FPGA-Based Logic Analyzer

CPE 426/526 Spring 2016

Ashton Johnson, David Hurt, Ian Swepston, Paul Henny

Topics Covered

- Overview
- Requirements
- Architecture
- Sub-Modules
- Integration
  - Module
- Device Targeting
  - Synthesis
- Demo
- Status
- Q&A

Overview

FPGA-Based Logic Analyzer Overview

- Captures and displays multiple signals from a system under test
- Can be triggered to begin recording by a specific pattern or sequence of signals
- Used as a final stage of testing to uncover hardware defects
- Also useful for reverse engineering

Goals

1. Using VHDL, develop a generic model of a logic analyzer that will require minimal modifications to target different FPGA vendors and their associated technologies
2. Provide a means of downloading captured data to a computer
3. Develop testbenches for functional validation of design at each hierarchical level
4. Develop a simple device-under-test model for exercising and demonstrating the system capabilities.
5. Demonstrate design implementation on FPGA development board, which is to be determined

Requirements

- Capture and store logic signals
- 32 channel input
- Capable of parallel triggering involving multiple channels
- Sample each channel up to 100 MHz
- Interface with 3rd party Sump GUI

Requirements - GUI

- Uses GNU GPL Licensed Sump Logic Analyzer GUI and UART protocol.
- Implements the following Commands:
  - Reset
  - Run
  - Set Trigger Mask
  - Set Trigger Values
  - Read & Delay Count
  - Set Divider
  - Device ID

Sump project: http://www.sump.org/projects/analyzer/protocol/

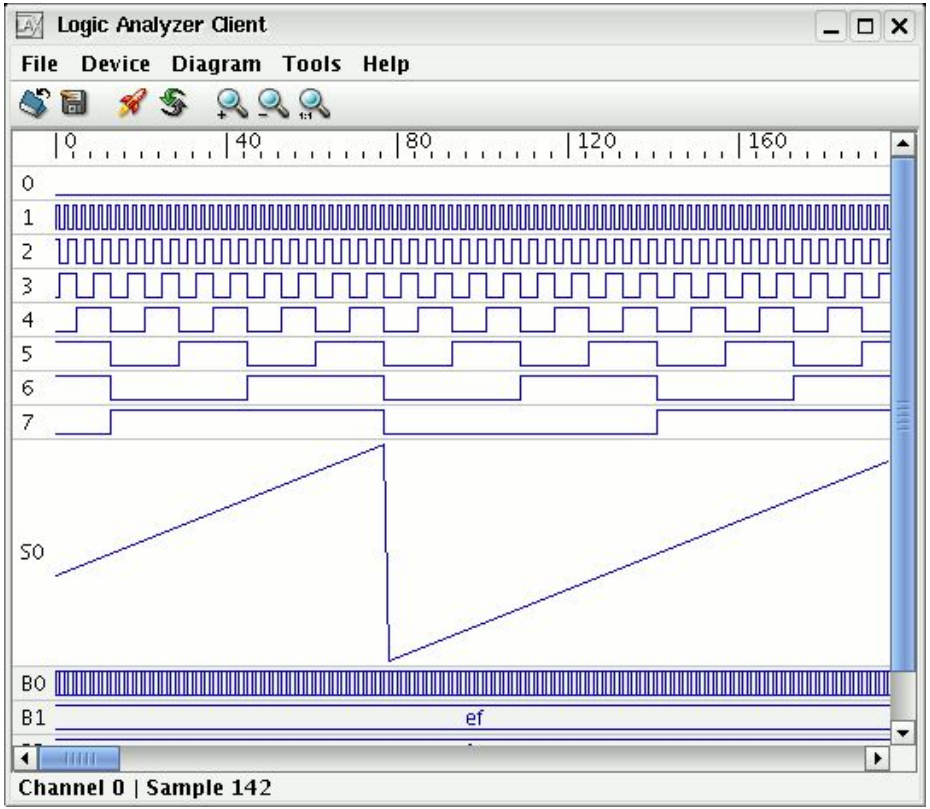
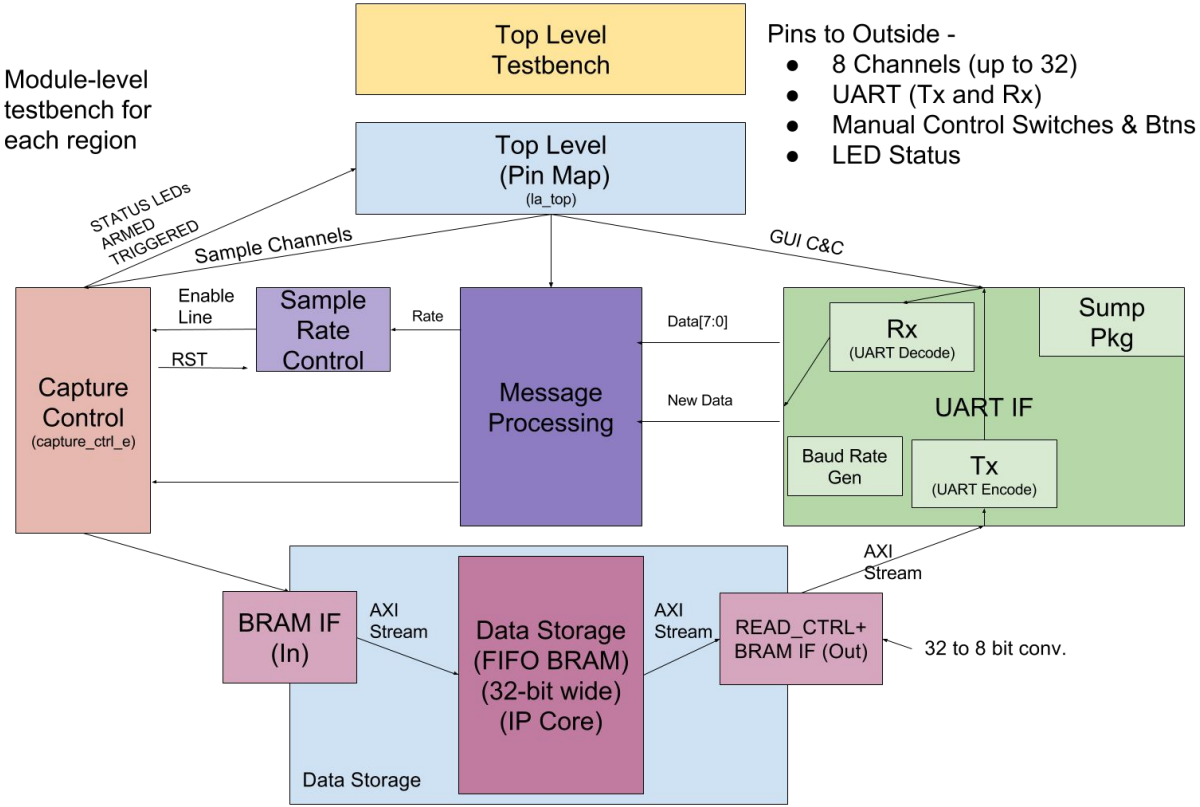


Image source: sump.org

Architecture

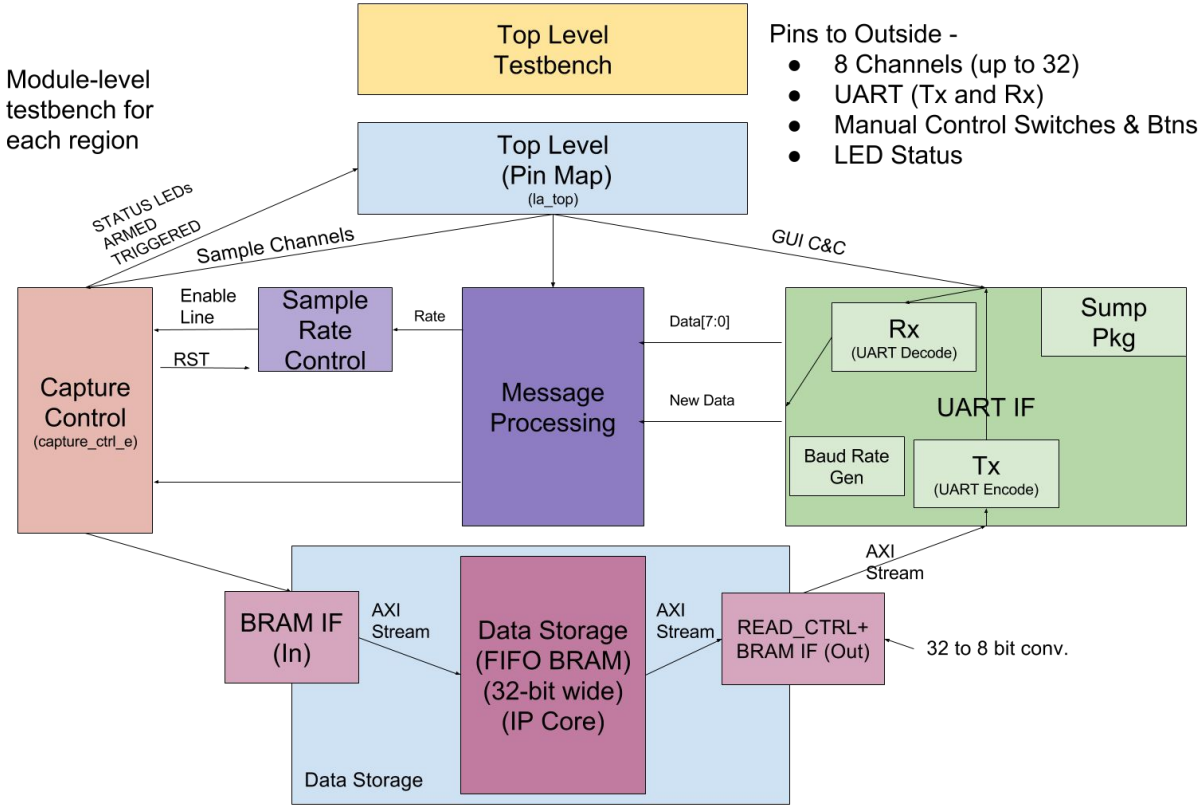


Sub-Modules

UART Module

Ian Swepston

Architecture



What is a UART?

- UART: Universal Asynchronous Receiver/Transmitter
  - Hardware device that translates data between parallel and serial forms. (Wiki)
  - More simply: a device for serial communication.
- Protocol (8n1, no parity):
  - This UART uses a 10 bit line code for both Tx and Rx.
  - Bits arrive at selected Baud Rate. (Ours: 115,200 bps)

Bit 0	Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6	Bit 7	Bit 8	Bit 9
Start Bit (Always ‘0’)	Data0 (LSB)	Data1	Data2	Data3	Data4	Data5	Data6	Data7 (MSB)	Stop Bit (Always ‘1’)

UART Communications Overview

- Interface between GUI (SUMP) and FPGA.
- Two “levels” both FSM based.
  - Top Level - Handles communication with other design units
  - UART – Implements UART serial protocol
- Each model contains independent FSMs for sending and receiving.
- UART driven by decimated clock to work with baud rate
- All resets are synchronous

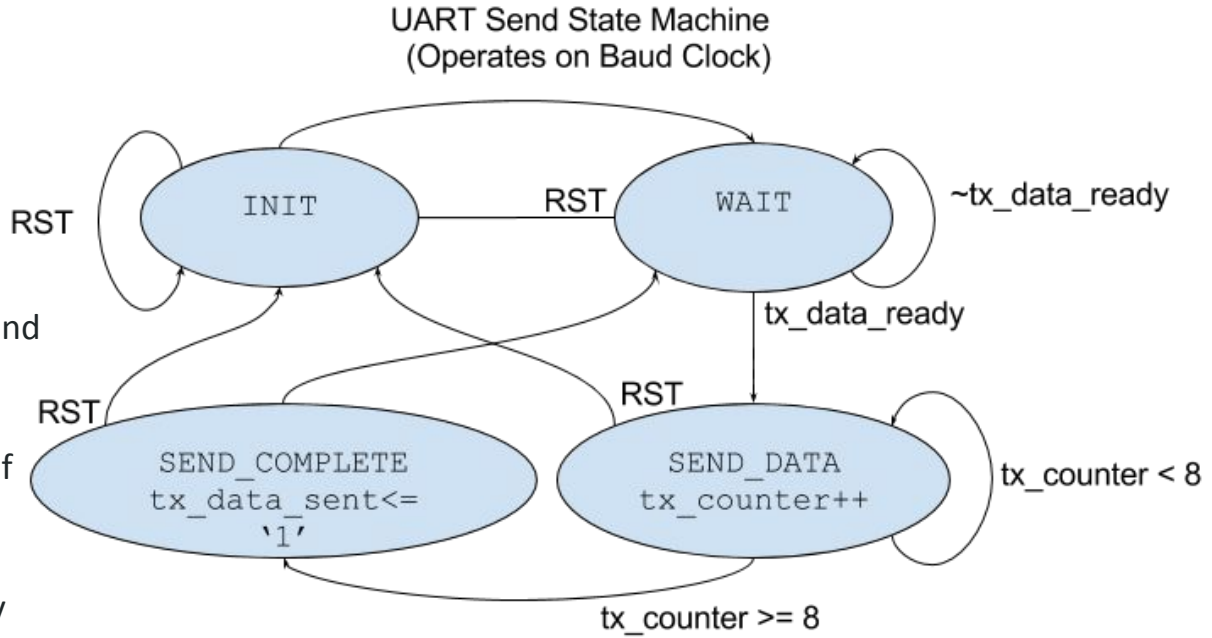
UART Interface

```
generic (baud_rate  : positive := 115_200;
         clock_freq : positive := 100_000_000);  -- Make sure we keep integer division here

port(clk              : in  std_logic;  -- clock
     rst              : in  std_logic;  -- reset logic
     rx_get_more_data : in  std_logic;  -- stop bit found for stream in
     rx_data_ready    : out std_logic;  -- stream out ready
     rx               : in  std_logic;  -- receive line
     data_in          : in  std_logic_vector(7 downto 0) := (others => '0');  -- data to be transmitted
     tx_data_ready    : in  std_logic;  -- stream out stop bit sent
     tx_data_sent     : out std_logic;  -- ready for rx
     tx               : out std_logic;  -- transmit line
     data_out         : out std_logic_vector(7 downto 0));
```

UART Send

USES BAUD CLOCK

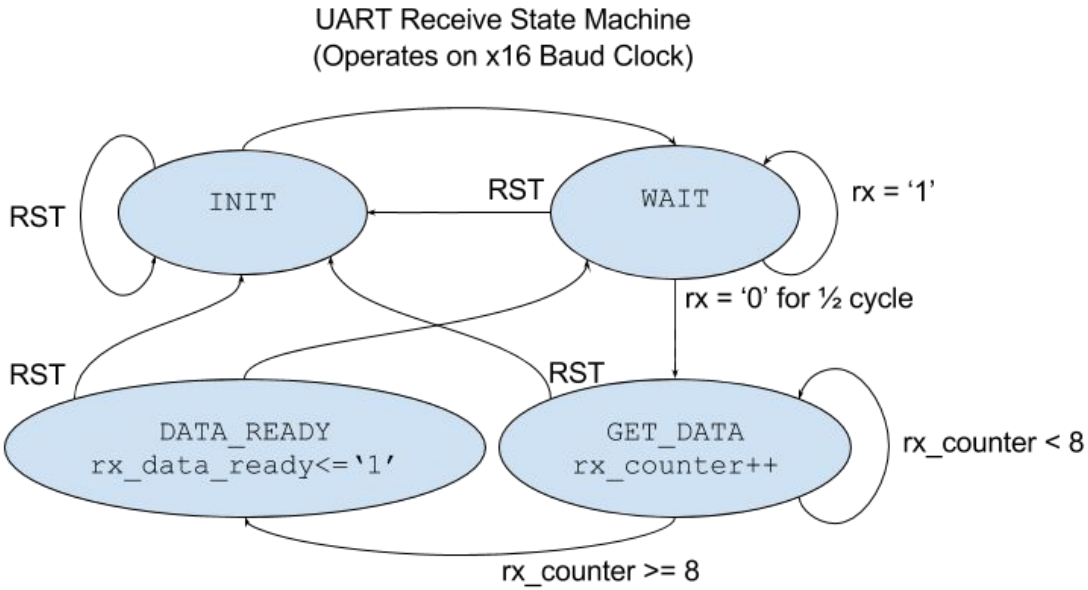
- INIT – reset state
- WAIT
  - Wait until outside signal to send buffer. Keep tx line high.
- SEND_DATA
  - Send start bit, then each bit of send buffer, then stop bit
- SEND_COMPLETE
  - Signal that data is sent. Ready for more.



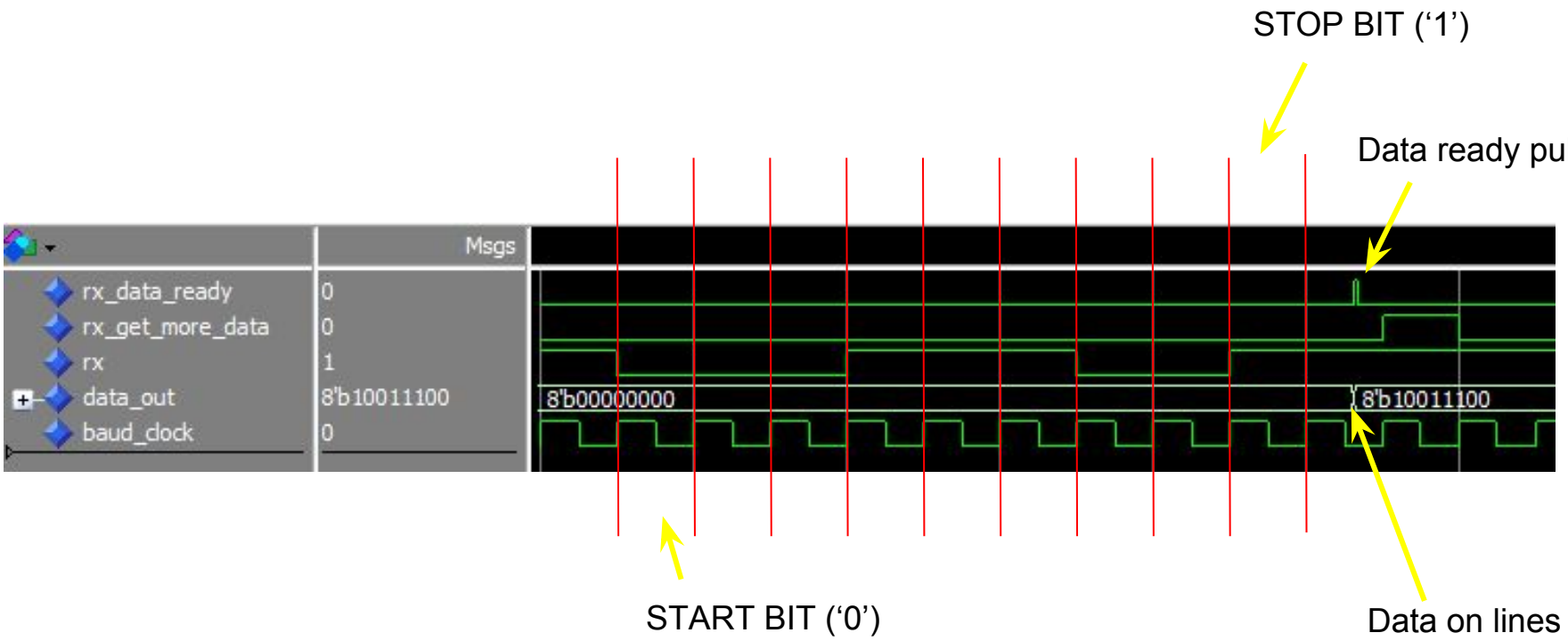
UART Receive

USES X16 BAUD CLOCK

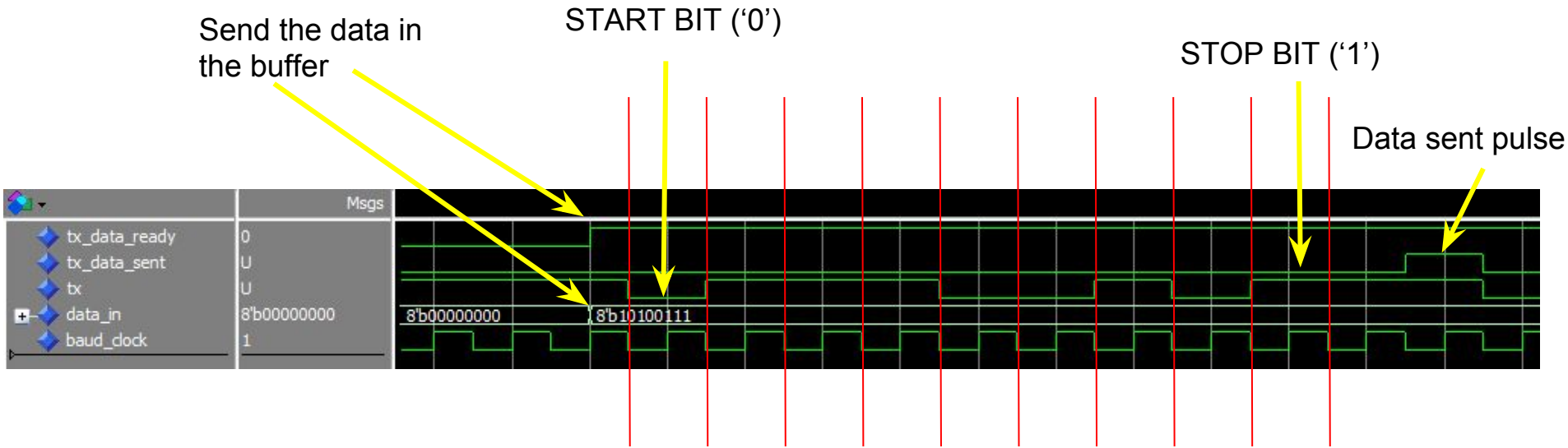
- INIT – reset state
- WAIT
  - Wait until there has been ‘0’ on the line for half a cycle
- GET_DATA
  - Start shifting data into buffer and counting bits
- DATA_READY
  - After 8 bits and a stop bit, send the rx_data_ready signal



UART Receiver



UART Sender

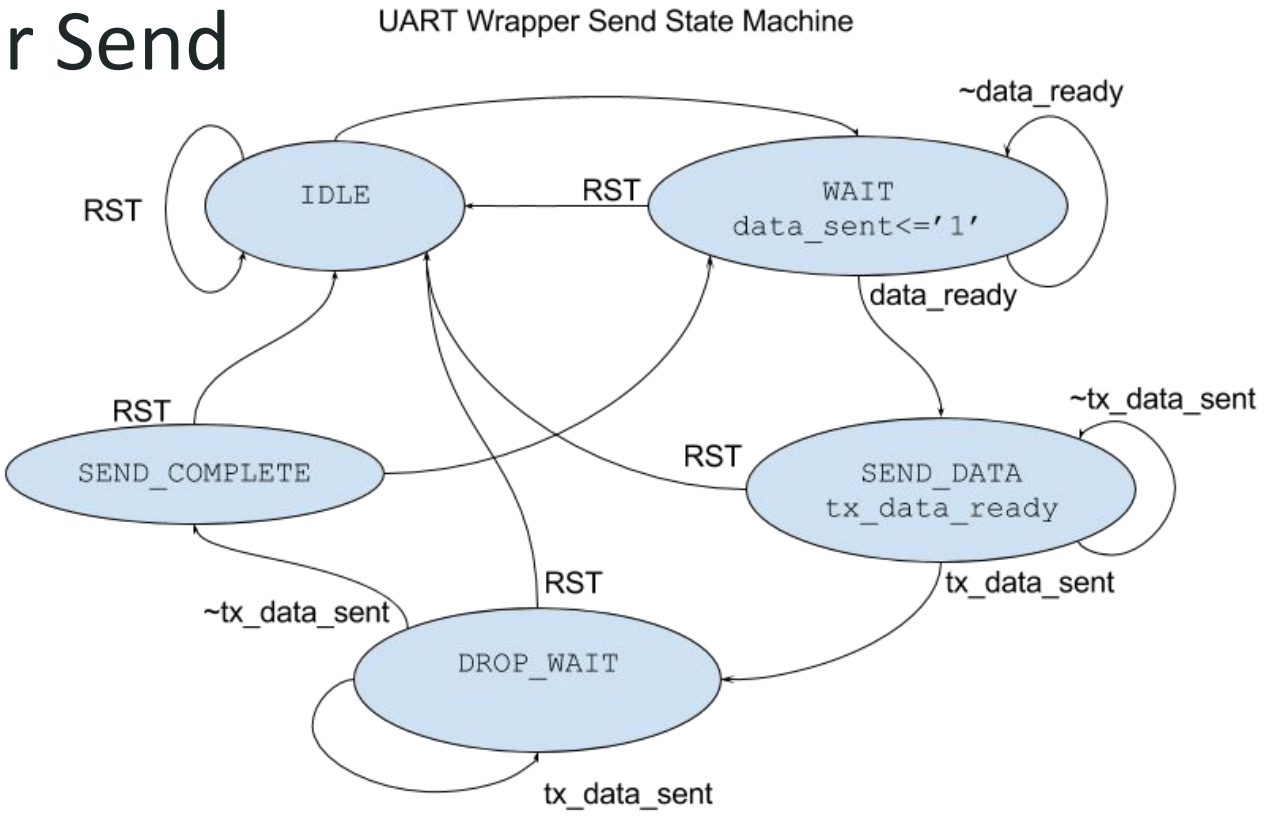

Send the data in the buffer
START BIT (‘0’)
STOP BIT (‘1’)
Data sent pulse
Msgs
tx_data_ready
tx_data_sent
tx
data_in
baud_clock
0
U
U
8'b00000000
1
8'b00000000
8'b10100111

UART Wrapper Interface

```
generic (baud_rate  : positive := 115_200;
         clock_freq : positive := 100_000_000);

 port(clk                : in  std_logic;      -- clock
      rst                : in  std_logic;      -- reset
      rx                 : in  std_logic;      -- data line from top level
      tx                 : out std_logic;
      tx_command         : in  std_logic_vector(7 downto 0);  -- data from storage

      data_ready                 : in  std_logic;      -- flag for transmit message
      data_sent          : out std_logic;      -- flag for transmit message

      command_ready      : out std_logic;   -- flags for data message collect
      command            : out std_logic_vector(7 downto 0));  -- commands for message handler
```

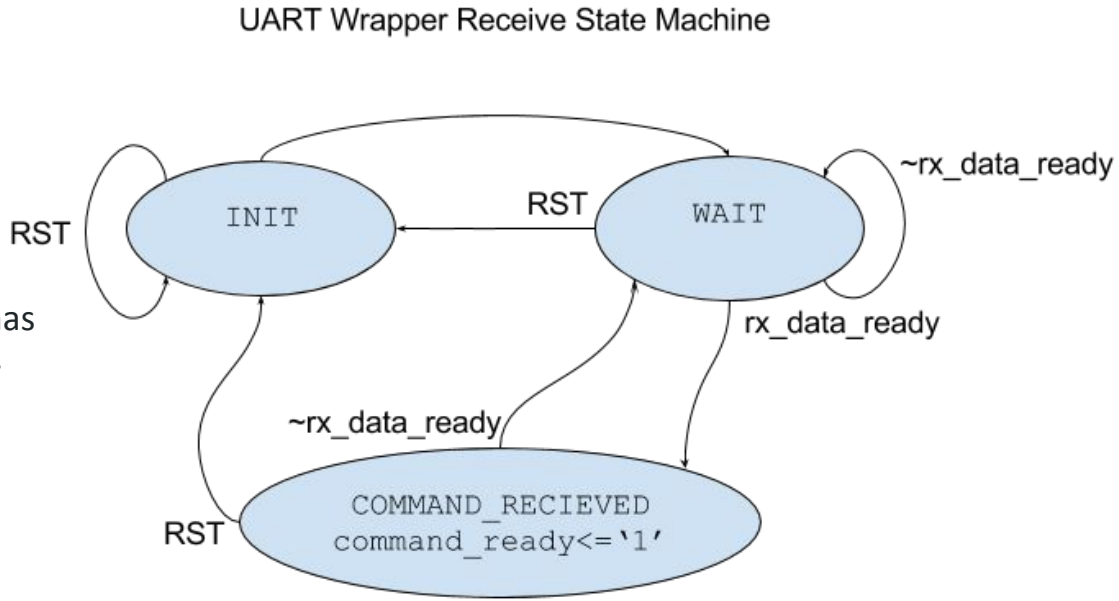
UART Wrapper Send

- INIT - reset
- WAIT – wait for send
- SEND_DATA
  - Send command to UART, wait for done
- DROP_WAIT
  - Wait for “done” to go low
- SEND_COMPLETE
  - Signal sending complete



UART Wrapper Receive

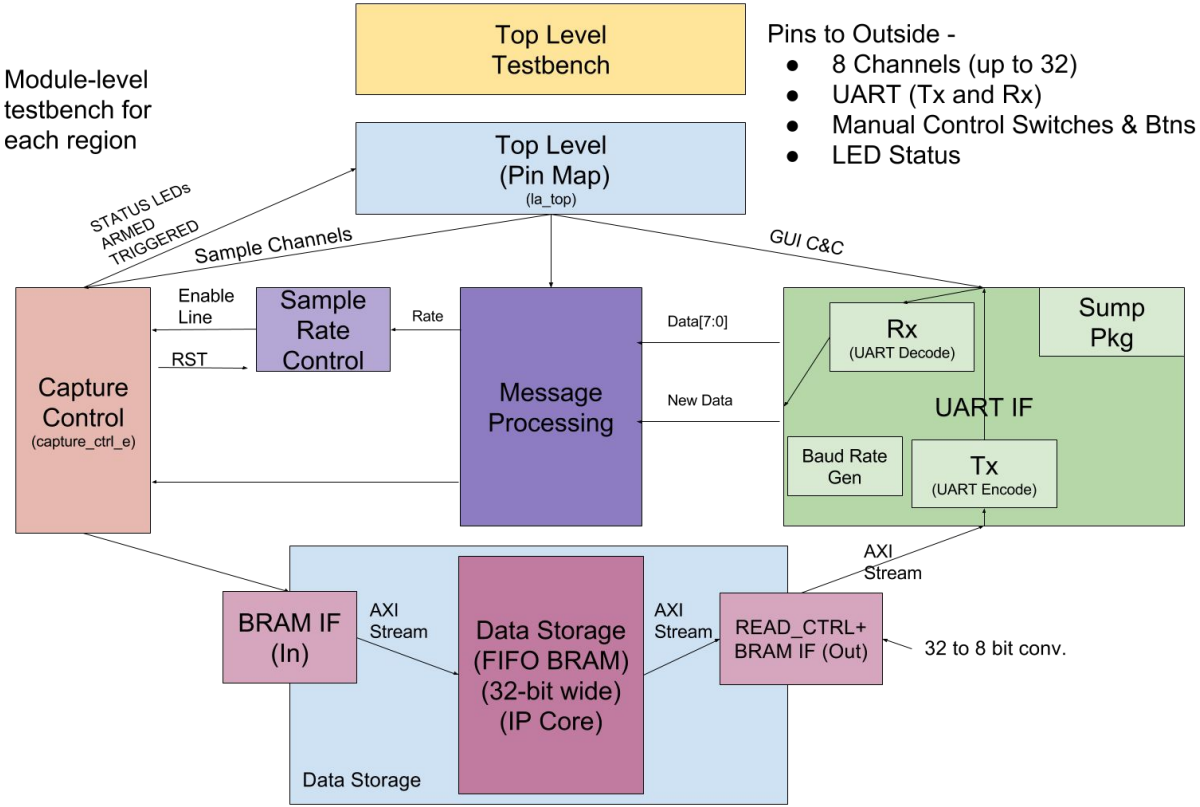
- INIT - reset
- WAIT
  - wait for something on the line
- COMMAND_RECIEVED
  - Entered when a complete byte has arrived. Pulses command_ready.



Message Processor Module

David Hurt

Architecture



Message Processor Overview

Role: Decode command bytes from the GUI into instructions for the logic analyzer

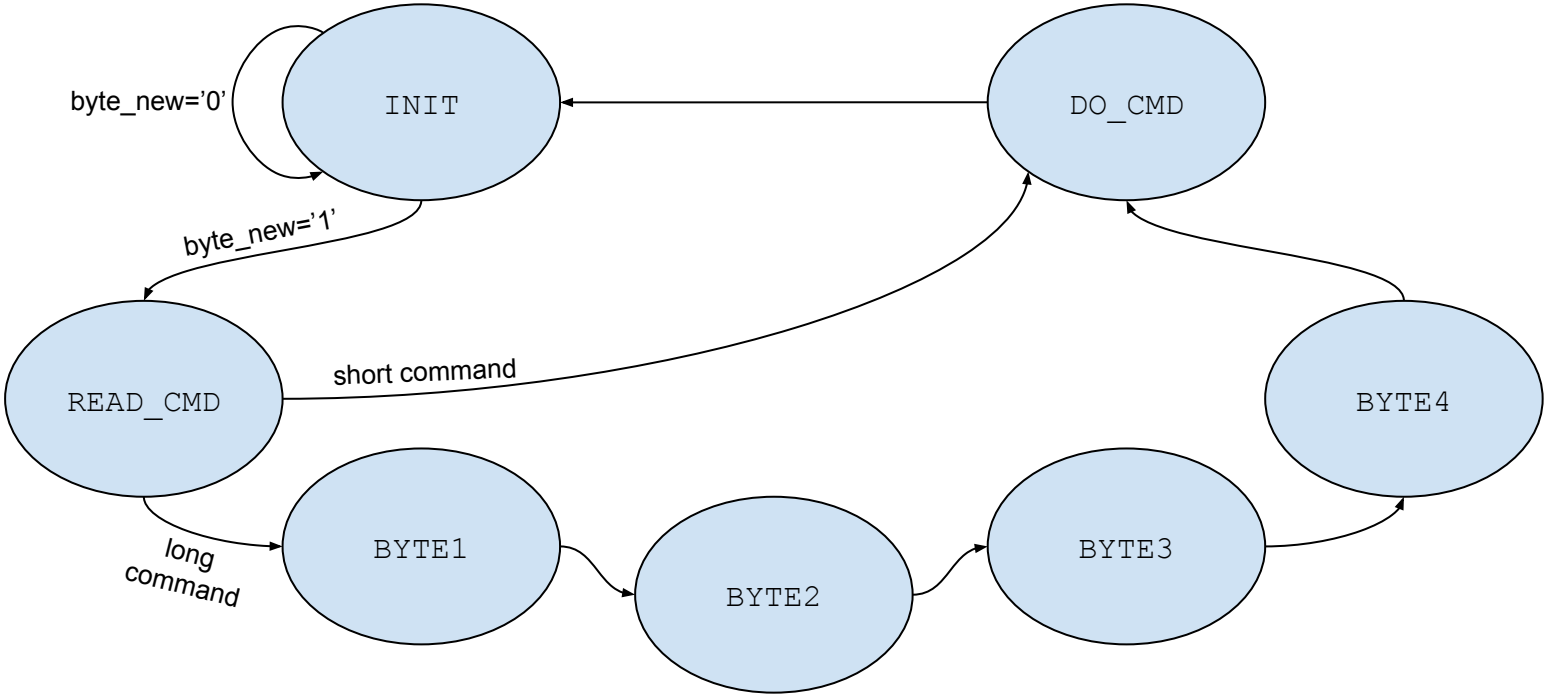
States:

- INIT -- Wait for complete command byte from UART
- READ_CMD -- Determine what kind of command is being sent
- BYTE1 - 4 -- For long commands, read the four bytes of data
- DO_CMD -- Set appropriate output signals for the command

Message Processor Interfaces

```
entity msg_processor is
 port(
   -- Global Signals
   clk : in std_logic;  -- Clock
   rst : in std_logic;  -- Synchronous reset


   -- UART Interface
   byte_in  : in std_logic_vector(7 downto 0);  -- Byte of command/data from UART
   byte_new : in std_logic;                     -- Strobe to indicate new byte


   -- Sample Rate Control Interface
   sample_f : out std_logic_vector(23 downto 0);  -- Sampling frequency to Sample Rate Control


   -- Capture Control Interface
   reset      : out std_logic;                      -- Reset capture control
   armed      : out std_logic;                      -- Arm capture control
   send_ID    : out std_logic;                      -- Send device ID
   send_debug : out std_logic;                      -- Send debug status
   read_cnt   : out std_logic_vector(15 downto 0);  -- Number of samples (divided by 4) to send to memory
   delay_cnt  : out std_logic_vector(15 downto 0);  -- Number of samples (divided by 4) to capture after trigger
   trig_msk   : out std_logic_vector(31 downto 0);  -- Define which trigger values must match
   trig_vals  : out std_logic_vector(31 downto 0)   -- Set the trigger's individual bit values
   );
end entity msg_processor;
```

Message Processor State Machine



Messages

Short Commands

These commands are exactly one byte long.

Reset (00h)

- Resets the device

Run (01h)

- Arms the trigger

ID (02h)

- Asks for device identification

Long Commands

These commands are five bytes long.

Set Trigger Mask (C0h)

- Defines which trigger values must match

Set Trigger Values (C1h)

- Defines what value each bit must have

Set Divider (80h)

- Set the sampling rate divider

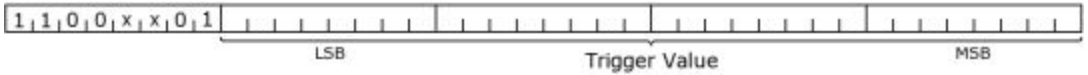
Set Read & Delay Count (81h)

- Set how many samples to read

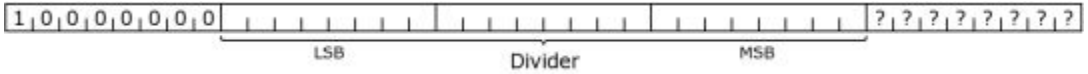
Messages (cont.)

Set Trigger Mask (C0h)

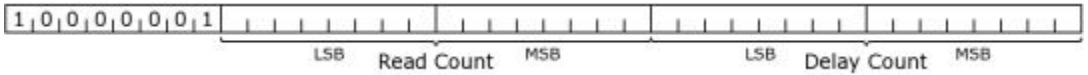
Set Trigger Values (C1h)



Set Divider (80h)



Set Read & Delay Count (81h)



Images source: sump.org

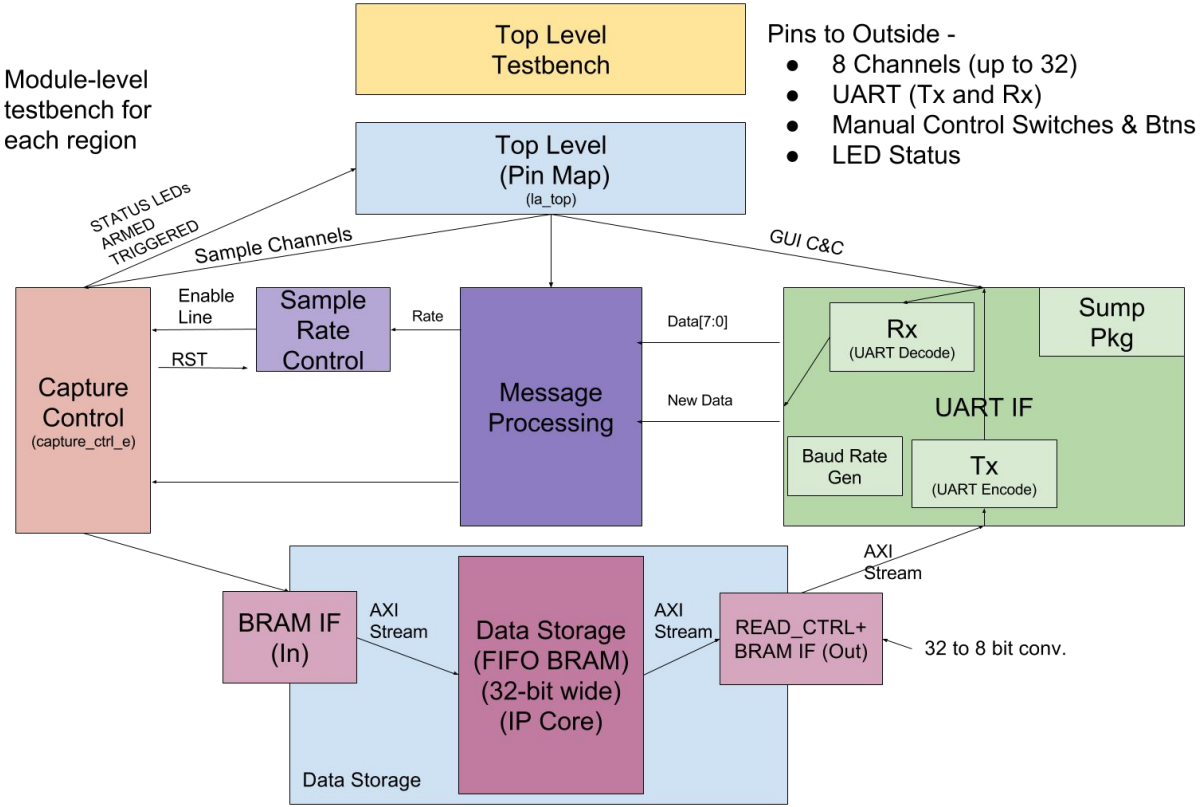
Sample Rate Control

- Enables sampling rates at less than the system clock rate
- Drives a sample enable line to the capture control module

Capture Controller Module

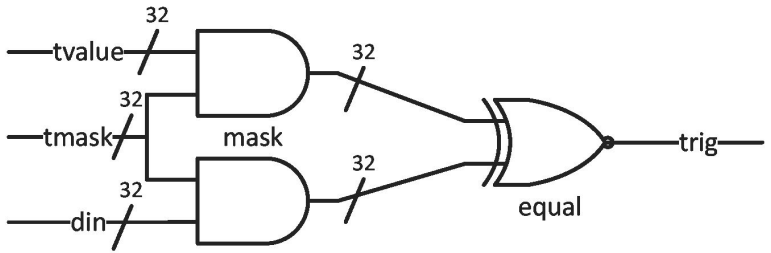
Ashton Johnson

Architecture



Capture Controller Overview

- **Purpose**: Match trigger masked value to sampled data bus.
- FSM based.
  - INIT:
    - Wait for downstream FIFO ready (fifo_tready)
    - Clear outputs
  - WAIT_FOR_ARM_CMD:
    - Reading trigger value and masks
    - Assert ‘ARMED’
  - WAIT_FOR_TRIGGER:
    - Async Compare of masked DIN & TRIGGER VALUE
    - Resets Sample Rate Divider
  - DELAY_HOLD:
    - Specified # of delay cycles
  - CAPTURE_DATA
    - Specified # of samples



Capture Controller Interfaces

```
GENERIC (
   DATA_WIDTH : POSITIVE RANGE 1 TO 32 := 32);
 PORT (
   --top level interfaces
   clk       : IN  STD_LOGIC;          -- Clock
   rst       : IN  STD_LOGIC := '0';   -- synchronous reset
   din       : IN  STD_LOGIC_VECTOR(DATA_WIDTH-1 DOWNTO 0);  -- input channels
   --status indicators
   armed     : OUT STD_LOGIC;          --latched indicator when armed.
   triggered : OUT STD_LOGIC;          --latched indicator when triggered.
```

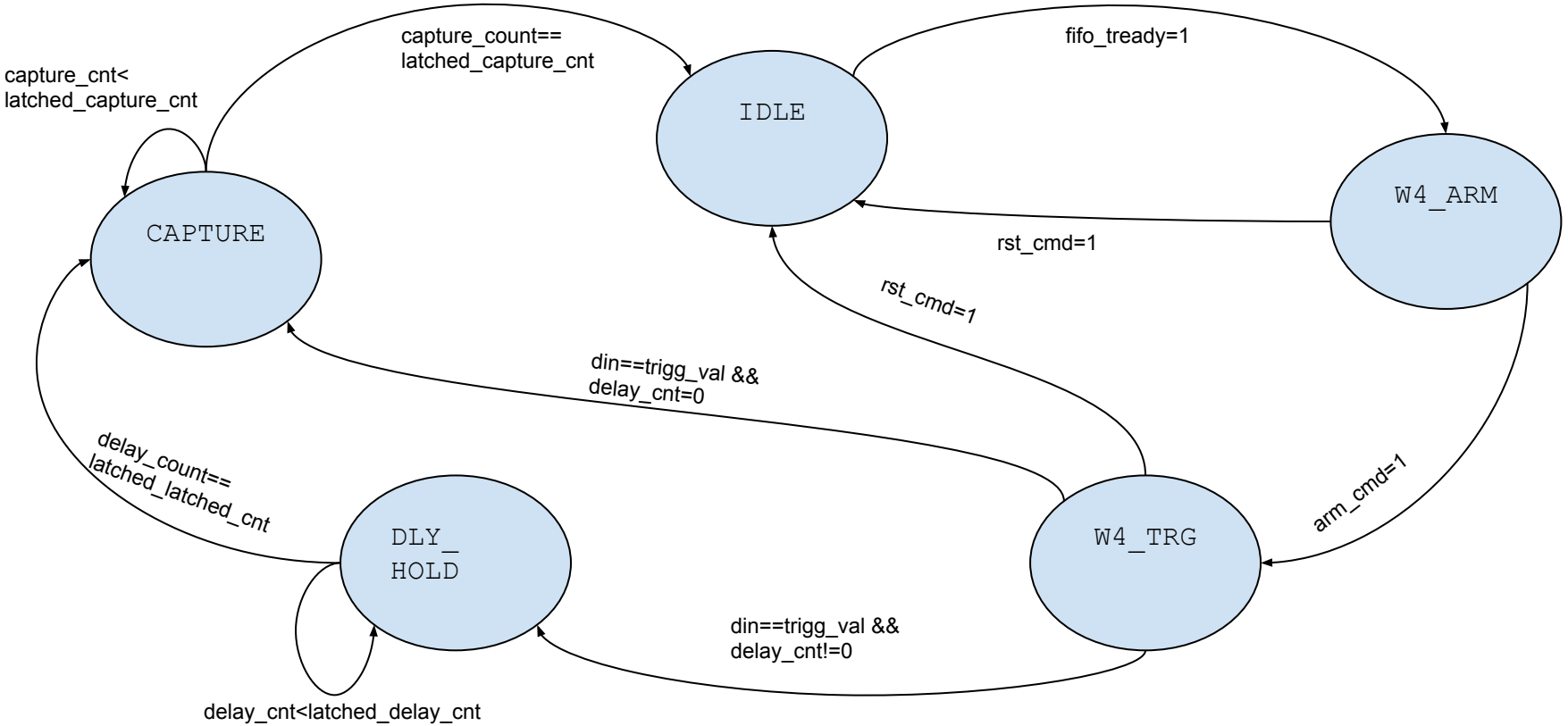
Capture Controller Interfaces (cont.)

```
------message processing interfaces
  --serially received reset command. one clock cycle required
  rst_cmd       : IN  STD_LOGIC                    := '0';
  --serially received arm command. one clock cycle required.
  arm_cmd       : IN  STD_LOGIC;
  --sample enable trigger. for subsampling data.
  sample_enable  : IN  STD_LOGIC                   := '1';
  --send a reset pulse to the sample rate clock
  sample_cnt_rst : OUT STD_LOGIC;
  --number of sample_rate cycles to delay capturing data after trigger has occurred.
  delay_cnt_4x : IN  STD_LOGIC_VECTOR(16-1 DOWNTO 0) := (OTHERS => '0');
  --number of samples to read, times four. max==262,140 samples
  read_cnt_4x    : IN  STD_LOGIC_VECTOR(16-1 DOWNTO 0) := (OTHERS => '1');
  --parallel trigger bit mask for par_trig_val. latched in on arm_cmd
  par_trig_msk   : IN  STD_LOGIC_VECTOR(32-1 DOWNTO 0) := (OTHERS => '0');
  --parallel trigger values, latched in on arm_cmd
  par_trig_val   : IN  STD_LOGIC_VECTOR(32-1 DOWNTO 0) := (OTHERS => '1');
  --ready_to_arm indicator
  capture_rdy    : OUT STD_LOGIC;
```

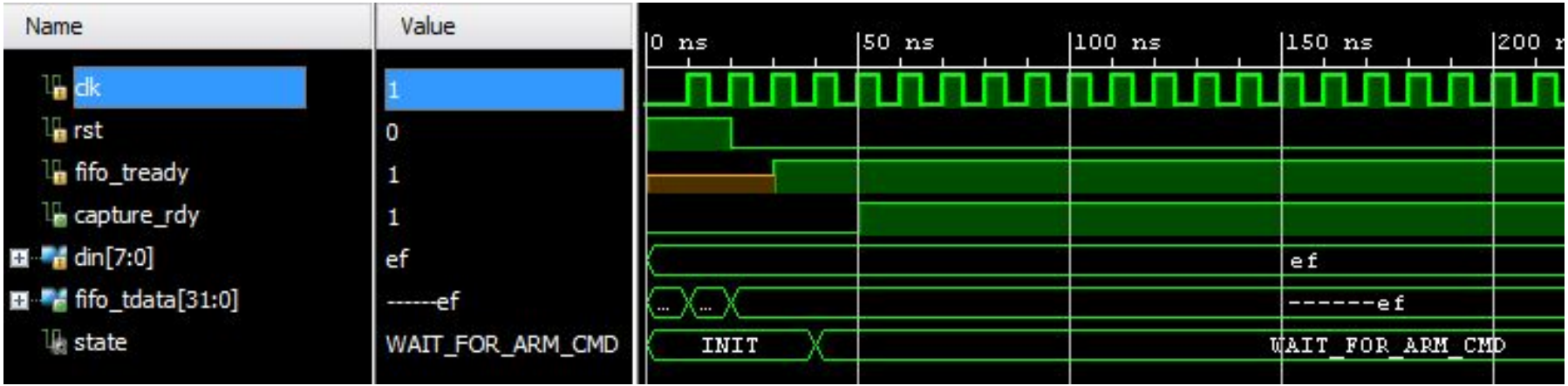
Capture Controller Interfaces (cont.)

```
--fifo interface
  fifo_tdata  : OUT STD_LOGIC_VECTOR(32-1 DOWNTO 0);  --captured
                                          --data output
  fifo_tvalid : OUT STD_LOGIC;        -- indicating tdata has valid data
  fifo_tlast  : OUT STD_LOGIC;        -- indicates last word of sampled data
  fifo_tready : IN  STD_LOGIC := '1';  -- only used on initial setup
  fifo_tfull  : IN  STD_LOGIC := '0'; --no intended use
  fifo_tempty : IN  STD_LOGIC := '1'; --needs to control capture_rdy
  fifo_aresetn : OUT STD_LOGIC;  --used to flush fifo in reset cmd.
```

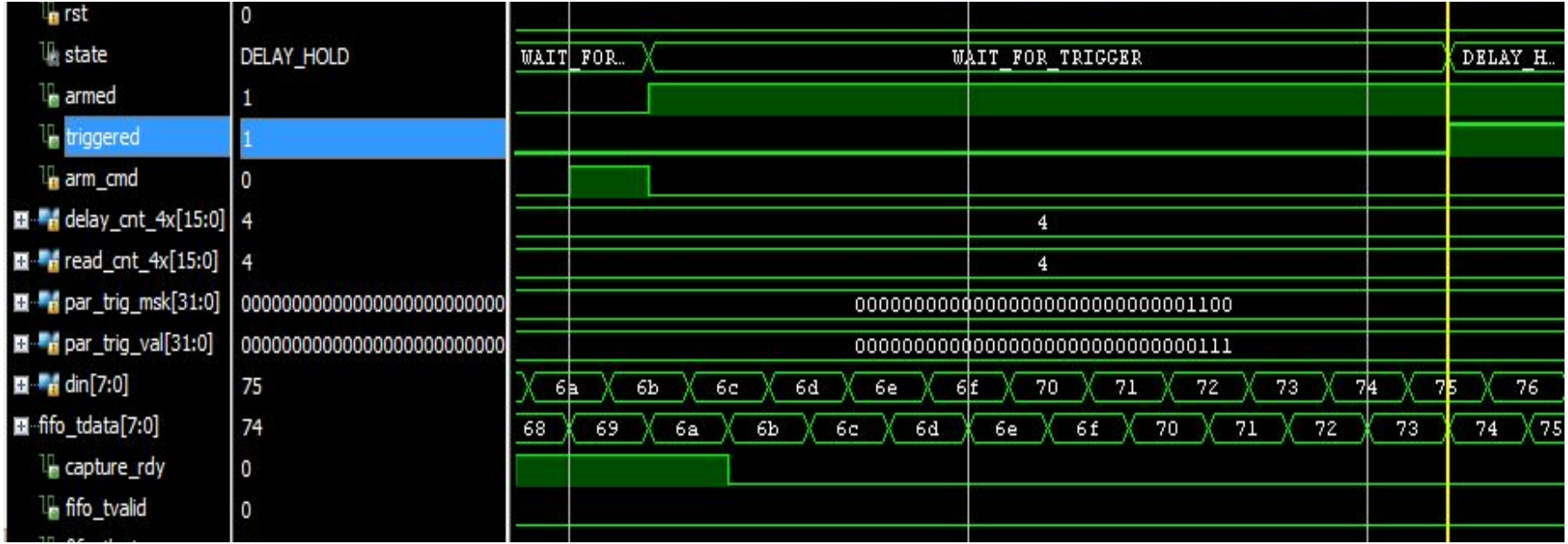
Capture Controller State Diagram



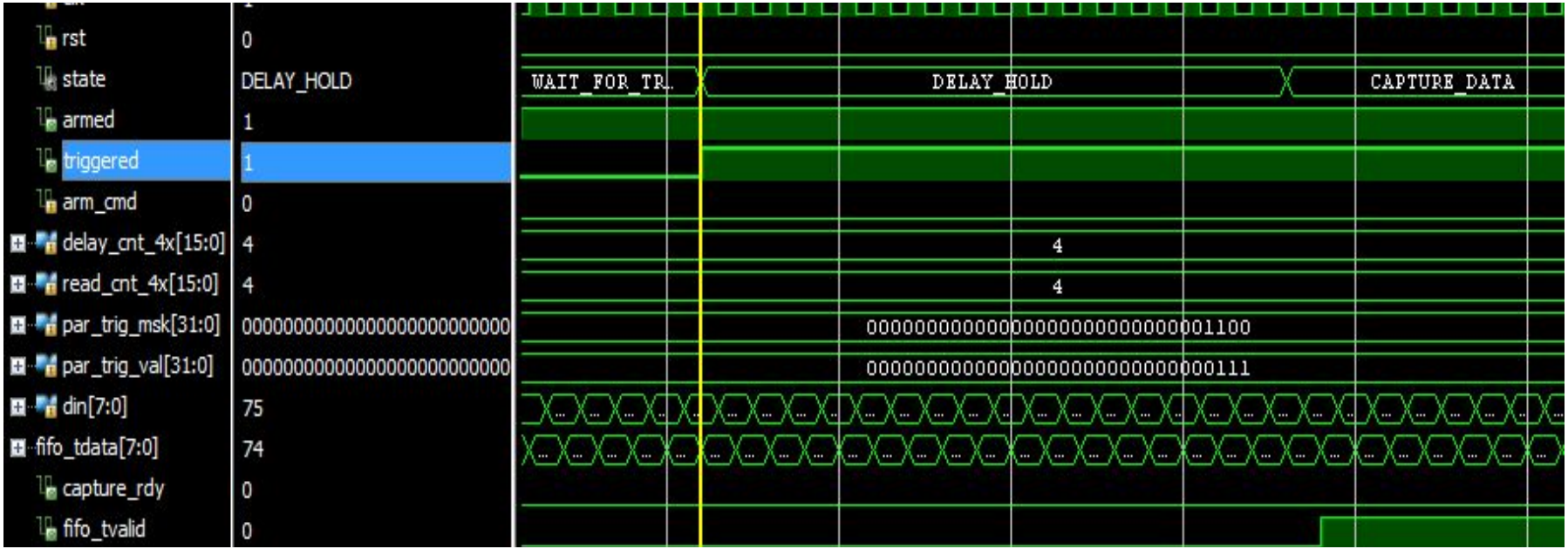
Capture Controller Init



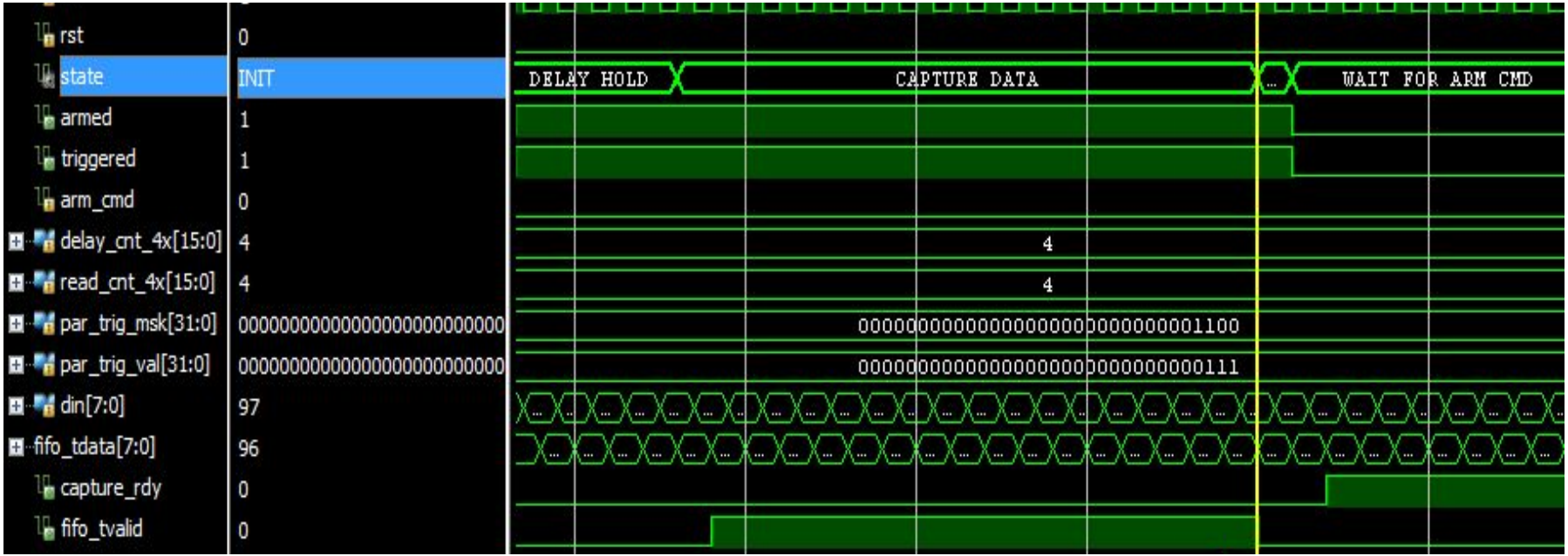
Capture Controller Arm & Trigger



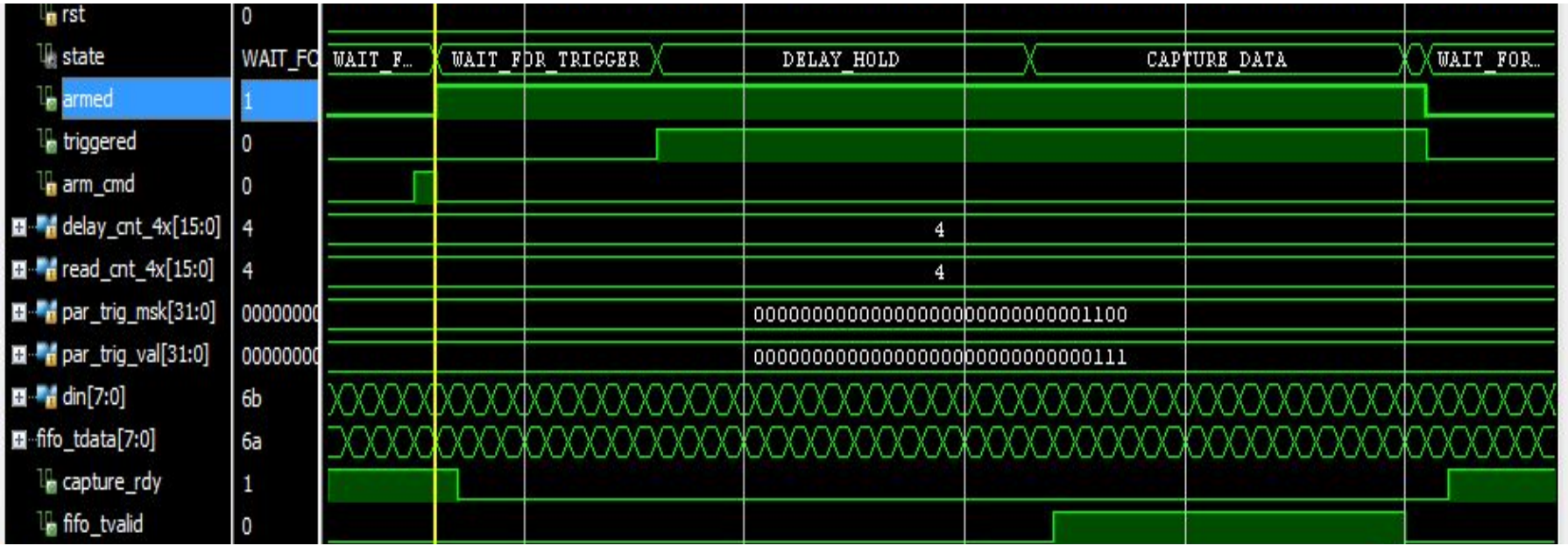
Capture Controller Delay & Capture



Capture Controller Delay & Capture



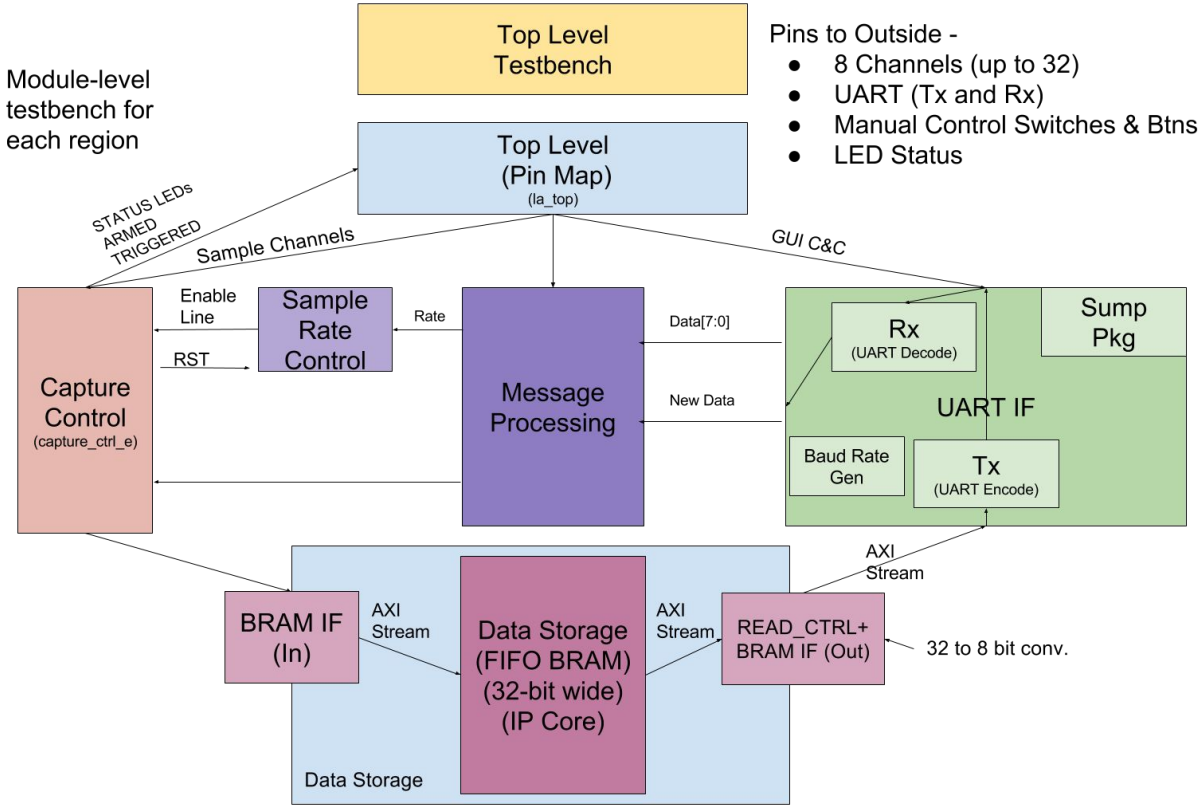
Capture Controller Full Cycle



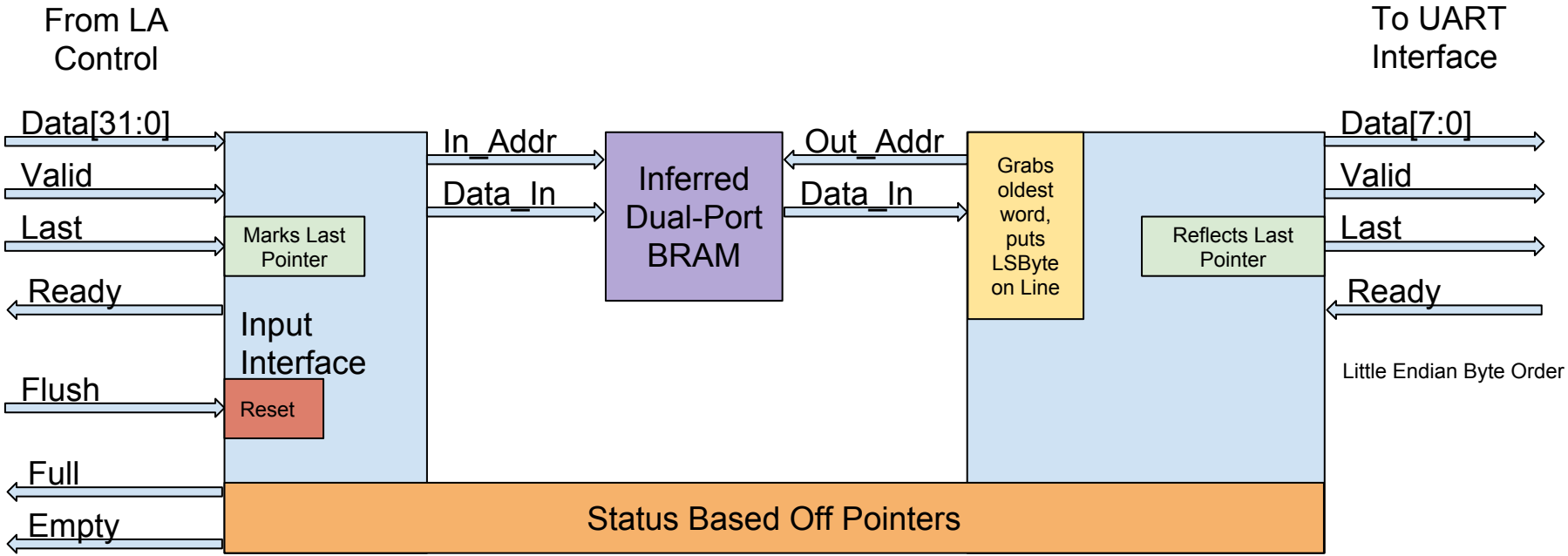
Data FIFO Module

Paul Henny

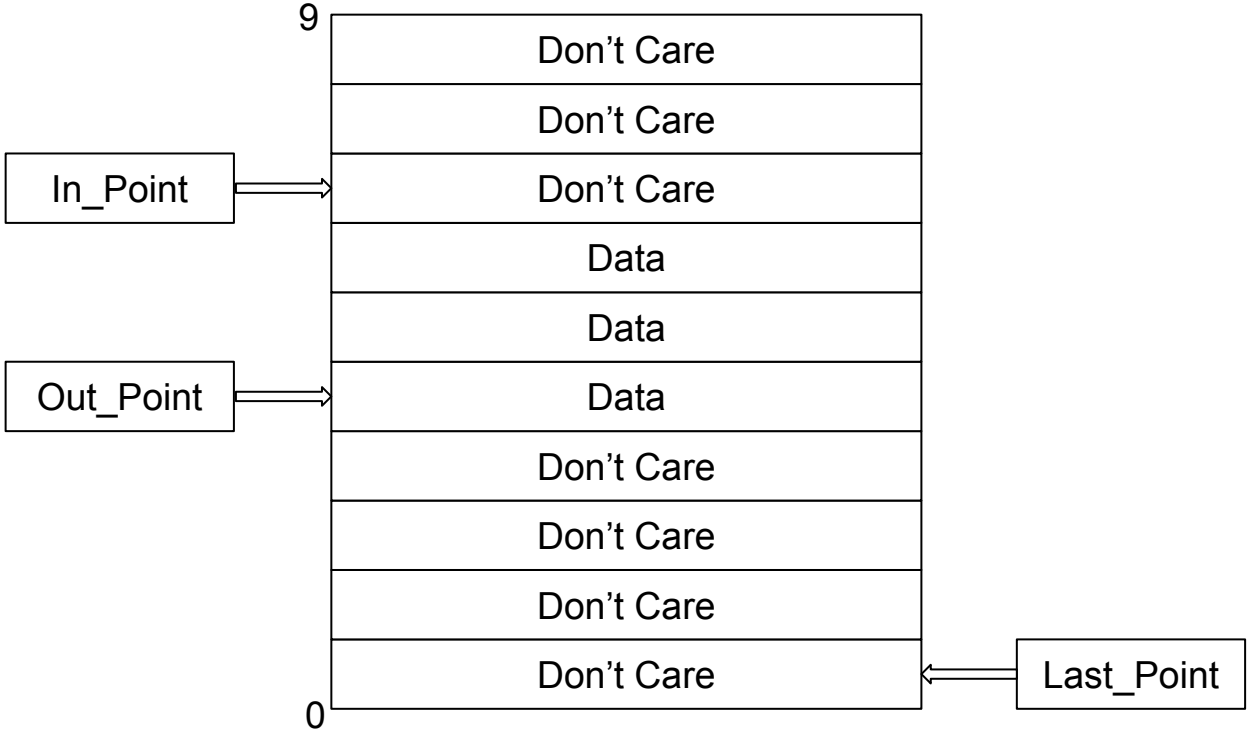
Architecture



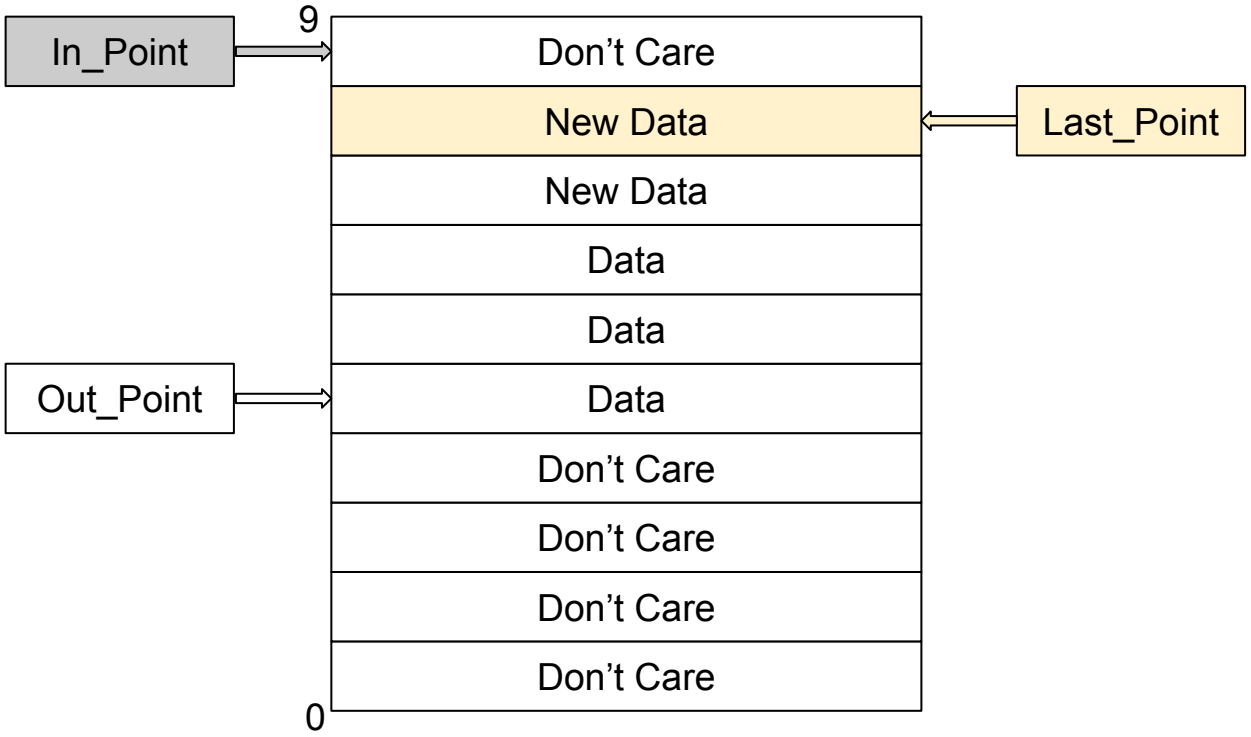
Data FIFO Interface Diagram



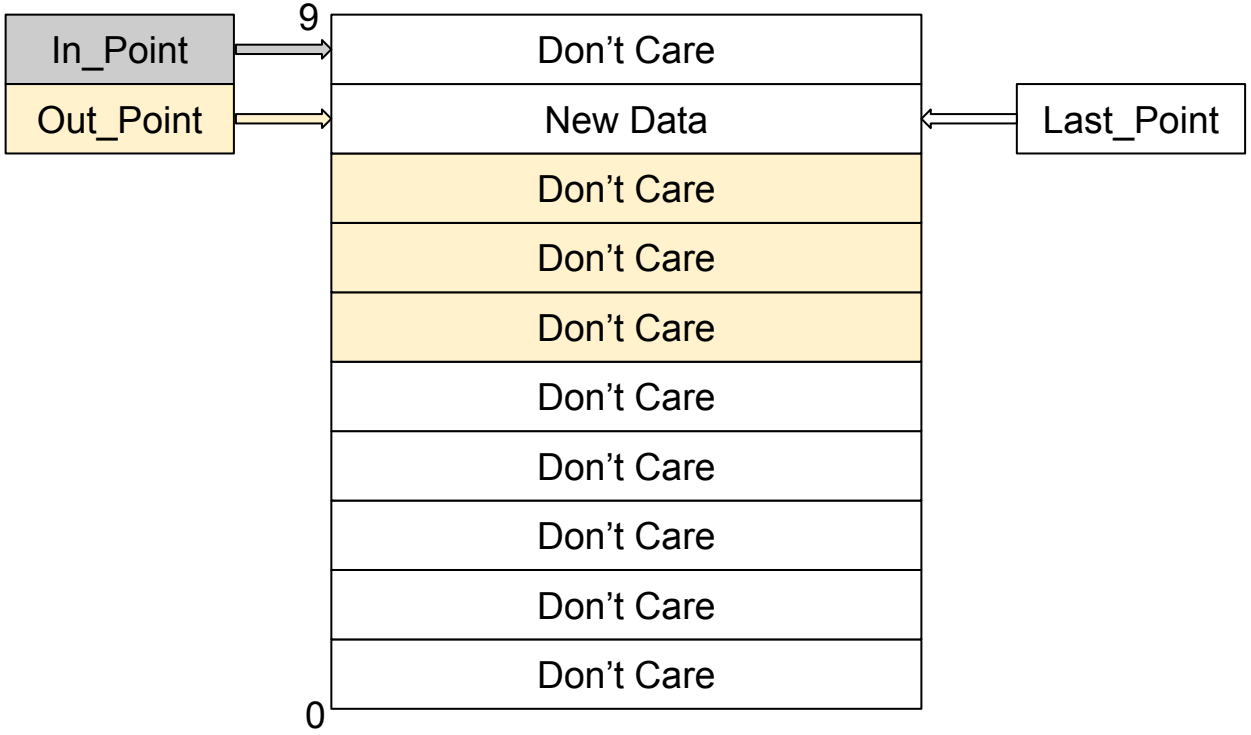
FIFO Design


9
Don’t Care
Don’t Care
In_Point
Don’t Care
Data
Data
Out_Point
Data
Don’t Care
Don’t Care
Don’t Care
Don’t Care
Last_Point
0

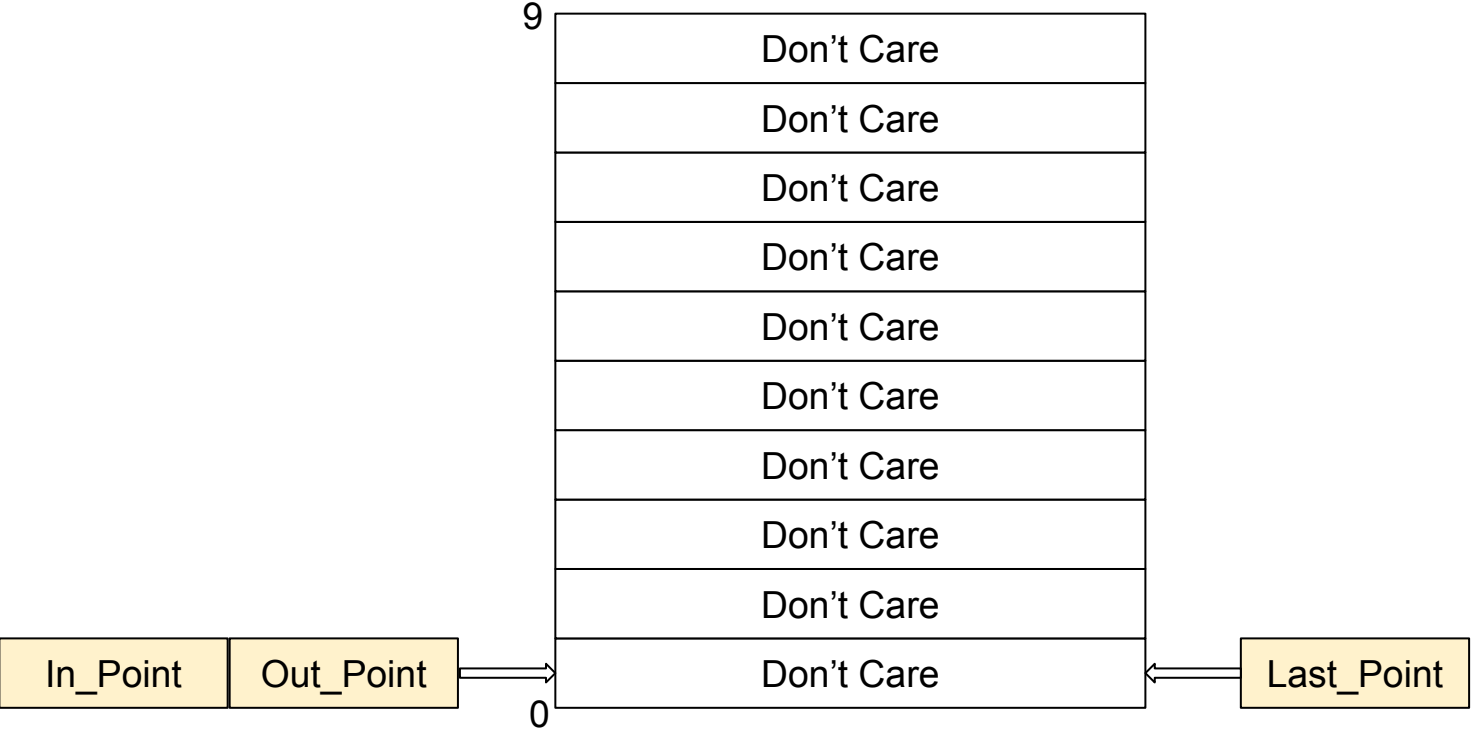
FIFO Design


In_Point
9
Don’t Care
New Data
Last_Point
New Data
Data
Data
Out_Point
Data
Don’t Care
Don’t Care
Don’t Care
Don’t Care
0

FIFO Design


In_Point
Out_Point
9
Don’t Care
New Data
Last_Point
Don’t Care
Don’t Care
Don’t Care
Don’t Care
Don’t Care
Don’t Care
Don’t Care
Don’t Care
0

FIFO Design


9
Don’t Care
Don’t Care
Don’t Care
Don’t Care
Don’t Care
Don’t Care
Don’t Care
Don’t Care
Don’t Care
Don’t Care
0
In_Point
Out_Point
Last_Point

Storage Testbench

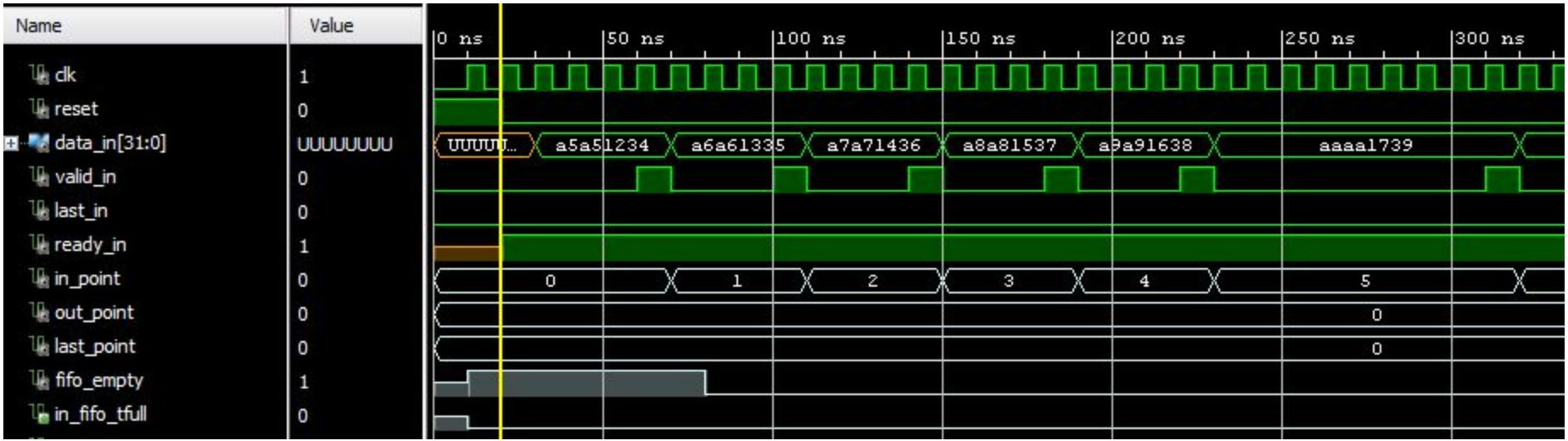
Testbench is made of two processes:
Input Interface and Output Interface

- Input Process
  - Every time store_word is called, a word is passed into FIFO
  - Increments word before sending another

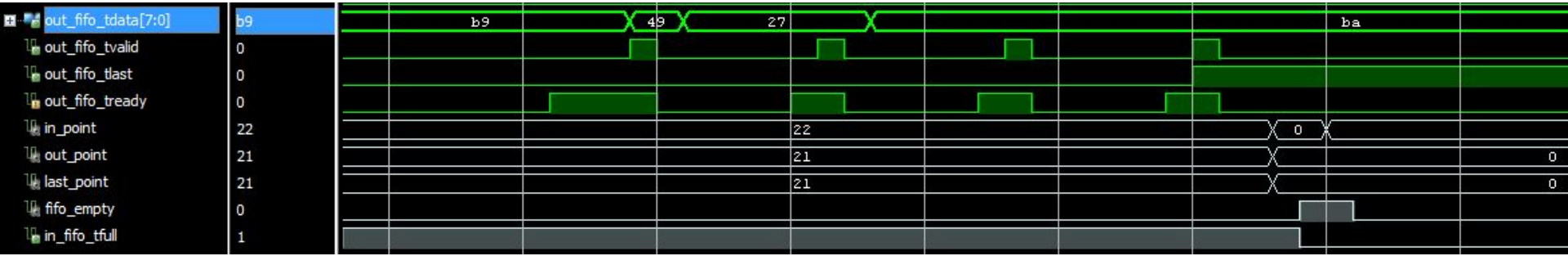
- Output Process
  - Reads out bytes from FIFO
  - Checks to see if data is expected value

```
-------------------------------------------------
-- inputs words into the "storage"
-------------------------------------------------
    -- Puts an incrementing word into storage, delay controls rate
    procedure store_word(delay : integer := 0) is
    begin
        valid_in <= '0'; -- will be overwritten if delay=0
        data_in <= std_logic_vector(data);
        data <= data + x"01010101";
        -- will wait that many clock cycles before writing another word
        for i in 1 to delay loop
            wait until rising_edge(clk);
        end loop;
        valid_in <= '1';
        wait until ready_in='1' and rising_edge(clk);
        valid_in <= '0';
    end procedure;
```

Testbench AXI Input



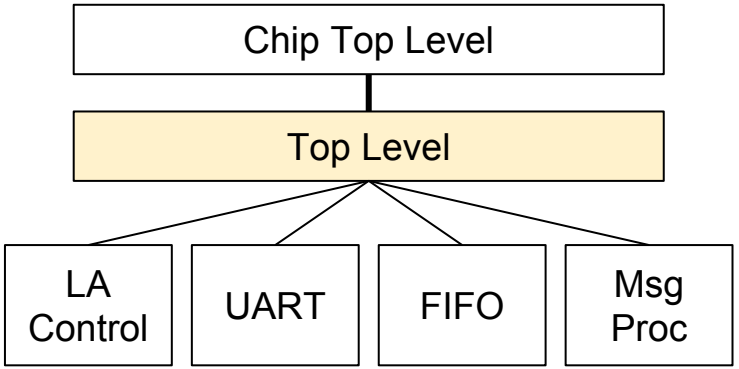
FIFO Testbench Last Word



Integration

Integration Steps

- Created top level file to call entities of all sub-modules
- Interconnected signals between modules
- Entity contains UART, DIN, clock, reset, and test lines to pins



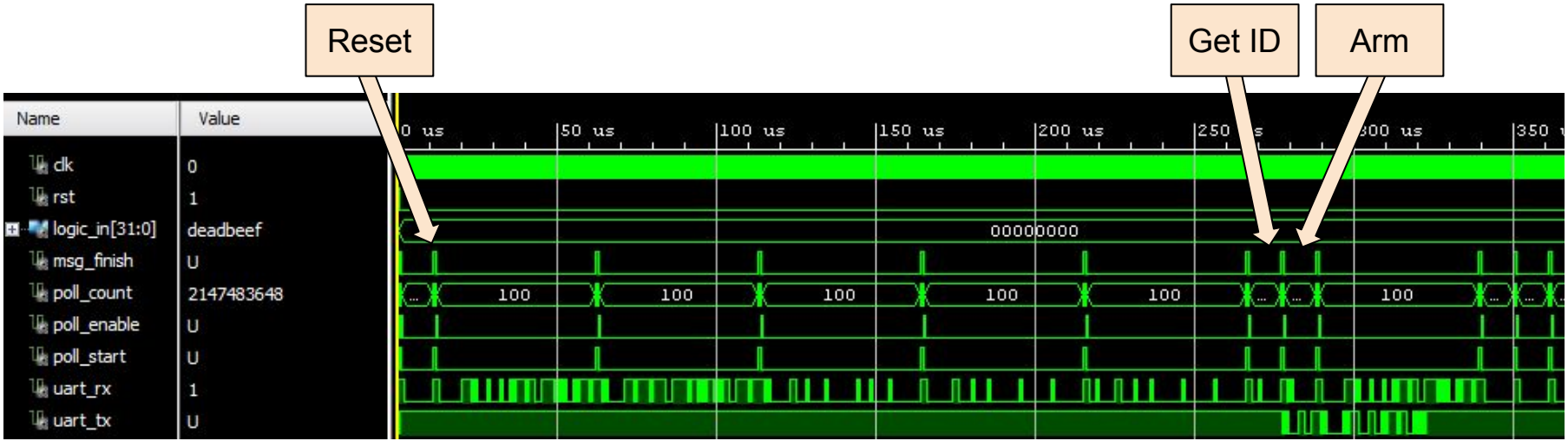
Top Level Testing

```
type slv8_arr is array (natural range <>) of std_logic_vector(7 downto 0);
type cmd_record is record
    mess : slv8_arr(0 to 4);
    length : integer range 0 to 5;
end record cmd_record;
```

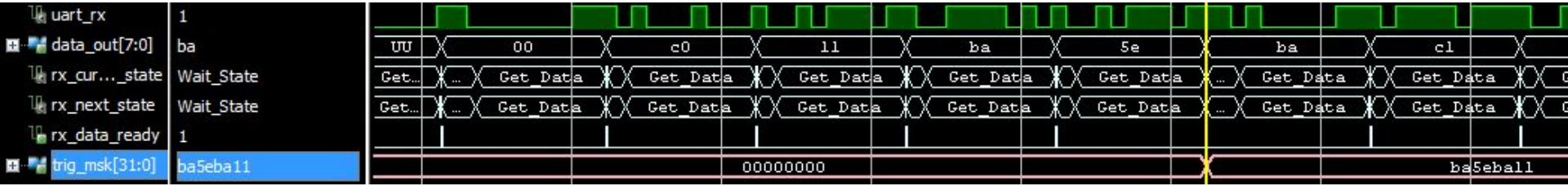
```
constant c_reset : cmd_record := ((x"00", others => x"00"), 1);
constant c_run : cmd_record := ((x"01", others => x"00"), 1);
constant c_id : cmd_record := ((x"02", others => x"00"), 1);
constant c_test_byte : cmd_record := ((x"A5", others => x"00"), 1);
constant c_trig_mask : cmd_record := ((x"C0", x"11", x"BA", x"5E", x"BA", others => x"00"), 5);
constant c_trig_val : cmd_record := ((x"C1", x"EF", x"BE", x"AD", x"DE", others => x"00"), 5);
constant c_read_cnt : cmd_record := ((x"81", x"04", x"00", x"04", x"00", others => x"00"), 5);
constant c_set_divide : cmd_record := ((x"80", x"08", x"00", x"00", x"00", others => x"00"), 5);
constant c_set_flags : cmd_record := ((x"82", x"08", x"00", x"00", x"00", others => x"00"), 5);
```

- Created top level testbench to test interfaces between each module
- After polling interval, would send UART commands to top level
- Sends the different commands to setup and arm the trigger
- Looks at waveform to confirm functionality

Top Level Testing - UART Input

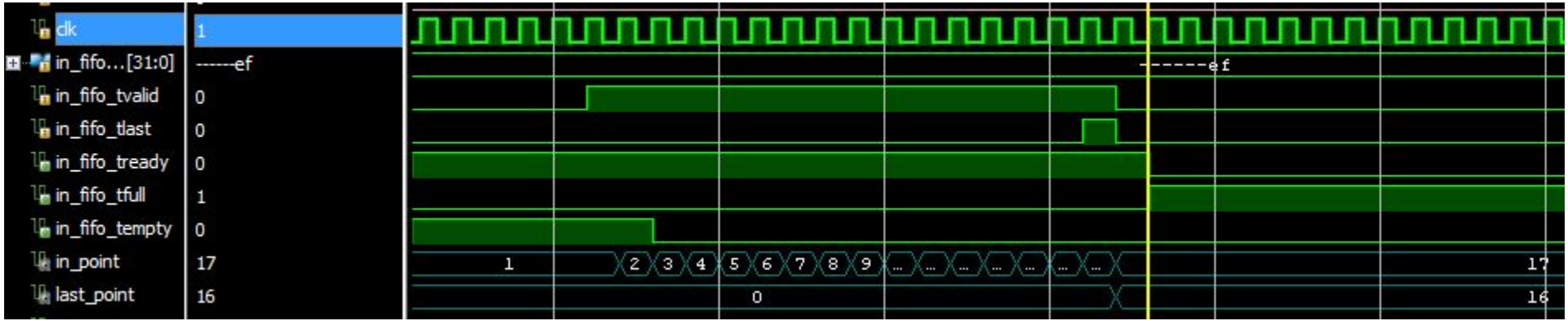


UART Receive & Message Processing



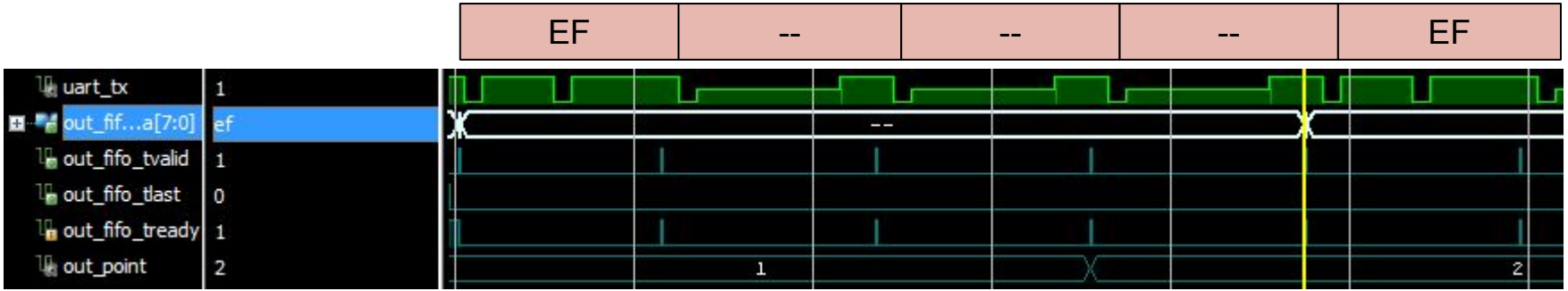
Receiving Set Trigger Mask Command

LA Control & Storage FIFO



In_point incrementing every clock cycle, storing another word

Storage FIFO to UART Transmit



Device Targeting

Digilent ZedBoard™

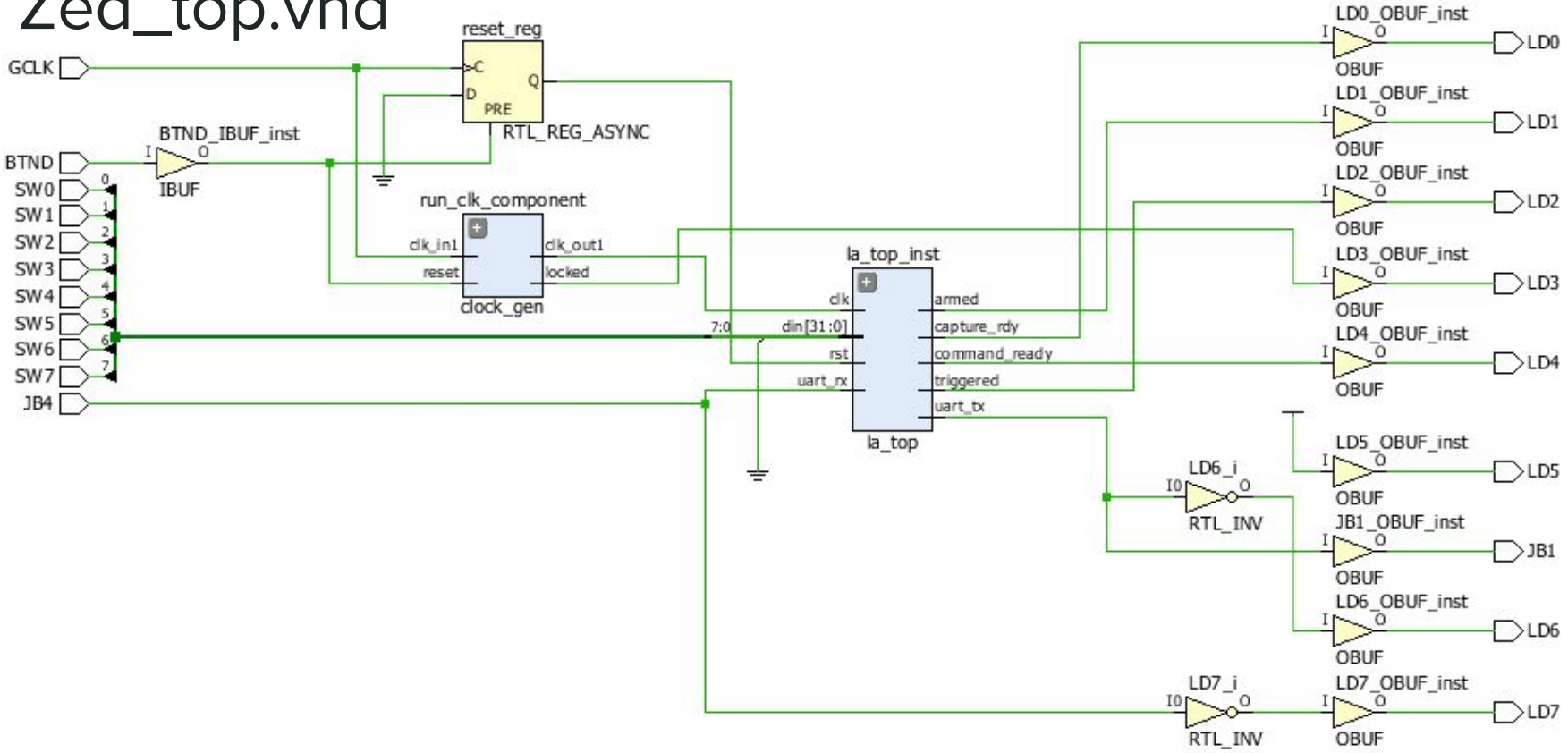
- Xilinx Zynq-7000 (XC7Z020-CLG484)
  - Dual ARM Cortex A9 + FPGA Logic
- 100 MHz Clock Source
- 8 logic accessible slide switches
- 5 logic accessible push buttons
- 8 logic accessible LEDs
- USB-UART converter
  - Not directly accessible to logic
  - Intended for processor access.
- 5 PMOD connectors (8 I/O per)
  - 1 for processor
  - 2 high-speed I/O (direct to chip)
  - 3 general purpose (in-line resistor)

Image source: digilent.com

Zed_Top Entity

```
ENTITY zed_top IS
 PORT (
  -- 100 MHz clock
   GCLK                              : in  std_logic;
    --LED Outputs
    LD0, LD1, LD2, LD3, LD4, LD5, LD6, LD7  : out std_logic;
    --Buttons
     BTND          : in  std_logic; --reset button
    --UART SIGNALS
    JB4                               : in  std_logic;  --RX
    JB1                               : out std_logic;  --TX
    --Switches
    SW7, SW6, SW5, SW4, SW3, SW2, SW1, SW0  : in  std_logic
   );

END ENTITY zed_top;
```

Zed_top.vhd


GCLK
reset_reg
C
D
Q
PRE
RTL_REG_ASYNC
BTND_IBUF_inst
I
O
IBUF
BTND
SW0
SW1
SW2
SW3
SW4
SW5
SW6
SW7
JB4
run_clk_component
clk_in1
reset
clk_out1
locked
clock_gen
la_top_inst
clk
din[31:0]
rst
uart_rx
armed
capture_rdy
command_ready
triggered
uart_tx
la_top
7:0
LD0_OBUF_inst
OBUF
LD0
LD1_OBUF_inst
LD1
LD2_OBUF_inst
LD2
LD3_OBUF_inst
LD3
LD4_OBUF_inst
LD4
LD5_OBUF_inst
LD5
LD6_i
I0
RTL_INV
JB1_OBUF_inst
JB1
LD6_OBUF_inst
LD6
LD7_i
LD7_OBUF_inst
LD7

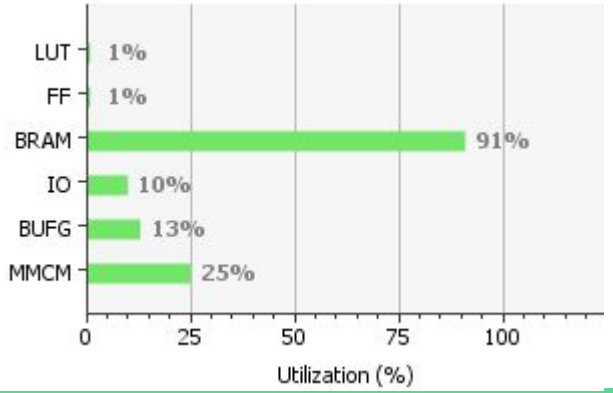
Digilent ZedBoard™ Timing & Utilization Report

Timing (100MHz sampling rate):

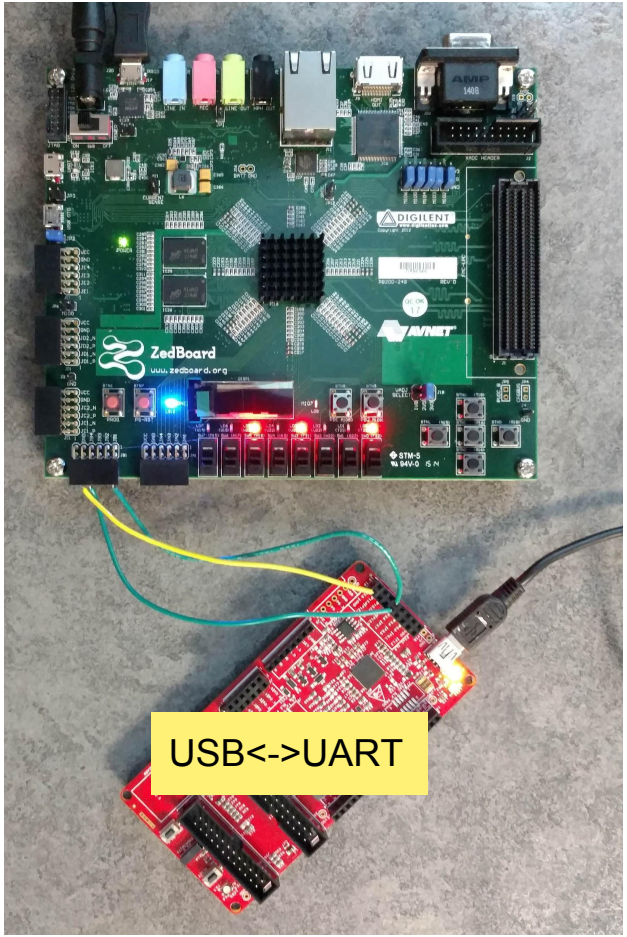
- Setup
  - WNS: 0.233 ns
  - TNS: 0 ns
  - Number of Failing Endpoints: 0
  - Total Number of Endpoints: 6045
- Hold
  - WHS: 0.02 ns
  - THS: 0 ns
  - Number of Failing Endpoints: 0
  - Total Number of Endpoints: 6045

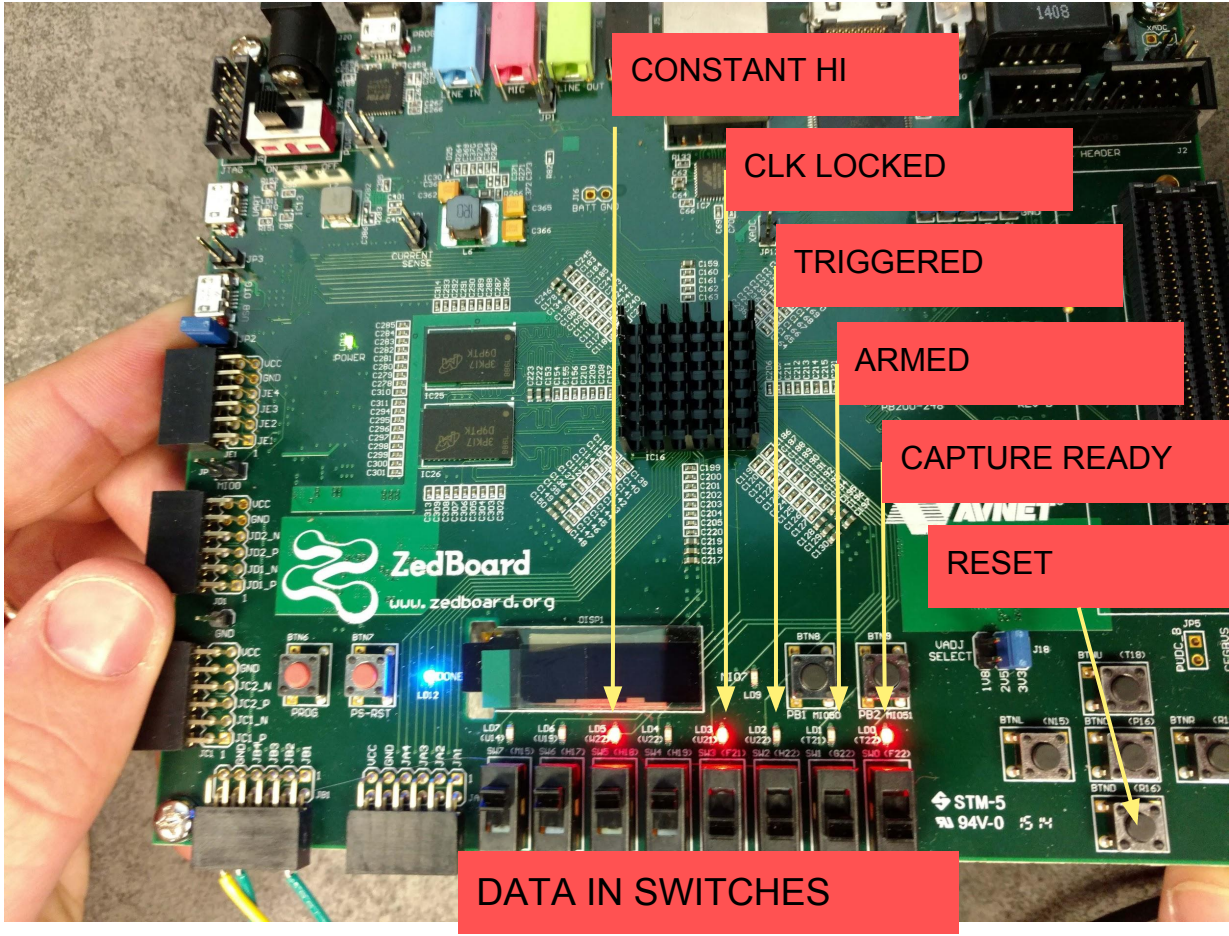
Utilization:

- LUT: 537-1%
- FF: 558-0.52%
- BRAM: 128-91% (2^17 capture depth)
- I/O: 20-10%
- BUFG: 4-12.5%
- MMCM: 1-25%

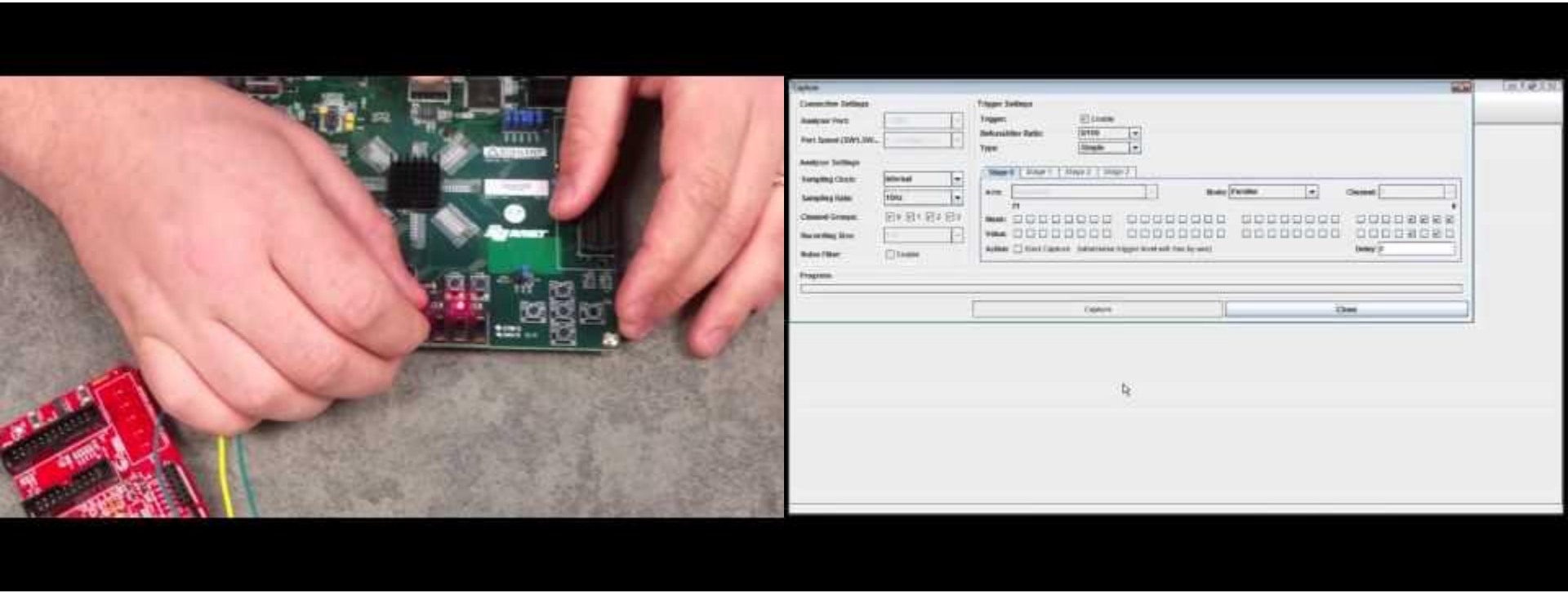


Demonstration


ZedBoard
www.zedboard.org
USB<->UART


CONSTANT HI
CLK LOCKED
TRIGGERED
ARMED
CAPTURE READY
RESET
DATA IN SWITCHES
ZedBoard
www.zedboard.org

Demo Video



Improvements

- Implement all SUMP commands
- Increase sample rate
- Increase sample depth
  - Implement external memory storage
- Increase data download rate
- Convert to IP.

Questions?

Project available on
https://github.com/ashtonchase/logic_analyzer

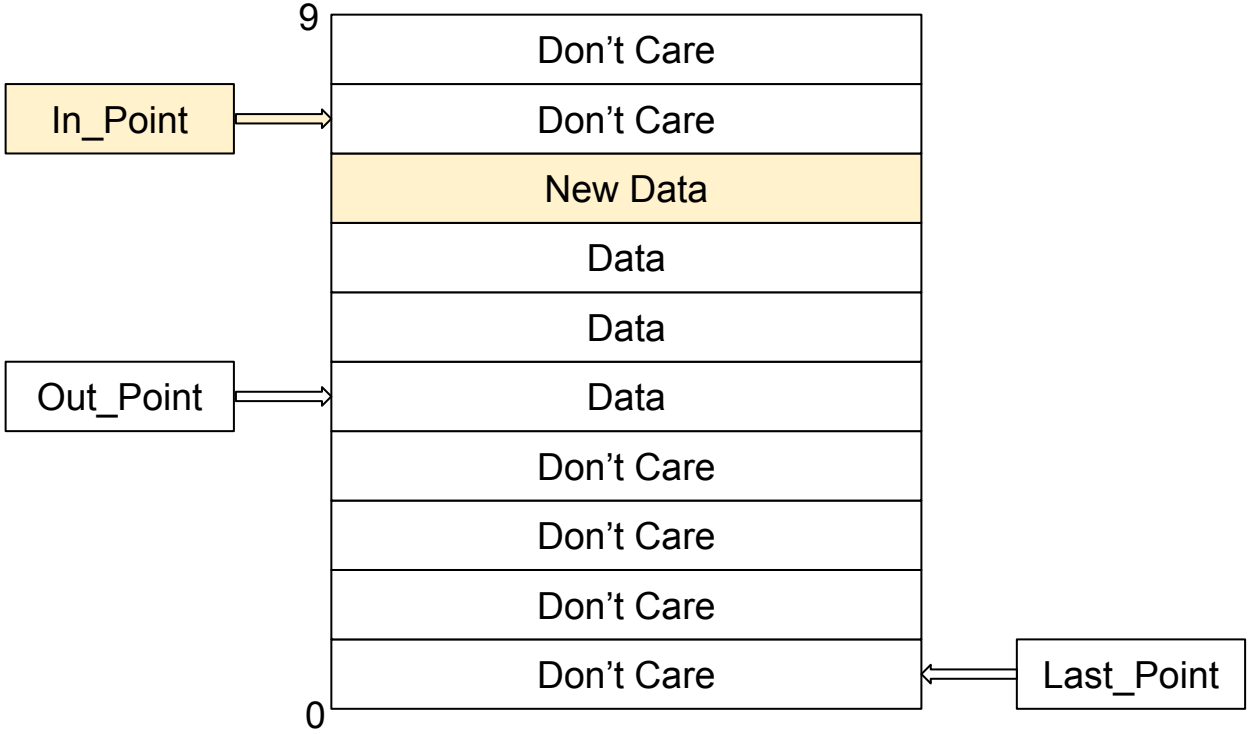
Backup Slides

Digilent Zybo™ Board

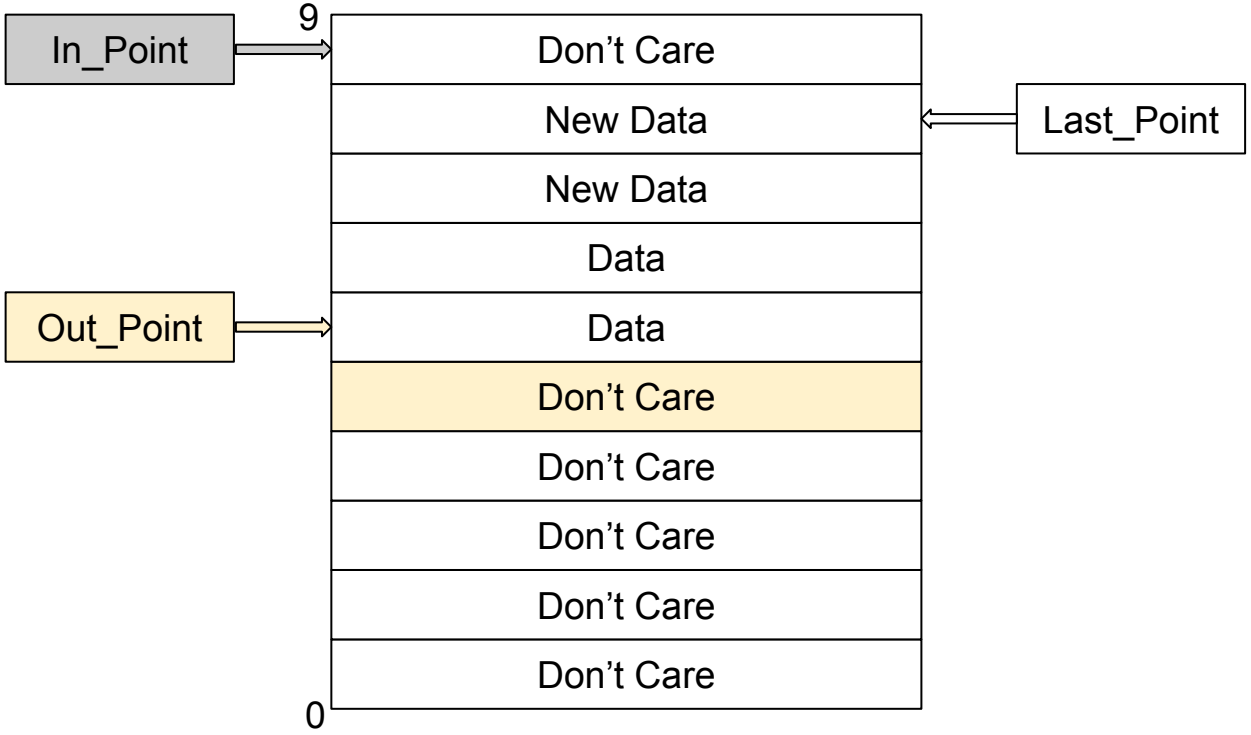
- Xilinx Zynq-7000 (XC7Z010-1CLG400C)
  - Dual ARM Cortex A9 + FPGA Logic
- 125 MHz Clock Source
- 4 logic accessible slide switches
- 4 logic accessible push buttons
- 4 logic accessible LEDs
- USB-UART converter
  - Not directly accessible to logic
  - Intended for processor access.
- 6 PMOD connectors (8 I/O per)
  - 1 for processor
  - 4 high-speed I/O (direct to chip)
  - 1 general purpose (in-line resistor)

Image source: digilent.com

FIFO Design

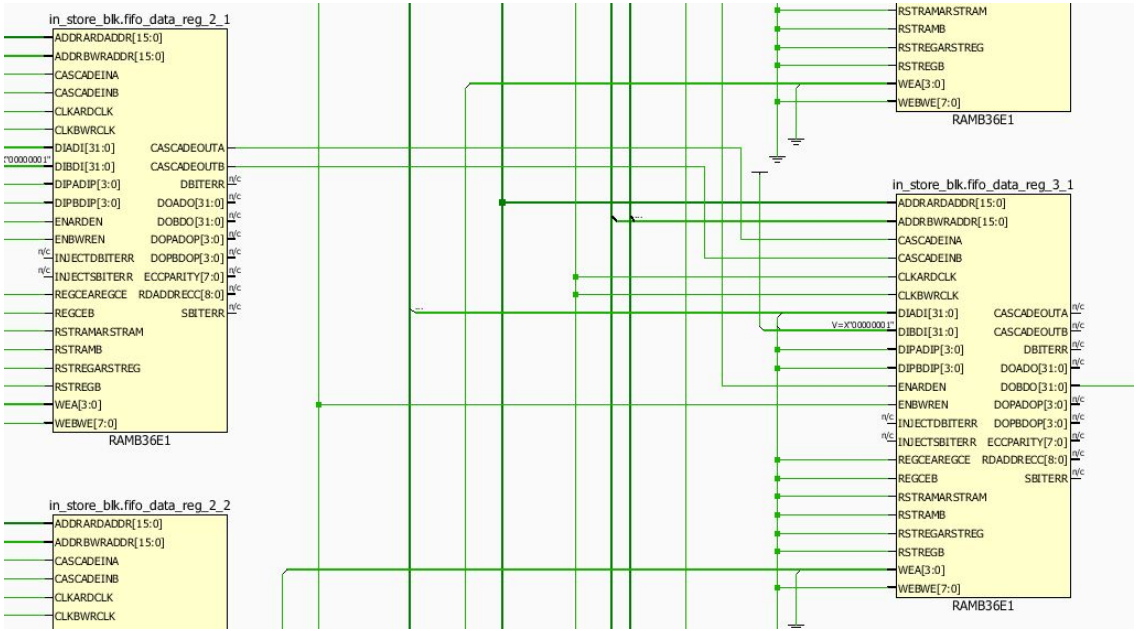

9
Don’t Care
In_Point
Don’t Care
New Data
Data
Data
Out_Point
Data
Don’t Care
Don’t Care
Don’t Care
Don’t Care
Last_Point
0

FIFO Design


In_Point
Out_Point
Last_Point
9
0
Don’t Care
New Data
New Data
Data
Data
Don’t Care
Don’t Care
Don’t Care
Don’t Care
Don’t Care

Synthesized FIFO

- Synthesized storage implemented RAM blocks
- Two RAMs cascaded together to increase depth
- Cascaded RAMs would store one bit of the input word



Testbench AXI Output