Homework9

1. We saw in class that the text's original algorithm for division can determine when there's been division by 0, since the quotient will overflow. Apply the improved algorithm for division to the 2-bit binary division 01/11. Show the contents of each of the registers at the start of execution and after each step of the algorithm. Does the improved algorithm detect overflow? If so, how does it detect this?

Dividend = 01
Divisor = 11

Remainder(4 bits)

Rem(2 bits)	Quot (2 bits)

Using 2-bit binary division:

	Step	Divisor	Dividend	Rem	Quot	Remainder
Start	Initial value	**011**	**01**	**00**	**01**	**00010**
Step0	1:Rem -= Divisor	011	01	**101**	**10**	**10110**
	2(a):Rem<0:Rem += Divisor	011	01	**000**	10	**00010**
	2(b):remainder <<= 1	011	01	**001**	10	**00100**
Step1	1:Rem -= Divisor	011	01	**110**	**00**	**11000**
	2(a):Rem<0:Rem += Divisor	011	01	**001**	00	**00100**
	2(b):remainder <<= 1	011	01	**010**	**00**	**01000**
End loop	Srl Rem	011	01	**001**	**00**	**00100**

So remainder = 1, quotient = 0

The improved algorithm cannot detect overflow.

2. Using a table similar to that shown in Figure 3.10, calculate 74 divided by 21 using the hardware described in Figure 3.8. You should show the contents of each register on each step. Assume both inputs are unsigned 6-bit integers.

74(octal) = 111100(bin)———>Dividend
21(octal) = 010001(bin)———>Divisor

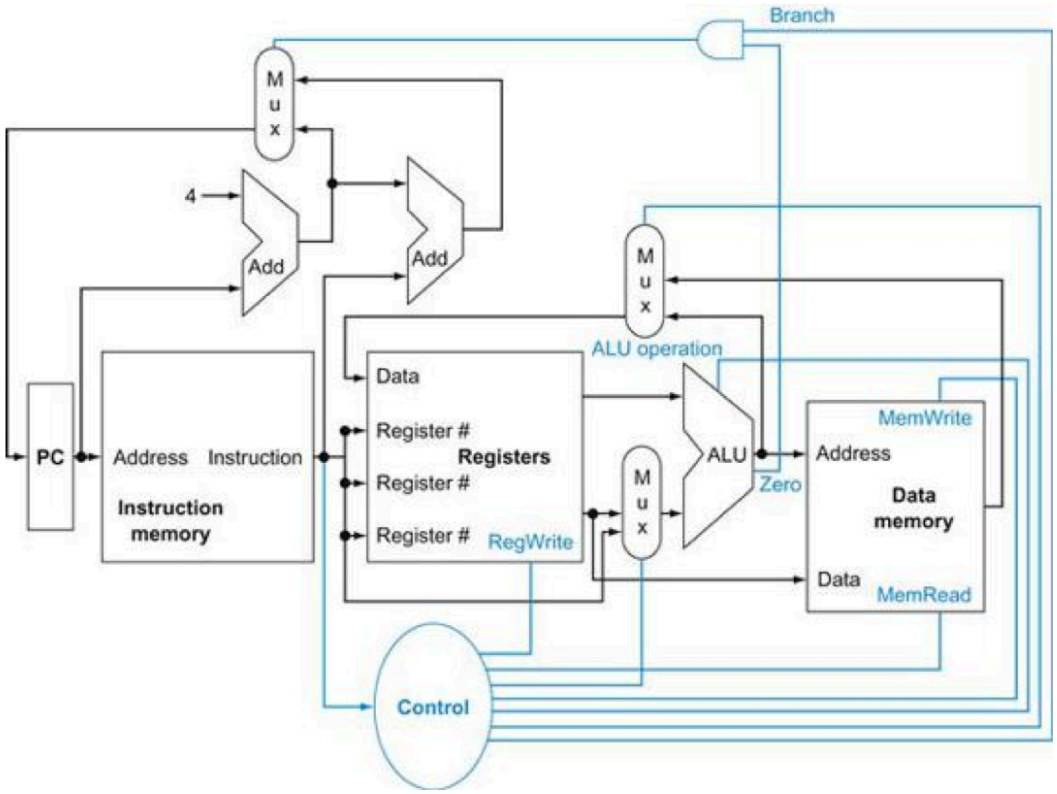
	Step	Divisor	Quotient	Remainder
Start	Initial values	010001 000000	000000	000000 111100
1	1： Rem -= Div	010001 000000	000000	**101111 111100**
	2b:Rem<0 -> +Div, sll Q, Q0=0	010001 000000	**000000**	**000000 111100**
	3:srl Div	**001000 100000**	000000	000000 111100
2	1： Rem -= Div	001000 100000	000000	**111000 011100**
	2b:Rem<0 -> +Div, sll Q, Q0=0	001000 100000	**000000**	**000000 111100**
	3:srl Div	**000100 010000**	000000	000000 111100
3	1： Rem -= Div	000100 010000	000000	**111100 101100**
	2b:Rem<0 -> +Div, sll Q, Q0=0	000100 010000	**000000**	**000000 111100**
	3:srl Div	**000010 001000**	000000	000000 111100
4	1： Rem -= Div	000010 001000	000000	**111110 110100**
	2b:Rem<0 -> +Div, sll Q, Q0=0	000010 001000	**000000**	**000000 111100**
	3:srl Div	**000001 000100**	000000	000000 111100
5	1： Rem -= Div	000001 000100	000000	**111111 111000**
	2b:Rem<0 -> +Div, sll Q, Q0=0	000001 000100	**000000**	**000000 111100**
	3:srl Div	**000000 100010**	000000	000000 111100
6	1： Rem -= Div	000000 100010	000000	**000000 011010**
	2a:Rem>=0->sll Q, Q0 =1	000000 100010	**000001**	000000 011010
	3:srl Div	**000000 010001**	000001	000000 011010
7	1： Rem -= Div	000000 010001	000001	**000000 001001**
	2a:Rem>=0->sll Q, Q0 =1	000000 010001	**000011**	000000 001001
	3:srl Div	**000000 001000**	000011	000000 001001

So 74(octal)/21(octal),
quotient = 11(bin)= 3(octal)
Remainder = 1001(bin) = 11(octal)

3. Write a C program that determines whether the C-compiler on the lab machines generates code that uses the "Mathematicians'" method or "Some Computer Scientists'" method for finding quotients and remainders. Include in your documentation which method is used.

```c
/* File: 3.c
 * Purpose: determines whether the C-compiler on the lab machines
generates code
 *          that uses the "Mathematicians'" method or "Some Computer
Scientists'" method.
 * Compile: gcc -g -Wall -o 3 3.c
 * Run:     ./3
 * Input: None
 * Output: whether "It's Some Computer Scientists'" or "It's
Mathematicians'"
 */


#include <stdlib.h>
#include <stdio.h>

const int MAX = 50;

int main( ) {
    int x = -7;
    int y = 2;
    int rem = x / y;

    char array1[50] = "It's Some Computer Scientists'";
    char array2[50] = "It's Mathematicians'";

    if (rem < 0)
        printf("%s\n", array1);
    else
        printf("%s\n", array2);

    return 0;
}
```

4.1 Consider the following instruction:
Instruction: AND Rd,Rs,Rt
Interpretation: Reg[Rd] = Reg[Rs] AND Reg[Rt]”



(1) What are the values of control signals generated by the control in Figure 4.2 for the following instruction?"

AND is an ALU Operation, so:

a. RegWrite: true
b. BottomMux: register
c. MidMux: ALU
d. ALUoperation: AND
e. MemRead: false
f. MemWrite: false
g. Branch: false

(2) Which resources (blocks) perform a useful function for this instruction?

There are PC, Instruction memory, Registers, ALU, Bottom MUX, Middle MUX, Top MUX

(3) Which resources (blocks) produce outputs, but their outputs are not used for this instruction? Which resources produce no outputs for this instruction?

There are branch, Data memory

4.2 The basic single-cycle MIPS implementation in Figure 4.2 can only implement some instructions. New instructions can be added to an existing Instruction Set Architecture (ISA), but the decision whether or not to do that depends, among other things, on the cost and complexity the proposed addition introduces into the processor datapath and control. The first three problems in this exercise refer to the new instruction:
Instruction: LWI Rt,Rd(Rs)
Interpretation: Reg[Rt] = Mem[Reg[Rd]+Reg[Rs]]

(1) Which existing blocks (if any) can be used for this instruction?

There are PC, Registers, Bottom MUX, ALU, Data memory,Top MUX, Middle MUX.

(2) Which new functional blocks (if any) do we need for this instruction?

There’s no new functional blocks.

(3) What new signals do we need (if any) from the control unit to support this instruction?

There’s no new signals