Memory II

Andrew Lukefahr

Always specify defaults for **`always_comb`!**

BLOCKING (=) FOR `always_comb`

NON-BLOCKING (<=) for `always_ff`

UART RX/TX LEDs on Basys3

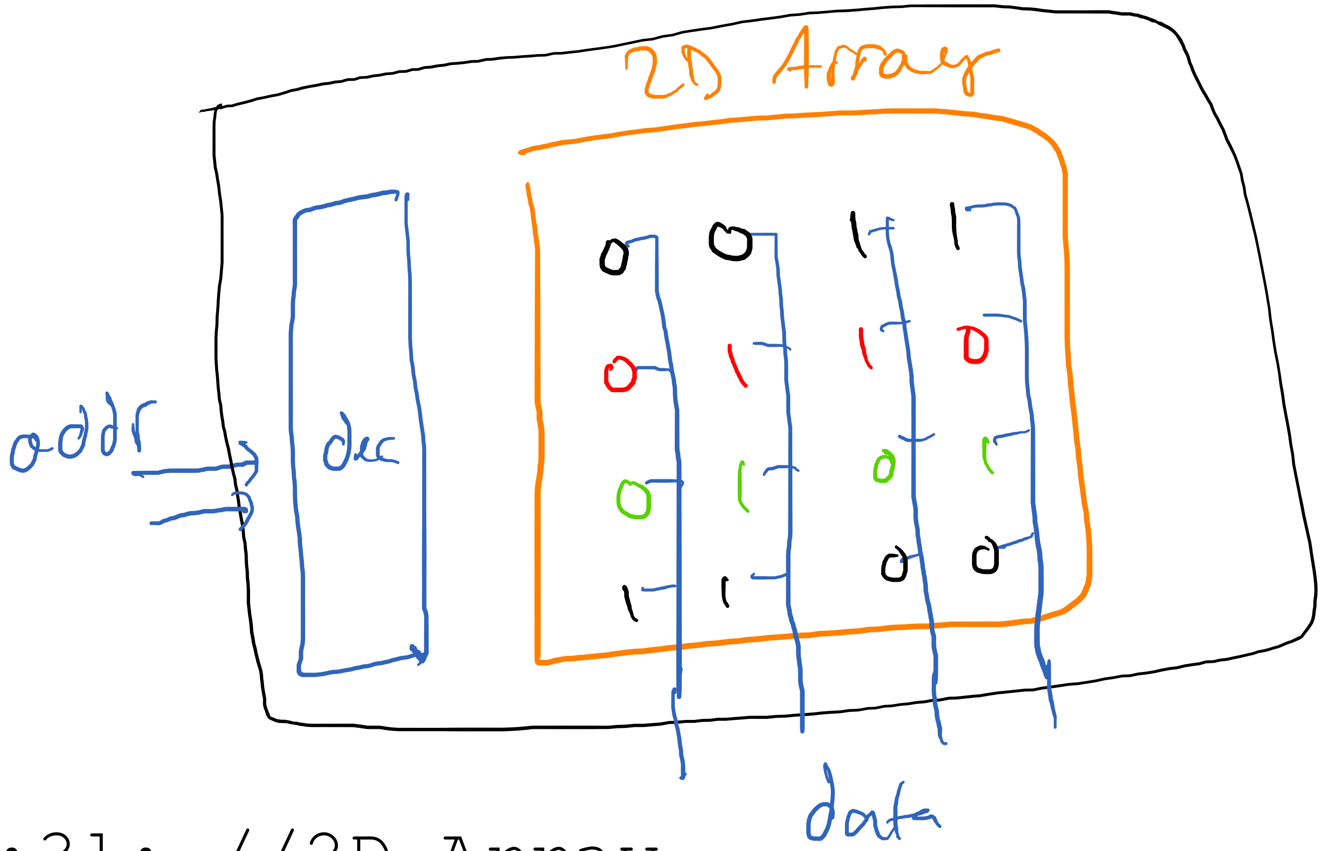
- A word of caution:
- The Basys3's RX + TX LEDs are backwards from what you expect.
- They are the USB adaptor chip's RX+TX, not the FPGAs.

ROM vs RAM

- ROM – Read-Only Memory
    - Input: address
    - Output: fixed value

- RAM – Random-Access Memory
    - Read/Write version of a ROM

ROM in Verilog



```
module ROM (
   input [1:0] addr,
   output [3:0] data
)
   logic [3:0] array [0:3]; //2D Array
   assign array = { 4’b0011, 4’b0110, 4’b0101, 4’b1100}
   assign data = array[addr];
endmodule
```

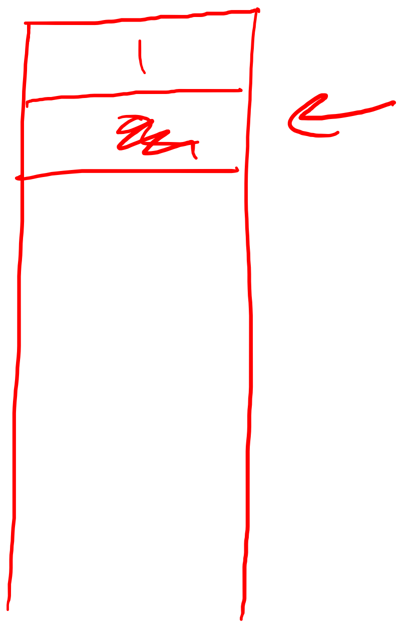
$\leftarrow$ select a row for output

Flip-Flop RAM in Verilog

```
module RAM (
  input        clk,
  input [1:0] addr,
  input        set,
  input [3:0]  set_data,
  output [3:0] read_data
)
  logic [3:0] array [0:3]; //2D Array
  always_ff @(posedge clk) begin
       if (set) array[addr] <= set_data;
  end
  assign read_data = array[addr];
endmodule
```

Stacks

push(1)
push(2)
pop() → 2



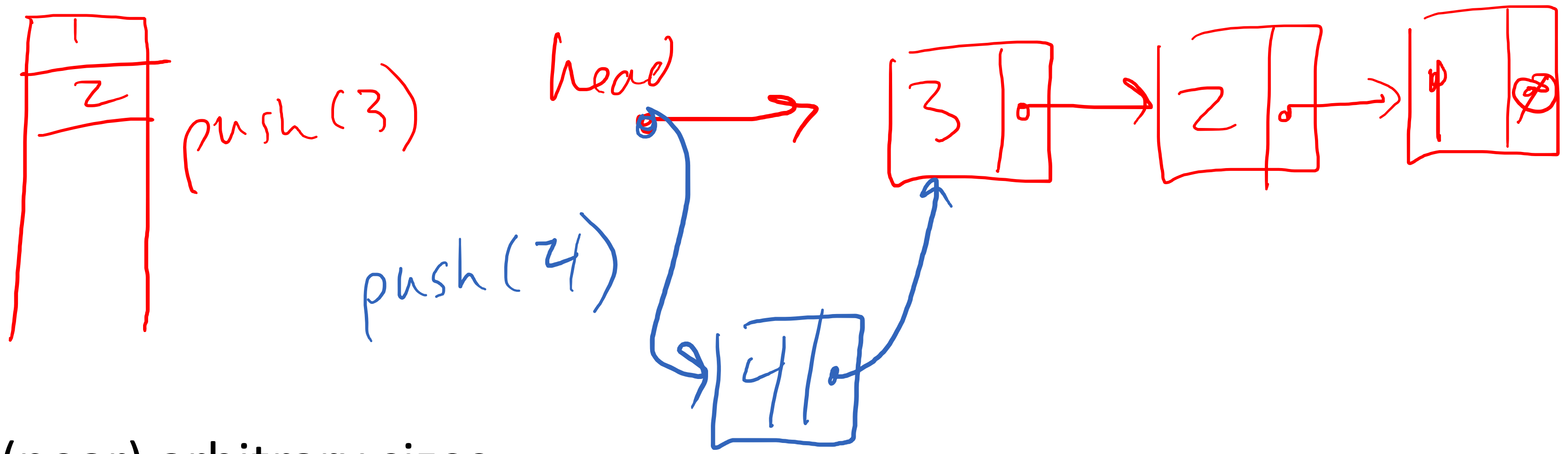
- First-In-Last-Out data structure
- Defines two operations:
  - push(x): adds an element to the end of the stack
  - X = pop(): returns most recently-added element from the stack

Stacks

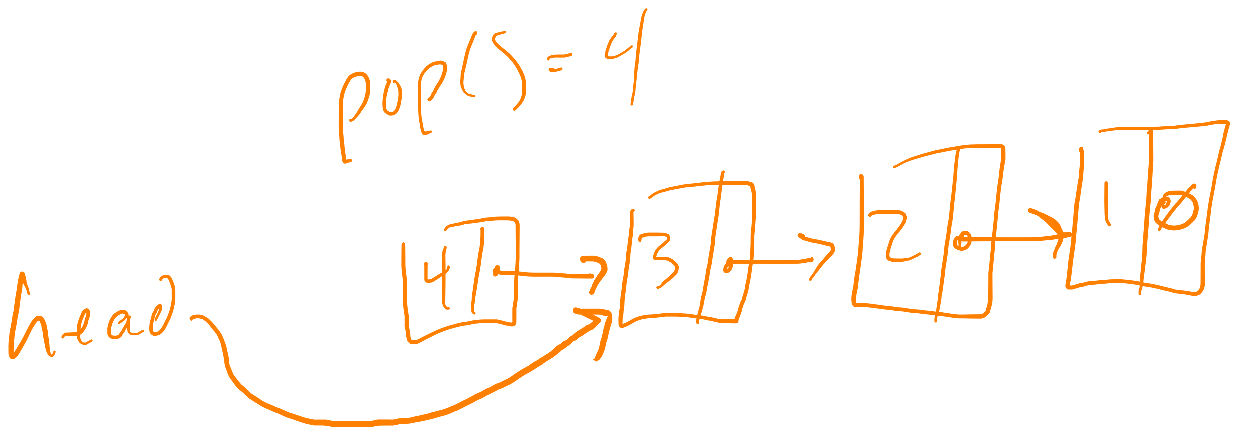
- In C/C++:
  - Can grow to (near) arbitrary sizes
  - Implemented with linked lists
  - `malloc()` allows more memory for bigger stacks

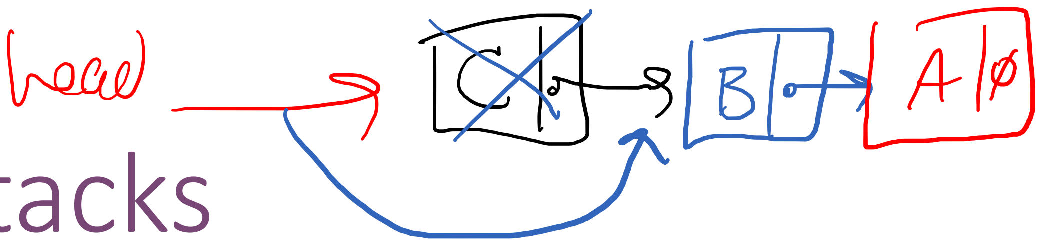
- In Hardware:
  - Don’t have `malloc()`
  - Can’t get “more gates”
  - Fixed size!

Stacks



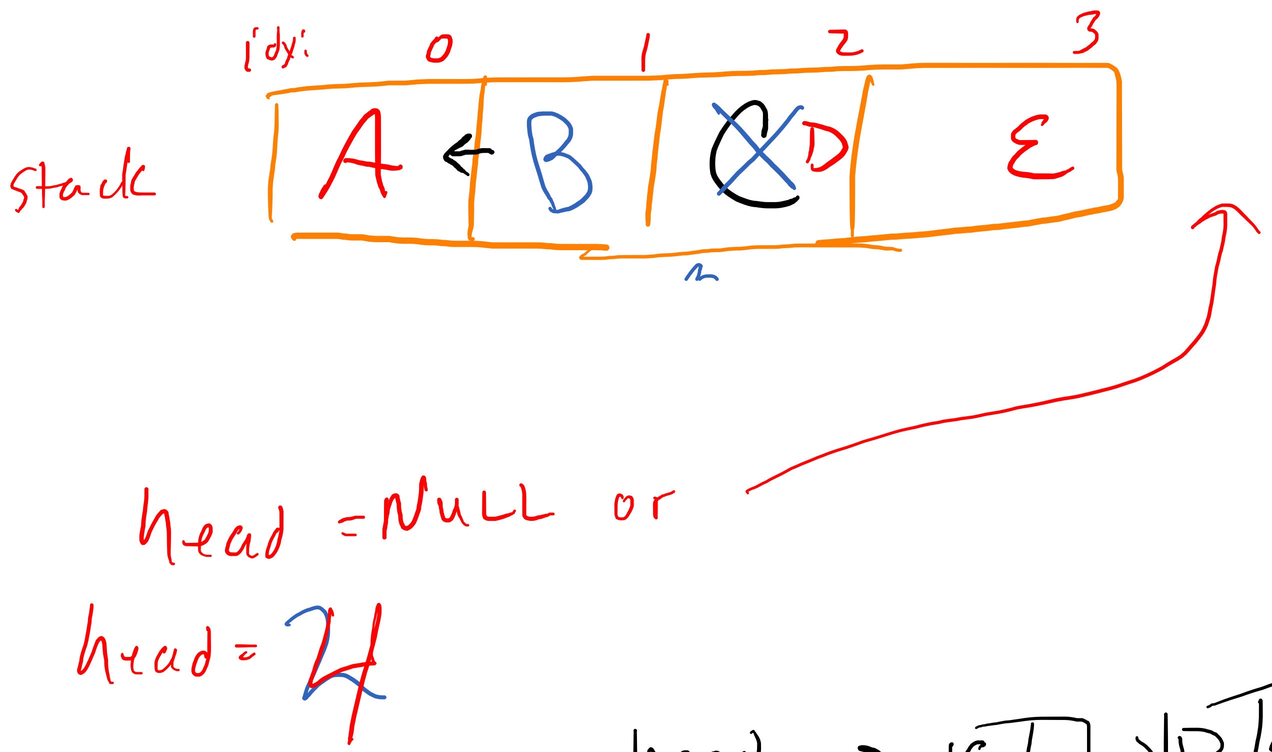
- In C/C++:
  - Can grow to (near) arbitrary sizes
  - Implemented with linked lists
  - `malloc()` allows more memory for bigger stacks
- In Hardware:
  - Don't have `malloc()`
  - Can't get "more gates"
  - Fixed size!

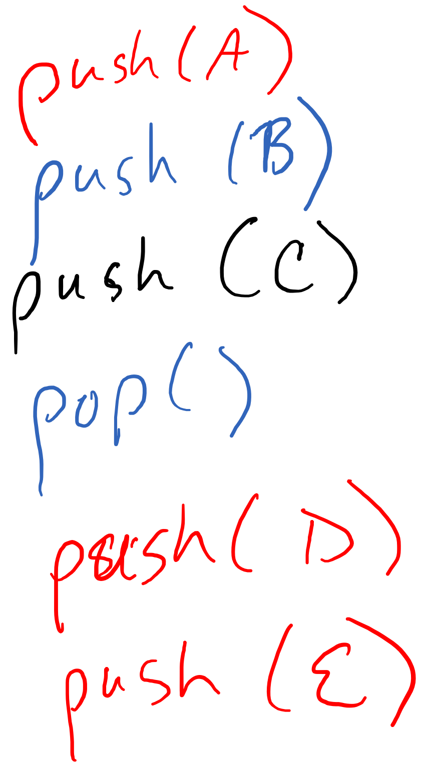


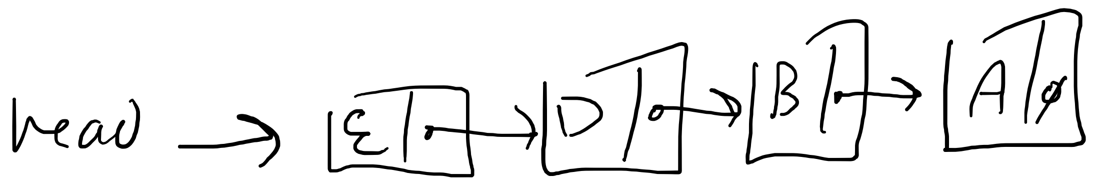


Fixed-Size Stacks

- Use an array as a fixed-size stack

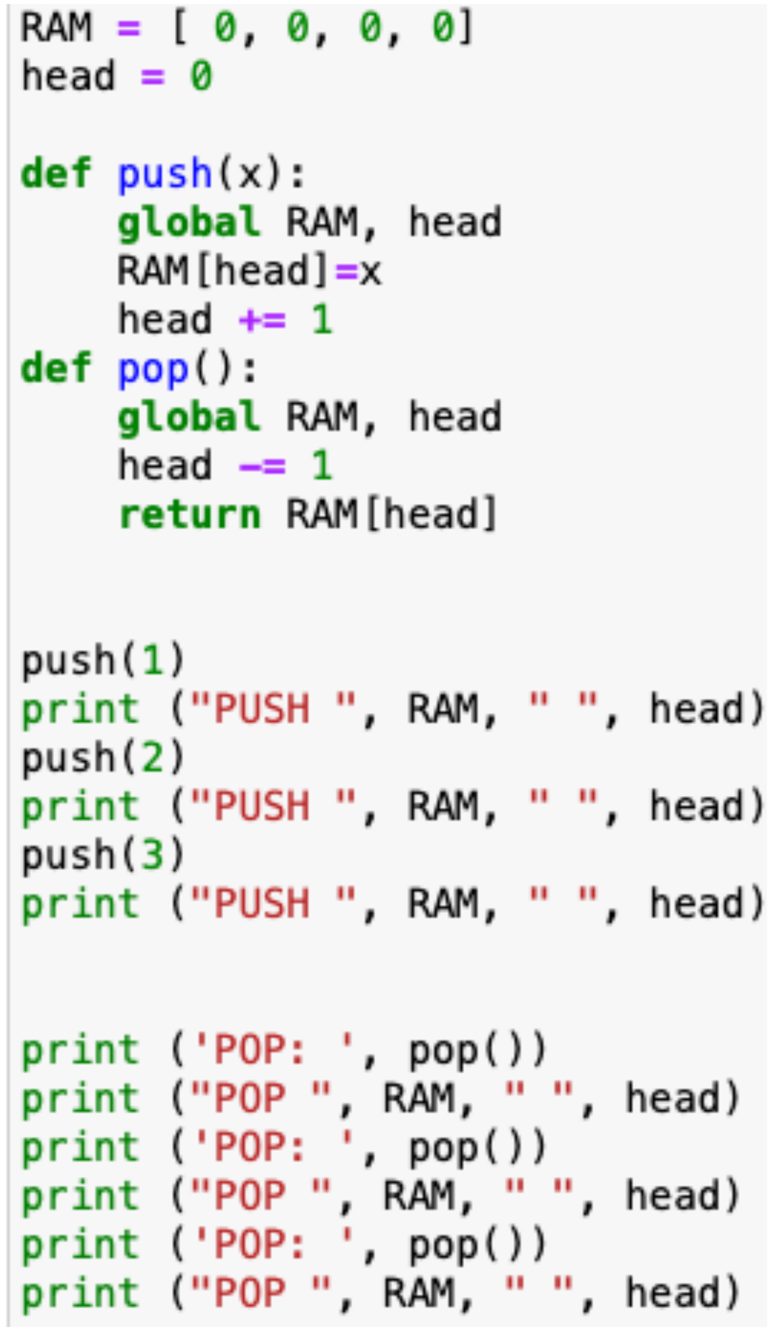




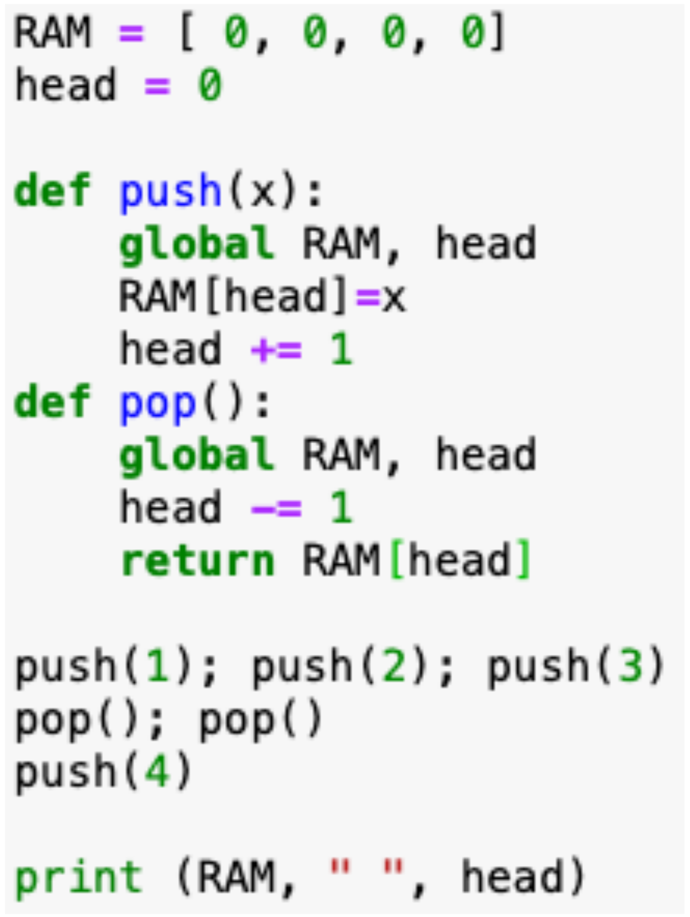


Python Example

```
PUSH [1, 0, 0, 0] 1
PUSH [1, 2, 0, 0] 2
PUSH [1, 2, 3, 0] 3
POP: 3
POP [1, 2, 3, 0] 2
POP: 2
POP [1, 2, 3, 0] 1
POP: 1
POP [1, 2, 3, 0] 0
```



```python
RAM = [ 0, 0, 0, 0]
head = 0

def push(x):
    global RAM, head
    RAM[head]=x
    head += 1
def pop():
    global RAM, head
    head -= 1
    return RAM[head]


push(1)
print ("PUSH ", RAM, " ", head)
push(2)
print ("PUSH ", RAM, " ", head)
push(3)
print ("PUSH ", RAM, " ", head)


print ('POP: ', pop())
print ("POP ", RAM, " ", head)
print ('POP: ', pop())
print ("POP ", RAM, " ", head)
print ('POP: ', pop())
print ("POP ", RAM, " ", head)
```

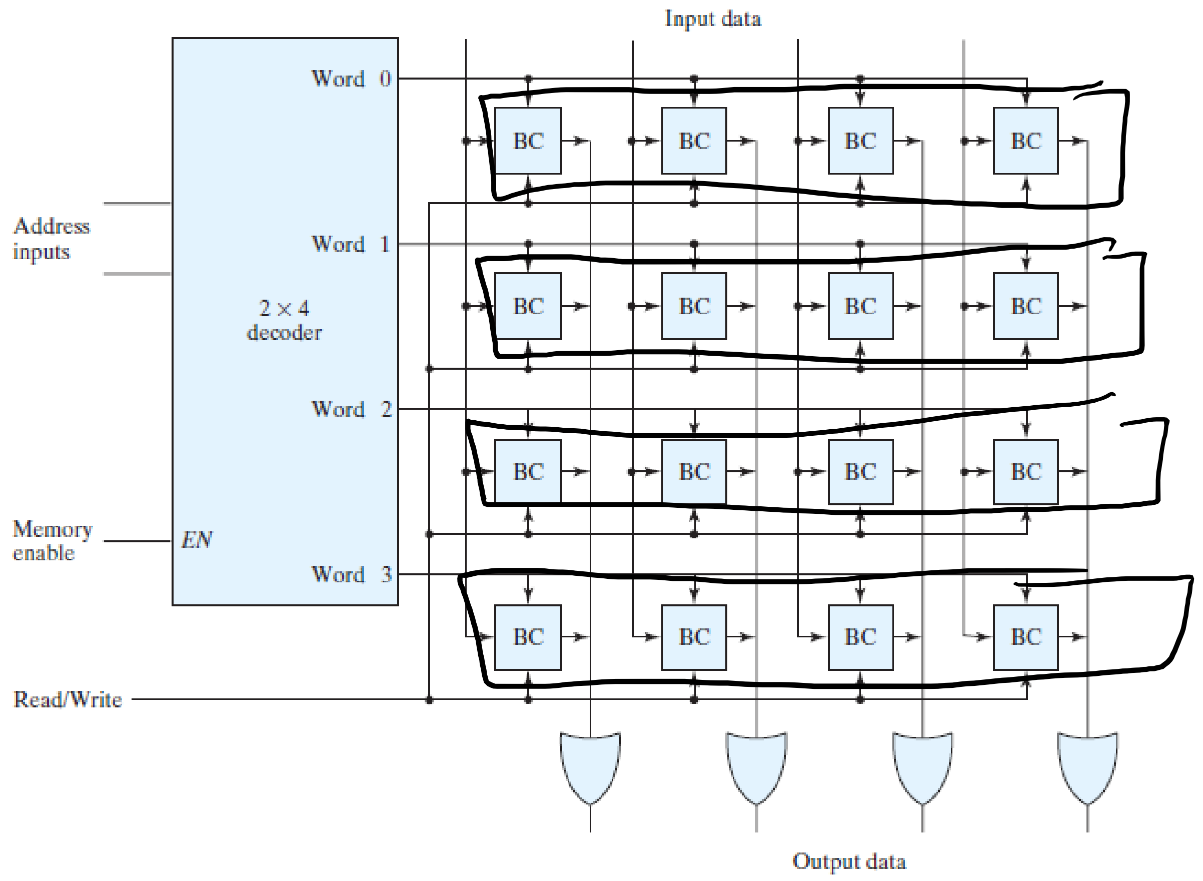


```python
RAM = [ 0, 0, 0, 0]
head = 0

def push(x):
    global RAM, head
    RAM[head]=x
    head += 1
def pop():
    global RAM, head
    head -= 1
    return RAM[head]

push(1); push(2); push(3)
pop(); pop()
push(4)

print (RAM, " ", head)
```

```
PUSH [1, 0, 0, 0] 1
PUSH [1, 2, 0, 0] 2
PUSH [1, 2, 3, 0] 3
POP: 3
POP [1, 2, 3, 0] 2
POP: 2
POP [1, 2, 3, 0] 1
PUSH [1, 4, 3, 0] 2


The stack values: [1, 4]
```

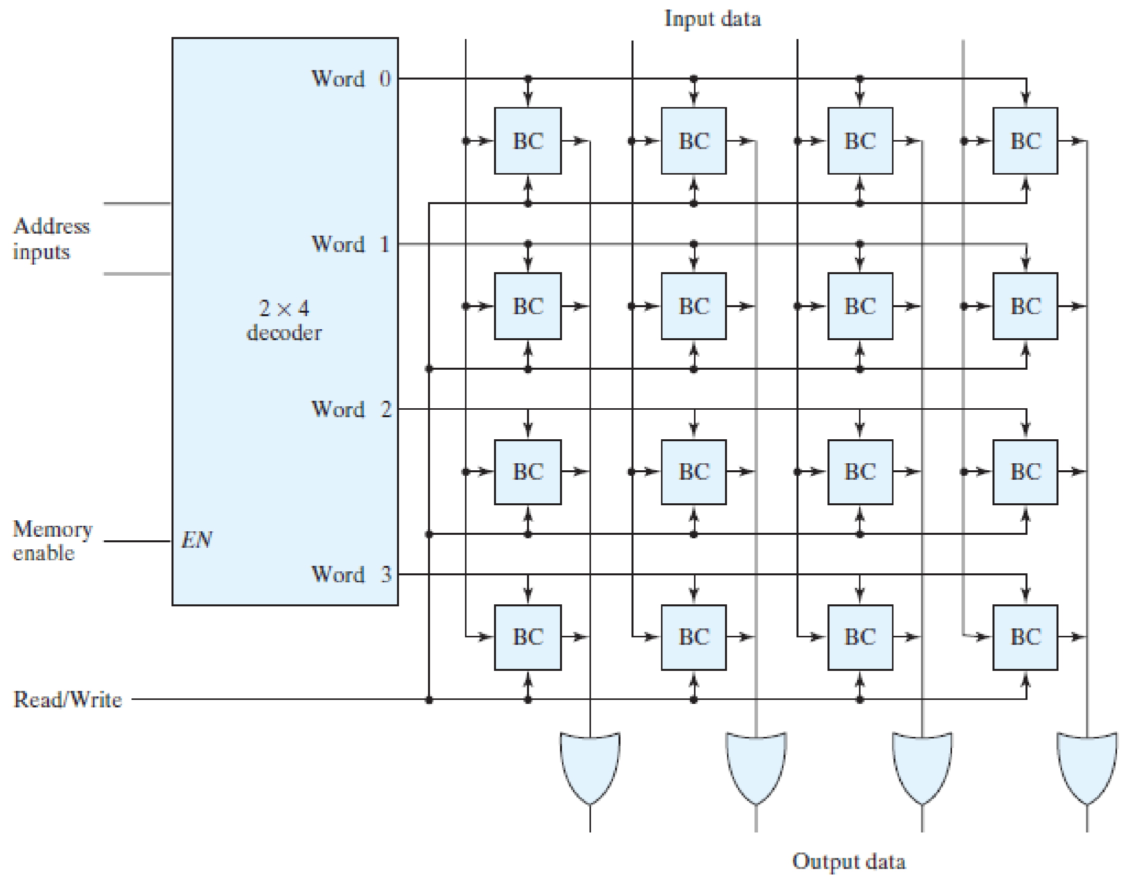
Fixed-Size Stack in Hardware

- We can use a RAM block as a stack
- Just need to add head index



Stack with RAMs

- Given: RAM array (shown)
- Make: 4-element 4-bit **Stack**
  - Recall: First-In-Last-Out
- Tip: Use a state machine!

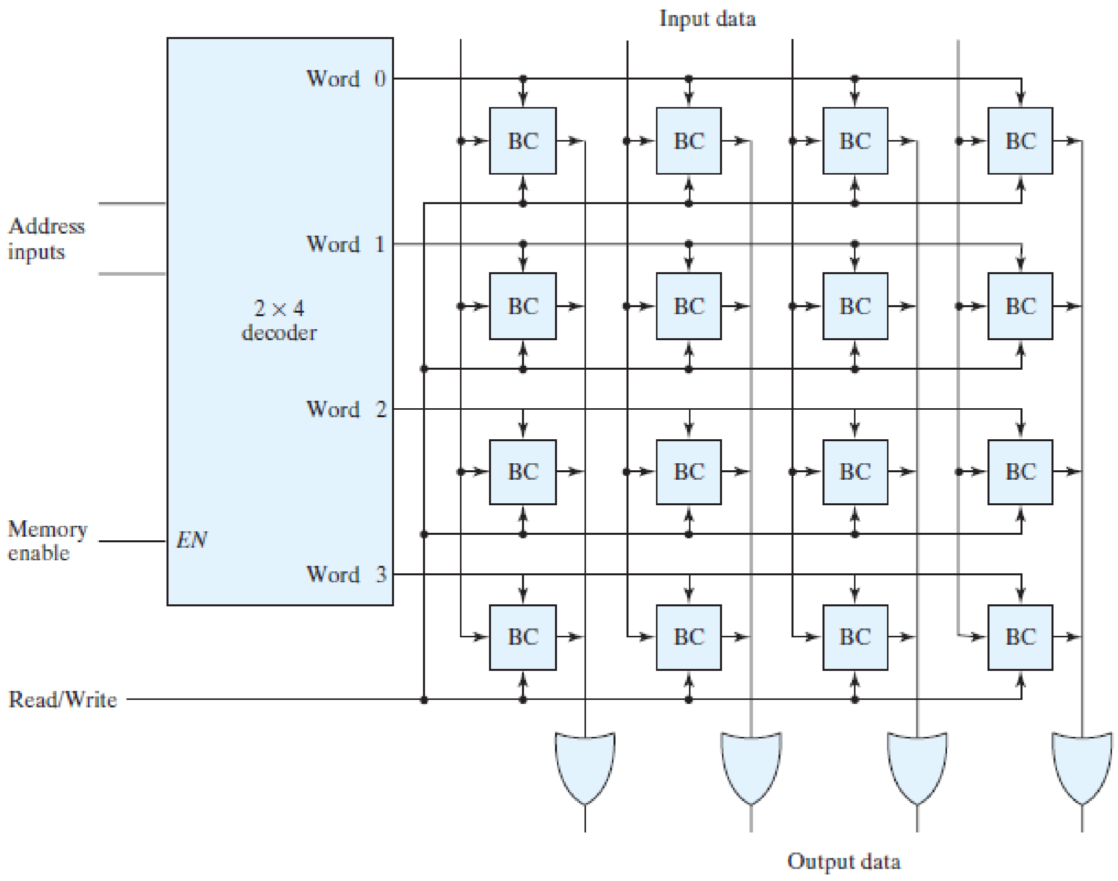


Stack with RAMs

- Two stack “functions”

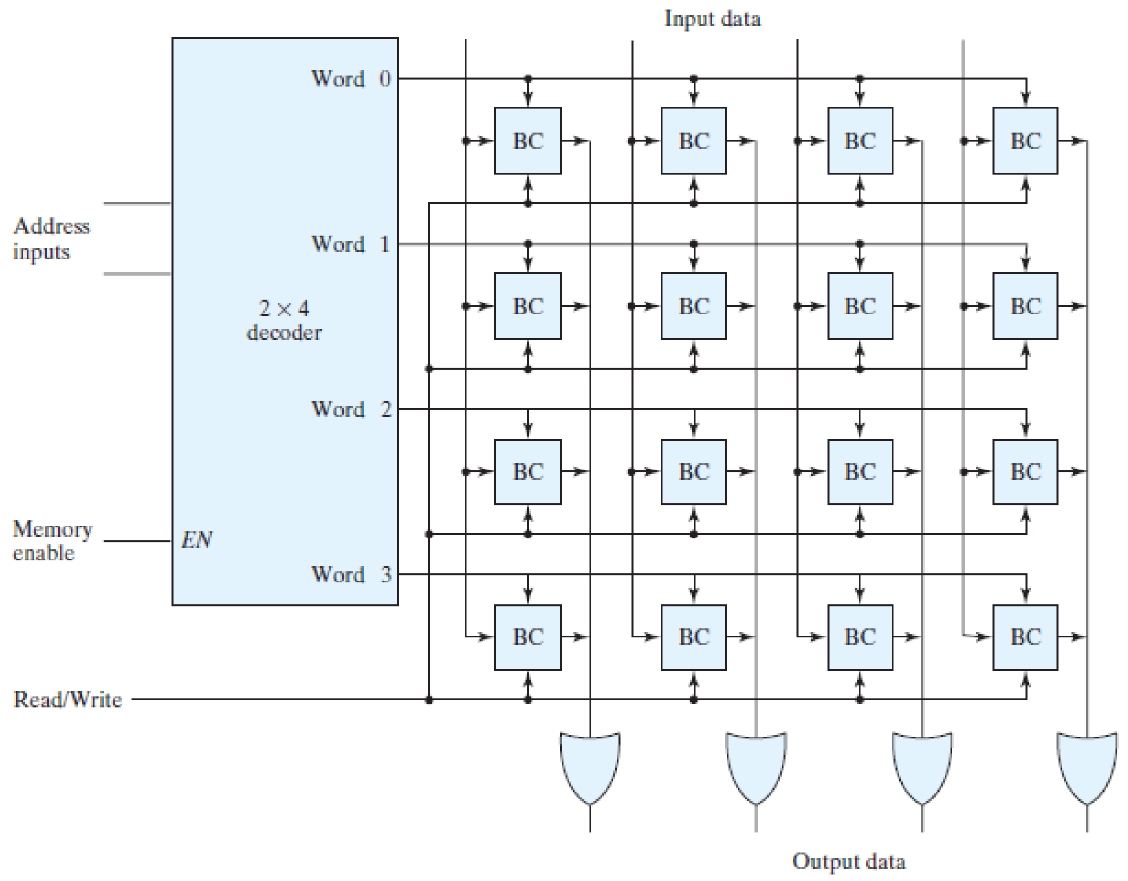
- push:
    - Adds element to stack
    - `push( 4’b XXXX)`

- pop:
    - Removes element from stack
    - `4’bXXXX = pop()`



Stack with RAMs

```
push( 4'b 0001)

push( 4'b 0010)

push( 4'b 0100)

push( 4'b 1000)
```



Stack with RAMs

push(4’b 0001)

head = 00, input = 0001, $Rd\overline{Wr}$ = 0, memEn = 1

head <= head + 1;

push(4’b 0010)

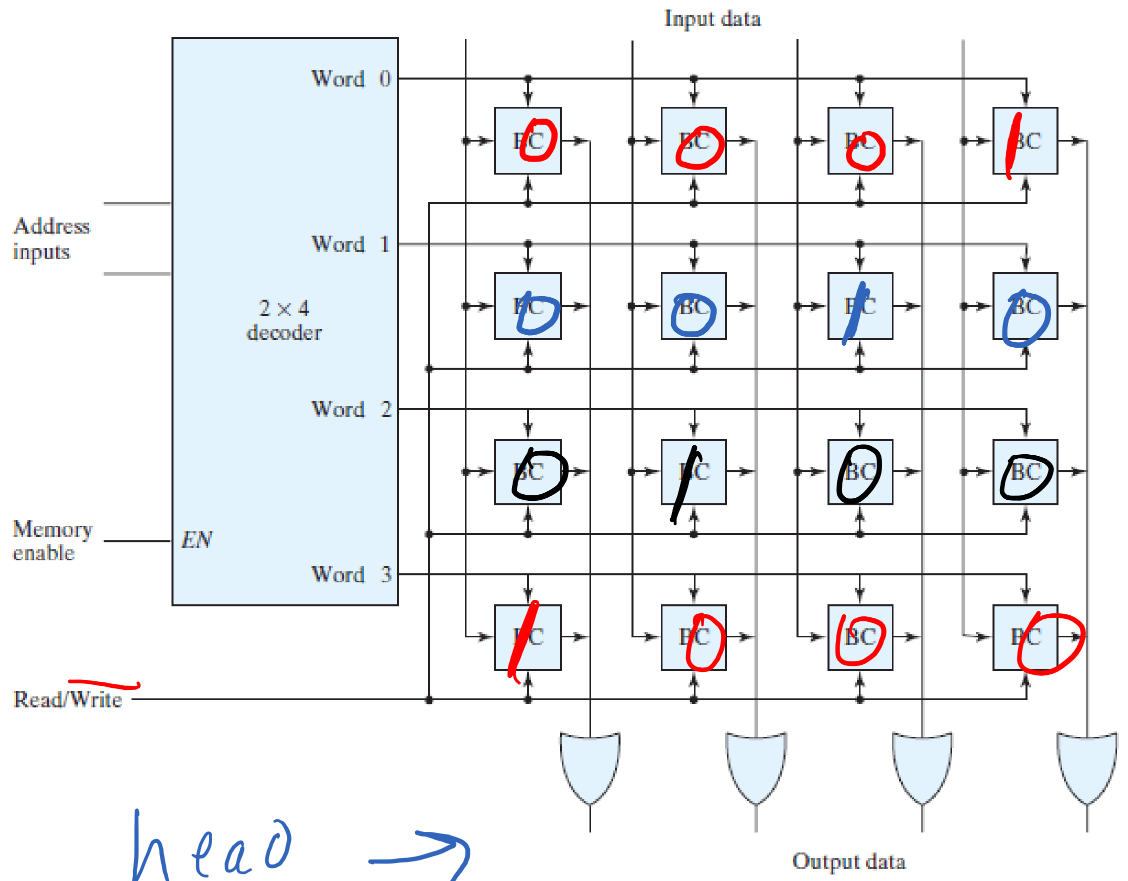
head =

head = 01, input = 0010, $Rd\overline{WR}$ = 0, memEn = 1

head <= head + 1

push(4’b 0100)

push(4’b 1000)

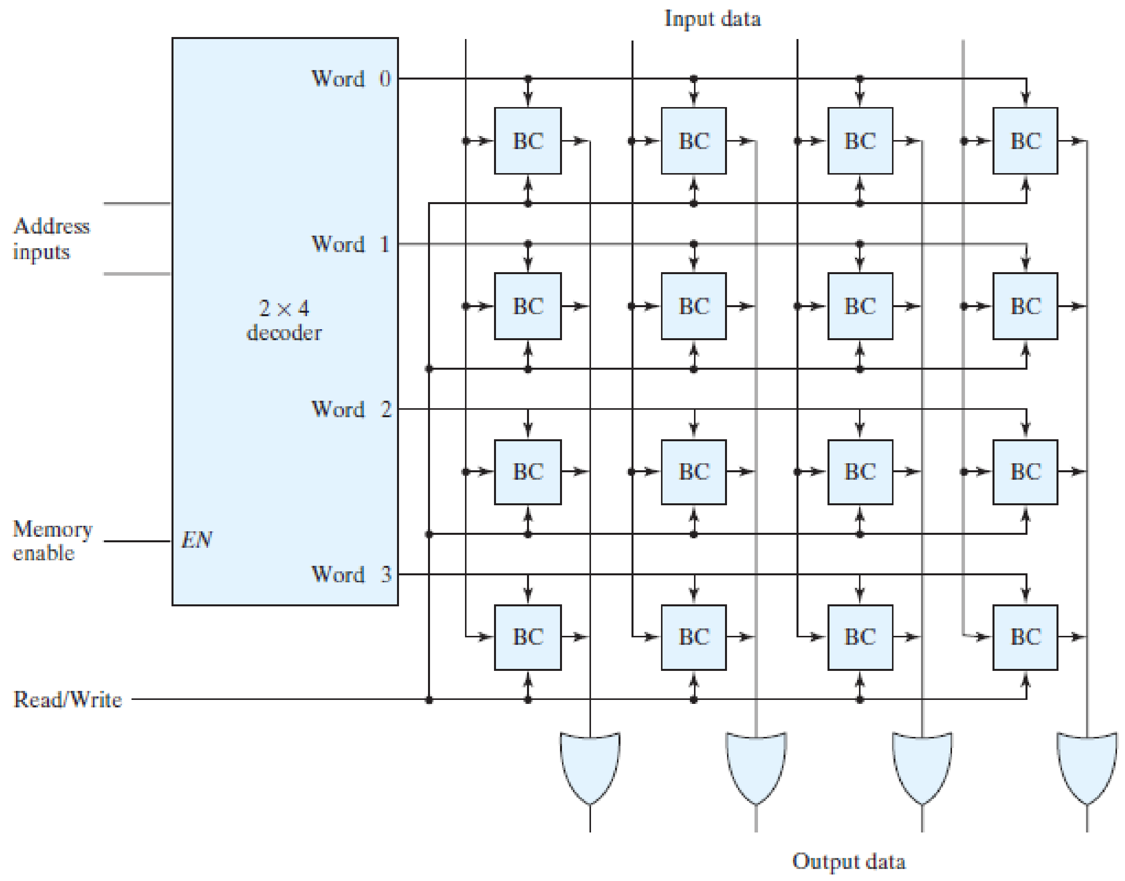


head $\rightarrow$

100

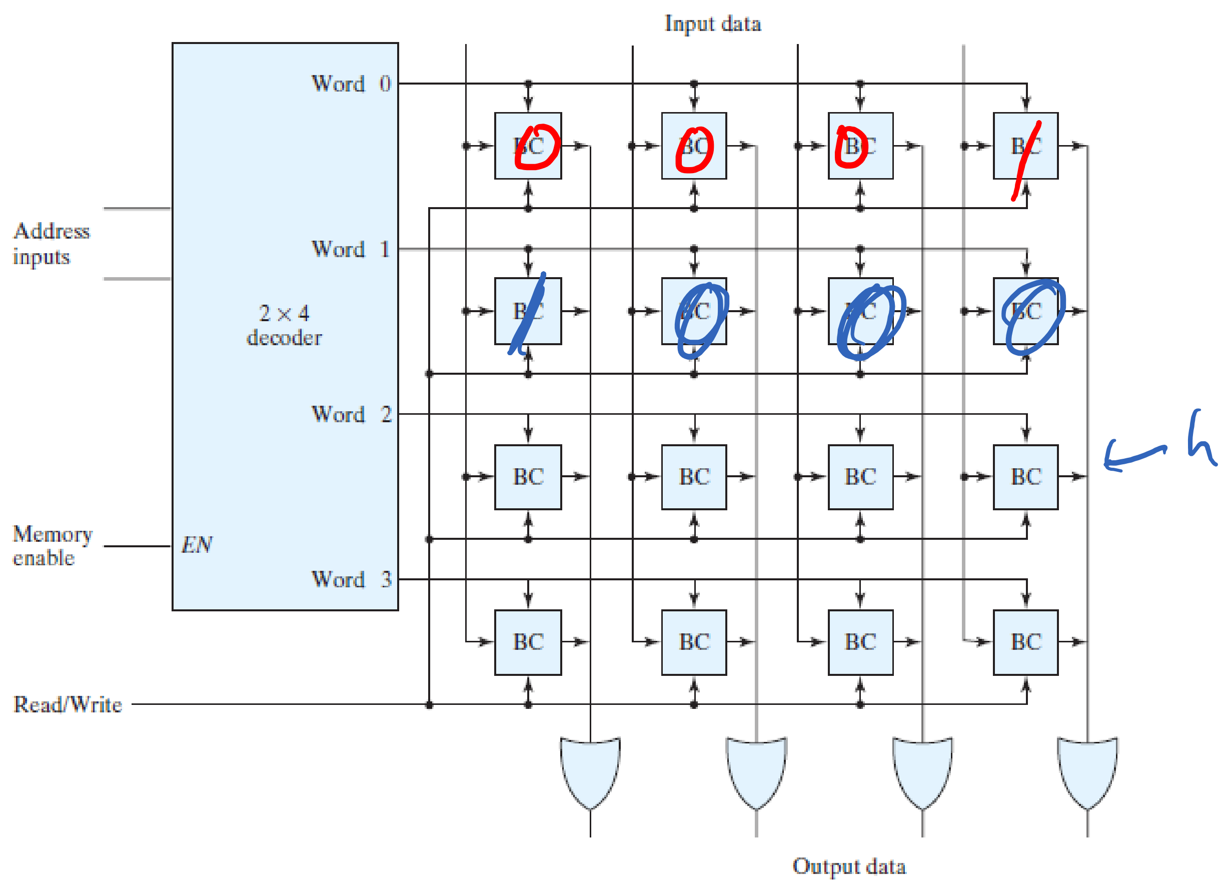
push/pop with RAMs

push(4’b 0001)
push(4’b 0010)
push(4’b 0100)
pop()
pop()
push(4’b 1000)
push(4’b 0011)
pop()
pop()
push(4’b 0110)
pop()



push/pop with RAMs

push(4'b 0001) ✓
push(4'b 0010) ✓
push(4'b 0100) ✓
pop() ⇒ 0100
pop() ⇒ 0010
push(4'b 1000) ✓
push(4'b 0011)
pop()
pop()
push(4'b 0110)
pop()



```
module RAM (
  input        clk,
  input [1:0] addr,
  input    set,
  input [3:0]  set_data,
  output [3:0] read_data
)
```

Stack Logic

- **Inputs:** `push_req, [3:0] push_data`
- **Inputs To RAM:** `addr, set, [3:0] set_data`

Push State Machine

```
module RAM (
  input        clk,
  input [1:0] addr,
  input     set,
  input [3:0]  set_data,
  output [3:0] read_data
)
```

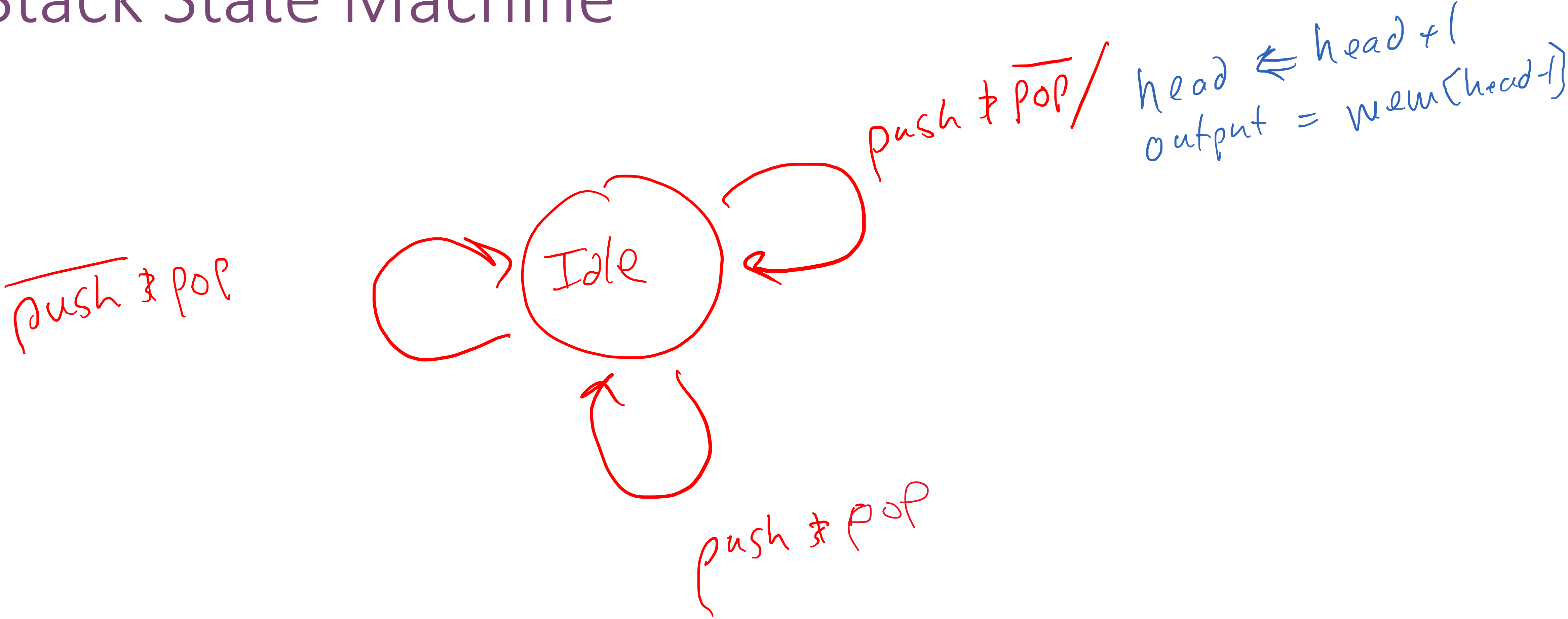
```
assign pop_data =                 ,

always_ff (@ posedge clk) begin
    if (rst) ....
    else begin

end          end
```

Pop Logic

- **Inputs:** `pop_req`
- **Outputs:** `[3:0] pop_data`
- **Inputs To RAM:** `addr, set`
- **From RAM:** `[3:0] read_data`

```
module RAM (
  input        clk,
  input [1:0] addr,
  input    set,
  input [3:0]  set_data,
  output [3:0] read_data
)
```

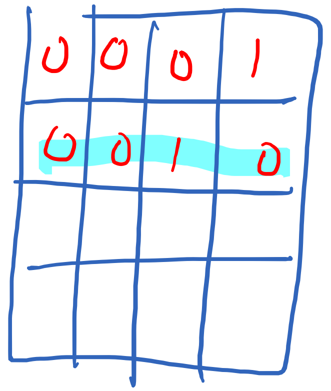
Stack State Machine


$\overline{\text{push}}$ & pop
Idle
push & $\overline{\text{pop}}$ /
head ⇐ head+1
output = mem[head-1]
push & pop

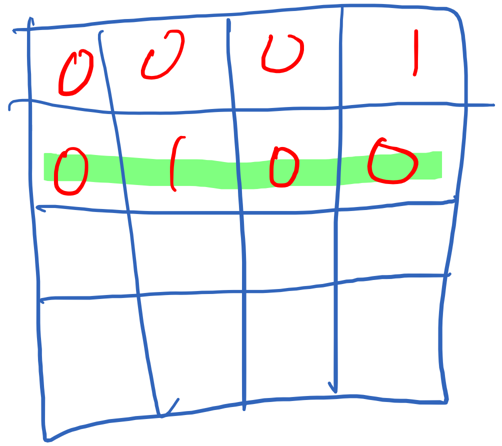
Error cases not shown.

Challenge: Push+Pop Together

- This needs to be a "replace" in the RAM.

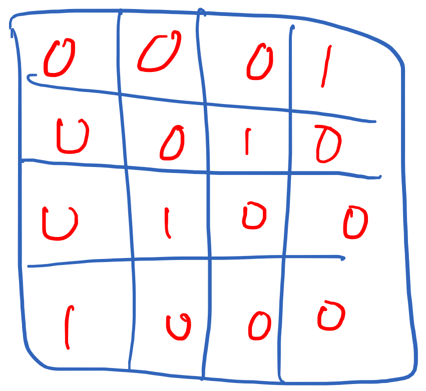


pop() + push(0100) $\Rightarrow$



Challenge: Push+Pop Error Logic

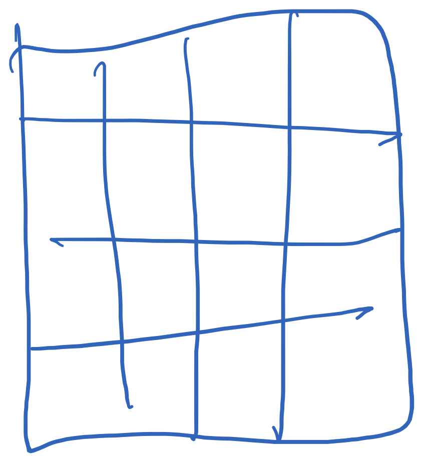
- What happens if the RAM is empty? Or Full?



push → fail

pop → succeed

push + pop → succeed



push → succeed

pop → fail

push+pop →

push-err = 0

pop-err = 1

Next Time

- FPGA Structures