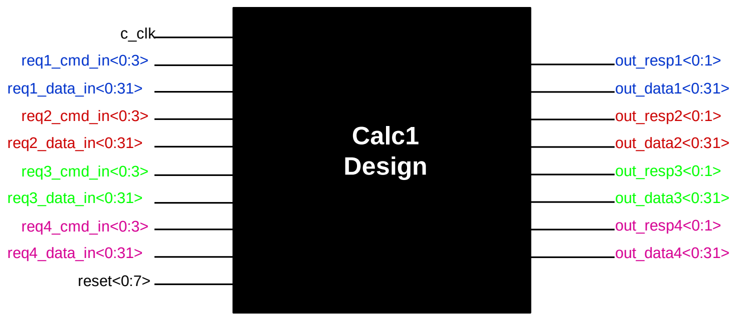
Calc1 Design Description

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!

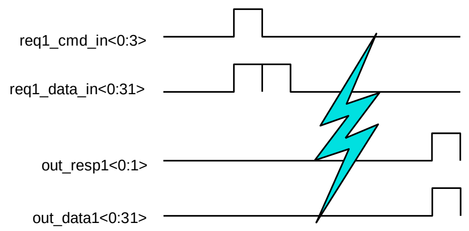


Figure 4.7: A timing diagram of a single port command sequence

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 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

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.

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 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. [1]

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