Verilog

RENZYM
Expanding Horizons

Overview

- Logic Design review
  - Basic Building Blocks
  - Design Examples: Incrementer, Fifo
- Verilog: Coding Basic blocks
  - Use of Modules
  - Gates, Mux/Demux, Registers
  - Data Values
  - Counter Example

Logic Design Review

Register Transfer Level (RTL) Design

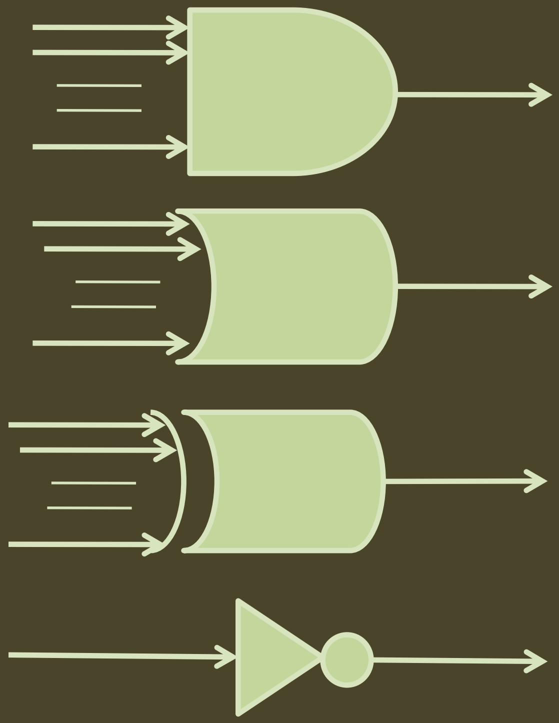
Logic Design

Basic building blocks for Logic design are

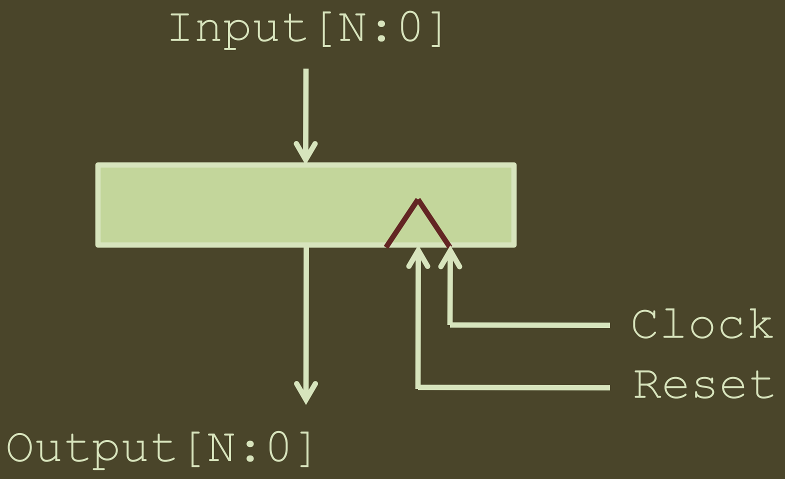
- Gates
- Muxes/Demuxes
- Registers/Memory
- Arithmetic Operations (Adder, Subtracter, Multipliers etc.)
- State machines
- Any combinations of Above

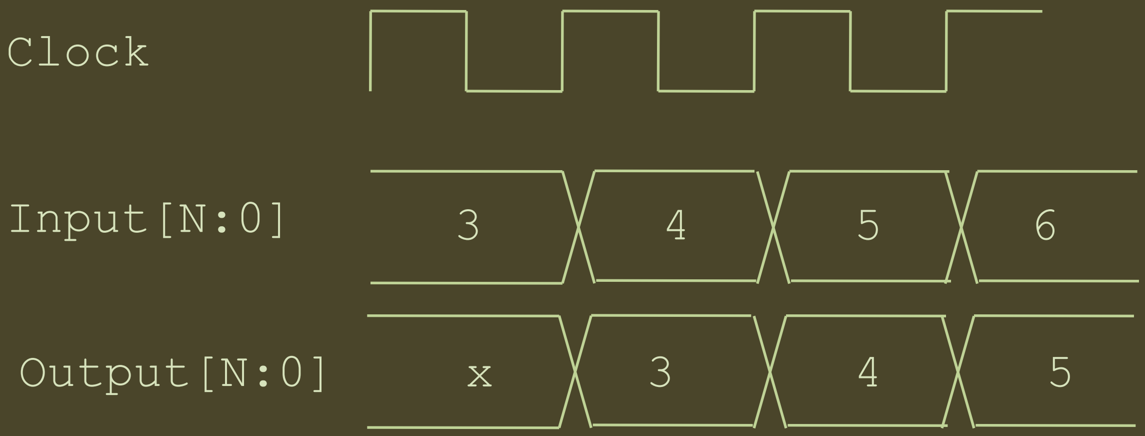
Gates

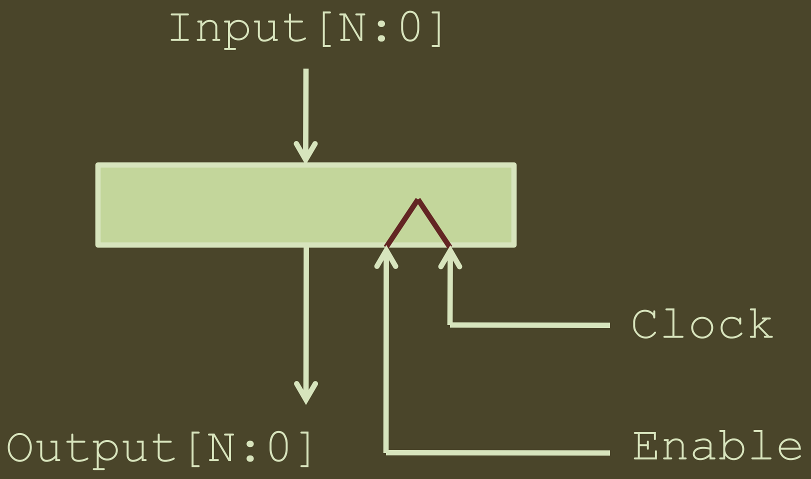
Inputs

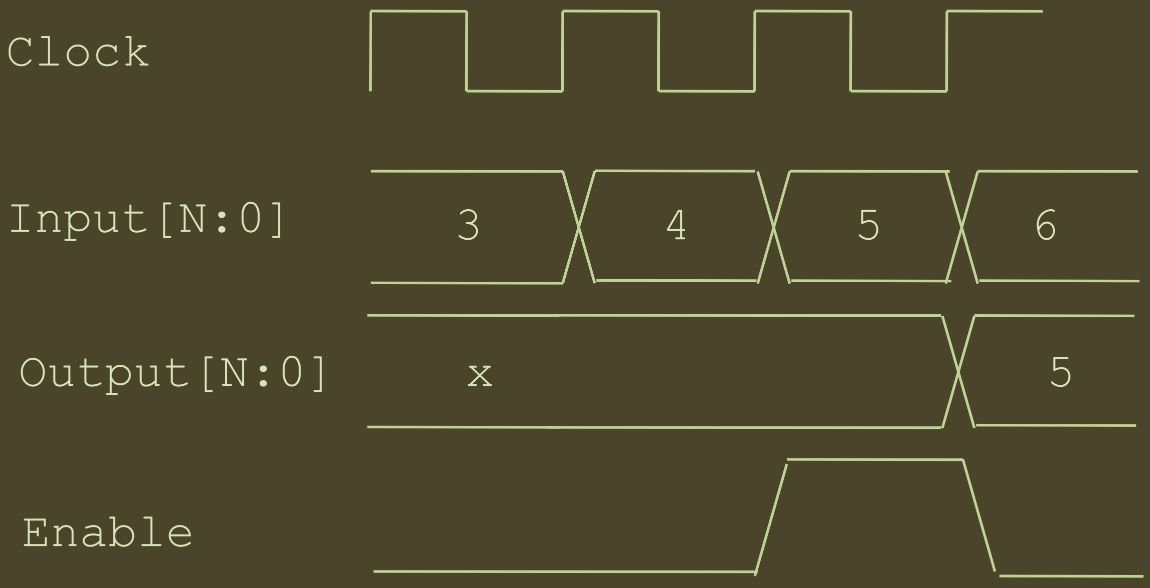


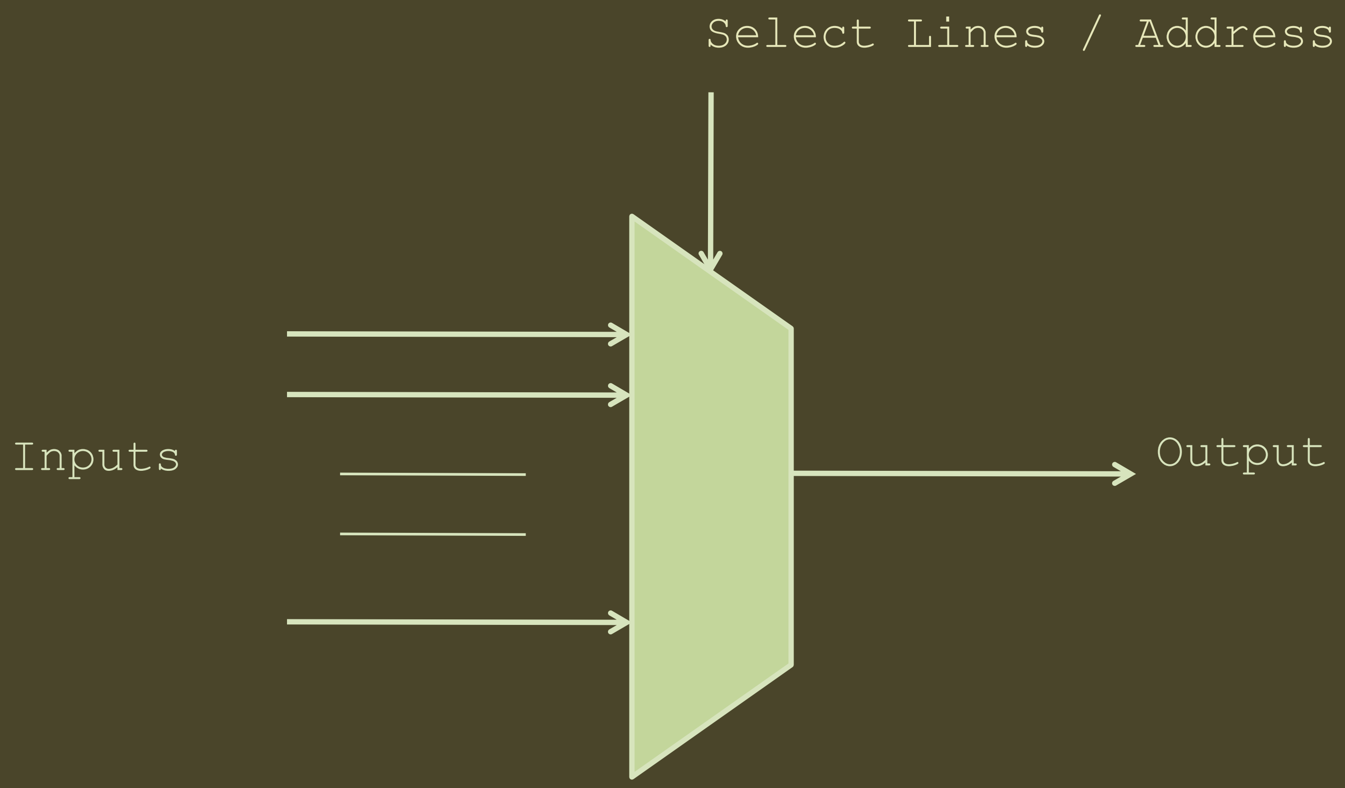
Outputs

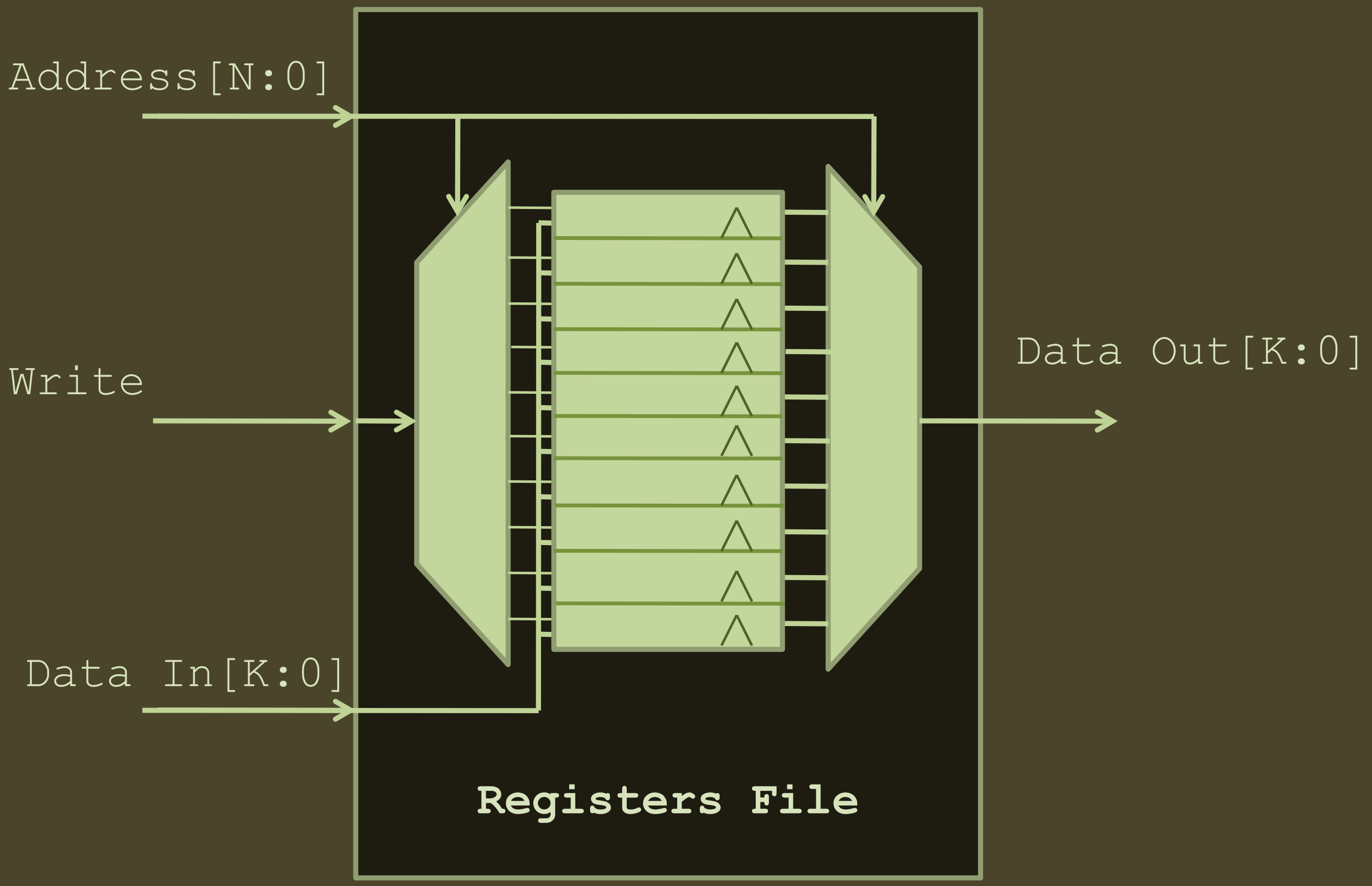

Input[N:0]
Clock
Reset
Output[N:0]

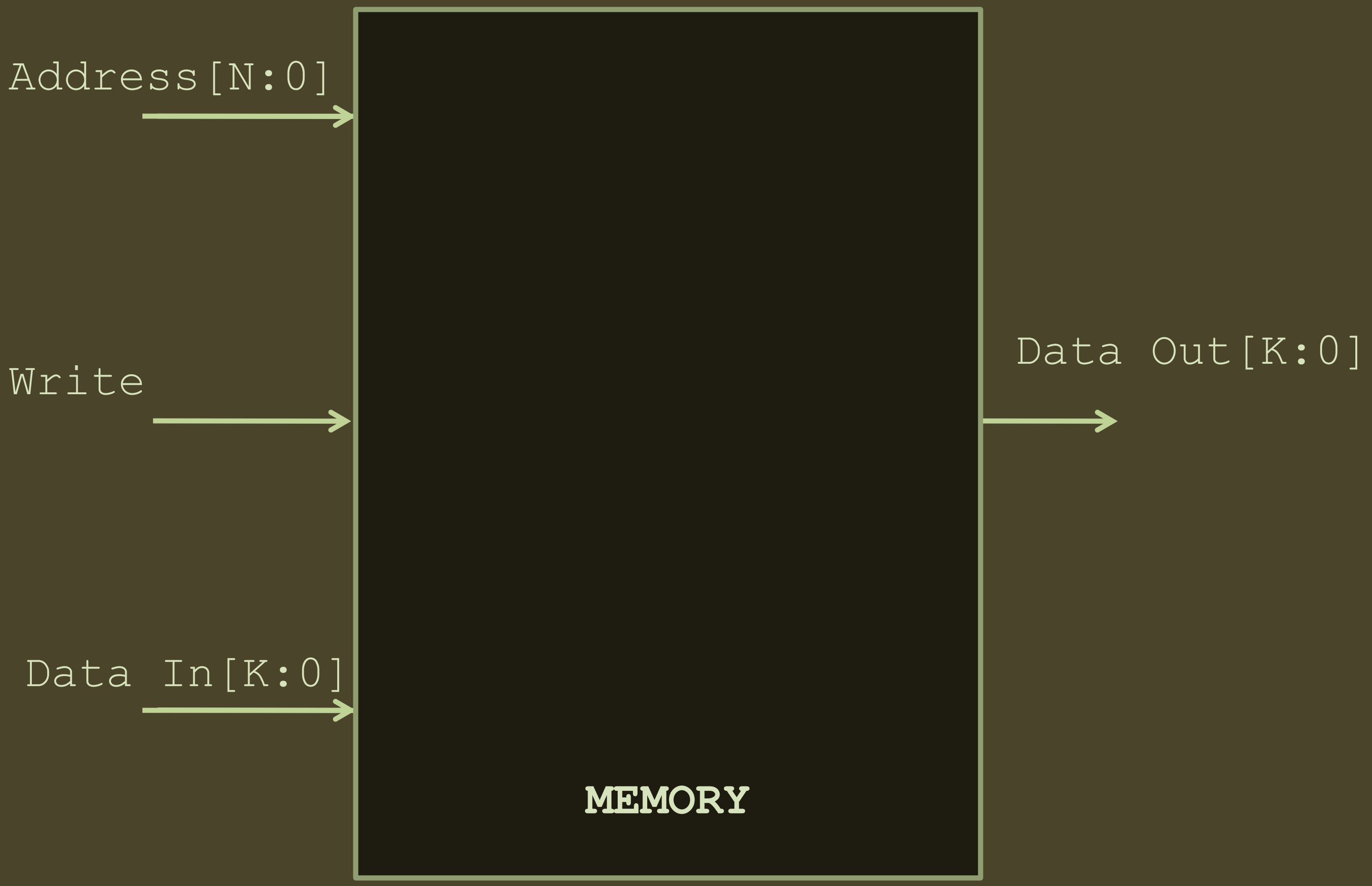

Clock
Input[N:0]
3
4
5
6
Output[N:0]
x
3
4
5

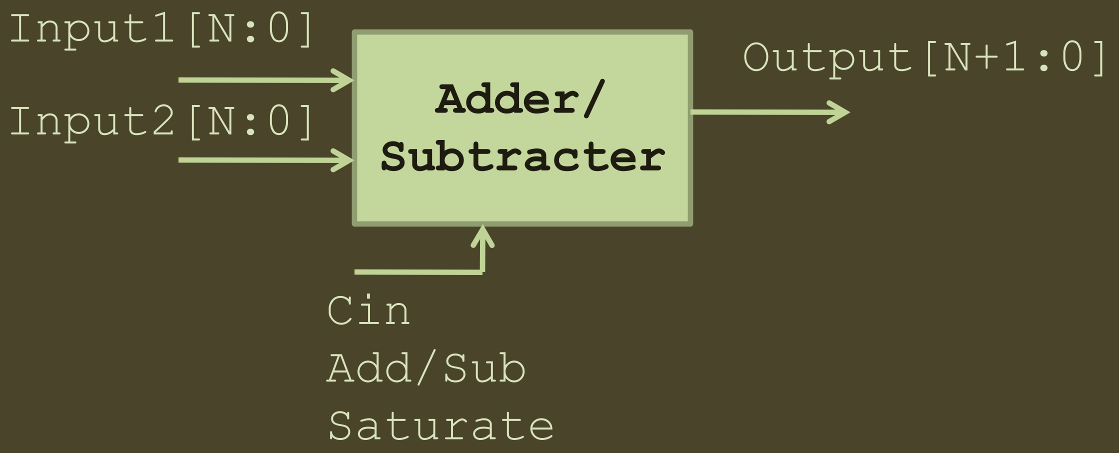

Input[N:0]
Clock
Output[N:0]
Enable

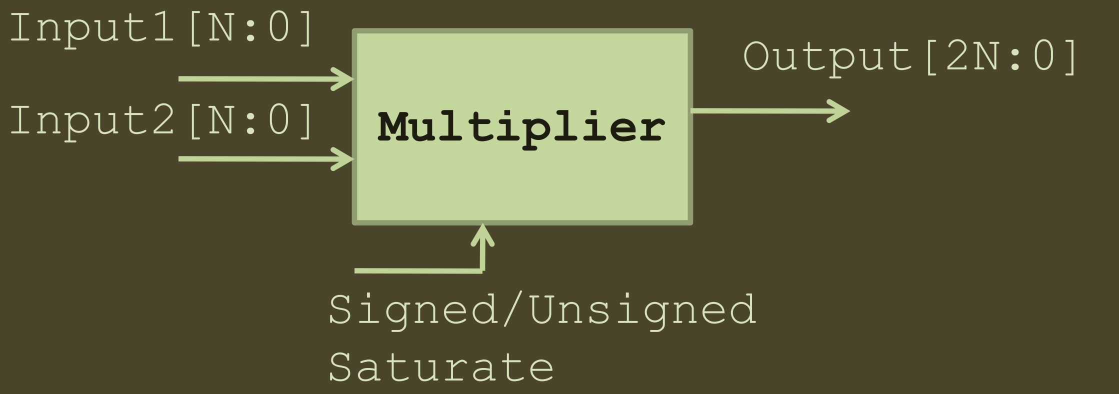

Clock
Input[N:0]
3
4
5
6
Output[N:0]
x
5
Enable


Select Lines / Address
Inputs
Output


Address[N:0]
Write
Data In[K:0]
Data Out[K:0]
Registers File


Address[N:0]
Write
Data In[K:0]
Data Out[K:0]
MEMORY


Input1[N:0]
Input2[N:0]
Adder/
Subtracter
Output[N+1:0]
Cin
Add/Sub
Saturate

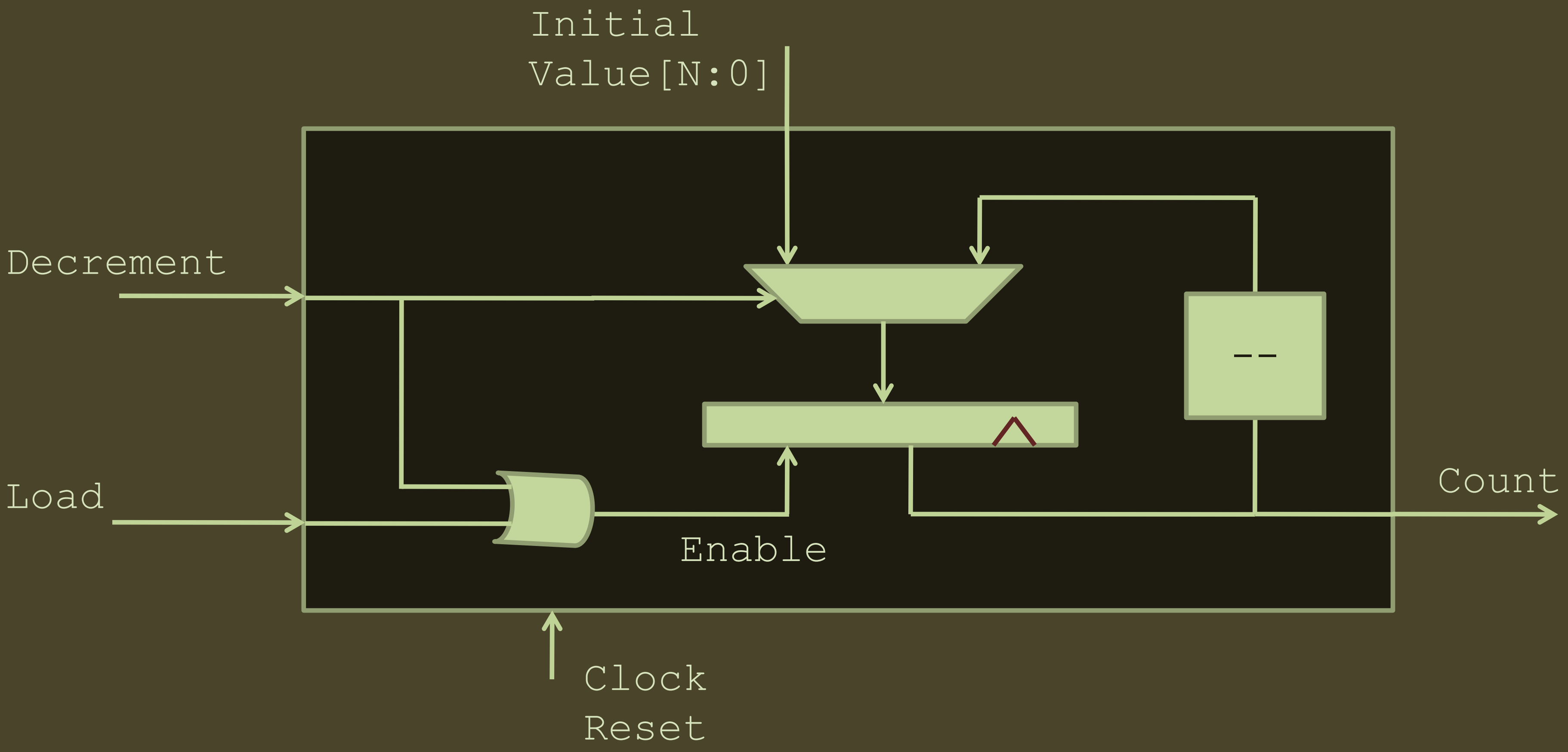

Input1[N:0]
Input2[N:0]
Multiplier
Output[2N:0]
Signed/Unsigned
Saturate

Making new Modules, Using Existing Modules

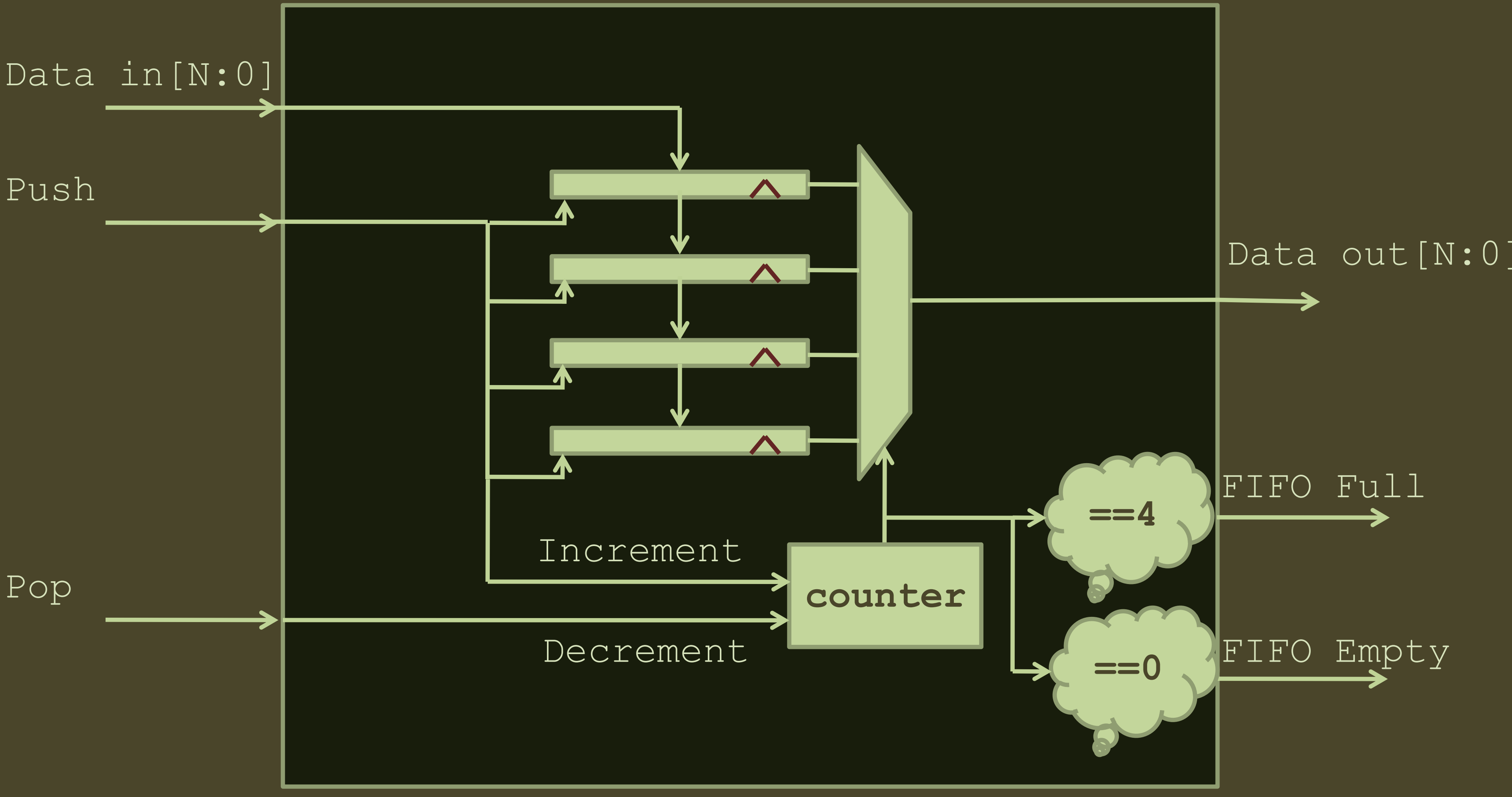
Design Examples

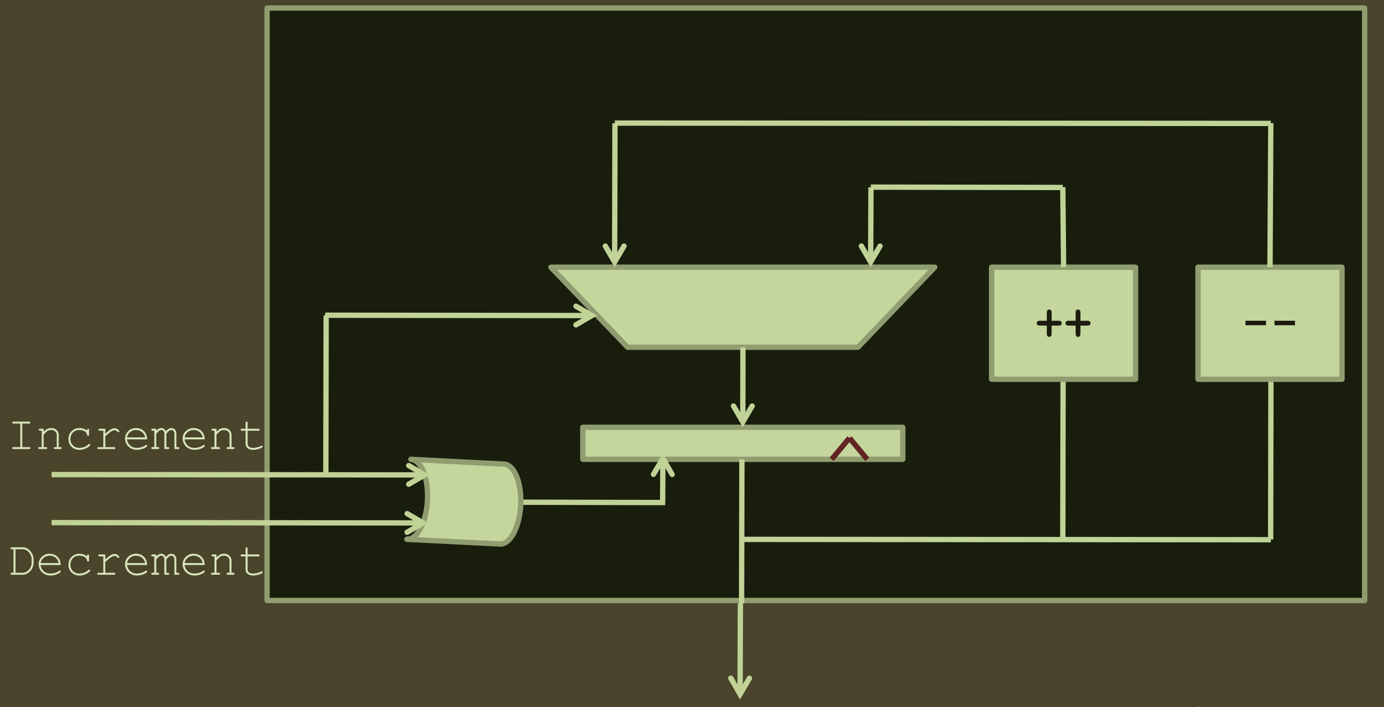
- **Example 1:**
  - A simple decrementer capable of counting number of clock cycles down to zero.
- **Example 2:**
  - A four deep FIFO

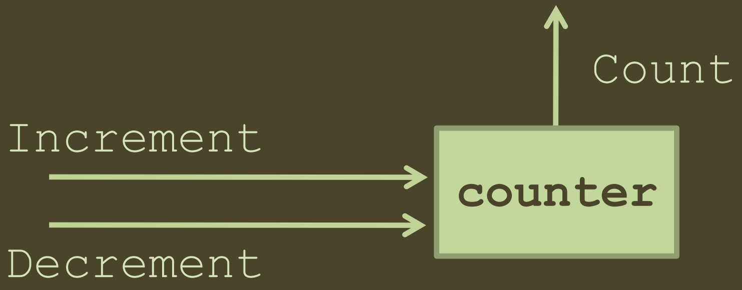
Decrementer



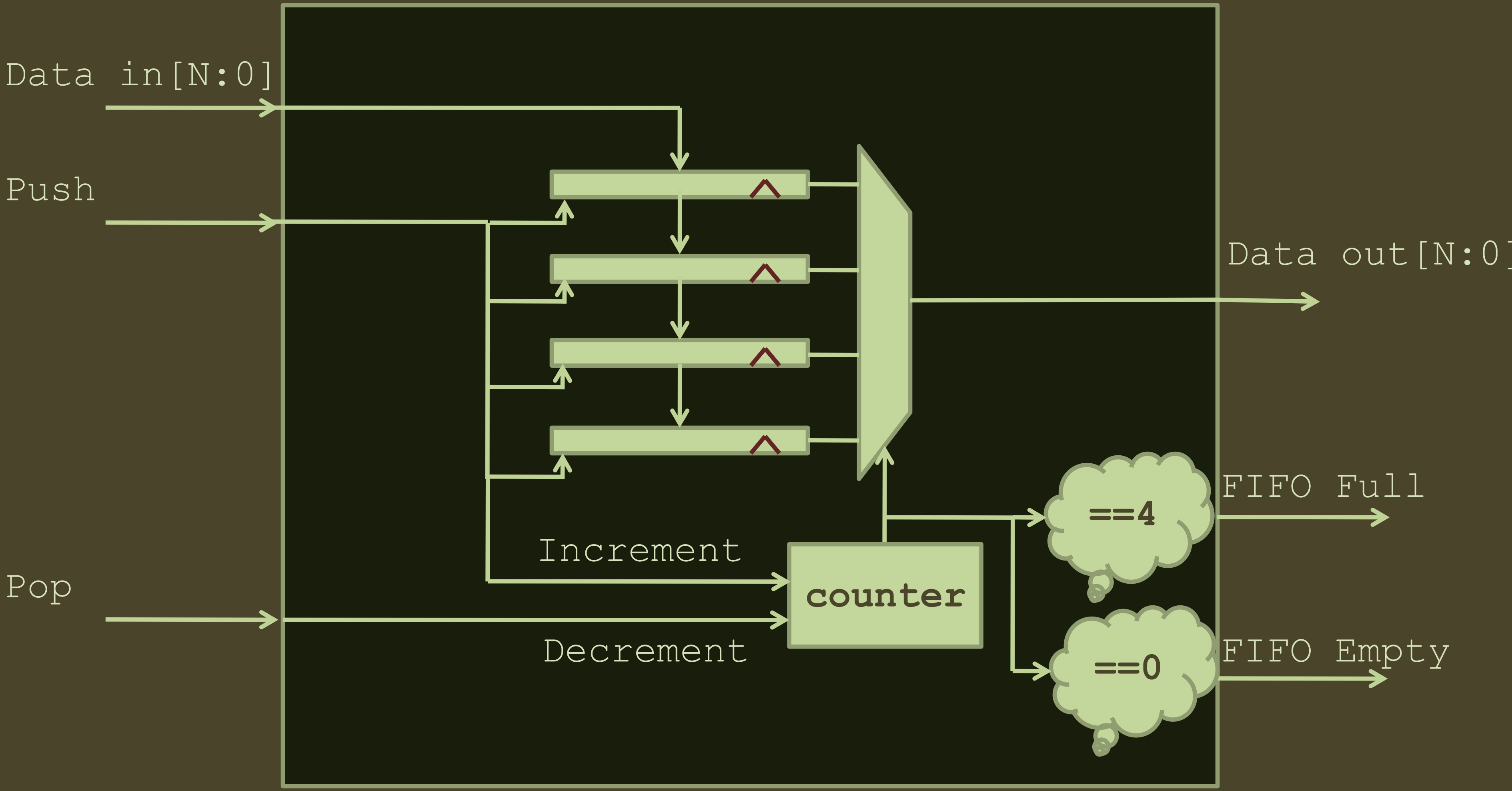
Four deep FIFO

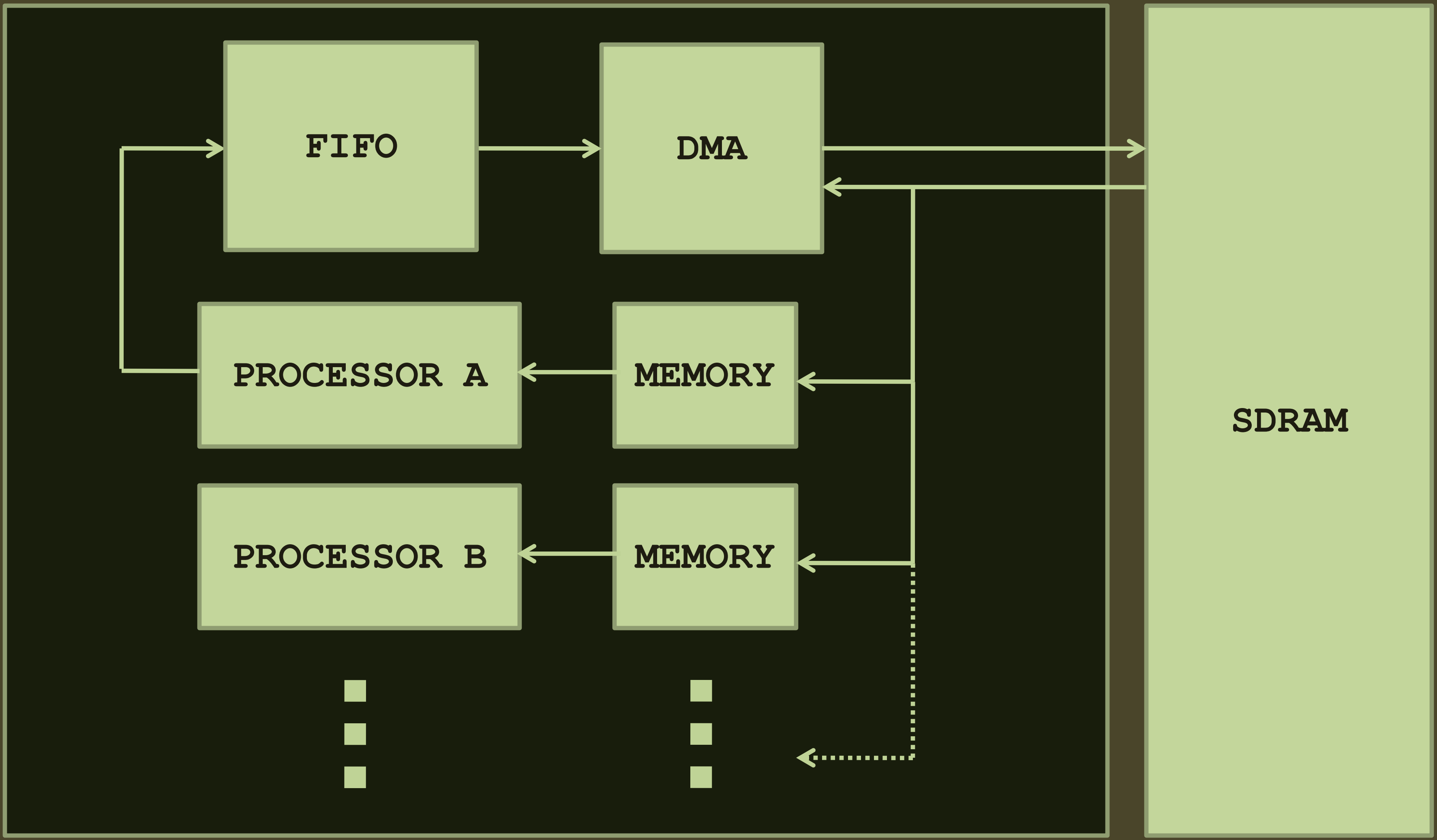



Increment
Decrement
++
--

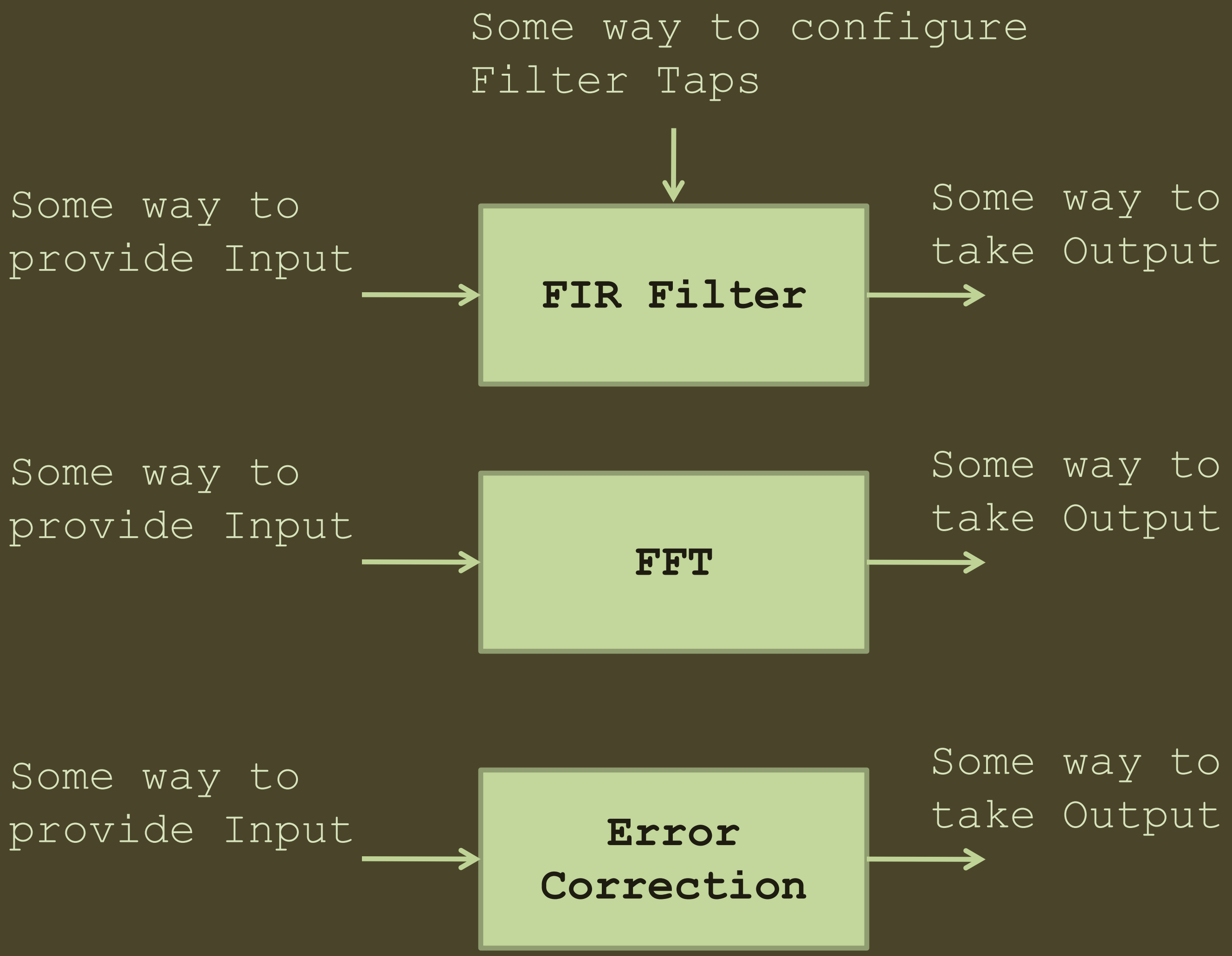

Count
Increment
counter
Decrement

FIFO


Data in[N:0]
Push
Pop
Data out[N:0]
FIFO Full
FIFO Empty
Increment
Decrement
counter
==4
==0


FIFO
DMA
PROCESSOR A
MEMORY
PROCESSOR B
MEMORY
SDRAM

Other components


Some way to configure Filter Taps
Some way to provide Input
FIR Filter
Some way to take Output
Some way to provide Input
FFT
Some way to take Output
Some way to provide Input
Error Correction
Some way to take Output

Verilog

RENZYM

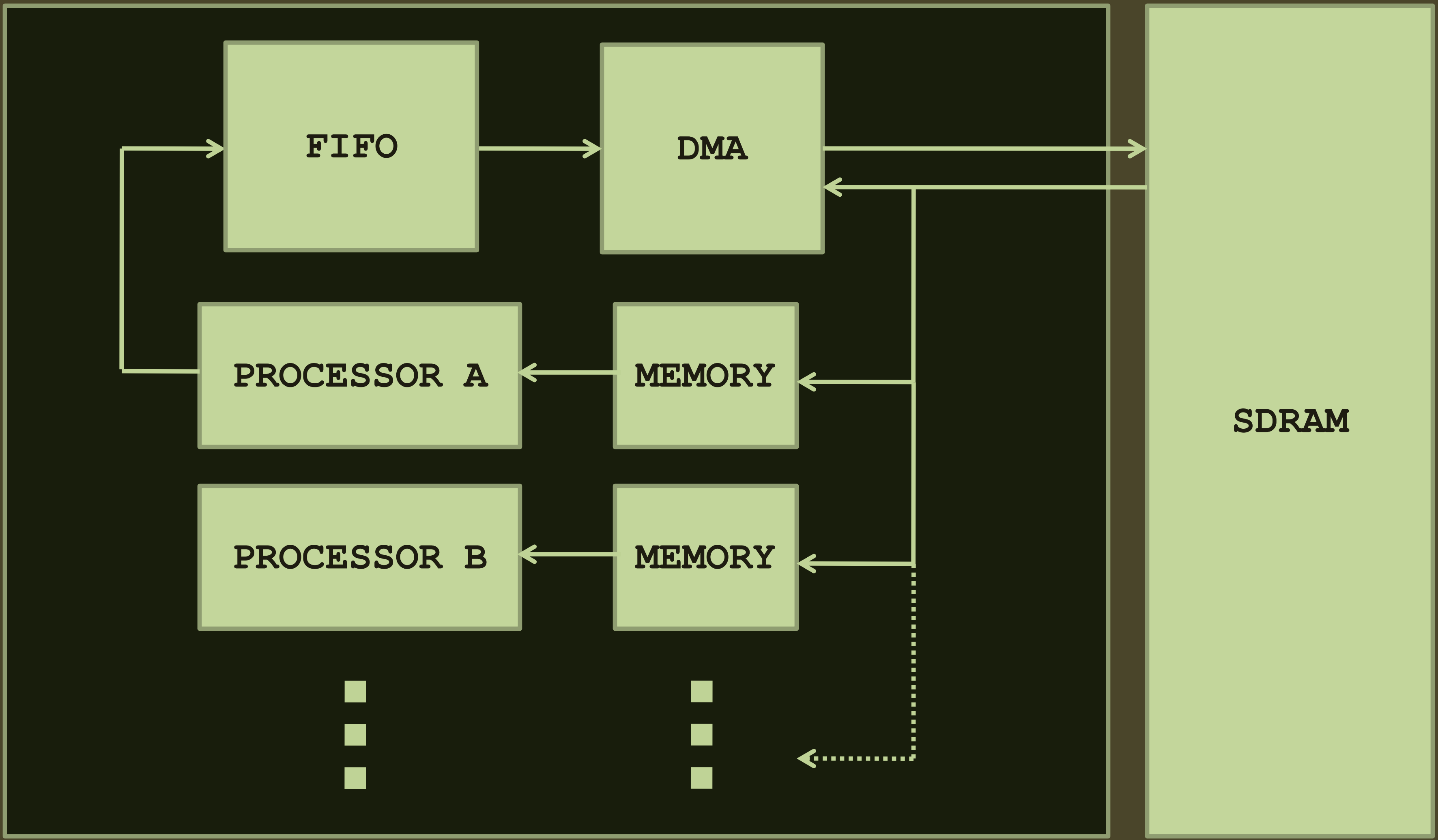
Expanding Horizons

Verilog

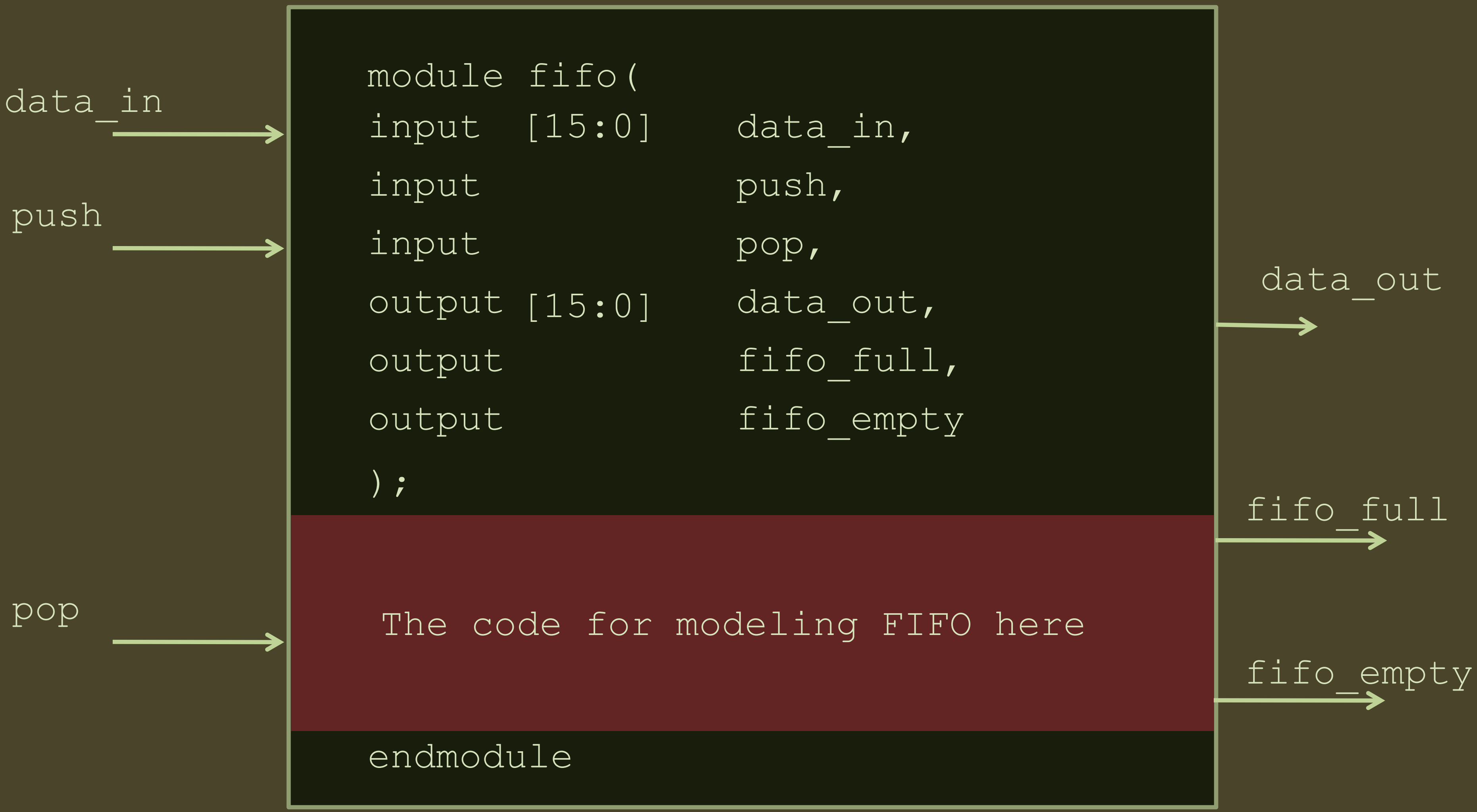
Introduction to Modules

Verilog

Module: A reasonable size replicate-able block e.g. FIFO, counter and decrementer that we covered are good candidates.


FIFO
DMA
PROCESSOR A
MEMORY
PROCESSOR B
MEMORY
SDRAM

fifo.v



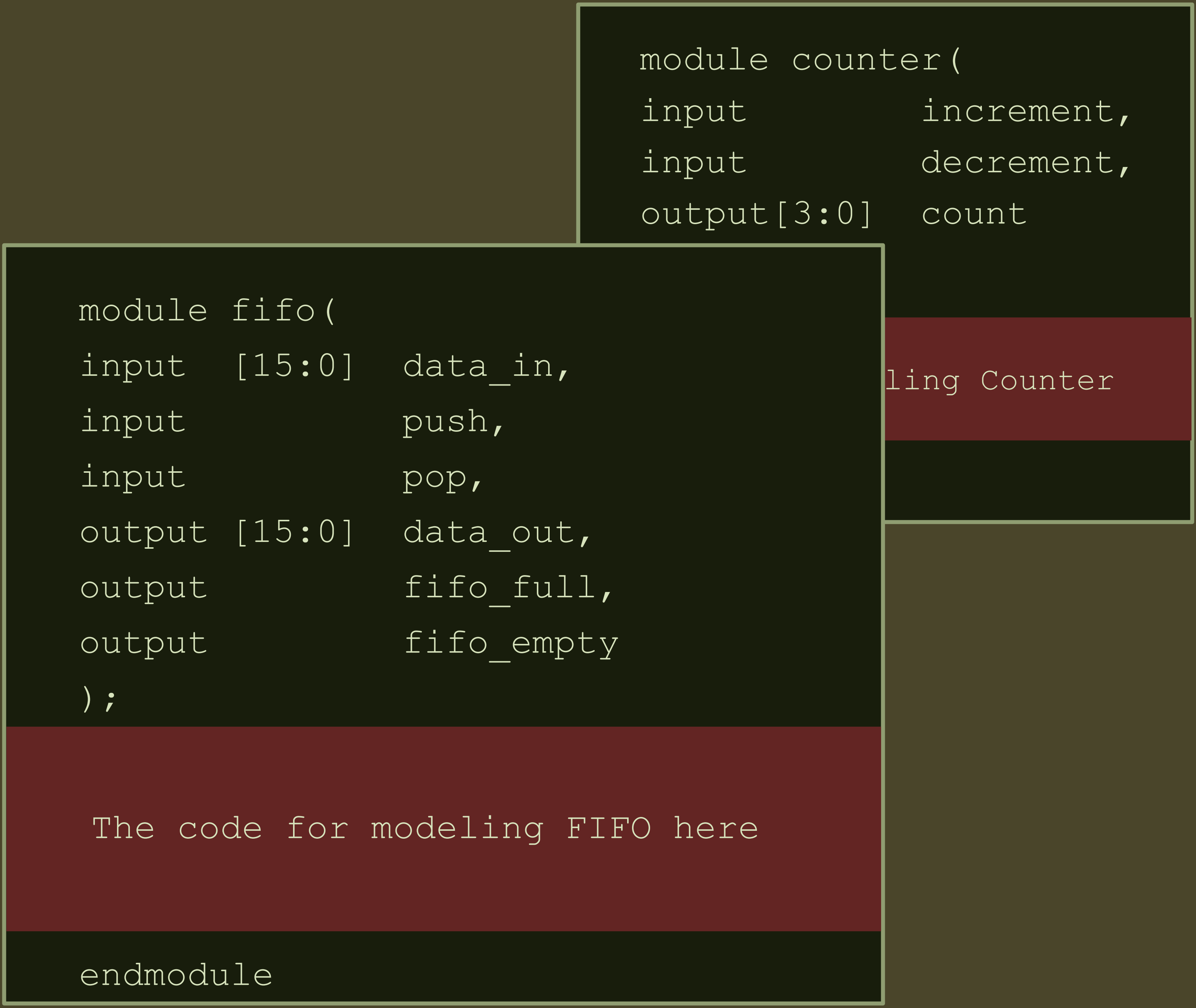
```
module fifo(
input  [15:0]    data_in,
input            push,
input            pop,
output [15:0]    data_out,
output           fifo_full,
output           fifo_empty
);

 The code for modeling FIFO here

endmodule
```

fifo.v

```
module fifo(
input  [15:0]  data_in,
input          push,
input          pop,
output [15:0]  data_out,
output         fifo_full,
output         fifo_empty
);

 The code for modeling FIFO here

endmodule
```

counter.v

```
module counter(
input          increment,
input          decrement,
output[3:0]    count
);

Code for modeling Counter

endmodule
```



```
module counter(
input           increment,
input           decrement,
output[3:0]     count
```

```
ling Counter
```

```
module fifo(
input  [15:0]  data_in,
input          push,
input          pop,
output [15:0]  data_out,
output         fifo_full,
output         fifo_empty
);

 The code for modeling FIFO here

endmodule
```

```
module counter(
input           increment,
input           decrement,
output[3:0]   count
```

ling Counter

```
wire push;
wire pop;
wire [3:0] count;

counter counter_inst1(
.increment    (push),
.decrement    (pop),
.count        (count)
);
```

counter

Summary

- A block level hierarchy can be modeled as modules.
- Module is the basic building block.
- A module can be instantiated inside another module.
- Module can be replicated with a different instance name.

Verilog

RENZYM

Expanding Horizons

Verilog

Basic building blocks

Basic Building Blocks

Basic building blocks for Logic design are

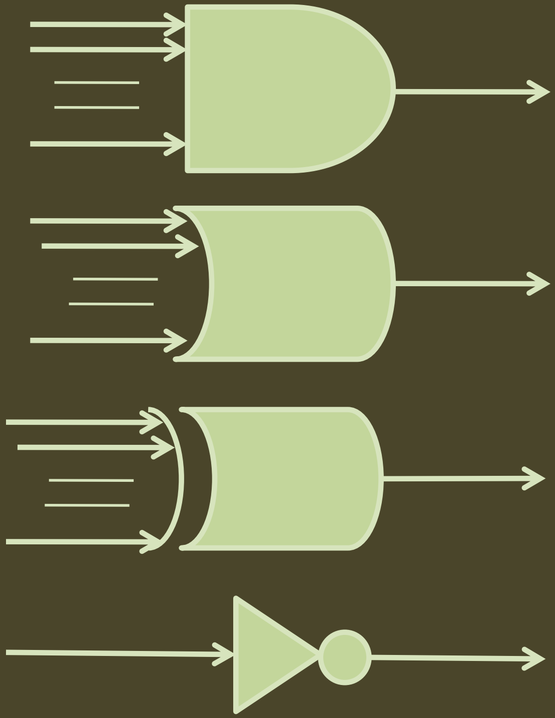
- Gates
- Muxes/Demuxes
- Registers/Memory
- Arithmetic Operations (Adder, Subtracter, Multipliers etc.)
- State machines

What we are trying to do?

We are trying to model our building blocks using words and characters organized in a set of files

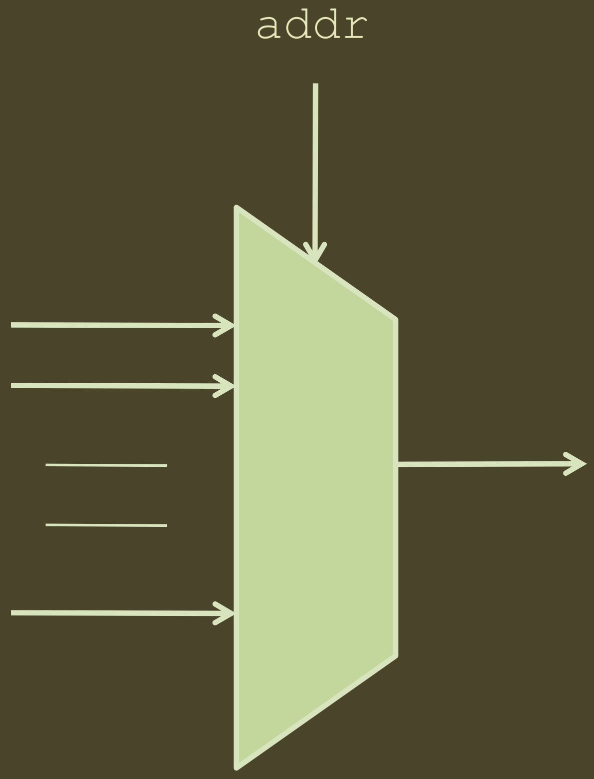
Modeling Gates

```
always @(*)
begin
    a = b & c & d;
end

    a = b | c | d;

    a = b ^ c ^ d;

    a = ~b;
```



Modeling Muxes/Demuxes

```
always @(*)
begin
    case(addr)
    2'd0: out = a;
    2'd1: out = b;
    2'd2: out = c;
    2'd3: out = d;
    endcase
end
```



MUX

Data Values

```
8’hXA
8’bXXXX_1010
8’hA
8’d10
8’b0000_1010
```

Underscores are ignored

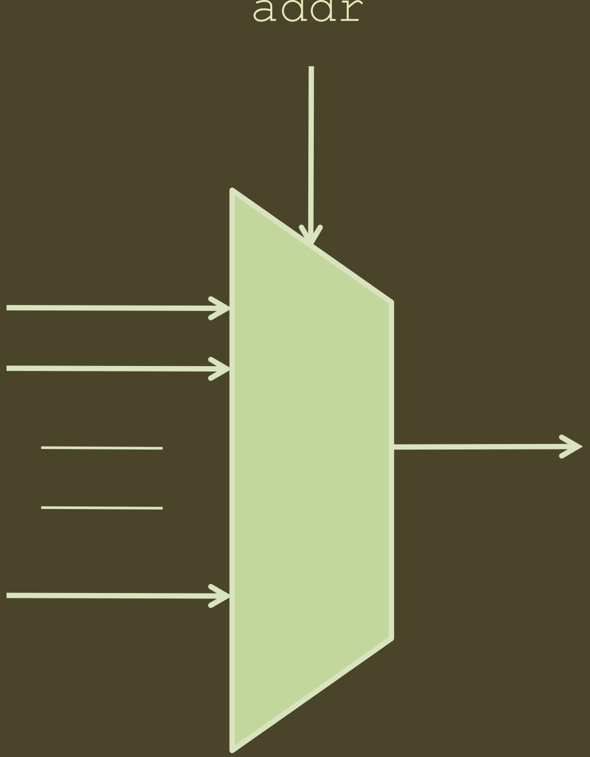
X’s show don’t care values

Base Format (b,o,d,h)

Width

Modeling Muxes/Demuxes

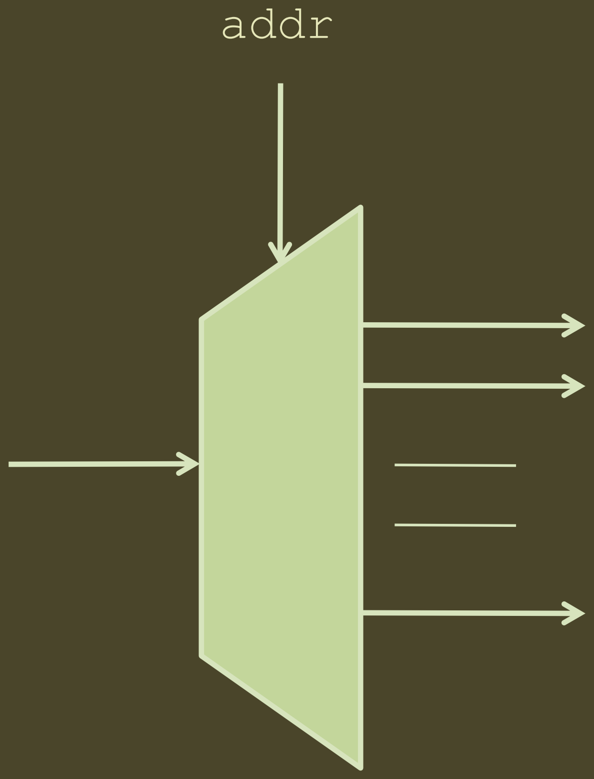
```
always @(*)
begin
   case(addr)
   2'd0: out = a;
   2'd1: out = b;
   2'd2: out = c;
   2'd3: out = d;
   endcase
end
```



DEMUX

Modeling Muxes/Demuxes

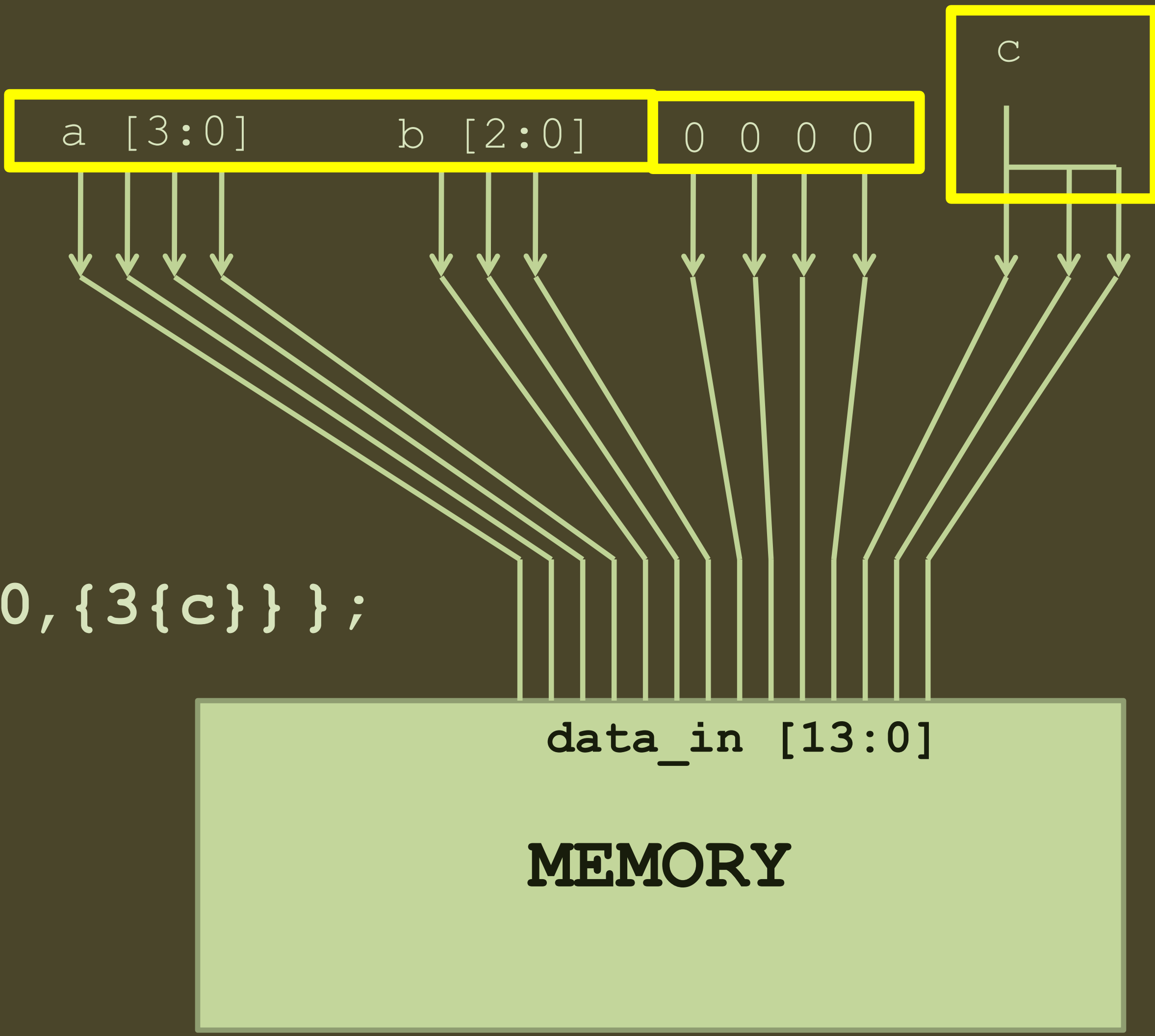
```
always @(*)
begin
  case(addr)
  2'd0: {a,b,c,d} = {in,3'd0};
  2'd1: {a,b,c,d} = {1'd0,in,2'd0};
  2'd2: {a,b,c,d} = {2'd0,in,1'd0};
  2'd3: {a,b,c,d} = {3'd0,in};
  endcase
end
```



DEMUX

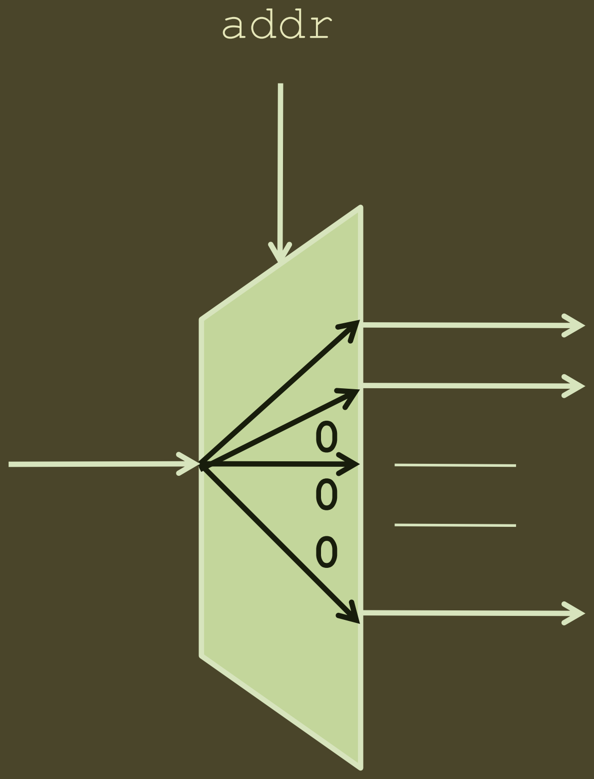
concatenation

```
data_in = {a,b,4'd0,{3{c}}};
```



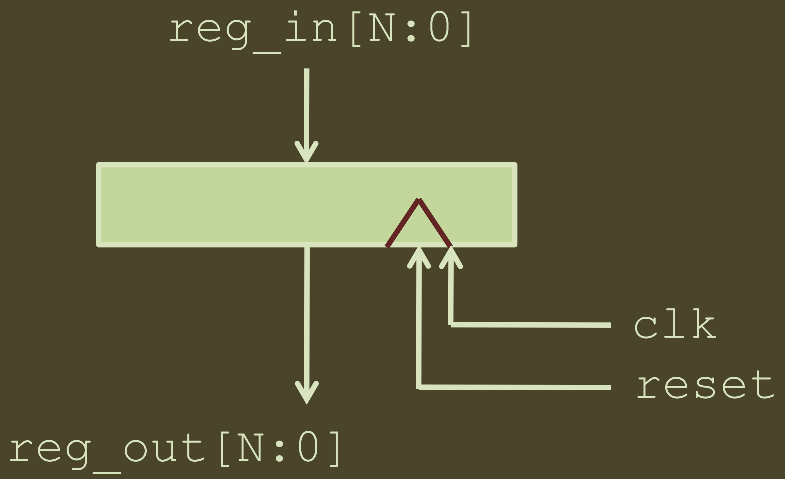
Modeling Muxes/Demuxes

```
always @(*)
begin
  case(addr)
  2'd0: {a,b,c,d} = {in,3'd0};
  2'd1: {a,b,c,d} = {1'd0,in,2'd0};
  2'd2: {a,b,c,d} = {2'd0,in,1'd0};
  2'd3: {a,b,c,d} = {3'd0,in};
  endcase
end
```



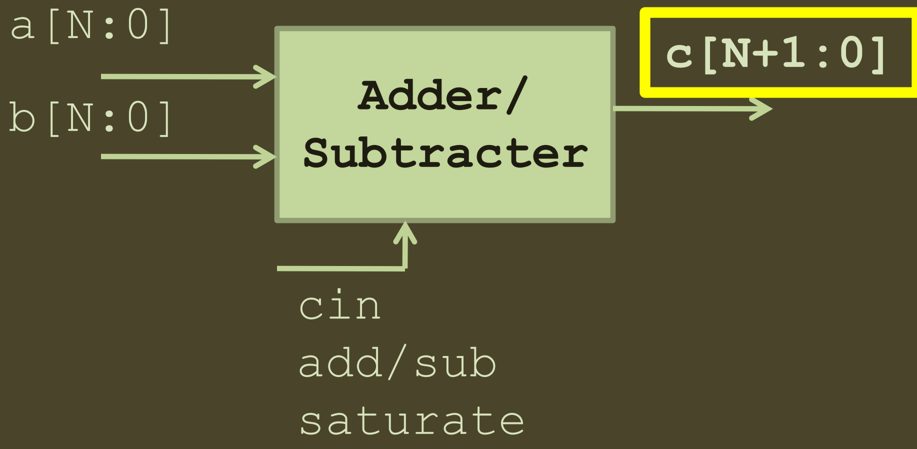
Modeling Registers

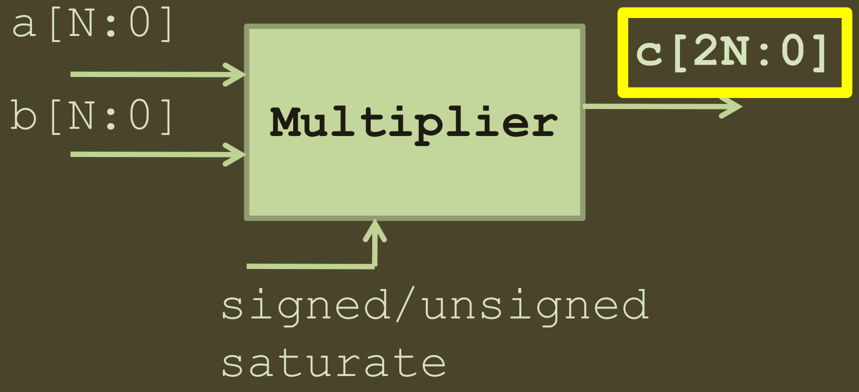
```
always@(posedge clk)
begin
   if(reset)   reg_out <= #1 0;
   else        reg_out <= #1 reg_in;
end
```



Arithmetic Operations

```
always@(*)
begin
     c = a + b;
end


     c = a - b;
     c = a * b;
```

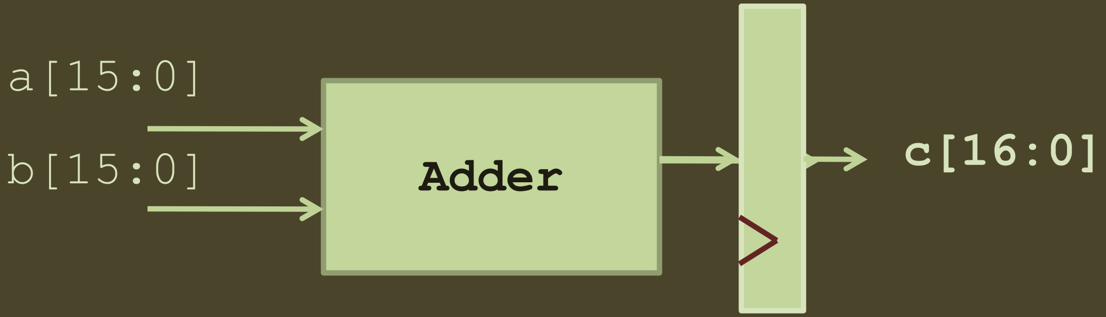




Declarations

```
reg  [16:0] c;            wire [16:0] c;

always@(*)                assign  c = a + b;
  c  =    a + b;
```

What about a & b?
Should they be
wire or **reg**?



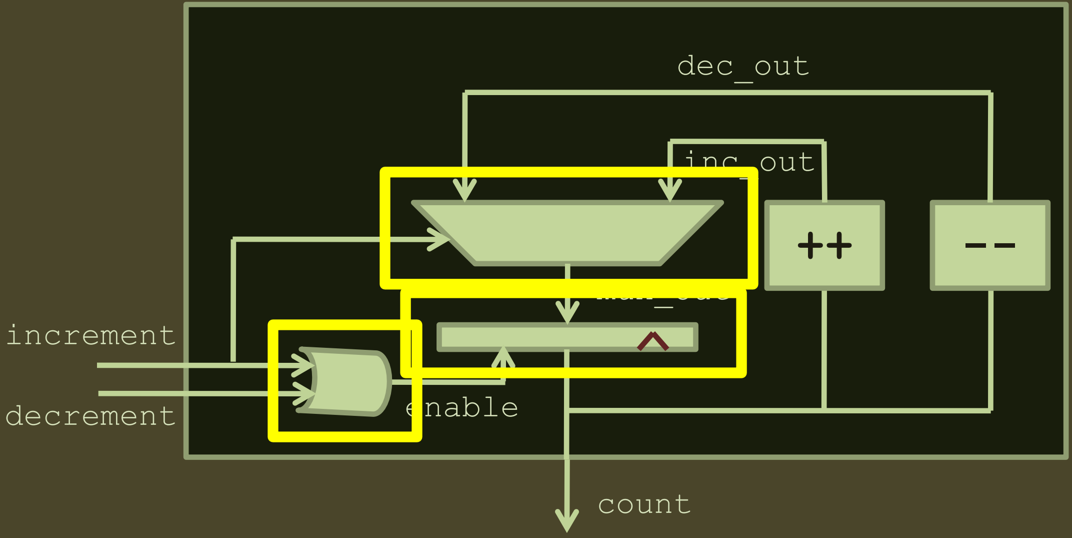
```
always@(posedge clk)
  c <= #1 a + b;
```

Verilog Coding

- Make a Design Diagram
- Code all the building blocks
  - Name the wires in the design diagram
  - Declare the module port list
  - Start coding the logic elements one by one
  - Declare Variables

Example 1: counter.v

```
module counter(
input       clk,
input       reset,
input       increment,
input       decrement,
output reg [3:0] count
);

 reg       enable;
 reg [3:0] mux_out;

 always@(*)
    enable = increment | decrement;

 always@(*)
 begin
    case(increment)
    1'b0: mux_out = count-1;
    1'b1: mux_out = count+1;
    endcase
 end

 always@(posedge clk)
 if(reset)
    count <= #1 0;
 else if(enable)
    count <= #1 mux_out;

endmodule
```


dec_out
inc_out
++
--
mux_out
increment
decrement
enable
count