Table of Contents

0. General Information

The zip file I have sent contains this document and some other folders. There are five folders as follows:

1. Verilog Code: This folder contains the Verilog code for the designs.
2. Test benches: This folder contains the test benches written to test just the program counter and the shifter. I have not included the test benches for adders, registers, counters, register files, CLOG2 function and others. Those are tested for the previous homework.
3. Do-TCL Files: This folder contains the scripts for running test benches as batch jobs.
4. Synthesis Results: This folder contains the ".syr" files of the shifter, program counter and program counter test circuit for the FPGA.
5. Bigger Schematics: This folder contains larger versions of the images used within this document. If the reader wants to get a better look at the designs or the simulation outputs, they can refer to here.

In general I have tried to use as much parameterization as possible. I also tried to follow the XST specification closely.

1. Primitives Implemented

In this section I will explain the design of some of the primitives that were used to implement the shifter the program counter and the FPGA implementation of the program counter.

1.1. Register Behavioral (Parameterized)

This is a parameterized behavioral register. For more details refer to the previous homework. This will be used in the program counter design for holding the current value of the instruction address.

1.2. Register File Behavioral (Parameterized)

This is a parameterized behavioral register file. For more details refer to the previous homework. This is used in the program counter design for Jump and Jump-and-Link type instructions.

1.3. Sign-Extender (Parameterized)

This is a parameterized signed extender circuit. It will extend any input of CURSIZE to an output of EXTSIZE. It also has a MODE parameter, which does unsigned extension when set to 1. By default, when it is set to 0, the circuit works as a signed extension circuit.

1.4. Signed Adder Behavioral (Parameterized)

This is a signed adder based on XST specification. For the purpose of the program counter we don't need a fully functional ALU. Although later it is possible to use the same ALU for program counter calculation and for arithmetic and logic operations of the CPU, in this design we are going to use only a simple adder.

1.5. Clock Slower (Parameterized)

This is basically an n-bit counter that outputs 1 when all bits are equal to 0. This means for example if we select n=19, every 2^{19} clock cycles the output will be 1 once. It is used in this project for two purposes, for slowing the clock for debouncer circuits, and for the seven segment display wrapper. The outputs of these clocks are used as clock enable for other components. They are not used as gated clocks.

1.6. Debouncer Circuit

This one is basically a shift register of 19 bits, that outputs one if the contents of the register is all 1's.

1.7. Synchronous Parallel Load, Asynchronous Reset Counter (Parameterized)

This one is a simple up counter that counts up by 1 at every clock cycle. It also has a load input that you can use to parallel load the counter. You can also reset it asynchronously.

1.8. Binary to Seven Segment Display

This one takes a 4 bit binary input and gives a 7 bit output that can light each part within the seven segment display. The inputs are active low.

1.9. Seven Segment Display Wrapper

This one is a wrapper that multiplexes between each of the 4 seven segment displays to be able to show the contents of the program counter on the 4 seven segment displays. The implementation is done according to the homework description. It refreshes each display around 60 times per second.

2. Program Counter Implementation

Here I will first explain the design of the simple program counter. Then I will show the simulation results.

2.1. Design

The design of the program counter is based directly on the homework description. We have basically a 16-bit register for holding the current instruction address. An Adder is used to calculate the next instruction address value. The first input to the adder is the current instruction register value. The second input to the adder is the output of a multiplexer which

chooses between the constant ‘1’ and a sign extended displacement value. For jump instruction we also have another multiplexer that chooses between the output of the adder and the value coming from a register file. For jump and link instructions it is also possible to push the result of the adder to the register file. Additional details about the design are as follows:

- The value within the program counter is unsigned.
- Branch and jump instructions are assumed to not result in negative results. The displacement can be negative but the case where the current value of the PC added to the displacement results in a negative value is not handled. It is left to the compiler to care for this.
- Both the instruction register and the Adder are 16-bits. But since the adder does signed addition, currently only 15 bits of the register can be used. It is possible to remedy this by extending the adder by one bit, but for now I stick to the description given within the homework assignment.
- Everything works correctly on normal operations which are
    - +1 up count that doesn’t result in overflow.
    - Branch instructions that don’t result in negative instruction address value.
    - Jump/Jump-And-Link type instructions.

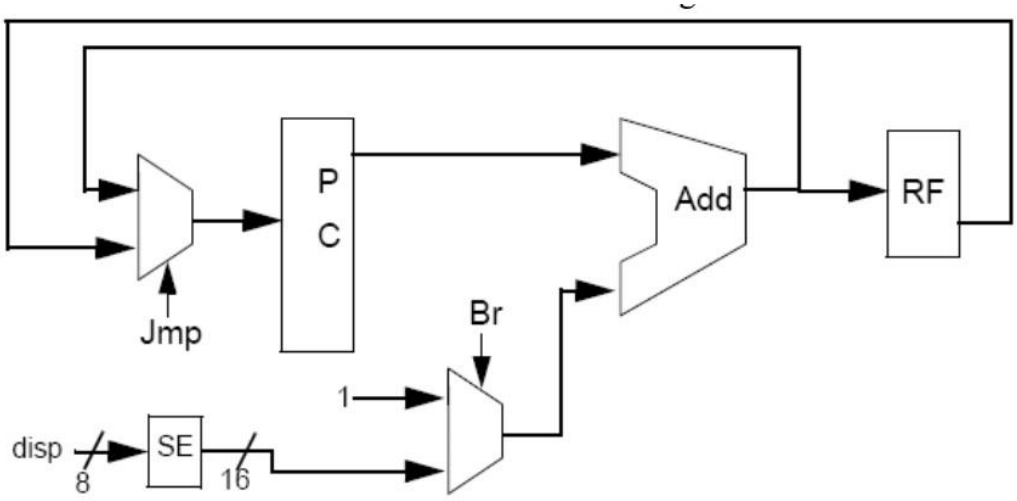


Figure 1. Program counter (PC) architecture.

2.2. Simulation

Since showing the waveform of this would take too much space, I am going to use the “display” function to print out the results after read and write operations to specific cells of the register file. At the beginning of the outputs I have also used “$time” function to show the current simulation time. First I assert reset. Then I test branching (short jumps) with positive (15) and negative (-13) values. The numbers are randomly selected. Then at time 305 I write the next PC

value to a random register in the register file. Since I didn't include an external write option for the register file, this is one way of setting one of the registers to a value other than zero. At time 355 I test jumping by using the register value that I have written at time step 305. And lastly at 445 I test jump and link operation. Jumping is done to the same location written at 305. To test whether linking is done correctly after some time I jump to the location referred by the register I have used to write during the JAL operation.

Simulator is doing circuit initialization process.
0 Simulation is started.
Finished circuit initialization process.
5 Current PC = 0
15 Reset is Asserted
15 Current PC = 0
25 Current PC = 1
35 Current PC = 2
45 Current PC = 3
55 Current PC = 4
65 Current PC = 5
75 Current PC = 6
85 Current PC = 7
95 Current PC = 8
105 Current PC = 9
115 Current PC = 10
125 Current PC = 11
135 Current PC = 12
145 Testing branching with positive value (15).
145 Current PC = 13
155 Current PC = 28
165 Current PC = 29
175 Current PC = 30
185 Current PC = 31
195 Current PC = 32
205 Current PC = 33
215 Current PC = 34
225 Current PC = 35
235 Current PC = 36
245 Current PC = 37
255 Current PC = 38
265 Current PC = 39
275 Testing branching with negative value.(-13)
275 Current PC = 40
285 Current PC = 27
295 Current PC = 28

305 Filling Register 0110 with value 29 + 1.
305 Current PC = 29
315 Current PC = 30
325 Current PC = 31
335 Current PC = 32
345 Current PC = 33
355 Testing jumping to value at Register 0110.
355 Current PC = 34
365 Current PC = 30
375 Current PC = 31
385 Current PC = 32
395 Current PC = 33
405 Current PC = 34
415 Current PC = 35
425 Current PC = 36
435 Current PC = 37
445 Filling Register 0100 with value 38 + 1.
445 Testing Jumping to value at Register 0110 at the same time. (JAL OP)
445 Current PC = 38
455 Current PC = 30
465 Current PC = 31
475 Current PC = 32
485 Current PC = 33
495 Testing jumping to value at Register 0100 which should be previous PC value + 1.
495 Current PC = 34
505 Current PC = 39
515 Current PC = 40
525 Current PC = 41
535 Current PC = 42
545 Current PC = 43
555 Current PC = 44
565 Current PC = 45
575 Current PC = 46
585 Current PC = 47
595 Current PC = 48
605 End of Simulation.

As can be seen from the simulation trace, the PC works correctly as described. For the following home works it is possible that the calculation of the PC will be changed. Depending on the decisions on the CPU, I will update the PC design. (Single-cycle vs. multi-cycle, pipelined or not)

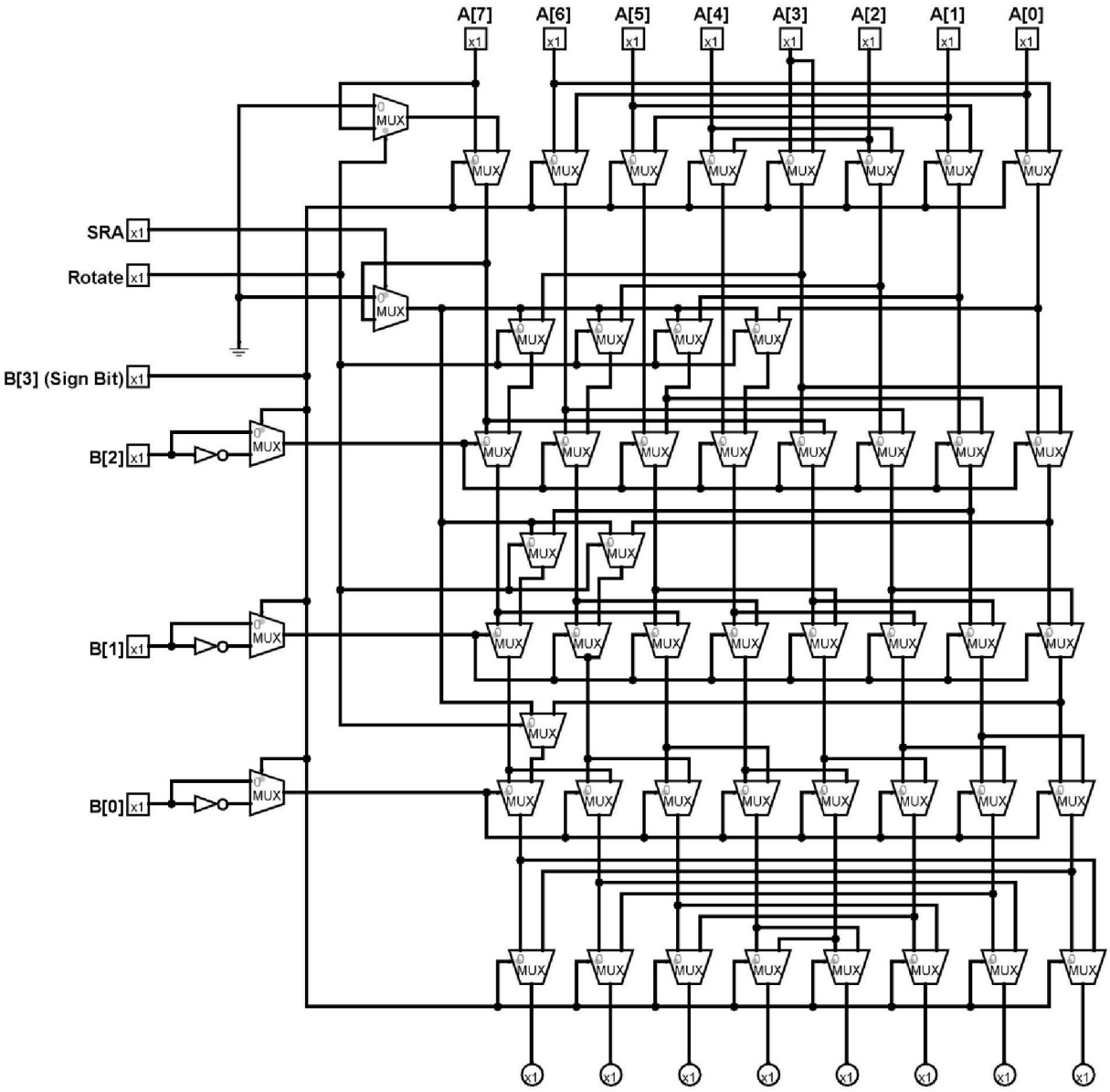
3. Logarithmic Barrel Shifter Implementation

The Logarithmic Barrel Shifter I have implemented is 16-bit. Its inputs are a 5 bit signed shift amount which enables shifts between -16 to 15, single bit rotate control which works for both left and right shifts and another single bit arithmetic shift control which only works for right shift. By default positive shift amount is assumed a right shift. A negative shift amount is assumed a left shift. 2's complement signed numbers representation is used for this purpose.

3.1. Design

The design is based on the combination of Figure 3 and Figure 4 (a copy of these can be found within the Bigger Schematics folder) on Matthew Pillmeier's work [1]. Drawing a 16-bit shifter would take too much space, so I have shown 8-bit version. My basic implementation is the following extended to 16 bits. First I started with the design in the first figure (Figure 3) and then added the data reversal multiplexers in front and end of the circuit. Note that the second figure (Figure 4) uses another control bit called "left". Since our circuit has to work on 2's complement shift amount, this design is not sufficient for us. To solve the problem I have used another multiplexer to reverse the lower 4 bits of the shift amount input if the highest bit of our shift amount is 1 (which means negative, which means left shift). But since 2's complement needs to add 1 after negating the bits, I had to modify the first data traversal multiplexers, such that in case "left=1", not only a reversal takes place, but also a single bit shift. Also since I had accommodated the rotate function within my design, the shifted bit in this first data reversal multiplexer group depends on the rotate input.

Basically the resulting circuit for 8-bits would look as follows (A larger Figure is included in the Bigger Schematics folder):



3.2. Simulation

Here I have tested each function with a couple of different inputs that check all the possible combinations. I have selected a random shift amount 6 for testing the middle values. For positive shift values I have tested the limits (which are 0 and 15) and for negative values I have also tested the limit (which is -16) in addition to some random values like -6 and -15. Both rotate and logical shift are tested at each value for two different inputs called “A”. The outputs are denoted as “Y”. For right shift I have also tested the arithmetic shift operation. The trace of the simulation is as follows:

0 Shift Right By 6.
20 A: 0000111111010110 == Y: 0000000000111111

40 A: 1100111111010110 == Y: 0000001100111111
40 Shift Arithmetic Right By 6.
60 A: 0000111111010110 == Y: 0000000000111111
80 A: 1100111111010110 == Y: 1111111100111111
80 Rotate Right By 6.
100 A: 0000111111010110 == Y: 0101100000111111
120 A: 1100111111010110 == Y: 0101101100111111
120 Shift Right By 15.
140 A: 0000111111010110 == Y: 0000000000000000
160 A: 1100111111010110 == Y: 0000000000000001
160 Shift Arithmetic Right By 15.
180 A: 0000111111010110 == Y: 0000000000000000
200 A: 1100111111010110 == Y: 1111111111111111
200 Rotate Right By 15.
220 A: 0000111111010110 == Y: 0001111110101100
240 A: 1100111111010110 == Y: 1001111110101101
240 Shift Right By 0.
260 A: 0000111111010110 == Y: 0000111111010110
280 A: 1100111111010110 == Y: 1100111111010110
280 Shift Arithmetic Right By 0.
300 A: 0000111111010110 == Y: 0000111111010110
320 A: 1100111111010110 == Y: 1100111111010110
320 Rotate Right By 0.
340 A: 0000111111010110 == Y: 0000111111010110
360 A: 1100111111010110 == Y: 1100111111010110
360 Shift Left By 6.
380 A: 0000111111010110 == Y: 1111010110000000
400 A: 1100111111010110 == Y: 1111010110000000
400 Rotate Left By 6.
420 A: 0000111111010110 == Y: 1111010110000011
440 A: 1100111111010110 == Y: 1111010110110011
440 Shift Left By 15.
460 A: 0000111111010110 == Y: 0000000000000000
480 A: 1100111111010110 == Y: 0000000000000000
480 Rotate Left By 15.
500 A: 0000111111010110 == Y: 0000011111101011
520 A: 1100111111010110 == Y: 0110011111101011
520 Shift Left By 16.
540 A: 0000111111010110 == Y: 0000000000000000
560 A: 1100111111010110 == Y: 0000000000000000
560 Rotate Left By 16.
580 A: 0000111111010110 == Y: 0000111111010110
600 A: 1100111111010110 == Y: 1100111111010110

Here we see that all the results are correct.

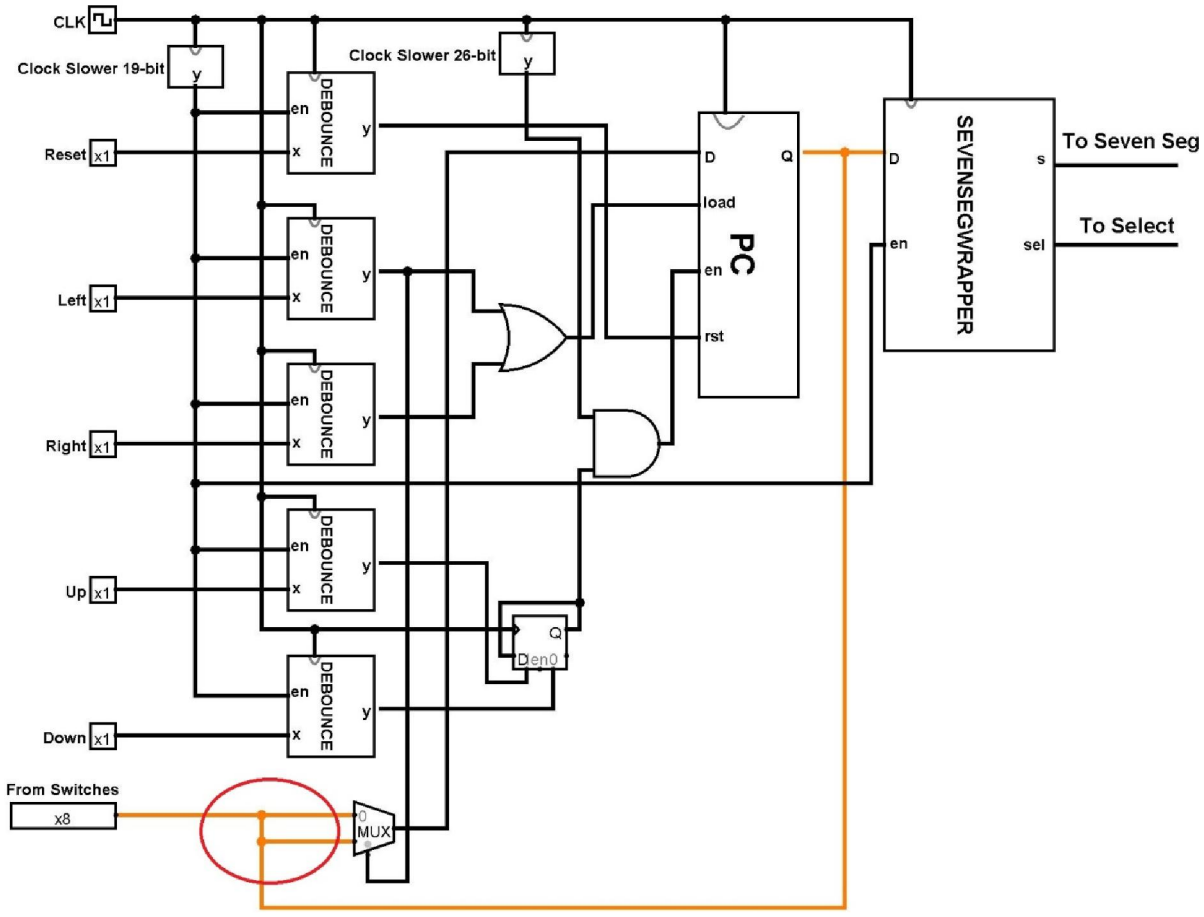
4. Program Counter Test Circuit for the FPGA

In this part I am going to quickly cover the design of the program counter test circuit. It mainly consists of an up counter with parallel load capability. All the five inputs are debounced using the debouncer circuits described in the first section. The test circuit can do the following.

- Middle Button => Resets the counter to 0.
- Left Button => Loads the upper 8 bits of the counter according the 8 switch-positions on the board.
- Right Button => Loads the lower 8 bits of the counter according the 8 switch-positions on the board.

- Up Button => starts the counting. If it already is started, does nothing.
- Down Button => stops the counting. If it is already stopped, does nothing.

I have used two main slowed clocks. The first one is for debouncers and seven segment wrapper. These are divided by 19. The second one is for the counter, which uses a 26 bit clock divider, because we want to see the circuit count up by 1 nearly every second and our clock source is 100 MHz. I have circled a multiplexer area with red in the following figure. Technically this part is wrong, but trying to display those connections bit by bit would take too much space, so I will instead draw it wrong here but explain what actually is done. Basically the select input of that multiplexer comes from the debounced left button. If the left button is pressed the 16-bit multiplexer gives the output as {switch[7:0], Q[7:0]}. If the left button is not pressed we give the output of the multiplexer as {Q[15:8], switch[7:0]}. (switch[7:0] is the input coming from the 8 switches on the FPGA board)This means that the multiplexer gives output as though the right button is pressed. But this doesn't result in any problems because the load input of the program counter is 0 when neither the left nor the right buttons are pressed. But note that in my design if you press both the left and right buttons on the FPGA at the same time, the left button takes precedence since it is the input controlling the multiplexer.



5. Synthesized Simulations

The synthesized models work just as expected. We compare the results of the simulation with the results from the previous two chapters. Inside the "Verilog Code" folder I have included an additional folder called "Post-Synthesis" which contains the post synthesis simulation models of the barrel shifter and the program counter. The only thing to do is after changing the unit under test inside the test bench file, the user needs to add 100 ns wait for global reset (glbl module) to finish. After that the post synthesis simulation model works as expected.

6. Synthesis Outputs

	Barrel Shifter	**Program Counter**
Advanced HDL Synthesis	• 100 x 1 bit 2to1 MUX • 1 x 16 bit 2to1 MUX	• 272 x FF • 3 x 16 bit 2to1 MUX • 240 x 1 bit 2to1 MUX • 16 x 1 bit 16to1 MUX
Number of Slice Registers	0	272
Number of Slice LUTs	92	353
Number used as Logic	92	353
Number of I/O	39	38
Total Delay	12.463 ns (5.502 ns logic)	3.788 ns (default period)

- Since the barrel shifter can handle many different operations (it is practically bloated), I had to add additional levels of logic to it. Therefore as expected it has high delay.
- The barrel shifter currently takes less space, but to decrease delay we may want to divide its functionality and create parallel blocks that do different things (right shift and left shift can be done in parallel and the output can be selected by a single multiplexer etc.) This would increase the used area but decrease the path delay.
- As expected the number of registers used in the barrel shifter is 0 since this is just a combinational system. The number of slice register used in the program counter is exactly equal to the number required by our design which is 272. 256 registers are used for the register file that we use and 16 for the instruction register.

References:

1. Design Alternatives for Barrel Shifters, by Pillmeier. http://www.princeton.edu/~rblee/ELE572Papers/Fall04Readings/Shifter_Schulte.pdf
2. BOUN - EE 540 - Spring 2015 - Assignment 3
3. RTL Hardware Design Using VHDL, Pong P. Chu
4. The Verilog Hardware Description Language, Thomas & Moorby
5. Digital Circuits and Systems, Video Lectures by Prof. S. Srinivasan
6. www.bing.com