ADC.h

The provided code is a Verilog module named `ADC` that handles analog-to-digital conversion (ADC) and manages data output. Here's a line-by-line breakdown of the code:

1. `timescale 1ns / 1ps`: Specifies the time unit for the simulation (`1ns` as the time unit and `1ps` as the precision).

2. `module ADC(clk, enable, I1, I2, I3, conv, done, ADC_in, ADC_out, data_out);`: Declares the module named `ADC` with input and output ports. The key ports are:
   - `clk`: Clock signal for synchronization.
   - `enable`: Enables the ADC process.
   - `I1, I2, I3`: Input signals from the sensors (FSR in this case).
   - `conv`: Conversion signal for ADC.
   - `done`: Signals when ADC conversion is complete.
   - `ADC_in`: Input to the ADC module.
   - `ADC_out`: Digital output from the ADC.
   - `data_out`: Final sampled output data (10 bits).

3. `input I1, I2, I3, clk, enable, ADC_out;`: Declares the input signals.

4. `output reg ADC_in;`: Declares `ADC_in` as a registered output.

5. `output reg conv, done;`: Declares `conv` and `done` as registered outputs.

6. `output reg [9:0] data_out;`: Declares `data_out` as a 10-bit register for sampled data.

7. `reg [2:0] Cs = 0;`: Declares a 3-bit state variable `Cs`, initialized to 0, which controls the state machine.

8. `reg [4:0] count = 0;`: Declares a 5-bit register `count`, used to track the bit positions during sampling.

9. `reg [4:0] bit_pos = 0;`: Declares `bit_pos`, which tracks the bit position for shifting data.

10. `always @(negedge clk) begin`: Triggers the block on the falling edge of the clock signal (`clk`).

11. `if (enable == 1) begin`: If `enable` is high, the ADC process starts.

12. `case (Cs)`: This case statement represents a finite state machine controlling the conversion process.

- State 0:
  - Initializes the conversion process by setting `conv = 0`, `done = 0`, and `ADC_in = 1`.
  - Increments the state and `count`.

- State 1, 2, 3, 4:
  - Same initialization (`conv = 0`, `done = 0`).
  - Sets `ADC_in` to the values of `I1`, `I2`, and `I3` respectively.
  - Moves to the next state after incrementing `count`.

- State 5:
  - If `count` is less than 8, it keeps `ADC_in = 0` and continues sampling.
  - Otherwise, it shifts the `ADC_out` bit into `data_out` based on the `bit_pos`.

- State 6:
  - If `count` is less than 17, it keeps sampling.
  - Otherwise, the process is marked as done (`done = 1`) and the data is stored.

- State 7:
  - Final state: Marks the conversion complete by setting `conv = 1` and resetting the state machine.

13. `default`: Resets all signals to 0 when no other state is active.

14. `else begin`: If `enable` is not active, all internal signals (`conv`, `done`, `Cs`, `count`, `data_out`) are reset.

This module uses a finite state machine to convert the analog signal into a digital one, and the result is stored in `data_out`.

For your FSR setup:
1. Connect the FSR to the ADC input pins (`I1`, `I2`, `I3`).
2. The ADC will sample the analog signals based on the clock signal and store the digital result in `data_out`.

Let me know if you need further clarification or specific details for implementation!

adc_tb.v

Here's a line-by-line explanation of the `adc_tb.v` file, which is a Verilog testbench used to simulate and verify the behavior of the `ADC` module:

1. ```verilog
   `timescale 1ns / 1ps
   ```
   - This sets the time unit for the simulation. Here, `1ns` is the time unit, and `1ps` is the precision.

2. ```verilog
   module ADC_tb;
   ```
   - Declares the `ADC_tb` testbench module, where the behavior of the `ADC` module will be tested.

3. ```verilog
   reg clk;
   reg enable;
   reg ADC_out;
   reg I1;
   reg I2;
   reg I3;
   ```
   ```

- These are the inputs for the `ADC_tb` module, declared as registers (`reg`), which can hold values and be used to simulate input signals:
  - `clk`: Clock signal for timing.
  - `enable`: Enables the `ADC` process.
  - `ADC_out`: Simulated output from the ADC (used for feedback).
  - `I1`, `I2`, `I3`: Input signals from sensors or other sources, which are typically FSR signals in your case.

4. ```verilog
   wire conv;
   wire done;
   wire ADC_in;
   wire [9:0] data_out;
   ```
   - These are the output signals of the testbench, declared as wires (`wire`), which connect to the outputs of the ADC module:
     - `conv`: Conversion signal from the ADC.
     - `done`: Indicates when the ADC process is complete.
     - `ADC_in`: Input to the ADC module.
     - `data_out`: The 10-bit digital output from the ADC.

5. ```verilog
   ADC_top uut (
       .clk(clk),
       .enable(enable),
       .I1(I1),
       .I2(I2),
       .I3(I3),
       .conv(conv),
       .done(done),
       .ADC_in(ADC_in),
       .ADC_out(ADC_out),
       .data_out(data_out)
   );
   ```
   - Instantiates the `ADC_top` module (Unit Under Test, `uut`) and connects the inputs and outputs of the testbench to the corresponding ports of the ADC module.

6. ```verilog
   always #5 clk = ~clk;
   ```
   - This block generates a clock signal with a period of 10 ns (5 ns high, 5 ns low). It toggles the clock every 5 ns.

7. ```verilog
   initial begin
   ```
   - This block runs once at the start of the simulation and defines the input stimulus for testing.

8. ```verilog
       clk = 0;
       enable = 1;

```
    ADC_out = 0;
    I1 = 0;
    I2 = 0;
    I3 = 0;
```
- Initializes all inputs at the beginning of the simulation:
  - `clk` is set to 0 (low).
  - `enable` is set to 1, enabling the ADC process.
  - `ADC_out`, `I1`, `I2`, `I3` are set to 0.

9. ```verilog
    #100;
```
- Waits for 100 ns before applying the next set of input changes.

10. The following lines introduce input stimulus by changing the value of `ADC_out` at various time intervals:
```verilog
    ADC_out = 0;
    #10;
    ADC_out = 1;
    #10;
    ADC_out = 1;
    #10;
    ...
```
- This applies a sequence of changes to `ADC_out`, simulating the digital signal generated by the ADC module during conversion.
- Each change occurs with a 10 ns delay (`#10`), which allows you to observe how the `ADC_top` module behaves when the `ADC_out` signal changes.

11. ```verilog
    // Add stimulus here
```
- This comment indicates that additional input stimuli or test cases can be added to further verify the module's behavior.

12. ```verilog
  end
```
- Ends the `initial` block, finishing the stimulus generation for the testbench.

13. ```verilog
  endmodule
```
- Ends the definition of the testbench module.

### Summary:
- The testbench `ADC_tb` simulates the behavior of the `ADC` module by providing it with input signals (`clk`, `enable`, `ADC_out`, `I1`, `I2`, `I3`).
- The clock signal is generated automatically using the `always` block.

- The `initial` block sets up a sequence of `ADC_out` values to test how the `ADC_top` module responds to different input conditions.

Let me know if you need more information or clarification!

**ADC_top.v**

Here is a line-by-line explanation of the `ADC_top.v` file, which defines the top-level module of the ADC system:

1. ```verilog
   `timescale 1ns / 1ps
   ```
   - Specifies the time unit and precision for simulation (`1ns` is the time unit, and `1ps` is the precision).

2. ```verilog
   module ADC_top(clk, enable, I1, I2, I3, ADC_in, ADC_out, ADCclk, conv, done, data_out);
   ```
   - Declares the `ADC_top` module, which is the top-level module for controlling the ADC.
   - Inputs:
     - `clk`: Clock signal for the system.
     - `enable`: Enables the ADC conversion process.
     - `I1`, `I2`, `I3`: Inputs from the sensors (Force Sensitive Resistors, for example).
     - `ADC_out`: The output from the ADC indicating the digital converted value.
   - Outputs:
     - `ADC_in`: Input to the ADC module.
     - `ADCclk`: Clock signal for the ADC.
     - `conv`: Conversion signal that indicates when ADC conversion is happening.
     - `done`: Signal that indicates when ADC conversion is complete.
     - `data_out`: 10-bit digital output from the ADC.

3. ```verilog
   input clk, enable, I1, I2, I3, ADC_out;
   ```
   - Declares the input ports for the module.

4. ```verilog
   output conv, done, ADC_in, ADCclk;
   output [9:0] data_out;
   ```
   - Declares the output ports for the module. `data_out` is a 10-bit bus, meaning it can hold 10 bits of data, which is the digital output from the ADC.

5. ```verilog
   wire tc;
   wire [31:0] out;
   wire [9:0] data_out1;
   ```
   - Declares internal wires:
     - `tc`: An internal wire used for timing control.
     - `out`: A 32-bit wire used for dividing the clock signal.
   ```

- `data_out1`: A 10-bit wire used to temporarily store data before final output.

6. ```verilog
   //clock division.....
   cnt1 counter(out, 32'b0, tc, enable, clk, tc, 32'b11111111111111111111111111111111);
   ```
   - This instantiates a clock division module (`cnt1`) to slow down the clock for the ADC. The 32-bit wire `out` receives the divided clock signal. It takes in:
     - `out`: Output of the clock divider.
     - `32'b0`: Initial value for the counter.
     - `tc`: Timing control signal.
     - `enable`: Enable signal for the ADC process.
     - `clk`: The main clock signal.
     - `32'b11111111111111111111111111111111`: A constant 32-bit value to control the frequency division.

7. ```verilog
   ADC m1(ADCclk, enable, I1, I2, I3, conv, done, ADC_in, ADC_out, data_out1);
   ```
   - This instantiates the `ADC` module, which performs the actual analog-to-digital conversion. The ADC receives signals:
     - `ADCclk`: The clock for the ADC.
     - `enable`: Enables the ADC operation.
     - `I1`, `I2`, `I3`: Input sensor signals.
     - `conv`, `done`, `ADC_in`, `ADC_out`: Control signals for the ADC conversion.
     - `data_out1`: 10-bit temporary digital output from the ADC.

8. ```verilog
   assign ADCclk = out[6];
   ```
   - Assigns the 7th bit (`out[6]`) from the clock divider to `ADCclk`, providing a slower clock signal for the ADC.

9. ```verilog
   fdc10 m2(data_out1, ADCclk, done, 1'b0, data_out);
   ```
   - Instantiates another module `fdc10` to process the data and handle the final digital output:
     - `data_out1`: The 10-bit data from the ADC.
     - `ADCclk`: Clock signal for synchronization.
     - `done`: Indicates when the ADC conversion is complete.
     - `1'b0`: A constant value of 0, potentially a reset signal for `fdc10`.
     - `data_out`: The final 10-bit output after processing.

10. ```verilog
    endmodule
    ```
    - Ends the module definition.

Summary:
- The `ADC_top` module is the top-level controller that integrates the ADC functionality. It divides the clock signal, manages the conversion process, and outputs the 10-bit digital result (`data_out`).
- The clock divider (`cnt1`) generates a slower clock (`ADCclk`) for the ADC module to operate.
    ```

- The `ADC` module performs the analog-to-digital conversion of the input signals (`I1`, `I2`, `I3`), and the `fdc10` module processes the digital output.

Let me know if you need further clarification or details!

Cnt1.v

Here's a line-by-line explanation of the `cnt1.v` file, which implements a counter module used for clock division:

1. ```verilog
   `timescale 1ns / 1ps
   ```
   - Sets the time unit and precision for the simulation (`1ns` for the time unit, and `1ps` for the precision).

2. ```verilog
   module cnt1(out, data, load, en, clk, tc, lmt);
   ```
   - Declares the `cnt1` module, which is a 32-bit counter. It takes input signals and produces output signals.
   - Inputs:
     - `data`: 32-bit input data used for loading a value into the counter.
     - `load`: A control signal to load the value of `data` into the counter.
     - `en`: Enable signal to control whether the counter increments.
     - `clk`: Clock signal to drive the counter's operation.
     - `lmt`: 32-bit limit value for the counter to compare against.
   - Outputs:
     - `out`: 32-bit output value representing the current count.
     - `tc`: A signal that becomes high (`1`) when the counter reaches the limit (`lmt`).

3. ```verilog
   output [31:0] out;
   output reg tc;
   ```
   - Declares the 32-bit output `out`, which stores the current value of the counter.
   - Declares the output `tc` as a register (`reg`), which will hold the value indicating when the counter reaches the limit.

4. ```verilog
   input [31:0] data;
   input load, en, clk;
   input [31:0] lmt;
   ```
   - Declares the input ports for the module:
     - `data`: 32-bit value to be loaded into the counter.
     - `load`: Control signal to trigger the loading of `data` into the counter.
     - `en`: Enable signal that controls whether the counter increments on each clock cycle.
     - `clk`: Clock signal to synchronize the counter's operation.
     - `lmt`: 32-bit limit value to compare the counter's value against.

5. ```verilog
   reg [31:0] out;
   ```
   - Declares the internal register `out`, which holds the 32-bit current count value.

6. ```verilog
   parameter reset = 0;
   ```
   - Declares a parameter `reset` with a value of 0. This parameter is used to reset the counter.

7. ```verilog
   initial begin
       out = 32'b0;
       tc = 0;
   end
   ```
   - Initializes the counter value `out` to 0 and the terminal count (`tc`) to 0 when the module is first instantiated.

8. ```verilog
   always @(negedge clk)
   ```
   - This block triggers on the falling edge of the clock signal (`clk`).

9. ```verilog
   if (reset) begin
      out <= 32'b0;
   end else if (load) begin
      out <= data;
   end else if (en) begin
      out <= out + 32'b00000000000000000000000000000001;
   end else begin
      out <= out;
   end
   ```
   - Describes how the counter behaves on the falling edge of the clock:
     - If `reset` is high (1), the counter is reset to 0 (`out <= 32'b0`).
     - If `load` is high (1), the counter is loaded with the value in `data` (`out <= data`).
     - If `en` (enable) is high (1), the counter increments by 1 (`out <= out + 1`).
     - Otherwise, the counter retains its current value (`out <= out`).

10. ```verilog
    always @(posedge clk)
    ```
    - This block triggers on the rising edge of the clock signal (`clk`).

11. ```verilog
    if (out == lmt) begin
       tc <= 1;
    end else begin
       tc <= 0;
    end

```
- On the rising edge of the clock:
  - If the counter value (`out`) equals the limit value (`lmt`), the terminal count (`tc`) is set to 1, indicating that the counter has reached the limit.
  - Otherwise, `tc` is set to 0.

12. ```verilog
endmodule
```
- Ends the module definition.

Summary:
- The `cnt1` module is a 32-bit counter with a `load` and `enable` function. It increments the counter based on the clock signal when enabled (`en = 1`).
- The counter can be reset, or it can load a value directly into the counter if `load = 1`.
- The counter compares its value with a limit (`lmt`), and when it reaches the limit, the terminal count signal (`tc`) is asserted to 1.

Let me know if you need further clarification on the code!

Fdce1.v

Here is a line-by-line explanation of the `fdce1.v` file, which implements a D flip-flop with clock enable and reset functionality:

1. ```verilog
`timescale 1ns / 1ps
```
- Specifies the time unit for simulation (`1ns`) and the precision (`1ps`).

2. ```verilog
module fdce1(q, clk, ce, reset, d);
```
- Declares the `fdce1` module.
- Inputs:
  - `d`: Data input to the flip-flop.
  - `clk`: Clock signal to synchronize operations.
  - `ce`: Clock enable signal, controls whether the data is latched.
  - `reset`: Resets the output.
- Output:
  - `q`: The registered output, which holds the state of the flip-flop.

3. ```verilog
input d, clk, ce, reset;
output reg q;
```
- Declares the input and output ports. The output `q` is declared as a register (`reg`), meaning it holds its value between clock cycles.

4. ```verilog
initial begin q = 0; end
```

   ```
   - Initializes the output `q` to 0 when the simulation or hardware is first powered on.

5. ```verilog
   always @ (negedge (clk)) begin
   ```
   - This block is triggered on the falling edge of the clock signal (`clk`).

6. ```verilog
   if (reset)
     q <= 1'b0;
   ```
   - If the `reset` signal is high (1), the output `q` is set to 0 (`1'b0`).

7. ```verilog
   else if (ce)
     q <= d;
   ```
   - If `reset` is not active and `ce` (clock enable) is high (1), the input data `d` is latched into the output `q`.

8. ```verilog
   else
     q <= q;
   ```
   - If neither `reset` nor `ce` are active, the output `q` retains its current value.

9. ```verilog
   end
   ```
   - Ends the `always` block.

10. ```verilog
   endmodule
   ```
   - Ends the module definition.

### Summary:
- The `fdce1` module is a D flip-flop that stores the value of `d` on the falling edge of the clock (`clk`), but only when the clock enable (`ce`) is high.
- If the `reset` signal is high, the output `q` is reset to 0.
- If `ce` is low, the output `q` holds its previous value.

Let me know if you need more clarification!

**Fdce10.v**

Here's a line-by-line explanation of the `fdce10.v` file, which is a module that uses multiple `fdce1` D flip-flops to process 10-bit data:

1. ```verilog
   ```

```
`timescale 1ns / 1ps
```
- Specifies the time unit and precision for simulation (`1ns` is the time unit, and `1ps` is the precision).

2. ```verilog
```
module fdc10(a, clk, en, reset, y);
```
- Declares the `fdc10` module.
- Inputs:
  - `a`: A 10-bit input vector, which represents the data to be stored.
  - `clk`: Clock signal to synchronize the operations.
  - `en`: Enable signal, controls whether the data is latched into the flip-flop.
  - `reset`: Resets the flip-flops, clearing the stored values.
- Output:
  - `y`: A 10-bit output vector, representing the stored data.

3. ```verilog
```
input [9:0] a;
input clk, en, reset;
output [9:0] y;
```
- Declares the input and output ports.
  - `a`: The 10-bit input data.
  - `y`: The 10-bit output data.
  - `clk`, `en`, and `reset`: The control signals for clock, enable, and reset functionality.

4. ```verilog
```
fdce1 d1(y[0], clk, en, reset, a[0]);
fdce1 d2(y[1], clk, en, reset, a[1]);
fdce1 d3(y[2], clk, en, reset, a[2]);
fdce1 d4(y[3], clk, en, reset, a[3]);
fdce1 d5(y[4], clk, en, reset, a[4]);
fdce1 d6(y[5], clk, en, reset, a[5]);
fdce1 d7(y[6], clk, en, reset, a[6]);
fdce1 d8(y[7], clk, en, reset, a[7]);
fdce1 d9(y[8], clk, en, reset, a[8]);
fdce1 d10(y[9], clk, en, reset, a[9]);
```
- Instantiates 10 instances of the `fdce1` module (which is a D flip-flop with enable and reset).
- Each instance corresponds to one bit of the 10-bit input vector `a` and produces the respective bit in the 10-bit output vector `y`.
- For example:
  - `fdce1 d1(y[0], clk, en, reset, a[0]);` processes the least significant bit of `a` and stores it in `y[0]`.
  - `fdce1 d10(y[9], clk, en, reset, a[9]);` processes the most significant bit of `a` and stores it in `y[9]`.

5. ```verilog
```
endmodule
```
- Ends the module definition.

Summary:
- The `fdc10` module takes in a 10-bit input vector `a`, processes each bit using individual `fdce1` D flip-flops, and stores the result in the 10-bit output vector `y`.
- The `fdce1` D flip-flops are controlled by the `clk`, `en`, and `reset` signals:
  - If `en` (enable) is high, the data from `a` is latched into the output `y`.
  - If `reset` is high, the output is cleared (set to 0).

Let me know if you need further clarification or details!