Tutorial 11

Part 1. Passing arguments on the stack

For Assignment 5, you will pass arguments on the stack using the same convention that the "C" compiler uses. This is useful to know to be able to call assembly functions from "C" programs (but we won't be doing that in this course).

Arguments will be passed using the stack. The assignment specifies that the stack should be set up in the following way before the mult32 procedure is called:

$sp

0	multiplicand
4	multiplier
8	&productH
12	&productL

How is this done?

1) Allocate memory for productH and productL:

```
            .data
prodH:      .word 0
prodL:      .word 0
```

2) Now, assuming that multiplicand is in $4 and multiplier in $5, can setup stack in function call (this is done in main):

```
            la   $6, prodH
            la   $7, prodL
            addi $sp, $sp, -16
            sw   $4, 0($sp)
            sw   $5, 4($sp)
            sw   $6, 8($sp)
            sw   $7, 12($sp)

            jal  mult32

            addi $sp, $sp, -16
```

3) Then, in the mult32 procedure, can get the arguments from the stack - but have to do this *after* we save the context, as follows:

```
mult32:   addi $sp, $sp, -36  # save context
          sw   $18, 0($sp)
          .
          .
          .
          sw   $fp, 32($sp)

          addi $fp, $sp, 36   # fp points to arguments

          lw   $18, 0($fp)    # Get multiplier
          lw   $19, 4($fp)    # Get multiplicand
```

4) Then, still in mult32, once we've computed the product, we can put it directly into memory by getting the memory addressed off of the stack. So, the following will appear at the end of mult32, assuming that prodH is computed and stored in $20, and prodL is computed and stored in $21:

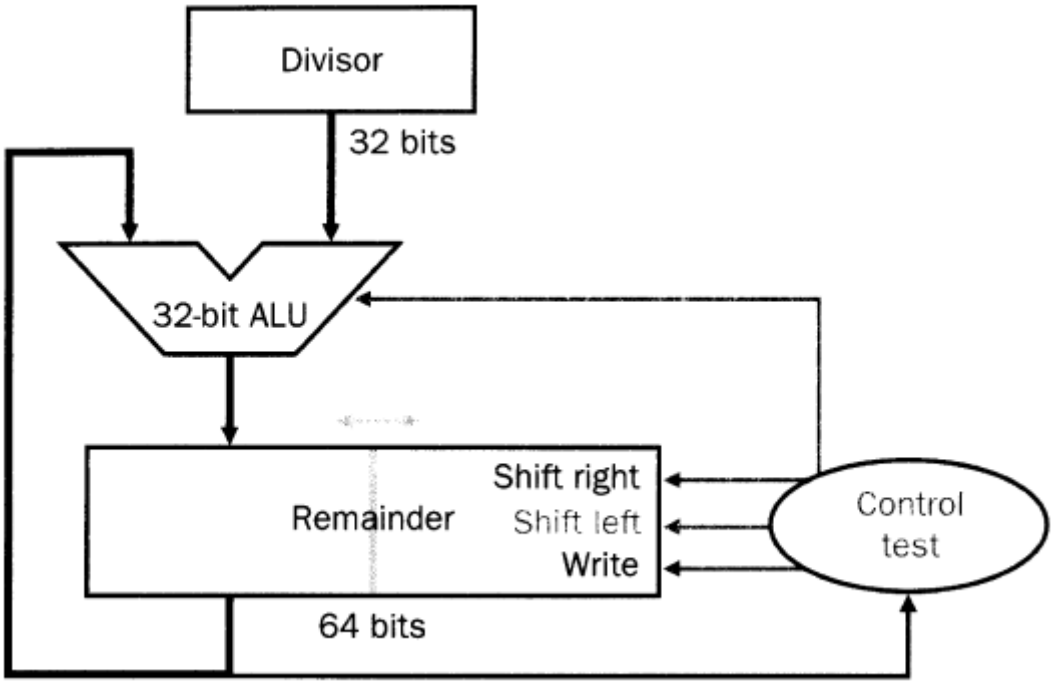
```
          lw   $23, 8($fp)    # Get address for prodH
          sw   $21, 0($23)    # Store prodH in memory
          lw   $23, 12($fp)   # Get address for prodL
          sw   $21, 0($23)    # Store prodL in memory
```

5) Finally, need to restore context at very end of mult32:

```
          lw   $18, 0($sp)
          .
          .
          .
          lw   $fp, 32($sp)
          addi $sp, $sp, 36
```

Part 2. Division

Division Datapath



Division Algorithm

0. **Init.**
   Place divisor in the Divisor register.
   Place dividend in the Remainder register.

1. **Shift**.
   Shift Remainder register to the left 1 bit.

2 **Subtract**.
   Subtract Divisor register from the upper half of Remainder register and leave result there.

3a. **If the result is zero or positive**.
   Shift Remainder register left setting righmost bit to 1.

3b. **If the result is negative strictly.**
   Restore initial value by adding Divisor register to the upper part of the Remainder register leaving the sum there. Shift Remainder register left setting rightmost bit to 0.

4. **Count.**
   If 32nd repetition then **done** else goto step 2.

Done. Shift upper part of Remainder register right 1 bit.

Non-Restoring Division

The division algorithm above is inefficient because it can involve both an addition *and* a subtraction each iteration. NRD is designed to do division more efficiently by only doing one addition or subtraction each iteration.

In regular division, if subtracting the divisor from the remainder results in a negative number, the usual practice is to add back the divisor and shift the remainder register left. Then, on the next iteration, we subtract the divisor again. Combining these steps, and remembering that multiplying by two is equivalent to shifting left, the result of all these operations is:

```
(remainder + divisor) x 2 – divisor
```

multiplying it out we get:

```
2 x remainder + 2 x divisor – divisor
```

which gives:

```
2 x remainder + divisor
```

So, we can skip the adding back (restore) step if we simply add instead of subtract on the NEXT iteration.

Non-Restoring Division Algorithm

0. **Init.**
   Place divisor in the Divisor register.
   Place dividend in the Remainder register.
   Set Sign to 0.

1. **Shift**.
   Shift Remainder register to the left 1 bit.

2a. **If Sign = 0:**
   Subtract Divisor register from the upper half of Remainder register and leave result there.

2b. **If Sign = 1:**
   Add Divisor register to the upper half of Remainder register and leave result there.

3a. **If the result is zero or positive**.
   Shift Remainder register left setting righmost bit to 1.
   Set sign to 0.

3b. **If the result is negative strictly.**
   Shift Remainder register left setting rightmost bit to 0.
   Set sign to 1.

4. **Count.**
   If 32nd repetition then **done** else goto step 2.

Done. Shift upper part of Remainder register right 1 bit.

Non-Restoring Division Example Program

See div32nrd.s in the Resources section of WebCT.

This example program is useful for the assignment since it has examples of:

- doing a 64-bit shift (using two 32-bit registers)
- passing arguments using the stack
- correctly saving and restoring context

However, it is also more complicated than the program you'll be writing since:

- it has to deal with signed numbers, but you don't need to worry about signs (you're doing unsigned multiplication)
- it does I/O on the console (you're iterating through a table of inputs)