# DESIGNING VIDEO GAME HARDWARE IN VERILOG

*An 8bitworkshop Book*

by Steven Hugg

**Designing Video Game Hardware in Verilog**
Copyright ©2018 by Steven Hugg

All rights reserved. No part of this book may be reproduced without written permission from the author.

First printing: October 2018

**Disclaimer**
Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions, nor is any liability assumed for damages resulting from the use of the information contained herein. No warranties of any kind are expressed or implied.

**Trademarks**
Some brands and product names mentioned in this work are trademarks or service marks of their respective companies. Use of a term in this work should not be regarded as affecting the validity of any trademark or service mark.

**Inquiries**
Please refer all inquiries to info@8bitworkshop.com.

# Contents

# Preface

The microprocessor was invented in the mid-1970s, but even before then, video arcade games existed.

These odd beasts were tucked away in unusual places – a movie theater lobby, a department store, the smoking section of a pizza parlor. Their (usually) black-and-white monitors would trace some simple shapes moving across the screen, silently implying that you might insert a coin and grab the wheel, paddle, periscope, or some other control device.

Unless you were born in this era, you probably don't remember many of these games – *Tank*, *Pong*, *Indy 800*, *Sea Wolf*, *Monaco GP*. They were quickly ushered out as they fell into disrepair, replaced by the next generation of technology. Some of these games were forgettable; some were reinvented in software and lived on.

In this era, video games were designed as circuits, not as programs. It seems impossible that someone could make an entire video game out of assorted chips and wires, but that's what happened. To these digital designers on the vanguard of technology, it was perfectly natural. Their job was to select from a stew of variously-priced components and make a marketable, profitable game for the lowest cost.

Moore's law was in full swing back then, and it wasn't long before commodity CPUs, graphics, and sound processors rendered the early bespoke designs obsolete. Software became king, and arcade games became computers with heavy cabinets. In some

cases, gamers could take the software home and play it on their television.

This book attempts to capture the spirit of the "Bronze Age" of video games. We'll delve into these circuits as they morph from *Pong* into programmable personal computers and game consoles.

Instead of wire-wrap and breadboards, we'll use modern tools to approximate these old designs in a simulated environment from the comfort of our keyboards.

At the end of this adventure, you should be well-equipped to begin exploring the world of FPGAs, and maybe even design your own game console.

*Many thanks to dB Baker for draft review!*

# 1

# Boolean Logic

> "Contrariwise, if it was so, it might be; and if it were so, it would be; but as it isn't, it ain't. That's logic."
>
> Lewis Carroll

## 1.1 Boolean Algebra

All digital computers are designed to operate by the rules of *Boolean algebra*, introduced by George Boole in 1847.

In Boolean algebra, all values are *Boolean* values, which means there are only two possibilities: true or false.

A Boolean is equivalent to a *bit*, which also has two possible values: 0 or 1. We'll use the terms Boolean and bit interchangeably in this book.

You can build Boolean expressions with the basic logic operations AND, OR, and NOT:

| Expression | Result |
|---|---|
| A AND B | true if both A and B are true, otherwise false |
| A OR B | false if both A and B are false, otherwise true |
| NOT A | true if A is false, false if A is true |

You can also use *De Morgan's laws* to rewrite Boolean expressions:

NOT (A AND B) = (NOT A) OR (NOT B)
NOT (A OR B) = (NOT A) AND (NOT B)

From these basic operations, you can derive the NAND (NOT-AND), NOR (NOT-OR), and XOR (eXclusive-OR) operations:

| Expression | Result |
|---|---|
| A NAND B | false if both A and B are true, otherwise true |
| A NOR B | true if both A and B are false, otherwise false |
| A XOR B | false if A = B, otherwise true |

NAND and NOR operations are *functionally complete*, meaning any logic operation can be expressed in terms of them. For example, here's how to convert (A OR B) to NAND using De Morgan's laws:

A OR B = NOT ((NOT A) AND (NOT B))
A OR B = (A NAND A) NAND (B NAND B)

This functional completeness is why computer chips can exclusively use NAND and NOR gates as universal building blocks.

## 1.2 Truth Tables

A *truth table* shows the result of a given Boolean function for every possible combination of inputs.

The truth table for the NOT operation is simple, since it only has one input, and so only two possibilities:

| A | NOT A |
|---|---|
| 1 | 0 |
| 0 | 1 |

Here are truth tables for the AND, OR, and XOR operations:

| A | B | A AND B | A OR B | A XOR B |
|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 0 | 0 | 0 | 0 |

The NAND, NOR, and NXOR operations are just negations of the above – e.g. (A NAND B) is the same as NOT (A AND B).

| A | B | A NAND B | A NOR B | A XNOR B |
|---|---|---|---|---|
| 1 | 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 |

## 1.3 Combinational Logic

A Boolean expression or its equivalent truth table is an example of *combinational logic*. This means that the present output depends solely on the present input, without any memory or retained state. Programmers might call this a *pure function*.

One nice thing about combinational logic is that we have powerful tools to analyze and optimize it, so we can build efficient logic circuits with a minimal number of gates. To reduce a Boolean expression to a less complex form, we can draw a *Karnaugh map* on paper. Nowadays, powerful heuristic computer algorithms like *ESPRESSO* can optimize dozens of variables.

Once a circuit has any kind of memory (registers, RAM, flip-flops, etc.), then it becomes *sequential logic*, which is more complicated to describe and analyze.

## 1.4 Binary Numbers

We can represent integers with bits by using *binary notation*. Each digit is a bit, and we list them in order, for example: `00011011`.

The least-significant bit, the rightmost, has a numeric value of 1. The next bit to the left has a value of 2, the next has 4, and so on. To find the integer value, we add up all the values of the bits that are set:

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

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

## 1.5 Addition and Subtraction

Computers add and subtract numbers using binary digits. Each binary digit is added, and if you have to carry the 1 (which only happens when adding the bits 1 and 1), you carry it to the next bit. An example of adding two 8-bit numbers:

| Bit # | | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|
| Byte A | | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| Byte B + | | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 |
| Carry | 1 | | 1 | | | 1 | | | |
| = | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |

*Table 1.1: Binary Addition of Two Bytes With Carry*

Our number is only 8 bits long, so the result of the addition is *truncated* to 8 bits. We also could call this an *integer overflow*. The extra carry bit could be used in another calculation, or saved for later.

An unsigned byte can only represent the values 0 to 255, so if $a + b >= 256$ then the 8-bit result is $(a + b) \bmod 256$ (mod is the *modulus* operator, which is basically the remainder of a division.) For 16-bit unsigned numbers, it'd be $(a + b) \bmod 2^{16}$ (65536).

If the two values have different bit lengths, we align them both by their rightmost digit, just like we would for decimal numbers.

## 1.6 Logical Operations

Logical operations work just like addition and subtraction, except there's no carry from one digit to the next. We just perform the logical operation on each bit independently:

| Bit # | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Byte A | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| Byte B | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 |
| A & B (AND) | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| A \| B (OR) | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 |
| A ^ B (XOR) | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 |

*Table 1.2: Binary AND, OR and XOR*

## 1.7 Signed vs. Unsigned

We've seen how binary numbers can be interpreted as a non-negative or *unsigned* integer. We can also interpret them as *signed* numbers, meaning that they have an implied sign (positive or negative).

This requires a trick known as *two's complement* arithmetic. To turn a positive integer into a negative, we flip all the bits, then add one. For example:

| Binary value | Unsigned value | Signed value | Operation |
|---|---|---|---|
| 00000001 | 1 | 1 | original value |
| 11111110 | 254 | -2 | flip all bits |
| 11111111 | 255 | -1 | add one |

In this scheme, half of possible values for a signed integer are positive, and half are negative. If the high (leftmost) bit is 1, we treat the value as negative:

| Binary value | Unsigned value | Signed value |
|---|---|---|
| 00000000 | 0 | 0 |
| 00000001 | 1 | 1 |
| 01111111 | 127 | 127 |
| 10000000 | 128 | -128 |
| 11111110 | 254 | -2 |
| 11111111 | 255 | -1 |

Here, integer overflow works to our advantage. For example, if we add 255 to an 8-bit value (whose maximum value is 256) it's the same as subtracting 1. Adding 254 is the same as subtracting 2. Most computers perform integer subtraction via this method.

Note that there's nothing in the binary number identifying it as signed — it's all in how you interpret it.

## 1.8 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 1.3: 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 1.4: Example Hex Conversion*

We see in Table 1.4 that the decimal number 27, represented as `11011` in binary, becomes `1B` in hexadecimal.

# 2

# Discrete Hardware

> "We may say most aptly that the Analytical Engine weaves algebraic patterns just as the Jacquard loom weaves flowers and leaves."
>
> Ada Lovelace, mathematician

## 2.1 History of the Logic Gate

In 1886, the American scientist Charles Sanders Peirce recognized that logical operations could be carried out by electrical switching circuits. One of his students, Allan Marquand, built a "logical machine" using a hotel annunciator and 16 electromagnetic coils. Pierce foresaw the need for much larger logical machines "corresponding to a Jacquard loom", using the same analogy as did Ada Lovelace decades prior.[1]

At the time, mechanical relays were the most common switching element, used first in telegraph systems to amplify signals, and later in telephone exchanges. In 1930, Bruno Rossi used vacuum tubes in the first practical *coincident circuit* (AND gate) to count electrons in Compton scattering experiments.

Logic design remained more-or-less "ad hoc" until 1936, when Akira Nakashima published the first paper on switching circuit theory. Claude Shannon would later cite it to prove that systems of relays could solve Boolean algebra problems. Designers could now use *De Morgan's laws* to analyze and simplify circuits.

The first practical transistor was invented in 1947 at Bell Labs, and soon computers were built using thousands of semiconductor devices. These devices continued to shrink, and British scientist Geoffrey Dummer proposed that several semiconductors could be embedded into a solid block. Jack Kilby (Texas Instruments) and Robert Noyce (Fairchild) turned this theory into practice in 1958, and thus the *integrated circuit* (or *IC*) was born.

## 2.2 TTL Logic

The first integrated circuits used *resistor-transistor logic*, where gates are constructed from resistor networks connected to a single transistor.[1] Later, *diode-transistor logic* would also be used, where diode networks performed the gating operation and a transistor provided amplification.

These technologies were largely made obsolete by *TTL* (*transistor-transistor logic*). TTL uses bipolar transistors which perform both gating and amplification. Its low cost, speed, noise immunity, and ease of manufacturing made it a dominant digital technology in the 1960s and 1970s.

TTL is a technology as well as a standardized interface. The term "*TTL-compatible*" refers to devices which communicate using compatible logic levels. In TTL logic, a signal between 0 and 0.4 volts is considered "low" (zero) and a signal between 2.4 and 5 volts is considered "high" (one).

*CMOS* technology was developing at the same time as TTL. It was especially suited for CPU designs, using MOSFETs[2] for lower power consumption and better miniaturization. Every commercial CPU today is made with a variant of CMOS technology.

[1] The Apollo Guidance Computer was constructed from RTL ICs.

[2] metal-oxide-semiconductor field effect transistor

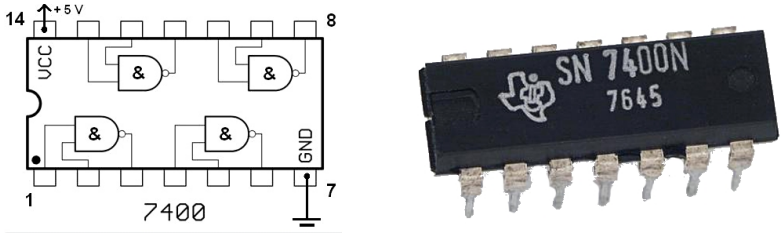



*Figure 2.1: TI 7400 4-NAND chip in PDIP package, with schematic (CC SA 3.0, by Stefan506)*

## 2.3 Discrete Logic and the 7400 Series ICs

A circuit composed mainly of logic gates is called a *discrete* or *digital* logic design. A discrete design handles mostly logical values (ones and zeroes) as opposed to an *analog* design which manipulates continuous values based on voltage. Video games of the 1970s were often a combination of discrete and analog.

*Discrete logic* also refers to the limited-function IC chips used in circuit designs, especially the *7400 series* – a popular family of TTL chips first produced in the 1960s and 1970s.[3] This series contains many varieties of logic gates, flip-flops, and counters, as well as complex *ALU*s for building custom CPUs.

For example, the chip in Figure 2.1 is a *quad-NAND gate* chip. It provides four independent NAND gates, each with two pins for input and one pin for output. One pin must be connected to $V_{cc}$ (+5 volts) and one pin to ground (+0 volts).

> FUN FACT: The seminal Atari game *PONG* was a discrete design using 66 IC chips, most of them in the 7400 family.

[3] Many of these chips also had/have CMOS versions, but many were TTL-logic-level compatible.

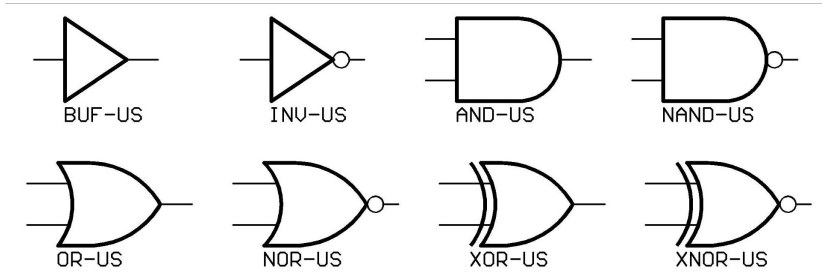



*Figure 2.2: Logic gates, IEEE standard*

## 2.4 Circuit Diagrams

A *circuit diagram* shows how electronic components are wired up. We'll deal mainly with logic gates in this book. Generally, inputs enter a gate on the left side, and outputs exit on the right.

There are two sets of symbols used to depict logic gates: the US-based Institute of Electrical and Electronics Engineers standard (IEEE Std91a-1991) and the International Electrotechnical Commission standard (IEC 60617-12). Since this book is about 1970s-era video games, we'll use the IEEE standard here.

The logic gate symbols are shown in Figure 2.2. Let's note a few things:

- A dot to the right of the gate means the output is inverted. Note that NAND, NOR, and XNOR all have dots.
- The BUF gate is a *buffer*, whose output is the same as its input. These are used in the "real world" to stabilize and isolate signals, but we won't deal with them much here.
- The INV (inverter) gate is the same as a NOT gate. Note that it's just a BUF with an inversion dot.

You might also see *multiplexers* (MUX) which has multiple inputs and a single output. These devices select one of the inputs and forward it to the output, using a control signal. They are often represented as trapezoids.

# 3

# Clocks and Flip-Flops

## 3.1 Sequential Logic

We've already discussed *combinational logic* which only considers the values of current inputs, not past inputs. To do anything interesting, your circuit has to have a memory.

*Sequential logic* is logic that considers the present, but also can remember the past. Components called *flip-flops* and *latches* store and recall individual bits.

This kind of logic is driven by a *clock signal*, each cycle advancing the circuit to the next state. Because all activity is synchronized to a clock, this is also called *synchronous logic*.

Most modern circuits use both combinational and sequential logic. One exception is a *ROM* (*read-only memory*) chip, which might be considered a combinational circuit since its state never changes, regardless of its inputs.

## 3.2 Clocks

In a *synchronous circuit*, all activity is synchronized with the *clock signal*. The *clock* is an approximate square wave which alternates between low and high states (i.e. 0 and 1) at a fixed *frequency*. For example, a 1 MHz clock changes from low to high and back again one million times per second.

Clock signals are usually generated by a crystal oscillator and a series of *clock divider*s (see Chapter 7.) A circuit often contains multiple clocks with different frequencies and/or phases.

## 3.3 Flip-Flops

A *flip-flop* is an essential component of a sequential logic circuit. It remembers one bit (0 or 1) which can only change at the edge of a clock signal.

A flip-flop has at least one *control input* and a *clock input*. There are multiple types of flip-flops, each of which responds to its inputs differently.

Flip-flops are *edge-triggered*. Their state only can change during the rising or falling edge of a clock signal – when the signal transitions from low to high (called the *rising edge* of the clock) or high to low (*falling edge*).

The signal must be stable (i.e. not changing) for a specific time period both before and after the clock edge transition. Violating these constraints results in a *setup time violation* or *hold time violation*, and leads to *metastability*. This makes your circuit go crazy, and is generally a bad thing.

## 3.4 Latches

A *latch* is similar to a flip-flop, but it is not synchronous – i.e. it does not react to a clock edge. Latches are *level-sensitive*.

The "D" latch is often called a *transparent latch*. It has an Enable input and a D (Data) input. When Enable is asserted, the latch is "transparent" – the input propagates directly to the output.

Generally speaking, you should avoid latches and only use flip-flops in your design. Since latches are *asynchronous* – aren't synchronized with the clock the same way as flip-flops – subtle flaws may emerge and lead to *metastability*.

# 4

# HDL (Hardware Description Language)

Before the 1980s, circuit design was largely a manual process. Designers would draw a schematic with pen and paper, then use truth tables and elbow grease to minimize the number of gates. To build an integrated circuit with their design, they'd hand-draw gates onto huge Mylar sheets, with limited or non-existent assistance from computer simulators.

In the 1980s, *electronic design automation* (*EDA*) tools became available. This era introduced the concept of the *hardware description language* (*HDL*): programming languages which could both describe and simulate a digital circuit.

Such languages can perform *logic synthesis* – automatically compiling the program into a network of logic gates, also known as a *netlist*.

The resulting netlist can be used for analysis, or translated into wires and gate masks to be printed on silicon. Ideally, this would all happen with minimal intervention from the designer.

The two dominant HDLs are *Verilog* and *VHDL*. They are functionally similar, though VHDL is a little more verbose and strict, and Verilog is a little more ambiguous and succinct. In this book we use Verilog, though all of the core concepts could apply to VHDL.

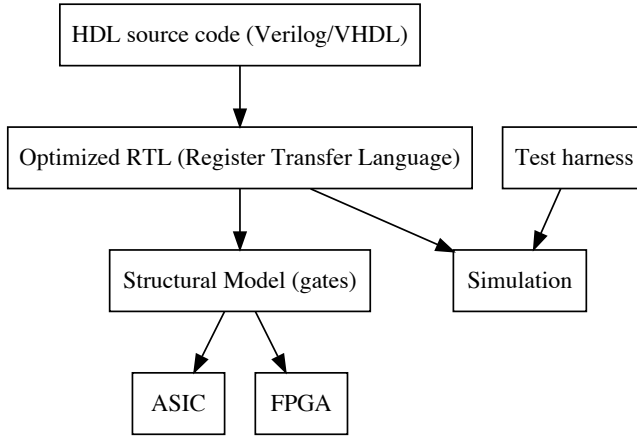



*Figure 4.1: EDA (Electronic Design Automation) tool chain flow*

## 4.1 Simulation

HDLs might initially seem a little weird for someone with a traditional programming background. Most mainstream programming languages are oriented around *control flow*, while HDLs are concerned with *data flow*.

Running a Verilog program is called *simulation*, and it occurs in discrete time steps. Often, each time step corresponds with the main clock – so a single clock cycle takes two simulator steps, one for the rising edge and one for the falling edge.

At each time step, the simulator might add events to a queue, and it might execute events from a queue. However, it can execute events in any order! This is a source of *non-determinism* that is tricky to deal with, and often leads to *race conditions*.

We'll cover some strategies for minimizing race conditions in this book. One thing to remember is that you're building a circuit, where everything is happening at the same time – not a program, where things work step-by-step.

## 4.2 Hardware Modeling

Software has various levels of abstraction – high-level languages, low-level languages, and assembly code. Similarly, HDLs have multiple levels of abstraction for modeling hardware:

- *Behavioral model* - The highest-level of modeling. Describes the behavior of the target hardware, but cannot be synthesized.
- *RTL* model - *Register transfer level*. Describes the hardware in terms of combinational and sequential logic, and is *synthesizable* into a gate-level model. **All the Verilog modules in this book are RTL models.**
- *Structural model* or *gate-level model* - Describes the hardware as a network of gates, flip-flops, and wires. Can be output as a *netlist* and loaded into a *FPGA* or burned into a custom *ASIC*.
- *Switch-level model* - Describes a network of individual transistors, lower-level than gates. Not covered in this book.

In theory, any code that executes on a computer is synthesizable (there are even synthesis tools for C, C++, Java, and .NET programs) but each design tool has limitations on what is acceptable. In this book we'll limit ourselves to a subset of Verilog RTL that our tools will synthesize.[1]

[1] The subset of *Verilog 2005* supported by the open-source *Verilator* tool.

# 5

# Intro to Verilog

## 5.1 Modules

The high-level building block of Verilog is a *module*.

Think of a module as a chip. A module has a number of *ports*, which correspond to the pins of the chip. The code you write in the module corresponds to the internals of the chip.

Here is a simple module which implements an AND gate:

```
module and_gate(a, b, y);
    input a, b;         // input ports
    output y;           // output port

    assign y = a & b;   // (a AND b)
endmodule  // no semicolon here!
```

In this example, we define the `and_gate` module with input ports named `a` and `b`, and an output port named `y`.

The `assign` keyword allows us to write combinational logic. The expression `a & b` essentially builds an AND gate, and `assign` connects its output to `y`.

Verilog supports *gate-level modeling*, which means explicitly writing out your gates instead of letting the compiler infer them from your code. For example, the `assign` statement in the previous example could be replaced with this:

```
    and U1(y, a, b);
```

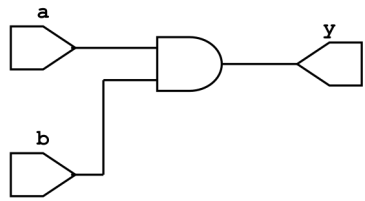



*Figure 5.1: and_gate module logical diagram*

But since HDL tools are really good at finding the optimal mix of gates to use, gate-level modeling is only used for specific applications.

## 5.2 Wires

Verilog variables come in two flavors: *variables*, which store things, and *nets* or *wires*, which connect things.

A wire doesn't store a value, but instead continuously assumes the value of its *driver*. This is analogous to how a gate drives a high or low voltage to a physical wire. In Verilog, a wire driver is usually an `assign` statement.

Typically, a wire will only have a single driver. You'll also get an error if a variable has no driver at all, unless it's the input of a module. Any number of connections can read from a variable, though.

## 5.3 Regs

The other common data type is *reg*, which is like a boolean variable in a traditional programming language. It remembers the last value assigned to it.

You can only assign a reg in an *always block*, which is triggered by an event, like a clock edge or reset signal. This is how we write *sequential logic* in Verilog.

## 5.4 Vector

A *bit vector* is a fixed-size array of bits, and can be applied to any type – wire, reg, input, output, etc.

The syntax for bit vectors is `[high bit index:low bit index]`. The rightmost (least significant) bit has index 0. For example, we can declare a 4-bit register like this:

```
reg [3:0] counter;  // 4-bit register
```

You can even concatenate bits or bit vectors together to make a longer value, like this:

```
output = {b, g, r};
```

You can extract a single bit or multiple bits using an array syntax:

```
counter[0]          // rightmost bit (low bit)
counter[3]          // leftmost bit (high bit)
counter[2:0]        // rightmost three bits
// this is the same as counter[2:0]
{counter[2], counter[1], counter[0]}
```

You can also treat a bit vector like a multi-bit integer, performing arithmetic and bitwise operations with it:

```
counter <= counter + 1;     // add 1 to counter
```

## 5.5 Literals

We can express a *literal* (a constant value) in several ways. The default is good old *decimal*, or *base 10*, but we can write literals in *hexadecimal*, *binary*, or even *octal notation*:

```
1234            // decimal
'b100101        // binary
'h8001          // hexadecimal (base 16)
'o12            // octal (base 8)
```

These are all *unsized literals*. For some operations, like concatenation or arithmetic, you need to make them *sized literals*, or you'll get a compilation error. You can just prepend the length in bits before the ':

```
8'1234          // 8-bit decimal
6'b100101       // 6-bit binary
16'h8001        // 16-bit hexadecimal
6'o12           // 6-bit octal
```

## 5.6 Blocking vs. Non-Blocking Assignments

Verilog has two *assignment operators*, *non-blocking* "<=" and *blocking* "=". For *synthesizable* code (the only kind we'll be writing in this book) these operators have special rules which are summarized in the table below.

| Keyword | Logic | Assign To | Operator |
|---|---|---|---|
| `always @(posedge clk)` | sequential | `reg` | `<=` |
| `always @(*)` | combinational | `wire` | `=` |
| `assign` | combinational | `wire` | `=` |

We'll revisit this table often in this book, so don't worry about absorbing it right now. Just remember that there are two distinct ways of assigning variables, the sequential (discrete) method, or the combinational (continuous) method.

# 6

# The 8bitworkshop IDE

In this chapter, we'll discuss the tools we'll use to develop and test our Verilog code. These tools comprise our interactive development environment, or IDE.

To start the IDE, go to `http://8bitworkshop.com/?preset=verilog` in a web browser that supports Javascript (for best results, use a recent version of Google Chrome, Mozilla Firefox, or Apple Safari).

## 6.1 Verilog Simulator

The IDE includes a *simulator* which compiles and executes your Verilog code clock cycle-by-cycle. It is translated into JavaScript, then executed in the browser.

In the default *scope view*, the IDE displays the waveforms of input and output signals. You can scroll forward/backward in time, and drag the cursor left and right to see numeric values at specific times.

In the *CRT view*, the IDE simulates a *CRT monitor* in real-time, allowing us to develop video games. This view is displayed when certain output signals are present in the main module. We'll talk more about how this works in Chapter 9.

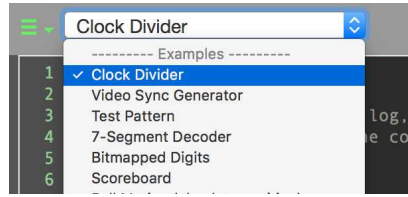



*Figure 6.1: IDE Pulldown*

## 6.2 Verilog Editor

The IDE is packaged with several example Verilog files, each roughly corresponding to a chapter in this book. At the top left of the screen, you can access a pulldown menu that allows you to select a file to load.

You can edit these files as much as you want – the browser will persist all changes locally and they'll be there if you close the tab and come back. To rollback all changes to an example file, select **File | Revert to Original** in the menu.

If you drag the leftmost gutter control to the right, you can expose the *sidebar* which lists all other files included in this project. Click on a file to view and edit.

The buttons at the top of the screen perform several debugging functions:

*Figure 6.2: Debugging Functions*

- **Reset** - Reset the simulator.
- **Pause** - Pause the simulator.
- **Run** - Resume the simuator after pausing.
- **Edit Bitmap** - Open the Bitmap Editor. Works with specially tagged data arrays.
- **Start/Stop Replay Recording** - Record and rewind simulator state.

There are some additional keyboard commands available after clicking on the *scope view*:

| | |
|---|---|
| + or = | Zoom in. |
| - | Zoom out. |
| Left | Move cursor left one clock cycle. |
| Right | Move cursor right one clock cycle. |
| Ctrl+Left | Move cursor left many cycles. |
| Ctrl+Right | Move cursor right many cycles. |
| Ctrl+Shift+Left | Move cursor to start of trace. |
| H | Toggle between decimal and hex numbers. |

## 6.3 Rate Control

When using the *CRT view*, the simulator's default clock rate is sufficient to display 60 frames per second. You can slow down the frame rate using the controls in Figure 6.3. If you go below 5 FPS, you'll start to see the electron beam scan the frame. You can also pull up the scope from the bottom of the screen and watch the signals evolve.


*Figure 6.3: Frame rate controls*

**NOTE:** The IDE is under active development and may differ from what we describe here.

# 7

# A Simple Clock Divider

To see it on `8bitworkshop.com`: Select the **Verilog** platform, then select the **Clock Divider** file.

For our first real Verilog module, we'll build a *clock divider*.

A clock divider takes a clock signal as input, and creates a new signal with the frequency divided by an integer. For example, a 10 MHz clock divided by 2 would become a 5 MHz clock.

Typically, a crystal oscillator generates a *master clock* at a specific frequency. Digital circuitry then shapes the signal into a clean square wave, and divides the clock down to lower frequencies for various purposes in the circuit.[1]

For this book we're going to assume the topmost module receives a clock signal called ***clk*** as input.

## 7.1 Timing Diagram

A *timing diagram* shows the relationship of multiple signals over time.

Figure 7.1 shows the timing diagram for our clock divider. The topmost row represents the main clock signal, `clk`. Note the

[1] Modern designs also have clock multipliers to turn MHz into GHz.

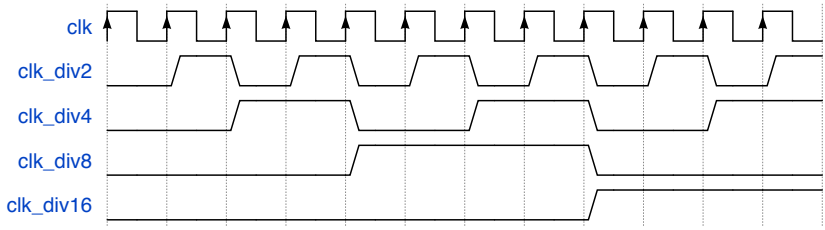



*Figure 7.1: Clock divider timing*

dashed vertical lines – each one represents a clock cycle, or two simulation steps in our simulator.

The next row shows the signal named `clk_div2`, which is the clock divided by 2. Note that its period (time between cycles) is twice as long as the main `clk` signal. Each successive row further divides the clock by 2.

You might note that `clk` looks a little different in the diagram. It's common to show the clock signal with square edges, since it's a stable reference signal. Other signals might be shown with slanted edges, as they take time to ramp up and down when responding.

Note also the arrows on the rising edges of the clock signal. This indicates that our flip-flops are triggered on the upward edge of the clock.

## 7.2 Ripple Counter

In the first method, we'll cascade several flip-flops in series. Each successive flip-flop will alternate between high and low on the upward edge of the previous signal. So, each successive signal output will have exactly half the frequency. (See the timing diagram in Figure 7.1 – each row represents an individual flip-flop.)

First we write the module declaration:

```
module clock_divider(
  input clk,
  input reset,
```

```
  output reg clk_div2,
  output reg clk_div4,
  output reg clk_div8,
  output reg clk_div16
);
```

Our inputs are two signals named `clk` (the clock) and `reset`, which the simulator feeds into the module. We also have four outputs, one for each successive divided-by-2 clock signal.

To drive our outputs, we're going to write `always` blocks. An `always` block is a Verilog process which is triggered by an event. The *sensitivity list* tells Verilog which conditions trigger the event.

Here, our sensitivity list is `(posedge clk)`, which means "on the positive edge of the `clk` signal". So this event will run each time the clock moves from 0 to 1. This is called a *sequential block* because the event happens on a clock edge.

```
  always @(posedge clk)
    clk_div2 <= ~clk_div2;
```

This `always` block says: Whenever the clock moves from 0 to 1, take the value of `clk_div2`, invert it (compute the logical NOT) then write it back.

Note that we added the `reg` keyword to each output in the module declaration. That's because we're driving these variables from an sequential `always` block. Therefore we can only use "`<=`", or non-blocking assignments, which can only drive `reg` variables. (Remember that table at the end of Chapter 5?)

## 7.3 Derived Clocks

We can cascade multiple signals with multiple `always` blocks:

```
  always @(posedge clk_div2)
    clk_div4 <= ~clk_div4;

  always @(posedge clk_div4)
    clk_div8 <= ~clk_div8;
```

```
always @(posedge clk_div8)
  clk_div16 <= ~clk_div16;
```

Instead of using `clk` in our sensitivity list, we're using another variable. When the `always` block that drives `clk_div2` fires, it could trigger the block that drives `clk_div4`, and so on.

**This is not always a good idea.** If you use a signal other than the main clock (`clk`) to trigger an event, it becomes a *derived clock*, and must be treated carefully. Careless use of derived clocks leads to excessive *clock skew* which leads to *metastability*. Modern design tools minimize clock skew by inserting small delays into the design.

The 1970s-era designs we're discussing in this book took clock skew and other design issues into account, so we'll also use derived clocks in this book to more easily demonstrate certain concepts.

## 7.4 Circuit Diagram

If we synthesized a circuit from this module, it'd look something like this:

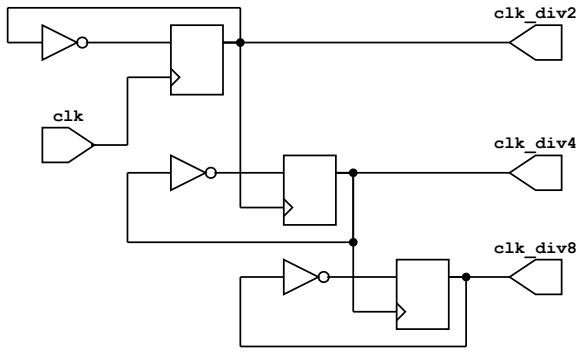



*Figure 7.2: Clock divider with cascading flip-flops (only 3 bits shown)*

Note the `clk` input pad on the left side, and the output pads on the right.

The square boxes are *D-type flip-flops*. The clock signals go into the control inputs on the bottom, and the data signal goes into the top. Whenever the control signal is high, the flip-flop remembers the value of the data signal, until the control signal is low again.

Note the feedback loop from the output through the inverter back to the input. This makes the flip-flop switch states whenever the clock signal goes high.

# 8

# A Binary Counter

Part of the power of Verilog comes from being able to manipulate multiple bits in a single expression. Instead of using flip-flops, we'll build a 4-bit counter using bit vectors.

A *binary counter* can easily divide a signal by powers of 2. This method uses a N-bit register which is incremented by 1 at each clock cycle. Successive bits represent successive divisions of the clock by 2. For example, bit 0 (the first bit) is $F_{clk}/2$, bit 1 is $F_{clk}/4$, etc.

(Food for thought: The ripple counter described in the previous chapter is really just a hand-made binary counter.)

First we declare our module, which has similar inputs and outputs, except we expose a 4-bit register called `counter`:

```
module clock_divider(
  input clk,
  output reg [3:0] counter,
  output cntr_div2,
  output cntr_div4,
  output cntr_div8,
  output cntr_div16
);
```

Note that only the `counter` output has the `reg` keyword. It will be driven from an `always` block, while the other outputs will be type `wire` (the default) and driven from combinational logic with an `assign` keyword.

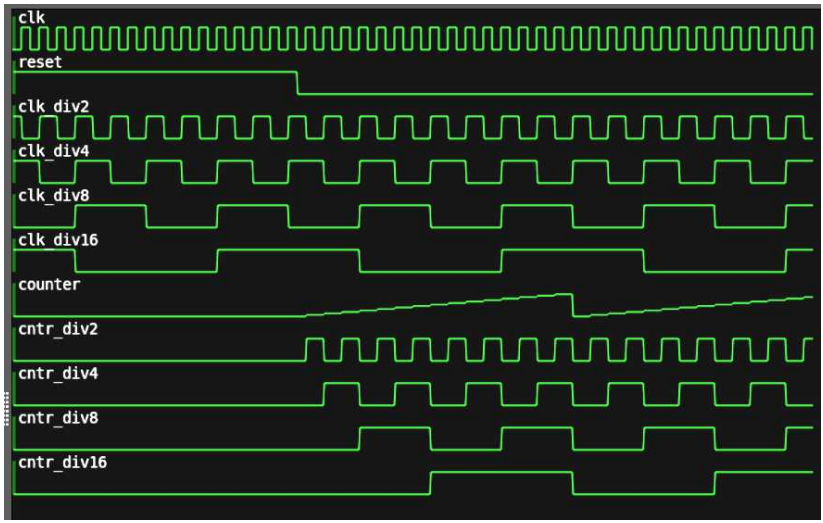



*Figure 8.1: Clock divider scope view in 8bitworkshop IDE*

We increment `counter` by 1 on each cycle of the clock:

```
  always @(posedge clk)
    counter <= counter + 1;
```

We use `assign` to extract each bit from the counter and expose it outside of the module:

```
  assign cntr_div2 = counter[0];
  assign cntr_div4 = counter[1];
  assign cntr_div8 = counter[2];
  assign cntr_div16 = counter[3];

endmodule
```

Note that we're using `assign` to drive these outputs, which do not have `reg` type – they default to `wire`. This is *sequential logic*, so we're using *blocking* "=" assignments. (Remember that table at the end of Chapter 5?)

(Note that these outputs duplicate the bits already exposed in the `counter` variable; this is just for debugging purposes.)

## 8.1 Synchronous Reset

In our simulator, all variables are initialized randomly on startup. We do this to emulate the behavior of real hardware, since flip-flops do not have a defined state when powered on (although in some *FPGA*s, they are initialized to zero.)

A *reset* condition on your synchronous `always` blocks can initialize registers to a known state.

To do this, we add an `if` statement to handle the case where `reset` is set:

```
always @(posedge clk)
begin
  if (reset)
    counter <= 0;
  else
    counter <= counter + 1;
end
```

Note that we also added a `begin` ... `end` pair to the `always` block. This allows us to add additional statements to the block. This isn't strictly necessary if you just have a single `if` statement, but it doesn't hurt either.

Why couldn't we have separate `always` blocks for `clk` and `reset`, like this?

```
always @(posedge clk)
  counter <= counter + 1;

always @(posedge reset)
  counter <= 0;
```

Seems simpler, right? But if you do this, the Verilog compiler will complain about "multiple driving blocks" and fail.

Verilog imposes this limitation upon you because it's how the hardware operates. A flip-flop can only be clocked from one source, so it's unclear how you'd build one that responds to multiple clock signals. Which signal would win in a race? It's indeterminate.

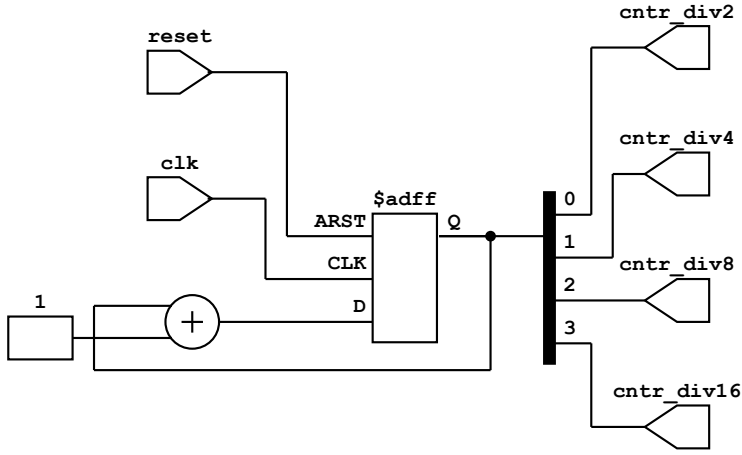



*Figure 8.2: Logical schematic of 4-bit counter circuit with asynchronous reset. The $adff block represents 4 separate D-type flip-flops, which have both CLK (clock) and ARST (asynchronous reset) inputs.*

## 8.2 Asynchronous Reset

A flip-flop can only have one clock signal, but it can also have a *reset signal*. This allows us to add an *asynchronous reset* condition to our always block. Asynchronous means it's independent of the clock.

All we have to do is change the sensitivity list to add `or posedge reset`:

```
always @(posedge clk or posedge reset)
```

Now the always block will be triggered when either the `clk` or `reset` signal goes from 0 to 1. We still need the `if` statement to see which one triggered the block.

In Figure 8.1, look at the signals on the bottom-half of the screen starting from `counter`; these are the signals from the module in this chapter.

You can see how the simulator holds the `reset` signal high for a number of clock cycles, then lowers it to 0. Note that the `counter` outputs are held to 0 while reset is high, and are then released.

In the simulator, both synchronous and asynchronous resets work pretty much the same way. But in actual hardware, there are subtle differences – asynchronous signals may lead to *metastability* unless the circuit is designed carefully. (Many FPGAs automatically reset their state upon power-up, so in some designs an explicit reset signal may not be neccessary.)

# 9

# Video Signal Generator

> "The game of Spacewar blossoms spontaneously wherever there is a graphics display connected to a computer."
>
> Alan Kay, computer scientist

The first commercial arcade game was born when Nolan Bushnell and Ted Dabney attempted to duplicate the PDP-1 minicomputer game *Spacewar!* using inexpensive custom hardware. Instead of using an expensive DEC Type 30 display, they used a GE Adventurer II black-and-white TV.

Throughout the 1970s, most arcade games would feature a *raster scan CRT*, either in black-and-white or color. They were more-or-less identical to the televisions of the era, except omitting the circuitry which decoded the over-the-air radio signal.

Think of a CRT as a device which converts a one-dimensional *video signal* to two dimensions. The *electron beam* draws successive *scanlines* from left-to-right, making pixels brighter or dimmer according to the amplitude of the video signal. A standard CRT has 262 scanlines from top to bottom.

The *sync signal* tells the CRT when to start and stop scanning. In our model, it's actually two signals. The *horizontal sync* tells the CRT when to start a new scanline. The *vertical sync* tells it when to start a new *field*, i.e. when to move the electron beam back to the upper-left corner.

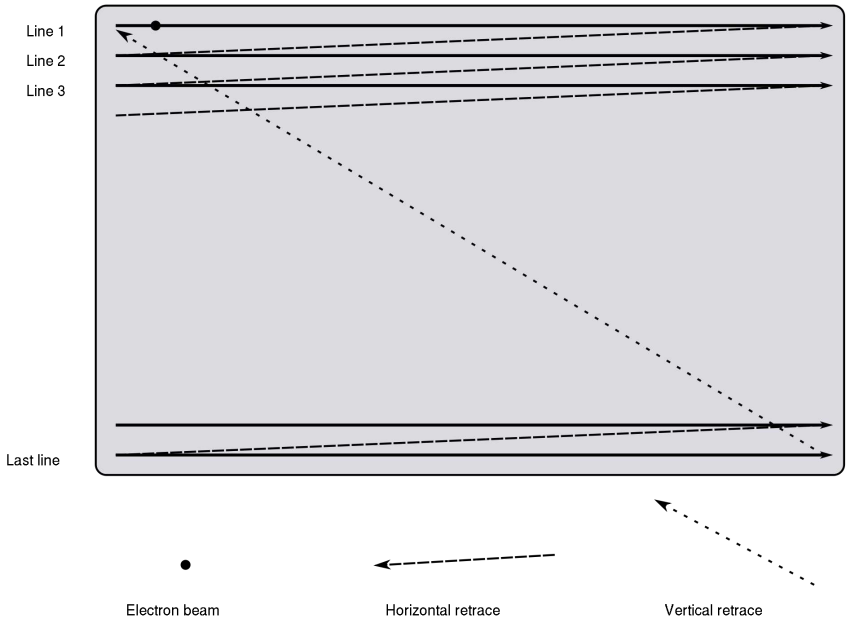



*Figure 9.1: CRT raster scanning*

To see it on `8bitworkshop.com`: Select the **Verilog** platform, then select the **Video Sync Generator** file.

The 8bitworkshop IDE has a simulated *CRT* to which we can feed signals in real-time.

The simulated CRT is designed to display an image of 256 x 240 pixels. Each pixel is a single clock cycle wide.

First, we'll write a Verilog module that generates the horizontal and vertical sync pulses at the correct frequency for our simulated CRT. Most of the examples in this book use this module, which we'll call `hvsync_generator`.

Our sync generator will count 262 scanlines and 309 clock cycles per scanline. At 60 frames per second, that's a clock speed of $309 * 262 * 60 = 4.857$ MHz.

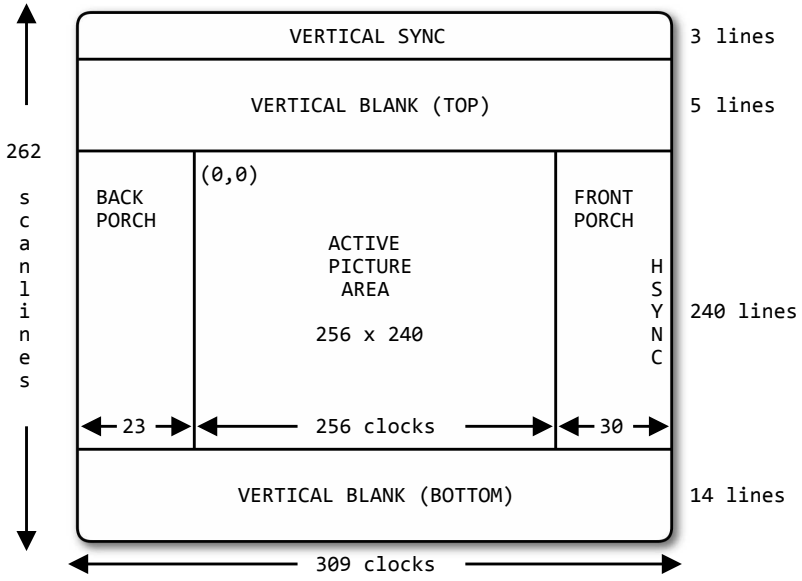



*Figure 9.2: Simulated CRT timing regions (not to scale)*

Let's write the module's declaration. We'll use a slightly different syntax, which is a matter of preference. In this format, we list the inputs and outputs in parenthesis, and define their types in a separate declaration:

```
module hvsync_generator(clk, reset, hsync, vsync, display_on,
    hpos, vpos);

  input clk;
  input reset;
  output hsync, vsync;
  output display_on;
  output [8:0] hpos;
  output [8:0] vpos;
```

This module has only two inputs: the clock signal (`clk`) and the reset signal (`reset`). The reset signal ensures that the module is initialized to a known state. It and the clock signal will be passed to most of our modules.

`hsync` and `vsync` are the horizontal and vertical sync outputs.

`display_on` is set whenever the current pixel is in a displayable area.

The `hpos` and `vpos` outputs are 9-bit counters that give the current horizontal and vertical position of the video signal. (0,0) is the upper-left corner. (255,255) is the lower-right corner of the *visible* frame. The *non-visible* portion extends to (308,261), which requires us to use 9-bit counters (8-bit counters only go up to 255.)

Our code would be clearer if we defined some symbols for the constant timing values. The `localparam` keyword can do this:

```
localparam H_DISPLAY = 256;  // horizontal display width
localparam H_BACK    =  23;  // left border (back porch)
localparam H_FRONT   =   7;  // right border (front porch)
localparam H_SYNC    =  23;  // horizontal sync width

localparam V_DISPLAY = 240;  // vertical display height
localparam V_TOP     =   4;  // vertical top border
localparam V_BOTTOM  =  14;  // vertical bottom border
localparam V_SYNC    =   4;  // vertical sync # lines
```

The `localparam` keyword can even do arithmetic, which we'll use to derive several values: the counter value where the sync signals start and end (`START_SYNC`, `END_SYNC`) and where the counter ends and wraps back to zero (`MAX`):

```
localparam H_SYNC_START  = H_DISPLAY + H_FRONT;
localparam H_SYNC_END    = H_DISPLAY + H_FRONT + H_SYNC - 1;
localparam H_MAX         = H_DISPLAY + H_BACK + H_FRONT +
  H_SYNC - 1;

localparam V_SYNC_START  = V_DISPLAY + V_BOTTOM;
localparam V_SYNC_END    = V_DISPLAY + V_BOTTOM + V_SYNC - 1;
localparam V_MAX         = V_DISPLAY + V_TOP + V_BOTTOM +
  V_SYNC - 1;
```

Our first `always` block handles the `hpos` counter:

```
// horizontal position counter
always @(posedge clk)
begin
  hsync <= (hpos>=H_SYNC_START && hpos<=H_SYNC_END);
```

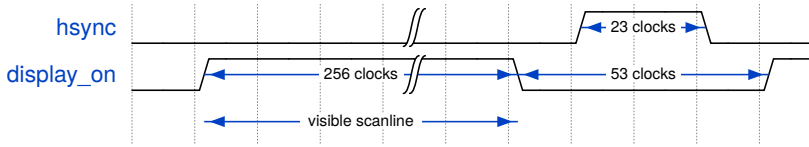



*Figure 9.3: HSYNC signal development for simulated CRT*

```
    if(hmaxxed)
      hpos <= 0;
    else
      hpos <= hpos + 1;
  end
```

This block does the following: On each clock cycle, increment `hpos` unless `hmaxxed` is asserted, in which case reset `hpos` to zero. It also sets `hsync` while the beam is near the right edge of the screen.

This `hmaxxed` wire performs the counter test. It is asserted if `hpos` is at its maximum value, or if the `reset` signal is high. There is a similar test for the vertical position:

```
  wire hmaxxed = (hpos == H_MAX) || reset;
  wire vmaxxed = (vpos == V_MAX) || reset;
```

Note: We're using logical operators here (`||` and `&&`) but for single-bit values they're functionally equivalent to the bitwise operators (`|` and `&`).

The vertical signal counter is handled in a separate `always` block:

```
  // vertical position counter
  always @(posedge clk)
  begin
    vsync <= (vpos>=V_SYNC_START && vpos<=V_SYNC_END);
    if(hmaxxed)
      if (vmaxxed)
        vpos <= 0;
      else
        vpos <= vpos + 1;
  end
```

Note that we only change `vpos` when `hpos` is at its maximum value or when we are resetting.

We also compute the `display_on` signal here:

```
  assign display_on = (hpos<H_DISPLAY) && (vpos<V_DISPLAY);
```

This signal defines the usable "safe" area of the CRT screen. Most of our example modules use this output to ensure they send the color black (0) outside of this area.

You'll notice that when loading this module into the 8bitworkshop IDE, it displays its outputs in Scope View. This module generates all of the signals we need to drive the simulated CRT, except for the `rgb` output. We'll add this signal in the next chapter, and make a simple test pattern.

## 9.1 Include Files

Note also that this module is wrapped by these lines which begin with a "`" character:

```
`ifndef HVSYNC_GENERATOR_H
`define HVSYNC_GENERATOR_H
...
`endif
```

These lines are *compiler directives*, which give instructions to the Verilog compiler as it compiles a program.

In the next chapter, we'll use the `` `include `` directive to include these modules in another file. Here, we use these directives to avoid an error if it is included twice.

The `` `define `` directive defines a *macro*, which is sort of like a parameter, but operates on a textual basis. The code between the `` `ifndef `` and `` `endif `` directives will not be compiled if the `HVSYNC_GENERATOR_H` macro is defined. So the first time the compiler encounters this file, the block will be evaluated, which defines the macro. On subsequent times, the compiler will skip the block.

# 10

# A Test Pattern

Now that we've built our video sync generator, let's use it to display a test pattern on the simulated CRT.

To see it on `8bitworkshop.com`: Select the **Verilog** platform, then select the **Test Pattern** file.

First, we *include* the Verilog file that contains the `hvsync_generator` module we just built:

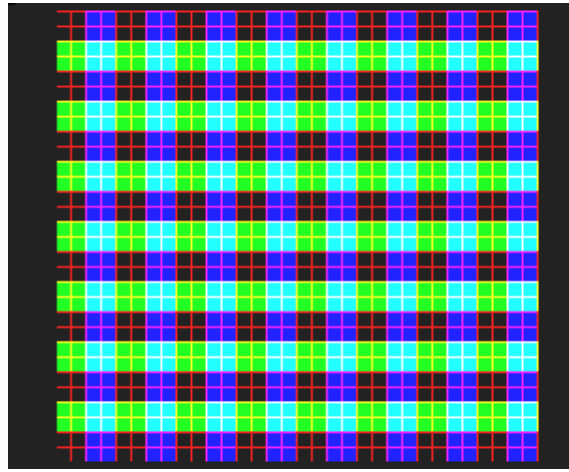


*Figure 10.1: Test pattern on simulated CRT*

```
`include "hvsync_generator.v"
```

Don't forget the "`" backtick at the beginning of the `include` directive.

Now, we declare our new test module. It has the suffix `_top` so that the IDE knows that it's the top module (and not the modules in our include files):

```
module test_hvsync_top(clk, reset, hsync, vsync, rgb);

  input clk, reset;
  output hsync, vsync;
  output [2:0] rgb;
  wire display_on;
  wire [8:0] hpos;
  wire [8:0] vpos;
```

As before, we have a `clk` signal as an input to this module, and also a `reset` signal.

The simulated CRT expects two outputs named `hsync` and `vsync`; these represent the horizontal and vertical sync signals. It also expects a 3-bit quantity named `rgb`, whose bits control the red, green, and blue signals for the simulated electron beam.

The next task is to instantiate a `hvsync_generator` module and wire it up:

```
  wire display_on;
  wire [8:0] hpos;
  wire [8:0] vpos;

  hvsync_generator hvsync_gen(
    .clk(clk),
    .reset(reset),
    .hsync(hsync),
    .vsync(vsync),
    .display_on(display_on),
    .hpos(hpos),
    .vpos(vpos)
  );
```

This creates a new instance of the `hvsync_generator` module (the instance is called `hvsync_gen`, but it could be any name) and wires up each input and output to our topmost module.

The ports are prefixed with a dot (`.`) and they have the same name as our topmost module variables. This makes it easy to remember which is which.

Next, we create a wire for each of the three color signals, and `assign` a value to each:

```
wire r = display_on && (((hpos&7)==0) || ((vpos&7)==0));
wire g = display_on && vpos[4];
wire b = display_on && hpos[4];
```

For the red color, we display a single-pixel grid by illuminating every 8th pixel. For blue, we alternate between on/off every 16 pixels, by grabbing bit 4 (zero-indexed) of the `hpos` variable. Green is the same as blue, but in the vertical direction.

All of these signals are AND'ed with `display_on` so that the beam is turned off outside of the normal visible area. When the signals are overlapped, this should create a nice plaid pattern on the CRT.

Now we use the *concatenation* operator to merge the red, green, and blue signals into a single 3-bit vector, which is `assign`ed to the `rgb` output:

```
assign rgb = {b,g,r};
```

In concatenation, the most-significant bit comes first, just like everywhere else in Verilog. So from left-to-right, the bits would be ordered `b` then `g` then `r`.

Figure 10.2 shows the flow of data from the video sync generator module to the RGB output. This is commonly called a *signal flow diagram*. You can see that this diagram is pretty complex, even considering that our module is fairly simple.

Figure 10.3 shows a *block diagram* of the test pattern generator. This conveys the flow of data at a higher level of abstraction, and is often more useful.

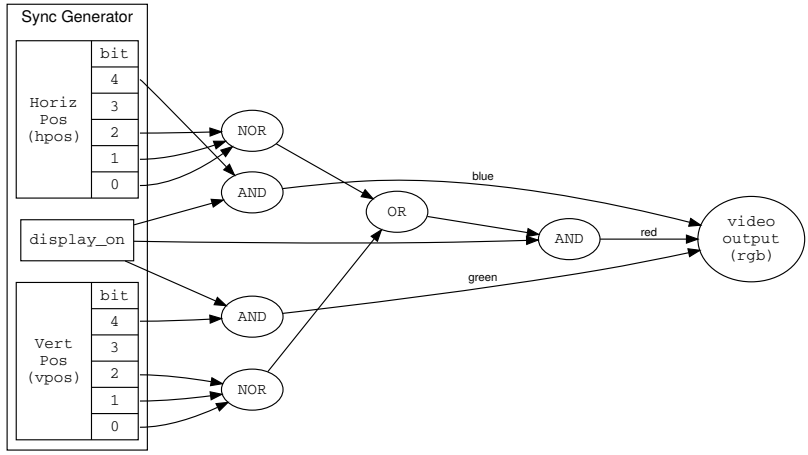



*Figure 10.2: Test pattern generator – signal flow*


*Figure 10.3: Test pattern generator – block diagram*

# 11

# Digits

Before video games came along, and pinball was king, there had to be a way to display the player's score. Mechanical score reels were common, but saw a lot of wear and tear. A few games used a *Nixie tube*, which stacked 10 numeral-shaped wires inside of neon-filled glass.

## 11.1 Seven-Segment Digits

To see it on `8bitworkshop.com`: Select the **Verilog** platform, then select the **7-Segment Decoder** file.

Starting in the 1970s, the *seven-segment display* became synonymous with high-tech, especially when LEDs and LCDs were

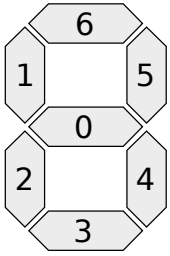



*Figure 11.1: A seven-segment digit, with bit index labels (CC BY-SA 3.0 by h2g2bob)*

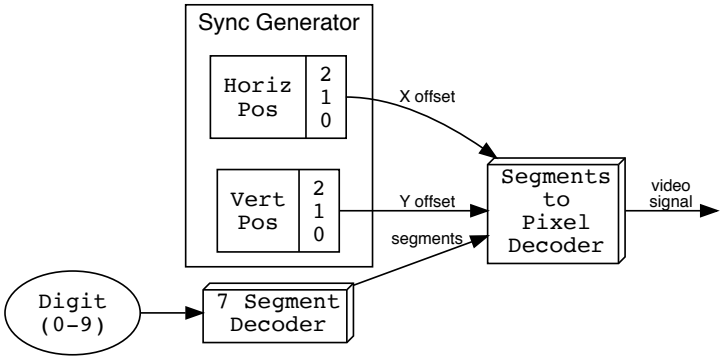



*Figure 11.2: Block diagram of 7-segment digits video display*

widely available. Every electronic device seemed to have one, any many still do. In fact, you may still see them in use on signs displaying gasoline prices.

The first video games tended to emulate the circuits of their pinball brethren, internally using the seven-segment representation even when displaying on a CRT. This was aided by the 7448 BCD to 7 Segment Display Decoder chip. This chip outputs a signal for each segment, which were then passed through a couple of *multiplexers* and gates to target the appropriate pixels.

Our Verilog version of this circuit contains a `seven_segment_decoder` module, which converts a 4-bit digit value into a 7-bit array, each bit representing a different segment. From there the `segments_to_bitmap` module takes the 7-bit array and the current scanline, turning that into a 5-bit array representing the bits to be output for that scanline.

We're not going to dwell too much on the seven-segment video generation method here, but it's described in detail in William Arkush's book cited in the bibliography.[2]

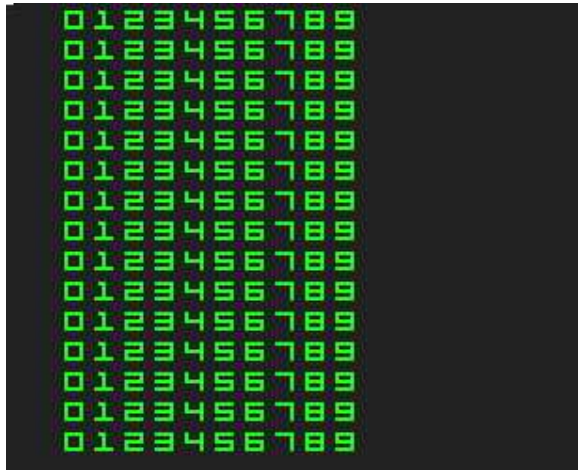



*Figure 11.3: Bitmapped digits on simulated CRT*

## 11.2 Bitmapped Digits and ROMs

To see it on `8bitworkshop.com`: Select the **Verilog** platform, then select the **Bitmapped Digits** file.

Using a *bitmap* font gives us the ability to display arbitrary shapes. The bit patterns of each character or digit is stored in a two-dimensional array, usually in a *ROM* (*read-only memory*). Typically, 1 is a lit pixel, and 0 is a dark pixel.

A ROM is a chip that stores an unchanging pattern of bits, essentially a lookup table. It's accessed by *address*, a number that determines which set of bits to fetch. For instance, if a ROM has a 10-bit address, it can serve $10^2 = 1024$ entries.

Most ROMs can serve multiple bits at a time, e.g. an ROM 8 bits wide stores 8 bits for each address.

Verilog doesn't have a way to precisely say "define a ROM here". Rather, the *synthesis tool* can infer a ROM from a couple of different constructs.

Here's the interface for our digits ROM module:

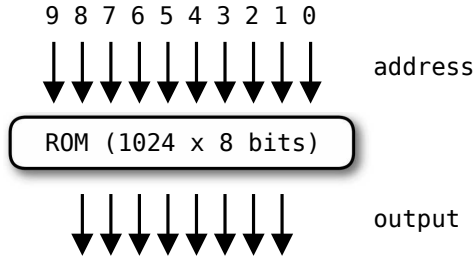



*Figure 11.4: Read-only memory*

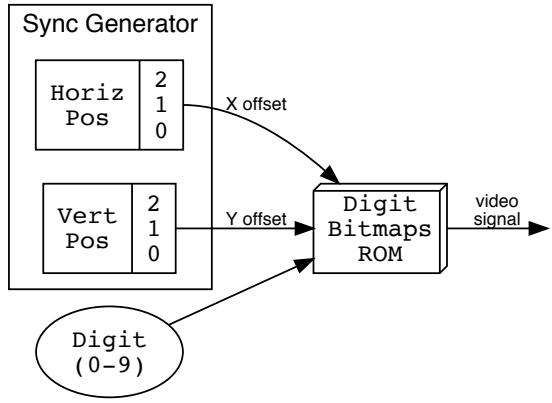



*Figure 11.5: Block diagram of bitmapped digits video display*

```
module digits10_case(digit, yofs, bits);

  input [3:0] digit;            // digit 0-9
  input [2:0] yofs;             // vertical offset (0-4)
  output reg [4:0] bits;        // output (5 bits)
```

The `bits` output contains the 5 pixels of the selected horizontal slice of the bitmap.

This module takes the following inputs:

- `digit` (4 bits) – A digit from 0 to 9.
- `yofs` (3 bits) – The vertical offset into the digit. Our digits are 5 pixels high, so this must be 0 to 4.

We combine these into a single 7-bit *address* which we use to look up the output:

```
wire [6:0] address = {digit,yofs};
```

Now we need to map each unique address to an output value. One way is to use a *case statement* to implement a lookup table:

```
always @(*)
  case (address)
    // digit "0"
    7'o00: bits = 5'b11111;   // *****
    7'o01: bits = 5'b10001;   // *   *
    7'o02: bits = 5'b10001;   // *   *
    7'o03: bits = 5'b10001;   // *   *
    7'o04: bits = 5'b11111;   // *****
    // digit "1"
    7'o10: bits = 5'b01100;   //  **
    7'o11: bits = 5'b00100;   //   *
    7'o12: bits = 5'b00100;   //   *
    7'o13: bits = 5'b00100;   //   *
    7'o14: bits = 5'b11111;   // *****
    // ... etc ...
  endcase
```

Note that we use *octal notation* – or base 8 – in the case statement. One of our selectors is 3 bits, so octal notation makes it a little more readable.

## 11.3 Initial Block

We can also write this module using the `initial` keyword to populate an array. Instead of a `case` statement, we use an `assign` statement to index the array and drive the `bits` output:

```
reg [4:0] bitarray[0:15][0:4]; // ROM array (16 x 5 x 5 bits)

assign bits = bitarray[digit][yofs]; // lookup ROM address

initial begin
  bitarray[0][0] = 5'b11111;  // *****
  bitarray[0][1] = 5'b10001;  // *   *
  // ... etc ...
```

Our array is two-dimensional, one dimension for the digit (16 entries) and one dimension for the Y offset (5 entries). So each

digit has 5 entries, each entry being a 5-bit value (indicated by `reg [4:0]`).

Note that we have 16 array slots, but only 10 digits. We can use a *for loop* to initialize the 6 leftover digits to zero:

```
for (int i = 10; i <= 15; i++)      // i = 10..15
  for (int j = 0; j <= 4; j++)      // j = 0..4
    bitarray[i][j] = 0;
```

Yes, Verilog has loops! But we can't use them in *synthesizable* code like we do in other programming languages. For example, we can use them to initialize arrays in an `initial` block. Other uses may not compile, or result in multiple copies of logic in our circuit.

## 11.4 External File

Verilog also lets you populate a read-only array from an external file, using the `$readmemb` or `$readmemh` directives. The former reads binary numbers from a text file (one value per line) while the latter reads hexadecimal numbers. For example:

```
initial begin
   $readmemb("digits5x5.txt", bitarray);
end
```

But since the 8bitworkshop IDE doesn't support reading external ROM files (as of this writing) we'll use one of the other methods described in our examples.

## 11.5 Test Module

To display the digits, we have a little test module which uses the `hvsync_generator` module and one of the two digits modules previously described. Here's how we hook up the digits ROM:

```
wire [3:0] digit = hpos[7:4]; // selected digit 0-9
wire [2:0] xofs = hpos[3:1];  // horiz. offset (2x size)
wire [2:0] yofs = vpos[3:1];  // vert. offset (2x size)
wire [4:0] bits;              // output bits from ROM
```

```
digits10_array numbers(       // ROM module
  .digit(digit),
  .yofs(yofs),
  .bits(bits)
);
```

The test module uses the `hpos` value from the sync generator – bits 7 to 4 select the digit, and bits 3 to 1 select the X offset (making the pixels double-size.) `vpos` selects the Y offset.

Now we just select the correct bit from the `bits` ROM output, and pipe it to the green bit of the `rgb` output:

```
wire r = 0;
wire g = display_on && bits[xofs ^ 3'b111];
wire b = 0;
assign rgb = {b,g,r};         // RGB output
```

We can now include our new digits modules in other Verilog programs with an `` `include `` directive. The test module will also be included, but ignored if not used. (Remember in the last chapter that we use the `_top` suffix to tell the IDE which module is topmost.)

# 12

# A Moving Ball

> "It might liven up the place to have a game that people could play, and which would convey the message that our scientific endeavors have relevance for society." – William A. Higinbotham, creator of *Tennis for Two*

One of the first electronic games with an animated display was *Tennis for Two*, developed at Brookhaven National Laboratory in 1958. The idea sprung from an example circuit in the instruction book for an analog computer, which demonstrated how to display a bouncing ball on an oscilloscope.

Later, the Magnavox Odyssey console would bring table tennis to the TV screen, and *Pong* would bring it to the arcades. Players would turn a knob to move an onscreen paddle, which deflected a moving square ball on-screen.

## 12.1 Absolute Position Method

To see it on `8bitworkshop.com`: Select the **Verilog** platform, then select the **Ball Motion (absolute position)** file.

In Chapter 9 we introduced the sync generator, which uses two counters to time the horizontal and vertical sync pulses, and

also represent the current position of the electron beam. How could we use these to put a moving ball on the screen?

An astute designer, especially if they had a software background, might say: "Let's have two registers that keep track of the horizontal and vertical position of the ball. Subtract these from the horizontal and vertical beam position, and display a ball whenever both of these are less than $N$ pixels apart. On each successive video frame, nudge the ball's position by a few pixels in some direction."

Let's try it that way. First we'll define registers for the ball's position. We make these 9 bits wide to match the video sync counters:

```
reg [8:0] ball_hpos;
reg [8:0] ball_vpos;
```

We'll start by drawing a 1-pixel ball. This is pretty easy – we just compare the ball's horizontal and vertical position to the video beam's position, and draw the ball when both of them are equal:

```
wire ball_hgfx = ball_hpos == hpos; // horizontal
wire ball_vgfx = ball_vpos == vpos; // vertical
wire ball_gfx = ball_hgfx && ball_vgfx; // AND operator
```

If you display the `ball_hgfx` signal on the CRT, it shows up as a horizontal line. The `ball_vgfx` signal displays as a vertical line. Where they intersect is the ball, drawn as a white dot – this is the combined `ball_gfx` signal.

If we wanted to make a bigger ball, we'd have to do a comparison rather than an equality operator. Subtract the ball's position from the video beam's position, and compare both values to the ball size (4 in this example):

```
wire [8:0] ball_hdiff = hpos - ball_hpos;
wire [8:0] ball_vdiff = vpos - ball_vpos;

wire ball_hgfx = ball_hdiff < BALL_SIZE;
wire ball_vgfx = ball_vdiff < BALL_SIZE;
wire ball_gfx = ball_hgfx && ball_vgfx;
```

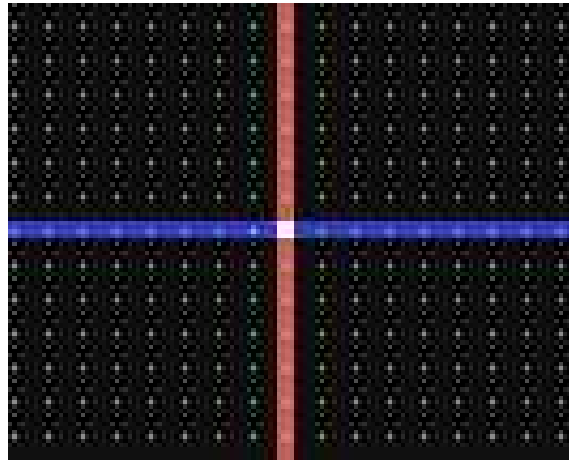


*Figure 12.1: Intersecting ball graphic signals*

Note that because Verilog defaults to *unsigned* numbers, the subtractions will overflow if they are negative. This is fine, because the overflowed value will always be greater than `BALL_SIZE` (unless we make the ball ludicrously large.)

We want to move the ball once per frame. For this we use an `always` block, triggered on the `vsync` signal edge, which rises at the end of each frame:

```
always @(posedge vsync or posedge reset)
begin
  if (reset) begin
    ball_hpos <= ball_horiz_initial;
    ball_vpos <= ball_vert_initial;
  end else begin
    ball_hpos <= ball_hpos + ball_horiz_move;
    ball_vpos <= ball_vpos + ball_vert_move;
  end
end
```

Note that this block implements an *asynchronous reset* (as described in Chapter 8) using the `if` statement to behave differently when reset than not-reset. Resetting sets the ball's position back to the center of the screen.

## 12.2 Bouncing Off Walls

We also would like the ball to bounce off the edges of the screen. First we need to determine if the ball has hit an edge of the screen. When we detect a collision, we reverse the ball's horizontal (left/right edge) or vertical (top/bottom edge) velocity – or both, if it hits a corner.

Let's write our edge-collision detector:

```
wire ball_vert_collide = ball_vpos <= 0 || ball_vpos >= 240;
wire ball_horiz_collide = ball_hpos <= 0 || ball_hpos >= 256;
```

We don't need the signed comparison, because as mentioned earlier we're using unsigned numbers, so we can simplify to just this:

```
wire ball_vert_collide = ball_vpos >= 240;
wire ball_horiz_collide = ball_hpos >= 256;
```

Let's not forget that the ball is a square, so we need to account for the ball's size on the right and bottom edges:

```
wire ball_vert_collide = ball_vpos >= 240 - BALL_SIZE;
wire ball_horiz_collide = ball_hpos >= 256 - BALL_SIZE;
```

Then we create a couple new events on the `_collide` signals which invert the ball's vertical or horizontal velocity:

```
// vertical bounce
always @(posedge ball_vert_collide)
  ball_vert_move <= -ball_vert_move;

// horizontal bounce
always @(posedge ball_horiz_collide)
  ball_horiz_move <= -ball_horiz_move;
```

Since we trigger on `posedge`, the events will only fire once when the signal goes high. As long as the signals pulse only once per frame (however long the pulse's duration) we'll only have one velocity reversal per frame.

# 13

# Slipping Counter

> Nolan comes by to me one day and he says, "When you adjust the vertical hold on a TV set, the picture moves back and forth like that." I said, "Yeah." He says, "Why is that?" So I explained to him why and he says, "Could we do that? Could we do something like that?" I said, "Yeah, but we'd have to do it digitally. You can't do it analog. You'd never have any control."
>
> Ted Dabney, Atari co-founder

## 13.1 Slipping Counter Method

To see it on `8bitworkshop.com`: Select the **Verilog** platform, then select the **Ball Motion (slipping counter)** file.

The absolute counter method described in Chapter 12 would be pretty familiar to anyone writing a game in software. But that's not what the first video games did in hardware. They did something a little more clever, called the *slipping counter* method. For historial completeness, we'll describe it a bit here. (You can skip this chapter if you're in a hurry.)

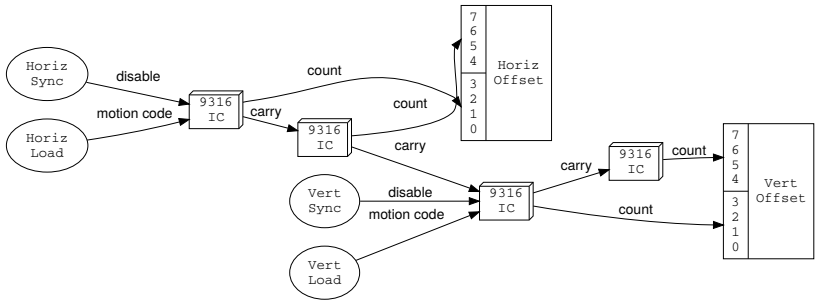



*Figure 13.1: Slip counter signal flow using 4-bit counter ICs*

Instead of two registers representing the X and Y coordinate of the ball, the *slipping counter* method keeps two registers which count in parallel with the horizontal and vertical sync counters. Think of it as keeping track of the difference between the ball's position and beam position, rather than the absolute ball position.

If the ball is stationary, the ball counters would be exactly in sync with the video sync counters, except offset by the position of the ball. However, if the ball is moving, they are allowed to "slip" by a certain amount, once per frame. The direction and magnitude of slip determines how fast and in which direction and how fast the ball moves.

For example, if the ball is moving right, the ball's horizontal counter would count an extra few cycles once per frame. If moving upward, the ball's vertical counter would count a few less cycles per frame.

Why did they do it this way? Because it takes a lot less hardware. This method can be implemented with four 9316 4-bit counter chips (two for horizontal, two for vertical), an additional flip-flop, and a handful of gates. This is similar to how the video sync generators would be implemented.

If we stored the absolute position of the ball, we'd have to use two 8-bit registers, two 8-bit subtractors, and logic for latching the position from the sync generators into the registers.

Here's how it works. There are two horizontal and vertical counters, just like the video sync counters:

```
reg [8:0] ball_htimer;
reg [8:0] ball_vtimer;
```

In the *Pong* circuit, these count upward to 511 (binary 111111111) then are reset to a known constant. When the ball is stationary, the horizontal counter should count exactly 309 pixels – the number of horizontal clocks in our simulated CRT. So the horizontal counter should start at 203, since 203 + 309 = 512.

Similarly, the vertical counter should start at 250, since there are 262 lines in the CRT, and 250 + 262 = 512.

The constants to be loaded are constructed by *concatenating* a 5-bit prefix, which is constant, and a 4-bit variable code. When the ball is stationary on either axis, a *stop code* is used. The stop code is chosen so that the counters start at 203 and 250, which as shown above make the ball stationary:

```
// 5-bit prefix codes
localparam ball_horiz_prefix = 5'b01100; // 192 + ?
localparam ball_vert_prefix = 5'b01111; // 240 + ?
// 4-bit stop codes
localparam ball_horiz_stop = 4'd11; // 192 + 11 = 203
localparam ball_vert_stop = 4'd10; // 240 + 10 = 250
```

When the ball is in motion, a *motion code* is used. These vary slightly from the stop codes, and cause the ball to move in different directions.

```
reg [3:0] ball_horiz_move;
reg [3:0] ball_vert_move;
```

The ball's vertical timer logic is fairly simple. We increment the timer at the end of each scanline, and when it hits the maximum value we reset it:

```
// update vertical timer
always @(posedge hsync or posedge ball_reset)
begin
```

```
  if (ball_reset || &ball_vtimer) // reset timer
    ball_vtimer <= {ball_vert_prefix, ball_vert_move};
  else
    ball_vtimer <= ball_vtimer + 1;
end
```

Note we use the AND reduction operator "&" to test if all of the bits are set. We could have also said `(ball_vtimer == 511)`.

The horizontal timer logic is a little more complex. We increment the timer on each clock cycle, and reset it when it hits the maximum value. But there's a special case, once per frame, when the vertical timer is also maximum value. We use this special case to load the motion code, which nudges the ball by the desired number of pixels:

```
// update horizontal timer
always @(posedge clk or posedge ball_reset)
begin
  if (ball_reset || &ball_htimer) begin
    if (ball_reset || &ball_vtimer) // nudge ball in horiz.
  dir
      ball_htimer <= {ball_horiz_prefix, ball_horiz_move};
    else // reset timer but don't move ball horizontally
      ball_htimer <= {ball_horiz_prefix, ball_horiz_stop};
  end else
    ball_htimer <= ball_htimer + 1;
end
```

Slipping counters were heavily used in early video games, but as you can see they're a little unintuitive. We'll use absolute coordinates in the rest of this book to make things a bit clearer.

# 14

# RAM

> "It is alleged in the Santa Clara [Silicon] Valley that the microprocessor was invented to sell programmable and read only memory chips."
>
> Steve Wozniak

Early arcade games did not have a lot of RAM, if any at all. Rather, the game state was kept in the various flip-flops, counters, and latches in the IC chips scattered around the circuit board. Without a CPU or video *frame buffer*, there was less need for RAM.

There were two main types of RAM ICs available in the 1970s, *static RAM* (*SRAM*) and *dynamic RAM* (*DRAM*).

Each bit in a static RAM chip is stored in a cell built from at least four flip-flops, while a dynamic RAM cell only require a single transistor and capacitor per bit. But dynamic RAMs are more complex to read and write, and memory cells must be refreshed periodically or their charge dissipates and bits are lost.

Games like *Breakout* used a 82S16 256-bit static RAM chip to store the on/off state of each brick on the playfield (actually two copies, one for each player.) This single part would have cost $5 to $10.

The first arcade game with a CPU, Midway's *Gun Fight*, contained a prodigious amount of dynamic RAM – 7 kilobytes, used

to display a 256 by 224 pixel frame buffer. Its 1975 production run used $3 million worth of RAM, estimated to be 60 percent of the world's supply.[3] The RAM chips alone would have cost around $400 per unit.

Most RAM chips of this era were 1 bit wide, i.e. each address stored a single bit. Since only one bit at a time could be accessed, they'd have separate pins for input and output. A game with a CPU like *Space Invaders* could organize eight 1-bit RAM chips in parallel to connect to the CPU's 8-bit data bus.[1] Later, RAM chips with 4-bit or "nibble-wide" outputs became available, reducing the number of chips needed.

## 14.1 RAM in Verilog

So far, our designs only have small *registers* that are hard-wired to specific functions in the circuit. For more complex designs, we'd like to have big blocks of *RAM* that we can read and write by address.

Verilog has no strict concept of RAM, but you can simulate RAM using indexed arrays. For example, here we declare a 1024-byte array named `mem`:

```
reg [7:0] mem [1024];  // 1024x8 bit array
```

We can read the value on the address bus (`addr`) like this:

```
dout <= mem[addr];  // read memory into dout
```

When the *write-enable* signal (`we`) is raised, we write to the array:

```
if (we)
  mem[addr] <= din;
```

We can package up these statements to make a complete RAM module. You can customize the size and word width using *parameters* – constants that can be overridden when the module is used.

[1] It used the Intel 2107C, a 4096-bit dynamic RAM.

Here's a simple RAM module in Verilog, which defaults to 1024 bytes:

```
module RAM_sync(clk, addr, din, dout, we);

  parameter A = 10; // # of address bits
  parameter D = 8;  // # of data bits

  input  clk;           // clock
  input  [A-1:0] addr;  // address
  input  [D-1:0] din;   // data input
  output [D-1:0] dout;  // data output
  input  we;            // write enable

  reg [D-1:0] mem [0:(1<<A)-1]; // (1<<A)xD bit memory

  always @(posedge clk) begin
    if (we)             // if write enabled
      mem[addr] <= din; // write memory from din
    dout <= mem[addr];  // read memory to dout
  end

endmodule
```

Note: The "<<" operator performs a *left shift*, which shifts the first argument by the number of bits in the second argument. Shifting left by one bit is equivalent to multiplying by two, so `(1<<A)` essentially computes $2^A$.

If you look at the `always` block, you may notice that when write is enabled, we appear to write and read the same memory address at the same time. Doesn't that create a problem? Does the read or write come first in such a situation?

The answer is: neither. Remember that `<=` is a *non-blocking assignment* and all such assignments in the same always block are executed simultaneously.

Because Verilog evaluates the *RHS* (right-hand side) of expressions before the *LHS* (left-hand side), the memory cell `mem[addr]` gets read before it is written, despite the ordering of statements in the code.

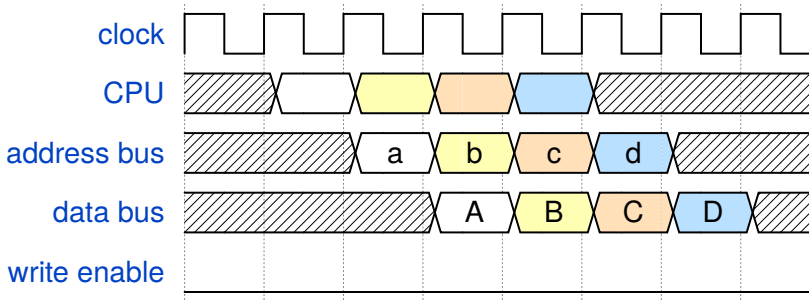



*Figure 14.1: Synchronous RAM module pipelining*

Because we have separate ports for reading and writing (`dout` and `din`) you can read the value from a memory location before you write to it, in the same clock cycle.

## 14.2 Synchronous vs Asynchronous RAM

The module we've built is *synchronous RAM*, because both reads and writes use non-blocking assignments in an `always` block.

Synchronous RAM introduces an extra delay cycle for reads, because the RAM module has to wait for the next rising clock edge. If the address bus contains a value on clock cycle $N$, the value is available on the data bus on cycle $N+1$.

In practice this often becomes a two clock cycle delay, if the module driving the address bus (e.g. CPU) sets this value in a synchronous block. You can *pipeline* requests, though (see Figure 14.1.)

An *asynchronous RAM* module makes the result available on the same clock cycle as the request. An `assign` statement performs the read asynchronously, like so:

```
always @(posedge clk) begin
  if (we)             // if write enabled
    mem[addr] <= din; // write memory from din
end

assign dout = mem[addr]; // read memory to dout (async)
```

DRAM chips until the mid-90s were usually asynchronous, and had specific timing requirements.The timing was often handled by a *memory controller*, which presents a synchronous interface to the world. Controllers also performed *DRAM refresh*, periodically re-reading each memory cell to prevent it from decaying.

In this book, we won't attempt to simulate asynchronous DRAM timing or DRAM refresh – we'll mostly use synchronous RAM (the `RAM_sync` module). Many FPGA toolchains will recognize this type of module and convert it to their own internal *block RAM*, which frees up resources for logic.

## 14.3 Tri-state buses and inout ports

We might want to connect our RAM chip to a common *bus*, shared with other devices. We'd like to have this bus function as input as well as output. We can do this using *tri-state logic*.

In addition to 0 and 1, we allow signals to take the value Z – the *high impedance* or *hi-Z* state, also known as *floating* or *tri-stated*.

When a signal is in hi-Z state, it has essentially "stepped out of the way", allowing another signal to drive the bus (literally!)

In our RAM module, we use it as a switch to control the flow on the `data` bus. When we're not writing (i.e. write-enable is not asserted) we keep open a signal path which sends a byte of data from memory to the `data` bus. But when we're writing, we put this signal path in hi-Z state, which cuts off the read path. This allows the `data` bus to be used as an input, driven from outside the module.

We give the `data` bus the `inout` keyword (which unfortunately looks confusingly similar to `input`, but it's a different keyword) to identify it as a tri-state bus.

```
inout  [D-1:0] data;  // data input/output
```

Our `assign` statement only slightly differs from the async RAM module. When the write-enable bit is set, we "float" the read path:

```
// if write is enabled, don't drive the data bus
// otherwise, read memory to data bus
assign data = we ? {D{1'bz}} : mem[addr];
```

The end result of the above line of Verilog is: "If the write-enable signal is active, step out of the way by the `data` bus in hi-Z state. Otherwise, look up the memory value at address `addr` and put that on the bus."

There are a couple of new language constructs in the above line. Let's go through them.

One thing we haven't yet seen is the *conditional operator*. It's sort of like an *if-else statement*, but more succinct. Its syntax is `(A ? B : C)`, and the meaning is "if A is true, the result is B, otherwise the result is C".

Also note the expression `{D{1'bz}}`, which is a *repetition operator*. It concatenates identical copies of the expression in the inner brackets, using the number in the outer brackets. For example, `{4{1'b1}}` makes 4 copies of a 1 bit – identical to `4'b1111`. Here, we're making `D` (the data width) copies of the Z value.

## 14.4 Writing to a tri-state bus

Since our read path "stepped out of the way" we can just write from the `data` bus to the `mem` array when the write enable signal is active:

```
always @(posedge clk) begin
  if (we)              // if write enabled
    mem[addr] <= data; // write memory from data bus
end
```

These are all *single-port RAM* modules, since they can read or write to only one address in a given clock cycle. *Dual-ported RAM* or *video RAM* allows reading and writing from different addresses simultaneously, but it was too expensive in the 1980s for video games.

# 15

# Tile Graphics

> "**Imagine** a dream machine ... **Imagine** raw video, plenty of it. **Imagine** autoscroll text, a full 16 lines of 64 characters."
>
> Steve Wozniak in a 1977 joke ad for ZALTAIR

Before computers had screens, they often sent their output to a teleprinter, which would punch holes or print characters on paper. As the price of integrated circuits and RAM got cheaper, these were replaced by CRT terminals that could display rows and columns of text on a monitor.

In 1970, Computer Terminal Corporation introduced the *Datapoint 2200* programmable terminal. Intended for mainframe users, it could be leased starting at $148 per month. However, its design would pave the way for personal computers and programmble video games. In fact, its CPU directly led to the development of the Intel x86 family.

As the price of RAM and CPUs continued to drop, video games would be designed along the same lines as CRT terminals, displaying rows and columns of uniform-sized cells. This scheme is now commonly known as *tile graphics*.

A tile graphics system would look up a value for a given cell in RAM, which determined what character to display in that row


*Figure 15.1: Tile graphics RAM coordinates and addresses*

and column. It would then lookup the tile data in a ROM, which contained the bitmaps (often 8x8) for each character.

We're going to make a simple RAM-based tile graphics generator, using the digits ROM we described in Chapter 11. It's similar to those examples, except the digit displayed in each cell will be read from a RAM module. (We'll still use a ROM to hold the digit bitmaps.)

To see it on `8bitworkshop.com`: Select the **Verilog** platform, then select the **RAM Text Display** file.

First, we hook up our 1024 x 8 bit RAM:

```
wire [9:0] ram_addr;  // address bus
wire [7:0] ram_read;  // data read
reg [7:0] ram_write;  // data write
reg ram_writeenable = 0; // write enable

// RAM to hold 32x32 array of bytes
RAM_sync ram(
```

```
  .clk(clk),
  .dout(ram_read),
  .din(ram_write),
  .addr(ram_addr),
  .we(ram_writeenable)
);
```

Then we have to decode the row and column of the current cell. Also the X and Y pixel within the current tile. We do this all with the `vpos` and `hpos` registers, pulling the pixel offset from the lower 3 bits (range 0-7) and the row and column index from the upper 5 bits (range 0-31):

```
wire [4:0] row = vpos[7:3];   // 5-bit row, vpos / 8
wire [4:0] col = hpos[7:3];   // 5-bit column, hpos / 8
wire [2:0] rom_yofs = vpos[2:0]; // scanline of cell
wire [4:0] rom_bits;             // 5 pixels per scanline
```

Now to get the final 10-bit address in RAM, we have to concatenate the row and column values together. The lower 5 bits will be the row, and next 5 bits the column.

```
assign ram_addr = {row,col};  // 10-bit RAM address

wire [3:0] digit = ram_read[3:0]; // read digit from RAM
```

Then we hook it up to the digits ROM where the bitmaps are stored. Then we output that to the screen, using the `xofs` value to figure out which bit to index:

```
wire [2:0] xofs = hpos[2:0];      // which pixel to draw
  (0-7)

// digits ROM module
digits10_case numbers(
  .digit(digit),
  .yofs(rom_yofs),
  .bits(rom_bits)
);

// extract bit from ROM output
wire digit_gfx = rom_bits[~xofs];     // ~ inverts xofs
```

Note that before we index the bitmap, we invert the bits in `xofs` (with the ~ operator). We do this because we've stored the bits in right-to-left order, but `xofs` counts from 0 upwards to 7. So this is just a slick way of computing `(7-x)`.

## 15.1 Animation

The above is fine for a static display, but we'd like to demonstrate some animation too. To do that, we'll have to write to RAM.

Our simple animation will increment every memory cell by 1 during each frame. This will make each cell cycle through all of the digits.

We'll perform the write only during the last scanline of each row, and only during the last two clock cycles of each character cell. Since our cells are 8x8 and our digits 5x5, we have plenty of blank space to ensure our writes don't conflict with reads.

We do the write in two clock cycles. In the first cycle, we set the `ram_write` address and `ram_writeenable` to 1. In the next cycle, we set `ram_writeenable` to 0.

```
// increment the current RAM cell
always @(posedge clk)
  case (hpos[2:0])
    // on 7th pixel of cell
    6: begin
      // increment RAM cell
      ram_write <= (ram_read + 1);
      // only enable write on last scanline of cell
      ram_writeenable <= (vpos[2:0] == 7);
    end
    // on 8th pixel of cell
    7: begin
      // disable write
      ram_writeenable <= 0;
    end
  endcase
```

This is a pretty simple animation scheme because we're writing to the same location currently being read – the update block doesn't touch `ram_addr`, in other words.

For a more complex animation (scrolling text for example) we'd need a more complex state machine, orchestrating reads and writes so that they happen quickly yet don't conflict. In Chapter 15 we'll have a much more complicated display scheme which uses memory bandwidth more efficiently, since we'll be sharing display RAM with a CPU.

# 16

# Switches and Paddles

## 16.1 Reading Switches

On/off switches are one of the primary input mechanisms for video games. Buttons, directional joysticks, and coin counters can all be implemented with switches.

One side of a mechanical switch is connected to +Vcc (positive voltage) and the other side connected to ground through a resistor. When the button is pressed, the switch is closed, allowing current to pass, and the logic level goes high. Otherwise, the switch is open, and current is shunted to ground, making the logic level low.

Because early video games were intended to make money, protecting the coin circuits was a priority. Players learned that they could get free games by zapping the coin case with static electricity. Video game designers responded by adding an antenna that detected unnatural levels of static and could respond by shutting off the game.

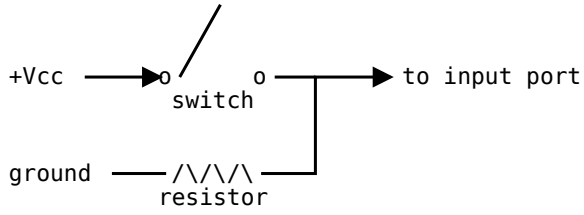



*Figure 16.1: Switch circuit.*

| Signal Name | Simulator Key |
|---|---|
| switches_p1[0] | Left Arrow |
| switches_p1[1] | Right Arrow |
| switches_p1[2] | Up Arrow |
| switches_p1[3] | Down Arrow |
| switches_p1[4] | Space Bar |
| switches_p1[5] | Shift Key |
| switches_p2[0] | A |
| switches_p2[1] | D |
| switches_p2[2] | W |
| switches_p2[3] | S |
| switches_p2[4] | Z |
| switches_p2[5] | X |
| hpaddle | (see below) |
| vpaddle | (see below) |

*Figure 16.2: List of 8bitworkshop simulator switches*

Luckily, our Verilog simulator is less complicated than the real world. Switches are simply sent as bit signals to the top-level module, in the `switches_p1` and `switches_p2` inputs. Figure 16.2 lists which keys in the 8bitworkshop IDE map to which bits.

## 16.2 Reading Paddles

The first commercial game console, Magnavox Odyssey, and the first successful arcade game, *PONG*, had *paddle controllers* as game inputs.

A paddle was essentially a knob which was connected to a *potentiometer*, or variable resistor. This moved the player's on-screen paddle, and so any spinnable knob in a video game came to be called a *paddle*.

Using TTL logic, paddle circuits were built with a *555 timer* IC and a resistor-capacitor network. The capacitor charges more quickly or more slowly depending on the position of the potentiometer.

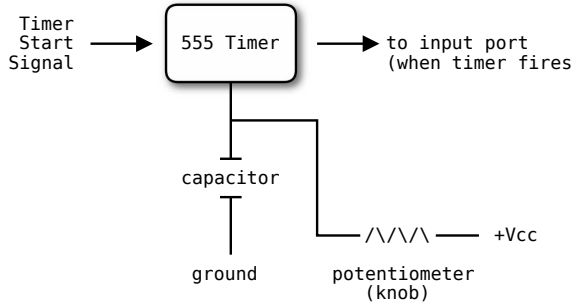



*Figure 16.3: Paddle timer circuit.*

As the capacitor charges, the 555 timer compares its voltage to a reference voltage, and triggers an output signal when they cross. Then the capacitor is drained for another read cycle.

Personal computers like the Apple ][ supported paddles using the same mechanism. Analog joysticks worked the same way, each axis essentially acting as a paddle.

In PONG-like games, including the Atari 2600 paddle games, the timer is started when the video generator starts to draw a new frame (i.e. at the top of the screen) and is tuned so that the maximum travel ends at the bottom scanline. When the 555 timer triggers, the player's paddle can start drawing on the current scanline, since the beam's vertical position is proportional to the knob's position.

To see it on `8bitworkshop.com`: Select the **Verilog** platform, then select the **Paddle Inputs** file.

In our simulator, the paddles are connected to the web browser pointer's horizontal and vertical position. They're calibrated so that the timer delay is proportional to the CRT vertical scan rate. Thus, the `vpaddle` output goes high when the simulated CRT reaches the pointer's vertical position. The same applies for the `hpaddle` input, if you substitute the pointer's horizontal position.

To measure the paddle's position in Verilog, we can just read the vertical video counter, and store it in a register. When the paddle output goes high, we store its current value:

```
always @(posedge hpaddle)
  paddle_x <= vpos[7:0];

always @(posedge vpaddle)
  paddle_y <= vpos[7:0];
```

This works fine in the simulator, but because of *metastability* we'd never do this in an actual circuit. The paddle inputs are *asynchronous* and should never be used as a clock, i.e. used as a `posedge` condition of an `always` block. They might be glitchy and violate the timing constraints of the flip-flops, causing cascading failures.

What we could do is clean up the signal by first running it through two cascading flip-flops. We'd also do the same for all of the joystick and button switches. This would ensure that the circuit sees a nice clean transition, synchronized with the clock.

# 17

# Sprites

> "We didn't do a square ball in 'Pong' because we thought it was cool. We did it because that was all we could do."
>
> Nolan Bushnell, Atari co-founder

Bouncing a ball is nice, but we'd like to draw some more complex shapes. We'll apply what we learned about drawing digits in Chapter 11 to draw *sprites*.

A sprite is more-or-less "a bitmapped shape which moves around the screen and overlays the background". The term was coined around 1979 by the creators of the TMS9918 chip, which powered graphics for the ColecoVision and MSX computers. Most games with hardware sprites from before that time used proprietary chips (e.g. Atari 2600) or discrete TTL logic (e.g. arcade games).

The first video arcade game *Computer Space* features two moving objects, each displaying a different spaceship. It uses the *slipping counter* method described in Chapter 13. The X and Y slip counters scan an 8 by 16 *diode matrix* (functionally similar to a ROM) which determines the pixels in the sprites.

The sprites are symmetrical, so only one side of the bitmap has to be stored in the diode matrix. The other half is displayed by scanning the matrix in reverse.

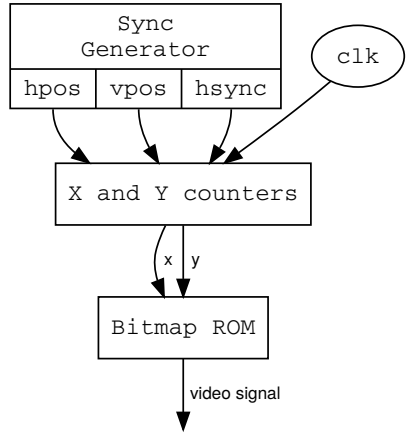



*Figure 17.1: Simple sprite renderer block diagram*

To see it on `8bitworkshop.com`: Select the **Verilog** platform, then select the **Sprite Bitmaps** file.

For our first example, we'll display a black-and-white sprite of a race car at a fixed position on the screen. Our sprite will be 16 pixels wide (8 pixels mirrored) and 16 pixels high.

First, we need the bitmap ROM, which we define similarly to the bitmapped digits in Chapter 11, in a module with an `initial` block:

```
module car_bitmap(
  input [3:0] yofs,
  output [7:0] bits
);
  reg [7:0] bitarray[16];       // 16 x 8 bit array
  assign bits = bitarray[yofs];
  initial begin/*{w:8,h:16}*/
    bitarray[0] = 8'b1100;
    bitarray[1] = 8'b11001100;
    bitarray[2] = 8'b11001100;
    // ... etc
    bitarray[15] = 8'b101110;
```

```
  end
endmodule
```

Similar to the digits ROM in Chapter 11, we pass in a Y offset (from 0 to 15) and return an 8-bit bitmap slice.

While we're drawing the sprite, we need two X and Y counters to scan through the pixels of the sprite's bitmap. They each have range 0-15, since our sprite is 16x16:

```
  reg [3:0] car_sprite_xofs;  // sprite X offset
  reg [3:0] car_sprite_yofs;  // sprite Y offset
```

We wire the ROM module's input to `car_sprite_yofs` and send the output to `car_sprite_bits`, just like the digits example:

```
  wire [7:0] car_sprite_bits; // selected line's output bits

  car_bitmap car(             // ROM module
    .yofs(car_sprite_yofs),
    .bits(car_sprite_bits));
```

We'll do the following each scanline:

- Is this scanline equal to the sprite Y position? If so, set the Y counter to 15.
- Otherwise, if the Y counter is not zero, decrement it.

We can clock on the `hsync` signal, which advances the scanline logic:

```
  // start Y counter when we hit the top border (player_y)
  always @(posedge hsync)
    if (vpos == player_y)
      car_sprite_yofs <= 15;
    else if (car_sprite_yofs != 0)
      car_sprite_yofs <= car_sprite_yofs - 1;
```

We'll do a similar thing with the X counter, but instead of triggering the event on `hsync` (once per scanline) we trigger on `clk` (once per horizontal pixel):

```
  // restart X counter when we hit the left border (player_x)
```

```
always @(posedge clk)
  if (hpos == player_x)
    car_sprite_xofs <= 15;
  else if (car_sprite_xofs != 0)
    car_sprite_xofs <= car_sprite_xofs - 1;
```

To compute the pixel to display, we should be able to index the sprite module output `car_sprite_bits` with the X counter `car_sprite_xofs`, like this:

```
wire car_gfx = car_sprite_bits[car_sprite_xofs];
```

But remember we only have 8 pixels in our bitmap, and we want to mirror these to make a total of 16 pixels. So, we do it like this:

```
// mirror sprite in X direction
wire [3:0] car_bit = car_sprite_xofs >= 8 ?
                              15 - car_sprite_xofs :
                              car_sprite_xofs;
// reduce 4-bit value to 3 bits and lookup bit in ROM
wire car_gfx = car_sprite_bits[car_bit[2:0]];
```

The expression `car_bit[2:0]` reduces the 4-bit value `car_bit` to 3 bits. This is safe because we can reason that it will always be 7 or less. Then, we use this value to look up the appropriate bit in the ROM byte we fetched.

You could also do a bit test and XOR, which is what the above code would reduce to when synthesized:

```
// mirror sprite in X direction
wire [3:0] car_bit = car_sprite_xofs[3] ?
                              car_sprite_xofs ^ 7 :
                              car_sprite_xofs;
// reduce 4-bit value to 3 bits and lookup bit in ROM
wire car_gfx = car_sprite_bits[car_bit[2:0]];
```

You can see why this works by iterating through the values – when you get to 8 (`4'b1000`) the conditional expression XOR's the first 3 bits, turning it into 15 (`4'b1111`) and then counting down from there.

We could easily move this sprite around the screen by modifying the `player_x` and `player_y` registers on every frame, perhaps with an `always` block on `(posedge vsync)`.

## 17.1 Improvements

- Modularize the sprite logic so we can reuse it and easily support multiple sprites by duplicating the module.
- This sprite renderer is hard-wired to a specific bitmap in ROM which is wired directly to the X and Y counters. Ideally, we'd want to read multiple bitmaps out of ROM, for instance to have multiple rotated versions of a sprite. We'll eventually create ROMs with multiple sprites and fetch data by *address*.
- To make the logic easier, we "cheat" by drawing a pixel even if the X or Y counter is zero, i.e. inactive. But if we have any lit pixels on the bottom or side borders, we'll get vertical streaks up and down the screen, as seen in Figure 17.2. A better design would have a state machine that explicitly starts and stops drawing. We'll do this in the next chapter.

We don't have a `reset` pin on this module, so all variables (flip-flops) will contain an undetermined value at power-up. This may cause glitches in the first frame of video. When we develop modules that interact with a CPU and RAM, we'll take care to reset them properly.

*Figure 17.2: Simple sprite renderer with streaking pixels, which occur because our state machine does not turn off the signal outside of the sprite area*

# 18

# Better Sprites

We're going to improve upon our simple sprite rendering circuit in Chapter 17.

To see it on `8bitworkshop.com`: Select the **Verilog** platform, then select the **Sprite Rendering** file.

The core of this module is a *finite state machine* (or *FSM*). In this design we explicitly define the various states our module can assume, the behavior of each state, and the rules for transitioning between states.

This design is clearer for humans to understand, and also many synthesis tools can detect FSMs and perform special optimizations.

A state machine in Verilog has three main components:

- State register – A register that holds the current state. Each state has a unique numeric code.
- Next state logic – The logic which determines the next state when the clock advances. This can be based on a condition, or not. The next state can also be the current state, for example if you're waiting for a condition.
- Output logic – The changes that occur on each clock cycle in each state.

In Verilog, we can use a *case statement* to implement a state machine, something like this:

```
always @(posedge clk)
  begin
    case (state)
      STATE_LABEL: begin
        register <= some_logic;
        out <= some_output;
        if (condition) state <= NEXT_STATE;
        if (other_condition) state <= YET_ANOTHER_STATE;
      end
      /// ... more case handlers
    endcase
  end
```

This is what we want our new sprite renderer to do:

- Wait until the scanline where our sprite starts, i.e. the top of the sprite.
- Activate the renderer.
- At the end of a scanline, load the next two bytes (our sprites are 16x16).
- Draw the 16-pixel slice on the next scanline.
- Repeat 16 times, then deactivate the renderer.

For brevity, we're going to describe the 8x16 version of the sprite renderer in this chapter. The 16x16 version is similar, but fetches two bytes from ROM for each scanline.

Each state is identified by a unique number. We'll use `localparam` to map these numbers to human-readable identifiers for our convenience:

```
localparam WAIT_FOR_VSTART = 0;
localparam WAIT_FOR_LOAD   = 1;
localparam LOAD1_SETUP     = 2;
localparam LOAD1_FETCH     = 3;
localparam WAIT_FOR_HSTART = 4;
localparam DRAW            = 5;
```

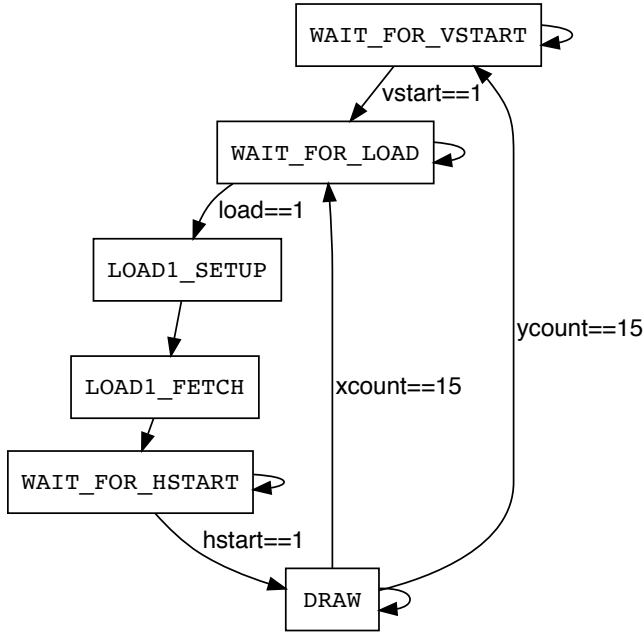



*Figure 18.1: Sprite renderer state machine diagram*

The states are defined as follows:

`WAIT_FOR_VSTART` – When the sprite renderer is inactive. Waits for the `vstart` input signal.

`WAIT_FOR_LOAD` – When active, but waiting for a scanline to start. Waits for the `load` input signal.

`LOAD1_SETUP` – Outputs the ROM address for this scanline's byte to `rom_addr`.

`LOAD1_FETCH` – Reads the ROM from `rom_bits`.

`WAIT_FOR_HSTART` – Wait for the `hstart` input signal to begin drawing a scanline.

`DRAW` – Output a pixel to `gfx`, then advance the X and Y counters. Repeats until done with the horizontal slice, then goes to `WAIT_FOR_LOAD`. Goes to `WAIT_FOR_VSTART` when done drawing the sprite.

Our first state `WAIT_FOR_VSTART` just initializes the `ycount` variable and waits for the `vstart` signal input to go high:

```
always @(posedge clk)
  begin
    case (state)
      WAIT_FOR_VSTART: begin
        ycount <= 0; // initialize vertical count
        gfx <= 0; // default pixel value (off)
        // wait for vstart, then next state
        if (vstart)
          state <= WAIT_FOR_LOAD;
      end
```

Note we also set `gfx <= 0` to make sure that the pixel output is turned off except when we're actively drawing the sprite.

When the `vstart` signal goes high, we are actively scanning the sprite. But we can't draw it until we load a byte from the ROM. The `WAIT_FOR_LOAD` state waits for the `load` signal to tell us that the ROM is free to use:

```
      WAIT_FOR_LOAD: begin
        xcount <= 0; // initialize horiz. count
        gfx <= 0;
        // wait for load, then next state
        if (load)
          state <= LOAD1_SETUP;
      end
```

The `load` signal might be sent in the non-visible portion of the scanline, since the ROM might be used for other things.

After the `load` signal goes high, we go to the `LOAD1_SETUP` state. It puts the desired ROM address on the address bus, based on the current scanline counter `ycount`:

```
      LOAD1_SETUP: begin
        rom_addr <= ycount; // load ROM address
        state <= LOAD1_FETCH;
      end
```

The next state `LOAD1_FETCH` grabs the sprite data off of the data bus and puts it into the `outbits` register:

```
    LOAD1_FETCH: begin
      outbits <= rom_bits; // latch bits from ROM
      state <= WAIT_FOR_HSTART;
    end
```

Now that we've loaded the sprite data for this scanline, we just need to wait until the beam gets to the sprite's horizontal position. The `hstart` signal tells us when this happens:

```
    WAIT_FOR_HSTART: begin
      // wait for hstart, then start drawing
      if (hstart)
        state <= DRAW;
    end
```

Now we are in the `DRAW` state. We decode pixels from the `outbits` register and increment counters:

```
    DRAW: begin
      // get pixel, mirroring graphics left/right
      gfx <= outbits[xcount<8 ? xcount[2:0] :
~xcount[2:0]];
      xcount <= xcount + 1;
      // finished drawing horizontal slice?
      if (xcount == 15) begin // pre-increment value
        ycount <= ycount + 1;
        // finished drawing sprite?
        if (ycount == 15) // pre-increment value
          state <= WAIT_FOR_VSTART; // done drawing sprite
        else
          state <= WAIT_FOR_LOAD; // done drawing this
scanline
      end
    end
```

Since our state register is 3 bits wide, it can hold 8 unique values, but we only have 5 valid states. Our module may start in an invalid state at power-up, since it has no `reset` pin. So we have a `default` case to handle unknown state, which just resets back to the initial state:

```
    // unknown state -- reset
    default: begin state <= WAIT_FOR_VSTART; end
```

## 18.1 Using the Sprite Renderer Module

First, we wire up the sprite ROM:

```
wire [3:0] car_sprite_addr;
wire [7:0] car_sprite_bits;

car_bitmap car(
  .yofs(car_sprite_addr),
  .bits(car_sprite_bits));
```

Now we define the `vstart` and `hstart` signals that define the upper-left corner of our sprite:

```
wire vstart = {1'b0,player_y} == vpos;
wire hstart = {1'b0,player_x} == hpos;
```

Note that we use *concatenation* to extend the player's 8-bit position to 9 bits, by tacking a zero bit (`1'b0`) onto the left side. This is to compare them with the video sync generator's 9-bit `hpos` and `vpos` outputs.

Then we wire up the `sprite_renderer` module to these signals:

```
wire car_gfx;         // sprite renderer output
wire in_progress;     // "in use" signal

sprite_renderer renderer(
  .clk(clk),
  .vstart(vstart),
  .load(hsync),
  .hstart(hstart),
  .rom_addr(car_sprite_addr),
  .rom_bits(car_sprite_bits),
  .gfx(car_gfx),
  .in_progress(in_progress));
```

Note that we hook up the `load` signal to `hsync` so that the sprite data gets loaded offscreen, during the horizontal sync pulse. Our sprite renderer might glitch if we load data while rendering to the screen. Later, we'll loosen this restriction by decoupling the video output logic from data loading logic.

# 19

# Racing Game

To see it on `8bitworkshop.com`: Select the **Verilog** platform, then select the **Racing Game** file.

To demonstrate multiple sprites and collision detection, we'll make a simple racing game using the sprite rendering module. We'll have two cars, one player-controlled and one computer-controlled. For the game we need two sprite rendering modules, but we'd like to have them share a single ROM module.

Verilog doesn't mind if there are multiple readers and writers of ROM/RAM on the same clock cycle, since as far as the language is concerned they're just array accesses. This does matter when you go to synthesize, since real circuits have limits on simultaneous access.

Our two sprite renderer modules share the same ROM bus and thus the same wires (`car_sprite_yofs` and `car_sprite_bits`). We have to trigger the memory load at different times so they don't conflict.

We allow the player sprite renderer 4 clock cycles to load its data from ROM. We do this right after the visible line ends, between horizontal position 256 and 259. We give the enemy sprite renderer ROM access at horizontal position 260 and beyond:

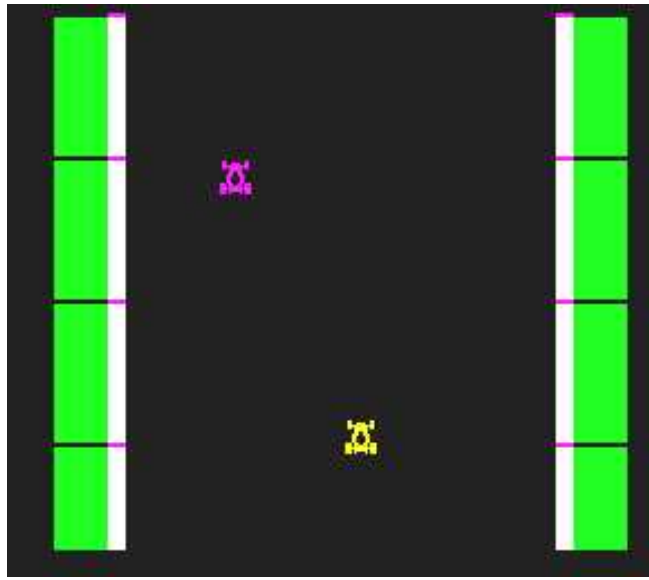


*Figure 19.1: Racing Game with two sprites*

```
wire player_load = (hpos >= 256) && (hpos < 260);
wire enemy_load = (hpos >= 260);
```

Using these two booleans, now we can *multiplex* the player and enemy ROM address (we actually only need the `player_load` boolean):

```
// wire up car sprite ROM
// multiplex between player and enemy ROM address
wire [3:0] player_sprite_yofs;
wire [3:0] enemy_sprite_yofs;
wire [3:0] car_sprite_yofs = player_load ?
  player_sprite_yofs : enemy_sprite_yofs;
```

Now we can wire up the ROM to the multiplexed `car_sprite_yofs` input, and its output to `car_sprite_bits`. We don't need to multiplex the ROM output, because it's ok to connect an output to multiple inputs. The sprite renderer modules will only read from the ROM when its their turn, i.e. when their `load` signal is asserted.

```
wire [7:0] car_sprite_bits;
```

```
car_bitmap car(
  .yofs(car_sprite_yofs),
  .bits(car_sprite_bits));
```

We also have to move the player and computer sprites, and make the track scroll down the screen. At the end of each frame (at vertical sync) we fire off an event that updates the positions of both sprites, and also updates the player's speed and track position (`trackpos`):

```
wire enemy_hit_left = (enemy_x == 64);
wire enemy_hit_right = (enemy_x == 192);
wire enemy_hit_edge = enemy_hit_left || enemy_hit_right;

always @(posedge vsync)
  begin
    // set player horizontal operation
    player_x <= paddle_x;
    player_y <= 180;
    // move track and enemy proportional to speed
    track_pos <= track_pos + {11'b0,speed[7:4]};
    enemy_y <= enemy_y + {3'b0, speed[7:4]};
    // move enemy back-and-forth
    if (enemy_hit_edge)
      enemy_dir <= !enemy_dir;
    // keep moving the enemy's car
    // unless they hit the edge during this frame
    if (enemy_dir ^ enemy_hit_edge)
      enemy_x <= enemy_x + 1;
    else
      enemy_x <= enemy_x - 1;
    // accelerate proportional to paddle position
    // or slow down when collision
    if (frame_collision)
      speed <= 16;
    else if (speed < ~paddle_y)
      speed <= speed + 1;
    else
      speed <= speed - 1;
  end
```

To detect collisions between the player and other objects, we run an event on each clock cycle. If the `player_gfx` signal and either `enemy_gfx` or `track_gfx` are on at the same time, we set the `frame_collision` flag. We clear it during each `vsync`:

```
reg frame_collision;

always @(posedge clk)
  // did player sprite intersect track or enemy?
  if (player_gfx && (enemy_gfx || track_gfx))
    frame_collision <= 1;     // yes, set collision bit
  else if (vsync)
    frame_collision <= 0;     // new frame, clear collision
  bit
```

To draw a track that appears to zoom past the player as their speed increases, we use `hpos`, `vpos`, and `track_pos`:

```
wire track_offside = (hpos[7:5]==0) || (hpos[7:5]==7);
wire track_shoulder = (hpos[7:3]==3) || (hpos[7:3]==28);
wire track_gfx = (vpos[5:1]!=track_pos[5:1]) &&
  track_offside;
```

This game is very simple, but it's a stepping stone to more complicated circuits. Note that a real arcade game would have a scoreboard or lap timer. It would also accept coins and have an *attract mode* that would run when no one is playing.

# 20

# Sprite Rotation

To see it on `8bitworkshop.com`: Select the **Verilog** platform, then select the **Sprite Rotation** file.

We'd like to make games with sprites that rotate. We'll do this by making several different versions of the sprite's bitmap at various rotations.

Our new sprite renderer is going to be able to *mirror* in the horizontal and vertical directions. With this feature, we'll only need $(N/4)+1$ bitmaps where $N$ is the number of total rotation steps.

For example, with 16 rotation steps, we need 5 sprites, from 0, 22.5, 45, 67.5, and 90 degrees.

Our new renderer will fetch two bytes during each scanline, instead of one. The total bitmap will be 16 x 16 pixels.

*Figure 20.1: 16x16 tank sprites, 5 rotations*

We also need to map each of the 16 rotation steps to one of the 5 bitmaps, and also to the values of the horizontal and vertical mirror flags. This could be easily done with a lookup table, but a simple module will also do the trick. We just need a `case` statement for each of the four quadrants of the rotation:

```
module rotation_selector(rotation, bitmap_num, hmirror,
    vmirror);

  input [3:0] rotation;    // angle (0..15)
  output [2:0] bitmap_num; // bitmap index (0..4)
  output hmirror, vmirror; // horiz & vert mirror bits

  always @(*)
    case (rotation[3:2])        // 4 quadrants
      0: begin                  // 0..3 -> 0..3
        bitmap_num = {1'b0, rotation[1:0]};
        hmirror = 0;
        vmirror = 0;
      end
      1: begin                  // 4..7 -> 4..1
        bitmap_num = -rotation[2:0];
        hmirror = 0;
        vmirror = 1;
      end
      2: begin                  // 8..11 -> 0..3
        bitmap_num = {1'b0, rotation[1:0]};
        hmirror = 1;
        vmirror = 1;
      end
      3: begin                  // 12..15 -> 4..1
        bitmap_num = -rotation[2:0];
        hmirror = 1;
        vmirror = 0;
      end
    endcase

endmodule
```

This looks like a complicated `case` statement, but when synthesized it reduces into a simple combinational function. We simply count up from 0 to 3 in the first and third quadrants of the circle, and count down from 4 to 1 in the second and fourth quadrants – see Figure 20.2.

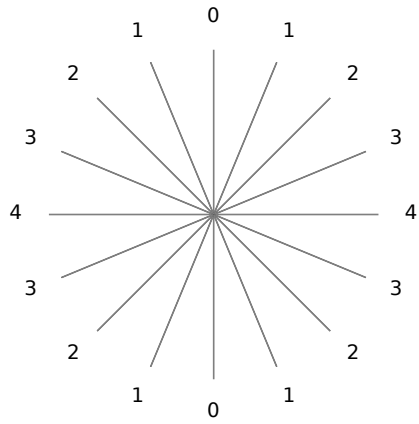



*Figure 20.2: Bitmap indices for 16 directions.*

# 21

# Motion Vectors

To see it on `8bitworkshop.com`: Select the **Verilog** platform, then select the **Sprite Rotation** file.

To demonstrate our rotating sprites, we're going to make a little tank that drives around the screen and bumps into stuff.

Most of this logic will be in the `tank_controller` module. This module handles rendering and movement for a single tank. It receives inputs from the video sync generator and joystick inputs, and outputs a video signal.

It also receives a `playfield` signal that represents the environment, and handles collisions between it and the player's tank.

## 21.1 Fixed-Point

The minimum amount we can move an object per-frame is one pixel. Our frame rate is 60 Hz, so the minimum speed we can move is 60 pixels per second.

This is pretty fast for a tank. If we want to move objects more slowly than this, we can use *fixed-point* numbers.

So far we've been dealing with whole binary numbers, assuming that the decimal place is missing or at the right side of the number, like this:

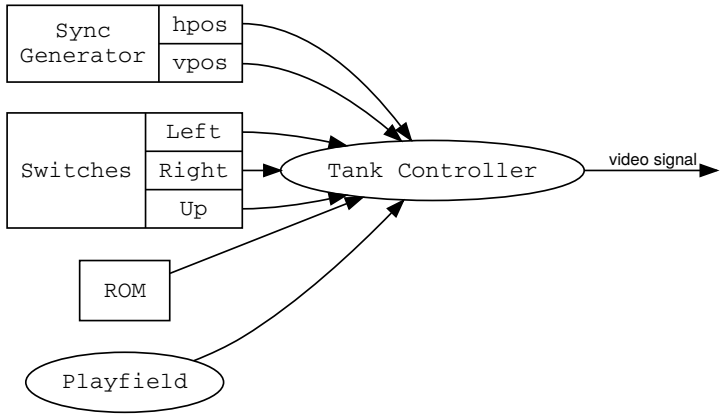



*Figure 21.1: Tank controller block diagram*

`100101100101.`

But let's say we move the decimal place four binary digits to the left:

`10010110.0101`

Now we have a *fixed-point* fractional number. Basically, we've turned our binary number into a fraction, dividing it by $2^4$ (16). The integer part is to the left of the decimal, and the fractional part is to the right.

You won't find a tiny little decimal point in the circuit design, but you can pretend it's there. All we have to do is grab our integer part and fractional part from different sides of the decimal point:

```
reg [11:0] player_x_fixed; // 8+4 bit fixed-point number
wire [7:0] player_x = player_x_fixed[11:4]; // integer part
wire [3:0] player_x_frac = player_x_fixed[3:0]; //
  fractional part
```

Both X and Y coordinates will be 12-bit fixed point numbers in our design – an 8-bit integer part, and a 4-bit fractional part. This will be enough precision to model our tanks' movement, as we'll explain in the next section.

## 21.2 The Sine Function

To determine the *vector* along which our tanks move, we'll have to use some basic trigonometry:

$$X' = X + v * sin(\theta)$$
$$Y' = Y + v * cos(\theta)$$

Where $(X, Y)$ is the current position, $(X', Y')$ is the new position, $v$ is the velocity, and $\theta$ is the angle of travel.

Verilog has no trig functions built-in, so we need to make a *sine function*. We don't need a lot of precision, however, since our tanks just need to point in one of 16 directions.

Our sine function needs to only accept 16 values, i.e. a 4-bit input. The output is four signed bits, with a range of -7 to 7. This function is similar to the `rotation_selector` module in the previous chapter, and has a `case` statement for each quadrant:

```
function signed [3:0] sin_16x4;
  input [3:0] in;     // input angle 0..15
  integer y;
  case (in[1:0])      // 4 values per quadrant
    0: y = 0;
    1: y = 3;
    2: y = 5;
    3: y = 6;
  endcase
  case (in[3:2])      // 4 quadrants
    0: sin_16x4 = 4'(y);
    1: sin_16x4 = 4'(7-y);
    2: sin_16x4 = 4'(-y);
    3: sin_16x4 = 4'(y-7);
  endcase
endfunction
```

We can use this function in multiple places in the Verilog code. (Be aware that the synthesis algorithm may instantiate a separate circuit each time this function is used.)

To get the cosine value, we can just advance the angle by 90 degrees – in our encoding, we just add 4 to the angle.

Note that the output of the sine function is 4 bits, the same width as the fractional part in our fixed-point coordinates. This ensures that we won't have to throw away precision when adding the sine and cosine values to our coordinates.

## 21.3 Move During the Hsync

In a game written in software, we'd want to multiply the sine and cosine of the angle by the speed of the vehicle to integrate it. But we don't have a multiplier circuit! (Actually, Verilog supports multiplication, but let's pretend it's the 1970s.)

So we're going to multiply the old fashioned way, by repeatedly adding. We'll update the player's position X times per frame, where X is the player's speed. This will happen during the `hsync`:

```
if (vpos < 9'(player_speed))
  if (vpos[0])
    player_x_fixed <= player_x_fixed +
    12'(sin_16x4(player_rot));
  else
    player_y_fixed <= player_y_fixed -
    12'(sin_16x4(player_rot+4));
```

What's with the `vpos[0]`? In a real circuit, we'd only want to have one instance of the `sin_16x4` function. So we ensure that only one is active at any given clock cycle, by multiplexing with `vpos[0]` and the other `if` statements.

## 21.4 Collisions

```
  // set if collision; cleared at vsync
  reg collision_detected;

  always @(posedge clk)
    if (vstart)
      collision_detected <= collision_gfx;
    else if (collision_gfx)
      collision_detected <= 1;
```

This `always` block handles collision detection. We set `collision_detected` whenever it happens in the frame, and we clear it when `vsync` comes around at the end of the frame.

So now we can detect a collision, but how do we respond to it? We're not going to actually prevent tanks from colliding with objects. Rather, we're going to handle the collision after the tank has already collided, by shoving the tank backwards a few pixels.

We'l first add 8 to the original angle, which adds 180 degrees (backwards). Then we move the tank repeatedly along this direction:

```
if (collision_detected && vpos[3:1] == 0) begin
  if (vpos[0])
    player_x_fixed <= player_x_fixed +
    12'(sin_16x4(player_rot+8));
  else
    player_y_fixed <= player_y_fixed -
    12'(sin_16x4(player_rot+12));
```

The `vpos[3:1] == 0` condition ensures that we move the tank 8 times per frame, or 8 times its normal speed, when there is a collision.

You'll notice there are some weird edge cases around collisions. For example, even when a tank is stationary it can contact a wall via rotation, then slip backwards through the wall.

To allow the player to control the tank, we use the `switch` inputs:

```
    frame <= frame + 1; // increment frame counter
    if (frame[0]) begin // only update every other frame
      if (switch_left)
        player_rot <= player_rot - 1; // turn left
      else if (switch_right)
        player_rot <= player_rot + 1; // turn right
      if (switch_up) begin
        if (player_speed != 15)       // max accel?
          player_speed <= player_speed + 1;
      end else
        player_speed <= 0;            // stop
    end
```

Our `frame` variable is a simple frame counter that helps us slow down the acceleration, by only reading inputs every other frame.

# 22

# Tank Game

To see it on `8bitworkshop.com`: Select the **Verilog** platform, then select the **Tank Game** file.

*Tank* was a 1974 two-player arcade game developed by Kee Games (which was actually a subsidiary of Atari secretly posing as a competitor, but I digress...)

*Tank* was designed by Steve Bristow, who also worked on Atari's *Computer Space*. It's almost the same game, except with two players, and a maze, and tanks instead of spaceships. It also used *interlaced video* to display twice the number of effective lines as previous games (better than our CRT simulator, in fact!)

*Tank* was one of the first games to use custom *ROM* chips, which could pack data much more efficiently than *diode arrays*. Its ROM is organized in four 512 byte segments, which store a playfield maze map, rotated tank sprites, digit bitmaps, and motion codes for tank movement. The ROM is *multiplexed* so that different segments of the ROM are read when the electron beam reaches various regions of the screen.

## 22.1 The Playfield Maze

The playfield maze is one of the simpler modules to implement. It's just scanning a 32x28 bitmap with the sync generator's horizontal and vertical position counters:

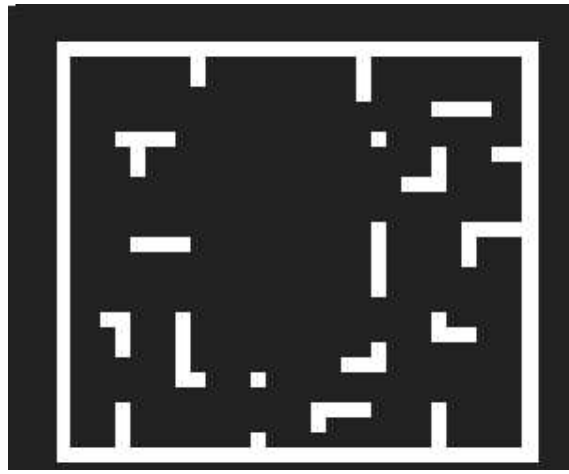


*Figure 22.1: Tank Game playfield maze*

```
module playfield(hpos, vpos, playfield_gfx);

  input [8:0] hpos;
  input [8:0] vpos;
  output playfield_gfx;

  reg [31:0] maze [0:27];

  wire [4:0] x = hpos[7:3];
  wire [4:0] y = vpos[7:3] - 2;

  assign playfield_gfx = maze[y][x];

  initial begin
    maze[0]  = 32'b11111111111111111111111111111111;
    maze[1]  = 32'b10000000000100000000001000000001;
    maze[2]  = 32'b10000000000100000000001000000001;
    // etc...
    maze[25] = 32'b10000010000000100000000000010001;
    maze[26] = 32'b10000010000000000010000000010001;
    maze[27] = 32'b11111111111111111111111111111111;
  end

endmodule
```

## 22.2 The Mine Field

The mine field is developed in several steps, each successively filtering the video signal from a uniform pattern into a sparse pattern of 16 mines.

First the `mine_pattern` signal draws a little X shape using some XOR gates:

```
// screen-wide pattern for mines
wire mine_pattern = ~(hpos[0] ^ hpos[1]) ^ (vpos[0] ^
  vpos[1]) && hpos[2] && vpos[2];
```

Then we limit the mine pattern to a specific rectangular window:

```
// limit mine pattern to a rectangular window
wire mine_field = (hpos >= 64 && hpos < 160)
  && (vpos >= 48 && vpos < 176)
  && mine_pattern;
```

There is only one mine per 8-pixel vertical row. We store each mine's horizontal position in the `mine_xpos` array:

```
// select only 1 mine per horizontal slice
wire mine_all = (hpos[6:3]==(mine_xpos[mine_vindex]^8))
              && mine_field;
```

When a player hits a mine, the mine explodes and we remove it from the screen. The `mine_exploded` bit vector tracks the existence of each mine, and we use it to filter our final `mine_gfx` output:

```
// only show mines that haven't exploded
wire mine_gfx = mine_all && !mine_exploded[mine_vindex];
```

## 22.3 The Players' Two Tanks

We've already described the `tank_controller` module in Chapter 18. Our main module creates two of these modules, one for each player, and wires them up thusly:

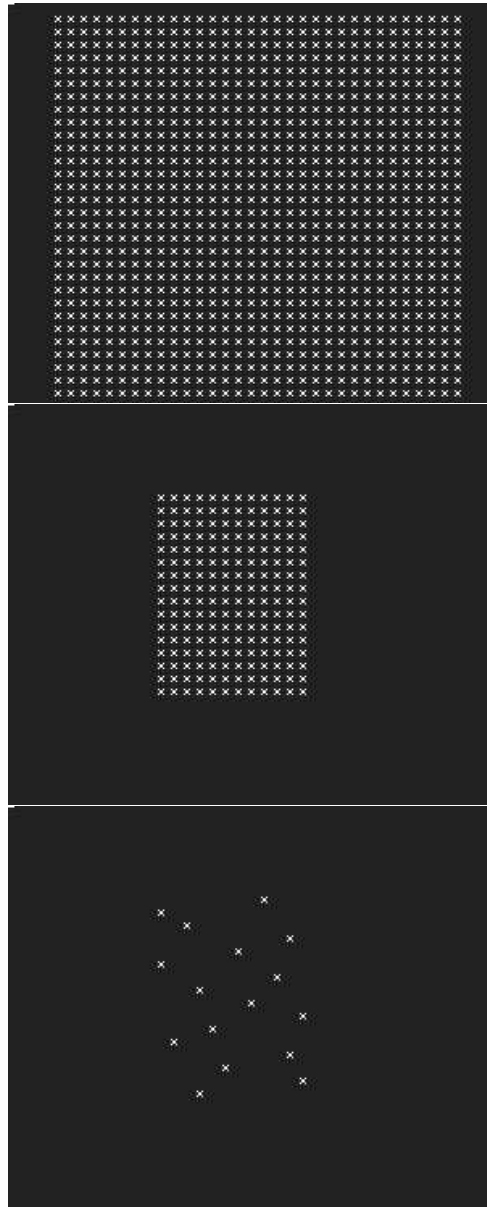


*Figure 22.2: Tank Game mine field development*

```
// sprite ROM inputs for each player
wire [7:0] tank1_sprite_addr;
wire [7:0] tank2_sprite_addr;
// multiplex sprite ROM output
wire [7:0] tank_sprite_bits;

tank_controller #(16,36,4) tank1(
  .clk(clk),
  .reset(reset),
  .hpos(hpos),
  .vpos(vpos),
  .hsync(hsync && !p2sel),
  .vsync(vsync),
  .sprite_addr(tank1_sprite_addr),
  .sprite_bits(tank_sprite_bits),
  .gfx(tank1_gfx),
  .playfield(playfield_gfx),
  .switch_left(switches_p1[0]),
  .switch_right(switches_p1[1]),
  .switch_up(switches_p1[2])
);
```

We have one problem: We have only one `tank_bitmap` module, but have to wire it to two different `tank_controller` modules.

The solution is to *multiplex* the connection, just like we did in Chapter 19. We effectively divide the `hsync` signal in two, and each controller can access the bitmap module during its half of the signal. This way, there's no contention.

```
// multiplex player 1 and 2 load times during hsync
wire p2sel = hpos > 280;

// bitmap ROM is shared between tank 1 and 2
tank_bitmap tank_bmp(
  .addr(p2sel ? tank2_sprite_addr : tank1_sprite_addr),
  .bits(tank_sprite_bits));

// module 1 sees (hsync && !p2sel)
// module 2 sees (hsync && p2sel)
```

This way, each module gets its own `hsync` signal and they're guaranteed not to overlap.

## 22.4 Improvements

Right now, we just have two tanks driving around a maze. If we wanted them to shoot at each other, we could add a module for each bullet, similar to the `tank_controller` module. We'd have to pass each module the other tank's `gfx` signal to detect a collision.

We could also make the tanks crash when they hit a mine. We'd need to use the `mine_gfx` signal, and also update the `mine_exploded` array when a collision is detected. A scoreboard would be nice, too.

It should be clear that designing games in pure logic is not scalable. To add an additional moving object, we need to add an entirely new circuit. For the remainder of the book, we'll shift focus to CPU-driven designs, and start developing games in software.

# 23

# Shift Registers

A *shift register* can be implemented with a cascade of flip-flops. Every time the clock ticks, each bit flows (shifts) to its neighbor. The bit that shifts out of the register can be read by another part of the circuit, and a new bit can be written into the vacant spot in the other end.

If you connect the output of a shift register to its input, you have a *ring counter*. For example, a 4-bit ring counter can generate a sequence of 4 different patterns before cycling. A *twisted ring counter* inverts its output before feeding back, and can count

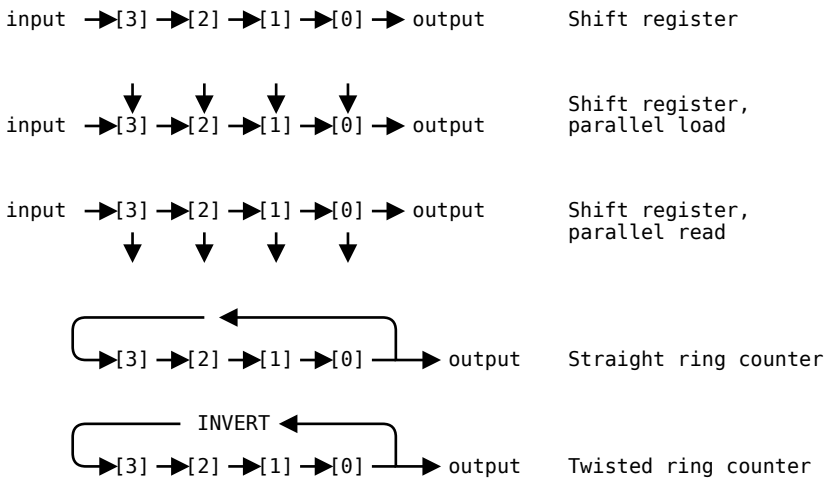



*Figure 23.1: Some simple 4-bit shift registers*

up to 8 different patterns. But more complex sequences an be generated by using multiple feedback bits.

## 23.1 Linear Feedback Shift Register

A LFSR generates numbers in a *pseudorandom* sequence. It can be designed with a specific *period* – the number of values generated before it repeats. They can operate as counters, or produce pseudorandom noise for audio and video signals.

While a binary counter has a period of $2^N$ (where $N$ is the number of bits) a LFSR has a maximal period of $2^N - 1$. So a maximal 8-bit counter will repeat a sequence of 255 unique values. The period depends on the choice of *taps* – the selection of bits to XOR and feed back.

Since a LFSR takes one-fourth the silicon area of a binary counter, they were often used in 8-bit designs. For example, the Atari 2600 uses them to count the horizontal position of objects. The catch is that the value of the counter cannot be set, only reset to its initial value. This makes programming the hardware a challenge, as one has to reset the counter at precisely the right time.

## 23.2 Types of LFSRs

There are two types of LFSR, *Fibonacci* and *Galois*. The main difference is how the taps are used and which bits are modified after the register is shifted.

A Fibonacci LFSR combines all the taps with XOR, generating a new bit to shift out. A Galois LFSR more-or-less operates in reverse – it modifies the taps, XOR-ing them with the bit that is shifted out.

To see it on `8bitworkshop.com`: Select the **Verilog** platform, then select the **Linear Feedback Shift Register** file.

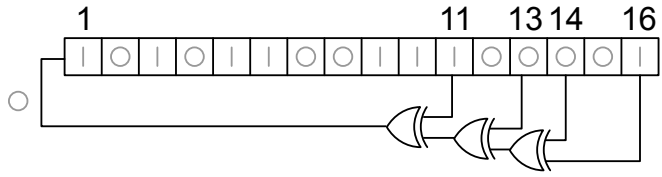



*Figure 23.2: 16-bit Fibonacci LFSR*

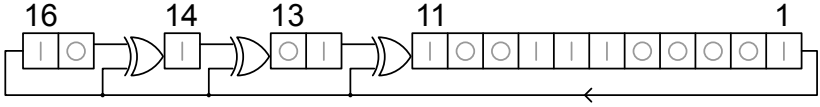



*Figure 23.3: 16-bit Galois LFSR*

Now we'll write a simple Galois LFSR module. We can add *parameters* to the module so that you can configure the size of the register, the taps, and whether to invert the feedback bit:

```
module LFSR(clk,reset,enable,lfsr);

  parameter TAPS   = 8'b11101;  // 8 bit shift register
  parameter INVERT = 0;         // invert feedback bit?
```

We can automatically derive the bit size of the register using the `$size` function. We make this a *localparam* because there's no need to override it:

```
  localparam NBITS  = $size(TAPS);
```

The implementation of the LFSR is pretty simple. Every clock cycle, we shift the register one bit to the left. We check the `feedback` bit – the leftmost bit we are shifting out of the register. If it and the `enable` bit are set, we XOR the register with the `TAPS` bit mask: Here's the bit of Verilog that does it:

```
    lfsr <= {lfsr[NBITS-2:0], 1'b0} ^ (feedback ? TAPS : 0);
```

And here it is in context with the rest of the module:

```
  input enable; // only perform shift when enable=1
  output reg [NBITS-1:0] lfsr; // shift register
```

```
wire feedback = lfsr[NBITS-1] ^ INVERT;

always @(posedge clk)
begin
  if (reset)
    lfsr <= {lfsr[NBITS-2:0], 1'b1}; // reset loads with 1s
  else if (enable)
    lfsr <= {lfsr[NBITS-2:0], 1'b0} ^ (feedback ? TAPS : 0);
end
```

Note that while `reset` is asserted, we just shift ones into the register. If the reset signal lasts long enough, all bits in the register will be set. For most LFSRs, this is a valid seed value.

## 23.3 Scrolling Starfield

To see it on `8bitworkshop.com`: Select the **Verilog** platform, then select the **Scrolling Starfield** file.

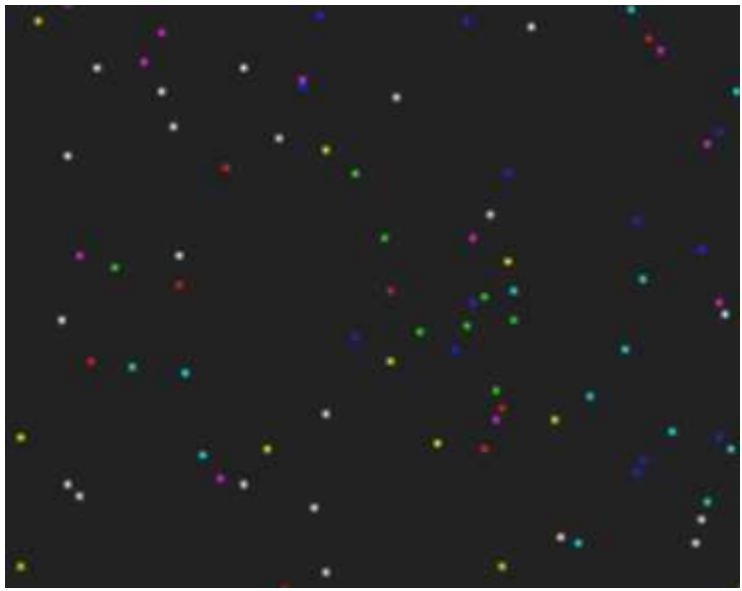


*Figure 23.4: Starfield with LFSR source*

A starfield is pretty easy to generate with a 16-bit LFSR, by selecting a subset of the random values (the `star_on` expression):

```
// enable LFSR only in 256x256 area
wire star_enable = !hpos[8] & !vpos[8];

// LFSR with period = 2^16-1 = 256*256-1
LFSR #(16'b1000000001011,0) lfsr_gen(
  .clk(clk),
  .reset(reset),
  .enable(star_enable),
  .lfsr(lfsr));

wire star_on = &lfsr[15:9]; // all 7 bits must be set
assign rgb = display_on && star_on ? lfsr[2:0] : 0;
```

The `star_enable` signal ensures that we only advance the LFSR during a 256 x 256 square region of the screen – 65,536 times per frame. Our LFSR's period is one less than that number – 65,535. Just like the slipping counter trick in Chapter 13, the starfield "slips" horizontally one pixel per frame, giving the appearance of motion.

# 24

# Sound Effects

> "Since I had the wire wrapped on the scope, I poked around the sync generator to find an appropriate frequency or a tone. So those sounds were done in half a day. They were the sounds that were already in the machine." [4]
>
> Al Alcorn, Pong designer

Video games have had sound since the first *Computer Space* debuted at a convention for jukeboxes. In early arcade game designs, the sound board was often custom-made for each title, and included a variety of sound circuits that could be triggered from the main CPU board. Since the sounds were hard-wired, there was usually little control except "start" and sometimes "stop."

Designers used analog circuits for sound generation throughout the 1970s. For example, the ubiquitous 555 timer chip could output a simple oscillating signal, sometimes modified by resistor-capacitor filters and/or envelopes.

Atari's *Pong* used this chip to generate three different sounds — bounce (a short 246 Hz pulse), score (a long 246 Hz pulse), and hit (a short 491 Hz pulse). [5] In *Tank*, the 555 timer was also used to generate rumbling motor sounds whose frequency varied with the speed of the player's onscreen vehicle.

For noisy sounds like explosions or gunfire, a random waveform is needed. For analog circuits, this could be produced using the avalanche breakdown properties of a reverse-biased zener diode. Transistors with "sharp knees" are preferred, referring to the shape of its frequency response graph.

## 24.1 Sound Chips

As the market for video games and electronic gadgets grew, chipmakers developed integrated circuits that produced various sorts of noises in a single chip. One of the first was the Texas Instruments SN76477 "Complex Sound Generator," which could be described as a hybrid digital-analog monophonic synthesizer.

The SN76477 was not programmable, but controlled by external capacitors and resistors, sometimes requiring other ICs for more complex noises. The data sheet describes example circuits such as "BIRD CHIRP." "STEAM TRAIN WITH WHISTLE," and "RACE CAR MOTOR/CRASH." In the post 8-bit era, this chip is beloved by DIY synthesizer tinkerers, and it had a production run as recently as 2008.

## 24.2 Sound Design

The simplest way for a digital device to generate sound is to output a square wave in the range from about 20 to 20,000 Hz. Luckily, we have a source of square waves – the *clock signal*!

Our Verilog simulator's master clock ticks every time a pixel is scanned – a rate of about 4,857,480 Hz when using the *sync generator* module. To get it into audible range, we need to use *clock divider*s.

To see it on `8bitworkshop.com`: Select the **Verilog** platform, then select the **Sound Generator** file.

We're going to try to make a digital version of the SN76477 sound chip. The basic data flow is shown in Figure 24.1. First, a few definitions:

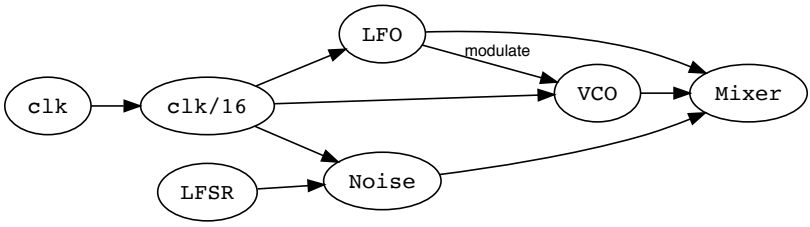



*Figure 24.1: Sound generator module data flow*

- clk - Master clock at 4,857,480 Hz.
- clk/16 - Master clock / 16 = 303,592 Hz.
- LFO - Low Frequency Oscillator. This is a triangle wave output at a relatively slow rate (1-1185 Hz).
- VCO - Voltage Controlled Oscillator. On the SN76477 this was an analog oscillator, but ours is digital. Its output frequency can be *modulated* by the LFO.
- Noise - Noise channel, courtesy of a 16-bit *LFSR* (see Chapter 23).

We have several inputs – 3 frequency values for the oscillators, two LFO modulation enable bits, a LFO depth value, and a 3-bit mixer select:

```
input clk, reset;              // clock and reset
input [9:0] lfo_freq;          // LFO frequency (10 bits)
input [11:0] noise_freq;       // noise frequency (12 bits)
input [11:0] vco_freq;         // VCO frequency (12 bits)
input vco_select;              // 1 = LFO modulates VCO
input noise_select;            // 1 = LFO modulates Noise
input [2:0] lfo_shift;         // LFO modulation depth
input [2:0] mixer;             // mix enable {LFO, Noise, VCO}
```

Our output waveform is just a single bit:

```
output reg spkr = 0;           // module output
```

The first thing we do is divide the master clock by 16. This reduces the resolution of our oscillators, but also improves performance and reduces the number of bits in our frequency registers:

```
always @(posedge clk) begin
  // divide clock by 16
  div16 <= div16 + 1;
  if (div16 == 0) begin
      // ... update waveform timers ...
  end
end
```

First we tackle the LFO. This is the simplest oscillator, since it is not modulated by any other oscillator. All we really have to do is repeatedly count downward from `lfo_freq * 256` and flip the `lfo_state` bit every time we reach zero:

```
    // LFO oscillator
    if (reset || lfo_count == 0) begin
      lfo_state <= ~lfo_state;
      lfo_count <= {lfo_freq, 8'b0}; // lfo_freq * 256
    end else
      lfo_count <= lfo_count - 1;
```

The LFO modulates the VCO, and to do that we need a smooth waveform. A triangle wave is easy to create. We just treat the LFO counter as a signed value, and invert it whenever the high bit is set:

```
// create triangle waveform from LFO
wire [11:0] lfo_triangle = lfo_count[17] ? ~lfo_count[17:6]
  : lfo_count[17:6];
wire [11:0] vco_delta = lfo_triangle >> lfo_shift;
```

The VCO is similar to the LFO, we just have to add `vco_delta` whenever modulation is enabled:

```
    if (reset || vco_count == 0) begin
      vco_state <= ~vco_state;
      if (vco_select)
        vco_count <= vco_freq + vco_delta;
      else
        vco_count <= vco_freq + 0;
    end else
      vco_count <= vco_count - 1;
```

| Signal | Base Hz | Max. Divisor |
|---|---|---|
| clk | 4857480 | |
| clk/16 | 303592 | |
| LFO | 1185 | 1023 |
| VCO | 303592 | 4095 + LFO |
| Noise | 303592 | 4095 + LFO |

*Figure 24.2: Sound generator oscillators*

The Noise oscillator is similar to the other two, but we only flip the waveform while the LFSR output is high:

```
reg [15:0] lfsr;              // LFSR output

LFSR #(16'b1000000001011,0) lfsr_gen(
  .clk(clk),
  .reset(reset),
  .enable(div16 == 0 && noise_count == 0),
  .lfsr(lfsr)
);

// ...in always block:
if (reset || noise_count == 0) begin
  if (lfsr[0])
    noise_state <= ~noise_state;
  if (noise_select)
    noise_count <= noise_freq + vco_delta;
  else
    noise_count <= noise_freq + 0;
end else
  noise_count <= noise_count - 1;
```

For the final output, we mix all three oscillator waveforms together, but only if their corresponding `mixer` bit is enabled:

```
    // Mixer
    spkr <= (lfo_state | ~mixer[2])
          & (noise_state | ~mixer[1])
          & (vco_state | ~mixer[0]);
```

# 25

# The ALU: Arithmetic Logic Unit

A computer's operation can be summed up thusly:

1. Grab a couple of numbers from somewhere.
2. Perform math on those numbers.
3. Move the resulting number somewhere else.
4. Decide what to do next.

The component that performs step 2 is called the *ALU* or *arithmetic logic unit*.

It's a *combinational circuit*, which if you remember back in Chapter 1 means that it doesn't contain any state, it just computes a Boolean function based on its current inputs.

A basic ALU takes two *operands* A and B, computes an operation (`aluop`) on them, and outputs the result Y.

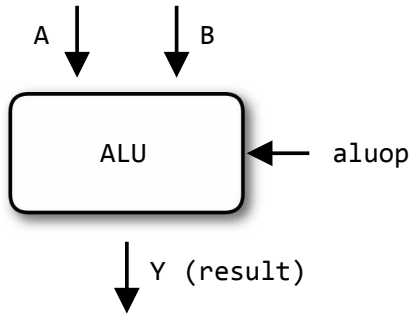



*Figure 25.1: Arithmetic Logic Unit (ALU)*

## 25.1 History

Early computers had a simple one-bit ALU. Minicomputers like the PDP-11 and VAX used the *74181 ALU* chip, a 32-function ALU in a single package. The 74181 can only process 4 bits at a time, but they can be chained together to support 8, 12, 16, or more bits.

Some early arcade games were designed with CPU-like functionality using the 74181. For example, Midway's *Wheels II* uses it to implement a complex counting and sequencing circuit driven by microcode. Cinematronics games use three of them to power a custom 12-bit CPU designed by Larry Rosenthal, a pioneer of vector video games.

The first microprocessors had primitive ALUs, with only a few operations like addition, AND/OR/XOR, and perhaps a bit shift. This forced CPU designers to be a little clever. For example, the 6502's ALU can only shift right, so it essentially performs a left shift by adding a value to itself, i.e. multiplying by two. The Z80 has a 4-bit ALU, but it makes multiple passes to build a 8-bit or 16-bit result.

## 25.2 Our ALU

Our ALU will only perform basic add/subtract, logical, and shift-by-one operations. (Modern ALUs can do things like multiply and perform arbitrary bit shifts, which early CPUs had to do one step at a time.)

We'll support 16 different operators, so our `aluop` input needs to be 4 bits wide.

We use the `parameter` keyword to define the various opcodes used for our ALU, assigning a numeric value to each opcode identifier. This allows us to use them outside from other modules.

We define the `ALU` module with just three inputs: the 8-bit A and B operands and the 4-bit operation code `aluop`. The 9-bit `Y` output holds the result of the operation.

| ALU op | Name | Expression |
|---|---|---|
| 0 | ZERO | 0 |
| 1 | LOAD_A | A |
| 2 | INC | A + 1 |
| 3 | DEC | A - 1 |
| 4 | ASL | A << 1 |
| 5 | LSR | A >> 1 |
| 6 | ROL | {A << 1, carry} |
| 7 | ROR | {carry, A >> 1} |
| 8 | OR | A \| B |
| 9 | AND | A & B |
| 10 | XOR | A ^ B |
| 11 | LOAD_B | B |
| 12 | ADD | A + B |
| 13 | SUB | A - B |
| 14 | ADC | A + B + carry |
| 15 | SBB | A - B + carry |

Why does `Y` have 9 bits instead of 8? Its leftmost bit holds the *carry bit* result, which is generated by the shift and arithmetic operations.

We define the `ALU` module with a parameter `N` which determines the bit width:

```
module ALU(A, B, carry, aluop, Y);

  parameter N = 8;      // default width = 8 bits
  input  [N-1:0] A;     // A input
  input  [N-1:0] B;     // B input
  input  carry;         // carry input
  input  [3:0] aluop;   // alu operation
  output [N:0] Y;       // Y output + carry
```

The ALU implementation is just a big `case` statement for each of our opcodes, assigning the result of each opcode to `Y`:

```
  always @(*)
    case (aluop)
```

First we have simple "load" operations which just copy the A or B inputs, or zero, to the Y output:

```
always @(*)
  case (aluop)
    // unary operations
    `OP_ZERO:         Y = 0;
    `OP_LOAD_A:       Y = {1'b0, A};
    `OP_LOAD_B:       Y = {1'b0, B};
```

Y is one bit wider than both A and B (remember, it contains a carry bit) and the Verilog compiler will complain if we just assign them to Y. So we use *concatenation* to add the carry bit, which we just set to zero.

The increment and decrement operations are straightforward:

```
    `OP_INC:          Y = A + 1;
    `OP_DEC:          Y = A - 1;
```

Left and right shifts and rotates are performed with *concatenation*, creating a 9-bit quantity from an 8-bit value. The *rotate* operations also use the carry flag as input.

```
    `OP_ASL:          Y = {A, 1'b0};
    `OP_LSR:          Y = {A[0], 1'b0, A[N-1:1]};
    `OP_ROL:          Y = {A, carry};
    `OP_ROR:          Y = {A[0], carry, A[N-1:1]};
```

The logical operations OR, AND, and XOR are pretty straightforward. The carry bit is always zero, so we again use concatenation:

```
    `OP_OR:           Y = {1'b0, A | B};
    `OP_AND:          Y = {1'b0, A & B};
    `OP_XOR:          Y = {1'b0, A ^ B};
```

Then we have addition and subtraction. ADC (add-with-carry) and SBB (subtract-with-borrow) both use the carry input, while ADD and SUB ignore it:

```
    `OP_ADD:          Y = A + B;
```

```
`OP_SUB:          Y = A - B;
`OP_ADC:          Y = A + B + (carry?1:0);
`OP_SBB:          Y = A - B - (carry?1:0);
```

All of these operations will also generate a carry bit in the output.

Note again that there are no registers inside of our module, nor a clock input; everything is implemented as combinational logic. The presence of `always @(*)` and the blocking assignment operator "=" (rather than the nonblocking "<=") indicates this.

It may seem like we're missing a couple of important operations – NOT, which inverts all the bits of an operand, and NEG, which converts a value to its *two's complement* negative. (The 6502 also lack these operations.) But a programmer can invert a value by XOR-ing with `$FF`, and negate a value by subtracting it from `0`.

Now that we have a functioning ALU, we're going to use it in the next chapter to start developing a simple CPU.

# 26

# A Simple CPU

> "I am terrible at video games, in fact, I struggle to make it past *Space Invaders*' first level."
>
> Tomohiro Nishikado, creator of *Space Invaders*

The arrival of the inexpensive *microprocessor*, also known as a *CPU* or *Central Processing Unit*, kick-started the video game and personal computer industries. The 4-bit microprocessors designed for handheld calculators gave way to 8-bit microprocessors designed to drive display terminals. As the prices of CPUs and RAM tumbled, designers pounced on the new technology.

The year 1975 saw the first video game to use a *microprocessor*. The video game manufacturer Taito originally designed *Western Gun* with discrete circuits. Dave Nutting and Jeff Frederiksen redesigned the U.S. version, *Gun Fight*, around the Intel 8080 CPU.[6] The original designer, Tomohiro Nishikado, was intrigued by the design and would later go on to create *Space Invaders* using a similar architecture.[7]

This was also the year that Steve Wozniak bought a discounted 6502 CPU at a consumer electronics show, the same CPU which powered his Apple I microcomputer design. The Atari 2600 design team would also discover the 6502 around this time.

## 26.1 What is a CPU?

Our Verilog programs can run pretty fast, because every event can potentially perform a computation on each clock cycle. But since more Verilog code generally means more circuitry, this can get wasteful quickly. We can *multiplex* some components to save hardware, for example sharing a ROM between two sprite generators. But for operations that don't need to run at the speed of the clock, we can use a CPU.

Most CPUs have a number of things in common:

- A number of *registers* which store internal state.
- An *instruction set* which the CPU interprets in a fetch-decode-execute cycle.
- The *ALU* or *arithmetic logic unit* which computes math and logic functions.
- An *address bus* which identifies which external components (e.g. RAM, ROM) and which address to read or write.
- A *data bus* which contains the data to send or receive from external components.
- An *instruction pointer* (or *program counter*) that contains the address of the next instruction to execute.

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). The CPU then figures out which instruction to grab next, and repeats the process.

The *instruction set* defines the various operations the CPU can perform. The most basic instruction takes two values (*operands*) from internal registers, configures the ALU to perform a simple calculation, and stores the result in a register. For example, the instruction "`ADD A,B`" adds the value in register B to register A, storing the result in A.

You can also operate on data in RAM. "`ADD A,[B]`" fetches the data at address B, then adds it to A.

We can even embed data inside the instruction itself. For example, "`ADD A,#127`" adds the constant value 127 to A. This is called *immediate mode*.

A CPU usually has *branch instructions* that modify the flow of the program if certain conditions are met. For example, `BZ` (Branch if Zero) changes the instruction pointer if the `Z` (zero) flag is set. These flags are set by previous instructions – for example, the `ADD` instruction sets the zero flag if the result of its addition is zero.

## 26.2 The FEMTO-8 Instruction Set

We're going to develop a simple 8-bit CPU which we'll call the *FEMTO-8*. It uses 8-bit data inputs, or *operands*. It also has an 8-bit *address bus*, limiting us to 256 bytes of directly addressable RAM + ROM.

A CPU can't read text very easily, so we use its preferred language: bits. The *instruction encoding* maps each combination of bits (in our case, 256 possible combinations) to a CPU operation. We call each of these values an *opcode*.

There are two competing goals in choosing an encoding. One is maximizing *code density* – the art of choosing the right mix of opcodes so that programmers can use fewer operations, reducing code size.

The other goal is simplifying *decoding*, or the process of converting the opcode to the proper control bits in the CPU. (Modern CPUs decode instructions into internal micro-operations, so the actual mix of bits in an opcode is less important.)

We describe the FEMTO-8 instruction set encoding in Figure 26.2. The first column describes the opcode – digits for static bits, letters for variable bits. The second column describes the operation performed. The third column gives an example of *assembly language* using this opcode.

The bit patterns for the instructions look arbitrary, but they are not. To reduce the complexity of decoding, some bits have consistent meaning across multiple opcodes:

| Opcode | Operation | Example |
|---|---|---|
| `00ddaaaa` | $A \leftarrow A \otimes B$ | `add A,B` |
| `01ddaaaa` | $A \leftarrow A \otimes imm8$ | `add A,#127` |
| `11ddaaaa` | $A \leftarrow A \otimes mem[B]$ | `add A,[B]` |
| `10000001` | $A \leftarrow B, B \leftarrow A$ | `swapab` |
| `1001nnnn` | $A \rightarrow mem[nnnn]$ | `sta 12` |
| `1010tttt` | conditional branch | `bz 123` |

| | |
|---|---|
| `dd` | destination (00=A, 01=B, 10=IP, 11=none) |
| `aaaa` | ALU operation ($\otimes$) |
| `nnnn` | 4-bit constant |
| `tttt` | flags test for conditional branch |

*Figure 26.1: FEMTO-8 instruction encoding*

- Bits 0-3 encode the ALU operation, a 4-bit constant, or the conditional branch flags.
- Bits 4-5 encode the destination register.
- Bit 6 is set if the ALU B input is read from the data bus, otherwise it reads from register B.

## 26.3 CPU Module

To see it on `8bitworkshop.com`: Select the **Verilog** platform, then select the **Simple 8-Bit CPU** file.

Here is our 8-bit CPU module. It has a very simple interface. The only inputs are clock, reset, and the 8-bit data bus input. The outputs are the 8-bit address bus, the write enable signal, and the data bus output (separate from the input.)

```
module CPU(clk, reset, address, data_in, data_out, write);

  input        clk;
  input        reset;
```

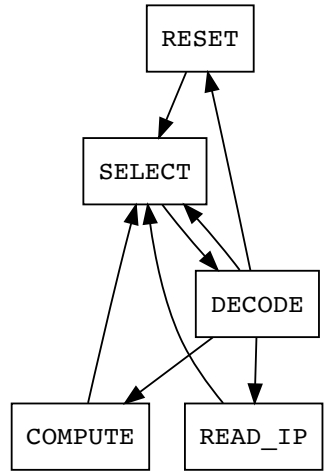



*Figure 26.2: A simple CPU state machine*

```
output [7:0] address;
input  [7:0] data_in;
output [7:0] data_out;
output       write;
```

## 26.4 State Machine

This state machine is a little more compact than the sprite renderer in Chapter 18, but each state's logic is more complex.

`SELECT` - Fetch the next opcode and increment the instruction pointer.

`DECODE` - Latch the opcode from the data bus and figure out what to do next: Write to memory, or proceed to the `COMPUTE` or `READ_IP` states.

`COMPUTE` - Latch the output of the ALU into a register; set the carry and zero flags.

`READ_IP` - Only used for taken conditional branches; reads a new IP from memory.

Our first state is `RESET`, where we set some default values for important registers:

```
S_RESET: begin
  IP <= 8'h80;  // instruction pointer = 128
  write <= 0;   // write disable
  state <= S_SELECT;    // next state
end
```

We initially set the *instruction pointer* (IP) register to `$80`, halfway into the memory space. Our assumption is that the ROM code will start there, leaving addresses 0x00-0x7f for RAM. We fall through to our next state `SELECT`, which is the beginning of our CPU loop.

First, we need to grab the *opcode* that lives at the address pointed to by the IP register. We start by putting the IP onto the address bus:

```
S_SELECT: begin
  address <= IP;
  IP <= IP + 1;
  write <= 0;
  state <= S_DECODE;
end
```

We also increment the IP to the next address at this time.

Our next step, `DECODE` is the most complex. The current opcode will be available on the data bus (`data_in`). We now have to pick its bits apart and figure out what to do next. We do this via a large `casez` statement:

```
// state 2: read/decode opcode
S_DECODE: begin
  opcode <= data_in;
  casez (data_in)
```

The `casez` statement is different from a normal `case` statement, in that you can mark individual bits as "don't care" values. This lets you match bit patterns.

For example, we decode a `COMPUTE` instruction if the two high bits are zero:

```
    8'b00??????: begin
```

```
        state <= S_COMPUTE;
      end
```

The COMPUTE state looks like this:

```
    S_COMPUTE: begin
      // transfer ALU output to destination
      case (opdest)
        DEST_A: A <= Y[7:0];
        DEST_B: B <= Y[7:0];
        DEST_IP: IP <= Y[7:0];
        DEST_NOP: ;
      endcase
      // set carry for certain operations (4-7,12-15)
      if (aluop[2]) carry <= Y[8];
      // set zero flag
      zero <= ~|Y[7:0];
      // repeat CPU loop
      state <= S_SELECT;
    end
```

The COMPUTE state grabs the output of the ALU (Y) and writes it to the desired destination, by decoding bits 4 and 5 of the opcode (opdest). It also computes the carry flag from the high bit of the ALU output, and the zero flag by NOR-ing all of the ALU output's bits (?). It then goes back to the SELECT state, and we do it all over again.

```
        // ALU A + B -> dest
        8'b00??????: begin
          state <= S_COMPUTE;
        end
```

What is not apparent here is that bit 6 has a specific meaning:

```
  wire B_or_data = opcode[6];
```

This bit selects the second (B) input of the ALU – either the B register, or the data bus input. You can see this where we wire up the ALU:

```
  ALU alu(
    .A(A),
```

```
.B(B_or_data ? data_in : B),
.Y(Y),
.aluop(aluop),
.carry(carry));
```

Note that in the opcode format here (`00??????`) bit 6 (second from left) is zero, so the ALU will read from the B register. In the following opcodes, we'll set the bit to 1, to read from memory instead.

We'd like to support *immediate* instructions, which is a two-byte instruction. We want to use the value of the byte immediately following the opcode.

Here we also go to the `COMPUTE` state, but we also put the IP's value on the address bus, and increment the IP. Since bit 6 is set, the ALU will read the result off the data bus. This effectively passes the next byte in the instruction stream to the ALU, giving us a two-byte instruction:

```
// ALU A + immediate -> dest
8'b01??????: begin
  address <= IP;
  IP <= IP + 1;
  state <= S_COMPUTE;
end
```

We can also read a byte from memory based on the value in the B register. This is even easier than immediate mode – we just put B on the address bus, and since bit 6 is set the ALU reads the result from the data bus:

```
// ALU A + read [B] -> dest
8'b11??????: begin
  address <= B;
  state <= S_COMPUTE;
end
```

Our 8-bit CPU only has a single write instruction, which is limited to the first 16 bytes of memory. We do not need to go to the `COMPUTE` state since the ALU is not used. Instead, we put the ALU operation (lower 4 bits of the opcode) on the address

bus, put the A register on the data bus, and set the write enable bit. Then we go immediately back to the SELECT state:

```
// A -> write [nnnn]
8'b1001????: begin
  address <= {4'b0, data_in[3:0]};
  data_out <= A;
  write <= 1;
  state <= S_SELECT;
end
```

In a limited instruction set like ours, it's handy to swap the A and B registers. This is a single opcode with no variable bits:

```
// swap A,B
8'b10000001: begin
  A <= B;
  B <= A;
  state <= S_SELECT;
end
```

## 26.5 Branch Instructions

A *conditional branch* modifies the IP based on the value of the *CPU flags*, potentially transferring control to a new instruction.

Our *branch condition* is kept in the lower 4 bits of the opcode:

| | |
|---|---|
| `0001` | branch if carry clear |
| `0011` | branch if carry set |
| `0100` | branch if zero clear |
| `1100` | branch if zero set |

We can combine the carry and zero conditions by OR-ing them, for example `0101` means "branch if carry is clear and zero is clear".

```
8'b1010????: begin
  // test CPU flags
  if (
    (data_in[0] && (data_in[1] == carry)) ||
    (data_in[2] && (data_in[3] == zero)))
  begin
```

```
      address <= IP;
      state <= S_READ_IP;     // branch taken
    end else begin
      state <= S_SELECT;      // branch not taken
    end
    IP <= IP + 1; // skip immediate value
  end
```

If the branch is taken, the CPU moves to the `READ_IP` state, which reads the data bus and puts the result into the IP register (we've already incremented IP in the previous state):

```
    // state 4: read new IP from memory (immediate mode)
    S_READ_IP: begin
      IP <= data_in;
      state <= S_SELECT;
    end
```

## 26.6 Using the CPU Module

To use the CPU module, we need to connect its inputs and outputs. When the CPU wants to read or write data, it places an address value on its *address bus*. We connect various components to this bus that respond to certain ranges of addresses.

This `always` block connects the address bus to 128 bytes of RAM and 128 bytes of ROM. We use the high bit to distinguish the two. Therefore, RAM is accessed at addresses `$00-$7F` and ROM is addressed at `$80-$FF`:

```
// read from CPU
always @(*)
  if (address_bus[7] == 0)            // high bit set?
    to_cpu = ram[address_bus[6:0]];   // read RAM
  else
    to_cpu = rom[address_bus[6:0]];   // read ROM
```

Note that the `always` block supporting reads is combinational, not clocked. This gives us an `asynchronous` read, where the result is available to the CPU in the same clock cycle as the request.

The CPU asserts the `write_enable` signal when it wants to write to RAM. We handle it in this `always` block:

```
  always @(posedge clk)
    if (write_enable) begin
      ram[address_bus[6:0]] <= from_cpu;
    end
```

Since our RAM is only 128 bytes, we only need to use 7 (of 8) bits of the address bus. We don't check to see if the RAM write is out of range. So if you write to address 128, you'll actually write to RAM location 0.

Now we can create our CPU module, hooking up the clock, reset signal, address bus, data bus and write enable flag:

```
  CPU cpu(.clk(clk),
          .reset(reset),
          .address(address_bus),
          .data_in(to_cpu),
          .data_out(from_cpu),
          .write(write_enable));
```

We'll test our CPU with a tiny little program that computes the *Fibonacci sequence*:

```
Start:
        zero    A       ; A <= 0
        ldb     #1      ; B <= 1
Loop:
        add     A,B     ; A <= A + B
        swapab          ; swap A,B
        bcc     Loop    ; repeat until carry set
        reset           ; end of loop; reset CPU
```

In the next two chapters, we'll describe the configurable assembler that turns this into machine code for our CPU, and convert the racing game in Chapter 19 to a CPU-driven architecture.

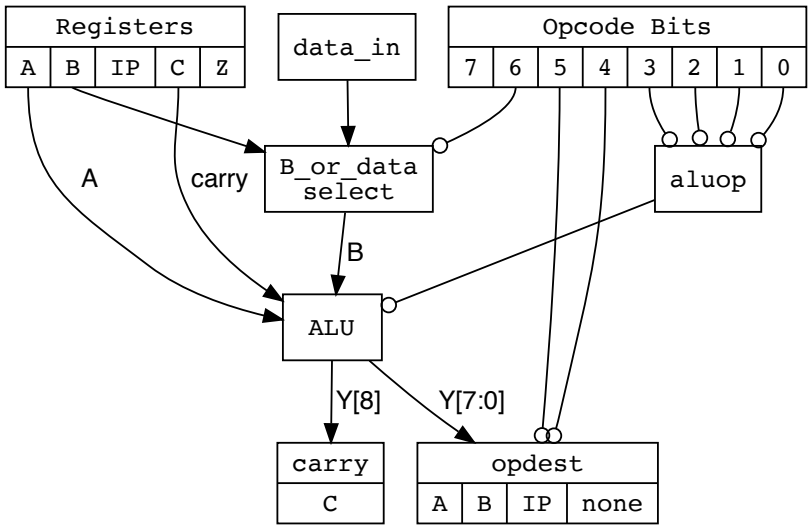



*Figure 26.3: COMPUTE instructions decoding through ALU. Note that the B input of the ALU is multiplexed between the B register and the data bus.*

# 27

# A Configurable Assembler

> "Programming in machine code is like eating with a toothpick."
>
> Charles Petzold, programmer and author

Now that we've designed our CPU and its instruction set, how do we make programs for it?

We could program it directly using machine code:

```
00 5B 01 0C 81 A1 83 8F
```

That seems pretty tricky. Another approach might be to use Verilog's *macros* – we could write macros that translate into CPU opcodes, then pack those into an array, like so:

```
initial begin
  rom = '{
   `I_CLEAR_CARRY,
   `I_ZERO_A,
   `I_CONST_IMM_B,
   1,
   `I_COMPUTE(`DEST_A, `OP_ADD),
   `I_SWAP_AB,
   `I_BRANCH_IF_CARRY(1'b0),
   4 + 'h80, // correct for ROM offset
   `I_STORE_A(4'd0),
   `I_RESET,
  }
end
```

That's a bit cumbersome. We could do better.

What we need is a custom *assembler* for our custom CPU. This is a tool that converts a human-readable language into machine instructions that the CPU recognizes. Then we can write something like this:

```
        clc             ; clear carry
        zero    a       ; A = 0
        ldb     #1      ; B = 1
Loop:   add     a,b     ; A = A + B
        swapab          ; swap A & B
        bcc     Loop    ; loop if carry bit clear
        sta     0       ; write A to address 0
        reset           ; reset
```

We're also going to make our assembler configurable, so we can easily target a given CPU.

## 27.1 JSASM Configuration Format

Verilog doesn't have a built-in assembler, but the 8bitworkshop IDE does. It can translate custom assembly language for Verilog CPUs. The CPU's language is defined in a a JSON configuration file.

To see it on `8bitworkshop.com`: Select the **Verilog** platform, then select the **Simple 8-Bit CPU** file. Click the vertical bar to the left of the editor, then drag it to the right to expose the sidebar. Then click on the **femto8.json** file to open it.

Our assembler's configuration format has a number of **rules**. Each rule has a format that matches a line of assembly code, and a bit pattern that is emitted when matched. For example, this is the rule for the `swapab` instruction:

```
{"fmt":"swapab", "bits":["10000001"]},
```

The "fmt" attribute defines the pattern to be matched, which in this case is just a simple instruction without any operands.

If the rule is matched, the "bits" attribute defines the machine code to be emitted. This can be a combination of binary constants and variables. Here we just emit the bits "10000001", i.e. the byte `$81`.

Let's say we want to match the following format, `sta <n>` where `<n>` is a variable 4-bit operand:

```
        sta [0-15]      ; 4-bit constant
```

We can specify different types of variables in the **vars** section of the configuration file. For eample, this defines a 4-bit variable named `const4`:

```
"const4":{"bits":4}
```

To include a variable in a rule, prefix the variable's name with a tilde (" "). For example, our `sta` rule takes one `~const4` variable:

```
{"fmt":"sta ~const4", "bits":["1001",0]},
```

We also have to include the value of the variable in the instruction encoding. To do this, we put an integer into the "bits" array – 0 for the first variable, 1 for the second, etc.

An example: The assembler is given the instruction `sta 15`. It matches the rule `sta ~const4`, and assigns 15 to the first variable slot. It then outputs the the bits "1001" and then the 4-bit value 15, or "1111". The final opcode is "10011111" or `$9f`.

Variables can also be defined by tokens. For example, this rule defines a variable `reg` with four possible values – "a", "b", "ip", or "none", encoded as two bits:

```
"reg":{"bits":2, "toks":["a", "b", "ip", "none"]},
```

Here's an example of a rule which uses it:

```
{"fmt":"mov ~reg,[b]", "bits":["11",0,"1011"]},
```

When decoding `mov a,[b]` the assembler sees that `a` is the first token in the variable, and substitutes the bit pattern "00". The final bit pattern is "11" "00" "1011" which makes a full byte.

More complex instructions are possible, by using multiple variables in a single rule:

```
{"fmt":"~binop ~reg,#~imm8", "bits":["01",1,"1",0,2]},
```

In this rule, `binop`, `reg`, and `imm8` (2) are variables, identified by the integers 0, 1, and 2. `add b,#123` is an example of a matching instruction. This rule emits an opcode 16 bits (two bytes) long.

## 27.2 Assembler Directives

An assembler recognizes certain *directives* that do not emit machine code, but configure various options. In our custom JSASM assembler, these directives are supported:

`.arch <arch>` – Required. Loads the file `<arch>.json` and configures the assembler.

`.org <address>` – The start address of the ROM, as seen by the CPU.

`.len <length>` – The length of the ROM file output by the assembler.

`.width <value>` – Specify the size in bits of an machine word. Default = 8.

`.define <label> <value>` – Define a label with a given numeric value.

`.data $aa $bb ...` – Includes raw data in the output.

`.string .....` – Converts a string to machine words, then includes it in the output.

`.align <value>` – Align the current IP to a multiple of <value>.

## 27.3 Inlining Assembly in Verilog Modules

The 8bitworkshop IDE has special extensions for inline assembly. You can include an assembled ROM in a Verilog module by defining a byte array. For example, if your ROM is 128 bytes long:

```
  reg [7:0] rom[0:127];
```

Then you can initialize the array with inline assembler by using the `__asm` and `__endasm` keywords:

```
  initial begin
`ifdef EXT_INLINE_ASM
    rom = '{
      __asm
        .arch femto8
        .len 128
        ... assembler instructions ...
      __endasm
    };
`endif
  end
```

The length of the assembler binary output (given by the `.len` directive) must match exactly the length of the `rom` array. We'll demonstrate this in the next chapter.

## 27.4 Including Assembly in a Separate File

The 8bitworkshop IDE can also read assembly code from a standalone `.asm` file, and link it to existing Verilog modules. This requires the use of two extra directives:

`.module <filename.v>` – Includes a top-level Verilog module.

`.include <filename.v>` – Includes additional Verilog modules.

The IDE will assemble your code, include the specified Verilog module, and load the output into the `rom` array. We'll demonstrate this in Chapter 34.

# 28

# Racing Game With CPU

To see it on `8bitworkshop.com`: Select the **Verilog** platform, then select the **Racing Game With CPU** file.

In Chapter 18, we made a little top-down racing game with two sprites. We wrote a block of Verilog code to update the speed and positions of the cars. Now we're going to replace the Verilog code with assembly code that runs on the CPU we just designed.

Our 8-bit CPU can only address 256 bytes of memory, which is enough for this task. We're going to define constants for all of the RAM addresses that the CPU will use:

```
parameter PADDLE_X = 0;       // paddle X coordinate
parameter PADDLE_Y = 1;       // paddle Y coordinate
parameter PLAYER_X = 2;       // player X coordinate
parameter PLAYER_Y = 3;       // player Y coordinate
parameter ENEMY_X = 4;        // enemy X coordinate
parameter ENEMY_Y = 5;        // enemy Y coordinate
parameter ENEMY_DIR = 6;      // enemy direction (1, -1)
parameter SPEED = 7;          // player speed
parameter TRACKPOS_LO = 8;    // track position (lo byte)
parameter TRACKPOS_HI = 9;    // track position (hi byte)
```

Some of these memory locations, like `PLAYER/ENEMY X/Y` and `TRACKPOS`, will be read directly by the sprite rendering hardware.

We also have a few special I/O addresses, which give information about the hardware and can be read from the CPU:

```
parameter IN_HPOS = 8'h40;    // CRT horizontal position
parameter IN_VPOS = 8'h41;    // CRT vertical position
parameter IN_FLAGS = 8'h42;   // bitmask of flags
```

`IN_HPOS` and `IN_VPOS` are the positions of the electron beam as per the video sync generator.

`IN_FLAGS` provides several bits of information, like paddle switches, video sync, and sprite collisions.

Now we define our RAM (16 bytes) and ROM (128 bytes) arrays:

```
reg [7:0] ram[0:15];  // 16 bytes of RAM
reg [7:0] rom[0:127]; // 128 bytes of ROM
```

## 28.1 Memory Map

We need to *multiplex* the CPU's *data bus* between both RAM, ROM, and special I/O addresses. This is a function that looks at the `address_bus` input and places the appropriate item on the `to_cpu` output.

We've set up our memory map so that we can use simple bit pattern matching:

| Start | End | Description |
|---|---|---|
| 00000000 | 00001111 | RAM (16 bytes) |
| 01000000 | 01000011 | special I/O region |
| 10000000 | 11111111 | ROM (128 bytes) |

A `casez` statement handles all of the memory regions that the CPU can read. The RAM, ROM, and special memory addresses all have their own patterns:

```
always @(*)
  casez (address_bus)
    // RAM ($00-$0F)
    8'b00??????: to_cpu = ram[address_bus[3:0]];
    // special read registers
    IN_HPOS:  to_cpu = hpos[7:0];
    IN_VPOS:  to_cpu = vpos[7:0];
    IN_FLAGS: to_cpu = {2'b0, frame_collision,
```

```
                vsync, hsync, vpaddle, hpaddle, display_on};
    // ROM ($80-$FF)
    8'b1???????: to_cpu = rom[address_bus[6:0]];
    default: to_cpu = 8'bxxxxxxxx;
  endcase
```

The ROM is selected if the topmost bit is set. This corresponds to memory addresses `$80-$FF`.

The RAM responds to the range `$00-$3F` – if the top two bits are zero. Note that this region is four times larger than the amount of RAM. That's okay, the CPU will just appear to see four identical copies of RAM.

The `IN_` special I/O registers live at `$40`, `$41`, and `$42` and are matched directly.

If any reads take place outside of these regions, the result is undefined. The value `8'bxxxxxxxx` tells the compiler this, and it will not try to synthesize or optimize for these cases.

## 28.2 Sprite Subsystem

The sprite subsystem uses the `PLAYER_X` and `PLAYER_Y` value from RAM to determine when the electron beam reaches the upper-left corner of the sprite:

```
wire player_vstart = {1'0,ram[PLAYER_Y]} == vpos;
wire player_hstart = {1'0,ram[PLAYER_X]} == hpos;
```

Otherwise, it's virtually identical to the circuit described in Chapter 19.

> Note: In this example, both the sprite renderer and the CPU can read from RAM in the same clock cycle. The simulator handles this just fine, but the equivalent circuit would require *dual-port RAM*, which would not be available in the 1970s. There are several ways to avoid the simultaneous access, but for simplicity's sake we'll just leave it.

## 28.3 Racing Game Program

Now we need to write a short program for our CPU to replace the Verilog in our previous example.

We'll start with some assembler directives. We'll specify the FEMTO-8 architecture (our 8-bit CPU design) and that our ROM starts at address 128 ($80) and has a length of 128:

```
.arch femto8
.org 128
.len 128
```

Now we use some `.define` directives to define constants, since our assembler code can't see our Verilog `parameter` statements:

```
.define PADDLE_X 0
.define PADDLE_Y 1
.define PLAYER_X 2
.define PLAYER_Y 3
; ... etc ...
```

The `Start` label denotes the beginning of our assembler program. We'll start by loading some constants into memory:

```
Start:
        lda     #128            ; load initial positions
        sta     PLAYER_X        ; player_x = 128
        sta     ENEMY_X         ; enemy_x = 128
        sta     ENEMY_Y         ; enemy_y = 128
        lda     #180
        sta     PLAYER_Y        ; player_y = 180
        zero    A
        sta     SPEED           ; player speed = 0
        inc     A
        sta     ENEMY_DIR       ; enemy dir = 1 (right)
```

We now read the horizontal paddle position. This requires we loop until the `vpaddle` flag is set, which is in the `IN_FLAGS` register from the CPU's point of view. When we see the flag is set, we read the CRT vertical position from `IN_VPOS` and store that in the player X register:

```
DisplayLoop:
        lda     #F_HPADDLE      ; paddle flag -> A
        ldb     #IN_FLAGS       ; addr of IN_FLAGS -> B
        and     none,[B]        ; read B, AND with A
        bz      DisplayLoop     ; loop until paddle flag set
        ldb     #IN_VPOS
        mov     A,[B]           ; load vertical position -> A
        sta     PLAYER_X        ; store player x position
```

We now wait for the CRT to complete VSYNC, because this means we can modify game registers without affecting the visible frame:

```
        lda     #F_VSYNC
        ldb     #IN_FLAGS
WaitForVsyncOn:
        and     none,[B]
        bz      WaitForVsyncOn  ; wait until VSYNC on
WaitForVsyncOff:
        and     none,[B]
        bnz     WaitForVsyncOff ; wait until VSYNC off
```

Now we check to see if any collisions occurred in the previous frame. If they did, the collision flag will be set, and we'll set the player's speed to a low value:

```
        lda     #F_COLLIDE
        ldb     #IN_FLAGS
        and     none,[B]        ; collision flag set?
        bz      NoCollision     ; skip ahead if not
        lda     #16
        sta     SPEED           ; speed = 16
NoCollision:
```

Regardless of what happens with the collision, we want to increment the player's speed each frame. If it reaches 255 it'll wrap around to zero, so we avoid storing it in this case:

```
        ldb     #SPEED
        mov     A,[B]           ; speed -> A
        inc     A               ; increment speed
        bz      MaxSpeed        ; speed wraps to 0?
        sta     SPEED           ; no, store speed
MaxSpeed:
```

```
        mov     A,[B]           ; reload speed -> A
```

To give the illusion of the track moving past the player's car, we need to add the player's speed to the track position variable. First we divide the speed by 16 (shift right by 4) so it doesn't go too fast:

```
        lsr     A
        lsr     A
        lsr     A
        lsr     A               ; divide speed by 16
```

The track position variable is 16-bit, so we add the low byte first. Note that we have to use the `swapab` instruction because we can only store values to memory from the A register:

```
        ldb     #TRACKPOS_LO
        add     B,[B]           ; B <- speed/16 + trackpos_lo
        swapab                  ; swap A <-> B
        sta     TRACKPOS_LO     ; A -> trackpos_lo
        swapab                  ; swap A <-> B again
        bcc     NoCarry         ; carry flag from earlier add
    op
```

The `add` instruction sets the carry flag, and we test it with the `bcc` (branch if carry clear) instruction. If the carry flag is clear, we skip the next part which increments the high byte of the track position register:

```
        ldb     #TRACKPOS_HI
        mov     B,[B]           ; B <- trackpos_hi
        inc     b               ; increment B
        swapab                  ; swap A <-> B
        sta     TRACKPOS_HI     ; A -> trackpos_hi
        swapab                  ; swap A <-> B again
NoCarry:
```

We could have also used the `adc` (add with carry) instruction, but we only have two registers, and we want to preserve A for the next part, where we update the enemy's vertical position:

```
        ldb     #ENEMY_Y
```

```
        add     A,[B]
        sta     ENEMY_Y         ; enemy_y = enemy_y + speed/16
```

We want the enemy to bounce off the sides of the road as it swerves left and right, so we need to test its X position. We decide that the enemy car will stay within the X range of 64 to 191. If we subtract 64 from the X position, the valid range becomes 0 to 127, and we can use an AND mask to test:

```
        ldb     #ENEMY_X
        mov     A,[B]
        ldb     #ENEMY_DIR
        add     A,[B]
        sta     ENEMY_X         ; enemy_x = enemy_x + enemy_dir
        sub     A,#64
        and     A,#127          ; A <- (enemy_x-64) & 127
        bnz     SkipXReverse    ; skip if enemy_x is in range
```

If the X position is out of range, we negate the `ENEMY_DIR` variable:

```
        zero    A
        sub     A,[B]
        sta     ENEMY_DIR       ; enemy_dir = -enemy_dir
SkipXReverse:
```

The final instruction repeats the process, starting at the `DisplayLoop` label.

```
        jmp     DisplayLoop
```

## 28.4 Lessons Learned

Phew! That was a lot of work, yet only translates to 92 bytes of machine code. The original Verilog code that it replaces is only 17 lines long. What have we learned from this exercise?

- A CPU and its program can replace a hardware circuit, if you have sufficient time to execute. We have thousands of clock cycles between the end of one frame and start of the next, which is plenty for this program.

- A CPU requires a different architecture. We had to add RAM, ROM, and an address bus that chooses between the two. We also had to allow information RAM to pass to the video hardware which renders the sprites and the track.
- For anyone with assembler programming experience, this CPU may seem a little awkward, even for 6502 programmers. We only have two registers, and we have to do a lot of juggling since all ALU and memory operations store into A. It turns out that early CPU designers did a pretty good job!
- We waste a lot of CPU cycles spinning around waiting for the paddle and VSYNC flags. Couldn't we offload some of the sprite rendering duties to the CPU, and have simpler sprite hardware? Yes, and such a system already exists! For details, check out my other book *Making Games For The Atari 2600*.

# 29

# A 16-bit CPU

To see it on `8bitworkshop.com`: Select the **Verilog** platform, then select the **16-Bit CPU** file.

When we talk about 8-bit CPUs, we are often speaking only of the width of the data bus. Many CPUs considered 8-bit have 16-bit features. For example, both the 6502 and Z80 have a 16-bit address bus, and the Z80 has some 16-bit arithmetic operations. But they both can only externally send or receive 8 bits at a time.

One notable exception was the CP1600 which powered the original Intellivision. It had 16-bit data and address buses, even though both were multiplexed through the same pins, and instructions were only 10 bits wide.

The CPU described in Chapter 26 is wholly 8-bit, including the address bus. To power our virtual game console we're going to design a wholly 16-bit CPU, called the *FEMTO-16*. We're going to take inspiration from our previous design, but we're going to have more registers and a completely different instruction set.

The FEMTO-16 has eight 16-bit registers:

| Index | Name | Description |
|---|---|---|
| 0 | AX | general purpose |
| 1 | BX | ... |
| 2 | CX | ... |
| 3 | DX | ... |
| 4 | EX | ... |
| 5 | FX | ... |
| 6 | SP | stack pointer |
| 7 | IP | instruction pointer |

The only "special" registers are SP and IP; the other six are general purpose, although most operations can be performed with any register. Note that IP is a bonafide register on par with the others, whereas in the previous design it was a "hidden" register.

Each instruction is 16 bits wide, and there are a lot more of them:

| HighByte LowByte | Operation | Example |
|---|---|---|
| `00000aaa 0++++bbb` | $A \leftarrow A \otimes B$ | `add ax,bx` |
| `00001aaa 0++++bbb` | $A \leftarrow A \otimes mem[B]$ | `sub cx,[bx]` |
| `00011aaa 0++++000` | $A \leftarrow A \otimes imm16$ | `add dx,#12345` |
| `00101aaa ########` | $A \leftarrow mem[const8]$ | `mov ex,[255]` |
| `00110aaa ########` | $A \rightarrow mem[const8]$ | `mov [255],ex` |
| `01001aaa #####bbb` | $A \rightarrow mem[B + const5]$ | `mov [ax+15],fx` |
| `01101aaa 0++++000` | $A \leftarrow A \otimes mem[imm16]$ | `add ax,[9876]` |
| `01110aaa 00cccbbb` | $A \rightarrow mem[B], IP \leftarrow C$ | `push fx` |
| `1000tttt ########` | conditional branch | `bz 123` |

| | |
|---|---|
| `aaa, bbb, ccc` | register index (0-7) |
| `++++` | ALU operation ($\otimes$) |
| `########` | 8-bit constant |
| `#####` | 5-bit constant |
| `tttt` | flags test for conditional branch |

*Figure 29.1: FEMTO-16 instruction encoding*

## 29.1 ALU

Like the 8-bit design, we have a couple bits in the encoding that have special meaning across several instructions.

| | |
|---|---|
| `........ .....bbb` | Source Register |
| `.....aaa ........` | Destination Register |
| `....x... ........` | Load memory -> ALU B? (`Bload`) |
| `x....... ........` | Lower 8 bits -> ALU B? (`Bconst`) |

The last two flags are used to route data to the ALU. The ALU is identical to the 8-bit design, we just use a parameter to extend it to 16 bits:

```
ALU #(16) alu(
  .A(regs[rdest]),
  .B(Bconst ? {8'b0, opcode[7:0]} // load 8-bit constant
    : Bload ? data_in             // load data bus
            : regs[rsrc]),        // load register
  .Y(Y),
  .aluop(aluop),
  .carry(carry));
```

Note that the ALU's B input can be selected from the lower 8 bits of the instruction, the data bus, or one of 8 registers.

## 29.2 Select and Hold/Busy

The `SELECT` state is similar to the 8 bit CPU – it sets up the address bus to read the next instruction.

But in this CPU, it also checks the `hold` input. If it is set, it will assert its `busy` output and idle until `hold` is clear.

This feature is used by the graphics subsystem to read directly from RAM, pausing the CPU to ensure they don't collide.

The `hold` input is only checked in the `SELECT` state, so an external device has to wait long enough to ensure the CPU has reached this state. In the current implementation, it's guaranteed that the CPU will enter this state within 5 clock cycles.

## 29.3 Decode

The first thing we do is copy the data bus (`data_in`) to the `opcode` register:

```
opcode <= data_in; // (only use opcode next cycle)
```

Note that since this is a non-blocking operation, the new `opcode` won't be available until the next state. We'll have to remember to use `data_in` until the next step.

The `DECODE` state is also similar in that it is a big `casez` statement. For example, here's a simple register-to-register operation:

```
//  00000aaa0++++bbb  operation A+B->A
16'b00000???0???????: begin
  aluop <= data_in[6:3];
end
```

Note that all we have to do is configure the `aluop` register; all other ALU inputs are directly wired to the `opcode` register.

The 16 bit opcode is enough to embed an 8-bit constant in some instructions. If bit 15 is set, the ALU gets its B input from the lower 8 bits of the opcode. So all we have to do is configure the `aluop` register, just like the register-to-register operation:

```
//  11+++aaa########  immediate binary operation
16'b11??????????????: begin
  aluop <= data_in[14:11];
end
```

## 29.4 Compute

The `COMPUTE` state looks more-or-less like the 8-bit version, except that the register file is an array:

```
// state 3: compute ALU op and flags
S_COMPUTE: begin
  // transfer ALU output to destination
  regs[rdest] <= Y[15:0];
  // set carry for certain operations (4-7,12-15)
```

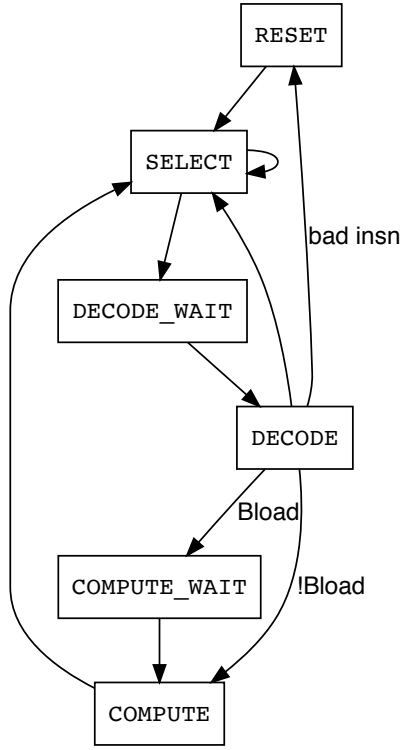



*Figure 29.2: State diagram of FEMTO16 CPU using memory wait states.*

```
        if (aluop[2]) carry <= Y[16];
        // set zero, negative flags
        zero <= ~|Y[15:0];
        neg <= Y[15];
        // repeat CPU loop
        state <= S_SELECT;
    end
```

## 29.5 Wait States

Since we are using *synchronous RAM*, we have to wait one clock cycle for the result after putting a value on the address bus. This requires that we put *wait states* in our CPU when reading memory.

A wait state is always added after the `SELECT` state. Whether a wait state is added after `DECODE` depends on bit 11 of the opcode. If this bit is set, the ALU will read from the data bus, so we must insert a wait state before going to the `COMPUTE` state:

```
state <= RAM_WAIT && data_in[11] ? S_COMPUTE_WAIT :
  S_COMPUTE;
```

## 29.6 Relative Branching

The branch instruction works like the 8-bit CPU, except it's a *relative branch*. Instead of an exact address, the branch target specifies an offset. If the branch is taken, we add the signed 8-bit branch offset to the current IP:

```
// relative branch, sign extended
regs[IP] <= regs[IP] + 16'($signed(data_in[7:0]));
```

The `$signed` function converts an unsigned value to a signed value.

The `16'()` function is a *cast* which extends the 8-bit value to a 16-bit value. Since the value is signed, this function performs *sign extension* which copies the high bit into the unused slots.

Our assembler has to be configured to emit relative branches. In the configuration file for the assembler, we define an 8-bit IP-relative variable like so:

```
"rel8":{"bits":8, "iprel":true, "ipofs":1}
```

The `ipofs` attribute biases the relative branch value. Since the IP has already been incremented when the branch instruction is decoded, we need to subtract 1 from the offset. So an instruction that branches to itself would have the value -1 (`$FF`).

## 29.7 Push and Pop

Our new CPU has a special *stack pointer* register. This allows us to *push* and *pop* to a *stack*.

A stack is a special region of memory with *LIFO* (last-in-first-out) access semantics. This means that a *push* adds values, and a *pop* returns these values in reverse order. The stack pointer register tracks the memory address of the top of the stack (i.e. the last value pushed).

Memory accesses using the stack pointer register are special: they modify the stack pointer after the load or store operation. So whenever the CPU stores to memory using the `[SP]` register, it decrements the `SP` register:

```
if (data_in[2:0] == SP)
  regs[SP] <= regs[SP] - 1;     // push to stack
```

Conversely, loading from memory using the `[SP]` register automatically increments the `SP` register:

```
if (data_in[2:0] == SP)
  regs[SP] <= regs[SP] + 1;     // pop from stack
```

## 29.8 Subroutines

Our new CPU supports *subroutines*. This allows the CPU to transfer control to another bit of code, and eventually return to the original instruction stream to pick up where it left off.

Before the CPU calls a subroutine, it needs to record the current value of the instruction pointer on the stack. When the subroutine finishes, it pops this value off the stack and back into the instruction pointer. This works even if there are multiple nested levels of subroutines.

The `jsr` opcode handles calling subroutines. It pushes the instruction that follows the `jsr` instruction (i.e. where to return) onto the stack, then transfers a new value into the IP register.

The `jsr` instruction decoding is similar to how `push ip` would be decoded, except it specifies an additional register which receives the old value of the IP register:

```
regs[IP] <= regs[data_in[5:3]];      // register -> IP
```

The address of the subroutine to be called must come from a register. So when writing assembly code, we must first load the subroutine address into a register, then call the subroutine via the `jsr` instruction:

```
mov     fx,<newip>
jsr     fx
```

The end of a subroutine is marked by a `rts` instruction, which is just a synonym for `pop ip`.

# 30

# Framebuffer Graphics

To see it on `8bitworkshop.com`: Select the **Verilog** platform, then select the **Framebuffer** file.

To display arbitrary pixels on the screen, you want a *framebuffer*. This is a (usually) large portion of RAM that represents pixels on the screen.

Some designs (Colecovision, NES) use *dedicated video RAM* for this purpose. This frees up memory bandwidth for rendering graphics to the screen, but makes it more difficult for the CPU to write to video RAM. For example, the NES's CPU has to wait for the *PPU* (the video processor, sometimes called a *VPU*) to become idle, then write several bytes to the PPU's input port telling it which bytes to alter.

Other computer designs (Apple ][, Williams games) use a *shared framebuffer* that the CPU can read and write directly. This makes it easier to program, but since the video circuitry and RAM share access to RAM, the hardware must be designed carefully to prevent them from conflicting.

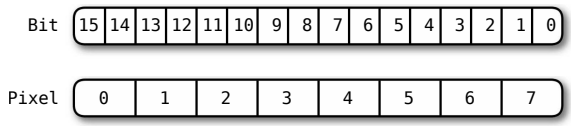
Steve Wozniak designed the Apple ][']s video generator to take advantage of the 6502's *two-phase clock* – the CPU only touches memory during one-half of its clock cycle, allowing the video generator and *memory refresh* circuit access to RAM during the other half.

Other systems like the Bally Astrocade and Atari 8-bit computers steal individual cycles from the CPU when the video subsystem needs to access memory.

In this example, we'll pretend we have *dual-port RAM*, and give the CPU and video RAM to their own channels for reading and writing. (We'll address the issue of bus contention in the next chapter.)

## 30.1 Design

Our frame buffer stores each 256-pixel scanline as 32 16-bit words. Each word has eight 2-bit pixels:




Note that these are arranged so that the upper two bits define the leftmost pixel in a word, and the lower two bits define the rightmost pixel. This is an arbitrary decision; some architectures reverse the pixels.

Each pixel has 4 possible values, each corresponding to a color in a *palette* array. We'll store our four RGB values in registers – the default values correspond to black, red, blue, and white:

```
reg [3:0] palette[0:3] = '{0,1,4,7};
```

When the electron beam is scanning a visible part of the frame, we read from RAM and display pixels. We read one 16-bit word every 8 pixels, loading each word into a *shift register*. On cycles that we're not reading from RAM, we shift the register two bits to the left (i.e. by one pixel):

```
// load next word from RAM every 8 pixels
if (0 == hpos[2:0]) begin
  vshift <= ram[{2'b10,vindex}]; // read into vshift
  vindex <= vindex + 1; // next video address
end else
  vshift <= vshift << 2; // shift next pixel in 16-bit word
```

On every cycle, we look up the upper two bits in our 4-entry palette table, and put the result into `rgb`:

```
rgb <= palette[vshift[15:14]];
```

When we're outside of the visible frame, we set the RGB color to black. When `vsync` is high, we also reset `vindex` (the current video address in RAM) to zero:

```
rgb <= 0; // set color to black
if (vsync) vindex <= 0; // reset vindex every frame
```

Now we'll write a little *FEMTO-16* assembler program to exercise our new frame buffer. It'll just XOR consecutive values to consecutive memory addresses, which gives us an animated test-pattern:

```
Start:
        mov ax,#0       ; ax = 0
        mov bx,ax       ; bx = ax
Loop:
        xor ax,[bx]     ; ax = ax ^ ram[bx]
        mov [bx],ax     ; ram[bx] = ax
        inc bx          ; bx = bx + 1
        inc ax          ; ax = ax + 1
        bnz Loop        ; loop until ax is 0
        reset           ; reset CPU
```

The frame buffer approach is simple and flexible, but has some drawbacks in these early constrained systems. For example, this 4-color frame buffer uses 15 KB of RAM, and about 10 percent of our memory bandwidth is dedicated to fetching pixels to display (it would be more like 20 percent with an 8-bit bus.) If we wanted more colors or higher resolution, we'd need to use even more RAM and bandwidth.

Another drawback of a frame buffer is that the CPU must perform all animation duties, erasing and redrawing objects in the frame buffer, and optionally overlapping them with a background. This can make for slow animation as the number of objects increases.

In the next chapter, we'll design a *tile graphics* renderer that uses much less memory, and allows for many more colors. We'll also use the *synchronous RAM* module, which introduces some new challenges.

# 31

# Tilemap Rendering

To see it on `8bitworkshop.com`: Select the **Verilog** platform, then select the **Tile Renderer** file.

In Chapter 15 we made a simple tile-graphics renderer that displayed 32 columns by 30 rows of digits. However, this module has several limitations:

- It uses dedicated video RAM.
- The video scanner hogs the RAM address bus, preventing other circuits (like a CPU) from sharing the RAM bandwidth.
- It does not have configurable colors.

We're going to make a new tile-graphics renderer that improves upon this, at the cost of some complexity.

The most important change is that the tile renderer is going to share RAM with the CPU, and read the tile map (the array that defines which characters are in which cells) from RAM. We'll still fetch the actual character patterns out of ROM.

We'll use 16-bit words instead of 8-bit bytes, allowing more data to be fetched per cycle.

Our tile renderer needs to know at which RAM addresses to look for video data. For flexibility, we'll use *lookup tables*.

The *row table* is the only fixed address in the system, spanning addresses $7E00 to $7E1F. It contains a list of 32 addresses, each pointing to the start of a row of tiles.

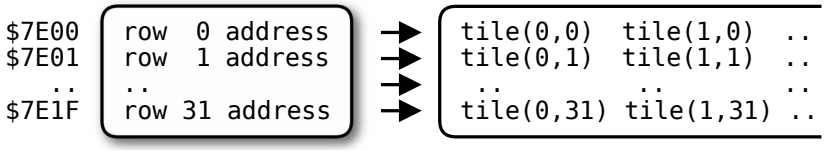



*Figure 31.1: Row table for tile renderer.*

Each row contains a list of 16-bit values, each of which combines pattern and color information for a single cell. The upper 8 bits gives the background and foreground color, while the lower 8 bits is an index into the character bitmap ROM. The color byte is split into two *nibbles* (4 bits) with the background in the upper nibble and foreground in the lower nibble.

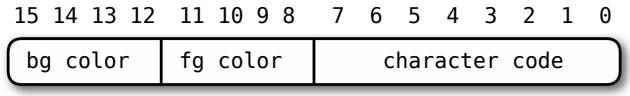



*Figure 31.2: Layout of a 16-bit tile word.*

## 31.1 Sharing RAM with the CPU

We've set up our *FEMTO-16* CPU so that it can receive a *hold signal*. When the CPU acknowledges this signal, it asserts its *busy output* and waits until the hold signal is cleared. It then resumes execution.

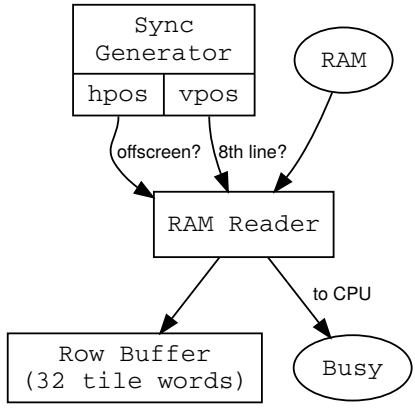
We can use these signals to mediate access between the CPU and video generator. The video generator has hard real-time requirements, as it has to read a scanline of data approximately every 63 microseconds. So before our tile renderer reads data from RAM, we'll send the `hold` signal to the CPU.

By design, the CPU may take up to five clock cycles before it responds to this signal and halts, so we'll make sure we assert `hold` five cycles before we start reading from RAM.

Ideally, we'd like to read an entire row of data in one contiguous RAM access. This will minimize the number of times we have to flip the CPU in and out of *busy mode*, since we waste cycles each time.

## 31.2 The Tile Renderer State Machine

The tile renderer has two conceptual parts: the fill stage and the output stage. The fill stage reads words from RAM and transfers them to an internal buffer. It uses the video sync generator's `hpos` and `vpos` signals to know when to start filling – at the end of every 8th scanline.




Our module will have an internal buffer of 32 16-bit words, to hold the tile data for the current row:

```
reg [15:0] row_buffer[0:31];
```

We're going to have to ensure that we don't output a value until the fill stage is done reading the value for that scanline. So we'd like to defer the RAM read as late as possible in the scanline. The `HLOAD` constant defines the horizontal position where we start reading:

```
// start loading cells from RAM at this hpos value
// first column read will be ((HLOAD-2) % 32)
parameter HLOAD = 272;
```

Since our tiles are 8x8, we only need to read from RAM every 8 scanlines. We'll read on the last scanline of each row:

```
// time to read a row?
if (vpos[2:0] == 7) begin
```

Our state machine is entirely dependent on `hpos`, the horizontal position of the video beam. We have a `case` statement that runs most of this logic:

```
    case (hpos)
```

At `HLOAD-8` we assert the `ram_busy` signal to tell the CPU to start getting ready to halt:

```
        // assert busy 5 cycles before first RAM read
        HLOAD-8: ram_busy <= 1;
```

We give the CPU a full five cycles to halt, allowing it to finish its current instruction. At `HLOAD-3` we set `ram_addr` (connected to the RAM address bus) to the current entry in the page table:

```
        // set address for row in page base table
        HLOAD-3: ram_addr <= {page_base, 3'b000, row};
```

Since `ram_addr` is a register set in a synchronous `always` block, it takes one cycle to set the address bus value. So the address bus value won't be set until `HLOAD-2`.

Also, in this example we use *synchronous RAM*. It takes a clock cycle to return a read result (see Chapter 14). So the read result won't be available until `HLOAD-1`.

At this point we grab the value on the `ram_read` bus and store it in the internal register `row_base`:

```
        // read row_base from page table (2 bytes)
        HLOAD-1: row_base <= ram_read;
```

The copying operation happens outside of the `case` statement, from `HLOAD` to `HLOAD+33`, a total of 34 cycles. Why 34, when our rows have 32 columns?

```
// load row of tile data from RAM
// (last two twice)
if (hpos >= HLOAD && hpos < HLOAD+34) begin
```

A synchronous RAM read takes two cycles (set address bus + wait for synchronous RAM data). We can use *pipelining* to fill our array as fast as possible. We'll set the address bus based on the current value of `hpos`:

```
// set address bus to (row_base + hpos)
ram_addr <= row_base + 16'(hpos[4:0]);
```

The value at that memory address will appear on the data bus in two clock cycles. So the value on the data bus in this cycle was requested two cycles ago. Thus, we'll subtract two from `hpos` before writing to the `row_buffer` array:

```
// store value on data bus from (row_base + hpos - 2)
// which was read two cycles ago
row_buffer[hpos[4:0] - 2] <= ram_read;
```

The first two cycles will have garbage on the read bus. Since `row_buffer` has 32 entries and the read operation lasts 34 clock cycles, the array will wrap around, and the first two entries will be written in the last two clock cycles.

Our `case` statement does nothing until `HLOAD+34`, where we increment the `row` counter and let the CPU get back to its business:

```
// deassert BUSY and increment row counter
HLOAD+34: begin
  ram_busy <= 0;
  row <= row + 1;
end
```

The entire row read timing is described in Figure 31.3.

## 31.3 Rendering the Buffer

Meanwhile, the output stage is reading from the internal buffer each scanline. It looks up the pattern for each tile in ROM,

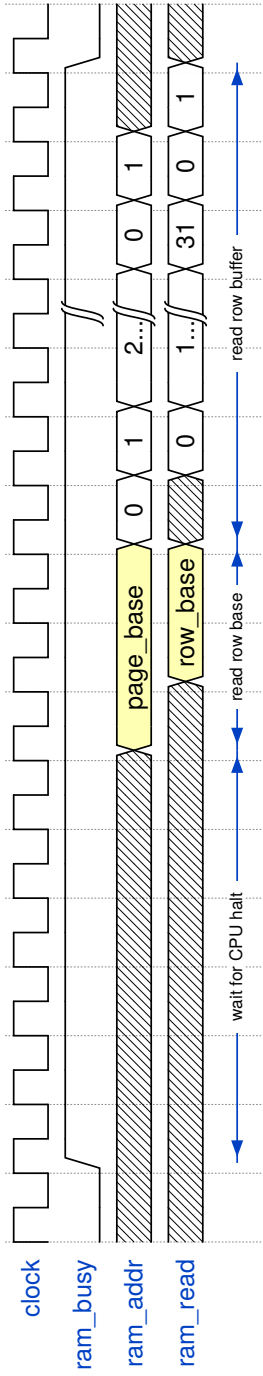
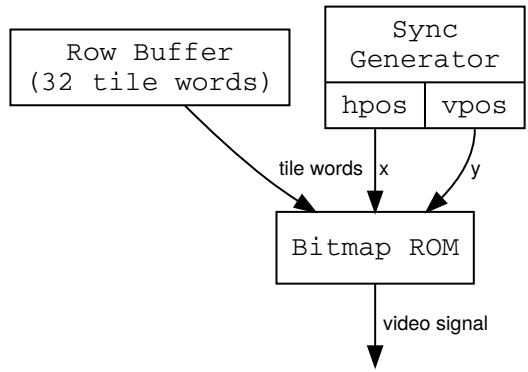



*Figure 31.3: Timing diagram for tile renderer*

putting it into an internal register. It uses this bitmask to select the foreground or background color for each pixel, and outputs to the `rgb` module port.




On the 7th pixel of each cell (right before the next cell begins) we latch the next cell (`col+1`) from `row_buffer`. We have a special case for `row_buffer[0]`, which we load on `hpos == 308`, right before the next scanline begins.

```
// latch character data
if (hpos < 256) begin
  case (hpos[2:0])
    7: begin
      cur_cell <= row_buffer[col+1];
    end
  endcase
end else if (hpos == 308) begin
  cur_cell <= row_buffer[0];
end
```

`cur_cell` is split into two bytes, `cur_char` and `cur_attr`. We can compute our ROM address by concatenating `cur_char` and `yofs` (essentially `cur_char*8 + yofs`):

```
assign rom_addr = {cur_char, yofs};
```

The final step is to lookup the data from the ROM. We use either the lower or upper nibble of `cur_attr` (foreground or

background) depending on whether the bit in the ROM is on or off:

```
// extract bit from ROM output
assign rgb = rom_data[~xofs] ? cur_attr[3:0] : cur_attr[7:4];
```

Each nibble of the attribute byte gives us 16 possible RGB colors.

## 31.4 Improvements

There are lots of different ways to implement this module. We designed it to share video memory with a CPU, which forces us to fetch memory a certain way. We could have more flexibility by removing this requirement, but we'd still have to design some sort of communication channel.

Right now we are limited to 16 colors, but we could also have implemented *palettes* to expand this range without requiring more RAM. Instead of 4-bit RGB colors, we'd have a 4-bit index into a color lookup table.

We could also implement *scrolling* by expanding the amount of RAM and adding new registers and logic. In our current architecture, horizontal scrolling would probably be easiest, since we'd just have to modify the order in which we scan the row buffer and write it to the display. We could even have a lookup table so that each row can have a different scroll offset.

We'd also probably want to expand the size of RAM and/or hide a portion of the screen. Then we can update cells offscreen to make it look like the tile map is bigger than the screen. We could even read character patterns out of RAM.

There are many other possible architectures. For example, the Atari 8-bit computers used a flexible *display list* system, which divides the screen into slices of scanlines. Each slice can be configured as one of several text or graphics modes, and scrolled independently.

# 32

# Scanline Sprite Rendering

To see it on `8bitworkshop.com`: Select the **Verilog** platform, then select the **Sprite Scanline Renderer** file.

We would like to have dozens of smoothly moving sprites on the screen that overlap the background. We could do this with a framebuffer and some complex logic, but having dedicated sprite hardware is pretty nice. Let's think through some ways to implement this.

We could create a `sprite_renderer` module for each sprite, then hook them all up in parallel. But this is a pretty complex module, and would replicate a ton of logic for each sprite. Also, the memory bandwidth scales linearly, as each sprite's ROM must be read before each scanline.

A more economical technique used by the NES is to have a number of *shift register*s. Before each scanline, these are loaded with sprite data depending on which subset of sprites intersect that scanline. At the appropriate horizontal position, the pixels are shifted out onto the screen. There are fewer shift registers than sprites, which means there's a limit to how many sprites can appear on a single scanline.

We'll do something similar, but instead of using shift registers, we'll write to a RAM buffer during each scanline. The *Galaxian* hardware uses a similar technique.

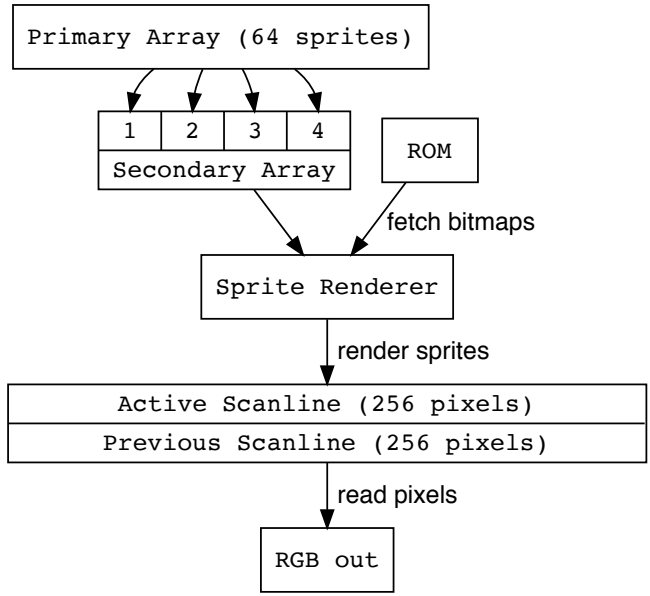



*Figure 32.1: Data flow of sprite scanline renderer.*

This renderer is a complex state machine. During each scanline, it runs through two phases:

1. Scan through the main sprite array, figuring out which intersect with the current scanline. Mark those indices in a secondary array.
2. Scan through the secondary array. For each active sprite, load its bitmap slice from ROM and render it to the active *line buffer*.

Meanwhile during all of this, the previous (inactive) line buffer is being output to the screen. It is *double buffered*, so the two line buffers switch every scanline – the active buffer becomes inactive, and vice-versa. During rendering, the line buffer is erased in preparation for the next scanline.

This is a similar strategy to the tile renderer described in Chapter 31, but even more complex.

Our first module will handle 32 sprites in the primary array, with a maximum of 8 sprites per scanline. The primary array is read from shared RAM, once per frame.

## 32.1 RAM and ROM

Our module will use a dedicated ROM (for sprite bitmaps) and shared RAM (for sprites' position and color).

```
module sprite_scanline_renderer(clk, reset, hpos, vpos, rgb,
                                ram_addr, ram_data, ram_busy,
                                rom_addr, rom_data);

  input clk, reset;     // clock and reset inputs
  input [8:0] hpos;     // horiz. sync pos
  input [8:0] vpos;     // vert. sync pos
  output [3:0] rgb;     // rgb output

  output [NB:0] ram_addr; // RAM for sprite data
  input [15:0] ram_data;  // (2 words per sprite)
  output ram_busy;        // set when accessing RAM

  output [15:0] rom_addr; // sprite ROM address
  input [15:0] rom_data;  // sprite ROM data
```

## 32.2 Scanline Buffer

The RGB scanline buffer is 512 entries long (2 scanlines of 256 pixels):

```
  // 4-bit RGB dual scanline buffer
  reg [3:0] scanline[0:511];

  // which offset in scanline buffer to read?
  wire [8:0] read_bufidx = {vpos[0], hpos[7:0]};
```

Every clock cycle, we read a pixel of the scanline buffer to the `rgb` output, and simultaneously set that pixel to zero:

```
    // read and clear buffer
    rgb <= scanline[read_bufidx];
    scanline[read_bufidx] <= 0;
```

## 32.3 Sprite State Machine

The sprite state machine has three phases:

1. Load sprite data from RAM and put them into the `sprite_xxx` arrays (once per frame)
2. Iterate through the sprite list and select the sprites that will appear, and map them into slots (once per scanline)
3. Iterate through the sprite slots, looking up each bitmap slice in ROM and rendering it to one of the two scanline buffers.

## 32.4 Sprites and Slots

Our sprite renderer can be configured with two parameters:

`NB` – The maximum number of sprites per frame.

`MB` – The maximum number of slots (sprites per scanline).

These parameters are powers of two, so `NB = 5` means $2^{NB}$ or 32 sprites.

```
parameter NB = 5; // 2^NB == number of sprites
parameter MB = 3; // 2^MB == slots per scanline

localparam N = 1<<NB; // number of sprites
localparam M = 1<<MB; // slots per scanline
```

Among other things, we use the parameters to size arrays. The module has local memory that keeps copies of the sprite data after it is read from RAM:

```
// copy of sprite data from RAM (N entries)
reg [7:0] sprite_xpos[0:N-1]; // X positions
reg [7:0] sprite_ypos[0:N-1]; // Y positions
reg [7:0] sprite_attr[0:N-1]; // attributes
```

And it holds similar data for the per-scanline slots, with a boolean marking whether the slot is active:

```
// M sprite slots
reg [7:0] line_xpos[0:M-1]; // X pos for M slots
```

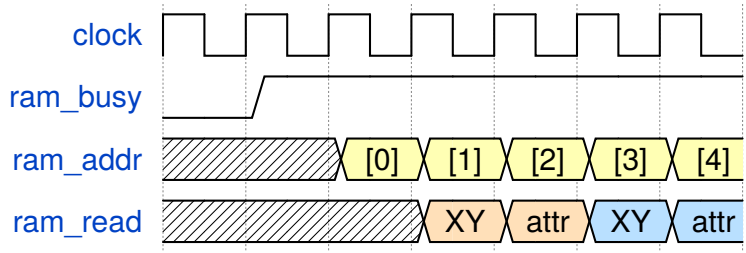



*Figure 32.2: Sprite RAM fetch timing*

```
reg [7:0] line_yofs[0:M-1]; // Y pos for M slots
reg [7:0] line_attr[0:M-1]; // attr for M slots
reg line_active[0:M-1];     // slot active?
```

## 32.5 Loading Sprite Data From RAM

The sprite renderer only accesses RAM on scanline 260, outside of the viewable CRT frame. We allow 2 clock cycles per sprite, and read two 16-bit words for each. These are copied into the `sprite_xxx` arrays.

The first few sprites may be corrupted because the CPU might be busy (it can take up to 5 cycles for it to halt) so we read them again after all other sprites are read.

```
// which sprite are we currently reading?
wire [NB-1:0] load_index = hpos[NB:1];

  ram_busy <= 0; // assume RAM not busy unless we set it
  // reset every frame, don't draw vpos >= 256
  if (reset || vpos[8]) begin
    // load sprites from RAM on line 260
    // 8 cycles per sprite
    // do first sprite twice b/c CPU might still be busy
    if (vpos == 260 && hpos < N*2+8) begin
      ram_busy <= 1;
      case (hpos[0])
        0: begin
          ram_addr <= {load_index, 1'b0};
          // load X and Y position (2 cycles ago)
          sprite_xpos[load_index] <= ram_data[7:0];
          sprite_ypos[load_index] <= ram_data[15:8];
```

```
      end
      1: begin
        ram_addr <= {load_index, 1'b1};
        // load attribute (2 cycles ago)
        sprite_attr[load_index] <= ram_data[7:0];
      end
    endcase
  end
```

We use the pipelining trick discussed in Chapter 15 to load a word from RAM in every clock cycle. Our read loop takes two cycles, and RAM latency is two cycles, so everything lines up nicely. It doesn't even matter if the sprite entries are out-of-order.

## 32.6 Iterating The Sprite List

At the beginning of each scanline, we scan through all the sprites and determine which intersect this scanline. Then we save its Y offset, mark it active, and update its `sprite_to_line` mapping. This takes 2 cycles per sprite.

```
end else if (hpos < N*2) begin
  // setup vars for next phase
  k <= 0;
  romload <= 0;
  // select the sprites that will appear in this scanline
  case (hpos[0])
    // compute Y offset of sprite relative to scanline
    0: z <= 8'(vpos - sprite_ypos[i]);
    1: begin
      // sprite is active if Y offset is 0..15
      if (z < 16) begin
        line_xpos[j] <= sprite_xpos[i]; // save X pos
        line_yofs[j] <= z; // save Y offset
        line_attr[j] <= sprite_attr[i]; // save attr
        line_active[j] <= 1; // mark sprite active
        j <= j + 1; // inc counter
      end
      i <= i + 1; // inc main array counter
    end
  endcase
```

## 32.7 Sprite Setup

After populating the slots, we go to rendering. We allow 18 cycles for each sprite slot, 2 for setup and 16 for rendering. The pseudocode for this process looks like this:

```
// if sprite shift register is empty, setup new sprite
if (out_bitmap == 0) begin
  // look up sprite ROM address (1st cycle)
  // load bitmap and sprite attributes (2nd cycle)
end else begin
  // write pixels into scanline buffer (up to 16 cycles)
end
```

The setup phase has to do a variety of operations:

- Compute the offset of the dual scanline buffer to which we'll write pixels, using the sprite's X coordinate.
- Compute the sprite's ROM address and fetch the 16-bit bitmap slice.
- Lookup the sprite's attribute (RGB color).
- If the sprite slot is inactive, we set its bitmap slice to all transparent pixels (zero).
- Clear the active bit and increment to the next slot.

We need at least two clock cycles to complete this, so we use the `romload` flag to switch between two phases. In the first cycle, we set the ROM address:

```
case (romload)
  0: begin
    // set ROM address and fetch bitmap
    rom_addr <= {4'b0, line_attr[k][7:4], line_yofs[k]};
  end
```

We do everything else in the second cycle:

```
  1: begin
    // load scanline buffer offset to write
    write_ofs <= {~vpos[0], line_xpos[k]};
    // fetch 0 if sprite is inactive
    out_bitmap <= line_active[k] ? rom_data : 0;
    // load attribute for sprite
```

```
      out_attr <= line_attr[k];
      // disable sprite for next scanline
      line_active[k] <= 0;
      // go to next sprite in 2ndary buffer
      k <= k + 1;
    end
  endcase
```

Then we toggle `romload` so that we complete another two-cycle slot setup:

```
  romload <= !romload;
```

## 32.8 Sprite Rendering

Once a slot is setup, rendering is relatively simple. For each of the 16 pixels, we see if the low bit is set. If it is, we write a pixel to the scanline buffer. Then we shift the `out_bitmap` variable right one bit, and increment `write_ofs`:

```
    // write color to scanline buffer if low bit == 1
    if (out_bitmap[0])
      scanline[write_ofs] <= out_attr[3:0];
    // shift bits right
    out_bitmap <= out_bitmap >> 1;
    // increment to next scanline pixel
    write_ofs <= write_ofs + 1;
  end
```

## 32.9 Improvements

The current module writes sprite pixels to a scanline buffer. It may be more efficient to bypass the buffer and instead use several *shift registers*, one for each sprite slot. Each register can be loaded with sprite data, and its bits shifted out whenever the beam reaches the corresponding sprite's horizontal position.

Another improvement is to have multicolor sprites with configurable palettes. We could do this by having multiple bitmap *planes* that stack on top of each other. The sprite renderer

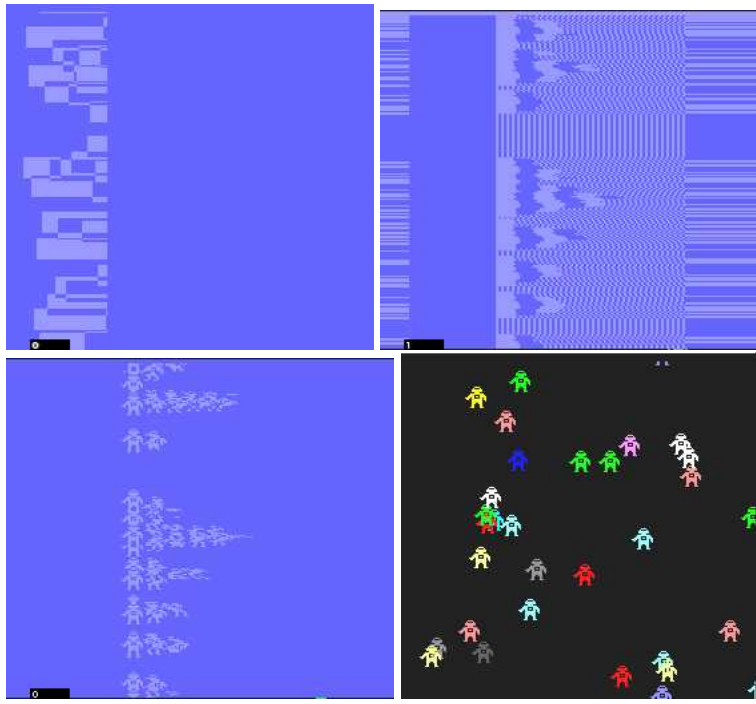


*Figure 32.3: Double-clicking on variables in the 8bitworkshop IDE activates the inspection feature, which shows the least-significant bit of variables during rendering. From left to right, the j variable (which slot is being filled), the k variable (which sprite is being rendered), the sprite data shift register, and the RGB output.*

combines the bits in all of the planes, then looks up that value in a palette ROM to fetch the pixel color.

We could also add flags for flipping the X/Y axis, and collision detection registers.

# 33

# A Programmable Game System

To see it on `8bitworkshop.com`: Select the **Verilog** platform, then select the **CPU Platform** file.

We've built several components which work similarly to those in an early 1980s game console or microcomputer:

- Video sync generator (Chapter 9)
- Synchronous RAM (Chapter 14)
- Tile graphics renderer (Chapter 31)
- Sprite scanline renderer (Chapter 32)
- Sound generator (Chapter 24)
- CPU (Chapter 29)

Now we're going to combine them all into a unified system, and write a game!

The specifications:

- 64 kilobytes (32,678 16-bit *words*) of RAM, shared between the CPU and video subsystems
- 16-bit CPU running at 4.857 MHz
- 32x30 tile graphics with 256 x 8 tile ROM
- 32 16 x 16 sprites per frame with sprite ROM
- 16 colors (two per tile, one per sprite)
- Two game controllers (four direction switches, two buttons)
- One paddle/analog stick controller

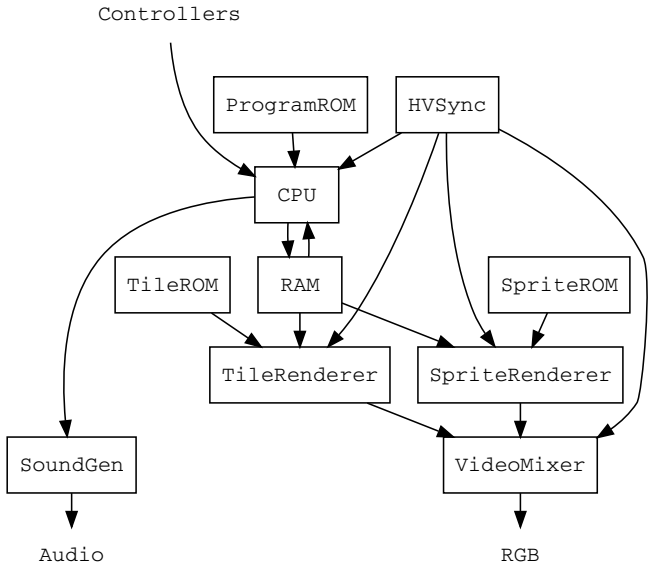



*Figure 33.1: Data flow diagram of game platform*

## 33.1 Shared RAM

One of the design decisions we've made is to allow RAM to be shared between CPU and video subsystems. We need to *multiplex* access to RAM between them, but also make sure they don't step on each other.

The tile and sprite renderers are designed so that they never access RAM at the same time and thus do not conflict. But they both need to tell the CPU to halt so they can safely access RAM.

Both renderers have a `ram_busy` output which they assert while they are conducting RAM transfers. We assign those outputs to the `tile_reading` and `sprite_reading` wires. We then feed those into the CPU's `busy` input:

```
.hold(tile_reading | sprite_reading),
```

When the CPU finally halts after a number of cycles (up to 5 in our design) it will assert its `busy` output. Normally, we'd select

the address requested by the CPU. When `cpu_busy` is active, we'll select the address requested by the tile renderer or sprite renderer, depending on which is active:

```
always @(*)
  if (cpu_busy) begin
    if (sprite_reading)
      mux_ram_addr = {9'b111111100, sprite_ram_addr};
    else
      mux_ram_addr = tile_ram_addr[14:0];
  end else
    mux_ram_addr = cpu_ram_addr[14:0];
```

The sprite renderer uses its own ROM, so it only needs to read sprite attributes in the range `$7F00-$7F3F` (32 sprites, 2 words each). The tile renderer and CPU have access to the full 32KB memory space.

## 33.2 Memory Map

Now we have to define our *memory map*. This defines how our RAM and ROM is laid out in address space, and any other special addresses that are used for I/O.

```
$0000-$7DFF  RAM
$7E00-$7EFF  Page Table (256 words)
$7F00-$7F3F  Sprite Table (64 words)
$7F40-$7FFF  Extra RAM
$8000-$FFFD  ROM
$FFFE        Controller switches register
$FFFF        Flags register
```

```
  wire [15:0] flags = {11'b0, vsync, hsync, vpaddle, hpaddle,
    display_on};
  wire [15:0] switches = {switches_p2, switches_p1};
```

When the CPU is active, we have to *multiplex* RAM, ROM, and the flags and switches registers. If the high bit is zero, we select the RAM region. If the high bit is set, we select either ROM or the special registers. We do this with a `casez` statement:

```
  // select ROM, RAM, switches ($FFFE) or flags ($FFFF)
```

```
always @(*)
  casez (cpu_ram_addr)
    16'hfffe: cpu_bus = switches;
    16'hffff: cpu_bus = flags;
    16'b0????????????????: cpu_bus = ram_read;
    16'b1????????????????: cpu_bus =
  program_rom[cpu_ram_addr[14:0]];
  endcase
```

## 33.3 Improvements

One obvious change we could make is to read the sprite and tile bitmaps from RAM instead of ROM. This would eat up a lot more RAM bandwidth, so we might want to do this with dedicated video RAM. The Colecovision has this architecture, and the NES allows game cartridges to map tile patterns to RAM or ROM as needed.

We could also improve the 16-bit CPU. It does not have a very efficient instruction encoding. For example, calling a subroutine with a `MOV` and a `JSR` instruction takes three words, or six bytes – an operation that takes three bytes on the Z80 or 6502. We could design a more compact encoding for many common instructions, at the cost of some additional complexity.

# 34

# A Demo Program

To demonstrate our new 16-bit CPU-driven platform, we're going to write a simple test program.

To see it on `8bitworkshop.com`: Select the **Verilog** platform, then select the **CPU Platform** file.

We're using our homegrown assembler described in Chapter 27. The test program is *inline code*, that is, embedded right inside of the Verilog, just like our racing game in Chapter 28.

First we declare an *initial block* to populate our code ROM array, then use the `__asm` keyword to start an inline assembler block:

```
initial begin
  program_rom = '{
    __asm
```

We tell the assembler that we're using the 16-bit CPU, that our ROM starts at address 0x8000, and the length of our ROM:

```
.arch   femto16
.org    0x8000
.len    32768
```

The first thing we do is initialize the stack pointer, to ensure it points to valid RAM:

```
Start:
      mov       sp,@$6fff
```

(Note: In FEMTO-16 assembly language, 16-bit constants start with "@", and 8-bit constants start with "#".)

We'll call three routines in an infinite loop.

- `InitRowTable` – Fill the row table with pointers to each row of the tile map.
- `ClearTiles` – Fill the tile map with a constant value.
- `MoveSprites` – Move all sprites (which are initialized with random data on powerup).

Here's the assembly code:

```
      mov       dx,@InitRowTable
      jsr       dx              ; init row table
      mov       ax,@$4ffe       ; tile word
      mov       dx,@ClearTiles
      jsr       dx              ; clear tile map
      mov       dx,@MoveSprites
      jsr       dx              ; move sprites
      reset                     ; infinite loop
```

Then we initialize the page table. Our tile graphics will be a 32 x 32 grid. We fill the first entry with address `$6000`, then `$6020`, then `$6040`, 32 rows in total:

```
InitPageTable:
      mov       ax,@$6000       ; tile map start
      mov       bx,@$7e00       ; page table start
      mov       cx,#32          ; counter = 32 rows
InitPTLoop:
      mov       [bx],ax         ; write line address
      add       ax,#32          ; add 32 words
      inc       bx              ; go to next entry
      dec       cx              ; count down
      bnz       InitPTLoop      ; repeat until zero
      rts                       ; return
```

We also need to clear the tile map RAM. This is similar to the previous routine, except we just store a constant value, which is passed in `ax` at the start of the routine:

```
ClearTiles:
      mov       bx,@$6000       ; screen buffer
      mov       cx,@$3c0        ; 32*31 tiles
ClearLoop:
      mov       [bx],ax         ; write ax to tile map [bx]
      inc       bx              ; add 1 to bx
      dec       cx              ; decrement count
      bnz       ClearLoop       ; repeat until zero
      rts                       ; return
```

The `InitSprites` routine copies the default position and colors for all objects from ROM to sprite RAM. The first word of each entry is the X and Y position, the second word is the color info:

```
InitSprites:
        mov     bx,@$7f00       ; sprite data start
        mov     cx,#64          ; 4*16 sprites = 64 bytes
InitSLoop:
        mov     ax,[bx]         ; read from ROM
        add     ax,@$0101       ; add 1 to X and 1 to Y
        mov     [bx],ax         ; write to RAM
        inc     bx              ; next sprite address
        dec     cx              ; count down
        bnz     InitSLoop       ; repeat until zero
        rts                     ; return
```

## 34.1 Maze Game

We've also made the skeleton of a simple maze game in FEMTO-16 assembly language. It demonstrates graphics, reading from controllers, and more complex assembly code.

To see it on `8bitworkshop.com`: Select the **Verilog** platform, then select the **16-bit ASM Game** file.

# 35

# Practical Considerations for Real Hardware

Until now, we've been simulating Verilog programs in software. You might be thinking how to transfer them to an actual working circuit. Here are some thoughts on doing so.

## 35.1 FPGAs

An *FPGA* or *field-programmable gate array* is a configurable integrated circuit. They usually contain an array of *logic blocks* that can be wired together in different ways – making up the so-called *fabric* of the FPGA. Each logic block can be configured as a logic gate, or more generally a *LUT* (lookup table). As we saw in Chapter 1, any Boolean function can be converted to a truth table, and vice-versa. Modern FPGAs have between 2 and 6 inputs per logic block.

The LUTs take care of the combinational logic. For the sequential logic, FPGAs may include *flip-flops* as part of each logic block. The logic blocks can be configured to implement small registers, but many FPGAs often provide banks of *synchronous RAM* on the chip to free up logic blocks. Some FPGAs even contain CPU cores and DSP modules.

To *synthesize* a Verilog program and put it onto a FPGA, you need a compatible *toolchain*. This is often proprietary software supplied by the FPGA vendor, though open-source toolchains like Yosys and IceStorm also exist.

## 35.2 Video Output

The simulated CRT used in the 8bitworkshop IDE is intended to be similar to a NTSC television. But the software is limited; the simulator only runs at 4.857 MHz, and one pixel always equals one clock cycle.

To drive a real CRT, you'd need to follow the NTSC/PAL/VGA specifications (see https://www.amstzone.org/ntsc/ for reference.) Many early personal computers and game consoles ran at some multiple of the *colorburst* frequency, 3.5795 MHz, since that crystal was widely available.

A color NTSC signal is more difficult. Whereas arcade monitors just received individual red, green, and blue signals, NTSC was designed with a complex encoding for color that kept the signal compatible with earlier black-and-white TVs. Generating this signal is devilishly hard to get just right, but some hobbyists have made progress.[8]

VGA monitors receive separate color inputs, but their vertical resolution is double that of a NTSC TV. You'll either need to write your Verilog with the extra vertical lines in mind, or use a line-doubling buffer to convert a 262-line output to 524. Your circuit would also have to operate at 25 MHz or higher.

Anything more detailed than 1-bit-per-channel RGB color requires some kind of *DAC* (digital-analog converter). A DAC converts a multi-bit signal into an analog waveform. Many FPGAs provide these; external chips are also available. You can also create a simple DAC with a network of resistors.

An FPGA can also generate a *HDMI* output, with some limitations. HDMI is transmitted as 4 parallel digital signals with a special encoding to reduce electromagnetic interference. Generating a 640x480 signal requires 30 bits per pixel and a 250 MHz clock source, or 125 MHz with a DDR (Double Data Rate) register. Higher resolutions are considerably more difficult, and may require manually placing components on the FPGA fabric to satisfy timing constraints.

## 35.3 Audio Output

Audio output isn't terribly different from video – the audio signal is continuous and doesn't have scanlines, frames, or any timing constraints. You can easily output 1-bit audio by just piping a digital output to an amplifier, which was the approach taken by some early personal computers. Anything with more resolution requires a DAC. You could also use *pulse-width modulation*, creating a series of digital pulses that are filtered to approximate an analog waveform.

Some FPGAs also have audio codec hardware that can be driven from a I2C interface. Embedding audio into a TV or HDMI signal is yet another level of effort.

## 35.4 Controller Inputs

In our simulator, control inputs are simply bit values sent to a module. The real world is a bit more messy.

Generally, a switch input is an *asynchronous* input, meaning that it is not related to the clock signal. To avoid *metastability*, we want to synchronize these inputs with the clock signal. An easy way is to pass any external input through two cascading flip-flops before it enters the main part of the circuit.

If those flip-flops are clocked at a low frequency, say around 200 Hz, they would serve as a simple *debouncing* circuit. This prevents any spurious signals as the switch opens or closes from being interpreted as several button pushes. A debouncer could also be implemented in Verilog with a counter.

You might think about using modern USB or Bluetooth game controllers. USB is tricky enough that it usually requires a dedicated controller, and can't be driven directly from a FPGA. Then you'd have to figure out how to talk to the controller over its preferred protocol. This doesn't sound like much fun, which is why ancient game controllers are still useful for their comparatively simple interface.

## 35.5 Hacking an Arcade Cabinet

Many arcade cabinets from the 1990s follow the *JAMMA* standard, which is a wiring standard that handles RGB video, sound, and controller inputs. This is relatively simple to interface to a FPGA, or even a fast Raspberry PI. Even an old PC running DOS can configure its VGA controller to output compatible signals (I've done it!)

## 35.6 Building With Discrete Components

This option is for the purist, the masocist, and those who just really like the smell of molten solder. Hobbyists often create projects out of individual transistors and/or 7400-series ICs. This is a largely manual process, though one could use HDL tools to convert and optimize Verilog code into gates or even larger components.

To connect the components, you could either print a circuit board, or wire-wrap by hand. (For inspiration, check out the MOnSter 6502, a 12 x 15 inch replica of the 6502 CPU using individual transistors, or the Nibbler, a 4-bit CPU made from 7400-series ICs.)

# 36

# FPGA Example: iCEstick NTSC Video

This example uses the **iCE40HX-1K iCEstick**, a low-cost FPGA board that supports open-source tools. We'll modify the Starfield example so that it outputs NTSC video.

First, we install the **IceStorm** toolchain. (Refer to http://www.clifford.at/icestorm/ for more details.)

Next, we need a Physical Constraints File (`.pcf`). This defines the mapping between our top-level module's input and output ports and the physical FPGA pins. This PCF file just accepts a clock input, and outputs a composite video signal:

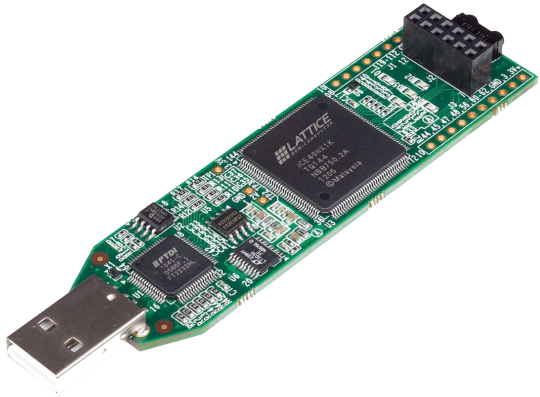


*Figure 36.1: iCEstick iCE40HX1K FPGA from Lattice Semiconductor.*

```
set_io clk 21
set_io out[0] 80
set_io out[1] 81
```

Note that the video `out` signal has two pins. This gives us four different levels for our signal. NTSC uses a range of 0-1 volts, and we need at least three levels (blank, black, and white) to generate a valid signal. We'll convert these two digital signals to an analog signal using a simple *DAC* (digital-analog converter) made out of a few resistors.[1]

For a real display device, we have to configure the video sync generator to output video at the right clock frequency. The FPGA runs at 12 MHz, but we need to output video at 6 MHz. So we divide the master clock by two:

```
reg clk2;
always @(posedge clk) begin
  clk2 <= !clk2;
end
```

We'll use `clk2` for all of our logic. At 6 MHz, there are about 381 clock cycles per scan line, so we modify the video sync generator's parameters thusly:

```
hvsync_generator #(256,60,40,25) // = 381 clocks/line
```

The final change is to change the 3-bit `rgb` signal to a 2-bit `out` signal. The output voltages are: blank (0 V, used during sync) black (0.3 V) grey (0.6 V) and white (1 V). We can combine the sync signals and convert RGB to greyscale like this:

```
assign out = (hsync||vsync) ? 0 : (1+g+(r|b));
```

The open-source **yosys** tool parses our Verilog source files and performs *synthesis*. The output of this process is a *BLIF* file, which contains a *netlist* that defines how the FPGA should be wired. For this FPGA, the logical components are 3-input

[1] http://www.rickard.gunee.com/projects/video/pic/howto.php

*LUT* (lookup table) cells, *D-type flip-flops*, *RAM*, and *carry propagators*.

```
yosys -p "synth_ice40 -blif starfield.blif" starfield.v
```

After this step, we need to map the BLIF file onto the target hardware. This step is called *place and route*, and we'll use the open-source **arachne-pnr** tool. Its output is a series of bitstreams, each of which configuring a particular *FPGA tile*. Each tile contains a number of LUT elements, RAM cells, or routing connections.

```
arachne-pnr -d 1k -p icestick.pcf starfield.blif -o
    starfield.asc
```

After the place and route step, the **icepack** tool creates a binary file ready to load into the target FPGA:

```
icepack starfield.asc starfield.bin
```

The final step is the **iceprog** tool, which loads the binary file into the FPGA:

```
iceprog starfield.bin
```

At this point, your FPGA should output a video signal displaying a scrolling starfield.

> There are some other examples and a Makefile in the 8bitworkshop git repository. For example, on Ubuntu Linux you can do this:
>
> ```
> sudo apt-get install yosys arachne-pnr
> git clone https://github.com/sehugg/fpga-examples
> cd fpga-examples/ice40
> make starfield.bin
> iceprog starfield.bin
> ```

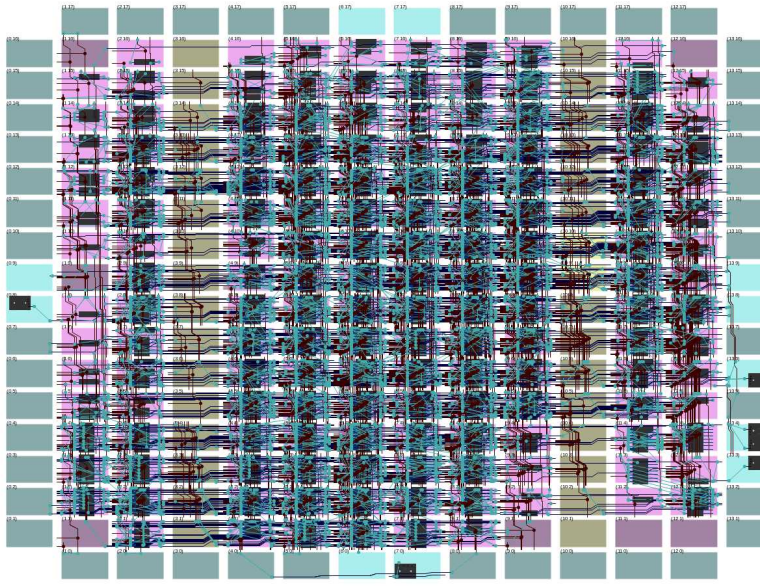


*Figure 36.2: Layout of FPGA after place and route step.*

# Appendix A: Verilog Reference

## Modules

```
// module definition
module binary_counter(clk, reset, outbits, flag, bus);
        input clk, reset;       // inputs
        output [3:0] outbits;   // outputs
        output reg flag;        // output register
        inout [7:0] bus;        // bi-directional
endmodule

// module instance
binary_counter bincount(.clk(clk), .reset(reset), ...);
```

## Literals

Format: `<bitwidth>'<signed?><radix><number>`

| | |
|---|---|
| Binary byte | `8'b10010110` |
| Decimal byte | `8'123` |
| Hexadecimal byte | `2'h1f` |
| 6-bit Octal | `2'071` |
| Unsized decimal | `123` |

## Registers, Nets, and Buses

```
reg bit;        // 1-bit register
reg x[7:0];     // 8-bit register
wire signal;    // 1-bit wire
wire y[15:0];   // 16-bit wire
```

## Bit Slices

```
x[0]            // first bit (rightmost, or least significant)
x[1]            // second bit
x[3:1]          // fourth, third, and second bits
{x[3], x[2], x[1]}      // concatenation, same as above
```

## Binary Operators

```
+       binary addition
-       binary subtraction
<<      shift left
>>      shift right
>       greater than
>=      greater than or equal to
<       less than
<=      less than or equal to
==      equals
!=      not equals
&       bitwise AND
^       bitwise XOR
|       bitwise OR
&&      logical AND
||      logical OR


*       multiply        // NOTE: these operators
/       divide          // don't always synthesize
\%      modulus         // very efficiently.
```

## Unary Operators

```
!       logical negation
~       bitwise negation
-       arithmetic negation
```

## Reduction Operators

These operators collapse several bits into one bit.

```
&       AND reduction (true if all bits are set)
|       OR reduction (true if any bits are set)
^       XOR reduction (true if odd number of bits set)
```

## Conditional Operator

```
        condition ? if_true_expr : if_false_expr
```

## If Statements

```
        if (condition) <expression> else <expression>;
```

## Always Blocks

Rules for usage:

| Keyword | Logic | Assign To | Operator |
|---|---|---|---|
| `always @(posedge clk)` | sequential | `reg` | <= |
| `always @(*)` | combinational | `wire` | = |
| `assign` | combinational | `wire` | = |

On rising clock edge:

```
always @(posedge clk) begin ... end
```

On rising clock edge or asynchronous reset:

```
always @(posedge clk or posedge reset) begin ... end
```

Non-blocking assignment (all right sides evaluated first):

```
variable <= <expression>;
```

Combinational logic (blocking assignment):

```
always @(*) ...

assign variable = <expression>;
```

## Miscellaneous

Include file:

```
`include "file.v"
```

Define macros:

```
// macro definitions
`define OP_INC 4'h2
`define I_COMPUTE(dest,op) { 2'b00, (dest), (op) }

// macro expansion (w/ parameters)
`I_COMPUTE(`DEST_A, `OP_ADD)
```

Define static array (e.g. ROM):

```
initial begin
        rom[0] = 8'7f;
        rom[1] = 8'1e;
        ...
end
```

Parameters:

```
localparam unchangable = 123;
parameter change_me = 234;
```

Case statement:

```
case (value)
        0: begin .. end;
        1: begin .. end;
endcase
```

Casez statement:

```
casez (opcode)
    8'b00??????: begin
      state <= S_COMPUTE;
    end
endcase
```

Log an error to console:

```
if (value == 0) $error("unexpected value");
```

Print to console (be careful, this can overflow your web browser):

```
$display("value =", value);
```

Driving a tri-state bus using a high-impedance (Z) value:

```
assign data = we ? 8'bz : mem[addr];
```

Functions:

```
 function signed [3:0] sin_16x4;
```

```
        ...
endfunction
```

`for` loops (for replicating logic or filling static arrays):

```
for (int i=0; i<16; i++) begin
        ...
end
```

# Appendix B: Troubleshooting

**UNOPTFLAT: Signal optimizable: Feedback to clock or circular logic**

This is a tricky condition to diagnose. It means that there is a feedback loop somewhere in your code. This happens with use of *combinational* code (`assign`) and/or triggering on derived clocks.

You can often fix this problem by converting `assign` statements to synchronous `always` blocks, breaking up the loop with a register. Keep in mind that this may introduce additional clock cycle delays in your design.

The compiler can't rule out loops at the bit level, so if you're using something like a flags register, you may have to split it up into separate variables if you get this warning.

**MULTIDRIVEN: Signal has multiple driving blocks**

This is a warning that a signal is being driven from two separate `always` blocks. For example, don't do this:

```
always @(posedge reset)
  xpos <= 0;

always @(posedge vsync)
  xpos <= xpos + 1;
```

Do this instead:

```
always @(posedge vsync or posedge reset)
  if (reset)
    xpos <= 0;
  else
    xpos <= xpos + 1;
```

**WIDTH: Operator ... expects N bits ...**

You'll get this when trying to perform math or logic operations on values of different bit widths. For example:

```
reg       x = 1; // single bit
reg [7:0] y = 1; // 8 bits

always @(posedge clk)
  y <= y + x;         // can't add 1 bit to 8 bits
```

This could be fixed by upcasting the 1 bit value to 8 bits:

```
  y <= y + 8'(x);
```

You could also use the `signed` function to first convert an unsigned value to signed before upcasting; this would perform *sign-extension*.

**CASEINCOMPLETE, CASEOVERLAP**

Make sure your `case` statements cover all of the bit patterns you want to match, and make sure the bit patterns don't overlap. If you add a `default` statement, you don't have to cover all of them.

If you're sure what you're doing, you could also add a *pragma* to disable the warnings:

```
/* verilator lint_off CASEOVERLAP */
/* verilator lint_off CASEINCOMPLETE */
```

**Don't use > 32 bits**

Our Javascript version of the Verilog simulator doesn't support bit vectors wider than 32 bits, so don't use them.

## Metastability

Metastability means the value of one or more flip-flops are not stable at a clock edge. This causes weird things to happen in your circuit.

This condition doesn't apply to the simulator, but in real hardware this condition can propagate and be hard to diagnose. To avoid it, follow these guidelines.

Trigger events on the main clock, or on derived clocks that have a simple phase relationship, e.g. the output of a clock divider.

Don't "gate the clock" – in other words, don't pass a clock signal through logic gates, because race conditions may lead to multiple clock pulses.

Don't run your clock too fast, or have too many logic gates between registers. This may violate the setup or hold time constraints of your flip-flops.

If you use asynchronous inputs (switches, buttons, reset, etc) make sure to clean the signal through two flip-flops before it enters the main circuit. Do the same when crossing *clock domains* – transferring data clocked at a different rate than your main clock.

# Bibliography

[1] Charles Sanders Pierce. Logical Machines, 1887.

[2] William Arkush. *The Textbook of Video Game Logic, volume I*. 1976.

[3] Tristan Donovan. *Replay: The History of Video Games*. 2010.

[4] Steven L. Kent. *The Ultimate History of Video Games*. 2010.

[5] Dr. H. Holden. ATARI PONG E CIRCUIT ANALYSIS & LAWN TENNIS: BUILDING A DIGITAL VIDEO GAME WITH 74 SERIES TTL IC's. http://www.pong-story.com/.

[6] The Dot Eaters. Gun Fight and Nutting Associates - Hired Guns. http://thedoteaters.com/.

[7] Simon Parkin. The Space Invader. *The New Yorker*, 2013.

[8] Juha Turunen. NTSC-composite-encoder. https://github.com/elpuri/NTSC-composite-encoder.

Some content is licensed under a Creative Commons license:

https://creativecommons.org/licenses/

# Index