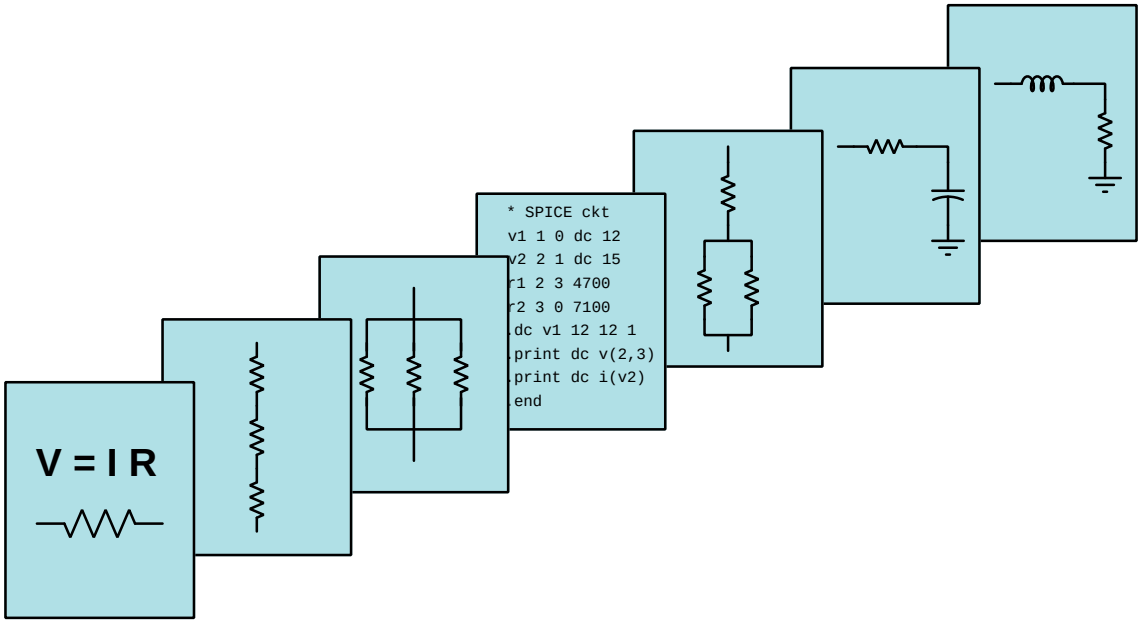
Modular Electronics Learning (ModEL) project



Introduction to Microprocessors

Last update = 23 January 2025

Contents

Chapter 1

Introduction

1.1 Recommendations for students

One of the most revolutionary inventions in the history of humanity is the microprocessor: the central processing unit (CPU) of a computer manufactured small enough to be contained within a single integrated circuit. This miniaturization of digital computer technology paved the way for computers to become part of everyday life, and to be accessible to far more people. Prior to the advent of the microprocessor, computers were necessarily bulky devices, too cumbersome and too expensive to ever be practical for personal use. Now, most people living in industrialized nations carry at least one computer with them wherever they go, and this same technology continues to revolutionize life in pre-industrial regions of the world as well.

Important concepts related to microprocessors include **logic** functions, **numeration** systems, **counters**, **codes**, **decoders**, **multiplexing**, **latches**, **shift registers**, **binary arithmetic**, **bitwise** operations, solid-state **memory**, **finite-state machines**, **functional composition**, **hex dump** displays of memory, the **fetch/execute** cycle, digital **busses**, bus **contention**, **stacks**, **memory maps**, **assembly** language, **machine** language, **masks**, **vectors**, and **interrupts**.

Here are some good questions to ask of yourself while studying this subject:

- How might an experiment be designed and conducted to test the sequence of a microcontroller's execution cycle? What hypothesis (i.e. prediction) might you pose for that experiment, and what result(s) would either support or disprove that hypothesis? Which signals would you need to measure in such a circuit to collect the necessary data?
- What distinguishes microprocessor-based circuits from traditional "hard-wired" logic circuits?
- Why is it important for a microprocessor to be able to "loop" in executing a program?
- What is the function of a microprocessor's *program counter*?
- What are *registers* useful for within a microprocessor?
- What role does *multiplexing* fulfill within a microprocessor?

- How is the *memory map* for a digital computer determined in hardware?
- What is a *bus* in a digital circuit?
- What does it mean for a digital bus to be *directional*, and which bus(es) in a digital computer are typically directional?
- What does it mean for there to be *contention* over a digital bus?
- What functions are necessary for an electronic circuit to be a universal computing device?
- How does a microprocessor coordinate data transfer between external devices such as RAM and ROM?
- What is the function of a "stack" in a microprocessor?
- By what means may a microprocessor perform advanced arithmetic and logic functions that it is not directly designed to execute?
- What are some of the different ways in which programs may be written for a microprocessor?
- How does an 8-bit microprocessor manage memory address values larger than eight bits?
- What does it mean for a microprocessor to be "little-endian" or "big-endian"?
- Why are all microprocessors equipped with *interrupt* capability?
- How does an interrupt function?
- What is an ISR, and what is an interrupt vector?
- Why is a microprocessor equipped with only RAM memory impractical?
- Why is a microprocessor equipped with only ROM memory impractical?
- What does it mean if the I/O for a microprocessor is "memory-mapped"?
- How does *bus notation* help de-clutter schematic diagrams?
- How does assembly language differ from machine language?
- What does a "monitor" program do?

A prominent feature of the Tutorial is the inclusion of step-by-step graphical examples showing the operation of microprocessor systems, some of these examples showing internal register states and others showing digital logic states (high and low). A recommended strategy for maximizing your understanding of these presentations is to read them *actively*: examine each diagram closely to trace where the data originates, where it goes, and what purpose(s) it serves. Don't just read what the text says and trust the diagram is correct; instead, take the necessary time to analyze each diagram thoroughly until you are able to express what it is showing you in your own words.

1.2 Challenging concepts related to microprocessors

The following list cites concepts related to this module's topic that are easily misunderstood, along with suggestions for properly understanding them:

- **Reading hex dumps** – arrays of hexadecimal values showing the contents of a memory module are often referred to as *hex dumps*, and properly interpreting their contents is a skill in itself. Just know that the addresses go in incrementing order as the hex dump is read from left to right, top to bottom, just like reading an English-written document. The "offset" address given in the left-most column simply shows the beginning address value for that row of hex data, each successive column in the hex dump array moving toward the right reflecting an increment in address value from that starting "offset value".

- **Microprocessor fetch/execute cycle** – the sheer number of steps required for a microprocessor to perform even the simplest of tasks is sufficiently daunting to prompt some new students to surrender on first sight. Rest assured, though, that close analysis of a practical example completely illustrated will yield understanding. To this end, the step-by-step example shown in the "Putting it all together – the processor" section of the Tutorial has been given for the reader's understanding. *Follow each of these steps* and read the given explanations, and the understanding will come! In this example the microprocessor reads instructions and data stored in memory to add two numbers together and place the sum in memory, the internal microprocessor diagram annotated to show the flow of data between logical elements at each step.

- **Memory maps** – microprocessors typically access different types of storage memory and I/O lines via specific addresses and address ranges. A *memory map* is a block-shaped illustration dividing the address ranges according to which device is being accessed. Again, the Tutorial goes into painstaking detail showing how this works for a simple computer (in the "A simple computer example" section), where close and patient reading will yield understanding. Follow along with the annotated schematic diagram to see how each set of address values enables one device at a time while disabling the others from access to the data bus.

1.3 Recommendations for instructors

This section lists realistic student learning outcomes supported by the content of the module as well as suggested means of assessing (measuring) student learning. The outcomes state what learners should be able to do, and the assessments are specific challenges to prove students have learned.

- **Outcome** – Demonstrate effective technical reading and writing

  Assessment – Students present their outlines of this module's instructional chapters (e.g. Case Tutorial, Tutorial, Historical References, etc.) ideally as an entry to a larger Journal document chronicling their learning. These outlines should exhibit good-faith effort at summarizing major concepts explained in the text.

  Assessment – Students add their own explanations to the annotated step-by-step examples shown in the Tutorial chapter.

- **Outcome** – Apply the concept of memory maps to simple computer circuits

  Assessment – sketch the memory map for a given schematic diagram based on address line connections; e.g. pose problems in the form of the "Memory map determination" Quantitative Reasoning question.

- **Outcome** – Write a simple assembly language program

  Assessment – Use a microprocessor simulator (application or interactive website) to write and test simple assembly-language programs. the legacy Motorola 6502 microprocessor is well-respected for its simple instruction set, so any simulator designed to emulate a 6502 is a good starting point.

- **Outcome** – Independent research

  Assessment – Read and summarize in your own words reliable source documents on the subject of microprocessors. Recommended readings include legacy microprocessor datasheets and patents, as the earliest examples of these device tend to be much less complex (and therefore easier to understand) than modern microprocessors with all their advanced features and architecture.

Chapter 2

Tutorial

2.1 Digital building-blocks

This section will review some of the conceptual and physical elements of digital systems, all of which are necessary to the function of electronic computers and specifically to the "central processing unit" constituting the heart of any computer. A "microprocessor" is simply a central processing unit (CPU) built on a single wafer of silicon and packaged as an integrated circuit. It is the fundamental task of any microprocessor to read and follow pre-written instructions, to implement logical and arithmetic functions, and to direct the flow of digital information between different sources and destinations. This Tutorial will explain how this works.

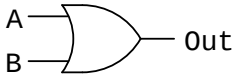
Each subsection briefly reviews the digital concept or component, and describes how it plays a role in digital computing. Since microprocessors utilize several different concepts and circuit functions within their internal design, it is imperative you understand how each of these works in order to comprehend the over-all function of a microprocessor.

2.1.1 Logic functions and Boolean algebra

A *logic function* is to discrete (on/off) digital signals as a mathematical function (e.g. addition, subtraction, multiplication, division) is to numbers: a relationship between one or more inputs to a output.

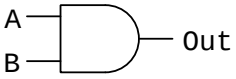
Basic logic functions and their respective symbols are shown below, in each case A and B representing inputs to the function and Out representing its output:

OR function



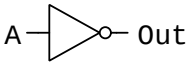
A	B	Out
0	0	0
0	1	1
1	0	1
1	1	1

AND function



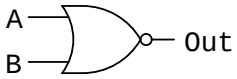
A	B	Out
0	0	0
0	1	0
1	0	0
1	1	1

NOT function



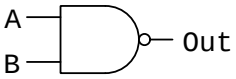
A	Out
0	1
1	0

NOR function



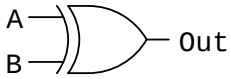
A	B	Out
0	0	1
0	1	0
1	0	0
1	1	0

NAND function



A	B	Out
0	0	1
0	1	1
1	0	1
1	1	0

Exclusive-OR function

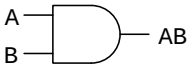


A	B	Out
0	0	0
0	1	1
1	0	1
1	1	0

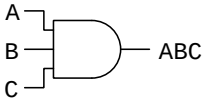
While logic *functions* are abstract just like arithmetic operations, they find physical application in both relay logic and semiconductor "gate" logic circuits. They are the fundamental "building-blocks" of all digital functions.

A special notation and mathematical system called *Boolean algebra* allows us to represent logical functions as mathematical equations terms. Here we see examples of several types of logic functions notated with Boolean variables at their inputs and Boolean expressions at their outputs:

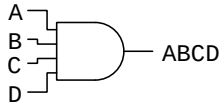
2-input AND function



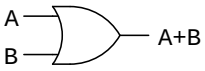
3-input AND function



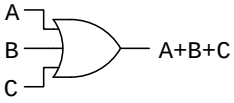
4-input AND function



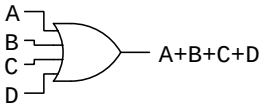
2-input OR function



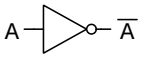
3-input OR function



4-input OR function

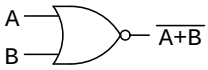


NOT function



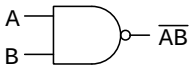
A	$\overline{A}$
0	1
1	0

NOR function



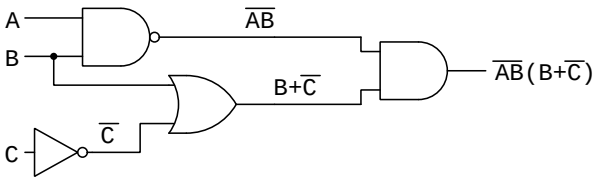
A	B	$\overline{A+B}$
0	0	1
0	1	0
1	0	0
1	1	0

NAND function



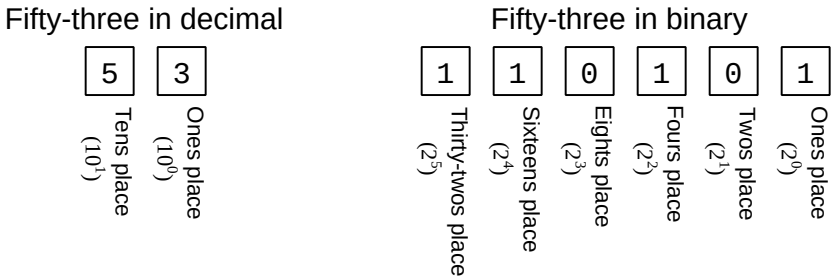
A	B	$\overline{AB}$
0	0	1
0	1	1
1	0	1
1	1	0

Combinations of logic functions (called *combinational logic*) are easily expressed using Boolean notation, as we see in the following example:



2.1.2 Binary numeration

Each discrete signal processed by a logic circuit may have its own meaning (e.g. door open/closed, switch pressed/unpressed, etc.), or it may be part of a larger multi-signal aggregate. When we combine multiple on/off signals into a larger "word" we have opportunity to represent more complex concepts than just true/false conditions. For example, we may regard a collection of discrete states as being individual *bits* within a *binary number*, as illustrated below:



Fifty-three = $(5 \times 10^1) + (3 \times 10^0) = 50 + 3$

Fifty-three = $(1 \times 2^5) + (1 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)$
Fifty-three = $(1 \times 32) + (1 \times 16) + (0 \times 8) + (1 \times 4) + (0 \times 2) + (1 \times 1)$

The ability to represent numerical quantities using nothing more than discrete (on/off) circuits is what makes mathematical calculations possible within a digital computer comprised of two-state (bistable) transistor circuits.

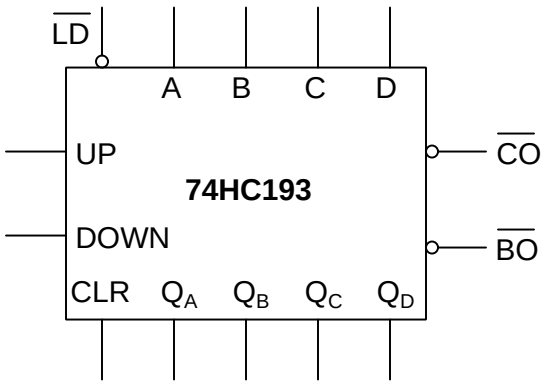
For any binary word, the number of unique combinations of 1 and 0 states is equal to two to the power of the number of bits (2^b). For example, an 8-bit binary word has a range from `00000000` to `11111111` with 256 possible combinations ($2^8 = 256$). This is important in digital computer design because the number of bits within a word are generally limited by physical counts of wires, gates, and ultimately individual transistors. This means the numerical range of any computer, the amount of data it can store and retrieve, etc. will be a function of how many bits it processes. Early digital computers had fairly short word widths, with 8-bit words common among the first personal computers. At the time of this writing (2020) most personal computers handle digital data in 64-bit words.

Representing large binary words can be problematic for human beings. For example, it is often difficult to even tell the difference between two large binary words (e.g. can you tell what's different between `11001010001101001101010001100101` and `11001010001101101101010001100101`?). For this reason we commonly denote binary words using *hexadecimal* notation, where every four bits condenses into a single character (`0` through `F`). For example, the two 32-bit binary numbers previously in this paragraph could be written as `CA34D465` and `CA36D465` in hexadecimal, respectively.

2.1.3 Counters

A *counter* is a digital logic circuit comprised of bistable latching circuits called *flip-flops*, the purpose of which being to generate sequential binary number outputs at the command of a periodically pulsing signal called a *clock*. The simplest types of counters merely count in an upward (incrementing) direction, while others are capable of incrementing and decrementing (i.e. counting up and down). Most counters also provide a means to *reset* the count value to zero, and many also provide a means to *preset* or *load* the counter with some non-zero starting value.

The model 74HC193 up/down counter is a typical integrated-circuit (IC) unit, exhibiting all of these functions:



Direction of count is controlled by applying the clock pulse to either the "UP" or "DOWN" input. To pre-load this counter with a starting value, you would apply the desired binary states to upper inputs and then activate the "load" input ($\overline{LD}$) by making it low. That data would then appear at the Q outputs.

As with any binary count sequence, the maximum number of unique "count" values representable by any counter circuit is 2^n, where n is the number of bits. For the model 74HC193 counter, sixteen different count states are possible, numbered zero (`0000`) through fifteen (`1111`).

Counters serve a very important purpose within microprocessors, and that is to index sequentially-stored instructions in the computer's memory. This *program counter* (often abbreviated PC) internal to the microprocessor instructs the memory device(s) which memory cell(s) to read and write, usually in incrementing order. On occasion the program may call for the PC to "jump" or "branch" to a non-sequential memory address, which is an example of the program counter being "loaded" with a new starting number value.

2.1.4 Codes

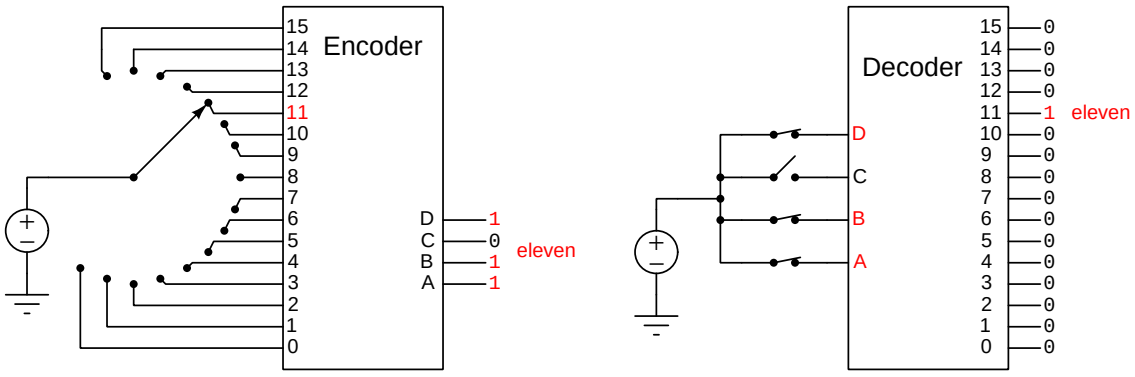
Multi-bit words are useful for representing more than just numbers. A 7-bit digital word, for example, has 128 unique combinations (`0000000` through `1111111`), which is more than enough to represent the entire English alphabet (upper- and lower-case) plus Arabic numerals (0 through 9) plus all the standard symbols and punctuation marks found on a typewriter or computer keyboard. A standardized code called *ASCII* does just this: assigning a unique 7-bit code to each of these printable characters as well as to some commands useful to printing machines.

Many other such codes exist, and may be invented, by assigning unique digital word combinations to specific things. *Unicode* is a different digital code intended to represent written language characters. *Gray code* is a code used to represent discrete positions within the range of some object's motion. Complex standards such as *bitmap* or *bmp* encode graphical images in the form of long "files" consisting of discrete (on/off) states, as does the *wav* format for encoding audio information. All that is required to form a digital code is an agreement between parties (and/or between computers) as to the meaning and format of all the on/off states.

Codes are particularly important in digital computer circuits, not just for representing external information such as language characters, color, or sound, but also for representing individual instructions the computer must follow. Every microprocessor has an *instruction set* consisting of a set of codes (called "operation codes" or *opcodes*) where each one represents a particular action the circuit is supposed to take. These instruction sets are entirely the invention of the microprocessor designer(s), and vary widely from one model to another.

2.1.5 Decoders

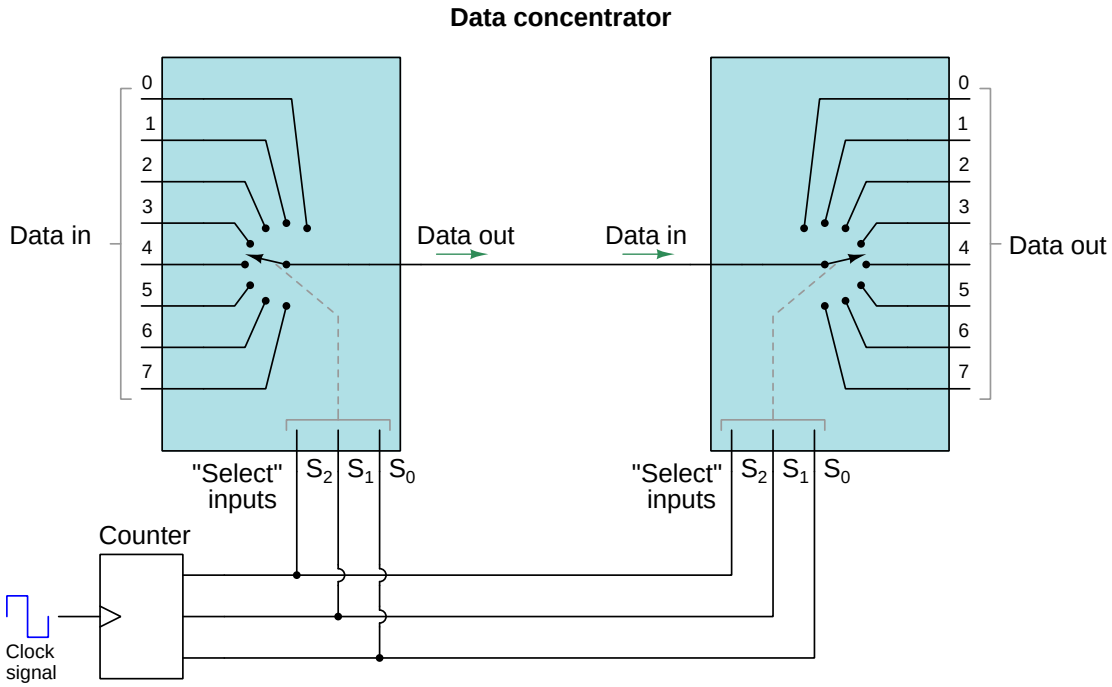
Any digital circuit that takes a 1-of-n input and converts that into a multi-bit word is called an *encoder*. A *decoder* does just the opposite, taking a digital word and activating one output or one function for each of the unique combinations of that word:



Decoders find widespread use in digital computer circuits for translating encoded instructions (called "opcodes") into definite actions. For example, a microprocessor using four bits to represent each opcode is capable of taking sixteen (2^4) different actions based on that code.

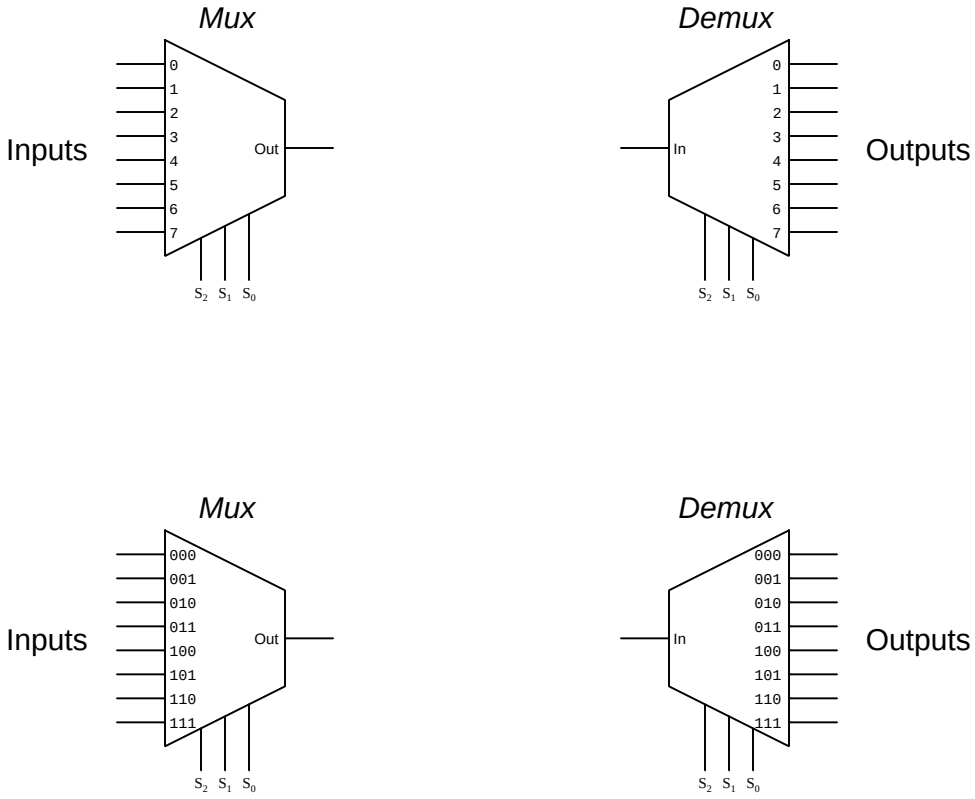
2.1.6 Multiplexers and Demultiplexers

Multiplexers and demultiplexers ("muxes" and "demuxes") function as *steering* networks for digital signals. Take for example the following mux/demux pair used to funnel one of eight discrete signals along a single line:



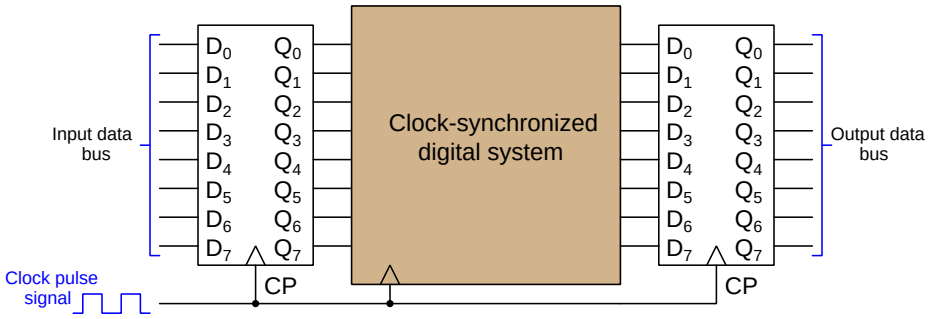
Both types of devices are critically important to computer circuits, including the internal circuitry of microprocessors, for their ability to direct digital words from different sources and to different destinations. If a computer needs to receive data from one device at one time, and from a different device at some other time, but using the same data pathway for both sources, the solution would be to multiplex those data sources. The computer would control which source was selected by the digital word sent to the mux's "select" input lines.

Note that both multiplexers and demultiplexers are often represented by trapezoidal symbols rather than rectangular box symbols, as shown below. Selector values may be written in decimal or in binary:



2.1.7 Latches and shift registers

A *latch* is a digital circuit that samples data and passes it through during some conditions, but blocks new input and holds old data for other conditions. These are useful as "gateways" for digital data into and/or out of digital circuits, permitting data to pass through at some moments but not at others. Microprocessors use many latches internally for such purposes, with the actions of the latches controlled by a *clock pulse*. The illustration below shows a pair of edge-triggered latch circuits controlling the flow of digital information into and out of some digital system:

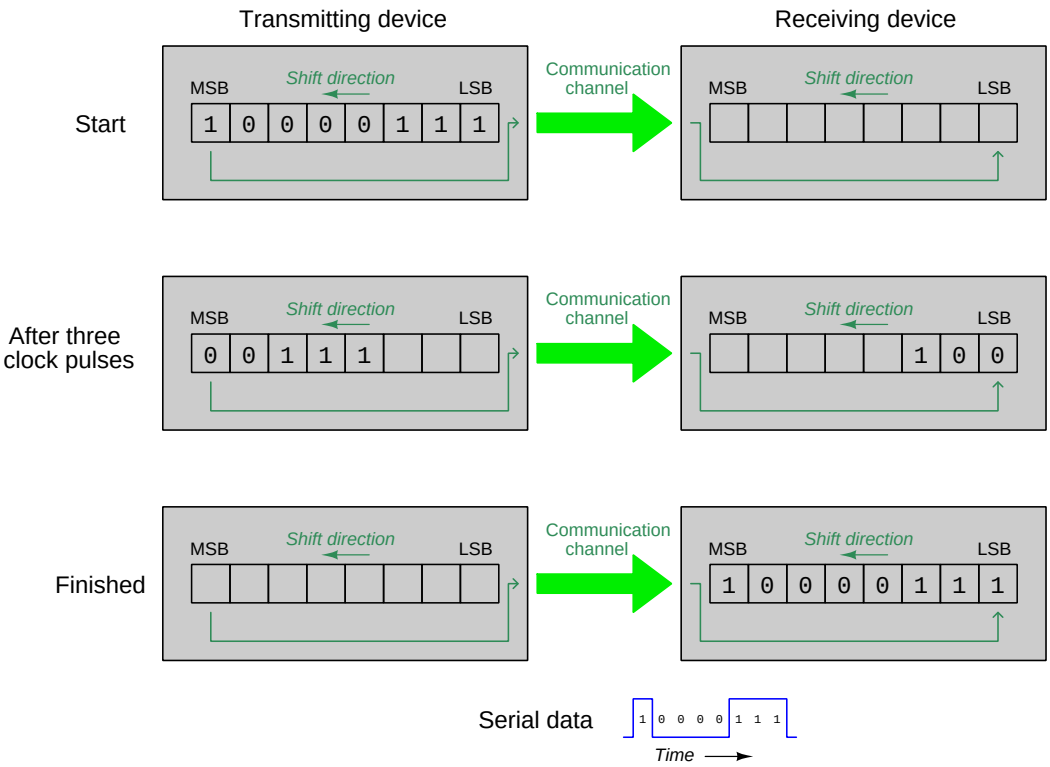


When latches or flip-flops are used to sequence the flow of digital data into or out of logic, that logic is often referred to as *registered*. Each of the eight-channel D-type flip-flops shown in the prior illustration may be thought of as a *parallel-in, parallel-out shift register* clocking the data into and out of the clock-synchronized logic block.

Non-registered logic functions produce new output states at whatever speed their logic gates will muster, limited chiefly by propagation time delays within each gate. However, with complex systems having many output lines and many gates, it is often the case that those different output lines do not update synchronously with each other[1], so the use of parallel shift registers to delay the input and output of digital data gives complex logic networks time for all their outputs to settle into stable states before sending that data on to another part of the digital system.

[1] A binary *ripple counter* circuit is an excellent example of this type of circuit.

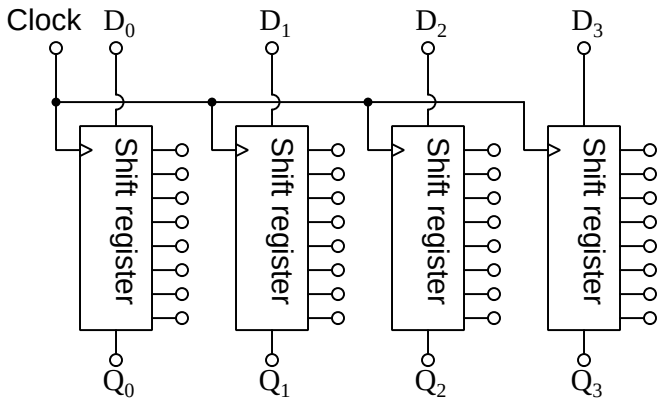
Shift registers take digital information and move the bits from one latch to another at the control of a clock pulse. A common application of this is in serial communication systems, where a digital word must be communicated one bit at a time over some channel and then reassembled back into its original form:



Not only does this find widespread use in digital communications, but also within the internal workings of digital computers. Shifting the bits of a binary number from one place to the next has the effect of either multiplying or dividing that number by two. For example, taking the binary value of six (`00110`) and shifting the bits to the left turns it into twelve (`01100`); shifting once to the right turns six into three (`00011`). Serial-type shift registers, therefore, provide a simple means of doubling or halving binary quantities.

A special type of shift register important to microprocessor operation is called a *stack*. This is a set of registers connected in such a way that a digital word applied to the input terminals will be received at the next clock pulse, and with successive clock pulses that word will be either shifted to a deeper level within the array or be shifted in reverse order where it will return to the beginning.

The following diagram shows a four-bit stack with eight levels. With each applied clock pulse, the data given to this stack (data bits D_0 through D_3) become shifted further down the stack until eventually they appear at the Q outputs:



If we make these shift registers bidirectional, we will have a means of controlling which direction data moves on the stack. Taking in new data through the D lines and shifting all other data toward Q is called *pushing* data onto the stack. Reversing the shift direction to bring data words back toward the top is called *popping* data off of the stack.

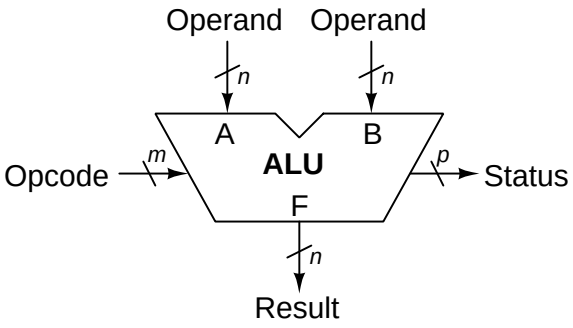
Stack registers are extremely important to the operation of a microprocessor because they allow the microprocessor to take on new tasks while "remembering" the incomplete status of older tasks. Stacks are analogous to a pile (stack) of paper notes. If a person is reading a book and they suddenly get told to turn to a different chapter to read a passage there, they may write the current page number on a note and "push" that note to the top of the stack so they won't forget it while turning to the new passage. After reading the new passage, the person retrieves their note from the top of the stack (i.e. "popping" it off the stack) and references it to return to the page where they left off. This may occur more than once, and the stack will "remember" not only the page numbers but also keep everything in the right order as the person eventually returns to their original place in the book.

The deeper a microprocessor's stack capacity, the more capacity the computer will have for interruptions and the better it will be able to divert computational resources toward time-sensitive tasks. Some small-scale and special-purpose microprocessors use internal stacks comprised of dedicated shift registers, while others reserve special regions of main memory (RAM) for use as a stack[2]. The latter strategy is most common, because with this the size of the stack is limited only by the amount of RAM available and the microprocessor's ability to address that RAM. Stacks built from dedicated shift registers inside the microprocessor are necessarily limited in size.

[2]In such designs, the microprocessor contains a binary counter circuit called the *stack pointer* which holds the address of the memory cell deemed to be at the top of the stack.

2.1.8 Arithmetic logic units

Arithmetic Logic Units, or *ALUs*, are complex digital circuits designed to perform certain mathematical and logical operations on digital words. They have their own symbols and nomenclature as shown here:

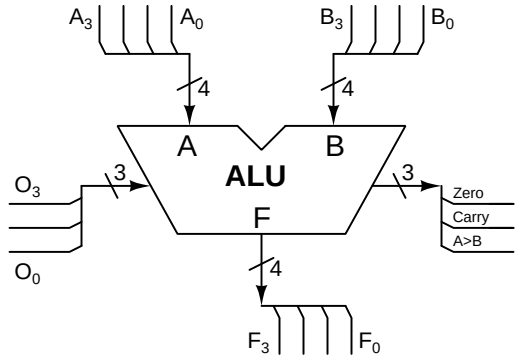


The *operands* are multi-bit digital words that the ALU acts upon. The *opcode* is also a digital word, instructing the ALU which logical or math function to apply to the operand(s). An ALU outputs the result of its operation, as well as bits representing the status of that operation (e.g. whether or not the result was zero, whether or not the result was negative, etc.).

An illustration of a very simple four-bit ALU appears below with an eight-opcode instruction set:

Opcode list

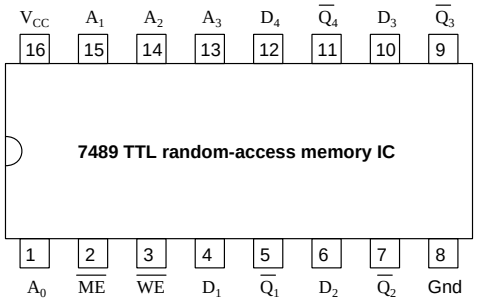
0 0 0 = Clear (F=0000)
0 0 1 = Add (A+B)
0 1 0 = Increment A
0 1 1 = AND (A • B)
1 0 0 = XOR (A ⊕ B)
1 0 1 = Rotate right A
1 1 0 = Rotate left A
1 1 1 = Set (F=1111)



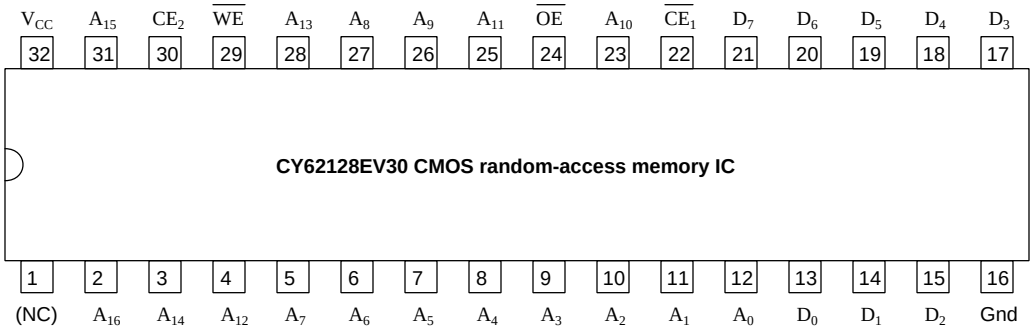
Every microprocessor contains at least one ALU. If the microprocessor is considered the center of a digital computer, then the ALU is the center of the microprocessor. Much of a microprocessor's internal circuitry exists simply to route information to and from the ALU. For example, a microprocessor uses shift registers to hold each operand prior to processing by the ALU; a microprocessor uses mux and demux circuits to steer different sources of data to the ALU's operand inputs as well as direct the ALU's output to different destinations.

2.1.9 Memory

Memory arrays built from semiconductor logic typically store digital words called *data* inside of latch circuits, with each latch accessed by another digital word called the *address*. To use an analogy, the address word sent to a memory device is equivalent to the address given to a postal worker who must find the appropriate mailbox, while the data word is information either placed in the mailbox (written) or retrieved from it (read). An example of a legacy TTL memory IC is shown below:



On this particular IC the memory lines are labeled A while the data input lines (for writing) are labeled D. Separate data output lines labeled Q provide a means to read the memory. Most modern memory ICs have *bidirectional* data lines able to both read and write data depending on the circuit's mode, like the following example of a model CY62128EV30 memory:

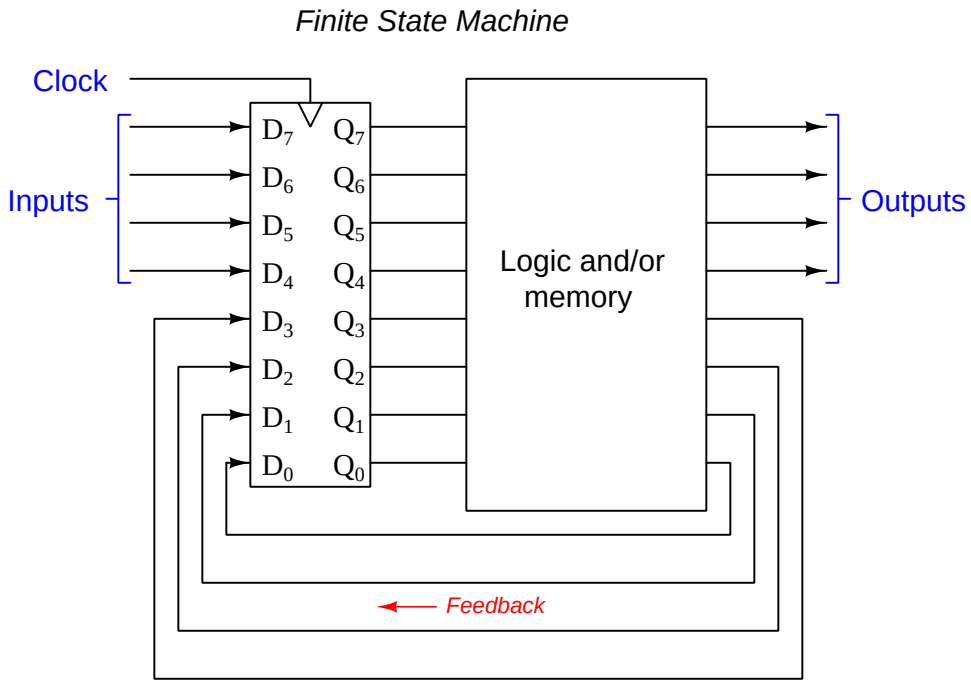


Most modern digital memory technologies are "random access" which means addresses may be selected in arbitrary order[3]. However, the term *RAM* (Random Access Memory) has come to define memory that is both random-access and *volatile*. Memory that is able to retain its data after losing electrical power is called *nonvolatile* and is generally known by the term *ROM* (Read-Only Memory) although this is often a misnomer as well. Some ROM memory types may be written to as well as being read.

[3] By contrast, a *sequential access* memory can only read or write information in one order. *Tape* is a good example of a sequential access technology because one cannot "jump" arbitrarily to any point along the tape, but instead must wait for the tape to come to the correct location for reading or writing.

2.1.10 Finite state machines

A *finite state machine* is a digital circuit using combinational logic and/or a memory array combined with signal feedback to create a system with a certain number of states, the transitions between states determined by clock pulses and input conditions:



Unlike a combinational logic circuit where all outputs are strictly a function of input conditions at any given time, for a finite state machine the outputs depend on both input conditions *and* the internal state of the circuit. That internal state is defined by the "feedback" bits of the circuit. For the finite state machine shown above, there are sixteen possible states based on the four feedback lines ($2^4 = 16$).

These circuits are useful for generating specific *sequences* which may be used to coordinate different events in a digital system over time. An application for finite state machines is inside microprocessors, for sequencing such events as the enabling of latches, the selection control of multiplexers and demultiplexers, and other internal actions which must transpire in particular orders.

2.2 Putting it all together – the processor

What defines a *computer* as such? What capabilities, at minimum, must a system possess in order to be deemed a computer? This is a difficult question to answer in definitive terms[4], but perhaps a more practical approach for our purposes would be to ask the question, *What capabilities have proven useful for a computer to possess?* The following list shows some of these capabilities:

- A computer should be able to execute tasks in an order prescribed by instructions – i.e. it needs to be able to follow directions
- A computer should be able to repeat sequences on demand – i.e. it needs to be able to *loop*
- A computer should be able to execute different sequences of tasks based on changing conditions – i.e. it needs to be able to conditionally *jump* or *branch*
- A computer should be able to direct information to and from different sources and destinations – i.e. it needs to be able to *direct* the flow of data
- A computer should be able to store and recall information – i.e. it needs to be able to *remember* data
- A computer should be able to implement basic mathematical and logical instructions – i.e. it needs to be able to apply *arithmetic* and *logic* to data

Most of these capabilities reside within the *processor* section of a digital computer. Early digital computers used processors made from discrete components (first relays and vacuum tubes, later transistors), but as semiconductor manufacturing techniques advanced it became possible to construct an entire processor on a single wafer of silicon, and thus the *microprocessor* was born.

An interesting and counter-intuitive fact of digital computing is that the bare-minimum functionality of a microprocessor need not be sophisticated to form a computer capable of amazing feats. For example, the arithmetical capability of the earliest microprocessors and many of their successor designs was limited to elementary addition and subtraction. This does not mean, however, that a computer built around such a microprocessor could only add and subtract! We may *program* simple microprocessors to perform advanced operations by compounding the functions they do offer.

Consider multiplication which is really just compounded addition, or division which is really just compounded subtraction. In other words, it is possible to *synthesize*[5] complex mathematical functions from more elementary functions if we have the ability to *loop* and *branch* and *remember* intermediate results. In computer science this is known as *function composition*: composing advanced functions by combining primitive functions.

[4] For those interested in a more complete exploration of this concept, research the definition of a *Turing machine*, *Turing completeness*, and the *Church-Turing Thesis*. Alan Turing was a British mathematician who developed the concept of a *Turing machine* based on readable and writable tape memory. Alonzo Church was an American mathematician who developed a mathematical system called lambda calculus (λ calculus) to describe computable functions. The Church-Turing Thesis and its related concepts revolve around the question of what is necessary to compute algorithmic functions – essentially, what is necessary to make a universal computing machine.

[5] This is not unlike the universality of NAND and NOR logic functions: with enough NAND gates or NOR gates we may construct any logical function at all!

This is relatively easily demonstrated in the case of multiplication as an example of compounded addition. Suppose we wished to have a computer multiply any two whole numbers A and B together to compute their product C using a microprocessor limited to addition, subtraction, looping, branching, and memory read/write capabilities. A simple set of instructions for multiplication is shown here in *pseudocode*[6]:

```
  Store first number into memory location A
  Store second number into memory location B
  Clear memory location C

Loop:
  Add B to C and store result in C
  Decrement A
  Check to see if A > 0
    If so, go to Loop and repeat
    If not, present product in memory location C
  (End)
```

Likewise, consider a pseudocode program that divides any two whole numbers A by B to compute quotient C:

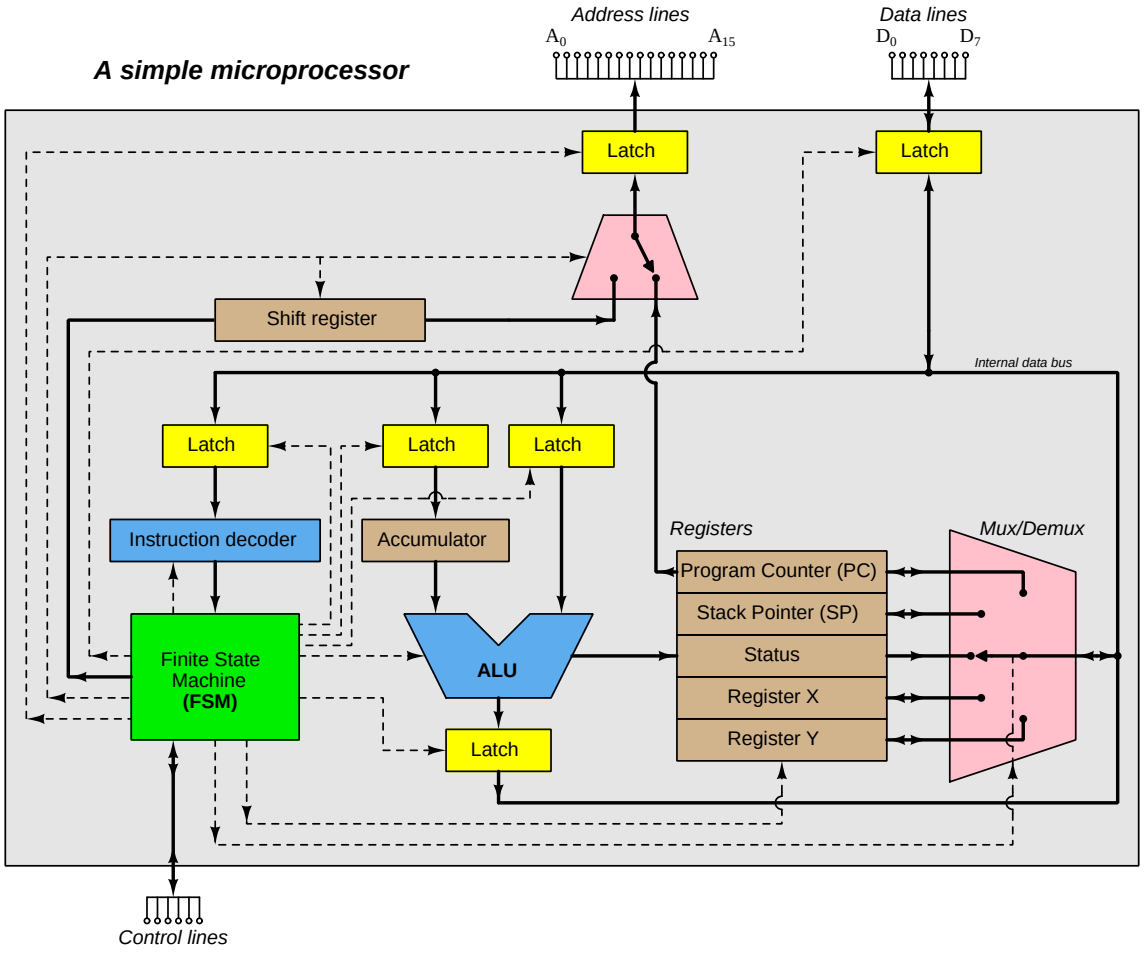
```
  Store first number into memory location A
  Store second number into memory location B
  Clear memory location C

Loop:
  Subtract B from A and store result in A
  Increment C
  Check to see if A > B
    If so, go to Loop and repeat
    If not, present quotient in memory location C and remainder in A
  (End)
```

These are elementary examples, of course, but they prove the point: by combining primitive functions with the ability to conditionally loop and the ability to store results in memory, we see that a microprocessor may be used to perform functions beyond its immediate capabilities.

[6] Pseudocode is not a formal computer programming language, but merely a set of brief statements written in regular words describing to any human reader what a real program should do.

Every microprocessor uses a unique arrangement of digital "building-blocks" such as counters, latches, multiplexers, etc. to form a functional processing unit, and these are typically so complex that we rarely if ever see a *schematic* diagram showing all the components and all the logic lines. Instead, it is customary to show the "architecture" of a microprocessor as a block diagram. Below is such an example:



Solid lines indicate individual conductors or collections called *busses*. In this case, we see three such busses for this microprocessor: *address*, *data*, and *control*. These busses connect to external devices such as memory banks and input/output (I/O) devices. Much of a microprocessor's tasks end up being memory read and write operations, where the microprocessor outputs digital logic states on its address lines and either sends or receives digital logic states on its data lines. The number of data lines typically defines the "width" of the microprocessor, and so in this case it is an 8-bit microprocessor. The microprocessor's "control" bus consists of lines for the clock signal, reset input, interrupt input(s), read/write and select outputs to control memory devices and I/O devices connected to the address and data busses, etc.

Note how the Finite State Machine (FSM) block of the microprocessor has dashed lines outputting to nearly every other block, for the purpose of coordinating the timing and/or functions of each. A single instruction read from memory and input to the microprocessor usually triggers a *sequence* of events inside of it – latches receiving and holding data, multiplexer/demultiplexer channels changing, and so forth – which must occur in the proper order for everything to work. Multiple cycles of the clock pulse signal must occur for most if not all of these events.

Even with the simplification of a block diagram, the internal workings of a microprocessor can be daunting to behold. A practical example will serve to illustrate how these components work together to execute a simple program. For our example we will consider a program that reads two numbers from addresses in an external RAM memory and adds them together. The pseudocode for this program is as follows:

```
(Program begins at RAM address 0x0200)
Read number stored at RAM address 0x0210 into the Accumulator
Add to that the number read from RAM address 0x0211
Write the result to RAM address 0x0212
```

This program consists of three instructions for the processor to follow: a *read* instruction, an *add* instruction, and a *write* instruction. Each of these instructions will consist of sub-steps, as we will see. The microprocessor, being nothing more than a collection of digital circuits, only understands digital words, not English words. In order to give it these instructions, we must *encode* each instruction type as a binary number. This is not unlike the ASCII code used to represent English characters: an arbitrary set of 7-bit binary words, each one representing a unique printed character. The designers of the microprocessor get to assign numerical codes to each of the microprocessor's instructions, and they publish a list of these instructions and their corresponding codes in the microprocessor's datasheet or manual. For our hypothetical processor, we will arbitrarily assign the code `7C` to the "read" instruction, the code `29` to the "add" instruction, and the code `B5` to the "write" instruction.

The reader is urged to closely examine the step-by-step analysis of the microprocessor's actions unfolding over the next several pages. There is a lot happening here, and it is easy to overlook small but important details. At each step the microprocessor block diagram will be annotated with red lines denoting the path of data movement between blocks. The RAM memory unit's hex dump will also be illuminated in red at each step showing the address and data values being either read from or written to that memory unit. Make your own annotations if necessary, and if you are keeping a journal[7] of your learning, articulate each of the microprocessor's steps in your own words.

[7] Journaling is a powerful tool for active learning, as it affords the learner the opportunity to express their learning in their own words, to pose questions and points of confusion along the way, and to more easily connect concepts that at first encounter may seem unrelated.

Each of these instructions will function as follows:

- **Read** (`7C`) is followed by two more bytes in memory declaring the 16-bit address where the numerical value is stored, in "little-endian" order (lowest-order byte first, highest-order byte next), and store this value in the Accumulator register.
- **Add** (`29`) is followed by two more bytes in memory declaring the 16-bit address where the next numerical value is stored, also in "little-endian" order (lowest-order byte first, highest-order byte next). This number will be added to what is already stored in the Accumulator, and the result will be placed in the Accumulator.
- **Write** (`B5`) is followed by two more bytes in memory declaring the 16-bit address where the sum will be written, also in "little-endian" order.

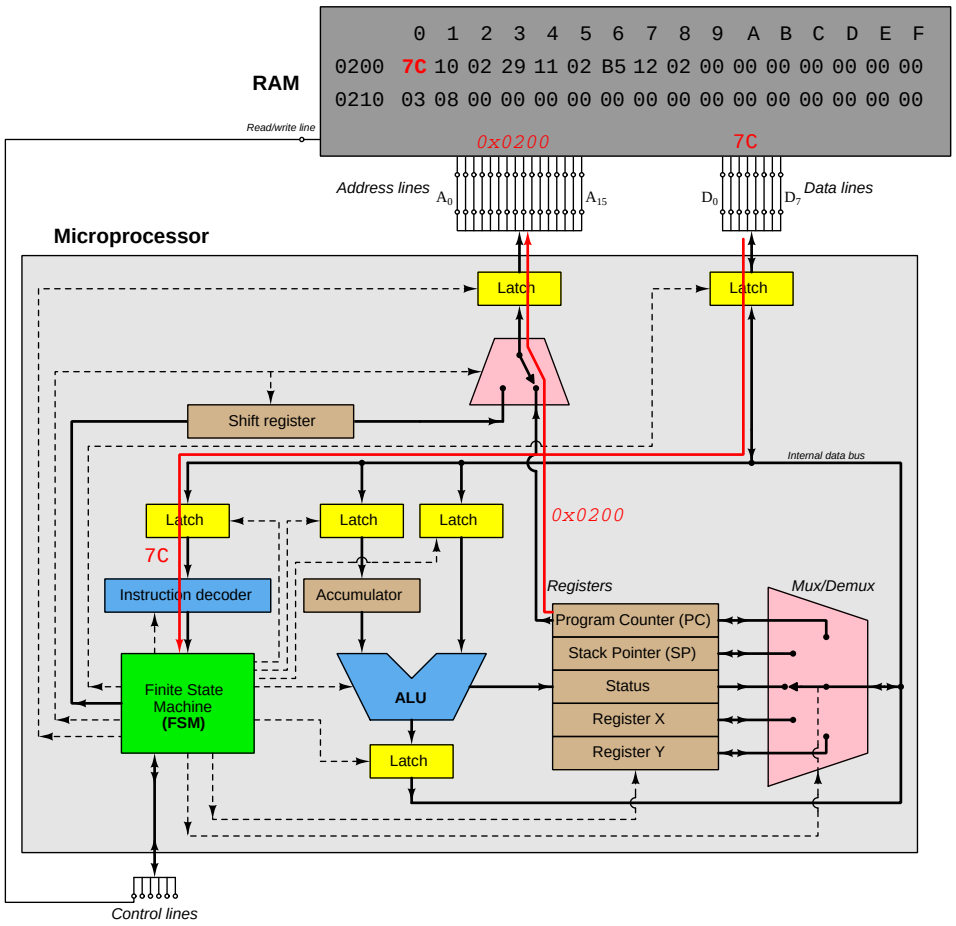
For the purposes of this illustration we will assume the number stored at address `0x0210` will be *three* and that the number stored at address `0x0211` will be *eight*. At the conclusion of the last instruction we should find a sum of *eleven* written to address `0x0212` in RAM. A "hex dump" of the RAM starting at `0x0200` will initially look like this (assuming all other addresses have been cleared and contain zeros):

```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0200 7C 10 02 29 11 02 B5 12 02 00 00 00 00 00 00 00
0210 03 08 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

The microprocessor's task will be to *fetch* each instruction from RAM and *execute* it. This basic fetch/execute cycle repeats as long as the program allows.

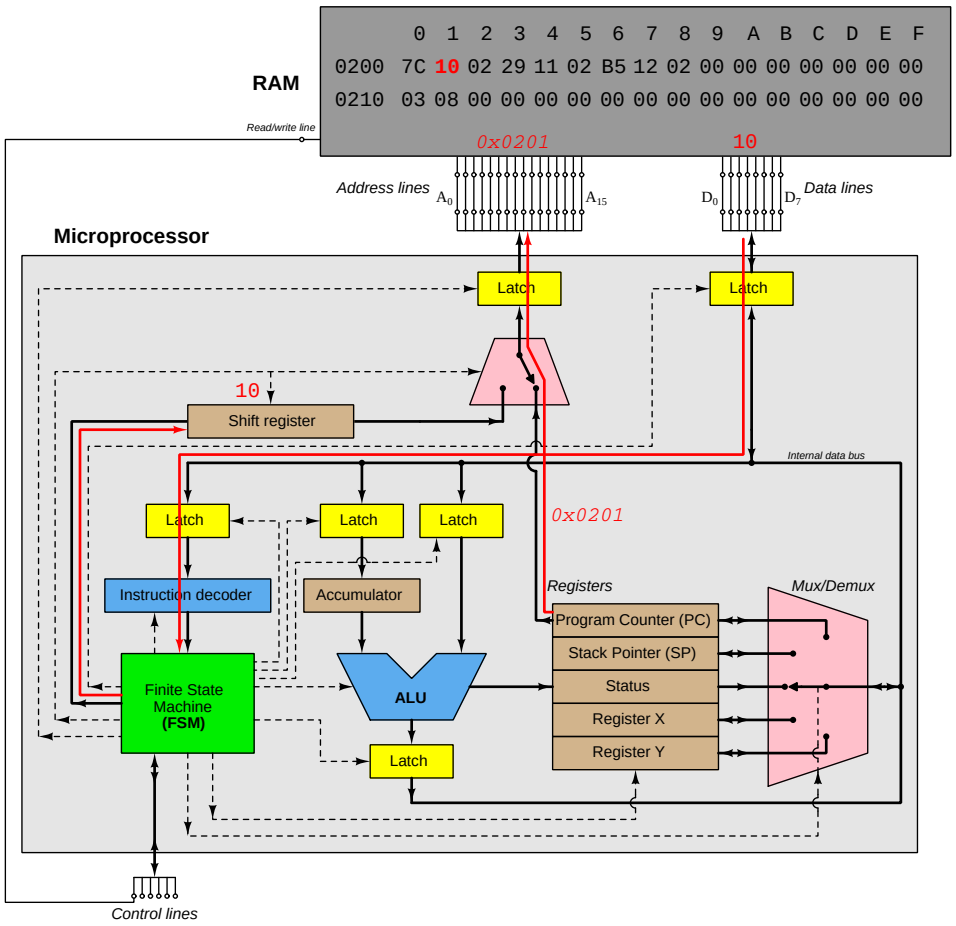
To the microprocessor, these digital words stored in RAM constitute what is called *machine code*: information written in the microprocessor's native language, ready to be decoded and sequenced. There is no standardization of machine code for different microprocessors, unlike higher-level programming languages: machine-code programming is always specific to that model of microprocessor. In the earliest days of microprocessors this was the *only* available form of programming, and it was very tedious. As we shall see later, there are ways to write microprocessor programs using symbols easier to interpret by human beings.

Fetching instruction `7C` from memory and decoding to initiate the FSM's sequence:



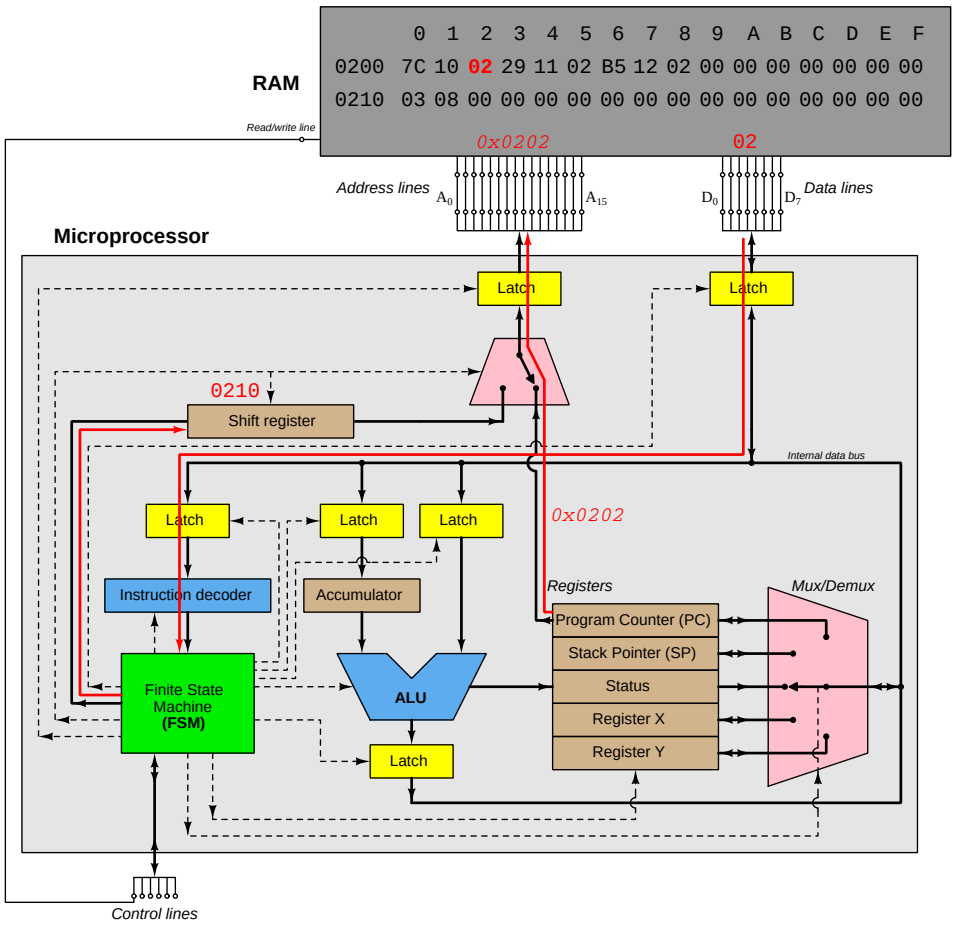
Recall that for this microprocessor the `7C` instruction commands it to read a value stored in memory, at an address which itself is stored in the next two addresses in memory. The first action taken by the FSM is therefore to increment the program counter (PC) to read the low-order byte of the address for the number which will be sent to the Accumulator register.

Reading `10` from memory as the first byte of a 16-bit address:



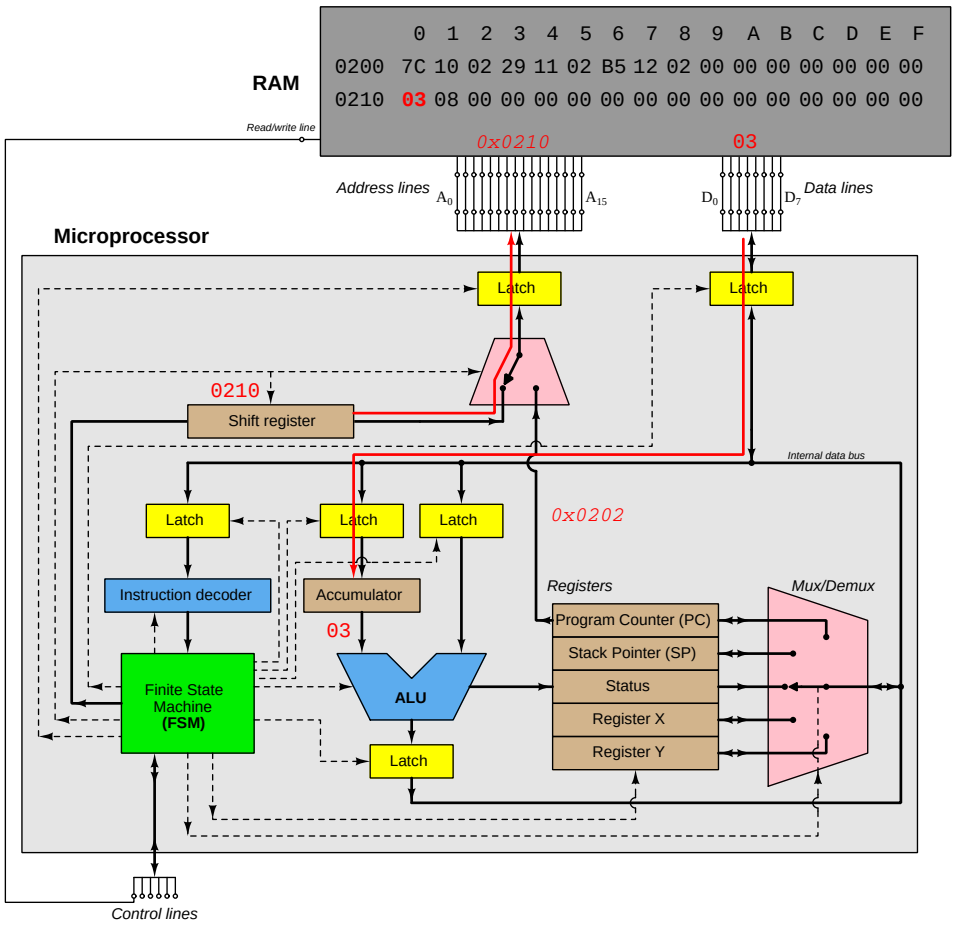
This first byte is routed to the shift register by the FSM. Next, the FSM increments the program counter again to read the next byte, in fulfillment of the `7C` instruction.

Reading `02` from memory as the second byte of a 16-bit address:



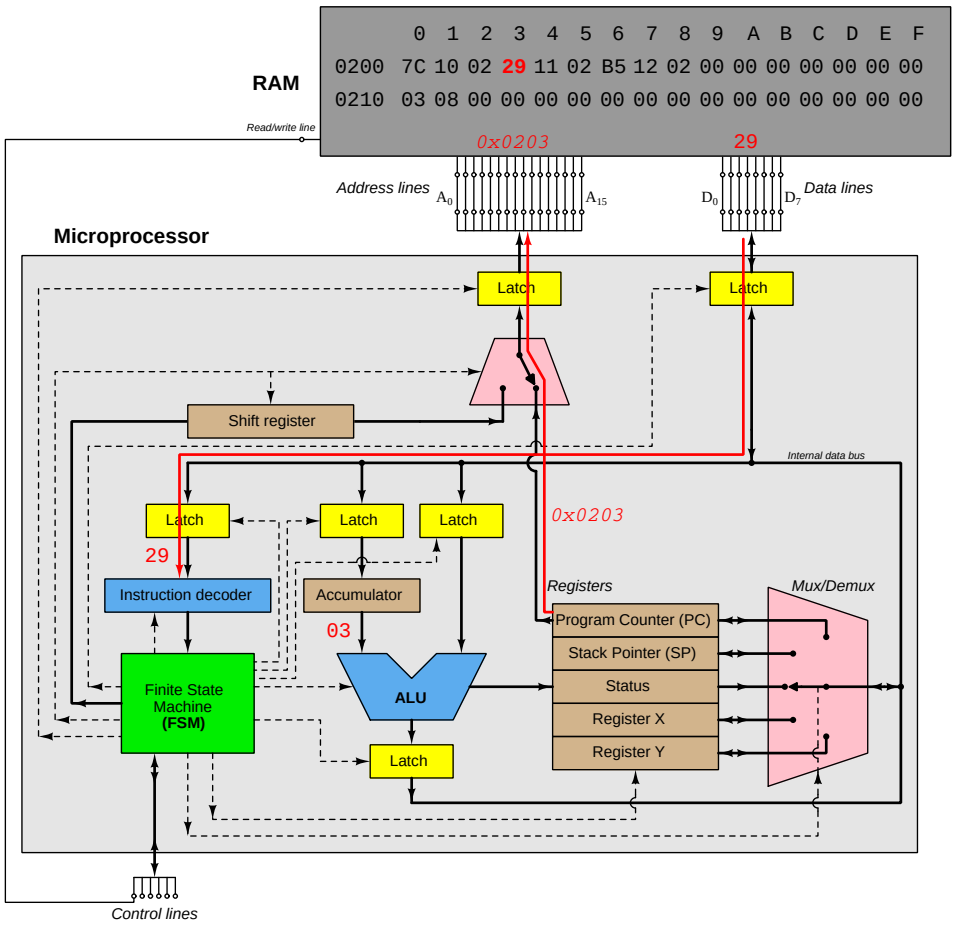
Now that both bytes of the 16-bit address are in the shift register, the FSM will direct this to be the next address (instead of the program counter) and read the data stored there in fulfillment of the `7C` instruction.

Reading from memory address `0x0210` the first number to be added (three):



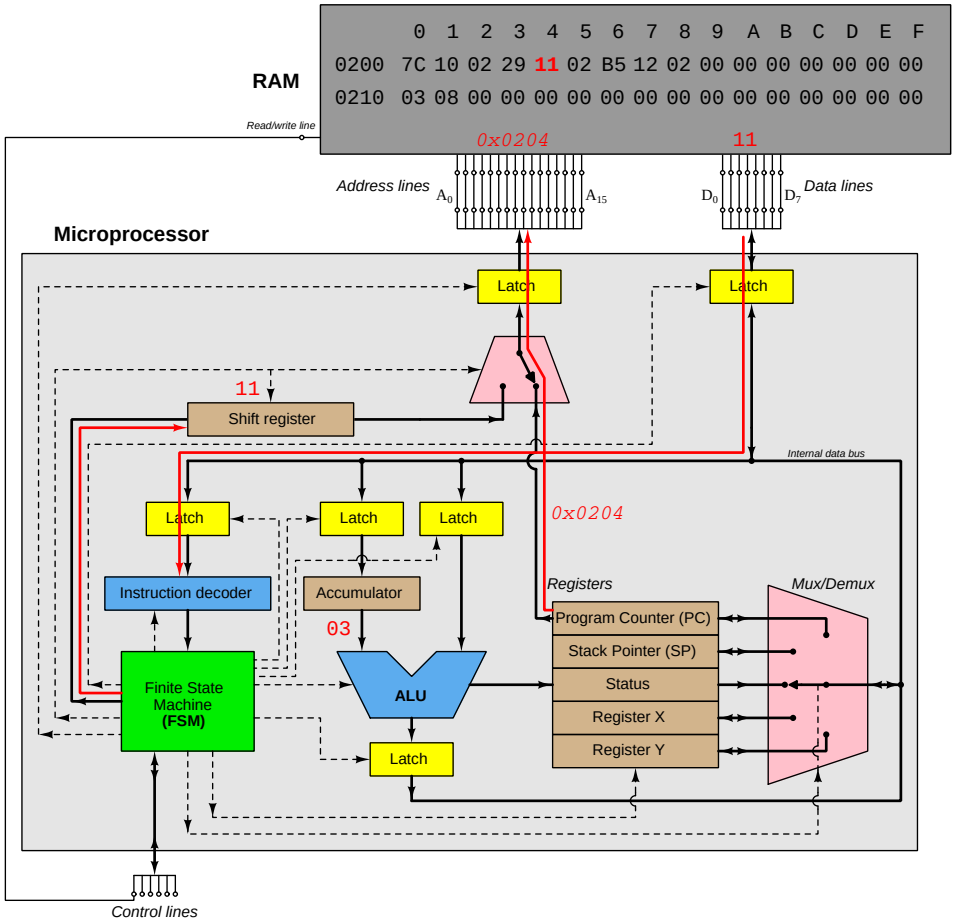
This completes the execution of the first instruction (`7C`). Now, the FSM will increment the program counter again and return control of the address to it once more.

Fetching instruction 29 from memory and decoding to initiate the FSM's sequence:



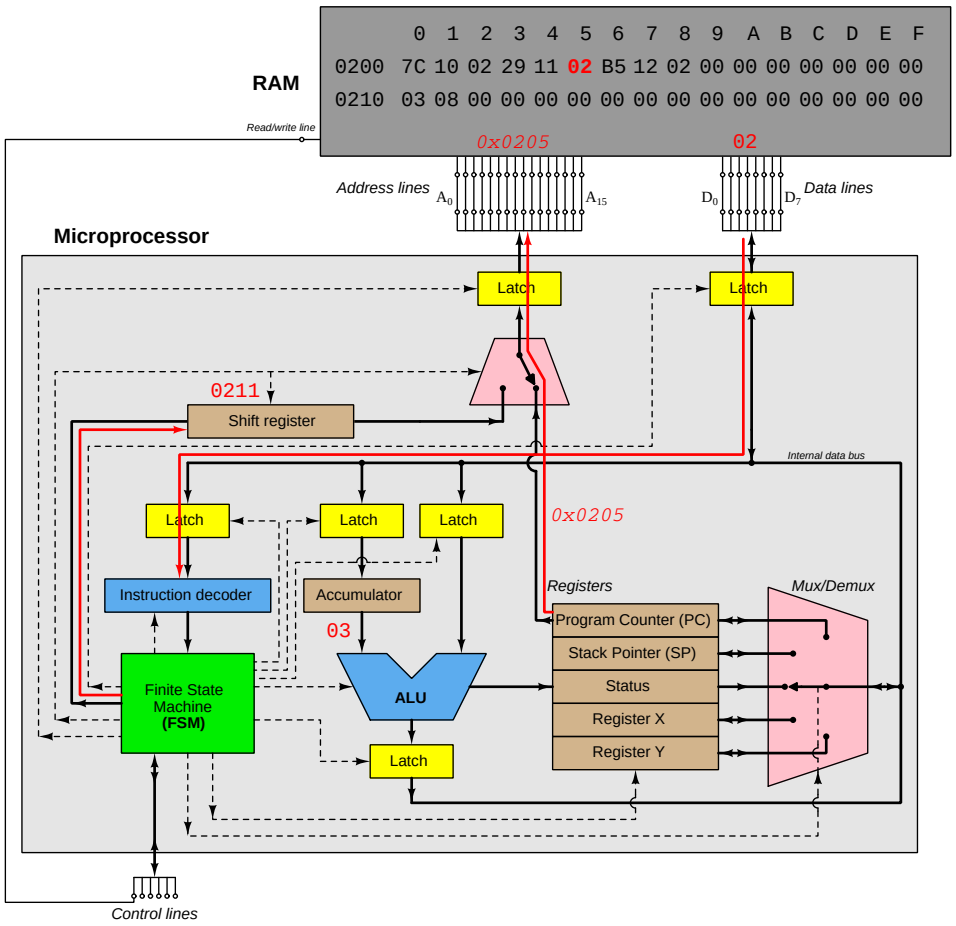
Recall that for this microprocessor the 29 instruction commands it to read a value stored in memory, at an address which itself is stored in the next two addresses in memory, and then add that value to whatever is already stored in the Accumulator. The first action taken by the FSM for the "read" instruction therefore is to increment the program counter (PC) to read the low-order byte of the address for the number which will be added to the one already stored in the Accumulator register.

Reading `11` from memory as the first byte of a 16-bit address:



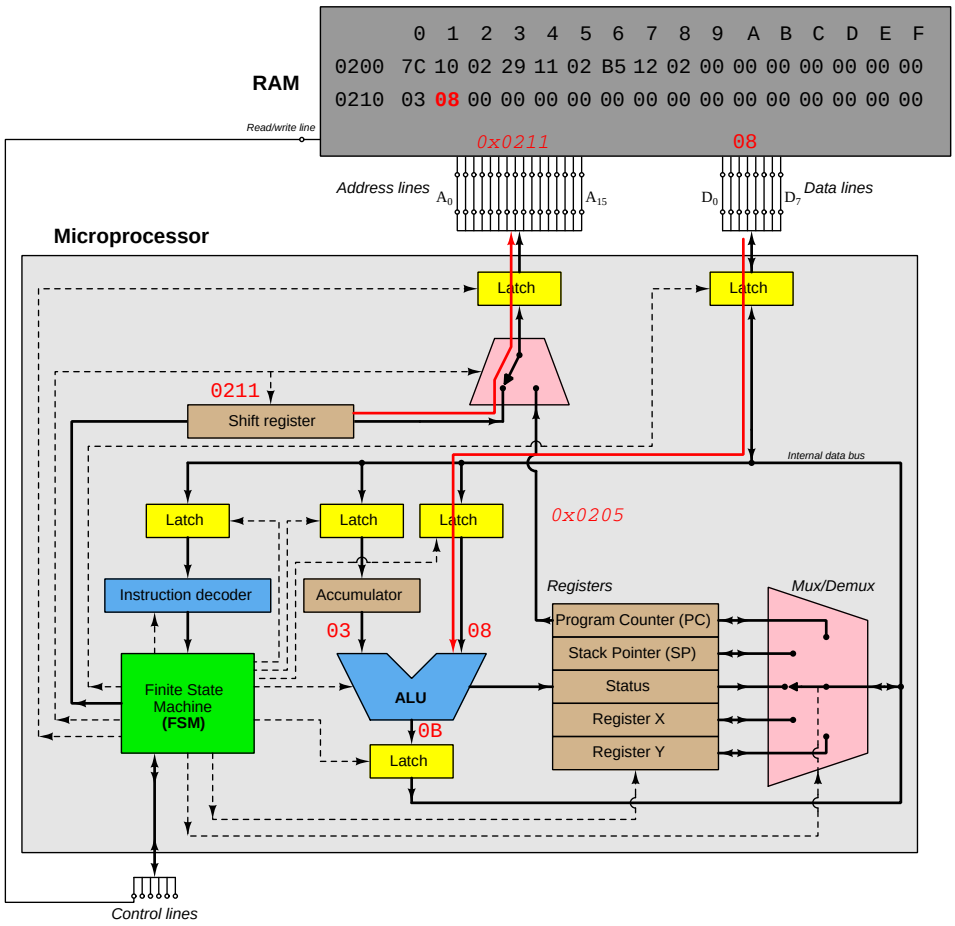
This first byte is routed to the shift register by the FSM. Next, the FSM increments the program counter again to read the next byte, in fulfillment of the `29` instruction.

Reading `02` from memory as the second byte of a 16-bit address:



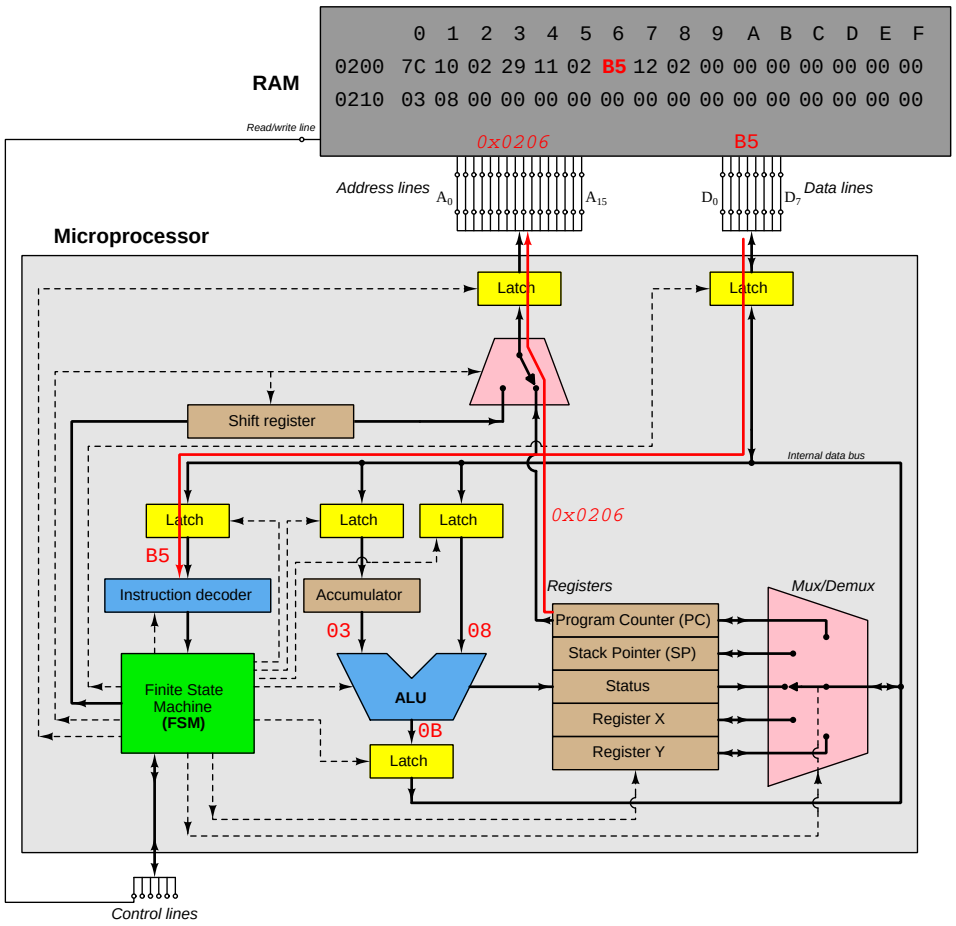
Now that both bytes of the 16-bit address are in the shift register, the FSM will use it as the next address (instead of the program counter) and read the data stored there in fulfillment of the `29` instruction.

Reading from memory address `0x0211` the second number to be added (eight):



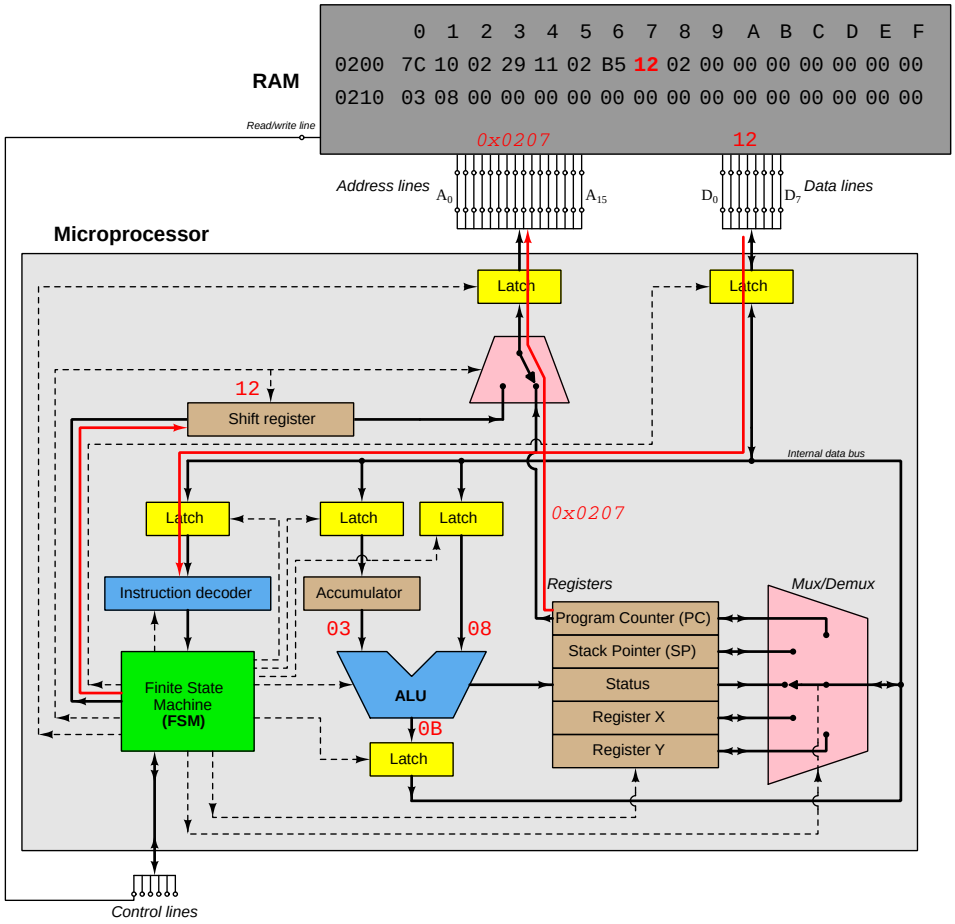
The ALU outputs a value of eleven (`0x0B`), thus completing the execution of the second instruction (`29`). Now, the FSM will increment the program counter again and return control of the address to it once more.

Fetching instruction B5 from memory and decoding to initiate the FSM's sequence:



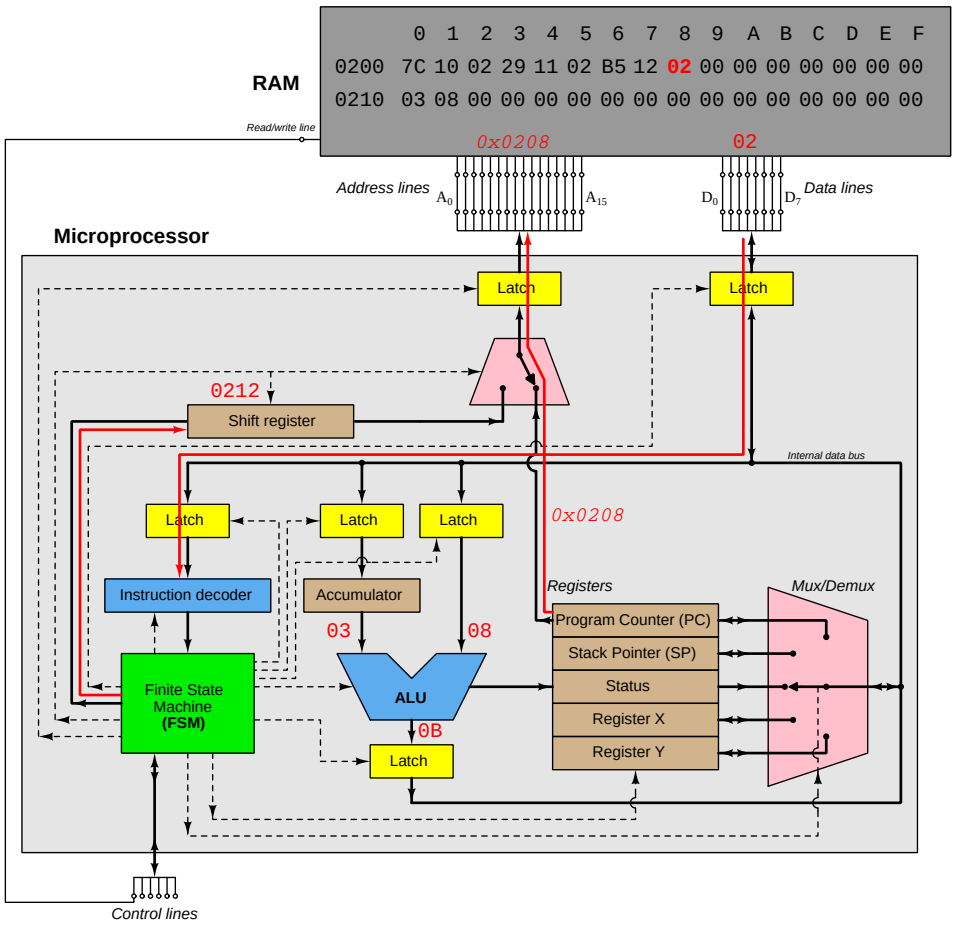
As with the 7C "read" and 29 "add" instructions, the B5 "write" instruction must also seek another address stored in RAM. The first action taken by the FSM for the "read" instruction therefore is to increment the program counter (PC) to read the low-order byte of the address where the sum will be written.

Reading 12 from memory as the first byte of a 16-bit address:



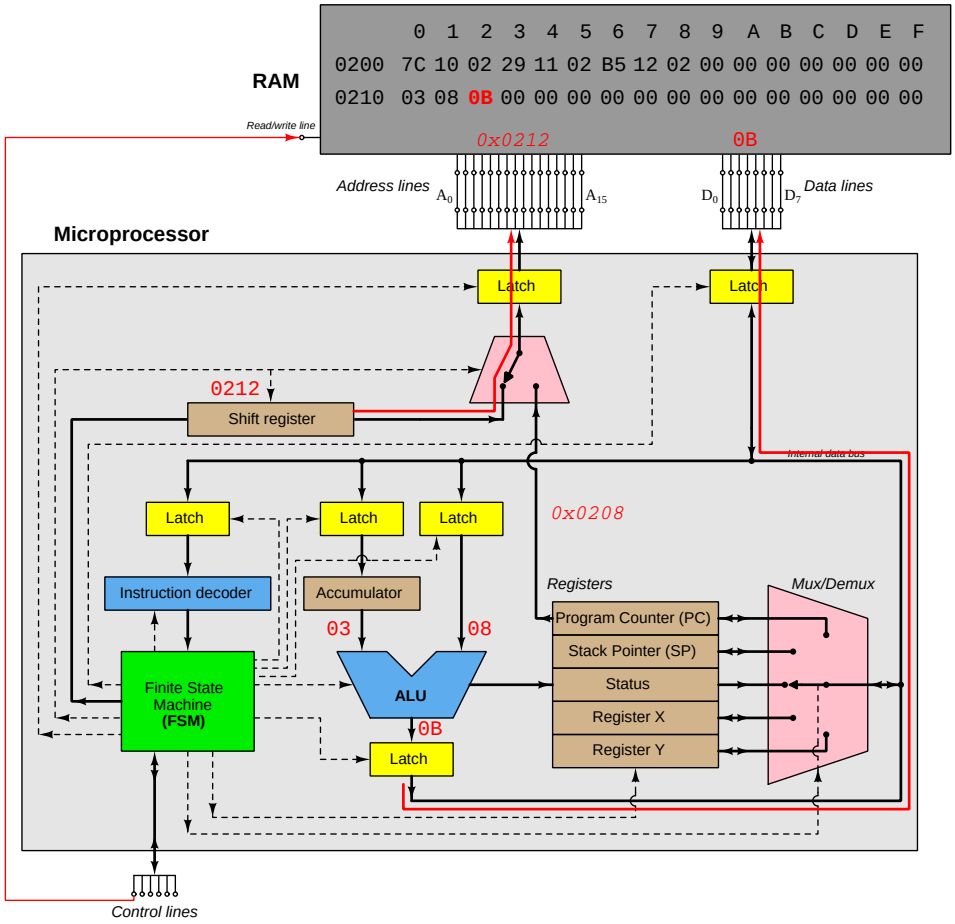
This first byte is routed to the shift register by the FSM. Next, the FSM increments the program counter again to read the next byte.

Reading `02` from memory as the second byte of a 16-bit address:



Now that both bytes of the 16-bit address are in the shift register, the FSM will use it as the next address (instead of the program counter) and write the ALU's sum there.

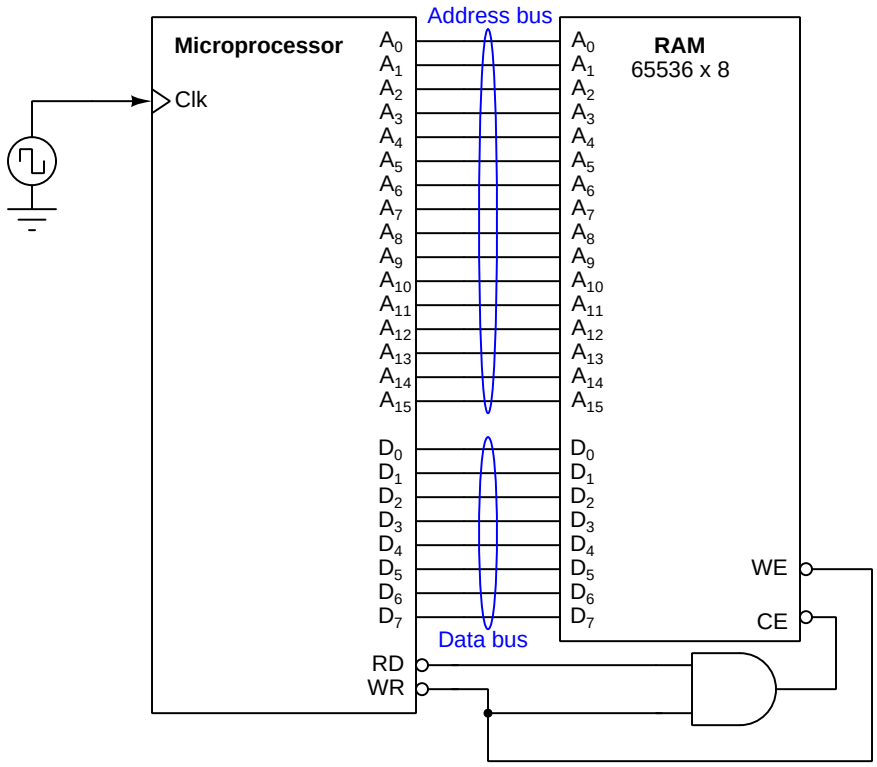
Writing the sum `0B` from the ALU's output to the RAM:



With this last cycle, both the `B5` instruction and the entire program's execution is complete. The sum of our two numbers *three* and *eight* has been stored in its own memory location.

2.3 A simple computer example

Perhaps the simplest possible implementation of a microprocessor is shown in the following schematic diagram, consisting solely of the microprocessor IC plus a RAM memory IC plus a single AND gate:

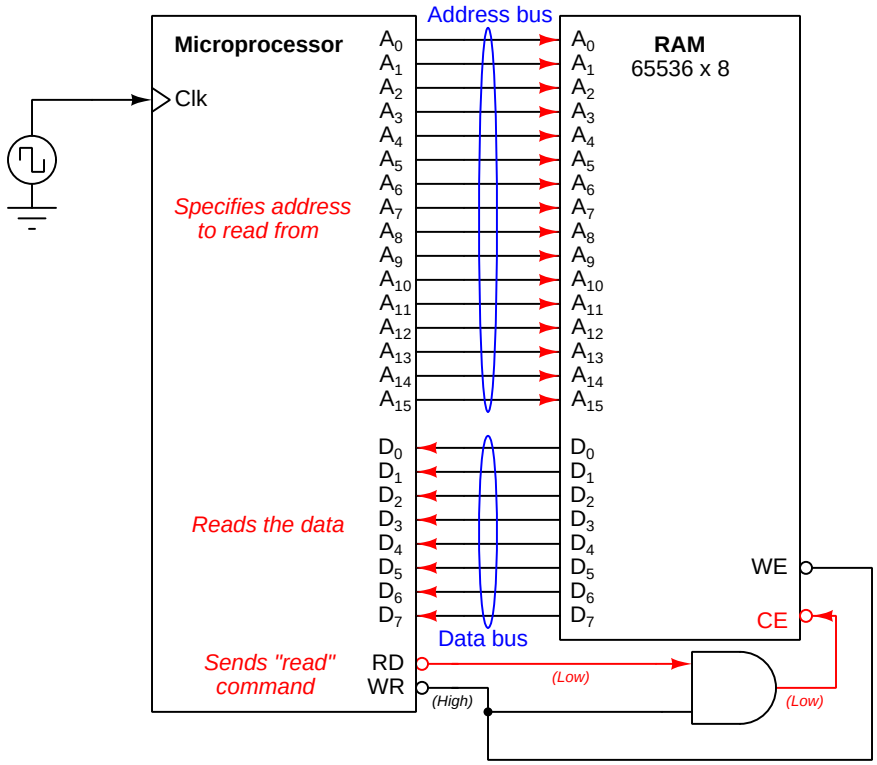


Note that these pin designations do not correspond to any particular microprocessor, but represent functions common to all of them. With the memory IC and microprocessor both having 16 address lines, this corresponds to $2^{16} = 65536$ unique addresses, each one storing an 8-bit value. The microprocessor has separate "read" (RD) and "write" (WR) control lines, and so the AND gate is necessary to activate the RAM memory IC's single "chip enable" (CE) input from those two output control lines[8].

We commonly refer to any collection of wires used to convey related electrical signals as a *bus*. In this schematic we can see two busses in the computer circuit, one for the address and one for the data. Often you will see busses represented as single lines in a diagram (often **thick** lines) to distinguish them as such, but here we will begin our exploration of this simple computer by showing all the individual conductors connecting the microprocessor to the memory circuit.

[8] We need the RAM to be enabled for every "read" as well as every "write" operation, and so the "chip enable" line needs to activate if either "read" or "write" lines activate. Although it may seem as though an OR gate would be best-suited for this task, since all the outputs and inputs are active-low the AND gate is actually best (i.e. an AND gate's output is guaranteed to be low if either one or the other input is low).

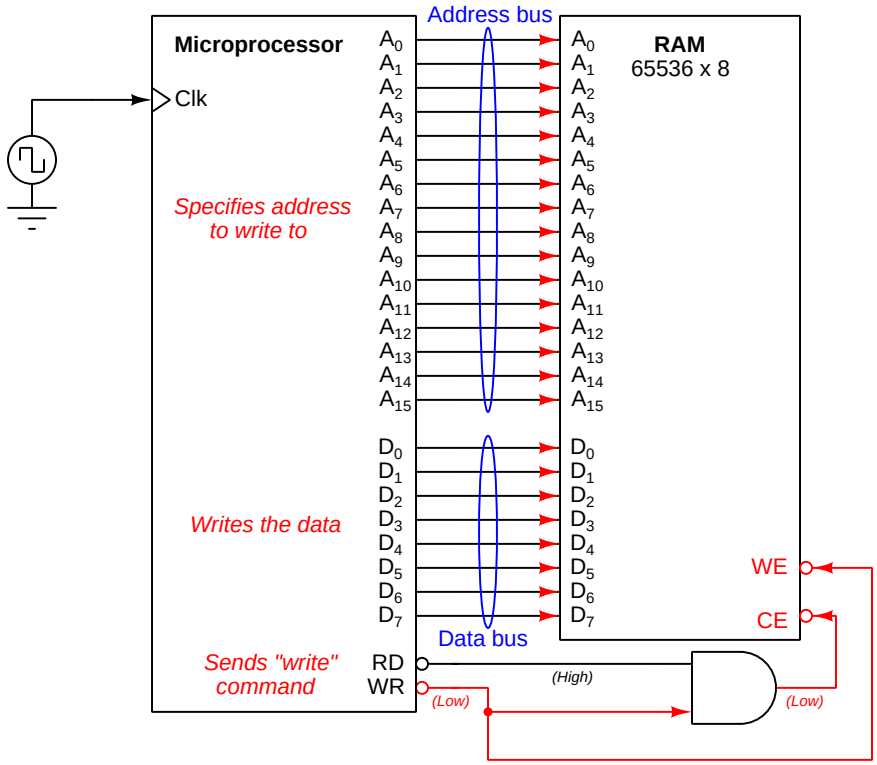
When the microprocessor executes an instruction requiring data to be read from memory, it activates[9] the Read (RD) output line and keeps the Write (WR) inactive. The RAM responds by outputting data stored at that address on the data lines to be received by the microprocessor:



A microprocessor's address bus is always unidirectional, with the microprocessor setting and clearing the address line states while the memory receives the specified address. The data bus is bidirectional, and so when the microprocessor commands the RAM to go into its "read" mode the microprocessor's own data lines act as inputs to receive that data. This received data is then held in a register internal to the microprocessor for use over future clock cycles.

[9] Note how the Read and Write control lines are both active-low, as are the Chip Enable (CE) and Write Enable (WE) inputs of the RAM memory IC.

When the microprocessor executes an instruction requiring data to be written to memory, it activates the Write (WR) output line and keeps the Read (RD) inactive. The RAM responds by receiving data output on the data bus by the microprocessor and stores it at the specified address:



The microprocessor still outputs address line states to the memory, but during this "write" cycle the microprocessor's data lines now function as *outputs* to send that data to memory. This data then becomes stored in the memory IC's cells selected by the specified address.

All input and output lines on the microprocessor involved with selecting external devices, synchronizing communications, and such are often considered to form another bus called the *control bus*. Thus, in any microprocessor-based system you will find an address bus, a data bus, and a control bus[10]. This particular microprocessor IC has an extremely minimal control bus, consisting of just Read (RD) and Write (WR) output lines. Practical microprocessors have many more control bus lines, but we are eliminating those just to show what is absolutely necessary to access memory.

[10] In simple systems you find one of each bus type. Complex computer systems may have others as well!

As helpful as this simple schematic may be to demonstrate read and write cycles between a memory IC and a microprocessor, it does not constitute a useful computer. The first major limitation is that it has no provision to accept data from any external devices (e.g. keyboard switches, sensors, data storage drives, networks). All it can do is read data from RAM and write data to RAM. The second major limitation is that it would be terribly inconvenient to program. Traditional "RAM" memory is volatile, and so it loses all its data when de-energized. This means every time we turn this circuit on the memory would contain random and useless information. Using *nonvolatile RAM* or *NVRAM* would solve the power-on problem, but we would still have to unplug this NVRAM memory IC from its socket on our computer's PCB and use some other circuitry to write a program into it, then plug it back into its socket on our computer PCB to re-connect it to the microprocessor after programming.

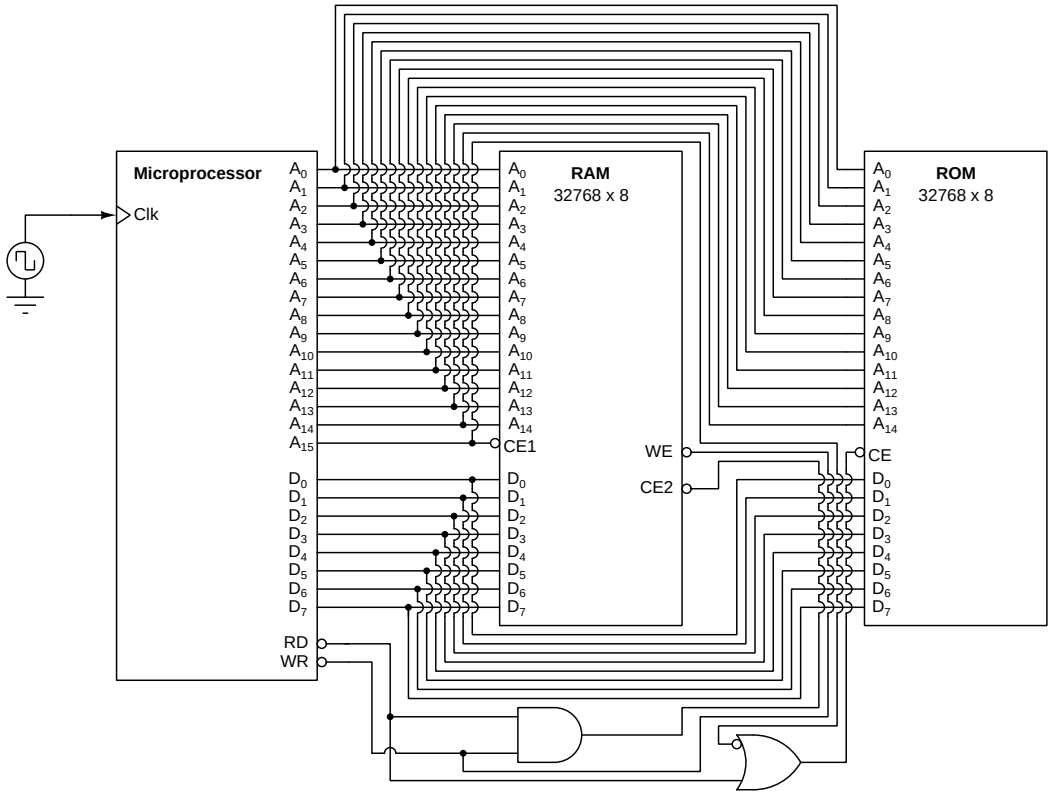
It makes more sense to use a ROM[11] – which may be pre-programmed and will retain its data indefinitely – to store the microprocessor's program while using traditional (volatile) RAM to store and retrieve data generated by the running program. Our next step in the evolution of this computer circuit's design is to add ROM in addition to the RAM it already has. After that, we will consider how to add external input and output capability to this circuit so our computer will be able to interface with sensors, lamps, and other devices other than memory arrays.

However, these expansions pose another challenge. If the microprocessor needs to communicate with two different memory devices as well as (eventually) input and output interfaces, we must somehow equip the microprocessor to select *which* device to communicate with. Somehow, we must find a way to allow the microprocessor to specify which device (memory IC or external interface) it will communicate with, while permitting all these devices to share the one and only 8-bit data bus our microprocessor possesses.

This problem is related to the concept of *multiplexing*, where multiple devices must alternately share the same communication channel with one another. When multiple devices exchange data over a common "bus" we refer to this general problem as *bus contention*. In order to prevent multiple devices from "contending" or "colliding" with one another over the bus (i.e. attempting to output contradicting logic states to one or more of the lines within that bus) we must have some means to selectively enable devices so only one of them will ever be able to output to the bus at any given time.

[11] Any suitable ROM-type memory will suffice. Mask-programmed ROMs of course would work if we didn't care about being able to ever edit the program. One-time programmable PROM memory ICs could also be used. In most cases it makes sense to apply some form of EPROM technology for this purpose, giving us the option of modifying the microprocessor's program to fix mistakes ("bugs") or to add features.

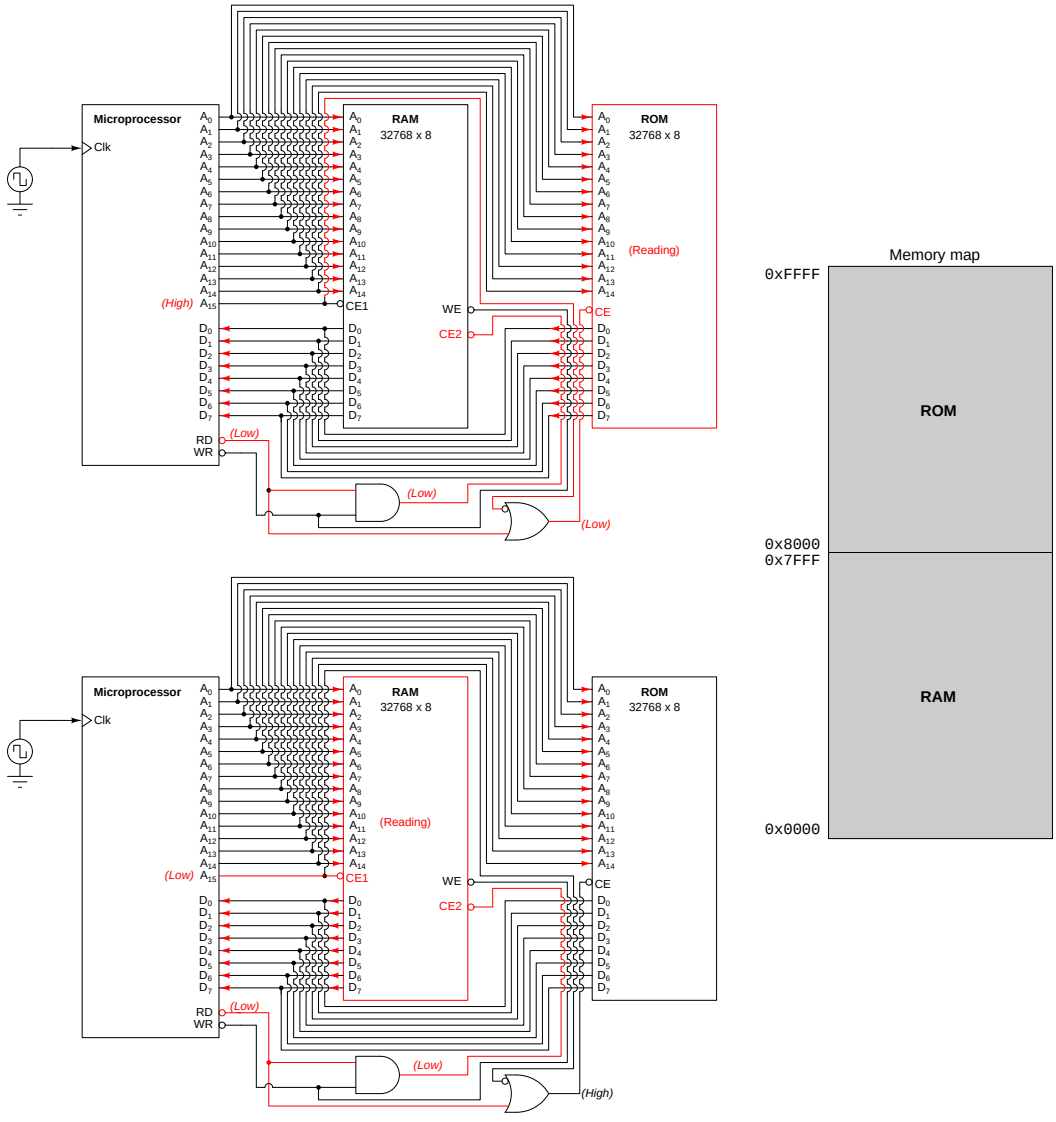
A more practical computer circuit incorporating both ROM and RAM is found in the following schematic:



Close inspection of this diagram reveals all eight data lines on each IC (D_0 through D_7) are simply paralleled together to form an 8-bit data bus. However, only the first fifteen address lines are shared in common: A_0 through A_{14}. The sixteenth address line is used as a chip-selecting bit, ensuring both RAM and ROM cannot be simultaneously enabled (i.e. ensuring they can never simultaneously output data to the 8-bit data bus). The RAM is enabled only if *both* its Chip Enable inputs ($CE1$ and $CE2$) are active (low), and the $CE1$ input connects to the microprocessor's sixteenth address line (A_{15}). In contrast, the ROM is enabled only if its single Chip Enable (CE) input is active, and its state is the complement of A_{15}. Thus, the RAM serves the lower half of the addresses space (`0x0000` through `0x7FFF`, when A_{15} is low) while the ROM serves the upper half of the address space (`0x8000` through `0xFFFF`, when A_{15} is high). The AND gate ensures the RAM cannot enable unless either a Read (RD) or[12] Write (WR) control signal activates; the OR gate ensures the ROM cannot enable unless the Read (RD) signal activates. Note the use of 32768 x 8 memory ICs instead of the single 65536 x 8 RAM – this is a necessary compromise because the microprocessor only has sixteen address lines which limits its *total* memory space to 65536 addresses.

[12] An AND gate provides an "OR" function if the logic is negative (i.e. active-low): all it takes is one *or* the other input of the AND to be low to guarantee a low output state.

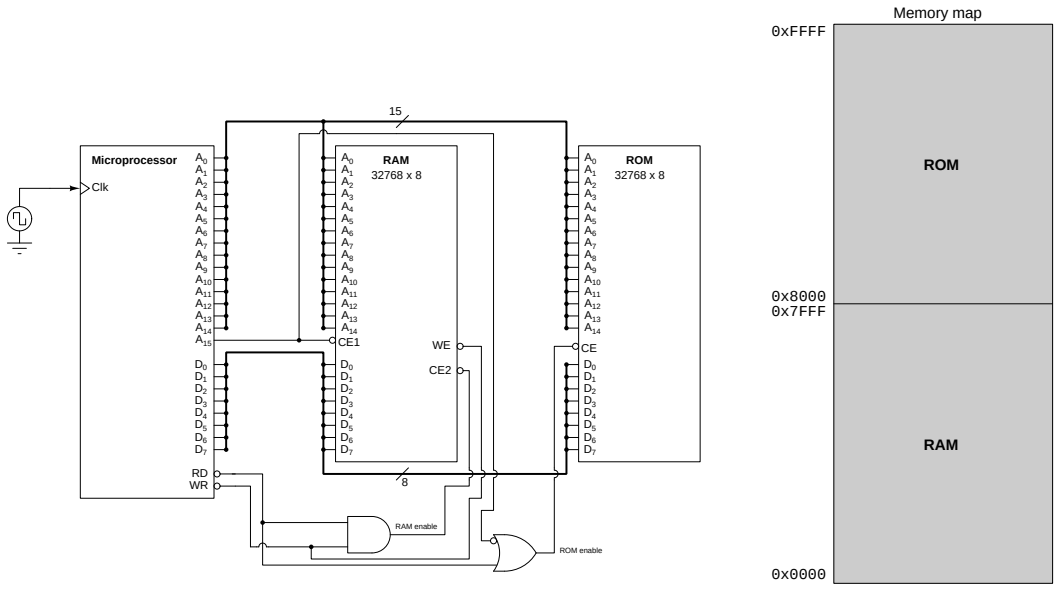
The following illustration shows this two-memory microprocessor system reading[13] from RAM versus reading from ROM, along with a "memory map" showing the two memory types dividing the shared 65536-address space:



ROM is where we would store the microprocessor's program since this data would be nonvolatile and therefore always be ready to read at power-up. RAM is where we could program the microprocessor to store and retrieve any new data generated by the program's instructions.

[13] The write cycle would look similar, the only difference being activation of the Write Enable (WE) input on the RAM memory IC. There is, of course, no write cycle for the ROM since it is read-only memory.

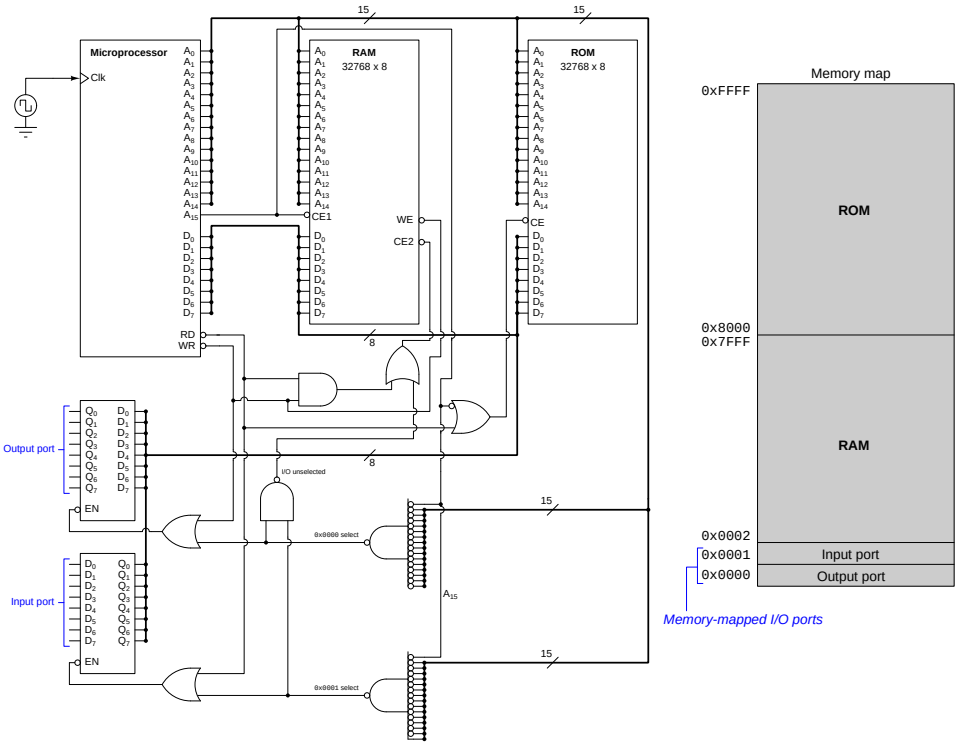
Before we proceed any further with the development of this computer's circuitry, let's unclutter the schematic diagram by using *bus notation* instead of individual wires for the address and data lines connecting these ICs. We will use thicks line to represent sets of individual wires, the number of wires in that set represented by a numeral next to a diagonal slash mark:



A notable limitation of this simple computer system is that it lacks provision for input or output to anything but its own memory banks. This means it cannot interface at all with the external world. Practical computing systems, by contrast, possess input-output (I/O) capability to interface with such devices as keyboards, monitors, lamps, switches, communications networks, external data-storage devices, etc.

We may augment this computer's design with I/O capability by utilizing the Read (RD) and Write (WR) control lines intended to control the functions of interface circuits in addition to the RAM and ROM memory ICs. If we just build some additional circuitry that is addressable just like cells within the RAM or ROM chips, and connect that circuitry to the address and data busses just like the RAM and ROM ICs, then the microprocessor will be able to read data from and write data to this circuitry as though it existed within a specific segment within the memory space.

Consider the following solution providing two 8-bit *I/O ports* for this system, each port's interface circuitry consisting of a single 8-bit D-type parallel-in, parallel-out register (latch):



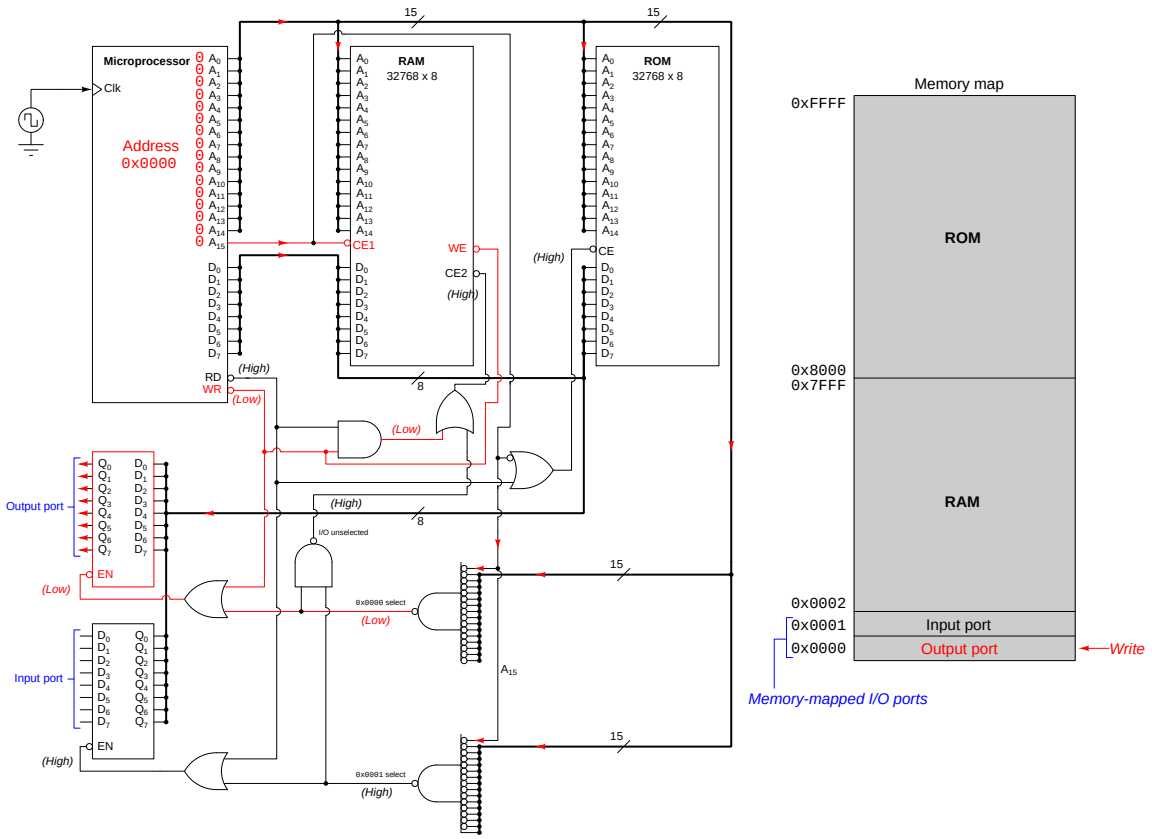
As complicated as this may appear, the purpose for each additional line and gate is remarkably simple. We want the output port to be enabled whenever the microprocessor writes to address `0x0000`, and we want the input port to be enabled when the microprocessor reads from address `0x0001`. To do this, we must first extend the 8-bit data bus to connect to both port latches, and then we must extend the address bus to bring all 16 address lines down to some new decoding logic. The two new 16-input NAND gates perform the address decoding (i.e. identifying `0x0000` and `0x0001` respectively) while the two new OR gates driving the latch enable inputs ensure proper read/write direction (e.g. so it will be impossible for the microprocessor to *write* to the input port or *read* from the output port). The last new gates we added – a two-input NAND gate and a two-input OR gate – add one more condition necessary to enable the RAM memory IC: the RAM can enable for reading or writing only if *both* I/O ports are unselected. This prevents a potential collision between the I/O latches and the RAM for addresses `0x0000` and `0x0001`[14].

To summarize, the new logic simply prevents more than one device from being enabled by the microprocessor at any given time: either RAM *or* ROM *or* the Input port *or* the Output port, but no simultaneous combination.

[14]This "memory-mapped I/O" design also wastes two otherwise usable memory spaces in the RAM memory IC. If we are using `0x0000` as the address for the Output port and `0x0001` as the address for the Input port, then we cannot use those same addresses within RAM. Considering our RAM has an address space of 32768, though, it will not make much of an impact to sacrifice two of them in order to have I/O ports for our computer system.

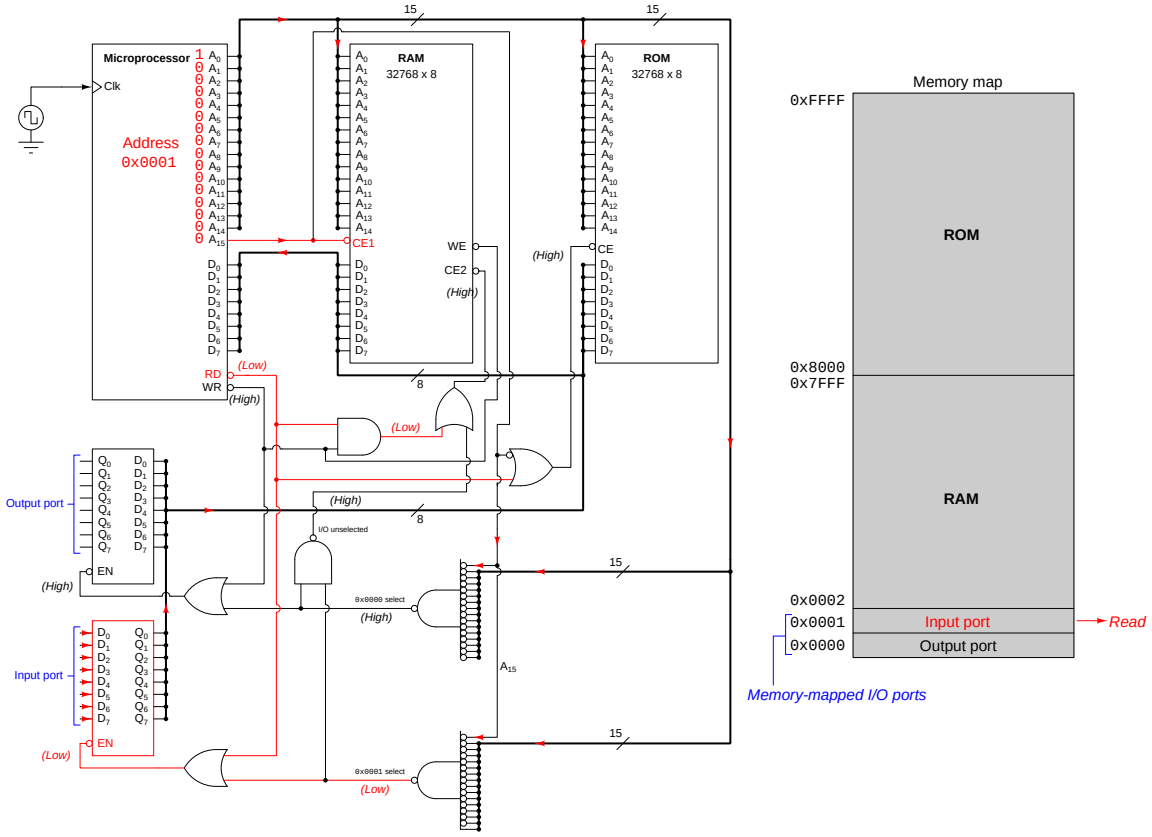
To show how this selection logic works, we will once again use red coloring to denote activated states and enabled devices, for each of the following conditions: writing to the Output port, reading from the Input port, reading from RAM, writing to RAM, and reading from ROM.

First, writing to the Output port at address `0x0000`:



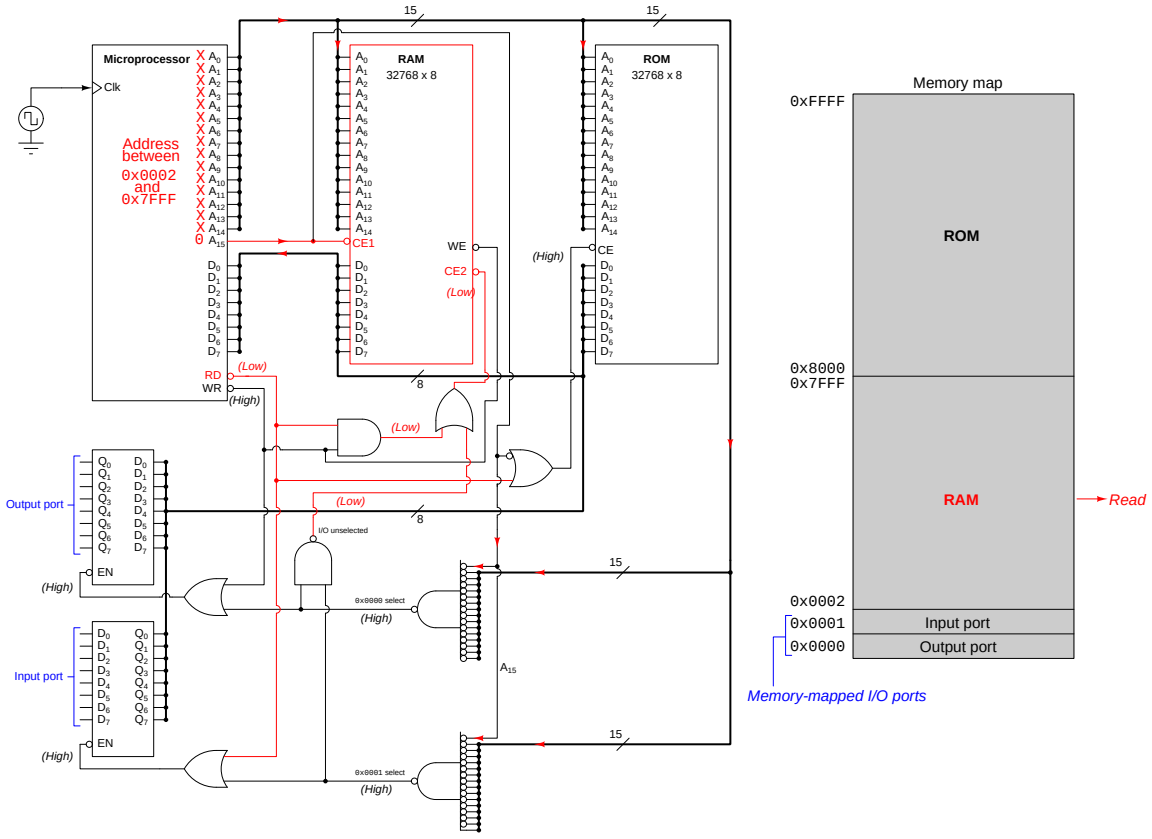
- The **Output port is enabled** to take data from the data bus because its 16-input NAND gate is enabled (all 16 address bits are low, which when run through the 16 inverted input lines of the NAND gate force that gate's output to go low) and because the microprocessor's Write output (WR) is active (low).
- The Input port is disabled because it lacks the correct address states to enable its 16-input NAND gate and also because the microprocessor's Read (RD) output is inactive (high).
- The RAM is disabled because is second Chip Enable input ($CE2$) is inactive (high), forced so by the "I/O unselected" line being inactive (high).
- The ROM is disabled because its Chip Enable input (CE) is inactive (high), due to address bit A_{15} being low.

Next, we show the microprocessor reading from the Input port at address `0x0001`:



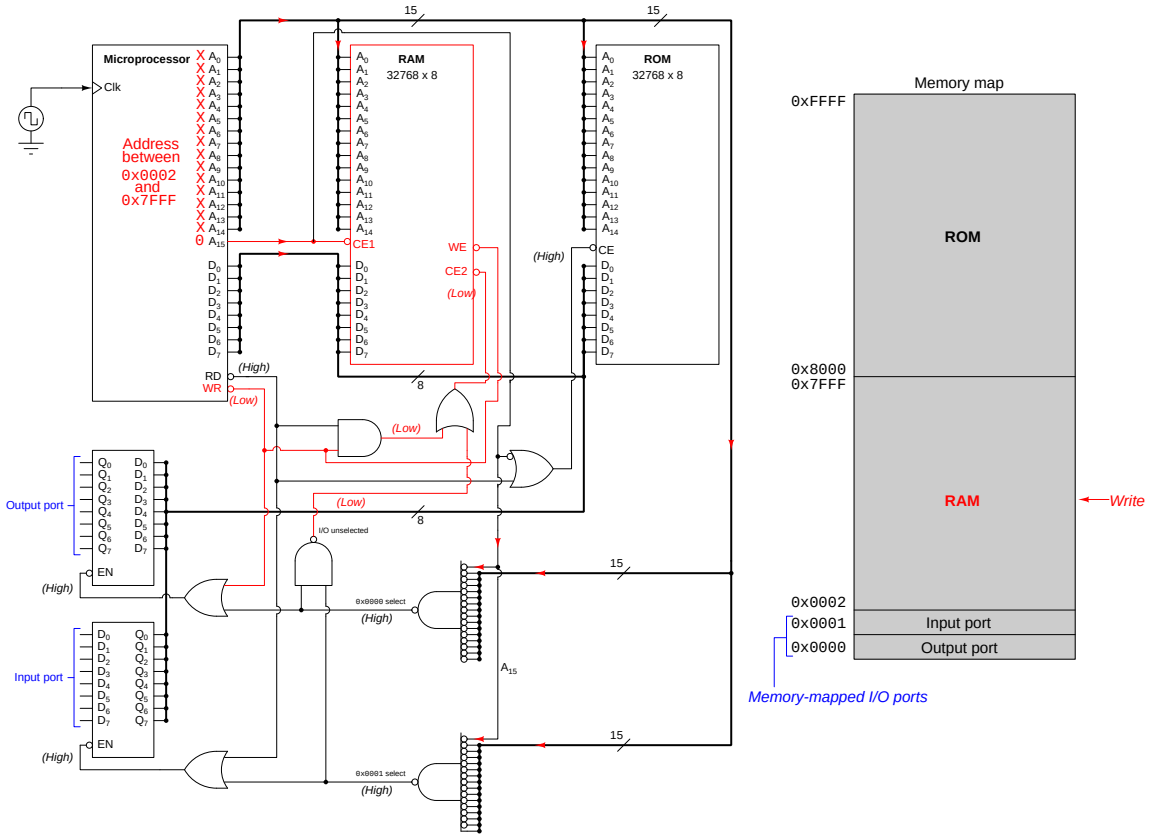
- The Output port is disabled because it lacks the correct address states to enable its 16-input NAND gate and also because the microprocessor's Write output (WR) is inactive (high).
- The **Input port is enabled** because its 16-input NAND gate is enabled (address bits `0000 0000 0000 0001`, which makes that 16-input NAND gate output go low because 15 of the 16 inputs are inverted, that LSB input being non-inverted which detects the "1" bit in address `0x0001`) and because the microprocessor's Read output (RD) is active (low).
- The RAM is disabled because is second Chip Enable input ($CE2$) is inactive (high), forced so by the "I/O unselected" line being inactive (high).
- The ROM is disabled because its Chip Enable input (CE) is inactive (high), due to address bit A_{15} being low.

Next, we show the microprocessor reading from RAM at any address between `0x0002` and `7FFF` (inclusive):



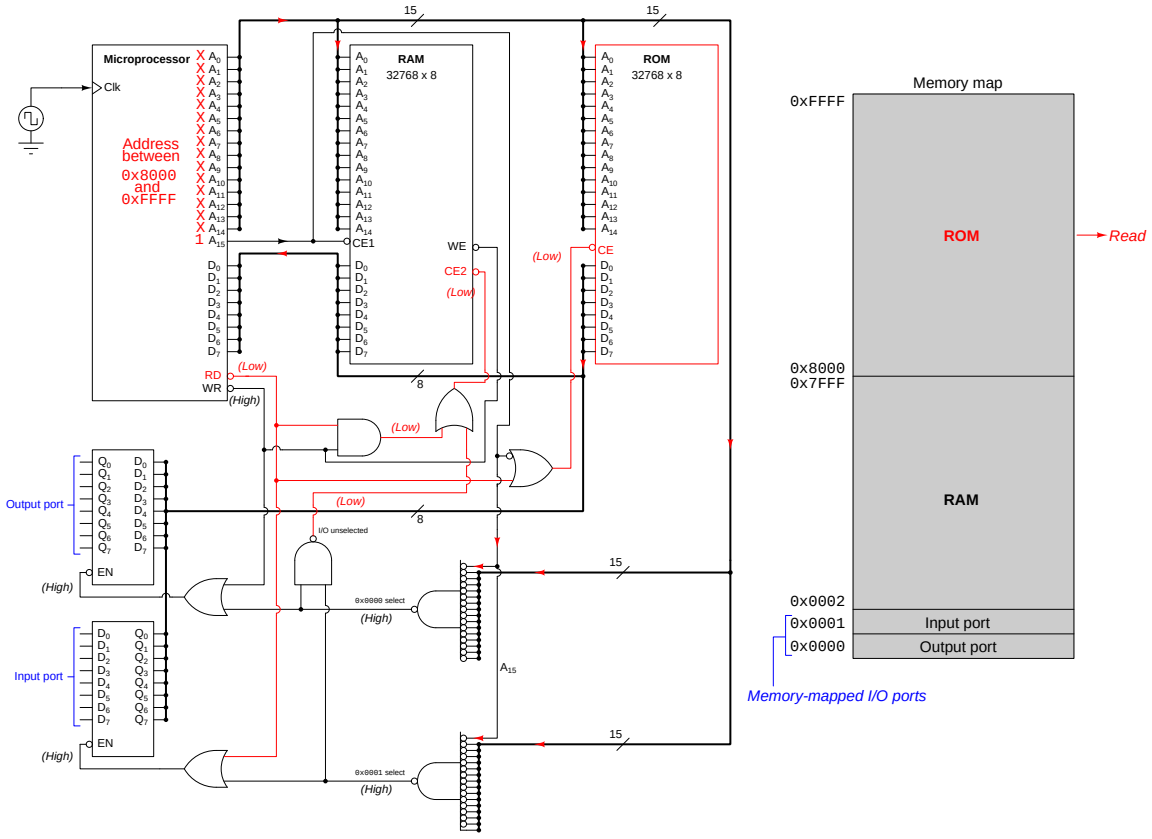
- The Output port is disabled because it lacks the correct address states to enable its 16-input NAND gate and also because the microprocessor's Write output (WR) is inactive (high).
- The Input port is disabled because it also lacks the correct address states to enable its 16-input NAND gate.
- The **RAM is enabled** because both its Chip Enable inputs ($CE1$ and $CE2$) are active (low).
- The ROM is disabled because its Chip Enable input (CE) is inactive (high), due to address bit A_{15} being low.

Next, we show the microprocessor writing to RAM at any address between `0x0002` and `7FFF` (inclusive):



- The Output port is disabled because it lacks the correct address states to enable its 16-input NAND gate.
- The Input port is disabled because it also lacks the correct address states to enable its 16-input NAND gate and also because the microprocessor's Read output (RD) is inactive (high).
- The **RAM is enabled** because both its Chip Enable inputs ($CE1$ and $CE2$) are active (low). It now accepts data from the data bus because its Write Enable input (WE) is active (low).
- The ROM is disabled because its Chip Enable input (CE) is inactive (high), due to address bit A_{15} being low.

Lastly, we show the microprocessor reading from ROM at any address between `0x8000` and `FFFF` (inclusive):



- The Output port is disabled because it lacks the correct address states to enable its 16-input NAND gate and also because the microprocessor's Write output (WR) is inactive (high).
- The Input port is disabled because it also lacks the correct address states to enable its 16-input NAND gate.
- The RAM is disabled because its first Chip Enable input ($CE1$) is inactive (high), due to address bit A_{15} being high.
- The **ROM is enabled** because its Chip Enable input (CE) is active (low).

This is how memory-mapped I/O works in a computer system: the I/O ports are wired such that they are enabled only when specific address values are asserted on the address bus. As far as the microprocessor "knows", the input port is nothing more than one address in memory where real-world input data somehow lands, and the output port is nothing more than a different address in memory where data written somehow goes out to the real world.

2.4 Machine code and assembly language

In order for a microprocessor to be useful it must have instructions in memory ready to fetch and execute. A natural question to ask at this point is, *"How do we get information placed in the computer's memory for the microprocessor to act upon?"* The answer to this question is multi-layered, so we will begin with the extremely simple and progress to the more complicated.

Early personal computers such as the Altair 8800 (which happened to use the 8-bit Intel model 8080 microprocessor) provided the user with a set of toggle switches and indicating LEDs as the primary interface. A photo of an Altair 8800 powered up at a conference appears below[15]:

Eight LEDs show the states of the microprocessor's data bus (D_7 through D_0) while sixteen LEDs below that show the state of the address bus (A_{15} through A_0). The remaining LEDs show the 8080 microprocessor's control bus line states. Sixteen toggle switches located immediately below the address LEDs provide a means for manual user input, which was the primary method for entering machine code into this computer's RAM memory. The user would set the address toggle switches for the desired memory address (up = high and down = low) and then momentarily toggle the "Examine" switch upwards to enter that address into the 16-bit address latch, followed by setting the eight lower-order switches for the desired data word and then momentarily toggling the "Deposit" switch upwards to write to the RAM memory. After entering all the machine codes for a program, the user could then reset the microprocessor to start at address `0x0000` and set the Stop/Run switch to the "Run" position to begin clocking the microprocessor to make it fetch the first instruction from that address and subsequently run the program.

If this sounds tedious to you, know that it absolutely is (or was). Thankfully, we don't have to do this sort of thing anymore to operate a computer. Even those pioneering users who were forced to toggle in their machine-language programs bit-by-bit invented methods to make life easier for themselves. This included toggling in a simple "bootloader" program that would then instruct the computer to receive information from a punched paper-tape, magnetic tape, or other durable medium for storing digital data. Bootloaders minimized the amount of "toggling" that one had to do and therefore expedited the process of loading large programs into the computer's RAM. The

[15] Credit goes to Fernando Sáenz for taking this photograph at the RetroMadrid conference on 29 April 2018, and for placing it into the public domain.

concept of bootloading is still alive and well today, with modern personal computers first accessing a small non-volatile memory bank[16] upon power-up in order to receive instructions on how to load the next phase of the software into RAM memory.

A welcome alternative to manually toggling in bootloader code into a freshly-powered computer is the use of nonvolatile (ROM) semiconductor memory to store some initial machine-language code instructing the microprocessor to be more useful to a human programmer. During the early days of personal computing this concept became quite refined to include *monitor* programs as well as entire *language interpreters* written to ROM ICs. A "monitor" program instructs the microprocessor to receive information from a keyboard and output to either a teletype machine or a video console, allowing the user to *type* and review machine-language code in the computer's memory upon power-up.

One of the more popular monitor programs was called *BUFFALO* ("Bit User Fast Friendly Aid to Logical Operation"), shown below displaying a portion of the RAM memory's contents for a Motorola model 68HC11 microcontroller[17]:

```
>md 0020

0020 B6 C1 00 BB C1 01 B7 C1  10 7E E0 00 FF FF FF FF
0030 FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF
0040 FF FF FF FF FF FF FF FF  FF FF E4 E4 E4 E4 64 E3                d
0050 F3 00 E4 64 E3 F3 00 57  00 20 E4 B6 E7 BA E1 FA     d d   X
0060 FF FF FF FF FF FF FF FF  D0 00 41 6D 64 20 30 30             Amd 00
0070 32 30 0D 20 63 31 30 30  0D FF FF FF FF FF FF FF  20  c100
0080 FF FF FF FF FF FF FF FF  FF FF FF FF FF FF 4D 44                MD
0090 20 1B 5B 42 4D 4D 00 20  B6 00 B7 FF 00 00 00 00    [BMM
00A0 00 00 00 00 01 01 01 00  04 21 00 20 00 00 00 72           @     r
```

Here, the BUFFALO monitor allows the user to communicate with the microprocessor via either a teletype machine (with keyboard and paper-printer mechanism) or another computer's keyboard and video monitor through a serial data connection[18]. The monitor program accepts certain typed commands and replies with text responses to those commands. In the example shown above, the command `md 0020` instructs the BUFFALO monitor to print a "hex dump" of the memory's contents beginning with address `0x0020`.

Being able to type data into the computer's RAM using a keyboard and view it on a screen is far more convenient than entering machine-language code using toggle switches and reading it using LEDs!

[16] For example, the *BIOS* memory in an IBM-compatible personal computer.

[17] A *microcontroller* is a microprocessor combined with on-board semiconductor memory and I/O interfacing circuitry to make it a functioning computer within a single integrated circuit.

[18] This `md 0020` command and response was captured from a serial terminal display I used on a personal desktop computer to communicate with the Motorola 68HC11 single-board computer. If this had been done in the mid 1980's I would have used a teletype machine instead, and the output being typed on a sheet of paper.

Despite the convenience afforded by a monitor program such as BUFFALO, programming a microprocessor in machine code – whether in binary form or in hexadecimal "shorthand" representation – is still tedious and error-prone. Codes such as `B6` and `BB` may make sense to the microprocessor, but they do not make natural sense to any human being. In order to write programs in this manner, one must either memorize the instruction set of the microprocessor and/or continually reference a book describing the machine codes for each instruction offered by the IC.

A solution to this problem is the use of software called *assemblers*. An "assembler" is a computer program written for the purpose of translating mnemonic codes into machine code the microprocessor can directly understand. The mnemonic codes resemble abbreviated English words or phrases, and are therefore easier to remember and easier to interpret when viewed on a sheet of paper or on a video display. For example, let us look again at the first line of the hex dump shown earlier for a Motorola model 68HC11's memory:

```
0020 B6 C1 00 BB C1 01 B7 C1 10 7E E0 00 FF FF FF FF
```

Each of these bytes (except for the `FF`'s toward the end) represents code for a simple program instructing the computer to add two numbers together. If we re-write this as separate lines, one per instruction, and add some comments for each, the program becomes easier for us to interpret:

```
0020     -- Starting memory address for the program
B6 C1 00 -- Load Accumulator A with number stored in address C100
BB C1 01 -- Add to value in Accumulator A with number stored in address C101
B7 C1 10 -- Store value in Accumulator A to address C110
7E E0 00 -- Jump to address E000 (the start of the BUFFALO monitor)
```

From these comments we can tell instruction `B6` means to "Load Accumulator A", instruction `BB` means "Add to Accumulator A", instruction `B7` means "Store from Accumulator A", and instruction `7E` means "Jump"[19]. However, an assembler is an improvement upon this. In the next code listing we see the *assembly language* source code for this exact same program:

```
    org $0020

    ldaa $C100
    adda $C101
    staa $C110
    jmp  $E000
```

[19] Astute readers will notice that the specified memory addresses in these instructions are in *big-endian* order, while the addresses specified for the instructions in the simple microprocessor example used in the "Putting it all together" section were in *little-endian* order. It just so happens that the Motorola model 68HC11 is a big-endian processor. Many other processors (including all Intel-brand processors) are little-endian.

The three- and four-letter mnemonics make far more sense to any human reader than hexadecimal codes. By comparison with the earlier commented machine code we can tell `org` directs the assembler to make the *origin* for this program at memory address `0x0020`; next, `ldaa` *loads Accumulator A* with the number stored at address `0xC100`; after that, `adda` *adds to Accumulator A* the value stored at address `0xC101`; then `staa` *stores Accumulator A's* result to address `0xC110`; finally, `jmp` tells the microprocessor to *jump* to address `0xE000`.

As with other programming languages, assembly language permits the use of *comments* to annotate the code for even better readability. Here is the same program with comments included, using the semicolon symbol (`;`)[20] preceding each comment:

```
; ============================
;   Two-number adder
;   Assembler used:  AS11
; ============================

    org $0020    ; Starting address

    ldaa $C100   ; Load A with value stored at C100
    adda $C101   ; Add to A the value stored at C101
    staa $C110   ; Store A to C110
    jmp  $E000   ; Jump to E000
```

Either this or the previous assembly code listing may be processed by an assembler and converted in executable machine code that the processor is able to directly understand, although the latter example is much easier for any human reader to understand thanks to all the comments. Some monitor programs (such as BUFFALO) had an assembler function built-in which meant you could directly enter the assembly codes using the keyboard and "assemble" that finished source code into machine code for the same microprocessor to run. A more modern approach to assembly-language programming is to type the assembly source code into an entirely different computer, and have that computer translate it into machine code for downloading to the target microprocessor's memory[21]. Once assembled into machine code, the program may be written to a PROM memory IC to be installed in the target computer, or sent to the target computer via a communications cable assuming the target has a monitor program to instruct it how to receive the communicated data.

Just as machine language is specific to the model of microprocessor being programmed, assembly language likewise is specific to both the microprocessor model and the assembler software used to translate assembly "source code" into machine language.

[20] Those familiar with higher-level programming languages such as C, C++, Java, and Python will recognize the semicolon symbol as having a very different meaning in those languages than a comment marker!

[21] This is called *cross-assembly* (using one computer to assemble code for a different type of computer to execute), and it is precisely what I did to create the machine-language executable code from the assembly source code in this example. I typed the assembly source code into a plain-text file (`myprogram.asm`) on my desktop computer and then used the assembler `as11` to translate this source code into an "object code" file (`myprogram.s19`) which was then downloaded to the Motorola 68HC11 computer through a serial data cable. Once loaded into the RAM of the target microprocessor, I could display the memory contents and run my program using commands available to me through the BUFFALO monitor.

2.5 Interrupts

An important concept in microprocessor and microcontroller operation is the notion of an *interrupt*. This is a signal, either received from some external source or generated internally, that causes the processor to halt its regular execution of the program and "jump" to a new location in program memory where other instructions exist to tell it how to *service* (i.e. deal with) this interrupting event.

Interrupts are often classified as being either *maskable* or *non-maskable*[22]. This simply refers to whether the user is able to disable the interrupt or not. A "maskable" interrupt is one that can be disabled by setting certain bits in a register of the processor, and a "non-maskable" is one that is always alert and cannot be disabled. The *reset* function on a microprocessor or microcontroller is a typical example of a non-maskable interrupt.

The ability to arbitrarily "interrupt" the normal execution of a program in order to perform a different task is important when the microprocessor must interface with external devices in a time-sensitive manner. For example, in order to handle incoming data transmissions from an external device, the microprocessor must be able to stop whatever else it might be doing at that time in order to avoid missing some or all of that incoming data. Interrupts are also important for "real-time" control applications where the microprocessor is tasked with receiving information and sending information regarding high-speed physical processes (e.g. fuel injection controls on an engine), where delays of even a few milliseconds could result in poor control or even in dangerous conditions going unchecked.

The way this usually works is that an interrupt event causes the Program Counter[23] to go to a pre-designated address in memory called a *vector*, which contains the memory address where the first instruction of the service routine (typically called an "Interrupt Service Routine" or *ISR*) exists. These interrupt vectors are often hard-coded into the processor at the time of manufacture, but the user (programmer) is free to place their ISR code wherever they wish in memory and to write the appropriate starting addresses for those routines into the appropriate interrupt vector location(s).

It should be noted, however, that interrupt handling varies significantly between different microprocessor designs, and that there is no "standard" method. Be sure to reference the manufacturer's literature on the particular microprocessor you are using in order that you use their interrupts correctly!

[22] In programming, a *mask* is a set of bits intended to apply a bit-wise logical operation to bits within a larger word of data. An "interrupt mask" would be a word whose bits acted as individual enable/disable instructions for different types (or sources) of maskable interrupt signals. A microprocessor's non-maskable interrupts would have no such mask word because those interrupts are very high-priority and should never be disabled.

[23] Recall that the Program Counter is a special register inside the microprocessor used to keep track of which memory address the next instruction or operand is located at. It is literally a counter in the sense that it typically increments from one memory location to the next in sequence, but it may also jump to other addresses as commanded by certain instructions.

The following table shows a list of interrupts and corresponding vectors for the Texas Instruments MSP430G2553 microcontroller:

Interrupt source	**Vector address**	**Assembly section**	**C/C++ name**
Reset	0xFFFE	.reset	RESET_VECTOR
Non-maskable	0xFFFC	.int14	NMI_VECTOR
TIMER1_A0	0xFFFA	.int13	TIMER1_A0_VECTOR
TIMER1_A1	0xFFF8	.int12	TIMER1_A1_VECTOR
Comparator A	0xFFF6	.int11	COMPARATORA_VECTOR
Watchdog timer	0xFFF4	.int10	WDT_VECTOR
TIMER0_A0	0xFFF2	.int09	TIMER0_A0_VECTOR
TIMER0_A1	0xFFF0	.int08	TIMER0_A1_VECTOR
USCIAB0RX	0xFFEE	.int07	USCIAB0RX_VECTOR
USCIAB0TX	0xFFEC	.int06	USCIAB0TX_VECTOR
ADC10	0xFFEA	.int05	ADC10_VECTOR
Port 2 I/O	0xFFE6	.int03	PORT2_VECTOR
Port 1 I/O	0xFFE4	.int02	PORT1_VECTOR

Each interrupt is listed by type (left-hand column), the memory address of its vector (next column) where the corresponding ISR service instructions begin, the assembly-language label ("section") for the ISR code, and lastly (right column) the pre-designated name of the function for the ISR code if programmed using the C or C++ languages. This list is ordered from top to bottom in descending priority, the Reset interrupt being the highest priority and the Port 1 I/O interrupts being the lowest priority.

Chapter 3

Derivations and Technical References

This chapter is where you will find mathematical derivations too detailed to include in the tutorial, and/or tables and other technical reference material.

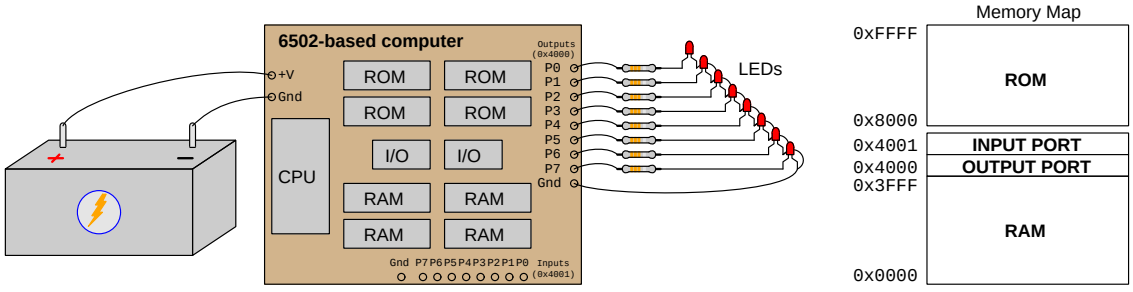
3.1 Introduction to assembly language programming

Microprocessors only understand *machine code* – instructions and data encoded as binary values, often written in hexadecimal "shorthand" form for better human-readability – but even in hexadecimal these codes are non-intuitive and confusing for human programmers to manage. *Assembly code* improves upon this situation by representing each machine-code instruction as an word-abbreviation called a *mnemonic*. The convenience of mnemonics, though, comes at price: since the mnemonics themselves are nonsense to the microprocessor, a special software package called an *assembler* must be used to translate these mnemonics into machine code that the microprocessor can understand. Just as microprocessors each have their own unique instruction set defining which codes perform which functions, assembler software has its own unique "vocabulary" and "grammar" one must abide by in order to write functioning programs. Assembly programming, therefore, is its own *language* and like all languages has specific rules.

Furthermore, assembly-language lacks the strict standardization found in higher-level programming languages such as C, C++, Java, and Python. Just as machine-code programming is specific to the model of microprocessor, assembly-language programming rules are often specific to the assembler software version, which in turn is often closely coupled to the microprocessor model. It is thus impossible to write a generic tutorial on assembly programming, just as it is impossible to write a generic tutorial on machine-code programming. What we will explore here are features of assembly code common to *most* assemblers.

We will use a specific hardware application as the foundation for this lesson on assembly language programming, in order to have a practical context for understanding what the program does and how it works. This means the examples given here will not work with any microprocessor other than the system described here, but that is okay. Many of the principles learned here find general application to other systems.

Our hypothetical computer is shown below, based on a Motorola model 6502 8-bit microprocessor. A single output port mapped to memory address `0x4000` provides the means for our program to turn LEDs on and off, by writing bit-states to that byte located at `0x4000`. Another port at address `0x0401` provides inputs, where our program may read bit-states of the byte stored there to detect logic signals applied to those pins by external circuitry (not shown). ROM begins at address `0x8000` and extends through address `0xFFFF`. RAM begins at address `0x0000` and extends through address `0x3FFF`:



The 6502 provides an Accumulator register plus two general-purpose registers (named X and Y) for temporary data storage, each one eight bits wide.

3.1.1 Machine code to blink an LED

Suppose we wish to make the LED connected to output pin P0 on the computer blink on and off. This means writing a 1 and then a 0 to bit 0 of the byte located at `0x4000` while clearing (making zero) all the other bits at that address. Our program will execute the following steps:

1. Write the binary value `0b00000001` to address `0x4000`
2. Toggle the least-significant bit of that byte in address `0x4000`
3. Go back to second instruction and repeat indefinitely

The same steps, using instructions available in the 6502's instruction set:

1. **Load** the binary value `0b00000001` into the Accumulator register
2. **Store** the Accumulator's value to address `0x4000`
3. Apply the **Exclusive-OR** function to the LSB in the Accumulator using the *mask*[1] `0b00000001`.
4. **Jump** to the second instruction and repeat indefinitely

Researching opcodes and operand formats for the model 6502 microprocessor, we find the following:

- The opcode for "Load Accumulator with immediate value" is `0xA9` followed by the desired value to load (`0x01`)
- The opcode for "Store Accumulator value to absolute memory address" is `0x8D` followed by two bytes specifying the destination address in little-endian order (low byte first, high byte last: `00 40` for address `0x4000`)
- The opcode for "Exclusive-OR with immediate value" is `0x49` followed by the mask value (`0x01`)
- The opcode for "Unconditional Jump to absolute memory address" is `0x4C` followed by the target address in little-endian order

[1] In programming, a *mask* is a set of bits intended to apply a bit-wise logical operation to bits within a larger word of data. Here, our mask of `0b00000001` means the LSB will be XOR'd with 1 (i.e. toggled, to make it switch states from whatever it was before to the opposite of that) while all other bits within the Accumulator's 8-bit word will be XOR'd with 0 (i.e. left alone).

If we write all these opcodes and operands in order, one line per complete instruction, we get the following *hand-assembled* machine code:

```
A9 01
8D 00 40
49 01
4C ?? ??
```

The question-marks are there in our code because we need to determine where our program will begin in the computer's ROM memory space in order to know which address we need to "jump" to in the last instruction. Let's assume our program starts at the very first address in ROM (`0x8000`) and re-write the program showing the starting address of each line[2]:

```
8000 A9 01
8002 8D 00 40
8005 49 01
8007 4C ?? ??
```

Recall that the purpose of our "Jump" instruction (last line, `0x4C`) was to go back to the "Store Accumulator" instruction (`0x8D`) so that our recently XOR'd data will be re-written to the output port at address `0x4000`. Therefore, the address we need to jump to is `0x8002`. Knowing this, we may edit our machine code listing to include this address in the last instruction, in "little-endian" byte order because the model 6502 microprocessor happens to be a little-endian machine:

```
8000 A9 01
8002 8D 00 40
8005 49 01
8007 4C 02 80
```

If we were to write this program to the computer's ROM and then read it back to display in conventional "hex dump" format, it would look like this:

```
8000 A9 01 8D 00 40 49 01 4C 02 80
```

If the ROM IC(s) in our computer are socketed and therefore easily removed for programming by an external device, we could use a PROM programmer[3] to write this short program into a programmable ROM memory chip and then re-insert it into our computer's board to run.

[2] This is beginning to resemble the common "hex dump" memory display format, except that each line of text is limited to just one instruction

[3] Commercially-available PROM programming tools consist of a unit connected to a personal computer with a

3.1.2 Assembly code to blink an LED

Now that we have seen the "hand-assembly" method of creating a simple LED-blinking program for our 6502-based computer, let us explore how we could do the same using *assembly language*. Starting with the original program specification telling us what we need the computer to do:

1. Write the binary value `0b00000001` to address `0x4000`
2. Toggle the least-significant bit of that byte in address `0x4000`
3. Go back to second instruction and repeat indefinitely

Next, we refer to instructions available in the 6502's instruction set, but this time we write the *mnemonic* abbreviations given in that instruction set instead of opcodes:

1. `LDA` value `0b00000001`
2. `STA` to `0x4000`
3. `EOR` mask `0b00000001`
4. `JMP` to second instruction

We are almost done with our assembly-language program! All we need to do now is write it using the proper syntax[4] expected by our assembler software, being sure to include a *directive*[5] to the assembler to begin at address `0x8000`.

```
      .ORG  $8000
      LDA   #$01
LOOP
      STA   $4000
      EOR   #$01
      JMP   LOOP
```

This program you see above is completely functioning, ready to be assembled into machine code and written to the computer's ROM.

zero-insertion-force (ZIF) IC socket to facilitate easy plugging and unplugging of memory ICs. Software provided with the programmer allow you to type the hex dump data into an editor window (or into a plain-ASCII text file) and then have that data written to the memory IC with the click of a button or a command-line instruction typed into the personal computer. If you *really* desire a low-level learning experience, you can program the PROM chip by connecting address, data, and write-enable lines to toggle switches, then toggling those switches to specify addresses, data to be written to those addresses, and pressing the write-enable switch to "burn" that data into the chip when you are ready. Needless to say, the latter option is the most tedious.

[4] We are assuming here that our assembler does not understand binary notation and expects all numerical values to be expressed in hexadecimal instead.

[5] A "directive" is an instruction given not to the microprocessor, but rather to the assembler software. It tells the assembler to translate the assembly "source" code into machine code in some particular manner.

Optional *comments* help make our code easier to read:

```
      .ORG  $8000   ; Begin at address 0x8000
      LDA   #$01    ; Load 0x01 into Accumulator
LOOP                ; Mark the beginning of the loop
      STA   $4000   ; Store Accumulator value to Output port
      EOR   #$01    ; XOR the least-significant bit
      JMP   LOOP    ; Jump to the beginning of the loop
```

Take note of some important details of this assembly-language program:

- Order of execution is left-to-right, top-to-bottom in the same order as reading English-language text.
- Any left-justified text (e.g. `LOOP`) is considered a *label* rather than an *instruction*, whether on its own line or preceding an instruction.
- Instructions (e.g. `LDA`, `STA`, etc.) must be preceded by whitespace. The number of space or tab characters doesn't matter.
- Operands to those instructions (e.g. `$4000`, etc.) must be separated to the right of those instructions by some whitespace as well. The number of space or tab characters doesn't matter. In this listing I've adjusted the number of spaces to make the columns neatly align.
- Any text to the right of a semicolon (`;`) character is considered a comment, ignored by the assembler and not included in the machine code at all.
- Any instruction preceded by a dot (e.g. `.ORG`) is a *directive* to the assembler, telling how to do some aspect of the translation into machine code, but not appearing in the final machine code itself.
- Note how the Jump instruction's target address was resolved by the assembler, with no need for us to count address numbers to figure out where it should jump to. We simply place a label and let the assembler figure out those details for us.
- This assembler uses a dollar-symbol (`$`) to denote any hexadecimal value.
- The pound symbol (`#`) denotes an *immediate* value, meaning the literal number value specified. Otherwise, the operand value is considered to be a memory address location.

It should be noted that some of these conventions vary from assembler to assembler. For example, some assemblers require all labels to contain a colon at the end (e.g. `LOOP:` rather than `LOOP`). Some assemblers allow C-style hexadecimal notation (e.g. `0x4000`) while others insist on the `$` character. Some assemblers allow binary notation (e.g. `0b00000001` or `%00000001`) while others don't. Some assemblers require directives be preceded by a dot (e.g. `.ORG`) while others insist directives *not* be preceded by any character. As always when using software, refer to the manufacturer's documentation for details!

3.1.3 Slowing down the blinking

If we were to actually assemble and write this program to ROM, then start the computer to initiate program execution, we would likely find the LED blinking on and off so fast that it appeared to be steadily lit (albeit dimmer than usual). The reason for this is the fast fetch/execute cycle time of a typical microprocessor. A model 6502 running at a clock speed of 1 MHz would blink the LED on and off at a rate far too quick for the human eye to see[6].

In order to make the blinking rate slow enough to see, we must somehow *delay* the loop's repetition. One easy way to do this is to insert a "counting" loop inside of our program's blinking loop. This counting loop keeps the microprocessor occupied by doing nothing but sequentially counting, in order to purposely waste time and thereby delay its toggling of the LED output bit.

Here is a section of assembly code using common 6502 instructions to perform this delay task by forcing the microprocessor to count backwards from 255 (`0xFF`) until it reaches zero, complete with explanatory comments:

```
      LDX #$FF          ; Load value 0xFF into register X
DELAY_LOOP
      DEX               ; Decrement (subtract 1 from) value stored in X
      CPX #$00          ; Compare that value to zero
      BNE DELAY_LOOP    ; If unequal, "branch" to DELAY_LOOP to repeat
```

One way to incorporate this delay code into our program is to simply insert it "in-line" with the original code between the `EOR` and `JMP` instructions, like this:

```
      .ORG  $8000       ; Begin at address 0x8000
      LDA   #$01        ; Load 0x01 into Accumulator
LOOP                    ; Mark the beginning of the loop
      STA   $4000       ; Store Accumulator value to Output port
      EOR   #$01        ; XOR the least-significant bit
      LDX   #$FF        ; Load value 0xFF into register X
DELAY_LOOP
      DEX               ; Decrement (subtract 1 from) value stored in X
      CPX   #$00        ; Compare that value to zero
      BNE   DELAY_LOOP  ; If unequal, "branch" to DELAY_LOOP to repeat
      JMP   LOOP        ; Jump to the beginning of the loop
```

[6] According the the model 6502 manual,, the `STA` instruction requires 4 clock cycles, the `EOR` instruction 2 clock cycles, and the `JMP` instruction 3 clock cycles. This means 9 clock cycles would be required for every pass through the program, with two passes required for a full cycle of the LED's blink (i.e. on *and* off). Dividing 1 MHz by the 18 clock cycles necessary to fully cycle the LED gives an LED blinking frequency of 55.556 kHz!

While this solution works quite well, there is a more sophisticated way to achieve the same "loop within a loop" structure, and that is to place the delay-time code within its own *subroutine*. A "subroutine" is a section of code that stands apart from the rest, ready to be *called* by the main portion of the program whenever needed.

Subroutines are particularly useful when that code must be invoked at multiple points within the main program. Instead of copying-and-pasting the necessary code repeatedly in-line where needed, we simply insert a "call" or "jump to subroutine" instruction where it's needed and the microprocessor will jump to that new address (its Program Counter being preset as needed). Then, at the end of the subroutine we place a "return" instruction that tells the microprocessor to resume where it left off in the main program.

The following shows a listing of the assembly code for the slow-blinking program using a subroutine called `DELAY`[7]:

```
      .ORG  $8000       ; Begin at address 0x8000
      LDA   #$01        ; Load 0x01 into Accumulator
LOOP                    ; Mark the beginning of the loop
      STA   $4000       ; Store Accumulator value to Output port
      EOR   #$01        ; XOR the least-significant bit
      JSR   DELAY       ; Call the DELAY subroutine
      JMP   LOOP        ; Jump to the beginning of the loop

DELAY
      LDX   #$FF        ; Load value 0xFF into register X
DELAY_LOOP
      DEX               ; Decrement (subtract 1 from) value stored in X
      CPX   #$00        ; Compare that value to zero
      BNE   DELAY_LOOP  ; If unequal, "branch" to DELAY_LOOP and repeat
      RTS               ; Return to the main program
```

Admittedly the use of subroutines doesn't appear to be any better than simply inserting the `DELAY` code in-line with the original program. However, if we had a need to call this subroutine multiple times within our program, all we would need to add is another `JSR DELAY` instruction. Thus, the subroutine strategy becomes more efficient than the in-line strategy proportional to how many times that routine must execute.

[7] The blank line between `JMP LOOP` and `DELAY` is there for esthetic purposes only, to help our eyes see the distinction between the main program and the subroutine. We could eliminate this blank line (or add more!) and the program would execute just as well.

An useful tool provided by most assemblers is a *disassembly* option. This takes the assembled machine code and translates it "backwards" into assembly code, displaying the memory addresses, machine code hex dump, and equivalent assembly side-by-side for comparison:

```
Address    Hexdump    Disassembly
---------------------------------
$8000      A9 01      LDA #$01
$8002      8D 00 40   STA $4000
$8005      49 01      EOR #$01
$8007      20 0D 00   JSR $000D
$800A      4C 02 80   JMP $8002
$800D      A2 FF      LDX #$FF
$800F      CA         DEX
$8010      E0 00      CPX #$00
$8012      D0 FB      BNE $800F
$8014      60         RTS
```

Note how the "jump" and "branch" instructions all specify address locations in one way or another, but not the "return instruction" at the end of the subroutine. How does the microprocessor know which memory address to return to after completing the subroutine? The answer lies in a feature of the microprocessor called the *stack*: a section of volatile memory used by the processor to remember such things as previous Program Counter values. A jump-to-subroutine instruction causes the current memory address value held in the Program Counter to be "pushed" onto the stack before jumping to the subroutine's starting memory address. A "return" instruction causes the microprocessor to "pop" the former address value off the stack and into the Program Counter again, so that execution resumes right where it left off[8]. The model 6502 processor uses a portion of its RAM memory space for its stack (`0x0100` through `0x01FF`).

If you examine this disassembled code closely, you will notice something strange with the "branch-if-not-equal" instruction: the disassembled code says `BNE $800F` but the machine code does not actually contain the `0x800F` address. Instead, it only contains the opcode for `BNE` (`D0`) and a byte with a value of `0xFB`. This is an example of *relative addressing*, where the operand to the instruction declares not the address itself, but rather *how many addresses to skip, either forward or backward*. As an eight-bit signed number, `0xFB` is equal to negative five. This tells the microprocessor to decrement its Program Counter by five to repeat the `DELAY_LOOP`. If you count from address `0x8013` where the `0xFB` operand was resides to `0x800F` where the delay loop begins, you count five addresses (inclusive). Relative addressing is more efficient than absolute addressing because we only need one byte telling the instruction how far to jump instead of two bytes to specify a 16-bit address. Interestingly, the disassembler opted to show us an absolute address even though that's not really how the 6502's `BNE` instruction works.

[8] Stacks are analogous to a pile (stack) of paper notes. If a person is reading a book and they suddenly get told to turn to a different chapter to read a passage there, they may write the current page number on a note and "push" that note to the top of the stack so they won't forget it while turning to the new passage. After reading the new passage, the person retrieves their note from the top of the stack (i.e. "popping" it off the stack) and references it to return to the page where they left off. This may occur more than once, and the stack will "remember" not only the page numbers but also keep everything in the right order as the person eventually returns to their original place in the book.

3.1.4 Simplifying with symbols

Another technique useful for making our assembly-language programs easier to read and to maintain is the use of *symbols* to represent numerical values. Consider the last version of our LED-blinking program re-written to incorporate three symbols, defined at the very beginning of the code listing:

```
DTIME .EQU  $FF          ; Create a symbol "DTIME" for the delay time parameter
OUTPT .EQU  $4000        ; Create a symbol "OUTPT" for the Output port address
LED   .EQU  $01          ; Create a symbol "LED" for the LED's bit number

      .ORG  $8000        ; Begin at address 0x8000
      LDA   #LED         ; Load 0x01 into Accumulator
LOOP                     ; Mark the beginning of the loop
      STA   OUTPT        ; Store Accumulator value to Output port
      EOR   #LED         ; XOR the least-significant bit
      JSR   DELAY        ; Call the DELAY subroutine
      JMP   LOOP         ; Jump to the beginning of the loop

DELAY
      LDX   #DTIME       ; Load value 0xFF into register X
DELAY_LOOP
      DEX                ; Decrement (subtract 1 from) value stored in X
      CPX   #$00         ; Compare that value to zero
      BNE   DELAY_LOOP   ; If unequal, "branch" to DELAY_LOOP and repeat
      RTS                ; Return to the main program
```

The `.EQU` directive tells the assembler to treat the symbol (on the left) as an alias of the value (on the right). This lets us use lettered symbols within our code rather than numerical values for important parameters such as I/O addresses, delay time, etc. These symbols may be used as many times as desired, and they will always mean the same thing (e.g. the `LED` symbol is used twice in this program, and it means `$01` both times).

3.1.5 Using the stack

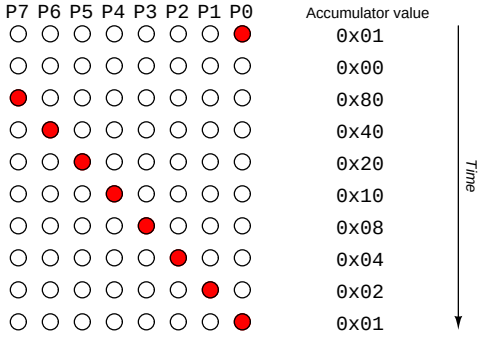
Previously we mentioned the microprocessor's *stack*, a section of RAM used to hold data in sequential order. Stacks may be thought of as a *Last-In First-Out* shift register, where data retrieved from the stack is in reverse order of how data is placed onto the stack. The analogy of a microprocessor's stack being a literal stack of paper sheets is helpful here: if we pull papers from the top of the stack, we will find their sequence is in reverse order of how we placed those sheets on the stack.

Microprocessors use their stack to manage subroutine calls, "pushing" the last Program Counter memory address to the stack prior to jumping to the subroutine's address, then "popping" that old address off the stack when the subroutine completes so it knows where to resume its previous place in the main program. This form of stack usage is automatic, being built-in to the finite state machine sequence as part of each subroutine "call" instruction and each subroutine "return" instruction. *Interrupts* also use a stack to remember where to jump back in the main program after completing the interrupt service routine (ISR).

Certain instructions exist to make use of the stack in ways that are not necessarily related to subroutines or interrupts, and these can be very useful. Here we will explore one such practical use of the stack. Consider this simple "chasing LED" program using the 6502's "Rotate Right" instruction to shift the place of a single "1" bit in the Accumulator byte, the goal being to create a "chasing" LED display on our computer where the light appears to repeatedly sweep along the row of LEDs:

```
      .ORG  $8000       ; Begin at address 0x8000
      LDA   #$01        ; Load 0x01 into Accumulator
LOOP                    ; Mark the beginning of the loop
      STA   $4000       ; Store Accumulator value to Output port
      ROR               ; Rotate Accumulator bits one place right
      JMP   LOOP        ; Jump to the beginning of the loop
```

This code is every bit as simple as our original LED-blinking program. When run, it produces the following pattern of light (sequence shown chronologically from top to bottom):



During the step where no LEDs are lit, the "1" bit resides in the *Carry* bit of the microprocessor's Status register, a special register used to store the results of certain mathematical and logical operations.

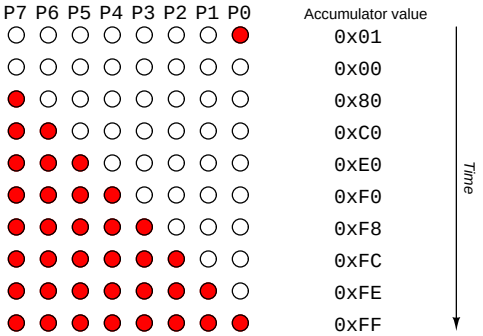
This pattern of light is what we expect the `ROR` instruction to produce after the Accumulator is initially loaded with `0x01`. All is well, except for the same problem we had with our original "blinking LED" program: the sequence runs too fast for our eyes to discern. It just looks like a blur of eight LEDs all (dimly) lit!

We already know how to slow programs down, by inserting a counting loop that "wastes" the microprocessor's time, so let's modify this program accordingly:

```
      .ORG  $8000       ; Begin at address 0x8000
      LDA   #$01        ; Load 0x01 into Accumulator
LOOP                    ; Mark the beginning of the loop
      STA   $4000       ; Store Accumulator value to Output port
      ROR               ; Rotate Accumulator bits one place right
      JSR   DELAY       ; Call the DELAY subroutine
      JMP   LOOP        ; Jump to the beginning of the loop

DELAY
      LDX   #DTIME      ; Load value 0xFF into register X
DELAY_LOOP
      DEX               ; Decrement (subtract 1 from) value stored in X
      CPX   #$00        ; Compare that value to zero
      BNE   DELAY_LOOP  ; If unequal, "branch" to DELAY_LOOP and repeat
      RTS               ; Return to the main program
```

However, when we run this program we get a different light sequence:



For some reason, the "0" states rotated off the LSB-end of the byte are becoming "1" states to fill the MSB. Recall that the `ROR` instruction draws from the *Carry* bit of the Status register to fill

the MSB at each iteration. A reasonable hypothesis is that something other than the ROR is setting the Carry bit.

Indeed something *does* act to set the Carry bit when we don't want it to: the "compare X" CPX instruction inside our *DELAY* subroutine. According to the 6502 instruction set manual, the CPX sets the Carry bit if ever the X register's value is equal to or greater than the value it's being compared against. In fact, this is how the BNE instruction knows when to branch: it checks the Status register which is updated by all mathematical, logical, and comparison instructions. Given the design of our time-delay subroutine where the X register begins at a large value and counts down toward the comparison value of zero, we are *guaranteed* to return from that subroutine with the Carry bit set.

This causes problems for our ROR instruction, which takes the "1" value left in the Carry bit from the subroutine's CPX instruction and adds it to our chasing light sequence, which we do not want. Somehow we need the ROR to act on the Carry bit *it* generated, not the *new* Carry bit generated by the subroutine's CPX instruction.

Our stack ends up being a simple solution to this problem. All we need to do is "push" the Status register's state to the stack prior to calling the subroutine, then "pop" that old data back off the stack and into the Status register again before the ROR instruction reads the Carry bit. In other words, we can use the stack as temporary storage for the Status bits, and recall those bits after the subroutine is done using the Status register for its own purposes.

All this requires is the addition of two new instructions surrounding the "jump to subroutine": a PHP instruction ("push processor status on stack") prior to the jump, and PLP instruction ("pull processor status from stack") after the jump:

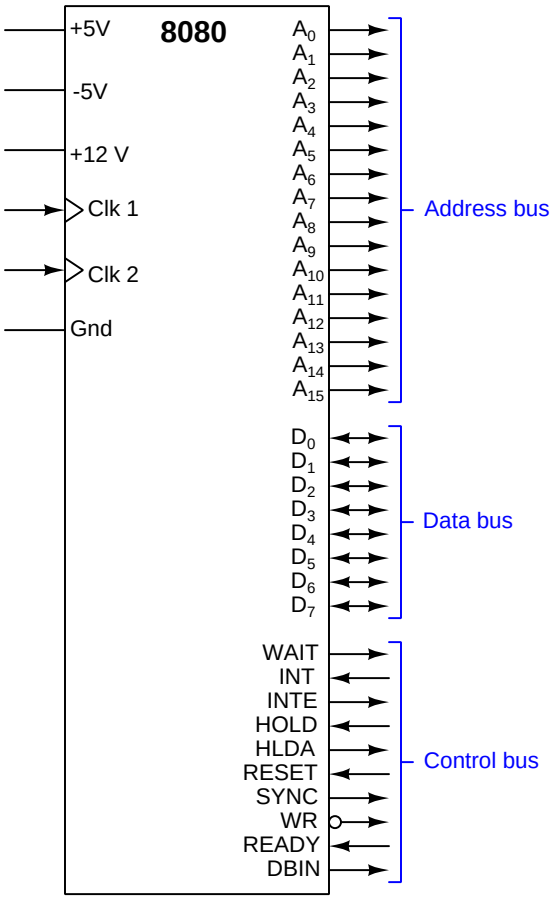
```
      .ORG  $8000       ; Begin at address 0x8000
      LDA   #$01        ; Load 0x01 into Accumulator
LOOP                    ; Mark the beginning of the loop
      STA   $4000       ; Store Accumulator value to Output port
      ROR               ; Rotate Accumulator bits one place right
      PHP               ; Push Status register to stack
      JSR   DELAY       ; Call the DELAY subroutine
      PLP               ; Pop Status register off stack
      JMP   LOOP        ; Jump to the beginning of the loop

DELAY
      LDX   #DTIME      ; Load value 0xFF into register X
DELAY_LOOP
      DEX               ; Decrement (subtract 1 from) value stored in X
      CPX   #$00        ; Compare that value to zero
      BNE   DELAY_LOOP  ; If unequal, "branch" to DELAY_LOOP and repeat
      RTS               ; Return to the main program
```

The stack is a very useful feature of any microprocessor, but it does have its limitations. If we push far more data onto the stack than we pop off, we can get a *stack overflow* where the oldest

data gets overwritten and is lost. Also, we need to be very careful that we push from the correct sources and pop to the correct destinations. For example, in the above program we pushed the Status register to the stack, then immediately after that the `JSR` instruction pushed the Program Counter value to the stack before going to the subroutine. When the subroutine completed, it popped the old Program Counter value off the stack, and then immediately after that we popped the old Status register off the stack. This works because those sources and destinations came in the correct sequence, and so the data going on and off the stack went where it should.

3.2 Intel 8080 microprocessor



Control bus pin functions are as follows:

- **WAIT** = this output acknowledges that the processor is in its "wait" state
- **INT** (Interrupt Request input) = receives an external signal to trigger an interrupt event
- **INTE** (Interrupt Enable output) = shows the state of the processor's internal interrupt flip-flop
- **HOLD** (Hold input) = an active signal applied here requests that the processor enter its "hold" state so that external devices may gain control of the address and data busses
- **HLDA** (Hold Acknowledge output) = indicates that the processor's address and bus lines are in the high-impedance state, allowing external devices to drive those busses
- **RESET** = this input forces the Program Counter to go to address `0x0000`

- **SYNC** (Synchronizing signal output) = activates at the beginning of each machine cycle
- **WR** (Write/Read output) = controls memory and I/O read/write operations
- **READY** = this input tells the processor when valid states exist on the data bus
- **DBIN** (Data Bus In output) = indicates when the processor is ready to receive data from memory or I/O

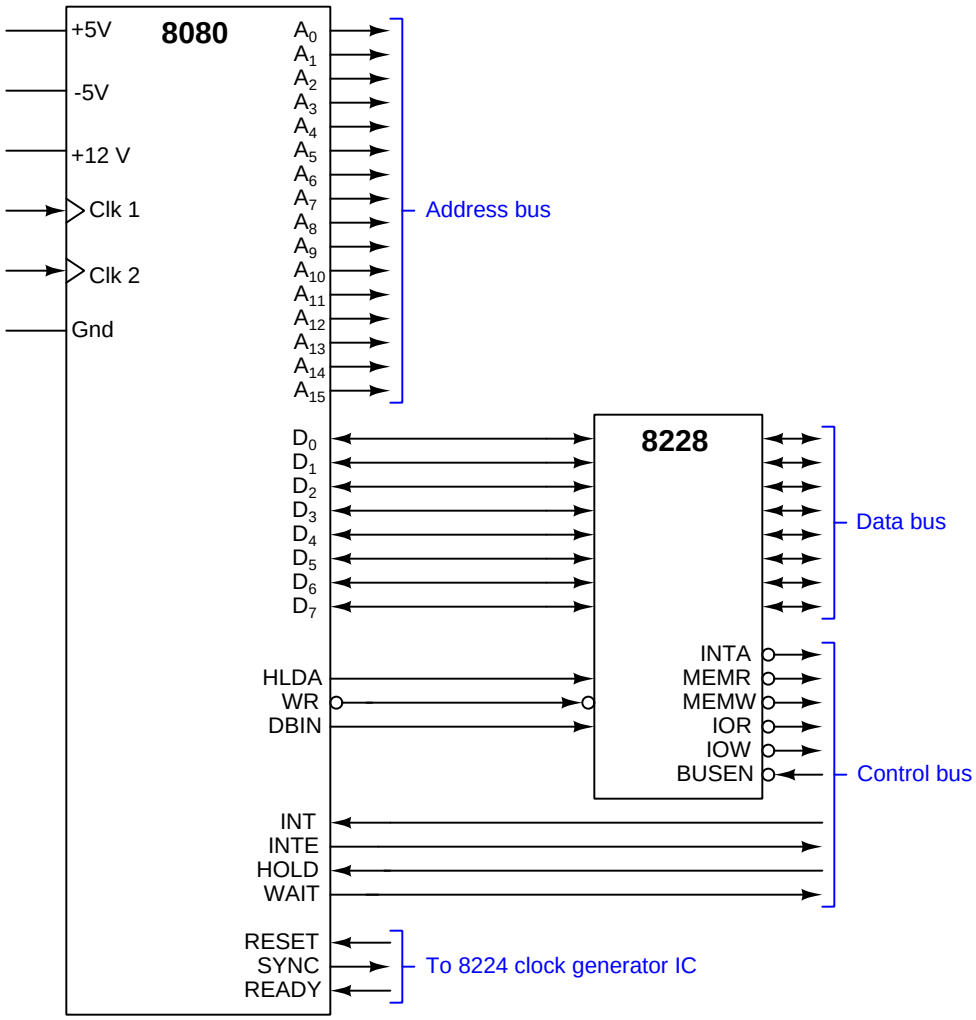
The 8080's control bus is actually more complicated than these ten pins would suggest. During the period of time when the **SYNC** output activates, the data bus lines output a *status word*. This is meant to be latched and decoded by a special-purpose IC called the Intel model 8228 "system controller". The function of each data bit in that status word is as follows:

- D_0 = **INTA** = acknowledges that a request for an interrupt has been made
- D_1 = **WO** = indicates a Write or an Output function, as opposed to a Read or Input function
- D_2 = **STACK** = indicates the address bus represents the stack address from the stack pointer
- D_3 = **HLTA** = acknowledges the HALT instruction
- D_4 = **OUT** = indicates the address bus is addressing an output device
- D_5 = **M1** = indicates the processor is fetching a new instruction
- D_6 = **INP** = indicates the address bus is addressing an input device
- D_7 = **MEMR** = indicates the data bus will be used to read data from memory

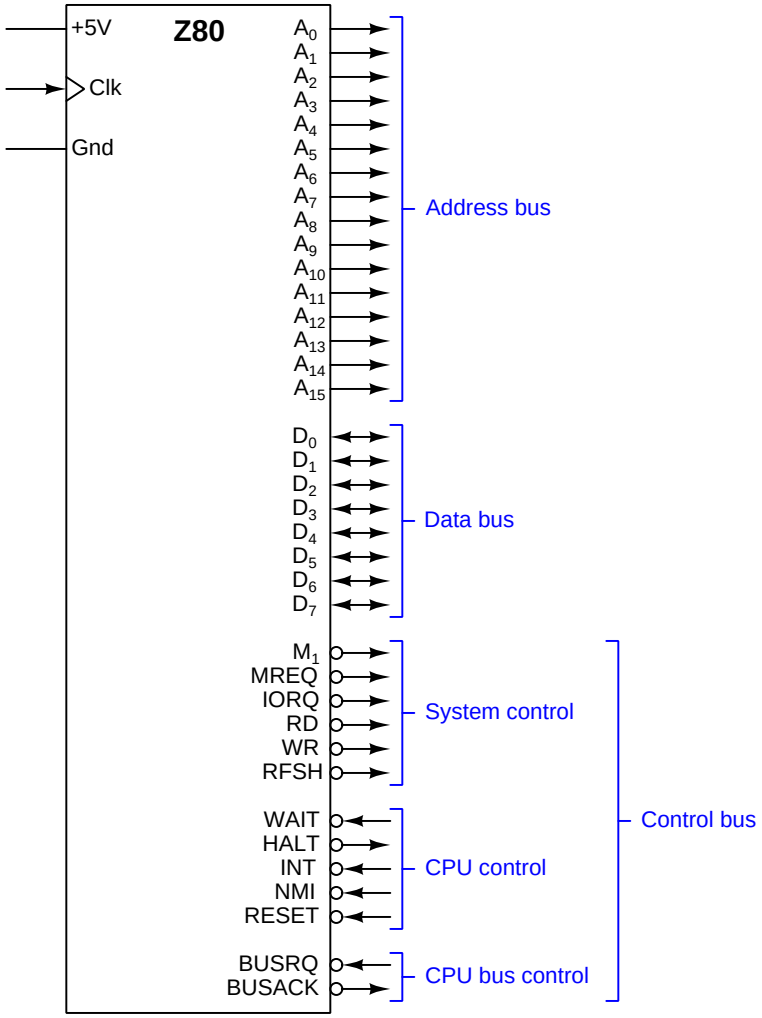
Outputs from the 8228 system controller included the following:

- **MEMR** (Memory Read output) = activates ROM/RAM memory ICs for a read cycle
- **MEMW** (Memory Write output) = activates RAM memory ICs for a write cycle
- **IOR** (I/O read output) = activates input port latches to read data
- **IOW** (I/O write output) = activates output port latches to write data
- **BUSEN** (Bus Enable input) = when disabled this forces data and control line buffers into high-impedance mode
- **INTA** (Interrupt acknowledge output) = acknowledges that a request for an interrupt has been made

The 8228 connects to the 8080 as shown below:



3.3 Zilog Z80 microprocessor



Compared to the Intel 8080, the Zilog Z80 has much more modest requirements for power supply and clock signal inputs. This frees up more pins in the 40-pin package for control bus lines, and makes it possible for the Z80 to directly drive a control bus with no need for a latch/decoder like the Intel 8228.

The following descriptive text comes from the 1984 US Patent 4,486,827 ("Microprocessor Apparatus") in which the authors give a rather complete description of the Z80's functionality. Although self-promoting in style (one of the patent's authors is Frederico Faggin, the primary inventor of the Z80 microprocessor), it is nevertheless educational for microprocessors in general.

Microcomputer systems are extremely simple to construct using Z-80 components. Any such system consists of three parts:

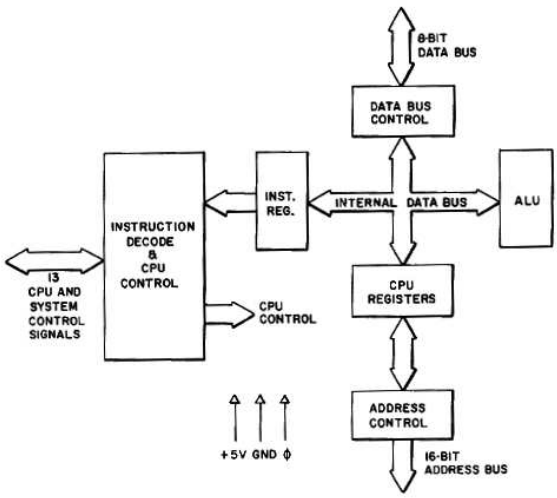
1. CPU (Central Processing Unit)

2. Memory

3. Interface Circuits to peripheral devices

The CPU is the heart of the system. Its function is to obtain instructions from the memory and perform the desired operations. The memory is used to contain instructions and in most cases data that is to be processed. For example, a typical instruction sequence may be to read data from a specific peripheral device, store it in a location in memory, check the parity and write it out to another peripheral device. Note that the Zilog component set includes the CPU and various general purpose I/O device controllers, while a wide range of memory devices may be used from any source. Thus, all required components can be connected together in a very simple manner with virtually no other external logic. The user's effort then becomes primarily one of software development. That is, the user can concentrate on describing his problem and translating it into a series of instructions that can be loaded into the microcomputer memory. Zilog is dedicated to making this step of software generation as simple as possible. A good example of this is our assembly language in which a simple mnemonic is used to represent every instruction that the CPU can perform. This language is self documenting in such a way that from the mnemonic the user can understand exactly what the instruction is doing without constantly checking back to a complex cross listing.

FIG. 1 shows a block diagram of the CPU, showing all of its major elements (digital devices).



CPU BLOCK DIAGRAM

FIG. 1

CPU REGISTERS

The Z-80 CPU contains 208 bits of R/W memory that are accessible to the programmer. FIG. 2 illustrates how this memory is configured into eighteen 8-bit registers and four 16-bit registers. All Z-80registers are implemented using static RAM. The registers include two sets of six general purpose registers that may be used individually as 8-bit registers or in pairs as 16-bit registers. These are also two sets of accumulator and flag registers.

Special Purpose Registers

1. Program Counter (PC). The program counter holds the l6-bit address of the current instruction being fetched from memory. The PC is automatically incremented after its contents have been transferred to the address lines. When a program jump occurs the new value is automatically placed in the PC, overriding the incrementer.

2. Stack Pointer (SP). The stack pointer holds the 16-bit address of the current top of a stack located anywhere in external system RAM memory. The external stack memory is organized as a last-in first-out (LIFO) file. Data can be pushed onto the stack from specific CPU registers or popped off of the stack into specific CPU registers through the execution of PUSH and POP instructions. The data popped from the stack is always the last data pushed onto it. The stack allows simple implementation of multiple level interrupts, unlimited subroutine nesting and simplification of many types of data manipulation.

3. Two Index Registers (IX & IY). The two independent index registers hold a 16-bit base address that is used in indexed addressing modes. In this mode, an index registers is used as a base to point to a region in memory from which data is to be stored or retrieved. An additional byte is included in indexed instructions to specify a displacement from this

base. This displacement is specified as a two's complement signed integer. This mode of addressing greatly simplifies many types of programs, especially where tables of data are used.

4. Interrupt Page Address Register (I). The Z-80 CPU can be operated in a mode where an indirect call to any memory location can be achieved in response to an interrupt. The I Register is used for this purpose to store the high order 8-bits of the indirect address while the interrupting device provides the lower 8-bits of the address. This feature allows interrupt routines to be dynamically located anywhere in memory with absolute minimal access time to the routine.

5. Memory Refresh Register (R). The Z-80 CPU contains a memory refresh counter to enable dynamic memories to be used with the same ease as static memories. This 7-bit register is automatically incremented after each instruction fetch. The data in the refresh counter is sent out on the lower portion of the address bus along with a refresh control signal while the CPU is decoding and executing the fetched instruction. This mode of refresh is totally transparent to the programmer and does not slow down the CPU operation. The programmer can load the R register for testing purposes, but this register is normally not used by the programmer.

Accumulator and Flag Registers

The CPU includes two independent 8-bit accumulators and associated 8-bit flag registers. The accumulator holds the results of 8-bit arithmetic or logical operations while the flag register indicates specific conditions for 8 or 16-bit operations, such as indicating whether or not the result of an operation is equal to zero. The programmer selects the accumulator and flag pair that he wishes to work with with a single exchange instruction so that he may easily work with either pair.

General Purpose Registers

There are two matched sets of general purpose registers, each set containing six 8-bit registers that may be used individually as 8-bit registers or as 16-bit register pairs by the programmer. One set is called BC, DE and III while the complementary set is called BC', DE' and III'. At any one time the programmer can select either set of registers to work with through a single exchange command for the entire set. In systems where last inter- [page 16]

rupt response is required, one set of general purpose registers and an accumulator/flag register may be reserved for handling this very last routine. Only a simple exchange commands need be executed to go between the routines. This greatly reduces interrupt service time by eliminating the requirement for saving and retrieving register contents in the external stack during interrupt or subroutine processing. These general purpose registers are used for a wide range of applications by the programmer. They also simplify programming, especially in ROM based systems where little external read/write memory is available.

ARITHMETIC & LOGIC UNIT (ALU)

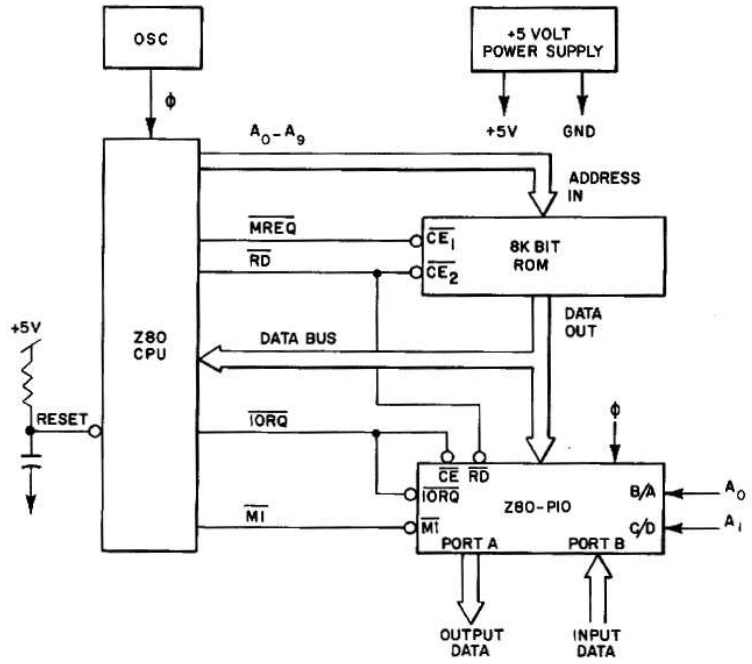
The 8-bit arithmetic and logical instructions of the CPU are executed in the ALU. Internally the ALU communicates with the registers and the external data bus on the internal data bus. The type of functions performed by the ALU include:

Add	Left or right shifts or rotates (arithmetic and logical
Subtract	Increment
Logical AND	Decrement
Logical OR	Set bit
Logical Exclusive OR	Reset bit
Compare	Test bit

INSTRUCTION REGISTER AND CPU CONTROL

As each instruction is fetched from memory, it is placed in the instruction register and decoded. The control sections performs this function and then generates and supplies all of the control signals necessary to read or write data from or to the registers, control the ALU and provide all required external control signals.

FIG. 3 shows a block diagram of a very simple digital processor system using the CPU. In a practical system the following five elements are required: power supply, oscillator (a source of clock signals), memory devices, I/O circuits, and the CPU.



MINIMUM COMPUTER SYSTEM

FIG. 3

Since the Z80-CPU only requires a single 5 volt supply, most small systems can be implemented using only this single supply.

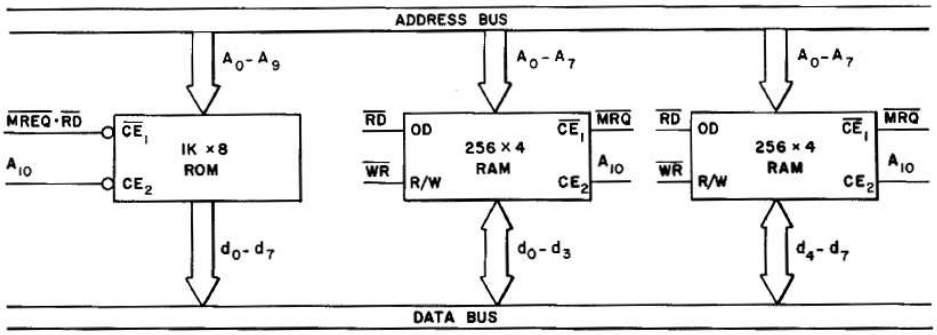
The oscillator can be very simple since the only requirement is that it be a 5 volt square wave. For systems not running at full speed, a simple RC oscillator can be used. When the CPU is operated near the highest possible frequency, a crystal oscillator is generally required because the system timing will not tolerate the drift or jitter that an RC network will generate. A crystal oscillator can be made from inverters and a few discrete components or monolithic circuits are widely available.

The external memory can be any mixture of standard RAM, ROM, or PROM. In this simple example we have shown a single 8K bit ROM (1K bytes) being utilized as the entire memory system. For this example we have assumed that the Z-80 internal register configuration contains sufficient Read/Write storage so that external RAM memory is not required.

Every computer system requires I/O circuits to allow it to interface to the "read world." In this simple example it is assumed that the output is an 8 bit control vector and the input is an 8 bit status word. The input data could be gated onto the data bus using any standard tri-state driver while the output data could be latched with any type of standard TTL latch. For this example we have used a Z80-PIO for the I/O circuit. This single circuit attaches to the data bus as shown and provides the required 16 bits of TTL compatible I/O. (Refer to the Z80-PIO manual for details on the operation of this circuit.) Notice in this example that with only three LSI circuits, a simple oscillator and a single 5 volt power supply, a powerful computer has been implemented.

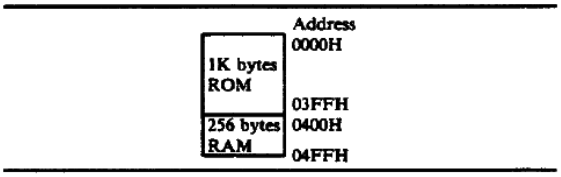
ADDING RAM

Most computer systems require some amount of external Read/Write memory for data storage and to implement a "stack." FIG. 4 illustrates how 256 bytes of static memory can be added to the previous example. In this example the memory space is assumed to be organized as follows:



ROM & RAM IMPLEMENTATION EXAMPLE

FIG. 4



In this diagram the address space is described in hexadecimal notation. For this example, address bit A_{10} separates the ROM space from the RAM space so that it can be used for the chip select function. For larger amounts of external ROM or RAM, a simple TTL decoder will be required to form the chip selects.

CPU TIMING

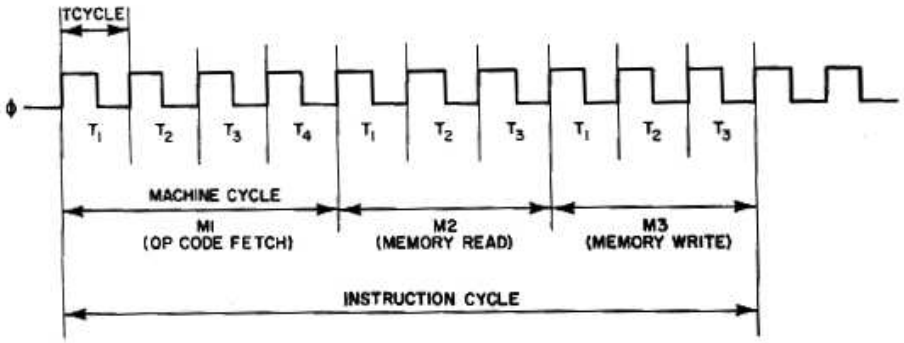
The Z-80 CPU executes instructions by stepping through a very precise set of a few basic operations. These include:

Memory read or write

I/O device read or write

Interrupt acknowledge

All instructions are merely a series of these basic operations. Each of these basic operations can take from three to six clock periods to complete or they can be lengthened to synchronize the CPU to the speed of external devices. The basic clock periods are referred to as T cycles and the basic operations are referred to as M (for machine) cycles. FIG. 5 illustrates how a typical instruction will be merely a series of specific M and T cycles.



BASIC CPU TIMING EXAMPLE
FIG. 5

Notice that this instruction consists of three machine cycles (M1, M2 and M3). The first machine cycle of any instruction is a fetch cycle which is four, five or six T cycles long (unless lengthened by the wait signal which will be fully described in the next section). The fetch cycle (M1) is used to fetch the OP code of the next instruction to be executed. Subsequent machine cycles move data between the CPU and memory or I/O devices and they may have anywhere from three to five T cycles (again they may be lengthened by wait states to synchronize the external devices to the CPU). The following paragraphs describe the timing which occurs within any of the basic machine cycles. In section 10, the exact timing for each instruction is specified.

The CPU of the present invention is particularly useful in a hardware/software development system. Such systems, typically employing microprocessors, have heretofore required two or more CPU's thus requiring additional logic and memory space at greatly added cost. [page 17]

Chapter 4

Questions

This learning module, along with all others in the ModEL collection, is designed to be used in an inverted instructional environment where students independently read[1] the tutorials and attempt to answer questions on their own *prior* to the instructor's interaction with them. In place of lecture[2], the instructor engages with students in Socratic-style dialogue, probing and challenging their understanding of the subject matter through inquiry.

Answers are not provided for questions within this chapter, and this is by design. Solved problems may be found in the Tutorial and Derivation chapters, instead. The goal here is *independence*, and this requires students to be challenged in ways where others cannot think for them. Remember that you always have the tools of *experimentation* and *computer simulation* (e.g. SPICE) to explore concepts!

The following lists contain ideas for Socratic-style questions and challenges. Upon inspection, one will notice a strong theme of *metacognition* within these statements: they are designed to foster a regular habit of examining one's own thoughts as a means toward clearer thinking. As such these sample questions are useful both for instructor-led discussions as well as for self-study.

[1] Technical reading is an essential academic skill for any technical practitioner to possess for the simple reason that the most comprehensive, accurate, and useful information to be found for developing technical competence is in textual form. Technical careers in general are characterized by the need for continuous learning to remain current with standards and technology, and therefore any technical practitioner who cannot read well is handicapped in their professional development. An excellent resource for educators on improving students' reading prowess through intentional effort and strategy is the book textitReading For Understanding – How Reading Apprenticeship Improves Disciplinary Learning in Secondary and College Classrooms by Ruth Schoenbach, Cynthia Greenleaf, and Lynn Murphy.

[2] Lecture is popular as a teaching method because it is easy to implement: any reasonably articulate subject matter expert can talk to students, even with little preparation. However, it is also quite problematic. A good lecture always makes complicated concepts seem easier than they are, which is bad for students because it instills a false sense of confidence in their own understanding; reading and re-articulation requires more cognitive effort and serves to verify comprehension. A culture of teaching-by-lecture fosters a debilitating dependence upon direct personal instruction, whereas the challenges of modern life demand independent and critical thought made possible only by gathering information and perspectives from afar. Information presented in a lecture is ephemeral, easily lost to failures of memory and dictation; text is forever, and may be referenced at any time.

General challenges following tutorial reading

- Summarize as much of the text as you can in one paragraph of your own words. A helpful strategy is to explain ideas as you would for an intelligent child: as simple as you can without compromising too much accuracy.
- Simplify a particular section of the text, for example a paragraph or even a single sentence, so as to capture the same fundamental idea in fewer words.
- Where did the text make the most sense to you? What was it about the text's presentation that made it clear?
- Identify where it might be easy for someone to misunderstand the text, and explain why you think it could be confusing.
- Identify any new concept(s) presented in the text, and explain in your own words.
- Identify any familiar concept(s) such as physical laws or principles applied or referenced in the text.
- Devise a proof of concept experiment demonstrating an important principle, physical law, or technical innovation represented in the text.
- Devise an experiment to disprove a plausible misconception.
- Did the text reveal any misconceptions you might have harbored? If so, describe the misconception(s) and the reason(s) why you now know them to be incorrect.
- Describe any useful problem-solving strategies applied in the text.
- Devise a question of your own to challenge a reader's comprehension of the text.

General follow-up challenges for assigned problems

- Identify where any fundamental laws or principles apply to the solution of this problem, especially before applying any mathematical techniques.
- Devise a thought experiment to explore the characteristics of the problem scenario, applying known laws and principles to mentally model its behavior.
- Describe in detail your own strategy for solving this problem. How did you identify and organized the given information? Did you sketch any diagrams to help frame the problem?
- Is there more than one way to solve this problem? Which method seems best to you?
- Show the work you did in solving this problem, even if the solution is incomplete or incorrect.
- What would you say was the most challenging part of this problem, and why was it so?
- Was any important information missing from the problem which you had to research or recall?
- Was there any extraneous information presented within this problem? If so, what was it and why did it not matter?
- Examine someone else's solution to identify where they applied fundamental laws or principles.
- Simplify the problem from its given form and show how to solve this simpler version of it. Examples include eliminating certain variables or conditions, altering values to simpler (usually whole) numbers, applying a limiting case (i.e. altering a variable to some extreme or ultimate value).
- For quantitative problems, identify the real-world meaning of all intermediate calculations: their units of measurement, where they fit into the scenario at hand. Annotate any diagrams or illustrations with these calculated values.
- For quantitative problems, try approaching it qualitatively instead, thinking in terms of "increase" and "decrease" rather than definite values.
- For qualitative problems, try approaching it quantitatively instead, proposing simple numerical values for the variables.
- Were there any assumptions you made while solving this problem? Would your solution change if one of those assumptions were altered?
- Identify where it would be easy for someone to go astray in attempting to solve this problem.
- Formulate your own problem based on what you learned solving this one.

General follow-up challenges for experiments or projects

- In what way(s) was this experiment or project easy to complete?
- Identify some of the challenges you faced in completing this experiment or project.

- Show how thorough documentation assisted in the completion of this experiment or project.
- Which fundamental laws or principles are key to this system's function?
- Identify any way(s) in which one might obtain false or otherwise misleading measurements from test equipment in this system.
- What will happen if (component X) fails (open/shorted/etc.)?
- What would have to occur to make this system unsafe?

4.1 Conceptual reasoning

These questions are designed to stimulate your analytic and synthetic thinking[3]. In a Socratic discussion with your instructor, the goal is for these questions to prompt an extended dialogue where assumptions are revealed, conclusions are tested, and understanding is sharpened. Your instructor may also pose additional questions based on those assigned, in order to further probe and refine your conceptual understanding.

Questions that follow are presented to challenge and probe your understanding of various concepts presented in the tutorial. These questions are intended to serve as a guide for the Socratic dialogue between yourself and the instructor. Your instructor's task is to ensure you have a sound grasp of these concepts, and the questions contained in this document are merely a means to this end. Your instructor may, at his or her discretion, alter or substitute questions for the benefit of tailoring the discussion to each student's needs. The only absolute requirement is that each student is challenged and assessed at a level equal to or greater than that represented by the documented questions.

It is far more important that you convey your *reasoning* than it is to simply convey a correct answer. For this reason, you should refrain from researching other information sources to answer questions. What matters here is that *you* are doing the thinking. If the answer is incorrect, your instructor will work with you to correct it through proper reasoning. A correct answer without an adequate explanation of how you derived that answer is unacceptable, as it does not aid the learning or assessment process.

You will note a conspicuous lack of answers given for these conceptual questions. Unlike standard textbooks where answers to every other question are given somewhere toward the back of the book, here in these learning modules students must rely on other means to check their work. The best way by far is to debate the answers with fellow students and also with the instructor during the Socratic dialogue sessions intended to be used with these learning modules. Reasoning through challenging questions with other people is an excellent tool for developing strong reasoning skills.

Another means of checking your conceptual answers, where applicable, is to use circuit simulation software to explore the effects of changes made to circuits. For example, if one of these conceptual questions challenges you to predict the effects of altering some component parameter in a circuit, you may check the validity of your work by simulating that same parameter change within software and seeing if the results agree.

[3] *Analytical* thinking involves the "disassembly" of an idea into its constituent parts, analogous to dissection. *Synthetic* thinking involves the "assembly" of a new idea comprised of multiple concepts, analogous to construction. Both activities are high-level cognitive skills, extremely important for effective problem-solving, necessitating frequent challenge and regular practice to fully develop.

4.1.1 Reading outline and reflections

"Reading maketh a full man; conference a ready man; and writing an exact man" – Francis Bacon

Francis Bacon's advice is a blueprint for effective education: reading provides the learner with knowledge, writing focuses the learner's thoughts, and critical dialogue equips the learner to confidently communicate and apply their learning. Independent acquisition and application of knowledge is a powerful skill, well worth the effort to cultivate. To this end, students should read these educational resources closely, journal their own reflections on the reading, and discuss in detail their findings with classmates and instructor(s). You should be able to do all of the following after reading any instructional text:

☑ Briefly SUMMARIZE THE TEXT in the form of a journal entry documenting your learning as you progress through the course of study. Share this summary in dialogue with your classmates and instructor. Journaling is an excellent self-test of thorough reading because you cannot clearly express what you have not read or did not comprehend.

☑ Demonstrate ACTIVE READING STRATEGIES, including verbalizing your impressions as you read, simplifying long passages to convey the same ideas using fewer words, annotating text and illustrations with your own interpretations, working through mathematical examples shown in the text, cross-referencing passages with relevant illustrations and/or other passages, identifying problem-solving strategies applied by the author, etc. Technical reading is a special case of problem-solving, and so these strategies work precisely because they help solve any problem: paying attention to your own thoughts (metacognition), eliminating unnecessary complexities, identifying what makes sense, paying close attention to details, drawing connections between separated facts, and noting the successful strategies of others.

☑ Identify IMPORTANT THEMES, especially GENERAL LAWS and PRINCIPLES, expounded in the text and express them in the simplest of terms as though you were teaching an intelligent child. This emphasizes connections between related topics and develops your ability to communicate complex ideas to anyone.

☑ Form YOUR OWN QUESTIONS based on the reading, and then pose them to your instructor and classmates for their consideration. Anticipate both correct and incorrect answers, the incorrect answer(s) assuming one or more plausible misconceptions. This helps you view the subject from different perspectives to grasp it more fully.

☑ Devise EXPERIMENTS to test claims presented in the reading, or to disprove misconceptions. Predict possible outcomes of these experiments, and evaluate their meanings: what result(s) would confirm, and what would constitute disproof? Running mental simulations and evaluating results is essential to scientific and diagnostic reasoning.

☑ Specifically identify any points you found CONFUSING. The reason for doing this is to help diagnose misconceptions and overcome barriers to learning.

4.1.2 Foundational concepts

Correct analysis and diagnosis of electric circuits begins with a proper understanding of some basic concepts. The following is a list of some important concepts referenced in this module's full tutorial. Define each of them in your own words, and be prepared to illustrate each of these concepts with a description of a practical example and/or a live demonstration.

Logic function

OR function

AND function

Truth table

NOT function

NOR function

NAND function

XOR function

Binary

Word width

Hexadecimal

ASCII

Hex dump

Unicode

Encoding

Decoding

Multiplexing

Demultiplexing

Latch

Bistable

Flip-flop

Register

Active-low versus Active-high

Clocking

Bus

Bit-shifting

Serial versus Parallel data

Stack

Arithmetic Logic Unit (ALU)

Opcode

Operand

Assembly language

Microprocessor

Program counter

Functional composition

Read versus Write

Memory address

Memory data

Volatility

RAM versus ROM

Enable

Pseudocode

Instruction

Fetch/Execute cycle

Bus contention

Memory map

Bus notation

Machine code

Assembly code

Assembler

Interpreter

Symbolic name

Address resolution

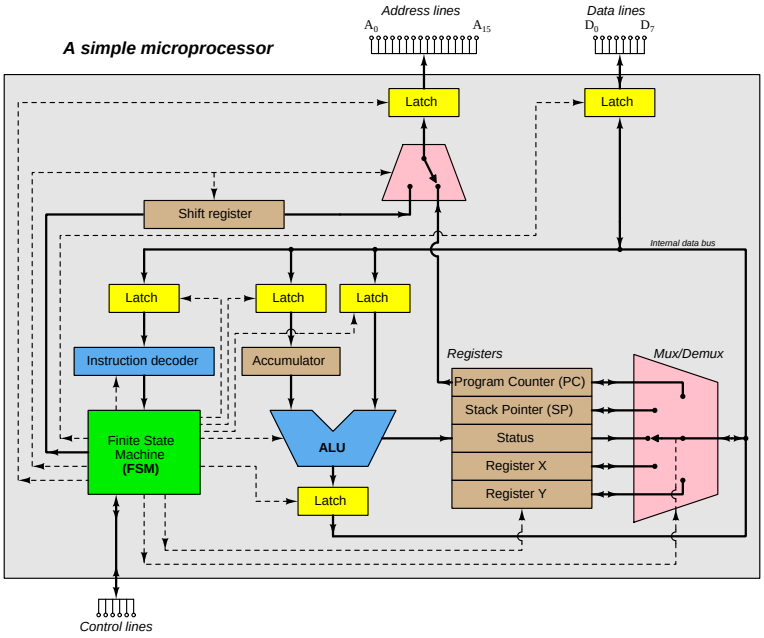
Microcontroller


Interrupt
Mask
Collision

4.1.3 Intel 8080 architecture

Locate a datasheet or user manual for the legacy *Intel model 8080* or *Intel model 8080A* microprocessor, and within that document locate a block diagram showing the internal architecture of the device.

Compare this block diagram against the simple microprocessor architecture shown here, and answer the following questions:



- Where is the Finite State Machine module represented within the 8080?
- How many registers are there inside the 8080?
- Identify what is unique about the W and Z register pair.

Challenges

- Would we classify the Intel 8080 as a 4-bit, 8-bit, 16-bit, 32-bit, or 64-bit microprocessor?
- Identify the power supply requirements for this microprocessor.

4.1.4 Intel 8080 processor cycles

Locate a user manual for the legacy *Intel model 8080* or *Intel model 8080A* microprocessor, and within that document locate the section discussing the *processor cycle*.

Define and differentiate the following terms, which are common to all microprocessors (not just the 8080):

- Instruction cycle
- Machine cycle (M-cycle)
- States (T-states)

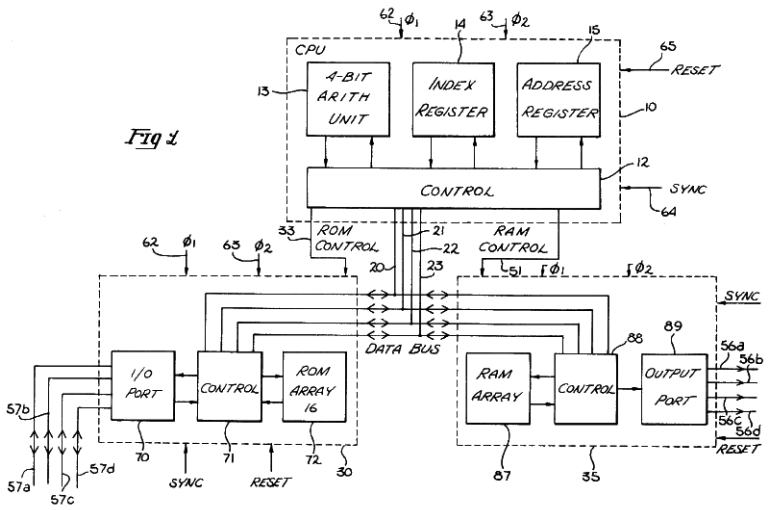
Explain how these terms relate to the Tutorial's presentation of a simple microprocessor executing a three-instruction *Read-Add-Write* program.

Challenges

- The Intel 8080 microprocessor outputs status information on its data bus at the beginning of every machine cycle (during the time the $SYNC$ output line is high). Identify some of the D_0-D_7 states expected for different types of machine cycles.
- The very first machine cycle (M_1) always represents the same operation – what is this operation, and why is it always the first one in an instruction cycle?

4.1.5 Early microprocessor timing diagram

US Patent number 3,821,715 ("Memory System for a Multi-Chip Digital Computer") granted in June 1974 describes a method for interfacing memory and I/O devices to a simple microprocessor. A simplified diagram of the circuit appears in Figure 1 of the patent:



Also included in this patent is Figure 2 showing signal timing for a single instruction cycle of this computer:

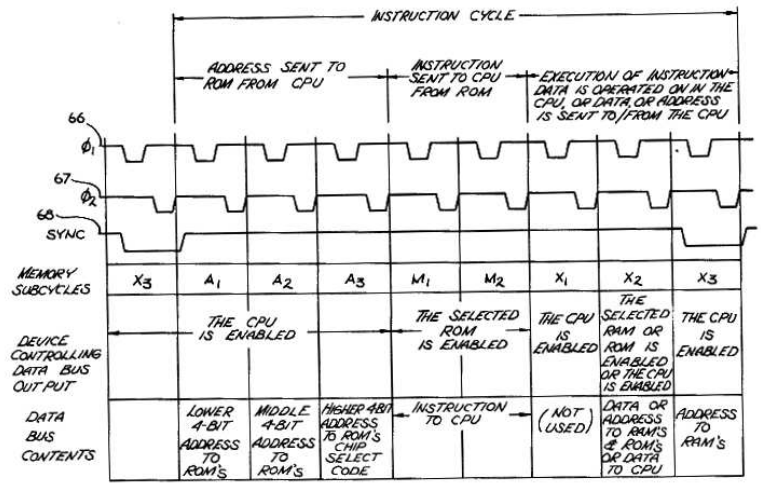


Fig. 2

Identify the M-cycles and T-states in this timing diagram.

Where in this timing diagram is the instruction being fetched?

Where in this timing diagram is the instruction being executed?

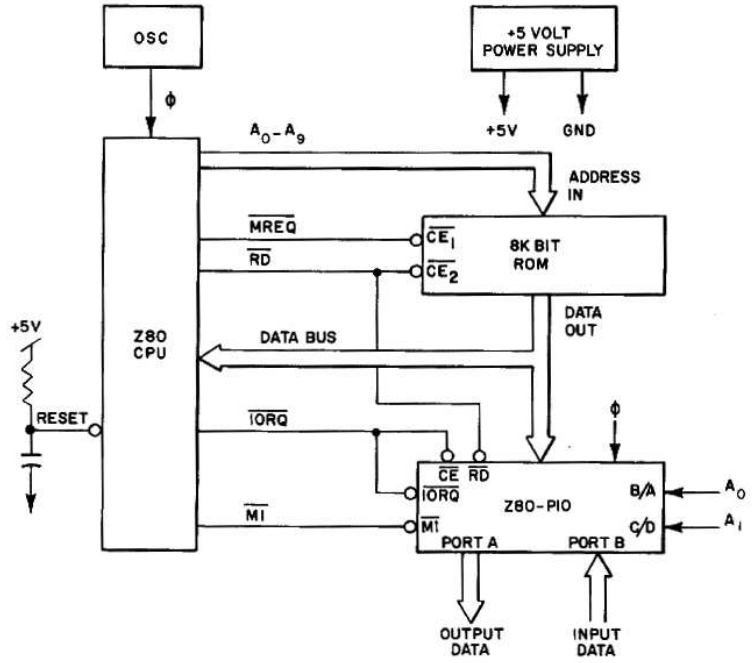
Note that this early computer design did not have separate address and data busses. How was this microprocessor able to properly read and write from memory using just one bus?

Challenges

- Why do you suppose modern microprocessors provide separate address and data bus lines?

4.1.6 Minimal Z80 computer

US Patent number 4,486,827 ("Microprocessor Apparatus") granted in December 1984 describes a development system based on Zilog's model Z80 microprocessor, a rival[4] to the Intel model 8080 microprocessor. In this patent the authors provide a diagram of a "minimal" computer for the Z80:



MINIMUM COMPUTER SYSTEM

FIG. 3

Identify directions of information "flow" on both the address and data busses.

Explain how you would enter a program for this simple computer to execute.

How can a computer such as this function with no RAM memory?

Challenges

- Explain how the RC network acts to reset the computer upon power-up.
- Explain why the $\overline{M_1}$ output line from the microprocessor connects to the Z80-PIO chip.

[4]The Z80 was largely developed by a former Intel employee (Frederico Faggin) and designed to be backward-compatible with the 8080. It offered a simpler interface design for external ICs (e.g. ROM, RAM), simpler power supply requirements, and also simpler clock requirements than the Intel 8080.

4.2 Quantitative reasoning

These questions are designed to stimulate your computational thinking. In a Socratic discussion with your instructor, the goal is for these questions to reveal your mathematical approach(es) to problem-solving so that good technique and sound reasoning may be reinforced. Your instructor may also pose additional questions based on those assigned, in order to observe your problem-solving firsthand.

Mental arithmetic and estimations are strongly encouraged for all calculations, because without these abilities you will be unable to readily detect errors caused by calculator misuse (e.g. keystroke errors).

You will note a conspicuous lack of answers given for these quantitative questions. Unlike standard textbooks where answers to every other question are given somewhere toward the back of the book, here in these learning modules students must rely on other means to check their work. My advice is to use circuit simulation software such as SPICE to check the correctness of quantitative answers. Refer to those learning modules within this collection focusing on SPICE to see worked examples which you may use directly as practice problems for your own study, and/or as templates you may modify to run your own analyses and generate your own practice problems.

Completely worked example problems found in the Tutorial may also serve as "test cases[5]" for gaining proficiency in the use of circuit simulation software, and then once that proficiency is gained you will never need to rely[6] on an answer key!

[5] In other words, set up the circuit simulation software to analyze the same circuit examples found in the Tutorial. If the simulated results match the answers shown in the Tutorial, it confirms the simulation has properly run. If the simulated results disagree with the Tutorial's answers, something has been set up incorrectly in the simulation software. Using every Tutorial as practice in this way will quickly develop proficiency in the use of circuit simulation software.

[6] This approach is perfectly in keeping with the instructional philosophy of these learning modules: *teaching students to be self-sufficient thinkers*. Answer keys can be useful, but it is even more useful to your long-term success to have a set of tools on hand for checking your own work, because once you have left school and are on your own, there will no longer be "answer keys" available for the problems you will have to solve.

4.2.1 Miscellaneous physical constants

Note: constants shown in **bold** type are *exact*, not approximations. Values inside of parentheses show one standard deviation (σ) of uncertainty in the final digits: for example, the magnetic permeability of free space value given as $1.25663706212(19) \times 10^{-6}$ H/m represents a center value (i.e. the location parameter) of $1.25663706212 \times 10^{-6}$ Henrys per meter with one standard deviation of uncertainty equal to $0.0000000000019 \times 10^{-6}$ Henrys per meter.

Avogadro's number (N_A) = **6.02214076** $\times 10^{23}$ **per mole** (mol^{-1})

Boltzmann's constant (k) = **1.380649** $\times 10^{-23}$ **Joules per Kelvin** (J/K)

Electronic charge (e) = **1.602176634** $\times 10^{-19}$ **Coulomb** (C)

Faraday constant (F) = **96,485.33212...** $\times 10^{4}$ **Coulombs per mole** (C/mol)

Magnetic permeability of free space (μ_0) = $1.25663706212(19) \times 10^{-6}$ Henrys per meter (H/m)

Electric permittivity of free space (ϵ_0) = $8.8541878128(13) \times 10^{-12}$ Farads per meter (F/m)

Characteristic impedance of free space (Z_0) = 376.730313668(57) Ohms (Ω)

Gravitational constant (G) = $6.67430(15) \times 10^{-11}$ cubic meters per kilogram-seconds squared ($m^3/kg\text{-}s^2$)

Molar gas constant (R) = **8.314462618... Joules per mole-Kelvin** (J/mol-K) = 0.08205746(14) liters-atmospheres per mole-Kelvin

Planck constant (h) = **6.62607015** $\times 10^{-34}$ **joule-seconds** (J-s)

Stefan-Boltzmann constant (σ) = **5.670374419...** $\times 10^{-8}$ **Watts per square meter-Kelvin**^{4} ($W/m^2 \cdot K^4$)

Speed of light in a vacuum (c) = **299,792,458 meters per second** (m/s) = 186282.4 miles per second (mi/s)

Note: All constants taken from NIST data "Fundamental Physical Constants – Complete Listing", from `http://physics.nist.gov/constants`, National Institute of Standards and Technology (NIST), 2018 CODATA Adjustment.

4.2.2 Introduction to spreadsheets

A powerful computational tool you are encouraged to use in your work is a *spreadsheet*. Available on most personal computers (e.g. Microsoft Excel), *spreadsheet* software performs numerical calculations based on number values and formulae entered into cells of a grid. This grid is typically arranged as lettered columns and numbered rows, with each cell of the grid identified by its column/row coordinates (e.g. cell `B3`, cell `A8`). Each cell may contain a string of text, a number value, or a mathematical formula. The spreadsheet automatically updates the results of all mathematical formulae whenever the entered number values are changed. This means it is possible to set up a spreadsheet to perform a series of calculations on entered data, and those calculations will be re-done by the computer any time the data points are edited in any way.

For example, the following spreadsheet calculates average speed based on entered values of distance traveled and time elapsed:

	A	B	C	D
1	Distance traveled	46.9	Kilometers	
2	Time elapsed	1.18	Hours	
3	Average speed	= B1 / B2	km/h	
4				
5				

Text labels contained in cells `A1` through `A3` and cells `C1` through `C3` exist solely for readability and are not involved in any calculations. Cell `B1` contains a sample distance value while cell `B2` contains a sample time value. The formula for computing speed is contained in cell `B3`. Note how this formula begins with an "equals" symbol (`=`), references the values for distance and speed by lettered column and numbered row coordinates (`B1` and `B2`), and uses a forward slash symbol for division (`/`). The coordinates `B1` and `B2` function as *variables*[7] would in an algebraic formula.

When this spreadsheet is executed, the numerical value `39.74576` will appear in cell `B3` rather than the formula `= B1 / B2`, because 39.74576 is the computed speed value given 46.9 kilometers traveled over a period of 1.18 hours. If a different numerical value for distance is entered into cell `B1` or a different value for time is entered into cell `B2`, cell `B3`'s value will automatically update. All you need to do is set up the given values and any formulae into the spreadsheet, and the computer will do all the calculations for you.

Cell `B3` may be referenced by other formulae in the spreadsheet if desired, since it is a variable just like the given values contained in `B1` and `B2`. This means it is possible to set up an entire chain of calculations, one dependent on the result of another, in order to arrive at a final value. The arrangement of the given data and formulae need not follow any pattern on the grid, which means you may place them anywhere.

[7] Spreadsheets may also provide means to attach text labels to cells for use as variable names (Microsoft Excel simply calls these labels "names"), but for simple spreadsheets such as those shown here it's usually easier just to use the standard coordinate naming for each cell.

Common[8] arithmetic operations available for your use in a spreadsheet include the following:

- Addition (`+`)
- Subtraction (`-`)
- Multiplication (`*`)
- Division (`/`)
- Powers (`^`)
- Square roots (`sqrt()`)
- Logarithms (`ln()` , `log10()`)

Parentheses may be used to ensure[9] proper order of operations within a complex formula. Consider this example of a spreadsheet implementing the *quadratic formula*, used to solve for roots of a polynomial expression in the form of $ax^2 + bx + c$:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

	A	B
1	x_1	= (-B4 + sqrt((B4^2) - (4*B3*B5))) / (2*B3)
2	x_2	= (-B4 - sqrt((B4^2) - (4*B3*B5))) / (2*B3)
3	a =	9
4	b =	5
5	c =	-2

This example is configured to compute roots[10] of the polynomial $9x^2 + 5x - 2$ because the values of 9, 5, and -2 have been inserted into cells `B3`, `B4`, and `B5`, respectively. Once this spreadsheet has been built, though, it may be used to calculate the roots of *any* second-degree polynomial expression simply by entering the new a, b, and c coefficients into cells `B3` through `B5`. The numerical values appearing in cells `B1` and `B2` will be automatically updated by the computer immediately following any changes made to the coefficients.

[8] Modern spreadsheet software offers a bewildering array of mathematical functions you may use in your computations. I recommend you consult the documentation for your particular spreadsheet for information on operations other than those listed here.

[9] Spreadsheet programs, like text-based programming languages, are designed to follow standard order of operations by default. However, my personal preference is to use parentheses even where strictly unnecessary just to make it clear to any other person viewing the formula what the intended order of operations is.

[10] Reviewing some algebra here, a *root* is a value for x that yields an overall value of zero for the polynomial. For this polynomial ($9x^2 + 5x - 2$) the two roots happen to be $x = 0.269381$ and $x = -0.82494$, with these values displayed in cells `B1` and `B2`, respectively upon execution of the spreadsheet.

Alternatively, one could break up the long quadratic formula into smaller pieces like this:

$$y = \sqrt{b^2 - 4ac} \qquad z = 2a$$

$$x = \frac{-b \pm y}{z}$$

	A	B	C
1	x_1	= (-B4 + C1) / C2	= sqrt((B4^2) - (4*B3*B5))
2	x_2	= (-B4 - C1) / C2	= 2*B3
3	a =	9	
4	b =	5	
5	c =	-2	

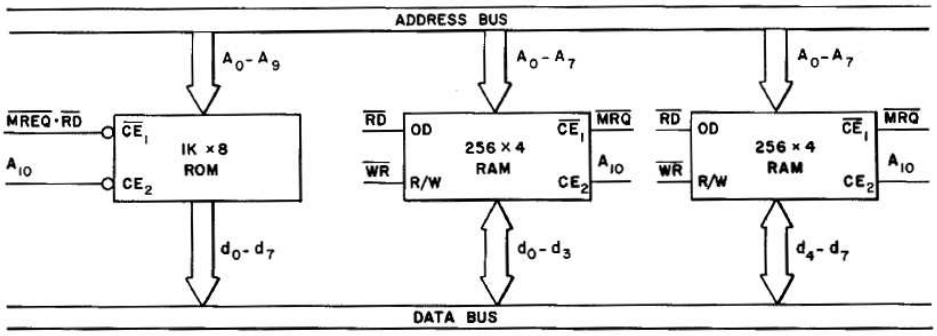
Note how the square-root term (y) is calculated in cell C1, and the denominator term (z) in cell C2. This makes the two final formulae (in cells B1 and B2) simpler to interpret. The positioning of all these cells on the grid is completely arbitrary[11] – all that matters is that they properly reference each other in the formulae.

Spreadsheets are particularly useful for situations where the same set of calculations representing a circuit or other system must be repeated for different initial conditions. The power of a spreadsheet is that it automates what would otherwise be a tedious set of calculations. One specific application of this is to simulate the effects of various components within a circuit failing with abnormal values (e.g. a shorted resistor simulated by making its value nearly zero; an open resistor simulated by making its value extremely large). Another application is analyzing the behavior of a circuit design given new components that are out of specification, and/or aging components experiencing drift over time.

[11] My personal preference is to locate all the "given" data in the upper-left cells of the spreadsheet grid (each data point flanked by a sensible name in the cell to the left and units of measurement in the cell to the right as illustrated in the first distance/time spreadsheet example), sometimes coloring them in order to clearly distinguish which cells contain entered data versus which cells contain computed results from formulae. I like to place all formulae in cells below the given data, and try to arrange them in logical order so that anyone examining my spreadsheet will be able to figure out *how* I constructed a solution. This is a general principle I believe all computer programmers should follow: *document and arrange your code to make it easy for other people to learn from it.*

4.2.3 Memory map determination

US Patent number 4,486,827 ("Microprocessor Apparatus") granted in December 1984 describes a development system based on Zilog's model Z80 microprocessor, a rival[12] to the Intel model 8080 microprocessor. In this patent the authors provide a diagram showing some RAM and ROM memory ICs connected to the address and data busses of the Z80:



ROM & RAM IMPLEMENTATION EXAMPLE

FIG. 4

Based on the connections you see in this diagram, sketch a memory map for this computer showing the locations of the addressable ROM and RAM space and explain how you were able to determine the address ranges for the ROM and RAM.

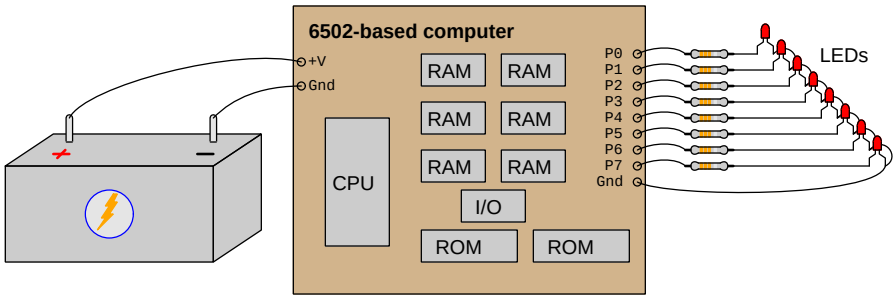
Challenges

- Why are there two RAM chips but just one ROM chip?
- Modify this circuit to give the RAM just as much total storage as the ROM.
- Modify the circuit to swap the locations of RAM and ROM in the memory space.

[12] The Z80 was developed largely by a former Intel employee (Frederico Faggin) and designed to be backward-compatible with the 8080. It offered a simpler interface design for external ICs (e.g. ROM, RAM), simpler power supply requirements, and also simpler clock requirements than the Intel 8080.

4.2.4 6502 turning on LEDs

Suppose you are working with a computer using the MOS model 6502 microprocessor, and it happens to have a memory-mapped 8-bit output port located at address `0x8000`:



Explain how the following assembly-language program, once assembled and loaded into the computer's memory, causes the LEDs connected to pins `P3` and `P5` and `P6` to energize:

```
OutPort .equ $8000
        .org $0200

Main:
        lda #$68      ; Loads value into Accumulator
        sta OutPort   ; Stores (copies) Accumulator's value to OutPort
        .end
```

Now, modify this program to turn on LEDs `P0` through `P5`, leaving the last two LEDs off.

Challenges

- What would happen if `OutPort .equ $8000` were altered to read `OutPort .equ $8001`?
- If the `lda #$68` line were modified to read `lda $68` the result would be very different. Instead of loading the hexadecimal value `68` into the Accumulator register, the instruction instead would refer to *address* `68` and load into the Accumulator whatever data value it found there. The `#` symbol generally means *immediate* in assembly-language programming. Explain why this is.

4.2.5 PIC 16F18346 subroutines

The following assembly-language program[13] instructs a Microchip model 16F18346 microcontroller to turn an LED on and off at a slow rate:

```
; Blinking LED program with nested delays

LOOP
    CALL DELAY1
    BSF PORTA,2          ; Set bit 2 on Port A
    CALL DELAY1
    BCF PORTA,2          ; Clear bit 2 on Port A
    GOTO LOOP

DELAY1
    MOVLW 0x0A           ; Delay time loaded into register W
    MOVWF 0x20           ; Move from register W to address 0x20 of RAM
INNERLOOP
    CALL DELAY2
    DECFSZ 0x20, 1       ; Decrement, skip next instruction if zero
    GOTO INNERLOOP
    RETURN

DELAY2
    MOVLW 0xFF           ; Delay time loaded into register W
    MOVWF 0x21           ; Move from register W to address 0x21 of RAM
INNERLOOP2
    DECFSZ 0x21, 1       ; Decrement, skip next instruction if zero
    GOTO INNERLOOP2
    RETURN
    END
```

This program uses *subroutines* to break the normal flow of execution. When a `CALL` instruction is executed, the microprocessor's Program Counter gets loaded with the address of the instruction located at the corresponding label. A `RETURN` instruction tells the Program Counter to go back to the next address after the `CALL` (i.e. to *return* where it left off).

Trace the order in which these instructions are executed as the program runs, and explain how it achieves a slow blinking rate.

[13] Note that this is a *partial* program listing. To make this program complete would require an "include" directive to inform the assembler as to the definitions of certain labels such as *PORTA*, as well as "configure" directives setting some of the important configuration bits for this particular microcontroller. Also, we would need several lines of code setting and clearing various bits to configure port A to be an output port to drive the LED.

Challenges

- Identify an edit to this program to make it blink exactly twice as fast as it blinks now.

4.2.6 Bitwise logical operations

Determine the results of the following bitwise logical operations:

- `0xBE` bitwise-OR `0x13` =
- `0x4A` bitwise-AND `0xFC` =
- `0x27` bitwise-XOR `0x5F` =

If an 8-bit register in a microprocessor happened to contain the value `0x6D` and we wished to invert the value of the most-significant bit while leaving the other seven bits unaltered, which bitwise operation would be best for this task and what would the necessary mask value be? Also, what would the result of this single-bit inversion be?

If an 8-bit register in a microprocessor happened to contain the value `0x97` and we wished to force the value of the least-significant bit to zero (low, 0) while leaving the other seven bits unaltered, which bitwise operation would be best for this task and what would the necessary mask value be? Also, what would the result of this single-bit force be?

If an 8-bit register in a microprocessor happened to contain the value `0xC3` and we wished to force the value of the 5th-most significant bit to one (high, 1) while leaving the other seven bits unaltered, which bitwise operation would be best for this task and what would the necessary mask value be? Also, what would the result of this single-bit force be?

Challenges

- Identify the symbols used for bitwise operations in either C or C++ programming.

4.3 Diagnostic reasoning

These questions are designed to stimulate your deductive and inductive thinking, where you must apply general principles to specific scenarios (deductive) and also derive conclusions about the failed circuit from specific details (inductive). In a Socratic discussion with your instructor, the goal is for these questions to reinforce your recall and use of general circuit principles and also challenge your ability to integrate multiple symptoms into a sensible explanation of what's wrong in a circuit. Your instructor may also pose additional questions based on those assigned, in order to further challenge and sharpen your diagnostic abilities.

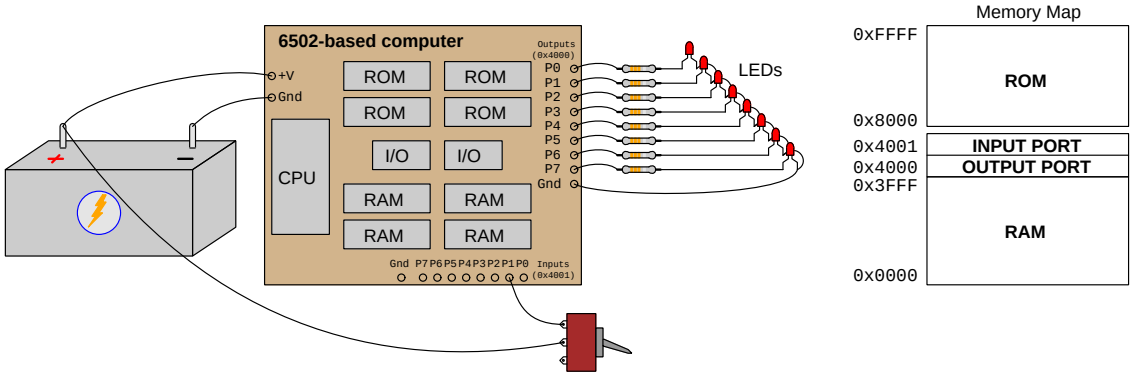
As always, your goal is to fully *explain* your analysis of each problem. Simply obtaining a correct answer is not good enough – you must also demonstrate sound reasoning in order to successfully complete the assignment. Your instructor's responsibility is to probe and challenge your understanding of the relevant principles and analytical processes in order to ensure you have a strong foundation upon which to build further understanding.

You will note a conspicuous lack of answers given for these diagnostic questions. Unlike standard textbooks where answers to every other question are given somewhere toward the back of the book, here in these learning modules students must rely on other means to check their work. The best way by far is to debate the answers with fellow students and also with the instructor during the Socratic dialogue sessions intended to be used with these learning modules. Reasoning through challenging questions with other people is an excellent tool for developing strong reasoning skills.

Another means of checking your diagnostic answers, where applicable, is to use circuit simulation software to explore the effects of faults placed in circuits. For example, if one of these diagnostic questions requires that you predict the effect of an open or a short in a circuit, you may check the validity of your work by simulating that same fault (substituting a very high resistance in place of that component for an open, and substituting a very low resistance for a short) within software and seeing if the results agree.

4.3.1 Random input states

A student connects a toggle switch to one of the input port terminals on this single-board computer, hoping to use the status of that switch as an input to their program:



Unfortunately, this results in unexpected behavior. When the toggle switch is closed the computer properly registers the "high" state at the input port. However, when the toggle switch is open the computer sometimes registers a "low" state but other times a "high" state. Interestingly, the student notices all the unused port states acting "random" as well.

Identify the problem here, and propose a solution for it.

Write the hexadecimal equivalent of the input port memory data as well as its memory address when the switch is turned on and all the other input pins on that port are low.

Challenges

- Write a program segment that takes the input port's bit states and copies them over to the output port.

Appendix A

Problem-Solving Strategies

The ability to solve complex problems is arguably one of the most valuable skills one can possess, and this skill is particularly important in any science-based discipline.

- Study principles, not procedures. Don't be satisfied with merely knowing how to compute solutions – learn *why* those solutions work.
- Identify what it is you need to solve, identify all relevant data, identify all units of measurement, identify any general principles or formulae linking the given information to the solution, and then identify any "missing pieces" to a solution. Annotate all diagrams with this data.
- Sketch a diagram to help visualize the problem. When building a real system, always devise a plan for that system and analyze its function *before* constructing it.
- Follow the units of measurement and meaning of every calculation. If you are ever performing mathematical calculations as part of a problem-solving procedure, and you find yourself unable to apply each and every intermediate result to some aspect of the problem, it means you don't understand what you are doing. Properly done, every mathematical result should have practical meaning for the problem, and not just be an abstract number. You should be able to identify the proper units of measurement for each and every calculated result, and show where that result fits into the problem.
- Perform "thought experiments" to explore the effects of different conditions for theoretical problems. When troubleshooting real systems, perform *diagnostic tests* rather than visually inspecting for faults, the best diagnostic test being the one giving you the most information about the nature and/or location of the fault with the fewest steps.
- Simplify the problem until the solution becomes obvious, and then use that obvious case as a model to follow in solving the more complex version of the problem.
- Check for exceptions to see if your solution is incorrect or incomplete. A good solution will work for *all* known conditions and criteria. A good example of this is the process of testing scientific hypotheses: the task of a scientist is not to find support for a new idea, but rather to *challenge* that new idea to see if it holds up under a battery of tests. The philosophical

principle of *reductio ad absurdum* (i.e. disproving a general idea by finding a specific case where it fails) is useful here.

- Work "backward" from a hypothetical solution to a new set of given conditions.
- Add quantities to problems that are qualitative in nature, because sometimes a little math helps illuminate the scenario.
- Sketch graphs illustrating how variables relate to each other. These may be quantitative (i.e. with realistic number values) or qualitative (i.e. simply showing increases and decreases).
- Treat quantitative problems as qualitative in order to discern the relative magnitudes and/or directions of change of the relevant variables. For example, try determining what happens if a certain variable were to increase or decrease before attempting to precisely calculate quantities: how will each of the dependent variables respond, by increasing, decreasing, or remaining the same as before?
- Consider limiting cases. This works especially well for qualitative problems where you need to determine which direction a variable will change. Take the given condition and magnify that condition to an extreme degree as a way of simplifying the direction of the system's response.
- Check your work. This means regularly testing your conclusions to see if they make sense. This does *not* mean repeating the same steps originally used to obtain the conclusion(s), but rather to use some other means to check validity. Simply repeating procedures often leads to *repeating the same errors* if any were made, which is why alternative paths are better.

Appendix B

Instructional philosophy

"The unexamined circuit is not worth energizing" – Socrates (if he had taught electricity)

These learning modules, although useful for self-study, were designed to be used in a formal learning environment where a subject-matter expert challenges students to digest the content and exercise their critical thinking abilities in the answering of questions and in the construction and testing of working circuits.

The following principles inform the instructional and assessment philosophies embodied in these learning modules:

- The first goal of education is to enhance clear and independent thought, in order that every student reach their fullest potential in a highly complex and inter-dependent world. Robust reasoning is *always* more important than particulars of any subject matter, because its application is universal.
- Literacy is fundamental to independent learning and thought because text continues to be the most efficient way to communicate complex ideas over space and time. Those who cannot read with ease are limited in their ability to acquire knowledge and perspective.
- Articulate communication is fundamental to work that is complex and interdisciplinary.
- Faulty assumptions and poor reasoning are best corrected through challenge, not presentation. The rhetorical technique of *reductio ad absurdum* (disproving an assertion by exposing an absurdity) works well to discipline student's minds, not only to correct the problem at hand but also to learn how to detect and correct future errors.
- Important principles should be repeatedly explored and widely applied throughout a course of study, not only to reinforce their importance and help ensure their mastery, but also to showcase the interconnectedness and utility of knowledge.

These learning modules were expressly designed to be used in an "inverted" teaching environment[1] where students first read the introductory and tutorial chapters on their own, then individually attempt to answer the questions and construct working circuits according to the experiment and project guidelines. The instructor never lectures, but instead meets regularly with each individual student to review their progress, answer questions, identify misconceptions, and challenge the student to new depths of understanding through further questioning. Regular meetings between instructor and student should resemble a Socratic[2] dialogue, where questions serve as scalpels to dissect topics and expose assumptions. The student passes each module only after consistently demonstrating their ability to logically analyze and correctly apply all major concepts in each question or project/experiment. The instructor must be vigilant in probing each student's understanding to ensure they are truly *reasoning* and not just *memorizing*. This is why "Challenge" points appear throughout, as prompts for students to think deeper about topics and as starting points for instructor queries. Sometimes these challenge points require additional knowledge that hasn't been covered in the series to answer in full. This is okay, as the major purpose of the Challenges is to stimulate analysis and synthesis on the part of each student.

The instructor must possess enough mastery of the subject matter and awareness of students' reasoning to generate their own follow-up questions to practically any student response. Even completely correct answers given by the student should be challenged by the instructor for the purpose of having students practice articulating their thoughts and defending their reasoning. Conceptual errors committed by the student should be exposed and corrected not by direct instruction, but rather by reducing the errors to an absurdity[3] through well-chosen questions and thought experiments posed by the instructor. Becoming proficient at this style of instruction requires time and dedication, but the positive effects on critical thinking for both student and instructor are spectacular.

An inspection of these learning modules reveals certain unique characteristics. One of these is a bias toward thorough explanations in the tutorial chapters. Without a live instructor to explain concepts and applications to students, the text itself must fulfill this role. This philosophy results in lengthier explanations than what you might typically find in a textbook, each step of the reasoning process fully explained, including footnotes addressing common questions and concerns students raise while learning these concepts. Each tutorial seeks to not only explain each major concept in sufficient detail, but also to explain the logic of each concept and how each may be developed

[1] In a traditional teaching environment, students first encounter new information via *lecture* from an expert, and then independently apply that information via *homework*. In an "inverted" course of study, students first encounter new information via *homework*, and then independently apply that information under the scrutiny of an expert. The expert's role in lecture is to simply *explain*, but the expert's role in an inverted session is to *challenge*, *critique*, and if necessary *explain* where gaps in understanding still exist.

[2] Socrates is a figure in ancient Greek philosophy famous for his unflinching style of questioning. Although he authored no texts, he appears as a character in Plato's many writings. The essence of Socratic philosophy is to leave no question unexamined and no point of view unchallenged. While purists may argue a topic such as electric circuits is too narrow for a true Socratic-style dialogue, I would argue that the essential thought processes involved with scientific reasoning on *any* topic are not far removed from the Socratic ideal, and that students of electricity and electronics would do very well to challenge assumptions, pose thought experiments, identify fallacies, and otherwise employ the arsenal of critical thinking skills modeled by Socrates.

[3] This rhetorical technique is known by the Latin phrase *reductio ad absurdum*. The concept is to expose errors by counter-example, since only one solid counter-example is necessary to disprove a universal claim. As an example of this, consider the common misconception among beginning students of electricity that voltage cannot exist without current. One way to apply *reductio ad absurdum* to this statement is to ask how much current passes through a fully-charged battery connected to nothing (i.e. a clear example of voltage existing without current).

from "first principles". Again, this reflects the goal of developing clear and independent thought in students' minds, by showing how clear and logical thought was used to forge each concept. Students benefit from witnessing a model of clear thinking in action, and these tutorials strive to be just that.

Another characteristic of these learning modules is a lack of step-by-step instructions in the Project and Experiment chapters. Unlike many modern workbooks and laboratory guides where step-by-step instructions are prescribed for each experiment, these modules take the approach that students must learn to closely read the tutorials and apply their own reasoning to identify the appropriate experimental steps. Sometimes these steps are plainly declared in the text, just not as a set of enumerated points. At other times certain steps are implied, an example being assumed competence in test equipment use where the student should not need to be told *again* how to use their multimeter because that was thoroughly explained in previous lessons. In some circumstances no steps are given at all, leaving the entire procedure up to the student.

This lack of prescription is not a flaw, but rather a feature. Close reading and clear thinking are foundational principles of this learning series, and in keeping with this philosophy all activities are designed to *require* those behaviors. Some students may find the lack of prescription frustrating, because it demands more from them than what their previous educational experiences required. This frustration should be interpreted as an unfamiliarity with autonomous thinking, a problem which must be corrected if the student is ever to become a self-directed learner and effective problem-solver. Ultimately, the need for students to read closely and think clearly is more important both in the near-term and far-term than any specific facet of the subject matter at hand. If a student takes longer than expected to complete a module because they are forced to outline, digest, and reason on their own, so be it. The future gains enjoyed by developing this mental discipline will be well worth the additional effort and delay.

Another feature of these learning modules is that they do not treat topics in isolation. Rather, important concepts are introduced early in the series, and appear repeatedly as stepping-stones toward other concepts in subsequent modules. This helps to avoid the "compartmentalization" of knowledge, demonstrating the inter-connectedness of concepts and simultaneously reinforcing them. Each module is fairly complete in itself, reserving the beginning of its tutorial to a review of foundational concepts.

This methodology of assigning text-based modules to students for digestion and then using Socratic dialogue to assess progress and hone students' thinking was developed over a period of several years by the author with his Electronics and Instrumentation students at the two-year college level. While decidedly unconventional and sometimes even unsettling for students accustomed to a more passive lecture environment, this instructional philosophy has proven its ability to convey conceptual mastery, foster careful analysis, and enhance employability so much better than lecture that the author refuses to ever teach by lecture again.

Problems which often go undiagnosed in a lecture environment are laid bare in this "inverted" format where students must articulate and logically defend their reasoning. This, too, may be unsettling for students accustomed to lecture sessions where the instructor cannot tell for sure who comprehends and who does not, and this vulnerability necessitates sensitivity on the part of the "inverted" session instructor in order that students never feel discouraged by having their errors exposed. *Everyone* makes mistakes from time to time, and learning is a lifelong process! Part of the instructor's job is to build a culture of learning among the students where errors are not seen as shameful, but rather as opportunities for progress.

To this end, instructors managing courses based on these modules should adhere to the following principles:

- Student questions are always welcome and demand thorough, honest answers. The only type of question an instructor should refuse to answer is one the student should be able to easily answer on their own. Remember, *the fundamental goal of education is for each student to learn to think clearly and independently.* This requires hard work on the part of the student, which no instructor should ever circumvent. Anything done to bypass the student's responsibility to do that hard work ultimately limits that student's potential and thereby does real harm.

- It is not only permissible, but encouraged, to answer a student's question by asking questions in return, these follow-up questions designed to guide the student to reach a correct answer through their own reasoning.

- All student answers demand to be challenged by the instructor and/or by other students. This includes both correct and incorrect answers – the goal is to practice the articulation and defense of one's own reasoning.

- No reading assignment is deemed complete unless and until the student demonstrates their ability to accurately summarize the major points in their own terms. Recitation of the original text is unacceptable. This is why every module contains an "Outline and reflections" question as well as a "Foundational concepts" question in the Conceptual reasoning section, to prompt reflective reading.

- No assigned question is deemed answered unless and until the student demonstrates their ability to consistently and correctly apply the concepts to *variations* of that question. This is why module questions typically contain multiple "Challenges" suggesting different applications of the concept(s) as well as variations on the same theme(s). Instructors are encouraged to devise as many of their own "Challenges" as they are able, in order to have a multitude of ways ready to probe students' understanding.

- No assigned experiment or project is deemed complete unless and until the student demonstrates the task in action. If this cannot be done "live" before the instructor, video-recordings showing the demonstration are acceptable. All relevant safety precautions must be followed, all test equipment must be used correctly, and the student must be able to properly explain all results. The student must also successfully answer all Challenges presented by the instructor for that experiment or project.

Students learning from these modules would do well to abide by the following principles:

- No text should be considered fully and adequately read unless and until you can express every idea *in your own words, using your own examples.*
- You should always articulate your thoughts as you read the text, noting points of agreement, confusion, and epiphanies. Feel free to print the text on paper and then write your notes in the margins. Alternatively, keep a journal for your own reflections as you read. This is truly a helpful tool when digesting complicated concepts.
- Never take the easy path of highlighting or underlining important text. Instead, *summarize* and/or *comment* on the text using your own words. This actively engages your mind, allowing you to more clearly perceive points of confusion or misunderstanding on your own.
- A very helpful strategy when learning new concepts is to place yourself in the role of a teacher, if only as a mental exercise. Either explain what you have recently learned to someone else, or at least *imagine* yourself explaining what you have learned to someone else. The simple act of having to articulate new knowledge and skill forces you to take on a different perspective, and will help reveal weaknesses in your understanding.
- Perform each and every mathematical calculation and thought experiment shown in the text on your own, referring back to the text to see that your results agree. This may seem trivial and unnecessary, but it is critically important to ensuring you actually understand what is presented, especially when the concepts at hand are complicated and easy to misunderstand. Apply this same strategy to become proficient in the use of *circuit simulation software*, checking to see if your simulated results agree with the results shown in the text.
- Above all, recognize that learning is hard work, and that a certain level of frustration is unavoidable. There are times when you will struggle to grasp some of these concepts, and that struggle is a natural thing. Take heart that it will yield with persistent and varied[4] effort, and never give up!

Students interested in using these modules for self-study will also find them beneficial, although the onus of responsibility for thoroughly reading and answering questions will of course lie with that individual alone. If a qualified instructor is not available to challenge students, a workable alternative is for students to form study groups where they challenge[5] one another.

To high standards of education,

Tony R. Kuphaldt

[4] As the old saying goes, "Insanity is trying the same thing over and over again, expecting different results." If you find yourself stumped by something in the text, you should attempt a different approach. Alter the thought experiment, change the mathematical parameters, do whatever you can to see the problem in a slightly different light, and then the solution will often present itself more readily.

[5] Avoid the temptation to simply share answers with study partners, as this is really counter-productive to learning. Always bear in mind that the answer to any question is far less important in the long run than the method(s) used to obtain that answer. The goal of education is to empower one's life through the improvement of clear and independent thought, literacy, expression, and various practical skills.

Appendix C

Tools used

I am indebted to the developers of many open-source software applications in the creation of these learning modules. The following is a list of these applications with some commentary on each.

You will notice a theme common to many of these applications: a bias toward *code*. Although I am by no means an expert programmer in any computer language, I understand and appreciate the flexibility offered by code-based applications where the user (you) enters commands into a plain ASCII text file, which the software then reads and processes to create the final output. Code-based computer applications are by their very nature *extensible*, while WYSIWYG (What You See Is What You Get) applications are generally limited to whatever user interface the developer makes for you.

The `GNU/Linux` computer operating system

> There is so much to be said about Linus Torvalds' `Linux` and Richard Stallman's `GNU` project. First, to credit just these two individuals is to fail to do justice to the *mob* of passionate volunteers who contributed to make this amazing software a reality. I first learned of `Linux` back in 1996, and have been using this operating system on my personal computers almost exclusively since then. It is *free*, it is completely *configurable*, and it permits the continued use of highly efficient `Unix` applications and scripting languages (e.g. shell scripts, Makefiles, `sed`, `awk`) developed over many decades. `Linux` not only provided me with a powerful computing platform, but its open design served to inspire my life's work of creating open-source educational resources.

Bram Moolenaar's `Vim` text editor

> Writing code for any code-based computer application requires a *text editor*, which may be thought of as a word processor strictly limited to outputting plain-ASCII text files. Many good text editors exist, and one's choice of text editor seems to be a deeply personal matter within the programming world. I prefer `Vim` because it operates very similarly to `vi` which is ubiquitous on `Unix`/`Linux` operating systems, and because it may be entirely operated via keyboard (i.e. no mouse required) which makes it fast to use.

Donald Knuth's TEX typesetting system

Developed in the late 1970's and early 1980's by computer scientist extraordinaire Donald Knuth to typeset his multi-volume magnum opus *The Art of Computer Programming*, this software allows the production of formatted text for screen-viewing or paper printing, all by writing plain-text code to describe how the formatted text is supposed to appear. TEX is not just a markup language for documents, but it is also a Turing-complete programming language in and of itself, allowing useful algorithms to be created to control the production of documents. Simply put, TEX *is a programmer's approach to word processing*. Since TEX is controlled by code written in a plain-text file, this means anyone may read that plain-text file to see exactly how the document was created. This openness afforded by the code-based nature of TEX makes it relatively easy to learn how other people have created their own TEX documents. By contrast, examining a beautiful document created in a conventional WYSIWYG word processor such as Microsoft `Word` suggests nothing to the reader about *how* that document was created, or what the user might do to create something similar. As Mr. Knuth himself once quipped, conventional word processing applications should be called WYSIAYG (What You See Is *All* You Get).

Leslie Lamport's LATEX extensions to TEX

Like all true programming languages, TEX is inherently extensible. So, years after the release of TEX to the public, Leslie Lamport decided to create a massive extension allowing easier compilation of book-length documents. The result was LATEX, which is the markup language used to create all ModEL module documents. You could say that TEX is to LATEX as `C` is to `C++`. This means it is permissible to use any and all TEX commands within LATEX source code, and it all still works. Some of the features offered by LATEX that would be challenging to implement in TEX include automatic index and table-of-content creation.

Tim Edwards' `Xcircuit` drafting program

This wonderful program is what I use to create all the schematic diagrams and illustrations (but not photographic images or mathematical plots) throughout the ModEL project. It natively outputs PostScript format which is a true vector graphic format (this is why the images do not pixellate when you zoom in for a closer view), and it is so simple to use that I have never had to read the manual! Object libraries are easy to create for `Xcircuit`, being plain-text files using PostScript programming conventions. Over the years I have collected a large set of object libraries useful for drawing electrical and electronic schematics, pictorial diagrams, and other technical illustrations.

`Gimp` graphic image manipulation program

Essentially an open-source clone of Adobe's `PhotoShop`, I use `Gimp` to resize, crop, and convert file formats for all of the photographic images appearing in the ModEL modules. Although `Gimp` does offer its own scripting language (called `Script-Fu`), I have never had occasion to use it. Thus, my utilization of `Gimp` to merely crop, resize, and convert graphic images is akin to using a sword to slice bread.

`SPICE` circuit simulation program

`SPICE` is to circuit analysis as TEX is to document creation: it is a form of markup language designed to describe a certain object to be processed in plain-ASCII text. When the plain-text "source file" is compiled by the software, it outputs the final result. More modern circuit analysis tools certainly exist, but I prefer `SPICE` for the following reasons: it is *free*, it is *fast*, it is *reliable*, and it is a fantastic tool for *teaching* students of electricity and electronics how to write simple code. I happen to use rather old versions of `SPICE`, version 2g6 being my "go to" application when I only require text-based output. `NGSPICE` (version 26), which is based on Berkeley `SPICE` version 3f5, is used when I require graphical output for such things as time-domain waveforms and Bode plots. In all `SPICE` example netlists I strive to use coding conventions compatible with all `SPICE` versions.

Andrew D. Hwang's `ePiX` mathematical visualization programming library

This amazing project is a `C++` library you may link to any `C/C++` code for the purpose of generating PostScript graphic images of mathematical functions. As a completely free and open-source project, it does all the plotting I would otherwise use a Computer Algebra System (CAS) such as `Mathematica` or `Maple` to do. It should be said that `ePiX` is *not* a Computer Algebra System like `Mathematica` or `Maple`, but merely a mathematical *visualization* tool. In other words, it won't determine integrals for you (you'll have to implement that in your own `C/C++` code!), but it can graph the results, and it does so beautifully. What I really admire about `ePiX` is that it is a `C++` programming library, which means it builds on the existing power and toolset available with that programming language. Mr. Hwang could have probably developed his own stand-alone application for mathematical plotting, but by creating a `C++` library to do the same thing he accomplished something much greater.

`gnuplot` mathematical visualization software

Another open-source tool for mathematical visualization is `gnuplot`. Interestingly, this tool is *not* part of Richard Stallman's GNU project, its name being a coincidence. For this reason the authors prefer "gnu" *not* be capitalized at all to avoid confusion. This is a much "lighter-weight" alternative to a spreadsheet for plotting tabular data, and the fact that it easily outputs directly to an X11 console or a file in a number of different graphical formats (including PostScript) is very helpful. I typically set my `gnuplot` output format to default (X11 on my Linux PC) for quick viewing while I'm developing a visualization, then switch to PostScript file export once the visual is ready to include in the document(s) I'm writing. As with my use of `Gimp` to do rudimentary image editing, my use of `gnuplot` only scratches the surface of its capabilities, but the important points are that it's *free* and that it *works well*.

`Python` programming language

Both Python and C++ find extensive use in these modules as instructional aids and exercises, but I'm listing Python here as a *tool* for myself because I use it almost daily as a *calculator*. If you open a Python interpreter console and type `from math import *` you can type mathematical expressions and have it return results just as you would on a hand calculator. Complex-number (i.e. *phasor*) arithmetic is similarly supported if you include the complex-math library (`from cmath import *`). Examples of this are shown in the Programming References chapter (if included) in each module. Of course, being a fully-featured programming language, Python also supports conditionals, loops, and other structures useful for calculation of quantities. Also, running in a console environment where all entries and returned values show as text in a chronologically-ordered list makes it easy to copy-and-paste those calculations to document exactly how they were performed.

Appendix D

Creative Commons License

Creative Commons Attribution 4.0 International Public License

By exercising the Licensed Rights (defined below), You accept and agree to be bound by the terms and conditions of this Creative Commons Attribution 4.0 International Public License ("Public License"). To the extent this Public License may be interpreted as a contract, You are granted the Licensed Rights in consideration of Your acceptance of these terms and conditions, and the Licensor grants You such rights in consideration of benefits the Licensor receives from making the Licensed Material available under these terms and conditions.

Section 1 – Definitions.

a. **Adapted Material** means material subject to Copyright and Similar Rights that is derived from or based upon the Licensed Material and in which the Licensed Material is translated, altered, arranged, transformed, or otherwise modified in a manner requiring permission under the Copyright and Similar Rights held by the Licensor. For purposes of this Public License, where the Licensed Material is a musical work, performance, or sound recording, Adapted Material is always produced where the Licensed Material is synched in timed relation with a moving image.

b. **Adapter's License** means the license You apply to Your Copyright and Similar Rights in Your contributions to Adapted Material in accordance with the terms and conditions of this Public License.

c. **Copyright and Similar Rights** means copyright and/or similar rights closely related to copyright including, without limitation, performance, broadcast, sound recording, and Sui Generis Database Rights, without regard to how the rights are labeled or categorized. For purposes of this Public License, the rights specified in Section 2(b)(1)-(2) are not Copyright and Similar Rights.

d. **Effective Technological Measures** means those measures that, in the absence of proper authority, may not be circumvented under laws fulfilling obligations under Article 11 of the WIPO Copyright Treaty adopted on December 20, 1996, and/or similar international agreements.

e. **Exceptions and Limitations** means fair use, fair dealing, and/or any other exception or

limitation to Copyright and Similar Rights that applies to Your use of the Licensed Material.

f. **Licensed Material** means the artistic or literary work, database, or other material to which the Licensor applied this Public License.

g. **Licensed Rights** means the rights granted to You subject to the terms and conditions of this Public License, which are limited to all Copyright and Similar Rights that apply to Your use of the Licensed Material and that the Licensor has authority to license.

h. **Licensor** means the individual(s) or entity(ies) granting rights under this Public License.

i. **Share** means to provide material to the public by any means or process that requires permission under the Licensed Rights, such as reproduction, public display, public performance, distribution, dissemination, communication, or importation, and to make material available to the public including in ways that members of the public may access the material from a place and at a time individually chosen by them.

j. **Sui Generis Database Rights** means rights other than copyright resulting from Directive 96/9/EC of the European Parliament and of the Council of 11 March 1996 on the legal protection of databases, as amended and/or succeeded, as well as other essentially equivalent rights anywhere in the world.

k. **You** means the individual or entity exercising the Licensed Rights under this Public License. **Your** has a corresponding meaning.

Section 2 – Scope.

a. License grant.

1. Subject to the terms and conditions of this Public License, the Licensor hereby grants You a worldwide, royalty-free, non-sublicensable, non-exclusive, irrevocable license to exercise the Licensed Rights in the Licensed Material to:

A. reproduce and Share the Licensed Material, in whole or in part; and

B. produce, reproduce, and Share Adapted Material.

2. Exceptions and Limitations. For the avoidance of doubt, where Exceptions and Limitations apply to Your use, this Public License does not apply, and You do not need to comply with its terms and conditions.

3. Term. The term of this Public License is specified in Section 6(a).

4. Media and formats; technical modifications allowed. The Licensor authorizes You to exercise the Licensed Rights in all media and formats whether now known or hereafter created, and to make technical modifications necessary to do so. The Licensor waives and/or agrees not to assert any right or authority to forbid You from making technical modifications necessary to exercise the Licensed Rights, including technical modifications necessary to circumvent Effective Technological Measures.

For purposes of this Public License, simply making modifications authorized by this Section 2(a)(4) never produces Adapted Material.

5. Downstream recipients.

A. Offer from the Licensor – Licensed Material. Every recipient of the Licensed Material automatically receives an offer from the Licensor to exercise the Licensed Rights under the terms and conditions of this Public License.

B. No downstream restrictions. You may not offer or impose any additional or different terms or conditions on, or apply any Effective Technological Measures to, the Licensed Material if doing so restricts exercise of the Licensed Rights by any recipient of the Licensed Material.

6. No endorsement. Nothing in this Public License constitutes or may be construed as permission to assert or imply that You are, or that Your use of the Licensed Material is, connected with, or sponsored, endorsed, or granted official status by, the Licensor or others designated to receive attribution as provided in Section 3(a)(1)(A)(i).

b. Other rights.

1. Moral rights, such as the right of integrity, are not licensed under this Public License, nor are publicity, privacy, and/or other similar personality rights; however, to the extent possible, the Licensor waives and/or agrees not to assert any such rights held by the Licensor to the limited extent necessary to allow You to exercise the Licensed Rights, but not otherwise.

2. Patent and trademark rights are not licensed under this Public License.

3. To the extent possible, the Licensor waives any right to collect royalties from You for the exercise of the Licensed Rights, whether directly or through a collecting society under any voluntary or waivable statutory or compulsory licensing scheme. In all other cases the Licensor expressly reserves any right to collect such royalties.

Section 3 – License Conditions.

Your exercise of the Licensed Rights is expressly made subject to the following conditions.

a. Attribution.

1. If You Share the Licensed Material (including in modified form), You must:

A. retain the following if it is supplied by the Licensor with the Licensed Material:

i. identification of the creator(s) of the Licensed Material and any others designated to receive attribution, in any reasonable manner requested by the Licensor (including by pseudonym if designated);

ii. a copyright notice;

iii. a notice that refers to this Public License;

iv. a notice that refers to the disclaimer of warranties;

v. a URI or hyperlink to the Licensed Material to the extent reasonably practicable;

B. indicate if You modified the Licensed Material and retain an indication of any previous modifications; and

C. indicate the Licensed Material is licensed under this Public License, and include the text of, or the URI or hyperlink to, this Public License.

2. You may satisfy the conditions in Section 3(a)(1) in any reasonable manner based on the medium, means, and context in which You Share the Licensed Material. For example, it may be reasonable to satisfy the conditions by providing a URI or hyperlink to a resource that includes the required information.

3. If requested by the Licensor, You must remove any of the information required by Section 3(a)(1)(A) to the extent reasonably practicable.

4. If You Share Adapted Material You produce, the Adapter's License You apply must not prevent recipients of the Adapted Material from complying with this Public License.

Section 4 – Sui Generis Database Rights.

Where the Licensed Rights include Sui Generis Database Rights that apply to Your use of the Licensed Material:

a. for the avoidance of doubt, Section 2(a)(1) grants You the right to extract, reuse, reproduce, and Share all or a substantial portion of the contents of the database;

b. if You include all or a substantial portion of the database contents in a database in which You have Sui Generis Database Rights, then the database in which You have Sui Generis Database Rights (but not its individual contents) is Adapted Material; and

c. You must comply with the conditions in Section 3(a) if You Share all or a substantial portion of the contents of the database.

For the avoidance of doubt, this Section 4 supplements and does not replace Your obligations under this Public License where the Licensed Rights include other Copyright and Similar Rights.

Section 5 – Disclaimer of Warranties and Limitation of Liability.

a. Unless otherwise separately undertaken by the Licensor, to the extent possible, the Licensor offers the Licensed Material as-is and as-available, and makes no representations or warranties of any kind concerning the Licensed Material, whether express, implied, statutory, or other. This includes, without limitation, warranties of title, merchantability, fitness for a particular purpose, non-infringement, absence of latent or other defects, accuracy, or the presence or absence of errors,

whether or not known or discoverable. Where disclaimers of warranties are not allowed in full or in part, this disclaimer may not apply to You.

b. To the extent possible, in no event will the Licensor be liable to You on any legal theory (including, without limitation, negligence) or otherwise for any direct, special, indirect, incidental, consequential, punitive, exemplary, or other losses, costs, expenses, or damages arising out of this Public License or use of the Licensed Material, even if the Licensor has been advised of the possibility of such losses, costs, expenses, or damages. Where a limitation of liability is not allowed in full or in part, this limitation may not apply to You.

c. The disclaimer of warranties and limitation of liability provided above shall be interpreted in a manner that, to the extent possible, most closely approximates an absolute disclaimer and waiver of all liability.

Section 6 – Term and Termination.

a. This Public License applies for the term of the Copyright and Similar Rights licensed here. However, if You fail to comply with this Public License, then Your rights under this Public License terminate automatically.

b. Where Your right to use the Licensed Material has terminated under Section 6(a), it reinstates:

1. automatically as of the date the violation is cured, provided it is cured within 30 days of Your discovery of the violation; or

2. upon express reinstatement by the Licensor.

For the avoidance of doubt, this Section 6(b) does not affect any right the Licensor may have to seek remedies for Your violations of this Public License.

c. For the avoidance of doubt, the Licensor may also offer the Licensed Material under separate terms or conditions or stop distributing the Licensed Material at any time; however, doing so will not terminate this Public License.

d. Sections 1, 5, 6, 7, and 8 survive termination of this Public License.

Section 7 – Other Terms and Conditions.

a. The Licensor shall not be bound by any additional or different terms or conditions communicated by You unless expressly agreed.

b. Any arrangements, understandings, or agreements regarding the Licensed Material not stated herein are separate from and independent of the terms and conditions of this Public License.

Section 8 – Interpretation.

a. For the avoidance of doubt, this Public License does not, and shall not be interpreted to, reduce, limit, restrict, or impose conditions on any use of the Licensed Material that could lawfully

be made without permission under this Public License.

b. To the extent possible, if any provision of this Public License is deemed unenforceable, it shall be automatically reformed to the minimum extent necessary to make it enforceable. If the provision cannot be reformed, it shall be severed from this Public License without affecting the enforceability of the remaining terms and conditions.

c. No term or condition of this Public License will be waived and no failure to comply consented to unless expressly agreed to by the Licensor.

d. Nothing in this Public License constitutes or may be interpreted as a limitation upon, or waiver of, any privileges and immunities that apply to the Licensor or You, including from the legal processes of any jurisdiction or authority.

Creative Commons is not a party to its public licenses. Notwithstanding, Creative Commons may elect to apply one of its public licenses to material it publishes and in those instances will be considered the "Licensor." Except for the limited purpose of indicating that material is shared under a Creative Commons public license or as otherwise permitted by the Creative Commons policies published at `creativecommons.org/policies`, Creative Commons does not authorize the use of the trademark "Creative Commons" or any other trademark or logo of Creative Commons without its prior written consent including, without limitation, in connection with any unauthorized modifications to any of its public licenses or any other arrangements, understandings, or agreements concerning use of licensed material. For the avoidance of doubt, this paragraph does not form part of the public licenses.

Creative Commons may be contacted at `creativecommons.org`.

Appendix E

References

Coll, John and Allen, David, *BBC Microcomputer System User Guide*, part number 0433 000, Issue 1, British Broadcasting Corporation (1982) and Acorn Computers Limited (October 1984).

"CY62128EV30 MoBL Automotive 1-Mbit (128 K × 8) Static RAM", document 001-65528, Revision E, Cypress Semiconductor Corporation, San Jose, CA, 15 April 2015.

Greenfield, Joseph D., *The 68HC11 Microcontroller*, Saunders College Publishing, Fort Worth, TX, 1992.

Hoff, Marcian Edward Jr.; Mazor, Stanley; Faggin, Frederico, *US Patent 3,821,715*, "Memory System for a Multi-Chip Digital Computer", application 22 January 1973, patent granted 28 June 1974.

Intel 8080 Microcomputer Systems User's Manual, document MCS-662-0975/40K, Intel Corporation, Santa Clara, CA, September 1975.

MacKenzie, I. Scott, *The 8051 Microcontroller*, MacMillan Publishing Company, New York, NY, 1992.

Olley, Allan, *Existence Precedes Essence – Meaning of the Stored-Program Concept*, University of Toronto, Toronto, CA, 2010.

Shima, Masatoshi; Faggin, Frederico; Ungermann, Ralph K. *US Patent 4,486,827*, "Microprocessor Apparatus", application 18 January 1982, patent granted 4 December 1984.

Uffenbeck, John, *Microcomputers and Microprocessors*, Prentice-Hall Incorporated, 1985.

Valvano, Jonathan W., *Embedded Microcomputer Systems: Real Time Interfacing*, second edition, Thomson Canada Limited, 2007.

Z80 PIO User's Manual, Zilog.

Z80 CPU User Manual, document UM008011-0816, Zilog, Incorporated, 2016.

Appendix F

Version history

This is a list showing all significant additions, corrections, and other edits made to this learning module. Each entry is referenced by calendar date in reverse chronological order (newest version first), which appears on the front cover of every learning module for easy reference. Any contributors to this open-source document are listed here as well.

23 January 2025 – minor edits to the Tutorial.

25-29 August 2024 – divided the Introduction chapter into sections, one with recommendations for students, one with a listing of challenging concepts, and one with recommendations for instructors. Also made some minor edits to the Tutorial and to some of the instructor notes.

24-29 January 2024 – added some images from the "Basic Principles of Digital" module showing fundamental logic functions associated with Boolean expressions to the "Logic functions and Boolean algebra" of this Tutorial. Also corrected a minor coloring error in image_3813, included a list of interrupts for the MSP430G2553 microcontroller (to the "Interrupts" section of the Tutorial), and made other minor edits to the Tutorial wording.

26 November 2023 – included trapezoidal symbols for mux and demux devices in that section of the Tutorial.

29 November 2022 – placed questions at the top of the itemized list in the Introduction chapter prompting students to devise experiments related to the tutorial content.

31 August 2021 – added more explanatory text to the "Putting it all together – the processor" section of the Tutorial.

10 May 2021 – commented out or deleted empty chapters.

1 February 2021 – minor additions to the Introduction chapter, Tutorial, and Technical References (intro to assembly language).

27-28 January 2021 – minor additions to the Introduction chapter, as well as minor edits to

images in the "Putting it all together" section of the Tutorial. Also, minor edits to questions.

4 September 2020 – added Quantitative Reasoning question on bitwise operations.

3 September 2020 – corrected some typographical errors found by Ty Weich.

2 September 2020 – added more helpful tips to the Introduction chapter. Also corrected an error in one of the diagrams within the "A simple computer example" section of the Tutorial.

26 August 2020 – significantly edited the Introduction chapter to make it more suitable as a pre-study guide and to provide cues useful to instructors leading "inverted" teaching sessions.

22 June 2020 – added some Foundational Concepts.

5-16 June 2020 – added content to the Tutorial, as well as questions.

4 June 2020 – document first created.

Index