Foothill College
Computer Science Department
CS10 Computer Architecture and Organization
Lab # 4

System Calls, Flow Control Structures and Shift Instructions

1 Introduction

This week we will look in more depth at some of the MIPS programming basics that we have been working with thus far.

For instance, we will extend our use of system calls to obtain user input. We will also extend our understanding of the MIPS instructions that change the flow of execution, allowing us to perform decisions and loops. We will consider how to translate high-level Java-esque pseudo-code into assembly code. Finally we will look in more depth at creative ways to exploit the shift instruction.

Before you can proceed to do this lab, be sure that you have fully picked up all the material from the last two lab sessions.

2 System Calls

A CPU by itself can do things like arithmetic and logical operations, read/write memory etc. There are no instructions, however, to perform high-level I/O like printing or reading values from the keyboard. To make this happen, we need some software which does all the hard work for us: libraries and the operating system.

To access the operating system, all CPUs provide a way of performing system calls. On the MIPS, the *syscall* instruction does this. We have to load the $v0 register with a number which indicates what type of operation is required, and put any argument into the $a0 register. Here is a list of the main system calls simulated by MARS.

Syscall	$v0	Args	Result
print_int	1	$a0 = integer	
print_float	2	$f12 = float	
print_double	3	$f12 = double	
print_string	4	$a0 = string	
read_int	5		integer in $v0
read_float	6		float in $f0
read_double	7		double in $f0
read_string	8	$a0 = buffer, $a1 = len	
sbrk	9	$a0 = amount	address in $v0

exit	10		

For now, we are interested mainly in the *print_int*, *print_string*, *read_string* and *read_int* system calls.

2.1 Task 1

Download the program **w4syscalls.asm** and load it into MARS. This performs the following pseudo-code:

```
int age;
print_string("Please tell me how old you are: ");
age= read_int();
print_string("You told me that your age is ");
print_int(age);
```

Note that both *print_int* and *print_string* require an argument in $a0 which is the thing to print. For strings, we pass the base address of the string using the *la* (load address instruction). The *read_int* syscalls returns the integer typed by the user as a twos-complement word in $v0. Also note that we had to copy the age entered by the user from $v0 into $t0, or it would be clobbered on the next system call.

Play around with the program: run it, modify it. Make sure you understand how to perform these system calls.

3 Branches and Jumps

The key thing that makes computers useful is *decision-making*: based on the values of data, do alternative operations, or loop back (or not) based on the data values. For high-level languages like C or Java, we have control structures like IF .. ELSE, FOR, WHILE and DO .. WHILE.

At the CPU, there are more primitive constructs. Based on the values of data, we can:

- branch forward or backward to a new instruction, not just the instruction immediately following this one; and
- jump to a specific instruction far away, and then jump back to where we came from.

3.1 Constructing IF .. ELSE Statements

Let's see how we can synthesize IF .. ELSE, FOR, WHILE and DO using branches. We will start with a high-level IF .. ELSE:

```
# Print error message if user's age is negative: high-level version
int age;
print_string("Please tell me how old you are: ");
age= read_int();
if (age >= 0) {
   print_string("You told me that your age is ");
   print_int(age);
```

```
    } else {
       print("Silly user, that number is too small");
    }
```

Here is how we can do it with branches and labeled instructions:

```
                # Print error message if user's age is negative: branching
version
                int age;

    main:       print_string("Please tell me how old you are: ");
                age= read_int();
                branch to else_clause if (age < 0);
                print_string("You told me that your age is ");
                print_int(age);
                branch always to end_if;
    else_clause:
                print("Silly user, that number is too small");

    end_if:     # Rest of the program
```

Note the two branch operations. Here's how they work:

1. Age is OK: the comparison at the first branch fails, so we drop into the good code. When that is finished, we have to branch past the *else_clause* code to continue the rest of the program.
2. Age is bad: the comparison is true, we branch past the good code to the *else_clause* code, print out the error message, and continue on with the rest of the program.

Note that the comparison in the first branch instruction is **OPPOSITE** the test in the high-level IF statement: we have to branch to skip the good code! Note also the branch to stop us dropping into the else_clause.

3.2 Constructing WHILE Loops

With loops, we have to loop backwards as well as forwards. With a WHILE loop, we decide to keep going or break out of the loop at the top, and also always loop backwards at the bottom. Consider this program to print out the numbers from 1 to 20.

```
    # Example of a while loop: high-level version
    int i;
    i=1;
    while (i <= 20) {
      print_int(i);
      i++;
    }
```

Now here is the same loop written with branches and labels.

```
                # Example of a while loop: branching version
                int i;

                i=1;
    top_of_loop:
                branch to end_of_loop if (i > 20);
                print_int(i);
```

```
                i++;
                branch always to top_of_loop;
end_of_loop: # Rest of the program
```

What you can see is that we are making explicit the flow of control which is only implicitly shown in the WHILE loop. Again note that the branch condition is the opposite of the high-level condition, as the branch is breaking us out of the loop.

3.3 Constructing DO .. WHILE Loops

DO .. WHILE loops always enter the loop at least once, and they only branch backwards, so we only need one branch instruction. Here is a DO .. WHILE loop to print out the numbers from 1 to 20.

```
# Example of a do .. while loop: high-level version
int i;
i=1;
do {
  print_int(i);
  i++;
} while (i <= 20);
```

Now here is the same loop written with branches and labels.

```
                # Example of a do .. while loop: branching version
                int i;

                i=1;
top_of_loop:
                print_int(i);
                i++;
                branch to top_of_loop if (i <= 20);
end_of_loop: # Rest of the program
```

3.4 Constructing FOR Loops

FOR loops are just WHILE loops repackaged, with the loop initializer and loop modifier written next to the loop decision. So, here is the FOR version to print out the numbers from 1 to 20.

```
# Example of a while loop: high-level version
int i;
for (i=1; i <= 20; i++) {
  print_int(i);
}
```

Unsurprisingly, the loop written with branches and labels is the same as the WHILE loop.

```
                # Example of a while loop: branching version
                int i;

                i=1;
top_of_loop:
                branch to end_of_loop if (i > 20);
                print_int(i);
                i++;
                branch always to top_of_loop;
end_of_loop: # Rest of the program
```

4 MIPS Branch Instructions

Now that you have got the concept of branching down, let's see some branches in real life. The list of basic MIPS branch instructions is given below.

Instruction	Result	Comment
beq Rs, Rt, label	Branch to label if Rs == Rt	
beqz Rs, label	Branch to label if Rs == 0	
bge Rs, Rt, label	Branch to label if Rs >= Rt	
bgeu Rs, Rt, label	Branch to label if Rs >= Rt	Unsigned comparison
bgez Rs, label	Branch to label if Rs >= 0	
bgt Rs, Rt, label	Branch to label if Rs > Rt	
bgtu Rs, Rt, label	Branch to label if Rs > Rt	Unsigned comparison
blt Rs, Rt, label	Branch to label if Rs < Rt	
bltu Rs, Rt, label	Branch to label if Rs < Rt	Unsigned comparison
bltz Rs, label	Branch to label if Rs < 0	
bne Rs, Rt, label	Branch to label if Rs != Rt	
bnez Rs, label	Branch to label if Rs != 0	
b label	Branch to label always	

4.1 Task 2

Download the program **w4whileloop.asm** and load it into MARS. This is a MIPS program to print out the numbers from 1 to 20. Read through and understand how it works. Note the two loop labels: one at the top of the loop where the break-out decision is made, and one immediately after the end of the loop branch back Also note that the second argument to the branch instructions can be a small integer like 20: the assembler converts this pseudo-instruction into a few real instructions.

4.2 Task 3

Modify the above program to print out some other ranges, to print backwards etc.

4.3 What about multiple loops?

Question: if you need multiple loops, multiple IF .. ELSE statements in your assembly program, you can't re-use the labels **loop:**or **end_of_loop:** etc., so what labels should be used?

Compilers, which translate high-level languages down to assembly code, typically give each label a number: L1, L2, L3, L4 etc., and output the assembly code to match the labels. For us humans, it makes sense to use descriptive labels while making sure that each label is unique.

5 Shift Left, Shift Right

If you take a bit pattern, say 0011, and shift the bits to the left by one (i.e. put a new 0 on the right, discard the leftmost bit), you get 0110. In Java and C, the << operator is the left-shift operator: `a << b` means shift the bits in *a* left by *b* columns.

Converting to decimal, we started with 3_{10} and ended up with 6_{10}. Shifting left by one bit doubles the value. Shifting left by two bits will multiply the value by 4 etc. Shifting right has the opposite property: it divides the value by powers of two, discarding the remainder.

5.1 Task 5

Download and load the **w4.shift.asm** program. Run it and see that 3 is doubled to make 6. Modify the program to try some different numbers and some different values. Also try the *srl* (shift right logical) instruction as well as the *sll* (shift left logical) instruction.

5.2 Shift Right Logical, Shift Right Arithmetic

The srl instruction always puts a zero bit on the left when shifting right. This is fine for unsigned or positive numbers, but it doesn't work for signed negative numbers: the left-most sign bit which is 1 is shifted right and replaced with a 0. The negative number becomes a positive one!

Try loading a negative number like -10 into $t0, and then doing `srl $a0, $t0, 1` which should shift -10 right by 1 and make -5. You will see that the result becomes positive. The *sra* (shift right arithmetic) instruction does a shift right, but preserves the sign bit: if it was 1, the new left bit is a 1. This preserves the sign of the result. Try it out and see if it works.

5.3Multiplication Using Shifts

We have not carefully considered multiplication yet. One way to do this is with left shifts.

Consider multiplying a value by 10:

value * 10 == (value * 8) + (value * 2)

This can be accomplished using left shifts:

value * 10 == (value <<3) + (value <<1)

5.3 Task 6

Write a program to multiply 5 * 10 using shifts. Use the algorithm shown above to do this setting *value* to the number 5.

6 Status Check and Some Exercises

At this point in your learning of assembly language, you have seen:

- the use of registers and main memory.
- basic assembly instruction for add.
- the basic data sizes, and the signed/unsigned difference.
- branch instructions using several comparisons.
- the equivalence of branching and high-level constructs such as IF .. ELSE, DO .. WHILE, WHILE and FOR.

This is about the same amount of knowledge as half of our introductory programming subject in Java. At this point, you should take stock of what you have learned and reinforce it. There are several ways to do this:

- read back through the example programs so far.
- write as many assembly program as you can. The only way to learn a language is to write code in it!

7 Outlook for the Next Lab

In the next lab, we will look at several MIPS addressing modes: immediate, direct, indirect and indexed addressing.