Topics

- ➢ Arithmetic

  - ✓ Adder   (practice 1-1)

  - ✓ Subtraction   (practice 1-2)

  - ✓ Multiplication   (practice 2-1)

  - ✓ Division

- ➢ Practice

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   001 + 011 = 110

     • adding two -ve operands, get +ve operand   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)

➢ The figure is the waveform of the circuit "adder"

  ✓ 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.

✓ Observe the waveform from 110ns to 120ns

✓ 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

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

```verilog
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
```

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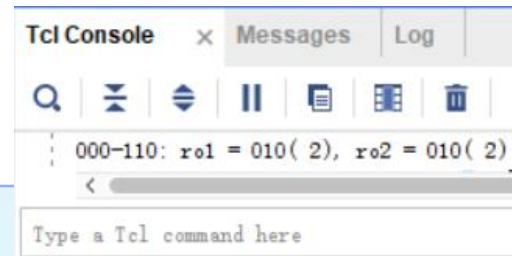
```verilog
module subO1(in1,in2,result);   //verilog
input [2:0] in1;
input [2:0] in2;
wire vin2p1;
wire [1:0] vin2p2;
output [2:0] result;
assign vin2p1 = ~in2[2];
assign vin2p2 = ~in2[1:0] + 1;
assign result = in1 + {vin2p1,vin2p2};
endmodule
```

```verilog
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
```

```verilog
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 window.

%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.

```
Tcl Console   ×   Messages   Log

Q  ⋝  ⬍  ‖  ▤  ▦  🗑

000-110: ro1 = 010( 2), ro2 = 010( 2)
<

Type a Tcl command here
```

6

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 ?
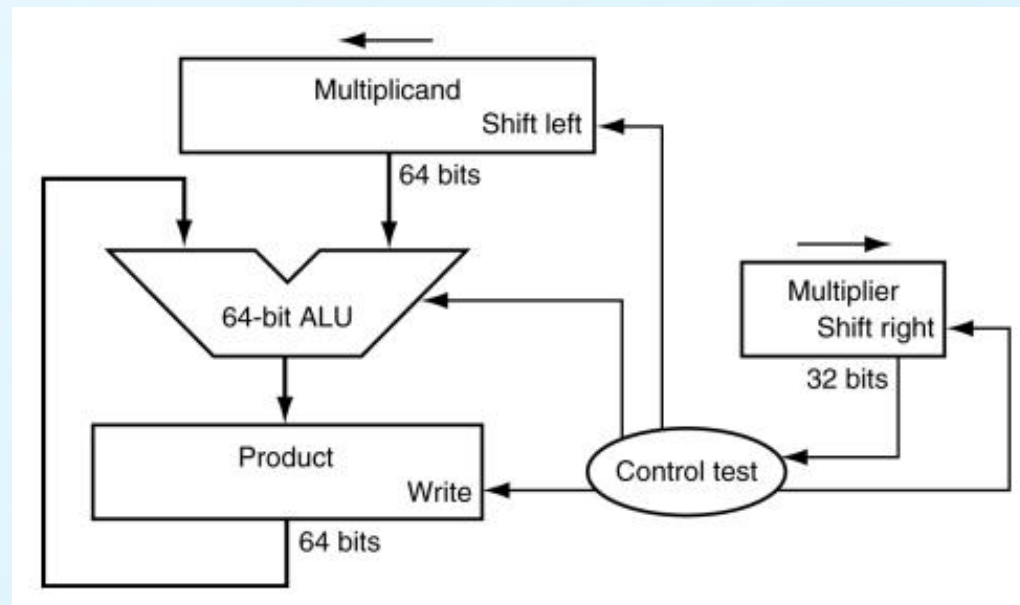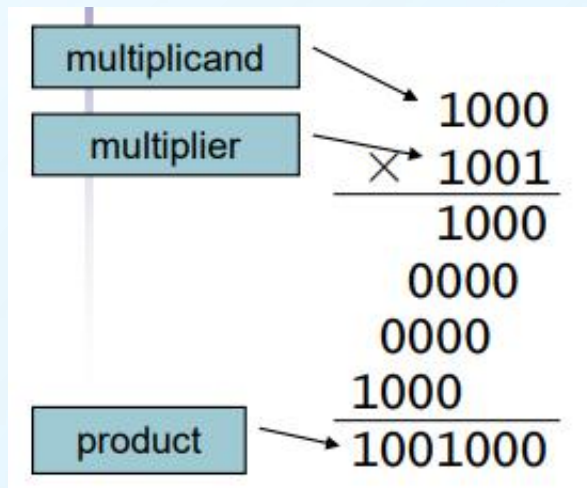
Multiplication (2)

➢ Can this piece of codes get the correct product result?

➢ If the length of multiplier is less than 4, could the assembly code work more effectively ?

```
# Piece 6-1
.data
      m1:.byte 8          # multiplicand
      m2:.byte 9          # multiplier
.text
      lb t0, m1
      lb t1, m2
      add t2, zero, zero
loop:
      li s1, 1
      and s2, s1, t1   #to determine the lowest bit of t1
      beq s2, zero, jumpAdd
      add t2, t0, t2
jumpAdd:
      slli t0, t0, 1
      srli t1, t1, 1
      addi a0, a0,1
      li a1, 4            # 4 is the length of 9 in binary
      blt a0, a1, loop

      mv a0, t2
      li a7, 1
      ecall

      li a7, 35
      ecall
```
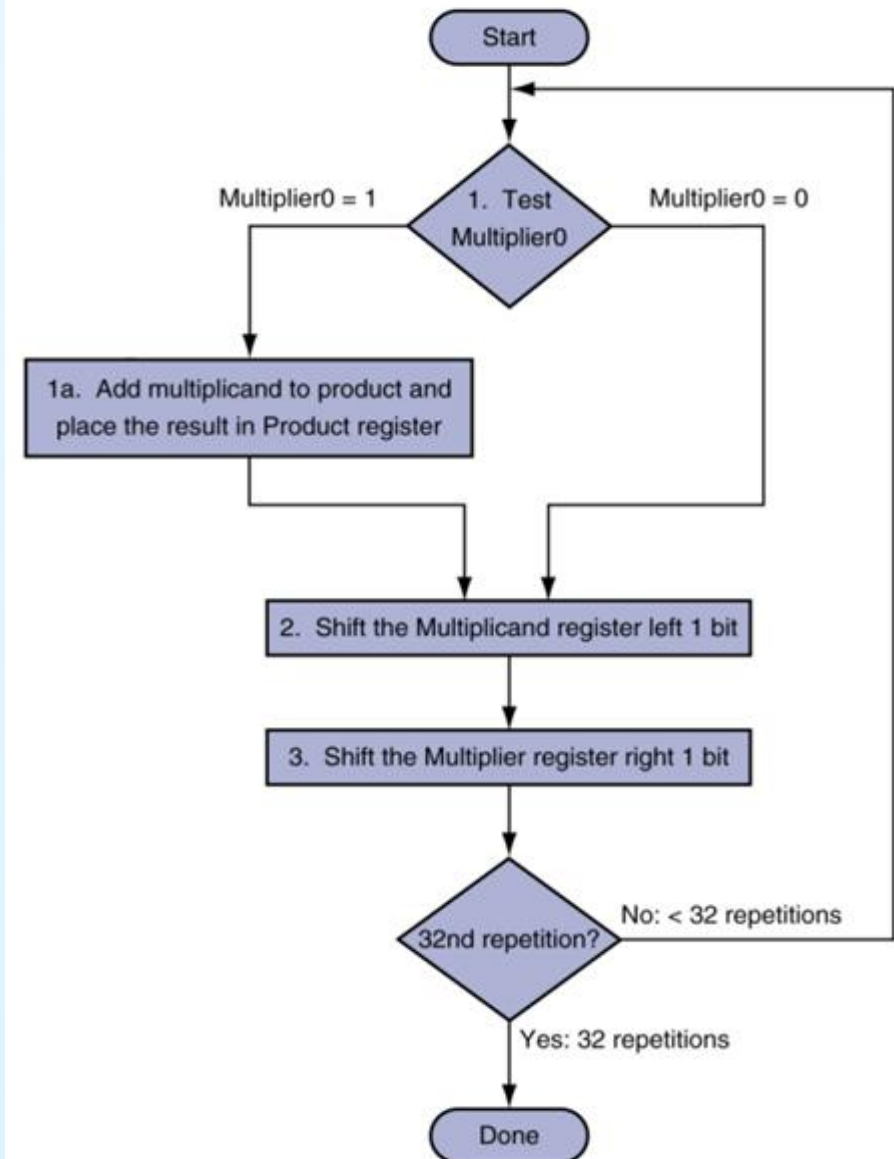
Multiplication instructions in RISC-V

- RV32M multiply extension

- mul, mulh, mulhsu, mulhu

| Inst	Name	FMT	Opcode	funct3	funct7	Description (C)
mul	MUL	R	0110011	0x0	0x01	rd = (rs1 * rs2)[31:0]
mulh	MUL High	R	0110011	0x1	0x01	rd = (rs1 * rs2)[63:32]
mulhsu	MUL High (S) (U)	R	0110011	0x2	0x01	rd = (rs1 * rs2)[63:32]
mulhu	MUL High (U)	R	0110011	0x3	0x01	rd = (rs1 * rs2)[63:32]

- If both high and low 32-bit multiplication product are required, the general code order is:

```
mulh rdh, rs1, rs2   #or mulhsu rdh, rs1, rs2  or mulhu rdh, rs1, rs2
mul rdl, rs1, rs2
```

Division (1)

➢ Check for 0 divisor by software.

➢ 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
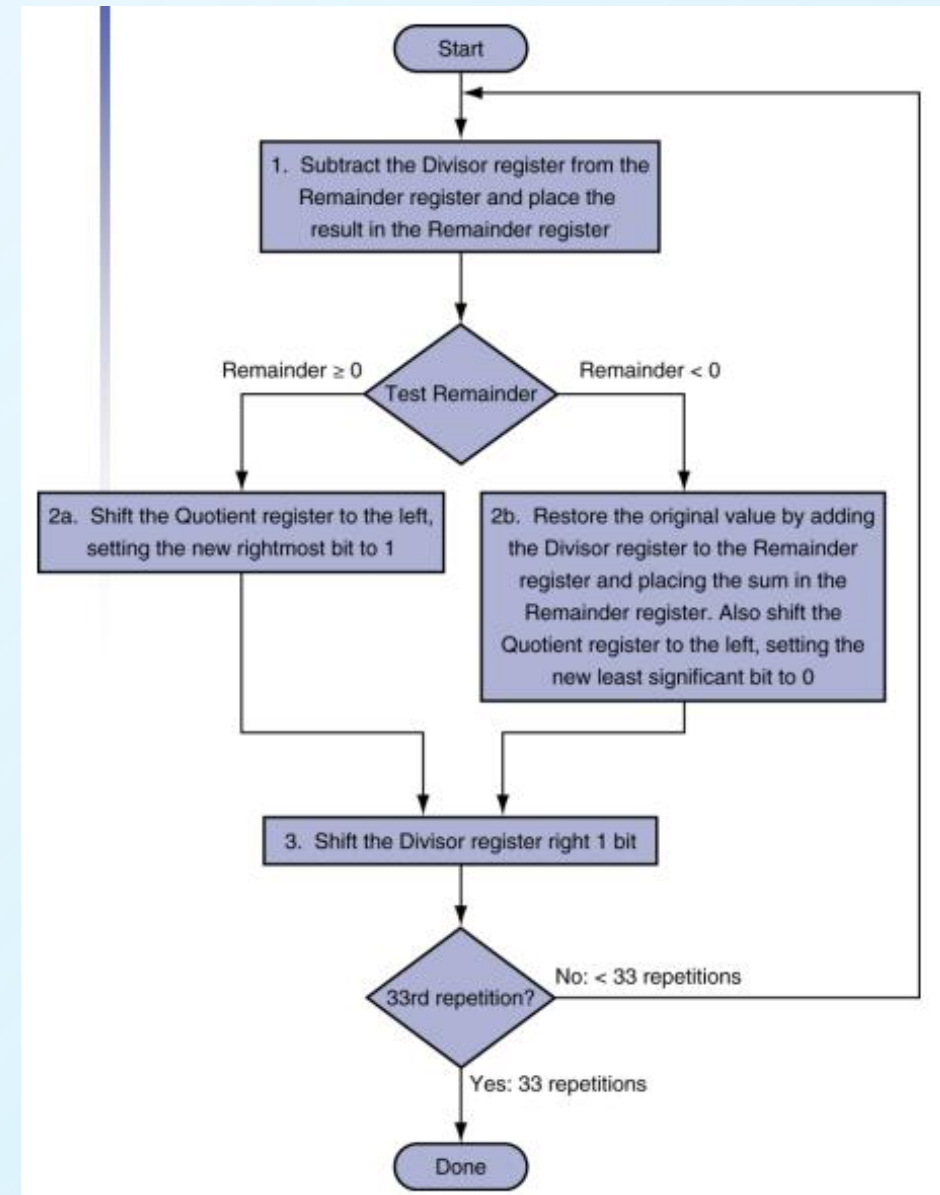
Division (2) long division approach

➢ Step0: prepare for the long division approach

```
# Piece 6-3
# Piece 6-3-1
.include "macro_print_str.asm"
.data
        dividend: .word 7
        divisor:  .word 2
        x: .word 0x8000
        looptimes: .byte 5
.text
        la t5, dividend
        lw t1, (t5)   # t1 : diviend
        la t5, divisor
        lw t2, (t5)   # t2 : divisor
        slli t2, t2, 4
        la t5, dividend
        lw t3, (t5)  # t3 store the remainder
        add t4, zero, zero     # t4 quotient

        la t5, x
        lw a1, (t5)          #a1 used to get the highest bit of remainder
        add t0, zero, zero   # t0: loop cnt
        la t5, looptimes
        lb s1, (t5)          #s1: 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

```
# Piece 6-3-2
loopb:
    # t1: dividend,   t2: divisor,    t3: remainder,    t4: quotient
    # a1: 0x8000,  s1: 5

    sub t3, t3, t2          #dividend  - divisor
    and s0, t3, a1          # get the highest bit of remainder to check if rem<0
    slli t4, t4, 1          # shift left quot with 1bit
    beq s0, zero, SdrUq     # if  rem>=0, shift Div right
    add t3, t3, t2          # if rem<0, rem=rem+div
    srli t2, t2, 1
    addi t4, t4, 0
    j loope

SdrUq:
    srli t2, t2, 1
    addi t4, t4, 1

loope:
    addi t0, t0, 1
    bne t0, s1, loopb
```

```
# Piece 6-3-3
    li a7, 1
    mv a0, t4  #print quotient
    ecall
    print_string("\n")
    li a7, 1
    mv a0, t3 #print remainder
    ecall

    li a7, 10
    ecall
```

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 instructions in RISC-V

➤ mul, mulh, mulhsu, mulhu

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)
div	DIV	R	0110011	0x4	0x01	rd = rs1 / rs2
divu	DIV (U)	R	0110011	0x5	0x01	rd = rs1 / rs2
rem	Remainder	R	0110011	0x6	0x01	rd = rs1 % rs2
remu	Remainder (U)	R	0110011	0x7	0x01	rd = rs1 % rs2

➤ If both quotient and remainder are required, the general code order is:

```
div rdq, rs1, rs2   # or divu rdq, rs1, rs2
rem rdr, rs1, rs2   # or remu rdr, rs1, rs2
```

Practice 1-1: Adder with overflow detector

➢ Please complete the test bench to finding all of the legal combinations of the two inputs，verify the function of the circuit "adder".

➢ A test bench on the bottom is for the reference.

```verilog
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
```

Practice 1-2: Subtraction with overflow detector

➢ Please complete the circuit to detect the overflow of the subtraction.

➢ Build a test bench 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
//verilog
module subtraction(in1,in2,result,overflow);

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


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

endmodule
```

Practice 2-1: Multiplication

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

✓ 1) The width of multiplicand and multiplier is 16.

✓ 2) The highest bit is take as the sign bit, to implement the signed multiplication.

✓ Note: Don't use the mul instruction.

```
# Piece 6-1
.data
    m1:.byte 8        # multiplicand
    m2:.byte 9        # multiplier
.text
    lb t0, m1
    lb t1, m2
    add t2, zero, zero
loop:
    li s1, 1
    and s2, s1, t1    #to determine the lowest bit of t1
    beq s2, zero, jumpAdd
    add t2, t0, t2
jumpAdd:
    slli t0, t0, 1
    srli t1, t1, 1
    addi a0, a0,1
    li a1, 4          # 4 is the length of 9 in binary
    blt a0, a1, loop

    mv a0, t2
    li a7, 1
    ecall

    li a7, 35
    ecall
```

Practice 2-2: Division

➢ The piece of assembly code of piece 6-3 on Page 13 and 14 is just for the **8 bit unsigned division**. Take piece 6-3 as reference, do the following tasks:

➢ 1) To implement a 32 bit division with detecting exception while the divisor is 0.

➢ 2) The highest bit is taken 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.

    ✓ (+7) ÷ (-2) = (-3) ⋯ (+1)   (-7) ÷ (-2) = (+3) ⋯ (-1)

➢ Note: Don't use the div instruction.