



ECE342 – Computer Hardware

Lecture 01

Bruno Korst, P.Eng.

Agenda

- Administrative matters
- Introduction to the course

Administrative matters

- Introductions
- 3 Lectures a week, M@4pm, T@5pm, R@4pm
- Lab on W@9am and F@9am – BA3155 exclusively
 - Lab workstations equipped for ECE342
- Lab manager Aslan Hepdogru (BA3115)
- This course has a heavy emphasis on **doing** (lab/practical work)

Administrative matters

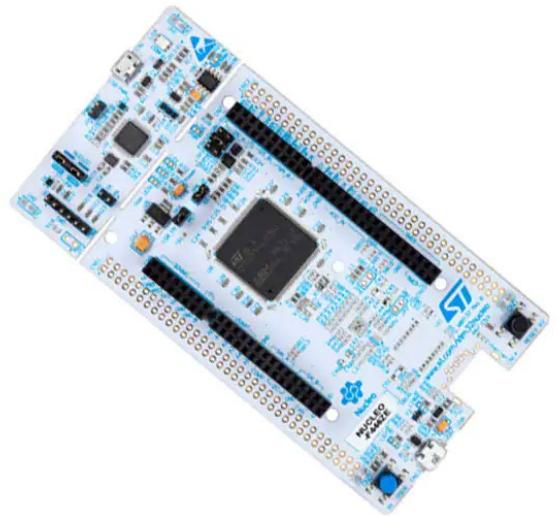
- Course will teach embedded systems from “off-chip” perspective and “on-chip” perspective.
- Students are required to purchase a microcontrolled-based platform, which will be used in all lab experiments (CAD\$21 from Digikey.ca)
 - The lab stations also provide the same, but not to sign-out
 - The platform chosen is the STM32F445ZE NUCLEO board, from STMicro
 - A very popular family of microcontrollers, widely used in industry.
- Work in the lab is done in pairs, but it is expected that much of the development will be tried at home

Administrative matters

- Marks breakdown:
 - Labs 25%
 - Midterm 25%
 - Two Quizzes 5% (combined)
 - Final exam 35%
 - Project 10%
- However, the project is OPTIONAL
 - The 10% of the project mark is distributed proportionally across the other parts
 - Check grading without project on Q

Administrative matters

- The Labs
 - Intro to ST Nucleo platform (connect and make things blink)
 - Interrupts
 - Intro to ADC and DAC
 - Floating point vs. Fixed Point representations
 - On Chip Buses
 - Interface with VGA Camera
 - Peripherals and DMA



Administrative matters

- The project
 - Students who desire to do a project will provide a 1 page description of the intended project (March 13);
 - Students are encouraged to explore an idea of their own;
 - Prior to starting, students will receive feedback on the plan and complexity, to make sure they are on the right track;
 - Projects should show a working design, which will be presented to instructor and TAs during a lab session (April 3);
 - Instructor and TAs will assist in guiding you through the work;

Administrative matters

- Other assessments
 - Two in class quizzes – Jan 26 and Mar 13
 - A midterm – Feb 13
 - A final (to be scheduled)
- The assessments will include questions related to the labs, in addition to more theoretical questions, e.g.
 - Timing
 - Floating point representation
 - Pipelining
- Marks paint a nice picture. Making things work help you tell a great story.

Any questions?

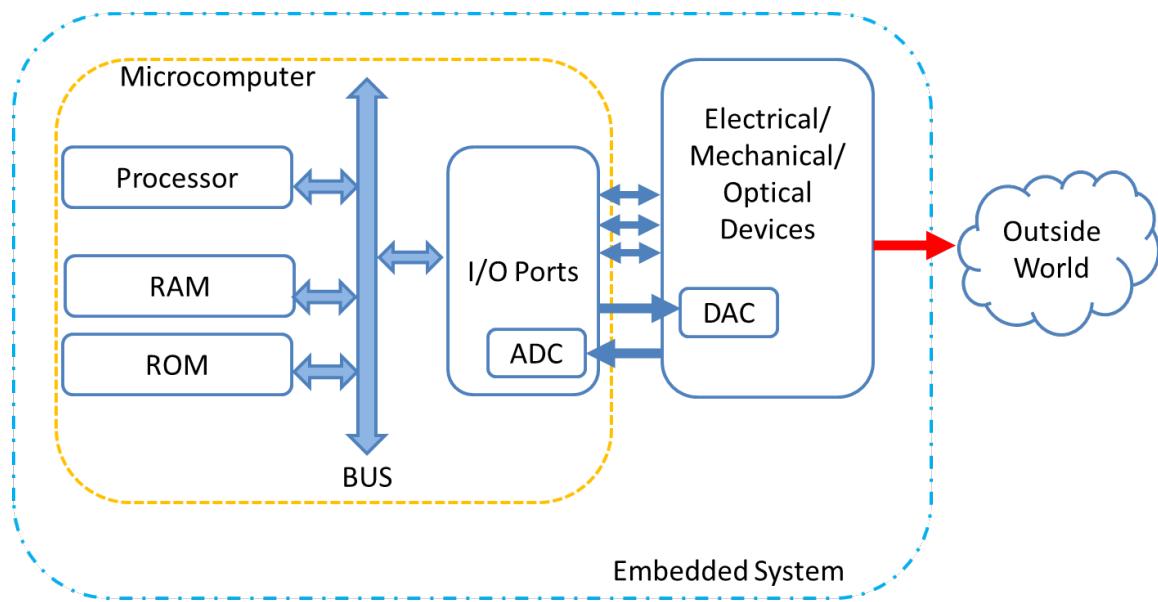
Introduction

- Embedded Systems (some of it a review)
 - Why is it called that way
 - Parts of an embedded system
 - Characteristics and constraints
 - Real-time requirements
 - An example
 - General architectural details
 - Harvard vs. Von Neumann
 - Memory Map

Embedded Systems

- Definition
 - It's a physical system that employs computer control for a specific purpose
 - Ex: cruise control in a car
 - Software interacts closely with the hardware handling “outside world” signals
 - “Embedded” = hidden inside, “on board”

Embedded Systems



Embedded Systems

- Processor
 - Hardware that executes a set of instructions, stored in memory in a well-defined manner
- Memory
 - Collection of hardware elements that store information
 - Can read from and write to specific addresses
 - Can be byte addressable, word addressable

Embedded Systems

- Memory (cont'd)
 - ROM – read-only, non volatile
 - Program resides in non-volatile FLASH memory
 - RAM – random access, temporary, volatile
 - Faster
- Ports (I/O – input/output)
 - Physical connection to the outside world
 - Connection follows a variety of protocols, providing interface to other devices

Embedded Systems

- Bus
 - Collection of wires used to pass information between modules
 - Information: data, address, control
- Microcomputer
 - Hardcore and Softcore
 - Hardcore: microprocessors/microcontrollers
 - Softcore: FPGA (a.k.a SoC – system on a chip or SoPC – system on a programmable chip)
 - on-chip modules interacting through on-chip means

Embedded Systems

- Microcomputer (cont'd)
 - Wide range of devices
 - Different speed, memory capacity, number/type of I/O, computational power, energy consumption
 - Microcontrollers
 - Devices that have processor, memory, I/O ports, timers, ADC/DAC in a single package
 - Low cost, low consumption, small size
 - Ex: TI MSP430, STM32 (ARM), PIC family, etc

Embedded Systems

- Operating System
 - A mechanism that allows for switch in execution between processes
 - Each process uses a set of registers, memory space, I/O, etc.
 - Handle multiple operations needed at varying times/rates.
 - No operating system
 - single program handles everything
 - low cost, low complexity, low performance
 - Ex: coffee machine

Embedded Systems

- Operating System (cont'd)
 - With operating system
 - Use a PC-type architecture, complex, high-performance
 - Ex: smart phone, car entertainment system, etc.
- Interaction with outside world: users
 - Simple user interface ex: smart tooth brush
 - Complex user interface – user “experience”
 - UI → subset of UX

ECE342 – Computer Hardware

Lecture 02

Bruno Korst, P.Eng.

Agenda

- General Embedded systems (cont'd)
- Specifics: Interrupts

Embedded Systems

- Operating System
 - A mechanism that allows for switch in execution between processes
 - Each process uses a set of registers, memory space, I/O, etc.
 - Handle multiple operations needed at varying times/rates.
 - No operating system
 - single program handles everything
 - low cost, low complexity, low performance
 - Ex: coffee machine

Embedded Systems

- Operating System (cont'd)
 - With operating system
 - Use a PC-type architecture, complex, high-performance
 - Ex: smart phone, car entertainment system, etc.
- Interaction with outside world: users
 - Simple user interface ex: smart tooth brush
 - Complex user interface – user “experience”
 - UI → subset of UX

Embedded Systems

- Data exchange
 - Data presented in a variety of formats
 - Analog data – encoded as continuous variable, read through an analog-to-digital converter (ADC)
 - Digital data – encodes a variety of information
 - Timing data: encodes frequency/period, pulse width, phase shift
 - Variety of protocols
 - » Serial interface: binary data available one bit at a time
 - » Parallel interface: binary data available in groups of bits/lines simultaneously

Embedded Systems

- Real-Time or not?
 - “real-time” = an upper bound in time constraints that the program must follow
 - Input-calculation-output within a (fast) max time t
 - Ex: medical device monitoring O₂ levels during surgery
 - Non real time – time constraint not stringent
 - Ex: your APS105 program running on your PC
 - Output typically not needed “immediately”

Embedded Systems

- Real-time (cont'd)
 - Factors affecting constraints
 - Interface latency – response time must be less than worst-case upper bound (less than...)
 - Periodic tasks – operation must be concluded within period between tasks
 - Ex: a timed interrupt
 - Ex: guitar signal routed through 5 different effect consoles
 - From pluck to loudspeaker, must be <200ms
 - Why? Ear perceives “delay” when difference ~200ms

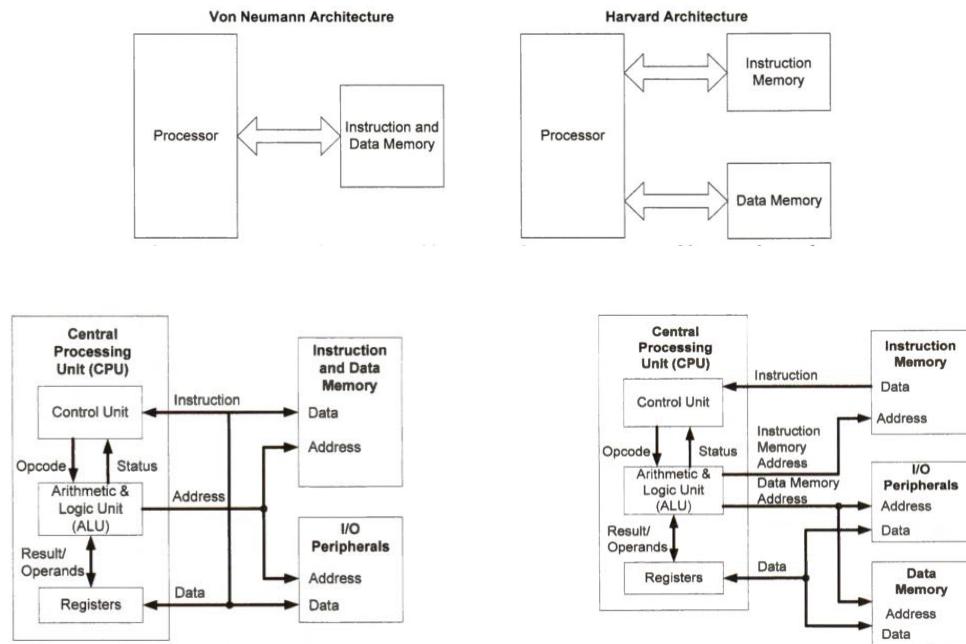
Embedded Systems

- Interface with external hardware
 - Hardware *will demand action* from system
 - Acknowledgement
 - Response to interrupt
 - Interrupt service routine (ISR)
 - Parameterless subroutine call
 - Main program is foreground thread, ISR is an occasional background thread
 - Multiple interrupts organized as a “vector”, each mapping to a subroutine, each with its own priority level (ex: int0 typically reset – highest priority)

Embedded Systems

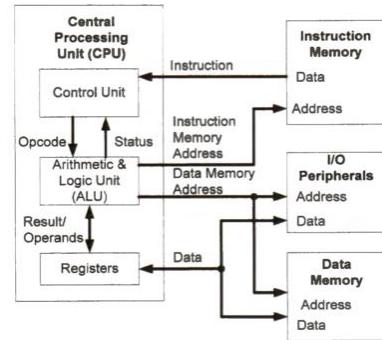
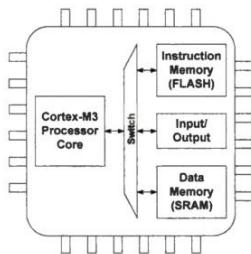
- Processor Architecture
 - Historically speaking, there are two architectures
 - Von Neumann
 - Program memory and data memory are part of the same continuum, and share the same “routes” for read/write
 - Retrieving program and data at the same time runs into a bottleneck
 - Harvard
 - Separate program and data memory, separate paths for data, address and control (different widths as well)
 - Program: non-volatile, data: volatile

Embedded Systems



Embedded Systems

- Harvard
 - Access instruction, R/W data at same time
 - Different size, separate lines (buses)
 - Can use slower clock
 - Save energy

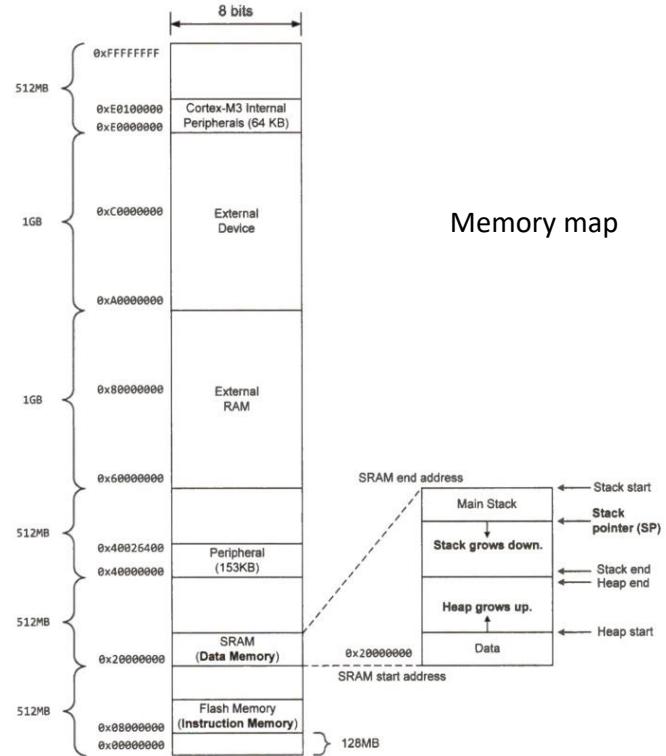
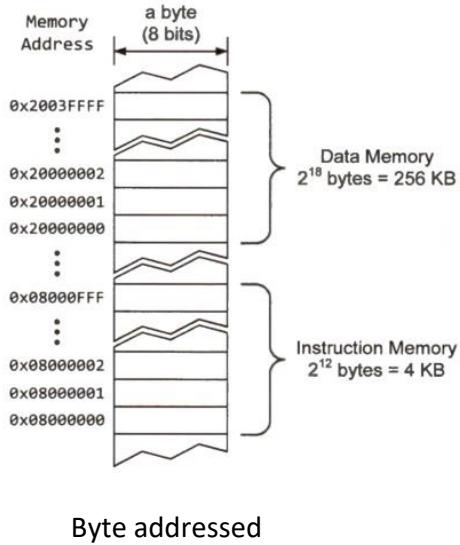


Embedded Systems

- Memory
 - Program memory – non-volatile
 - FLASH
 - EEPROM (electrically erasable programmable read only memory)
 - Data memory – volatile
 - RAM – random access memory

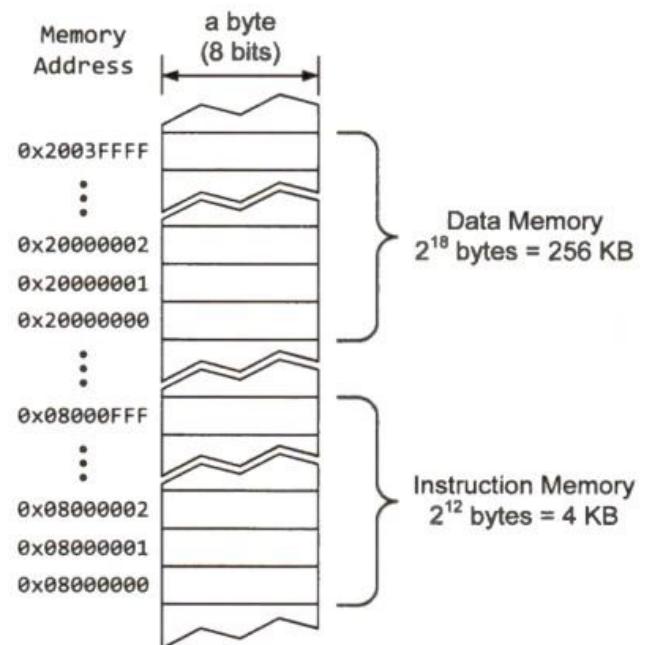
Embedded Systems

EXAMPLES



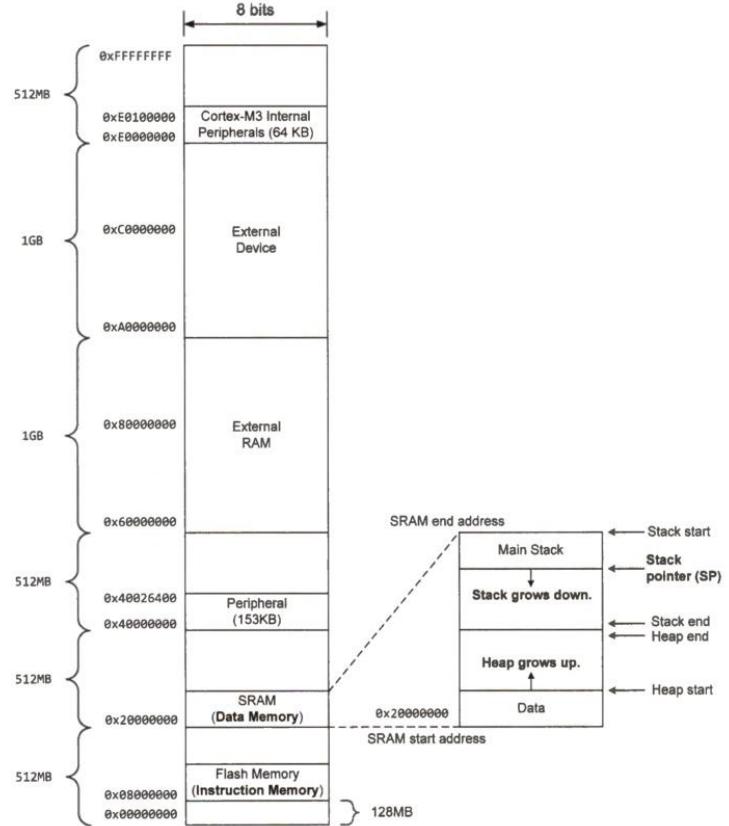
Embedded Systems

- Memory map
 - Design determines range and addresses per type of memory



Embedded Systems

- Maps also
- “Peripherals” → GPIO/ USART
- External RAM → off chip RAM
- External Device → SD card
- Internal peripherals
 - Modules within the device

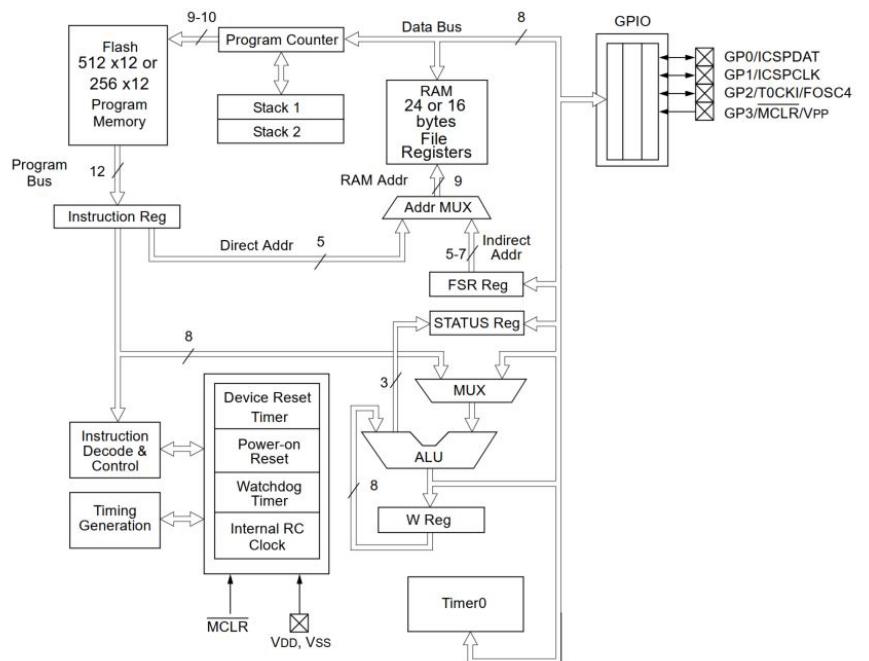


Bruno Korst - Winter 2023

15

Embedded Systems

- A simple architecture
 - PIC Microcontroller



Bruno Korst - Winter 2023

16

Specifics: Interrupts

- From 243..
- Interrupts are hardware-generated exceptions
 - I/O device sends request to processor (IRQ) – asynchronous!
 - Processor stops execution of current program, remembers where it is (stack pointer), and go to service the interruption.
 - “Interrupt Service Routine” (ISR)
- Say you have a program in an infinite loop “waiting” for things to happen
 - The interruption will cause a branch to an “interrupt handler”
 - That part of the code will run, when done, returns

Specifics: Interrupts

- Recall: an assembly program on infinite loop

```
Loop: instruction
      instruction
      instruction
      ...
B Loop
```
- Program sits in loop, interrupt happens (via request from I/O or other), program stores current location, branches to service interrupt, comes back, carries on.

Specifics: Interrupts

- Infinite loop in C
 - Within the main function
 - You will have the library calls, preprocessor directives (#), maybe some definitions, declarations, and main
 - Within main you will have declarations, functions initializing different parts of the system and the infinite loop

```
int main(void)
{
    // initialize subsystems
    DSys_Init();
    // main stalls here, interrupts drive operation
    while(1) {
        ;
    }
}
```

Specifics: Interrupts

- Different processors will require different configuration of interrupts
 - May have a Generic Interrupt Controller (GIC) (as you saw in 243)
 - A memory-mapped IO device that receives all IRQ signals and sends the request to the processor
 - May have registers to be configured for multiple interrupts available
 - A register to enable, a register to set a flag, a register to set priority...
 - May have a graphical interface that you click (very convenient)

Specifics: Interrupts

- What is the interrupt with highest priority?
- The hard way to set interrupts is via “masking”
 - This is a logic operation between the bits of the register and the “mask” that you wish to apply to it
 - To CLEAR – AND with zeros in the register area to be cleared
 - To SET – OR with ones what you want to set
 - To TOGGLE – XOR with one the bits to be toggled
 - It is the hard way because you’d have to do this for ALL interrupt related registers (very easy to miss a bit!)

Specifics: Interrupts

- Some processors have provisions for non-maskable interrupts (a process that you do not want to interrupt), some have “global enable” (you must enable all first...)
 - Masking is also a means to enable/disable interrupts for the execution of critical tasks
- There are also multiple sources of interrupts, not only external. Each processor will have its own resources
 - Timers, Memory, a number of externals, DMA, etc.

Specifics: Interrupts

- Recall that in C we use pointers to gain access to specific addresses (APS105!)

```
volatile int * somePtr = 0xFF200000;
volatile int * anotherPtr = 0xFF200040;

int a;

while(1) {
    a = *somePtr;
    *anotherPtr = a;
}
```

- Remember “**what’s at**” -- *ptr read “what’s at ptr” (dereferencing)
- Volatile** means the variable will always be accessed by using its memory address

Specifics: Interrupts

- Interrupt Vector table
 - For interrupt-based operation, a file (.asm) will hold a “what to do” for all interrupts of a processor
 - This may be “prepared” for you depending on the toolchain used

```
.ref _c_int00
.ref _Codec_ISR
.sect "vectors" ; this will be a sector on a memory map
RESET_RST: MVKL.S2 _c_int00, B0
            MVKH.S2 _c_int00, B0
            B .S2 B0
            NOP
            NOP
NMI_RST: b NMI_RST ; stall here if interrupt occurs
            NOP
            NOP
RESV1:   b RESV1 ; stall here if interrupt occurs
            NOP
            NOP
RESV2:   b RESV2 ; stall here if interrupt occurs
            NOP
            NOP
INT4:    b INT4 ; stall here if interrupt occurs
            NOP
            NOP
INT5:    b _Codec_ISR
            NOP
            NOP
```

Specifics: Interrupts

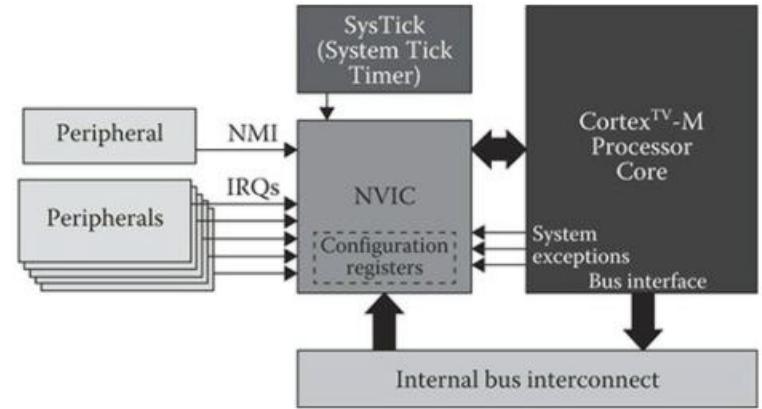
- Polling vs. Interrupt operation
 - In polling, even though the processor is in an infinite loop, it is constantly inquiring whether there is data coming from a particular source.
 - When the data is available, it is handled by a routine
 - In interrupt operation, the processor will address a particular interrupt service routine (pertaining to that interrupt)

Specifics: Interrupts

- For the microcontroller we'll be using, it is designed with a Nested Vectored Interrupt Controller (NVIC)
 - Able to process interrupts from peripherals and exceptions from core
 - Allows priority levels to exceptions and interrupts (except some...)
 - “exceptions” are software generated interrupt (abnormal program termination)
 - Allows for masking (except NMI)
 - Latency is ensured at 12 cycles for Cortex M4 cores (that's the F446)

Specifics: Interrupts

- NVIC on Cortex-M



ECE342 – Computer Hardware

Lecture 03

Bruno Korst, P.Eng.

Agenda

- Interrupts
- A/D

With thanks to Dr. Matthew MacKay

Interrupts

- We saw that interrupts are asynchronous events that happen and demand service from the processor
 - Interrupt happens → ISR served
 - Changes the program flow
- We cannot assume anything about the program or state of microcontroller upon entering the ISR
- We cannot modify the state of the microcontroller upon exit from the ISR as the continued execution expects the “original” state
- We should have **deterministic** conditions and execute as quickly as possible, as not to interfere with other constraints of the program.
 - (**always same output from a given starting condition or state**)

Interrupts

- When writing an ISR, one should **avoid**
 - Calls to additional subroutines
 - Waiting (fixed delays) --- polling is an example of that...
 - Wait for a status flag
 - Loops without definitive end conditions
 - You want to get out of the ISR and move on
- Interrupts may occur **during** an ISR execution
 - Use priorities (when available) or masking to enable/disable selectively the interrupt sources
- Non-maskable interrupts: bootloader programs, fail-safe, emergency, reset

Interrupts

- The good practice is
 - At the start of the ISR: Set a mask to all PREVENT interrupts
 - Service the ISR
 - Clear the mask to ALLOW interrupts again
- When interrupts are masked (or when an interrupt is disabled) and an interrupt “happens”...
 - Data is buffered until interrupts are enabled back again
 - OR
 - The event is solemnly ignored

Interrupts

- If interrupts happen simultaneously, or if multiple happen while an ISR is serviced
 - They are prioritized (as we saw with a priority register or another provision)
- Say we have a system monitoring a nuclear reactor at multiple locations
 - If two temperatures are equal, all is good
 - If they are not, reactor is entering meltdown
 - Initial values are $\text{temp}[] = (42, 42)$
 - Next set values read are (or will be) (59, 59)
 - That is, it “will be good”...

Interrupts

How do you
find such bug?

```
static int temp[2];

void isrReadTemp()
{
    temp[0] = ** read temperature **;
    temp[1] = ** read temperature **;
}

void main()
{
    int iTemp0, iTemp1;
    while (true)
    {
        iTemp0 = temp[0];
        iTemp1 = temp[1];
        if (iTemp0 != iTemp1)
            ** Sound warning **
    }
}
```

Interrupt occurs here

Interrupts

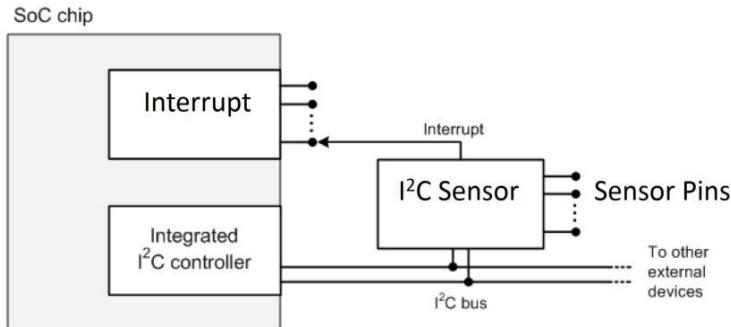
```
static int temp[2];
ISR

void isrReadTemp()
{
    temp[0] = ** read temperature **;
    temp[1] = ** read temperature **;
}

void main()
{
    int iTemp0, iTemp1;
    while (true)
    {
        sei();   Prevent selected interrupts
        iTemp0 = temp[0];
        iTemp1 = temp[1];
        cli();   Allow them back
        if (iTemp0 != iTemp1)
            ** Sound warning **
    }
}
```

Interrupts

- In connecting, SoC/Microcontrollers will have one or more external interrupt pins available.



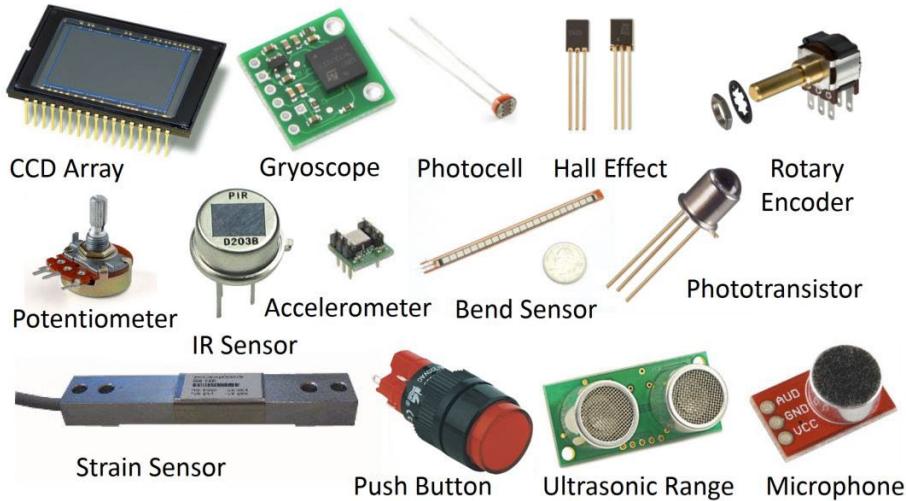
- Number of external interrupts is limited, and sources CAN be combined. However, one must know the source to enter the right ISR...

Interrupts

- Overloading external interrupt can be a problem as one would need to scan multiple devices
- Software interrupts (mentioned prior as an exception, or something going awry) can be used for multi-tasking for instance, or inter-program communication
- Internal features can also generate interrupts
 - Timers – count reached, count reset
 - Protocol (ports) – ready to send, receive
 - A/D – sample ready

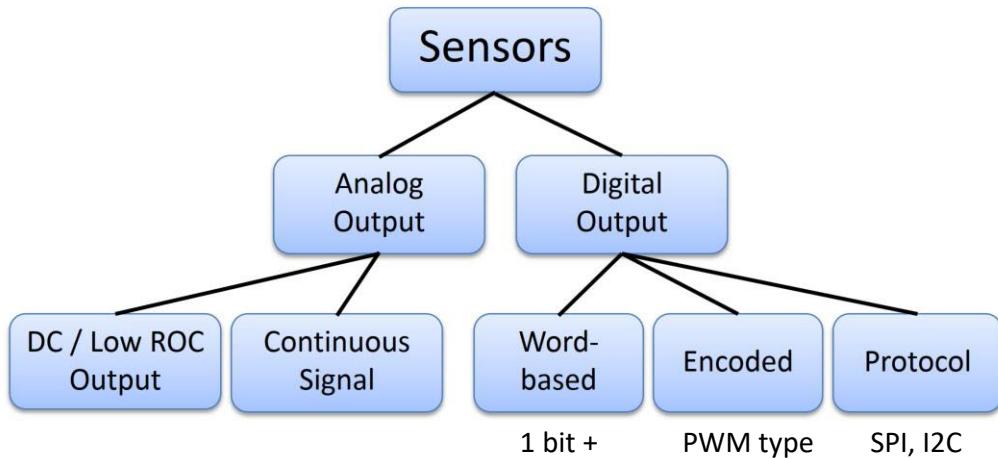
Analog to Digital Conversion

- In embedded systems, interfacing with sensors is paramount



Analog to Digital Conversion

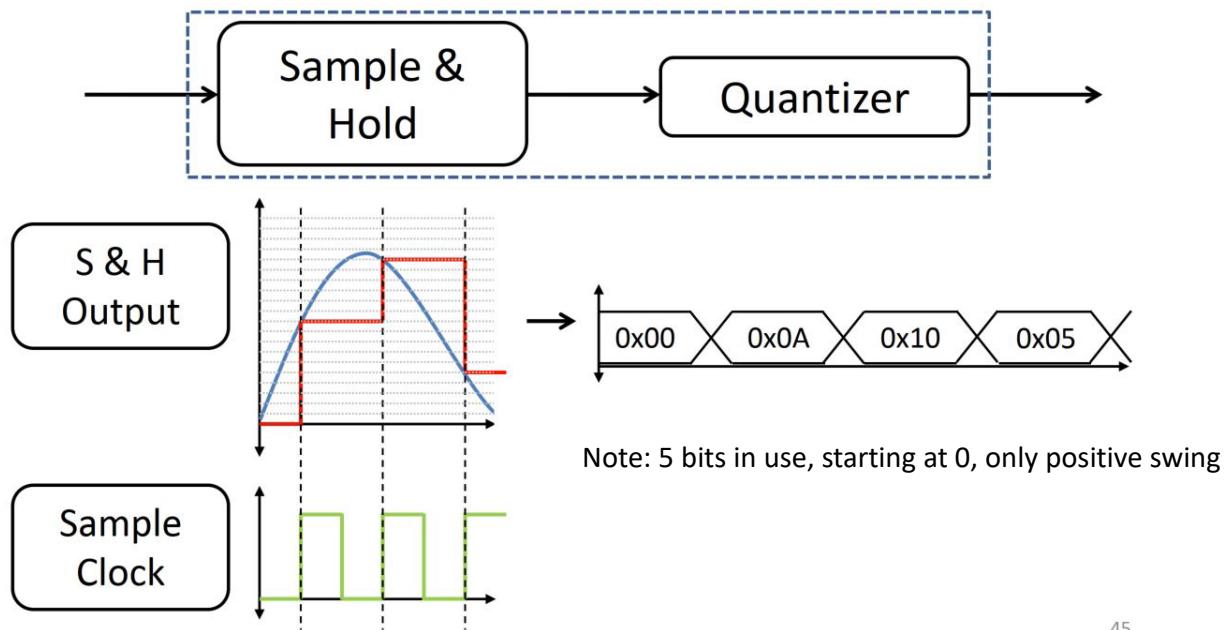
- Sensors are mostly application dependent
- The type of sensor output relates directly to the choice of device in the design



Analog to Digital Conversion

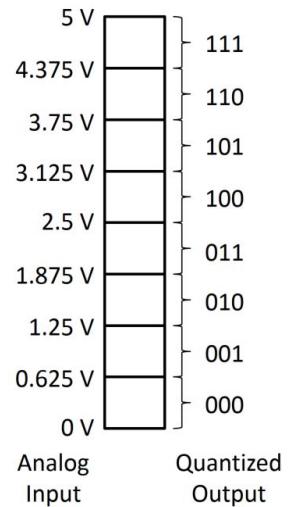
- Will concentrate on **analog** for now
- For analog output, we can have
 - DC or a low rate of change (ROC) signal
 - General continuous time signal
- For a continuous, time-varying signal
 - We need to convert the analog signal with a continuous range into a digital signal with a specified range of discrete values
- There are two stages to ADC: sampling and quantization

Analog to Digital Conversion

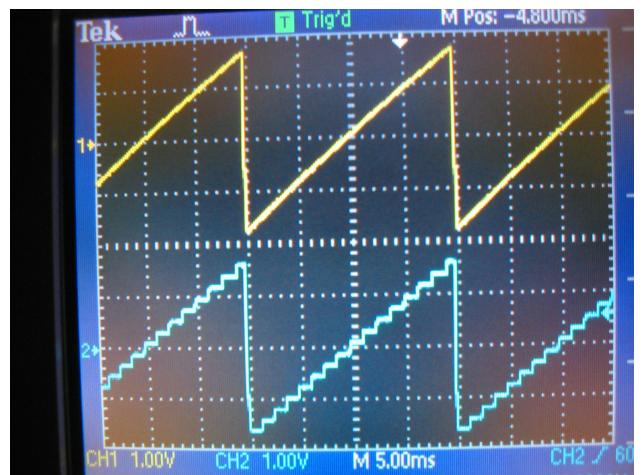
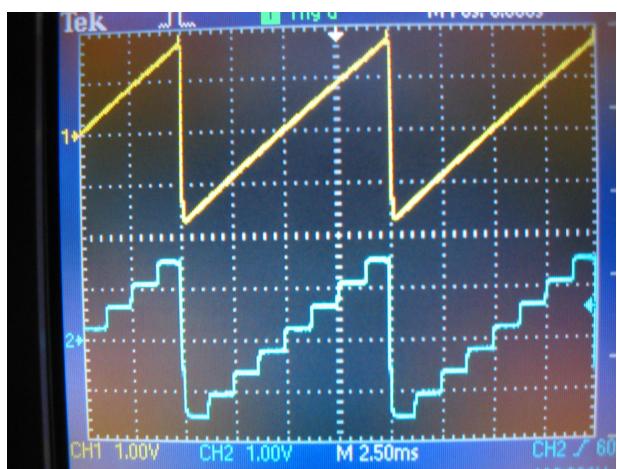


Analog to Digital Conversion

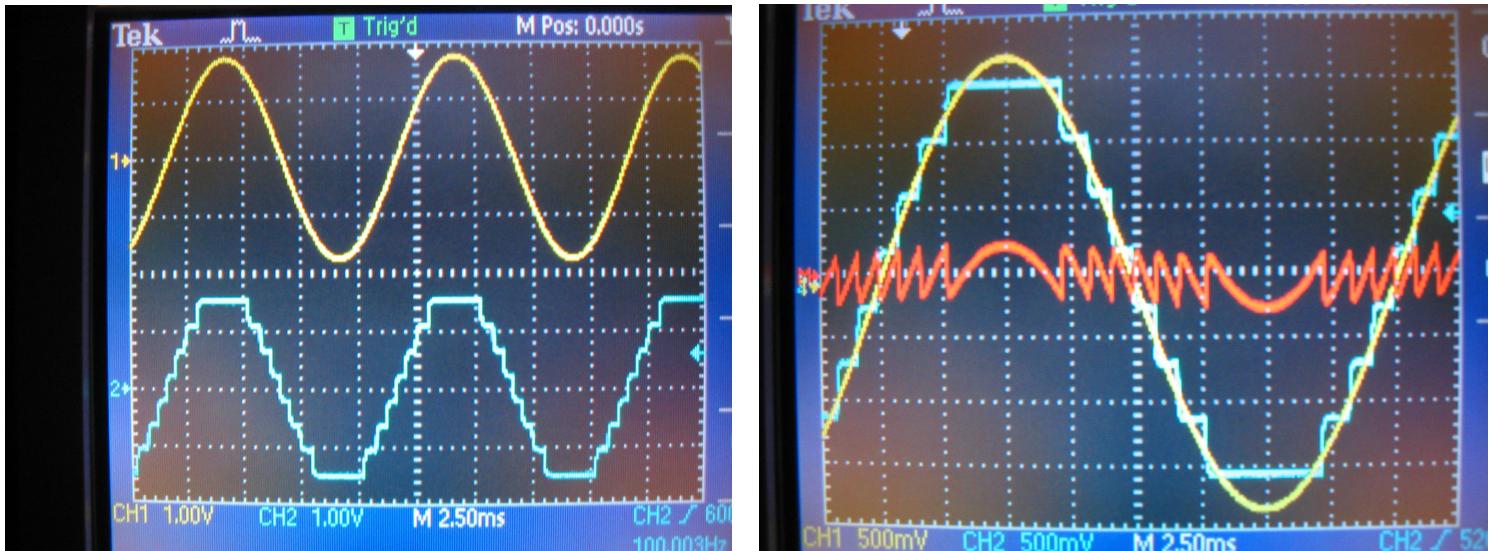
- A converter will have an input range and a reference voltage
 - Reference voltage may be adjustable or fixed
 - What would be an estimate of the quantization error?
 - 3 bit per sample, 5Vpp
 - 16 bit per sample, 3.8Vpp max input



Analog to Digital Conversion



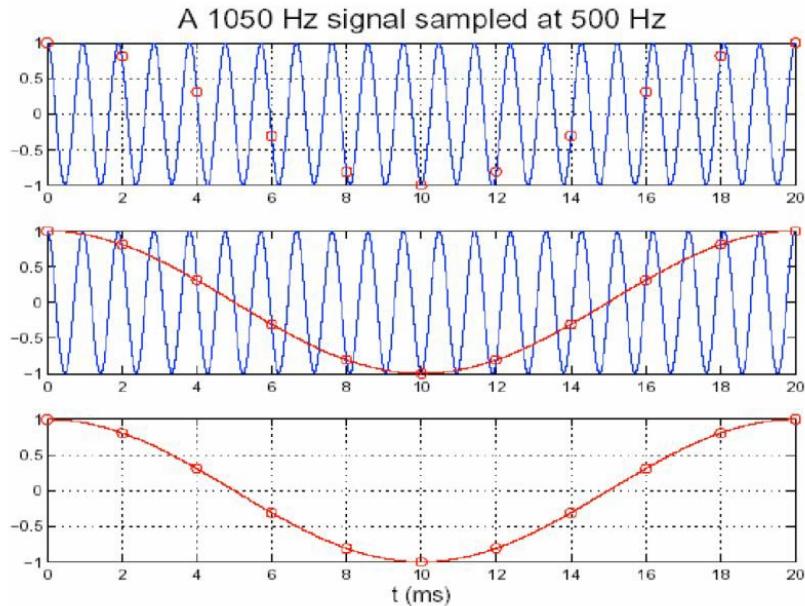
Analog to Digital Conversion



Analog to Digital Conversion

- Typically A/D converters sample data at fixed intervals (*sampling rate*)
 - There is also set up time, hold time, output time that add delay
 - This only comes into play with very fast signals
- Sampling follows the Nyquist theorem
 - The sampling rate should be at least twice the highest frequency present in the signal.
 - Ex: voice range is up to 4KHz → minimum sampling rate is at least 8KHz.
- Sampling “rate” or “frequency” is in number of samples per second
 - Each sample is assigned the number of bits available by the quantizer

Analog to Digital Conversion



Analog to Digital Conversion

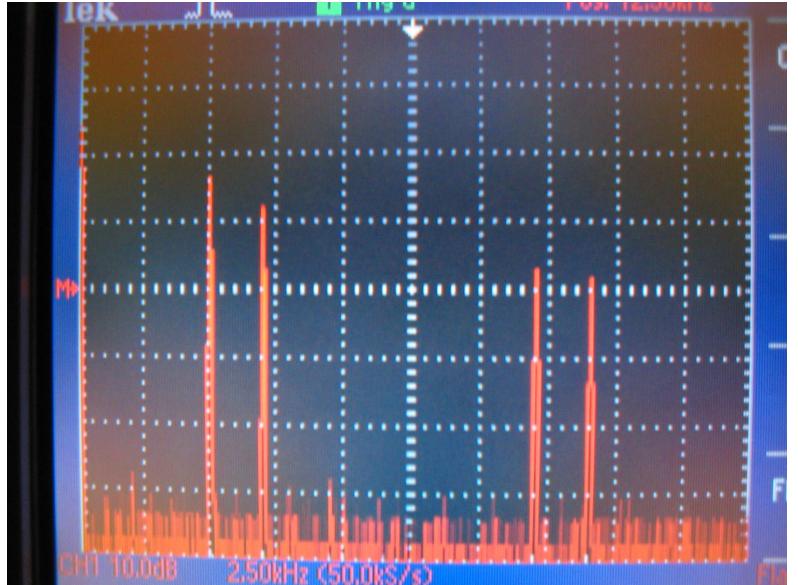
This picture presents a frequency domain plot of a sinusoid being sampled at 12KHz

Between 6Khz and 12Khz we have a mirror image of what we have between DC and 6Khz.

Between 12Khz and 24Khz everything repeats.

Questions:

If input is 5Khz, where is it?
If input is 7Khz, where is it?



Analog to Digital Conversion

- The main points are:
 - Be aware of the kind of analog input you have, in time domain and frequency domain
 - Use the appropriate range (level) on the input of your device, select the appropriate sampling rate
 - Some devices (like the F446RE/ZE) offer A/D conversion internally
 - Their range, speed, accuracy may be limited for your application
 - If so, you will need an external ADC

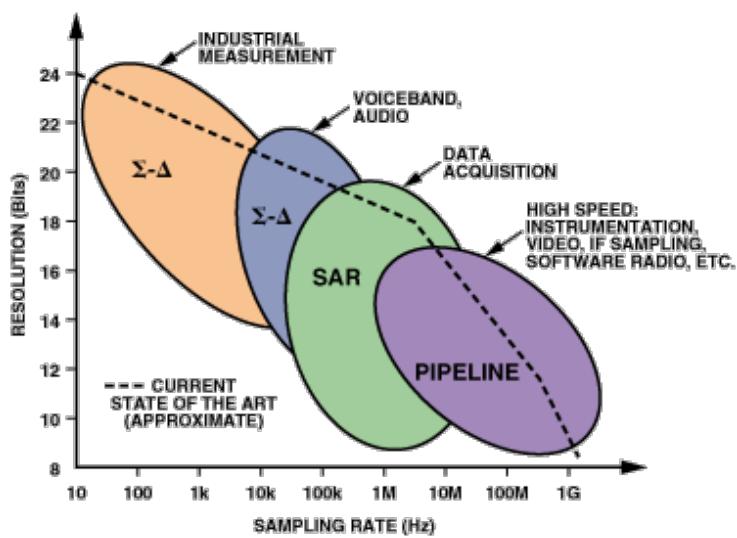
Analog to Digital Conversion

- Internal ADCs must be set
- A reference is set through a register/memory location or an external (voltage) supply;
- Sampling rate, channel select/enable is set via a settings register/memory location
- Requires a value register/memory location
- A status register may be used for conversion status, timing, etc.

Analog to Digital Conversion

- External ADC (ex: TI PCM3003 – search it)
- Wide variety of ADC integrated circuits
 - Pros: Self-contained, simpler to use, faster and offer better specs
 - Cons: price, maybe the communication to processor.
- Sampling (that is, samples per second)
 - Done via external or internal reference (clock) or a separate external or internal oscillator.

Analog to Digital Conversion



ECE342 – Computer Hardware

Lecture 04/05

Bruno Korst, P.Eng.

Agenda

- A/D conversion (wrap)
- Numerical Representation and Arithmetic review

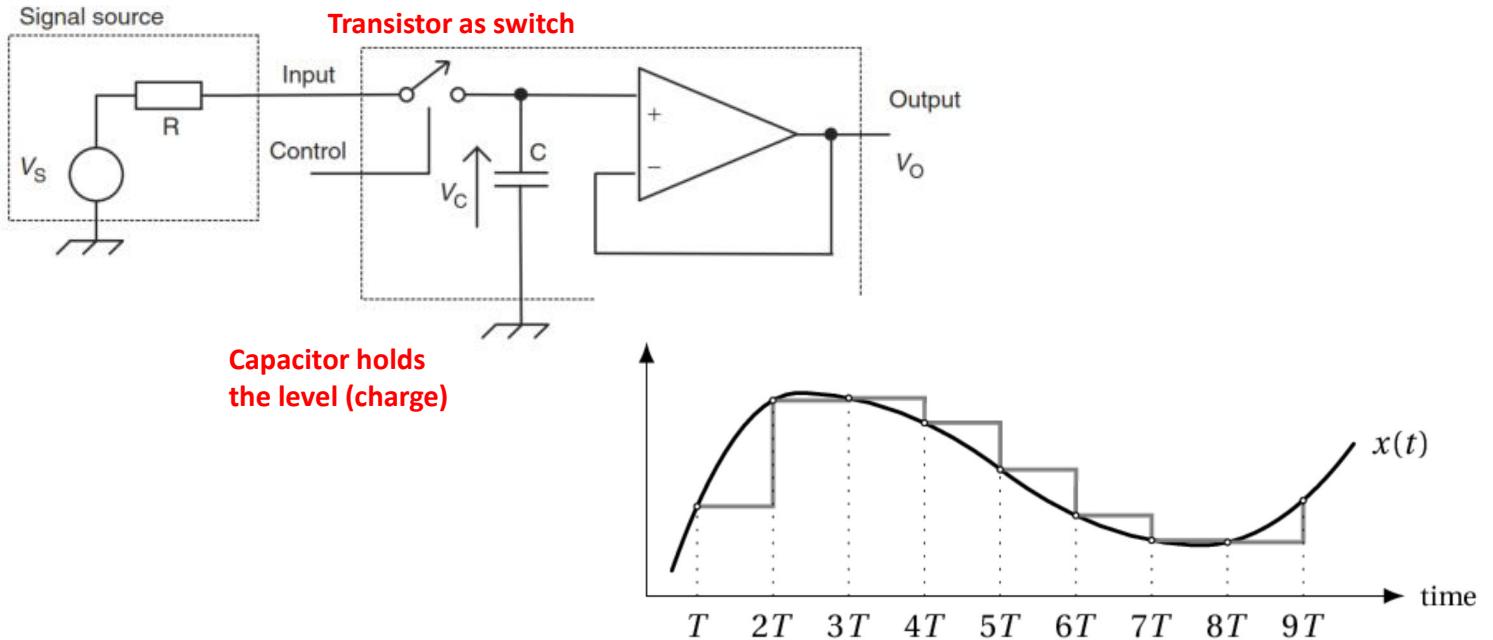
Analog to Digital Conversion

- From last class, the main points are:
 - Key performance parameters: **sampling rate, resolution and power dissipation**
 - Power is particularly an issue in mobile applications
 - Be aware of the kind of analog input you have, in time domain and frequency domain
 - Use the appropriate range (level) on the input of your device, select the appropriate sampling rate and resolution
 - Some devices (like the F446RE/ZE) offer A/D conversion internally
 - Their range, speed, accuracy may be limited for your application
 - If so, you will need an external ADC

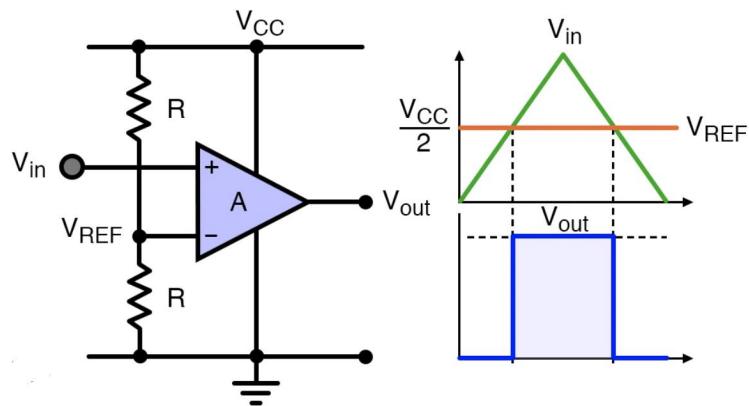
Analog to Digital Conversion

- External ADC (ex: TI PCM3003, TI AIC3104, WM5102, MAX1247)
- Wide variety of ADC integrated circuits
 - Pros: Self-contained, simpler to use, faster and offer better specs
 - Cons: price, maybe the communication to processor.
- Sampling (that is, samples per second)
 - Done via external or internal reference (clock) or a separate external or internal oscillator.

Sample and Hold

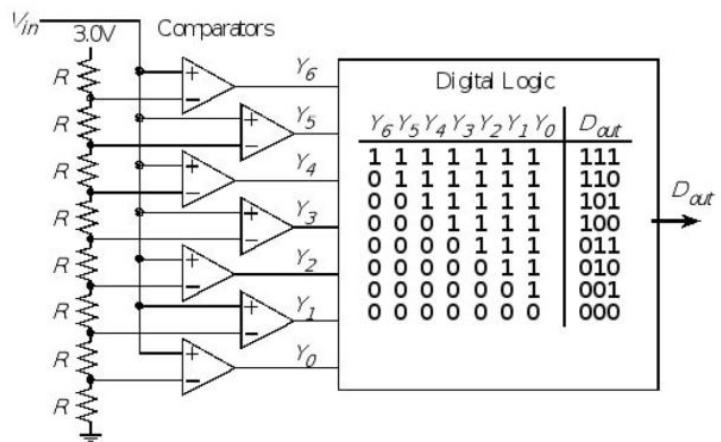


Comparator

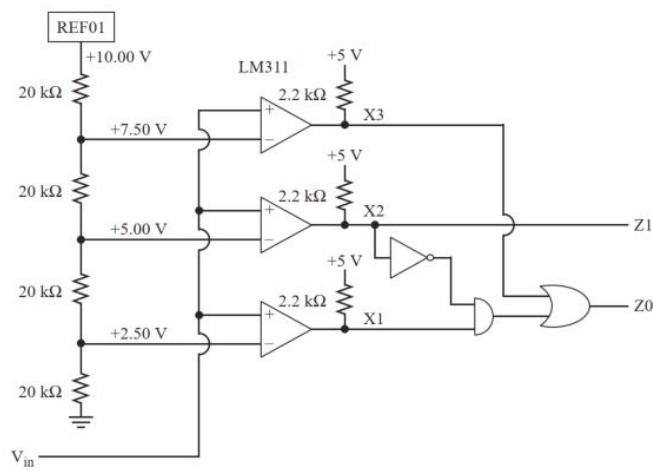


Flash ADC

- Uses a ladder of $2^n - 1$ comparators, quantize input directly
- They are very fast; outputs are available simultaneously
- GSamples per second
- Comparator $\sim 100\text{ns}$



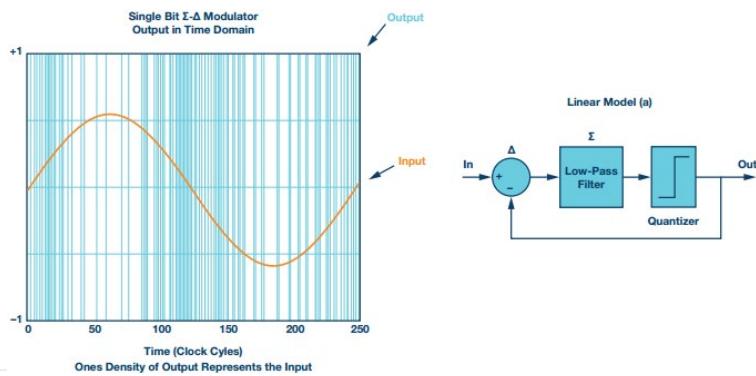
Flash ADC



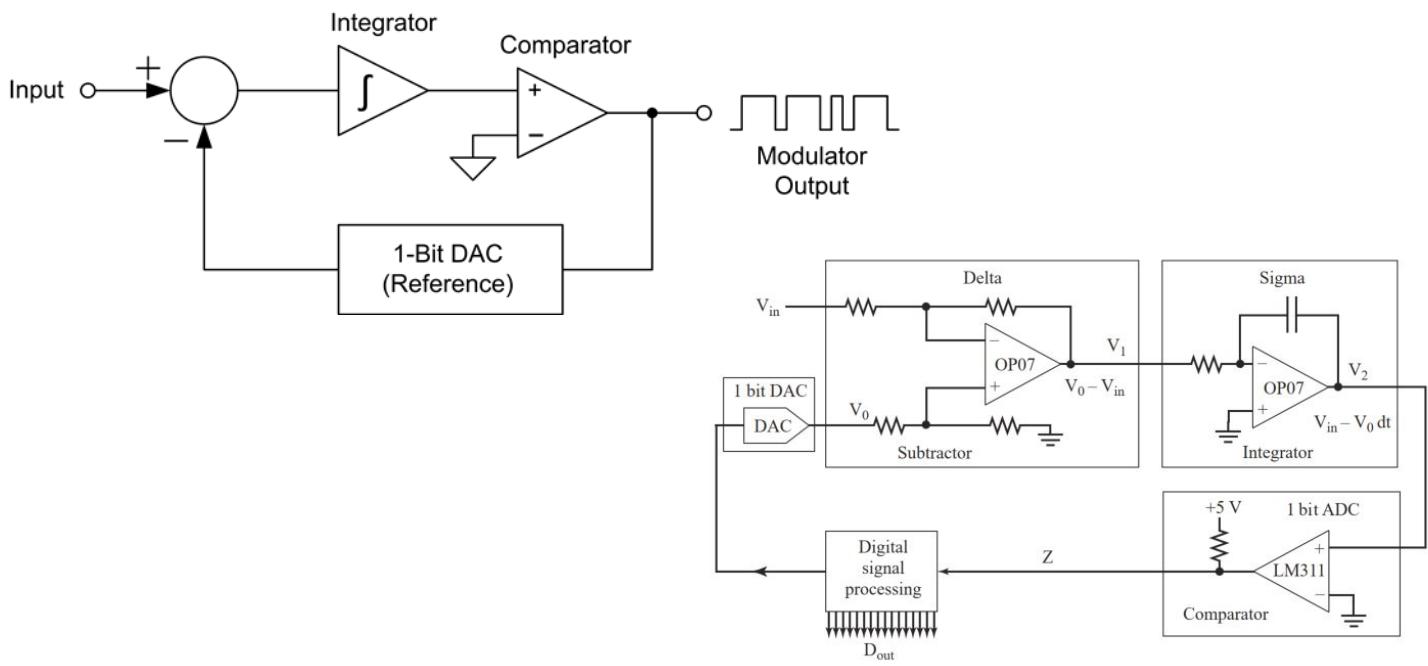
V_{in}	X_3	X_2	X_1	Z_1	Z_0
$2.5 > V_{in}$	0	0	0	0	0
$5.0 > V_{in}$	2.5	0	0	1	0
$7.5 > V_{in}$	5.0	0	1	1	0
$V_{in} = 7.5$	1	1	1	1	1

Sigma Delta ADC

- Best for low speed applications with high resolution
 - Less than 100KS/s and 12 to 24 bits
 - Widely used in cell phones
 - These converters have a 1-bit DAC, a comparator and some digital processing running at a very fast (oversampled) sampling rate.



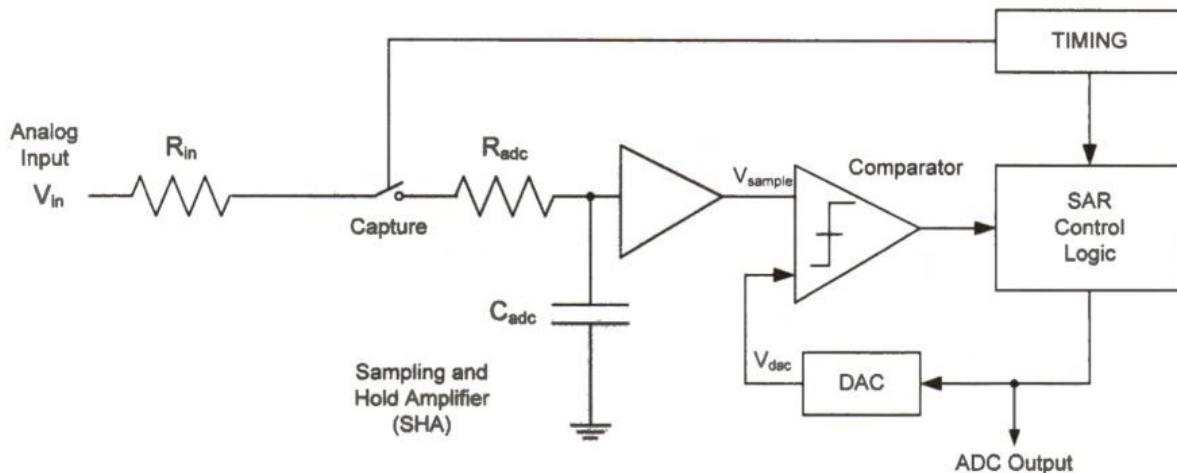
Sigma Delta ADC



Successive Approximation (SAR) ADC

- This is the ADC on the STM32
- Architecture includes a sample and hold amplifier (SHA) and a successive approximation digital quantization
- Suitable for applications with low power data acquisition and moderate sampling rates (<5MSPS)
- SAR uses a binary search algorithm in hardware
 - Find the digital number that represents the analog input must closely
 - ADC output is changed to approach the voltage at the input

Successive Approximation (SAR) ADC



Successive Approximation (SAR) ADC

- Conversion process
 - Internal DAC is set to 0.5Vref, and output is compared to Vin
 - If Vin is larger than $\frac{1}{2}$ Vref, SAR logic controller sets the most significant bit (MSB) of the ADC result (otherwise, it is cleared)
 - Next DAC is set to 0.75Vref (or $\frac{1}{4}$ Vref) depending on the comparison between Vin and Vdac
 - Process is repeated until all bits of the ADC have been determined
 - For n bits resolution, the process takes n steps to complete

Successive Approximation (SAR) ADC

- Tradeoff between resolution and sampling rate

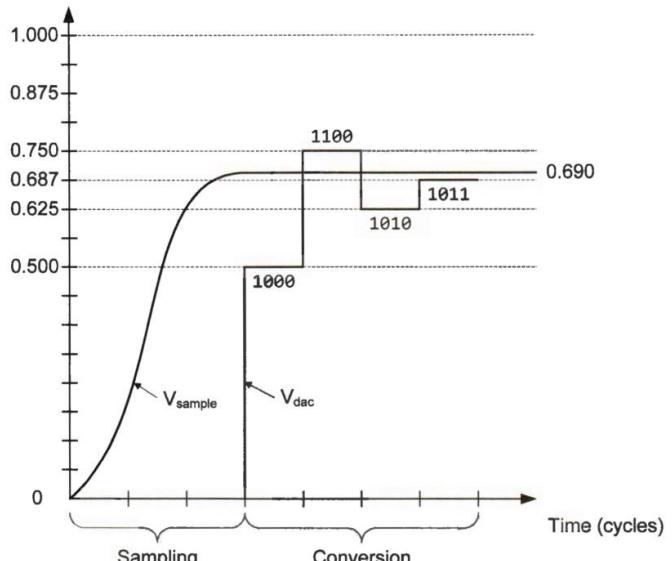
After input voltage sampled, SAR control logic sets Vdac as 0.5V (0b1000) and compare with Vsamp

Since it is larger, set Vdac to 0.75V (0b1100)

Now Vsamp < Vdac, set Vdac to 0.625

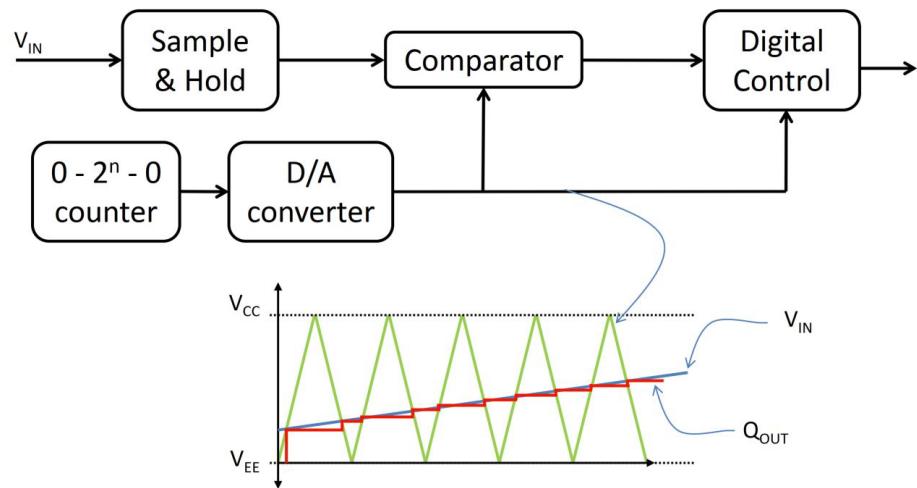
Repeat until reaching 0b1011

Conversion took 4 cycles (4 bits)

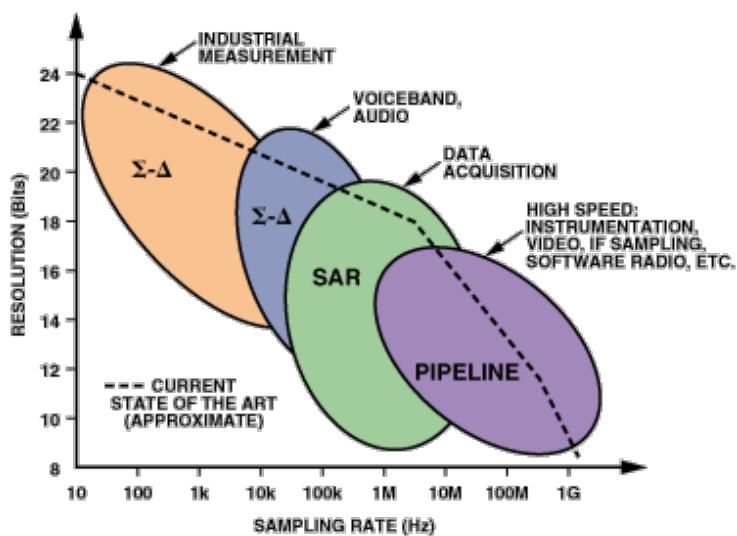


Dual Slope ADC

- Uses a triangle wave plus D/A converter to count through voltage range and find point of equality



Analog to Digital Conversion



Analog to Digital Conversion

- Internal ADCs must be set
- A reference is set through a register/memory location or an external (voltage) supply;
- Sampling rate, channel select/enable is set via a settings register/memory location
- Requires a value register/memory location
- A status register may be used for conversion status, timing, etc.

Timers

- System Timer (SysTick)
 - 24 bit down counter to produce a fixed time amount
 - Use SysTick to create time delays or generate periodic interrupts
 - Timer counts from N-1 to 0, processor generates a SysTick interrupt when counter reaches 0
 - After reaching 0, SysTick counter loads a value held in the SysTick Reload register, and counts down again
 - SysTick timer does NOT stop counting when the processor is halted
 - Useful for RTOS in creating regular time intervals according to schedule policy

Numerical Representation and Arithmetic

- In ECE241 we looked at how numbers are represented digitally
 - We also saw some basic operations
 - We represented integers, positive and negative
- In ECE342 we will review that and expand on it
 - Will look at multiplication and division
 - Will look at different numbering formats: fixed point and floating point
 - We must understand the format of the data we will operate on (from the real world)

Numerical Representation

- We represented decimal (Base 10) numbers in two formats
 - in binary (Base 2)
 - hexadecimal (Base 16)
- Initially we used unsigned integers (positional number representation)

$$B_n = b_{n-1} \quad b_{n-2} \quad \dots \quad b_1 \quad b_0$$

$$B_n = 2^{n-1} \quad 2^{n-2} \quad \dots \quad 2^1 \quad 2^0$$

Numerical Representation

- Convert unsigned integer to integer decimal

$$B_n = 2^{n-1} + 2^{n-2} + \dots + 2^1 + 2^0 \quad \text{for } n \text{ bits}$$

Ex: 1101 (4 bits)

$$1101_2 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 13_{10}$$

Numerical Representation

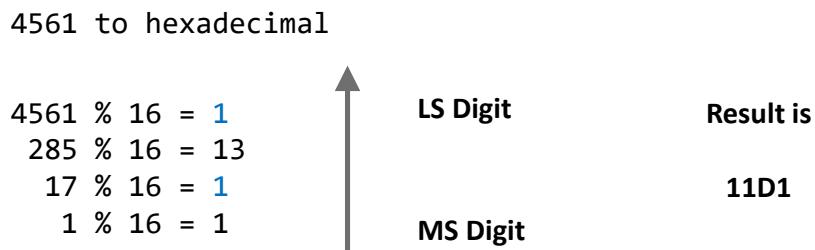
- Converting decimal integer to binary unsigned integer
 - Successive division by two, read remainders backward

4561 to binary unsigned integer

4561 % 2 =	1	↑	LSB
2280 % 2 =	0		
1140 % 2 =	0		
570 % 2 =	0		
285 % 2 =	1		
142 % 2 =	0		
71 % 2 =	1		
35 % 2 =	1		
17 % 2 =	1		
8 % 2 =	0		
4 % 2 =	0		
2 % 2 =	0		
1 % 2 =	1		MSB

Numerical Representation

- Converting decimal integer to hexadecimal
 - Successive division by SIXTEEN, read remainders backward



- Think of hexadecimal as a short form for writing binary
- Convenient when you have 16/32/64 bits registers, addresses

Numerical Representation

- Converting hexadecimal to decimal integer
 - Convert individual digits and then operate on powers of 16

3FF hexadecimal to decimal

F hex is 15 dec

$$3 \times 16^2 + 15 \times 16^1 + 15 \times 16^0$$

$$1023_{10}$$

- Please review how to convert back and forth between these formats binary, decimal, hex

Numerical Representation

- Two's complement
 - This is what is used to represent negative integers

What two's complement represents

Ex: 3 bits

000	0	100	-4	011	3
001	1	101	-3	010	2
010	2	110	-2	001	1
011	3	111	-1	000	0
				111	-1
				110	-2
				101	-3
				100	-4

- To convert, flip all bits and add one

Arithmetic

- Now that we have positive and negative numbers, we get to operate

Addition

$$\begin{array}{r} \textcolor{red}{1} \\ 05 \\ + 06 \\ \hline \textcolor{red}{11} \end{array}$$

Subtraction

$$\begin{array}{r} 08 \quad 08 \\ - 05 \quad +(-05) \\ \hline 03 \quad 03 \end{array}$$

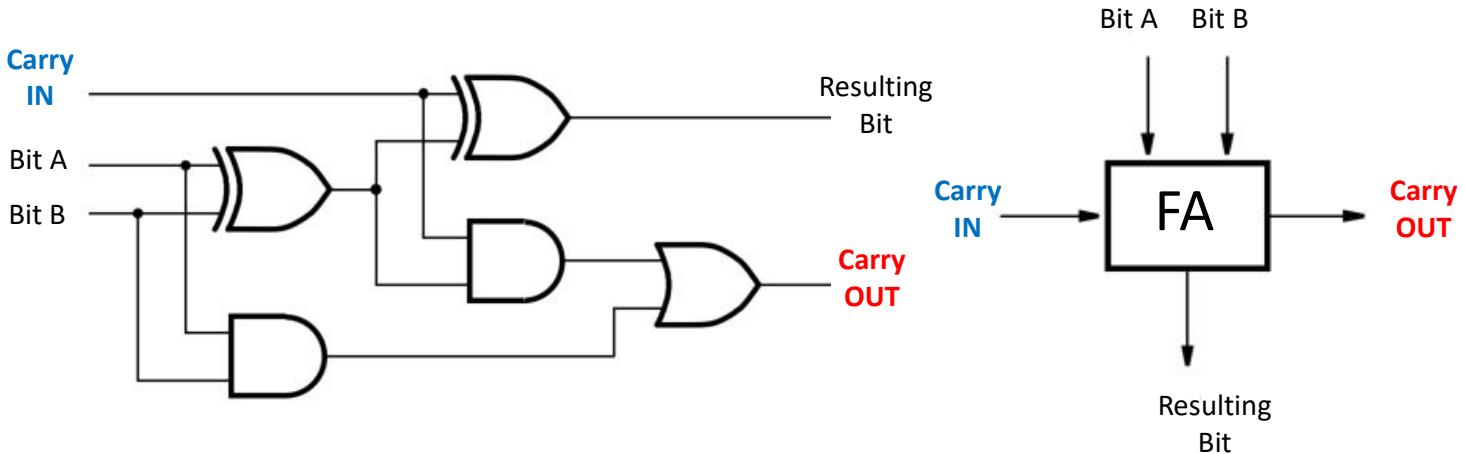
1

$$\begin{array}{r} 0101 \\ + \underline{0110} \\ 1011 \text{ (not 2's compl)} \end{array}$$

1000 positive 8, 2's compl
+ 1011 negative 5, 2's compl
1 0011 (03, carry ignored)

Arithmetic

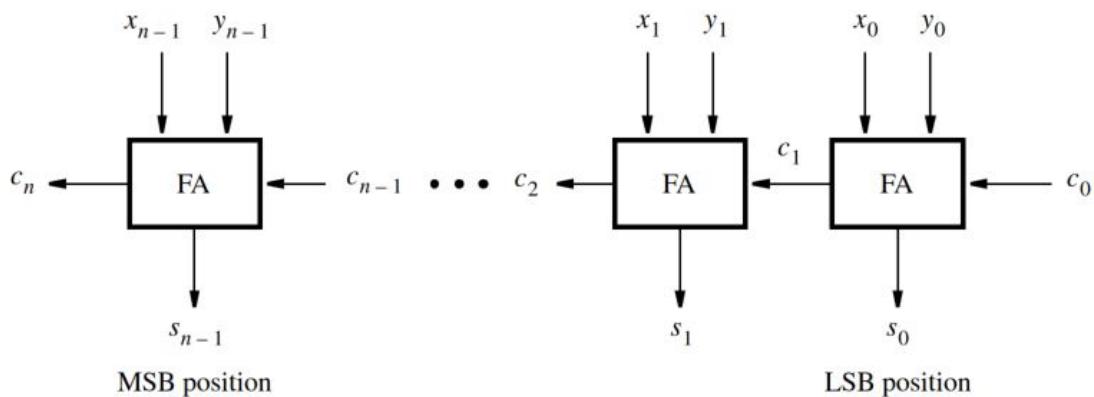
- How is this done in hardware (there are different configurations)



- This is a FULL ADDER

Arithmetic

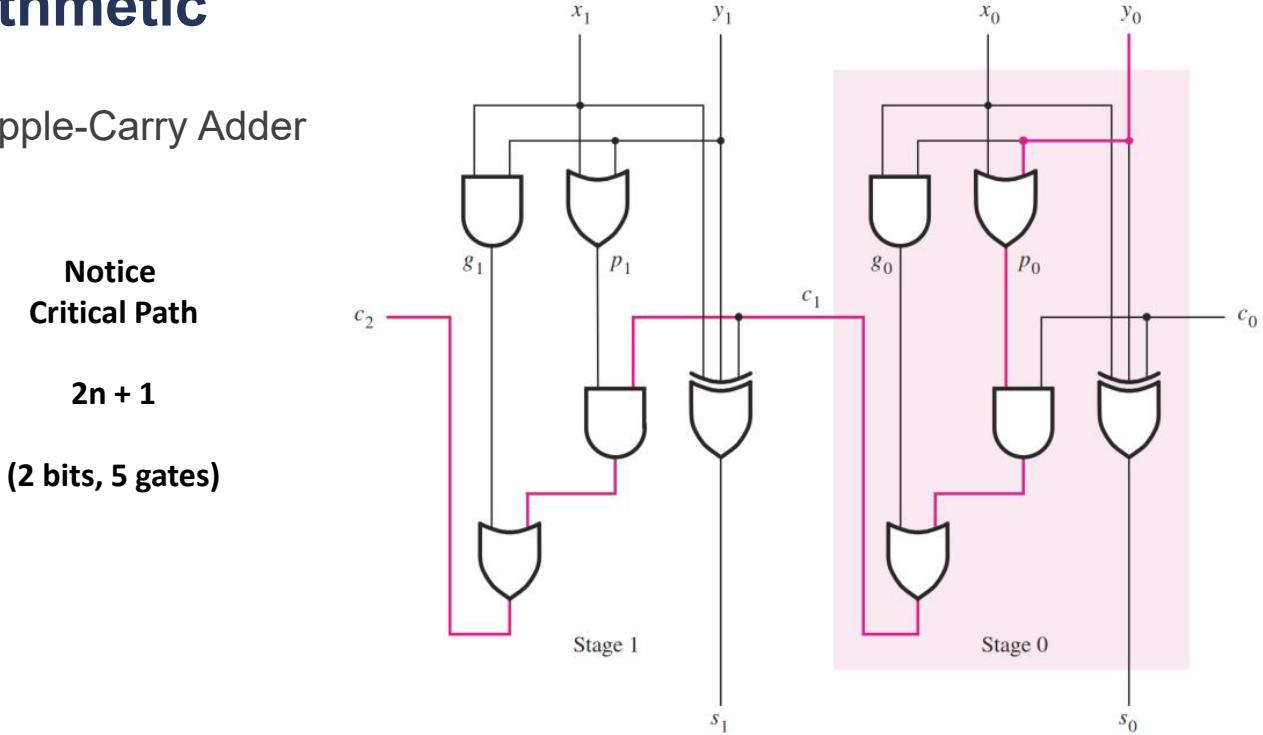
- To perform addition, we can combine these full adders in different ways



- For this topology, the carry will ripple through, bit by bit

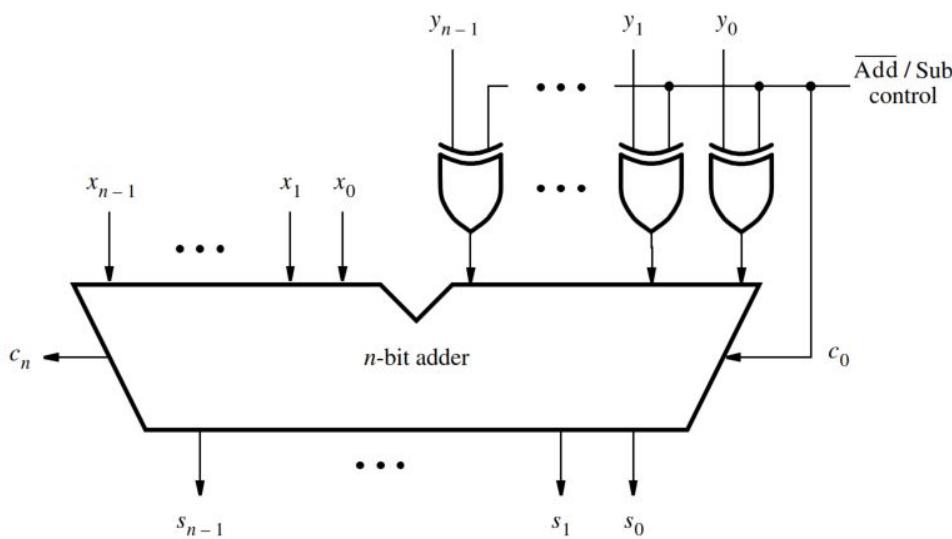
Arithmetic

- Ripple-Carry Adder



Arithmetic

- If we have signed integers...



Note:

Carry in plays the role of the “plus one” in performing subtraction as an addition to a negative number

Sub will happen when carry in is 1, and XOR operation will cause all bits to be flipped

- then c_0 , which is 1, will be the “flip plus one” to make the two’s complement

ECE342 – Computer Hardware

Lecture 06

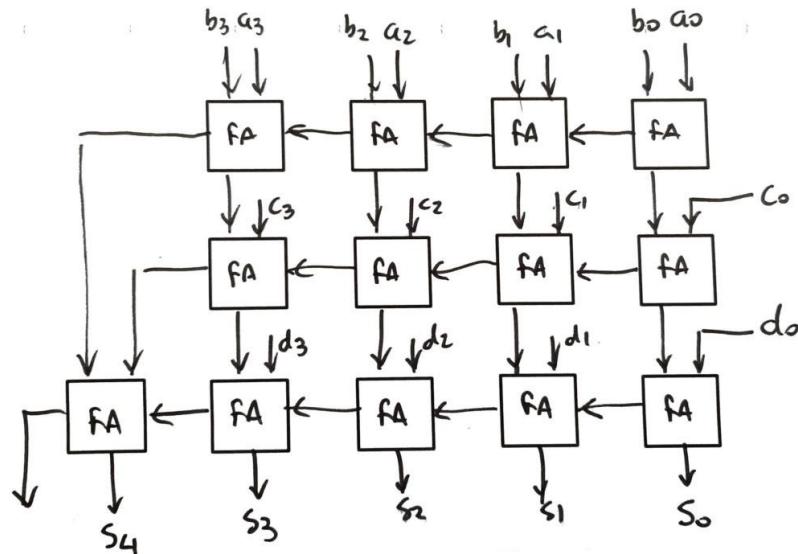
Bruno Korst, P.Eng.

Agenda

- Introduction on the board
 - Sine wave generation strategies (for lab), FIR filter structure (for project)
 - Comments on synthesizer using keypad + sine wave generation
- Arithmetic (cont'd)
 - Structures in RTL (register-transfer level) design
 - Addition, Multiplication, Division
- Non integral values

Arithmetic - Addition

- Fast Addition (cont'd) – We saw the ripple carry adder
 - four 4-bit numbers

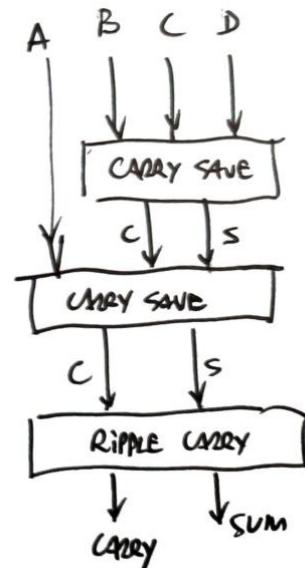
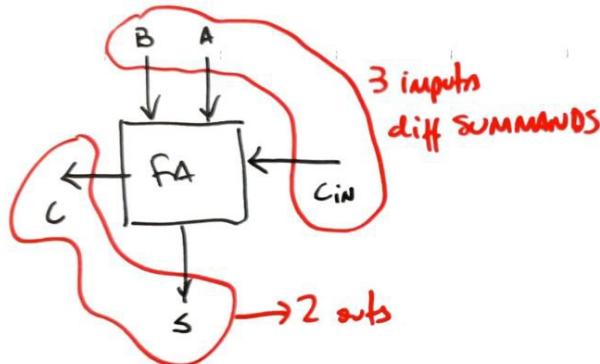


Arithmetic - Addition

- Carry Lookahead Adder (CLA) – problem with fanning
- Carry-Save Adder (CSA) – ex: four 4-bit numbers
 - Instead of letting the carry ripple through, we can feed the carry at the right place on the next stage
 - This frees up one input to each full adder at the top row
 - We can add one operand
 - On the second row, can bring in a 4th operand
 - The final row we have a “traditional” ripple-carry adder
 - This configuration allows also for faster multiplications

Arithmetic - Addition

- Carry-Save Adder



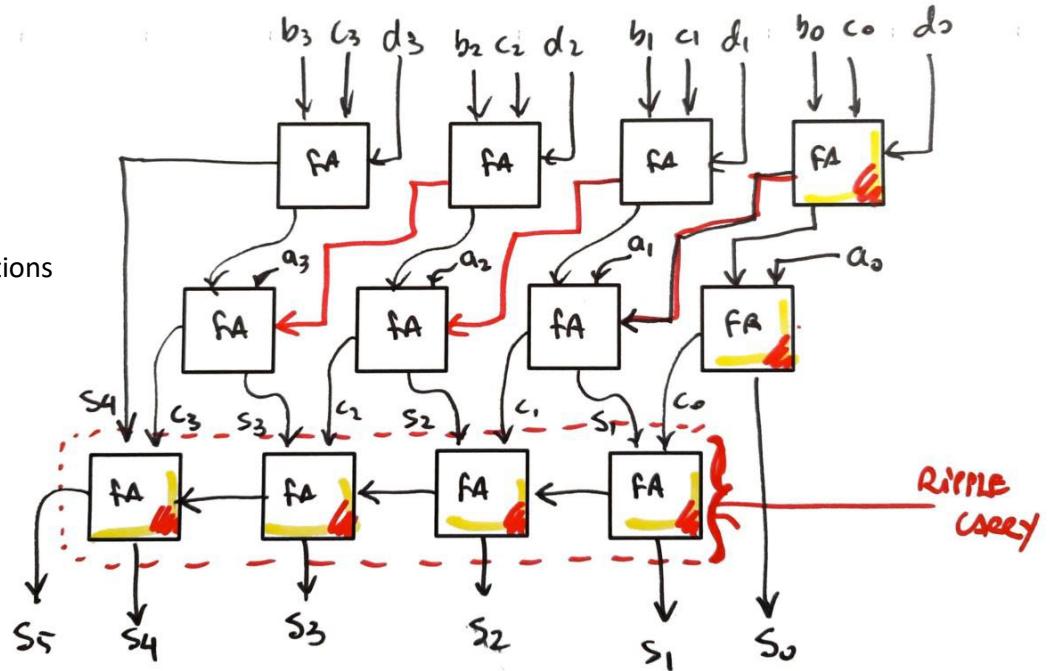
Bruno Korst, P.Eng. - Winter 2023

5

Arithmetic - Addition

- Carry-Save Adder

Faster and smaller in size
Relative to other RTL implementations



Bruno Korst, P.Eng. - Winter 2023

6

Arithmetic - Multiplication

- What happens now to multiplication? Let's try 13×11 (unsigned int)

$$\begin{array}{r}
 (13) \quad 1 \quad 1 \quad 0 \quad 1 \\
 (11) \quad \times \quad 1 \quad 0 \quad 1 \quad 1 \\
 \hline
 \end{array}$$

4 bit

4 bit

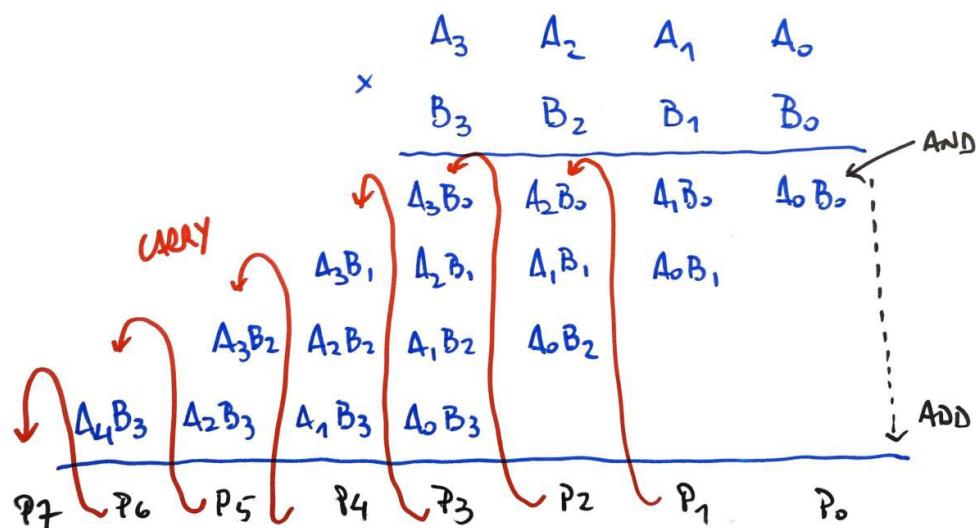
Partial Products

$\rightarrow 0 \quad 0 \quad 0 \quad 1 \quad 1 \quad 0 \quad [1]$ ← EACH ARE AND OPERATION
 $\rightarrow 0 \quad 0 \quad 1 \quad 1 \quad 0 \quad 1 \quad 0$
 $\rightarrow 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0$
 $\rightarrow 1 \quad 1 \quad 0 \quad 1 \quad 0 \quad 0 \quad 1$ ← 400

$\boxed{1 \quad 0 \quad 0 \quad 0 \quad 1 \quad 1 \quad 1 \quad 1}$ \sum PARTIAL PRODUCTS

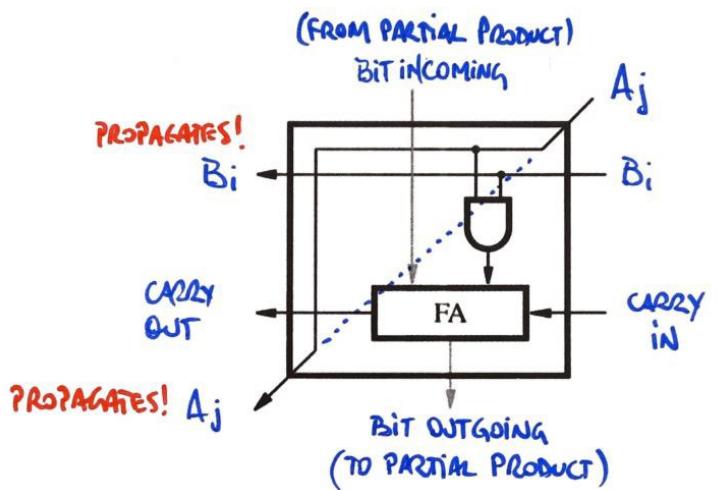
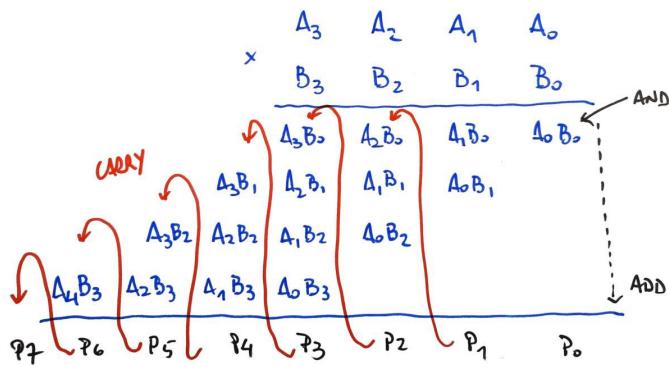
Arithmetic - Multiplication

- We can observe the structure and the operations, bit by bit



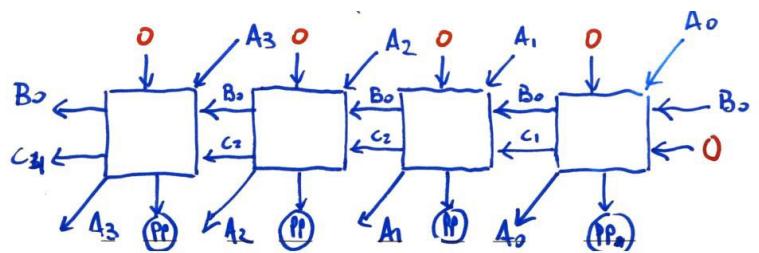
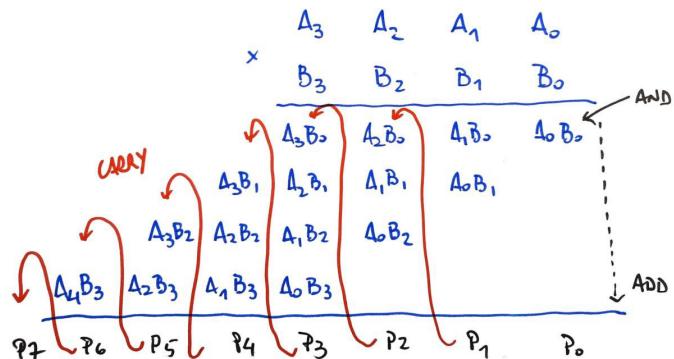
Arithmetic - Multiplication

- How is this implemented “in hardware”?

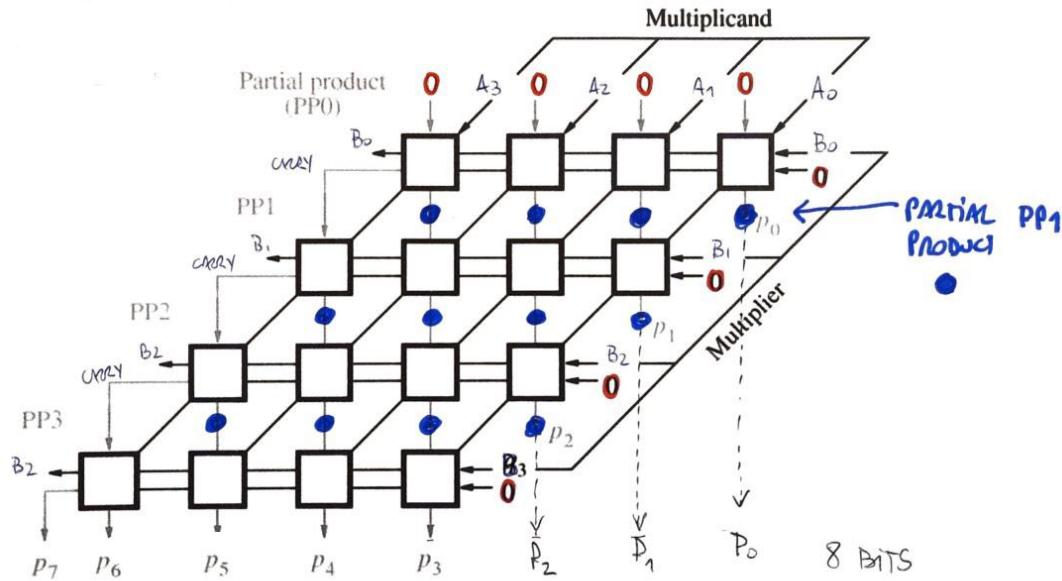


Arithmetic - Multiplication

- The first stage of such multiplier will have the configuration

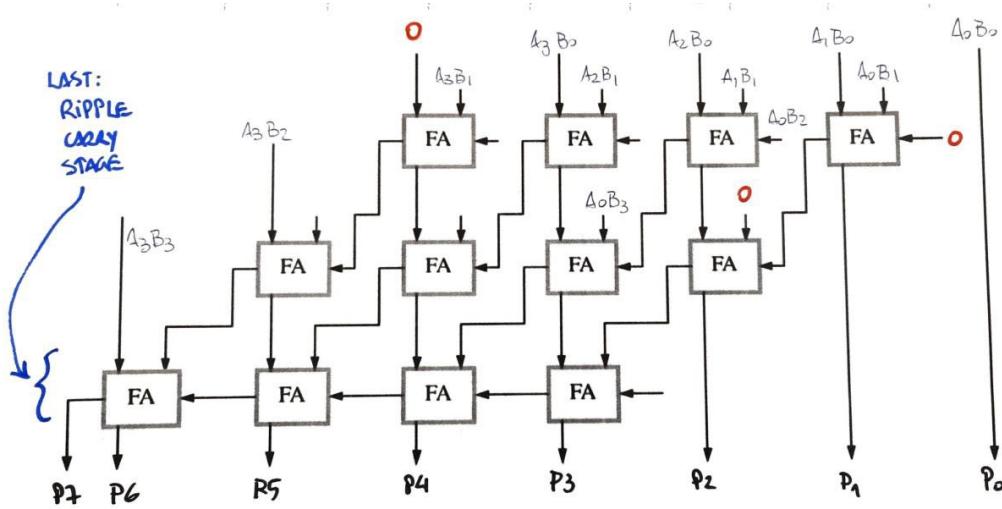


Arithmetic - Multiplication



Arithmetic - Multiplication

- This can be implemented as a fast multiplier using the Carry Save structure



Arithmetic - Division

- The idea (binary division)

$$\text{Dividend (2n bits)} / \text{Divisor (n bits)} = \text{Quotient (n bits)}, \text{Remainder (n bits)}$$

- When remainder is less than the divisor, stop
- Remainder will have the same sign as the dividend
- See this as an inverse of multiplication
 - Multiply → add and shift
 - Divide → shift and subtract (iterations depend on the result of subtraction)

Arithmetic - Division

- Example:

Handwritten binary division diagram:

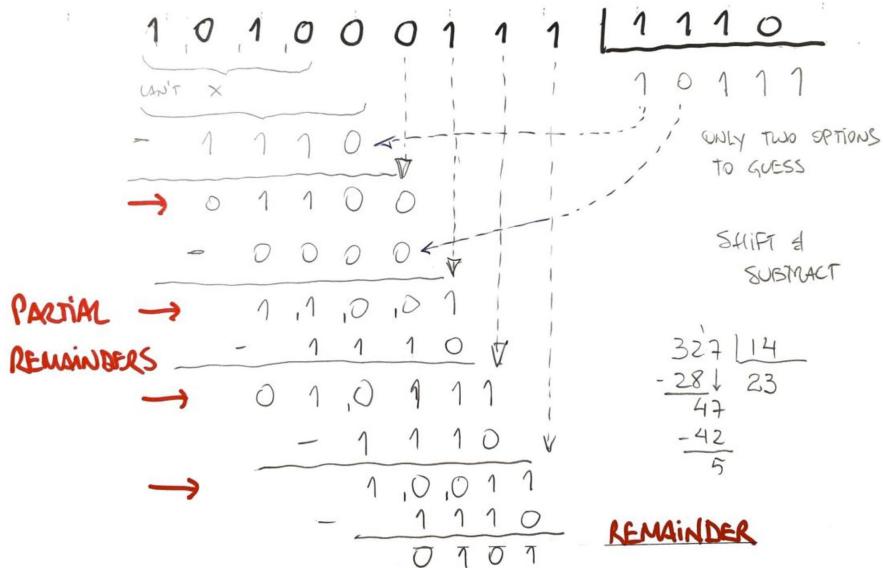
Dividend: 1 0 1 0 0 0 1 1 1 1
Divisor: 1 0 1 1 1

Quotient: 1 1 1 0

Annotations:

- Partial remainders: PARTIAL →, REMAINDERS →
- Shift & Subtract: SHIFT & SUBTRACT
- Only two options to guess: ONLY TWO OPTIONS TO GUESS
- Binary subtraction:
$$\begin{array}{r} 327 \\ - 28 \\ \hline 47 \end{array}$$
 and
$$\begin{array}{r} 47 \\ - 42 \\ \hline 5 \end{array}$$
- Remainder: REMAINDER

Arithmetic - Division



- Keep subtracting divisor from dividend at every shift
- If positive
 - Put 1 in quotient
 - Shift, continue
- If negative
 - Put 0 in quotient
 - Restore original value
 - Shift, continue

$$\begin{array}{r} 327 \mid 14 \\ -28 \downarrow 23 \\ \hline 47 \\ -42 \\ \hline 5 \end{array}$$

For signed division, if signs of dividend and divisor are different → negative quotient

Non-integral values

- Integers are nice, but we need to represent non-integral values
- Say we have an analog signal ranging from -7 to 7 Volts
 - We need to determine the acceptable precision to represent that
 - Say we choose 0.001 (10^{-3})
 - If we have

6.125 189	These will all become 6.125
6.125 237	(can we represent it in 8 bits?)
6.125 328	

Non-integral Values

- 6.125 in 8 bits

$$0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 0 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 0 \times 2^{-4}$$

6 0.125

This is 0110.0010

The binary point is FIXED, and its position depends on the architecture used

This is **FIXED POINT representation**

Non-integral Values

- “Fixed Point Format” is an approximation of the real value, since it has a finite number of bits with the binary point at a fixed position
- If the dynamic range of the application is small, fixed point is preferred
 - Ex. Pixels – Values are fixed, regular fomat
- Typically fixed point would be cheaper, better battery life, simplicity of operation, at times faster
- We will use the Q notation, with predefined integer and fractional parts

ECE342 – Computer Hardware

Lecture 07

Bruno Korst, P.Eng.

Agenda

- Non integral values – Fixed Point Representation

Non-integral values

- Integers are nice, but we need to represent non-integral values
- Say we have an analog signal ranging from -7 to 7 Volts
 - We need to determine the acceptable precision to represent that
 - Say we choose 0.001 (10^{-3})
 - If we have

6.125~~189~~ These will all become 6.125
6.125~~237~~ (can we represent it in 8 bits?)
6.125~~328~~

Non-integral Values

- 6.125 in 8 bits

$$0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 0 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 0 \times 2^{-4}$$

$$6 \qquad \qquad \qquad 0.125$$

This is 0110.0010

The binary point is FIXED, and its position depends on the architecture used

This is **FIXED POINT representation**

Fixed Point Representation

- We can generalize this idea for any base

$$+/- \quad d_0 \quad d_1 \quad \dots \quad d_{n-1} \quad \times \quad b^e \quad \text{where } d \text{ is digit number, } b \text{ is the base}$$
$$0 \leq d_i < b$$

- Digit value will be between 0 and (less than) the base
- Example

$$101.011 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3}$$

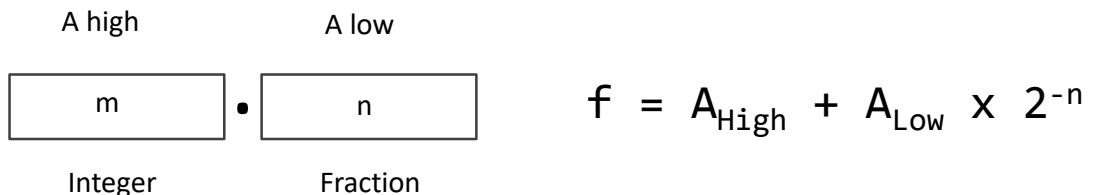
This number is an approximation of 5.375

Non-integral Values

- “Fixed Point Format” is an approximation of the real value, since it has a finite number of bits with the binary point at a fixed position
- If the dynamic range of the application is small, fixed point is preferred
 - Ex. Pixels – Values are fixed, regular fomat
- Typically fixed point would be cheaper, better battery life, simplicity of operation, at times faster
- We will use the Q notation, with predefined integer and fractional parts

Fixed Point Representation

- Fixed point representation has a predefined number of bits, with integer and fractional part
 - Q Notation → $UQ_{m,n}$
 - m = number of bits of the integer part (if m is zero, notation is Uq_n)
 - n = number of bits of the fraction part



Fixed Point Representation

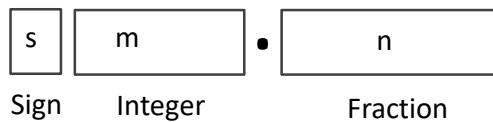
- Example: 01101.101
 - 8 bits, Format Q5.3 ($m=5$, $n=3$)
 - 1101 (integer part) → 13
 - 101 (fractional part) → 5
 - Number results in $13 + 5 \times 2^{-3} = 13.625$
 - Note that the integer in binary is (full) 01101101, which is $2^6+2^5+2^3+2^2+2^1+2^0$
 - In decimal, the number is 109

Fixed Point Representation

- The number 01101101 is decimal 109
 - Format is Q5.3
 - $n=3$
 - Take 109 and divide by 2^n , which results in 13.625
 - This means that we can look at the fixed point numbers “as integers” when we operate on them
- Signed Fixed Point
 - We use the $Qm.n$ notation, but the total number of bits is $N = m + n + 1$

Fixed Point Representation

- Signed Fixed Point
 - $Qm.n$ with total number of bits $N = m + n + 1$ (for the sign bit)
 - Often m is zero
 - Q0.7 (8 bits) → abbreviated as Q7
 - Q0.15 (16 bits) → abbreviated as Q15



Fixed Point Representation

- Say we have a binary number $b_{N-1} b_{N-2} \dots b_0$
- The two's complement (signed) integer value represented is

$$A = -1 \times b_{N-1} \times 2^{N-1} + \sum_{i=0}^{N-2} (b_i \times 2^i) \quad \text{recall that } N = m + n + 1$$

$$A = -1 \times b_{m+n} \times 2^{m+n} + \sum_{i=0}^{m+n-1} (b_i \times 2^i)$$

Fixed Point Representation

- The fixed point value of the number is

$$f = A / 2^n = -1 \times b_{(m+n)} \times 2^m + \sum_{i=0}^{m+n-1} (b_i \times 2^{(i-n)})$$

- Ex: find Q4.3 signed fixed point value represented by 10010011

That is 1 0010 011 = $(-1 \times 2^7 + 1 \times 2^4 + 1 \times 2^1 + 1 \times 2^0) / 2^3 = -109 / 8 = -13.625$

Fixed Point Representation

- Convert to Fixed Point Format
 - Multiply by 2^n ($n = \text{number of bits of the fraction}$) and round the product to the nearest integer
 - $Qm.n$ format $A = \text{round}(f \times 2^n)$

Ex: Convert $f = 3.141593$ to UQ4.12 (16 bit)

$f \times 2^{12} = 12867.964928$ --- round to 12868, resulting in 11 0010 0100 0100

In format UQ4.12 → 0011 0010 0100 0100 or 0x3244

Fixed Point Representation

- What would be the error? (previous example)

$$e = (12868 / 2^{12}) - f$$

- Exercise: convert $f = -3.141593$ to signed Q3.12
 - (hint: there will be a two's complement step)

- Fixed point resolution and range
 - Resolution = smallest non-zero real number representable
 - The “gap” between two consecutive numbers

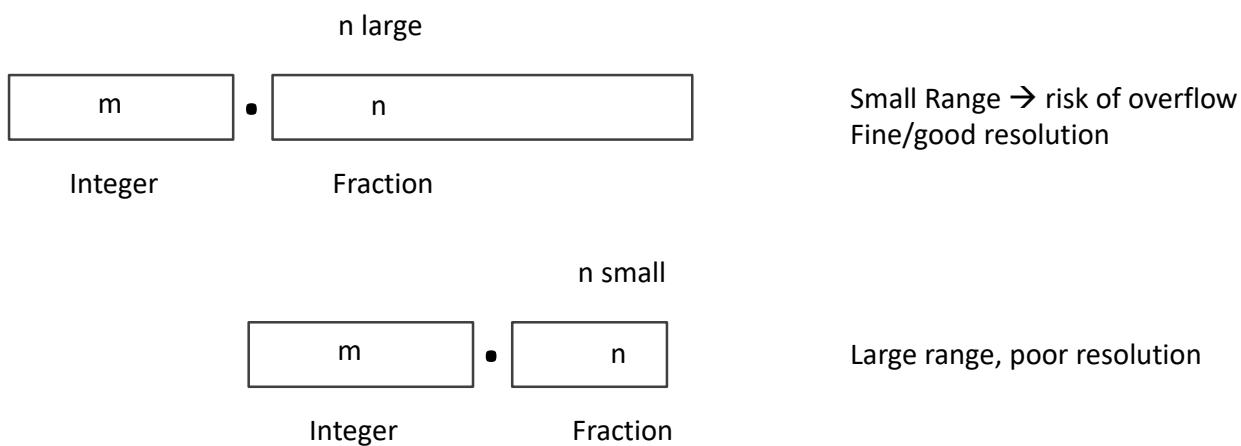
- Ex: UQm.n and Qm.n → resolution is 2^{-n} Q4.3 → 2^{-3} (0.125)

Fixed Point Representation

- Resolution differs from accuracy!
 - Accuracy refers to how close the representation is to the true value
- Range
 - Max and min values that can be represented
 - Unsigned int $\rightarrow [0, 2^{m+n} - 1]$ (factor of two difference from signed integers!)
 - Unsigned Fixed point $\rightarrow [0, 2^{m+n} - 1] \times 2^{-n}$
 - Signed fixed point $\rightarrow [-2^{m+n}, 2^{m+n} - 1] \times 2^{-n}$

Fixed Point Representation

- In fixed point format there is a trade-off between RANGE and RESOLUTION





ECE342 – Computer Hardware

Lecture 08

Bruno Korst, P.Eng.

Agenda

- Fixed Point (cont'd)

ref: Embedded Systems with ARM Cortex M Microcontrollers – Y. Zhu

Fixed Point Representation

- Q Notation

- Unsigned – UQm.n

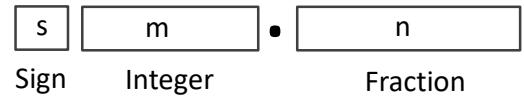


- m = #bits of integer portion, n = #bits of fractional portion

- Signed – Qm.n

- # bits = N = n + m + 1

- Q0.7 or Q7 (8 bits), Q0.15 or Q15 (16 bits)



- Accuracy: how close the number is to the real value
- Resolution: smallest non-zero number that can be represented
- Range: max and min values that can be represented

Fixed Point Representation

- Note: we choose how to store a number based on the needed resolution
 - If my 8-bit ADC has a range of 0 to +5V, its resolution is $5/256 = 0.02V$
 - Store voltages as 16 bit unsigned fixed point with resolution of 0.01V (finer) 9 bit
- What would be the accuracy error (difference between the real value and the number represented)?

$$e = (\text{Signed Integer Rounded Value} / 2^{\text{Fractional Bits}}) - \text{actual number}$$

- Exercise: convert $f = -3.141593$ to signed Q3.12
 - (hint: there will be a two's complement step)

Fixed Point Representation

- What would be the accuracy error (difference between the real value and the number represented)?

$$e = (12868 / 2^{12}) - f$$

- Ex: convert $f = -3.141593$ to signed Q3.**12**
 - Calculate $f \times 2^{12} \rightarrow -12867.964928$
 - Round the result to nearest integer $\rightarrow -12868$
 - Convert the absolute value to binary $\rightarrow 11\ 0010\ 0100\ 0100$
 - We're using 16 bits, so make it 16 bits $\rightarrow 0011\ 0010\ 0100\ 0100$
 - The number is negative, so find 2's complement (16b) $\rightarrow 1100\ 1101\ 1011\ 1100$
 - Error: $-12868 / 2^{12} - f$

Fixed Point Representation

- One common format is Q15 (1 bit for sign, 15 bits for magnitude)



- This number is **normalized** to the fractional range from -1 to +1
 - Normalizing solves the problem of overflow in multiplication
- The range is divided into 2^{16} intervals, each of size 2^{-15}
- Most negative number is 1, and most positive is $1-2^{-15}$
- Any result from multiplication is within the range -1 to +1

Fixed Point Representation

- Q15 representation of 0.560123
 - (alternative conversion)
 - Multiply by 2 successively, bit per bit
 - If the result of the multiplication is >1
 - carry bit one as a MSB
 - copy the fractional part to the next mult. by 2
 - if the product is <1, carry 0 to MSB
 - Result in Q15 → 0.10001110110010

Number	Product	Carry	Bit
0.560123	1.120246	MSB 1	0
0.120246	0.240492	0	1
0.240492	0.480984	0	2
0.480984	0.961968	0	3
0.961968	1.923936	1	4
0.923936	1.847872	1	5
0.847872	1.695744	1	6
0.695744	1.391488	1	7
0.391488	0.782976	0	8
0.782976	1.565952	1	9
0.565952	1.131904	1	10
0.131904	0.263808	0	11
0.263808	0.527616	0	12
0.527616	1.055232	1	13
0.055232	0.110464	LSB 0	14

Fixed Point Representation

- Q15 representation of 0.560123
 - Result in Q15 → 0.10001110110010
 - Note: we only have 16 bits (could keep going)
 - Lose accuracy after conversion
 - Error is introduced
 - This error should be less than the interval size, which is $2^{-15} = 0.0000305017$

Number	Product	Carry	Bit
0.560123	1.120246	MSB 1	0
0.120246	0.240492	0	1
0.240492	0.480984	0	2
0.480984	0.961968	0	3
0.961968	1.923936	1	4
0.923936	1.847872	1	5
0.847872	1.695744	1	6
0.695744	1.391488	1	7
0.391488	0.782976	0	8
0.782976	1.565952	1	9
0.565952	1.131904	1	10
0.131904	0.263808	0	11
0.263808	0.527616	0	12
0.527616	1.055232	1	13
0.055232	0.110464	LSB 0	14

Fixed Point Representation

- Let us convert back the Q15 number 0.100 0111 1011 0010

$$2^{-1} + 2^{-5} + 2^{-6} + 2^{-7} + 2^{-8} + 2^{-10} + 2^{-11} + 2^{-14} = 0.560120$$

The original number was 0.560123, which gives an error of 0.000003

We calculated the error to be less than $2^{-15} = 0.000030517$

(note: the larger the number of bits, the smaller the error)

Fixed Point Representation

- What is the Q15 representation of -0.160123?
 - First find Q15 representation of the positive number 0.160123 (as previous)

Q15 → 0.001010001111110

Apply 2's complement → 1.110101110000010

Alternatively, do $-0.160123 \times 2^{15} = -5246.9$ and convert the rounded -5247 to 16 bit, 2's complement → 1110101110000001 (error introduced by rounding up)

Fixed Point Representation

- Convert Q15 signed 1.110 1011 1000 0010
- Number is negative, start by applying 2's complement

Flip	0001 0100 0111 1101
Plus 1	0001 0100 0111 1110

We know it's negative

The decimal is $\rightarrow - (2^{-3} + 2^{-5} + 2^{-9} + 2^{-10} + 2^{-11} + 2^{-12} + 2^{-13} + 2^{-14})$

Resulting in - 0.160095

Fixed Point Representation

- Addition of Fixed Point numbers

$$\begin{array}{r} 1.110 \ 1011 \ 1000 \ 0010 \\ + 0.100 \ 0111 \ 1011 \ 0010 \\ \hline 10.011 \ 0011 \ 0011 \ 0100 \end{array} \quad \begin{array}{l} (-0.160095) \\ (+0.560120) \\ \text{Format is Q15!} \end{array}$$

Extended sign bit is dropped

Result is 0.011 0011 0011 0100 \rightarrow 0.400024

Fixed Point Representation

- Multiplication: 0.25 and 0.5 in Q3 format

$$\begin{array}{r} & \quad 0.010 \\ \times & \quad 0.100 \\ \hline & \quad 0000 \\ & \quad 0000 \\ & \quad 0010 \\ & \quad 0000 \\ 0.001000 & \quad \leftarrow \text{This is Q6 format} \end{array}$$

- Format is Q3, so result is 0.001 (truncate) $\rightarrow 1 \times 2^{-3} = 0.125$

Fixed Point Representation

- Format precision loss
 - Two Q3.12 numbers multiplied, available memory is 16 bit wide

$$\begin{array}{rcl} \text{Q3.12} \longrightarrow 7.5 & & 0111.1000\ 0000\ 0000 \\ \text{Q3.12} \longrightarrow 7.25 & * & 0111.0100\ 0000\ 0000 \\ \text{Q6.24} \longrightarrow 54.375 & \hline & 0011\ 0110.0110\ 0000\ 0000\ ... \\ & & \boxed{\qquad\qquad\qquad\qquad\qquad\qquad} \\ & & \text{Q3.12} \longrightarrow 6.375 \\ & & \boxed{\qquad\qquad\qquad\qquad\qquad\qquad} \\ & & \text{Q6.9} \longrightarrow 54.375 \end{array}$$

- Storing in Q3.12 format will lead to the wrong result
- A lower precision format will allow for the correct value stored

Fixed Point Representation

- Lessons
 - When converting decimal to Q_n (n = fractional or magnitude bits), we may lose accuracy due to rounding error
 - Error is bounded by the size of the interval 2^{-n}
 - Adding and subtracting fixed-point numbers can be done as if integers
 - Addition and subtraction may cause overflow – flipping of sign bit
 - Adding to positives leading to a negative, negatives lead to positive, etc.
 - Multiplication of two Q_n numbers results in Q_{2n}
 - It is typical that register holding result is double sized.

Fixed Point Representation

- Lessons
 - Use bit shift operations to correct sign and size.
 - Shift left by one for extended sign
 - Shift right by 16 to take the 16 upper MSB of a Q31 into a Q15 format
 - Underflow results when result of multiplication is too small to be represented
 - Ex: multiplying two Q2 numbers (0.01×0.01) will result in 0.0001. This will be truncated into Q2, leading to 0.00 (underflow)

Fixed Point Representation

- Example:

```
for ( j = 0; j < BUFFER_SIZE ; j++) {  
    for ( i = N-1; i > 0; i --)  
        samples[i] = samples[i-1];  
  
    samples[0] = input[j];  
  
    result = 0;  
  
    for (i = 0; i < N; i++)  
        result += (samples[i] * coeff[i]) << 1;  
  
    result = result >> 16;  
  
    output[j] = (short) (result << 1);  
}
```

Fixed Point Representation

- Lessons
 - Overflow is when the result of a calculation exceeds the range of the format used
 - One solution is to increase precision of the inputs, operate and reduce precision again

Ex: calculate $M = (53 * N)/100$ with M and N integers.

- If 8 bit unsigned, can overflow.
- solution: make all operands 16 bit unsigned (53, N and 100), calculate, if the result is within 8 bit range, convert back

Fixed Point Representation

- Lessons
 - One may need to scale/offset pre and post operation
 - Make the number “fit” or position the data within range of bits available

$$D_{\text{new}} = \underline{D \times 2^{\text{scale}}} + \text{offset}$$

Here I shift

Here I calibrate

Fixed Point Representation

- Lessons
 - When adding/subtracting fixed point numbers, convert the numbers to the expected format of the result first, then operate
 - Attention on the order of operations when performing multiple integer operations
 - Because division can be problematic – with intermediate results too small – always divide last
 - Adjust if needed

Fixed Point Representation

- Multiplication of two 32-bit fixed point numbers (Q15.16 or UQ16.16)

$$fc = fa \times fb \quad \text{where } fn = I_n \times 2^{-16} \quad (\text{convert to integer})$$

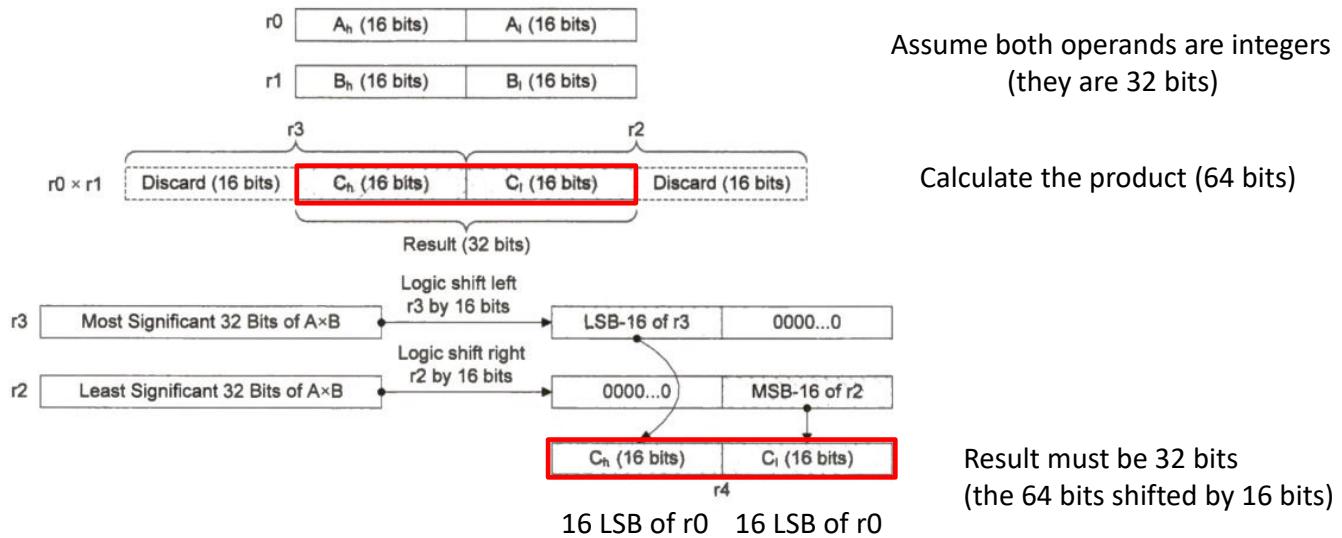
$$I_c \times 2^{-16} = (I_a \times 2^{-16}) \times (I_b \times 2^{-16})$$

$$\text{So... } I_c = (I_a \times I_b) \times 2^{-16}$$

Note that the result has a shift RIGHT by 16 bits

Fixed Point Representation

- Multiplication



Fixed Point Representation

- Division (ARM does not have instructions to perform 64 bit integer division)

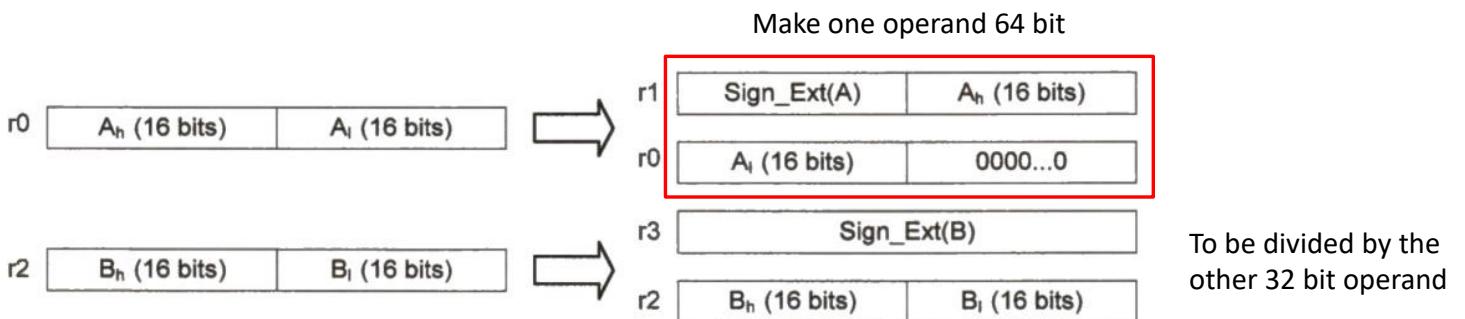
$$fc = fa / fb \quad \text{where } fn = In \times 2^{-16} \text{ (convert to integer)}$$

$$Ic \times 2^{-16} = (Ia \times 2^{-16}) / (Ib \times 2^{-16})$$

$$\text{So... } Ic = (Ia / Ib) \times 2^{+16}$$

Note that the result has a shift LEFT by 16 bits

Fixed Point Representation



Result will be a 32 bit quotient

ECE342 – Computer Hardware

Lecture 08

Bruno Korst, P.Eng.

Agenda

- Fixed Point (cont'd)
- Floating Point format

ref: Embedded Systems with ARM Cortex M Microcontrollers – Y. Zhu

Fixed Point Representation

- Q Notation

- Unsigned – UQm.n

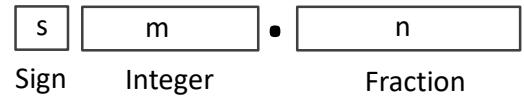


- m = #bits of integer portion, n = #bits of fractional portion

- Signed – Qm.n

- # bits = N = n + m + 1

- Q0.7 or Q7 (8 bits), Q0.15 or Q15 (16 bits)



- Accuracy: how close the number is to the real value
- Resolution: smallest non-zero number that can be represented
- Range: max and min values that can be represented

Fixed Point Representation

- Note: we choose how to store a number based on the needed resolution
 - If my 8-bit ADC has a range of 0 to +5V, its resolution is $5/256 = 0.02V$
 - Store voltages as 16 bit unsigned fixed point with resolution of 0.01V (finer) 9 bit
- What would be the accuracy error (difference between the real value and the number represented)?

$$e = (\text{Signed Integer Rounded Value} / 2^{\text{Fractional Bits}}) - \text{actual number}$$

- Exercise: convert $f = -3.141593$ to signed Q3.12
 - (hint: there will be a two's complement step)

Fixed Point Representation

- What would be the accuracy error (difference between the real value and the number represented)?

$$e = (12868 / 2^{12}) - f$$

- Ex: convert $f = -3.141593$ to signed Q3.**12**
 - Calculate $f \times 2^{12} \rightarrow -12867.964928$
 - Round the result to nearest integer $\rightarrow -12868$
 - Convert the absolute value to binary $\rightarrow 11\ 0010\ 0100\ 0100$
 - We're using 16 bits, so make it 16 bits $\rightarrow 0011\ 0010\ 0100\ 0100$
 - The number is negative, so find 2's complement (16b) $\rightarrow 1100\ 1101\ 1011\ 1100$
 - Error: $-12868 / 2^{12} - f$

Fixed Point Representation

- One common format is Q15 (1 bit for sign, 15 bits for magnitude)



- This number is **normalized** to the fractional range from -1 to +1
 - Normalizing solves the problem of overflow in multiplication
- The range is divided into 2^{16} intervals, each of size 2^{-15}
- Most negative number is 1, and most positive is $1-2^{-15}$
- Any result from multiplication is within the range -1 to +1

Fixed Point Representation

- Q15 representation of 0.560123
 - (alternative conversion)
 - Multiply by 2 successively, bit per bit
 - If the result of the multiplication is >1
 - carry bit one as a MSB
 - copy the fractional part to the next mult. by 2
 - if the product is <1, carry 0 to MSB
 - Result in Q15 → 0.10001110110010

Number	Product	Carry	Bit
0.560123	1.120246	MSB 1	0
0.120246	0.240492	0	1
0.240492	0.480984	0	2
0.480984	0.961968	0	3
0.961968	1.923936	1	4
0.923936	1.847872	1	5
0.847872	1.695744	1	6
0.695744	1.391488	1	7
0.391488	0.782976	0	8
0.782976	1.565952	1	9
0.565952	1.131904	1	10
0.131904	0.263808	0	11
0.263808	0.527616	0	12
0.527616	1.055232	1	13
0.055232	0.110464	LSB 0	14

Fixed Point Representation

- Q15 representation of 0.560123
 - Result in Q15 → 0.10001110110010
 - Note: we only have 16 bits (could keep going)
 - Lose accuracy after conversion
 - Truncation error is introduced
 - This error should be less than the interval size, which is $2^{-15} = 0.0000305017$

Number	Product	Carry	Bit
0.560123	1.120246	MSB 1	0
0.120246	0.240492	0	1
0.240492	0.480984	0	2
0.480984	0.961968	0	3
0.961968	1.923936	1	4
0.923936	1.847872	1	5
0.847872	1.695744	1	6
0.695744	1.391488	1	7
0.391488	0.782976	0	8
0.782976	1.565952	1	9
0.565952	1.131904	1	10
0.131904	0.263808	0	11
0.263808	0.527616	0	12
0.527616	1.055232	1	13
0.055232	0.110464	LSB 0	14

Fixed Point Representation

- Let us convert back the Q15 number 0.100 0111 1011 0010

$$2^{-1} + 2^{-5} + 2^{-6} + 2^{-7} + 2^{-8} + 2^{-10} + 2^{-11} + 2^{-14} = 0.560120$$

The original number was 0.560123, which gives an error of 0.000003

We calculated the truncation error to be less than $2^{-15} = 0.000030517$

(note: the larger the number of bits, the smaller the truncation error)

Fixed Point Representation

- What is the Q15 representation of -0.160123?

- First find Q15 representation of the positive number 0.160123 (as previous)

Q15 → 0.001010001111110

Apply 2's complement → 1.110101110000010

Alternatively, do $-0.160123 \times 2^{15} = -5246.9$ and convert the truncated -5246 to 16 bit, 2's complement → 1110101110000010

Fixed Point Representation

- Convert Q15 signed 1.110 1011 1000 0010
- Number is negative, start by applying 2's complement

Flip	0001 0100 0111 1101
Plus 1	0001 0100 0111 1110

We know it's negative

The decimal is $\rightarrow - (2^{-3} + 2^{-5} + 2^{-9} + 2^{-10} + 2^{-11} + 2^{-12} + 2^{-13} + 2^{-14})$

Resulting in - 0.160095

Fixed Point Representation

- Addition of Fixed Point numbers

$$\begin{array}{r} 1.110 \ 1011 \ 1000 \ 0010 \\ + 0.100 \ 0111 \ 1011 \ 0010 \\ \hline 10.011 \ 0011 \ 0011 \ 0100 \end{array} \quad \begin{array}{l} (-0.160095) \\ (+0.560120) \\ \text{Format is Q15!} \end{array}$$

Extended sign bit is dropped

Result is 0.011 0011 0011 0100 \rightarrow 0.400024

Fixed Point Representation

- Multiplication: 0.25 and 0.5 in Q3 format

$$\begin{array}{r} & \quad 0.010 \\ \times & \quad 0.100 \\ \hline & \quad 0000 \\ & \quad 0000 \\ & \quad 0010 \\ & \quad 0000 \\ 0.001000 & \quad \leftarrow \text{This is Q6 format} \end{array}$$

- Format is Q3, so result is 0.001 (truncate) $\rightarrow 1 \times 2^{-3} = 0.125$

Fixed Point Representation

- Format precision loss
 - Two Q3.12 numbers multiplied, available memory is 16 bit wide

$$\begin{array}{rcl} \text{Q3.12} \longrightarrow 7.5 & & 0111.1000\ 0000\ 0000 \\ \text{Q3.12} \longrightarrow 7.25 & * & 0111.0100\ 0000\ 0000 \\ \text{Q6.24} \longrightarrow 54.375 & \hline & 0011\ 0110.0110\ 0000\ 0000\ ... \\ & & \boxed{\qquad\qquad\qquad\qquad\qquad\qquad} \\ & & \text{Q3.12} \longrightarrow 6.375 \\ & & \boxed{\qquad\qquad\qquad\qquad\qquad\qquad} \\ & & \text{Q6.9} \longrightarrow 54.375 \end{array}$$

- Storing in Q3.12 format will lead to the wrong result
- A lower precision format will allow for the correct value stored

Fixed Point Representation

- Lessons
 - When converting decimal to Q_n (n = fractional or magnitude bits), we may lose accuracy due to truncation error
 - Error is bounded by the size of the interval 2^{-n}
 - Adding and subtracting fixed-point numbers can be done as if integers
 - Addition and subtraction may cause overflow – flipping of sign bit
 - Adding to positives leading to a negative, negatives lead to positive, etc.
 - Multiplication of two Q_n numbers results in Q_{2n}
 - It is typical that register holding result is double sized.

Fixed Point Representation

- Lessons
 - Use bit shift operations to correct sign and size.
 - Shift left by one for extended sign
 - Shift right by 16 to take the 16 upper MSB of a Q31 into a Q15 format
 - Underflow results when result of multiplication is too small to be represented
 - Ex: multiplying two Q2 numbers (0.01×0.01) will result in 0.0001. This will be truncated into Q2, leading to 0.00 (underflow)

Fixed Point Representation

- Example:

```
for ( j = 0; j < BUFFER_SIZE ; j++) {  
    for ( i = N-1; i > 0; i --)  
        samples[i] = samples[i-1];  
  
    samples[0] = input[j];  
  
    result = 0;  
  
    for (i = 0; i < N; i++)  
        result += (samples[i] * coeff[i]) << 1;  
  
    result = result >> 16;  
  
    output[j] = (short) (result << 1);  
}
```

Fixed Point Representation

- Lessons
 - Overflow is when the result of a calculation exceeds the range of the format used
 - One solution is to increase precision of the inputs, operate and reduce precision again

Ex: calculate $M = (53 * N)/100$ with M and N integers.

- If 8 bit unsigned, can overflow.
- solution: make all operands 16 bit unsigned (53, N and 100), calculate, if the result is within 8 bit range, convert back

Fixed Point Representation

- Lessons
 - One may need to scale/offset pre and post operation
 - Make the number “fit” or position the data within range of bits available

$$D_{\text{new}} = \underline{D \times 2^{\text{scale}}} + \text{offset}$$

Here I shift

Here I calibrate

Fixed Point Representation

- Lessons
 - When adding/subtracting fixed point numbers, convert the numbers to the expected format of the result first, then operate
 - Attention on the order of operations when performing multiple integer operations
 - Because division can be problematic – with intermediate results too small – always divide last
 - Adjust if needed

Fixed Point Representation

- Multiplication of two 32-bit fixed point numbers (Q15.16 or UQ16.16)

$$fc = fa \times fb \quad \text{where } fn = I_n \times 2^{-16} \quad (\text{convert to integer})$$

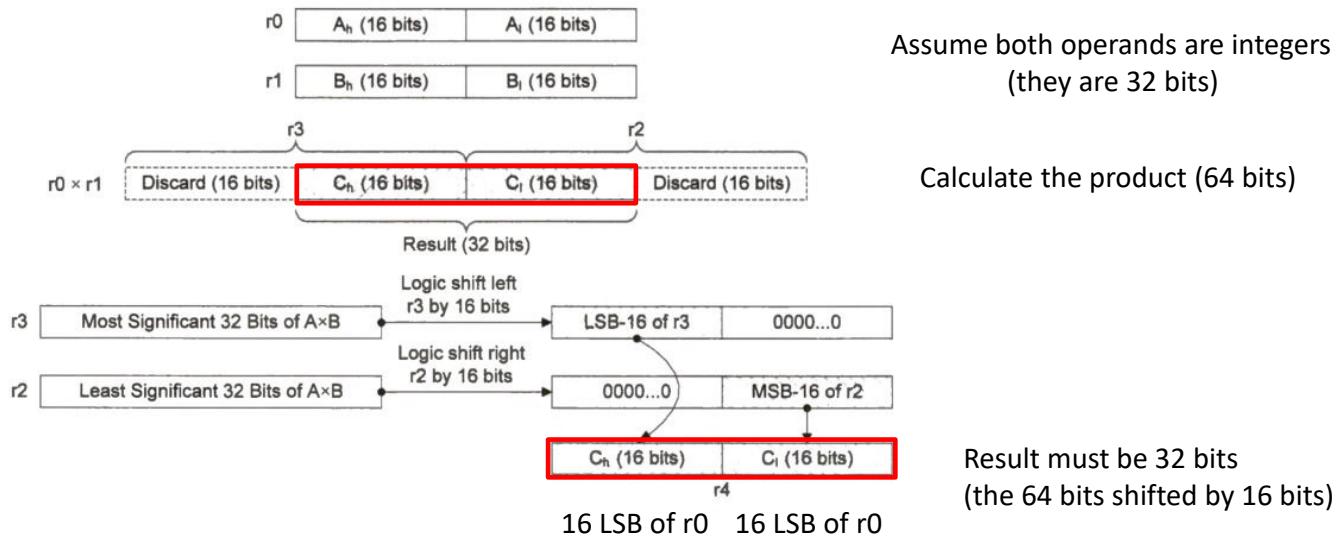
$$I_c \times 2^{-16} = (I_a \times 2^{-16}) \times (I_b \times 2^{-16})$$

$$\text{So... } I_c = (I_a \times I_b) \times 2^{-16}$$

Note that the result has a shift RIGHT by 16 bits

Fixed Point Representation

- Multiplication



Fixed Point Representation

- Division (ARM does not have instructions to perform 64 bit integer division)

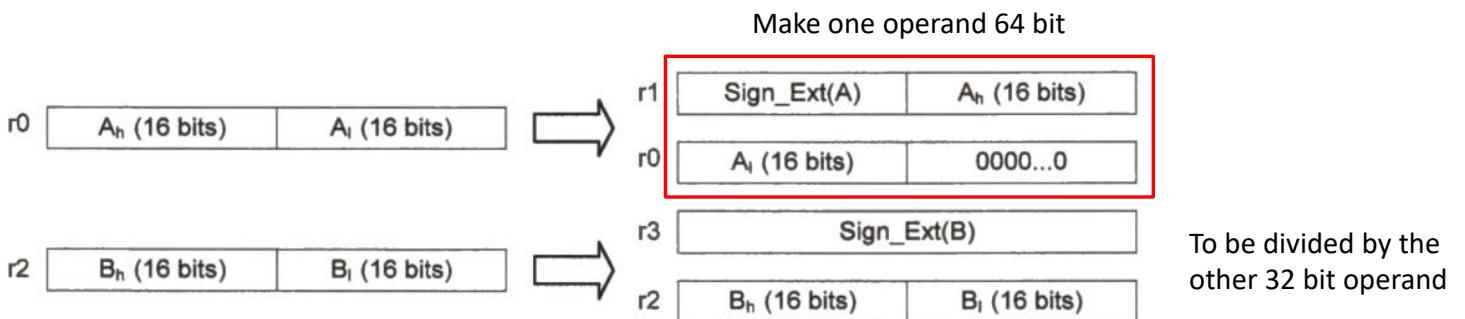
$$fc = fa / fb \quad \text{where } fn = In \times 2^{-16} \quad (\text{convert to integer})$$

$$Ic \times 2^{-16} = (Ia \times 2^{-16}) / (Ib \times 2^{-16})$$

$$\text{So... } Ic = (Ia / Ib) \times 2^{+16}$$

Note that the result has a shift LEFT by 16 bits

Fixed Point Representation



Result will be a 32 bit quotient

ECE342 – Computer Hardware

Lecture 09/10

Bruno Korst, P.Eng.

Agenda

- 1st half – Quiz 1
- Intro to floating point

ref: Embedded Systems with ARM Cortex M Microcontrollers – Y. Zhu

Floating Point Representation

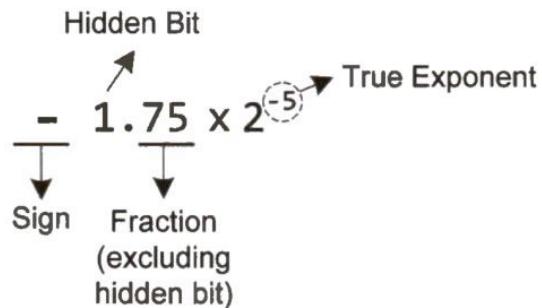
- Real numbers are operated on using Floating Point arithmetic
- Currently, the widely adopted standard is IEEE754
 - Determines format for storage and operation on real numbers
- Many, if not all processors (FPUs) and software libraries support it.

Floating Point Representation

- Representation
 - Uses a normalized notation with **3 bit fields**
 - Sign
 - Fraction – “Mantissa” or fractional part
 - Exponent – a binary field holding a “corrected” or “biased” exponent
 - The “normalized notation” means that there is only one digit before (to the left of) the decimal point, whose value is ONE
 - This is called the “hidden one” or “implicit one”

Floating Point Representation

- Notation of IEEE 754



- Resembles scientific notation, but it is expressed with **power of two**
- Before conversion to binary, the integer part of the number must be **ONE**
 - “normalized”

Floating Point Representation

- Examples:
 - 10.746×2^6 is not normalized (integer part is not one)
 - 1.025×10^3 is not normalized (power of 10)
- We seek to express the number **prior to binary conversion** in the form:



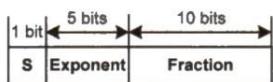
Floating Point Representation

- Let us attempt to convert 10.746×2^6
 - Try to make the integer part equal 1 $\rightarrow (10.746/8) \times 8 \times 2^6$
 - For this case, divide by 8, multiply by 8, adjust the exponent accordingly
 - The converted number is 1.34325×2^9
- Most commonly used formats are single precision and double precision
 - Respectively “float” and “double” in C

Floating Point Representation

- Field bits for floating point data types

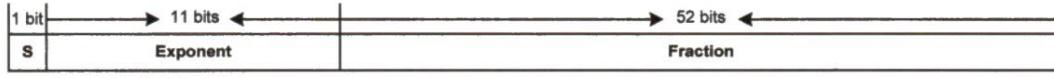
Half Precision (16 bits)



Single Precision (32 bits)



Double Precision (64 bits)



- “Exponent” is a corrected exponent

Floating Point Representation

- Why not just use double?
 - Maybe the hardware does not support that format
 - Maybe single precision will do the job just fine and cost less
- Bit Fields
 - Sign – 0 for positive, 1 for negative
 - Biased Exponent – it's the TRUE exponent of the power of two, corrected to represent positive and negative exponents
 - $\text{TRUE Exponent} = \text{BIASED exponent} - \text{BIAS}$

Floating Point Representation

- Bias
 - Half precision (16 bits) \rightarrow bias = 15
 - Single precision (32 bits) \rightarrow bias = 127
 - Double precision (64 bits) \rightarrow bias = 1023
- Fraction
 - All of the bits (representing the negative powers of two) are to the right of the binary point
 - The ONLY value to the left of the binary point is the “hidden” or “implicit” one
 - This “implicit one” is there but is not stored in the fraction field

Floating Point Representation

- Ex: fraction is 0b0100101 is actually 1.0100101
- The IEEE754 Floating Point formatted real number is calculated as

$$F = (-1)^s \times (1 + \text{fraction}) \times 2^{(\text{biased exp} - \text{bias})}$$

Floating Point Representation

- XX

ECE342 – Computer Hardware

Lecture 09/10

Bruno Korst, P.Eng.

Agenda

- (1st quiz)
- Floating point

ref: Embedded Systems with ARM Cortex M Microcontrollers – Y. Zhu

Floating Point Representation

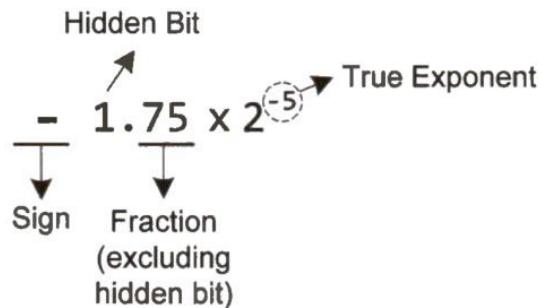
- Real numbers are operated on using Floating Point arithmetic
- Currently, the widely adopted standard is IEEE754
 - Determines format for storage and operation on real numbers
- Many, if not all processors (FPUs) and software libraries support it.

Floating Point Representation

- Representation
 - Uses a normalized notation with **3 bit fields**
 - Sign
 - Fraction – “Mantissa” or fractional part
 - Exponent – a binary field holding a “corrected” or “biased” exponent
 - The “normalized notation” means that there is only one digit before (to the left of) the decimal point, whose value is ONE
 - This is called the “hidden one” or “implicit one”

Floating Point Representation

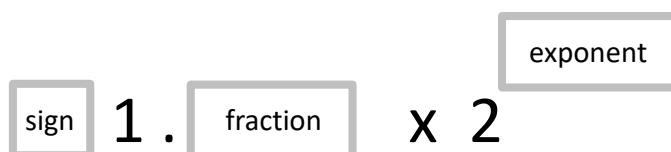
- Notation of IEEE 754



- Resembles scientific notation, but it is expressed with **power of two**
- Before conversion to binary, the integer part of the number must be **ONE**
 - “normalized”

Floating Point Representation

- Examples:
 - 10.746×2^6 is not normalized (integer part is not one)
 - 1.025×10^3 is not normalized (power of 10)
- We seek to express the number **prior to binary conversion** in the form:



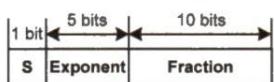
Floating Point Representation

- Let us attempt to convert 10.746×2^6
 - Try to make the integer part equal 1 $\rightarrow (10.746/8) \times 8 \times 2^6$
 - For this case, divide by 8, multiply by 8, adjust the exponent accordingly
 - The converted number is 1.34325×2^9
- Most commonly used formats are single precision and double precision
 - Respectively “float” and “double” in C

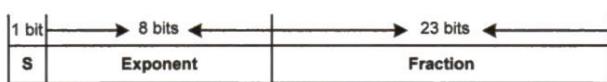
Floating Point Representation

- Field bits for floating point data types

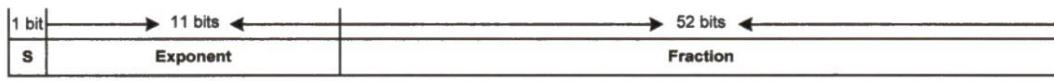
Half Precision (16 bits)



Single Precision (32 bits)



Double Precision (64 bits)



- “Exponent” is a corrected exponent

Floating Point Representation

- Why not just use double?
 - Maybe the hardware does not support that format
 - Maybe single precision will do the job just fine and cost less
- Bit Fields
 - Sign – 0 for positive, 1 for negative
 - Biased Exponent – it's the TRUE exponent of the power of two, corrected to represent positive and negative exponents
 - $\text{TRUE Exponent} = \text{BIASED exponent} - \text{BIAS}$

Floating Point Representation

- Bias
 - Half precision (16 bits) \rightarrow bias = 15
 - Single precision (32 bits) \rightarrow bias = 127
 - Double precision (64 bits) \rightarrow bias = 1023
- Fraction
 - All of the bits (representing the negative powers of two) are to the right of the binary point
 - The ONLY value to the left of the binary point is the “hidden” or “implicit” one
 - This “implicit one” is there but is not stored in the fraction field

Floating Point Representation

- Ex: fraction is 0b0100101 is actually 1.0100101
- The IEEE754 Floating Point formatted real number is calculated as

$$F = (-1)^s \times (1 + \text{fraction}) \times 2^{(\text{biased exp} - \text{bias})}$$

Floating Point Representation

- Example:
 - A register reads 0xC1FF0000
 - What is the decimal value if the number is represented as IEEE754 FP Format?

0xC1FF0000 →  1 | 100 0001 1111 1111 0000 0000 0000 0000

- 32 bit number: 1 bit sign, 8 bit biased exponent, 23 bit fractional part
- s = 1 (negative)
- Biased exponent = $2^7 + 2^1 + 1^0 = 131$
- Fraction is 0.111 1111 0000...0

Floating Point Representation

- Example:
 - A register reads 0xC1FF0000 – Decimal value?
 - $0xC1FF0000 \rightarrow 1 | 100\ 0001\ 1111\ 1111\ 0000\ 0000\ 0000\ 0000$
 - $s = 1$ (negative)
 - Biased exponent = $2^7 + 2^1 + 1^0 = 131$
 - Fraction is $0.111\ 1111\ 0000...0$ ($2^{-1} + 2^{-2} + 2^{-3} + 2^{-4} + 2^{-5} + 2^{-6} + 2^{-7}$)
 - Fraction = 0.9921875
 - Decimal number is $f = (-1)^1 \times (1 + 0.9921875) \times 2^{(131-127)}$
- The decimal number is -31.875**

Floating Point Representation

- Why this bias?
- Say we have a number with the following configuration, in a similar format to IEEE754, with 3 bits for the exponent
 - The 3 bits are to represent both positive and negative integers (exponents)
 - Can represent 0 → 7 (000 → 111) a range of 8
 - I can go from -3 to +4 or from -4 to +3 (in both cases a range of 8)

Floating Point Representation

- Say we choose the range -3 to 4

-3	-2	-1	0	1	2	3	4
000	001	010	011	100	101	110	111
(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)

- We need to find a correction from the “binary” exponent to the **true exponent** of the power of two
 - That is the bias
- If my true exponent is -1, my “binary” exponent is 2. My bias is then 3

Floating Point Representation

- For the three bits used as the exponent field in our example

-3	-2	-1	0	1	2	3	4
000	001	010	011	100	101	110	111
(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)

- The bias is 3 $-3 = 0 - \text{bias}$
 $-2 = 1 - \text{bias}$
 $-1 = 2 - \text{bias}$... $4 = 7 - \text{bias}$
- So $E_{\text{true}} = E_{\text{bias}} - \text{Bias}$

Floating Point Representation

- If we follow this format (3 bits to for the exponent), we must define the exceptions

sign	Exponent	Fractional	Number
X	000	00...0	+/- ZERO
0	111	00...0	+ infinity
1	111	00...0	- infinity
X	111	Any non-zero	NaN

- The question is: why not just use two's complement?
 - During operations exponents need to be compared, and the biased format offers a simpler implementation

Floating Point Representation

- Special values
 - **Zero** – all exponent bits are zero and all fraction bits are zero (sign can be 0 or 1)
 - **Plus infinity** – sign bit is zero all exponent bits are one and all fraction bits are zero
 - **Minus infinity** – sign bit is one, all exponent bits are one and all fraction bits are zero
 - **NaN** (not a number) – all exponent bits are one and fraction is any non-zero
 - Examples of NaN → 0/0, log(-10) , sqrt(-1)

Floating Point Representation

- If the floating point unit (FPU) encounters these, it will indicate via flag (must check...)
- Subnormal numbers (numbers between zero and minimum possible value)
 - Will not deal with them here
 - They occur when all exponent bits are 0 and the fraction is non-zero
- Overflow and Underflow
 - Occur when the value of a number (or result from an operation) fall over the maximum acceptable value or below the minimum acceptable value

Floating Point Representation

- Value limits (finite values)
 - Closest value to zero is exponent 0000 0001 with fraction all zeros (23 bits)

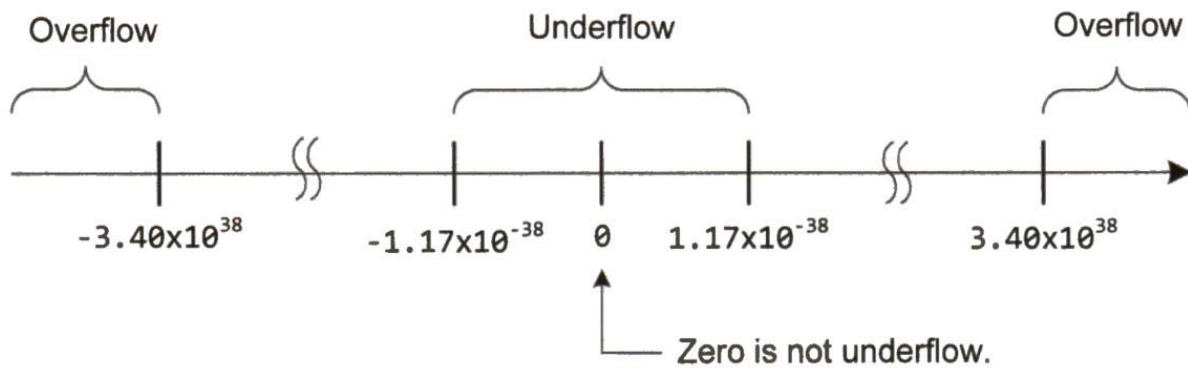
$$(-1)^s \times (1 + 0) \times 2^{1-127} = \pm 2^{-126} \approx \pm 1.18 \times 10^{-38}$$

note: implicit one and all zeros in fraction

- Farthest value from zero is exponent 1111 1110 with fraction all ones (23 bits)

$$(-1)^s \times (1 + (1 - 2^{-23})) \times 2^{254-127} = \pm(2^{128} - 2^{104}) \approx \pm 3.40 \times 10^{38}$$

Floating Point Representation



Floating Point Representation

- Underflow
 - If the exact result is non-zero and smaller than the smallest representable value, an underflow occurs
 - May cause loss of precision and create computational error
- Overflow
 - If the exact result is finite but exceeds the largest representable value, an overflow happens
 - Another source of software failures
- When either are detected, hardware returns zero or max + exception signal

Floating Point Representation

- The hardware raising an exception signal (a flag in a control register) will provide an opportunity for the software to handle abnormalities
- An easy check is to keep an eye on the exponent
- Example: Ariane 5 rocket (1996)
 - Rocket veered off course and was exploded/destructed
 - Cause: a floating point number was assigned to an integer
 - Estimated loss: US\$370M

Floating Point Representation

- Range and Resolution
 - Floating point numbers are **not distributed uniformly across range**
 - Compared to fixed point, for the same number of bits they can represent a larger range but at the cost of resolution
 - **Resolution degrades as we move far away from zero**
 - That is, resolution degrades as the exponent increases
 - Gap between two floating point numbers increases

Floating Point Representation

- Rounding
 - We have a finite number of bits → error in representing real numbers
 - Must approximate, which leads to rounding
 - IEEE754 Few types: to nearest value, toward zero, toward +inf, toward -inf
- Round to nearest value
 - Usually default rounding rule, will go to the nearest value
 - Statistically not biased; sometimes it goes one way, sometimes the other way
 - Note: TRUNCATION always goes only one way

Floating Point Representation

Rounding Rule	Data	Rounded Result
Nearest	+0.123456	+0.12346
	-0.123456	-0.12346
Truncate	+0.123456	+0.12345
	-0.123456	-0.12345
Rounding up	+0.123456	+0.12346
	-0.123456	-0.12345
Rounding down	+0.123456	+0.12345
	-0.123456	-0.12346

Floating Point Representation

- Note:
- When repeatedly adding a set of randomly generated numbers, the rounded sums are
 - Statistically smaller than the true sums if rounded down
 - Statistically larger than the true sums if rounded up
- For a set of negative numbers, the sums are statistically larger than the exact sum if truncated

Floating Point Representation

- Software-based Floating-point operations
 - Some processors do not have an FPU (floating point unit) – a coprocessor – available (Cortex M4 does)
 - Without an FPU coprocessor, the compiler makes the program call the software floating point library to perform multiplication. The library uses integer-based instructions to implement floating point multiplication
 - If FPU is available, multiplication is carried out simply by calling the corresponding assembly instruction.

Floating Point Representation

- Floating point addition between two numbers (assume E1 > E2)

$$f_1 = (-1)^s \times (1 + F_1) \times 2^{E1}$$

$$f_2 = (-1)^s \times (1 + F_2) \times 2^{E2}$$

- It is implemented as

$$\begin{aligned} f_1 + f_2 &= (-1)^s \times (1 + F_1) \times 2^{E1} + (-1)^s \times (1 + F_2) \times 2^{E2} \\ &= (-1)^s \times ((1 + F_1) + (1 + F_2) \times 2^{E2-E1}) \times 2^{E1} \\ &= (-1)^s \times \left((1 + F_1) + \frac{1 + F_2}{2^{E1-E2}} \right) \times 2^{E1} \end{aligned}$$

Floating Point Representation

- Note that the division is really a shift!
- Procedure
 - Shift smaller fraction to match that with larger exponent (that is: make exponents be the same)
 - Add/subtract fraction (depending on the sign bit)
 - Round sum to the appropriate bits
 - Detect overflow/underflow

Floating Point Representation

- Example: $x = (1.1011011101)_2 \times 2^4$

$$y = (1.1110101111)_2 \times 2^5$$

- Shift the smaller to match the larger $x = (0.11011011101)_2 \times 2^5$
- Perform addition

$$\begin{array}{r} x = 0 . \quad 1 \quad 1 \quad 0 \quad 1 \quad 1 \quad 0 \quad 1 \quad 1 \quad 1 \quad 1 \quad 0 \quad 1 \\ y = 1 . \quad 1 \quad 1 \quad 1 \quad 0 \quad 1 \quad 0 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 0 \\ \hline x + y = 1 \quad 0 . \quad 1 \quad 1 \quad 0 \quad 0 \quad 0 \quad 0 \quad 1 \quad 1 \quad 1 \quad 0 \quad 1 \quad 1 \end{array}$$

Floating Point Representation

- The result is not normalized (put one to the left of the binary point)

$$x + y = (1.011000111011)_2 \times 2^6$$

- Check for overflow

Floating Point Representation

- Multiplication
 - Add sign exponent, multiply the 1+fractional parts, and add exponents

$$\begin{aligned}f_1 \times f_2 &= ((-1)^{S1} \times (1 + F_1) \times 2^{E1}) \times ((-1)^{S2} \times (1 + F_2) \times 2^{E2}) \\&= (-1)^{S1+S2} \times (1 + F_1) \times (1 + F_2) \times 2^{E1+E2}\end{aligned}$$

- Procedure:
 - Identify the sign of the product
 - Add exponents (check too large/too small), handle the two biases
 - Multiply fractional part including the implicit one
 - Normalize the result

ECE342 – Computer Hardware

Lecture 09/10

Bruno Korst, P.Eng.

Agenda

- (1st quiz)
- Floating point

ref: Embedded Systems with ARM Cortex M Microcontrollers – Y. Zhu

Floating Point Representation

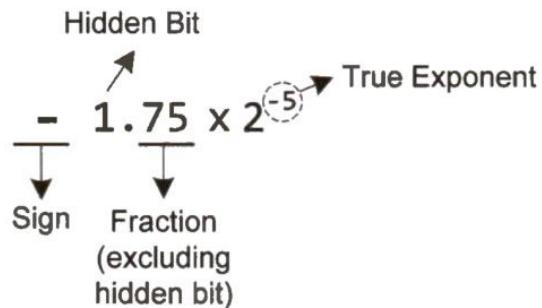
- Real numbers are operated on using Floating Point arithmetic
- Currently, the widely adopted standard is IEEE754
 - Determines format for storage and operation on real numbers
- Many, if not all processors (FPUs) and software libraries support it.

Floating Point Representation

- Representation
 - Uses a normalized notation with **3 bit fields**
 - Sign
 - Fraction – “Mantissa” or fractional part
 - Exponent – a binary field holding a “corrected” or “biased” exponent
 - The “normalized notation” means that there is only one digit before (to the left of) the decimal point, whose value is ONE
 - This is called the “hidden one” or “implicit one”

Floating Point Representation

- Notation of IEEE 754



- Resembles scientific notation, but it is expressed with **power of two**
- Before conversion to binary, the integer part of the number must be **ONE**
 - “normalized”

Floating Point Representation

- Examples:
 - 10.746×2^6 is not normalized (integer part is not one)
 - 1.025×10^3 is not normalized (power of 10)
- We seek to express the number **prior to binary conversion** in the form:



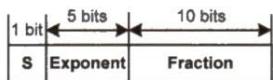
Floating Point Representation

- Let us attempt to convert 10.746×2^6
 - Try to make the integer part equal 1 $\rightarrow (10.746/8) \times 8 \times 2^6$
 - For this case, divide by 8, multiply by 8, adjust the exponent accordingly
 - The converted number is 1.34325×2^9
- Most commonly used formats are single precision and double precision
 - Respectively “float” and “double” in C

Floating Point Representation

- Field bits for floating point data types

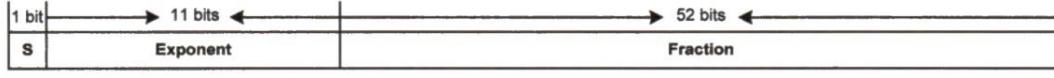
Half Precision (16 bits)



Single Precision (32 bits)



Double Precision (64 bits)



- “Exponent” is a corrected exponent

Floating Point Representation

- Why not just use double?
 - Maybe the hardware does not support that format
 - Maybe single precision will do the job just fine and cost less
- Bit Fields
 - Sign – 0 for positive, 1 for negative
 - Biased Exponent – it's the TRUE exponent of the power of two, corrected to represent positive and negative exponents
 - $\text{TRUE Exponent} = \text{BIASED exponent} - \text{BIAS}$

Floating Point Representation

- Bias
 - Half precision (16 bits) \rightarrow bias = 15
 - Single precision (32 bits) \rightarrow bias = 127
 - Double precision (64 bits) \rightarrow bias = 1023
- Fraction
 - All of the bits (representing the negative powers of two) are to the right of the binary point
 - The ONLY value to the left of the binary point is the “hidden” or “implicit” one
 - This “implicit one” is there but is not stored in the fraction field

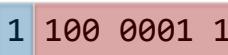
Floating Point Representation

- Ex: fraction is 0b0100101 is actually 1.0100101
- The IEEE754 Floating Point formatted real number is calculated as

$$F = (-1)^s \times (1 + \text{fraction}) \times 2^{(\text{biased exp} - \text{bias})}$$

Floating Point Representation

- Example:
 - A register reads 0xC1FF0000
 - What is the decimal value if the number is represented as IEEE754 FP Format?

0xC1FF0000 →  1 | 100 0001 1111 1111 0000 0000 0000 0000

- 32 bit number: 1 bit sign, 8 bit biased exponent, 23 bit fractional part
- s = 1 (negative)
- Biased exponent = $2^7 + 2^1 + 1^0 = 131$
- Fraction is 0.111 1111 0000...0

Floating Point Representation

- Example:
 - A register reads 0xC1FF0000 – Decimal value?
 - $0xC1FF0000 \rightarrow 1 | 100\ 0001\ 1111\ 1111\ 0000\ 0000\ 0000\ 0000$
 - $s = 1$ (negative)
 - Biased exponent = $2^7 + 2^1 + 1^0 = 131$
 - Fraction is $0.111\ 1111\ 0000...0$ ($2^{-1} + 2^{-2} + 2^{-3} + 2^{-4} + 2^{-5} + 2^{-6} + 2^{-7}$)
 - Fraction = 0.9921875
 - Decimal number is $f = (-1)^1 \times (1 + 0.9921875) \times 2^{(131-127)}$
- The decimal number is -31.875**

Floating Point Representation

- Why this bias?
- Say we have a number with the following configuration, in a similar format to IEEE754, with 3 bits for the exponent
 - The 3 bits are to represent both positive and negative integers (exponents)
 - Can represent 0 → 7 (000 → 111) a range of 8
 - I can go from -3 to +4 or from -4 to +3 (in both cases a range of 8)

Floating Point Representation

- Say we choose the range -3 to 4

-3	-2	-1	0	1	2	3	4
000	001	010	011	100	101	110	111
(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)

- We need to find a correction from the “binary” exponent to the **true exponent** of the power of two
 - That is the bias
- If my true exponent is -1, my “binary” exponent is 2. My bias is then 3

Floating Point Representation

- For the three bits used as the exponent field in our example

-3	-2	-1	0	1	2	3	4
000	001	010	011	100	101	110	111
(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)

- The bias is 3 $-3 = 0 - \text{bias}$
 $-2 = 1 - \text{bias}$
 $-1 = 2 - \text{bias}$... $4 = 7 - \text{bias}$
- So $E_{\text{true}} = E_{\text{bias}} - \text{Bias}$

Floating Point Representation

- If we follow this format (3 bits to for the exponent), we must define the exceptions

sign	Exponent	Fractional	Number
X	000	00...0	+/- ZERO
0	111	00...0	+ infinity
1	111	00...0	- infinity
X	111	Any non-zero	NaN

- The question is: why not just use two's complement?
 - During operations exponents need to be compared, and the biased format offers a simpler implementation

Floating Point Representation

- Special values
 - **Zero** – all exponent bits are zero and all fraction bits are zero (sign can be 0 or 1)
 - “minus zero” can play a role in a division by zero (“minus infinity”)
 - **Plus infinity** – sign bit is zero all exponent bits are one and all fraction bits are zero
 - **Minus infinity** – sign bit is one, all exponent bits are one and all fraction bits are zero
 - **NaN** (not a number) – all exponent bits are one and fraction is any non-zero
 - Examples of NaN → 0/0, log(-10) , sqrt(-1)

Floating Point Representation

- If the floating point unit (FPU) encounters these, it will indicate via flag (must check...)
- Subnormal numbers (numbers between zero and minimum possible value)
 - Will not deal with them here
 - They occur when all exponent bits are 0 and the fraction is non-zero
- Overflow and Underflow
 - Occur when the value of a number (or result from an operation) fall over the maximum acceptable value or below the minimum acceptable value

Floating Point Representation

- Value limits (finite values)
 - Closest value to zero is exponent 0000 0001 with fraction all zeros (23 bits)

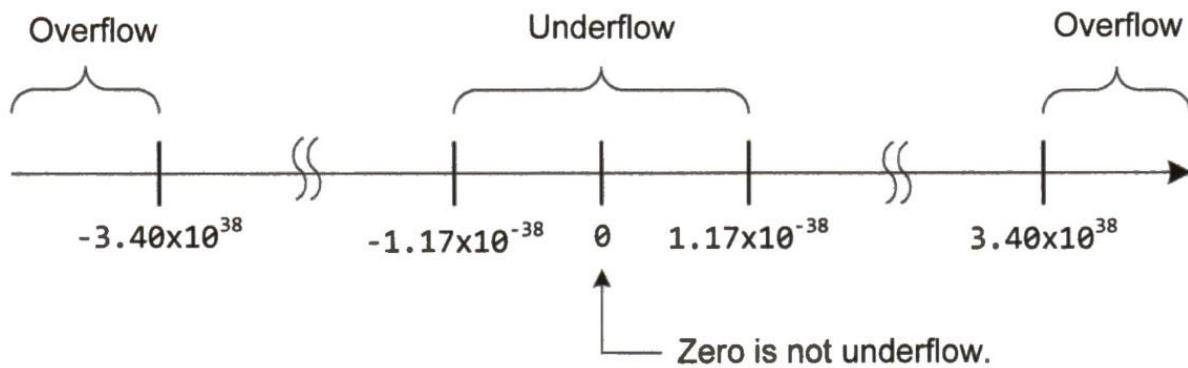
$$(-1)^s \times (1 + 0) \times 2^{1-127} = \pm 2^{-126} \approx \pm 1.18 \times 10^{-38}$$

note: implicit one and all zeros in fraction

- Farthest value from zero is exponent 1111 1110 with fraction all ones (23 bits)

$$(-1)^s \times (1 + (1 - 2^{-23})) \times 2^{254-127} = \pm(2^{128} - 2^{104}) \approx \pm 3.40 \times 10^{38}$$

Floating Point Representation



Floating Point Representation

- Underflow
 - If the exact result is non-zero and smaller than the smallest representable value, an underflow occurs
 - May cause loss of precision and create computational error
- Overflow
 - If the exact result is finite but exceeds the largest representable value, an overflow happens
 - Another source of software failures
- When either are detected, hardware returns zero or max + exception signal

Floating Point Representation

- The hardware raising an exception signal (a flag in a control register) will provide an opportunity for the software to handle abnormalities
- An easy check is to keep an eye on the exponent
- Example: Ariane 5 rocket (1996)
 - Rocket veered off course and was exploded/destructed
 - Cause: a floating point number was assigned to an integer
 - Estimated loss: US\$370M

Floating Point Representation

- Range and Resolution
 - Floating point numbers are **not distributed uniformly across range**
 - Compared to fixed point, for the same number of bits they can represent a larger range but at the cost of resolution
 - **Resolution degrades as we move far away from zero**
 - That is, resolution degrades as the exponent increases
 - Gap between two floating point numbers increases

Floating Point Representation

- Rounding
 - We have a finite number of bits → error in representing real numbers
 - Must approximate, which leads to rounding
 - IEEE754 Few types: to nearest value, toward zero, toward +inf, toward -inf
- Round to nearest value
 - Usually default rounding rule, will go to the nearest value
 - Statistically not biased; sometimes it goes one way, sometimes the other way
 - Note: TRUNCATION always goes only one way

Floating Point Representation

Rounding Rule	Data	Rounded Result
Nearest	+0.123456	+0.12346
	-0.123456	-0.12346
Truncate	+0.123456	+0.12345
	-0.123456	-0.12345
Rounding up	+0.123456	+0.12346
	-0.123456	-0.12345
Rounding down	+0.123456	+0.12345
	-0.123456	-0.12346

Floating Point Representation

- Note:
- When repeatedly adding a set of randomly generated numbers, the rounded sums are
 - Statistically smaller than the true sums if rounded down
 - Statistically larger than the true sums if rounded up
- For a set of negative numbers, the sums are statistically larger than the exact sum if truncated

Floating Point Representation

- Software-based Floating-point operations
 - Some processors do not have an FPU (floating point unit) – a coprocessor – available (Cortex M4 does)
 - Without an FPU coprocessor, the compiler makes the program call the software floating point library to perform multiplication. The library uses integer-based instructions to implement floating point multiplication
 - If FPU is available, multiplication is carried out simply by calling the corresponding assembly instruction.

Floating Point Representation

- Floating point addition between two numbers (assume E1 > E2)

$$f_1 = (-1)^s \times (1 + F_1) \times 2^{E1}$$

$$f_2 = (-1)^s \times (1 + F_2) \times 2^{E2}$$

- It is implemented as

$$\begin{aligned} f_1 + f_2 &= (-1)^s \times (1 + F_1) \times 2^{E1} + (-1)^s \times (1 + F_2) \times 2^{E2} \\ &= (-1)^s \times ((1 + F_1) + (1 + F_2) \times 2^{E2-E1}) \times 2^{E1} \\ &= (-1)^s \times \left((1 + F_1) + \frac{1 + F_2}{2^{E1-E2}} \right) \times 2^{E1} \end{aligned}$$

Floating Point Representation

- Note that the division which shows in the operation is really a shift!
- Procedure
 - Shift smaller fraction to match that with larger exponent (that is: make exponents be the same)
 - Note: shift RIGHT the fraction of the operand with the SMALLER exponent
 - Shifting LEFT the fraction of the larger exponent can lead to overflow...
 - Add/subtract fraction (depending on the sign bit)
 - Round sum to the appropriate bits
 - Detect overflow/underflow

Floating Point Representation

- Example: Adding two numbers

$$x = (1.1011011101)_2 \times 2^4$$

$$y = (1.1110101111)_2 \times 2^5$$

- Shift the smaller FRACTION to match the larger EXPONENT

$$x = (0.11011011101)_2 \times 2^5$$

- Perform addition

$$\begin{array}{r} x = 0 . \quad 1 \quad 1 \quad 0 \quad 1 \quad 1 \quad 0 \quad 1 \quad 1 \quad 1 \quad 0 \quad 1 \\ y = 1 . \quad 1 \quad 1 \quad 1 \quad 0 \quad 1 \quad 0 \quad 1 \quad 1 \quad 1 \quad 1 \quad 0 \\ \hline x + y = 1 0 . \quad 1 \quad 1 \quad 0 \quad 0 \quad 0 \quad 1 \quad 1 \quad 1 \quad 0 \quad 1 \quad 1 \end{array}$$

Floating Point Representation

- The result is computed and needs normalization

- That is: put one to the left of the binary point, correct the exponent)

$$x + y = (1.011000111011)_2 \times 2^6$$

- Check for overflow

Floating Point Representation

- Multiplication
 - Add sign exponent, multiply the 1+fractional parts, and add exponents

$$\begin{aligned}f_1 \times f_2 &= ((-1)^{S1} \times (1 + F_1) \times 2^{E1}) \times ((-1)^{S2} \times (1 + F_2) \times 2^{E2}) \\&= (-1)^{S1+S2} \times (1 + F_1) \times (1 + F_2) \times 2^{E1+E2}\end{aligned}$$

- Procedure:
 - Identify the sign of the product
 - Add exponents (check too large/too small), handle the two biases
 - Multiply fractional part including the implicit one
 - Normalize the result

ECE342 – Computer Hardware

Lecture 11

Bruno Korst, P.Eng.

Agenda

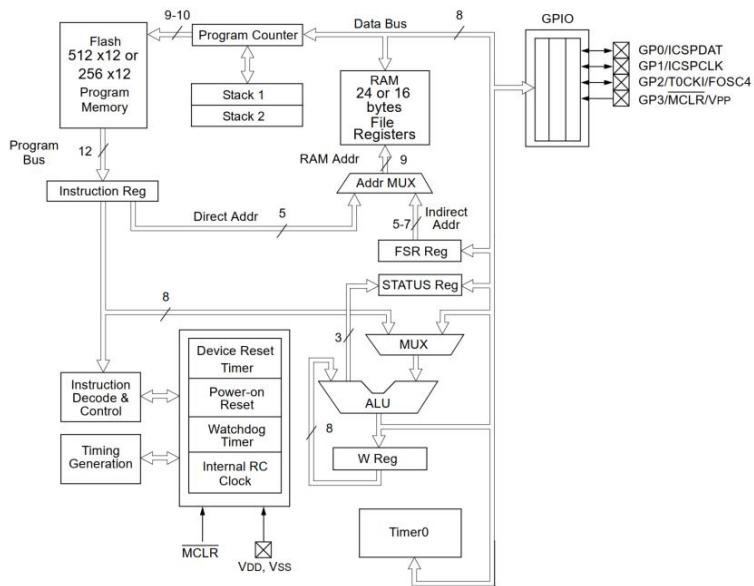
- Revisiting the architecture
- The BUS
 - Parts, examples
 - On-chip and Off-Chip
- Business:
 - next lab → RTC module using I2C
 - Midterm: Feb13 – all available 4pm to 6pm?

ref: Embedded Systems with ARM Cortex M Microcontrollers – Y. Zhu

Revisiting architecture

- Processor / CPU
 - Hardcore, Softcore – hardware that executes instructions
 - Performs mathematical operations in different numerical formats
- Bus
 - Collection of wires that pass information between modules (data, addr., control)
- Memory
 - Hardware elements that store information – read/write, addresses
- Ports
 - Physical connection to the outside, through a variety of protocols
 - We've seen A/D, D/A with the use of interrupts

General (simple) architecture

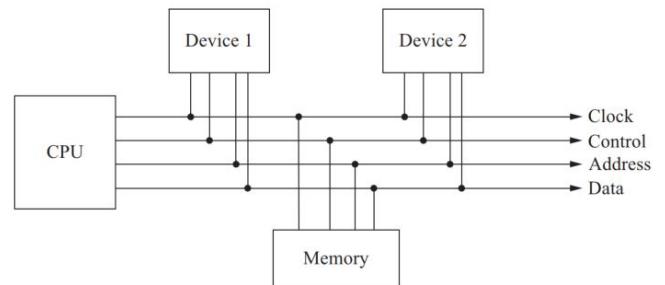


Bus

- BUS
 - Mechanism of communication between CPU, memory and devices
 - There has to be a protocol (rules for transferring)
 - Can be
 - “on-chip” - connecting cores within the device (FPGA – SoC) or
 - “off-chip” – connecting modules on a PCB
 - It's referred to as a “bundle” of wires as well as a protocol (USB, I2C)
 - “The I2C bus”
 - Its width (bits) and clock (“per second”) determine throughput
 - Typically carry clock, control, address and data lines

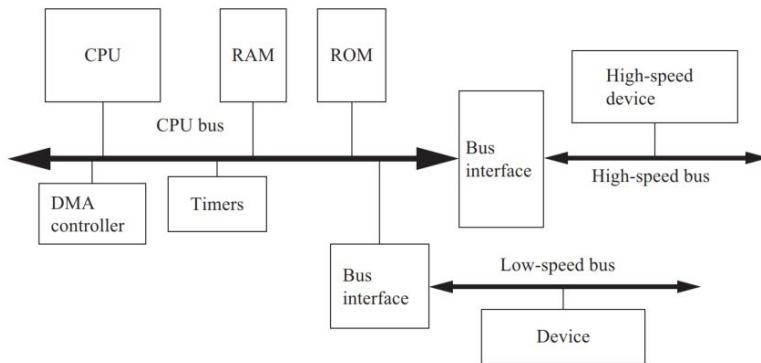
Bus

- CPU serves as a bus master – initiates transfers
 - “master/slave” configuration
- Control handles requests, acknowledgement and other protocol specific data
- Sometimes bus has multiple masters
 - Needs arbitration
 - who controls the bus



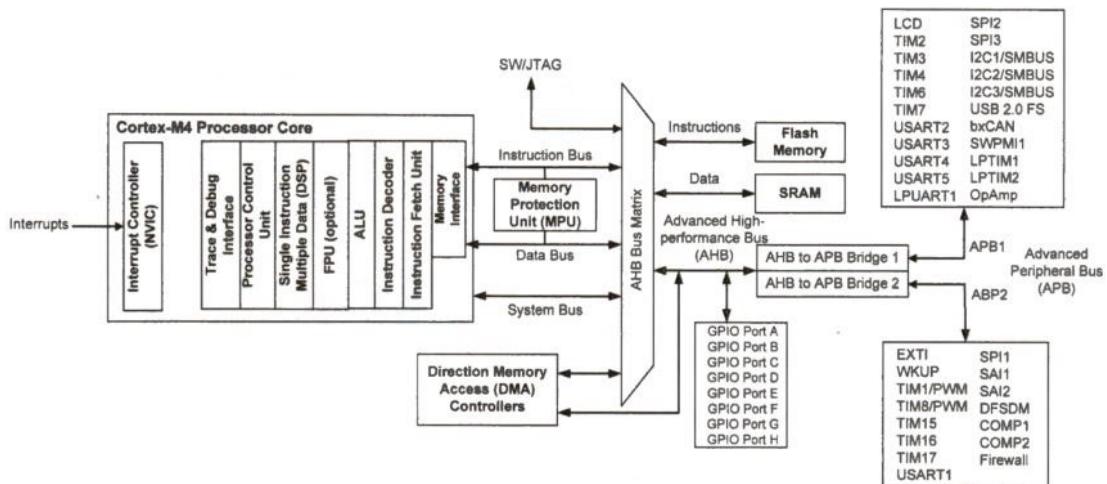
Bus

- One may have multiple buses
 - Operating at different speeds, connecting different modules
 - When that is the case, some bus interface is needed for them to interact



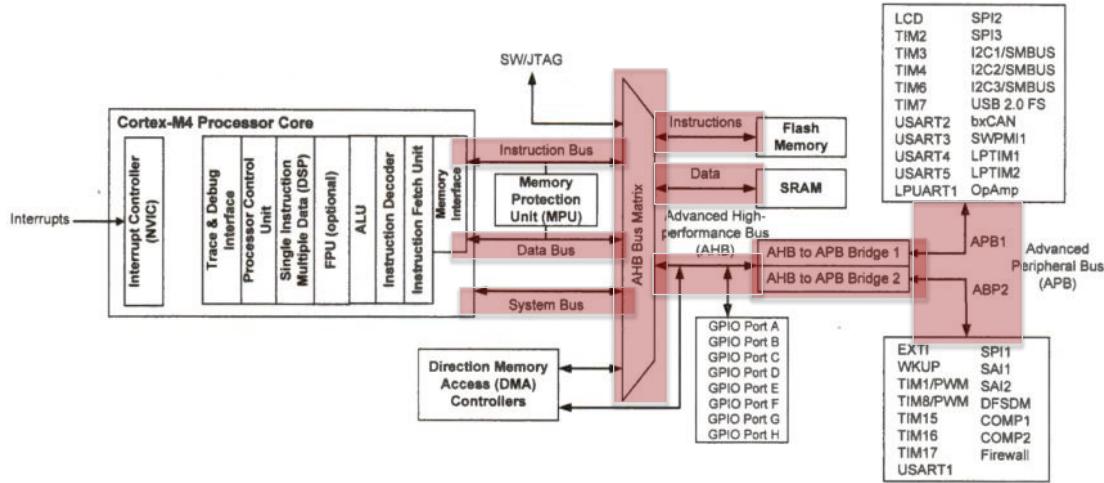
Bus

- Multiple buses – Cortex M4 (our device)



Bus

- Multiple buses – Cortex M4 (our device)

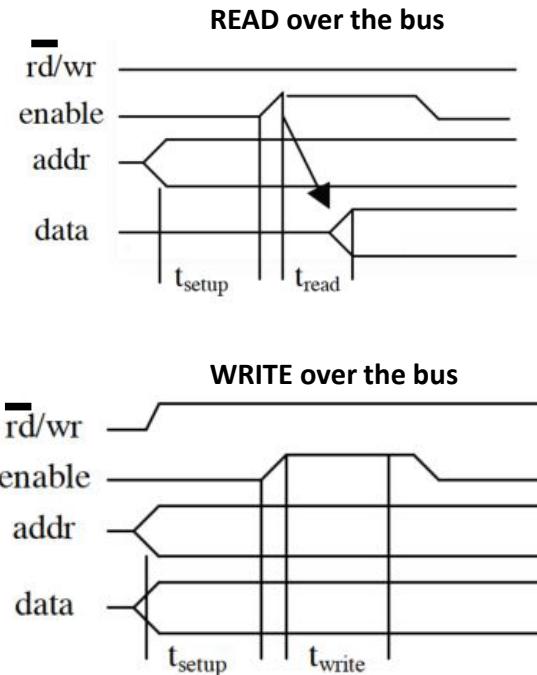
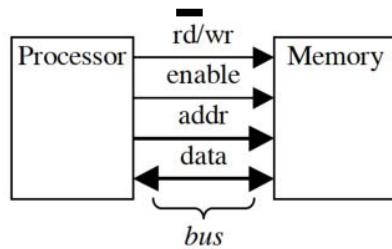


Bus - Protocol

- The protocol (rules of transfer) is described by the timing diagram
- Example of a protocol
 - Set rd/wr low to read, high to write
 - Address must be placed before enabling the line
 - At least a “set up time” before
 - High enable triggers the data to be made available (after tread)
 - Control lines represented as high/low
 - Data/address lines represented as valid/not valid
 - Active high/low depends on protocol

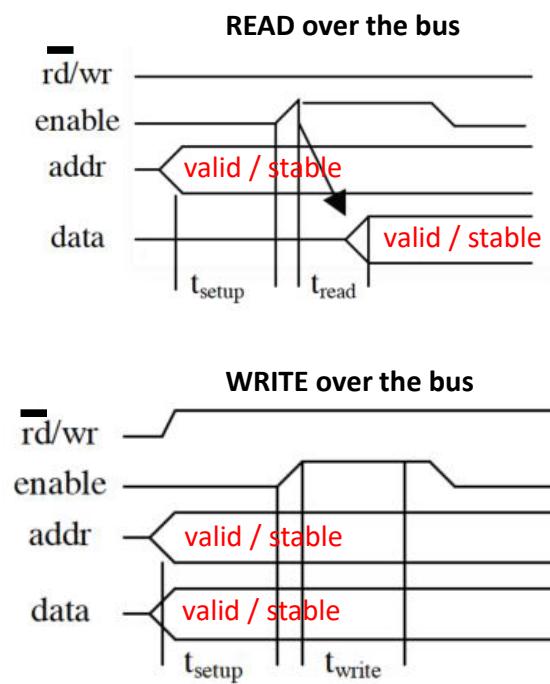
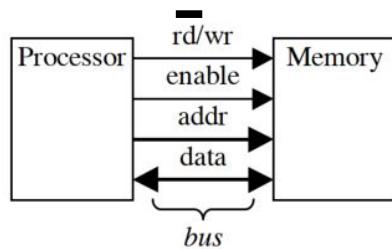
Bus - Protocol

- Protocol example



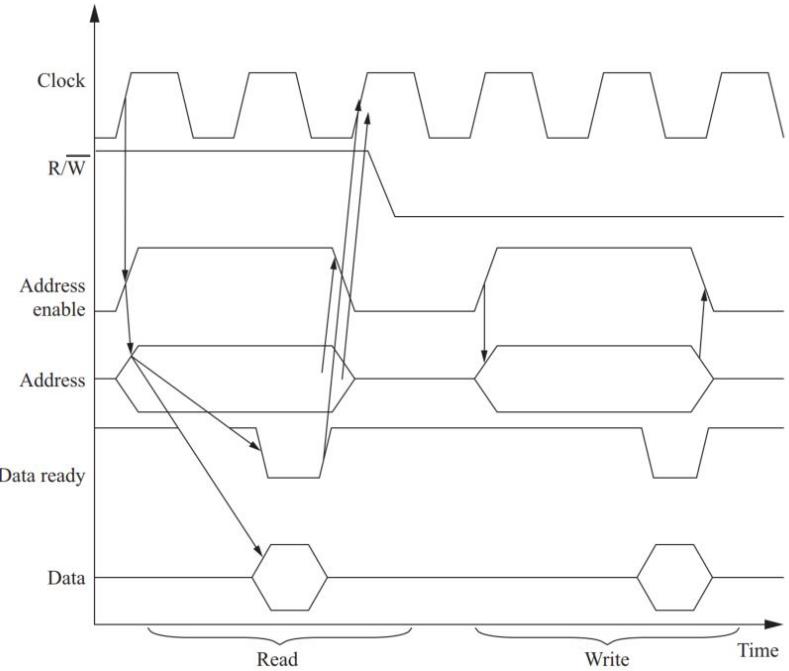
Bus - Protocol

- Protocol example



Bus - Protocol

- Protocol example: read/write
 - read high (default)
 - address enable high
 - will read
 - address lines set
 - active low data rdy
 - data read

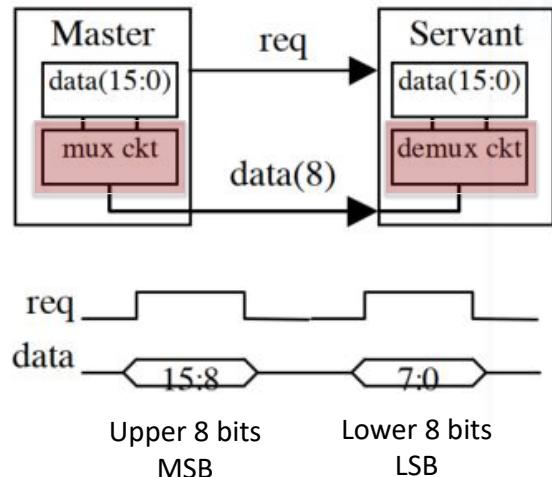


Bus - Protocol

- Defines the actors: primary/secondary
 - Literature in digital electronics often refers to master/slave or master/servant
 - Master initiates data transfer (usually the processors)
 - Slave responds to initiation request
 - Can be other processors, but typically these are peripherals/memory
- Defines the data direction: receive or transmit
- Defines the address
 - Indicates where the data goes or comes from
- Defines how the data shares the wire(s)

Bus – Protocol

- Same lines are shared – time multiplexing (one piece at a time)
 - Multiplex on one side, demultiplex on the other side
 - Ex: 8 bits bus transferring 16 bits

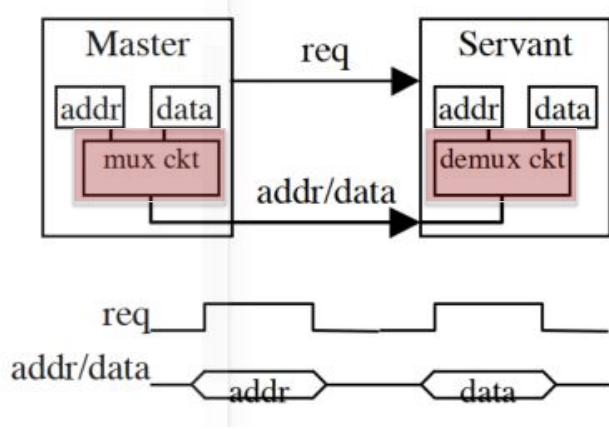


Bruno Korst, P.Eng. - Winter 2023

15

Bus – Protocol

- Same lines are shared – time multiplexing (one piece at a time)
 - Ex: Bus being shared between address and data

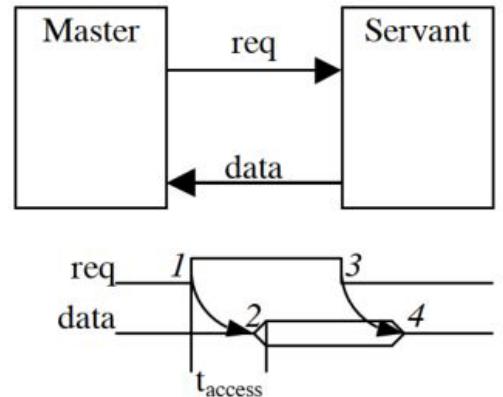


Bruno Korst, P.Eng. - Winter 2023

16

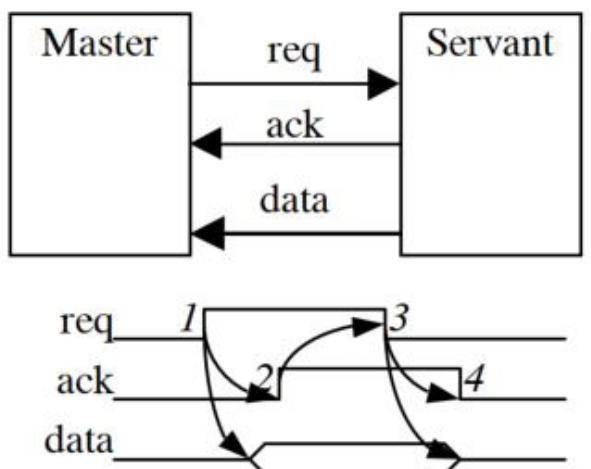
Bus - Protocol

- Control methods
 - Strobe – request and time
 - 1 - Master asserts request
 - 2 - Servant puts data on bus **within the deadline**
 - 3 - Master receives data and deasserts request
 - 4 - Servant ready for next request



Bus - Protocol

- Control methods
 - Handshake
 - 1 – master asserts request
 - 2 – Servant puts data on bus and acknowledges that data is ready
(master reads data)
 - 3 – Master deasserts request
 - 4 – Servant stops transmitting data and deasserts acknowledge
 - Both ready for next request
 - Note: this method has no deadline and can adjust to the servant's response time

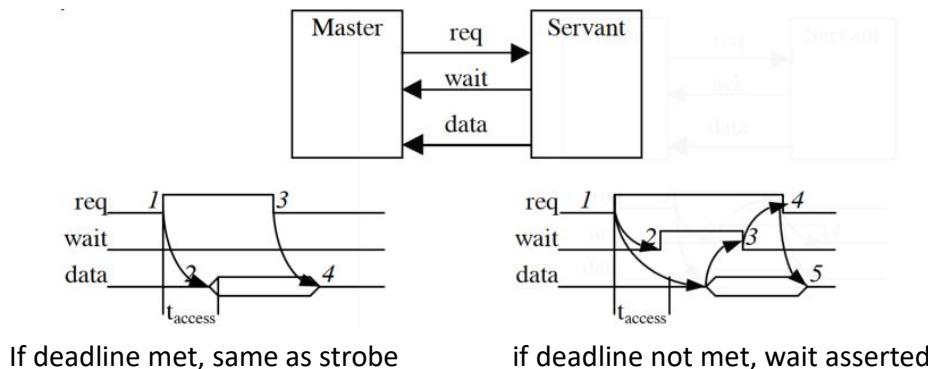


Bus - Protocol

- Control methods
 - Handshake
 - may be slower than strobe – requires master to detect acknowledgement
 - requires extra line for acknowledgement
 - Strobe
 - when response time from servant(s) is known, no need to have an extra line or to wait for acknowledgement

Bus - Protocol

- There is the possibility of missing the time constraints
- Adding a wait control signal
 - Request is asserted; if time is met, operate same as strobe
 - Otherwise, wait – servant will assert wait if deadline is missed

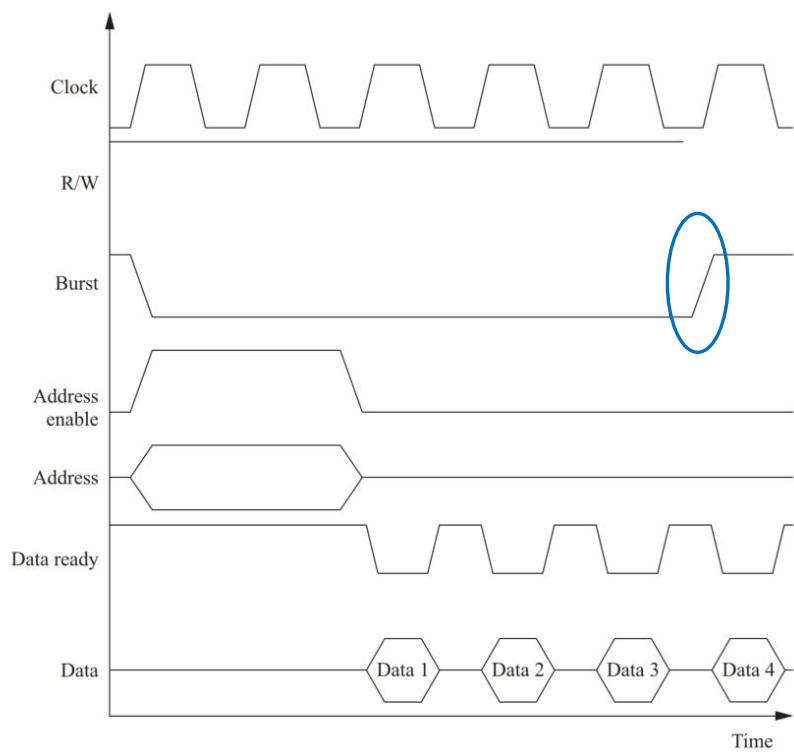


Bus - Protocol

- Control method – Burst transfer
 - Once start address is given by CPU, bursts of data are received from n successive locations
 - One extra line needed to indicate a burst transfer
 - Releasing the burst line indicates enough data has been transmitted
 - In the following diagram, burst line is released after data3 in order to stop receiving data at the end of data4
 - It takes time to recognize the end of the burst

Bus - Protocol

- Burst transfer



Bus – Arbitration

- When multiple actors request the use of the bus, some sort of arbitration is needed
- If all requests are simultaneous, there is a problem and a decision needs to be made as to who controls the bus
- Several methods
 - Priority arbitration
 - Daisy-chain arbitration
 - Network oriented arbitration
 - Time division arbitration
 - Dynamic, programmable, etc...

Bus - Arbitration

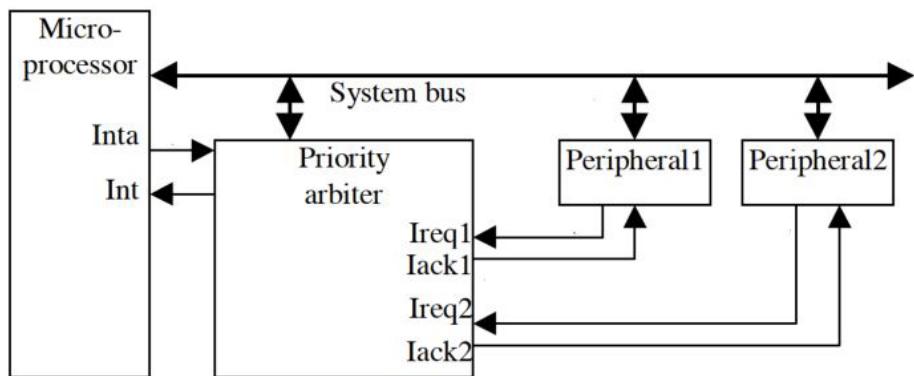
- The process
 - Peripherals make request to arbiter
 - Arbiter interrupts processor
 - Processor acknowledges
 - The arbiter decides which peripheral will place its vector address on the bus
 - This is the address to which the processor will jump to execute the routine corresponding to the peripheral

Bus - Arbitration

- **Fixed Priority:** the arbiter has a rank of peripherals
 - Some processors allow for interrupt priorities, as we saw
 - Higher priority wins – this may lead to a problem for the lower priority requests
- **Preemptive / Non-preemptive Priority**
 - Lower priority terminated by higher or allowed to compete
- **Round-robin priority** (rotating priority) – requires a more complex arbiter

Bus - Arbitration

- Fixed priority is better if there is a clear difference in priority
 - Interrupt controllers will handle it
- If priorities are somewhat equal, a rotating scheme is acceptable

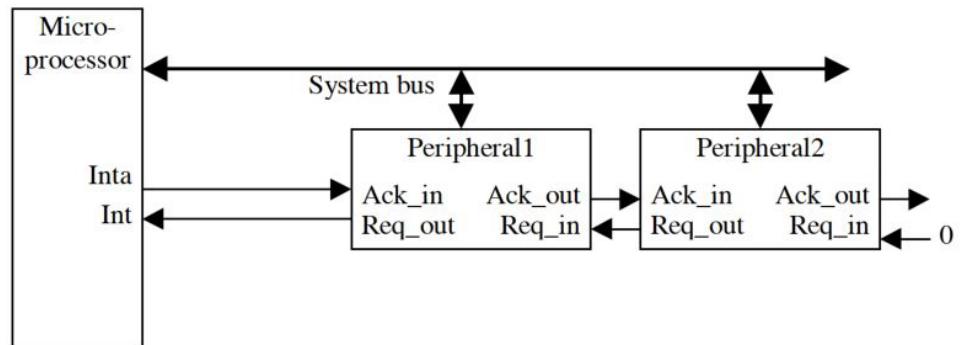


Bus - Arbitration

- Daisy-chain arbitration
 - arbitration is **moved to the peripherals**
 - peripherals are daisy-chained
 - upstream peripherals “receive” requests from downstream and pass it along
 - Processor sees only one request
 - when processor acknowledges request, requesting peripheral closest to processor gets serviced
 - Highest priority peripheral is at the top of the chain, closest to processor

Bus - Arbitration

- Daisy-chain arbitration



- Adding/removing peripherals is simplified
- Problem: if chain is too long, timing may become an issue
- Problem: chain is broken if one peripheral stops working

Bus - Arbitration

- Network Oriented Arbitration
 - when system has **multiple processors using a shared bus**
 - arbitration is build into the bus protocol
 - collision detection
 - if data collision happens wait, transmit again
 - simultaneous addressing
 - two processors writing same address to the bus
 - address written by processor with highest priority overrides the one with lowest priority

Bus - Arbitration

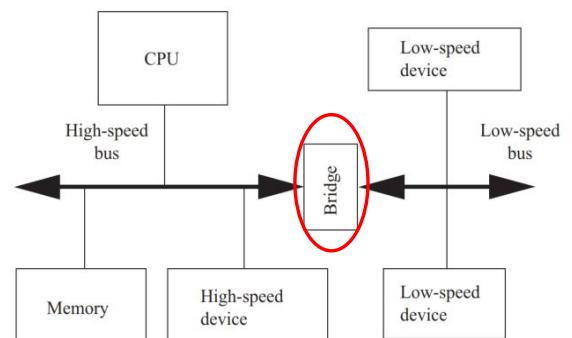
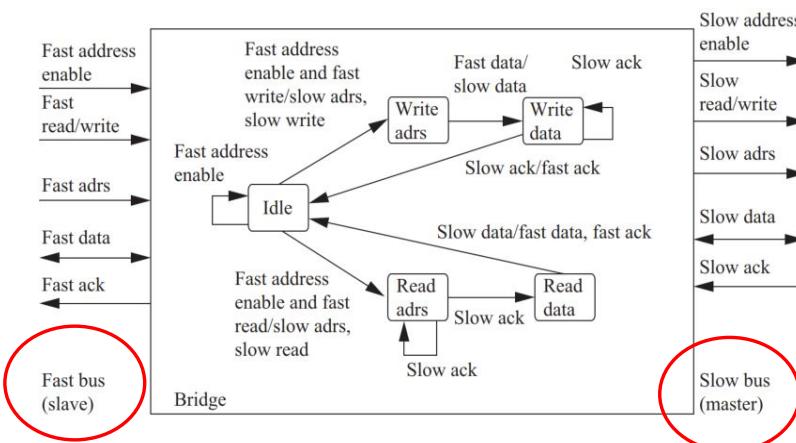
- Time Division Multiple Access
 - fixed, higher bandwidth to masters with higher data transfer requirements
 - **ensure lower priorities do not starve**
 - Each master is assigned time slots of varying lengths depending on bandwidth requirements
- Dynamic Priority
 - dynamically vary the priorities of masters during execution (analyze traffic at runtime)
- Programmable Priority – write to arbiter's reg.

Bus – Multi-level Bus

- Often there is more than one bus
 - High speed devices connected to high performance bus
 - Low speed devices connected to a different bus
- High speed → wider data connection
- Bridge – the logic that connects between buses
 - It's a state machine or dedicated processor
 - It also allows buses to operate independently
 - I/O operations may be done in parallel

Bus – Multi-level Bus

- Bridge is a servant of the fast bus
- Bridge is a master of the slow bus



Bus – Multi-level Bus

- Bridge
 - Takes commands from the fast bus
 - Issues those commands to the slow bus, then returns the results from the slow bus to the fast bus
 - It also serves as a protocol translator
 - Ex: ARM has a particular bus specification (AMBA – details later)
 - Includes a high performance bus and a peripherals bus
 - Supports CPUs, memories and peripherals

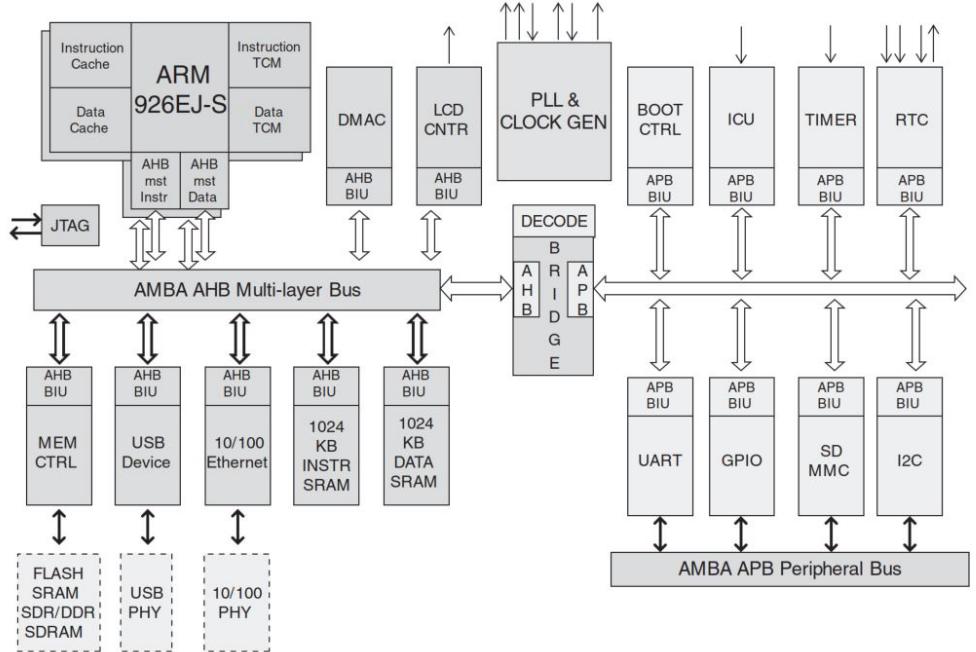
Bus – Multi-level Bus

- Some components can be either master or servant at a time (hybrid)
 - Example: a memory controller is a servant when being configured, but is a master when writing blocks to memory

Bus - Arbitration

AHB
Advanced High Performance Bus

APB
Advanced Peripheral Bus



Bruno Korst, P.Eng. - Winter 2023

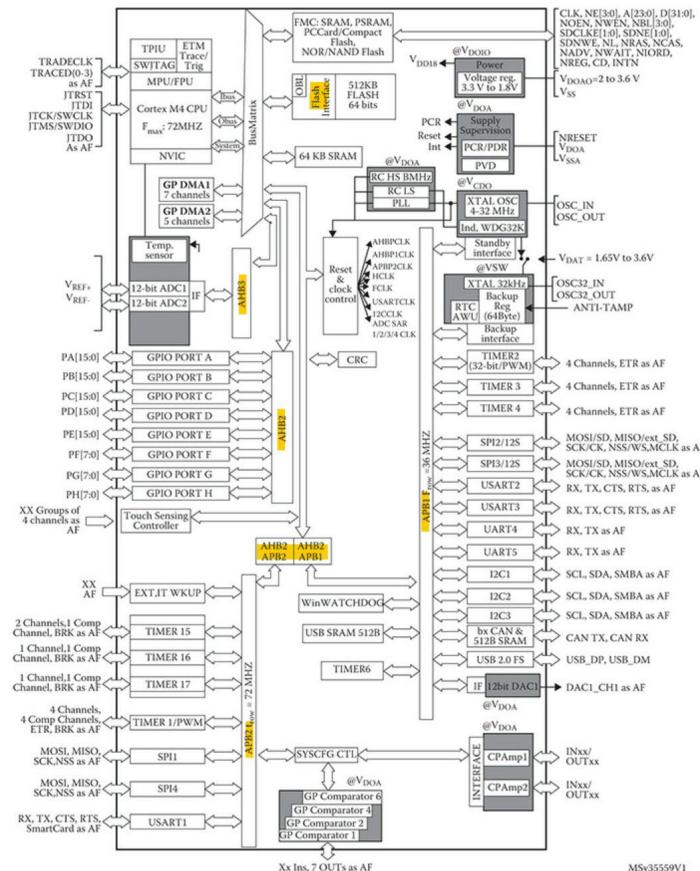
35

Bus - Arbitration

STM32F4x

Two AHB
Advanced High Performance Bus

Two APB
Advanced Peripheral Bus



Bruno Korst, P.Eng. - Winter 2023

36

MSV35559V1

ECE342 – Computer Hardware

Lecture 12

Bruno Korst, P.Eng.

Agenda

- Housekeeping
- I2C protocol
- Continuing with Bus

Housekeeping

- Have asked for extra hours – This Friday 12pm to 2pm
- Some groups did not finish lab on time
 - No scope? No problem!
 - Last question on the quiz
- Don't get stuck on the “no global” declaration. That's for APS105
- Need to find a date for midterm. Consulted the UG office.

Bus

- We saw:
 - What it is: “collection of wires”
 - **Protocol**, which establishes the communication between...
 - Actors, such as processor, peripherals and bus, where...
 - **Control, address and data** are exchanged.
 - Actors can have roles of
 - **Master**/primary – **initiates** data transfer
 - **Servants**/secondary – **respond** to the initiation request
 - When multiple actors/masters request control of the bus, we need an **arbitration** strategy
 - When multiple buses are present (different widths, frequencies), a **bridge** provides the logic to connect them

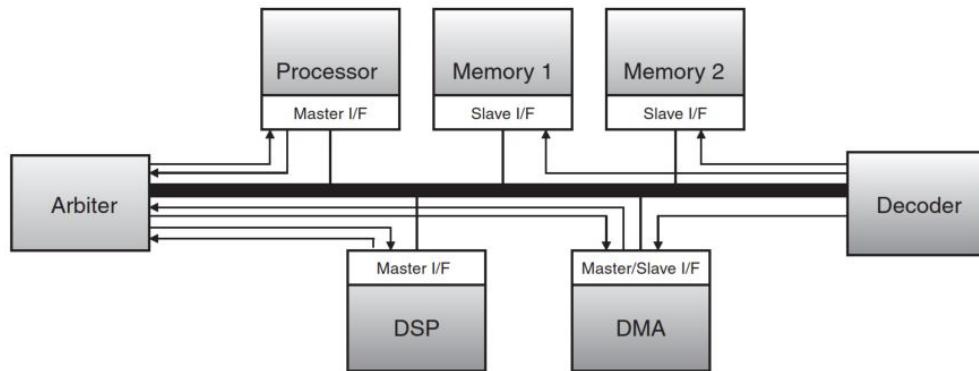
Bus

- Memory-mapped peripherals are assigned an address (or a range of addresses)
 - a **Decoder** decodes the address to select which component receives the data
 - **Centralized** decoding
 - master issues address, decoder selects component (servant)
 - sends signal to component indicating r/w
 - if more components need to be added, process is simplified
 - works **in conjunction with arbiter**
 - **Distributed** decoding
 - components (servant) have their own decoder (and arbiter if necessary)
 - local decoder determines if it (the servant) is the correct addressee
 - there is more hardware duplication but fewer control lines

5

Bus

- Centralized decoding

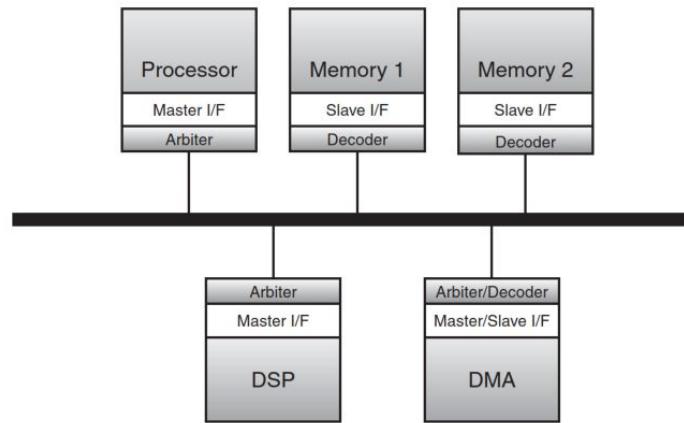


- I/F stands for “interface”

6

Bus

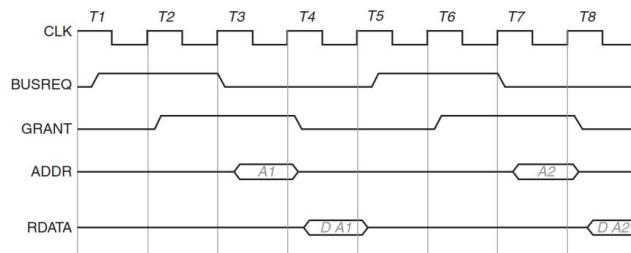
- Distributed decoding



7

Bus

- Let's revisit data transfer
 - Simplest transfer (read)
 - master requests access to bus from the arbiter
 - arbiter grants
 - next cycle → master sends out address
 - decoder identifies correct slave from address given
 - slave sends data to be read (if multiple masters, longer arbitration)



8

Bus

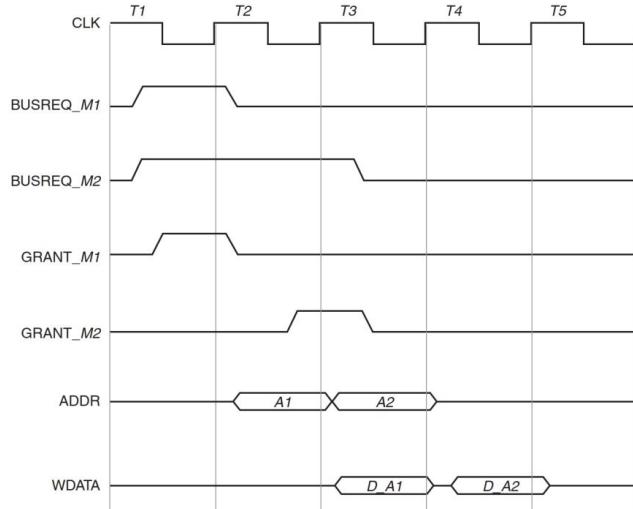
- Pipelined transfer
 - we wish to improve bus performance

- note here: 2 masters

- overlap address and data phase of multiple data transfers

- grant to m2 happens before transfer to m1 is finished

- possible when address/data buses are separate (that is, no time multiplex of address/data)



9

Bus

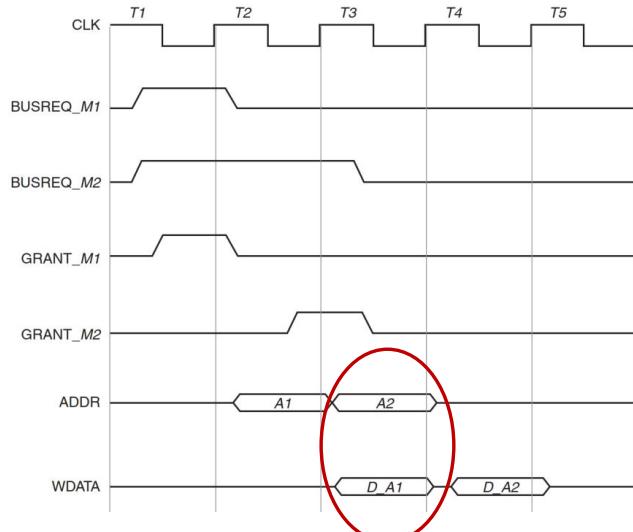
- Pipelined transfer
 - we wish to improve bus performance

- note here: 2 masters

- overlap address and data phase of multiple data transfers

- grant to m2 happens before transfer to m1 is finished

- possible when address/data buses are separate (that is, no time multiplex of address/data)

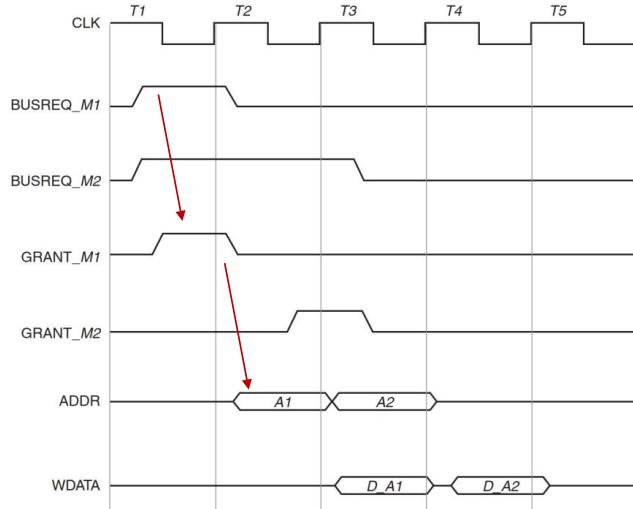


10

Bus

- Pipelined transfer
 - we wish to improve bus performance

- note here: 2 masters
- overlap address and data phase of multiple data transfers
- grant to m2 happens before transfer to m1 is finished
- possible when address/data buses are separate (that is, no time multiplex of address/data)

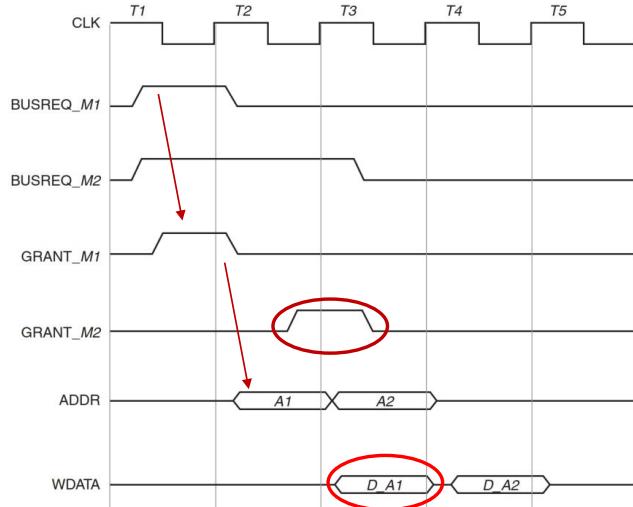


11

Bus

- Pipelined transfer
 - we wish to improve bus performance

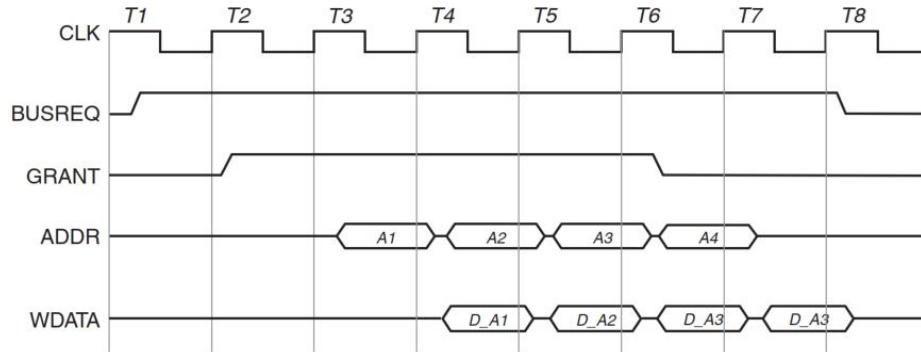
- note here: 2 masters
- overlap address and data phase of multiple data transfers
- grant to m2 happens before transfer to m1 is finished
- possible when address/data buses are separate (that is, no time multiplex of address/data)



12

Bus

- We saw burst transfer...
 - if multiple bursts from same master – reduced arbitration overhead
 - we can pipeline (overlap address/data) bursts



13

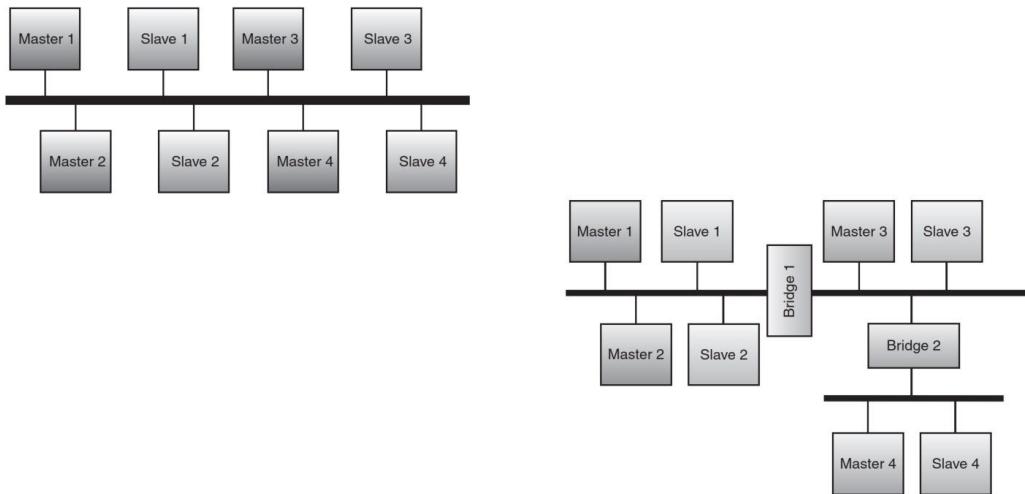
Bus

- Other types of transfer
 - **Split** – takes advantage of wait states to use for data transfer by other masters
 - Slave informs arbiter that split will take place (transfer too long)
 - **Out of order** – data transfer done in parts, but out of order
 - sometimes second transfer faster than first
 - each transfer with ID
 - Overhead: additional signals, complex interface and decoding logic
 - **Broadcast**
 - source transmits data on the bus, multiple components pick up

14

Bus

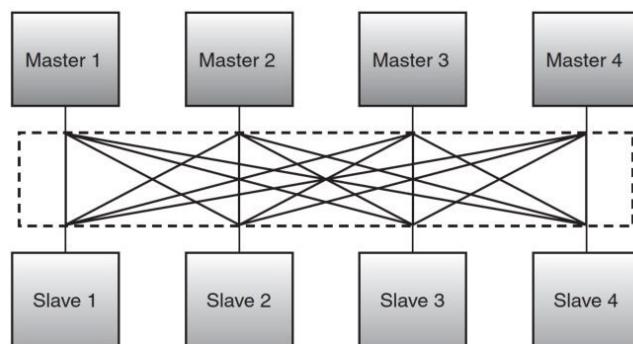
- Bus Topologies – have seen single bus and multilevel



15

Bus

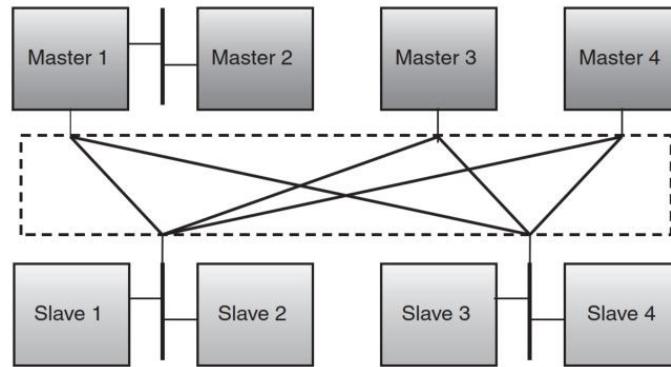
- Bus topologies
 - **Split bus** – multiple bus, but simplified tristate buffer inbetween
 - **Full bus matrix** – every master connected to every slave
 - improves parallelism, but requires separate arbitration for every slave



16

Bus

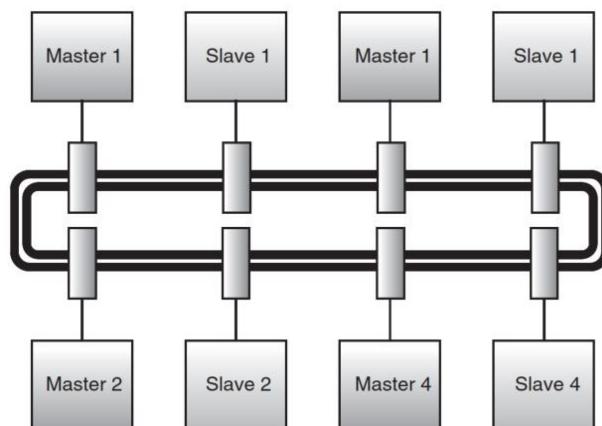
- Bus topologies
 - **Full bus matrix**: large area, increased power consumption, difficult routing
 - Alternative: **partial crossbar** topology
 - reduced parallelism compared to full matrix



17

Bus

- Bus Topologies
 - **Ring** – one or more concentric rings (token pass protocol)
 - availability and distance to destination determines direction
 - Token is circulated, receiving interface initiates transaction



18

Bus

- Motivation for **standard bus protocols**
 - Emergence of SoC (for the last 2 decades...)
 - integration of multiple processing cores and peripherals
 - development of IP cores
 - reuse (and cost reduction of costs)
- Many available SoC bus architectures, encouraged (free) by manufacturers to be used in chip design
- Will look at AMBA - ARM
 - Later will look at AVALON – Altera/Intel

19

Bus

- **AMBA** – advanced microcontroller bus architecture
 - On-chip Bus protocol specifications devised by ARM
 - Processors, memory interfaces and peripherals
 - Designed for on-chip communications
 - Removed overhead of circuit board level requirements (SoC now!)
 - **Open standard** available for chip design industry
 - very popular interface for embedded 32 bit processors
 - Main characteristics:
 - Synchronous operation – use only posedge
 - No on-chip bidirectional signals – no tristate buffers
 - Three distinct bus protocols

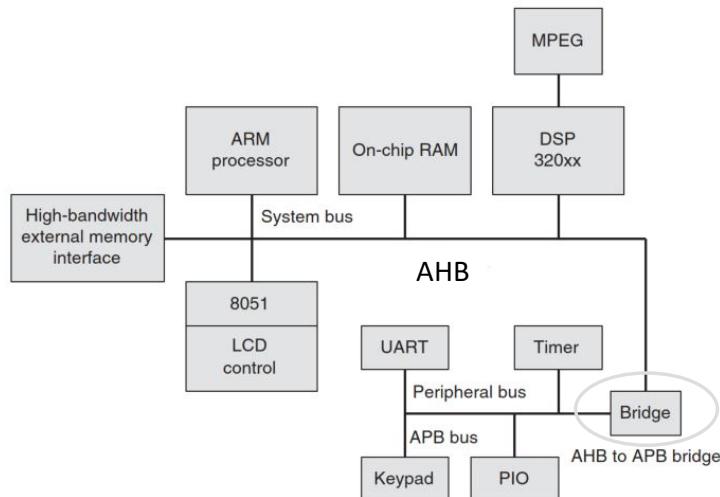
20

Bus - AMBA

- Most common AMBA bus protocols in uC include
 - AHB – advanced high-performance bus
 - high BW, lightweight, pipelined
 - majority of ARM Cortex-M, MIPS, AVR, TI-DSP...
 - APB – advanced peripheral bus
 - protocol for simple peripherals (timer, UART, etc), low BW
 - AXI – advanced extensible interface
 - **high performance protocol** for processors such as Cortex M7, Cortex-R and majority of Cortex-A
 - multiple data channels at high clock frequency
 - pipelined transfers
 - data security
 - AHB, APB – most commonly used in embedded design

21

Bus - AMBA



22

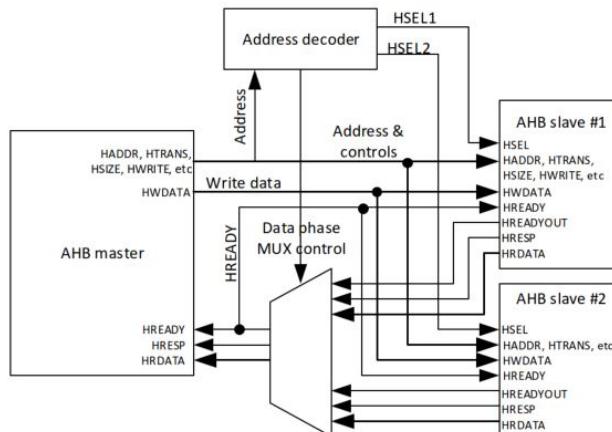
Bus - AMBA

- AHB – presently in version 5 (AHB5), still highly compatible with previous version
 - operates with common clock (HCLK)
 - all masters, slaves and bus infrastructure blocks in a segment
 - All registers on AHB trigger at rising edges of HCLK
 - Asynchronous (active low) reset (HRESETn)
 - For compatibility with older versions
 - Master – arbiter and arbiter-slave signals also present
 - A minimal AHB system (single Arm Cortex-M0 processor) can be created with 11 signals

23

Bus - AMBA

- AHB – Basic operations (**one bus master, two bus slaves**)

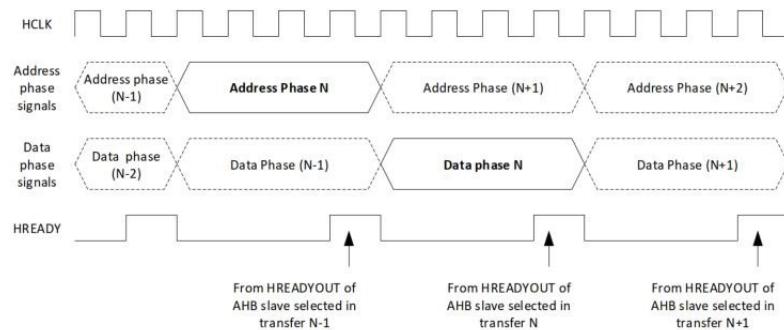


- if multiple bus masters sharing bus – master multiplexer controlled by bus arbiter

24

Bus - AMBA

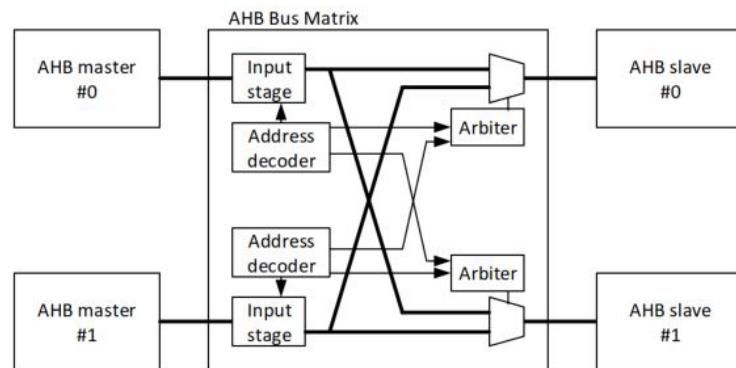
- AHB – transfers have **address phase** and **data phase**
 - signals grouped similarly
 - address phase: HADDR, HTRANS, HSEL, HWRITE, HSIZE
 - data phase: HWDATA, HRDATA, HRESP, HREADY, HREADYOUT
 - each phase terminated by HREADYOUT from the activated AHB slave



25

Bus - AMBA

- AHB – **multiple bus masters**
 - Latest : AHB Bus Matrix (configurable)



- note: **each master has an arbiter** to resolve conflict
- matrix → enhanced BW

26

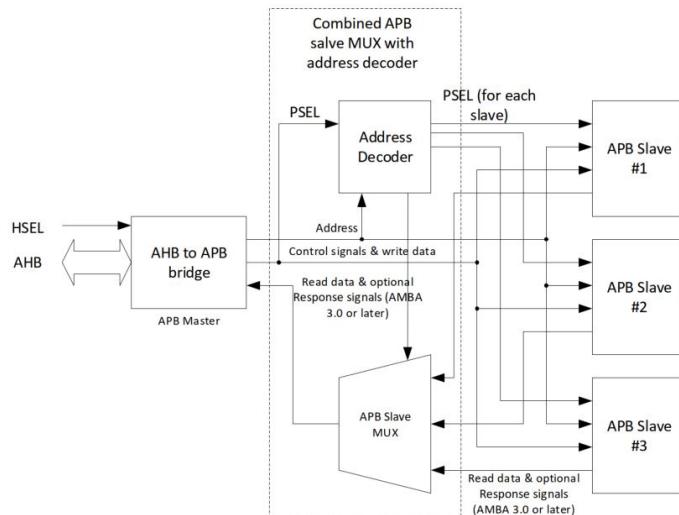
Bus - AMBA

- APB – mainly targeted for connections with peripherals
 - common to use 32-bit peripheral bus
 - Why connect to APB?
 - If number of peripherals large, connecting all to AHB may reduce max frequency (signal fan out, more complex decoding logic). Connecting to APB mitigates that
 - Peripheral subsystem can be run at different clock, and be powered down without affecting AHB
 - APB interfaces use a simpler protocol
 - Since APB transfers are not pipelined, peripherals designed for traditional processors can be easily connected to it.
 - Bus master to APB usually AHB-to-APB bridge
 - Common clock PCLK (can be same or different than AHB - HCLK)
 - All registers on APB trigger on rising edge of PCLK, Asynch reset

27

Bus - AMBA

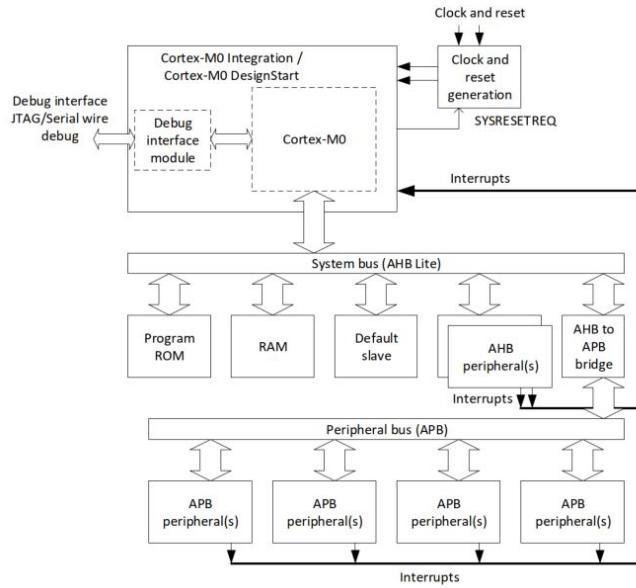
- APB subsystem



28

Bus

- Cortex-M0 -- one AHB



29

Bus

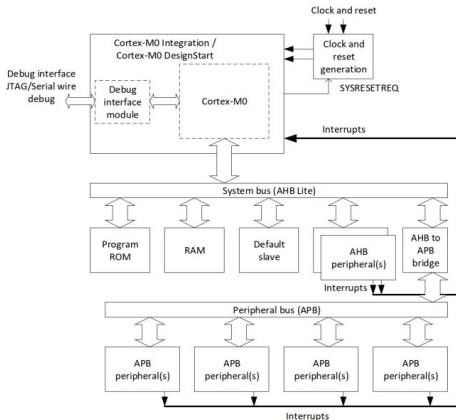
- Cortex-M0 -- one AHB

ROM – embedded flash (along with memory controller \$\$), program image placed at 0x00000000

RAM – address 0x20000000 (start of SRAM)

Peripherals on APB run at lower clock (design for AHB or APB accordingly)

AHB and APB peripherals placed between 0x40000000 and 0xFFFFFFFF (it's your design!)



30

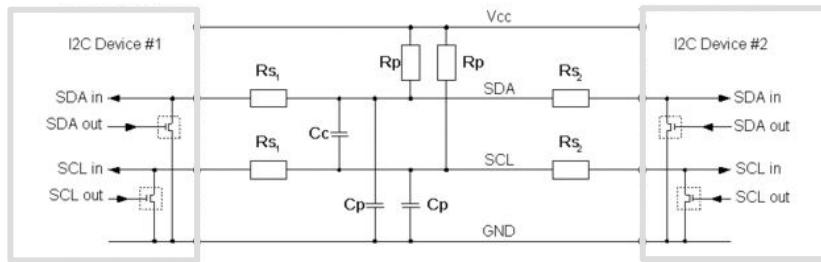
Recap

- Bus master – initiates communication
 - Use specific servant addresses and control signals
 - Controls the flow of data directly
 - Decoder – decodes address of target/peripheral/slave
 - Arbitration – grants bus ownership
 - Protocol – determines type; order of data; acknowledgement; end of transmission; how arbitration is resolved
 - Bridge
 - Conversion between two buses (with different protocols)
 - Keep traffic within bus segments when buses operating independently
 - Buffer between master on one bus and servant on the other
-

Peripherals – I2C

- I2C – Inter-integrated circuit
 - Serial bus protocol
 - Multiple masters, multiple slaves
 - Uses two wires
 - Serial Data line (SDA)
 - Serial Clock line (SCL)
 - Simple, reduced number of pins
 - Speed
 - 100Kbps standard
 - 400Kbps fast
 - 3.4Mbps high speed
-

Peripherals – I2C



Note: R_p is a pull-up resistor (I2C “termination”)

Size of pull-up resistor is a function of line capacitance

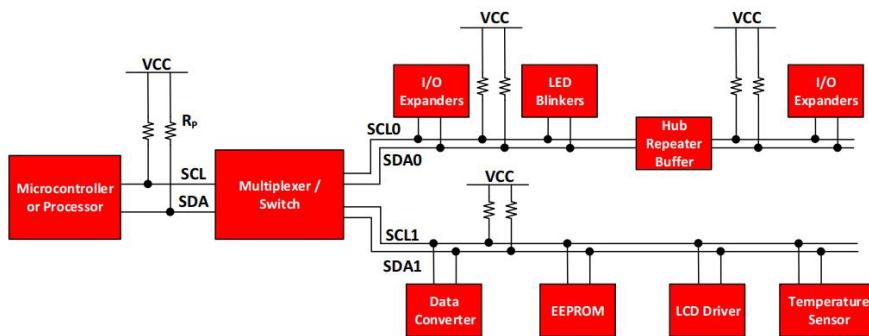
C_p is wire capacitance

C_c is cross-channel capacitance

Vcc is 1.2V to 5.5V

Peripherals – I2C

- Example of configuration



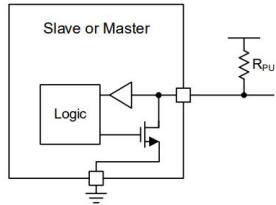
For this case, microcontroller is the master, with multiple slaves, all connected through only two pins

Peripherals – I2C

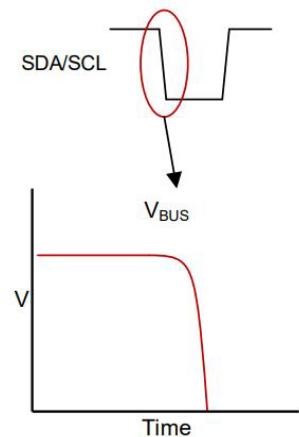
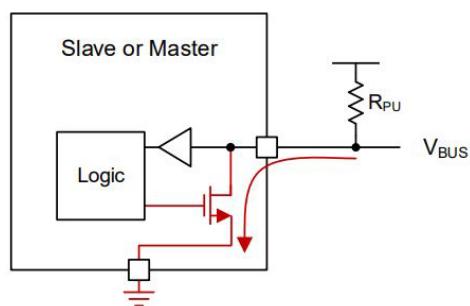
- Both lines use “open drain”
 - Can pull the bus down (to ground) or
 - When bus is in use
 - Can pull the bus up via a pull-up resistor
 - When the bus is released by a master or a slave
- No device can hold the bus at high level
- When one device sees that the line is low, it will not transmit
 - Another device is using the bus

Peripherals – I2C

- Transmission: Pull Down / Pull Low

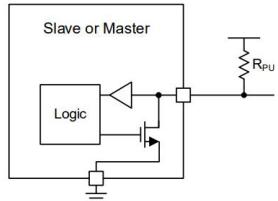


Want to transmit logic low

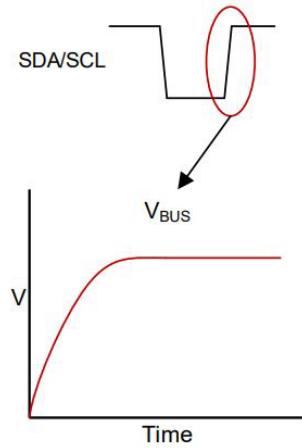
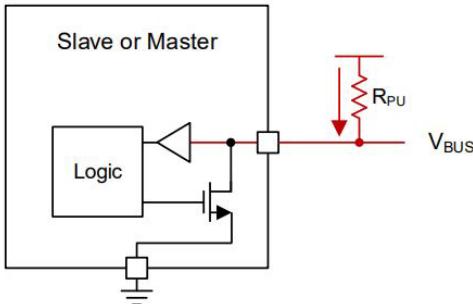


Peripherals – I2C

- Transmission: Pull High



Want to transmit logic high



Peripherals – I2C

- It's bidirectional
 - Master/controller communicates with slaves
 - Slaves are on the same bus
 - Each has a unique address
 - May not transmit unless master uses their address
 - Devices can have multiple registers for data storage (can be read/written)
 - May need to be configured at startup
 - If no transmission, line is high
 - Data transfer may only be initiated when bus idle
 - BOTH SDA and SCL are high → bus is idle

Peripherals – I2C

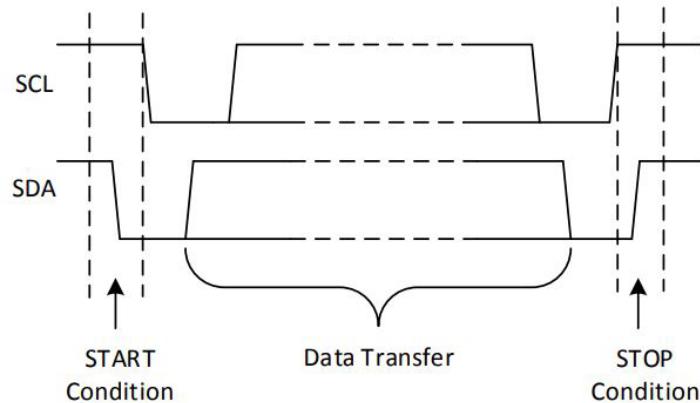
- Master writes to slave
 - Master/TX sends start condition and address to slave/RX
 - Master/TX sends data to slave/RX
 - Master/TX terminates transfer with STOP condition
-

Peripherals – I2C

- Master reads from slave (or receives from)
 - Master/RX sends a START condition and address of slave/TX
 - Master/RX sends to slave/TX the requested register from which it wishes to read
 - Master/RX receives the data from slave/TX
 - Master/RX terminates transfer with STOP condition
-

Peripherals – I2C

- Transfer is initiated with **master sending** START and ends with **master sending** STOP
 - High-to-low on SDA while SCL is high = START
 - Low-to-high on SDA while SCL is high = STOP



Peripherals – I2C

- In multi-master situation
 - Master transmitting does not want to lose control of the bus and have its operation interrupted
 - Possible to send multiple start conditions
 - Stop condition always ends transmission

start-address-write-data-start-address-read-data-stop

(we're missing a detail here, but it will come shortly)

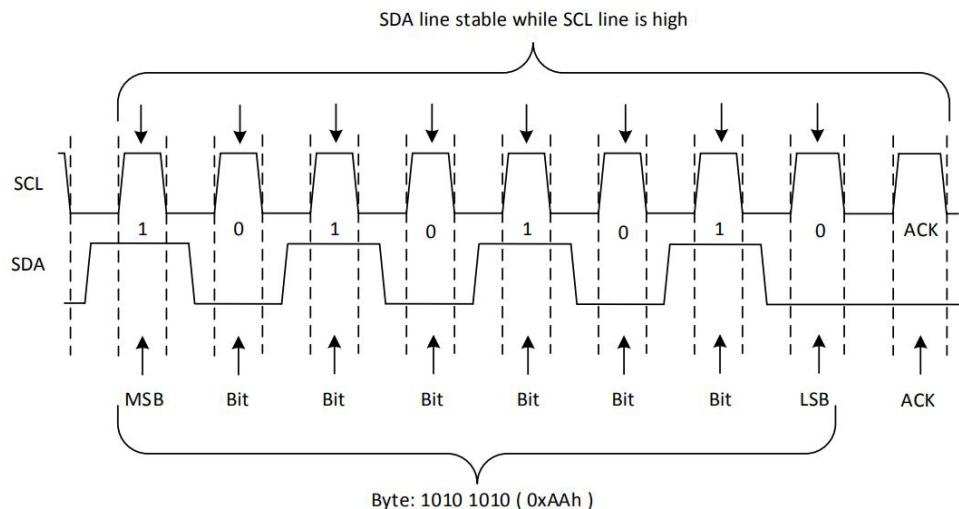
Peripherals – I2C

- Transmission
 - One data bit transmitted each pulse of SCL
 - One byte on SDA line
 - Device address
 - Register address
 - Data written to slave
 - Data read from slave
 - Data is transmitted MSB first
 - Any number of bytes can be transmitted between Start and Stop
 - During high phase of clock period, data on SDA must remain stable (otherwise, it will be START/STOP!)
-

Peripherals – I2C

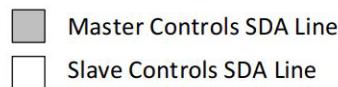
- Acknowledge and Not Acknowledge
 - Each byte is followed by ACK from RX
 - Byte: data or address
 - Receiver indicates it received info okay
 - Before RX sends ACK, TX must release line
 - RX pulls low the SDA line
 - NACK if
 - RX not ready to start comm, busy with other task
 - RX cannot receive any more bytes
 - Master is done reading – sends NACK to slave
 - Then STOP
-

Peripherals – I2C

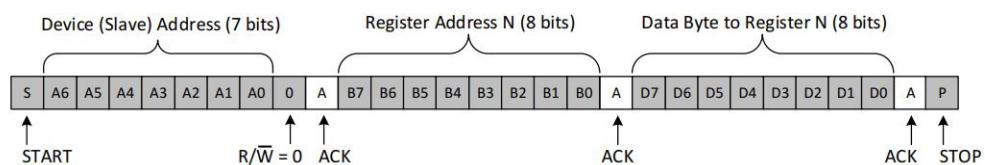


Peripherals – I2C

- Writing to slave



Write to One Register in a Device



Note: presence of write bit
presence of acknowledge

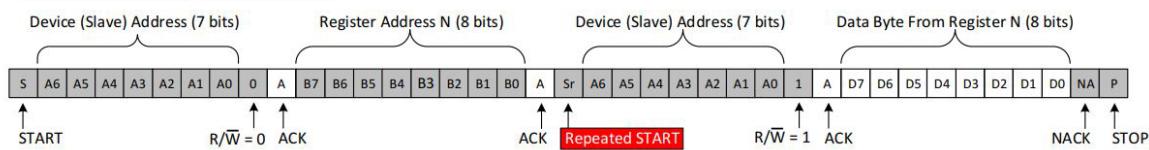
Peripherals – I2C

- Read from a register on a slave device

 Master Controls SDA Line

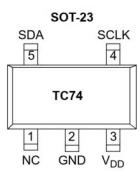
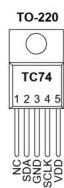
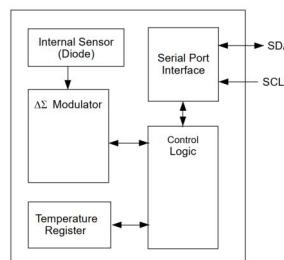
 Slave Controls SDA Line

Read From One Register in a Device



Peripherals – I2C

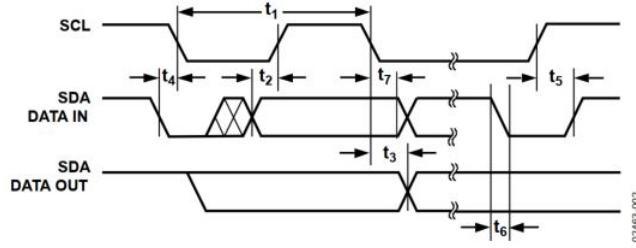
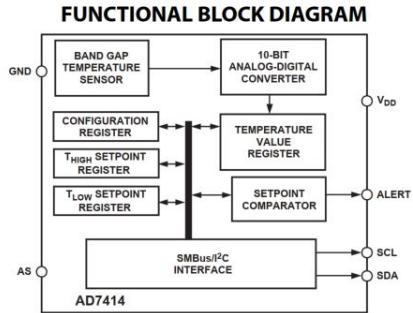
- MCU may have a number of i2c modules available
- Examples: TC74 (sensor)
 - 8 bit (+/- 1deg)
 - 8 samples/sec



Pin No. (5-Pin SOT-23)	Pin No. (5-Pin TO-220)	Symbol	Type	Description
1	1	NC	None	No Internal Connection
2	3	GND	Power	System Ground
3	5	V _{DD}	Power	Power Supply Input
4	4	SCLK	Input	SMBus/I ² C Serial Clock
5	2	SDA	Bidirectional	SMBus/I ² C Serial Data

Peripherals – I2C

- Examples: AD7414 (temperature sensor)
 - 10 bit, +/- 0.5 deg accurate, with alert



02463-002

Serial Clock Period, t_1	2.5	μs min
Data In Setup Time to SCL High, t_2	50	ns min
Data Out Stable after SCL Low, t_3	0	ns min
SDA Low Setup Time to SCL Low (Start Condition), t_4	50	ns min
SDA High Hold Time after SCL High (Stop Condition), t_5	50	ns min
SDA and SCL Fall Time, t_6	90	ns max
Data Hold Time, t_7	35	ns min

ECE342 – Computer Hardware

Lecture 13

Bruno Korst, P.Eng.

Agenda

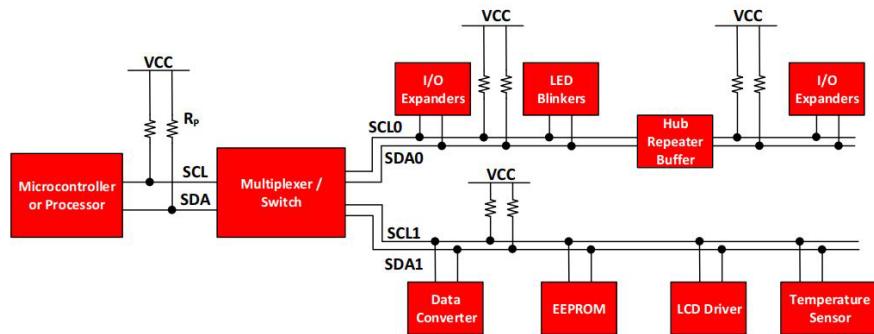
- Housekeeping
- I2C

Housekeeping

- Need to find a date for midterm
- Some groups have taken two RTC modules out – please return
- Lab: please prepare ahead of time
 - Fixed point particularly problematic...

Peripherals – I2C

- Example of configuration



For this case, microcontroller is the master, with multiple slaves, all connected through only two pins
- Two wires plus pull-up resistors

Peripherals – I2C

- Two lines: Serial Data Line (SDA), Serial Clock Line (SCL)
 - Less pins, easier to interface
 - Both lines use “open drain” (both pulled up to positive V)
 - Bus is pulled down (“to ground”) when bus is in use
 - When bus is released by a master or a slave, it is pulled back up via pull-up resistor
 - No device can hold the bus at high level
 - When one device sees that the line is low, it will not transmit
 - Another device is using the bus
-

Peripherals – I2C

- Data transfer
 - 100Kb/s in standard mode
 - 400Kb/s in fast mode
 - 3.4Mb/s in high-speed mode
 - Each device has a unique address (7 bits, 10 bits or 16 bits)
 - Depending on function can operate as TX, RX or both
 - Temperature sensor: TX
 - Display: RX
 - Memory: both TX and RX
-

Peripherals – I2C

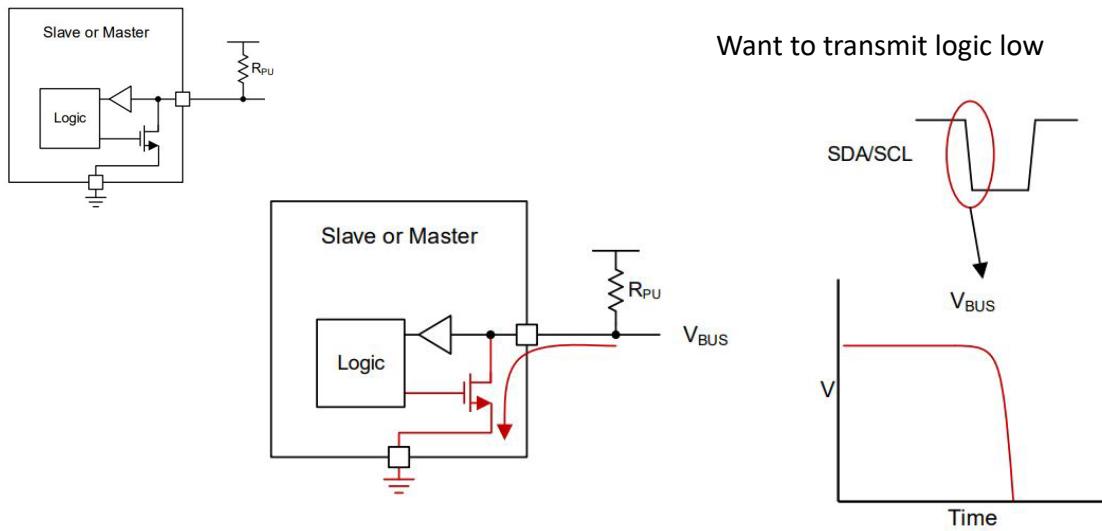
- Multiple masters can coexist on the same I2C bus
 - When multiple masters want to control the bus, clock synchronization and arbitration are needed
 - Clock synchronization – one master pulls SCL low, no other can pull it high
 - Arbitration
 - On the SCL rising edge, each master checks whether the SDA voltage level matches what it has sent
 - Whenever a master tries to transmit a high but detects a low on SDA, this master loses the arbitration
 - Each losing master immediately switches to slave receive mode, in case the winning master tries to address it
 - A losing master will restart the transfer after it detects a STOP bit.

Peripherals – I2C

- What limits the number of devices that can be connected to the bus is bus capacitance
 - That will also determine the value of the pull up resistor.
 - Recommended resistor value is 3Kohm for standard speed

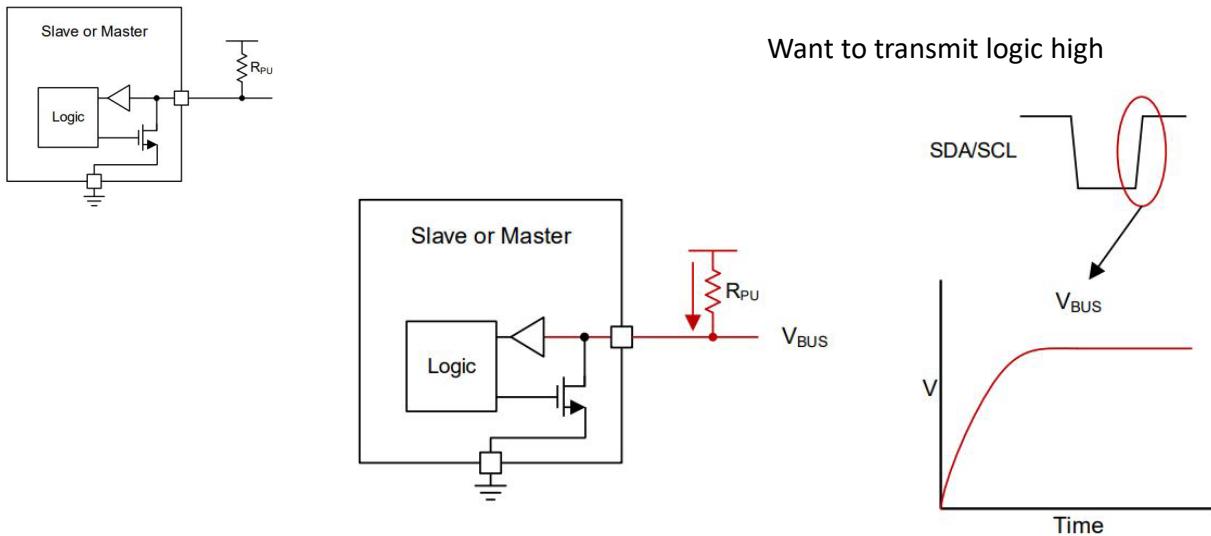
Peripherals – I2C

- Transmission: Pull Down / Pull Low



Peripherals – I2C

- Transmission: Pull High



Peripherals – I2C

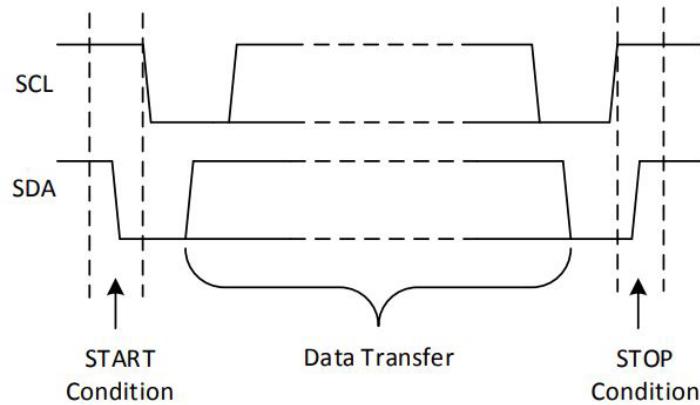
- It's bidirectional
 - Master/controller communicates with slaves
 - Slaves are on the same bus
 - Each has a **unique address**
 - May not transmit unless master uses their address
 - Devices can have multiple registers for data storage (can be read/written)
 - May need to be configured at startup
 - If no transmission, line is high
 - Data transfer may only be initiated when bus idle
 - BOTH SDA and SCL are high → bus is idle

Peripherals – I2C

- In general lines: Master **writes to slave**
 - Master/TX sends START condition and address to slave/RX
 - Master/TX sends data to slave/RX
 - Master/TX terminates transfer with STOP condition
- In general lines: Master **reads from slave** (or receives from)
 - Master/RX sends a START condition and address of slave/TX
 - Master/RX receives the data from slave/TX
 - Master/RX terminates transfer with STOP condition

Peripherals – I2C

- Transfer is initiated with **master sending** START and ends with **master sending** STOP
 - High-to-low on SDA while SCL is high = START
 - Low-to-high on SDA while SCL is high = STOP



Peripherals – I2C

- Master transmitting does not want to lose control of the bus and have its operation interrupted
 - Possible to send multiple start conditions
 - Stop condition always ends transmission

start-address-write-data-start-address-read-data-stop

(we're missing a detail here, but it will come shortly)

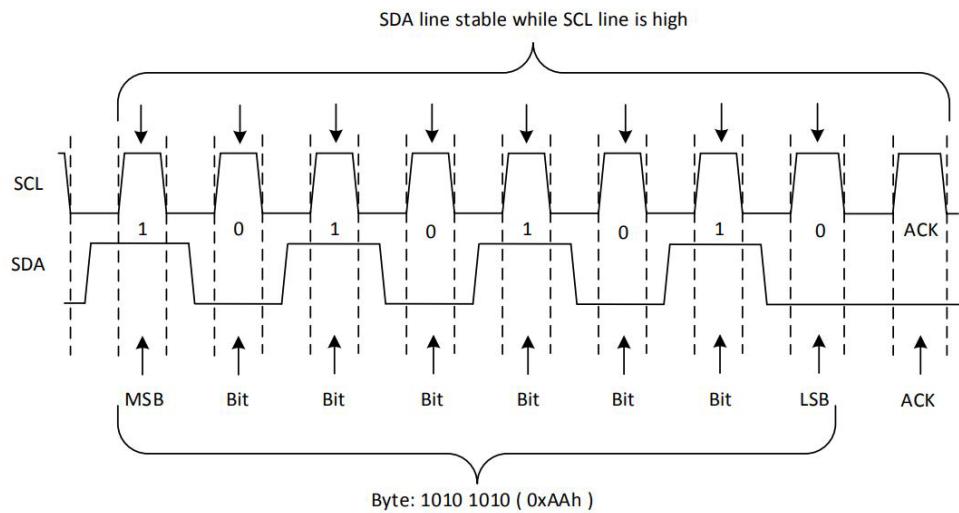
Peripherals – I2C

- **Transmission**
 - One data bit transmitted each pulse of SCL
 - One byte on SDA line
 - Device address
 - Register address
 - Data written to slave
 - Data read from slave
 - Data is transmitted MSB first
 - Any number of bytes can be transmitted between Start and Stop
 - During high phase of clock period, data on SDA must remain stable (otherwise, it will be START/STOP!)
-

Peripherals – I2C

- Acknowledge and Not Acknowledge
 - Each byte is followed by ACK from RX
 - Byte: data or address
 - Receiver indicates it received info okay
 - Before RX sends ACK, TX must release line
 - RX pulls low the SDA line
 - NACK if
 - RX not ready to start comm, busy with other task
 - RX cannot receive any more bytes
 - Master is done reading – sends NACK to slave
 - Then STOP
-

Peripherals – I2C (Data Frame)



Peripherals – I2C

- Data frame – 7 bit address slave (two bytes)

S	Slave Address	R/W	A	Data	A	Data	A	P
1 bit	7 bits	1 bit	1 bit	8 bits	1 bit	8 bits	1 bit	1 bit

- Master begins by sending start bit, slave address and single direction bit
 - Read (1) – master requests to receive data from slave
- Slave whose address matches, answers with ACK bit
- Data transfer takes place (byte by byte). Each byte is followed by ACK or NACK
 - Best case scenario: 9 cycles to transfer 1 byte
- Master completes comm by sending stop bit

Peripherals – I2C

- Data frame – 10 bit address slave (two bytes)

S	Slave Address (higher 2 bits)	R/W	A1	Slave Address (lower 8 bits)	A2	Data	A	Data	A	P
1 bit	7 bits (11110xx)	1 bit (0)	1 bit	8 bits	1 bit	8 bits	1 bit	8 bits	1 bit	1 bit

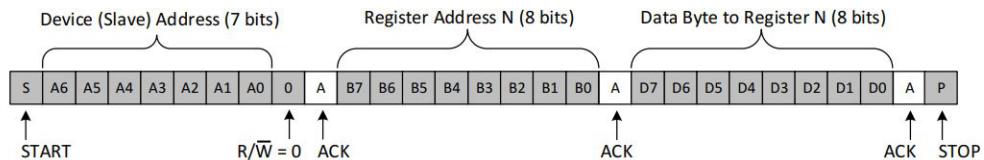
- Master begins by sending start bit, after start bit, master sends a five bit constant
 - Followed by the two MSB of the slave address and a single direction bit (W)
 - It could be that multiple ACK are received in A1
- Master sends the lower 8 bits of the address (only one ACK in A2)
- Data transfer takes place (byte by byte). Each byte is followed by ACK or NACK
 - Best case scenario: 9 cycles to transfer 1 byte
- Master completes comm by sending stop bit

Peripherals – I2C

- Writing to slave register

 Master Controls SDA Line
 Slave Controls SDA Line

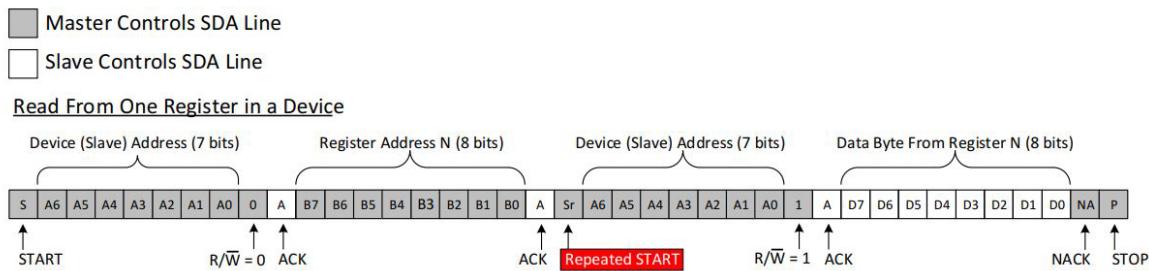
Write to One Register in a Device



Note: presence of write bit
presence of acknowledge

Peripherals – I2C

- Read from a register on a slave device



Peripherals – I2C

- Timing parameters to be met in the I2C standard

Timing Parameters	Standard		Fast		High-speed		Unit
	Min	Max	Min	Max	Min	Max	
SCL clock frequency	0	100	0	400	0	1000	kHz
Rise time of SDA & SCL	-	1000	20	300	-	120	µs
Fall time of SDA & SCL	-	300	20	300	-	120	µs
Low time of SCL	4.7	-	1.3	-	0.50	-	µs
High time of SCL	4.0	-	0.6	-	0.25	-	µs
Data hold time	5.0	-					µs
Data setup time	250	-	100	-	50	-	µs

- Program appropriate data field of TIMINGR register
 - Clock prescaler, high and low period counter, setup and hold time counter

Peripherals – I2C

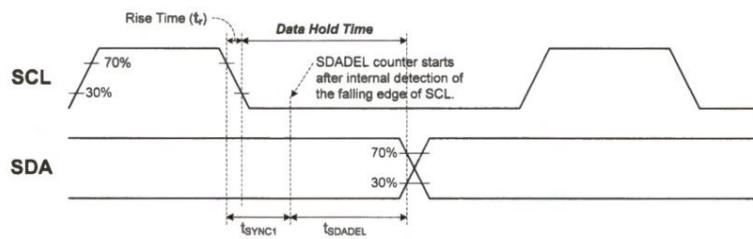
- Timing: Rise and Fall time
 - Time from low to high (Rise) or from high to low (Fall)
 - go from 30% to 70% of final value or vice-versa
 - Fall time typically shorter
 - On circuit boards capacitance formed by traces, plates, wires, pins, connections is known as parasitic capacitance
 - Impacts high-frequency signals
 - Rise time is determined by the value of the pull up resistor and the bus capacitance
-

Peripherals – I2C

- Timing: Rise and Fall time
 - Time from low to high (Rise) or from high to low (Fall)
 - go from 30% to 70% of final value or vice-versa
 - Fall time typically shorter
 - On circuit boards capacitance formed by traces, plates, wires, pins, connections is known as parasitic capacitance
 - Impacts high-frequency signals
 - Rise time is determined by the value of the pull up resistor and the bus capacitance
-

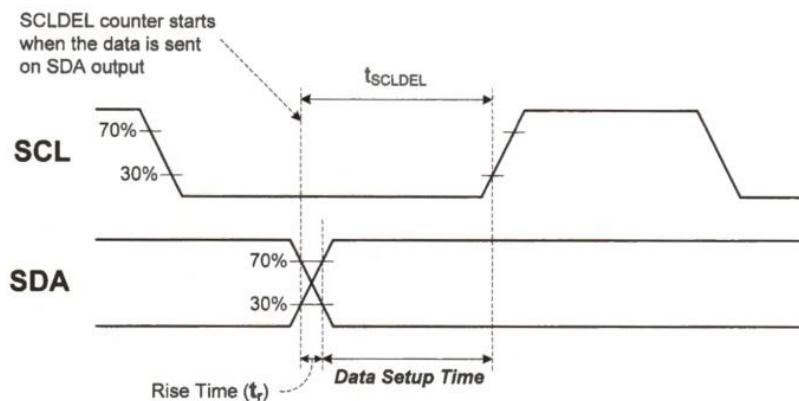
Peripherals – I2C

- Data hold time
 - When falling edge of SCL is detected, a delay is inserted before the data is sent out.
 - Detector of falling edge takes t_{sync} to detect
 - Data hold time = $t_{sync} + t_{inserted_delay} - \text{rise_time}$



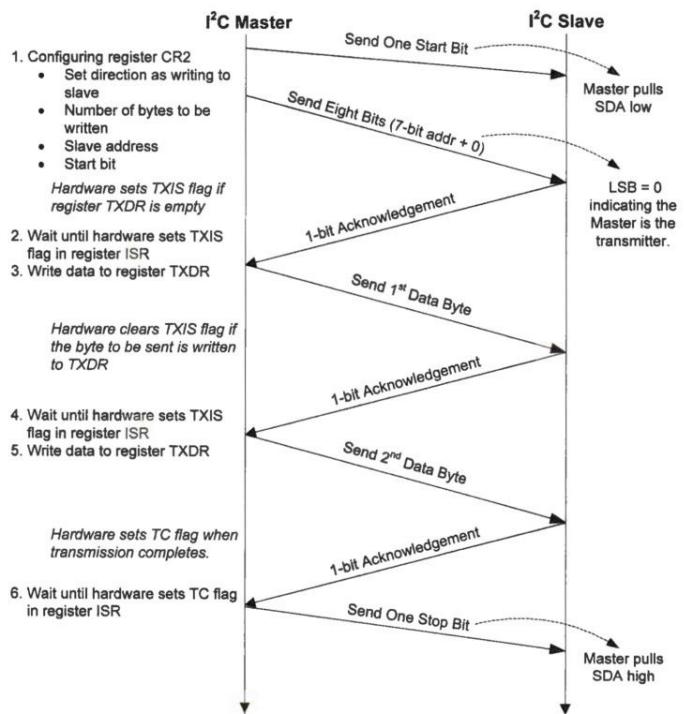
Peripherals – I2C

- Data set up time
 - Amount of time SCL is held low after a data bit has been placed on SDA
 - SDA must remain at same level to be properly sampled



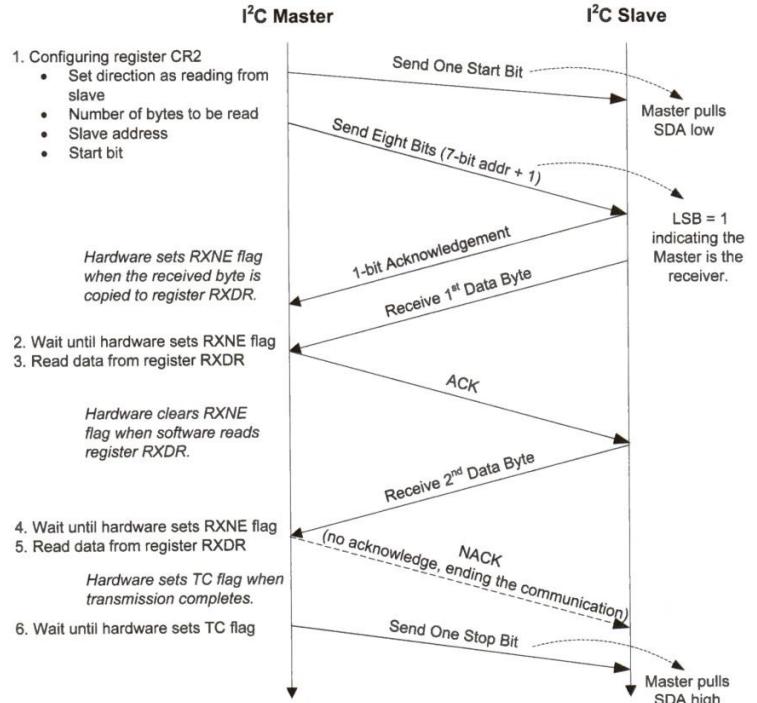
Peripherals – I2C

- Sending data to I2C via Polling



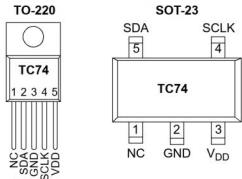
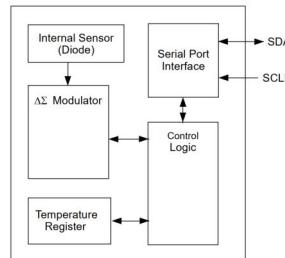
Peripherals – I2C

- Receiving data from I2C via polling



Peripherals – I2C

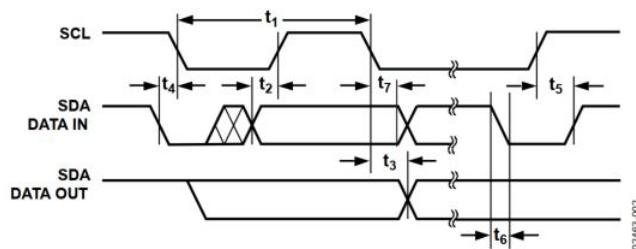
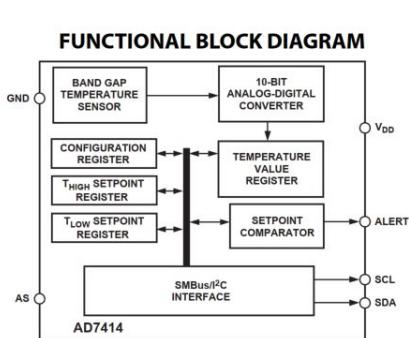
- MCU may have a number of i2c modules available
- Examples: TC74 (sensor)
 - 8 bit (+/- 1deg)
 - 8 samples/sec



Pin No. (5-Pin SOT-23)	Pin No. (5-Pin TO-220)	Symbol	Type	Description
1	1	NC	None	No Internal Connection
2	3	GND	Power	System Ground
3	5	V _{DD}	Power	Power Supply Input
4	4	SCLK	Input	SMBus/I ² C Serial Clock
5	2	SDA	Bidirectional	SMBus/I ² C Serial Data

Peripherals – I2C

- Examples: AD7414 (temperature sensor)
 - 10 bit, +/- 0.5 deg accurate, with alert



Serial Clock Period, t ₁	2.5	μs min
Data In Setup Time to SCL High, t ₂	50	ns min
Data Out Stable after SCL Low, t ₃	0	ns min
SDA Low Setup Time to SCL Low (Start Condition), t ₄	50	ns min
SDA High Hold Time after SCL High (Stop Condition), t ₅	50	ns min
SDA and SCL Fall Time, t ₆	90	ns max
Data Hold Time, t ₇	35	ns min

ECE342 – Computer Hardware

Lecture 14

Bruno Korst, P.Eng.

Agenda

- I2C
 - Examples
- SPI
 - Examples

Peripherals - I2C

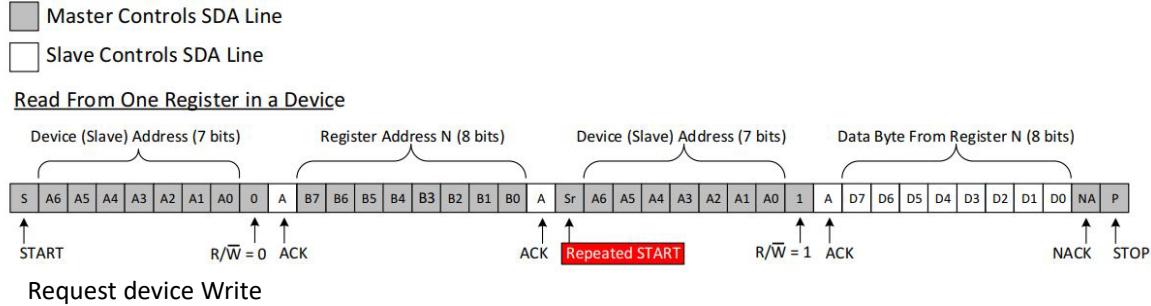
- We saw the I2C data frame
 - Start – address – r/w – ack – data – ack – stop
 - Transmission done byte by byte, with acknowledgement
- May have repeated starts – after ack, if data transmission is not done
- May have NACK – when message received in error or line held
 - Repeat transmission
- Start – SDA transitions high → low WHILE SCL is held high
- Stop – SDA transitions low → high WHILE SCL is held high
 - Note: in transmission, SDA can only transition when SCL is LOW
 - Otherwise, it is interpreted as start/stop

Peripherals - I2C

- The master is always the one driving the clock line SCL
- In case there are multiple masters, there is arbitration
 - While SCL=1, if a master sends a 1 on the SDA and receives a 0, has lost the arbitration
 - The losing master will bring down its data line and will keep trying to gain control of the bus
- Address can be 7, 10 or 16 bits long, transmitted in 1, 2 and 2 bytes resp.
 - For 10 bits, a preamble is added to the MSB 3 bits of the address
- Slave can slow transmission between bytes by holding the SCL low and forcing the master to be on “wait” condition

Peripherals – I2C

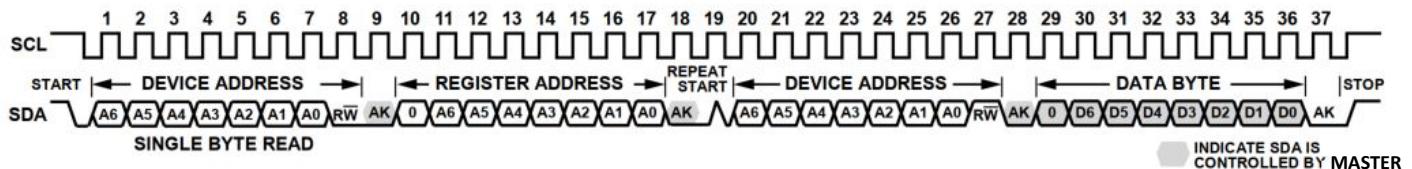
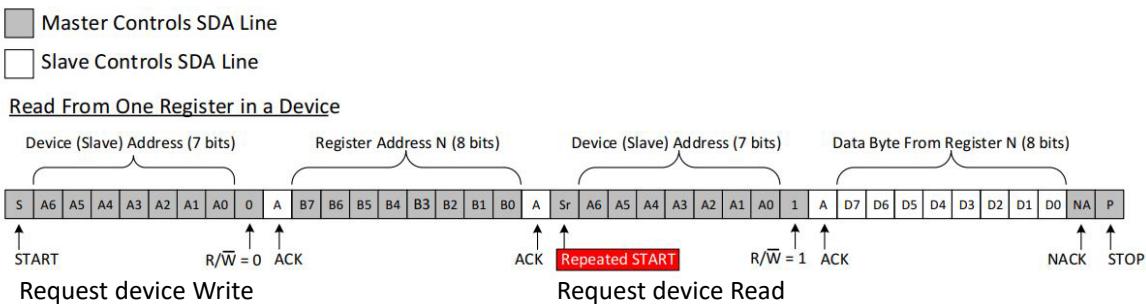
- If after slave ends transmission the master does not acknowledge reception, transmission is considered stopped, and a new start can be expected
- Example
 - Read from a register on a slave device
 - One address for the device, one address for a register within a device



Peripherals - I2C

- Describing the example:
 - Master reading from a register within a slave on the I2C bus
 - Master STARTs tx with address (7bit) + R/W bit indicating a WRITE
 - Slave acknowledges
 - Master sends the address of the register within that slave (8 bits)
 - Slave acknowledges
 - Master sends **another START**, the slave address and R/W indicating READ
 - Slave acknowledges, master releases SDA and keeps providing clock
 - Master turns into receiver, and slave transmits
 - At end of every byte, master sends acknowledgement
 - When all expected bytes are received, Master sends NACK (cause slave halt)
 - Master issues a stop condition.

Peripherals – I2C



Peripherals - I2C

- The I2C peripheral is initialized via drivers provided through HAL (Hardware Abstraction Layer)
 - `Stm32f4xx_hal_i2c.c` (and `.h`) or, you are given `config.c`
- HAL
 - An upgrade from a Standard Peripheral Library
 - Simplifies porting code to different STM32 sub-families (F0, F1...)
 - Specific peripheral mapping is done through handlers (structs) which map to specific (real) addresses

Peripherals - I2C

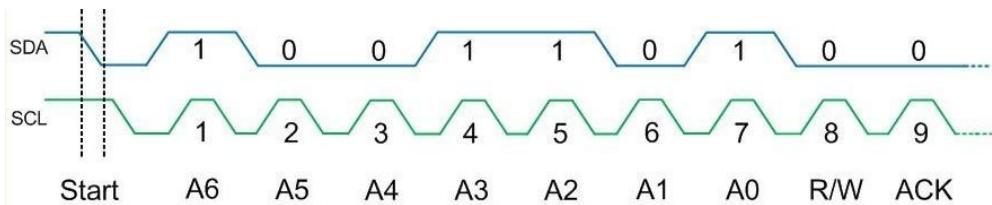
- Example: I2C_HandleTypeDef (a struct in stm32f4xx_hal_i2c.h)
 - Initialization of I2C (hi2c1 defined as I2C_HandleTypeDef in config.h)
 - Clock frequency
 - Duty cycle of the clock (2:1 or 16:9)
 - Own address (the slave address)
 - Addressing mode (7 or 10 bits)
 - Dual addressing mode (in case two addresses are specified)
 - Own address 2 (second address of the slave)
 - General call mode (enable/disable general call – seven zeros + r/w=0)
 - No stretch mode (whether will allow stretching)

Peripherals - I2C

- You will have functions for master/slave comms via I2C
 - Ex: HAL_I2C_
 - Master_Transmit
 - Master_Receive
 - Slave_Transmit
 - Slave_Receive
 - Mem_Write
 - Mem_Read
 - Hal_I2C_IsDeviceReady, etc.

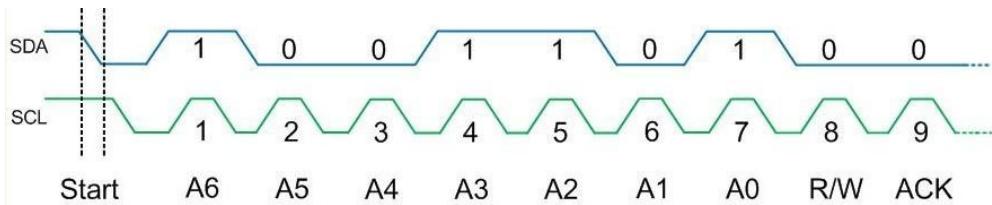
Peripherals - I2C

- The following is an example of



Peripherals - I2C

- The following is an example of



- Master initiating a write to address 1001101
- What if the address was 0000000?
 - Useful when master wants to send (tx, no rx!) the same data byte to all slaves

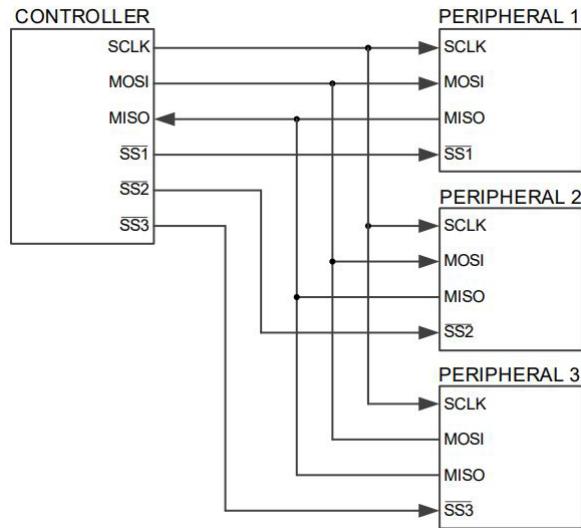
Peripherals - I2C

- Describe how a master writes data value 11110000 to a slave with address 1001 101
 - Master puts high to low transition on SDA while SCL is high, generate START condition
 - Master transmit 1001101-0 on the bus (master will write the next byte to slave)
 - Slave pulls SDA line low at the 9th clock pulse to signal ACK – say it is ready to receive data
 - After receiving ACK, master will transmit data byte (11110000) on the SDA line, MSB first
 - After slave receives data, will leave SDA line high to signal NACK and inform master that no more data is needed
 - After NACK received, master changes SDA from low to high – STOP condition

Peripherals - SPI

- SPI – Serial Peripheral Interface
 - Synchronous, low power, high throughput
 - Does not support multiple masters
 - Slaves cannot start communication
 - 4 Wires
 - Master In, Slave Out (MISO) – data line
 - Master Out, Slave In (MOSI) – data line
 - Serial clock line (SCLK)
 - Slave select (SS) (active low – SS “bar”)

Peripherals - SPI



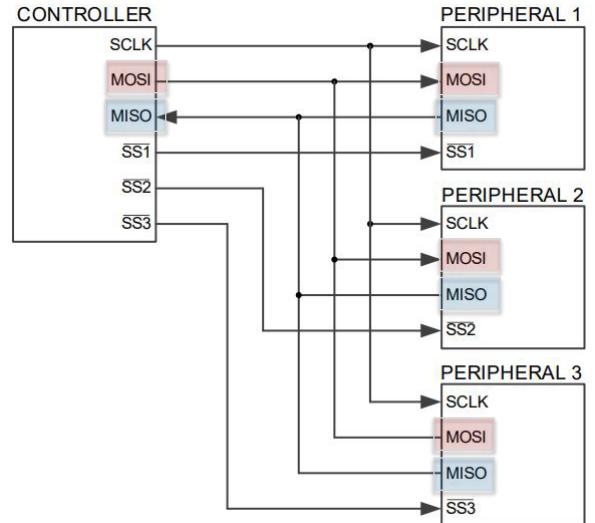
- Note direction of communication (out to in)
- Each peripheral has a peripheral select (SS)
 - Also labelled CS-bar, SYNC-bar, SS#, etc.

Peripherals - SPI

- Serial clock – SCLK
 - Synchronizes data transmission between master/controller and slave/peripheral
 - Originates from master/controller
 - Connected to **ALL slaves**
 - May idle high or low
 - Data is clocked in on rising or falling edge of clock
 - Depends on SPI Mode configuration

Peripherals - SPI

- **MOSI** – master out, slave in
 - Used to send data to the peripheral device
 - Can be shared between peripheral devices
 - Commonly labelled SIMO as well
 - At slave/peripheral – SDI/DI/DIN/SI
 - At master/controller – SDO/DO/DOUT/SO
- **MISO** – master in, slave out
 - Used to send data to the controller
 - Slave out: high Z when SS not selected



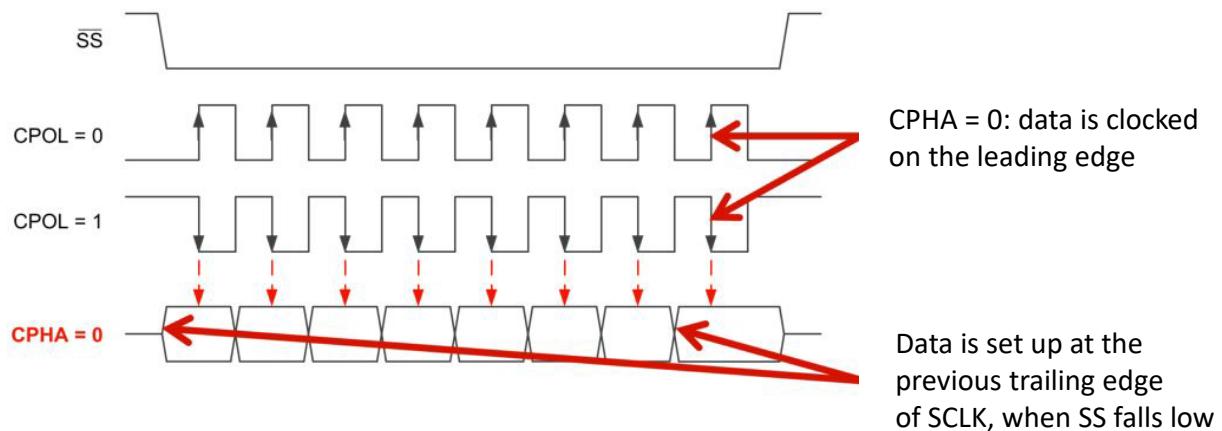
Peripherals - SPI

- Clock polarity (CPOL)
 - “leading edge” = first edge of the pulse
 - “trailing edge” = second edge of the pulse
- If SCLK idles low (CPOL=0)
 - Leading edge is RISING edge
 - Trailing edge is FALLING edge
- If SCLK idles high (CPOL=1) – reverse

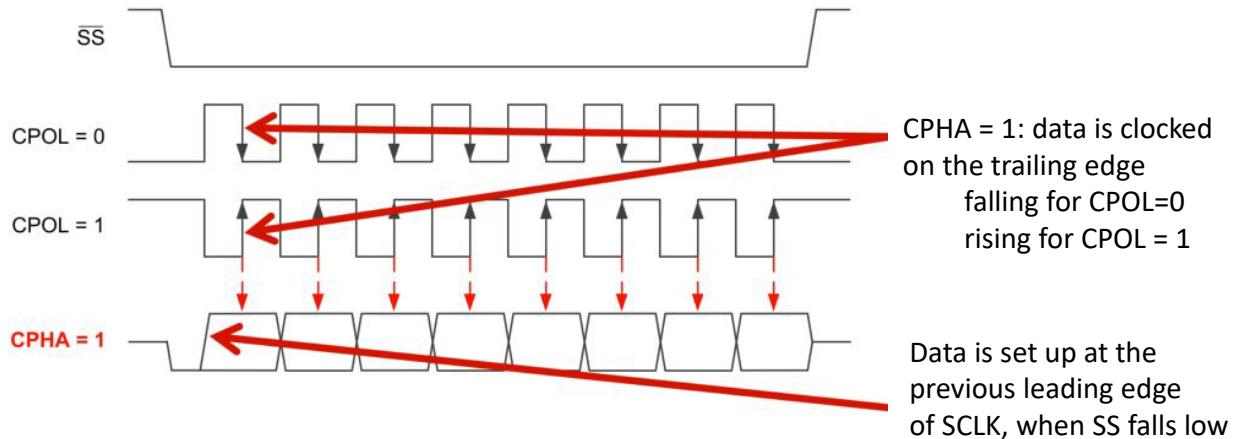
Peripherals - SPI

- Clock Phase (CPHA)
 - Data is set up when SS goes low
 - CPHA = 0 – data is clocked on the leading edge
 - CPHA = 1 – data is clocked on the trailing edge
- This allows for 4 SPI Mode settings
 - Mode 0 – CPOL = 0, CPHA = 0
 - Mode 1 – CPOL = 0, CPHA = 1
 - Mode 2 – CPOL = 1, CPHA = 0
 - Mode 3 – CPOL = 1, CPHA = 1

Peripherals - SPI

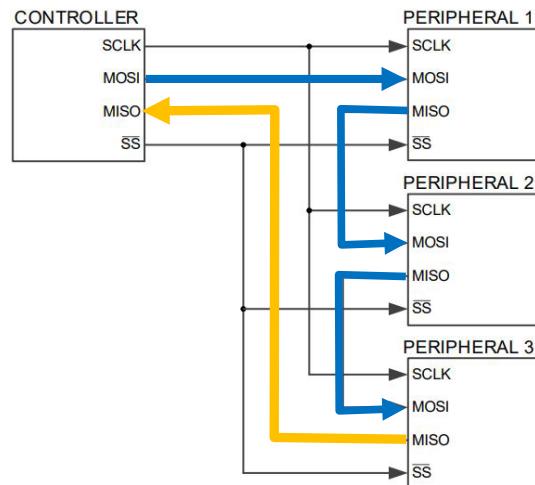


Peripherals - SPI



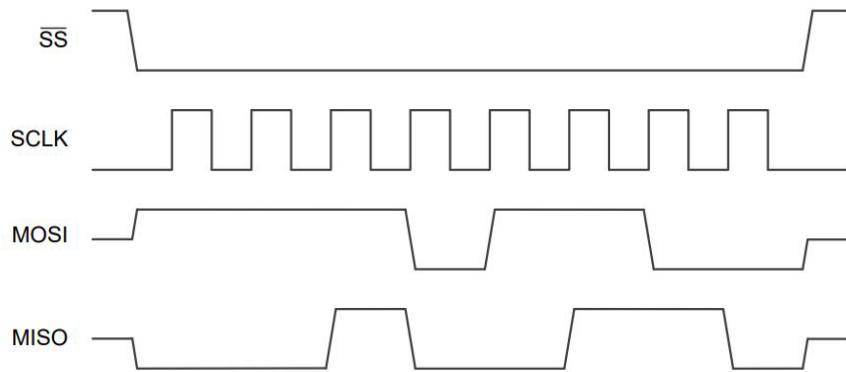
Peripherals - SPI

- Alternative configuration – Daisy chain
 - Not all devices support it



Peripherals - SPI

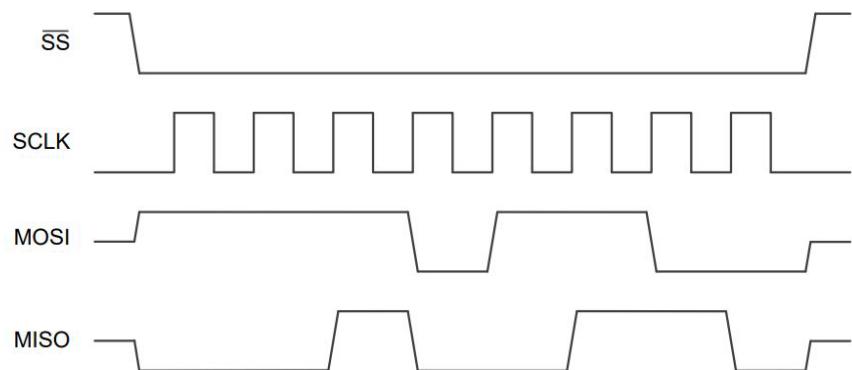
- Say we have an SPI device configured in Mode 1, whose traces are shown below.



- What byte (hex) is sent from the peripheral to the controller

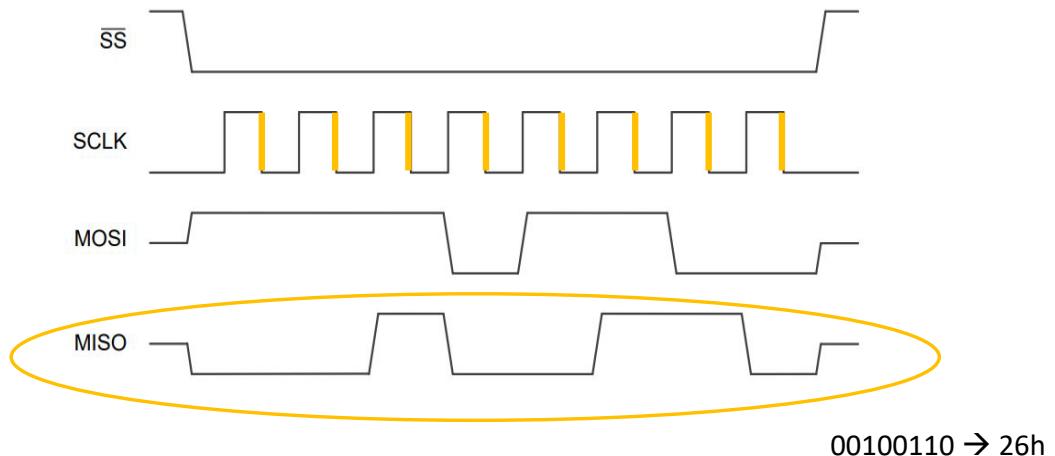
Peripherals - SPI

- Mode 1 \rightarrow CPOL = 0, CPHA = 1
- This means: idle low, trailing edge
- From peripheral to controller means MISO



Peripherals - SPI

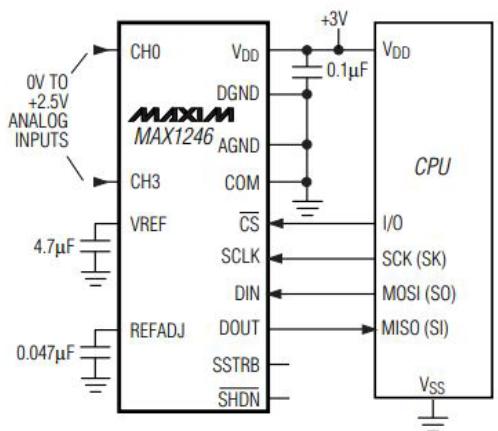
- Mode 1 → CPOL = 0, CPHA = 1
- This means: idle low, trailing edge
- From peripheral to controller means MISO



Peripherals - SPI

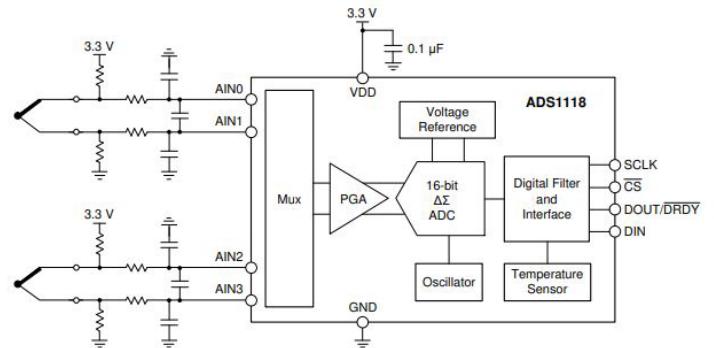
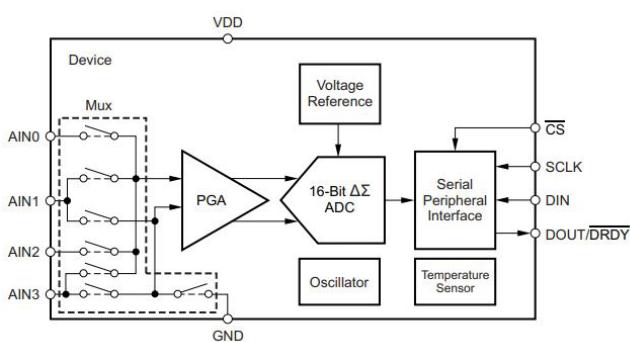
- Example from datasheet – Maxim ADC (1246/1247)

- Note: 4 analog inputs
 - Different designation
 - CS_bar for Slave Select
 - DIN for MOSI (input to slave)
 - DOUT for MISO (output from slave)



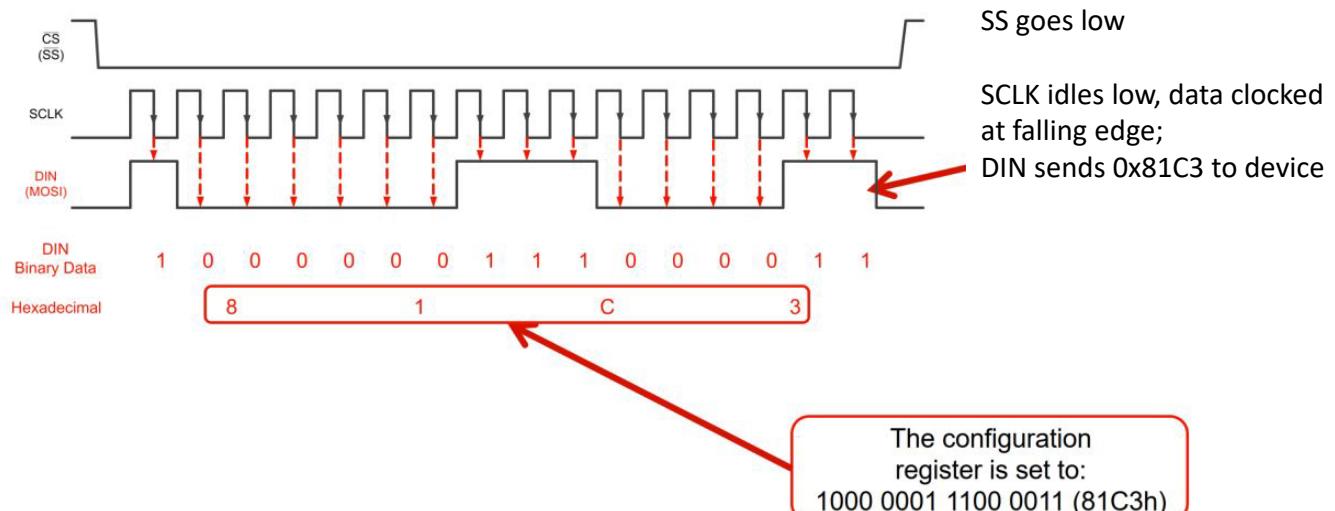
Peripherals - SPI

- Example from datasheet – ADS1118
 - Ultrasmall low power 16bit A/D with temperature sensor



Peripherals - SPI

- Setting up ADS1118 – using SPI Mode 1,
- Writing configuration register to 0x81C3



ECE342 – Computer Hardware

Lecture 15 - Examples

Bruno Korst, P.Eng.

Agenda

- Examples
 - Fixed point
 - Floating point
 - A/D, I2C



ECE342 – Computer Hardware

Lecture 16 - Midterm

Bruno Korst, P.Eng.

Agenda

- Midterm
 - Written in Lab



ECE342 – Computer Hardware

Lecture 17

Bruno Korst, P.Eng.

Agenda

- Serial Communication

Serial Comm

- Up to now, we have looked at two communication protocols utilized to “talk” to peripheral devices
 - These are serial communication protocols
 - They are bit-by-bit TX/RX, where both sides know when communication starts/ends, the size of data and the speed being used.
 - In contrast, parallel communication takes place with multiple lines/wires/traces that is more complex and costly
 - On chip as opposed to off-chip
- We shall focus on serial communication

Serial Comm

- Buffers
 - Buffers are used to prevent missing data
 - Transmission (TX) buffers store data prior to sending
 - Reception (RX) buffers store data until it can be retrieved
 - Buffers can be fixed (in hardware) or configured according to need and capacity (in software)

Serial Comm

- Asynchronous and Synchronous Communication
 - Asynchronous
 - Does not include clock like (that is the A in Asynchronous)
 - Devices provide their own clocks as timing reference
 - Both sides agree on the clock frequency that they have to match locally
 - They need to know when to start/stop and the duration
 - Synchronous
 - The interface includes a clock line controlled by one of the parts

Serial Comm

- Examples
 - I2C – synchronous serial
 - Max 40 devices, up to 18 ft, 3.4Mbps max.
 - SPI – synchronous serial
 - Max 8 devices, up to 10ft, 2.1Mbps max.
 - RS232 – Asynchronous serial (UART)
 - Max 2 devices, up to 100ft, 20kbps
 - RS485 – Asynchronous serial (UART)
 - Max 256 devices, up to 4000 ft, 10Mbps
 - USB – Asynchronous serial – up to 127 devices

Serial Comm

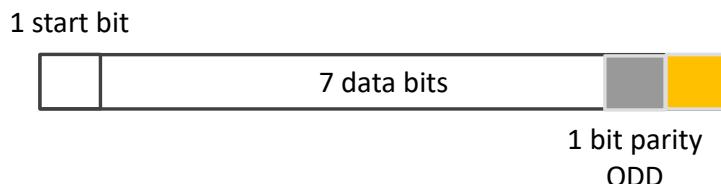
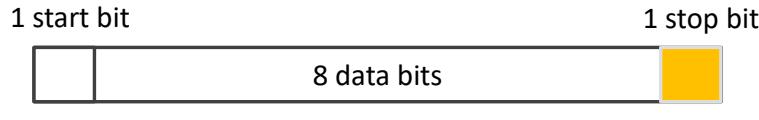
- Universal Asynchronous RX/TX (UART)
 - Back in Lab01 you learned how to communicate via UART to the PC
 - Short cable (your usb cable)
 - Some program (“Hercules”) to provide a “screen” to printf messages on a PC
 - Could have used putty, teraterm, etc.
 - You opened that program, clicked on “serial”
 - You set some “rate”, same between the board and the PC
 - You were told to use COM port, baud rate 38400, data size 8, parity none
 - On your code, you had a MX_USART2_UART_INIT();
 - Then config.c did the magic.
 - Let’s see what this actually does

Serial Comm

- Universal Asynchronous RX/TX (UART)
 - Internal or external to the device
 - Devices agree on transfer rate, frame format
 - Common frame formats
 - “8N1” – data = 8 bits, N = no parity, 1 stop bit
 - “7E1” – data = 7 bits, E = even parity, 1 stop bit
 - Parity: a basic form of error detection
 - Even parity – word transmitted has an even number of “ones”
 - Odd parity – word transmitted has an odd number of “ones”

Serial Comm - UART

- Data frame

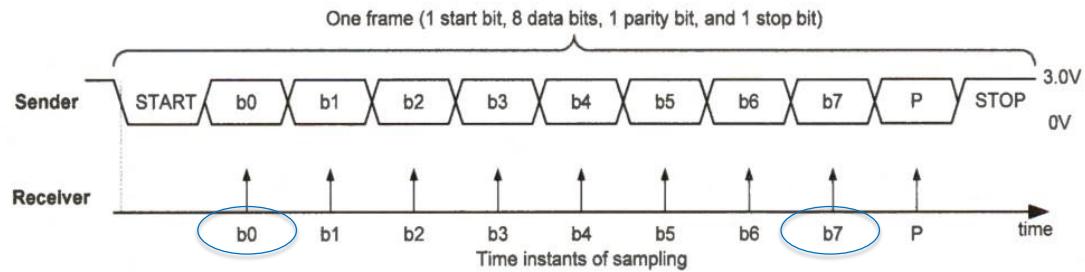


Serial Comm - UART

- Example of UART transmission:
 - I wish to transmit 0000011 (7 bits)
 - If I use 7E1 (7 data, 1 stop, Even parity)
 - I will transmit 0000 0110 (plus start and stop bits)
 - I already have an even number of “ones”, so parity is 0 if I wish to use EVEN parity
 - If I use 7O1 (7 bits data, 1 stop bit, Odd parity)
 - I will transmit 0000 0111 (plus start and stop)
 - The total amount of “ones” transmitted is three – two from the data and one from the parity itself

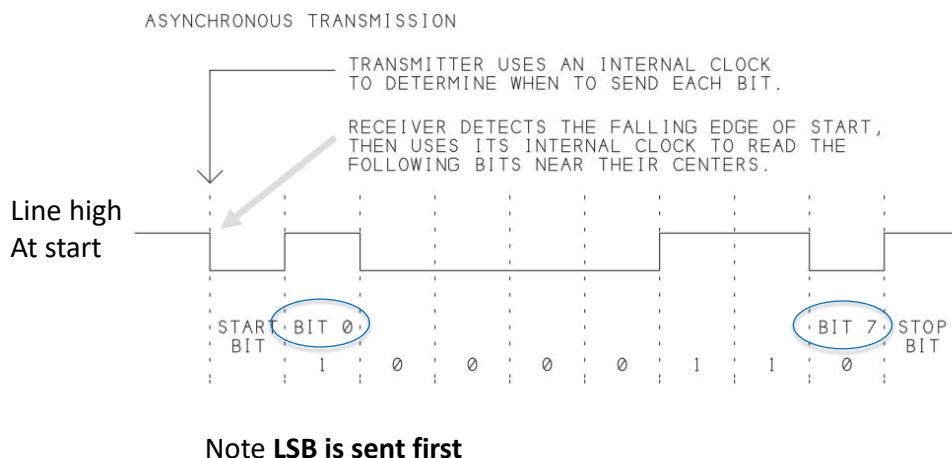
Serial Comm - UART

- This is how an 8-P-1 data frame will look like



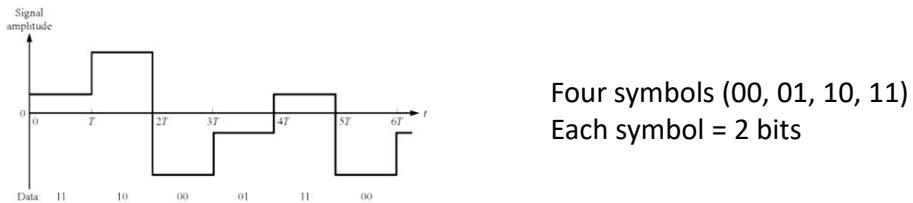
Serial Comm - UART

- This is how an 8N1 will look like



Serial Comm - UART

- Understanding Bit rate vs. Baud rate
 - Ex: 8N1 – 1 byte transmitted with 10 bits
 - If transmission is one bit at a time
 - Baud rate = bit rate
 - This is “bits per second”
 - In some systems (MODEMS), transmission is done in symbols
 - Baud rate = “symbols per second”



Serial Comm - UART

- Example:
 - An 8-N-1 frame consists of 8 bits of data, a stop bit and no parity bit
 - The baud rate is given as 9600 (this is bps)
 - The transmission rate in bytes is not simply 9600/8
 - One must account for protocol overhead of 1 start bit and 1 stop bit
 - Actual data rate is $9600 / (1 + 8 + 1) = 960$ bytes (of actual data) per second

Serial Comm - UART

- Note that setting up baud rate is device specific
 - A register is set to indicate the desired baud rate in UART serial comm.
 - Example: STM32L4
 - Register is BRR
 - Write the value calculated in USARTDIV

$$\text{USARTDIV} = ((1 + \text{oversample rate}) \times \text{fclock}) / \text{desired baud}$$

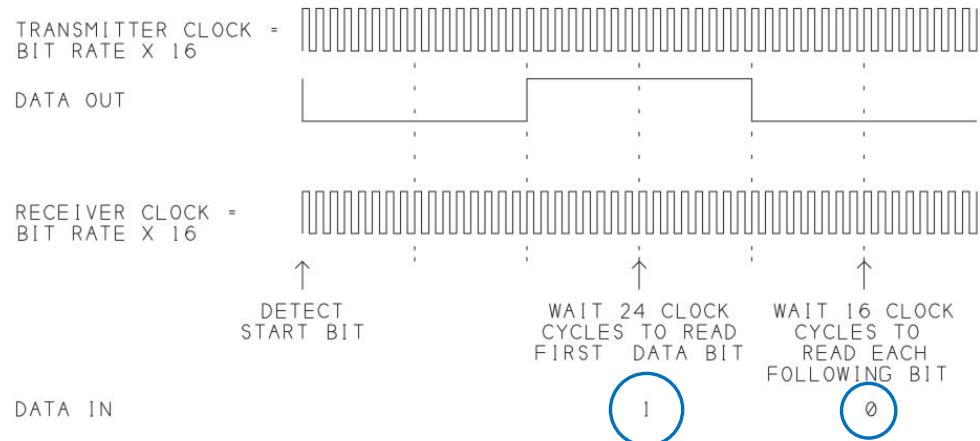
- fclock = clock of processor
- Oversample rate → set register OVER8
 - 0 is oversample by 16, 1 is oversample by 8

Serial Comm - UART

- Oversample is used to prevent errors
 - Sample many times during bit duration
 - Ex: 16x
 - Once start is detected
 - Count 24 cycles, read 1st data bit
 - Count 16 more, read 2nd data bit, etc.
 - Clocks should match within 3% of each other
 - Typically very accurately generated, via crystal oscillators

Serial Comm - UART

- Detect bit rate on oversampled receiver clock

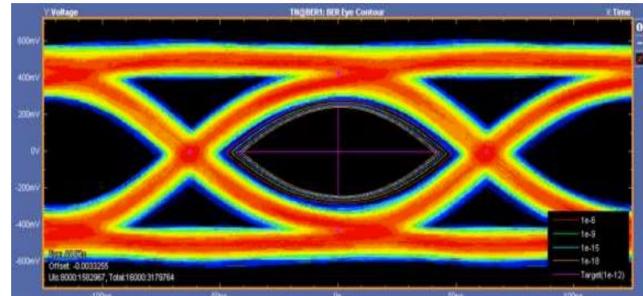
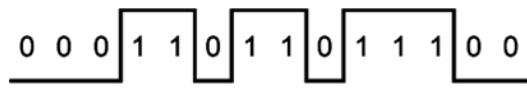


- Clock difference may introduce error

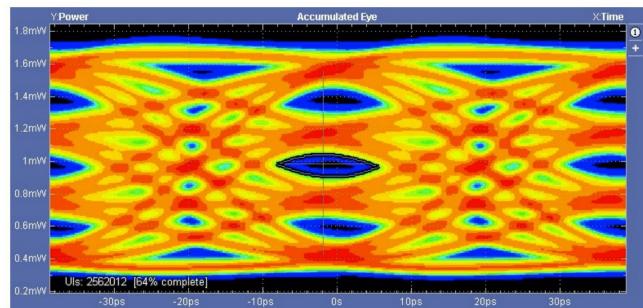
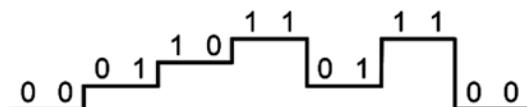
Serial Comm - UART

- Channel (medium) may also introduce error

PAM2-NRZ



PAM4



Serial Comm - UART

- Example
 - Desire a 9600 baud rate transmission
 - fclock is 80MHz
 - Want to oversample by 16 (that will be value 0 on the formulation)

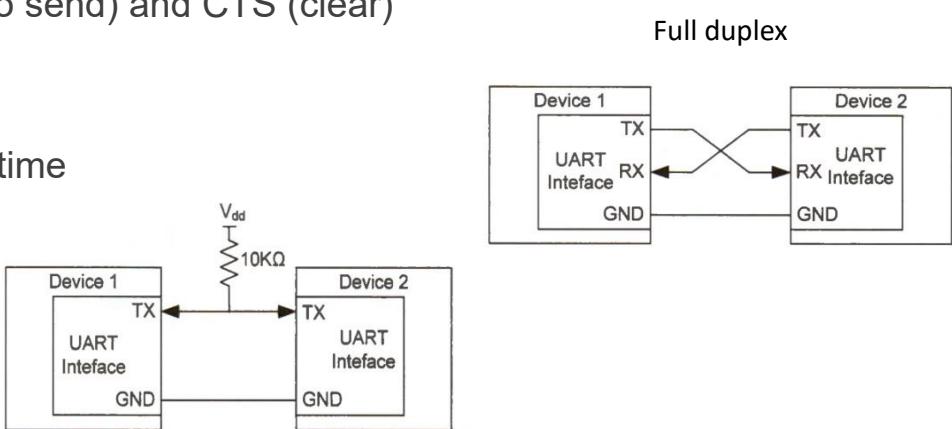
$$\text{USARTDIV} = (1+0) \times (80 \times 10^6) / 9600$$

$$\text{USARTDIV} = 8333.333$$

Result: write to BRR the value 0x208D (8333)

Serial Comm - UART

- 3 configurations
 - SYNCH (in case of USART) – either synch (send clock) or asynch
 - One device provides clock
 - Will use RTS (request to send) and CTS (clear)
 - ASYNCH in Full Duplex
 - Rx and Tx at the same time
 - ASYNCH in Half-Duplex
 - Only Rx or only Tx
 - Selected internally
 - Tx held high



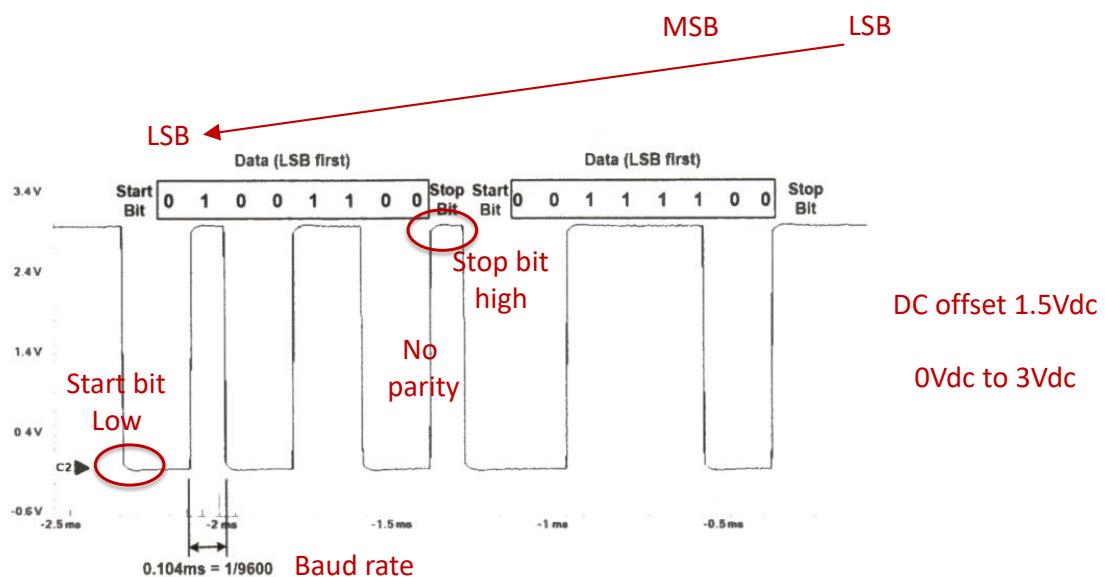
Serial Comm

- In case you did not look into config.c from Lab01...

```
void MX_USART3_UART_Init(void)
{
    /* USER CODE END USART3_Init 1 */
    huart3.Instance = USART3;
    huart3.Init.BaudRate = 38400;
    huart3.Init.WordLength = UART_WORDLENGTH_8B;
    huart3.Init.StopBits = UART_STOPBITS_1;
    huart3.Init.Parity = UART_PARITY_NONE;
    huart3.Init.Mode = UART_MODE_TX_RX;
    huart3.Init.HwFlowCtl = UART_HWCONTROL_NONE;
    huart3.Init.OverSampling = UART_OVERSAMPLING_16;
    if (HAL_UART_Init(&huart3) != HAL_OK)
    {
        Error_Handler();
    }
}
```

Serial Comm

- An illustration of 0x32 transmitted (recall this is 0b 0011 0010)

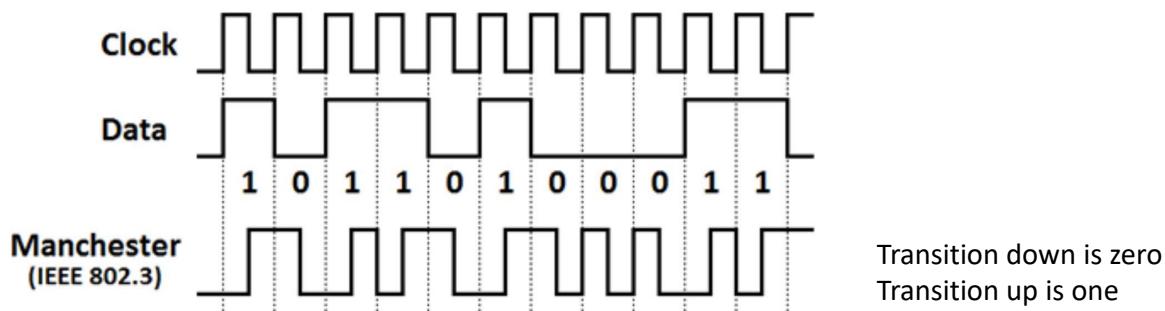


Serial Comm - UART

- How to keep track of / recover clock?
 - Manchester encoding
 - Replaces a standard UART encoding -- synchronous
 - Combine (encodes) clock at data in the same line
 - More tolerant to clock drifts (such as in found in RC oscillators)
 - Include a transition “in the middle” of every bit
 - Transition DOWN = 0 (IEEE 802.3)
 - Transition UP = 1

Serial Comm

- Manchester encoding



- Detect bit rate (and clock)
 - Rely on RX to keep track
 - Use a phase locked loop (PLL) at RX

Serial Comm – RS232

- Typically refers to TIA232F
- “Interface between data terminal equipment” – old but still found
 - 2 devices, 50 ft max, 20kbps, master/slave, full duplex
- PC to peripheral
 - “data terminal equipment” to “data circuit-terminating equipment”
- Uses 9 lines (9 pins)
 - Essential are RX, TX and GND
 - Other lines assert status/flow control

Serial Comm – RS 232

- Note that levels are not TTL/CMOS friendly
 - Conversion is needed (look up MAX220)
 - +15V to -15 V
 - @TX, logic 0 is +5V → +15V and logic 1 is -5V → -15V
 - @RX, logic 0 is +3V → +15V and logic 1 is -3V → -15V
- Note the existence of a -3V to +3V “transition zone”

Serial Comm – RS-232

- Dynamic of RS232
 - Data terminal (PC) asserts “Data Terminal Ready”
 - Data “circuit terminating” equipment (peripheral) senses readiness
 - Responds with Data Set Ready (comm is established)
 - PC asserts Request to Send
 - Peripheral responds to request with Clear to Send

send

Serial Comm - USB

- Universal Serial Bus
- More flexible interconnection between systems
 - Peripherals can be added / removed without the need to recognize the system
 - “plug and play”
 - Will concentrate on USB2.0 for simplicity
 - 4 wire interconnection D+, D-, power, ground
 - Host supplies 5V, 100mA

Serial Comm - USB

- Multiple types (know your type!)

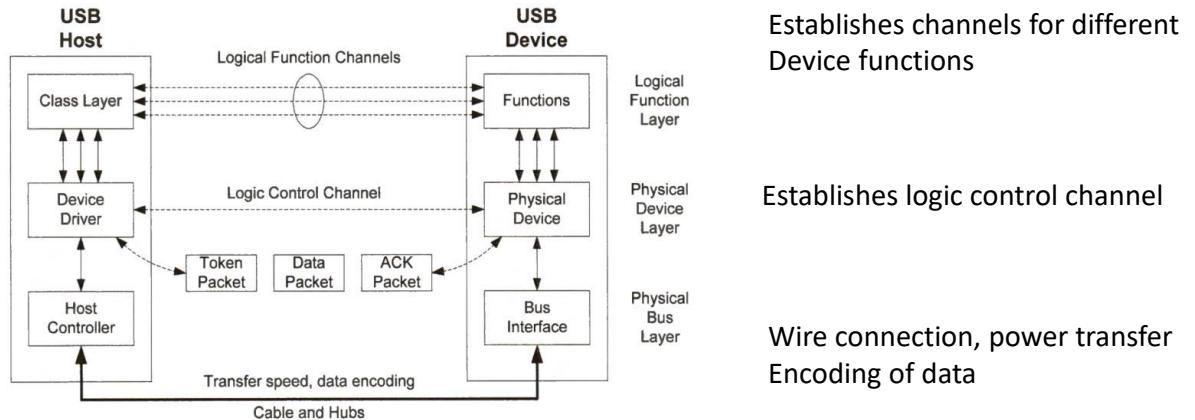


Serial Comm - USB

- How it works
 - When device is attached to bus
 - Host resets it
 - Assigns an address to it
 - Enumerates it
 - Device is identified
 - » What type it is
 - » What power it needs
 - » What data rate it uses
 - Data transfer initiated by host only
 - Packets contain information about type, direction of transfer, address of target

Serial Comm - USB

- Host – Device communication



Serial Comm - USB

- Bus Layer
 - 4 wires, twisted to prevent EM
 - <0.3 V is logic 0, > 2.8V is logic 1
 - High speed must support accuracy of +/- 500PPM
 - 12 PPM means 1 sec offset on clock per day
 - 500 is about 43 sec per day
 - Device may need external oscillator to maintain accuracy
 - Transmits using differential signals



ECE342 – Computer Hardware

Lecture 18

Bruno Korst, P.Eng.

Agenda

- Serial Communication

Serial Comm – RS232

- Typically refers to TIA232F
- “Interface between data terminal equipment” – old but still found
 - 2 devices, 50 ft max, 20kbps, master/slave, full duplex
- PC to peripheral
 - “data terminal equipment” to “data circuit-terminating equipment”
- Uses 9 lines (9 pins)
 - Essential are RX, TX and GND
 - Other lines assert status/flow control

Serial Comm – RS 232

- Note that levels are not TTL/CMOS friendly
 - Conversion is needed (look up MAX220)
 - +15V to -15 V
 - @TX, logic 0 is +5V → +15V and logic 1 is -5V → -15V
 - @RX, logic 0 is +3V → +15V and logic 1 is -3V → -15V
- Note the existence of a -3V to +3V “transition zone”

Serial Comm – RS-232

- Dynamic of RS232
 - Data terminal (PC) asserts “Data Terminal Ready”
 - Data “circuit terminating” equipment (peripheral) senses readiness
 - Responds with Data Set Ready (comm is established)
 - PC asserts Request to Send
 - Peripheral responds to request with Clear to Send

send

Serial Comm - USB

- Universal Serial Bus
- More flexible interconnection between systems
 - Peripherals can be added / removed without the need to recognize the system
 - “plug and play”
 - Will concentrate on USB2.0 for simplicity
 - 4 wire interconnection D+, D-, power, ground
 - Host supplies 5V, 100mA

Serial Comm - USB

- Multiple types (know your type!)



Serial Comm - USB

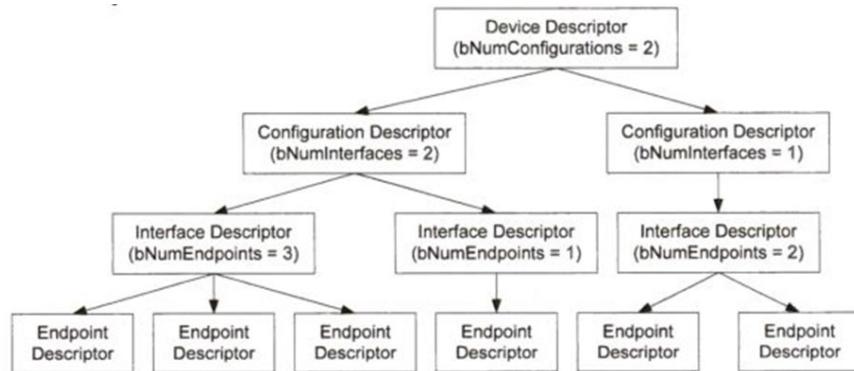
- How it works
 - When device is attached to bus
 - Host resets it
 - Assigns an address to it
 - Enumerates it
 - Device is identified
 - » What type it is
 - » What power it needs
 - » What data rate it uses
 - Data transfer initiated by host only
 - Packets contain information about type, direction of transfer, address of target

Serial Comm - USB

- What happens during enumeration (host)
 - Detects a device
 - Determines USB speed
 - Retrieves device descriptor
 - Retrieves configuration descriptor
 - Loads the corresponding device driver

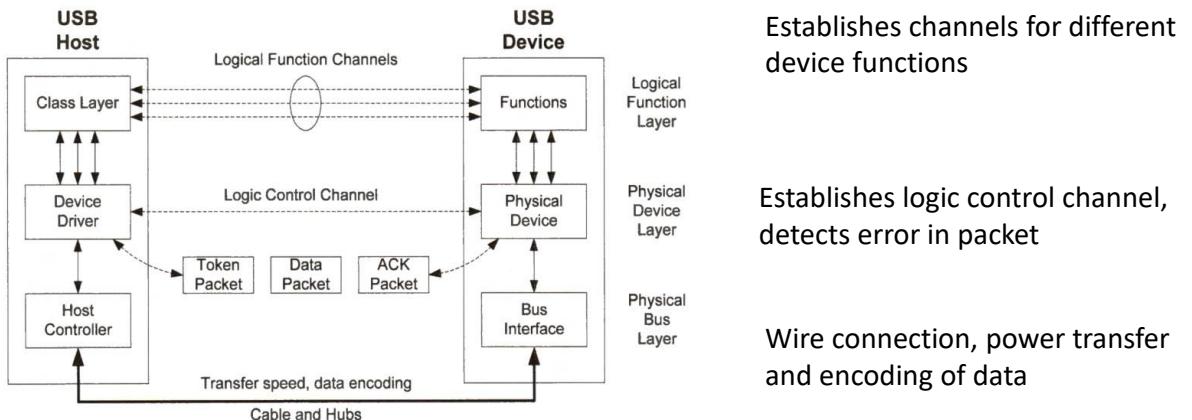
Serial Comm - USB

- During enumeration (host)
 - Note: for this example, we have one device and two configurations (like a camera/mic)



Serial Comm - USB

- Host – Device communication

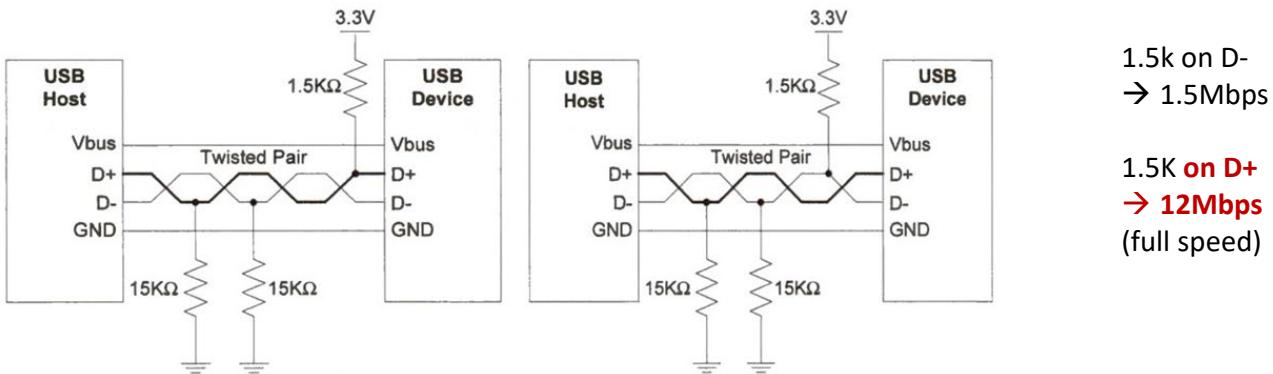


Serial Comm - USB

- Bus Layer
 - 4 wires, twisted to prevent EM
 - <0.3 V is logic 0, > 2.8V is logic 1
 - High speed must support accuracy of +/- 500PPM
 - 12 PPM means 1 sec offset on clock per day
 - 500 is about 43 sec per day
 - Device may need external oscillator to maintain accuracy
 - Transmits using differential signals

Serial Comm - USB

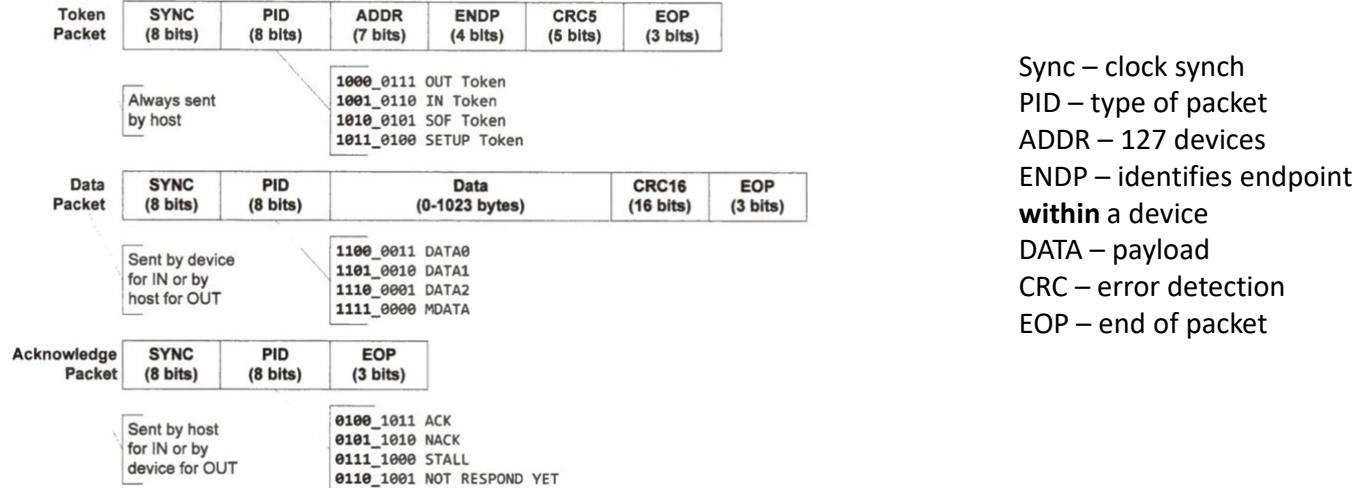
- Bus (Physical) layer
 - 4 wires, twisted
 - Note speed is determined by pull-up resistors



Serial Comm - USB

- Device Layer
 - Packeted data
 - Info on type/direction of transfer, address of target, acknowledgement
 - Three types of packets
 - Token
 - “in” token – host requests read (“out” → write)
 - Data
 - Acknowledge

Serial Comm - USB



Sync – clock synch
 PID – type of packet
 ADDR – 127 devices
 ENDP – identifies endpoint **within** a device
 DATA – payload
 CRC – error detection
 EOP – end of packet

Serial Comm - USB

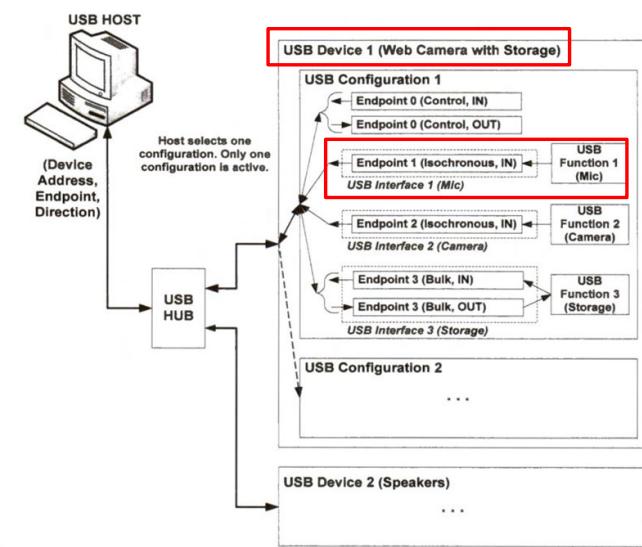
- Transactions
 - Host sends token
 - Indicate recipient (endpoint), address and direction of transfer
 - PID ok → host sends data, device sends ack
 - Case of a “write”
 - If token packet is “in” token
 - Device sends data packet, host sends ack

Serial Comm - USB

- Data transfer types
 - Control
 - Host obtains descriptors (all devices must support)
 - Bulk
 - Large burst-type data (mass storage/print/scan)
 - Interrupt
 - Host queries – mouse/keyboard
 - Isochronous
 - Microphones/cameras – guaranteed latency but no error detection

Serial Comm - USB

- Device may have multiple functions

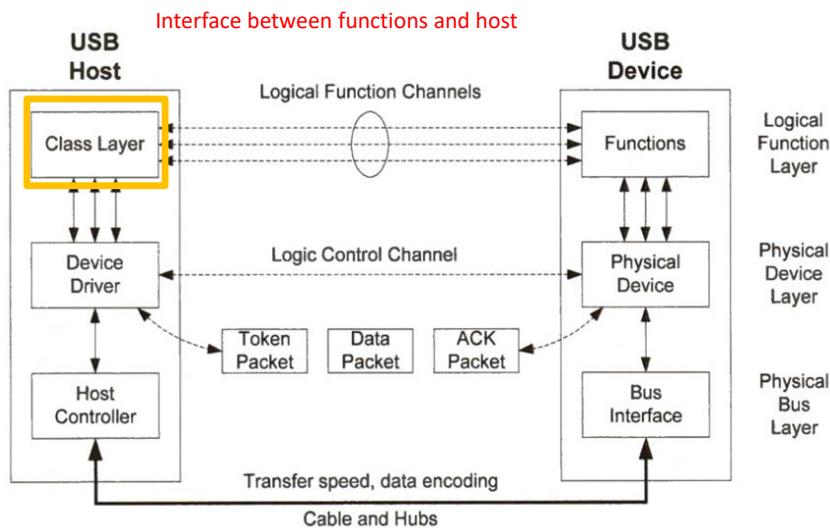


The “endpoint” is the interface between a function and the host.

Interface descriptors have information on device and are associated with functions

Serial Comm - USB

- From host side



Class Layer:
functions have predefined
class protocols to
handle requests

Serial Comm - USB

- Class Layer
 - HID – human interface device
 - CDC – communication device class
 - PHDC – personal healthcare device
 - MSC – mass storage device
 - Audio
 - Video

Memory

- The element that stores information
 - In embedded systems, most basic: register
 - Separate registers if individual items
 - Memory components if multiple items
 - An array of storage registers
- We saw in ECE241
 - FF – registers
 - Memory blocks
 - Down to the transistor-level storage
 - Saw how to combine blocks (wider/longer)

Memory

- Distinct address (identify the location)
- Each location stores m bits of encoded information (word size)
 - 2^n locations = addresses from $0 \rightarrow 2^n - 1$
 - Total storage in bits: $m \times 2^n$
- Can be organized in different ways
 - $1\text{Mbit} = 32\text{K} \times 32\text{bit}, 64\text{K} \times 16\text{bit}$ or $16\text{K} \times 64\text{bit}$

Memory

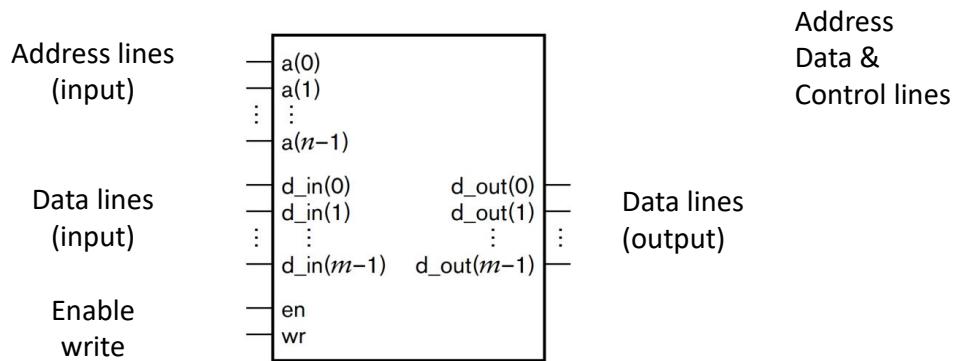
- Say a memory module has 32,768 locations, each 32 bits
 - Total capacity : $1,048,576 = 1 \text{ Mega bits}$
 - How many address bits: locations = 2^{15}
- Kilo = $2^{10} = 1,024$
- Mega = $1,024 \times 2^{10} = 1,048,576 (2^{20})$
- Giga = $1,024 \times 2^{20} = 1,073,741,824 (2^{30})$
 - (careful: binary prefixes different than decimal)

Memory

- Two basic operations:
 - Write binary to a location
 - Provide address and data to write
 - Address on address lines, data on data lines
 - Read binary from a location
 - Provide address (on address lines)
- Address encoded as unsigned binary number
- Need to provide control signals (binary)
 - Enable and write

Memory

- Example:
 - Have enable = 1, write = 0
 - Memory **reads** contents of location
 - Input is the address lines
 - Drives the data value on the data output lines



Memory

- In ECE243 (DE1SoC) you saw
 - OnChip (Static RAM)
 - Keeps info as long as power is applied
 - DRAM (asynch)
 - Dynamic, needs refreshing (frequent read/write)
 - Memory controller responsible for refreshing
 - Synchronous DRAM
 - Uses clock to incorporate controller
 - Refresher is built-in
 - Large memory units (DIMM modules)

Memory

- ECE243 cont'd
 - DDR – double data rate SDRAM
 - Data transferred both on rising and falling edges of the clock
 - Presently at DDR5 version
 - Cache
 - Reduce memory access time
 - Fast memory closer to processor, holds active parts of program and data
 - ROM – read only (PROM, EPROM, EEPROM)

ECE342 – Computer Hardware

Lecture 19

Bruno Korst, P.Eng.

Agenda

- FIRST WEEK AFTER READING WEEK
- Memory
- A look at Lab 5
 - Camera module
 - Image
 - Functionality
- DMA

Memory

- Before the break, we had reviewed memory (from 243)
- Memory modules connections
 - Address lines
 - Data lines
 - Chip Select (enable)
 - Read/write
- Types
 - On-chip Static RAM, Dynamic RAM, Synchronous DRAM, DDR
 - Cache
 - ROM – read only (PROM, EPROM, EEPROM)

Memory

- In ECE243 (DE1SoC) you saw
 - OnChip (Static RAM)
 - Keeps info as long as power is applied
 - DRAM (asynch)
 - Dynamic, needs refreshing (frequent read/write)
 - Memory controller responsible for refreshing
 - Synchronous DRAM
 - Uses clock to incorporate controller
 - Refresher is built-in
 - Large memory units (DIMM modules)

Memory

- ECE243 cont'd
 - DDR – double data rate SDRAM
 - Data transferred both on rising and falling edges of the clock
 - Presently at DDR5 version
 - Cache
 - Reduce memory access time
 - Fast memory closer to processor, holds active parts of program and data
 - ROM – read only (PROM, EPROM, EEPROM)

Memory

- Add external memory to a microcontroller
 - STM32F7x comes with capability to interface external memory (FMC)
 - Example: MT48LC4M32B2 SDRAM – 128Mb (1Meg x 32 x 4banks)
 - Configuration of memory and microcontroller allows the external SDRAM to be seen as internal memory to the microcontroller
 - Other alternatives for other families
 - SPI Flash memory module, such as W25Q64BV (64Mb)
 - SD card via SPI

Memory

- What do we need to design a digital camera
 - STM32F4x – based board
 - Memory card
 - SD card, connecting via SPI
 - LCD display
 - Interfaces using flexible static memory controller (FSMC)
 - A button – via GPIO
 - A rotary encoder – if we want a timer...
 - A camera module
 - Interfaces via I2C (or a variation of it) and a digital camera interface

Let's look at Lab 5

- For Lab5 you will interface a camera with the STM32
- You will use the DCMI (Digital Camera Interface) and will use DMA
- The camera module is the OV7670
 - Uses an I2C compatible protocol (SCCD)

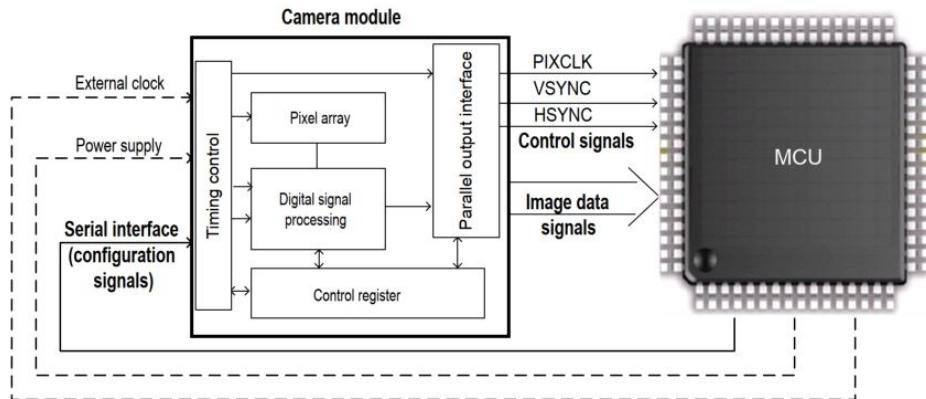
Lab 5

- Digital camera modules
 - Image sensor (CMOS or CCD)
 - Lens
 - PCB
 - Interconnect
 - Control signals
 - Camera configuration
 - Image data signals
 - Power supply



Lab 5

DCMI



- Control signals: clock and synchronization (horizontal and vertical)
 - Horizontal relates to line, vertical relates to frame
- Image data signals: these are the bits representing image pixels
- Configuration: resolution, format, frame rate, type of interface
- Power supply

Lab 5

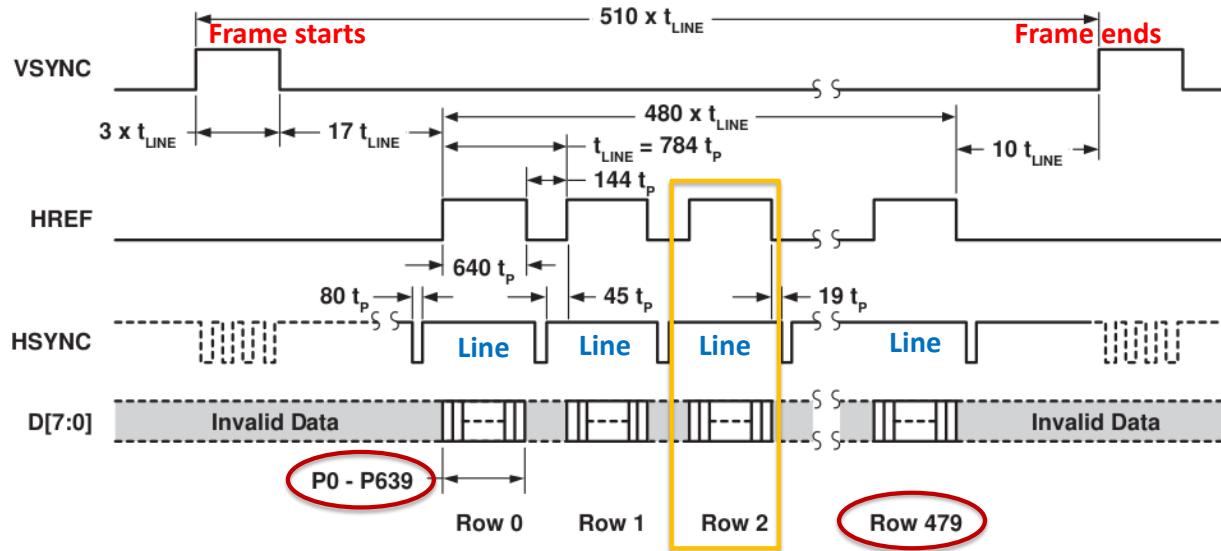
- Formatting images
 - A video is a succession of frames
 - A frame is comprised of lines
 - A line is comprised of pixels
- Pixels
 - monochrome (8 bit gray scale: 0 white, 255 black)
 - RGB – red, green, blue
 - Camera module OV7670 supports RGB565, RGB555 and RGB444
 - Ex: RGB444 → 4 bits red, 4 bits green, 4 bits blue
 - 0 represents dark (no light) and 15 (4 bits) full intensity
 - YCbCr – a format to encode RGB using luma (bright) and chroma (blue and red)

Lab 5

- Signaling
 - OV7670 uses parallel synchronous (needs clock)
 - Data is sampled at the rising edge of PCLK (pixel clock) only when HREF (horizontal synchronization) is high.
 - HREF rising edge → start of a line (falling edge → end of a line)
 - All bytes taken when HREF is high are the pixels in one line.
 - Depending on the format chosen, different number of bytes per pixel
 - VSYNCH will indicate start and end of frames
 - PCLK will dictate the frame rate (24MHz will produce 30 fps)

Lab 5

- Example: signaling for VGA frame format (640 x 480)

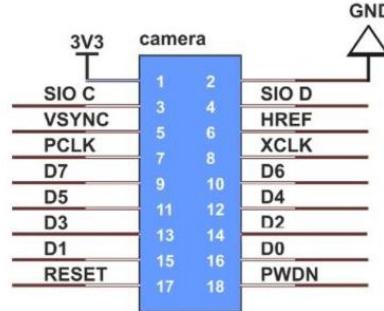


Bruno Korst - Winter 2023

13

Lab 5

- The OV7670 Camera module
 - I2C refers to SCCD
 - Pixel data output is 8 bits in parallel



Pin	Type	Description
VDD**	Supply	Power supply
GND	Supply	Ground level
SDIOC	Input	SCCB clock
SDIOD	Input/Output	SCCB data
VSYNC	Output	Vertical synchronization
HREF	Output	Horizontal synchronization
PCLK	Output	Pixel clock
XCLK	Input	System clock
D0-D7	Output	Video parallel output
RESET	Input	Reset (Active low)
PWDN	Input	Power down (Active high)

Bruno Korst - Winter 2023

14

Lab 5

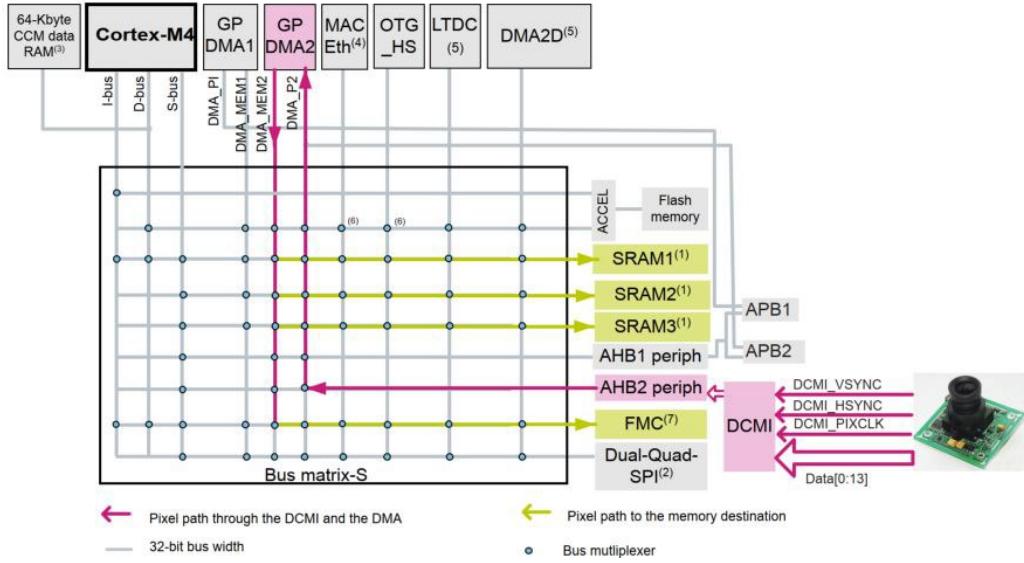
- Now we defined the specs for the picture, need to set up DCMI configuration
 - GPIO configuration
 - Need to configure the I2C, and enable interrupts via NVIC
 - Clock and timing configuration
 - Configure the DCMI peripheral
 - Capture mode, data format, image size and resolution
 - Configure the memory communication (DMA, seen below...)
 - Configure the camera module

Lab 5

- SCCB – serial camera control bus
 - It's an I2C compatible interface
 - Two wire interface: only one master, at least one peripheral
 - SIOC and SIOD
 - Serial bus clock signal (SCL), serial bus data signal (SDA)
 - All transmissions are initiated by the master

Lab 5

- Connection between camera and device is via DCMI to a memory controller

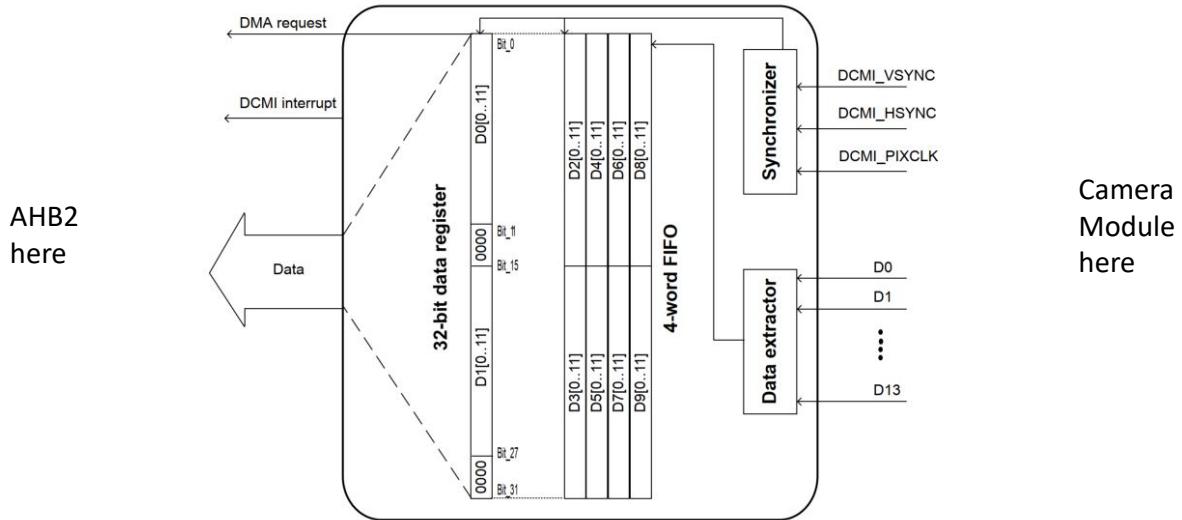


Bruno Korst - Winter 2023

17

Lab 5

- The DCMI connects the module signals with the bus and a memory controller (DMA)



Bruno Korst - Winter 2023

18

Lab 5

- DCMI functional description
 - Synchronizer controls data flow through data extractor, FIFO and 32 bit register
 - Data is packed into 4 word FIFO, then ordered into 32 bit register
 - A request to the memory controller is generated
 - Transfers the data to the corresponding memory destination
- Note: where is the CPU in all this?

Memory

- Suppose you have a system collecting data from an I/O port, and the data needs to be stored as it is collected.
 - CPU runs a program, part of it is to collect data from the I/O port
 - When that part of the program is reached, the port is read
 - Possible outcomes between the processor and external device
 - Program tries to read but device is not ready (“busy waiting”)
 - Device is ready with data, program was running another part (did not try to read)
 - Program is made to read frequently (polling) so that it will not miss any data
 - Device interrupts CPU operation with data ready
 - » CPU goes to handle the interruption
 - » Collects data, stores it, goes back to doing what it was doing

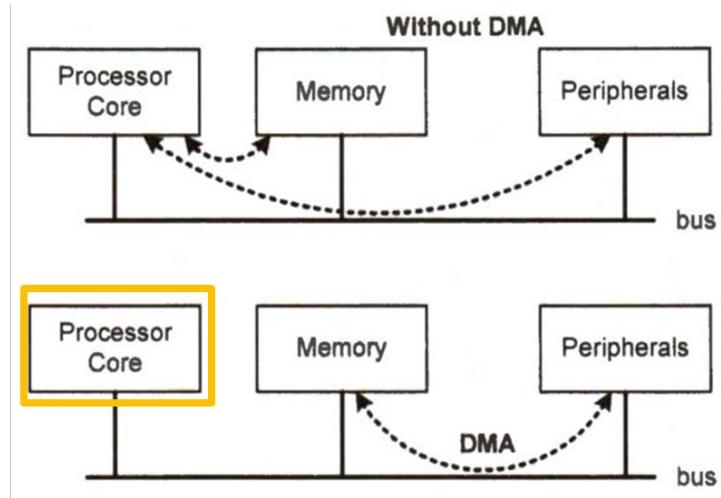
Memory

- The options demand that the CPU stop its execution to handle the incoming data
- A **better option**: spare the CPU of having anything to do with the transfer
 - This is **Direct Memory Access (DMA)**
 - In this case, the CPU only gets involved when the data is all transferred and ready

Direct Memory Access

- With DMA, the transfer is done via the DMA controller
- Improves the energy efficiency of the system
 - It transfers between peripherals and memory and between memory and memory
- Two types of strategies
 - Cycle stealing – uses spare/idle cycles of the CPU
 - Independent DMA controller

Direct Memory Access



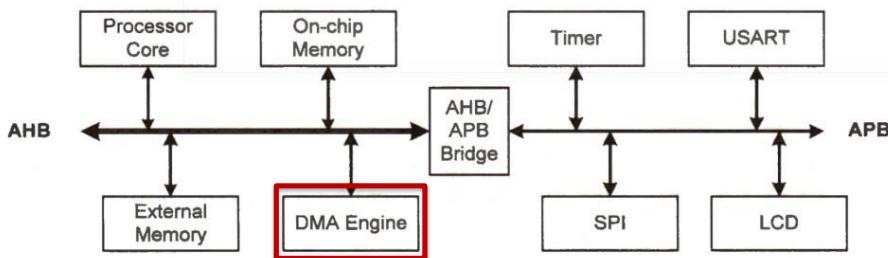
Processor is not involved and can execute other tasks

Direct Memory Access

- Offers an efficient way to interface peripherals
 - Slow peripherals (such as a sensor)
 - DMA releases the processor from waiting for data
 - This frees the CPU to perform other tasks
 - Fast peripherals (such as an external ADC)
 - DMA improves data throughput
 - Memory access does not involve the CPU
 - In high-speed, DMA helps to reduce interrupt rate, and its associated overhead

Direct Memory Access

- Storing data with DMA
 - Peripheral does not need local memory
 - Data is stored efficiently in data memory
- For the STM32 platform, recall the AMBA bus architecture



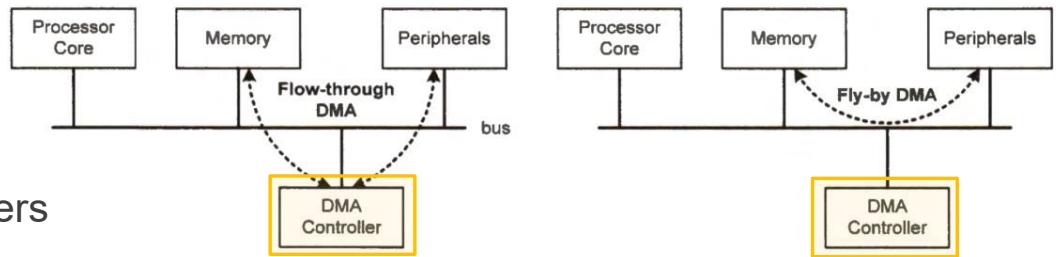
Direct Memory Access

- The On-chip DMA controller resides on the AHB bus
 - A high-performance bus, which allows for multiple masters (with arbitration)
- When data is coming from the APB bus (low speed peripherals)
 - Data goes through the bridge
 - The bridge will act as a slave for AHB, and as a master for APB
 - Provides buffering for address, control and data
 - No loss due to difference in speed between buses

Direct Memory Access

- On-chip DMA controller
 - Can act as bus master and bus slave
 - As a master
 - Initiates transfer within the AHB
 - Initiates data transfer across the AHB/APB bridge
 - As a slave
 - Takes data and commands from processor when DMA transfers are set up
 - It manages multiple channels simultaneously, each with own interface to peripherals

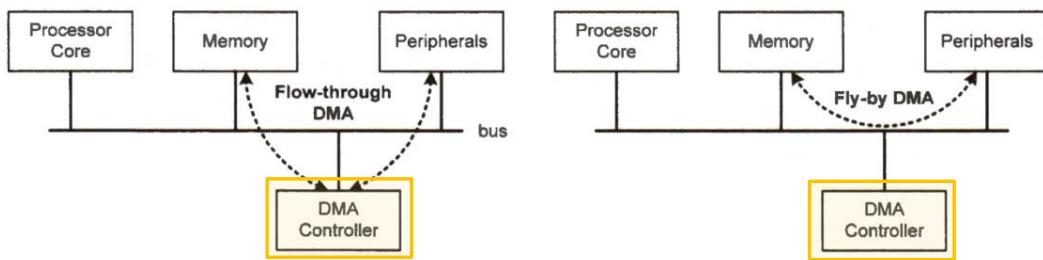
Direct Memory Access



- Types of DMA Transfers
 - Flow-through
 - Uses register in the controller
 - Data is read one at a time from source to register
 - Data is written from register into destination
 - Use when:
 - Devices have registers of different sizes
 - When memory to memory transfer cannot be read/written in one cycle

Direct Memory Access

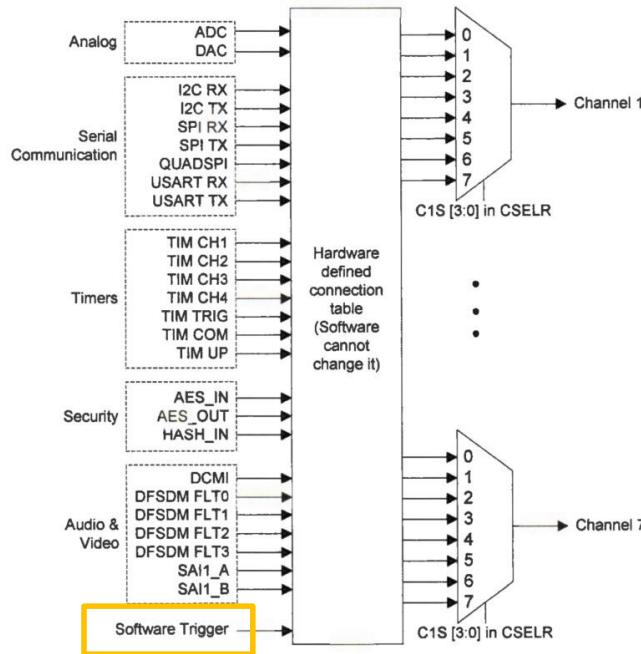
- Fly-by DMA
 - Data goes from source to destination without using controller register
 - More efficient than flow-through; it's used when flow-through is not imperative



Direct Memory Access

- DMA controllers offer multiple channels
 - Channels transfer data independently
 - Each has source, destination, transfer direction, transfer width, data amount and trigger
 - When the channel is enabled and the trigger is received, DMA transfers occur automatically
 - A register is used to select events for each channel

Direct Memory Access



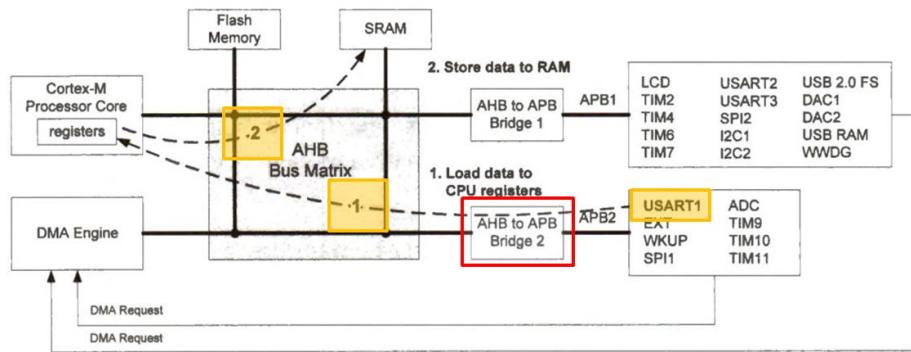
- For one DMA controller
 - Each channel performs transfers independently
 - One trigger is available per channel
 - 7 channels, 8 event types
 - Any of the event types can be selected as trigger
 - Trigger is selected in a register, plus a software trigger if needed.

Direct Memory Access

- When programming DMA
 - Which DMA controller should be used? (if multiple)
 - Which channel should be used?
 - Different channels have access to different resources (ADC, USART, I2C, etc)
 - Which trigger should be selected?
- Priorities
 - Hardware priority is higher, channel 1 is highest among them
 - If using software priority, highest should be channel demanding highest BW

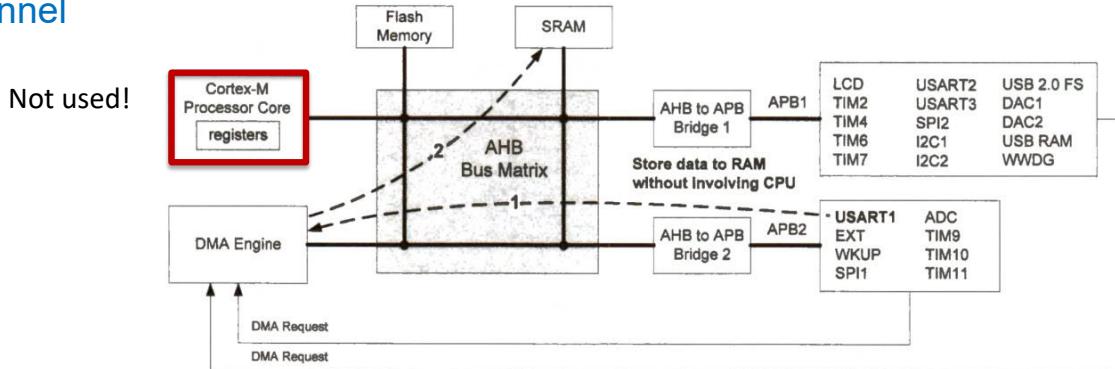
Direct Memory Access

- Example: Transfer USART to RAM
 - First: **without DMA** (i.e.: CPU is occupied)
 1. Processor executes load data to CPU registers
 2. Processor executes store data to RAM



Direct Memory Access

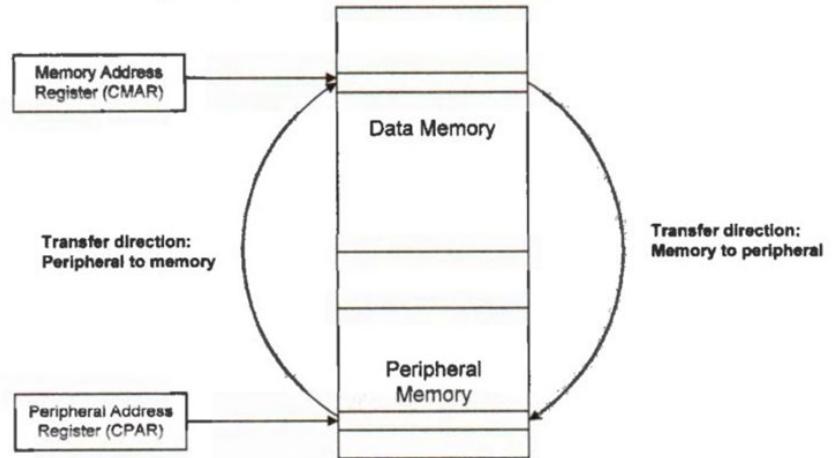
- Example: Transfer USART to memory – with DMA
 - Processor **programs DMA controller** (i.e. sets up DMA transfer) and **enables the channel**



- At the end of transfer, DMA engine sends interrupt to inform data is saved

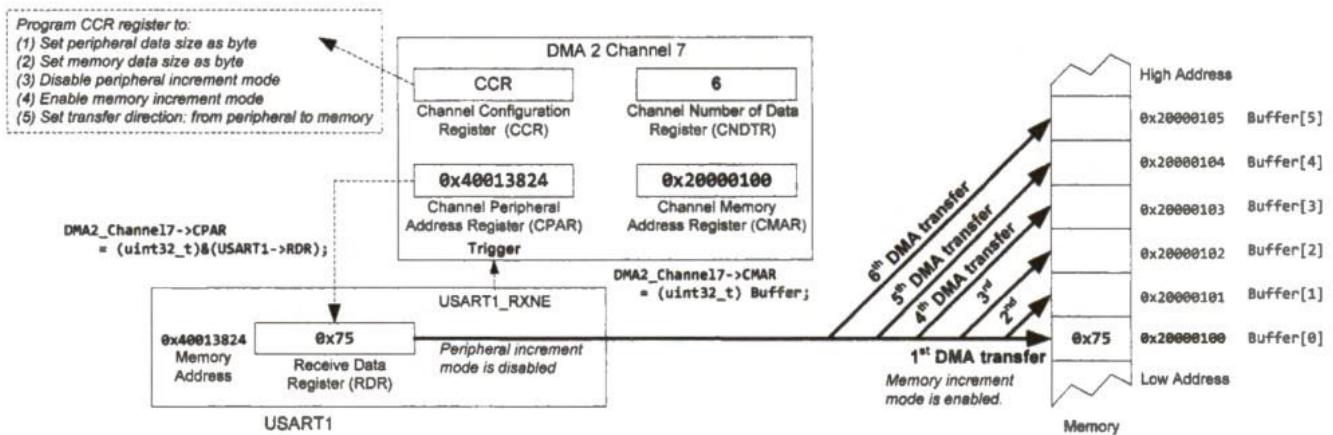
Direct Memory Access

- Each DMA channel has 4 registers, indicating
 - Start memory address
 - Start peripheral address
 - Transfer size (how much data)
 - Direction of transfer and other channel configuration



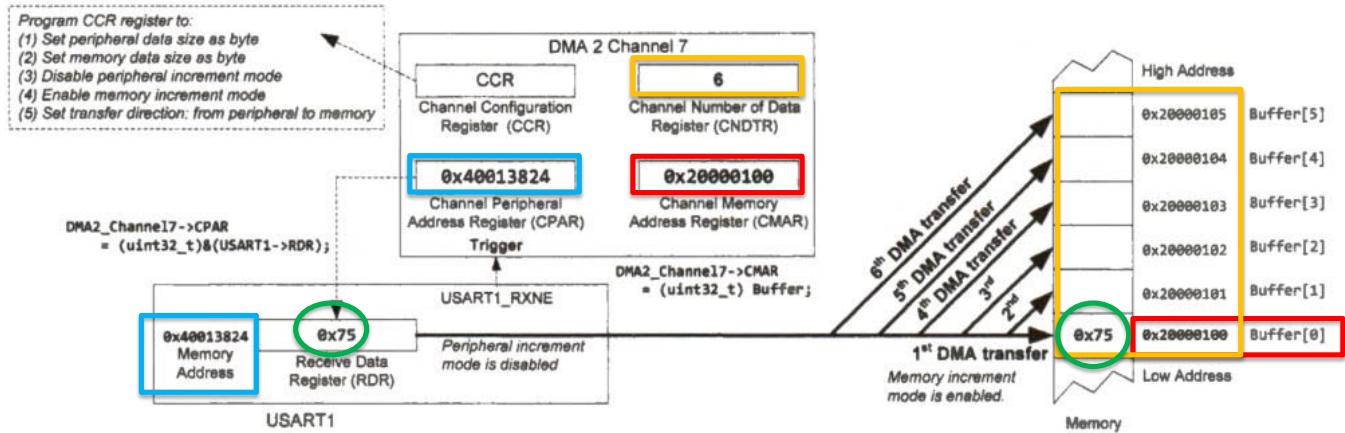
Direct Memory Access

- DMA controller 2, channel 7 – transferring 6 bytes
 - Peripheral address 0x40013824, memory (buffer) address 0x20000100



Direct Memory Access

- DMA controller 2, channel 7 – transferring 6 bytes
 - Peripheral address 0x40013824, memory (buffer) address 0x20000100



Direct Memory Access

- Note in setting up DMA
 - May use both TX and RX to two separate buffers (memory), which means two separate channels to set up
 - Each channel will be triggered by the peripheral when the respective register is “filled”
 - Interrupts can indicate DMA transfer finished/half finished/error
 - Buffer may be set up as “circular” for continuous data streams (when counter reaches the amount of data to be transmitter, address returns to start)

ECE342 – Computer Hardware

Lecture 20

Bruno Korst, P.Eng.

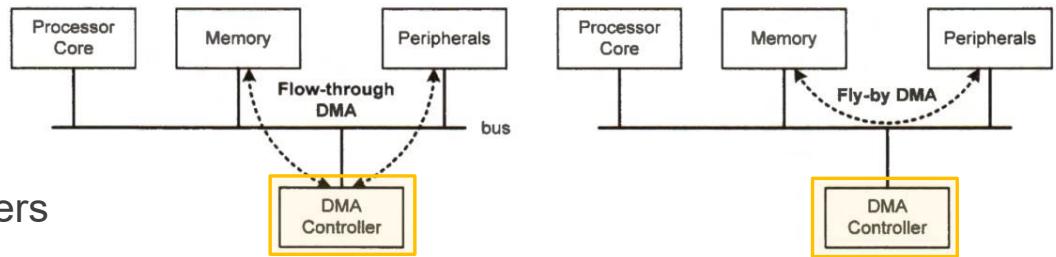
Agenda

- Clarification from last class
- DMA Examples
 - Ping pong
- Processing Unit
 - Intro to CPU – measures of performance

Direct Memory Access

- With DMA, transfer between peripheral and memory is done via the DMA controller
- Two types of strategies
 - Cycle stealing – uses spare/idle cycles of the CPU
 - With single bus, the bus can be used by either CPU or DMA controller
 - Data transfer happens only when the CPU is idle
 - DMA controller issues a “hold” to the CPU, and CPU acknowledges
 - When transfer is done, hold is deasserted and CPU regains control of bus
 - Independent DMA controller on separate bus

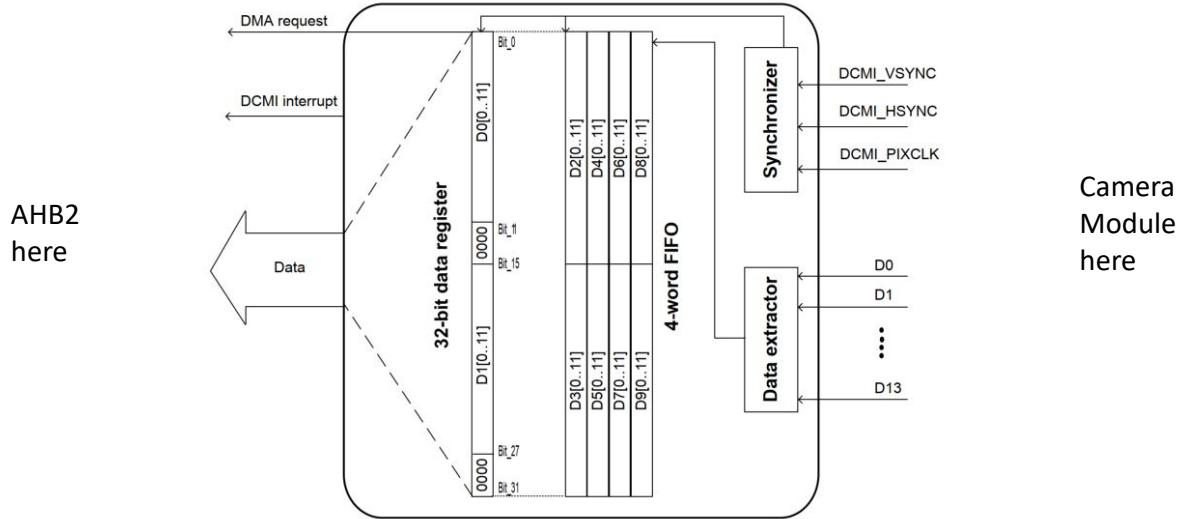
Direct Memory Access



- Types of DMA Transfers
 - Flow-through
 - Uses register in the controller
 - Data is read **one word at a time** from source to register
 - Data is written from register into destination
 - Use when:
 - Devices have registers of different sizes
 - When memory to memory transfer cannot be read/written in one cycle

Direct Memory Access

- The DCMI connects the module signals with the bus and a memory controller (DMA)



Direct Memory Access

- In sum: DMA is about no CPU participation in the data transfer process
 - After DMA configuration, CPU is free
 - As data becomes available through DMA, CPU engages in processing the data
 - Transferring of data takes time, adds delay
 - If processing is done in frames/blocks, processing by CPU must be done prior to next frame being available (real-time constraint)
 - Depending on the application, delay may not be tolerable
 - Ex: control of autonomous process with data from multiple sensors

Direct Memory Access

- Examples:
 - Say we have 2KBytes of data to transmit – UART at 9600 baud
 - Using polling, CPU waits for UART TX data register to be available (empty – TXE flag), reads data from memory and writes it to the UART TX data register
 - CPU is occupied for the entire time, not able to do any other task
 - At 9600 baud, 9600 bits/sec (104.1us per bit) or 1200 bytes/sec
 - Actual speed with 8 data bits, no parity, one stop bit – 960 bytes/sec (1.042 ms per byte)
 - For 2 Kbytes, the CPU will be **fully occupied for about 2 seconds**

Direct Memory Access

- Let's now transfer 2KBytes from memory to UART **using interrupts**
 - When UART TX data register becomes empty, generates an interrupt
 - CPU halts, saves current CPU registers on stack, goes to serve the handler
 - Handler reads a byte of data from memory, writes it to UART TX data register
 - CPU restores registers from stack and resumes the task that it was executing
- All this takes a couple of **microseconds** of CPU time to move one byte of data, for a 16MHz clock
 - 2KBytes of data will then take a **couple of milliseconds**
 - A lot better than polling

Direct Memory Access

- If we were to do this same 2KBytes transmission with DMA:
 - Software configures DMA controller (CPU occupied)
 - Set beginning **address** for data in **memory** (source address register)
 - Set **UART TX register address** (destination register)
 - Set **number of bytes** to transmit (data count register)
 - Set **trigger** to start transmission when **UART TX register is empty**
 - Recall, TXE flag goes up when data register does not feed any data into the UART TX register (that is: ready for new data)
 - When trigger happens, the DMA controller takes over without involving CPU
 - When UART TX register empty again, controller is notified (transfer complete)

Direct Memory Access

- Recall for the next examples that the STM32F4xx has two **DMA controllers** with 8 streams each, and 8 channels per stream.
 - These connect a variety of peripherals
- Example:
 - A loop program that takes inputs from ADC and puts them back out to DAC
 - DMA-based transfer will make use of **ping-pong buffering**
 - Buffer size is 256, and samples are alternating 16 bit words (L and R)
 - That is, each buffer holds 128 samples L and 128 samples R, alternating

Direct Memory Access

- Two IRQHandlers are used
 - Transfer from I2C to memory (IN)
 - Transfer from memory to I2C (OUT)
 - Determine which buffer is currently in use
- Streams 4 and 3 are configured
 - 3 for “input” arrays (pingIN, pongIN)
 - RX buffer
 - 4 for “output” arrays (pingOUT, pongOUT)
 - TX buffer

```
void process_buffer()
{
    int i;
    uint16_t *rxbuf, *txbuf;

    if (rx_proc_buffer == PING)
        rxbuf = pingIN;
    else
        rxbuf = pongIN;
    if (tx_proc_buffer == PING)
        txbuf = pingOUT;
    else
        txbuf = pongOUT;

    for (i=0 ; i<(BUFSIZE/2) ; i++)
    {
        *txbuf++ = *rxbuf++;
        *txbuf++ = *rxbuf++;
    }
    TX_buffer_empty = 0;
    RX_buffer_full = 0;
}
```

Direct Memory Access

- This function
 - Copies the most recently filled input buffer to the most recently emptied output buffer
 - Note two copies per processing
 - Takes care of L and R samples
- Any processing must be completed before the next DMA transfer is done.

```
void process_buffer()
{
    int i;
    uint16_t *rxbuf, *txbuf;

    if (rx_proc_buffer == PING)
        rxbuf = pingIN;
    else
        rxbuf = pongIN;
    if (tx_proc_buffer == PING)
        txbuf = pingOUT;
    else
        txbuf = pongOUT;

    for (i=0 ; i<(BUFSIZE/2) ; i++)
    {
        *txbuf++ = *rxbuf++;
        *txbuf++ = *rxbuf++;
    }
    TX_buffer_empty = 0;
    RX_buffer_full = 0;
}
```

Direct Memory Access

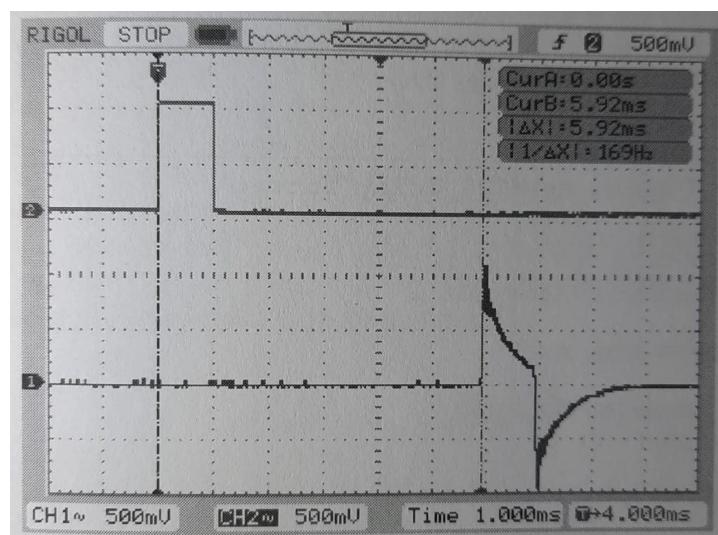
- For this example, function main sets up the peripheral communication and **waits**
- When RX buffer is full and TX buffer is empty...
 - Signals via GPIO (for measurement)
 - Sends buffer to be “processed”
- Frame processing, then, takes place within process_buffer()

```
int main(void)
{
    stm32_wm5102_init(FS_48000_HZ,
                        WM5102_LINE_IN,
                        IO_METHOD_DMA);

    while(1)
    {
        while (!(RX_buffer_full && TX_buffer_empty)){}
        GPIO_SetBits(GPIO_D, GPIO_Pin_15);
        process_buffer();
        GPIO_ResetBits(GPIO_D, GPIO_Pin_15);
    }
}
```

Direct Memory Access

- At a sampling rate of 48KHz, and a buffer size of 256 (128 per channel), the corresponding delay would be $256/48000 = 5.3\text{ms}$
- Measured: 5.92ms



Direct Memory Access

- Say we wish to implement a DMA-based filter (a convolution)
 - Main is unchanged, IRQs unchanged
 - When buffers are filled and swap/flip, action happens in process_buffer()

```
for (i = 0; i < (BUFSIZE/2) ; i++)
{
    y[i] = 0.0;
    x[k++] = (float32_t)(prbs(8000));

    // x[k++] = rdbuf++;
    // rdbuf++;

    if (k >= N) k = 0;
    for (j=0 ; j<N ; j++)
    {
        y[i] += h[j]*x[k++];
        if (k >= N) k = 0;
    }
}
```

Direct Memory Access

- In process_buffer()
 - Input is brought in
 - Here: a pseudo-random sequence
 - (**commented out**) one of the inputs from the receive buffer (that is, data from ADC)
 - Convolution happens between the input BUFFER and a given sequence $h[]$.
 - An output array is filled up ($y[]$)

```
for (i = 0; i < (BUFSIZE/2) ; i++)
{
    y[i] = 0.0;
    x[k++] = (float32_t)(prbs(8000));

    // x[k++] = rdbuf++;
    // rdbuf++;

    if (k >= N) k = 0;
    for (j=0 ; j<N ; j++)
    {
        y[i] += h[j]*x[k++];
        if (k >= N) k = 0;
    }
}
```

Direct Memory Access

- After the convolution is done
 - resulting array $y[]$ needs to be transferred to the output (Transmit) buffer
 - the DMA transfer is completed
 - TX empty and RX full flags will indicate the need to flip ping→pong

```
for (i=0 ; i<(BUFSIZE/2) ; i++)  
{  
    *txbuf++ = (short)(y[i]);  
    *txbuf++ = (short)(y[i]);  
}  
  
TX_buffer_empty = 0;  
RX_buffer_full   = 0;
```

Direct Memory Access

- Same can be applied to a variety of algorithms
 - Is it possible to apply this strategy to the frame you have from a camera, along with some processing of the image?
 - Frequency domain filtering
 - Done for very large filters, when there is a need to convert to frequency domain, perform multiplication and convert back to time domain.
 - Any algorithm that involves a prior collection of a relatively large amount of data in real-time, prior to processing
 - The question to be asked is: can I fit the processing within the time constraints?

Processing Unit

Processing Unit

- Measures of performance
- Say we run a program on **2 desktop computers**
- “**Fastest**” → gets the job done first
 - Related to “**response time**” or “**execution time**”
- Say we have a **data centre with servers**, running many jobs from many users
- “**Fastest**” → completes most jobs in a day
 - Related to “**throughput**” or “**bandwidth**”

Processing Unit

- Response/Execution Time
 - Total time to complete a task
 - Disk access, memory access, I/O operation, OS-related overhead
- Throughput
 - Number of tasks completed per unit time
 - (in communication is data units per time)
- Typically maximizing performance → minimizing execution time
 - Then we can compare systems via “performance ratio”
- TIME is typically the measure of performance

Processing Unit

- Using TIME can be tricky; which time are we measuring?
- CPU Execution time
 - Many tasks are taking place as CPU executes a program
 - Which tasks can give me “seconds per program” ?
- Want to target the time that CPU is actually computing a task
 - USER CPU Time – time spent on program
 - SYSTEM CPU Time – OS-related tasks
- Will concentrate on time spent on program
- “Time” relates to CPU clock
 - Cycles / Frequency / “ticks”
 - Ex: 4GHz clock → 250ps clock cycle

Processing Unit

- Let's define performance

$$\text{CPU Execution Time} = \# \text{ CPU Clock cycles for the program} \times \text{Clock cycle time}$$

Can you speed this up?

Processing Unit

- Performance is:

$$\text{CPU Execution Time} = \# \text{ CPU Clock cycles for the program} \times \text{Clock cycle time}$$

$$\# \text{ CPU Clock cycles for the program} = \# \text{ instructions for the program} \times \text{Avg # Clock cycles per instruction}$$

That depends on you
(programmer), the language
chosen, the compiler...

Depends on the
architecture

Processing Unit

- The classic CPU performance equation to compare two systems is

$$\text{CPU Time} = \text{Instruction Count} \times \text{Cycles per Instruction} \times \text{Cycle Time}$$
$$(IC) \qquad \qquad \qquad (CPI)$$

By using this I can compare the performance of two different implementations

Using only one or two of these factors (as it is often done) is misleading

Processing Unit

- Program Performance depends on
 - Algorithm
 - Language
 - Compiler
 - Architecture of the hardware
- The Algorithm
 - Will affect instruction count (IC) and CPI
 - Number of processor instructions executed
 - Number of lines of code
 - Example: an algorithm with multiple divisions → more cycles

Processing Unit

- The programming language
 - Affects instruction count and CPI as well
 - Performance depends on how it is translated
 - Less processor instructions performs better
 - Will determine the level of data abstraction; how data is called and retrieved
- Compiler
 - Affects instruction count and CPI
 - The compiler IS the translator
 - It can make things a lot better with optimization and pipelining

Processing Unit

- The Instruction Set Architecture
 - Will affect the CPI and clock rate
 - Determines how long – in terms of cycles – instructions will take to get executed
 - Ex: Compiler designer has to make a decision regarding two code sequences

	Instruction A	Instruction B	Instruction C
Sequence 1	2	1	2
Sequence 2	4	1	1
CPI	1	2	3

Processing Unit

- Questions to be asked

	Instruction A	Instruction B	Instruction C
Sequence 1	2	1	2
Sequence 2	4	1	1

	Instruction A	Instruction B	Instruction C
CPI	1	2	3

- Which sequence executes the most instructions?
- Which sequence is faster?
- What is the CPI for each sequence?

Processing Unit

- Which sequence executes the most instructions? (IC)

Sequence A → 2 + 1 + 2 instructions, to a total of 5
Sequence B → 4 + 1 + 2 instructions, to a total of 6

- Which sequence is faster? (in total cycle count)

$$\sum (\text{CPI} \times \text{count})$$

Sequence A → $2 \times 1 + 1 \times 2 + 2 \times 3 = 10$
Sequence B → $4 \times 1 + 1 \times 2 + 1 \times 3 = 9$

It executes more instructions but it is faster!

Processing Unit

- Which sequence executes the most instructions? (IC)

Sequence A → 2 + 1 + 2 instructions, to a total of 5
Sequence B → 4 + 1 + 2 instructions, to a total of 6

- Which sequence is faster? (in total cycle count)

$$\sum (\text{CPI} \times \text{count}) \quad \begin{array}{l} \text{Sequence A} \rightarrow 2 \times 1 + 1 \times 2 + 2 \times 3 = 10 \\ \text{Sequence B} \rightarrow 4 \times 1 + 1 \times 2 + 1 \times 3 = 9 \end{array}$$

- What is the CPI?

$$\text{CPI} = (\text{total Cycle Count})/\text{IC} \quad \begin{array}{l} \text{CPI Sequence A} \rightarrow 10/5 = 2 \\ \text{CPI Sequence B} \rightarrow 9/6 = 1.5 \end{array}$$

Processing Unit

- Consider two implementations of the **same** program using the same ISA (Instruction Set Architecture)
 - Computer A, clock cycle = 250ps, CPI of 2.0
 - Computer B, clock cycle = 500ps, CPI of 1.2
- At first glance, A → 4GHz, and B → 2GHz (so... A is faster)
- If **I** is the number of instructions for that one program being run, I know that it should be the same for both
 - but I also know that the CPI is not the same.

Processing Unit

- I can calculate my CPU (execution) time in terms of **I**

For Computer A, CPUtime = $I \times 2 \times 250 = 500 \text{ I}$

For Computer B, CPUtime = $I \times 1.2 \times 500 = 600 \text{ I}$

- Execution time for B = $1.2 \times$ Execution time for A
- **A performs better than B**
 - Smaller execution time, better performance

Processing Unit

- It was mentioned earlier that using time alone as a performance metric could be misleading
- The alternative is to use MIPS
 - This is a measure of program execution speed based on “millions of instructions per second”
 - The idea is: faster computers → greater MIPS metric

$$\text{MIPS} = \frac{\text{Instruction Count}}{(\text{Execution time} \times 10^6)}$$

Processing Unit

- Expanding, we have

$$\text{MIPS} = \frac{\frac{\text{Instruction Count}}{\text{Clock rate}} \times 10^6}{\text{Instruction Count} \times \text{CPI}} = \text{Clock rate} / (\text{CPI} \times 10^6)$$

- Exercise

	Computer A	Computer B
Instruction Count	10 billion	8 billion
Clock freq	4GHz	4GHz
CPI	1.0	1.1

Processing Unit

- The Processor
 - (243) “The processor can perform whatever task is needed, by executing programs. It can read data from an external device, write data to a device, compute calculation”
 - We saw that it also participates in the management of devices and the bus.
 - Two main parts:
 - **Datapath** – registers holding data currently in use, and ALU for calculation
 - **Control** – FSM that controls transfers of data in/out of the processor (via registers), data transfers between registers and control of the ALU in each clock cycle

ECE342 – Computer Hardware

Lecture 21

Bruno Korst, P.Eng.

Agenda

- Processing Unit
 - Measures of performance
 - Instruction processing and CPU components

Processing Unit

- Consider two implementations of the **same** program using the same ISA (Instruction Set Architecture)
 - Computer A, clock cycle = 250ps, CPI of 2.0
 - Computer B, clock cycle = 500ps, CPI of 1.2
- At first glance, A → 4GHz, and B → 2GHz (so... A is faster)
- If **I** is the number of instructions for that one program being run, I know that it should be the same for both
 - but I also know that the CPI is not the same.

Processing Unit

- I can calculate my CPU (execution) time in terms of **I**

For Computer A, CPUtime = $I \times 2 \times 250 = 500 I$

For Computer B, CPUtime = $I \times 1.2 \times 500 = 600 I$

- Execution time for B = 1.2 x Execution time for A
- **A performs better than B**
 - Smaller execution time, better performance

Processing Unit

- It was mentioned earlier that using time alone as a performance metric could be misleading
- The alternative is to use MIPS
 - This is a measure of program execution speed based on “millions of instructions per second”
 - The idea is: faster computers → greater MIPS metric

$$\text{MIPS} = \text{Instruction Count} / (\text{Execution time} \times 10^6)$$

Processing Unit

- Expanding, we have

$$\text{MIPS} = \frac{\frac{\text{Instruction Count}}{\text{Clock rate}}}{\frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock rate}}} \times 10^6 = \text{Clock rate} / (\text{CPI} \times 10^6)$$

- Exercise

	Computer A	Computer B
Instruction Count	10 billion	8 billion
Clock freq	4GHz	4GHz
CPI	1.0	1.1

- MIPS(A) greater

Processing Unit

- The Processor
 - (243) “The processor can perform whatever task is needed, by executing programs. It can read data from an external device, write data to a device, compute calculation”
 - We saw that it also participates in the management of devices and the bus.
 - Two main components:
 - **Datapath** – involves registers holding data currently in use, and ALU for calculation some muxes and interconnect
 - **Control** – FSM that controls transfers of data in/out of the processor (via registers), data transfers between registers and control of the ALU in each clock cycle

Processing Unit

- We wish now to understand the different stages of instruction processing
- We will describe the functionality of different parts, put them together and describe how the CPU works for different instructions
- For now, they are
 - Register File
 - ALU
 - Datapath (how they connect)
 - Under the fetch stage
 - Under the execution stage

Processing Unit

Real version, but relatively simple

Simplified version

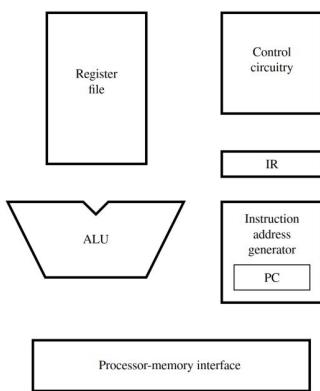
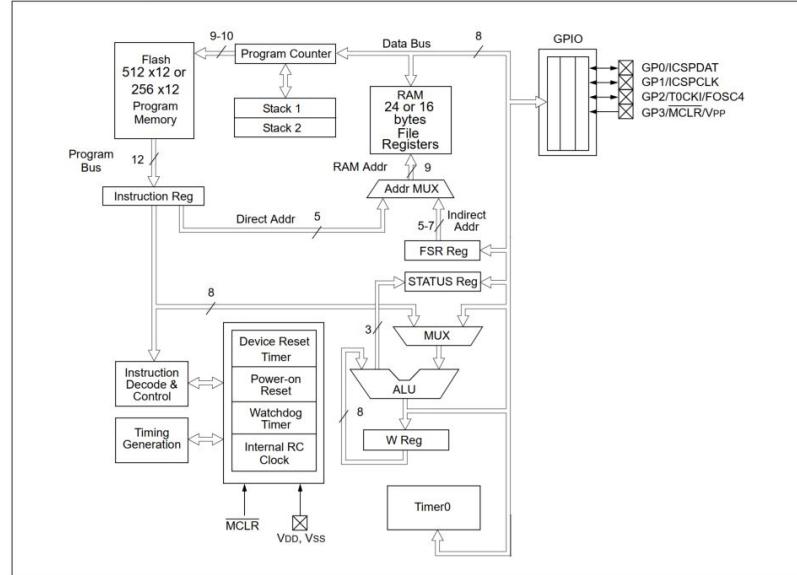


FIGURE 3-1: PIC10F200/202 BLOCK DIAGRAM

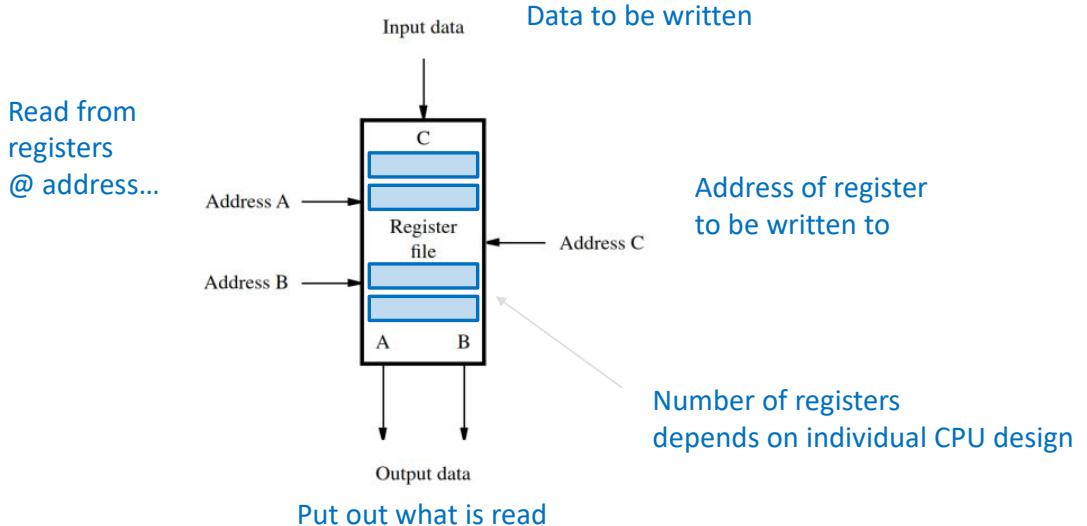


Processing Unit

- Register File
 - Small and fast memory block
 - Array of storage elements with access circuitry to read/write
 - Circuitry designed to read 2 at a same time
 - 2 address inputs connected to Instruction Register (IR – instruction contains addresses)
 - These are sources – where to read
 - 2 other inputs: one data input and one address input
 - Destination – where to write

Processing Unit

- Register File



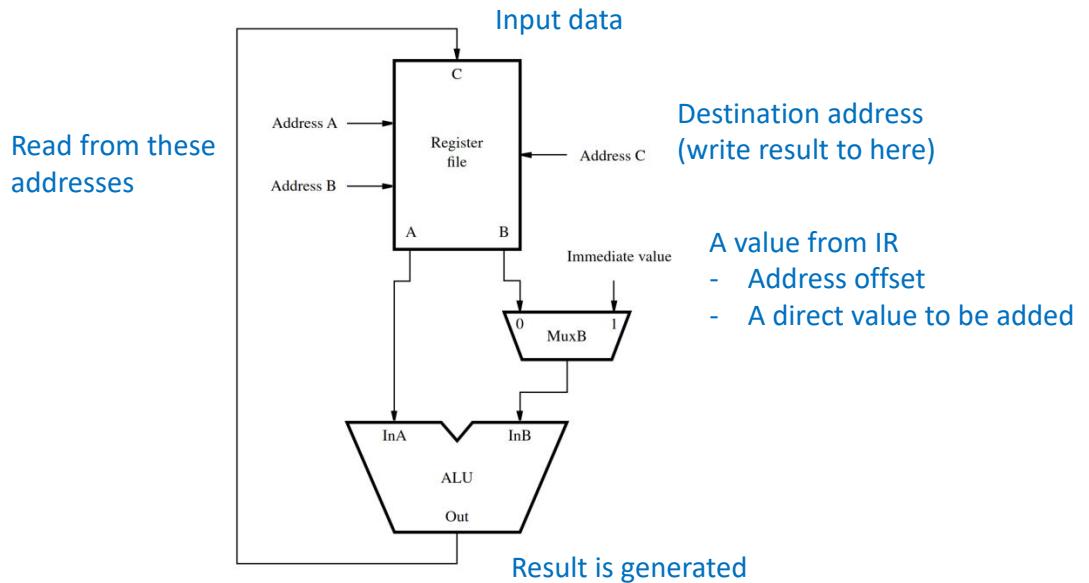
Processing Unit

- ALU (Arithmetic and Logic Unit)

- It is used to manipulate data
 - Arithmetic operations (ADD/SUB)
 - Logic operations (AND/OR/XOR)
- When an instruction is **executed**
 - Contents of the two registers (in the instruction) are read from the register file and made available @ outputs
 - MUX may specify a direct value or address offset prior to input to ALU

Processing Unit

- In the context of an operation...



Processing Unit

- Datapath
 - Instruction processing consists of 2 phases
 - Fetch phase
 - Execution phase
 - Fetch
 - “Fetch & Decode” the instruction
 - Generate control signals that result in appropriate actions during the execution phase
 - Execution
 - Read data operands, perform operation, stores results

Processing Unit

- Let's look at a **RISC style** instruction
 - All follow a 5 step sequence of actions
 - It is one word long
 - Only LOAD and STORE instructions access operands in memory
 - Computations take data from
 - General purpose registers, or...
 - Direct/immediate value coming from the instruction

Processing Unit

- RISC instruction stages
 - 1. Fetch instruction and increment PC
 - 2. Decode instruction and read registers from register file
 - 3. Perform arithmetic/logic operation
 - 4. Read from or write to memory (if needed)
 - 5. Write result in destination register

Later on we'll see IF-ID-EX-MEM-WB

Processing Unit

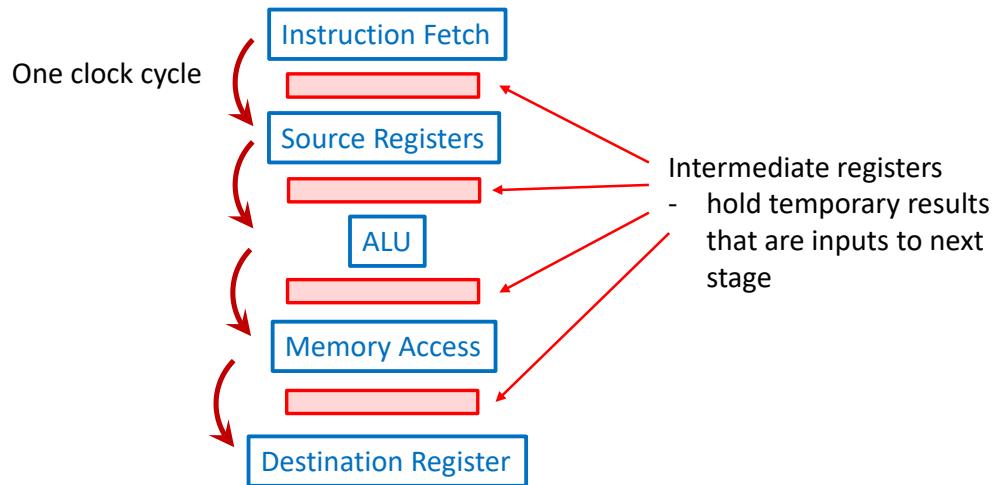
- Example
 - LOAD R5, x(R7)
 - Memory location $x + R7$ loaded into R5
 - 1) Fetch and increment PC
 - 2) Decode and read R7
 - 3) Compute address (ALU)
 - 4) Read memory
 - 5) Load it into R5

Processing Unit

- Example
 - ADD R3, R4, R5
 - Add contents of R4 to contents of R5, put in R3
 - Note: these are not in memory → no step 4
 - 1) Fetch, increment PC
 - 2) Decode, read R4 and R5
 - 3) Perform addition R4+R5
 - 4) do nothing
 - 5) Load result into R3

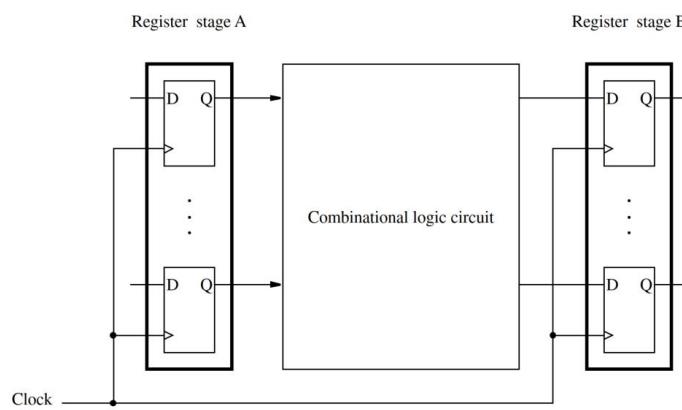
Processing Unit

- If the instructions operate this way, the hardware is organized accordingly



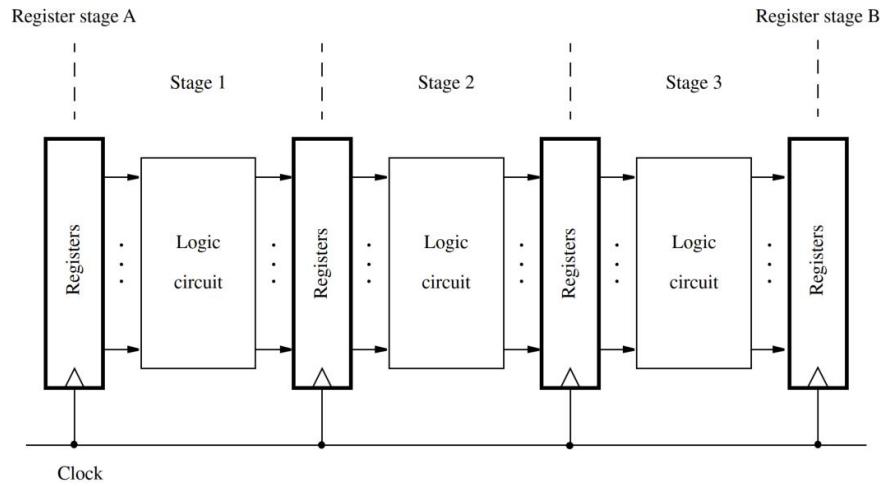
Processing Unit

- Let's pause for a moment
 - "stored" → registers (edge-triggered FF)
 - "processed" → combinational logic
 - "transferred" → clock tick (how fast?)



Processing Unit

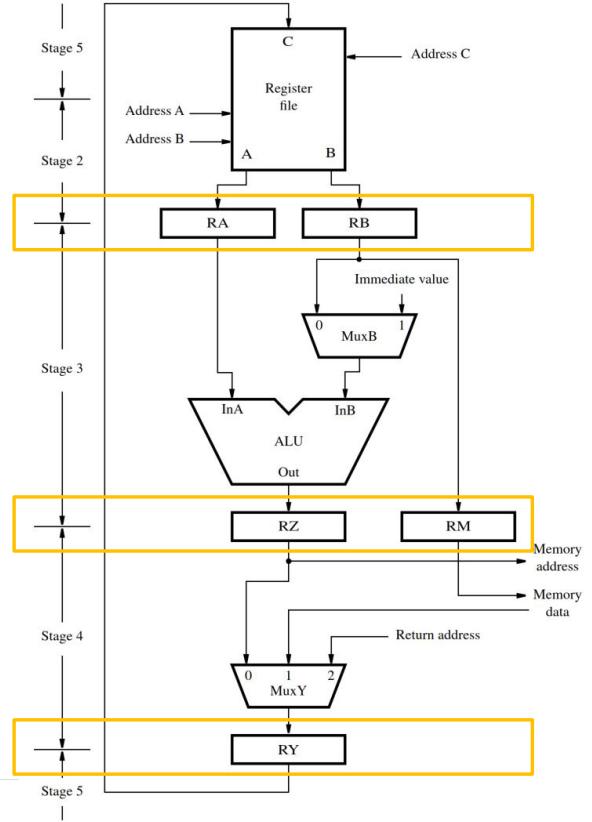
- This process can also be broken down further: stages within stages



Processing Unit

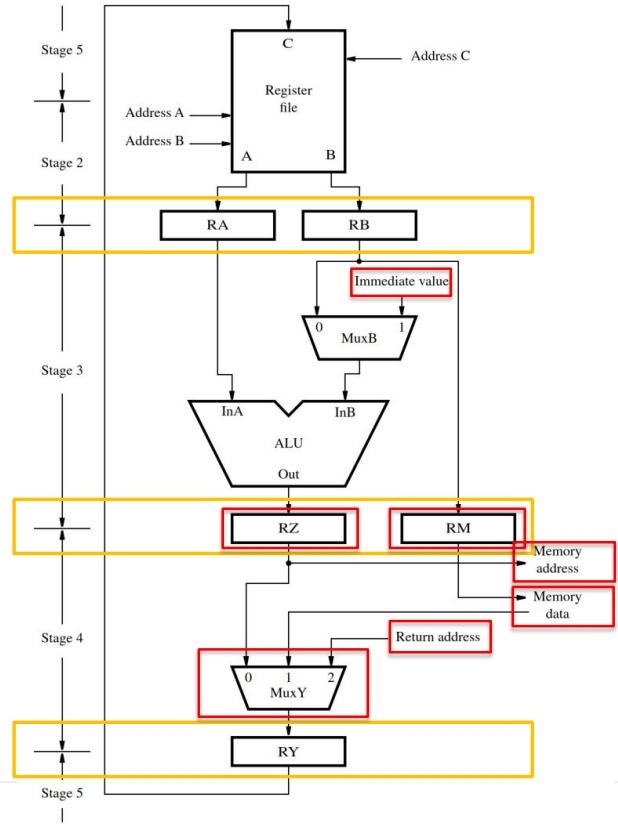
- Adding registers to datapath

Note: fetch/decode has been done
(stage 1)
Stage 2 – source registers
Stage 3 – ALU
Stage 4 – Mem access (if needed)
Stage 5 – writes RY into Register File



Processing Unit

- Immediate value
 - Comes from IR
 - It is within the instruction
- RZ can be number or address
 - Sent to MEM or moved on to RY
 - Ex: instruction LOAD or STORE
- RM data to be written to MEM
- Return address
 - Return from subroutine
- MUXY selector examples
 - Instruction ADD – zero
 - Instruction LOAD – one
 - Return from subroutine - two



Processing Unit

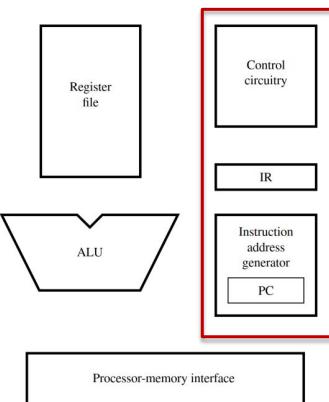
- Datapath: Fetch
 - When **fetching instructions**
 - Address comes from PC (program counter)
 - Instruction is read from memory, placed in IR
 - IR – instruction register
 - Instruction address generator updates PC
 - PC is part of the instruction address generator
 - Instruction stays in IR until execution completed
 - Next instruction is fetched

Processing Unit

- Datapath: **Fetch** (cont'd)
 - When **fetching operands**
 - Address comes from register RZ
 - MUX selects what goes to processor-memory interface
- Fetch circuitry
 - Involves an **instruction address generator** (with the program counter)
 - Involves **control circuit** to **generate signals to other hardware in processor**

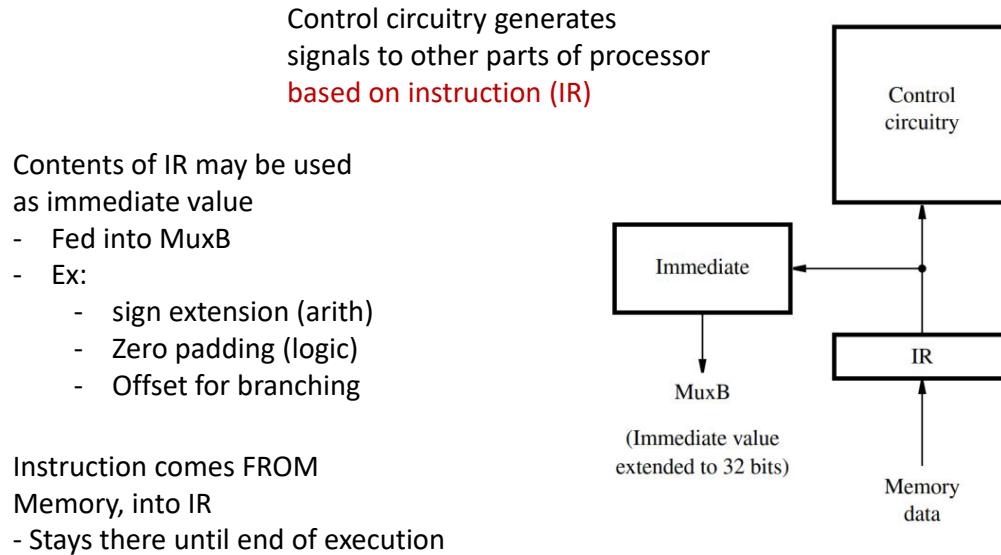
Processing Unit

- Fetch circuitry
 - Involves an **instruction address generator** (with the program counter)
 - Involves **control circuit** to **generate signals to other hardware in processor**



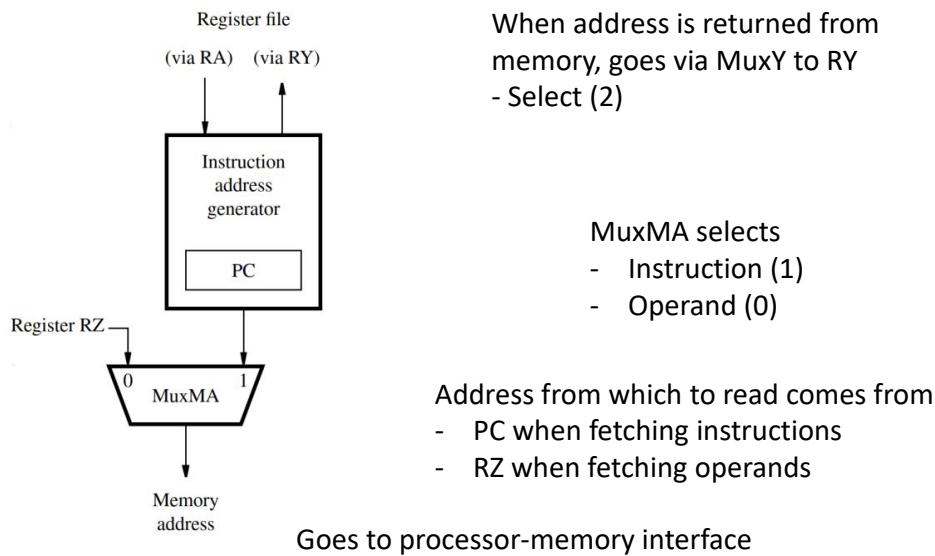
Processing Unit

- Control circuitry OF fetch section



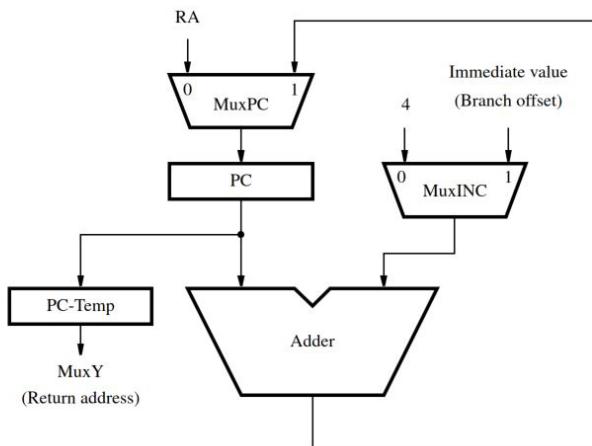
Processing Unit

- Instruction fetch section



Processing Unit

- Going further into the Address Generator box



MuxINC

- Increments PC by 4 (next address)
- Increments PC by branch offset (subroutine call)

Offset comes from immediate field within the IR, extended by the "Immediate" block

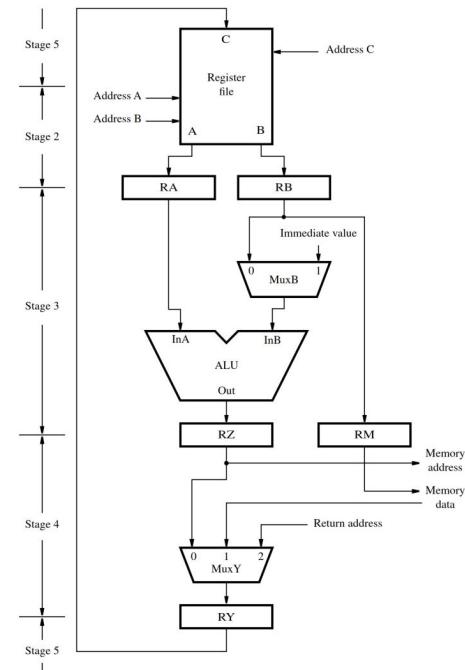
Output routed to PC via MuxPC

RA – subroutine linkage

PC Temp holds return address

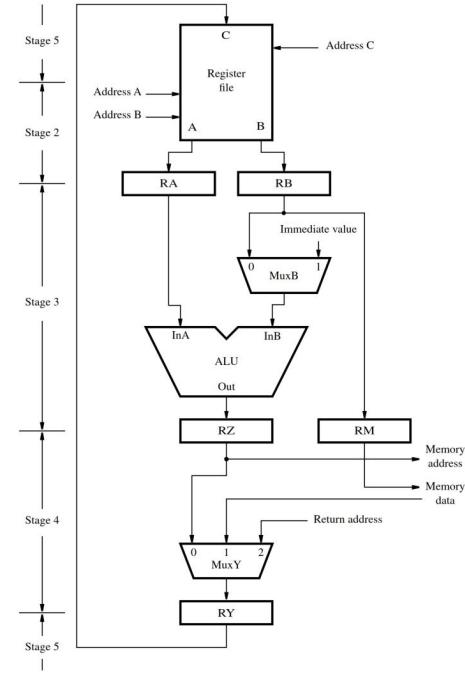
Processing Unit

- Example: ADD R3, R4, R5
 - 1) mem addr \leftarrow PC
 - Read memory
 - IR \leftarrow Memory data
 - PC \leftarrow PC + 4
 - 2) Decode instruction
 - RA \leftarrow [R4]
 - RB \leftarrow [R5]
 - 3) Operate
 - RZ \leftarrow [RA] + [RB]



Processing Unit

- Example: ADD R3, R4, R5
 - 4) $RY \leftarrow [RZ]$
 - 5) $R3 \leftarrow [RY]$
 - Destination register (in register file)



Processing Unit

- In general
 - Reading from memory takes a lot longer than from the register file
 - Keep a cache nearby – close and fast (particularly if data needed is in the cache...)
 - Branch instructions
 - Branch offset has limited # of bits on the “immediate”
 - Limits the memory to be accessed.

ECE342 – Computer Hardware

Lecture 22

Bruno Korst, P.Eng.

Agenda

- Processing Unit
 - Hardware components cont'd
 - Control
 - Instruction Set

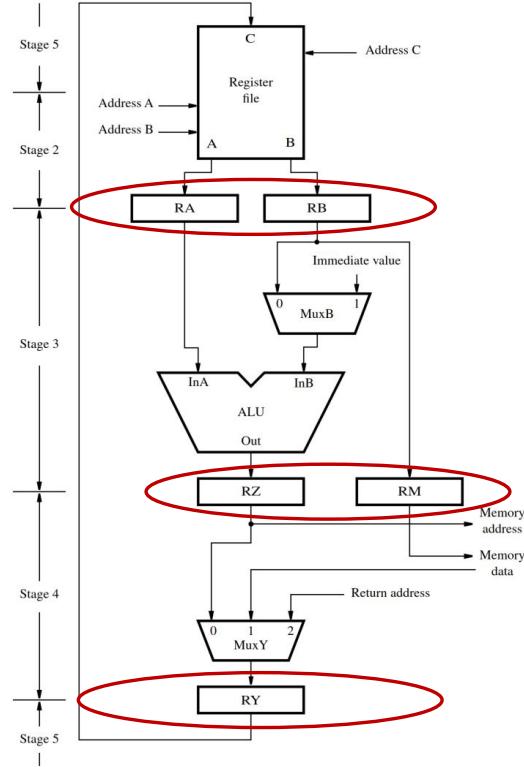
Processing Unit

- We saw
 - Datapath
 - Fetch mechanism – address and control
 - Step-by-step instruction fetch/execute
- Common actions to all instructions
 - Retrieve address from PC (increment), go fetch;
 - Read one or two registers (or memory), write to a register (or memory)

Processing Unit

- Operation of hardware is governed by **control signals**
 - Which multiplexer input is selected
 - Which operation is executed by ALU...
 - At **each clock cycle**, results of actions of one stage are stored to be available for use in next stage
 - Some registers (PC, IR, register file) are NOT changed every cycle (enabled when needed)

Processing Unit

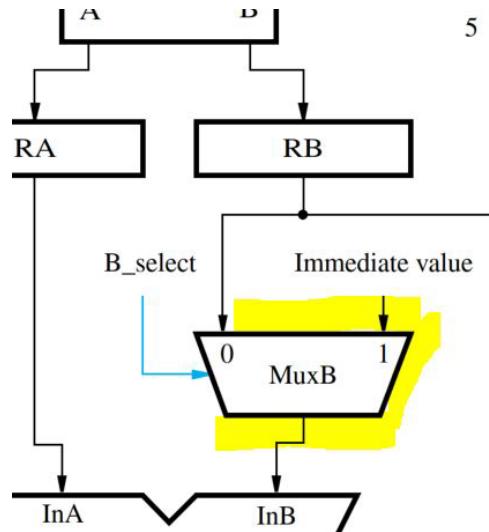
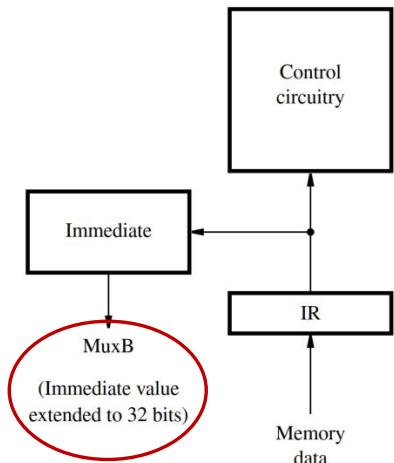


Processing Unit

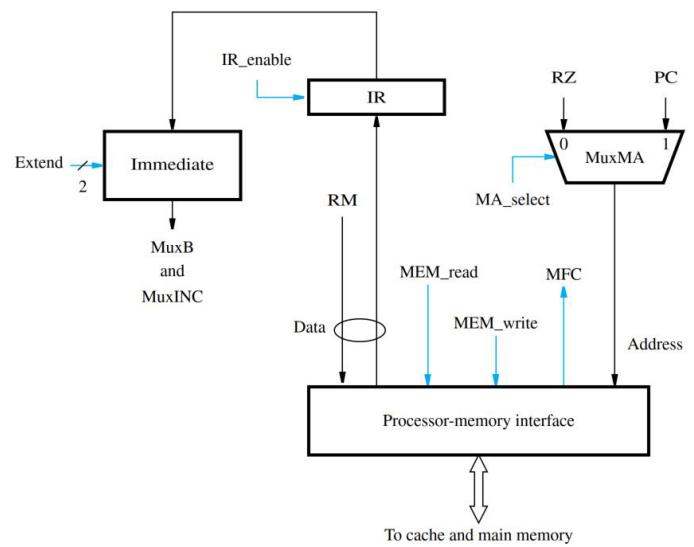
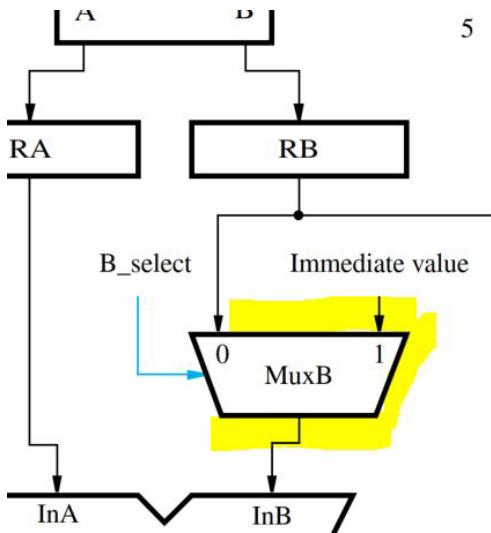
- Will look at some examples for how control signals are used
- 1) consider **MuxB**
 - If an **instruction uses immediate operand**, **MuxB** selects the immediate field in the IR, OR
 - Selects immediate field if instruction uses immediate data OR
 - Selects register RB
- Only needed in step 3
- **MuxB** is needed in step 3 and selection done there does not need to change during next steps. The **Mux control signal** will determine that goes where

Processing Unit

- MuxB – immediate value comes from Instruction Register (IR)



Processing Unit



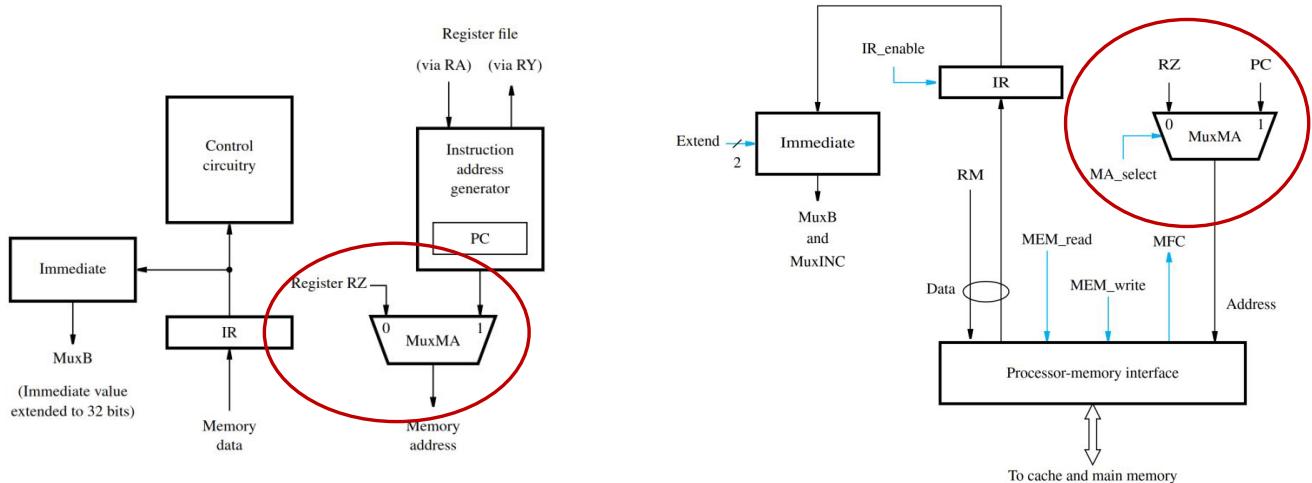
Processing Unit

- 2) Consider MuxMA

- This mux is involved in instruction fetching and in retrieving operands
- Its selection is changed depending at which execution step the instruction is
 - Step 1 – selects PC as source of memory address (for fetching)
 - Step 4 – selects RZ if instruction is load or store
 - RZ has the address of the memory operand

Processing Unit

- Control signals involved in fetching



Note: refer to the large datapath diagram, note that RZ outputs to "Memory Address" and MuxY(input 0)

Processing Unit

- 3) consider a **register file** with 32 general purpose registers
 - Each register is addressed by 5 bits
- The instruction in the IR will determine how its bit fields will be used
- If the instruction is identified as a 3 register instruction (such as ADD)
- Operands are read from 2 source registers, placed in one destination register
 - This is to say that the bit fields are connected to the register file somehow

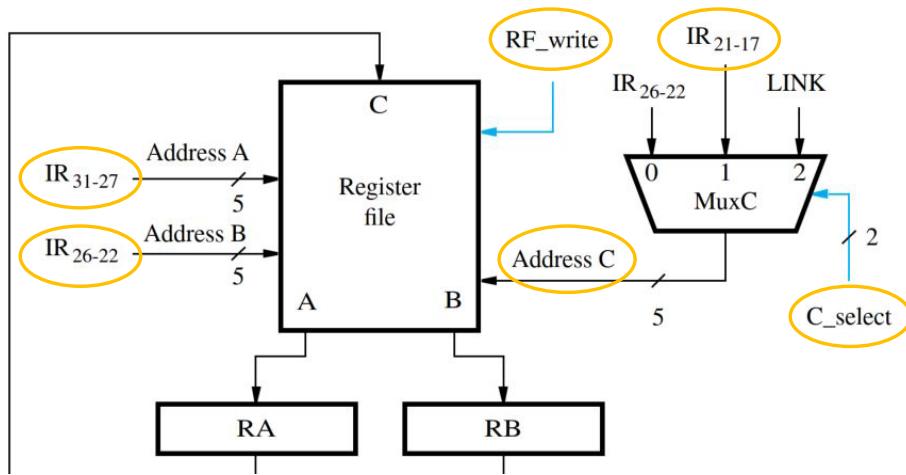
Processing Unit

- An ADD instruction will use registers A and B (sources) and register C (destination) on the register file
 - ADD R3, R4, R5 (destination, source 1, source 2)
 - There must be three 5 bit “fields” within the IR that will hold these addresses

Processing Unit

- On the IR, the addresses can be located at:
 - Instruction register bits IR_{31-27} for address A
 - Instruction register bits IR_{26-22} for address B
 - Instruction register bits IR_{21-17} for address C
- The addresses will be decoded and will point to the correct register within the register file to be read or written
- Control signals are asserted
 - C_select to select the correct destination
 - RF_write to enable writing to destination (write only when asserted)

Processing Unit



Example: ADD instruction (3 registers) will involve these lines

Note:

- Two other lines on MUX for other types of instructions

Processing Unit

- 4) The Memory Interface
 - Control signals initiate read/write
 - MEM_read
 - MEM_write
 - When completed, interface asserts MFC
- It relates to the Instruction Register
 - IR_enable determines when instruction is loaded into IR
 - Since instruction is fetched from memory, IR_enable is activated AFTER MFC asserted

Processing Unit

- 5) The Address Generator

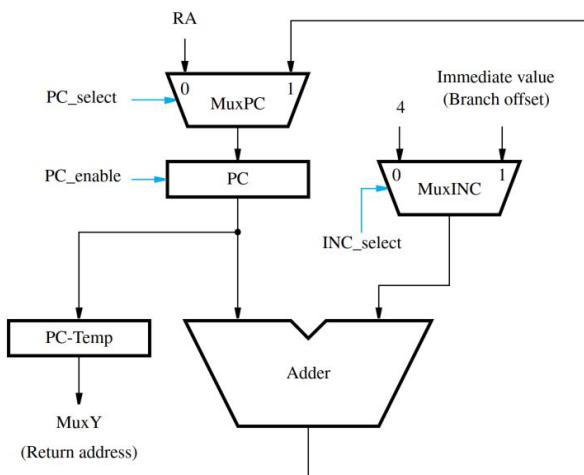
INC_select: value to be added to the PC

PC_select

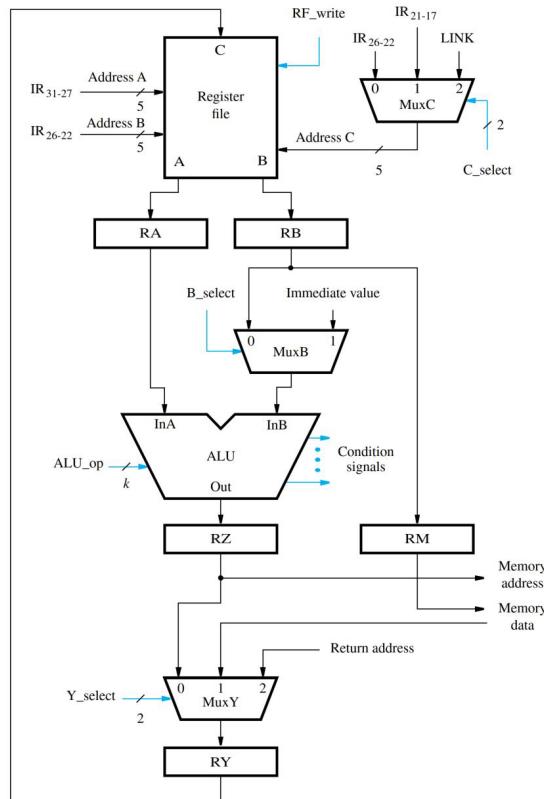
- Value of updated (next) address
- Address coming from RA

PC_enable

- Determines when address is loaded onto PC



Processing Unit



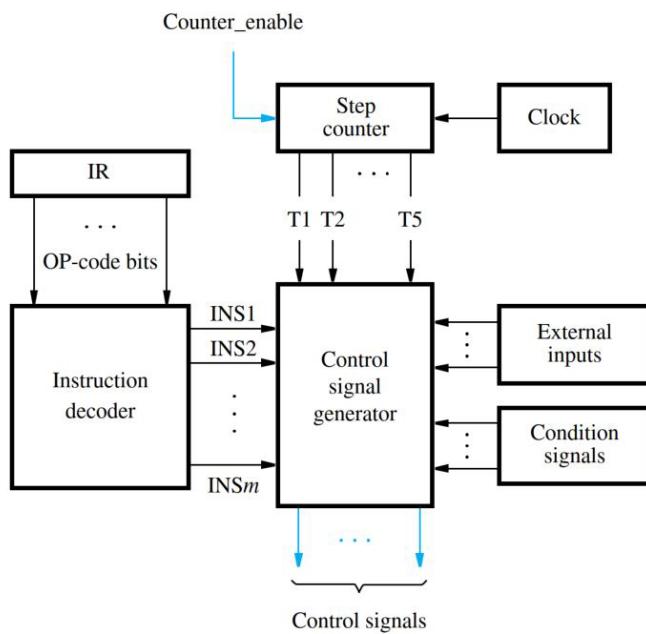
Processing Unit

- Generation of control signals
 - Hardwired or microprogrammed (not generated directly by hardware)
- Different instructions will need a variety of actions
 - Comparison, computation, branches, etc.
- These are done in a sequence of steps, and one needs to keep track of the steps
 - hence the need for the control signals

Processing Unit

- Setting control signals will depend on
 - Contents of step counter
 - That is: where the execution is at any given time
 - Contents of the instruction register
 - Which are the units needed and what are the values
 - What needs to be done
 - The results of a computation/comparison operation
 - External inputs (such as interrupts)

Processing Unit



Decoder looks at OP code
Sets input to generator

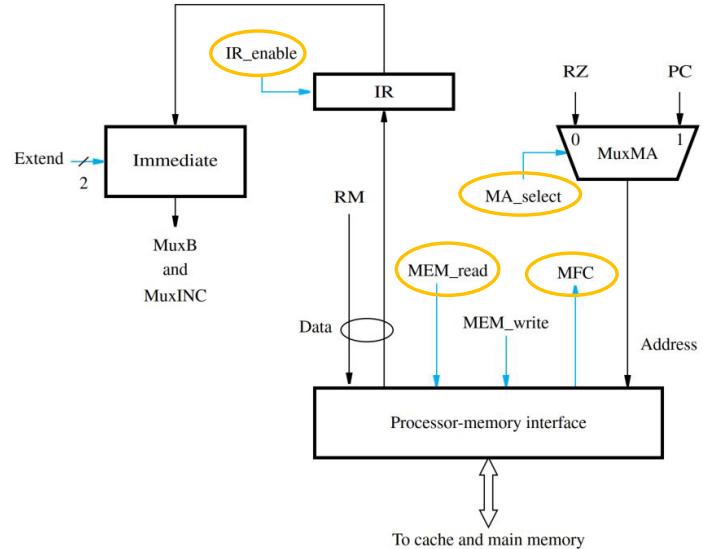
As clock ticks, one T is asserted
(indicates which cycle)

Generator is combinational
logic that takes all inputs into
account and outputs control
signals

Processing Unit

- Example: events in the FETCH stage of execution (step 1)

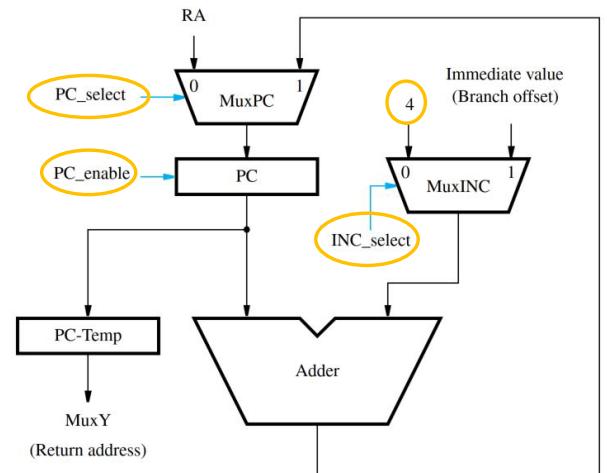
- T1** signal is asserted (step counter)
- During clock period, **MA_select** is up
 - Indicating memory address comes "from PC"
- MEM_read** asserted
- Data received from memory into **IR** by activating **IR_enable** after **MFC** asserted
- At same time, **INC_select** deasserted and **PC_select** asserted
- PC_enable** activated \rightarrow $PC = PC + 4$
 - At clock edge we are at end of STEP 1



Processing Unit

- Example: events in the FETCH stage of execution (step 1)

- T1** signal is asserted (step counter)
- During clock period, **MA_select** is up
 - Indicating memory address comes "from PC"
- MEM_read** asserted
- Data received from memory into **IR** by activating **IR_enable** after **MFC** asserted
- At same time, **INC_select** deasserted and **PC_select** asserted
- PC_enable** activated \rightarrow $PC = PC + 4$
 - At clock edge we are at end of STEP 1



Processing Unit

- The **instruction set** (243) is the vocabulary of the low-level language
- Two historical paradigms:
 - RISC – executed in a multi-stage processor (simple), leading to pipelining
 - CISC – complex instructions, multi-clock, high cycles/sec
 - Problem is that hardware must follow
 - X86 – around 10 years ago, over 300 instructions

Processing Unit

- Instruction classes within the instruction set
 - Arithmetic – add/sub/add immediate
 - Data transfer – load/store (of different sizes)
 - Logical – and/or/nor/and-immediate/shiftL(R)
 - Conditional branch – PC-relative (BE/BNE), comparison
 - Unconditional branch – jump

Processing Unit

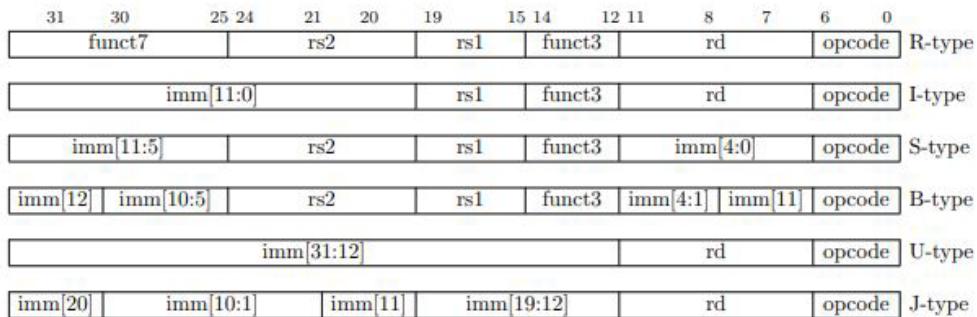
- Different architectures will have different set of instructions and bit field arrangements
 - Ex: ARM (243)
 - Ex: RISC-V
 - In this architecture, operands of arithmetic instructions come from registers only
 - There are 32 registers, doubleword (64 bits)
 - Compiler tries to keep most frequently used variables in registers
 - Recall instructions with “immediate operand” – much faster, less energy

Processing Unit

- We will look at some RISC-V instructions
 - All 32 bits
 - Instruction types
 - R – register
 - S – store
 - I – immediate
 - U – upper immediate (constants are larger)
 - B (SB) – branch
 - J (UJ) – unconditional jump

Processing Unit

- RISC-V Instruction Fields (general)



Opcode – basic operation and format of instruction

Rs2 – second reg. source

Rd – register destination

Func 7 – additional opcode

Func3 – additional Opcode field

Rs1 – first register source operand

Processing Unit

- Examples of instructions
 - Arithmetic
 - Add immediate → ADDI x5, x6, 20
 - $x5 = x6 + 20$
 - Data transfer
 - Load doubleword → LD x5, 40(x6)
 - Load from memory to register
 - $x5 = \text{memory}[x6+40]$
 - Logical
 - And immediate → ANDI x5, x6, 20
 - $X5 = x6 \& 20$

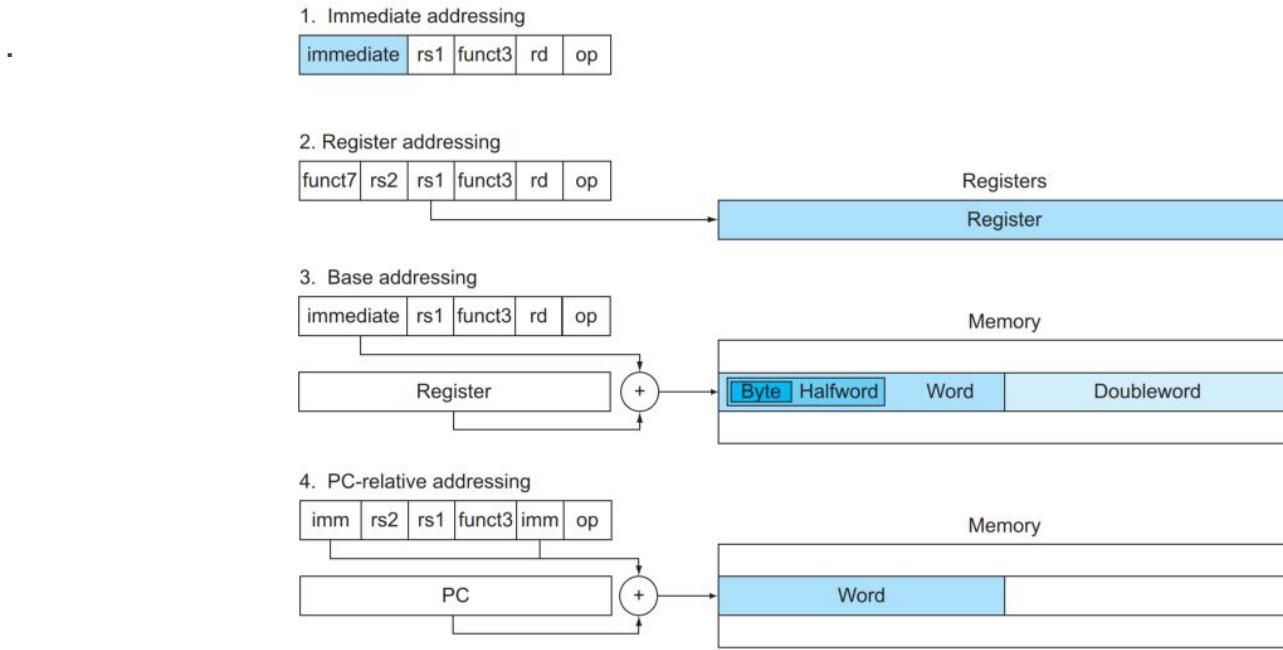
Processing Unit

- Shift
 - Shift left logical → SLL x5, x6, x7
 - “shift left by register” → $x5 = x6 \ll x7$
- Conditional Branch
 - Branch if equal → BEQ x5, x6, 100
 - If ($x5 == x6$), go to PC+100
- Unconditional Branch
 - Jump and link → JAL x1, 100
 - PC relative procedure call
 - $x1 = PC + 4 \rightarrow$ go to PC+100

Processing Unit

- RISC-V addressing modes
 - Immediate – operand is a constant within instruction
 - Register addressing – Operand is in register
 - Base/Displacement addressing – Operand is in memory
 - Address is determined by the sum of the register value AND a constant in the instruction
- PC-relative addressing (branch)
 - Sum of PC and a constant in the instruction

Processing Unit



Processing Unit

- All instructions have two steps in common
 - Fetch via PC (go where the instruction is)
 - Read one or two registers
- Some use ALU
 - Load/store – calculate address
 - Add/subtr – arithmetic or logic operations
 - Branch – compare

Processing Unit

Control for

ADD x3, x4, x5

Control unit
Inputs

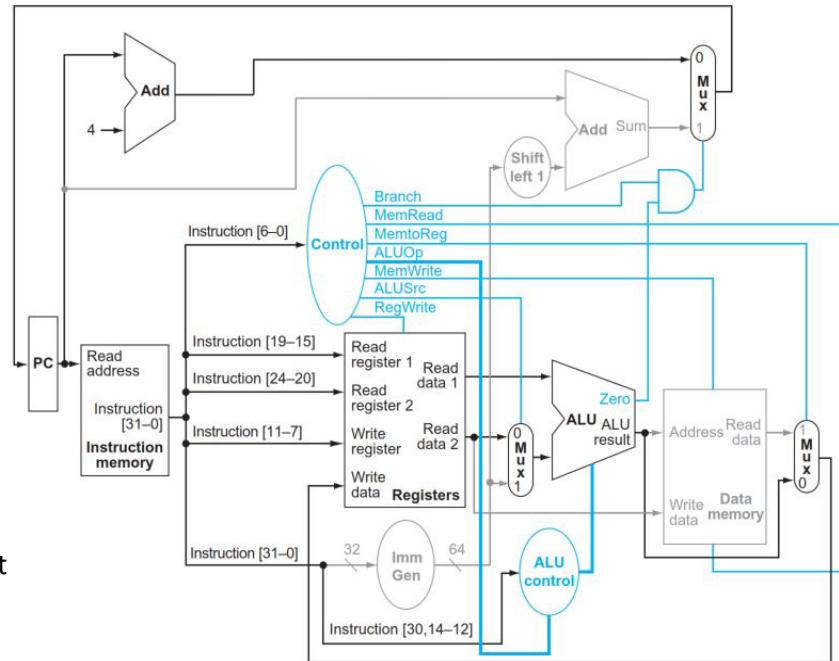
Opcode bits – type of instruction

Will generate

Write signals for each state element

Selector control for each MUX

ALU control



Name (Bit position)	31:25	24:20	Fields	19:15	14:12	11:7	6:0
R-type	funct7	rs2	rs1	funct3	rd	opcode	

Processing Unit

- Datapath

- There are elements that **operate on data**
 - Given same inputs, always the same output
 - These are combinational
 - There are elements that **contain “state”**
 - Internal **storage**
 - If we pull the plug, can be restarted by loading values prior to having pulled the plug
 - Instruction and data memory, and registers
 - Clock** determines when things happen
 - From state element to state element (sequential)

Processing Unit

- If state element not updated on **every clock**, an explicit **control signal** is required
 - State changes when
 - **Control asserted**
 - **Clock edge occurs**
 - Clock needs to be long enough for all inputs to be stable when active clock edge happens
- Datapath design needs
 - How elements will operate
 - How it is to be clocked

Processing Unit

- Some design takeaways from RISC
 - Simplicity favours regularity: keep it simple and regular
 - Smaller is faster
 - Good design is made of good compromises

ECE342 – Computer Hardware

Lecture 23

Bruno Korst, P.Eng.

Agenda

- Processing Unit – wrap
 - Control – single cycle, multi-cycle, ROM-based
- Intro to pipelining

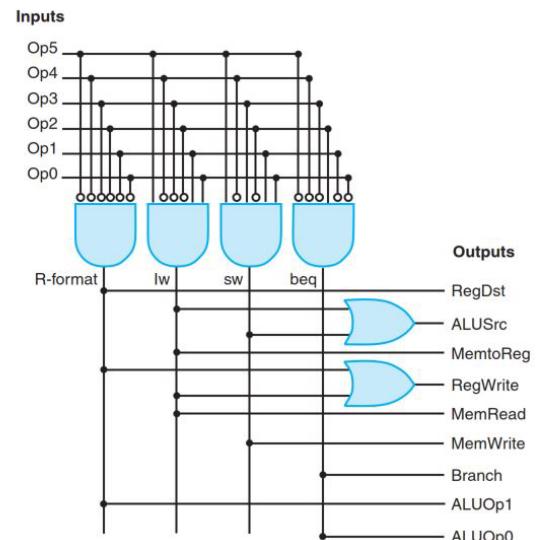
Processing Unit

- We saw
 - Datapath
 - Elements and control signals
 - Instructions
 - Stages of execution, hardware involved
 - RISC-V – instruction set, types and addressing modes
 - If interested, check out RED-V RedBoard and RED-V Thing from SparkFun

Processing Unit

- One cycle control example for an instruction: all logic

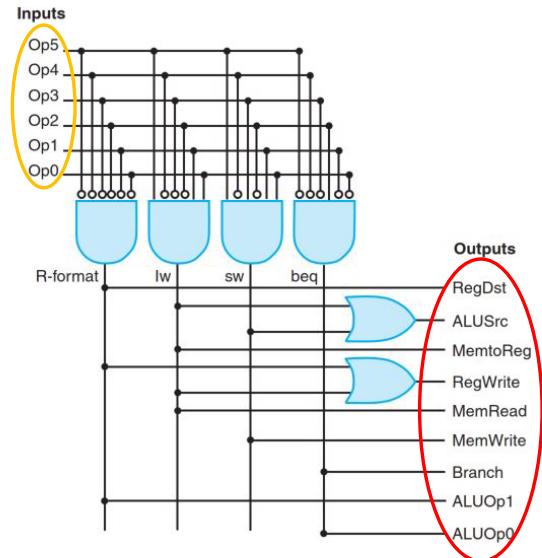
Control	Signal name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
	RegDst	1	0	X	X
Outputs	ALUSrc	0	1	1	0
	MemtoReg	0	1	0	0
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1



Processing Unit

- An example of one cycle control for an instruction: all logic

Control	Signal name	R-format	Iw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

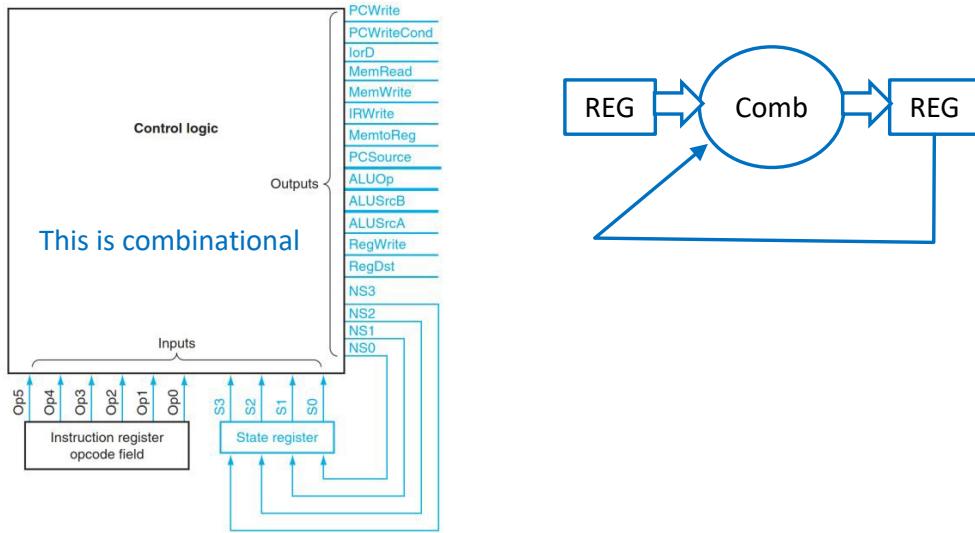


Processing Unit

- For this particular one-cycle controller
 - Input to control unit: 6 bit opcode from instruction
 - Output
 - Two 1-bit signals: mux control
 - ALUSrc and MemToReg
 - Four 1-bit read/write in register file
 - RegDst, RegWrite, MemRead, MemWrite
 - One 1-bit for branch
 - One 2-bit for ALU control
 - ALUop (select operation)

Processing Unit

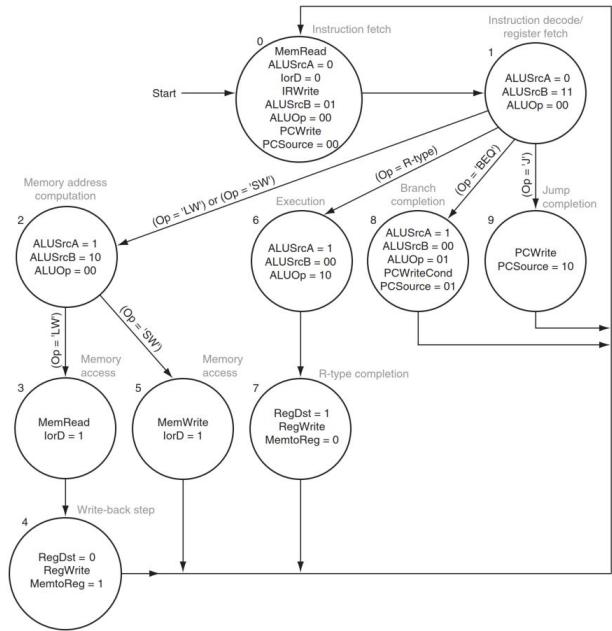
- An example of multi-cycle control via FSM



Processing Unit

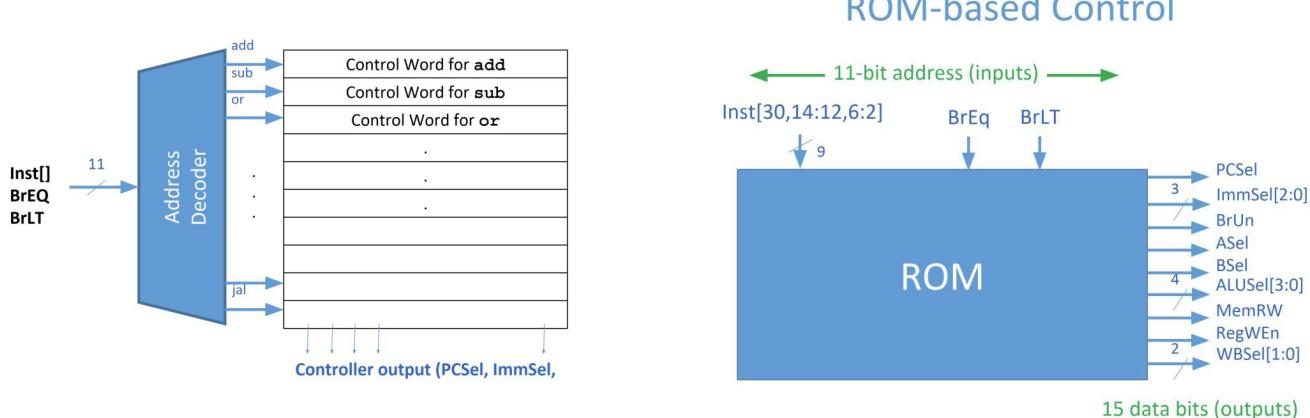
- A more involved multi-cycle control using an FSM

- Outputs based on previous state plus input (Mealy)



Processing Unit

- ROM-based control (microprogrammed) – CISC



Processing Unit

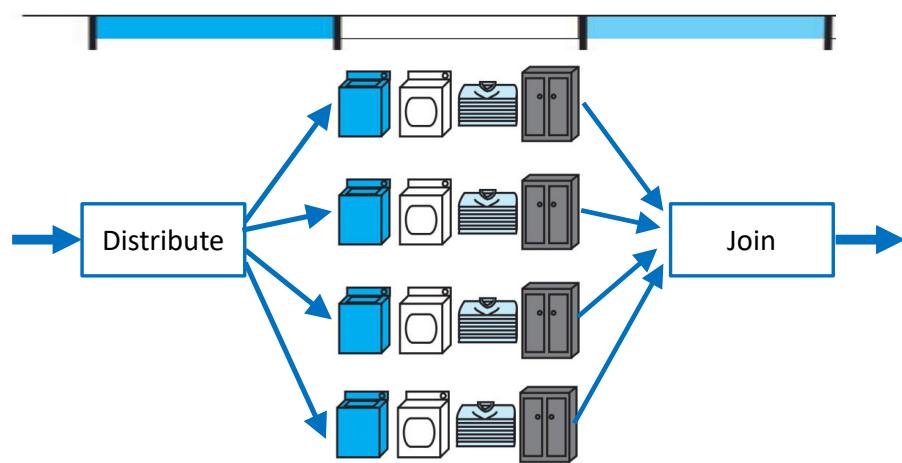
- So far the execution is fairly inefficient: instruction per instruction
- If there is a way to execute instructions simultaneously this **process would be improved**
 - I can **replicate in hardware** all that is needed for each instruction and do things in parallel
 - Or perhaps we can **overlap some of the steps** in a deliberate manner to see if we can improve the process

Pipelining

- Let us look at a well-known step-by-step process: doing laundry
 - 1) We place the clothes in the washer, run the washer
 - 2) We place the wet clothes in the dryer, run the dryer
 - 3) We fold the dried clothes
 - 4) We put them away
- If I have multiple sets of all that is needed, I can do things in parallel
 - This should improve things...

Pipelining

- Maybe I have the resources (\$\$\$) and personnel to do everything at the same time



Pipelining

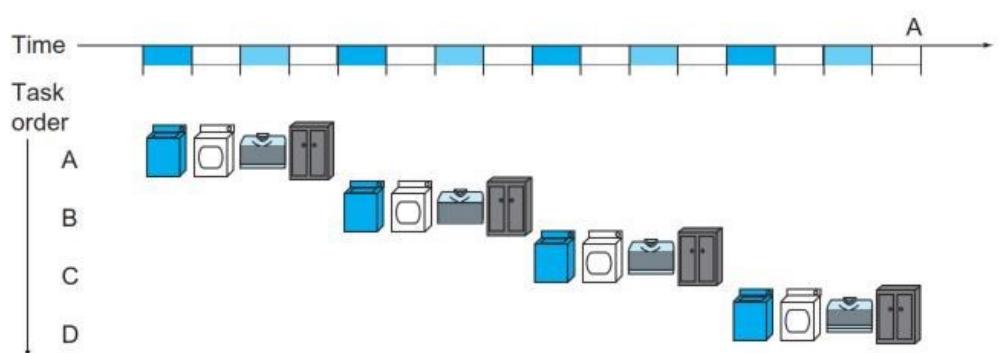
- Looking at the sequence of steps again, there is no need to wait until the end of 4) to place another load in 1). In fact, there is no need to wait until the end of 2)... As soon as the washer is done, another load can go in.
 - 1) We place the clothes in the washer, run the washer
another load can go in here to wash
 - 2) We place the wet clothes in the dryer, run the dryer
the first load is here drying
 - 3) We fold the dried clothes
 - 4) We put them away

Pipelining

- Let's look at the time it takes to get four loads of laundry done, sequentially

Four loads, each step takes 30 mins
 $16 \times 30\text{mins}$

Note that the icons (steps) are repeated, but in reality there is only one of each

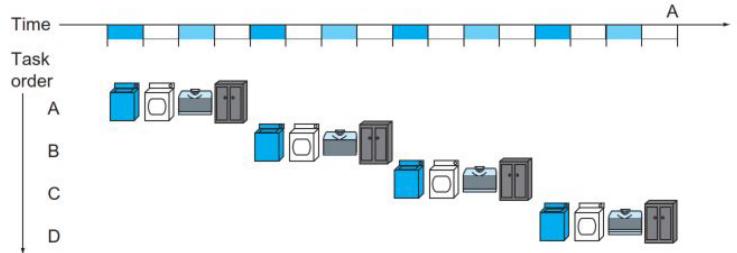


This is similar to a sequential assembly line; one car is finished being built before the next starts

Pipelining

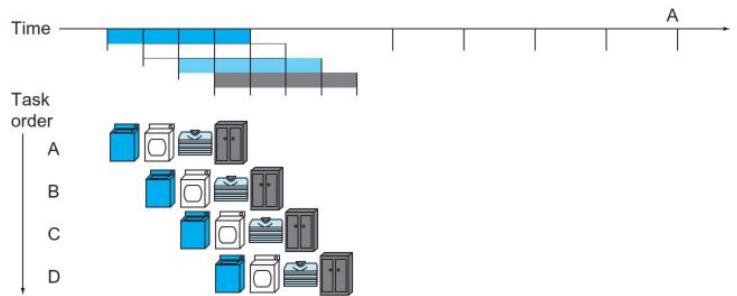
- We can load the washer again (step1) as soon as it is done...
 - This takes 8 hours

Four loads, each step takes 30 mins



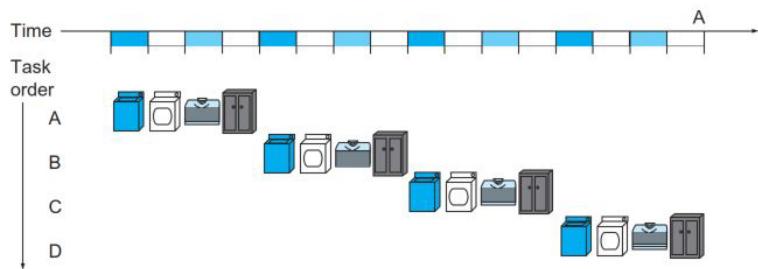
- This takes 3.5 hours

Pipelined



Pipelining

Four loads, each step takes 30 mins



Note:

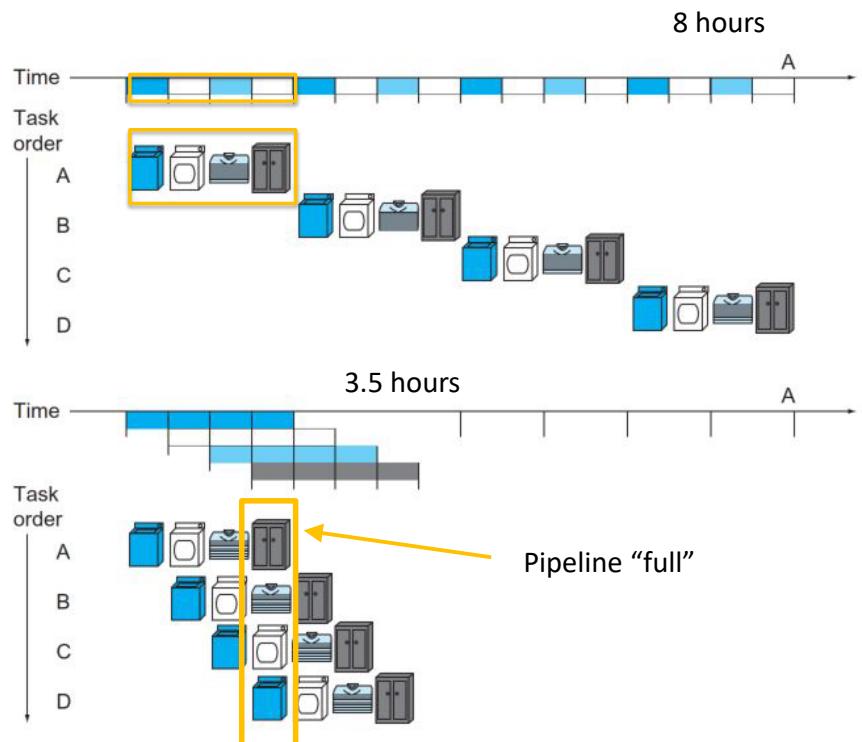
- The time it takes to wash, dry and fold a shirt is the same in both cases
- HOWEVER
- More loads are finished per hour**
- That is: the **THROUGHPUT** of the system is improved

Pipelining

- What is the improvement?
 - There are 4 stages in a load, all take the same time to complete
 - Without pipeline
 - 20 loads would take 20 times longer than one load ($20 \times 4 \times 30\text{min}$)
 - With pipeline
 - 20 loads take 5.75 times longer than one load
 - 1 load takes $4 \times 30\text{min}$, 20 loads take $23 \times 30\text{ min}$
 - Improvement of 20 relative to 1 $\rightarrow 23/4$
- 20 loads improvement: not pipelined takes 3.47 times longer to do 20 loads

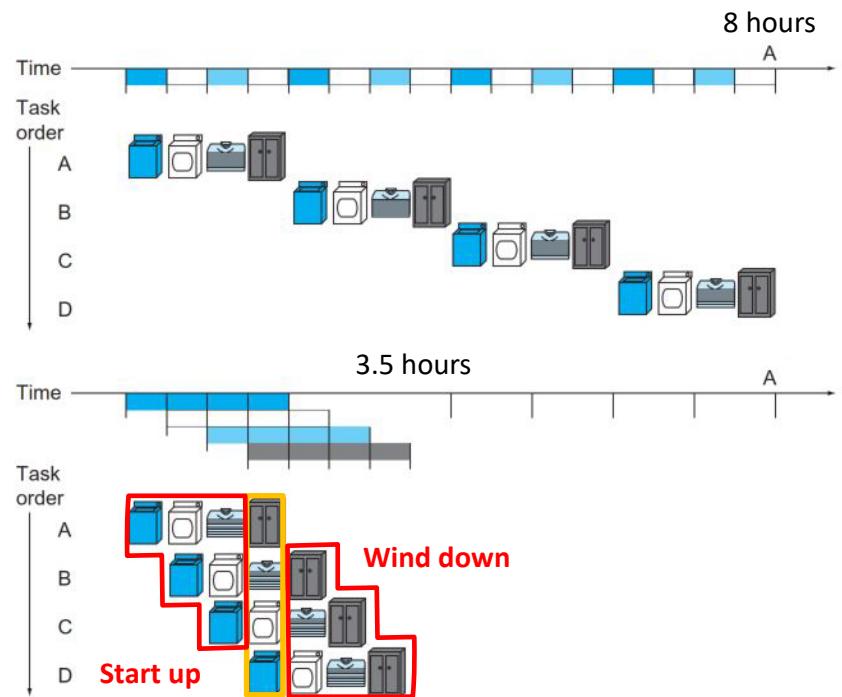
Pipelining

- We call “**latency**” the time it takes to complete a task from beginning to end
- A **pipeline is full** when the last step of the first task aligns with the last step of a future task

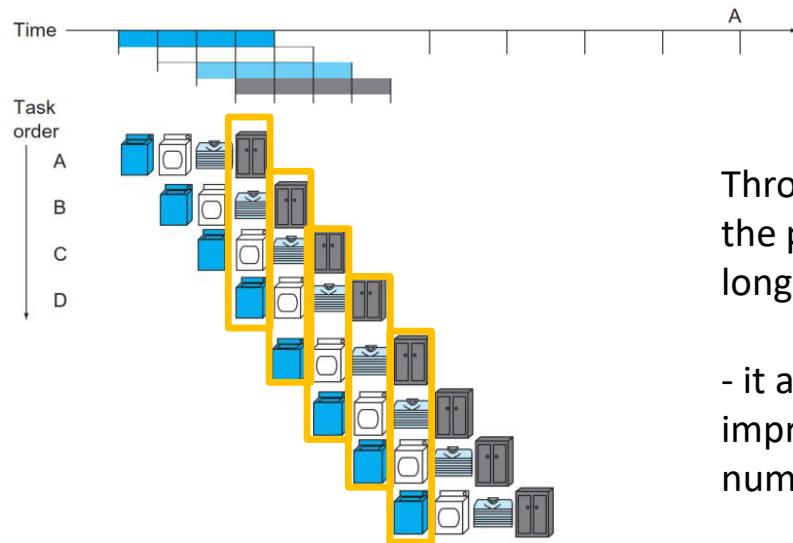


Pipelining

- Pipeline **start up time** is how long it takes for the pipeline to become full.



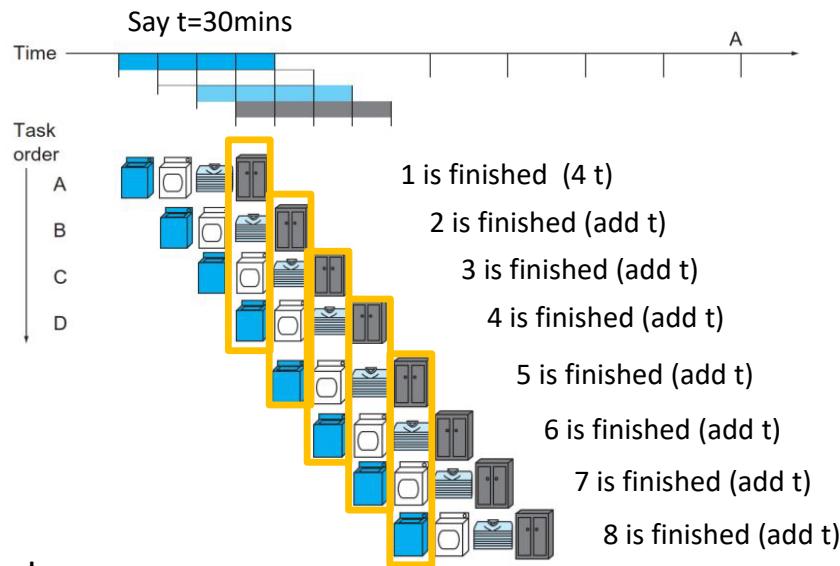
Pipelining



Throughput is better as the pipeline stays full longer

- it approaches an improvement equal to number of stages

Pipelining



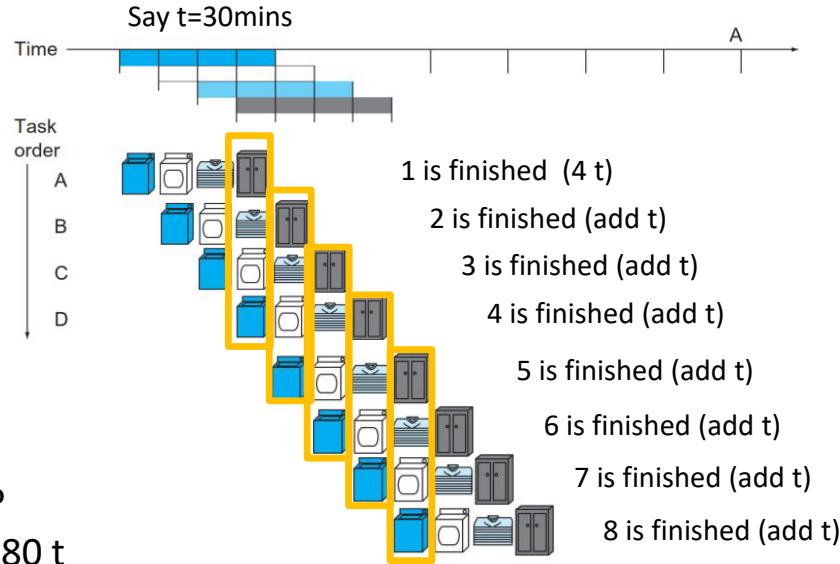
After 8 loads:

Not pipelined = 32t

Pipelined = 11t → that is 4t + 7t

Improvement of **2.9**

Pipelining



After 20 loads?

Not pipelined: 80 t

Pipelined: 4t + 19t = 23t

Improvement of **3.4 (approaches 4)**

Pipelining

- Any repetitive execution of a task can be pipelined if the task can be broken into steps
 - The more tasks you do in this pipelined format, the better
- Let us consider an instruction, which can be broken into 5 steps
 - Fetch and PC increment
 - Register read from register file (control unit computes setting of control lines)
 - ALU operates on data from register file (uses opcode to generate add/sub/etc...)
 - Read to / Write from memory (if needed)
 - Result from ALU is written to destination in register file

Pipelining

- Example: five steps of a load instruction (Id – load doubleword)

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load doubleword (Id)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store doubleword (sd)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, and, or)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps

Time to complete individual instruction is the “latency”

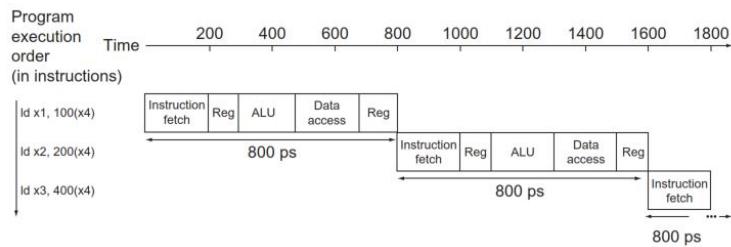
NOTE: the total time to complete an instruction cannot be changed...

- we are NOT decreasing latency
- we are increasing throughput

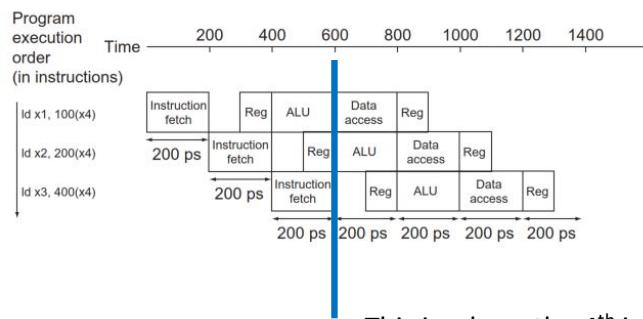
- We also must attend to the different times each step will take on the execution
Longest stage (200ps) will determine the organization of the pipeline

Pipelining

- Let's look at 3 load instructions called sequentially by the program



A 4th instruction will come at 2400ps



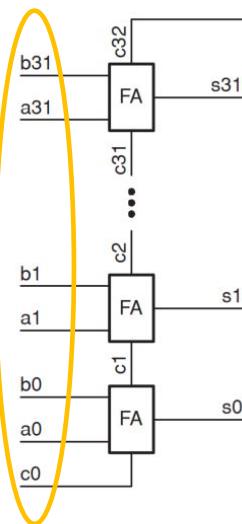
A 4th instruction will come at 600ps

4 fold improvement

This is where the 4th instruction starts

Pipelining

- Application example: a 32 bit ripple-carry adder



Each Full Adder (FA) stage 100ps from carry-in to carry-out

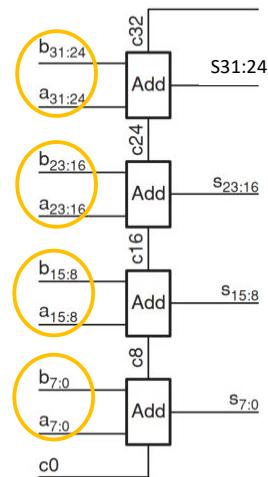
Worst case, carry propagates all the way from LSB to MSB (32*100ps)

An add is performed in 3.2ns

How can we improve throughput?

Pipelining

- Improving throughput: adder is divided into submodules



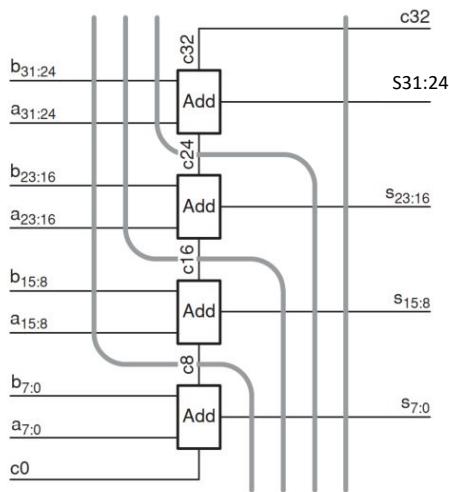
Each module will take care of 8 bits
(a “slice” of 8 bits of the 32 bit input)
Along with a carry

Will generate also 8 bits and a carry

The delay of each is 800ps (8*100ps)

Pipelining

- Improving throughput: submodules are organized into stages

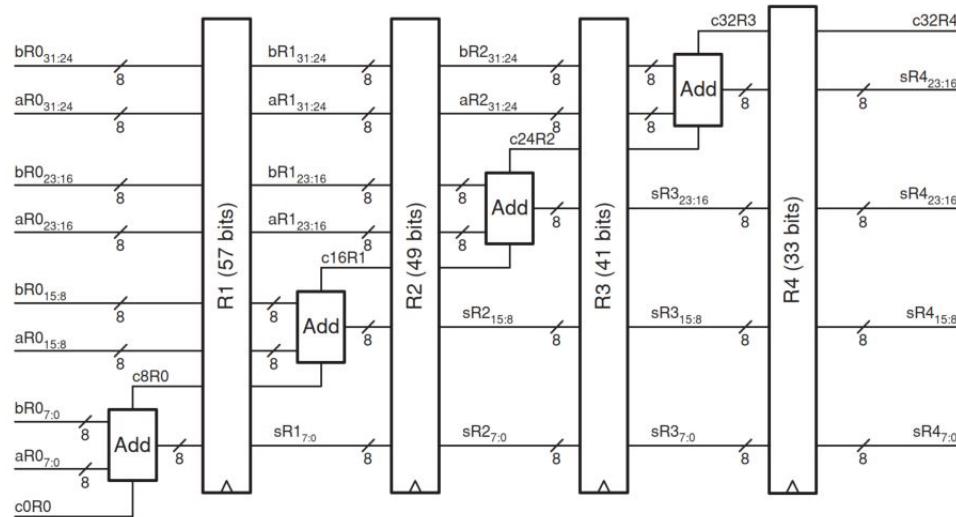


If I store the (partial) result of every stage, I can let each stage operate separately on different parts of different inputs

“store” → use registers!

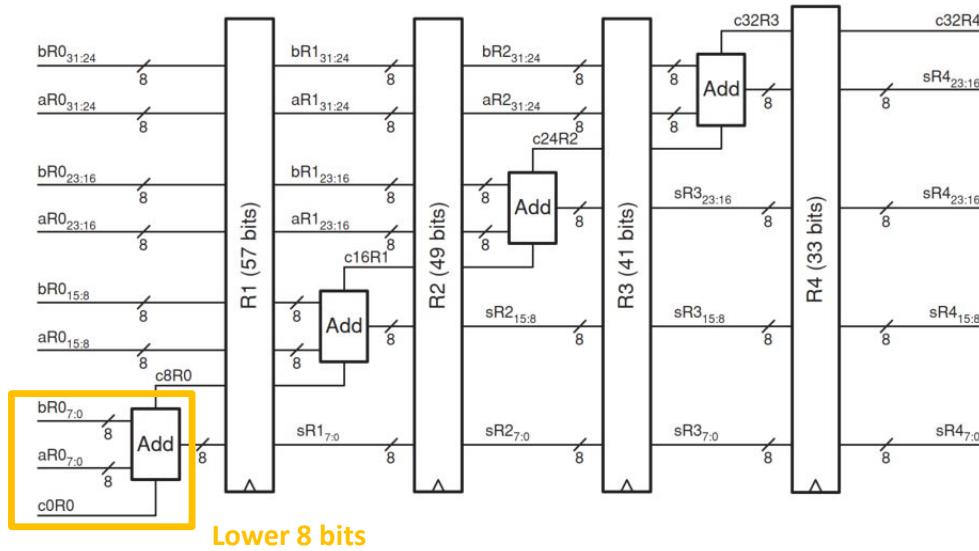
Pipelining

- Improving throughput: use registers for partial results, organize stages



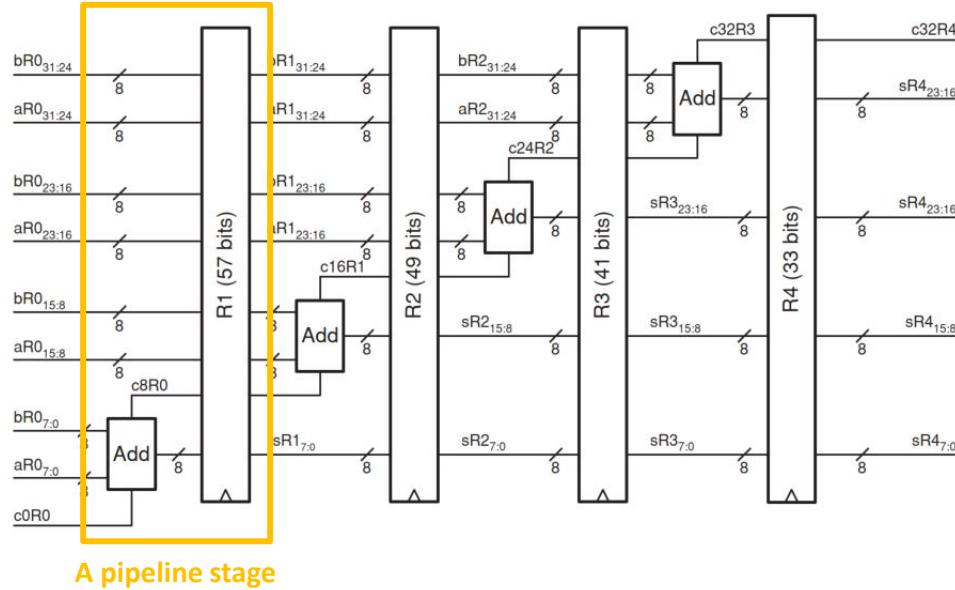
Pipelining

- Improving throughput: use registers for partial results, organize stages



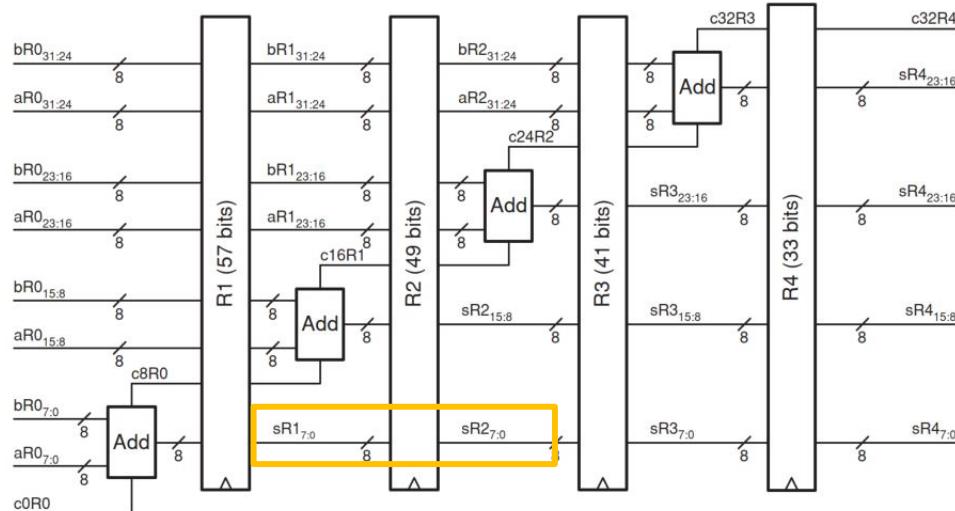
Pipelining

- Improving throughput: use registers for partial results, organize stages



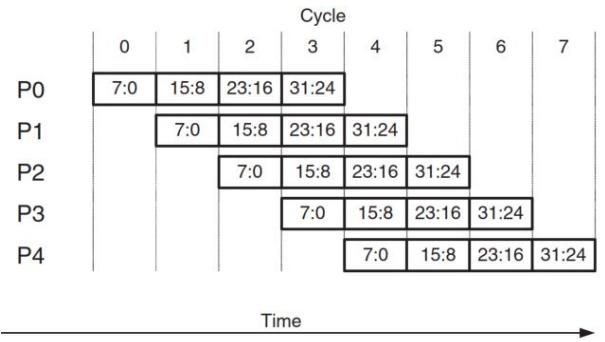
Pipelining

- Improving throughput: use registers for partial results, organize stages



Pipelining

- Improving throughput: a pipelined adder



- The register will add overhead (say, 200ps) and cost

$$t_{\text{reg}} = t_{\text{setup}} + t_{\text{dcq}} + t_{\text{skew}}$$

- Each pipeline stage is now

$$t_{\text{pipe}} = t_{\text{max}} + t_{\text{setup}} + t_{\text{dcq}} + t_{\text{skew}}$$
 Where t_{max} is longest comb. logic stage

Pipelining

- Takeaway

- If a larger task needs to be done repetitively, break into stages
 - Store temporary results between stages
- Organize execution taking advantage of stages that can be pipelined
- Be mindful of the number of stages used as they will determine how wide the pipeline needs to be
- Be mindful of potential overheads per stage or stages with different latency

ECE342 – Computer Hardware

Lecture 25

Bruno Korst, P.Eng.

Agenda

- Pipelining cont'd

Pipelining

- Example: five steps of a load instruction (Id – load doubleword)

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load doubleword (Id)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store doubleword (sd)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, and, or)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps

Time to complete individual instruction is the “latency”

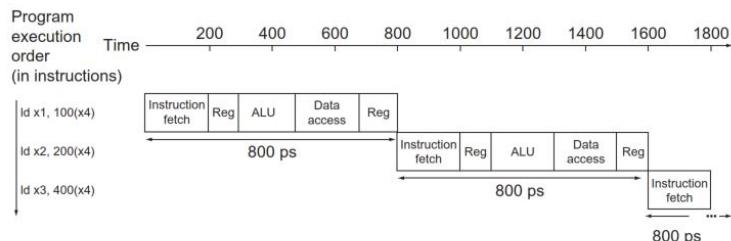
NOTE: the total time to complete an instruction cannot be decreased...

- we are NOT decreasing latency
- **we are increasing throughput**

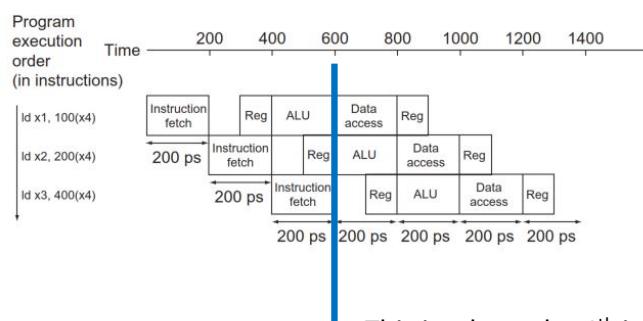
- We also must attend to the different times each step will take on the execution
Longest stage (200ps) will determine the organization of the pipeline

Pipelining

- Let's look at 3 load instructions called sequentially by the program



A 4th instruction will come at 2400ps



A 4th instruction will come at 600ps

4 fold improvement

This is where the 4th instruction starts

But execution of instruction (latency) now is increased

Pipelining

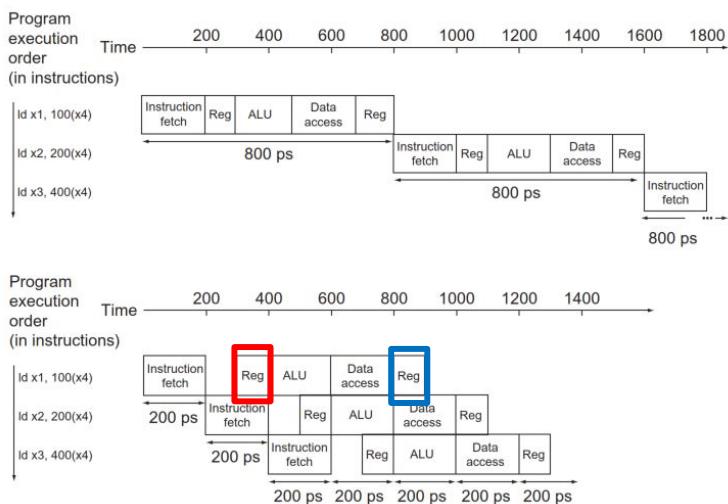
- So far, we see...
 - If a larger task needs to be done repetitively, break into stages
 - Store temporary results between stages
 - Organize execution taking advantage of stages that can be pipelined
 - Be mindful of the number of stages used as they will determine how wide the pipeline needs to be
 - Be mindful of potential overheads per stage or stages with different latency

Pipelining

- If stages are **perfectly balanced**
 - Improvement is approximately equal to the number of stages (as we saw)
- In **reality**, they are **not perfectly balanced**
 - We also saw that there is other overhead
 - Slowest resource: ALU op, memory access
- What is the improvement?

Pipelining

THREE LOAD INSTRUCTIONS



From 800ps
time inbetween
instructions to
200ps

Four fold...

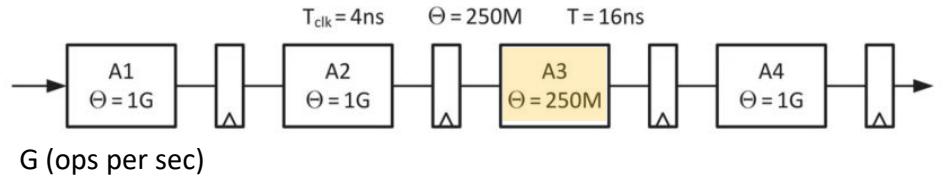
NOTE:
Write register
 happens on 1st
 half of cycle
Read register
 happens on 2nd
 half of cycle

Pipelining

- For **three load instructions**
 - We have now
 - 2400ps non pipelined (3 instr @ 800ps)
 - 1400ps pipelined
- Say we **add 1 million** instructions
 - (pipeline) $1,000,000 * 200\text{ps} + 1400$
 - (non-pipeline) $1,000,000 * 800\text{ps} + 2400$
- Improvement ~ ratio of time between instructions

Pipelining

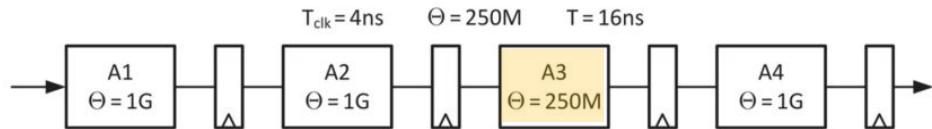
- Balancing loads
 - The slowest stage in the instruction determines throughput
 - We need **strategies to balance the load** in the pipeline
 - Consider this pipeline



- The faster stages upstream must idle to prevent overflow buffers...
AND
- The faster stages downstream must idle as they do not yet have data...

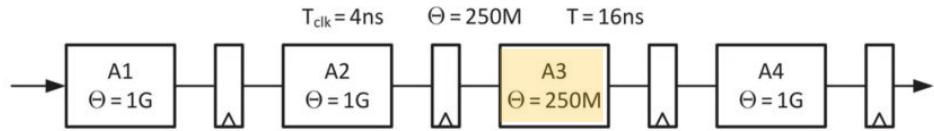
Pipelining

- The 250 M ops per sec limits the clock to a 4ns period
- There are 4 stages at 4ns, resulting in a 16ns overall latency

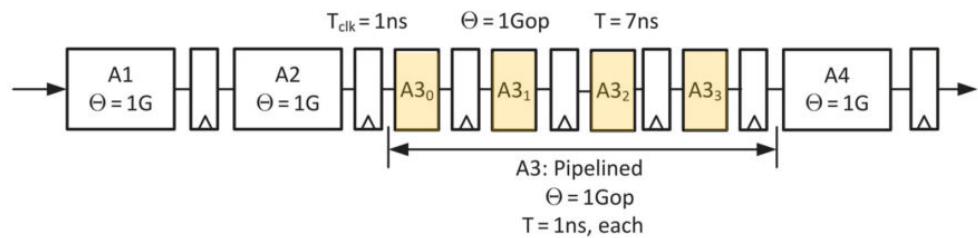


- To improve throughput, one must try to match the throughput of the other stages
 - Go from 250M to 1G ops per sec

Pipelining

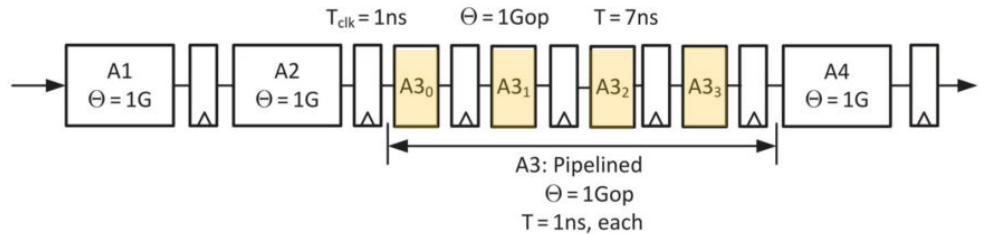


- Breaking stage A3 into four 1ns stages (replicate and pipeline) we have



- Improved throughput, with low latency per each A3 stage

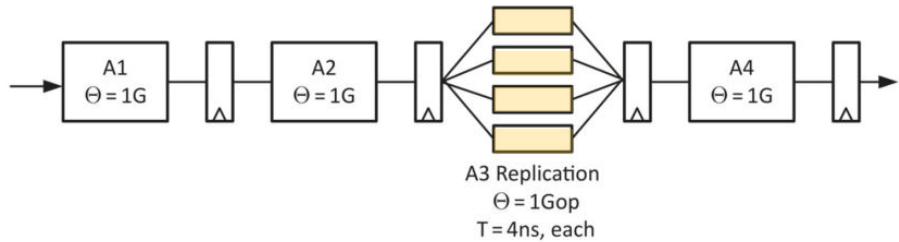
Pipelining



- Now each stage is 1G ops per sec
 - clock speed is set at 1ns (improved)
 - overall latency is 7 stages at 1ns each $\rightarrow 7\text{ns}$
 - throughput achieved and lower overall latency

Pipelining

- Another option is to replicate stage 3 in parallel...



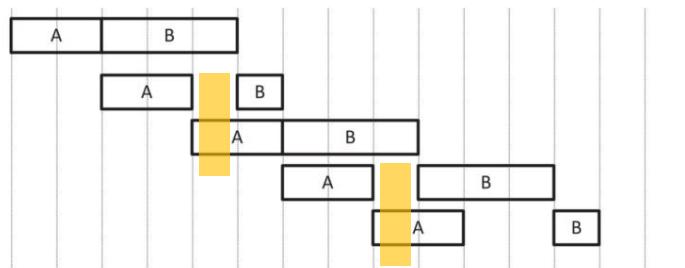
- Latency at stage 3 $\rightarrow 4\text{ns}$ (clock also 4ns)
- Throughput is 1Gops for all...
- However, all other stages need to produce 4 outputs per clock cycle to maintain throughput

Pipelining

- Variable Loads
 - In the context of instructions, stage delays are different, but fixed
 - Delay at pipeline stage may not always be constant
 - In a rigid pipeline, this will create problems
 - **FIFO buffers** may be inserted between stages
 - It averages out the variation

Pipelining

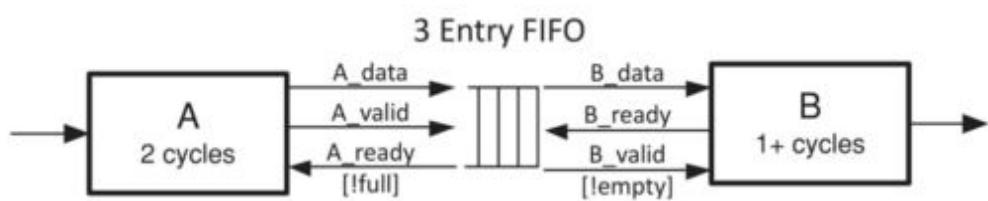
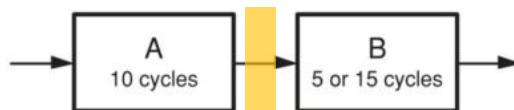
- Instead of a register, now there is a FIFO buffer (it's a queue)



When A is finished on one stage, can start at another without delay

Pipelining

- A FIFO queue instead of a register



Pipelining

- Another strategy that can be used is to **repurpose resources** that are lightly used
 - “Resource sharing”
- May be shared **between stages** of one pipeline or **between independent pipelines**
 - Arbiter resolves conflicts
 - May need to use a combination of arbiter and FIFO to deal with variable throughput as result of arbitration

Pipelining - Hazards

- Hazards
 - Events when the next instruction cannot start to execute in the next cycle
- Three types
 - Structural Hazard
 - Data Hazard
 - Control Hazard

Pipelining

- Structural Hazard
 - Hardware cannot support the combination of instructions that are to be executed in the same clock cycle
 - Laundry analogy: if one machine is washer AND dryer, we cannot organize the pipe as we wish
- Data Hazard
 - Pipeline must be stalled as one step is waiting for the other to complete.
 - Data needed has not been made available

Pipelining

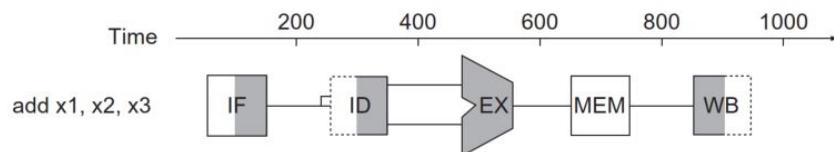
- Control Hazard
 - Also called Branch hazard
 - A decision needs to be made based on results of one instruction while other instructions are executing
 - What do we do? We may need to stall the pipeline or we may attempt to predict what the next instruction will need

Pipelining

- Structural
 - It's a matter of resource conflict between instructions
 - May occur due to design of execution units
 - Ex
 - Execution unit takes more than one cycle and it is not pipelined.
 - Instructions using this execution unit cannot be issued in sequence
 - Number of writes to register file exceeds number of write ports

Pipelining

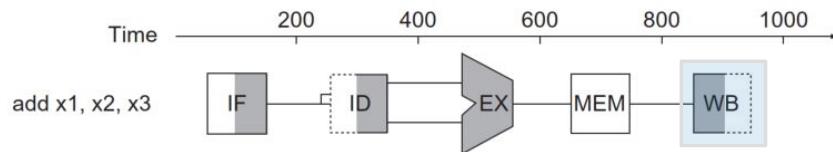
- Data Hazards
 - Pipeline must be stopped as one step waits for another to complete
 - “Stall”
 - Consider the instructions:
 - add x19, x0, x1
 - sub x2, x19, x3
 - Recall the execution stages



Pipelining

add **x19**, x0, x1

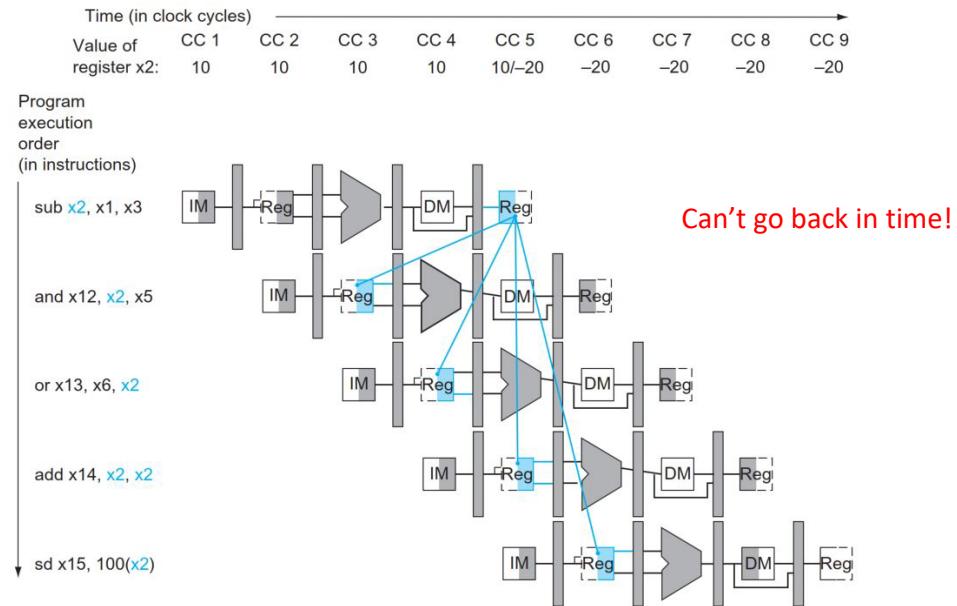
sub x2, **x19**, x3



- Writing the results will not happen until stage 5!
- Instruction decode (ID) happens in stage 2...
 - **add** will provide the result much later than **sub** could use...

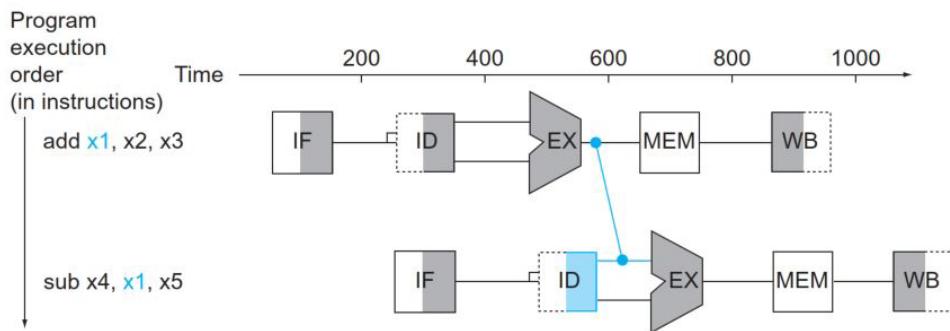
Pipelining

- Data Hazards



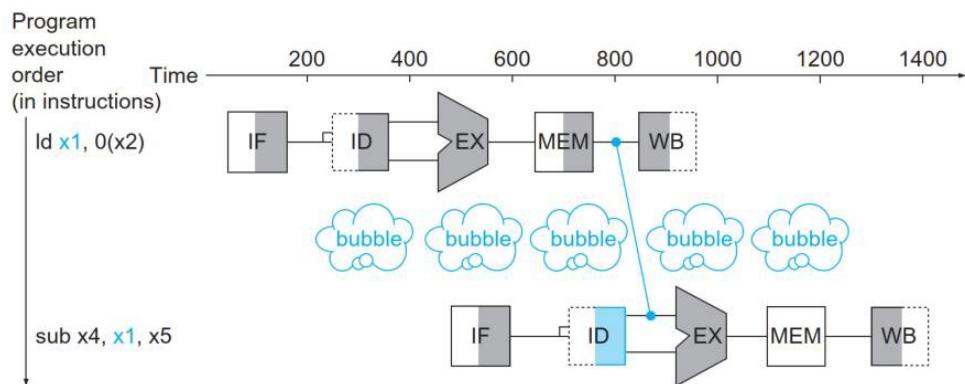
Pipelining

- Data Hazards
 - One solution is to stall the pipe and wait...
 - We don't really need to wait if we use “forwarding” (bypassing)
 - As soon as the execution is done, the data is actually ready



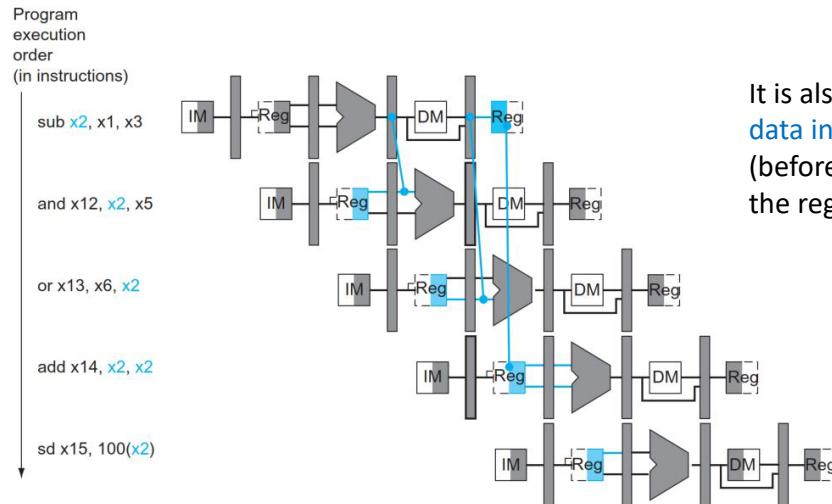
Pipelining

- Data Hazards (cont'd)
 - We may actually need a combination of stalling and forwarding
 - This example: a load instruction followed by a subtraction



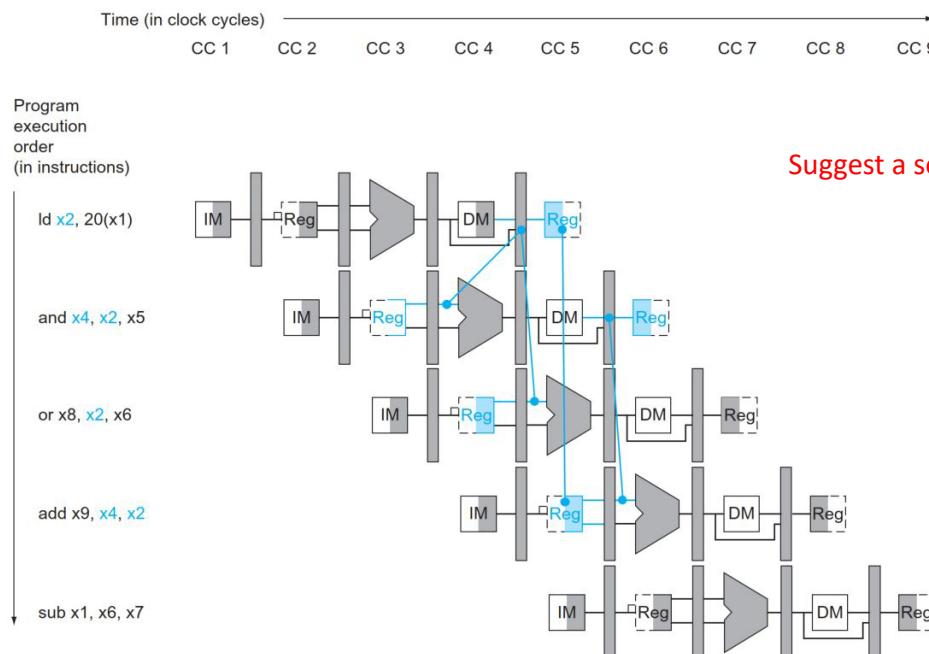
Pipelining

	Time (in clock cycles)	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
value of register x2:		10	10	10	10	10–20	-20	-20	-20	-20

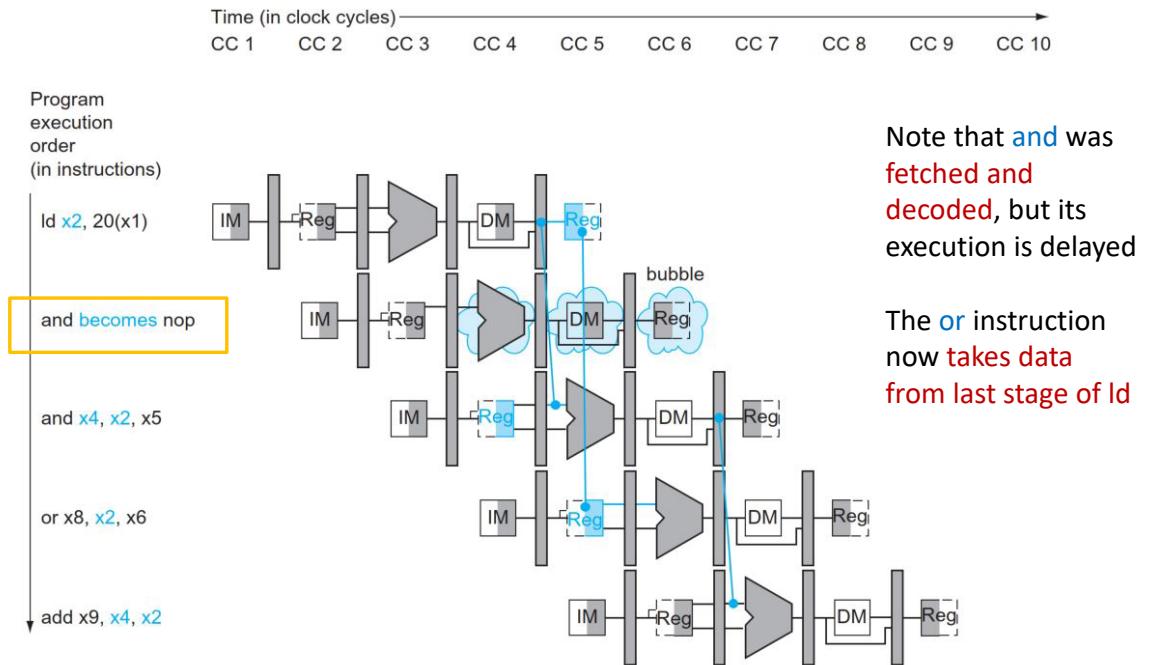


It is also possible to use the data in the pipeline registers (before they are written to the register file)

Pipelining



Pipelining



Pipelining

- Three primary **types of data hazards**
 - Read after write (RAW)
 - Write after read (WAR)
 - Write after write (WAW)
- Assume two instructions i1, i2, and i2 should be executed after i1
 - **Read after Write (RAW)** – i2 reads data source before i1 writes to it
 - (i1) add R2, R3, R4
 - (i2) add R5, R2, R1

invalid result if i2 reads R2 before i1 writes to it

Pipelining

- Assume two instructions i1, i2, and i2 should be executed after i1

- Write after Read (WAR)

- i2 writes to a location before i1 reads it

(i1) add R2, R3, **R4**

(i2) add **R4**, R5, R6

invalid result if i2 writes to R4 before i1 reads it

-- does not happen if the order of execution is preserved
(problem if execution and completion is out of order)

Pipelining

- Assume two instructions i1, i2, and i2 should be executed after i1

- Write after Write (WAW)

- i2 writes to a location before i1 writes to it

(i1) add **R2**, R3, R4

(i2) add **R2**, R5, R6

- invalid result if i2 writes to R2 before i1 writes it.

- does not happen if the order of execution is preserved
(problem if execution and completion is out of order)

Pipelining

- Dependencies are checked
 - statically by the compiler
 - Dynamically, at run time, by the hardware
- Dynamic check: actual memory addresses of load and store need to be known to resolve a dependency
 - these are not available at compile time, only at run time

Pipelining

- Control Hazards
 - These are **delay in determining proper instruction to fetch**
 - Relate to instructions that control flow
 - **Branches**
 - About 30% of instructions are branches
 - In the pipeline, this can **reduce throughput significantly** if not handled properly
 - Each branch
 - New address is loaded into PC

Pipelining

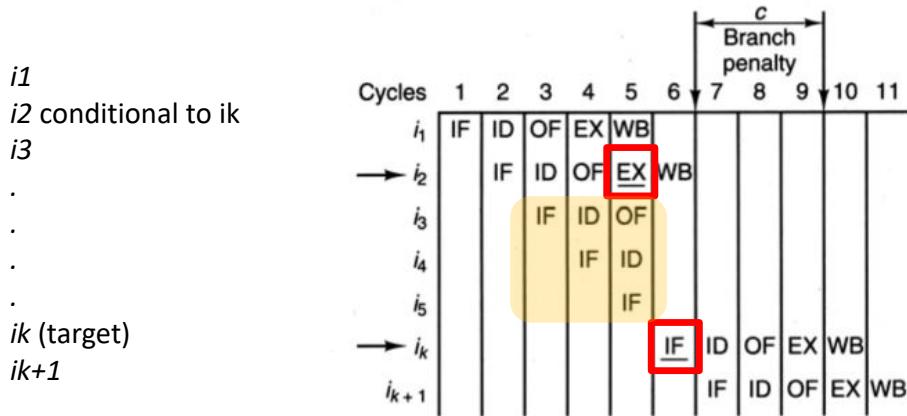
- Control Hazards
 - At each branching
 - New address loaded on PC **may invalidate existing instructions** already in the pipe or prefetched in the buffer
 - **Requires draining the pipeline and refilling it**
 - Branches **taken** degrade throughput
 - If branch is not taken, there is no need to drain/refill the pipeline

Pipelining

- Three groups of branching
 - unconditional – new target address into PC
 - conditional
 - if met – new target path
 - if not met – sequential path
 - Loop – jumps to beginning of loop and executes n times
- Conditional branches offer most problems

Pipelining

- Problem with conditional branch



Pipelining

- To reduce the effect of branching
 - Branch prediction
 - Delayed Branching
 - Multiple prefetching
- Branch prediction
 - Static – just prefetch one of the paths (maybe penalty)
 - Dynamic – keep history (table) and choose most often used

Pipelining

- Delayed branching
 - A number of instructions after the branch is prefetched and executed
 - Regardless of which path will be taken
 - Compiler reorganizes and tries to fill slots with instructions independent from the branch
 - NOP placed in empty slots
- Multiple prefetching
 - Processor fetches both possible paths
 - If branch occurs, one of the contents is invalidated
- Typically what is used is a combination of these techniques



ECE342 – Computer Hardware

Lecture 26

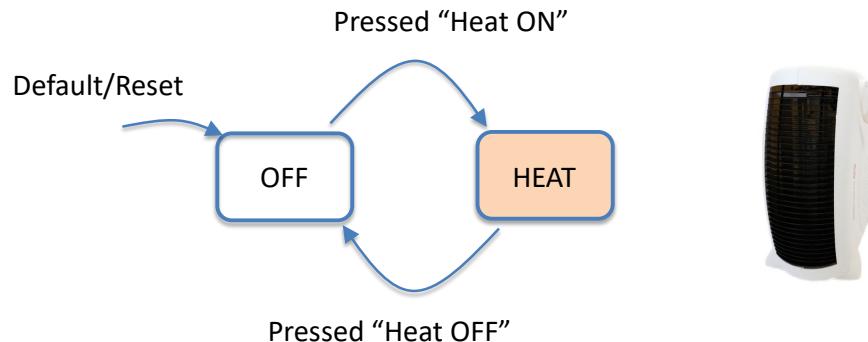
Bruno Korst, P.Eng.

Agenda

- FSM intro

First Scenario

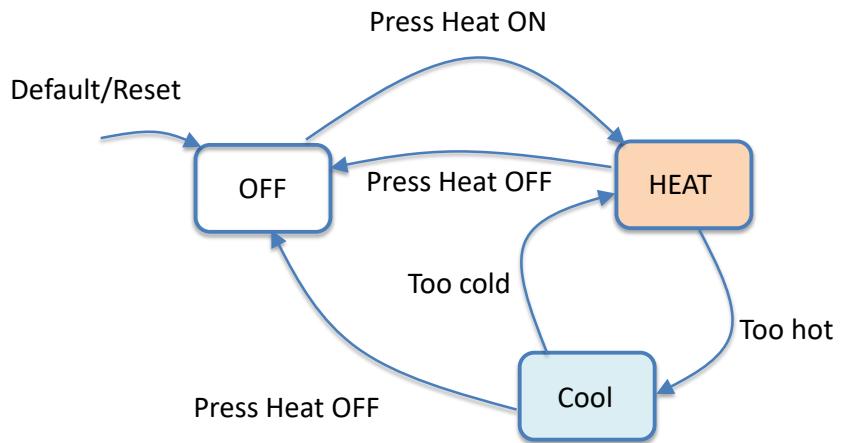
- A simple heater



Maybe this model is too simple...

First Scenario

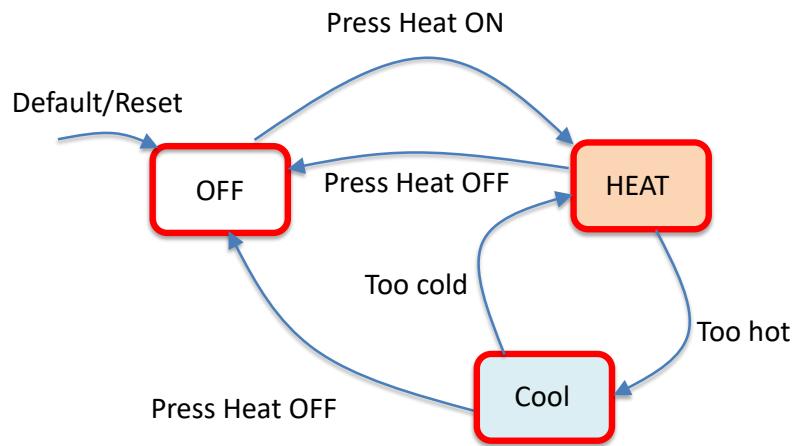
- A better heater



Next winter upgrade? Fan (speeds), timers, remote control etc.

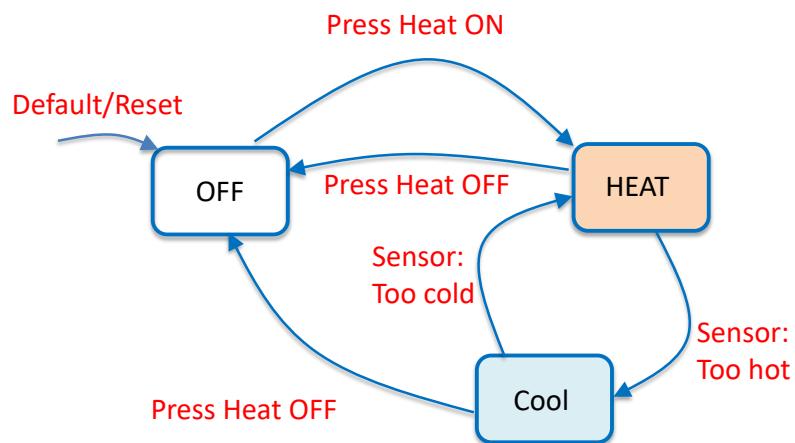
State Transition Diagram

- A number of **states / conditions**



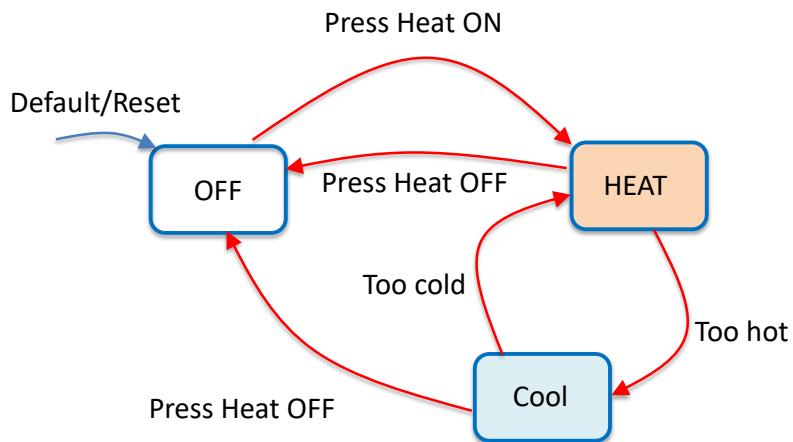
State Transition Diagram

- A number of **inputs**



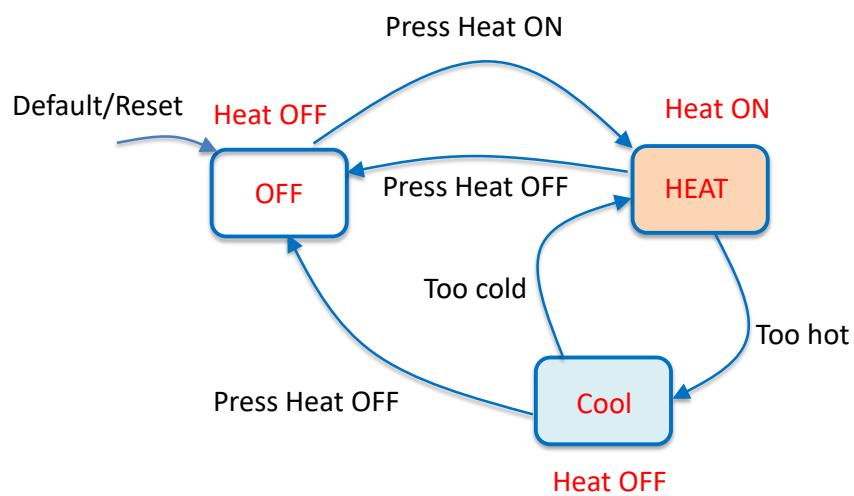
State Transition Diagram

- A number of **transitions** caused by the inputs



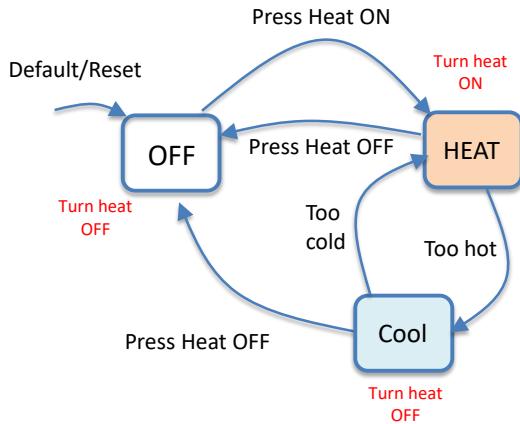
State Transition Diagram

- A number of **outputs**



From Diagram to Tables

State Transition Diagram

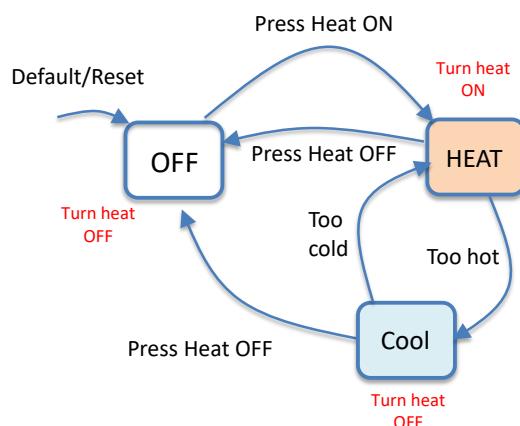


State Transition Table

Current	Button	Sensor	Next
OFF	ON		HEAT
HEAT	OFF		OFF
HEAT			
Cool			
Cool	OFF		OFF

From Diagram to Tables

State Transition Diagram

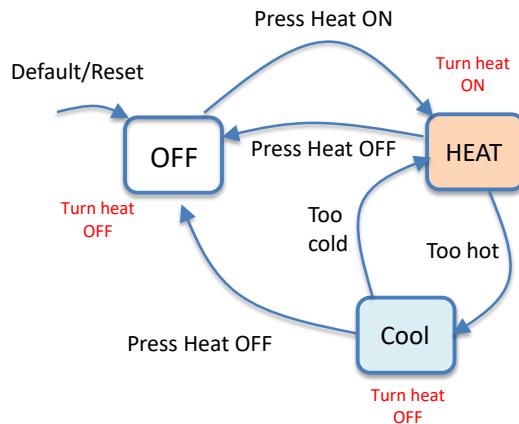


State Transition Table

Current	Button	Sensor	Next
OFF	ON		HEAT
HEAT	OFF		OFF
HEAT		Too hot	Cool
Cool		Too cold	Heat
Cool	OFF		OFF

From Diagram to Tables

State Transition Diagram

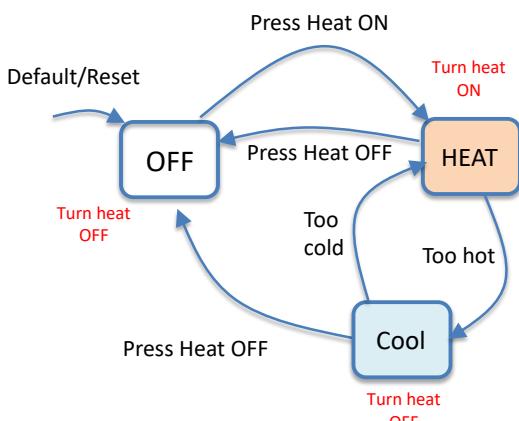


State Transition Table

Current	Button	Sensor	Next
OFF	ON	X	HEAT
HEAT	OFF	X	OFF
HEAT	X	Too hot	Cool
Cool	X	Too cold	HEAT
Cool	OFF	X	OFF

From Diagram to Tables

State Transition Diagram



State Transition Table

Current	Button	Sensor	Next
OFF	ON	X	HEAT
HEAT	OFF	X	OFF
HEAT	X	Too hot	Cool
Cool	X	Too cold	HEAT
Cool	OFF	X	OFF

Output Table

Current	Outputs
OFF	Turn heat OFF
HEAT	Turn heat ON
Cool	Turn heat OFF

Pause

- Think about different devices that can fit the description that we just gave. Have you considered...
 - The Ikea elevator (two floors only!)
 - Your washing machine
 - Vending machine
 - Automatic door
 - Furnace
- Let's recap the terminology, define all this formally

Terminology

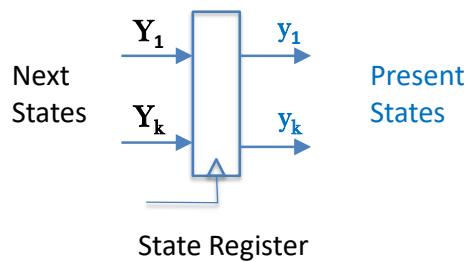
- Now we have a **machine**
 - With a **finite** number of **states**
 - With **inputs** (actions) that cause **transitions**
 - With **outputs** (actions) to be **performed** at each state
- We have a diagram to represent the machine, called
 - **State Transition Diagram**
 - documents how the machine operates
- Based on the State Transition Diagram we create a **State Transition Table** and an **Output Table**

Definitions

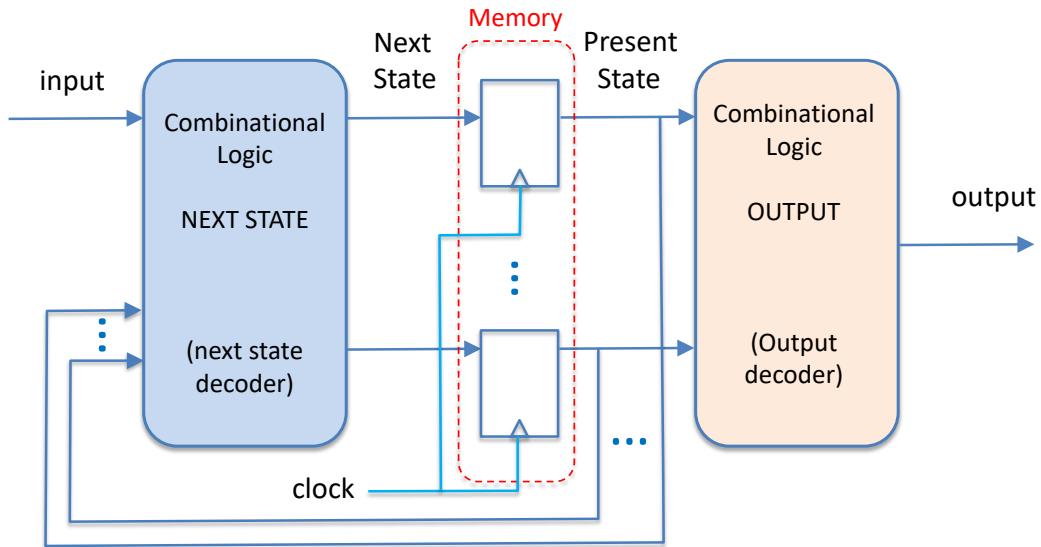
- We have a Synchronous Sequential Machine
 - Machine whose present outputs are a **function of the present state**
OR
 - Machine whose present outputs are a **function of the present inputs and the present state.**
- Synchronous → “responds to a clock”
- Sequential logic system → combinational logic + storage

Definitions

- **State** is “*a set of values that is measured at different locations within the machine*” (stored, in clocked D-FF, registers)
- **Present state:** state of the system at the present
- **Next state:** state **to which the system will enter** on the next clock edge.

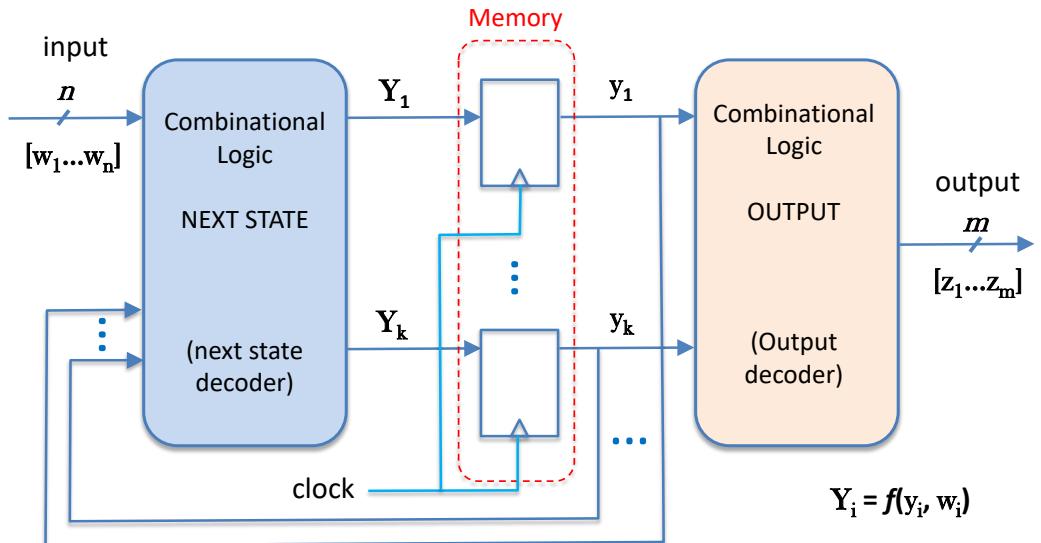


Definitions



- Note: the **clock** is not an input, it is a “heartbeat”
- Rising edge determines when state transitions occur → “next” becomes “present”

Definitions



$$Y_i = f(y_i, w_i)$$

“MOORE MACHINE”

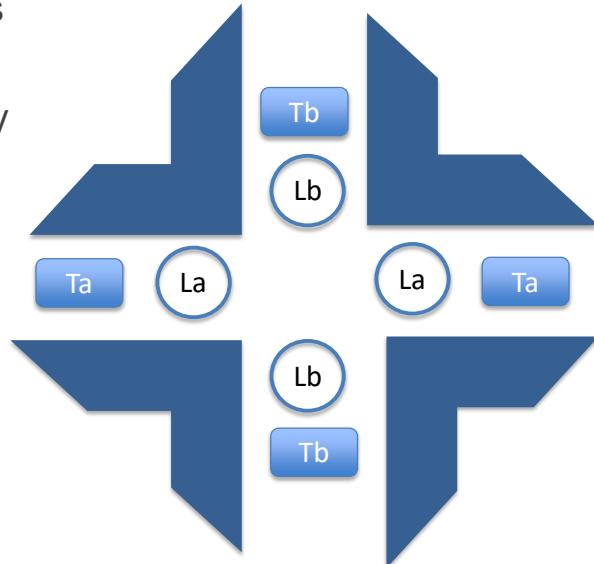
- k state variables (number of FF), make possible to represent 2^k states
- On clock edge, Y_i becomes y_i

Traffic Light FSM

- Let us look at another problem
 - There is an intersection of two busy roads which needs traffic lights to prevent collisions.
 - These lights are to be controlled by traffic sensors, which indicate whether there are cars present or not.
 - We are to design the controller

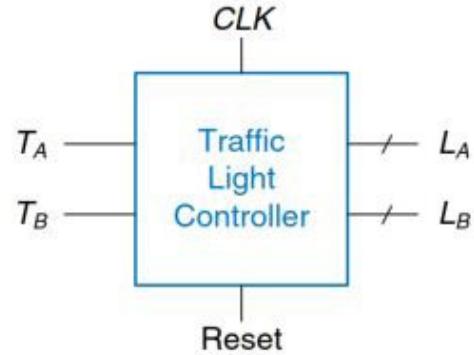
Traffic Light FSM

- Ta, Tb – traffic sensors
 - TRUE if cars present
 - FALSE if street empty
- La, Lb – lights
 - Green
 - Yellow
 - Red



Traffic Light FSM

- Two inputs – T_A , T_B
- Two outputs – L_A , L_B
- Clock
 - Say, at 5 sec period
 - In reality, a timer...



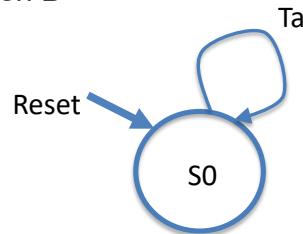
- Reset brings it to a known initial state (asynchronously)

Traffic Light FSM

- **A systematic approach to the design**
 1. Draw the state transition diagram
 2. Create the state transition table
 3. Encode the states on the state transition table
 4. Derive the logic expressions, minimize them
 5. Create the output table, encode it
 6. Derive the logic expressions, minimize them
 7. Implement it
 - On “old-school” hardware, in code, on HDL

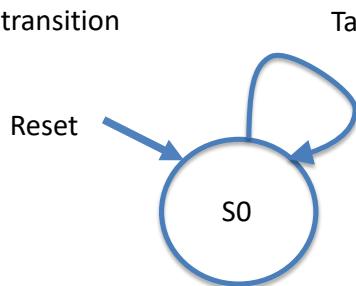
Traffic Light FSM

- State Transition Diagram
 - States are represented by circles, transitions by arcs
 - Reset – green on “A” street, red on “B” street
 - Every 5 seconds (clock), examine sensors (Ta, Tb) and decide:
 - Is traffic present on A – if so, lights do not change
 - When no more traffic on A for 5 seconds – light turns yellow on A
 - 5 seconds later, red on A, green on B



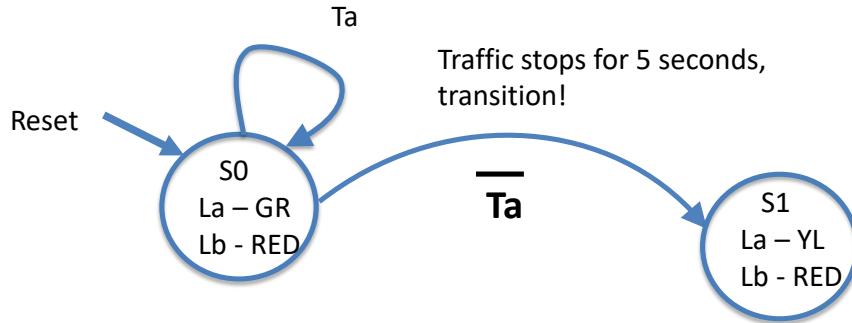
Traffic Light FSM

Reset does not depend
on the initial transition

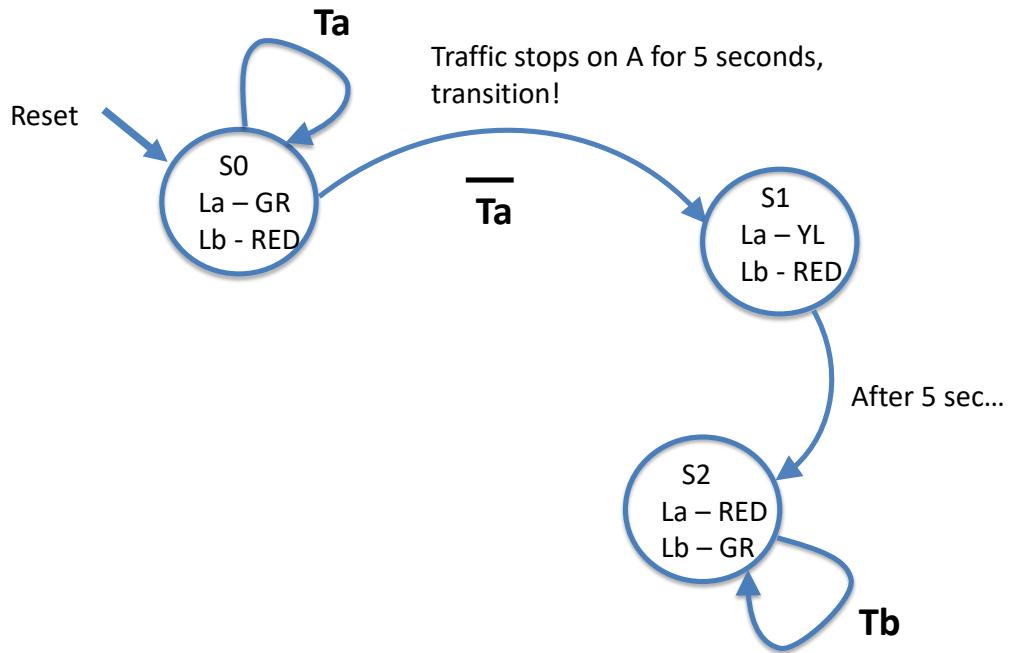


While there IS traffic,
there is no change.
(Ta is TRUE)

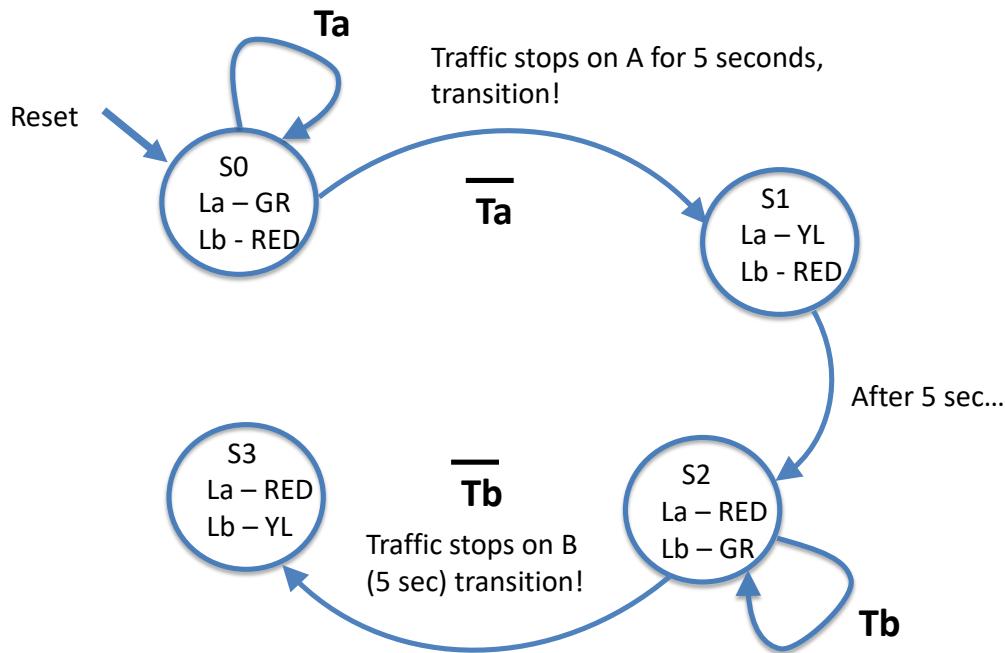
Traffic Light FSM



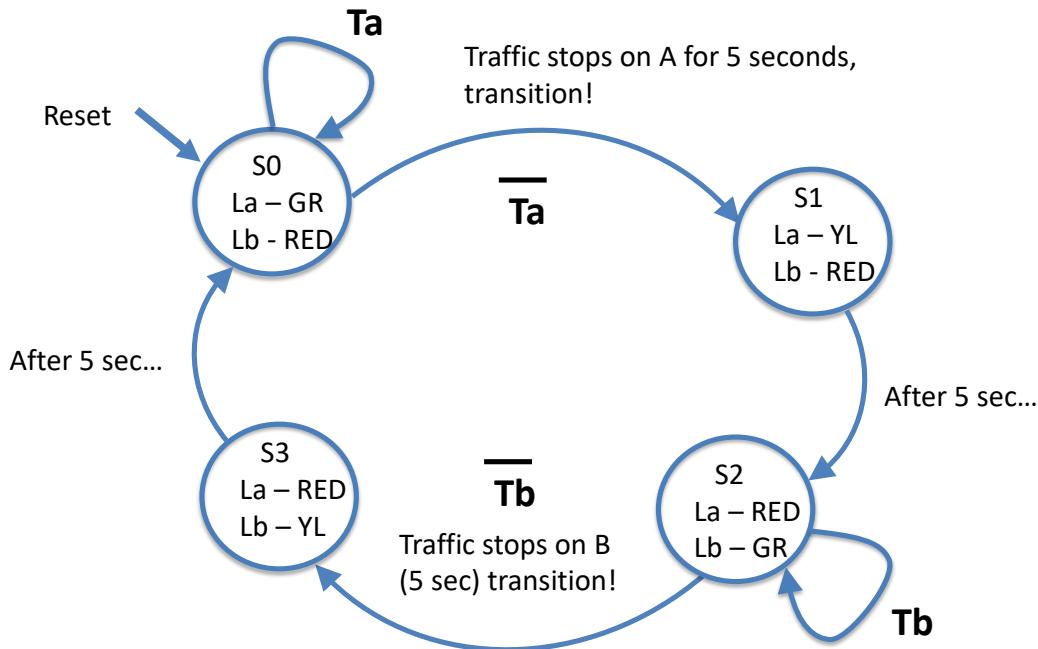
Traffic Light FSM



Traffic Light FSM



Traffic Light FSM

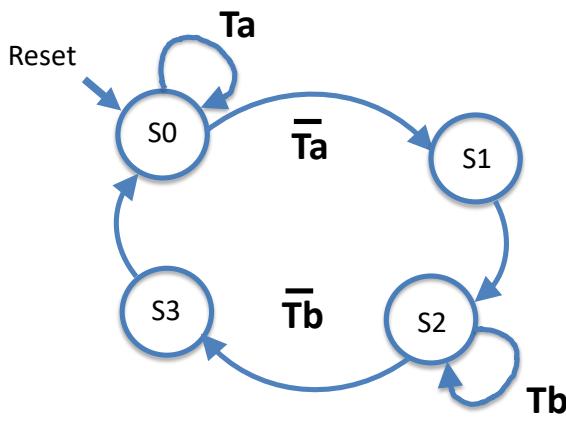


Traffic Light FSM

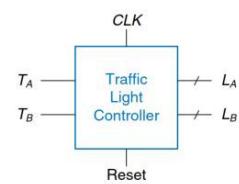
- A systematic approach to the design
 1. Draw the state transition diagram
 2. Create the state transition table
 3. Encode the states on the state transition table
 4. Derive the logic expressions, minimize them
 5. Create the output table, encode it
 6. Derive the logic expressions, minimize them
 7. Implement it
 - On “old-school” hardware, in code, on HDL

Traffic Light FSM

- State Transition Table



Current	Ta	Tb	Next
S0	0	X	S1
S0	1	X	S0
S1	X	X	S2
S2	X	0	S3
S2	X	1	S2
S3	X	X	S0



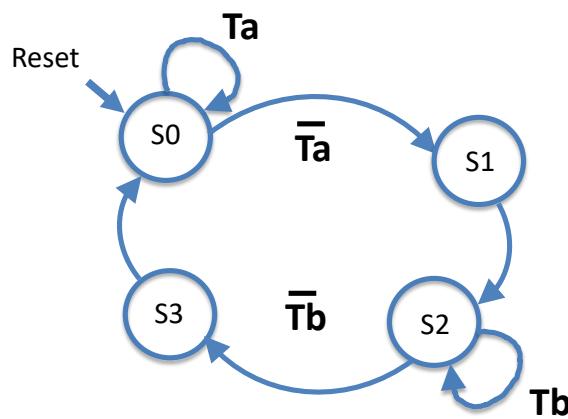
Like we saw on the heater example before

Traffic Light FSM

- A systematic approach to the design
 1. Draw the state transition diagram
 2. Create the state transition table
 3. Encode the states on the state transition table
 4. Derive the logic expressions, minimize them
 5. Create the output table, encode it
 6. Derive the logic expressions, minimize them
 7. Implement it
 - On “old-school” hardware, in code, on HDL

Traffic Light FSM

- State Encoding



Current	Ta	Tb	Next
0 0	0	X	0 1
0 0	1	X	0 0
0 1	X	X	1 0
1 0	X	0	1 1
1 0	X	1	1 0
1 1	X	X	0 0

S'b1 S'b0 S'b1 S'b0

Traffic Light FSM

- A systematic approach to the design
 1. Draw the state transition diagram
 2. Create the state transition table
 3. Encode the states on the state transition table
 4. Derive the logic expressions, minimize them
 5. Create the output table, encode it
 6. Derive the logic expressions, minimize them
 7. Implement it
 - On “old-school” hardware, in code, on HDL

Traffic Light FSM

- Logic Expressions

Current	Ta	Tb	Next
0 0	0	X	0 1
0 0	1	X	0 0
0 1	X	X	1 0
1 0	X	0	1 1
1 0	X	1	1 0
1 1	X	X	0 0

↑ ↑ ↑ ↑
 S'b1 S'b0 S'b1 S'b0

$$S'b_1 = \overline{Sb}_1 Sb_0 + Sb_1 \overline{Sb}_0 \overline{T_b} + Sb_1 \overline{Sb}_0 T_b$$

$$S'b_0 = \overline{Sb}_1 \overline{Sb}_0 \overline{T_a} + Sb_1 \overline{Sb}_0 \overline{T_b}$$

Traffic Light FSM

- Logic Expressions

Current	Ta	Tb	Next
0 0	0	X	0 1
0 0	1	X	0 0
0 1	X	X	1 0
1 0	X	0	1 1
1 0	X	1	1 0
1 1	X	X	0 0



S'b₁ **S'b₀** **S'b₁** **S'b₀**

$$S'b_1 = \overline{Sb}_1 Sb_0 + Sb_1 \overline{Sb}_0 \overline{T}_b + Sb_1 \overline{Sb}_0 T_b$$

$$S'b_0 = \overline{Sb}_1 \overline{Sb}_0 \overline{T}_a + Sb_1 \overline{Sb}_0 \overline{T}_b$$

Traffic Light FSM

- State Transition Table – next states from current states

Current	Ta	Tb	Next
0 0	0	X	0 1
0 0	1	X	0 0
0 1	X	X	1 0
1 0	X	0	1 1
1 0	X	1	1 0
1 1	X	X	0 0



Sb₁ **Sb₀** **S'b₁** **S'b₀**

$$S'b_1 = \overline{Sb}_1 Sb_0 + Sb_1 \overline{Sb}_0$$

$$S'b_0 = \overline{Sb}_1 \overline{Sb}_0 \overline{T}_a + Sb_1 \overline{Sb}_0 \overline{T}_b$$



$$S'b_1 = Sb_1 \oplus Sb_0$$

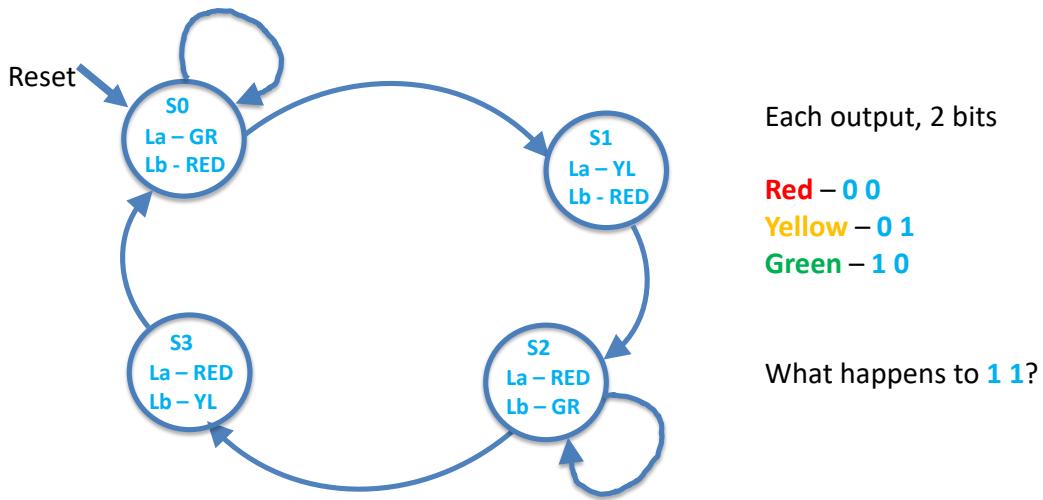
$$S'b_0 = \overline{Sb}_1 \overline{Sb}_0 \overline{T}_a + Sb_1 \overline{Sb}_0 \overline{T}_b$$

Traffic Light FSM

- A systematic approach to the design
 1. Draw the state transition diagram
 2. Create the state transition table
 3. Encode the states on the state transition table
 4. Derive the logic expressions, minimize them
 5. Create the output table, encode it
 6. Derive the logic expressions, minimize them
 7. Implement it
 - On “old-school” hardware, in code, on HDL

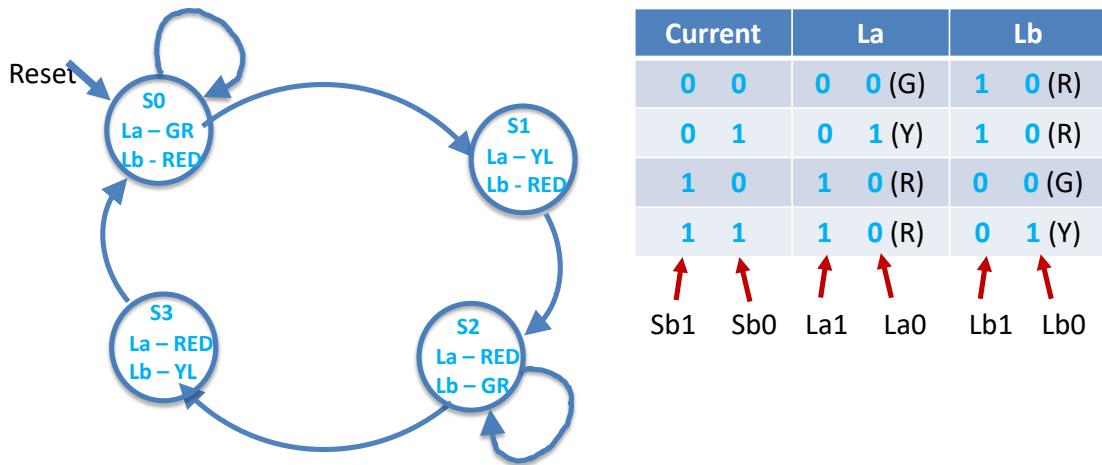
Traffic Light FSM

- Output table – we look at La and Lb, codified



Traffic Light FSM

- Output table – we look at La and Lb, codified



Traffic Light FSM

- A systematic approach to the design
 1. Draw the state transition diagram
 2. Create the state transition table
 3. Encode the states on the state transition table
 4. Derive the logic expressions, minimize them
 5. Create the output table, encode it
 6. Derive the logic expressions, minimize them
 7. Implement it
 - On “old-school” hardware, in code, on HDL

Traffic Light FSM

- **Output table** – we look at La and Lb, codified

Current	La	Lb
0 0	0 0 (G)	1 0 (R)
0 1	0 1 (Y)	1 0 (R)
1 0	1 0 (R)	0 0 (G)
1 1	1 0 (R)	0 1 (Y)



 Sb1 Sb0 La1 La0 Lb1 Lb0

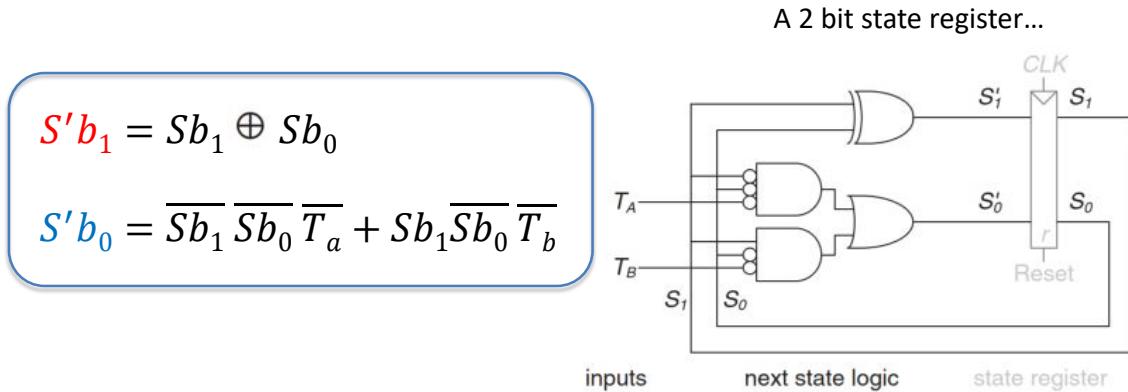
$$\begin{aligned}
 La_1 &= Sb_1 \\
 La_0 &= \overline{Sb}_1 Sb_0 \\
 Lb_1 &= \overline{Sb}_1 \\
 Lb_0 &= S_1 S_0
 \end{aligned}$$

Traffic Light FSM

- **A systematic approach to the design**
 1. Draw the state transition diagram
 2. Create the state transition table
 3. Encode the states on the state transition table
 4. Derive the logic expressions, minimize them
 5. Create the output table, encode it
 6. Derive the logic expressions, minimize them
 7. Draw the circuit, implement it
 - On “old-school” hardware, in code, on HDL

Traffic Light FSM

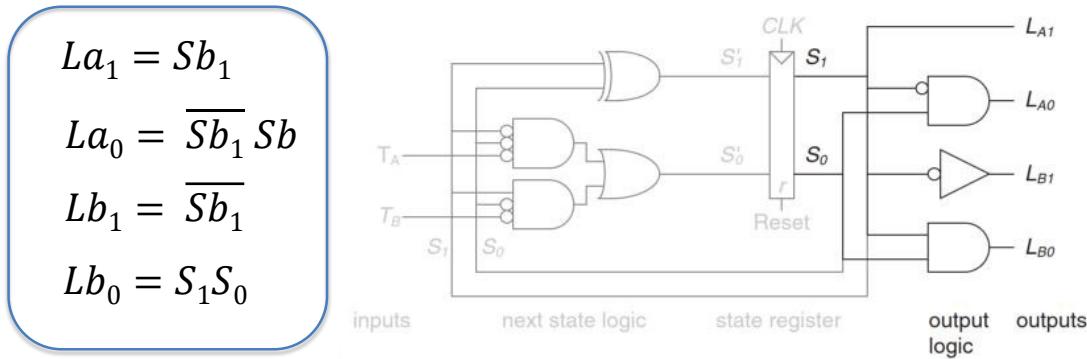
- We've got equations for next states from current states



- But that does not say anything about the output...

Traffic Light FSM

- We've ALSO got equations for outputs, which result in...



Note: Outputs depend only on the present state → MOORE MACHINE

Traffic Light FSM

Pause

- We just saw a systematic approach to solve a given problem using a Finite State Machine...

- Once you move on to implementing it, may other decisions come to play, beyond this approach
 - What happened on the video?
 - Did the designer invert a signal that should not have been inverted?
 - Did a sensor fail under the low temperature?

FSM in Code

- Previously seen:
Verilog
implementation

```
module simple (Clock, Resetn, w, z);
    input Clock, Resetn, w;
    output z;
    reg [2:1] y, Y;
    parameter [2:1] A = 2'b00, B = 2'b01, C = 2'b10;

    // Define the next state combinational circuit
    always @(w, y)
        case (y)
            A: if (w)    Y = B;
                else      Y = A;
            B: if (w)    Y = C;
                else      Y = A;
            C: if (w)    Y = C;
                else      Y = A;
            default:   Y = 2'bxx;
        endcase

    // Define the sequential block
    always @ (posedge Clock, negedge Resetn)
        if (Resetn == 0) y <= A;
        else             y <= Y;

    // Define output
    assign z = (y == C);

endmodule
```

FSM in Code

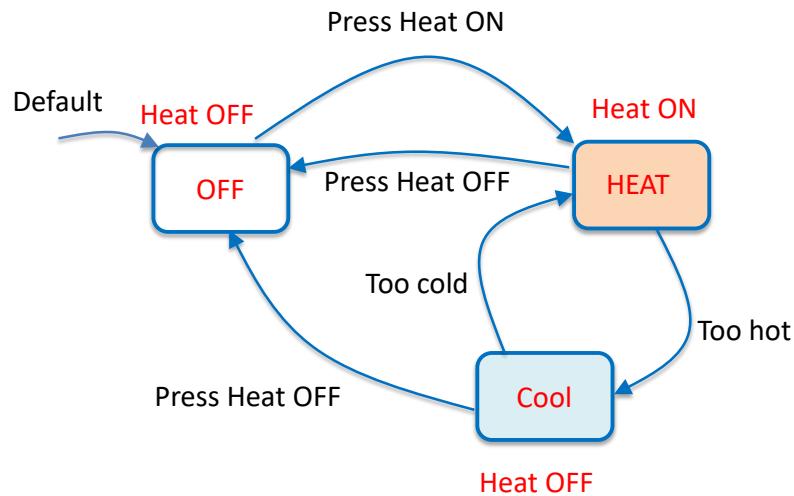
- In firmware, this means

```
State = 0;
SWITCH (State)
{
    CASE 0: IF (in_key() == 5) THEN state = 1;
              Break;
    CASE 1: IF (in_key() == 8) THEN State = 2;
              Else State = 0;
              Break;
    CASE 2: IF (in_key() == 3) THEN State = 3;
              Else State = 0;
              Break;
    CASE 3: IF (in_key() == 2) THEN UNLOCK();
              Else State = 0;
              Break;
}
```

- (open a lock)

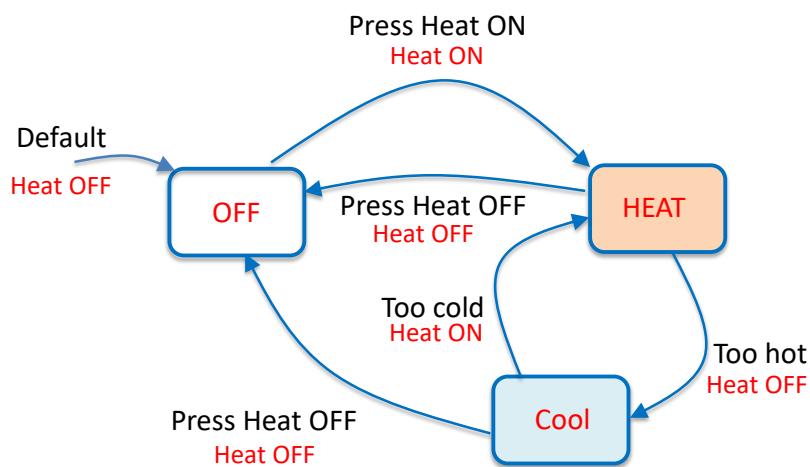
Moore Machine

- A machine whose present outputs are a function of the **present state ONLY**

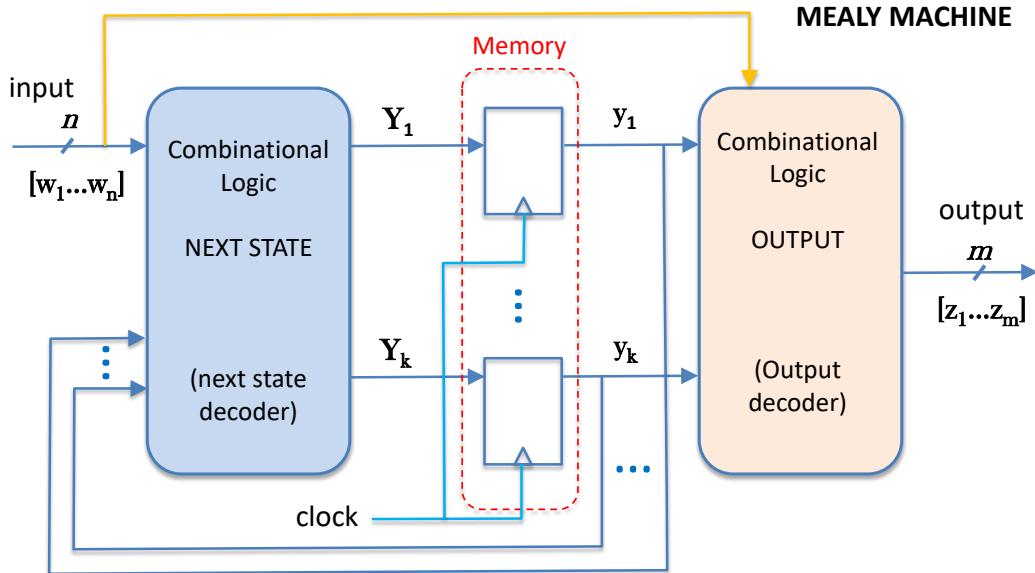


Mealy Machine

- A machine whose present outputs are a function of the **present state and the present input**.



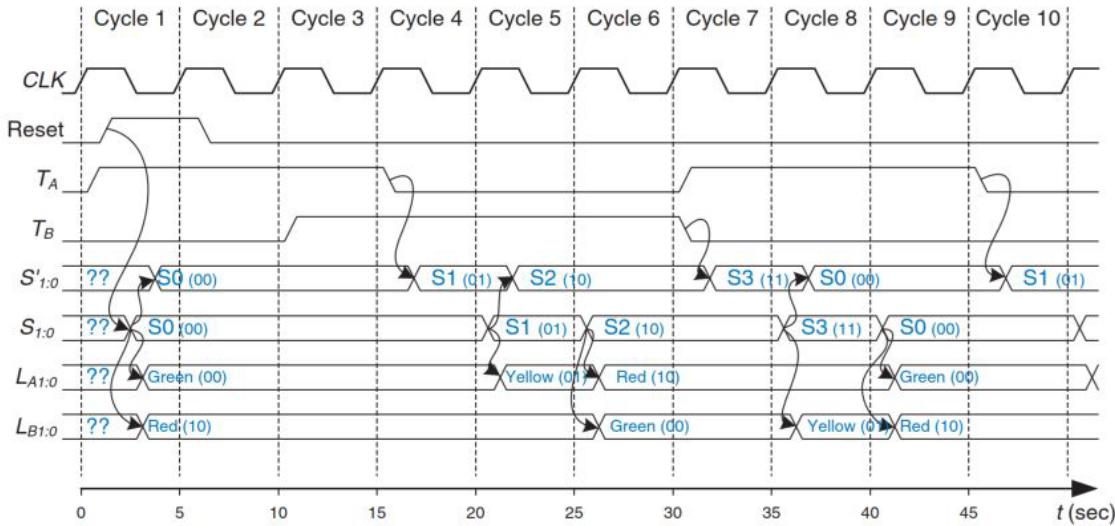
Definitions



- On clock edge, Y_i becomes y_i

Traffic Light FSM

- As for the timing of all this...



ECE342 – Computer Hardware

Lecture 27

Bruno Korst, P.Eng.

Agenda

- ASM
- FSM Implementation

Algorithmic State Machines

- This is a structural approach to developing the design of finite state machines through the use of a graphical representation
 - The state diagram is also a graphical approach, but does not have details about the algorithm
 - Algorithm details will help writing the state machine in code (be it C or HDL), as well as helping with interaction with other designers while working on a project

Algorithmic State Machines

- The ASM chart focuses on the activity that unfolds during the operation, to describe the state of the machine
 - It describes **both** an algorithm and a state machine
- In the design process, the ASM comes after the state diagram is defined, and before the design is written in code
- The graphical representation is **similar to a flow chart**, which may be typically used to assist in software or in planning a procedure.

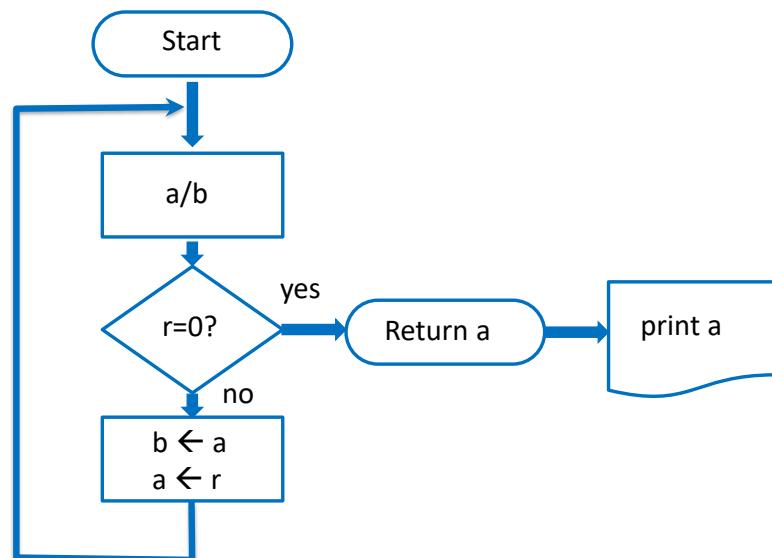
Algorithmic State Machines

- Example: say you want to find the greatest common divisor (GCD) using the Euclidian approach
- We describe it in words
 - Have two positive integers (a,b)
 - Find the remainder of a/b
 - Assign to a variable r
 - Test it for zero
 - If so, terminate, it's a
 - Else, interchange
 - Substitute variables
 - Repeat

```
int gcd(int a, int b)
{
    int r;
    while (b != 0)
    {
        r = a % b;
        a = b;
        b = r;
    }
    return a;
}
```

Algorithmic State Machines

- This can be represented graphically in a flow chart as follows



Algorithmic State Machines

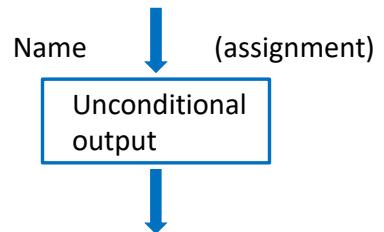
- ASM is a chart similar to a flow chart, containing three elements
 - State box – contains the state name, state code (binary) and state outputs
 - This is the state of the machine inbetween events
 - Decision box – describes inputs to the state machine
 - Has two exit paths: true or false
 - Conditional output box
 - Describes other outputs that are **dependent on one or more inputs**
 - This is a “Mealy output”

Algorithmic State Machines

- An ASM block is a structure consisting of one state box and a network of decision boxes and conditional output boxes
 - It has ONE entrance, and any number of exit paths
- The ASM chart consists of one or more ASM blocks
 - The blocks are equivalent to the states in a sequential machine; but contain more detail

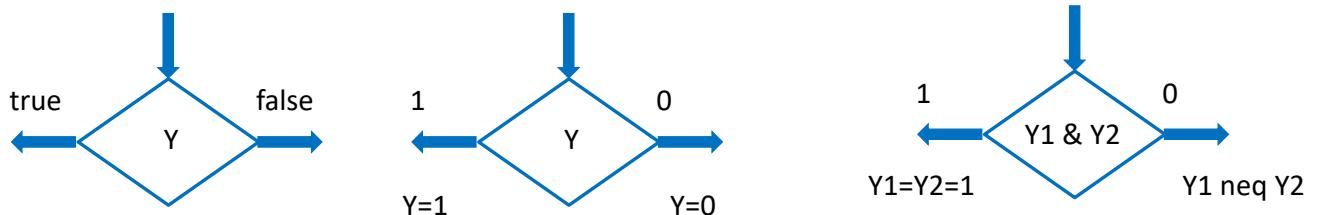
Algorithmic State Machines

- Details of each box
- State box – indicates the state of the FSM, similar to a note on a state transition diagram
 - It has only one exit
 - Name of the state at top left, state assignment (optional) on the top right
 - Moore-type outputs are listed inside
 - These are the current state outputs
 - Write any action taken
 - Write only signals that are to be logic 1
 - Those which are “asserted”



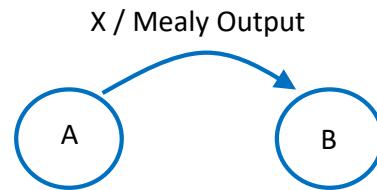
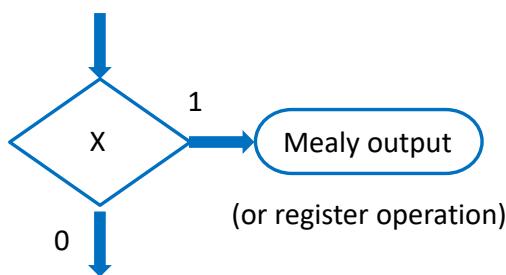
Algorithmic State Machines

- Decision Box
 - Represented by a diamond, showing the condition to be tested and the chosen paths
 - The condition may be an asserted signal or an expression
 - There is one input path and two exit paths
 - Usually follow a state box



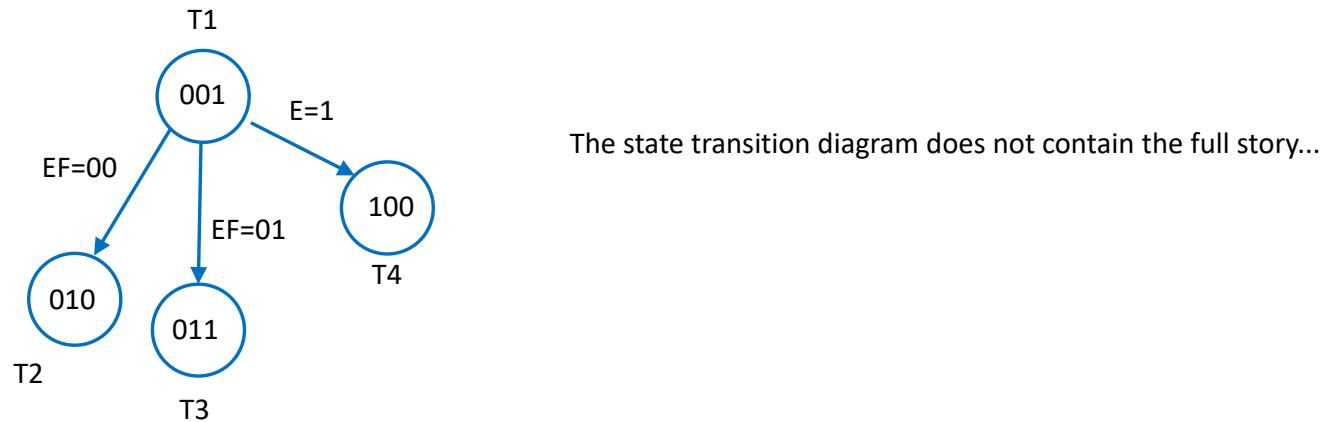
Algorithmic State Machines

- Conditional (output) box -- oval in shape
 - lists Mealy type outputs – depend on present state AND input
 - May list the operations as well
 - Can only follow decision boxes



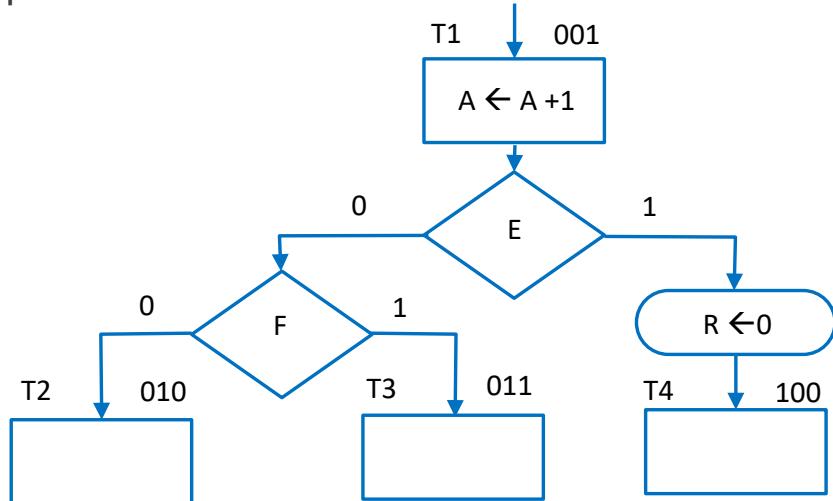
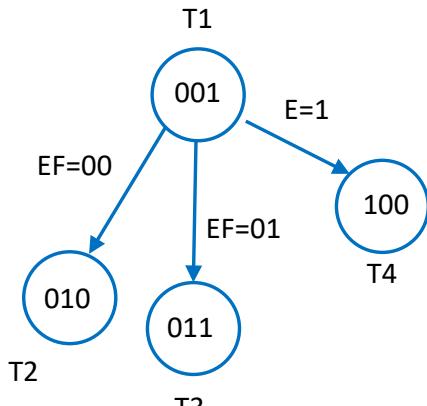
Algorithmic State Machines

- Note that no new information is held on the ASM graph, just more detail about the system being implemented



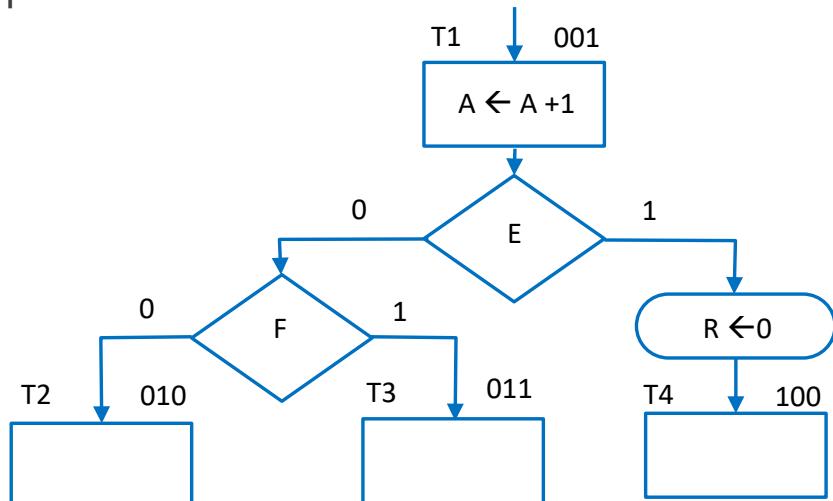
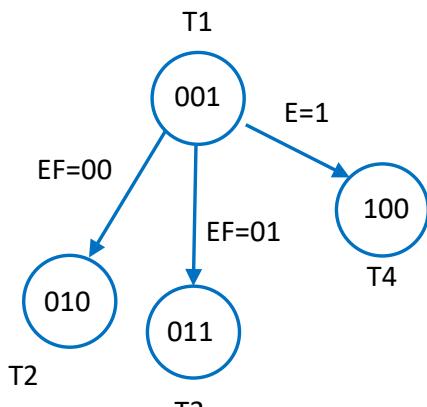
Algorithmic State Machines

- Note that no new information is held on the ASM graph, just more detail about the system being implemented



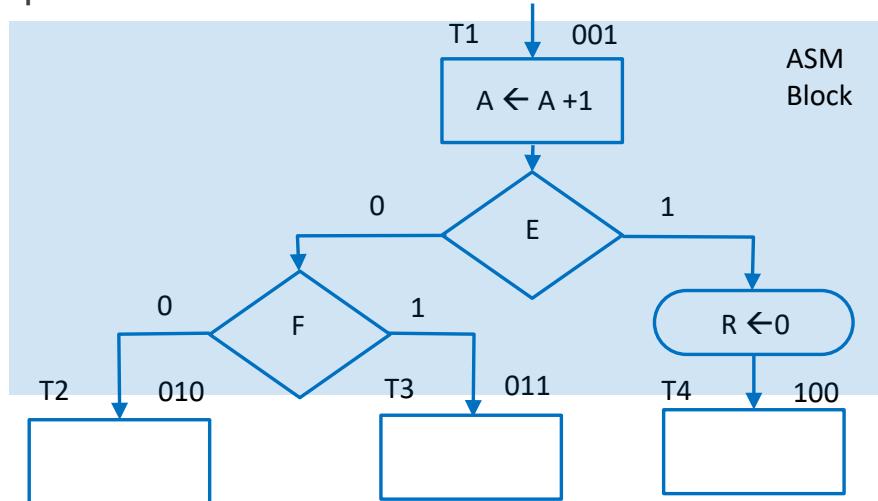
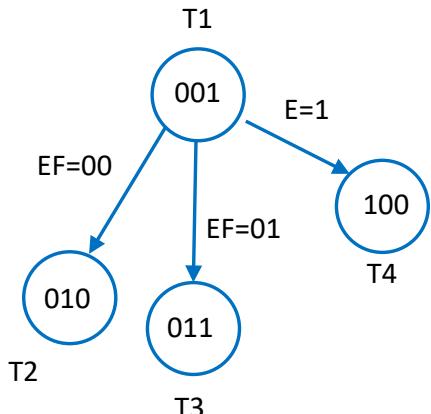
Algorithmic State Machines

- Note that no new information is held on the ASM graph, just more detail about the system being implemented



Algorithmic State Machines

- Note that no new information is held on the ASM graph, just more detail about the system being implemented



FSM Demo

- Few examples

ECE342 – Computer Hardware

Lecture 29

Bruno Korst, P.Eng.

Agenda

- FSM state minimization
- Exercise with state transition diagram
- Real-Time Operating System (intro)

FSM

- Recap
 - State – system's condition at a particular point in time
 - It's a summary of the past
 - Encodes everything about the past that has an effect on the system's reaction to current and future inputs
 - Input – determine whether the transition may be taken upon a reaction
 - Output – it's a reaction
 - what is instantaneously triggered by the environment in which the system operates

FSM

- A state machine is a collection of states (steps) selected for execution based on the value of a state variable
 - The state variable is used to determine which state is to be executed next
 - This variable tells us “where we are”
- Using the FSM we can put other functions between the calls to the state machine
 - Check timer flag, poll keyboard, monitor serial port, etc.

FSM

- Basic FSM types
 - Execution indexed – uses a switch/case and if/else within the case statement
 - Sequential execution
 - Data indexed – data (input) drives transition between state variables
 - Ex: monitoring temperature, gas, volume etc
 - Each input from a separate channel, with own calibration/limits
 - Hybrid – combines both execution and data driven
 - Ex: transition to each state brings a payload (information) that will dictate what the FSM will do next

FSM

- Hybrid type (cont'd)
 - Example: software based serial communication
 - Frame has start bit, 8 bit payload, parity, stop bit)
 - This determines what the program will do once an input causes the FSM to be/arrive at a state
 - The state output may be a value written somewhere
 - The state output may be a change of state in another state machine

FSM – State Minimization

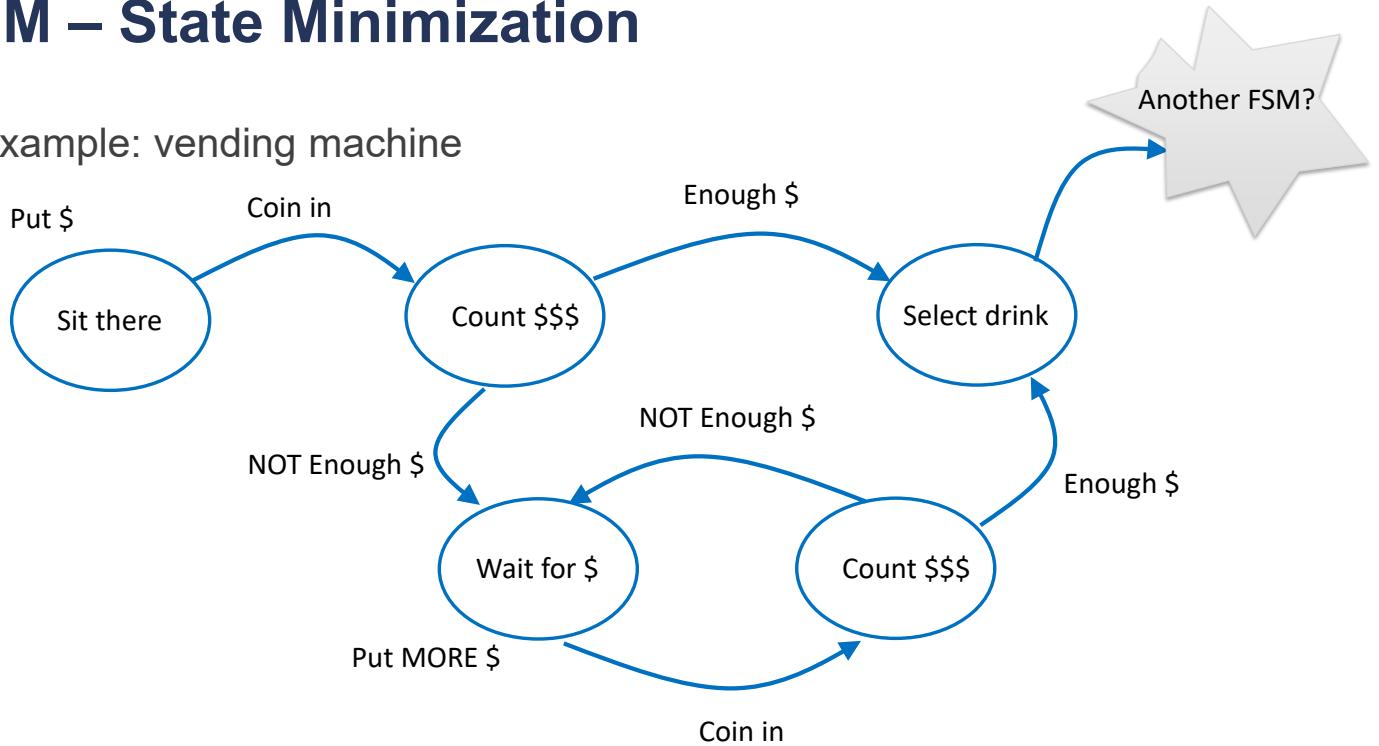
- Combining states in an FSM
 - Any two states can be combined IF
 - They have the same output

AND

- Transition from them is to the same set of states if given the same input

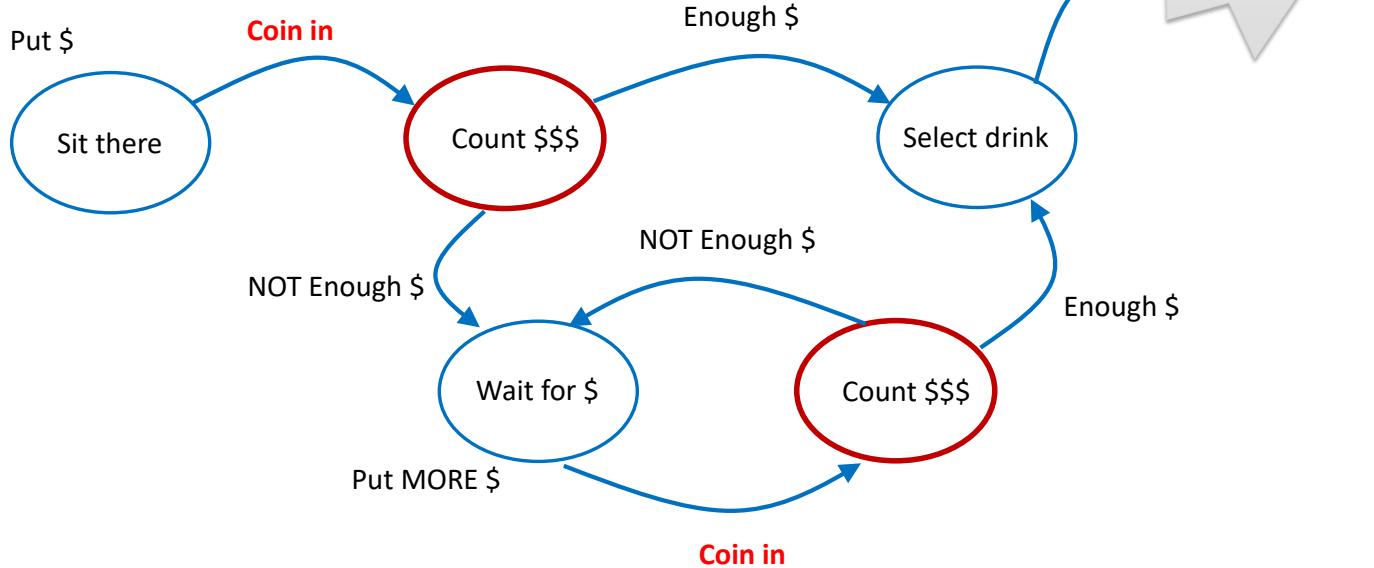
FSM – State Minimization

- Example: vending machine



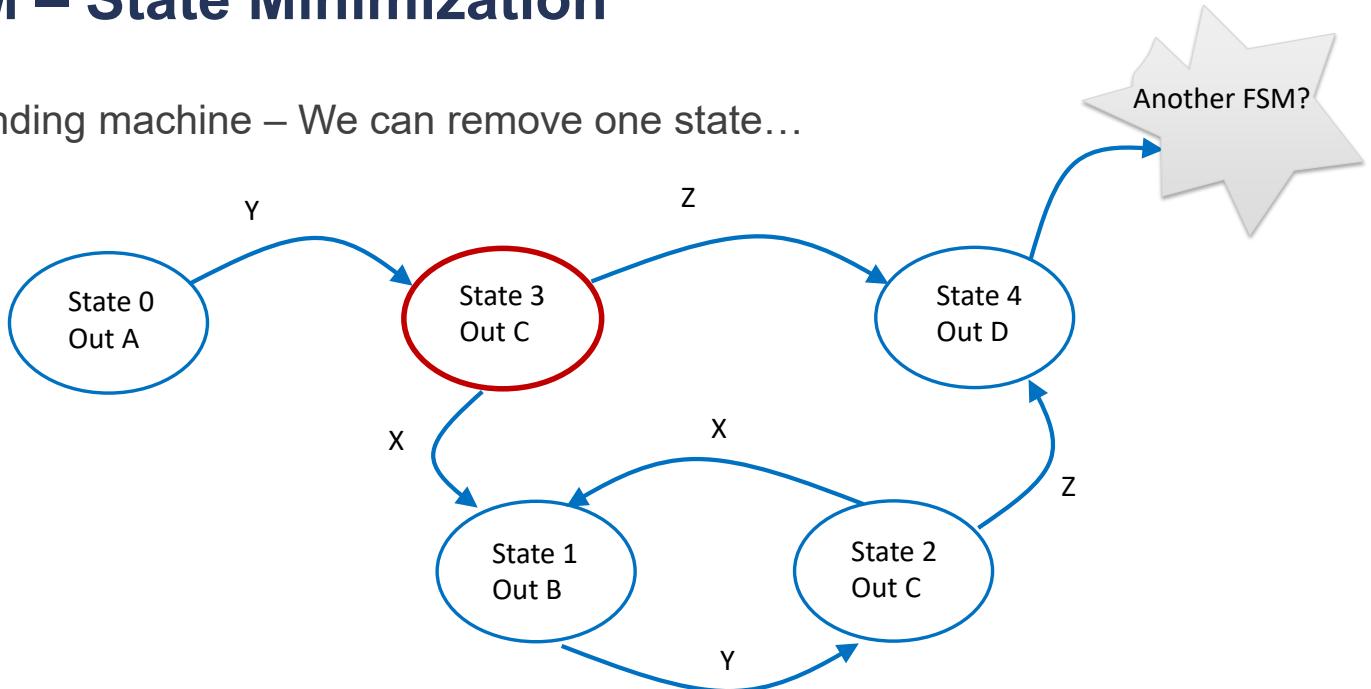
FSM – State Minimization

- Example: vending machine



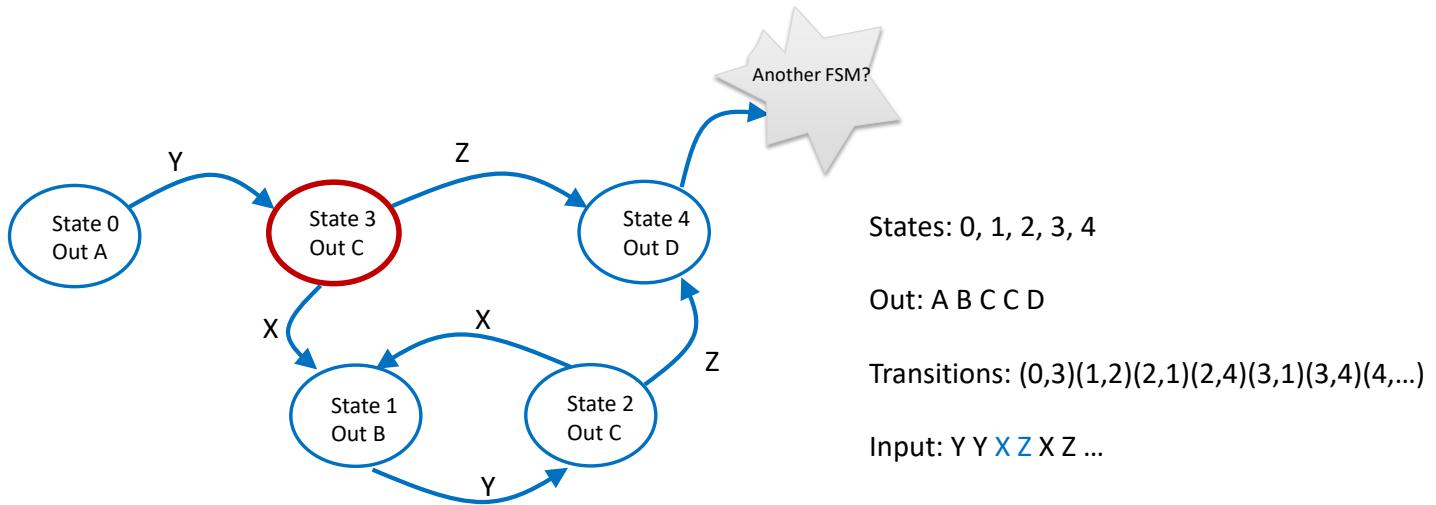
FSM – State Minimization

- Vending machine – We can remove one state...



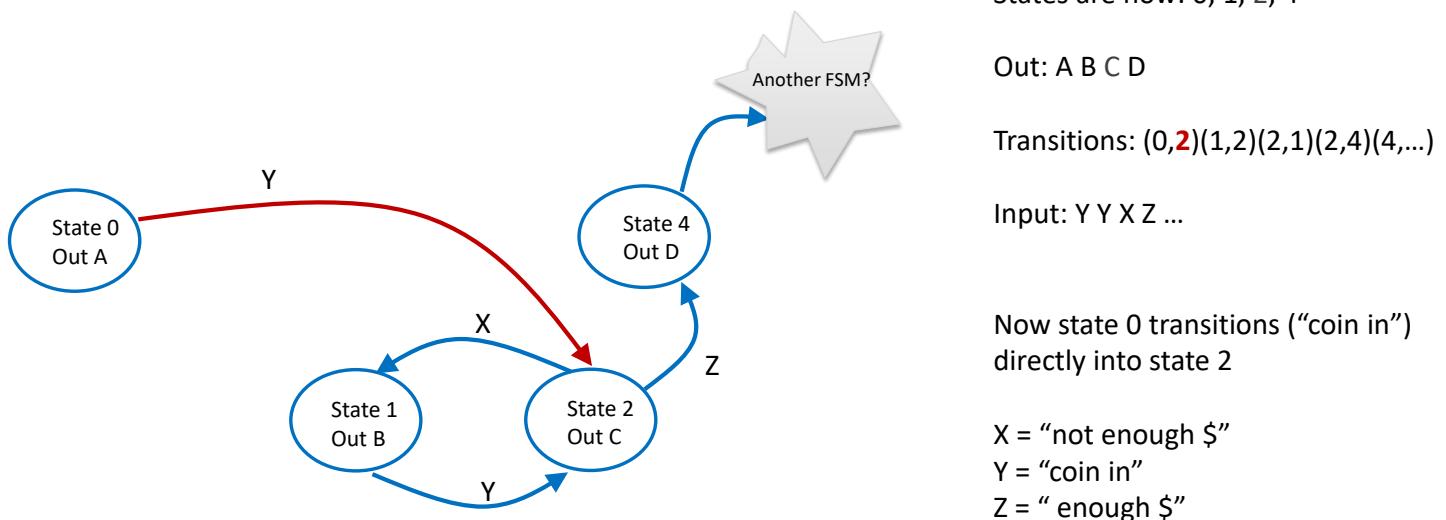
FSM – State Minimization

- Vending machine – We can remove one state...



FSM – State Minimization

- Vending machine – Remove (former) State 3



FSM – State Minimization

- The previous approach is fine for a few states; another approach is through the identification and grouping of equivalent states
- Say we have an FSM with the following state transition table

Present State	Next State w=0	Next state w=1	Output Z
A	B	C	1
B	D	F	1
C	F	E	0
D	B	G	1
E	F	C	0
F	E	D	0
G	F	G	0

FSM – State Minimization

Present State	Next State w=0	Next state w=1	Output Z
A	B	C	1
B	D	F	1
C	F	E	0
D	B	G	1
E	F	C	0
F	E	D	0
G	F	G	0

- Group states with similar output → (ABD) (CEFG)
 - If they have the same output, they **may** be equivalent...
 - Process: to eliminate from the groupings the states that ARE NOT equivalent

FSM – State Minimization

- Look at the next transition for each group, given each input
- Group ABD

$W = 0$	$W = 1$
$A \rightarrow B$	$A \rightarrow C$
$B \rightarrow D$	$B \rightarrow F$
$D \rightarrow B$	$D \rightarrow G$

B, D and B
ALL belong to
group ABD C, F and G
ALL belong to
group ABD

- For each input, if all transition to nodes grouped with the same output, leave them as is

FSM – State Minimization

- Group CEFG

$W = 0$	$W = 1$
$C \rightarrow F$	$C \rightarrow E$
$E \rightarrow F$	$E \rightarrow C$
$F \rightarrow E$	$F \rightarrow D$
$G \rightarrow F$	$G \rightarrow G$

All belong to
group CEFG Two different groups
(one transitions to group ABD)

- State F is NOT equivalent to states CEG, therefore it is grouped separately
- The new grouping is now (ABD) (CEG) (F)
 - The **approach is repeated**, knowing that F now belongs to a separate group

FSM – State Minimization

- Repeating the approach
- Group ABD

W = 0	W = 1
A → B	A → C
B → D	B → F
D → B	D → G

B, D and B C and G belong
ALL belong to To one group
group ABD F belongs to another

- F belongs to a separate group, so state B is NOT equivalent to A and D

FSM – State Minimization

- Repeating the approach for the new group (CEG) – that is, without the F

W = 0	W = 1
C → F	C → E
E → F	E → C
G → F	G → G

ALL belong to E, C and G belong
One group (F) to one group (CEG)

- F belongs to a separate group, and now B belongs to a separate group
 - The new grouping is now (AD) (B) (CEG) (F)

FSM – State Minimization

- Repeating the approach a second time
- Group AD (since ABD has been broken)

$W = 0$	$W = 1$
$A \rightarrow B$	$A \rightarrow C$
$D \rightarrow B$	$D \rightarrow G$

ALL belong to One group ALL belong to One group

- Group CEG

$W = 0$	$W = 1$
$C \rightarrow F$	$C \rightarrow E$
$E \rightarrow F$	$E \rightarrow C$
$G \rightarrow F$	$G \rightarrow G$

ALL belong to One group (F) E, C and G belong to one group (CEG)

FSM – State Minimization

- Since we no longer find a state that does NOT belong in a grouping, we conclude that the states in the groups are equivalent
- We will minimize the FSM, to have only one state accounting for the states found to be equivalent
- The final groupings are: (AD) (B) (CEG) (F)
 - Therefore state A is equivalent to state D → will call the new state P
 - State C is equivalent to state E, which is equivalent to state G → new state Q

FSM – State Minimization

- The new, minimized state machine will have 4 states: P, B, Q and F

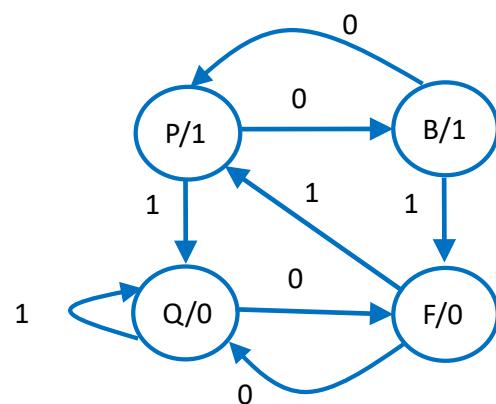
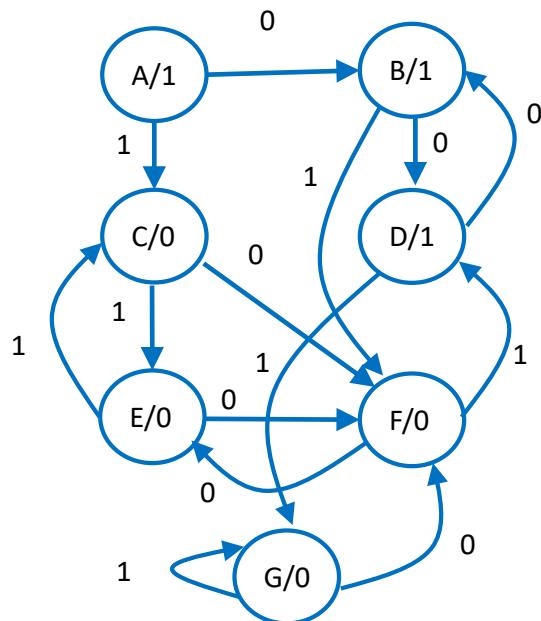
Present	W=0 next	W=1 next	Output Z
A	B	C	1
B	D	F	1
C	F	E	0
D	B	G	1
E	F	C	0
F	E	D	0
G	F	G	0

Original State Transition Table

New State Transition Table

Present	W=0 next	W=1 next	Output Z
P	B	Q	1
B	P	F	1
Q	F	Q	0
F	Q	P	0

FSM – State Minimization



FSM – State Minimization

- As the number of states increase in an FSM, one may have an FSM within an FSM
 - No simplification by hand
- Depending on the case, going from Moore to Mealy may reduce the number of states needed, but it may increase complexity (output functions)
- Also, partitioning approach may be less useful when don't care conditions appear on the table

Exercise

- Let us draw the state transition diagram of a typical washing machine

FSM – Exercise

- Examples of what we are controlling
 - Hot water valve
 - Cold water valve
 - Pump (drain)
 - Motor, etc. etc.
- Say we are at the drain state
 - We'd call upon a function

```
void drain (void){  
    pump = ON;  
}
```

FSM – Exercise

- Likewise, if the water is low
 - Data from the sensor will indicate
 - Need an interrupt to serve the sensor and send to appropriate next state

```
void ISR_waterLow (void){  
    pump = OFF;  
    if (state == drainWash)  
        state = fillRinse;  
    elseif (state == drainRinse)  
        state == spin;  
    elseif (state == drain)  
        state == idle;  
}
```

- What if something fails? Do we need redundancy? Do we use a counter? Timer?

Real-Time Operating Systems

- One should think of “real-time” as meeting deadlines
- Real time computing is not necessarily fast computing
- Real time computing is, however, predictable computing
 - You know that a task must be done by a certain time
 - The collection of data must happen within an interval
 - The processing of an algorithm must be done within an interval
 - The handling of an interrupt must be done before the next interrupt arrives

Real-Time Operating Systems

- Even with limited computing resources, an application CAN be called “real-time”
 - If the purpose is met within the time constraints, it was done “in real time”
- What we are looking for is:
 - The computation done must be logically correct
 - The results are produced on time
- The response time of a real-time system is constant over all iterations, which may not be true for a non-real-time system
 - “soft-real-time” for small deviations in response time

Real-Time Operating Systems

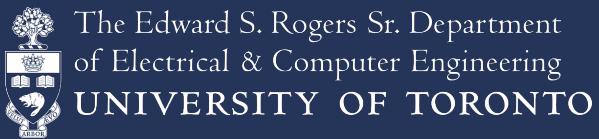
- Some systems must operate under “hard real time”, in which the tasks must be completed within a certain schedule or it results in total failure
 - Air bag
 - Anti-lock brakes
 - Fly-by-wire
- Soft real time
 - Streaming, stock market sites, home temperature monitor

Real-Time Operating Systems

- A real time operating system is needed when there are critical operations that need scheduling
 - Need precise timing and high degree of reliability
 - Need to handle interrupts and system exceptions timely
 - Need to ensure critical sections of code are done within the allocated time
- General purpose operating systems care about throughput
 - Tasks are organized to maximize throughput
 - This is the case with personal computers

Real-Time Operating Systems

- General purpose operating systems organize tasks to keep throughput
 - One may have a high-priority task slightly delayed to favour a number of low priority tasks to keep throughput
 - Their policies do NOT guarantee that the execution of a high-priority process will necessarily have preference over a low priority task
- Real time operating systems execute processes in order of priority, so a low priority task will lose to a higher priority task
 - This does not necessarily mean low throughput, but it means “predictable”



ECE342 – Computer Hardware

Lecture 30

Bruno Korst, P.Eng.

Agenda

- Real-Time Operating Systems
 - References: Amos, B. – Hands-On RTOS with Microcontrollers, 2020
freertos.org

Real-Time Operating Systems

- One should think of “real-time” as meeting deadlines
- Real time computing is not necessarily fast computing
- Real time computing is, however, predictable computing
 - You know that a task must be done by a certain time
 - The collection of data must happen within an interval
 - The processing of an algorithm must be done within an interval
 - The handling of an interrupt must be done before the next interrupt arrives

Real-Time Operating Systems

- Even with limited computing resources, an application CAN be called “real-time”
 - If the purpose is met within the time constraints, it was done “in real time”
- What we are looking for is:
 - The computation done must be logically correct
 - The results are produced on time
- The response time of a real-time system is constant over all iterations, which may not be true for a non-real-time system
 - “soft-real-time” for small deviations in response time

Real-Time Operating Systems

- Some systems must operate under “hard real time”, in which the tasks must be completed within a certain schedule or it results in total failure
 - Air bag
 - Anti-lock brakes
 - Fly-by-wire
- Soft real time
 - Streaming, stock market sites, home temperature monitor

Real-Time Operating Systems

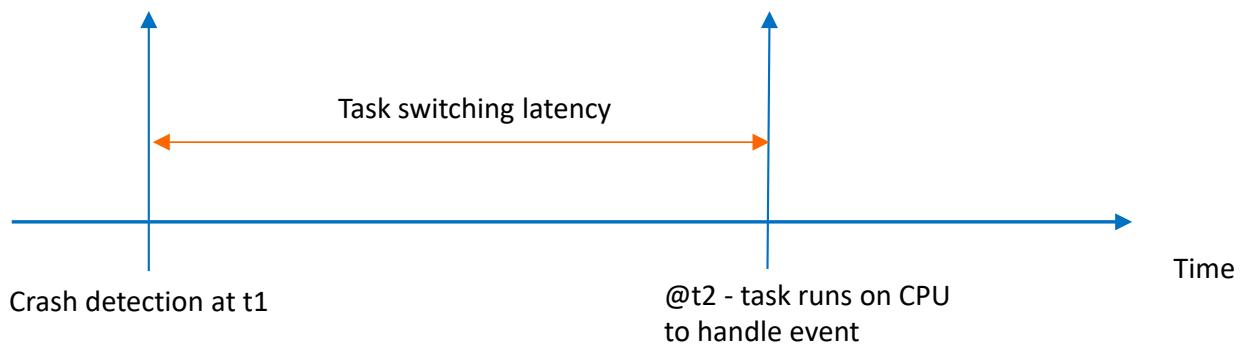
- A real time operating system is needed when there are critical operations that need scheduling
 - Need precise timing and high degree of reliability
 - Need to handle interrupts and system exceptions timely
 - Need to ensure critical sections of code are done within the allocated time
- General purpose operating systems care about throughput
 - Tasks are organized to maximize throughput
 - This is the case with personal computers

Real-Time Operating Systems

- General purpose operating systems organize tasks to keep throughput
 - One may have a high-priority task slightly delayed to favour a number of low priority tasks to keep throughput
 - Their policies do NOT guarantee that the execution of a high-priority process will necessarily have preference over a low priority task
- Real time operating systems execute processes in order of priority, so a low priority task will lose to a higher priority task
 - This does not necessarily mean low throughput, but it means “predictable”

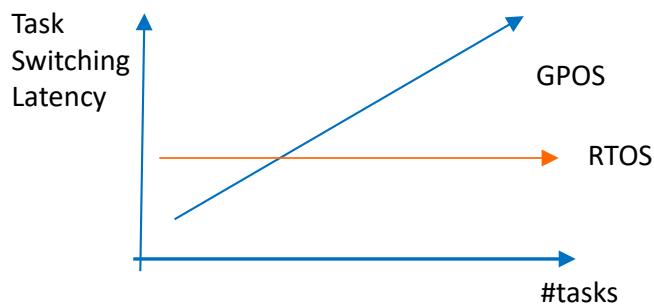
Real-Time Operating Systems

- Latency
 - In general, it is the time elapsing between a stimulus and a response to it
- For an OS, task switching latency is the time gap between the triggering of an event and the time for the task handling it to be allowed to run on the CPU



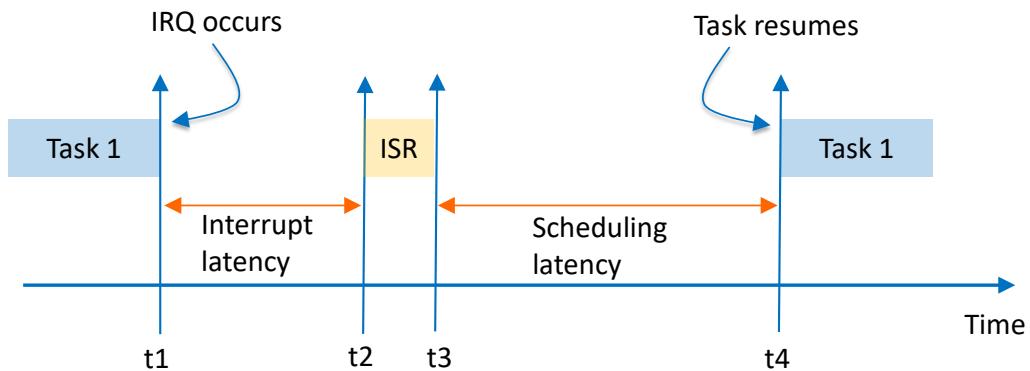
Real-Time Operating Systems

- Task Switching Latency
 - GPOS – this time may vary
 - RTOS – the latency is always time bounded and must not vary
- RTOS is the tool that will assist in designing a system where this latency does not vary as the number of tasks increase



Real-Time Operating Systems

- Interrupt Latency – time it takes between the interrupt request taking place and the beginning of the ISR execution.
 - The task must resume afterwards, as determined by a scheduler



Real-Time Operating Systems

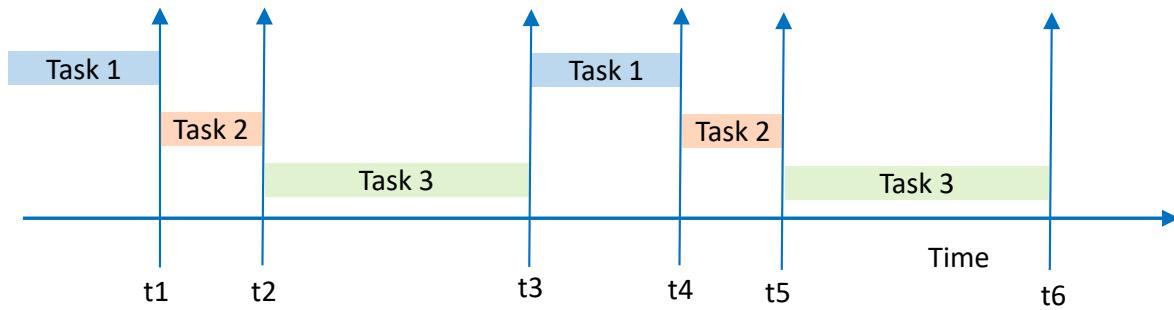
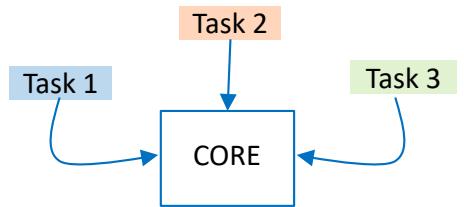
- Interrupt latency and scheduling latency for RTOS are both as small as possible, and are time bounded
 - For GPOS these depend on system load and may vary
- Multitasking
 - Multiple tasks need to be completed between a specific time interval, and each task takes a certain time to be completed
 - The RTOS will measure time in “ticks”, following a strict temporal accuracy.
 - A timer interrupt keeps control of the tick count

Real-Time Operating Systems

- Analogy with daily activities
 - Two ways for us to get all daily tasks done in a day
 - “slice” some amount of time for each activity/task
 - or
 - Delegate tasks to multiple assistants
 - Same in software, as a “task” is a piece of code that can be scheduled for execution
- Ex: temperature monitor system
 - Task1 – sensor read
 - Task2 – update display
 - Task3 – process user input

Real-Time Operating Systems

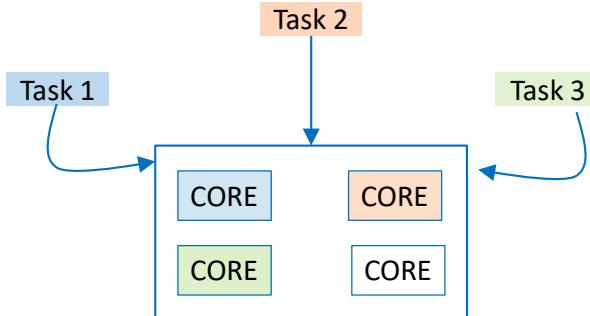
- Say we only have one CPU to get the task done
 - We depend on a scheduler for this to work



- A task can run over multiple “ticks”

Real-Time Operating Systems

- Should we have multiple cores (say 4), the tasks can be delegated each to a separate core
 - All three tasks are run concurrently, without the need of a scheduler

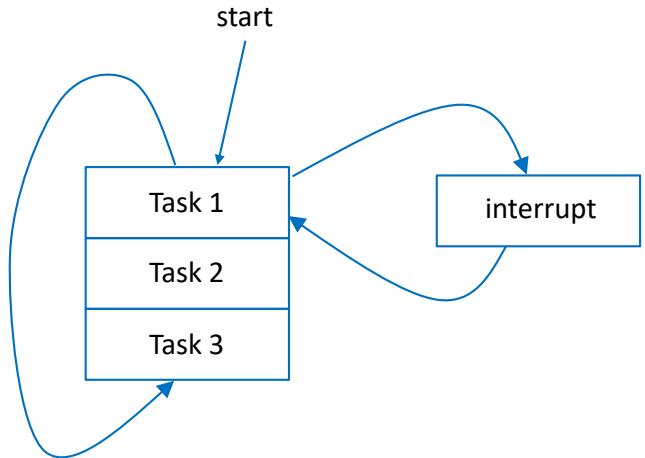


- In embedded systems, the typical case is one core
 - The RTOS provides the resources for the execution of multiple tasks to appear as though it is run simultaneously

Real-Time Operating Systems

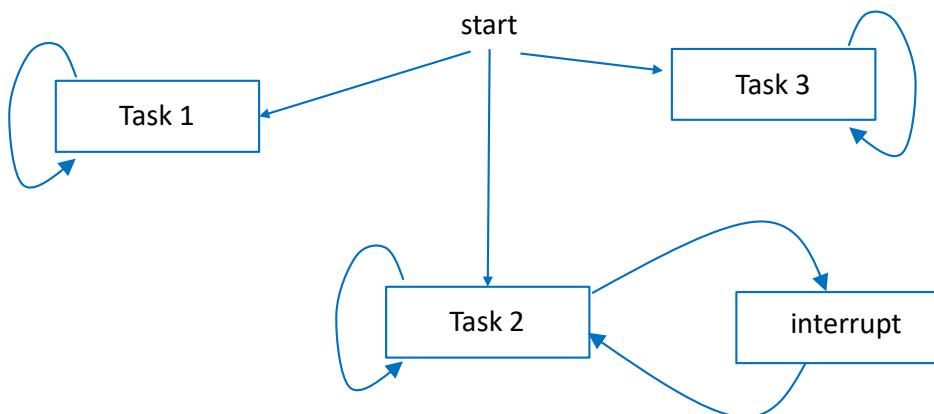
- For the first case, our execution is implemented with a “super loop”

- Typical in embedded systems
- Works well for slower time interval
 - Interruptions to collect data
- If task completion time takes longer...
 - Lose data
 - Increase latency in returning



Real-Time Operating Systems

- RTOS allows us to run the tasks “in parallel”, each of the tasks with its own infinite loop



- They are called after a scheduler is started

Real-Time Operating Systems

In terms of the programming...

Super Loop

```
func1()
{
    //functionality 1
}

func2()
{
    //functionality 2
}

func3()
{
    //functionality 3
}

main( )
{
    while(1)
    {
        func1();
        func2();
        func3();
    }
}
```

RTOS Tasks

```
Task1( )
{
    while(1)
    {
        //functionality of task 1
    }
}

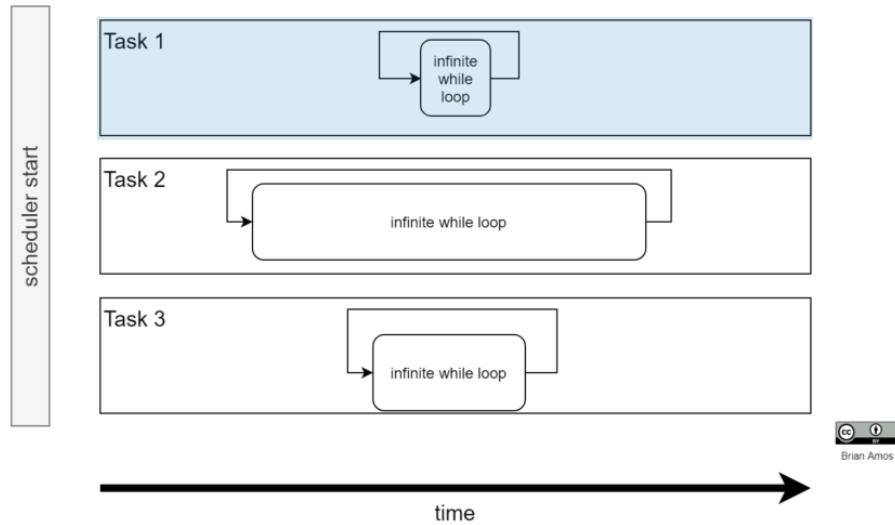
Task2( )
{
    while(1)
    {
        //functionality of task 2
    }
}

Task3( )
{
    while(1)
    {
        //functionality of task 3
    }
}

main( )
{
    createTask1(&Task1);
    createTask2(&Task2);
    createTask3(&Task3);
    StartScheduler();    //never returns
}
```

Real-Time Operating Systems

Basic Task Setup (Programming Model)



Real-Time Operating Systems

- When use RTOS or “super loop”?
 - If a module is resource-intensive (BLE) and need to combine it with buttons, memory r/w and control actuators, RTOS provides the scheduling
- Priorities
 - GPOS task priority can be somewhat flexible
 - High priority tasks held to allow low priority tasks to be handled
 - Focus on throughput
 - RTOS – time constraint for completion is rigid
 - Priority is based on preemptive scheduling

Real-Time Operating Systems

- The **scheduler**
 - Decides which task executes when
 - Can suspend and resume tasks
 - Tasks can suspend themselves by delaying their execution (**sleep**) for a fixed period, or wait for a resource to become available or an event to occur (**block**)
 - No processing time is allocated when a task is sleeping or blocked
- Each task will use resources as needed, and the set of resources is called the **task execution context**

Real-Time Operating Systems

- Context switching
 - Multiple times being executed → switching needs to take place
 - Task 1 switches out to Task2 at some point
 - After some time execution switches back into Task1
 - The context of the tasks need to be saved as the task is suspended for a switch to happen.
 - Example: switch happens in the middle of an ADD instruction (register set but operation not executed)
 - The context needs to be identical to what it was when the task resumes

Real-Time Operating Systems

- Scheduling Policy
 - As real-time systems are constrained by deadlines, the scheduling policy must take into account priorities assigned by the designer
 - Highest priority gets processing time
 - Question: what is the consequence of a missed deadline?
 - In switching between tasks with different priorities, the RTOS may create **idle times**, to make sure the deadlines given are met.

Real-Time Operating Systems

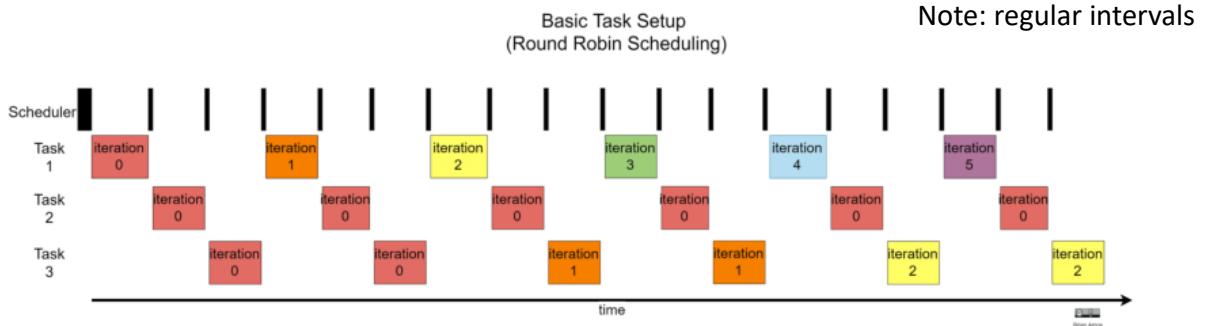
- Super-loop model
 - Tasks with different durations are executed sequentially, and the longer task may interfere with the other(s) of shorter duration
- Task-based model
 - programmer does not need to be immediately concerned with the duration of each task (that is, the longer task interfering on the execution of the shortest)
 - Scheduler will take care of how to split tasks with no loss
 - The scheduler has control of the processor use
 - It seems as though the tasks are executed in parallel, but they are actually being scheduled

Real-Time Operating Systems

- Round Robin Scheduling
 - Each task gets a small slice of time to use the processor
 - The task will have full use of the processor while it is executing
 - As long as the task has work to perform, it will execute
 - Scheduler will take care of all the context switching involved in going from one task to the next

Real-Time Operating Systems

- Round Robin Scheduling



- Task 1 is the shortest, goes through 6 iterations of its loop by the time task 2 (the longest) goes through only one iteration
- Notice the similarity with pipelining?
 - On the “super loop” implementation, task 3 would only start after task 2 done!

Real-Time Operating Systems

- Round Robin Scheduling

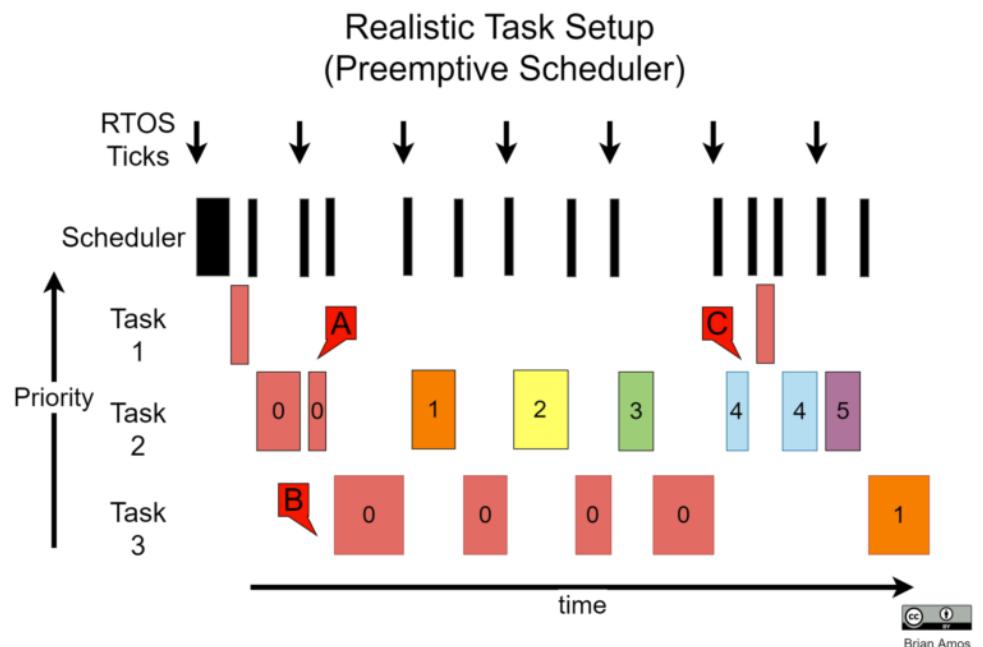
- The cost of this implementation is the number of times the scheduler is invoked at every context switch
 - Though it is efficient, it still imposes an overhead in cycles
 - Power consumption in the switching...
- Question: could you do the same with very strict timing control of a super loop/interrupt approach? (if cycles are scarce, maybe...)
 - The problem here may also be nested interrupts, which may be hard to sort out

Real-Time Operating Systems

- Preemptive based scheduling
 - Gives priority to the most important task
 - Since scheduler does not control interrupts – which happen anyway – this scheduling ensures the most important task is always performed.
 - Interrupts always have a higher priority
 - Previous example: the scheduler runs at regular intervals
 - Scheduler can be called inbetween RTOS “ticks”

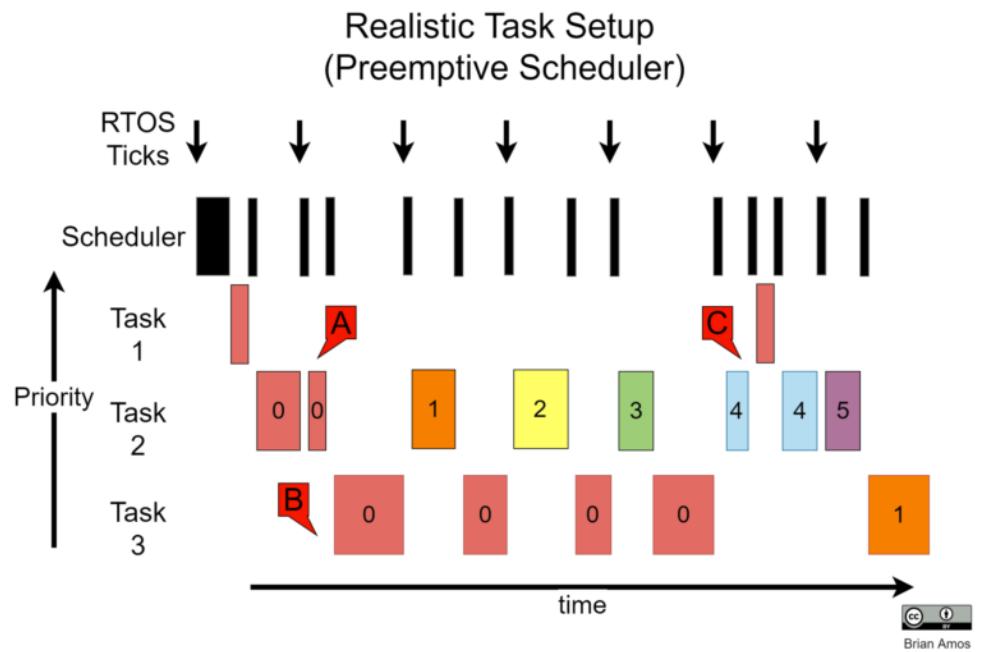
Real-Time Operating Systems

- Note:
 - Task 1 highest priority and very quick exec.
 - Task 2 set up to execute once per tick
 - Task 3 lowest priority, and gets context when there is nothing else to do



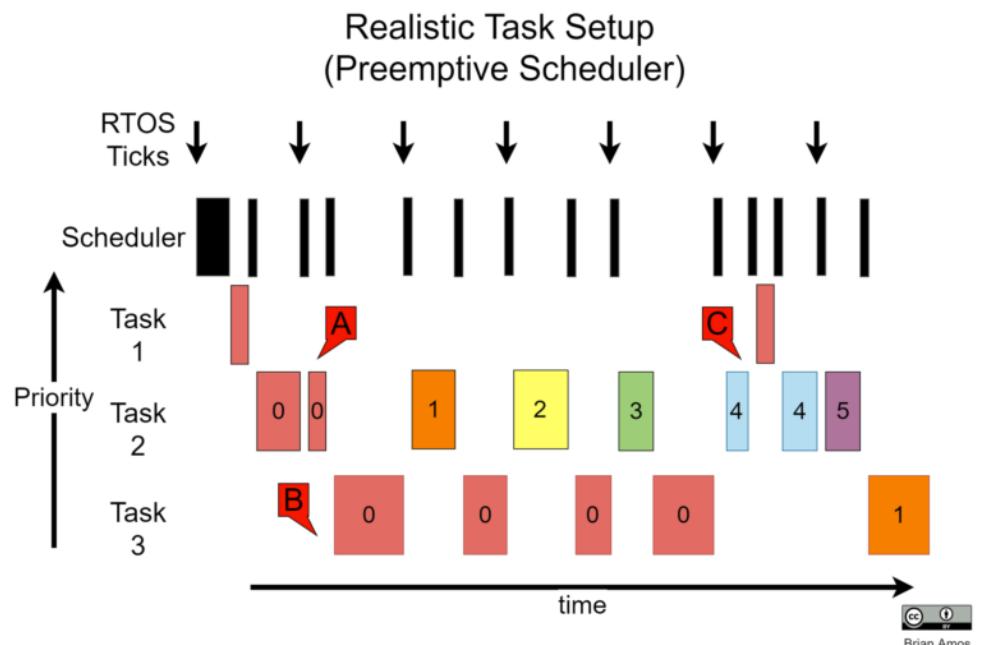
Real-Time Operating Systems

- Note:
- Point A
 - Task 2 was not done and is interrupted by tick
 - Context given back immediately to finish execution of task



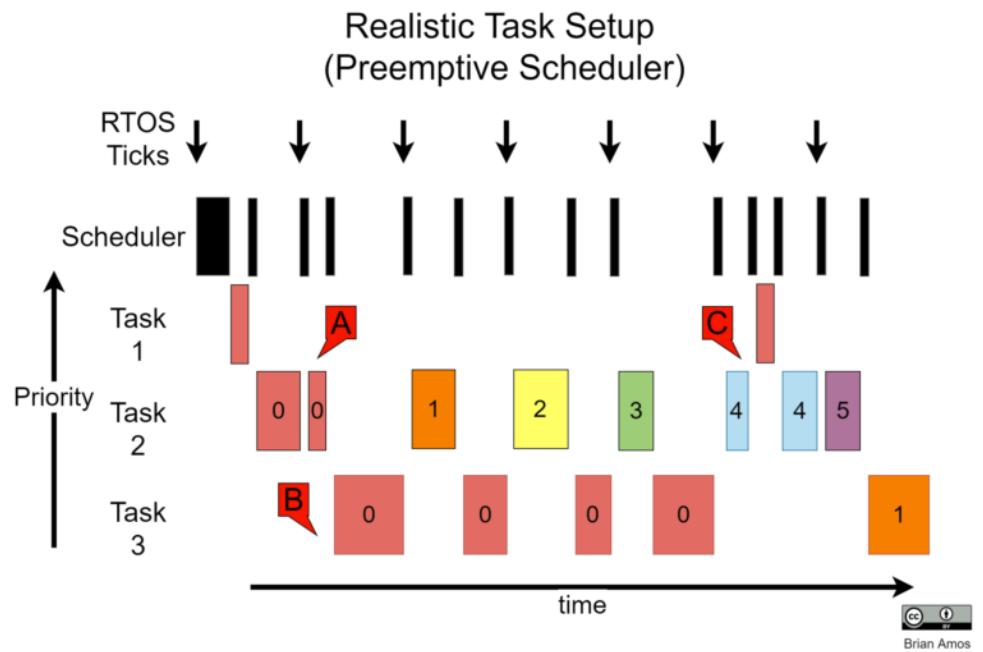
Real-Time Operating Systems

- Note:
- Point B
 - Task 2 done, nothing else *required* to run, task 3 can have processor time



Real-Time Operating Systems

- Note:
- Point C
 - Task 3 is done its first iteration, priority given to task 2 (iteration 4)
 - Task 1 needs to be serviced, so at tick scheduler switches context to task 1

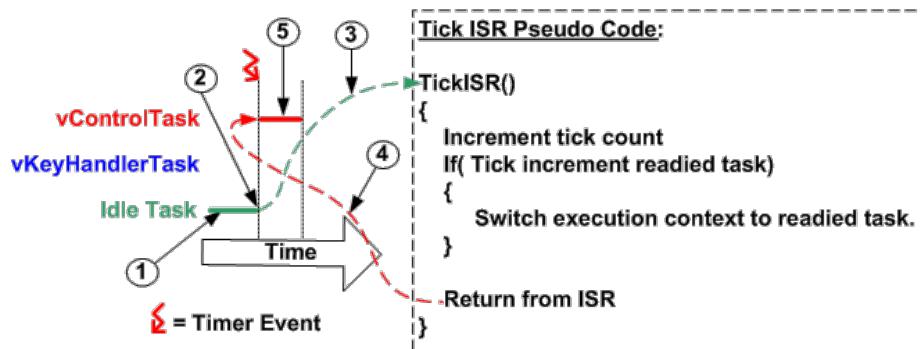


Real-Time Operating Systems

- The RTOS Tick
 - It is how the RT kernel measures time
 - It is a counter variable, incremented by an interrupt
 - This increment establishes the temporal accuracy (resolution) of the system
- At every tick count, kernel checks: is it time to switch execution? What is the priority of the tasks seeking execution?

Real-Time Operating Systems

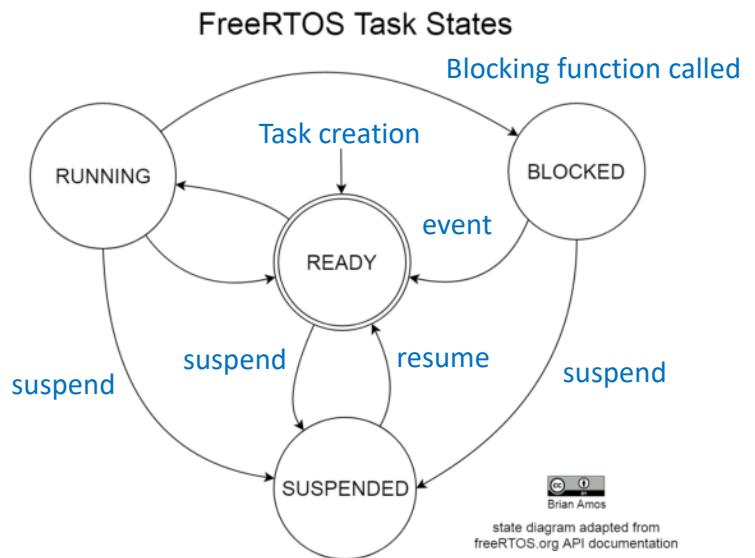
- The RTOS tick



- At 1, kernel idle, at 2 tick takes place, control transfers to ISR (returns at 4)
- The ISR makes function vControlTask ready to run
 - Since it is of higher priority than “idle”, vControlTask starts executing

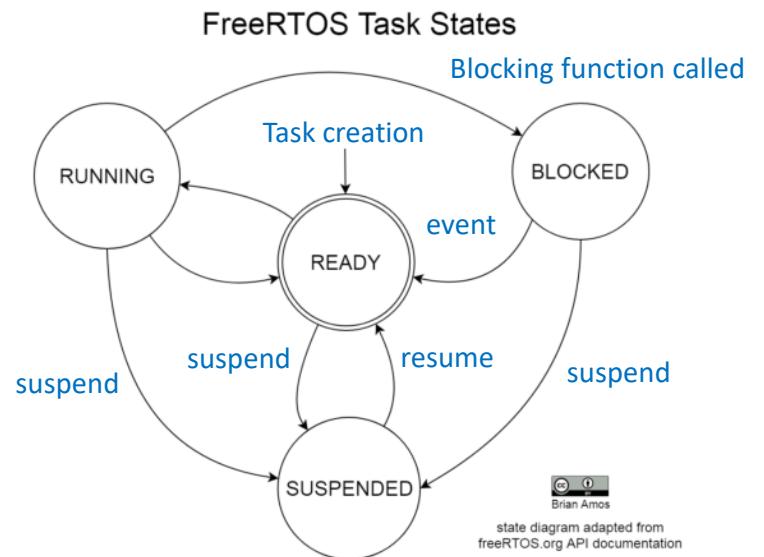
Real-Time Operating Systems

- Task States
- Tasks transition through the states either by scheduler calls or RTOS calls made through the code running



Real-Time Operating Systems

- Running
 - Task performing work
 - Suspend → switched out of context
- Ready – waiting for the scheduler to give processor context
- Blocked – waiting for something
 - It is time bound
- Suspend – ignored by scheduler



ECE342 – Computer Hardware

Lecture 32

Bruno Korst, P.Eng.

Agenda

- Real-Time Operating Systems cont'd
- Communications - Wireless
 - References: Amos, B. – Hands-On RTOS with Microcontrollers, 2020
freertos.org

Real-Time Operating Systems

- Reviewing...
 - One should think of “real-time” as meeting deadlines
 - Deadline and reliability
 - Right result within a set time
 - It runs in a predictable way

Real-Time Operating Systems

- A note on Context switching
 - Task ~ thread – a task becomes a thread as it is executing
 - As the execution switches between thread, the “context” is switched
 - Example: switching from thread 1 to thread 2
 - Save execution state of thread 1
 - Restore the execution state of thread 2
- As a thread is run, registers from the register file are used
 - Also includes stack pointer, link register (return from branch), program counter
- As threads are created, a STACK is allocated for each in memory

Real-Time Operating Systems

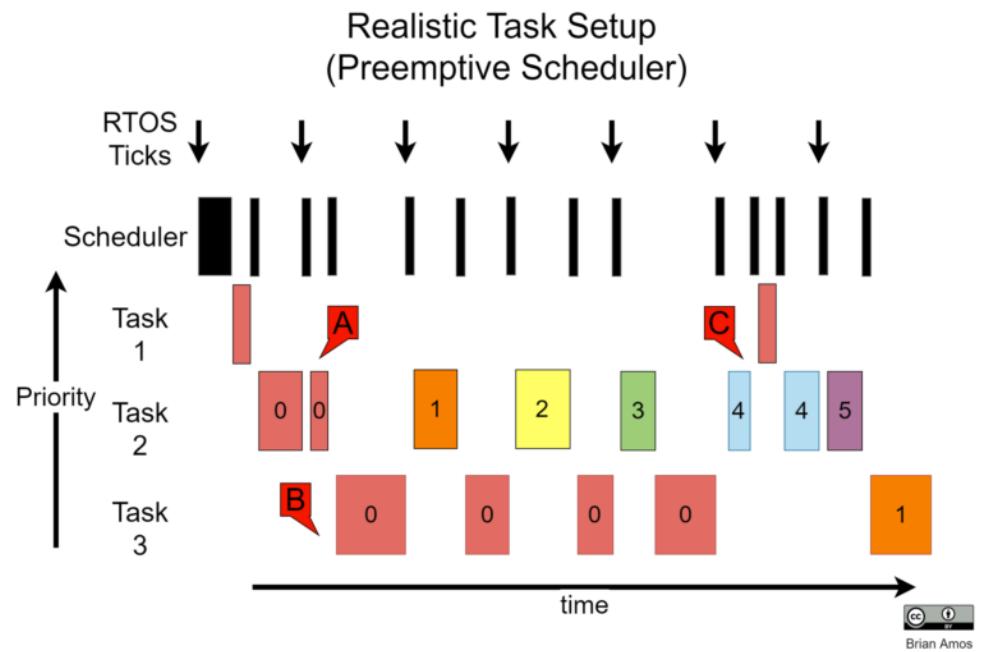
- The stack for the thread will hold the saved execution state of the thread as the context is switched
 - It's an array in memory
 - These values are then restored back to the registers when the context is switched back
- The scheduler will take care of when the switch happens
 - We have seen round robin scheduling
 - Each task taking turns after a fixed amount of time

Real-Time Operating Systems

- Preemptive based scheduling
 - Gives priority to the most important task
 - Since scheduler does not control interrupts – which happen anyway – this scheduling ensures the most important task is always performed.
 - Interrupts always have a higher priority
 - Previous example: the scheduler runs at regular intervals
 - Scheduler can be called inbetween RTOS “ticks”

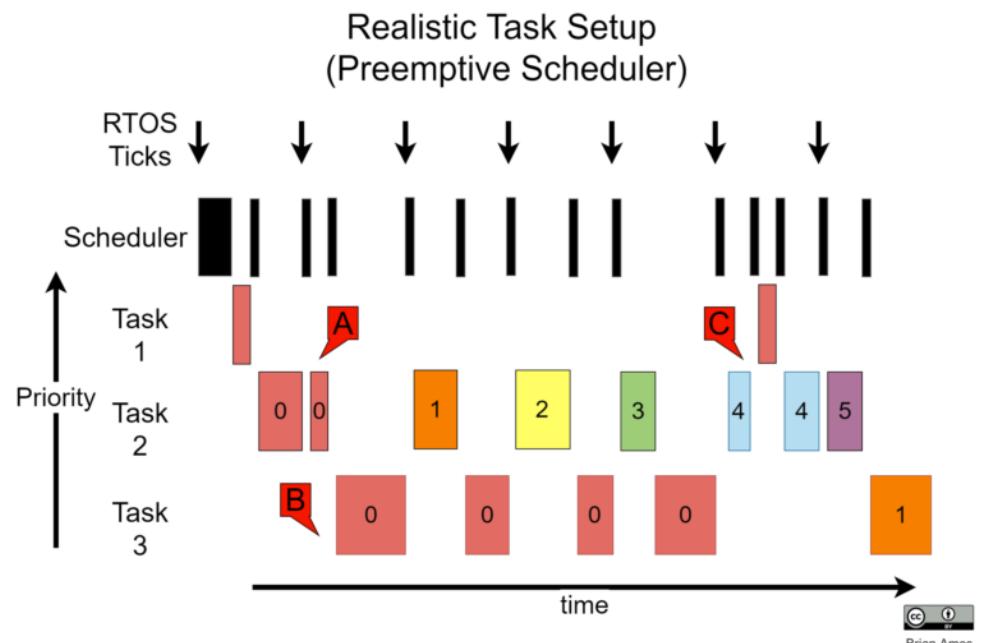
Real-Time Operating Systems

- Note:
 - Task 1 highest priority and very quick exec.
 - Task 2 set up to execute once per tick
 - Task 3 lowest priority, and gets context when there is nothing else to do



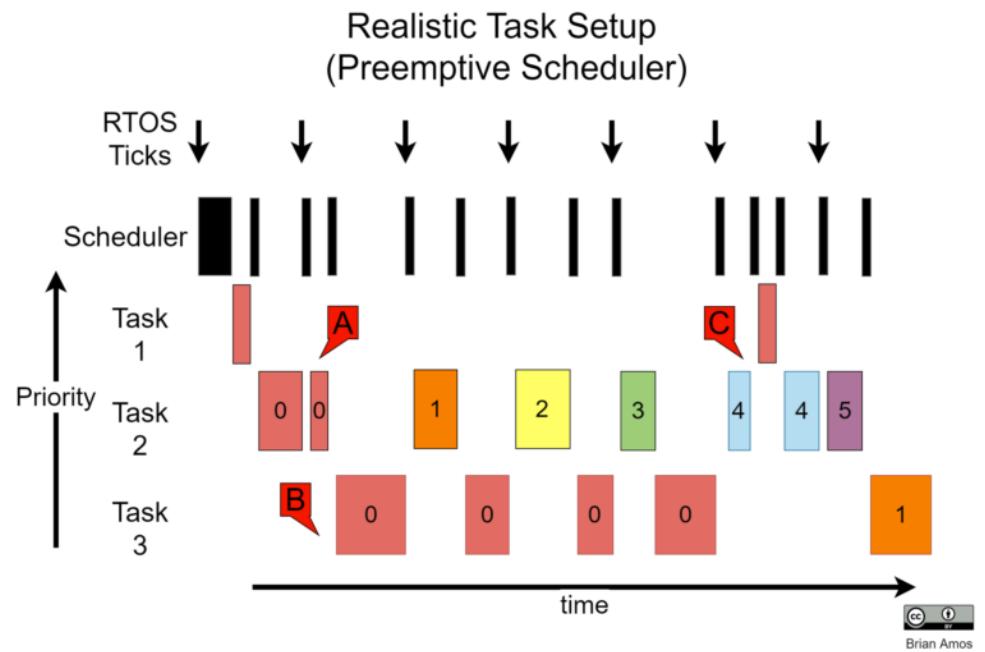
Real-Time Operating Systems

- Note:
- Point A
 - Task 2 was not done and is interrupted by tick
 - Context given back immediately to finish execution of task



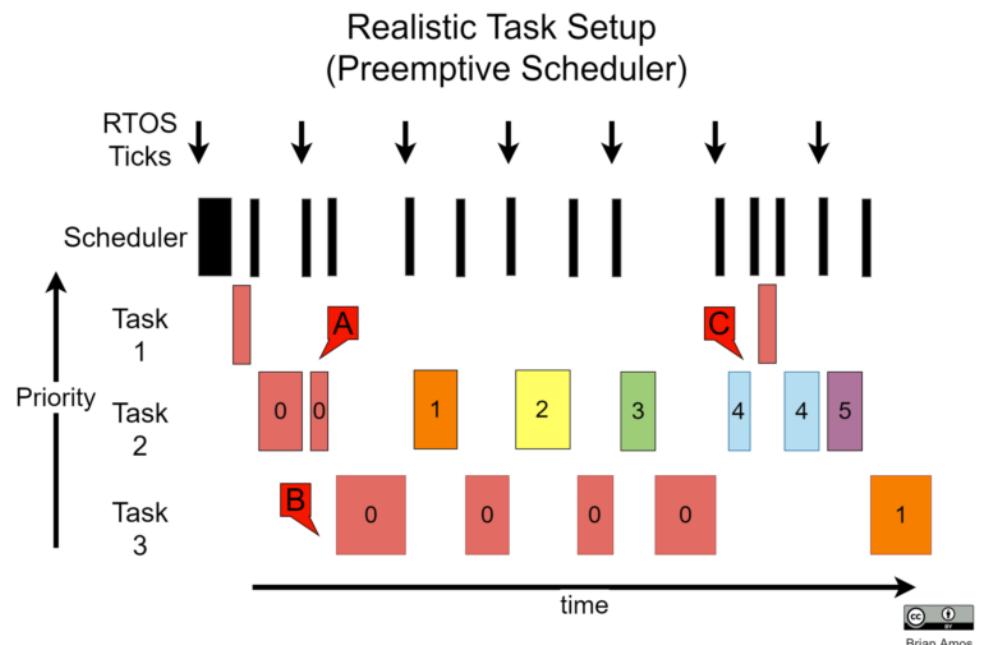
Real-Time Operating Systems

- Note:
- Point B
 - Task 2 done, nothing else *required* to run, task 3 can have processor time



Real-Time Operating Systems

- Note:
- Point C
 - Task 3 is done its first iteration, priority given to task 2 (iteration 4)
 - Task 1 needs to be serviced, so at tick scheduler switches context to task 1

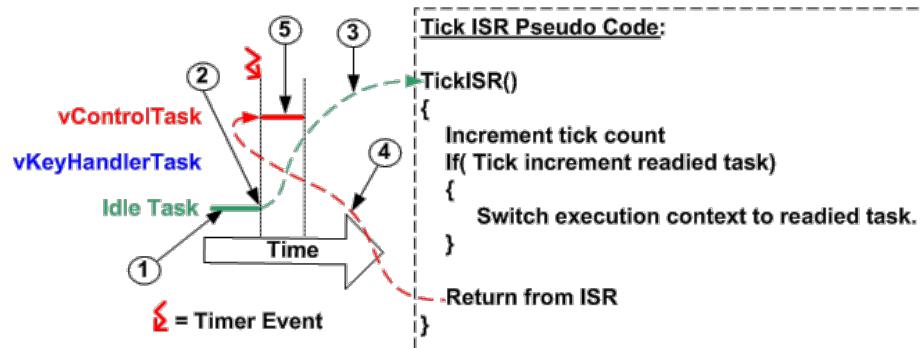


Real-Time Operating Systems

- The RTOS Tick
 - It is how the RT kernel measures time
 - It is a counter variable, incremented by an interrupt
 - This increment establishes the temporal accuracy (resolution) of the system
 - At every tick count, kernel checks: is it time to switch execution? What is the priority of the tasks seeking execution?

Real-Time Operating Systems

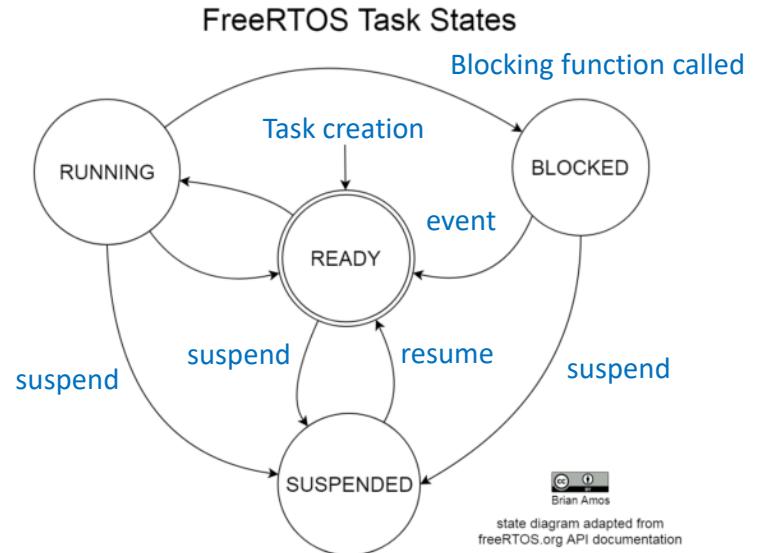
- The RTOS tick



- At 1, kernel idle, at 2 tick takes place, control transfers to ISR (returns at 4)
- The ISR makes function vControlTask ready to run
 - Since it is of higher priority than “idle”, vControlTask starts executing

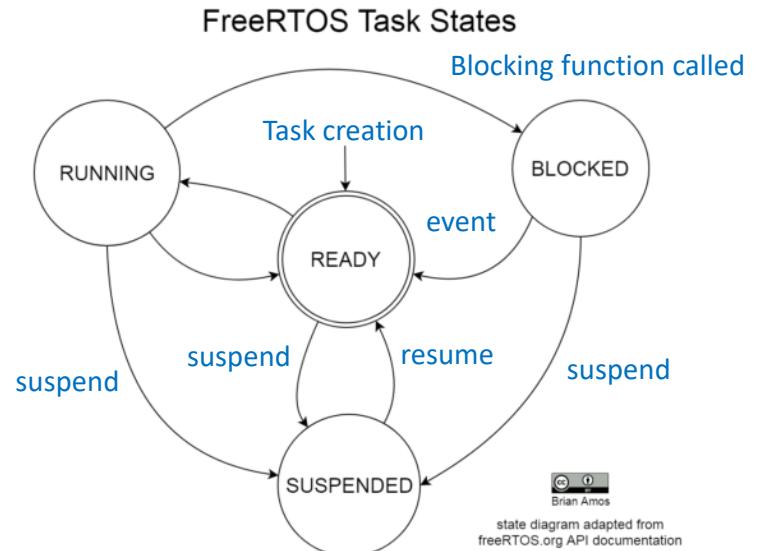
Real-Time Operating Systems

- Task States
- Tasks transition through the states either by scheduler calls or RTOS calls made through the code running



Real-Time Operating Systems

- Running
 - Task performing work
 - Suspend → switched out of context
- Ready – waiting for the scheduler to give processor context
- Blocked – waiting for something
 - It is time bound
- Suspend – ignored by scheduler



Real-Time Operating Systems

- Scheduler will be started and perform the initial tasks
 - This is architecture specific
 - At start, tick is set up, priorities are set up and first task is started
 - Will use kernel interrupt handlers, which are architecture-specific
 - One of these (a port service handler) will launch once
 - One will handle the context switching
 - One will handle the ticks

Real-Time Operating Systems

- Semaphores
 - Semaphores are “signal carriers”, who signal events, indicating something has happened.
 - They are sent between tasks or between task and interrupt
 - Do not carry data
 - When ISR is finished serving a peripheral, it “gives” a semaphore (data ready)
 - A task that needs to wait may use a semaphore to synchronize with other tasks
 - A resource may use a semaphore to indicate that the number of simultaneous users has been reached
 - If only one user allowed, use a “mutex” (mutual exclusion)

Real-Time Operating Systems

- Example: task related to serial port
 - Initializes hardware
 - Goes on infinite loop. Within the loop
 - Does what it needs to do (reads from sensors)
 - Inquires if it can “take” the semaphore for the serial port
 - If it can, it does TX/RX and returns the semaphore
 - If not, it waits for the scheduler to set something up (and tries again)

Recap

- Up to this point
 - We have looked at how computers interface with peripherals using particular protocols, mainly “on board”
 - We have looked at on-chip buses and the on-chip hardware components that come into play when data is transferred from peripheral to processor or peripheral to memory
 - Now we will look at wireless modules for communication/data transfer

Communication (Wireless)

- Communication between computer/embedded systems
 - Wireless offers great advantage
 - 2 or more points
 - Wide range
 - From few feet – low power FM, bluetooth
 - To thousands of kilometers – microwave link, GPS, satellite
 - Mobility, convenience, almost no set up time, reduced cost, access to remote/inhospitable areas
 - Where are they?
 - Remote controls, key fobs, GPS, RFID
-

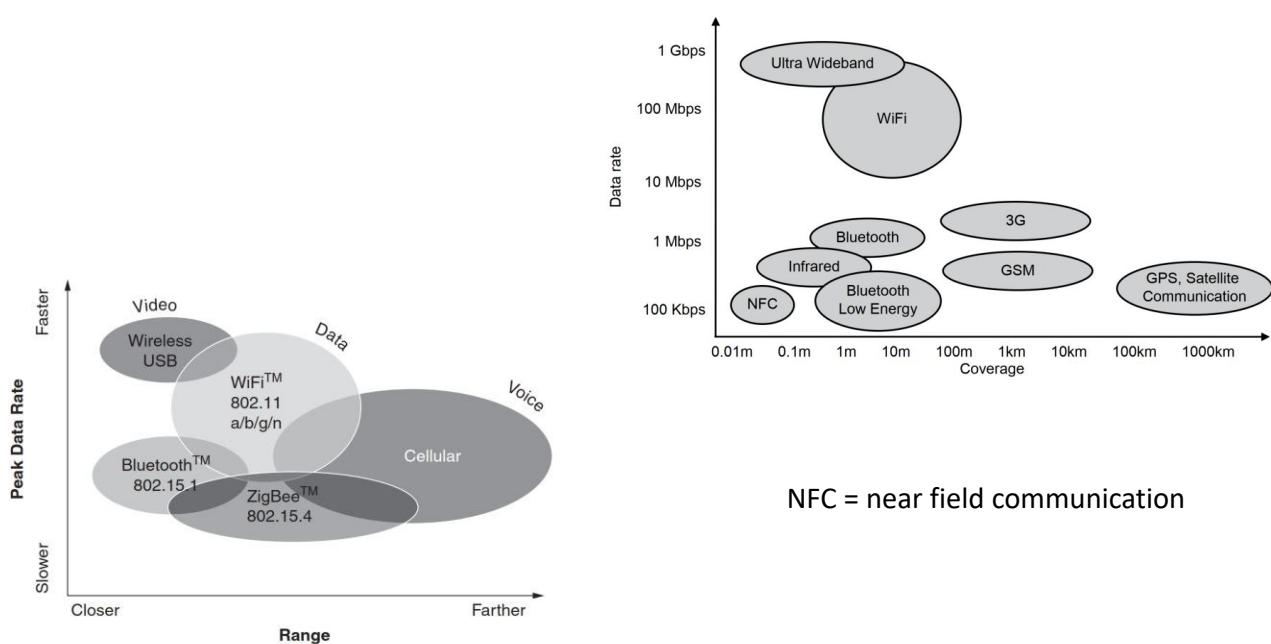
Communication (Wireless)

- Data speed
 - Presently: user perception as far as wireless data rate is similar to wired
 - Lower end: Bluetooth (BLE), zigbee, RFID
 - Higher end: wireless USB, WiFi, ultrawideband comm.
 - Terminology
 - WAN – wide area network; broad geographical area – LTE/4G
 - MAN – Metropolitan area network – microwave links
-

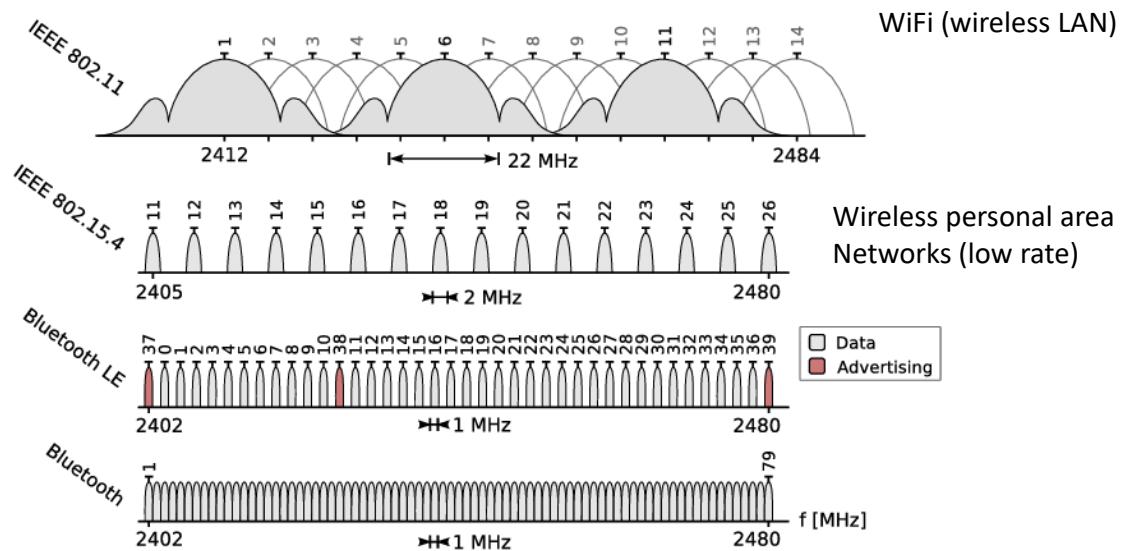
Communication (Wireless)

- Terminology
 - LAN – local area network – WiFi
 - PAN – personal area network – BLE, Zigbee, printers, keyboards, headphones
 - (BAN – body area network) – heart rate monitors, blood pressure monitors, wireless implants
- Concentrate on physical layer (PHY) – “signal meets the air”

Communication (Wireless)



Communication (Wireless)



Communication (Wireless)

- Will look at 3 popular technologies
 - Versatile, popular, straight-forward interface
 - Standards
 - A device from one manufacturer is guaranteed to “talk” to a device from another manufacturer using the same
 - Bluetooth Low Energy (BLE)
 - Zigbee (low power and mesh networking)
 - Touch on WiFi

Wireless

- Bluetooth
 - First: “classical Bluetooth”
 - Short range (~10m)
 - IEEE Standard 802.15.1
 - Uses ISM band (industrial, scientific, medical)
 - 79 sub-bands, FH-SS (frequency hop), 1MHz guard
 - FH-SS – search Hedy Lamarr for an interesting story
 - Packet based protocol
 - Each packet sent in one of 79 channel (hopping), at 1600 times/sec
-

Wireless

- Classical BT (cont'd)
 - PHY layer – two modulations
 - Basic rate (GFSK – Gaussian FSK) – 1Mbps
 - Enhanced data rate (DPSK – differential PSK)
 - 3Mbps (with 8 DPSK – 3bits/symbol)
 - 2Mbps (with 4 DPSK – 2 bits/symbol)
 - In practice (due to MAC layer overhead)
 - Basic: ~0.72Mbps
 - Enhanced ~1.45Mbps – 2.2Mbps
-

Wireless

- Bluetooth (cont'd)
 - Protocol: host – controller
 - With HCI (Host-controller interface)
 - Controller – resides in BT chipset
 - Host – resides in application
 - HCI ensures two vendors can manufacture compatible devices
 - Ex: Samsung – polar HRM
-

Wireless

- Bluetooth Low Energy (BLE)
 - Designed for low battery consumption
 - Sleep most of the time
 - Communicate with bursts, off otherwise
 - Version 5 has mesh network like Zigbee
 - Uses same band as classical, with FH-SS
 - Introduced “data advertising”
 - To discover and connect devices
 - (look at spectrum)
-

Wireless

- BLE Cont'd
 - Radio
 - Hops less frequently; can stay longer in channel
 - 40 channels @ 1MHz BW, 2MHz guard
 - Half as many as in classic BT
 - Uses GFSK
 - Max 1Mbps (1MHz channel BW)
 - Or Max 2Mbps (can also use 2MHz BW channel)
 - Slower rate → more range
 - Typical operation 2m-5m (possible up to 30m)
 - Less power → less battery → less range
-

Wireless

- BLE (cont'd)
 - Peripheral device
 - Typically small, low power, resource constrained
 - Think sensor / headphone
 - Central device
 - Not resource constrained, power not as much of a concern
 - Notebook / Mobile
 - Peripherals broadcast or are directly connected
 - “broadcast” send data to any device in range
-

Wireless

- Peripheral broadcast
 - Peripherals “advertise” to observers
 - Any in range
 - Send packet describing device + custom information
 - Central devices can connect to many peripheral
 - BLE4.1 – peripheral can connect to many central
-

Wireless

- BLE profiles (connection)
 - Generic Access Profile (GAP)
 - Defines role of device, guides advertisement and discovery
 - After connected, moves to GATT
 - Generic Attribute Profile
 - Defines how data is to be transferred
 - When connection is established, client gets a list of **services** offered by the server
-

Wireless

- BLE (cont'd)
 - Universally Unique Identifier
 - Contains further information on services and characteristic of device
 - Ex: light bulb service

	UUID	Properties	Values	Comments
Light switch	FF11	R/W	1	1 on, 0 off
Dimmer	FF12	R/W	0x7F	0x00 – 0x7F
Power Info	FF16	R	340	Watt/hours

Communication (Wireless)

- WiFi
 - 802.11 family of standards
 - LAN internet access
 - Designed to work with Ethernet 2.4GHz – 5GHz
 - Packeted data
 - Lately OFDM – frequency division (subcarriers)
 - Overlap but don't interfere (orthogonal)
 - Various modulations (up to 1024-QAM)
 - To a max of Gbps on a wide channel
 - 802.11ax (WiFi6) – on 6GHz unlicensed band

ECE342 – Computer Hardware

Lecture 33

Bruno Korst, P.Eng.

Agenda

- Communications - Wireless

Wireless

- Bluetooth
 - First: “classical Bluetooth”
 - Short range (~10m)
 - IEEE Standard 802.15.1
 - Uses ISM band (industrial, scientific, medical)
 - 79 sub-bands, FH-SS (frequency hop), 1MHz guard
 - FH-SS – search Hedy Lamarr for an interesting story
 - Packet based protocol
 - Each packet sent in one of 79 channel (hopping), at 1600 times/sec
-

Wireless

- Classical BT (cont'd)
 - PHY layer – two modulations
 - Basic rate (GFSK – Gaussian FSK) – 1Mbps
 - Enhanced data rate (DPSK – differential PSK)
 - 3Mbps (with 8 DPSK – 3bits/symbol)
 - 2Mbps (with 4 DPSK – 2 bits/symbol)
 - In practice (due to MAC layer overhead)
 - Basic: ~0.72Mbps
 - Enhanced ~1.45Mbps – 2.2Mbps
-

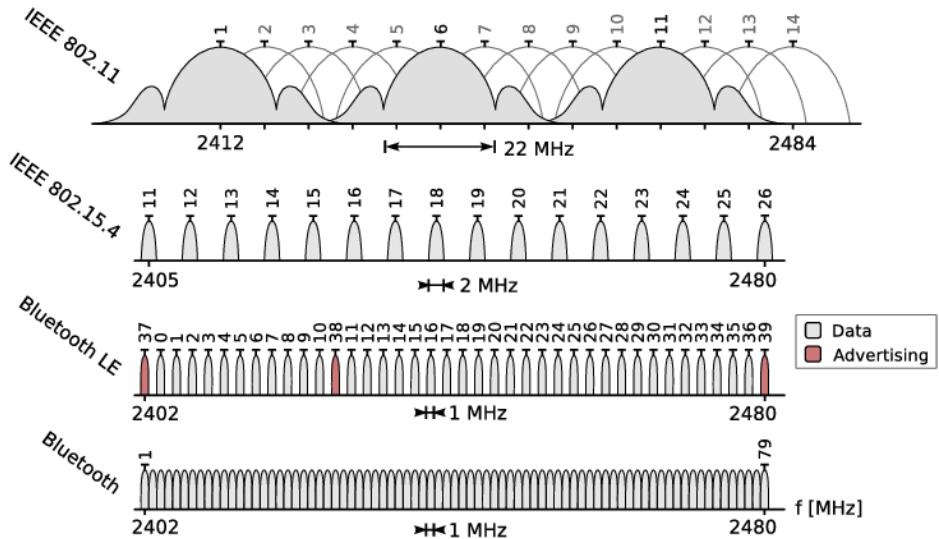
Wireless

- Bluetooth (cont'd)
 - Protocol: host – controller
 - With HCI ([Host-controller interface](#))
 - Controller – resides in BT chipset
 - Host – resides in application
 - [HCI ensures two vendors can manufacture compatible devices](#)
 - Ex: Samsung – polar HRM
-

Wireless

- Bluetooth [Low Energy](#) (BLE)
 - Designed for low battery consumption
 - Sleep most of the time
 - Communicate with bursts, off otherwise
 - Version 5 has mesh network (like Zigbee, as we will see)
 - Uses same band as classical, with Freq. Hopping – Spread Spectrum
 - Introduced “data advertising”
 - To discover and connect devices
-

Wireless



Wireless

- BLE Cont'd
 - Radio
 - Hops less frequently; can stay longer in channel
 - 40 channels @ 1MHz BW, 2MHz guard
 - Half as many as in classic BT
 - Uses GFSK
 - Max 1Mbps (1MHz channel BW)
 - Or Max 2Mbps (can also use 2MHz BW channel)
 - Slower rate → more range
 - Typical operation 2m-5m (possible up to 30m)
 - Less power → less battery → less range

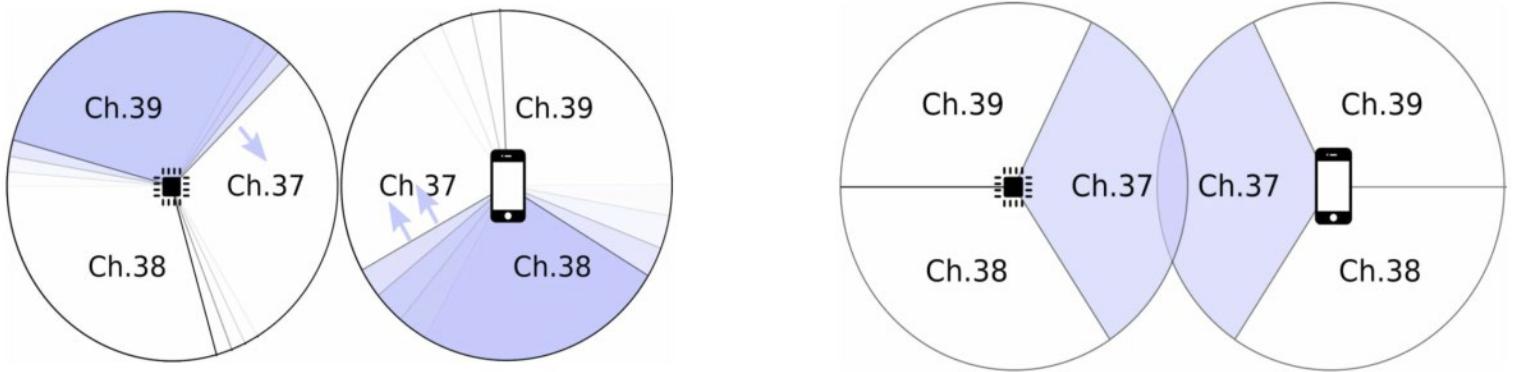
Wireless

- BLE (cont'd)
 - Peripheral device
 - Typically small, **low power**, resource constrained
 - Think sensor / headphone
 - Central device
 - Not resource constrained, power not as much of a concern
 - Notebook / Mobile
 - Peripherals broadcast or are directly connected
 - “broadcast” send data to any device in range
-

Wireless

- Peripheral broadcast
 - Peripherals “advertise” to observers
 - Any in range
 - Send packet describing device + custom information
 - Central devices can connect to many peripheral
 - BLE4.1 – peripheral can connect to many central
-

Wireless

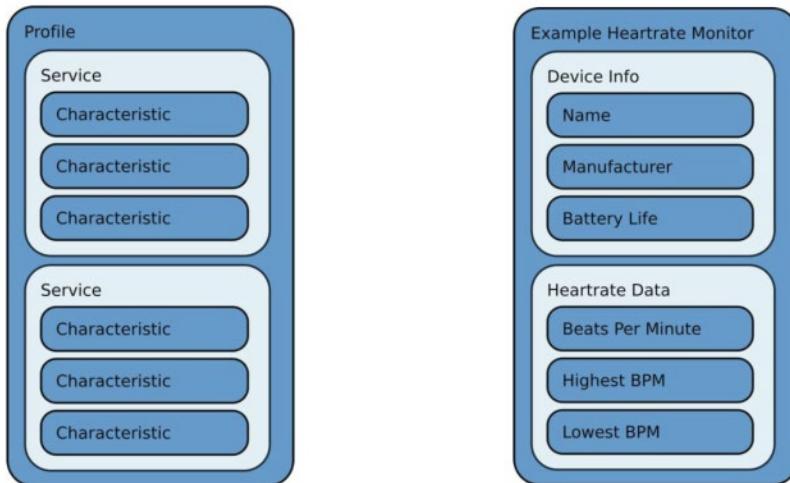


Wireless

- BLE profiles (connection)
 - Generic Access Profile (GAP)
 - Defines **role of device**, guides advertisement and discovery
 - After connected, moves to GATT
 - Generic Attribute Profile (GATT)
 - Defines how **data** is to be **transferred**
 - When connection is established, client gets a list of **services** offered by the server

Wireless

- BLE GATT structure



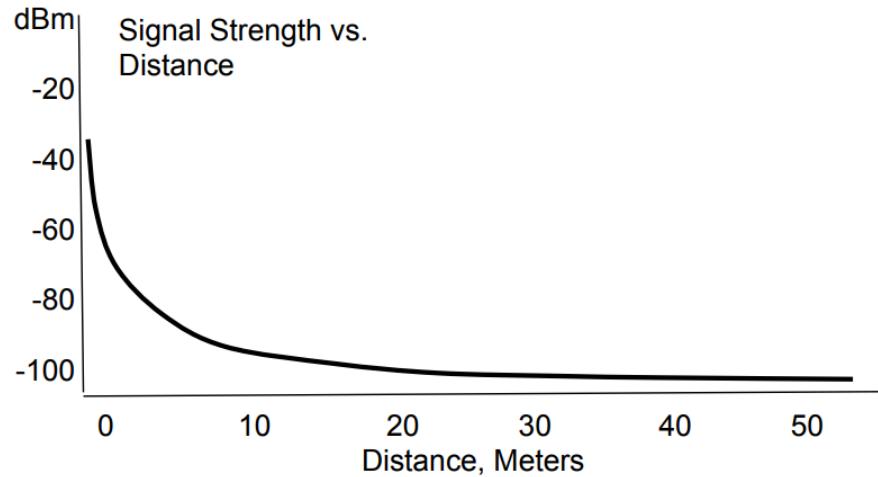
Wireless

- BLE (cont'd)
 - Universally Unique Identifier
 - Contains further information on services and characteristic of device
 - Ex: light bulb service

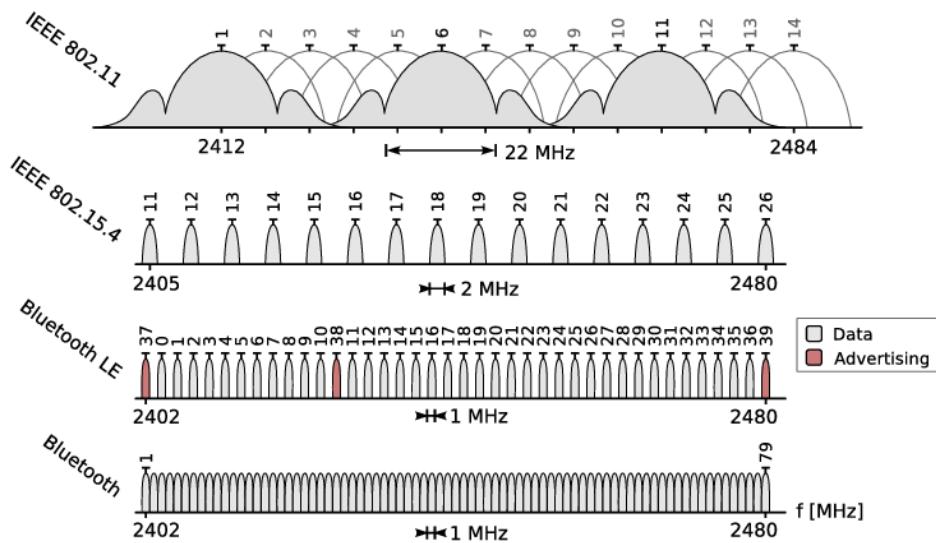
	UUID	Properties	Values	Comments
Light switch	FF11	R/W	1	1 on, 0 off
Dimmer	FF12	R/W	0x7F	0x00 – 0x7F
Power Info	FF16	R	340	Watt/hours

Wireless

- Signal strength as a function of distance



Wireless



Wireless

- Zigbee
 - Adds to the IEEE 802.15.4 standard
 - “Standard for Low-Rate Wireless Networks”
 - Define PHY and MAC (medium access control) specifications for **low data rate wireless** connectivity with fixed, portable and moving devices with no battery or very limited battery consumption.
 - Focuses on
 - Power management
 - Addressing
 - Error correction
 - Messaging format
-

Wireless

- Zigbee (cont'd)
 - Routing
 - One radio is able to **pass messages THROUGH** to other radios
 - Ad-hoc creation
 - Networks/subnetworks are created “**as needed**”
 - Self-healing
 - Network is **automatically reconfigured** if one or more nodes are down
-

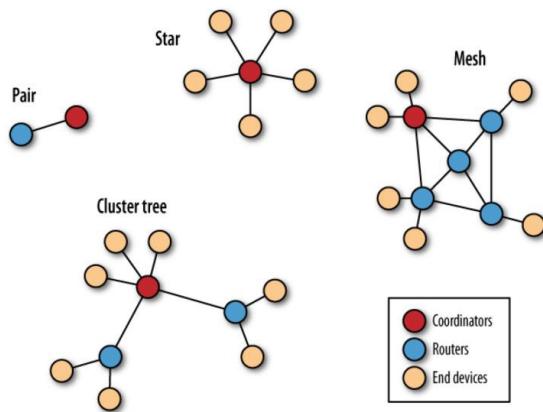
Wireless

- Zigbee (cont'd)
 - Every zigbee network has
 - ONE coordinator
 - AT LEAST ONE router or one end device
 - Topologies
 - Pair – only two
 - Star – one coordinator, and end devices that do not communicate directly with each other
-

Wireless

- Zigbee topologies (cont'd)
 - Mesh
 - Coordinator + router nodes
 - Coordinator manages network AND routers
 - Can route around potential problems/barriers
 - Can go for any distance provided there are enough radios
 - Cluster Tree
 - Similar to mesh, but routers don't communicate directly
-

Wireless



Coordinator

- Only one
- Forms the network
- Distributes addresses
- Maintains/manages other network functions

Router

- It's a full node
- send/receive/route information
- Can join existing networks

End Device

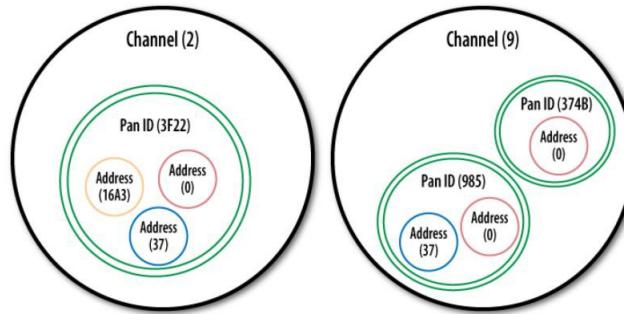
- sends/receives data
- Can join network
- Always needs a router or coordinator as "parent" device

Wireless

- Zigbee addressing
 - All radios have a permanent unique address (64 bit serial number)
 - Short 16 bit address is dynamically assigned to each radio by the coordinator when network is being set up
 - 16 bit address is unique WITHIN the network
 - Each zigbee network created has a 16 bit PAN ID
 - 64 bit extended address used in conflict resolution

Wireless

- Zigbee addressing (cont'd)
 - Radios have to tune to the same channels (as assigned) for communication to happen
 - In addition to the unique addresses



Wireless

- Zigbee radios
 - Use 2.4GHz unlicensed ISM band
 - Industrial, scientific and medical
 - Coordinator picks a channel (out of 12), all nodes will use that channel
 - Off set QPSK and DS-SS
 - Encryption and authentication
 - Advanced Encryption Standard (AES) 128
 - Use collision avoidance
 - Listen first, transmit if quiet

Wireless

- WiFi
 - 802.11 family of standards
 - LAN internet access
 - Designed to work with Ethernet 2.4GHz – 5GHz
 - Packeted data
 - Lately OFDM – frequency division (subcarriers)
 - Overlap but don't interfere (orthogonal)
 - Various modulations (up to 1024-QAM)
 - To a max of Gbps on a wide channel
 - 802.11ax (WiFi6) – on 6GHz unlicensed band
 - Look up module ESP8266
-