Jon Crawford

Assignment #5

(max = 95)

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	0 0000 0000
1	1a: 1 => Prod = Prod + Mcand	1011	0000 1010	0 0000 1010
	2: Shift Left Multiplicand	1011	0001 0100	0 0000 1010
	3: Shift Right Multiplier	0101	0001 0100	0 0000 1010
2	1b: 1 => Prod = Prod + Mcand	0101	0001 0100	0 0001 1110
	2: Shift Left Multiplicand	0101	0010 1000	0 0001 1110
	3: Shift Right Multiplier	0010	0010 1000	0 0001 1110
3	1a: 0 => No operation	0010	0010 1000	0 0001 1110
	2: Shift Left Multiplicand	0010	0101 0000	0 0001 1110
	3: Shift Right Multipler	0001	0101 0000	0 0001 1110
4	1b: 1 => Prod = Prod + Mcand	0001	0101 0000	0 0110 1110
	2: Shift Left Multiplicand	0001	1010 0000	0 0110 1110
	3: Shift Right Multiplier	0000	1010 0000	0 0110 1110

The answer is in the bottom 8 bits of the Product register: $0110\ 1110_2 = 110_{10}$

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

The answer is in the bottom 8 bits of the Product register: $1000\ 0100_2 = 132_{10}$

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 1110
	2b: Rem<0 => +Div,sll Q,Q0=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,Q0=1	0001	0000 1100	0000 0010
	3: Shift Div Right	0001	0000 0110	0000 0010
4	1: Rem = Rem – Div	0001	0000 0110	1111 0010
	2b: Rem<0 => +Div,sll Q,Q0=0	0010	0000 0110	0000 0010
	3: Shift Div Right	0010	0000 0011	0000 0010
5	1: Rem = Rem – Div	0010	0000 0011	1111 0010
	2b: Rem<0 => +Div,sll Q,Q0=0	0100	0000 0011	0000 0010
	3: Shift Div Right	0100	0000 0001	0000 0010

The answer is in the Quotient and Remainder registers: 0100_2 R $0010_{10} = 4_{10}$ R 2_{10}

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,R0=0,+Div	0011	1 1110 1100
2	2a: Rem<0 => sll R,R0=0,+Div	0011	0 0000 1000
3	2b: Rem>=0 => sll R,R0=1	0011	0 0001 0001
4	1: Rem = Rem – Div	0011	1 1110 0001
	2a: Rem<0 => sll R,R0=0,+Div	0011	1 1111 0010
5	2a: Rem<0 => +Div,sll R,R0=0	0011	0 0100 0100

The quotient is in the bottom of Remainder register 0100_2. After shifting the remainder one bit to the right we get 0010_2 : 0100_2 R $0010_{10} = 4_{10}$ R 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)

   "MIPS code begin next page"

```
# Jon Crawford -- 4/2/18
# altfib.s - A simple program to print the alternating
positive and negative value from the fibonacci
#     sequence, print the values 5 per line, while testing
for overflow of the next value, printing
#     the offending overflow value on the screen and
terminating the sequence.

# Register use:
#     $a0   parameter for syscall
#     $v0   syscall parameter
#     $s0-1 used for n, n-1 of sequence
#     $s2-3 used for loop index range [0,1,2,3,4]
#     $t0-2 temporary use for calculation

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

      li    $s2, 0                # int index = 0;
      li    $s3, 4                # int max = 4;

      li    $s0, 0                # int n-1 = 0;
      li    $s1, 1                # int n = 1;


loop: move  $a0, $s1              # print n value
      li    $v0, 1
      syscall

      li    $a0, 32               # print ASCII Space
      li    $v0, 11
      syscall

      negu  $t1, $s1              # get n opposite

      addu  $t0, $s0, $t1         # get altfib value

      xor   $t2, $s0, $t1         # check signs
      slt   $t2, $t2, $zero
      bne   $t2, $zero, ok        # signs pass, continue
      xor   $t2, $t0, $s0         # check altfib sign
      slt   $t2, $t2, $zero
      bne   $t2, $zero, bad       # signs fail, overflow


ok:   move  $s0, $s1              # move n into n-1
      move  $s1, $t0              # save altfib to n

      addi  $s2, $s2, 1           # ++index;

      blt   $s3, $s2, car         # if (index == max) "\n"

      j loop
```

```
car:  la   $a0, cr              # carriage return
      li   $v0, 4

      li   $s2, 0               # int index = 0;
      syscall

      j loop


bad:  la   $a0, overflow        # print overflow msg
      li   $v0, 4
      syscall

      move $a0, $t0             # print overflow value
      li   $v0, 1
      syscall

exit: li   $v0, 10              # exit
           syscall

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

"End MIPS Code"

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)

   N! returns the incorrect answer at 13! Because there is no room left and the product is so big it has entered the 33rd bit causing an overflow and skewing the return values.

   N! returns all zeroes after 33! because there is no room left for bits to be values anymore they are all 0s up to the top. If you look at the chart below the trailing 0s are keeping pace with the iteration number.

N	Iteration	N!	trailing zeroes
1	1	1	0
10	2	10	1
11	3	110	1
100	4	1 1000	3
101	5	111 1000	3
110	6	10 1101 0000	4
111	7	1 0011 1011 0000	4
1000	8	1001 1101 1000 0000	6
1001	9	101 1000 1001 1000 0000	7
1010	10	11 0111 0101 1111 0000 0000	8
1011	11	10 0110 0001 0001 0101 0000 0000	8
1100	12	1 1100 1000 1100 1111 1100 0000 0000	10
1101	13	1 0111 0011 0010 1000 1100 1100 0000 0000	10