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

```verilog
// Module: LaserTimer
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
  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.
High-Level State Machine Behavior

- **Regs process**
  - Updates registers on rising clock

```verilog
// Regs always @(posedge Clk) begin
if (Rst == 1 ) begin
    State <= S_Off;
    Cnt <= 0;
end else begin
    State <= StateNext;
    Cnt <= CntNext;
end
end
```

Combinational logic

State register

State

Cnt register

Cnt

Inputs: B; Outputs: X;

On

Off

X

B

State

Cnt

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

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
Top-Down Design: HLSM to Controller and Datapath

- Deriving a datapath from the HLSM
  - Inputs: B; Outputs: X; Register: Cnt(2)
  - On Off
    - X=1 Cnt=Cnt-1
    - B Cnt=2
  - Datapath
    - Cnt_Sel
    - Cnt_Ld
    - Cnt_Eq_0
    - Cnt

Top-Down Design: HLSM to Controller and Datapath

- Deriving a controller
  - Replace HLSM by FSM that uses the datapath
  - Inputs: B, Cnt_Eq_0; Outputs: X, Cnt_Sel, Cnt_Ld
  - On Off
    - X=1 Cnt=Cnt-1
    - B Cnt=2
  - Datapath
    - Cnt_Sel
    - Cnt_Ld
    - Cnt_Eq_0
    - Cnt

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

// 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
  if (Cnt_Sel==1)
    CntNext <= 2;
  else
    CntNext <= Cnt - 1;
  Cnt_Eq_0 <= (Cnt==0)?1:0;
end

// DP Regs
always @(posedge Clk) begin
  if (Rst == 1 )
    Cnt <= 0;
  else if (Cnt_Ld==1)
    Cnt <= CntNext;
end

Note use of previously-introduced conditional operator
**Describing the Controller Behaviorally**

- Standard approach for describing FSM
  - Two procedures

**Inputs:** B, Cnt_Sel, Cnt_Ld, Inputs: B, Cnt_Sel; Output: Cnt_Ld

- Controller and Datapath Behavior
  - Result is one module with four procedures
    - Datapath procedures (2)
      - Combilogic
      - Registers
    - Controller procedures (2)
      - Combilogic
      - Registers

**One-Procedure State Machine Description**

- Previously described HLSM 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 HLSM
  - One variable per register
  - Simpler code with clear grouping of functionality
  - But may change timing

**Controller and Datapath Behavior**

- One procedure model
  - Datapath procedures (2)
    - Combilogic
    - Registers
  - Controller procedures (2)
    - Combilogic
    - Registers

**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
  - Changes the combinational logic procedure, which configures next state to be S_On, On next rising clock.

**Controller and Datapath Behavior**

- One procedure model
  - Timing changes
  - See next slide for timing diagrams

**One-Procedure State Machine Description**

- Previous two procedure model
  - Changes in b immediately noticed by combinational logic procedure, which configures next state to be S_On, On next rising clock.

**Controller and Datapath Behavior**

- One procedure model
  - Timing changes
  - See next slide for timing diagrams
### 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). Likewise, initial reset also delayed.

Synchronization of input can delay impact of input changes.

### 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.

### 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

- **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
    - 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

Note: Above algorithm written in pseudo-code, not in a particular language
Algorithmic-Level Behavior for SAD

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

```verilog
// 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
```

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

```verilog
... 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

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

```verilog
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;
  if (SAD_Out_s != 4) begin
    $display("SAD failed -- should equal 4");
  end
end
endmodule
```

Convert Algorithm to HLSM

- Local registers: Sum, SAD_Reg (32 bits), 1 (integer)
- Delay of 50 ns added just so that result doesn’t appear instantaneously
Describe HLSM in Verilog

```verilog
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 $readmemh("MemA.txt", A);
initial $readmemh("MemB.txt", B);
// 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
```

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

```verilog
module SADMem(Addr, Data);
input [7:0] Addr;
output [7:0] Data;
reg [7:0] Memory [0:255];
assign Data = Memory[Addr];
endmodule
```

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

```verilog
module SADMem(Addr, Data);
input [7:0] Addr;
output [7:0] Data;
reg [7:0] Memory [0:255];
assign Data = Memory[Addr];
endmodule
```
**Accessing Memory**

- **Modify SAD’s HLSM with extra state (S3a) for reading the memories**
  - Extra state necessary due to use of fully-synchronous HLSM (modeled as one procedure sensitive only to clock)
  - A_addr & B_addr are output ports connected to memory address inputs
  - A_data and B_data are input ports connected to memory data outputs

![Flowchart of SAD HLSM](image)

- **Go**
- **S0**: Go
- **S1**: Sum = 0
- **S2**: (0-256)
- **S3a**: A_Addr = I, B_Addr = I
- **S3**: Sum=Sum+ABS(A_Data - B_Data)
  - I=I+1
- **S4**: SAD_Reg = Sum

---

**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

```verilog
timescale 1 ns/1 ns
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
end
endmodule
```

---

**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