WUSTL CSE 400 Independent Study

Professor: Dr. Roger Chamberlain

Neil E. Olson

Date: 7/23/2018

Table of Contents

I. Introduction

A common problem with hardware implementations of various algorithms is that the design may require many layers of combinational logic in-between registers. This is especially true when a variety of arithmetic operations must be completed in a single clock cycle. As a result, the clock rate must be dramatically reduced to accommodate the gate delays. This may slow down the arrival of data from other hardware blocks and therefore decrease overall throughput. Thus, one slow hardware module can jeopardize the performance of the entire system.

A remedy to this issue is to place intermittent clocked registers in the combinational logic path to shorten the signal's propagation time. This way, an algorithm could be executed at a significantly higher clock rate. This technique is known as "c-slowing" the pathway. With this solution, we place 'C' registers in the pathway and decrease the original shortest-path propagation time T_p to an average of $\frac{T_p}{C}$. C-slowing requires slightly more hardware (additional registers), but it eliminates the original issue almost entirely.

An additional benefit of c-slowing a combinational logic path is that it creates a pipeline capable of performing different computations at each stage. If there are many sources of data rather than just one, the c-slowed pipeline can act as a server for many jobs at once while only consisting of a single pathway. In this way, a c-slowed pipeline is "virtualized" logic. This distinction means that the single pipeline is doing the same work (albeit slower) as many pipelines, each connected to one of the data sources. This saves significant hardware resources. For a single c-slowed pipeline, say there are N data sources. Only one job can enter the pipeline at once, so a queue is required for each data source. Figure 1 shows a c-slowed pipeline with queues and a feedback path. As jobs arrive in the queues, the hardware must decide which queue is going to be serviced. The hardware block that arbitrates which job enters the pipeline is known as a "scheduler". Depending on the arrival rate of the jobs, the "fullness" of the queues, and the priority of completing different jobs, it may be beneficial to have different scheduler configurations. Some primary goals of the scheduler are to keep the pipeline active at all times and make sure all the queues are being serviced.

The goal of this independent study project is to create new scheduler designs for c-slowed algorithms that have $N > C$ and require feedback of the output for the next computation. These two properties together create the requirement that the pipeline output (feedback) be stored in memory. This is known as "coarse grained" context switching. Applications of this type greatly increase the potential of a sophisticated scheduler design. This work is an addition to the paper "Modeling and Simulation of Virtualized Logic Computations using an M/G/1 Queueing Model with Vacations" by Dr. Roger Chamberlain and Dr. Michael Hall. To assist with the paper, I created queues, a c-slowed pipeline, and various scheduler designs in HDL. I tested each design in Vivado simulations. The benefits of simulating the queue hardware from HDL are two-fold.

Theoretical queuing models can be reliably validated and the efficacy of different scheduler designs can be tested. These benefits will be expounded upon later in this report.

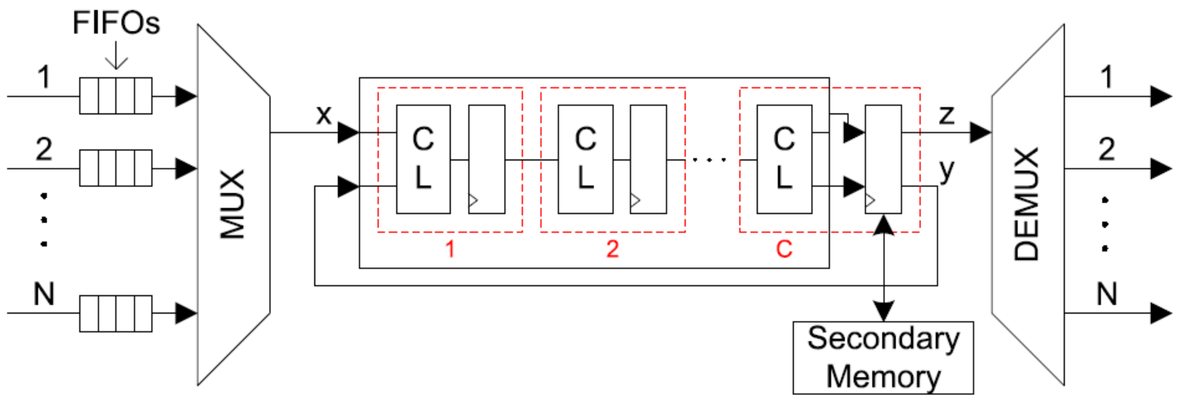


Figure 1: An example of a virtualized pipeline with N inputs and C pipeline stages. A path for output feedback is present, which in this case is necessary for completing the next job. Credit to Dr. Chamberlain and Dr. Hall for creating this graphic.

II. Project Scope

Among the deliverables for this project were HDL models of a c-slowed pipeline with feedback and queuing, multiple scheduler designs, max clock rate information for the schedulers, and testing data showing mean queue occupancy for the schedulers. At the project's completion, 24 different HDL modules were created to be used as test benches or implemented designs. The complexity of the modules completed range from simple D-type flip flops to full queue structures and schedulers. Each module required original logic design, description in VHDL, and testing for functionality. During testing, fixing design flaws was a frequent occurrence that sometimes took hours to execute.

Once designs achieved basic functionality, performance data was required to validate theoretical queuing models created by Dr. Chamberlain and Dr. Hall. A strong indicator of scheduler performance is the "mean queue occupancy" of the system. Mean queue occupancy is the average number of jobs in the queues over a certain period of time with a given load factor (tied to arrival rate). The lower the mean queue occupancy, the better job a scheduler is doing at servicing the jobs arriving in the queues. Indirectly, this information also provides the average latency for a job to exit the pipeline. As a stand-alone statistic, MQO does a good job displaying the effectiveness of different queuing models. Several test bench modules were created specifically for collecting this data. Data on queue throughput for high load factors was also directly collected. It turned out that the high load factor data was not needed for the paper.

An important piece of information for the paper was the max clock rate each scheduler could operate at. This data is crucial for determining whether the scheduler will be a bottleneck for the system by having the lowest maximum clock rate (thus slowing down the pipeline). As a section of his dissertation, Dr. Hall investigated the theoretical performance of c-slowed pipelines applied to common algorithms such as SHA-256, AES encryption, and synthetic cosine generation. For each algorithm, he determined the maximum possible clock rate for the Xilinx Virtex-7 family of devices. If the different scheduler designs are implemented with these algorithms, comparing their timing performance indicates whether the scheduler will hamper overall system timing.

III. <u>Design of Testing Model</u>

In order to test the scheduler designs, a basic pipeline and queue model was created. Since the sought-after information about the scheduler is general (instead of for a specific algorithm), the pipeline and queuing models were dramatically simplified. The most time-efficient solution to designing the pipeline was to have it be constructed with four D-type flip flops. Since the data in this system carries no information other than whether it is a valid job, I made data with a value of ‘1’ correspond to a valid job and ‘0’ correspond to an invalid one. This simplicity made collecting data on queue occupancy and throughput much easier. The pipeline and input queue have dimensions of C = 4 and N = 8 respectively. This setup is one that was being explored in the paper, so I built the system to match. Each of the N input queues was eight elements deep. This was sufficient to prevent data overflows in the queue.

Figure 2 shows all of the hardware modules in the system and how data flows between them. A brief description is given in the graphic as to what each module is doing, but I will elaborate here as well. Starting from the beginning of the data path, a test bench module, “data feeder”, selects which queue a valid job is going into. This module allows for the arrival distribution (among queues) and load factor to be tweaked for different tests. Inside the module, a ROM filled with queue selections (generated by MATLAB) is read each clock cycle, and the data is then passed on to the input queues.

When a single queue (of the N) is selected by the data feeder, a ‘1’ is inserted into the forward-most slot of that queue. The queue selection from the data feeder is an 8-bit, one-hot value (with the one selecting the queue). For each queue slot, there is an associated flag bit. The flag is a symbol that the corresponding queue slot is filled with a valid job. The flags are absolutely essential for letting the scheduler know which queues have jobs, how many jobs are waiting, and if there is an overflow of the queue. The flags are also sent to the “occupancy counter” module, which keeps track of mean queue occupancy. When a job is sent to the pipeline, the last valid flag in the chosen queue is cleared and all of the jobs shift one spot closer to being sent out to the pipeline. One challenge with designing the input queues is coordinating how flags are reset when the queue is written to, read from, or both at the same time. Details of my implementation can be seen in the associated HDL module. An 8-input multiplexer selects which queue outputs its job to the pipeline. The scheduler controls that multiplexer with its "selection" signal.

Data from the input and feedback queue selected by the scheduler are put into the pipeline as a valid job (if both feedback and input have a value of one). The job is then passed along through the four stages to the output. The feedback queue works largely the same way as the input queue. However, the feedback queue is written to by the pipeline output. The scheduler's selection for a given job is saved in a companion pipeline consisting of only of a shift register. When the data reaches the output, the feedback queue knows where to place the valid output result because the scheduler's earlier selection had been saved.

Outputs also go to the "result counter" module, which is a test bench that keeps track of basic performance statistics such as number of valid jobs completed and how many clock cycles has the system been through. Because of the general nature of this queue and pipeline model, all of the unique schedulers can be simulated without changing the underlying model. Despite the simplicity of some of the non-scheduler hardware, designing and testing all the components still took a large amount of time.

IV. Scheduler Designs

a. Round Robin – Before Dr. Chamberlain and I had the idea of creating sophisticated schedulers for c-slowed pipelines with queued inputs, he and Dr. Hall had been using a "round robin" scheduler for all of their theoretical models. The round robin scheduler shifts the selection up one (with looping at queue N back to one) each clock cycle. The round robin chooses empty queues fairly frequently with a low load factor or non-uniform queue selection, but it does a decent job at making sure each queue is given a chance to send its jobs to the pipeline. This is one of the most simplistic schedulers imaginable that prevents rapid queue overflow.

Its implementation was also very simple. For my N = 8 queue structure, I made an 8-element shift register with the ends connected. The queue selection is initialized to one, and the actual output signal is the 'Q' signal from each flip flop. The shift register was initialized to "10000000". The round robin scheduler is used as a baseline from which to judge the performance of the other scheduler designs.

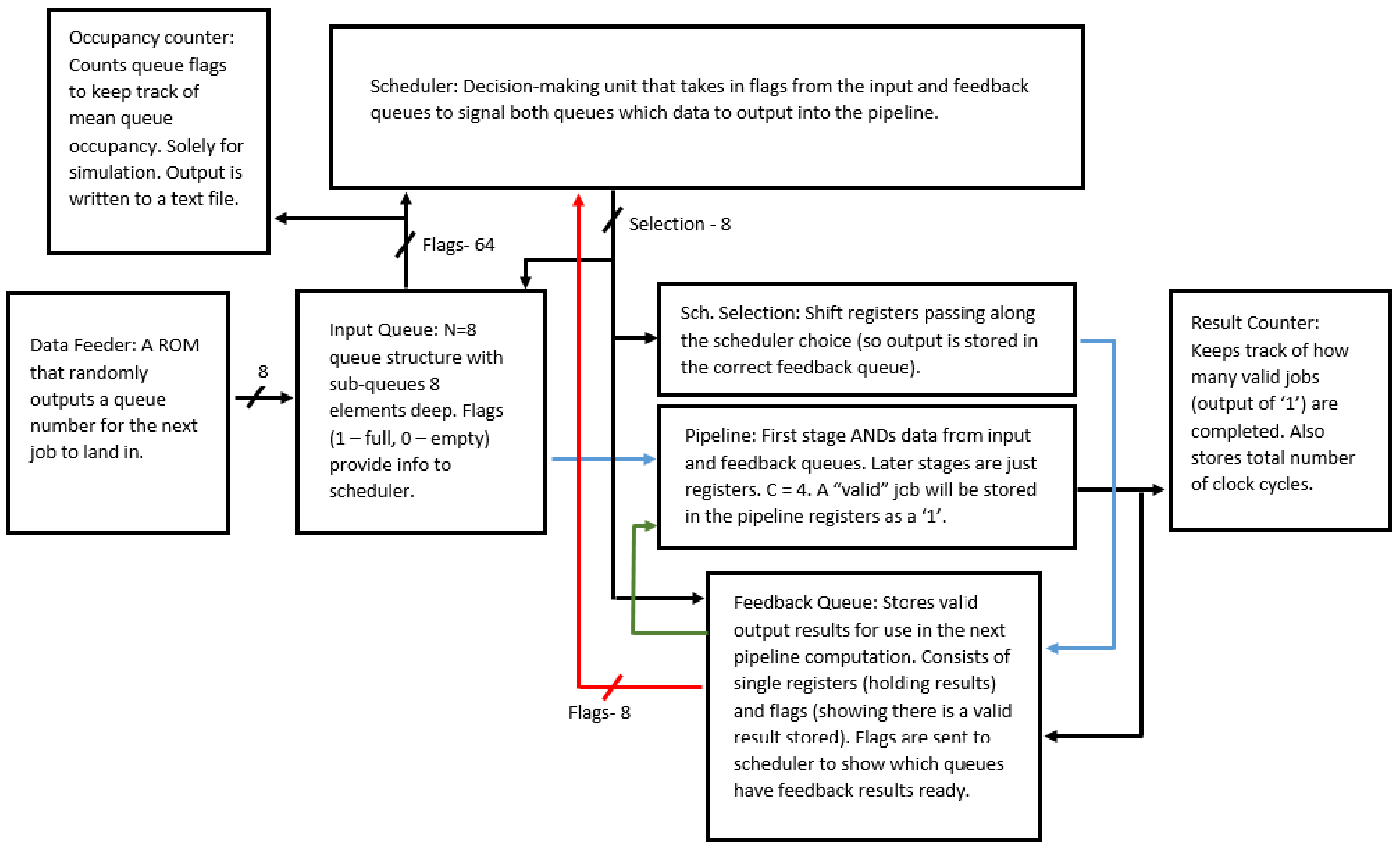

Occupancy counter: Counts queue flags to keep track of mean queue occupancy. Solely for simulation. Output is written to a text file.
Scheduler: Decision-making unit that takes in flags from the input and feedback queues to signal both queues which data to output into the pipeline.
Flags- 64
Selection - 8
Data Feeder: A ROM that randomly outputs a queue number for the next job to land in.
8
Input Queue: N=8 queue structure with sub-queues 8 elements deep. Flags (1 – full, 0 – empty) provide info to scheduler.
Sch. Selection: Shift registers passing along the scheduler choice (so output is stored in the correct feedback queue).
Pipeline: First stage ANDs data from input and feedback queues. Later stages are just registers. C = 4. A "valid" job will be stored in the pipeline registers as a '1'.
Result Counter: Keeps track of how many valid jobs (output of '1') are completed. Also stores total number of clock cycles.
Feedback Queue: Stores valid output results for use in the next pipeline computation. Consists of single registers (holding results) and flags (showing there is a valid result stored). Flags are sent to scheduler to show which queues have feedback results ready.
Flags- 8

Figure 2: A flow chart of the system components with simple descriptions.

b. **Round robin skip one** – A somewhat more efficient version of the round robin with the same general idea is a scheduler that skips one queue selection if either the input or feedback of the next selection is empty. With this scheme, if the scheduler's last choice is queue number one and the next queue doesn't have a job ready, the scheduler's next choice would be number three, effectively skipping one spot. This scheme is simple in concept, but the logic design is slightly tricky. The decision-making of which queue to select is governed by the current scheduler state, the flags from the first element of each input queue, and the flags of the feedback queues. Figure 3 shows the logic for half of the scheduler design (with the other half being the same).

The passing of the selection signal (the one) is done by multiplexing between two registers. There are a few ways of thinking about the decision making, but the easiest is to trace where the one goes from its source. Say the last selection was queue four. That means that the '1' is residing in register U1. There are two places that '1' could go, register U2 or U3, which correspond to selecting queues one and two respectively.

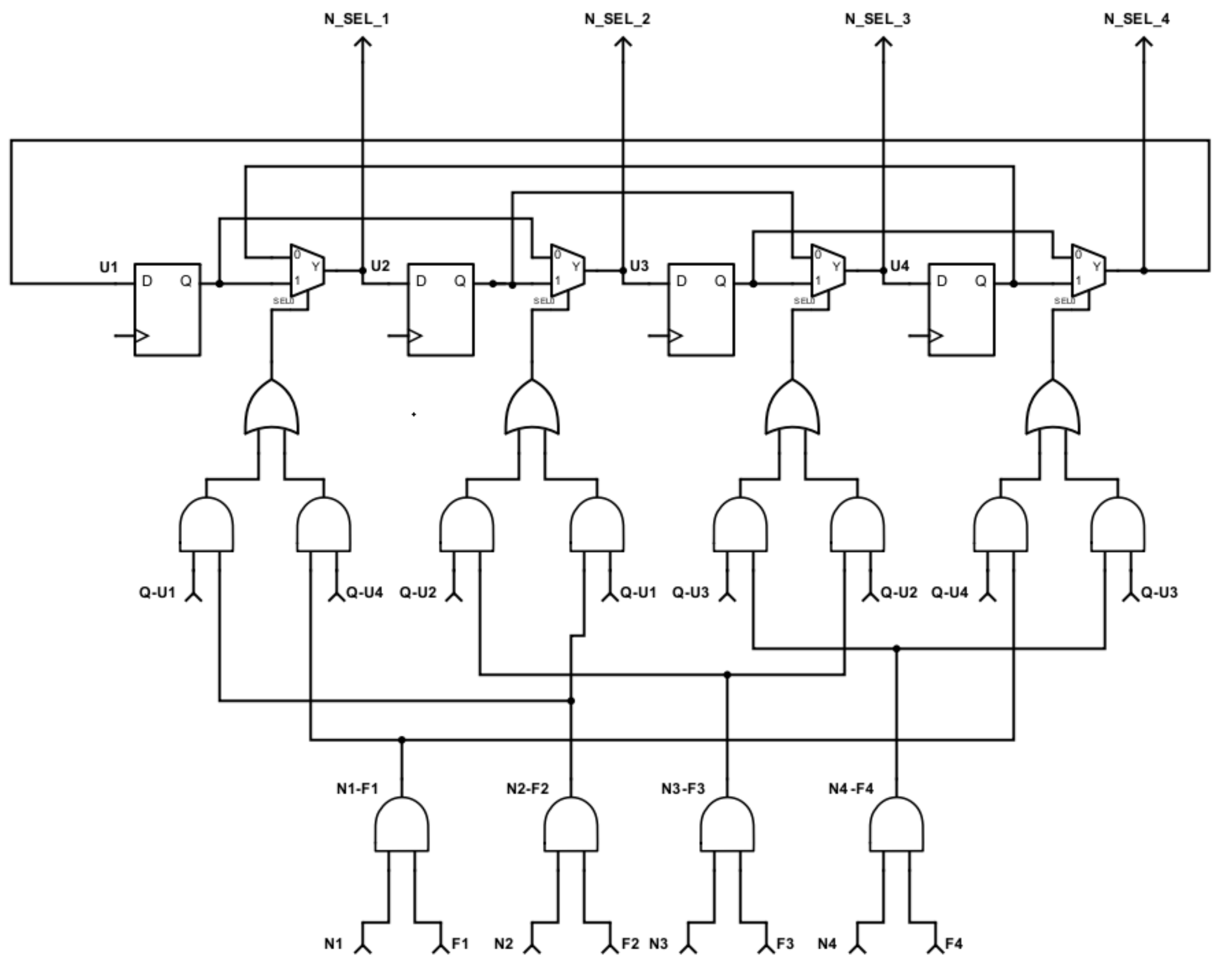


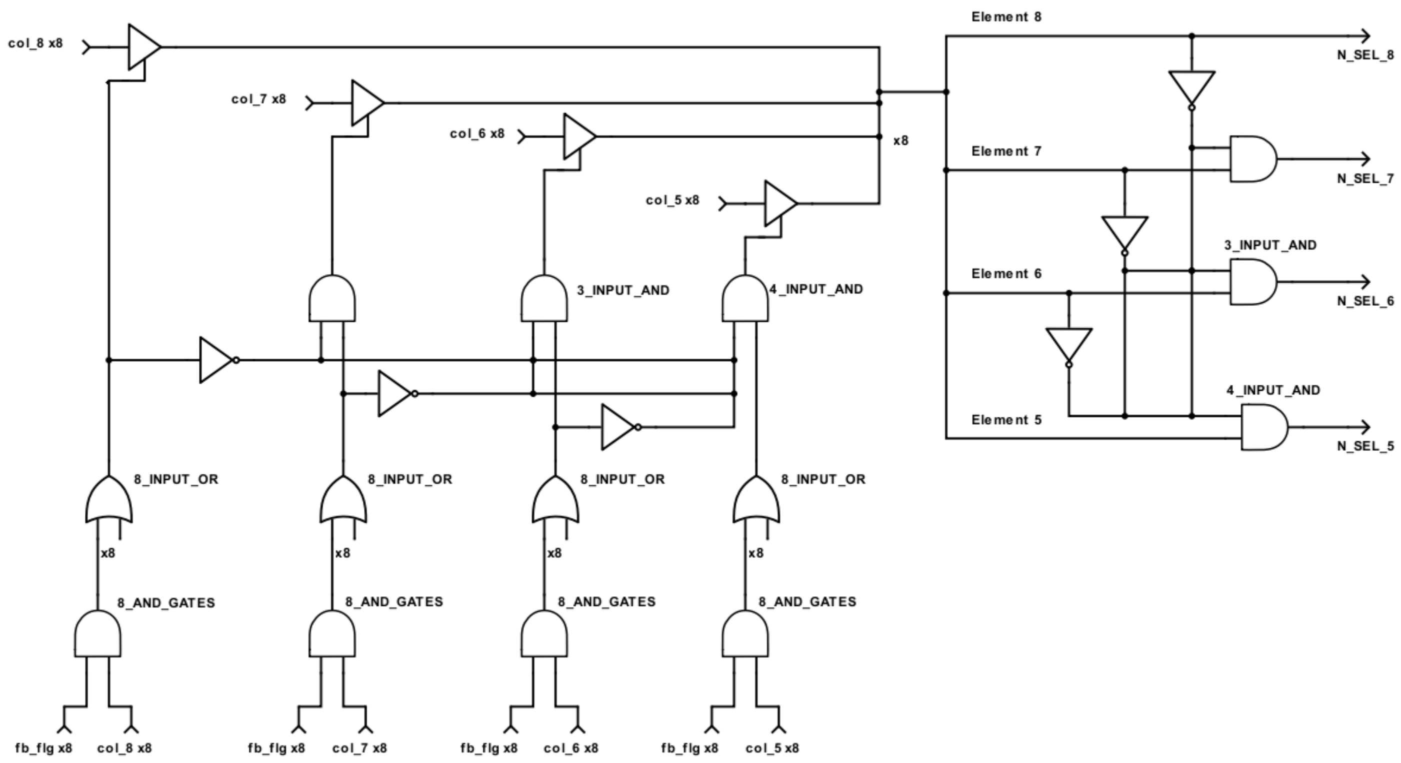
Figure 3: Round robin skip one scheduler. The scheduler selection signal is the output from the multiplexers in-between registers, which decide between the last register and the one two back (equivalent of skipping one). The label "Q-U#" is the Q output of a given register. "N#" is the first flag from the given input queue number. "F#" is the feedback flag for a given queue.

By default, the scheduler will skip a spot if the queue one isn't ready to output a job. This allows the scheduler to miss fewer jobs during operation. Therefore, a test of the flags needs to be done to see if queue one is ready (takes place in the first logic level). If it's not, then the '1' will pass to U3. The multiplexers in-between registers are set up so that a control signal of '0' will accept the data from two registers back. This being the default, the three combinational logic levels just need to test for whether conditions are correct for passing the data forward one spot. In this scheme, the '1' is always preserved, no matter where it's being passed.

c. **Capacity prioritization scheduler** – The third scheduler design is fundamentally different than the two round-robin-based designs. This scheduler, which is also referred to as the "most-full" scheduler in the HDL modules, chooses the queue with the greatest number of elements filled with valid jobs. Theoretically, if a job exists in any of the input queues, the scheduler will not provide an invalid job to the pipeline (given the feedback is ready too). It also does a terrific job of preventing overflows. The tradeoff is that this design increases in hardware utilization when more elements are added to each individual input queue. That is because it gets all of the flags from all input queues, which in this case is a total of 64 flags (8 queues 8 elements long). In summary, this design has the potential to be the most efficient, but it is also the most resource hungry.

When imagining the way this scheduler works, it helps to look at the input queues as a grid like they are presented in Figure 1. Each row represents a different queue, and each column, M, as the M^{th} element in each queue. To successfully select a queue, the scheduler needs to pinpoint how many elements the most-full queue has and which queue it is. Figure 4 shows the logic diagram of half of the capacity prioritization scheduler. Take a look at the first level of logic. What it's doing is making sure the determination for how full the most-full queue is only conducted amongst queues with valid feedback. If the feedback is invalid, selecting a queue that is the fullest would lead to an invalid output to the pipeline, which is unacceptable. The OR gates in the second level determine which columns have valid elements. For example, if the far left OR gate produced a '1', then we would know that the 8th column has at least one valid element. Based on the output from the OR gates, a priority encoder selects the highest numbered column with valid elements.

Up to this point, how full the fullest queue is has been determined. Now, the scheduler must select a queue. A second priority encoder gets the whole column in which the highest element resides. Based on the bit order of the column, we can see which queues have the highest elements. In order to give consistent results, the priority encoder gives priority to higher-numbered queues first. Due to the nature of this scheduler, if two or more queues are tied for being fullest, they will all get serviced in the next few clock cycles granted that none of them grow during that time. In this way, the scheduler is actually "fair" in its treatment of the queues. Given that this scheduler is implemented entirely in deep combinational logic, the timing of this design is practically guaranteed to be worse than the other two. That being said, its increased performance may be worth it.



Firgure 4: The capacity prioritization scheduler (most-full). Due to limitations with the schematic drawing tool, gates that appear as two inputs may actually have more. See the labels by the gates. The label "fb_flg" includes all eight feedback flags (one for each queue). Each "col" label is for the corresponding input queue column.

V. <u>Testing Results</u>

As mentioned previously, the most important test results are the mean queue occupancy (MQO) and the longest path timing for each scheduler. To measure MQO, I generated ten unique data sets in MATLAB. Each of those data sets consisted of 1000 elements. The target load factor for the test was 0.5, meaning that one job would come every other clock cycle. I did this deterministically in the data sets by giving all even elements a value of zero. In a real system, the probability of a given element being non-zero would be P = 0.5 rather than having a deterministic ordering like I chose to do. I learned of this after doing the data collection and chose not to redo the experiment due to time constraints.

Each data set was run through all the scheduler designs, resulting in a total of 30 test runs. The tests were conducted in ISim, a hardware simulation tool, in the Xilinx Vivado application. During the test runs, the "occupancy counter" module counted the number of flags active and wrote the results to a text file. In an effort to make the results more accurate, only the last 80% (the steady state) of the data was used. Those results were mean-averaged for each test. To compute the MQO of a given scheduler, the ten tests for that scheduler were averaged. This is known as the technique of batch means. Results of the calculations are shown in Figure 5.

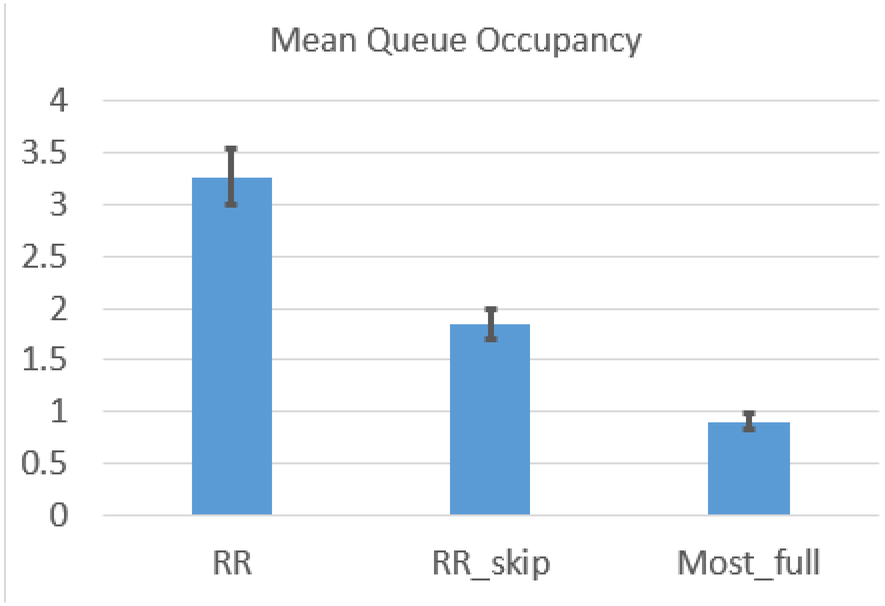


Figure 5: Mean queue occupancy for the different scheduler designs for a load factor of 0.5. The whiskers show one standard deviation. Credit to Dr. Chamberlain for creating this graph.

The results of the test validate the theoretical MQO found by Dr. Hall. He computed a MQO of 3.5 for the round robin scheduler with the same queue and pipeline dimensions. The results from the test are within one standard deviation, which is an acceptable error. Dr. Chamberlain ran a student's T-test on the data, and the null hypothesis was rejected with p-values below 0.01. This indicates that the data is statistically valid for the sake of checking the queue model. Looking at the graph of the test data, it is apparent that the round robin scheduler does a poorer job emptying the queues than the other two designs. This means that the queues will back up more and the overall latency for getting a job through the pipeline is greater. This was expected given the round robin's static behavior. The round robin skipping one performs somewhat better due to the fact that it can get through the queues faster if there are large gaps between the valid queue selections. That is a common occurrence with a low load factor. The capacity prioritization scheduler performs the best, which is not surprising. If there is a valid job in any one of the queues, it will be serviced. That's why you see a MQO of below one for that scheduler.

For assessing timing performance, each of the scheduler designs were synthesized and implemented with Xilinx Vivado. The target device was a Virtex-7 FPGA. Particularly, the device used was the xc7v585ttfg1157-3. Notice that this is a speed grade 3 part. As stated earlier, the point of determining the maximum clock rate for the different scheduling schemes is to see whether they will have a lower max than the algorithm implementations they're trying to speed up. Fortunately, none of the schedulers' maximums dropped below the highest for the algorithm implementations. The maximum clock rates were 1.38 GHz, 884.9 MHz, and 172.1 MHz for the round robin, round robin skip one, and the most-full scheduler respectively. Of the three algorithms that Dr. Hall had made implementations of, SHA-256 had the highest maximum at 168.8 MHz. This result is pleasantly surprising given the long combinational logic pathway for the most-full scheduler.

VI. Conclusion

This independent study was a worthwhile learning experience in that it was an opportunity to practice logic design, coding in HDL, and using the Vivado tools. All of these are skills I will almost certainly be using professionally. Personally, I thought the challenge of designing the logic for the schedulers was quite fun. It was an added bonus that I was able to provide some useful results to add to Dr. Chamberlain and Dr. Hall's current body of work. Using a scheduler to optimize queue latency has many potential uses outside of the algorithms mentioned in this paper. Going forward, it could be rewarding to find new algorithms for the scheduler to be used on or develop new scheduler designs. Either of these would be fitting projects for undergraduate research in the future.

Appendix A: List of HDL Modules

All entries are in alphabetical order.

Format: name.filetype – description

bb_pipeline.vhd – “Black box” pipeline consisting of 4 registers.

c_slow_top – An early top module that was later adapted to be unique for each scheduler.

Constraints.xdc – A constraints file for implementing both the round robin type schedulers.

Constraints_MF.xdc – A constraint file for the most-full scheduler with actual pin assignments on the Virtex-7. Using the false path constraints file is preferable.

Constraints_MF_false_paths.xdc – Constraints for the most-full scheduler using false paths for every signal.

d_ff_en.vhd – A simple d-type flip flop with a reset signal. Resets to a value of ‘0’. Also has enable. Enables were critical for the input queue design.

data_feeder.vhd – The ROM-based module assigning jobs to different queues.

fb_ff.vhd – A d-type flip flow that resets to ‘1’. This was specifically for initializing the feedback queue flags high so pipeline computations can begin.

fb_queue.vhd – The main feedback queue module.

fb_reg_en.vhd – An early piece of code that was storing pipeline results in the feedback queue as a 2-bit number.

in_queue.vhd – An early iteration of the code for the input queue. Deprecated by a version which sends all of the queue flags to the scheduler.

in_queue_all_flgs.vhd – The final version of the input queue.

most_full_sch.vhd – The HDL module containing the most-full scheduler.

most_full_sch_for_imp.vhd – A version of the most-full scheduler for implementation. This version includes a clocked flip flop necessary for getting a worst negative slack value.

n_eight_q.vhd – An early sub-module of the input queue. It was much faster to make the input queue in multiple component modules rather one big one. This module is now deprecated by a version that sends all its flags to the scheduler.

n_eight_q_all_flgs.vhd – The final sub-module for the input queue.

occupancy_count.vhd – A test bench that counts all of the input queue flags and writes it to a text file.

priority_scheduler.vhd – A failed scheduler design that gave precedence to lower numbered input queues.

reg.vhd – A basic 2-bit register without an enable signal.

reg_8.vhd – An 8-bit register for the companion pipeline keeping track of the past scheduler decisions.

reg_en.vhd – A basic 2-bit register with enable.

result_counter.vhd – A module collecting output data and counting valid jobs completed.

rr_scheduler.vhd – The round robin scheduler module.

rr_skip_one.vhd – The round robin skip one module.

top_most_full_sch.vhd – The top module for the most-full scheduler. Contains the pipeline, queues, and other system modules as well.

top_rr_scheduler.vhd – The top module for the round robin scheduler. Contains the pipeline, queues, and other system modules as well.

top_rr_skip_one.vhd - The top module for the round robin skip one scheduler. Contains the pipeline, queues, and other system modules as well.