5.1 HLSMs: Introduction

An FSM has single-bit inputs and outputs. Sometimes a sequential system has multibit data input and output, such as a system with 8-bit data input D. An extended FSM allows defining data inputs and outputs, and allows actions that assign a data output with an arithmetic expression, like T = D + 5.

PARTICIPATION ACTIVITY 5.1.1: Extended FSM with data input and output: A value-booster example.

Animation captions:

1. Extended FSM has an 8-bit data input D, and 8-bit data output T.
2. State RawL assigns T with D.
3. Until a button press is detected, the extended FSM remains in State RawL.
4. BoostH's state action assigns T with D + 5.

A multi-bit wire is typically drawn as a single wire with a slash (/), as above.

The above extended FSM implements the common functionality of "toggling" between two situations, in this case assigning T with either D or with D + 5 (implementing for example a sound "boost" button on a music amplifier).

PARTICIPATION ACTIVITY 5.1.2: Extended FSM.

Consider the above extended FSM.

1) Input D is 4 bits.
   - True
   - False

2) The slash through D's input wire means that wire represents multiple bits.
   - True
   - False

3) Input b is 8 bits.
   - True
   - False

4) If D is 00110000, T = D + 5 assigns T

with 00110101.

- True
- False

5) Assume D is 00000000, and present state is RawL. If, on the next four rising clock edges, b is 1, then 0, then 1, then 0, then output T is 00000101.

- True
- False

An extended FSM can also allow transition conditions to have arithmetic expressions that evaluate to true (1) or false (0).

PARTICIPATION ACTIVITY 5.1.3: Extended FSM with data inputs in transition conditions: A threshold alarm.

Animation captions:

1. Extended FSM's data inputs may appear in transition conditions.
2. Expressions evaluate to true or false.
3. True/false arithmetic expressions can be combined with single-bit inputs.
4. Transition conditions are re-evaluated on every rising clock edge.

The above implements the common behavior of responding (perhaps sounding an alarm) if a data input exceeds some value called a threshold. Examples include a home alarm that sounds if detected motion exceeds a threshold, or a car alarm that sounds if detected vibration exceeds a threshold.

PARTICIPATION ACTIVITY 5.1.4: Extended FSM with data inputs in transition conditions.

Consider the above threshold-alarm extended FSM.

1) If E is 0011 (3), to what does (E > 6)' evaluate? Type true or false.

Check Show answer

2) The state is initially OK and input E is initially 3. E changes to 8. What is the next state?

Check Show answer

3) When in state Alarm, what is output Z?

Check Show answer

4) Suppose state is Alarm, E is 8, b is 0. b changes to 1. What is the next state?

Check Show answer

5) If in the Alarm state and E is 8, does pressing the button shut off the alarm? Answer yes or no.

Check Show answer

6) If in the Alarm state, E becomes 2, b is 0, and then the clock rises, does the alarm still sound? Answer yes or no.

Check Show answer

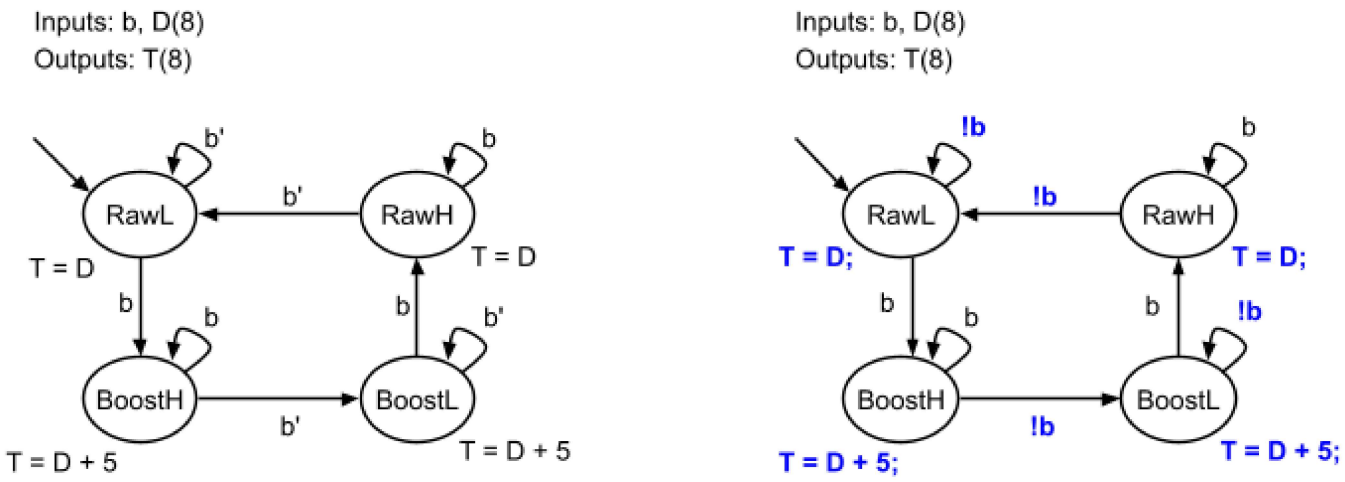
A ***high-level state machine*** is a form of extended FSM that supports multi-bit data input/output and data operations. ***HLSM*** is short for high-level state machine.

Some HLSMs use notation from the C programming language to avoid confusion between arithmetic and Boolean operations. Ex: ((A + B) > 0) + d uses + for both addition and OR, which is confusing; C notation would be ((A + B) > 0) || d, where + means addition and || means OR. *C programming knowledge is not necessary to understand this material's HLSMs.*

Table 5.1.1: C notation for HLSMs.

Item	C	Notes
AND: ab, a*b	a && b	
OR: a + b	a || b	The || is made from two vertical bar characters, not from the letter "el" (l) or the letter "ay" (I).
NOT: a'	!a	The ! comes before a, rather than after.
Equality	==	a == b compares; true if same. In contrast, y = a assigns y with a's value.
End of action	y = a; x = b;	In C notation, each action is a "statement" that ends with a semicolon.

Figure 5.1.1: HLSM notation: A value-booster example.



Extended FSM notation

HLSM notation

PARTICIPATION ACTIVITY 5.1.5: HLSM notation.

Convert, as directly as possible, the state action or transition condition to C notation used in HLSMs.
Inputs: a, b. Outputs: x, y.

1) ab

Check Show answer

2) a + b

Check Show answer

3) y = a

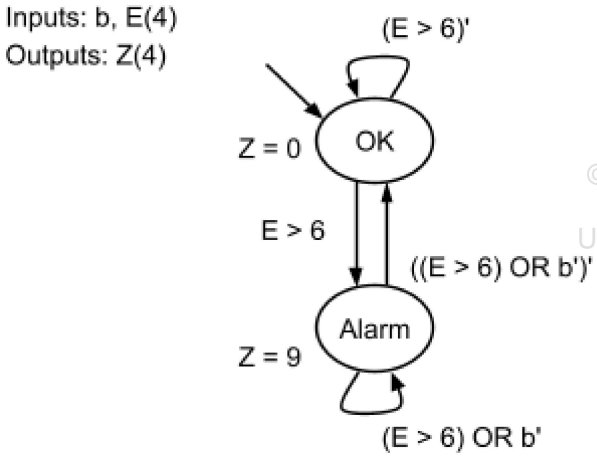
Check Show answer

4) a EQUALS b
(on a transition condition)

Check Show answer

PARTICIPATION ACTIVITY 5.1.6: HLSM notation.

Convert the shown extended FSM to use HLSM notation.



1) E > 6

- E > 6
- E > 6;

2) (E > 6)'

- !(E > 6)
- (E > 6)'

3) (E > 6) OR b'

- (E > 6) + !b
- (E > 6) || !b

4) ((E > 6) OR b')'

- ((E > 6) || b!)!
- !((E > 6) || !b)

5.2 HLSMs with variables

An HLSM supports not just data inputs/outputs, but also variables. A ***variable*** is a data item that maintains a value, and can be read or assigned.

PARTICIPATION ACTIVITY 5.2.1: An HLSM supports variables: Pulse counter.

Animation captions:

1. HLSMs can declare variables.
2. A variable can be assigned a value.
3. A variable's value can be read.
4. In fact, a variable can be read, used in an expression, and assigned again.
5. A timing diagram can be extended to show a variable's value in each state.
6. I is read and then assigned.
7. The variable maintains a value throughout the HLSM's execution.

The above is an example of the common HLSM behavior of incrementing a counter variable each time an input pulses (changes from 0 to 1 and back to 0), as in a system that counts people walking through a door or cars driving over a sensor.

PARTICIPATION ACTIVITY 5.2.2: Pulse counter HLSM.

Refer to the above pulse-counting HLSM.

1) When in the initial state, l = ?
   - 1
   - 0

2) Present state is Init, clock rises. After entering WaitRise and executing WaitRise's actions, what is Y?
   - 0
   - 1

3) Present state is WaitRise, b is 1, l is 0, clock rises. After entering CountUp and executing CountUp's actions, what is l?
   - 0
   - 1

4) Present state is WaitFall, b is 0, l is 1, clock rises. After executing the next state's actions, what is Y?
   - 0
   - 1

The following HLSM is an example of a simple network data analyzer. Data flows between computers via a network. When one computer sends data over the network, the computer pulses the newData signal. The data analyzer counts the number of occurrences of a particular data value known as a pattern. Perhaps the pattern is the Internet address of a suspected terrorist, being monitored by law enforcement.

PARTICIPATION ACTIVITY 5.2.3: HLSM with variables: Network data analyzer.

Animation captions:

1. If newData rises and Pattern equals DataIn (and rst is not 1), next state is DataMatch, which increments variable Count.
2. But if newData rises and Pattern doesn't equal DataIn, next state is just WaitFall, so no increment occurs.

3. Output Matches is assigned with Count in WaitRise. HLSM thus outputs numbers of matches detected. rst resets values to 0.

PARTICIPATION ACTIVITY 5.2.4: Network data analzyser HLSM.

Refer to the above network data analyzer HLSM.

1) Present state is Reset, rst is 0, clock rises. After entering and executing the next state's actions, Matches = ?
   - 1
   - 0

2) Present state is WaitRise, rst is 0, newData is 1, Pattern is 5, DataIn is 5. After the next rising clock edge, what is the next state?
   - DataMatch
   - WaitFall

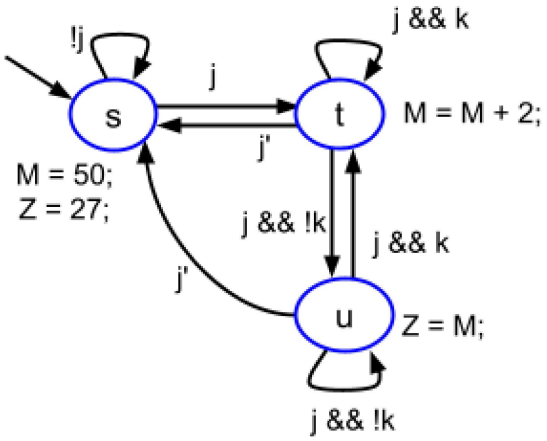
3) Present state is WaitRise, rst is 1. After the next rising clock edge, what is the next state?
   - Reset
   - WaitFall

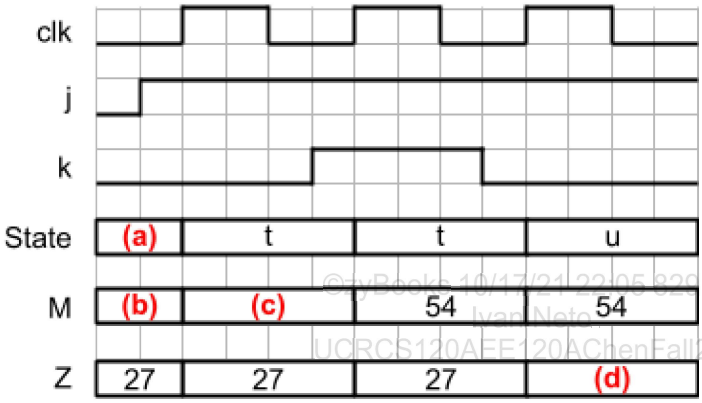
4) Present state is WaitFall, Count is 1, newData is 0, clock rises. After entering and executing the next state's actions, Matches = ?
   - 1
   - 0

PARTICIPATION ACTIVITY 5.2.5: HLSM introduction.

Complete the timing diagram.

Inputs: j, k
Outputs: Z(8)
Variable: M(8)





1) (a)

Check Show answer

2) (b)

Check Show answer

3) (c)

Check Show answer

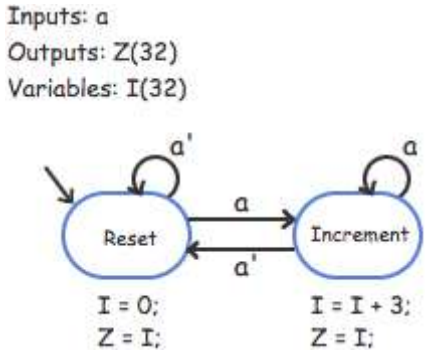
4) (d)

Check Show answer

CHALLENGE ACTIVITY 5.2.1: Indicate Z's value over time.

347136.1658324.qx3zqy7

Start



a

clk

Z 0 0 0 0 0 0 0

1 2 3 4 5 6

Check Next

5.3 HLSMs with a loop

The below HLSM involves a common state/transition/variable pattern similar to a "for" loop in programming languages. A variable I is used to execute a particular state 5 times. The below HLSM uses a loop to generate an output pulse lasting 5 clock cycles.

PARTICIPATION ACTIVITY 5.3.1: HLSM example with common 'for' loop behavior: Pulse generator.

Animation captions:

1. Common 'for' loop structure.
2. Wait state waits for b to become 1.
3. b is 1 on next clock edge, so next state is GenPulse. State actions are executed.
4. I is less than 5 so the transition is taken.

5. Keep counting until I less than 5 is false.
6. (5 < 5) is false. Transition from GenPulse to Wait is taken on rising clock edge.
7. HLSM sets output z to 1 for five clock cycles.

PARTICIPATION ACTIVITY 5.3.2: Pulse generator HLSM.

Refer to the above HLSM describing 'for' loop behavior to generate a pulse.

1) When in the initial state, what is z's value?
   - 0
   - 1

2) When in the initial state, what is I's value?
   - 0
   - 1

3) Present state is Wait, b is 1, clock rises. After executing next state's actions, what is I?
   - 0
   - 1

4) Present state is GenPulse, I is 4, clock rises. After entering GenPulse and executing GenPulse's actions, what does (I < 5) evaluate to?
   - True
   - False

5) Present state is GenPulse, I is 5, clock rises. After entering the next state and executing the state's actions, I = ?
   - 0
   - 5

6) The HLSM is modified to set z to 1 for three clock cycles by replacing the statement (I < 5) with _____.
   - (I < 2)

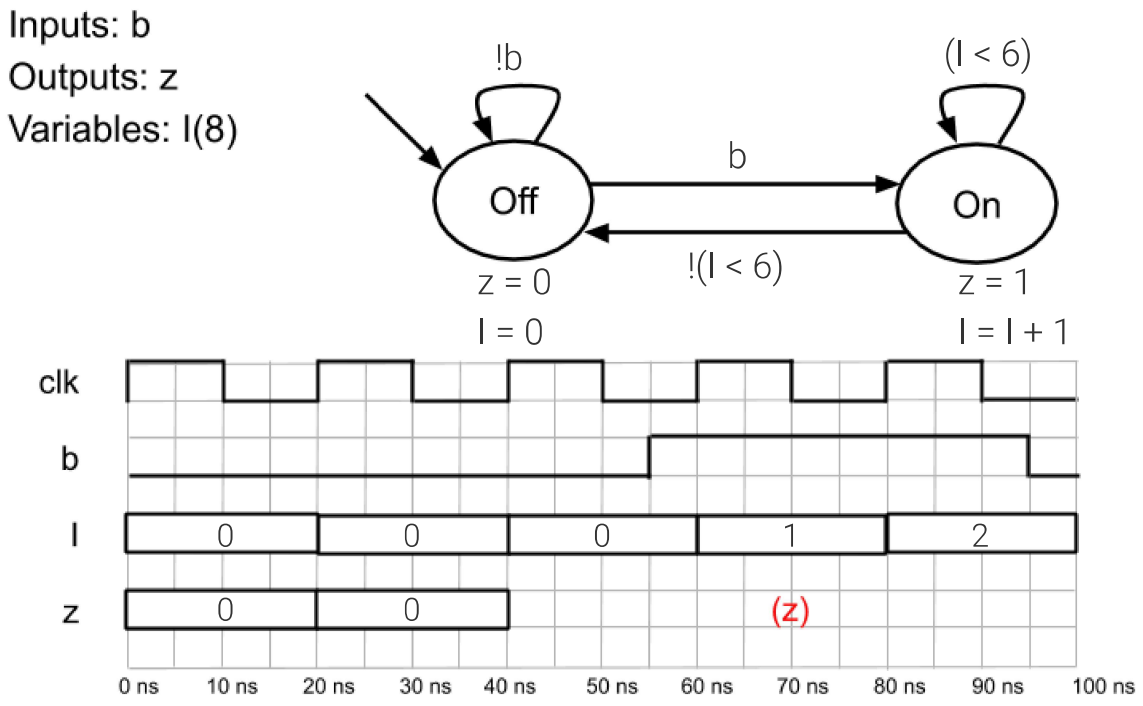
○ (I < 3)

CHALLENGE ACTIVITY 5.3.1: HLSMs with a loop.

347136.1658324.qx3zqy7

Start

Indicate the value of z for the HLSM for the missing clock cycles.



(z) = Ex: 100

1 2

Check Next

5.4 HLSM simulator

This material includes a simple browser-based HLSM simulator.

PARTICIPATION ACTIVITY 5.4.1: HLSM simulator.

Full screen

- Press "Simulate". Increase input E above 100, note state change and new value on output Z. Press "Simulate".
- Insert a new state Init that assigns u, v, w, and x with 1's, assigns Y = 9 and Z = 0, and make Init the initial state. Simulate again.

Simulate | Pause | Insert state | Insert transition

LoHi ✖ +

a 0 ☐

b 0 ☐

c 0 ☐

d 0 ☐

E(8)
0

F(8)
0

Export | Import

Paste design here.

PARTICIPATION ACTIVITY 5.4.2: HLSM simulator basics.

1) "Period" indicates the HLSM's clock period.
   - True
   - False

2) The HLSM's initial state is set by double-clicking the state.
   - True
   - False

3) After inserting a state, actions can be included by typing in the "Actions" box on the far right.
   - True
   - False

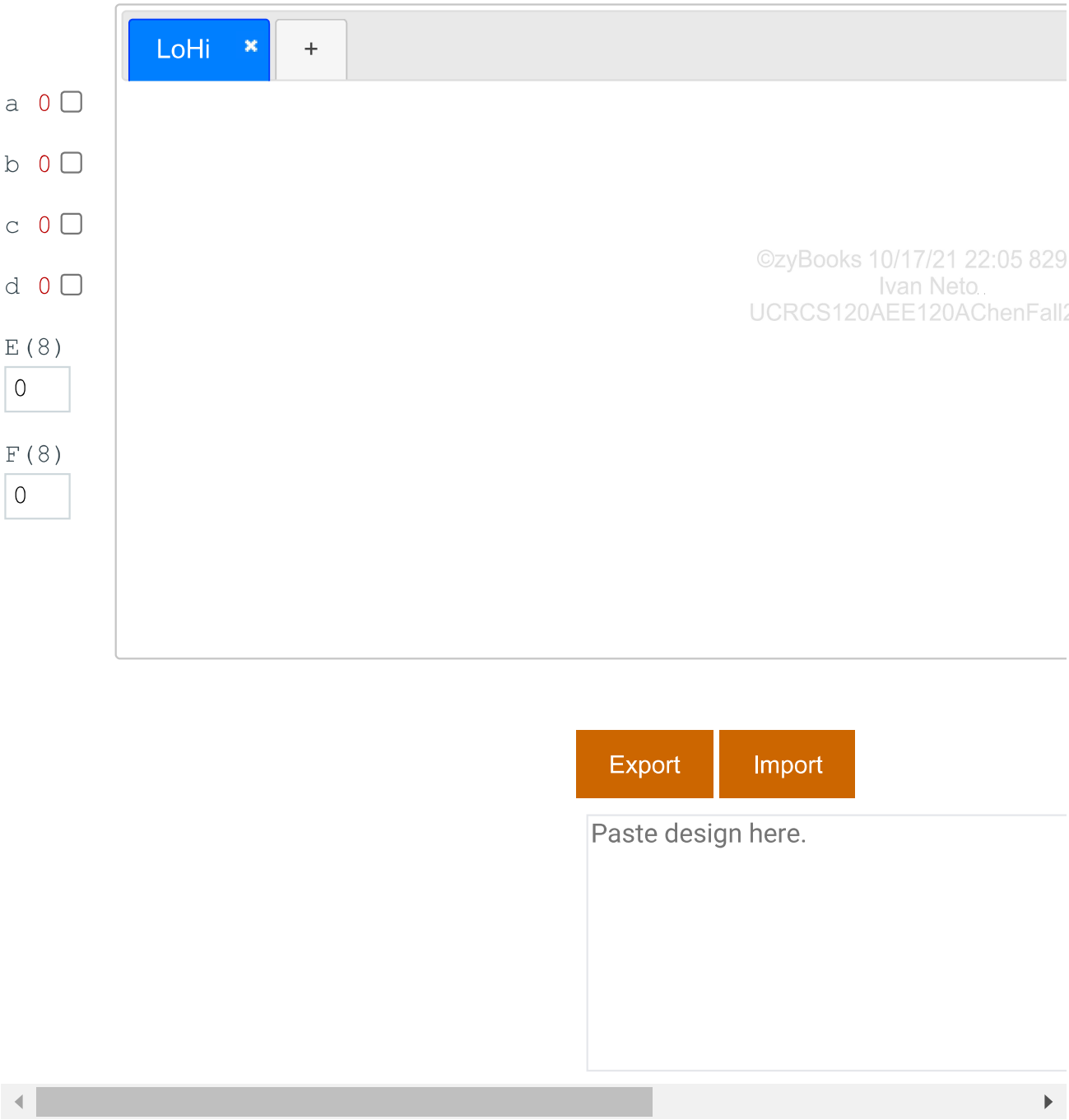
4) A state or transition is deleted by dragging the item off-screen.
   - True
   - False

5) When an HLSM is executing (by pressing "Simulate"), the values of inputs a, b, c, d, E, and F, cannot be changed.
   - True
   - False

PARTICIPATION ACTIVITY 5.4.3: HLSM simulator with export/import. **Full screen**

- HLSMs can be saved. Press "Export" and copy-paste the exported text. Modify the HLSM somehow. Then paste the text into the box and press "Import" -- the previously-exported HLSM is restored. (Users can save the exported text in a file or email for future use.)

Simulate Pause Insert state Insert transition



PARTICIPATION ACTIVITY 5.4.4: HLSM simulator with export.

1) Pressing "Export" saves the current HLSM to a file.

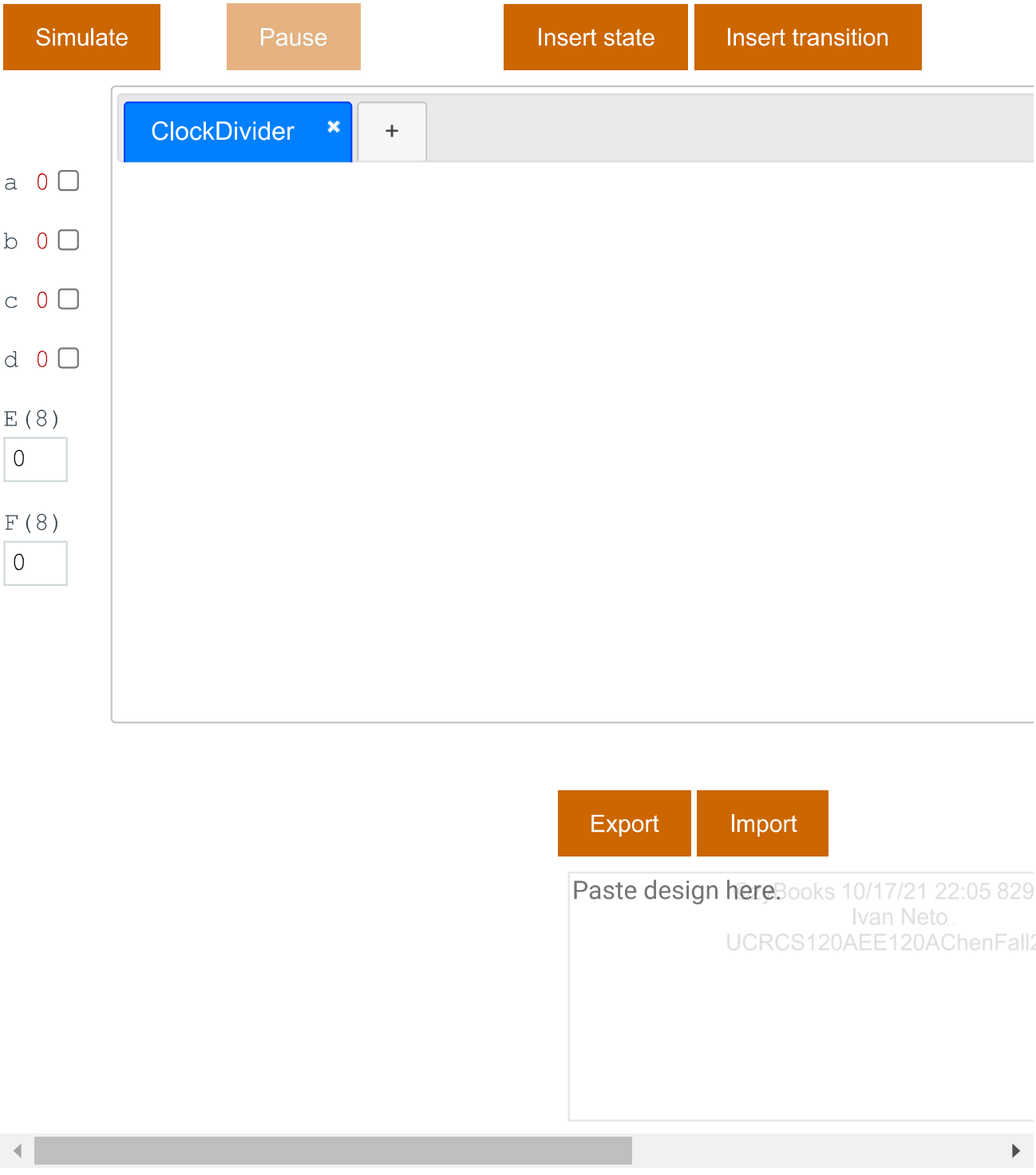
- True
- False

5.5 Capturing behavior with HLSMs

A ***clock divider*** converts a clock signal into a lower-frequency clock signal. Ex: Given a 1 MHz clock signal, a clock divider might divide by 1,000,000 to output a 1 Hz clock (ticks every 1 second). The following HLSM divides a 2 Hz clock by 10 to yield a 0.2 Hz clock (500 ms period becomes 5 sec). Note the HLSM stays in s1 for 9 cycles, not 10, because s0 uses 1 cycle too.

PARTICIPATION ACTIVITY 5.5.1: Example HLSM: Clock divider.

Full screen



PARTICIPATION ACTIVITY 5.5.2: Clock divider HLSM.

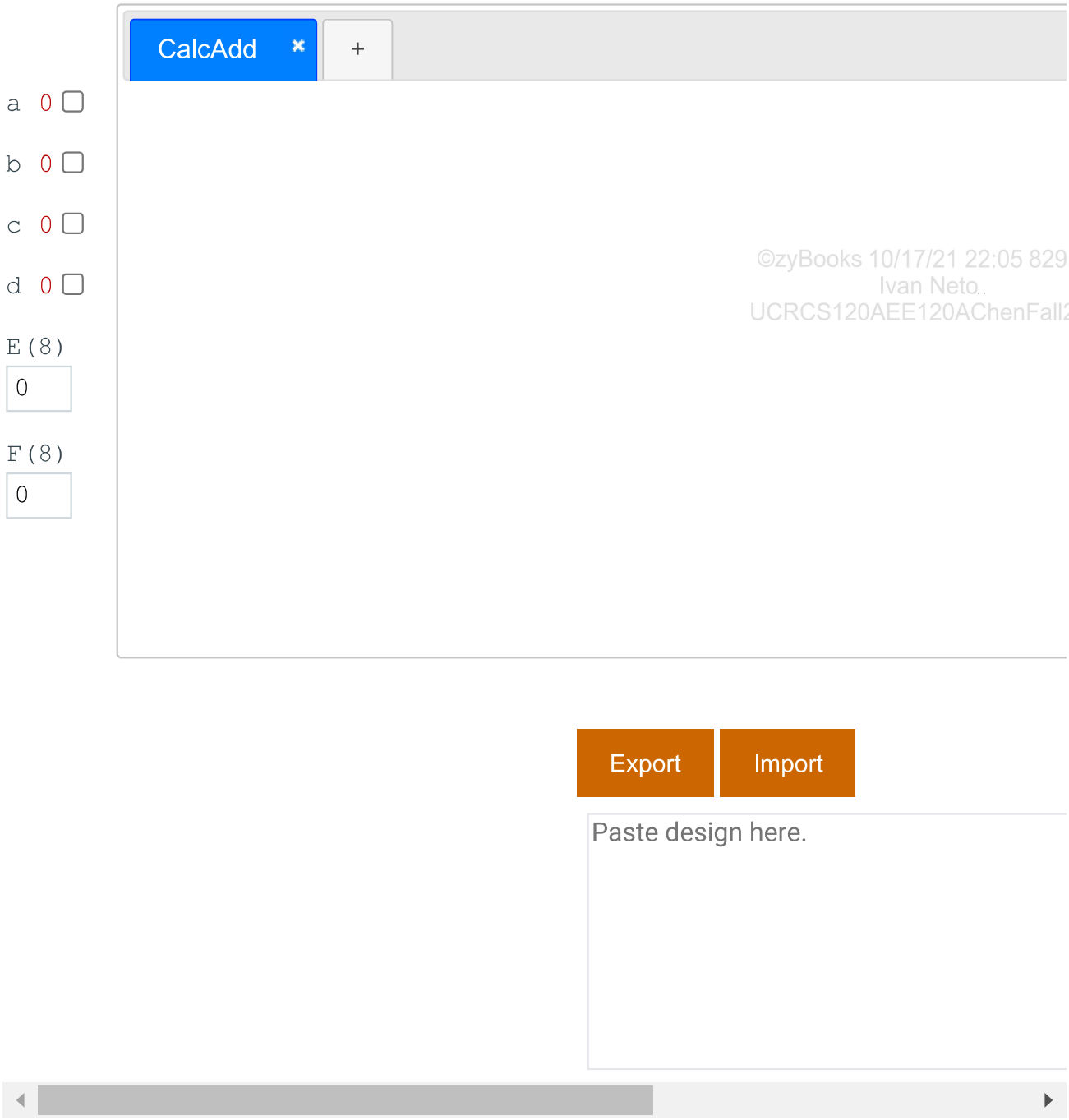
1) To modify the clock divider HLSM to divide by 50 rather than 10, the transition condition from s1 to s0 should be _____.

Check Show answer

A ***calculator*** executes a user's desired computation like add, subtract, etc. In the following, the user enters a number on E. Input a means +, and d means =. The result is output on Y. To add 21 + 19, the user sets E to 21, presses a (+), sets E to 19, and presses d (=) to yield 40 on Y.

PARTICIPATION ACTIVITY 5.5.3: Example HLSM: Calculator (addition only). Full screen

Simulate Pause Insert state Insert transition



PARTICIPATION ACTIVITY 5.5.4: Calculator.

Refer to the above calculator HLSM.

1) To add 4 + 7, the user first sets a to 1, then types 4 into E.
   - ○ True
   - ○ False

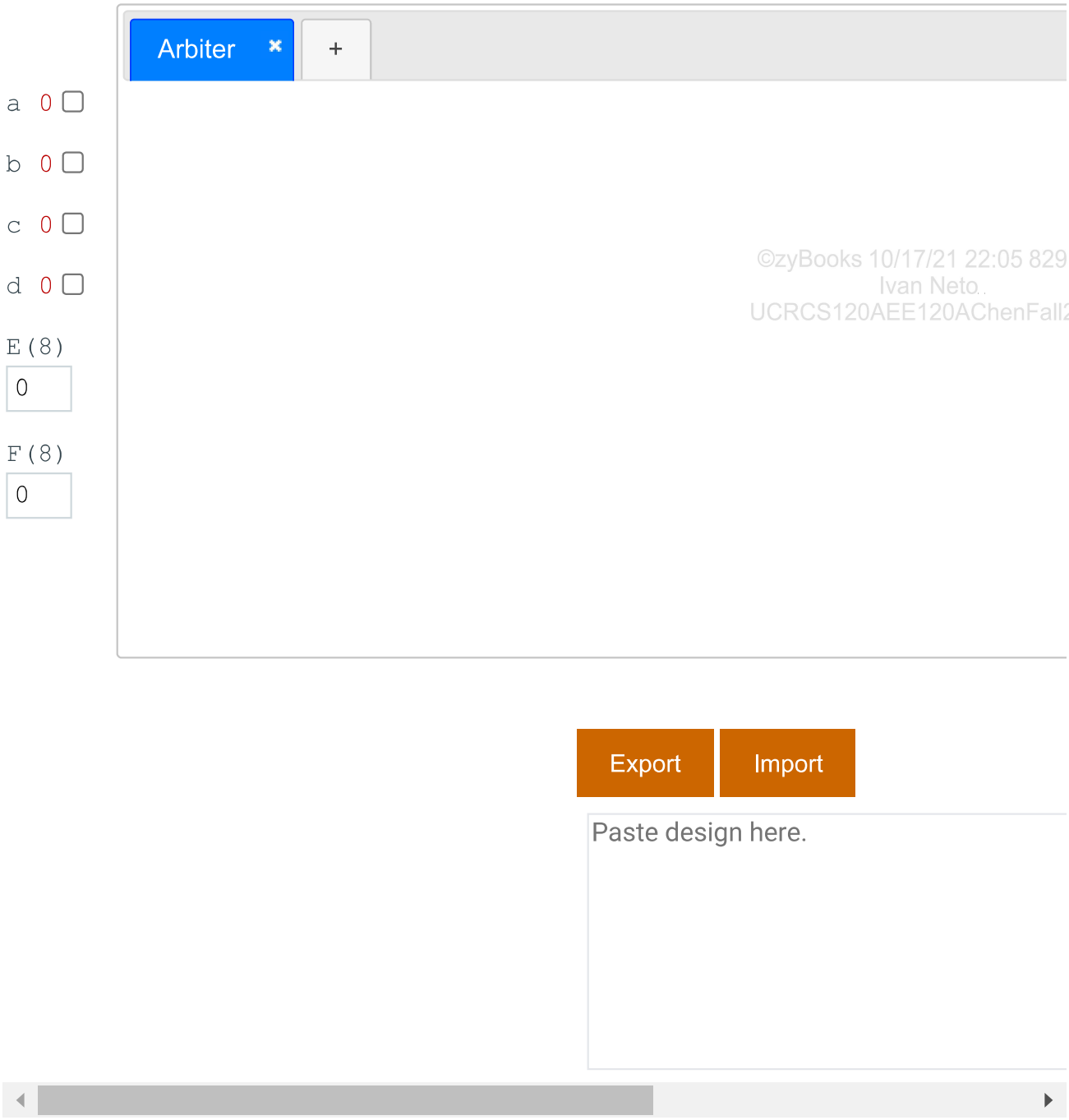
2) To add 4 + 7, the user will type 4 into E and 7 into F.

- True
- False

An ***arbiter*** decides which of multiple requestors gets access to a shared resource. An arbiter is akin to a phone operator who decides which of multiple callers gets to talk to a particular person. In the following, requestor A requests access to a shared resource by setting a to 1. Requestor B sets b to 1. The arbiter grants access to A by setting u to 1, or to B by setting v to 1. The arbiter decides using priority. ***Priority*** indicates a requestor's importance compared to other requestors. E indicates requestor A's priority, F indicates requestor B's. Larger values mean higher priority.

PARTICIPATION ACTIVITY 5.5.5: Example HLSM: Arbiter. Full screen

Simulate Pause Insert state Insert transition



PARTICIPATION ACTIVITY 5.5.6: Arbiter.

Refer to the above arbiter.

1) What is output when a = 1, b = 0?

- u = 0, v = 1
- u = 1, v = 0
- u = 1, v = 1

2) What is the output when both a and b change to 1 at the same time and E =

5, F = 4.

- u = 0, v = 1
- u = 1, v = 0

3) After a = 1 and b = 1 with E = 5 and F = 4, A completes access so sets a = 0. What is then output?

- u = 0, v = 1
- u = 1, v = 0

CHALLENGE ACTIVITY 5.5.1: Capturing behavior with an HLSM. Full screen

347136.1658324.qx3zqy7

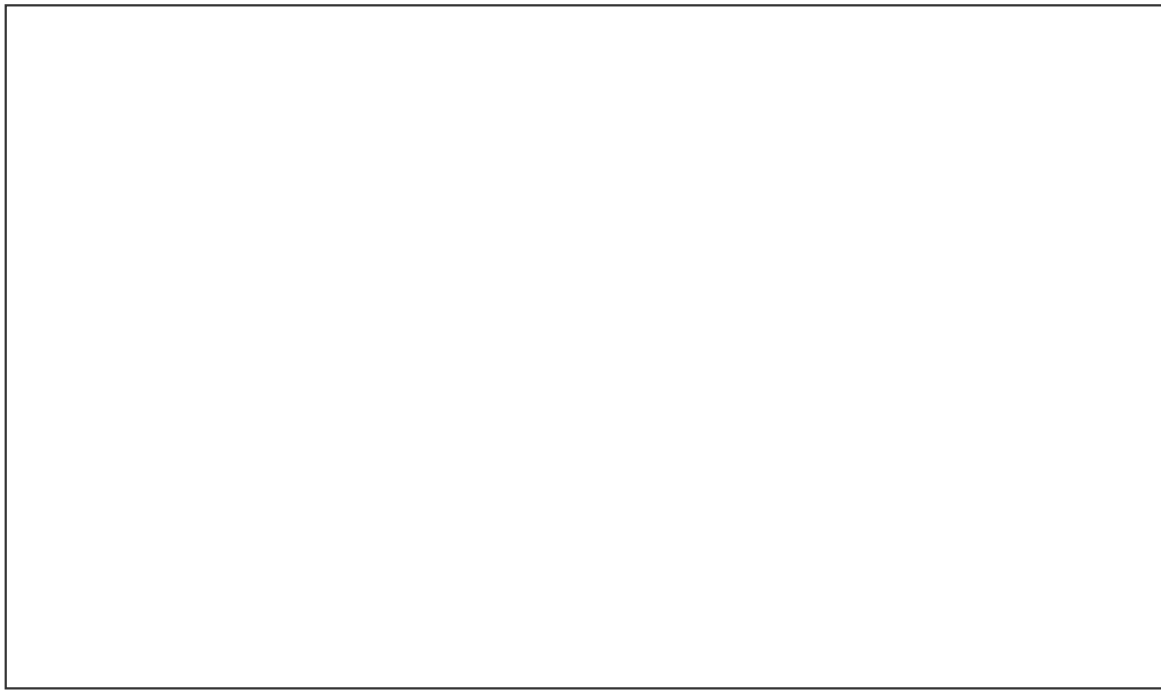
Start

Simulate Pause Insert state Insert transition

a 0
b 0
c 0
d 0

E(8)
0

F(8)
0



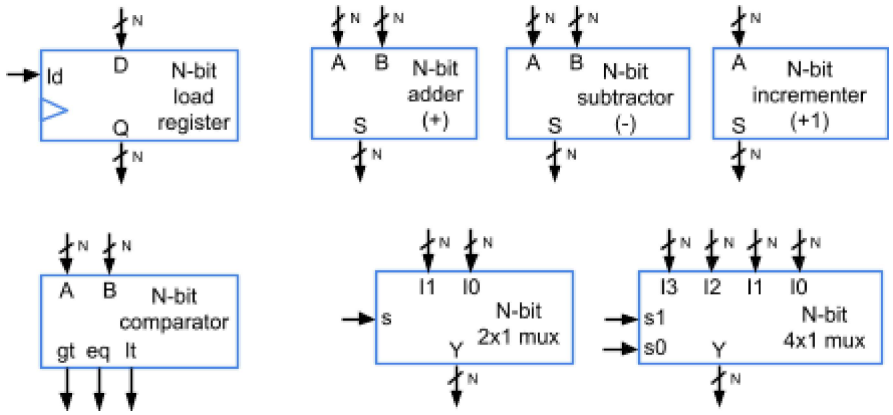
1 2 3

Check Next

5.6 Datapaths for HLSMs

Implementing an HLSM as a circuit requires first creating a datapath. A ***datapath*** is a circuit that can support the data operation and data storage parts of system behavior, the datapath being built from components like registers, adders, subtractors, incrementers, comparators, and muxes.

Figure 5.6.1: Sample available datapath components.



Each data operation can be supported by instantiating an appropriate datapath component, along with instantiating wires and muxes as needed. ***Instantiate*** means to introduce an item into a circuit. Each instantiated item is called an ***instance***.

Minimally, every data output requires a register to hold the output's value between rising clock edges. Other components are instantiated as needed.

PARTICIPATION ACTIVITY 5.6.1: Datapath support for data operation in an action.

Animation captions:

1. Datapath will support any operations involving data.
2. Each data output requires a register to hold the output's value.
3. Support RawL's state action by instantiating a wire from D to T_reg.
4. Support BoostH's state action of D + 5 by instantiating an adder.
5. Support T = D + 5 by instantiating a mux, because T_reg now has two possible input sources.
6. The actions of RawH and BoostL are already supported by the datapath.
7. Since wire size is shown, component sizes are clear and are typically not shown.

Note in the above animation that a number like 5 (00000101 in binary) can be input directly to a component.

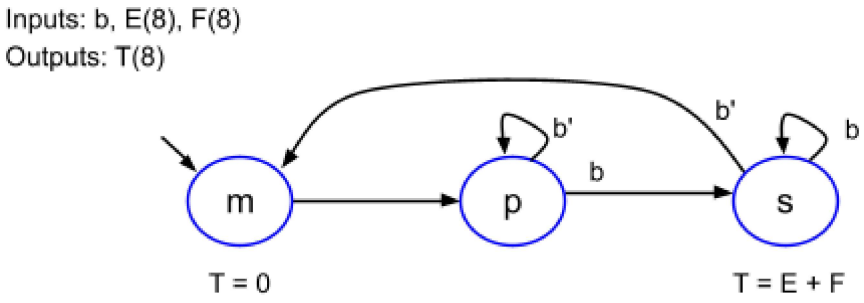
PARTICIPATION ACTIVITY 5.6.2: Datapath support for variables.

Animation captions:

1. Every data output and variable requires a register.
2. Support Init's state actions by connecting a 0 to each register's input.
3. Support Y = I by instantiating a mux, because Y_reg now has two possible input sources: 0 and I.
4. Support I + 1 by instantiating an adder.
5. Support I = I + 1 by instantiating a mux, because I_reg now has two possible input sources: 0 and I + 1.
6. Datapath supports any operations involving data.

PARTICIPATION ACTIVITY 5.6.3: Datapath support for data operations.

Given the following HLSM:



1) The data operation E + F should be supported in the datapath.
   - True
   - False

2) b is an input to the datapath.
   - True
   - False

3) The datapath would require an 8-bit 2x1 mux to support both T = 0 and T =

E + F.

- ○ True
- ○ False

4) The datapath would not instantiate any registers.

- ○ True
- ○ False

PARTICIPATION ACTIVITY 5.6.4: Datapath support for data operations in transition conditions.

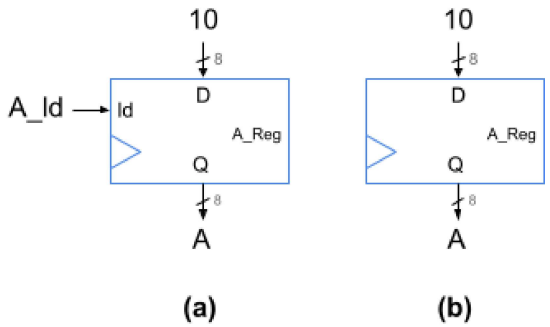
Animation captions:

1. Datapath supports any data input or output.
2. Any data operation in a transition's condition must be supported by the datapath.
3. Other conditions have the same data operation; datapath already supports.
4. State actions are supported using a 2x1 mux.
5. Note that lt and eq are unused. Not all outputs must be connected.

PARTICIPATION ACTIVITY 5.6.5: Instantiating datapath components.

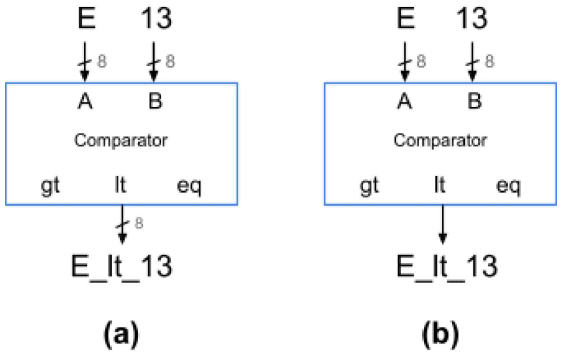
Select the datapath that best implements the given data operations. Assume all datapath inputs and outputs are 8 bits.

1) Data operation: A = 10
(Note: The assignment operation is NOT executed in every state).



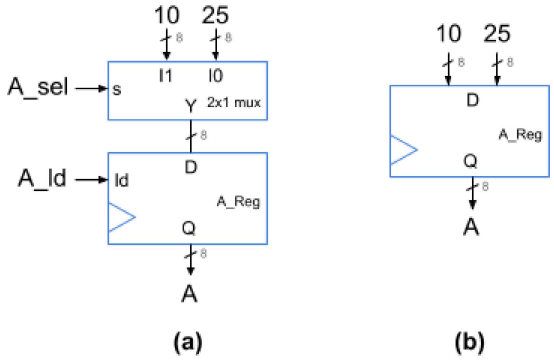
- ○ (a)
- ○ (b)

2) Data operation: E < 13



- (a)
- (b)

3) Data operation in state s: A = 10
Data operation in state t: A = 25



- (a)
- (b)

PARTICIPATION ACTIVITY 5.6.6: Datapath support for variables: Network data analyzer example.

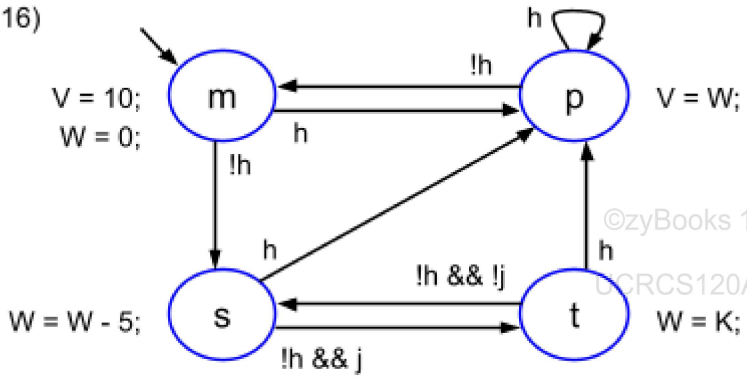
Animation captions:

1. Each data output and variable requires a register.
2. Instantiate components to support transition conditions involving data.
3. Instantiate components to support state actions involving data.
4. Datapath supports data output, variables, and data operations.

PARTICIPATION ACTIVITY 5.6.7: Datapath support for variables.

Given the following HLSM:

Inputs: h, j, K(16)
Outputs: V(16)
Variables: W(16)



1) A load register should be instantiated for variable W.
   - True
   - False

2) A subtractor should be instantiated to support W - 5.
   - True
   - False

3) A comparator should be instantiated to support !h && !j.
   - True
   - False

4) The register for variable W will require a single 16-bit 2x1 mux.
   - True
   - False

PARTICIPATION ACTIVITY 5.6.8: Datapath simulator.

- Change a's value to 1 by clicking on input a. Observe how the output changes.
- Change the value of the data inputs A and B, by clicking on A and B. Observe how the output changes.
- Modify the datapath by adding a 2x1 mux with two new data inputs, C and D. Add an input, b, for the select line. Delete the constant, 1, that connects to the adder's B input by selecting the constant, and clicking the Delete selected item(s) button. Connect the

output of the new 2x1 mux to the adder's B input. Change the values of inputs a, b, C, D and observe how the output changes.

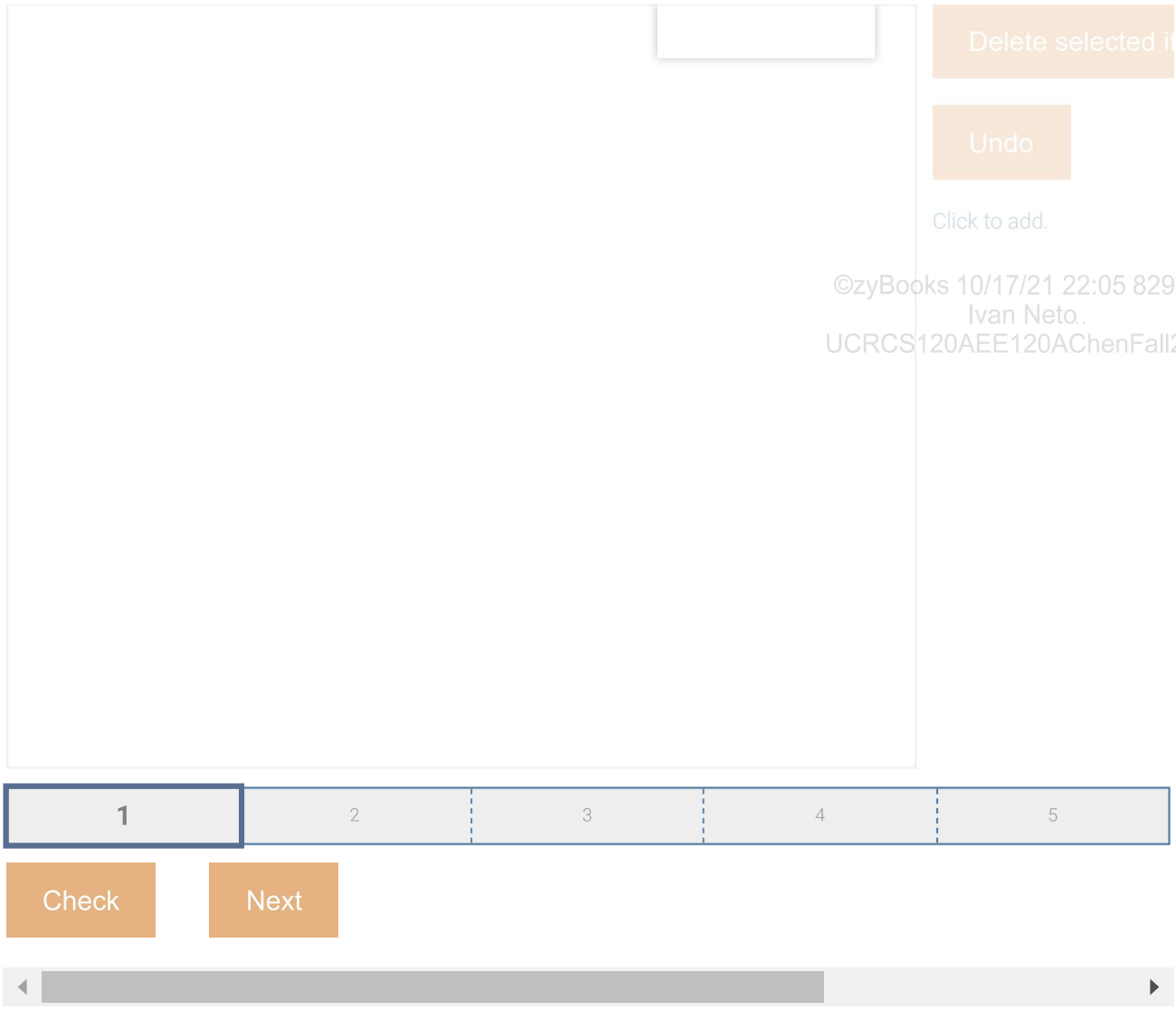
Reset

Export ↓ Import ↑ Example datapath with mux and adder

CHALLENGE ACTIVITY 5.6.1: Datapath building.

347136.1658324.qx3zqy7

Start



5.7 Example: Soda dispenser

Soda dispenser HLSM

A controller for a soda machine dispenser that dispenses a soda when enough money has been inserted can be captured by an HLSM.

PARTICIPATION ACTIVITY 5.7.1: Soda dispenser HLSM.

Animation content:

undefined

Animation captions:

1. c detects a coin, and A indicates the coin's value. S indicates the cost of a soda. change indicates if change is dispensed. Possible values for A are: 25 cents (00011001) or 10 cents (00001010).
2. A variable tot stores the current total of coins inserted. The HLSM initially sets output d to 0 and sets variable tot to 0. The HLSM then waits in state Wait to detect a coin being inserted.
3. When a coin is detected, the HLSM transitions to state Add, which adds the coin's value A to tot, after which the HLSM returns to Wait.
4. If tot < S, the HLSM continues to wait for more coins. Otherwise, the HLSM transitions to state DispSoda, which sets the output d to 1 to dispense a soda.
5. If exact change was not inserted, !(tot == S), the HLSM transitions to the DispChange state. The DispChange state sets change = 1 and returns to state Init to clear tot start over again.

PARTICIPATION ACTIVITY 5.7.2: Soda dispenser example.

1) If the HLSM is in state DispSoda, tot is 100, and S is 100, what is the next state?
   - DispChange
   - Init
   - Wait

2) The HLSM can be extended to output the amount of change to return using an 8-bit output R. Which statement should be added to the DispChange state to assign R with the change amount?
   - R = tot
   - R = S - tot
   - R = tot - S

3) In DispSoda, if tot == S, for how many clock cycles is d set to 1?
   - 1 clock cycle
   - 2 clock cycles
   - Unknown

4) Why is a HLSM chosen for the above example instead of an FSM?

- The example has multiple inputs
- The example has multi-bit arithmetic operations, including addition and comparisons.
- The example has implicit transitions.

Datapath for soda dispenser HLSM

A datapath for the soda dispenser HLSM can be created using a load register, a comparator, and an adder.

PARTICIPATION ACTIVITY 5.7.3: Soda dispenser datapath.

Animation content:

undefined

Animation captions:

1. The datapath for the soda dispenser HLSM supports any operation involving data, so has inputs S and A.
2. A register, tot_reg, is instantiated for the variable tot.
3. tot is cleared in state Init and assigned with a value in state Add. So, tot_reg has control inputs clr and ld, which are connected to the tot_clr and tot_ld inputs, respectively.
4. To support the Add state's action tot = tot + A, an adder is instantiated to add tot_reg's output Q and the input A. The adder's S output connects to the tot_reg's input D.
5. A magnitude comparator is instantiated for the comparisons tot < S and tot == S. The comparator's lt and eq outputs connect to outputs tot_lt_s and tot_eq_s, respectively.

PARTICIPATION ACTIVITY 5.7.4: Soda dispenser datapath.

Consider the following modifications to the above example.

1) Which outputs of the magnitude comparator are used?
   - only lt
   - gt
   - lt and eq

2) Which datapath output is needed to support the comparison tot < S?
   - tot_ld
   - tot_lt_s
   - tot_lt_eq

3) In which state is the tot_ld input needed to support the state's actions?
   - Init
   - DispSoda
   - Add

4) In state DispChange, instead of detecting change = 1, if the HLSM is modified to compute the change value, which components need to be added to the datapath?
   Data operation in state DispChange: CVal = tot - S
   - Only a subtractor
   - A subtractor and a register
   - A subtractor, a register, and a comparator

5.8 HLSMs to circuits: RTL design

An HLSM can be implemented as a circuit called a custom processor. A ***custom processor*** is a circuit consisting of a controller and datapath. After capturing behavior as an HLSM, a designer builds a datapath that supports the HLSM's variables and data operations, derives an FSM with the same states as the HLSM but using datapath control signals, and designs a controller for the FSM.

The process of capturing behavior as an HLSM and converting to a circuit is known as ***register-transfer-level design***, due to the HLSM essentially indicating what data transfers occur among registers during each clock cycle. ***RTL design*** is short for register-transfer-level design.

PARTICIPATION ACTIVITY 5.8.1: HLSMs to circuits: RTL design.

Animation captions:

1. Create the controller/datapath architecture with inputs/outputs. Instantiate a register for each data output and variable.
2. Support data operations.
3. Instantiate a mux. T_reg now has two possible input sources: 0 and N.
4. Instantiate adders and muxes as needed.
5. Create controller FSM: Same HLSM structure, control inputs/outputs.
6. Include bit operations from HLSM.
7. Replace data operations by datapath control values.
8. Continue for each data operation.
9. Create table for logic.
10. Derive equations for controller outputs and draw circuit.

PARTICIPATION ACTIVITY 5.8.2: HLSMs to circuits.

1) An HLSM is converted into a custom processor consisting of a controller and datapath.
   - True
   - False

2) Given the following HLSM declaration, only one register is needed in the datapath.
   Inputs: a, b
   Outputs: x, Y(8), Z(8)
   Variables: M(8)
   - True
   - False

3) The controller's FSM contains the same states and transitions as the HLSM.
   - True
   - False

4) Bit operations from the HLSM, like y = 0, are copied to the FSM.
   - True
   - False

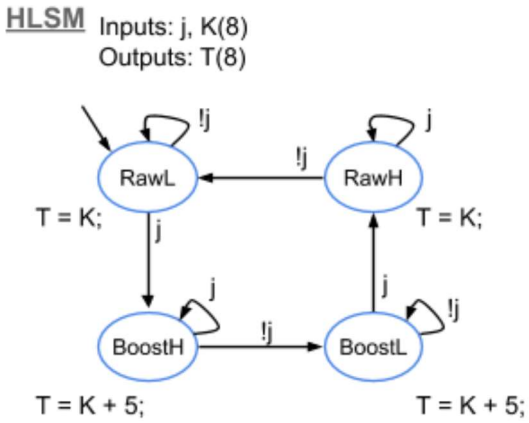
5) The FSM is converted into a controller

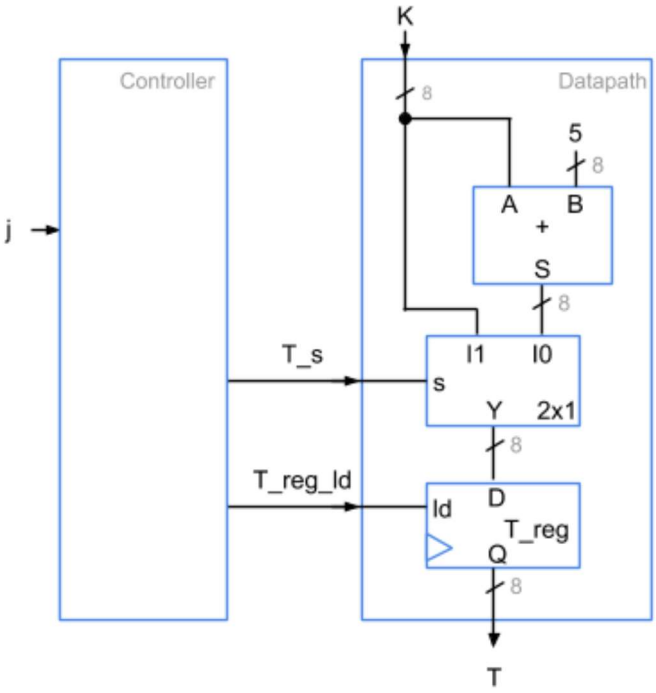
consisting of a state register and logic block.

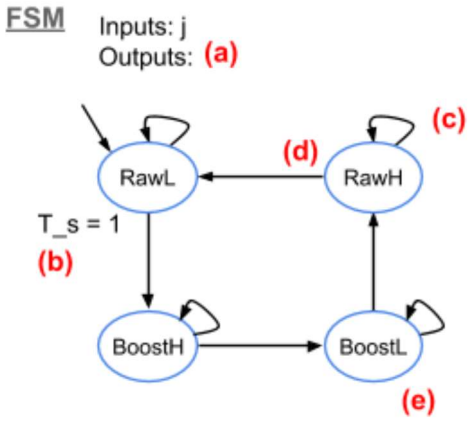
- True
- False

PARTICIPATION ACTIVITY 5.8.3: HLSM to FSM.

Complete the HLSM to FSM conversion.







1) (a) FSM outputs

Check Show answer

2) (b) State RawL: second action

Check Show answer

3) (c) RawH to RawH transition condition

Check Show answer

4) (d) RawH to RawL transition condition

Check Show answer

5) (e) BoostL state actions

Check Show answer

A datapath also supports data operations on transition conditions, such as a data comparison. The FSM's transition condition replaces the data comparison with a comparator output that is input to the controller.

PARTICIPATION ACTIVITY 5.8.4: HLSM transitions to FSM transitions.

Animation content:

undefined

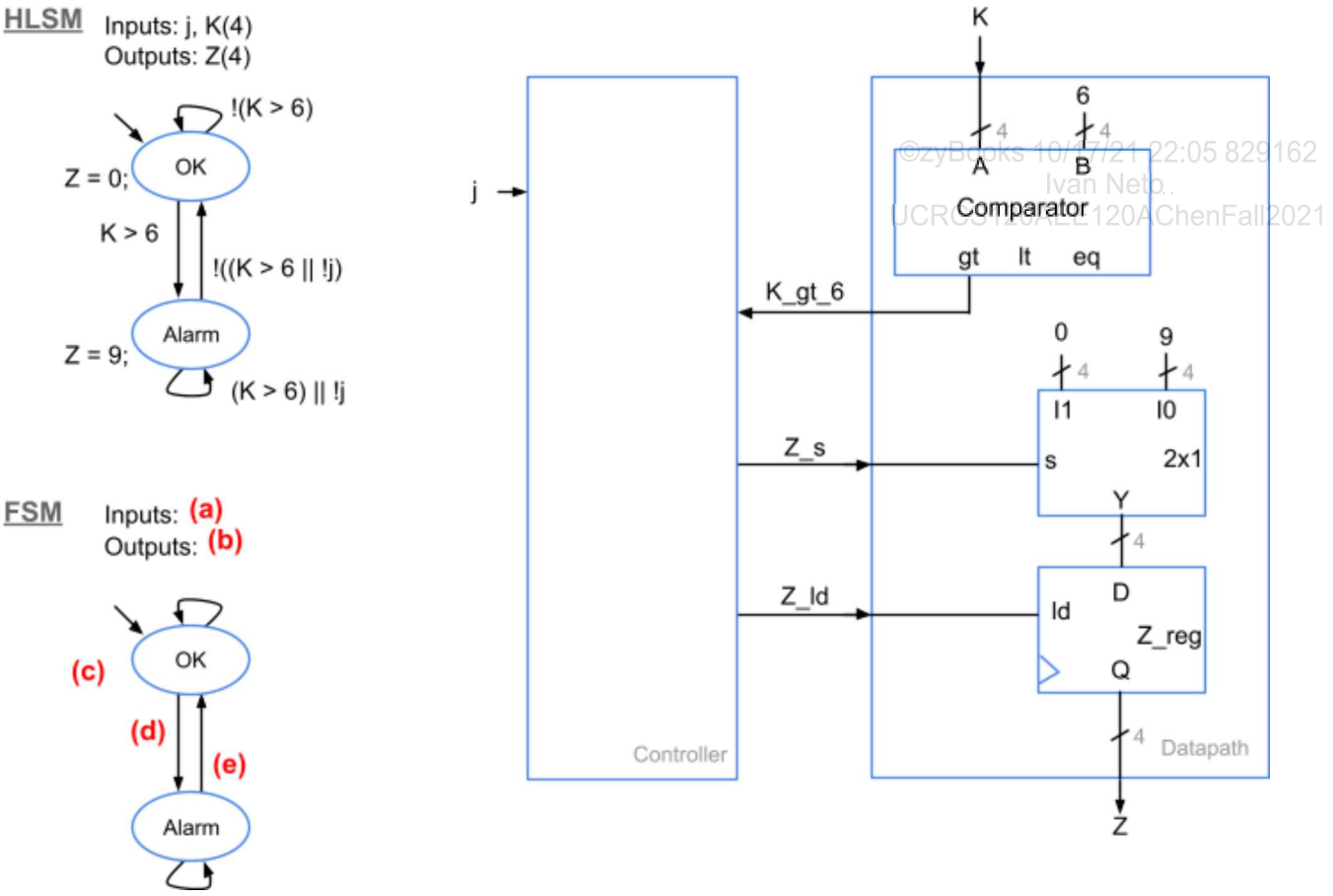
Animation captions:

1. Create the controller/datapath architecture with inputs/outputs.
2. Begin datapath construction: Create a register for each data output and variable.
3. Support data operations.
4. Data operations include transition conditions.
5. Create controller FSM: Same HLSM structure, control inputs/outputs.
6. Include bit operations from HLSM. Replace data operations by datapath control values.
7. Create table for logic.
8. Derive equations for controller outputs and draw circuit.

PARTICIPATION

ACTIVITY 5.8.5: HLSM transitions to FSM transitions.

Complete the HLSM to FSM conversion.



1) (a) FSM inputs

- j, K(4)
- j, K_gt_6
- j

2) (b) FSM outputs

- Z_s, Z_ld
- (none)
- j

3) (c) OK state action

- Z_s = 0;
  Z_ld = 1;
- Z_s = 1
  Z_ld = 1
- K_gt_6 = 1

4) (d) OK to Alarm transition condition

- K > 6
- K_gt_6
- !K_gt_6

5) (e) Alarm to OK transition condition

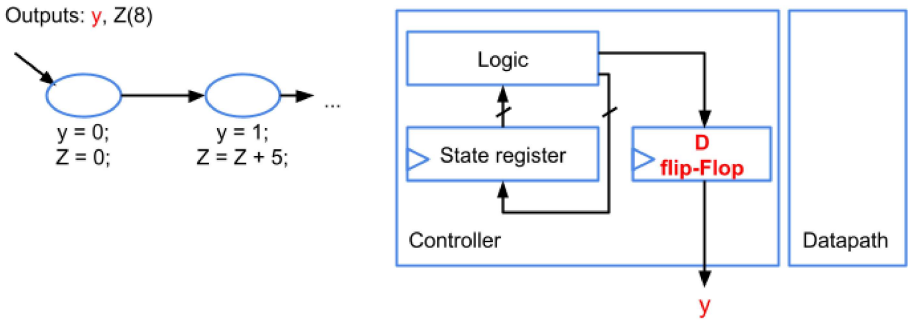
- !((K > 6)) || !j)
- !((K_gt_6)) || !j)
- (K_gt_6 + j')'

A ***registered output*** is an output immediately preceded by a register. Benefits include:

- A registered output prevents glitches from appearing on outputs. A ***glitch*** is an undesired temporary 1 (or 0) on a signal, commonly caused by non-ideal delays in a circuit.
- A registered output shortens the longest register-to-register path in a circuit, which has several implications not discussed here.

Thus, in addition to registering every datapath output, a custom processor should register every control output too, achieved by placing a D flip-flop at each control output. Such registering also ensures all custom processor outputs have the same timing, which is discussed in a later section.

Figure 5.8.1: Registered control outputs.



PARTICIPATION ACTIVITY 5.8.6: Registered outputs.

1) A registered output has a register followed by gates that lead to the output.

- True
- False

2) Only datapath outputs should be registered.

- True
- False

3) Control signals going from the controller to the datapath should be registered.

- True
- False

4) Registering a control output can be achieved simply by introducing a single D flip-flop.

- True
- False

5.9 Example: Laser-based distance measurer

Laser-based distance measurer overview

Many applications require accurately measuring the distance of an object from a known point. Ex: Road builders need to accurately determine the length of a stretch of road. In such applications, stringing out a tape measure to measure the distance is not very practical. A better method involves laser-based distance measurement.

PARTICIPATION ACTIVITY 5.9.1: Laser based distance measurer.

Animation content:

`undefined`

Animation captions:

1. A laser is pointed at the object of interest. The laser is briefly turned on and a timer is started. The laser light, traveling at the speed of light, travels to the object and reflects back.
2. A sensor detects the reflection of the laser light, causing the timer to stop. T is the time to travel to the object and back, the distance D is computed using the speed of light.

3. The time T can be measured by counting the number of clock cycles and multiplying with the clk period. Assuming the clock frequency is 300 MHz, or $3x10^8$ Hz, D can be calculated as D = cycles counted / 2.
4. Input b = 1 when the user presses a button to start the measurement. Output l = 1 turns the laser on. Input s = 1 when the reflected laser is detected. N-bit output D indicates the measured distance.

PARTICIPATION ACTIVITY 5.9.2: Laser based distance measurer.

1) If D is 8-bits, can the laser distance measurer a distance of 2000 meters?
   - Yes
   - No

2) Is the distance to the object is 950 meters, how many clock cycles will be counted?
   - 950
   - 1900

3) What is the shortest distance in meters that can be measured?
   - 0.5
   - 1

Laser-based distance measurer HLSM and datapath

The laser-based distance measurer can be implemented using a HLSM and then converted to a datapath and controller.

PARTICIPATION ACTIVITY 5.9.3: HLSM and datapath.

Animation content:

undefined

Animation captions:

1. S0 is the initial state. After initialization, the system waits for the user to press the button to initiate the measurement process.

2. When b = 1, the laser is turned on for one cycle. State S2 sets output l to 1 (laser on) for one cycle. The HLSM then transitions to S3 and sets l to 0 (laser off).
3. The HLSM remains in S3 while s is 0 (laser not sensed) and increments Dctr by 1 to count the clock cycles between the laser being turned on and the laser's reflection being sensed.
4. When s = 1, the HLSM computes the distance D measured. For a clock frequency of 300 MHz, each cycle represents one meter travelled. So, the output D is assigned Dctr / 2.
5. The data output D and variable Dctr both require a register in the datapath.
6. To support Dctr = Dctr + 1 in S3, an adder is instantiated in the datapath.
7. To support D = Dctr / 2 in S4, a 16-bit right-shift-by-1 component is instantiated between Dctr and Dreg. A right shift by one bit is equivalent to a divide by 2.

PARTICIPATION ACTIVITY 5.9.4: HLSM and datapath.

1) How many clock cycles is the laser turned on for?
   - 1 clock cycle
   - As long as the button is pressed
   - Until the laser is detected

2) Why is a shift-right-by-1 component instantiated in the datapath for the laser-based distance measurer?
   - To perform division by 2
   - To count the number of clock cycles until the sensor detects the laser
   - To perform multiplication by 2

3) What happens in S4 after Dctr / 2 is assigned to D?
   - The HLSM stops executing.
   - The HLSM waits in S4 until s = 0.
   - The HLSM transitions back to S1 and waits for a button press.

Controller FSM for laser-based distance measurer datapath

A controller for the laser-based distance measurer is implemented as an FSM with the same states as the HLSM but using the datapath control signals.

PARTICIPATION ACTIVITY 5.9.5: HLSM to FSM conversion.

Animation content:

undefined

Animation captions:

1. The controller FSM has the same states and transition structure as the HLSM structure. The FSM inputs and outputs include the datapath's control inputs/outputs.
2. The bit operations from the HLSM are included in the FSM.
3. The data operations are replaced with the datapath control values. The FSM controller can then be implementing using a 3-bit state register and the combinational logic to describe the next state and the output logic.

PARTICIPATION ACTIVITY 5.9.6: Controller FSM for laser-based distance measurer datapath.

1) Dctr_ld = 0 in S4 clears the value in Dctr_reg.
   - ○ True
   - ○ False

2) D_reg_ld = 1 in S4 loads the final calculated distance into D_reg.
   - ○ True
   - ○ False

5.10 RTL timing

Digital circuits usually cannot ideally-achieve designer-specified behavior. For example, logic gates and wires have delays, so a Boolean function implemented as a circuit experiences a non-ideal delay between input and output value changes. Likewise, an FSM implemented as a circuit experiences a non-ideal delay between rising clock edges and the next state's output value changes.

HLSMs implemented as a circuit experience an even greater non-ideal delay. In particular, the register instantiated for a datapath output introduces a one-clock-cycle delay.

PARTICIPATION 5.10.1: HLSM outputs implemented as a circuit experience a one-clock-cycle

ACTIVITY delay.

Animation captions:

1. T is expected to immediately output 1, 2, 3 ...
2. T_reg introduces a one-clock-cycle delay.
3. States assign the datapath control values to assign T a new value. Register does not load until a rising clock edge.
4. Rising clock edge. 1 is loaded into T_reg.
5. In the same clock cycle, state n assigns the datapath control values to load 2 into T_reg.
6. Rising clock edge. 2 is loaded into T_reg.
7. T(actual) is delayed by 1 clock cycle.

PARTICIPATION ACTIVITY 5.10.2: One-clock-cycle delay in RTL design.

Consider an HLSM (see above) that simply sequences among states m (T = 1), n (T = 2), and p (T = 3), followed by m again. Assume T_reg implements variable T.

1) What is the ideal value of T_reg when in state n?
   - 1
   - 2
   - 3

2) What is the actual value of T_reg when in state n?
   - 1
   - 2
   - 3

3) In what state will T_reg contain 2?
   - m
   - n
   - p

5.11 Assigning and reading variables

In an HLSM, a state action assignment to a variable ideally updates the variable's value immediately upon entering a state. However, when implemented as a circuit using a register, each register's update occurs non-ideally on the *next* rising clock edge, thus experiencing a one-clock-cycle delay. Circuits are non-ideal and thus usually cannot ideally achieve designer-specified behavior.

A variable register's one-clock-cycle delay becomes an issue if an HLSM state assigns a variable a next value *and then reads that variable in the same state.* When designing the datapath, the read should be from that next value, and not from the variable register's output, since that register's update has a one-clock-cycle delay.

PARTICIPATION ACTIVITY 5.11.1: Datapath support for assigning and then reading a variable.

Animation captions:

1. State s assigns I = 1. 1 is loaded into I_reg after one cycle.
2. Z is expected to be assigned with the next value of I.
3. If Z_reg is connected to I_reg's output, Z_reg will be loaded with the present, not the next, value of I.
4. If Z_reg is connected to the incrementer's output, Z_reg will be loaded with the next value of I.

PARTICIPATION ACTIVITY 5.11.2: Datapath support for assigning and then reading a variable.

Consider the above example of assigning and then reading a variable.

1) What value is in Z_reg in the middle of state s?
   - ○ 0
   - ○ 1

2) Just after entering state t for the first time, what value is in I_reg?
   - ○ 0
   - ○ 1

3) Just after entering state t for the first time, what should variable Z be?
   - ○ 1
   - ○ 2

4) Consider nearing the end of state t's first execution. If Z_reg is connected to

the output of I_reg, what value is waiting at the input of Z_reg?

- 1
- 2

5) Consider nearing the end of state t's first execution. If Z_reg is connected to the output of the incrementer, what value is waiting at the input of Z_reg?

- 1
- 2

The following provides a simple example of designing a datapath that uses a register's next value rather than the register's output. The example counts the number of clock cycles that an input pulse lasts, outputting that count.

PARTICIPATION ACTIVITY 5.11.3: Using the next value of a variable: A pulse-duration counter.

Animation captions:

1. Z is expected to be assigned with the next value of I.
2. If Z_reg is connected to I_reg's output, Z_reg will be loaded with the present, not the next, value of I.
3. If Z_reg is connected to the incrementer's output, Z_reg will be loaded with the next value of I.

PARTICIPATION ACTIVITY 5.11.4: Datapath support for assigning and then comparing a variable.

Animation content:

undefined

Animation captions:

1. V > 99 is expected to compare to the next value of V.
2. If comparator input is connected to V_reg's output, comparator compares to the present, not the next, value of V.
3. If comparator input is connected to adder's output, comparator correctly compares to the next value of V.

Note that connecting to a register's next value is only necessary when a given state has an assignment that is then read in the same state, either by a subsequent action, or by a transition condition. Otherwise, connecting to a register's output is appropriate.

PARTICIPATION ACTIVITY 5.11.5: Datapath support for assigning and then reading a variable in a transition condition.

1) J and L are 8-bit variables. State x has one action: J = 9. The state has an outgoing transition with condition J == L. With what should a datapath's comparator connect to read J?
   - ○ J_reg's output
   - ○ L_reg's output
   - ○ 9

2) J, K, and L are 8-bit variables. State x has one action: J = K. The state has an outgoing transition with condition J == L. With what should a datapath's comparator connect to read J?
   - ○ J_reg's output
   - ○ K_reg's output
   - ○ K_reg's input

3) J, K, and L are 8-bit variables. State x has one action: J = K. The state has an outgoing transition with condition K == L. With what should a datapath's comparator connect to read K?
   - ○ K_reg's output
   - ○ J_reg's output
   - ○ No connection will work

©zyBooks 10/17/21 22:05 829162
Ivan Neto
UCRCS120AEE120AChenFall2021

5.12 Product Profile: Digital video

Digital video overview

Digitized video works by sampling an analog video signal and transforming the samples to digital values. Digitized video can be transmitted, stored, replayed, and copied with much higher-quality than analog video. A digital video consists of a sequence of pictures, where each picture is known as a frame. Each frame in a digital video may contain millions of pixels, and each pixel is stored as several bytes. For practical use, a video needs to be compressed into much smaller files.

Video compression ratios range from 30:1 to 100:1 or more depending on the video compression standard used. Popular video compression standards include MPEG-4 and H.264. The compression ratio is determined by dividing the number of bits of the digitized video before compression, by the number of bits after compression. If a digitized video requires 400 gigabytes uncompressed but only 4 gigabytes compressed, the compression ratio would be 400/4 = 100:1. Packing 1500 Gbits of a movie into 37.6 Gbits would require a compression ratio of 1500 Gbits/37.6 Gbits = 40:1.

Videos typically have much interframe redundancy. Most video compression standards do not merely encode each frame as a distinct picture, but encodes each frame as one of the following:

- An I-frame, or intracoded frame, is a complete picture.
- A P-frame, or predicted frame describes the difference between the current frame and the previous frame. To derive the picture for this frame, one must combine the P-frame with the previous frame.
- A B-frame, or bidirectional predicted frame, stores differences from previous and future frames.

PARTICIPATION ACTIVITY 5.12.1: Digitized video.

Animation content:

undefined

Animation captions:

1. A video is a series of quickly displayed still pictures, known as frames. One second of video consists of about 30 frames that appear as a smooth, continuous video.
2. Each frame may consist of many pixels, often millions, where each pixel is encoded using red, blue, and green values (rgb).
3. Digitized video can be compressed by only encoding differences between a base frame and the following frames.
4. In MPEG-4 video compression, a frame can be encoded as an intracoded frame (I-frame), a predicted frame (P-frame), or a bidirectional predicted frame (B-frame). P-frames require fewer bits than I-frames.

PARTICIPATION ACTIVITY 5.12.2: Digital video and MPEG-2.

Consider a video with 12 frames. Frame sizes are 8 Mbits for I-frames, and 2 Mbits for P-frames. A video compression method uses the following sequence of frames to represent the video: I, P, P, P, P, I, P, P, P, P, P, P.

1) What is the number of bits before video compression?
   - 16 Mbits
   - 24 Mbits
   - 96 Mbits

2) What is the number of bits after the video compression?
   - 16 Mbits
   - 20 Mbits
   - 36 Mbits

3) What is the compression ratio?
   - 100:1
   - 2.6:1
   - 12:8

Video compression basics

A common video compression technique uses a base frame followed by data describing just the difference between the base frame and the next frame. Each frame is divided into blocks of fixed sizes, such 16x16 pixels. To compress each block in the next frame, the most similar block in the first frame is found and only the difference between these two blocks is encoded in the video file. Thus, video compression needs to quickly estimate the similarity between two blocks. A common method to determine the similarity is to compute the sum of absolute differences (SAD) between the blocks' pixel values.

PARTICIPATION ACTIVITY 5.12.3: Sum of absolute differences.

Animation content:

undefined

Animation captions:

1. A frame can be encoded as the difference between the frame's blocks and the most similar blocks in a previous frame. For block B in frame 2, the encoding searches for the most similar

block in frame 1.

2. Only the difference between blocks B and A need to be stored for frame 2. While Frame 1 requires 1 Mbyte to encoded the entire frame, encoding frame 2 only requires 0.01, resulting in a smaller file.

PARTICIPATION ACTIVITY 5.12.4: Sum of absolute differences.

Consider the following two 2x2 blocks.
Block 1:

	0	1
0	143	125
1	145	147

Block 2:

	0	1
0	123	125
1	140	147

1) What is the absolute difference between the pixels at row 0 column 0?
   - 0
   - -20
   - 20

2) What is the sum of absolute differences between all pixels of the blocks?
   - 5
   - 20
   - 25

3) A video with high similarity among frames will achieve a ____ compression than a video with many differences.
   - lower

- higher

Sum of absolute differences HLSM

Computing the sum of absolute differences can be performed using a custom digital circuit designed using an HLSM. While modern video compression methods support encoding different sized block sizes, the following HLSM computes the similarity between corresponding 16x16 pixel blocks.

PARTICIPATION ACTIVITY 5.12.5: Sum of absolute differences HLSM.

Animation content:

undefined

Animation captions:

1. The SAD circuit's inputs are two 256-byte register files A and B, that hold the contents of two corresponding 16x16 block of pixels of frame 1 and frame 2. go = 1 begins computing. The sad output holds the sum of absolute differences data.
2. The HLSM initially waits for the input go to become 1. The HLSM then initializes registers sum and I to 0. Next, if I is less than 256, the HLSM computes the absolute value of the difference of the two blocks' pixels indexed by I.
3. Next, if I < 256, the HLSM computes the absolute value of the difference of the blocks' pixels indexed by I, adds that result to sum, increments I, and repeats. If !(I < 256), sadreg is assigned with sum, which is the final computed similarity.
4. The datapath for the HLSM has a 32-bit sum register that holds the running sum of differences, a 9-bit I register indexes into the current pixel with a range from 0 to 255. The 32-bit sad_reg registers the sad output.
5. AB_addr is used as the address for both register files A and B. The data outputs from register files A and B connect to the datapath's inputs A_data and B_data, respectively.
6. The subtractor computes A[i] - B[i], and the abs component computes the absolute value. The adder adds the absolute value with the sum, which then connects to the sum register's input. The comparator compares i with 256.

PARTICIPATION ACTIVITY 5.12.6: Sum of absolute differences HLSM and datapath.

1) How many cycles after go becomes 1 will the sad output be assigned with the computed sum of absolute differences?

- 4 cycles

- 258 cycles
- 514 cycles

2) The AB_addr output is connected to _____.

- only register file A's addr input
- only register file B's addr input
- both register file A's addr input and register file B's addr input

Comparing Microprocessor and Custom Circuit Implementations

If the SAD algorithm ran on a microprocessor, the algorithm would likely need more than two clock cycles per loop iteration. The software execution would need about two cycles to load internal registers, one cycle to subtract the values, two cycles to compute the absolute value, and one cycle to add the results, for a total of six cycles per iteration. The custom circuit built in the above example, at two cycles per iteration, is about three times faster for computing SAD, assuming equal clock frequencies.

5.13 Behavioral-level design: Programs to gates

Behavioral level design: If-else templates

Behavioral-level design is the process of specifying a system's behavior using a high-level programming language, like C, C++, or Java, and then converting to an RTL description. A key advantage of behavioral level design is that specifying the behavior in a high-level language is often faster. The overall behavioral-level design process to convert a high-level program to gates consists of several steps:

- Write a program to implement the system functionality in a high-level language.
- Convert the program to a HLSM. The conversion from a program to HLSM can use templates for each programming construct.
- Convert the HLSM to a circuit consisting of a controller and datapath.
- Convert the controller and datapath to gates.

The template for converting an assignment statement to an HLSM uses a separate state for each assignment. The template for converting an if-else statement to an HLSM uses one state to check the condition with transitions to the respective states for the if and else blocks, and a final state after the if or else blocks execute.

PARTICIPATION ACTIVITY 5.13.1: Assignment and if-else templates.

Animation content:

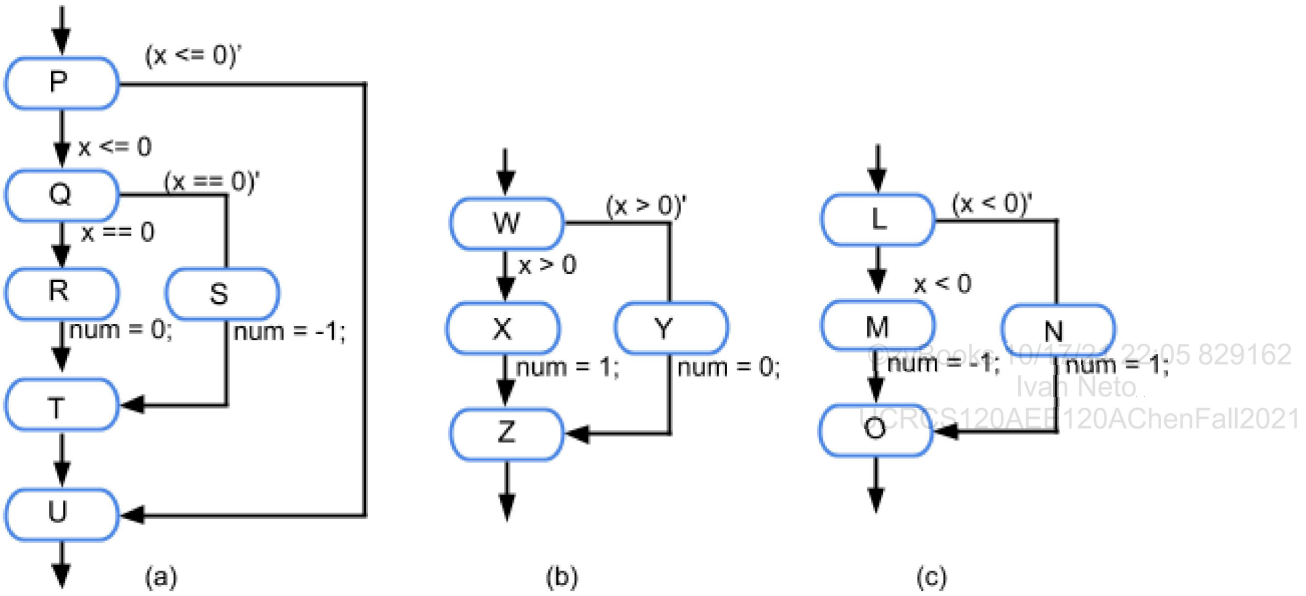
undefined

Animation captions:

1. An assignment statement in a program translates into one HLSM state. The state's actions implement the assignment.
2. An if-else statement uses one state to check the condition with two transitions: one transition for when the condition is true and one transition for when the condition is false.
3. The example shows a program snippet for finding the greater of two numbers. If x > y is true, then the HLSM transitions to the next state which implements max = x.
4. If x > y is false, the HLSM transitions to a state which executes max = y. A final state executes indicating the end of the if-else statement.

PARTICIPATION ACTIVITY 5.13.2: If-else templates.

Match each snippet of C code to the correct HSLM template.



```
if (x < 0) {
   num = -1;
}
else {
   num = 1;
}
```

```
if (x <= 0) {
   if (x == 0) {
      num = 0;
   }
   else {
      num = -1;
   }
}
```

```
if (x > 0) {
   num = 1;
}
else {
   num = 0;
}
```


(a)

(b)

(c)

Reset

While loop template

The template for converting a while loop to an HLSM uses a state to check the loop's condition and transitions to the state for loop's statements if the condition is true or transitions to a final state after the loop if false. The last state implementing the loop's statement will transition back to the state to check the loop condition.

PARTICIPATION ACTIVITY 5.13.3: While loop template.

Animation content:

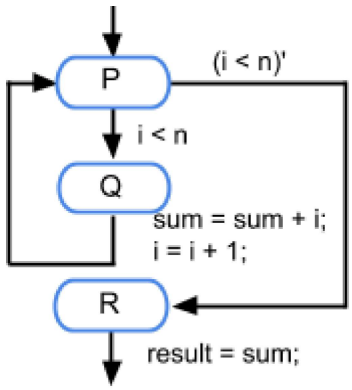
undefined

Animation captions:

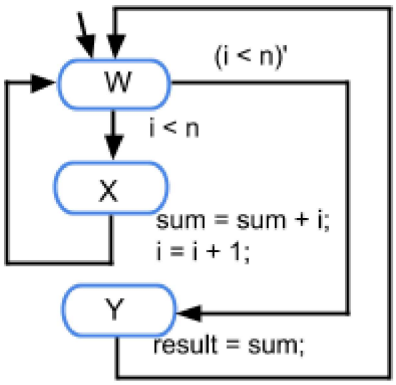
1. The HLSM for a while loop uses a state to check if the loop's condition is true. If true, the HLSM transitions to a state that executes the loop's statements. If false, the HLSM transitions to a final state.
2. The example shows a while loop to find the factorial of a number. The HLSM uses a state to check if i <= n. If true, the assignment statements are implemented using two assignment states.
3. The two assignment statements can be combined into one state.
4. When the condition is false, or (i <= n)' is true, the HLSM transitions to a final state. The assignment statement after the while loop, result = product, is implemented in that final state.
5. Using the template sometimes introduces extra states that can be removed, yielding a smaller HLSM.

PARTICIPATION ACTIVITY 5.13.4: While loop templates.

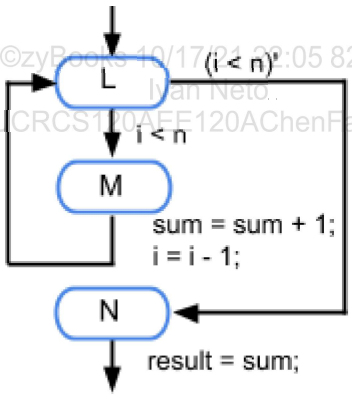
Match each snippet of C code to the correct HSLM template.



(a)



(b)



(c)

```
while (1) {
   while(i < n) {
      sum = sum + i;
      i = i + 1;
   }
   result = sum;
}
```

```
while(i < n) {
   sum = sum + i;
   i = i + 1;
}
result = sum;
```

```
while(i < n) {
   sum = sum + 1;
   i = i - 1;
}
result = sum;
```

(b)

(c)

(a)

Reset

SAD C code to high-level state machine conversion

A previous section introduced the concept of the sum of differences in the context of video compression. The high-level description of the sum of differences example is converted to an HLSM using the assignment, if-else and while loop templates.

PARTICIPATION ACTIVITY 5.13.5: SAD C code to high-level state machine conversion.

Animation content:

```
undefined
```

Animation captions:

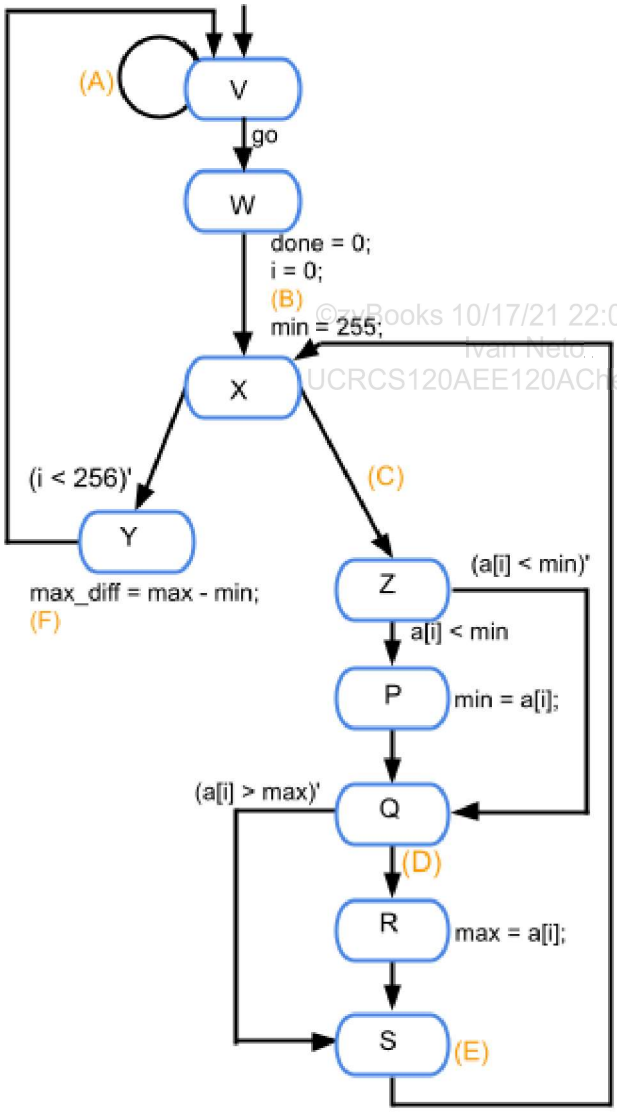
1. The SAD program waits for go = 1 to start computation. The while(!go) loop does nothing until go becomes 1. The HLSM uses a while loop template for the while(!go) loop.
2. The template can be reduced to remove the extra state so the HLSM will stay in state A when go = 0.
3. The transition (go')' is rewritten as go. The assignment statements sum = 0 and i = 0 are implemented in a single state B.
4. A while loop template is used for the inner while loop. If i < 256, the HLSM transitions to state D, which implements the while loop's statements that compute the sum and increment i.
5. When i < 256 is false, the HLSM transitions state E. The statement after the while loop is then implemented in state E.
6. Finally, the while(1) loop is implemented as a transition from the state E to the state A. The while(1) loop is used to constantly execute the computation.

PARTICIPATION ACTIVITY 5.13.6: Converting C code to a HLSM.

The following code snippet calculates the maximum difference between any two numbers within an array A consisting of 256 8-bit values. Complete the HLSM implementation of that code.

```
while(1) {
   while(!go) {
      // do nothing
   }
   done = 0;
   i = 0;
   max = 0;
   min = 255;

   while(i < 256) {
      if(a[i] < min){
         min = a[i];
      }
      if(a[i] > max){
         max = a[i];
      }
      i = i + 1;
   }
   max_diff = max - min;
   done = 1;
}
```



1) (A)

Check Show answer

2) (B)

Check Show answer

3) (C)

Check Show answer

4) (D)

Check Show answer

5) (E)

Check Show answer

6) (F)

Check Show answer

High-level synthesis

High-level synthesis (***HLS***) *is the automated process of converting a program from a high-level language to a digital circuit. HLS tools are increasing in usage by designers. HLS tools typically only support a subset of the programming language, with the subset increasing as HLS tools advanced.*

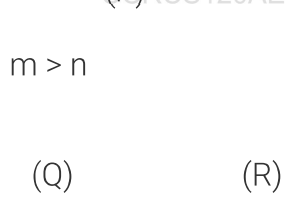
CHALLENGE ACTIVITY 5.13.1: Behavioral-level design: Programs to gates.

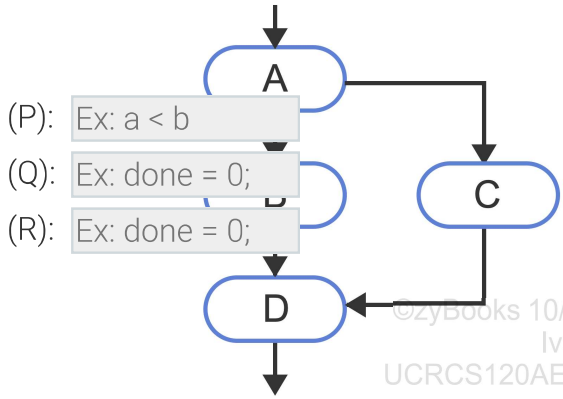
347136.1658324.qx3zqy7

Start

Given the following C code, complete the HLSM template by filling in the missing values (P), (and (R).


```
if (m > n) {
   answer = m;
}
else {
   answer = n;
}
```





1 2 3

Check Next