Project Part 3: Input Memory Module

Part 3: Input Memory Module [20 points]

The goal of Part 3 is to construct and test the Input Memory module, which holds two memories, which will be used to buffer the input matrix and the sparse input vector. This module is parameterized by `M`, `N`, and `INW`. (See Project Overview Section 4 for the requirements on these parameters.) The following diagram shows the top-level block diagram of this module (Figure 3.1).

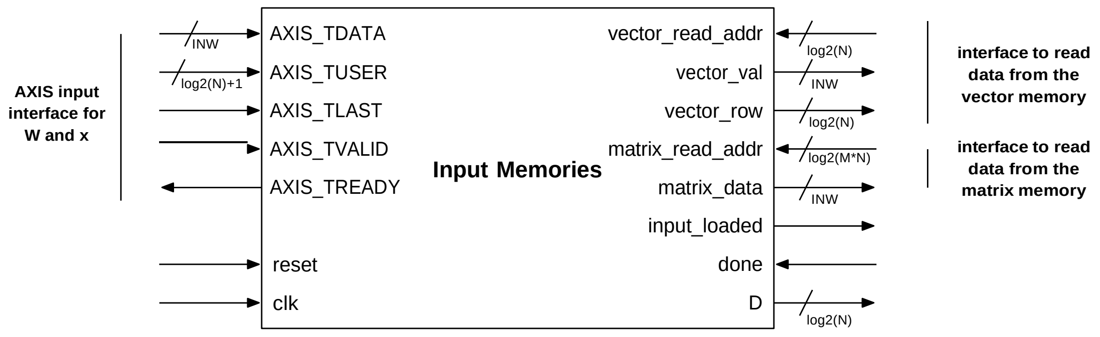

Figure 3.1. Input Memory module.

Internally, this module will contain control logic and two memories: one for holding the matrix *W* (called the *matrix memory*) and one for holding the sparse input vector *x* (called the *vector memory*) in the compressed sparse format described in Project Overview Section 3. On the left of Figure 3.1 you will see an AXI-Stream input interface, which will receive the input data (a matrix and sparse vector). On the right of Figure 3.1 you will see `vector_*` and `matrix_*` ports. These provide an interface that allows the rest of your MSpVM system to read the data stored in the input memories. Here is a specification of each port:

- `AXIS_TDATA`, `AXIS_TUSER`, `AXIS_TLAST`, `AXIS_TVALID`, and `AXIS_TREADY` collectively form an AXI-Stream interface used to load input data into the module. See Project Overview Section 5 for a description of AXI-Stream, and see the subsection labeled "Input Protocol" below for information on how this AXI-Stream interface will be used to load both input matrices and vectors. In your top-level MSpVM system, these signals will be directly connected to the top-level `INPUT_T*` signals (as shown in Figure 1 in Project Overview Section 4).

- The `input_loaded` signal is an output that your module will use to indicate when its internal memories hold a complete matrix and vector. We will refer to this as the "input_loaded" state. That is, when your module is in the input_loaded state, this signal will equal 1. (Later, your top-level MSpVM system will use this signal to determine when it is time to begin performing the computation.)

- `matrix_read_addr` and `matrix_data` are used to read data from the matrix memory. When your module is in the input_loaded state, then the `matrix_read_addr` signal will select an address in the matrix memory, and the matrix memory's output will be provided on `matrix_data`.

- `vector_read_addr`, `vector_val`, and `vector_row` are used to read data from the vector memory. When your module is in the input_loaded state, then the `vector_read_addr` signal will select an address in the vector memory, and the memory's output will be provided on `vector_val` and `vector_row` (indicating the values and rows for our sparse vector—see the explanation of CSC in Project Overview Section 3.3).

- `D` is an output that indicates the *D* value of the vector currently stored in the vector memory. (Recall from Project Overview Section 3.3 that *D* indicates the number of non-zero values in a sparse vector.) We require the `D` output to be correct and valid anytime `input_loaded` is 1.

- The `done` signal is an input that tells your module when it is done computing the MSpVM on the matrix and sparse vector values stored in memory. When `done` is asserted on a positive clock edge, your module should set `input_loaded` to 0 and go back to its initial state to begin taking in new input values.

Input Protocol

Matrix and sparse vector inputs will be provided to this module via the AXI-Stream interface signals shown on the left of Figure 3.1. Recall the discussion of AXI-Stream from Project Overview Section 5. Here there is some complexity, so we will break this description down into three parts: (a.) loading a matrix, (b.) loading a vector, and (c.) reusing an old matrix.

(a.) Loading a Matrix

When loading the matrix, your system will use the `AXIS_TVALID`, `AXIS_TREADY`, and `AXIS_TDATA` signals. If `AXIS_TVALID` and `AXIS_TREADY` are both equal to 1 on a positive clock edge, then your system has received data. Your system will control the value of `AXIS_TREADY`, so you must set it to 0 when the system is not "ready" for new inputs.

Matrix values will be provided in row-major order. This means that the first row will be provided, then the second row, etc. That is, the first valid input will be `W[0][0]`, then `W[0][1]`, …. Eventually after the end of the row (`W[0][N-1]`), then it will go to the next row `W[1][0]`, and so on. Your module must store the matrix values in the internal matrix memory in this order.

This process is illustrated in Figure 3.2 for a 3x3 matrix. Note that in this example, `AXIS_TVALID` and `AXIS_TREADY` are both 1, so the timing is simple. However, if either of these signals were to become 0, then nothing would be transmitted at that time, and the system must stall.

(Please also note in this figure we include a signal called `new_matrix`. This signal will be defined and explain in part (c.) below.)

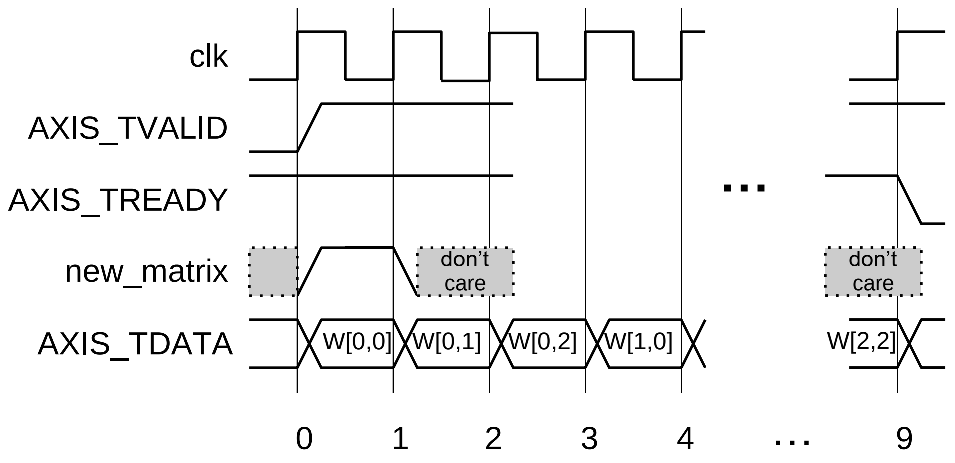

Figure 3.2. Example of loading the matrix.

(b.) Loading a Vector

After the matrix is transmitted, then the input stream will transfer the input vector. Loading the vector over the AXI-Stream interface is slightly more complex for two reasons:

1. We know that our vector will be stored in CSC format (see Project Overview Section 3.3.) This means that in addition to the `values` of the vector, we will also need the `row` indices. To transmit the `row` indices, we will use some of the bits from the `AXIS_TUSER` signal.

   We will define `AXIS_TUSER` as being `$clog2(N)+1` bits wide.
   - Its least significant bit, `AXIS_TUSER[0]` will be used for a different purpose. (We will discuss this in (c.) below.)
   - The remaining bits, `AXIS_TUSER[$clog2(N):1]` will be used to hold the `row` encoding of the element of the sparse vector. In the rest of this specification, we will refer to these bits as `row`.

   For simplicity, I suggest you make the following assignment in your SystemVerilog description of this module:
   ```
   logic [$clog2(N)-1:0] row;
   assign row = AXIS_TUSER[$clog2(N):1];
   ```

2. We also need to deal with the fact that the length of the compressed vector will be variable. That is, if *D* is 1, then only one input element (a pair of `val` and `row`) will stream into the system. However, if *D* is 100, then 100 input elements will stream in. To deal with the variable length problem, we will use the `AXIS_TLAST` signal. When loading the vector, if `AXIS_TLAST` is 1 on any positive clock edge (where `AXIS_TVALID` and `AXIS_TREADY` are both 1), then this indicates that this is the *last* value of the vector.

   For example, Figure 3.3 shows an example of first loading a 3x3 matrix, then a vector (where *D*=2). The hardware system only knows the vector is done when it sees the `AXIS_TLAST` signal.

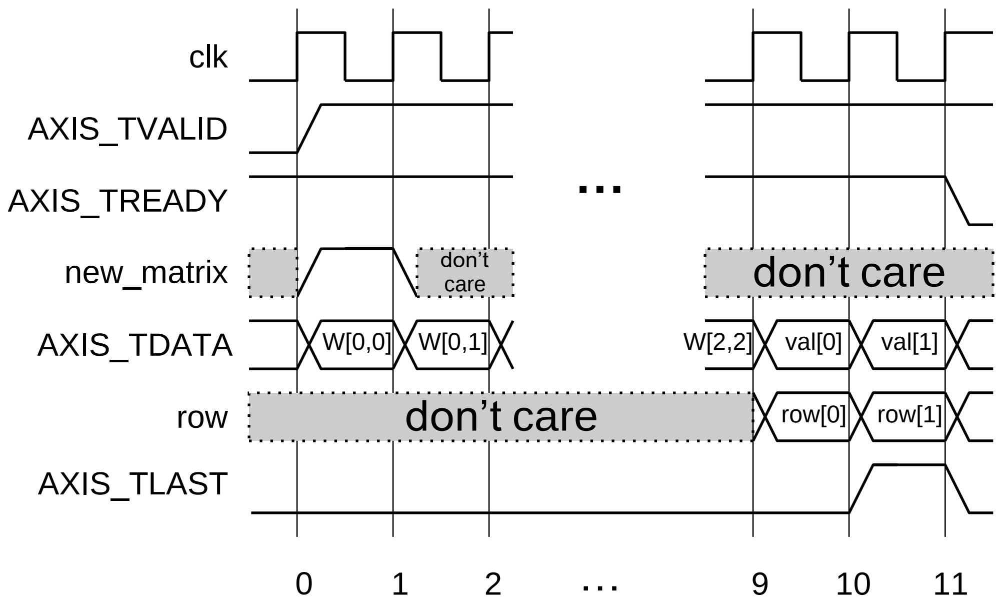

Figure 3.3. Example of loading a new 3x3 matrix, followed by a vector with *D*=2.

(c.) Reusing an Old Matrix

Lastly, your system has the capability of *remembering* the previous matrix, rather than loading a new matrix. This can allow operations where the system loads a matrix, and then multiplies that matrix with many different vectors without having to reload the matrix values as inputs. (Since the matrix can be large, this can save a lot of time.)

To make this possible, we need to define a control bit as part of the input stream to tell the system when it is loading a new matrix and when it should use the old one. For this we will use `AXIS_TUSER[0]`, the least significant bit of `AXIS_TUSER`. We will call this signal `new_matrix`.

For simplicity, you may want to make the following assignment in your SystemVerilog description of this module:

```
logic new_matrix;
assign new_matrix = AXIS_TUSER[0];
```

The `new_matrix` signal only matters during the first cycle of input data transfer. That is, as your system begins processing inputs for a new MSpVM, it will check `new_matrix` on the first valid input cycle. If `new_matrix==1` at that time, then the data represents the first value of the matrix. If `new_matrix==0` at that time, then the data represents the first value of the vector, and the system will continue to use the *old* matrix already stored in memory from the last operation.

The `new_matrix` signal is only meaningful during the first cycle of data transfer; at all other times, its value is ignored. Figure 3.3 above shows the process when `new_matrix==1`, so the input data represents a matrix and then eventually a vector. Then Figure 3.4 below shows an example where `new_matrix==0`, so the system skips reading the matrix and immediately reads a vector.

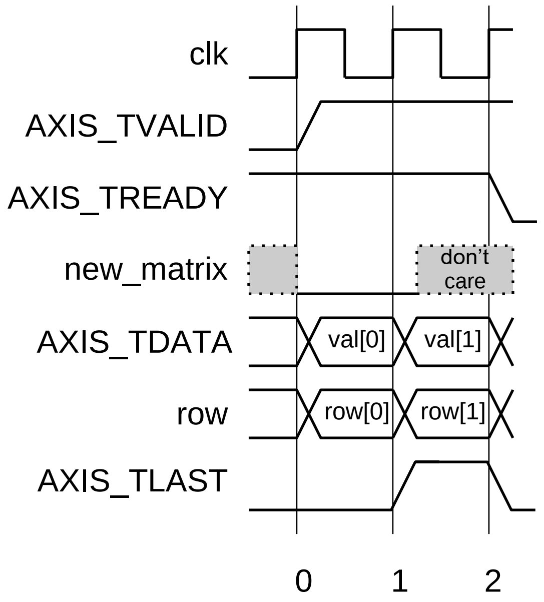

Figure 3.4. Example where `new_matrix==0` at the time of the first input, so a new matrix is not loaded, and the vector immediately begins.

Keep in mind that all transfers are done using the AXI-Stream protocol. This means that `AXIS_TDATA` and `AXIS_TUSER` are only transferred on positive clock edges when `AXIS_TVALID` and `AXIS_TREADY` are both equal to 1. Your system will set the value of `AXIS_TREADY`, so you must set it to 0 when the system is not "ready" for new inputs.

Module Operation

Here we will give a brief overview of this module's phases of operation and the sequence of steps it will take. You must construct control logic that will make the system undergo the following steps. Your control logic will likely need to include an FSM and counters. The following phases are not necessarily the only FSM states your system will use, but you should use them as a guide for how your control logic for this module should operate.

1. *Input Matrix*: In this phase, the module will take in matrix data via its AXI-Stream input interface. This data will be stored in the matrix memory. This phase is complete after the entire matrix is loaded. This step will be skipped if `new_matrix` is 0. (See "Input Protocol" subsection above.)

2. *Input Vector*: In this phase, the module will take in sparse vector data via its AXI-Stream input interface. The data elements of the vector will be provided on `AXIS_TDATA` and the `AXIS_TUSER` signals, as described in the "Input Protocol" subsection above. Your system will monitor the `AXIS_TLAST` signal to know when to exit this phase.

3. *Input Loaded*: The input_loaded phase begins after the input vector is complete. Your system will output `input_loaded=1` during this phase. In this phase, your system should ensure that the `D` output signal holds the correct value of *D* (the number of non-zero elements in this vector).

   During this phase, your system will allow the vector and memory read interfaces (on the right side of Figure 3.1) to read data from the vector and matrix memories. This means that during this phase, the `vector_read_addr` and `matrix_read_addr` signals should be used to provide addresses to the vector and matrix memories, respectively.

While in this phase, your system should monitor the `done` input signal. When `done==1` on a positive clock edge, your system should exit this phase, set `input_loaded` to 0, and go back to the beginning, waiting for new input data.

Memory

This module should contain two memory instances: one to hold the matrix and one to hold the encoded vector. Each memory must use synchronous reads and have a single address port. Use the following module for both memories. You can find this memory at:

`/home/home4/pmilder/ese507/proj/part3/memory.sv`

```
module memory #(
        parameter                   WIDTH=16, SIZE=64,
        localparam                  LOGSIZE=$clog2(SIZE)
    )(
        input [WIDTH-1:0]           data_in,
        output logic [WIDTH-1:0]    data_out,
        input [LOGSIZE-1:0]         addr,
        input                       clk, wr_en
    );

    logic [SIZE-1:0][WIDTH-1:0] mem;

    always_ff @(posedge clk) begin
        data_out <= mem[addr];
        if (wr_en)
            mem[addr] <= data_in;
    end
endmodule
```

There are several important things to understand about this memory:

- The memory has one read port, one write port, and one address input used for both reads and writes. All reads and writes are synchronous (that is, they will occur on a positive clock edge).

- Unlike the memory used in Part 2, this memory module only contains a single address input, which will be used for both reading and writing. (So, you cannot read and write to different locations of this memory at the same time.)

- The memory's parameters are the same as in the dual port memory in Part 2.

The timing of reading and writing from this memory is the same as the dual port memory from Part 2 (Figures 2.2 and 2.3), except now there is a single address signal `addr` that is shared for both reads and writes.

Recall from class that this RTL memory module will not synthesize to SRAM—instead it will result in a memory structure built out of flip-flops. If these memories were large, this would be very inefficient. However, the memories you will need in this project will be fairly small, so we will simply let the logic synthesis tool implement them using registers.

- The matrix memory should have `WIDTH=INW` and `SIZE=M*N`. Your system should store the matrix in row-major order (see above).
  - The `data_in` port of this memory should connect directly to the `AXIS_TDATA` input of this module.
  - The `data_out` port of this memory should connect directly to the `matrix_data` output of this module.
  - The `addr` and `wr_en` ports of this memory should be driven by your module's internal logic.

- The vector memory should have `WIDTH=INW+$clog2(N)` and `SIZE=N`. In each memory entry, you should store the concatenation of the vector element's `val` (which is `INW` bits) and `row` (which is `$clog2(N)` bits). Each vector will need *D* entries, and *D* can be as large as *N*, so we must make this memory of size *N* to accommodate the largest possible vector.
  - The `data_in` port of this memory should be driven by a concatenation of the `AXIS_TDATA` input and the `row` signal.
  - The `data_out` port of this memory should connect directly to the `vector_data` and `vector_row` outputs of this module (with bits partitioned to match the way the value and row were concatenated on the memory's input port).
  - The `addr` and `wr_en` ports of this memory should be driven by your module's internal logic.

Reading from the memories

Once your module is in the input_loaded phase (and it is setting `input_loaded` to 1), your control logic should allow the memory read interfaces (on the right side of Figure 3.1) to read data from the matrix and vector memories. That is, when `input_loaded==1`, then the matrix memory's address input should follow `matrix_read_addr`, and the vector memory's address input should follow `vector_read_addr`. At all other times (when `input_loaded` is 0), your internal logic will determine the addresses on these memories.

Number of Non-Zero Vector Entries D

The number of non-zero values in the sparse vector is not known ahead of time—it is input-dependent. As described in the Input Protocol subsection above, your module will use the `AXIS_TLAST` signal to determine when you have reached the last of the input vector entries. Your module should contain a register to hold the value of *D*, and this register's output should go to the `D` output of this module. When your system determines that the entire input vector has been stored, it should set the correct value of *D* and hold it until new input data is loaded.

Code

Store all of your files for part3 in a subdirectory called `part3/` . Implement this design in a file called `input_mems.sv`. Your system should use the memory module given above. (You may either copy/paste that into your code or include the given `memory.sv` file alongside your file.) Use the following top-level module name, port names, and port declarations:

```
module input_mems #(
        parameter INW=16,
        parameter M=24,
        parameter N=32,
        localparam LOGN = $clog2(N),
        localparam LOGMN = $clog2(M*N)
    )(
        input clk, reset,

        input [INW-1:0] AXIS_TDATA,
        input           AXIS_TVALID,
        input           AXIS_TLAST,
        input [LOGN:0]  AXIS_TUSER,
        output logic    AXIS_TREADY,

        output logic            input_loaded,
        input                   done,
        output logic [LOGN-1:0] D,

        input        [LOGN-1:0] vector_read_addr,
        output logic [INW-1:0]   vector_val,
        output logic [LOGN-1:0]  vector_row,
        input        [LOGMN-1:0] matrix_read_addr,
        output logic [INW-1:0]   matrix_data
    );
```

Testbench

You are provided with a testbench to test your `input_mems` module. This testbench will generate random data (matrices and sparse vectors) and then provide the inputs to your module using the AXI-Stream interface (with random timing on the `AXIS_TVALID` signal). Then the testbench will wait until your module asserts the `input_loaded` signal, and then it will read the data back from your module's internal memories, checking that the data retrieved matches what was transmitted. You can find the testbench in the following two files.

```
/home/home4/proj/part3/inputs_mems_tb.sv
/home/home4/proj/part3/test_helper.c
```

Copy these files into your `part3/` work directory where your part 3 design is. Before you use it, please read the following brief explanation of how the testbench works. Then read through the testbench files and read the comments.

The testbench module includes five parameters, which will look familiar after your experience with the previous testbenches.

- `INW`: the number of bits for each data word
- `M`: the number of rows in the matrix
- `N`: the number of columns in the matrix
- `TESTS`: the number of inputs to simulate
- `TVALID_PROB`: a decimal value that represents the probability that the testbench will assert `AXIS_TVALID` on any given cycle. This should be between 0.001 and 1, inclusive. For

example, if you set `TVALID_PROB` to 0.3, then there will be a 30% chance that the testbench will assert `AXIS_TVALID` (and try to transmit input data) on each cycle. If you set this parameter to 0, the testbench will randomly pick a probability at the beginning of the simulation (and tell you what it chose).

The testbench uses a SystemVerilog class called `testdata`. This class is somewhat different than the one in Part 1's testbench. Here, the class holds the input data for an entire MSpVM operation. That is, it holds the values of the matrix, the vector (in sparse encoding), the value of *D*, and the value of `new_matrix`. The testbench uses SystemVerilog's internal randomization capabilities to randomize the values of the matrix, *D*, and `new_matrix`. Then, it uses a C function (called through DPI) to generate a sparse random vector.

For each test, the testbench will randomize the test data and feed the test data into the system via the AXI-Stream interface. The timing of this transaction will be randomized based on `TVALID_PROB`.

After feeding in a set of test data, the testbench will wait for the DUT to set `input_loaded` to 1. Then, the testbench will use the DUT's `vector_read_addr` and `matrix_read_addr` inputs to read the stored data back from the DUT's internal memories, checking the values.

Compile and simulate your code, using a variety of different parameters. Don't forget that you can set top-level testbench parameters from the command like with `-G` like `-G INW=32`. (For more examples, see the description of the Part 1 testbench.)

Report and Code Submission

After implementing and simulating the designs with a variety of parameters, complete the following tasks. In your report, provide the requested information and answer the following questions.

1. This part of the project required you to design a significant amount of control logic that interacts with the AXI-Stream interface, the memories, the *D* register, and the `input_ready` and `done` signals. Carefully and thoroughly document this module including your control logic. Your documentation should allow the reader to fully understand how your `input_mems` module works (and any submodules)without looking at the code.

2. The number of cycles required by this module is largely determined by the parameters (`M`, `N`), the input vector's `D`, and by how the testbench asserts `AXIS_TVALID`. However, there are places where you as the designer could make choices that affect the number of cycles required by your module. For example, if your system unnecessarily sets `AXIS_TREADY` to 0, or it adds extra cycles of delay between steps, the system will be less efficient.

   One way to quantify this is to measure how long your system takes to complete a task. Run a simulation where you set `INW=16`, `M=24`, `N=32`, and `TVALID_PROB=1`. When `new_matrix==1`, the testbench will begin by feeding in a 24*32=768 matrix values and *D* vector elements (where $1 \leq D \leq 32$). In other tests where `new_matrix==0`, the system will simply feed in *D* vector elements.

   Simulate this design in QuestaSim's waveform view and count the number of cycles between when your design sets `AXIS_TREADY` to 1, and when it sets `input_ready` to 1.

Do this for a set of inputs where `new_matrix==1` and a set of inputs where `new_matrix==0`.)

Hint: you can view the amount of simulated time that passes in the waveform and then divide it by 10ns to get the number of simulated clock cycles. In your report, give this cycle count, and the value of *D* for the vector (for both `new_matrix` of 0 and 1).

In the report, quantify how efficient your system is with respect to the number of clock cycles by computing the following ratios:

- When `new_matrix==1`, use efficiency = cycles/(*M***N*+*D*)
- When `new_matrix==0`, use efficiency = cycles/*D*

In these metrics, 1.0 is perfectly efficient—a good implementation will be close to this. Report both metrics and the data you collected.

If your efficiency number is not close to 1, where could your logic be improved?

3. In the previous question, you measured the cycle count. Now, use your understanding of your system's behavior to write equations for the cycle count with respect to `M`, `N`, and `D`. You should have one equation for `new_matrix==1`, and one equation for `new_matrix==0`.

4. Describe how your system's hardware changes when you change the parameters `INW`, `M` and `N`. Be specific about how the hardware components in your design will change as you change these parameters. Which of these changes do you expect to be important to the area and power of the system? Explain your answers.

5. Synthesize the design using DesignCompiler twice, with parameters:
   - `INW`=16, `M`=4, `N`=5
   - `INW`=16, `M`=24, `N`=32

   Correct any synthesis problems you find. For each set of parameters, find the maximum possible clock frequency, area, power, and critical path location. (Here you only need to report statistics for the highest clock period for each design.) Carefully explain each design's critical path. Don't just list its start and end points; explain what the path means and why it makes sense. Does the critical path change between these two designs? Explain why or why not.

   Save both of your synthesis reports with descriptive names and include them with your submission.