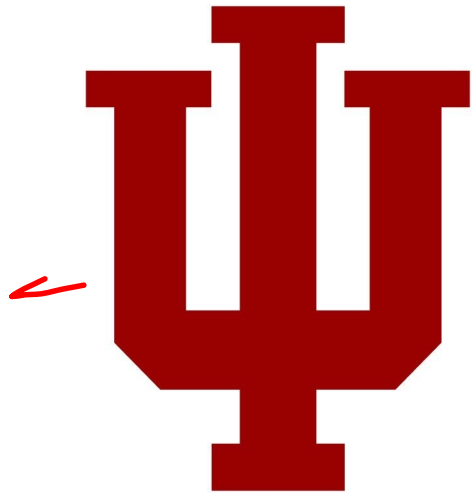
06: Memory-Mapped I/O

Engr 315: Hardware / Software Codesign

Andrew Lukefahr
Indiana University



Announcements

r/asset ↔ Slack

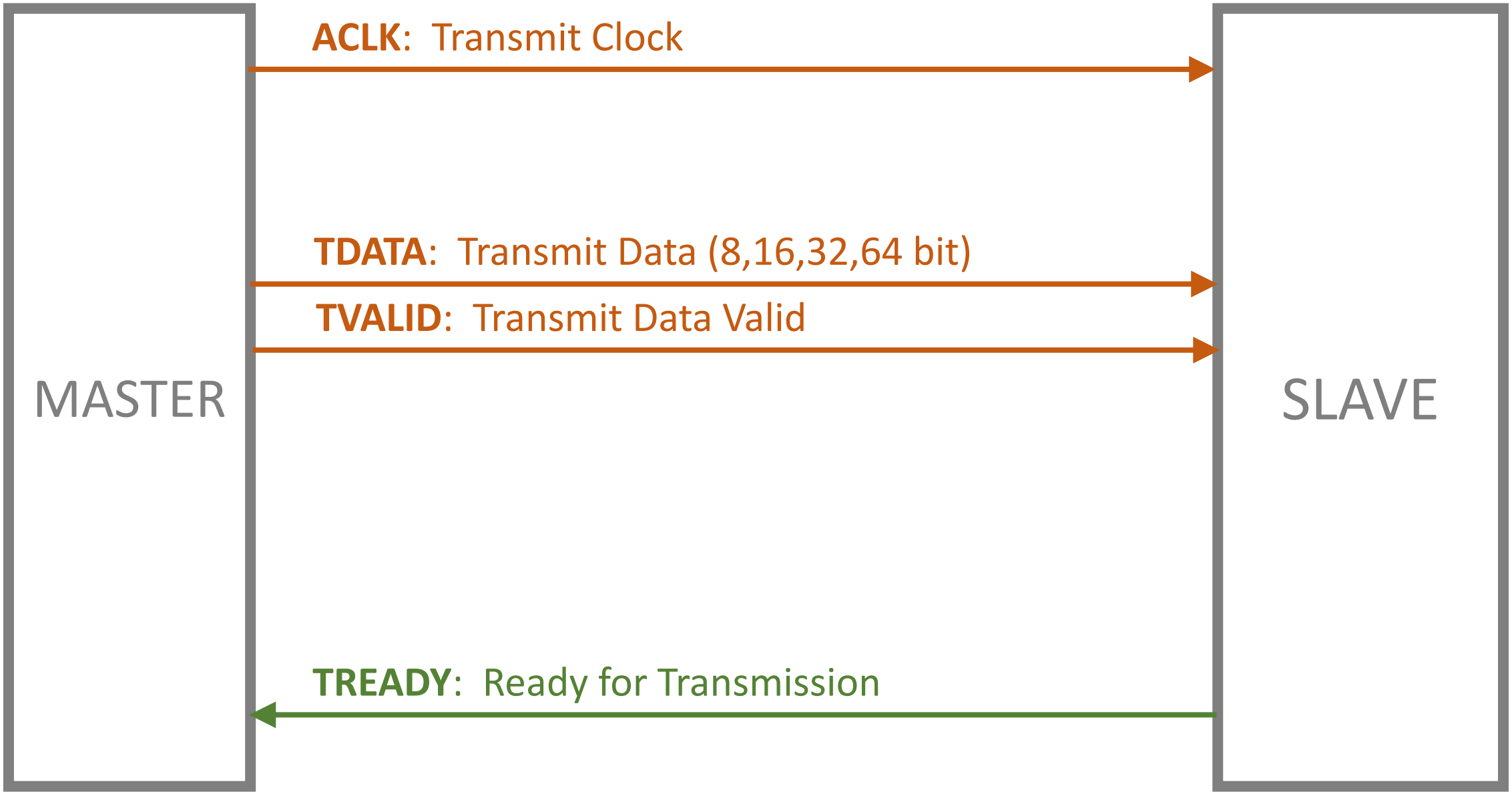
- Office Hours – See Website / Syllabus
  - *Maybe change Thursday due to conflict
- P2: Due Tonight
  - (New Project, could be some bumps)
  - Need a Pynq
  - Groups of 2 allowed
- P3: Out now!

Optimizations thus far

- Algorithmic complexity
- Removing redundant computation
- ~~Multithreading~~
- ~~Multiprocessing*~~
- Python/C/Asm Interfacing
- Map to Hardware

We could also map popcount to hardware

```
import cPopcount
print (cPopcount.cPopcount(0))

import hwPopcount
print hwPopcount.hwPopcount(0))
```


ACLK: Transmit Clock
TDATA: Transmit Data (8,16,32,64 bit)
TVALID: Transmit Data Valid
MASTER
SLAVE
TREADY: Ready for Transmission

Data (TDATA) is only transferred when

TVALID is **1.**

This indicates the **MASTER** is trying to transmit new data.

TREADY is **1.**

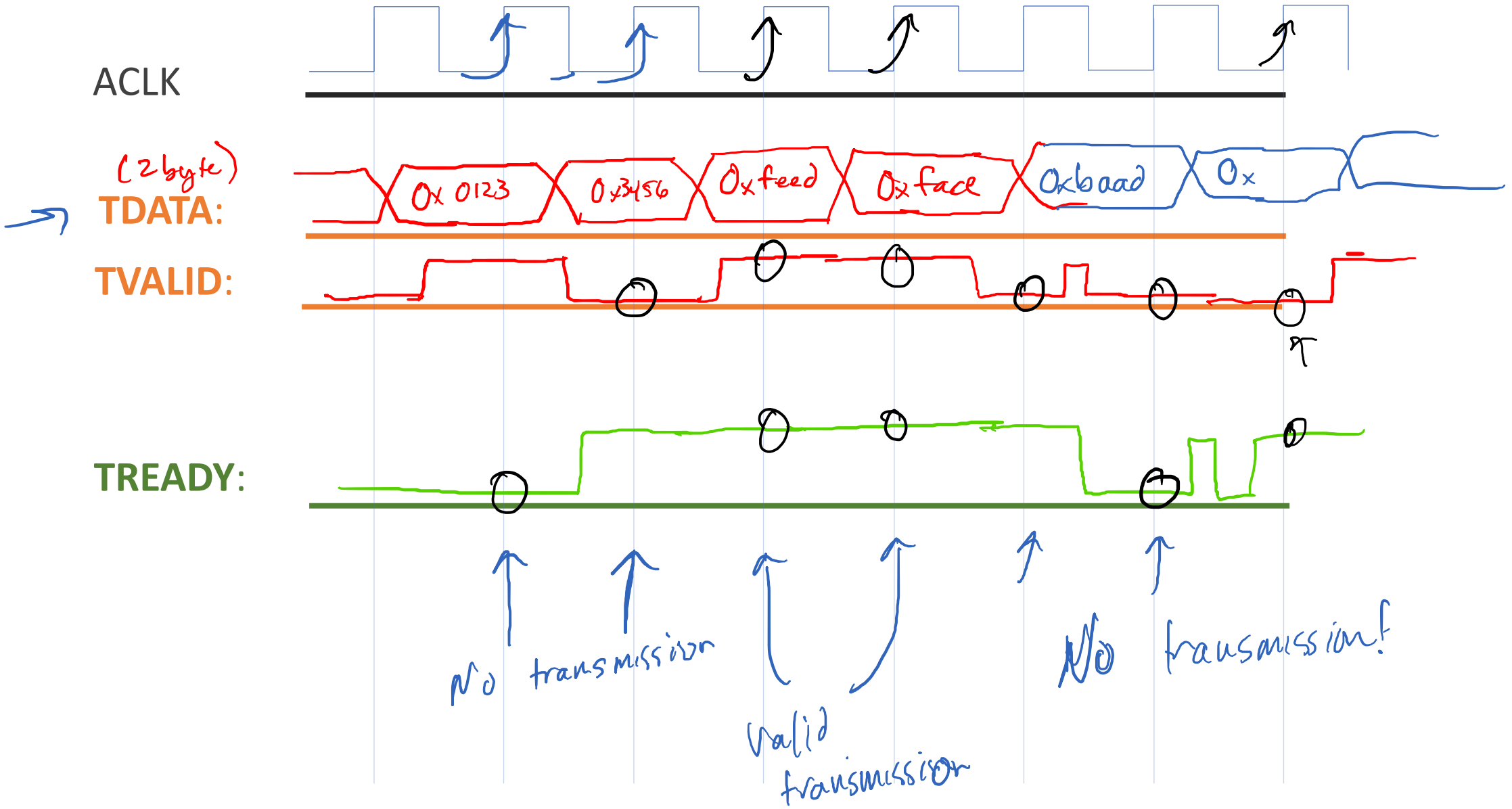
This indicates the **SLAVE** is ready to receive the data.

If either **TVALID** or **TREADY** are 0, **no data is transmitted**.

If **TVALID** and **TREADY** are 1, **TDATA** is transmitted

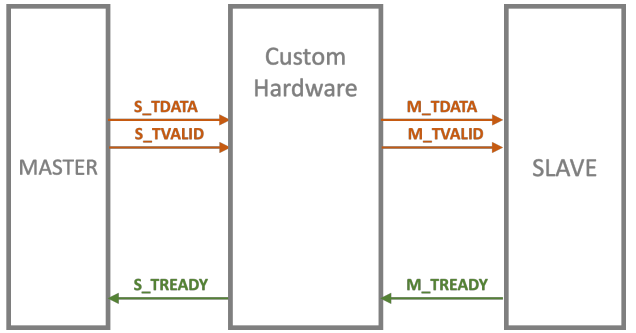
at the positive edge of ACLK

Transferring data on a AXI4-Stream Bus.

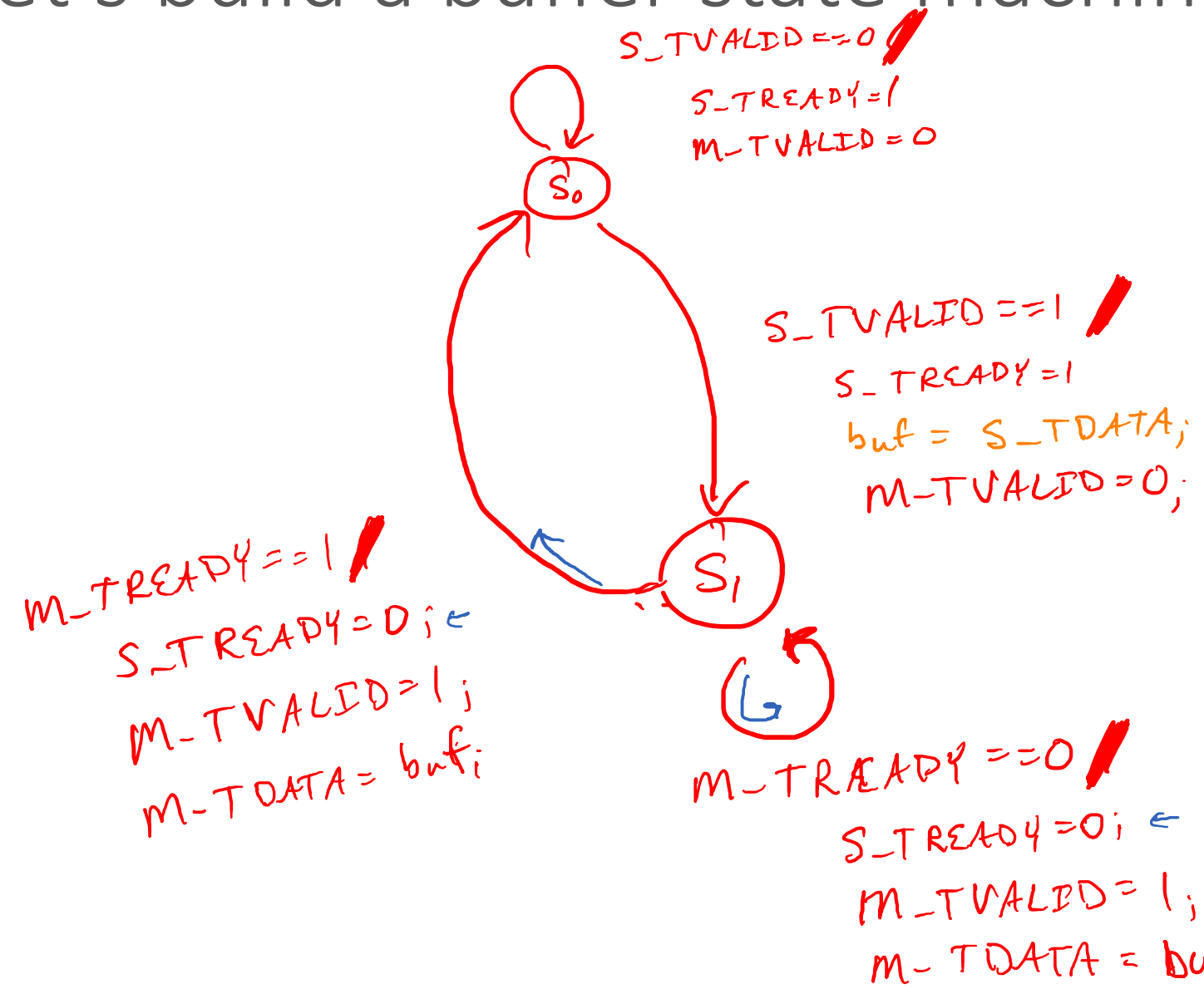


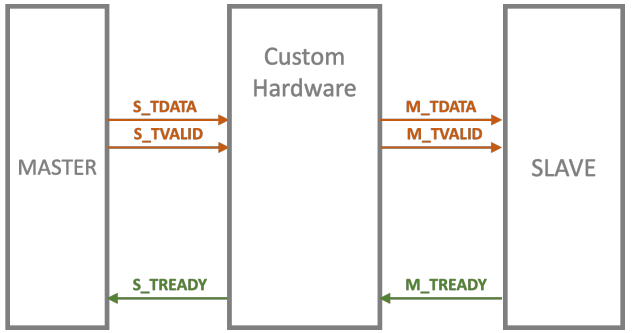
How would I flip all the bits of TDATA?

```
module custom_hw (
        input           ACLK,
        input           ARESET,
        input [31:0]    S_TDATA,
        input           S_TVALID,
        output          S_TREADY,
        output [31:0]   M_TDATA,
        output          M_TVALID,
        input           M_TREADY
);

assign M_TDATA = ~S_TDATA;
assign M_TVALID = S_TVALID;
assign S_TREADY = M_TREADY;

endmodule
```

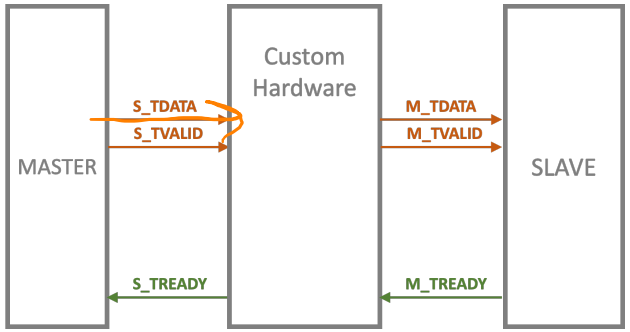


Let's build a buffer state machine.





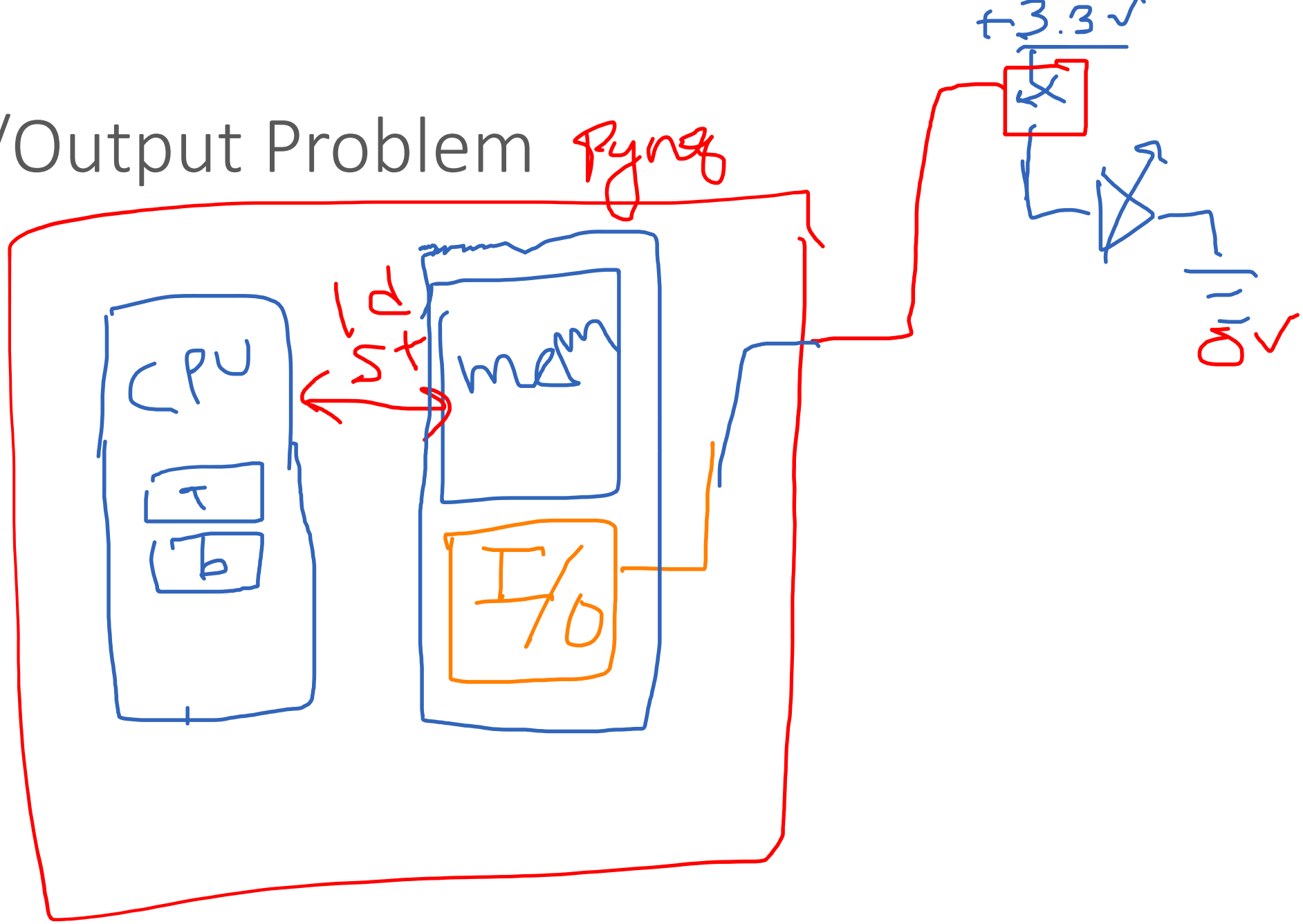
Let's build a buffer state machine.



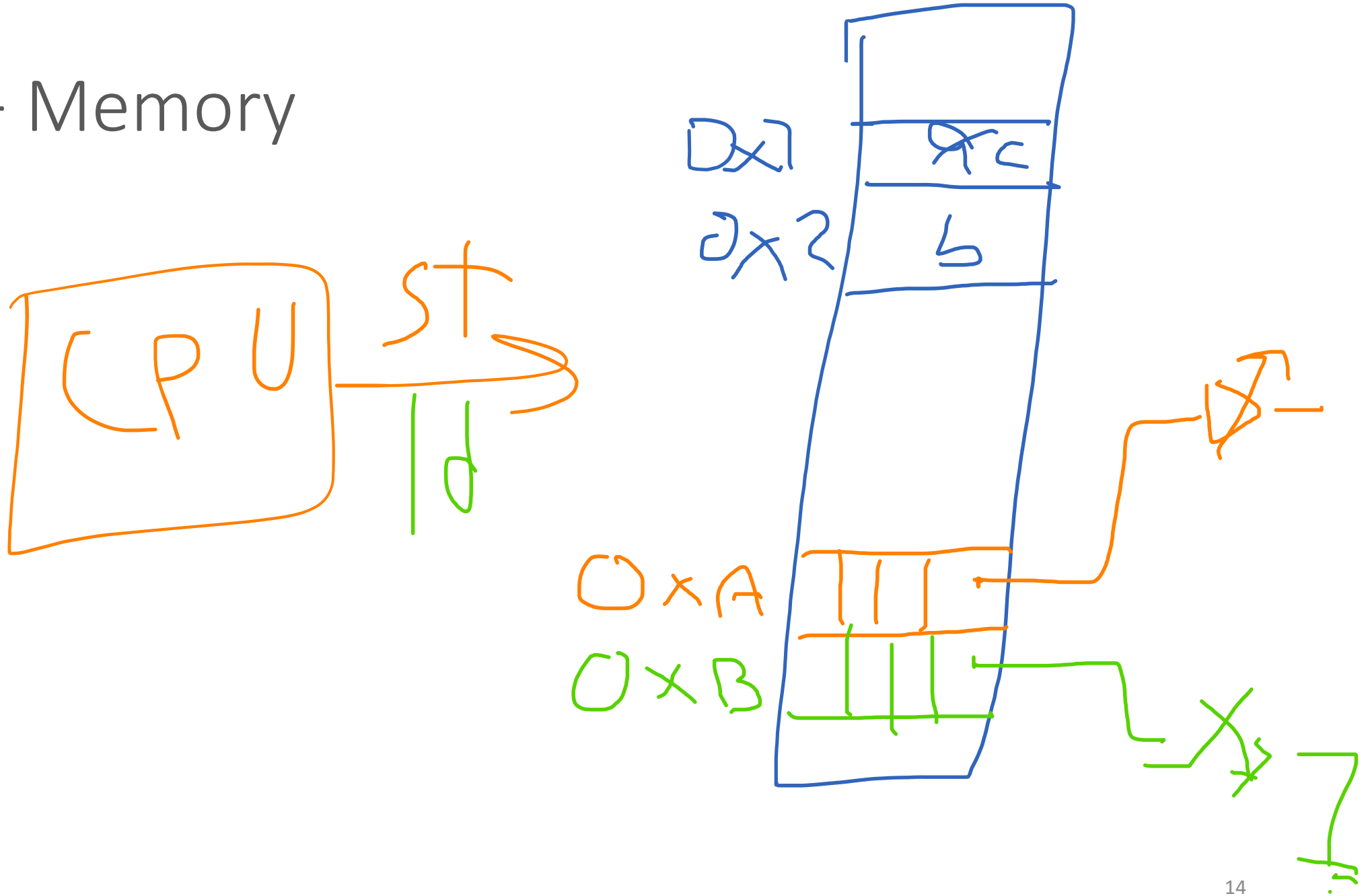

```
module custom_hw_buf (
        input           ACLK,
        input           ARESET,
        input [31:0]    S_TDATA,
        input           S_TVALID,
        output          S_TREADY,
        output [31:0]   M_TDATA,
        output          M_TVALID,
        input           M_TREADY
);

enum {S0, S1} state, nextState;
reg [31:0] nextVal;

always_ff @(posedge ACLK) begin
    if (ARESET)begin
        state <= S0;
        M_TDATA <= 32'h0
    end else begin
        state <= nextState;
        M_TDATA <= nextVal;
    end
end
```

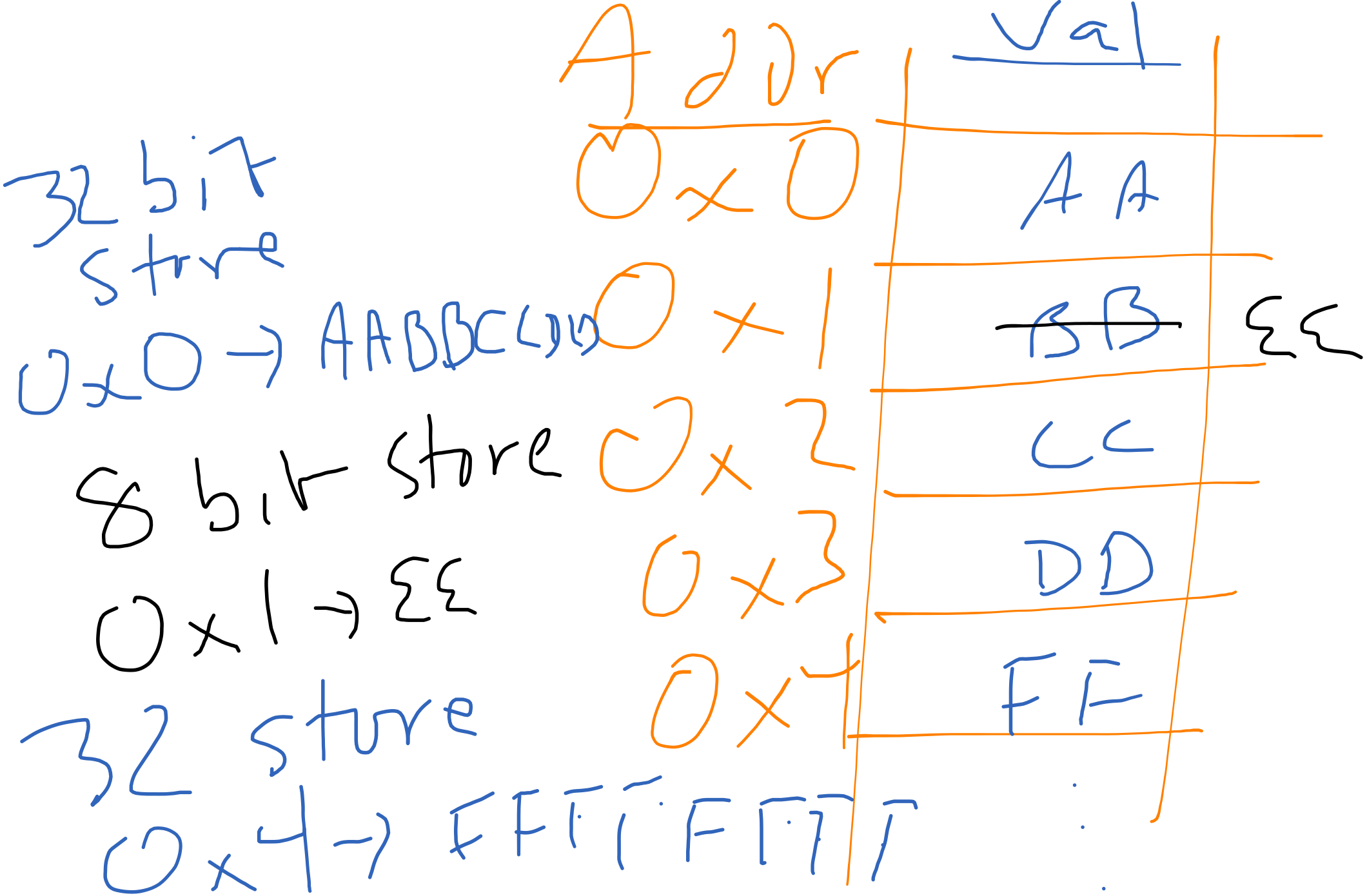
```
always_comb begin
    S_TREADY = 'h1;
    M_TVALID = 'h0;
    nextState = state;
    nextVal = M_TDATA;
    case(state)
        S0:  begin
            if (S_TVALID) begin
                nextState = S1;
                nextVal = S_TDATA;
            end
        end
        S1: begin
            S_TREADY = 'h0;
            M_TVALID = 'h1;
            if (M_TREADY) begin
                nextState = S0;
        end
    endcase
end

endmodule
```

The Input/Output Problem



CPUs + Memory




32 bit
store
0x0 → AABBCCDD
8 bit store
0x1 → EE
32 store
0x4 → FFFFFFFF
Addr
Val
0x0
AA
0x1
BB
EE
0x2
CC
0x3
DD
0x4
FF

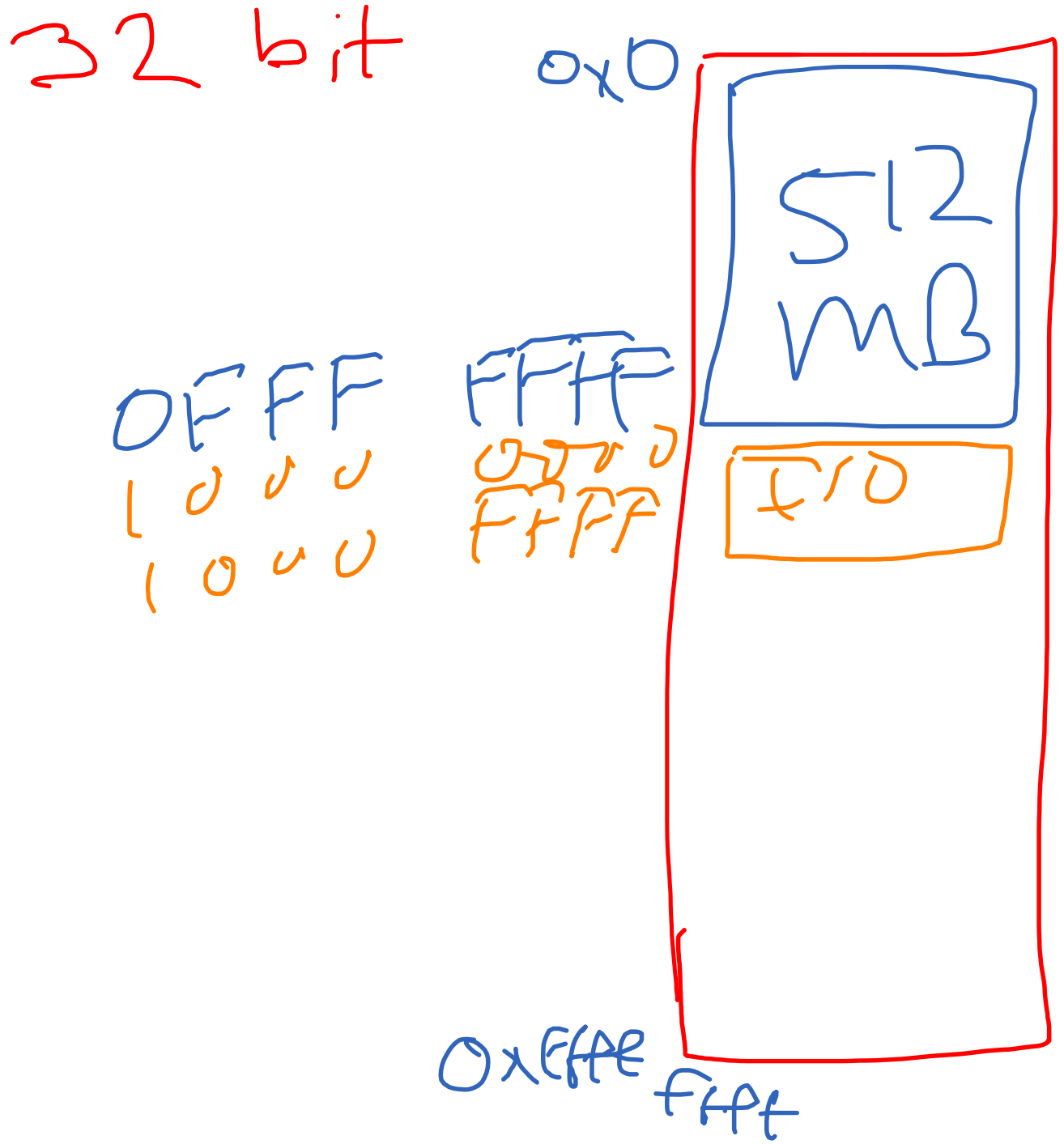
8bit Addr

00000000
00000001
...
11111110
11111111

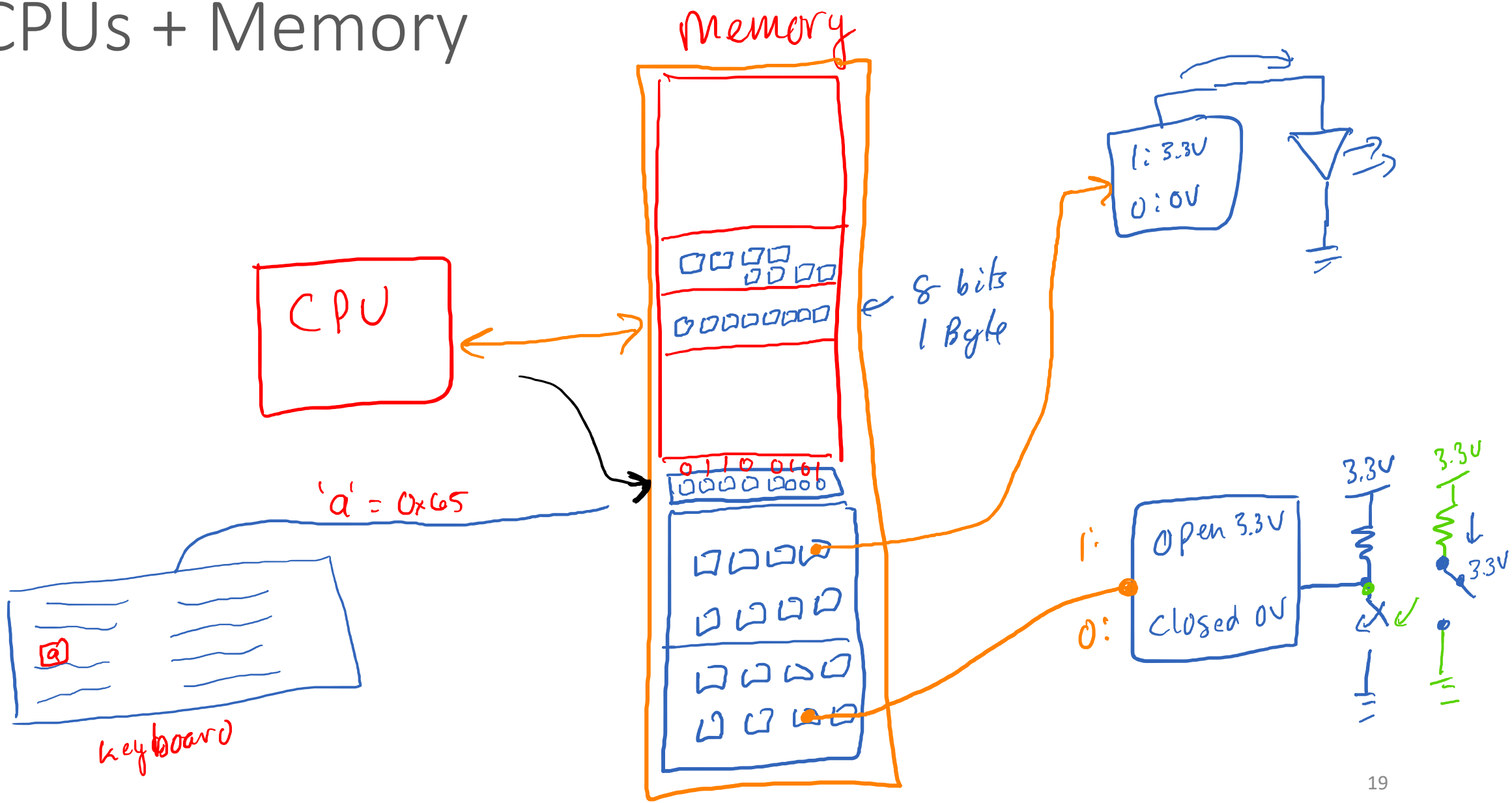
$2^8 = 256$

32bit 0000 0000 0000 0000 0000 0000 0000 0000

$2^{32} = \sim 4GB$ / $2^{64} = ?$


32 bit
0x0
512
MB
OFFF FFFF
1000 0000
1000 FFFF
1000
0xffff ffff

CPUs + Memory


Memory
CPU
8 bits
1 Byte
'a' = 0x65
a
keyboard
1: 3.3V
0: 0V
1:
Open 3.3V
0:
Closed 0V
3.3V
3.3V
3.3V

Memory-Mapped I/O

Goal: Connect I/O to memory address

Memory-Mapped I/O

- I/O devices pretend to be memory
- “Pretend Memory” I/O accessed with native CPU load/store instructions

MMIO from Assembly

- First, we need to see ARM assembly…

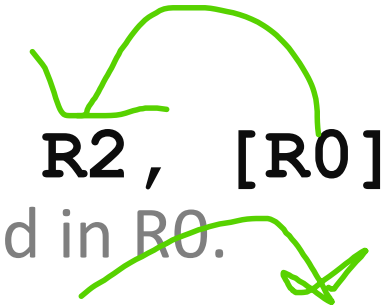
ARM Registers

ARM	Description	x86
R0	General Purpose	EAX
R1-R5	General Purpose	EBX, ECX, EDX, ESI, EDI
R6-R10	General Purpose	–
R11 (FP)	Frame Pointer	EBP
R12	Intra Procedural Call	–
R13 (SP)	Stack Pointer	ESP
R14 (LR)	Link Register	–
R15 (PC)	<- Program Counter / Instruction Pointer ->	EIP
CPSR	Current Program State Register/Flags	EFLAGS

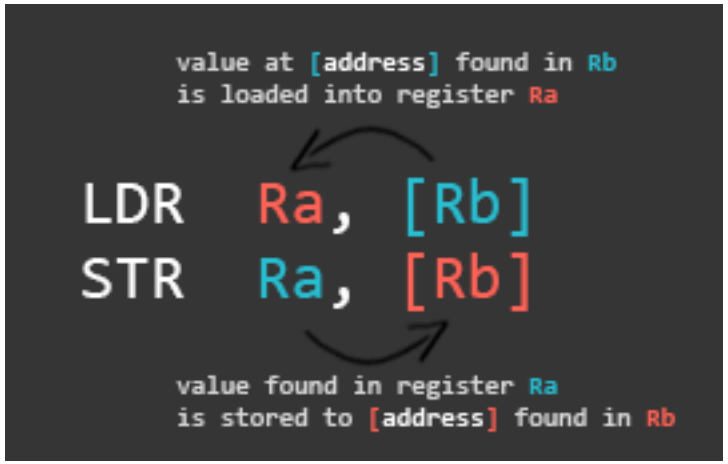
ARM Instructions

Instruction	Description	Instruction	Description
MOV	Move data	EOR	Bitwise XOR
MVN	Move and negate	LDR	Load
ADD	Addition	STR	Store
SUB	Subtraction	LDM	Load Multiple
MUL	Multiplication	STM	Store Multiple
LSL	Logical Shift Left	PUSH	Push on Stack
LSR	Logical Shift Right	POP	Pop off Stack
ASR	Arithmetic Shift Right	B	Branch
ROR	Rotate Right	BL	Branch with Link
CMP	Compare	BX	Branch and eXchange
AND	Bitwise AND	BLX	Branch with Link and eXchange
ORR	Bitwise OR	SWI/SVC	System Call

ARM Load + Store



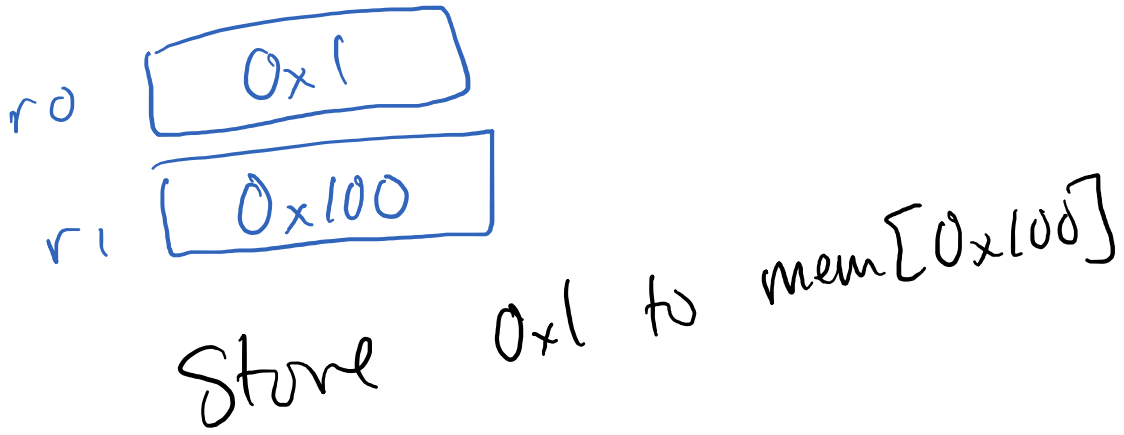
```
LDR R2, [R0]   @ [R0] - origin address is the value found in R0.
STR R2, [R1]   @ [R1] - destination address is the value found in R1.
```

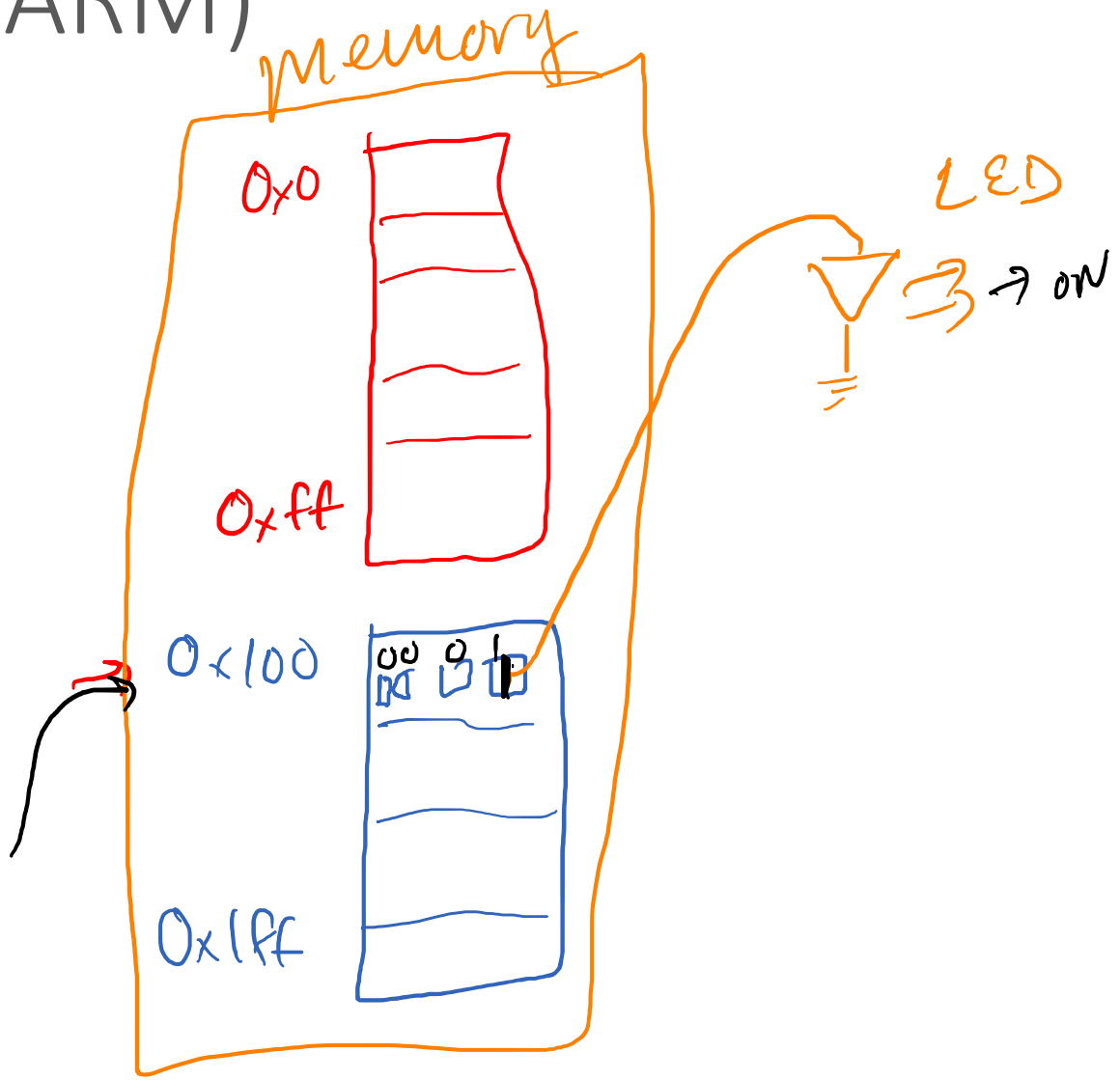


MMIO Store from ASM (ARM)

MMIO Store from ASM (ARM)

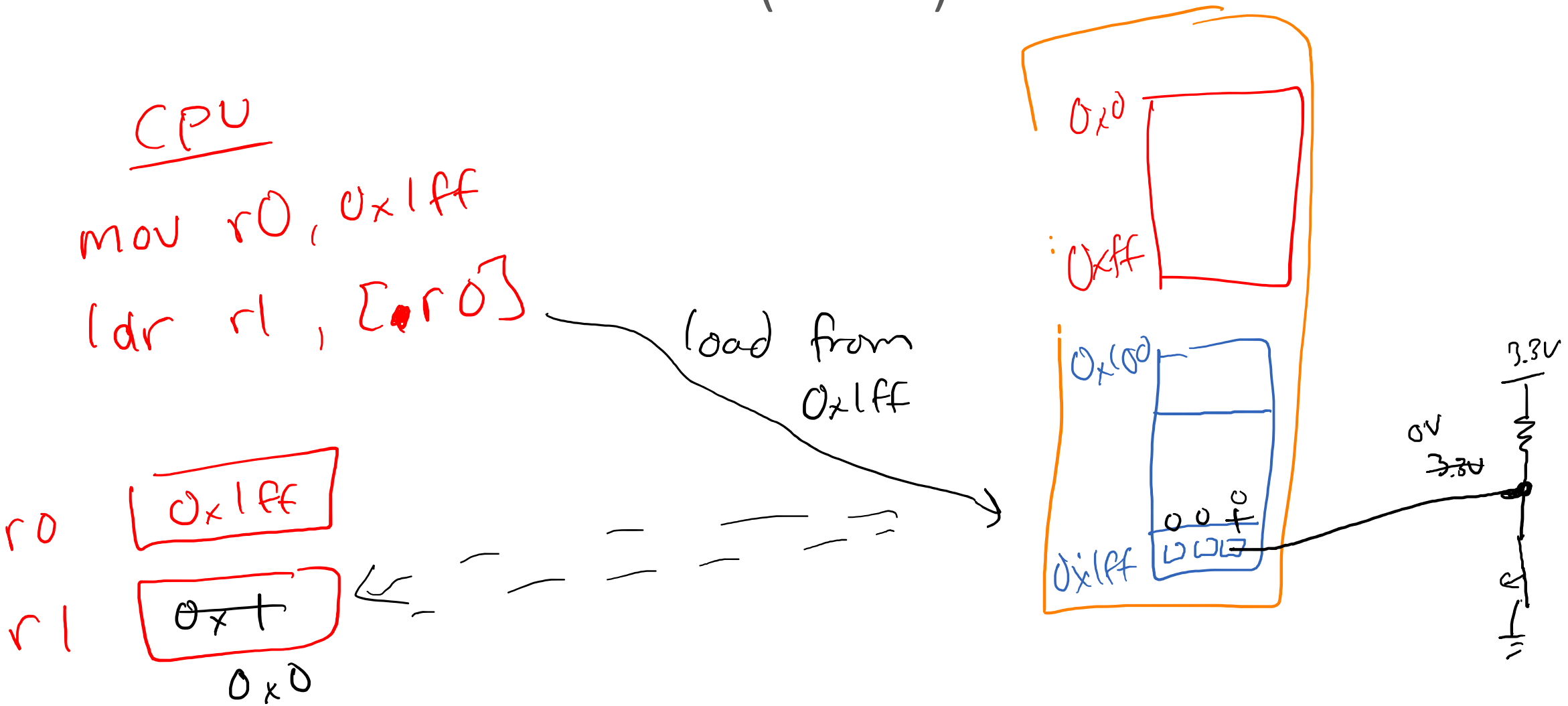
```
mov r0, 0x1
mov r1, 0x100
str r0, [r1]
```





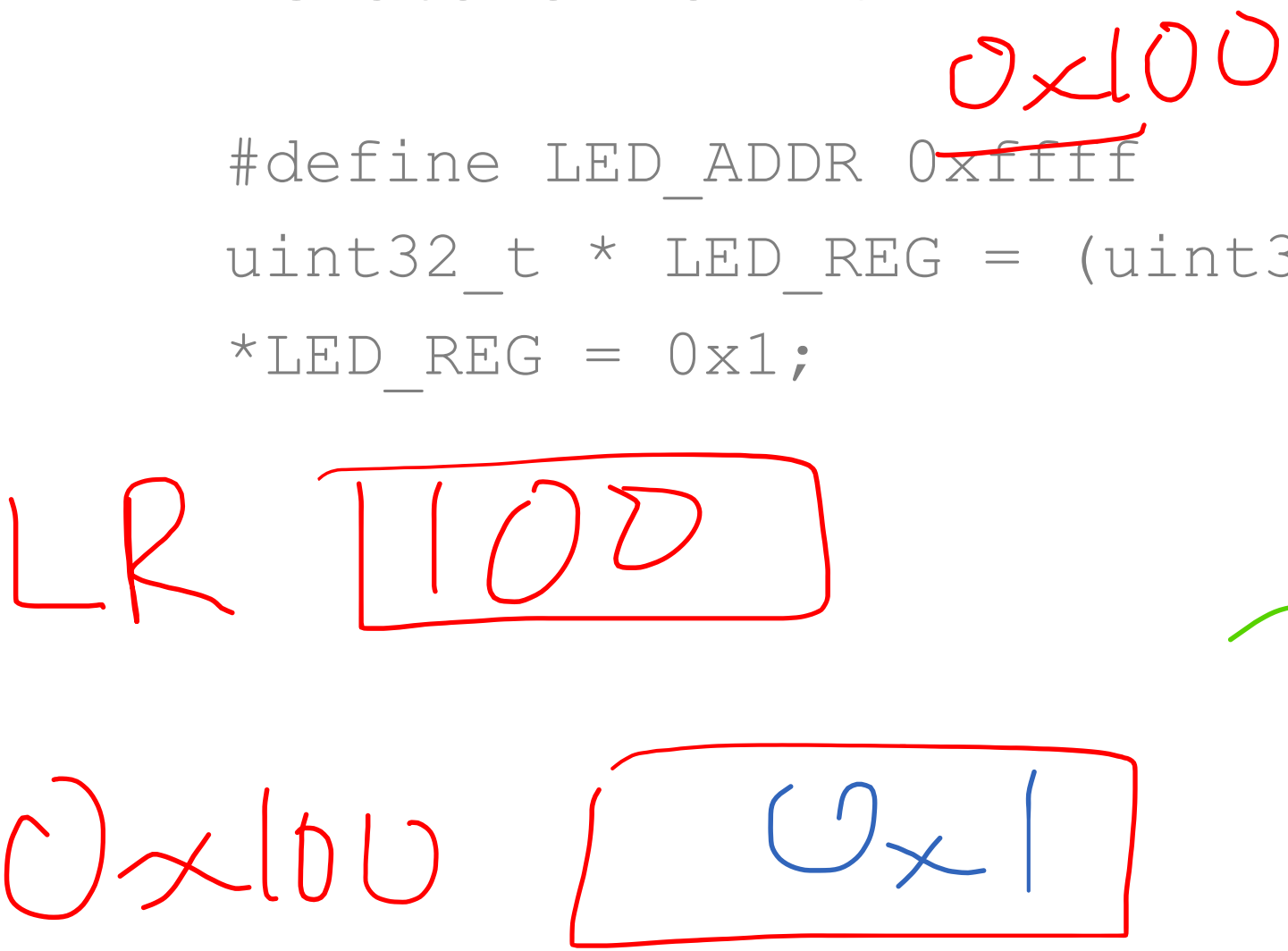
MMIO Load from ASM (ARM)

MMIO Load from ASM (ARM)



MMIO Store from C

```
#define LED_ADDR 0xffff
uint32_t * LED_REG = (uint32_t *)(LED_ADDR);
*LED_REG = 0x1;
```



MMIO Store from C

```
#define LED_ADDR 0xffff
uint32_t * LED_REG = (uint32_t *)(LED_ADDR);
*LED_REG = 0x1;
```

MMIO Load from C

MMIO Load from C

stop

```
#define SW_ADDR 0xfffe
uint32_t * SW_REG = (uint32_t *)(SW_ADDR);
int y = (*SW_REG);
```

cast value as an address

dereference to get memory at the address

Problem: Does `quit` ever change here? Do I need to recompute `(!quit)`? (-O3 edition)

```
int y = 0;

int quit = y;
while(!quit)
{
    //more code
    quit = y;
}
```

Problem: Does `quit` ever change here? Do I need to recompute `(!quit)`? (-O3 edition)

```
int y = 0;
int quit = y;
while(!quit)
{
    //your code
    quit = y; //What if y is a switch?

}
```

What about here?

```
int y = 0;
uint32_t * SW_REG = &y;

int quit = (*SW_REG);
while(!quit)
{
    //your code
    quit = (*SW_REG);
}
```

What about here?

```
int y = 0;
uint32_t * SW_REG = &y;

int quit = (*SW_REG);
while(!quit)
{
    //your code
    quit = (*SW_REG);
}
```

Use `volatile` for MMIO addresses!

```
#define SW_ADDR 0xfffe
volatile uint32_t * SW_REG = (uint32_t * SW_ADDR);

int quit = (*SW_REG);
while(!quit)
{
    //more code
    quit = (*SW_REG);
}
```

`volatile` Variables

- `volatile` keyword tells compiler that the memory value is subject to change randomly.

- **Use `volatile` for all MMIO memory. The values change randomly!**

Use `volatile` for all MMIO memory.

What happens here?

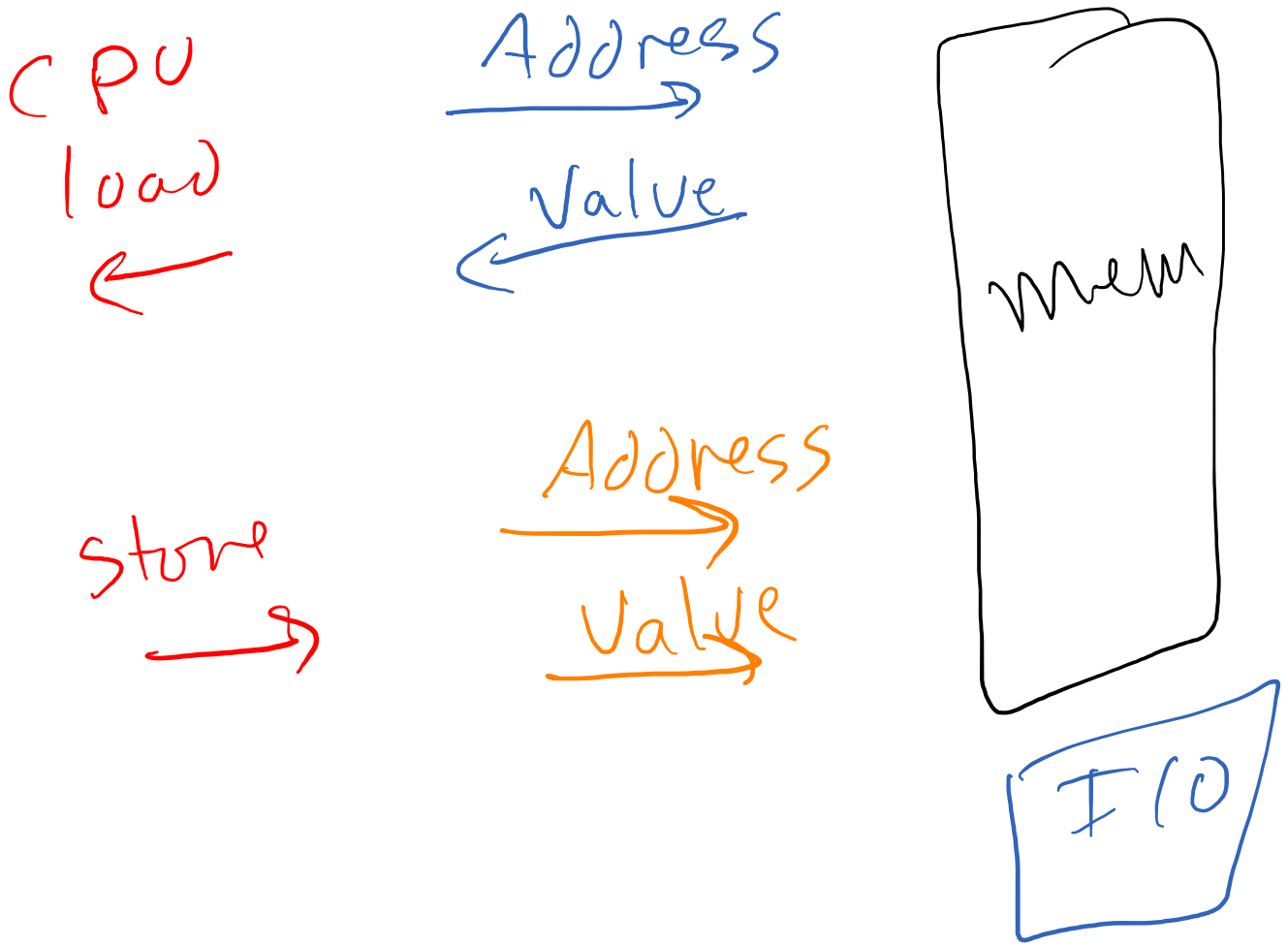
```
#include <stdio.h>
#include <inttypes.h>

#define REG_FOO 0x40000140

int main () {
  volatile uint32_t *reg = (uint32_t *)(REG_FOO);
  *reg += 3;

  printf("0x%x\n", *reg); // Prints out new value
}
```

Let’s find out…

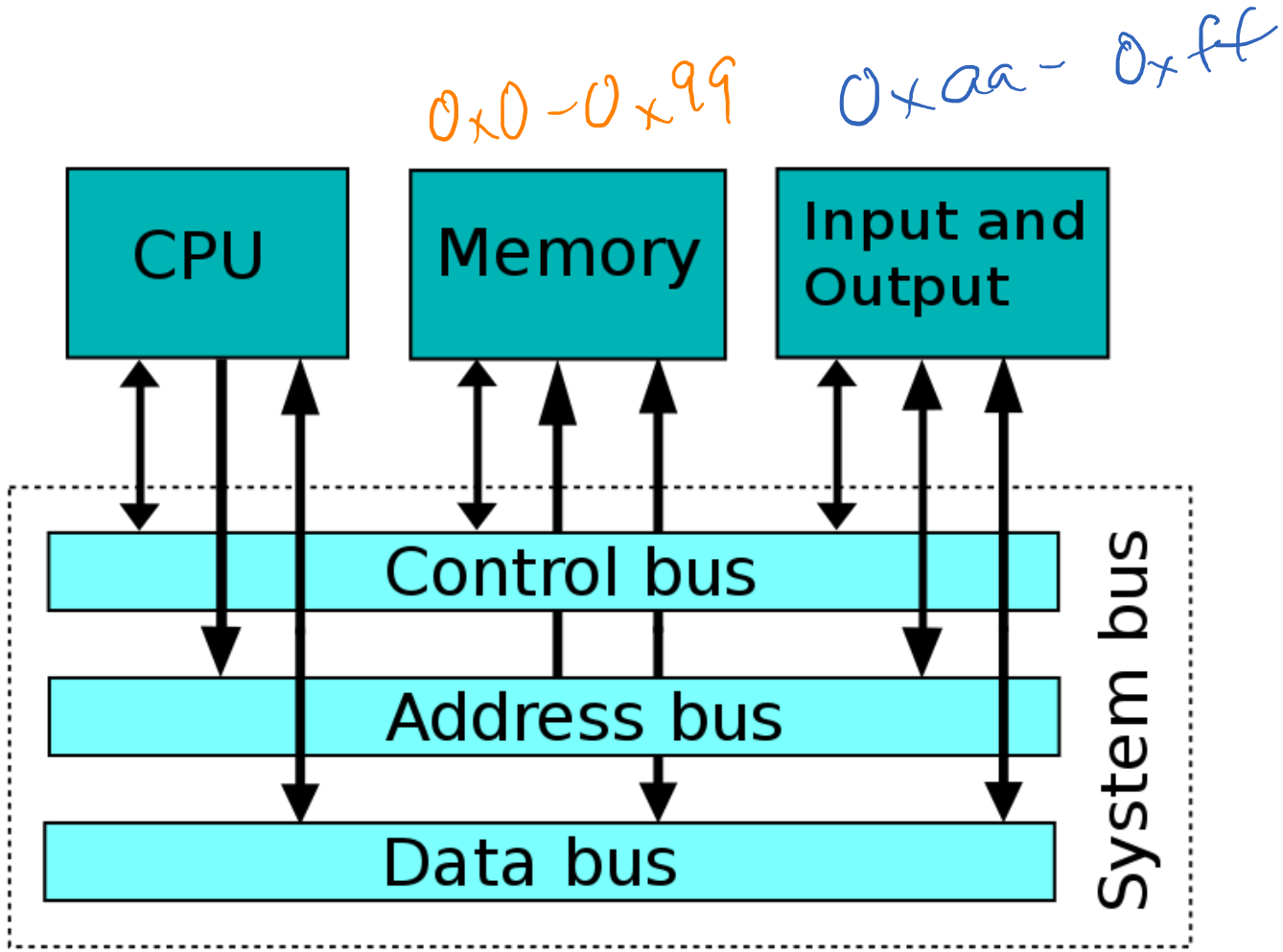
```
vi test.c
arm-none-eabi-gcc -o test.o test.c -g
                                -O1 --specs=nosys.specs
arm-none-eabi-objdump -DS test.o > test.dis
```

What do the CPU and Memory need to communicate?

or I/O



The System Bus



Next Time

- Combine MMIO + AXI Bus

06: Memory-Mapped I/O

Engr 315: Hardware / Software Codesign

Andrew Lukefahr
Indiana University