Fundamentals of Embedded and Real Time Systems

MODULE 01

TAMER AWAD

People First

What is your professional background?

Where are you joining from?

What brings you to this class?

A random fact about yourself

EMBEDDED & REAL-TIME SYSTEMS PROGRAMMING CERTIFICATE

EMBSYS 100:

FUNDAMENTALS OF EMBEDDED & REAL-TIME SYSTEMS

- Instructor: ***Tamer Awad***
- Intro to computer architecture and hardware concepts
- Number system and Boolean algebra
- C programming
- Intro to the ARM architecture and ARM assembly essentials
- Embedded development tools and evaluation kit.

EMBSYS 105:

PROGRAMMING WITH EMBEDDED & REAL-TIME OPERATING SYSTEMS

- Instructor: ***Nick Strathy***
- RTOS Services
- Port RTOS to a microcontroller
- How to program a multitasking system
- Device driver frameworks
- Use of display, audio and touch screen modules

EMBSYS 110:

DESIGN & OPTIMIZATION OF EMBEDDED & REAL-TIME SYSTEMS

- Instructor: ***Lawrence Lo***
- C++ for embedded systems and object-oriented design
- Hierarchical State Machines (HSM) and event driven design.
- State-chart design, patterns, and framework
- Use of connectivity and sensor modules

About this class

- **Time:**
  - Monday evenings, 6-9pm (Pacific Time), Weekly from October 7th to December 16th.
  - Online only
  - NO CLASS ON NOVEMBER 11TH.
- ~~**In Person:**~~
  - ~~2445 140th Avenue N.E., Ste. B100, Bellevue, WA 98005 1879~~
- **Webcast Zoom Link:**
  - https://washington.zoom.us/my/embsys
- **Canvas link:**
  - https://canvas.uw.edu/courses/1325909

Grading and Attendance

This is a **pass/fail** course that is dependent on performance and participation.

~~Classroom students must attend at least 80% of sessions, in-person, to be eligible to pass the course.~~

Online students must **participate online**, ~~or in person~~, for **at least 60% of sessions** to be eligible to pass the course.

All students must **complete a minimum of 80% of total assignments** to be eligible to pass the course. Individual assignments will have prescribed weights and due dates.

Course materials and technologies

Textbooks:

An Embedded Software Primer, by David E. Simon

The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors, by Joseph Yiu

Hardware: Custom development kit.

Software: Students will be expected to install software on their personal computer: IAR, Teraterm, ST Link USB driver, …etc.

In addition, there will be reference materials on selected topics, including datasheets and user manuals

Class Format

LECTURES

DISCUSSIONS

DEMOS

LABS

BREAKS

Course Outline
(subject to change)

- Introduction to Embedded Systems
- Number systems
- Basic computer math, arithmetic and logic
- Embedded development environment
- C Programming Language
- Embedded evaluation board
- Realtime Operating System (RTOS) and its services
- Embedded software architecture
- Introduction to ARM architecture
- Assembly language
- Exceptions, interrupts and interrupt handling

Canvas course site

HTTPS://CANVAS.UW.EDU/COURSES/1325909

Module 01

- Introduction to embedded systems
- Basic hardware architecture
- Number Systems
- Basic computer math, arithmetic and logic
- Embedded development environment overview
- Software version control using Git
- LAB 01

Introduction to Embedded Systems

- Embedded Systems
- Embedded system vs general-purpose computer
- Embedded application example
- Embedded software development challenges

Embedded Systems

- An **embedded system** is a dedicated computer system designed for a specific functionality.

- It is **embedded** as a part of a product that includes hardware, such as electrical and mechanical components.

- *According to Wikipedia: https://en.wikipedia.org/wiki/Embedded_system*
  - An **embedded system** is a controller with a **dedicated** function within a larger mechanical or electrical system, often with real-time computing constraints
  - Ninety-eight percent of all microprocessors manufactured are used in embedded systems.
  - Modern embedded systems are often based on microcontrollers (i.e. microprocessors with integrated memory and peripheral interfaces).

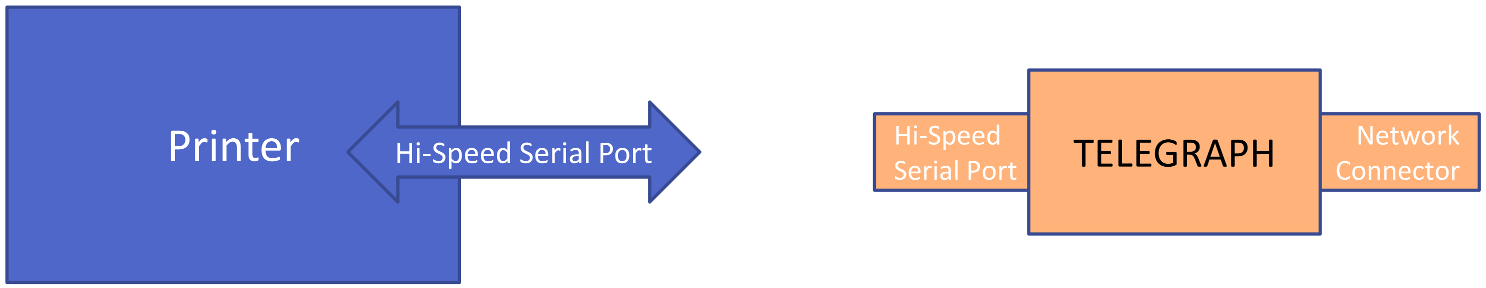
Examples of Embedded Systems

- Digital watches
- MP3 players
- Traffic light controllers
- Hybrid vehicles
- Medical equipment
- Avionic systems
- Programmable Logic Controllers (industrial digital computers adapted for the control of manufacturing processes, assembly lines, robotic devices, automobile manufacturing…etc.)
- Internet of Things devices
- David E. Simon "More products have microprocessors **embedded** in them to make them **smart**."

Embedded systems vs General Purpose Computers

Embedded Systems	General Purpose Computers
Single/Specific functionality	Multiple functionality
Time sensitive	Relaxed-timing
Limited hardware resources	Large hardware resources
No human interaction/intervention	Human accessible

Telegraph Example



Telegraph allows you to connect a printer that has only a high-speed serial port to a network.

Must receive data from the network and copy it onto the serial port.

Development Challenges

Throughput
- Handling a lot of data in a short time.

Response
- Reacting to events "quickly"

Testability
- Difficulties in setting up equipment to test embedded software

Debugability
- No screen and no keyboard

Reliability
- No human intervention once deployed

Memory Space
- Limited memory

Program Installation and upgrade
- Special tools to install and upgrade the software.

Power consumption
- Devices could be running on battery and software needs to help conserve power.

Security:
- Who can access the printer?

Cost
- Reducing the cost is usually a high priority for these systems.

Expect that your software operates on hardware that is barely adequate for the job.

So why then do it?

- The challenge of building a reliable software solution on such constrained devices is always a **rewarding** experience from an engineering perspective.
- Directly working with the **hardware**.
- Develop in depth understanding of the inner workings of a **computer**.
- The global **market** for embedded systems (hence embedded engineers) is **growing** at a very rapid pace with the revolution of Internet of Things and expected to continue to grow for the foreseeable future.

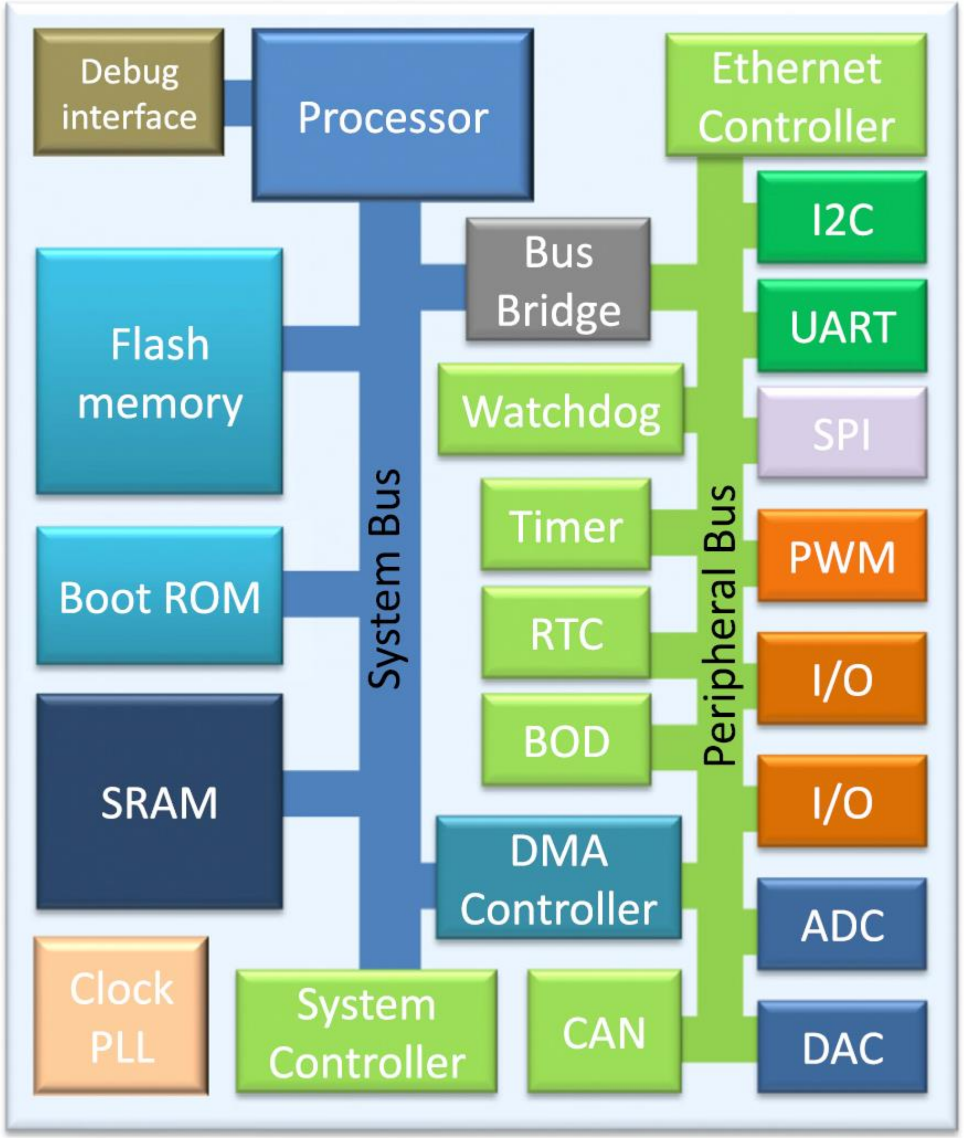
Assignment 1.a

Using the telegraph example, and the challenges listed in the book and the module as reference, describe another device that you would like to discuss. Describe how you think its embedded system works, and what design challenges it presents.

BREAK 1

Basic Hardware Architecture

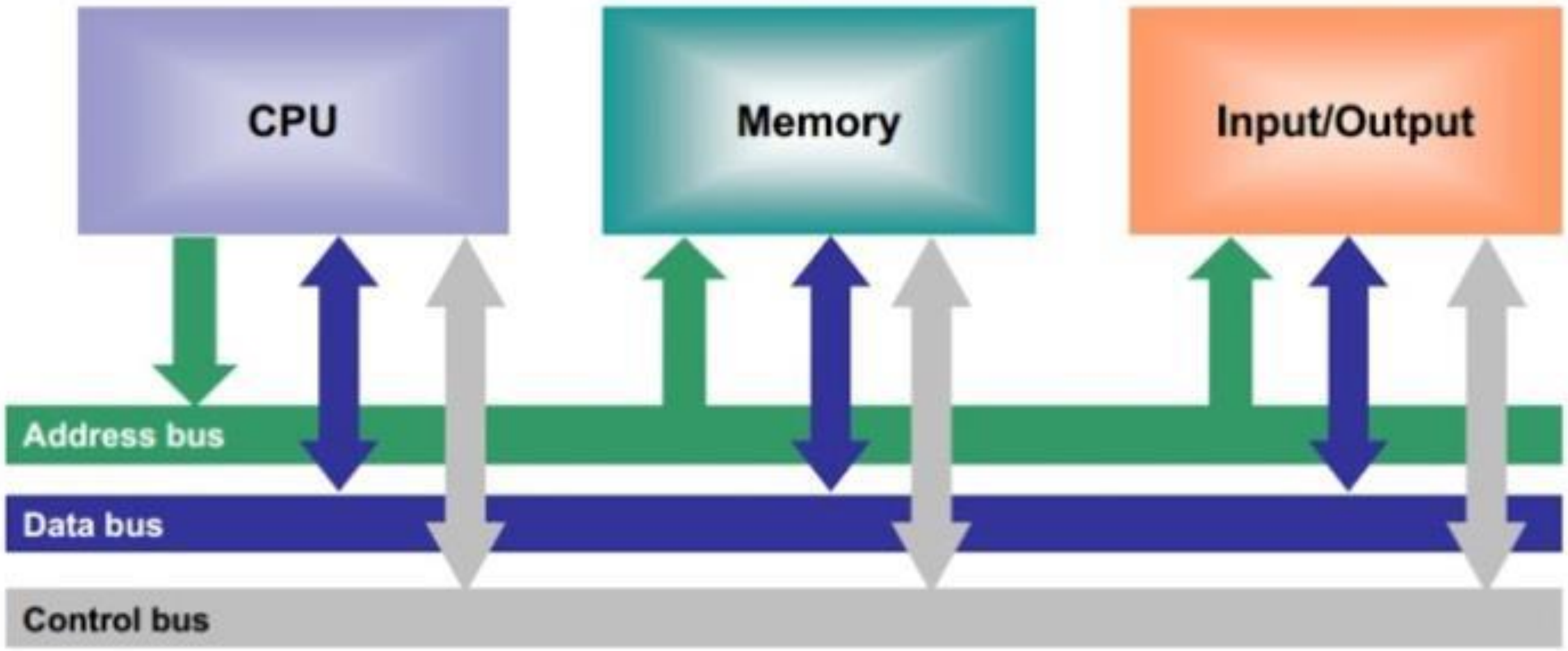
- Microcontroller vs. Microprocessor
- Typical embedded hardware
- Microprocessor
- Memory basics, types, size, address lines, Memory Map
- Bus
- Clock
- Hardware Registers
- Peripherals



Microcontroller vs. Microprocessor

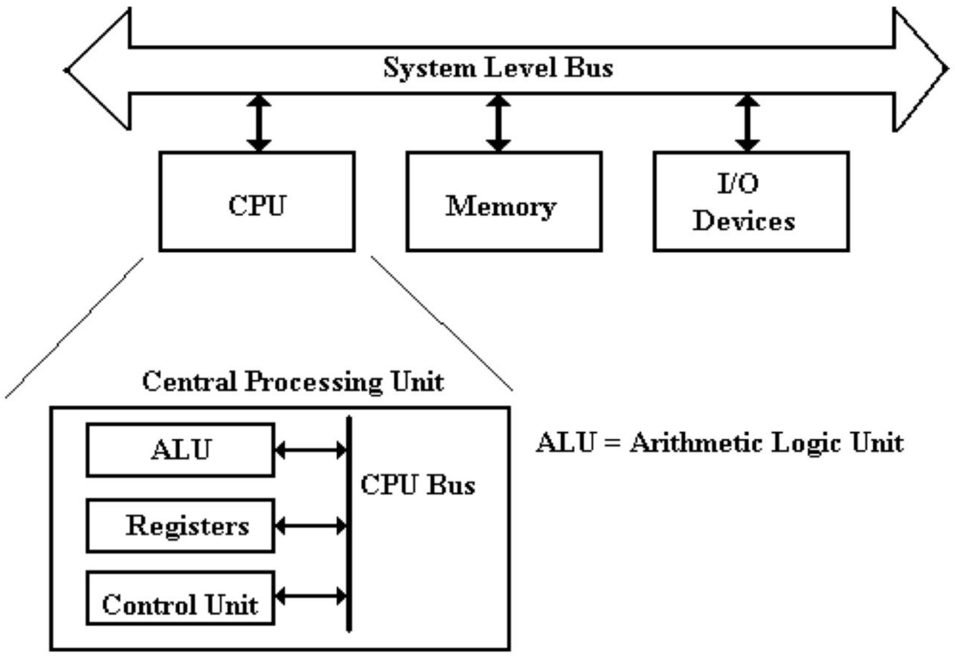
SOURCE: *JOSEPH YIU “THE DEFINITIVE GUIDE TO ARM CORTEX M3 & M4”*

A Typical Embedded Hardware



Source: http://collagenrestores.com /architecture block diagram image/the building blocks of embedded systems its all about embedded/

A Typical Embedded Hardware



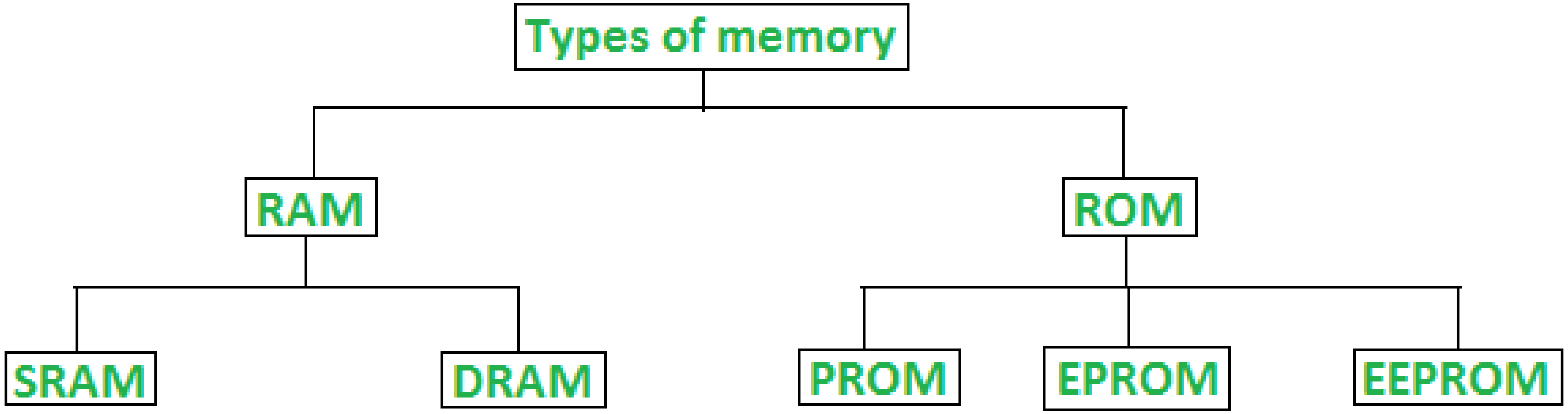
- CPU:
  - Control Logic
  - Data operations
  - Arithmetic (add, subtract, etc)
  - Logical (and, or, xor, etc)
- Memory:
  - Stores instructions (programs instructions)
  - Stores data (programs variables)
- Busses:
  - Connect the processor to memory
  - Data Bus
  - Address Bus
- I/O:
  - Pins that connect to the “outside” world

Memory

- **What is memory for?**
  - Stores **data**
  - Stores **instructions**

- **How does processor communicate with the memory?**
  - Address bus
  - Data bus
  - Control signals

Memory Types



Classification of computer memory

Memory Types - RAM

Random Access Memory (RAM)

- *Read write memory*/the *main memory*/the *primary memory*
- The code and data that the CPU requires during execution of a program are stored in this memory
- A volatile memory as the data is lost when the power is turned off

RAM is further classified into two types-

- *SRAM (Static Random-Access Memory)*
- *DRAM (Dynamic Random-Access Memory)*
  - *Requires RAM refresh (reading data periodically)*
  - *Cheaper than SRAM*

Memory Types - ROM

Read Only Memory (ROM)

- Stores information essential to operate the system, like the program to boot the computer
- It is not volatile
- Used in embedded systems or where the programming needs no change
- ROM is further classified into 3 types
  - PROM (Programmable read-only memory)
    - Can only programmed once
  - EPROM (Erasable Programmable read only memory)
    - Can erase and rewrite the program
  - EEPROM (Electrically erasable programmable read only memory)
    - Similar to PROM but can be erased and rewritten via special input/output signals.
    - Typically used to store configuration information (network address, user name…etc.)

Memory Types – Flash Memory

- A form of EEPROM (Electrically erasable programmable read only Memory)
- Faster than EEPROM
- Larger in size
- Usage ranging from Flash memory USB sticks to Compact Flash cards used for cameras
- Stores the program in embedded devices.

Memory Types – RAM vs ROM

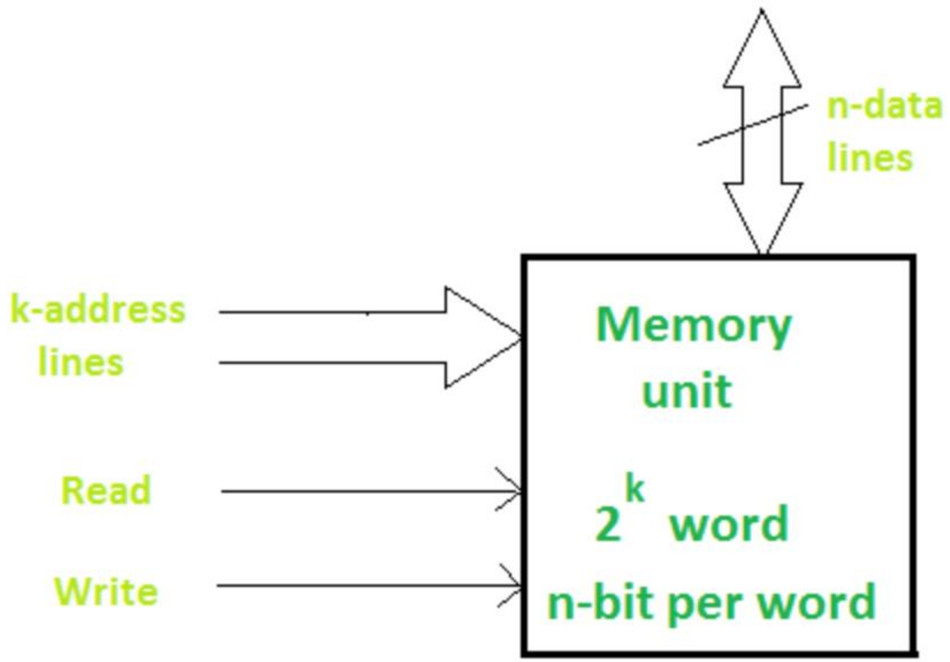
RAM	ROM
1. Temporary Storage.	1. Permanent storage.
2. Store data in MBs.	2. Store data in GBs.
3. Volatile.	3. Non-volatile.
4.Used in normal operations.	4. Used for startup process of computer.
5. Writing data is faster.	5. Writing data is slower.

Difference between RAM and ROM

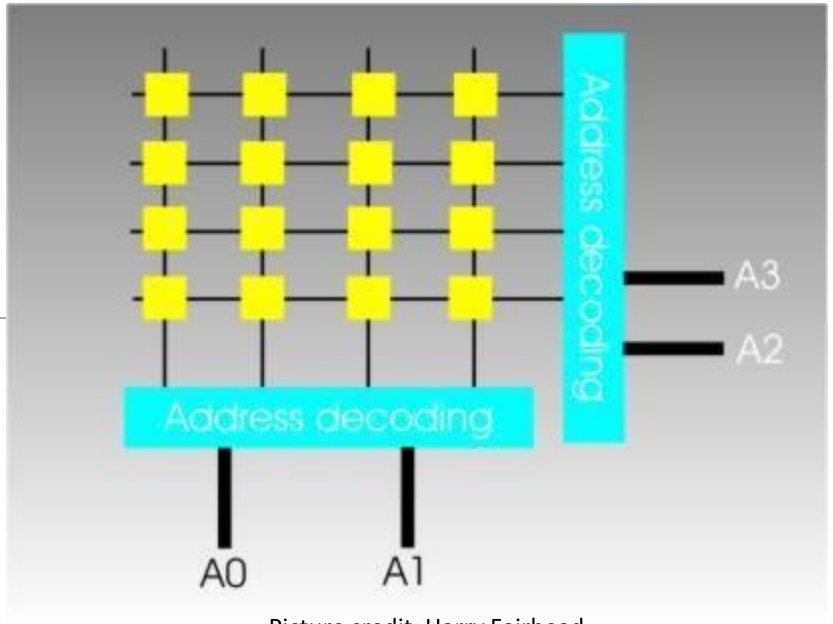
Memory Table

Memory is a table of data (2-d Matrix)

- Table width determined by the number of data lines (8, 16, 32, 64-bit)
- Address provides table index
- Memory size depends on the number of address lines



Memory Address Lines



Picture credit: Harry Fairhead

- Address lines are converted to row/column selects
- The number of bits required to address a memory is the number of lines required to access that memory
- With n number of lines, we can represent 2^n number of addresses
- A system with a 32-bit address bus can address 2^32 (4,294,967,296) memory locations.

Example - Memory Address Lines

Question: If you have 3 address lines, how many memory values can you access?

Example - Memory Address Lines

Answer: the values which can be formed using 3 bits are 2^3:

A2 A1 A0

000

001

010

011

100

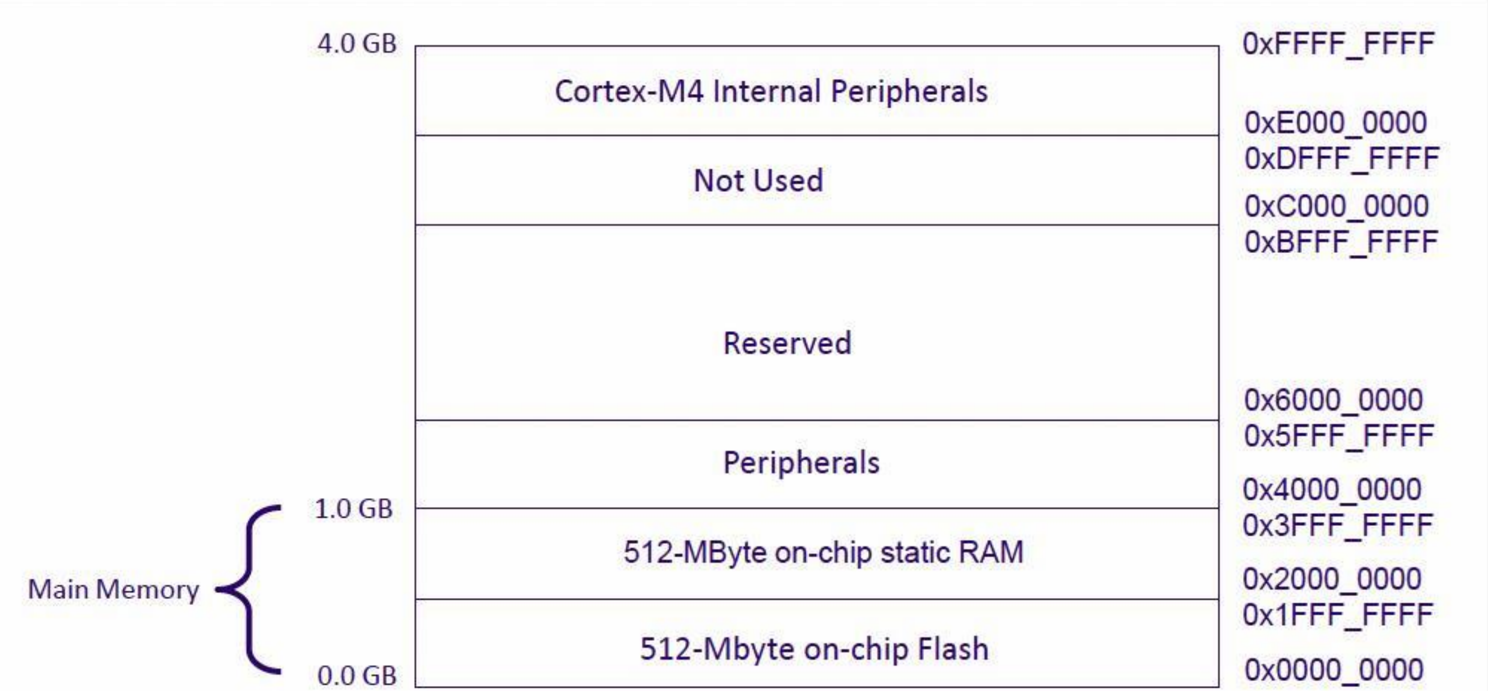
101

110

111

Memory Map

A Memory Map describes how memory is structured in the system, so the CPU can access the right parts of the system



Bus

- What’s in a bus
  - Address lines – microprocessor to memory chips (RAM, ROM)
  - Data lines – (same as address)
  - Read line – output enable signal
  - Write line – input enable signal
- Protocol - a set of rules for exchanging data
- Bandwidth of the bus depends on
  - width of the wire
  - clock speed

Clock

Why need a clock?

- Provides a rising and falling edge to the circuit, so that different parts can detect what is a 1 (high) and what is a 0 (low)
- Two types
  - Oscillators: 4pins, generates clock signal by itself
  - Crystals: 2pins, more accurate and temp resistant; however need to build a circuit around it to get clock signal out.
- A wide range of frequencies

Hardware Registers

- Hardware Registers are used extensively throughout computer systems
- The primary interface between hardware and software
- When you’re doing low level programming, registers are one of the first things you ask for documentation on.

Peripherals

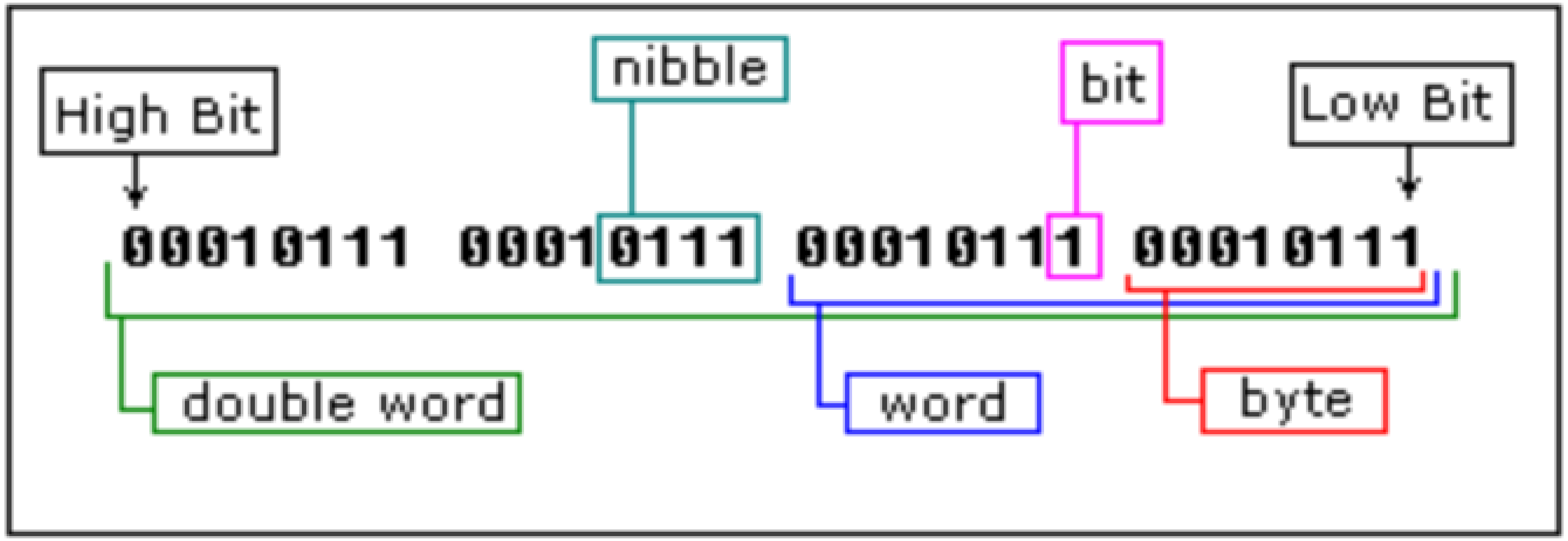
- **What are peripherals?**
  Hardware devices that reside outside of the processor chip & communicate with it by interrupts and I/O or memory-mapped registers.

- **What is the basic interface between processor & peripherals?**
  A set of controls (pins) and status registers in memory space, or I/O space

Number Systems

- Bit, Byte, Word, Double Word
- Binary, Octal, Decimal, and Hexadecimal Numbers
- Signed, unsigned
- Overflow
- Byte orders

Bit, Byte, Nibble, Word, Double Word



Number Systems

Number System	Base / Radix (# of digits)
Decimal	Base 10 (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
Binary	Base 2 (0, 1)
Octal	Base 8 (0, 1, 2, 3, 4, 5, 6, 7)
Hexadecimal	Base 16 (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F)

- **Hexadecimal** numerals are widely **used** by **computer system** designers and programmers, as they provide a human-friendly representation of binary-coded values.
- Each **hexadecimal** digit represents four binary digits, also known as a nibble, which is half a byte. (https://en.wikipedia.org/wiki/Hexadecimal)

Bits & Bytes

Binary	Hex	Decimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

- 1 Byte = 8 bits
- Binary = 0000 0000 – 1111 1111
- Decimal = 0 – 255
- Hexadecimal = 0x00 – 0xFF

Binary to Hex

What is binary 10101011 in Hex?

1. Split into nibbles (4 bits) --> 1010 1011
2. Convert each nibble to its equivalent hex digit --> AB
3. Denote the value with “0x” --> 0xAB

Hint: How do you remember the nibbles to hex conversion?

- Sum up the position value of all the digits that are set to 1 (demo)

Binary addition & subtraction

A)

```
  0001 1010
+ 0000 1011
-----------------
  0010 0101
```

B)

```
  0001 1010
- 0000 1100
-----------------
  0000 1110
```

Binary multiplication by 2 & division by 2

A)

0001 1010

* 0000 0010

0011 0100

B)

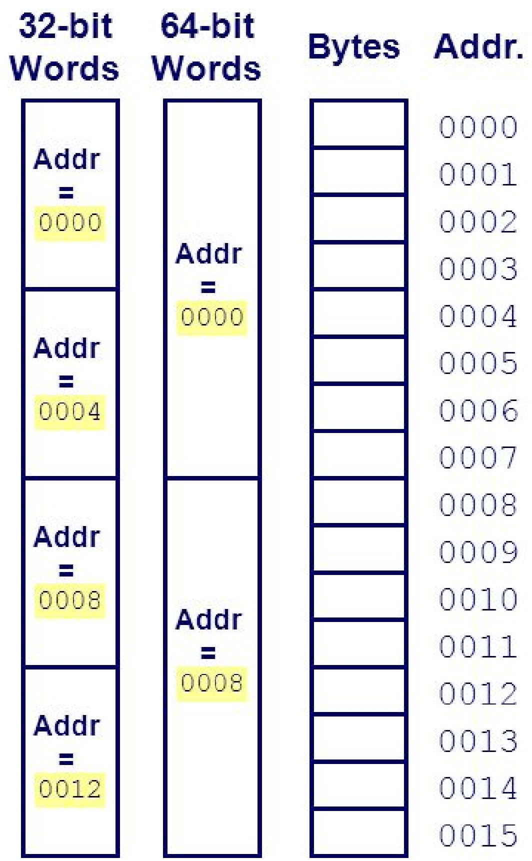
0001 1010

/ 0000 0010

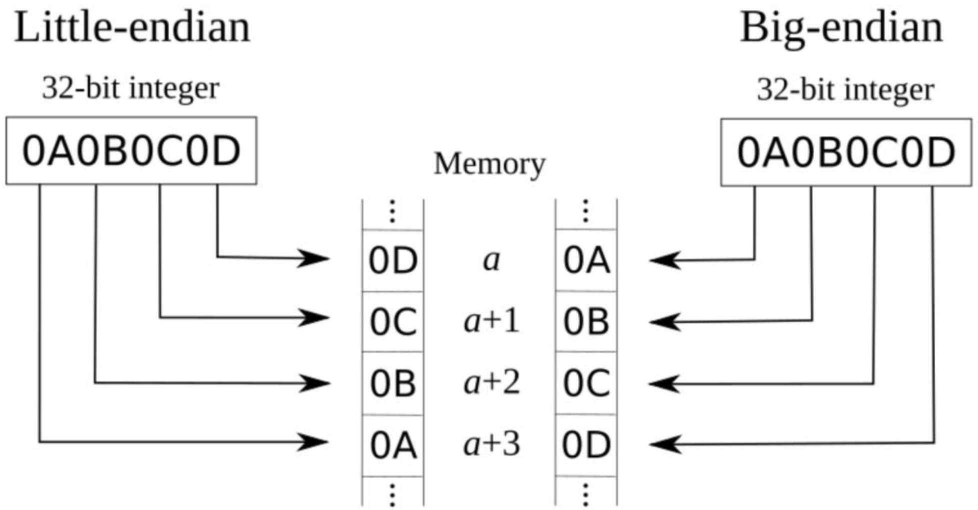
0000 1101

Machine “word”

- A **word** is a fixed-sized piece of data handled as a unit by the instruction set or the hardware of the processor.
- It is the natural unit of data used by a particular processor design
- An **address** specifies a word location
- Modern processors **word size**:
  - 8, 16, 24, 32, or 64 bits



Byte ordering



- **Big-endian** is an order in which the "**big** end" (most significant value in the sequence) is stored first (at the lowest storage address).
- **Little-endian** is an order in which the "**little** end" (least significant value in the sequence) is stored first.

- MSB -Most significant byte
- LSB -Least significant byte
- **Potential portability bugs**

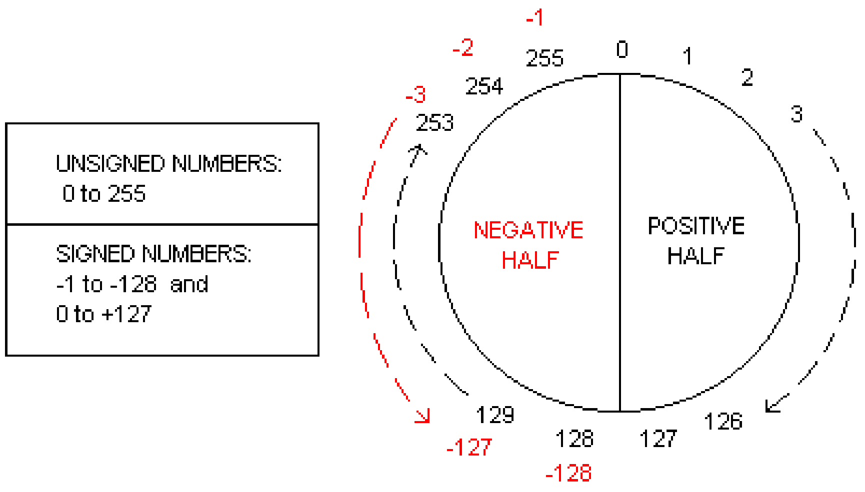
Unsigned type

Unsigned: positive

for example, 8-bit range:

- Binary: 0000 0000 -1111 1111
- Decimal: 0 -255
- Hex: 00 –FF
- Question: what is the range of 8-bit signed number?
  - Binary:
  - Decimal:
  - Hex:

Signed



8-BIT NUMBER SYSTEM

- Signed: positive & negative
  - Most significant bit becomes the “sign” bit
  - 1 = negative number
  - 0 = positive number
- The range is -128 to 127 (instead of 0 to 255)
  - 128 = 1000 0000
  - 127 = 0111 1111
- Using two's complement:
  - Positive integers are represented the same way as unsigned integers.
  - For negative integers:
    - Step1: produce positive value in binary
    - Step2: invert all the bits
    - Step3: add 1

Signed (example)

- **Represent “-12” as a signed byte**
- Using two's complement:
  - Produce the positive value of the integer in binary
  - Invert all the bits
  - Add one to the result
- Binary: 0000 1100
- Invert: 1111 0011
- Add 1: **1111 0100**

- **Finding the scalar (positive) value of a negative binary?**
  - Step1: subtract 1
  - Step2: invert all the bits

Assignment 1.b

Why use two’s complement to represent negative numbers?

Overflow

Occurs when a value is too large for the data type

1111 1111

+ 0000 0001

10000 0000

Binary addition of signed numbers can change sign

Signed: 0111 1111 + 0000 0001 = 1000 0000 = -128

Because 128 exceeds the range of a signed byte, an overflow changes the sign and give an incorrect result

Programmer must always be aware of the range of variables.

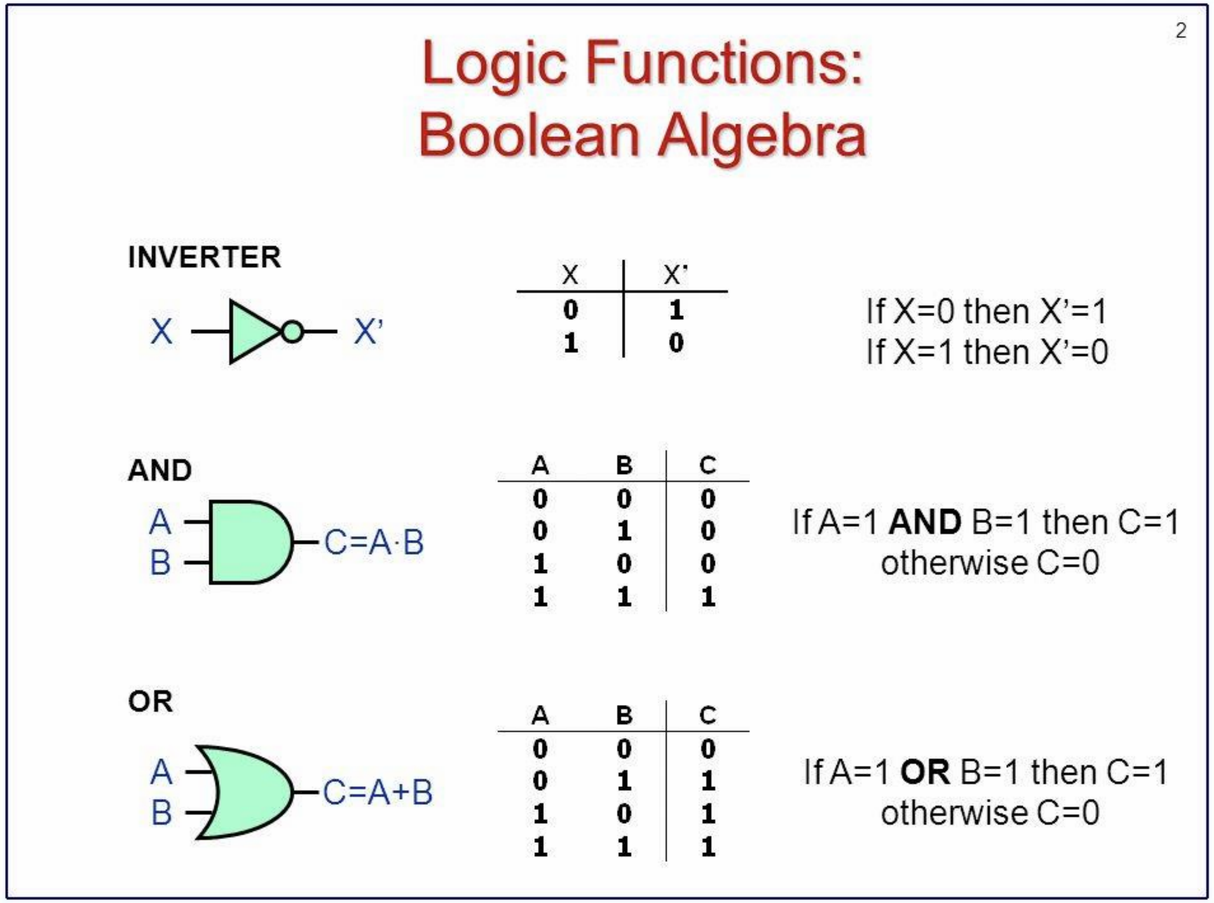
Basic computer math

- Boolean Algebra
- Bitwise operations
- Truth table

Boolean Algebra

- Boolean algebra was invented by mathematician George Boole in 1854.
- Boolean Algebra is used to analyze and simplify the digital (logic) circuits. It uses only the binary numbers i.e. 1 and 0. Or true and false, or high and low.
- AND, OR, and NOT are the primary operations of Boolean logic.

Boolean Algebra



Boolean Algebra - Logical Operators (For hardware, Simon 2.2 Gates)

NOT

- Output is true, if input is false
- Symbolically: !A, ~A, or

AND

- Output is true, if both or all inputs are true.
- Symbolically: A · B or A & B

OR

- Output is true, if any input is true (inclusive)
- Symbolically: A + B, or A | B

XOR

- Output is true, if one or the other is true, but not both (exclusive)
- Symbolically: A ^ B or A$\oplus$ B

Bitwise Operations

Bitwise NOT Example

~

```
typedef unsigned char uint8_t foo;  uint8_t bar;
foo = 0x8E; /* binary 1000 1110 */
bar = ~foo;
```

----

bar result: 0x71 binary 0111 0001

Bitwise Operations

Bitwise OR Example

|

typedef unsigned char uint8_t foo; uint8_t bar; uint8_t qux;

foo = 0xC3; /* binary 1100 0011 */

bar = 0xA2; /* binary 1010 0010 */

qux = foo | bar;

----

qux result: 0xE3 binary 1110 0011

Bitwise Operations

Bitwise XOR Example

^

```
typedef unsigned char uint8_t foo;  uint8_t bar; uint8_t qux;
foo = 0xC3; /* binary 1100 0011 */
bar = 0xA2; /* binary 1010 0010  */
qux = foo ^ bar;
-------------------------------------------
qux result: 0x61 binary 0110 0001  */
```

Shift operators

There are two bitwise shift operators. They are

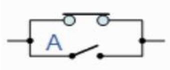
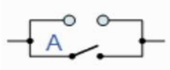
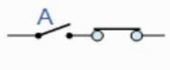
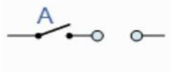
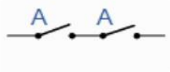
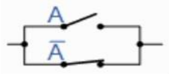
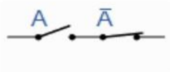
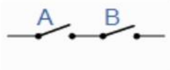
- Right shift (>>)
- Left shift (<<)

Right shift can be used to divide a bit pattern by 2 as shown:
int i = 14; *// Bit pattern 0000 1110*
int j = i >> 1; *// here we have the bit pattern shifted by 1 thus we get 0000 0111 = 7 which is 14/2*

Left shift can be used to multiply an integer by powers of 2 as shown:
int i = 4; */* bit pattern equivalent in binary 0100 */*
int j = i << 2; */* makes it binary 1000, which multiplies the original number by 2 which is 4*2*

Truth Table

https://www.electronics-tutorials.ws/boolean/bool_6.html

Boolean Expression	Description	Equivalent Switching Circuit	Boolean Algebra Law or Rule
A + 1 = 1	A in parallel with closed = "CLOSED"		Annulment
A + 0 = A	A in parallel with open = "A"		Identity
A . 1 = A	A in series with closed = "A"		Identity
A . 0 = 0	A in series with open = "OPEN"		Annulment
A + A = A	A in parallel with A = "A"		Idempotent
A . A = A	A in series with A = "A"		Idempotent
NOT $\overline{A}$ = A	NOT NOT A (double negative) = "A"		Double Negation
A + $\overline{A}$ = 1	A in parallel with NOT A = "CLOSED"		Complement
A . $\overline{A}$ = 0	A in series with NOT A = "OPEN"		Complement
A+B = B+A	A in parallel with B = B in parallel with A		Commutative
A.B = B.A	A in series with B = B in series with A		Commutative
$\overline{A+B} = \overline{A}.\overline{B}$	invert and replace OR with AND		de Morgan's Theorem
$\overline{A.B} = \overline{A}+\overline{B}$	invert and replace AND with OR		de Morgan's Theorem

BREAK 2

Embedded Development Environment overview

- IDE
- IAR
- Common views

Integrated Development Environment (IDE)

1

Editor you use to write program

2

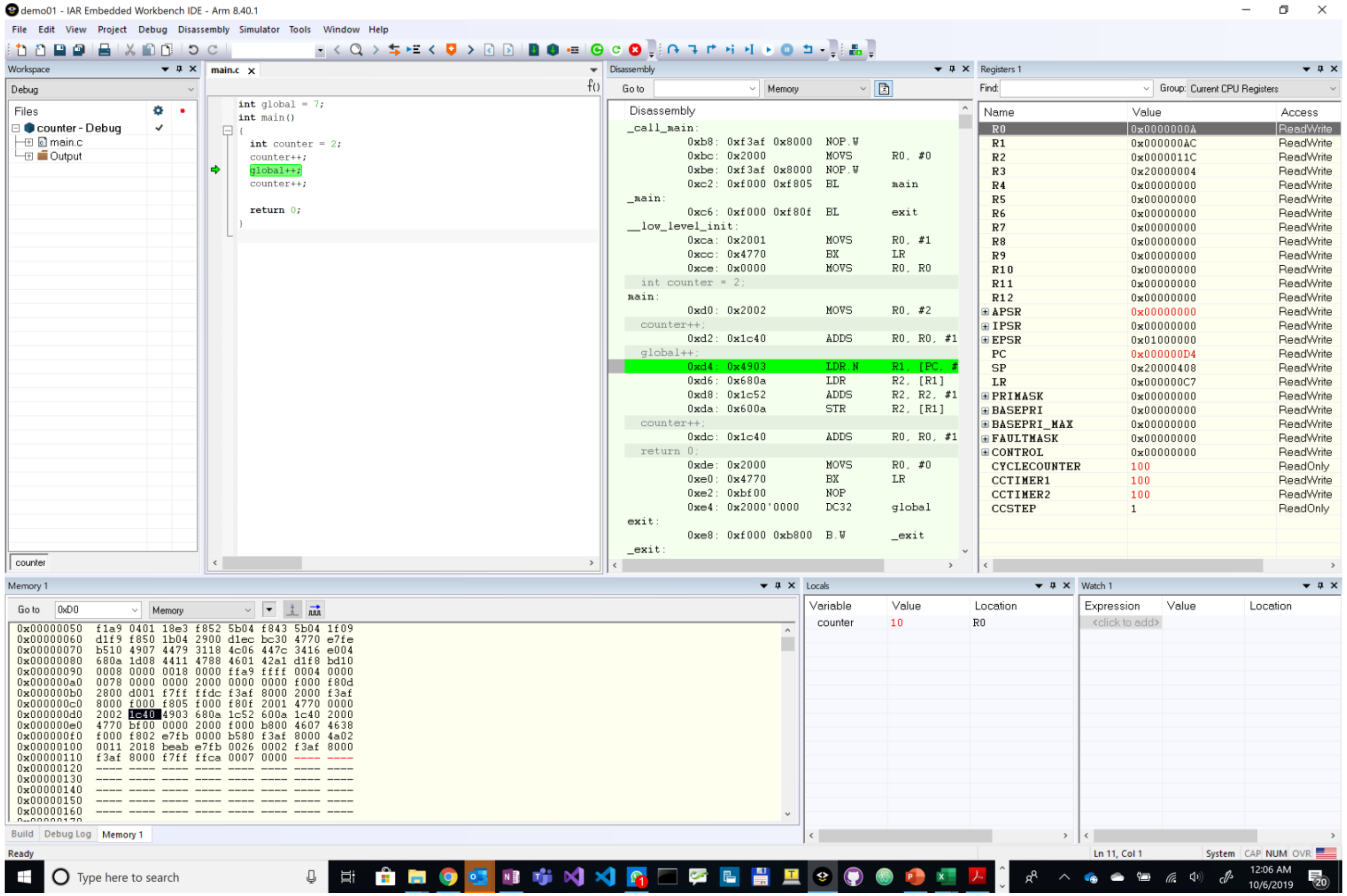
Compiler convert code to binary file that is readable by the processor

3

Load compiled code to flash

4

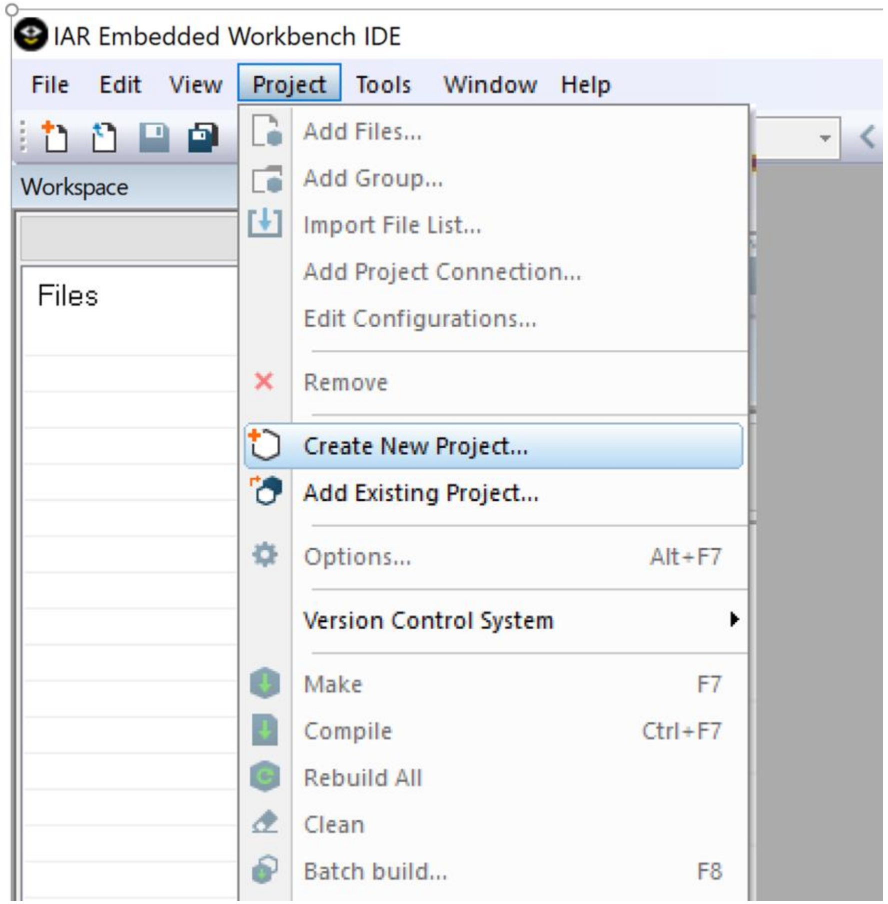
Instruction set simulator to run the program on the simulated environment as if running on the real processor

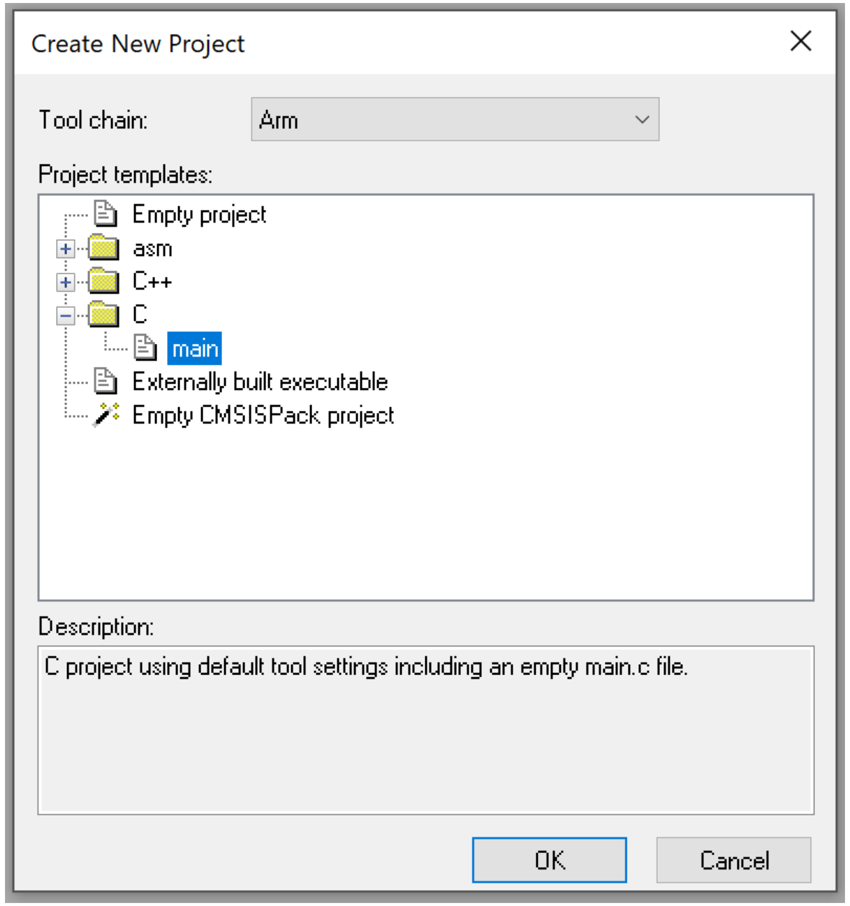


IAR Embedded Work Bench

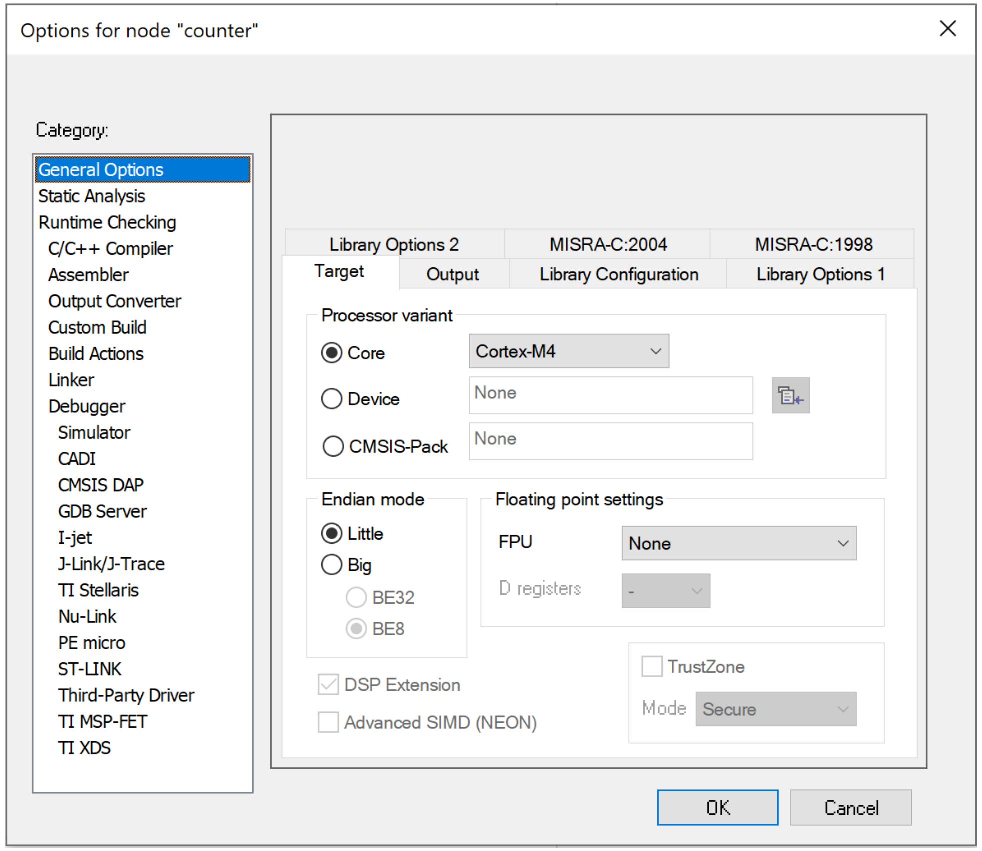
DEMO

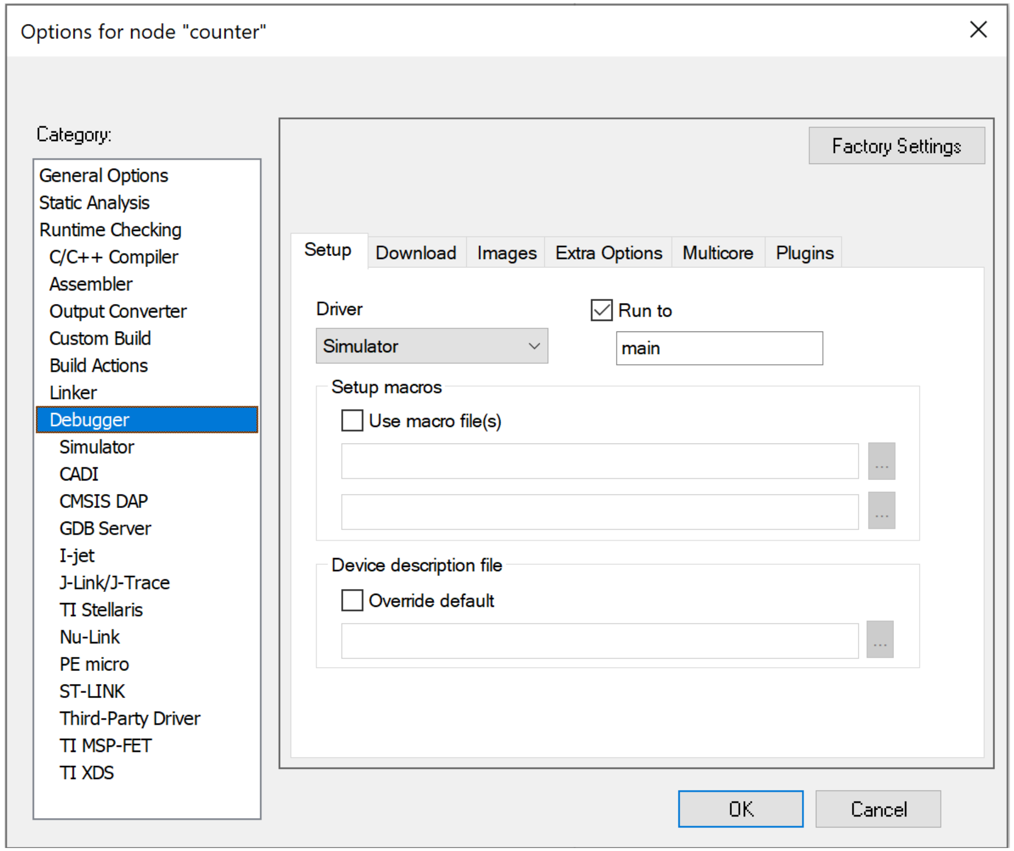
Create New Project



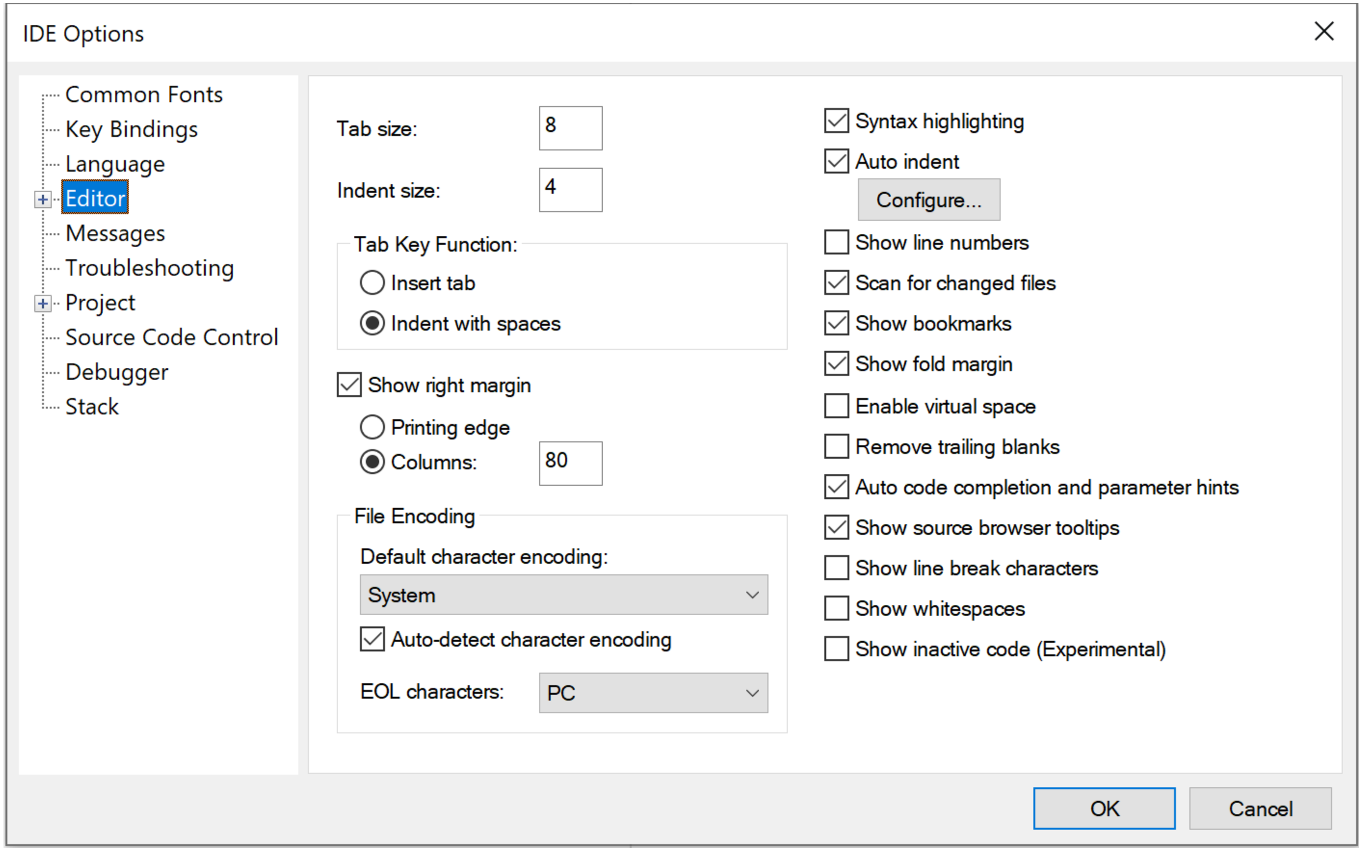


Project --> Options





IDE Options


IDE Options
Common Fonts
Key Bindings
Language
Editor
Messages
Troubleshooting
Project
Source Code Control
Debugger
Stack
Tab size:
8
Indent size:
4
Tab Key Function:
Insert tab
Indent with spaces
Show right margin
Printing edge
Columns:
80
File Encoding
Default character encoding:
System
Auto-detect character encoding
EOL characters:
PC
Syntax highlighting
Auto indent
Configure...
Show line numbers
Scan for changed files
Show bookmarks
Show fold margin
Enable virtual space
Remove trailing blanks
Auto code completion and parameter hints
Show source browser tooltips
Show line break characters
Show whitespaces
Show inactive code (Experimental)
OK
Cancel

Hello "counter" example

```
int global = 7;
int main()
{
  int counter = 2;
  counter++;
  global++;
  counter++;
    return 0;
}
```

Running IAR – Debug with simulator

- Allows execution of one instruction at a time, so you can observe registers and memory that are changing
- We’ll be using the simulator for ARM Cortex M4 processor

Running IAR - Views

- Disassembly
- Register
- Memory
- Locals
- Watch

Running IAR – Disassembly View

- Each instruction has an address
- Machine instruction inside are just binary numbers
- Mnemonics added by IAR for readability
  - "A **mnemonic** is a symbolic name for a single executable machine **language** instruction (an opcode)",
    - Source: https://en.wikipedia.org/wiki/Assembly_language

Running IAR - Registers

- What is a register?
  - Very fast memory
  - Associated with the processor
- ARM has 16x 32-bit registers
- Machine instructions manipulate registers directly
- What are their purpose?
  - R0 – R12: general purpose
  - R13 – Stack pointer: used when calling a function
  - R14 – Link register: used when returning from a function
  - R15 – Program counter: points to the current instruction

Running IAR – Memory View

- Big table of bytes
- Address – numbered sequentially
- 0x00000000 → 0xFFFFFFFF
- Memory regions – appear in the same space, in different area access with different buses
  - Executable space
  - Global data space
  - Shared memory space
  - Peripherals

Assignment 1.c

Follow instructions in the "EMBSYS100_Toolchain_Instructions.pdf" document to install IAR and run "Hello World" program.

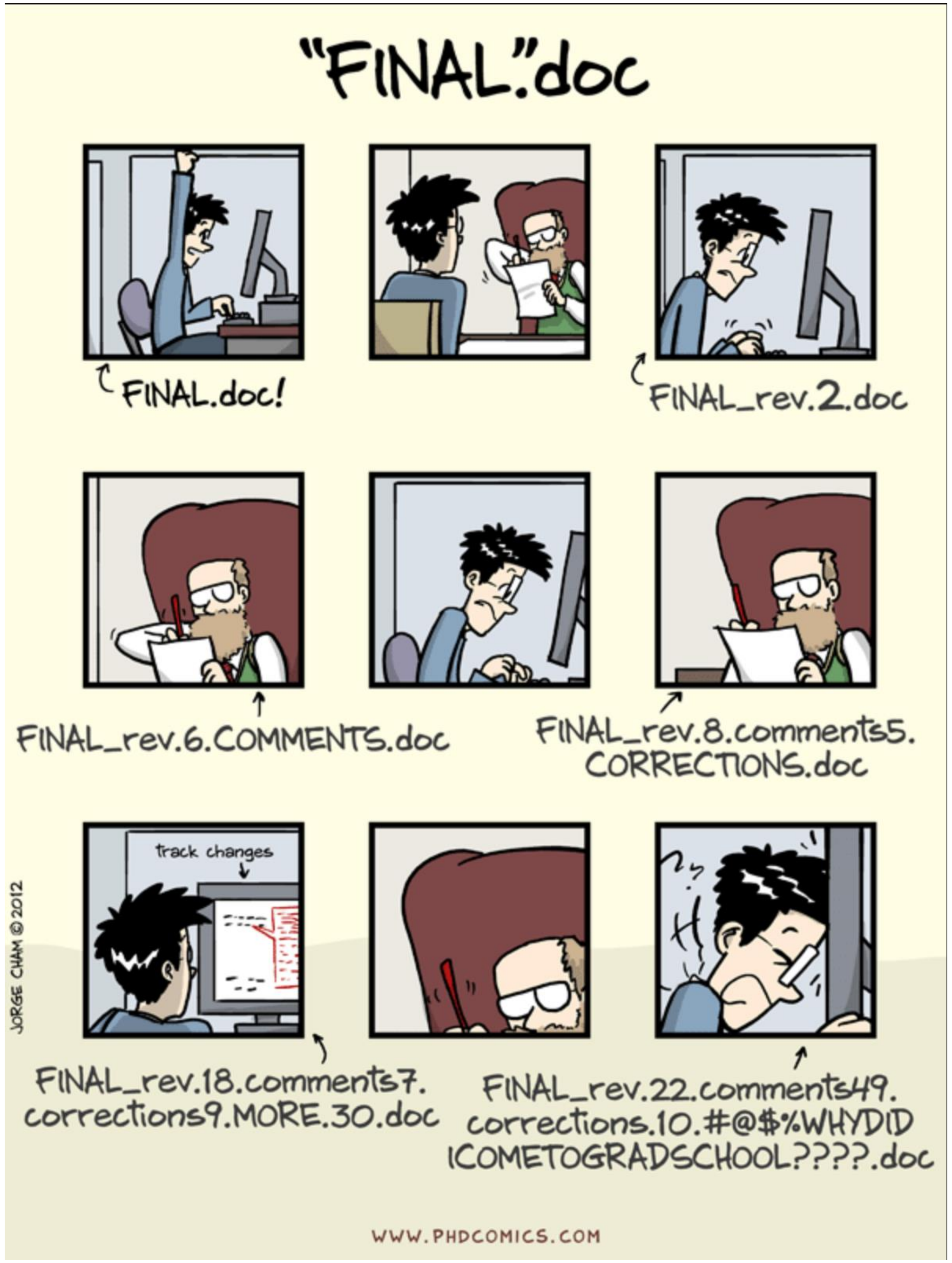
Software Version Control

GIT | BASIC GIT FLOW | SETTING UP REPOSITORIES

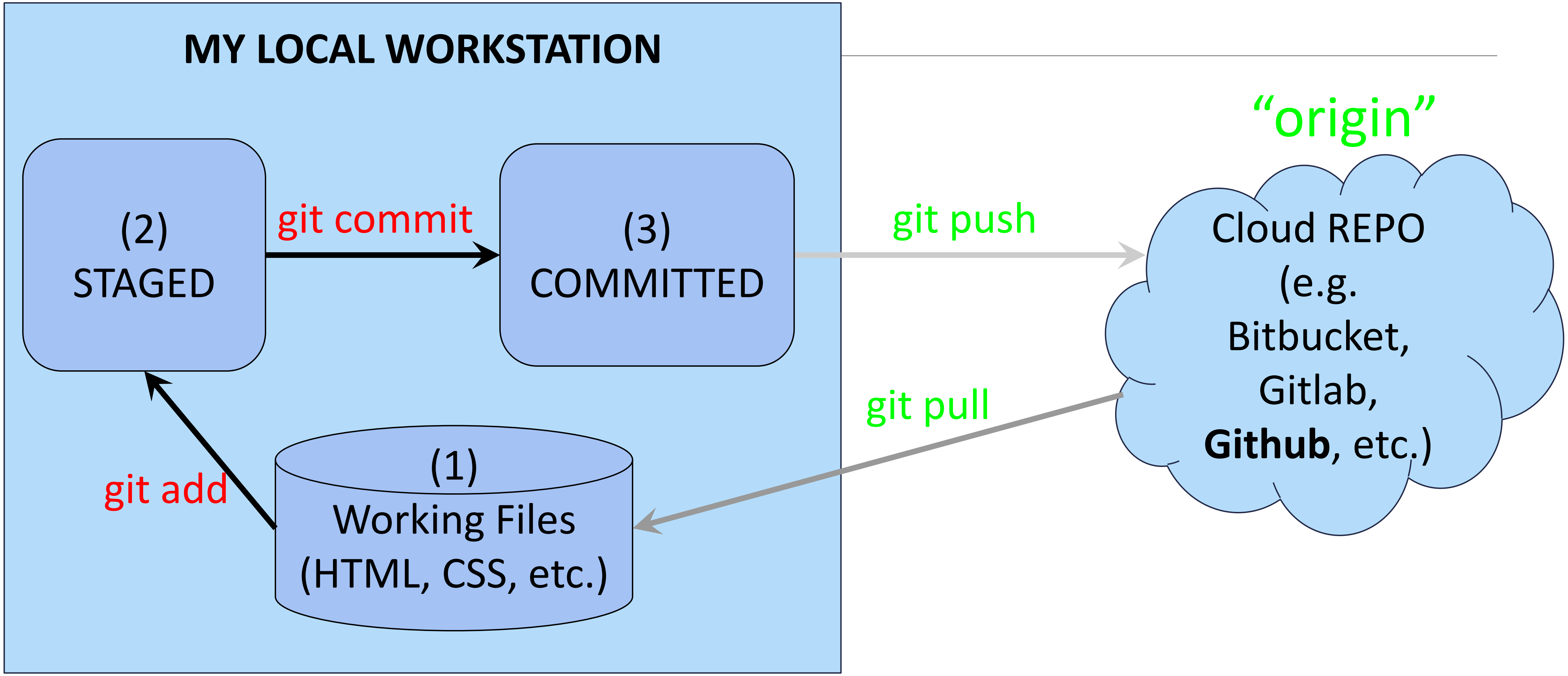
Why?

"Piled Higher and Deeper"

by Jorge Cham: www.phdcomics.com



GIT : Workflow



GIT : Some Commands

GIT CMD	Description
`# git clone <repo url>`	Start a brand new GIT Repository (***after creating it in the cloud***)
`# git status`	Checks the current state of your local GIT repository
`# git add -A`	Adds untracked files to your staging area
`# git commit -m “comment”`	Adds staged files to the committed area
`# git push`	Check-in (or synchronize) with the cloud ***origin*** repository
`# git log`	See what GIT activities we’ve been accomplishing
`# git pull`	Gets the latest up-to-date content from the ***origin*** repository in the cloud

GIT : Some Resources

LINK	Description
GIT	Main GIT Website for git tools, Windows Cmd Line
BitBucket	Popular GIT Repo Site with great tutorials
GITHUB	Most popular site on the web for GIT Repos
GITLAB	Another popular site on the web for GIT Repos
TRY GIT	GIT Tutorials
GitHub Guides	Tutorials for using GitHub

Lab01 – GitHub “Hello World”

- Sign Up for a FREE GitHub Account
- Please follow the online instructions:
  - https://guides.github.com/activities/hello-world/
- In this lab you will learn how to:
  - Create a repo
  - Create a branch
  - Make and commit changes
  - Open a Pull Request
  - Merge a Pull Request

Using GitHub for the assignments

- Create a new repo and name it:
  - https://github.com/<yourname>/embsys100
- Example:
  - https://github.com/tameraw/embsys100
- Create a folder for each assignment and check-in your assignment files under that folder.
  - For example, all files for the first assignment should be checked-in under “assignment01”
  - Feel free to add subfolders under the assignment folder if it helps with organization.
- For homework submission in the Canvas website, simply provide a link to your folder in your GitHub repo
  - For example: https://github.com/tameraw/embsys100/assignment01

Assignment 01

Reading

- https://en.wikipedia.org/wiki/Embedded_system
- ***"An Embedded Software Primer" by David E. Simon***
  - Chapter 1
  - Chapter 2.5