Final Project Report First Page. Must match this format (Title) – attach other pages as needed

Project Responsibilities (module names, other assigned tasks):

200113166. Tristan Gibson

Delay (ns to run provided example).

Clock period: 6.28

cycles”: 66

Delay = 6.28*66 = 414.48

Area: (um^2)

Logic: 15208.55

1/(delay.area) (ns^{-1}.um^{-2})

1/(414.48*15208.55) = 158e-9

Delay (TA provided example. TA to complete)

1/(delay.area) (TA)

Abstract

This report characterizes the design and synthesis of a hardware implementation of the merge sort algorithm. The register transfer language design in captured in Verilog and synthesized with Synopsys. The requirements of the project were to use Register File holding 32 8-bit unsigned numbers that are to be merge Sorted, and written to Register File. There are no constraints on the Register design (including number of read ports), except that Register File counts towards area. In this design, I describe a parallel solution that use 4 stages of FIFOs to merge the data.   The logic in each stage operates in parallel and the FIFOs full flags are used as the trigger for the next stage. This design sorts 32 values in 66 clock cycles. This is a synthesizable design that occupies 15208.55 um^2.   The Verilog modules are parameterized to allow easy instantiation which allows the core modules to be adaptable to any size list.

Merge Sort Algorithm Implemented in HDL

Tristan Gibson

1. Abstract:

The merge sort algorithm is a popular method for sorting lists of numbers. This algorithm has been researched by computer science academics for use on traditional computers. The fundamentals of the merge sort are best exploited when the processing is done in parallel. Because of this, traditional computers don't see a performance improvement. The parallelism of ASICs makes this a good algorithm for a HDL implementation. In this report, the design of using multiple stages of FIFOs operating in parallel to implement the merge sort is discussed.

2. Introduction:

The merge sort algorithm is a divide and conquer algorithm. The list of numbers if broken done to single element list, and then each list is merged together until the full list has been reassembled in a sorted result. In this implementation, there are 32 8-bit unsigned numbers that are unsorted in a register file. There are 32 read buses for this register file that are connected to a simple 2 element sorter. This sorter pushes the contents to a FIFO. There are 4 stages of FIFO that are cascaded together so that each subsequent stage doubles the depth of the previous (ie, the 4th stage has 2 FIFOs with a depth of 16). There is a controller between each stage that compares two FIFOs. The controller looks at the top of the FIFO, and outputs the lowest value until the FIFO is empty. The controller is outputting the values into the next FIFO stage. Since each stage is running in parallel, the list can be sorted in 66 clock cycles. In the following sections, the details of the design are presented including the architecture, interface specifications, performance and verification method.

3. Architecture:

A high level diagram of the architecture can be seen in Figure 1. There 4 main aspects to the design: Register File, FIFOs, sort_2_values, and mergeFIFO.

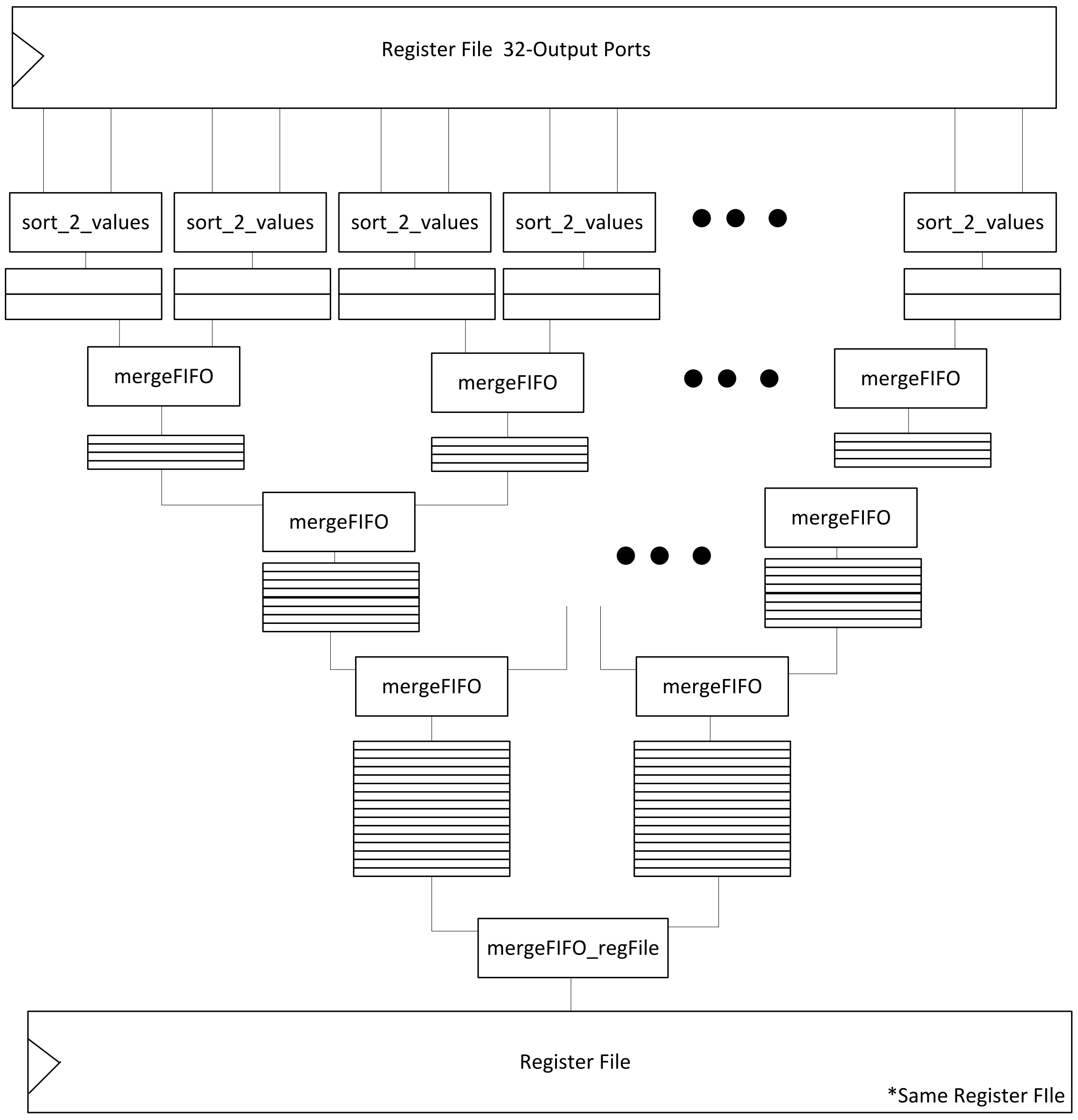

Figure 1 Parallel Merge Sort Using FIFOs

3.1. Register File

The Register File is a bank of flip-flops that 32 Read Addresses and 1 Write Address. The Register File is preload with data from the test bench. At the last stage of the merge sorting, values in the register file are overwritten with the sorted elements.

3.2.FIFO

The FIFO used is a DesignWare IP from Synopsys. The FIFO is a synchronous (single clock) FIFO with Static flags (DW_fifo_s1_sf). This FIFO has fully registered synchronous outputs and uses D flip-flops that parameterize the width and depth. The empty and full flags are used to control the flow of data in/out of the FIFOs. The controls are active low.

3.3.Sort 2 Values

This module receives 2 values (8 bit) and outputs them sequentially in ascending order. The next stage is a FIFO, so there is also a control signal to "push" into the FIFO. A simple FSM determines the output order (and handles reset). I decided to go with extra states to make handling transitioning to the S0 state so that the Push signal is not asserted after the sort. Figure 2 show the FSM diagram.

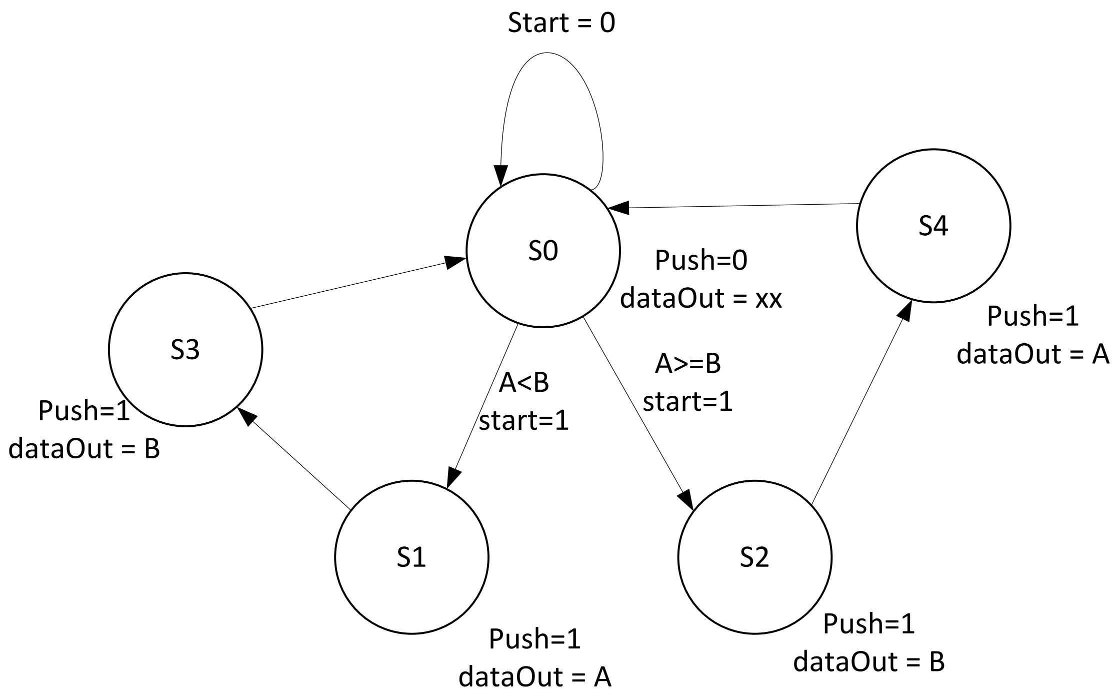

Figure 2 FSM diagram for sort_2_values

3.4.Merge FIFO

This module looks at the top of two FIFOs and outputs the lowest value to next FIFO. In every stage, the FIFO is already sorted, so I am able to merge the FIFOs. The module is parametrized so that only one module is coded, but can control each stage of FIFOs (2,4,8,16,32). My design is to use a combination of FSM and counter. I use the FSM to keep the module Idle while waiting for the prior to fill up. It is important for this module to only start after both FIFOs are full. Once the control signals from both FIFOs are asserted, a counter will increment until the depth of the next FIFO. The module will compare the current FIFOs output, and only push the lower value. When a lower value is pushed, a pop request is sent to the prior FIFO to get the next value. The counter will use the "pop" signal as the incrementer. There is some priority logic that needs to handle the case when one FIFO is empty and the remaining values come from the other FIFO. Figure 3 shows the FSM diagram and counter used to control the state. Figure 4 shows how this module compares two FIFO outputs and determines which value to push into the next FIFO based on the control flags.

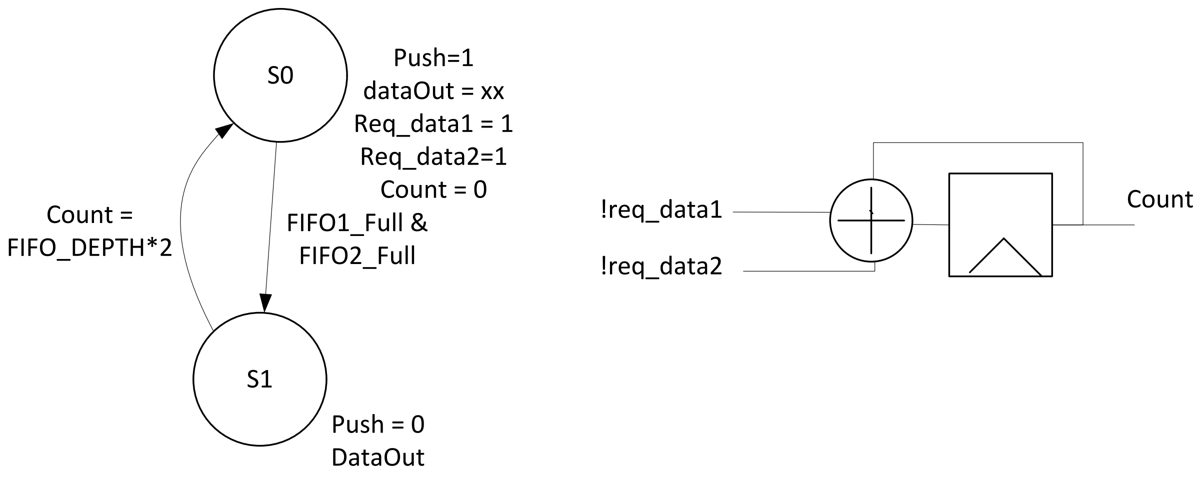

Figure 3 FSM diagram for MergeFIFO

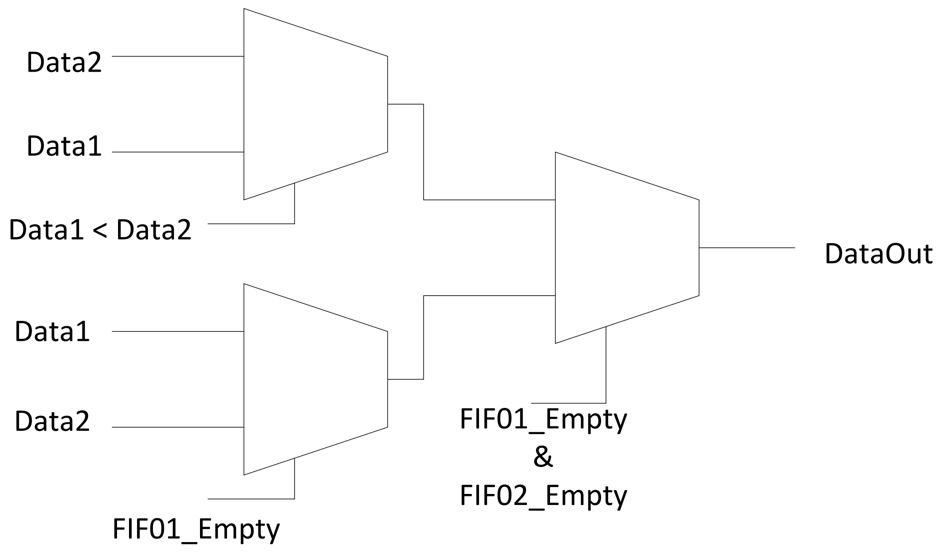

Figure 4 Comparison Logic for fifoMerge

3.5.Merge FIFO into Register File

This module is the same basic design as the Merge FIFO, with the exception of outputting into the Register File instead of a FIFO. The main difference with interfacing with the Register File is providing a Write Address for the data to be written into. I am already using a counter to control the flow of the data, and I know that this is the final Merge of the data. That allows me to repurpose the counter value as the Write Address in to the Register File. I also change the "push" control signal to be active High for Write Enable.

4. Interface Specifications

This section defines the interfaces for each module used in the design.

4.1. Top

Table 1 Interfaces for Top

Name	Width (bits)	Pin Direction	Description
Clock	1	Input	Clock
Start	1	Input	Control signal to identify start of sort
Reset	1	Input	Global reset

ReadBus0 – ReadBus31	8	Internal	Read Lines of Register 32 connecting Register File to first stage of sorting.
WriteAddress	5	Internal	Write Address for the Register File to determine where to write value to
WriteBus	8	Internal	Write Bus for the Register file that contains data that is written to Register File when WE is high, at the address specified in WriteAddress
WE	1	Internal	Write Enable for Register File
Done	1	Output	Flag indicating that the Merge Sort is complete.

4.2. Sort_2_values

Table 2 Interfaces for sort_2_values

Name	**Width (bits)**	**Pin Direction**	**Description**
Clock	1	Input	Clock
Start	1	Input	Control signal to identify start of sort
Reset	1	Input	Global reset
Data_In_A	8	Input	First unsigned number
Data_In_B	8	Input	Second unsigned number
Data_out	8	Output	The sorted order of the two inputs outputted on rising edge of clock for 2 cycles
push	5	Output	Control signal that is active low for when sorted values are being outputted (2 cycles)

4.3. fifoMerge

Table 3 Interfaces for fifoMerge

Name	**Width (bits)**	**Pin Direction**	**Description**
Clock	1	Input	Clock
Reset	1	Input	Global reset
Data_In1	8	Input	First unsigned number connected to a FIFOs output
Data_In2	8	Input	Second unsigned number connected to a FIFOs output
FIFO1_full	1	Input	Flag from prior FIFO1 indicating its full and ready for accessing
FIFO2_full	1	Input	Flag from prior FIFO2 indicating its full and ready for accessing

FIFO1_empty	1	Input	Flag from prior FIFO1 indicating its empty
FIFO2_empty	1	Input	Flag from prior FIFO2 indicating its empty
dataOut	8	Output	Unsigned value that is outputs on the rising edge of clock the lowest value of the Data_in1 and Data_In2.
push	1	Output	Active low control signal that is enabled when valid data is on the dataOut bus and should be written to the next FIFO.

4.4. fifoMerge_regFile

Table 4 Interfaces for fifoMerge_regFile

Name	**Width (bits)**	**Pin Direction**	**Description**
Clock	1	Input	Clock
Reset	1	Input	Global reset
Data_In1	8	Input	First unsigned number connected to a FIFOs output
Data_In2	8	Input	Second unsigned number connected to a FIFOs output
FIFO1_full	1	Input	Flag from prior FIFO1 indicating its full and ready for accessing
FIFO2_full	1	Input	Flag from prior FIFO2 indicating its full and ready for accessing
FIFO1_empty	1	Input	Flag from prior FIFO1 indicating its empty
FIFO2_empty	1	Input	Flag from prior FIFO2 indicating its empty
dataOut	8	Output	Unsigned value that is outputs on the rising edge of clock the lowest value of the Data_in1 and Data_In2.
Push_dataOut	1	Output	Active high control signal that is enabled when valid data is on the dataOut bus and should be written to the Register File.
count	1	Output	The address of where the dataOut should be written in the Register File

4.5. DW_fifo_s1_sf

Table 5 Interfaces for DW_fifo_s1_sf			
Name	**Width (bits)**	**Pin Direction**	**Description**
Clk	1	Input	Clock
Rst_n	1	Input	Reset input, active low
Push_req_n	1	Input	FIFO push request, active low
Pop_req_n	1	Input	FIFO pop request, active low
Diag_n	1	Input	Diagnostic control, active low
Data_in	8(parameter)	Input	FIFO data to push
empty	1	Outpu	FIFO empty output, active high
Almost_empty	1	Output	FIFO almost empty output, active high
Half_full	1	Output	FIFO half full output, active high
Almost_full	1	Output	FIFO almost full output, active high
full	1	Output	FIFO full output, active high
error	1	Output	FIFO error output, active high
Data_out	8(parameter)	Output	FIFO data to pop

5. Verification

The design was verified in multiple stages of simulation and synthesis. First, each module has a corresponding test bench that simulates it and verifies the functionality. The test bench evolved with the coding of the module so that each module could be tested in smaller pieces instead of one large block. This design has a simple solution, which is repeated many times in each stage. Because the building blocks are replicated so much, it allowed testing and verification to be straightforward. The main building block is the fifoMerge module. This module is parameterized for the depth of the FIFOs it interacts with. Therefore, I could isolate my unit test to the simple case of merging two FIFOs into one, and have pretty high confidence that it would translate well for the overall design.

With every major milestone in simulating a module, I performed quick synthesis runs to verify that the hardware was buildable. This reduced the risk of not having a complete design that had major synthesis issues. And since the design is based on some core blocks, this did not slow down development too much.

I initially prototyped the design in python. The python implementation used the standard computer science algorithm, expect I saved each stage in a different vector to enable testing. I compared each stage of the merge sort in HDL to Python to isolate bugs. Once the code got more stable, I transitioned to using SystemVerilog after learning about its features in lecture.

The last (and highest level) of simulation testing used the instructor provided data files. These files were read into the Register File and into a test vector. In the test vector, I used the SystemVerilog sort() function to sort the values. After the mergeSort is complete, I read in the outputted file and compared the mergeSort version to the sorted test vector.

6. Results

The performance of the merge sort algorithm is typically O(nlogn). The means the number of calculations would take n*log2(N). For this problem that would 160 calculations. My simulation runs show that this algorithm takes 66 clock cycles to perform. The start time is determined to be after the Register File is initially loaded with the test data. I used a “start” control flag to start the processing, which I call time 0. After the merge sort is complete and the original values are sorted in the Register File, my design outputs a “done” signal for clock period.

This design is synthesizable and contains few minor warnings from Synopsys. The outputted warnings are LINT-1 and a high fan-out. Both of these were evaluated and deemed minor to the successful synthesis of the design. This achieves a clock period of 6.28ns with Slack available. The total cell area is 15208.55 um^2.

7. Conclusion

This report has presented a complete synthesizable hardware implementation of the Merge Sort algorithm. The fundamental approach to this design was to cascade a series of FIFOs that were merged in 4 stages. The design used parameterized modules for both the control and actual FIFOs. This made the code simple, and adaptable to large problems. The simulation runs show that a list of 32 unsigned numbers can be sorted in 66 clock cycles. The synthesis runs with Synopsys show that design can be built into hardware and achieves a 6.28ns clock period with an area of 15208.55 um^2.

8. Appendix

This table below lists the files included with report and their description.

penultimate*.dat and sorted*.dat	Simulation outputs from the list*.dat files
simulationRuns.txt	Simulation Text output from simulations runs
simulationOutput.bmp	Simulation Waveform example (just one run)
Top.v	top module
fifoMerge.v	fifo Merge module
fifoMerg_regFile.v	fifo Merge into Reg File module
Sort_2values.v	sort 2 values module
Regfile.v	register File module
DW_fifo_s1_sf.v	DW FIFO module for simulation
DW_fifoctl_s1_sf.v	DW FIFO module for simulation
DW_ram_r_w_s_dff.v	DW FIFO module for simulation
all_modules.v	all modules (except DW) in one file
tb_Top.v	test bench for top
tb_fifoMerge.v	test bench for fifoMerge
tb_Sort_2values.v	test bench for sort_2_Values
tb_Regfile.v	test bench for regFile
evaluationsheet.xlsx	Evaluation sheet for TA
mergeSort.py	Python Implementation of merge Sort for testing
read.tcl	Read Scipt
Constraints.tcl	Contraints script
ComplileAnalyze.tcl	Compile and Analyize Script
read.output	Results of the read.tcl script
constrainsts.output	Results of the Constraints.tcl script
compile.output	Results of the ComplileAnalyze.tcl
command.log	Synopsys File
cell_report_final.rpt	Outputted from CompleAnalyze.tcl
timing_max_slow.rpt	Outputted from CompleAnalyze.tcl
timing_max_slow_holdfixed_tut1.rpt	Outputted from CompleAnalyze.tcl
timing_min_fast_holdcheck_tut1.rpt	Outputted from CompleAnalyze.tcl
top_final.v	Outputted from CompleAnalyze.tcl