ECE 464/520 Projects 2015 – Technical Description

All projects are to be performed individually.

ECE 520 – Bellman-Ford Algorithm

Many problems can be captured as graphs. A very common task that needs to be performed on a graph is to work out the shortest spanning tree. For example, a network can be mapped as a graph and we might want to direct a packet via the shortest route. A server farm can also be mapped as a graph and we might want to move files or MPI packets with the fewest hops.

The problem of finding the shortest spanning tree between a pair of graph nodes is often solved using Dijkstra's algorithm. There are many explanations of Dijkstra's algorithm on the web including one at Wikipedia, and even a YouTube animation. However, Dijkstra's algorithm can not handle negative edge weights. A more general algorithm that can is the Bellman-Ford algorithm. Again, there are many explanations out there, including one on Wikipedai.

Your task is to build a hardware accelerator for the Bellman Ford algorithm. You will be given the following:

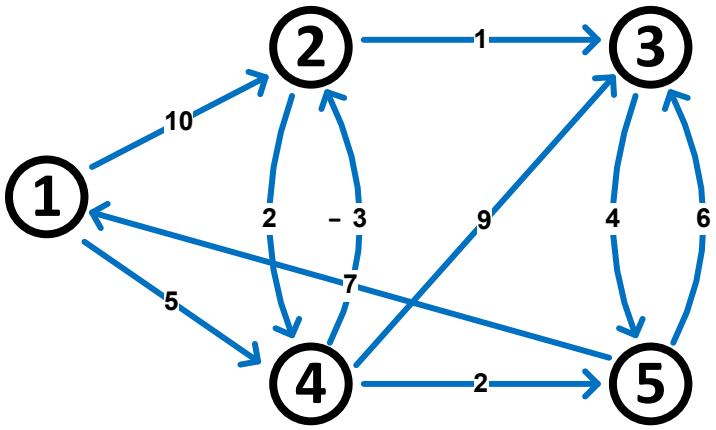
- A 3-port 128-bits wide SRAM. It will have two read ports and one write port. It will have a 4 ns access time.
- Graph.dat. This will be a specification of the graph in the format discussed below. It will be formatted to be read directly into the SRAM. We will initially provide you with a small and large example. An additional large example might be generated during the project.
- Input.dat. This will be a list of pairs of nodes that you are to run through your hardware in order to find the shortest span. You should read this in through your test fixture. The format is given below.
- Output.dat. This is a file you will produce with solutions to the problems specified in input.dat. The format is given below.

Problem Inputs

We are going to provide you with the following inputs via files that you can read into memories in Verilog or into a higher level simulation.

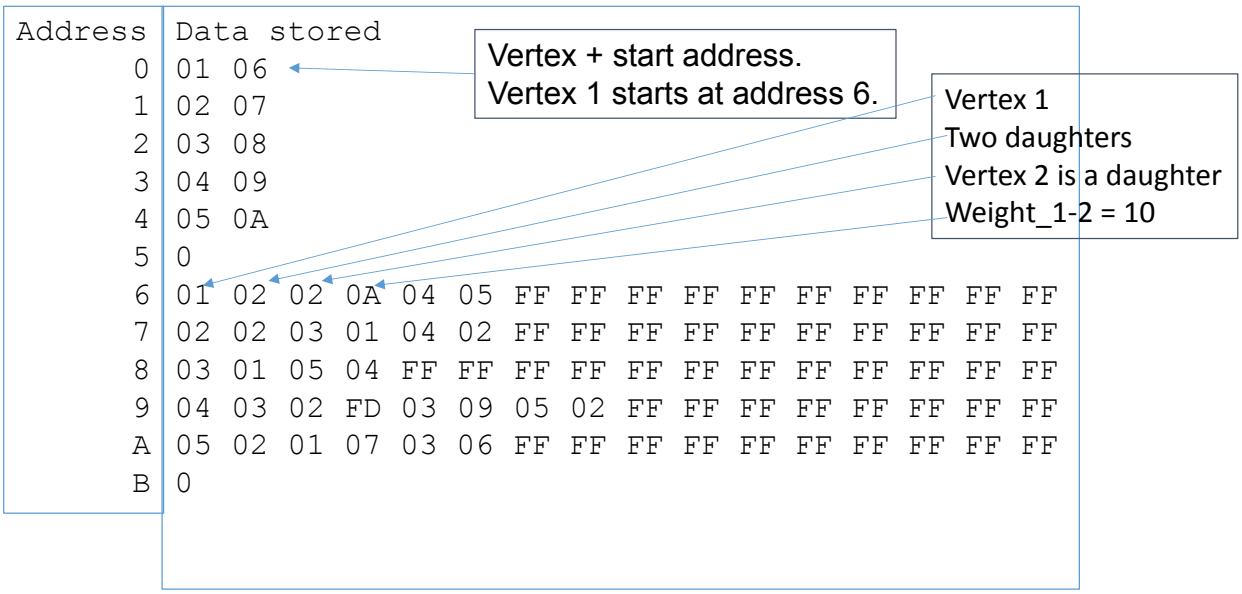
Graph.dat	- Contains an index of starting addresses for each vertex, followed by a weighted graph represented in the form of adjacent list. - For each vertex, we keep a list of all adjacent vertices and their weights. - Weight value is given as a two's complement 8-bit number - "FF"s are used to fill up rest of the 128 bits entry - **0** represents the end of the graph.
Input.dat	- Contains multiple sets of source and destination vertices. **FF** is the end of a set of source and destination pair, and **0** represents the end of all input sets.
Output.dat	- After your hardware has completed its simulation run, write the results to Output.dat. For each set of inputs, write out the weighted path length, followed by vertices on the entire shortest path, from source to destination. - If there are multiple paths with the same weighted shortest path length, just show the one with the largest father node number. - **FFFF** is the end of a set of input, and **0** represents the end of all inputs.

This small scale example help you better understand the project inputs. Note the weight going from 4 to 2 is negative.



The appropriate entries would be

Graph.dat:



Input.dat	1 # Source 2 # Destination FF # End of Pair 1 5 FF 5 2 0 # End of Inputs

Output.dat	2 1 4 2 FFFF 7 1 4 5 FFFF 9 5 1 4 2 0	# Weighted Path Length # Vertices on Path # End of Pair # End of all Pairs

If the graph contains a negative cycle weight, then the weighted path length is listed as FF.

Note these files are for high level code, corresponding HEX files will be given for verilog design.

Available Memory and data layouts

Name	Purpose	Size	Number of Ports
sram_2R (Graph)	Store graph to search	8K × 128 bits	2 Read
sram_1R (Input)	An input buffer for all source&destination pairs	1024 × 8 bits	1 Read
sram_1R1W (Output)	An output buffer to write results to	16K × 16 bits	1Read + 1 Write
sram_2R1W (Working)	A working memory. You can use it to store intermediate results if you like BUT you **CAN NOT use it to store copy of Graph**	8K × 128 bits	2 Read + 1 Write

To accommodate the delays inherent in the SRAMs, change ALL the output load delays in your design to 4 ns. i.e. The SRAMs take 4 ns of asynchronous time to respond and this is how you tell synthesis to allow for the 4 ns.

Graph layout in sram_2R:

Address Index Part: each number takes 64 bits wide, each entry stores address information for one vertex. As shown here (one entry in sram):

1	6
64 bits	**64 bits**

Adjacent List Part: each number takes 8 bits wide; each entry stores up to information for 8 daughter vertices (note each daughter vertex includes its vertex number and edge weight). One entry in sram is shown here:

1	2	2	10	4	5	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
①-----															

128 bits

① 16 bits

Additional Project Guidelines

Part I Design Rules Checklist

In this part, common concerned design rules are listed, some of them have already been discussed and clarified on forum. Designs that fail to meet these rules will not be eligible for performance credits.

1. Memory use
    a) The four sram modules are the only memories allowed to use, it is not allowed to generate your own memory.
    b) For each sram module, only one instance is allowed to use in testbench. E.g. you can only have one "working sram"
    c) It is not allowed to change the number of ports and width of ports.
    HOWEVER, the depth of srams can be adjusted to meet your need. Note that this is not a nothing-to-lose benefit, by making the sram larger, it increases the addressing overhead.
    d) You are not going to synthesize the sram modules, so only put them in your top level testing module.
    e) It is not allowed to keep a copy of graph content in working memory.
    f) The graph and input content should be loaded into corresponding srams at the beginning of your testbench. At the end of your testbench, please load the content of "output buffer" to an output textfile, in hex or decimal.
2. Register files
    a) It is allowed to generate and use register files in forms of flip-flips in your design. Note that though such register files can be used as "sram" without

huge delay, they add a lot area.

3. Synthesis

a) Only synthesize your design, excluding srams

b) It is allowed to modify the synthesis script to achieve a better performance, but all the timing numbers are NOT allowed to be changed with only one exception.

c) The **exception** is: Change output delay to 4ns to include the sram read/write delay.

Part II Submission Guidelines

1. Grading Policy

TA will run the functional simulation as well as synthesis for the grading.
Waveforms will also be checked. Here is the grading breakdown:

a) Simulation

Full credits: 50

i. Successfully compiled without any warnings or errors

ii. Match both small and large cases in **content and format**

iii. No un-defined values in waveform ("red lines")

Points Offs:

-1 per warning in compile

-2 per error in compile

-1 per un-defined value in waveform

-2 per output none-matching

b) Synthesis

Full credits: 30

i. Successfully compiled without any non-ignorable warnings or errors (check Tutorial #1 for the list of ignorable warnings in synthesis)

ii. Met setup and hold checks

Points Offs:

-5 any latches generated

-1 per warning in compile

-2 per error in compile

c) Performance

Full Credits: 10

Metric: area × (# of cycles for large input file) × clock period

d) Report

Full Credits: 10

2. Submission Checklist

You must submit a single zip file called **ECE520_project.zip**.
ECE520_project.zip must contain the following (any deviation from the following requirements may delay grading your project and may result in points off, late penalties, etc.):

a) Project Report. This must be a single PDF document named report.pdf. You must follow the cover page format as posted on course website.

b) Verilog code: including all your design code (.v files) and testbench

c) Input files for functional simulation:

i. Small case: input_small.dat, graph_small.dat

ii. Large case: input_large.dat, graph_large.dat
iii. You don't need to submit output files, HOWEVER, please make sure write the content of "output buffer" into an output text file in your testbench.

d) All four sram files

e) modelsim.ini

f) Setup files for synthesis:
   i. Synopsys_dc.setup
   ii. Setup.tcl, read.tcl, Constraints.tcl, CompileAnalyze. Tcl

g) Synthesis reports:
   i. Setup check report
   ii. Hold check report
   iii. Area report