



Computer Hardware Engineering (IS1200)

Computer Organization and Components (IS1500)

Spring 2021

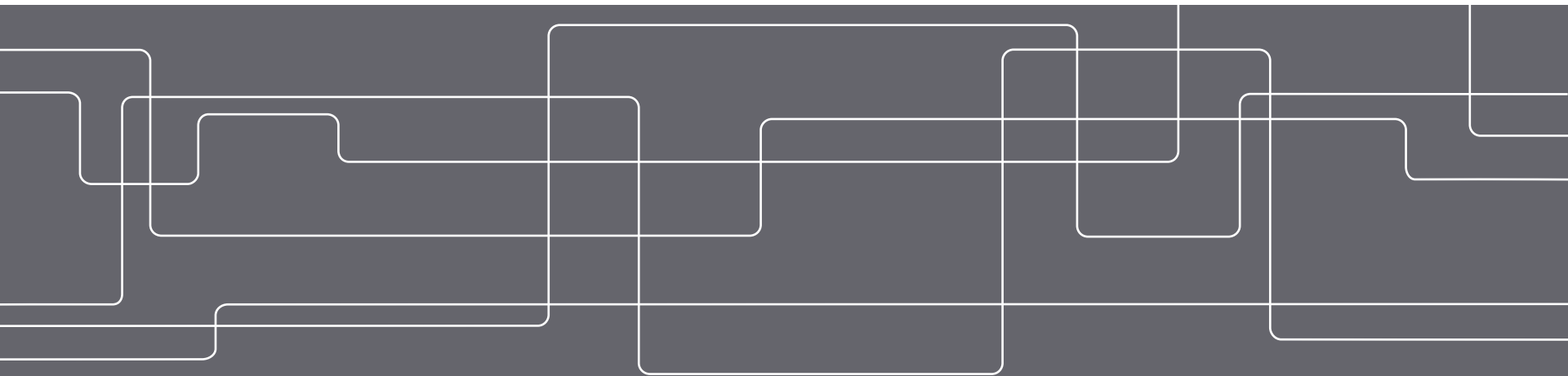
Lecture 5: I/O Systems, part I

Note: We start 13:15

Artur Podobas

Researcher, KTH Royal Institute of Technology

Slides by David Broman, KTH (Extensions by Artur Podobas)





Announcements

Announcement 1:

We have been working hard trying to solve the problem of distributing ChipKITs now during the pandemic. It is now almost solved, and we'll hopefully be able to post another announcement regarding this tomorrow. However, it is possible that not all of you will be able to receive a ChipKIT before Lab 1. Because of this, **we'll allow presenting assignment 7 of Lab 1 (the only part of Lab 1 requiring the ChipKIT) at a later lab session.** It is worth noting that assignment 7 is very minor, so this will **not** affect you negatively in any way.

Also, we discovered that some of you have been able to collect ChipKITs from the service center already. This was not our intention, so for the rest of you: **please do not go to the service center for your ChipKITs until you receive further instructions from us.**

Announcement 2:

Still missing student representatives. *It is imperative* that we form good communication channels that provide early feedback on what works and what does not early on, so that we can remedy them as the course evolves.

Please send me (podobas@kth.se) or Daniel (dlunde@kth.se) a mail if you are interested!

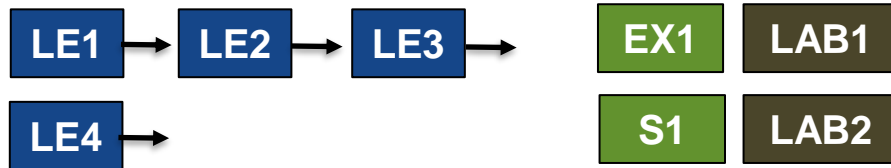
Part I
Basic I/O

Part II
Buses and
DMA

Course Structure



Module 1: C and Assembly Programming



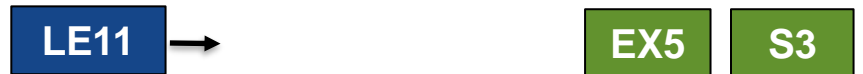
Module 4: Processor Design



Module 2: I/O Systems



Module 5: Memory Hierarchy



Module 3: Logic Design (IS1500 only)

**PROJ
START**



Module 6: Parallel Processors and Programs



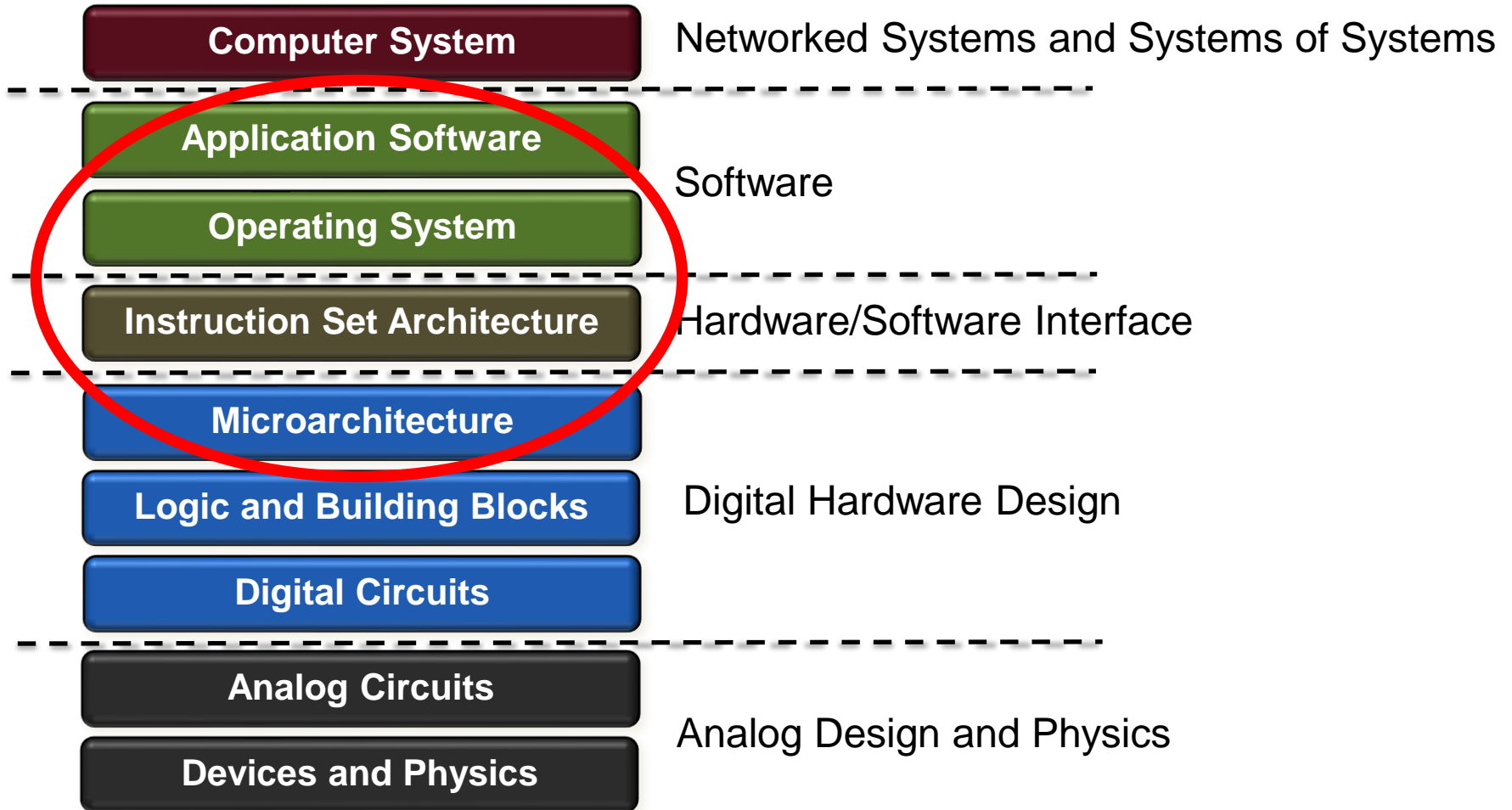
Proj. Expo

LE14

Part I
Basic I/O

Part II
Buses and
DMA

Abstractions in Computer Systems



Agenda

Part I Basic I/O



Part II Buses and DMA



Part I
Basic I/O

Part II
Buses and
DMA

Part I

Basic I/O



Acknowledgement: The structure and several of the good examples are derived from the book “Digital Design and Computer Architecture” (2013) by D. M. Harris and S. L. Harris.

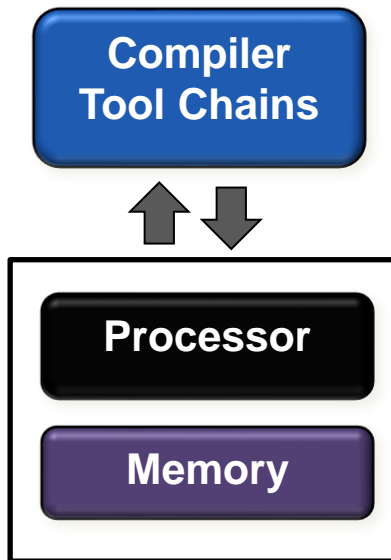


Part I
Basic I/O

Part II
Buses and
DMA

I/O Devices - Examples

I/O Devices (also called **peripherals**) for a PC or mobile device.



I/O = Input / Output

Device	Behavior	Partner	Data Rate
Keyboard	Input	Human	0.0001 Mbit/sec
Mouse	Input	Human	0.0038 Mbit/s
Laser Printer	Output	Human	3.2 Mbit/s
Graphics Display	Output	Human	~ 256 Gbit/s
Network/LAN	Input/Output	Machine	~10-100 Gbit/s
Flash Memory	Storage	Machine	32.0 – 200.0 Mbit/s

Source: Patterson & Hennessy,
“Computer Organization and Design” 4th edition, 2012

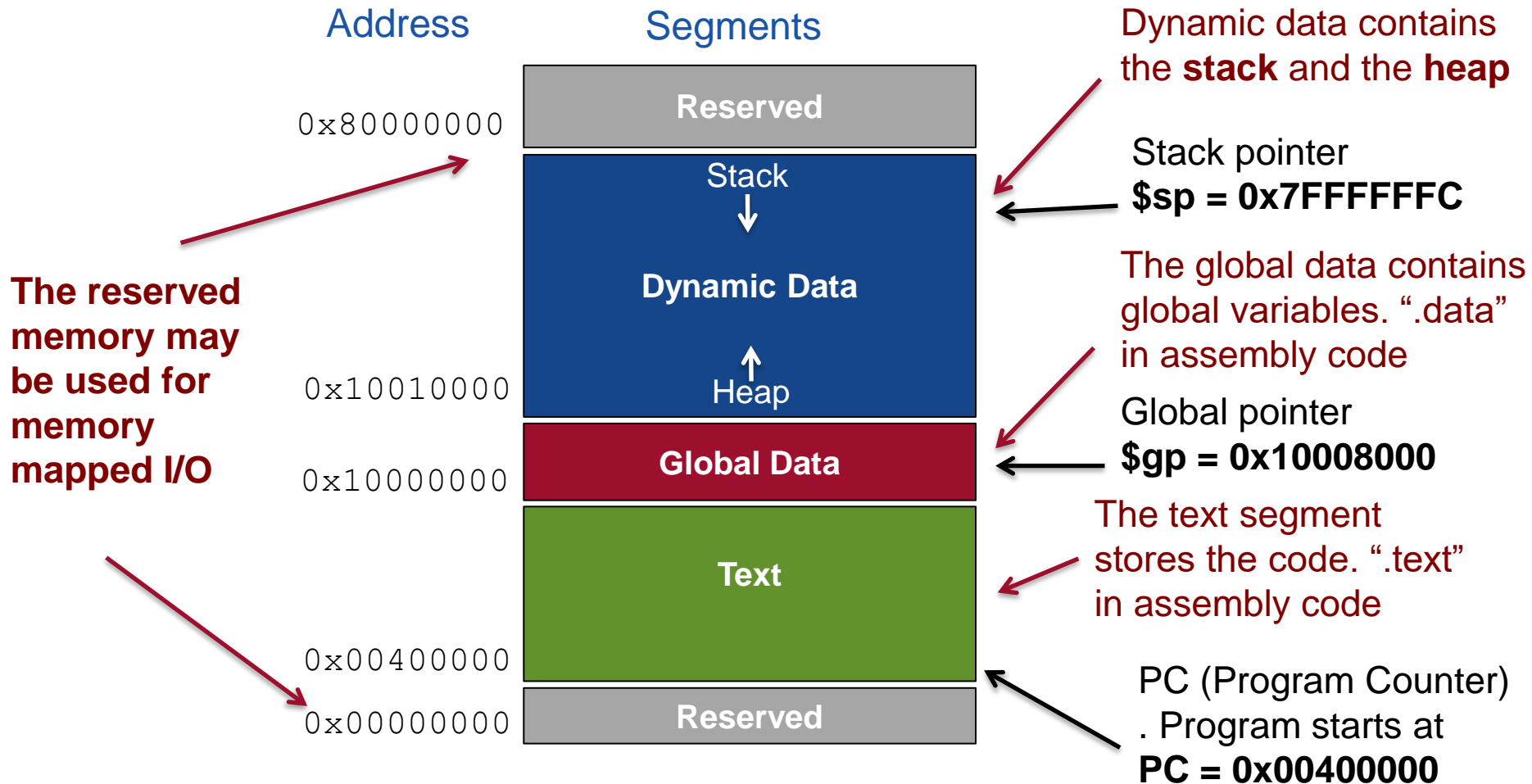
In **embedded systems**, I/O devices can be, for instance, microwave controller, fuel injector in an engine, motors, etc.



Part I
Basic I/O

Part II
Buses and
DMA

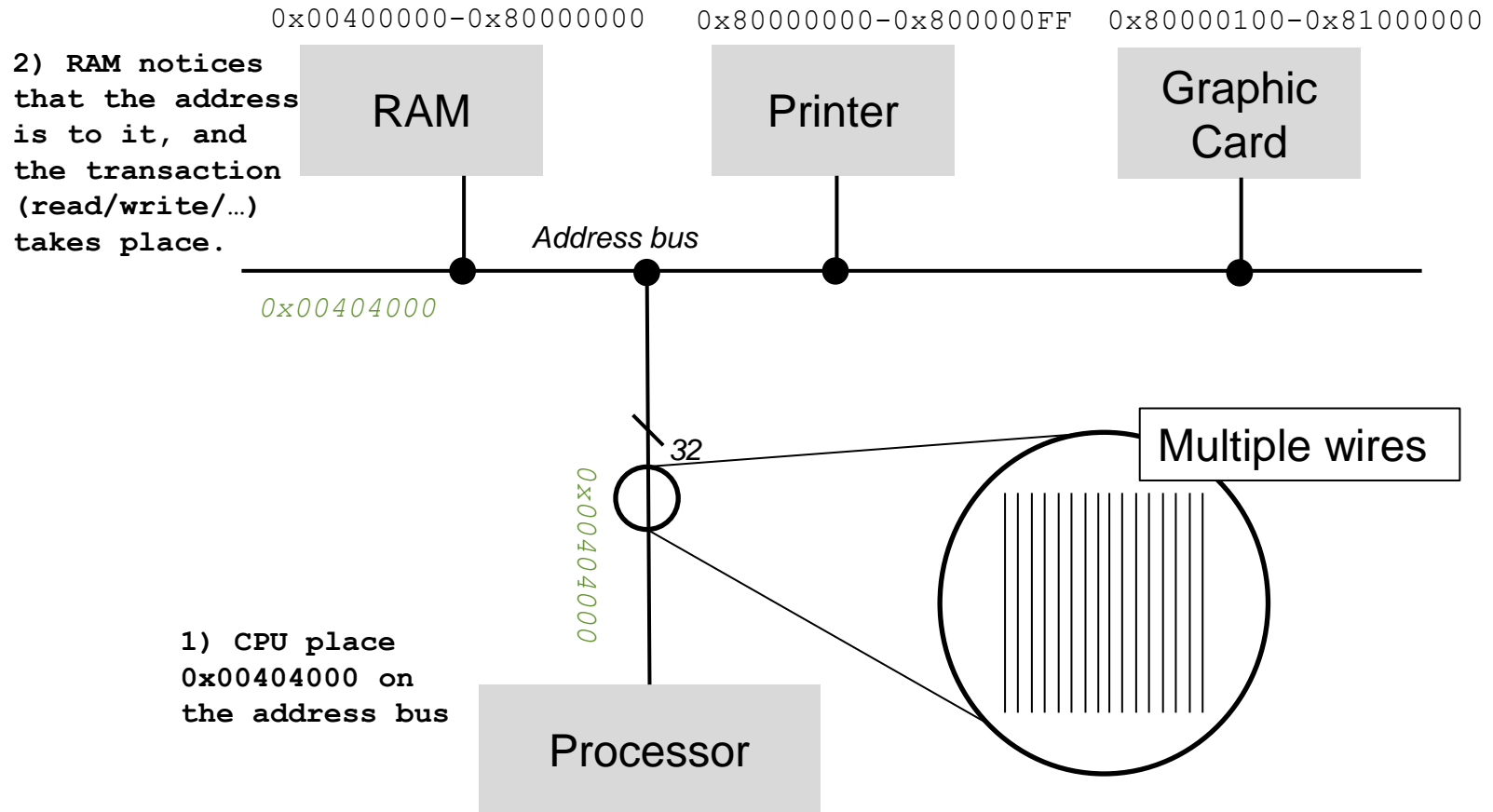
Memory Mapped I/O



Part I
Basic I/O

Part II
Buses and
DMA

Memory Mapped I/O



Memory Mapped I/O

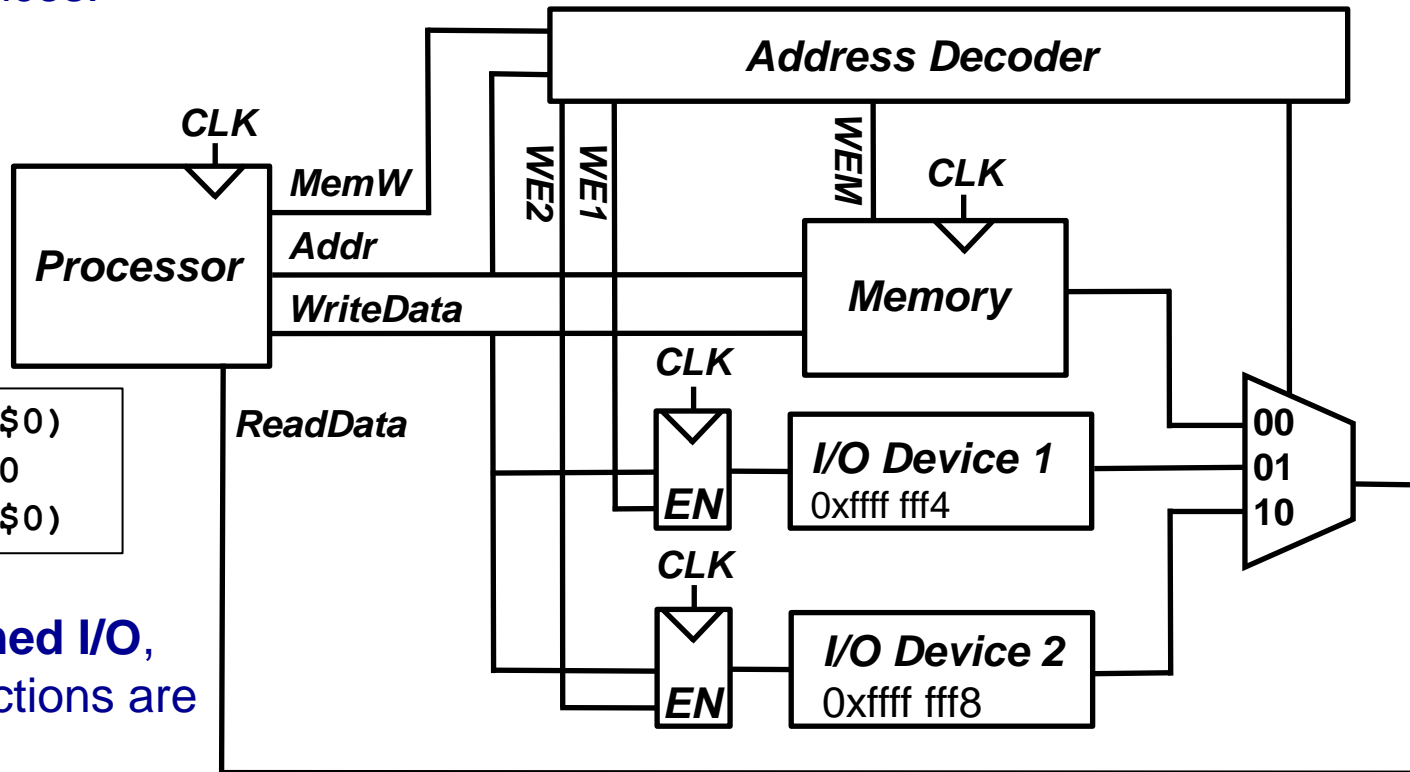
In **memory mapped I/O**, parts of the address space is dedicated to I/O devices.

What is the following MIPS code doing, assuming that device 1 is mapped to 0xffff fff4 and device 2 to 0xffff fff8?

```
lw    $t0, 0xffff8($0)
addi  $t0, $t0, 100
sw    $t0, 0xffff4($0)
```

x86 uses **programmed I/O**, where special instructions are instead used for I/O.

Historically, I/O devices and memory were connected using *buses* (shared parallel wires), but today it is also common with dedicated wires.



Part I
Basic I/O

Part II
Buses and
DMA

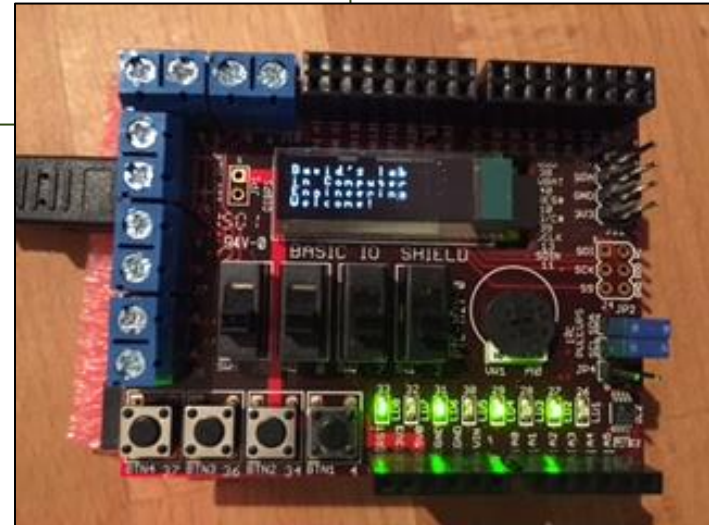
Simple I/O on the ChipKIT Board (1/2)

Exercise:

On the chipKIT board, the 8 green LEDs are memory mapped to the least 8 significant bits of the word starting at address 0xbf886110. Write a MIPS assembly function **led_test** (without pseudo instructions) that can be called with the following C statement, and lights up the LEDs as show in the picture.

```
led_test(0xAA);
```

```
.global led_test
led_test:
    lui    $t0, 0xbf88
    ori    $t0, $t0, 0x6110
    sw     $a0, 0($t0)
    jr     $ra
```



Part I
Basic I/O

Part II
Buses and
DMA

Simple I/O on the ChipKIT Board (2/2)

Exercise:

Discuss what the following MIPS code is doing.

Hint: the toggle switches are memory mapped to the word address 0xbf8860d0 (bits 11 through 8) and the green LEDs to 0xbf886110 (bits 7 through 0).

loop:

```
    lui    $t0, 0xbf88
    lw     $t1, 0x60D0($t0)
    srl    $t1, $t1, 8
    andi   $t1, $t1, 0xf
    mul    $t1, $t1, $t1
    sw     $t1, 0x6110($t0)
    j      loop
```

Put the 16 most significant bits in \$t0

Load the switch status using index (immediate) value

Shift bits to the right (11 through 8 contains the information), mask, and multiply.

Multiply by itself and display the power of two by using the LEDs.



7-Segment Display

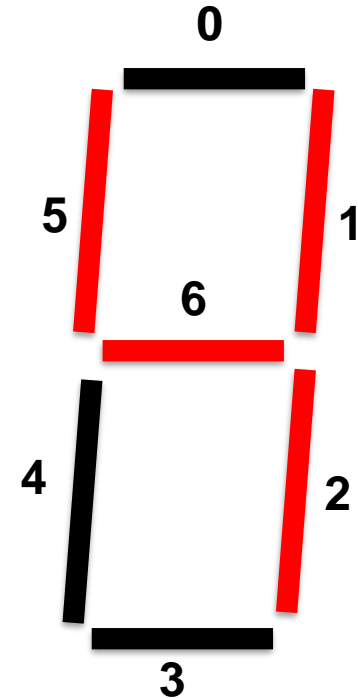
Exercise:

Write a C function that displays digit 4 on a 7-segment display. Assume that the bits should be 1 if the diode should be on (see bit indices in the figure). The memory mapped I/O address is 0x9f0.

```
void display4() {
    volatile int* segm7 = (volatile int*) 0x9f0;
    *segm7 = 102;
}
```

Why did we use the keyword **volatile**?

The value for displaying 4 is
 $0b01100110 = 64 + 32 + 4 + 2 = 102$



The `volatile` C Keyword

The **`volatile`** keyword is used on variables in C programs to avoid that the C compiler performs certain optimizations.

```
int get_toggle(){  
    int * toggle18 = (int *) 0x850;  
    while(*toggle18 == 0);  
    return *toggle18;  
}
```

The C compiler might optimize this code, assuming that the value `*toggle18` does not change.

```
int* toggle18 = (int*) 0x850;  
int tmp = *toggle18;  
while(tmp == 0);
```

Pseudo-code for a possible “optimization” that the compiler might perform.

```
volatile int* toggle18 = (volatile int*) 0x850;
```

To be on the safe side, always use `volatile` when using memory mapped I/O.



General-Purpose Digital I/O

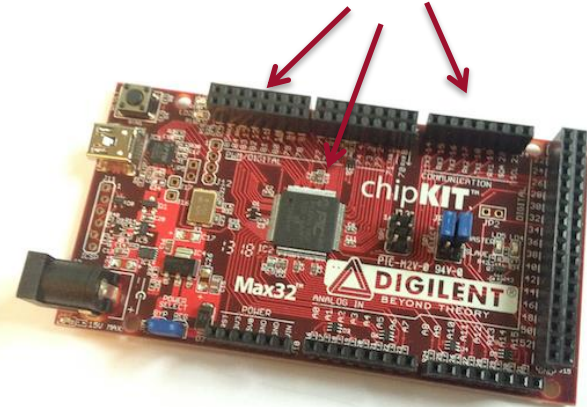
On microcontrollers, there are typically many **General-Purpose I/O (GPIO)** pins that can be configured to be either input or output.

Example. On a PIC32, there are several ports (port A, port B, ...), where each port can have up to 16 GPIOs.

Two important registers for each port ←

- **TRISx** (where x is the port). This registers tells if the port is an output (bit value is 0) or input (bit value is 1)
- **PORTx** (where x is the port). This is the register for writing (if output) or reading (if input).

GPIOs on PIC32 board (32-bit MIPS processor)



Note that we say registers, although the registers are memory mapped.

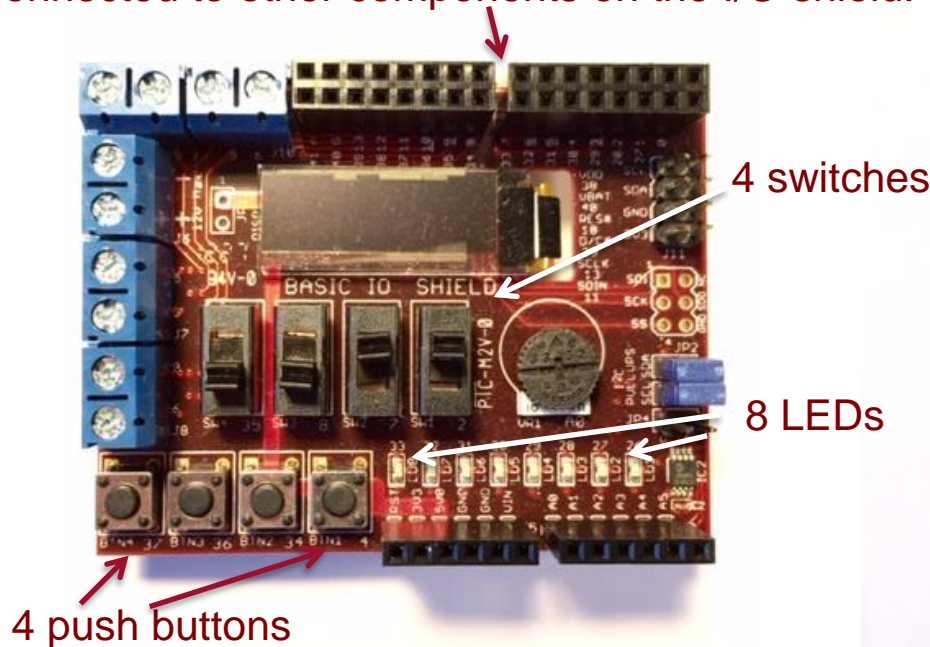


Part I
Basic I/O

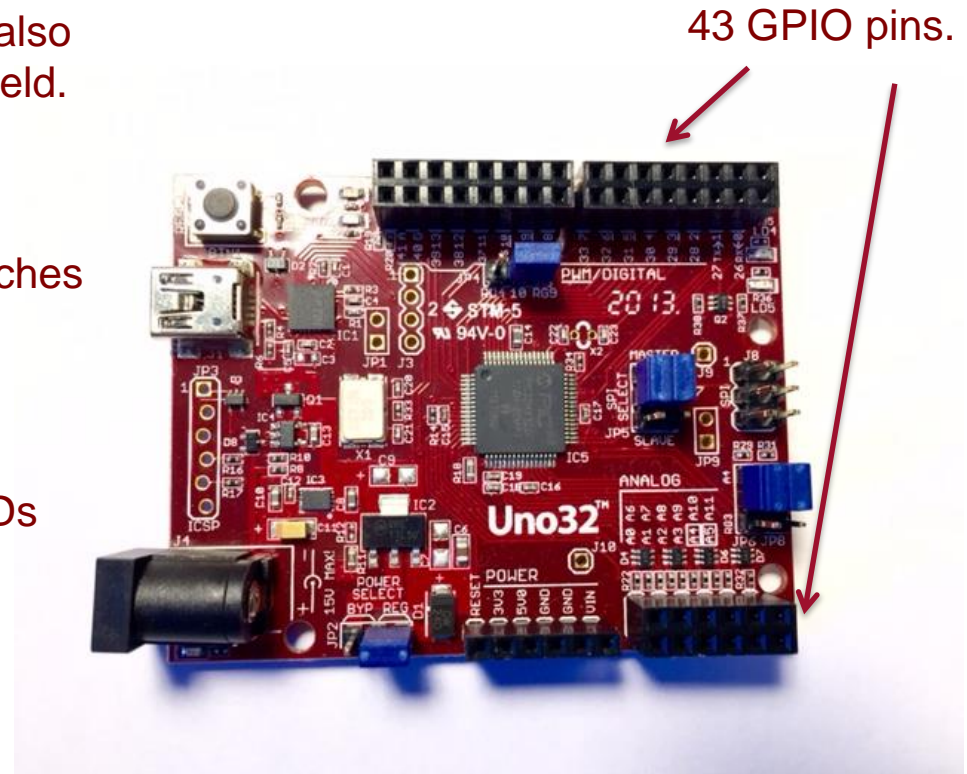
Part II
Buses and
DMA

GPIO and Basic I/O for chipKIT boards

GPIO pins are connected when the shield is attached to the Uno32 board. These pins are also connected to other components on the I/O shield.



Basic IO shield.



Uno32



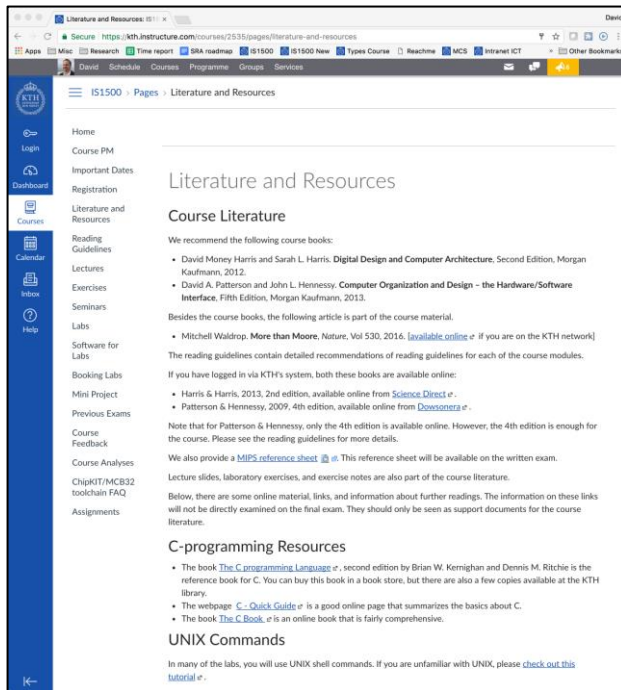
Part I
Basic I/O

Part II
Buses and
DMA

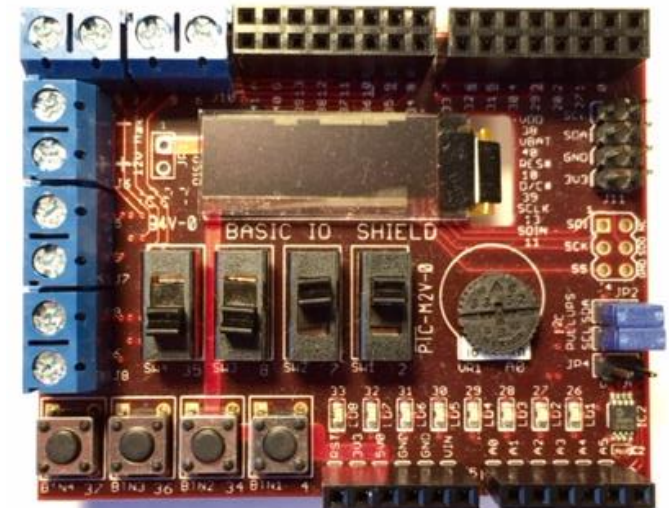
General-Purpose Digital I/O Information

How can we find out how to use the different ports?

Answer: Reference manuals, data sheets, and header files.



Links on the course webpage under "Literature and Resources"



Example: which port and bit should be used for reading the status of push button number 1?



Part I
Basic I/O

Part II
Buses and
DMA

General-Purpose Digital I/O Manuals

Page 8 of chipKIT Basic I/O Shield Reference Manual Page 11-12 of the Uno32 Board reference manual

Appendix A: chipKIT Basic I/O Shield Pinout Table

Uno32 pin #	Max32 pin #	Function	Description
10	10	RES	OLED reset
39	83	DC	OLED data/command select
13	13	SCLK	OLED serial clock
11	11	SDIN	OLED serial data in
40	84	VBAT_EN	OLED VBAT enable
38	82	VDD_EN	OLED VDD enable
33	77	LD1	User LED
32	76	LD2	User LED
31	75	LD3	User LED
30	74	LD4	User LED
29	73	LD5	User LED
28	72	LD6	User LED
27	71	LD7	User LED
26	70	LD8	User LED
4	4	BTN1	Push button
34	78	BTN2	Push button
36	79	BTN3	Push button

3.7.1 Pinout Table by Logical Pin Number

chipKIT Pin #	Connector Pin #	PIC32 Pin #	PIC32 Signal
0	J6-01	34	U1RX/SDI1/RF2
1	J6-03	33	U1TX/SDO1/RF3
2	J6-05	42	IC1/RTCC/INT1/RD8
3	J6-07	46	OC1/RD0
4	J6-09	59	RF1
5	J6-11	49	OC2/RD1
6	J6-13	50	OC3/RD2

chipKIT pin #4 uses port **RF1**, which means that **TRISF** and **PORTF** with bit index **1** should be used.


Push button **BTN1** uses Uno32 pin #4.



Part I
Basic I/O

Part II
Buses and
DMA

Writing the I/O C code

```
TRISE &= ~0xff; /* Port E is used for the LED
                  Set bits 0 through 7 to 0 (output) */
TRISF |= 0x2;  /* Set bit index 1 to 1 (input) */
while(1)
    PORTE = (PORTF >> 1) & 0x1;
```

Light up the LED (PORTE)
if the button is pushed.

Extract bit 1 for representing
the push button (PORTF).

What is the
program doing?



The magic is defined in pix32mx.h

```
#define TRISF          PIC32_R (0x86140)
#define TRISFCLR       PIC32_R (0x86144)
#define TRISFSET       PIC32_R (0x86148)
#define TRISFINV       PIC32_R (0x8614C)
#define PORTF          PIC32_R (0x86150)
```

A C macro (#define)

Volatile pointer that is
dereferenced!

```
#define PIC32_R(a) *(volatile unsigned*)(0xBF800000 + (a))
```



Part I
Basic I/O

Part II
Buses and
DMA

Part II

Buses and DMA



Part I
Basic I/O



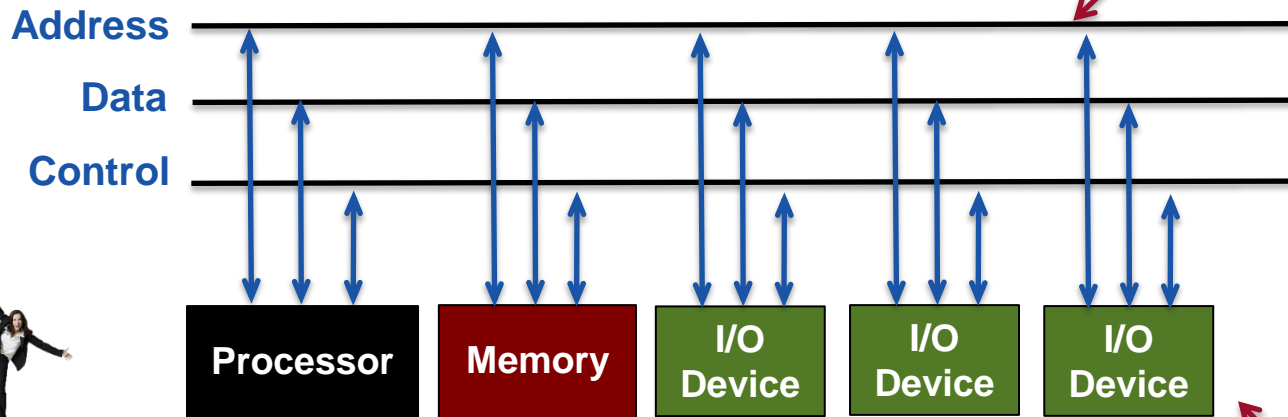
Part II
Buses and
DMA

Bus: Address, Data, and Control

A **bus** is a shared interconnection network where the processor can communicate with different **peripherals**, such as memory and I/O devices.

Many different types: AMBA, Avalon, Wishbone, etc.

When writing or reading to a peripheral, the **address** is given on the address lines. The address can for instance be 32 bits.



Data is written to the data line. The data line can for instance be 32-bit

The **control** lines may include handshaking signals.

Note that modern computers often separate I/O and memory.

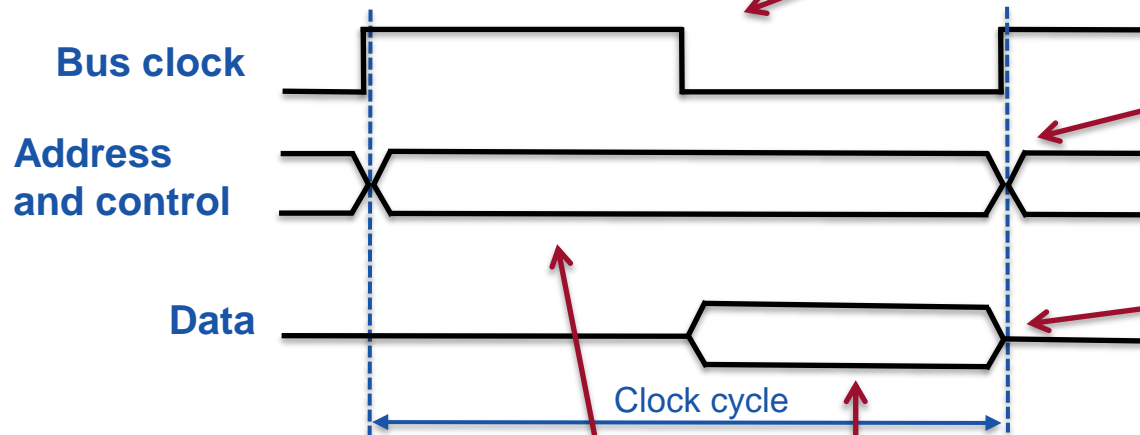
When a peripheral is not the **bus driver** (not communicating), the output must be turned off (have high impedance). This is done by using **tri-state gates**.

Each peripheral has a **unique address** (or range of addresses).

Synchronous Bus

On a **synchronous bus**, data transmission is synchronized using a bus clock.

The bus clock triggers the peripherals to send or receive data



1. For a **read operation**, a control signal indicates that it is a read and the processor puts the address on the address lines.

2. All devices reads the address and the device with the assigned address answers by putting out the data.

For a **write operation** (not shown above) the processor puts both the data and the address on the bus. The device with the assigned address then reads the data.

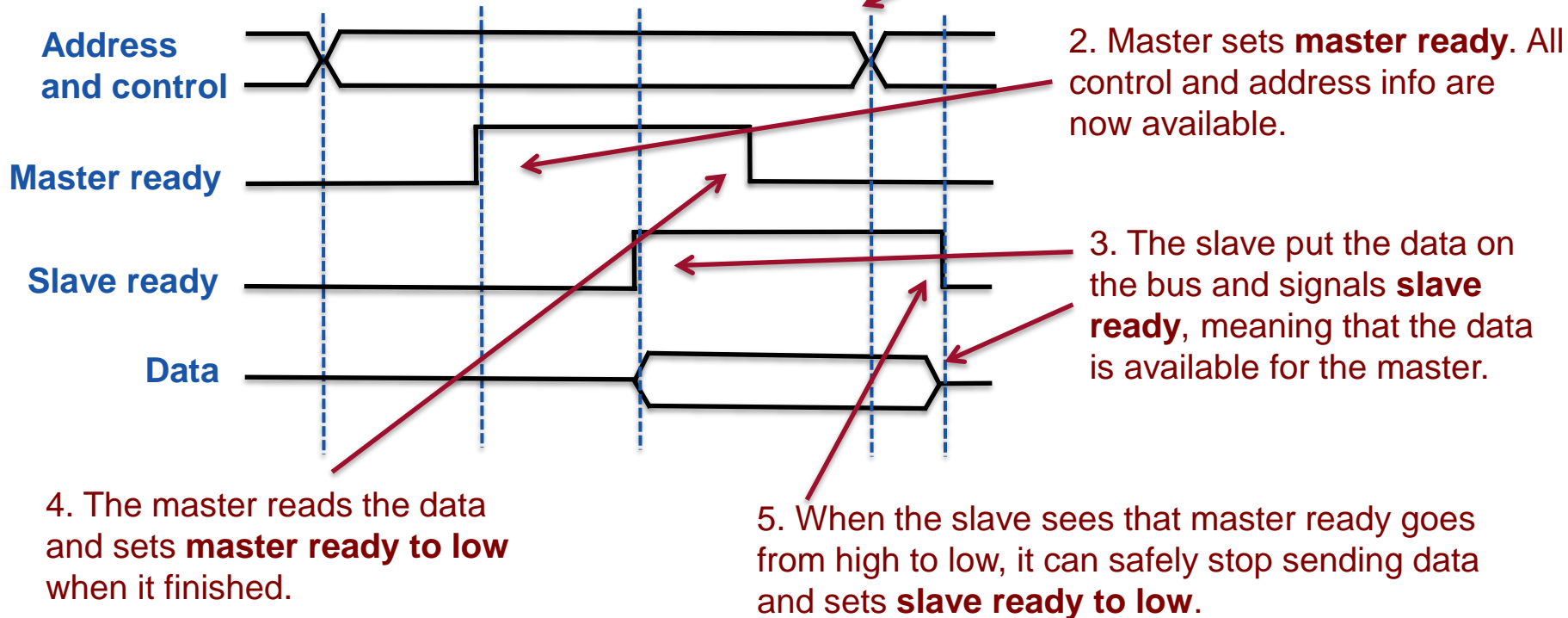
Convention: The diagram with two lines shows that the data or the address can have bits set to both high and low.

Notes: i) This is an idealized figure. In reality, there is a delay in the communication. ii) Synchronous buses can be designed so that it takes several cycles to transmit data.

Asynchronous Bus

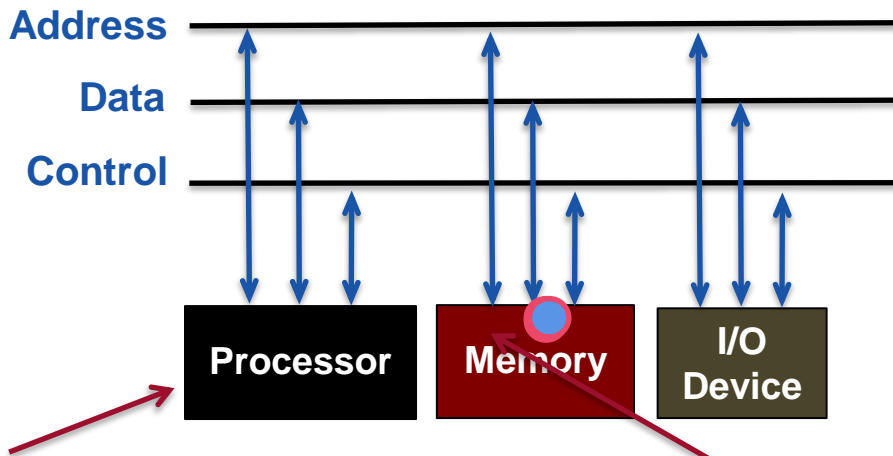
An **asynchronous bus** does not have a clock. Instead, a handshaking protocol is used between the master and the slave (the communicating peers).

Example of an asynchronous read



Direct Memory Access (DMA)

Problem: Programs often move data between different I/O devices. One option is to use the processor to move the data.

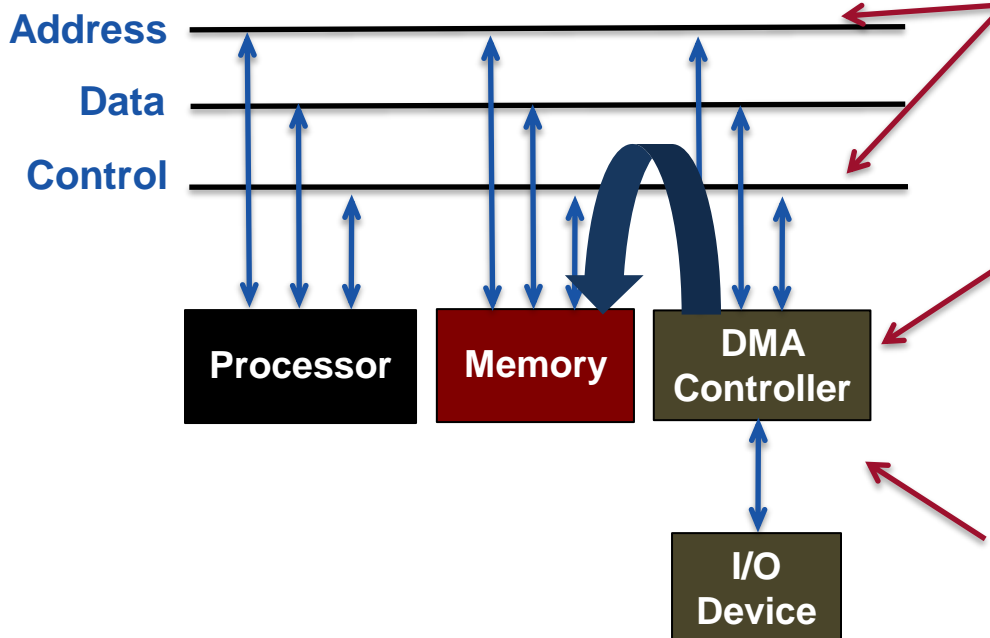


If a lot of data should be transferred between I/O devices (e.g., disk or Ethernet) and memory, or between different parts of the memory, we do not want the processor to be occupied doing a lot of loads and stores.

Processor is busy transferring data and cannot perform any other “useful” work. Also, moving data e.g. word-by-word like this is ***slow and inefficient***.

Direct Memory Access (DMA)

Direct Memory Access (DMA) enables a memory transfers to occur, without continuous intervention by the processor.



The processor issues an **DMA request**, stating addresses and a word count (how much data that should be transferred), stride (the increase in address), etc.

A **DMA controller** takes care of performing the actual transfer. The processor can perform other tasks in parallel.

The **DMA controller** may issue an interrupt request (IRQ) when the transfer has finished.

Yes, there will be coffee in just a second...



Part I
Basic I/O

Part II
Buses and
DMA



Reading Guidelines



Module 2 (I/O Systems)

H&H Chapters 8.5-8.7

For the labs, focus on 8.6.2-8.6.5 (GPIO, Timers, and Interrupts).

The rest is useful for the project.

Reading Guidelines

See the course webpage for more information.

Part I
Basic I/O

Part II
Buses and
DMA



Summary

Some key take away points:

- **Memory mapped I/O** uses a portion of the address space for communicating with I/O devices.
- **GPIO** can be enabled as either input or output. It is very common in embedded systems.
- **Buses** are either synchronous or asynchronous.
- **DMA** enables fast memory transfer without the need of the processor.



Thanks for listening!

Part I
Basic I/O

Part II
Buses and
DMA