CENG 3420
Computer Organization & Design
Lecture 06: ALU

Textbook: Chapter 3.2-3.4 & A.5-A.6

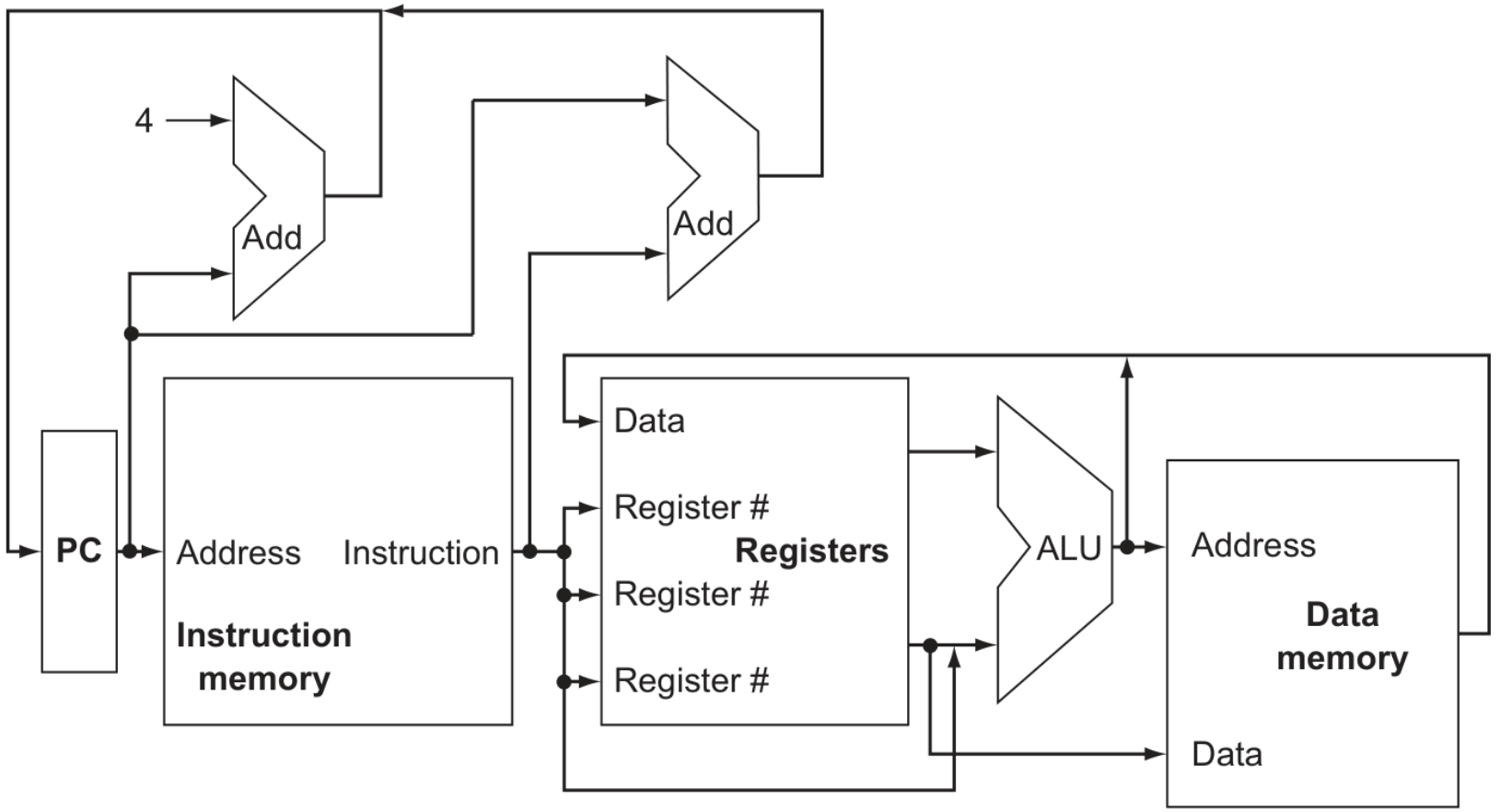
Zhengrong Wang

CSE Department, CUHK

zhengrongwang@cuhk.edu.hk

Overview

Processor Overview

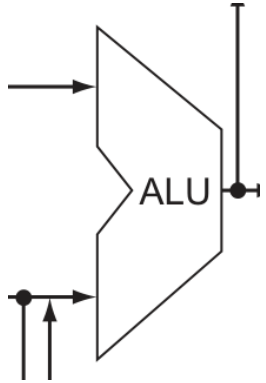


Arithmetic-Logic Unit (ALU)

- We have seen the abstraction.
  - Instruction set architecture (ISA).
  - Machine code and instructions.

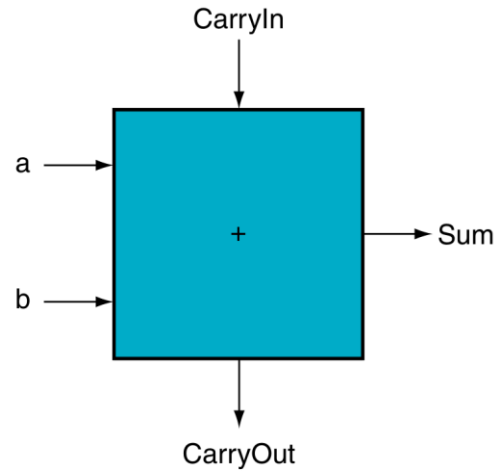
- Next: we will design the arithmetic-logic unit (ALU).
  - All instructions need to use the ALU.
  - E.g., add two numbers, calculate the address.

- Let's start with addition.



Addition Unit

1-bit Full Adder



Inputs			Outputs		Comments
a	b	CarryIn	CarryOut	Sum	
0	0	0	0	0	$0 + 0 + 0 = 00_{two}$
0	0	1	0	1	$0 + 0 + 1 = 01_{two}$
0	1	0	0	1	$0 + 1 + 0 = 01_{two}$
0	1	1	1	0	$0 + 1 + 1 = 10_{two}$
1	0	0	0	1	$1 + 0 + 0 = 01_{two}$
1	0	1	1	0	$1 + 0 + 1 = 10_{two}$
1	1	0	1	0	$1 + 1 + 0 = 10_{two}$
1	1	1	1	1	$1 + 1 + 1 = 11_{two}$

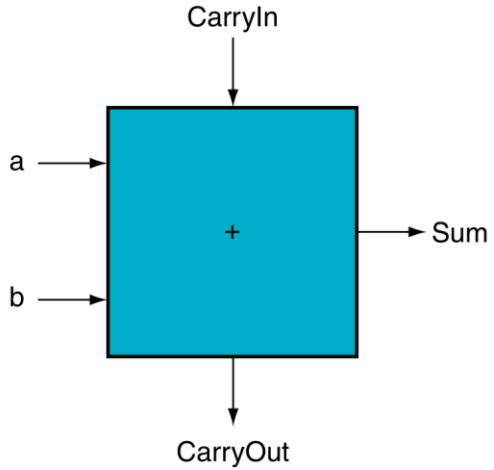
FIGURE A.5.3 Input and output specification for a 1-bit adder.

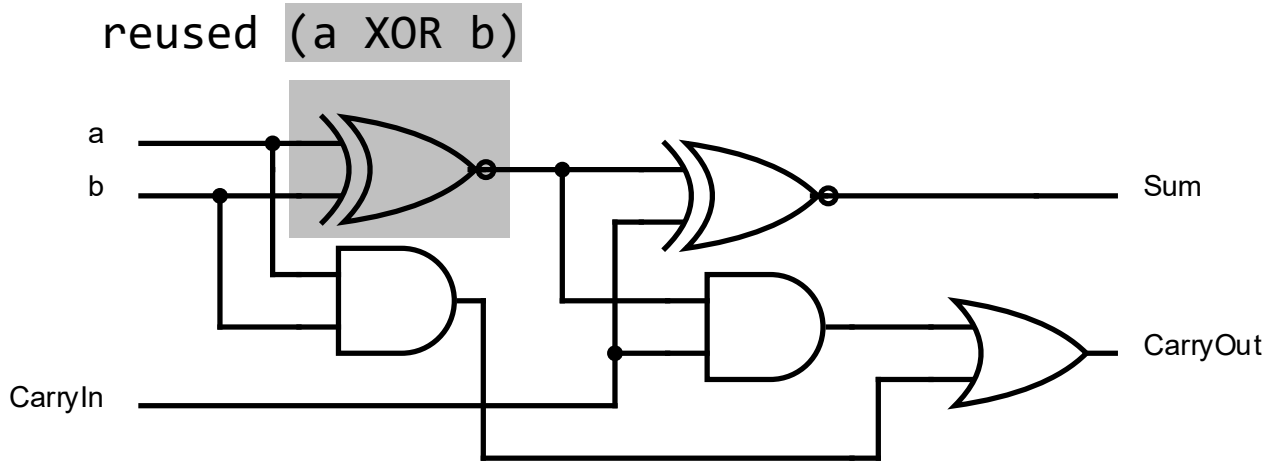
- `Sum = a XOR b XOR CarryIn`
- `CarryOut = (a AND b) OR (a AND CarryIn) OR (b AND CarryIn)`

- How to implement it in logical gates?

1-bit Full Adder in Logical Gates

- `Sum = (a XOR b) XOR CarryIn`
- `CarryOut = (a AND b) OR (a AND CarryIn) OR (b AND CarryIn)`
- `CarryOut = (a AND b) OR ((a OR b) AND CarryIn)`
- We can replace OR with XOR because `(a AND b)` already handles the case a=b=1
- `CarryOut = (a AND b) OR ((a XOR b) AND CarryIn)`
- So that we can reuse `(a XOR b)` from Sum.

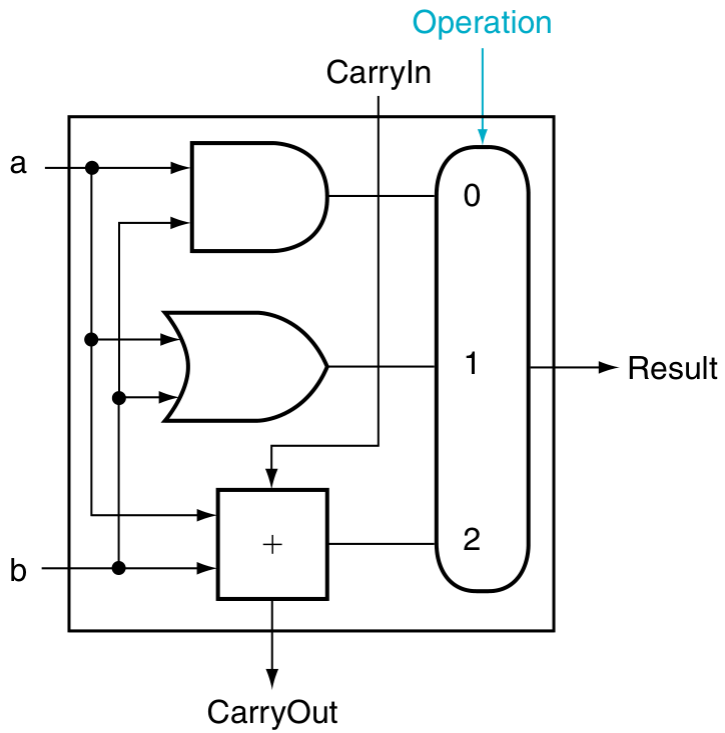




Support Logical Operation

- Add AND/OR and select the output based on instruction opcode/funct.

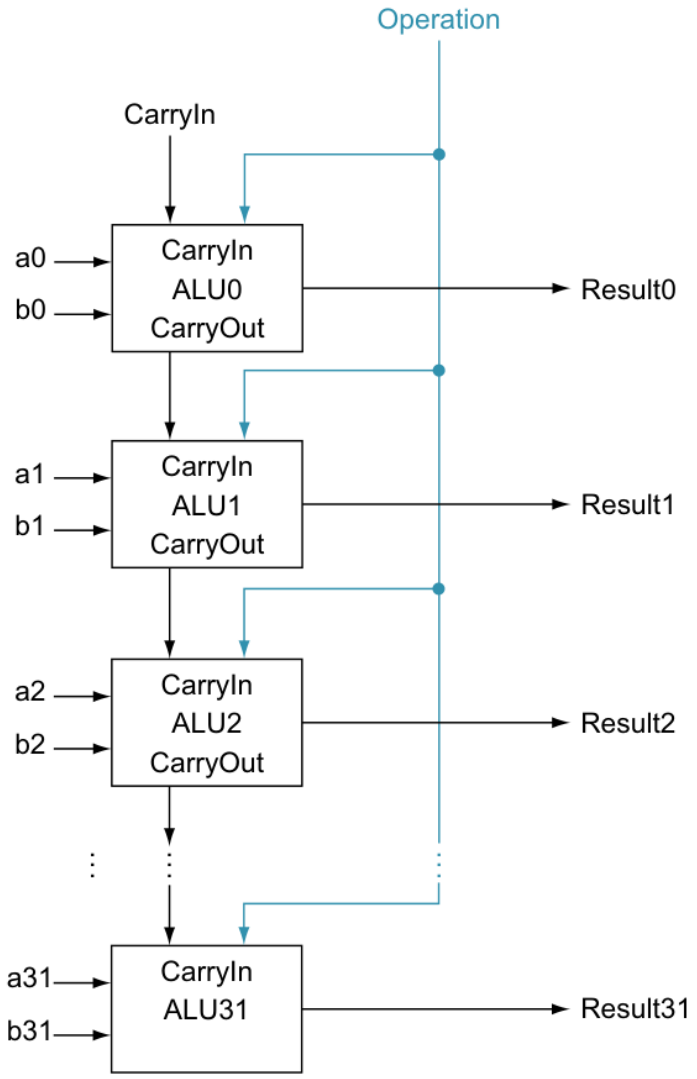
Operation	Result
00	a & b
01	a | b
10	a + b



- This is a 1-bit ALU with AND/OR/ADD operation.
- How to support 32-bit operand?

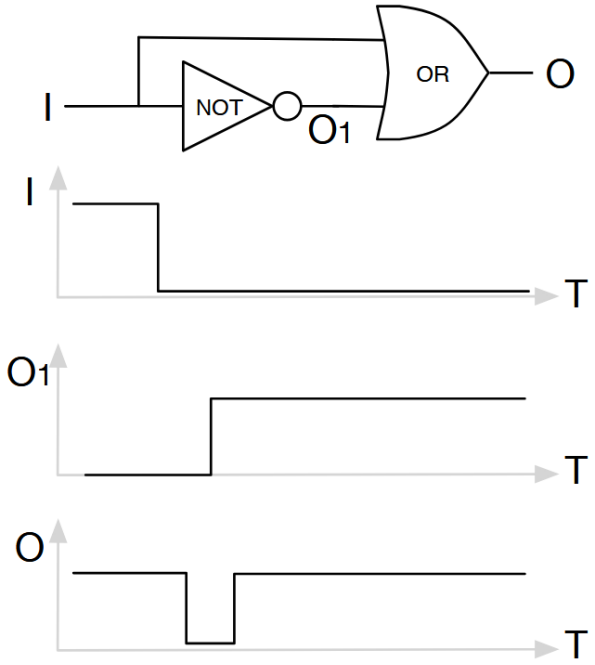
Support 32-bit Operand

- Concatenate 32 1-bit ALUs.
- Ripple carry adder.
  - CarryOut → CarryIn
- Pro: Simple, low hardware cost.
- Con: Long latency, glitch, high energy consumption.



Glitch

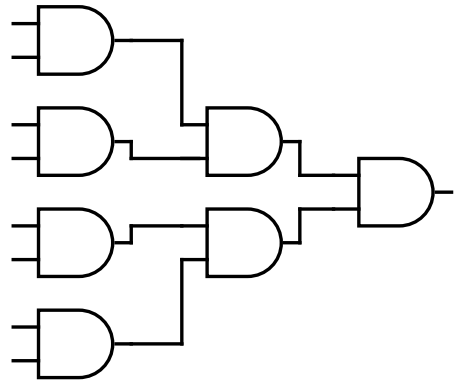
- A brief, unintended signal spike or transition that deviates from the expected output due to timing mismatches or propagation delays.
  - Increase energy consumption.
  - May cause error in next stage.
- In ripple-carry ALU, long propagation for the carry.
- How can we improve it?



Carry Look-ahead Adder

- Ripple carry adder: carry propagates one-by-one, O(n).
- Rewrite carry logic with notation $\text{OR} \rightarrow +,\ \text{AND} \rightarrow \cdot$
  - Generate: whether a carry is generated at i-th bit: $G_i = a_i \cdot b_i$
  - Propagate: whether a carry would be propagated at i-th bit: $P_i = a_i + b_i$
  - Carry at i-th bit is: $C_i = G_i + P_i \cdot C_{i-1}$
    - Either generated or propagated.
  - Expand this:

$$C_i = G_i + P_i \cdot C_{i-1} = G_i + \left(P_i \cdot \left(G_{i-1} + (P_{i-1} \cdot C_{i-2})\right)\right)$$
$$= G_i + P_i \cdot G_{i-1} + \ \ldots + P_i \cdot \ldots \cdot P_2 \cdot P_1 \cdot P_0 \cdot C_{in}$$

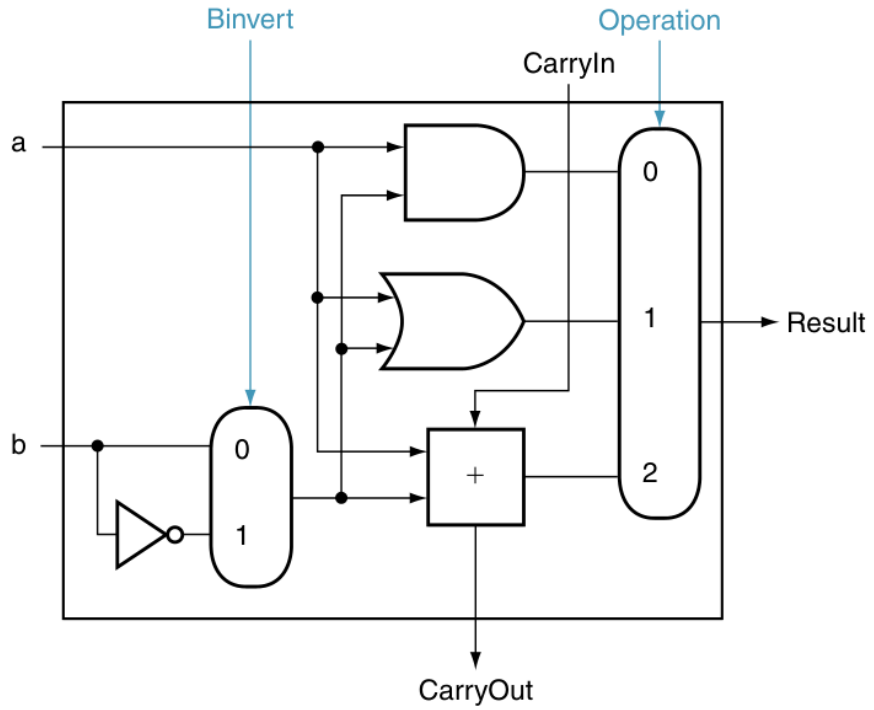


- The trick: P_i and G_i can be computed in parallel (O(1)).
- Therefore: chain of $P_i \cdot \ldots \cdot P_0 \cdot C_{in}$ can be computed with tree (O(log(n))).
- Used more gates, but less latency and glitch.

Support Subtraction

- Remember two's complement: $-x = \bar{x} + 1$
- $a - b = a + (-b) = a + \bar{b} + 1$
- Subtract is implement as:
  - Inverting b.
  - Setting CarryIn to 1 for bit 0 (LSB).
- Control signal table (x is don't care).

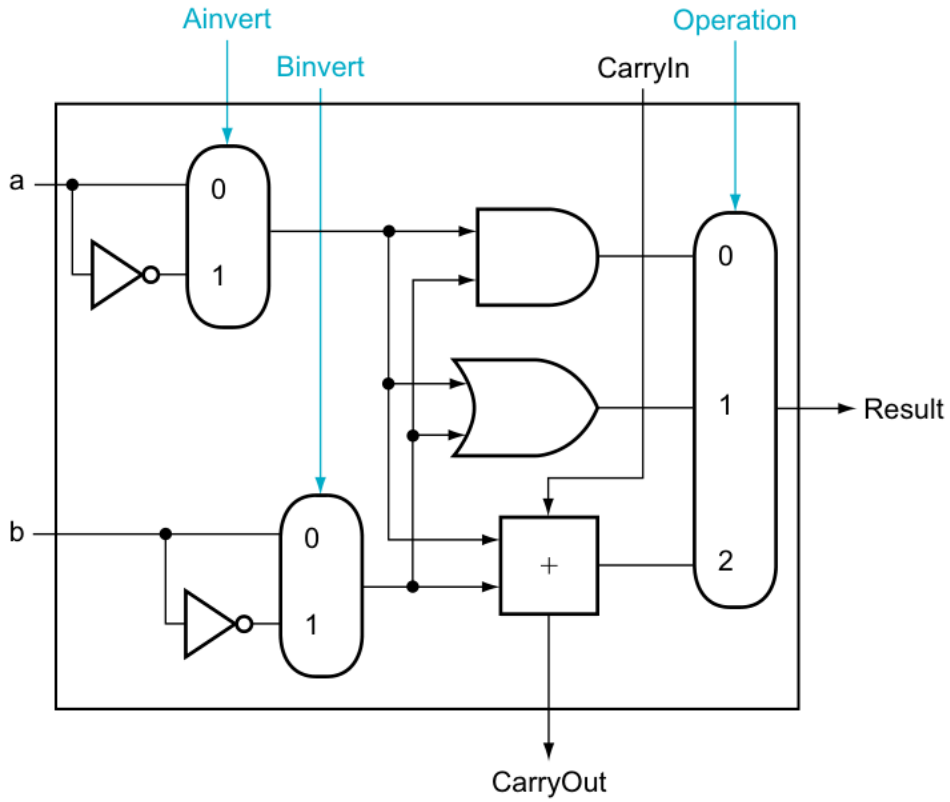
Operation	Binvert	CarryIn (LSB)	Result
00	0	x	a & b
01	0	x	a | b
10	0	0	a + b
10	1	1	a - b



Support NOR

- a NOR b = NOT (a OR b) = (NOT a) AND (NOT b)
- Invert a and b, then AND.

Operation	Ainvert	Binvert	CarryIn (LSB)	Result
00	0	0	x	a & b
00	1	1	x	~(a | b)
01	0	0	x	a | b
10	0	0	0	a + b
10	0	1	1	a - b



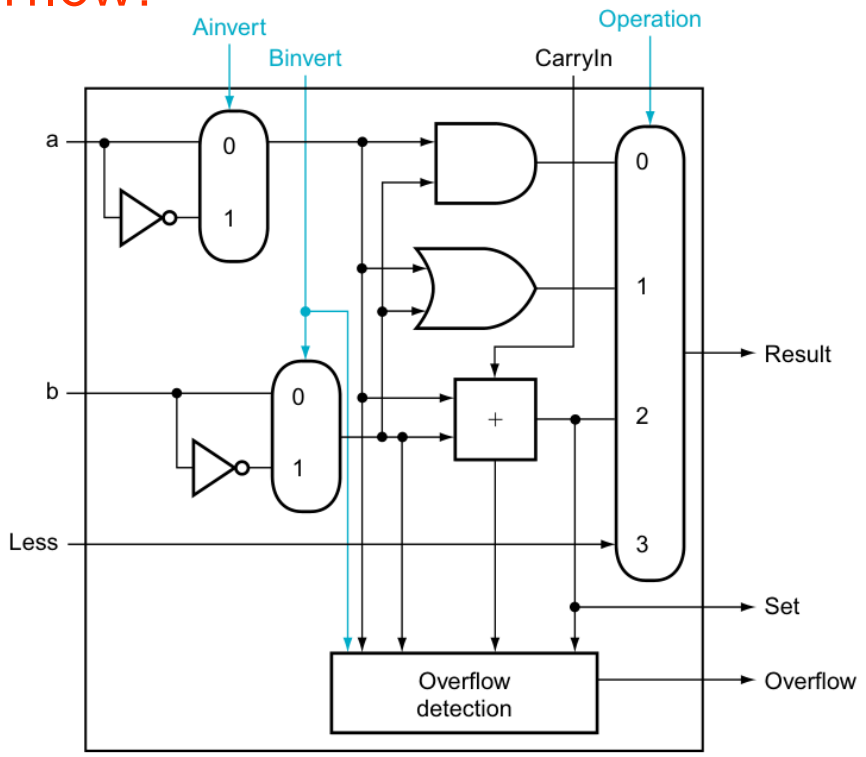
Detect Overflow

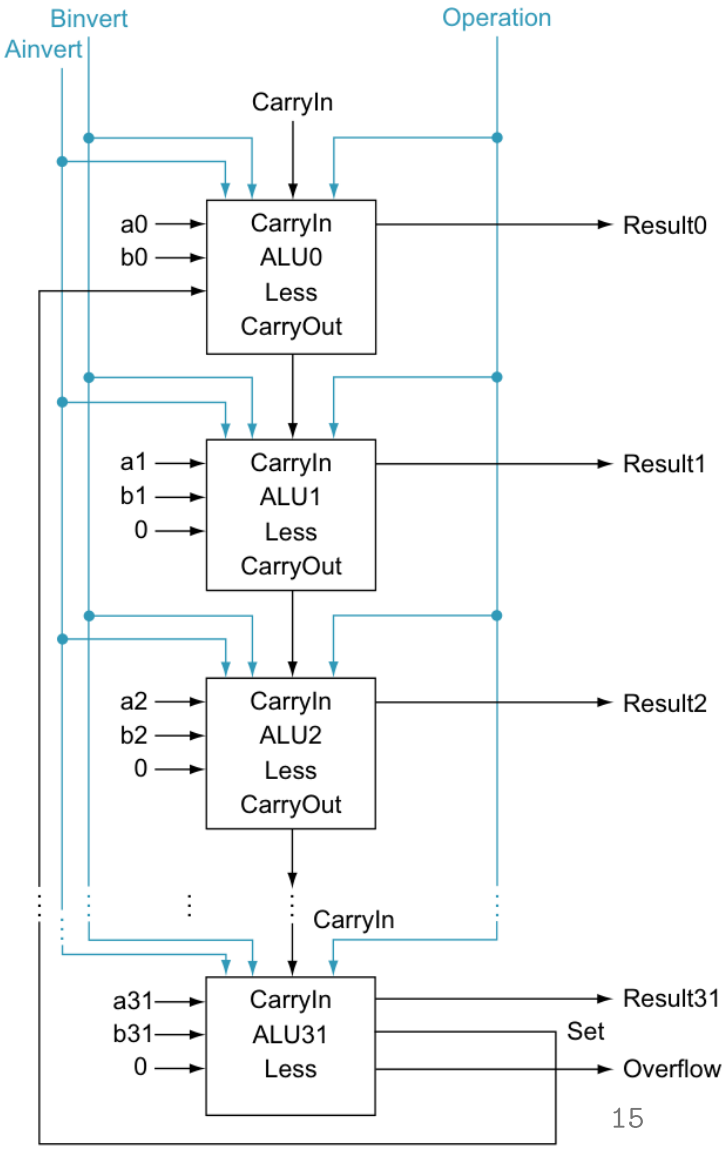
- We have limited hardware to represent integers (here 32-bit).
- Overflow: result can not be presented by hardware.
  - E.g., add two large positive/negative number exceeds the range.
  - In hardware, we discard the CarryOut of MSB.
- We can detect overflow by check sign of operands and result (MSB).
- Ex. Write the logical expression for overflow detection.
  - CarryIn (MSB) XOR CarryOut (MSB)

Operation	Operand A	Operand B	Result indicating overflow
$A + B$	≥ 0	≥ 0	< 0
$A + B$	< 0	< 0	≥ 0
$A - B$	≥ 0	< 0	< 0
$A - B$	< 0	≥ 0	≥ 0

Support Set Less Than (slt)

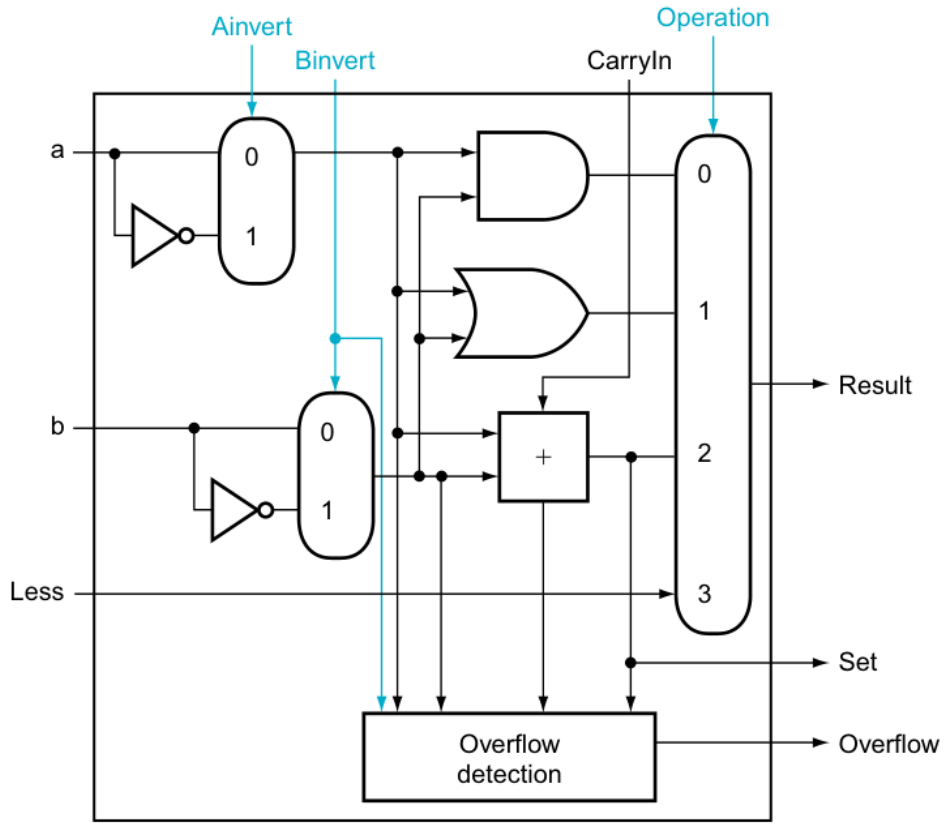
- Do subtraction.
- Set bit 0 to the output of MSB adder.
- Other bits to 0.
  - E.g. if a < b, 00...1, else, 00...0
- Only works if not underflow!





Exercise: Handling Underflow for slt

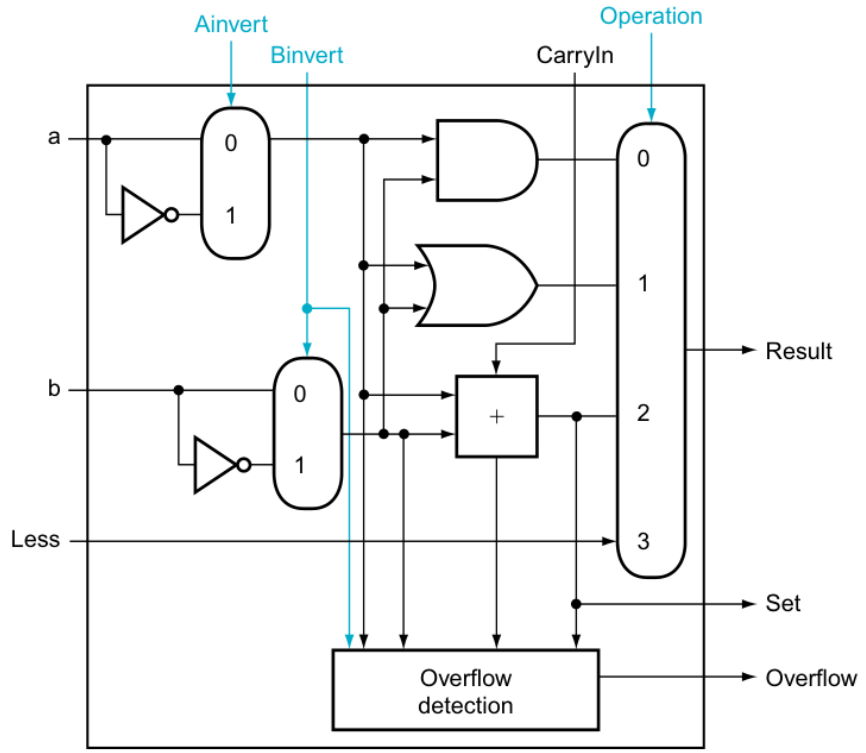
- Assume 4-bit signed integer, a = -7_{10}, b = 6_{10}, compute a – b.
- Does checking the sign bit (MSB) work?
- How to fix this?
- Set=XOR(MSB, Overflow)
  - Overflow checks if the sign bit flips incorrectly.



Support Set Less Than (slt)

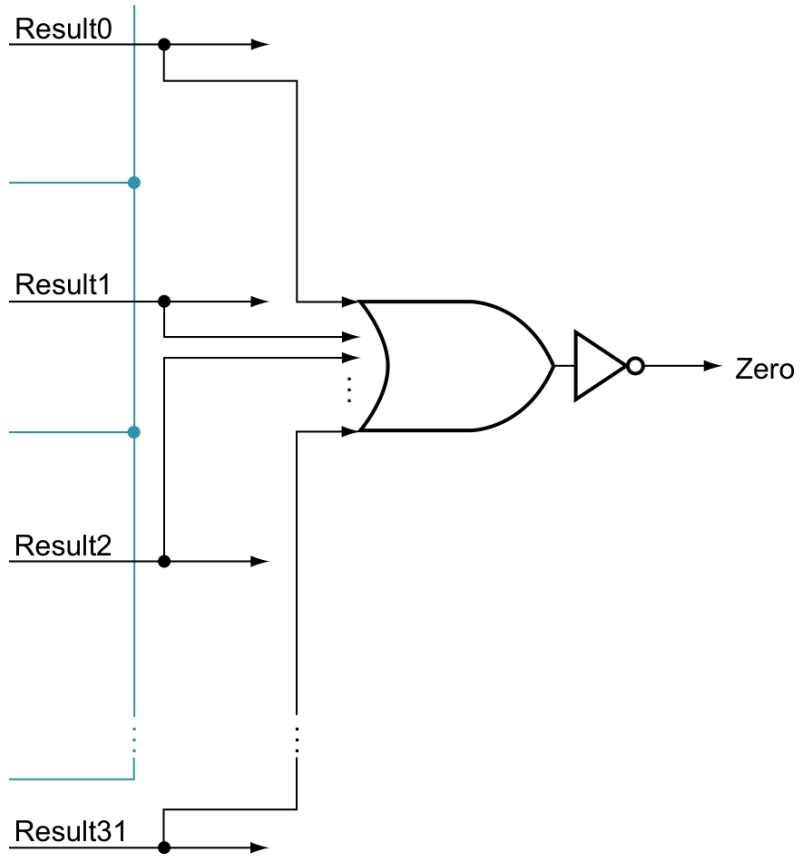
- Notice overflow detection is added to the MSB ALU unit.
- Updated control signal table:

Operation	Ainvert	Binvert	CarryIn (LSB)	Result
00	0	0	x	a & b
00	1	1	x	~(a | b)
01	0	0	x	a | b
10	0	0	0	a + b
10	0	1	1	a - b
11	0	1	1	a < b



Support Branch if Equal (beq)

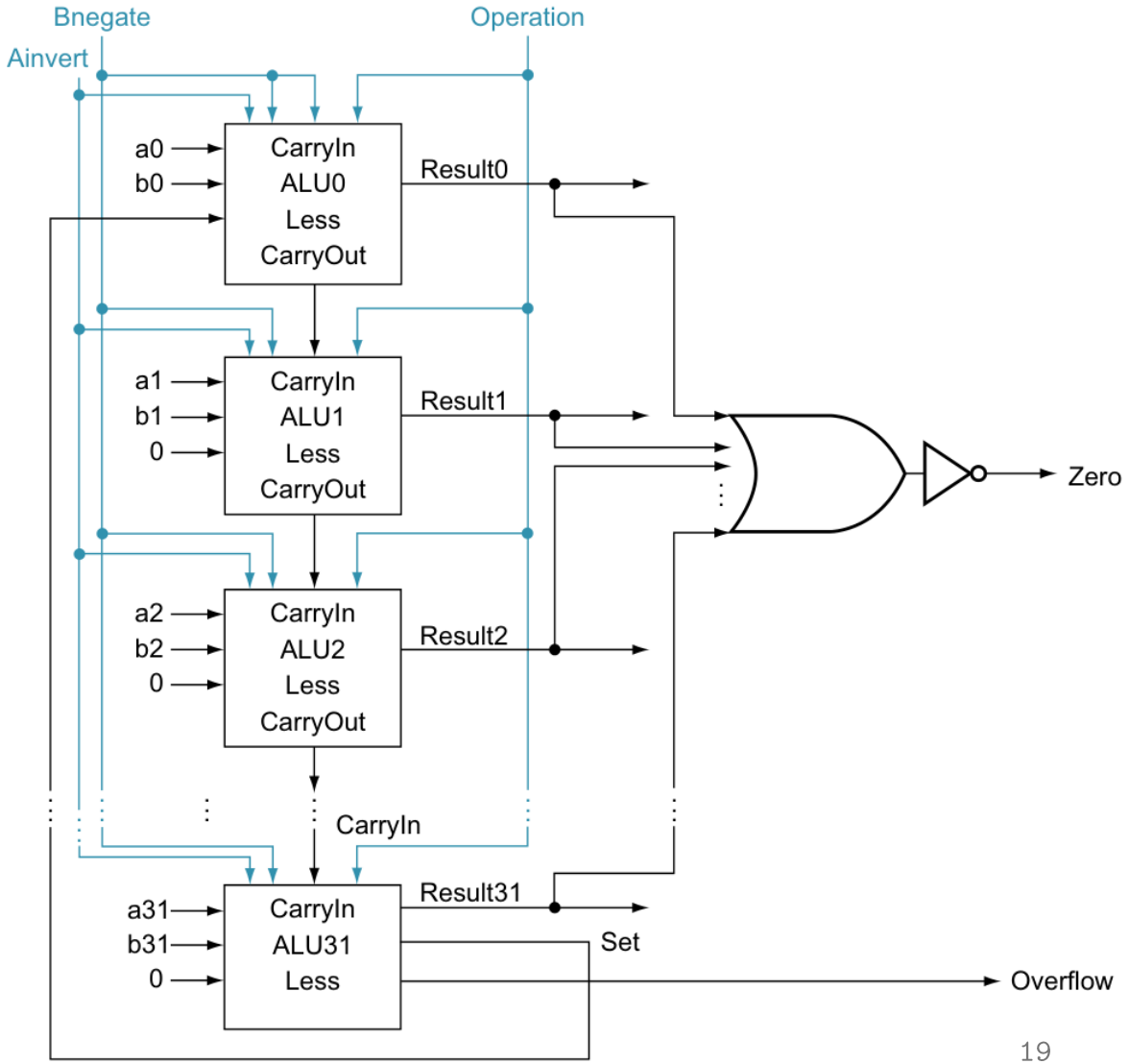
- Need to check for a == b.
- Do subtraction, and check result is 0.
- OR all result bits and negate.



Put Everything Together

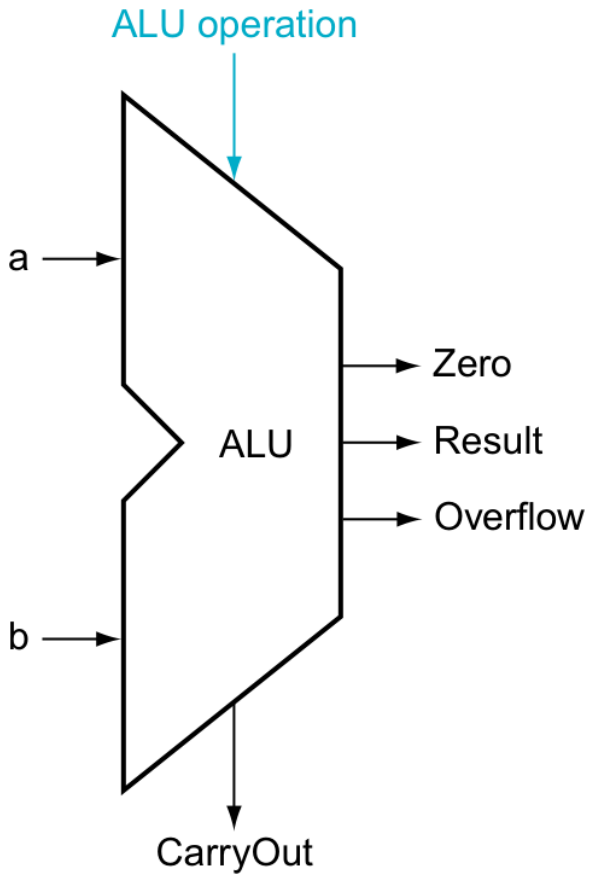
- Binvert and CarryIn (LSB) can be merged into one signal Bnegate.
- As we do not care x value in CarryIn

Operation	Ainvert	Bnegate	Result
00	0	0	a & b
00	1	1	~(a | b)
01	0	0	a | b
10	0	0	a + b
10	0	1	a - b
11	0	1	a < b



Put Everything Together

- We now support many instructions!
  - Arithmetic: add, sub, addi, addui, ...
  - Logical: and, or, ...
  - Compare: slt, ...
- The decoder translates instructions to control signals.
  - I.e., ALU operation here.

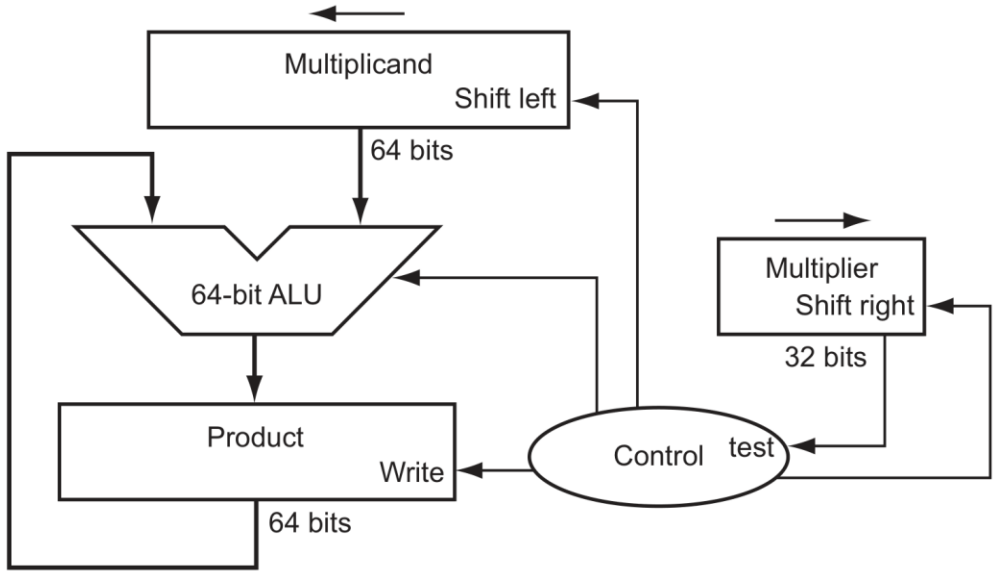


Multiplication

Multiplication

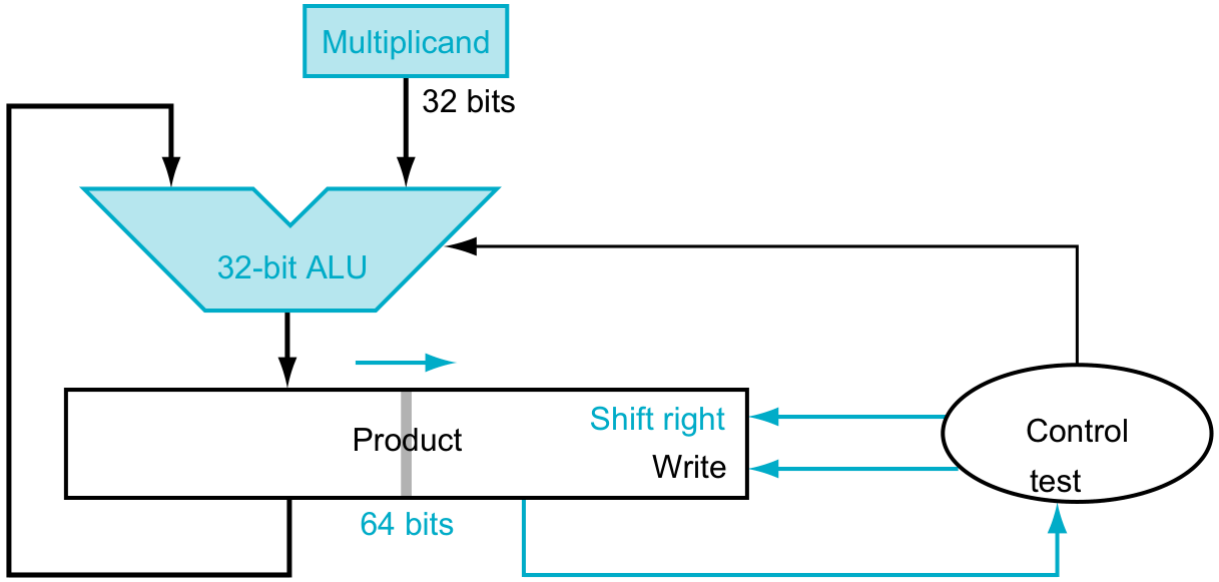
- Handle as series of shift and addition.
- A simple implementation requires 2n-bit adder.

Multiplicand		1000_{ten}
Multiplier	X	1001_{ten}
		1000
		0000
		0000
		1000
Product		1001000_{ten}



Optimized Multiplier

- Require only 32-bit adder.
- Store product and multiplier in the result register (64-bit + 1-bit carry).



Optimized Multiplier Example

- $1101_2 \times 1001_2$
- Initial state

0000_2 1101_2

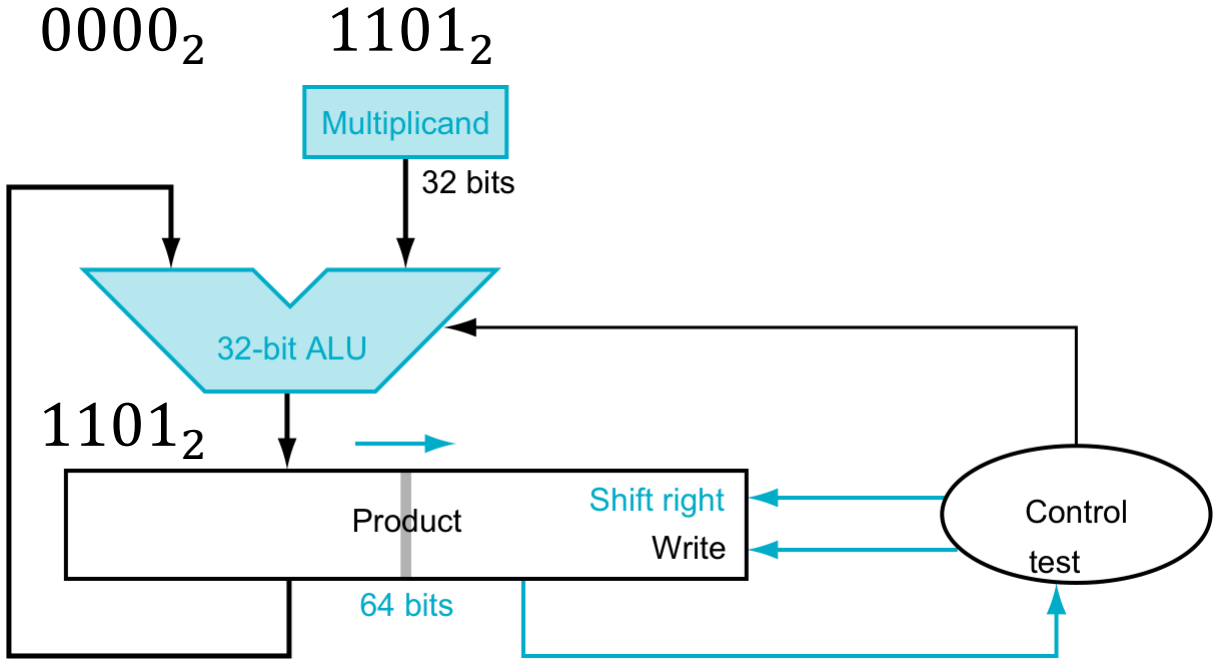
- $1101_2 \times 1001_2$
- Initial state

Product					Multiplier			
0	0	0	0	0	1	0	0	1

Notice the color separates Product and Multiplier, which is shifted right every iteration.

Optimized Multiplier Example

- $1101_2 \times 1001_2$
- Add

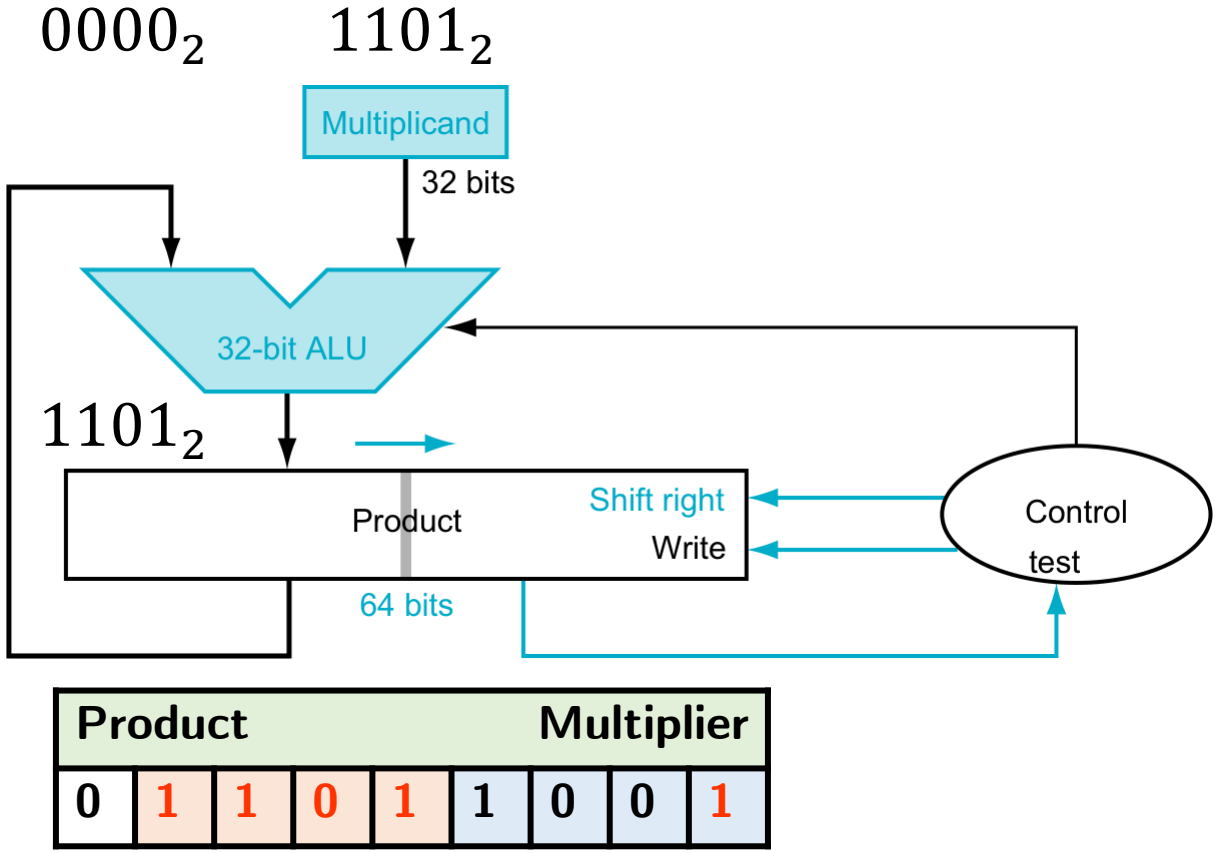


Product					Multiplier			
0	0	0	0	0	1	0	0	1

Notice the color separates Product and Multiplier, which is shifted right every iteration.

Optimized Multiplier Example

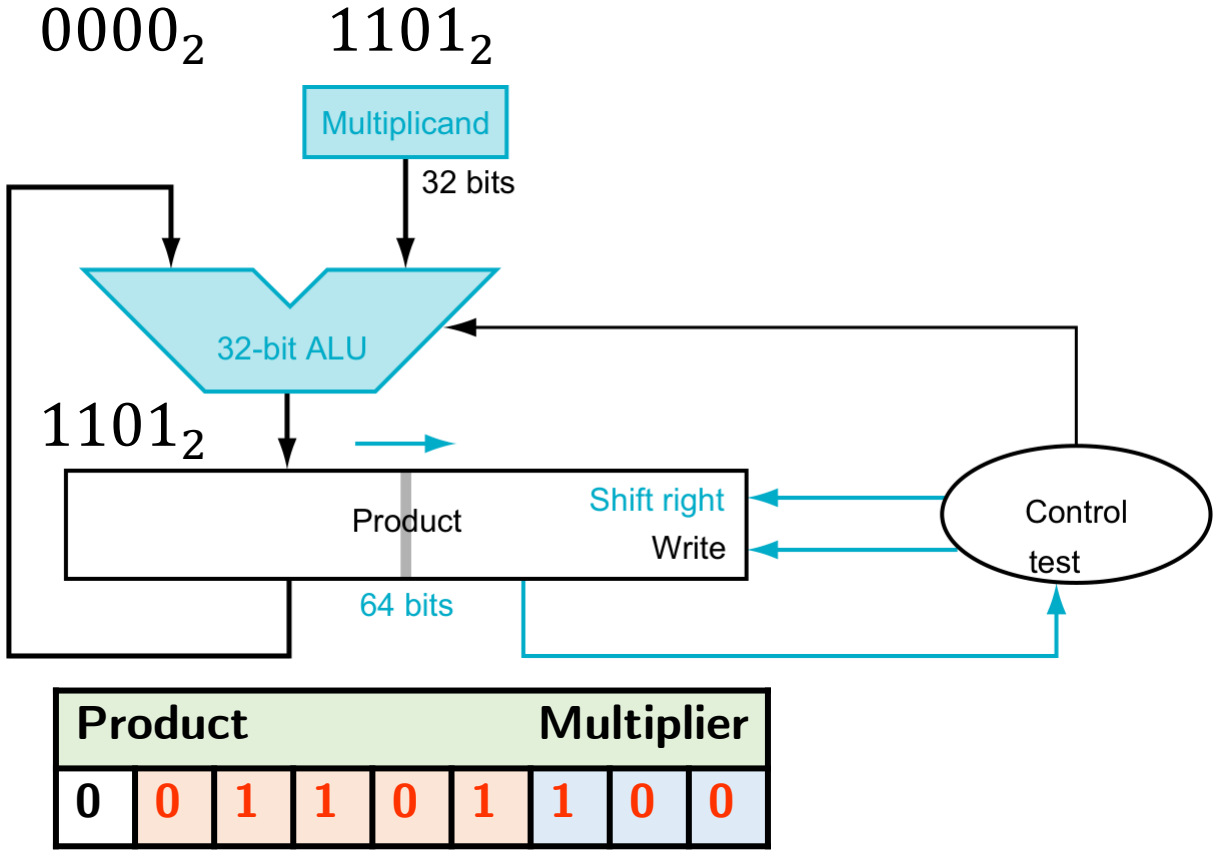
- $1101_2 \times 1001_2$
- Write product.



Notice the color separates Product and Multiplier, which is shifted right every iteration.

Optimized Multiplier Example

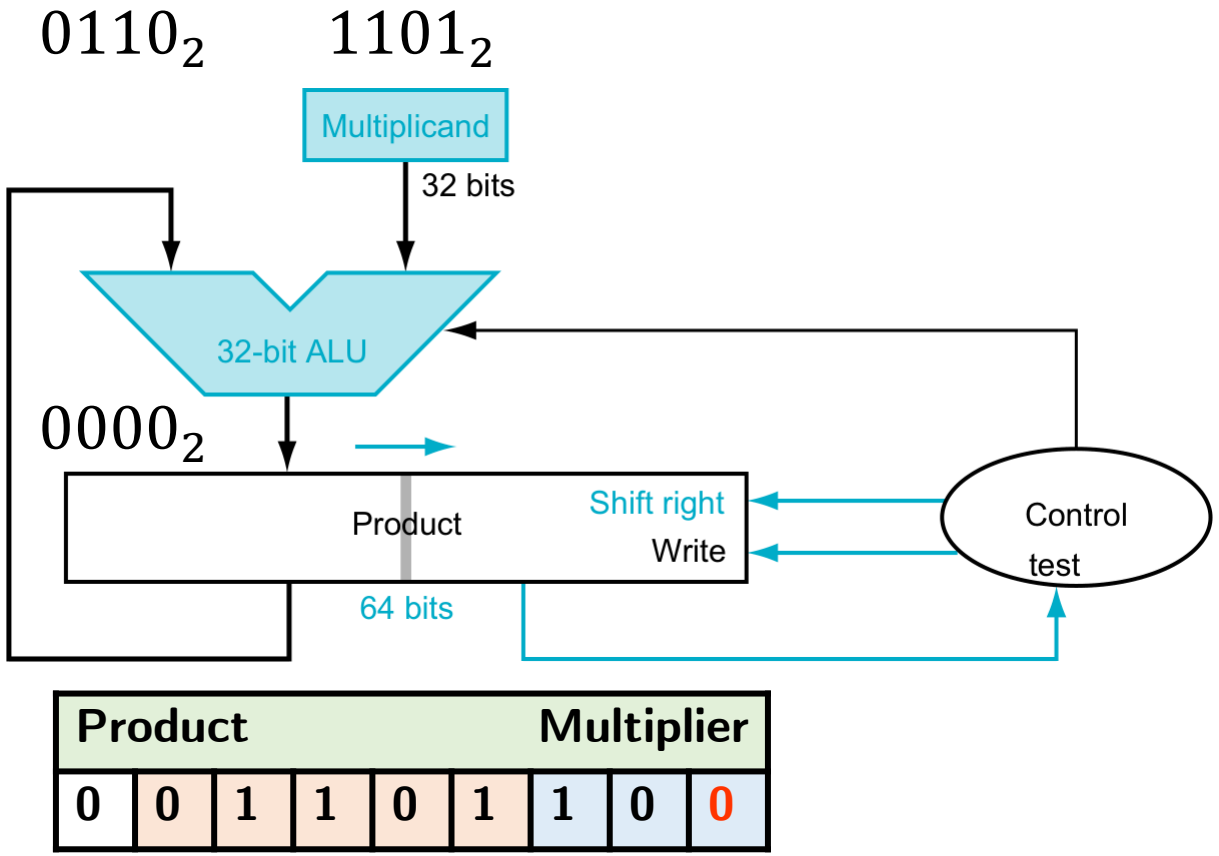
- $1101_2 \times 1001_2$
- Shift right.



Notice the color separates Product and Multiplier, which is shifted right every iteration.

Optimized Multiplier Example

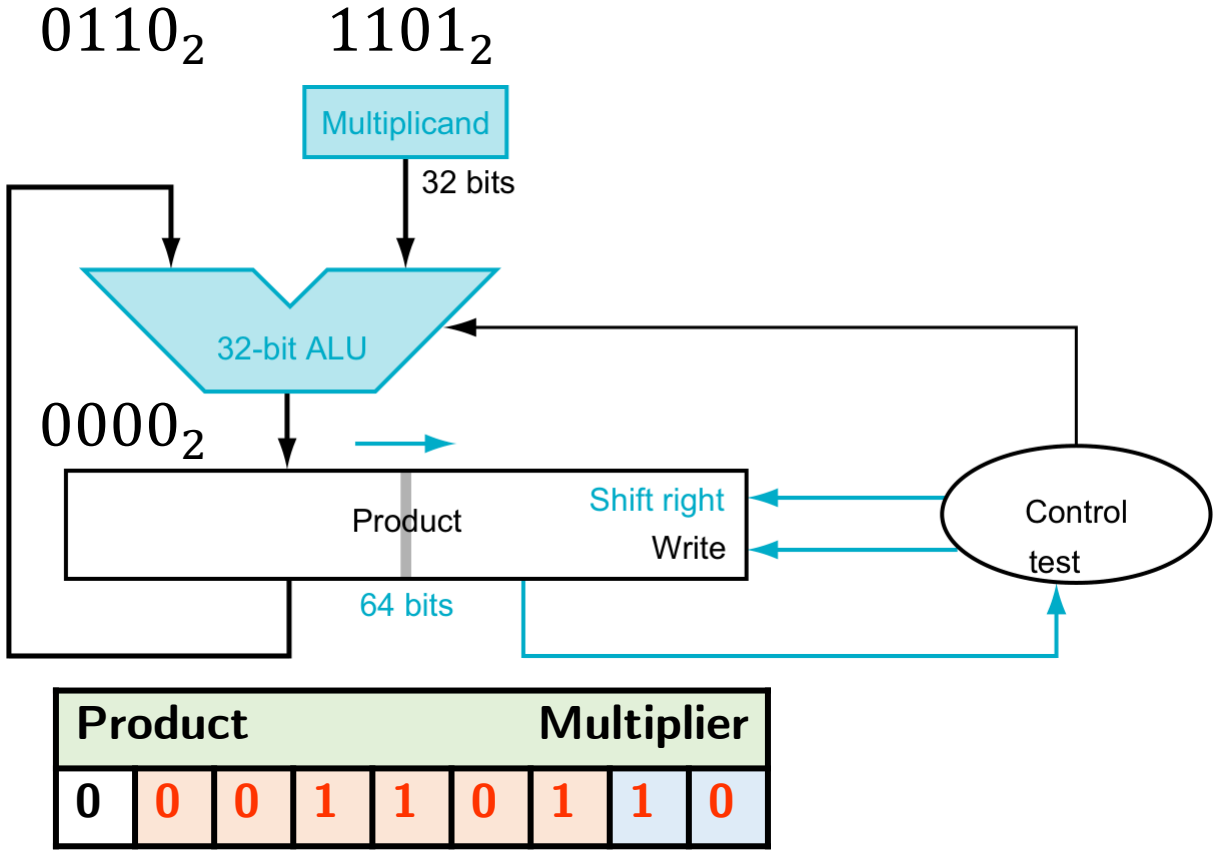
- $1101_2 \times 1001_2$
- Skip as 0 multiplier



Notice the color separates Product and Multiplier, which is shifted right every iteration.

Optimized Multiplier Example

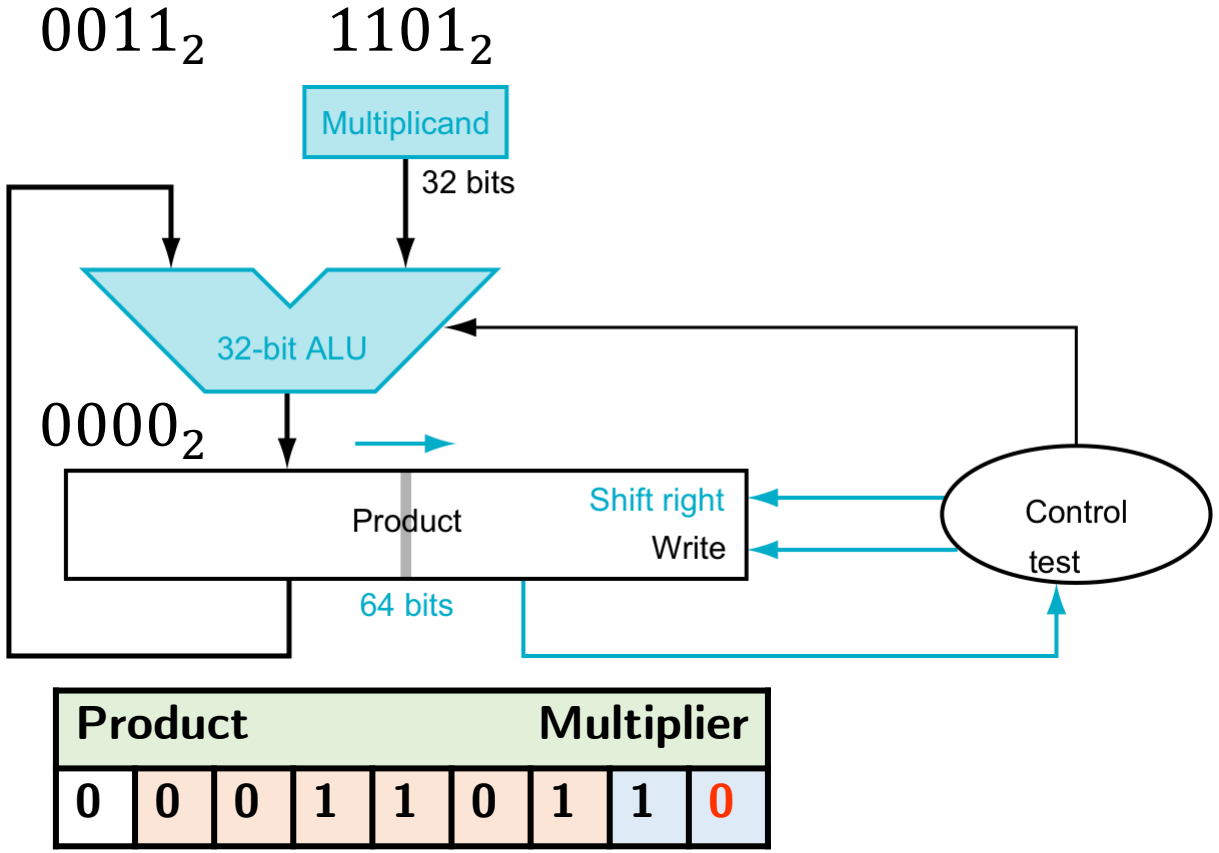
- $1101_2 \times 1001_2$
- Shift right.



Notice the color separates Product and Multiplier, which is shifted right every iteration.

Optimized Multiplier Example

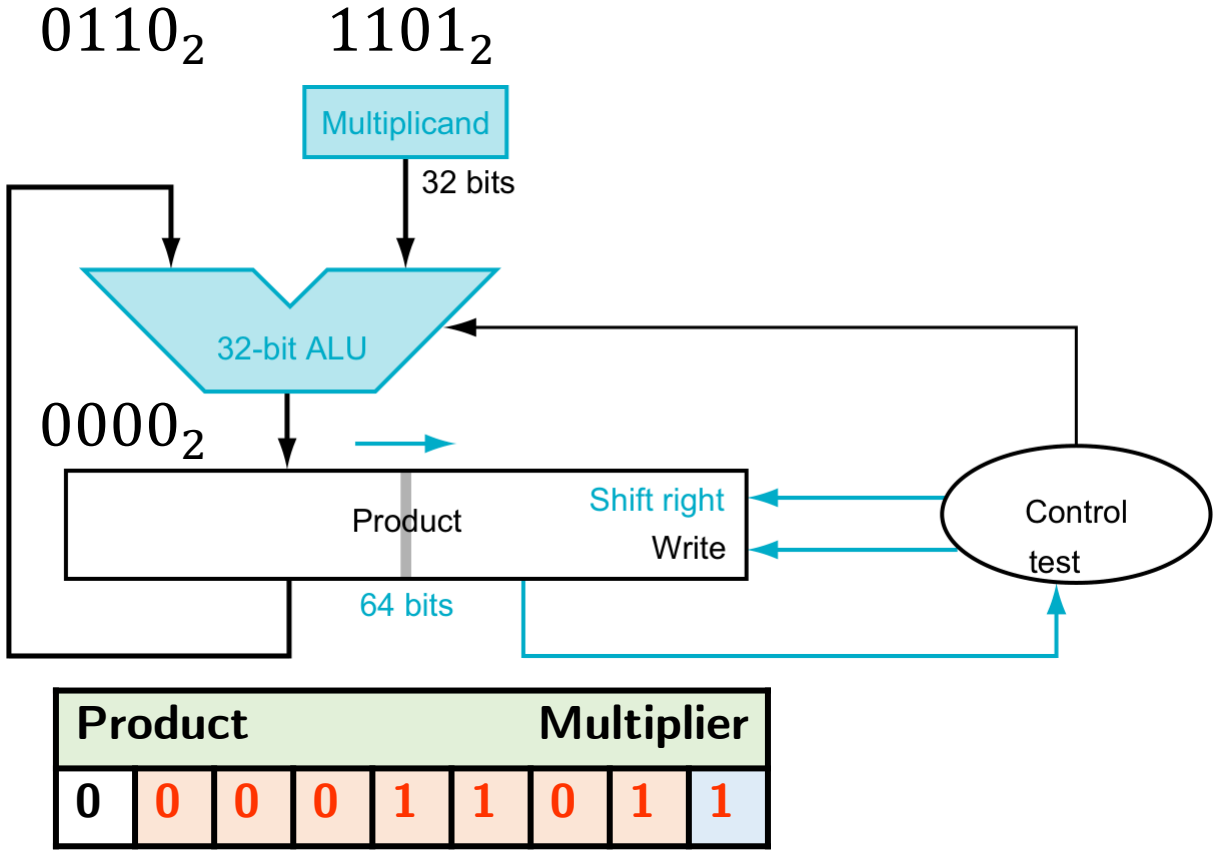
- $1101_2 \times 1001_2$
- Skip as 0 multiplier



Notice the color separates Product and Multiplier, which is shifted right every iteration.

Optimized Multiplier Example

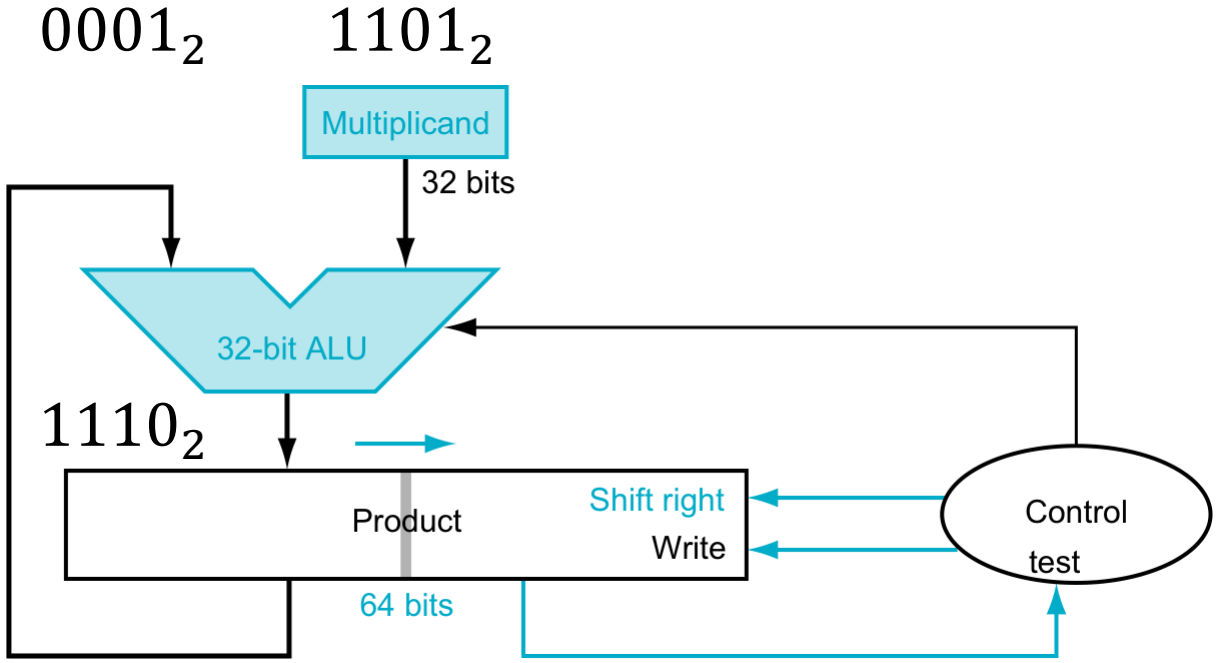
- $1101_2 \times 1001_2$
- Shift right.



Notice the color separates Product and Multiplier, which is shifted right every iteration.

Optimized Multiplier Example

- $1101_2 \times 1001_2$
- Add

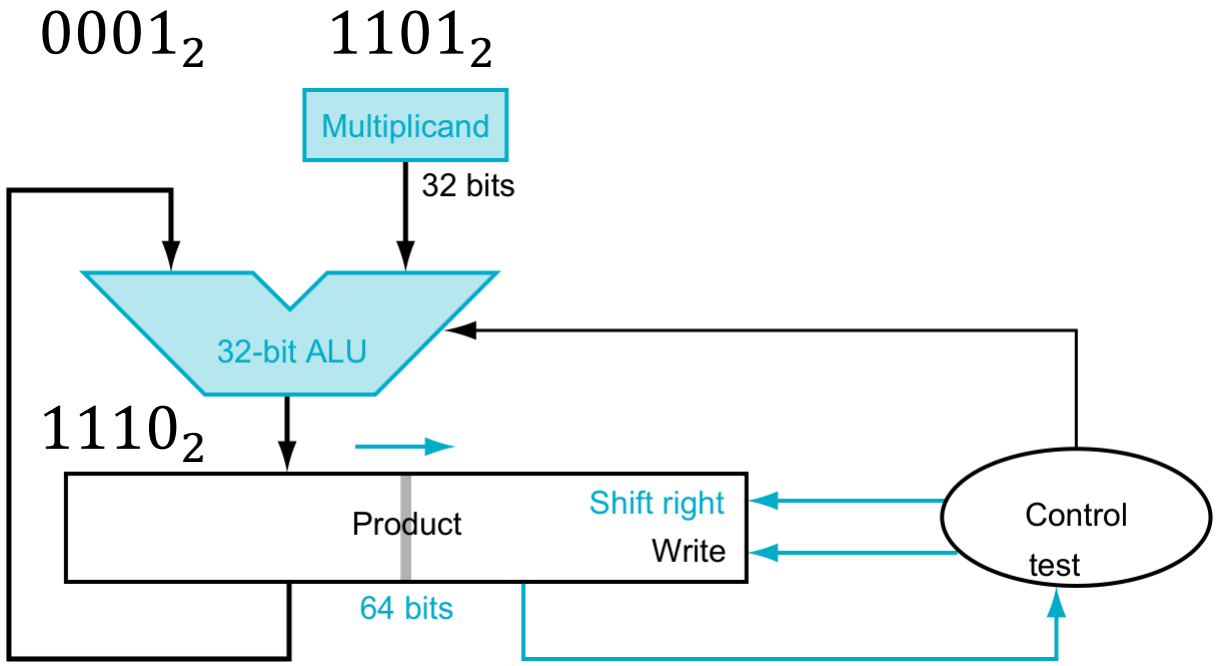


Product								Multiplier
0	0	0	0	1	1	0	1	1

Notice the color separates Product and Multiplier, which is shifted right every iteration.

Optimized Multiplier Example

- $1101_2 \times 1001_2$
- Write product

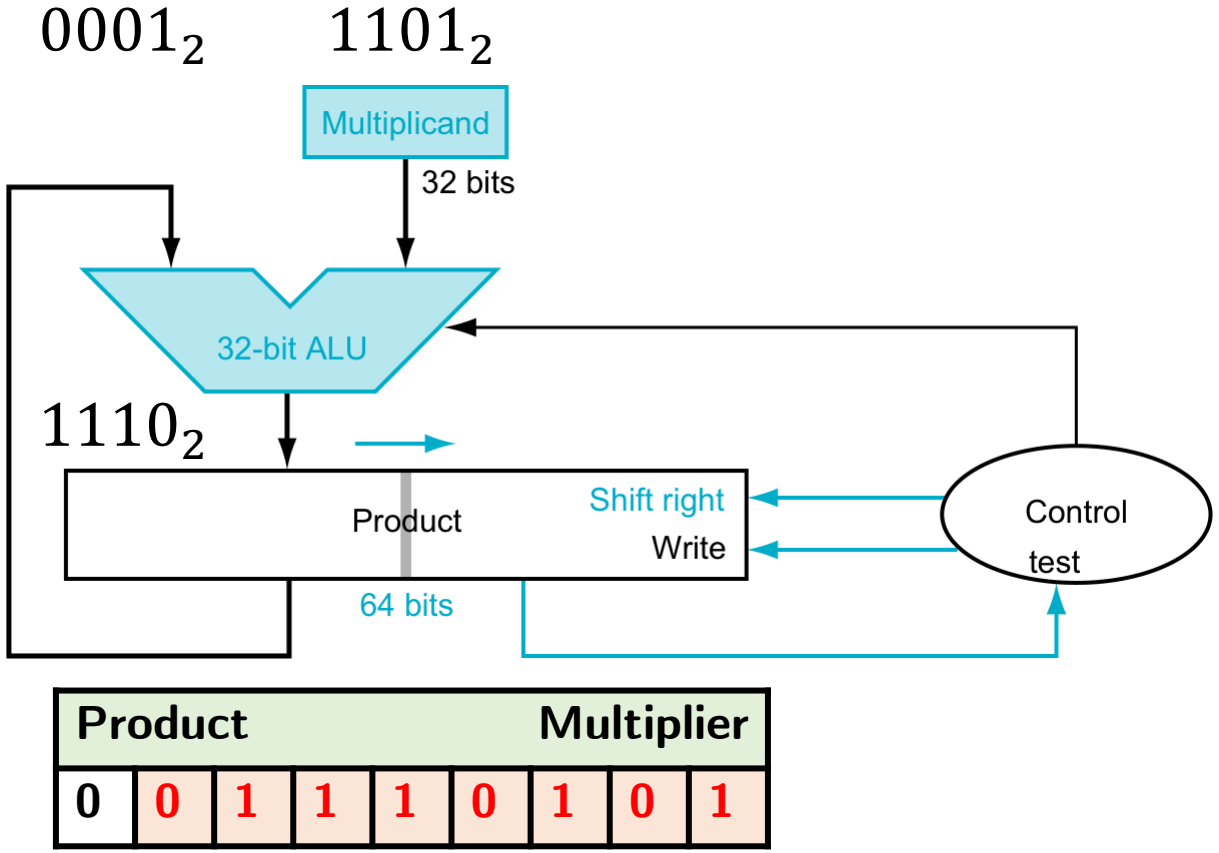


Product								Multiplier
0	1	1	1	0	1	0	1	1

Notice the color separates Product and Multiplier, which is shifted right every iteration.

Optimized Multiplier Example

- $1101_2 \times 1001_2$
- Shift right
- Result: 01110101_2



Notice the color separates Product and Multiplier, which is shifted right every iteration.

Multiply Instructions

- mul performs an 32-bit × 32-bit multiplication and places the lower 32 bits in the destination register.
  - `mul rd, rs1, rs2`

- mulh, mulhu, and mulhsu perform the same multiplication but return the upper 32 bits of the full 64-bit product, for signed×signed, unsigned×unsigned, and signed×unsigned multiplication respectively.

Division

Division

- Division is just a bunch of quotient digit guesses and left shifts and subtracts.

$$Dividend = Quotient \times Divisor + Remainder$$

$$
\begin{array}{rll}
 & 1001_{ten} & \text{Quotient} \\
\text{Divisor } 1000_{ten} & \overline{)\,1001010_{ten}} & \text{Dividend} \\
 & -1000 & \\
 & \overline{10} & \\
 & 101 & \\
 & 1010 & \\
 & -1000 & \\
 & \overline{10_{ten}} & \text{Remainder}
\end{array}
$$

Division Hardware

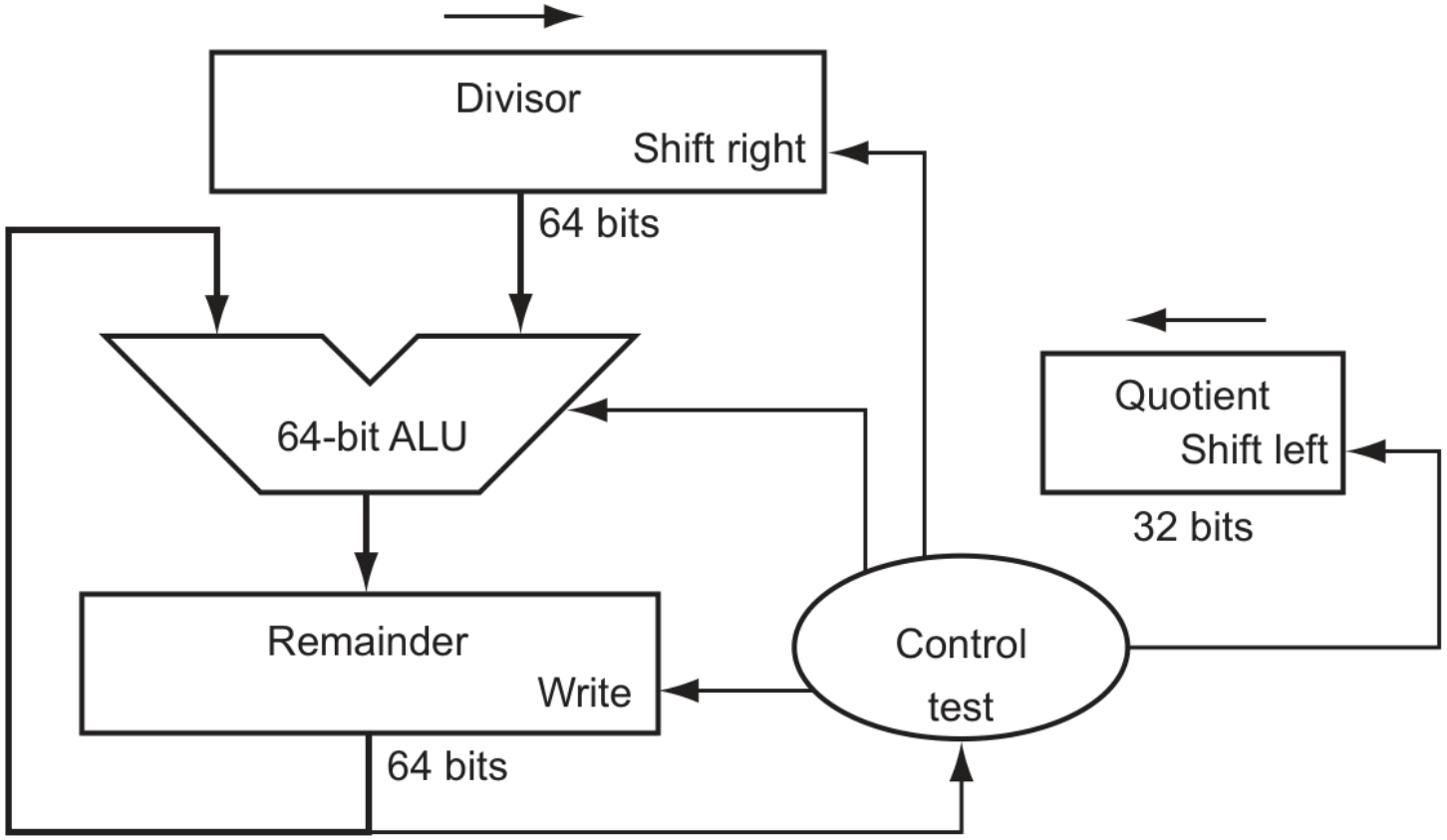


FIGURE 3.8 First version of the division hardware. The Divisor register, ALU, and Remainder register are all 64 bits wide, with only the Quotient register being 32 bits. The 32-bit divisor starts in the left half of the Divisor register and is shifted right 1 bit each iteration. The remainder is initialized with the dividend. Control decides when to shift the Divisor and Quotient registers and when to write the new value into the Remainder register.

Division Example

- Divide 0111_2 by 0010_2

Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem – Div	0000	0010 0000	1110 0111
	2b: Rem < 0 ⟹ +Div, SLL Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem – Div	0000	0001 0000	1111 0111
	2b: Rem < 0 ⟹ +Div, SLL Q, Q0 = 0	0000	0001 0000	0000 0111
	3: Shift Div right	0000	0000 1000	0000 0111
3	1: Rem = Rem – Div	0000	0000 1000	1111 1111
	2b: Rem < 0 ⟹ +Div, SLL Q, Q0 = 0	0000	0000 1000	0000 0111
	3: Shift Div right	0000	0000 0100	0000 0111
4	1: Rem = Rem – Div	0000	0000 0100	0000 0011
	2a: Rem ≥ 0 ⟹ SLL Q, Q0 = 1	0001	0000 0100	0000 0011
	3: Shift Div right	0001	0000 0010	0000 0011
5	1: Rem = Rem – Div	0001	0000 0010	0000 0001
	2a: Rem ≥ 0 ⟹ SLL Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

Division Instruction

- div perform a 32 bits by 32 bits signed integer division of rs1 by rs2, rounding towards zero.
  - `div rd, rs1, rs2`

- div and divu perform signed and unsigned integer division of 32 bits by 32 bits.
- rem and remu provide the remainder of the corresponding division operation.