Simple Pipeline

EEL 4930/5934 – Reconfigurable Computing

Objective:

In this lab, you will be implementing a simple pipelined circuit. The circuit will utilize one 32-bit block RAM to continually provide 4 8-bit inputs per cycle, and one 17-bit block RAM to store an output each cycle. In software, you will initially transfer data from the microprocessor into the input block RAM, specify the size of the input, start the circuit, and then wait for completion, at which point the software will read data from the output blockRAM and output it to the screen.

Important: make sure to start with the provided code template given on the lab website.

Pseudocode:

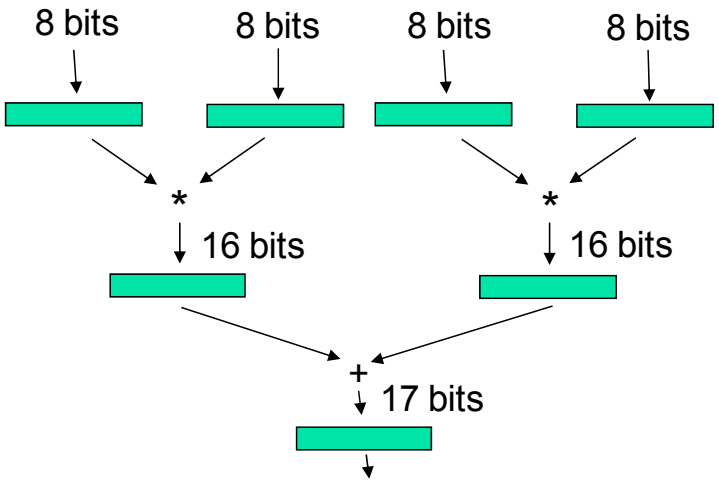
The pseudo-code you will be implementing on the FPGA is shown below. Note that your actual code will look nothing like this. This pseudocode is simply intended to help you understand the functionality.

```
for(i=0,j=0; j < OUTPUT_SIZE; i += 4, j++) {

  a[j] = b[i]*b[i+1] + b[i+2]*b[i+3];
}
```

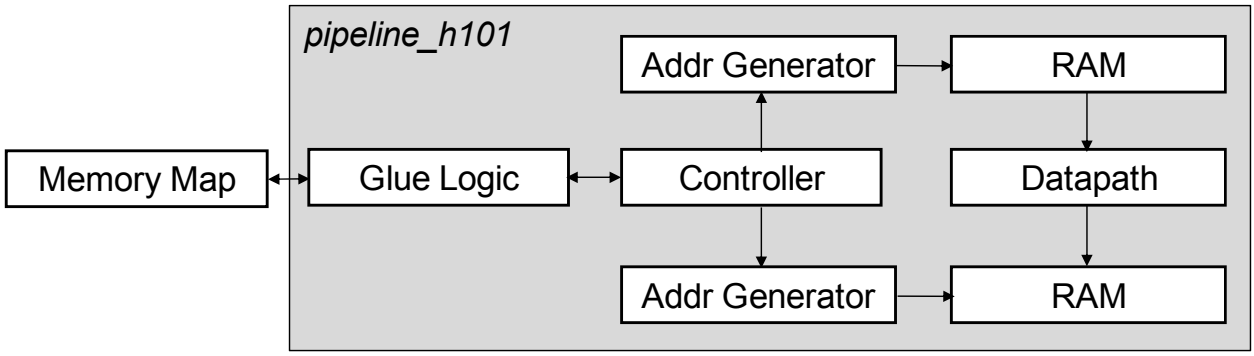
Datapath:

The datapath for your circuit implementation of the pseudocode should match the following schematic:



As shown, the datapath has 4 8-bit inputs which connect to two multipliers followed by an adder. The green boxes are all registers of the specified widths. You can create the datapath entity however you want, but it must synthesize to the exact structure shown. Therefore, I would recommend a structural description of registers, multipliers, and an adder. The multipliers should generate a 16-bit output and the adder should produce a 17-bit output to prevent overflow.

Overall circuit:



The overall structure of the circuit is shown above. You will use a Nallatech memory map and glue logic similar to the previous labs. However, for this lab, the glue logic also handles transferring data to and from block RAMs. This code is already provided for you. However, you must create all other components.

The controller works similarly to previous labs. It will initially wait for a go signal to be asserted from software via the memory map. When go is asserted, the controller will read from a size register (not shown) that is also specified from software using the memory map, and will then enable the input address generator which will produce the corresponding number of addresses needed to produce the input stream of the specified size from the top RAM. The controller will also enable the output address generator used for storing outputs from the datapath into the bottom RAM.

The address generators in this lab are essentially counters that count from a specified starting address (in this case, from address 0) for the specified number of addresses. The address generator's output (i.e., the current address) should connect to each RAM. The address generator will likely also include control signals for reading or writing to RAM, although there are numerous different implementations.

The datapath should be implemented as shown earlier. However, each register in the datapath has an enable signal (not shown) that controls stalls. There are numerous ways to control these enable signals. One potential way is to add a control signal from the controller that enables the datapath at the appropriate time. With this approach, the controller would start the input address generator, wait a certain number of cycles, then enable the datapath, and then enable the output address generator at the exact cycle that would store the first datapath output. Although this approach works for this lab, it does not work well for more complex circuits where latencies are not known or where latencies change. A more flexible way of enabling the datapath is to include an extra flip-flop at each level of the pipeline that represents whether or not the data at that level is valid. With this approach, you wouldn't ever stall the datapath, but would instead change the output address generator to ignore outputs where the valid bit is 0. Either approach is acceptable for this lab.

The RAMs could potentially be implemented using the Nallatech block RAMs from earlier labs, but instead, this lab demonstrates how to write VHDL that the synthesis tool will infer as a RAM. The advantage of creating your own block RAMs is that you can make them any width and size that you want. The disadvantage is that the Nallatech block RAMs already handle

communication with software. To enable communication with your custom block RAMs, I have included code in the glue logic entity that demonstrates this functionality. Although you don't need to write any code for the block RAMs in this lab (other than the control signals), I encourage you to understand the provided code because you will need it for future labs.

Software:

To communicate with the custom circuit, you will again need software that transfers data to the input block RAM, sets the input size, sets the go signal, waits for the done signal, and then reads the outputs from the output block RAM. For this lab, I have provided all software code to demonstrate how to communicate with the block RAMs through the memory map. Make sure you understand the provided code.

IMPORTANT: Do not change the software code! The project will be graded with a script that expects the output in the current format.

Summary of steps:

1) Create the VHDL for the datapath and top-level pipeline_h101 entity using the provided template.
2) Simulate your VHDL with the provided testbench (tb.vhd). This testbench mimics the functionality of the memory map, so your simulation should be very similar to execution on the FPGA. Make sure there are no errors before trying your code on the board.
3) Create a Dimetalk project with the basic PCI-X edge, clocks module, and memory map.
4) Import your VHDL, connect it to the memory map as shown, and generate a bitfile.
5) Run the provided C code and verify that the circuit works correctly.

Turn in all vhdl files, your Dimetalk DT3 file, and the generated bitfile. There are no changes to the C code so you do not need to submit it.

Final Instructions

You only need one submission per group. Also, include a readme.txt that names all group members and explains the status of the lab (e.g., unresolved problems, tool bugs, etc.).