- Transaction-level communication between objects
- Testbench stimulus (UVM Sequences) separated from the testbench structure

By the end of the book you will have a firm grasp of each of these concepts and how they contribute to your testbenches.

Online Components

The *UVM Primer* discusses the UVM through code. This presents a challenge to the reader and author in terms of the detailed code discussion. On the one hand, a line-by-line description of the code is boring. On the other hand, a high-level discussion of a code snippet's behavior can leave some readers scratching their heads as to how the code implements what is promised. These details should not be left as an exercise to the reader.

I've addressed this problem by supplementing high-level discussions of code behavior in the book with detailed videos available through www.uvmprimer.com. A video is available for each chapter's code example.

People reading a primer are bound to have additional questions about the topic. You can ask these questions on the *UVM Primer* Facebook page. This is a central location for readers to discuss the book's concepts and code examples.

You can download the code examples from www.uvmprimer.com either as a gzipped tar file or as a pull from a GIT repository on www.git-hub.com.

Example Design

We will examine the UVM by verifying a simple design: the TinyALU. By doing this, we can focus our energy on the testbench without getting distracted by DUT complexity.

The TinyALU is a simple ALU written in VHDL. It accepts two eight-bit numbers (A and B) and produces a 16-bit result. Here is the top level of the TinyALU[2]:

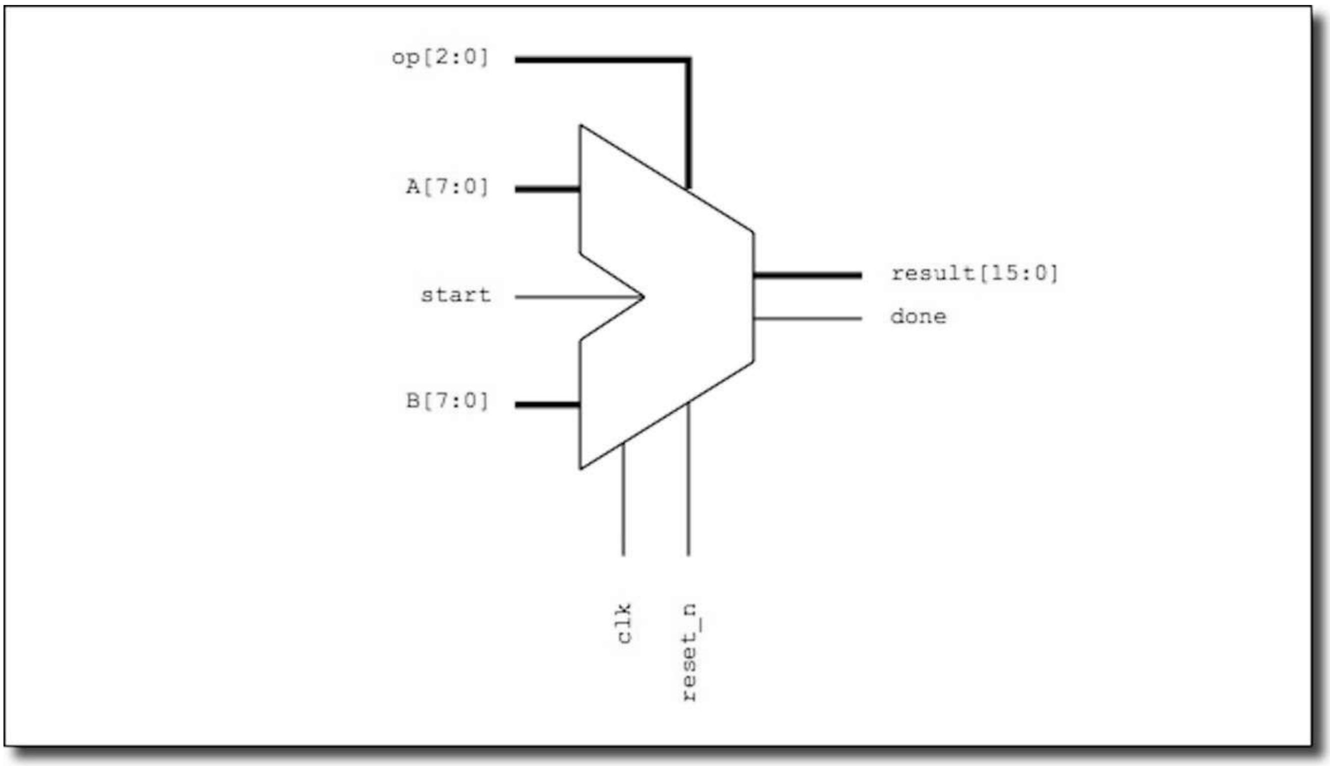
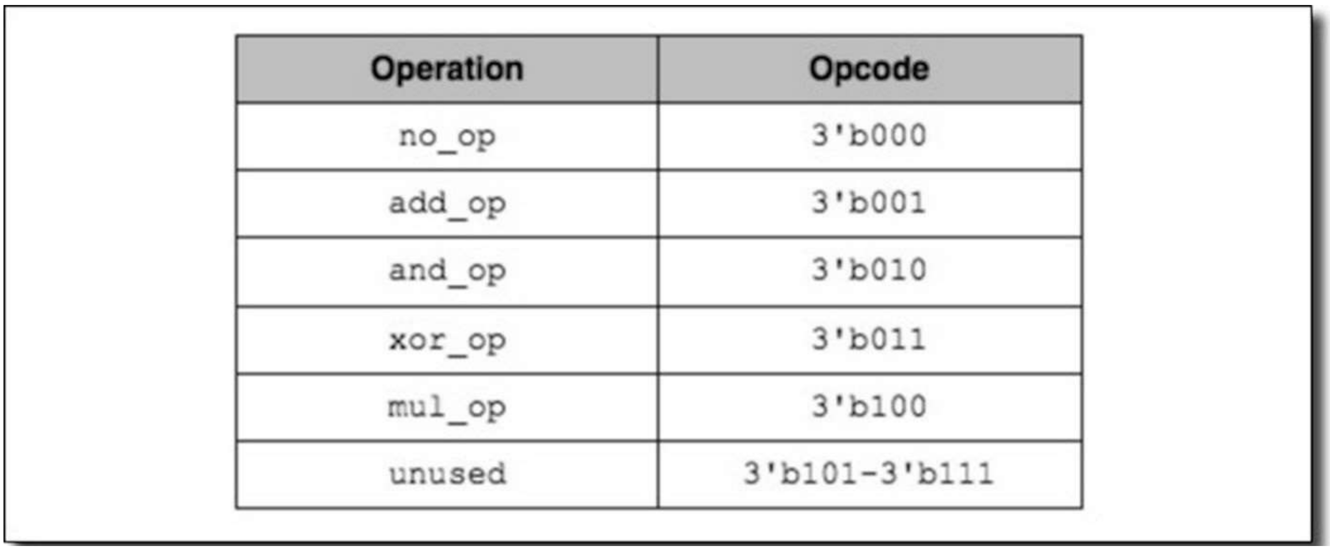


Figure 1: TinyALU Block Diagram

The ALU works at the rising edge of the clock. When the `start` signal is active, the TinyALU reads operands off the `A` and `B` busses and an operation off the `op` bus, and delivers the result based on the operation. Operations can take any number of cycles. The TinyALU raises the `done` signal when the operation is complete.

The `reset_n` signal is an active-low, synchronous reset.

The TinyALU has five operations: NOP, ADD, AND, XOR, and MULT. The user encodes the operations on the three bit `op` bus when requesting a calculation. Here are the encodings:



Operation	Opcode
no_op	3'b000
add_op	3'b001
and_op	3'b010
xor_op	3'b011
mul_op	3'b100
unused	3'b101-3'b111

Figure 2: TinyALU Operations and Opcodes

Here is the waveform for the TinyALU:

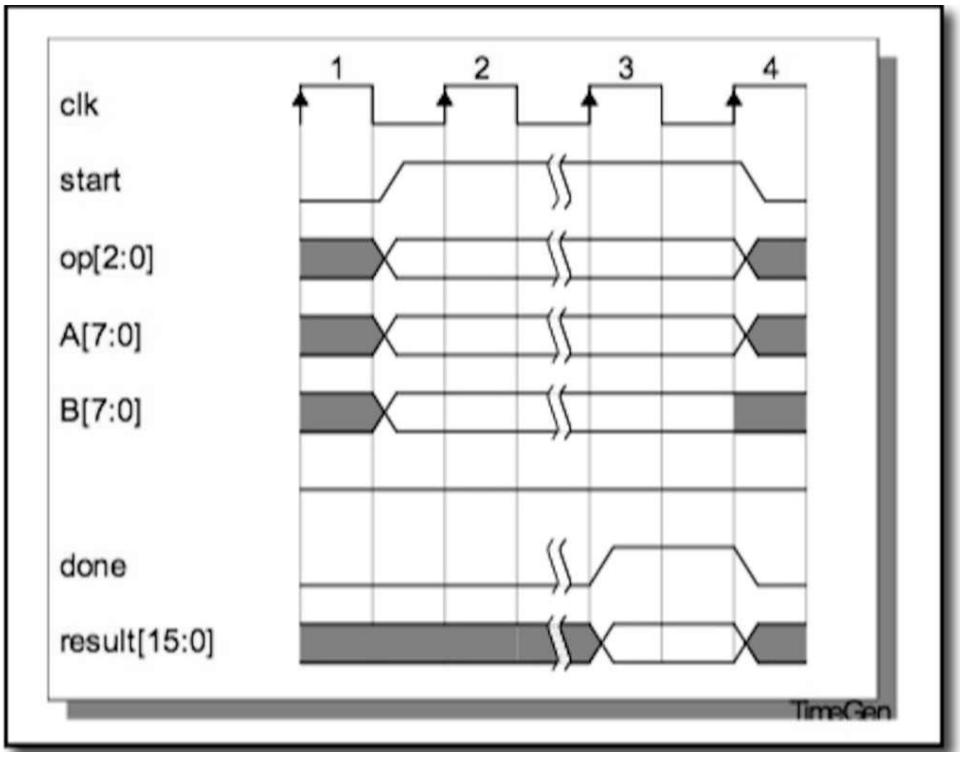


Figure 3: TinyALU Protocol Waveform

The start signal must remain high, and the operator and operands must remain stable until the TinyALU raises the `done` signal. The `done` signal stays high for only one clock. There is no `done` signal on a `NOP` operation. On a `NOP`, the requester lowers the `start` signal one cycle after raising it.

We'll start our journey through the UVM by creating a conventional testbench of the TinyALU. Then each chapter will modify the testbench to make it UVM compliant. We'll discuss the advantages of using the UVM as we transform the testbench.

A Note on the Code Font

Talking about code forces us to mention variable names. Since these can be confusing in a sentence (e.g., "waiting for start to go high"), I will use `this code font` to identify variables and SystemVerilog keywords (e.g., "waiting for `start` to go high").