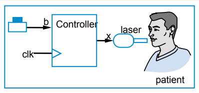
ECE 474A/57A
Computer-Aided Logic Design

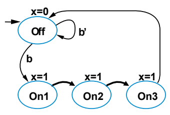
REVIEW

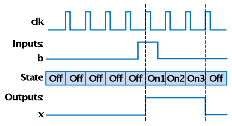
Sequential Logic and RTL Design

FSM Example: Three-Cycles High Laser Timer

- State Diagram or Finite-State Machine (FSM)
  - A way to describe desired behavior of sequential circuit
  - List states, and transitions among states
- Laser Timer
  - When button pressed (b=1), turn laser on (x=1) for 3 clock cycles
- Four states
  - Off state
    - Keep laser turned off
    - While b=0 (b'), we are in a wait state
    - When b=1 and rising clock edge (b • clk^), transition to On1 state
  - On1 state
    - Turns laser on (x=1)
    - On next rising clock edge (clk^) transition to On2 state
  - On2/On3 state
    - Also turns laser on (x=1)
    - Transitions on next rising clock edge

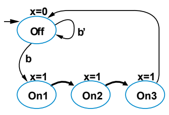


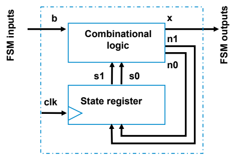




Graphical and Textual Sequential Circuit Descriptions

- FSM
  - Graphical representation
  - Formal method to describe sequential circuits
- State Table
  - Textual Representation
- How do we implement a sequential circuit?
  - Standard Controller architecture
    - Need to store state
      - State register (encoded state)
    - Need to determine next state
      - Current state and external input to combinational logic
    - Need to determine output
      - Current state input to combinational logic

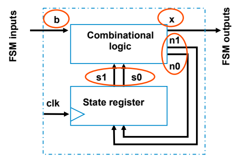




Controller architecture for laser timer example

State Table Example: Laser Timer (cont')

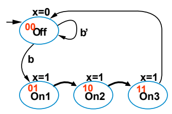
- State Table
  - Inputs
    - Current state (encoded) - Two bits s1 and s0 encode the current state
    - FSM Input - Input b indicates button press
  - Outputs
    - Next state (encoded) - Two bits n1 and n0 encode the next state
    - FSM Output - Output x controls when the laser is on/off



Inputs			Outputs		
s1	s0	b	n1	n0	x

State Table Example: Laser Timer (cont')

- State Table
  - Next state
    - Based on current state and FSM input what is the next state?
  - FSM Output
    - Output depends on current state only (Moore FSM)
    - For each state we are currently in, what is the output?



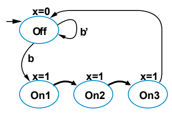
	Inputs			Outputs		
	s1	s0	b	n1	n0	x
Off	0	0	0	0	0	0
	0	0	1	0	1	0
On1	0	1	0	1	0	1
	0	1	1	1	0	1
On2	1	0	0	1	1	1
	1	0	1	1	1	1
On3	1	1	0	0	0	1
	1	1	1	0	0	1

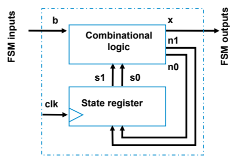
(Condensed) Controller Design Process

	Step	Description
Step 1:	Capture the FSM	Create an FSM (state diagram) that describes the desired behavior of the circuit
Step 2:	Create the architecture	Create the standard architecture by using a state register of appropriate width, and combinational logic with inputs being the state register bits and the FSM inputs, and outputs being the next state bits and the FSM outputs
Step 3:	Encode the states	Assign a unique binary number to each state. Each binary number representing a state is know as an encoding. Any encoding will do as long as they are unique.
Step 4:	Create the state table	Create a truth table for the combinational logic such that the logic will generate the correct FSM output and next state signals. Ordering the inputs with state bits first make the truth table describe the state behavior, giving us a state table.
Step 5:	Implement the combinational logic	Implement the combinational logic using any method.

Controller Design: Laser Timer

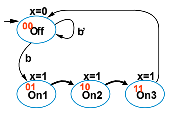
- Example: Laser Timer
- Step 1: Capture the FSM
  - Already done
- Step 2: Create architecture
  - Customize generic controller architecture to our system
    - State Register
      - 2-bit state register (for 4 states)
      - s1, s0 – current state bits
      - n1, n0 – next state bits
    - FSM Input
      - Button signal b
    - FSM Output
      - Laser control x





Controller Design: Laser Timer (Cont')

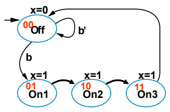
- Step 3: Encode the states
  - Any encoding with each state unique will work
- Step 4: Create state table
  - Done this already



	Inputs			Outputs		
	s1	s0	b	n1	n0	x
Off	0	0	0	0	0	0
	0	0	1	0	1	0
On1	0	1	0	1	0	1
	0	1	1	1	0	1
On2	1	0	0	1	1	1
	1	0	1	1	1	1
On3	1	1	0	0	0	1
	1	1	1	0	0	1

Controller Design: Laser Timer (Cont')

- Step 5: Implement the combinational logic



n1 = s1's0b' + s1's0b + s1s0'b' + s1s0'b
n1 = s1's0 + s1s0'

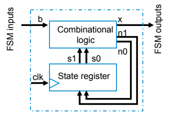
n0 = s1's0'b + s1s0'b' + s1s0'b
n0 = s1's0'b + s1s0'

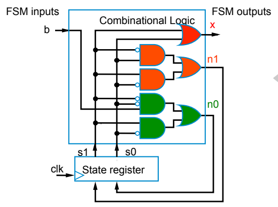
x = s1's0'b + s1s0'b' + s1s0'b
x = s1 + s0

	Inputs			Outputs		
	s1	s0	b	n1	n0	x
Off	0	0	0	0	0	0
	0	0	1	0	1	0
On1	0	1	0	1	0	1
	0	1	1	1	0	1
On2	1	0	0	1	1	1
	1	0	1	1	1	1
On3	1	1	0	0	0	1
	1	1	1	0	0	1

Controller Design: Laser Timer (Cont')

- Step 5: Implement combinational logic (cont)



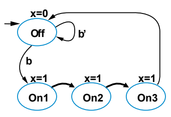


x = s1 + s0
n1 = s1's0 + s1s0'
n0 = s1's0'b + s1s0'

FSM Formal Definition

- FSM defined by quintuple
  - M = (Σ , Γ , S, δ, λ, s_o)
  - Σ is the input alphabet
  - Γ is the output alphabet
  - S is a finite set of states
  - δ is the transition function, δ: X x S→S
    - Given and input and state, what is the next state
  - λ is the output function, λ: S → Y
    - Mealy FSM, λ: X x S → Y
  - s_o is the initial state

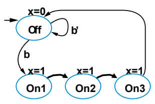
Inputs: b
Outputs: x



FSM Formal Definition

- Formally specify the Laser Timer FSM
  - M = (Σ , Γ , S, δ, λ, s_o)

Inputs: b
Outputs: x



LaserTimer = (Σ, Γ, S, δ , λ , qo), where

Σ = {0, 1}

Γ = {0, 1}

S = {Off, On1, On2, On3}

δ (Off, 0) = Off, δ (Off, 1) = On1
δ (On1, 0) = On2, δ (On1, 1) = On2
δ (On2, 0) = On3, δ (On2, 1) =On3
δ (On3, 0) = Off, δ (On3, 1) = Off

λ (Off) = 0, λ (On1) = 1, λ (On2) = 1, λ (On3) = 1

s_o = Off

Σ is the input alphabet

Y is the output alphabet

S is a finite set of states

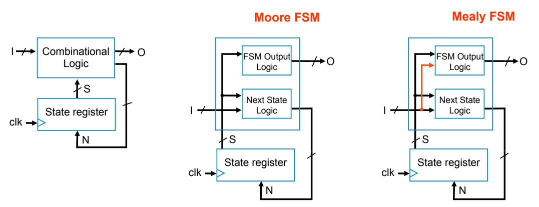
δ is the transition function, δ : X x S→S
Given and input and state, what is the next state

λ is the output funciton, λ : S → Y

s_o is the initial state

Moore vs. Mealy FSM - Architecture

- Why does the timing change?
  - More detailed view of FSM implementation architecture



Moore FSM	Mealy FSM
Next state logic – function of present state and FSM inputs	Next state logic – function of present state and FSM inputs
Output logic - function of present state only	Output logic - function of present state and FSM inputs

ECE 474A/57A
Computer-Aided Logic Design

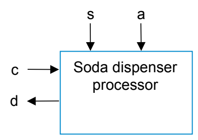
Register-Transfer Level (RTL) Design

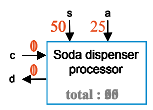
RTL Design Method

	Step	Description
Step 1:	Capture the high-level FSM	Describe the system's desired behavior as a high-level state machine. The state machine consists of states and transitions. The state machine is "high-level" because the transition conditions and the state actions are more than just Boolean operations on bit inputs and outputs
Step 2:	Create a datapath	Create a datapath to carry out the data operations on the high-level state machine
Step 3:	Connect the datapath to the controller	Connect the datapath to the controller block. Connect external Boolean inputs and output to the controller block
Step 4:	Derive the controller's FSM	Convert the high-level state machine to a finite-state machine (FSM) for the controller, by replacing data operations with setting and reading of control signals to and from the datapath

RTL Design Method
Soda Dispenser Example

- Soda dispenser
  - *c*: bit input, 1 when coin deposited
  - *a*: 8-bit input having value of deposited coin
  - *s*: 8-bit input having cost of a soda
  - *d*: bit output, processor sets to 1 when total value of deposited coins equals or exceeds cost of a soda



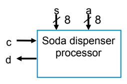


How can we precisely describe this processor's behavior?

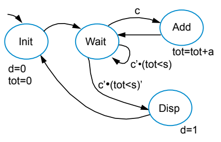
RTL Design Method

Soda Dispenser - Step 1: Capture the high-level FSM

- Step 1: Describe behavior using high-level FSM
  - Start with inputs/output of system
  - Declare local register *tot*
  - **Init** state
    - Don't dispense soda (d=0), clear running total (tot=0)
  - **Wait** state
    - Wait for coin, if see coin go to Add state
  - **Add** state
    - Update total value: tot = tot + a
      - Remember, *a* is present coin's value
    - Go back to Wait state
  - In Wait state
    - If tot < = s, Wait
    - If tot >= s, go to Disp(ense) state
  - **Disp** state
    - Set d=1 (dispense soda)
    - Return to Init state



Inputs: c (bit), a (8 bits), s (8 bits)
Outputs: d (bit)
Local registers: tot (8 bits)

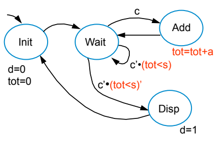


RTL Design Method

FSM vs. High-level FSM

- Created a high-level FSM, not an FSM, because
  - Multi-bit (data) inputs *a* and *s*
  - Local register *tot*
  - Data operations *tot=0, tot<s, tot=tot +a.*
- High-level state machines are useful,
  - Data types beyond just bits
  - Local registers
  - Arithmetic equations/expressions

Inputs: c (bit), a (8 bits), s (8 bits)
Outputs: d (bit)
Local registers: tot (8 bits)

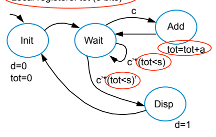


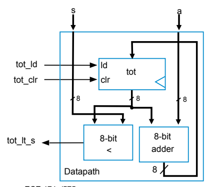
RTL Design Method

Soda Dispenser - Step 2: Create a datapath

- Step 2: Create a Datapath
  - What's going in and out of datapath?
    - Multi-bit values – a, s
  - Need *tot* register
  - Need 8-bit comparator to compare *s* and *tot*
  - Need 8-bit adder to perform tot = tot + a

Inputs: c (bit), a (8 bits), s (8 bits)
Outputs: d (bit)
Local registers: tot (8 bits)

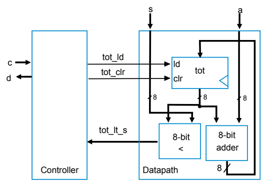




RTL Design Method

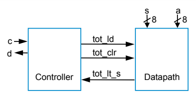
Soda Dispenser - Step 3: Connect the datapath to a controller

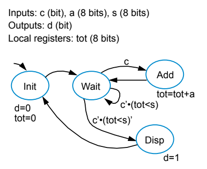
- Step 3: connect datapath to controller
  - Controller's inputs
    - External input *c* (coin detected)
    - Input from datapath comparator's output, which we named *tot_lt_s*
  - Controller's outputs
    - External output *d* (dispense soda)
    - Outputs to datapath to load and clear the *tot* register

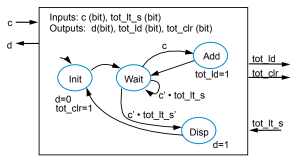


RTL Design Method
Soda Dispenser - Step 4: Derive the controller's FSM

- Step 4: Derive the Controller's FSM
  - Same states and arcs as high-level state machine
  - Transitions and state assignment are bit operations
    - Set/read datapath control signals for all datapath operations and conditions



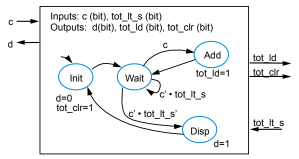




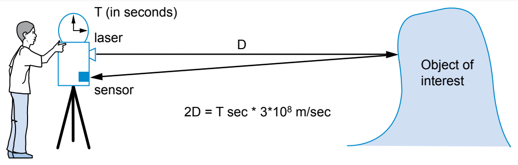
RTL Design Method
Soda Dispenser - Completing the design

- Once we have FSM
  - Implement the FSM as a state register and logic
  - State table shown on right

	s1	s0	c	tot_lt_s	n1	n0	d	tot_ld	tot_dr
Init	0	0	0	0	0	1	0	0	1
	0	0	0	1	0	1	0	0	1
	0	0	1	0	0	1	0	0	1
	0	0	1	1	0	1	0	0	1
Wait	0	1	0	0	1	1	0	0	0
	0	1	0	1	0	1	0	0	0
	0	1	1	0	1	0	0	0	0
	0	1	1	1	1	0	0	0	0
Add	1	0	0	0	0	1	0	1	0
		• • •				• • •			
Disp	1	1	0	0	0	0	1	0	0
		• • •				• • •			

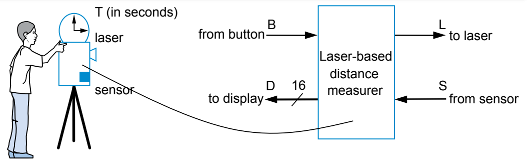


Laser-Based Distance Measurer
Step 1: Capture a high-level state machine



- Example of how to create a high-level state machine to describe desired processor behavior
- Laser-based distance measurement – pulse laser, measure time T to sense reflection
  - Laser light travels at speed of light, $3*10^8$ m/sec
  - Distance is thus D = T sec * $3*10^8$ m/sec / 2

Laser-Based Distance Measurer
Step 1 : Capture a high-level state machine

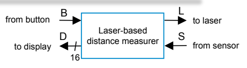


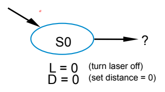
- Inputs/outputs
  - *B*: bit input, from button to begin measurement
  - *L*: bit output, activates laser
  - *S*: bit input, senses laser reflection
  - *D*: 16-bit output, displays computed distance

Laser-Based Distance Measurer

Step 1 : Capture a high-level state machine

Inputs: B, S (1 bit each)
Outputs: L (bit), D (16 bits)



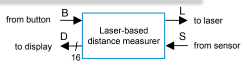


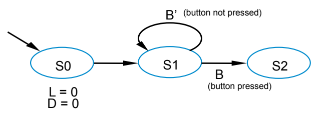
- Step 1: Create high-level state machine
- Begin by declaring inputs and outputs
- Create initial state, name it **S0**
  - Initialize laser to off (L=0)
  - Initialize displayed distance to 0 (D=0)

Laser-Based Distance Measurer

Step 1 : Capture a high-level state machine

Inputs: B, S (1 bit each)
Outputs: L (bit), D (16 bits)





- Add another state, call **S1**, that waits for a button press
  - B' – stay in **S1**, keep waiting
  - B – go to a new state **S2**

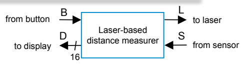
Q: What should S2 do?

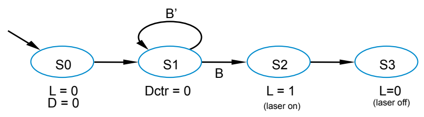
A: Turn on the laser

Laser-Based Distance Measurer

Step 1 : Capture a high-level state machine

Inputs: B, S (1 bit each)
Outputs: L (bit), D (16 bits)





- Add a state **S2** that turns on the laser (L=1)
- Then turn off laser (L=0) in a state **S3**

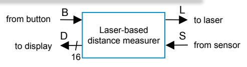
Q: What do next?

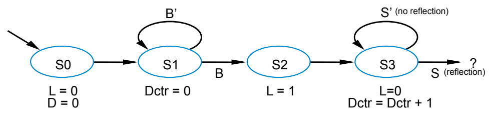
A: Start timer, wait to sense reflection

Laser-Based Distance Measurer

Step 1 : Capture a high-level state machine

Inputs: B, S (1 bit each)
Outputs: L (bit), D (16 bits)
Local Registers: Dctr (16 bits)



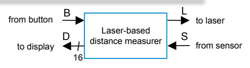


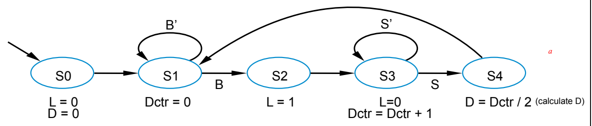
- Stay in **S3** until sense reflection (S)
- To measure time, count cycles for which we are in **S3**
  - To count, declare local register *Dctr*
  - Increment *Dctr* each cycle in **S3**
  - Initialize *Dctr* to 0 in **S1**. **S2** would have been O.K. too

Laser-Based Distance Measurer

Step 1 : Capture a high-level state machine

Inputs: B, S (1 bit each)
Outputs: L (bit), D (16 bits)
Local Registers: Dctr (16 bits)





- Once reflection detected (S), go to new state **S4**
  - Calculate distance
  - Assuming clock frequency is $3x10^8$, *Dctr* holds number of meters, so D=Dctr/2
- After **S4**, go back to **S1** to wait for button again

Laser-Based Distance Measurer

Step 2: Create a Datapath

- Datapath must
  - Implement data storage
  - Implement data computations
- Look at high-level state machine, do three substeps
  a) Make data inputs/outputs be datapath inputs/outputs
  b) Instantiate declared registers into the datapath (also instantiate a register for each data output)
  c) Examine every state and transition, and instantiate datapath components and connections to implement any data computations

Instantiate: to introduce a new component into a design.

Laser-Based Distance Measurer

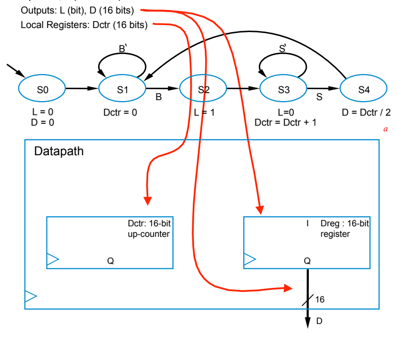
Step 2: Create a Datapath

a) Make data inputs/outputs be datapath inputs/outputs

b) Instantiate declared registers into the datapath (also instantiate a register for each data output)

c) Examine every state and transition, and instantiate datapath components and connections to implement any data computations

Inputs: B, S (1 bit each)
Outputs: L (bit), D (16 bits)
Local Registers: Dctr (16 bits)



Laser-Based Distance Measurer

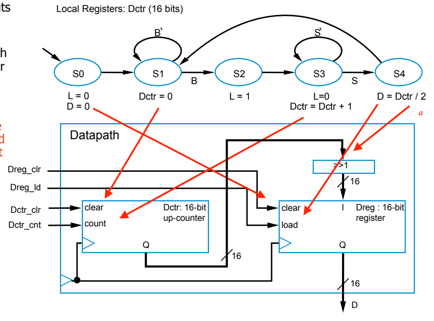
Step 2: Create a Datapath

a) Make data inputs/outputs be datapath inputs/outputs

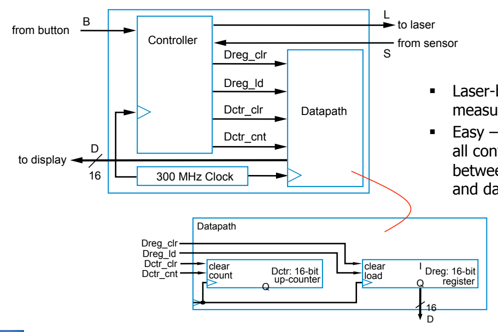
b) Instantiate declared registers into the datapath (also instantiate a register for each data output)

c) Examine every state and transition, and instantiate datapath components and connections to implement any data computations

Inputs: B, S (1 bit each)
Outputs: L (bit), D (16 bits)
Local Registers: Dctr (16 bits)



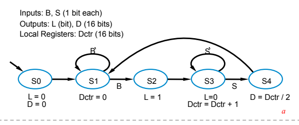
Laser-Based Distance Measurer
Step 3: Connecting the Datapath to a Controller

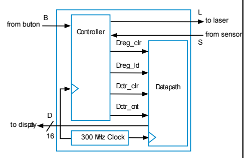


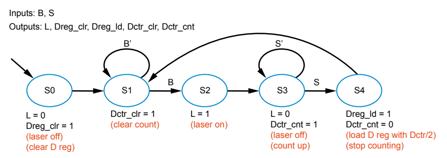
- Laser-based distance measurer example
- Easy – just connect all control signals between controller and datapath

Laser-Based Distance Measurer
Step 4: Deriving the Controller's FSM

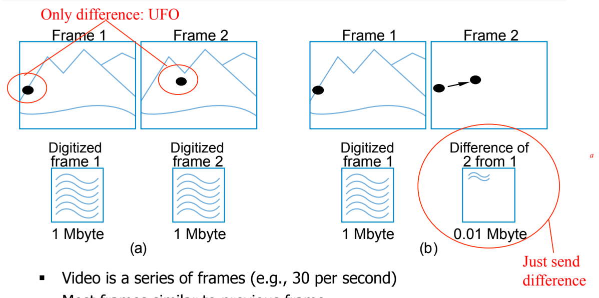
- FSM has same structure as high-level state machine
  - Inputs/outputs all bits now
  - Replace data operations by bit operations using datapath


Inputs: B, S (1 bit each)
Outputs: L (bit), D (16 bits)
Local Registers: Dctr (16 bits)
B'
S'
S0
S1
B
S2
S3
S
S4
L = 0
D = 0
Dctr = 0
L = 1
L=0
Dctr = Dctr + 1
D = Dctr / 2
a





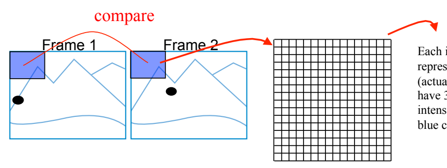
Video Compression – Sum of Absolute Differences



- Video is a series of frames (e.g., 30 per second)
- Most frames similar to previous frame
  - Compression idea: just send difference from previous frame

Video Compression – Sum of Absolute Differences

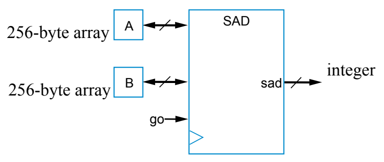
- Need to quickly determine whether two frames are similar enough to just send difference for second frame
  - Compare corresponding 16x16 "blocks"
    - Treat 16x16 block as 256-byte array
  - Compute the absolute value of the difference of each array item
  - Sum those differences – if above a threshold, send complete frame for second frame; if below, can use difference method (using another technique, not described)



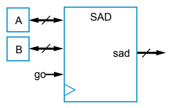
Each is a pixel, assume represented as 1 byte (actually, a color picture might have 3 bytes per pixel, for intensity of red, green, and blue components of pixel)

Video Compression – Sum of Absolute Differences

- Want fast sum-of-absolute-differences (SAD) component
  - When *go=1*, sums the differences of element pairs in arrays *A* and *B*, outputs that sum

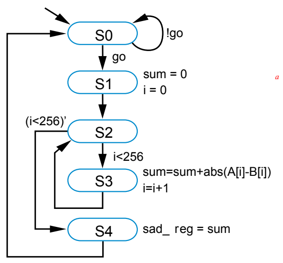


Video Compression – Sum of Absolute Differences



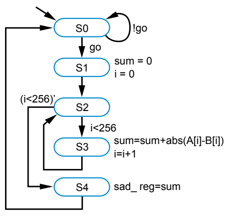
- Step 1: Create high-level state machine
  - **S0** - wait for *go*
  - **S1**- initialize *sum* and *index*
  - **S2** - check if done (*i>=256*)
  - **S3** - add difference to *sum*, increment index
  - **S4** - done, write to output *sad_reg*

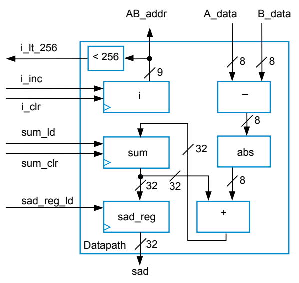
Inputs: A, B (256 byte memory); go (bit)
Outputs: sad (32 bits)
Local registers: sum, sad_reg (32 bits); i (9 bits)



Video Compression – Sum of Absolute Differences

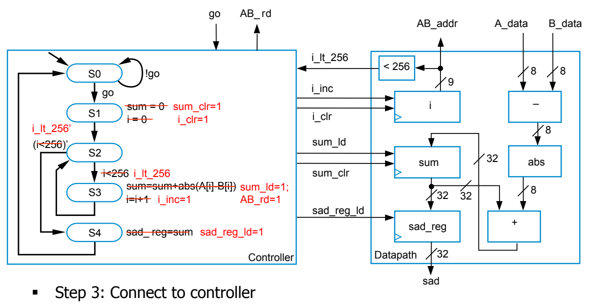
Inputs: A, B (256 byte memory); go (bit)
Outputs: sad (32 bits)
Local registers: sum, sad_reg (32 bits); i (9 bits)





- Step 2: Create datapath

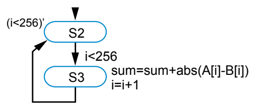
Video Compression – Sum of Absolute Differences



- Step 3: Connect to controller
- Step 4: Replace high-level state machine by FSM

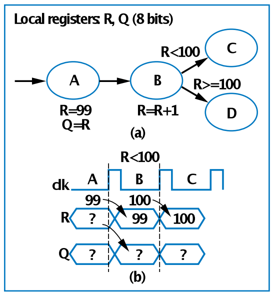
Video Compression – Sum of Absolute Differences

- Comparing software and custom circuit SAD
  - Circuit: Two states (**S2** & **S3**) for each *i*, 256 *i*'s→ 512 clock cycles
  - Software: Loop (*for i = 1 to 256*), but for each *i*, must move memory to local registers, subtract, compute absolute value, add to sum, increment *i* – say about 6 cycles per array item → 256*6 = 1536 cycles
  - Circuit is about *3 times* (300%) faster



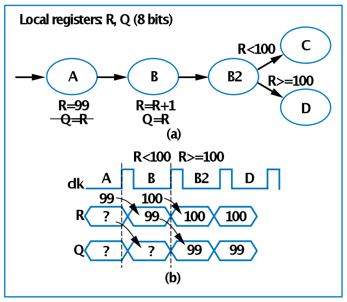
RTL Design Pitfalls and Good Practice

- Common pitfall: Assuming register is update in the state it's written
- Example
  - Final value of *Q*? Final state?
  - Answers may surprise you
    - Value of *Q* unknown
    - Final state is **C**, not **D**
- Why?
  - State **A**: *R=99* and *Q=R* happen simultaneously
  - State **B**: *R* not updated with *R+1* until next clock cycle, simultaneously with state register being updated



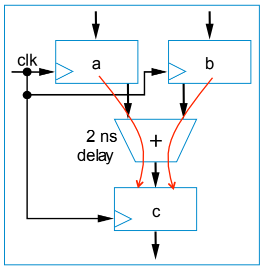
RTL Design Pitfalls and Good Practice

- Solutions
  - Read register in following state (*Q=R*)
  - Insert extra state so that conditions use updated value
  - Other solutions are possible, depends on the example



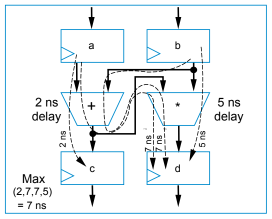
Determining Clock Frequency

- Designers of digital circuits often want fastest performance
  - Means want high clock frequency
- Frequency limited by ***longest register-to-register delay***
  - Known as ***critical path***
  - If clock is any faster, incorrect data may be stored into register
  - Longest path on right is 2 ns
    - Ignoring wire delays, and register setup and hold times, for simplicity



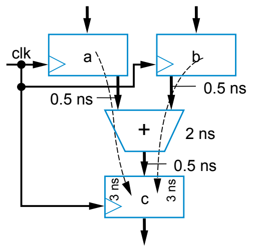
Critical Path

- Example shows four paths
  - a to c through + (2 ns)
  - a to d through + and * (7 ns)
  - b to d through + and * (7 ns)
  - b to d through * (5 ns)
- Longest path is thus 7 ns
- Fastest frequency
  - 1 / 7 ns = 142 MHz



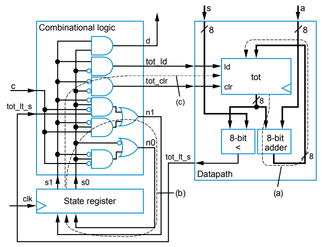
Critical Path Considering Wire Delays

- Real wires have delay too
  - Must include in critical path
- Example shows two paths
  - Each is 0.5 + 2 + 0.5 = 3 ns
- Trend
  - 1980s/1990s: Wire delays were tiny compared to logic delays
  - But wire delays not shrinking as fast as logic delays
    - Wire delays may even be greater than logic delays!
- Must also consider register setup and hold times, also add to path
- Then add some time to the computed path, just to be safe
  - e.g., if path is 3 ns, say 4 ns instead



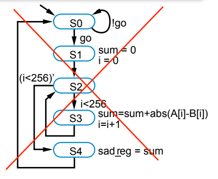
A Circuit May Have Numerous Paths

- Paths can exist
  - In the datapath
  - In the controller
  - Between the controller and datapath
  - May be hundreds or thousands of paths
- Timing analysis tools that evaluate all possible paths automatically very helpful



Behavioral Level Design: C to Gates

- Earlier sum-of-absolute-differences example
  - Started with high-level state machine
  - C code is an even better starting point -- easier to understand



C code

```
int SAD (byte A[256], byte B[256]) // not quite C syntax
{
    uint sum; short uint I;
    sum = 0;
    i = 0;
    while (i < 256) {
        sum = sum + abs(A[i] – B[i]);
        i = i + 1;
    }
   return sum;
}
```

Behavioral-Level Design
Start with C (or Similar Language)

- Replace first step of RTL design method by two steps
  - Capture in C, then convert C to high-level state machine
  - How convert from C to high-level state machine?

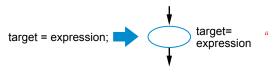
Step 1A: Capture in C

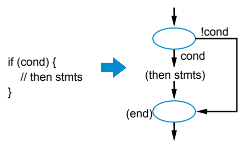
Step 1B: Convert to high-level state machine

	Step	Description
Step 1	~~*Capture a high-level state machine*~~	~~Describe the system's desired behavior as a high-level state machine. The state machine consists of states and transitions. The state machine is "high-level" because the transition conditions and the state actions are more than just Boolean operations on bit inputs and outputs.~~
Step 2	*Create a datapath*	Create a datapath to carry out the data operations of the high-level state machine.
Step 3	*Connect the datapath to a controller*	Connect the datapath to a controller block. Connect external Boolean inputs and outputs to the controller block.
Step 4	*Derive the controller's FSM*	Convert the high-level state machine to a finite-state machine (FSM) for the controller, by replacing data operations with setting and reading of control signals to and from the datapath.

Converting from C to High-Level State Machine

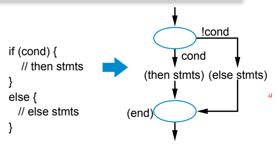
- Convert each C construct to equivalent states and transitions
- *Assignment* statement
  - Becomes one state with assignment
- *If-then* statement
  - Becomes state with condition check, transitioning to "then" statements if condition true, otherwise to ending state
    - "then" statements would also be converted to states

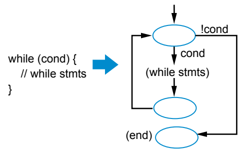




Converting from C to High-Level State Machine

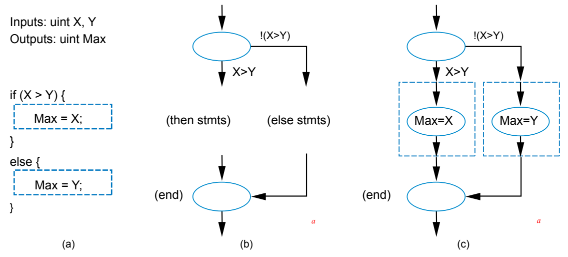
- *If-then-else*
  - Becomes state with condition check, transitioning to "then" statements if condition true, or to "else" statements if condition false
- *While loop* statement
  - Becomes state with condition check, transitioning to while loop's statements if true, then transitioning back to condition check

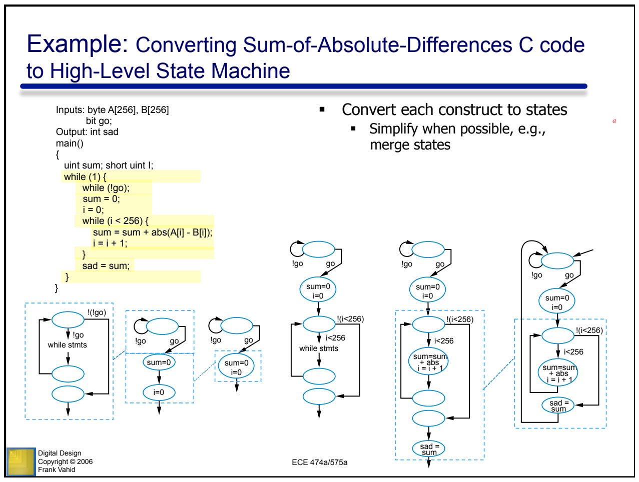




Simple Example of Converting from C to High-Level State Machine

- Simple example: Computing the maximum of two numbers
  - Convert if-then-else statement to states (b)
  - Then convert assignment statements to states (c)



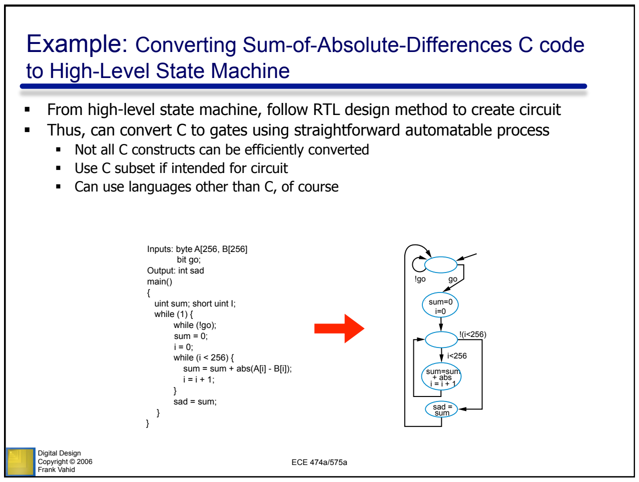


Example: Converting Sum-of-Absolute-Differences C code to High-Level State Machine

```
Inputs: byte A[256], B[256]
        bit go;
Output: int sad
main()
{
  uint sum; short uint I;
  while (1) {
      while (!go);
      sum = 0;
      i = 0;
      while (i < 256) {
         sum = sum + abs(A[i] - B[i]);
         i = i + 1;
      }
      sad = sum;
  }
}
```

- Convert each construct to states
  - Simplify when possible, e.g., merge states

!(!go)
!go
while stmts
!go
go
sum=0
i=0
!go
go
sum=0
i=0
!go
go
sum=0
i=0
!(i<256)
i<256
while stmts
!go
go
sum=0
i=0
!(i<256)
i<256
sum=sum + abs i = i + 1
sad = sum
!go
go
sum=0
i=0
!(i<256)
i<256
sum=sum + abs i = i + 1
sad = sum



Example: Converting Sum-of-Absolute-Differences C code to High-Level State Machine

- From high-level state machine, follow RTL design method to create circuit
- Thus, can convert C to gates using straightforward automatable process
  - Not all C constructs can be efficiently converted
  - Use C subset if intended for circuit
  - Can use languages other than C, of course

```
Inputs: byte A[256, B[256]
        bit go;
Output: int sad
main()
{
  uint sum; short uint I;
  while (1) {
      while (!go);
      sum = 0;
      i = 0;
      while (i < 256) {
         sum = sum + abs(A[i] - B[i]);
         i = i + 1;
      }
      sad = sum;
  }
}
```