Debug Governor

with AXI-Lite

By Isamu Poy

Index

1.0 Introduction

Computer hardware, particularly FPGAs, are growing in popularity as data centres and machine learning becomes more computationally demanding. Ironically, there is little software support for the tools that allow for hardware design, and developers consequently have to endure countless hours debugging trivial discrepancies. The tools required to offer a software-like IDE experience still remain unsupported on CAD platforms (i.e. Vivado and Quartus) unlike software supported with built in tools, such as Netbeans and Eclipse. This issue presents the need for a new debugging tool that implements a backend for the fundamental building blocks of a debugger, a Debug Governor.

1.1 Past Work

The original version of the Debug Governor supports AXI-Stream interfaces, and can be controlled to carry out several functions: Pause, Log, Drop, and Inject.

- **Pause**: Put a transaction on hold
- **Log**: Log data from transaction
- **Drop**: Cancel a transaction and move on
- **Inject**: Forcibly add new data on a stream

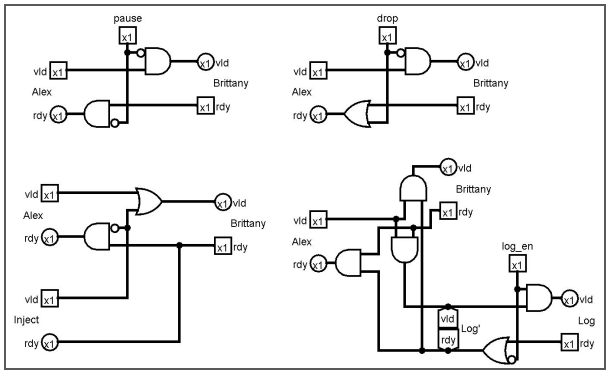

Figure 1: logic circuits of Pause, Log, Drop, and Inject

In AXI-Stream, transactions occur on data streams connecting a Master to a Slave. AXI-Stream follows a protocol in which transactions occur upon a handshake: when either the Master or Slave sends out VALID and READY signals. It is the role of the Debug Governor to manipulate the handshaking protocol to implement the four functions mentioned above.

The Debug Governor contains a high-level component that manages the command input, and low level components known as handshake governors. The handshake governors use the Pause, Log,

Drop, and Inject signals to control the logic in the following figure to manipulate the AXI Streams.

2.0 Motivation

AXI Stream can be limiting. The protocol does not provide any support for addressing, which can be problematic for applications that use memory. Arguably, this implementation is trivial given the current iteration of the Debug Governor. It would seem intuitive to add an AXI Stream debug governor on each stream of the AXI Lite protocol, and send in commands. Interestingly, it fails to take into account the discrepancies between the AXI Lite and AXI Stream protocols.

While it is still possible to implement an AXI Lite Debug Governor with five AXI Stream governors in theory, it will be practically impossible to use. The user must manually handle the response signals and manually input every piece of information needed which occurs in nanoseconds.

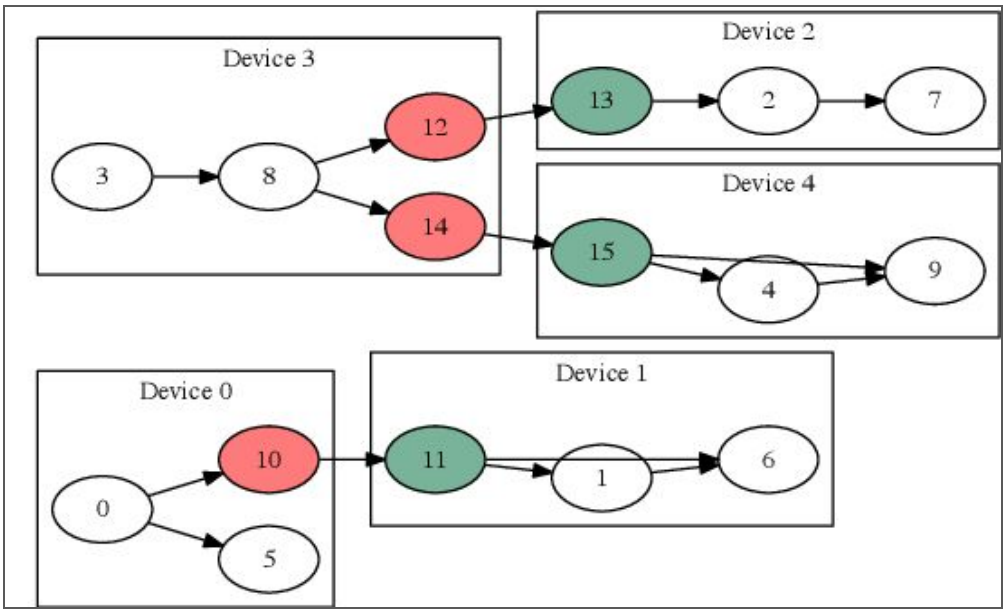

Figure 2: Debug Governors "pilot" a single communication stream

2.1 Transition to AXI-Lite

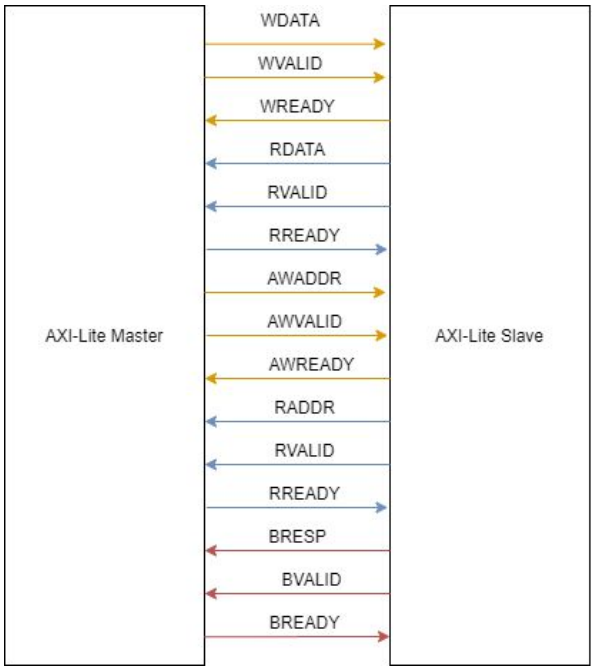

Figure 3: AXI-Lite Interface

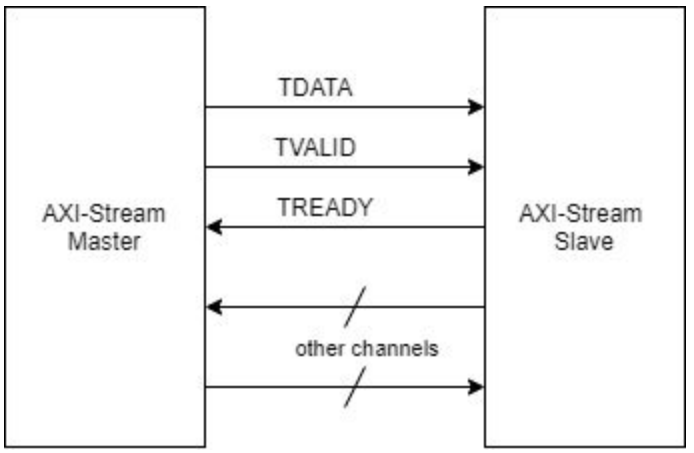

Figure 4: AXI-Stream Interface

The AXI-Stream protocol contains 3 main channels: TDATA for sent data, TVALID a one directional handshake signal from the master, and TREADY a one directional handshake signal from the slave. Information is only sent when TVALID and TREADY are both high, otherwise nothing occurs.

The AXI-Lite protocol contains multiple AXI-Stream interfaces to allow for memory-mapped transactions. The protocol contains the following: WDATA for write data, AWADDR for an address associated with written data, RDATA for read data, RADDR for an address associated with read data, and BRESP as an acknowledgement signal. Each data stream has a respective address stream, and both pieces of information must be provided. Additionally, when a write transaction successfully sends data to a slave, a BRESP signal is sent to the master.

In order to transition from AXI-Stream to AXI-Lite, the debug governor must consider corner cases in implementing injecting to WDATA/RDATA and rudimentary write/read transactions.

1. For each WDATA and RDATA being, there must be AWADDR and ARADDR
2. For injecting into WDATA, BRESP must immediately be dropped with the drop command, so it never reaches the master. Otherwise this will violate the protocol.
3. For injecting into RDATA, one must consider supplying ARADDR, and dropping ARADDR before it reaches the slave.

2.2 Redesign of Hardware

Debug Governor for AXI-Lite implemented the hardware using an FSM in Verilog HDL. However, Marco, the author of the original Debug Governor, mentions that it is “Spaghetti Code” due to its complexity. It is difficult to understand for a developer with no prior experience with Debug Governors because everything is done inside a single module, with the exception of the handshake governors. I propose a modularized approach that will make the system easier to understand and less cumbersome to debug (see 3.0 Implementation).

3.0 Implementation

The handshake governors are an important component in the Debug Governor for AXI Lite. Each stream on the AXI Lite protocol contains an Axis Governor to carry out the four functions of the original Debug Governor. The objective was to condense everything into one Intellectual Property (IP) core and implement an interface that takes in a command to perform a function on any of the AXI Streams. The design was divided into three components: a top-level arbiter for command input, a control path, and a datapath.

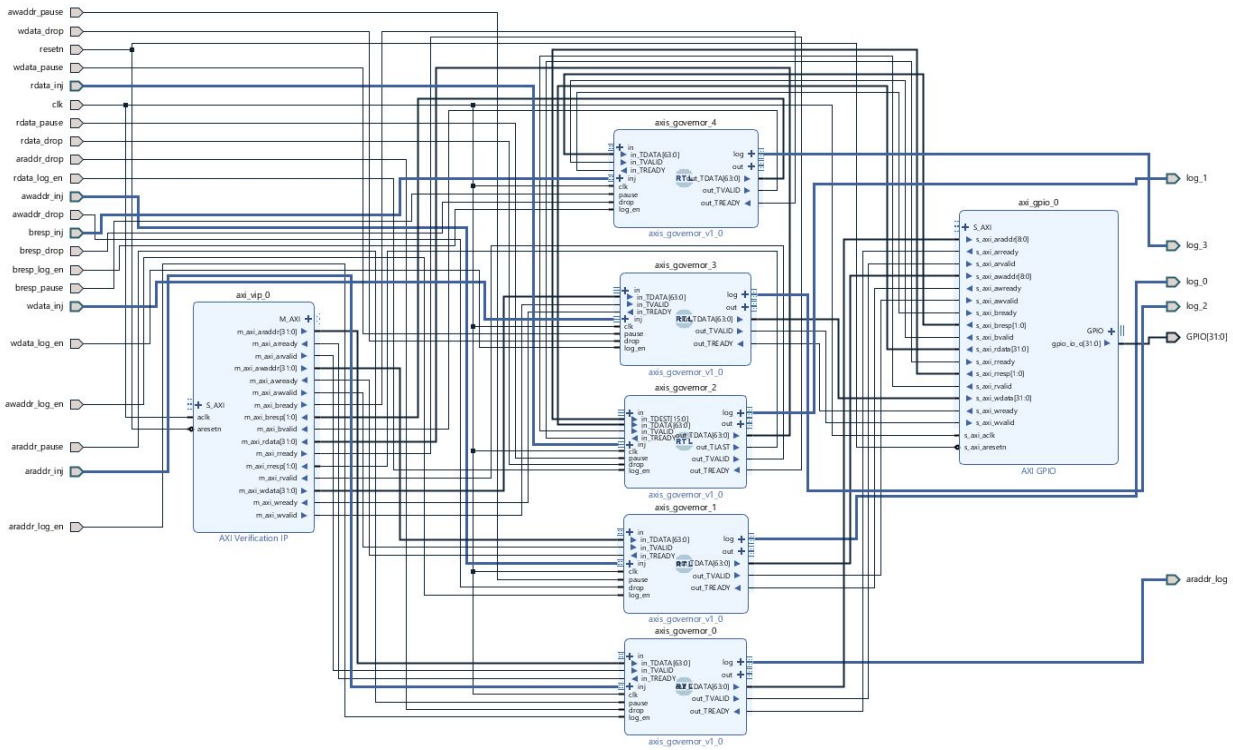

Figure 5: Master-Slave interface containing handshake governors in the middle column

3.1 Top-Level Design

The top level module takes in a command input that is structured as an operation code (OP code). Part of the OP code consists of an encoded command on a certain stream, and the remaining bits correspond to data for write data.

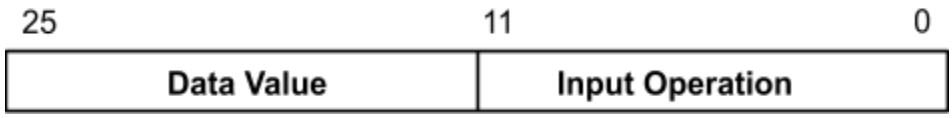

Figure 6: Operational code for Debug Governor

The input command also follows the handshaking protocol, and will only be registered when both VALID and READY are received. The Input operation is decoded inside the control path, and the Finite State Machine (FSM) iterates through several states in order to satisfy the AXI Lite Protocol. Each state has a different behaviour and controls a different wire on one of the handshake governors, which control the streams on AXI-Lite. For example, by definition, a save will send a BRESP DATA to the master when a write to WDATA is successful. Inside the inject state for the WDATA stream, we set inject enable wire high, connected to a handshake governor. This causes the handshake governor to set VALID high and DATA 0 for BRESP. This is done such that the master will not unexpectedly get a success signal BRESP, after injection.
(see Appendix C)

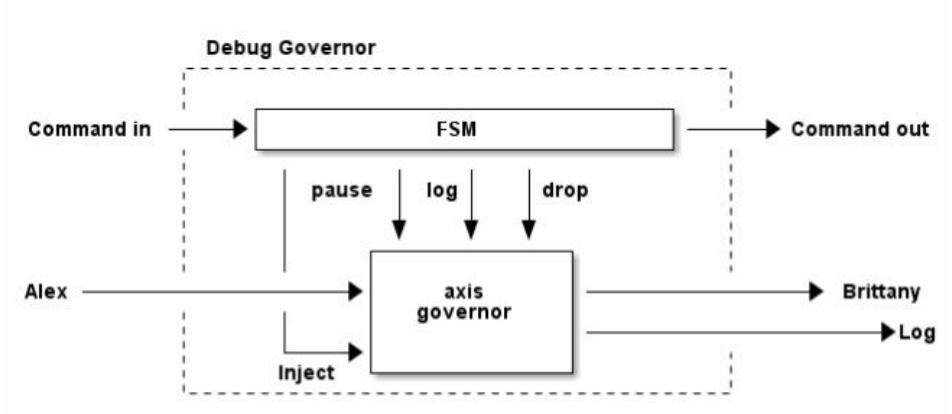

Figure 7: High level architecture of previous Debug Governor

3.2 Datapath

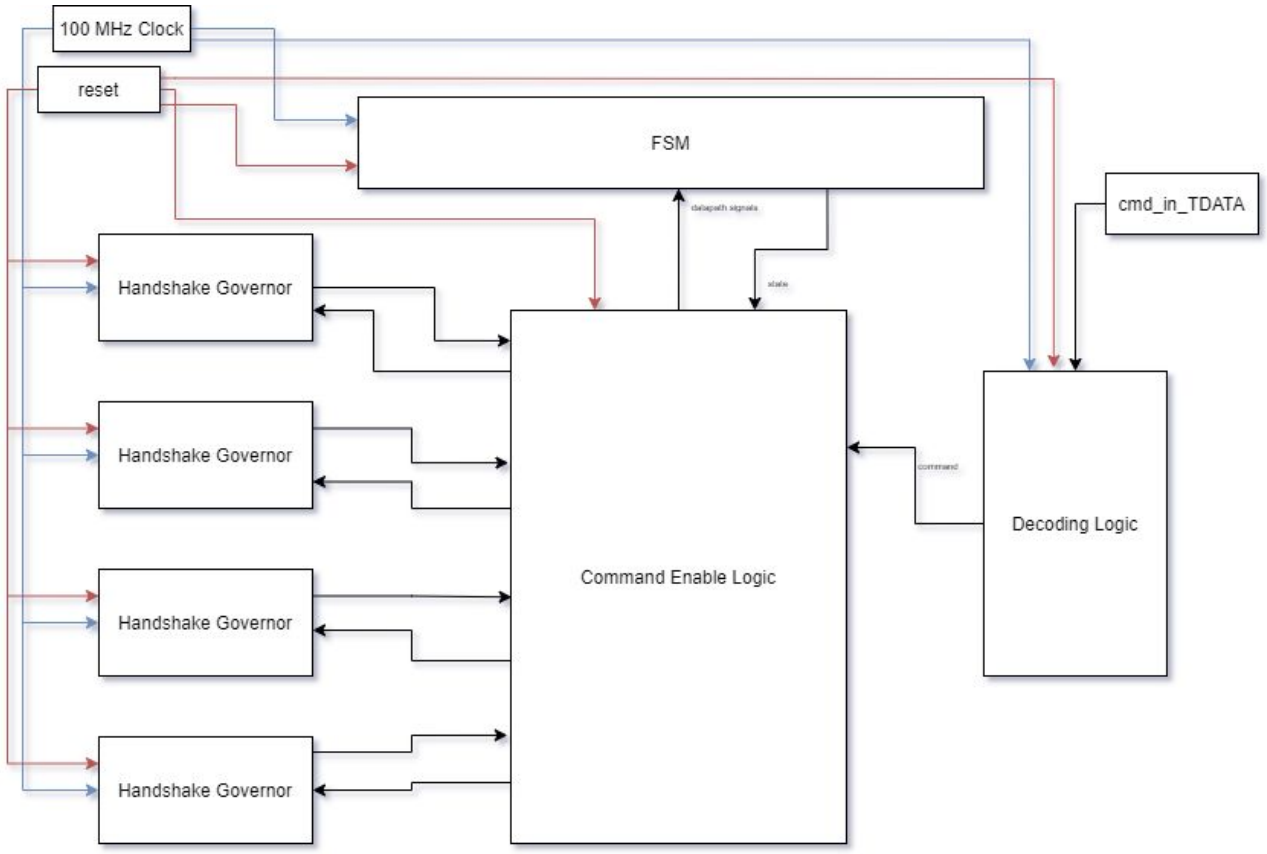

Figure 8: Datapath

(see Appendix B)

3.3 Control path

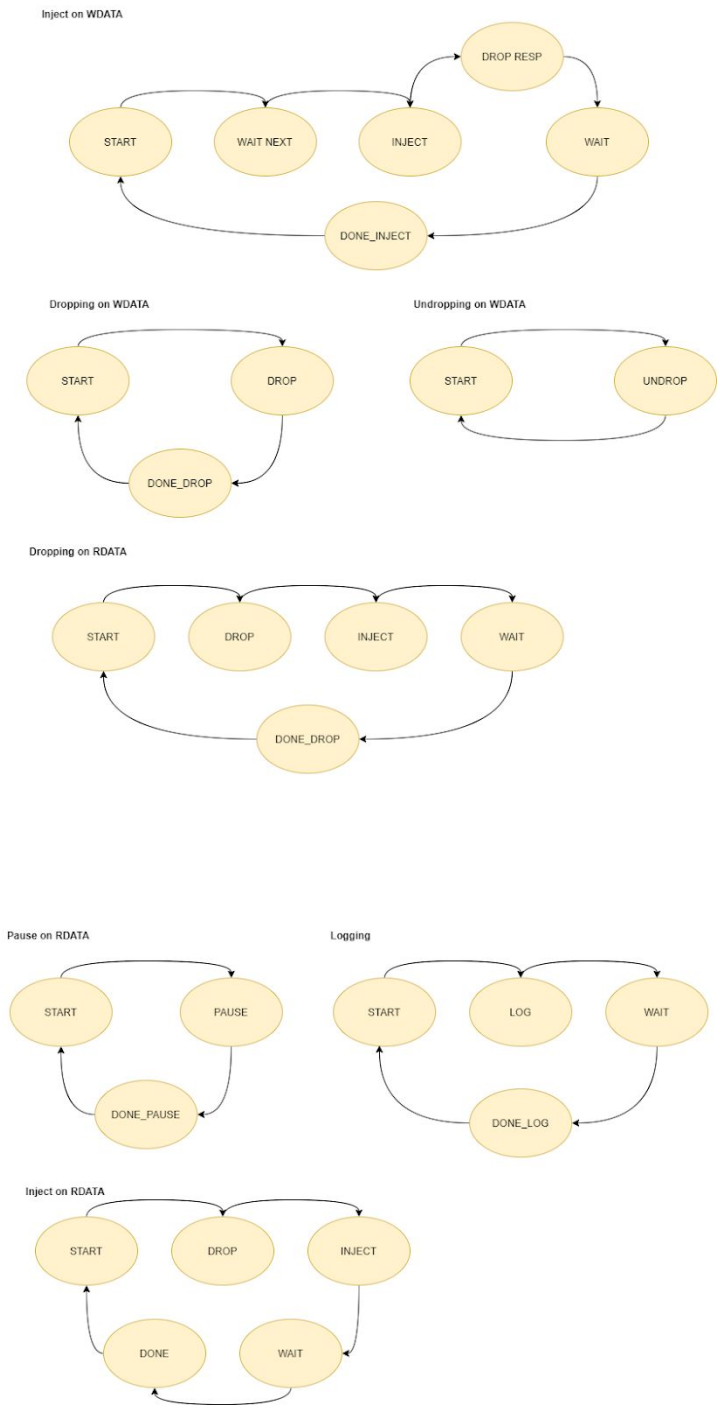

Figure 9: ControlPath

In total, the control path contains 12 states: START, DROP, INJECT, LOG, PAUSE, DONE_DROP, DONE_LOG, DONE_INJECT, DONE_PAUSE, WAIT_NEXT, UNPAUSE, and QUIT_DROP.

Wait states are used to pause the FSM until handshaking occurs, and done states are provided to reset signals back to low such as drop enable and inject enable. Without done states, the system will continue to trigger debugging actions forever.

The most complex function was found to be injecting on write data because it involved the most states. Three streams had to be monitored: BRESP, Write data, and Write Address. Thus, injecting on write data was decided to be a measure of success for this project. (see Appendix A)

3.4 Block Design

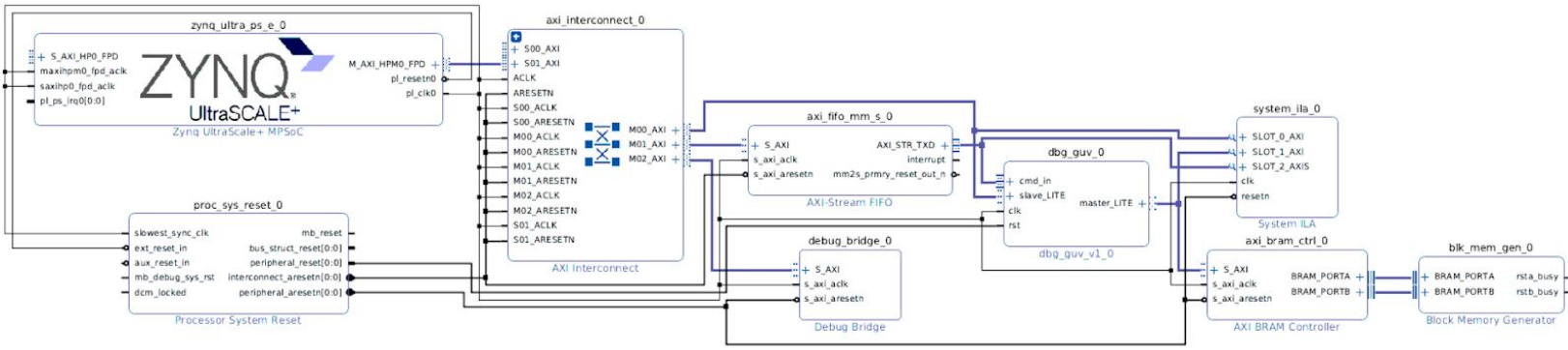

Figure 10: Debug Governor block design

The Debug Governor interface contains the following IPs:

1. Zync Ultrascale+ MPSoC
2. Processor System reset
3. AXI-Interconnect
4. AXI FIFO
5. Debug Bridge
6. Debug Governor
7. System ILA (for debugging purposes)
8. AXI BRAM
9. Block Mempory Generator

4.0 Report of Success

The Debug Governor was successfully packaged into a single IP core, and was used to create a block design on Vivado which utilized a BRAM and MPSoC FPGA. For measures of success, testbenches were constructed on Vivado to verify the IP's correct behaviour. A testbench to observe injecting on write data was successful in demonstrating that it will satisfy AXI Lite protocol while artificially placing new data on the write data stream. Additional testing was also performed using ModelSim to verify the correct state transitions of each operation.

For a final test, the generated bitstream was converted into a .bin file to be programmed on the MPSoC via XVC server and ssh tunneling to Vivado's Hardware Manager. In observing that inject WDATA, pause, and log were able to be carried out, it showed that the debug governor

functioned properly. We do not mention operations on RDATA because although the operations worked, it was not practical to be used for debugging. I left it as a basic building block for further development.

4.1 Testbenches

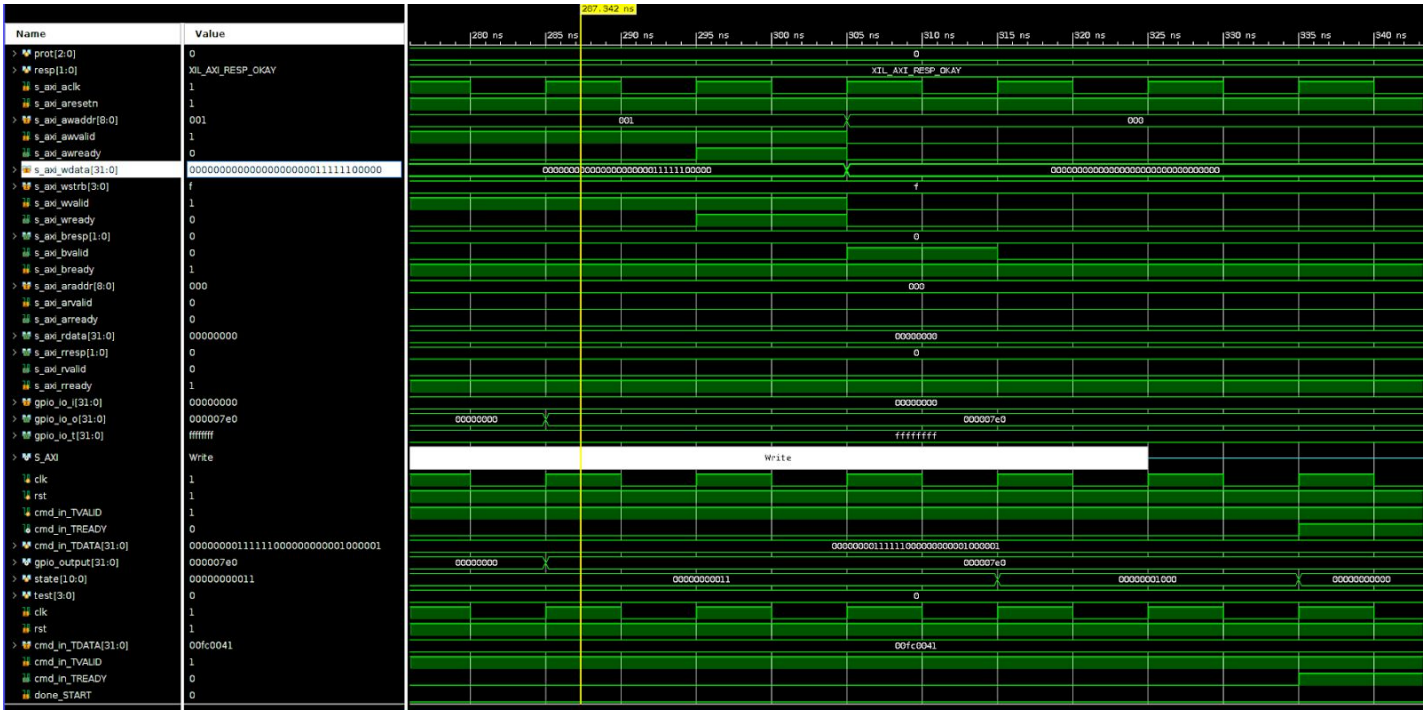

Figure 11: Injecting on WDATA (with Vivado)

*For testbenching on Vivado, a block diagram was made using an AXI-VIP master and AXI-GPIO slave

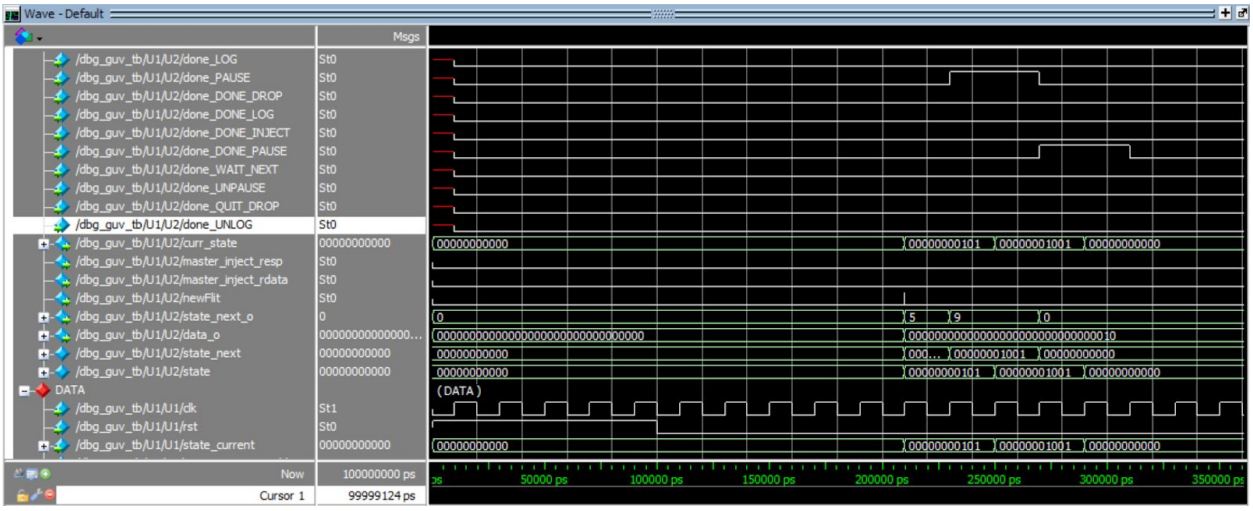

Figure 12: Pausing on RDATA (with ModelSim)

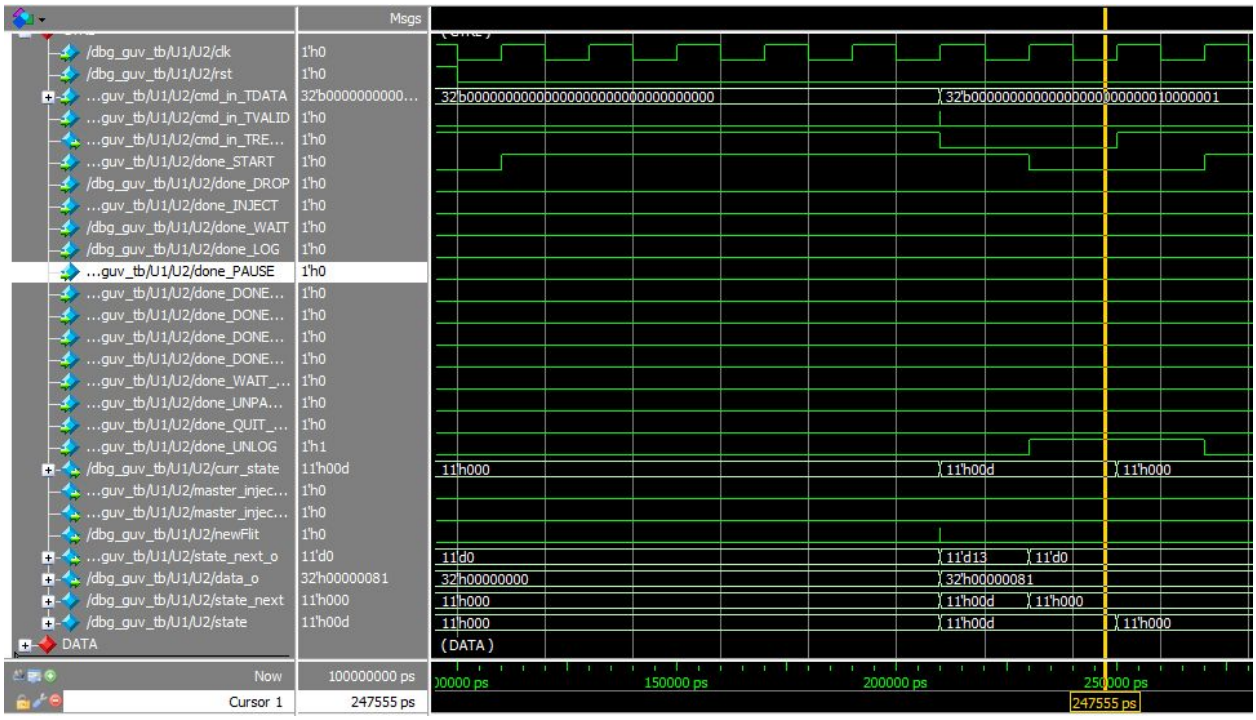

Figure 13: Logging on RDATA (with ModelSim)

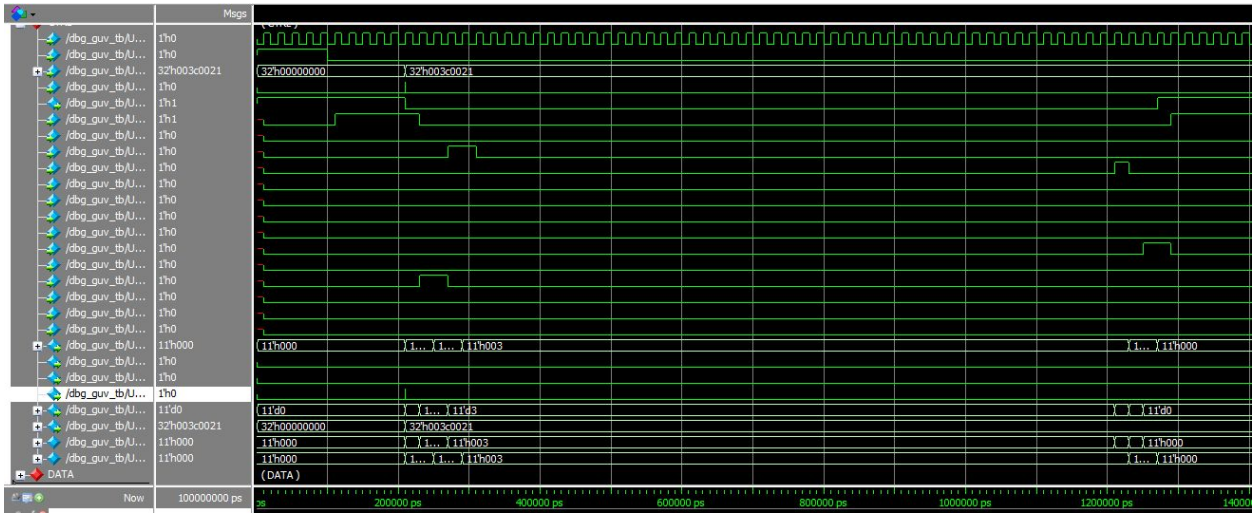

Figure 14: Inject on RDATA (with ModelSim)

4.2 Test on MPSoC

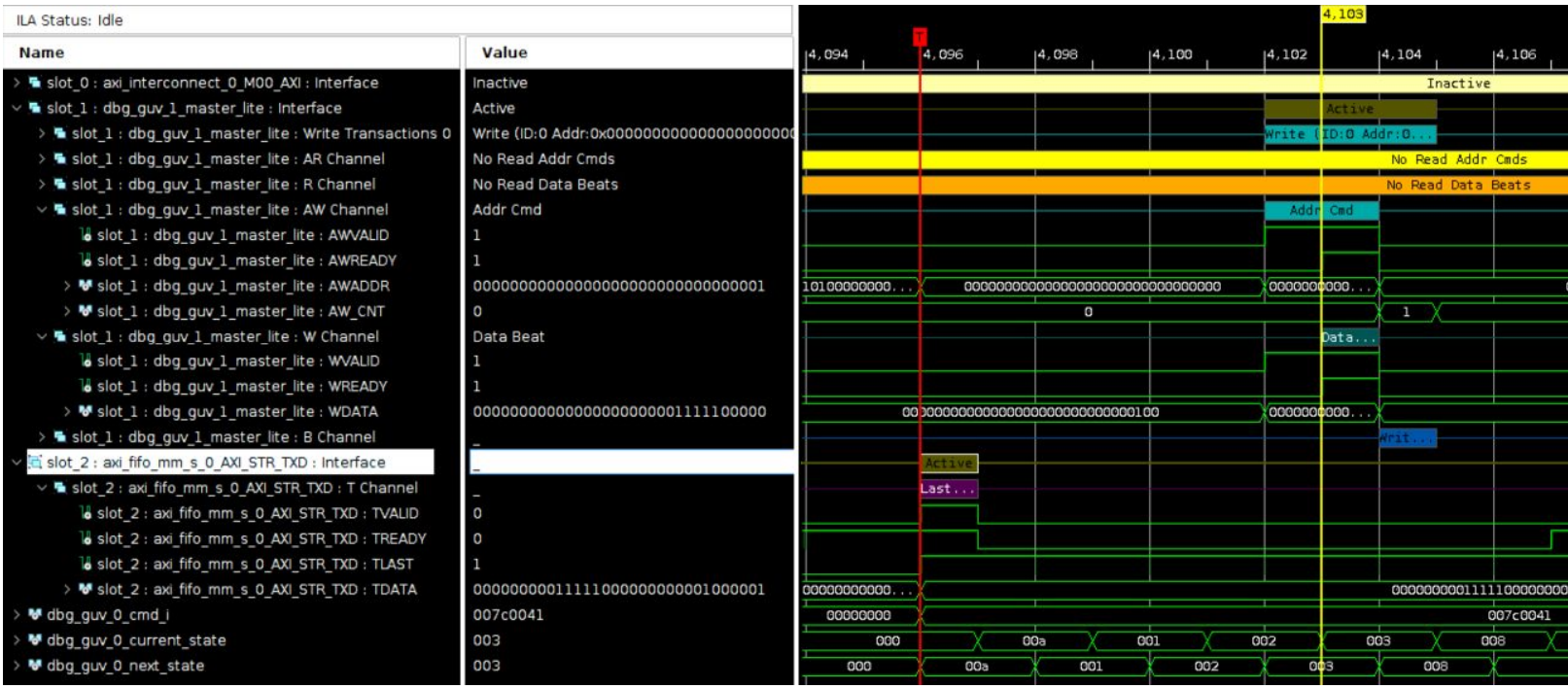

Figure 15: Sending test data for injecting on WDATA

Criteria for success for inject WDATA:

1. WDATA was updated with the proper data value from the op code (cmd_in_TDATA)

2. AWADDR was simultaneously updated with a new address value for WDATA

3. No red flags were raised by the MPSoC, which proved that BRESP was dropped before it could reach the master

4. VALID and READY signals went high for both WDATA and AWADDR when inject command was sent in

5. In addition to inject, this also tests the functionality of dropping. Inject and drop are the more complicated functions of the 4 available

5.0 Summer Timeline

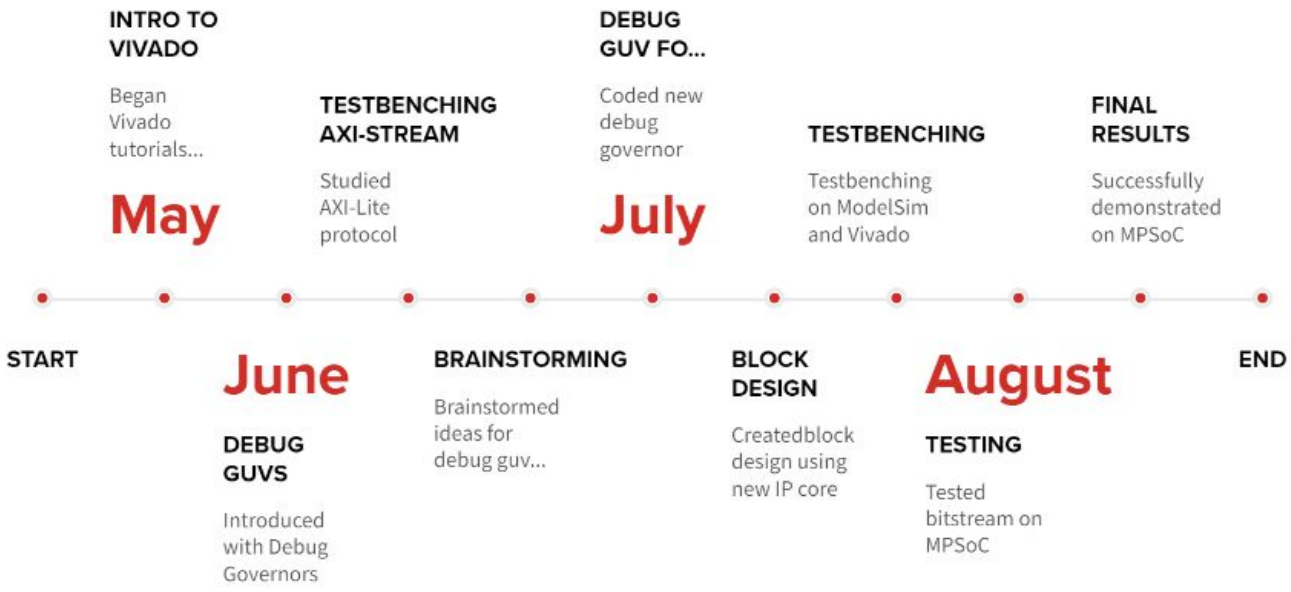

Figure 16: Progress timeline over 3-4 months

6.0 Conclusion and Future Work

The new iteration of the Debug Governor presents a novel backend framework for AXI-Lite systems i.e. allowing for memory-mapped transactions. In future work, the design may be optimized via pipelining techniques for more efficient performance, and further state optimizations may be made to reduce code complexity. A developer may use this code to develop more practical functions that resemble software debugging tools such as single-stepping.

Finally, the current design needs improvement on ease of use. Since users with no background on Debug Governors should be able to use the tool, an intuitive user interface should be designed to interact between the FPGA and Debug Governor. It is important for the user to easily observe that the debug governor is communicating correctly with the hardware.

Link to Github: https://github.com/isamumu/dbg_guv.git

Appendix A: Controlpath code

```
`timescale 1ns / 1ps

////////////////////////////////////////////////////
////////// Step 1: Make Control  FSM ///////////////
////////////////////////////////////////////////////

module control_FSM(

    input logic clk,
    input logic rst,

    //Input command stream

    input wire [31:0] cmd_in_TDATA,
    input logic cmd_in_TVALID,
    output logic cmd_in_TREADY,

    input wire done_START,
    input wire done_DROP,
    input wire done_INJECT,
    input wire done_WAIT,
    input wire done_LOG,
    input wire done_PAUSE,
    input wire done_DONE_DROP,
    input wire done_DONE_LOG,
    input wire done_DONE_INJECT,
    input wire done_DONE_PAUSE,
    input wire done_WAIT_NEXT,
    input wire done_UNPAUSE,
    input wire done_QUIT_DROP,
    input wire done_UNLOG,

    output wire [10:0] curr_state,
    output wire master_inject_resp,
    output wire master_inject_rdata,

    output wire newFlit,
    output logic [10:0] state_next_o,
    output logic [31:0] data_o
```

```
);

    localparam START = 6'd0;
    localparam DROP = 6'd1;
    localparam INJECT = 6'd2;
    localparam WAIT = 6'd3;
    localparam LOG= 6'd4;
    localparam PAUSE = 6'd5;
    localparam DONE_DROP = 6'd6;
    localparam DONE_LOG = 6'd7;
    localparam DONE_INJECT = 6'd8;
    localparam DONE_PAUSE = 6'd9;
    localparam WAIT_NEXT = 6'd10;
    localparam UNPAUSE = 6'd11;
    localparam QUIT_DROP = 6'd12;
    localparam UNLOG = 6'd13;

    logic [10:0] state_next = 0; // to be determined in the state, not reset
    logic [10:0] state = 0;
    //logic [31:0] data = cmd_in_TDATA;

    assign data_o = cmd_in_TDATA;
    assign state_next_o = state_next;

    // we don't want to use this below
    assign master_inject_resp = cmd_in_TDATA[4]; // just make sure we have these
    assign master_inject_rdata = 0; //data[3];

    // will this throw an error if state doesnt have a value?
    assign curr_state = state;
    assign newFlit = cmd_in_TREADY && cmd_in_TVALID;

    always@(posedge clk or posedge rst) begin
        if(rst)
            state <= START;
        else
            state <= state_next;
    end

    always_comb begin

        // cmd_in_TREADY = 0; // avoid a latch
```

```
        case (state)
            START: begin

                cmd_in_TREADY = 1; // show the user that we are READY for an input
command
                                   // has to be here bc, command should only update as
we stall in the start state
                if(cmd_in_TREADY && cmd_in_TVALID) begin

                    if(((cmd_in_TDATA[1] && !cmd_in_TDATA[0]) || (cmd_in_TDATA[2] &&
!cmd_in_TDATA[0])) && done_START)
                        state_next = PAUSE; // pause rdata // wdata

                    else if(cmd_in_TDATA[3] && cmd_in_TDATA[0] && done_START)
                        state_next = QUIT_DROP;

                    else if((cmd_in_TDATA[3] || cmd_in_TDATA[4]) && done_START)
                        state_next = DROP; // drop rdata // wdata

                    else if((cmd_in_TDATA[5] || cmd_in_TDATA[6]) && done_START)
                        state_next = WAIT_NEXT;// inj rdata // inj wdata

                    else if(cmd_in_TDATA[12] && done_START)
                        state_next = INJECT;  // resp

                    else if(((cmd_in_TDATA[7] && !cmd_in_TDATA[0]) || (cmd_in_TDATA[8]
&& !cmd_in_TDATA[0]) || (cmd_in_TDATA[9] && !cmd_in_TDATA[0]) || (cmd_in_TDATA[10] &&
!cmd_in_TDATA[0]) || (cmd_in_TDATA[11] && !cmd_in_TDATA[0])) && done_START)
                        state_next = LOG; // rdata // wdata // raddr // awaddr // resp

                    else if(((cmd_in_TDATA[7] && cmd_in_TDATA[0]) || (cmd_in_TDATA[8]
&& cmd_in_TDATA[0]) || (cmd_in_TDATA[9] && cmd_in_TDATA[0]) || (cmd_in_TDATA[10] &&
cmd_in_TDATA[0]) || (cmd_in_TDATA[11] && cmd_in_TDATA[0])) && done_START)
                        state_next = UNLOG; // rdata // wdata // raddr // awaddr //
resp

                    else if((cmd_in_TDATA[1] && cmd_in_TDATA[0]) || (cmd_in_TDATA[2]
&& cmd_in_TDATA[0]) && done_START)
                        state_next = UNPAUSE; // unpause rdata // wdata

                    else
                        state_next = START;
                end
```

```
        else
            state_next = START;

    end


    DROP: begin

        if(cmd_in_TDATA[3] && done_DROP)
            state_next = DONE_DROP; // rdata

        else if(cmd_in_TDATA[4] && done_DROP)
            state_next = INJECT; // wdata

        else if(cmd_in_TDATA[6] && done_DROP)
            state_next = INJECT; // inj wdata

        else
            state_next = DROP;

        cmd_in_TREADY = 0;

    end

    QUIT_DROP: begin

        state_next = START;
        cmd_in_TREADY = 0;

    end

    INJECT: begin

        if(cmd_in_TDATA[3] && done_INJECT)
            state_next = WAIT; // for drop rdata

        else if(cmd_in_TDATA[4] && done_INJECT)
            state_next = DROP; // for drop wdata

        else if(cmd_in_TDATA[5] && done_INJECT)
            state_next = WAIT; // for inject rdata
```

```
                else if(cmd_in_TDATA[6] && done_INJECT)
                    state_next = WAIT; // for inject wdata

                else if(cmd_in_TDATA[12] && done_INJECT)
                    state_next = WAIT; // inject resp

                else
                    state_next = INJECT;

                cmd_in_TREADY = 0;

            end

            WAIT: begin

                if((cmd_in_TDATA[3] || cmd_in_TDATA[4]) && done_WAIT)
                    state_next = DONE_DROP; // drop rdata // wdata

                else if((cmd_in_TDATA[5] || cmd_in_TDATA[12]) && done_WAIT)
                    state_next = DONE_INJECT;// inj rdata // inj resp

                else if(cmd_in_TDATA[6] && done_WAIT)
                    state_next = DONE_INJECT; // inj wdata

                /*
                else if((cmd_in_TDATA[7] || cmd_in_TDATA[8] || cmd_in_TDATA[9] ||
cmd_in_TDATA[10] || cmd_in_TDATA[11]) && done_WAIT)
                    state_next = DONE_LOG; // rdata // wdata // raddr // awaddr //
resp
                */
                else
                    state_next = WAIT;

                cmd_in_TREADY = 0;

            end

            WAIT_NEXT: begin

                if(cmd_in_TDATA[6] && done_WAIT_NEXT)
                    state_next = DROP; // inj wdata
                else if(cmd_in_TDATA[5] && done_WAIT_NEXT)
                    state_next = INJECT; // inj rdata
```

```
            else
                state_next = WAIT_NEXT;

            cmd_in_TREADY = 0;
        end


        LOG: begin

            if(done_LOG)
                state_next = WAIT;
            else
                state_next = LOG;

            cmd_in_TREADY = 0;

        end

        UNLOG: begin
            if(done_UNLOG)
                state_next = START;
            else
                state_next = UNLOG;

            cmd_in_TREADY = 0;
        end

        PAUSE: begin

            if(done_PAUSE)
                state_next = DONE_PAUSE;
            else
                state_next = PAUSE;

            cmd_in_TREADY = 0;

        end

        UNPAUSE: begin

            if(done_UNPAUSE)
                state_next = START;
            else
```

```
                state_next = UNPAUSE;

            cmd_in_TREADY = 0;

        end

        DONE_DROP: begin

            if(done_DONE_DROP)
                state_next = START;

            else
                state_next = DONE_DROP;

            cmd_in_TREADY = 0;

        end

        DONE_LOG: begin

            if(done_DONE_LOG)
                state_next = START;

            else
                state_next = DONE_LOG;

            cmd_in_TREADY = 0;

        end

        DONE_INJECT: begin

            if(done_DONE_INJECT)
                state_next = START;

            else
                state_next = DONE_INJECT;

            cmd_in_TREADY = 0;

        end

        DONE_PAUSE: begin
```

```
                if(done_DONE_PAUSE)
                    state_next = START;

                else
                    state_next = DONE_PAUSE;

                cmd_in_TREADY = 0;

            end

            default: begin
                cmd_in_TREADY = 0;
                state_next = START;
            end

        endcase
    end
endmodule
```

Appendix B: Datapath code

```
`timescale 1ns / 1ps
`include "axis_governor.sv"

`define SAFE_ID_WIDTH (ID_WIDTH < 1 ? 1 : ID_WIDTH)
`define SAFE_DEST_WIDTH (DEST_WIDTH < 1 ? 1 : DEST_WIDTH)

// direction of wires: input (going into the axis guv) and output (going out of axis
guv)

module datapath # (

    // will move this to dbg_guv module later
    parameter DATA_WIDTH = 64,
    parameter DEST_WIDTH = 16,
    parameter ID_WIDTH = 16

) (

    input logic clk,
    input logic rst,
    input [10:0] state_current,
    input wire master_inject_enable_resp,
    input wire master_inject_enable_rdata,

    //Input command stream
    input wire [31:0] cmd_in_TDATA,

    //Input AXI Stream rdata.
    input wire [DATA_WIDTH-1:0] din_TDATA_rdata,
    input wire [DEST_WIDTH -1:0] din_TDEST_rdata,
    input wire din_TVALID_rdata,
    output wire din_TREADY_rdata,

    //Input AXI Stream wdata.
    output wire [DATA_WIDTH-1:0] din_TDATA_wdata,
    output wire din_TVALID_wdata,
    input wire din_TREADY_wdata,

    //Input AXI Stream raddr.
    output wire [DATA_WIDTH-1:0] din_TDATA_raddr,
```

```
    output wire din_TVALID_raddr,
    input wire din_TREADY_raddr,

    //Input AXI Stream awaddr.
    output wire [DATA_WIDTH-1:0] din_TDATA_awaddr,
    output wire din_TVALID_awaddr,
    input wire din_TREADY_awaddr,

    //Input AXI Stream resp.
    input wire [DATA_WIDTH-1:0] din_TDATA_resp,
    input wire din_TVALID_resp,
    output wire din_TREADY_resp,

    ////////////////////////////////////////////////////

    //Output AXI Stream rdata.
    output wire [DATA_WIDTH-1:0] dout_TDATA_rdata,
    output wire [DEST_WIDTH -1:0] dout_TDEST_rdata,
    output wire dout_TVALID_rdata,
    input wire dout_TREADY_rdata,

    //Output AXI Stream wdata.
    input wire [DATA_WIDTH-1:0] dout_TDATA_wdata,
    input wire dout_TVALID_wdata,
    output wire dout_TREADY_wdata,

    //Output AXI Stream raddr.
    input wire [DATA_WIDTH-1:0] dout_TDATA_raddr,
    input wire dout_TVALID_raddr,
    output wire dout_TREADY_raddr,

    //Output AXI Stream awaddr.
    input wire [DATA_WIDTH-1:0] dout_TDATA_awaddr,
    input wire dout_TVALID_awaddr,
    output wire dout_TREADY_awaddr,

    //Output AXI Stream resp.
    output wire [DATA_WIDTH-1:0] dout_TDATA_resp,
    output wire dout_TVALID_resp,
    input wire dout_TREADY_resp,

    // Done signals
    output logic done_START,
```

```
    output logic done_DROP,
    output logic done_INJECT,
    output logic done_WAIT,
    output logic done_LOG,
    output logic done_PAUSE,
    output logic done_DONE_DROP,
    output logic done_DONE_LOG,
    output logic done_DONE_INJECT,
    output logic done_DONE_PAUSE,
    output logic done_WAIT_NEXT,
    output logic done_UNPAUSE,
    output logic done_QUIT_DROP,
    output logic done_UNLOG,

    input wire newFlit,

    output wire [DATA_WIDTH-1:0] log_TDATA_rdata_o,
    output wire log_TVALID_rdata_o,
    input wire log_TREADY_rdata_i,
    output wire [DEST_WIDTH -1:0] log_TDEST_rdata_o,

    output wire [DATA_WIDTH-1:0] log_TDATA_wdata_o,
    output wire log_TVALID_wdata_o,
    input wire log_TREADY_wdata_i,

    output wire [DATA_WIDTH-1:0] log_TDATA_raddr_o,
    output wire log_TVALID_raddr_o,
    input wire log_TREADY_raddr_i,

    output wire [DATA_WIDTH-1:0] log_TDATA_awaddr_o,
    output wire log_TVALID_awaddr_o,
    input wire log_TREADY_awaddr_i,

    output wire [DATA_WIDTH-1:0] log_TDATA_resp_o,
    output wire log_TVALID_resp_o,
    input wire log_TREADY_resp_i


);

    ////////////////////////////
    //axis governor connections//
    ////////////////////////////
```

```
// rdata
wire [DATA_WIDTH-1:0] log_TDATA_rdata;
wire [DEST_WIDTH -1:0] log_TDEST_rdata;
wire log_TREADY_rdata;
wire log_TVALID_rdata;

wire inj_TVALID_rdata;
wire inj_TREADY_rdata;
logic [DATA_WIDTH -1:0] inj_TDATA_rdata = 0;
logic [DEST_WIDTH -1:0] inj_TDEST_rdata = 0;

// wdata
wire [DATA_WIDTH-1:0] log_TDATA_wdata;
wire log_TREADY_wdata;
wire log_TVALID_wdata;
wire inj_TVALID_wdata;
wire inj_TREADY_wdata;
logic [DATA_WIDTH -1:0] inj_TDATA_wdata = 0;

// raddr
wire [DATA_WIDTH-1:0] log_TDATA_raddr;
wire log_TREADY_raddr;
wire log_TVALID_raddr;
wire inj_TVALID_raddr;
wire inj_TREADY_raddr;
logic [DATA_WIDTH -1:0] inj_TDATA_raddr = 0;

// waddr
wire [DATA_WIDTH-1:0] log_TDATA_awaddr;
wire log_TREADY_awaddr;
wire log_TVALID_awaddr;
wire inj_TVALID_awaddr;
wire inj_TREADY_awaddr;
logic [DATA_WIDTH -1:0] inj_TDATA_awaddr;

// resp
wire [DATA_WIDTH-1:0] log_TDATA_resp;
wire log_TREADY_resp;
wire log_TVALID_resp;
wire inj_TVALID_resp;
wire inj_TREADY_resp;
logic [DATA_WIDTH -1:0] inj_TDATA_resp = 0;
```

```
assign log_TDATA_rdata_o = log_TDATA_rdata;
assign log_TVALID_rdata_o = 1;
assign log_TDEST_rdata_o = log_TDEST_rdata;

assign log_TDATA_wdata_o = log_TDATA_wdata;
assign log_TVALID_wdata_o = log_TVALID_wdata;

assign log_TDATA_raddr_o = log_TDATA_raddr;
assign log_TVALID_raddr_o = log_TVALID_raddr;

assign log_TDATA_awaddr_o = log_TDATA_awaddr;
assign log_TVALID_awaddr_o = log_TVALID_awaddr;

assign log_TDATA_resp_o = log_TDATA_resp;
assign log_TVALID_resp_o = log_TVALID_resp;

////////////////////////////
//assert operation signals //
////////////////////////////

wire pause; // will replace later with rdata subnames
wire drop;
wire log_en;

wire pause_wdata;
wire drop_wdata;
wire log_en_wdata;

wire pause_rdata;
wire drop_rdata;
wire log_en_rdata;

wire pause_raddr;
wire drop_raddr;
wire log_en_raddr;

wire pause_awaddr;
wire drop_awaddr;
wire log_en_awaddr;

wire pause_resp;
```

```
wire drop_resp;
wire log_en_resp;

////////////////
//HELPER WIRES//
////////////////

wire inj_success_rdata;
wire inj_success_wdata;
wire inj_success_raddr;
wire inj_success_awaddr;
wire inj_success_resp;

// only inject values when this is true (!inj_failed || inj_TVALID_r == 0)

wire [DEST_WIDTH -1:0] dout_TDEST_internal_rdata;

////////////////////////////////////////////////////////////////////
//Also need to treat situations when TLAST or TKEEP are not given//
////////////////////////////////////////////////////////////////////

localparam KEEP_WIDTH = DATA_WIDTH/8 -1;

/////////////////
//reg variables//
/////////////////

logic inj_TVALID_rdata_r = 0;
logic inj_TVALID_wdata_r = 0;
logic inj_TVALID_resp_r = 0;
logic inj_TVALID_awaddr_r = 0;
logic inj_TVALID_raddr_r = 0;

logic log_TREADY_r = 0;

logic pause_enable_rdata = 0;
logic log_enable_rdata = 0;
logic drop_enable_rdata = 0;

logic pause_enable_wdata = 0;
logic log_enable_wdata = 0;
logic drop_enable_wdata = 0;
```

```
    logic pause_enable_raddr = 0;
    logic log_enable_raddr = 0;
    logic drop_enable_raddr = 0;

    logic pause_enable_awaddr = 0;
    logic log_enable_awaddr = 0;
    logic drop_enable_awaddr = 0;

    logic pause_enable_resp = 0;
    logic log_enable_resp = 0;
    logic drop_enable_resp = 0;

    logic [15:0] AWADDR_cnt = 1;
    logic [15:0] ARADDR_cnt = 1;

    // rdata assignments
    assign dout_TDEST_rdata = dout_TDEST_internal_rdata;
    assign inj_success_rdata = inj_TVALID_rdata && inj_TREADY_rdata; //NOTE NOT
!inj_TREADY_rdata

    // wdata assignments
    assign inj_success_wdata = inj_TVALID_wdata && inj_TREADY_wdata;
    assign inj_success_awaddr = inj_TVALID_awaddr && inj_TREADY_awaddr;
    assign inj_success_raddr = inj_TVALID_raddr && !inj_TREADY_raddr; // not used

    // rdata assignments
    assign inj_success_resp = inj_TVALID_resp && inj_TREADY_resp;

    // assign TREADY values (we reuse the log_TREADY register)
    assign log_TREADY_rdata = log_TREADY_r;
    assign log_TREADY_wdata = log_TREADY_r;
    assign log_TREADY_raddr = log_TREADY_r;
    assign log_TREADY_awaddr = log_TREADY_r;
    assign log_TREADY_resp = log_TREADY_r;

    // pause, log, drop, and inject for rdata
    assign pause_rdata = pause_enable_rdata;
    assign drop_rdata = drop_enable_rdata;
    assign log_en_rdata = log_enable_rdata;

    // pause, log, drop, and inject for wdata
    assign pause_wdata = pause_enable_wdata;
    assign drop_wdata = drop_enable_wdata;
```

```
    assign log_en_wdata = log_enable_wdata;

    // pause, log, drop, and inject for araddr
    assign pause_raddr = pause_enable_raddr;
    assign drop_raddr = drop_enable_raddr;
    assign log_en_raddr = log_enable_raddr;

    // pause, log, drop, and inject for awaddr
    assign pause_awaddr = pause_enable_awaddr;
    assign drop_awaddr = drop_enable_awaddr;
    assign log_en_awaddr = log_enable_awaddr;

    // pause, log, drop, and inject for resp
    assign pause_resp = pause_enable_resp;
    assign drop_resp = drop_enable_resp;
    assign log_en_resp = log_enable_resp;

    // inj_TVALID signals
    assign inj_TVALID_rdata = inj_TVALID_rdata_r;
    assign inj_TVALID_wdata = inj_TVALID_wdata_r;
    assign inj_TVALID_raddr = inj_TVALID_raddr_r;
    assign inj_TVALID_awaddr = inj_TVALID_awaddr_r;
    assign inj_TVALID_resp = inj_TVALID_resp_r;

    ////////////////////////////////////////////////
    ///////////   INITIALIZE SIGNALS   ////////////
    ////////////////////////////////////////////////

    localparam START = 6'd0;
    localparam DROP = 6'd1;
    localparam INJECT = 6'd2;
    localparam WAIT = 6'd3;
    localparam LOG = 6'd4;
    localparam PAUSE = 6'd5;
    localparam DONE_DROP = 6'd6;
    // localparam DONE_LOG = 6'd7;
    localparam DONE_INJECT = 6'd8;
    localparam DONE_PAUSE = 6'd9;
    localparam WAIT_NEXT = 6'd10;
    localparam UNPAUSE = 6'd11;
    localparam QUIT_DROP = 6'd12;
    localparam UNLOG = 6'd13;
    // depend on the control path to bring us to the DONE state
```

```
    //////// Available functions/operations ////////////////////////////////////////
    ////////////////////////////////////////////////////////////////////////////////
    ///////  pause: rdata (0), wdata (1)                                   ////////
    ///////  drop: rdata (2), wdata (3)                                    ////////
    ///////  inject: rdata (4), wdata (5), resp(11)                        ////////
    ///////  log: rdata (6), wdata (7), raddress (8), waddress (9), resp(10) ////////
    ////////////////////////////////////////////////////////////////////////////////

    ////////////// OP CODE VISUAL //////////////////
    ////////////////////////////////////////////////
    //// ################ || %%%%%%%%%%%% ////////
    ////////////////////////////////////////////////

    // OP code: 16bits for info used for both address and data(#), 12bits for
operation (%), insignificant bit can be used for anything later ($)

    always @(posedge clk) begin

        if(rst) begin

            // rdata
            pause_enable_rdata = 0;
            log_enable_rdata = 0;
            drop_enable_rdata = 0;

            // wdata
            pause_enable_wdata = 0;
            log_enable_wdata = 0;
            drop_enable_wdata = 0;

            // raddr
            pause_enable_raddr = 0;
            log_enable_raddr = 0;
            drop_enable_raddr = 0;

            // awaddr
            pause_enable_awaddr = 0;
            log_enable_awaddr = 0;
            drop_enable_awaddr = 0;

            // resp
            pause_enable_resp = 0; // will not use
```

```
            log_enable_resp = 0;
            drop_enable_resp = 0; // will not use

            // done signals
            done_START = 0;
            done_DROP = 0;
            done_INJECT = 0;
            done_WAIT = 0;
            done_LOG = 0;
            done_PAUSE = 0;
            done_UNPAUSE = 0;
            done_QUIT_DROP = 0;
            done_UNLOG = 0;

            done_DONE_DROP = 0;
            done_DONE_LOG = 0;
            done_DONE_INJECT = 0;
            done_DONE_PAUSE = 0;
            done_WAIT_NEXT = 0;

            inj_TVALID_rdata_r = 0;
            inj_TVALID_wdata_r = 0;
            inj_TVALID_resp_r = 0;
            inj_TVALID_awaddr_r = 0;
            log_TREADY_r = 0;
        // note to self: might have to check for valid streams here

        end else begin

            // update current state
            // reset done signals back to 0
            done_START = 0;
            done_DROP = 0;
            done_INJECT = 0;
            done_WAIT = 0;
            done_LOG = 0;
            done_PAUSE = 0;
            done_DONE_DROP = 0;
            done_DONE_LOG = 0;
            done_DONE_INJECT = 0;
            done_DONE_PAUSE = 0;
            done_WAIT_NEXT = 0;
            done_UNPAUSE = 0;
```

```
            done_QUIT_DROP = 0;
            done_UNLOG = 0;

            case (state_current)

                START: begin
                    done_START = 1;
                end

                DROP: begin

                    if(cmd_in_TDATA[3] == 1) begin
                        drop_enable_rdata = 1; //this should automatically assign to
the wire
                        done_DROP = 1;
                    end
                    else if(cmd_in_TDATA[4] == 1) begin
                        drop_enable_wdata = 1;
                        drop_enable_awaddr = 1;
                        done_DROP = 1;
                    end

                    else if(cmd_in_TDATA[6]) begin
                        drop_enable_resp = 1;
                        done_DROP = 1;
                    end

                    else
                        done_DROP = 0;

                end

                QUIT_DROP: begin

                    drop_enable_rdata = 0;
                    drop_enable_wdata = 0;
                    drop_enable_awaddr = 0;
                    drop_enable_resp = 0;

                end

                INJECT: begin
```

```
                    // extract data information
                    if(cmd_in_TDATA[5] || master_inject_enable_rdata) begin

                        inj_TVALID_rdata_r = 1;
                        //inj_TVALID_raddr_r = 1;
                        inj_TDATA_rdata = cmd_in_TDATA[28:13]; // use side channels?
                        //inj_TDATA_raddr = ARADDR_cnt;

                        done_INJECT = 1;

                    end

                    else if(cmd_in_TDATA[6]) begin // NOTE: i don't think we inject to
write although we can

                        inj_TVALID_awaddr_r = 1;
                        inj_TVALID_wdata_r = 1;
                        inj_TDATA_wdata = cmd_in_TDATA[28:13]; // use side channels?
                        inj_TDATA_awaddr = AWADDR_cnt;

                        done_INJECT = 1;

                    end

                    else if(cmd_in_TDATA[12] || master_inject_enable_resp) begin

                        inj_TVALID_resp_r = 1;
                        inj_TDATA_resp = 1; // use side channels?
                        done_INJECT = 1;
                    end

                    else
                        done_INJECT = 1;

                end

                WAIT: begin

                    if(cmd_in_TDATA[5] || master_inject_enable_rdata) begin // inject
or drop rdata
                        done_WAIT = inj_success_rdata; // && inj_success_raddr;

                        if(done_WAIT) begin
```

```
                        inj_TVALID_rdata_r = 0; //MODIFIED
                        //inj_TVALID_raddr_r = 0;
                    end

                end

                else if(cmd_in_TDATA[6]) begin // inject wdata
                    done_WAIT = inj_success_wdata && inj_success_awaddr;

                    if(done_WAIT) begin
                        inj_TVALID_wdata_r = 0; //MODIFIED
                        inj_TVALID_awaddr_r = 0;
                    end
                end

                else if(cmd_in_TDATA[12] || master_inject_enable_resp) //inject
resp or drop wdata
                    done_WAIT = inj_success_resp;

                else
                    done_WAIT = 0;

            end

            WAIT_NEXT: begin

                if(cmd_in_TDATA[5])
                    done_WAIT_NEXT = !(din_TVALID_rdata && dout_TREADY_rdata); //
for inject rdata

                else if(cmd_in_TDATA[6])
                    done_WAIT_NEXT = !(dout_TVALID_wdata && din_TREADY_wdata); //
for inject wdata

            end


            LOG: begin

                if(cmd_in_TDATA[7]) begin
                    log_enable_rdata = 1;
                    log_TREADY_r = 1;
                end
```

```
        else if(cmd_in_TDATA[8]) begin
            log_enable_wdata = 1;
            log_TREADY_r = 1;
        end

        else if(cmd_in_TDATA[9]) begin
            log_enable_raddr = 1;
            log_TREADY_r = 1;
        end

        else if(cmd_in_TDATA[10]) begin
            log_enable_awaddr = 1;
            log_TREADY_r = 1;
        end

        else if(cmd_in_TDATA[11]) begin
            log_enable_resp = 1;
            log_TREADY_r = 1;
        end

        done_LOG = 1;

    end

    UNLOG: begin
        if(cmd_in_TDATA[7] && cmd_in_TDATA[0]) begin
            log_enable_rdata = 0;
            log_TREADY_r = 0;
        end

        else if(cmd_in_TDATA[8] && cmd_in_TDATA[0]) begin
            log_enable_wdata = 0;
            log_TREADY_r = 0;
        end

        else if(cmd_in_TDATA[9] && cmd_in_TDATA[0]) begin
            log_enable_raddr = 0;
            log_TREADY_r = 0;
        end

        else if(cmd_in_TDATA[10] && cmd_in_TDATA[0]) begin
            log_enable_awaddr = 0;
```

```
                        log_TREADY_r = 0;
                    end

                    else if(cmd_in_TDATA[11] && cmd_in_TDATA[0]) begin
                        log_enable_resp = 0;
                        log_TREADY_r = 0;
                    end

                    done_UNLOG = 1;
                end

                PAUSE: begin

                    if(cmd_in_TDATA[1])
                        pause_enable_rdata = 1;

                    else if(cmd_in_TDATA[2])
                        pause_enable_wdata = 1;

                    done_PAUSE = 1;

                end

                UNPAUSE: begin

                    if(cmd_in_TDATA[1] && cmd_in_TDATA[0])
                        pause_enable_rdata = 0;
                    else if(cmd_in_TDATA[2] && cmd_in_TDATA[0])
                        pause_enable_wdata = 0;

                    done_UNPAUSE = 1;

                end

                DONE_DROP: begin

                    //drop_enable_rdata = 0; //this should automatically assign to the
wire
                    drop_enable_wdata = 0;
                    drop_enable_resp = 0;
                    drop_enable_awaddr = 0;

                    inj_TVALID_rdata_r = 0;
```

```
                inj_TVALID_resp_r = 0;
                inj_TDATA_rdata = 0;
                inj_TDATA_resp = 0;
                inj_TDATA_wdata = 0;

                done_DONE_DROP = 1;

            end
            /*
            DONE_LOG: begin

                log_enable_rdata = 0;
                log_enable_wdata = 0;
                log_enable_raddr = 0;
                log_enable_awaddr = 0;
                log_enable_resp = 0;
                log_TREADY_r = 0;

                done_DONE_LOG = 1;

            end
            */
            DONE_INJECT: begin

                inj_TVALID_rdata_r = 0;
                inj_TVALID_wdata_r = 0;
                inj_TVALID_resp_r = 0;
                inj_TVALID_awaddr_r = 0;
                inj_TVALID_raddr_r = 0;

                /*
                inj_TDATA_rdata = 0;
                inj_TDATA_resp = 0;
                inj_TDATA_wdata = 0;
                inj_TDATA_awaddr = 0;
                inj_TDATA_raddr = 0;
                */

                AWADDR_cnt += 1;
                ARADDR_cnt += 1;

                drop_enable_resp = 0;
                drop_enable_rdata = 0;
```

```
                done_DONE_INJECT = 1;

            end

            DONE_PAUSE: begin

                done_DONE_PAUSE = 1;

            end

        endcase
    end
end

////////////////////////////////////////////////////////////////
////////////////// INSTANTIATE 5 AXIS GUVS /////////////////////
////////////////////////////////////////////////////////////////

axis_governor #(
        .DATA_WIDTH(DATA_WIDTH),
        .DEST_WIDTH(`SAFE_DEST_WIDTH),
        .ID_WIDTH(`SAFE_ID_WIDTH)

) guv_rdata (

        .clk(clk),

        //Input AXI Stream.

        .in_TDATA(din_TDATA_rdata),
        .in_TVALID(din_TVALID_rdata),
        .in_TREADY(din_TREADY_rdata),
        .in_TDEST(din_TDEST_rdata),

        //Inject AXI Stream.
        .inj_TDATA(inj_TDATA_rdata),
        .inj_TVALID(inj_TVALID_rdata),
        .inj_TREADY(inj_TREADY_rdata),
        .inj_TDEST(inj_TDEST_rdata),

        //Output AXI Stream.
        .out_TDATA(dout_TDATA_rdata),
```

```
            .out_TVALID(dout_TVALID_rdata),
            .out_TREADY(dout_TREADY_rdata),
            .out_TDEST(dout_TDEST_internal_rdata),

            //Log AXI Stream.
            .log_TDATA(log_TDATA_rdata),
            .log_TVALID(log_TVALID_rdata),
            .log_TREADY(log_TREADY_rdata),
            .log_TDEST(log_TDEST_rdata),

            //Control signals
            .pause(pause_rdata),
            .drop(drop_rdata),
            .log_en(log_en_rdata)

        );


    axis_governor #(

            .DATA_WIDTH(DATA_WIDTH),
            .DEST_WIDTH(`SAFE_DEST_WIDTH),
            .ID_WIDTH(`SAFE_ID_WIDTH)

    ) guv_wdata (

            .clk(clk),

            //Input AXI Stream.
            .in_TDATA(dout_TDATA_wdata),
            .in_TVALID(dout_TVALID_wdata),
            .in_TREADY(dout_TREADY_wdata),
            .in_TDEST(),

            //Inject AXI Stream.
            .inj_TDATA(inj_TDATA_wdata),
            .inj_TVALID(inj_TVALID_wdata),
            .inj_TREADY(inj_TREADY_wdata),
            .inj_TDEST(),

            //Output AXI Stream.
            .out_TDATA(din_TDATA_wdata),
            .out_TVALID(din_TVALID_wdata),
```

```
        .out_TREADY(din_TREADY_wdata),
        .out_TDEST(),

        //Log AXI Stream.
        .log_TDATA(log_TDATA_wdata),
        .log_TVALID(log_TVALID_wdata),
        .log_TREADY(log_TREADY_wdata),
        .log_TDEST(),

        //Control signals
        .pause(pause_wdata),
        .drop(drop_wdata),
        .log_en(log_en_wdata)

    );

axis_governor #(

        .DATA_WIDTH(DATA_WIDTH),
        .DEST_WIDTH(`SAFE_DEST_WIDTH),
        .ID_WIDTH(`SAFE_ID_WIDTH)

) guv_raddr (

        .clk(clk),

        //Input AXI Stream.
        .in_TDATA(dout_TDATA_raddr),
        .in_TVALID(dout_TVALID_raddr),
        .in_TREADY(dout_TREADY_raddr),
        .in_TDEST(),

        //Inject AXI Stream.
        .inj_TDATA(inj_TDATA_raddr),
        .inj_TVALID(inj_TVALID_raddr),
        .inj_TREADY(inj_TREADY_raddr),
        .inj_TDEST(),

        //Output AXI Stream.
        .out_TDATA(din_TDATA_raddr),
        .out_TVALID(din_TVALID_raddr),
        .out_TREADY(din_TREADY_raddr),
        .out_TDEST(),
```

```
        //Log AXI Stream.
        .log_TDATA(log_TDATA_raddr),
        .log_TVALID(log_TVALID_raddr),
        .log_TREADY(log_TREADY_raddr),
        .log_TDEST(),

        //Control signals
        .pause(pause_raddr),
        .drop(drop_raddr),
        .log_en(log_en_raddr)

    );

axis_governor #(

        .DATA_WIDTH(DATA_WIDTH),
        .DEST_WIDTH(`SAFE_DEST_WIDTH),
        .ID_WIDTH(`SAFE_ID_WIDTH)

) guv_awaddr (

        .clk(clk),

        //Input AXI Stream.
        .in_TDATA(dout_TDATA_awaddr),
        .in_TVALID(dout_TVALID_awaddr),
        .in_TREADY(dout_TREADY_awaddr),
        .in_TDEST(),

        //Inject AXI Stream.
        .inj_TDATA(inj_TDATA_awaddr),
        .inj_TVALID(inj_TVALID_awaddr),
        .inj_TREADY(inj_TREADY_awaddr),
        .inj_TDEST(),

        //Output AXI Stream.
        .out_TDATA(din_TDATA_awaddr),
        .out_TVALID(din_TVALID_awaddr),
        .out_TREADY(din_TREADY_awaddr),
        .out_TDEST(),

        //Log AXI Stream.
```

```
        .log_TDATA(log_TDATA_awaddr),
        .log_TVALID(log_TVALID_awaddr),
        .log_TREADY(log_TREADY_awaddr),
        .log_TDEST(),

        //Control signals
        .pause(pause_awaddr),
        .drop(drop_awaddr),
        .log_en(log_en_awaddr)

    );

axis_governor #(

        .DATA_WIDTH(DATA_WIDTH),
        .DEST_WIDTH(`SAFE_DEST_WIDTH),
        .ID_WIDTH(`SAFE_ID_WIDTH)

) guv_resp (

        .clk(clk),

        //Input AXI Stream.
        .in_TDATA(din_TDATA_resp),
        .in_TVALID(din_TVALID_resp),
        .in_TREADY(din_TREADY_resp),
        .in_TDEST(),

        //Inject AXI Stream.
        .inj_TDATA(inj_TDATA_resp),
        .inj_TVALID(inj_TVALID_resp),
        .inj_TREADY(inj_TREADY_resp),
        .inj_TDEST(),

        //Output AXI Stream.
        .out_TDATA(dout_TDATA_resp),
        .out_TVALID(dout_TVALID_resp),
        .out_TREADY(dout_TREADY_resp),
        .out_TDEST(),

        //Log AXI Stream.
        .log_TDATA(log_TDATA_resp),
        .log_TVALID(log_TVALID_resp),
```

```
            .log_TREADY(log_TREADY_resp),
            .log_TDEST(),

            //Control signals
            .pause(pause_resp),
            .drop(drop_resp),
            .log_en(log_en_resp)

        );

endmodule
```

Appendix C: Top module code

```
`timescale 1ns / 1ps

module dbg_guv # (

    // will move this to dbg_guv module later
    parameter DATA_WIDTH = 64,
    parameter DEST_WIDTH = 16,
    parameter ID_WIDTH = 16
) (

    input clk,
    input rst,
    input wire [31:0] cmd_in_TDATA,
    input wire cmd_in_TVALID,
    output wire cmd_in_TREADY,

    //Input AXI Stream rdata.
    input wire [DATA_WIDTH-1:0] din_TDATA_rdata,
    input wire [DEST_WIDTH -1:0] din_TDEST_rdata,
    input wire din_TVALID_rdata,
    output wire din_TREADY_rdata,

    //Input AXI Stream wdata.
    output wire [DATA_WIDTH-1:0] din_TDATA_wdata,
    output wire din_TVALID_wdata,
    input wire din_TREADY_wdata,

    //Input AXI Stream raddr.
    output wire [DATA_WIDTH-1:0] din_TDATA_raddr,
    output wire din_TVALID_raddr,
    input wire din_TREADY_raddr,

    //Input AXI Stream awaddr.
    output wire [DATA_WIDTH-1:0] din_TDATA_awaddr,
    output wire din_TVALID_awaddr,
    input wire din_TREADY_awaddr,

    //Input AXI Stream resp.
    input wire [DATA_WIDTH-1:0] din_TDATA_resp,
```

```
    input wire din_TVALID_resp,
    output wire din_TREADY_resp,

    ////////////////////////////////////////////////////////////////////////////////

    //Output AXI Stream rdata.
    output wire [DATA_WIDTH-1:0] dout_TDATA_rdata,
    output wire [DEST_WIDTH -1:0] dout_TDEST_rdata,
    output wire dout_TVALID_rdata,
    input wire dout_TREADY_rdata,

    //Output AXI Stream wdata.
    input wire [DATA_WIDTH-1:0] dout_TDATA_wdata,
    input wire dout_TVALID_wdata,
    output wire dout_TREADY_wdata,

    //Output AXI Stream raddr.
    input wire [DATA_WIDTH-1:0] dout_TDATA_raddr,
    input wire dout_TVALID_raddr,
    output wire dout_TREADY_raddr,

    //Output AXI Stream awaddr.
    input wire [DATA_WIDTH-1:0] dout_TDATA_awaddr,
    input wire dout_TVALID_awaddr,
    output wire dout_TREADY_awaddr,

    //Output AXI Stream resp.
    output wire [DATA_WIDTH-1:0] dout_TDATA_resp,
    output wire dout_TVALID_resp,
    input wire dout_TREADY_resp,

    output logic [10:0] current_state,
    output logic [10:0] next_state,
    output logic [31:0] cmd_i,

    output wire [DATA_WIDTH-1:0] log_TDATA_rdata_o,
    output wire log_TVALID_rdata_o,
    input wire log_TREADY_rdata_i,
    output wire [DEST_WIDTH -1:0] log_TDEST_rdata_o,

    output wire [DATA_WIDTH-1:0] log_TDATA_wdata_o,
    output wire log_TVALID_wdata_o,
    input wire log_TREADY_wdata_i,
```

```
    output wire [DATA_WIDTH-1:0] log_TDATA_raddr_o,
    output wire log_TVALID_raddr_o,
    input wire log_TREADY_raddr_i,

    output wire [DATA_WIDTH-1:0] log_TDATA_awaddr_o,
    output wire log_TVALID_awaddr_o,
    input wire log_TREADY_awaddr_i,

    output wire [DATA_WIDTH-1:0] log_TDATA_resp_o,
    output wire log_TVALID_resp_o,
    input wire log_TREADY_resp_i

);

    // CONTROLPATH signals
    wire done_START;
    wire done_DROP;
    wire done_INJECT;
    wire done_WAIT;
    wire done_LOG;
    wire done_PAUSE;
    wire done_DONE_DROP;
    wire done_DONE_LOG;
    wire done_DONE_INJECT;
    wire done_DONE_PAUSE;
    wire done_WAIT_NEXT;
    wire done_UNPAUSE;
    wire done_QUIT_DROP;
    wire done_UNLOG;

    wire newFlit;

    wire [10:0] curr_state;
    wire master_inject_enable_resp;
    wire master_inject_enable_rdata;

    wire [9:0] state_next_o;

    logic [26:0] data = 0;

    wire [28:0] data_in_o;
    assign current_state = curr_state;
```

```
control_FSM U2(
    clk,
    rst,
    // axi stream
    cmd_in_TDATA,
    cmd_in_TVALID,
    cmd_in_TREADY,

    done_START,
    done_DROP,
    done_INJECT,
    done_WAIT,
    done_LOG,
    done_PAUSE,
    done_DONE_DROP,
    done_DONE_LOG,
    done_DONE_INJECT,
    done_DONE_PAUSE,
    done_WAIT_NEXT,
    done_UNPAUSE,
    done_QUIT_DROP,
    done_UNLOG,

    curr_state,
    master_inject_enable_resp,
    master_inject_enable_rdata,
    newFlit,
    next_state,
    cmd_i

);

// init module
datapath U1 (
    clk,
    rst,
    curr_state,
    master_inject_enable_resp,
    master_inject_enable_rdata,

    //Input command stream
```

```
        cmd_in_TDATA,

        //Input AXI Stream rdata.
        din_TDATA_rdata,
        din_TDEST_rdata,
        din_TVALID_rdata,
        din_TREADY_rdata,

        //Input AXI Stream wdata.
        din_TDATA_wdata,
        din_TVALID_wdata,
        din_TREADY_wdata,

        //Input AXI Stream raddr.
        din_TDATA_raddr,
        din_TVALID_raddr,
        din_TREADY_raddr,

        //Input AXI Stream awaddr.
        din_TDATA_awaddr,
        din_TVALID_awaddr,
        din_TREADY_awaddr,

        //Input AXI Stream resp.
        din_TDATA_resp,
        din_TVALID_resp,
        din_TREADY_resp,

        //Output AXI Stream rdata.
        dout_TDATA_rdata,
        dout_TDEST_rdata,
        dout_TVALID_rdata,
        dout_TREADY_rdata,

        //Output AXI Stream wdata.
        dout_TDATA_wdata,
        dout_TVALID_wdata,
        dout_TREADY_wdata,

        //Output AXI Stream raddr.
        dout_TDATA_raddr,
        dout_TVALID_raddr,
        dout_TREADY_raddr,
```

```
    //Output AXI Stream awaddr.
    dout_TDATA_awaddr,
    dout_TVALID_awaddr,
    dout_TREADY_awaddr,

    //Output AXI Stream resp.
    dout_TDATA_resp,
    dout_TVALID_resp,
    dout_TREADY_resp,

    // Done signals
    done_START,
    done_DROP,
    done_INJECT,
    done_WAIT,
    done_LOG,
    done_PAUSE,
    done_DONE_DROP,
    done_DONE_LOG,
    done_DONE_INJECT,
    done_DONE_PAUSE,
    done_WAIT_NEXT,
    done_UNPAUSE,
    done_QUIT_DROP,
    done_UNLOG,
    newFlit,

    log_TDATA_rdata,
    log_TVALID_rdata,
    log_TREADY_rdata,
    log_TDEST_rdata,

    log_TDATA_wdata,
    log_TVALID_wdata,
    log_TREADY_wdata,

    log_TDATA_raddr,
    log_TVALID_raddr,
    log_TREADY_raddr,

    log_TDATA_awaddr,
    log_TVALID_awaddr,
```

```
        log_TREADY_awaddr,

        log_TDATA_resp,
        log_TVALID_resp,
        log_TREADY_resp

    );

endmodule
```