Assignment #5 Key

(max = 95)

Read pages 178-196 in the *Computer Organization and Design* text. I have provided a set of notes ("Notes for Assignment #5) on this reading that can be found under Course Notes. Please refer to these notes as you carefully work through the assigned reading.

Afterwards, submit answers for the following problems:

1. Multiply 10_{10} by 11_{10} (the multiplier) using the hardware of Figure 3.3. Produce a table similar to Figure 3.6. As the text has done, use 4-bit (unsigned) numbers, rather than 32-bit numbers! (10 points)

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	1011	0000 1010	0000 0000
1	1a: 1 => Prod = Prod + Mcand	1011	0000 1010	0000 1010
	2: Shift left Multiplicand	1011	0001 0100	0000 1010
	3: Shift right Multiplier	0101	0001 0100	0000 1010
2	1a: 1 => Prod = Prod + Mcand	0101	0001 0100	0001 1110
	2: Shift left Multiplicand	0101	0010 1000	0001 1110
	3: Shift right Multiplier	0010	0010 1000	0001 1110
3	1: 0 => No operation	0010	0010 1000	0001 1110
	2: Shift left Multiplicand	0010	0101 0000	0001 1110
	3: Shift right Multiplier	0001	0101 0000	0001 1110
4	1a: 1 => Prod = Prod + Mcand	0001	0101 0000	0110 1110
	2: Shift left Multiplicand	0001	1010 0000	0110 1110
	3: Shift right Multiplier	0000	1010 0000	0110 1110

I have underlined the bit that is used for the 0/1 check in the Step 1 instance that immediately follows. The answer is in the Product register: $01101110_2 = 110_{10}$.

2. This time, multiply 11_{10} by 12_{10} (the multiplier). Use the refined version of the hardware given in Figure 3.5, producing a table similar to the one that appears in the course notes. Use 4-bit (unsigned) numbers. (10 points)

Iteration	Step	Multiplicand	Product
0	Initial values	1011	0 0000 1100
1	1b: 0 => No operation	1011	0 0000 1100
	2: Shift Product right	1011	0 0000 0110
2	1b: 0 => No operation	1011	0 0000 0110
	2: Shift Product right	1011	0 0000 0011
3	1a: 1 => Prod = Prod + Mcand	1011	0 1011 0011
	2: Shift Product right	1011	0 0101 1001

4	1a: 1 => Prod = Prod + Mcand	1011	1 0000 1001
	2: Shift Product right	1011	0 1000 0100

I have underlined the bit that is used for the 0/1 check in the Step 1 instance that immediately follows. The answer is in the bottom 8 bits of the Product register: $10000100_2 = 132_{10}$.

3. Divide 14_{10} by 3_{10} using the hardware of Figure 3.8. Produce a table similar to Figure 3.10 (use my slightly modified algorithm that starts with Step 3 for the first iteration). Use 4-bit (unsigned) numbers. (10 points)

Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0011 0000	0000 1110
1	3: Shift Div right	0000	0001 1000	0000 1110
2	1: Rem = Rem – Div	0000	0001 1000	1111 0110
	2b: Rem<0 => +Div,sll Q,Q_0=0	0000	0001 1000	0000 1110
	3: Shift Div right	0000	0000 1100	0000 1110
3	1: Rem = Rem – Div	0000	0000 1100	0000 0010
	2a: Rem≥0 => sll Q,Q_0=1	0001	0000 1100	0000 0010
	3: Shift Div right	0001	0000 0110	0000 0010
4	1: Rem = Rem – Div	0001	0000 0110	1111 1100
	2b: Rem<0 => +Div,sll Q,Q_0=0	0010	0000 0110	0000 0010
	3: Shift Div right	0010	0000 0011	0000 0010
5	1: Rem = Rem – Div	0010	0000 0011	1111 1111
	2b: Rem<0 => +Div,sll Q,Q_0=0	0100	0000 0011	0000 0010
	3: Shift Div right	0100	0000 0001	0000 0010

I have underlined the bit that is used for the negative check in the Step 2 instance that immediately follows. The resulting quotient is 4, and the remainder is 2.

4. Divide 14_{10} by 3_{10} again. This time use the improved non-restoring version of the division algorithm. Produce a table like the one that appears in the course notes. Use 4-bit (unsigned) numbers. (10 points)

Iteration	Step	Divisor	Remainder
0	Initial values	0011	0 0000 1110
1	1: Rem = Rem – Div	0011	1 1101 1110
	2a: Rem<0 => sll R,R_0=0,+Div	0011	1 1110 1100
2	2a: Rem<0 => sll R,R_0=0,+Div	0011	0 0000 1000
3	2b: Rem≥0 => sll R,R_0=1	0011	0 0001 0001
4	1: Rem = Rem – Div	0011	1 1110 0001
	2a: Rem<0 => sll R,R_0=0,+Div	0011	1 1111 0010
5	2a: Rem<0 => +Div,sll R,R_0=0	0011	0 0100 0100

I have underlined the bit that is used for the negative check in the Step 2 instance that immediately follows. Notice that the Step 2a in the 5th iteration is using the "last cycle" part of our revised algorithm. The final quotient is the bottom 4 bits of the Remainder register: $0100_2 = 4_{10}$. The final remainder is the next 4 bits of the Remainder register, but only after doing a right shift of 1 bit: $0010_2 = 2_{10}$.

5. Consider the following sequence, which I'll refer to as the alternating Fibonacci sequence:

   1 -1 2 -3 5 -8 13 …

   Here $altfib_1 = 1$, $altfib_2 = -1$ and $altfib_n = altfib_{n-2} – altfib_{n-1}$ for $n > 2$. Write a MIPS program (call it **altfib.s**) that will produce and print numbers (5 per line) in the alternating Fibonacci sequence in such a way that the code detects when overflow takes place. The "offending" number should not be in your list of numbers, but you should display the bogus value that is produced [see my output; the next value in the list would have been 1134903170 – (-1836311903) = 1134903170 + 1836311903 = 2971215073, which is too large for a 32-bit 2's complement number; instead, it is interpreted as -1323752223]. You should use the elaboration on page 182 as a guide, but notice that you will need to alter things slightly since you are taking the difference of two numbers, not the sum. You might want to (carefully) use the "negu" instruction on page A-54. Here is output from my program:

```
Here are the alternating Fibonacci numbers that I produced:

1 -1 2 -3 5
-8 13 -21 34 -55
89 -144 233 -377 610
-987 1597 -2584 4181 -6765
10946 -17711 28657 -46368 75025
-121393 196418 -317811 514229 -832040
1346269 -2178309 3524578 -5702887 9227465
-14930352 24157817 -39088169 63245986 -102334155
165580141 -267914296 433494437 -701408733 1134903170
-1836311903

Value causing overflow = -1323752223
```

   Don't forget to document your code! Submit a separate file called **altfib.s** as well as placing your code in this assignment submission; the Mentor will clarify what I mean by this. (50 points)

```
# Stephen P. Leach -- 10/29/15
# altfib.s - a simple program that tests for overflow by producing and printing
#       the alternating sequence of Fibonacci numbers until overflow takes place
# Register use:
#       $a0     parameter for syscall
#       $v0     syscall parameter
#       $t0     temporary calculation
#       $s0     number of values printed thus far on current line
#       $s1     altfib(n-2)
#       $s2     altfib(n-1)
```

```
#       $s3     altfib(n) = altfib(n-2) - altfib(n-1)


main:   la      $a0, intro       # print intro
        li      $v0, 4
        syscall

        li      $s0, 2           # count number of values on a line
        li      $s1, 1           # altfib(n-2)
        li      $s2, -1          # altfib(n-1)

loop:   negu    $t0, $s2         # $t0 is -altfib(n-1)
        addu    $s3, $s1, $t0    # $s3 is altfib(n) = altfib(n-2) - altfib(n-1)
        xor     $t0, $s1, $t0    # if signs of operands differ,
        slt     $t0, $t0,$zero
        bne     $t0, $zero, ok   # no overflow is possible
        xor     $t0, $s3, $s1    # if signs of operands are the same,
        slt     $t0, $t0,$zero
        bne     $t0,$zero,done   # sign of sum must match the operands, or overflow

ok:     move    $a0, $s3         # print value
        li      $v0, 1
        syscall

        la      $a0, space       # and space afterwards
        li      $v0, 4
        syscall

        move    $s1, $s2         # shift values for next iteration
        move    $s2, $s3

        addi    $s0, $s0, 1      # we have now printed one more number on the line

        slti    $t0, $s0, 5
        bne     $t0,$zero,loop   # if five numbers have been printed,

        la      $a0, cr          # go to the next line
        li      $v0, 4
        syscall

        move    $s0, $zero       # and start counter back at zero

        j       loop             # next iteration

done:   beq     $s0,$zero,nocr   # if partial line exists,

        la      $a0, cr          # go to the next line
        li      $v0, 4
        syscall

nocr:   la      $a0, bad         # show value that caused the overflow
        li      $v0, 4
        syscall

        move    $a0, $s3
        li      $v0, 1
        syscall

        la      $a0, cr
        li      $v0, 4
        syscall

        li      $v0, 10          # exit from the program
        syscall

        .data
intro:  .asciiz "Here are the alternating Fibonacci numbers that I produced:\n\n1 -1 "
space:  .asciiz " "
bad:    .asciiz "\nValue causing overflow = "
cr:     .asciiz "\n"
```

6. Recall that in question 9 in Assignment #2, we displayed the values of fact(n) for various values of n. We saw that you could only go up to a certain value of n and still expect to get a valid result. We also saw that eventually (as the value of n increased) the values being returned were simply zero. Now that you know how multiplication works, explain both of these phenomena (incorrect result and zero

result). There is a way that you could have predicted the first value of n that would produce a result of zero. Explain that process. (5 points)

Recall that the last "correct" value that we produced was fact(12) with a value of 479001600_{10}, or $0001\ 1100\ 1000\ 1100\ 1111\ 1100\ 0000\ 0000_2$. The value for fact(13) should be 13 * fact(12), which is 6227020800_{10} or $1\ 0111\ 0011\ 0010\ 1000\ 1100\ 1100\ 0000\ 0000_2$. But this binary number has 33 bits, and we know now that the product retains only the bottom 32 bits (it is somewhat an accident that the number even appears to be positive … since bit 31 is a zero); this explains why the value displayed by our program for fact(13) was 1932053504_{10}, or $0111\ 0011\ 0010\ 1000\ 1100\ 1100\ 0000\ 0000_2$. As we continue to compute values beyond fact(13), it also explains why they are wrong and why some appear as negative numbers (I have never explicitly told you this, but the syscall that prints an integer <u>assumes</u> that the integer is two's complement, not unsigned).

Moving on to why the factorials eventually become zero, notice that if two binary numbers end in a "1" (i.e., they are odd integers), their product will also end in "1". But any other combination of numbers (even/odd, odd/even, even/even) will have one or more zeros on the right end. In fact, if the two binary numbers have m and n zeroes on the end, the product will have m + n zeroes. As an example, using smaller numbers

```
     0000 0000 0011 0110
x    0000 0000 1001 1100
     -------------------------
     0010 0000 1110 1000
```

Notice that the answer has three zeroes on the right … the sum of the number of zeroes for the two operands. It isn't hard to see why that is the case. So, as you compute larger and larger factorials, the zeroes on the right mount up until it reaches 32; when that happens, the product will be zero, since it only keeps the rightmost 32 bits.

How can we tell exactly which factorial will be the first to become zero? Let's list the values from 1_{10} to 34_{10}, indicating how many "zeroes on the right" each number has (when represented in binary). I will only show the even numbers, since all odd numbers end in a '1' and have no "zeroes on the right".

2 – 1	4 – 2	6 – 1	8 – 3	10 – 1	12 – 2	14 – 1	16 – 4	18 – 1
20 – 2	22 – 1	24 – 3	26 – 1	28 – 2	30 – 1	32 – 5	34 – 1	

If you add up these "zeroes on the right", you get exactly 32; so when the value of fact(34) is computed the result will be zero. Notice that the number of "zeroes on the right" for fact(32) would be 31, so the binary representation for that number would have to be

1000 0000 0000 0000 0000 0000 0000 0000.

This is the “largest” negative number, -2147483648_{10}. You should verify that this is, in fact, what is produced by our fact program when 32 is used as the input.

Your assignment is due by 11:59 PM (Eastern Time) on the assignment due date (consult Course Calendar on course website).