# CENG 3420
# Computer Organization & Design

## Lecture 06: Arithmetic and Logic Unit

Bei Yu
CSE Department, CUHK
`byu@cse.cuhk.edu.hk`

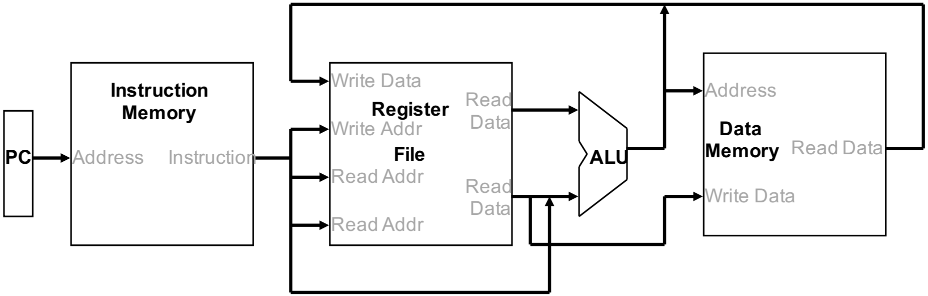
(Textbook: Chapters 3.2 & A.5)

2024 Spring

## Overview

## Overview

## Abstract Implementation View



## Arithmetic

Where we've been: **abstractions**

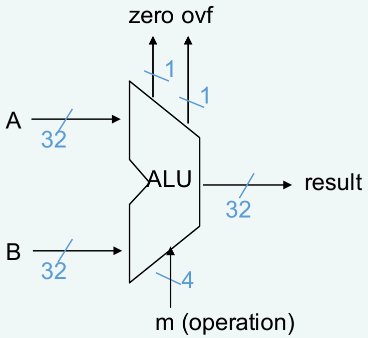
- Instruction Set Architecture (ISA)
- Assembly and machine language

## Arithmetic

Where we've been: **abstractions**

- Instruction Set Architecture (ISA)
- Assembly and machine language

### What's up ahead: Implementing the ALU architecture



## Machine Number Representation

- Bits are just bits (have no inherent meaning)[1]
- Binary numbers (base 2) – integers

Of course, it gets more complicated:

- storage locations (e.g., register file words) are finite, so have to worry about overflow (i.e., when the number is too big to fit into 32 bits)
- have to be able to represent negative numbers, e.g., how do we specify -8 in

```
addi    $sp, $sp, -8    #$sp = $sp - 8
```

- in real systems have to provide for more than just integers, e.g., fractions and real numbers (and floating point) and alphanumeric (characters)

[1]conventions define the relationships between bits and numbers

## RISC-V Representation

32-bit signed numbers (2's complement):

$$
\begin{array}{l}
\texttt{0000 0000 0000 0000 0000 0000 0000 0000}_{\text{two}} = 0_{\text{ten}} \\
\texttt{0000 0000 0000 0000 0000 0000 0000 0001}_{\text{two}} = +\ 1_{\text{ten}} \\
\texttt{0000 0000 0000 0000 0000 0000 0000 0010}_{\text{two}} = +\ 2_{\text{ten}} \\
\ldots \\
\texttt{0111 1111 1111 1111 1111 1111 1111 1110}_{\text{two}} = +\ 2{,}147{,}483{,}646_{\text{ten}} \\
\texttt{0111 1111 1111 1111 1111 1111 1111 1111}_{\text{two}} = +\ 2{,}147{,}483{,}647_{\text{ten}} \\
\texttt{1000 0000 0000 0000 0000 0000 0000 0000}_{\text{two}} = -\ 2{,}147{,}483{,}648_{\text{ten}} \\
\texttt{1000 0000 0000 0000 0000 0000 0000 0001}_{\text{two}} = -\ 2{,}147{,}483{,}647_{\text{ten}} \\
\texttt{1000 0000 0000 0000 0000 0000 0000 0010}_{\text{two}} = -\ 2{,}147{,}483{,}646_{\text{ten}} \\
\ldots \\
\texttt{1111 1111 1111 1111 1111 1111 1111 1101}_{\text{two}} = -\ 3_{\text{ten}} \\
\texttt{1111 1111 1111 1111 1111 1111 1111 1110}_{\text{two}} = -\ 2_{\text{ten}} \\
\texttt{1111 1111 1111 1111 1111 1111 1111 1111}_{\text{two}} = -\ 1_{\text{ten}}
\end{array}
$$

What if the bit string represented addresses?

- need operations that also deal with only positive (unsigned) integers

## Two’s Complement Operations

- Negating a two’s complement number – complement all the bits and then add a 1
  - remember: “negate” and “invert” are quite different!

- Converting n-bit numbers into numbers with more than n bits:
  - 16-bit immediate gets converted to 32 bits for arithmetic
  - sign extend: copy the most significant bit (the sign bit) into the other bits

```
0010  -> 0000 0010
1010  -> 1111 1010
```

  - sign extension versus zero extend (`lb` vs. `lbu`)

We use 32-bit signed number "0x 00 00 00 0F" to represent 15. Which one can represent -15?

- A: 0x FF FF FF E0
- B: 0x FF FF FF EF
- C: 0x FF FF FF F0
- D: 0x FF FF FF F1

We use 32-bit signed number "0x 00 00 00 0F" to represent 15. Which one can represent -15?

- A: 0x FF FF FF E0
- B: 0x FF FF FF EF
- C: 0x FF FF FF F0
- D: 0x FF FF FF F1

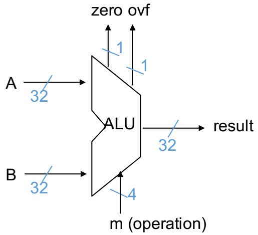
Answer: D

## Design the RISC-V Arithmetic Logic Unit (ALU)

- Must support the Arithmetic/Logic operations of the ISA

```
RV 32I:
add, sub, mul, mulh, mulhu, mulhsu,
div, divu, rem, li, addi, sll, srl,
sra, or, xor, not, slt, sltu, slli,
srli, srai, andi, ori, xori, slti,
sltiu,

RV 64I:
addw, subw, remu, mulw, divw, divuw,
remw, remuw, addiw, sllw, srlw, sraw,
srliw, sraiw,
```



- With special handling for:
  - sign extend: `addi, slti, sltiu`
  - zero extend: `andi, xori`
  - Overflow detected: `add, addi, sub`

# RISC-V Arithmetic and Logic Instructions

| | 31 … 20 | 19 … 15 | 14 … 12 | 11 … 7 | 6 … 0 |
|---|---|---|---|---|---|
| I-Type | Imm[11:0] | rs1 | funct3 | rd | opcode |

| | 31 … 25 | 24 … 20 | 19 … 15 | 14 … 12 | 11 … 7 | 6 … 0 |
|---|---|---|---|---|---|---|
| R-Type | funct7 | rs2 | rs1 | funct3 | rd | opcode |

I-Type

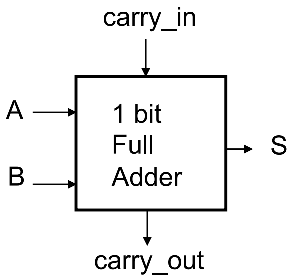
| Type | opcode | funct | Imm[11:5] |
|---|---|---|---|
| ADDI | 0010011 | 000 | xx (any) |
| SLLI | 0010011 | 001 | 0000000 |
| SLTI | 0010011 | 010 | xx |
| SLTIU | 0010011 | 011 | xx |
| SRLI | 0010011 | 101 | 0000000 |
| SRAI | 0010011 | 101 | 0100000 |
| ORI | 0010011 | 110 | xx |
| ANDI | 0010011 | 111 | xx |

R-Type

| Type | opcode | funct |
|---|---|---|
| ADD | 0110011 | 0000000 000 |
| SUB | 0110011 | 0100000 000 |
| SLL | 0110011 | 0000000 001 |
| SLT | 0110011 | 0000000 010 |
| SLTU | 0110011 | 0000000 011 |
| XOR | 0110011 | 0000000 100 |
| SRL | 0110011 | 0000000 101 |
| SRA | 0110011 | 0100000 101 |

## Addition Unit

## Building a 1-bit Binary Adder

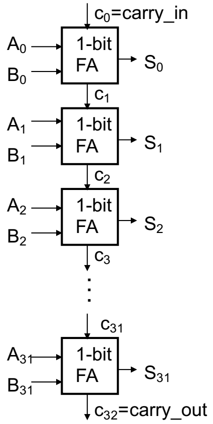


| A | B | carry_in | carry_out | S |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

S = A xor B xor carry_in
carry_out = A&B | A&carry_in | B&carry_in
(majority function)

- How can we use it to build a 32-bit adder?
- How can we modify it easily to build an adder/subtractor?

## Building 32-bit Adder



- Just connect the carry-out of the least significant bit FA to the carry-in of the next least significant bit and connect ...

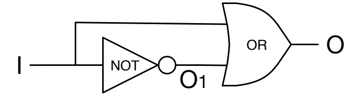
- Ripple Carry Adder (RCA)
  - ☺: simple logic, so small (low cost)
  - ☹: slow and lots of glitching (so lots of energy consumption)

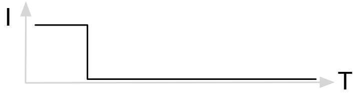
# Glitch

## Glitch

invalid and unpredicted output that can be read by the next stage and result in a wrong action

Example: Draw the propagation delay



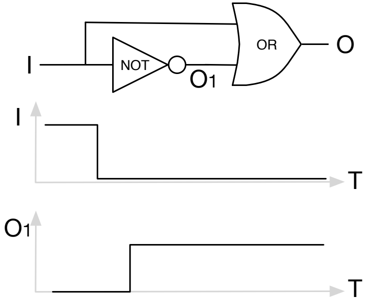


## Glitch

**Glitch**

invalid and unpredicted output that can be read by the next stage and result in a wrong action

Example: Draw the propagation delay

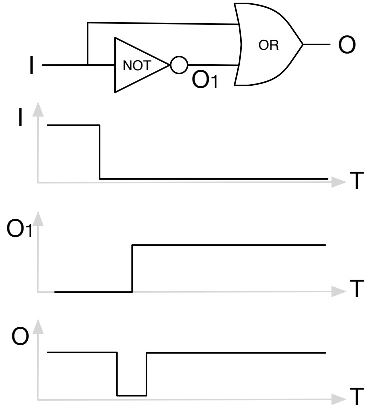


## Glitch

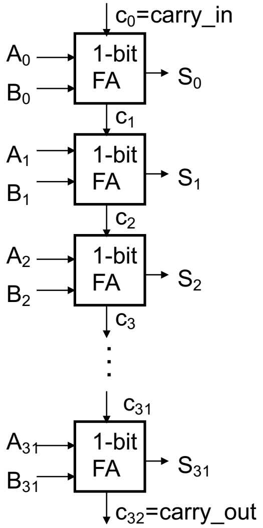
### Glitch

invalid and unpredicted output that can be read by the next stage and result in a wrong action

Example: Draw the propagation delay



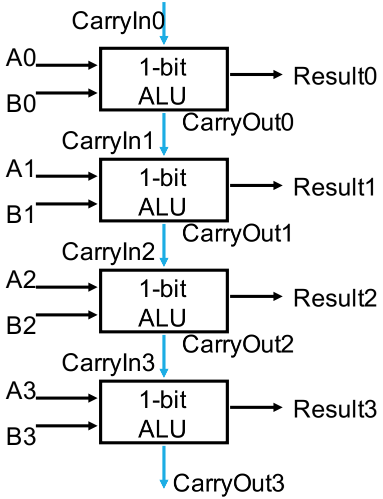
## Glitch in RCA



| A | B | carry_in | carry_out | 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** |

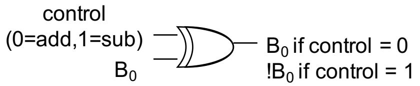
## But What about Performance?

- Critical path of n-bit ripple-carry adder is $n \times CP$
- Design trick: throw hardware at it (Carry Lookahead)



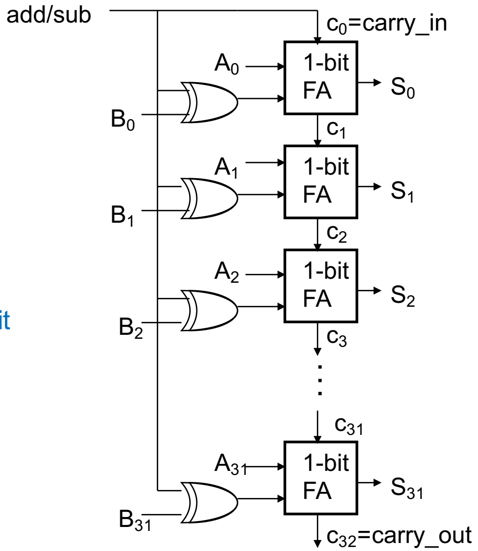
# A 32-bit Ripple Carry Adder/Subtractor

- complement all the bits



- add a 1 in the least significant bit

```
A     0111   ->    0111
B   - 0110   ->  + 1001
      ----            1
      0001        -----
                 1 0001
```



To implement an adder, we need to calculate: $S = A \oplus B \oplus carry\_in$. Which part in this equation makes ripple-carry adder slow? Why?

- A: $A$
- B: $B$
- C: $\oplus$ (XOR)
- D: $carry\_in$

To implement an adder, we need to calculate: $S = A \oplus B \oplus carry\_in$. Which part in this equation makes ripple-carry adder slow? Why?

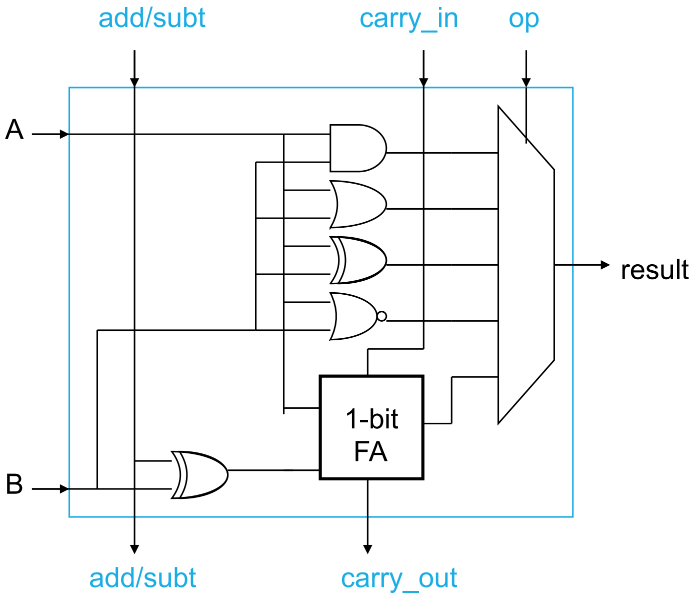
- A: $A$
- B: $B$
- C: $\oplus$ (XOR)
- D: $carry\_in$

Answer: D

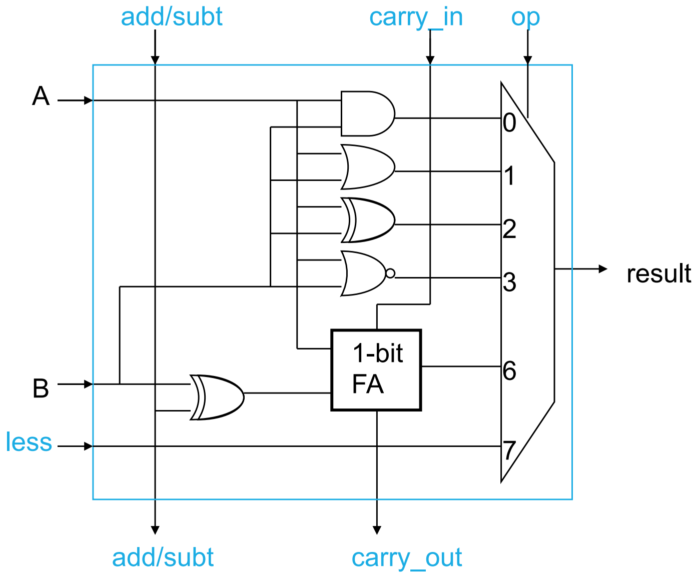
## Tailoring the ALU to the ISA

- Also need to support the logic operations (`and`, `nor`, `or`, `xor`)
  - Bit wise operations (no carry operation involved)
  - Need a logic gate for each function and a mux to choose the output
- Also need to support the set-on-less-than instruction (slt)
  - Uses subtraction to determine if $(a - b) < 0$ (implies $a < b$)
- Also need to support test for equality (`bne`, `beq`)
  - Again use subtraction: $(a - b) = 0$ implies $a = b$
- Also need to add overflow detection hardware
  - overflow detection enabled only for `add, addi, sub`
- Immediates are sign extended outside the ALU with wiring (i.e., no logic needed)

## A Simple ALU Cell with Logic Op Support



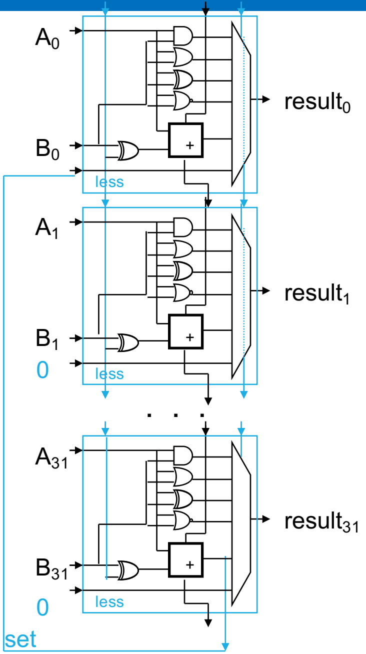
## A Simple ALU Cell with Logic Op Support



Modifying the ALU Cell for `slt`

# Modifying the ALU for `slt`

- First perform a subtraction
- Make the result 1 if the subtraction yields a negative result
- Make the result 0 if the subtraction yields a positive result
- Tie the most significant sum bit (sign bit) to the low order less input



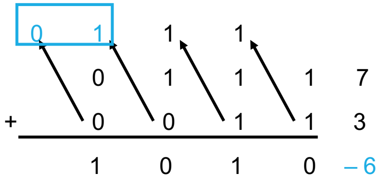
## Overflow Detection

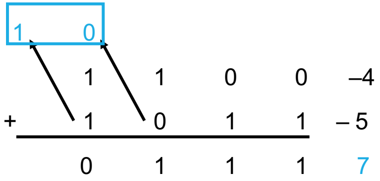
Overflow occurs when the result is too large to represent in the number of bits allocated

- adding two positives yields a negative
- or, adding two negatives gives a positive
- or, subtract a negative from a positive gives a negative
- or, subtract a positive from a negative gives a positive

Question: **prove** you can detect overflow by:

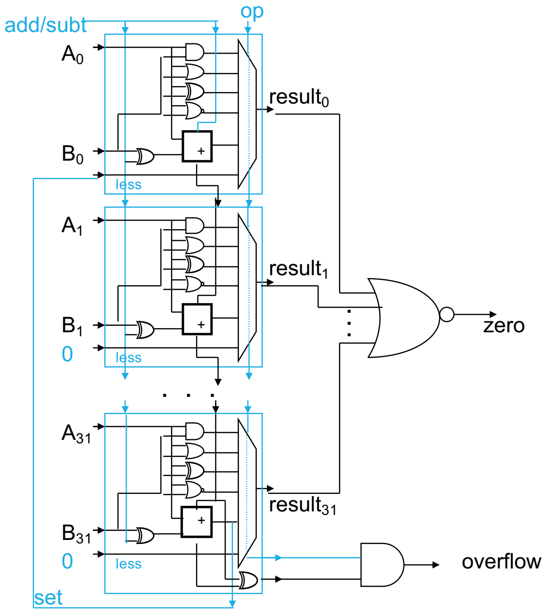
Carry into MSB xor Carry out of MSB





## Modifying the ALU for Overflow

- Modify the most significant cell to determine overflow output setting
- Enable overflow bit setting for signed arithmetic (`add, addi, sub`)



## Overflow Detection and Effects

- On overflow, an exception (interrupt) occurs
- Control jumps to predefined address for exception
- Interrupted address (address of instruction causing the overflow) is saved for possible resumption
- Don't always want to detect (interrupt on) overflow

## Which one will cause overflow?

- A: 0x7EDDCCBB + 0x03000000
- B: 0x44444444 + 0x33333333
- C: 0xFFFFFFFF + 0xFFFFFFFF
- D: 0xFFFFFFFF + 0x00000001

## Which one will cause overflow?

- A: 0x7EDDCCBB + 0x03000000
- B: 0x44444444 + 0x33333333
- C: 0xFFFFFFFF + 0xFFFFFFFF
- D: 0xFFFFFFFF + 0x00000001

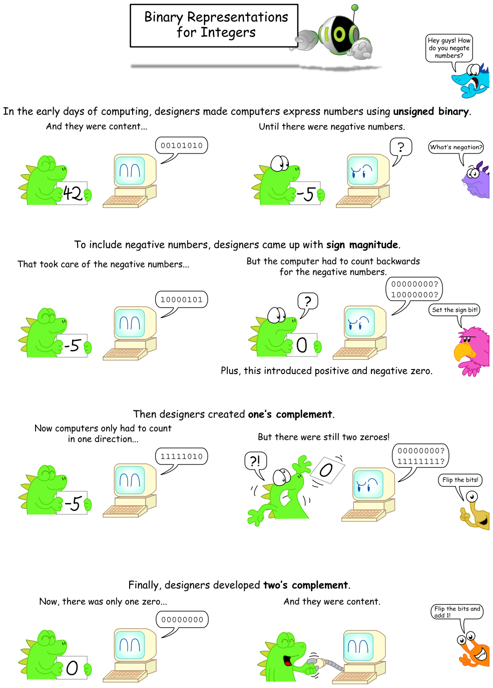
Answer: A
note: NOT C, as 0xFFFFFFFF = -1.

## New Instructions

| Category | Instr | Op Code | Example | Meaning |
|---|---|---|---|---|
| Arithmetic (R & I format) | add unsigned | 0 and 21 | addu $s1, $s2, $s3 | $s1 = $s2 + $s3 |
| | sub unsigned | 0 and 23 | subu $s1, $s2, $s3 | $s1 = $s2 - $s3 |
| | add imm.unsigned | 9 | addiu $s1, $s2, 6 | $s1 = $s2 + 6 |
| Data Transfer | ld byte unsigned | 24 | lbu $s1, 20($s2) | $s1 = Mem($s2+20) |
| | ld half unsigned | 25 | lhu $s1, 20($s2) | $s1 = Mem($s2+20) |
| Cond. Branch (I & R format) | set on less than unsigned | 0 and 2b | sltu $s1, $s2, $s3 | if ($s2<$s3) $s1=1 else $s1=0 |
| | set on less than imm unsigned | b | sltiu $s1, $s2, 6 | if ($s2<6) $s1=1 else $s1=0 |

- Sign extend: `addi, addiu, slti`
- Zero extend: `andi, ori, xori`
- Overflow detected: `add, addi, sub`

# Binary Representations for Integers


Hey guys! How do you negate numbers?
In the early days of computing, designers made computers express numbers using **unsigned binary**.
And they were content...
42
00101010
Until there were negative numbers.
-5
?
What's negation?
To include negative numbers, designers came up with **sign magnitude**.
That took care of the negative numbers...
-5
10000101
But the computer had to count backwards for the negative numbers.
?
0
00000000?
10000000?
Set the sign bit!
Plus, this introduced positive and negative zero.
Then designers created **one's complement**.
Now computers only had to count in one direction...
-5
11111010
But there were still two zeroes!
?!
0
00000000?
11111111?
Flip the bits!
Finally, designers developed **two's complement**.
Now, there was only one zero...
0
00000000
And they were content.
Flip the bits and add 1!

CS Illustrated berkeley.edu

## Multiplication & Division

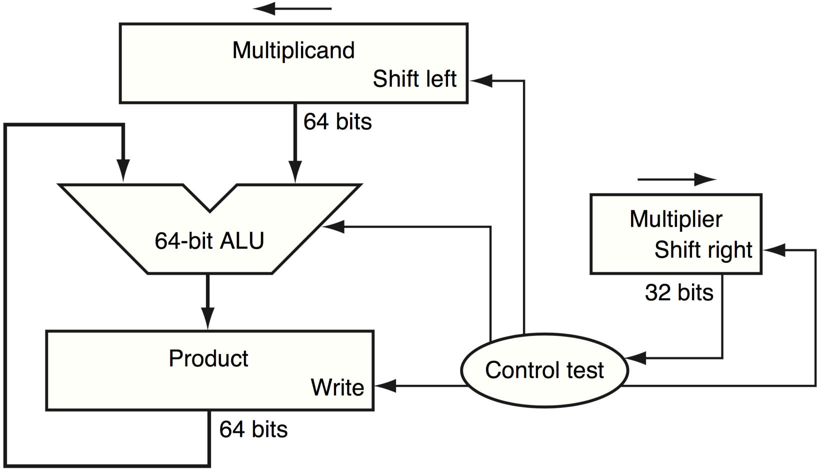
## Multiplication

- More complicated than addition
- Can be accomplished via shifting and adding

```
       0010    (multiplicand)
     x 1011    (multiplier)
      ------
      0010   }
     0010    }  (partial product
    0000     }    array)
   0010      }
  ---------
  0001 0110     (product)
```

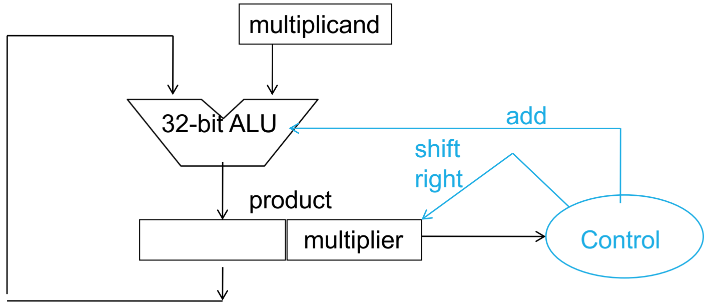
- Double precision product produced
- More time and more area to compute

## First Version of Multiplication Hardware



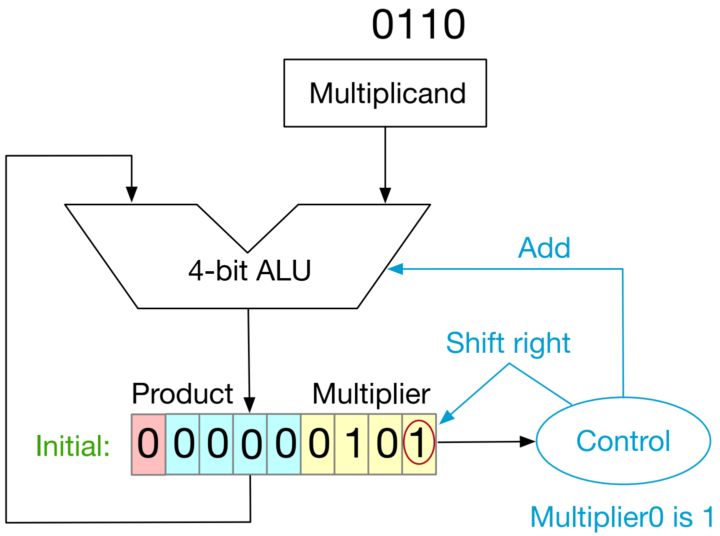
Note: n-bit $\times$ n-bit needs 2n-bit adder

## Second Version of Multiplication Hardware

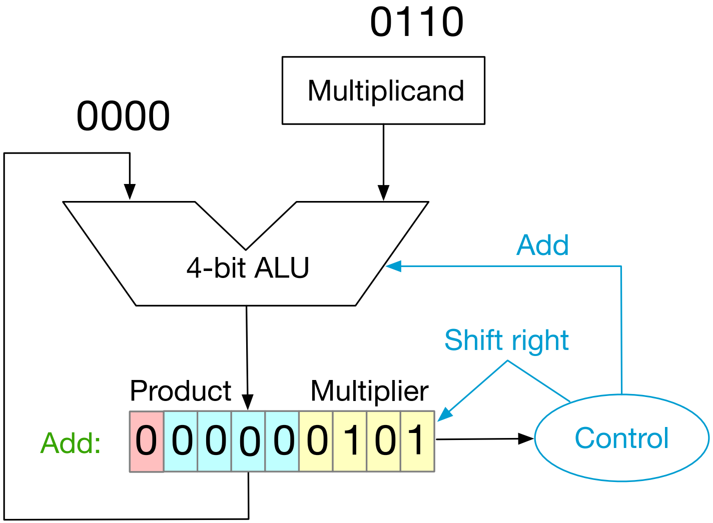


Note: n-bit × n-bit needs only n-bit adder

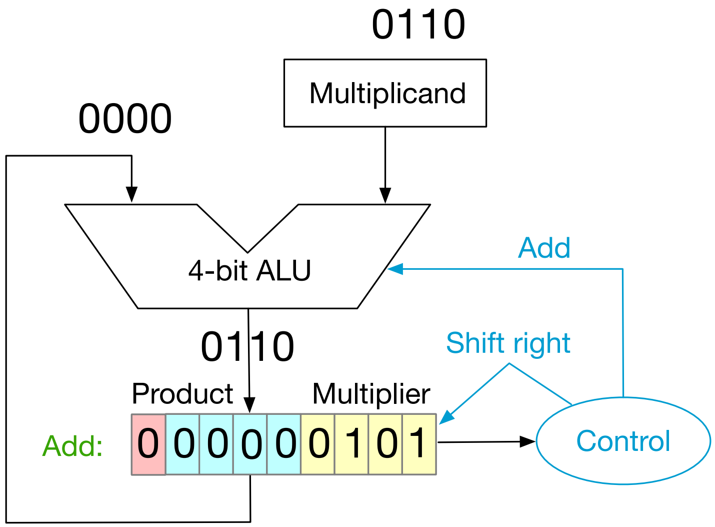
## Second Version: Example



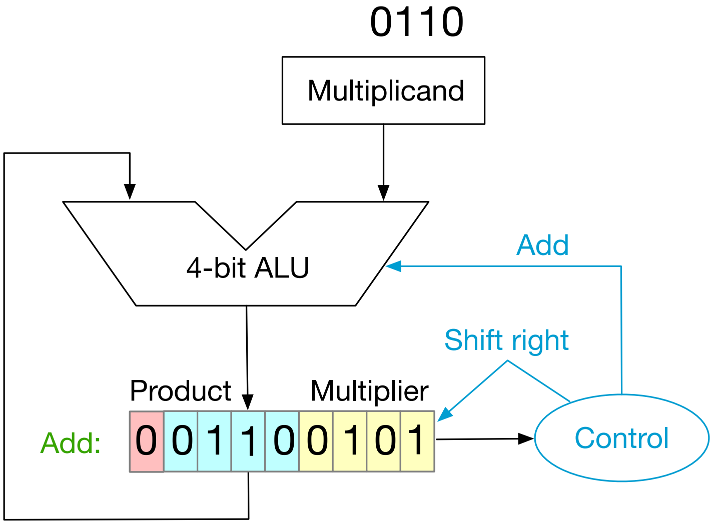
## Second Version: Example



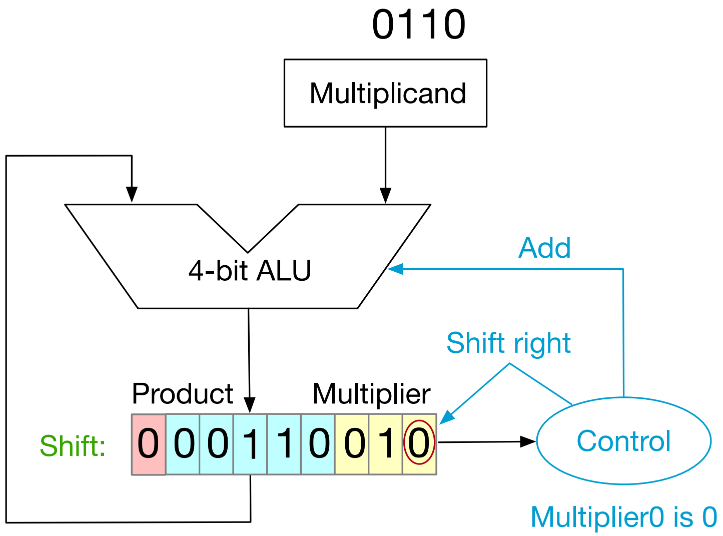
## Second Version: Example



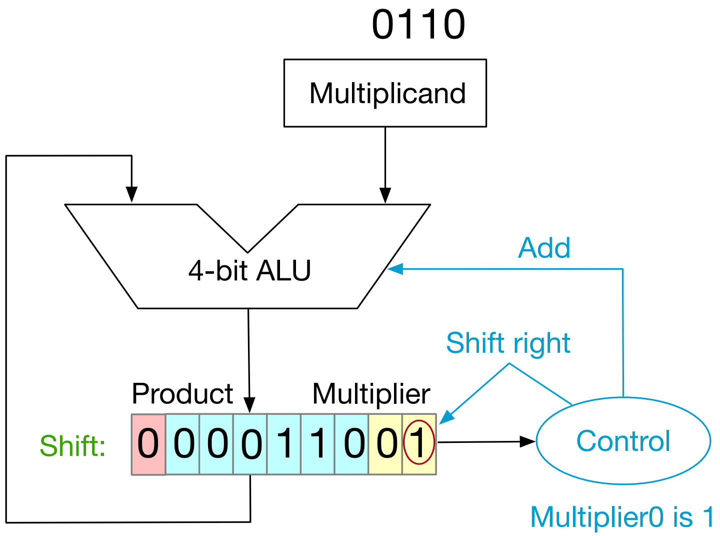
## Second Version: Example



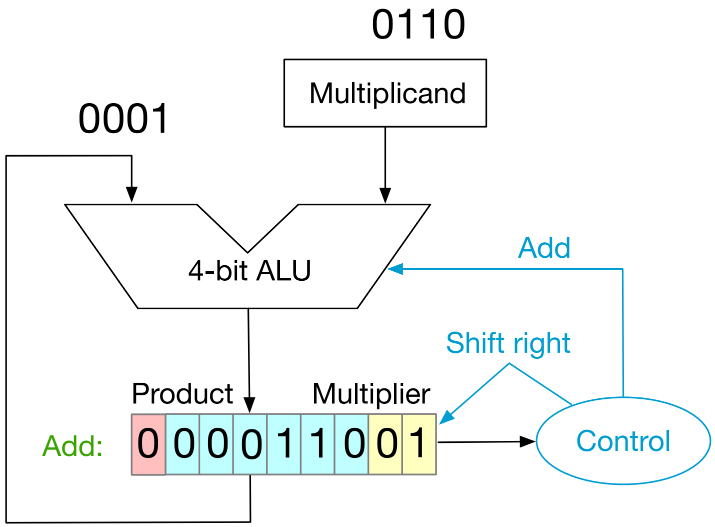
## Second Version: Example



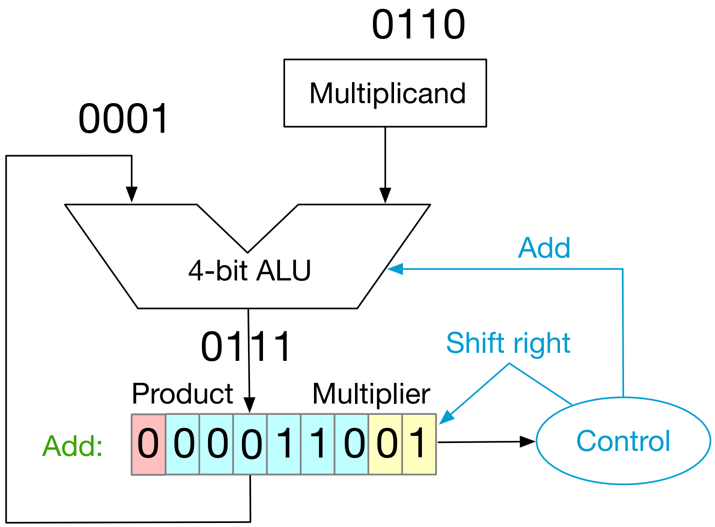
## Second Version: Example



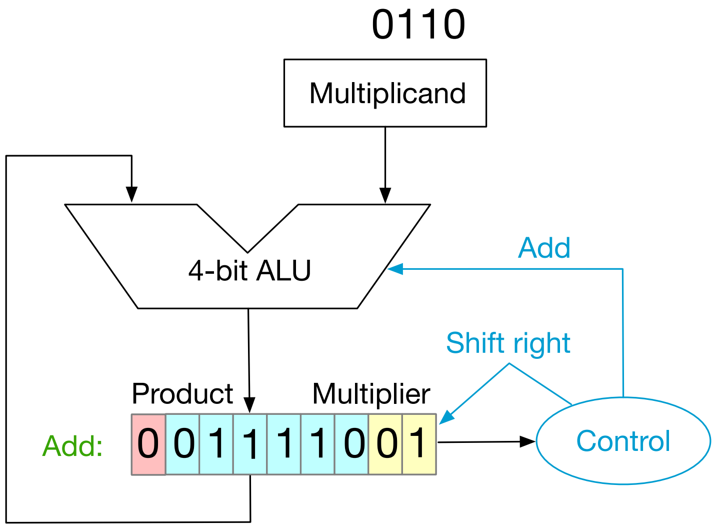
## Second Version: Example



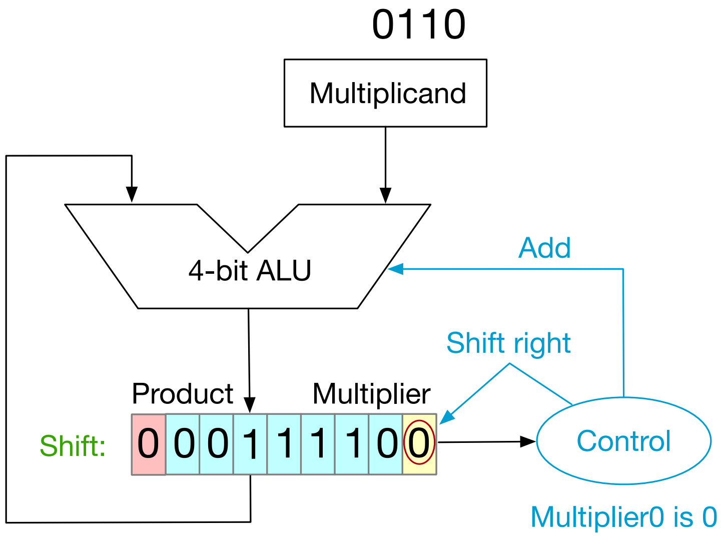
# Second Version: Example



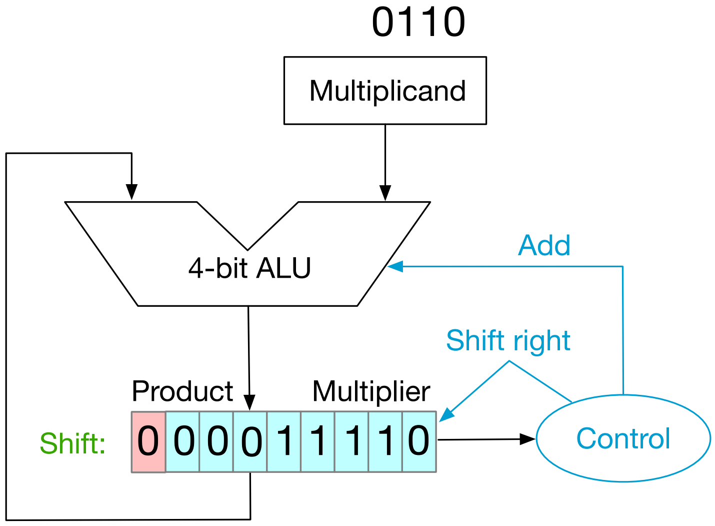
## Second Version: Example



## Second Version: Example



## Second Version: Example



Final Result: 00011110 = 30

## RISC-V Multiply Instruction

- `mul` performs an 32-bit $\times$ 32-bit multiplication and places the lower 32 bits in the destination register.

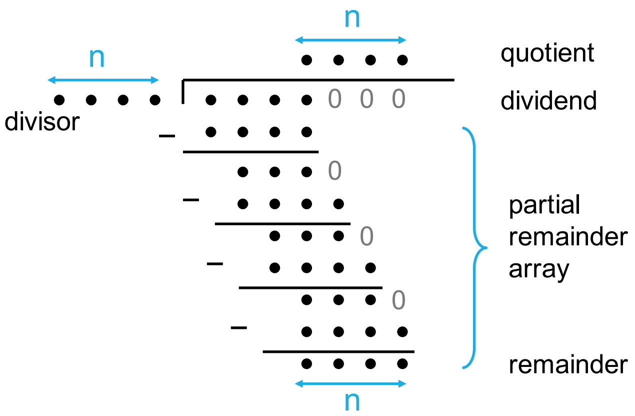
```
mul rd, rs1, rs2
```

| 31 … 25 | 24 … 20 | 19 … 15 | 14 … 12 | 11 … 7 | 6 … 0 |
|---|---|---|---|---|---|
| funct7 | rs2 | rs1 | funct3 | rd | opcode |
| 7 | 5 | 5 | 3 | 5 | 7 |

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

## Division

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



## Division Hardware

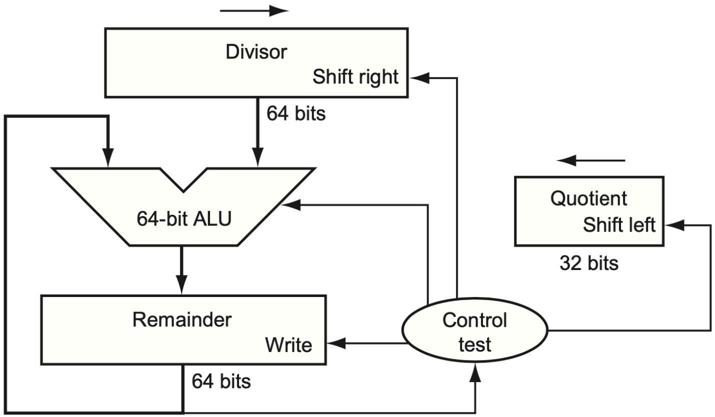


**FIGURE 3.8 First version of the division hardware.** The Divisor register, ALU, and Remainder register are all 64 bits wide, with only the Quotient register being 32 bits. The 32-bit divisor starts in the left half of the Divisor register and is shifted right 1 bit each iteration. The remainder is 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.

**Question: Division**

Dividing 1001010 by 1000

**Question**: Division

Dividing 1001010 by 1000

**Answer**: 1001 (remainder: 10)

# RISC-V Divide Instruction

- `div` generates the reminder in `hi` and the quotient in `lo`

```
div rd, rs1, rs2
```

| 31 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 |
|---|---|---|---|---|---|
| funct7 | rs2 | rs1 | funct3 | rd | opcode |
| 7 | 5 | 5 | 3 | 5 | 7 |

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

## Shifter

## Shift Operations

- Shifts by a constant are encoded as a specialization of the I-type format. The operand to be shifted is in `rs1`, and the shift amount is encoded in the lower 5 bits of the I-immediate field.

```
srli rd, rs1, imm[4:0]
srai rd, rs1, imm[4:0]
```

| | | | | | | |
|---|---|---|---|---|---|---|
| 31 26 | 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 |
| imm[11:6] | imm[5] | imm[4:0] | rs1 | funct3 | rd | opcode |
| 6 | 1 | 5 | 5 | 3 | 5 | 7 |

- `slli` is a logical left shift; `srli` is a logical right shift; and `srai`. is an arithmetic right shift.
- Logical shifts fill with zeros, arithmetic left shifts fill with the sign bit

**The shift operation is implemented by hardware separate from the ALU**

Using a barrel shifter, which would takes lots of gates in discrete logic, but is pretty easy to implement in VLSI

Assume that we have -16 in "t0". Which one is the result of "srai t1, t0, 4"?

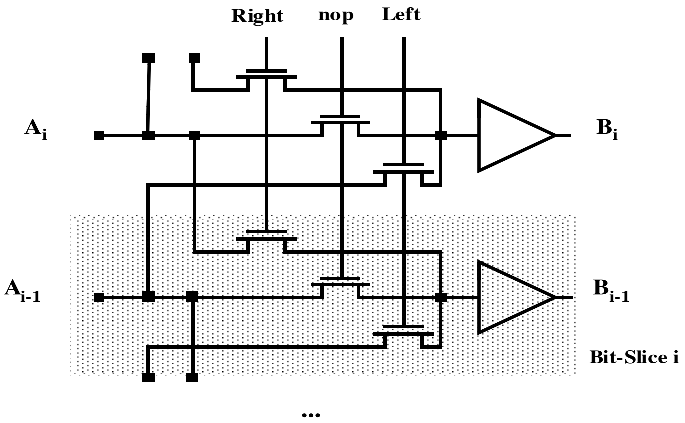
- A: 0x0FFFFFFE
- B: 4
- C: -4
- D: -1

Assume that we have -16 in "t0". Which one is the result of "srai t1, t0, 4"?

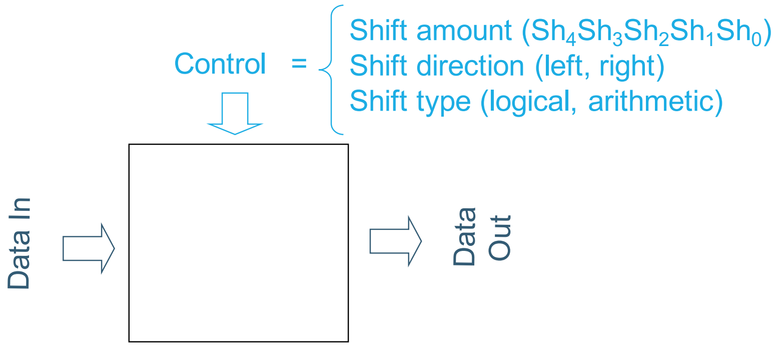
- A: 0x0FFFFFFE
- B: 4
- C: -4
- D: -1

Answer: D

## A Simple Shifter

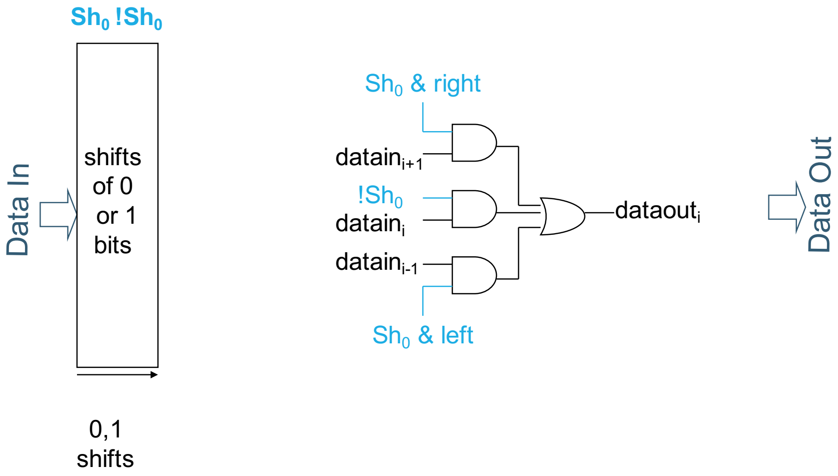


## Parallel Programmable Shifters

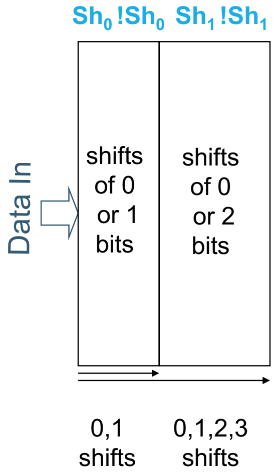


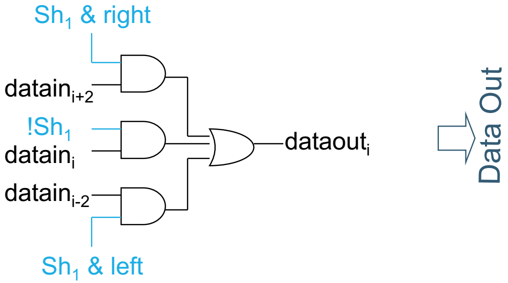
Control =
Shift amount ($Sh_4Sh_3Sh_2Sh_1Sh_0$)
Shift direction (left, right)
Shift type (logical, arithmetic)
Data In
Data Out

# Logarithmic Shifter Structure

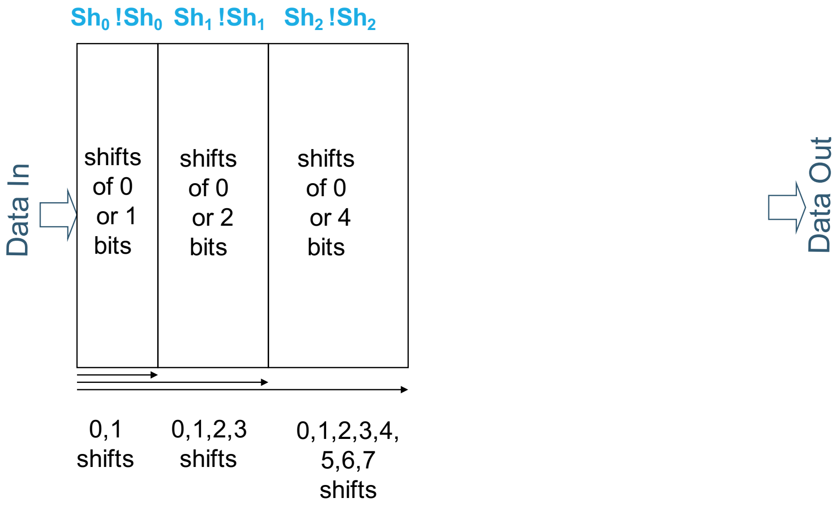


## Logarithmic Shifter Structure

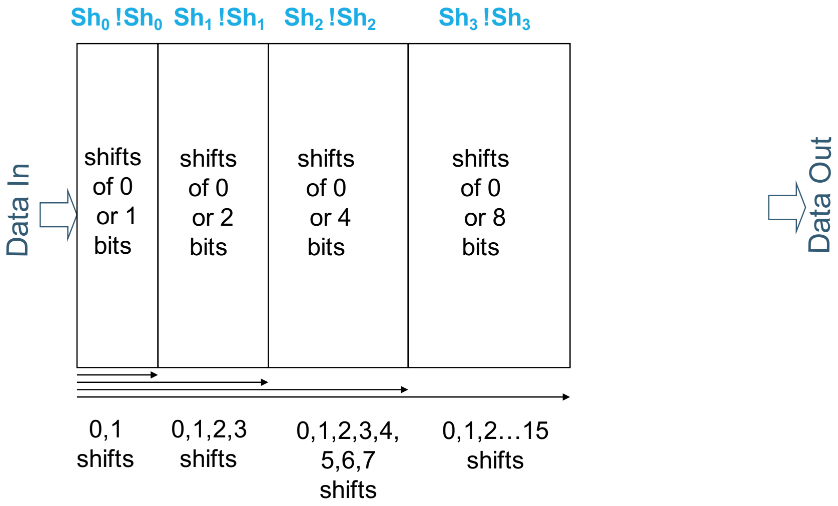




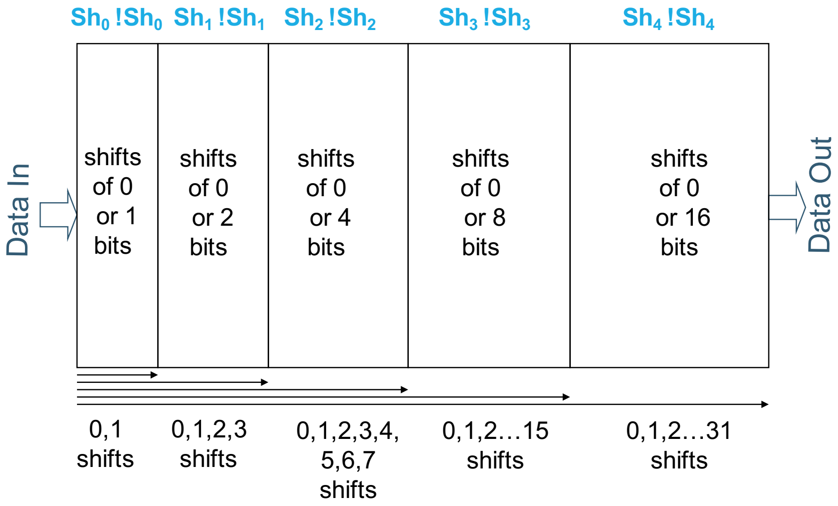
## Logarithmic Shifter Structure



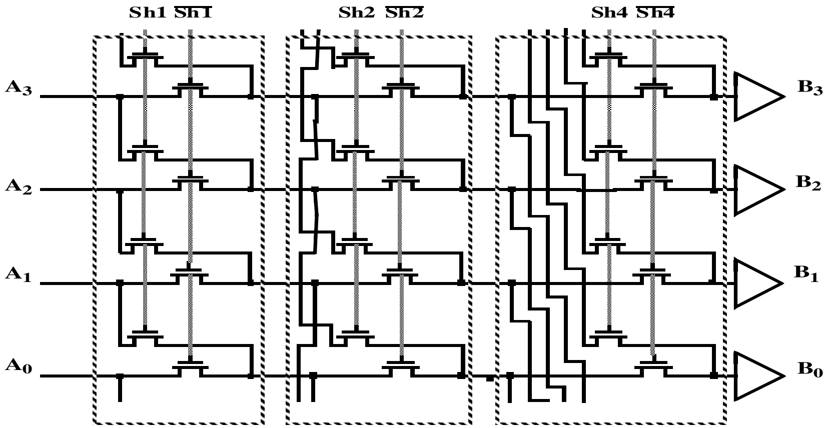
# Logarithmic Shifter Structure



## Logarithmic Shifter Structure



# Logarithmic Shifter Structure


Sh1 $\overline{\text{Sh1}}$
Sh2 $\overline{\text{Sh2}}$
Sh4 $\overline{\text{Sh4}}$
$A_3$
$A_2$
$A_1$
$A_0$
$B_3$
$B_2$
$B_1$
$B_0$