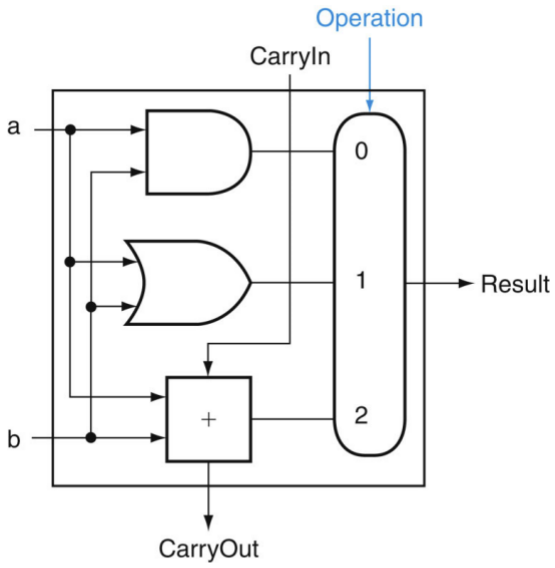
Lecture 6 and Lab 6 notes

ALU and basic operations

1. one-bit ALU perform **AND, OR, addition**



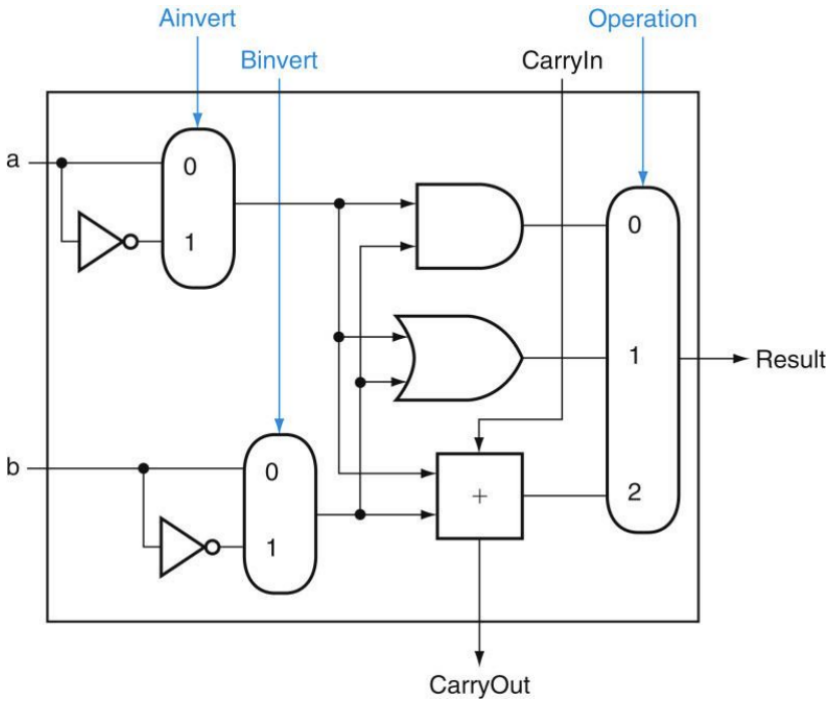
op = 0, o = a & b (AND)
op = 1, o = a | b (OR)
op = 2, o = a + b (ADD)

4

Operation is a Multiplexer with 2 bits selection

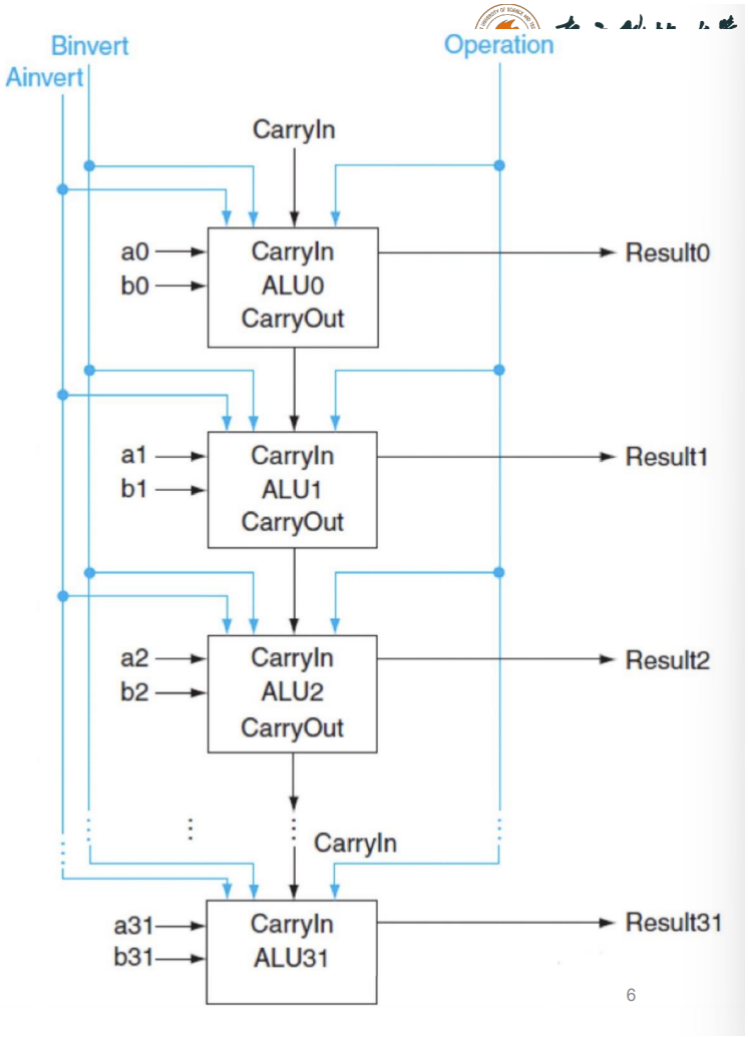
2. Additionally, ALU can also perform **NAMD, NOR, subtraction**

	formula	**{Ainvert, Binvert, Operation[1:0]}**
NAND	$(ab)' = a' + b'$	1101
NOR	$(a + b)' = a'b'$	1100
subtraction	$a - b = a + b' + (1^{st}\ CarryIn = 1)$	0110



3. 32-bit ALU

Notes that for the same task, **{Ainvert, Binvert, Operation[1:0]}** is always the same.



4. Implementation of `slt`

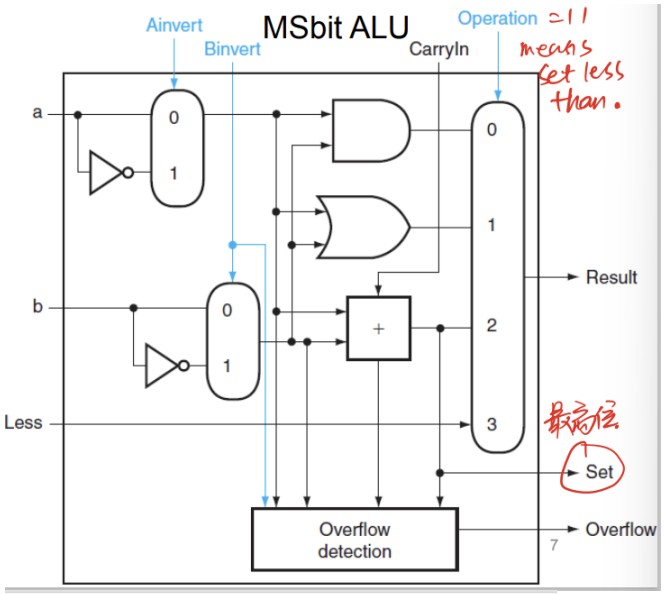
Recap:

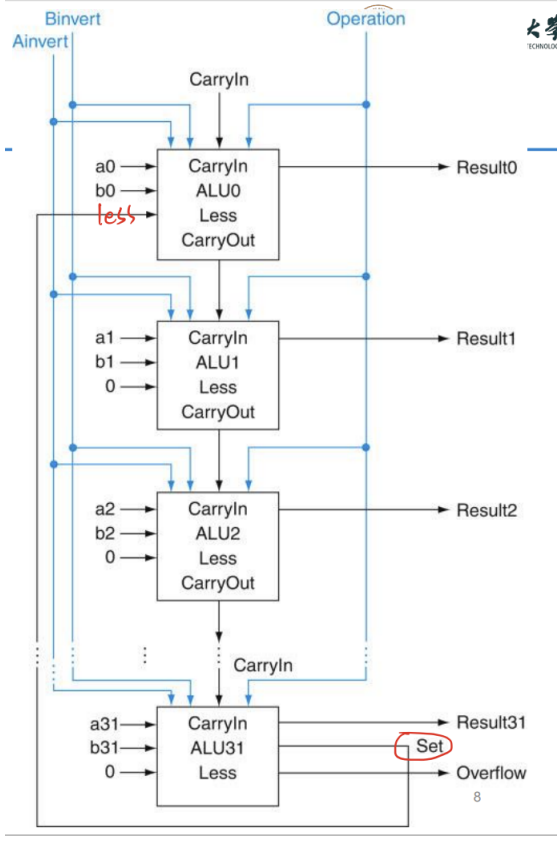
```
slt $t0, $t1, $t2
# equivalent form
# => t0 = t1 - t2 < 0
# => t0 = MSbit of (t1 - t2) == 1
# => t0 = MSbit of (t1 - t2)
```

Solution:

calculate `t1 - t2`, and set `t0` as `MSbit of (t1 - t2)`

add a operation: `11` means `slt`

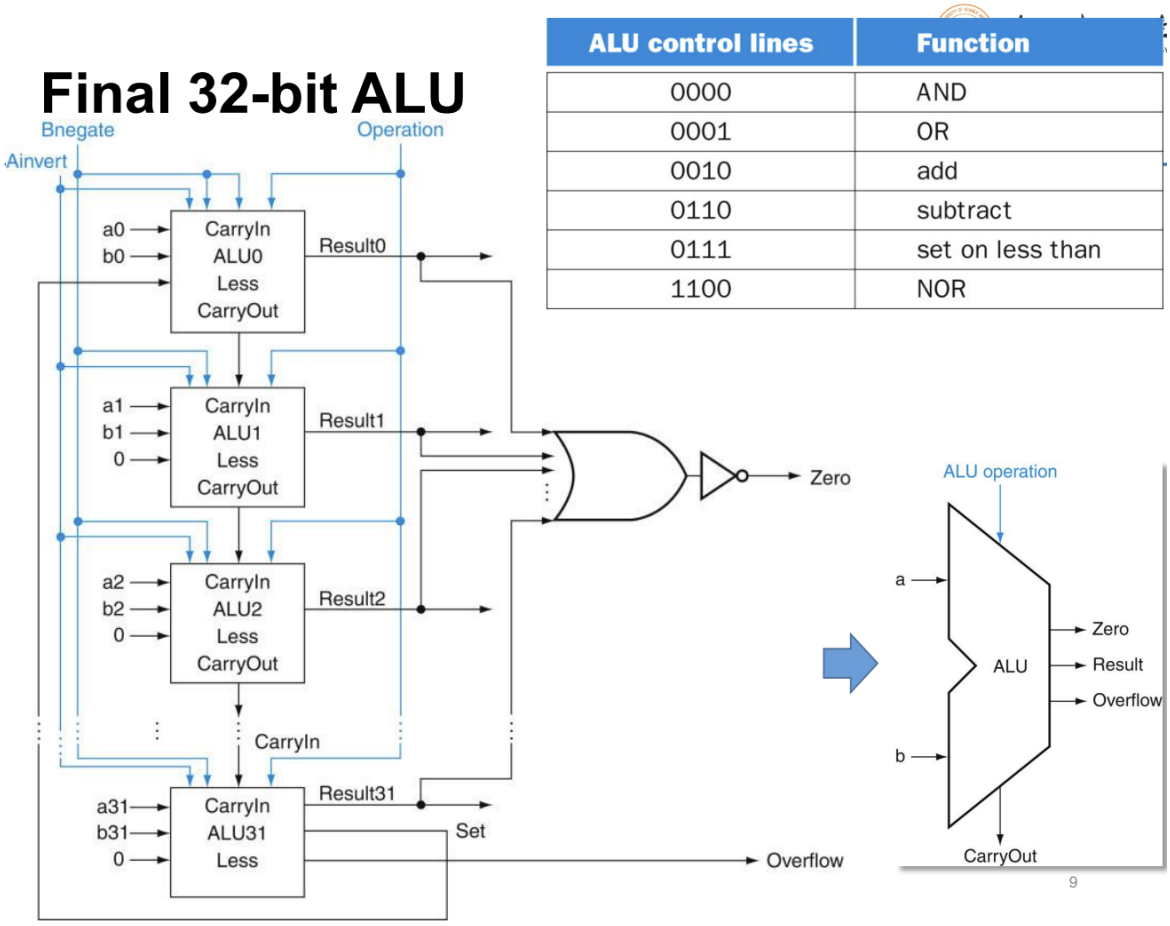




5. Final 32-bit ALU

   Notes that:

   - The `CarryIn == 1` if Fucntion is subtraction or slt, and `CarryIn == 0` if Fucntion is addition. Otherwise `CarryIn` can be either `1 or 0`.
   - `Bnegate` can be a input as `CarryIn`



ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR

Dealing with Overflow

1. Definition:
   - Only happens in **subtraction** and **addition**
   - `(+v1) + (+v2) = (-v3)` or `(-v1) + (-v2) = (+v3)` in the **addition**.
   - `(+v1) - (-v2) = (-v3)` or `(-v1) - (v2) = (+v3)` in the **subtraction**.
2. handling the overflow
   - Ignore overflow

     Use MIPS `addu, addiu, subu` instructions
   - Raising an exception

     Use MIPS `add, addi, sub` instructions
3. Overflow detection machanism
   - signed addition

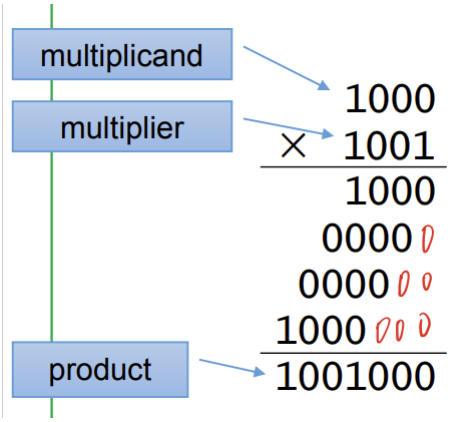
```
# to avoid exception, use addu
# remark that it will not exceed an unsigned number
addu $t0, $t1, $t2
# check whether they have different sign
xor $t3, $t1, $t2 # (different sign <=> MSbit of t3 is 1 <=>
no_overflow)
blt $t3, 0, no_overflow
# check whether the MSbit of $t3 is differs from $t1
xor $t3, $t3, $t1 # (different sign <=> MSbit of t3 is 1 <=> overflow)
bge $t3, 0, no_overflow
overflow:
#exception handling
no_overflow:
```

- unsigned addition

```
# compute 0xffff - $t1
nor $t0, $t1, 0
# 0xffff - $t1 >= t2 than no_overflow
bgt $t0, $t2, no_overflow
oveflow:
#exception handling
no_overflow:
```

Multiplication

1. Original thought:
   - from vertical multiplication

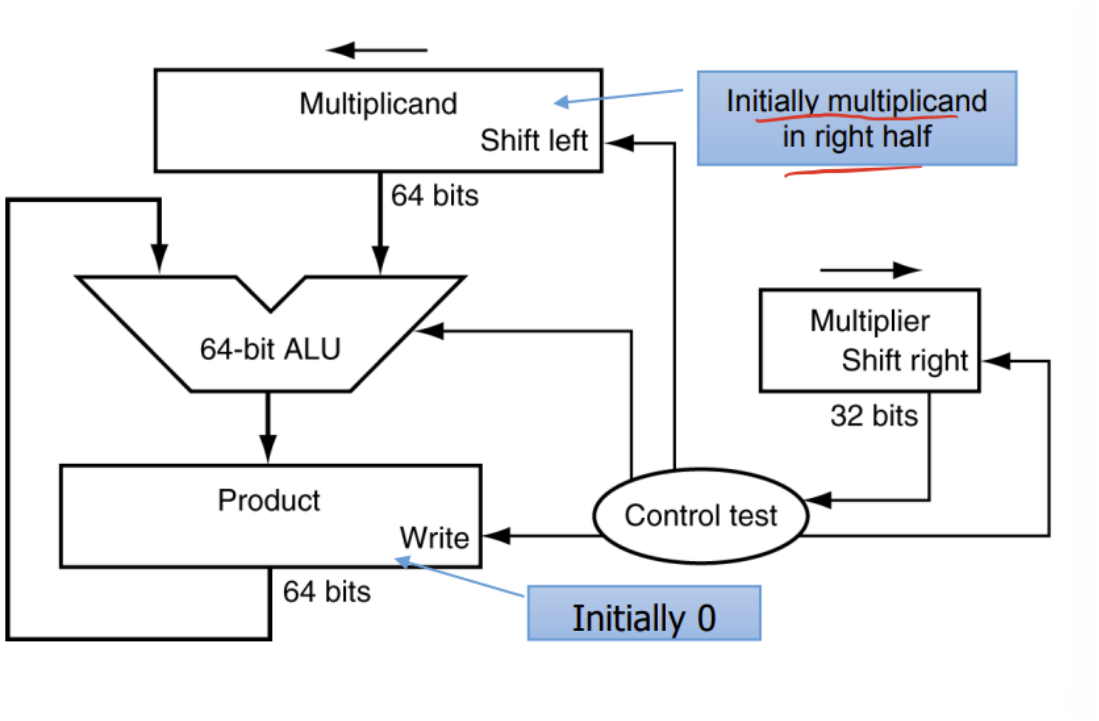


   - Algorithm

```
we have multiplicand[63:0] in the right half, multiplier[31:0],
product[63:0]

repeat(32):
    if multiplier[0] == 1:
        product += multiplicand
    multiplier >>= 1;
    multiplicand <<= 1;
```

- graph



- expense

  $5 * Register_{32bit} + 2 * ALU_{32bit}$

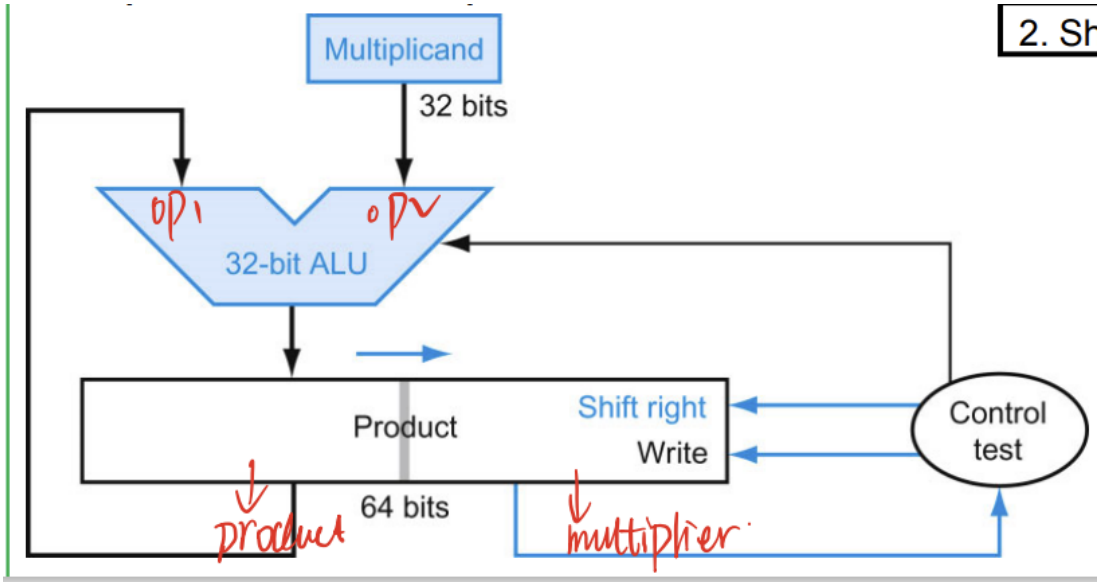
2. Optimized Multiplier Hardware

- Algorithm

```
Concatenate the right half of original product(32'b0) to
multiplier[31:0], say Product_Multiplier[63:0]
Store multiplicand in 32 bits, say multiplicand[31:0]
repeat(32):
 if(Product_Multiplier[0] == 1)
     Product_Multiplier[63:32] += multiplicand
 Product_Multiplier >> 1;

# remark that shifting is after the addition
```

- graph



- expense

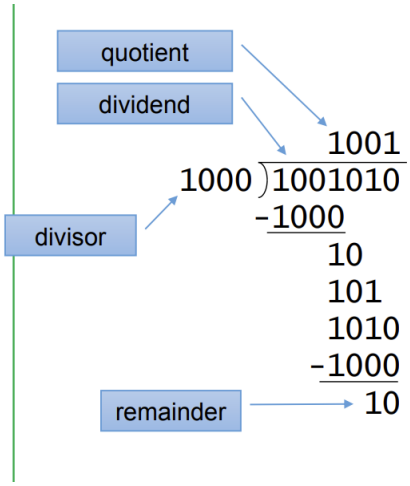
  $3 * Register_{32bit} + 1 * ALU_{32bit}$

3. Mips Multiplication

```
mult rs, rt
multu rs, rt
# 64-bit product in HI/LO
mfhi rd
mflo rd
# Move from HI/LO to rd
# Can test HI value to see if product overflows 32 bits
mul rd, rs, rt
# Least-significant 32 bits of product -> rd
```

Division

1. Orginal thought
   - division steps

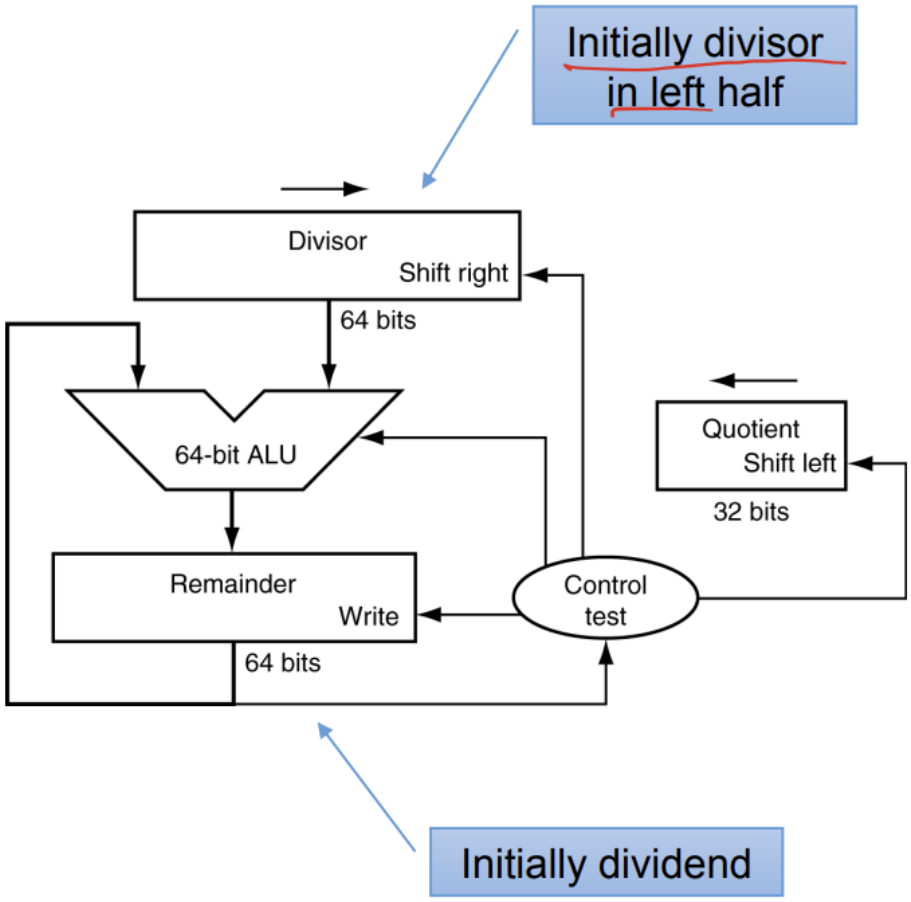


   - Algorithm

```
remainder[63:0] will store the remainder in the right half, initialized with the dividend
divisor in the left half of the divisor[63:0]
quotient in the quotient[31:0], initialized with 0

repeat(32):
    divisor >>= 1
    remainder -= divisor
    if(remainder[63] == 0)
        quotient <<= 1
        quotient[0] = 1
    else
        quotient <<= 1
        quotient[0] = 0
        remainder += divisor
```

- graph



- expense

  $5 * Register_{32bit} + 2 * ALU_{32bit}$

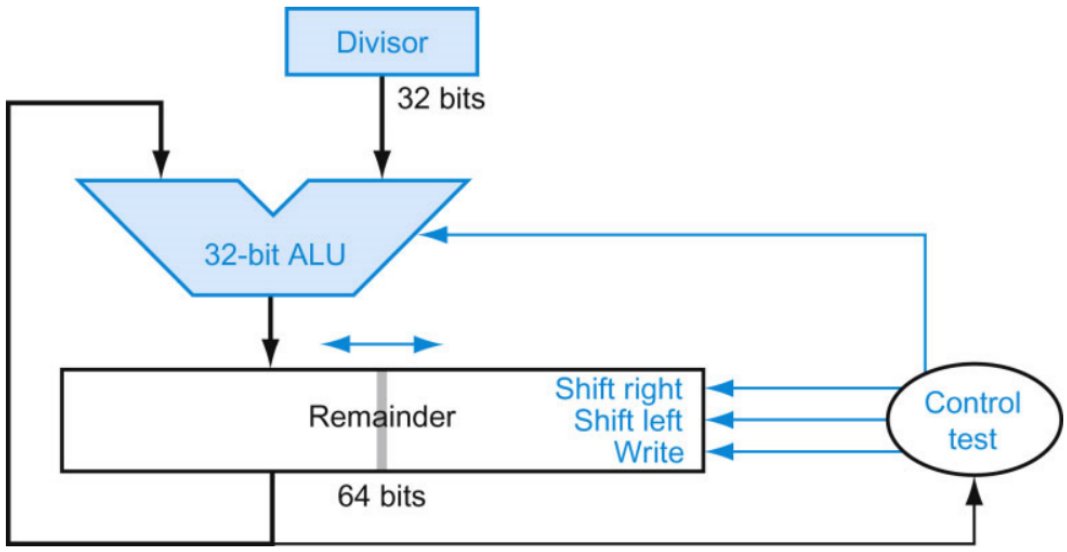
2. Optimized divider

- algorithm

```
store divisor in divisor[31:0]
store divident in the right half of Divident_remainder[63:0]
remainder will be stored in the Divident_remainder[63:32], quotient
will be stored in the Divident_remainder[31:0]

repeat(32):
    Divident_remainder <<= 1
    Divident_remainder[63:32] -= divisor
    if(remainder[63] == 0)
        Divident_remainder[0] = 1
    else
        Divident_remainder[0] = 0
        remainder += divisor
```

- graph



- expense

  $3 * Register_{32bit} + 1 * ALU_{32bit}$

Signed Multiplication and Division

1. Signed Multiplication
   - algorithm

```
step1: check whether they have same sign, this determin the sign of the
result
step2: convert all the number to their absolute value
step3: multiply them to get a positive number
step4: negate the result if the sign is negative
```

2. Signed division

- convention
  - dividend and remainder have the same sign
  - quotient is negative iff signs of dividend and divisor disagree

$$7 \div 2 = 3 \cdots 1$$
$$-7 \div 2 = -3 \cdots -1$$
$$7 \div -2 = -3 \cdots 1$$
$$-7 \div -2 = 3 \cdots -1$$

- algorithm

```
step1: check the sign of each operand, this determin the sign of the
quotient and remainder
step2: convert all the number to their absolute value
step3: use division to get a positive quotient and remainder
step4: negate the remainder if the dividend's sign is negative
step5: negate the quotient if the signs desagree
```