Preface

Like many American kids in 1979, I woke up to find that Santa had left a brand new Atari VCS[1] under the tree (thanks, Mom and Dad, for paying Santa's invoice!). This was a pretty big deal for a six-year-old who could tell you the location and manufacturer of every standup arcade cabinet within a five mile radius. Having an "arcade in your home" wasn't just a thing you saw on *Silver Spoons*, it was now a real thing.

The sights and sounds that jumped off of our little Panasonic color TV probably deserve a gigantic run-on sentence worthy of Dylan Thomas, as my brother and I bounced tiny pixellated missiles off of walls in *Combat*, combed through the perplexing game modes of *Space Invaders*, battled angry duck-like dragons in *Adventure*, and became *Superman* as we put flickering bad guys in a flickering jail. These cartridges were opaque obelisks packaged in boxes with fantastically unattainable illustrations, available at K-Mart for $30 or so.

You could tell these species of video games weren't related to arcade games, though they had a unique look-and-feel of their own. We also had an Apple][by this time, so I tried to fit all of these creatures into a digital taxonomy. Atari games had colors and fast motion, but not as much as arcade games, and they never were as complex as Apple][games. What made them tick? Why were Activision games so much more detailed? Would the missile still blow up your spaceship if you turned the TV off? (Turns out the answer is yes.)

[1] It wasn't sold as "Atari 2600" until 1982. We'll use "VCS" in this book, which stands for Video Computer System.

An Atari 2600 four-switch "wood veneer" version, dating from 1980-1982 (photo by Evan Amos)

Soon afterwards, I would start dissecting the Apple][, and never really got my mitts on the viscera inside those VCS cartridges. It wasn't until the Internet came around that I'd discover the TIA chip, scanlines, and emulators like Stella. I'd also read about the people who wrote the games, often uncredited, who pushed the envelopes of both game design and technology while working solo against impossible deadlines.

It's now been 37 years since that Christmas morning, and thanks to the Web, very fast modern CPUs, and lots of open-source sharing, you can program Atari VCS games in your browser. It's probably the most effort you can expend for the fewest number of pixels, but it's also really rewarding.

If the modern software ecosystem is a crowded and bureaucratic megalopolis, programming the VCS is like tinkering in a tiny cabin in the woods with 10-foot snow drifts outside. At least the stove is working, and there's plenty of wood. Enjoy.

1

Introduction to 6502

In 1974, Chuck Peddle was a Motorola employee trying to sell their 6800 microprocessor to customers for $300 each. He and a few co-workers left the company with the vision of a much cheaper alternative, and landed at MOS Technology in Valley Forge, Pennsylvania.

They began drawing the layout for the chip on a huge sheet of paper in one of the offices. Later, they'd cut the table-sized Rubylith photomask for the 3,510 transistors by hand, wearing clean socks so they wouldn't damage the mask when they had to step over something. The design (mostly) worked on the first run, and the 6502 was sold out of large jars for $25 at the 1975 Wescon trade show.[1] It would sell tens of millions of units over the next decade.

The 6502 CPU was not that much different from other microprocessors in function; it was just cheap and widely available. Yet it powered the Apple I and Apple][computers, the Commodore 64, the Nintendo Entertainment System, and the Atari 2600/VCS, as well as a myriad of other computers and game devices.

While there are plenty of books and online resources devoted to 6502 programming, we're going to cover the basics in this chapter before we jump straight into programming the Atari 2600. Feel free to skip to the next chapter if you already know most of this stuff; we won't cover VCS-specific topics until Chapter Two.

1.1 Bits, Bytes, and Binary

All digital computers operate on bits and bytes and, on the VCS, you'll be manipulating them directly. Let's review a few things about them.

A bit is a binary value – it can be either zero (0) or one (1). A byte is a sequence of eight bits.

We can create a written representation of a byte in *binary notation*, which just lists the bits from left to right, for example: `%00011011`. We can then shorten the byte notation by removing the leading zeros, giving us `%11011`. The % denotes a binary number, and we'll use this notation throughout the book.

The eight bits in a byte are not just independent ones and zeros; they can also express numbers. We assign values to each bit and then add them up. The least-significant bit, the rightmost (our index starts at zero, i.e. *bit 0*), has a value of 1. For each position to the left, the value increases by a power of two until we reach the most-significant bit, the leftmost (*bit 7*) with a value of 128. Here are the values for an entire byte:

Bit #	7	6	5	4	3	2	1	0
Value	128	64	32	16	8	4	2	1

Let's line up our example byte, `%11011`, with these values:

Bit #	7	6	5	4	3	2	1	0
Value	128	64	32	16	8	4	2	1
Our Byte	0	0	0	1	1	0	1	1
Bit*Value				16	8		2	1

When we add up all the bit values, we get $16 + 8 + 2 + 1 = 27$.

1.2 Hexadecimal Notation

Binary notation can be unwieldy, so it's common to represent bytes using *hexadecimal notation*, or *base 16*. We split the byte into two 4-

bit halves, or *nibbles*. We treat each nibble as a separate value from 0 to 15, like this:

Bit #	7	6	5	4	3	2	1	0
Value	8	4	2	1	8	4	2	1

Table 1.1: Bit Values in Hexadecimal Notation

We then convert each nibble's value to a symbol – 0-9 remains `0` through `9`, but 10-15 becomes `A` through `F`.

Let's convert the binary number `%11011` and see how it would be represented in hexadecimal:

Bit #	7	6	5	4	3	2	1	0
Value	8	4	2	1	8	4	2	1
Our Byte	0	0	0	1	1	0	1	1
Bit*Value				1	8		2	1
Decimal Value	1				11			
Hex Value	`1`				`B`			

Table 1.2: Example Hex Conversion

We see in Table 1.2 that the decimal number 27, represented as `%11011` in binary, becomes `$1B` in hexadecimal format. (The `$` prefix indicates a hexadecimal number.)

1.3 Signed vs. Unsigned Bytes

One more thing about bytes: We've described how they can be interpreted as any value from 0 through 255, or an *unsigned* value. We can also interpret them as negative or *signed* quantities.

This requires a trick known as *two's complement* arithmetic. If the high bit is 1 (in other words, if the unsigned value is 128 or greater), we treat the value as negative, as if we had subtracted 256 from it:

```
  0-127 ($00-$7F):      positive
128-255 ($80-$FF):      negative (value - 256)
```

Note that there's nothing in the byte identifying it as signed – it's all in how you interpret it, as we'll see later.

Now that we know what bits and bytes are, let's see how the CPU manipulates them.

1.4 The CPU and the Bus

Think of the CPU as an intricate timepiece. An electronic spring unwinds and an internal clock ticks 1.19 million times per second. On every tick, electrons turn tiny gears, and the CPU comes to rest in a different state. Each tick is called a *clock cycle*, or *CPU clock*, and you'll learn to become aware of their passing as you learn how to program the VCS.

All the CPU does is execute instructions, one after another, in a fetch-decode-execute cycle. It fetches an instruction (reads it from memory), decodes it (figures out what to do) and then executes it (does some things in a prescribed order). Each instruction may take several clock cycles to execute, each clock cycle performing a specific step. The CPU then figures out which instruction to grab next, and repeats the process. The CPU keeps the address of the next instruction in a 16-bit register called the *Program Counter (PC)*.

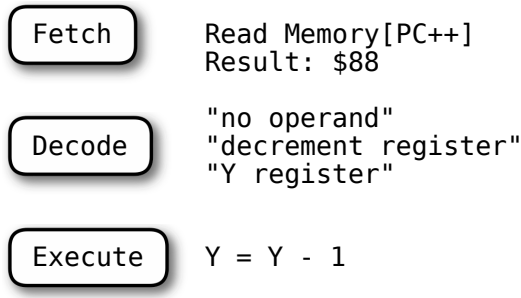

Figure 1.1: CPU Cycle

During each clock cycle, the CPU can read from or write to the bus. The bus is a set of "lanes" where each lane can hold a single bit at a time. The 6502 is an 8-bit processor, so the *data bus* is eight bits (one byte) wide.

Devices like memory and graphics chips are attached to the bus, and receive read and write signals. The CPU doesn't know which devices are connected to the bus – all it knows is that it either receives eight bits back from a read, or sends eight bits out into the world during a write.

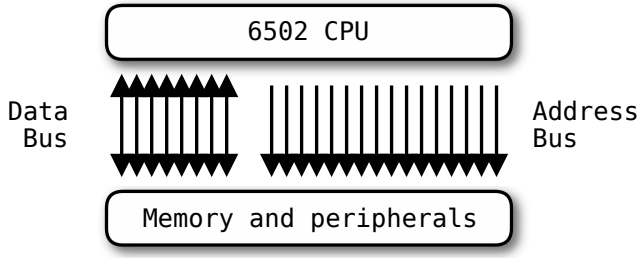

Figure 1.2: Bus

Besides the 8-bit data bus, the 6502 has a 16-bit *address bus*. The address bus describes "where" and the data bus describes "what."

Let's look at what happens when the CPU executes this example instruction, `LDA` (LoaD A):

```
lda $1234
```

The CPU will set the pins on the address bus to the binary encoding for `$1234`, set the read/write pin to "read," and wait for a response on the data bus. Devices on the bus look at the address `$1234` and determine whether the message is for them – by design, only one device should respond. The CPU then reads the value from the data bus and puts it in the A register.

Let's say we are executing the `STA` instruction (STore A):

```
sta $1234
```

The CPU will set the address bus to `$1234` and the data bus to whatever is in the A register, then set the read/write pin to "write." Again, the bus devices look at the address bus and the write signal and decide if they should listen or ignore it. Let's say a memory chip responds – the memory chip would read the 8-bit value off the data bus and store it in the memory cell corresponding to address `$1234`. The CPU does not get a response from a write; it just assumes everything worked out fine.

You'll note that both of these instructions operate on the A register. The 6502 has three general-purpose registers: A, X, and Y. These are all 8-bit variables that you can manipulate at will. You'll often have to use the registers as temporary storage, for instance: Load a constant value into A, then store A to a given address.

You'll notice that the CPU instructions have a three-letter format. This is called a *mnemonic*, and it's part of the human-readable language used by the CPU, called *assembly language*. The CPU doesn't understand this, but it understands a compact code called *machine code*. A program called an *assembler* takes the human-readable assembly code and produces machine code.

Let's take another example instruction:

```
lda $1234 -> ad 34 12
```

The machine code for this instruction is three bytes, `$ad`, `$34`, and `$12`. `$ad` is the *opcode* which identifies the instruction and addressing mode. `$34` and `$12` are part of the *operand*, which in this case is

a 16-bit number spanning two bytes. You'll note that the `$34` is first and the `$12` is second – this is because the 6502 is a *little-endian* processor, expecting the least-significant parts of multibyte quantities first.

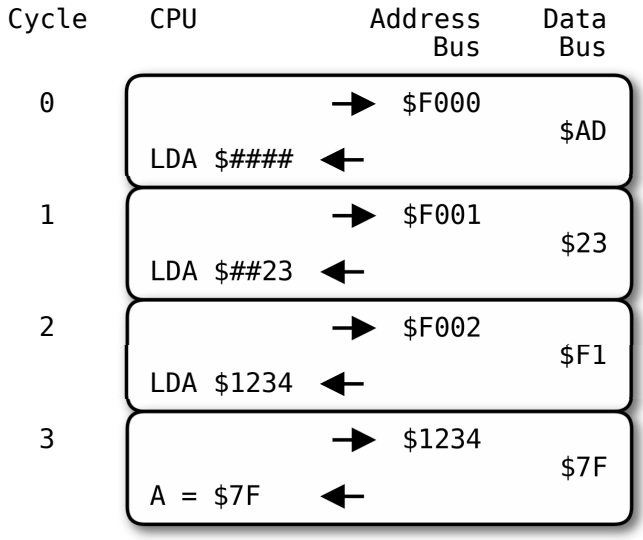

Figure 1.3: LDA Cycle

1.5 Writing Loops

Now we're ready to write a program. Typically, we'd start with the classic example that prints "Hello, World" on the display, but we don't have a display yet! The equivalent program on the Atari 2600 would require us to define the bitmaps for all of the letters in "Hello, World" and we'd also need to learn how CRTs work. So we'll start with something simpler: a loop that counts from 100 (decimal) down to zero.

```
        ldy #100        ; Y = 100
Loop    dey             ; subtract 1 from Y
        bne Loop        ; repeat until Y == 0
```

Here we have three instructions and one label named `Loop`. In our dialect of 6502 assembler (DASM), instructions are always

indented, and labels are always flush against the left margin. Labels can be on their own line or coexist with an instruction. Comments are denoted with a ";" and go until the end of the line.

The first instruction `LDY` (LoaD Y) loads the Y register with a constant value, 100. Constants start with a "#" and tell the assembler to use the value directly, not as a memory-load or memory-store instruction.

The next instruction `DEY` (DEcrement Y) subtracts 1 from the Y register. It also sets the Zero (Z) flag in the CPU, which is an internal bit that is set to 1 if the result of an instruction is zero. We use these *flags* to test for conditions in the code.

The final instruction `BNE` (Branch Not Equal) is a branch instruction, which means the next instruction may be one of two choices. `BNE` transfers control to its target label if the Z flag is unset, and will fall through to the next instruction if it is set. In our case, since `DEY` just set the Z flag, the branch will be taken until the Y register decreases to zero, and so the loop will repeat 100 times.

Let's make a loop that uses the different *addressing modes* of the 6502. These allow you to target areas of memory beyond a single constant location, by adding another register to an address. For example, this demonstrates the *absolute indexed* addressing mode with the `STA` instruction:

```
        lda #0          ; A <- 0
        ldy #$7F        ; Y <- 127
Loop    sta $100,y      ; store A in [$100+y]
        dey             ; decrement Y, set flags
        bne Loop        ; repeat until Y == 0
```

This loop makes use of two registers, A and Y. A is initialized to zero and Y counts down from `$7F` (127) to zero. The `STA` (STore Accumulator) instruction stores A to an address at every loop iteration. We use the addressing mode "absolute,indexed" here, which means we compute the destination address by adding a register (Y in this case) to a constant (`$100` in this case). Since Y counts from `$7F` down to zero, we'll store A (which we set to 0) to addresses `$17F` to `$101` in decreasing order.

In 6502 parlance, the *absolute indexed* mode means add an 8-bit value (Y register in this case) to a 16-bit constant. There is another mode, *zero page* mode, which operates only on 8-bit values. Zero page refers to the memory locations `$00-$FF` which get special treatment. Instructions using zero-page addressing modes generate smaller code, and most of the VCS registers live in zero-page space.

There are restrictions to these modes, and all combinations do not have a corresponding encoding. For example, only X and Y can be used as indices, the A register cannot be used as an index. Also, Y can only be used as a zero-page index with the `LDX` and `STX` instructions – otherwise it is expanded to an absolute index. Your assembler will throw an error if you try to use an invalid addressing mode.

Our last loop has a problem, though. We used the `BNE` instruction to repeat the loop until Y is zero. But since the store happens *before* we decrement Y, we don't store anything when Y is zero (i.e. at address `$100`). To fix this, we just change the loop so that the `DEY` happens before the `STA`, and add 1 to the starting Y value:

```
        lda #0
        ldy #$80        ; Y <- 128
Loop    dey             ; set flags
        sta $100,y      ; does not modify flags
        bne Loop        ; repeat while Y != 0
```

Since `STA` does not modify any flags, we can `DEY` first (which does modify flags) and then exit the loop when Y==0 rather than Y<0. There will be lots of opportunities to tweak loops like this for optimal performance, and VCS programming often demands it.

We could also count upwards from zero using the CPY (ComPare Y) instruction:

```
        lda #0
        tay             ; Y <- 0
Loop    sta $100,y
        iny
        cpy #$80        ; set flags as if (Y - 128)
        bne Loop        ; branch until Y == 128
```

The CPY instruction performs a comparison: It subtracts the operand from the Y register and sets flags, but discards the result. So in this example if Y is $80, (Y-$80) will be zero and the Zero flag will be set.

We can also compare the A register with CMP (CoMPare accumulator) and the X register with CPX (ComPare X register).

1.6 Condition Flags and Branching

We've covered the Z (Zero) flag already, but there are others. Here's the list of condition flags you'll be using most often:

Flag	Name	Description
Z	Zero	Set when the result is zero.
N	Negative/Sign	Set when the result is negative (high bit set).
C	Carry	Set when an arithmetic operation wraps and carries the high bit.
V	Overflow	Set when an arithmetic operation overflows; i.e. if the sign of the result changes due to overflow.

Table 1.3: Condition Flags

A lot of instructions just set the Zero and Negative flags, which makes it easy to test for zero values or to test the high bit. The Carry flag is set by compare, add, subtract, and shift operations.

The Overflow bit is less commonly used than the Carry bit, but it's worth explaining the difference between *wrapping* and *overflow*.

When we say a value *wraps*, we mean that an operation exceeds the boundaries of its byte and the result is truncated. So if you add `$01` to `$FF`, you'll wrap around to `$00`.

Overflow is set when the result of a addition or subtraction changes its sign – for example, `$40 + $40 = $80` which overflows because `$80` is a negative number in two's complement representation. If you are using unsigned numbers, you can generally ignore this flag.

Mnem.	Description	Flag Test	Condition
BNE	Not Equal	Zero clear	A != B
BEQ	Equal	Zero set	A == B
BCC	Carry Clear	Carry clear	A < B (unsigned)
BCS	Carry Set	Carry set	A ≥ B (unsigned)
BMI	Minus	Negative set	A < B (signed)
BPL	Plus	Negative clear	A ≥ B (signed)
BVC		Overflow clear	no signed overflow
BVS		Overflow set	signed overflow
JMP	Jump	—	always taken

Table 1.4: Branch Instructions

The `JMP` instruction doesn't test any flags but just moves the PC directly to the target. The branch instructions can only modify the PC by -128 to +127 bytes, so for longer distances you'll need `JMP`.

It's good to memorize the `BCC` (less than) and `BCS` (greater than or equal) instructions, since these are used often. Also note that the `BPL` and `BMI` instructions are the same for signed quantities, so we could use them to stop when a value goes negative, like this:

```
        lda #0          ; A <- 0
        ldy #$7F        ; Y <- 127
Loop    sta $100,y      ; store A in [$100+y]
        dey             ; decrement Y, set flags
        bpl Loop        ; repeat until signed(Y) < 0
```

Note that this technique would not work if we started with Y = `$81` or higher, because the first `DEY` would result in a negative number, exiting the loop on the first iteration!

1.7 Addition and Subtraction

We've covered DEY, but there is a whole group of instructions that increment (add one) or decrement (subtract one):

```
DEC  -1 from memory location
DEX  -1 from X register
DEY  -1 from Y register
INC  +1 to memory location
INX  +1 to X register
INY  +1 to Y register
```

There's no INC or DEC for the A register, but you can add or subtract the A register to/from another memory location or constant. ADC adds, and SBC subtracts. An example of addition:

```
lda $81 ; load memory location $81 -> A
clc     ; clear carry flag
adc #10 ; add 10 to A
sta $82 ; store A -> memory location $82
```

Note the CLC (Clear Carry Flag) instruction. The ADC instruction adds the Carry flag to the result (0 or 1) so usually it must be cleared before addition. For subtraction, it must be set first using SEC (Set Carry Flag):

```
lda $81 ; load memory location $81 -> A
sec     ; set carry flag
sbc #10 ; subtract 10 from A
sta $82 ; store A -> memory location $82
```

The increment/decrement instructions modify the Negative and Zero flags, while the addition/subtraction additionally modify the Carry flag.

1.8 The Stack

In computing terminology, a *stack* is a list of values that can grow and shrink. You grow the stack by *pushing* a value on top, and shrink by *pulling* a value off the top.

On the 6502, the stack is stored in RAM, and the top of the stack is a memory location stored in the S (Stack pointer) register. It usually starts at `$FF`.

The `PHA` instruction pushes the A register to the stack, storing it to the memory location pointed to by S. It then decrements S by 1. We say the stack "grows upward" because the stack pointer decreases as new values are added.

You can retrieve the top value on the stack with the `PLA` instruction. It first increments S by 1, then reads the location pointed to by S into A.

Another important instruction that uses the stack is `JSR`. It pushes the Program Counter to the stack, then transfers control to another location, just like a `JMP`. When the `RTS` instruction is encountered, the CPU pulls the top address off of the stack and transfers control there. We'll demonstrate this in Chapter 11.

1.9 Logical Operations

The "logical" instructions combine the bits of the A register and the operand, performing a bit (logic) operation on each bit.

`AND`	A&B	Set bit if A and B are set.
`ORA`	A|B	Set bit if A or B (or both) are set.
`EOR`	A^B	Set bit if either A or B are set, but not both (exclusive-or).
`BIT`	A&B	Same as AND, but just set flags and throw away the result.

Table 1.5: Logical Instructions

For example, let's combine `$55` and `$f0` with the `AND` operation:

```
        lda #$55
        and #$f0
```

For `AND`, if a bit was set in both the A register and the operand, it'll be set in A after the instruction executes:

```
        $55  01010101
AND     $f0  11110000
--------------------
        $50  01010000
```

The `AND` operation is useful for limiting the range of a value. For example, `AND #$1F` is the same as `(A mod 32)`, and the result will have a range of 0..31.

What if we did an `ORA` instead?

```
        $55  01010101
ORA     $f0  11110000
--------------------
        $f5  11110101
```

`ORA` sets bits if they are set in either A or the operand, i.e. unless they are clear in both.

What about an `EOR`?

```
        $55  01010101
EOR     $f0  11110000
--------------------
        $a5  10100101
```

`EOR` (exclusive-or) is like an `OR`, except that bits that are set in both A and the operand are cleared. Note that if we do the same `EOR` twice, we get the original value back.

1.10 Shift Operations

ASL	Shift Left	Shift left 1 bit (multiply by 2), bit 7 → Carry
LSR	Shift Right	Shift right 1 bit (divide by 2), bit 0 → Carry
ROL	Rotate Left	Same as ASL except Carry → bit 0
ROR	Rotate Right	Same as LSR except Carry → bit 7

Table 1.6: Shift and rotate instructions

There is also the family of "shift" operations that move bits left and right by one position within a byte. The bit that is shifted off the edge of the byte (i.e. the high bit for shift left, and the low bit for shift right) gets put into the Carry flag.

The "rotate" operations are similar, but they also shift the previous Carry flag into the other end of the byte. So for rotate left, the Carry flag is copied into the rightmost (low) bit. For rotate right, it's copied into the leftmost (high) bit.

Example of ASL (shift left):

```
        lda #$83
        asl     ; shift left
```

Result (C means carry flag is set):

```
                $83  10000011
ASL     ->      $06  00000110   C
```

Remember that just like decimal notation, we consider the "leftmost" bit to be the most significant. So if we shift left one bit, we are essentially multiplying by 2. If we shift right one bit, we essentially divide by 2, discarding the remainder.

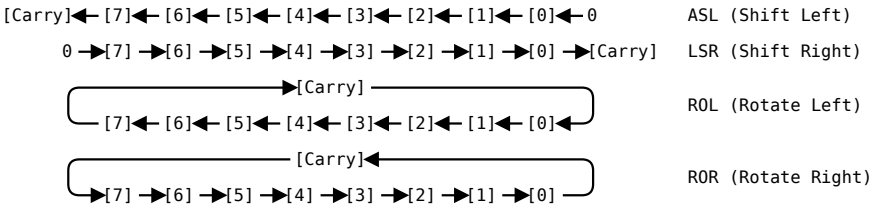

Figure 1.4: Shift and rotate bit flow

Another example, this time of ROR (rotate right):

```
        lda #$03
        sec     ; set carry flag
        ror     ; rotate right
        ror     ; rotate right
        ror     ; rotate right
```

Note that we SEC to set the carry first. Here's the result:

```
                $03  00000011   C
ROR     ->      $81  10000001   C
ROR     ->      $81  11000000   C
ROR     ->      $81  11100000
```

Note that if you ROL or ROR nine times in succession, you'd have the original byte.

Now that you have a working knowledge of the 6502, we'll use an online tool to program it in the next chapter.