Foreign Exchange Currency High-Frequency Trading (Forex HFT)

Spring 2016

Members: Graham Gobieski, Kevin Kwan, Ziyi Zhu, Shang Liu

UNIs: gsg2120, kjk2150, zz2374, sl3881

Table of Contents

1 Abstract

Our project goal is to create a High Frequency Trading (HFT) platform on an Altera Cyclone V SoCKit board that can detect arbitrage opportunities in the Foreign Exchange (FOREX) Market. Simulated data is streamed to the FPGA and stored in on-chip memory. The FPGA runs the Bellman-Ford algorithm on the data and looks for negative weight cycles.

2 Motivation

High frequency trading is a trading platform that uses computer algorithms and powerful technology tools to perform a large number of trades at very high speeds. Initially, HFT firms operated on a time scale of seconds, but as technology has improved, so has the time required to execute a trade. Firms now compete at the milli- or even microsecond level. This has led to many firms turning to field programmable gate arrays (FPGAs) to achieve greater performance.

Our project focuses on triangular arbitrage opportunities on the foreign exchange market (Forex). The Forex market is a decentralized marketplace for trading currency. All trading is conducted over the counter via computer networks between traders around the world. Unlike the stock market, the Forex market is open 24 hours for most of the week.

Currencies are priced in relation to each other and quoted in pairs that look like this: `EUR/USD 1.1837`. The currency on the left is the base currency and the one on the right is called the cross currency or quote. The base currency is always assumed to be one unit, and the quoted price is what the base currency is equal to in the other currency. In this example, `1 Euro = 1.1837 USD.`

Triangular arbitrage takes advantage of pricing inefficiencies across three or more different currencies. In a three currency situation, one currency is exchanged for a second, the second for a third , and finally the third back to the original currency. For example, if the exchange rates for the following currency pairs were `EUR/USD 1.1837`, `EUR/GBP 0.7231`, and `GBP/USD 1.6388` a trader could use 11,847 USD to buy 10,000 Euros. Those Euros could be sold for 7231 British Pounds, which could then be sold for 11,850 USD, netting a profit of 13 USD. Unfortunately, acting on these price inefficiencies quickly corrects them, meaning traders must be ready to act immediately when an arbitrage opportunity occurs.

Our group implements a Forex arbitrage calculator on an FPGA using a hardware implementation of the Bellman-Ford algorithm.

3. Design Overview and Previous Work

3.1 Arbitrage Identification

Triangular arbitrage opportunities arise when a cycle is determined such that the edge weights satisfy the following expression:

$$\mathbf{w_1 * w_2 * w_3 * \ldots * w_n > 1}$$

However, cycles that adhere to the above requirement are particulary difficult to find in graphs. Instead we must transform the edge weights of the graph so that standard graph algorithms can be used. First we take the logarithm of both sides, such that:

$$\mathbf{\log(w_1) + \log(w_2) + \log(w_3) + \ldots + \log(w_n) > 0}$$

If instead we take the negative log, this results in a sign flip:

$$\mathbf{\log(w_1) + \log(w_2) + \log(w_3) + \ldots + \log(w_n) < 0}$$

Thus, if we look for negative weight cycles using the logarithm of the edge weights, we will find cycles that satisfy the requirements outlined above. Luckily, the Bellman-Ford algorithm is a standard graph algorithm that can be used to easily detect negative weight cycles in O(VE) time.

3.2 Bellman-Ford Algorithm

Algorithm 3.1: Standard Bellman-Ford

- Let *G(V, E)* be a graph with vertices, *V*, and edges, *E*.

- Let *w(x)* denote the weight of vertex *x*.
- Let *w(i, j)* denote the weight of the edge from source vertex *i* to destination vertex *j*.
- Let *p(j)* denote the predecessor of vertex *j*.

```
for each vertex x in V do
    if x is source then
        w(x) = 0
    else
        w(x) = INFINITY
        p(x) = NULL
    end if
end for

for i = 1 to v - 1 do
    for each edge(i, j) in E do
        if w(i) + w(i, j) < w(j) then //Relaxation
            w(j) = w(i) + w(i, j)
            p(j) = i
        end if
    end for
end for

for each edge(i, j) in E do
    if w(j) > w(i) + w(i, j) then
        //Found Negative-Weight Cycle
    end if
end for
```

The Bellman-Ford algorthm is a standard graph algorithm that seeks to solve the single-source shortest path problem. Mainly this problem describes the situation in which a source node is selected and the shortest paths to every other node in the graph need to be determined. In unit graphs, breath first search may be used, but in graphs that have non-unit edge weights the Bellman-Ford algorthm must be used.

Briefly, in the Bellman-Ford algorithm "each vertex maintains the weight of the shortest path from the source vertex to itself and the vertex which precedes it in the shortest path. In each iteration, all edges are relaxed [w(i) + w(i, j) < w(j)] and the weight of each vertex is updated if necessary. After the ith iteration, the algorithm finds all shortest paths consisting of at most i edges." After all shortest paths have been identified, the algorithm loops through all of the edges and looks for edges that can further decrease the value of the shortest path. If this case then a negative weight cycle has been found since a path can have at most v-1 edges. Proof of correctness can be found in *Introduction to Algorithms* by Cormen, Leiserson, Rivest, and Stein.

3.3 Previous Work

The Bellman-Ford Algorithm is well studied on hardware systems. Early approaches constructed circuits that represented graphs by using logical units as graph nodes. However, these approaches required recompilation for each new graph. As a result, several new methods were developed to process graphs stored in on-chip memory. These methods ranged from a fairly-standard port of the sequential algorithm presented above to approaches that parallized different parts of Bellman-Ford.

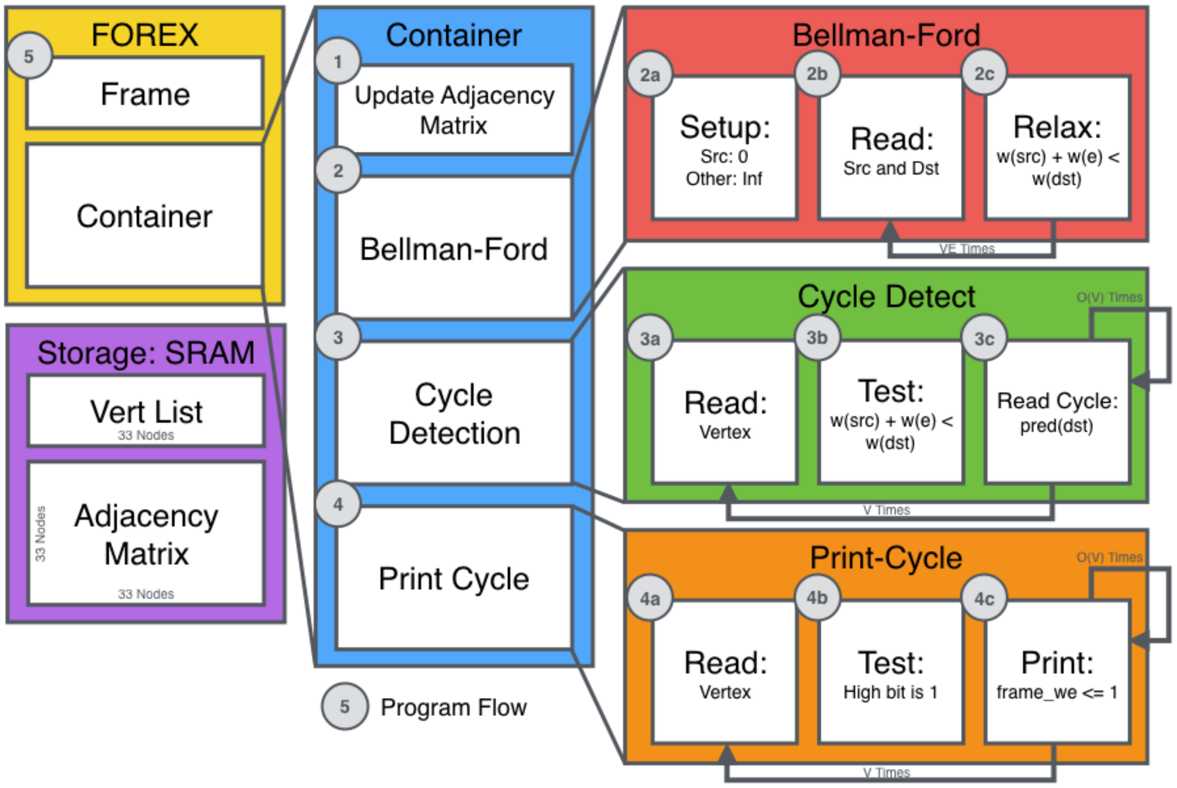
On-chip memory, however, soon proved to be a bottleneck and researchers, accordingly, developed new approaches for utilizing off-chip DDR3 memory and streaming edges onto the board. Specifically Prassana et. al. presented an approach that stored edges and vertice weights in DDR3 memory and used data-forwarding and streaming to process an entire graph in record time. Initially, our approach utilized the methods originally described in this paper. However, we eventually settled on a more standard port of the Bellman-Ford algorithm as our graph had a maximum number of nodes and edges, which can easily fit in on-chip SRAM.

4. Implementation

The implementation of the system is divided into two parts: a hardware portion that encompasses the HDL code running on the FPGA and a software portion that is respondsible for streaming data to the FPGA.

4.1 Hardware Design

Figure 4.1: Overview of hardware design



There are two overarching modules that dictate the control flow of the program – FOREX.sv and Container.sv. The FOREX module is respondsible for controlling the interaction between the frame buffer (contained in the Frame module), the Container module, and any new data that has arrived over the bus. The container module is respondsible for running in sequence the Bellman-Ford algorithm, the cycle detection algorithm, and the print cycle protocol. Additionally, this module updates the adjacency matrix and delegates which module has access to memory at any point in time. As such the Container module implements a finite state machine that keeps track of what is running and what wires need to be connected to the memory modules in order to give the correct results.

4.1.1 Bellman-Ford Algorithm

The Bellman-Ford algorithm is implemented on hardware as a finite state machine. When a reset signal is high the module resets and starts Bellman-Ford by moving into the setup state and resetting all of the vertice weights to either infinity if its not the source or zero if it is the source. Once setup is complete, the module moves into a cycle of read, relax, and write states that read a source vertex, a destination vertex, and edge; determine if the vertex in question should be relaxed (satisfies the inequality in Figure 4.1); and writes the new weight of the vertex to memory if it should be relaxed. This cycle runs O(VE) times and the current vertex, source, and destination are maintained by variables stored in registers.

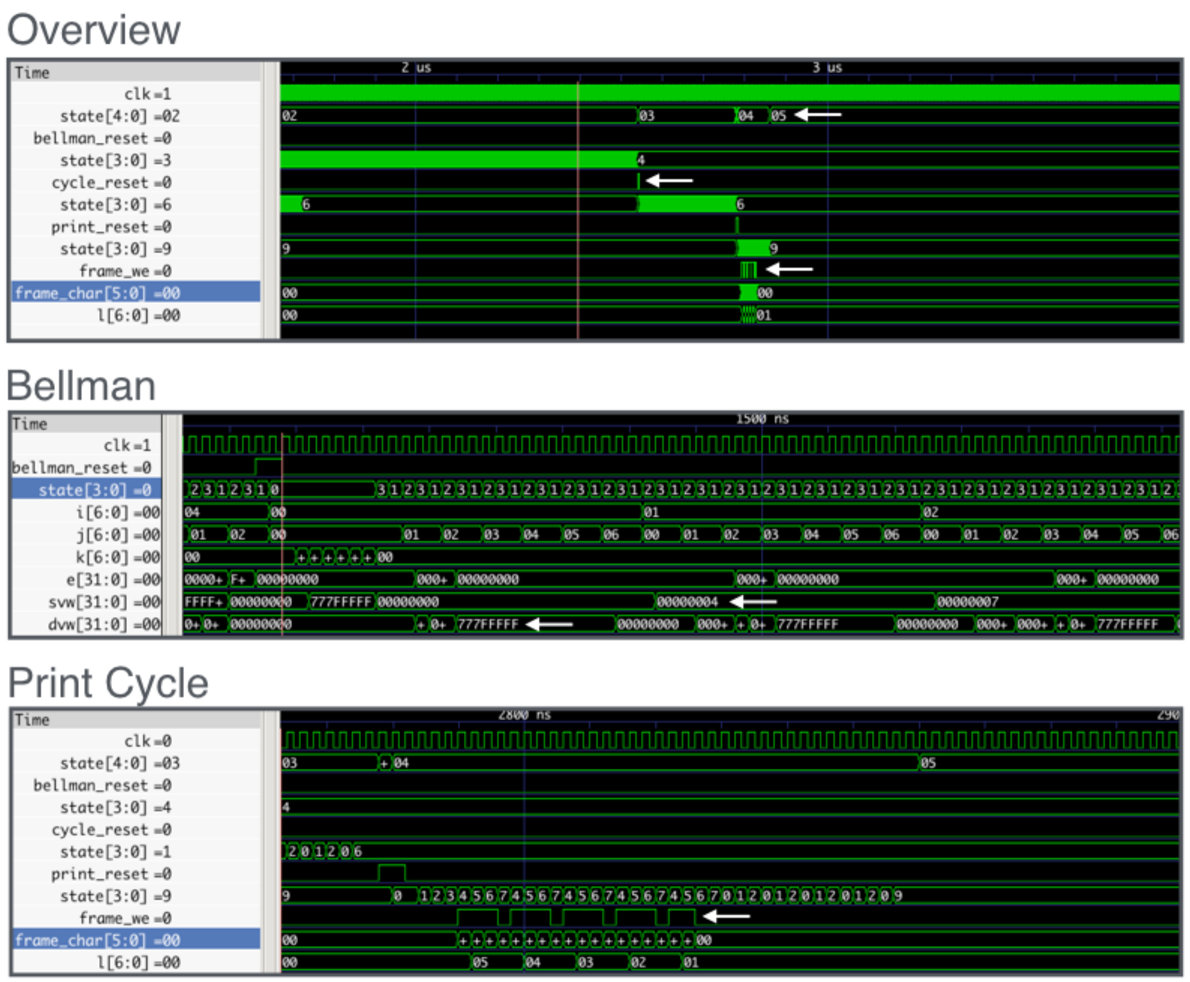
4.1.2 Cycle Detection

On reset the cycle detection algorithm begins looping through the array of vertices that has been updated by the Bellman-Ford algorithm. If it finds an edge, source, and destination that satisfy the inequality in Figure 4.1, the module moves to the read cycle state. The read cycle state reads the predecessor of the vertex and sets the highest bit of the vertex to high (to track that it has found this cycle). Then it follows the path dictated by the predecessors until it arrives back at the vertex that it started with. This represents a complete cycle and the module can move onto reading additional cycles and vertices by picking up where it left off before it started reading the cycle. Variables stored in registers are used to maintain state and place in the vertex list.

The Print Cycle module works in a very similar manner to the Cycle Detect module just described, except that instead of testing for an inequality it tests the highest bit and when a cycle is determined it turns off the highest bit so that the same cycle is not printed multiple times.

4.1.3 Memory Access and Timing

Figure 4.2: Timing Diagrams



Timing and coordinated memory accesses are not trivial in the system, especially since the systems relies on a finite state machine that has more than thirty states. Timing diagrams, generated by GTKWave, were important in order to get the memory accesses correct. In Figure 4.2, we can see the how the state variable contained in the Container module changes over time, moving from running Bellman-Ford (= 2) to running cycle detection (= 3). Each time the state changes, we can also see a brief reset signal that pulses to high resetting the module before entering the states contained in the module. In the Bellman-Ford segment we can see the values of the source vertex and the destination vertex change over time. Finally we can see the frame write enable go high for several clock cycles indicating that a negative cycle has been identified and is being print to screen.

One important problem to note here is that of nesting non-blocking assignments. Since we use several modules in sequence, coordinated by the Container module, and since, at times, memory access depends on several non-blocking assignments, writing or reading new values will take more cycles than initially thought because the last variable in a sequence of n non-blocking assignments will only be updated after n cycles. However, in some situations there is a simple solution to use combinational blocking assignments that coordinate with the non-blocking assignments. In the system described in Figure 4.1, the container module coordinates memory access using combinational logic that checks the state of the system and, using blocking assignments, assigns the pins of the memory modules to the correct output ports of the current running module. In additon, idle states were introduced so that memory updates would be felt before an algorithm continued.

4.1.4 Graph Storage

There are two standard ways to store a graph: an adjacency list and and an adjaceny matrix. We choose the second format because it is easier represented on the FPGA using a large, flattented, two-dimensional vector. In addition the Bellman-Ford algorithm is just as capable at processing an adjacency matrix as it is an adjacency list.

Specifically speaking, there are several pieces of information that must be stored. The weights of edges between vertex *i* and vertex *j* denoted w(i, j) will be stored in the adjacency matrix. Additionally the predecessor, denoted p(i), and the weight, denoted w(i), must be stored for each vertex. These values will be stored in a large one-dimensional vector in which the index will correspond to the vertex.

Using rough calculations we estimate the total memory usage in the following way:

V = 33 nodes

Edges = 33^2 = 1089 edges

Total Bits = (1089 edges) * (32 bits/weight of edge) + (33 nodes) * [(32 bits/weight of vertex)

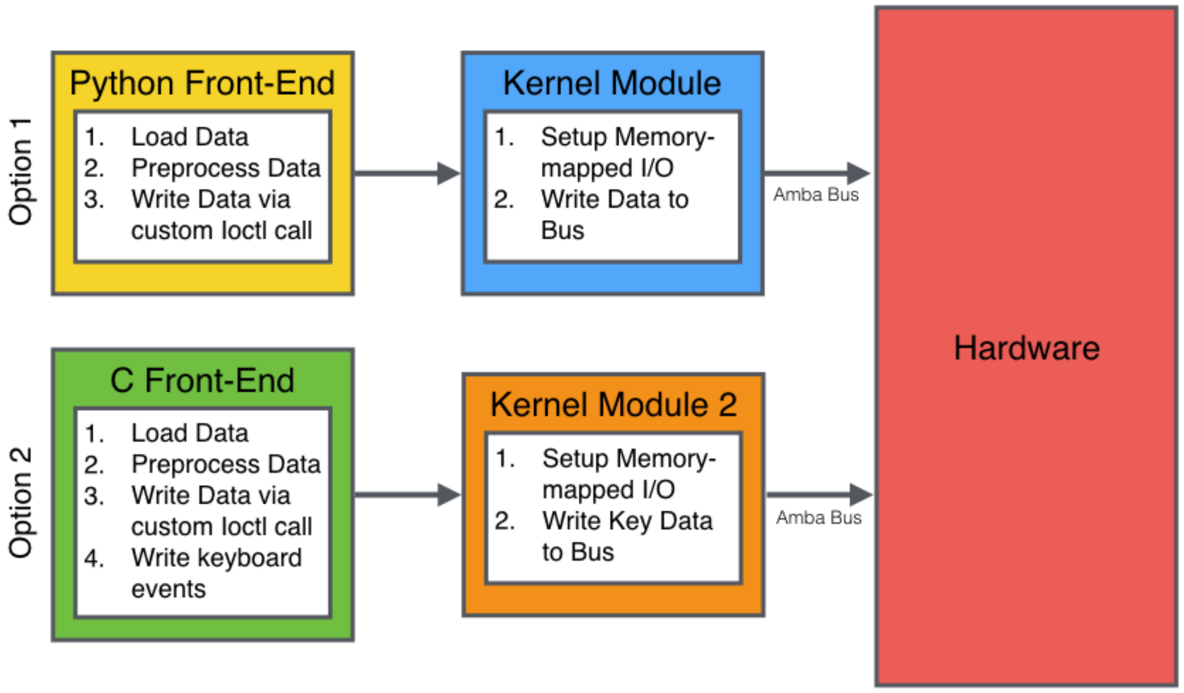
+ (7 bits/predecessor of vertex) + (1 bit/cycle on/off)] = 35,970 bits ~ 4.5kb

4.1.5 VGA Display

To display negative cycles (paths for exchanging currencies to make profits), we use a frame buffer and the VGA module to display numbers in sequence, indicating each currency within a cycle. When a new negative loop is discovered, the ***Print Cycle*** module writes indicies to the frame buffer. This frame buffer is then read every clock cycle to determine what and where a character needs to be displayed. The frame buffer is a two-dimensional vector that is 40 units wide and 30 units tall. Each unit represents a character and is a total of 16x16 pixels large. If a certain position is enabled, the module will look in ROM and read the corresponding character. In total there are 38 characters sprites that can be used to represent indices and currencies. These sprites include 26 capital letters, 10 numeric characters, 1 arrow and 1 character that represents a space. The characters are initially stored in a .mif file, but are loaded on startup into ROM generated by the MegaWizard Plug-in Manager in Quartus II.

4.2 Software Design

Figure 4.3: Overview of software design



4.1.2 Data Format and Preprocesing

Data is either read from files placed in a particular directory or is hard-coded (mostly for testing purposes) into the program itself. The .csv files have the following format:

Figure 4.4: Sample Forex data

Date	Time	Open	High	Low	Close	Volume
2016.02.01	00:00	1.08481	1.08494	1.08481	1.08494	0
2016.02.01	00:01	1.08492	1.08492	1.08471	1.08471	0
2016.02.01	00:02	1.08471	1.08476	1.0846	1.08474	0
2016.02.01	00:03	1.08474	1.08474	1.08468	1.08472	0
2016.02.01	00:04	1.08471	1.08472	1.0847	1.08471	0
2016.02.01	00:05	1.08473	1.08477	1.08473	1.08476	0
2016.02.01	00:06	1.08477	1.08477	1.08469	1.08472	0
2016.02.01	00:07	1.08473	1.08477	1.08473	1.08477	0
2016.02.01	00:08	1.08478	1.08484	1.08477	1.08481	0

Data is preprocessed before being streamed to the fpga because floating-point arithmetic on the FPGA is not a trivial task and may require a custom implementation of various operations. We, therefore, decided to utilize the floating-point arithmetic resources of the AMD chip onboard the FPGA. As such, we propose a two-step process that manipulates the data in such a way where only integers are streamed to the FPGA. This preprocessing operation is described below:

4.1.2.1 Logarithm

As part of the algorithm to detect arbitrage the logarithm of rate is required so that negative-weight cycles are possible (please see seciont 3 for more discussion on the algorithm). We will use the logarithm mechanism on the AMD chip to calculate the logarithm of each rate.

4.1.2.2 Rounding

Once the logarithm has been taken we will convert the resulting floating point to an integer by multiplying by a sufficiently large factor of 10 (the greater the factor the higher the precision) and then throw-away the remaining decimal. In this way we will be left with large integers that can be streamed and operated on the FPGA efficiently.

4.1.2 Communication

After preprocessing we stream the data via the Amba bus to the FPGA using custom memory-mappied I/O device drivers. The integers that we will stream will be fixed at 32 bits, and we stream as fast as possible with minimum delay so that we can effectively simulate reality.

We wrote two different device drivers with two separate interfaces in order to investigate how our desgined performed in two different situations. The first device driver pairs a custom linux kernel module with a python front-end that can efficienty read CSV files and then call our custom sycall to communicate (via memory-mapped I/O operations) to the FPGA.

The second device driver pairs a C front-end with a custom linux kernel module and supports the use of the keyboard. Not only is data read from files streamed to the FPGA (via memory-mapped I/O operations), but so are keyboard events.

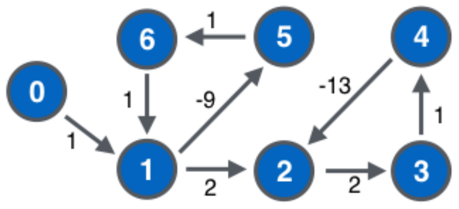
4.3.3 Keyboard

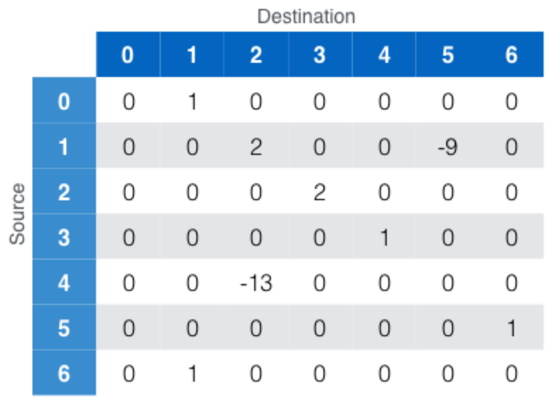
We use the keyboard in the second display driver to control different views on the screen. The keyboard is connected to the FPGA as a peripheral and communicates via libusb and a custom linux kernel module. Four keys are defined “esc”, “enter”, “up” and “down”. When the software built in the arm processor receives those four keys, it transmits the corresponding operation to the FPGA via the Avalon Bus. The "up" and "down" keys allows us to select a loop. The “enter” key gives us the name of each currency involved in the selected loop. Finally, the “esc” key exits from this detailed view and returns to the standard loop view.

5. Results and Performance

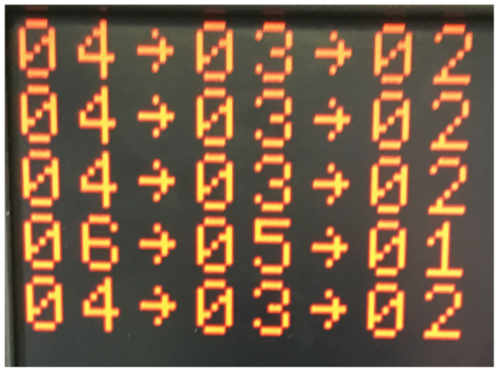
We were able to successfully detect negative weight cycles using test graphs that we created. We tested using graphs with a variety of sizes and configurations. As the following figure shows, our system was successful at not only detecting single cycles in graphs as the edges were updated, but also multiple cycles. The first three rows (of the VGA output) represent the cycles in the graph before the second cycle is streamed into memory. The last two rows represent the two cycles in the graph after all edges have been streamed in.

Figure 5.1: Results



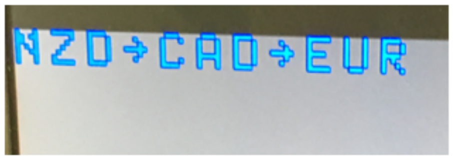


Source \ Destination	0	1	2	3	4	5	6
0	0	1	0	0	0	0	0
1	0	0	2	0	0	-9	0
2	0	0	0	2	0	0	0
3	0	0	0	0	1	0	0
4	0	0	-13	0	0	0	0
5	0	0	0	0	0	0	1
6	0	1	0	0	0	0	0



```
04→03→02
04→03→02
04→03→02
06→05→01
04→03→02
```

Cycle View



```
NZD→CAD→EUR
```

Expanded View

It is important to be able to detect multiple cycles because we want to be able to see all arbitrage opportunities at a given moment, not just the cycle with the most negative weight.

We did test our project with some historical data, but unfortunately were not able find any negative cycles. However, given our results from smaller tests, we believe it is more likely that the historical data we used, simply did not have any arbitrage opportunities. Our initial research had said that these opportunities were fairly uncommon. In this case, the project should not show any output because there are no negative cycles.

Overall, we feel our results are promising. Our project can successfully detect multiple negative cycles within a graph and display the corresponding nodes ("Expanded View" in Figure 5.1)that make up the cycle. We can also stream in historical data from the FOREX market and run it through the FPGA.

We estimate the time our project takes to run the Bellman-Ford algorithm and cycle detection is approximately 2.7ms and is able to process up to one million edges per second. This is only about 200x times slower than the slowest algorithm on the Graph500 index.

6. Lessons Learned and Future Work

6.1 Lessons Learned

The main takeaway we had from this project was that timing diagrams are very important, especially with memory access. We had issues with our project that seemed impossible to solve until we were able to generate timing diagrams. Once we could look at those, it was easy to see where the problems were occurring.

We also learned that memory is a major constraint on the FPGA. We initially thought that memory would be as simple as using one of the pre-made templates in Quartus. However, properly writing and reading from memory quickly became the biggest hurdle for our project. Making sure the timing was correct on all memory accesses was a major challenge.

Both Verilator and GTKWave were important tools we used during project development. We used Verilator for simulating our project, which let us analyze how our hardware and fix errors much quicker than continuously recompiling in Quartus. We also used GTKWave to trace signals and look at the timing of our project. This helped us fix hard to see timing issues without having to take the time to manually create a timing diagram.

6.2 Future Work

In the future we would like to be able to fully integrate our project to run on the live FOREX market. This would involve modifying our project to be able to receive live FOREX data and to be able to act on arbitrage opportunities when they arise.

Unfortunately, we did not have an opportunity to test with live data because real-time FOREX data is hard to get without paying for access to it. We instead opted to use historical data stored in a .csv file and stream it over to the FPGA. To work in a live environment, we will have to modify our software to pull data directly from a live stream before sending it to the FPGA as opposed to reading from a file. This should not be too difficult and most FOREX brokers provide APIs to do this.

We also need to change our program to let it act on arbitrage opportunities. Currently we only output the appropriate currency cycle to the screen by using the hardware to send data over the VGA port. In addition to this, we also want our project to be able to send buy and sell requests to a broker. We would probably need to modify the FPGA to send cycles to the software via the serial or USB ports. The software could then interface with a trader's API and make trades to take advantage of the arbitrage opportunity.

Our project performs fairly quickly, but there is still room for improvement. We believe that we can still cut down on the amount of cycles our program needs to run, which will give a small boost in performance. More importantly, we can get greater speeds by parallelizing the Bellman-Ford algorithm. Although we initially based our project on a paper parallelizing Bellman-Ford on a FPGA, our own implementation ended up mostly serial. A parallel Bellman-Ford algorithm would allow us to take greater advantage of running the project on a FPGA.

7. Conclusion

Our final implementation of the Bellman-Ford Algorithm and cycle detection was considerably different than our initial design. We thought we could modify an existing design that used off-chip memory, to work with Quartus' SRAM templates, but this change required us to completely reconsider our design. In the end though, we are able to successfully detect negative weight cycles and, in theory, arbitrage opportunities. With just a little bit of tweaking our project should be able to run on the live FOREX market.

8. References

1. Fundamental-reading about high frequency trading https://en.wikipedia.org/wiki/High-frequency_trading
2. Discussion of different types of arbitrage https://en.wikipedia.org/wiki/Arbitrage
3. Bellman-Ford implementation on FPGA. Accelerating Large-Scale Single-Source Shortest Path on FPGA
4. StackOverflow discussion that explains some of the theory behind calculating triangle arbitrage http://stackoverflow.com/questions/2282427/interesting-problem-currency-arbitrage
5. Verilator: converts Verilog code to C++ for easy simulation http://www.veripool.org/wiki/verilator
6. GTKWave: tool to visualize timing diagram generated by Verilator http://gtkwave.sourceforge.net
7. Quartus: Altera's proprietary development suite http://altera.com

9. Code Appendix

9.1 Hardware

AdjMat.sv

```
// Quartus II Verilog Template
// Single port RAM with single read/write address
`include "Const.vh"

module AdjMat
#(parameter DATA_WIDTH=`WEIGHT_WIDTH, parameter ADDR_WIDTH=(2*`PRED_WIDTH+1))
(
    input [DATA_WIDTH:0] data,
    input [`PRED_WIDTH:0] row_addr,
    input [`PRED_WIDTH:0] col_addr,
    input we, clk,
    output [DATA_WIDTH:0] q
);

    logic [ADDR_WIDTH:0] addr;
    assign addr = row_addr*`NODES + col_addr;
    // Declare the RAM variable
    reg [DATA_WIDTH:0] ram[2**ADDR_WIDTH:0];

    // Variable to hold the registered read address
    reg [ADDR_WIDTH:0] addr_reg;

    always @ (posedge clk)
    begin
        // Write
        if (we)
            ram[addr] <= data;

        addr_reg <= addr;
    end

    // Continuous assignment implies read returns NEW data.
    // This is the natural behavior of the TriMatrix memory
    // blocks in Single Port mode.
    assign q = ram[addr_reg];

endmodule
```

Bellman.sv

```
`include "Const.vh"

module Bellman(input logic clk, bellman_reset,
              input logic [`PRED_WIDTH:0] src,
              /*Vertmat/Adjmat Read Inputs*/
              input logic [`VERT_WIDTH:0] vertmat_q_a,
              input logic [`VERT_WIDTH:0] vertmat_q_b,
              input logic [`WEIGHT_WIDTH:0] adjmat_q,
              /*VertMat Memory*/
              output logic [`VERT_WIDTH:0] vertmat_data_a,
              output logic [`VERT_WIDTH:0] vertmat_data_b,
              output logic [`PRED_WIDTH:0] vertmat_addr_a,
              output logic [`PRED_WIDTH:0] vertmat_addr_b,
              output logic vertmat_we_a,
              output logic vertmat_we_b,
            /*AdjMat Memory*/
            output logic [`PRED_WIDTH:0] adjmat_row_addr,
            output logic [`PRED_WIDTH:0] adjmat_col_addr,
            output logic bellman_done);

  enum logic [3:0] {SETUP, READ, IDLE, WRITE, DONE} state;
  logic [`PRED_WIDTH:0] i, j, k; //Indices
  logic signed [`WEIGHT_WIDTH:0] svw, dvw; //Source Vertex Weight, Destination Vertex Weight,
Signed
  logic signed [`WEIGHT_WIDTH:0] e; //Edge Weight, Signed

  assign adjmat_row_addr = i;
  assign adjmat_col_addr = j;
  assign e = adjmat_q;
  assign svw = vertmat_q_a[`WEIGHT_WIDTH:0];
  assign dvw = vertmat_q_b[`WEIGHT_WIDTH:0];

  always_comb begin
    vertmat_we_a = 0;
    vertmat_addr_a = 0;
    vertmat_addr_b = 0;
    vertmat_data_a = 0;
    vertmat_we_b = 0;
    vertmat_data_b = 0;
    case (state)
      SETUP: begin
        vertmat_we_a = 1;
        vertmat_addr_a = k;
        if (k + 1 == `NODES) ;
        else if (k == src) begin
          vertmat_data_a[`WEIGHT_WIDTH:0] = 0;
        end else begin
          vertmat_data_a[`WEIGHT_WIDTH:0] = 31'h777fffff; //INT MAX; THIS WILL CHANGE WITH WIDTH
        end
      end
      READ: begin
        vertmat_addr_a = i;
        vertmat_addr_b = j;
      end
      IDLE: begin
        vertmat_addr_a = i;
        vertmat_addr_b = j;
      end
      WRITE:begin
        vertmat_addr_a = i;
        vertmat_addr_b = j;
        if (e != 0 && $signed(svw + e) < $signed(dvw)) begin
          vertmat_we_b = 1;
          vertmat_data_b = {1'b0,i, $signed(svw + e)};
        end
      end
      default: ;
    endcase
  end

  always_ff @(posedge clk) begin
    if (bellman_reset) begin
```

```
      i <= 0;
      j <= 0;
      k <= 0;
      bellman_done <= 0;
      state <= SETUP;
    end else case (state)
      SETUP: begin
        if (k + 1 == `NODES) begin
                k <= 0; //V Iterations below
          state <= WRITE;
        end else if (k == src) begin
          k <= k + 1;
          state <= SETUP;
        end else begin
          k <= k + 1;
          state <= SETUP;
        end
      end
      READ: begin
        if (j + 1 == `NODES && i + 1 == `NODES && k + 1 == `NODES) begin //V Times
          state <= DONE;
        end else if (j + 1 == `NODES && i + 1 == `NODES) begin
          i <= 0;
          j <= 0;
          k <= k + 1;
          state <= IDLE;
        end else if (j + 1 == `NODES) begin
          i <= i + 1;
          j <= 0;
          state <= IDLE;
        end else begin
          j <= j + 1;
          state <= IDLE;
        end
      end
      IDLE: state <= WRITE;
      WRITE: state <= READ;
      DONE: bellman_done <= 1;
      default: state <= DONE;
    endcase
  end
endmodule
```

Character.v

```
// megafunction wizard: %ROM: 1-PORT%
// GENERATION: STANDARD
// VERSION: WM1.0
// MODULE: altsyncram

// ============================================================
// File Name: Character.v
// Megafunction Name(s):
//         altsyncram
//
// Simulation Library Files(s):
```

```
//          altera_mf
// ============================================================
// ************************************************************
// THIS IS A WIZARD-GENERATED FILE. DO NOT EDIT THIS FILE!
//
// 13.1.1 Build 166 11/26/2013 SJ Full Version
// ************************************************************
...

// synopsys translate_off
`timescale 1 ps / 1 ps
// synopsys translate_on
module Character (
    address,
    clock,
    q);

    input   [13:0]  address;
    input     clock;
    output  [0:0]  q;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_off
`endif
    tri1      clock;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_on
`endif

    wire [0:0] sub_wire0;
    wire [0:0] q = sub_wire0[0:0];

    altsyncram  altsyncram_component (
                .address_a (address),
                .clock0 (clock),
                .q_a (sub_wire0),
                .aclr0 (1'b0),
                .aclr1 (1'b0),
                .address_b (1'b1),
                .addressstall_a (1'b0),
                .addressstall_b (1'b0),
                .byteena_a (1'b1),
                .byteena_b (1'b1),
                .clock1 (1'b1),
                .clocken0 (1'b1),
                .clocken1 (1'b1),
                .clocken2 (1'b1),
                .clocken3 (1'b1),
                .data_a (1'b1),
                .data_b (1'b1),
                .eccstatus (),
                .q_b (),
                .rden_a (1'b1),
                .rden_b (1'b1),
                .wren_a (1'b0),
                .wren_b (1'b0));
    defparam
        altsyncram_component.address_aclr_a = "NONE",
        altsyncram_component.clock_enable_input_a = "BYPASS",
        altsyncram_component.clock_enable_output_a = "BYPASS",
        altsyncram_component.init_file = "character.mif",
        altsyncram_component.intended_device_family = "Cyclone V",
        altsyncram_component.lpm_hint = "ENABLE_RUNTIME_MOD=NO",
        altsyncram_component.lpm_type = "altsyncram",
        altsyncram_component.numwords_a = 16384,
        altsyncram_component.operation_mode = "ROM",
        altsyncram_component.outdata_aclr_a = "NONE",
        altsyncram_component.outdata_reg_a = "CLOCK0",
        altsyncram_component.widthad_a = 14,
        altsyncram_component.width_a = 1,
        altsyncram_component.width_byteena_a = 1;


endmodule
```

Const.vh

```
`ifndef _CONST_VH_
`define _CONST_VH_

`define NODES 7
/*1 less than width*/
`define VERT_WIDTH 39
`define WEIGHT_WIDTH 31
`define PRED_WIDTH 6

`endif
```

Container.sv

```
`include "Const.vh"
module Container(input logic clk, container_reset,
                 input logic [`PRED_WIDTH:0] src,
                           input logic [`PRED_WIDTH:0] u_src,
                           input logic [`PRED_WIDTH:0] u_dst,
                           input logic [`WEIGHT_WIDTH:0] u_e,
                 output logic [5:0] frame_char,
                 output logic [5:0] frame_x,
                 output logic [5:0] frame_y,
                 output logic frame_we,
                 output logic container_done,
                 output logic [31:0] test);

  enum logic [4:0] {UPDATE_FOR, UPDATE_REV, RUN_BELLMAN, RUN_CYCLE_DETECT, RUN_PRINT_CYCLE, DONE,
IDLE} state, next_state;
  logic bellman_done, cycle_done, bellman_reset, cycle_reset, print_reset, print_done; //Reset and
done registers

  //Memory
  /*Vertmat/Adjmat Read Outputs*/
  logic [`VERT_WIDTH:0] vertmat_q_a;
  logic [`VERT_WIDTH:0] vertmat_q_b;
  logic [`WEIGHT_WIDTH:0] adjmat_q;
  /*VertMat Memory*/
  logic [`VERT_WIDTH:0] vertmat_data_a;
  logic [`VERT_WIDTH:0] vertmat_data_b;
  logic [`PRED_WIDTH:0] vertmat_addr_a; //Both write
  logic [`PRED_WIDTH:0] vertmat_addr_b; //Both write
  logic vertmat_we_a;
  logic vertmat_we_b;
  /*AdjMat Memory*/
  logic [`WEIGHT_WIDTH:0] adjmat_data;
  logic [`PRED_WIDTH:0] adjmat_row_addr; //Both write
  logic [`PRED_WIDTH:0] adjmat_col_addr; //Both write
  logic adjmat_we;
  /*Memory specific module vars*/
  logic [`PRED_WIDTH:0] bellman_vertmat_addr_a;
  logic [`PRED_WIDTH:0] bellman_vertmat_addr_b;
  logic [`PRED_WIDTH:0] bellman_adjmat_row_addr;
  logic [`PRED_WIDTH:0] bellman_adjmat_col_addr;
  logic [`VERT_WIDTH:0] bellman_vertmat_data_b;
  logic bellman_vertmat_we_b;

  logic [`PRED_WIDTH:0] cycle_vertmat_addr_a;
  logic [`PRED_WIDTH:0] cycle_vertmat_addr_b;
  logic [`PRED_WIDTH:0] cycle_adjmat_row_addr;
  logic [`PRED_WIDTH:0] cycle_adjmat_col_addr;
  logic [`VERT_WIDTH:0] cycle_vertmat_data_b;
  logic cycle_vertmat_we_b;

  logic [`PRED_WIDTH:0] print_vertmat_addr_b;
  logic [`VERT_WIDTH:0] print_vertmat_data_b;
  logic print_vertmat_we_b;

  Bellman bellman(.vertmat_addr_a(bellman_vertmat_addr_a), .vertmat_addr_b(bellman_vertmat_addr_b),
                  .vertmat_data_b(bellman_vertmat_data_b), .vertmat_we_b(bellman_vertmat_we_b),
                  .adjmat_row_addr(bellman_adjmat_row_addr),
.adjmat_col_addr(bellman_adjmat_col_addr), .*);
  CycleDetect cycle_detect(.vertmat_addr_a(cycle_vertmat_addr_a),
.vertmat_addr_b(cycle_vertmat_addr_b),
                          .vertmat_data_b(cycle_vertmat_data_b), .vertmat_we_b(cycle_vertmat_we_b),
                          .adjmat_row_addr(cycle_adjmat_row_addr),
.adjmat_col_addr(cycle_adjmat_col_addr), .*);
  PrintCycle print_cycle(.vertmat_addr_b(print_vertmat_addr_b),
.vertmat_data_b(print_vertmat_data_b),
                          .vertmat_we_b(print_vertmat_we_b),.*);
  VertMat vertmat(.data_a(vertmat_data_a), .data_b(vertmat_data_b),
                  .addr_a(vertmat_addr_a), .addr_b(vertmat_addr_b),
                  .we_a(vertmat_we_a), .we_b(vertmat_we_b),
                  .q_a(vertmat_q_a), .q_b(vertmat_q_b), .*);
  AdjMat adjmat(.data(adjmat_data), .row_addr(adjmat_row_addr), .col_addr(adjmat_col_addr),
                     .we(adjmat_we), .q(adjmat_q), .*);

  always_ff @(posedge clk) begin
    if (container_reset) begin
      state <= UPDATE_FOR;
      container_done <= 0;
    end else case (state)
        UPDATE_FOR: state <= UPDATE_REV;
      UPDATE_REV: begin
        bellman_reset <= 1;
        state <= IDLE;
        next_state <= RUN_BELLMAN;
      end
      RUN_BELLMAN: begin
        bellman_reset <= 0;
        if (bellman_done) begin
          cycle_reset <= 1;
          state <= IDLE;
          next_state <= RUN_CYCLE_DETECT;
        end
      end
      IDLE: state <= next_state;
      RUN_CYCLE_DETECT: begin
        cycle_reset <= 0;
        if (cycle_done) begin
          print_reset <= 1;
          next_state <= RUN_PRINT_CYCLE;
          state <= IDLE;
        end
      end
      RUN_PRINT_CYCLE: begin
        print_reset <= 0;
        if (print_done) state <= DONE;
      end
      DONE: container_done <= 1;
      default: state <= DONE;
    endcase
  end

  always_comb begin
    adjmat_we = 0;
    adjmat_data = 0;
    adjmat_row_addr = 0;
    adjmat_col_addr = 0;
    adjmat_data = 0;
    vertmat_addr_a = 0;
    vertmat_addr_b = 0;
    vertmat_data_b = 0;
    vertmat_we_b = 0;
    case (state)
      UPDATE_FOR: begin
        adjmat_we = 1;
        adjmat_data = u_e;
        adjmat_row_addr = u_src;
        adjmat_col_addr = u_dst;
      end
      UPDATE_REV: begin
        adjmat_we = 1;
        adjmat_data = 0;//-1*u_e;
        adjmat_row_addr = u_dst;
        adjmat_col_addr = u_src;
      end
      RUN_BELLMAN: begin
        adjmat_we = 0;
        adjmat_row_addr = bellman_adjmat_row_addr;
        adjmat_col_addr = bellman_adjmat_col_addr;
        vertmat_addr_a = bellman_vertmat_addr_a;
        vertmat_addr_b = bellman_vertmat_addr_b;
```

```
        vertmat_data_b = bellman_vertmat_data_b;
        vertmat_we_b = bellman_vertmat_we_b;
      end
      RUN_CYCLE_DETECT: begin
        adjmat_row_addr = cycle_adjmat_row_addr;
        adjmat_col_addr = cycle_adjmat_col_addr;
        vertmat_addr_a = cycle_vertmat_addr_a;
        vertmat_addr_b = cycle_vertmat_addr_b;
        vertmat_data_b = cycle_vertmat_data_b;
        vertmat_we_b = cycle_vertmat_we_b;
      end
      RUN_PRINT_CYCLE: begin
        vertmat_addr_b = print_vertmat_addr_b;
        vertmat_data_b = print_vertmat_data_b;
        vertmat_we_b = print_vertmat_we_b;
      end
      default: ;
    endcase
  end

endmodule
```

Container.sv pretty print

```
`include "Const.vh"
module Container(input logic clk, container_reset,
                 input logic [`PRED_WIDTH:0] src,
                           input logic [`PRED_WIDTH:0] u_src,
                           input logic [`PRED_WIDTH:0] u_dst,
                           input logic [`WEIGHT_WIDTH:0] u_e,
                 output logic [5:0] frame_char,
                 output logic [5:0] frame_x,
                 output logic [5:0] frame_y,
                 output logic [5:0] line_start,
                 output logic [5:0] line_end,
                 output logic frame_we,
                 output logic container_done,
                 output logic [31:0] test);

  enum logic [4:0] {UPDATE_FOR, UPDATE_REV, RUN_BELLMAN, RUN_CYCLE_DETECT, RUN_PRINT_CYCLE, DONE,
IDLE} state, next_state;
  logic bellman_done, cycle_done, bellman_reset, cycle_reset, print_reset, print_done; //Reset and
done registers

  //Memory
  /*Vertmat/Adjmat Read Outputs*/
  logic [`VERT_WIDTH:0] vertmat_q_a;
  logic [`VERT_WIDTH:0] vertmat_q_b;
  logic [`WEIGHT_WIDTH:0] adjmat_q;
  /*VertMat Memory*/
  logic [`VERT_WIDTH:0] vertmat_data_a;
  logic [`VERT_WIDTH:0] vertmat_data_b;
  logic [`PRED_WIDTH:0] vertmat_addr_a; //Both write
  logic [`PRED_WIDTH:0] vertmat_addr_b; //Both write
  logic vertmat_we_a;
  logic vertmat_we_b;
  /*AdjMat Memory*/
  logic [`WEIGHT_WIDTH:0] adjmat_data;
  logic [`PRED_WIDTH:0] adjmat_row_addr; //Both write
  logic [`PRED_WIDTH:0] adjmat_col_addr; //Both write
  logic adjmat_we;
  /*Memory specific module vars*/
  logic [`PRED_WIDTH:0] bellman_vertmat_addr_a;
  logic [`PRED_WIDTH:0] bellman_vertmat_addr_b;
  logic [`PRED_WIDTH:0] bellman_adjmat_row_addr;
  logic [`PRED_WIDTH:0] bellman_adjmat_col_addr;
  logic [`VERT_WIDTH:0] bellman_vertmat_data_b;
  logic bellman_vertmat_we_b;

  logic [`PRED_WIDTH:0] cycle_vertmat_addr_a;
  logic [`PRED_WIDTH:0] cycle_vertmat_addr_b;
  logic [`PRED_WIDTH:0] cycle_adjmat_row_addr;
  logic [`PRED_WIDTH:0] cycle_adjmat_col_addr;
  logic [`VERT_WIDTH:0] cycle_vertmat_data_b;
  logic cycle_vertmat_we_b;

  logic [`PRED_WIDTH:0] print_vertmat_addr_b;
  logic [`VERT_WIDTH:0] print_vertmat_data_b;
  logic print_vertmat_we_b;

  Bellman bellman(.vertmat_addr_a(bellman_vertmat_addr_a), .vertmat_addr_b(bellman_vertmat_addr_b),
                  .vertmat_data_b(bellman_vertmat_data_b), .vertmat_we_b(bellman_vertmat_we_b),
                  .adjmat_row_addr(bellman_adjmat_row_addr),
.adjmat_col_addr(bellman_adjmat_col_addr), .*);
  CycleDetect cycle_detect(.vertmat_addr_a(cycle_vertmat_addr_a),
.vertmat_addr_b(cycle_vertmat_addr_b),
                          .vertmat_data_b(cycle_vertmat_data_b), .vertmat_we_b(cycle_vertmat_we_b),
                          .adjmat_row_addr(cycle_adjmat_row_addr),
.adjmat_col_addr(cycle_adjmat_col_addr), .*);
  PrintCycle print_cycle(.vertmat_addr_b(print_vertmat_addr_b),
.vertmat_data_b(print_vertmat_data_b),
                          .vertmat_we_b(print_vertmat_we_b),.*);
  VertMat vertmat(.data_a(vertmat_data_a), .data_b(vertmat_data_b),
                  .addr_a(vertmat_addr_a), .addr_b(vertmat_addr_b),
                  .we_a(vertmat_we_a), .we_b(vertmat_we_b),
                  .q_a(vertmat_q_a), .q_b(vertmat_q_b), .*);
  AdjMat adjmat(.data(adjmat_data), .row_addr(adjmat_row_addr), .col_addr(adjmat_col_addr),
                   .we(adjmat_we), .q(adjmat_q), .*);

  always_ff @(posedge clk) begin
    if (container_reset) begin
      state <= UPDATE_FOR;
      container_done <= 0;
    end else case (state)
        UPDATE_FOR: state <= UPDATE_REV;
      UPDATE_REV: begin
        bellman_reset <= 1;
        state <= IDLE;
        next_state <= RUN_BELLMAN;
      end
      RUN_BELLMAN: begin
        bellman_reset <= 0;
        if (bellman_done) begin
          cycle_reset <= 1;
          state <= IDLE;
          next_state <= RUN_CYCLE_DETECT;
        end
      end
      IDLE: state <= next_state;
      RUN_CYCLE_DETECT: begin
        cycle_reset <= 0;
        if (cycle_done) begin
          print_reset <= 1;
          next_state <= RUN_PRINT_CYCLE;
          state <= IDLE;
        end
      end
      RUN_PRINT_CYCLE: begin
        print_reset <= 0;
        if (print_done) state <= DONE;
      end
      DONE: container_done <= 1;
      default: state <= DONE;
    endcase
  end

  always_comb begin
    adjmat_we = 0;
    adjmat_data = 0;
    adjmat_row_addr = 0;
    adjmat_col_addr = 0;
    adjmat_data = 0;
    vertmat_addr_a = 0;
    vertmat_addr_b = 0;
    vertmat_data_b = 0;
    vertmat_we_b = 0;
    case (state)
      UPDATE_FOR: begin
        adjmat_we = 1;
        adjmat_data = u_e;
        adjmat_row_addr = u_src;
        adjmat_col_addr = u_dst;
      end
      UPDATE_REV: begin
        adjmat_we = 1;
     //   adjmat_data = 0;//-1*u_e;
       // adjmat_row_addr = u_dst;
        //adjmat_col_addr = u_src;
      end
      RUN_BELLMAN: begin
        adjmat_we = 0;
        adjmat_row_addr = bellman_adjmat_row_addr;
        adjmat_col_addr = bellman_adjmat_col_addr;
        vertmat_addr_a = bellman_vertmat_addr_a;
        vertmat_addr_b = bellman_vertmat_addr_b;
        vertmat_data_b = bellman_vertmat_data_b;
        vertmat_we_b = bellman_vertmat_we_b;
      end
      RUN_CYCLE_DETECT: begin
        adjmat_row_addr = cycle_adjmat_row_addr;
        adjmat_col_addr = cycle_adjmat_col_addr;
        vertmat_addr_a = cycle_vertmat_addr_a;
```

```
        vertmat_addr_b = cycle_vertmat_addr_b;
        vertmat_data_b = cycle_vertmat_data_b;
        vertmat_we_b = cycle_vertmat_we_b;
      end
      RUN_PRINT_CYCLE: begin
        vertmat_addr_b = print_vertmat_addr_b;
        vertmat_data_b = print_vertmat_data_b;
        vertmat_we_b = print_vertmat_we_b;
      end
      default: ;
    endcase
  end

endmodule
```

CycleDetect.sv

```
`include "Const.vh"

module CycleDetect(input logic clk, cycle_reset,
                   /*Vertmat/Adjmat Read Inputs*/
                   input logic [`VERT_WIDTH:0] vertmat_q_a,
                   input logic [`VERT_WIDTH:0] vertmat_q_b,
                               input logic [`WEIGHT_WIDTH:0] adjmat_q,
                   /*VertMat Memory*/
                   output logic [`PRED_WIDTH:0] vertmat_addr_a,
                   output logic [`PRED_WIDTH:0] vertmat_addr_b,
                   output logic [`VERT_WIDTH:0] vertmat_data_b,
                   output logic vertmat_we_b,
                   /*AdjMat Memory*/
                   output logic [`PRED_WIDTH:0] adjmat_row_addr,
                   output logic [`PRED_WIDTH:0] adjmat_col_addr,
                   output logic cycle_done);

      enum logic [3:0] {READ, IDLE, CHECK_CYCLE, READ_CYCLE, WRITE_CYCLE, FINISH_CYCLE, DONE}
state;
      logic [`PRED_WIDTH:0] i, j, k, l; //Indices
      logic signed [`WEIGHT_WIDTH:0] svw, dvw; //Source Vertex Weight, Destination Vertex Weight,
Signed
      logic signed [`WEIGHT_WIDTH:0] e; //Edge Weight, Signed

        assign adjmat_row_addr = i;
        assign adjmat_col_addr = j;
        assign e = adjmat_q;
      assign svw = vertmat_q_a[`WEIGHT_WIDTH:0];
      assign dvw = vertmat_q_b[`WEIGHT_WIDTH:0];
      assign vertmat_addr_a = i;

      always_comb begin
        vertmat_addr_b = 0;
        vertmat_we_b = 0;
        vertmat_data_b = 0;
        case (state)
          READ: vertmat_addr_b = j;
          IDLE: vertmat_addr_b = j;
          CHECK_CYCLE: begin
            vertmat_addr_b = j;
            //if (e != 0 && $signed(svw + e) < $signed(dvw)) begin //Found Negative Weight Cycle
            //  vertmat_addr_b = l;
            //end
          end
          READ_CYCLE: begin
            vertmat_we_b = 0;
            vertmat_addr_b = l;
          end
          WRITE_CYCLE: begin
            vertmat_we_b = 1;
            vertmat_data_b = {1'b1, vertmat_q_b[`VERT_WIDTH - 1:0]};
            vertmat_addr_b = l;
          end
          FINISH_CYCLE: begin
            vertmat_we_b = 0;
            vertmat_addr_b = j;
          end
          default: vertmat_addr_b = j;
        endcase
      end

      always_ff @(posedge clk) begin
        if (cycle_reset) begin
          i <= 0;
          j <= 0;
          k <= -1;
          cycle_done <= 0;
          state <= READ;
        end else case (state)
          READ: begin
            k <= -1;
            if (j + 1 == `NODES && i + 1 == `NODES) begin
              state <= DONE; //All finished looping through edges //GETS TRIGGERED SOMEHOW ON RESET
            end else if (j + 1 == `NODES) begin
              i <= i + 1;
              j <= 0;
              state <= IDLE;
            end else begin
              j <= j + 1;
              state <= IDLE;
            end
          end
          IDLE: begin
            k <= j;
            state <= CHECK_CYCLE;
          end
          CHECK_CYCLE: begin
            if (e != 0 && $signed(svw + e) < $signed(dvw)) begin //Found Negative Weight Cycle
              l <= vertmat_q_b[(`VERT_WIDTH-1):(`WEIGHT_WIDTH+1)];
              state <= READ_CYCLE;
            end else state <= READ;
          end
          READ_CYCLE: begin
            if (l == k) begin //Read Cycle
              state <= FINISH_CYCLE;
            end else state <= WRITE_CYCLE;
          end
```

```
          WRITE_CYCLE: begin
            l <= vertmat_q_b[(`VERT_WIDTH-1):(`WEIGHT_WIDTH+1)];
            state <= READ_CYCLE;
          end
          FINISH_CYCLE: state <= READ;
          DONE: cycle_done <= 1;
          default: state <= DONE;
        endcase
      end
endmodule
```

FOREX.sv

```
/*
 * Avalon memory-mapped peripheral for the VGA LED Emulator
 *
 * Stephen A. Edwards
 * Columbia University
 */
`include "Const.vh"

module FOREX(input logic clk,
                 input logic reset,
                 input logic [`WEIGHT_WIDTH:0] writedata,
                 input logic write,
                 input chipselect,
                 input logic [2:0] address,
                 output logic [7:0] VGA_R, VGA_G, VGA_B,
                 output logic VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_n,
                 output logic VGA_SYNC_n,
                 output logic frame_we,
                 output logic [5:0] frame_char,
                 output logic [31:0] test);

        enum logic [3:0] {RESET, CONTAINER, IDLE} state, next_state;

        logic [`PRED_WIDTH:0] src = 0;
        logic [`PRED_WIDTH:0] u_src;
        logic [`PRED_WIDTH:0] u_dst;
        logic [`WEIGHT_WIDTH:0] u_e;
        logic [5:0] frame_x, frame_y;
        //logic [5:0] frame_char;
        //logic frame_we;

        logic container_done;
        logic container_reset;

        //Frame buffer(.x(frame_x), .y(frame_y), .char(frame_char), .we(frame_we), .*);
        Container container(.*);

        always_ff @(posedge clk) begin
            if (reset) ;
            else if (chipselect && write) begin
                case (address)
                    3'd0 : begin //Write new source and dest
                     u_src <= writedata[2 * `PRED_WIDTH + 1:`PRED_WIDTH + 1];
                     u_dst <= writedata[`PRED_WIDTH:0];
                    end
                    3'd1 : begin
                     u_e <= writedata;
                     state <= RESET;
                    end
                    default: ;
                endcase
            end
            case(state)
                RESET: begin
                    container_reset <= 1;
                    state <= IDLE;
                    next_state <= CONTAINER;
                end
                CONTAINER: container_reset <= 0;
                IDLE: state <= next_state;
                default: ;
            endcase
        end

endmodule
```

FOREX.sv with keyboard support and pretty print

```
/*
 * Avalon memory-mapped peripheral for the VGA LED Emulator
 *
 * Stephen A. Edwards
 * Columbia University
 */
`include "Const.vh"

module FOREX(input logic clk,
                 input logic reset,
                 input logic [`WEIGHT_WIDTH:0] writedata,
                 input logic write,
                 input chipselect,
                 input logic [2:0] address,
                 output logic [7:0] VGA_R, VGA_G, VGA_B,
                 output logic VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_n,
                 output logic VGA_SYNC_n,
                 output logic frame_we,
                 output logic [5:0] frame_char,
                 output logic [31:0] test);

        enum logic [3:0] {RESET, CONTAINER, IDLE} state, next_state;

        logic [`PRED_WIDTH:0] src = 0;
        logic [`PRED_WIDTH:0] u_src;
        logic [`PRED_WIDTH:0] u_dst;
        logic [`WEIGHT_WIDTH:0] u_e;
        logic [5:0] frame_x, frame_y;
        logic [5:0] frame_char;
        logic frame_we;
        logic [2:0] adv;
        logic container_done;
        logic container_reset;

        logic [5:0] line_start, line_end;

        Frame buffer(.x(frame_x), .y(frame_y), .char(frame_char), .we(frame_we), .*);
        Container container(.*);

        always_ff @(posedge clk) begin
            if (reset) ;
            else if (chipselect && write) begin
                case (address)
                    3'd0 : begin //Write new source and dest
                     u_src <= writedata[2 * `PRED_WIDTH + 1:`PRED_WIDTH + 1];
                     u_dst <= writedata[`PRED_WIDTH:0];
                    end
                    3'd1 : begin
                     u_e <= writedata;
                     state <= RESET;
                    end
                    3'd2 : adv <= writedata[2:0];
                    default: ;
                endcase
            end
            case(state)
                RESET: begin
                    container_reset <= 1;
                    state <= IDLE;
                    next_state <= CONTAINER;
                end
                CONTAINER: container_reset <= 0;
                IDLE: state <= next_state;
                default: ;
            endcase
        end

endmodule
```

Frame.sv

```
/*
 * Seven-segment LED emulator
 *
 * Stephen A. Edwards, Columbia University
 */

module Frame(input logic clk, reset,
             input logic [5:0] x, y,
             input logic [5:0] char,
             input logic we,
             output logic [7:0] VGA_R, VGA_G, VGA_B,
             output logic VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_n, VGA_SYNC_n);

    /*
    * 640 X 480 VGA timing for a 50 MHz clock: one pixel every other cycle
    *
    * HCOUNT 1599 0             1279       1599 0
    *             _______________              ________
    * ___________|    Video      |____________|  Video
    *
    *
    * |SYNC| BP |<-- HACTIVE -->|FP|SYNC| BP |<-- HACTIVE
    *       _______________________      _____________
    * |____|       VGA_HS          |____|
    */
   // Parameters for hcount
   parameter HACTIVE      = 11'd 1280,
             HFRONT_PORCH = 11'd 32,
             HSYNC        = 11'd 192,
             HBACK_PORCH  = 11'd 96,
             HTOTAL       = HACTIVE + HFRONT_PORCH + HSYNC + HBACK_PORCH; // 1600

   // Parameters for vcount
   parameter VACTIVE      = 10'd 480,
             VFRONT_PORCH = 10'd 10,
             VSYNC        = 10'd 2,
             VBACK_PORCH  = 10'd 33,
             VTOTAL       = VACTIVE + VFRONT_PORCH + VSYNC + VBACK_PORCH; // 525

   logic [10:0] hcount; // Horizontal counter
                        // Hcount[10:1] indicates pixel column (0-639)
   logic endOfLine;

   always_ff @(posedge clk or posedge reset)
     if (reset) hcount <= 0;
     else if (endOfLine) hcount <= 0;
     else hcount <= hcount + 11'd 1;

   assign endOfLine = hcount == HTOTAL - 1;

   // Vertical counter
   logic [9:0] vcount;
   logic endOfField;

   always_ff @(posedge clk or posedge reset)
     if (reset) vcount <= 0;
     else if (endOfLine)
       if (endOfField) vcount <= 0;
       else vcount <= vcount + 10'd 1;

   assign endOfField = vcount == VTOTAL - 1;

   // Horizontal sync: from 0x520 to 0x5DF (0x57F)
   // 101 0010 0000 to 101 1101 1111
   assign VGA_HS = !( (hcount[10:8] == 3'b101) & !(hcount[7:5] == 3'b111));
   assign VGA_VS = !( vcount[9:1] == (VACTIVE + VFRONT_PORCH) / 2);

   assign VGA_SYNC_n = 1; // For adding sync to video signals; not used for VGA

   // Horizontal active: 0 to 1279     Vertical active: 0 to 479
   // 101 0000 0000  1280          01 1110 0000  480
   // 110 0011 1111  1599          10 0000 1100  524
   assign VGA_BLANK_n = !( hcount[10] & (hcount[9] | hcount[8]) ) &
```

```
              !( vcount[9] | (vcount[8:5] == 4'b1111) );

   /* VGA_CLK is 25 MHz
    *             __    __    __
    * clk    __|  |__|  |__|
    *
    *             _____       __
    * hcount[0]__|     |_____|
    */
    assign VGA_CLK = hcount[0]; // 25 MHz clock: pixel latched on rising edge

    /* Logic required for drawing characters*/
    logic [10:0] wp;
    logic [10:0] rp;
    logic [5:0] num_char;
    logic [5:0] frame_buffer[1199:0]; //40 accross by 30 down

    logic [13:0] char_addr;
    logic char_show;

    assign wp = 40 * y + x;
    assign rp = 40 * vcount[9:4] + hcount[10:5];
    assign {VGA_R, VGA_G, VGA_B} = char_show && num_char != 0 ? 24'hff0000 : 24'd0;
    assign num_char = frame_buffer[rp];
    assign char_addr = (hcount[10:1] - hcount[10:5] * 16) + (vcount[9:0] - vcount[9:4] * 16) * 16 +
(num_char - 1) * 256; //Simplify

    Character character(.address(char_addr),.clock(clk),.q(char_show));

    always_ff @(posedge clk) begin
      if (we) frame_buffer[wp] <= char;
    end

endmodule // VGA_LED_Emulator
```

Frame.sv with keyboard support and pretty print

```
/*
 * Seven-segment LED emulator
 *
 * Stephen A. Edwards, Columbia University
 */

module Frame(input logic clk, reset,
             input logic [5:0] x, y,
             input logic [5:0] char,
             input logic we,
             input logic [2:0] adv,
             input logic [5:0] line_start,
             input logic [5:0] line_end,
             output logic [7:0] VGA_R, VGA_G, VGA_B,
             output logic VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_n, VGA_SYNC_n);

    /*
    * 640 X 480 VGA timing for a 50 MHz clock: one pixel every other cycle
    *
    * HCOUNT 1599 0             1279       1599 0
    *             _______________              ________
    * ___________|    Video      |____________|  Video
    *
    *
    * |SYNC| BP |<-- HACTIVE -->|FP|SYNC| BP |<-- HACTIVE
    *       _______________________      _____________
    * |____|       VGA_HS          |____|
    */
   // Parameters for hcount
   parameter HACTIVE      = 11'd 1280,
             HFRONT_PORCH = 11'd 32,
             HSYNC        = 11'd 192,
             HBACK_PORCH  = 11'd 96,
             HTOTAL       = HACTIVE + HFRONT_PORCH + HSYNC + HBACK_PORCH; // 1600

   // Parameters for vcount
   parameter VACTIVE      = 10'd 480,
             VFRONT_PORCH = 10'd 10,
             VSYNC        = 10'd 2,
             VBACK_PORCH  = 10'd 33,
             VTOTAL       = VACTIVE + VFRONT_PORCH + VSYNC + VBACK_PORCH; // 525

   logic [10:0] hcount; // Horizontal counter
                        // Hcount[10:1] indicates pixel column (0-639)
   logic endOfLine;

   always_ff @(posedge clk or posedge reset)
     if (reset) hcount <= 0;
     else if (endOfLine) hcount <= 0;
     else hcount <= hcount + 11'd 1;

   assign endOfLine = hcount == HTOTAL - 1;

   // Vertical counter
   logic [9:0] vcount;
   logic endOfField;

   always_ff @(posedge clk or posedge reset)
     if (reset) vcount <= 0;
     else if (endOfLine)
       if (endOfField) vcount <= 0;
       else vcount <= vcount + 10'd 1;

   assign endOfField = vcount == VTOTAL - 1;

   // Horizontal sync: from 0x520 to 0x5DF (0x57F)
   // 101 0010 0000 to 101 1101 1111
   assign VGA_HS = !( (hcount[10:8] == 3'b101) & !(hcount[7:5] == 3'b111));
   assign VGA_VS = !( vcount[9:1] == (VACTIVE + VFRONT_PORCH) / 2);

   assign VGA_SYNC_n = 1; // For adding sync to video signals; not used for VGA

   // Horizontal active: 0 to 1279     Vertical active: 0 to 479
   // 101 0000 0000  1280          01 1110 0000  480
   // 110 0011 1111  1599          10 0000 1100  524
   assign VGA_BLANK_n = !( hcount[10] & (hcount[9] | hcount[8]) ) &
            !( vcount[9] | (vcount[8:5] == 4'b1111) );
```

```
   /* VGA_CLK is 25 MHz
    *             __    __    __
    * clk    __|  |__|  |__|
    *
    *             _____       __
    * hcount[0]__|     |_____|
    */
    assign VGA_CLK = hcount[0]; // 25 MHz clock: pixel latched on rising edge

    /* Logic required for drawing characters*/
    logic [10:0] wp;
    logic [10:0] rp;
    logic [5:0] num_char;
    logic [5:0] frame_buffer[1199:0]; //40 accross by 30 down
<<<<<<< HEAD
    logic [5:0] mid_frame_buffer[439:0]; // 40*11
=======
>>>>>>> 9ec8d5612d16fa1250b92a8d96b1ed7225904e55

    logic [13:0] char_addr;
    logic char_show;

    logic [5:0] X,Y;
    assign Y = vcount[9:4];
    assign X = hcount[10:5];
    assign wp = 40 * y + x;
    assign rp = 40 * vcount[9:4] + hcount[10:5];

//    assign {VGA_R, VGA_G, VGA_B} = char_show && num_char != 0 ? 24'hff0000 : 24'd0;
//    assign num_char = frame_buffer[rp];

    assign char_addr = (hcount[10:1] - hcount[10:5] * 16) + (vcount[9:0] - vcount[9:4] * 16) * 16 +
(num_char - 1) * 256; //Simplify

    Character character(.address(char_addr),.clock(clk),.q(char_show));

<<<<<<< HEAD
    logic [5:0] y_out;
    logic [5:0] y_record_out;
    logic [3:0] size;
    logic [2:0] stateFlag; // 3'd1: Mid_Forward state, 3'd2: Mid_Idler state, 3'd3: Mid_Back state,
    logic [9:0] mid_buffer_pos;
    logic [9:0] buffer_pos;

    logic [23:0] red = 24'hff0000;
    logic [23:0] blue = 24'h0000ff;
    logic [23:0] white = 24'hffffff;
    logic [23:0] char_color;

    FanShow fanshow (.bottom(6'd29), .*);

    logic [5:0] number_char_1;
     logic [5:0] number_char_2;
    logic [5:0] name_char_1;
    logic [5:0] name_char_2;
    logic [5:0] name_char_3;
```

```
Translation translation (.*);

enum logic [4:0] {Normal, Mid_Forward, Mid_Translation, Mid_Show, Mid_Back} state;

always_ff @(posedge clk) begin
    if (reset)  begin
        state <= Normal ;
        buffer_pos<=0;
        mid_buffer_pos<=0;
        number_char_1 <=0;
            number_char_2 <=0;
    end
else case (state)
Normal : begin
    if(stateFlag==1) state <= Mid_Forward;
    else state <= Normal;
end
Mid_Forward : begin

    if(stateFlag==2) begin
        state <= Mid_Translation;
        number_char_1<=frame_buffer[y_record_out*40+buffer_pos];
            number_char_2<=frame_buffer[y_record_out*40+buffer_pos+1];
    end
    else state <= Mid_Forward;
end
Mid_Translation : begin
    if(frame_buffer[y_record_out*40+buffer_pos]!=0&&buffer_pos<40) begin
        mid_frame_buffer[mid_buffer_pos]<=name_char_1;
        mid_frame_buffer[mid_buffer_pos+1]<=name_char_2;
        mid_frame_buffer[mid_buffer_pos+2]<=name_char_3;

            if(frame_buffer[y_record_out*40+buffer_pos+2]!=0) begin
                mid_frame_buffer[mid_buffer_pos+3]<=37;
                buffer_pos<=buffer_pos+3;
                mid_buffer_pos<=mid_buffer_pos+4;
                state<=Mid_Translation;
                number_char_1<=frame_buffer[y_record_out*40+buffer_pos+3];
                      number_char_2<=frame_buffer[y_record_out*40+buffer_pos+4];
            end
            else begin

                mid_buffer_pos<=mid_buffer_pos+3;
                buffer_pos<=buffer_pos+2;
            end
    end
    else begin
        if(mid_buffer_pos<440) begin
            mid_frame_buffer[mid_buffer_pos]<=0;
            mid_buffer_pos<=mid_buffer_pos+1;
        end
        else begin
            state<= Mid_Show;
            mid_buffer_pos<=0;
            buffer_pos<=0;
        end
    end
end
```

```
    Mid_Show : begin
        if(stateFlag==3) state <= Mid_Back;
        else state <= Mid_Show;
    end
    Mid_Back : begin
        if(stateFlag==4) state <= Normal;
        else state <= Mid_Back;
    end
     default: ;
    endcase
    end

    always_comb begin
        if(line_start<=line_end) begin
            if(Y>=line_start&&Y<=line_end) char_color<= red;
            else char_color <= white;
        end
        else begin
            if((Y>=line_start&&Y<=29)||(Y>=0&&Y<=line_end)) char_color<= red;
            else char_color <= white;
        end
    end

    always_comb begin
        case (state)
            Normal: begin
                num_char = frame_buffer[rp];
                if(Y==y_out) {VGA_R, VGA_G, VGA_B} = char_show && num_char != 0 ? char_color :
24'h404040;
                else  {VGA_R, VGA_G, VGA_B} = char_show && num_char != 0 ? char_color : 24'h000000;
            end
            Mid_Forward : begin
                     num_char = 0;
                if(Y<=y_out+size&&Y>=y_out-size) {VGA_R, VGA_G, VGA_B}=24'h404040;
                else {VGA_R, VGA_G, VGA_B}=24'h000000;
            end
            Mid_Show : begin
                if(Y<=y_out+size&&Y>=y_out-size) begin
                num_char = mid_frame_buffer[(Y-(y_out-size))*40+X];
                {VGA_R, VGA_G, VGA_B} = char_show && num_char != 0 ? blue : 24'h404040;
                end else begin
                            {VGA_R, VGA_G, VGA_B}=24'h000000;
                            num_char=0;
                            end
            end
            Mid_Back : begin
                      num_char = 0;
                if(Y<=y_out+size&&Y>=y_out-size) {VGA_R, VGA_G, VGA_B}=24'h404040;
                else {VGA_R, VGA_G, VGA_B}=24'h000000;
            end
                default : begin
                    num_char=0;
                    {VGA_R, VGA_G, VGA_B}=24'h000000;
                    end
        endcase
    end

=======
>>>>>>> 9ec8d5612d16fa1250b92a8d96b1ed7225904e55
    always_ff @(posedge clk) begin
      if (we) frame_buffer[wp] <= char;
    end

endmodule // VGA_LED_Emulator


module FanShow (
    input logic [5:0] bottom,
    input logic clk, reset,
    input logic [2:0] adv, // 3'd0 0, 3'd1 -1, 3'd2 +1, 3'd3 enter, 3'd4 esc
    output logic [5:0] y_out,
    output logic [5:0] y_record_out,
    output logic [3:0] size,
    output logic [2:0] stateFlag // 3'd1: Mid_Forward state, 3'd2: Mid_Idler state, 3'd3: Mid_Back
state, 3'd4: Mid_Back finish
    );

logic [5:0] y_record;
assign y_record_out = y_record;
logic [5:0] y_pos;
assign y_out = y_pos;

logic [23:0] clk_count;

enum logic [4:0] {Initial, Idler, Plus, Minus, Mid, Mid_Idler, Mid_Forward, Mid_Back, Image} state;

    always_ff @(posedge clk)
    if (reset)  state <= Initial ;
    else case (state)
    Initial: begin
    y_pos <= 6'd0;
    y_record <= 6'd0;
    size <= 4'd0;
    stateFlag <= 3'd0;
    clk_count <= 10'd0;
    state <= Idler;
    end
    Idler: begin
    if (y_pos == 6'd0) begin
    if (adv == 3'd0 || adv == 3'd4 || adv == 3'd1) begin
    state <= Idler;
    end else if (adv == 3'd2) begin
    state <= Plus;
    end else if (adv == 3'd3) begin
    state <= Mid;
    end
    end else if (y_pos == bottom) begin
    if (adv == 3'd0 || adv == 3'd4 || adv == 3'd2) begin
    state <= Idler;
    end else if (adv == 3'd1) begin
    state <= Minus;
    end else if (adv == 3'd3) begin
    state <= Mid;
    end
    end else begin
    if (adv == 3'd1) state <=  Minus;
```

```
else if (adv == 3'd2) state <= Plus;
else if (adv == 3'd3) state <= Mid;
else state <= Idler;
end
end
Plus: begin
y_pos <= y_pos +1;
state <= Image;
end
Minus: begin
y_pos <= y_pos - 1;
state <= Image;
end
Image: begin
if (adv == 3'd0)
state <= Idler;
else state <= Image;
end
Mid: begin
y_record <= y_pos;
stateFlag <= 3'd1;
state <= Mid_Forward;
end
Mid_Forward: begin
if(clk_count<24'd500000) begin
clk_count <= clk_count+1;
state <= Mid_Forward;
end
else begin
clk_count <= 0;
if (y_pos < 6'd15) begin
y_pos <= y_pos +1;
if(size<4'd5) size <= size +1;
state <= Mid_Forward;
end else if (y_pos > 6'd15) begin
y_pos <= y_pos -1;
if(size<4'd5) size <= size +1;
state <= Mid_Forward;
end else if (y_pos == 6'd15) begin
if(size<4'd5) begin
size <= size +1;
state <= Mid_Forward;
end
else begin
state <= Mid_Idler;
stateFlag <=3'd2;
end
end
end
end
Mid_Idler: begin
if (adv == 3'd4) begin
state <= Mid_Back;
stateFlag <= 3'd3;
end else begin
state <= Mid_Idler;
end
end
```

```
    Mid_Back: begin
    if(clk_count<24'd500000) begin
    clk_count <= clk_count+1;
    state <= Mid_Back;
    end
    else begin
    clk_count <= 0;
    if (y_pos < y_record) begin
    y_pos <= y_pos + 1;
    if(size>4'd0) size <= size -1;
    state <= Mid_Back;
    end else if (y_pos > y_record) begin
    y_pos <= y_pos - 1;
    if(size>4'd0) size <= size -1;
    state <= Mid_Back;
    end else begin
    if(size>4'd0) begin
    size <= size -1;
    state <= Mid_Back;
    end
    else begin
    state <= Idler;
    stateFlag <= 3'd4;
    end
    end
    end
    end
    default:          state <= Initial;
    endcase
endmodule

module Translation (
    input logic [5:0] number_char_1,
     input logic [5:0] number_char_2,
    output logic [5:0] name_char_1,
    output logic [5:0] name_char_2,
    output logic [5:0] name_char_3);

always_comb begin
    case(number_char_1)
    6'd10: begin  //0
         case(number_char_2)
         6'd10: begin
         name_char_1 <= 31; //U
         name_char_2 <= 29; //S
         name_char_3 <= 13; //D
         end
         6'd1: begin
         name_char_1 <= 15; //E
         name_char_2 <= 31; //U
         name_char_3 <= 28; //R
         end
         6'd2: begin
         name_char_1 <= 13; //C
         name_char_2 <= 11; //A
         name_char_3 <= 14; //D
         end
         6'd3: begin
         name_char_1 <= 17; //G
         name_char_2 <= 12; //B
         name_char_3 <= 26; //P
         end
         6'd4: begin
         name_char_1 <= 24; //N
         name_char_2 <= 36; //Z
         name_char_3 <= 14; //D
         end
         6'd5: begin
         name_char_1 <= 11; //A
         name_char_2 <= 31; //U
         name_char_3 <= 14; //D
         end
         6'd6: begin
         name_char_1 <= 13; //C
         name_char_2 <= 18; //H
         name_char_3 <= 16; //F
         end
         default: begin
         name_char_1 <= 0; //
         name_char_2 <= 0;
         name_char_3 <= 0;
         end
         endcase
    end
    default: begin
         name_char_1 <= 0; //
         name_char_2 <= 0;
         name_char_3 <= 0;
         end
    endcase
end

endmodule
```

PrintCylce.sv

```
module PrintCycle(input logic clk, print_reset,
                   /*Vertmat Read Inputs*/
                   input logic [`VERT_WIDTH:0] vertmat_q_b,
                   /*VertMat Memory*/
                   output logic [`VERT_WIDTH:0] vertmat_data_b,
                   output logic [`PRED_WIDTH:0] vertmat_addr_b,
                   output logic vertmat_we_b,
                   /*Screen*/
                   output logic [5:0] frame_char,
                   output logic [5:0] frame_x,
                   output logic [5:0] frame_y,
                   output logic frame_we,
                   output logic print_done);

    enum logic [3:0] {READ, IDLE, CHECK_CYCLE, START_WRITE, FIRST_DIGIT, SECOND_DIGIT, ARROW,
RECHECK_CYCLE, FINISH_CYCLE, DONE} state;

    logic [`PRED_WIDTH:0] i, j, k, l; //Indices
    logic [5:0] px, py = -1;
    assign frame_x = px;
    assign frame_y = py;

    always_comb begin
      frame_we = 0;
      frame_char = 0;
      vertmat_addr_b = j;
      vertmat_data_b = 0;
      vertmat_we_b = 0;
      case (state)
        READ: ;
        IDLE: ;
        CHECK_CYCLE: begin
          vertmat_addr_b = j;
          //if (vertmat_q_b[`VERT_WIDTH]) vertmat_addr_b = l;
        end
        START_WRITE: begin
          vertmat_addr_b = j;
        end
        FIRST_DIGIT: begin
          vertmat_addr_b = l;
          vertmat_data_b = {1'b0, vertmat_q_b[`VERT_WIDTH - 1:0]};
          vertmat_we_b = 1;
          frame_we = 1;
          if (l < 10) frame_char = 10;
          else if (l >= 60) frame_char = 6; //To avoid division
          else if (l >= 50) frame_char = 5;
          else if (l >= 40) frame_char = 4;
          else if (l >= 30) frame_char = 3;
          else if (l >= 20) frame_char = 2;
          else frame_char = 1;
        end
        SECOND_DIGIT: begin
          vertmat_we_b = 0;
          vertmat_addr_b = l;
          frame_we = 1;
          if (l < 10) frame_char = l;
          else if (l >= 60) frame_char = l - 60;
          else if (l >= 50) frame_char = l - 50;
          else if (l >= 40) frame_char = l - 40;
          else if (l >= 30) frame_char = l - 30;
          else if (l >= 20) frame_char = l - 20;
          else frame_char = l- 10;
        end
        ARROW: begin
          vertmat_addr_b = l;
          if (l != k) begin
            frame_we = 1;
            frame_char = 37;
          end
        end
        RECHECK_CYCLE: vertmat_addr_b = l;
        default: ;
      endcase
    end
```

```
always_ff @(posedge clk) begin
  if (print_reset) begin
    i <= 0;
    j <= 0;
    k <= -1;
    l <= 0;
    print_done <= 0;
    state <= READ;
  end else case (state)
    READ: begin
      k <= -1;
      if (j + 1 == `NODES) begin
        state <= DONE;
      end else begin
        j <= j + 1;
        state <= IDLE;
      end
    end
    IDLE: begin
      k <= j;
      state <= CHECK_CYCLE;
    end
    CHECK_CYCLE: begin
      //l <= vertmat_q_b[(`VERT_WIDTH-1):(`WEIGHT_WIDTH+1)];
      if (vertmat_q_b[`VERT_WIDTH]) state <= START_WRITE;
      else state <= READ;
    end
    START_WRITE: begin
      if (py + 1 == 30) begin
        py <= 0;
        px <= 0;
      end else begin
        py <= py + 1;
        px <= 0;
      end
         state <= FIRST_DIGIT;
    end
    FIRST_DIGIT: begin
      l <= vertmat_q_b[(`VERT_WIDTH-1):(`WEIGHT_WIDTH+1)];
      if (px + 1 == 40 && py + 1 == 30) begin
        py <= 0;
        px <= 0;
      end else if (px + 1 == 40) begin
        py <= py + 1;
        px <= 0;
      end else px <= px + 1;
      state <= SECOND_DIGIT;
    end
    SECOND_DIGIT: begin
      if (px + 1 == 40 && py + 1 == 30) begin
        py <= 0;
        px <= 0;
      end else if (px + 1 == 40) begin
        py <= py + 1;
        px <= 0;
      end else px <= px + 1;
      state <= ARROW;
    end
```

```
        ARROW: begin
          if (px + 1 == 40 && py + 1 == 30) begin
            py <= 0;
            px <= 0;
          end else if (px + 1 == 40) begin
            py <= py + 1;
            px <= 0;
          end else px <= px + 1;
          state <= RECHECK_CYCLE;
        end
        RECHECK_CYCLE: begin
          if (l == k) state <= READ;
          else begin
            state <= FIRST_DIGIT;
          end
        end
        DONE: print_done <= 1;
        default: state <= DONE;
      endcase
    end
endmodule
```

PrintCycle.sv with pretty print

```
module PrintCycle(input logic clk, print_reset,
                   /*Vertmat Read Inputs*/
                   input logic [`VERT_WIDTH:0] vertmat_q_b,
                   /*VertMat Memory*/
                   output logic [`VERT_WIDTH:0] vertmat_data_b,
                   output logic [`PRED_WIDTH:0] vertmat_addr_b,
                   output logic vertmat_we_b,
                   /*Screen*/
                   output logic [5:0] frame_char,
                   output logic [5:0] frame_x,
                   output logic [5:0] frame_y,
                   output logic [5:0] line_start,
                   output logic [5:0] line_end,
                   output logic frame_we,
                   output logic print_done);

    enum logic [3:0] {READ, IDLE, CHECK_CYCLE, START_WRITE, FIRST_DIGIT, SECOND_DIGIT, ARROW,
RECHECK_CYCLE, FINISH_CYCLE, CLEAN, DONE} state;

    logic [`PRED_WIDTH:0] i, j, k, l; //Indices

    logic [5:0] px, py = -1;
    assign frame_x = px;
    assign frame_y = py;

    always_comb begin
      frame_we = 0;
      frame_char = 0;
      vertmat_addr_b = j;
      vertmat_data_b = 0;
      vertmat_we_b = 0;
      case (state)
    READ: ;
    IDLE: ;
    CHECK_CYCLE: begin
      vertmat_addr_b = j;
      //if (vertmat_q_b[`VERT_WIDTH]) vertmat_addr_b = l;
    end
    START_WRITE: begin
      vertmat_addr_b = j;
    end
    FIRST_DIGIT: begin
      vertmat_addr_b = l;
      vertmat_data_b = {1'b0, vertmat_q_b[`VERT_WIDTH - 1:0]};
      vertmat_we_b = 1;
      frame_we = 1;
      if (l < 10) frame_char = 10;
      else if (l >= 60) frame_char = 6; //To avoid division
      else if (l >= 50) frame_char = 5;
      else if (l >= 40) frame_char = 4;
      else if (l >= 30) frame_char = 3;
      else if (l >= 20) frame_char = 2;
      else frame_char = 1;
    end
    SECOND_DIGIT: begin
      vertmat_we_b = 0;
      vertmat_addr_b = l;
      frame_we = 1;
           if (l == 0) frame_char = 0;
      else if (l < 10) frame_char = l;
      else if (l >= 60) frame_char = l - 60;
      else if (l >= 50) frame_char = l - 50;
      else if (l >= 40) frame_char = l - 40;
      else if (l >= 30) frame_char = l - 30;
      else if (l >= 20) frame_char = l - 20;
      else frame_char = l- 10;
    end
    ARROW: begin
      vertmat_addr_b = l;
      if (l != k) begin
        frame_we = 1;
        frame_char = 37;
      end
      else begin
         frame_we = 1;
        frame_char = 0;
      end
    end
    RECHECK_CYCLE: vertmat_addr_b = l;
    CLEAN: begin
        frame_we =1;
        frame_char =0;
    end
    default: ;
  endcase
end

always_ff @(posedge clk) begin
  if (print_reset) begin
    i <= 0;
    j <= 0;
    k <= -1;
    l <= 0;
    print_done <= 0;
    state <= READ;
    line_start <= py +1;
  end else case (state)
    READ: begin
      k <= -1;
      if (j + 1 == `NODES) begin
        state <= DONE;
      end else begin
        j <= j + 1;
        state <= IDLE;
      end
    end
    IDLE: begin
      k <= j;
      state <= CHECK_CYCLE;
    end
    CHECK_CYCLE: begin
      //l <= vertmat_q_b[(`VERT_WIDTH-1):(`WEIGHT_WIDTH+1)];
      if (vertmat_q_b[`VERT_WIDTH]) state <= START_WRITE;
      else state <= READ;
    end
    START_WRITE: begin
      if (py + 1 == 30) begin
        py <= 0;
        px <= 0;
      end else begin
        py <= py + 1;
        px <= 0;
      end
         state <= FIRST_DIGIT;
    end
    FIRST_DIGIT: begin
      l <= vertmat_q_b[(`VERT_WIDTH-1):(`WEIGHT_WIDTH+1)];
      if (px + 1 == 40 && py + 1 == 30) begin
        py <= 0;
        px <= 0;
      end else if (px + 1 == 40) begin
        py <= py + 1;
        px <= 0;
      end else px <= px + 1;
      state <= SECOND_DIGIT;
    end
    SECOND_DIGIT: begin
      if (px + 1 == 40 && py + 1 == 30) begin
        py <= 0;
        px <= 0;
      end else if (px + 1 == 40) begin
        py <= py + 1;
        px <= 0;
      end else px <= px + 1;
      state <= ARROW;
    end
    ARROW: begin
      if (px + 1 == 40 && py + 1 == 30) begin
```

```
            py <= 0;
            px <= 0;
          end else if (px + 1 == 40) begin
            py <= py + 1;
            px <= 0;
          end else px <= px + 1;
          state <= RECHECK_CYCLE;
        end
        RECHECK_CYCLE: begin
          if (l == k) state <= CLEAN;
          else begin
            state <= FIRST_DIGIT;
          end
        end
        CLEAN: begin
            if(px+1==40) state <= READ;
            else px <= px +1;
        end
        DONE: begin
            print_done <= 1;
            line_end <= py;
        end
        default: state <= DONE;
      endcase
    end
endmodule
```

VertMat.sv

```
// Quartus II Verilog Template
// Single port RAM with single read/write address
`include "Const.vh"

module VertMat
#(parameter DATA_WIDTH=`VERT_WIDTH, parameter ADDR_WIDTH=`PRED_WIDTH)
(
    input [DATA_WIDTH:0] data_a, data_b,
    input [ADDR_WIDTH:0] addr_a, addr_b,
    input we_a, we_b, clk,
    output reg [DATA_WIDTH:0] q_a, q_b,
    output [31:0] test
);

    // Declare the RAM variable
    reg [DATA_WIDTH:0] ram[2**ADDR_WIDTH-1:0];
    assign test = ram[2][`VERT_WIDTH - 1: `WEIGHT_WIDTH + 1];
    // Port A
    always @ (posedge clk)
    begin
        if (we_a)
        begin
            ram[addr_a] <= data_a;
            q_a <= data_a;
        end
        else
        begin
            q_a <= ram[addr_a];
        end
    end

    // Port B
    always @ (posedge clk)
    begin
        if (we_b)
        begin
            ram[addr_b] <= data_b;
            q_b <= data_b;
        end
        else
        begin
            q_b <= ram[addr_b];
        end
    end

endmodule
```

9.2 Software

stream.py

```
import sys, os, time #system
import fcntl, termios, ctypes #ioctl
import csv, linecache #CSV
import math
```

```
DECIMAL_PLACES = 1000000
DRIVER_FILE = "/dev/forex_driver"

currencies = {}

edges = [(0, 1, 1),
         (1, 2, 2),
         (2, 3, 2),
         (3, 4, 1),
         (4, 2, -13),
         (1, 5, -9),
         (5, 6, 1),
         (6, 1, 1)]

class driver_arg_t(ctypes.Structure):
    _fields_ = [
        ('w', ctypes.c_int),
        ('src', ctypes.c_uint),
        ('dst', ctypes.c_uint),
    ]

def main(path):
    driver_fd = open(DRIVER_FILE)
    hash_i = 0
    row = 0
    driver_args = driver_arg_t()
    while 1:
        #for fn in os.listdir(path):
               #print "READING: " + fn
            #try:
                #Take in input
                '''
                pair = fn.split('_')[2]
                if(not pair[:3] in currencies):
                    currencies[pair[:3]] = hash_i
                    hash_i += 1

                if(not pair[3:6] in currencies):
                    currencies[pair[3:6]] = hash_i
                    hash_i +=  1

                src = currencies[pair[:3]]
                dst = currencies[pair[3:6]]
                line = linecache.getline(path+"/"+fn, row)
                tick = csv.reader(line.splitlines())
                tick_arr = next(tick)
                #Do some maths
                rate = float(tick_arr[len(tick_arr) - 2]) #For testing -1, but in reality -2
                op_rate =  -1*math.log(rate, 10) * DECIMAL_PLACES
                round_int = int(op_rate)
                '''
                #Data Struct
                driver_args.src = edges[row][0]#src
                driver_args.dst = edges[row][1]#dst
                driver_args.w = edges[row][2]#round_int
                #print "Writing..."
                #print "Row: " + str(row)
                #print "Src: " + str(driver_args.src)
                #print "Dst: " + str(driver_args.dst)
                #print "W: " + str(driver_args.w)
                fcntl.ioctl(driver_fd, 0, driver_args) #check for ref or not
                #driver_args.src = dst
                #driver_args.dst = src
                #driver_args.w = -1*round_int
                #fcntl.ioctl(driver_fd, 0, driver_args) #check for ref or not
                time.sleep(0.01) #Wait 1000 ms
            #except:
            #    print "Error"

                row = row + 1
                print "Row:" + str(row)
                if(row == 8):
                    print "ALL DONE"
                    return

if __name__ == "__main__":
    if (len(sys.argv) == 2):
        main(sys.argv[1])
```

my_driver.h

```
#ifndef _MY_DRIVER_H
#define _MY_DRIVER_H

#include <linux/ioctl.h>

typedef struct {
    int w;
    unsigned int src;
  unsigned int dst;
} driver_arg_t;

#define MY_DRIVER_MAGIC 'q'
#define MY_DRIVER_WRITE _IOW(MY_DRIVER_MAGIC, 1, driver_arg_t *)
#endif
```

my_driver.c

```
/*
 * Device driver for the VGA driver Emulator
 *
 * Graham Gobieski and Kevin Kwan
 * gsg2120, kjk2150
 *
 * References:
 * http://www.linuxforu.com/tag/linux-device-drivers/
 */

#include <linux/module.h>
#include <linux/version.h>
#include <linux/kernel.h>
```

```
#include <linux/types.h>
#include <linux/kdev_t.h>
#include <linux/device.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/uaccess.h>
#include <asm/io.h>

#include "my_driver.h"
#include "hps.h"
#include "hps_0.h"

/* Memory region visible through the lightweight HPS-FPGA bridge */
#define HW_REGS_BASE ALT_LWFPGASLVS_OFST
#define HW_REGS_SIZE 0x200000
#define HW_REGS_MASK (HW_REGS_SIZE - 1)
#define PRED_WIDTH 7

static void __iomem *registers; /* Base of mapped memory */
static dev_t firstdev;
static struct class *cl;
static struct cdev c_dev;

static void *driver_registers; /* Start of registers for drivers */

/* Low-level write routine: digit assumed to be in range; remember the state */
static void write_edge(int w, unsigned int src, unsigned int dst) {
    printk(KERN_INFO "WRITING EDGE");
    unsigned int combo_vert = (src << PRED_WIDTH) | dst;
    iowrite32(combo_vert, driver_registers);
    iowrite32(w, driver_registers + sizeof(int));
}

static int my_open(struct inode *i, struct file *f) {
  return 0;
}

static int my_close(struct inode *i, struct file *f) {
  return 0;
}

/* Handle ioctls(): write to the display registers or read our state */
static long my_ioctl(struct file *f, unsigned int cmd, unsigned long arg) {
    driver_arg_t da;
    printk(KERN_INFO "Got to ioctl");
    if (copy_from_user(&da, (driver_arg_t *) arg, sizeof(driver_arg_t)))
        return -EACCES;

    write_edge(da.w, da.src, da.dst);
    return 0;
}

static struct file_operations my_fops = {
  .owner = THIS_MODULE,
  .open = my_open,
  .release = my_close,
  .unlocked_ioctl = my_ioctl
};
```

```
/* Initialize the driver: map the hardware registers, register the
 * device and our operations, and display a welcome message */
static int __init my_driver_init(void) {

  printk(KERN_INFO "Forex Driver: init\n");

  if ( (registers = ioremap(HW_REGS_BASE, HW_REGS_SIZE)) == NULL ) {
    printk(KERN_ERR "driver: Mapping hardware registers faidriver\n");
    return -1;
  }

  driver_registers = registers +
    ((unsigned long) MY_DRIVER_BASE & (unsigned long) HW_REGS_MASK);

  if (alloc_chrdev_region(&firstdev, 0, 1, "forex_driver") < 0) goto unmap;
  if ((cl = class_create(THIS_MODULE, "chardrv")) == NULL) goto unregister;
  if (device_create(cl, NULL, firstdev, NULL, "forex_driver") == NULL) goto del_class;
  cdev_init(&c_dev, &my_fops);
  if (cdev_add(&c_dev, firstdev, 1) == -1) goto del_device;

  return 0;

  /* Clean up if something went wrong */
  unmap:      iounmap(registers);
  del_device: device_destroy(cl, firstdev);
  del_class:  class_destroy(cl);
  unregister: unregister_chrdev_region(firstdev, 1);
  return -1;
}

/* Disable the driver; undo the effects of the initialization routine */
static void __exit my_driver_exit(void) {
  printk(KERN_INFO "forex driver: exit\n");

  cdev_del(&c_dev);
  device_destroy(cl, firstdev);
  class_destroy(cl);
  unregister_chrdev_region(firstdev, 1);
  iounmap(registers);
}

module_init(my_driver_init);
module_exit(my_driver_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Graham Gobieski gsg2120, Kevin Kwan kjk2150, etc.");
MODULE_DESCRIPTION("FOREX Driver");
```

hps.h

Not reproduced here as file is generated by Quartus.

hps_0.h

```
#ifndef _ALTERA_HPS_0_H_
#define _ALTERA_HPS_0_H_

/*
 * This file was automatically generated by the swinfo2header utility.
 *
 * Created from SOPC Builder system 'lab3' in
 * file './lab3.sopcinfo'.
 */

/*
 * This file contains macros for module 'hps_0' and devices
 * connected to the following masters:
 *   h2f_axi_master
 *   h2f_lw_axi_master
 *
 * Do not include this header file and another header file created for a
 * different module or master group at the same time.
 * Doing so may result in duplicate macro names.
 * Instead, use the system header file which has macros with unique names.
 */

/*
 * Macros for device 'vga_led_0', class 'vga_led'
 * The macros are prefixed with 'MY_DRIVER_'.
 * The prefix is the slave descriptor.
 */
#define MY_DRIVER_COMPONENT_TYPE my_driver
#define MY_DRIVER_COMPONENT_NAME my_driver
#define MY_DRIVER_BASE 0x0
#define MY_DRIVER_SPAN 8
#define MY_DRIVER_END 0x7


#endif /* _ALTERA_HPS_0_H_ */
```

Makefile

```
ifneq (${KERNELRELEASE},)

# KERNELRELEASE defined: we are being compiled as part of the Kernel
        obj-m := my_driver.o

else

# We are being compiled as a module: use the Kernel build system

    KERNEL_SOURCE := /usr/src/linux
        PWD := $(shell pwd)

default: module my_driver

module:
    ${MAKE} -C ${KERNEL_SOURCE} SUBDIRS=${PWD} modules

clean:
    ${MAKE} -C ${KERNEL_SOURCE} SUBDIRS=${PWD} clean
    ${RM} my_driver

endif
```

hello.c with keyboard support

```
/*
 * Userspace program that communicates with the led_vga device driver
 * primarily through ioctls
 *
 * Stephen A. Edwards
 * Columbia University
 */

#include <stdio.h>
#include "vga_led.h"
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include <math.h>
#include <stdlib.h>
#include <dirent.h>
#include "usbkeyboard.h"
#include <pthread.h>

struct libusb_device_handle *openkeyboard(uint8_t *endpoint_address);
void *keyboard_thread_f(void *);



int vga_led_fd;

int transmitFlag=0;
int continueTransmitFlag=0;
int breakFlag=0;

#define currencyNumber 5

double getRate(char * tempLine);
void getSouDes(char *filename, int * tSou, int *tDes);

char * currency[currencyNumber]={"USD","EUR","CAD","GBP","NZD"};
int message[VGA_LED_DIGITS] = {0};


/* Read and print the segment values */
void print_segment_info() {
  vga_led_arg_t vla;
  int i;

  for (i = 0 ; i < VGA_LED_DIGITS ; i++) {
    vla.digit = i;
    if (ioctl(vga_led_fd, VGA_LED_READ_DIGIT, &vla)) {
      perror("ioctl(VGA_LED_READ_DIGIT) failed");
      return;
    }
    printf("%02x ", vla.segments);
    //printf("%d ", vla.segments);
  }
  printf("\n");
}

/* Write the contents of the array to the display */
void write_segments(int segs[VGA_LED_DIGITS])
{
  vga_led_arg_t vla;
  int i;
  for (i = 0 ; i < VGA_LED_DIGITS ; i++) {
    vla.digit = i;
    vla.segments = segs[i];
    if (ioctl(vga_led_fd, VGA_LED_WRITE_DIGIT, &vla)) {
      perror("ioctl(VGA_LED_WRITE_DIGIT) failed");
      return;
    }
  }

}

void new_write_segments(int segs, int i)
{
    vga_led_arg_t vla;

    vla.digit = i;
    vla.segments = segs;
    if (ioctl(vga_led_fd, VGA_LED_WRITE_DIGIT, &vla)) {
        perror("ioctl(VGA_LED_WRITE_DIGIT) failed");
        return;
    }
}
```

```
//
int main()
{



    char dir[] = "original_data/";
    FILE *fp_name;
    char CSV_file_name[100][20];
    int fileNumber=0;

    system("cd original_data/\nls *.csv > CSV_file_name.txt");
    fp_name=fopen("original_data/CSV_file_name.txt","r");
    while(1){
        if ( fgets(CSV_file_name[fileNumber],20,fp_name)==NULL) break;
        CSV_file_name[fileNumber][strlen(CSV_file_name[fileNumber])-1]='\0';
        printf("%s\n",CSV_file_name[fileNumber]);
        fileNumber++;
    }
    fclose(fp_name);


    FILE *fp[100];
    int sou[100]={0};
    int des[100]={0};

    int source_number;

    int n;
    int tempSou,tempDes;
    char fileNameTemp[50]={'\0'};
    for(n=0;n<fileNumber;n++){
        strcpy(fileNameTemp,dir);
        strcat(fileNameTemp,CSV_file_name[n]);
        fp[n]=fopen(fileNameTemp, "rb");
        printf("path: %s\n",fileNameTemp);
        getSouDes(CSV_file_name[n],&(sou[n]),&(des[n]));

        printf("sou: %d  des: %d\n",sou[n],des[n]);
/*
if(n==0){
       source_number=sou[n];
       printf("source_number: %d\n",source_number);
    }
*/
    }





  vga_led_arg_t vla;
  int i;
  static const char filename[] = "/dev/vga_led";

//  static int message[VGA_LED_DIGITS] = {0};
```

```
 printf("VGA LED Userspace program started\n");

 if ( (vga_led_fd = open(filename, O_RDWR)) == -1) {
   fprintf(stderr, "could not open %s\n", filename);
   return -1;
 }

 printf("initial state: ");
 print_segment_info();

 write_segments(message);

 printf("current state: ");
 print_segment_info();

 pthread_t keyboard_thread;
 pthread_create(&keyboard_thread, NULL, keyboard_thread_f, NULL);

 int rate[currencyNumber][currencyNumber]={0};
/*
 int fake_rate[currencyNumber][currencyNumber]={{0,100,0,0,0,0,0},
                       {0,0,200,0,0,-1200,100},
                       {0,0,0,200,0,0,0},
                       {0,0,0,0,100,0,0},
                       {0,0,-1300,0,0,0,0},
                       {0,0,0,0,0,0,100},
                       {0,100,0,0,0,0,0}};

*/
 int total_running_time=1;
 int running_times=0;
 int ii=1;
 int jj=1;
 while(breakFlag!=1){
     if(continueTransmitFlag==1||transmitFlag==1){
         if(transmitFlag==1){
             transmitFlag=0;
         }
         char tempLine[200]={'\0'};
         double tempData;
         int logData;
         for(n=0;n<fileNumber;n++){
             fscanf(fp[n],"%s",tempLine);
             tempData=getRate(tempLine);
             logData=(int)(-log10(tempData)*1000000);
             rate[sou[n]][des[n]]=logData;
             //printf("sou: %d, des: %d, rate: %d\n",sou[n],des[n],rate[sou[n]][des[n]]);
             rate[des[n]][sou[n]]=-logData;
             //printf("sou: %d, des: %d, rate: %d\n",des[n],sou[n],rate[des[n]][sou[n]]);
         }

         int row,col;
         for(row=0;row<currencyNumber;row++){
             for (col=0;col<currencyNumber;col++){
         if(rate[row][col]!=0&&col!=0){
                     message[0]=(row<<7)+col;
                     printf("%x :",message[0]);
```

```
                        message[1]=rate[row][col];
                //message[1] = fake_rate[row][col];
                        printf("%d \n",message[1]);
                new_write_segments(message[0],0);
                        new_write_segments(message[1],1);
                usleep(100000);
            }
            /*
                    message[0]=row;
                    message[1]=col;
            if(col%2!=0) message[3]=37;
            else
            {message[3]=ii;
            ii++;
              }
            if(ii==7) ii=1;

                    //message[3]=row*5+col+ii;

                    new_write_segments(message[0],0);
                    new_write_segments(message[1],1);
                    new_write_segments(message[3],3);

                    if(message[2]!=0){
                        message[2]=0;
                    }

            */

                }
                //printf("\n");
            }

            //usleep(1000000);
        }
    }

      /* Terminate the keyboard thread */
      pthread_cancel(keyboard_thread);

      /* Wait for the keyboard thread to finish */
      pthread_join(keyboard_thread, NULL);

      for(n=0;n<fileNumber;n++){
          fclose(fp[n]);
      }

    printf("current state: ");
    print_segment_info();

      printf("\n");
      printf("\n");

    printf("VGA LED Userspace program terminating\n");

    return 0;
  }
```

```
void getSouDes(char *filename, int * tSou, int *tDes){
    char temp[4]={'\0'};
    int i,j;
    for(i=0;i<3;i++){
        temp[i]=filename[i];
    }

    for(j=0;j<currencyNumber;j++){
        if(strcmp(temp,currency[j])==0){
            *tSou =j;
            break;
        }
    }
    if(j==currencyNumber){
        printf("getSou error!\n");
    }
    for(i=4;i<7;i++){
        temp[i-4]=filename[i];
    }

    for(j=0;j<currencyNumber;j++){
        if(strcmp(temp,currency[j])==0){
            *tDes =j;
            break;
        }
    }
    if(j==currencyNumber){
        printf("getDes error!\n");
    }
}


double getRate(char * tempLine){
    double rate;
    int j,commaNumber=0,k=0;
    char tempNumber[20]={'\0'};
    for(j=0;j<strlen(tempLine);j++){
        if(tempLine[j]==','){
            commaNumber++;
            continue;
        }
        if(commaNumber==5){
            tempNumber[k]=tempLine[j];
            k++;
        }
        if(commaNumber==6){
            break;
        }
    }

    rate=atof(tempNumber);

    return rate;
}

void *keyboard_thread_f(void *ignored)
{
    struct libusb_device_handle *keyboard;
```

```
uint8_t endpoint_address;

struct usb_keyboard_packet packet;
int transferred;
char keystate[12];

/* Open the keyboard */
if ( (keyboard = openkeyboard(&endpoint_address)) == NULL ) {
    fprintf(stderr, "Did not find a keyboard\n");
    exit(1);
}

for (;;) {
    libusb_interrupt_transfer(keyboard, endpoint_address,
                              (unsigned char *) &packet, sizeof(packet),
                              &transferred, 0);
    if (transferred == sizeof(packet)) {
        sprintf(keystate, "%02x %02x %02x", packet.modifiers, packet.keycode[0],
                packet.keycode[1]);
                printf("%s\n", keystate);

        switch (packet.keycode[0]) {
            case 0x2C:  //space
                transmitFlag=1;
                break;
            case 0x1E:  //1
                continueTransmitFlag=1;
                break;
            case 0x1F:  //2
                continueTransmitFlag=0;
                break;
            case 0x20:  //3
                breakFlag=1;
                break;
        case 0x28:  //enter
        message[2]=3;
        new_write_segments(message[2],2);
        break;
            case 0x51:  //dowm arrow
        message[2]=2;
        new_write_segments(message[2],2);
        break;
            case 0x52:  //up arrow
                message[2]=1;
                new_write_segments(message[2],2);
        break;
    case 0x29:  //ESC
        message[2]=4;
        new_write_segments(message[2],2);
        break;
    case 0x00:  // release
        message[2]=0;
        new_write_segments(message[2],2);
        break;
            default:
                break;
        }
        }
    }


    return NULL;
}


struct libusb_device_handle *openkeyboard(uint8_t *endpoint_address) {
    libusb_device **devs;
    struct libusb_device_handle *keyboard = NULL;
    struct libusb_device_descriptor desc;
    ssize_t num_devs, d;
    uint8_t i, k;

    /* Start the library */
    if ( libusb_init(NULL) < 0 ) {
        fprintf(stderr, "Error: libusb_init failed\n");
        exit(1);
    }

    /* Enumerate all the attached USB devices */
    if ( (num_devs = libusb_get_device_list(NULL, &devs)) < 0 ) {
        fprintf(stderr, "Error: libusb_get_device_list failed\n");
        exit(1);
    }

    /* Look at each device, remembering the first HID device that speaks
     the keyboard protocol */

    for (d = 0 ; d < num_devs ; d++) {
        libusb_device *dev = devs[d];
        if ( libusb_get_device_descriptor(dev, &desc) < 0 ) {
            fprintf(stderr, "Error: libusb_get_device_descriptor failed\n");
            exit(1);
        }

        if (desc.bDeviceClass == LIBUSB_CLASS_PER_INTERFACE) {
            struct libusb_config_descriptor *config;
            libusb_get_config_descriptor(dev, 0, &config);
            for (i = 0 ; i < config->bNumInterfaces ; i++)
                for ( k = 0 ; k < config->interface[i].num_altsetting ; k++ ) {
                    const struct libusb_interface_descriptor *inter =
                    config->interface[i].altsetting + k ;
                    if ( inter->bInterfaceClass == LIBUSB_CLASS_HID &&
                        inter->bInterfaceProtocol == USB_HID_KEYBOARD_PROTOCOL) {
                        int r;
                        if ((r = libusb_open(dev, &keyboard)) != 0) {
                            fprintf(stderr, "Error: libusb_open failed: %d\n", r);
                            exit(1);
                        }
                        if (libusb_kernel_driver_active(keyboard,i))
                            libusb_detach_kernel_driver(keyboard, i);
                        //libusb_set_auto_detach_kernel_driver(keyboard, i);
                        if ((r = libusb_claim_interface(keyboard, i)) != 0) {
                            fprintf(stderr, "Error: libusb_claim_interface failed: %d\n", r);
                            exit(1);
                        }
                        *endpoint_address = inter->endpoint[0].bEndpointAddress;
                        goto found;
                    }
                }
        }
    }

found:
    libusb_free_device_list(devs, 1);

    return keyboard;
}
```

usbkeyboard.h with keyboard support

```
#ifndef _USBKEYBOARD_H
#define _USBKEYBOARD_H

#include <libusb-1.0/libusb.h>

#define USB_HID_KEYBOARD_PROTOCOL 1

/* Modifier bits */
#define USB_LCTRL  (1 << 0)
#define USB_LSHIFT (1 << 1)
#define USB_LALT   (1 << 2)
#define USB_LGUI   (1 << 3)
#define USB_RCTRL  (1 << 4)
#define USB_RSHIFT (1 << 5)
#define USB_RALT   (1 << 6)
#define USB_RGUI   (1 << 7)

struct usb_keyboard_packet {
  uint8_t modifiers;
  uint8_t reserved;
  uint8_t keycode[6];
};

/* Find and open a USB keyboard device.  Argument should point to
   space to store an endpoint address.  Returns NULL if no keyboard
   device was found. */
extern struct libusb_device_handle *openkeyboard(uint8_t *);

#endif
```

vga_led.h with keyboard support

```
#ifndef _VGA_LED_H
#define _VGA_LED_H

#include <linux/ioctl.h>

#define VGA_LED_DIGITS 4

typedef struct {
  unsigned int digit;    /* 0, 1, .. , VGA_LED_DIGITS - 1 */
  int segments; /* LSB is segment a, MSB is decimal point */
} vga_led_arg_t;

#define VGA_LED_MAGIC 'q'

/* ioctls and their arguments */
#define VGA_LED_WRITE_DIGIT _IOW(VGA_LED_MAGIC, 1, vga_led_arg_t *)
#define VGA_LED_READ_DIGIT  _IOWR(VGA_LED_MAGIC, 2, vga_led_arg_t *)

#endif
```

vga_led.c

```
/*
 * Device driver for the VGA LED Emulator
 *
 * A Platform device implemented using the misc subsystem
 *
 * Stephen A. Edwards
 * Columbia University
 *
 * References:
 * Linux source: Documentation/driver-model/platform.txt
 *               drivers/misc/arm-charlcd.c
 * http://www.linuxforu.com/tag/linux-device-drivers/
 * http://free-electrons.com/docs/
 *
 * "make" to build
 * insmod vga_led.ko
 *
 * Check code style with
 * checkpatch.pl --file --no-tree vga_led.c
 */

#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
```

```
#include <linux/uaccess.h>
#include "vga_led.h"


#define DRIVER_NAME "vga_led"

/*
 * Information about our device
 */
struct vga_led_dev {
    struct resource res; /* Resource: our registers */
    void __iomem *virtbase; /* Where registers can be accessed in memory */
    int segments[VGA_LED_DIGITS];
} dev;

/*
 * Write segments of a single digit
 * Assumes digit is in range and the device information has been set up
 */
static void write_digit(int digit, int segments)
{
    iowrite32(segments, dev.virtbase + digit*4);
    dev.segments[digit] = segments;
}

/*
 * Handle ioctl() calls from userspace:
 * Read or write the segments on single digits.
 * Note extensive error checking of arguments
 */
static long vga_led_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{
    vga_led_arg_t vla;

    switch (cmd) {
    case VGA_LED_WRITE_DIGIT:
        if (copy_from_user(&vla, (vga_led_arg_t *) arg,
                   sizeof(vga_led_arg_t)))
            return -EACCES;
        if (vla.digit > VGA_LED_DIGITS)
            return -EINVAL;
        write_digit(vla.digit, vla.segments);
        break;

    case VGA_LED_READ_DIGIT:
        if (copy_from_user(&vla, (vga_led_arg_t *) arg,
                   sizeof(vga_led_arg_t)))
            return -EACCES;
        if (vla.digit > VGA_LED_DIGITS)
            return -EINVAL;
        vla.segments = dev.segments[vla.digit];
        if (copy_to_user((vga_led_arg_t *) arg, &vla,
                 sizeof(vga_led_arg_t)))
            return -EACCES;
        break;

    default:
        return -EINVAL;
    }

    return 0;
}

/* The operations our device knows how to do */
static const struct file_operations vga_led_fops = {
    .owner      = THIS_MODULE,
    .unlocked_ioctl = vga_led_ioctl,
};

/* Information about our device for the "misc" framework -- like a char dev */
static struct miscdevice vga_led_misc_device = {
    .minor      = MISC_DYNAMIC_MINOR,
    .name       = DRIVER_NAME,
    .fops       = &vga_led_fops,
};

/*
 * Initialization code: get resources (registers) and display
 * a welcome message
 */
static int __init vga_led_probe(struct platform_device *pdev)
{
    static int welcome_message[VGA_LED_DIGITS] = {0};
    int i, ret;

    /* Register ourselves as a misc device: creates /dev/vga_led */
    ret = misc_register(&vga_led_misc_device);

    /* Get the address of our registers from the device tree */
    ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
    if (ret) {
        ret = -ENOENT;
        goto out_deregister;
    }

    /* Make sure we can use these registers */
    if (request_mem_region(dev.res.start, resource_size(&dev.res),
                   DRIVER_NAME) == NULL) {
        ret = -EBUSY;
        goto out_deregister;
    }

    /* Arrange access to our registers */
    dev.virtbase = of_iomap(pdev->dev.of_node, 0);
    if (dev.virtbase == NULL) {
        ret = -ENOMEM;
        goto out_release_mem_region;
    }

    /* Display a welcome message */
    for (i = 0; i < VGA_LED_DIGITS; i++)
        write_digit(i, welcome_message[i]);

    return 0;

out_release_mem_region:
    release_mem_region(dev.res.start, resource_size(&dev.res));
out_deregister:
    misc_deregister(&vga_led_misc_device);
    return ret;
}

/* Clean-up code: release resources */
static int vga_led_remove(struct platform_device *pdev)
{
    iounmap(dev.virtbase);
    release_mem_region(dev.res.start, resource_size(&dev.res));
    misc_deregister(&vga_led_misc_device);
    return 0;
}

/* Which "compatible" string(s) to search for in the Device Tree */
#ifdef CONFIG_OF
static const struct of_device_id vga_led_of_match[] = {
    { .compatible = "altr,vga_led" },
    {},
};
MODULE_DEVICE_TABLE(of, vga_led_of_match);
#endif

/* Information for registering ourselves as a "platform" driver */
static struct platform_driver vga_led_driver = {
    .driver = {
        .name   = DRIVER_NAME,
        .owner  = THIS_MODULE,
        .of_match_table = of_match_ptr(vga_led_of_match),
    },
    .remove = __exit_p(vga_led_remove),
};

/* Called when the module is loaded: set things up */
static int __init vga_led_init(void)
{
    pr_info(DRIVER_NAME ": init\n");
    return platform_driver_probe(&vga_led_driver, vga_led_probe);
}

/* Called when the module is unloaded: release resources */
static void __exit vga_led_exit(void)
{
    platform_driver_unregister(&vga_led_driver);
    pr_info(DRIVER_NAME ": exit\n");
}

module_init(vga_led_init);
module_exit(vga_led_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Stephen A. Edwards, Columbia University");
MODULE_DESCRIPTION("VGA 7-segment LED Emulator");
```

Makefile with keyboard support

```
ifneq (${KERNELRELEASE},)

# KERNELRELEASE defined: we are being compiled as part of the Kernel
        obj-m := vga_led.o

else

# We are being compiled as a module: use the Kernel build system

    KERNEL_SOURCE := /usr/src/linux
        PWD := $(shell pwd)

default: module hello

CFLAGS = -Wall

OBJECTS = hello.o #vga_led.o

hello: $(OBJECTS)
    cc $(CFLAGS) -o hello $(OBJECTS) -lusb-1.0 -lm -pthread

hello.o : hello.c vga_led.h usbkeyboard.h
#vga_led.o : vga_led.c vga_led.h

module:
    ${MAKE} -C ${KERNEL_SOURCE} SUBDIRS=${PWD} modules

clean:
    ${MAKE} -C ${KERNEL_SOURCE} SUBDIRS=${PWD} clean
    ${RM} hello

socfpga.dtb : socfpga.dtb
    dtc -O dtb -o socfpga.dtb socfpga.dts

endif
```

9.3 Testbench

Bellman_TB.cpp

```
#include <iostream>
#include <bitset>
#include <stdio.h>

#include "VFOREX.h"
#include "verilated.h"
#include "verilated_vcd_c.h"

#define WORD_WIDTH 32
#define PRED_WIDTH 7
#define WEIGHT_WIDTH 32

#define NODES 4
#define CYCLES 2000
#define PRINT_CYCLE 5
```

```
int extend_signed_int_width(unsigned int val, unsigned int width);
unsigned int lower_half_int(unsigned long in, int bits);
unsigned int upper_half_int(unsigned long in, int bits);
unsigned int src_dst(unsigned int src, unsigned int dst);
std::string binary_to_string(long in);

int main(int argc, char **argv, char **env) {
  int i, j;
  int clk;
  int old_char = 0;
  Verilated::commandArgs(argc, argv);
  // init top verilog instance
  VFOREX* top = new VFOREX;
  // init trace dump
  Verilated::traceEverOn(true);
  VerilatedVcdC* tfp = new VerilatedVcdC;
  top->trace (tfp, 99);
  tfp->open ("Container.vcd");
  // initialize simulation inputs
  top->clk = 1;
  top->reset = 1;
  std::cout << "\nRUNNING SIM\n";
  // run simulation for 500 clock periods
  for (i=0; i<CYCLES; i++) {
    // dump variables into VCD file and toggle clock
    for (clk=0; clk<2; clk++) {
      tfp->dump (2*i+clk);
      top->clk = !top->clk;
      top->eval ();
    }
    top->reset = 0;

    if((int)top->frame_we == 1){
      std::cout << "Char: " << (int)top->frame_char << "Pred: " << (int)top->test << "\n";
    }
    top->write = 0;
    top->chipselect = 0;
    switch (i) {
      case 0:
        top->write = 1;
        top->chipselect = 1;
        top->writedata = src_dst(0, 1);
        top->address = 0;
        break;
      case 5:
        top->write = 1;
        top->chipselect = 1;
        top->writedata = 4;
        top->address = 1;
        break;
      case 200:
        top->write = 1;
        top->chipselect = 1;
        top->writedata = src_dst(1, 2);
        top->address = 0;
        break;
      case 205:
        top->write = 1;
        top->chipselect = 1;
        top->writedata = 3;
        top->address = 1;
        break;
      case 300:
        top->write = 1;
        top->chipselect = 1;
        top->writedata = src_dst(2, 3);
        top->address = 0;
        break;
      case 305:
        top->write = 1;
        top->chipselect = 1;
        top->writedata = 4;
        top->address = 1;
        break;
      case 400:
        top->write = 1;
        top->chipselect = 1;
        top->writedata = src_dst(3, 4);
        top->address = 0;
        break;
      case 405:
        top->write = 1;
        top->chipselect = 1;
        top->writedata = -12;
        top->address = 1;
        break;
      case 500:
        top->write = 1;
        top->chipselect = 1;
        top->writedata = src_dst(4, 5);
        top->address = 0;
        break;
      case 505:
        top->write = 1;
        top->chipselect = 1;
        top->writedata = 1;
        top->address = 1;
        break;
      case 600:
        top->write = 1;
        top->chipselect = 1;
        top->writedata = src_dst(5, 6);
        top->address = 0;
        break;
      case 605:
        top->write = 1;
        top->chipselect = 1;
        top->writedata = 1;
        top->address = 1;
        break;
      case 700:
        top->write = 1;
        top->chipselect = 1;
        top->writedata = src_dst(6, 1);
        top->address = 0;
        break;
      case 705:
        top->write = 1;
        top->chipselect = 1;
        top->writedata = -7;
        top->address = 1;
        break;
    }

    if (Verilated::gotFinish())  exit(0);
  }
  tfp->close();
  exit(0);
}

int extend_signed_int_width(unsigned int val, unsigned int width) {
  int i;
  int mask = ~0;
  int tmp = val >> (width-1);
  if ((tmp & 1) == 0) return val;
  mask <<= width;
  return val | mask;
}

std::string binary_to_string(long in) {
  std::string binary = std::bitset<WORD_WIDTH>(in).to_string();
  return binary;
}

unsigned int lower_half_int(unsigned long in, int bits) {
  int i;
  int mask = 0;
  for(i = 0; i < bits; i++) {
    mask |= 1;
    if (i < bits - 1) mask <<= 1;
  }
  return in & mask;
}
unsigned int upper_half_int(unsigned long in, int bits) {
  return (unsigned int)(in >> (WORD_WIDTH-bits));
}

unsigned int src_dst(unsigned int src, unsigned int dst){
  return (src << PRED_WIDTH) | dst;
}
```

Makefile

```
main: Bellman.sv Const.vh Container.sv CycleDetect.sv AdjMat.sv Vertmat.sv FOREX.sv PrintCycle.sv
    verilator  --unroll-count 9999 -Wno-lint --cc --trace FOREX.sv

test: Bellman.sv Const.vh PrintCycle.sv CycleDetect.sv AdjMat.sv Vertmat.sv Container.sv FOREX.sv
testbench/Bellman_TB.cpp
    verilator  --unroll-count 9999 -Wno-lint --cc --trace FOREX.sv --exe testbench/Bellman_TB.cpp
    make -j -C obj_dir/ -f VFOREX.mk VFOREX
    ./obj_dir/VFOREX

.PHONY: clean
clean:
        rm -rf *.o a.out obj_dir
```