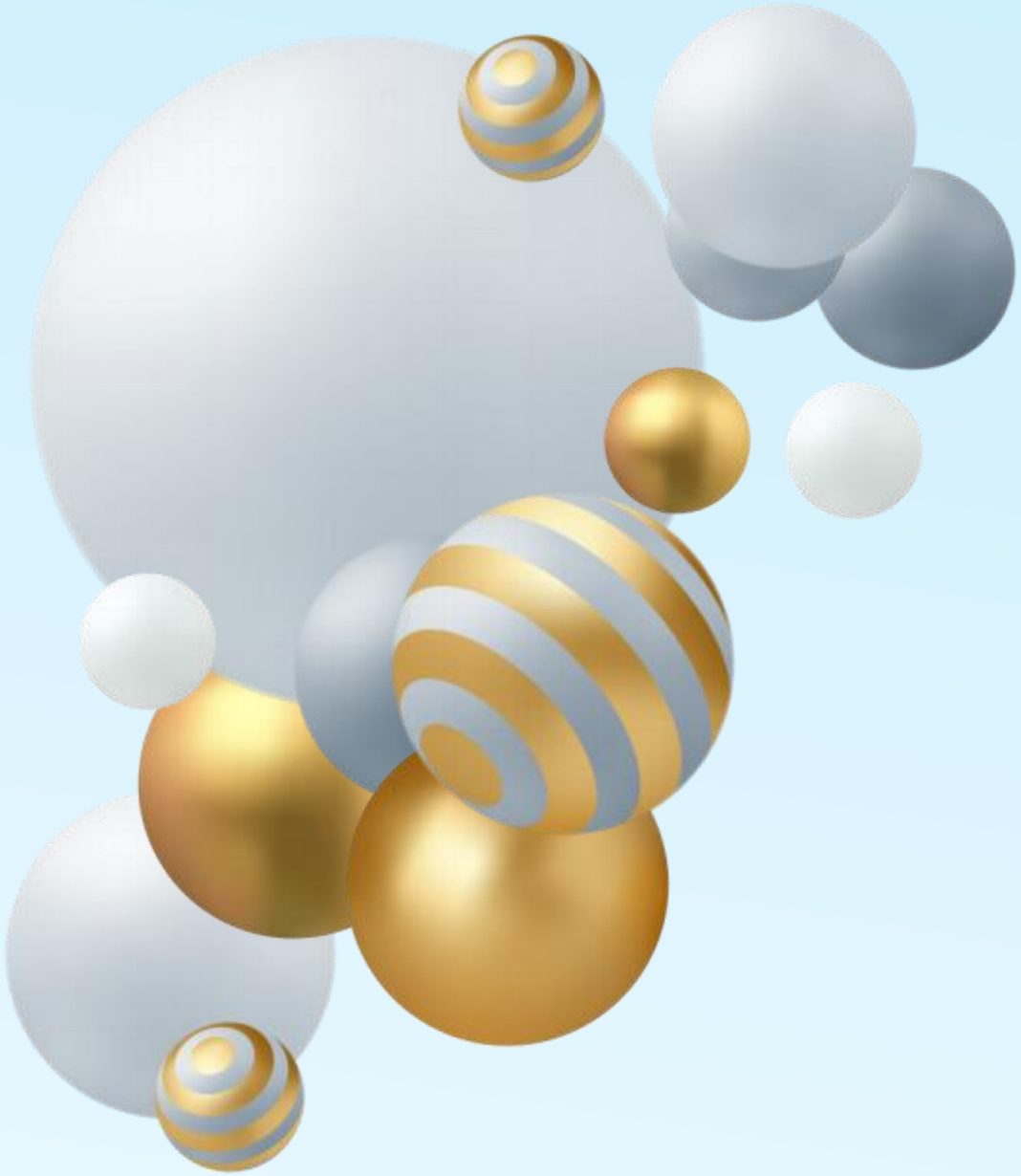
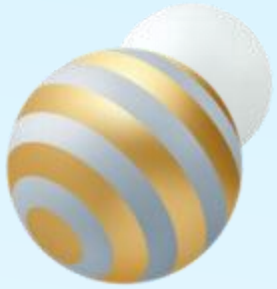


Computer Organization

Lab6 Integer Arithmetic

Integer Arithmetic; Exception



Topic

- Arithmetic
  - Adder
  - Subtraction (practice 1-1,1-2)
  - Multiplication (practice 2-1)
  - Division (practice 2-2)
- Exception (optional) (practice 3)

Tools: Vivado, Mars

Adder with overflow detector(1)

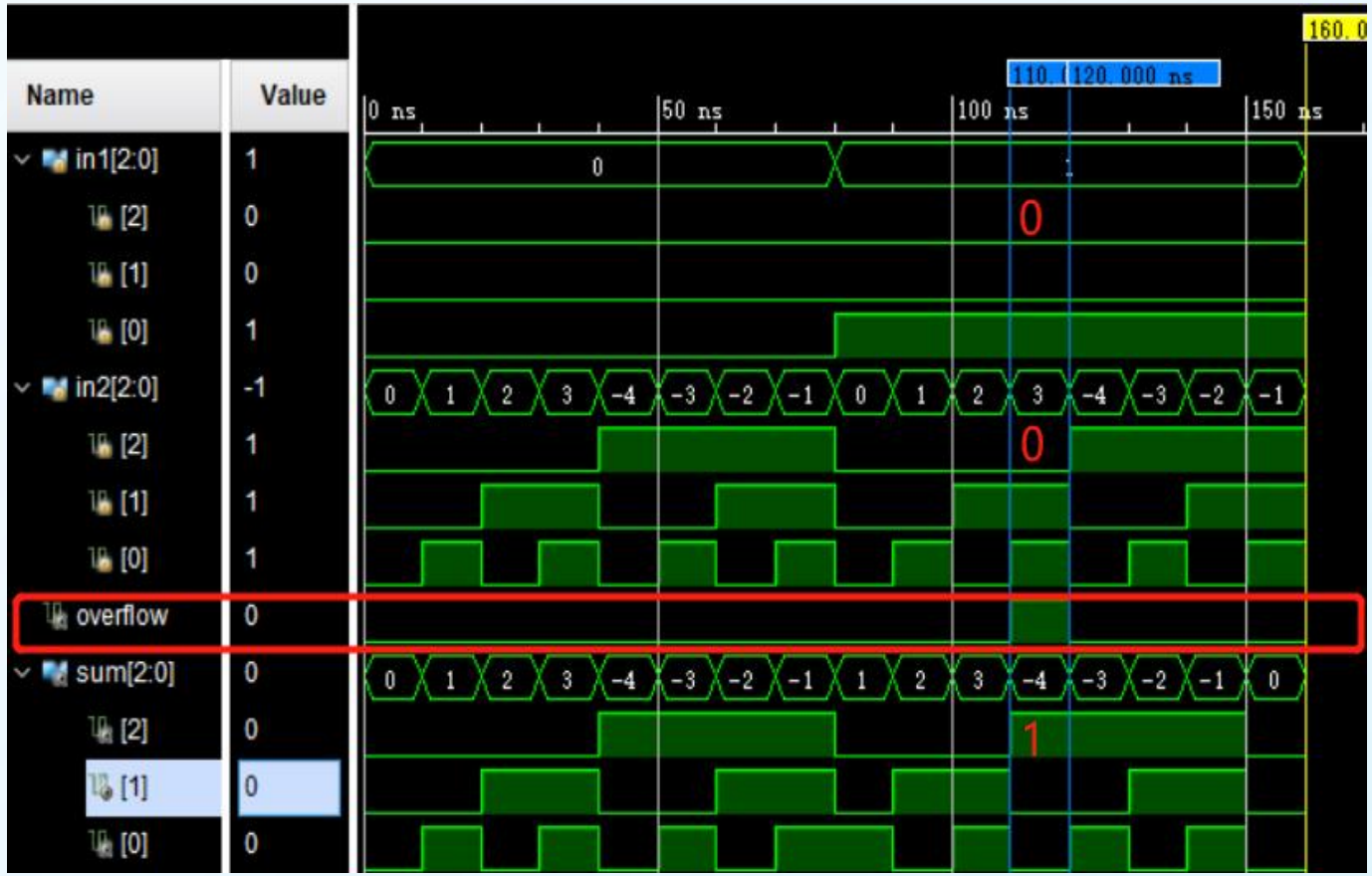
The ruler about overflow if result out of range on adder

- 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

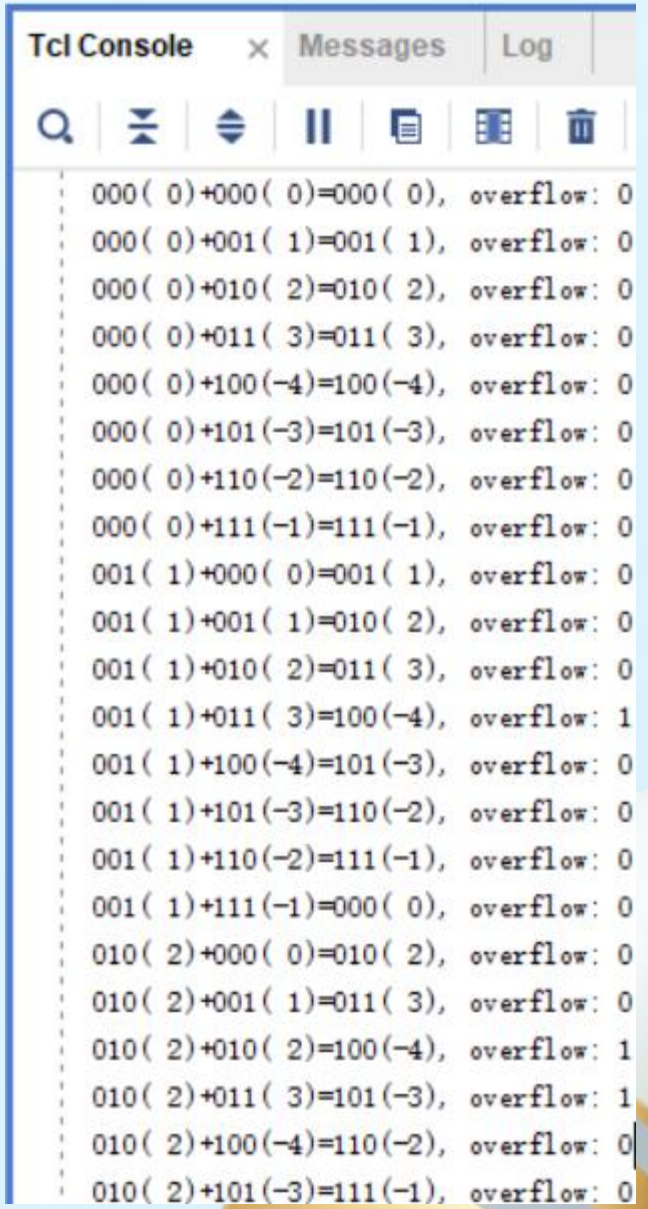
Please complete the testbench to finding all of the legal combinations of the two inputs will cause overflow.
A testbench on the bottom is for the reference.

```
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;
  $monitor( "%3b(%d)+%3b(%d)=%3b(%d), overflow: %b",
in1,$signed(in1),in2,$signed(in2),
sum,$signed(sum),overflow);

   repeat(15) #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.
In **Vivado**, the print information is showed in **Tcl Console**.

%**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(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.
In Vivado, the print information is showed in **Tcl Console**.

%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(practice1-1)

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
```

Signed calculation with overflow detection (practice1-2)

Design a circuit with 3 inputs(in1,in2,ctrl) and two outputs(result, overflow), the circuit do the signed addition or subtraction on ‘in1’ and ‘in2’, the caculation type is determined by the input ‘ctrl’, the calculation result is show on the output of ‘result’, if there is arithmetic overflow, the output ‘overflow’ is 1’b1, otherwise it is 1’b0.

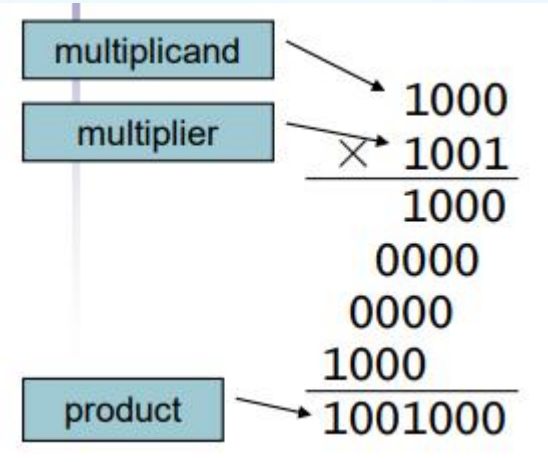
The bitwidth of in1, in2 and result is 3, the bitwidth of ctrl and overflow is 1.
if ctrl is 1’b1: result = in1 + in2
if ctrl is 1’b0: result = in1 - in2

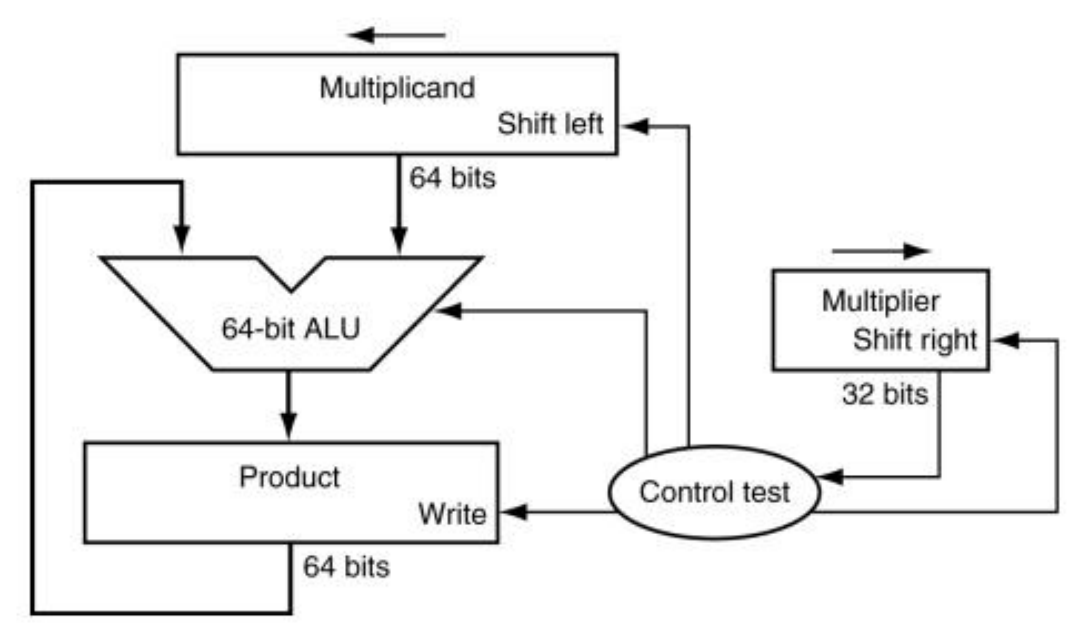
Tips: the demo of this lab on adder and subtraction could be used as sub-module, using structrue design would helps you to finish the design more effieciently.

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?



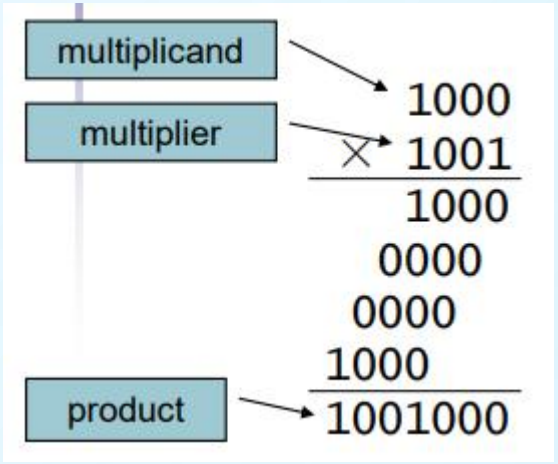


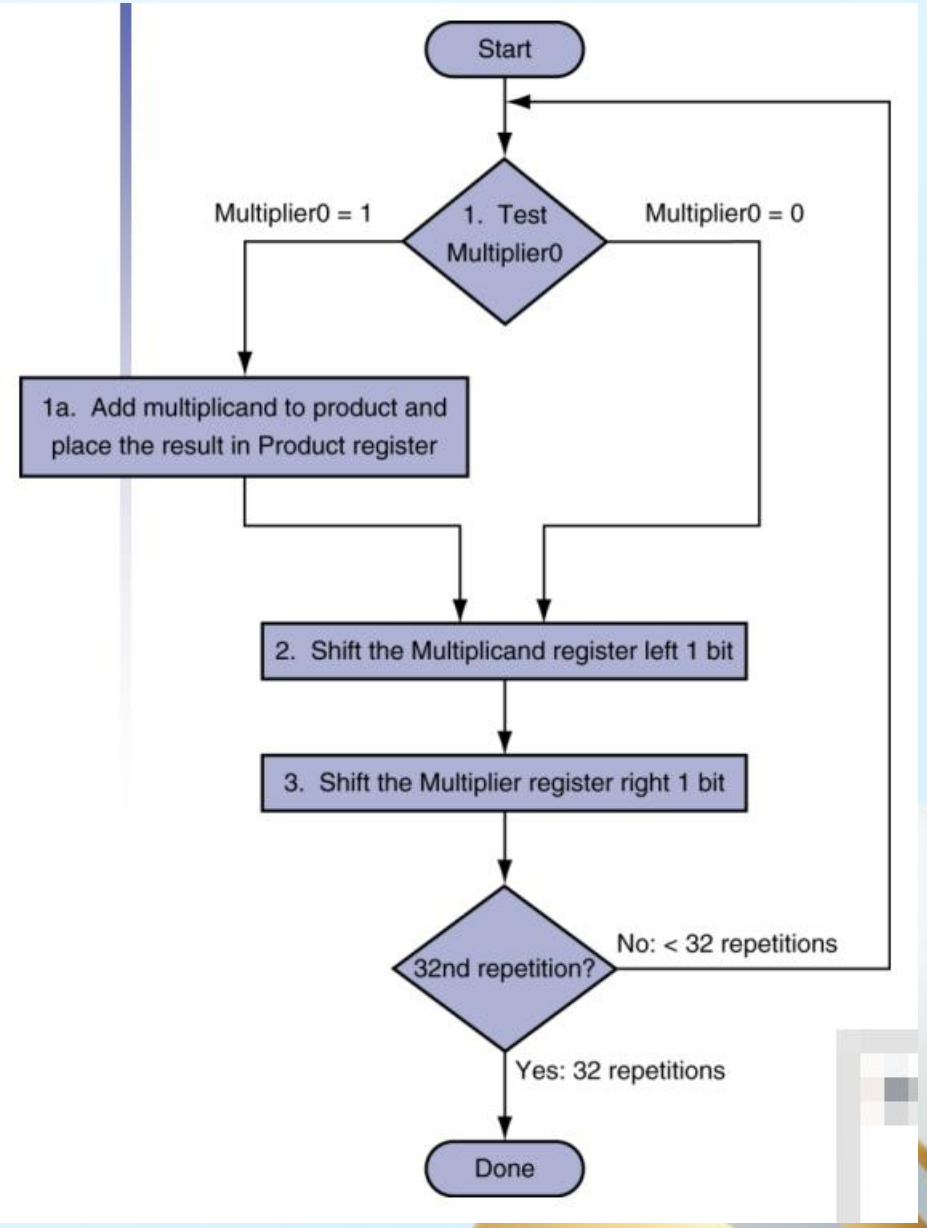
Multiplication(2)

```
.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
```

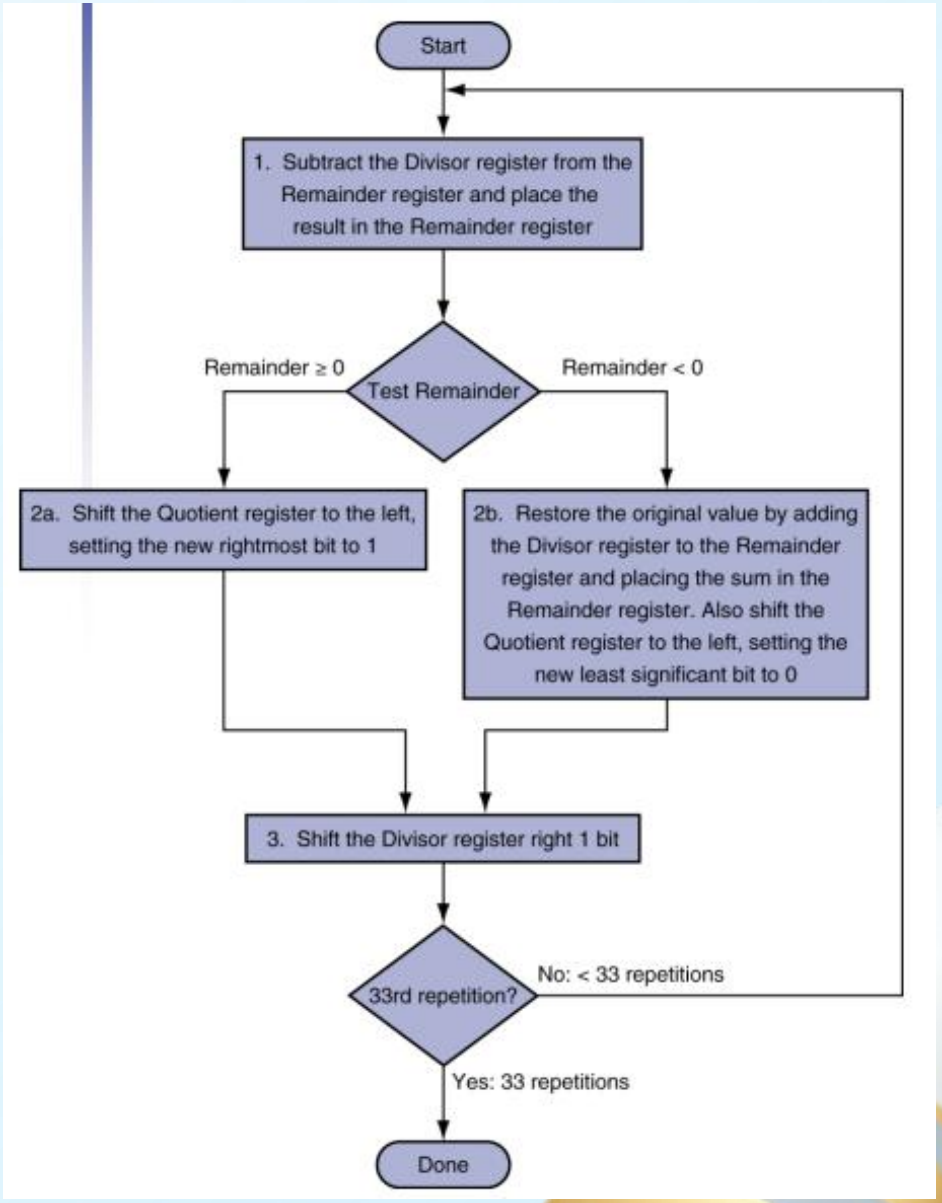
Division (1)

- 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.

Exception (1) defination

An **exception** is an event that disrupts the normal flow of the execution of your code.

- When an exception occurs, the CPU will figure out what is wrong by checking its status, see if it can be corrected and then continue the execution of the normal code like nothing happened.
- E.g. Accessing to the 0x0 address in user mode will trigger an exception.

The following exceptions are the most common in the main processor:

- Address error exceptions
  - Which occur when the machine references a data item that is NOT on its proper memory alignment or when an address is invalid for the executing process.
- Overflow exceptions
  - Which occur when arithmetic operations compute signed values and the destination lacks the precision to store the result.
- Bus exceptions
  - Which occur when an address is invalid for the executing process.
- Divide-by-zero exceptions
  - Which occur when a divisor is zero.

Exception (2) demo1

```
.text
print_string:
        addi $sp,$sp,-4
        sw $v0,($sp)

        li $v0,4
        syscall

        lw $v0,($sp)
        addi $sp,$sp,4

        jr $ra
```

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$8 (vaddr)	8	0x00000000
$12 (status)	12	0x0000ff13
$13 (cause)	13	0x00000010
$14 (epc)	14	0x0040000c

Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x23bdfffc	addi $29, $29, 0xfffffffc	5: addi $sp, $sp, -4
	0x00400004	0xafa20000	sw $2, 0x00000000($29)	6: sw $v0, ($sp)
	0x00400008	0x24020004	addiu $2, $0, 0x00000004	9: li $v0, 4
	0x0040000c	0x0000000c	syscall	10: syscall
	0x00400010	0x8fa20000	lw $2, 0x00000000($29)	12: lw $v0, ($sp)
	0x00400014	0x23bd0004	addi $29, $29, 0x00000004	13: addi $sp, $sp, 4
	0x00400018	0x201fffff	addi $31, $0, 0xffffffff	15: addi $ra, $zero, 0xffffffff
	0x0040001c	0x03e00008	jr $31	16: jr $ra

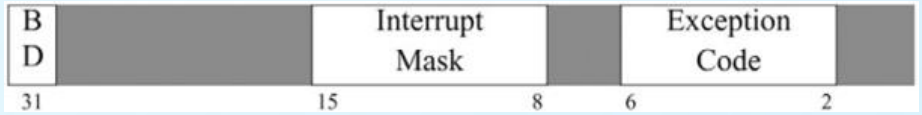
```
Runtime exception at 0x0040000c: address out of range 0x00000000
```

The unit of memory which is addressed by 0x00000000 is not allowed to access here.

Exception (3) - Coprocessor0

The Register in Coprocessor 0

Register name	Register number	Usage
VAddr	8	memory address at which an offending memory reference occurred
Status	12	interrupt mask and enable bits
Cause	13	exception type and pending interrupt bits
EPC	14	address of instruction that caused exception



Registers | Coproc 1 | Coproc 0

Name	Number	Value
$8 (vaddr)	8	0x00000000
$12 (status)	12	0x0000ff13
$13 (cause)	13	0x00000030
$14 (epc)	14	0x00400010

EXCEPTION CODES

Number	Name	Cause of Exception	Number	Name	Cause of Exception
0	Int	Interrupt (hardware)	9	Bp	Breakpoint Exception
4	AdEL	Address Error Exception (load or instruction fetch)	10	RI	Reserved Instruction Exception
5	AdES	Address Error Exception (store)	11	CpU	Coprocessor Unimplemented
6	IBE	Bus Error on Instruction Fetch	12	Ov	Arithmetic Overflow Exception
7	DBE	Bus Error on Load or Store	13	Tr	Trap
8	Sys	Syscall Exception	15	FPE	Floating Point Exception

Exception (3) arithmetic overflow exception

Will the 'sub', 'mul', 'div' trigger an exception in MIPS? If yes, what's(are) the type(s) of the exception(s)

```
.data
        addend1: .word 0x7fffffff
        addend2: .word 0x7fffffff
.text
print_string:
        lw $t0,addend1
        lw $t1,addend2
        add $a0,$t0,$t1

        li $v0,1
        syscall

        li $v0,10
        syscall
```

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$8 (vaddr)	8	0x00000000
$12 (status)	12	0x0000ff13
$13 (cause)	13	0x00000030
$14 (epc)	14	0x00400010

```
Runtime exception at 0x00400010: arithmetic overflow
```

Text Segment

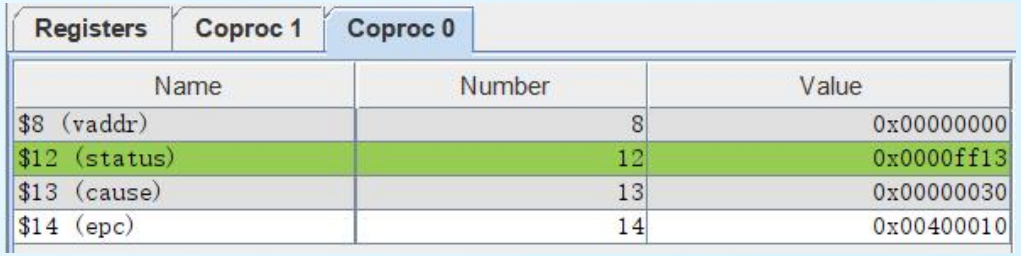
Bkpt	Address	Code	Basic		Source
	0x00400000	0x3c011001	lui $1, 0x00001001	6:	lw $t0, addend1
	0x00400004	0x8c280000	lw $8, 0x00000000($1)		
	0x00400008	0x3c011001	lui $1, 0x00001001	7:	lw $t1, addend2
	0x0040000c	0x8c290004	lw $9, 0x00000004($1)		
	0x00400010	0x01092020	add $4, $8, $9	8:	add $a0, $t0, $t1
	0x00400014	0x24020001	addiu $2, $0, 0x00000001	10:	li $v0, 1
	0x00400018	0x0000000c	syscall	11:	syscall
	0x0040001c	0x2402000a	addiu $2, $0, 0x0000000a	13:	li $v0, 10
	0x00400020	0x0000000c	syscall	14:	syscall

If the bitwidth of both addend1 and adden2 are **8** bit, the initial value of them are both **0x7f**, would a **arithmetic overflow exception** be invoked by using the instrucion add to do the addition on them in MIPS?

Exception (4) actions with registers

How MIPS Acts When Taking An Exception?

- **Step1.** It sets up the **EPC register** (**$14** of Coproc 0) to point to the restart location
- **Step2.** CPU changes into **kernel mode** and **disables the interrupts** (MIPS does this by setting EXL bit of **Status register** (**$12** of Coproc 0))
- **Step3.** Set up **the Cause register**(**$13 of Coproc 0)** to indicate which is wrong so that software can tell the reason for the exception. If it is for **address exception**, for example, TLB miss and so on, the **BadVaddr register**(**$8 of** Coproc 0) is set.
- **Step4.** CPU starts fetching instructions from the exception entry point and then goes to the **exception handler.**

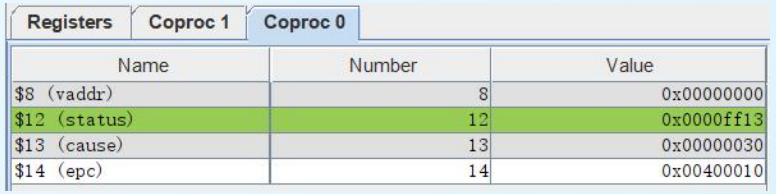


Name	Number	Value
$8 (vaddr)	8	0x00000000
$12 (status)	12	0x0000ff13
$13 (cause)	13	0x00000030
$14 (epc)	14	0x00400010

Exception (5) exception kernel mode

```
.data
        dmsg:    .asciiz "\ndata over"
.text
main:   li $v0,5
        syscall
        teqi $v0,0 #自陷
        la $a0,dmsg
        li $v0,4
        syscall
        li $v0,10
        syscall
```

```
.ktext      0x80000180
        move $k0,$v0
        move $k1,$a0
        la $a0,msg
        li $v0,4
        syscall

        move $v0,$k0
        move $a0,$k1

        mfc0 $k0,$14
        addi $k0,$k0,4
        mtc0 $k0,$14

        eret
.kdata
msg: .asciiz "\nTrap generated"
```



Tips

.kdata subsequent items are stored in the **kernel data segment.** *If the optional argument addr is present, subsequent items are stored starting at address addr.*

.ktext subsequent items are stored in the **kernel text segment**, *In SPIM, these items may only be instructions or words . If the optional argument addr is present, subsequent items are stored starting at address addr.*

eret instruciton is used to return to the original location before falling into the exception

practice3

1) Implementing a procedure(named addB) to do the additon on two operands(whose bitwidth is8bit) with arithmetic overflow detection:

1. There are two parameters for the procedure as two oprands
2. The bitwidth of the two operands is 8
3. The two operands are taken as signed number, if there is overflow on the addtion, invoke an exception. (tips: trap instruction on MIPS)
4. if there is no exception, store the sum of addition to $v0 register as return value.

2) Implementing the exception handler with following function:

1. Stop the program running
2. Output prompt information, including “runtime exception at 0x_***“(the address of the instruction which triggered the exception), the cause of the exception (“arithmetic overflow on 8bit signed addtion”).
3. Exit the program.