Memory & Memory Mapping

Profs. John McLeod & Arash Reyhani

ECE3375, Winter 2023

This lesson provides some basic information about memory in computer systems, and introduces memory mapping. Memory mapping is a very common and useful technique used to assign the memory address space to different physical devices. Clever students will recognize that memory mapping is a subject that is highly likely to be seen on the midterm and final exams.

Introduction to 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 1.

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.

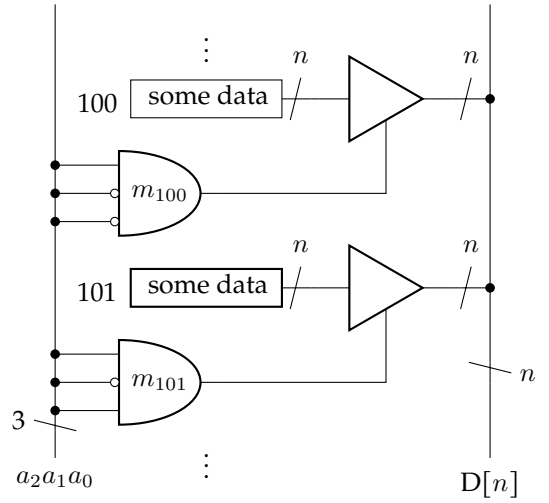


Figure 1: A simplified schematic of using minterms to *load* data. For simplicity, there is only a 3-bit memory space — obviously this is too small to be practical.

- 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 1 that nothing in memory is capable of putting information onto the address bus.

- The memory circuit in Figure 1 only allows the **load** operation, so it is not a circuit with full functionality. [1]
- To allow the CPU to **store** to memory, the circuit in Figure 1 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.

[1] Because I (J. McLeod) 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 1) and the size of the **data bus** (just referenced as n in Figure 1).

- 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 in the previous lesson with regards to the various buses in a microprocessor.

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. [2]

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

[2] 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!

Computer Programs

You have all taken some sort of computer programming course, so you are probably familiar with the basics of high-level programming. You also took ECE2277, so you *should* be familiar with the construction of basic computer hardware, like registers and adders. But what happens in between?

- Software is typically written in a high-level language (Python, Java, C++, etc.; for this course even C is considered "high-level").
- The **compiler** converts the program to an optimized low-level language (i.e., Assembly).
- The **assembler** maps the low-level code to machine language, a hardware-specific binary representation of instructions and data.

For example, consider the following snippet of code written in C. [3]

```
int i;                          // declare loop counter
int total = 0;                  // declare running sum
for (i = 0; i < 10; i++)        // loop 10 times
{
   total += i;                  // add counter to sum
}
```

This code snippet calculates the running sum of numbers from 0 to 9, and stores the result in the integer variable `total`. This is simple enough to understand in human terms, but what exactly does a

[3] You may not be familiar with C (I think your programming course(s) were in Java?), and we will review the code later. This example here is simple enough you can probably understand it even if you are unfamiliar with C syntax.

computer processor do? For one thing, there is no such thing as a "named integer variable" in hardware: there is just storage for binary numbers. An equivalent snippet of assembly code (using ARMv7 assembly syntax) is given below.

```
    mov r0, #0        @ initialize r0 for sum
    mov r1, #0        @ initialize r1 for counter
loop:                 @ flag for start of loop
    add r1, r1, r0    @ increment running sum
    adds r0, r0, #1   @ increment counter
    cmp r0, #10       @ check if counter reached 10
    blt loop          @ branch back to loop if not
```

The syntax for assembly is probably unfamiliar, and we will discuss it in much more detail in the upcoming lessons. For now it suffices to explain that `r0` and `r1` are CPU **registers** that are explicitly used to hold the running sum and the loop counter, respectively. Values are added together appropriately (using `add` and `adds`), then the program **branches** back to the start of the loop if necessary (with `blt`). The assembly language code can be converted line-by-line into the 32-bit machine language used by the ARM®Cortex-A9, as shown in Table 1. Therefore:

- 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. [4]

In the machine language example given in Table 1 each 32-bit num-

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

ber contains the **opcode** and, if necessary, the **data**.

Memory Address	Machine Code	Meaning
0x0000 0000	0xE3A0 0000	Initialize running sum
0x0000 0004	0xE3A0 1000	Initialize loop counter
0x0000 0008	0xE081 1000	Increment running sum
0x0000 000C	0xE290 0001	Increment counter
0x0000 0010	0xE350 000A	Check if counter reached 10
0x0000 0014	0xBAFF FFFB	Branch back to loop if not

Table 1: Machine language for ARM®Cortex-A9 of example assembly code snippet.

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.

Program Execution & Pipelining

To expand on the concepts discussed above, we can elaborate on the CPU execution cycle. As previously mentioned, a CPU contains several **registers** for manipulating data.

Definition: The **program counter** (`pc`) is a special register that holds the address in memory of the next instruction in the program currently running.

When the CPU is powered on, `pc` is initialized in hardware to some default (often `0x0000 0000`). The CPU then cycles through the following steps:

1. Fetches the instruction stored in the memory address held in `pc`.
2. Decodes the instruction.
3. Executes the instruction.

Each of these steps is conducted by special-purpose hardware in the CPU, and is largely transparent to the programmer (and certainly transparent to the user). This hardware is fully utilized by **pipelining** these steps.

- The CPU is a **synchronous sequential circuit**, so each of the above steps takes a clock cycle.
- However the CPU can operate faster than one complete instruction for every four clock cycles.
- Each of these steps uses separate hardware, so they can all be queued.

- If a 32-bit architecture uses a 16-bit instruction set, [5] then two instructions can be fetched at once and held for sequential execution.

This is shown schematically in Figure 2. Note that `pc` must also be incremented — this is not shown explicitly but it should occur basically simultaneously with the fetch step.

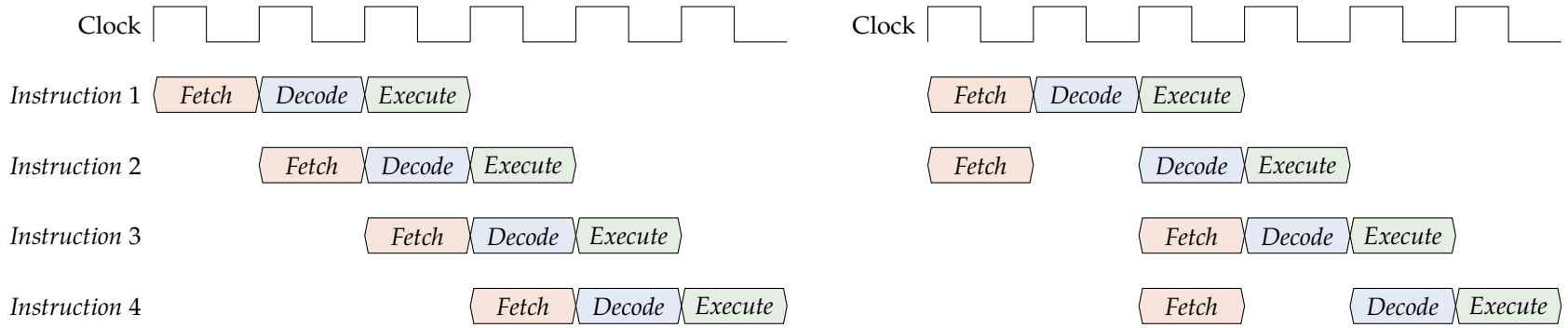


Figure 2: Pipelining four instructions using (left) 32-bit instructions with 32-bit architecture, and (right) 16-bit instructions with 32-bit architecture.

In reality program execution can be more complicated than the simple pipeline schematic shown in Figure 2: consider what might need to happen if the execution step for instruction 2 directs the program to conditionally branch to another part of the program. Fortunately, in this class, you just need a conceptual appreciation for the basics of CPU operation.

[5] An ARM processor can be configured to operating in "thumb mode", which uses 16-bit instructions — but we won't do that in this course.

Program Execution & Hardware

As we have discussed, the CPU has a special register called the program counter (`pc`) that keeps track of the current position in the program currently running. The CPU also has several other general purpose registers for storing and manipulating data during the program execution. Imagine we wish to run the following program:

1. Compute the sum $S = 2 + 3$
2. Store that sum S in memory at address `0x2000`.

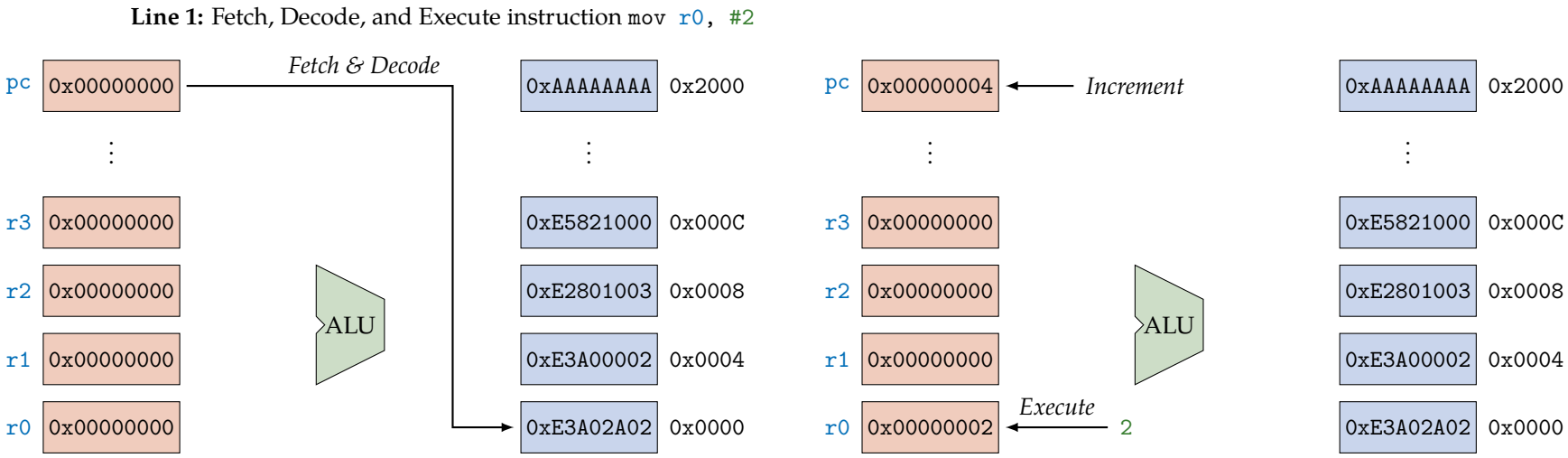
Pretty intense, eh? I bet Microsoft will hire me to work on Windows 12, any day now! This program is pretty simple, but the CPU can't simply add two *actual numbers* together. The CPU only has one "hand" — the data bus — for holding a number. So the simple program above has to be expanded to four steps:

1. Move the data 2 in a CPU register, say `r0`.
2. Add the data 3 to the contents of register `r0`, store that result in register `r1`.
3. Move the data `0x2000` in register `r2`.
4. Store the data in register `r1` at the memory address in register `r2`.

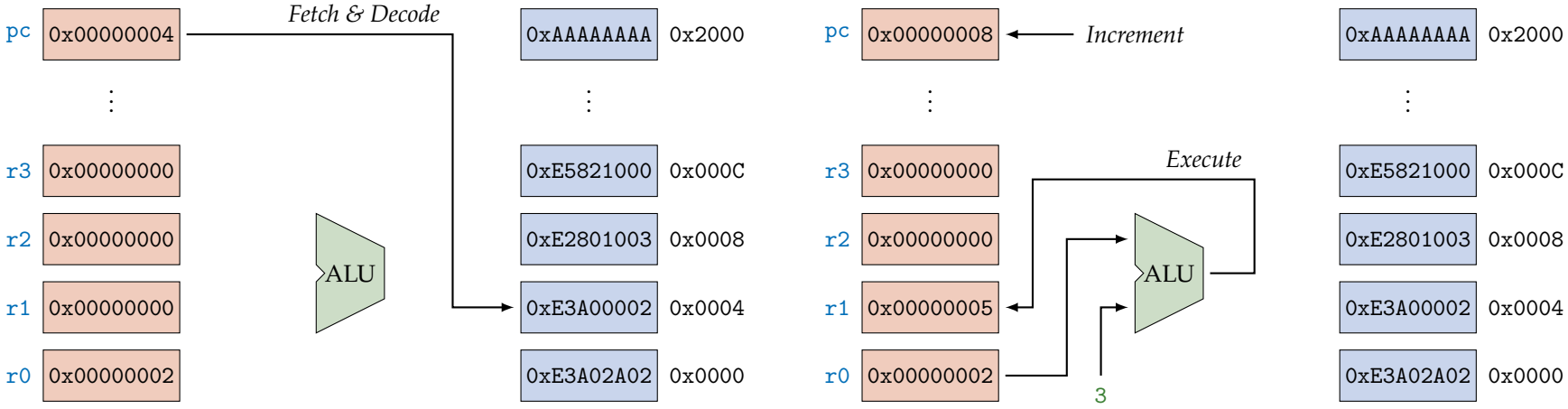
These steps, written out explicitly in terms of what the CPU is actually doing, will hopefully make the ARMv7 assembly code more readable. The assembly code and the corresponding **opcodes** are given below.

```
    mov r0, #2          @ opcode: 0xE3A02A02
    add r1, r0, #3      @ opcode: 0xE3A00002
    mov r2, #0x2000     @ opcode: 0xE2801003
    str r1, [r2]        @ opcode: 0xE5821000
```

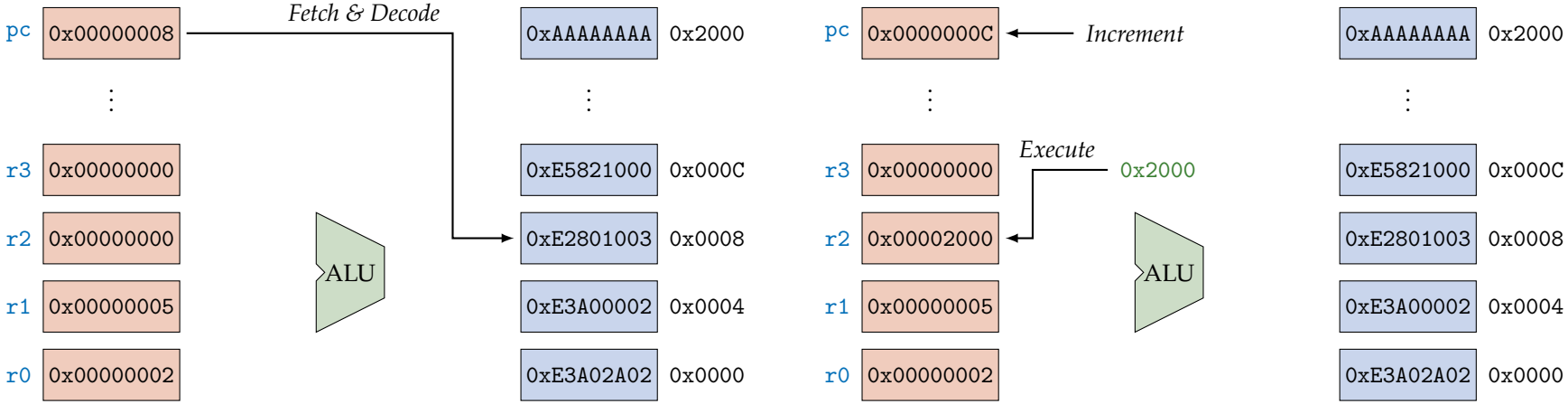
When this code is compiled in the simulator, the opcodes are placed sequentially in memory starting at 0x00000000 and pc is initialized to 0x00000000. The steps in executing this program are shown schematically in Figure 3.



Line 2: Fetch, Decode, and Execute instruction `add r1, r0, #3`



Line 3: Fetch, Decode, and Execute instruction `mov r2, #0x2000`



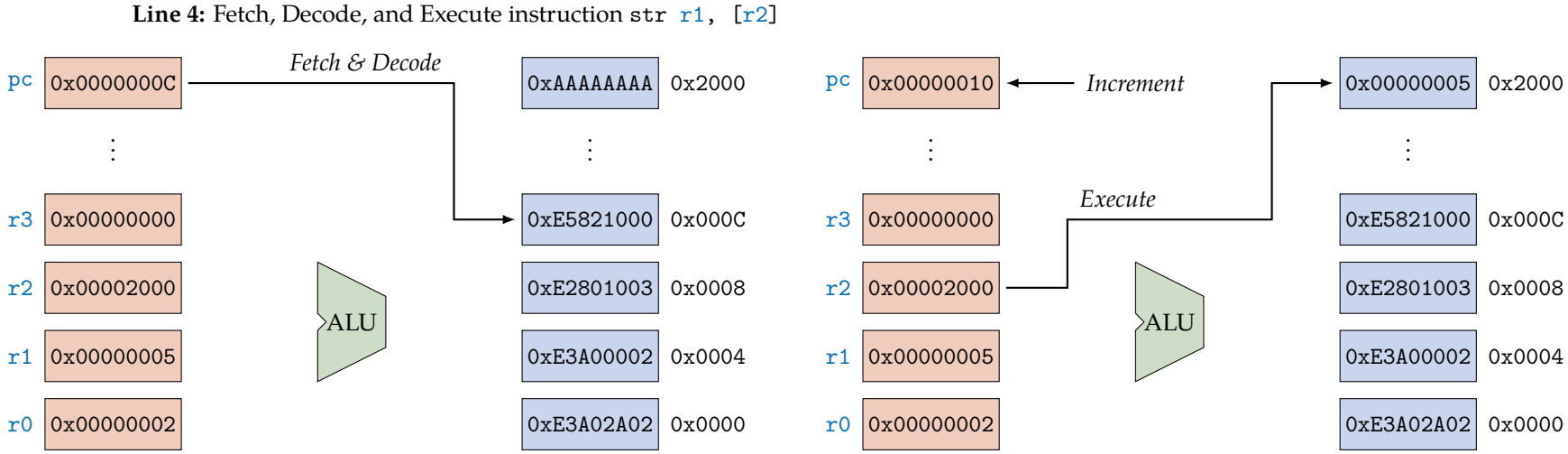


Figure 3: A schematic representation of how a simple program is executed in hardware. The fetch and decode steps are combined and simplified; in reality a large circuit acting as a look-up table would be used to decode the machine code instruction. Also the ALU would be used to increment `pc` after each instruction (effectively executing `add pc, pc, #4`) but this is not explicitly shown.

Understanding the register model of a CPU is helpful for writing code in Assembly language, because it clarifies the limitations of computer hardware. In higher-level programming languages, or when doing our math homework on paper, we are used to writing things like $x = 2(7+5)$, and treating that as a single step. But a CPU can't keep track of three numbers (or two operations, for that matter) at once; they must be stored in a register while the CPU is working with them.

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 re-

moved 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.

There are several other types of electronic circuit that can be considered a form of memory, including programmable logic devices (PLDs) such as programmable array logic (PAL), field-programmable gate arrays (FPGAs), and configurable logic blocks (CLBs). In fact, you used the FGPA chip on the DE10-Standard development board for the lab exercises in ECE2277. We will not discuss these circuits in this course.

RAM Architecture

The fastest and most stable form of RAM uses D flip-flops. A 1-bit memory cell must have the capability to be enabled/disabled, to allow the CPU to **load** it's memory content, and to allow the CPU to **store** new memory to the cell. [6] The basic structure of a 1-bit memory cell of RAM is shown in Figure 4.

[6] Note that in ECE2277 we used "load" to refer to *writing data to the flip-flop*. In this course we assume the perspective of the CPU, so "load" is used to refer to *reading data from memory into the CPU*. Of course the CPU stores this data in it's own flip-flops (in **registers**).

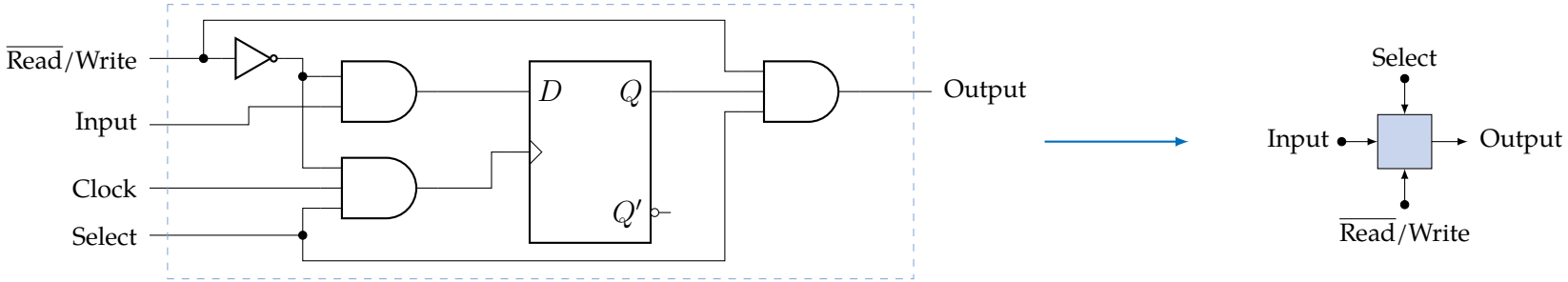


Figure 4: Circuit construction of a 1-bit memory cell of RAM using a D flip-flop, and again as a simple block. Note that the clock input is not explicitly shown in the block diagram (as all parts of the microcomputer have the same clock). An alternative construction using more gates and a SR latch is shown in Prof. Reyhani's notes (and originally from the ECE2277 textbook).

This memory circuit has the following operation:

- When Select$= 1$ and $\overline{\text{Read}}$/Write$= 1$ the value Q in the flip-flop is available on the Output line. The Input line is ignored. This is the **load** operation.
- When Select$= 1$ and $\overline{\text{Read}}$/Write$= 0$ the flip-flop excitation D is set to the value of the Input line on the subsequent clock cycle. The Output line is 0 regardless of the value of Q. This is the **store** operation.
- When Select$= 0$ the Input line is ignored and the Output line is always 0 regardless of the value of Q.

- When Select$= 0$ or $\overline{\text{Read}}$/Write$= 1$, the flip-flop is prevented from seeing a clock edge, so the value of Q remains constant.

The Select input is used to to address a memory unit (usually a **byte**, so eight 1-bit RAM cells will have a common Select input). To achieve this, a $k \times 2^k$ decoder is used. This converts a given k-bit address (i.e. a binary number) into one of 2^k select inputs. The RAM block is then $2^k \times n$ 1-bit RAM cells, where n is the width of the memory unit (again, usually $n = 8$). An example of a RAM block of memory (with $n = 4$ instead of $n = 8$, because I am too lazy to draw that many units) is shown in Figure 5.

The RAM block in Figure 5 has a global enable, used to enable the 2×4 decoder. This is advantageous because it allows us to easily combine multiple RAM blocks to achieve more memory. As we will discuss in more detail when we describe *memory mapping*, we can create a second layer of addresses (with an additional decoder) to use to active at a specific memory block; and as only one memory block is active at a time, each of these blocks can have the same *internal* addresses.

It is worthwhile studying Figure 5 to understand how (and where) data is written to (or read from) for various combinations of inputs, control signals, and addresses.

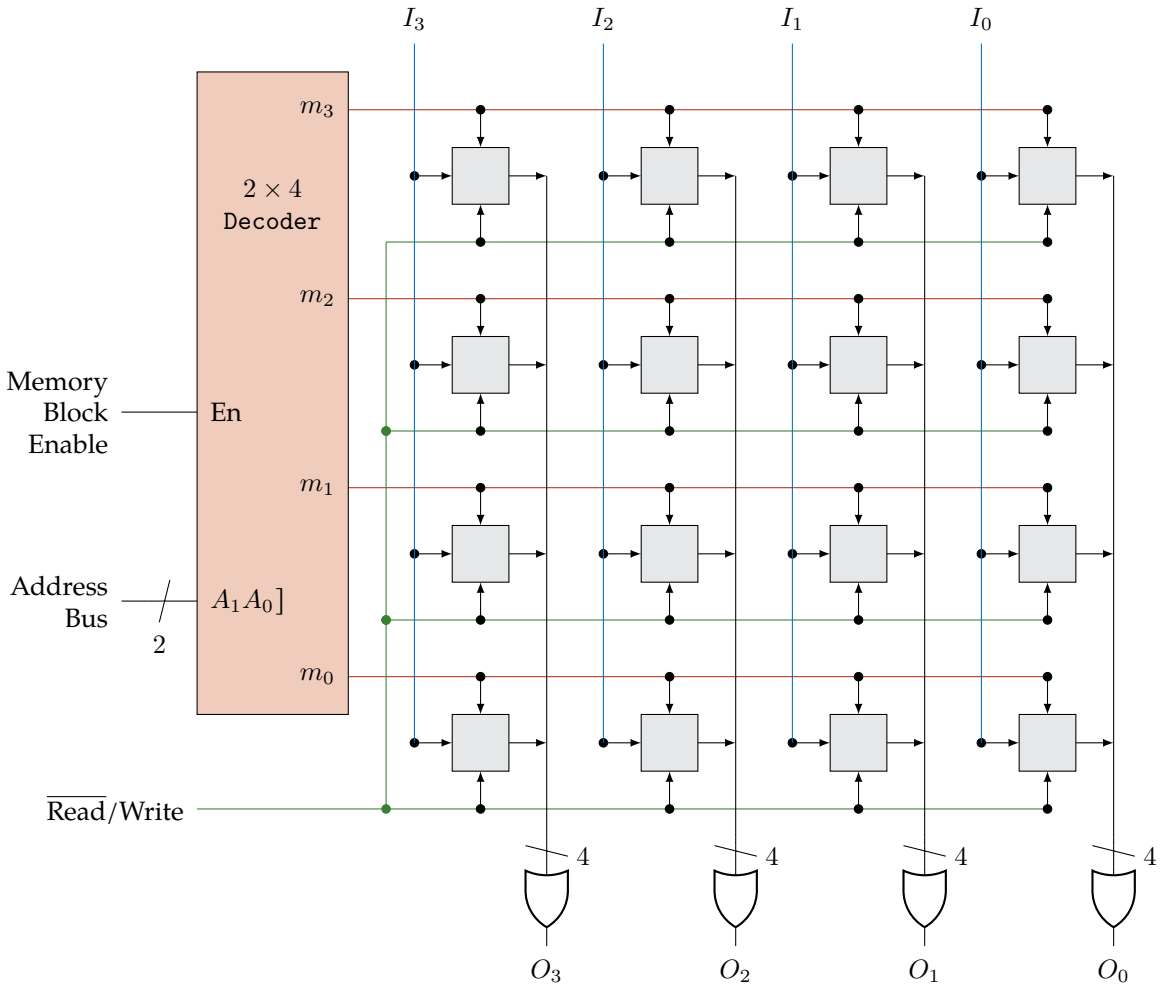


Figure 5: Circuit diagram of a $2^k \times n$ block of RAM memory, where $k = 2$ and $n = 4$.

ROM Architecture

ROM is **non-volatile** (or persistent), retaining information even when powered off. For this reason ROM must be implemented by physically modifying a circuit. The simplest model for $2^k \times n$ ROM memory just uses a $k \times 2^k$ decoder to generate 2^k minterm signals from the k inputs, then have each of the n memory outputs connected to a 2^k-input `OR` gate.

- The statement "each memory address from 0 to $2^k - 1$ contains n-bits of data" is the same as saying "each of the n-bits of data is a Boolean function of k inputs".
- Consequently, "storing data" in the ROM block means deciding whether to connect one of the inputs of the 2^k-input `OR` gates to the equivalent decoder minterm, or whether to ground that input.

This probably sounds confusing, but it isn't that bad. The approach to ROM memory is to draw a grid of horizontal decoder outputs and vertical `OR` gate inputs, then indicate where connections should be made. If no explicit connections is shown, the corresponding wire in the bus is considered grounded. ROM memory is conceptually simpler than RAM memory because the CPU does not have the ability to **store** data to the ROM — the data present in the ROM is a consequence of how the circuit is physically wired.

An example of a ROM block is shown in Figure 6. From this diagram hopefully you see that we can make truly persistent ROM — memory that really cannot be rewritten — by physically wiring

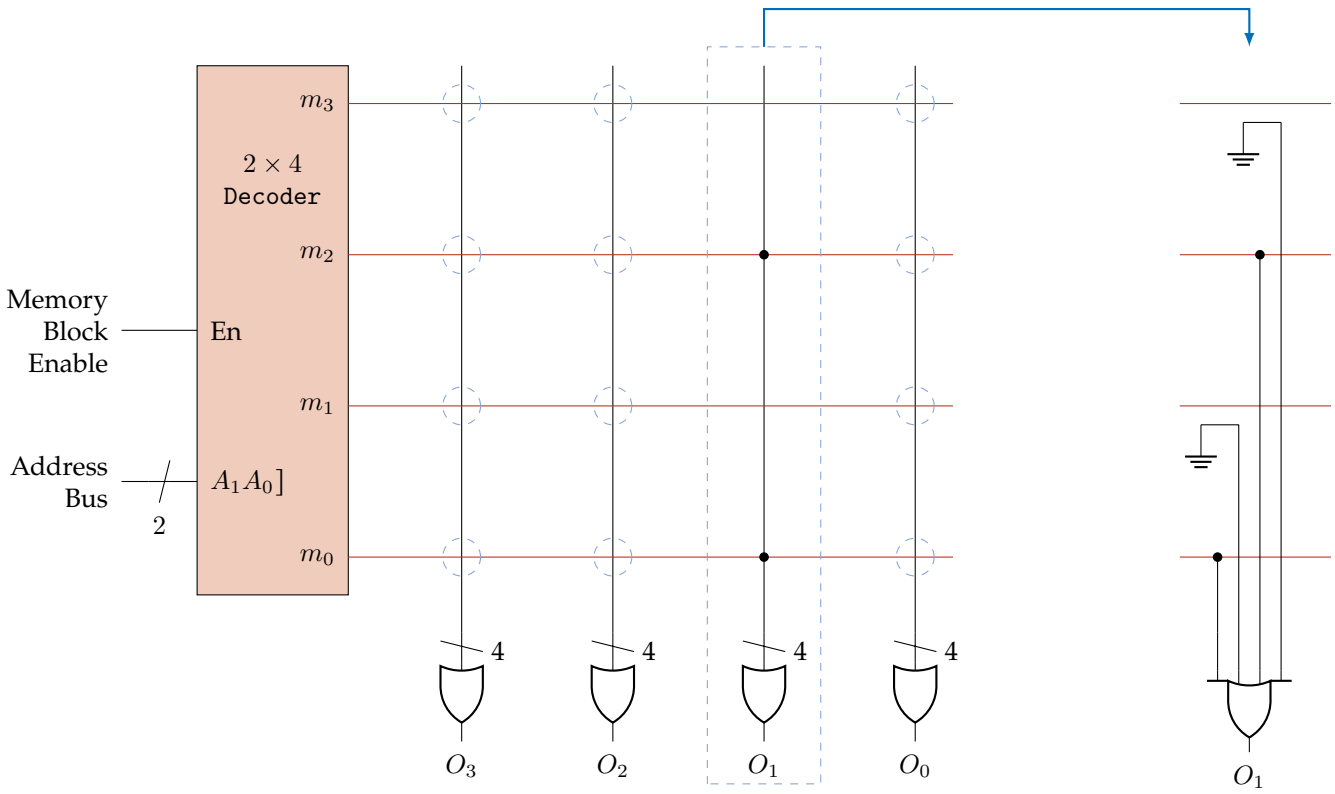


Figure 6: Circuit diagram of a $2^k \times n$ block of ROM memory, where $k = 2$ and $n = 4$. This memory block is abstract — the open circles on the busses for outputs O_3, O_2, and O_1 indicate that connections may or may not be present. The connections on the bus for O_1 are shown explicitly, and in the expanded view on the right the individual wires in the bus are shown.

decoder outputs to `OR` gates. Less permanent solutions include connecting decoder outputs to `OR` gates using fuses or relay switches, allowing the memory to be rewritten. Perhaps you may also see a parallel between the address/output grid in Figure 6 and the **punch card** used in the very early days of computers. In essence, these ROM blocks are really just tables of data: addresses are rows, outputs are columns, and the value in each cell is a 1 if there is a connecting dot, and 0 otherwise.

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 $N\text{U}\times n$, 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 $64\,\text{Kb}$ of RAM is fine for impressing your friends, but as microcontroller engineers it is more useful to know if the RAM is organized as $8\,\text{K}\times 8$ ($8\,\text{K}$ memory cells, each 8 bits wide) or $4\,\text{K}\times 16$ ($4\,\text{K}$ 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 have **von Neumann architecture**. [7]

- Programming a microprocessor with Harvard architecture

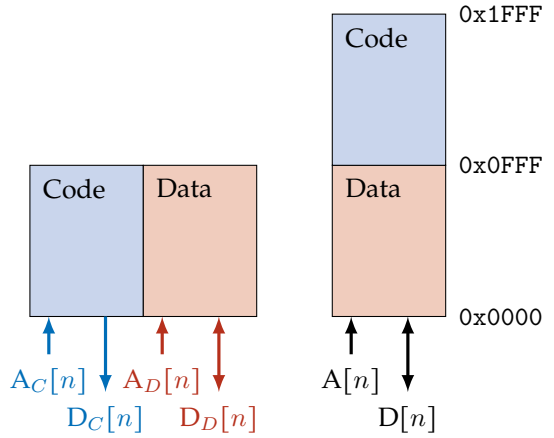


Figure 7: Equal amounts of memory for program storage ("code") and data storage ("data"): Harvard architecture on the left, von Neumann architecture on the right. The Harvard architecture requires fewer addresses, but twice as many buses.

[7] In previous iterations of this course I claimed that the ARM®Cortex-A9 used in this course was von Neumann. However upon reading the technical specifications it is actually **modified-Harvard architecture**, in which instructions and data can be fetched simultaneously, but program code can be moved through memory the same as data — so it is kind of a hybrid of both.

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. [8] 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.

[8] Notably, the ARM®Cortex-M3 and -M4 discussed in the textbook are Harvard architecture.

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 8. 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.

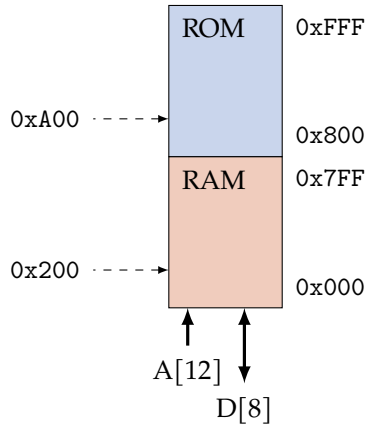


Figure 8: Memory map used in the example.

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 `0b010 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: [9]

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

[9] 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: [10]

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

[10] 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 9. 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 1, 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.

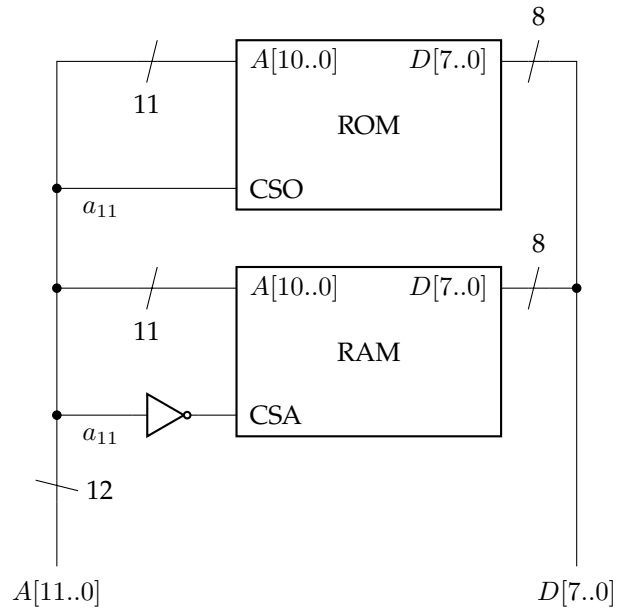


Figure 9: A schematic representation of how the ROM and RAM memory chips are connected to the address and data bus.

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 10. 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.

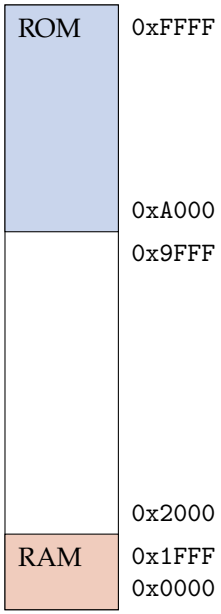


Figure 10: Memory map used in the example.

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 memory 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	$a_{15}\bar{a}_{14}a_{13}$	$a_{15}\bar{a}_{14}$
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 po-

tential 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.

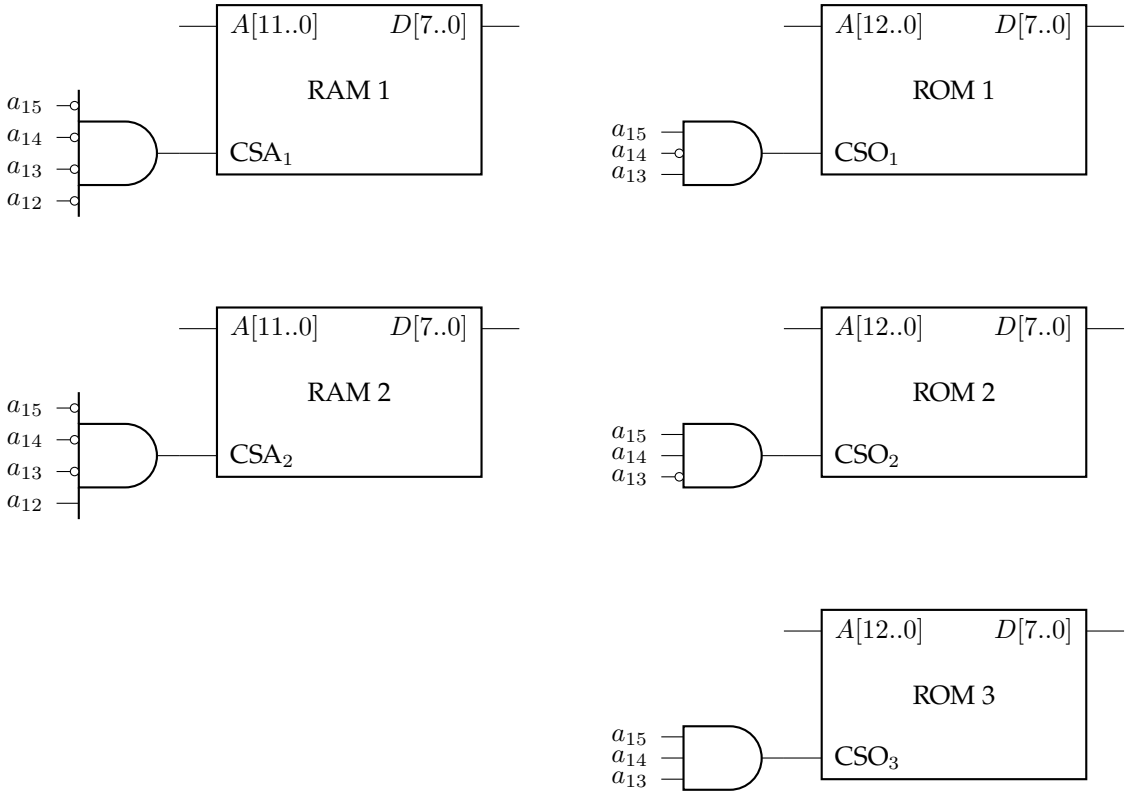


Figure 11: Chip selection logic for this example.