Arithmetic and Memory

Dov Kruger

Department of Electrical and Computer Engineering
Rutgers University

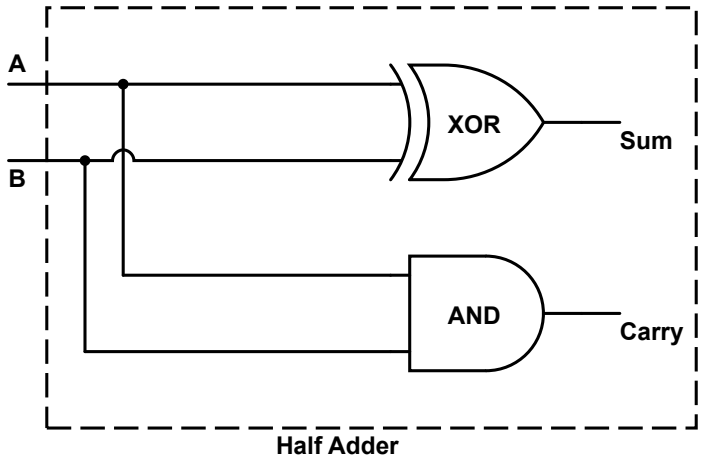
October 17, 2024

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

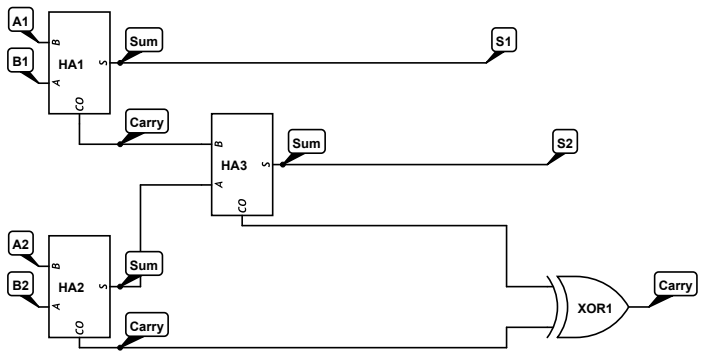
Half Adder

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



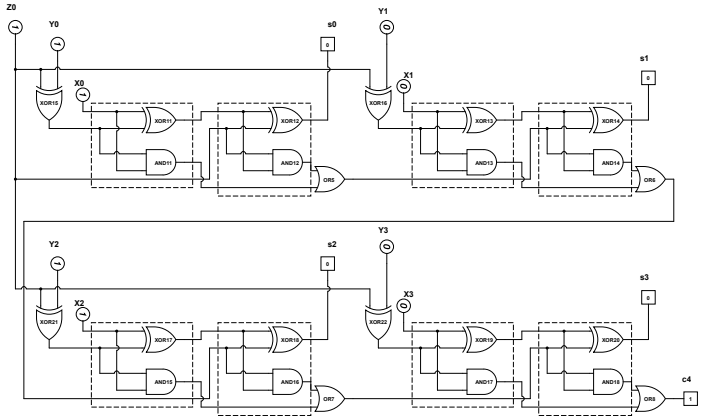
Full Adder

- Full Adder: Adds two bits and outputs the sum and carry
- Inputs: A, B, Carry
- Outputs: Sum, Carry
- Count the gate delays: 2
- 74LS gate delays typically 10ns
- modern gates in a CPU 10-20ps?



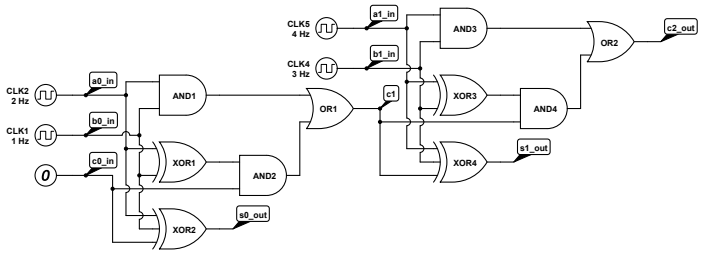
Ripple Carry Adder

- 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



Multiplication

- Multiplication is repeated addition and shifting
- Long Multiplication example

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

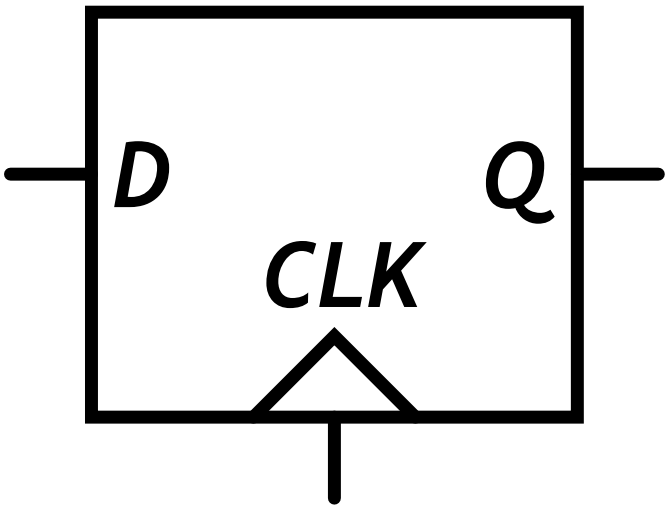
- Then add the results

Multiplication in Binary

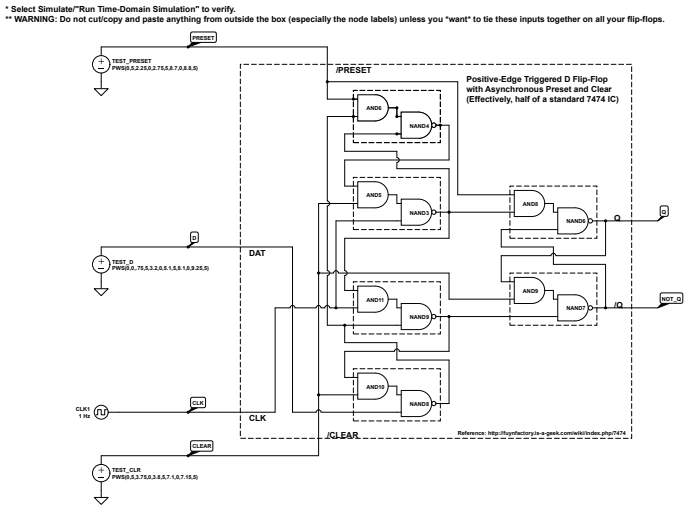
```
        01000001   (65)
    x   00110001   (49)

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

D Type Flipflop

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

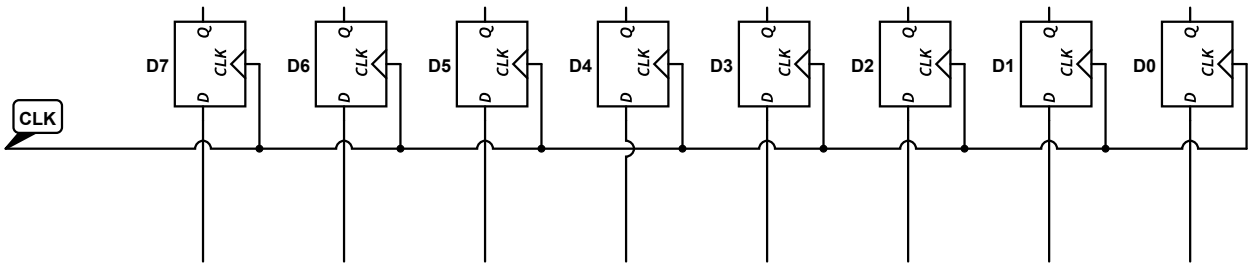


D Type Flipflop Implementation

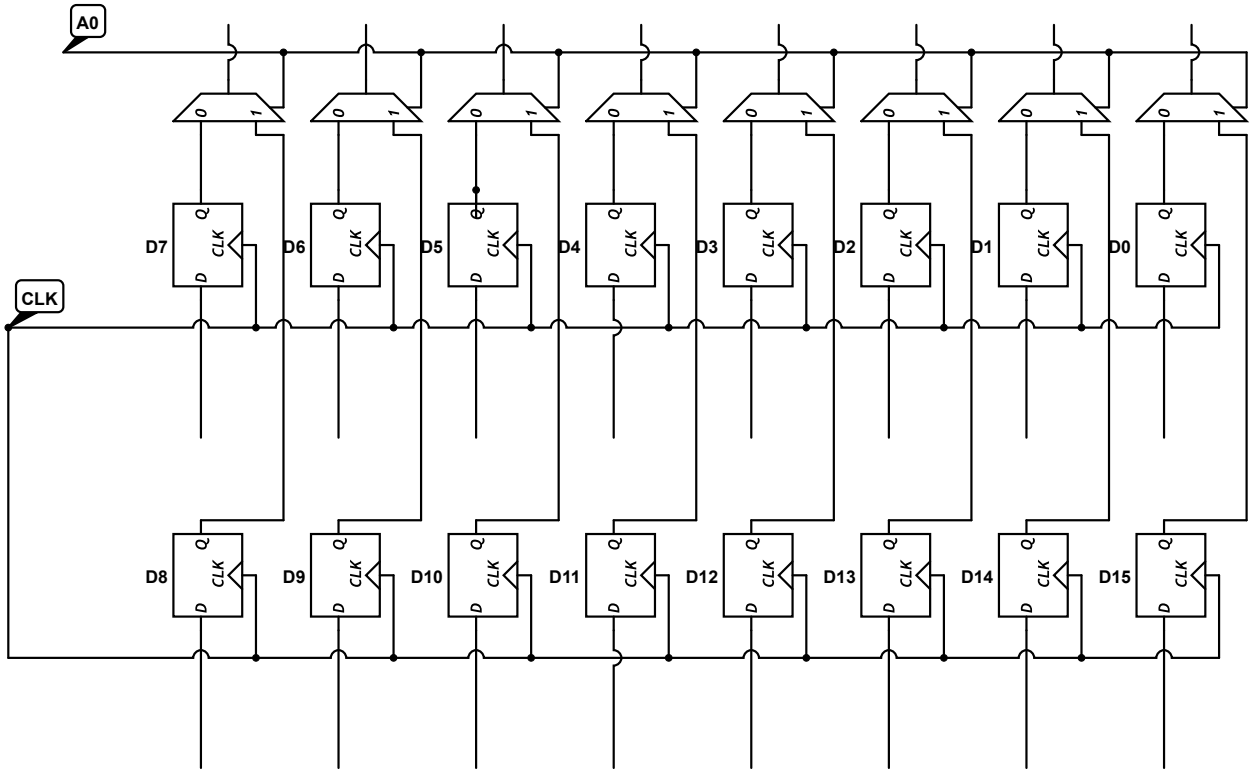


Building a Register

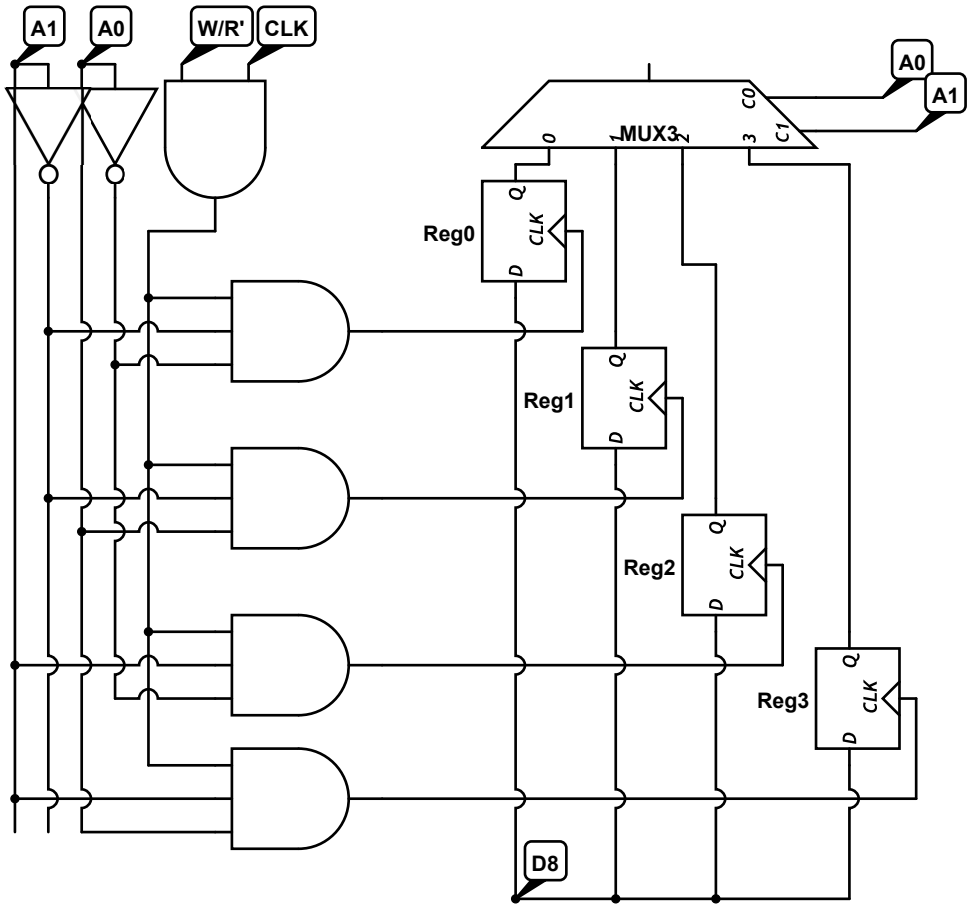
- Fastest computer memory is an array of flipflops
- Here is an 8-bit register



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
```