Embedded Systems & Microcontrollers

Prof. John McLeod

ECE9047/9407, Winter 2021

A whole lot of concepts regarding embedded systems and microcontrollers are presented. The life cycle of product design, and some debugging theory are introduced. Finally, a general description of microprocessor and microcontroller architecture is given. This lesson concludes with the technique of memory mapping in a microcontroller.

Embedded Systems

What is important about embedded systems? Embedded systems are ubiquitous, and are increasingly added to just about everything.

Definition: **Embedded systems** are special-purpose computers designed for a specific, and often limited purpose.

As information technology becomes cheaper and more reliable, more and more technologies are based on embedded systems. Cell phones, traffic light controllers, programmable thermostats, watches, smart appliances, etc. are all embedded systems, when a few decades ago all of these would have had all control systems based on circuits, relays, or mechanical parts.
Embedded systems are an increasingly important part of modern society, and as engineers we need to understand how they work.

- Embedded systems are typically built around a microcontroller core.
- Microcontrollers often don't have operating systems, and only run limited, custom programs.
- Microcontrollers can often be programmed by conventional techniques (many can be programmed in C, for example), but as they often have limited memory and often operate at lower speeds than desktops or laptops, careful consideration of memory usage and program flow is often necessary to optimize performance.

Product Life Cycle

From the initial concept through to real-world deployment, successful products follow a fairly similar development path. For more complex systems, or those that are frequently refreshed, this path may come full circle multiple times. A schematic of a product's life cycle is shown in Figure 1.
During the *analysis phase*, the requirements and constraints are determined, and a set of specifications that meets the requirements is written down. For a commercial product, or something designed under contract, the mutually-agreed-upon requirements form a legally-binding *Requirements Document*.

Definition: A **requirement** is a specific parameter or capability that the system must satisfy. For example, "this device must be handheld" is a requirement.

Definition: A **specification** is a precise set of details that demonstrate how a requirement is met. To continue the above example, "this device will measure $10\,\mathrm{cm} \times 6\,\mathrm{cm} \times 2\,\mathrm{cm}$ and weigh $300\,\mathrm{g}$" is a specification that meets the "handheld" requirement for the device. [1]

[1] For most hands, anyway.

Definition: A **constraint** is a limitation or a boundary on the construction or operation of the system. For example, "this device must not electrocute the user under normal circumstances" is a common constraint for most personal electronics.

During the *high-level design phase* a conceptual model of the hardware and software for the system is developed. This model is used

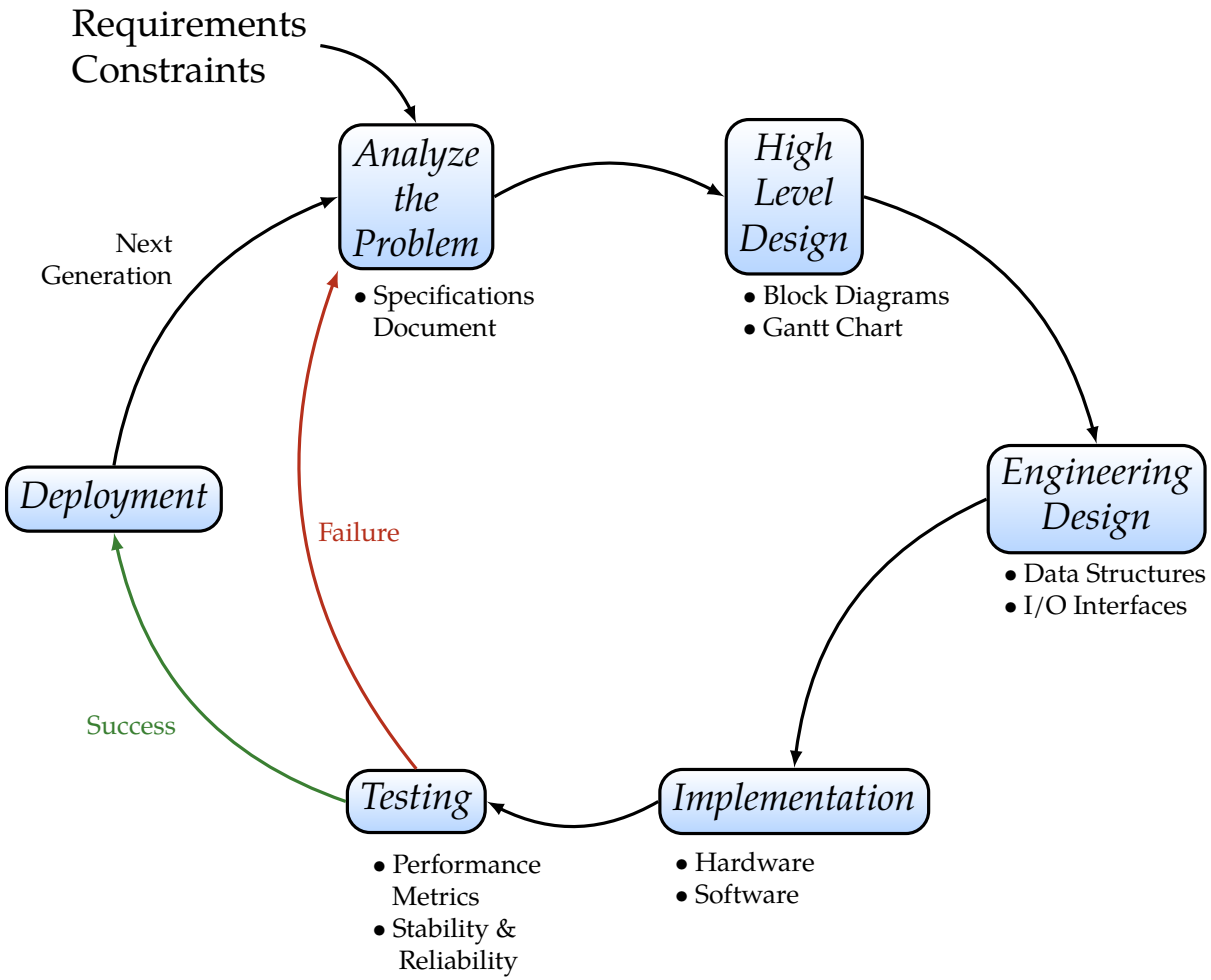

Figure 1: Schematic of a product life cycle. Figure adapted from *Valvano 2.3*.

to break to the project into subcomponents or modules. As much as possible planning should be done for the eventual development, including block diagrams of the system or *Gantt charts* to help identify bottlenecks in the design processes.
The model for the system is planned out in greater detail during the *engineering design phase*. A specific microcontroller should be selected, and the general data structures required for the software, and the I/O peripherals required for the hardware should be determined. Ideally, a complete set of circuit diagrams and CAD models will be developed.
The system, or at least a prototype of the system, is constructed during the *implementation phase*. For a complicated product that will undergo multiple development cycles before it is deployed, the first few implementation phases might be entirely simulations. The primary goal of this phase, especially in early cycles, is to implement a prototype that is a sufficiently accurate approximation of the final product to allow the next phase to provide meaningful results.
The viability and performance metrics of the system are evaluated during the *testing phase*. The prototype must meet all the specifications originally described in the Requirements Document, and other metrics like lifespan, reliability, and likely modes of failure should be evaluated. The final design must be thoroughly debugged before it can be approved. If a product fails in one or more tests, then the life cycle starts again, possibly with some revised requirements or constraints. Otherwise, the product is prepared for release in the *deployment phase*.

Debugging Theory

Everyone who has written a piece of code more complicated than "hello world!" [2] has some experience with the joy of **debugging** software. Any anyone who completed electrical, mechanical, or some equivalent engineering Bachelor's degree is all-to familiar with the joy of **debugging** a robot or some other piece of self-build electromechanical hardware as their design project.

- Let's not argue whether hardware or software debugging is the worst, let's just agree that they both suck.

Since debugging is an almost-unavoidable part of engineering design, it is good to be familiar with the technique.

Definition: A **debugging instrument** is any device (in hardware or in software) that is used to debug, or troubleshoot, a system.

Definition: The **intrusiveness** of a debugging instrument is the degree to which it interferes with the system's normal operation.

If you haven't thought much about the process of debugging before, it may come as a surprise to realize that *all* debugging instruments are intrusive to some degree. If a system has X amount of resources to complete some operation, and measuring that operation with a debugger uses ΔX of that resource, then a *minimally* intrusive debugging instrument is one in which:

$$\frac{\Delta X}{X} \to 0.$$

[2] And some of us even had trouble with that! You know who you are, and there is no shame in it.

For software debugging, often the most sensible resource is the **time** it takes the code to run.

- If the only experience you have writing code is in writing relatively simple programs for your classes, given that modern computers are really fast you probably never noticed any difference between your programming running with and without `print` statements.
- But anyone who has written scripts for batch processing lots of data recognizes that they take significantly longer to complete if they are constantly dumping information to the screen.
- Some software debugging instruments, like setting a **breakpoint** in your code, can be considered *maximally intrusive* — they completely prevent the proper operation of your code.

Embedded systems often have less computing power than desktop or laptop computers, so software debugging instruments like `print` statements or variable tracing in the background can significantly slow down normal operation. Another resource appropriate to software is **memory** — again, in a modern desktop or laptop computer there is often a surplus of memory, and a few `print` statements or running traces on variables does not affect the performance very much. However in an embedded system with limited memory resources, too many of these debugging instruments can make the system unusable.
As another example, in a hardware circuit **electrical power** is another resource. Common debugging instruments such as oscilloscopes, logic probes, or multimeters, are not ideal. A multimeter

measuring voltage, for example, is supposed to have an infinite internal impedance to allow it to measure the voltage without conducting any current through the multimeter itself. Of course a real multimeter has a finite impedance, and in a cheap multimeter this impedance can be relatively low. Measuring the voltage across a component in the circuit, therefore, creates a parallel circuit element (the multimeter) that can conduct electricity. The multimeter therefore dissipates power from the circuit, and as a parallel circuit element it can change the operation of the circuit in unexpected ways — reducing the usefulness of the multimeter as a debugging tool.

The purpose of debugging is to fix errors in the system. However, in complex systems, the measurements from debugging instruments rarely address errors directly (especially subtle ones): a lot of thought and modelling may be necessary to understand how the measurements from debugging indicate the state of the system, and describe how the system is failing to produce the desired performance outcome. Often the first step in debugging is just **stabilizing** the system.

Definition: **Stabilization** is the process of fixing the inputs and the system so the same outputs are reliably produced.

After a system is stabilized, a combination of **black box** and **white box testing** will allow you to zero in on the problem.

Definition: **Black box testing** is the process of recording the outputs of a system for various inputs.

Definition: **White box testing** is the process of tracing how specific inputs lead to specific outputs by monitoring all the internal processes of a system.

Black box testing is important initially, to provide a summary of system performance and identify a set of test inputs that produce failures. Once this is done, white box testing on select cases will help correct the errors in the system.
Once debugging is complete and the system is operating properly, careful consideration has to be given on what do to with the debugging instruments.

- Optimal system performance will theoretically occur when all system resources are devoted to the desired task, so removing all debugging instruments is a good idea in principle.
- However, depending on the **intrusiveness** of the instrument, removing it can change the operation of the system. Many engineers have been unpleasantly surprised to learn that their system only "works" when the debugging instrument is left in place, meaning they accidentally designed their system around the inefficiency introduced by that debugging instrument.
- Even in software, removing debugging instruments can still cause unexpected behaviour. Timing is very important in embedded systems, and as processing speed (and the amount of data that can be processed in one shot) is usually more limited than it is in laptops and desktops, the time delay from

implementing the software debugger can unexpectedly become crucial in the correct operation of the system.

- Ultimately, leaving debugging instruments in place is also a valid design choice, as it makes troubleshooting in the field after deployment that much easier.

Since your ultimate goal is to have a working system, often leaving debugging instruments in place (but with the results of the debugging hidden from users) is a good option if it does not detract from the operation of the system.

Microcontrollers and Family

What is the difference between a **microprocessor**, a **microcontroller**, and a **microcomputer**? I am pretty sloppy with these terms, but let's get some definitions out of the way.

Definition: A **bus** is a set of wires used to move binary information between various electronic circuits.

Definition: A **microprocessor** is a central processing unit (CPU) on a single integrated circuit (i.e., a chip). Outside of the registers in the CPU, a microprocessor has no memory storage, has no permanent memory storage, and has no other peripherals. The main bus for controlling all these components is therefore *external* to the chip.

Definition: A **microcontroller** contains a microprocessor, some memory (often even some permanent memory), basic input/output (I/O) functionality, and often several peripherals (timers, analog-to-digital converters, etc.) all built into the same chip.

Definition: A **microcomputer** is a computer, just made a small as possible. Usually these are designed to run a full-fledged and user-friendly operating system (like Linux) or a crippled and user-unfriendly operating system (like Windows). A microcomputer is usually a device with a small physical footprint, although everything may not be on the same chip. A microcomputer can be considered a microcontroller with more bells and whistles.

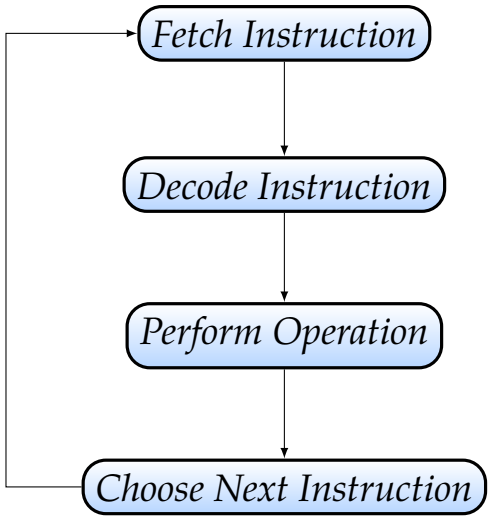

Figure 2: An *extremely* simplified representation of a microprocessor finite state machine.

Signals in a Computer System

There are three types of signals in a computer system: **data**, **addresses**, and **controls**.

Definition: **Data** signals represent multi-bit values (information represented as a binary number).

Definition: **Address** signals represent some location in memory.

Definition: **Control** signals are a collection of bits that provide instructions (to enable devices, tell registers to load a value, etc.).

These signals are all fundamentally the same — they are binary sequences — but they may be generated and handled differently. For **data** signals:

- Data can be *generated* by the CPU, *stored* in the CPU, *read into* the CPU from an external device, or *written to* an external device from the CPU.
- As most memory cells are one **byte** (8 bits) in size, data is usually grouped into integer multiples of a **byte**.

For **address** signals:

- Addresses are *exclusively generated* by the CPU.
- All addresses have the same **width**, and this determines the *address space* accessible by the CPU: If addresses are n bits wide, then the CPU can access 2^n memory locations. [3]

[3] Note that this does not necessarily mean there is 2^n accessible *bits* of memory — as usually each location is one **byte**, there will typically be 2^{n+3} bits of memory, but individual bits cannot be accessed independently.

- Typically the address space is a **word** in size (32 bits, for the ARM®Cortex-A9). This does not necessarily mean that there is that much physical memory available, but it does mean that if more physical memory is present the CPU will not be able to read/write to all of it.

For **control** signals:

- Controls are *exclusively generated* by the control logic.
- They are always given with respect to the CPU. For example, a "read" control means that the CPU is reading data from memory. It doesn't matter how other devices might see the event. For example, after reading from memory, the CPU needs to write that data to a register, but this entire process is still referred to as the "CPU reading from memory", rather than the "memory writing to the CPU", even though the two are equivalent.
- *External* control signals are used to control the behaviour of peripherals.
- *Internal* control signals are used to control the behaviour of the CPU's components (i.e., the arithmetic logic unit, registers, etc.).
- A microprocessor "has as many control lines as it needs". Logically these lines all belong together, like a bus, but because each bit may mean something completely different it does not make sense to think of n control lines as an n bit number.

Memory Sizes

Finally, we should just mention what units of K, M, and G are. These are *binary scales* for memory sizes. By lucky coincidence, $2^{10} = 1024$. Since we are used to thinking in metric and using prefixes that increment by factors of 10^3, it is convenient to define the following prefixes for memory:

- **kilo-** (K) is $2^{10} = 1024$, or approximately 10^3.
- **mega-** (M) is $2^{20} = (1\,\mathrm{K})^2 = 1\,048\,576$, or approximately 10^6.
- **giga-** (G) is $2^{30} = (1\,\mathrm{K})^3 = 1\,073\,741\,824$, or approximately 10^9.

These are usually used as prefixes for units of **bytes**, so $3\,\mathrm{KB}$ is 3×2^{10} bytes or 3×2^{13} bits.
The ARM®Cortex-A9 can access 2^{32} individual memory locations, or $2^2 \times 2^{30} = 4\,\mathrm{G}$ of memory addresses. This is a lot, and writing out 8-digit hex addresses is a bit tedious!

- If I ask you to solve memory-related problems by hand, I will use a more manageable $64\,\mathrm{K}$ of address space $(2^{16}$, so memory address are 4-digit hex numbers$)$. This amount of memory space is common in older microprocessors.

Computer System Memory

For our purposes, memory can be thought of as a long series of **cells**, each cell can store the same fundamental quantity of data.

- As previously mentioned, memory cells are most commonly one byte in width.

Each cell is identified by an **address** and contains a **number**. An **address** is a number that serves as the unique identifier for a single memory cell. The memory chip or circuitry **decodes** the address to identify the memory cell and either read or write to that cell as requested. The actual address decoding process uses a minterm.

- Every memory cell is connected to the n bit **address bus** through an n-input AND gate.
- A unique combination of lines or their complements from the address bus are used for each AND gate, such that for a given address on the bus only one AND gate will be high at a time.

This is shown schematically in Figure 3.
To be useful, memory should support two operations:

Definition: **Load** (sometimes called **read**) refers to the CPU retrieving data from memory and storing it in one (or more) of the registers inside the CPU. In this case, the CPU provides the address and the memory provides the number from the appropriate memory cell.

Definition: **Store** (sometimes called **write**) refers to the CPU putting data into memory. In this case the CPU provides both the address and the number to be placed in the appropriate memory cell.

it should be clear, from the description of these operations, that the **data bus** has to accommodate information travelling into and out of the CPU.

- The **address bus** only has to operate in "one direction": information passes from the CPU to memory. It should be clear from the memory circuit in Figure 3 that nothing in memory is capable of putting information onto the address bus.
- The memory circuit in Figure 3 only allows the **load** operation, so it is not a circuit with full functionality. [4]
- To allow the CPU to **store** to memory, the circuit in Figure 3 should provide a second set of tri-state drivers pointing from the data bus into the memory, and additional combinational logic to ensure that when a memory cell is selected, either only one of the two tri-state drivers are active, or there is an additional control on the memory cell itself to select whether data is being written or read.
- Some sort of global "memory enable" bit may also be necessary if the data bus is used for other purposes — the address line will always have a valid address (as $(000)_2$ is a valid address) regardless of whether or not the CPU wants to do anything with the memory.

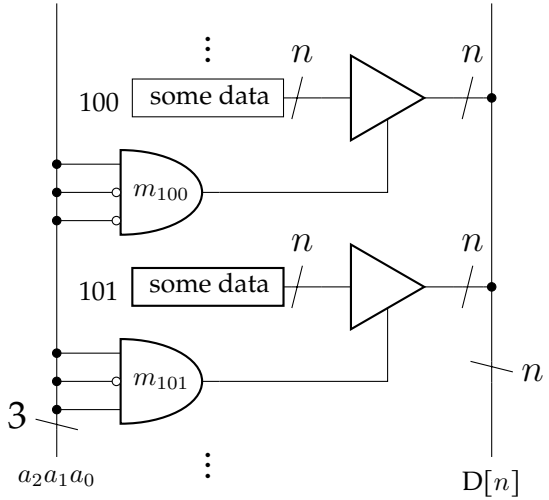

Figure 3: A simplified schematic of using minterms to *load* data. For simplicity, there is only a 3-bit memory space — obviously this is too small to be practical.

[4] Because I am lazy and the figure is already cluttered enough.

Furthermore, note that there is not necessarily any correlation between the size of the **address bus** (the laughably small, but easy to draw, value of 3 is used in Figure 3) and the size of the **data bus** (just referenced as n in Figure 3).

- Most microcontrollers have **byte addressable** memory, where the memory cell (and consequently the data bus) are eight bits wide.
- Some microcontrollers have **word addressable** memory, where either memory cells are larger than a byte, or (more commonly) multiple byte-sized memory cells are grouped under the same address. If you want to access a byte of memory, you have to read the entire word-sized block and then extract the particular 8 bits of interest from a register in the CPU.

The data is referred to as a **number**, but in this context what is a number? Fundamentally, computers understand three different kinds of numbers. These are closely related to the signals discussed above.

Definition: **Data** is a number stored in memory that represents some form of information used in computation. Data is the type of number that is of primary interest to the system's users.

Definition: Memory can also store the **address** of *other* memory cells. When this is implemented in high-level computer languages, it is called a **pointer**. Remember, addresses are also numbers.

Definition: Finally, memory can store **instructions** for the CPU. These are numbers that index a pre-defined table of **operations**. These are often referred to as **opcodes**.

It is important to recognize that the only way the CPU can tell the difference between data, addresses, and opcodes is by context. High-level programming languages typically provide a lot of protection against accidentally mixing these types of numbers, low-level programming languages provide less protection.

- At the lowest level, it is perfectly possible to take two data values and add them together, then treat the result as an address, look up the number stored at that address, add it to the address number, then execute that result as an opcode.
- This is probably a terrible idea... but a CPU doesn't know any better. [5]

[5] The compiler for the ARM®Cortex-A9 that we will use in this course is sophisticated enough to have some protection against doing these sort of shenanigans, but it is still good practice to carefully check your code!

Finally, we will always use hex or binary for addresses (and usually hex, since addresses can be quite large). It never makes sense to use a decimal value for an address. We will always use hex or binary for opcodes. It never makes sense to talk about an opcode as a decimal value. Furthermore, we will typically avoid talking about opcodes in numeric form whenever possible. Human-readable assembly language is difficult enough! Data, however, can be discussed as a hex, binary, or decimal number, depending on what kind of data it is.

- This distinction is important when writing code for a microcontroller. For example, the compiler will not accept an ad-

dress written as a decimal number, but often it will accept (and properly convert) a decimal value stored as data.

Computer Programs

You have probably all taken some sort of computer programming course, so you are probably familiar with the basics of high-level programming. May also be familiar with the construction of basic computer hardware, like registers and adders. But what happens in between?

- A computer program is a set of instructions in memory, usually stored sequentially.
- Each instruction consists of an **opcode** to tell the CPU what do to, and often a **address** and/or some **data** to tell the CPU what work with. [6]

[6] Consequentially, each instruction is often larger than a single memory cell.

The CPU's job is to fetch these instructions one at a time, perform that instruction, then advance to the next instruction.

- Typically, the next instruction is literally at the next address in memory.
- Certain instructions only serve to explicitly provide the address for the next instruction, allowing the CPU to **branch** somewhere else.

Types of Memory

Memory is subdivided into two basic types: **random-access memory** (RAM) and **read-only memory** (ROM). The name "random-access" is a bit archaic, since all modern memory is random-access. This is in contrast to *extremely* obsolete storage technologies like relays, mechanical counters, or delay lines, which can only be sequentially accessed.

- We can both *read from* (i.e., non-destructively access) and *write to* RAM. Both of these processes are "fast" in microcontroller terms (i.e. nanoseconds or less).
- RAM is usually **volatile**, which means it needs an active power source to preserve data. It is possible to make **non-volatile RAM** (often using battery back-ups) but these are expensive and only used in certain specialist applications.

There are also two common subtypes of RAM: **dynamic** and **static** RAM (DRAM and SRAM, respectively).

- In DRAM, data is usually retained as a stored charge on a capacitor. Each time the data is read the capacitor loses a bit of charge, and even when the data is not being read the capacitor will gradually "leak charge". Consequently, DRAM must be periodically refreshed to preserve data quality. The major advantage of DRAM is that it is relatively cheap.
- The RAM listed in the specs for your laptop/tablet/phone/smart watch/poptart is a type of DRAM.

- in SRAM, data is usually retained in a flip flop. These always preserve data quality, hence they are called "static". SRAM is also faster than DRAM. However each bit of SRAM is physically larger and more complex, consequently SRAM is significantly more expensive.

- The *cache size* listed in the specs for the processor of your laptop/table/phone/smart watch/doorbell is usually a type of SRAM.

The name "read-only" is a bit archaic, since most modern ROM can, technically, be rewritten.

- To further complicate matters, ROM is also "random-access".

- Reading data from ROM is "fast" in microcontroller terms (i.e., nanoseconds or less), however writing to ROM is typically "slow" in microcontroller terms (milliseconds or seconds), and sometimes impossible while the microcontroller is operating — the ROM can only be rewritten with an external tool use to program the microcontroller.

For our purposes, ROM only stores programs, not data: The set of instructions telling the CPU what to do are stored in ROM, and the data telling the CPU what to do it with is stored in RAM. There are a few different subtypes of ROM that are still relevant.

- Mask-programmable ROM (PROM) can be written to only once, usually at the factory. After that the programs are essentially hard-coded — no updates are possible!

- Erasable PROM (EPROM) can be erased and rewritten, but it is an invasive procedure: often the EPROM needs to be removed and bathed in uv light. EPROM is a pretty old standard, and is not used much any more.

- Electrically-erasable PROM (EEPROM) can be rewritten with only electrical signals. They are cleverly constructed from transistors with floating gates. EEPROMs allow individual memory cells to be selectively erased and rewritten. Currently EEPROMs have relatively small amounts of storage, and are relatively expensive, but have very long lifetimes.

- Most of you are familiar with **flash**, a form of EEPROM. Flash is cheaper than EEPROM because it compromises by only erasing and rewriting data in blocks considerably larger than one memory cell (512 or more), and having a shorter lifetime than EEPROM.

Most modern microcontrollers have at least one ROM (usually Flash) and at least one static RAM on the chip. Additional memory of any type can be added as a peripheral.

Philosophy of Memory Spaces

Memory is classified by the number of **bits** of storage, and by how these bits are organized. Typically a memory chip is described by the mnemonic $N\mathrm{U}\times n$, where N and n are integers, and U is one of the prefixes for memory sizes (typically K, M, or G).

- The first part, NU, describes the number of addresses on the chip.
- The second part, n, describes the width of each address in bits.

Therefore, knowing your system has $64\,\mathrm{Kb}$ of RAM is fine for impressing your friends, but as microcontroller engineers it is more useful to know if the RAM is organized as $8\,\mathrm{K}\times 8$ ($8\,\mathrm{K}$ memory cells, each 8 bits wide) or $4\,\mathrm{K}\times 16$ ($4\,\mathrm{K}$ memory cells, each 16 bits wide).

Memory organization can further be broken down by the number of address spaces available.

- In **Harvard architecture** there are separate address spaces (i.e., different address buses) for program storage (typically on ROM, as discussed above) and data storage (typically on RAM, as discussed above).
- In a **von Neumann architecture** the same address space is used for program and data storage.

Many microprocessors, including the ARM®Cortex-A9 used in this course, have **von Neumann architecture**.

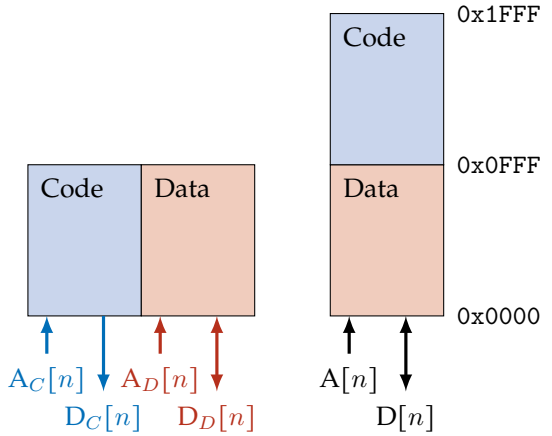

Figure 4: Equal amounts of memory for program storage ("code") and data storage ("data"): Harvard architecture on the left, von Neumann architecture on the right. The Harvard architecture requires fewer addresses, but twice as many buses.

- Programming a microprocessor with Harvard architecture requires a different read/load **opcode** for instructions and data.
- A small advantage of a Harvard architecture system is that it removes the ambiguity on whether a number stored in memory is an instruction or data (however, data and addresses can still both be stored in the same memory space).
- A larger advantage of a Harvard architecture system is that it can fetch instructions and data simultaneously, making it faster than a von Neumann architecture system.
- A von Neumann architecture system benefits from a simpler instruction set, and simpler (cheaper) hardware.

Many microcontrollers prioritize low cost and simplicity, and use von Neumann architecture, but several others do not. Most modern personal computers use a modified Harvard architecture, which preserves separate buses for instructions and data, but have a fuzzier definition of "instruction" and "data" — information that is technically data (i.e. not an opcode) can be stored in instruction space if it is not modified by the program (for example, the constant π). But enough about that.
The relationship between memory and other peripherals — particularly input/output peripherals, is also important.

- In a **I/O mapped** microprocessor, I/O peripherals occupy a separate, special memory space. This again necessitates separate read/load opcodes for working with peripherals, as they

are physically connected to the CPU by different hardware (buses) than the memory.

- In a **memory mapped** microprocessor, all the peripherals occupy the same memory space as the data (and the instructions, in a von Neumann architecture).

The ARM®Cortex-A9 microprocessor uses memory mapping. A major advantage of memory mapped devices is in simplicity: reading from an input device (a bank of switches, for example, or something more complicated) or writing to an output device (a set of LEDs, for example, or something more complicated) is done exactly the same way, and with exactly the same opcodes, as reading from or writing to memory.

Memory Decoding

A Harvard architecture I/O mapped microprocessor will have an excessive amount of separate buses all connecting different devices to the CPU. It may give the hardware engineering a headache, but programming the microprocessor is straightforward. A von Neuman architecture memory mapped microprocessor, however, has only one address bus and only one data bus connecting to a variety of different devices. It makes the hardware engineering easier, but presents us with a challenge: *How do all of these different devices know when it is their turn to talk?* This problem is solved by the following steps:

1. Choose a **memory map**. This is the decision of which block of addresses to assign to RAM, ROM, and I/O.

2. Use low-order address bits to select locations *within each device*. If the address bus is 16 bits, and it is connected to a $8\,\mathrm{K} \times 8$ RAM chip, then the lowest 13 address bits go into the chip (as $8\,\mathrm{K} = 2^{13}$).

3. Design combinational logic using the high-order address bits to control the **chip enables** to select that particular chip.

Often these steps are already done in an off-the-shelf microcontroller, although some microcontrollers have a memory map that can be dynamically configurable. Even though you can get a lot done with a good microcontroller without messing around with the memory map, it is still important to study this subject as it is an excellent exam problem!

Memory Mapping

To explain memory mapping in detail, it is best to present a simple example. Consider a von Neumann memory mapped system with $4\,\mathrm{K}$ of memory space. Since $4\,\mathrm{K} = 2^2 \times 2^{10}$, this system must have an address bus with 12 lines. In hex, these are the addresses from `0x000` to `0xFFF`. In this address space we have a $2\,\mathrm{K} \times 8$ RAM chip and a $2\,\mathrm{K} \times 8$ ROM chip

- Since $2\,\mathrm{K} + 2\,\mathrm{K} = 4\,\mathrm{K}$, these two chips completely fill the memory space available.
- However it is completely up to us which specific $2\,\mathrm{K}$ memory addresses are assigned to the ROM chip (and the remaining go to the RAM chip).
- Unless you are a sociopath, you will also recognize that the only practical option is to assign a contiguous block of addresses to each chip.

As an arbitrary design choice, we will put the RAM chip at the "bottom" of the memory space, using addresses `0x000` to `0x7FF`. The ROM chip will then go at the "top" of the memory space, using addresses `0x800` to `0xFFF`. This is shown schematically in Figure 5. Typically memory maps are drawn vertically this way, with the lowest address at the bottom and the highest address on the top. However, I reserve the right to draw memory maps sideways, especially when the memory maps with very large address space is large.

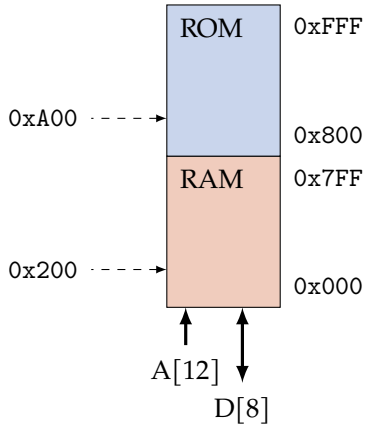

Figure 5: Memory map used in the example.

Chip Select Logic

To continue the above example, as each of these memory chips has $2\,\mathrm{K}$ of memory cells, they need 11 input lines for the address bus, and both will have an 8 input/output lines for the data bus. What happens if we try to retrieve the number stored at address `0xA00`?

- The address `0xA00` in binary is `0b1010 0000 0000`.
- However, as each memory chip accepts only the 11 lowest-order lines from the address bus, *both* memory chips will recieve a signal of `0b0010 0000 0000`, which means that the RAM chip will try to return the number at `0x200` as well.
- We can't have two numbers on the data bus at the same time!

It is a problem if both memory chips try to "talk" at the same time, since there is only one shared data bus.

- The solution is to use the remaining high-order address bit as a chip select (CS) to specify *which* memory chip to enable.

The RAM chip accepts addresses from `0x000` to `0x7FF`, giving us the a range in binary of:

Address bit	a_{11}	a_{10}	a_9	a_8	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0
Lowest	0	0	0	0	0	0	0	0	0	0	0	0
Highest	0	1	1	1	1	1	1	1	1	1	1	1
Equivalent	0	x	x	x	x	x	x	x	x	x	x	x

As shown above, all possible values of the 11 lowest-order bits must be accepted by the RAM chip, so we can replace these with "don't-cares". We could do a 12-variable K-map if we wanted to... but it should be pretty clear from the table above that the logic that determines whether or not an address is *within the range accepted by the RAM chip* is just: [7]

$$\text{CSA} = \overline{a}_{11}.$$

[7] The notation "CSA" is commonly used for "chip select R**A**M", with additional subscripts added for identification if there is more than one RAM chip.

Similarly, the ROM chip accepts addresses from `0x800` to `0xFFF`, giving us the a range in binary of:

Address bit	a_{11}	a_{10}	a_9	a_8	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0
Lowest	1	0	0	0	0	0	0	0	0	0	0	0
Highest	1	1	1	1	1	1	1	1	1	1	1	1
Equivalent	1	x	x	x	x	x	x	x	x	x	x	x

Again, all possible values of the 11 lowest-order bits must be accepted by the ROM chip, so we can replace these with "don't-cares". The logic that determines whether or not an address is *within the range accepted by the ROM chip* is just: [8]

$$\text{CSO} = a_{11}.$$

[8] The notation "CSO" is commonly used for "chip select R**O**M", with additional subscripts added for identification if there is more than one ROM chip.

To summarize:

- We have a system with an n bit address bus and memory chips that require m address lines.
- We pass the lowest-order m bits to the memory chip, where they will be used *internally* to find the particular memory cell.

- We design combinational logic with the remaining $(n - m)$ highest-order bits to identify when that memory chip should be enabled. This is the **chip select logic**. *Designing a combinational circuit to select all addresses belonging to that chip* is equivalent to finding the logical expression for when the $(n-m)$-input AND evaluates as true for the $(n - m)$ highest order bits that identify that memory chip.

- Alternatively, designing a combinational circuit to select all addresses belonging to that chip is equivalent to designing a decoder to detect a particular $(n - m)$-bit number.

The connections to the RAM and ROM chips, and the chip select logic, is shown in Figure 6. Note that as only one chip can be enabled at a given time, there is no need to use tri-state drivers or any other circuitry at the address and data ports. As discussed above, and shown in Figure 3, the tri-state driver and other circuitry is contained *inside* each memory chip, at the level of each memory cell.
Note that it is not necessary for all memory chips to be the same size. The same address lines that are passed *internally* to larger-sized memory chips can also be used as the chip select logic for smaller-sized memory chips without any issues.

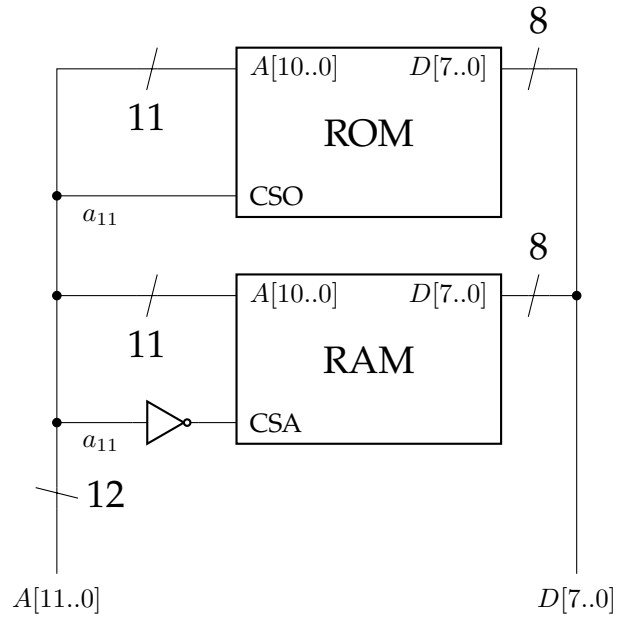

Figure 6: A schematic representation of how the ROM and RAM memory chips are connected to the address and data bus.

Memory Mapping Example

To reinforce what we learned, let's do a more detailed example! Consider the following situation, which is based on the old 68HC12 microprocessors that used to be used in this class. These microprocessors have only a 16 bit address space, and consequently can map to $64\,\mathrm{K}$ of memory cells. In contrast, the ARM®Cortex-A9 which we presently use in this course have a 32-bit address space and can map to $4\,\mathrm{G}$ of memory cells. As noted previously, we'll generally stick to $64\,\mathrm{K}$ address space for "pencil-and-paper" problems because writing really long address numbers gets tedious.

Problem: Design a decoding scheme for the given memory map, where RAM is located between address `0x0000` and `0x1FFF` (inclusive), and ROM is located between addresses `0xA000` and `0xFFFF` (inclusive). This memory map is shown in Figure 7. Note that there is plenty of unused memory space that could be assigned to peripherals.

We have access to as many $4\,\mathrm{K}\times 8$ RAM chips and $8\,\mathrm{K}\times 8$ ROM chips as needed.

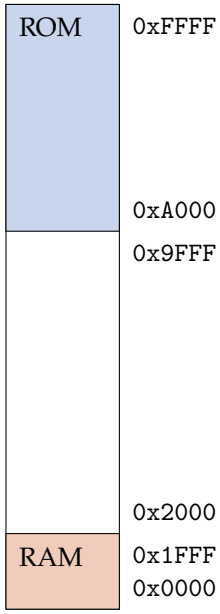

Figure 7: Memory map used in the example.

Solution: First we will calculate how many addresses are needed for the RAM:

$$(\texttt{0x1FFF} - \texttt{0x0000}) + 1 = \texttt{0x2000} = (2^1)(2^4)(2^4)(2^4) = 2^{13} = (2^3)(2^{10}) = 8\,\mathrm{K}.$$

Here I add $+1$ to the difference because the difference is *inclusive* of the end points. As we have $4\,\mathrm{K}\times 8$ RAM chips, it is

pretty clear that we will need two chips to store the desired RAM memory space.

Alternatively, we can convert the size of the RAM chip to hexadecimal:

$$4\,\mathrm{K} = (2^2)(2^{10}) = 2^{12} = (2^4)(2^4)(2^4) = \texttt{0x1000}.$$

Since $2 \times \texttt{0x1000} = \texttt{0x2000}$ it is also clear by this method that we need to two chips for the RAM.

- The *total address space* of RAM (`0x0000` to `0x1FFF`) needs to get divided between the two chips. The only sensible choice is to have one chip take the first half of the addresses (`0x0000` to `0x0FFF`) and the second take the second half (`0x1000` to `0x1FFF`).

For the ROM, we have:

$$\begin{aligned}(\texttt{0xFFFF} - \texttt{0xA000}) + 1 = \texttt{0x6000} &= 3(2^1)(2^4)(2^4)(2^4) \\ &= 3 \times 2^{13} = 3 \times (2^3)(2^{10}) = 24\,\mathrm{K}.\end{aligned}$$

We therefore need three $8\,\mathrm{K}\times 8$ ROM chips to store the desired ROM memory space.

Again we could also convert the size of the ROM chip to hexadecimal:

$$8\,\mathrm{K} = (2^3)(2^{10}) = 2^{13} = 2(2^4)(2^4)(2^4) = \texttt{0x2000}.$$

Since $3 \times \texttt{0x2000} = \texttt{0x6000}$ it is also clear by this method that we need to three chips for the ROM.

- The only sensible choice is to have one chip take the first third of the addresses (`0xA000` to `0xBFFF`), the the second take the second third (`0xC000` to `0xDFFF`), and the third take the final third (`0xE000` to `0xFFFF`).

Writing out the start and end addresses for each chip explicitly in hex and binary, we have:

Chip	Position	Hex Address	Binary Address
RAM 1	Start	`0x0000`	0000 0000 0000 0000
	End	`0x0FFF`	0000 1111 1111 1111
RAM 2	Start	`0x1000`	0001 0000 0000 0000
	End	`0x1FFF`	0001 1111 1111 1111
ROM 1	Start	`0xA000`	1010 0000 0000 0000
	End	`0xBFFF`	1011 1111 1111 1111
ROM 2	Start	`0xC000`	1100 0000 0000 0000
	End	`0xDFFF`	1101 1111 1111 1111
ROM 3	Start	`0xE000`	1110 0000 0000 0000
	End	`0xFFFF`	1111 1111 1111 1111

This makes it fairly clear how to implement the logic for chip selection — the bits used for chip select are highlighted in blue. Notice that address bit a_{12} is used for RAM chip select, but is an internal address bit for ROM chips (as the ROM chips hold more memory cells).

- Students who excel at simplifying Boolean expressions will notice there is a choice here: Since most of the mem-

ory space is not mapped, there is multiple solutions for the chip select logic.

- For example, RAM 1 is uniquely selected when $\overline{a}_{15}\overline{a}_{14}\overline{a}_{13}\overline{a}_{12} = 1$.
- However, given the memory map, RAM in general is selected when $\overline{a}_{15} = 1$, and selecting between RAM 1 and RAM 2 can be accomplished using only bit a_{12}, so a simplified chip select logic for RAM 1 is $\overline{a}_{15}\overline{a}_{12} = 1$.

The full and simplified chip select logic is given below.

Chip	Full Logic	Simplified Logic
RAM 1	$\bar{a}_{15}\bar{a}_{14}\bar{a}_{13}\bar{a}_{12}$	$\bar{a}_{15}\bar{a}_{12}$
RAM 2	$\bar{a}_{15}\bar{a}_{14}\bar{a}_{13}a_{12}$	$\bar{a}_{15}a_{12}$
ROM 1	$a_{15}\bar{a}_{14}\bar{a}_{13}$	$a_{15}\bar{a}_{14}$
ROM 2	$a_{15}a_{14}\bar{a}_{13}$	$a_{15}a_{14}\bar{a}_{13}$
ROM 3	$a_{15}a_{14}a_{13}$	$a_{15}a_{14}a_{13}$

Which type of logic should be used? In this course, the **full logic** expression should be used.

- The simplified logic is "cheaper" in terms of requiring fewer logic gates. But... if a boneheaded programmer tries to read from unmapped memory space, the simplified logic can happily return a number from the wrong address.
- For example, if the programmer tries to read from address `0x2400` = `0b0010 0100 0000 0000`, the simplified

chip select logic will activate RAM 1, and return the number stored at address `0x0400`.

- So the simplified logic is "expensive" in terms of the potential for user errors — based on what you know about programmers (present company excluded, of course) you can guess why using the full logic is a better option.

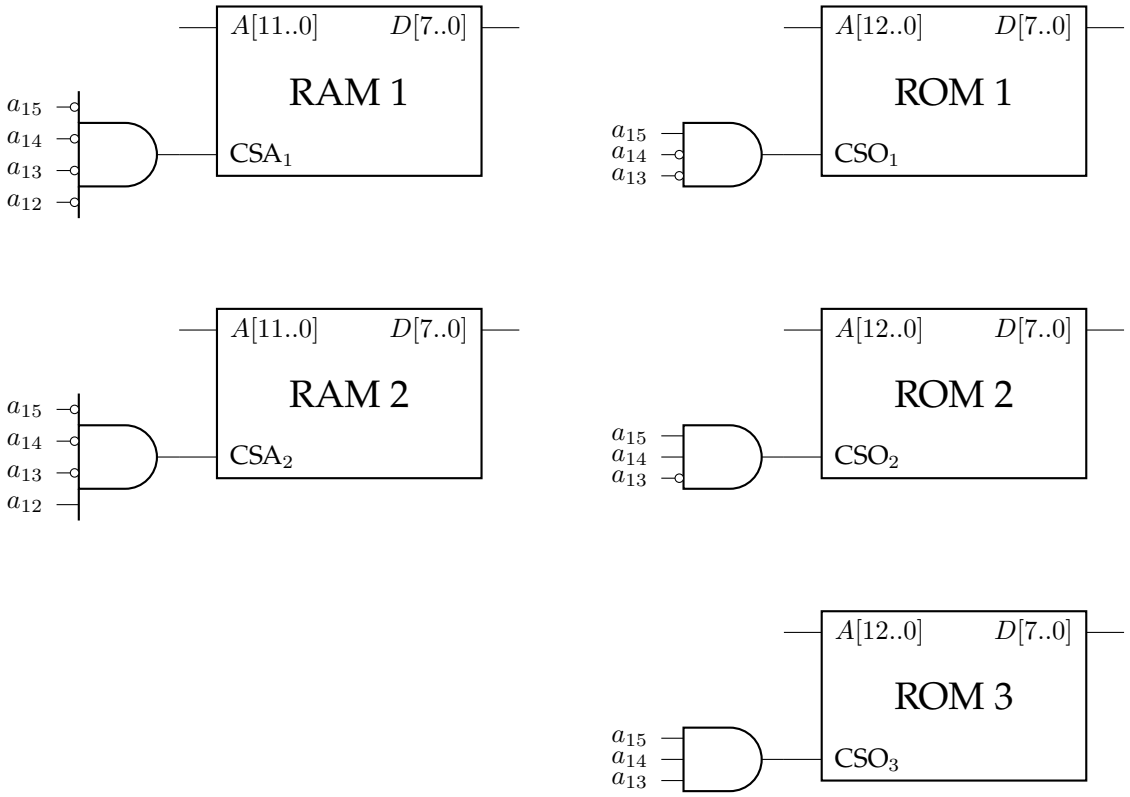

Figure 8: Chip selection logic for this example.