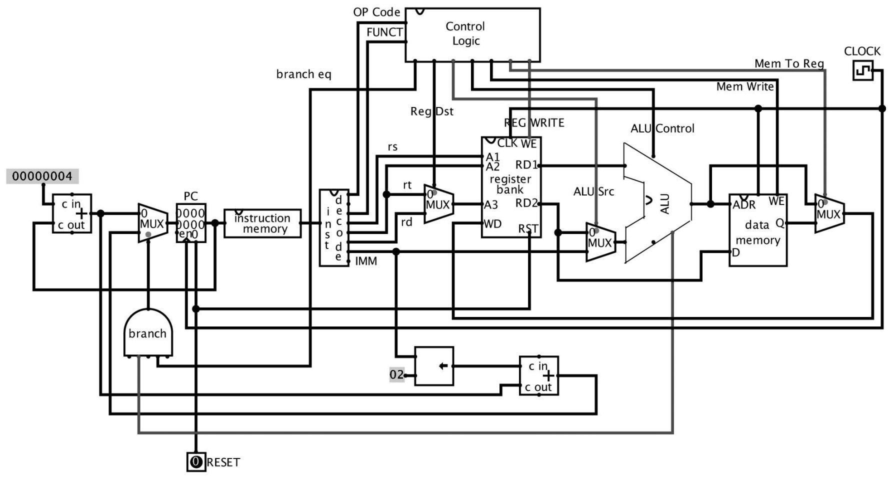
1. Preregistration Pizza at 9 on Thursday.

A Mini-MIPS Microarchitecture

1. Last time, we completed the design of the data flow paths for a circuit that would implement a subset of the MIPS architecture.

2. The subset we will work with includes lw (load word), sw (store word), beq (branch if equal), five of the register to register R-type instructions (add, sub, and, or, sly (set less than)).

3. The complete diagram for the data flow path we developed is:



Control Circuitry

1. The data path we have described requires a significant number of control signals that depend on the bits of the instruction being processed:

brancheq a 1-bit signal that tells whether the current instruction is a branch equal.

RegDst a 1-bit signal that determines whether the second or third 5-bit register field in the encoded instruction should determine the register changed (if any). Basically, this should be 1 for R-type instructions and 0 for I-type.

RegWrite a 1-bit signal that determines whether or not a register should be updated. It should be one for R-type instructions and lw.

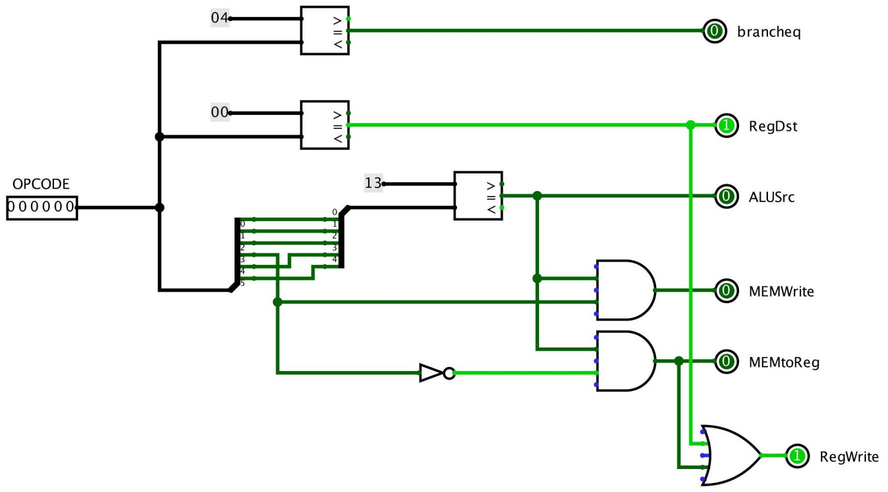
ALUSrc Determines whether the ALU takes its input from the register bank or the 16-bit IMM field of an I-type instruction.

ALUControl a 3-bit field that determines the function performed by the ALU. It depends on both the instruction opcode and if the opcode is 0, then the funct field.

MemWrite set for the sw instruction to indicate that memory should be updated.

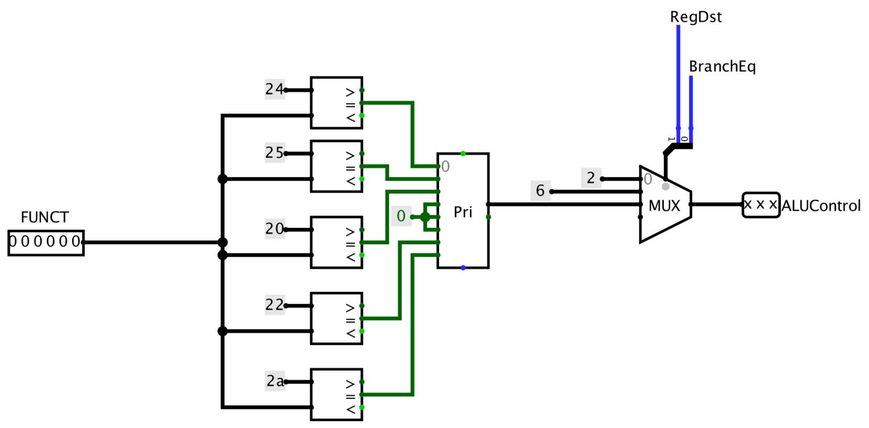
MemToReg Determines whether the value sent to the register bank to update one of its registers should be the output of the ALU or of the data memory. Should be 1 for lw and 0 otherwise.

2. With the exception of the ALUControl, the values of all of these signals is determined completely by the contents of the current instruction's opcode. There are many ways one could derive these signals from the opcode. One circuit to accomplish this is:



- Given the small number of instructions we are trying to support, we have taken a rather simple-minded approach to the design of this circuit.
- We use comparators directly to identify branch equal and R-type instructions.
- We use a comparator to recognize the common bits of the sw and lw instructions.
- We extract the single bit from the opcode that distinguishes lw from sw.
- We add a few and and or gates to the mix and we are done.
- If we were supporting all of the instructions, it might have made sense to use a decoder rather than lots of comparators.

3. The ALUControl signals depend on both the opcode and the funct field of R-type instructions. The dependence of this group of signals on the opcode is limited to knowing whether the opcode is beq or one of the R-type instructions. Therefore, we can build a circuit to determine the settings of ALUControl by giving it as input the funct field, the brancheq signal and the RegDst signal.



Assembly Language Programming with Pointers and Structures

1. The last (and last) topic related to C programming was the use of pointers, structures and dynamic memory allocation in C.
2. Now, we will look briefly at how two of these C language features can be implemented using elements of the MIPS architecture. (We will work on pointers and structs, but give little attention to how dynamic memory allocation really works. That can be a major topic!)
3. Pointers in C are just addresses in assembly language. The C & operator is often implemented using the LA instruction. The * operation is typically just a LW performed using a register holding the value to which the * is applied as a base register.
4. Structs are a bit more interesting. The way we implement a struct is much like the way we handle function stack frames:
   - First we figure out what has to be in our struct and how much memory each item needs (much as we first decide what registers we need to save in a stack frame and what local variables should be allocated there).
   - With information about its contents determined, we can then layout a template for what the memory used to hold all the components of the struct will look like just as we would lay out a stack frame.
   - Then, when we want to access a component of a struct, we will first get the address of the entire area of memory that holds the struct in a register and use that register together with the offset to the component we want to access in an I-Type instruction to access the element of the struct.
5. As an example, consider the simple struct we used to represent tree nodes in our “sort using a binary search tree” example:

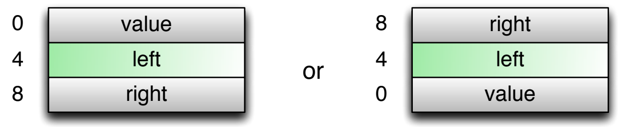
```
typedef struct intTree {
  int value;
  struct intTree *left;
  struct intTree *right;
}
```

6. This structure includes three components. To store an int we usually use 4 bytes. The same is true for pointer. Therefore (and this is not uncommon), each element in this structure needs 4 bytes.

7. Clearly, the total area of memory that must be provided for any variable of this type is 12 bytes.

8. An obvious way to position the components within this 12 bytes is in order, placing value at offset 0, left at offset 4, and right at offset 8.

9. Depending on whether you think memory addresses should increase top-to-bottom or bottom-to-top, you can sketch out the layout of this structure as:



10. To give some examples of how to access components of such a structure and to talk a bit about how dynamic memory allocation can be preformed in an assembly language program, let's consider how to implement the helper function newTree:

```
// Make a new, one-node tree containing a give value
intTree newTree( int value ) {
  intTree result = (intTree)
     malloc( sizeof( struct intTree ) );

  result->value = value;
  result->left = result->right = NULL;
  return result;
}
```

11. First, as we would for any function, we need to think about what registers and stack frame space this function will need.

    - Looking at the C code, newTree appears to call two other functions: malloc and sizeof.
    - sizeof isn't really a function, it is a C operator, and we already know that in this case it just produces the size of an intTree struct which is 12.
    - malloc under MARS is implemented as a system call rather than a function call, so other than \$v0 and \$a0 which will need to be set to make the system call, we don't need to worry about any other registers changing (including \$ra).
    - Therefore, newTree is a leaf function.
    - This means that newTree does not require any stack frame. We can just use temporary (callee-saved) registers to hold important values (include the parameter value).

12. Dynamic allocation is easy. Dynamic reallocation is hard. That is, if all we want to do is give away pieces of a large block of free memory, that is quite easy. If, however, when previously allocated pieces are no longer needed, we want to somehow be able to reuse them to satisfy future allocation requests, things get interesting. We don't want this to get too interesting, so we will just use a feature in the MARS implementation that lets you allocate memory in the heap without any means of reusing it later.

13. MARS accomplishes this by providing a syscall:

    - 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.

14. Once the syscall returns, we can use the value in \$v0 as the base register in instructions designed to reference components of the structure stored in the allocated memory. For example, to set the left pointer to NULL (0), we would say:

```
sw    $0,4($v0)
```

15. Putting all of this together, the complete code for newTree would be:

```
#    // 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
#    }
```

16. To make sure this all makes sense, I would like you to spend a little time here in class 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 );
  }
}
```