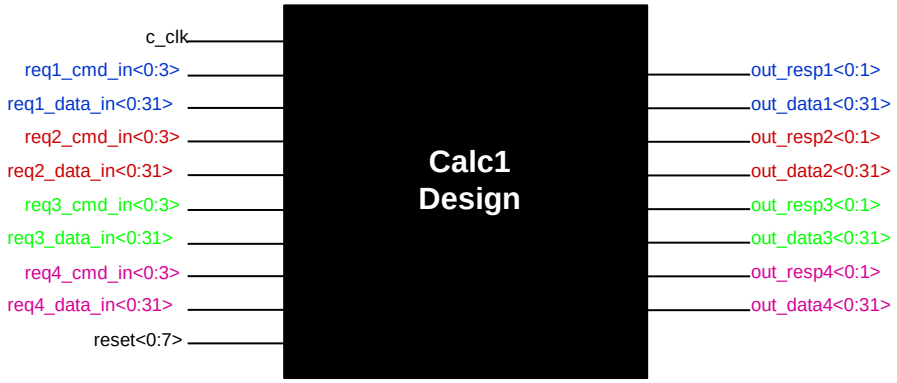
4.3.1 Calc1 Design Description

Calc1 is an RTL design implementation of a four-operation calculator. The four operators are add, subtract, shift-left, and shift-right. A "requestor" may send an operator into the calculator on the command input bus, accompanied by operand data. Each "requestor" uses one of four input "ports" to send the command and operand data. The four ports can each handle a single command in parallel.

Each command will receive a response from the calculator design. Except in the case of an error condition, the response will include the result of the operation. This section describes the exact protocols.

Figure 4.6 shows the input and output wires for Calc1. As with all designs, a clock signal input is supplied to the RTL. For Calc1, the clock signal is called c_clk.

Figure: 4.6: Input/Output description of Calculator design 1



Each of the four Calc1 ports has two separate input busses and two output busses. The first input bus, reqX_cmd_in(0:3) (where X is replaced by port numbers 1, 2, 3, or 4) is a 4 bit bus used to transmit the command to the Calc design. The command and decode values for the command bus are shown in table 4.1:

Table 4.1: Calc1 command decode values

Command	Decode value
No operation	'0000'b
Add	'0001'b
Subtract	'0010'b

Shift left	‘0101’b
Shift right	‘0110’b
Invalid	All others

The second input bus, reqX_data_in(0:31), is the operand data bus. The requestor ports send operand 1 data and operand 2 data on sequential cycles, with operand 1 data concurrent with the command. Therefore, it takes two cycles to send a complete command and data sequence.
Table 4.2 shows how the Calc1 design operates on the two operands.

Table 4.2 Calc1 operation details

Command	Effect on operands
Add	Result is operand1 + operand2
Subtract	Result is operand1 – operand2
Shift left	Result is operand1 shifted left operand2 places*. Bits shifted out are dropped. Zeros are always shifted in.
Shift right	Result is operand1 shifted right operand2 places*. Bits shifted out are dropped. Zeros are always shifted in.

*For both shift commands, only the low order 5 bits (reqX_data_in(27:31)) of operand2 (the shift amount) are used. The Calc1 logic ignores bits 0 to 26 of the shift operand2. This allows the operand1 data to be shifted any amount from 0 to 31 places (inclusive).

The two output lines for each port are the response bus (out_respX(0:1)) and result data bus (out_dataX(0:31)). The response bus goes active for one cycle when the Calc1 design completes the computation for the port. The number of cycles that it takes to complete an operation is dependent upon the amount of activity on the three other ports, but will always be at least three cycles. Table 4.3 shows the possible responses for a given operation:

Table 4.3: Calc1 response values

Response decode	Response meaning
‘00’b	No response on this cycle
‘01’b	Successful response. Response data is on the output data bus.
‘10’b	Overflow, underflow, or invalid command. Overflow/underflow only valid for the add or subtract commands.
‘11’b	Unused response value

The output data bus (out_dataX(0:31)) should only be sampled when out_respX(0:1) contains the successful response decode value (‘01’b). At that time, the value on the output bus will contain the result of the operation for that port.

Figure 4.7 shows a timing diagram of the command and response sequence for a single successful command on port 1. The command and first operand of data are sent in the first cycle of the sequence. The second operand data follows on the second cycle. A few

cycles pass, and the response appears on the output of the design, accompanied by the result data.

Figure 4.7: Input/Output Timing of Calculator design 1

Each port must wait for its response prior to sending the next command!

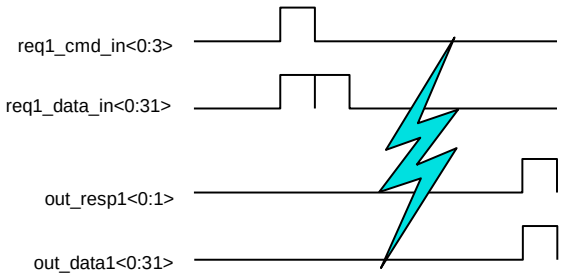


Figure 4.7: A timing diagram of a single port command sequence

Each port may have one operation ongoing at a time. Once a port sends a command, it may not send another command until a response has been received for the preceding command. The protocols do not require that a requestor port send a new command whenever the preceding command completes. The port may be idle for any number of cycles in between commands.

Each port is independent of the others. All four ports may send commands concurrently or any combination of commands across cycles (with the stated restriction of only one outstanding command per port). Therefore, at any given point, the Calc1 design may be working on any number of commands up to a maximum of four.

If all four ports send commands concurrently, the responses will not be concurrent. While each port has equal priority, there are limited resources inside the design. Specifically, there is one ALU for adds and subtracts, and a second ALU for shift commands. Hence, if all four ports sent concurrent add commands, the Calc1 logic would serialize the responses, as only one add command could be processed by the ALU at a time.

Internal to the design is a priority logic scheme that sends commands to one ALU or the other, depending on the command decode. Commands are serviced on a first-come, first-serve basis. Commands that arrive on the same cycle may be serviced in any order.

The design has a reset bus input used to clear the internal state of the design. During verification, the test case initially should activate the reset to put the design in a cleared state. Setting the reset line, reset(0:7), to '11111111'b activates the reset. This input value needs to be held for seven consecutive cycles in order for the reset to propagate through the design. All other input busses except the c_clk[1] should be set to zero while resetting the Calc1 design.

Calc1 treat arithmetic operands as unsigned data. The most significant bit, bit 0, is a data bit, not a sign bit. An overflow occurs on an add operation when the high order bit (bit 0) has a carry-out. An underflow occurs on a subtract operation when a larger number is subtracted from a smaller number. Table 2.5 shows examples:

Table 2.5: Add and subtract overflow/underflow and successful response examples

Command	Operand1	Operand2	Response	Result data
Add	"80002345"X	"00010000"X	Successful	"80012345"X
Add	"FFFFFFFF"X	"00000001"X	Overflow	None
Subtract	"FFFFFFFF"X	"11111111"X	Successful	"EEEEEEEE"X
Subtract	"11111111"X	"20000000"X	Underflow	None

4.3.2 Creating the Test Plan for Calc1

Now that the specification is in place, it is time to create the test plan for the Calc1 design.
Even for a relatively simple design like Calc1, it is still best not to jump into test case writing before thinking through the entire test plan requirements.

The above design description details the intent of the Calc1 design. Now, the verification engineer must prove that the actual design implementation matches the intent. Therefore, none of the above Calc1 design details should be assumed to be correct in the implementation.

4.3.2.1 The Calc1 Verification Plan

Description of Verification Levels

Calc1 is a simple design used as an initial introduction to simulation based verification (however, verification engineers have successfully applied formal verification to the design as well). Therefore, this design requires verification only at the top level of the DUV hierarchy. Furthermore, the available specification only describes the top-level

[1] The test case must drive the c_clk. However, the value of the c_clk input is dependent upon the type of simulation engine. For event simulation, the c_clk must be toggled every cycle. For cycle simulation, the c_clk should be stuck to '1'b.

interfaces. Verification at a lower level of hierarchy, such as the ALUs, would require an input and output description of that sub-unit.

However, if the people resources exist, it is better to do unit level verification. This would place the priority logic and ALUs under a microscope of verification, allowing a higher level of control and stress on the design. Furthermore, in real world designs, it is common for one designer's logic to be available prior to others' logic. If the priority logic HDL is ready for verification before the ALUs, the priority logic unit level verification can commence without waiting for the entire chip. This level of verification would add a dependency on the design team to document the unit interfaces.

Required Tools

The tools inventory for Calc1 comprises a single software simulation engine (and license to run it) and one workstation, a waveform viewer, and a test case language or infrastructure. The language or infrastructure must communicate with the simulation engine through the engine's Application Programming Interface (API). The API provides the means to drive the inputs, check the outputs, and clock the model of the design, which is simulated by the engine itself. [2]

Risks and Dependencies

This exercise does not have risks worthy of documentation. In a larger project with a more complex design, there might be risks inherent in the delivery of the specification of esoteric operations (e.g. binary floating point) or in the ability to verify the design in a short schedule. If Calc1 were part of a large system, this section would detail schedule integration factors. The system level verification process has a dependency on the correctness of the Calc1 function so that broader system integration verification can occur without being concerned with the quality of the add, subtract, or shift functions.

The Calc1 exercise does have a dependency on the delivery of the specification and availability of the required tools. The specification has been delivered (above). To perform this exercise, it is assumed that you have the required tools.

Functions to be Verified and Test Scenarios (Matrix)

Because Calc1 is simple, we have combined the *Functions to Be Verified* and the *Testcase Scenarios (Matrix)* sections of the verification plan. The details are listed here.

[2] Basic tools such as text editors (to write test cases) need not be included in the Required Tools section of the verification plan.

1. Certain test case requirements jump out at the verification engineer. From the Calc1 design description, it is clear that the following functions must be verified:
   1.1. Check the basic command-response protocol on each of the four ports
   1.2. Check the basic operation of each command on each port
   1.3. Check overflow and underflow cases for add and subtract commands

2. Beyond these basic functions are a series of tests that involve scenarios that are more complex. These scenarios include:
   2.1. Check that each command can have any command follow it without leaving the state of the design dirty, such that the following command is corrupted.
      2.1.1. Check for each port
      2.1.2. Check across all ports (e.g. four concurrent adds don't interfere with each other)
   2.2. Check that there is fairness across all four ports such that no port has higher priority than the others
   2.3. Check that the high order 27 bits are ignored in the second operand of both shift commands
   2.4. Check data dependent corner cases of each type of command:
      2.4.1. Add two numbers that overflow by 1 ("FFFFFFFF"X + 1)
      2.4.2. Add two numbers whose sum is "FFFFFFFF"X
      2.4.3. Subtract two equal numbers
      2.4.4. Subtract a number that underflows by 1 (operand 2 is one greater than operand 1)
      2.4.5. Shift 0 places (should return operand 1 unchanged)
      2.4.6. Shift 31 places (the max allowable shift places)
   2.5. Check that the design ignores data inputs unless the data is supposed to be valid (concurrent with the command, and the following cycle). Remember that "00000000"X is a data value just as any other 32 bit combination. Here, the check must include verifying that the design latches the data only when appropriate, and does not key off non-zero data.
3. Finally, there are generic tests and checks that are applicable to all verification plans:
   3.1. Check that the design correctly handles illegal commands
   3.2. Check all outputs all of the time. Calc1 should not generate superfluous output values.
   3.3. Check that the reset function correctly resets the design

Specific Tests and Methods (Environment)

Type of Verification (Black box, White box, Grey box)

All of the functions listed under the *Functions to be Verified* section can be created by driving the chip inputs, and most can be checked by monitoring the chip outputs. This would indicate that block box checking is adequate. However, checkers placed on certain

logic functions internal to the design might flag logic flaws faster. These checkers include:

- Checks on internal queues or stacks, especially in the priority logic. This would verify that no command leaves the state of the machine dirty (item 2.1 in the list of functions to be verified).
- Checks that verify fairness in the priority logic by monitoring the dispatch of commands to the ALUs (item 2.2 in the list of functions to be verified)

These checkers indicate that Grey box verification is appropriate. Therefore, all stimulus will be driven on the chip inputs, while most checkers will monitor the chip outputs. A limited number of checkers will be place on internal logic.

Verification Strategy

Deterministic, random, and formal verification are all viable technologies for Calc1. However, formal verification could have trouble verifying correct ALU results across all 32 bits simultaneously. And a full-blown random environment might be overkill for this simple design, as the number of test cases required to verify the functions is limited. Therefore, for the Calc1 exercise, we choose the deterministic verification method.[3]

Randomization Controls

Given the deterministic test case method, questions such as randomization controls become moot. The test case writer will encode the input values and the expected outputs into the test case itself.

Abstraction Level

The level of abstraction depends on the test case language. If the infrastructure exists, driving the Calc1 at the packet level is optimal for quickly coding test cases. A packet level syntax might look like this:

Test case 4.1:

/* Port number	Cmd	Operand1	Operand2	Result	Response*/
Port1	ADD	“00012345”X	“00054321”X	“00066666”X	Good
Port2	SHL	“22222222”X	“00000002”X	“88888888”X	Good
Port1	SUB	“00000001”X	“00000003”X	“00000000”X	Underflow

This syntax is very convenient, as the infrastructure that reads and manages the test case handles many mundane tasks. This allows the test case writer to focus on the intent of

[3] As we advance to the Calc2 exercise in Chapter 4, the number of permutations and the level of complexity quickly exceed the number of deterministic tests that a verification team can reasonably write. Therefore, we leave the example of random based methods to Calc2.

the testcase. In the above syntax, the infrastructure environment manages the following tasks for the testcase writer:

- Translating the actual command decode values to bits (e.g. ADD = "0001"b). This has a potentially huge added benefit: if the decode value of the ADD is ever changed (e.g. ADD = "1001"b), all that's required is a simple modification in the infrastructure. If the decode values are coded in the testcase itself, then every testcase with the ADD command must be changed. The same is true for the encoding of the response value.
- Driving the bit level values into the Calc1 design. The infrastructure knows to place the operand1 data value on the bus concurrent with the command and to send operand2 data the next cycle.
- Waiting for the valid response. The test case writer cannot predict when the response will come back, so the infrastructure waits for the response event and checks the value when it finally appears on the outputs.
- Sending in the next command to the port when it is available. In the above example test case, there are two commands destined for Port 1. The protocol states that only one command may be outstanding at a time on a given port. Therefore, the infrastructure can send the first two commands concurrently, but must wait for the ADD on Port 1 to finish before initiating the SUB. Without an infrastructure, this is a very tedious task involving trial and error.
- Resetting the logic. The infrastructure automatically raises the reset line for seven cycles as required.
- Driving the clocks. The infrastructure manages the setting of the c_clk.

An even more robust infrastructure might further simplify the test case writing task by including:

- Data prediction. If the infrastructure included a golden model that predicts the result and response fields, it frees the test case writer from having to "do the math".
- Random Operand Data. If desired, an operand data value might be replaced with the keyword "Random." This would tell the packet generator to pick a random number value. This requires data prediction capabilities in the infrastructure.[4] The testcase writer maintains deterministic control over the command selection and flow of the testcase, but optionally can allow the infrastructure to choose values.

When compared to the above packet level of abstraction, the following bit-stream level test case language illustrates the power of a robust infrastructure:

Test case 4.2:

```
SET INPUT "reset(0:7)" "11111111"b;
CLOCK 7;
```

[4] This type of random data generation should not be confused with "Random Based Verification methodologies" as described in Chapter 4. Random based verification go well beyond random data by using biasing tables to randomly select commands and ports, as well as idle cycles between port commands.

```
SET INPUT “reset(0:7)” “00000000”b;
CLOCK 1;
SET INPUT “req1_cmd_in(0:3)” “0001”b;
SET INPUT  “req1_data_in(0:31)” “00012345”x;
...
CLOCK 3;
EXPECT  OUTPUT “out_resp1(0:1)” “01”b;
…
```

This bit-stream level of abstraction is very tedious for a verification engineer to write test cases. The above test case sample does not even complete the sending of a single Port command. Even worse, the writer must predict when the response will return from Calc1 through trial and error. This entails adjusting the clock value in line 8 to the actual cycle that the Calc1 logic returns the response.

Output Checking

The above section indicates the need for the Golden Vector approach to checking. The verification engineer supplies the expected output values in these deterministic sequences.

Coverage Requirements

Coverage goals for the Calc1 example are based on the tests defined in the *Functions to be Verified* section. The goals require that the verification tests create all of the cases described in the *Functions to be Verified* section. These goals are simple because we are able to list and articulate the test scenarios in this design. However, in robust designs, coverage goals require effort and rigor in the verification process.

While there is a detailed explanation of coverage and coverage tools in Chapter 6, the fundamental notion of coverage is to provide feedback and confirmation of what has been exercised in the design. Coverage feedback provides the verification engineer with insight into the actual data of what the testcase has done. This data will show either that the intent of the test case has been met, or that the test case failed to exercise the intended function within the design. For example, in the *Functions to be Verified* section of the test plan, item 2.1.1 calls for tests that verify that each command can be followed by any other command on each port. This statement calls for a series of 16 pairs of commands on each port for a total of 64 different test scenarios. Coverage feedback would keep track of which of the 64 sequences were completed.

Resource Requirements

For the Calc1 exercise, the Resource Requirements call for a single verification engineer. As mentioned above, the compute resources call for a single workstation on which the simulation engine runs.

Schedule Details

The schedule details for the Calc1 exercise are straightforward. A verification engineer should expect to complete the Calc1 design example in a single workday.

In the case of the packet level abstraction, the Calc1 verification engineer has a dependency on the existence and quality of the infrastructure. Usually, the verification engineer must create the infrastructure, or at least personalize it for the design under test. If this is the case, extra schedule time is required.

4.3.3 Deterministic Verification of Calc1

With the design intent and the verification plan in place, it is time to begin verifying the Calc1 design. The verification plan called for deterministic testing of the design. For clarity, the following description will use a packet level of abstraction to detail the deterministic tests. As stated in the verification plan, the packet level requires an infrastructure to be in place.

Figure 4.8: Packet Level Infrastructure

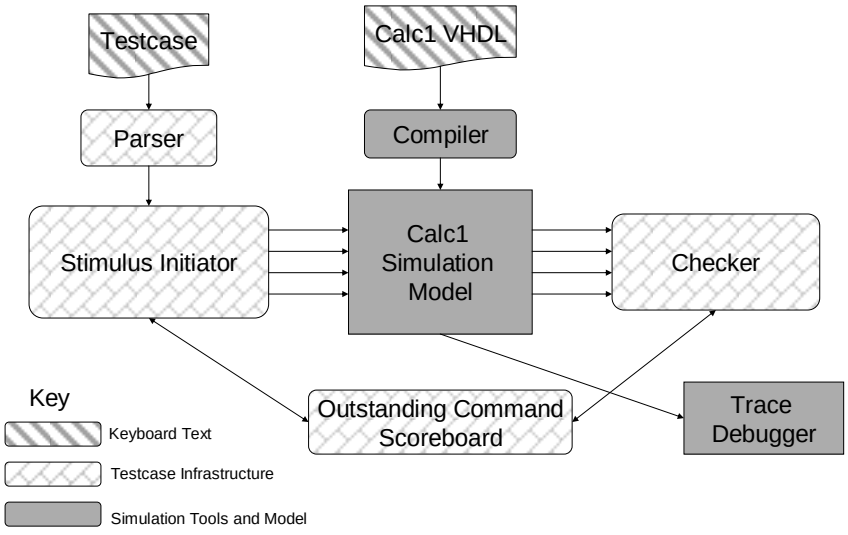


Figure 4.8: Infrastructure for sending simple command packets into the Calc1 simulation model.

Figure 4.8 shows the high-level view of the components in the infrastructure. The test case parser reads the text-based commands from the test case and converts them into packet structures for the stimulus initiator. The initiator passes the packets into the model on the designated port, performing the packet to bit-stream conversion. Each command packet requires two cycles to transmit into the Calc1 simulation model as dictated by the specification. The stimulus initiator can multiplex across all four ports, sending simultaneous or staggered commands based on the test case. When the interface driver transmits a command packet, it also posts the command to a scoreboard, which keeps track of commands currently under execution by the Calc1 simulation model. As the Calc1 simulation model completes commands, the output checker pulls the expected response and expected data from the scoreboard for comparison to the actual response. If there is a response or data mismatch, the test case execution ceases and the output checker records the error and the actual vs. expected data. Otherwise, the output checker removes the command from the scoreboard, indicating that the stimulus initiator may dispatch a new packet to that port. When the stimulus initiator transmits all of the test case packets and the checker receives all responses, the test case ends successfully.

The infrastructure also takes care of the seven-cycle reset sequence as well as driving the clocks into the model.

For this exercise, we will build upon the test case syntax introduced in test case 4.1, described in the above Verification Test Plan section. The new syntax includes an additional field, DelayN, to control the number of cycles between packets.

Test case 4.3:

/*	Port #	DelayN	Cmd	Operand1	Operand2	Result	Response*/
	Port1	0	ADD	“00012345”X	“00054321”X	“00066666”X	Good
	Port2	2	SHL	“22222222”X	“00000002”X	“88888888”X	Good
	Port1	3	SUB	“00000001”X	“00000003”X	“00000000”X	Underflow

The delay field causes the interface driver to wait N cycles before dispatching the packet. In the above example, the first packet on Port1 (ADD command) will dispatch as soon as the reset completes (cycle 8), and the Port2 command will be initiated two cycles later. When the Port1 command completes, the port will remain idle for 3 cycles before the interface driver initiates the SUB command packet.

With this packet-level infrastructure in place, the verification engineer focuses on the intent of each test case. We can now write the first set of test cases in the plan.

Verification plan tests 1.1 to 1.3

1. Certain test case requirements jump out at the verification engineer. From the Calc1 design description, it is clear that the following functions must be verified:
   1.1. Check the basic command-response protocol on each of the four ports
   1.2. Check the basic operation of each command on each port
   1.3. Check overflow and underflow cases for add and subtract commands

/* Test case 1.1.1 Basic command-response protocol on Port 1 */

/*	Port #	DelayN	Cmd	Operand1	Operand2	Result	Response*/
	Port1	0	ADD	“00000005”X	“00000008”X	“0000000D”X	Good

Our first test case runs to completion and the simulation engine captures the trace. The response and data are correct and test case 1.1.1 is successful. Figure 4.9 shows the trace of test case 1.1.1. Test cases 1.1.2, 1.1.3, and 1.1.4 verify the basic command-response flow of each port. However, test case 1.1.4 does not complete in 13 cycles, as the others did. Instead, it runs until the test times-out. Figure 4.10 shows the trace for Test Case 1.1.4.

Figure 4.9: Test Case 1.1.1 trace

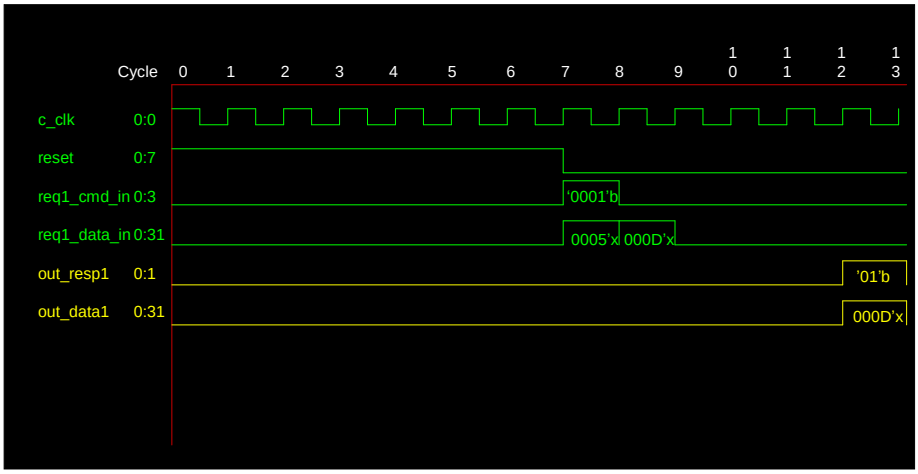
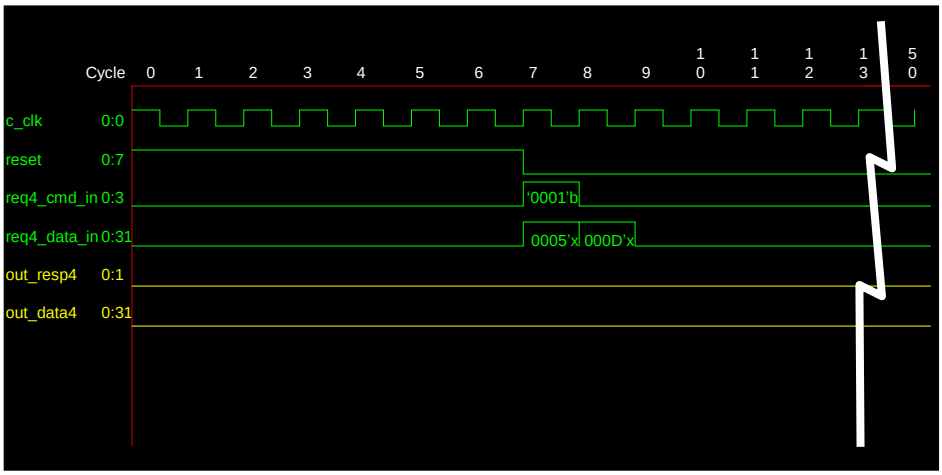


Figure 4.10: Test Case 1.1.4 trace



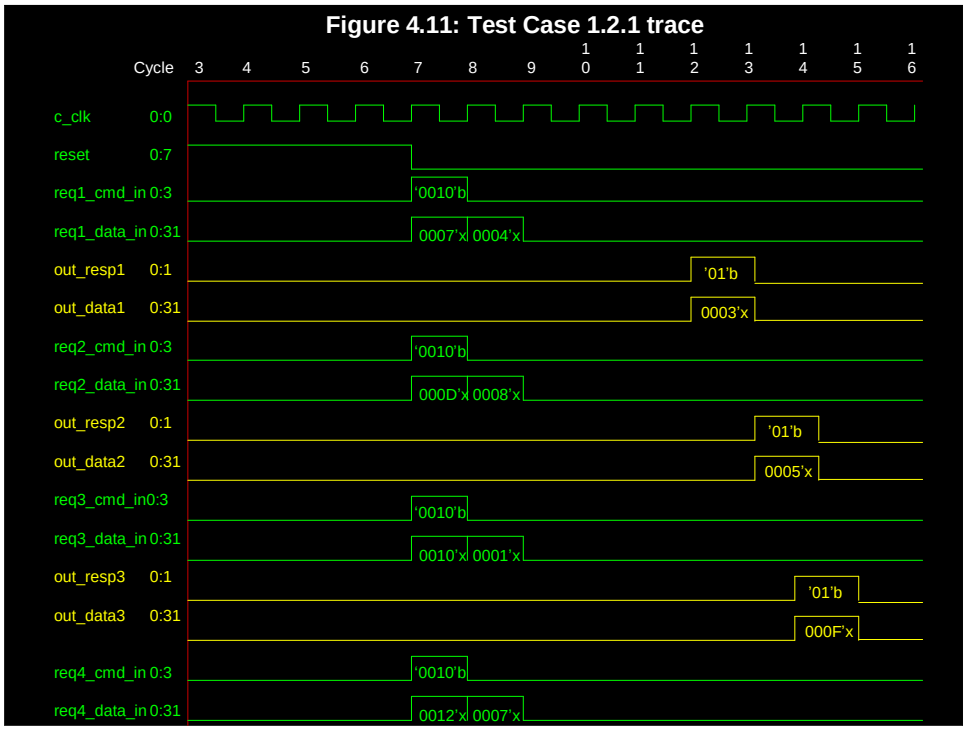
The out_resp4 wire never returns a valid response. At this point, the verification engineer has potentially found a bug where port4 hangs. After further study of the trace, the verification engineer confers with the designer and the designer concludes that the priority logic does have a bug. The designer makes a fix and recompiles the model. The verification engineer re-runs the exact same test case to validate the fix, and concludes that the basic command-response sequence on port4 now works. For good measure, the verification engineer re-runs the first three test cases to ensure that the fix did not break any other logic that had been working. The first bug in Calc1 has been uncovered and fixed!

Section 1.2 of the verification plan prescribes test cases for each operation on each port. Test cases 1.1.1 – 1.1.4 already verified the add operation on each port, so the new test case verifies the subtract operation on each port in parallel.

/* Test case 1.2.1 SUB commands on each port */

/* Port #	DelayN	Cmd	Operand1	Operand2	Result	Response*/
Port1	0	SUB	“00000007”X	“00000004”X	“00000003”X	Good
Port2	0	SUB	“0000000D”X	“00000008”X	“00000005”X	Good
Port3	0	SUB	“00000010”X	“00000001”X	“0000000F”X	Good
Port4	0	SUB	“00000012”X	“00000007”X	“0000000B”X	Good

The test case runs to completion in 16 cycles. Figure 4.11 shows the trace.



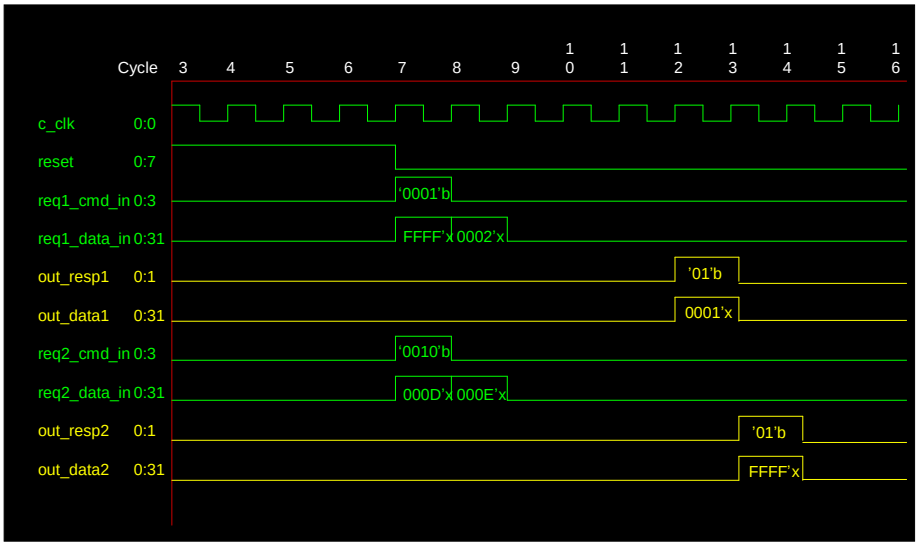
Next, test cases 1.2.2 and 1.2.3 verify the shift left and shift right operations on each port. Both test cases run successfully against the Calc1 model.

Section 1.3 of the verification test plan calls for add and subtract overflows and underflows.

/* Test case 1.3.1 Overflow and underflow							*/
/*	Port #	DelayN	Cmd	Operand1	Operand2	Result	Response*/
	Port1	0	ADD	“FFFFFFFF”X	“00000002”X	“00000000”X	Overflow
	Port2	0	SUB	“0000000D”X	“0000000E”X	“00000000”X	Underflow

When test case 1.3.1 runs against the Calc1 model, the test case fails with miscompares on both the result and response values. Figure 4.12 shows the trace.

Figure 4.12: Test Case 1.3.1 trace



Rather than responding with the overflow/underflow response of ‘10’b, the Calc1 model returns a good response. Further examination reveals that the results for both the add and subtract would be correct in 2’s complement, but that does not match the specification. The designer is consulted and the second bug is confirmed. The Calc1 design ignored the carry_out bit from the adder ALU instead of using it to generate the overflow/underflow response. The verification engineer re-runs the test case with the fix and it runs successfully.

Further overflow and underflow test case permutations run against each of the ports and find no more bugs in this area.

Exercises

We have found two bugs thus far using deterministic test cases. We leave the test cases for sections 2 and 3 of the verification plan as exercises. How many more bugs can you find using deterministic test cases?
Describe any further bugs found and test plan section under which the bug was found.