Hardware Accelerator for Bellman-Ford Algorithm

Name: Parth Bhogate NCSU ID: 200108628

Modules designed in the project: datapath.v, fsm.v, top_without_mem.v, top_with_mem.v and test_bench.v

Delay (ns, to run input_sercret1.mem).

Clock period: 8.05 ns

# cycles: 6789953

Area: 3694.47 um^2

Logic: 3694.47 um^2

Memory: N/A

1/(delay.area)

= 4.95×10^{-12} $ns^{-1}.um^{-2}$

Delay (TA provided example. TA to complete)

1/(delay.area) (TA)

Abstract:

The project aims at designing a hardware accelerator for implementing the Bellman-Ford Algorithm. The Bellman-Ford algorithm is an algorithm to compute the shortest path from one vertex in a given graph to any other vertex. In contrast to the Djikstra's algorithm which is a greedy algorithm, Bellman-Ford works on the principle of relaxing each vertex of the graph repeatedly until the shortest distance to each vertex from the source vertex has been obtained. Bellman-Ford can also be used on graphs with negative edge weights (unlike Djikstra's algorithm).

In this project, hardware is designed to solve for the shortest path in a given graph with a maximum of 255 total vertices. The design aims to achieve a reasonably fast implementation of the algorithm with least possible chip area. Architectural principles such as pipelining and prefetching have been used to have minimum possible delay in processing.

The design is interfaced with four SRAM memories in order to read the input graph data, do the computation on the data and finally store the shortest path result. All the memories are assumed to have a delay of 4 ns, with synchronous write and asynchronous read functionality.

The design is described in Verilog HDL and simulated using ModelSim. The design is synthesized with Synopsys Design Vision to obtain an optimal clock period and area. The performance is measured using a benchmark workload and the results are documented.

A Project

On

Hardware Accelerator for Bellman-Ford Algorithm

As a Part of

ECE 520: Digital ASIC Design

Dr. Paul Franzon

North Carolina State University

By:

Parth Bhogate

NCSU ID: 200108628

11th November, 2015

Table of Contents

Abstract

The project aims at designing a hardware accelerator for implementing the Bellman-Ford Algorithm. The Bellman-Ford algorithm is an algorithm to compute the shortest path from one vertex in a given graph to any other vertex. In contrast to the Djikstra's algorithm which is a greedy algorithm, Bellman-Ford works on the principle of relaxing each vertex of the graph repeatedly until the shortest distance to each vertex from the source vertex has been obtained. Bellman-Ford can also be used on graphs with negative edge weights (unlike Djikstra's algorithm).

In this project, hardware is designed to solve for the shortest path in a given graph with a maximum of 255 total vertices. The design aims to achieve a reasonably fast implementation of the algorithm with least possible chip area. Architectural principles such as pipelining and prefetching have been used to have minimum possible delay in processing.

The design is interfaced with four SRAM memories in order to read the input graph data, do the computation on the data and finally store the shortest path result. All the memories are assumed to have a delay of 4 ns, with synchronous write and asynchronous read functionality.

The design is described in Verilog HDL and simulated using ModelSim. Further, the design is synthesized with Synopsys Design Vision to obtain an optimal clock period and area. The performance is measured using a benchmark workload and the results are documented.

Introduction

Bellman-Ford algorithm is an algorithm used to solve for the shortest path from a source vertex to all other vertices in the graph. Although conceptually similar to the Djikstra's algorithm, Bellman-Ford has the ability to calculate shortest paths in a graph with negative edge weights as well. The algorithm has usage in myriad applications such as network routing algorithms, big-data cluster algorithms.etc. It is critical to make the execution of such routing algorithms fast, since the delay in processing the data is reflected real-time to the users. Thus, special ASIC's to speedup these critical computations are usually employed in network routing architectures.

In this project, a hardware accelerator is designed to implement the Bellman-Ford algorithm. The design assumes that the graph data is stored in a specific format in a SRAM interfaced to the design. Also, source and destination vertex node pairs are stored in a separate SRAM. While processing the data, the design uses a SRAM to store intermediate values. Finally, the output for a source-destination pair is written to an output SRAM. The distance from source to destination as well as the shortest path is written in the output SRAM. Finally, if a negative weight cycle exists in the given graph, the algorithm exits as soon as it detects the negative weight cycle. An error value denoting the presence of a negative weight cycle is also written into the output SRAM.

The hardware design of the algorithm solver is done in Verilog HDL. Mentor Graphics ModelSim is used to simulate the design. The hardware was synthesized using Synopsys Design Vision to obtain optimal timing and area. Also, various design rule checks were carried out to ensure correctness of the synthesized hardware.

The special architectural features to obtain a fast execution time are as follows:

1. **Pipelining of daughter vertex fetch and computation:** The SRAM's connected to the design specify a delay of 4 ns for reading as well as writing the data. Since registers are used to read the data from SRAM, stall clock cycles are used up while waiting for the data to arrive. To hide this latency, the computation for a particular daughter vertex and the fetch for the next daughter vertex is done in parallel. As a result, the addition and comparison unit ("the computation unit") is not kept idle while the distance for the next daughter vertex is being fetched from Work SRAM.

2. **Prefetching of next parent vertex data from Graph SRAM:** After doing the computation for the daughters of a particular vertex, fetching the daughter vertex data for the next node in Graph SRAM incurs an extra delay of 2 clock cycles if not done in parallel. To avoid this extra delay in processing, the data for the next vertex (or the daughter data on the next line for the same vertex, if the daughter data continues onto multiple lines) is prefetched while the computation for the last daughter of the previous vertex is being done. This prefetch mechanism avoids any stall cycles for the computation unit.

These architectural features and the benefit obtained by implementing these features are explained in greater detail later in the report. To begin with, the report looks at the high-level architectural details used in hardware implementation. The algorithmic approach used to solve the problem in hardware is described. This includes a high-level view of the micro-operations involved in solving the problem.

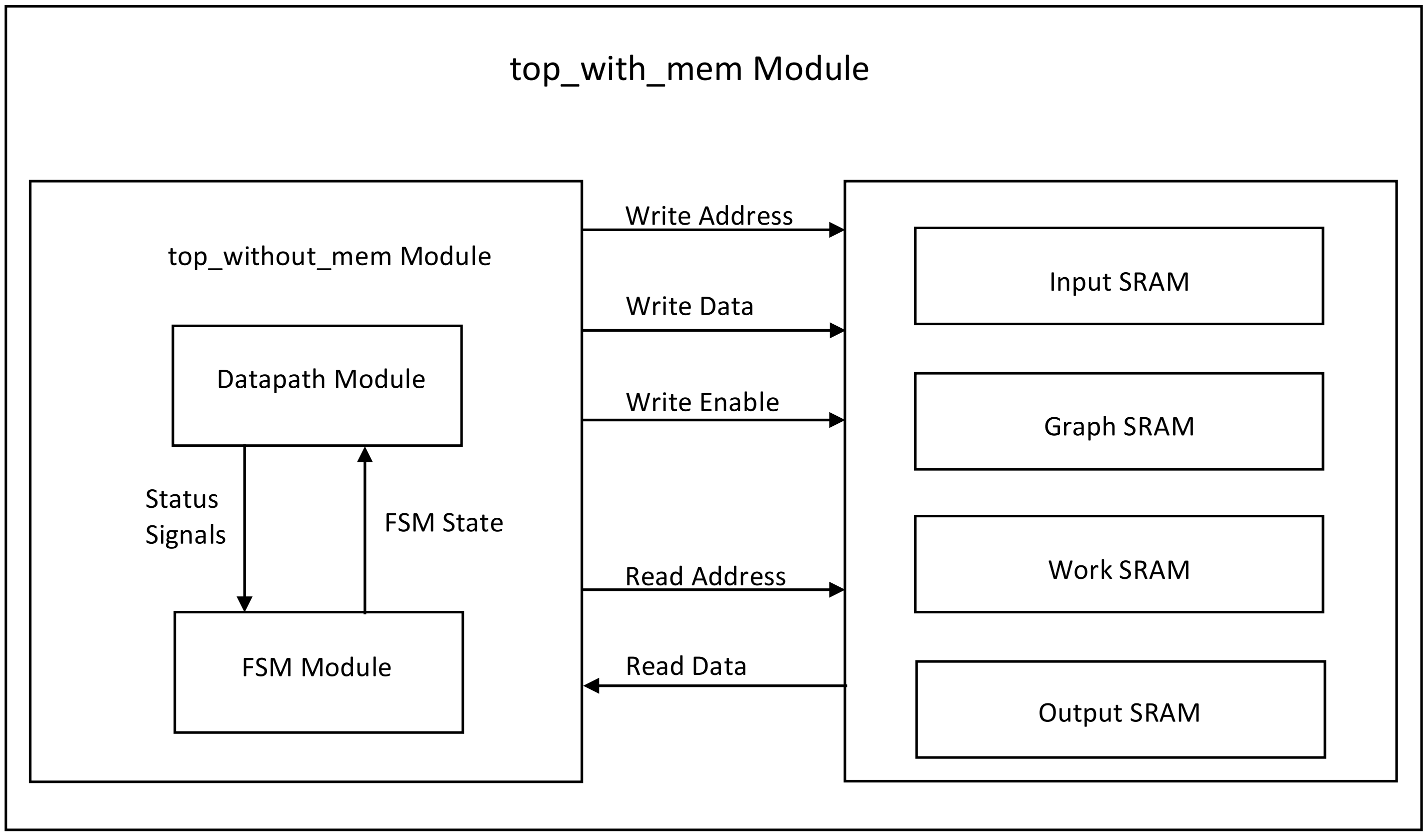
As was mentioned previously, the design is intended to be interfaced with four SRAMs for taking the data input regarding the Graph structure, the source-destination pair to be processed, the working memory to store intermediate values while doing the computation and to store the calculated shortest path and the corresponding shortest distance. Details about the memory interface including the bus widths and expected timing delays are presented in detail.

Further, the dataflow in the design is presented with a supporting hardware block diagram. In this section, a brief description of the sequence of steps used as well as the detail of the cycle timing is given. Low level circuit implementation details regarding the control strategy and the Finite State Machine controller are explained in detail. Also, various registers used in the design and their specific functionality are explained.

Finally, the results obtained on running various test cases are presented. The area obtained for the design after synthesis using Synopsys Design Vision is given. An overall performance metric taking into account both the execution latency and the chip area is calculated.

High – Level Architecture

The hardware is organized into a datapath, a finite state machine controller, and then the memories. The diagram below shows the interfacing between the various modules:



1. **Datapath module:** This module has the main dataflow hardware to execute the algorithm. The addition and comparison circuitry is implemented in the datapath module, along with counters and status flags to be sent to the controller to control the execution flow of the program. This module is also interfaced with SRAM memories to read and write data after computation.

2. **FSM module:** This is the controller module which moves the program through various states. The state transition depends upon the status lines received from the datapath, and depending upon the state of the FSM, control lines are sent back to datapath.

3. **top_without_mem Module:** This module has the Datapath module, FSM module and the interface between the two. Also, this module is synthesized to determine the clock period and area.

4. **top_with_mem Module:** This module interfaces the top_without_mem module with the four SRAM memories. This is the module which is simulated to verify the design.

Besides these modules, the memories are initially loaded with input values from the test fixture, using the $readmemh system call. The files used to load these memories are hex files. Similarly, after the program execution, the $writememh call is used to write the output SRAM to a file in hex format. Also, the output in decimal is written to a .dat file for easy validation. The test fixture is also used to pass initial signals to start the design.

Memory Interface Specifications

Four SRAMs are interfaced externally with the design. These SRAMs are used for storing the input data, providing a working memory for the design to store intermediate values, and to write the output results.

Details about the four SRAMs interfaced to the design are as follows:

1. **Graph SRAM:** Details about the structure of the graph are stored in the Graph SRAM.

2. **Input SRAM:** The source destination pairs between which the shortest path needs to be computed is stored in the Input SRAM.

3. **Work SRAM:** The work SRAM is used as the working memory for the design. All the intermediate values for the computation are stored in Work SRAM.

4. **Output SRAM:** The shortest distance between a source-destination pair and the path from source to destination are stored in the Output SRAM.

Each of the SRAM memories is assumed to have a read and write delay of 4 ns each. Writing to the memory is synchronous, and is completed at the positive clock edge when the Write Enable signal to the memory is asserted. Reading from the memories, however, is asynchronous. The widths and functionality of the buses used to connect to the SRAMs are described below:

Input SRAM:

Port	Signal Name	Type/Width (bits)	Function
Read Address	inputSRAM_Addr	Input/10	Address to read from SRAM
Read Data	inputSRAM_Data	Output/8	Output Data from SRAM

Graph SRAM:

Port	Signal Name	Type/Width(bits)	Function
Read Address 1	graphRead1_Addr	Input/13	Address to read from port 1
Read Data 1	graphRead1_DataInput	Output/128	Output data from port 1
Read Address 2	graphRead2_Addr	Input/13	Address to read from port 2
Read Data 2	graphRead2_DataInput	Output/128	Output data from port 2

Work SRAM:

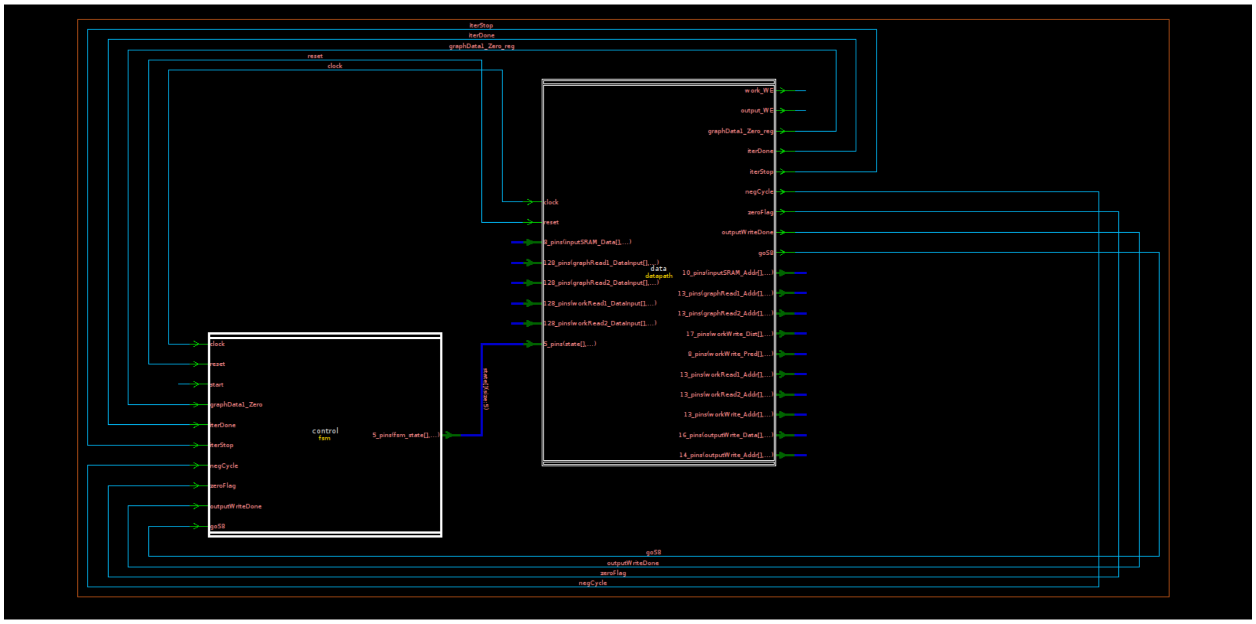
Port	Signal Name	Type/Width(bits)	Function
Read Address 1	workRead1_Addr	Input/13	Address to read from port 1
Read Data 1	workRead1_DataInput	Output/128	Output data from port 1
Read Address 2	workRead2_Addr	Input/13	Address to read from port 2
Read Data 2	workRead2_DataInput	Output/128	Output data from port 2
Write Address	workWrite_Addr	Input/13	Address to write data
Write Data	workWrite_Dist/Pred	Output/128	Data to be written
Write Enable	work_WE	Input/1	Enable signal to write data

Output SRAM:

Port	Signal Name	Type/Width(bits)	Function
Read Address 1	ReadAddress	Input/14	Address to read from port 1
Read Data 1	ReadBus	Output/16	Output data from port 1
Write Address	outputWrite_Addr	Input/14	Address to read from port 2
Write Data	outputWrite_Data	Input/16	Output data from port 2
Write Enable	output_WE	Input/1	Enable signal to write data

In the top_without_mem module, the datapath and the FSM are interfaced using status lines going to the FSM and control lines coming from the FSM to the datapath.

The following schematic obtained from Synopsys Design Compiler shows the interface lines between the datapath and the controller.


iterStop
iterDone
graphData1_Zero_reg
reset
clock
work_WE
output_WE
graphData1_Zero_reg
iterDone
iterStop
negCycle
zeroFlag
outputWriteDone
goS8
clock
reset
8_pins(inputSRAM_Data[],...)
128_pins(graphRead1_DataInput[],...)
128_pins(graphRead2_DataInput[],...)
128_pins(workRead1_DataInput[],...)
128_pins(workRead2_DataInput[],...)
5_pins(state[],...)
data
datapath
10_pins(inputSRAM_Addr[],...)
13_pins(graphRead1_Addr[],...)
13_pins(graphRead2_Addr[],...)
17_pins(workWrite_Dist[],...)
8_pins(workWrite_Pred[],...)
13_pins(workRead1_Addr[],...)
13_pins(workRead2_Addr[],...)
13_pins(workWrite_Addr[],...)
16_pins(outputWrite_Data[],...)
14_pins(outputWrite_Addr[],...)
state[](size:5)
clock
reset
start
graphData1_Zero
iterDone
iterStop
negCycle
zeroFlag
outputWriteDone
goS8
control
fsm
5_pins(fsm_state[],...)
goS8
outputWriteDone
zeroFlag
negCycle

Technical Implementation Details

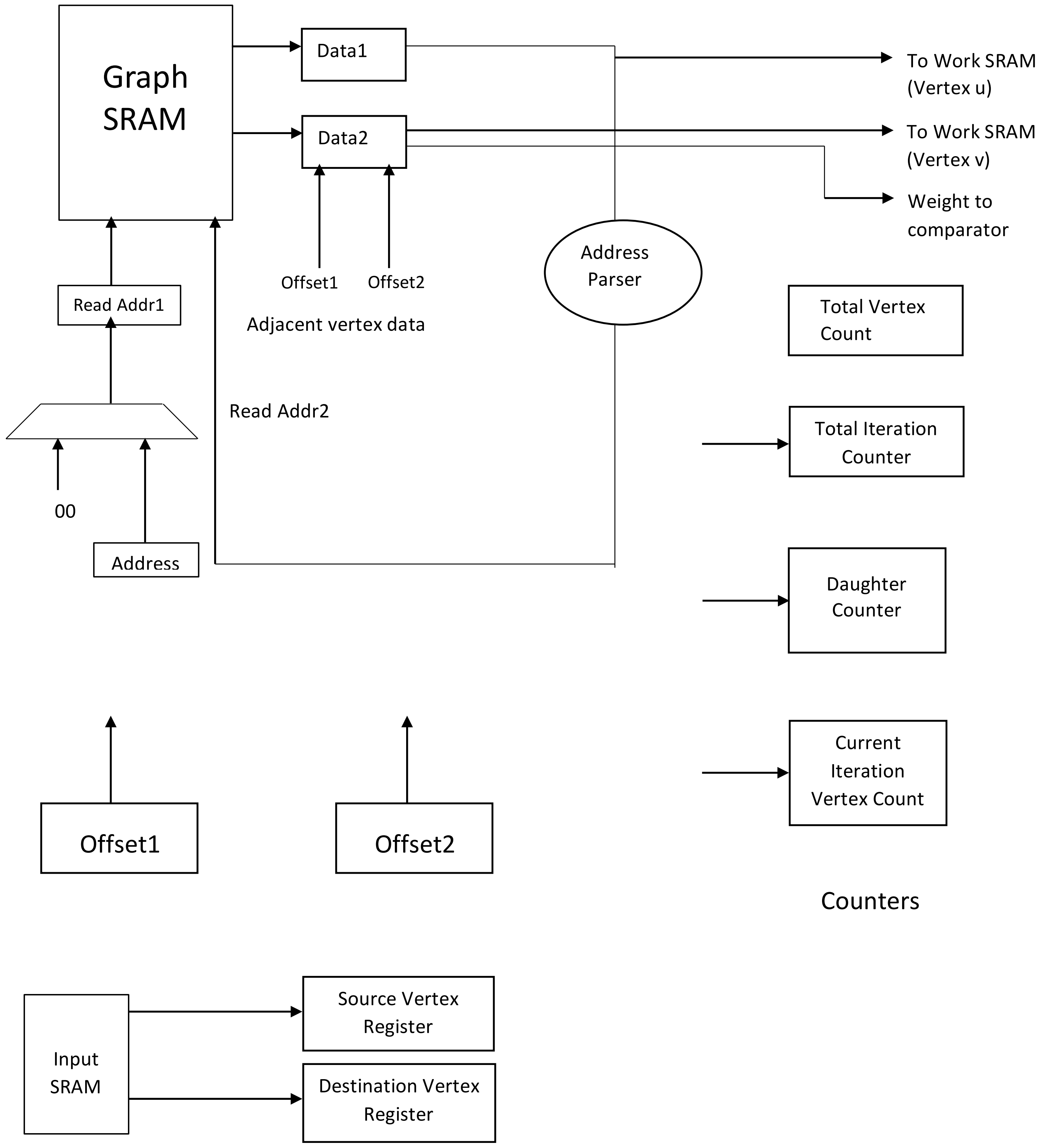
The hardware design can be divided into three categories: Circuitry for handling the reading/writing from memory by manipulation addresses, the main algorithmic computation logic and the finite state machine controller. The algorithm flow is controlled by the finite state machine which drives the inputs of the various multiplexers. To transition from one state to the next, status lines (which are counter values, comparator flags) are sent from the datapath to the finite state machine.

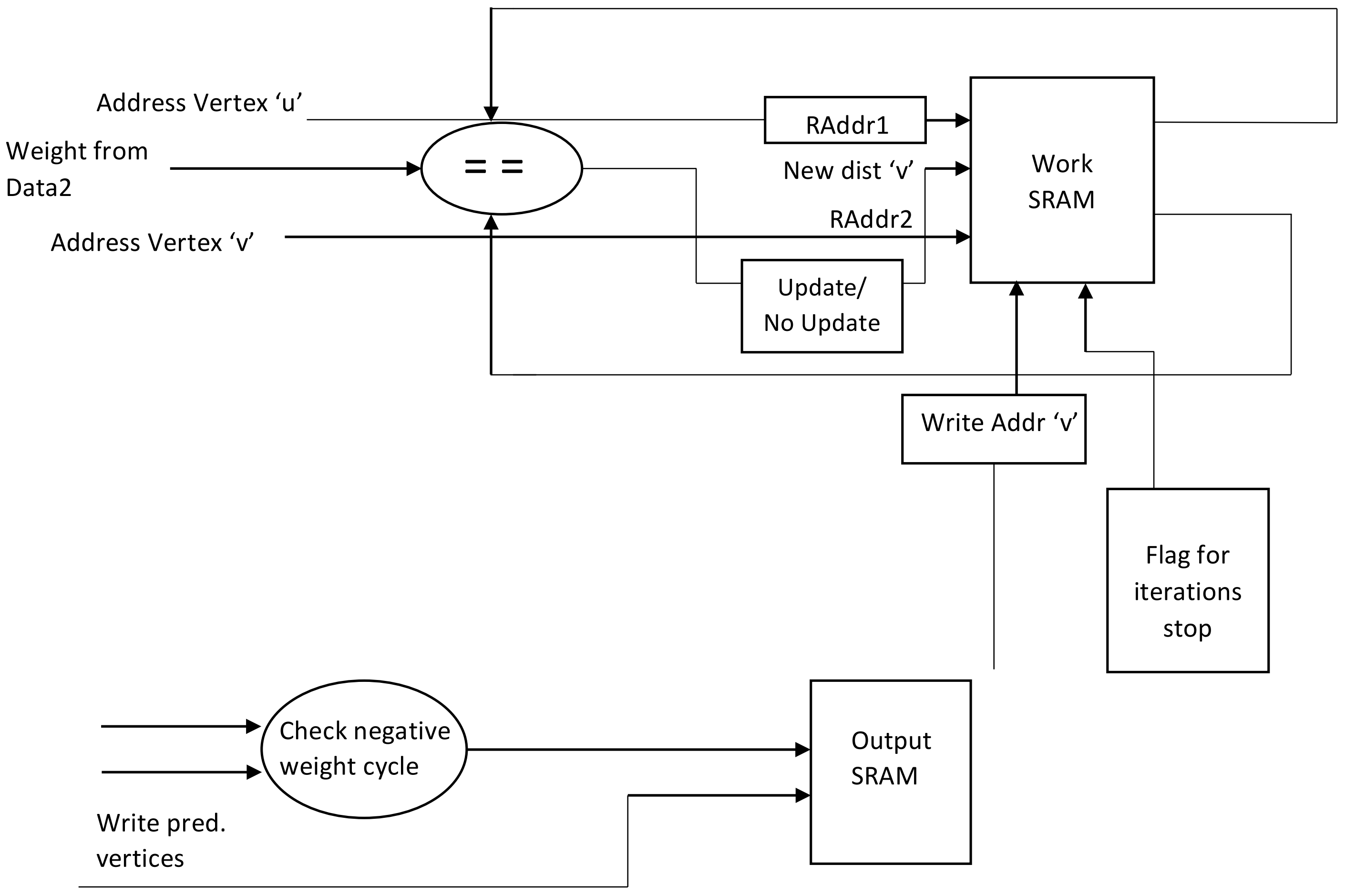
All the interfaces to memories are via registers. Thus, reading and writing to the memories is all done synchronously. The use of registers to interface with memory also has the added advantage of breaking the critical path into smaller chunks, thus reducing the overall clock period. The data read from memories is available in registers. From these registers, the data required in each cycle is read using multiplexers which are controlled according to the FSM state. Apart from the data fetch logic, the computation hardware is an adder along with a signed comparator. To control the flow of the program, various counters are employed which are incremented and decremented appropriately. Also, combinational comparator flags are employed to detect certain condition and these are sent to the controller.

The diagram below shows the various hardware logic constructs used in the design. This is a high-level view, where the function of each of the registers in a particular state is controlled by control lines from the finite state machine. To control the transition of states in the finite state machine, comparators and counters are used to send specific status signals to the controller. The operations to be performed during each FSM state are decided by multiplexer select lines in the datapath. The timing of the datapath is carefully considered while designing the controller.

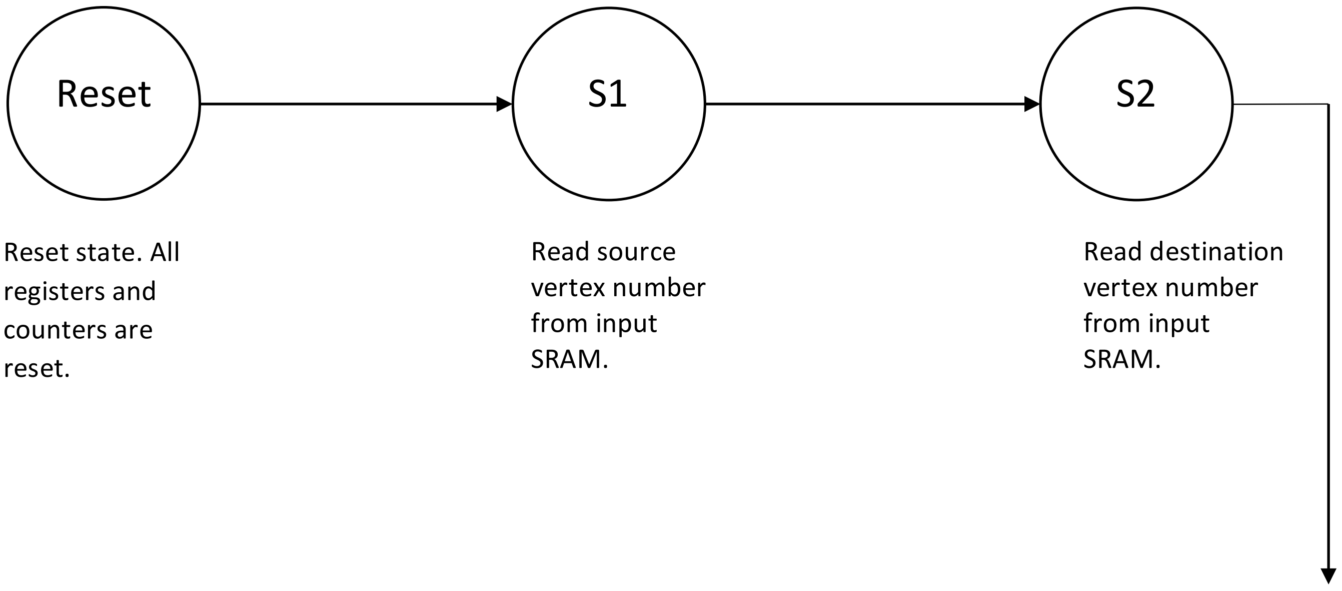
In the hardware diagram, the datapath is shown in three parts – the Graph SRAM reading logic and counters, the computation circuitry along with interface to the work SRAM and the output SRAM writing circuitry. After the datapath, the FSM is shown with a brief description of the tasks completed in each state.

Datapath


Graph SRAM
Data1
Data2
To Work SRAM (Vertex u)
To Work SRAM (Vertex v)
Weight to comparator
Offset1
Offset2
Adjacent vertex data
Address Parser
Read Addr1
Read Addr2
00
Address
Total Vertex Count
Total Iteration Counter
Daughter Counter
Current Iteration Vertex Count
Offset1
Offset2
Counters
Input SRAM
Source Vertex Register
Destination Vertex Register



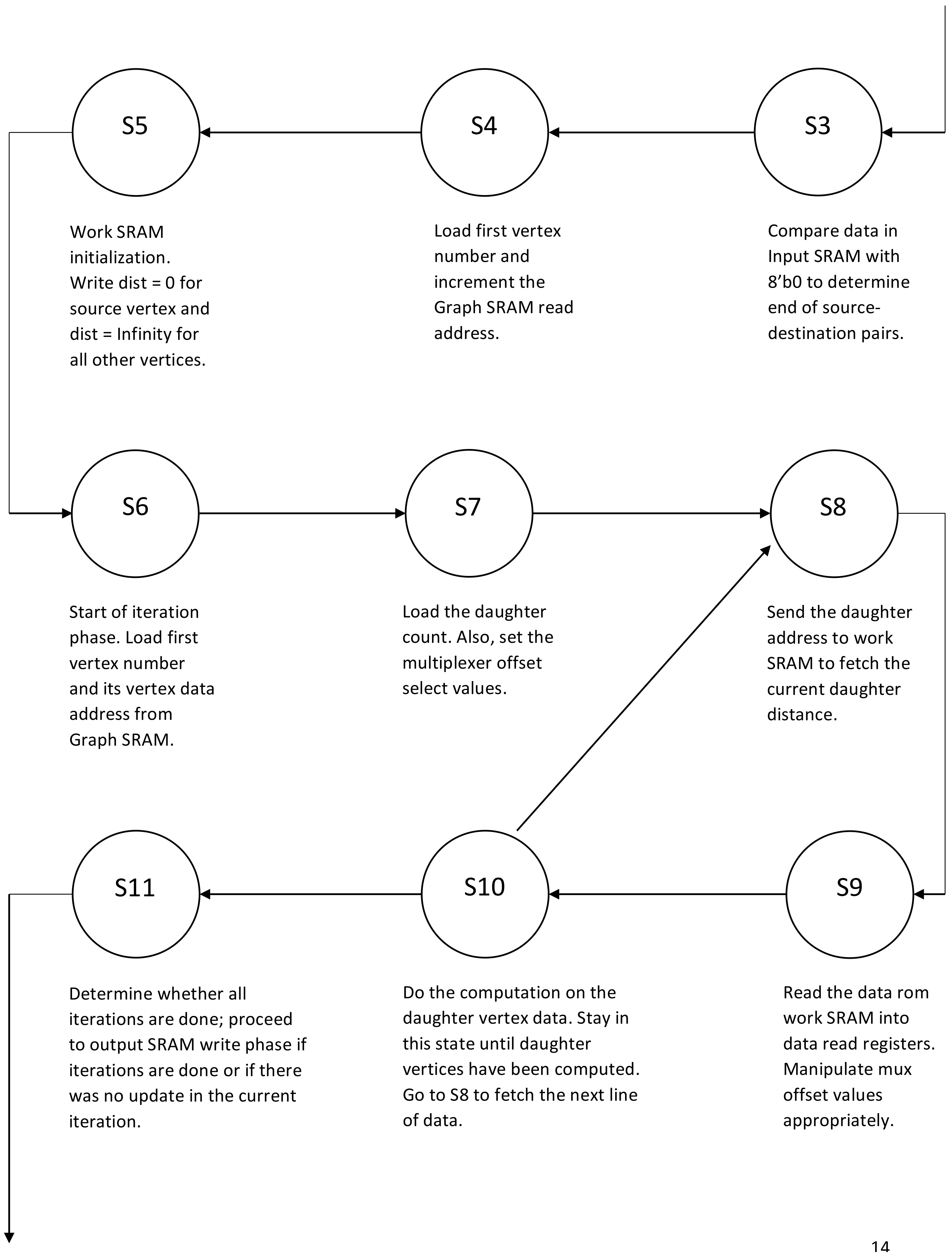
Finite State Machine Controller



Reset state. All registers and counters are reset.

Read source vertex number from input SRAM.

Read destination vertex number from input SRAM.



S5: Work SRAM initialization. Write dist = 0 for source vertex and dist = Infinity for all other vertices.

S4: Load first vertex number and increment the Graph SRAM read address.

S3: Compare data in Input SRAM with 8’b0 to determine end of source-destination pairs.

S6: Start of iteration phase. Load first vertex number and its vertex data address from Graph SRAM.

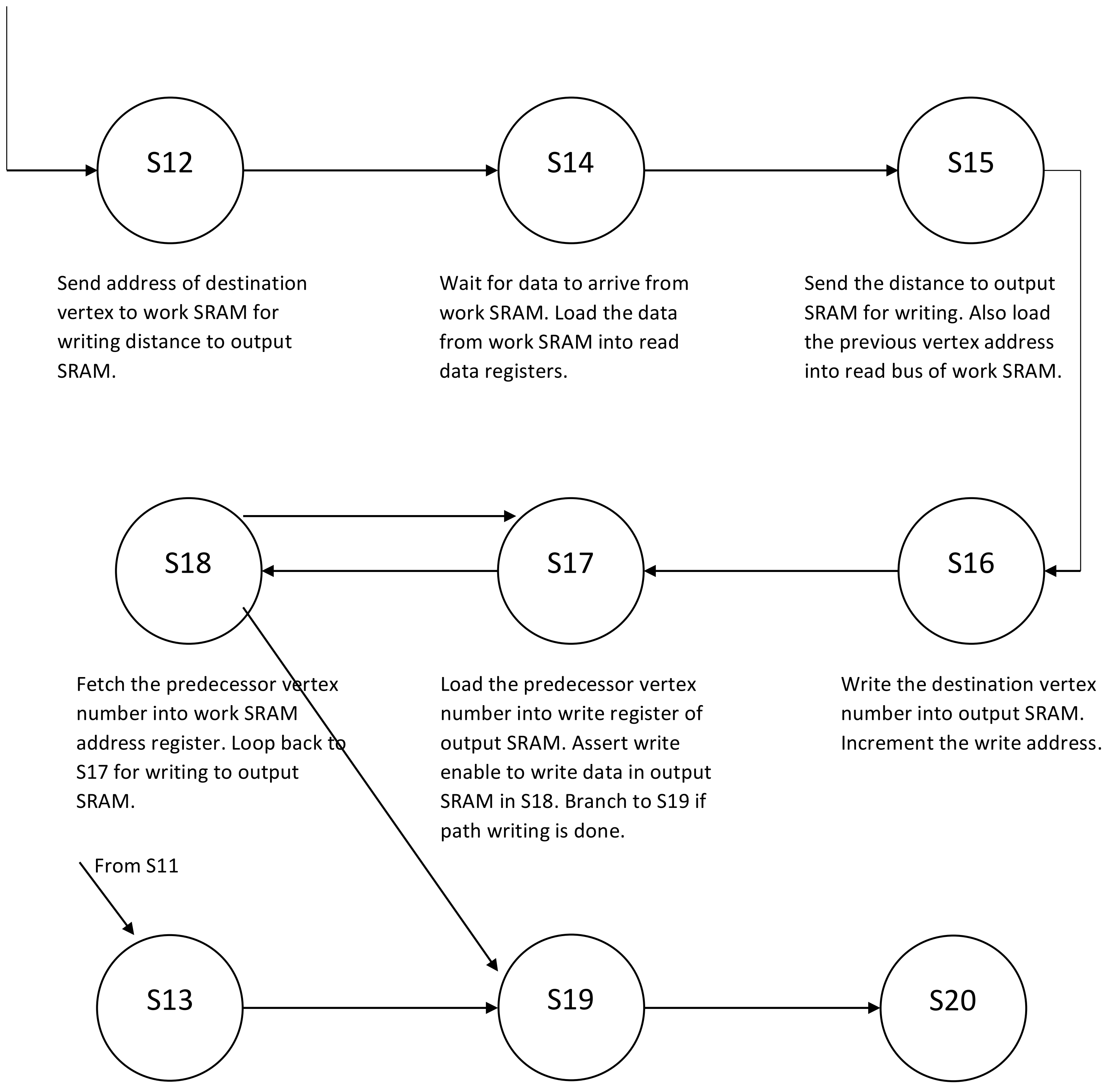
S7: Load the daughter count. Also, set the multiplexer offset select values.

S8: Send the daughter address to work SRAM to fetch the current daughter distance.

S11: Determine whether all iterations are done; proceed to output SRAM write phase if iterations are done or if there was no update in the current iteration.

S10: Do the computation on the daughter vertex data. Stay in this state until daughter vertices have been computed. Go to S8 to fetch the next line of data.

S9: Read the data rom work SRAM into data read registers. Manipulate mux offset values appropriately.



Send address of destination vertex to work SRAM for writing distance to output SRAM.

Wait for data to arrive from work SRAM. Load the data from work SRAM into read data registers.

Send the distance to output SRAM for writing. Also load the previous vertex address into read bus of work SRAM.

Fetch the predecessor vertex number into work SRAM address register. Loop back to S17 for writing to output SRAM.

Load the predecessor vertex number into write register of output SRAM. Assert write enable to write data in output SRAM in S18. Branch to S19 if path writing is done.

Write the destination vertex number into output SRAM. Increment the write address.

Negative cycle exists state. Load 16’hFFFF into output SRAM write register. Go to S20 to denote end of program execution.

Finish output SRAM writing by writing the source vertex number and sending 16’hFFFF or 16’h0 depending on whether it is the last source-destination pair or not. Branch to S1 for next S-D phase.

Program execution END state.

Description of Dataflow in the Design:

The execution starts with reading the source and destination source numbers from the Input SRAM. These vertex numbers are stored in registers. The work SRAM is then initialized with 0 or Infinity distance values in preparation for the iteration phase.

In the iteration phase, one line from the Graph SRAM is read at a time. From the data on this line, the information for each daughter vertex is extracted one by one and then processed. The address of the daughter is sent to the work SRAM for reading out the currently stored distance. After reading out the distance, it is compared with the distance obtained after adding the weight to the current parent vertex. If this distance is shorter, the distance and the previous vertex number is updated in the work SRAM. To mask the extra latency is getting the data from work SRAM, the fetch address for the next daughter is sent to the address register while the computation for the current daughter is in progress. This speeds up the program execution by avoiding any stall cycles. Similarly, to avoid latency in fetching from graph SRAM, the address to fetch the next line from graph SRAM is sent when the last daughter of the previous graph SRAM data line is being processed. The technique of doing the daughter vertex computation in parallel with data fetch and prefetching the next line data from graph SRAM reduce the execution latency of the design.

The iteration phase is controlled using counters for the current parent vertex daughter count, the two offset values – one for reading the daughter weight for computation and the other for sending the daughter address to the work SRAM, the iteration counter and the vertex count for the current iteration. The total vertex count for the graph is also stored in a register to be loaded into the vertex count register at the start of each new iteration. To reduce the execution latency of iteration phase, the computation for a daughter vertex and the address fetching for the next daughter is done in parallel. In hardware, two offset values are used to get data for two daughters at the same time. The daughter vertex computation finish is signaled when all the daughters in a particular have been computed. Similarly, the next line of data from Graph SRAM is prefetched when the last daughter of a previous Graph SRAM data line is being processed. This reduces the latency in switching from one line to the next. In the iteration phase, a flag register is used to detect if there has been no update in the distance values in two consecutive iterations. At the end of every iteration, this flag is checked and if it detects that there was no update, the iterations are stopped since the optimum distance has already been computed. The same flag is also used in the case of negative cycle checking, by checking whether there was a distance in the 'n^{th}' iteration as well for a graph with 'n' number of vertices. After each iteration, depending upon the status of these flags and the various counter the execution either continues with iterations or moves to the output SRAM writing phase.

In the output SRAM write phase, the flag is first checked to see whether a negative cycle has been detected in the graph. If a negative cycle is present, the program proceeds to writing '16'hFFFF' in the output SRAM which implies presence of a negative cycle. In the no error scenario, the design first writes the distance of the destination vertex into the output SRAM and then proceeds to trace the path from destination to the source. Once the entire path is written in output SRAM, the input SRAM is checked to see whether there are any other source-destination to be computed. The program finishes execution once all the source-destination pairs have been processed.

Verification Methodology

Testing of the Verilog code was done using various strategies such as self-checking test fixtures, viewing appropriate signals on the ModelSim waveform viewer to verify correctness of timing and state, and final output checking after each stage of the algorithm. Since the design is controlled by a Finite State Machine which moves through discrete phases such as source-destination pair reading, work SRAM initialization and iteration phase, the design was verified according to the expected outcome after each phase.

The transitions between FSM states and the values of the various design registers were verified using $display() and $monitor() system calls in the test fixture, as well as by viewing signals on the waveform viewer to check for cycle timing correctness. Also, the contents of the memories interfaced to the design were verified after each phase using the Memory List feature in ModelSim.

For functional verification, multiple test graph inputs were generated using a Matlab script. The test graph SRAM inputs used were such that all possible scenarios like daughter vertex data for a particular vertex being present on multiple lines in graph SRAM, only one daughter for a particular vertex and the presence of negative cycle in the given graph would be covered. Specific cases of the graph test inputs were also used for debugging purposes. The outputs obtained from simulation were then validated against the ideal outputs obtained from Matlab. Also, the benefit obtained by adding features like prefetching were analyzed by comparing the cycle latency obtained by running the same workload with and without those enhancements.

Performance Results

The metric used to measure the quality of the design accounted for both the area and the performance of the design. Performance of the design was measured taking into account the cycle period of the design as well as the number of clock cycles taken to complete a particular benchmark workload, and then multiplying it by the design area obtained after synthesis. Thus, both the area and the execution latency of the design were taken into account to measure the efficiency of the design.

The design was synthesized using Synopsys Design Vision. In the synthesis script, the Input and Output Delay were set as 4 ns to account for the interface with external SRAM memories. The design goal was set to achieve the least possible area while meeting setup and hold timing constraints. The clock period and area achieved after synthesis were as follows:

Clock Period: 8.05 ns

Area: 3694.47 um^2

The benchmark used to measure the performance of the design was the 'Graph_secret1.mem' file along with 'input_secret1.mem' as the test input. For this workload, the number of clock cycles for execution were obtained by simulating the design in ModelSim. The total execution latency was then obtained by taking the product with the actual clock period obtained from synthesis.

Total Simulation Time in ModelSim = 67899535 ns

Clock Period used in the test bench = 10 ns

Total Clock Cycles needed to execute the program = 6789953

Total Execution Time = 6789953 x 8.05

= **54659121.65 ns**

Conclusion

Bellman-Ford algorithm is an algorithm used to find the shortest path between any two vertices in a directed graph. The algorithm has widespread usage in networking and routing. In this project, a hardware accelerator has been designed for implementing the Bellman-Ford Algorithm. Using the given constraints for the memory interfaces, the design aims to minimize the execution time by keeping the area to a minimum at the same time.

While designing the hardware, architectural features like pipelining and prefetching were exploited to reduce the execution latency to a minimum value. The design was captured using Verilog HDL, and simulated using ModelSim. To verify the correctness of the design, multiple test workloads were executed on the simulator and the outputs were validated. The test workloads were also used to analyze the improvements in performance on account of adding the architectural enhancements.

The RTL design was then synthesized using Synopsys Design Vision. The design goal was set to obtain the least possible chip area while meeting the setup and hold timing constraints. The input and output delays to the module were set to 4 ns to account for the delays from the SRAM memories. The best possible clock period meeting the timing constraints was then obtained and the total design cell area was reported.

To summarize the performance of the design, a benchmark program was executed and the execution delay was calculated. The overall performance metric was then calculated taking the design area also into account.

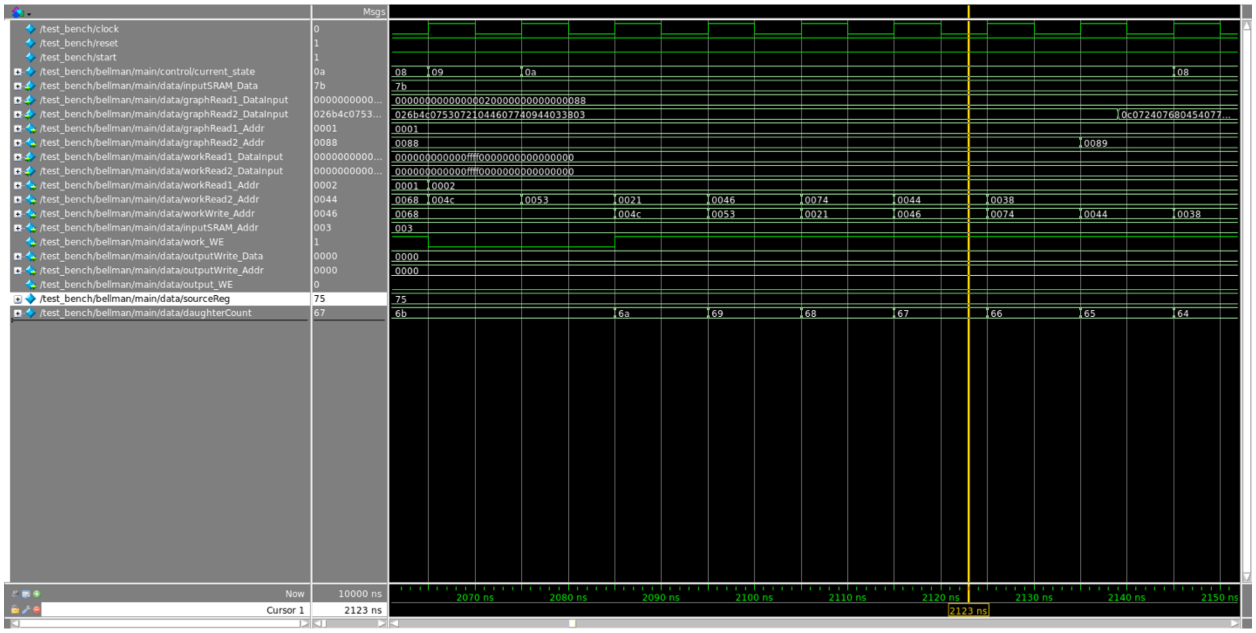
References

1. Verilog HDL – Quick Reference Guide, by Stuart Sutherland
2. Lecture Notes of ECE 520, Dr. Paul Franzon at North Carolina State University
3. Advanced Digital Design with the Verilog HDL, by Michael Ciletti
4. ModelSim User's Manual, by Mentor Graphics
5. Synthesis Quick Reference, May 2000
6. Synopsys Design Compiler Documentation

Appendix

Simulation Waveforms

1. The following waveform shows the state of the SRAM interface signals while the computation is in progress.

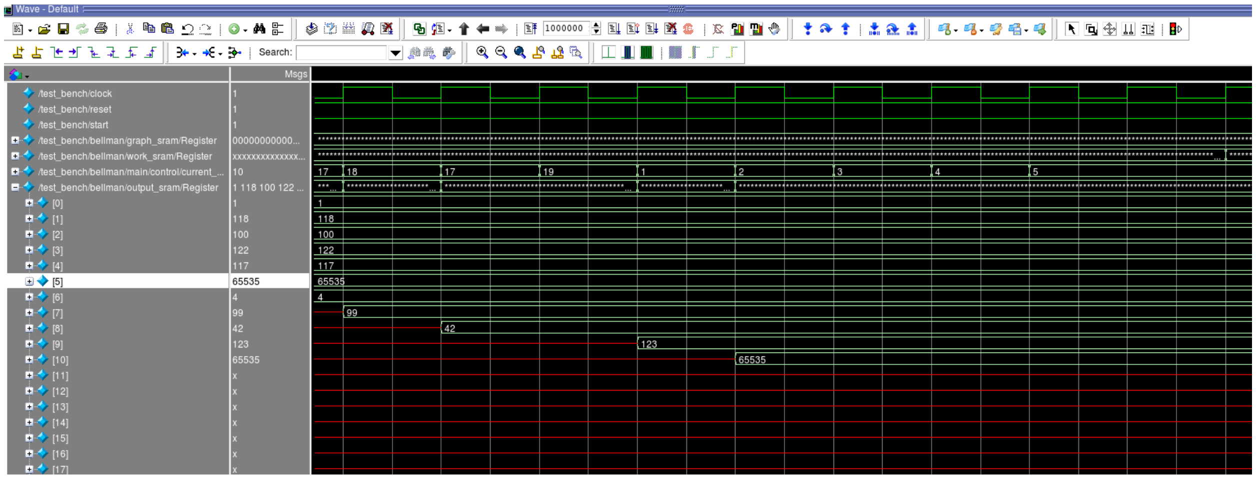


```
                                                     Msgs
/test_bench/clock                                    0
/test_bench/reset                                    1
/test_bench/start                                    1
/test_bench/bellman/main/control/current_state       0a
/test_bench/bellman/main/data/inputSRAM_Data         7b
/test_bench/bellman/main/data/graphRead1_DataInput   0000000000...
/test_bench/bellman/main/data/graphRead2_DataInput   026b4c0753...
/test_bench/bellman/main/data/graphRead1_Addr        0001
/test_bench/bellman/main/data/graphRead2_Addr        0088
/test_bench/bellman/main/data/workRead1_DataInput    0000000000...
/test_bench/bellman/main/data/workRead2_DataInput    0000000000...
/test_bench/bellman/main/data/workRead1_Addr         0002
/test_bench/bellman/main/data/workRead2_Addr         0044
/test_bench/bellman/main/data/workWrite_Addr         0046
/test_bench/bellman/main/data/inputSRAM_Addr         003
/test_bench/bellman/main/data/work_WE                1
/test_bench/bellman/main/data/outputWrite_Data       0000
/test_bench/bellman/main/data/outputWrite_Addr       0000
/test_bench/bellman/main/data/output_WE              0
/test_bench/bellman/main/data/sourceReg              75
/test_bench/bellman/main/data/daughterCount          67

Now        10000 ns
Cursor 1   2123 ns
```

08 09 0a 08
7b
00000000000000002000000000000000088
026b4c07530721044607740944033803 0c072407680454077...
0001
0088 0089
000000000000ffff0000000000000000
000000000000ffff0000000000000000
0001 0002
0068 004c 0053 0021 0046 0074 0044 0038
0068 004c 0053 0021 0046 0074 0044 0038
003
0000
0000
75
6b 6a 69 68 67 66 65 64
2070 ns 2080 ns 2090 ns 2100 ns 2110 ns 2120 ns 2130 ns 2140 ns 2150 ns
2123 ns

2. The waveform attached below shows the output SRAM being written after the end of computation for a source-destination pair.



```
                                                Msgs
/test_bench/clock                               1
/test_bench/reset                               1
/test_bench/start                               1
/test_bench/bellman/graph_sram/Register         00000000000...
/test_bench/bellman/work_sram/Register          xxxxxxxxxxxxxx...
/test_bench/bellman/main/control/current_...    10
/test_bench/bellman/output_sram/Register        1 118 100 122 ...
  [0]                                           1
  [1]                                           118
  [2]                                           100
  [3]                                           122
  [4]                                           117
  [5]                                           65535
  [6]                                           4
  [7]                                           99
  [8]                                           42
  [9]                                           123
  [10]                                          65535
  [11]                                          x
  [12]                                          x
  [13]                                          x
  [14]                                          x
  [15]                                          x
  [16]                                          x
  [17]                                          x
```