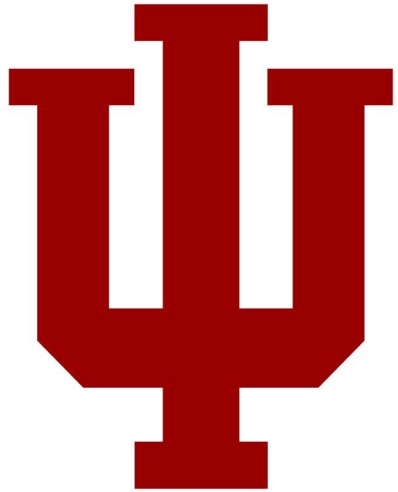
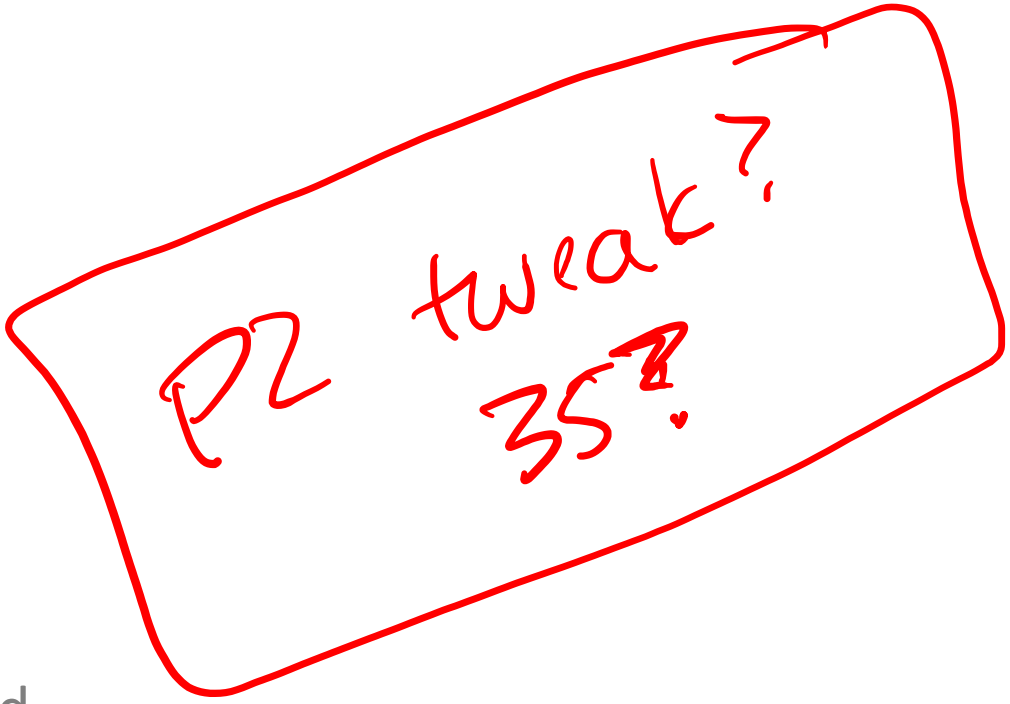
06: Memory-Mapped I/O

Engr 315: Hardware / Software Codesign

Andrew Lukefahr
Indiana University



Announcements



- P2: Due Tonight
  - Need a Pynq
  - Groups of 2 allowed
- P3: Out now!
  - Dates fixed.

AG →

- Guest Lecturer on Monday

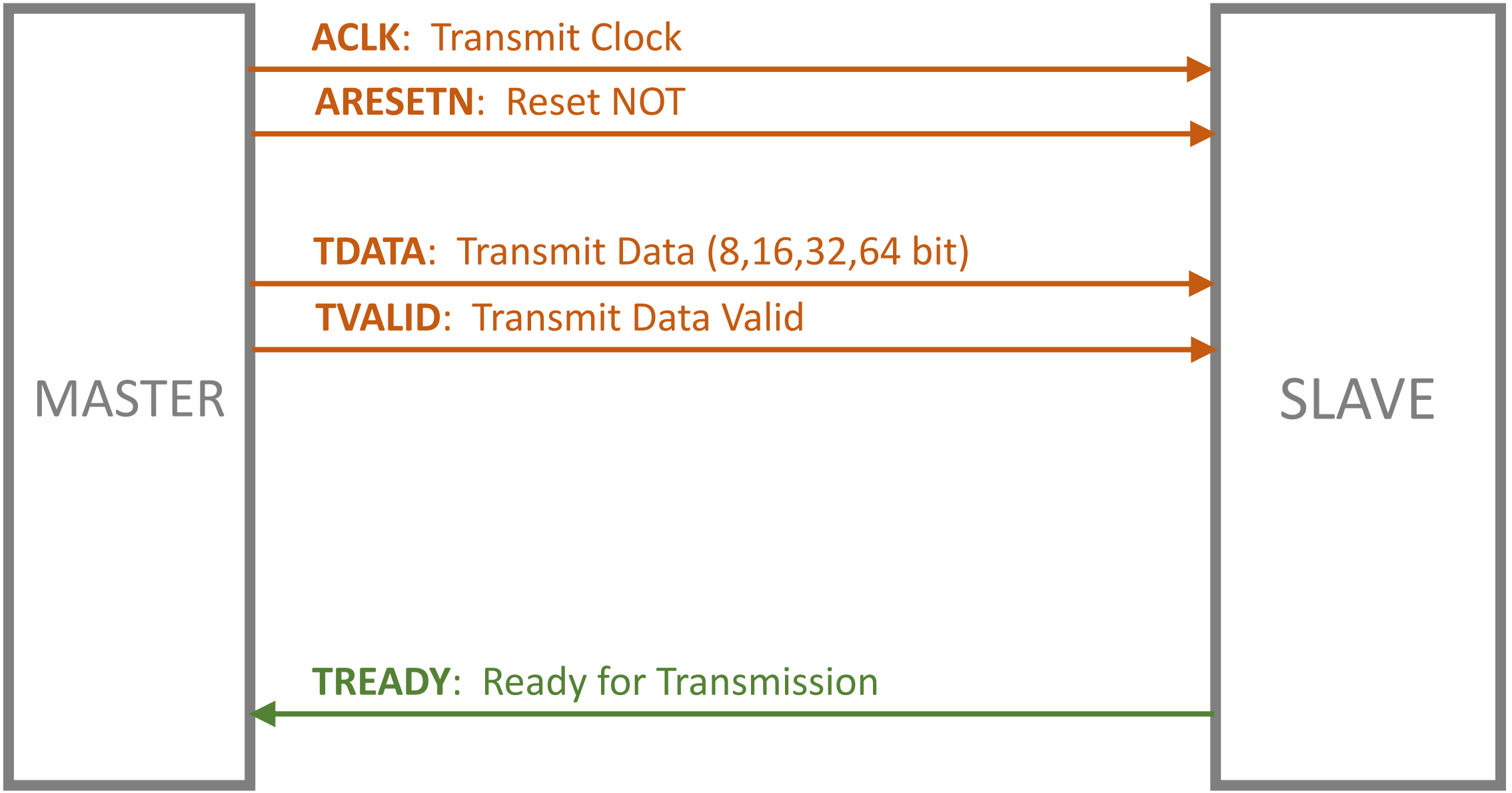
Chris

Optimizations thus far

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

P3 maps EMA to hardware

```
import cEMA
print (cEMA.cEMA(0))

import hwEMA
print (hwEMA.hwEMA(0))
```



AXI Stream Interface

Data (TDATA) is only transferred when

TVALID is **1.**

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

TREADY is **1.**

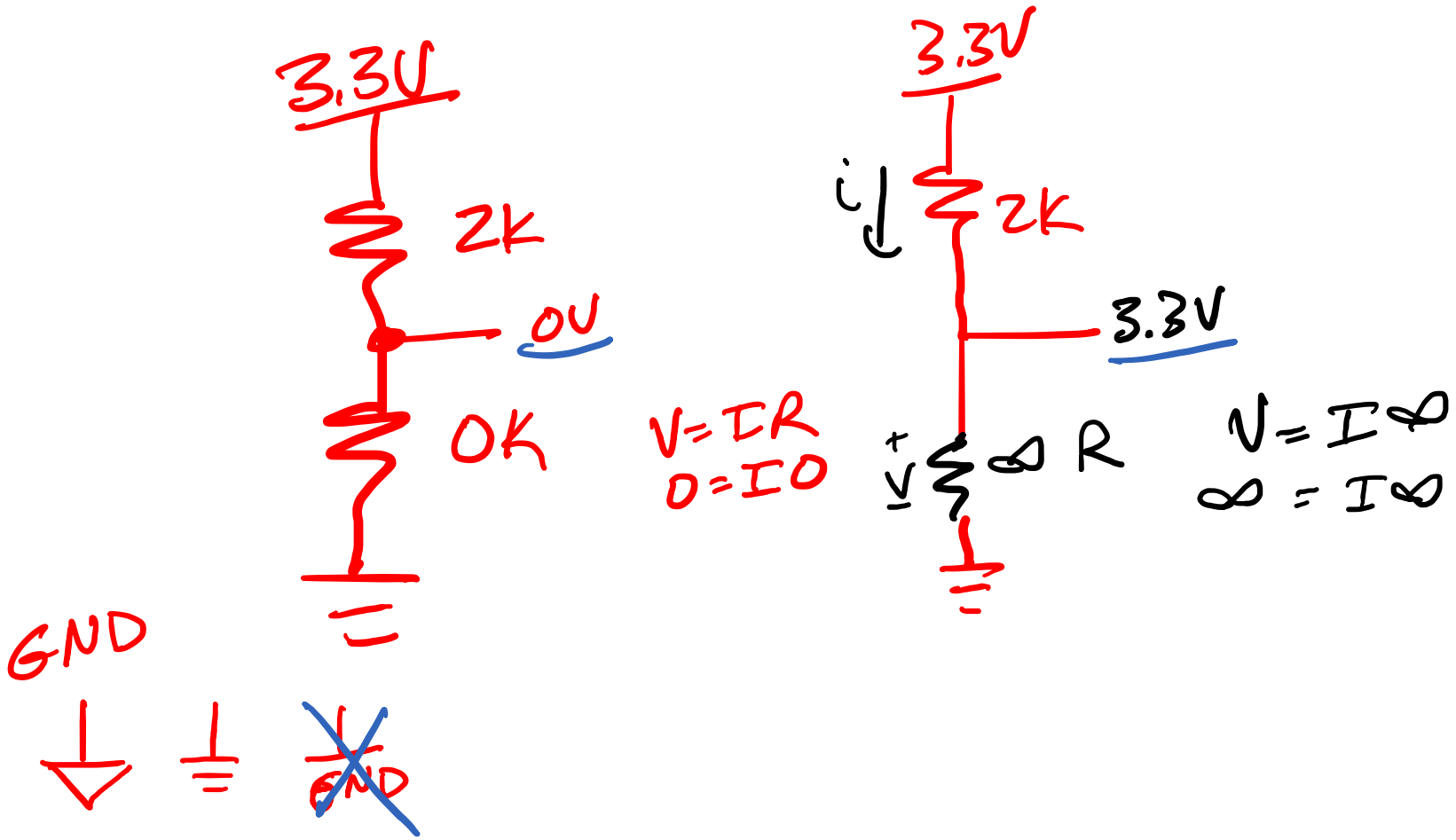
This indicates the **SLAVE** is ready to receive the data.

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

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

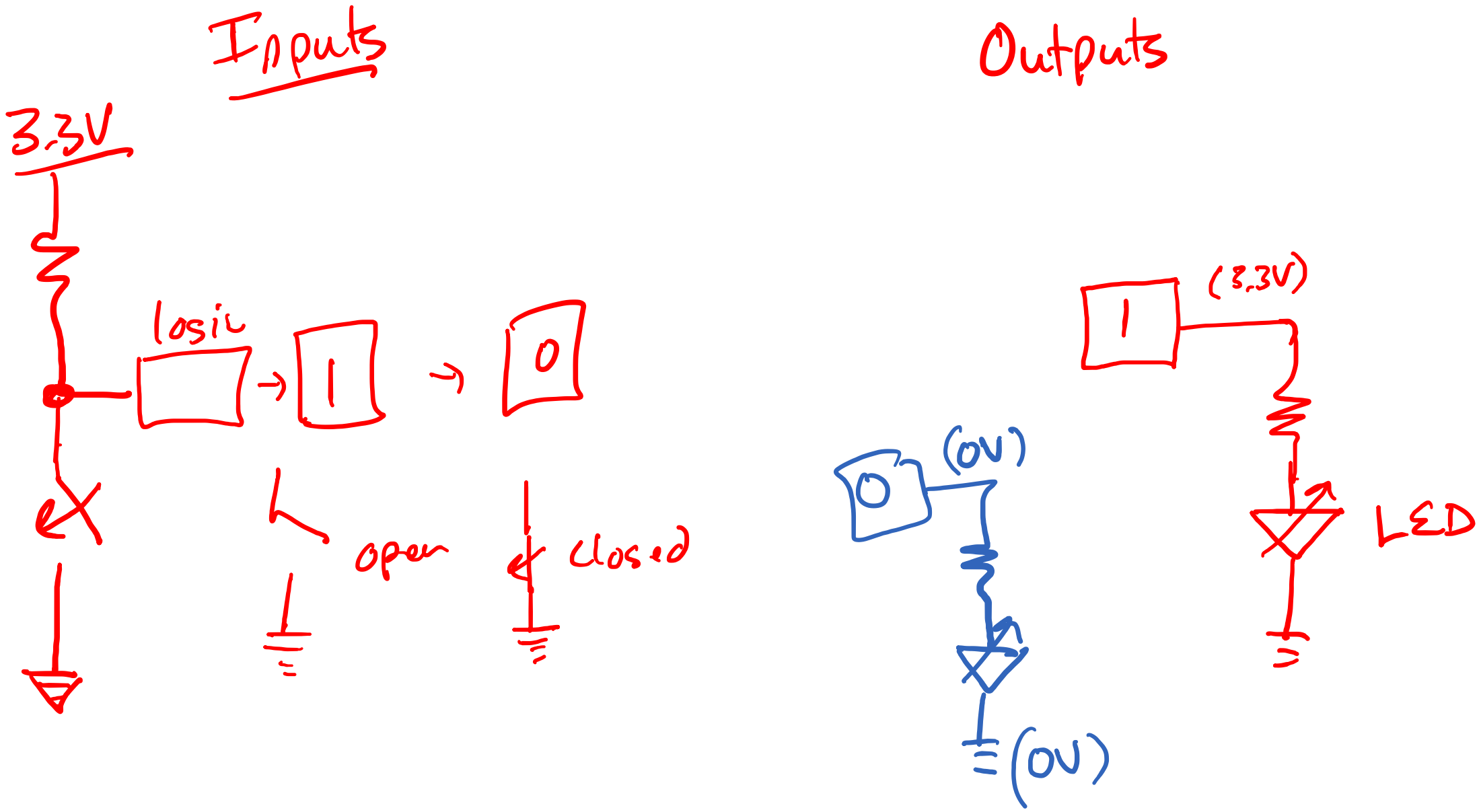
at the positive edge of ACLK

Inputs and Outputs

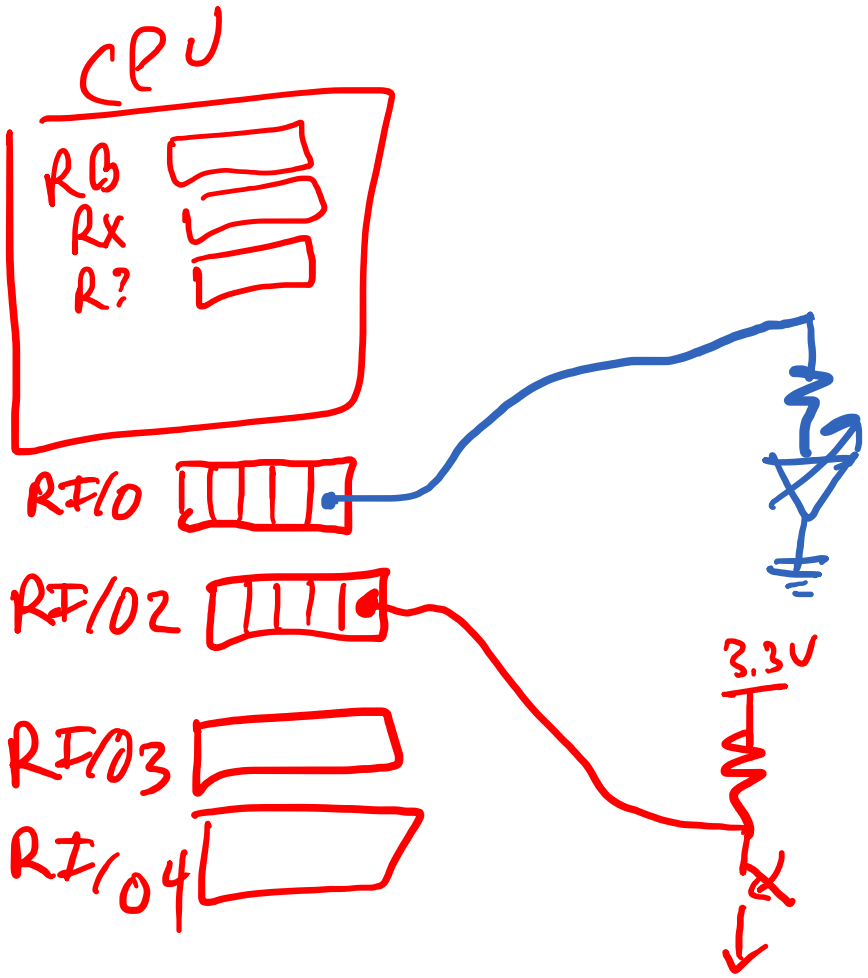


3.3V = max = logic 1

0V = min = logic 0


Inputs
3.3V
logic
1
0
open
closed
Outputs
1
(3.3V)
LED
0
(0V)
(0V)

The Input/Output Problem: Bad Solution



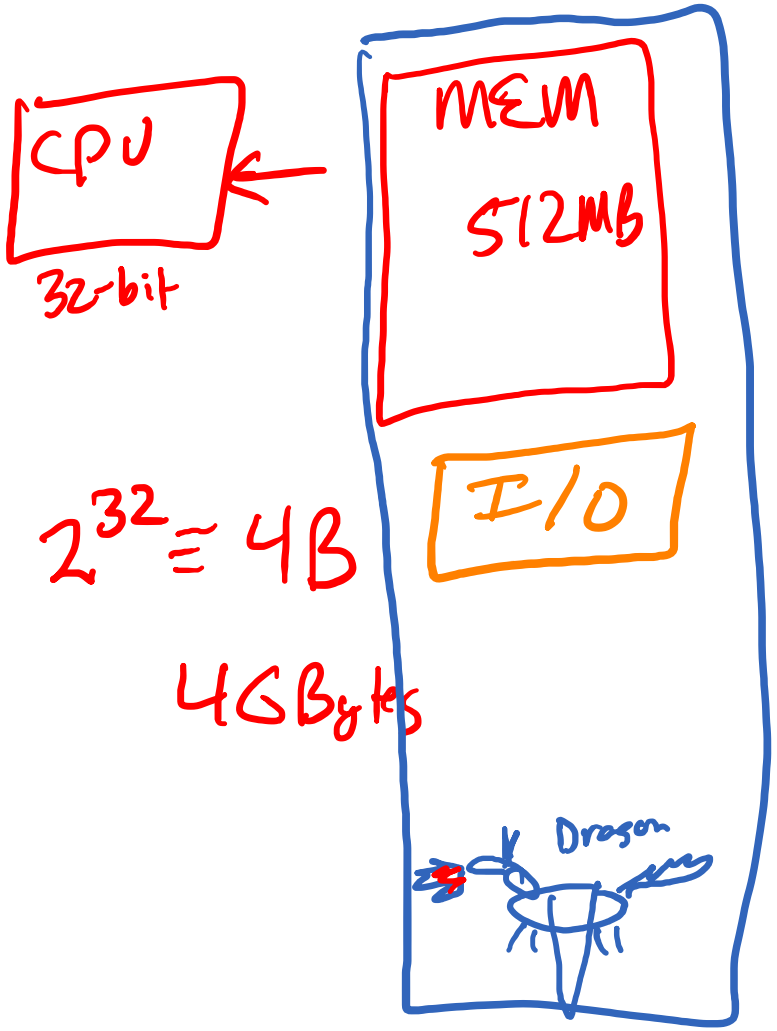
Put I/Os into CPU registers

Memory-Mapped Inputs / Outputs

- Connect I/O to memory address

- "Pretend Memory" I/O accessed with native CPU load/store instructions

Memory Mapped I/O

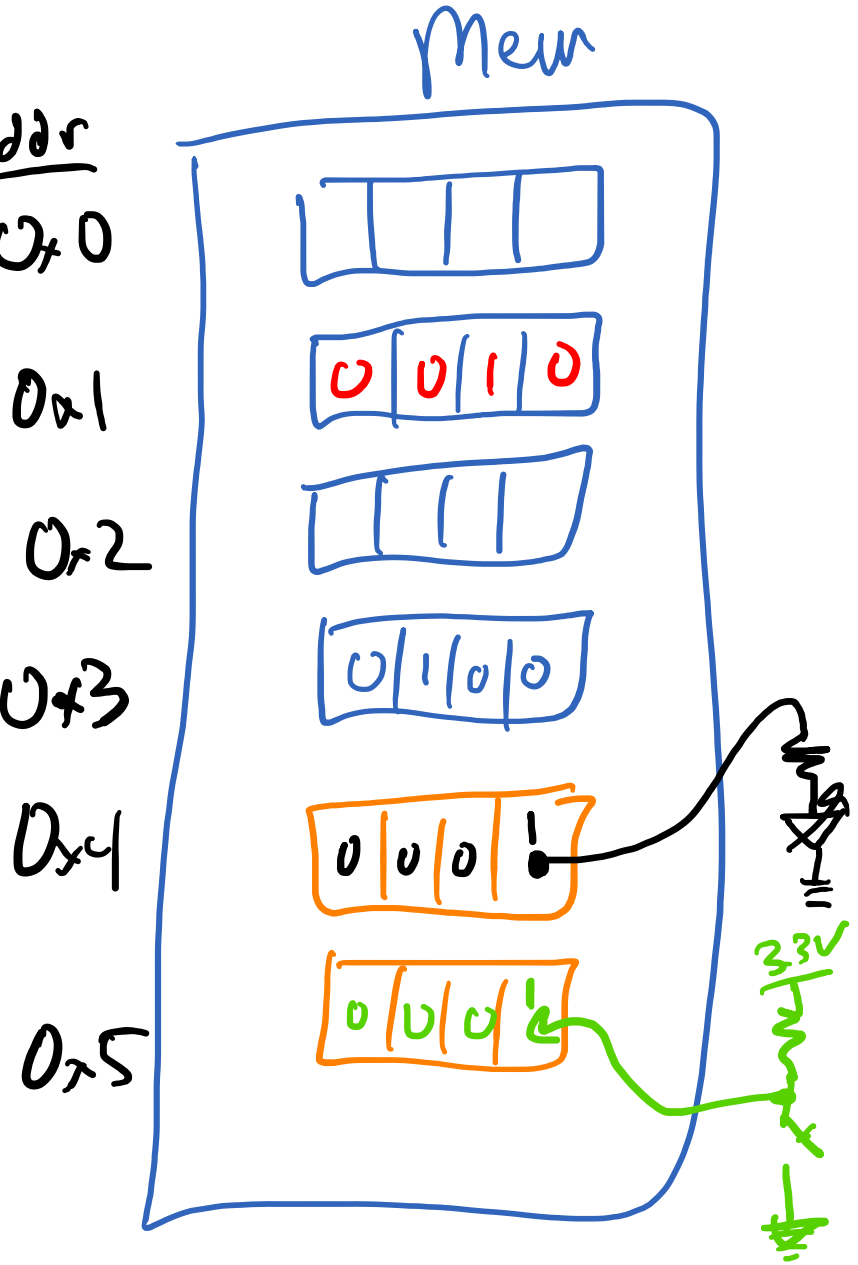


Store
0010 to 0x1

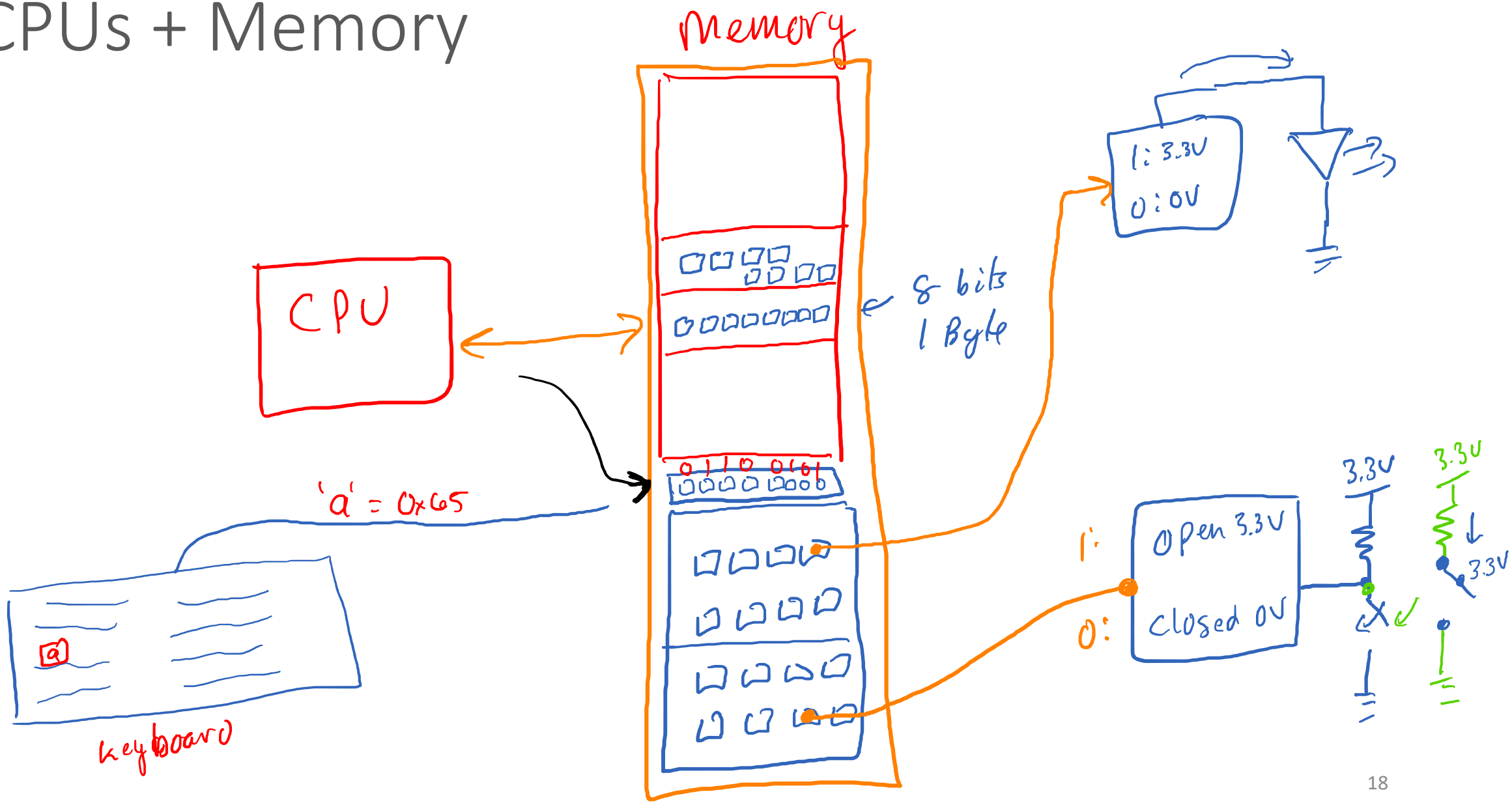
load
from 0x3
⇒ 0100

Store
0001 to 0x4

load
from 0x5
⇒ 0001



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

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

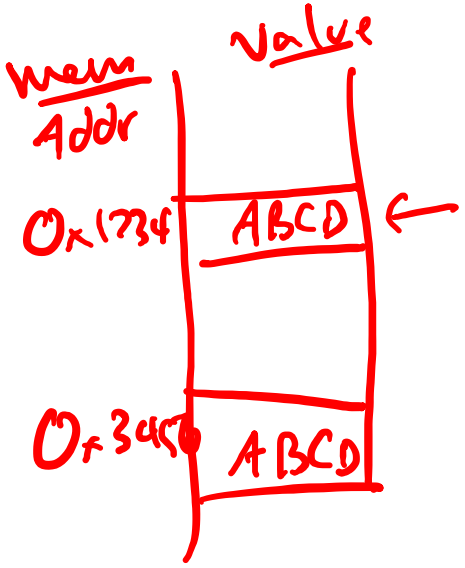
Instruction	Description	Instruction	Description
MOV	Move data	EOR	Bitwise XOR
MVN	Move and negate	LDR	Load
ADD	Addition	STR	Store
SUB	Subtraction	LDM	Load Multiple
MUL	Multiplication	STM	Store Multiple
LSL	Logical Shift Left	PUSH	Push on Stack
LSR	Logical Shift Right	POP	Pop off Stack
ASR	Arithmetic 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

[] → "address of"

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

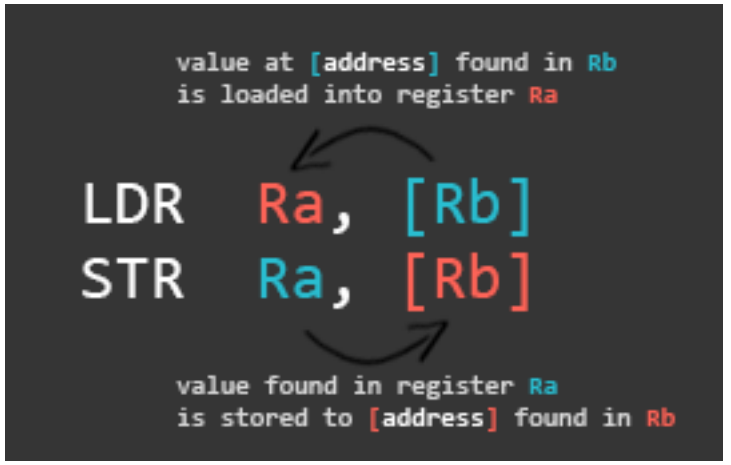
STR R2, [R1] @ [R1] - destination address is the value found in R1.



r0 = 0x1234

r2 = 0xABCD

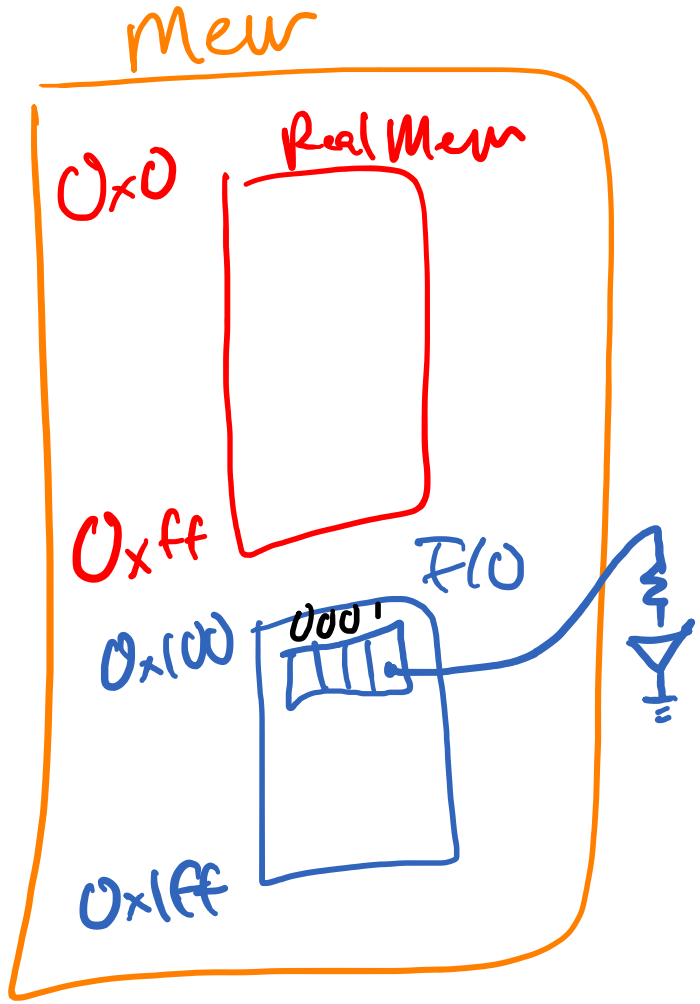
r1 = 0x3456



MMIO Store from ASM (ARM)

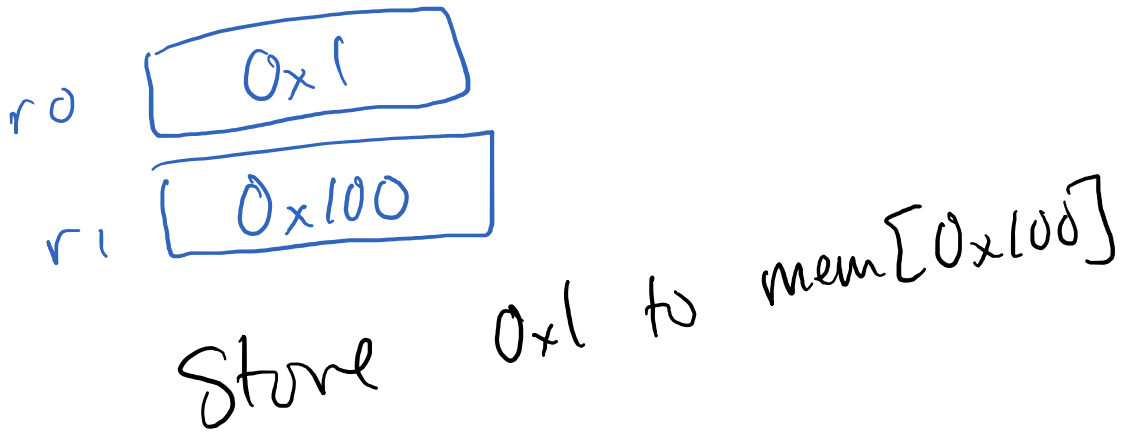
```
mov r0, 0x1
mov r1, 0x100
str r0, [r1]
```

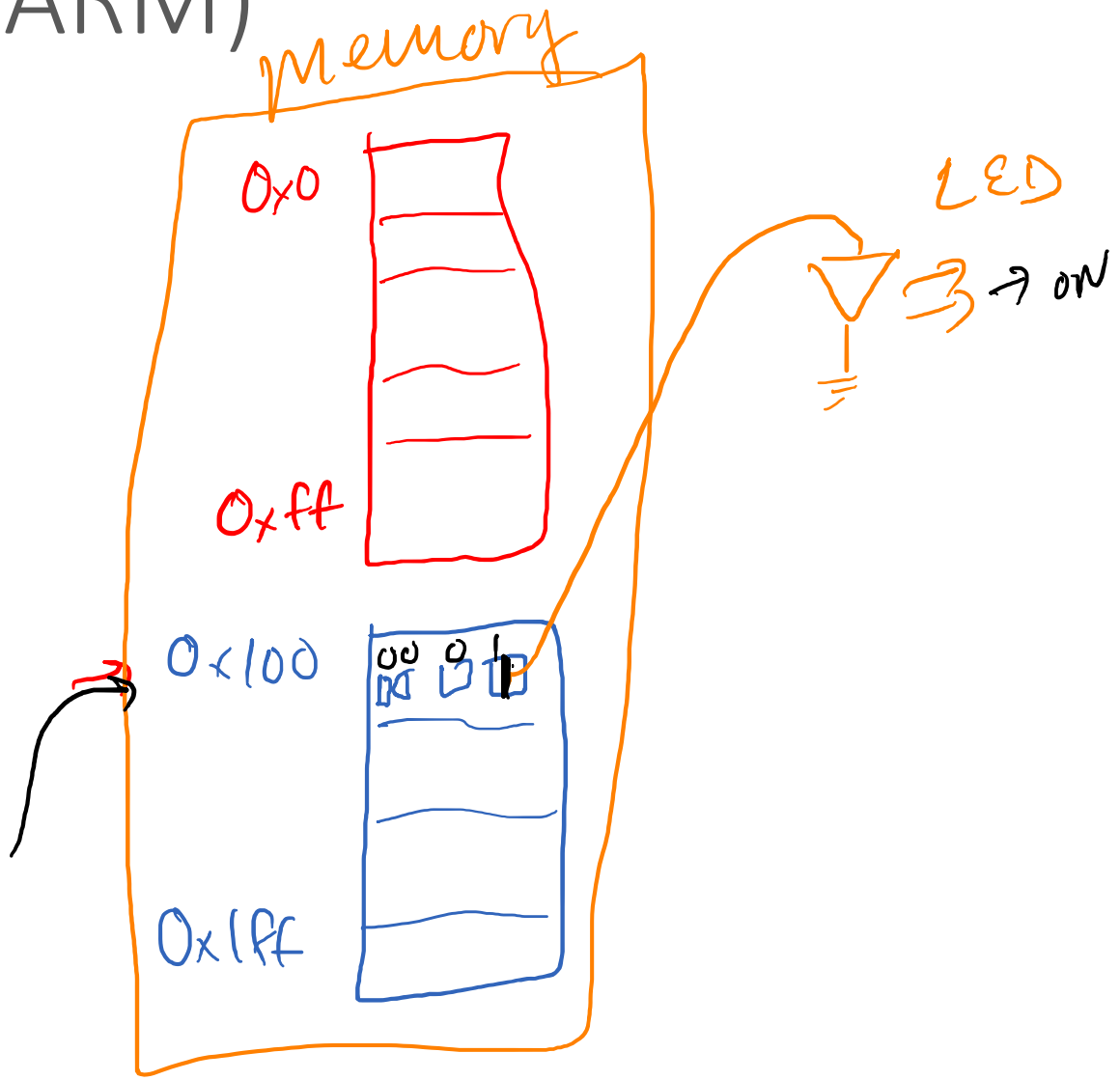
CPU
r0: 0x1
r1: 0x100



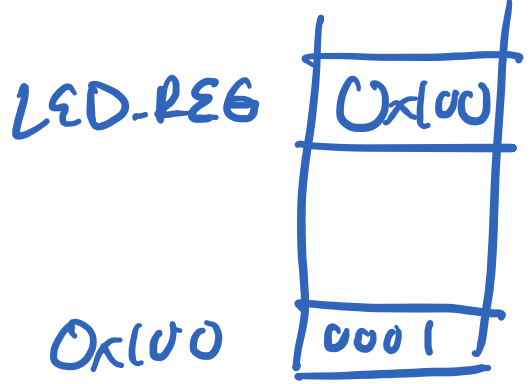
MMIO Store from ASM (ARM)

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





MMIO Store from C




```
#define LED_ADDR 0x100
uint32_t * LED_REG = (uint32_t *)(LED_ADDR);
*LED_REG = 0x1;
```

#define LED 0x100 ← number

uint32_t * LED_REG = (uint32_t *)(LED) ← pointer

* LED_REG = 0x1;

(OR) *((uint32_t *)(LED)) = 0x1;

MMIO Load from ASM (ARM)

```
mov r0, 0x1ff
ldr r1, [r0]
```

MMIO Load from C

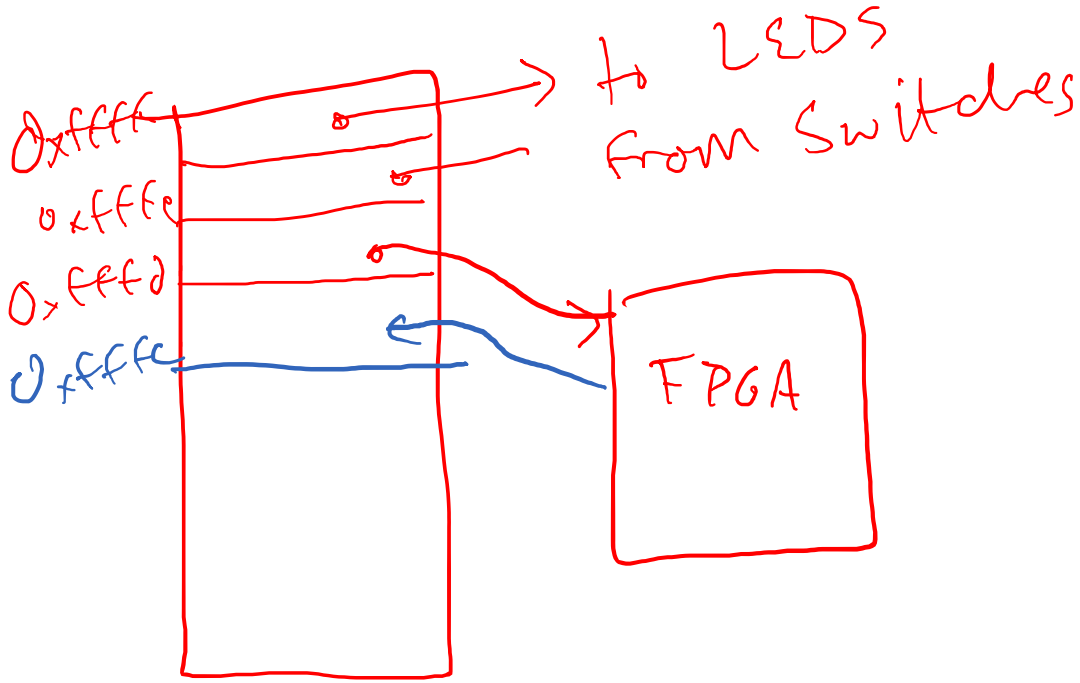
```
#define SW_ADDR 0x1ff
uint32_t * SW_REG = (uint32_t *)(SW_ADDR);
int y = (*SW_REG);
```

MMIO for the FPGA

Memory-Mapped I/O

Stop

- I/O devices pretend to be memory
- Devices accessed with native CPU load/store instructions



Question: What does this code do?

```
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)
{
    //more code
    quit = y;
}
```

Problem: The compiler is “helping”. (-O3 edition)

```
int y = 0;

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

```
~~int y = 0;~~

~~int quit = y;~~
while(1)
{
     //more code
     ~~quit = y;~~
}
```

What's the difference here?

```
int y = 0;



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

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

int quit = (*SW_REG);
while(!quit)
{
     //your 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.

(hint: P4)

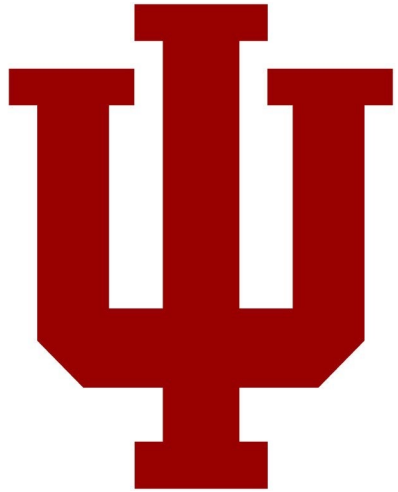
Next Time

- Combine MMIO + AXI Bus

06: Memory-Mapped I/O

Engr 315: Hardware / Software Codesign

Andrew Lukefahr
Indiana University



What happens here?

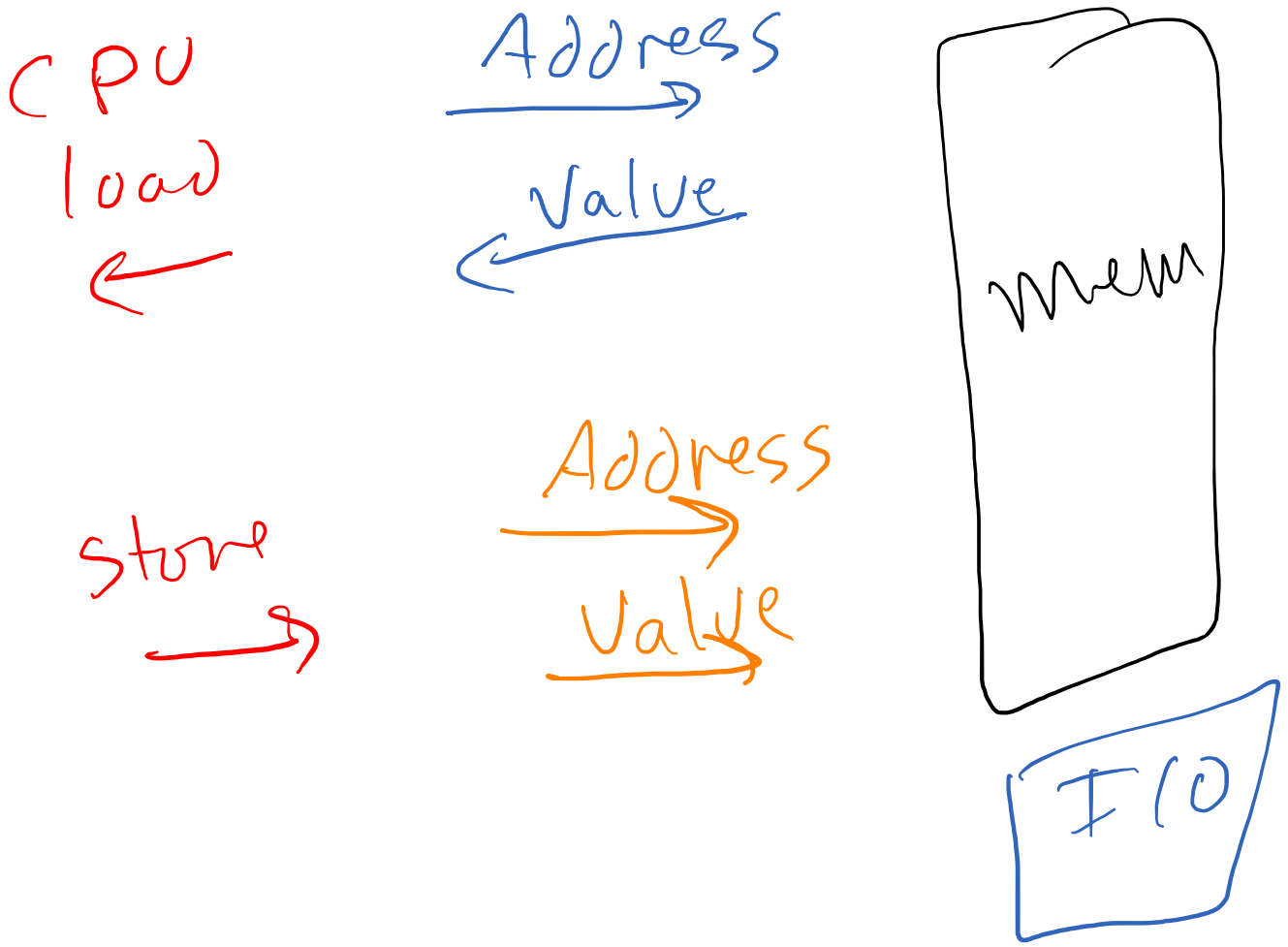
```
#include <stdio.h>
#include <inttypes.h>

#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