ECE 464/520 Projects 2013 – Technical Description

All projects are to be performed individually. Please keep checking the announcements on moodle for updated information.

ECE 520 – Dijkstra's Algorithm (Modified Version with highlights)

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.

Your task is to build a hardware accelerator for Dijkstra's algorithm. You will be given the following:

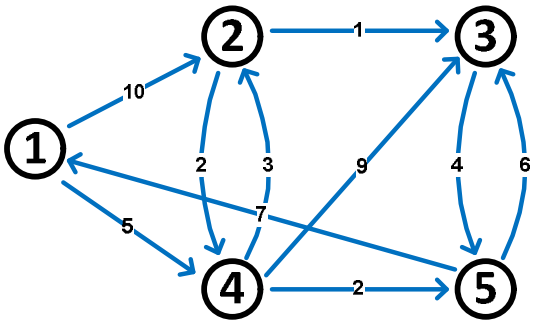
- A 3-port 128-bits wide SRAM. It will have two read ports and one write port. It will have a 4 ns access time.
- Graph.dat. This will be a specification of the graph in the format discussed below. It will be formatted to be read directly into the SRAM. We will initially provide you with a small and 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	- Number of vertices is on first line(sram entry), Followed by a weighted graph represented in the form of adjacent list - For each vertex, we keep a list of **all vertices** and their weights - Weight value is less than 10 (decimal) - Weight of "0" indicates no edge between the two vertices - "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



The appropriate entries would be

Graph.dat	5 1 0 2 10 3 0 4 5 5 0 FF FF FF FF FF FF 1 0 2 0 3 1 4 2 5 0 FF FF FF FF FF FF 1 0 2 0 3 0 4 0 5 4 FF FF FF FF FF FF 1 0 2 3 3 9 4 0 5 2 FF FF FF FF FF FF 1 7 2 0 3 6 4 0 5 0 FF FF FF FF FF FF 0
Input.dat	1 # Source 2 # Destination FF # End of Pair 1 5 FF 5 2 0 # End of Inputs
Output.dat	8 # Weighted Path Length 1 # Vertices on Path 4 2 FFFF # End of Pair 7 1 4 5 FFFF 15 5 1 4 2 0 # End of all Pairs

批注 [W1]: Number of vertices

批注 [W2]: List for Vertex "1" Note only "2" and "4" are daughter to "1", other vertices have weight value of 0

批注 [W3]: List for Vertex "2"

批注 [W4]: End of graph

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

Note of sram using:

1. It is not allowed to change the ports of sram, as well as the width of sram.
2. The HEX version of graph and input will need to be preloaded into corresponding srams, in your testbench.
3. You are free to formatting the working sram as you like, but it is not allowed to change the formatting of graph.

Graph layout in sram_2R:

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-----|

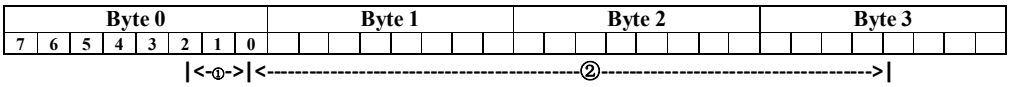
|---------- **128 bits** ----------|

① 16 bits

ECE 464 – Simple Packet Router

You are to design a simple 4x4 packet router. A packet router uses information in each packet on the input ports to determine which output port to send it to. The specifications are as follows:

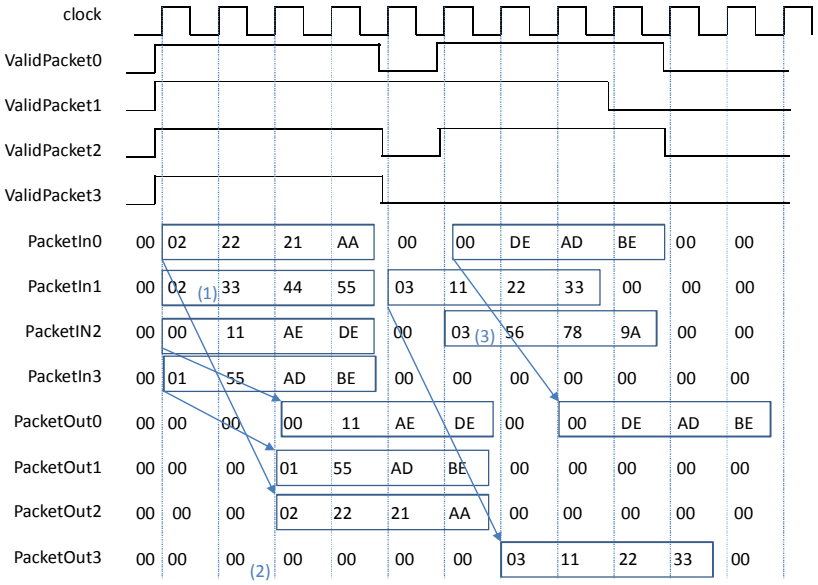
- There are four input ports, each 8-bits wide (PacketIn0 to PacketIn3)
- There are four input flags, each one-bit wide. These are high when valid packets are arriving at the appropriate input port. (ValidPacket0 to ValidPacket3). ValidPacket is low initially until a valid packed is presented at that input.
- There are four output ports, each 8-bits wide (PacketOut0 to PacketOut3)
- Each packet is 4 Bytes long, including the header
- The low order two bits of the header byte packet specify the output port. (1) Ignore the rest of the first (header) byte of the packet. The other bits in the header byte are 0. The bytes are big-endian (i.e. bit[7] is the high order bit.



① Output Port ② Payload

- The next 3 Bytes of the packet are the payload.
- The entire packet, including header byte is to be transmitted unless it is dropped because of the lack of capacity at the output port (see below). If a packet is dropped it simply disappears and is not forwarded.
- You must be able to process Packets **arriving at full rate.** i.e. There is no means to selectively reduce the rate of arrival at any of the input ports. The packets can arrive without any gaps in between if the ValidInput flag stays high.
- At any time more than one of the incoming packets might have the same output port destination. You only need to transmit one of these packets. You are to drop the packet arriving later or coming from the higher numbered port. (Thus there is no need for FIFOs buffers)
- The delay through the router must be two clock cycles. . Put all 0's out if not outputing a valid packet.
- Reset is an input to this design to get it into a suitable initial state (not shown below).

Exemplar Timing Diagram



Notes:
1. Packet going port 2, same destination as packet coming in port 0, a lower numbered port. This packet is dropped.
2. Output 0's when not outputting packets
3. Output port 3 already in use – this packet is dropped.

Requirements

Together with this project description you are provided with three files:

1. PacketInV0.h
2. PacketInV1.h
3. PacketOutV0.h

The PacketIn files specify sample inputs in hex format (see below) while the PacketOutV0.h file is the sample output associated with PacketInV0.h.

You are to turn in three files:

1. PacketProcessor.v. This is a complete Verilog descripton of your Packet processor including the synthesizable modules and test fixture AS ONE FILE. This file will be run through code comparison tools comparing it to all other examples of this project.

2. PacketOutV1.h . The output file corresponding to PacketInV1.h in the required format.
3. ProjectDescription.pdf . A PDF of your project report, including codes (as an appendix), and sufficient synthesis output results to convince a reader that the design is synthesis clean.

The formats of the input and output files are as follows, using the example above to illustrate. All data is in hex. Note, don't start processing bits until the reset is inactive.

PacketInV0.h

```
// ValidPacket (4 bits) : PacketIn0  (8) : Packet In1 (9): PacketIN 2 (8) : PacketIn3 (8)
0 00 00 00 00 // clock cycle 0
F 02 02 00 01 // clock cycle 1
F 22 33 21 AA
F 21 44 AE AD
F AA 55 DE BE
1 00 03 00 00
7 00 11 03 00
7 DE 22 56 00
7 AD 33 78 00
5 BE 00 9A 00
0 00 00 00 00
0 00 00 00 00
```

PacketOutV0.h

```
// PacketOut0 (8 bits) : PacketOut1 (8) : PacketOut2 (8) : PacketOut3 (8)
00 00 00 00
00 00 00 00
00 00 00 00
00 01 02 00
11 55 22 00
AE Ad 21 00
DE BE AA 00
00 00 00 00 03
00 00 00 11
DE 00 00 22
AD 00 00 33
BE 00 00 00
```