

ECE7387 Digital Systems Design Final Project:
Application Specific Vector Processor

Marc Pham

Professor Jason Moore

ECE7387 Digital Systems Design

November 29th, 2025

Project Overview	3
Truncated Adder	4
Truncated Adder Architecture	4
Truncated Adder Testbench	5
Truncated Multiplier	5
Truncated Multiplier Architecture	5
Truncated Multiplier Testbench	6
Dataflow Diagram	7
Resource Scheduling	8
Resource Scheduling Table for the Final Design	9
Minimum Register Allocation for the Final Design	10
Datapath	11
Datapath Inputs	11
Datapath Outputs	11
Datapath Diagram	12
Datapath Testbench	13
Controller	13
Controller Signals	13
ASM Chart	15
State Encoding Table	16
Controller Testbench	18
Top Module Testbench	18
Timing Analysis	21
Functionality Claims	21
Bibliography	22
Appendix A: Module Files	22
Appendix A.1: Half Adder	22
Appendix A.2: Full Adder	22
Appendix A.3: Full Adder without the Sum	22
Appendix A.4: Half Adder without the Sum	22
Appendix A.5: Truncated Adder (8 bit)	23
Appendix A.6: Truncated Multiplier (8-bit)	24
Appendix A.7: Datapath	27
Appendix A.8: Controller	30
Appendix A.9: Top	32
Appendix B: Testbenches	33
Appendix B.1: Truncated Adder (8-bit) Testbench	33
Appendix B.2: Truncated Multiplier (8-bit) Testbench	34
Appendix B.3: Datapath Testbench	36
Appendix B.4: Controller Testbench	38
Appendix B.5: Top Testbench	40

Project Overview

Before getting into the project, it is important to understand how each of the individual calculations will be handled in our vector processor. Here is a brief summary of how each is calculated.

Equation 1: $c_i = a_i \times b_i$.

Elements a_i and b_i are part of vectors a and b respectively, each being of size 4. Since a_i and b_i each have 8 bits, the final result c_i would naturally have a maximum of 16 bits, which we will truncate to the 8 most significant bits (MSB).

Equation 2: $e_i = c_i + d_i$.

Elements c_i and d_i have the 8 MSB of a 16-bit number. Adding the two will result in e_i having the 9 MSB of a 17-bit number, which we will truncate to the 8 MSB.

Equation 3: $f_i = e_i - (256 \times b_i)$.

Our process for determining how we will calculate this equation is shown in the diagram below and is elaborated upon in this paragraph. Element e_i is the 8 MSB of a 17-bit number. Element b_i is an 8-bit number that becomes a 16-bit number when multiplied by 256 (i.e. shifted to the right by 8 bits). To align with 17 bits in e_i , $256 \times b_i$ must be sign extended by adding b_7 as the most significant 17th bit. To represent the 2's complement of $-(256 \times b_i)$, we must invert the result of $256 \times b_i$ and add 1. Since we are truncating f_i to its 8 MSB, we only need to add the 8 MSB of e_i and $-(256 \times b_i)$ and have a carry-in of $(1 \& \sim b_i[0]) = \sim b_i[0]$. In conclusion, we can calculate this equation as $f_i = e_i + \{\sim b_i[7], \sim b_i[7:0]\} + \sim b_i[0]$. The output f_i will be the 8 MSB of an 18-bit signed number.

final carry-out of the previous stage: $sum_trunc[7] = a[7] \oplus b[7] \oplus c8$. This ensures that the truncated result reflects the correct sign and captures overflow from the lower bits as much as possible within the truncated representation. Careful propagation of carries through the higher bits guarantees that the two's complement arithmetic is respected even after truncation.

Truncated Adder Testbench

We can be confident that the truncated adder architecture works correctly based on the structure and results of the provided testbench. The testbench systematically exercises the adder across a wide range of inputs, including edge cases, boundary conditions, and random inputs, which collectively provide strong verification of the design. Originally, we designed a testbench that outputted the full sum, so we could determine that the adder properly truncated the final sum. However, to reduce the amount of inner logic and number of output pins, we removed the full sum from the final version of the truncated adder.

First, the directed tests demonstrate that the truncated adder handles all combinations of positive and negative numbers near the maximum and minimum representable 8-bit signed values. For example, inputs like $127 + 1$ and $-128 + -1$ test how the adder manages overflow conditions, while $127 + 127$ and $-128 + -128$ test the extremes of the signed range. In each case, the \$monitor output confirms that the top 8 bits of the 9-bit sum are correctly propagated to `sum_trunc`. By including both the decimal result of the full sum and the binary representation of the truncated sum, the testbench makes it easy to verify that the truncation logic matches the intended architecture.

Second, the boundary conditions such as $0 + 1$, $-1 + 2$, and $127 + -127$ check how the adder behaves when the sum is small or zero, ensuring that truncation does not accidentally discard significant information. The testbench confirms that the adder correctly outputs zero when appropriate, demonstrating that the design handles carry propagation and sign bits correctly even for edge cases.

Finally, the random tests further increase confidence in the design by exercising arbitrary input combinations, which may include unusual carry patterns or sign interactions not explicitly covered by the directed tests. Observing that random cases produce the expected truncated top 8 bits reinforces that the combinational logic is robust and functions correctly for all possible 8-bit signed inputs.

Overall, the testbench validates the truncated adder architecture by systematically covering corner cases, normal operation, and random inputs, showing that the implementation consistently produces the correct truncated result according to the designed logic. This combination of structured and random testing provides strong evidence of functional correctness.

Truncated Multiplier

Truncated Multiplier Architecture

The multiplier design is built on a shift-and-add architecture, which breaks the multiplication process into smaller, manageable pieces called partial products. Each bit of the

multiplier operand B is used to mask the multiplicand A, producing partial products p0 through p7. If a bit of B is set to 1, the corresponding partial product is equal to A; if it is 0, the partial product is zero. These partial products are then aligned by shifting them according to their bit position. For example, p0 is not shifted, p1 is shifted left by one, p2 by two, and so on, up to p7 which is shifted left by seven. This shifting ensures that each partial product contributes to the correct column of the final multiplication result, much like the manual long multiplication process. To handle signed values correctly, each partial product is sign-extended so that negative numbers propagate properly in two's complement form. The most significant partial product, p7, requires special handling because it represents the sign contribution of B. In this design, p7 is inverted and combined with an extra correction term when B is negative, ensuring the final product matches the expected signed result.

The truncated nature of the design means that we only care about the top 8 most significant bits of the product rather than the full 16-bit result. This allows us to save on logic because we do not need to compute the least significant 8 bits. In fact, this reduces the number of additions required by seven, since those lower-order sums would normally be finalized in a full multiplier. However, because the architecture relies on a column-by-column accumulation of partial products, the intermediate sums and carries in the lower columns cannot be ignored entirely. These carries ripple upward and directly affect the accuracy of the higher-order bits. As a result, the design still computes the intermediate additions in the lower columns, but it avoids committing extra logic to finalize the least significant outputs. This approach strikes a balance between efficiency and correctness, reducing gate count and improving performance while ensuring that the truncated MSB result remains accurate.

A key challenge was handling signed values in two's complement form. For partial products p0 through p6, sign extension ensures that negative numbers propagate correctly through the addition chain. The most significant bit p7 requires special treatment because it represents the sign contribution of B. Instead of a straightforward shift, the design inverts the bits and adds a correction term (`extra_add`) when B is negative. This adjustment ensures that the final product matches the expected two's complement result without requiring a full signed multiplier block. The combination of half-adders and full-adders in a carefully staged pipeline allows the design to accumulate the partial products efficiently while maintaining correctness for signed inputs.

Truncated Multiplier Testbench

This testbench is designed to exercise all the important corner cases and boundary conditions for the truncated signed multiplier, giving us confidence that the design works correctly. It begins with the simplest cases such as multiplying zero by zero and one by one, which verify the basic functionality of the partial product generation and addition network. It then moves to the extreme values of the signed 8-bit range: maximum positive (127) and minimum negative (-128). Testing these extremes ensures that the sign extension and special handling of the most significant partial product (p7) behave correctly, especially when both

operands are negative or when one is positive and the other negative. These cases are critical because they stress the two's complement arithmetic and confirm that the correction term (`extra_add`) is applied properly.

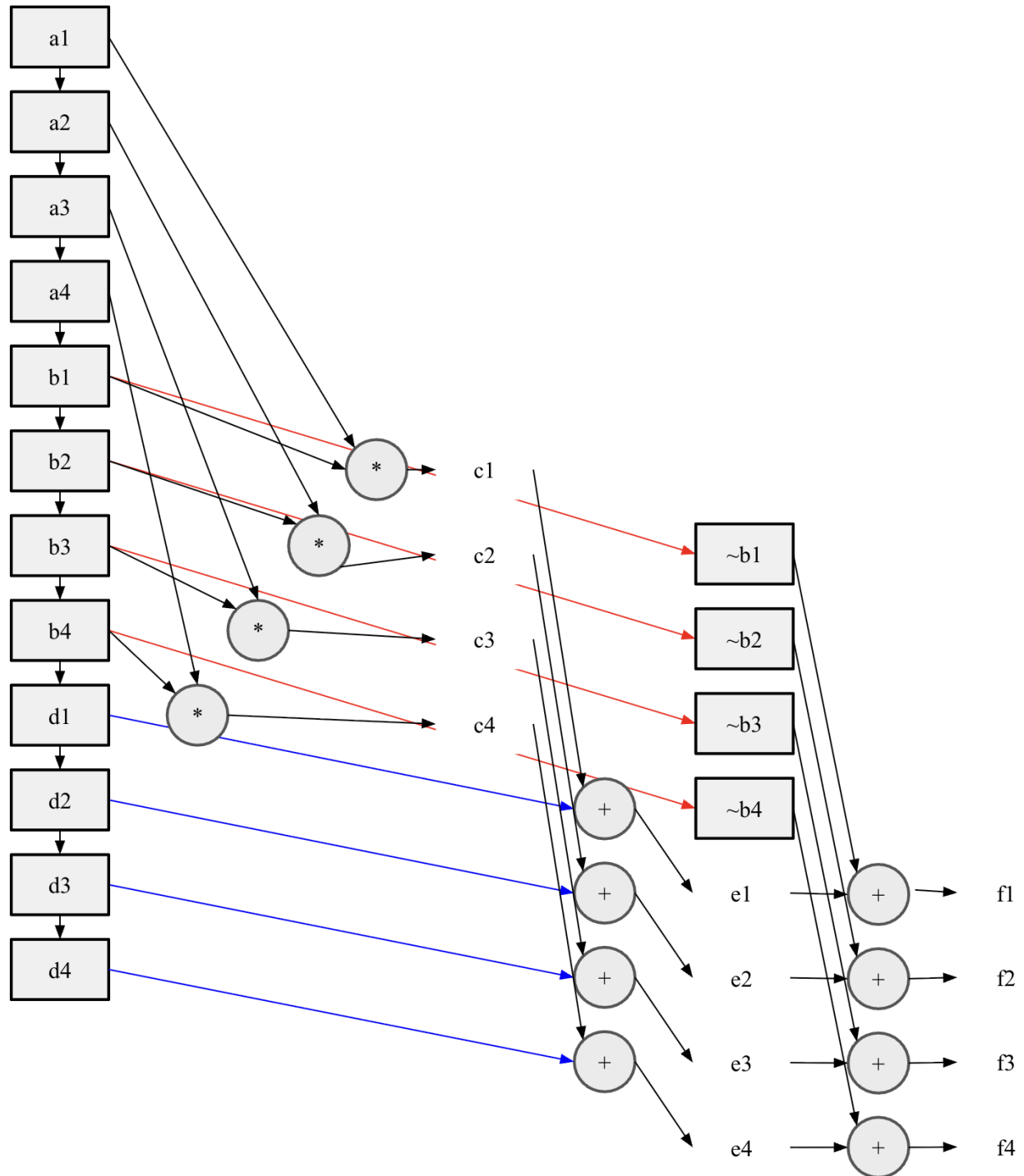
The testbench also includes boundary tests around the sign bit, such as multiplying 127 by 2 or -128 by 2, which check that carries propagate correctly into the MSBs and that truncation does not distort the result. Additional cases like multiplying ± 64 by ± 64 further validate the handling of mid-range signed values, ensuring that both positive and negative combinations produce the expected outputs. Finally, the inclusion of randomized test vectors provides broader coverage across the input space, catching any subtle errors that might not appear in directed edge cases. By combining directed tests for extremes, sign boundaries, and random values, the testbench thoroughly validates the truncated multiplier's correctness and gives confidence that the design produces accurate MSB results under all possible input conditions.

Dataflow Diagram

Using the designed Truncated Adder and Truncated Multiplier, we designed a dataflow diagram to determine how we will use these components to calculate the Equations 1 - 3.

Comments: Outputs `c1` - `c4` and `e1` to `e4` are considered to be automatically registered after the operation that calculates them finishes. So, even though they are not labeled as a box since they are outputs, they can still be used in subsequent operations since they have been registered after the operation that calculated them. Second, for the calculation of `f`, I am doing the following operation: `TruncatedAdder(e[7:0], {~b[7], ~b[7:1]}, cin = ~b[0])`. This means I have to calculate and register the values of `~b1` to `~b4` somewhere in the dataflow diagram. Their values will override the value in the registers of `b1` to `b4` since `b1` to `b4` are no longer needed for further calculations.

The longest path through the dataflow diagram is 14 clock cycles, taking 12 clock cycles to get from input `a1` to `d4` then taking an additional 2 clock cycles to get from `d4` to `f4`. In total, there are 4 truncated multiplications and 8 truncated additions. Thus, the estimated minimum resources needed are $\text{ceiling}(4 \text{ multiplications} / 14 \text{ clock cycles}) = 1 \text{ multiplier}$ and $\text{ceiling}(8 \text{ truncated additions} / 14 \text{ clock cycles}) = 1 \text{ addition}$.



Resource Scheduling

If we try to schedule resources with 1 adder and 1 multiplier, the smallest latency we can achieve is 17 clock cycles, which does not meet our latency requirements. Thus, our only option is to add more resources because we cannot add more clock cycles. Adding a multiplier would

not help because we are already doing our multiplications as early as we can due to the availability of its needed inputs a and b. Our best bet is to add a second adder, so we can calculate the values for vectors f and e concurrently.

Note: When we are calculating $e1 + -256*b1 \rightarrow f1$, we are actually calculating $\text{TruncatedAdder}(e[7:0], \{\sim b[7], \sim b[7:1]\}, \text{cin} = \sim b[0])$. No truncated multiplications are needed. The only hardware needed is a truncated adder, an inverter and a shifter. Since we were only meant to include the adder and multipliers as the functional elements in our dataflow diagram, I did not include the inverter and shifter in the resource scheduling table.

Clock Cycle	Inputs	Adder 1	Multiplier 1	Outputs
1	Input a1	Idle	Idle	
2	Input a2	Idle	Idle	
3	Input a3	Idle	Idle	
4	Input a4	Idle	Idle	
5	Input b1	Idle	Idle	
6	Input b2	Idle	$a1*b1 \rightarrow c1$	Output c1
7	Input b3	Idle	$a2*b2 \rightarrow c2$	Output c2
8	Input b4	Idle	$a3*b3 \rightarrow c3$	Output c3
9	Input d1	Idle	$a4*b4 \rightarrow c4$	Output c4
10	Input d2	$c1+d1 \rightarrow e1$	Idle	Output e1
11	Input d3	$c2+d2 \rightarrow e2$	Idle	Output e2
12	Input d4	$c3+d3 \rightarrow e3$	Idle	Output e3
13		$c4+d4 \rightarrow e4$	Idle	Output e4
14		$e1 + -256*b1 \rightarrow f1$	Idle	Output f1
15		$e2 + -256*b2 \rightarrow f2$	Idle	Output f2
16		$e3 + -256*b3 \rightarrow f3$	Idle	Output f3
17		$e4 + -256*b4 \rightarrow f4$	Idle	Output f4

Resource Scheduling Table for the Final Design

By adding an additional adder, the system's latency becomes 14 clock cycles, since the final output f4 is produced at cycle 14. The initiation interval is 12 clock cycles, meaning a new set of inputs can only be processed every 12 cycles. This initiation rate cannot be improved unless the design is modified to accept multiple inputs per clock cycle.

Clock Cycle	Inputs	Adder 1	Adder 2	Multiplier 1	Outputs
1	Input a1	Idle	Idle	Idle	
2	Input a2	Idle	Idle	Idle	

3	Input a3	Idle	Idle	Idle	
4	Input a4	Idle	Idle	Idle	
5	Input b1	Idle	Idle	Idle	
6	Input b2	Idle	Idle	$a1 * b1 \rightarrow c1$	Output c1
7	Input b3	Idle	Idle	$a2 * b2 \rightarrow c2$	Output c2
8	Input b4	Idle	Idle	$a3 * b3 \rightarrow c3$	Output c3
9	Input d1	Idle	Idle	$a4 * b4 \rightarrow c4$	Output c4
10	Input d2	$c1 + d1 \rightarrow e1$	Idle	Idle	Output e1
11	Input d3	$c2 + d2 \rightarrow e2$	$e1 + -256 * b1 \rightarrow f1$	Idle	Output e2, f1
12	Input d4	$c3 + d3 \rightarrow e3$	$e2 + -256 * b2 \rightarrow f2$	Idle	Output e3, f2
13	Input a1	$c4 + d4 \rightarrow e4$	$e3 + -256 * b3 \rightarrow f3$	Idle	Output e4, f3
14	Input a2	Idle	$e4 + -256 * b4 \rightarrow f4$	Idle	Output f4
15	Input a3	Idle	Idle	Idle	
16	Input a4	Idle	Idle	Idle	
17	Input b1	Idle	Idle	Idle	
18	Input b2	Idle	Idle	Idle	
19	Input b3	Idle	Idle	Idle	
20	Input b4	Idle	Idle	$a1 * b1 \rightarrow c1$	Output c1
21	Input d1	Idle	Idle	$a2 * b2 \rightarrow c2$	Output c2
22	Input d2	Idle	Idle	$a3 * b3 \rightarrow c3$	Output c3
23	Input d3	Idle	Idle	$a4 * b4 \rightarrow c4$	Output c4
24	Input d4	$c1 + d1 \rightarrow e1$	Idle	Idle	Output e1

Minimum Register Allocation for the Final Design

With this resource scheduling, we need at least 9 registers depending on if you want to have access to all the outputs c1 to c4, e1 to e4, and f1 to f4 after 14 clock cycles. If you are fine with overriding the intermediate outputs c and e, then you would need at least 9 registers. If you would like to keep the outputs c and e until the end, you need at least 13 registers.

At first, you need 8 registers to keep the unique values of a1 to a4 and b1 to b4. When c1 to c4 are calculated, you can reuse the registers with a1 to a4 since they are no longer needed in future computations. The registers for b1 to b4 will be reused to store the values of $\sim b1$ to $\sim b4$ for future calculations of f1 to f4. Only 1 register is needed for d1 to d4 because, once one d input enters the system, another d input will be used in the calculation of e and will be no longer needed in the future. For example, when input d2 enters the system, input d1 will be used in the calculation for e1 and can be overwritten because d1 is not used in any future calculations.

For the calculation of e1 to e4, we can reuse the registers for c1 to c4 since they will no longer be used in future calculations. However, if you would like to keep the outputs c until the

end, you will need 4 additional registers to store e1 to e4. Finally, for the calculation of f1 to f4, we can reuse the registers for ~b1 to ~b4 because they are only used to calculate f. During clock cycles 13 and 14, we also have to account for the incoming inputs a1 and a2 of the next computation. During clock cycle 13, we can replace the register for d4 with the incoming a1 since it no longer has a purpose after computing e4. During clock cycle 14, we can replace the register e1 with the incoming a2, since e1 is no longer used in a computation. However, if you would like to keep the output e1 until the end, you will need 1 additional register for a2.

Datapath

Datapath Inputs

Datapath Inputs	Explanation
clk	Clock signal used to control the controller.
rst	Asynchronous Reset to reset all the values within the datapath to 0 and send the controller back to its initial state.
din	8-bit signed input value. Corresponds to the values a_1 to a_4 , b_1 to b_4 , or d_1 to d_4 depending on the clock cycle.

Datapath Outputs

Datapath Outputs	Explanation
f1	8-bit signed output value corresponding to the output f_1 .
f2	8-bit signed output value corresponding to the output f_2 .
f3	8-bit signed output value corresponding to the output f_3 .
f4	8-bit signed output value corresponding to the output f_4 .

Datapath Diagram

In our diagram, each register is connected to the reset (rst) input that will set it back to zeroes. Each register also has an associated Enable bit that specifies when it can change its value, which isn't specified here since the Enable bits are controller bits. The notation $\{[7], [7:1]\}$ represents the concatenation of bit 7 with bits 7 - 1 of whatever 8-bit value is on the wire.

This testbench provides thorough coverage of the datapath while keeping the sequence of tests straightforward. It begins with a reset pulse to ensure all registers start from a known state, verifying proper initialization. Positive values are then loaded into the A registers (a1–a4) to confirm that each responds correctly to its enable signal and can store signed data independently.

This testbench provides thorough coverage of the datapath while keeping the sequence of tests straightforward. It begins with a reset pulse to ensure all registers start from a known state, verifying proper initialization. Positive values are then loaded into the A registers (a1–a4) to confirm that each responds correctly to its enable signal and can store signed data independently.

Negative values are loaded into the B registers (b1–b4), which checks handling of signed inputs and prepares them for later inversion and carry-in logic tests.

The multiplier path is exercised using the `save_c` flag, ensuring that A registers can capture either direct input or the multiplication result depending on control. The `add1` paths are validated by sequentially enabling `en_add1_1` through `en_add1_4` with different inputs, confirming that each A register can pair with input data for addition. The `add2` paths are then tested by enabling `en_add2_1` through `en_add2_4`, which drives `c_add1` and the bitwise-inverted version of each B register into the second addition unit, verifying inversion and carry-in behavior.

Output registers f1–f4 are checked by enabling each one in turn, confirming that results from the second addition can be stored independently. Finally, several corner cases are included: maximum positive (127) and minimum negative (-128) inputs test signed overflow boundaries; overlapping enable bits shows that multiple registers can be updated in the same cycle; and simultaneous `add1/add2` operations confirm that concurrent arithmetic paths function correctly.

Altogether, these vectors ensure that every register, arithmetic path, control signal, and edge case of signed 8-bit data is exercised. The testbench balances completeness with clarity, covering both normal operation and boundary conditions without unnecessary redundancy.

Controller

Controller Signals

In our controller, all of our controller signals are internal outputs (with respect to the controller) since they are generated by the controller to control elements in the datapath. Additionally, unlike external outputs, our controller signals do not leave our system.

Signal	Type	Description
en_a1	Internal Output	Allows the a1 register to store either the input data (din) or the multiplication result (c_mult) depending on the save_c flag.
en_a2	Internal Output	Allows the a2 register to store either din or c_mult .
en_a3	Internal Output	Allows the a3 register to store either din or c_mult .
en_a4	Internal Output	Allows the a4 register to store either din or c_mult .
en_b1	Internal Output	Loads the first B register (b1) with the input data (din) and also sets the multiplication inputs (a_mult from a1 , b_mult from din).

en_b2	Internal Output	Loads the second B register (b2) with din and sets multiplication inputs (a_mult from a2 , b_mult from din).
en_b3	Internal Output	Loads the third B register (b3) with din and sets multiplication inputs (a_mult from a3 , b_mult from din).
en_b4	Internal Output	Loads the fourth B register (b4) with din and sets multiplication inputs (a_mult from a4 , b_mult from din).
en_add1_1	Internal Output	Sends c1 (stored in a1) and d1 (input as din) into the first addition unit (a_add1 , b_add1), with a carry-in of 0.
en_add1_2	Internal Output	Sends c2 (stored in a2) and d2 (input as din) into the first addition unit, with a carry-in of 0.
en_add1_3	Internal Output	Sends c3 (stored in a3) and d3 (input as din) into the first addition unit, with a carry-in of 0.
en_add1_4	Internal Output	Sends c4 (stored in a4) and d4 (input as din) into the first addition unit, with a carry-in of 0.
en_add2_1	Internal Output	Sends inputs into the second addition unit (a_add2 , b_add2) in the following manner: <ul style="list-style-type: none"> • One input is the result of the first addition (c_add1). • Another input is the bitwise-inversion of b1, which is the arithmetically shifted right to maintain its sign bit in its MSB. • Carry-in is set to the inversion of the LSB of b1.
en_add2_2	Internal Output	Same as above, but uses the second B register (b2).
en_add2_3	Internal Output	Same as above, but uses the third B register (b3).
en_add2_4	Internal Output	Same as above, but uses the fourth B register (b4).
en_f1	Internal Output	Outputs the result of the second addition (c_add2) as the output f1 .
en_f2	Internal Output	Outputs the result of the second addition (c_add2) as the output f2 .

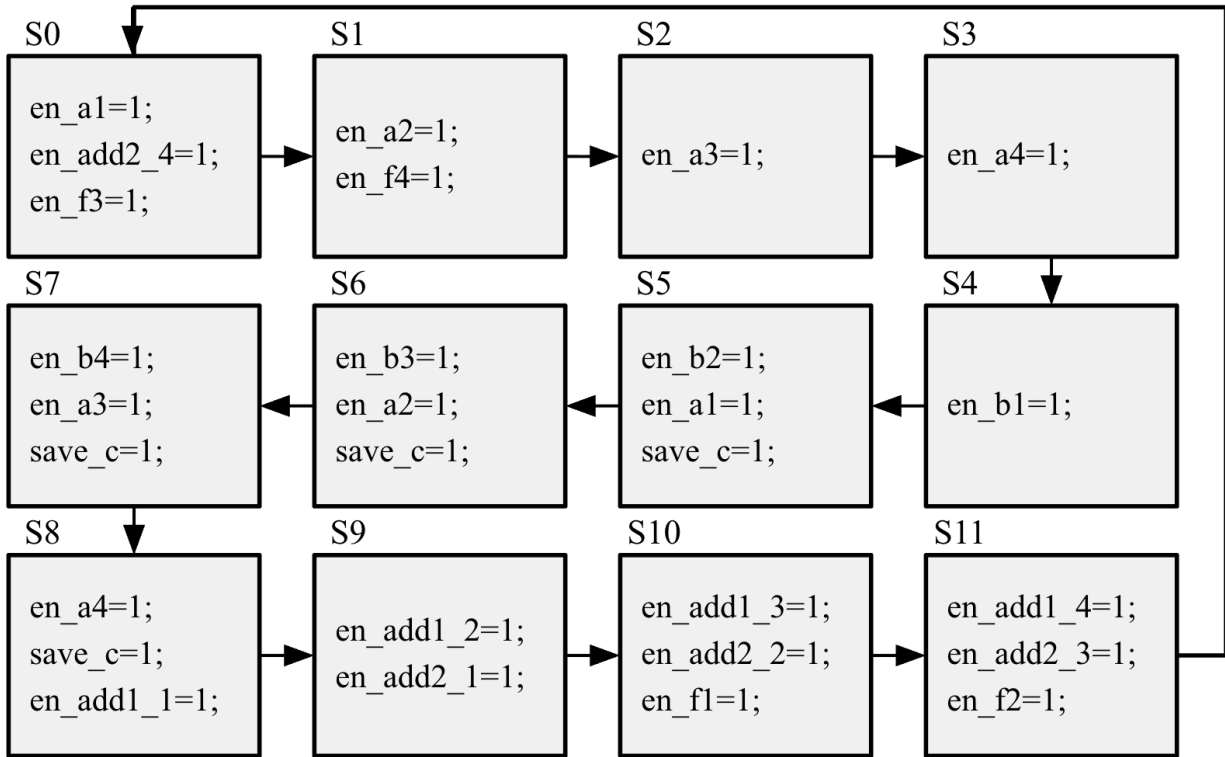
en_f3	Internal Output	Outputs the result of the second addition (c_add2) as the output f3 .
en_f4	Internal Output	Outputs the result of the second addition (c_add2) as the output f4 .
save_c	Internal Output	Control flag that decides whether the registers a1–a4 load from the multiplication result (c_mult) instead of the input data (din).

ASM Chart

To simplify the control of the datapath and control signals, I decided to use 12 states, which is equal to our initiation rate of 12 clock cycles. This decision removed the complexity of trying to use counters and conditional variables to reduce the number of states. Since each state is associated with a unique clock cycle, it was much easier to determine and test the value of the controller signals for each state and the commands that should be run in the datapath in response. Additionally, only 4 bits are needed to represent 12 states, which is not much higher than a controller using fewer states.

In the ASM chart, states S0 to S4 have the enable bits for registers a1 to a4 (**en_a1** to **en_a4**) because the values of a1 to a4 are accepted during the first 4 clock cycles. States S4 to S7 have the enable bits for registers b1 to b4 (**en_b1** to **en_b4**) since the second 4 clock cycles is when the values of b1 to b4 are accepted. These enable bits also function as the enable bits for the truncated multiplication of a and b since they can start the calculation as soon as b is accepted. The results of the multiplication – which we call c – can be saved after 1 clock cycle, which corresponds to states S5 to S8. Enable bits **en_a1** to **en_a4** specify that the datapath will store c into the registers of a and **save_c** ensures that the value of c will be saved instead of the data input din. We reuse the registers of a1 to a4 like we discussed in the Section [Minimum Register Allocation for the Final Design](#).

States S8 to S11 have the enable bits for the first truncated adder (**en_add1_1** to **en_add1_4**) because we can start the first addition as soon as we accept the value of d. We use the results of the first truncated adder as an input for the second truncated adder, so we must wait an extra clock cycle to start the second addition. Thus, States S9 to S11 and S0 have the enable bits for the second truncated adder (**en_add2_1** to **en_add2_4**). We include S0 because we are using pipelining to overlap the end of one operation and the beginning of another operation. Finally, we must wait for the results of the second truncated addition to produce the values of f1 to f4. Thus, States S10 to S11 and S0 to S1 have the enable bits for f1 to f4 (**en_f1** to **en_f4**).



State Encoding Table

In order to determine our optimal state encoding, we can use Gray Code encoding. This encoding scheme allows us to have at most 1 bit change between each of our 12 states. However, the transition from state 11 to state 0 has a total of 3 bit changes, which we would like to avoid.

State	Gray Code
0	0000
1	0001
2	0011
3	0010
4	0110
5	0111

6	0101
7	0100
8	1100
9	1101
10	1111
11	1110

To have only 1 bit change from state 11 to state 0, so we can rearrange the provided Gray code encoding like so. This ensures that there is at most 1 bit change between any state transition.

State	Gray Code
0	0000
1	0001
2	0011
3	0010
4	0110
5	0111
6	1111
7	1110
8	1100
9	1101

10	0101
11	0100

Controller Testbench

This controller testbench is structured to verify that the finite state machine (FSM) driving the datapath behaves correctly in both normal operation and under corner conditions. It does this by carefully sequencing reset and clock events, then monitoring all of the controller's output signals as the FSM cycles through its states.

The test begins with an initial reset pulse. This ensures that the controller starts from a known state and that all outputs are cleared before the FSM begins cycling. Once reset is deasserted, the test allows the FSM to run through all of its states twice. This guarantees that every control signal is asserted at least once, confirming that the state machine activates each datapath path as intended.

A key corner case is then introduced: reset is asserted in the middle of a cycle, not just at startup. This checks that the FSM can recover gracefully from an unexpected reset, returning to its initial state and resuming correct sequencing once reset is released. By doing this mid-cycle, the test ensures that the controller does not lock up or produce invalid outputs when reset occurs at arbitrary times.

Finally, the FSM is allowed to run again after the mid-cycle reset, verifying that it resumes normal operation and continues to cycle through states correctly. The \$monitor statement continuously prints all control signals, making it easy to confirm that each enable and flag behaves as expected across normal cycles, reset conditions, and recovery.

Top Module Testbench

The 40 test vectors were chosen to provide comprehensive confidence that the top module behaves correctly under every meaningful condition. The first groups establish basic functionality and sign combinations, ensuring the design correctly interprets positive and negative inputs, zero, and the canonical extremes like 0, 1, 127, -128, and -1. By covering all sign permutations — (+,+,+), (+,+,-), ... , (-,-,-) — the testbench validates that two's complement arithmetic and sign extension are consistently handled. From there, the vectors deliberately stress overflow interactions: multiplication cases where mid-range and extreme values exceed the intermediate product width, addition cases where $c + d$ overflows a 16-bit sum, and subtraction cases where $(256*b)$ dominates the result, probing whether the design correctly subtracts large shifted terms without collapsing precision.

Beyond arithmetic extremes, the suite incorporates pattern-based bugs and boundary edges. Alternating bit patterns like 0x55 and 0xAA are notorious for exposing hardware logic mistakes such as mis-wired XORs or masking errors, while boundary transitions around 0x7F, 0x80, and 0xFF catch off-by-one and sign-bit propagation flaws. The vectors also include

symmetric and asymmetric interactions, where small values multiply or add with large ones, testing cancellation and reinforcement effects that could otherwise slip through. Finally, the inclusion of random-but-covering mid-range patterns ensures the design behaves correctly outside contrived extremes, simulating realistic arithmetic cases. Taken together, these criteria create a balanced suite that validates correctness under normal operation, stresses the design under pathological overflow, and probes for hidden logic flaws with structured patterns — giving very high confidence that the top module works as intended.

Below is a table of the various test vectors used for the top module and their correct output values (f). Keep in mind that **d** represents the top 8 MSB of a 16-bit number, so the listed decimal value is actually the value of the 8 MSB if they were to be treated as a standalone number. The decimal value listed is what the value of **d** will appear as in a behavioral simulation. In general, it will take 12 clock cycles to accept the coefficient **a1**, **a2**, **a3**, or **a4** of a test vector and receive its corresponding output **f1**, **f2**, **f3**, or **f4**.

Test Vector	a (binary, decimal)	b (binary, decimal)	d (binary, decimal)	f (binary, decimal)
1	00000000, 0	00000000, 0	00000000, 0	00000000, 0
2	00000001, 1	00000001, 1	00000001, 1	11111111, -1
3	01111111, 127	01111111, 127	00000000, 0	11101111, -17
4	10000000, -128	10000000, -128	00000000, 0	00110000, 48
5	01111111, 127	10000000, -128	00000000, 0	00010000, 16
6	10000000, -128	01111111, 127	00000000, 0	11010000, -48
7	00000000, 0	01111111, 127	00010000, 16	11100100, -28
8	01111111, 127	00000000, 0	00100000, 32	00001000, 8
9	00000000, 0	10000000, -128	11110000, -16	00011100, 28
10	10000000, -128	00000000, 0	11110001, -15	11111100, -4
11	01111111, 127	11111111, -1	10000000, -128	11011111, -33
12	11111111, -1	01111111, 127	10000000, -128	10111111, -65
13	10000000, -128	11111111, -1	01111111, 127	00011111, 31
14	11111111, -1	10000000, -128	01111111, 127	00111111, 63

15	01000000, 64	01000000, 64	00000000, 0	11110100, -12
16	01100000, 96	01100000, 96	00000000, 0	11110001, -15
17	01010000, 80	11010000, -48	00000000, 0	00001000, 8
18	10010000, -112	10010000, -112	00000000, 0	00101000, 40
19	01111111, 127	00000010, 2	01111111, 127	00011111, 31
20	01111111, 127	01111111, 127	01111111, 127	00001111, 15
21	10000000, -128	00000010, 2	10000000, -128	11011111, -33
22	10000000, -128	10000000, -128	10000000, -128	00010000, 16
23	00000001, 1	11110000, -16	00010000, 16	00000111, 7
24	00000010, 2	11110000, -16	00000000, 0	00000011, 3
25	00000011, 3	11110000, -16	11111111, -1	00000011, 3
26	00010000, 16	00001111, 15	10000000, -128	11011100, -36
27	00000001, 1	10000000, -128	01111111, 127	00111111, 63
28	01010101, 85	01010101, 85	01010101, 85	00000110, 6
29	10101010, -86	10101010, -86	10101010, -86	00000111, 7
30	01010101, 85	10101010, -86	01010101, 85	00100011, 35
31	10101010, -86	01010101, 85	10101010, -86	11001101, -51
32	00000001, 1	10000000, -128	01111111, 127	00111111, 63
33	01111111, 127	00000001, 1	10000000, -128	11011111, -33
34	10000000, -128	00000001, 1	01111111, 127	00011111, 31
35	11111111, -1	00000001, 1	11111111, -1	11111111, -1
36	00010011, 19	11100111, -25	10100100, -92	11101110, -18
37	01101011, 107	10010010, -110	00010000, 16	00010100, 20

38	11110011, -13	00001100, 12	01111110, 126	00011100, 28
39	00100000, 32	00000100, 4	10000000, -128	11011111, -33
40	00010000, 16	11110000, -16	00001111, 15	00000111, 7

Timing Analysis

Using Vivado's Timing Analysis Tool, we can determine the max clock frequency with the following equation¹:

$$\text{Maximum Clock Frequency} = 1 / (\text{Target Clock Period} - \text{Worst Negative Slack})$$

Since we synthesized our design with a period of 100ns and the Worst Negative Slack is 86.046ns according to the Timing Analysis Tool, our Maximum Clock Frequency is $1 / (100\text{ns} - 86.046\text{ns}) = 71.664 \text{ MHz}$.

Timing	
Worst Negative Slack (WNS):	86.046 ns
Total Negative Slack (TNS):	0 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	252
Implemented Timing Report	

Functionality Claims

The design takes as inputs a 1-bit reset, a clock signal, an 8-bit input bus, and produces results on an 8-bit output bus. Over the course of 12 clock cycles, it receives the 2's complement values a1 through a4, b1 through b4, and d1 through d4. The computation executes 3 main equations:

Equation 1: $c_i = a_i \times b_i$.

Equation 2: $e_i = c_i + d_i$.

Equation 3: $f_i = e_i - (256 \times b_i)$.

¹ The equation comes from Vivado's Documentation: AMD. *UltraFast Design Methodology Guide for FPGAs and SoCs (UG949)*. Version 2025.1, May 29, 2025. Available at: [AMD Documentation Portal](#)

As discussed in the Section [Project Overview](#), c_i holds the top 8 MSB of a 16-bit number, e_i holds the 8 MSB of a 17-bit number, and the final output f_i holds the 8 MSB of an 18-bit signed number.

The outputs, f1 through f4, are produced as 2's complement values across 4 clock cycles, beginning once d3 has been received. The design achieves an initiation rate of 12 clock cycles and a latency of 14 clock cycles, meeting the project constraints of an initiation rate no greater than 13 cycles and a latency no greater than 15 cycles.

Bibliography

AMD. *UltraFast Design Methodology Guide for FPGAs and SoCs (UG949)*. Version 2025.1, May 29, 2025. Available at: [AMD Documentation Portal](#)

Appendix A: Module Files

Appendix A.1: Half Adder

```
module half_adder(input a, b, output sum, carry);
    assign sum  = a ^ b;
    assign carry = a & b;
endmodule
```

Appendix A.2: Full Adder

```
module full_adder(input a, b, cin, output sum, carry);
    assign sum  = a ^ b ^ cin;
    assign carry = (a & b) | (a & cin) | (b & cin);
endmodule
```

Appendix A.3: Full Adder without the Sum

Calculates the carry-out of the Full Adder without producing the final sum.

```
module full_carry_only(input a, b, cin, output carry);
    assign carry = (a & b) | (a & cin) | (b & cin);
endmodule
```

Appendix A.4: Half Adder without the Sum

Calculates the carry-out of the Half Adder without producing the final sum

```
module half_carry_only(input a, b, output carry);
```

```

    assign carry = a & b;
endmodule

```

Appendix A.5: Truncated Adder (8 bit)

```

module truncated_adder_8bit (
    input  wire signed [7:0] a,
    input  wire signed [7:0] b,
    input  wire c0,
    output reg signed [7:0] sum_trunc
    // output reg signed [8:0] full_sum // full 9-bit sum
);

    reg c1, c2, c3, c4, c5, c6, c7, c8; // carry bits

    always @(*) begin

        // No need to calculate the first digit since we are truncating it anyway.
        c1 = (a[0] & b[0]) | (a[0] & c0) | (b[0] & c0);

        sum_trunc[0] = a[1] ^ b[1] ^ c1;
        c2 = (a[1] & b[1]) | (a[1] & c1) | (b[1] & c1);

        sum_trunc[1] = a[2] ^ b[2] ^ c2;
        c3 = (a[2] & b[2]) | (a[2] & c2) | (b[2] & c2);

        sum_trunc[2] = a[3] ^ b[3] ^ c3;
        c4 = (a[3] & b[3]) | (a[3] & c3) | (b[3] & c3);

        sum_trunc[3] = a[4] ^ b[4] ^ c4;
        c5 = (a[4] & b[4]) | (a[4] & c4) | (b[4] & c4);

        sum_trunc[4] = a[5] ^ b[5] ^ c5;
        c6 = (a[5] & b[5]) | (a[5] & c5) | (b[5] & c5);

        sum_trunc[5] = a[6] ^ b[6] ^ c6;
        c7 = (a[6] & b[6]) | (a[6] & c6) | (b[6] & c6);

        sum_trunc[6] = a[7] ^ b[7] ^ c7;
        c8 = (a[7] & b[7]) | (a[7] & c7) | (b[7] & c7);

        // 9th bit (carry-out) using the sign bits
        sum_trunc[7] = a[7] ^ b[7] ^ c8;
    end
end

```

```
endmodule
```

Appendix A.6: Truncated Multiplier (8-bit)

```
module truncated_multiplier_8bit (
    input signed [7:0] A, B,
    output [7:0] c
);

    wire [7:0] p0, p1, p2, p3, p4, p5, p6, p7;
    wire [15:0] p0_ext, p1_ext, p2_ext, p3_ext, p4_ext, p5_ext, p6_ext, p7_ext;
    wire [15:0] extra_add;

    // Generate partial products
    assign p0 = A & {8{B[0]}};
    assign p1 = A & {8{B[1]}};
    assign p2 = A & {8{B[2]}};
    assign p3 = A & {8{B[3]}};
    assign p4 = A & {8{B[4]}};
    assign p5 = A & {8{B[5]}};
    assign p6 = A & {8{B[6]}};
    assign p7 = A & {8{B[7]}};

    // Sign-extend and shift partial products
    assign p0_ext = {{8{p0[7]}}, p0}; // shift 0
    assign p1_ext = {{7{p1[7]}}, p1, 1'b0}; // shift 1
    assign p2_ext = {{6{p2[7]}}, p2, 2'b0}; // shift 2
    assign p3_ext = {{5{p3[7]}}, p3, 3'b0}; // shift 3
    assign p4_ext = {{4{p4[7]}}, p4, 4'b0}; // shift 4
    assign p5_ext = {{3{p5[7]}}, p5, 5'b0}; // shift 5
    assign p6_ext = {{2{p6[7]}}, p6, 6'b0}; // shift 6

    // Handle sign row (p7)
    assign p7_ext = B[7] ? {~p7[7], ~p7, 7'b0} : {p7, 7'b0}; // invert + shift
    assign extra_add = B[7] ? 16'b0000000010000000 : 16'b0; // +1 if negative

    // Final product
    // assign product = A * B;

    wire c1a;
    half_carry_only ha1a(p0_ext[1], p1_ext[1], c1a);
```

```

wire s2a, c2a, c2b;
full_adder fa2a(p0_ext[2], p1_ext[2], c1a, s2a, c2a);
half_carry_only fa2b(s2a, p2_ext[2], c2b);

wire s3a, s3b, c3a, c3b, c3c;
full_adder fa3a(p0_ext[3], p1_ext[3], c2a, s3a, c3a);
full_adder fa3b(s3a, p2_ext[3], c2b, s3b, c3b);
half_carry_only fa3c(s3b, p3_ext[3], c3c);

wire s4a, s4b, s4c, c4a, c4b, c4c, c4d;
full_adder fa4a(p0_ext[4], p1_ext[4], c3a, s4a, c4a);
full_adder fa4b(s4a, p2_ext[4], c3b, s4b, c4b);
full_adder fa4c(s4b, p3_ext[4], c3c, s4c, c4c);
half_carry_only fa4d(s4c, p4_ext[4], c4d);

wire s5a, s5b, s5c, s5d, c5a, c5b, c5c, c5d, c5e;
full_adder fa5a(p0_ext[5], p1_ext[5], c4a, s5a, c5a);
full_adder fa5b(s5a, p2_ext[5], c4b, s5b, c5b);
full_adder fa5c(s5b, p3_ext[5], c4c, s5c, c5c);
full_adder fa5d(s5c, p4_ext[5], c4d, s5d, c5d);
half_carry_only fa5e(s5d, p5_ext[5], c5e);

wire s6a, s6b, s6c, s6d, s6e, c6a, c6b, c6c, c6d, c6e, c6f;
full_adder fa6a(p0_ext[6], p1_ext[6], c5a, s6a, c6a);
full_adder fa6b(s6a, p2_ext[6], c5b, s6b, c6b);
full_adder fa6c(s6b, p3_ext[6], c5c, s6c, c6c);
full_adder fa6d(s6c, p4_ext[6], c5d, s6d, c6d);
full_adder fa6e(s6d, p5_ext[6], c5e, s6e, c6e);
half_carry_only fa6f(s6e, p6_ext[6], c6f);

wire s7a, s7b, s7c, s7d, s7e, s7f, c7a, c7b, c7c, c7d, c7e, c7f, c7g;
full_adder fa7a(p0_ext[7], p1_ext[7], c6a, s7a, c7a);
full_adder fa7b(s7a, p2_ext[7], c6b, s7b, c7b);
full_adder fa7c(s7b, p3_ext[7], c6c, s7c, c7c);
full_adder fa7d(s7c, p4_ext[7], c6d, s7d, c7d);
full_adder fa7e(s7d, p5_ext[7], c6e, s7e, c7e);
full_adder fa7f(s7e, p6_ext[7], c6f, s7f, c7f);
full_carry_only fa7g(s7f, p7_ext[7], extra_add[7], c7g);

// First actual output that matters
wire s8a, s8b, s8c, s8d, s8e, s8f, c8a, c8b, c8c, c8d, c8e, c8f, c8g;
full_adder fa8a(p0_ext[8], p1_ext[8], c7a, s8a, c8a);
full_adder fa8b(s8a, p2_ext[8], c7b, s8b, c8b);

```

```

full_adder fa8c(s8b, p3_ext[8], c7c, s8c, c8c);
full_adder fa8d(s8c, p4_ext[8], c7d, s8d, c8d);
full_adder fa8e(s8d, p5_ext[8], c7e, s8e, c8e);
full_adder fa8f(s8e, p6_ext[8], c7f, s8f, c8f);
full_adder fa8g(s8f, p7_ext[8], c7g, c[0], c8g);

```

```

wire s9a, s9b, s9c, s9d, s9e, s9f, c9a, c9b, c9c, c9d, c9e, c9f, c9g;
full_adder fa9a(p0_ext[9], p1_ext[9], c8a, s9a, c9a);
full_adder fa9b(s9a, p2_ext[9], c8b, s9b, c9b);
full_adder fa9c(s9b, p3_ext[9], c8c, s9c, c9c);
full_adder fa9d(s9c, p4_ext[9], c8d, s9d, c9d);
full_adder fa9e(s9d, p5_ext[9], c8e, s9e, c9e);
full_adder fa9f(s9e, p6_ext[9], c8f, s9f, c9f);
full_adder fa9g(s9f, p7_ext[9], c8g, c[1], c9g);

```

```

wire s10a, s10b, s10c, s10d, s10e, s10f, c10a, c10b, c10c, c10d, c10e, c10f, c10g;
full_adder fa10a(p0_ext[10], p1_ext[10], c9a, s10a, c10a);
full_adder fa10b(s10a, p2_ext[10], c9b, s10b, c10b);
full_adder fa10c(s10b, p3_ext[10], c9c, s10c, c10c);
full_adder fa10d(s10c, p4_ext[10], c9d, s10d, c10d);
full_adder fa10e(s10d, p5_ext[10], c9e, s10e, c10e);
full_adder fa10f(s10e, p6_ext[10], c9f, s10f, c10f);
full_adder fa10g(s10f, p7_ext[10], c9g, c[2], c10g);

```

```

wire s11a, s11b, s11c, s11d, s11e, s11f, c11a, c11b, c11c, c11d, c11e, c11f, c11g;
full_adder fa11a(p0_ext[11], p1_ext[11], c10a, s11a, c11a);
full_adder fa11b(s11a, p2_ext[11], c10b, s11b, c11b);
full_adder fa11c(s11b, p3_ext[11], c10c, s11c, c11c);
full_adder fa11d(s11c, p4_ext[11], c10d, s11d, c11d);
full_adder fa11e(s11d, p5_ext[11], c10e, s11e, c11e);
full_adder fa11f(s11e, p6_ext[11], c10f, s11f, c11f);
full_adder fa11g(s11f, p7_ext[11], c10g, c[3], c11g);

```

```

wire s12a, s12b, s12c, s12d, s12e, s12f, c12a, c12b, c12c, c12d, c12e, c12f, c12g;
full_adder fa12a(p0_ext[12], p1_ext[12], c11a, s12a, c12a);
full_adder fa12b(s12a, p2_ext[12], c11b, s12b, c12b);
full_adder fa12c(s12b, p3_ext[12], c11c, s12c, c12c);
full_adder fa12d(s12c, p4_ext[12], c11d, s12d, c12d);
full_adder fa12e(s12d, p5_ext[12], c11e, s12e, c12e);
full_adder fa12f(s12e, p6_ext[12], c11f, s12f, c12f);
full_adder fa12g(s12f, p7_ext[12], c11g, c[4], c12g);

```

```

wire s13a, s13b, s13c, s13d, s13e, s13f, c13a, c13b, c13c, c13d, c13e, c13f, c13g;
full_adder fa13a(p0_ext[13], p1_ext[13], c12a, s13a, c13a);

```

```

full_adder fa13b(s13a, p2_ext[13], c12b, s13b, c13b);
full_adder fa13c(s13b, p3_ext[13], c12c, s13c, c13c);
full_adder fa13d(s13c, p4_ext[13], c12d, s13d, c13d);
full_adder fa13e(s13d, p5_ext[13], c12e, s13e, c13e);
full_adder fa13f(s13e, p6_ext[13], c12f, s13f, c13f);
full_adder fa13g(s13f, p7_ext[13], c12g, c[5], c13g);

wire s14a, s14b, s14c, s14d, s14e, s14f, c14a, c14b, c14c, c14d, c14e, c14f, c14g;
full_adder fa14a(p0_ext[14], p1_ext[14], c13a, s14a, c14a);
full_adder fa14b(s14a, p2_ext[14], c13b, s14b, c14b);
full_adder fa14c(s14b, p3_ext[14], c13c, s14c, c14c);
full_adder fa14d(s14c, p4_ext[14], c13d, s14d, c14d);
full_adder fa14e(s14d, p5_ext[14], c13e, s14e, c14e);
full_adder fa14f(s14e, p6_ext[14], c13f, s14f, c14f);
full_adder fa14g(s14f, p7_ext[14], c13g, c[6], c14g);

wire s15a, s15b, s15c, s15d, s15e, s15f, c15a, c15b, c15c, c15d, c15e, c15f, c15g;
full_adder fa15a(p0_ext[15], p1_ext[15], c14a, s15a, c15a);
full_adder fa15b(s15a, p2_ext[15], c14b, s15b, c15b);
full_adder fa15c(s15b, p3_ext[15], c14c, s15c, c15c);
full_adder fa15d(s15c, p4_ext[15], c14d, s15d, c15d);
full_adder fa15e(s15d, p5_ext[15], c14e, s15e, c15e);
full_adder fa15f(s15e, p6_ext[15], c14f, s15f, c15f);
full_adder fa15g(s15f, p7_ext[15], c14g, c[7], c15g);

```

```
endmodule
```

Appendix A.7: Datapath

```

module datapath (
    input wire    clk,
    input wire    rst,
    input wire    en_a1, en_a2, en_a3, en_a4,
    input wire    en_b1, en_b2, en_b3, en_b4,
    input wire    en_f1, en_f2, en_f3, en_f4,
    input wire    en_add1_1, en_add1_2, en_add1_3, en_add1_4,
    input wire    en_add2_1, en_add2_2, en_add2_3, en_add2_4,
    input wire    save_c,
    input signed [7:0] din,
    output reg signed [7:0] f1, f2, f3, f4
);

reg cin;

```

```

reg signed [7:0] a1, a2, a3, a4, b1, b2, b3, b4, a_mult, b_mult, a_add1, b_add1, a_add2, b_add2;
wire signed [7:0] c_mult, c_add1, c_add2;
truncated_multiplier_8bit uut (.A(a_mult), .B(b_mult), .c(c_mult));
truncated_adder_8bit add1 (a_add1, b_add1, 1'b0, c_add1);
truncated_adder_8bit add2 (a_add2, b_add2, cin, c_add2);

```

```

always @(posedge clk or posedge rst) begin
  if (rst) begin
    a1 <= 8'd0; a2 <= 8'd0; a3 <= 8'd0; a4 <= 8'd0;
    b1 <= 8'd0; b2 <= 8'd0; b3 <= 8'd0; b4 <= 8'd0;
    a_mult <= 8'd0; b_mult <= 8'd0;
    a_add1 <= 8'd0; b_add1 <= 8'd0; a_add2 <= 8'd0; b_add2 <= 8'd0;
    cin <= 1'b0;
    f1 <= 8'd0; f2 <= 8'd0; f3 <= 8'd0; f4 <= 8'd0;
  end else begin
    // Check each enable independently
    if (en_a1) begin
      if (save_c) a1 <= c_mult;
      else      a1 <= din;
    end

    if (en_a2) begin
      if (save_c) a2 <= c_mult;
      else      a2 <= din;
    end

    if (en_a3) begin
      if (save_c) a3 <= c_mult;
      else      a3 <= din;
    end

    if (en_a4) begin
      if (save_c) a4 <= c_mult;
      else      a4 <= din;
    end

    if (en_b1) begin
      a_mult <= a1;
      b_mult <= din;
      b1 <= din;
    end
    if (en_b2) begin
      a_mult <= a2;

```

```

    b_mult <= din;
    b2    <= din;
end

if (en_b3) begin
    a_mult <= a3;
    b_mult <= din;
    b3    <= din;
end

if (en_b4) begin
    a_mult <= a4;
    b_mult <= din;
    b4    <= din;
end

if (en_add1_1) begin
    a_add1 <= a1; // Put c1 into the addition.
    b_add1 <= din; // Put d1 into the addition.
end
if (en_add1_2) begin
    a_add1 <= a2; // c2
    b_add1 <= din; // d2
end
if (en_add1_3) begin
    a_add1 <= a3; // c3
    b_add1 <= din; // d3
end
if (en_add1_4) begin
    a_add1 <= a4; // c4
    b_add1 <= din; // d4
end

if (en_add2_1) begin
    a_add2 <= c_add1; // e1
    b_add2 <= {~b1[7], ~b1[7:1]};
    cin <= ~b1[0];
end
if (en_add2_2) begin
    a_add2 <= c_add1; // e2
    b_add2 <= {~b2[7], ~b2[7:1]};
    cin <= ~b2[0];
end
if (en_add2_3) begin

```

```

        a_add2 <= c_add1; // e3
        b_add2 <= {~b3[7], ~b3[7:1]};
        cin <= ~b3[0];
    end
    if(en_add2_4) begin
        a_add2 <= c_add1; // e4
        b_add2 <= {~b4[7], ~b4[7:1]};
        cin <= ~b4[0];
    end

    if(en_f1) f1 <= c_add2; // f1
    if(en_f2) f2 <= c_add2; // f2
    if(en_f3) f3 <= c_add2; // f3
    if(en_f4) f4 <= c_add2; // f4

end
end

endmodule

```

Appendix A.8: Controller

```

module controller (
    input wire clk,
    input wire rst,
    // Control signals
    output reg en_a1, en_a2, en_a3, en_a4,
    output reg en_b1, en_b2, en_b3, en_b4,
    output reg en_f1, en_f2, en_f3, en_f4,
    output reg en_add1_1, en_add1_2, en_add1_3, en_add1_4,
    output reg en_add2_1, en_add2_2, en_add2_3, en_add2_4,
    output reg save_c
);

parameter S0=4'b0000, S1=4'b0001, S2=4'b0011, S3=4'b0010,
           S4=4'b0110, S5=4'b0111, S6=4'b1111, S7=4'b1110,
           S8=4'b1100, S9=4'b1101, S10=4'b0101, S11=4'b0100;
reg [3:0] state;

// State register with synchronous reset
always @(posedge clk or posedge rst) begin
    if (rst) begin
        state <= S0;
    end else begin

```

```

case (state)
  S0: state <= S1;
  S1: state <= S2;
  S2: state <= S3;
  S3: state <= S4;
  S4: state <= S5;
  S5: state <= S6;
  S6: state <= S7;
  S7: state <= S8;
  S8: state <= S9;
  S9: state <= S10;
  S10: state <= S11;
  S11: state <= S0;
  default: state <= S0;
endcase
end
end

// Combinational control logic
always @(*) begin
  // Default all signals to 0
  en_a1=0; en_a2=0; en_a3=0; en_a4=0;
  en_b1=0; en_b2=0; en_b3=0; en_b4=0;
  en_f1=0; en_f2=0; en_f3=0; en_f4=0;
  en_add1_1=0; en_add1_2=0; en_add1_3=0; en_add1_4=0;
  en_add2_1=0; en_add2_2=0; en_add2_3=0; en_add2_4=0;
  save_c=0;

  case (state)
    S0: begin en_a1=1; en_add2_4=1; en_f3=1; end
    S1: begin en_a2=1; en_f4=1; end
    S2: begin en_a3=1; end
    S3: begin en_a4=1; end
    S4: begin en_b1=1; end
    S5: begin en_a1=1; save_c=1; en_b2=1; end
    S6: begin en_a2=1; save_c=1; en_b3=1; end
    S7: begin en_a3=1; save_c=1; en_b4=1; end
    S8: begin en_a4=1; save_c=1; en_add1_1=1; end
    S9: begin en_add1_2=1; en_add2_1=1; end
    S10: begin en_add1_3=1; en_add2_2=1; en_f1=1; end
    S11: begin en_add1_4=1; en_add2_3=1; en_f2=1; end
  endcase
end

```

```
endmodule
```

Appendix A.9: Top

```
module top (
    input wire    clk,
    input wire    rst,
    input wire signed [7:0] din,
    output wire signed [7:0] f1, f2, f3, f4
);

    // Control signals
    wire en_a1, en_a2, en_a3, en_a4;
    wire en_b1, en_b2, en_b3, en_b4;
    wire en_f1, en_f2, en_f3, en_f4;
    wire en_add1_1, en_add1_2, en_add1_3, en_add1_4;
    wire en_add2_1, en_add2_2, en_add2_3, en_add2_4;
    wire save_c;

    // Instantiate controller
    controller ctrl_inst (
        .clk(clk),
        .rst(rst),
        .en_a1(en_a1), .en_a2(en_a2), .en_a3(en_a3), .en_a4(en_a4),
        .en_b1(en_b1), .en_b2(en_b2), .en_b3(en_b3), .en_b4(en_b4),
        .en_f1(en_f1), .en_f2(en_f2), .en_f3(en_f3), .en_f4(en_f4),
        .en_add1_1(en_add1_1), .en_add1_2(en_add1_2), .en_add1_3(en_add1_3), .en_add1_4(en_add1_4),
        .en_add2_1(en_add2_1), .en_add2_2(en_add2_2), .en_add2_3(en_add2_3), .en_add2_4(en_add2_4),
        .save_c(save_c)
    );

    // Instantiate datapath
    datapath dp_inst (
        .clk(clk),
        .rst(rst),
        .en_a1(en_a1), .en_a2(en_a2), .en_a3(en_a3), .en_a4(en_a4),
        .en_b1(en_b1), .en_b2(en_b2), .en_b3(en_b3), .en_b4(en_b4),
        .en_f1(en_f1), .en_f2(en_f2), .en_f3(en_f3), .en_f4(en_f4),
        .en_add1_1(en_add1_1), .en_add1_2(en_add1_2), .en_add1_3(en_add1_3), .en_add1_4(en_add1_4),
        .en_add2_1(en_add2_1), .en_add2_2(en_add2_2), .en_add2_3(en_add2_3), .en_add2_4(en_add2_4),
        .save_c(save_c),
        .din(din),
        .f1(f1), .f2(f2), .f3(f3), .f4(f4)
    );
endmodule
```

```
endmodule
```

Appendix B: Testbenches

Appendix B.1: Truncated Adder (8-bit) Testbench

```
`timescale 1ns / 1ps

module tb_truncated_adder_8bit;

    reg signed [7:0] a;
    reg signed [7:0] b;
    reg cin;
    wire signed [7:0] sum_trunc;
    // wire signed [8:0] full_sum;

    // DUT instance
    truncated_adder_8bit dut (
        .a(a),
        .b(b),
        .c0(cin),
        .sum_trunc(sum_trunc)
        // .full_sum(full_sum)
    );

    initial begin
        // Monitor prints whenever any displayed signal changes
        $monitor("TIME=%0t a=%d (%b) b=%d (%b) cin=%d sum_trunc=%b",
            $time, a, a, b, b, cin, sum_trunc);

        cin = 1'b0;
        // Directed Tests (with expected full_sum in decimal and sum_trunc in top 8 bits binary)
        a = 8'sh7F; b = 8'sh01; #1; // 127 + 1 = 128 then trunc 010000000 = 01000000
        a = 8'sh80; b = 8'shFF; #1; // -128 + -1 = -129 then trunc 101111111 = 10111111
        a = 8'sh7F; b = 8'sh7F; #1; // 127 + 127 = 254 then trunc 011111110 = 01111111
        a = 8'sh80; b = 8'sh80; #1; // -128 + -128 = -256 then trunc 100000000 = 10000000
        a = 8'sh00; b = 8'sh00; #1; // 0 + 0 = 0 then trunc 000000000 = 00000000

        a = 8'sh01; b = 8'shFF; #1; // 1 + -1 = 0 then trunc 000000000 = 00000000
        a = 8'sh7E; b = 8'sh02; #1; // 126 + 2 = 128 then trunc 010000000 = 01000000
        a = 8'shFF; b = 8'sh01; #1; // -1 + 1 = 0 then trunc 000000000 = 00000000
        a = 8'shFF; b = 8'shFF; #1; // -1 + -1 = -2 then trunc 111111110 = 11111111
    end
endmodule
```

```

// Boundary conditions
a = 8'sh00; b = 8'sh01; #1; // 0 + 1 = 1 then trunc 000000001 = 00000000
a = 8'shFF; b = 8'sh02; #1; // -1 + 2 = 1 then trunc 000000001 = 00000000
a = 8'sh7F; b = 8'sh81; #1; // 127 + -127 = 0 then trunc 000000000 = 00000000

cin = 1'b1;
a = 8'sh7F; b = 8'sh01; #1; // 127 + 1 + carry = 129 then trunc 010000001 = 01000000
a = 8'sh80; b = 8'shFF; #1; // -128 + -1 + carry = -128 then trunc 110000000 = 11000000

a = 8'sh00; b = 8'sh00; #1; // 0 + 0 + carry = 1 then trunc 000000001 = 00000000
a = 8'shFF; b = 8'sh01; #1; // -1 + 1 + carry = 1 then trunc 000000001 = 00000000

a = 8'sh7F; b = 8'sh7F; #1; // 127 + 127 + carry = 255 then trunc 011111111 = 01111111

// Boundary conditions with cin = 1
a = 8'sh00; b = 8'sh01; #1; // 0 + 1 + carry = 2 then trunc 000000010 = 00000001
a = 8'shFF; b = 8'sh02; #1; // -1 + 2 + carry = 2 then trunc 000000010 = 00000001

a = 8'sh7F; b = 8'sh81; #1; // 127 + -127 + carry = 1 then trunc 000000001 = 00000000

cin = 1'b0;
// Random tests
repeat (10) begin
    a = $random;
    b = $random;
    #1;
end
$finish;
end

endmodule

```

Appendix B.2: Truncated Multiplier (8-bit) Testbench

```

`timescale 1ns/1ps

module tb_multiplier_8bit_signed;

    reg signed [7:0] a, b;
    // wire signed [15:0] product;
    // wire [7:0] p0,p1,p2,p3,p4,p5,p6,p7;
    // wire [15:0] p0_ext,p1_ext,p2_ext,p3_ext,p4_ext,p5_ext,p6_ext,p7_ext;
    // wire [15:0] extra_add;

```

```

wire [7:0] c;

// Instantiate the multiplier
truncated_multiplier_8bit uut (
    .A(a),
    .B(b),
    .c(c)
);

initial begin
    // Print header
    //
    $display("Time\tA\tB\tp0\tp1\tp2\tp3\tp4\tp5\tp6\tp7\tp0_ext\tp1_ext\tp2_ext\tp3_ext\tp4_ext\tp5_ext\tp6_ext\tp7_ext\textra_add\tproduct");
    $monitor(
        "%0dns\nA=%08b\nB=%08b\nprod_o=%b\n\n",
        $time, a, b, c
    );

    a = 8'sd0; b = 8'sd0; #1; // 0 * 0 = 0
    a = 8'sd1; b = 8'sd1; #1; // 1 * 1 = 1
    a = 8'sd127; b = 8'sd127; #1; // max positive * max positive
    a = -8'sd128; b = -8'sd128; #1; // min negative * min negative
    a = -8'sd128; b = 8'sd127; #1; // min negative * max positive
    a = 8'sd127; b = -8'sd128; #1; // max positive * min negative
    a = -8'sd1; b = -8'sd1; #1; // -1 * -1 = 1
    a = -8'sd1; b = 8'sd1; #1; // -1 * 1 = -1
    a = 8'sd1; b = -8'sd1; #1; // 1 * -1 = -1

    // -----
    // Boundary tests around sign bit
    // -----
    a = 8'sd127; b = 8'sd2; #1; // 127*2 = 254
    a = -8'sd128; b = 8'sd2; #1; // -128*2 = -256
    a = 8'sd64; b = -8'sd64; #1; // positive * negative
    a = -8'sd64; b = 8'sd64; #1; // negative * positive
    a = -8'sd64; b = -8'sd64; #1; // negative * negative

    repeat (10) begin
        a = $random;
        b = $random;
        #1;
    end

```

```

        end

        #5 $finish;
    end

endmodule

```

Appendix B.3: Datapath Testbench

```

`timescale 1ns/1ps

module datapath_tb;

    // DUT inputs
    reg clk;
    reg rst;
    reg en_a1, en_a2, en_a3, en_a4;
    reg en_b1, en_b2, en_b3, en_b4;
    reg en_f1, en_f2, en_f3, en_f4;
    reg en_add1_1, en_add1_2, en_add1_3, en_add1_4;
    reg en_add2_1, en_add2_2, en_add2_3, en_add2_4;
    reg save_c;
    reg signed [7:0] din;

    // DUT outputs
    wire signed [7:0] f1, f2, f3, f4;

    // Instantiate DUT
    datapath uut (
        .clk(clk), .rst(rst),
        .en_a1(en_a1), .en_a2(en_a2), .en_a3(en_a3), .en_a4(en_a4),
        .en_b1(en_b1), .en_b2(en_b2), .en_b3(en_b3), .en_b4(en_b4),
        .en_f1(en_f1), .en_f2(en_f2), .en_f3(en_f3), .en_f4(en_f4),
        .en_add1_1(en_add1_1), .en_add1_2(en_add1_2),
        .en_add1_3(en_add1_3), .en_add1_4(en_add1_4),
        .en_add2_1(en_add2_1), .en_add2_2(en_add2_2),
        .en_add2_3(en_add2_3), .en_add2_4(en_add2_4),
        .save_c(save_c),
        .din(din),
        .f1(f1), .f2(f2), .f3(f3), .f4(f4)
    );

    // Clock generation

```

```

always #5 clk = ~clk;

// Task to clear all enables
task clear_enables;
begin
    en_a1=0; en_a2=0; en_a3=0; en_a4=0;
    en_b1=0; en_b2=0; en_b3=0; en_b4=0;
    en_f1=0; en_f2=0; en_f3=0; en_f4=0;
    en_add1_1=0; en_add1_2=0; en_add1_3=0; en_add1_4=0;
    en_add2_1=0; en_add2_2=0; en_add2_3=0; en_add2_4=0;
    save_c=0;
end
endtask

initial begin
    $display("Starting datapath testbench...");
    $monitor("T=%0t | din=%d | f1=%d f2=%d f3=%d f4=%d | a1=%d a2=%d a3=%d a4=%d | b1=%d
b2=%d b3=%d b4=%d",
        $time, din, f1, f2, f3, f4,
        uut.a1, uut.a2, uut.a3, uut.a4,
        uut.b1, uut.b2, uut.b3, uut.b4);
    clk = 0;
    rst = 1;
    clear_enables;
    din = 0;

    // Reset pulse
    #10 rst = 0;

    // Test loading a1–a4 with positive values
    din = 8'd10; en_a1=1; #10; en_a1=0;
    din = 8'd20; en_a2=1; #10; en_a2=0;
    din = 8'd30; en_a3=1; #10; en_a3=0;
    din = 8'd40; en_a4=1; #10; en_a4=0;

    // Test loading b1–b4 with negative values
    din = -8'sd5; en_b1=1; #10; en_b1=0;
    din = -8'sd15; en_b2=1; #10; en_b2=0;
    din = -8'sd25; en_b3=1; #10; en_b3=0;
    din = -8'sd35; en_b4=1; #10; en_b4=0;

    // Test multiplier save_c functionality
    din = 8'd3; en_b1=1; #10; en_b1=0;
    save_c=1; en_a1=1; #10; en_a1=0; save_c=0;

```

```

// Test add1 paths
din = 8'd7; en_add1_1=1; #10; en_add1_1=0;
din = 8'd8; en_add1_2=1; #10; en_add1_2=0;
din = 8'd9; en_add1_3=1; #10; en_add1_3=0;
din = 8'd10; en_add1_4=1; #10; en_add1_4=0;

// Test add2 paths
en_add2_1=1; #10; en_add2_1=0;
en_add2_2=1; #10; en_add2_2=0;
en_add2_3=1; #10; en_add2_3=0;
en_add2_4=1; #10; en_add2_4=0;

// Test outputs f1-f4
en_f1=1; #10; en_f1=0;
en_f2=1; #10; en_f2=0;
en_f3=1; #10; en_f3=0;
en_f4=1; #10; en_f4=0;

// Corner case: maximum positive input
din = 8'sd127; en_a1=1; #10; en_a1=0;
// Corner case: minimum negative input
din = -8'sd128; en_a2=1; #10; en_a2=0;

// Corner case: overlapping enables
din = 8'd55; en_a3=1; en_b3=1; #10; clear_enables;

// Corner case: simultaneous add1 and add2
din = 8'd12; en_add1_4=1; en_add2_4=1; #10; clear_enables;

// Finish simulation
#50;
$display("Testbench completed.");
$finish;
end

endmodule

```

Appendix B.4: Controller Testbench

```

`timescale 1ns/1ps

module controller_tb;

```

```

// DUT inputs
reg clk;
reg rst;

// DUT outputs
wire en_a1, en_a2, en_a3, en_a4;
wire en_b1, en_b2, en_b3, en_b4;
wire en_f1, en_f2, en_f3, en_f4;
wire en_add1_1, en_add1_2, en_add1_3, en_add1_4;
wire en_add2_1, en_add2_2, en_add2_3, en_add2_4;
wire save_c;

// Instantiate DUT
controller uut (
    .clk(clk), .rst(rst),
    .en_a1(en_a1), .en_a2(en_a2), .en_a3(en_a3), .en_a4(en_a4),
    .en_b1(en_b1), .en_b2(en_b2), .en_b3(en_b3), .en_b4(en_b4),
    .en_f1(en_f1), .en_f2(en_f2), .en_f3(en_f3), .en_f4(en_f4),
    .en_add1_1(en_add1_1), .en_add1_2(en_add1_2),
    .en_add1_3(en_add1_3), .en_add1_4(en_add1_4),
    .en_add2_1(en_add2_1), .en_add2_2(en_add2_2),
    .en_add2_3(en_add2_3), .en_add2_4(en_add2_4),
    .save_c(save_c)
);

// Clock generation
always #5 clk = ~clk;

// Monitor signals
initial begin
    $monitor("T=%0t | rst=%b | en_a1=%b en_a2=%b en_a3=%b en_a4=%b | en_b1=%b en_b2=%b
en_b3=%b en_b4=%b | en_f1=%b en_f2=%b en_f3=%b en_f4=%b | add1_1=%b add1_2=%b
add1_3=%b add1_4=%b | add2_1=%b add2_2=%b add2_3=%b add2_4=%b | save_c=%b",
        $time, rst,
        en_a1,en_a2,en_a3,en_a4,
        en_b1,en_b2,en_b3,en_b4,
        en_f1,en_f2,en_f3,en_f4,
        en_add1_1,en_add1_2,en_add1_3,en_add1_4,
        en_add2_1,en_add2_2,en_add2_3,en_add2_4,
        save_c);
end

initial begin
    $display("Starting controller testbench...");

```

```

    clk = 0;
    rst = 1;

    // Apply reset
    #12 rst = 0;

    // Let FSM cycle through all states twice
    repeat(24) @(posedge clk);

    // Corner case: assert reset mid-cycle
    #2 rst = 1;
    #10 rst = 0;

    // Let FSM run again
    repeat(12) @(posedge clk);

    // Finish simulation
    #20;
    $display("Controller testbench completed.");
    $finish;
end

endmodule

```

Appendix B.5: Top Testbench

```

`timescale 1ns/1ps

module tb_top;

    reg    clk;
    reg    rst;
    reg signed [7:0] din;

    wire signed [7:0] f1, f2, f3, f4;

    // Instantiate DUT
    top dut (
        clk, rst, din, f1, f2, f3, f4
    );

    // Clock generator: 10 ns period
    always #5 clk = ~clk;

```

```

initial begin
    $monitor("T=%d | rst=%b | din=%0d | f1=%0d f2=%0d f3=%0d f4=%0d",
        $time, rst, din, f1, f2, f3, f4);

    // Initial values
    clk = 0;
    din = 8'd1;
    rst = 0;

    #0.001;
    rst = 1;

    #20; rst = 0; // Hold reset

    // Initial group
    din = 8'd0;    // 0
    #10 din = 8'd1; // +1
    #10 din = 8'd127; // +127
    #10 din = -8'd128; // 8'h80 interpreted as -128

    #10 din = 8'd0;    // 0
    #10 din = 8'd1;    // +1
    #10 din = 8'd127;  // +127
    #10 din = -8'd128; // -128

    #10 din = 8'd0;    // 0
    #10 din = 8'd1;    // +1
    #10 din = 8'd0;    // 0 // f1 = 0
    #10 din = 8'd0;    // 0 // f2 = -1

    // --- A values for 4 consecutive vectors ---
    #10 din = 8'd127;  // a for vector 5, f3 = -17
    #10 din = -8'd128; // a for vector 6, f4 = 48
    #10 din = 8'd0;    // a for vector 7
    #10 din = 8'd127;  // a for vector 8

    // --- B values for those 4 vectors ---
    #10 din = -8'd128; // b for vector 5
    #10 din = 8'd127;  // b for vector 6
    #10 din = 8'd127;  // b for vector 7
    #10 din = 8'd0;    // b for vector 8

    // --- D values for those 4 vectors ---
    #10 din = 8'd0;    // d for vector 5

```

```

#10 din = 8'd0;    // d for vector 6
#10 din = 8'd16;   // d for vector 7, f1 = 16
#10 din = 8'd32;   // d for vector 8, f2 = -48

// --- Next group of 4 vectors (9-12) ---
// A values
#10 din = 8'd0;    // a for vector 9, f3 = -28
#10 din = -8'd128; // a for vector 10, f4 = 8
#10 din = 8'd127;  // a for vector 11
#10 din = -8'd1;   // a for vector 12 (8'hFF = -1)

// B values
#10 din = -8'd128; // b for vector 9
#10 din = 8'd0;    // b for vector 10
#10 din = -8'd1;   // b for vector 11
#10 din = 8'd127;  // b for vector 12

// D values
#10 din = -8'd16;  // d for vector 9 (8'hF0 = -16)
#10 din = -8'd15; // d for vector 10 (8'hF1 = -15)
#10 din = -8'd128; // d for vector 11, f1 = 28
#10 din = -8'd128; // d for vector 12, f2 = -4

// A values
#10 din = -8'd128; // a for vector 13 (0x80), f3 = -33
#10 din = -8'd1;   // a for vector 14 (0xFF), f4 = -65
#10 din = 8'd64;   // a for vector 15 (0x40)
#10 din = 8'd96;   // a for vector 16 (0x60)

// B values
#10 din = -8'd1;   // b for vector 13 (0xFF)
#10 din = -8'd128; // b for vector 14 (0x80)
#10 din = 8'd64;   // b for vector 15 (0x40)
#10 din = 8'd96;   // b for vector 16 (0x60)

// D values
#10 din = 8'd127;  // d for vector 13 (0x7F)
#10 din = 8'd127;  // d for vector 14 (0x7F)
#10 din = 8'd0;    // d for vector 15 (0x00), f1 = 31
#10 din = 8'd0;    // d for vector 16 (0x00), f2 = 63

// A values
#10 din = 8'd80;   // a for vector 17 (0x50), f3 = -12

```

```
#10 din = -8'd112; // a for vector 18 (0x90), f4 = -15
#10 din = 8'd127; // a for vector 19 (0x7F)
#10 din = 8'd127; // a for vector 20 (0x7F)
```

```
// B values
```

```
#10 din = -8'd48; // b for vector 17 (0xD0)
#10 din = -8'd112; // b for vector 18 (0x90)
#10 din = 8'd2; // b for vector 19 (0x02)
#10 din = 8'd127; // b for vector 20 (0x7F)
```

```
// D values
```

```
#10 din = 8'd0; // d for vector 17
#10 din = 8'd0; // d for vector 18
#10 din = 8'd127; // d for vector 19, f1 = 8
#10 din = 8'd127; // d for vector 20, f2 = 40
```

```
// A values
```

```
#10 din = -8'd128; // a for vector 21 (0x80), f3 = 31
#10 din = -8'd128; // a for vector 22 (0x80), f4 = 15
#10 din = 8'd1; // a for vector 23 (0x01)
#10 din = 8'd2; // a for vector 24 (0x02)
```

```
// B values
```

```
#10 din = 8'd2; // b for vector 21
#10 din = -8'd128; // b for vector 22
#10 din = -8'd16; // b for vector 23 (0xF0)
#10 din = -8'd16; // b for vector 24 (0xF0)
```

```
// D values
```

```
#10 din = -8'd128; // d for vector 21
#10 din = -8'd128; // d for vector 22
#10 din = 8'd16; // d for vector 23 (0x10), f1 = -33
#10 din = 8'd0; // d for vector 24 (0x00), f2 = 16
```

```
// A values
```

```
#10 din = 8'd3; // a for vector 25, f3 = 7
#10 din = 8'd16; // a for vector 26, f4 = 3
#10 din = 8'd1; // a for vector 27
#10 din = 8'd85; // a for vector 28
```

```
// B values
```

```
#10 din = -8'd16; // b for vector 25 (0xF0)
#10 din = 8'd15; // b for vector 26 (0x0F)
#10 din = -8'd128; // b for vector 27 (0x80), f1 = 3
```

```
#10 din = 8'd85;    // b for vector 28 (0x55), f2 = -36
```

```
// D values
```

```
#10 din = -8'd1;    // d for vector 25 (0xFF), f3 = 63
```

```
#10 din = -8'd128;  // d for vector 26 (0x80), f4 = 6
```

```
#10 din = 8'd127;   // d for vector 27 (0x7F)
```

```
#10 din = 8'd85;    // d for vector 28 (0x55)
```

```
// A values
```

```
#10 din = -8'd86;   // a for vector 29 (0xAA)
```

```
#10 din = 8'd85;    // a for vector 30 (0x55)
```

```
#10 din = -8'd86;   // a for vector 31 (0xAA)
```

```
#10 din = 8'd1;     // a for vector 32 (0x01)
```

```
// B values
```

```
#10 din = -8'd86;   // b for vector 29 (0xAA)
```

```
#10 din = -8'd86;   // b for vector 30 (0xAA)
```

```
#10 din = 8'd85;    // b for vector 31 (0x55)
```

```
#10 din = -8'd128;  // b for vector 32 (0x80)
```

```
// D values
```

```
#10 din = -8'd86;   // d for vector 29 (0xAA)
```

```
#10 din = 8'd85;    // d for vector 30 (0x55)
```

```
#10 din = -8'd86;   // d for vector 31 (0xAA), f1 = 7
```

```
#10 din = 8'd127;   // d for vector 32 (0x7F), f2 = 35
```

```
// A values
```

```
#10 din = 8'd127;   // a for vector 33 (0x7F), f3 = -51
```

```
#10 din = -8'd128;  // a for vector 34 (0x80), f4 = 63
```

```
#10 din = -8'd1;    // a for vector 35 (0xFF)
```

```
#10 din = 8'd19;    // a for vector 36 (0x13)
```

```
// B values
```

```
#10 din = 8'd1;     // b for vector 33 (0x01)
```

```
#10 din = 8'd1;     // b for vector 34 (0x01)
```

```
#10 din = 8'd1;     // b for vector 35 (0x01)
```

```
#10 din = -8'd25;   // b for vector 36 (0xE7)
```

```
// D values
```

```
#10 din = -8'd128;  // d for vector 33 (0x80)
```

```
#10 din = 8'd127;   // d for vector 34 (0x7F)
```

```
#10 din = -8'd1;    // d for vector 35 (0xFF), f1 = -33
```

```
#10 din = -8'd92;   // d for vector 36 (0xA4), f2 = 31
```

```

// A values
#10 din = 8'd107; // a for vector 37 (0x6B), f3 = -1
#10 din = -8'd13; // a for vector 38 (0xF3), f4 = -18
#10 din = 8'd32; // a for vector 39 (0x20)
#10 din = 8'd16; // a for vector 40 (0x10)

// B values
#10 din = -8'd110; // b for vector 37 (0x92)
#10 din = 8'd12; // b for vector 38 (0x0C)
#10 din = 8'd4; // b for vector 39 (0x04)
#10 din = -8'd16; // b for vector 40 (0xF0)

// D values
#10 din = 8'd16; // d for vector 37 (0x10)
#10 din = 8'd126; // d for vector 38 (0x7E)
#10 din = -8'd128; // d for vector 39 (0x80), f1 = 20
#10 din = 8'd15; // d for vector 40 (0x0F), f2 = 28

// f3 = -33
// f4 = 7
#30 rst = 1;

#10;
$finish;
end

endmodule

```