CS 237 Meeting 5 — 9/17/12

Announcements

1. Homework 1 is now online.

Quick(?) Review

1. Last time, we examined some simple examples of C code intended to motivate the use of several features of the MIPS architecture. This code included two methods designed to compute the "integer square root" (by which we mean the largest integer less than or equal to the square root) of a number.

   - One version used a linear search:

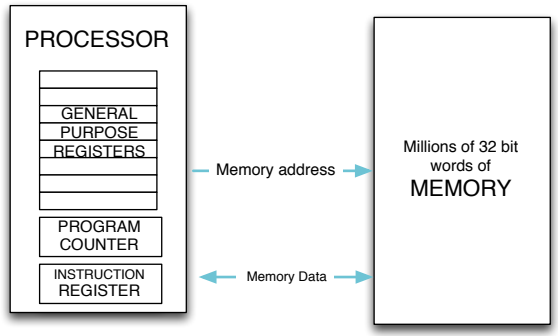
```
int dumbSqrt( int v ) {
  int ans=1;

  while ( ans*ans <= v ) {
    ans = ans + 1;
  }
  ans = ans - 1;
  return ans;
}
```

   - The other mimicked binary search:

```
int smartSqrt( int v ) {

  int lower, upper;
  lower = 1;
  upper = v;

  while ( lower < upper ) {
    int t = lower + upper;
    int mid = t / 2;

    if ( mid > v/mid ) {
      upper = mid-1;
    } else if ( (mid+1) > v/(mid+1) ) {
      lower = upper = mid;
    } else {
      lower = mid + 1;
    }

  }
  return lower;
}
```

2. We also discussed several abstract features of the MIPS and most other architectures:

   - The separation between processor and memory (and the connections between the two).



   - The use of the "fetch/execute" cycle.

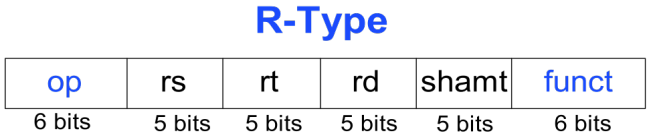
```
PC = address of first instruction
while ( hell has not frozen over ) {
    IR = word at in memory at address stored in PC
    PC = PC + 1;
    execute the operation described by the value in IR
}
```

   - We saw that the bits that were used to encode instructions had to be divided (by those designing the architecture) into subfields used to describe the operation that should be performed when the instruction was executed and the operands that should be used.

- Given how few bits would typically be available to describe the operands we explained that in many architectures, a small memory of "general purpose registers" is included in the processor itself and used to hold all operands. (typically between 8 and 64 of them) is included.
- In addition, we saw that instructions could easily be designed that moved data between the large external memory and the internal collection of registers making it possible to write programs that used memory effectively.

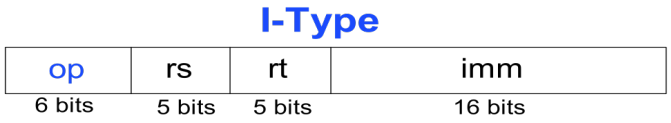
3. Finally, we introduced two of the instruction formats used in the MIPS architecture, several MIPS instructions, and the basics of MIPS assembly language.

   - The MIPS machine uses what it calls R-formal (R for register) to encode instructions that perform operations on pairs of data values stored in registers.



     – An "op" field of 0 tells the machine that this is a three-operand arithmetic instruction. The "funct" field indicates which one (funct = 32 means add, funct = 34 means subtract, etc.)
     – The rs and rt fields are used to encode the numbers of the registers containing the operand values.
     – The rd fields used to encode the number of the register where the answer should be placed.
     – The shamt field is used for instructions that shift the bits of a value and ignored otherwise.

   - Other instructions using a different layout for their 32 bits, are used to tell the processor to copy a value from a particular register to a memory location or to copy a value from a memory location to a register:



     For now, we will not explore the details of how this encoding is used to specify address in memory.
   - The I-Type is also used to encode instructions that involve small constants. For example

     lower = mid + 1;

     might be translated as

     ADDI $s1,$s2,1

     if lower and mid were held in registers $s1 and $s2.

RISC & the MIPS Assembler

1. The designers of the MIPS architecture were dedicated to the idea that keeping a machine's instruction set small and simple would make it easier to build efficient circuits to implement the architecture. With this in mind, they omitted many convenient but unnecessary instructions from the machine.

   - For C/Java instructions like

     upper = v;

     most architectures provide a MOVE instruction that simply copies a value from one register to another.
   - The MIPS designers observed that

     upper = v;

     can be rewritten as

     upper = v + 0;

     and then translated as an ADDI instruction, so they included no MOVE.

- The MIPS assembler allows you to include a MOVE instruction in your code, but it gets translated into an ADD or ADDI adds 0 to the source.

2. MOVE and other instructions supported by the assembler but not included in the architectures are referred to as *pseudo-instructions*.

   - There is no SUBI instruction in the architecture but you can use it as a pseudo-instruction (It gets translated into an ADDI with the constant negated).
   - A very handy instruction is LI which is used to load a constant into a register. For example,

     lower = 1;

     might be translated as

     LI $s3,1

System Calls

1. In a real computer, the operating system controls all access to devices that need to be shared among several programs including the disk, keyboard, and screen.

2. As a result, input/output operations require code that essentially invokes a method/function that is part of the operating systems API.

3. Invocations of operating system functions requires a special mechanism because the operating system code is executed with greater privileges than user code.

4. For this purpose, the MIPS architecture includes a SYSCALL instruction.

   - Oddly, it is encoded as an R-Type instruction even though it takes no operands.
   - Before executing SYSCALL a program must put appropriate values in registers designed by the operating system API to indicate what operation is being requested.

5. MARS, the simulator we will be using to run MIPS programs provides a very simple set of SYSCALL operations (also very similar to those provided by another popular simulator named SPIM). The operation to be performed by each SYSCALL is determined by the value in register $v0.

   Terminate the program SYSCALL with $v0 equal to 10.

   Read an int from the keyboard SYSCALL with $v0 equal to 5. The value read is in $v0 when the SYSCALL completes.

   Write an integer to the screen SYSCALL with $v0 equal to 1. The value to be printed should be in $a0 before the call.

   Write a string to the screen SYSCALL with $v0 equal to 4. The address of the first character of the string should be in $a0 before the call.

Assembler Directives

1. To make all of this concrete, I would like to use MARS to show how to translate some parts of our square root program into MIPS assembly language and run this code. There are a few more details about MIPS and the assembler we need to cover first.

2. You have seen a number of examples of instructions as they might be typed into the assembler. To output a message like "Enter a number:", you also need a way to tell the assembler to encode some text in ASCII (binary) and include in somewhere in the machine's memory.

   - The assembler directive `.asciiz` provides a way to do this.
   - Assembler directives are used like operation codes (ADD, SUB, etc.). They appear first followed by "operands". In this case, the operand is the string to encode.

     ```
     .asciiz "Enter a number:"
     ```

3. The .ascii directive gets a string into memory, but to get a SYSCALL to print the string, you need to get its address into $a0. Two assembler features make this possible:

- You can associate a name with the place that the assembler puts a string or an instruction in memory by preceding the instruction or .asciiz directive with a *label*. A label is simply a name followed by a colon.

4. The memory occupied by a program in the MIPS machines is divided into several *segments*. The *text* segment holds the machine language encoding of the program's code. The *data* segment holds program variables (that are not in registers) and certain constants (like the strings you want to include in messages).

   - Including a .text directive in your code tells the assembler that the items (constants or instructions) that follow should be added to the text segment.
   - Including a .data directive in your code tells the assembler that the items (constants or instructions) that follow should be added to the data segment.

MARS Demonstration

1. With this background, it is time to actually do a little assembly language programming. We will start by translating some code we haven't really looked at much, the main program used to test our square root functions.

```
int main( int argc, char * argv[] ) {
  int number;

  printf( "Enter a number: " );
  scanf( "%d", & number );
  while ( number > 0 ) {
    printf( "The integer square root of %d is %d\n",
                number, smartSqrt( number ) );
    printf( "Enter another number: " );
    scanf( "%d", & number );
  }
}
```

2. I take an odd approach to assembly language programming. (And needless to say, I would like you to take the same approach.) I start by writing the program in a high-level language (or something close to that) and include that in my assembly code file as comments. Then I "refine" the comments into assembly language.

3. The MIPS assembly language comment character is #. So I start by putting a # at the start of every line of the C code for integer square root.

4. Now, since our assembly language knowledge is limited, let us ignore the loop in main and the invocation of the square root function. That is, let us concentrate on the simple input/output for a moment.

   - First, put a .text directive before the area where we will type the code for main.
   - Next, make sure the program terminates nicely by adding code to do a SYSCALL with 10 in register $v0:

```
        li \$v0,10
        syscall
```

   - After this, put a directive to switch to the data segment and define a string constant for the enter prompt:

```
        .data

enterPrompt:
        .asciiz  "Enter a number:"
```

   - Place the code needed to print this message under the comment for the first printf. Note that if you cannot remember the details of the SYSCALL to print a string you can find them in the MARS help window.

```
#  printf( "Enter a number: " );
        li  $v0,4
        la  $a0,enterPrompt
        syscall
```

5. This isn't quite the entire program, but let us try it anyway.

- First press the button with the little hand tools icon to tell the assembler to try to translate your code into machine language. If there are typos, fix them.
- Once the program assembles correctly, press the green arrow button to run the program. The prompt should appear and the program should immediately terminate.

Branches, Jumps, Loops, and Conditionals

1. Most of the rest of the code for main is a lot like what we just typed in. It requires several SYSCALLs to print prompts and results, and one to input a value. Rather than type that all in now, I have a version of the program that has complete code for main, except for handling the loop.

   - The most interesting part of this code from our perspective is the handling of the variable "number".
   - First, we have to decide where we will keep the value of number. This decision is recorded in a comment (number = $s0). This comment does not help the assembler, but it will help us.
   - Next, note that when we read in a value for number, it arrives in $v0 (following the SYSCALL API) so we have to move it to $s0.
   - Similarly, when we want to print its value or pass it to the square root function, we have to move the value of number to the appropriate location.

2. Completing the loop requires knowledge of three additional MIPS instructions

   j label Go to the instruction that follows label. (J stands for jump.)

   beq $rd,$rt, label Go to the instruction that follows label if the values in the register operands are equal.

   bne $rd,$rt, label Go to the instruction that follows label if the values in the register operands are not equal.

3. To use these instruction effectively, we must add two labels to our code:

   - One before the first instruction of the loop.
   - One after the last instruction (which will eventually be a j).