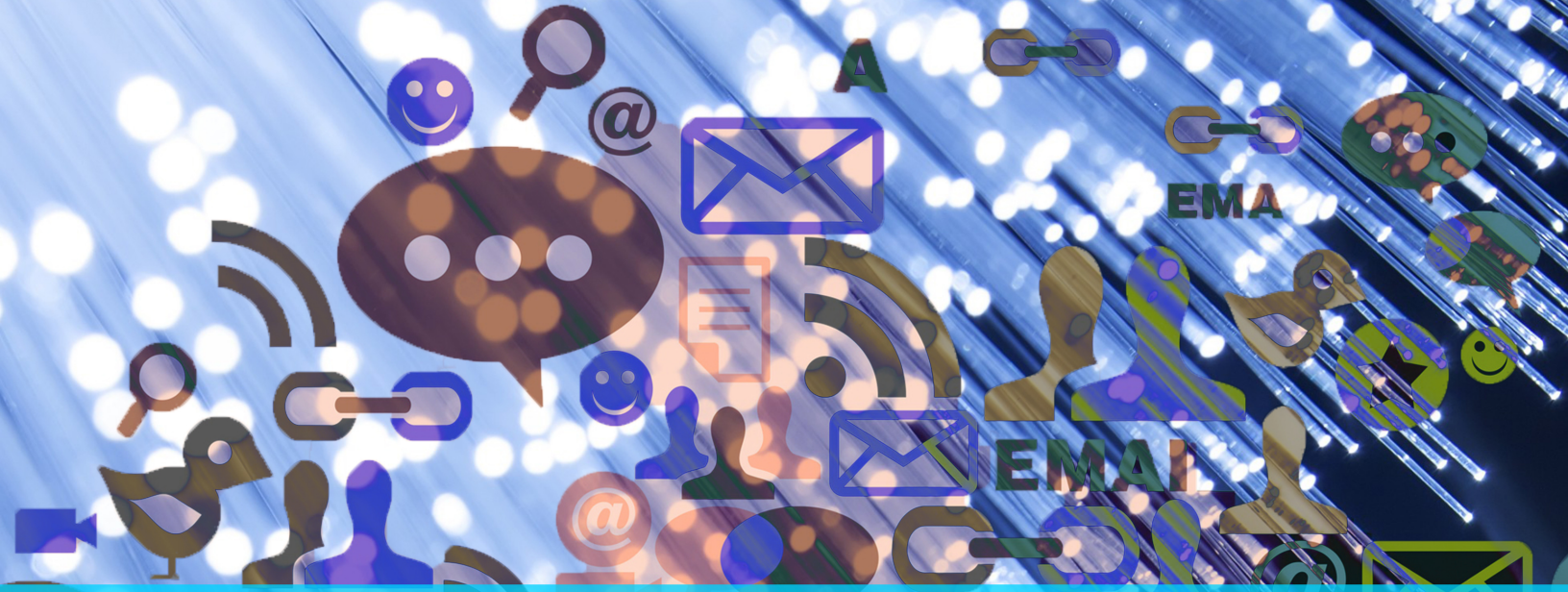
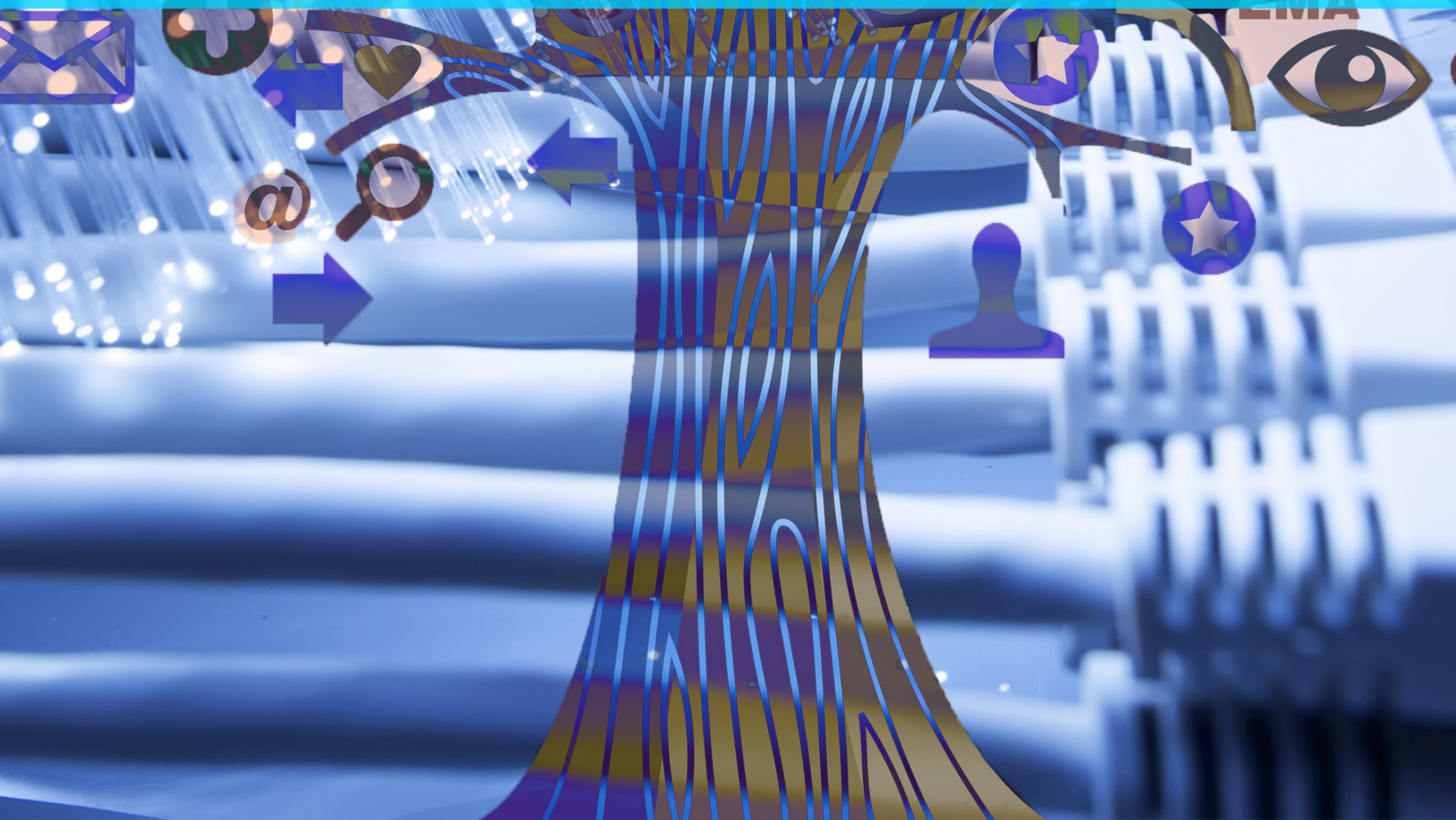
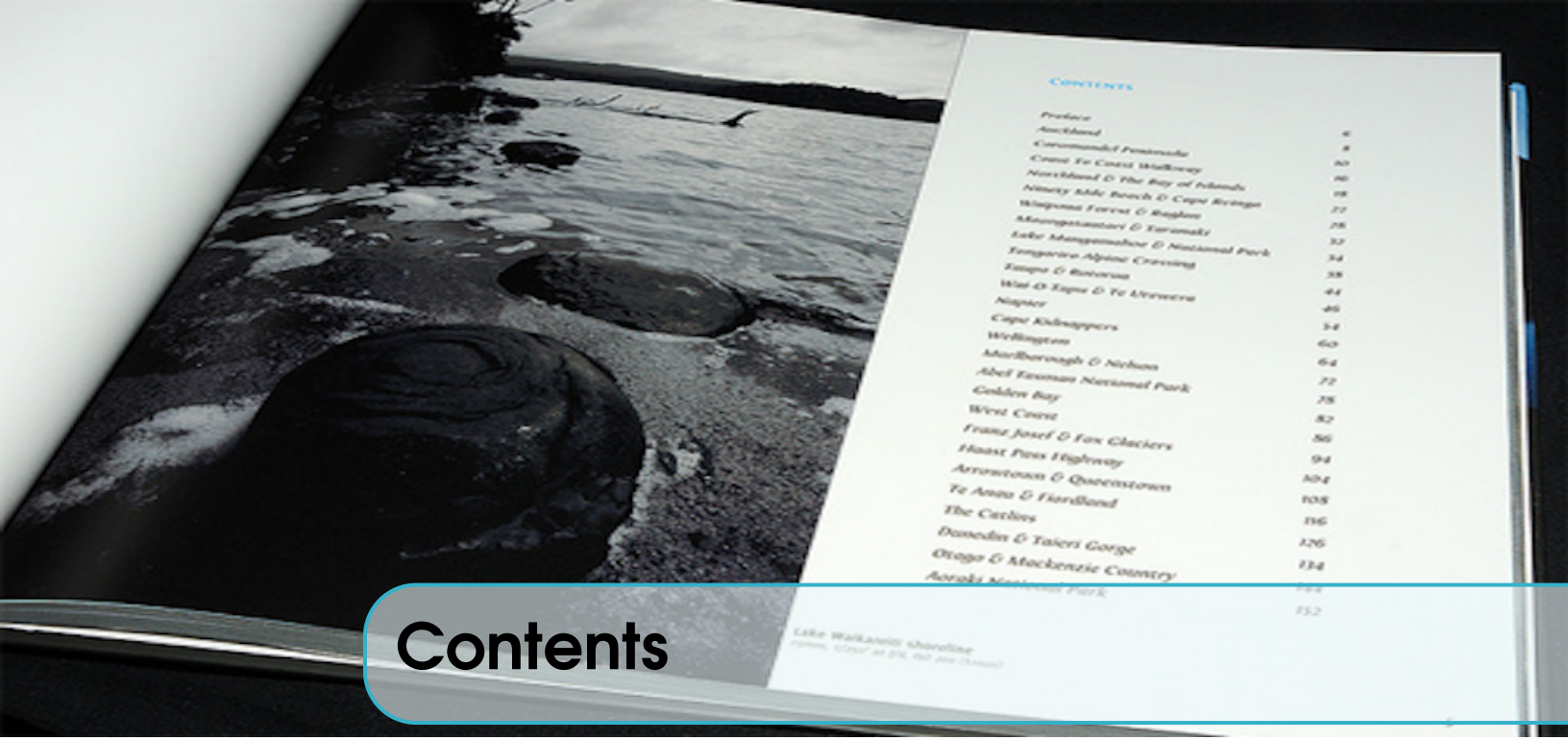


# Switch ON

An FPGA based Switch

Ayush Jain(aj2672)
Donovan Chan(dc3095)
Shivam Choudhary(sc3973)





# Contents

# 1. Introduction

## 1.1 Aim

The aim of the project is to create a FPGA based `switch`. The main focus of the project is in optimising the throughput of a network switch through the implementation of a scheduler. Decoding of actual incoming packets will not be considered in this project.[1] Therefore the packets being generated will contain a few items:

- Randomly generated data payload of variable length
- 8 bits of header that determines the destination port
- 8 bits storing the random seed number used for the payload generation

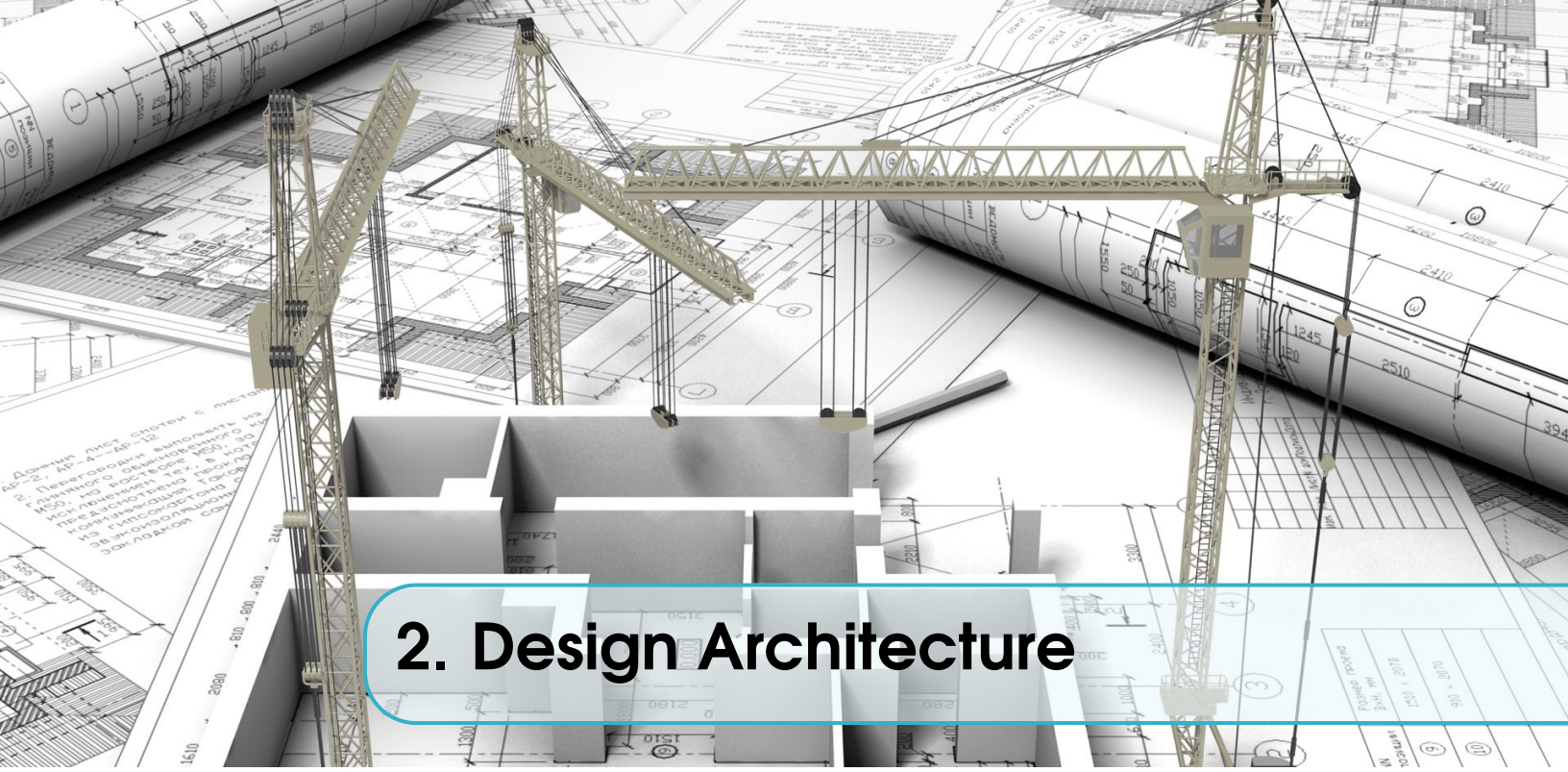
## 1.2 Overview

The FPGA contains a few components that make up the entire switch. The routing algorithm is handled by the scheduler within the FPGA to optimise the amount of throughput that the switch can handle[2]. The scheduler has to maintain correctness while working towards maximum efficiency. Random Access Memory (RAM) blocks also exist on the FPGA and model the real world input and output ports.

The user space consist of the packet generator and validator which interface with FPGA. They are responsible for generating packets with random destination ports and feed them into the FPGA module. The validator then reads from the output RAM and ensures that packets are routed correctly and no segments are dropped.

[1] There will be no decoder in the mainframe of the project and so any packet that is generated and sent to the switch will pass through to the port specified.

[2] Throughput is defined as the number of packets received at the output port in `one clock cycle`.



# 2. Design Architecture

## 2.1 System Architecture

The design architecture of the system is as shown in Figure 2.1 where both software components and hardware components are exhibited in the block diagram.

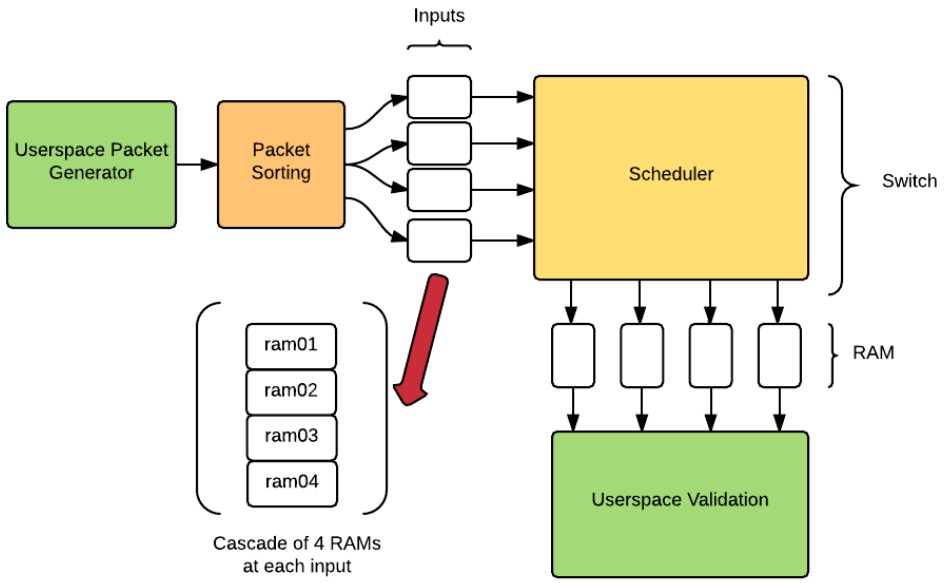



Figure 2.1: Block diagram showing the overall functionality and flow of the system

The userspace packet generator is responsible for generating random packets each with:

- 8 bits of header representing the destination port
- 8 bits storing the random seed that the data is generated based upon
- variable length data payload up to 64 bits

These packets are then sent to the packet sorting fabric on the FPGA which will decide which RAMs the packets will be sorted into based on the source port and the destination port. Each of the 4 inputs to the Scheduler has a cascade of 4 RAMs which identify which destination port the corresponding packet has to be routed to. The Scheduler then runs and proceeds to route the packets from the source RAMs into the corresponding destination port. The main aim of the scheduler is to maximise throughput by routing the most number of packets through the switch at every clock cycle. The RAMs located at the output then store these outputs. Each of the corresponding RAMs will only contain packets whose destination port corresponds to that specific output. The final step in the system is the Userspace Validation where the data stored in the memory locations of the output RAM will be retrieve and used to validate the integrity of the packets being sent through the switch.

## 2.2 Hardware Section

The hardware section of the entire system consists of the the following blocks as shown in Figure 2.2. The hardware segment of the system is responsible for storing and routing the input packets into the correct destination output port. The hardware segment being implemented on the FPGA interfaces with the userspace software program using the master-slave architecture (CHECK). One thing to note is that the hardware architecture is not affected by length of the packet that needs to be routed, it will continuously route that same packet to the destination port until an 'end-of-packet' identifier has been reached. This is being transmitted as zeros of 32 bits in length.

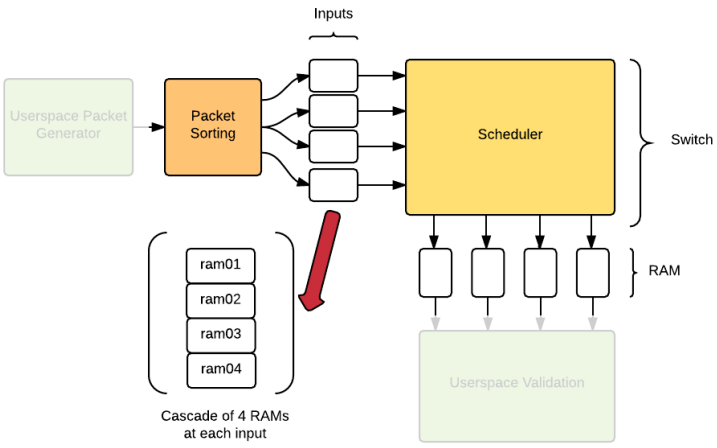



Figure 2.2: Hardware segment of the system

### 2.2.1 Avalon Bus

The userspace talks to the FPGA using avalon bus. Userspace has access to various registers which are registered to to the device drivers which communicate through ioctl 32 read and write calls. In this project this is the only part that has not been evaluated on Verilator because the slides actually show the real scenario for the assert signals.Figure 2.3 and 2.4 shows the readdata and writedata transfer timing diagram that is used throughout in the project.

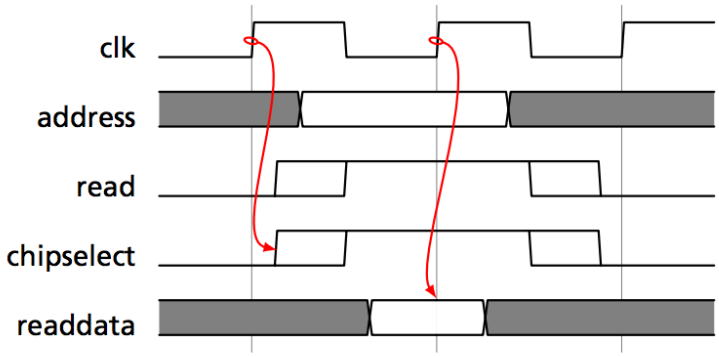



Figure 2.3: Avalon Bus Read Signal

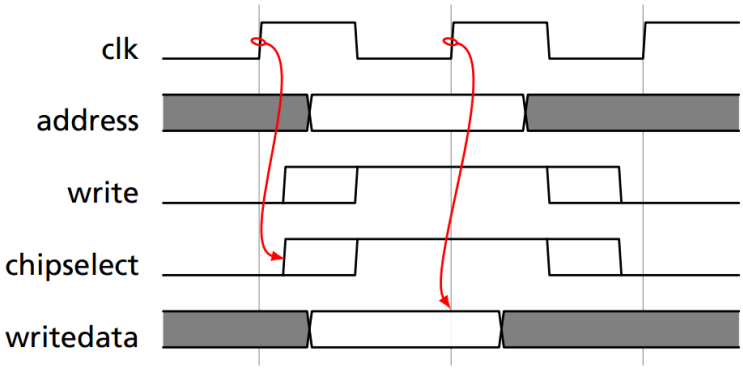



Figure 2.4: Avalon Bus Read Signal

## 2.3 Software Section

The software segment of the system is responsible for generating the input packets and validating the output packets after it has been routed through the switch. This consists of the userspace packet generator and the userspace validator as shown in Figure 2.5 below. The userspace packet generator uses a random number generator to generate data payload of variable length of up to 64 bits in length. It also includes within that packet a header

containing the destination port and the seed number that is used in that generation. This is done to ensure that at the validation side of the userspace, the software program can regenerate the given packet using that same seed number to verify the integrity of the packet. This will be explained in detail in a later section.

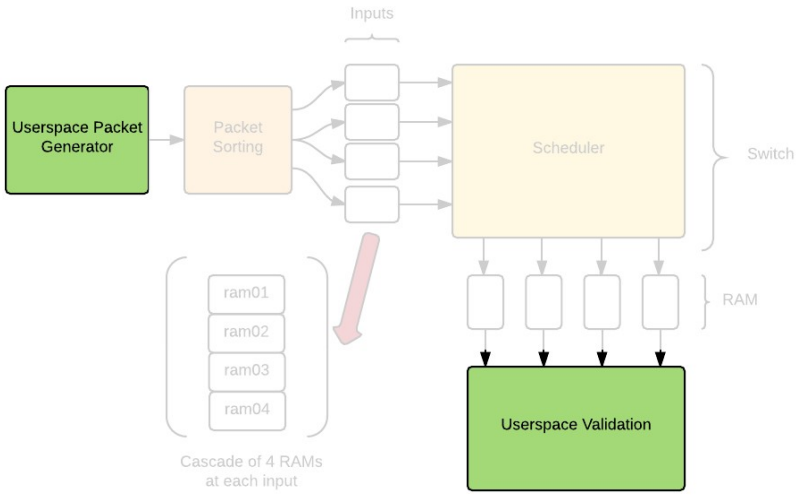



Figure 2.5: Software segment of the system


# 3. Simulation

## 3.1 Introduction

The project depends heavily on simulations, so a robust test suite is created for the simulations. Out of the various compilers available for simulations Verilator was chosen for compiling the hardware code. An exhaustive test bench was created in C++ for interfacing with the compiled hardware code. Now it should be noted that the hardware code that is compiled is actually used in Quartus to compile it down on the hardware and therefore has some nuances and quirks. For example, Altera's compiler limits the number of iterations in the for loop to 255 which Verilator does not. Furthermore there are many such differences between the Verilator simulations and the actual hardware implementation and one should be careful while experimenting.

## 3.2 Simulation Test Bench

Simulating Altera's IP core in Verilator was an integral as well as the most challenging part of the simulation. Since the project's progress contained different IP cores like Fifo , MUX and Ram. In the final design after many iterations several of these IP cores were removed/replaced but that would not have been possible without getting a deeper understanding of timing diagrams as well as the designed issues that needed to be resolved.

SwitchON can simulate altera's IP core into the design using several caveats. For instance, the RAM module is defined in altera_mf.v file in the eda simulation directly, but that file is not standard i.e it cannot be compiled by Verilator, hence several of the other components ( 100k lines) have to be removed. Further more several helper functions needs to be added. Altera uses lots of tri state logic which does not simulate properly in Verilator, these can be removed but then care must be taken to add extra warning lints for, they cause the values to be used in block and no block. The veripool community is very helpful and

some of the scripts they provided using Veripool-perl was instrumental in simulating the altsync RAMS. Again the idea was similar to converting the tri state logics to wire logics.

### 3.2.1 Component Simulation

The test bench can compile each component separately as well as the full model suite. The ingenuity of such a modelling style allowed for amazing level of detailing that can be put to each module. This This allowed to optimize each clock cycle and made us achieve really high throughput through the scheduler. Each IP core has it's quirks and though it was frustrating when they didn't work the expected, it was really a nice learning experience.

### 3.2.2 RAM Simulation

The simulation of altsync RAM was the most challenging part the project. First challenge was actually to find the library in which the module was defined. Running grep system wide did help to locate the module in the eda simulation directory of Quartus. But the RAM that altera uses has lots of tri state logics which prevent the data from coming to the output q port during verilator simulation, in Verilator's defense it did warn about those tri-state logics. Finally converting all such tri-states to wire leads to easy simulation of altsync RAM in verilator [1]. Simulating results for the particular sections is shown in Evaluation.

### 3.2.3 Conclusion

Verilator provides a really easy to use platform which is fast and actually simulates what goes into the hardware. The compilation time is actually nothing compared to Quartus and it provides a natural way to input any random signal into the model so that it can be tested to it's limit. Furthermore the output signals can be verified by just scripting the generated signals. It does have a steep learning curve but it's actually worth simulating.

[1] There is a script by Todd Strader here `https://github.com/twosigma/verilator_support`. It can convert the tri-state logics directly to wire logic. Also, for a quick solution just convert the tri-state logics to wire and comment that section using appropriate verilator escape lint.

# 4. Hardware Design

## 4.1 Interfacing with Software

This is the front interface of the hardware. The packet data coming from the user space is received here. The main function of this module is to channel the packet data into appropriate RAMs. These RAMs are symbolic of the input ports of a network switch.

### 4.1.1 Memory - RAM

The Random Access Memory(RAM) modules are used in the system to simulate the input and output ports. These modules are implemented on the FPGA in the form of an embedded memory IP block supported by the Altera's Mega Wizard plugin in the Altera Quartus software.

A RAM is typically a type of computer data storage that allows data items stored into the memory module to be accessed quickly. It has typically much faster read and write times but is a form of volatile memory that loses its stored data when it loses power

#### The Altera Embedded Memory IP Block

The RAM modules that are implemented on the FPGA are of the form of a Simple dual-port RAM. This supports simultaneous one read and one write operations to different locations which is important in this system to minimise the number of clock cycles required to access data from the RAMs. Figure 4.1 below shows the inputs and outputs that are configured for the RAM module. It takes in the input clock from the overall clock of the system, has a word length of 32 bits and a storage space of 4096 words. These are controlled by the input signals rden(read enable) and wren(write enable).

Figure 4.2 shows the timing diagram of the Altera RAM module.

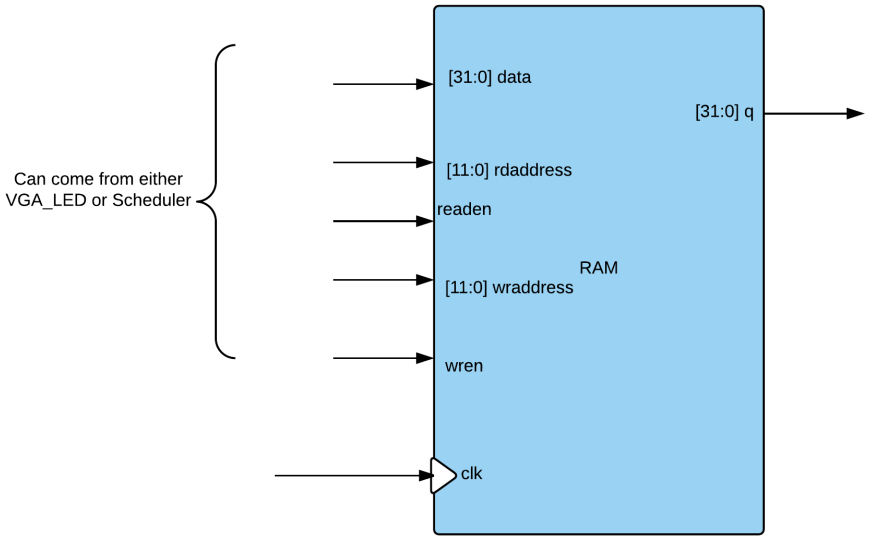



Figure 4.1: Snippet of code showing the inputs and outputs attached to the RAM module

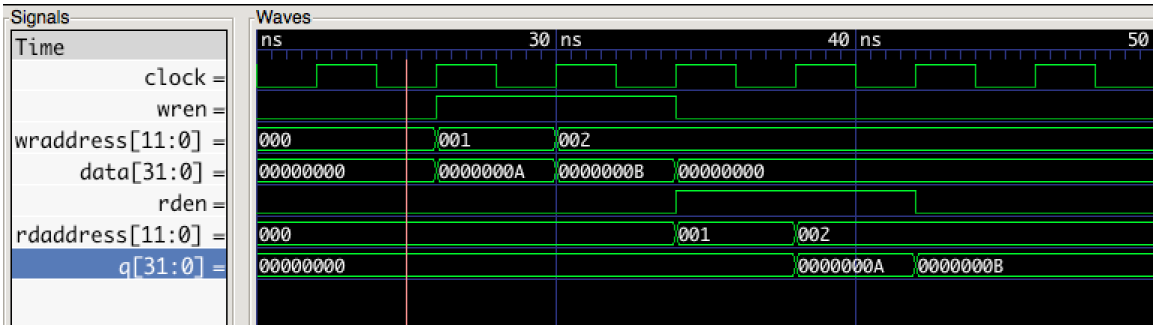



Figure 4.2: Timing diagram of the read and write operations of the RAM

## 4.2 Network Fabric

A network switching fabric is the hardware topology of the network that is laid out and is responsible for transporting the input packet to its respective output port. The network fabric being employed in this project is the crossbar architecture. The crossbar architecture is basically a network topology that is in the form of a matrix as shown in Figure 4.3 below:

### 4.2.1 Crossbar Switch Model

In this project, a single layer 4×4 topology with 4 inputs and 4 outputs is utilised. The Figure 4.3 above illustrates how every input is being connected to every output by the intersections of the matrix, termed crosspoints. The implementation of the crossbar switch model is done on the FPGA. Each input to output connection is completely independent of each other and can therefore support simultaneous communications, except in the case

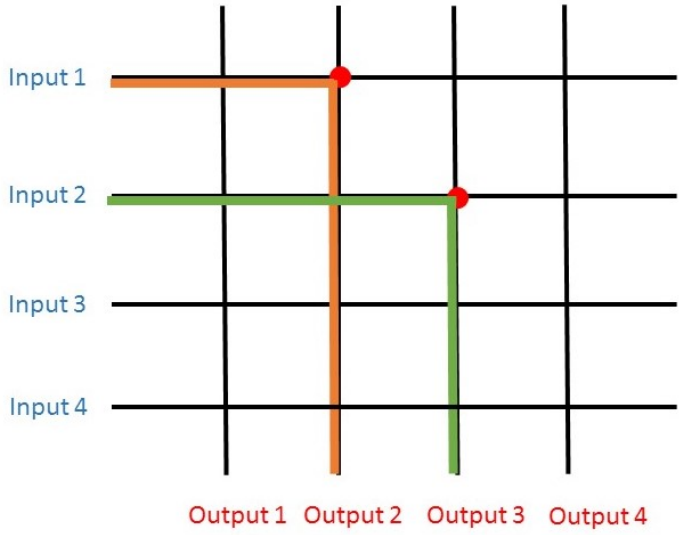



Figure 4.3: Illustration of the Crossbar Architecture that will be responsible for the network switching fabric

when two ports wish to use the same output port.

### How the Crossbar Switch Works

The crossbar switch architecture works in a similar way to that of active addressing in an LED(Light emitting diode) matrix. The inputs are connected to every output by lines that can be turned on and off depending on the destination of the source packet. For example in Figure 4.3, the orange line shows how the input 1 is able to send a packet through the network fabric to output 2 by turning on it's horizontal line and the vertical line that corresponds to output 2. As mentioned earlier, the lines are independent of one another and therefore in a single time slot, both input 1 and input 2 can send packets to outputs 2 and 3 respectively without colliding. Theoretically and in some cases practically it is possible to get $n$[1] number of packets in the output.

### Implementation in the system

In our implementation, 16 RAM modules are used at the input port of the network fabric. This means that each input port of the network fabric has exactly 4 RAMs, one for each output port. They have the above mentioned capacity and word length. This is as shown in Figure 4.4. The functionality of these input rams are to store the packets distributed according to their output ports. When the scheduling algorithm runs and selects the packet to be routed through the network fabric to the output ports, the stored packets are accessed and removed from the input RAMs and routed through. The exact same RAMs are utilised in the output port for storage of the routed packets.

[1] where n is number of output ports,in this case 4

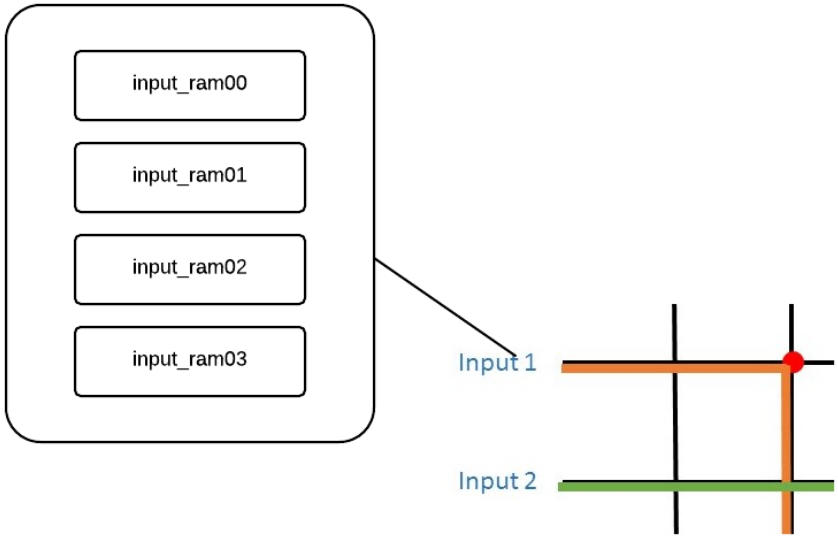



Figure 4.4: An exploded view of a single input port to the network fabric

## 4.3 Scheduling Algorithm

The Scheduling algorithm is at the core of the network switch. The scheduler makes all the decisions regarding the routing of the packets. It looks at the incoming packets and based on the header decides the output port of the packet and routes the content accordingly.

The first preference to the scheduler is always correctness; to make sure none of the packets are lost. The priority is that all the information is transferred as required. Then comes the efficiency. How fast the scheduler can route the data on the input ports utilizing the least number of clock cycles. The scheduling algorithm for our project was also developed in similar two stages.

Another important thing to add is that the hardware code is agnostic to the size of the packet. It marks the start of a packet with a header, containing the port information, followed by unknown number of 32 bit words followed by a 32-bit zero value to mark the end of packet. Once the zero value is encountered, the Scheduler understands that the packet has ended and prepares itself for the next packet.

One performance constraint with both the designs is the RAMs that have been used to simulate the input and output ports. 3 clock cycles are required to analyze and transfer each 32 segment of the data. This is required by the RAM. It takes one clock cycle to increase the address and another for the output to appear. While working with two clock cycles also, sometimes the data would appear late resulting in consistent errors. Another clock cycle has to be spared to make sure that the data is stabilized. So the speeds that are achieved can be

scaled by an appropriate factor considering the real world scenario.

We here discuss the Scheduler algorithm. The initial design focuses only on correctness while the second one tries to improve the performance with some additional hardware logic. Both designs will be discussed below.

### 4.3.1 Single Input Queue Scheduler

The initial design of the scheduler was a simple crossbar switch. The scheduler looks at the head of the packets on different ports, and simply routes the data according to that information. In case of collision, the data is transferred one by one, holding the data at one of the ports while the other one is transferred and then transferring the data from the next port.

The preference in case of a collision is always given to the lower numbered port. What this means is that if there are two packets at ports 1 and 2 both waiting to go to the output port 2, the preference will always be given to the packet at port 1. The upside to this approach is that it is very simple to implement. The downside being that if the next packet on port 1 also has to go to port 2, it will still precede the packet on port 2. This can lead to starvation and theoretically the port 2 packet may never flow through, if all the packets on port 1 are to destined to port 2.

The way the scheduler achieves this is by storing state of different input ports. One variable per port to store the destination port of the current packet coming in through the input port. Another variable is used to indicate the End-of-packet signal which meant the scheduler had to to refresh its transfer information in the next cycle.

#### PPS Architecture

In the second part of the project an attempt to optimize the performance by adding hardware complexity is discussed. Instead of using one RAM per port, four (the number of output ports) such RAMs are used per port. For the user space, the packets are still being sent to the four input ports instead of sixteen. However, a layer inside the hardware divides these packets based on their output destination.

Figure 4.5 show how the Scheduler looks like. the inp[4][4] signals are the input signals from the rams. input_ram_rd_add signals control the address of the ram from which data is being read. The outp[4] signals are the output signals which contain the packets on their destination port and out_ram_wr are to control writes to the output RAMs.

The way it helps is that segregation of packets based on their destination ports greatly improves timing efficiency. A packet meant for the output port two does not have to wait behind another packet meant for output port one. It eliminates the time where one or more output ports has to wait lying empty because none of those packets were at the front of the

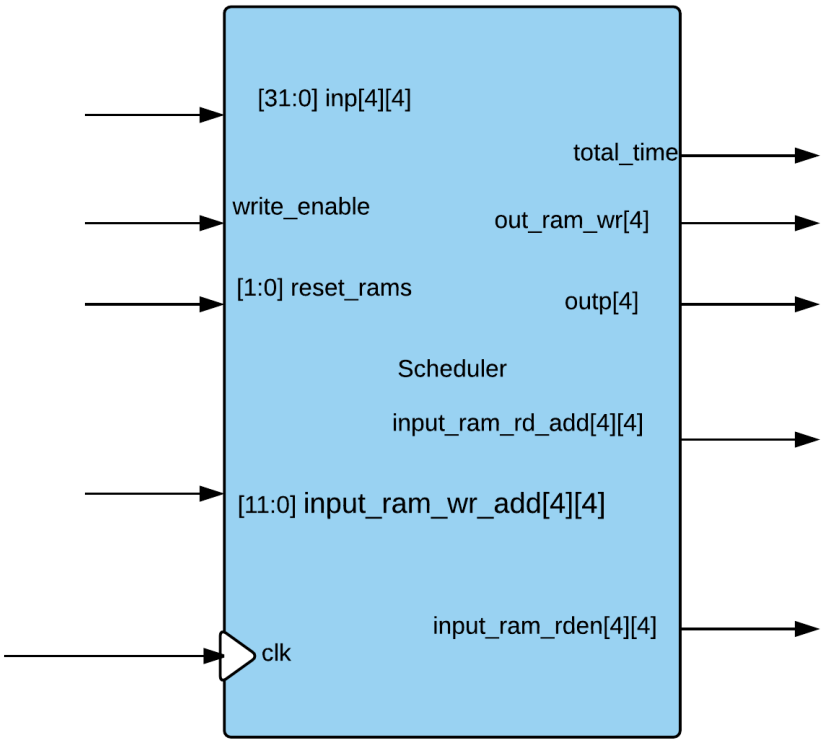



Figure 4.5: Block Diagram of the Scheduler

queue on their respective input ports.

This approach optimizes for different ports but still faces the starvation problem faced by the initial design because here also the preference always goes to the packet on the port one. It still performs better because packets for other ports does not have to be stuck because the front packet cannot pass through.

Figure 4.6 and 4.7 give the timing diagrams of the scheduler implemented. While Figure 4.6 shows the situation where packets to different output ports appear on the input ports, the Figure 4.7 shows the case, where all packets are meant for the same output port.

### 4.3.2 Performance Comparison

A comparison of performance between both Scheduling algorithm is done to investigate the differences. Random test runs of 100 packets of lengths ranging from 4 to 64 uniformly spread across the source ports but randomly across the destination ports. Figure 4.8 shows the relative average speeds achieved by using the two different Schedulers. The performance statistics for the initial design is shown in red while blue highlights the performance of the optimized scheduler.

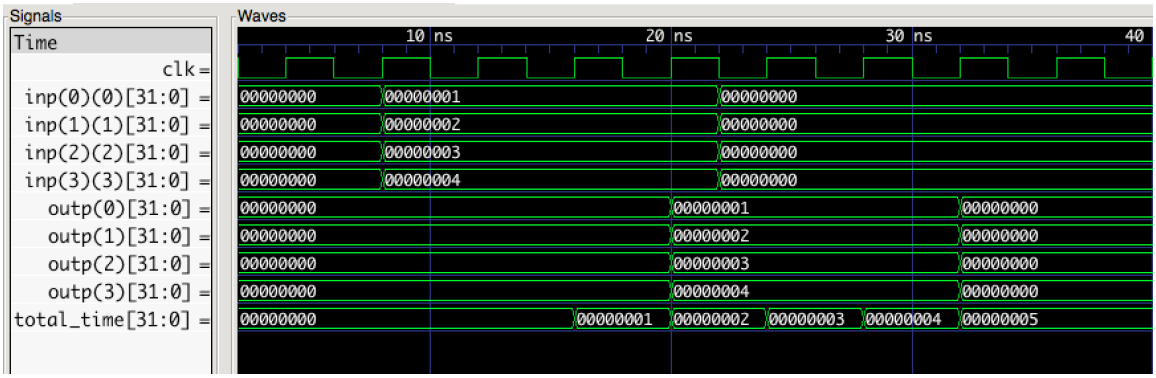



Figure 4.6: Scheduler timing diagram showing packets to different output ports

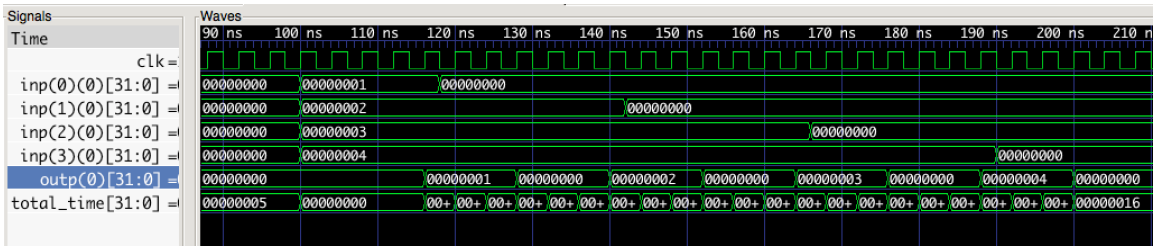



Figure 4.7: Scheduler timing diagram showing packets to the same output port

It is clearly visible that the optimized algorithm shows higher performance both in terms of average as well as the most optimal performance. The performance of the initial design is lower which is consistent with the expectation from the algorithm. While it is easy to see that the average performance is higher, the worst case scenario for both cases occurs when all the packets are scheduled to the same output port. The calculation for such a case is as follows: Since, 32-bits of data is transferred every three clock cycles:

$$Speed = \frac{32}{3} bits/cycle$$

$$Speed = \frac{32}{3 \times 20 \times 10^{-9}}$$

$$Speed = 0.533 \times 10^{9}$$

$$Speed = 508.626\, Mb/s$$

This comes out to be consistent with the data presented in the test runs. If the test bench is modified to ensure that all packets are sent to the same output port, the transfer speed matches the above mentioned speed up to three decimal places.

This also gives a logical explanation for the wider variance seen in the optimized algorithm

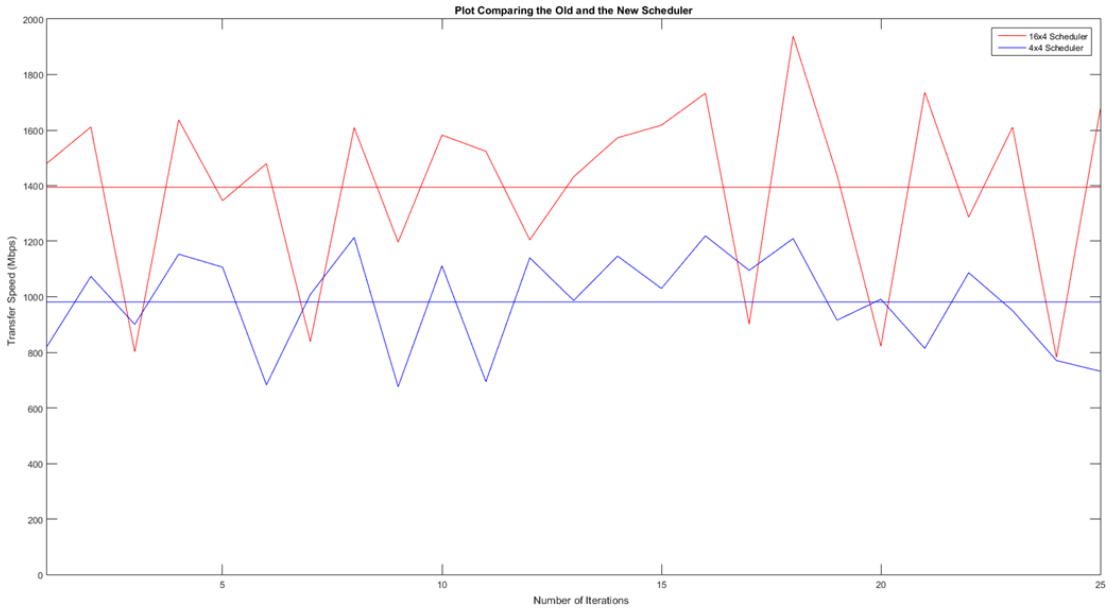



Figure 4.8: Comparison plot of the performance achieved with the two Scheduler algorithm

graph as compared to the initial one. Seen the spectrum for the optimized algorithm is higher with a higher average, the variation and the peaks are also higher.

### 4.3.3 The whole suite

Figure 4.9 shows the block diagram of the entire suite and its interaction signals with the avalon slave bus. Figure 4.10 and 4.11 show the timing diagrams of the flow of packets to and from the user space. It can be seen that the sanctity of the packets is maintained in its movement through the system.

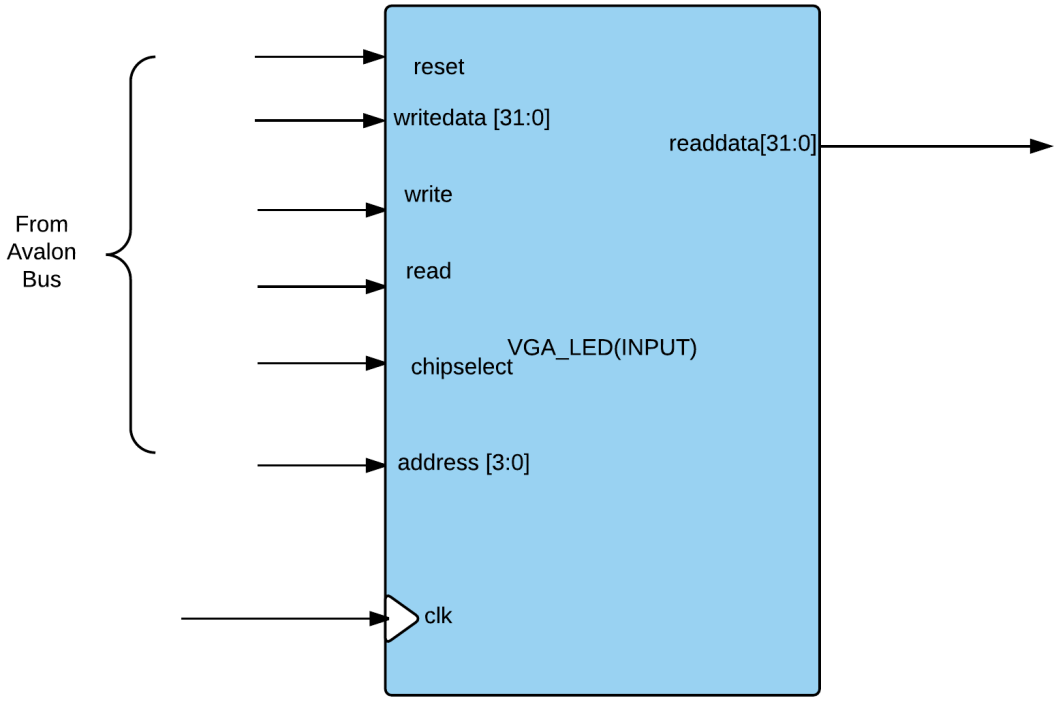



Figure 4.9: Flow of packet from the user space

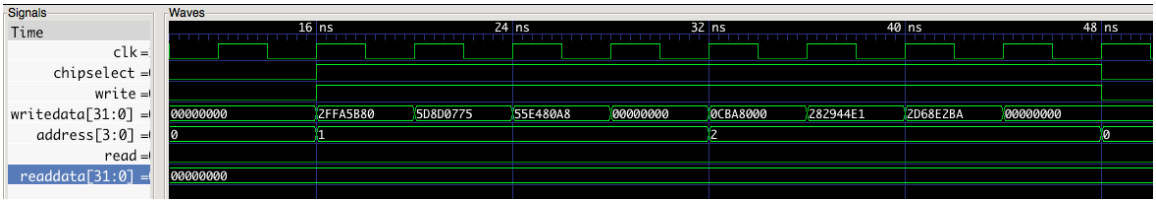



Figure 4.10: Flow of packet from the user space

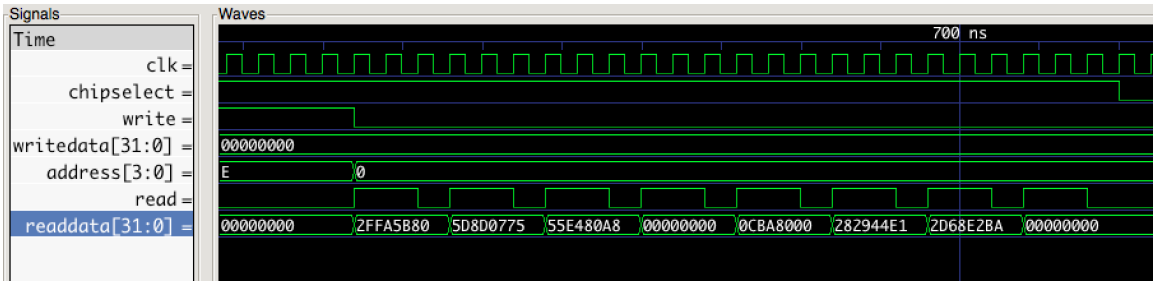



Figure 4.11: Flow of packet to the user space


# 5. Software

The software side talks to the Hardware using the ioctl32 calls. It also generates the packets which are to be routed through the Switch. Furthermore it reads back from the Output RAMS and locally generates the packet and verifies it, if all packets pass the verification it then calculates the throughput through the Switch for that iteration.

## 5.1 Implementation details

The userspace consists of:-

- Packet Generator: It generates seeded random number of packets upto NUM_PACKETS(defined in packetgen.h file).
- Validator: After packet transfer is complete validator runs over the contents of each RAM verifying for consistency in terms of content and port matching.

### 5.1.1 Packet Generator

Packet Generator consists of packetgen.c and packetgen.h. The entry point to these modules is through main.c. Based on the seeded input value it generates a 32 bit random number of which each 8 bits except the first 2 MSB bits have their own `minimum requirements` which are again defined in packetgen.h header file.

Once the packet generator has sent all the packets using the ioctl32 calls before shutting itself off, it sends the WRITE_ENABLE_SCHEDULER and READ_ENABLE opcodes to the module which kicks in the Scheduler.

### 5.1.2 Validator

After the FPGA processing when the packets are routed to their appropriate output ports (modeled by memory locations) the validator runs and checks that the packets should be

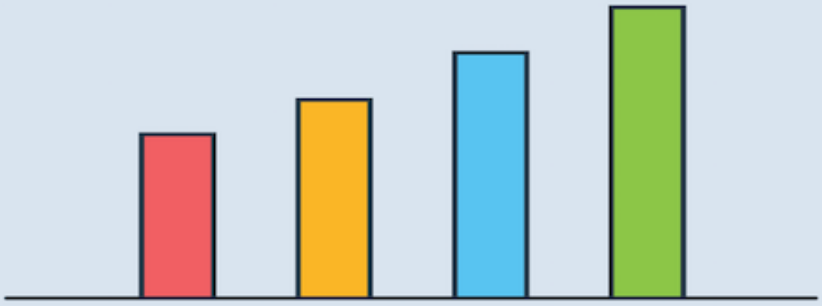
| Last Bits | Output Port |
|---|---|
| 0000 0000 | Port 0 |
| 0000 0001 | Port 1 |
| 0000 0010 | Port 2 |
| 0000 0011 | Port 3 |

Table 5.1: I/O Mapping from RAM

stored on correct memory locations. As discussed above, the generated packet consists of random sequence of bits with the last two bits representing the destination port. The validator makes sure that this values matches the memory space in which the packet is stored and reports any errors encountered.

The validator validates the packets which are received from the output RAMS. Now using the stored seed, the validator seeds itself to stored seed and then starts generating the packets locally. Now each octal of the received packet part is compared against the locally generated octal, if match happens it that part of the packet is marked OK and the validator moves to check other packet. Now should a packet not match the generated seed, error is thrown and the program exits. For the simulation scenario,it is ensured that none of the packets are dropped and none of the packets are wrongly stored.

After all the validation passes, the validator sends an OP-Code to the FPGA which then returns the total_clock_cycles it took to transmit that data. Using this information the throughput of the Switch for that particular iteration can be calculated. It has been observed that both in case of PPS and Single Input Architecture the throughput fairly remains constant with a small swing along the average $\pm$ 200 Mbits/s, which is fair in terms of packets that are being sent. In other words since the packets are being generated randomly it might so happen that the generation might be skewed towards a particular output port, which leads to number of cycles being increased as packets are now queued thus leading to more time for transfer. This issue will be fairly common in both the architecture because all the packets can now go to only one part,so in each clock cycle a single packet will be transferred, in other words packet transfer would be linear.



# 6. Evaluation

## 6.1 FPGA Switch Performance

Having implemented and tested the functionality of the FPGA switch, the next step would be to evaluate the performance of the switch and how it performs under various types of load.

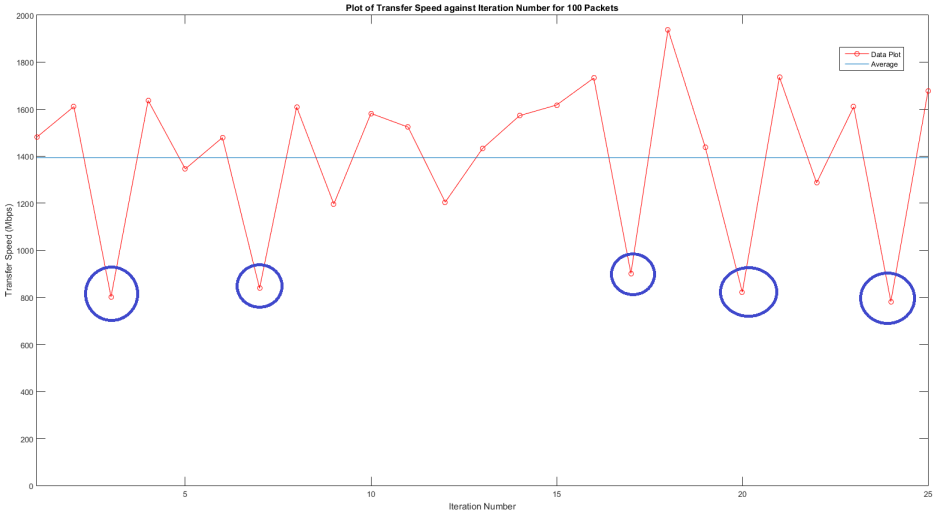



Figure 6.1: Plot showing the data transfer speed over 25 iterations

The first test is to measure and find the average speed of data transfer that the switch is capable of achieving. This is shown in Figure 6.1 above. It can be seen that there are massive fluctuations in the data plot, there are however some points to note which are circled in

blue. These points are outliers in the data plot that occurs whenever there is a concentrated number of packets that are sent to a specific port. Due to the random distribution of packets that are sent to each destination port, there will be a case where for example 50 of the 100 packets generated are destined to output port 3. Such a data point will then result in an outlier where the data transfer speed is severely crippled because a higher number of clock cycles are required to process that concentration of packets destined for a single output port. The average transfer rate of the switch including these outliers still remains at an impressive speed of approximately 1400 Megabits/second (Mbps)

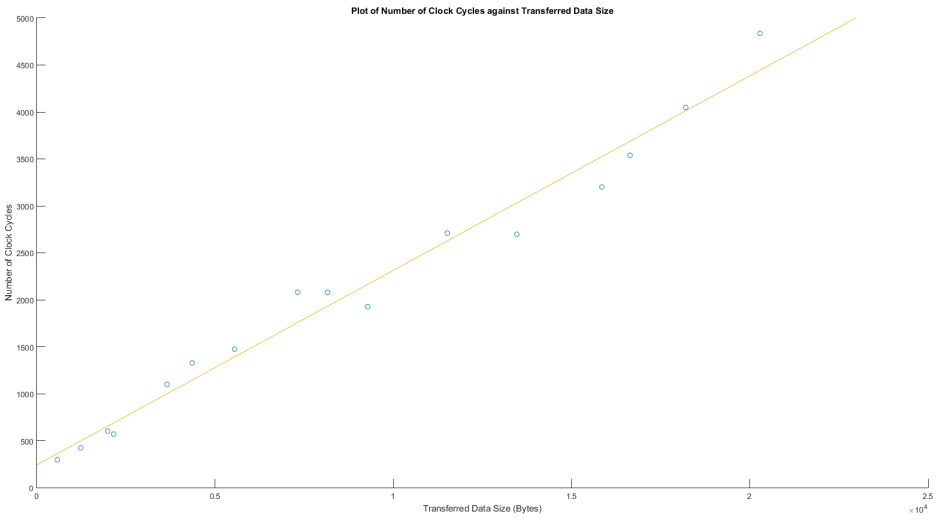



Figure 6.2: Plot showing the the number of clock cycles needed to process a given data size

The second test involves incrementing the total transferable data size to investigate how the number of clock cycles required to complete the routing changes. It can be seen from the Figure 6.2 that the number of clock cycles required increments linearly with increasing data sizes. The gradient of the slope gives the data transfer speed at that given point, seeing as how the graphs proves to be a linear plot, it is safe to say that the transfer speed of the switch remains constant regardless of the transferable data size. This means that under heavy data loads, the switch will still be able to perform at its maximum capacity.


# 7. Conclusion

Using the simulated Switch implemented in hardware, it can be concluded that for all real cases in which the packet arrival can be Poisson, PPS architecture would be really helpful because then Head of line blocking can be avoided and throughput would increase as can be seen from the results. Furthermore it's futile to expect that the throughput would increase in the order of the input ports because of the distribution in which the packets arrive. Though this can be avoided using fixed input ports for a particular packet sizes, but is not entirely avoidable.

## 7.1 Lessons Learnt

Throughout the course of the project, there are a few lessons to be learnt:

- Hardware is hard. Like seriously programming hardware is very different from working on just purely software. In software, logic take precedence where a good logic will mean an efficient and perfectly functioning piece of code. This differs greatly in hardware where logic has to be perfect but also timing of every hardware module must be taken into consideration due to data stability reasons,few modules might be slow in giving out the data so everything cannot work as per the designed clock cycles.
- Simulating what hardware does in software that produces timing diagrams is the best way to debug hardware issues. Timing diagrams while tedious and time-consuming to do and set up, provide an insight into what the hardware is actually doing, saving you time in the end.
- While simulations provide insight, it may not be truly representative of what actually goes on in hardware. More often than not, the simulations hold true. But every once in awhile, it goes way off tangent so always check what the actual hardware is telling you. For example there were cases in which Verilator would actually simulate the

altsync ram, but there would be no output on the q port. Furthermore Quartus limits the number of iterations to 255 in a for loop, but since verialtor is platform agnostic it synthesizes the code,hence there might be a case in which the logic would work in Simulation only to fail in hardware.

- Hardware documentation is not as robust as those that you will find on open sourced stuff such as python libraries. Simulating and creating test benches early and often is a nice way to debug. Also the simulated code should be as close as possible to the code synthesized in Quartus.

## 7.2 Future Work

These are the below directions that can be taken to take the project forward.

- DMA can be implemented so that the simulation can be run for large number of packets.
- Different scheduling algorithms, which are less greedy in practice can be simulated to check the perfomance.

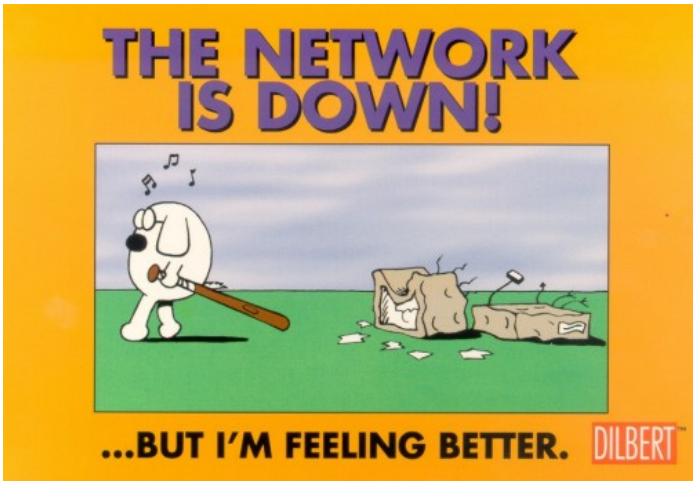



Figure 7.1: Finally

# 8. Appendix

## 8.1 File Listings

Following are the files included:

1. Hardware
   (a) **VGA_LED.sv** - Interfaces with Avalon slave. Responsible to handle the incoming packets from the Slave bus.
   (b) **Scheduler.sv** - Routes the data through the Switch. Contains the PPS algorithm.
   (c) **Buffer.sv** - Interfaces with the avalon slave again. Responsible to send the packets through the slave bus to the validator.
   (d) **oScheduler.sv** - The old scheduler implementation with single input queues. For reference purposes.
   (e) Does not include the RAM ipcore files generated by Mega Wizard required for simulation.
2. Verilator
   (a) **vgacounter.cpp** - cpp file used to simulate the top level vcd. Includes appropriate signal changes throughout the Switch.
   (b) **schedulercounter.cpp** - cpp file to simulate the scheduler in verilator.
   (c) **ramcounter.cpp** - cpp file to simulate the ram
   (d) **buffercounter.cpp** -
   (e) **Makefile**
   (f) Does not include the modified RAM files used by verilator for compilation.
3. Software
   (a) **main.c** - top level file used to generate and send packets through the avalon bus.
   (b) **packetgen.h** - header file for the packet generator.
   (c) **packetgen.c** - packet generator file reponsible for generating the packets. Used by main.c

(d) **validator.c** - validator responsible for reading the packets from output RAMs. Validates the count, sequence and length and also calculates the transfer speed.
(e) **vga_led.h** - header file for vga_led.c
(f) **vga_led.c** - vga_led.c file similar included as part of lab3, with code changes to support 32 bit transfers, 4-bit addresses and read from the slave.
(g) **Makefile** - make main for main.c and make validator for validator.c. make for vga_led.c and insmod for installing it to the kernel

**Hardware:VGA_LED.sv**

```
//START_MODULE_NAME ------------------------------------------------------------
//
// Module Name      :  VGA_LED
//
// Description      :  Reads values from RAMS and enables the scheduler.
//
// Limitation       : None
//
// Results expected:  Enables the Scheduler and Buffer, communicate with
//                    ioctl.
//
//END_MODULE_NAME --------------------------------------------------------------

module VGA_LED(input logic      clk,
        input logic       reset,
        input logic [31:0]  writedata,
        input logic       write, read,
        input               chipselect,
        input logic [3:0]   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 [31:0] readdata);

    // Naming convention is the part of module the signal is for
    followed by
    // the use of the signal, written in camel case. For example,
    fifo_in
    logic [31:0]    inp[4][4], outp[4], input_ram_wr_in[4][4];
    logic [11:0]    input_ram_rd_add[4][4], input_ram_wr_add[4][4];
    logic           input_ram_rden[4][4], input_ram_wren[4][4];
  logic           out_ram_wr[4];

    // logic signals to enable write and read to the output RAM.
    logic           write_enable, read_enable;
    // signal to reset rams. Not being used right now. Was giving us
    problems.
    // We burn the hardware again after each test run of packets.
    // The only visible option.
    logic [1:0]     reset_rams;
    //  Calculates the number of clock cycles it takes to transfer the
    entire
    //  data from the input rams. Necessary to calculate the effective
    speed.
    logic [31:0]    total_time;
    logic [1:0]     port[4];
    logic           eop[4];
```

```
44
45  initial begin
46    reset_rams = 0; write_enable = 0; read_enable = 0; total_time = 0;
47        for(int i=0; i<4; i++)begin
48            for(int j=0; j<4; j++)begin
49                input_ram_rd_add[i][j] = 0;
50                input_ram_wr_add[i][j] = 0;
51                input_ram_rden[i][j] = 0;
52                input_ram_wren[i][j] = 0;
53            end
54            port[i] = 0;
55            eop[i] = 1;
56        end
57  end
58
59    //Incoming packets modeled as 16 rams, 1 for each combination of
     input and
60    //output port
61   RAM input_ram00(.clock(clk), .data(input_ram_wr_in[0][0]),
62        .rdaddress(input_ram_rd_add[0][0]), .rden(input_ram_rden[0][0]),
63          .wraddress(input_ram_wr_add[0][0]), .wren(input_ram_wren
     [0][0]), .q(inp[0][0]));
64   RAM input_ram01(.clock(clk), .data(input_ram_wr_in[0][1]),
65        .rdaddress(input_ram_rd_add[0][1]), .rden(input_ram_rden[0][1]),
66          .wraddress(input_ram_wr_add[0][1]), .wren(input_ram_wren
     [0][1]), .q(inp[0][1]));
67   RAM input_ram02(.clock(clk), .data(input_ram_wr_in[0][2]),
68        .rdaddress(input_ram_rd_add[0][2]), .rden(input_ram_rden[0][2]),
69          .wraddress(input_ram_wr_add[0][2]), .wren(input_ram_wren
     [0][2]), .q(inp[0][2]));
70   RAM input_ram03(.clock(clk), .data(input_ram_wr_in[0][3]),
71        .rdaddress(input_ram_rd_add[0][3]), .rden(input_ram_rden[0][3]),
72          .wraddress(input_ram_wr_add[0][3]), .wren(input_ram_wren
     [0][3]), .q(inp[0][3]));
73   RAM input_ram10(.clock(clk), .data(input_ram_wr_in[1][0]),
74        .rdaddress(input_ram_rd_add[1][0]), .rden(input_ram_rden[1][0]),
75          .wraddress(input_ram_wr_add[1][0]), .wren(input_ram_wren
     [1][0]), .q(inp[1][0]));
76   RAM input_ram11(.clock(clk), .data(input_ram_wr_in[1][1]),
77        .rdaddress(input_ram_rd_add[1][1]), .rden(input_ram_rden[1][1]),
78          .wraddress(input_ram_wr_add[1][1]), .wren(input_ram_wren
     [1][1]), .q(inp[1][1]));
79   RAM input_ram12(.clock(clk), .data(input_ram_wr_in[1][2]),
80        .rdaddress(input_ram_rd_add[1][2]), .rden(input_ram_rden[1][2]),
81          .wraddress(input_ram_wr_add[1][2]), .wren(input_ram_wren
     [1][2]), .q(inp[1][2]));
82   RAM input_ram13(.clock(clk), .data(input_ram_wr_in[1][3]),
83        .rdaddress(input_ram_rd_add[1][3]), .rden(input_ram_rden[1][3]),
84          .wraddress(input_ram_wr_add[1][3]), .wren(input_ram_wren
     [1][3]), .q(inp[1][3]));
85   RAM input_ram20(.clock(clk), .data(input_ram_wr_in[2][0]),
86        .rdaddress(input_ram_rd_add[2][0]), .rden(input_ram_rden[2][0]),
```

```
87          .wraddress(input_ram_wr_add[2][0]), .wren(input_ram_wren
    [2][0]), .q(inp[2][0]));
88    RAM input_ram21(.clock(clk), .data(input_ram_wr_in[2][1]),
89          .rdaddress(input_ram_rd_add[2][1]), .rden(input_ram_rden[2][1]),
90          .wraddress(input_ram_wr_add[2][1]), .wren(input_ram_wren
    [2][1]), .q(inp[2][1]));
91    RAM input_ram22(.clock(clk), .data(input_ram_wr_in[2][2]),
92          .rdaddress(input_ram_rd_add[2][2]), .rden(input_ram_rden[2][2]),
93          .wraddress(input_ram_wr_add[2][2]), .wren(input_ram_wren
    [2][2]), .q(inp[2][2]));
94    RAM input_ram23(.clock(clk), .data(input_ram_wr_in[2][3]),
95          .rdaddress(input_ram_rd_add[2][3]), .rden(input_ram_rden[2][3]),
96          .wraddress(input_ram_wr_add[2][3]), .wren(input_ram_wren
    [2][3]), .q(inp[2][3]));
97    RAM input_ram30(.clock(clk), .data(input_ram_wr_in[3][0]),
98          .rdaddress(input_ram_rd_add[3][0]), .rden(input_ram_rden[3][0]),
99          .wraddress(input_ram_wr_add[3][0]), .wren(input_ram_wren
    [3][0]), .q(inp[3][0]));
100   RAM input_ram31(.clock(clk), .data(input_ram_wr_in[3][1]),
101         .rdaddress(input_ram_rd_add[3][1]), .rden(input_ram_rden[3][1]),
102         .wraddress(input_ram_wr_add[3][1]), .wren(input_ram_wren
    [3][1]), .q(inp[3][1]));
103   RAM input_ram32(.clock(clk), .data(input_ram_wr_in[3][2]),
104         .rdaddress(input_ram_rd_add[3][2]), .rden(input_ram_rden[3][2]),
105         .wraddress(input_ram_wr_add[3][2]), .wren(input_ram_wren
    [3][2]), .q(inp[3][2]));
106   RAM input_ram33(.clock(clk), .data(input_ram_wr_in[3][3]),
107         .rdaddress(input_ram_rd_add[3][3]), .rden(input_ram_rden[3][3]),
108         .wraddress(input_ram_wr_add[3][3]), .wren(input_ram_wren
    [3][3]), .q(inp[3][3]));
109
110   Scheduler scheduler(.*);
111   Buffer buffer(.*);
112
113   always_ff @(posedge clk)begin
114     if(reset_rams == 1) begin
115       reset_rams = 2;
116     end
117     else if(reset_rams == 2)begin
118       reset_rams = 0;
119     end
120
121         for(int i=0; i<4; i++) begin
122             for(int j=0; j<4; j++) begin
123                 if(input_ram_wren[i][j])begin
124                     input_ram_wren[i][j] = 0;
125                     input_ram_wr_add[i][j] = input_ram_wr_add[i][j] + 1;
126                 end
127             end
128         end
129
130     if (chipselect && write) begin
```

```
131     case(address)
132       0 : begin
133           // If the previous packet has finished
134           // transferring (characterized by 32 bit zero values, the
135           // port information has to be re-established from the
136           // packet header.
137                 if(eop[0] && writedata) begin
138                     eop[0] = 0;
139                     port[0] = writedata[1:0];
140                 end
141                 // If in between transfer of a packet, continue
142                 // transferring to the same port.
143                 if(!eop[0]) begin
144                     for(int i=0; i<4; i++)begin
145                         if(port[0] == i) begin
146                             input_ram_wr_in[0][i] = writedata;
147                             input_ram_wren[0][i] = 1;
148                         end
149                     end
150                 end
151                 // If the end of packet is reached(32 bit zero value
    ), eop
152                 // signal is set to high. In the next cycle the port
153                 // information will be re-established.
154                 if(!writedata)begin
155                     eop[0] = 1;
156                 end
157       end
158
159       1 : begin
160                 if(eop[1] && writedata) begin
161                     eop[1] = 0;
162                     port[1] = writedata[1:0];
163                 end
164                 if(!eop[1]) begin
165                     for(int i=0; i<4; i++)begin
166                         if(port[1] == i) begin
167                             input_ram_wr_in[1][i] = writedata;
168                             input_ram_wren[1][i] = 1;
169                         end
170                     end
171                 end
172                 if(!writedata)begin
173                     eop[1] = 1;
174                 end
175       end
176
177       2 : begin
178                 if(eop[2] && writedata) begin
179                     eop[2] = 0;
180                     port[2] = writedata[1:0];
181                 end
```

```
182                     if(!eop[2]) begin
183                         for(int i=0; i<4; i++)begin
184                             if(port[2] == i) begin
185                                 input_ram_wr_in[2][i] = writedata;
186                                 input_ram_wren[2][i] = 1;
187                             end
188                         end
189                     end
190                     if(!writedata)begin
191                         eop[2] = 1;
192                     end
193         end
194
195         3 : begin
196                     if(eop[3] && writedata) begin
197                         eop[3] = 0;
198                         port[3] = writedata[1:0];
199                     end
200                     if(!eop[3]) begin
201                         for(int i=0; i<4; i++)begin
202                             if(port[3] == i) begin
203                                 input_ram_wr_in[3][i] = writedata;
204                                 input_ram_wren[3][i] = 1;
205                             end
206                         end
207                     end
208                     if(!writedata)begin
209                         eop[3] = 1;
210                     end
211         end
212         // Special signal to control the flow of data within the
213         // switch from input port to output port. Required to
214         // specifically determine the number of cycles it took for
215         // data transfer and hence the speed.
216                 15 : write_enable = 1;
217                 // Controls the read from the output rams. Not really
218                 // necessary, but we have added this in our user space
    code
219                 // and may have a valid use case.
220         14 : read_enable = 1;
221         // Reset all rams. Not being used.
222         13 : begin
223                     for(int i=0; i<4; i++)begin
224                         for(int j=0; j<4; j++)begin
225                             input_ram_wr_add[i][j] = 0;
226                         end
227                     end
228           reset_rams = 1;
229          end
230       endcase
231     end
232     else begin
```

```
233         // Disable rights to all rams, if write was low.
234             for(int i=0; i<4; i++)begin
235                 for(int j=0; j<4; j++)begin
236                     input_ram_wren[i][j] = 0;
237                 end
238             end
239         end
240   end
241 endmodule
```

**Hardware:Scheduler.sv**

```
1  //START_MODULE_NAME
       ------------------------------------------------------------------
2  //
3  // Module Name      :  Scheduler
4  //
5  // Description      :  Reads values from RAMS and schedules to prevent
6  //                     collisions.
7  //
8  // Limitation       :  NONE
9  //
10 // Results expected:  Packets routed to proper ports.
11 // //
12 //END_MODULE_NAME
       ------------------------------------------------------------------
13
14
15 module Scheduler(input logic clk,
16         input logic [31:0]  inp[4][4],
17         input logic         write_enable,
18         input logic [1:0]   reset_rams,
19         input logic [11:0]  input_ram_wr_add[4][4],
20
21         output logic [31:0] total_time,
22         output logic        out_ram_wr[4],
23         output logic [31:0] outp[4],
24         output logic [11:0] input_ram_rd_add[4][4],
25         output logic        input_ram_rden[4][4]);
26
27     //Write cycle, to make sure that the signal is stable on the output
       wires
28     //of the RAMs. It usually takes three clock cycles for the data to
29     //stabilize: one clock for the address to be incremented, second for
        the
30     //data to be appear on the output wire. Theoritically, it should
       take two,
31     //but sometimes there was a delay and it didnt. Hence, added the
       third.
32     logic [1:0] write_cycle;
33     // For end of packets.
34     logic       eop[4];
35     // Source packet information.
36     logic [1:0] sport[4];
37     // To determine if the total time should be incremented.
38     logic       time_inc;
39
40     initial begin
41         write_cycle = 0;
42         for(int i=0; i<4; i++) begin
43             eop[i] = 1;
44             sport[i] = 0;
```

```
45          end
46      end
47
48      always_ff @(posedge clk) begin
49          // Reset ram code. Not being used.
50          if(reset_rams) begin
51              for(int i=0; i<4; i++) begin
52                  for(int j=0; j<4; j++) begin
53                      input_ram_rd_add[i][j] = 0;
54                  end
55              end
56          end
57
58          // If the write enable is high.
59          if(write_enable) begin
60              time_inc = 0;
61              for(int i=0; i<4; i++)begin
62                  for(int j=0; j<4; j++)begin
63                      // We tried setting these read signals high once and
     for
64                      // all in the beginning, but if the ram is empty
    this
65                      // tends to go low. Hence, doing this in every cycle
    . May
66                      // be a better way, but going with brute force to
    avoid
67                      // any unnecessary nuisance,
68                      input_ram_rden[i][j] = 1;
69                      // time_inc = 1 if for any ram, the read address is
    less
70                      // than the right address, which means there is data
     to be
71                      // read.
72                      time_inc = time_inc |
73                                  (input_ram_rd_add[i][j] <
    input_ram_wr_add[i][j]);
74                  end
75              end
76              total_time = total_time + time_inc;
77
78              if(write_cycle==2) begin
79                  write_cycle = 0;
80                  // Here i represents the output rams and corresponding j
    ,i
81                  // represent the input ram from which the information is
82                  // flowing. So, in essence internally its a 16x4 flow
    network.
83                  for(int i=0; i<4; i++)begin
84                      for(int j=0; j<4; j++) begin
85                          //Similar to the code in VGA_LED.
86                          //If eop is reached and there is a next packet,
    set
```

```
87                          //eop low and set the port informtion.
88                          if(eop[i] && inp[j][i] &&
89                                  input_ram_rd_add[j][i] <
    input_ram_wr_add[j][i]) begin
90                              eop[i] = 0;
91                              sport[i] = j;
92                          end
93                          // If eop is not reached(eop is low), check from
     which
94                          // input roam is the information is flowing,
    transfer
95                          // the word and increment the address for the
    next
96                          // word. Also, if the word is empty, set eop
    high.
97                          if(!eop[i] && sport[i]==j)begin
98                              outp[i] = inp[j][i];
99                              out_ram_wr[i] = 1;
100                             input_ram_rd_add[j][i] = input_ram_rd_add[j
    ][i] + 1;
101                             if(!inp[j][i])begin
102                                 eop[i] = 1;
103                 break;
104                             end
105                         end
106                     end
107                 end
108             end
109             else begin
110                 write_cycle = write_cycle + 1;
111                 for(int i=0; i<4; i++) begin
112                     // Set write enable signals to the rams low.
113                     out_ram_wr[i] = 0;
114                 end
115             end
116         end
117     end
118 endmodule
```

**Hardware: Buffer.sv**

```
1
2  //START_MODULE_NAME
       ------------------------------------------------------------------
3  //
4  // Module Name      :  Buffer
5  //
6  // Description      :  Stores the data coming from Scheduler into the
       RAMS
7  //
8  // Limitation       :  NONE
9  //
10 // Results expected:  Packets stored with appropriate lengths to proper
       RAM.
11 // //
12 //END_MODULE_NAME
       ------------------------------------------------------------------
13
14
15 module Buffer(input logic clk,
16         input logic          chipselect, read, read_enable,
17      input logic [1:0]    reset_rams,
18         input logic [3:0]    address,
19         input logic [31:0]   outp[4],
20         input logic          out_ram_wr[4],
21         input logic [31:0]   total_time,
22
23         output logic [31:0] readdata);
24
25      // Output RAM signals. Read & Write address, enable signals and
       output
26      // signals.
27      logic[11:0] ram0_rdaddress, ram1_rdaddress, ram2_rdaddress,
       ram3_rdaddress;
28      logic[11:0] ram0_wraddress, ram1_wraddress, ram2_wraddress,
       ram3_wraddress;
29    logic        ram0_wren, ram1_wren, ram2_wren, ram3_wren;
30      logic        ram0_rden, ram1_rden, ram2_rden, ram3_rden;
31      logic[31:0] ram0_q, ram1_q, ram2_q, ram3_q;
32      // read cycle signals to ensure that address is incremented only
       once
33      // while reading from the RAM. We toggle these logic signals to
       ensure
34      // that all the work at Buffer happens only during one clock cycle
       out
35      // of the two used by the Avalon bus.
36      logic        read_cycle0, read_cycle1, read_cycle2, read_cycle3;
37
38      // Four output RAMs that model the four output ports.
39     RAM output_ram0(.clock(clk), .data(outp[0]), .rdaddress(
       ram0_rdaddress),
```

```
40          .rden(ram0_rden), .wraddress(ram0_wraddress), .wren(ram0_wren),
41          .q(ram0_q));
42      RAM output_ram1(.clock(clk), .data(outp[1]), .rdaddress(
    ram1_rdaddress),
43          .rden(ram1_rden), .wraddress(ram1_wraddress), .wren(ram1_wren),
44          .q(ram1_q));
45      RAM output_ram2(.clock(clk), .data(outp[2]), .rdaddress(
    ram2_rdaddress),
46          .rden(ram2_rden), .wraddress(ram2_wraddress), .wren(ram2_wren),
47          .q(ram2_q));
48      RAM output_ram3(.clock(clk), .data(outp[3]), .rdaddress(
    ram3_rdaddress),
49          .rden(ram3_rden), .wraddress(ram3_wraddress), .wren(ram3_wren),
50          .q(ram3_q));
51
52      initial begin
53        ram0_wraddress = 0; ram1_wraddress = 0; ram2_wraddress = 0;
    ram3_wraddress = 0;
54      ram0_rdaddress = 0; ram1_rdaddress = 0; ram2_rdaddress = 0;
    ram3_rdaddress = 0;
55          ram0_wren = 0; ram1_wren = 0; ram2_wren = 0; ram3_wren = 0;
56          ram0_rden = 0; ram1_rden = 0; ram2_rden = 0; ram3_rden = 0;
57          read_cycle0 = 1; read_cycle1 = 1; read_cycle2 = 1; read_cycle3 =
     1;
58      end
59
60      // We store the values in the outp[i] signals in the RAM passed by
    the
61      // Scheduler along with the write signals controlled by the same.
62      // Here we are delaying the storage by one clock cycle just to make
    sure
63      // that the signal is strong when we save it to the RAM.
64      always_ff @(posedge clk) begin
65      if(reset_rams)begin
66        ram0_wraddress <= 0; ram1_wraddress <= 0; ram2_wraddress <= 0;
    ram3_wraddress <= 0;
67      end
68          if(out_ram_wr[0])
69              if(ram0_wren)
70                  ram0_wraddress <= ram0_wraddress + 1;
71              else
72                  ram0_wren <= 1;
73          else
74              if(ram0_wren)begin
75                  ram0_wren <= 0;
76                  ram0_wraddress <= ram0_wraddress + 1;
77              end
78
79          if(out_ram_wr[1])
80              if(ram1_wren)
81                  ram1_wraddress <= ram1_wraddress + 1;
82              else
```

```
83                  ram1_wren <= 1;
84          else
85              if(ram1_wren)begin
86                  ram1_wren <= 0;
87                  ram1_wraddress <= ram1_wraddress + 1;
88              end
89
90          if(out_ram_wr[2])
91              if(ram2_wren)
92                  ram2_wraddress <= ram2_wraddress + 1;
93              else
94                  ram2_wren <= 1;
95          else
96              if(ram2_wren)begin
97                  ram2_wren <= 0;
98                  ram2_wraddress <= ram2_wraddress + 1;
99              end
100
101         if(out_ram_wr[3])
102             if(ram3_wren)
103                 ram3_wraddress <= ram3_wraddress + 1;
104             else
105                 ram3_wren <= 1;
106         else
107             if(ram3_wren)begin
108                 ram3_wren <= 0;
109                 ram3_wraddress <= ram3_wraddress + 1;
110             end
111     end
112
113     // The read signals to all the RAMs are turned high as soon as the
114     // read_enable signal is turned on.
115     always_ff @(posedge clk) begin
116         if(read_enable)begin
117             ram0_rden <= 1; ram1_rden = 1; ram2_rden = 1; ram3_rden = 1;
118         end
119     end
120
121     // Block to control the reads from the output RAM.
122     always_ff @(posedge clk) begin
123     if(reset_rams)begin
124       ram0_rdaddress <= 0; ram1_rdaddress <= 0; ram2_rdaddress <= 0;
    ram3_rdaddress <= 0;
125     end
126         if(chipselect && read) begin
127             case(address)
128                 7 : readdata <= total_time;
129                 8 : readdata <= ram0_rdaddress;
130                 9 : readdata <= ram1_rdaddress;
131                 10 : readdata <= ram2_rdaddress;
132                 11 : readdata <= ram3_rdaddress;
133                 12 : readdata <= ram0_wraddress;
```

```
134                 13 : readdata <= ram1_wraddress;
135                 14 : readdata <= ram2_wraddress;
136                 15 : readdata <= ram3_wraddress;
137
138                 0 : begin
139                     if(ram0_rdaddress <= ram0_wraddress) begin
140                         if(read_cycle0) begin
141                             ram0_rdaddress <= ram0_rdaddress + 1;
142                             read_cycle0 <= 0;
143                              readdata <= ram0_q;
144                        end
145                         else begin
146                             read_cycle0 <= 1;
147                         end
148                     end
149               else
150                 readdata <= ram0_q;
151                 end
152
153                 1 : begin
154                     if(ram1_rdaddress <= ram1_wraddress) begin
155                         if(read_cycle1) begin
156                             ram1_rdaddress <= ram1_rdaddress + 1;
157                             read_cycle1 <= 0;
158                              readdata <= ram1_q;
159                        end
160                         else begin
161                             read_cycle1 <= 1;
162                         end
163                     end
164               else
165                 readdata <= ram1_q;
166                 end
167
168                 2 : begin
169                     if(ram2_rdaddress <= ram2_wraddress) begin
170                         if(read_cycle2) begin
171                             ram2_rdaddress <= ram2_rdaddress + 1;
172                             read_cycle2 <= 0;
173                             readdata <= ram2_q;
174                         end
175                         else begin
176                             read_cycle2 <= 1;
177                         end
178                     end
179               else
180                 readdata <= ram2_q;
181                 end
182
183                 3 : begin
184              if(ram3_rdaddress <= ram3_wraddress) begin
185                 if(read_cycle3) begin
```

```
186                     ram3_rdaddress <= ram3_rdaddress + 1;
187                     read_cycle3 <= 0;
188                     readdata <= ram3_q;
189                   end
190                   else begin
191                     read_cycle3 <= 1;
192                   end
193               end
194               else
195                   readdata <= ram3_q;
196           end
197                 default : readdata <= 255;
198             endcase
199         end
200     end
201 endmodule
```

**Hardware:oscheduler.sv**

```
1
2  //START_MODULE_NAME
       ------------------------------------------------------------------
3  //
4  // Module Name      :  Old Scheduler
5  //
6  // Description      :  Reads values from RAMS (4 x 4 architecture) and
       //                          schedules to prevent collisions.
7  //
8  // Limitation       :  None
9  //
10 // Results expected:  Schedules without collisions to appropriate RAM's
11 // //
12 //END_MODULE_NAME
       ------------------------------------------------------------------
13
14
15 module Scheduler(input logic clk,
16         input logic [31:0]  input1, input2, input3,
17         input logic [11:0]  input_ram_wr_add1, input_ram_wr_add2,
       input_ram_wr_add3,
18         input logic         write_enable,
19
20         output logic        out_ram_wr1, out_ram_wr2, out_ram_wr3,
21         output logic [31:0] output1, output2, output3,
22         output logic [11:0] input_ram_rd_add1, input_ram_rd_add2,
       input_ram_rd_add3,
23         output logic        input_ram_rden1, input_ram_rden2,
       input_ram_rden3);
24
25     logic empty1, empty2, empty3;
26     logic[1:0] write_cycle;
27
28     initial begin
29         write_cycle = 0;
30         output1 = 0; output2 = 0; output3 = 0;
31         out_ram_wr1 = 0; out_ram_wr2 = 0; out_ram_wr3 = 0;
32         input_ram_rd_add1 = 0; input_ram_rd_add2 = 0; input_ram_rd_add3
       = 0;
33     end
34
35     function logic set_rd(logic [31:0] data, logic empty);
36         if(!empty)
37             case(data[1:0])
38                 2'b00 : if(!out_ram_wr2) begin
39                     output2 = data;
40                     out_ram_wr2 = 1;
41                     return 1;
42                 end
43                 else
```

```
44                      return 0;
45                  2'b10 : if(!out_ram_wr2) begin
46                      output2 = data;
47                      out_ram_wr2 = 1;
48                      return 1;
49                  end
50                  else
51                      return 0;
52                  2'b01 : if(!out_ram_wr1) begin
53                      output1 = data;
54                      out_ram_wr1 = 1;
55                      return 1;
56                  end
57                  else
58                      return 0;
59                  2'b11 : if(!out_ram_wr3) begin
60                      output3 = data;
61                      out_ram_wr3 = 1;
62                      return 1;
63                  end
64                  else
65                      return 0;
66              endcase
67          else
68              return 0;
69      endfunction
70
71      always_ff @(posedge clk) begin
72          input_ram_rden1 = 1; input_ram_rden2 = 1; input_ram_rden3 = 1;
73          //all packets have been written to RAM
74          if(write_enable)begin
75              if(write_cycle == 2) begin
76                  write_cycle = 0;
77                  if(input_ram_rd_add1 < input_ram_wr_add1)
78                      empty1 = 0;
79                  else
80                      empty1 = 1;
81                  if(input_ram_rd_add2 < input_ram_wr_add2)
82                      empty2 = 0;
83                  else
84                      empty2 = 1;
85                  if(input_ram_rd_add3 < input_ram_wr_add3)
86                      empty3 = 0;
87                  else
88                      empty3 = 1;
89
90                  input_ram_rd_add1 = input_ram_rd_add1 + set_rd(input1,
    empty1);
91                  input_ram_rd_add2 = input_ram_rd_add2 + set_rd(input2,
    empty2);
92                  input_ram_rd_add3 = input_ram_rd_add3 + set_rd(input3,
    empty3);
```

```
93              end
94              else begin
95                  write_cycle = write_cycle + 1;
96                  out_ram_wr1 = 0; out_ram_wr2 = 0; out_ram_wr3 = 0;
97              end
98          end
99      end
100 endmodule
```

**Verilator:vgacounter.cpp**

```
// Instantiates the VGA_LED.sv and exercises it for 200 input and 200 read // cycles
#include "VVGA_LED.h"
#include "verilated.h"
#include "verilated_vcd_c.h"
#include <stdlib.h>
#include <time.h>
#include <iostream>
// This is required otherwise the module doesn't get instantiated and the linker
// throws an error.
vluint64_t main_time = 0;       // Current simulation time
        // This is a 64-bit integer to reduce wrap over issues and
        // allow modulus.  You can also use a double, if you wish.
        double sc_time_stamp () {       // Called by $time in Verilog
            return main_time;           // converts to double, to match
                                        // what SystemC does
        }
int main(int argc, char** argv)
{
    Verilated::commandArgs(argc, argv);
    time_t t;
    // init top verilog instance
    VVGA_LED* top = new VVGA_LED();
    // init trace dump
    Verilated::traceEverOn(true);
    VerilatedVcdC* tfp = new VerilatedVcdC;
    top->trace(tfp, 99);
    tfp->open("vgaled.vcd");
    // initialize simulation inputs
    top->clk    = 1;
    top->write = 0;
    top->reset =0;
    top->read = 0;
    int num_packets = 10;
    srand((unsigned) time(&t));
    // run simulation for 100 clock periods
    for(int i = 0; i < 300; i++)
    {
        if(i>=8 && i<8+8*num_packets){
            top->write=1;
            top->chipselect = 1;
            //top->address = 1;
            if(i%8==0)
                top->address = i/8%4;
            if(i%2 == 0 && i%8 < 6)
                top->writedata = rand()+1;
            else if(i%8 == 6)
                top->writedata = 0;
        }
```

```
        else if(i>=10+8*num_packets && i<12+8*num_packets && i%2==0){
                top->write=1;
                top->chipselect = 1;
                top->address =15;
                top->writedata = 0;
        }
        else if(i%2 == 0){
                top->write=0;
                top->chipselect = 0;
                top->address =0;
                top->writedata = 0;
        }

        for(int clk = 0; clk < 2; ++clk)
        {
            top->eval();
            tfp->dump((2 * i) + clk);
            if (clk==1){
                    top->clk =!top->clk;
            }
         }
    }

    int ram0_size = top->v__DOT__buffer__DOT__ram0_wraddress;
    int ram1_size = top->v__DOT__buffer__DOT__ram1_wraddress;
    int ram2_size = top->v__DOT__buffer__DOT__ram2_wraddress;
    int ram3_size = top->v__DOT__buffer__DOT__ram3_wraddress;
    int j = 0;


    for(int i = 300; i < 600; i++)
    {
        if(i < 312){
            top->chipselect = 1;
            top->address = 14;
            top->write = 1;
        }else if(j < ram0_size){
            top->write = 0;
            top->chipselect = 1;
            top->address = 0;
            if(i%6 < 4)
                top->read = 1;
            else
                top->read = 0;
        }
        else if(j >= ram0_size && j < ram1_size + ram0_size){
            top->write = 0;
            top->chipselect = 1;
            top->address = 1;
            if(i%6 < 4)
                top->read = 1;
            else
```

```
101                 top->read = 0;
102         }
103         else if(j >= ram1_size + ram0_size && j < ram1_size + ram2_size
    + ram0_size){
104             top->write = 0;
105             top->chipselect = 1;
106             top->address = 2;
107             if(i%6 < 4)
108                 top->read = 1;
109             else
110                 top->read = 0;
111         }
112         else if(j >= ram1_size + ram0_size + ram2_size && j < ram1_size
    + ram2_size + ram3_size + ram0_size){
113             top->write = 0;
114             top->chipselect = 1;
115             top->address = 3;
116             if(i%6 < 4)
117                 top->read = 1;
118             else
119                 top->read = 0;
120         }else if(i >= 590 && i<592){
121             top->write = 1;
122             top->chipselect = 1;
123             top->address = 13;
124             top->read = 0;
125         }else{
126             top->write = 0;
127             top->chipselect = 0;
128             top->address = 0;
129             top->read = 0;
130         }
131
132         if(i>312 && i%6==5)
133             j++;
134
135         for(int clk = 0; clk < 2; ++clk)
136         {
137             top->eval();
138             tfp->dump((2 * i) + clk);
139             if (clk==1){
140                     top->clk =!top->clk;
141             }
142          }
143     }
144     tfp->close();
145 }
```

**Verilator:schedulercounter.cpp**

```
//For easy interfacing with the scheduler.

#include "VScheduler.h"
#include "verilated.h"
#include "verilated_vcd_c.h"

int main(int argc, char** argv)
{
    Verilated::commandArgs(argc, argv);

    // init top verilog instance
    VScheduler* top = new VScheduler();

    // init trace dump
    Verilated::traceEverOn(true);
    VerilatedVcdC* tfp = new VerilatedVcdC;
    top->trace(tfp, 99);
    tfp->open("scheduler.vcd");
    // initialize simulation inputs
    top->clk = 1;
    top->write_enable = 1;
    top->reset_rams = 0;

    // run simulation for 100 clock periods
    for(int i = 0; i < 24; i++)
    {
        if (i==8){
            top->input_ram_wr_add[0][0] = 2;
            top->input_ram_wr_add[1][1] = 2;
            top->input_ram_wr_add[2][2] = 2;
            top->input_ram_wr_add[3][3] = 2;
        }

        if(top->input_ram_rd_add[0][0] == 0)
            top->inp[0][0] = 1;
        else
            top->inp[0][0] = 0;
        if(top->input_ram_rd_add[1][1] == 0)
            top->inp[1][1] = 2;
        else
            top->inp[1][1] = 0;
        if(top->input_ram_rd_add[2][2] == 0)
            top->inp[2][2] = 3;
        else
            top->inp[2][2] = 0;
        if(top->input_ram_rd_add[3][3] == 0)
            top->inp[3][3] = 4;
        else
            top->inp[3][3] = 0;

```

```
51          for(int clk = 0; clk < 2; ++clk)
52          {
53              top->eval();
54              tfp->dump((2 * i) + clk);
55              if (clk==1){
56                  top->clk =!top->clk;
57              }
58          }
59      }
60      for(int j = 0; j < 4; j++){
61          for(int k = 0; k < 4; k++){
62              top->input_ram_rd_add[j][k] = 0;
63          }
64      }
65      top->input_ram_wr_add[0][0] = 0;
66      top->input_ram_wr_add[1][1] = 0;
67      top->input_ram_wr_add[2][2] = 0;
68      top->input_ram_wr_add[3][3] = 0;
69      top->total_time = 0;
70
71      for(int i = 24; i < 96; i++)
72      {
73          if (i==32){
74              top->input_ram_wr_add[0][0] = 2;
75              top->input_ram_wr_add[1][0] = 2;
76              top->input_ram_wr_add[2][0] = 2;
77              top->input_ram_wr_add[3][0] = 2;
78          }
79          if(top->input_ram_rd_add[0][0] == 0)
80              top->inp[0][0] = 1;
81          else
82              top->inp[0][0] = 0;
83          if(top->input_ram_rd_add[1][0] == 0)
84              top->inp[1][0] = 2;
85          else
86              top->inp[1][0] = 0;
87          if(top->input_ram_rd_add[2][0] == 0)
88              top->inp[2][0] = 3;
89          else
90              top->inp[2][0] = 0;
91          if(top->input_ram_rd_add[3][0] == 0)
92              top->inp[3][0] = 4;
93          else
94              top->inp[3][0] = 0;
95
96
97          for(int clk = 0; clk < 2; ++clk)
98          {
99              top->eval();
100             tfp->dump((2 * i) + clk);
101             if (clk==1){
102                 top->clk =!top->clk;
```

```
103                     }
104                 }
105             }
106             tfp->close();
107         }
```

**Verilator:ramcounter.cpp**

```
//For easy interfacing with the Scheduler
#include "VRAM.h"
#include "verilated.h"
#include "verilated_vcd_c.h"
vluint64_t main_time = 0;       // Current simulation time
        // This is a 64-bit integer to reduce wrap over issues and
        // allow modulus.  You can also use a double, if you wish.
        double sc_time_stamp () {       // Called by $time in Verilog
            return main_time;           // converts to double, to match
                                        // what SystemC does
        }

int main(int argc, char** argv)
{
    Verilated::commandArgs(argc, argv);

    // init top verilog instance
    VRAM* top = new VRAM();

    // init trace dump
    Verilated::traceEverOn(true);
    VerilatedVcdC* tfp = new VerilatedVcdC;

    top->trace(tfp, 99);
    tfp->open("ram.vcd");

    // initialize simulation inputs
    top->clock   = 0;
     // run simulation for 100 clock periods
    for(int i = 0; i < 100; i++)
    {
            if (i>=13 && i<15){
                    top->data = 0xA;
                    top->wren = 0x1;
                    top->wraddress = 0x1;
            }
            else if (i>=15 && i<17){
                    top->data = 0xB;
                    top->wren = 0x1;
                    top->wraddress = 0x2;
            }
            else{
                    top->data = 0;
                    top->wren = 0;
            }

            if (i>=17 && i<19){
                    top->rden = 0x1;
                    top->rdaddress = 0x1;
            }
```

```
51              else if (i>=19 && i<21){
52                      top->rden = 0x1;
53                      top->rdaddress = 0x2;
54              }
55              else{
56                      top->rden = 0;
57              }
58
59
60          for(int clk = 0; clk < 2; ++clk){
61              top->eval();
62              tfp->dump((2 * i) + clk);
63              if (clk==1){
64                      top->clock =!top->clock;
65              }
66           }
67
68      }
69
70      tfp->close();
71  }
```

**Verilator:buffercounter.cpp**

```
// For simulating Buffer, its better to simulate the full suite
#include "VBuffer.h"
#include "verilated.h"
#include "verilated_vcd_c.h"
#include "iostream"
vluint64_t main_time = 0;       // Current simulation time
        // This is a 64-bit integer to reduce wrap over issues and
        // allow modulus.  You can also use a double, if you wish.
        double sc_time_stamp () {       // Called by $time in Verilog
            return main_time;           // converts to double, to match
                                        // what SystemC does
        }
int main(int argc, char** argv)
{
    Verilated::commandArgs(argc, argv);

    // init top verilog instance
    VBuffer* top = new VBuffer();

    // init trace dump
    Verilated::traceEverOn(true);
    VerilatedVcdC* tfp = new VerilatedVcdC;

    top->trace(tfp, 99);
    tfp->open("buffer.vcd");
    top->read_enable = 1;

    // initialize simulation inputs
    top->clk    = 1;
     // run simulation for 100 clock periods
    int add = 0;
    for(int i = 0; i < 100; i++){

        // Place a dummy data on write bus. You need to write first.
        // Write to RAM 1
        //RAM 0 & RAM 1
        if (i>=10 && i<14){
                top->out_ram_wr[0] = 1; //Enable ramen1 for 1 clock
   cycles
                top->outp[0] = 1; //Put data on the result signal

                top->out_ram_wr[1] = 1;
                top->outp[1] = 2;
        }
        else{
                top->out_ram_wr[0]=0; //Toggle ramen1
                top->outp[0] = 0; //Toggle result 1
        }
        //RAM 2 & RAM 3
        if (i>=14 && i<18){
```

```
50                  top->out_ram_wr[2] = 1;
51                  top->outp[2] = 3;
52
53                  top->out_ram_wr[3] = 1;
54                  top->outp[3] = 4;
55          }
56          else{
57                  top->out_ram_wr[2] = 0;
58                  top->outp[2] = 0;
59          }
60          // Generate read signals
61          if(i>=20 && i<36){
62                  top->chipselect = 1;
63                  top->read = 1;
64                  top->address = add;
65                  if(i%4 == 3)
66                      add = add + 1;
67                  printf("%i\n", add);
68          }
69          else{
70                  top->chipselect = 0;
71                  top->address = 0;
72                  top->read = 0;
73
74          }
75          for(int clk = 0; clk < 2; ++clk)
76          {
77              top->eval();
78              tfp->dump((2 * i) + clk);
79              if (clk==1){
80                      top->clk = !top->clk;
81              }
82           }
83      }
84      tfp->close();
85  }
```

**Verilator:Makefile**

```
# SwitchON hardware simulation file. Compiles all the modules individually or
# can compile them into one top module.

# List the includes here
# altera_mf.v contains scfifo and altsync modules.
INCLUDES=altera_mf.v
# List all the warning flags with the reason to skip them.

WFLAGS= -Wno-INITIALDLY -Wno-lint -Wno-MULTIDRIVEN -Wno-UNOPTFLAT -Wno-COMBDLY
#WFLAGS=
# Warning Flags Description(http://www.veripool.org/projects/verilator/wiki/
# Manual-verilator)
# 1)-Wno-INITIALDLY:-
# Warns that you have a delayed assignment inside of an initial or final
# block.If this message is suppressed, Verilator will convert this to a
# non-delayed assignment. See also the COMBDLY warning.Ignoring this
# warning may make Verilator simulations differ from other simulaors.
# Our Observation:
# ----------------
# Since some of the Altera modules (more than hundreds)did not have
# this explicitly set we disabled it, and have not faced any issue as such.
#
# 2)-Wno-lint:-
#   Disable all lint related warning messages, and all style warnings.
#   This is equivalent to "-Wno-ALWCOMBORDER -Wno-CASEINCOMPLETE
#   -Wno-CASEOVERLAP -Wno-CASEX -Wno-CASEWITHX -Wno-CMPCONST -Wno-ENDLABEL
#   -Wno-IMPLICIT -Wno-LITENDIAN -Wno-PINCONNECTEMPTY -Wno-PINMISSING
#   -Wno-SYNCASYNCNET -Wno-UNDRIVEN -Wno-UNSIGNED -Wno-UNUSED -Wno-WIDTH
#   "
#   plus the list shown for Wno-style.
# It is strongly recommended you cleanup your code rather than using this
# option, it is only intended to be use when running test-cases of code
# received from third parties.
#
# 3)-Wno-MULTIDRIVEN:-
#   Warns that the specified signal comes from multiple always blocks. This
#   is often unsupported by synthesis tools, and is considered bad style.
#   It will also cause longer runtimes due to reduced optimizations. Ignoring
#   this warning will only slow simulations, it will simulate correctly.
#
# 4)-Wno-UNOPTFLAT:-
```

```
41 #   Warns that due to some construct, optimization of the specified
       signal
42 #   or block is disabled. The construct should be cleaned up to improve
43 #   runtime.A less obvious case of this is when a module instantiates
44 #   two submodules. Inside submodule A, signal I is input and signal O
       is
45 #   output. Likewise in submodule B, signal O is an input and I is an
       output.
46 #   A loop exists and a UNOPT warning will result if AI & AO both come
       from
47 #   and go to combinatorial blocks in both submodules, even if they are
48 #   unrelated always blocks. This affects performance because Verilator
49 #   would have to evaluate each submodule multiple times to stabilize
       the
50 #   signals crossing between the modules.Ignoring this warning will only
51 #   slow simulations, it will simulate correctly.
52 # 5)-Wno-COMBDLY:-
53 #   Warns that you have a delayed assignment inside of a combinatorial
       block.
54 #   Using delayed assignments in this way is considered bad form, and
       may
55 #   lead to the simulator not matching synthesis. If this message is
56 #   suppressed, Verilator, like synthesis, will convert this to a
57 #   non-delayed assignment, which may result in logic races or other
       nasties
58 #   .See http://www.sunburst-design.com/papers/
       CummingsSNUG2000SJ_NBA_rev1_2.pdf
59 # Ignoring this warning may make Verilator simulations differ from other
60 # simulators.
61
62
63 TOPMODULE=VGA_LED # Name of the TOP MODULE into which all modules will
       be mushed.
64
65
66 # Define individual modules below with the appropriate simulators.
67 # Notation to define simulation file is <modulenamecounter.cpp>
68
69 # TOP level module depends on Fifo.v Scheduler.v Buffer.v megamux.v
70 VGA_LED_SIM=vgacounter.cpp # Define the simulation file you for this
       module.
71 vgaled:
72   verilator $(WFLAGS) -top-module $(TOPMODULE) -I $(INCLUDES) -cc \
73     -trace VGA_LED.sv --exe $(VGA_LED_SIM)
74   make -j -C obj_dir/ -f VVGA_LED.mk  VVGA_LED
75   obj_dir/VVGA_LED
76
77 # The RAM's on the output port of the Switch
78 buffer_SIM=buffercounter.cpp # Define the simulation file you for this
       module.
79 buffer:
80   verilator -Wno-lint -top-module Buffer -I $(INCLUDES) -cc \
```

```
81      --trace Buffer.sv --exe $(buffer_SIM)$
82    make -j -C obj_dir/ -f VBuffer.mk VBuffer
83    obj_dir/VBuffer
84
85  #Compiles the scheduler depends on None. This is the Crossbar switch
86  scheduler_SIM=schedulercounter.cpp
87  scheduler:
88    verilator -Wno-lint -cc --trace Scheduler.sv --exe $(scheduler_SIM)$
89    make -j -C obj_dir/ -f VScheduler.mk VScheduler
90    obj_dir/VScheduler
91
92  # Compiles into Altera's scfifo depends on scfifo.v
93  #fifo_SIM=fifocounter.cpp
94  #fifo:
95    #verilator -Wno-INITIALDLY -Wno-lint -Wno-MULTIDRIVEN --top-module
        Fifo \
96      #-cc --trace Fifo.v --exe $(fifo_SIM)
97    #make -j -C obj_dir/ -f VFifo.mk VFifo
98    #obj_dir/VFifo
99  ## Compiles the Megamuxes
100 #mux_SIM=muxcounter.cpp
101 #mux:
102   #verilator -Wno-lint -cc --trace lpm_mux.v --top-module lpm_mux --exe
        \
103     #$(mux_SIM)$
104   #make -j -C obj_dir/ -f Vlpm_mux.mk Vlpm_mux
105   #obj_dir/Vlpm_mux
106
107 #Compiles the scheduler depends on None. This is the Crossbar switch
108 ram:
109   verilator $(WFLAGS) -I $(INCLUDES)$  -cc --trace RAM.v -top-module RAM
        --exe ramcounter.cpp
110   make -j -C obj_dir/ -f VRAM.mk VRAM
111   obj_dir/VRAM
112
113 clean:
114   rm -rf obj_dir
115   rm -f  *.vcd
```

**Software:main.c**

```
/*
 * Userspace program that communicates with the led_vga device driver
 * primarily through ioctls
 * Based on Stephen Edwards's Code.
 * Specific Words(see packetgen.h) reserved for RAM/Scheduler control.
 * Architecuture of the Switch
        |Address|       |Status|
            15          write_enable // Kicks the Scheduler into motion.
            14          read_enable //  Kicks the output RAMS.
 */
#include <stdio.h>
#include <stdlib.h>
#include <time.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 "packetgen.h"

int vga_led_fd;
int sent[VGA_LED_DIGITS], received[VGA_LED_DIGITS];

int main()
{
    vga_led_arg_t vla;
    int i;
    time_t t; // Use the system time to seed the pseudo random generator
    srand((unsigned) time(&t));
    static const char filename[] = "/dev/vga_led";
    printf("Switch ON Packet Generator started\n");
    if ( (vga_led_fd = open(filename, O_RDWR)) == -1) {
        fprintf(stderr, "could not open %s\n", filename);
        return -1;
    }
    for(i=0; i<VGA_LED_DIGITS; i++){
        sent[i] = 0;
        received[i] = 0;
    }
    int* input;
    char* packet_info;
    // Generate the packet and sends it.
    for (i = 0 ; i < NUM_PACKETS; i++) {
        packet_info = mkpkt();
        input = generate(packet_info);
        int sport = i%4;
        printf("Sending packet to port: %u, of length: %u, with seed: %u
   \n", packet_info[0], packet_info[2], packet_info[1]);
```

```
50          write_segments(vga_led_fd, input, sport, packet_info[2]);
51          sent[packet_info[0]%4]++;
52      }
53      for(i=0; i<VGA_LED_DIGITS; i++){
54          printf("Packets sent to RAM %i: %i\n", i, sent[i]);
55      }
56      printf("Done Sending Packets, run validator to check!!,terminating\n
        ");
57      vla.digit = WRITE_ENABLE_SCHEDULER; // For starting the Scheduler
58      vla.segments = 0; // No address needed
59      if (ioctl(vga_led_fd, VGA_LED_WRITE_DIGIT, &vla)) {
60          perror("ioctl(VGA_LED_WRITE_DIGIT) failed");
61          return;
62      }
63      vla.digit = READ_ENABLE_SCHEDULER; // For Read Enabling the
        Scheduling
64      vla.segments = 0;
65      if (ioctl(vga_led_fd, VGA_LED_WRITE_DIGIT, &vla)) {
66          perror("ioctl(VGA_LED_WRITE_DIGIT) failed");
67          return;
68      }
69      return 0;
70  }
```

**Software:packetgen.h**

```
/*
 * packetgen headers:
 * Contains various headers defining packet parameters.
 *
 * Team SwitchON
 * Columbia University
 */
#include <stdint.h>
#ifndef __PACKETGEN_H__
#define __PACKETGEN_H__

/*Packet parameters */

//Crossbar Architecture
/*
        -1 -|-|-|
        -2 -|-|-|
        -3 -|-|-|
            1 2 3
*/
//Packet Structure (all length in bytes)
//       |LENGTH|LENGTH|SEED|DPORT|
//         1      1      1      1
// Destination port parameters
#define MIN_DPORT 1 // Minimum dst port that must be generated
#define DPORT_BITS 256 //1 Byte
#define NUM_PACKETS 150 // Total Packets to be sent.
#define SEED_BITS 256 // Keep the seed of 1 byte
#define WRITE_ENABLE_SCHEDULER 15 // Write Enable the scheduler.
#define READ_ENABLE_SCHEDULER 14 // Read Enable the Ouput Rams.
#define NUM_RAMS 4 // Define the number of RAMS.
#define TIME_PER_CYCLE 20*10^-9
char* mkpkt();
#endif
void write_segments(int vga_led_fd, int* input, int sport, int len);
int * generate();
```

**packetgen.c**

```c
/*
 * Userspace program that generates packets with random contents
 * Headers are defined in packetgen.h
 * Define the function prototypes in the packetgen.h headers
 */

#include <stdlib.h>
#include "packetgen.h"
#include "vga_led.h"
#include <stdio.h>

// Mkpkt returns a char pointer to the input. Generates an array with
// randonly generated packets.
char* mkpkt(){
        char* input = (char *) malloc(4);
        input[0] = rand()%DPORT_BITS;  // LSB 8 bits destination port.
        input[1] = rand()%SEED_BITS;  // Seed for the data.
        input[2] = rand()%60+4; // Length of the packet.
        input[3] = 0; // Length of packet  MSB
        return input;
}
// Writes the packet to vla.segment.
void write_segments(int vga_led_fd, int* packet, int sport, int len)
{
    vga_led_arg_t vla;
    int i;
    vla.digit = sport; // Make source port on which to send.
    for (i = 0 ; i < len; i++) {
        vla.segments = packet[i];
        if (ioctl(vga_led_fd, VGA_LED_WRITE_DIGIT, &vla)) {
            perror("ioctl(VGA_LED_WRITE_DIGIT) failed");
            return;
        }
    }
}
// Pushes the 32 bits and then generates a packet which is exactly 32
    bytes.
int* generate(char packet_info[4]){
    int i = 0;
    int len = (int) packet_info[2];
    int* input = (int *) malloc(len*4);
    input[0] = (packet_info[3]<<24)|(packet_info[2]<<16)|
                (packet_info[1]<<8)|(packet_info[0]);
    srand((unsigned)(packet_info[1]));
    for (i=1; i < len-1; i++){
        input[i] = rand() + 1;
    }
    input[len-1] = 0;
    return input;
}
```

**Software:validator.c**

```
/*
 * Switch ON validator,after main has completed sending the packets,
   this  * connects to the vga_led device and extracts all the
   information about t * he current status of the RAM's,based on which
   it extracts the packet fr * om each RAM. Now it also locally seeds
   itself with the encoded packet's * seed and then matches the
   information one by one till EOP((End of packe * t), at which stage
   it resets it's seed and waits for another packet.
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include <time.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 "packetgen.h"
int vga_led_fd;
int received[VGA_LED_DIGITS], packets[VGA_LED_DIGITS];

int main()
{
    vga_led_arg_t vla;
    int i,j,k,total_packets = 0,transferred_data=0;
    static const char filename[] = "/dev/vga_led";
    printf(" Userspace Validation of sent data \n");
    if ( (vga_led_fd = open(filename, O_RDWR)) == -1) {
        fprintf(stderr, "could not open %s\n", filename);
        return -1;
    }
    for(i=0; i<VGA_LED_DIGITS; i++){
        received[i] = 0;
        packets[i] = 0;
    }
    // Output Ram's count
    for(i=0; i<VGA_LED_DIGITS; i++){
        vla.digit = 12+i;
        if (ioctl(vga_led_fd, VGA_LED_READ_DIGIT, &vla)) {
            perror("ioctl(VGA_LED_READ_DIGIT) failed");
            return;
        }
        received[i] = vla.segments;
        printf("RAM %i (32 Bits Transferred,includes all 4) : %i\n", i,
   received[i]);
        transferred_data = transferred_data + received[i];
```

```
44      }
45      printf("Transferred Data (Bytes) : %i\n",transferred_data*4);
46      for(i=0; i<VGA_LED_DIGITS; i++){
47          vla.digit = 8+i;
48          if (ioctl(vga_led_fd, VGA_LED_READ_DIGIT, &vla)) {
49              perror("ioctl(VGA_LED_READ_DIGIT) failed");
50              return;
51          }
52      }
53      for(i = 0; i<VGA_LED_DIGITS; i++){
54          // Start extracting values from the Output Rams
55          printf("Validating from RAM: %i\n", i);
56          vla.digit = i;
57          for(j=0; j<received[i]; j++){
58              // Extract the values from the rams.
59              if (ioctl(vga_led_fd, VGA_LED_READ_DIGIT, &vla)) {
60                  perror("ioctl(VGA_LED_READ_DIGIT) failed");
61                  return;
62              }
63              if(vla.segments == 0){
64                  printf("Received 0");
65                  continue;
66              }
67              unsigned int seedMask = 65280; // Extract the middle bits
68              int length = vla.segments;
69              int seed = length;
70              int dport = seed;
71              length = length>>16;
72              seed = ((seed & seedMask)>>8); // Extracts the seed from
    packet
73              dport = dport%4;
74              if(dport!=i){
75                  printf("Invalid RAM location and dport from packet
    header\n");
76                  exit(1);
77              }
78              srand(seed);
79              // Do some error handling.
80              for(k=1; k<length; k++){
81                  if (ioctl(vga_led_fd, VGA_LED_READ_DIGIT, &vla)) {
82                      perror("ioctl(VGA_LED_READ_DIGIT) failed");
83                      return;
84                  }
85                  if(k<length-1){
86                      int a = rand() + 1;
87                      if(vla.segments != a){
88                          printf("Packet value does not match: %i, %i\n",
    a, vla.segments);
89                          exit(1);
90                      }
91                  } else if(k == length-1 && vla.segments != 0){
92                      printf("Length of packet reached but 0 not received
```

```
    .\n");
93                      exit(1);
94                  }
95                  j++;
96              }
97          packets[i]++;
98          total_packets++; // Increment the total packet sent counter.
99          }
100     }
101     printf("All RAM's have passed Validation!! \n");
102     printf("Total Packets Sent : %i\n", total_packets);
103     for(i = 0; i < 4; i++){
104         printf("Output RAM: %i Packet Count: %i\n", i, packets[i]);
105     }
106     vla.digit = 7;
107     if (ioctl(vga_led_fd, VGA_LED_READ_DIGIT, &vla)) {
108         perror("ioctl(VGA_LED_READ_DIGIT) failed");
109         return;
110     }
111     int num_clock_cycles = 0; // Number of clock cycles it took in total
112     num_clock_cycles = vla.segments;
113     printf("Number of cycles required for transfer: %i\n",
    num_clock_cycles);
114     float var = 20E-9; // Assuming FPGA runs on 50 MHZ clock.
115     printf("Speed through the Switch is:%f (in Mbits/s) \n",(
    transferred_data*4*8)/(var*1024*1024*num_clock_cycles));
116     return 0;
117 }
```

**Software:vga_led.h**

```
#ifndef _VGA_LED_H
#define _VGA_LED_H

#include <linux/ioctl.h>

#define VGA_LED_DIGITS 4

typedef struct {
  unsigned char digit;
  unsigned 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
```

**Software: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
    */
  u32 segments[VGA_LED_DIGITS];
} dev;

/*
 * Write segments of a single digit
```

```
50  * Assumes digit is in range and the device information has been set up
51  */
52 static void write_digit(unsigned int digit, u32 segments)
53 {
54   iowrite32(segments, dev.virtbase + 4*digit);
55   dev.segments[digit] = segments;
56 }
57
58 /*
59  * Handle ioctl() calls from userspace:
60  * Read or write the segments on single digits.
61  * Note extensive error checking of arguments
62  */
63 static long vga_led_ioctl(struct file *f, unsigned int cmd, unsigned
      long arg)
64 {
65     vga_led_arg_t vla;
66   switch (cmd) {
67   case VGA_LED_WRITE_DIGIT:
68     if (copy_from_user(&vla, (vga_led_arg_t *) arg,
69                sizeof(vga_led_arg_t)))
70       return -EACCES;
71     /*if (vla.digit > 8)*/
72       /*return -EINVAL;*/
73     write_digit(vla.digit, vla.segments);
74     break;
75
76   case VGA_LED_READ_DIGIT:
77     if (copy_from_user(&vla, (vga_led_arg_t *) arg,
78                sizeof(vga_led_arg_t)))
79       return -EACCES;
80     if (vla.digit > 15)
81       return -EINVAL;
82     int a;
83         a = ioread32(dev.virtbase + 4*vla.digit);
84         vla.segments = a;
85   //   vla.segments = dev.segments[vla.digit];
86     if (copy_to_user((vga_led_arg_t *) arg, &vla,
87           sizeof(vga_led_arg_t)))
88       return -EACCES;
89     break;
90
91   default:
92     return -EINVAL;
93   }
94
95   return 0;
96 }
97
98 /* The operations our device knows how to do */
99 static const struct file_operations vga_led_fops = {
100   .owner     = THIS_MODULE,
```

```
101   .unlocked_ioctl = vga_led_ioctl,
102 };
103
104 /* Information about our device for the "misc" framework -- like a char
      dev */
105 static struct miscdevice vga_led_misc_device = {
106   .minor     = MISC_DYNAMIC_MINOR,
107   .name    = DRIVER_NAME,
108   .fops    = &vga_led_fops,
109 };
110
111 /*
112  * Initialization code: get resources (registers) and display
113  * a welcome message
114  */
115 static int __init vga_led_probe(struct platform_device *pdev)
116 {
117 //  static unsigned char welcome_message[VGA_LED_DIGITS] = {
118 //    0x3E, 0x7D, 0x77, 0x08, 0x38, 0x79, 0x5E, 0x00};
119   int i, ret;
120   static unsigned char welcome_message[4] = {0, 0, 0, 0};
121
122   /* Register ourselves as a misc device: creates /dev/vga_led */
123   ret = misc_register(&vga_led_misc_device);
124
125   /* Get the address of our registers from the device tree */
126   ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
127   if (ret) {
128     ret = -ENOENT;
129     goto out_deregister;
130   }
131
132   /* Make sure we can use these registers */
133   if (request_mem_region(dev.res.start, resource_size(&dev.res),
134              DRIVER_NAME) == NULL) {
135     ret = -EBUSY;
136     goto out_deregister;
137   }
138
139   /* Arrange access to our registers */
140   dev.virtbase = of_iomap(pdev->dev.of_node, 0);
141     if (dev.virtbase == NULL) {
142     ret = -ENOMEM;
143     goto out_release_mem_region;
144   }
145
146   /* Display a welcome message */
147   for (i = 0; i < VGA_LED_DIGITS; i++)
148 //    write_digit(i, welcome_message[i]);
149   return 0;
150
151 out_release_mem_region:
```

```
152   release_mem_region(dev.res.start, resource_size(&dev.res));
153 out_deregister:
154   misc_deregister(&vga_led_misc_device);
155   return ret;
156 }
157
158 /* Clean-up code: release resources */
159 static int vga_led_remove(struct platform_device *pdev)
160 {
161   iounmap(dev.virtbase);
162   release_mem_region(dev.res.start, resource_size(&dev.res));
163   misc_deregister(&vga_led_misc_device);
164   return 0;
165 }
166
167 /* Which "compatible" string(s) to search for in the Device Tree */
168 #ifdef CONFIG_OF
169 static const struct of_device_id vga_led_of_match[] = {
170   { .compatible = "altr,vga_led" },
171   {},
172 };
173 MODULE_DEVICE_TABLE(of, vga_led_of_match);
174 #endif
175
176 /* Information for registering ourselves as a "platform" driver */
177 static struct platform_driver vga_led_driver = {
178   .driver = {
179     .name = DRIVER_NAME,
180     .owner  = THIS_MODULE,
181     .of_match_table = of_match_ptr(vga_led_of_match),
182   },
183   .remove = __exit_p(vga_led_remove),
184 };
185
186 /* Called when the module is loaded: set things up */
187 static int __init vga_led_init(void)
188 {
189   pr_info(DRIVER_NAME ": init\n");
190   return platform_driver_probe(&vga_led_driver, vga_led_probe);
191 }
192
193 /* Called when the module is unloaded: release resources */
194 static void __exit vga_led_exit(void)
195 {
196   platform_driver_unregister(&vga_led_driver);
197   pr_info(DRIVER_NAME ": exit\n");
198 }
199
200 module_init(vga_led_init);
201 module_exit(vga_led_exit);
202
203 MODULE_LICENSE("GPL");
```

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

**Software:Makefile**

```
# Use gcc for compilation
CC = gcc

# Include extra directories
INCLUDES =

# Compilation Options:
# -g for debugging -Wall enables all warnings
CFLAGS = -g -Wall $(INCLUDES)

# Linking Oprions:
# -g for debugging info
LDFLAGS = -g

#List of Libraries which need to be linked in LDLIBS
LDLIBS =

# Specify Targets in a recursive way.
# We rely on make's implicit rules:
#   $(CC) $(LDFLGAGS) <all-dependent-.o-files> $(LDLIBS)

# Main is the main target that is compiled,it contains references to other
# functions.

# The philosophy is pretty simple main depends on everything so include all the
# *.o files in main,now other files might have internal dependencies like packet_gen
# depends on common hence it is compiled together.
#
.PHONY:
main: main.o packetgen.o common.o

main.o: main.c packetgen.h

packet_gen.o: packetgen.c packetgen.h common.h

common.o:common.c common.h

# Target based compilation

packetgen:
  $(CC) $(CFLAGS) packetgen.c packetgen.h common.c common.h

.PHONY: clean
clean:
  rm -f *.o a.out main core packet_gen common executable
.PHONY: all

```

```
48 all: clean packet_gen
```