

## Verilog for Digital Design

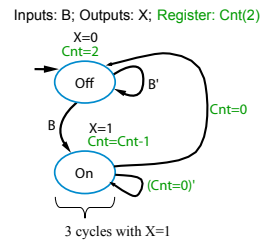
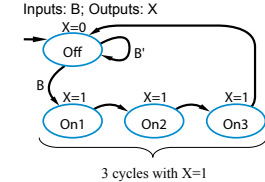
### RTL Design using Verilog

*Verilog for Digital Design Ch. 5*



## High-Level State Machine Behavior

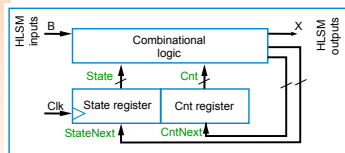
- Register-transfer level (RTL) design captures desired system behavior using high-level state machine
  - Earlier example – 3 cycles high, used FSM
    - What if 512 cycles high? 512-state FSM?
    - Better solution – High-level state machine that uses register to count cycles
      - Declare explicit register Cnt (2 bits for 3-cycles high)
      - Initialize Cnt to 2 (2, 1, 0 → 3 counts)
      - "On" state
        - Sets X=1
        - Configures Cnt for decrement on next cycle
        - Transitions to Off when Cnt is 0
        - Note that transition conditions use current value of Cnt, not next (decremented) value
      - For 512 cycles high, just initialize Cnt to 511



2

## High-Level State Machine Behavior

- Module ports same as FSM
- Same two-procedure approach as FSM
  - One for combinational logic, one for registers
  - Registers now include explicit registers (Cnt)
    - Two reg variables per explicit register (current and next), just like for state register



```

`timescale 1 ns/1 ns
module LaserTimer(B, X, Clk, Rst);
    input B;
    output reg X;
    input Clk, Rst;

    parameter S_Off = 0,
              S_On = 1;

    reg [0:0] State, StateNext;
    reg [1:0] Cnt, CntNext;

    // CombLogic
    always @(State, Cnt, B) begin
        ...
    end

    // Regs
    always @(posedge Clk) begin
        ...
    end
endmodule
    
```

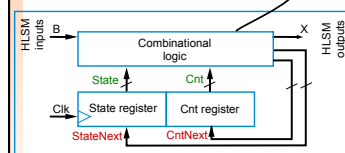
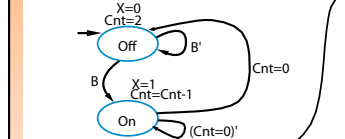
vldd\_ch5\_LaserTimerHLSM.v

3

## High-Level State Machine Behavior

- CombLogic process
  - Describes actions and transitions

Inputs: B; Outputs: X; Register: Cnt(2)



```

...
reg [0:0] State, StateNext;
reg [1:0] Cnt, CntNext;

// CombLogic
always @(State, Cnt, B) begin
    case (State)
        S_Off: begin
            X <= 0;
            CntNext <= 2;
            if (B == 0)
                StateNext <= S_Off;
            else
                StateNext <= S_On;
        end
        S_On: begin
            X <= 1;
            CntNext <= Cnt - 1;
            if (Cnt == 0)
                StateNext <= S_Off;
            else
                StateNext <= S_On;
        end
    endcase
end
    
```

Note: Writes are to "next" variable, reads are from "current" variable. See target architecture to understand why.

vldd\_ch5\_LaserTimerHLSM.v

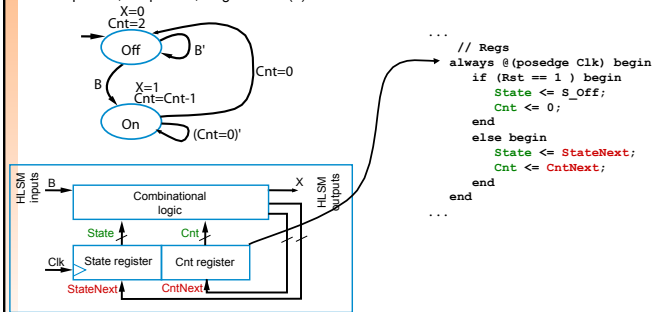
4

## High-Level State Machine Behavior

- Regs process

- Updates registers on rising clock

Inputs: B; Outputs: X; Register: Cnt(2)



vidd\_ch5\_LaserTimerHLSM.v

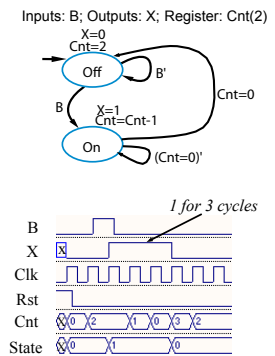
5

## Simulating the HLSM

- Use same testbench as in Chapter 3

- Waveforms below also show Cnt and State variables, even though not a port on the LaserTime module

- Simulators allow one to zoom into modules to select internal variables/nets to show
- Note reset behavior
  - Until Rst=1 and rising clock, Cnt and State undefined
  - Upon Rst=1 and rising clock, Cnt set to 0, and State set to S\_Off (which is defined as 0)
- Note how system enters S\_On on first rising clock after B becomes 1, causing Cnt to be initialized to 2
  - Cnt is decremented in S\_On
  - Cnt wrapped from 0 to 3, but the 3 was never used



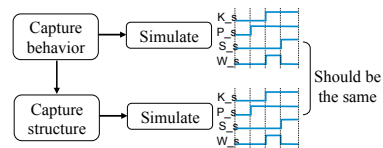
6

## Top-Down Design: HLSM to Controller and Datapath

- Recall from Chapters 2 & 3

- Top-down design

- Capture behavior, and simulate
    - Capture structure (circuit), simulate again
    - Gets behavior right first, unfettered by complexity of creating structure



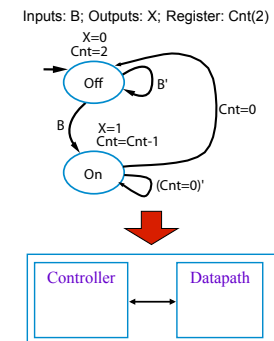
7

## Top-Down Design: HLSM to Controller and Datapath

- Recall from Chapters 2 & 3

- Top-down design

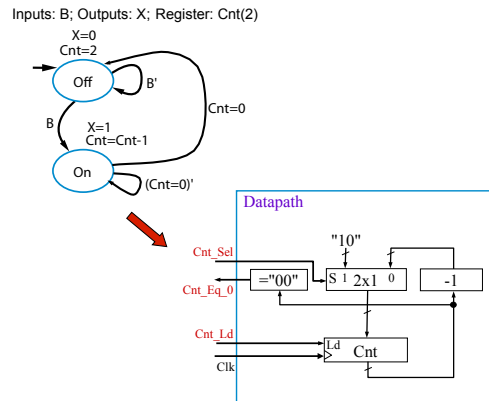
- Capture behavior, and simulate
    - Capture structure (circuit), simulate again
    - Gets behavior right first, unfettered by complexity of creating structure
- At RTL level
  - Capture behavior → HLSM
  - Capture structure: Controller and datapath



8

## Top-Down Design: HLSM to Controller and Datapath

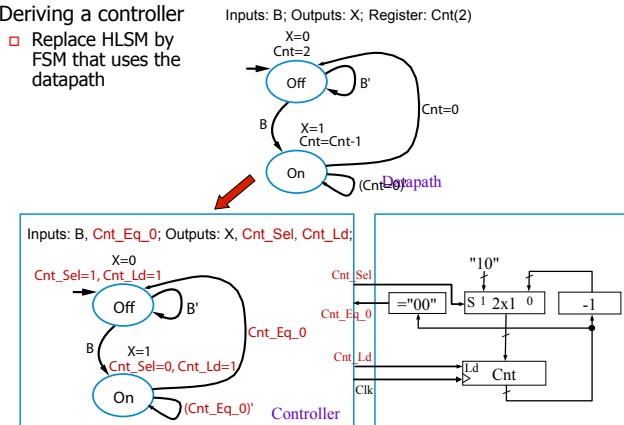
- Deriving a datapath from the HLSM



9

## Top-Down Design: HLSM to Controller and Datapath

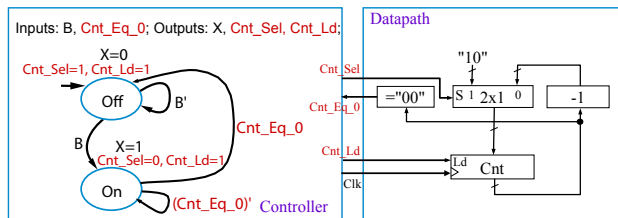
- Deriving a controller
  - Replace HLSM by FSM that uses the datapath



10

## Top-Down Design: HLSM to Controller and Datapath

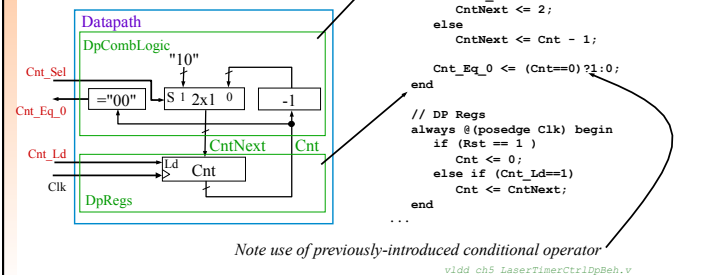
- Describe controller and datapath in VHDL
  - One option: structural datapath, behavioral (FSM) controller
  - Let's instead describe both behaviorally



11

## Describing a Datapath Behaviorally

- Two procedures
  - Combinational part and register part
  - Current and next signals shared between the two parts
  - Just like for FSM behavior



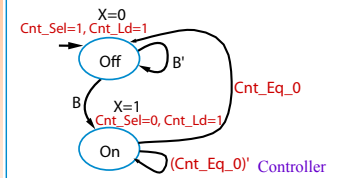
vldd\_ch5\_LaserTimerCtrlDpBeh.v

12

## Describing the Controller Behaviorally

- Standard approach for describing FSM
  - Two procedures

Inputs: B, Cnt\_Eq\_0; Outputs: X, Cnt\_Sel, Cnt\_Ld;



```

... ----- Controller Procedures ----- //
// Ctrl CombLogic
always @(State, Cnt_Eq_0, B) begin
    case (State)
        S_Off: begin
            X <= 0; Cnt_Sel <= 1; Cnt_Ld <= 1;
            if (B == 0)
                StateNext <= S_Off;
            else
                StateNext <= S_On;
        end
        S_On: begin
            X <= 1; Cnt_Sel <= 0; Cnt_Ld <= 1;
            if (Cnt_Eq_0 == 1)
                StateNext <= S_Off;
            else
                StateNext <= S_On;
        end
    endcase
end

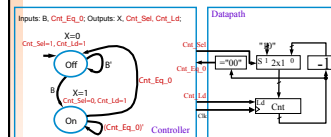
// Ctrl Regs
always @(posedge Clk) begin
    if (Rst == 1) begin
        State <= S_Off;
    end
    else begin
        State <= StateNext;
    end
end
end
...
vldd_ch5_LaserTimerCtrlDpBeh.v

```

13

## Controller and Datapath Behavior

- Result is one module with four procedures
  - Datapath procedures (2)
    - Combinational logic
    - Registers
  - Controller procedures (2)
    - Combinational logic
    - Registers



```

... module LaserTimer(B, X, Clk, Rst);
...
    parameter S_Off = 0,
              S_On = 1;

    // Shared variables
    reg Cnt_Eq_0, Cnt_Sel, Cnt_Ld;
    // Controller variables
    reg [0:0] State, StateNext;
    // Datapath variables
    reg [1:0] Cnt, CntNext;

    // ----- Datapath Procedures ----- //
    // DP CombLogic
    always @(Cnt_Sel, Cnt) begin
        ...
    end

    // DP Regs
    always @(posedge Clk) begin
        ...
    end

    // ----- Controller Procedures ----- //
    // Ctrl CombLogic
    always @(State, Cnt_Eq_0, B) begin
        ...
    end

    // Ctrl Regs
    always @(posedge Clk) begin
        ...
    end
end
endmodule
vldd_ch5_LaserTimerCtrlDpBeh.v

```

14

## One-Procedure State Machine Description

- Previously described HLMSM using two procedures
  - One for combinational logic, one for registers
  - Required "current" and "next" signals for each register
- A one-procedure description is possible too
  - One procedure per HLMSM
  - One variable per register
  - Simpler code with clear grouping of functionality
  - But may change timing

```

... module LaserTimer(B, X, Clk, Rst);
...
    reg [0:0] State, StateNext;
    reg [1:0] Cnt, CntNext;

    // CombLogic
    always @(State, Cnt, B) begin
        ...
    end

    // Regs
    always @(posedge Clk) begin
        ...
    end
endmodule
vldd_ch5_LaserTimerHLMSM.v

... module LaserTimer(B, X, Clk, Rst);
...
    reg [0:0] State;
    reg [1:0] Cnt;

    always @(posedge Clk) begin
        ...
    end
endmodule
vldd_ch5_LaserTimerHLMSMOneProc.v

```

15

## One-Procedure State Machine Description

- Procedure sensitive to **clock only**
- If not reset, then executes current state's actions, and assigns next value of state
- Why didn't we describe with one procedure before?
  - Main reason – Procedure synchronous to clock only → *Inputs become synchronous*
    - Changes the state machine's timing
    - e.g., In state S\_off, and B changes to 1 in middle of clock cycle
    - Previous two procedure model
      - Change in b immediately noticed by combinational logic procedure, which configures next state to be S\_On. State changes to S\_On on next rising clock.
    - One procedure model
      - Change in b not noticed until next rising clock, so next state still S\_Off. After rising clock, procedure configures next state to be S\_On. State changes to S\_On on the next rising clock – or two rising clocks after b changed
  - See next slide for timing diagrams

```

... module LaserTimer(B, X, Clk, Rst);
...
    parameter S_Off = 0,
              S_On = 1;

    reg [0:0] State;
    reg [1:0] Cnt;

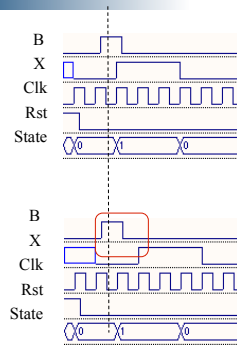
    always @(posedge Clk) begin
        if (Rst == 1) begin
            State <= S_Off;
            Cnt <= 0;
        end
        else begin
            case (State)
                S_Off: begin
                    X <= 0;
                    Cnt <= 2;
                    if (B == 0)
                        State <= S_Off;
                    else
                        State <= S_On;
                end
                S_On: begin
                    X <= 1;
                    Cnt <= Cnt - 1;
                    if (Cnt == 0)
                        State <= S_Off;
                    else
                        State <= S_On;
                end
            endcase
        end
    end
endmodule
vldd_ch5_LaserTimerHLMSMOneProc.v

```

16

## Timing Differences Between Two and One Procedure Descriptions

- Previous two procedure description
  - Change in B *immediately noticed* by combinational logic procedure, which configures next state to be S\_On. State changes to S\_On (1) on *next* rising clock (setting X=1).
- One procedure description
  - Change in B *not noticed until next rising clock*, so next state still S\_Off (0). After rising clock, procedure configures next state to be S\_On (1). State changes to S\_On on the next rising clock (setting X=1) – *two* rising clocks after b changed
    - Likewise, initial reset also delayed

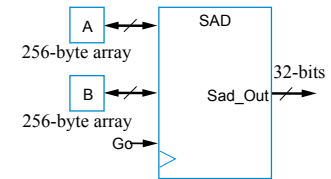


*Synchronization of input can delay impact of input changes*

17

## Algorithmic-Level Behavior

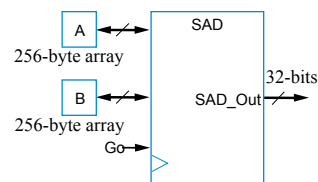
- Higher than HLSM?
  - HLSM high-level, but even higher sometimes better
- Algorithmic-level
  - Behavior described as sequence of steps
  - Similar to sequential program
- SAD example
  - Compute sum of absolute differences of corresponding pairs in two arrays
    - Common task in video compression to determine difference between two successive video frames



18

## Algorithmic-Level Behavior for SAD

- Most easily described as an algorithm
  - Input is two arrays A and B
  - Wait until Go is '1'
  - Use for loop to step through each array item
    - Adds absolute difference to running sum
  - Update output after loop completes
- Note: No intention of providing this description to synthesis tool
  - Merely used for early system simulation



```
Wait until Go equals 1
Sum = 0
for I in 0 to 255 loop
    Sum = Sum + |A(I) - B(I)|
end loop
SAD_Out = Sum
```

*Note: Above algorithm written in pseudo-code, not in a particular language*

19

## Algorithmic-Level Behavior for SAD

- Description uses several language features to be described on next several slides
- User-defined function
  - Returns a value to be used in an expression
  - This function named "ABS"
    - Will compute absolute value
    - Returns integer value
    - One input argument, integer
    - Contents assign return value to be positive version of input argument
    - This function contains only one statement, but may contain any number
  - ABS can then be called by later parts of the description

```
`timescale 1 ns/1 ns
module SAD(Go, SAD_Out);
    input Go;
    output reg [31:0] SAD_Out;

    reg [7:0] A [0:255];
    reg [7:0] B [0:255];
    integer Sum;
    integer I;

    function integer ABS;
        input integer IntVal;
        begin
            ABS = (IntVal>=0)?IntVal:-IntVal;
        end
    endfunction

    // Initialize Arrays
    initial $readmemb("MemA.txt", A);
    initial $readmemb("MemB.txt", B);

    always begin
        if (!(Go==1)) begin
            @(Go==1);
        end
        Sum = 0;
        for (I=0; I<=255; I=I+1) begin
            Sum = Sum + ABS(A[I] - B[I]);
        end
        #50;
        SAD_Out <= Sum;
    end
endmodule
```

*vldd\_ch5\_SADAig.v*

20

## Algorithmic-Level Behavior for SAD

- Description declares **A** and **B** as 256-element arrays of bytes
- Initializes those arrays using built-in system task **\$readmemh**
  - Reads file of hex numbers (first argument)
  - Each number placed into subsequent element of array (second argument)
    - Number of numbers in file should match number of elements
    - No length or base format for numbers
    - Separated by white space
    - e.g., 00 FF A1 04 ...
  - **\$readmemb**: Reads file of binary numbers
- Note: Called as only statement of "initial" procedure
  - Could have used begin-end block:

```
initial begin
    $readmemh("MemA.txt", A);
end
```

```

`timescale 1 ns/1 ns

module SAD(Go, SAD_Out);

    input Go;
    output reg [31:0] SAD_Out;

    reg [7:0] A [0:255];
    reg [7:0] B [0:255];
    integer Sum;
    integer I;

    function integer ABS;
        input integer IntVal;
        begin
            ABS = (IntVal >= 0) ? IntVal : -IntVal;
        end
    endfunction

    // Initialize Arrays
    initial $readmemh("MemA.txt", A);
    initial $readmemh("MemB.txt", B);

    always begin
        if (!(Go==1)) begin
            @ (Go==1);
        end
        Sum = 0;
        for (I=0; I<=255; I=I+1) begin
            Sum = Sum + ABS(A[I] - B[I]);
        end
        #50;
        SAD_Out <= Sum;
    end
endmodule
vldd_ch5_SADAlg.v

```

21

## Event Control with an Expression

- Uses **@(Go==1)** – Event control with an expression
  - If Go not 1, wait until Go becomes 1
  - Previous event control expressions were either one event, such as **@(X)** or **@(posedge Clk)**, or a list of events such as **@(X,Y)**
  - But expression can be a longer expression too, like **@(Go==1)**
    - Waits not just for change on Go, but for a change on Go such that Go equals 1

```

...
always begin
    if (!(Go==1)) begin
        @(Go==1);
    end
    Sum = 0;
    for (I=0; I<=255; I=I+1) begin
        Sum = Sum + ABS(A[I] - B[I]);
    end
    #50;
    SAD_Out <= Sum;
end
...

```

*Delay of 50 ns added just so that result doesn't appear instantaneously*

vldd\_ch5\_SADAlg.v

22

## Algorithmic-Level Behavior

- Testbench
  - Setup similar to previous testbenches
  - Activate SAD with **Go\_s <= 1**
- Simulation generates **SAD\_out**
  - **SAD\_out** is uninitialized at first, resulting in unknown value
  - 50 ns later, result is 4
    - Which is correct for the value with which we initialized the memories **A** and **B**
- More thorough algorithmic-level description would be good
  - Perhaps try different sets of values

```

`timescale 1 ns/1 ns

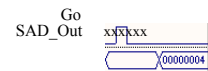
module Testbench();

    reg Go_s;
    wire [31:0] SAD_Out_s;

    SAD CompToTest(Go_s, SAD_Out_s);

    // Vector Procedure
    initial begin
        Go_s <= 0;
        #10 Go_s <= 1;
        #10 Go_s <= 0;
        #60 if (SAD_Out_s != 4) begin
            $display("SAD failed -- should equal 4");
        end
    end
endmodule
vldd_ch5_SADAlgTB.v

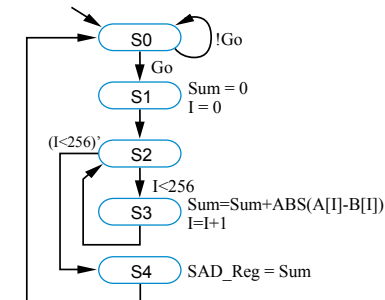
```



23

## Convert Algorithm to HLSM

Local registers: Sum, SAD\_Reg (32 bits); I (integer)



24

## Describe HLSM in Verilog

```
// High-level state machine
always @(posedge Clk) begin
    if (Rst==1) begin
        State <= S0;
        Sum <= 0;
        SAD_Reg <= 0;
        I <= 0;
    end
    else begin
        case (State)
            S0: begin
                if (Go==1)
                    State <= S1;
                else
                    State <= S0;
            end
            S1: begin
                Sum <= 0;
                I <= 0;
                State <= S2;
            end
            S2: begin
                if (!I==255)
                    State <= S3;
                else
                    State <= S4;
            end
            S3: begin
                Sum <= Sum + ABS(A[I]-B[I]);
                I <= I + 1;
                State <= S2;
            end
            S4: begin
                SAD_Reg <= Sum;
                State <= S0;
            end
        endcase
    end
end

module SAD(Go, SAD_Out, Clk, Rst);
    input Go;
    output [31:0] SAD_Out;
    input Clk, Rst;

    parameter S0 = 0, S1 = 1,
              S2 = 2, S3 = 3,
              S4 = 4;

    reg [7:0] A [0:255];
    reg [7:0] B [0:255];
    reg [2:0] State;
    integer Sum, SAD_Reg;
    integer I;

    function integer ABS;
        input integer IntVal;
        begin
            ABS = (IntVal>=0)?IntVal:-IntVal;
        end
    endfunction

    // Initialize Arrays
    initial $readmemb("MemA.txt", A);
    initial $readmemb("MemB.txt", B);

    vldd_ch5_SADHLSM.v end
```

25

## Describe HLSM in Verilog

```
`timescale 1 ns/1 ns

module Testbench();

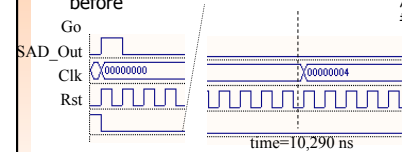
    reg Go_s;
    reg Clk_s, Rst_s;
    wire [31:0] SAD_Out_s;

    SAD CompToTest(Go_s, SAD_Out_s, Clk_s, Rst_s);

    // Clock Procedure
    always begin
        Clk_s <= 0;
        #10;
        Clk_s <= 1;
        #10;
    end // Note: Procedure repeats

    // Vector Procedure
    initial begin
        Rst_s <= 1;
        Go_s <= 0;
        @ (posedge Clk_s);
        Rst_s <= 0;
        Go_s <= 1;
        @ (posedge Clk_s);
        Go_s <= 0;
        #((256*2+3)*20) if (SAD_Out_s != 4) begin
            $display("SAD failed -- should equal 4");
        end
    end
endmodule

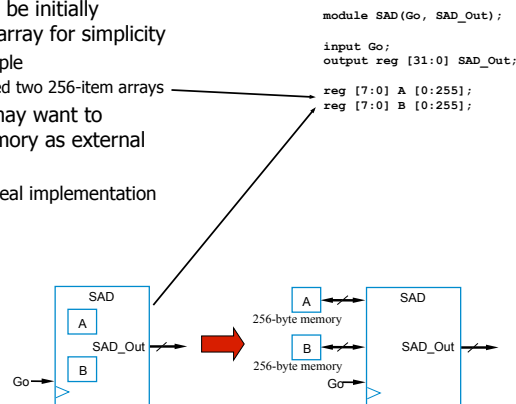
vldd_ch5_SADHLSMTB.v
```



26

## Accessing Memory

- Memory may be initially accessed as array for simplicity
  - SAD example
    - Declared two 256-item arrays
- Eventually, may want to describe memory as external component
  - Closer to real implementation



27

## Simple Memory Entity

- Simple read-only memory
  - Addr and data ports only
  - Declares array named Memory for storage
  - Procedure uses continuous assignment statement to always output Memory data element corresponding to current address input value

```
`timescale 1 ns/1 ns

module SADMem(Addr, Data);
    input [7:0] Addr;
    output [7:0] Data;

    reg [7:0] Memory [0:255];

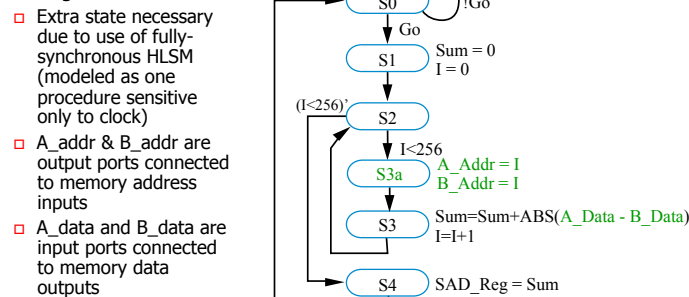
    assign Data = Memory[Addr];
endmodule

vldd_ch5_SADMem.v
```

28

## Accessing Memory

- Modify SAD's HLMS with extra state (S3a) for reading the memories



29

```

`timescale 1 ns/1 ns

module SAD(Go, A_Addr, A_Data,
           B_Addr, B_Data,
           SAD_Out, Clk, Rst);

    input Go;
    input [7:0] A_Data, B_Data;
    output reg [7:0] A_Addr, B_Addr;
    output [31:0] SAD_Out;
    input Clk, Rst;

    always @(posedge Clk) begin
        if (Rst==1) begin
            A_Addr <= 0;
            B_Addr <= 0;
            State <= S0;
            Sum <= 0;
            SAD_Reg <= 0;
            I <= 0;
        end
        else begin
            case (State)
                S0: begin
                    if (Go==1)
                        State <= S1;
                    else
                        State <= S0;
                end
                S1: begin
                    Sum <= 0;
                    I <= 0;
                    assign SAD_Out = SAD_Reg;
                    State <= S2;
                end
                S2: begin
                    if (! (I==255))
                        State <= S3a;
                    else
                        State <= S4;
                end
                S3a: begin
                    A_Addr <= I;
                    B_Addr <= I;
                    State <= S3;
                end
                S3: begin
                    Sum <= Sum +
                        ABSDiff(A_Data, B_Data);
                    I <= I + 1;
                    State <= S2;
                end
                S4: begin
                    SAD_Reg <= Sum;
                    State <= S0;
                end
            endcase
        end
    end

    function [7:0] ABSDiff;
    input [7:0] A;
    input [7:0] B;
    begin
        if (A>B) ABSDiff = A - B;
        else ABSDiff = B - A;
    end
endfunction

endmodule

```

Created AbsDiff function for improved readability

30

## Testbench

- Instantiate and connect modules for SAD, SADMemA, and SADMemB
  - Note two instances of same memory
- Initialize memories
  - Note how we can access SADMemA.Memory (and B) from testbench
- Vector procedure adjusted from earlier to account for extra state
  - $((256*2+3)*20)$  changed to  $((256*3+3)*ClkPeriod)$
  - Also note use of parameter **ClkPeriod** – allows us to change period in one place

```

module Testbench();
    reg Go_s;
    wire [7:0] A_Addr_s, B_Addr_s;
    wire [7:0] A_Data_s, B_Data_s;
    reg Clk_s, Rst_s;
    wire [31:0] SAD_Out_s;

    parameter ClkPeriod = 20;

    SAD CompToTest(Go_s, A_Addr_s, A_Data_s,
                  B_Addr_s, B_Data_s,
                  SAD_Out_s, Clk_s, Rst_s);
    SADMem SADMemA(A_Addr_s, A_Data_s);
    SADMem SADMemB(B_Addr_s, B_Data_s);

    // Clock Procedure
    always begin
        Clk_s <= 0; # (ClkPeriod/2);
        Clk_s <= 1; # (ClkPeriod/2);
    end

    // Initialize Arrays
    initial $readmemh("MemA.txt", SADMemA.Memory);
    initial $readmemh("MemB.txt", SADMemB.Memory);

    // Vector Procedure
    initial begin
        ... // Reset behavior not shown
        Go_s <= 1;
        @ (posedge Clk_s);
        Go_s <= 0;
        # ((256*3+3)*ClkPeriod) if (SAD_Out_s != 4) begin
            $display("SAD failed -- should equal 4");
        end
    end
endmodule

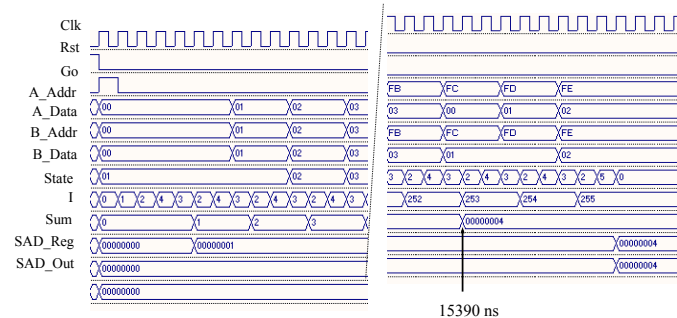
```

vldd\_ch5\_SADHLMS\_MemTB.v

31

## Waveforms

- Waveforms now include address and data lines with memories
  - We also added some internal signals, State, I, Sum, and SAD\_Reg
- SAD\_Out result appears later than previously, due to extra state



32