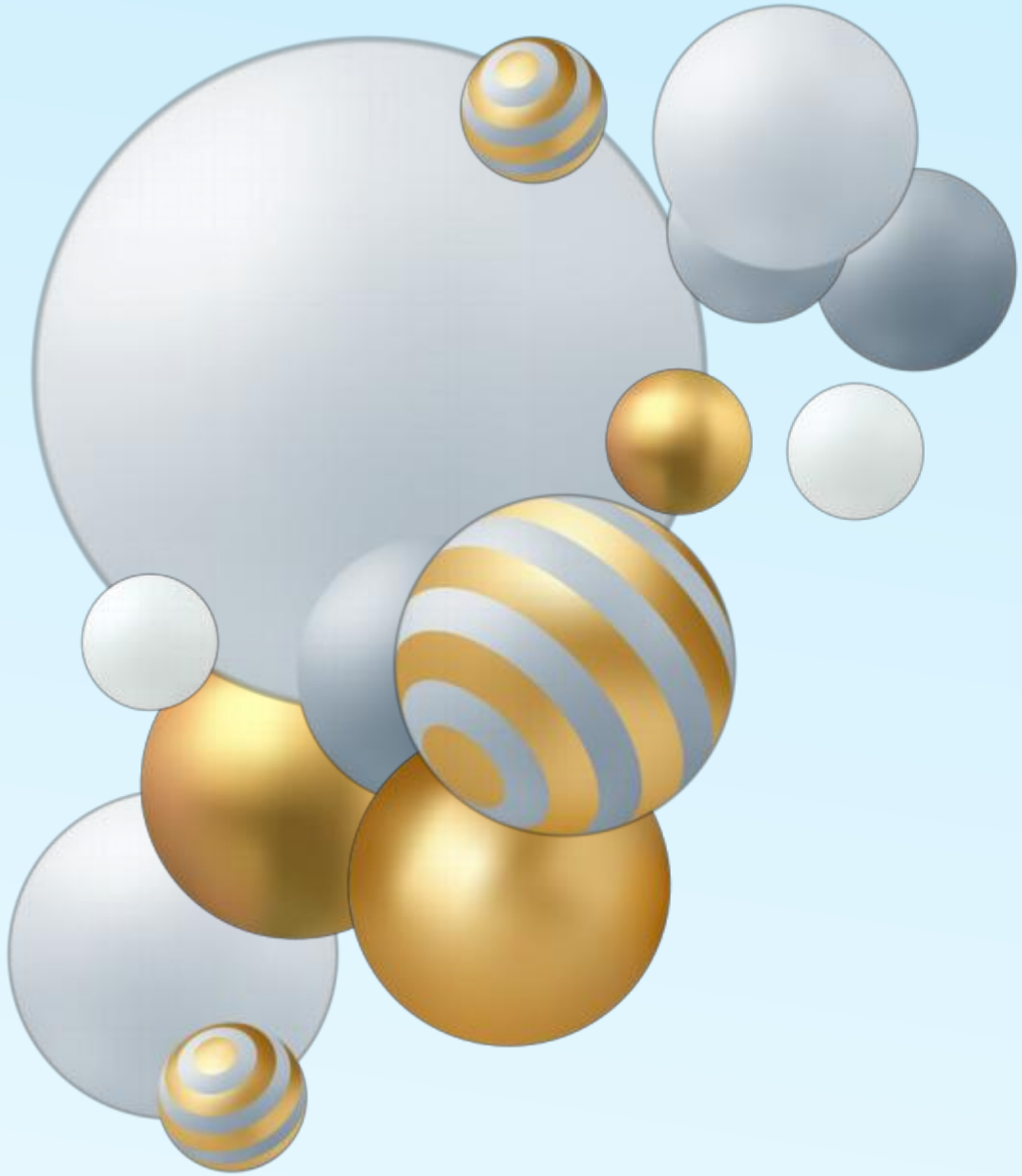


Computer Organization

Lab6 Integer Arithmetic

Integer Arithmetic; Exception

Topic

- Arithmetic
  - Adder (practice 1-1)
  - Subtraction (practice 1-2)
  - Multiplication (practice 2-1)
  - Division (practice 2-2)

Tools: Vivado, Mars

Adder with overflow detector(1)

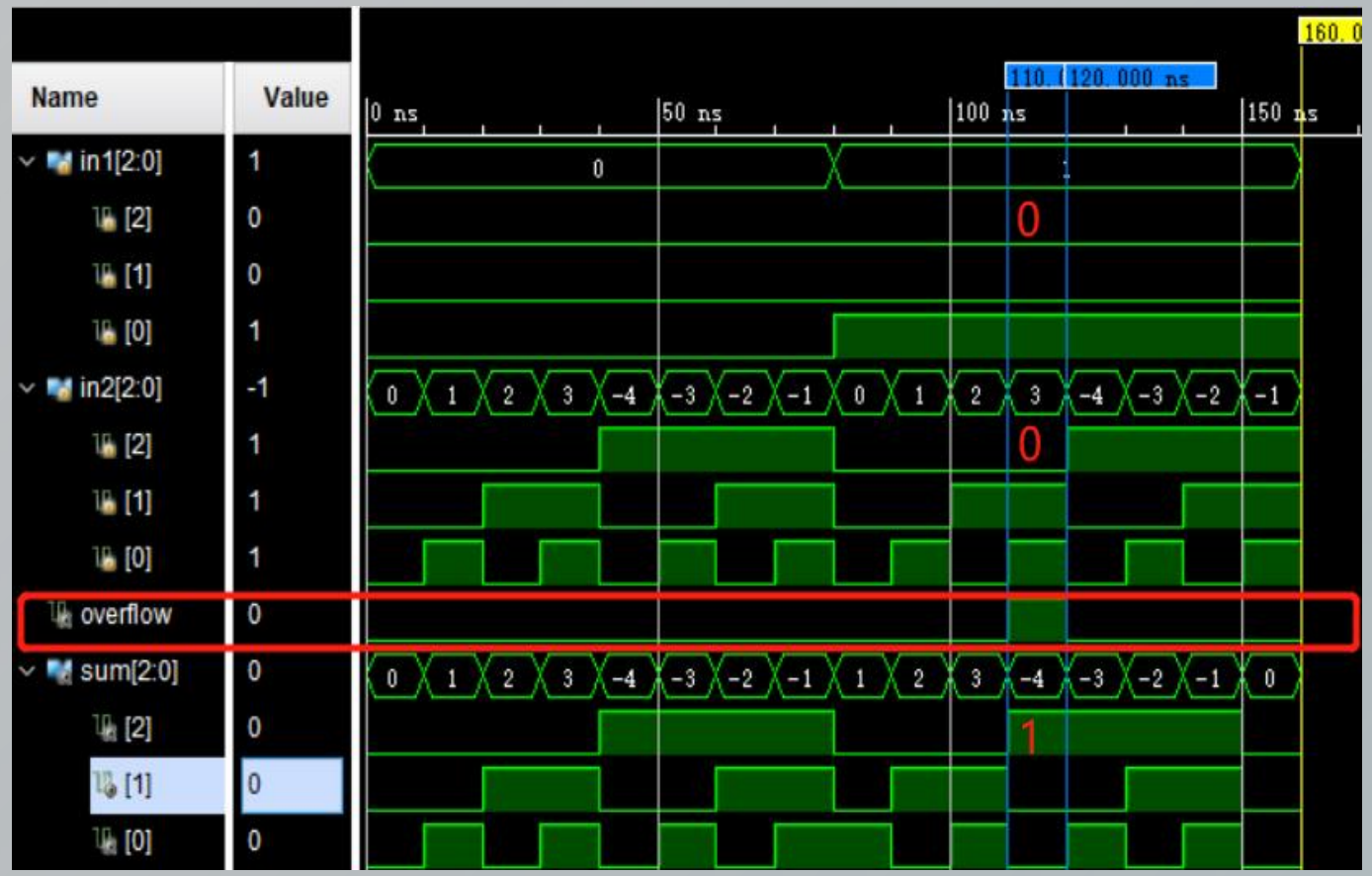
The rule about overflow if result out of range on adder

- no overflow, if adding +ve and -ve operands
- **overflow, if**
  - **adding two +ve operands, get -ve operand .e.g 001 + 011 = 110**
  - **adding two -ve operands, get +ve operand .e.g 101 + 101 = (1) 010**

```
module adder (in1,in2,sum,overflow);                    //in verilog
input [2:0]in1,in2;
output [2:0] sum;
output overflow;

assign sum = in1 + in2;
assign overflow =
 (  in1[2] &   in2[2] & ~sum[2] )    |    /* two +ve operands, get -ve operand*/
 (~in1[2] & ~in2[2] &   sum[2] );        /* two -ve operands, get +ve operand*/

endmodule
```

Adder with overflow detector(2)

Here is the waveform of the circuit ‘add’:In1 is the addend and in2 is the addend. while the value of overflow is 1’b1, it means there is a overflow, otherwise means not.



From 110ns to 120ns of the waveform on the left hand:

the **in1** is **3’b001** and **in2** is **3’b011**, the **sum** is **3’b100**

The **signed bit** of **in1** and **in2** is **0** (means they are both **+ve**), the **singed bit** of **sum** is **1**(means it is **-ve**)

In this situation, an **overflow** is detected !

Adder with overflow detector(practice1-1)

Please complete the testbench to finding all of the legal combinations of the two inputs，verify the function of the circuit “adder”.
A testbench on the bottom is for the reference.

```
module adderTb( );     //verilog
reg [2:0] in1,in2;
wire overflow;
wire [2:0] sum;

adder  ua(in1,in2,sum,overflow);

initial begin
   {in1,in2} = 6'b0;
   repeat(15) #10 {in1,in2} = {in1,in2} + 1;
   #10 $finish;
end

endmodule
```

Subtraction(1)

Implement the **subtraction** with **adder**:
add **negation of second operand**

- How to get the **negation of a number?**

  Which of the following option(s) is(are) right?

  - Option1:

    step1: Invert the sign bit. e.g. **vin2p1 = ~in2[2]**

    step2: add 1 after inverting the value bits.

    e.g. **~in2[1:0] + 1**

  - Option2:

    Inverting the bits and adding on1

    e.g. **~in2+1**

```
module subO1(in1,in2,result);    //verilog
input [2:0] in1;
input [2:0] in2;
wire vin2p1;
wire [1:0] vin2p2;
output [2:0] result;
assign vin2p1 = ~in2[2];
assign vin2p2 = ~in2[1:0] + 1;
assign result = in1 + {vin2p1,vin2p2};
endmodule
```

```
module subO2(in1,in2,result);    //verilog
input [2:0] in1;
input [2:0] in2;
output [2:0] result;
wire [2:0] vin2;
assign vin2= ~in2 + 1;
assign result = in1 + vin2;
endmodule
```

Subtraction(2)

Verify the function of the circuit '**subO1**', '**subO2**', Which implement(s) of sub is(are) correct?

```
module subO1(in1,in2,result);    //verilog
input [2:0] in1;
input [2:0] in2;
wire vin2p1;
wire [1:0] vin2p2;
output [2:0] result;
assign vin2p1 = ~in2[2];
assign vin2p2 = ~in2[1:0] + 1;
assign result = in1 + {vin2p1,vin2p2};
endmodule
```

```
module subO2(in1,in2,result);    //verilog
input [2:0] in1;
input [2:0] in2;
output [2:0] result;
wire [2:0] vin2;
assign vin2= ~in2 + 1;
assign result = in1 + vin2;
endmodule
```

```
module subTb( );          //verilog
reg [2:0] in1,in2;
wire [2:0] rO1, rO2;
subO1  usubO1(in1,in2,rO1);
subO2  usubO2(in1,in2,rO2);

initial begin
   {in1,in2} = 6'b0;
   $monitor( "%3b-%3b: ro1 = %3b(%d), ro2 =
%3b(%d)",
in1,in2,rO1,$signed(rO1) ,rO2,$signed(rO2) );
   repeat(63)   #10 {in1,in2} = {in1,in2} + 1;
   #10 $finish();
end

endmodule
```

TIPs:
$monitor is a system service in **verilog**, which is valid only in **simulation**. It monitor the datas: whenever any of them changes, it prints the datas in the specified format.

%3b: means print the data in **binary**, the bitwidth is **3**
%d: means print the data in **decimal**

$signed is a system service in verilog, which change the data tobe signed value.

Subtraction with overflow detector(practice1-2)

Please complete the circuit to dectecte the overflow of the subtraction.
Build a testbench to verify the function of the circuit.

The description about the overflow of the subtraction is described as bellow:

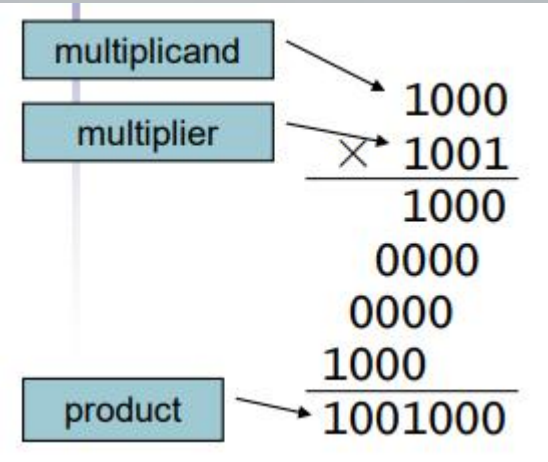
- Overflow if result out of range
  - No overflow, if subtracting two +ve or two –ve operands
  - Overflow, if:
    - Subtracting +ve from –ve operand, and the result sign is 0 (+ve)
    - Subtracting –ve from +ve operand, and the result sign is 1 (-ve)

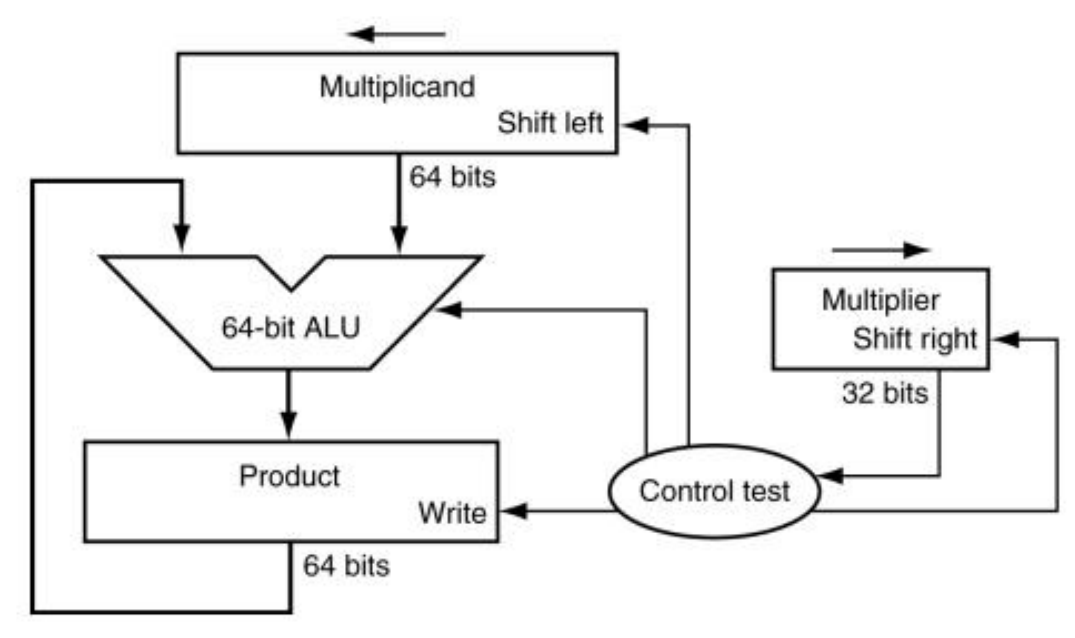
```
//verilog
module subtraction(in1,in2,result,overflow);

input [2:0]in1,in2;
output [2:0] result;
output overflow;

assign sum = in1 - in2;
assign overflow = _____;

endmodule
```

Multiplication(1)

Here is a digital circuit which implement the **long-multiplication** approach:

- Shift registers for Multiplicand and Multiplier
  - store and shift
- **Adder** with two inputs and a control signal
  - add or not
- A register to store the product
  - when to get the data from the product register ?
- Any clk , rst or other signals?



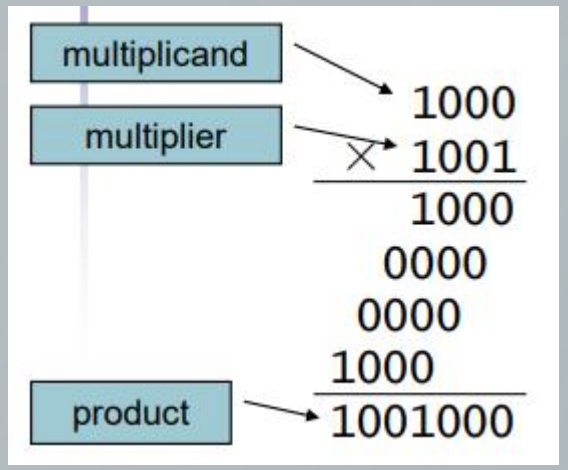


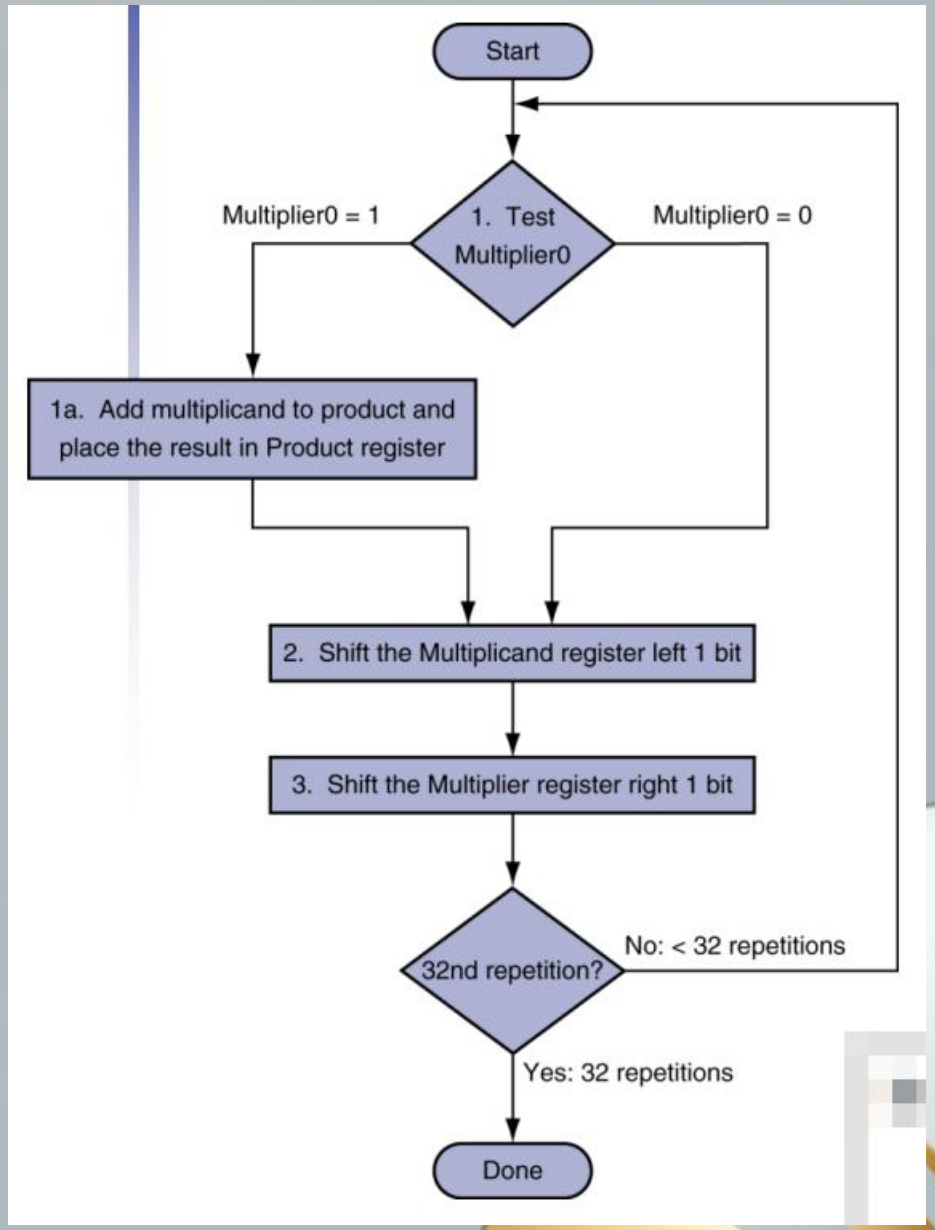
Multiplication(2)

```
.data                #MIPS
m1:.byte 8           #multiplicand
m2:.byte 9           #multiplier

.text
lb $t0,m1
lb $t1,m2
add $t2,$0,$0

loop:
li $s1,1
and $s2,$s1,$t1   #to determine the lowest bit of $s1
beq $s2, $0, jumpAdd
add $t2, $t0, $t2
jumpAdd:
sll $t0,$t0,1
srl $t1,$t1,1
addi $a0,$a0,1
addi $a1,$0,4          #4 is the length of 9 in binary
blt $a0,$a1,loop

add $a0,$0,$t2
li $v0,35
syscall
```

Can this assembly code get the correct product result?

If the muliplier is less than 4, could be the assembly code work more effectively ?





Multiplication(practice2-1)

The assembly code on the right hand is just for the multiplier whose bitwith is not larger than 4, and only for the unsigned multiplication, modify the code to achive the following function:

1) The bitwidth of multiplicand and multiplier is 16
2) The higest bit is take as the sign bit, to implement the signed multiplication.

Note: Don't using the mul instruction.

```
.data                #MIPS
m1:.byte 8           #multiplicand
m2:.byte 9           #multiplier

.text
lb $t0,m1
lb $t1,m2
add $t2,$0,$0

loop:
li $s1,1
and $s2,$s1,$t1   #to determine the lowest bit of $s1
beq $s2, $0, jumpAdd
add $t2, $t0, $t2
jumpAdd:
sll $t0,$t0,1
srl $t1,$t1,1
addi $a0,$a0,1
addi $a1,$0,4          #4 is the length of 9 in binary
blt $a0,$a1,loop

add $a0,$0,$t2
li $v0,35
syscall
```

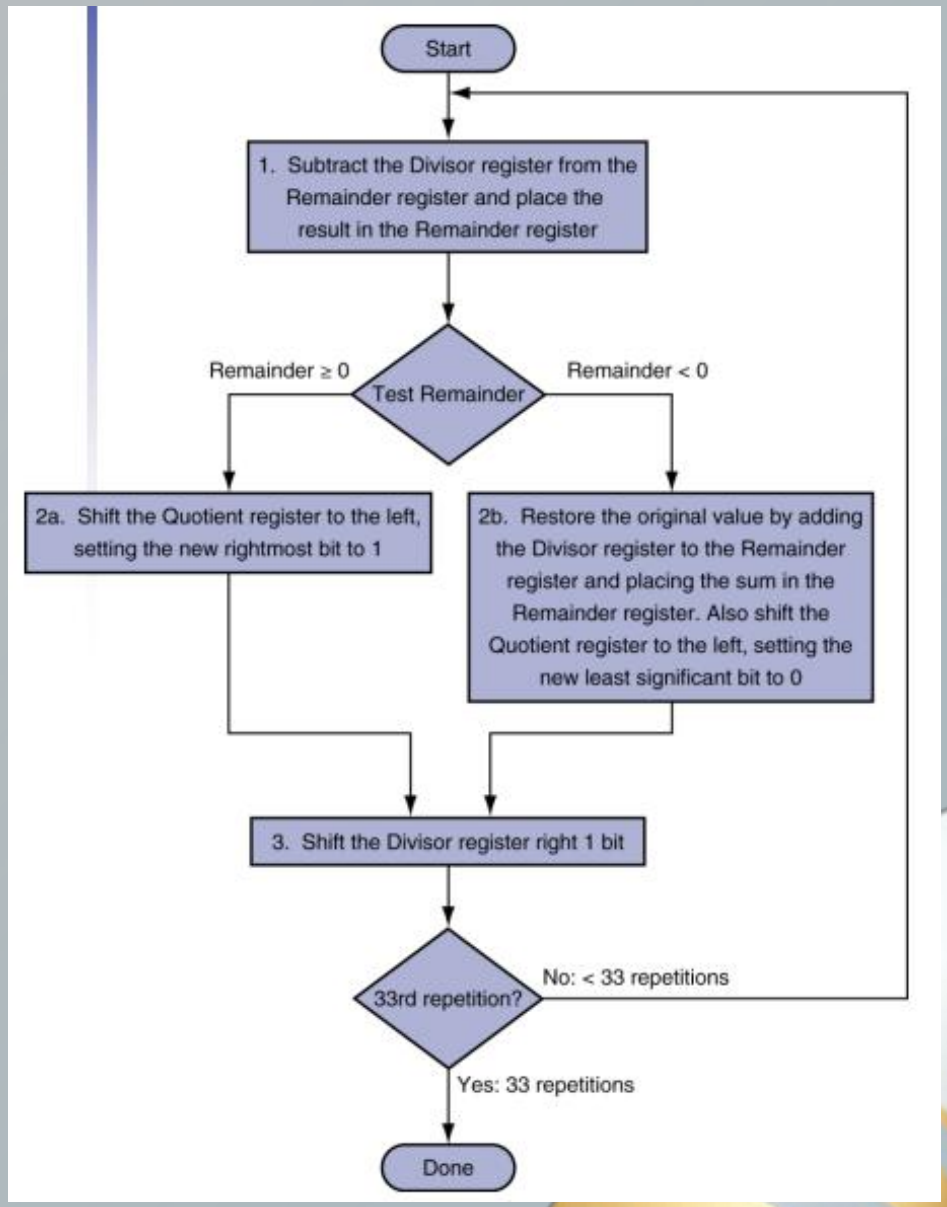
Division (1)

- Check for 0 divisor

- Long division approach
  - If divisor ≤ dividend bits: 1 bit in quotient, subtract
  - Otherwise: 0 bit in quotient, bring down next dividend bit

- Restoring division
  - Do the subtract, and if remainder goes < 0, add divisor back

- Signed division
  - Divide using absolute values
  - Adjust sign of quotient and remainder as required



Division （2） long division approach

Step0: prepare for the long division approach

```
.data                                   #MIPS piece1/3
dividend: .word 7
divisor:   .word 2
q: .word 0
remainder: .word 0
x: .word 0x8000
looptimes: .byte 5

.text
lw $t1,dividend  # t1 : diviend
lw $t2,divisor
sll $t2,$t2,4       # t2 : dividor
lw $t3,dividend  # t3 store the remainder
add $t4,$0,$0     # t4 Quot

lw $a0,x          #a0 used to get the higest bit of rem
add $t0,$0,$0   # t0: loop cnt
lb $v0,looptimes    #v0: looptimes
```

■ Divide 7_{dec} ($0000\ 0111_{bin}$) by 2_{dec} (0010_{bin})

Iter	Step	Quot	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	Rem = Rem – Div	0000	0010 0000	1110 0111
	Rem < 0 ➔ +Div, shift 0 into Q	0000	0010 0000	0000 0111
	Shift Div right	0000	0001 0000	0000 0111
2	Same steps as 1	0000	0001 0000	1111 0111
		0000	0001 0000	0000 0111
		0000	0000 1000	0000 0111
3	Same steps as 1	0000	0000 0100	0000 0111
4	Rem = Rem – Div	0000	0000 0100	0000 0011
	Rem >= 0 ➔ shift 1 into Q	0001	0000 0100	0000 0011
	Shift Div right	0001	0000 0010	0000 0011
5	Same steps as 4	0011	0000 0001	0000 0001

Division (3) long division approach

Step1-5: Do the long division approach

```
loopb:                                  #MIPS piece2/3
# $t1: dividend,   $t2: divisor,    $t3: remainder,     $t4: quot
# $a0: 0x8000,  $v0: 5

sub $t3,$t3,$t2          #dividend  - dividor
and $s0,$t3,$a0          # get the higest bit of rem to check if rem<0
sll $t4,$t4,1            # shift left quot with 1bit
beq $s0,$0, SdrUq        # if  rem>=0, shift Div right
add $t3,$t3,$t2          # if rem<0, rem=rem+div
srl $t2,$t2,1
addi $t4,$t4,0
j loope

SdrUq:
srl $t2,$t2,1
addi $t4,$t4,1

loope:
addi $t0,$t0,1

bne $t0,$v0,loopb
```

```
li $v0,1        #MIPS piece3/3

add $a0,$0,$t4  #print quot
syscall

add $a0,$0,$t3 #print remainder
syscall

li $v0,10
syscall
```

- Divide 7_{dec} ($0000\ 0111_{bin}$) by 2_{dec} (0010_{bin})

Iter	Step	Quot	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	Rem = Rem – Div	0000	0010 0000	1110 0111
	Rem < 0 → +Div, shift 0 into Q	0000	0010 0000	0000 0111
	Shift Div right	0000	0001 0000	0000 0111
2	Same steps as 1	0000	0001 0000	1111 0111
		0000	0001 0000	0000 0111
		0000	0000 1000	0000 0111
3	Same steps as 1	0000	0000 0100	0000 0111
4	Rem = Rem – Div	0000	0000 0100	0000 0011
	Rem >= 0 → shift 1 into Q	0001	0000 0100	0000 0011
	Shift Div right	0001	0000 0010	0000 0011
5	Same steps as 4	0011	0000 0001	0000 0001

Division （practice2-2）

The assembly code on the last two pages is just for the 8 bit unsigned division, do the following task:

1) To implement a 32 bit division with detect exception while the divisor is 0

2) The higest bit is take as the sign bit, to implement the signed division.

For signed division:

Step1: Divide using absolute values

Step2: Adjust sign of quotient and remainder as required

- **The quotient is "+", if the signs of divisor and dividend agrees, otherwise, quotient is "–".**
- **The sign of the remainder matches that of the dividend**

e.g.

(+7) ÷ (-2) = (-3) … (+1)

(-7) ÷ (-2) = (+3) … (-1)

Note: Don't using the div instruction.