Lecture Slides → 210 → fix

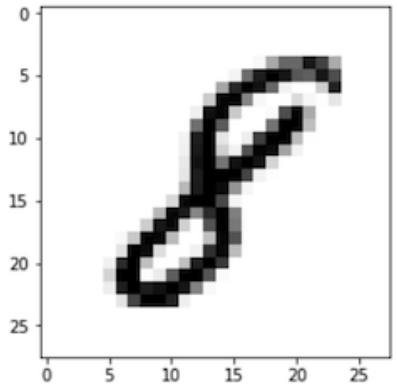
Zynq Link

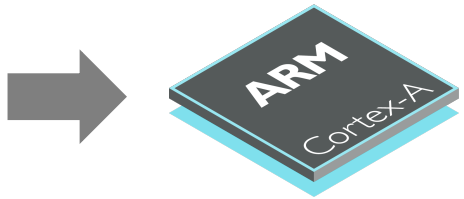
Introduction

Engr 315: Hardware / Software Codesign

Andrew Lukefahr
Indiana University

The goal

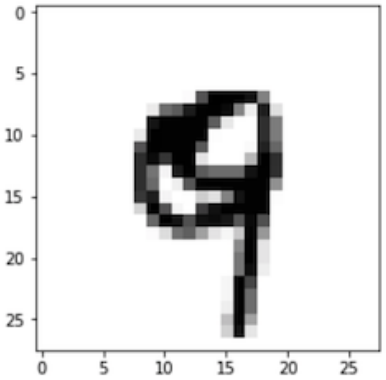


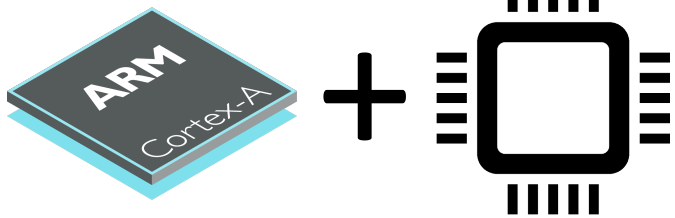


8

10K

3.4 seconds





9

~~you~~ 4 / second

Course Website

engr315.github.io

Write that down!

This class is ***NOT*** about computing.

It's about computing ***FAST***

How can we make our computation FAST?

multi-threads

optimize code

over clock

parallelize (GPU)

custom chip

How can we make our computation FAST?

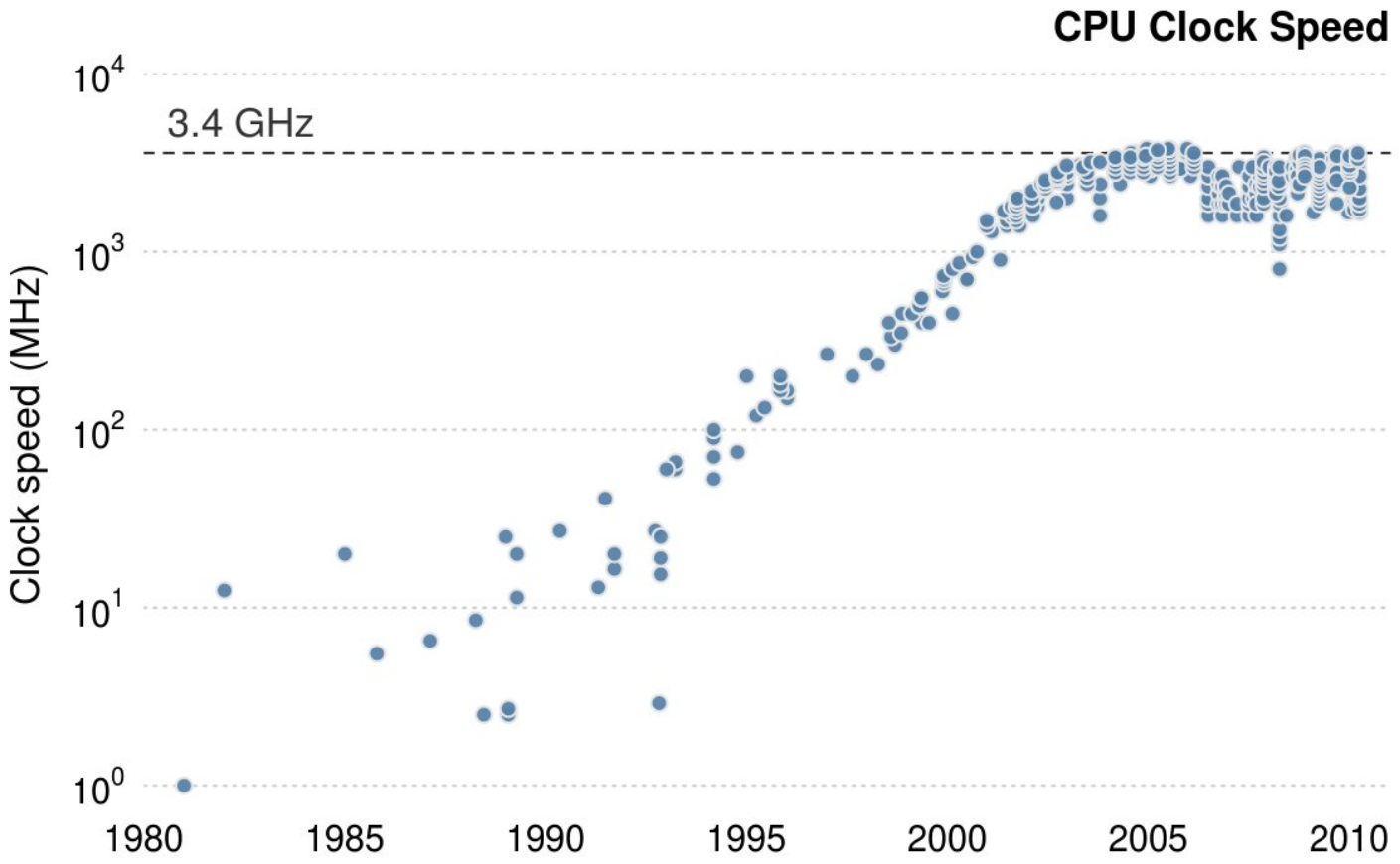
- Do less work.
- Do work faster.
- Do work in parallel.

Doing less work?

- Algorithmic complexity
- Languages:
  - Python vs. C++ vs. C/ASM
- Optimizing compiler
  - `gcc -O3`

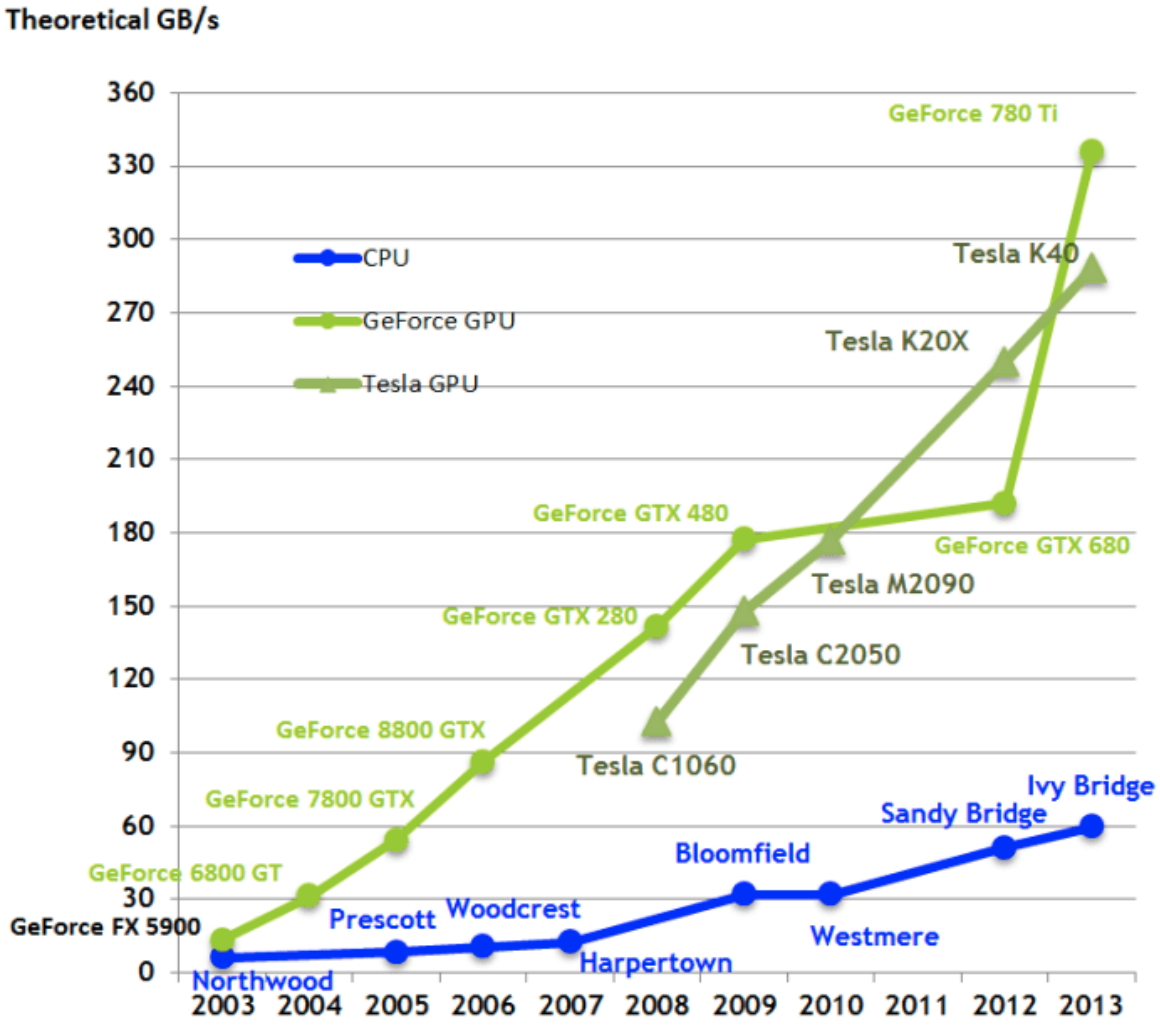
Yep. What else?

Do work faster?



Tried it. Next?

Do work in parallel?



If it works,
it really works!

How to do work in parallel?

1. "Find stuff to parallalize"
2. Parallize it
3. Profit!

The primary goal of this class is:

Use FPGAs to accelerate applications

Parallilze them!

The secondary goals of this class are:

- Find performance bottlenecks in applications
- Accelerate applications using parallelized hardware
- Learn computer systems architectures!

About Me

Andrew Lukefahr, Assistant Professor

Office: 2032 Luddy Hall

Email: *lukefahr@Indiana.edu*

Office Hours: M/W 3-4pm

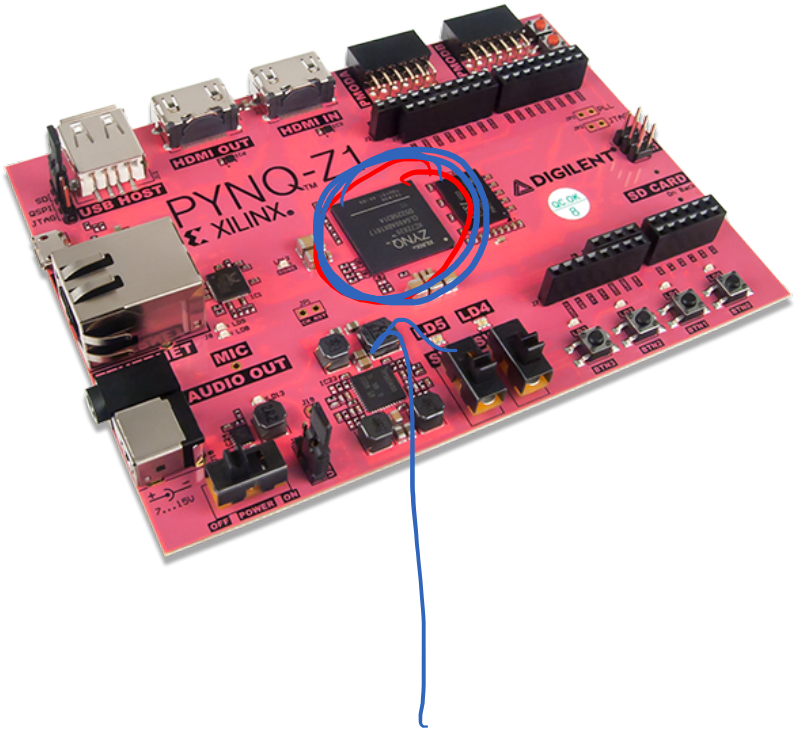
Research work on security for FPGA-based systems.

About Email

- I do not respond to technical questions via email. Use Piazza.
- I treat email as “e”-mail, not instant massaging
- I bulk respond ~1 time / day. Sometimes ~1 time / 2 days.

We'll be using the Pynq-Z1

- System-on-Chip
  - SoC - "S-O-C" or "Sock"
- Contains both FPGA and CPU
- Linux-capable



Course Website

engr315.github.io

Write that down!

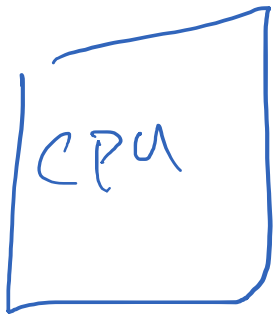
Pynq MNIST Example

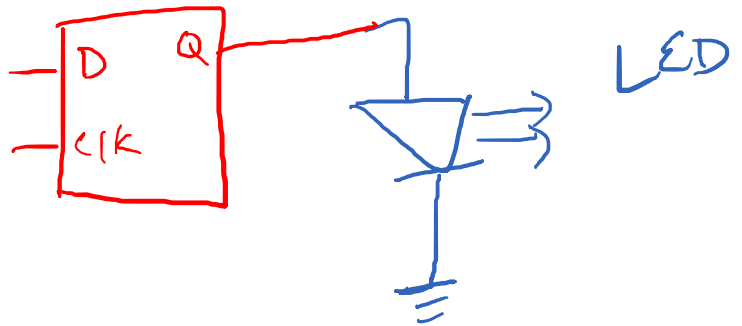
Memory-Mapped Input/Output

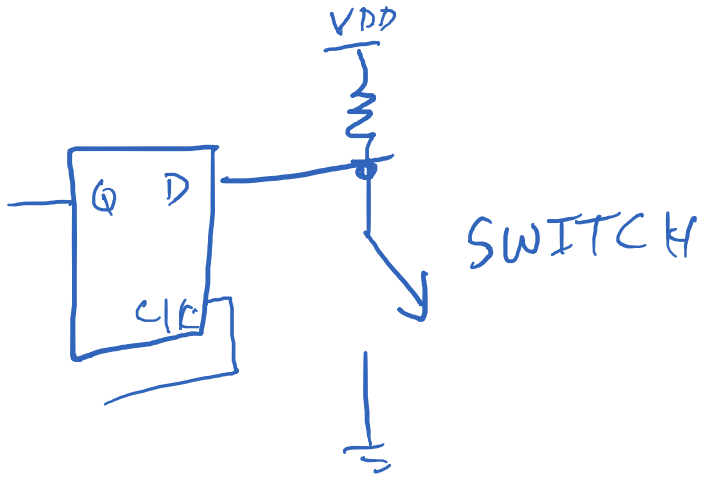
(MMIO)

The I/O Problem

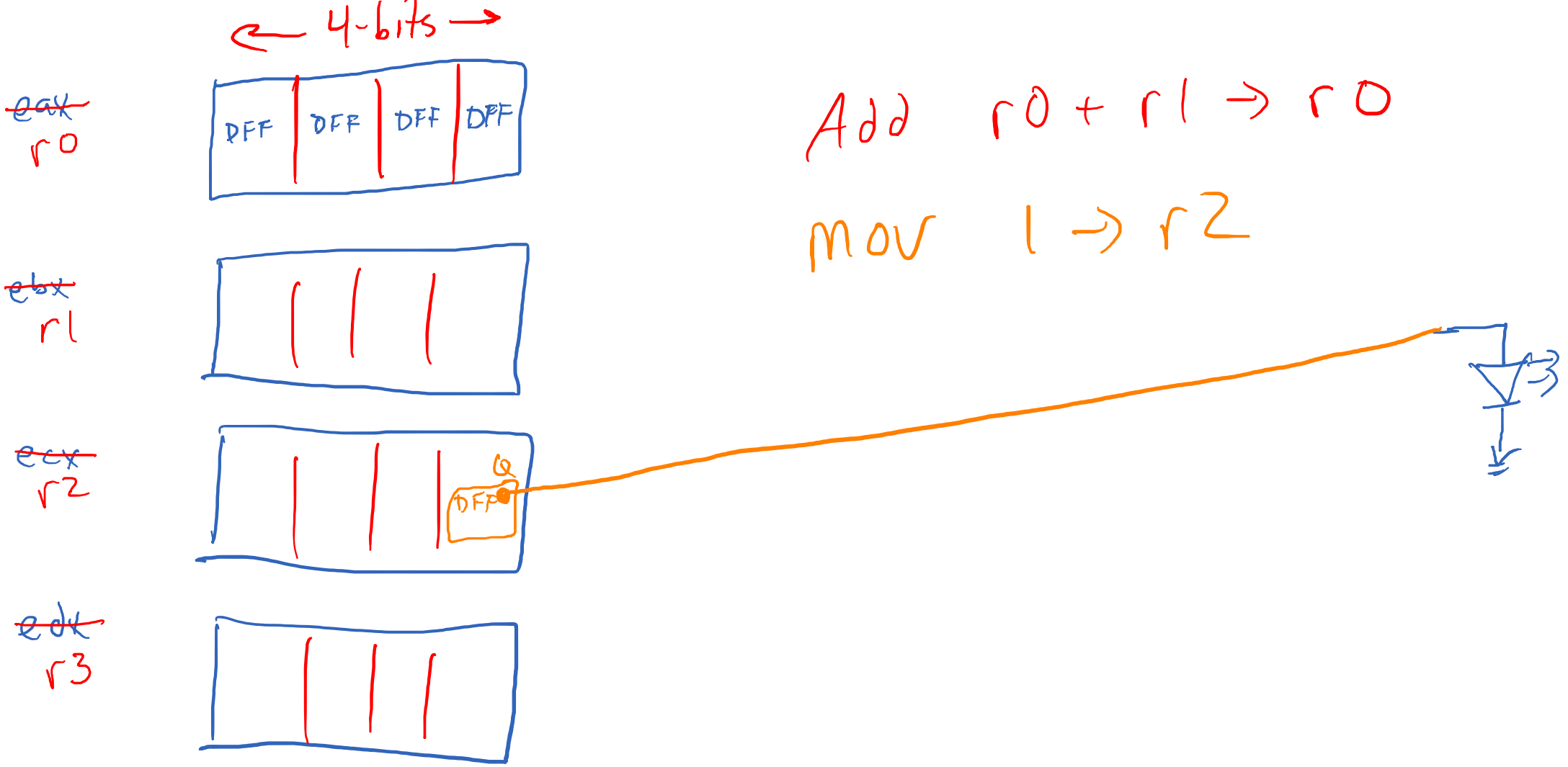
Input / output



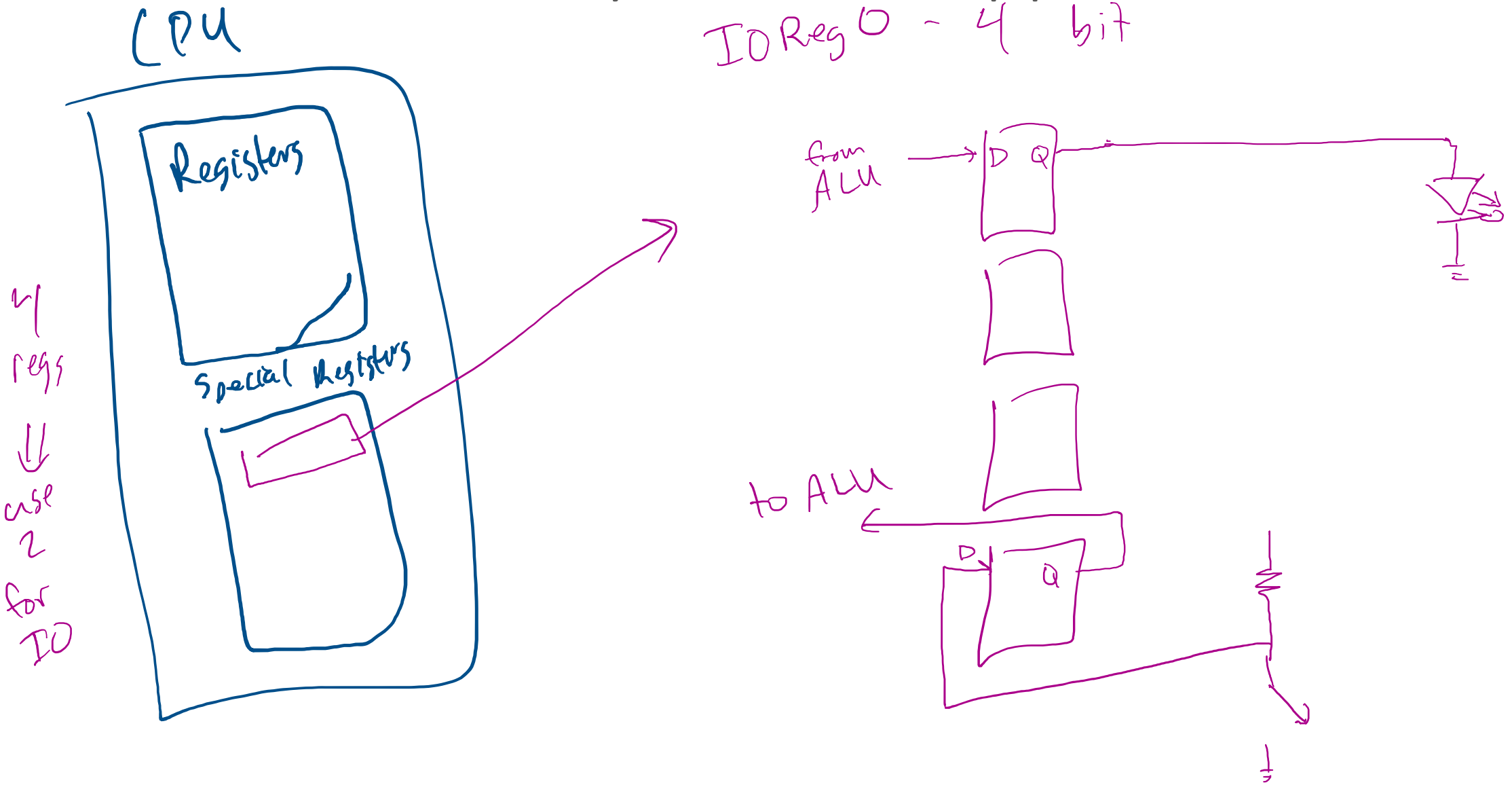




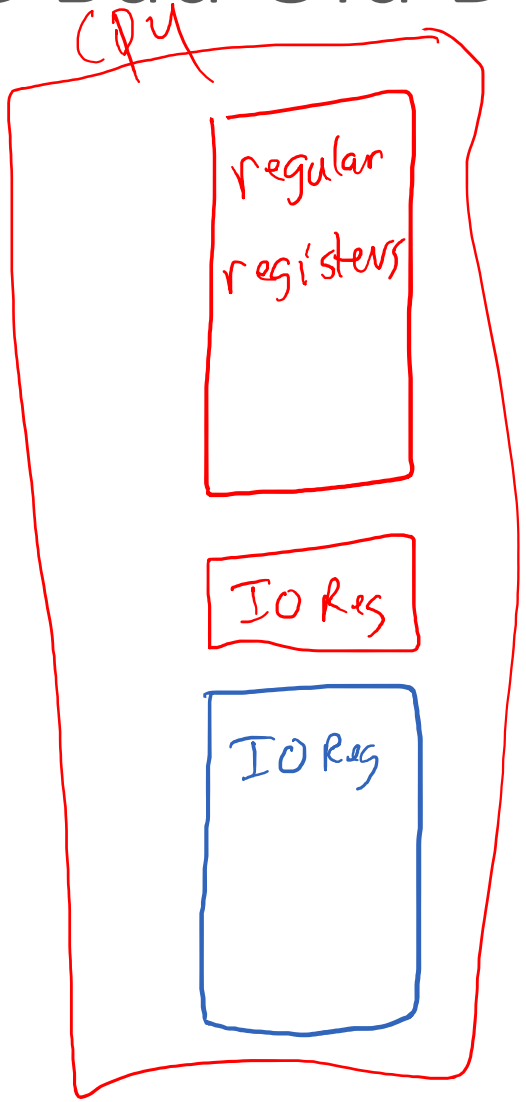
The Bad Old Days: Port-Mapped I/O



The Bad Old Days: Port-Mapped I/O



The Bad Old Days: Port-Mapped I/O



Simple, but doesn't scale to bigger IO

mov IOReg0 ← RAX

Modern systems have 1000s of I/O registers...

This works, but doesn't scale

Memory-Mapped I/O

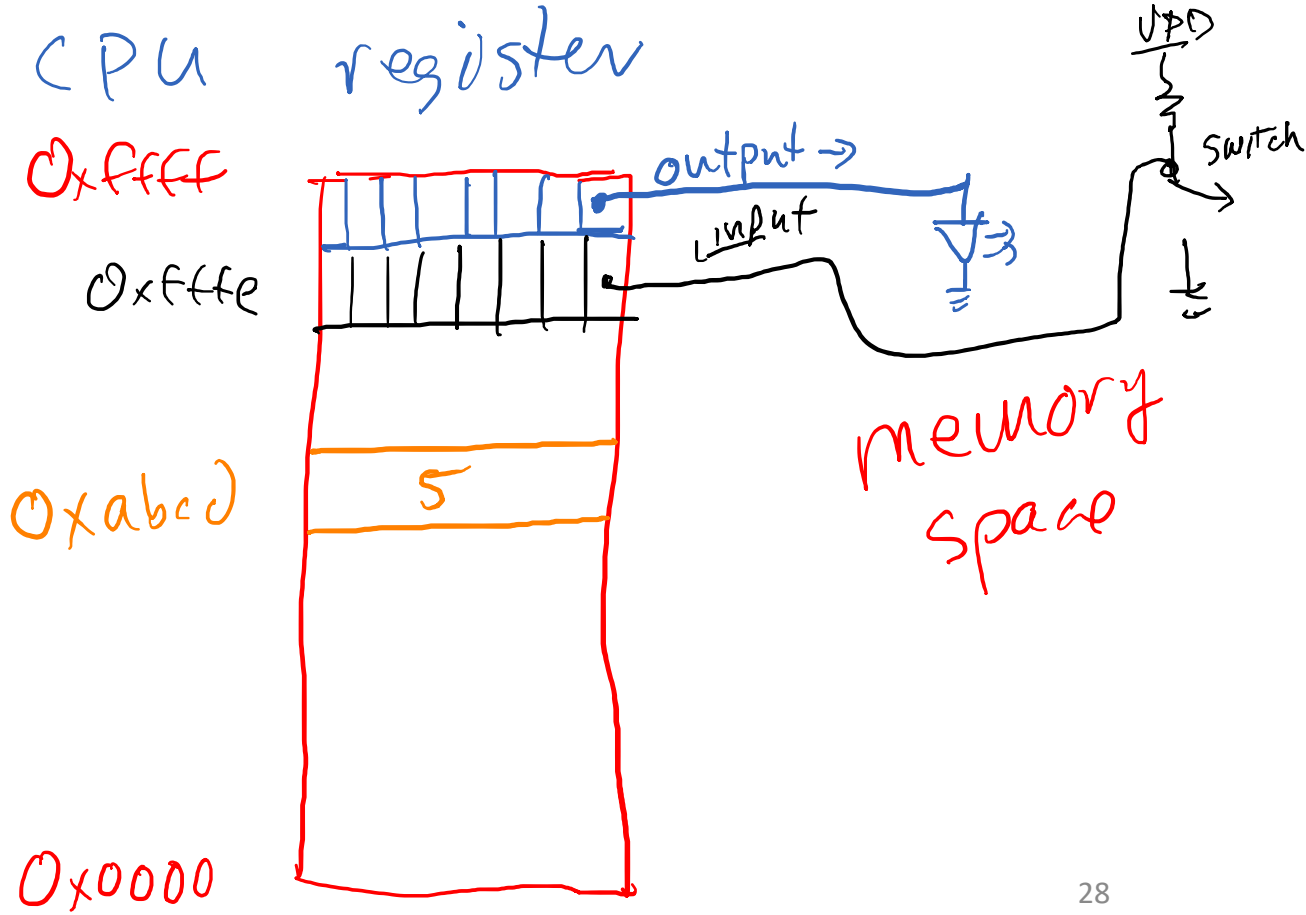
connect I/O to memory address
not CPU register

load 0xabcd → 5

store 5 ⇒ 0xabcd

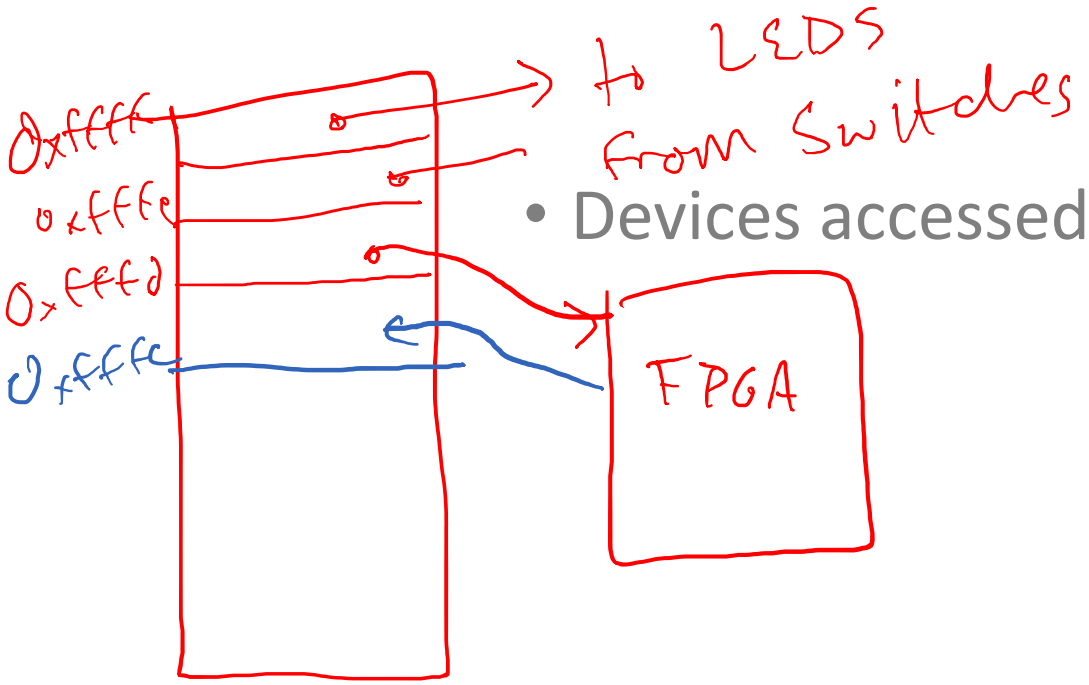
store 1 ⇒ 0xffff

load 0xfffe → r0



Memory-Mapped I/O

- I/O devices pretend to be memory



- Devices accessed with native CPU load/store instructions

ARM Assembly Crash Corse

- Registers
- Instructions

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

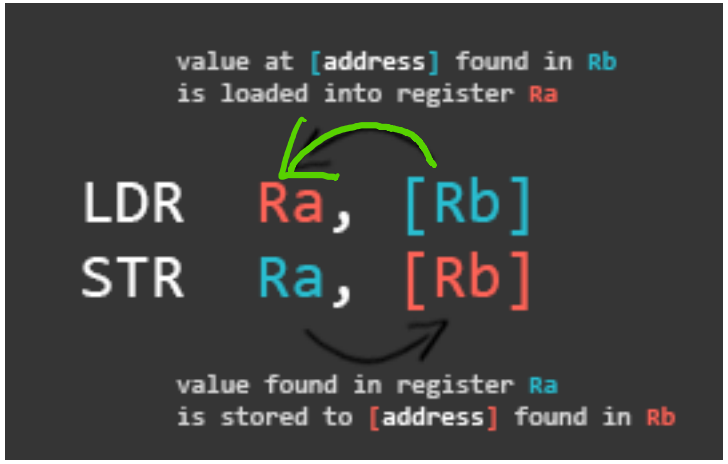
ARM Instructions

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 C

```
#define LED_ADDR 0xffff

uint32_t * LED_REG = (uint32_t *)(LED_ADDR);
(unsigned int)

*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 ASM

```
mov r1, 0xfffe
ldr r2, [r1]
```

MMIO Load from C

Does `quit` ever change here? Do I need to recompute `(!quit)`?

```
int y = 0;

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

```
while(1){
    // your code
}
```

What about here?

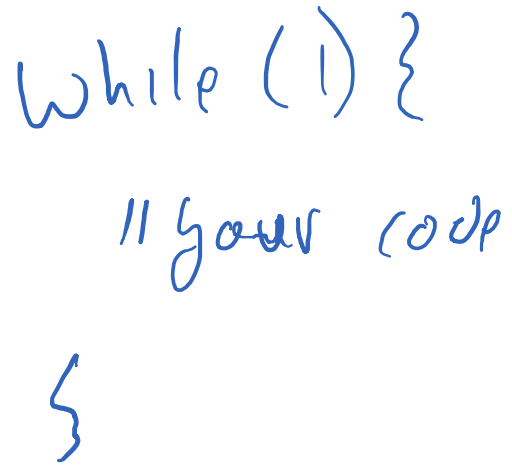
```
int y = 0;
uint32_t * SW_REG = &y;

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

constant

constant

constant



What about here?

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

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!

Next Time

- Blinking LEDs Example Project

Introduction

Engr 315: Hardware / Software Codesign

Andrew Lukefahr
Indiana University