Santa Clara University

COEN 210

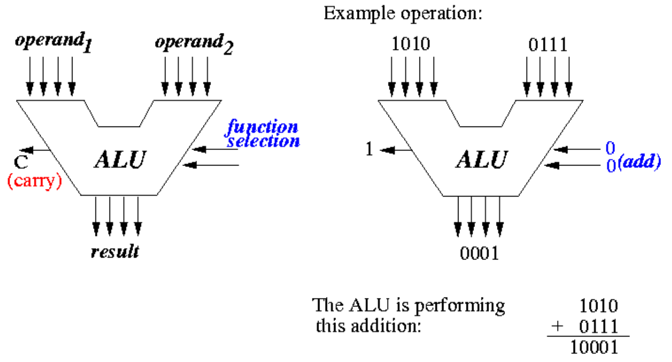
Chapter 3– Part 2. Arithmetic for Computers

Yan Cui

ycui4@scu.edu

Arithmetic Circuits

- Arithmetic circuits perform basic arithmetic operations
  - Such as addition, subtraction and etc.
- They join together and form a Arithmetic Logic Unit (ALU)
  - Like a Swiss army knife, single unit perform multiple arithmetic and logical operations

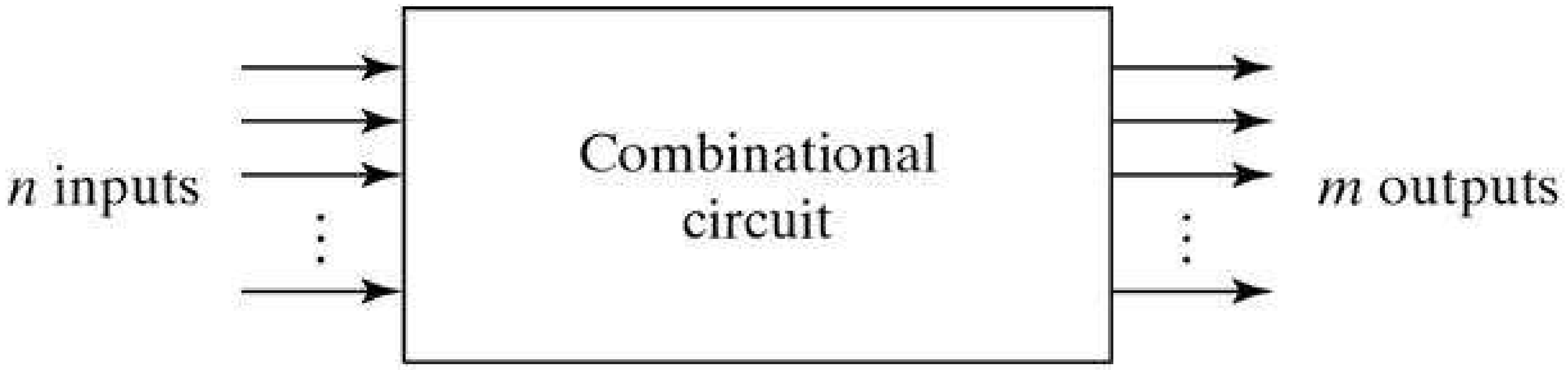


- An ALU is the part of a computer processor(CPU) that carries out arithmetic and logic operations on the operands in computer instruction words.

Combinational Circuits

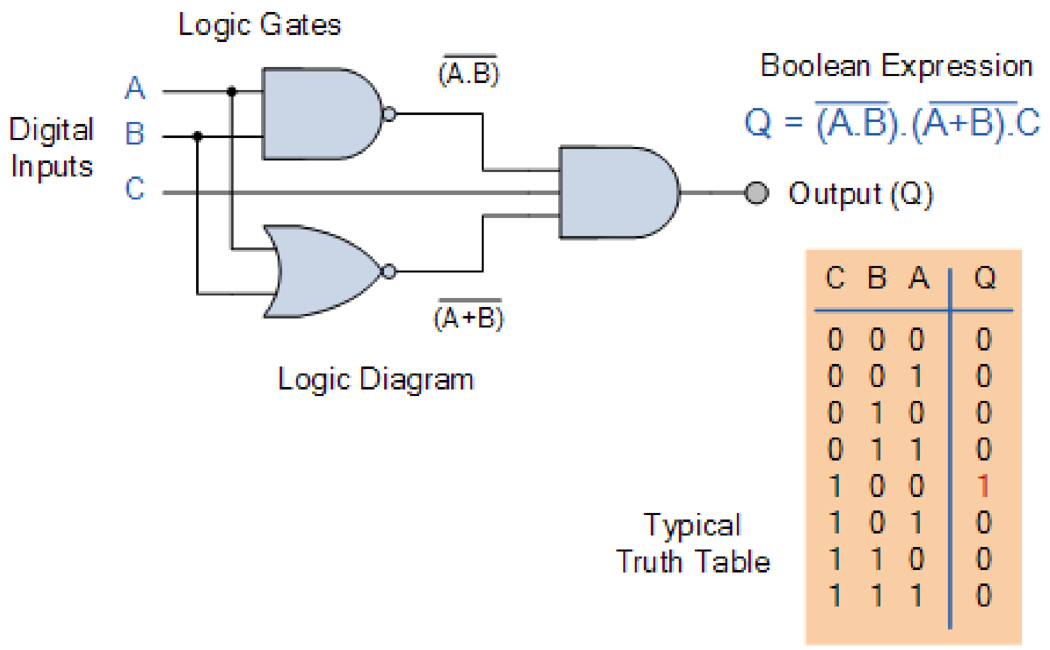
What is a Combinational Circuit

- Consists of an **acyclic** network of logic gates
  - Continuously responds to changes in inputs
  - Outputs become (after a short delay) boolean functions of the inputs.
  - memoryless digital logic circuits
  - whose output at any instant in time depends only on the combination of its inputs



Three ways for a combinational logic circuit

- Boolean Algebra
- Truth Table
- Logic Diagram

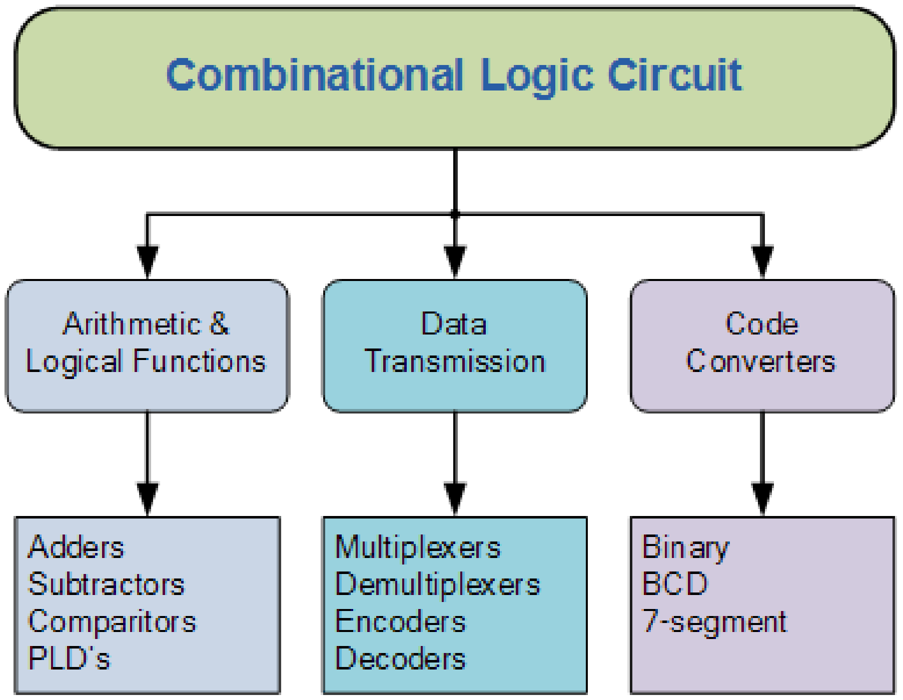


$Q = \overline{(A.B)}.\overline{(A+B)}.C$

C	B	A	Q
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

Classification of Combinational Logic

Multiplexers, *Decoders*, *Full* and *Half Adders*, Comparators, etc.



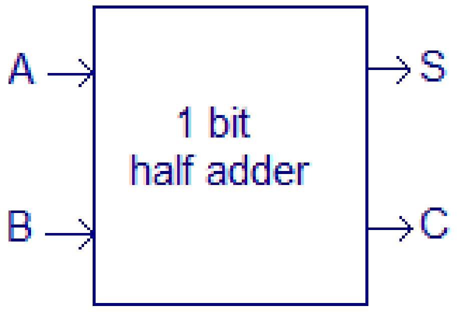
Addition & Subtraction

Single-Bit Half Adder

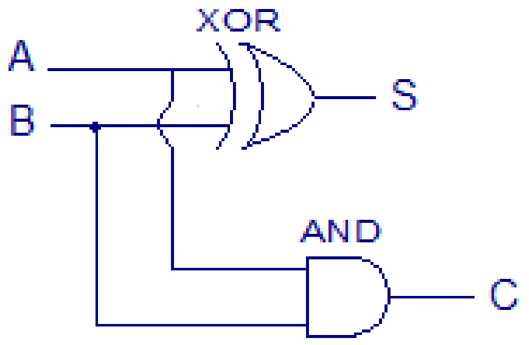
- Input : single bit A and B
- Output: single bit Sum S and Carry C

Inputs		Outputs	
A	B	S	C
0	0	0	0
1	0	1	0
0	1	1	0
1	1	0	1

Truth table



Schematic

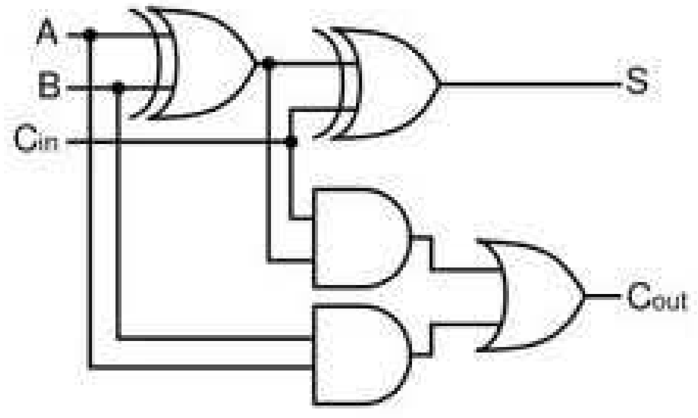


Realization

Single Bit Full Adder

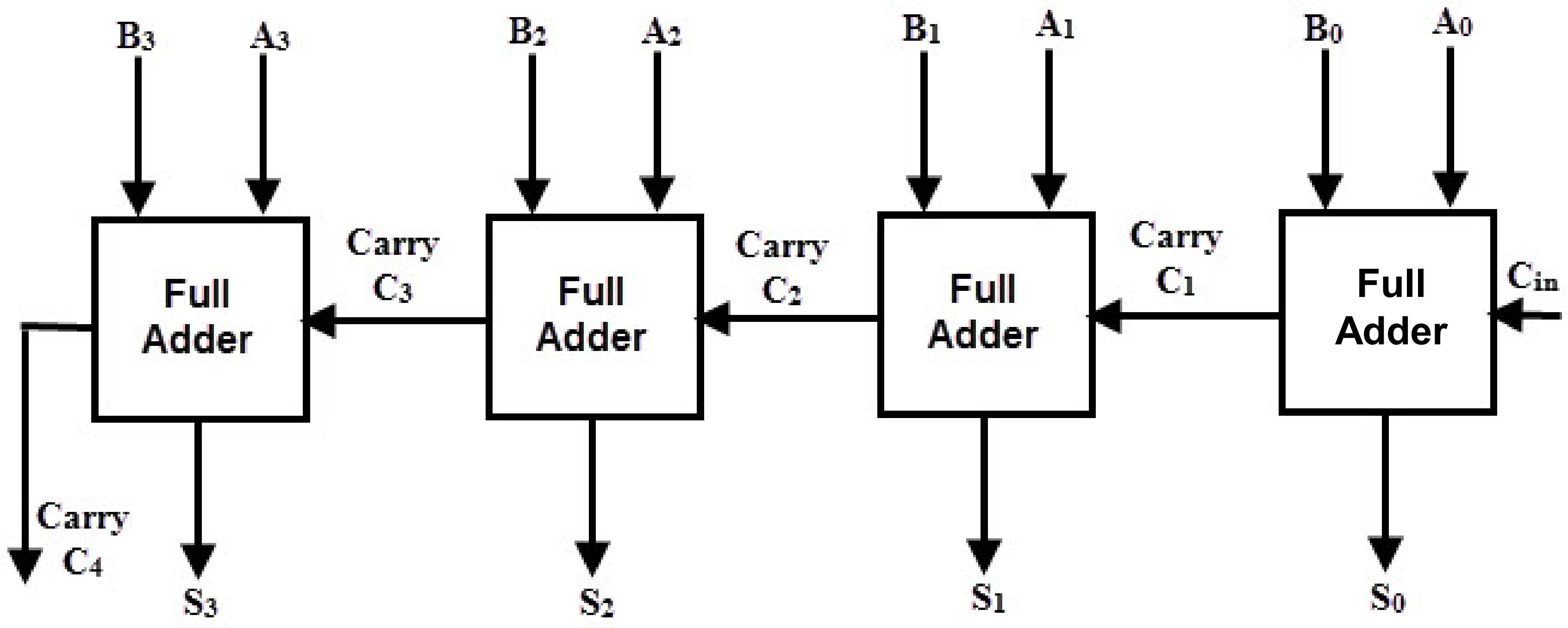
- same as half adder except it has an carry input
- Output: sum and carry output

A	B	Cin	Cout	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



Cin: carry input
Cout: carry output

Adding a Pair of 4-bit Integers

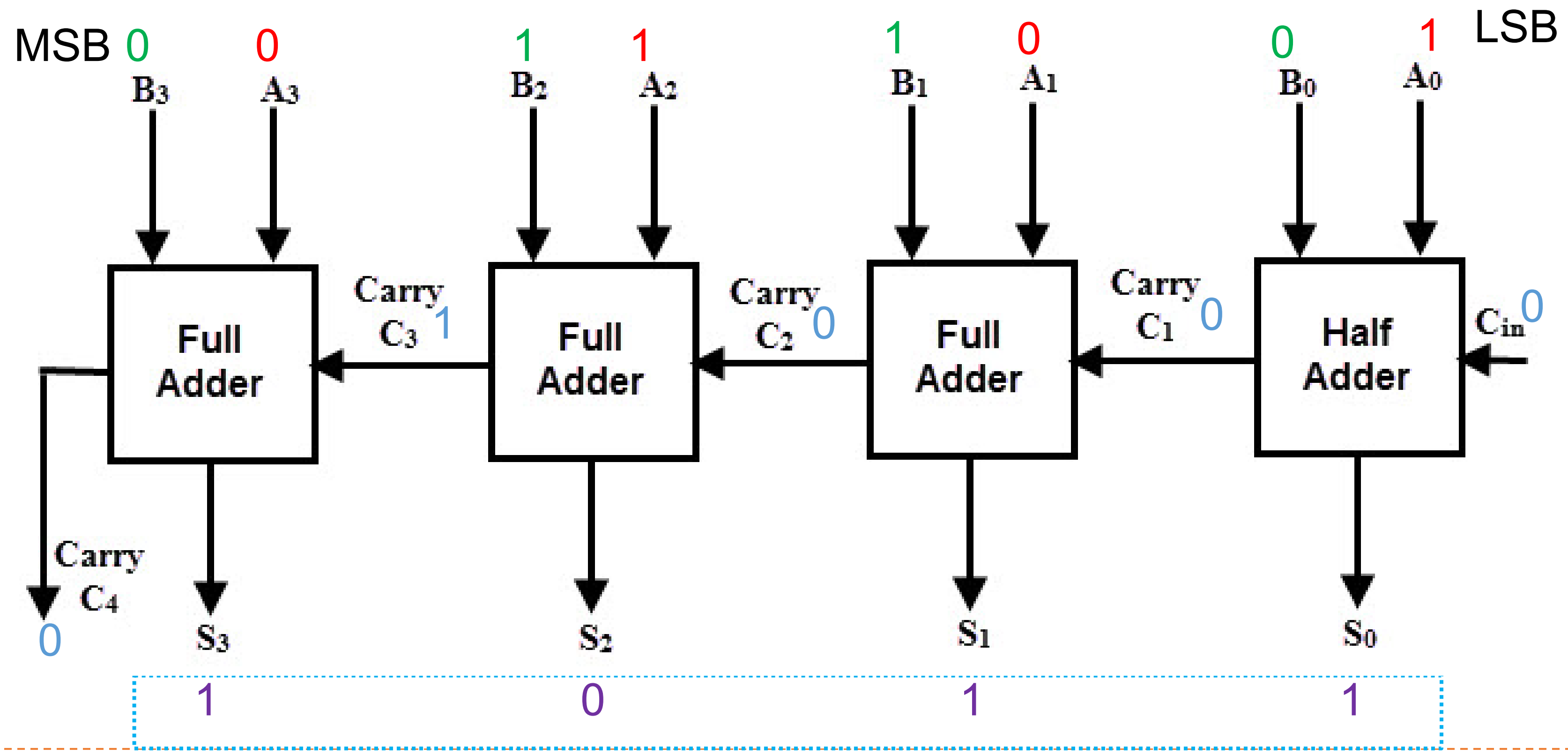


Example:

Let A = 0101_2 and B=0110_2, What is A + B ?

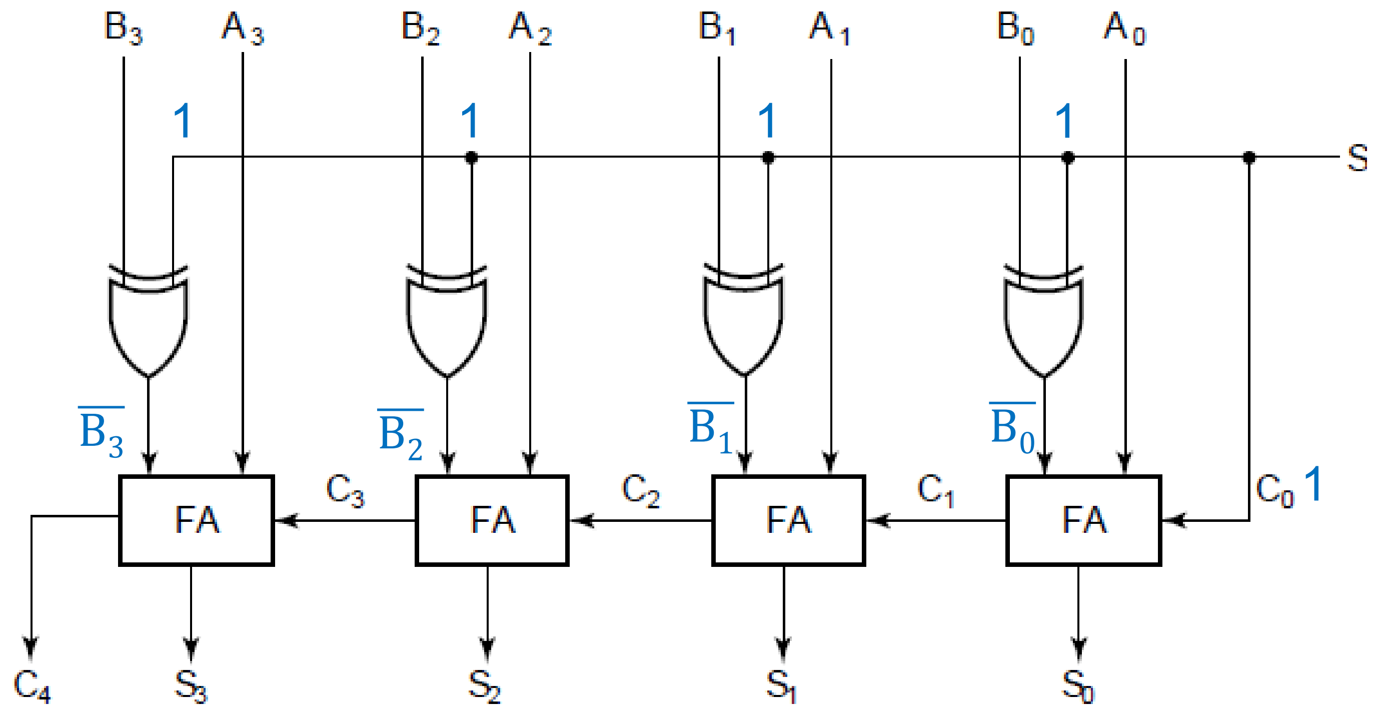
A 4-bit Ripple Carry Adder

- A0 – A3 1st 4-bit number $0\ 1\ 0\ 1_2$
- B0 – B3 2nd 4-bit number $0\ 1\ 1\ 0_2$
- A + B ?



How about Subtraction?

- $-B = \overline{B} + 1 \rightarrow A - B = A + \overline{B} + 1$
- S=1 ➔ subtraction



Overflow

Preface

- "Overflow" is only meaning for systems with fixed number of digits
  - E.g, computers
- Overflow occurs when there are insufficient bits in a binary number representation to portray the result of an arithmetic operation
- Overflow can apply to both signed and unsigned

Overflow Summary - Addition for Signed Number

- Adding a positive operand and a negative operand, no overflow

- Adding two positive operands
  - Overflow if result is negative (i.e., sign bit is 1)

- Adding two negative operands
  - Overflow if result is positive (i.e., sign bit is 0)

Overflow Examples – Addition of Signed Number

Overflow!

$$\begin{array}{rrr} & 0011 & 3 \\ + & 0110 & 6 \\ \hline & 1001 & -7 \end{array}$$

OK!

$$\begin{array}{rrr} & 1101 & -3 \\ + & 0110 & 6 \\ \hline & 0011 & 3 \end{array}$$

OK!

$$\begin{array}{rrr} & 1011 & -5 \\ + & 1110 & -2 \\ \hline & 1001 & -7 \end{array}$$

Overflow!

$$\begin{array}{rrr} & 1001 & -7 \\ + & 1000 & -8 \\ \hline & 0001 & 1 \end{array}$$

Overflow Summary – Subtraction for Signed Number

- Subtracting between two positive or two negative operands
  - no overflow
- Subtracting a positive operand from a negative operand
  - Overflow if result is positive (i.e., result sign bit is 0)
- Subtracting a negative operand from a positive operand
  - Overflow if result is negative (i.e., sign bit is 1)

Overflow Examples – Subtract for Signed Number

OK!

```
     0011     3
-    0110     6
-------------
     0011
     1001
+       1
-------------
     1101   - 3
```

Overflow!

```
     1101    -5
-    0110     6
-------------
     1101
     1001
+       1
-------------
     0111     7
```

OK!

```
     1011    -5
-    1110    -2
-------------
     1011
     0001
+       1
-------------
     1101   - 3
```

Overflow!

```
     0001     1
-    1000    -8
-------------
     0001
     0111
+       1
-------------
     1001    -7
```

How about Unsigned Numbers?

- For addition,
  - Overflow happens when carry bit is 1
- For subtraction
  - Underflow happens when first operand is less than second operand
- In RISC-V
  - No overflow detection
    - No special instruction set support for overflow checks on base instruction sets
      - Some overflow checks can be implemented using RISC-V branches
        - E.g. unsigned addition, add t0, t1, t2; bltu t0, t1, overflow.
    - Floating point generate a exception

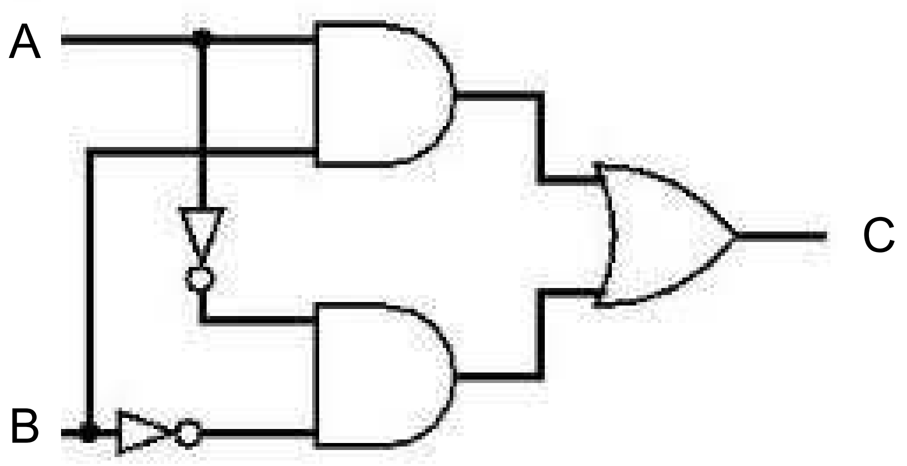
Dealing with Overflow in High-Level Languages

- Some languages (e.g., C and Java) ignore overflow
  - So C compiler always use MIPS `addu`, `addui`, `subu` instructions, even for signed variables

- Other languages (e.g., Ada, Fortran) require raising an exception

One-bit Equality Comparator

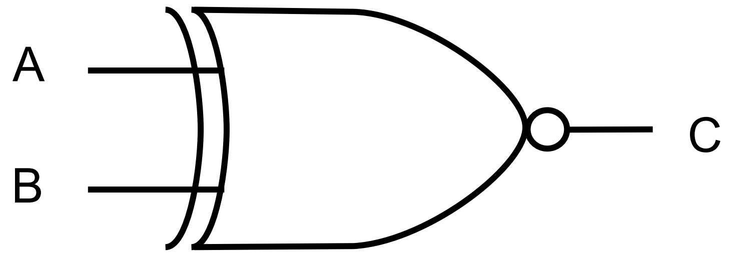
- Output 1 when both bits are same
  - C = (A * B) || (A' * B')
  - When A = B, either A = B = 1 (HIGH) or A = B = 0 (LOW)
- *Can it be simpler?*



One-Bit Equality Comparator

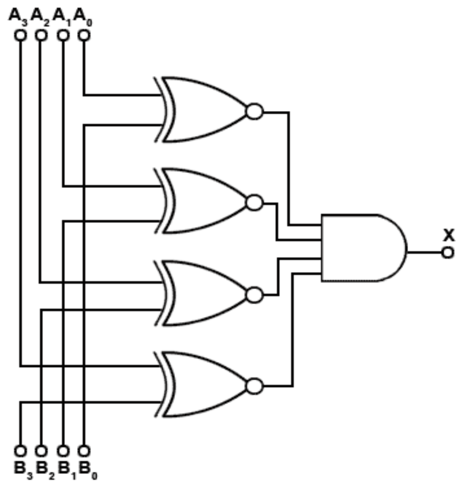
- Output 1 when both bits are same
  - C = (A * B) || (A' * B')

- *Can it be simpler?*
  - *XNOR gate*



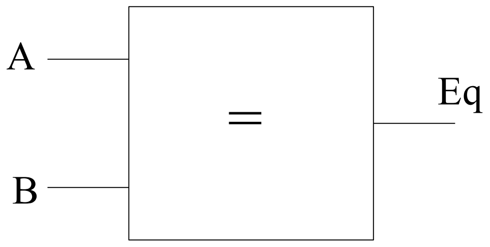
Four Bit Equality Comparator

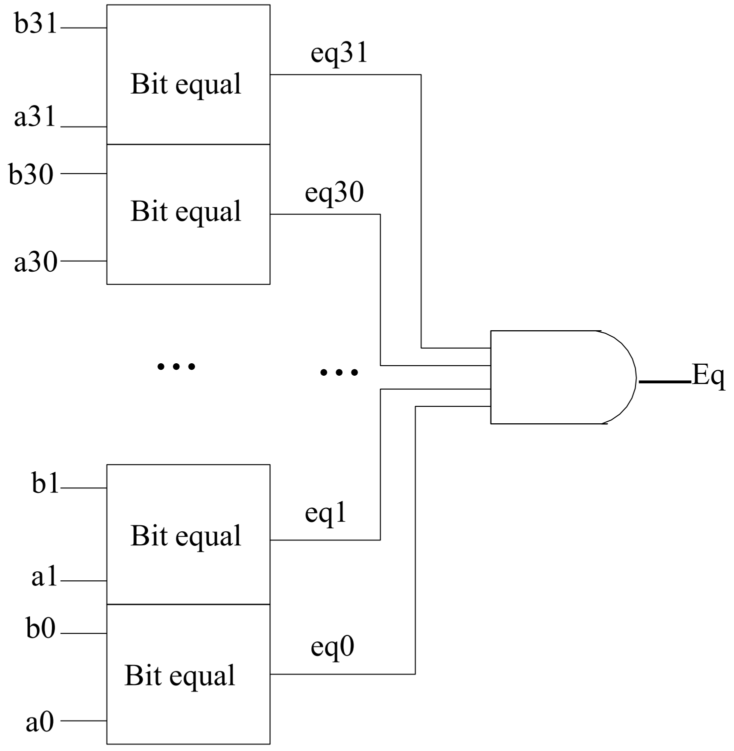
- The circuit of the equality comparator consists of an exclusive NOR gate (XNOR) per pair of input bits. If the two inputs are identical (both 1s or both 0s) an output of logic 1 is obtained.
- The outputs of the XNOR gates are then combined in an AND gate, the output of which will be 1, only when all the XNOR gates indicate matched inputs.



Word Comparator

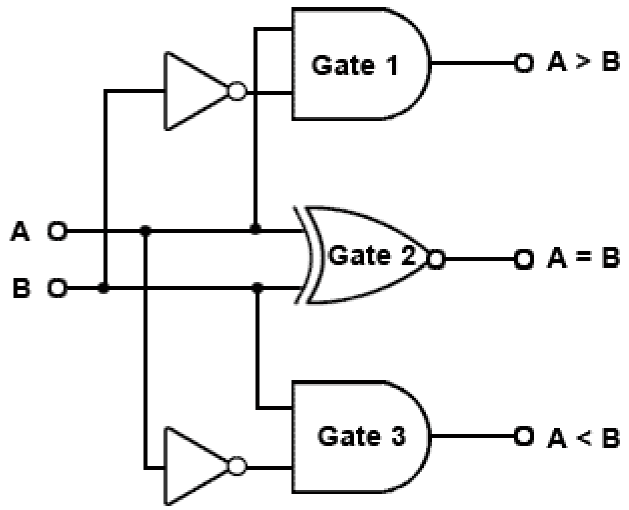
- Assume 32bit word





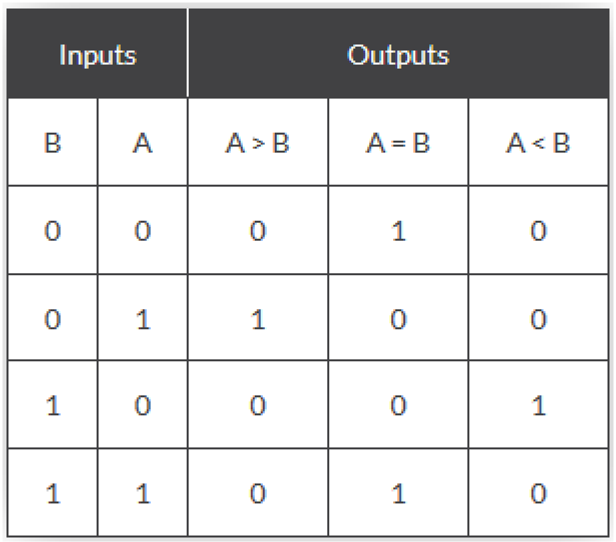
Magnitude Comparators

- Indicate equality A = B, A is greater than B (A>B), A is less than B (A<B)
- A simple 1-bit magnitude comparator is shown in figure
  - Gate 1 produces the function A>B
  - gate 3 gives A<B
  - gate 2 is an XNOR gate giving an equality output.



A simple 1-bit magnitude comparator

Comparator Truth Table



Inputs		Outputs		
B	A	A > B	A = B	A < B
0	0	0	1	0
0	1	1	0	0
1	0	0	0	1
1	1	0	1	0

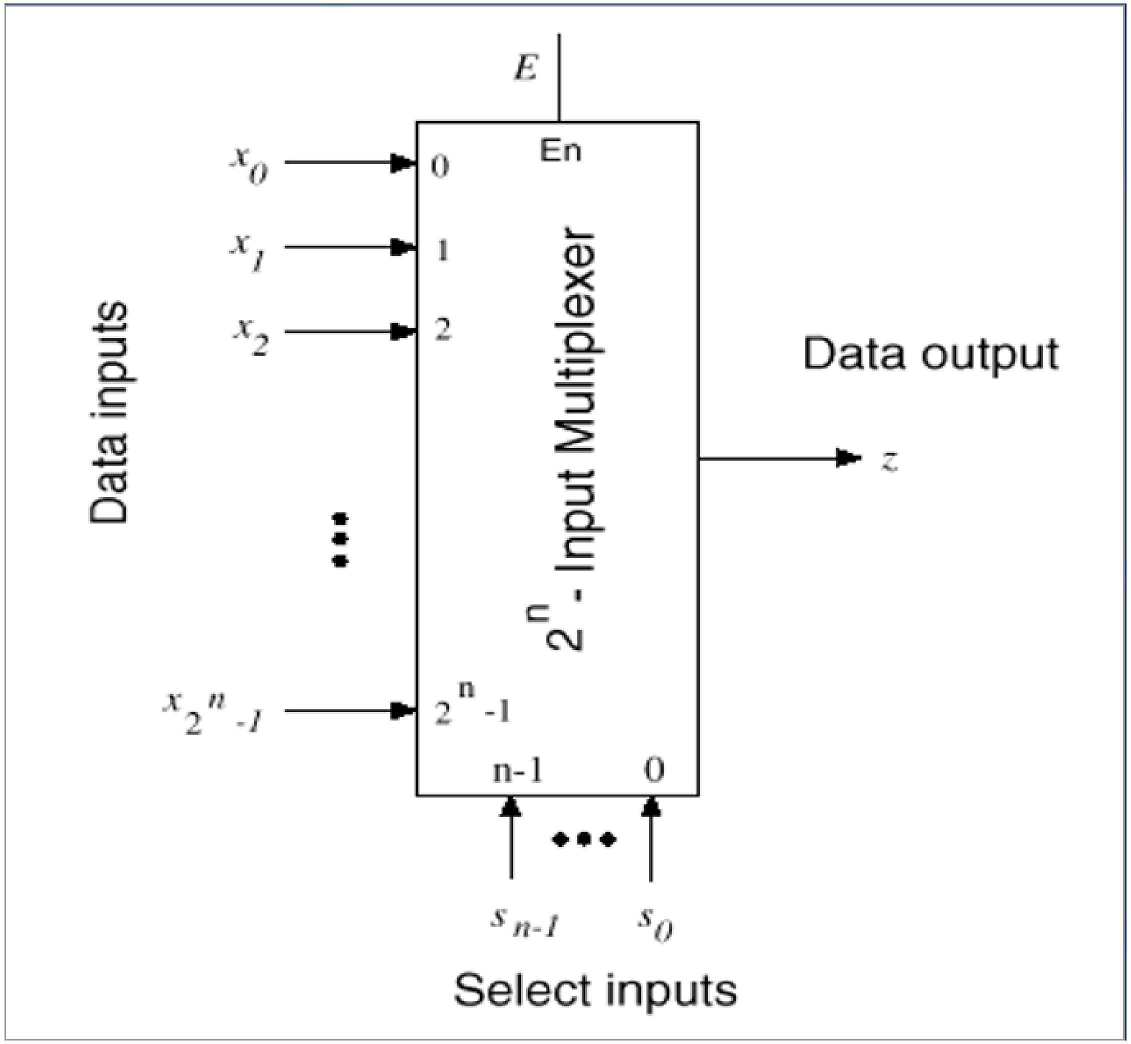
Multiplexer

- *selects* one of several input signals and passes it on to the output
  - A combinational logic circuit
  - Also know as the “selector” circuit

- Selection is controlled by a particular set of inputs lines whose # depends on the # of the data input lines.

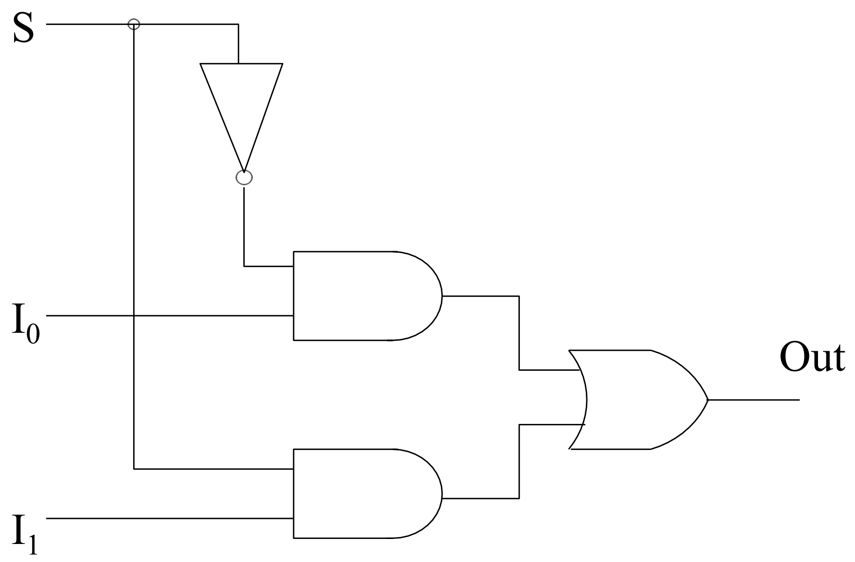
- For a 2^n-to-1 multiplexer, there are 2^n data input lines and n selection lines whose bit combination determines which input is selected.
- shortened to “MUX” or “MPX”

Multiplexer (Cont.)



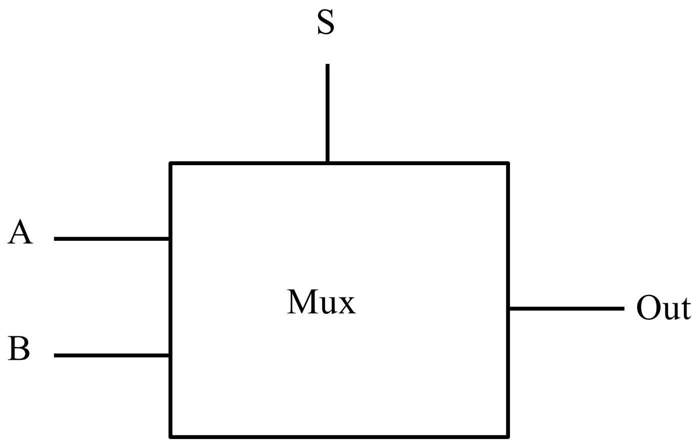
Example: 2-to-1-Line Multiplexer

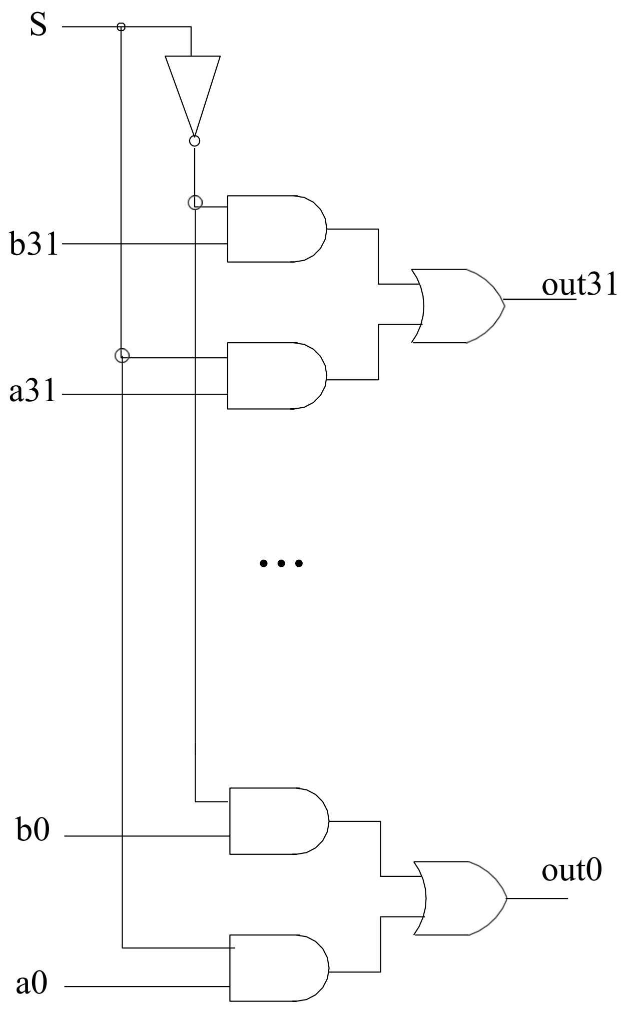
- Since $2 = 2^1$, $n = 1$
- The single selection variable S has two values:
  - S = 0 selects input I_0 Out = I_0
  - S = 1 selects input I_1 Out = I_1
- The circuit:



Word Multiplexer

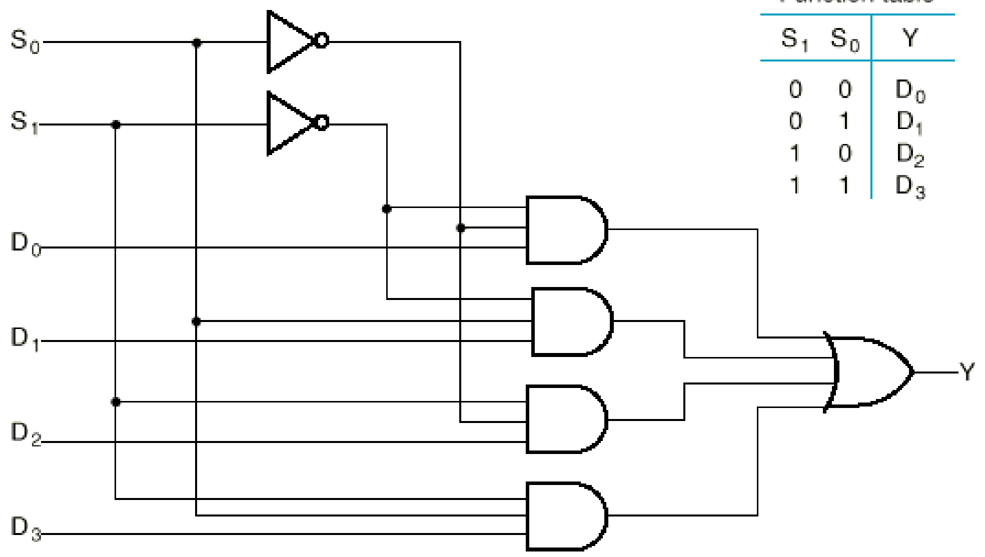
- Output word A when S == 1
  - A: 32-bit word a31 a30 a29….a1 a0
- Output word B when S == 0
  - B: 32-bit word b31 b30 b29…b1 b0





Example: 4-to-1 MUX

- Since $4 = 2^2$, $n = 2$
- The two selection variable S_0, S_1 has four values:
  - $S_1 = 0$, $S_0 = 0$ selects input D_0, $Y = D_0$
  - $S_1 = 0$, $S_0 = 1$ selects input D_1, $Y= D_1$
  - $S_1 = 1$, $S_0 = 0$ selects input D_3, $Y = D_3$
  - $S_1 = 1$, $S_0 = 1$ selects input D_4, $Y = D_4$
- The circuit:



Function table

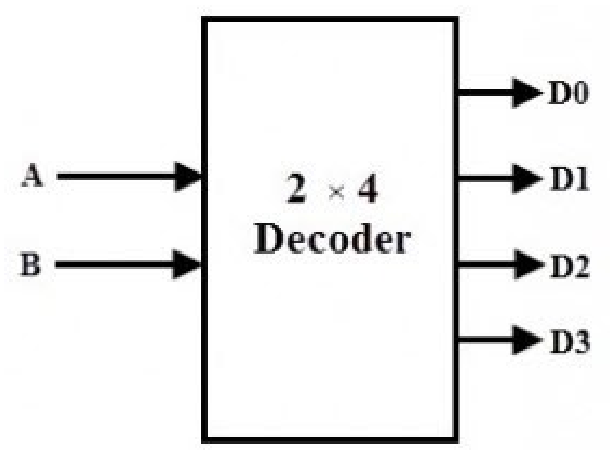
S_1	S_0	Y
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3

Binary Decoder

- A combinational circuit that converts binary information from *n* coded inputs to a maximum 2^n coded outputs
- Accepts a set of inputs that represents a binary number and activates only the output that corresponds to that input number.
  - *n-to-* 2^n decoder
  - Only 1 output is asserted for each input combination
  - Translates the n-bit input into a signal that corresponds to the binary value
  - Transforms n-bit input signals into an equivalent code using 2^n outputs.

Example: 2-to-4 Binary Decoder

- The two input lines are decoded into four output lines.
- Only one output line will be active (logic 1) and the other three output lines are maintained at logic zero



2 to 4 Decoder

Example: 2-to-4 Binary Decoder (Cont.)

- Truth Table:

Binary Value	A_1	A_0	D_3	D_2	D_1	D_0
0	0	0	0	0	0	1
1	0	1	0	0	1	0
2	1	0	0	1	0	0
3	1	1	1	0	0	0

Accepts a set of inputs that represents a binary number and activates only the output that corresponds to that input number.

- If the two inputs are both 0, then the output of D_0 is logic 1 and the other outputs are logic 0.
- If both the inputs are 1, then the output of D_3 is logic 1 and other output pins are logic 0.
- If the input pins A_1= 0 & A_0 = 1 then the D_1 will be logic 1, other output are 0.
- if the inputs A_1= 1 & A_0 = 0 then the output D_2 will be logic 1, others are 0.

Example: 2-to-4 Binary Decoder (Cont.)

- Boolean expressions of each output:

Equations

$$D_0 = \overline{A_1} \cdot \overline{A_0}$$

$$D_1 = \overline{A_1} \cdot A_0$$

$$D_2 = A_1 \cdot \overline{A_0}$$

$$D_3 = A_1 \cdot A_0$$

Example: 2-to-4 Binary Decoder (Cont.)

- By using the above equation, we can implement the logic circuit design:

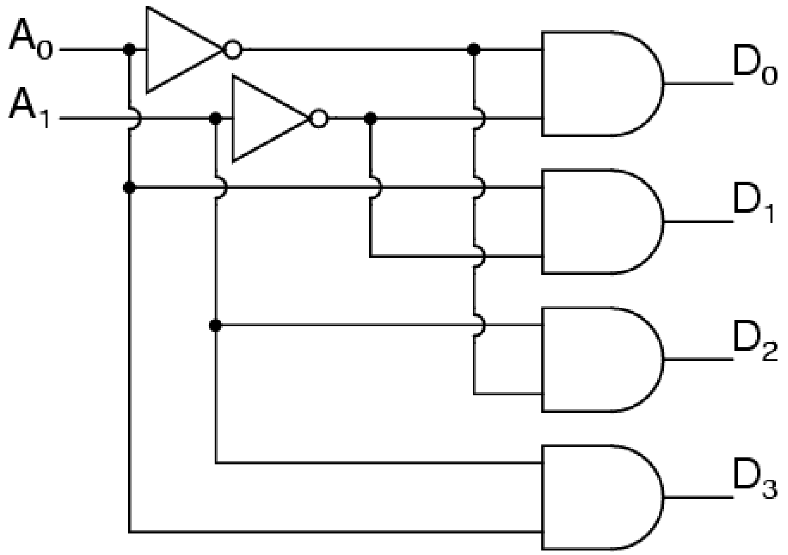
Equations

$D_0 = \overline{A_1} \cdot \overline{A_0}$

$D_1 = \overline{A_1} \cdot A_0$

$D_2 = A_1 \cdot \overline{A_0}$

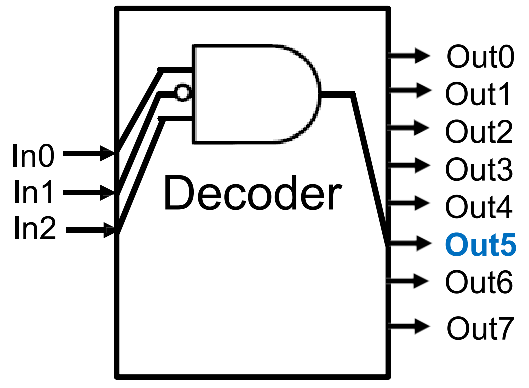
$D_3 = A_1 \cdot A_0$



- There are two NOT gates and four AND gates are used.
- The output depends on the product of the two inputs

Example: 3-to-8 Binary Decoder

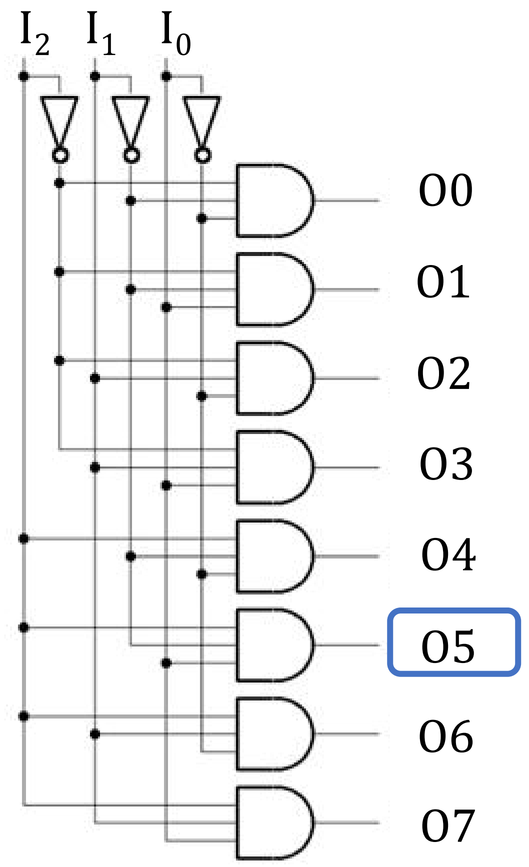
- Each output can be implemented with a 3-input AND gate
- Our example would need 8 3-input AND gates



3-to-8 Decoder

Inputs			Outputs							
I2	I1	I0	O7	O6	O5	O4	O3	O2	O1	O0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

Example: 3-to-8 Binary Decoder (Cont.)



Logic Circuit

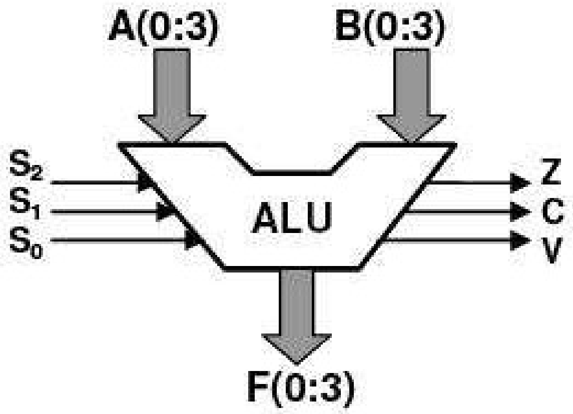
Arithmetic Logic Unit (ALU)

What is ALU?

- Perform arithmetic and bitwise operations on integer binary numbers
- Multiple functions: add, subtract, and, xor, others
- Combinational logic to perform functions.
- Control signals select function to be performed.
- Modular with multiple instances of 1-bit ALU

An Example of 4-bit ALU

- Combination logic : continuously responding to inputs
- 3-bit control signal selects the operation
- In addition to output, also compute flags

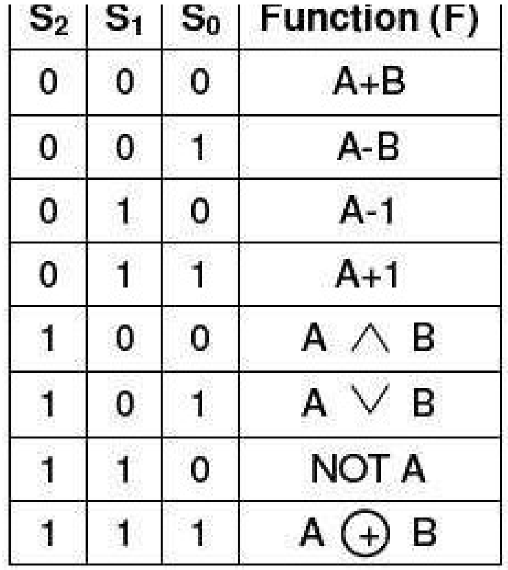


Z, C and V are status flags

Z = 1 if F=0

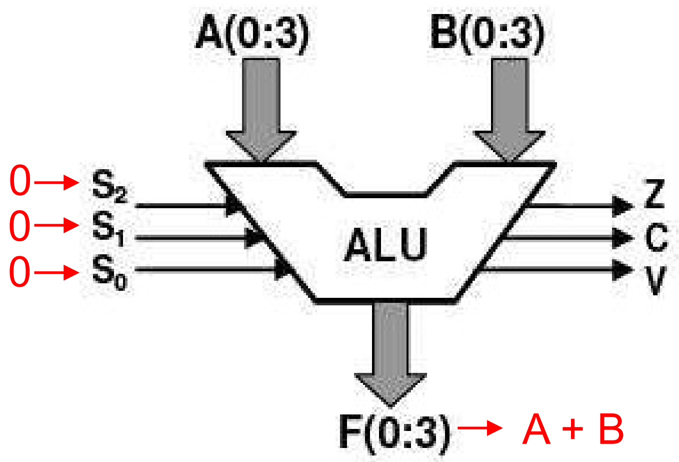
C = Carry or Borrow

V = Overflow



S_2	S_1	S_0	Function (F)
0	0	0	A+B
0	0	1	A-B
0	1	0	A-1
0	1	1	A+1
1	0	0	$A \wedge B$
1	0	1	$A \vee B$
1	1	0	NOT A
1	1	1	$A \oplus B$

An Example of 4-bit ALU

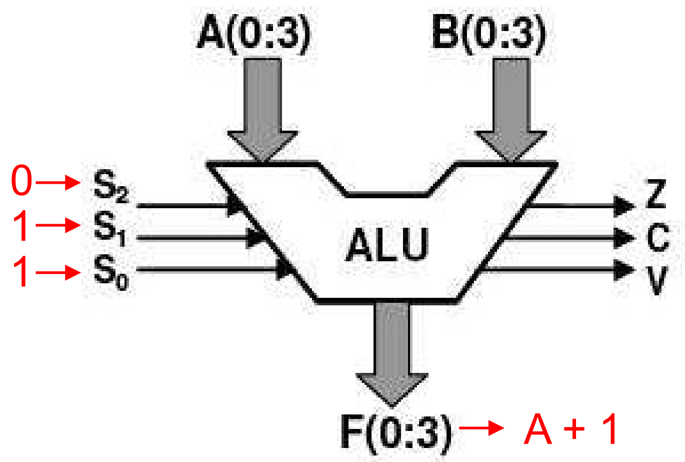


Z, C and V are status flags

Z = 1 if F=0
C = Carry or Borrow
V = Overflow

S_2	S_1	S_0	Function (F)
0	0	0	A+B
0	0	1	A-B
0	1	0	A-1
0	1	1	A+1
1	0	0	$A \wedge B$
1	0	1	$A \vee B$
1	1	0	NOT A
1	1	1	$A \oplus B$

An Example of 4-bit ALU



Z, C and V are status flags

Z = 1 if F=0

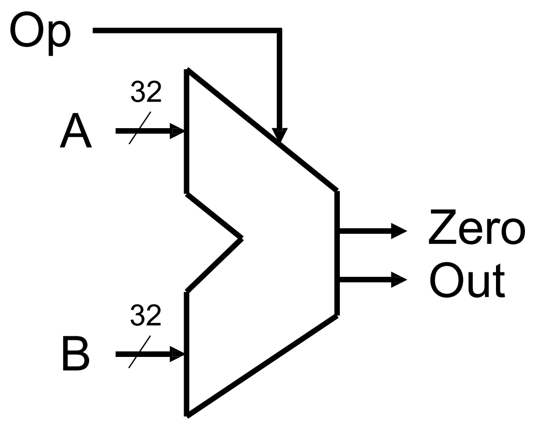
C = Carry or Borrow

V = Overflow

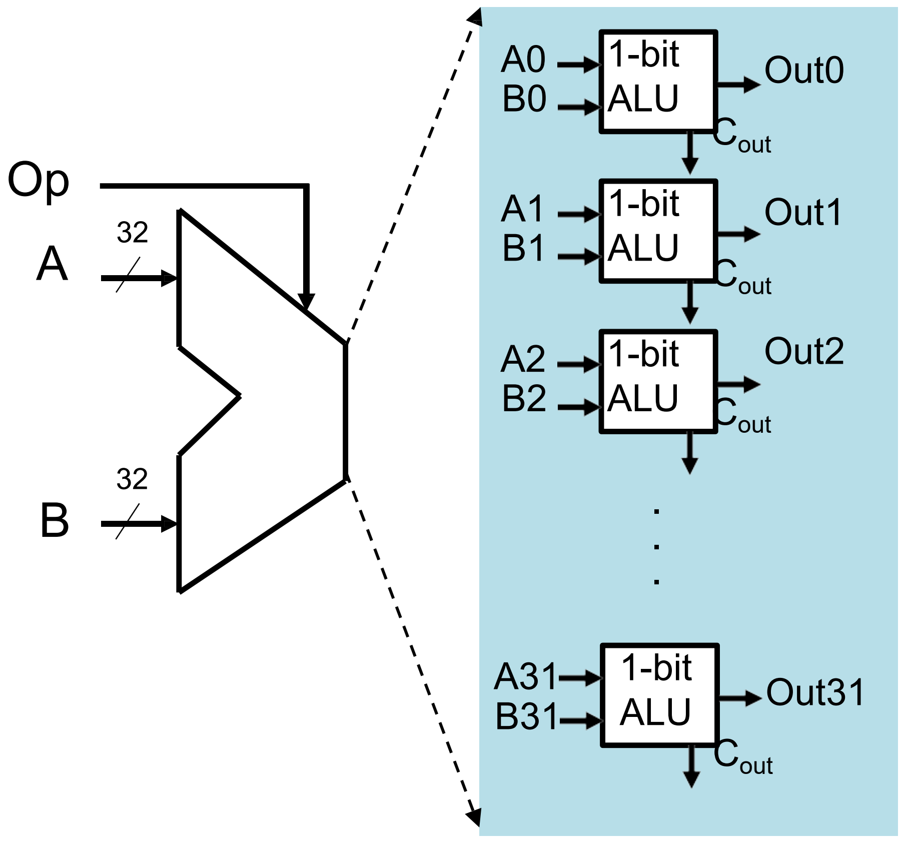
S_2	S_1	S_0	Function (F)
0	0	0	A+B
0	0	1	A-B
0	1	0	A-1
0	1	1	A+1
1	0	0	$A \wedge B$
1	0	1	$A \vee B$
1	1	0	NOT A
1	1	1	$A \oplus B$

Let Us Design a 32-bit ALU

- Inputs
  - Two 32-bit binary values
  - 3-bit control input, Op
- Outputs
  - 32-bit calculated result, Out
  - 1-bit output flag, Zero
    - Set to 1 if Out are all 0's
    - Useful for checking branch conditions (i.e. if (A == B))
- Functions
  - 32-bit-wide arithmetic operations
    - Op == 000 : Out = A AND B
    - Op == 001 : Out = A OR B
    - Op == 010 : Out = A + B
    - Op == 110 : Out = A - B

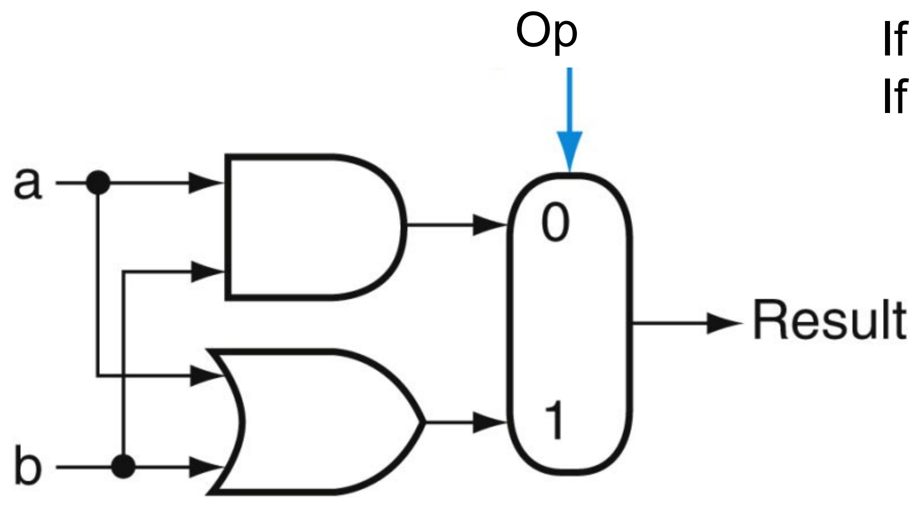


… Which is 32 units of 1-bit ALU.



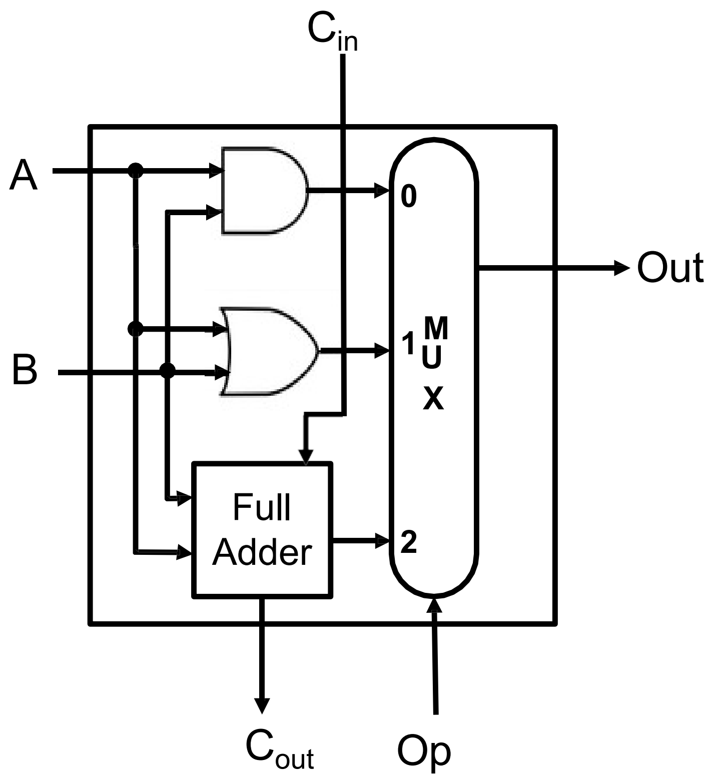
C_{out} : Carry-out

1-bit AND/OR ALU



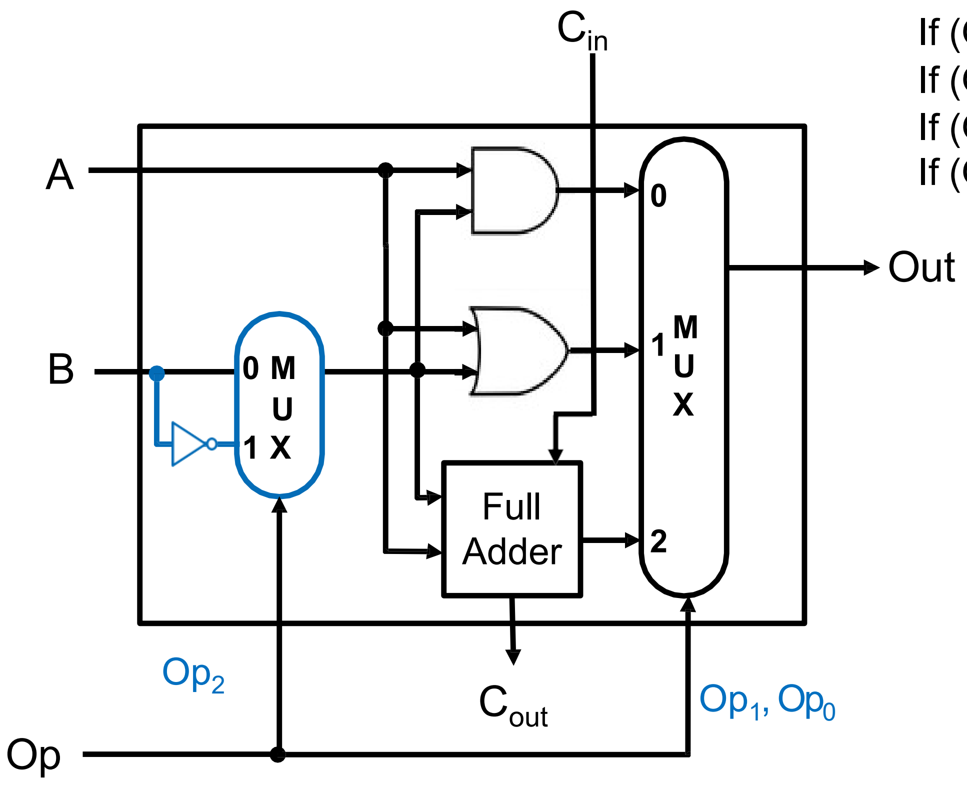
If (Op == 0), Out = A AND B
If (Op == 1), Out = A OR B

Merge Full Adder



If (Op == 00), Out = A AND B
If (Op == 01), Out = A OR B
If (Op == 10), Out = A + B

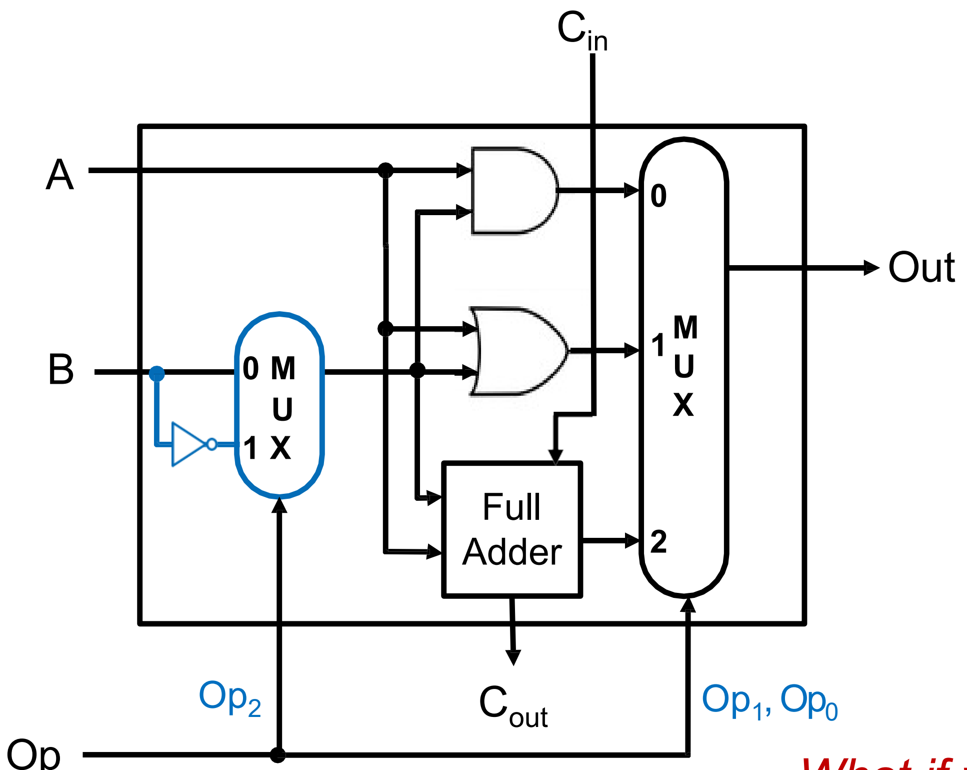
Merge More Logics



If (Op ($Op_2Op_1Op_0$) == 000), Out = A AND B
If (Op ($Op_2Op_1Op_0$) == 001), Out = A OR B
If (Op ($Op_2Op_1Op_0$) == 010), Out = A + B
If (Op ($Op_2Op_1Op_0$) == 110), Out = A – B

Why designer choose subtract op to be 110?

Merge More Logics – What if subtract is op 11?



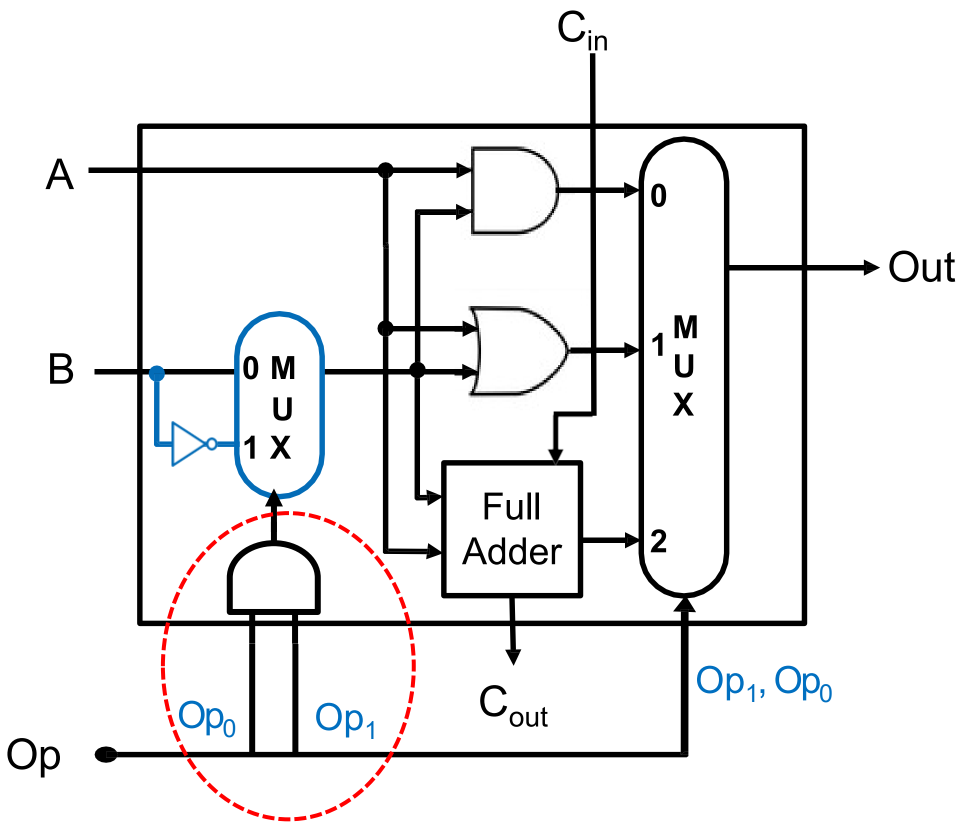
If (Op ($Op_2Op_1Op_0$) == 000), Out = A AND B
If (Op ($Op_2Op_1Op_0$) == 001), Out = A OR B
If (Op ($Op_2Op_1Op_0$) == 010), Out = A + B
If (Op ($Op_2Op_1Op_0$) == 110), Out = A – B

$\Downarrow$

If (Op (Op_1Op_0) == 00), Out = A AND B
If (Op (Op_1Op_0) == 01), Out = A OR B
If (Op (Op_1Op_0) == 10), Out = A + B
If (Op (Op_1Op_0) == 11), Out = A – B

What if we want subtraction to be Op =11?

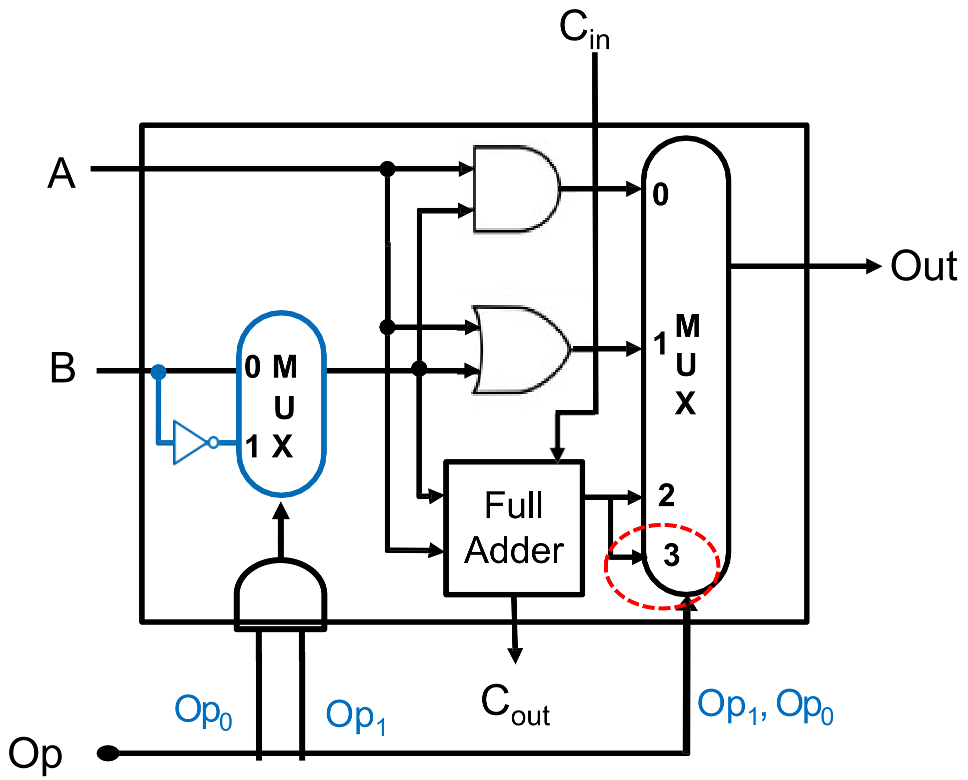
Improvement– What if subtract is op 11? (1)



If (Op (Op_1Op_0) == 00), Out = A AND B
If (Op (Op_1Op_0) == 01), Out = A OR B
If (Op (Op_1Op_0) == 10), Out = A + B
If (Op (Op_1Op_0) == 11), Out = A – B

Step1: AND gate of Op_0 and O Op_1 to simulate Op_2

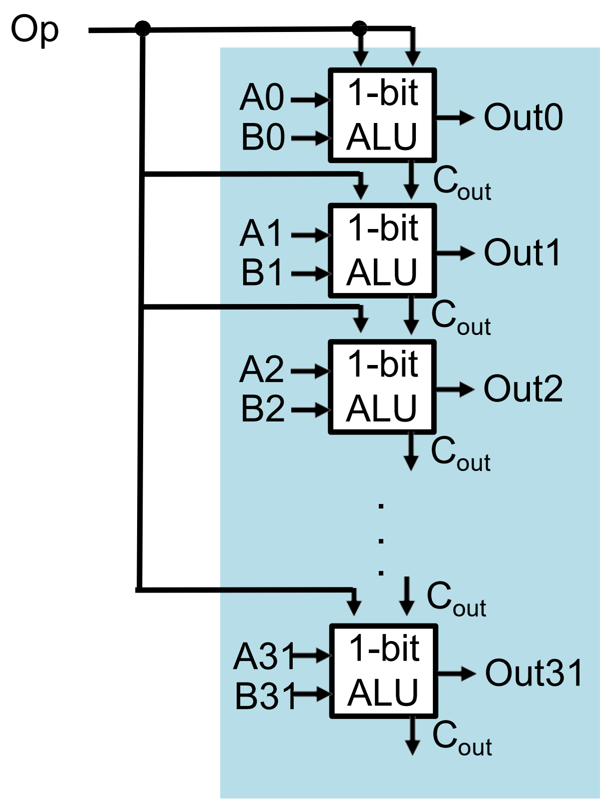
Improvement– What if subtract is op 11? (2)

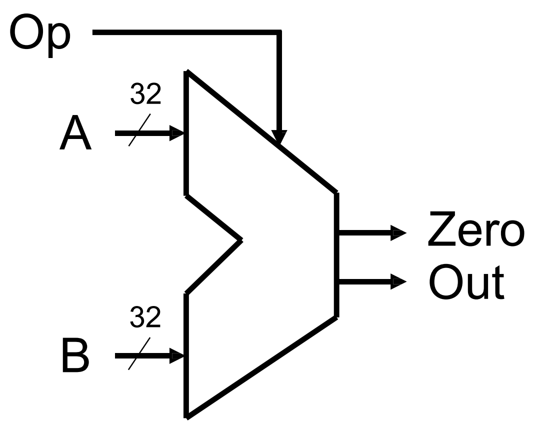


If (Op (Op_1Op_0) == 00), Out = A AND B
If (Op (Op_1Op_0) == 01), Out = A OR B
If (Op (Op_1Op_0) == 10), Out = A + B
If (Op (Op_1Op_0) == 11), Out = A – B

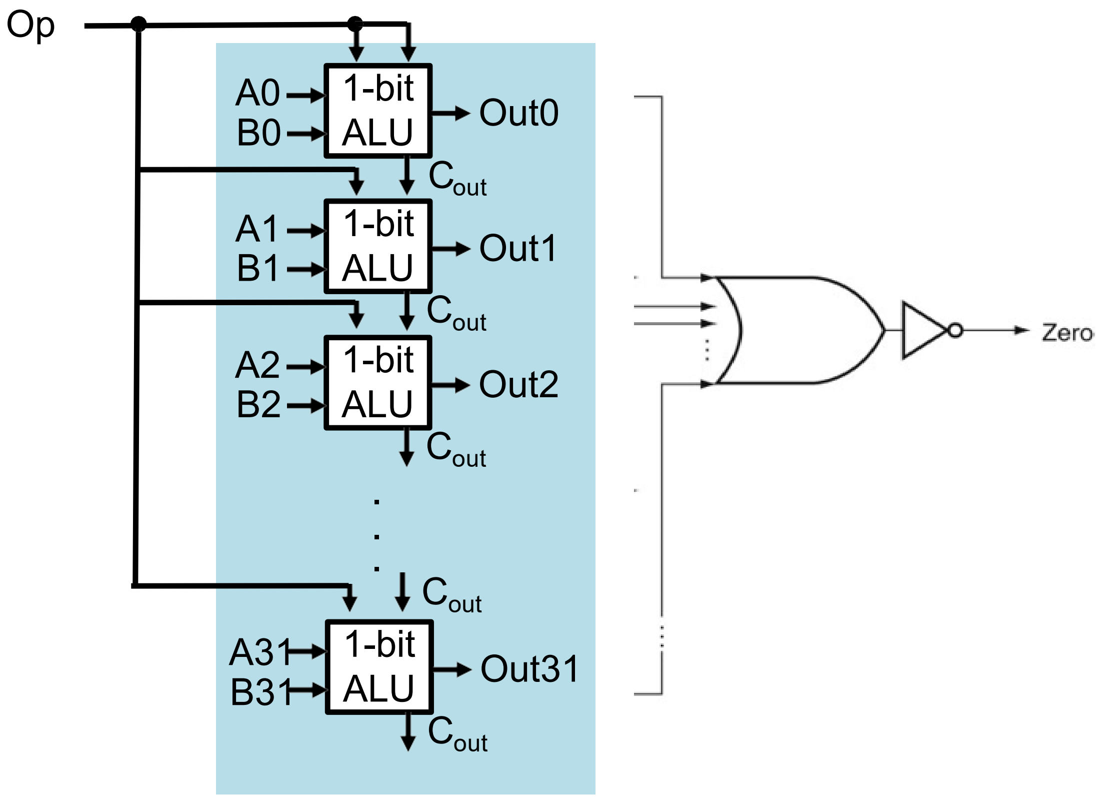
Step 2: Route full adder output to MUX input 3 as well

Put It Altogether



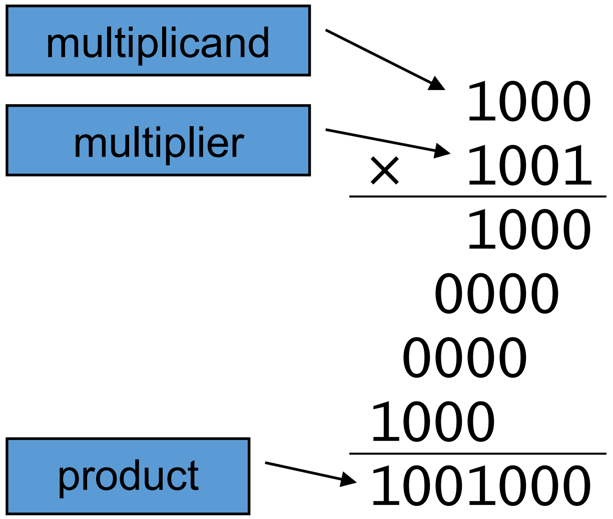


What about Zero Flag?



Multiplication

Multiplication – Learn from Paper-n-Pencil

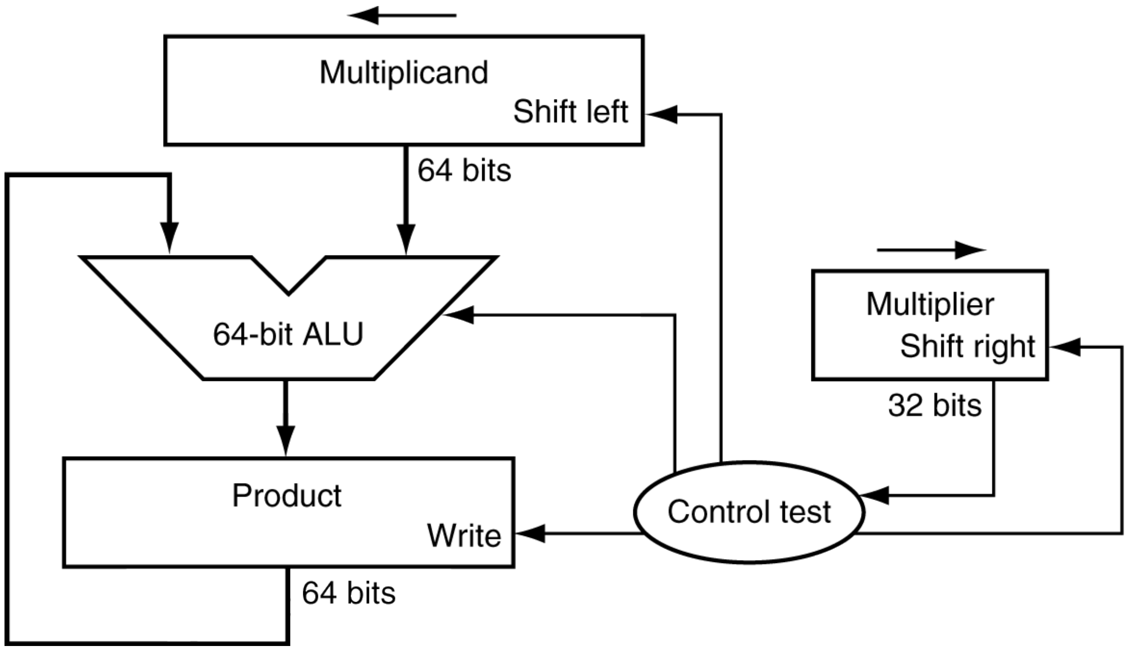


```
     1000
  ×  1001
  -------
     1000
    0000
   0000
  1000
  -------
  1001000
```

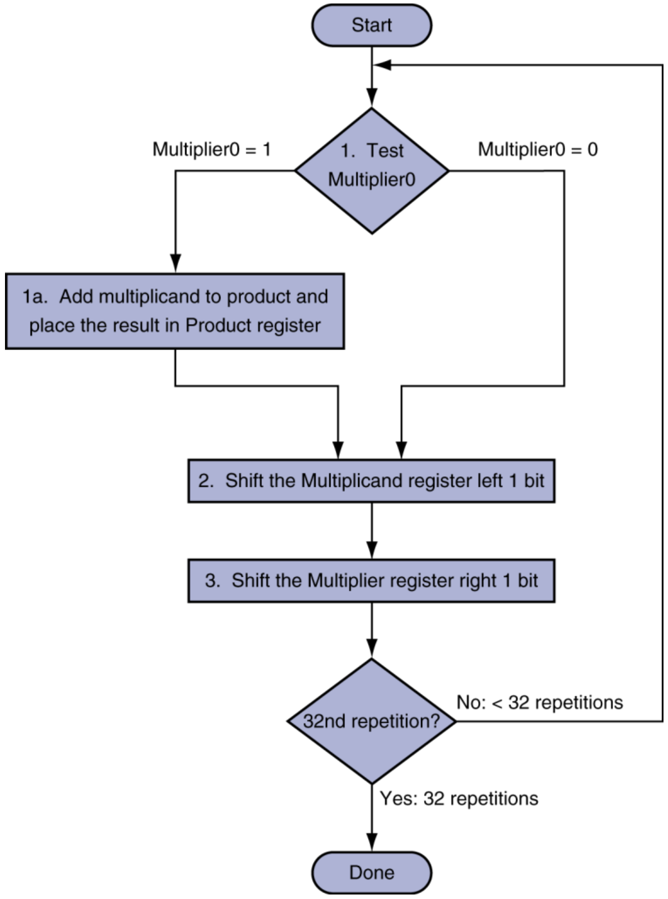
Length of product is the sum of operand lengths

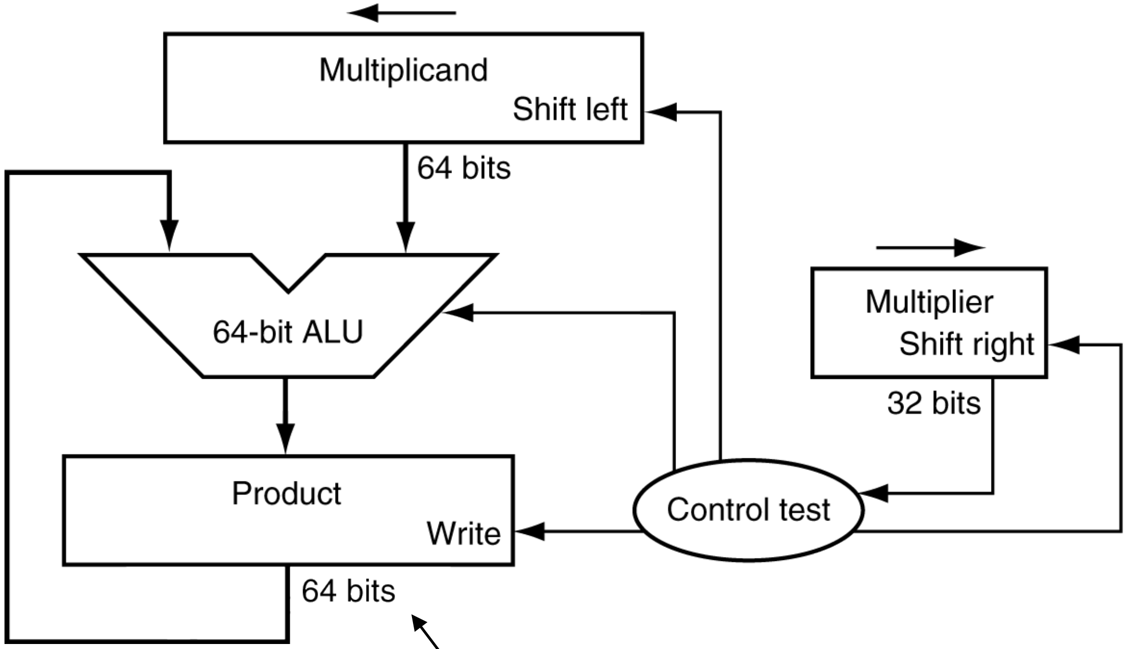
Multiplication – Learn from Basics

- Long-multiplication approach
  - Iterative addition of shifted multiplicands



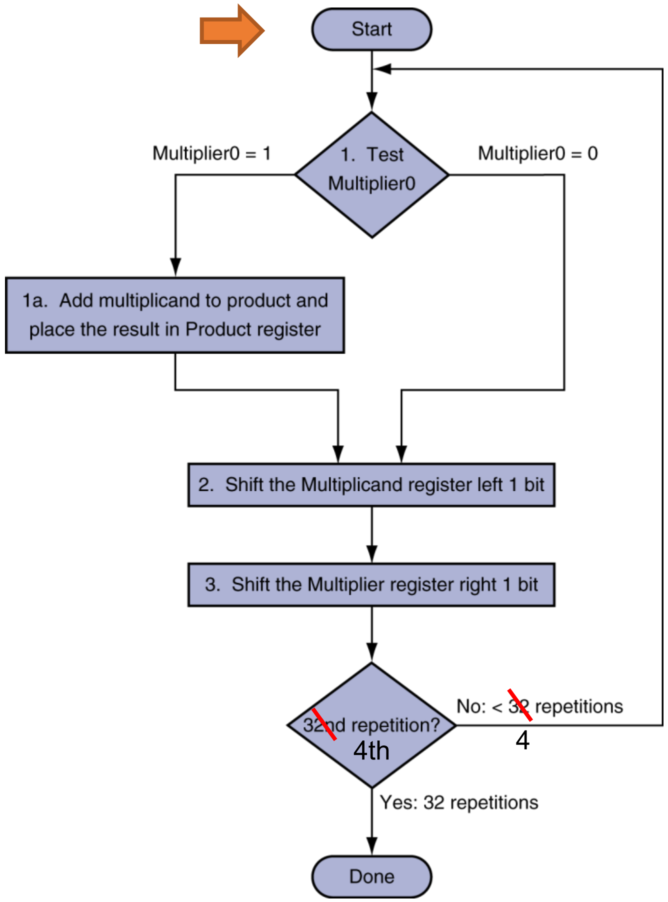
Multiplication Hardware

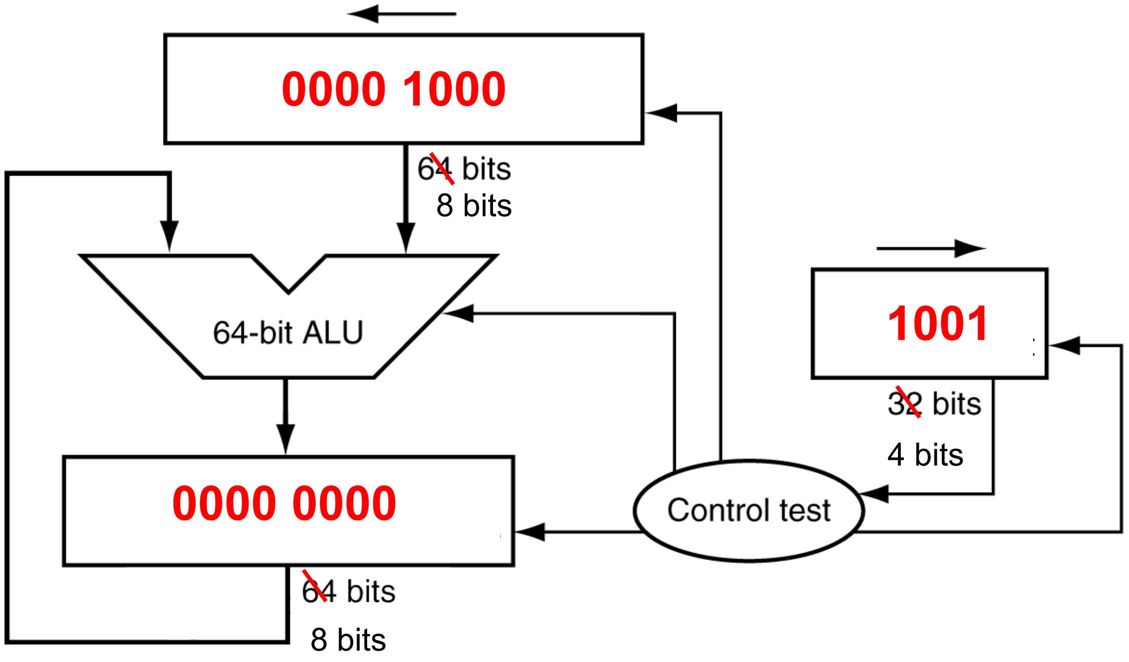

Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done



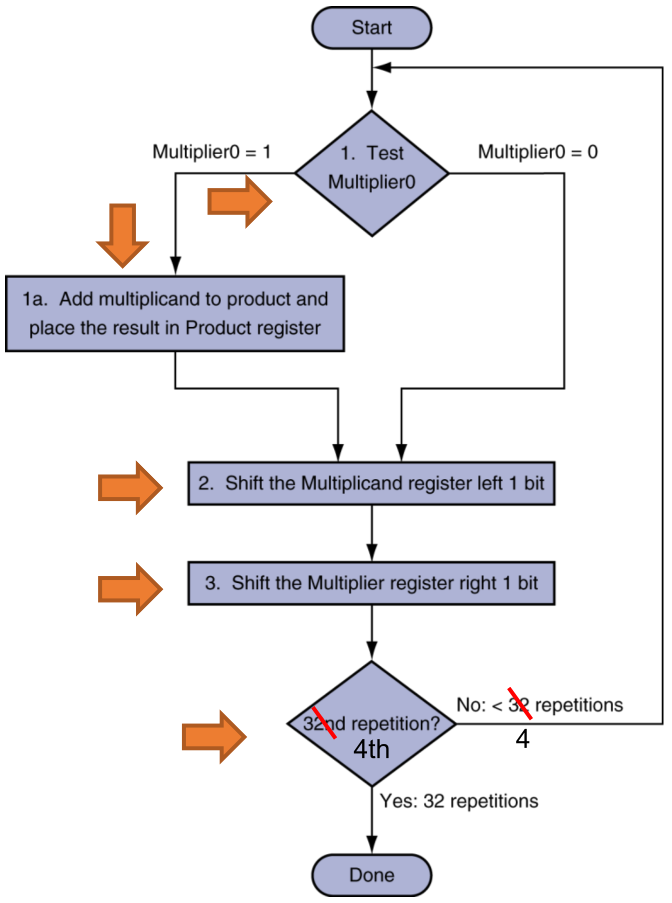
Initially 0

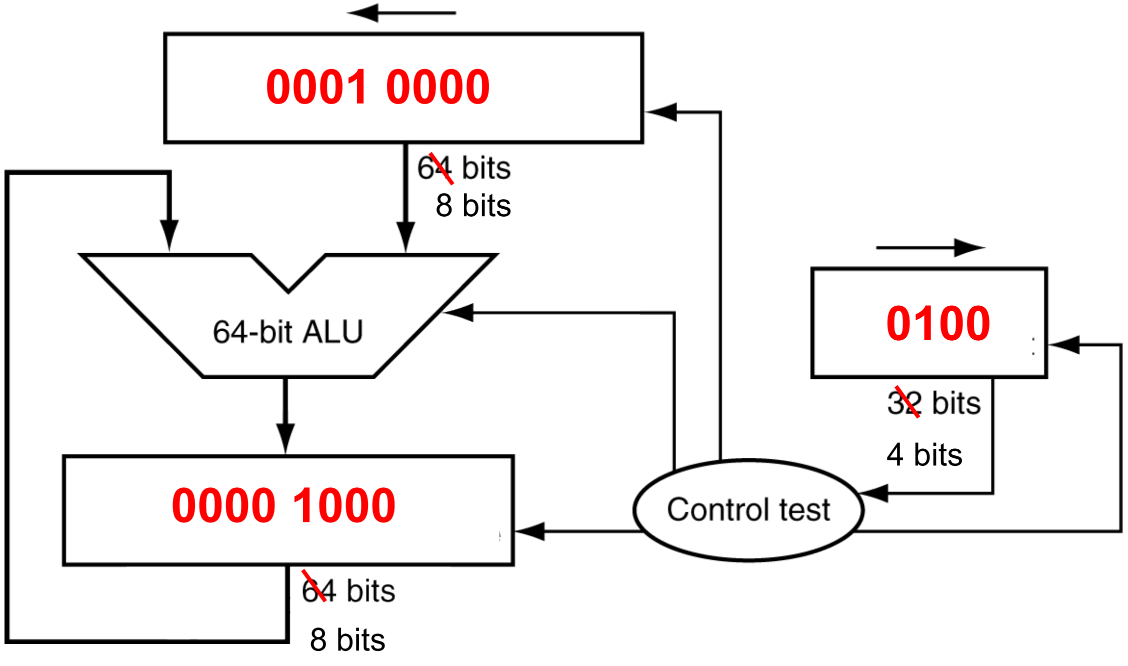
Multiplication Example, 8 x 9 or 1000 x 1001 – Start


Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
~~32nd~~ repetition?
4th
No: < ~~32~~ repetitions
4
Yes: 32 repetitions
Done

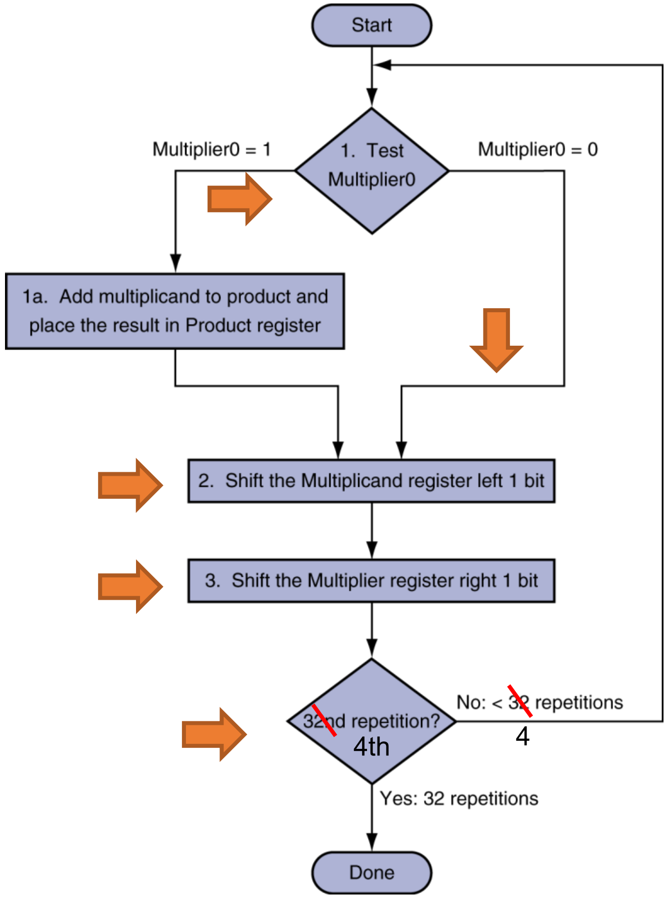


Multiplication Example, 1000 x 1001 – Iteration 1

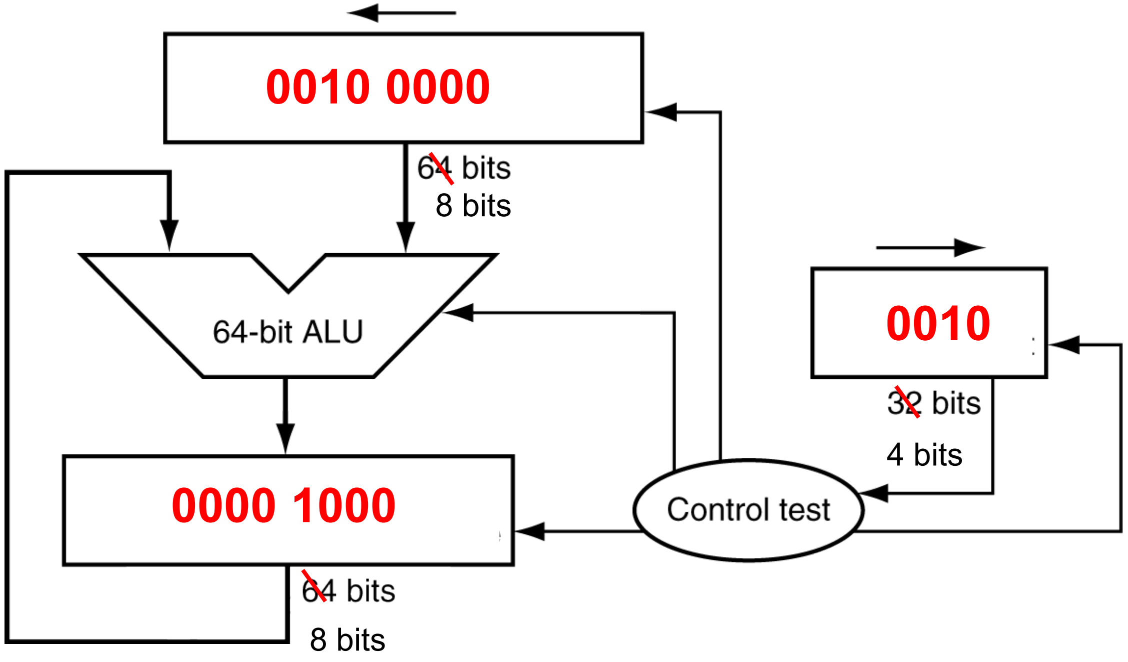

Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
~~32nd~~ repetition?
4th
No: < ~~32~~ repetitions
4
Yes: 32 repetitions
Done



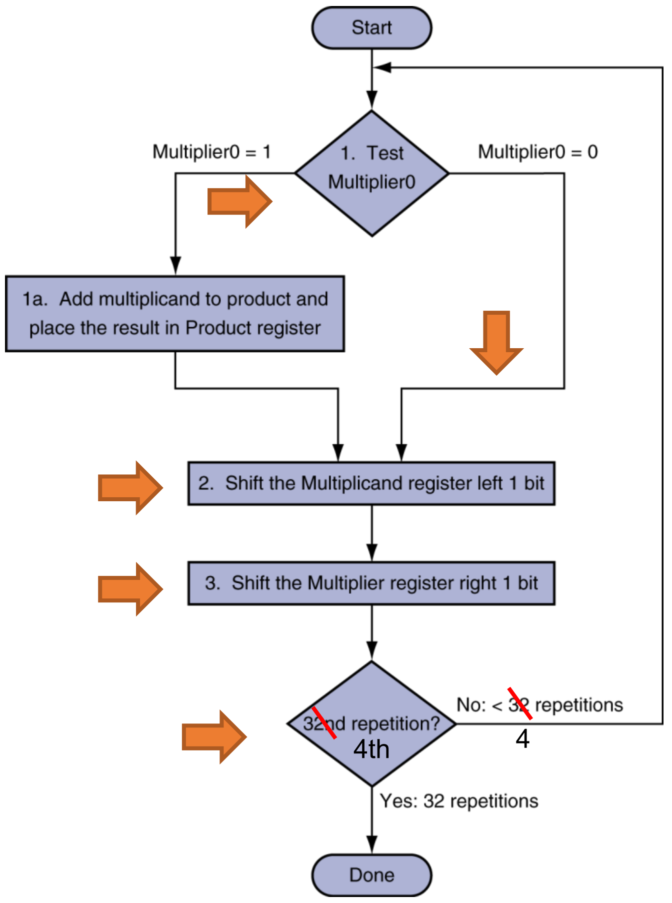
Multiplication Example, 1000 x 1001 – Iteration 2



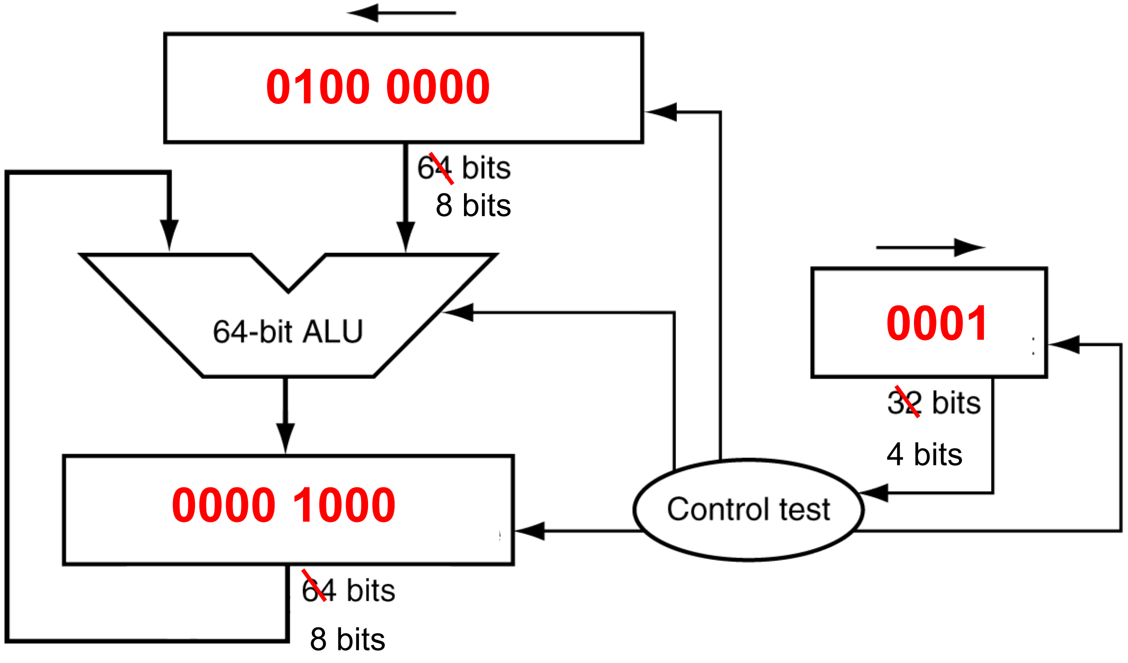
Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
~~32nd~~ repetition?
4th
No: < ~~32~~ repetitions
4
Yes: 32 repetitions
Done



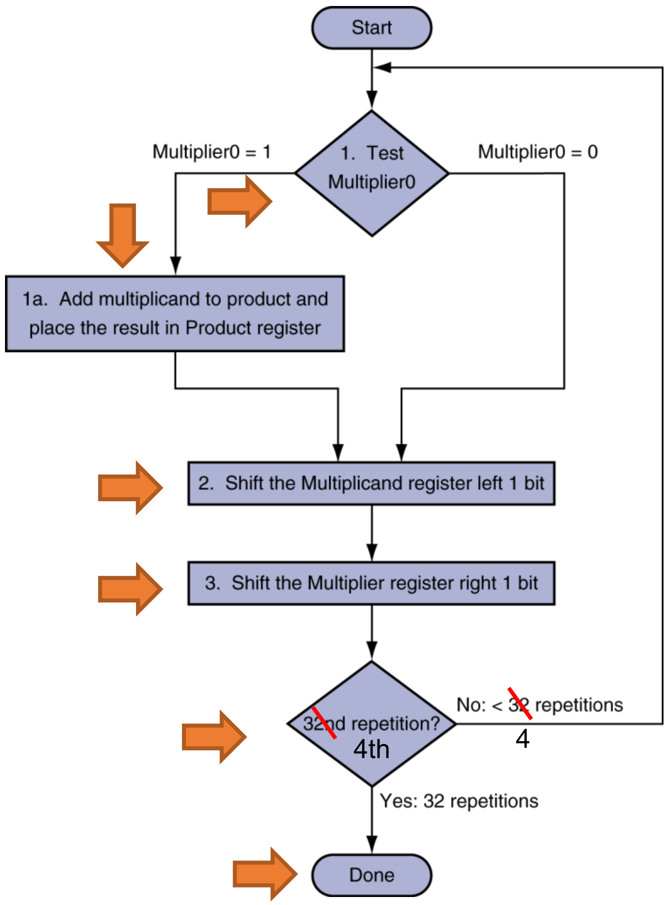
Multiplication Example, 1000 x 1001 – Iteration 3



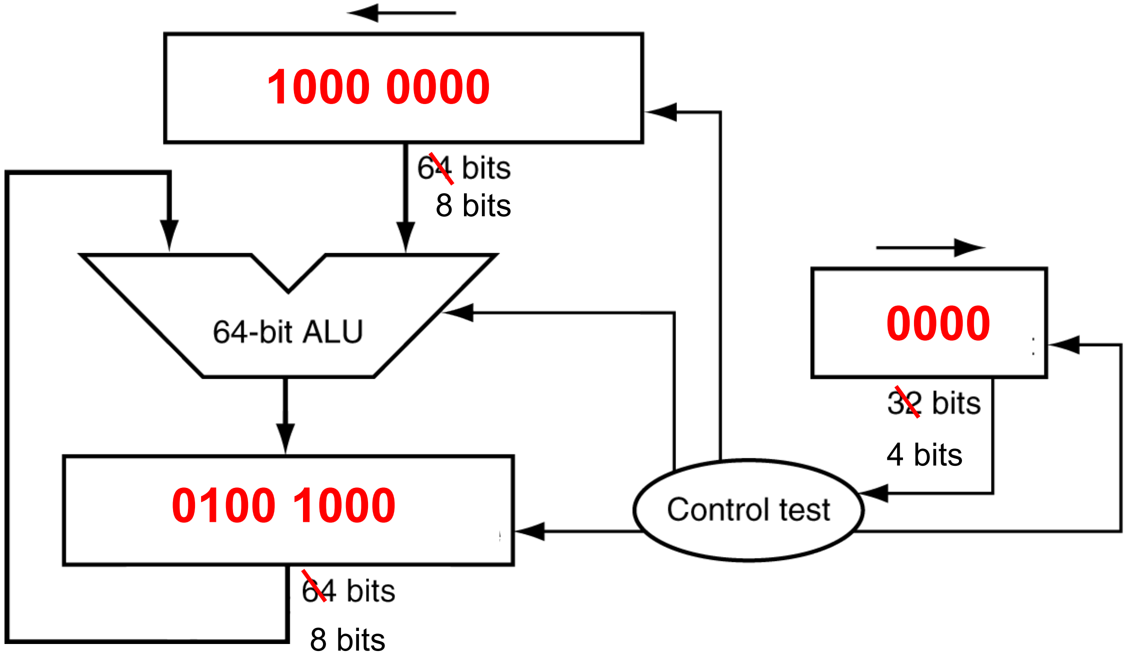
Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
~~32nd~~ repetition?
4th
No: < ~~32~~ repetitions
4
Yes: 32 repetitions
Done



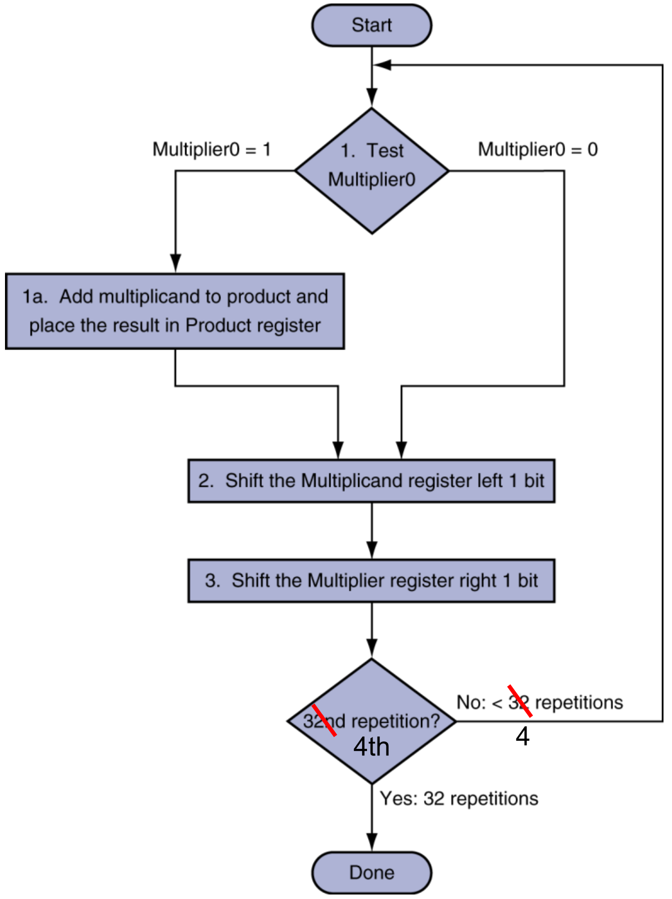
Multiplication Example, 1000 x 1001 – Iteration 4

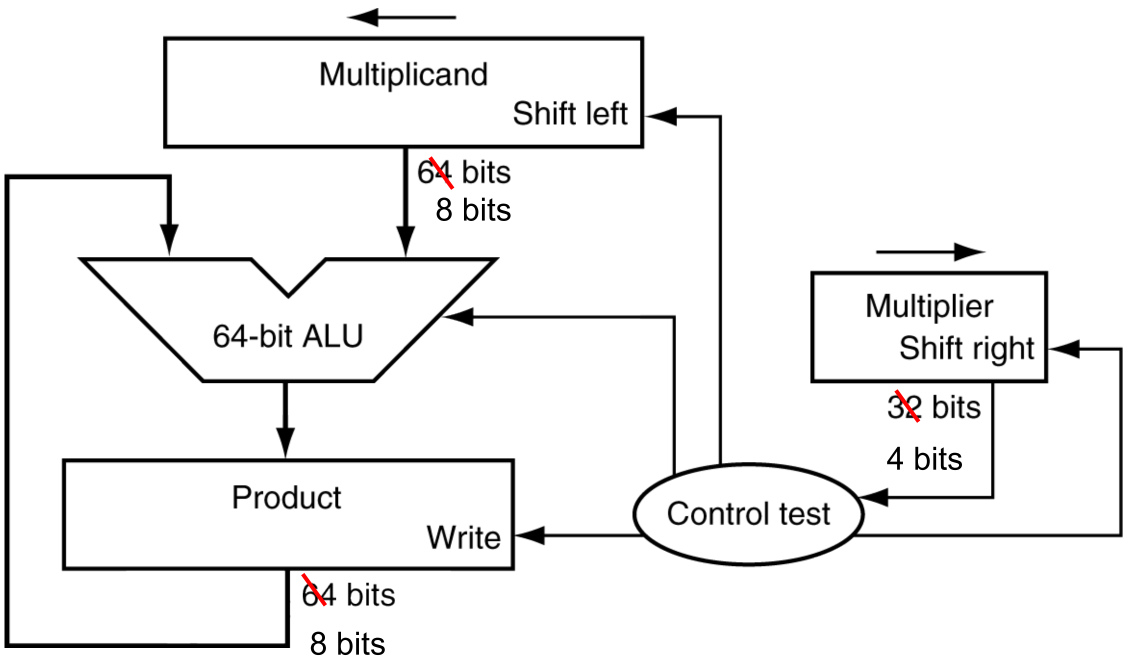


Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
~~32nd~~ repetition?
4th
No: < ~~32~~ repetitions
4
Yes: 32 repetitions
Done



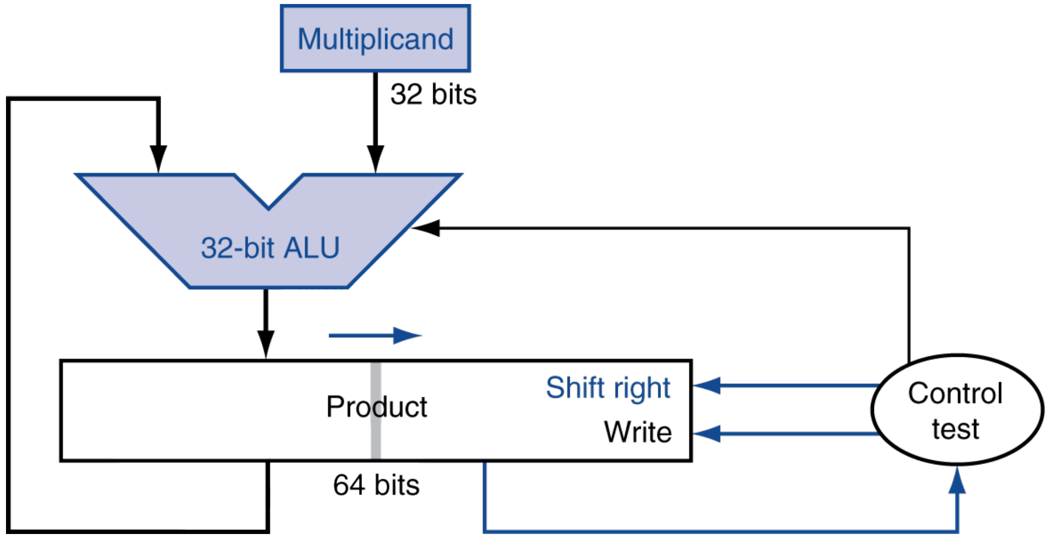
Multiplication Hardware - Summary


Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
~~32nd~~ repetition?
4th
No: < ~~32~~ repetitions
4
Yes: 32 repetitions
Done



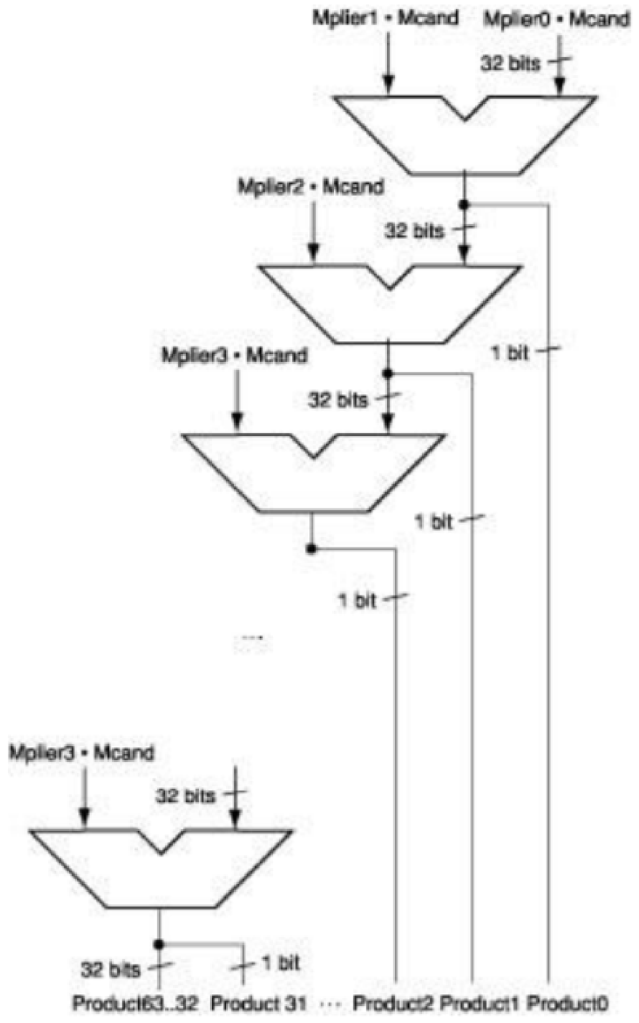
Optimized Multiplier

- Perform steps in parallel: add/shift
  - Partial sum stored in upper 32 bits
  - Multiplier in lower 32 bits
  - Multiplier shifted out bit by bit
  - Partial sum shifts in place of multiplier
  - Do not need to extend multiplicand!
  - Everything happens in place
  - Faster, simpler hardware!



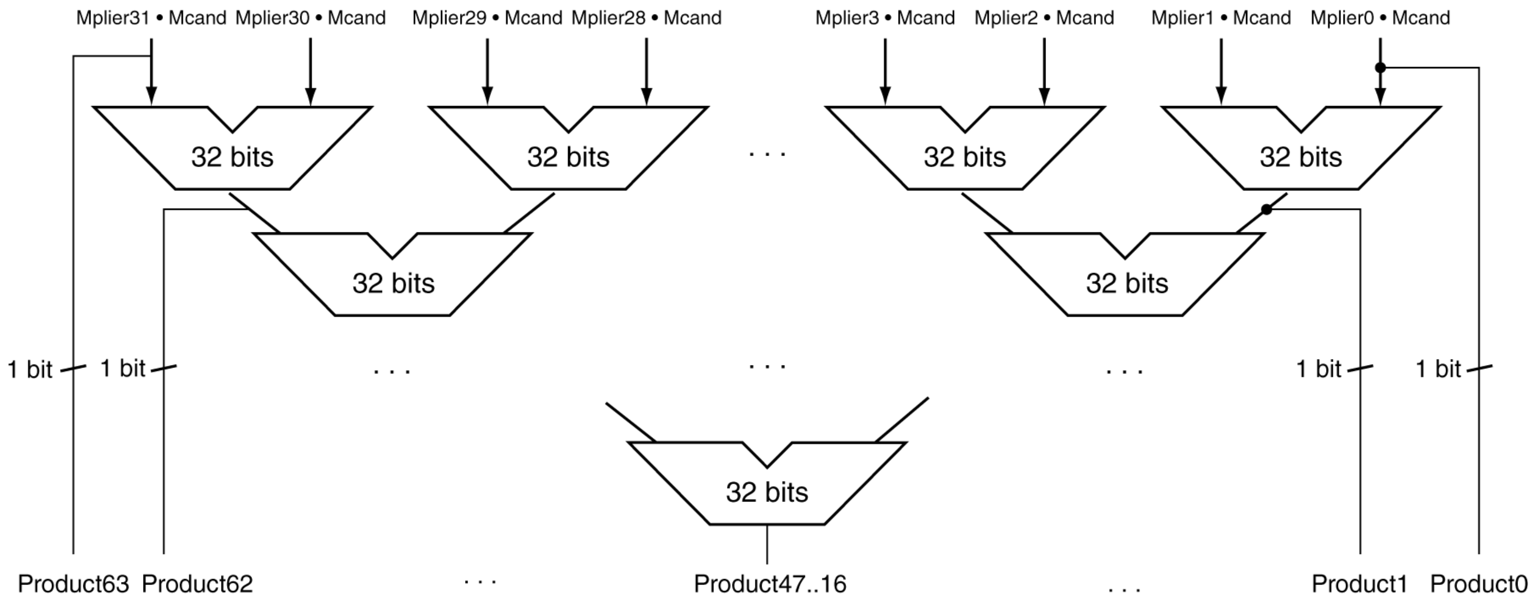
Even Faster Multiplication

- Unroll the addition "loop" and associated shifting
- Use 32 32-bit adders
  - Left input is multiplicand times a bit in multiplier
- Each adder produces 32-bits and a carry-out
  - The least significant bit of each intermediate sum is a bit of the product.
  - The other 31 bits and the carry-out are passed along as the right input to the next adder.



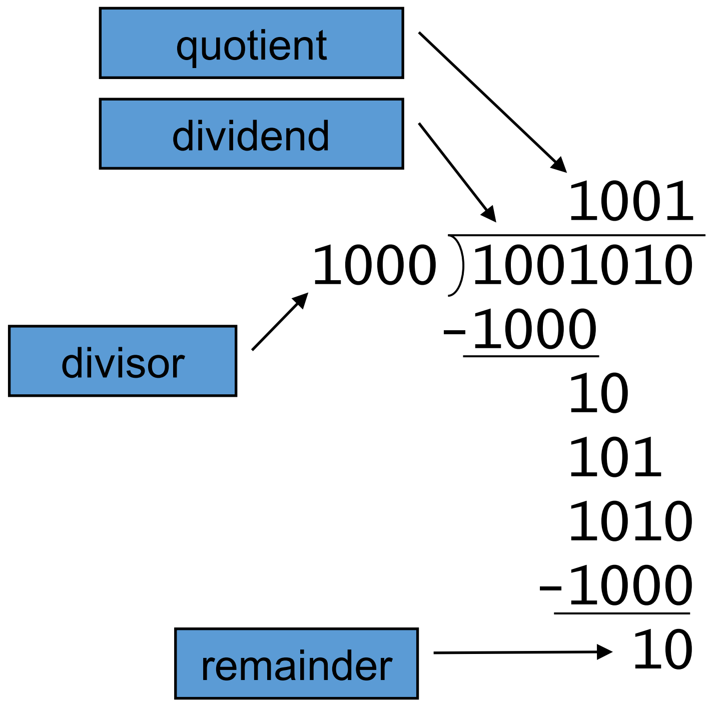
Faster Multiplication

- Uses multiple adders
- Several additions performed in parallel
- Takes $\log_2(32)$ or 5 add cycles



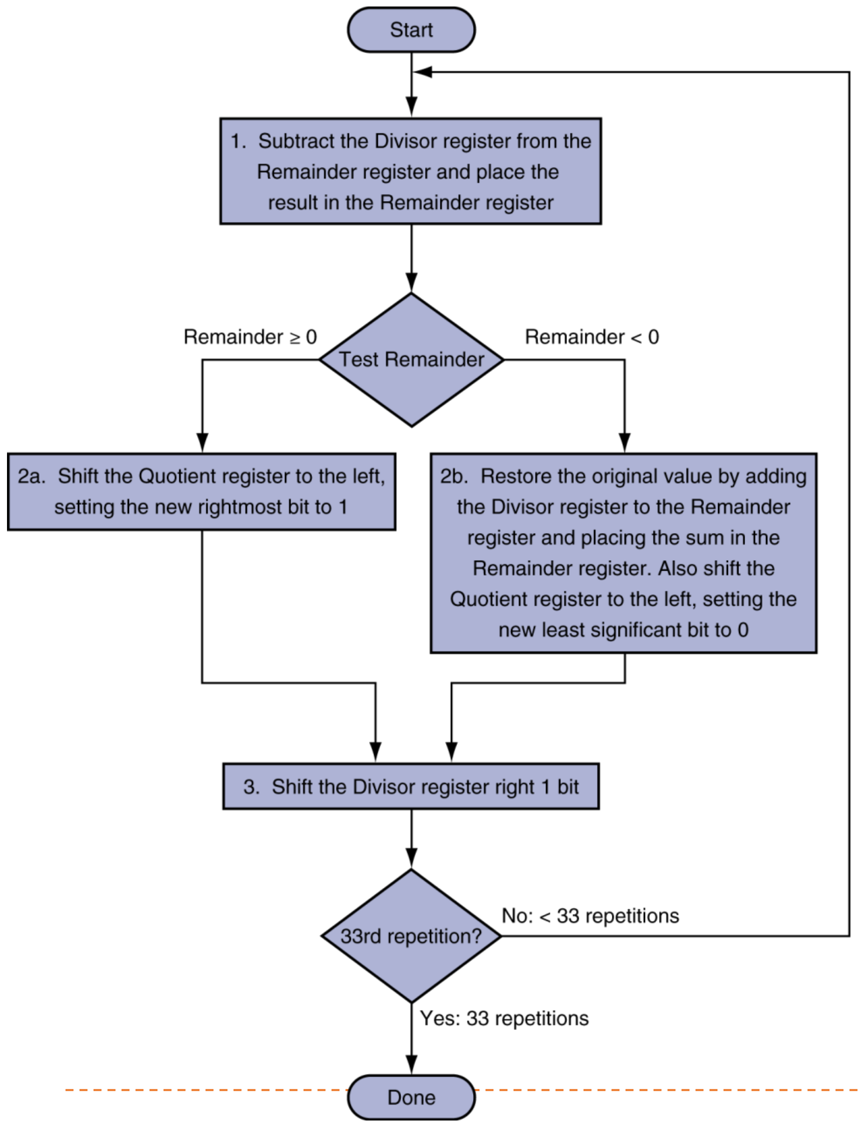
Division

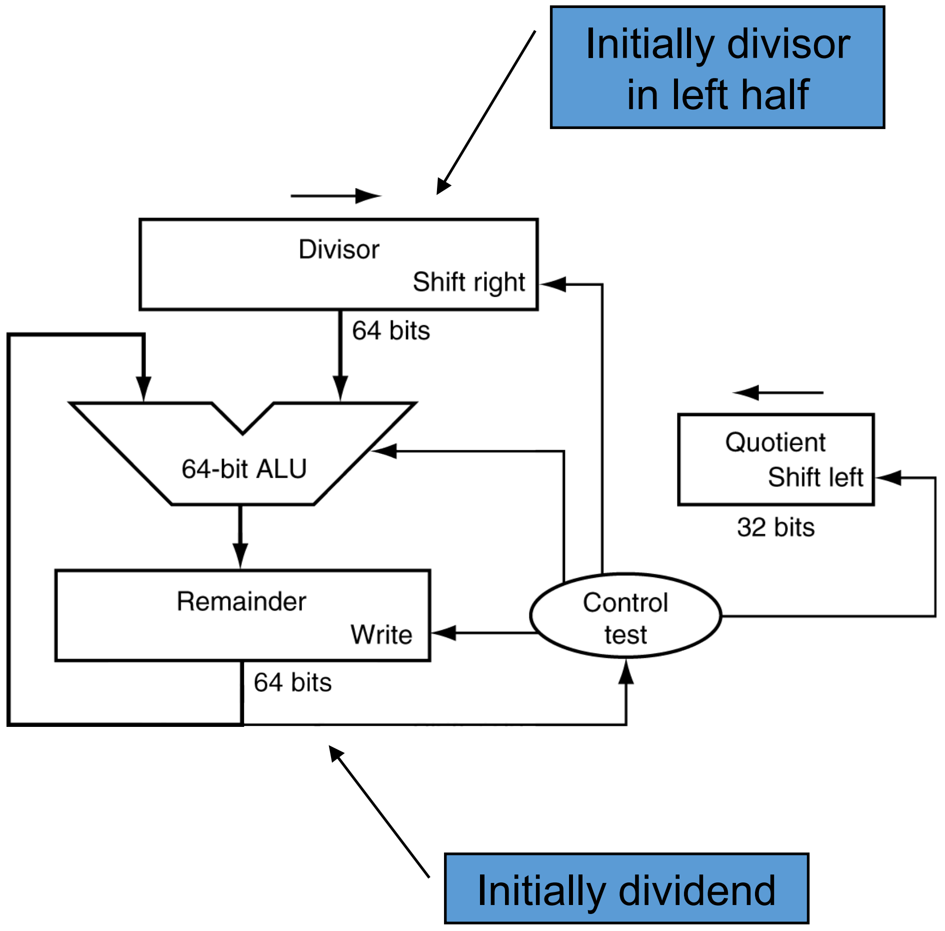
Division Algorithm from Grammar school



- Basic Division algorithm
  - Line up dividend and divisor from left side
  - Let reminder be leading bits of dividend
  - [LOOP]
    - If reminder >= divisor
      - Set 1 bit in quotient, subtract divisor from reminder
    - Otherwise
      - Set 0 bit in quotient,
    - Bring down next dividend bit to reminder
      - If no more bit, done
      - Jump back to [LOOP]

Division Hardware


Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Test Remainder
Remainder ≥ 0
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0
3. Shift the Divisor register right 1 bit
33rd repetition?
No: < 33 repetitions
Yes: 33 repetitions
Done



Example – 4-bit Division

- Dividend = 13 (1101_2)
- Divisor = 2 (0010_2)

```
                         ←quotient
dividend→
                   ________.
             10 ) 1101←reminder
                  1
```

Division Example

- Dividend = 13 (1101_2)
- Divisor = 2 (0010_2)

```
                              ←quotient
Dividend→            0        .
                   ____________
              10 ) 1101
                               ←reminder
                     1
                     11
```

Division Example

- Dividend = 13 (1101_2)
- Divisor = 2 (0010_2)

```
                   ←quotient
Dividend→      01   .
             _________
          10 ) 1101
               1
               11  ←reminder
              -10
              ___
                1
```

Division Example

- Dividend = 13 (1101_2)
- Divisor = 2 (0010_2)

```
                        ←quotient
Dividend→        01        .
               ___________________
           10 ) 1101
                1
                11      ←reminder
               -10
               ---
                 10
```

Division Example

- Dividend = 13 (1101_2)
- Divisor = 2 (0010_2)

```
                      ←quotient
Dividend→         011     .
                ____________
           10 ) 1101
                1
                11
               -10
               ---
                 10 ←reminder
                -10
                ---
                  0
```

Division Example

- Dividend = 13 (1101_2)
- Divisor = 2 (0010_2)

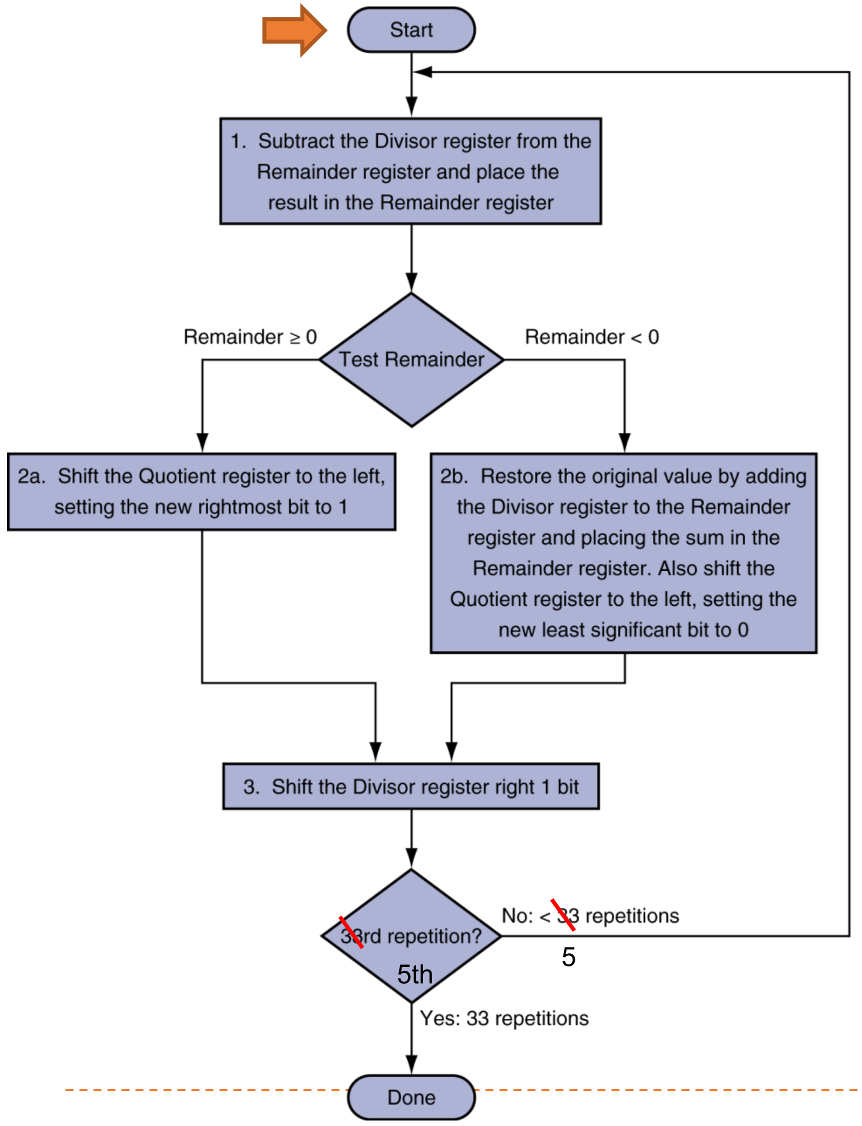
```
                   ←quotient
Dividend→      011     .
          10 ) 1101
               1
               11
              -10
                10 ←reminder
               -10
                 01
```

Division Example

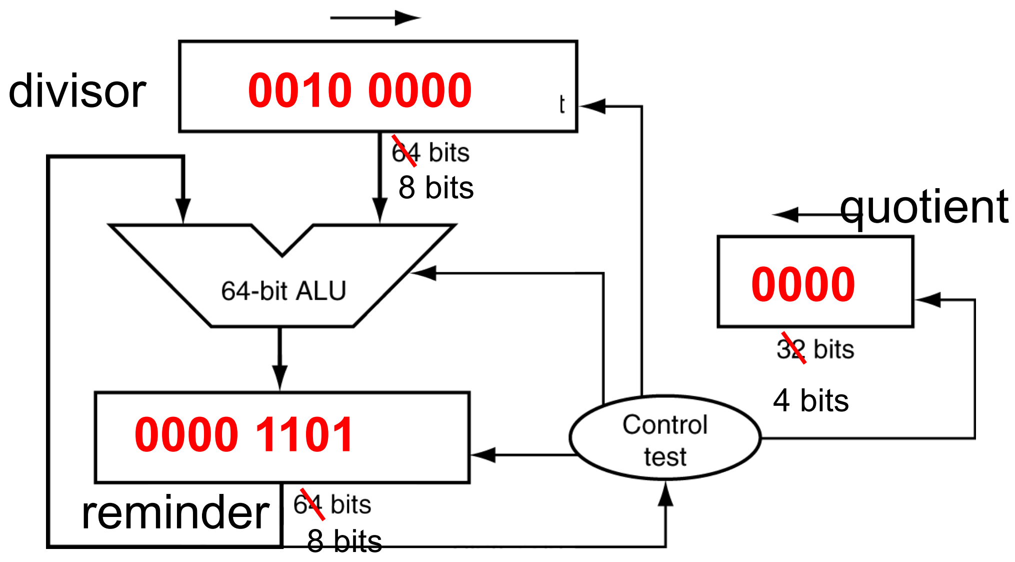
- Dividend = 13 (1101_2)
- Divisor = 2 (0010_2)

```
                       ←quotient
Dividend→        0110   .
             ___________________
          10 ) 1101
               1
               11
              -10
              ---
                10 ←reminder
               -10
               ---
                 01
```

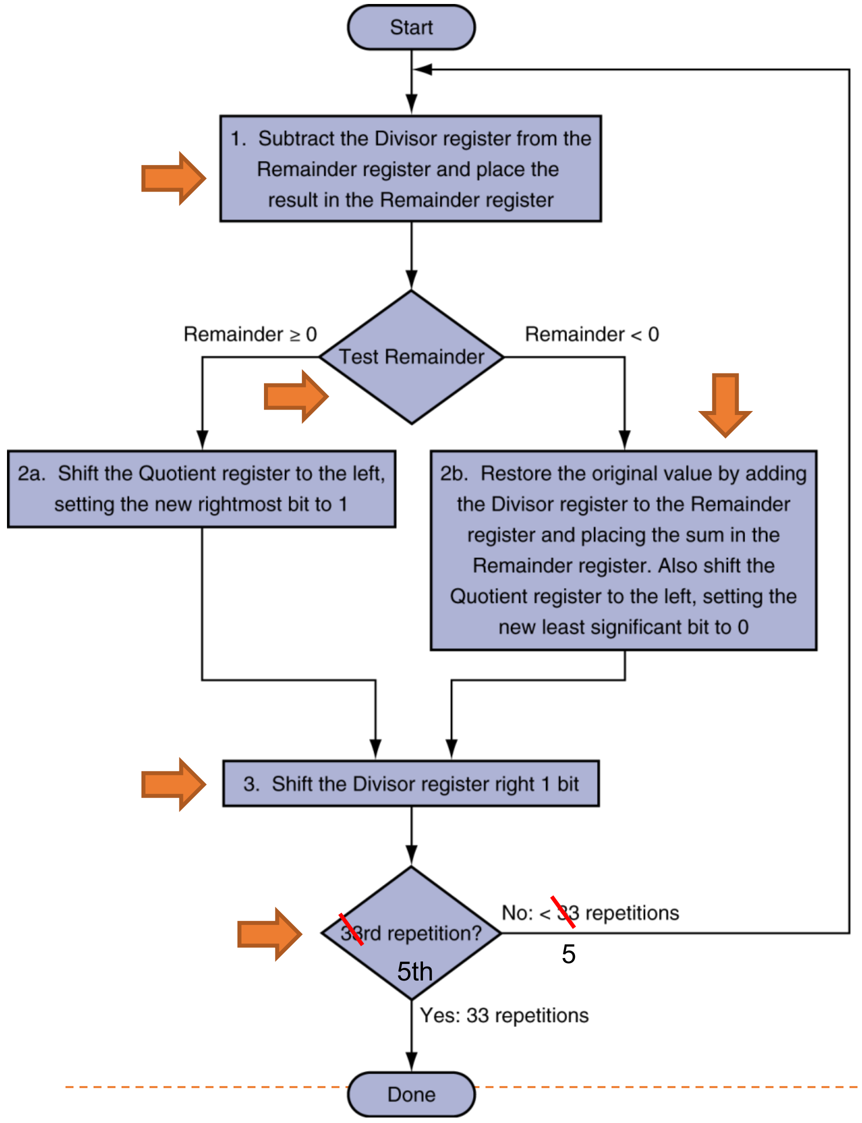
Division Example $1101_2 / 0010_2$ – start

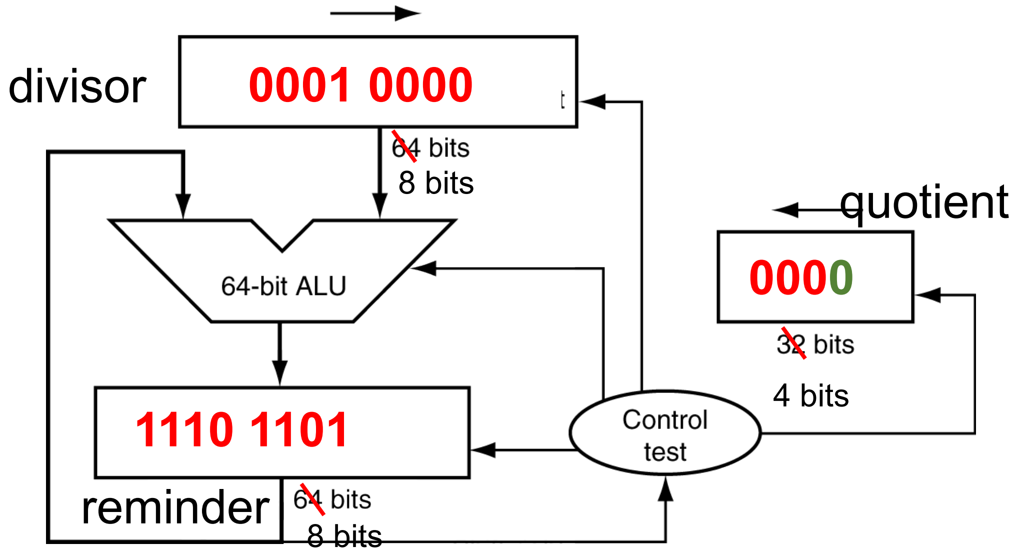


Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Test Remainder
Remainder ≥ 0
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0
3. Shift the Divisor register right 1 bit
~~33~~rd repetition?
5th
No: < ~~33~~ repetitions
5
Yes: 33 repetitions
Done

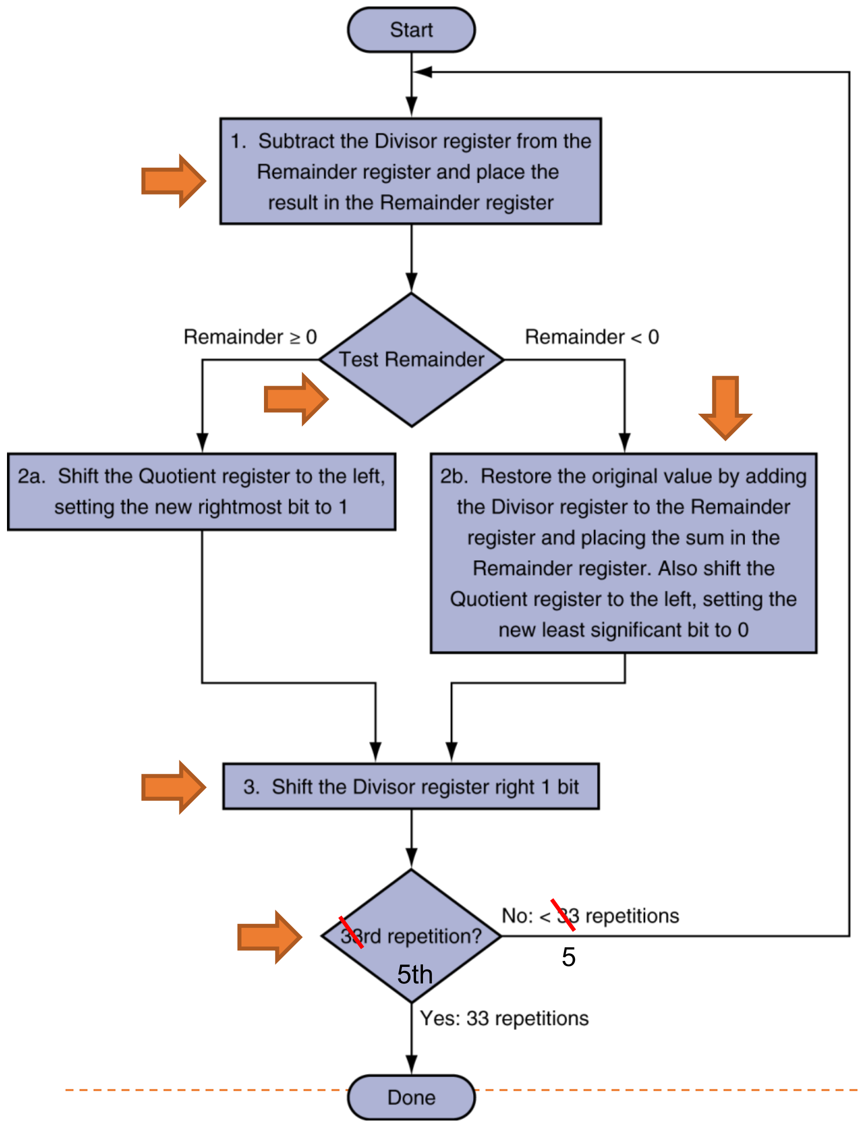


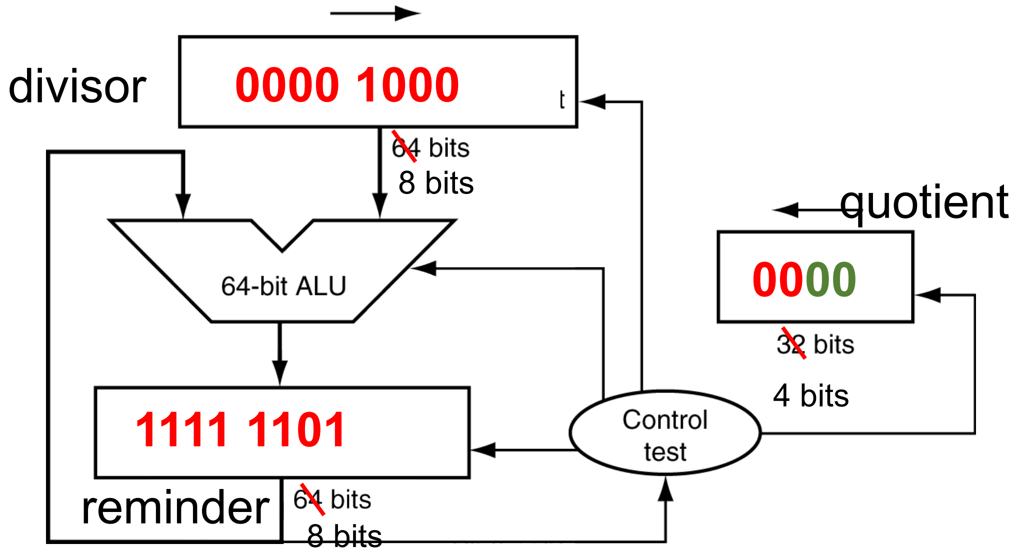
Division Example – Iteration 1


Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Test Remainder
Remainder ≥ 0
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0
3. Shift the Divisor register right 1 bit
~~33~~rd repetition?
5th
No: < ~~33~~ repetitions
5
Yes: 33 repetitions
Done

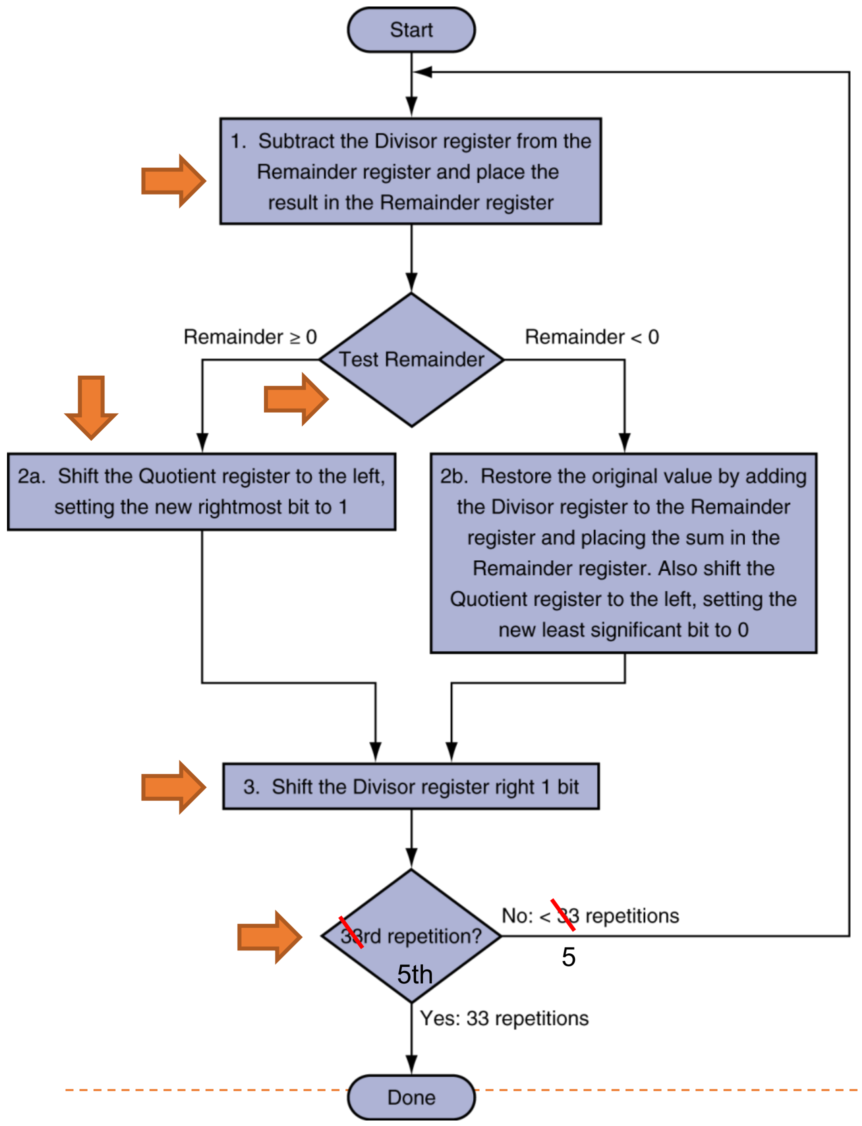


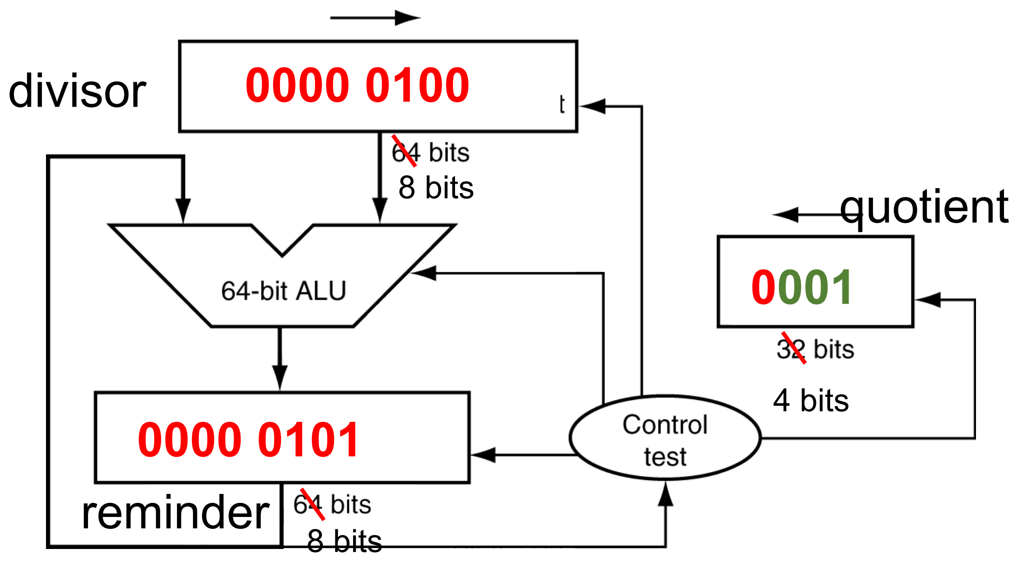
Division Example – Iteration 2


Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Test Remainder
Remainder ≥ 0
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0
3. Shift the Divisor register right 1 bit
~~33~~rd repetition?
5th
No: < ~~33~~ repetitions
5
Yes: 33 repetitions
Done

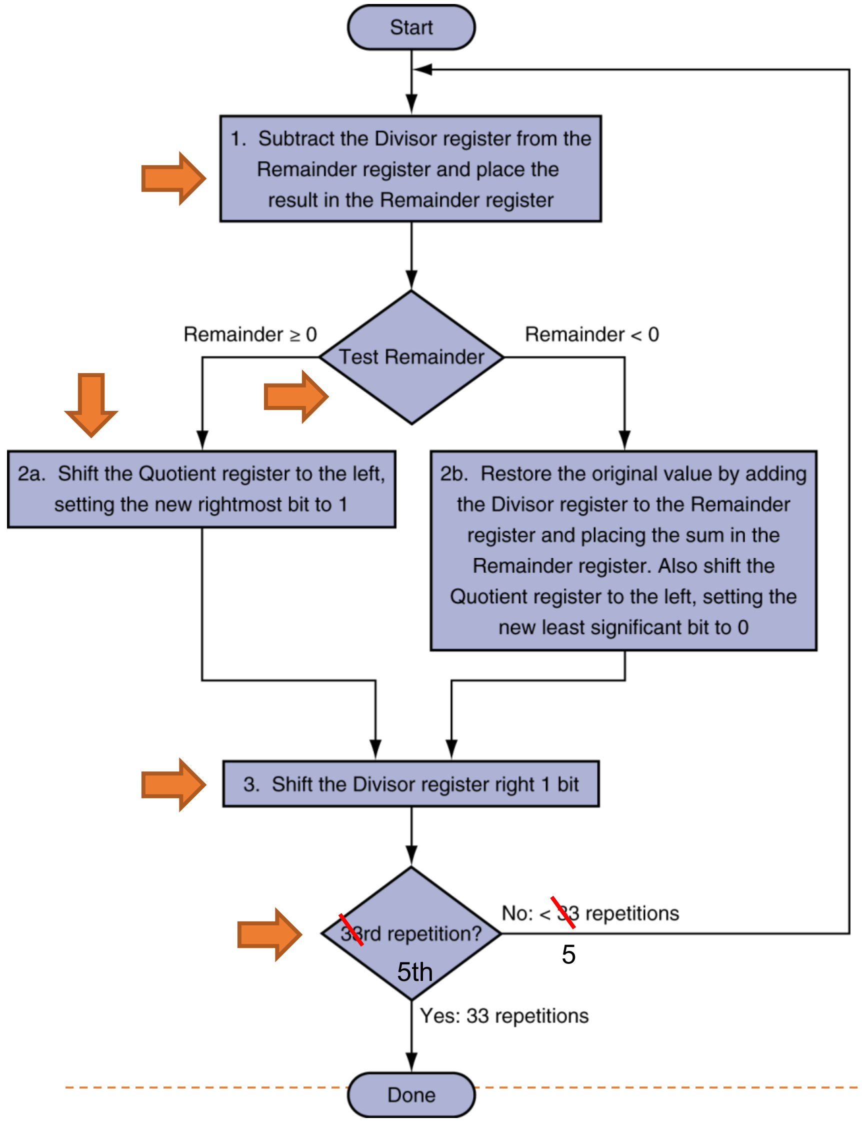


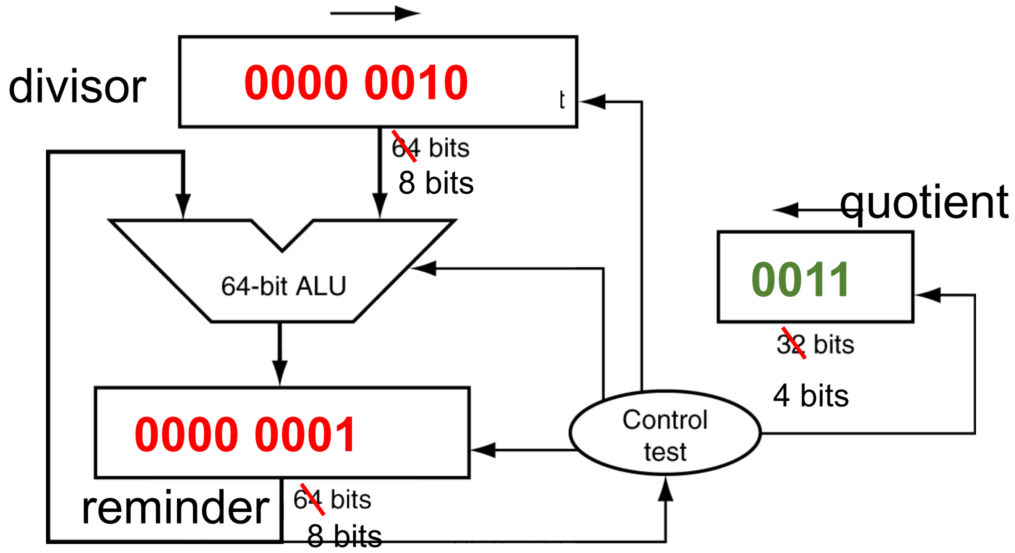
Division Example – Iteration 3


Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Test Remainder
Remainder ≥ 0
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0
3. Shift the Divisor register right 1 bit
~~33~~rd repetition?
5th
No: < ~~33~~ repetitions
5
Yes: 33 repetitions
Done

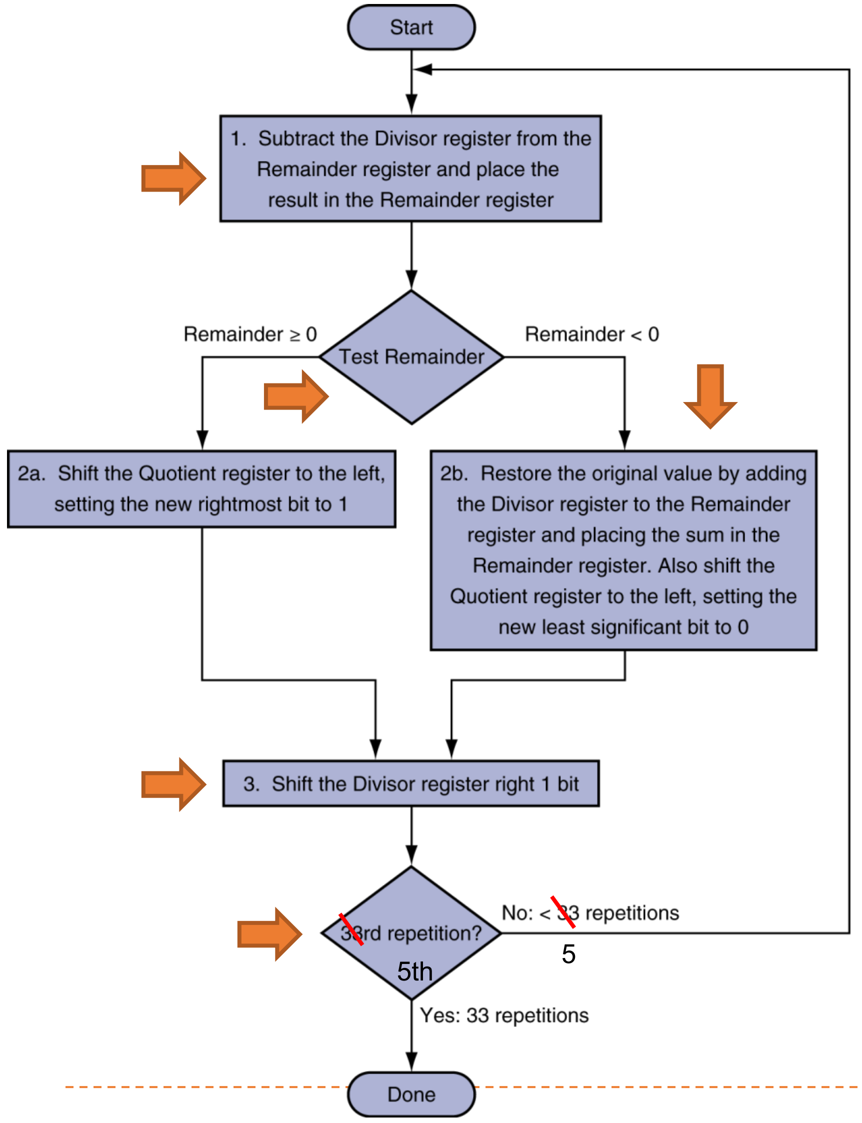


Division Example – Iteration 4

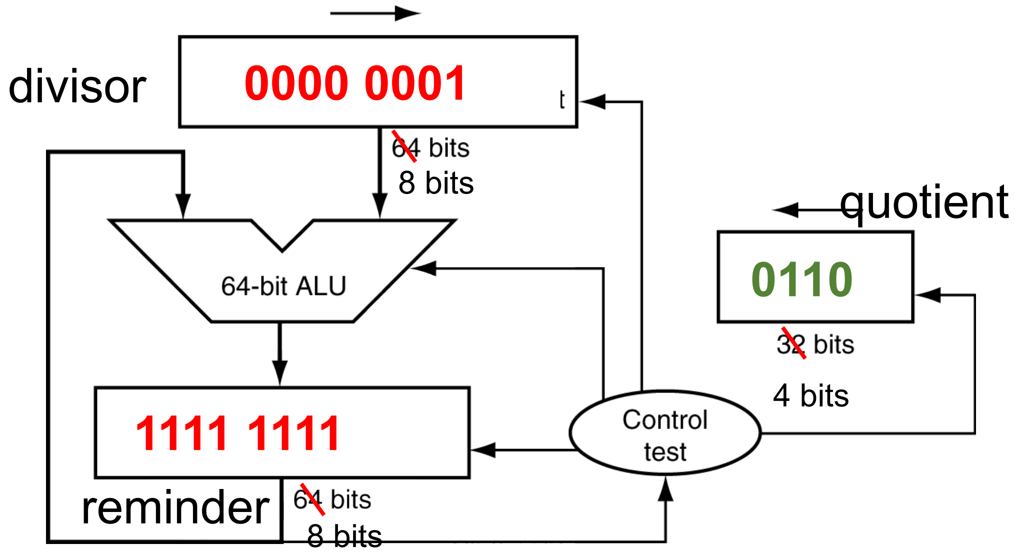

Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Remainder ≥ 0
Test Remainder
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0
3. Shift the Divisor register right 1 bit
~~33~~rd repetition?
5th
No: < ~~33~~ repetitions
5
Yes: 33 repetitions
Done



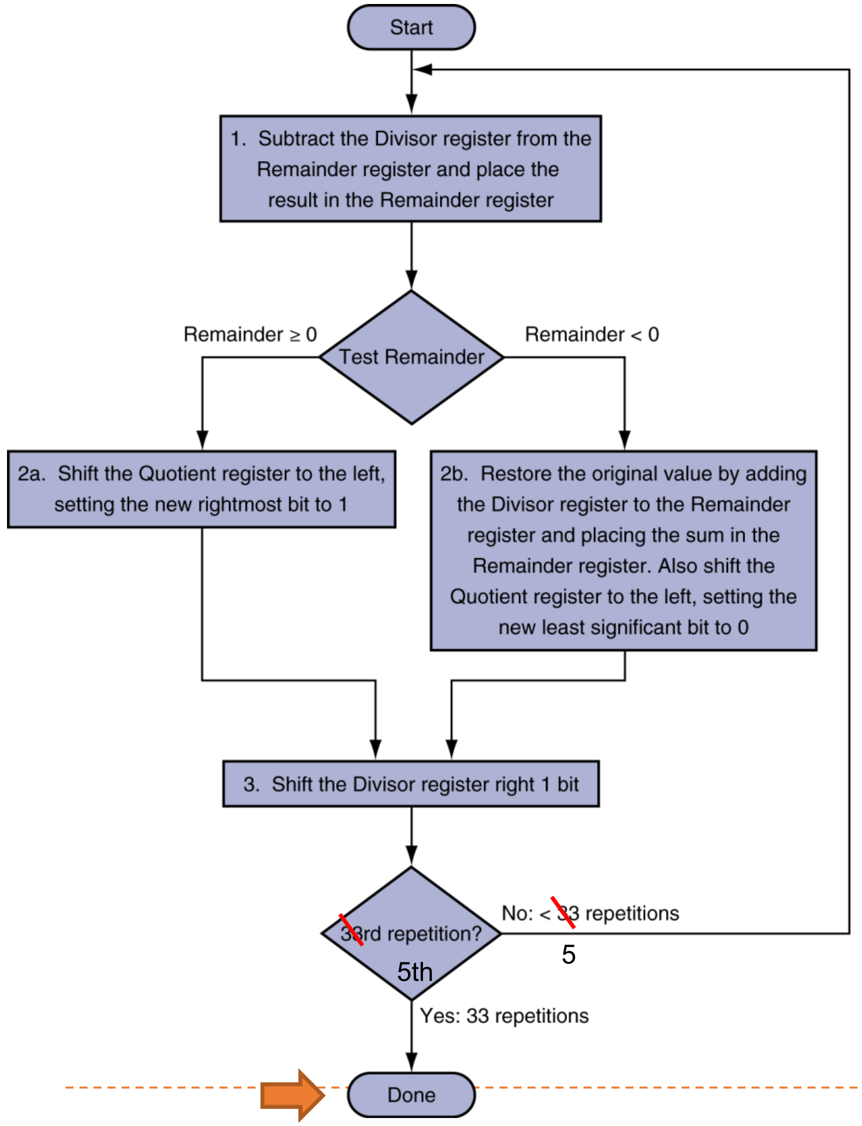
Division Example – Iteration 5



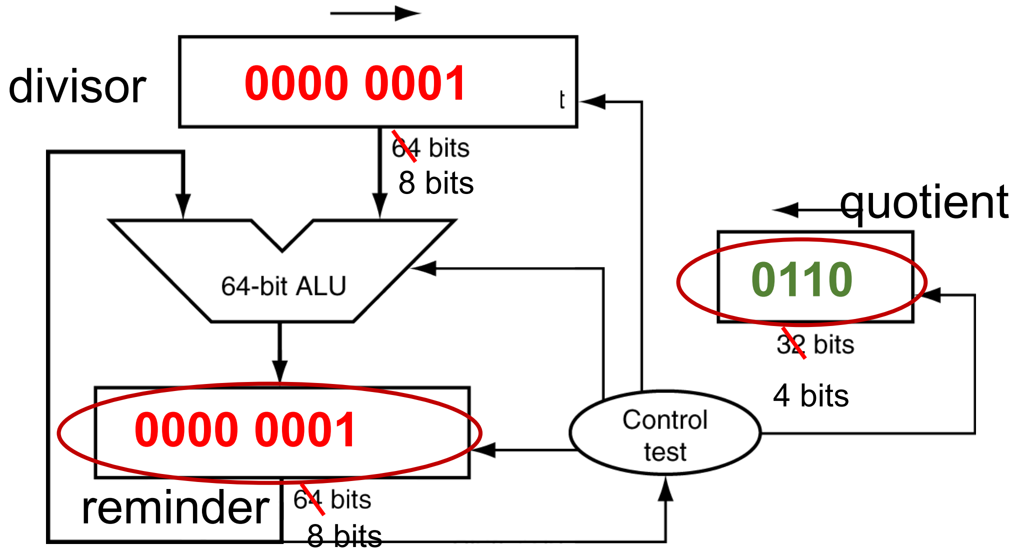
Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Test Remainder
Remainder ≥ 0
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0
3. Shift the Divisor register right 1 bit
~~33~~rd repetition?
5th
No: < ~~33~~ repetitions
5
Yes: 33 repetitions
Done



Division Example – Done

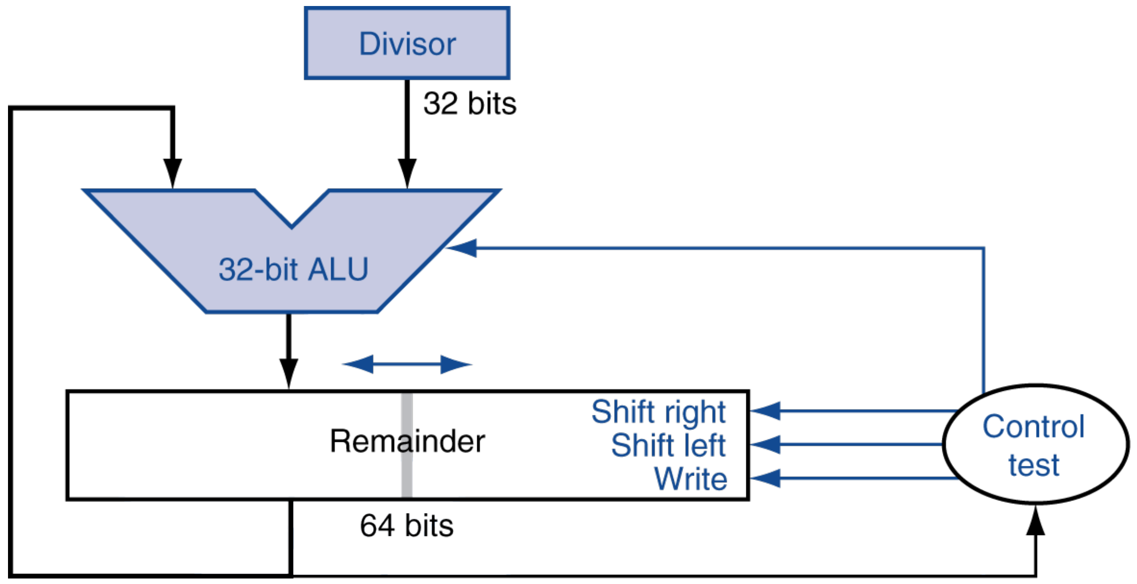


Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Test Remainder
Remainder ≥ 0
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0
3. Shift the Divisor register right 1 bit
~~33~~rd repetition?
5th
No: < ~~33~~ repetitions
5
Yes: 33 repetitions
Done



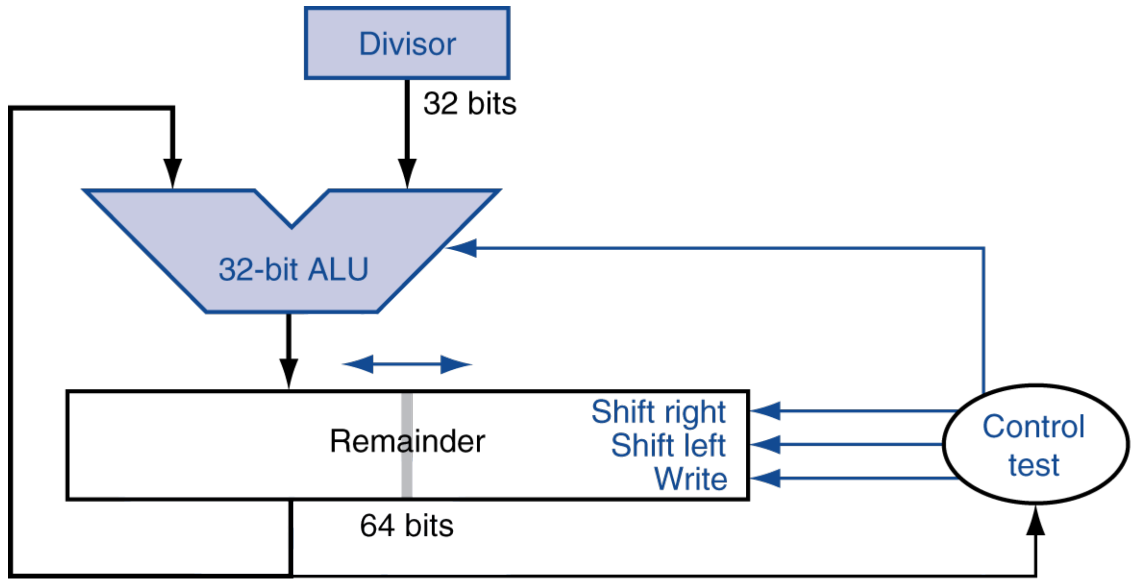
Q: Can we do better?

An Improved Version



- Initially, dividend is put into lower 32bit of Reminder
- For each iteration
  - Subtraction and possible restore to Remainder
  - Reminder shift 1 bit to the left
  - Quotient bit is stored to right side of Remainder

- At the end, top 32bit is remainder, lower 32bit is quotient



- Looks a lot like a multiplier!
  - Same hardware can be used for both

Faster Division

- Can’t use parallel hardware as in multiplication
  - Subtraction is conditional on sign of remainder

- Faster dividers (e.g. SRT division) generate multiple quotient bits per step
  - Still require multiple steps

RISC-V M Extension

1. `mul` performs a 32 x 32 register multiply and stores the lower 32 bits of the 64-bit result in a register.
2. `mulh` and `mulhu` perform signed and unsigned register multiply, respectively, and store the upper 32 bits of the 64-bit result in a register.
3. `mulhsu` performs a signed x unsigned register multiply and stores the upper 32 bits of the 64-bit result in a register.
4. `div` and `divu` perform signed and unsigned 32/32 register division
5. `Rem` and `remu` perform signed and unsigned 32%32 remainder

Reference Readings

- Patterson, "Computer Organization And Design"
  - A.2, A.3, 3.3, 3.4
- Tanenbaum, "Structured Computer Organization"
  - Sec 3.2
- Bryant, "Computer Systems"
  - Sec 4.2.2 – 4.2.4