EECS151/251A Discussion 5

Christopher Yarp

Feb. 21, 2019

Plan for Today

- FSM Practice
- Ready/Valid Interfaces
- Transistor Level Logic Implementation
  - Logic with Mux & Pass Gates

FSM Practice

Designing a Door Controller for Cory 125

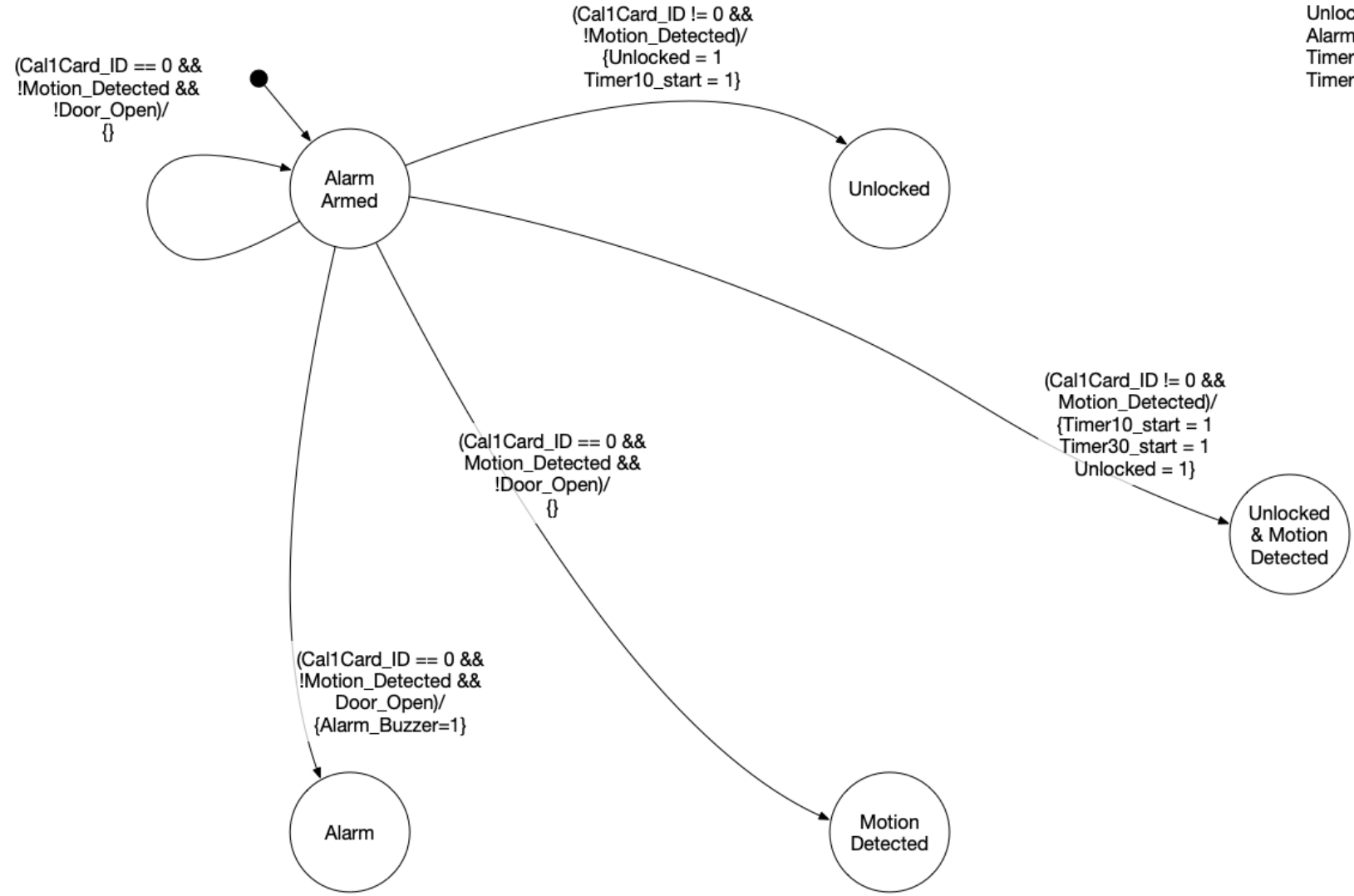
- The door is normally locked
- Any Cal1Card will unlock the door for 10 seconds.
  - The card reader will not respond to another card during that time.
- Cory 125 has a motion detector on the inside of the door. When motion is detected, the door may be opened from the inside for up to 30 seconds without sounding the alarm.
  - Any use of a Cal1Card during this time should unlock the door for the appropriate amount of time.
  - The alarm should not go off while the door is unlocked.
  - The alarm may still be disabled by the motion detector after the door is locked again.
  - Any activation of the motion detector while the alarm is disabled will delay the re-arming of the alarm until 30 seconds later.
- If the door is opened without the motion detector being triggered or a Cal1Card being used, the alarm sounds.
  - The door can be returned to normal with the lab manager's Cal1Card (ID = 0001).
  - All entry into Cory 125 is disabled when the alarm is sounding.
- Timers for 10 and 30 seconds are provided
  - Both have a start input which resets and starts the counter when asserted
  - Both timers have a "done" signal which will be brought high for 1 cycle when the timer expires

Designing a Door Controller for Cory 125

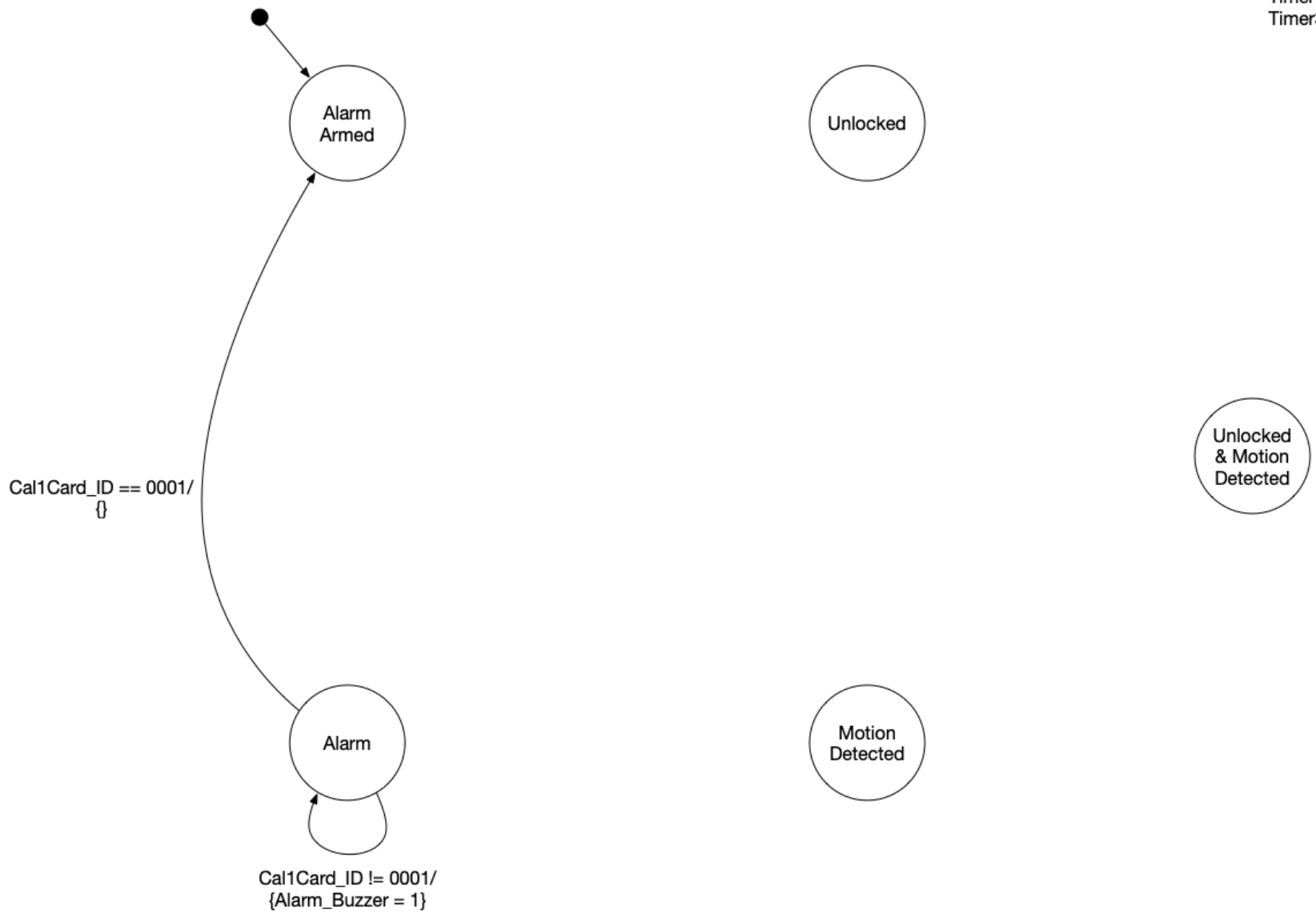
- Despite what seems like a simple job, the state machine for this problem can get complex
  - Mostly because of special cases we need to consider
- We'll make this manageable by looking at each state, one at a time

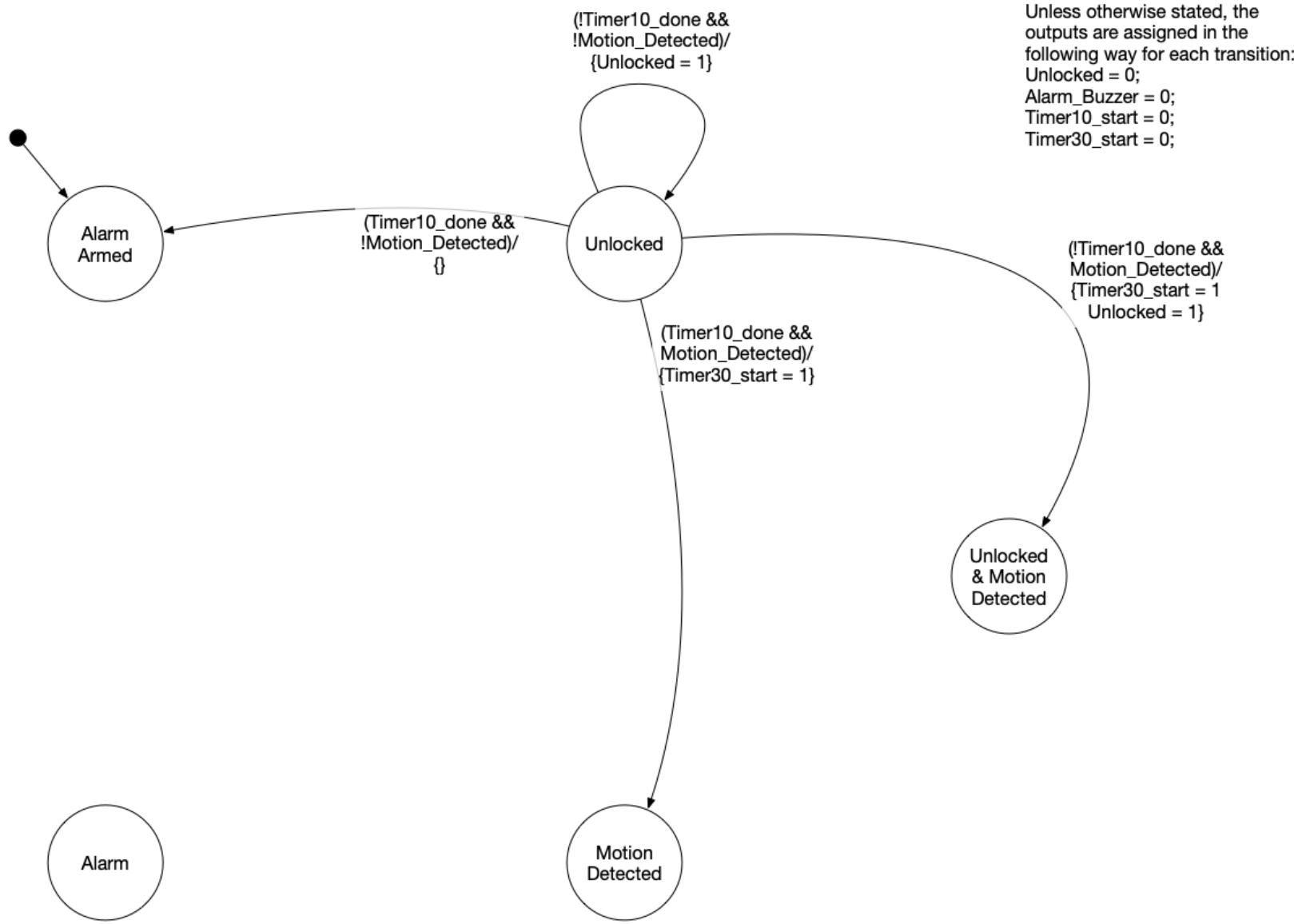
Unless otherwise stated, the outputs are assigned in the following way for each transition:
Unlocked = 0;
Alarm_Buzzer = 0;
Timer10_start = 0;
Timer30_start = 0;

Unless otherwise stated, the outputs are assigned in the following way for each transition:
Unlocked = 0;
Alarm_Buzzer = 0;
Timer10_start = 0;
Timer30_start = 0;

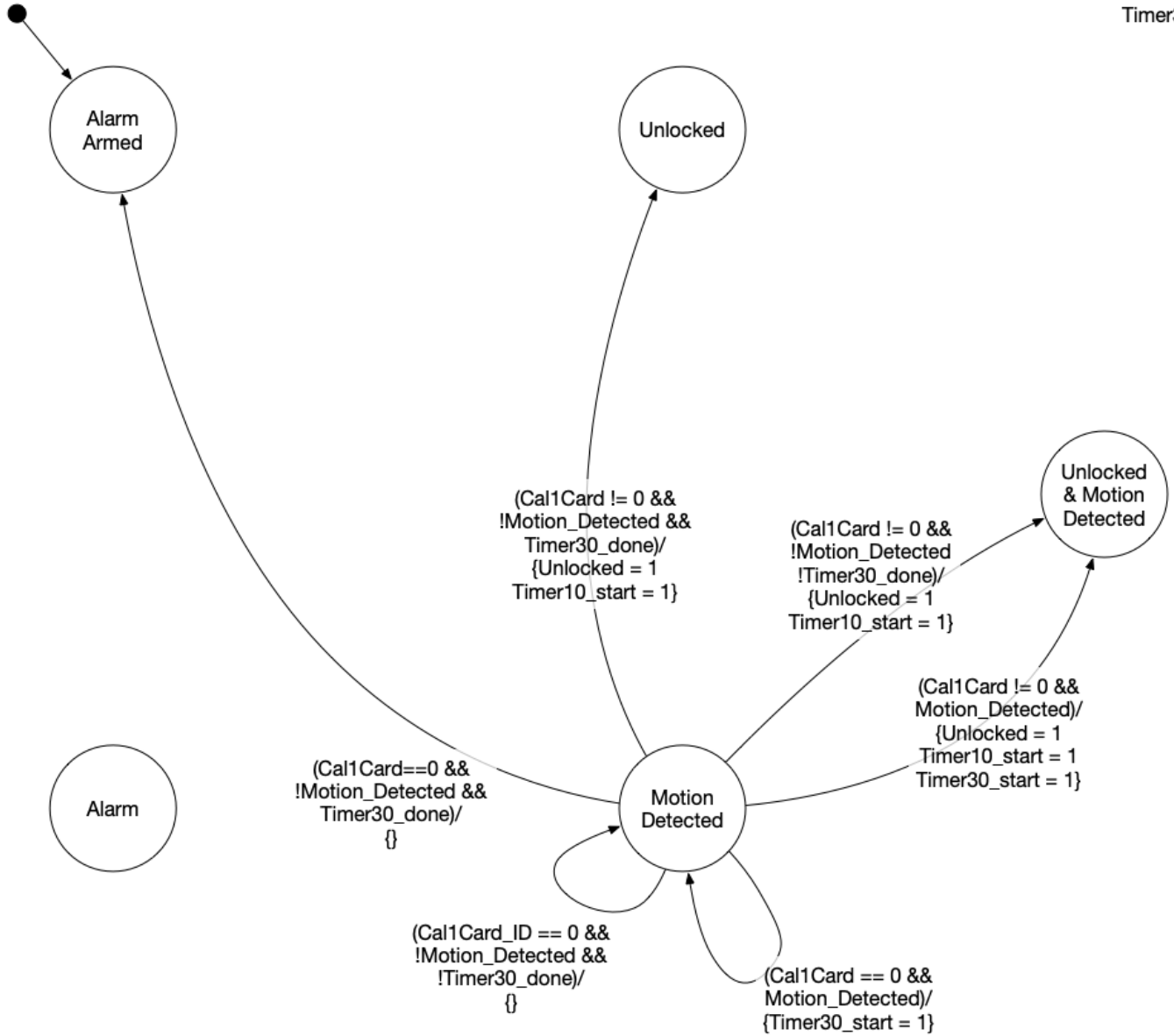


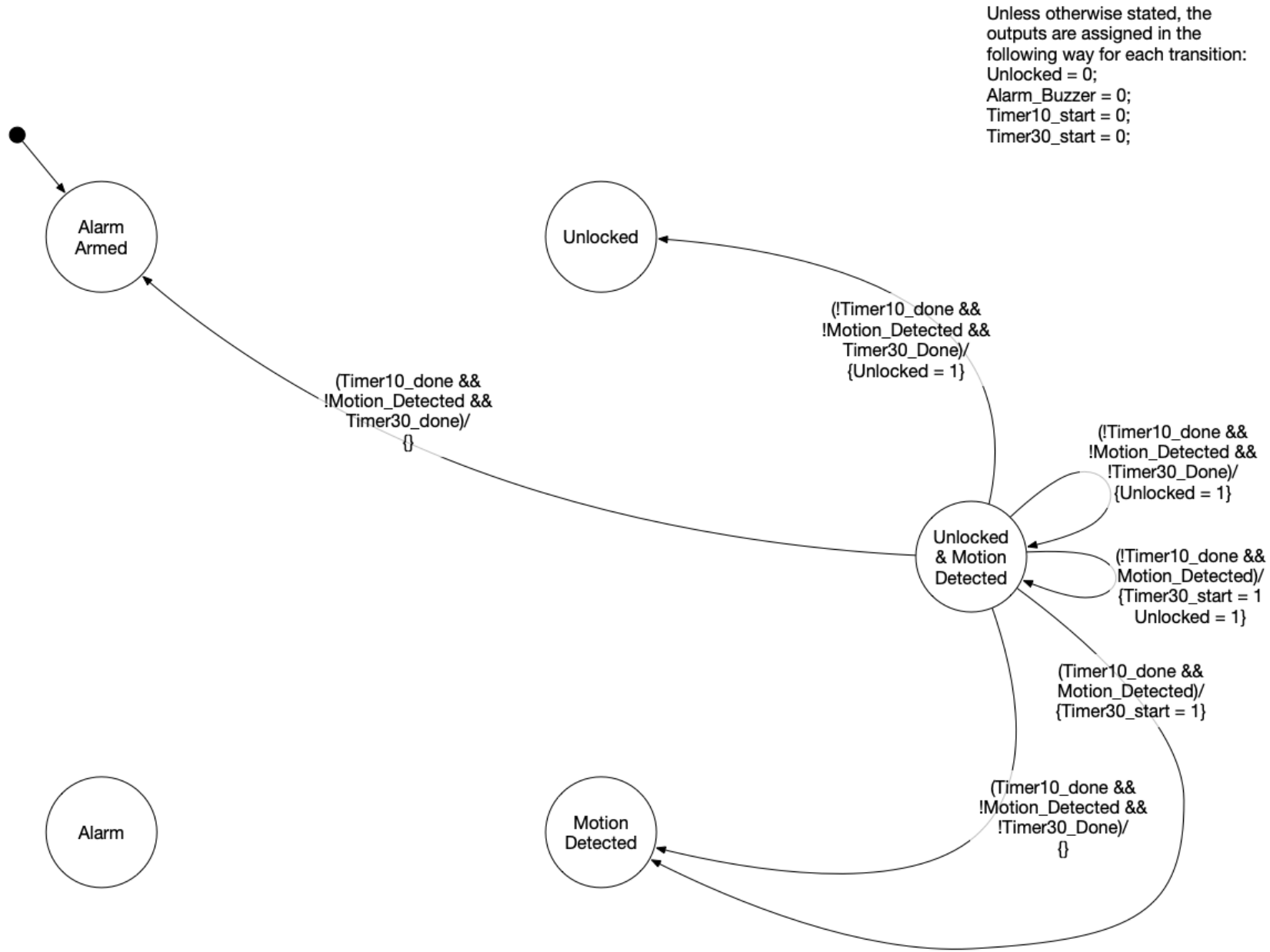
Unless otherwise stated, the outputs are assigned in the following way for each transition:
Unlocked = 0;
Alarm_Buzzer = 0;
Timer10_start = 0;
Timer30_start = 0;

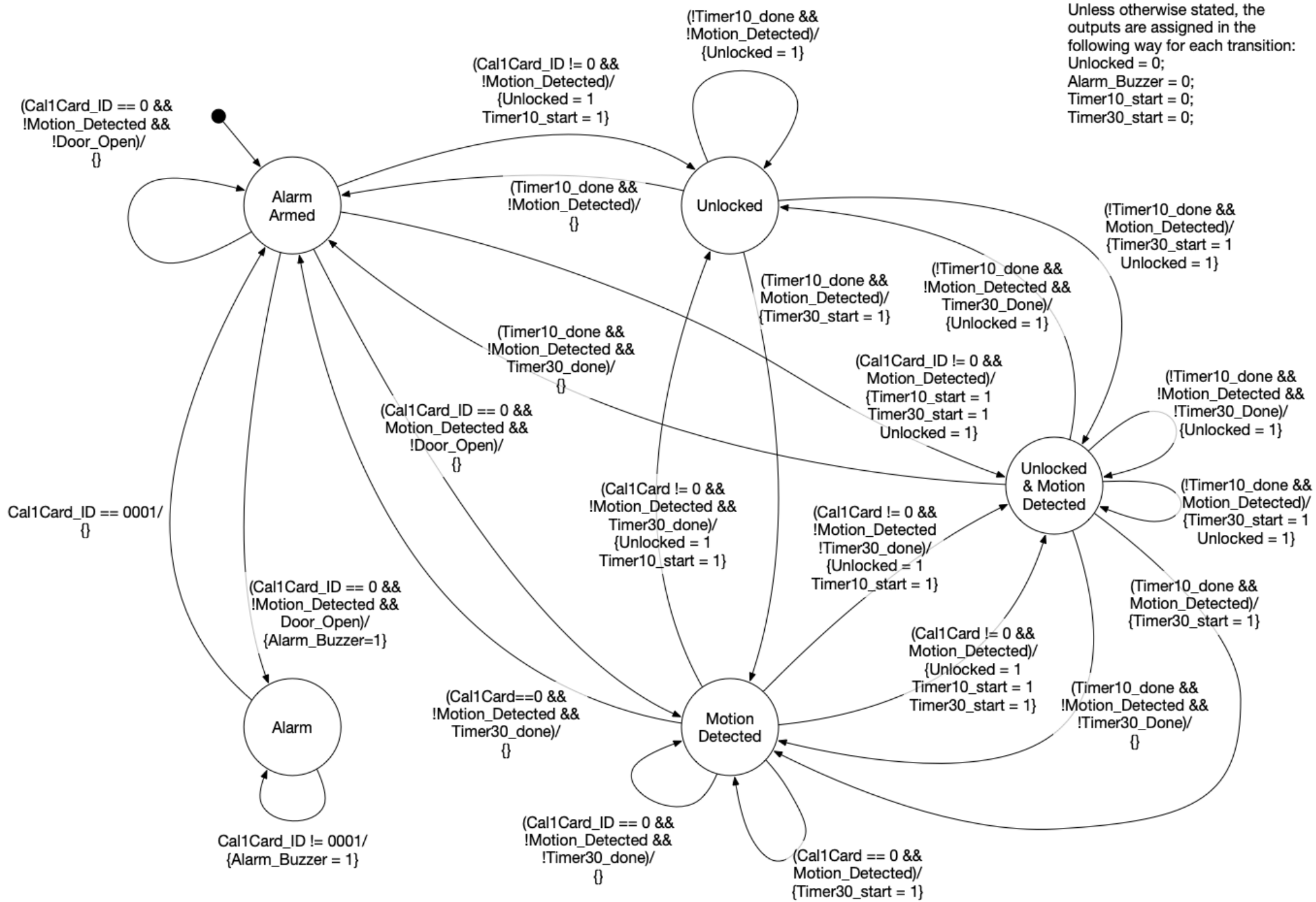



(!Timer10_done &&
!Motion_Detected)/
{Unlocked = 1}
Unless otherwise stated, the
outputs are assigned in the
following way for each transition:
Unlocked = 0;
Alarm_Buzzer = 0;
Timer10_start = 0;
Timer30_start = 0;
Alarm
Armed
(Timer10_done &&
!Motion_Detected)/
{}
Unlocked
(!Timer10_done &&
Motion_Detected)/
{Timer30_start = 1
Unlocked = 1}
(Timer10_done &&
Motion_Detected)/
{Timer30_start = 1}
Unlocked
& Motion
Detected
Alarm
Motion
Detected

Unless otherwise stated, the outputs are assigned in the following way for each transition:
Unlocked = 0;
Alarm_Buzzer = 0;
Timer10_start = 0;
Timer30_start = 0;




Unless otherwise stated, the outputs are assigned in the following way for each transition:
Unlocked = 0;
Alarm_Buzzer = 0;
Timer10_start = 0;
Timer30_start = 0;
Alarm Armed
Unlocked
(!Timer10_done && !Motion_Detected && Timer30_Done)/ {Unlocked = 1}
(Timer10_done && !Motion_Detected && Timer30_done)/ {}
(!Timer10_done && !Motion_Detected && !Timer30_Done)/ {Unlocked = 1}
Unlocked & Motion Detected
(!Timer10_done && Motion_Detected)/ {Timer30_start = 1 Unlocked = 1}
(Timer10_done && Motion_Detected)/ {Timer30_start = 1}
(Timer10_done && !Motion_Detected && !Timer30_Done)/ {}
Alarm
Motion Detected


Unless otherwise stated, the outputs are assigned in the following way for each transition:
Unlocked = 0;
Alarm_Buzzer = 0;
Timer10_start = 0;
Timer30_start = 0;
(Cal1Card_ID == 0 && !Motion_Detected && !Door_Open)/ {}
Alarm Armed
(Cal1Card_ID != 0 && !Motion_Detected)/ {Unlocked = 1 Timer10_start = 1}
(!Timer10_done && !Motion_Detected)/ {Unlocked = 1}
Unlocked
(Timer10_done && !Motion_Detected)/ {}
(!Timer10_done && Motion_Detected)/ {Timer30_start = 1 Unlocked = 1}
(Timer10_done && Motion_Detected)/ {Timer30_start = 1}
(!Timer10_done && !Motion_Detected && Timer30_Done)/ {Unlocked = 1}
(Timer10_done && !Motion_Detected && Timer30_done)/ {}
(Cal1Card_ID != 0 && Motion_Detected)/ {Timer10_start = 1 Timer30_start = 1 Unlocked = 1}
(!Timer10_done && !Motion_Detected && !Timer30_Done)/ {Unlocked = 1}
(Cal1Card_ID == 0 && Motion_Detected && !Door_Open)/ {}
Unlocked & Motion Detected
(!Timer10_done && Motion_Detected)/ {Timer30_start = 1 Unlocked = 1}
Cal1Card_ID == 0001/ {}
(Cal1Card != 0 && !Motion_Detected && Timer30_done)/ {Unlocked = 1 Timer10_start = 1}
(Cal1Card != 0 && !Motion_Detected !Timer30_done)/ {Unlocked = 1 Timer10_start = 1}
(Cal1Card_ID == 0 && !Motion_Detected && Door_Open)/ {Alarm_Buzzer=1}
(Timer10_done && Motion_Detected)/ {Timer30_start = 1}
(Cal1Card != 0 && Motion_Detected)/ {Unlocked = 1 Timer10_start = 1 Timer30_start = 1}
(Timer10_done && !Motion_Detected && !Timer30_Done)/ {}
Alarm
(Cal1Card==0 && !Motion_Detected && Timer30_done)/ {}
Motion Detected
Cal1Card_ID != 0001/ {Alarm_Buzzer = 1}
(Cal1Card_ID == 0 && !Motion_Detected && !Timer30_done)/ {}
(Cal1Card == 0 && Motion_Detected)/ {Timer30_start = 1}

Observations about this FSM

- Mealy vs. Moore
  - We designed a Mealy style FSM for the door controller
  - Would we need any extra states to formulate it as a Moore machine?
    - How about all those times we restarted timers?
- If we had information on whether or not timers were running, we probably could have eliminated the “Unlocked & Motion Detected” state
  - Would have required changing/adding arcs to the FSM

Ready/Valid Interfaces

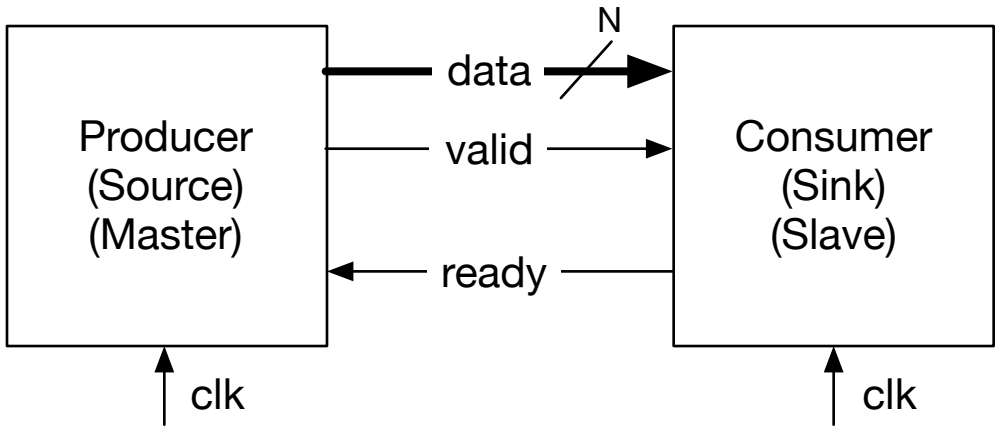
Interfacing Between Modules & Ready/Valid

- Up to this point, we have been primarily using modules that we have written ourselves.
  - We know how the control logic for our modules works and how to interface with it
- What if we want to integrate with modules written by others, without knowing all the specifics of their control logic?

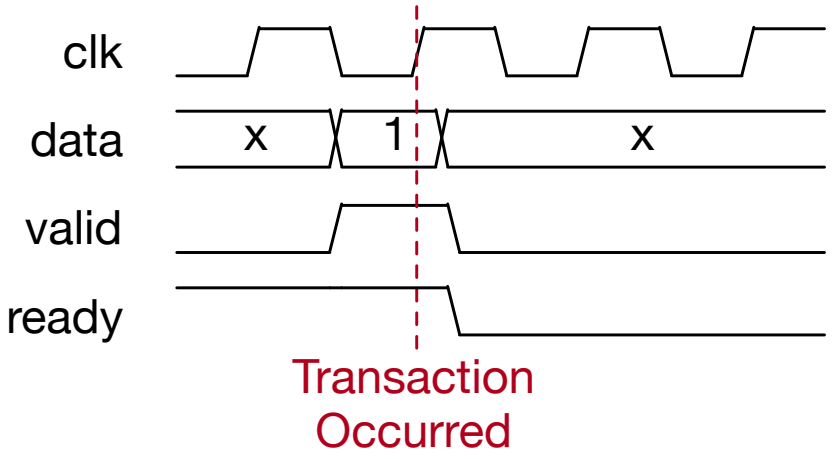
- Standard interfaces, such as ready/valid, help us do this

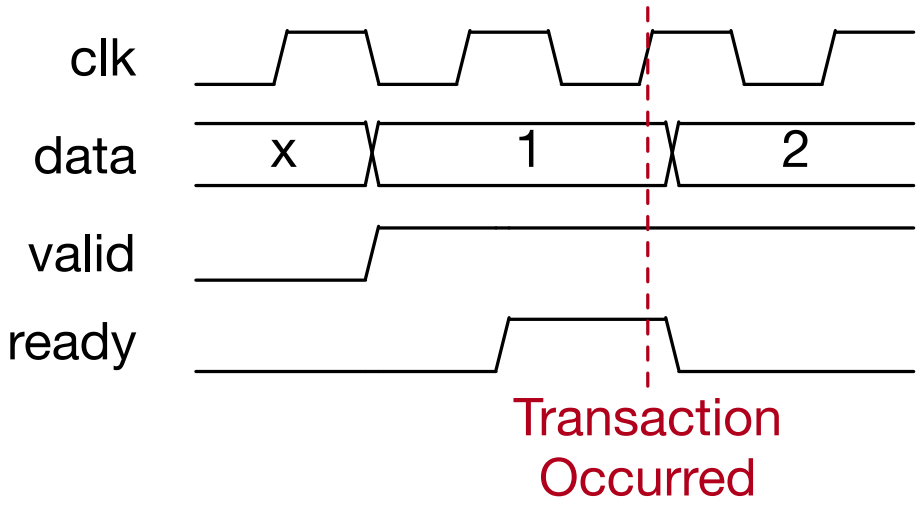
Semantics of Ready/Valid Transfer

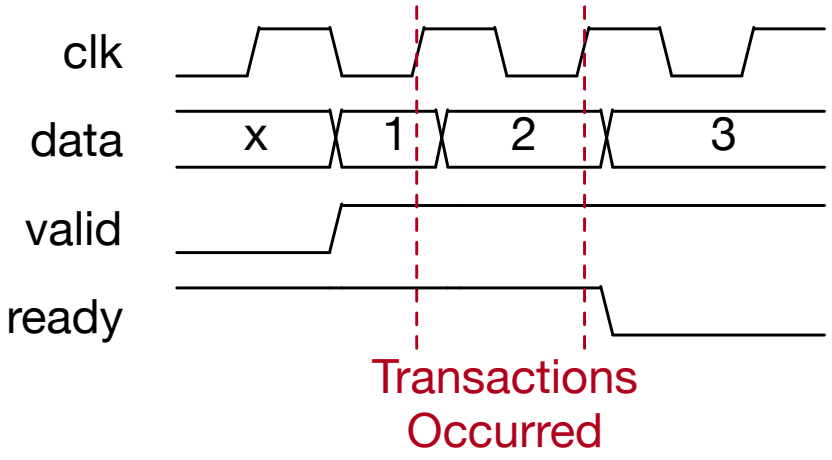
- The producer (source, master) is sending data to the consumer (sink, slave)
- The data to be transferred is placed on the ‘data’ line
- Valid = 1 indicates that the values on the ‘data’ line are valid
- Ready = 1 indicates that the consumer is ready to consume ‘data’ on the next posedge of the clock
- A transfer occurs at the positive edge of the clock when ready and valid are both high



Semantics of Ready/Valid Transfer







Semantics behind Ready/Valid Use

- What are the responsibilities for the producing and consuming modules?
  - When should producer required to pull the valid line high?
    - Should the producer wait for the ready line to go high before pulling valid?
  - When should the consumer pull the ready line high?
    - Should the consumer wait for valid data to be present before pulling ready?
- Module writers taking opposing assumptions can result in deadlock
  - Producer is waiting for consumer to become ready before raising valid high
  - Consumer is waiting for the producer to present valid data before becoming ready
  - Nothing will ever happen! They will wait for each other forever!

AXI4-Stream Ready/Valid Semantics

- AXI4 Streaming is a Ready/Valid style interface standard which answers the question of what the producer and consumer are responsible for
- Standard developed by ARM as part of the AMBA Series of Standards
  - A common bus standard used in SoC and FPGA projects
- Most signal names have a ‘t’ prepended to them
  - tvalid, tdata, tready
- Signal clock is called aclk
- Reset is called aresetn (active low)
- Contains additional optional control lines
  - ex. tlast denotes the last word transferred in a packet
- Full specification available at: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0051a/index.html (requires registration with ARM)

AXI4-Stream Ready/Valid Semantics Summary

- Transfer occurs when *both* valid and ready are asserted (at a positive clock edge)
- Producer:
  - The producer is *cannot* to wait for ready to be high before asserting valid
    - The producer *must* assert valid when possible
  - The producer *must* hold valid high and data constant until transaction occurs
- Consumer:
  - The consumer *may* wait for valid to be true before asserting ready
    - This can waist a clock cycle
  - The consumer *may* assert ready when no valid data is present
    - This is the best option for most consumers as transactions will occur on the next clk rising edge
  - The consumer is *allowed* to de-assert ready (if ready was already asserted) before valid is asserted

Implementing Logic With Pass Gates

Implementing Logic with Pass Gates