ECE 464/520 Projects 2013 – Technical Description

All projects are to be performed individually.

ECE 520 – Dijkstra’s 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.

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** | - 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 less than 10 (decimal)  
- “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

<table>
<thead>
<tr>
<th>Graph.dat</th>
<th>1 6 2 7 3 8 4 9 5 10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 2 2 10 4 5 FF FF FF FF FF FF FF FF</td>
</tr>
<tr>
<td></td>
<td>2 2 3 1 4 2 FF FF FF FF FF FF FF FF</td>
</tr>
<tr>
<td></td>
<td>3 1 5 4 FF FF FF FF FF FF FF FF FF</td>
</tr>
<tr>
<td></td>
<td>4 3 2 3 3 9 5 2 FF FF FF FF FF FF FF</td>
</tr>
<tr>
<td></td>
<td>5 2 1 7 3 6 FF FF FF FF FF FF FF FF</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Purpose</th>
<th>Size</th>
<th>Number of Ports</th>
</tr>
</thead>
<tbody>
<tr>
<td>sram_2R (Graph)</td>
<td>Store graph to search</td>
<td>8K × 128 bits</td>
<td>2 Read</td>
</tr>
<tr>
<td>sram_1R (Input)</td>
<td>An input buffer for all source&amp;destination pairs</td>
<td>1024 × 8 bits</td>
<td>1 Read</td>
</tr>
<tr>
<td>sram_1R1W (Output)</td>
<td>An output buffer to write results to</td>
<td>16K × 16 bits</td>
<td>1 Read + 1 Write</td>
</tr>
<tr>
<td>sram_2R1W (Working)</td>
<td>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</td>
<td>8K × 128 bits</td>
<td>2 Read + 1 Write</td>
</tr>
</tbody>
</table>
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):

<table>
<thead>
<tr>
<th>1</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>----------------------------------------</td>
<td>----------------------------------------</td>
</tr>
<tr>
<td>64 bits</td>
<td>64 bits</td>
</tr>
</tbody>
</table>

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 10 4 5 FF FF FF FF FF FF FF FF FF FF FF FF FF FF

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 packet is presented at that input.
- There are four output ports, each 8-bits wide (PacketOut0 to PacketOut3)
- There are four output ports associated with each INPUT port. These ports are labeled Fail0 to Fail3. They must go high if an incoming packet can not be accepted because the
intended output ports buffer is full. Your test fixture should stop sending packets to that port until Failx goes low.

- 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.)

<table>
<thead>
<tr>
<th>Byte 0</th>
<th>Byte 1</th>
<th>Byte 2</th>
<th>Byte 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

① Output Port  ② Payload

- The next 3 Bytes of the packet are the payload.
- 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. If the corresponding buffer is empty, then in this case, you need to transmit the packet from lower numbered port, and store the one from higher numbered port, or the one coming late. Also, you might have a backlog of packets waiting for a specific output port. Each output port should have a buffer of 4 packets, namely 16 bytes. If the buffer becomes full, you have to notify the failed input connection via the Failx line.
- 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, UNLESS the buffer becomes full. The packets can arrive without any gaps in between if the ValidInput flag stays high.
- Once more than one input port need buffering, the order of buffering should be from the lowest numbered port to highest numbered port.
- If ONE of the 4 buffers gets full, then we should wait by re-sending the entire testbench line, until the buffer has vacant position. At the same time, all the FIFOs are sending out the byte at their head, unless empty. If empty, then we make up a “00” as the output.
- The delay through the router must be two clock cycles unless buffered. Put all ’00’s out if not outputing a valid packet.

If the corresponding output buffer is not empty, you need to ensure its pop-out immediately after the previous packet finished transmitting.
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 0, same destination as packet coming in port 0, a lower numbered port. This packet is buffered.
2. Output 0’s when not outputting packets

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 description 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) : Packet In0 (8) : Packet In1 (8) : PacketIn 2 (8) : PacketIn3 (8)
0 00 00 00 00 // clock cycle 0
F 02 02 00 01 // clock cycle 1: All ports have input streams, so valid bit = 4'b1111 = 4'hF
F 22 33 11 55 //
F 21 44 AE AD
F AA 55 DE BE
2 00 03 00 00 // Only Port 1 has input stream, so valid bit = 4'b0010 = 4'h2
7 00 11 03 00 // Now Port 0 and 2 also have input streams, so valid bit = 4'b0111 = 4'h7
7 DE 22 56 00
7 AD 33 78 00
5 BE 00 9A 00 // In this cycle, Port 1 has finished the previous 4-Byte Packet, and no new input stream,
//so valid bit = 4'b0101 = 4'h5
0 00 00 00 00
0 00 00 00 00

PacketOutV0.h

// PacketOut0 (8 bits) : PacketOut1 (8) : PacketOut2 (8) : PacketOut3 (8) : Fail(4)
00 00 00 00 0
00 00 00 00 0
00 00 00 00 0
00 01 02 00 0 // Now the 1st byte of the packet from PacketIn1 is buffered, i.e., 02
11 55 22 00 0 // Now the 2nd byte of the packet from PacketIn1 is buffered, i.e., 33
AE AD 21 00 0 // Now the 3rd byte of the packet from PacketIn1 is buffered, i.e., 44
DE BE AA 00 0 // Now the 4th byte of the packet from PacketIn1 is buffered, i.e., 55
00 00 02 03 0 // Now the buffered 4-byte packet at OutPort2 could be transmitted
00 00 33 11 0 // Now the 1st byte of the packet from PacketIn2 is buffered, i.e., 03
DE 00 44 22 0 // Now the 2nd byte of the packet from PacketIn2 is buffered, i.e., 56
AD 00 55 33 0 // Now the 3rd byte of the packet from PacketIn2 is buffered, i.e., 78
BE 00 00 03 0 // Now the 4th byte of the packet from PacketIn2 is buffered, i.e., 9A, however,
//port 3 now is vacant, we need to pop the 1st byte of FIFO at port 3, i.e., 03
00 00 00 56 0 // Though the input stream has finished, we still need to finish poping out the FIFO
00 00 00 78 0
00 00 00 9A 0

Some error-prone special cases:

Case 1:
……………… // Hundreds of lines omitted here….
// Suppose now the size of the buffer at OutPort 2 is 3, i.e., only 1 vacant slot left
F 02 02 02 01 // Now another 3 packets will be routed to OutPort 2.
// So now, the solution will be: put the Fail[2] to High, and stall at this input line. Outputing the
head of buffers for all OutPorts, until there are 3 vacant slots at OutPort 2. Then we can proceed
to the next input line.

Case 2:

……………… // Lines omitted here….
3 01 02 00 00
7 AA AA 02 00 // Suppose now OutPort2 is full. Now the input stalls at this line.
//Outputing the head of buffer at OutPort2 until a vacant slot is reserved. The //corresponding
output vector should be(suppose all other OutPort FIFOs are not full):
// If the buf is empty, then we use “00” to make up.
7 BB BB AA 00

Case 3:

// Suppose now OutPort2 and OutPort2.buf[0] are engaged.
2 00 02 00 00 // The new routed packet(@InPort1) enters Buf[1]
3 02 aa 00 00 // The new routed packet(@InPort0) enters Buf[2]
7 dd bb 02 00 // The new routed packet(@InPort2) enters Buf[3]
f ee cc ff 02 // The new routed packet(@InPort3) sees the overflow of
 // OutPort2’s buffer and gets stalled..

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>XX</td>
<td>XX</td>
<td>02</td>
<td>02</td>
<td>02</td>
<td>02</td>
</tr>
<tr>
<td>XX</td>
<td>XX</td>
<td>aa</td>
<td>dd</td>
<td>ff</td>
<td></td>
</tr>
<tr>
<td>XX</td>
<td>XX</td>
<td>bb</td>
<td>ee</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XX</td>
<td>XX</td>
<td>cc</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Here by illustrating this example, I’m trying to say that the Packet organization in the buffers are
aligned. When the 4th line at InPort3 (02) enters the port, it sees the overflow, since there is no
room for a new packet to be buffered, based on the buffer’s data layout organization.