Arithmetic and Memory

Dov Kruger

Department of Electrical and Computer Engineering
Rutgers University

October 27, 2025

How do Computers do Arithmetic?

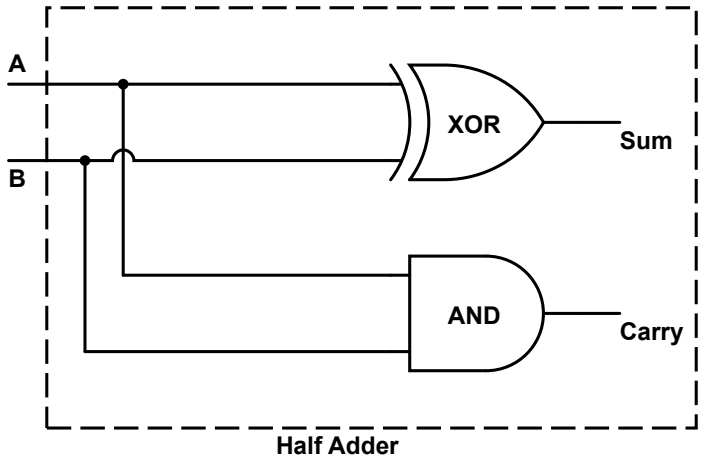
- This module describes how computers crunch numbers
- By the end you will be able to
  - Build a ripple carry adder
  - Calculate gate delays to know how long circuit takes
  - Design a faster carry look-ahead adder
  - Understand how multiplication works on a computer
  - Identify optimizations for multiplication and division (shift)
  - Learn what a flip-flop is and how it stores bits
  - Learn how to address memory

Terminology

- ALU: Arithmetic Logic Unit
- LSB: Least Significant Bit
- MSB: Most Significant Bit
- Overflow: When the result of an operation is too large to be represented in the available bits

Half Adder

- Half Adder: Adds two bits and outputs the sum and carry
- Inputs: A, B
- Outputs: Sum, Carry

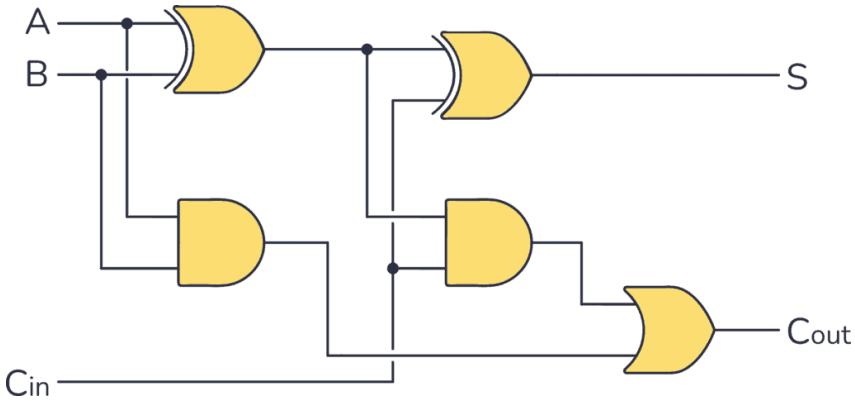


Full Adder

A	B	Cin	Sum	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

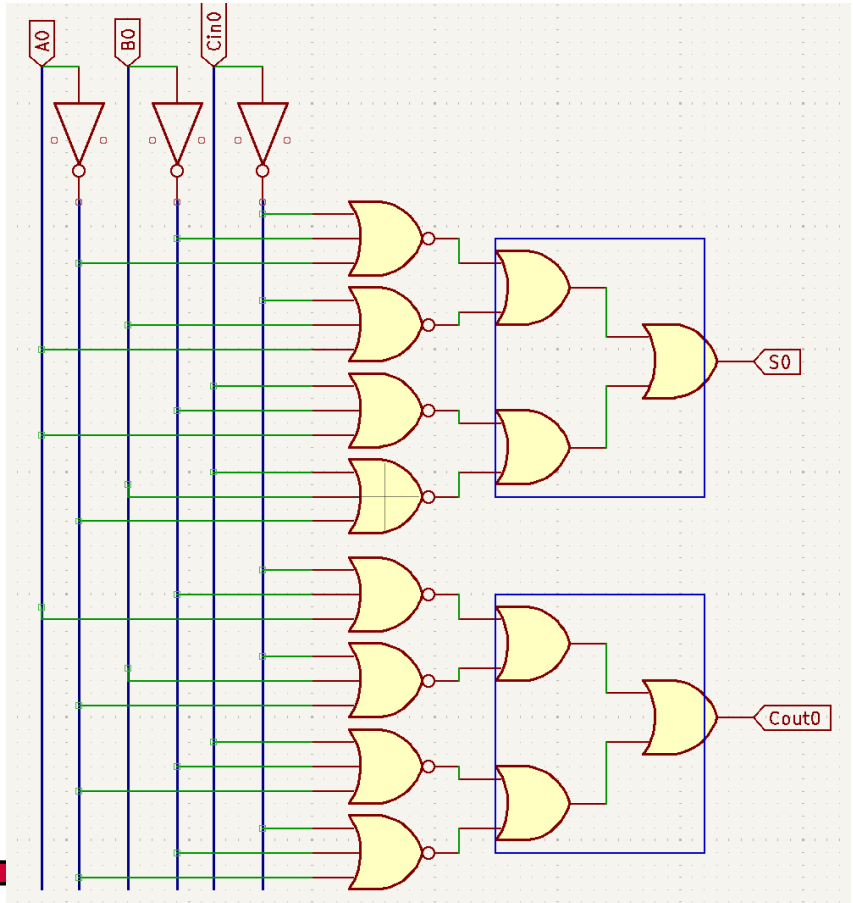
Full Adder

- Full Adder: Adds two bits and outputs the sum and carry
- Inputs: A, B, Carry
- Outputs: Sum, Carry
- Count the gate delays
- 74LS gate delays typically 10ns
- modern gates in a CPU 5-20ps?



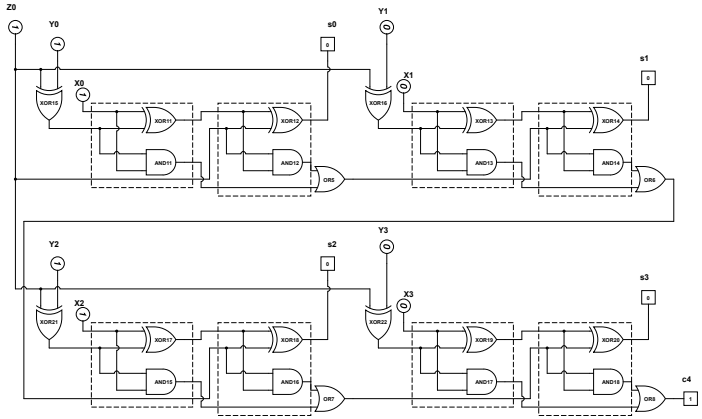
More Realistic Full Adder

- XOR doesn't really exist
- Do as much as possible in parallel (not entirely, my CAD skills are not good)



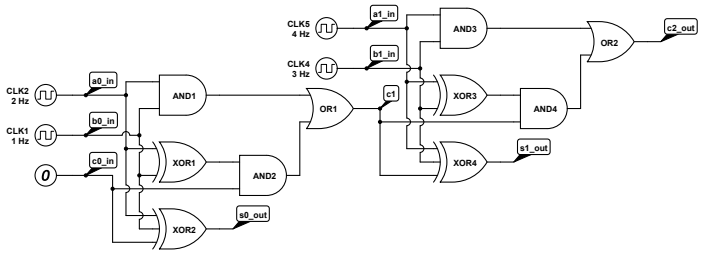
Ripple Carry Adder

- Ripple Carry Adder: Adds two numbers using full adders
- Inputs: A[3:0], B[3:0]
- Outputs: Sum[3:0], Carry
- Count the gate delays: 4



Carry Look-Ahead Adder

- Carry Look-Ahead Adder: Adds 4 bits and calculates the carry
- Inputs: A[3:0], B[3:0], Carry
- Outputs: Sum[3:0], Carry
- Count the gate delays: 2



When does Addition Overflow?

When the result is too large to be represented in the available bits

Unsigned

-
- Example: $1111 + 0001 = 0000$ (overflow)
- Example: $0111 + 0001 = 1000$ (no overflow)

Signed

-
- Example: $0111 + 0001 = 1000$ (-8, overflow)
- Example: $0011 + 0011 = 0110$ (6, no overflow)

Multiplication

- Multiplication is repeated addition and shifting
- Long Multiplication example

```
    64
    986
*   157
   6902
  49300
 986000
```

- Then add the results

Multiplication in Binary

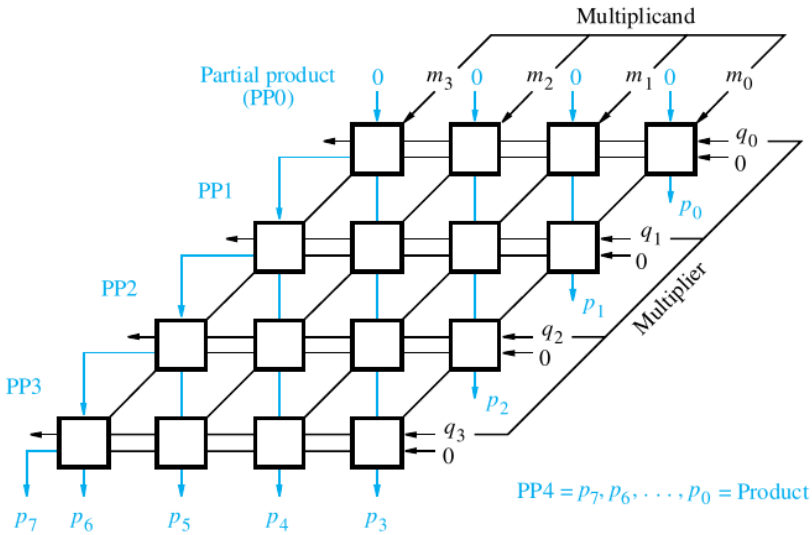
Notice: multiplying two n-bit numbers produces a 2n-bit number!

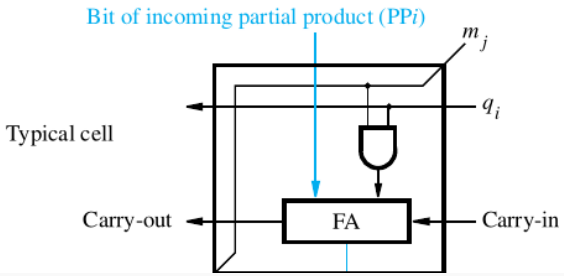
```
      01000001   (65)
  x   00110001   (49)

      01000001   (This is 65 shifted by 0)
  010000010000   (This is 65 shifted by 4)
 0100000100000   (This is 65 shifted by 5)
000011001110001
```

Multiplication Implemented as a loop

- Using a clock, early computers would repeatedly add and shift
- for n-bit multiplication, takes n clock cycles
- For CPUs without a multiply, it's a program
- Doing it in hardware in a single operation requires n^2 additions
- Faster than doing it in a loop, but still much slower than addition

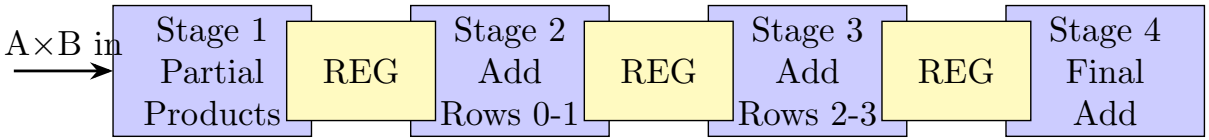


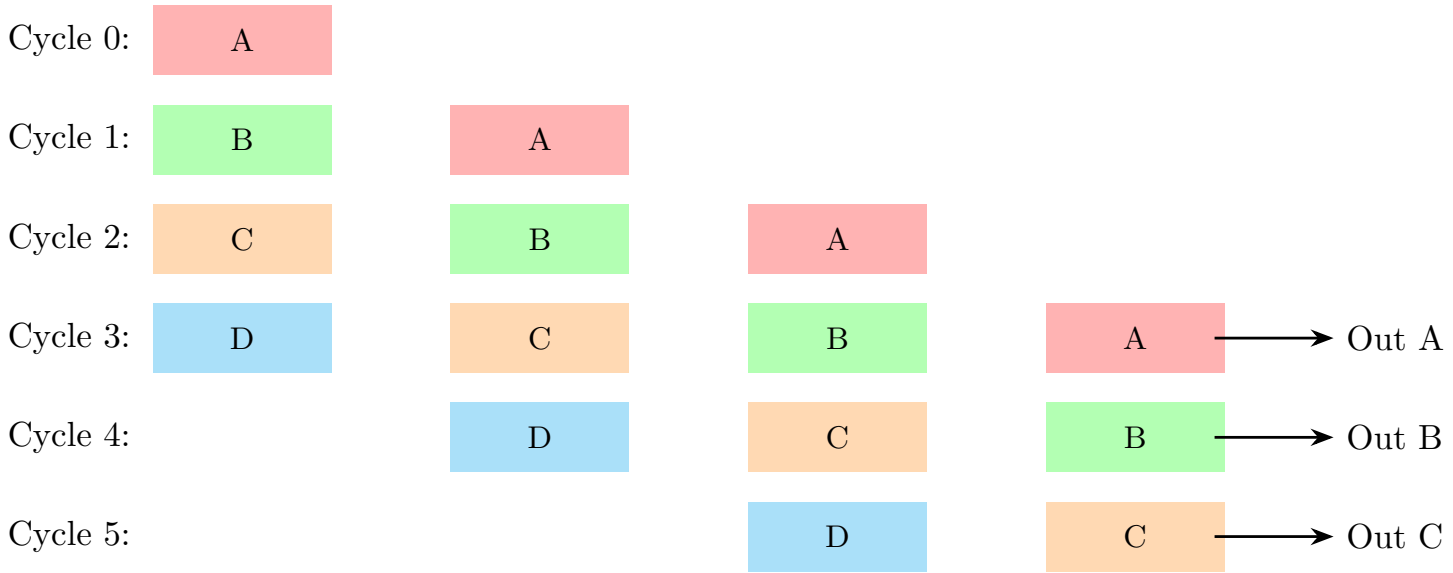


Pipelining

- Since multiplication is much slower than addition, can we go faster?
- For a single multiplication, not this way
- But if we have many numbers, we can make the multiplier even slower, but pipeline the circuit
- Example: AMD 7700X CPU can
  - latency: It takes 4 clock cycles to get the answer
  - multiply 1 number every clock
  - Two multiplier units, so it can do 2 multiplies every clock

Pipelining Multiplication

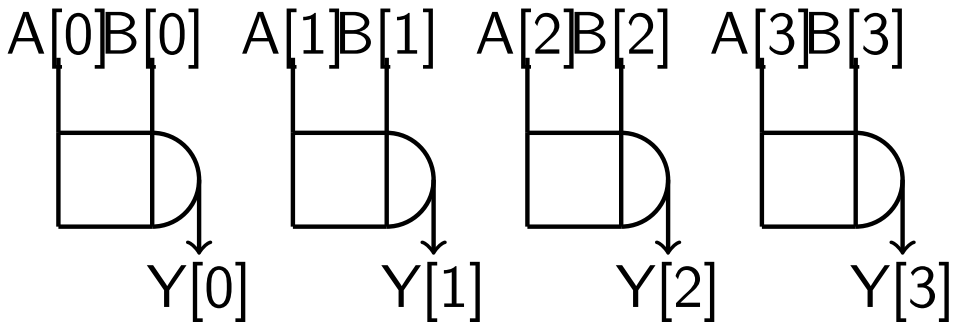




4-bit Example ALU Operations

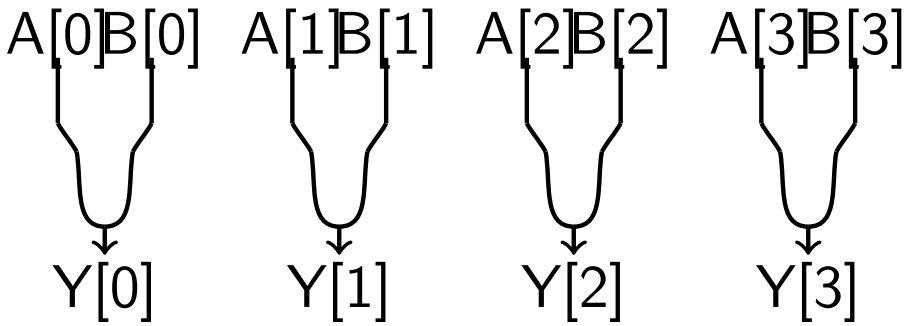
- All operations shown are typical of computer ALUs
- Each operation completes in a single clock cycle
- 4-bit examples generalize to 8, 16, 32, 64+ bits
- Operations include: logic, shifts, rotates, bit masking

ALU: Bitwise AND



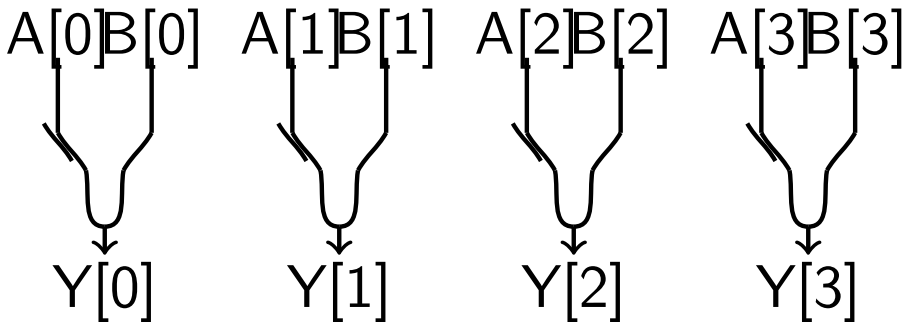
Example: `1010 AND 1100 = 1000`

ALU: Bitwise OR



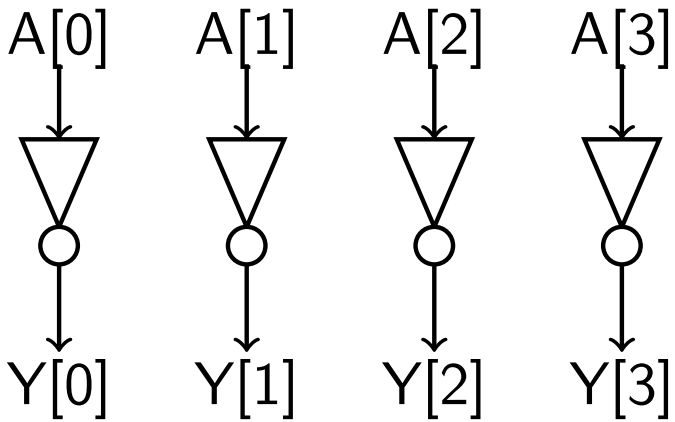
Example: `1010 OR 1100 = 1110`

ALU: Bitwise XOR



Example: `1010 XOR 1100 = 0110`

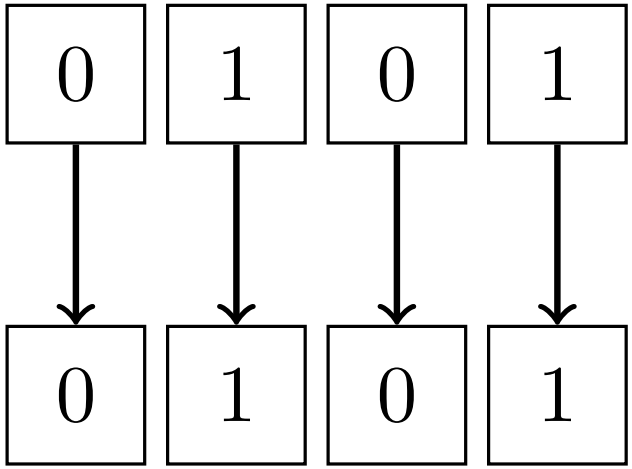
ALU: Bitwise NOT

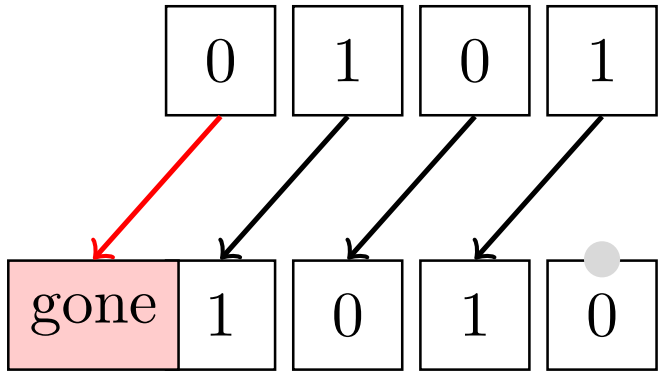


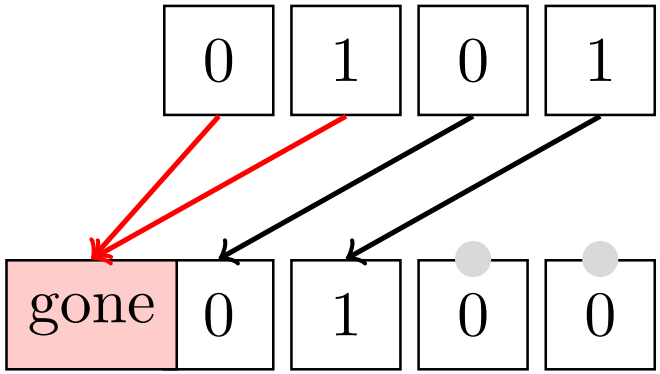
Example: `NOT 1010 = 0101`

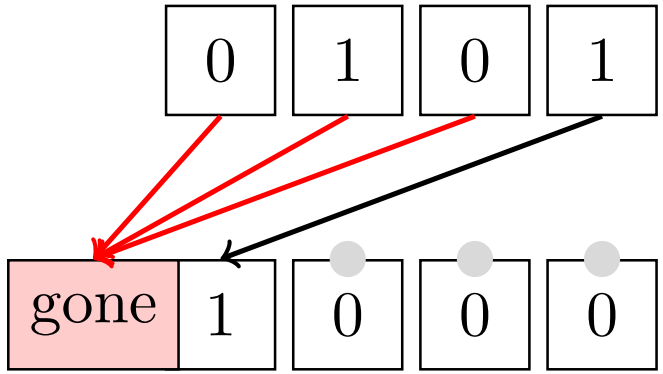
ALU: Logical Shift Left

Shifting left by n bits is equivalent to multiplying by 2^n



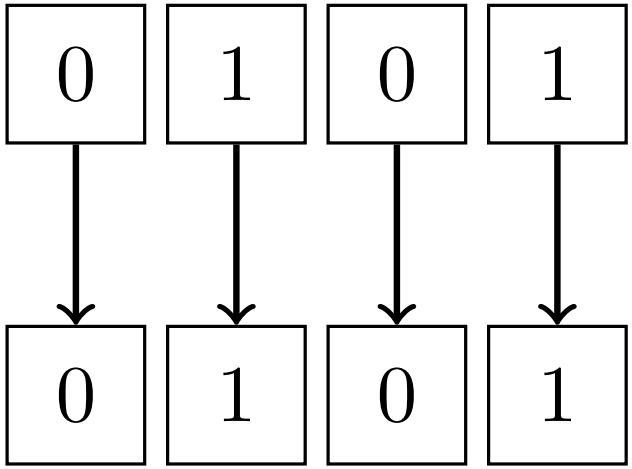


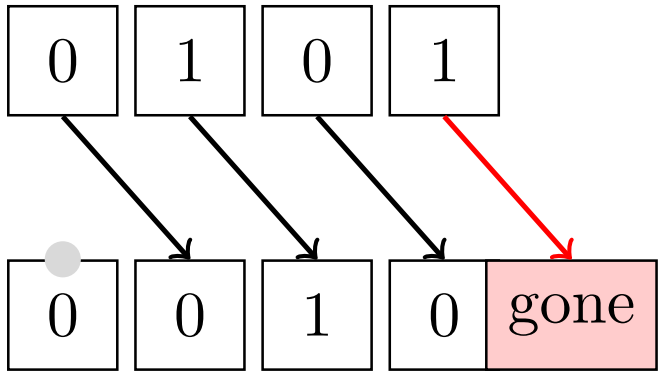


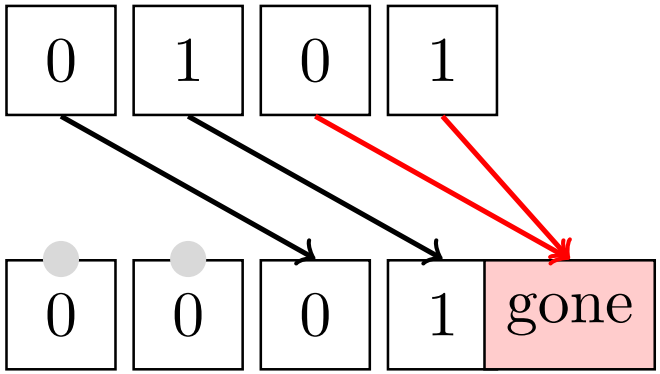


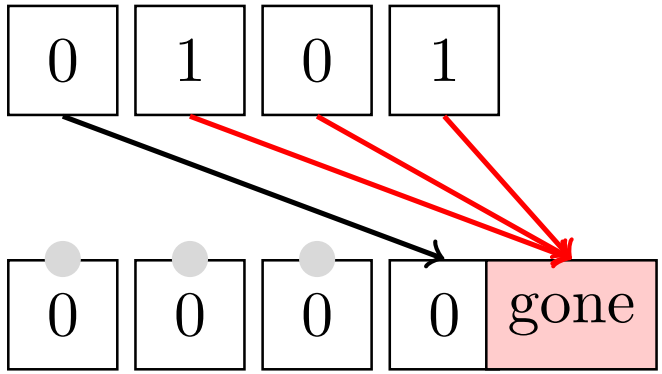
ALU: Logical Shift Right

Shifting right by n bits is equivalent to dividing by 2^n



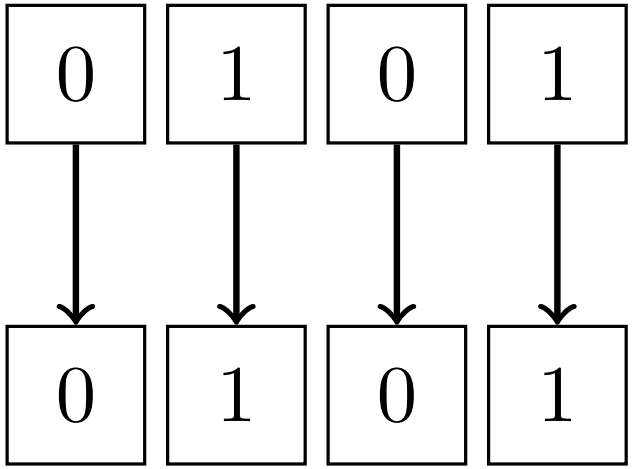


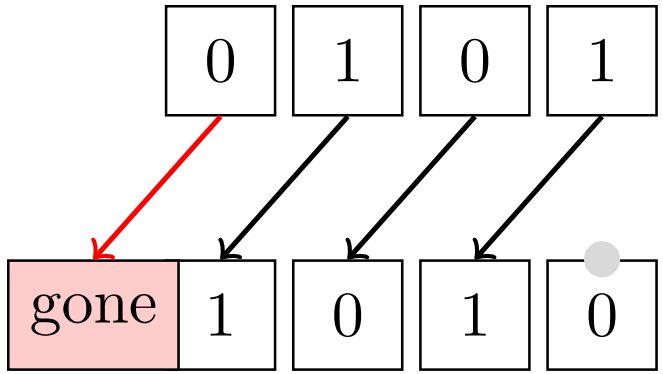


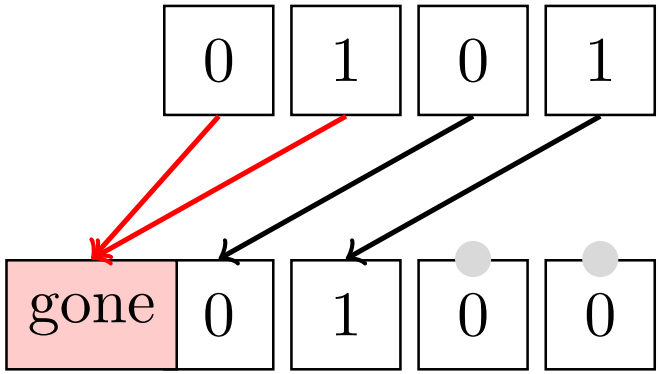


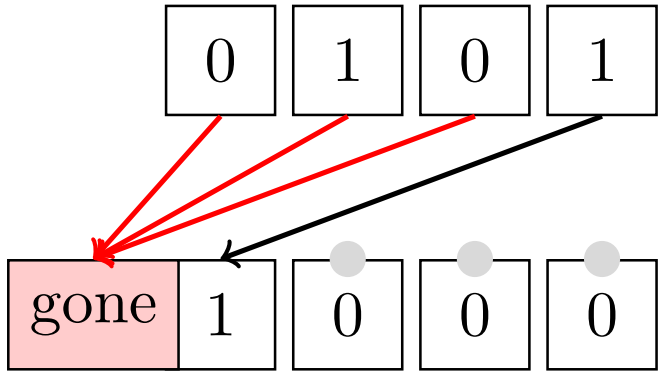
ALU: Arithmetic Shift Left

Shifting left by n bits is equivalent to multiplying by 2^n



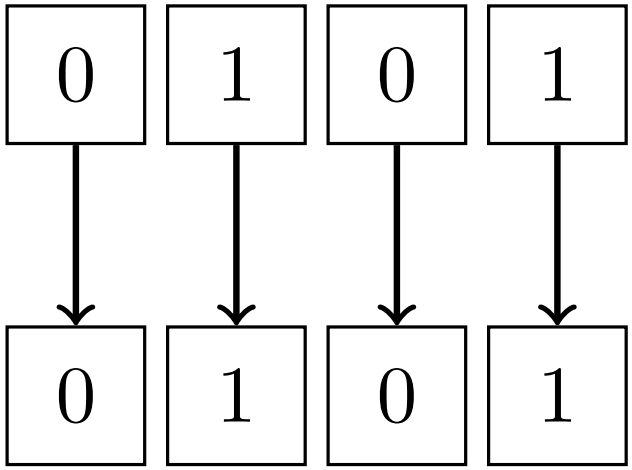


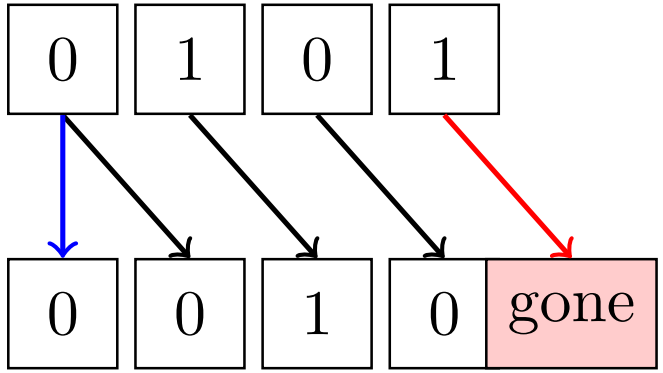


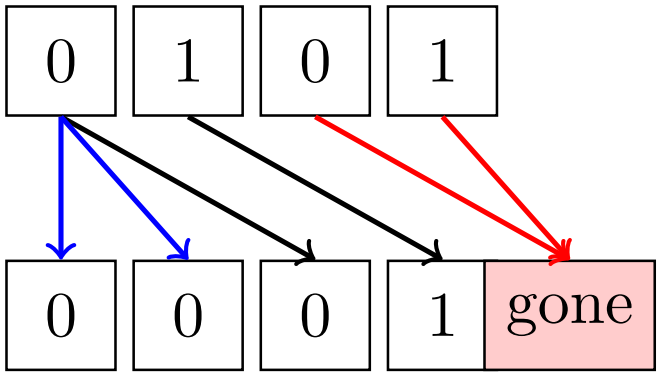


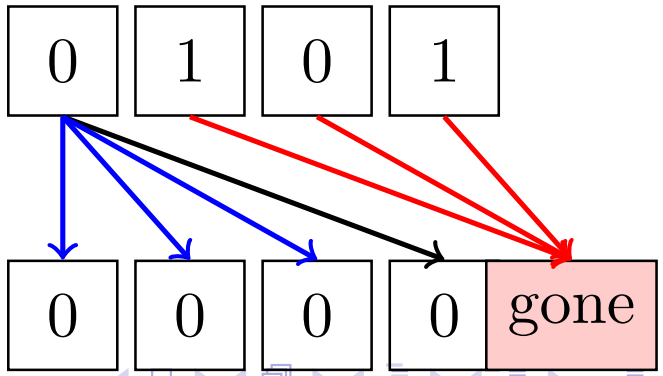
ALU: Arithmetic Shift Right

Shifting right by n bits is equivalent to dividing by 2^n
Sign bit is preserved (sign extension).

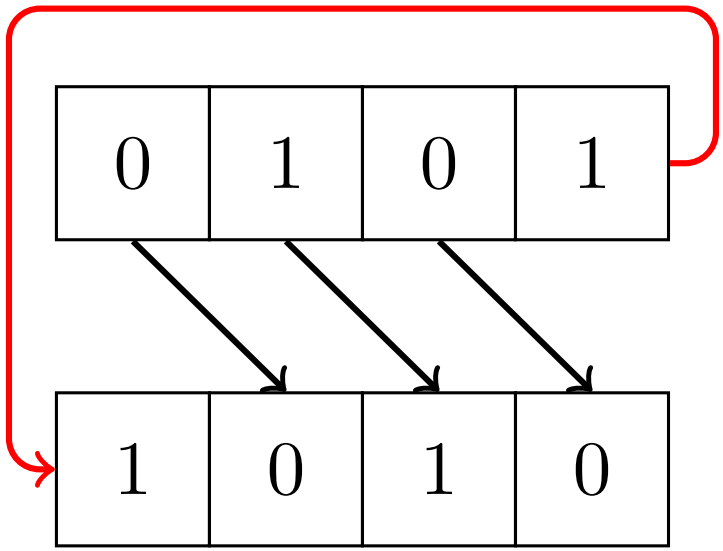






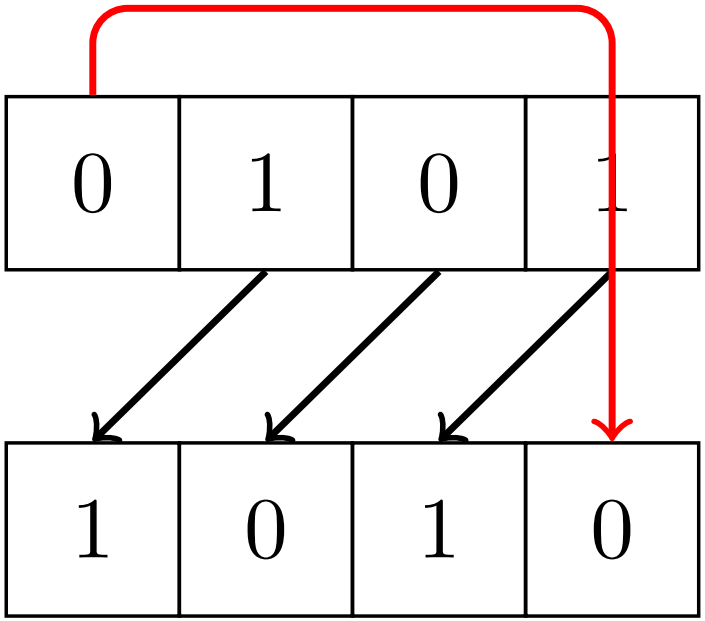


ALU: Rotate Left



MSB wraps around to LSB. No bits lost.

ALU: Rotate Right



LSB wraps around to MSB. No bits lost.

Rotate in C++ and in Verilog

- In the C family of languages there is no rotate operation.
- However, we can construct one

```
rotl(x, n, size) = (x << n) | (x >> (size - n))
rotr(x, n, size) = (x >> n) | (x << (size - n))
```

- In Verilog we can construct it using:

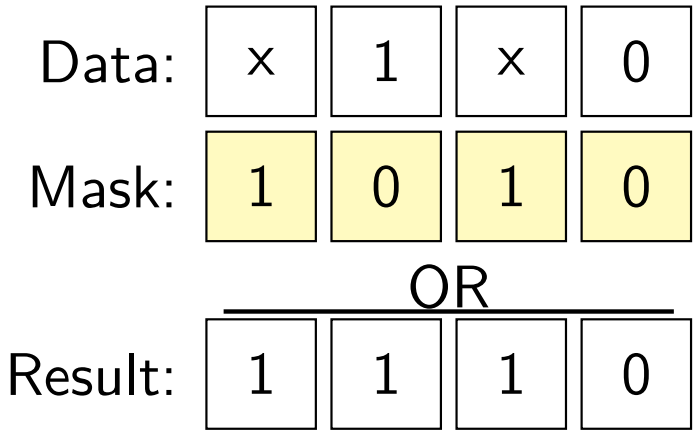
```
rotl(x, n, size) = {x[n-1:0], x[size-1:n]}
rotr(x, n, size) = {x[size-n-1:0], x[size-1:size-n]}
```

Verilog example to rotate an a left by 2, and b right by 3

```
logic [7:0] a, b;
assign a_rotl = {a[5:0], a[7:6]};
assign b_rotr = {b[2:0], b[7:3]};
```

Bit Masking: Set Bits with OR

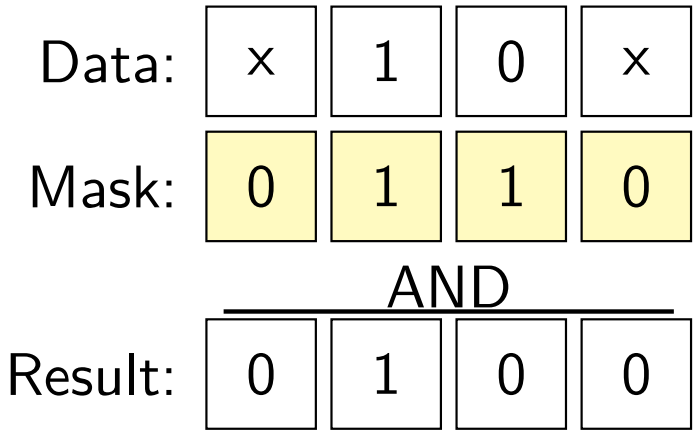
Set specific bits to 1 using OR with a mask.



Mask 1 bits force to 1; 0 bits pass through unchanged.

Bit Masking: Clear Bits with AND

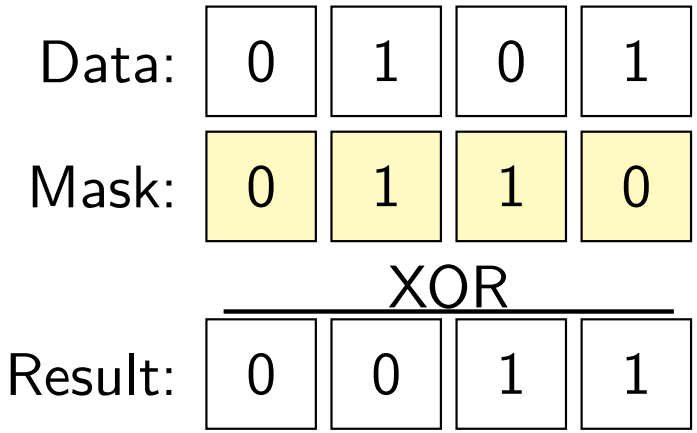
Clear specific bits to 0 using AND with inverted mask.



Mask 0 bits force output to 0; 1 bits pass through.

Bit Masking: Toggle Bits with XOR

Toggle specific bits using XOR with mask.



Mask 1 bits flip the corresponding data bit; 0 bits leave unchanged.

Bit Masking: Testing bits

To test if a bit is set, AND with 1 in that position

Bit Masking: Testing multiple Bits

To test if one of n bits are set, AND with 1 in those positions
Example: Test if any of the bits marked x are set

```
10001xxxx010100010
00000111000000000
```

First AND. All the rest of the bits turn to 0
if the number is not zero, then at least one of the bits is set
Example:
$Data = F32D, Mask = 0F00, DataANDMask = 0300$
Not zero, so at least one of the bits where the mask is 1 are set

Bit Masking: Testing multiple Bits

To test if EVERY one of n bits are set, AND with 1 in those positions
Example: Test if ALL of the bits marked x are set

```
10001xx010x1000x10
0000011000010000100
```

First AND. All the rest of the bits turn to 0
if the result $(AANDMASK) = MASK$ then ALL the bits are set

```
1000110010110001l0 AND
0000011000010000100
0000010000010000100 = DATA AND MASK
```

Example: DATA AND MASK $\neq$ MASK, so not all the bits are set

ALU Status Flags

- **Z (Zero):** Set if result = 0
  - Example: `0101 XOR 0101 = 0000` ⇒ Z=1
- **N (Negative):** Set if MSB = 1 (signed interpretation)
  - Example: `1010` ⇒ N=1 (= -6 in 4-bit signed)
- **V (Overflow):** Set if signed operation overflows
  - Example: `0111 + 0001 = 1000` ⇒ V=1 (7+1=-8 wrong!)
- **C (Carry):** Set if unsigned operation carries out
  - Example: `1111 + 0001 = 0000` with carry ⇒ C=1

How Does the ALU Work?

- Each circuit is a simple function of the inputs
- Combining them is a multiplexer
- We have to ignore all functions that aren't selected
- Two ways to do this
  - AND output with an enable signal for each function
  - Use a tri-state buffer to disable every function except the selected one

Simple Example: ALU Combining Two functions

Very simple ALU:

- OP = 0, A AND B
- OP = 1, A OR B

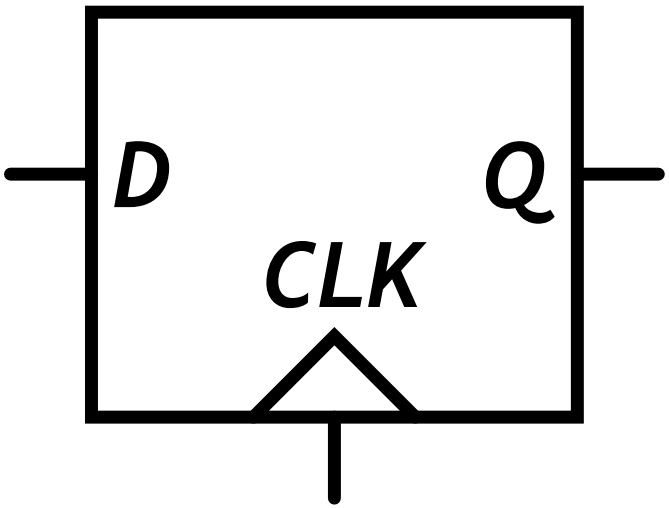
Simple ALU Using Tri-State Buffers

Latches, Flipflops and Registers

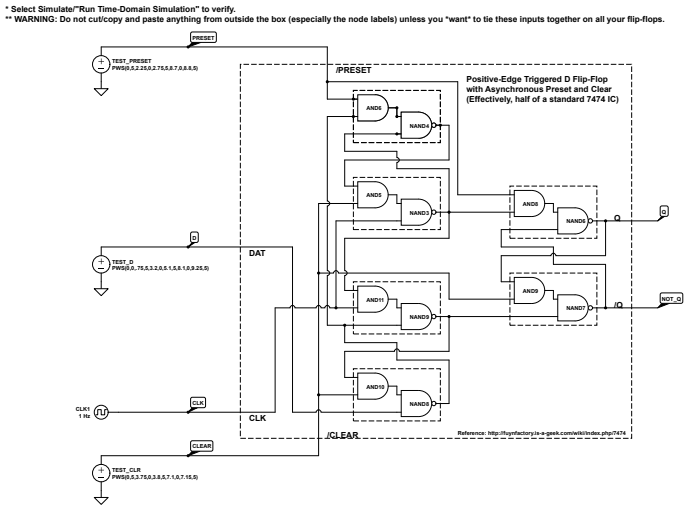
- Latches are level-triggered memory elements
- Flipflops are edge-triggered memory elements
- A D-type flipflop is the fundamental building block of memory
- Registers are collections of flipflops
- Registers are used to store data
- Registers are used to store instructions
- Registers are used to store addresses

D Type Flipflop

- D Type Flipflop: Stores a bit
- Inputs: D, Clock
- Outputs: Q
- Every time the clock ticks, data is stored

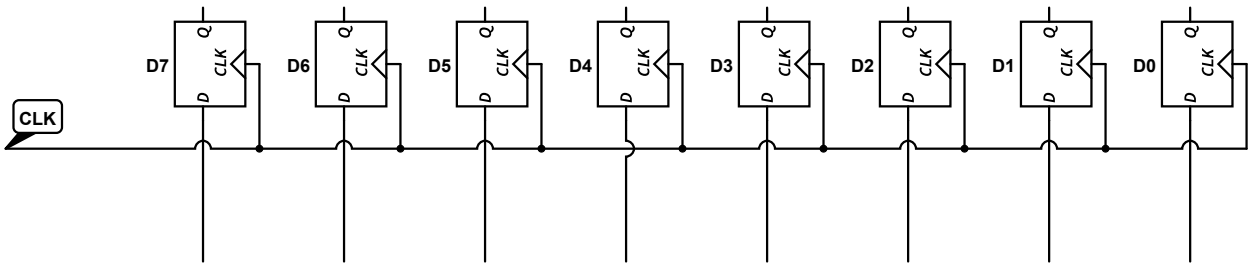


D Type Flipflop Implementation

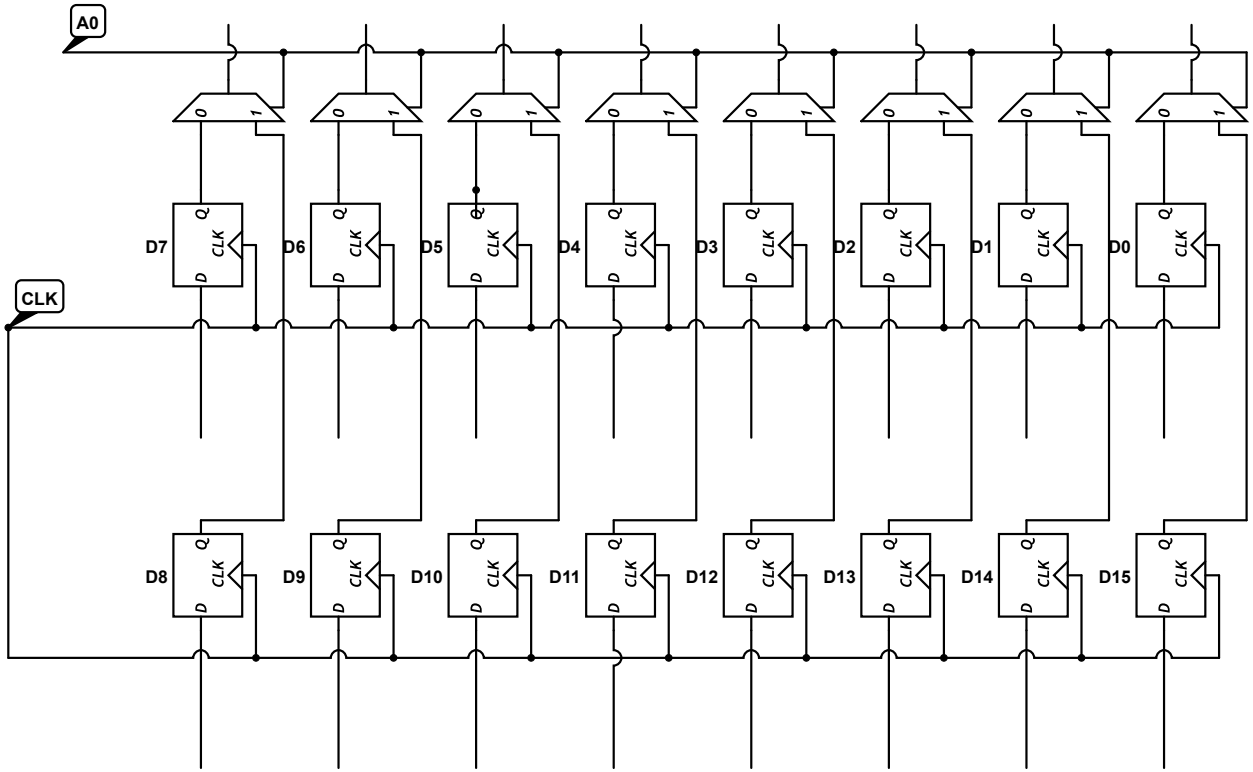


Building a Register

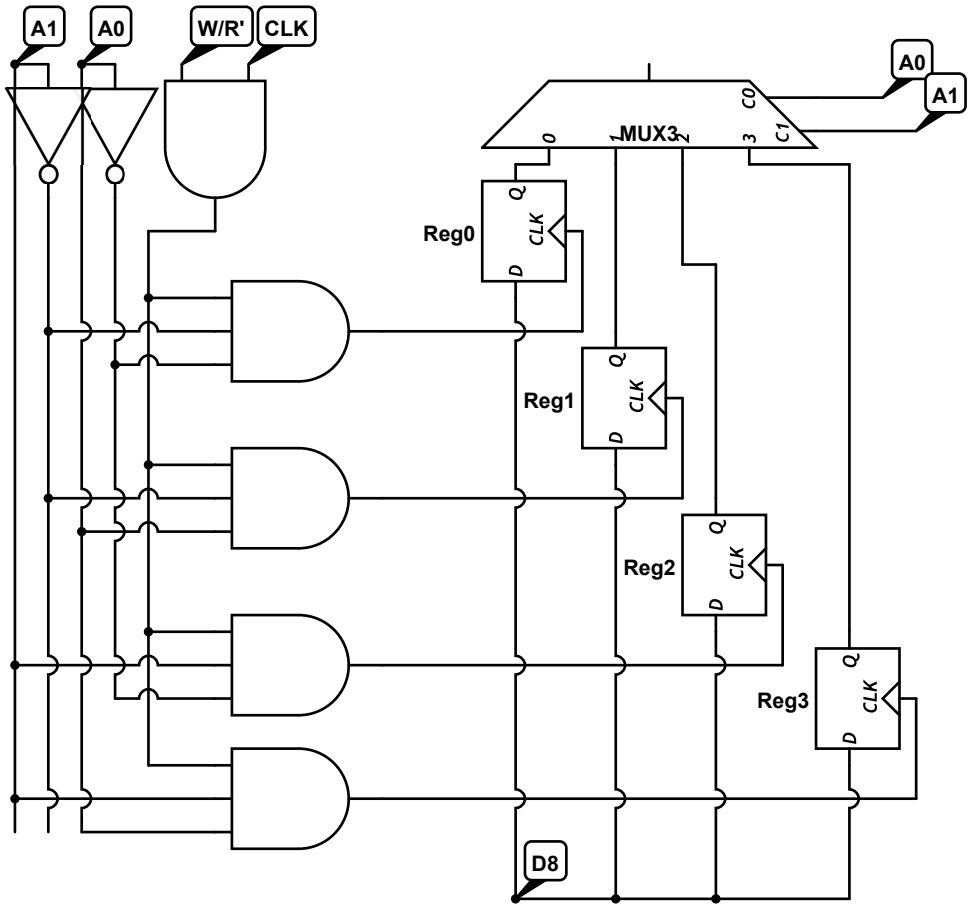
- Fastest computer memory is an array of flipflops
- Here is an 8-bit register



Simplified Addressing Read Example: 1 bit



Simplified Addressing Write: 1 bit



Simulating Registers in Verilog

- First approach is a case statement
- However, this will get unwieldy many registers

```
module register(
    input [1:0] addr,
    input [7:0] data,
    input write,
    output [7:0] out
);
```

Simulating Registers in Verilog

```
reg [7:0] a, b, c, d
always @(posedge clk) begin
    if (write) begin
        case (addr)
            2'b00: a <= data;
            2'b01: b <= data;
            2'b10: c <= data;
            2'b11: d <= data;
        endcase
    end
    case (address)
        2'b00: out = a;
        2'b01: out = b;
        2'b10: out = c;
        2'b11: out = d;
    endcase
end
```

Arrays in Verilog

- Arrays are ordered lists of variables

```
integer x; // one 32-bit integer
integer y[10]; // 10 integers, from y[0] to y[9
integer z[3:0]; // 4 integers, from z[0] to z[3
reg [7:0] r[3:0]; // 4 8-bit registers, r[0] to
```

Register File using Arrays

```
module register_file(
    input [1:0] addr,
    input [7:0] data,
    input write,
    input clk,
    output [7:0] out
);
    reg [7:0] registers [3:0];
    always @(posedge clk) begin
        if (write) begin
            registers[addr] <= data;
        end
        out <= registers[addr];
    end
endmodule
```