ECSE 487
COMPUTER ARCHITECTURE LAB

ASSIGNMENT 1:
MULTI MODE BARREL SHIFTER

By:
Simon Foucher
260 223 197
Simon.foucher@mail.mcgill.ca

Monday, Jan 26th 2009

2.0 Lab description

Design a multi-mode barrel shifter. The unit has 3 inputs: IN, N and S, and has a single output: OUT. The shifter accepts an array of 8 bits in parallel through IN and shifts it by any number of bits between 0 and 7, encoded in binary in the input N. The shifted vector is outputted in OUT.

The shifter has 4 modes of operation, encoded by the 3rd input S as follows:

- "00": Rotate shift right
- "01": Shift Logical Left
- "10": Shift Logical Right
- "11": Shift Arithmetic Right

The design has to be done in 3 different approaches:

- Behavioral: the VHDL describes the behavior of the data manipulation
- Structural: the design is separated between data buses which carry the data from the input to the output, and control lines which select how the data is manipulated along its path. In this type of design, every connection is explicitly described at the gate level.
- Pipelined: The structural design is split up into data flow stages, and registers are placed between each stage. Like an assembly line, the pipelined shifter can work on multiple operations at once. This design is optimized for maximum clock speed, since once loaded, it will produce one operation per clock pulse.

3.1. Behavioral Shifter

I) Description of the methodology followed.
[file: /VHDL/Q1_BEHAVIORAL.vhd]

This design was made using a single process and 2 case statements. The first case statement reads the S input and detects the mode of operation. From there, the second case statement reads the value in the N input. Based on those 2 cases, we use a simple concatenation statement to produce the shift.

When in barrel shift mode, we concatenate the input with itself. When in logical shift, we concatenate the input with the required amount of 0 and when in arithmetic shift, we concatenate the input with one or many times it's most significant bit.

II) Documented VHDL source code for all entities.

/VHDL/Q1_BEHAVIORAL.vhd

III) Simulation results (traces) to show that your design operates correctly.

Using the macro found here, we tested a few critical cases to assess the quality of the shifter.
/MACRO/MACRO_BEHAV.TXT
And here is the simulated result:

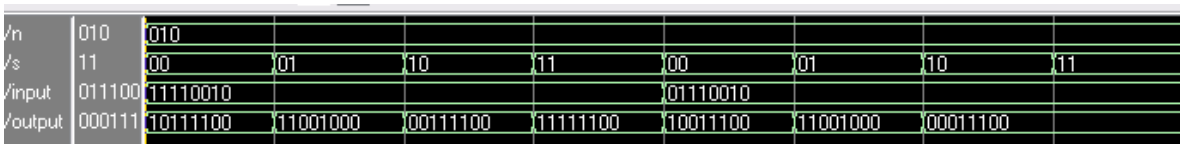

FIGURE 3.1.1: Wave pattern when running /MACRO/MACRO_BEHAV.TXT

We performed a shift by 2 bits in all 4 modes of operation with the most significant bit set to 0 (first series of 4 tests) then set to 1 (second set of 4). This was only to tweak any functional errors. A more exhaustive test was performed with the testbench.

IV) Testbench

/VHDL/TESTBENCH_BEHAVIORAL.vhd
Notes:

- Ensure to include VHDL ’93 Language Syntax before compiling for shift operators libraries
- Best viewed with 4 spaces tab indents (for aligned code)
- With the selected clock speed, 0.33mS of simulation is sufficient to run through all the cases (run 0.33ms)
- A more efficient version of this testbench was developed for the pipelined shifter (see 3.3.IV)

The testbench operated using a finite state machine with 5 state named with what function they perform and the S code associated with this function (ex: SLL_10 = Shift Logical Left, S <= “10”). After a reset, the process starts in mode ROR_00 with every bits of the test signals N_s and INPUT_s set to 0.

INPUT_s is 9 bits long (one more than the required input for the DUT). The 8 least significant bits are fed as an input to the DUT and then INPUT_s in incremented by 1 at every clock pulse. The most significant bit is used as a flag to notify when all the possible cases have been tested. In other words, if we start by “000000000” and increment by 1 at every iterations, once we reach “100000000”, every single 8 bit combinations have been exposed to the DUT by the 8 LSBs of the vector. This condition is detected by the statement “while INPUT_s < 256”.

The same strategy is used for the N_s vector, 4 bits long, of which the 3 LSBs are fed as an N input to the DUT. N_s is initialized at “0000” and every time we exhaust a list of 256 INPUT_s vectors, N_s is incremented by one. Once we reach N = “1000” or N = 8, we have exhausted every required values of N (0 to 7) for the DUT and the Finite State Machine transitions to the next state.

At any moment of the testing, the 8 LSBs of INPUT_s are manually shifted by N_s places by the VHDL Code, and the result is compared to the output of the DUT. If a divergence is detected, ERR is set to 1 to notify of the error mode.

After all 4 states have been fully tested, the FSM enters its “finished” state and ERR is set to 1 (the signal is recycled as a ‘done’ signal; combined with the FSM’s ‘finished’ state).

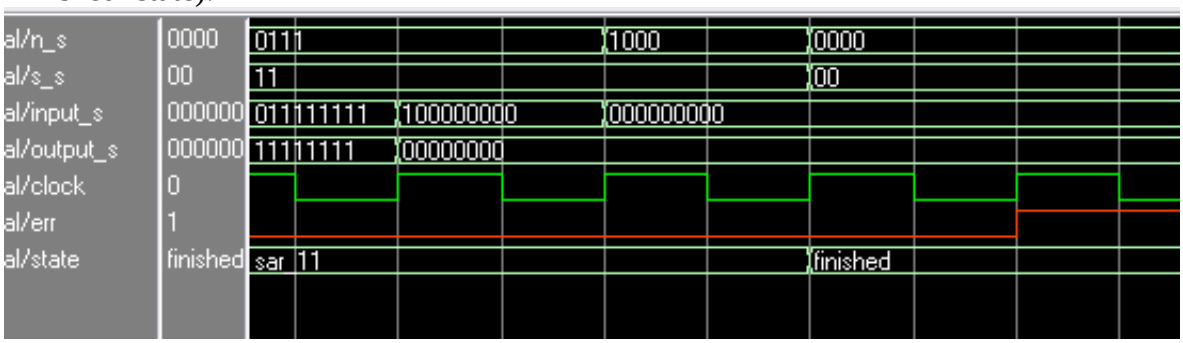

FIGURE 3.1.2: ERR (in red) is asserted and state is set to ‘finished’ when the list of all possible inputs ih exhausted.

V) Synthesis for maximum speed and minimum area.
VI) Summary of resources and performance achieved in term of throughput and latency.

3.2. Structural Shifter

I)Description of the methodology followed

/SCEMATICS/STRUCTURAL_DRAWING.pdf : High resolution schematic
/VHDL/Q1_STRUCTURAL.vhd : Main device
/VHDL/MUX4.vhd and MUX8.vhd: Components used

The main design strategies used was to develop the structural design in data flow stages, to facilitate the conversion into a pipelined version in question 2. We also tried to create components for repeated code (mostly muxes).

Every stage is composed of an 8 bit wide bus Multiplexes with a single select line, designed separately as a component (MUX8.vhd). The barrel shifter was initially split into 3 stages. Stage one (controlled by the LSB of N) selects between the input ant the input shifted by one place. Stage 2 selects between the input from stage 1 and this input shifted by 2 places, and is controlled by the central bit of N. Stage 3, controlled by the MSB of N selects between the output of stage 2 and this output shifted by 4 places.

By doing so, every combinations of the 3 bits of N can select between any combinations of 1+2+4, or shift anywhere from 0 to 7 inclusively.

It might be more intuitive to do the reverse order of stages (i.e. have the first stage connected to the MSB of N and shift by 4), but the design was based on the following site and works just as good either way:
http://tams-www.informatik.uni-hamburg.de/applets/hades/webdemos/10-gates/60-barrel/shifter8.html

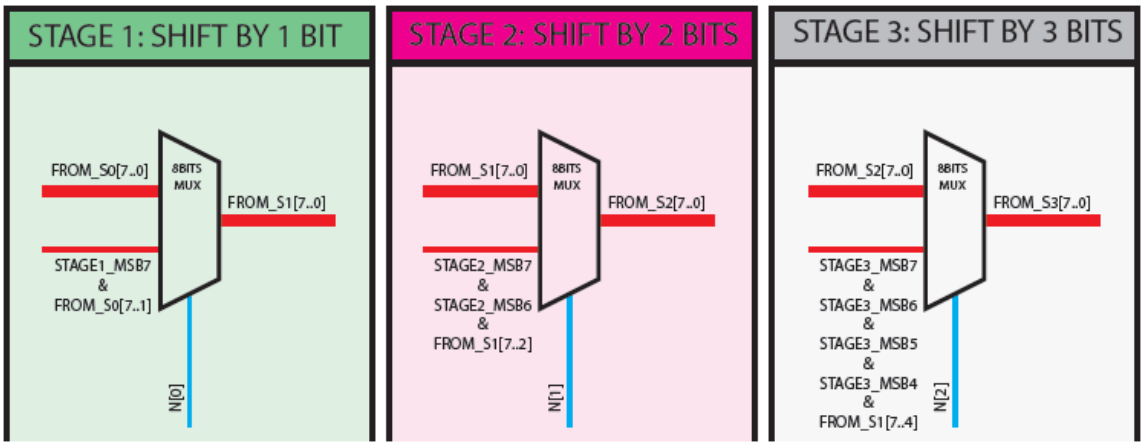

FIGURE 3.2.1: main stages of the structural shifter (red = Data Path, Blue = control lines)

To simplify the logic, the barrel shifter is only capable of shifting right (since this covers ¾ of required operations). In order to accommodate for a SLL, both the input and the outputs can be reversed simultaneously in 2 additional stages (stage 0 and stage 4). The muxes used in those stages are controlled by a signal which gets asserted only when shifting left (INVERT_BITS <= NOT S(1) AND S(0)).

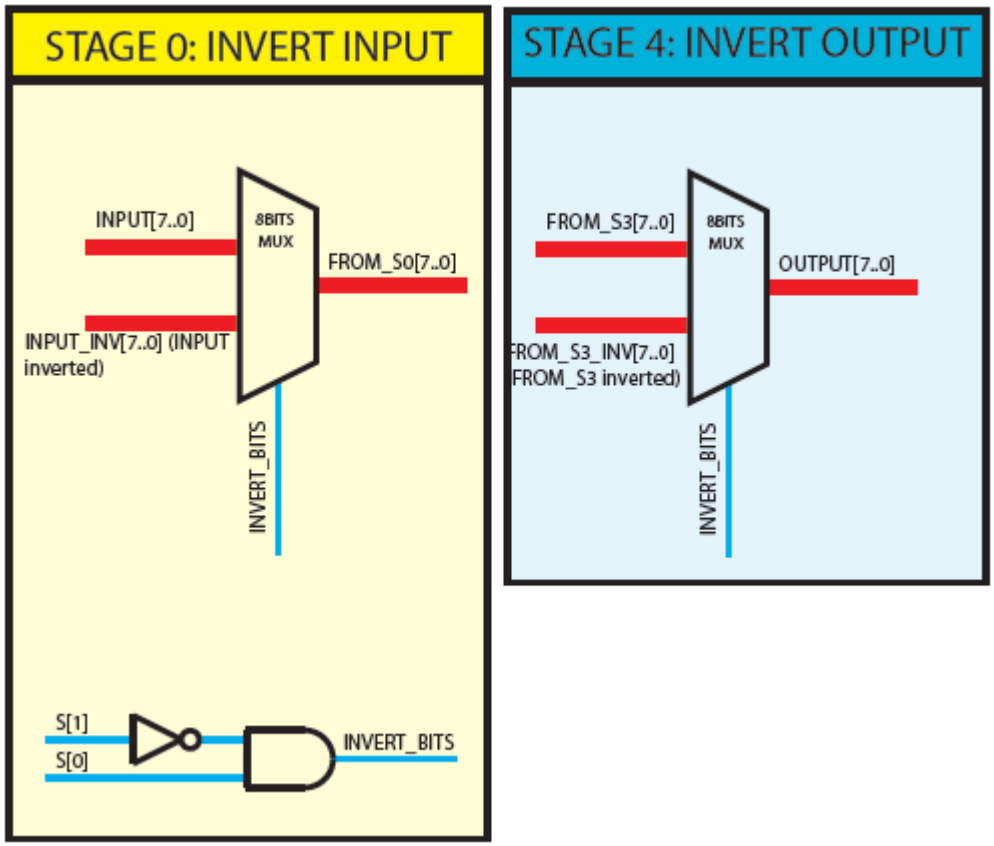

FIGURE 3.2.2: Added stages 0 and 4 to invert the bits when performing a left shift with a right barrel shifter (red = Data Path, Blue = control lines)

The most significant bit(s) fed to the shifted input passes through a special purpose multiplexer (MUX4.vhd). Using S (mode of operation) as a control line, this mux selects between the LSB(s) of the previous stage when in barrel mode, '0' when on shift Logical Mode, and the MSB of the previous stage when shifting in arithmetic mode.

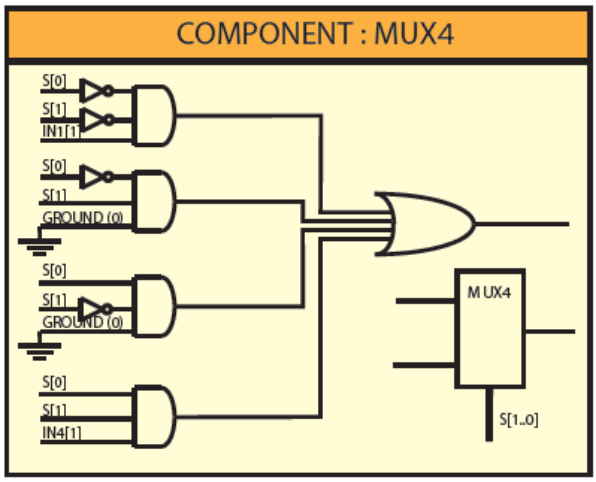

FIGURE 3.2.3: Special purpose mux to select the MSB feed when shifting

The stage 1 (shifts 1 place) has one of these units, stage 2 (shifts 2 places) has 2 and stage 3 (shifts 4 places) has 4.

II) Documented VHDL source code for all entities.
/VHDL/Q1_STRUCTURAL.vhd : Main device
/VHDL/MUX4.vhd: Component used
/VHDL/MUX8.vhd: Component used

III) Simulation results (traces) to show that your design operates correctly.

Since the most delicate part of this design was a correct data path between the input and the output, before running the testbench, we used a macro to check the proper functioning of the shifting multiplexers. The main focus was to test every combination of N. In order to do so, individual bits of the N vector were forced into clocks at different frequencies to naturally oscillate in a 3 bit binary count.
/MACROS/MACRO_STRUCT.TXT (Note: every mode of operation is the macro is meant to be used on SEPARATE simulations)

We can observe the proper amounts of bits being shifted (in yellow), as well as a look at the shifted output of every stages (even is not selected)

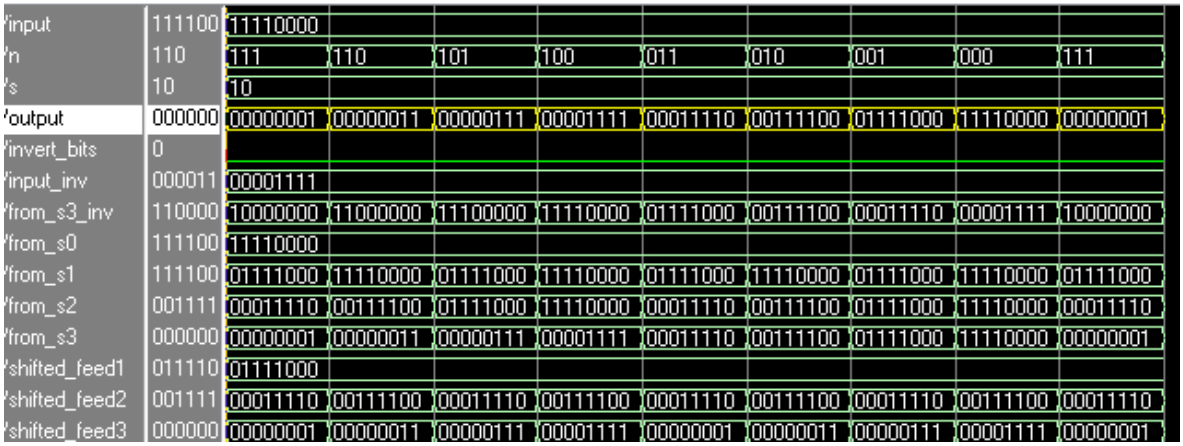

FIGURE 3.2.4: Shifting logical right, result observed in yellow.

IV) Testbench
/VHDL/TESTBENCH_STRUCTURAL.vhd

The testbench used is identical to the one used for the behavioral tests, with the replacement of the DUT (see 3.1. IV for details)

V) Synthesis for maximum speed and minimum area.

VI) Summary of resources and performance achieved in term of throughput and latency.

3.3. Pipelined Shifter

I) Description of the methodology followed (including any figures that help document your design)
/SCEMATICS/PIPELINED_DRAWING.pdf : High resolution schematic
/VHDL/Q2_PIPELINED.vhd : Main device
/VHDL/MUX4.vhd and MUX8.vhd: Component Multiplexers used
/VHDL/DFF_8BITS.vhd: a regular 8 bit wide bus DFF
/VHDL/DFF_VAR_DELAY.vhd: the main timing controller

Since the structural design was made with pipelining in mind, only a few upgrades from that model were required to enable pipelining.

The datapath was modified by inserting D flip flop between each stage to capture the data after the last stage has processed it.

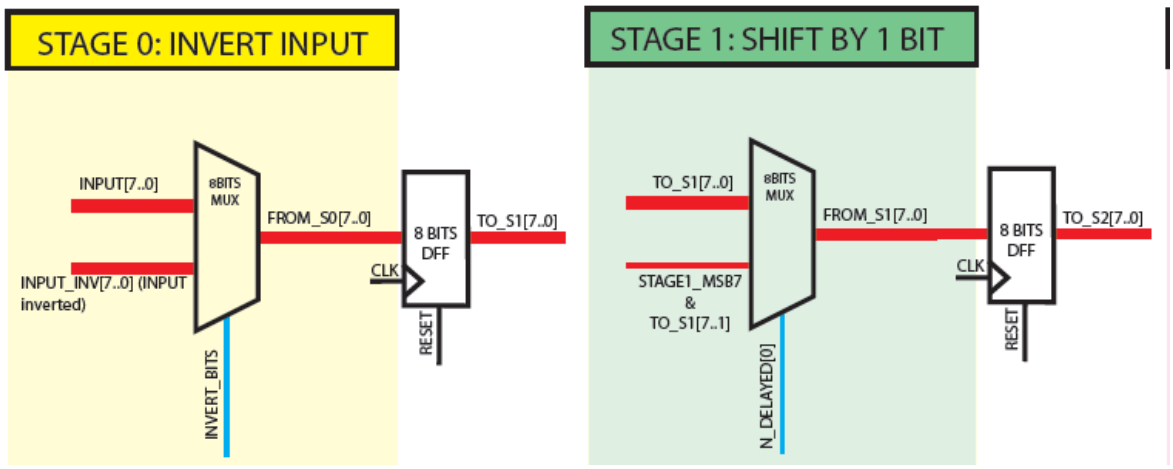

Figure 3.3.1: D Flip Flops were added between each Staged of data manipulations. Here we can observe 2 of them after Stage 0 and Stage 1.

Because of the pipelining, different stages would be executing different instructions simultaneously, so it was necessary to design a timing controller. The controller was encapsulated in a component to facilitate the potential for future upgrades of the shifter. (called: DFF_VAR_DELAY.vhd)

The controller inputs N and S and outputs them at different delays. The delays for the S signals match the Stage's names. (Stage 0 takes S_WAIT_1CLK, Stage 3 takes S_WAIT_3CLK, etc…). There was no absolute necessity to output S_WAIT_0CLK, as it is the input directly fed back as an output, but is was implemented in this matter such that only the controller was "allowed" to control the data. (i.e. instead of connecting both S into the controller and into Stage 0, S in only fed to the controller and the controller manages all the Stages). As for N, all 3 bits are delayed by 1, 2 and 3 clock cycle to arrive on time to stages 1, 2 and 3 (Stages 0 and 4 do not need N). Further down the controller, a single bit from each level of delay is selected and recombined in a vector called "N_DELAYED" which mimics the original N vector which can be reconnected the same way as N was connected in the non Pipelined design. It was not necessary to age all 3 bits by 3 clock cycles (the bare minimum was 1 clock pulse for all 3 lines, 2 clock pulses for the lines going to Stages 2 and 3, and 3 clock pulses only for the bit going to Stage 3). It has been implemented this way for easier understanding and could be changed to reduce the hardware consumption if necessary).

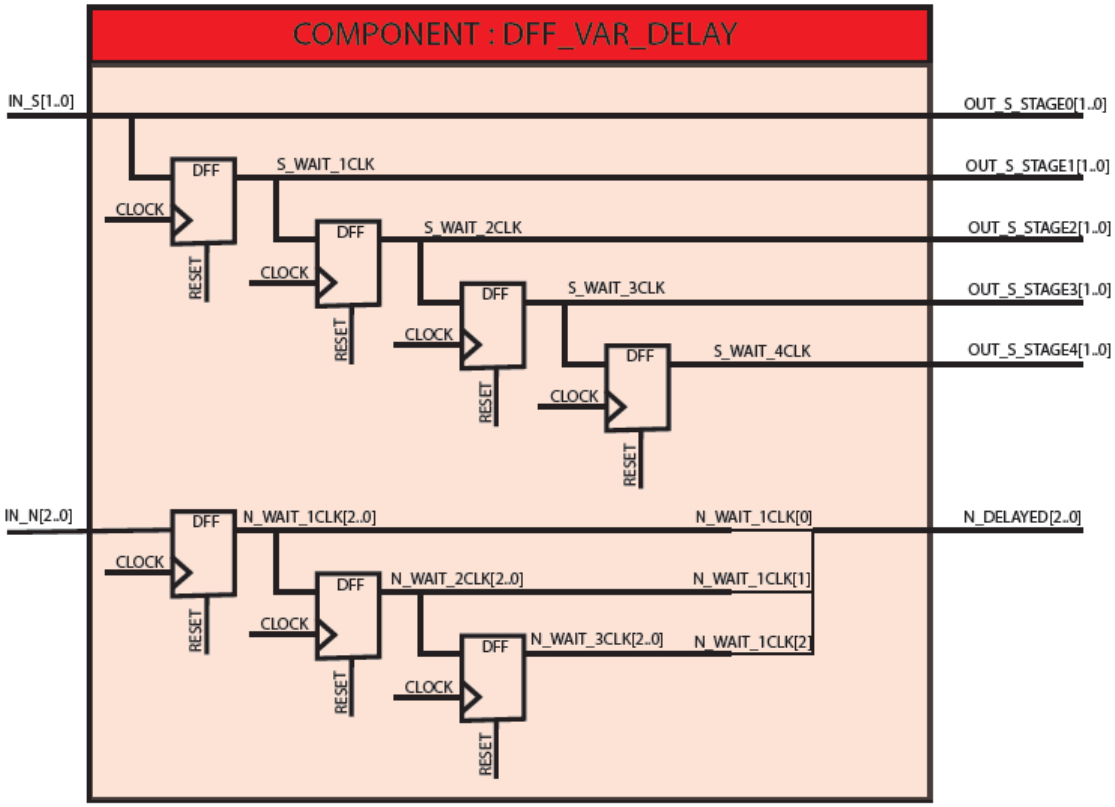

Figure 3.3.2: The controller inputs S and outputs it at different delays for every stages. It also receives N and delays the signals to arrive on time at the muxes.

The input S and S got connected to this controller, and the output of the controller was connected to the control lines of the circuit.

The only other modification made was to add a second AND to stage 4 in order to interpret a reverse signal if needed. To save hardware, we could also have captured the INVERT_S0 bit and age it 4 clock pulses before sending it to stage 5, but this approach was easier and faster to implement.

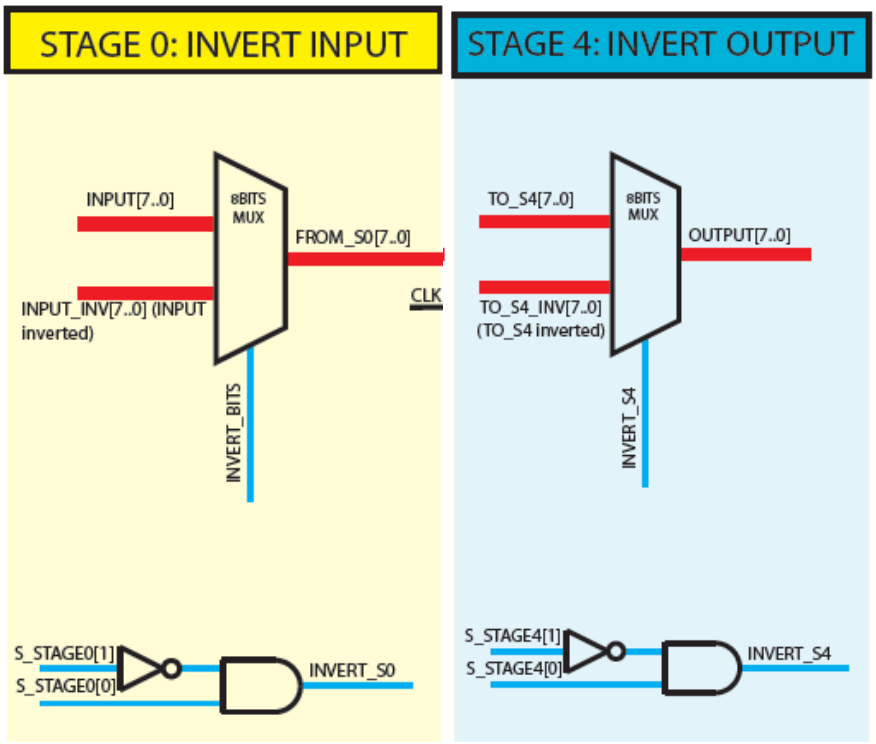

Figure 3.3.3: The inverting select hardware was duplicated and the S signal got aged. A better approach would have been to age the invert signal and reuse it in Stage 4. This would have saved an inverter, an AND gate and the aging flipFlopc could be a single bit wide instead of 2 bits for the S signal.

II) Documented VHDL source code for all entities.

VHDL/Q2_PIPELINED.vhd : Main device
/VHDL/MUX4.vhd and MUX8.vhd: Component Multiplexers used
/VHDL/DFF_8BITS.vhd: a regular 8 bit wide bus DFF
/VHDL/DFF_VAR_DELAY.vhd: the main timing controller

III) Simulation results (traces) to show that your design operates correctly.

Before running a full testbench, the following macro was used to make sure the timing was accurate.
/MACROS/MACRO_PIPELINED.txt

The first set of tests performs a left shift, which requires the use of Stage 0, 1 and 4. In the following wave result, we can observe the result trickling down the data pats as required.

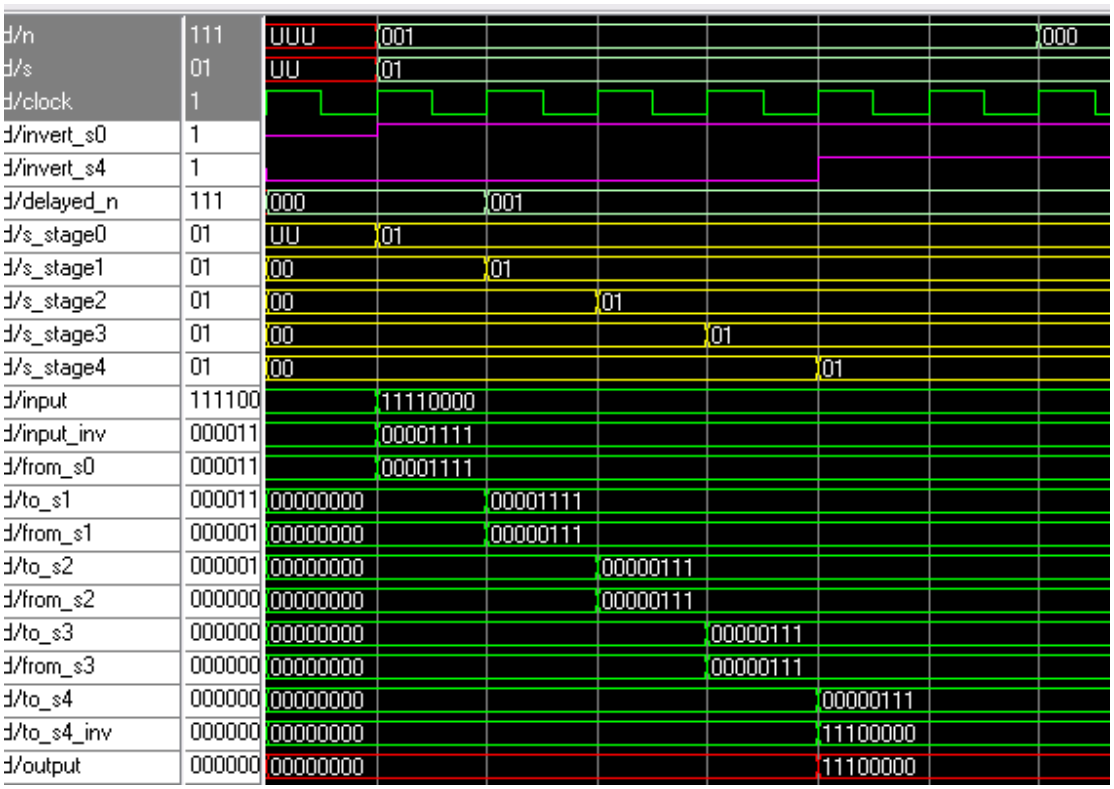

Figure 3.3.4: The top signals shows us a shift by N=0 in mode SLL S=01. We can observe the proper functioning of the S delay trickling down the controller as the clock ticks in yellow. We can also observe the data getting manipulated in every stage in green, to form the required output in red.

The macro was slightly modified to check the cases N = “100”, N = “010” and N = “001” to observe every shifting multiplexers operate individually.

The second thing this Macro was design to observe is the proper functioning of pipelined instructions. Here we can observe the same string of bits get shifted by 0 to 7 spaces. We can see the loading of the device and the proper output appear, delayed by 5 clock pulses from the input.

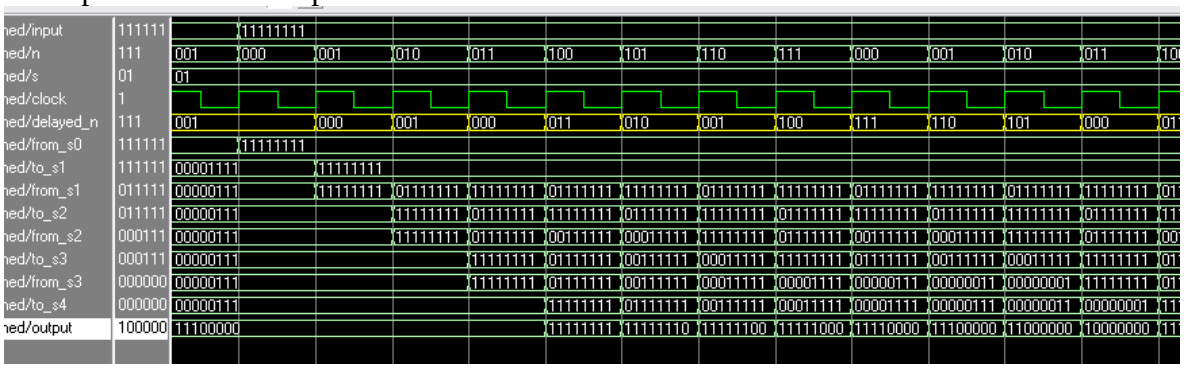

Figure 3.3.5: Pipelined instructions appearing with a 5 clock pulse delay (We count 4, but model sim does not account for the gate delay of the last stage, so in reality, this would take 5 pulses).

We can also ovserve the “de-loading” of the device and the final output appearing 5 pulses (4 + invisible logic) after the final input was received.

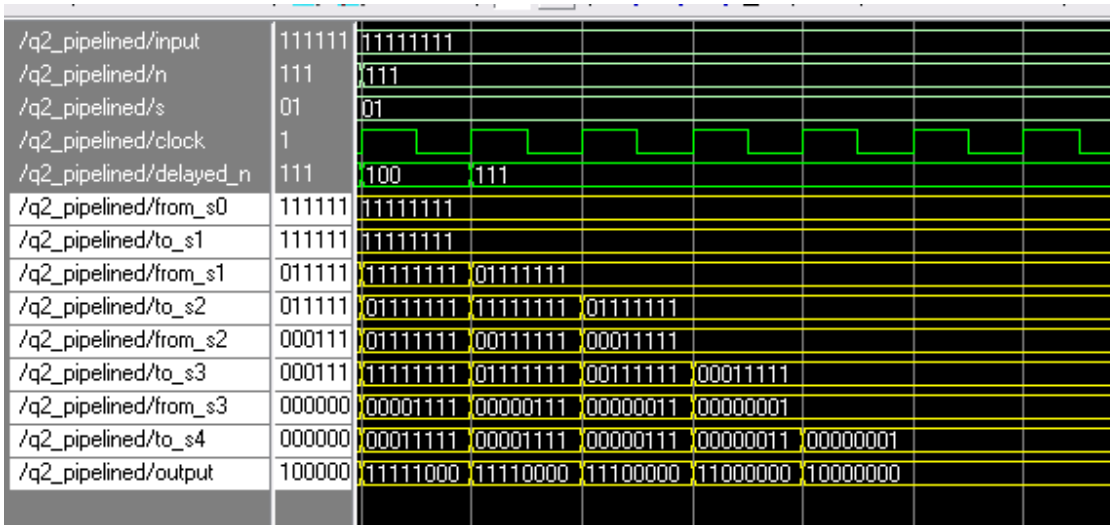

Figure 3.3.6: "DeLoading" of the pipelined shifter.

IV) Testbench

/VHDL/TESTBENCH_PIPELINED.vhd

The testbench code was greatly modified from the previous ones to simplify its logic. A single test vector was used to synthesize every case. The vector is bits long. The 8 LSBs are interpreted as the 8 BIT input for the DUT, the next 3 bits are N and bits 11 and 12 are S. The MSB is used as a STOP bit. The test vector is reset to 0 at the start if the simulation and incremented by 1. It scans every input possibilities in mode "00" and with N = "000", then N = "001", etc… until N = "111", after which, the input and N goes back to 0 and mode S = "01". As soon as we reach the vector "1000000000000", all possibilities have been tested as the binary count was increased.

TEST_VECTOR <= "STOP BIT" & "SS" & "NNN" & "IIIIIIII"

In order to perform the tests, the input vector was aged 4 clock cycles and the aged version was compared to the device's output. The tests were performed using an "if" statement reading the test vector bits representing S, and shifting the aged input bits by the aged N.

One more addition to the testbench was to let it run an extra 4 clock pulses after all possibilities had been tried to give the DUT time to "unload" its content. In this case also, the ERR bit get asserted to signal the end of the simulation.

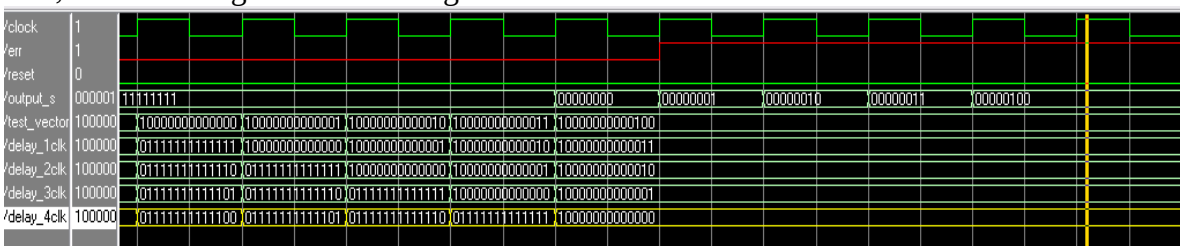

Figure 3.3.7: Note that ERR (in red) is asserted when all the cases have been tested and that the final input vector had time to age 4 clock pulses (Delay_4CLK = "1000000000000") Note: the outputs

observed afterwards are residuals from the fact that as the last value of the test vector gets aged, new inputs are still fed into the device, causing these delayed outputs.

V) Synthesis for maximum speed and minimum area.

VI) Summary of resources and performance achieved in term of throughput and latency.

VII) Discussion of how well your design meets its speedup objective.

4. Comment on the differences and usefulness between behavioral and structural descriptions.

Suggest when each type of description is useful.

A behavioral description is advantageous for complex logic performing basic tasks. The programmer can the easily specify the behavior of the data and let the compiler figure out how to implement it in hardware.

A Structural approach is much more precise and is preferred when certain specific objectives are followed (other than just performing a task, the programmer might want to perform it a specific way). It allows for better logic manipulation and can help design pipelined units as we have done here. Structural descriptions enable device specific optimization either for timing, throughput or minimal area.

Comment on the advantages/drawbacks of having both types of descriptions in VHDL

Advantages

Behavioral: higher level of abstraction good for algorithms. RTL level synthesis

Drawbacks:
Confusing for new programmers
Xtra libraries/complexity for the compiler
Mixed design might come up messed up

Are there situations where both descriptions might be useful?