Computer Architecture – Microprogrammed Control Simulation Results

Brandon Dooley (#16327446)

Disclaimer : This assignment was worked on with another student John Carbeck.

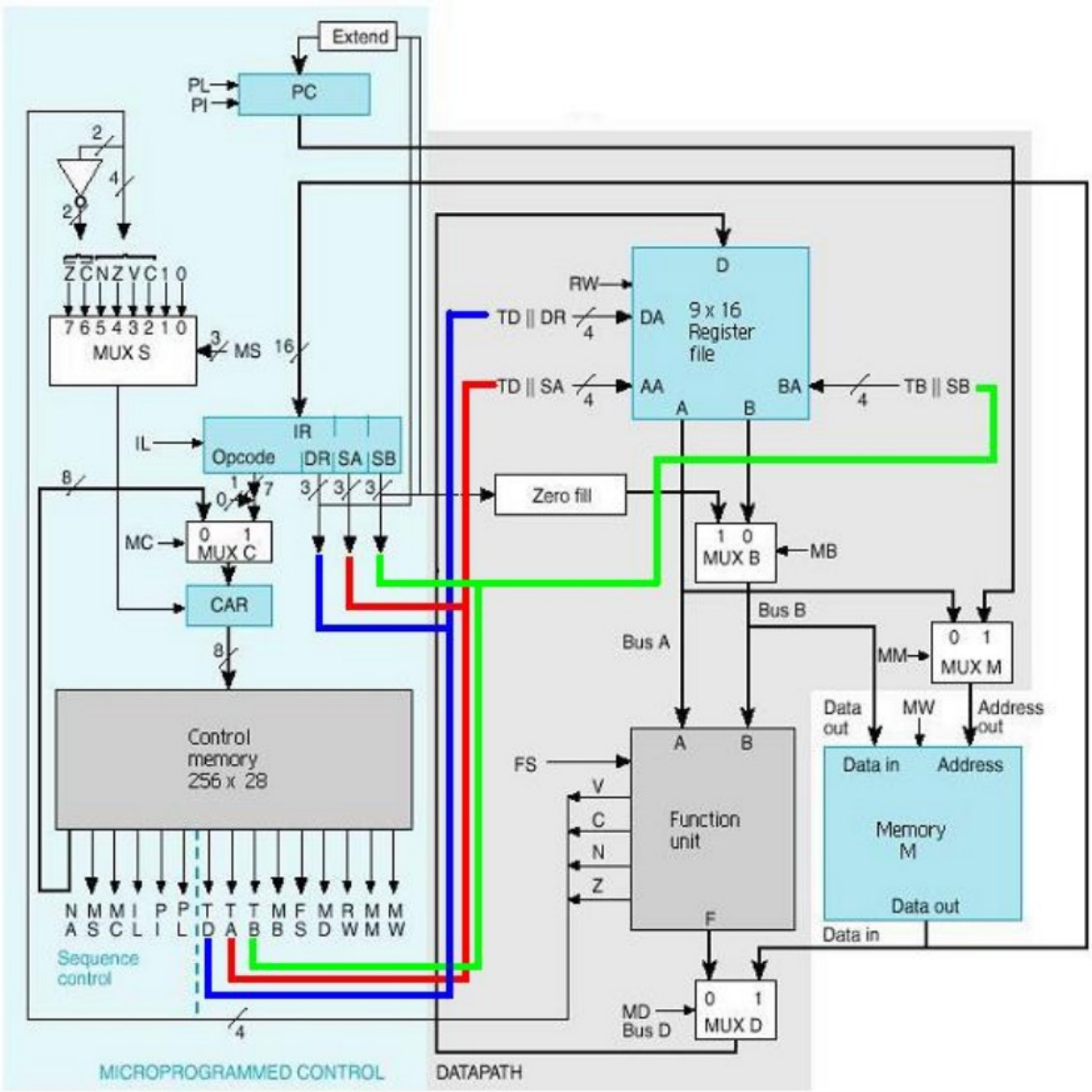


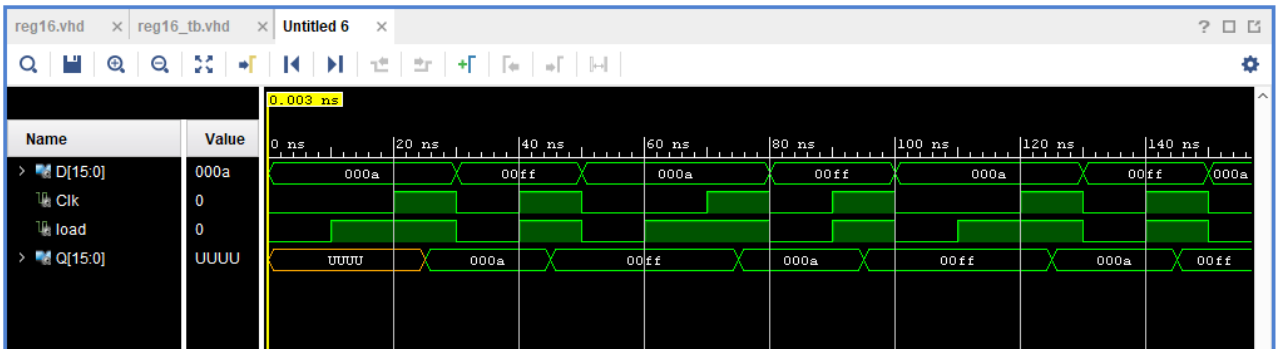
Figure 1: Multiple-Cycle Microprogrammed Control

Table of Contents

DATAPATH

Register File **(9 x 16bit Register File)**

1. Reg16 (16bit Data Register)

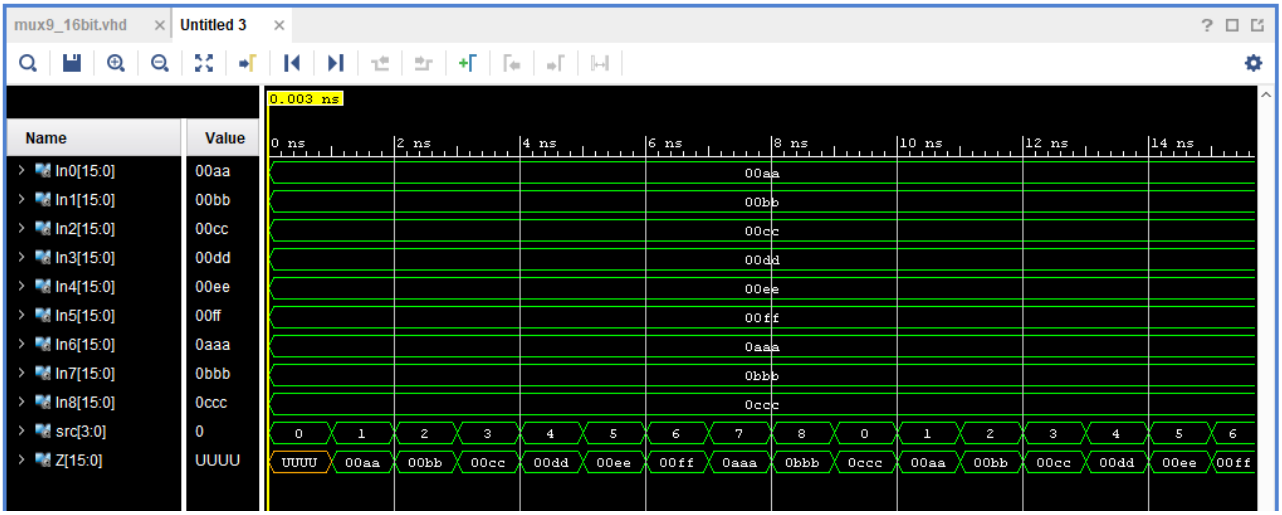


Register is loaded based on Clk and Load. When Clk and Load are both high the contents of the register is loaded with the 16bit value of D. When either Clk or Load are low, contents of the register do not change.

2. 4-to-9 Decoder (16bit)

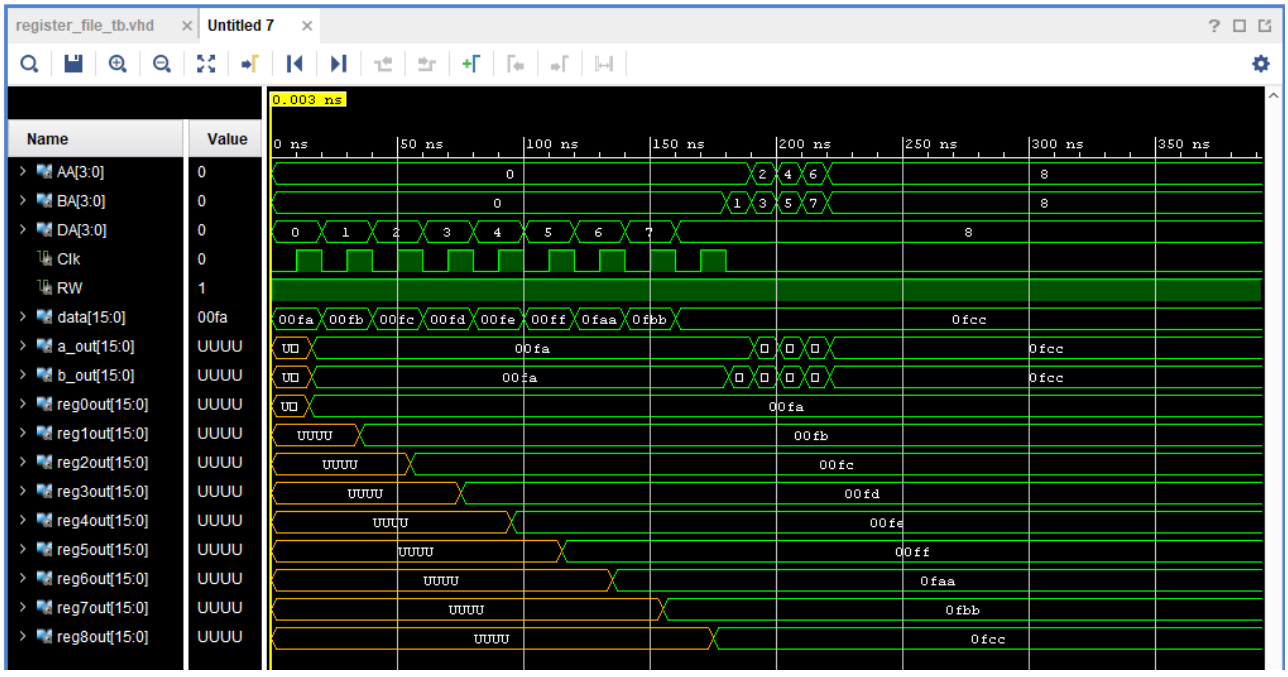
The value at the output lines of the decoder depend on the input vector des. Here each output line is individually tested to ensure selection works correctly. The output line only goes high when RW (Read-Write) is also high.

3. 9-to-1 MUX (16bit)



Each input line In0 – In8 contains a unique 16 bit value. The value of the output line Z is dependent on the vector src. This testbench individually selects each input line and ensures the correct value is carried through to the output line.

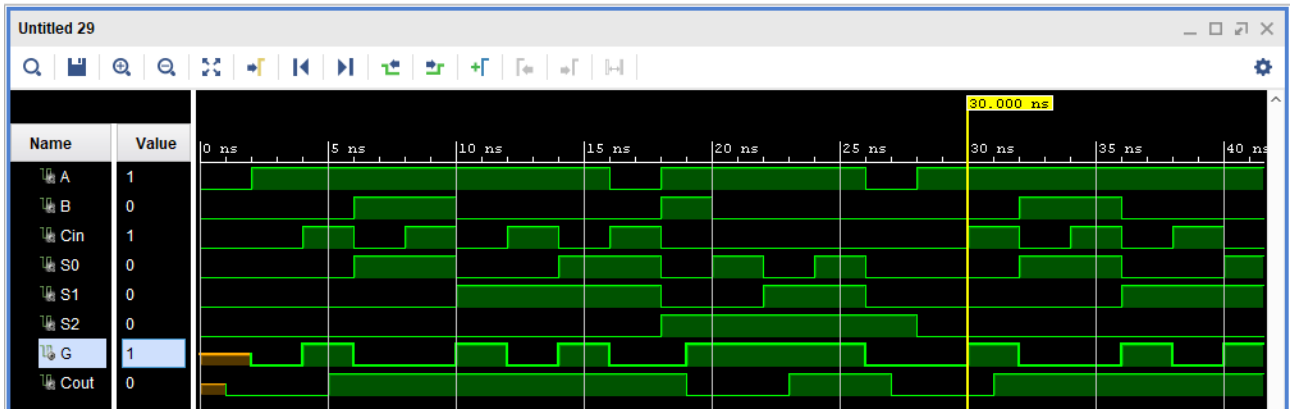
4. Register File (9 x 16bit Register File)



Between 0 and 175ns each individual register is loaded with a unique value upon every Clk cycle. This can be seen in the respective regXout lines. The destination register is decided by the DA vector and the value to be loaded is found in the Data vector. Around the 175ns mark tests are carried out to test the AA (A Register) and BA (B Register) select vectors. The correct corresponding outputs can be seen in the testbench above.

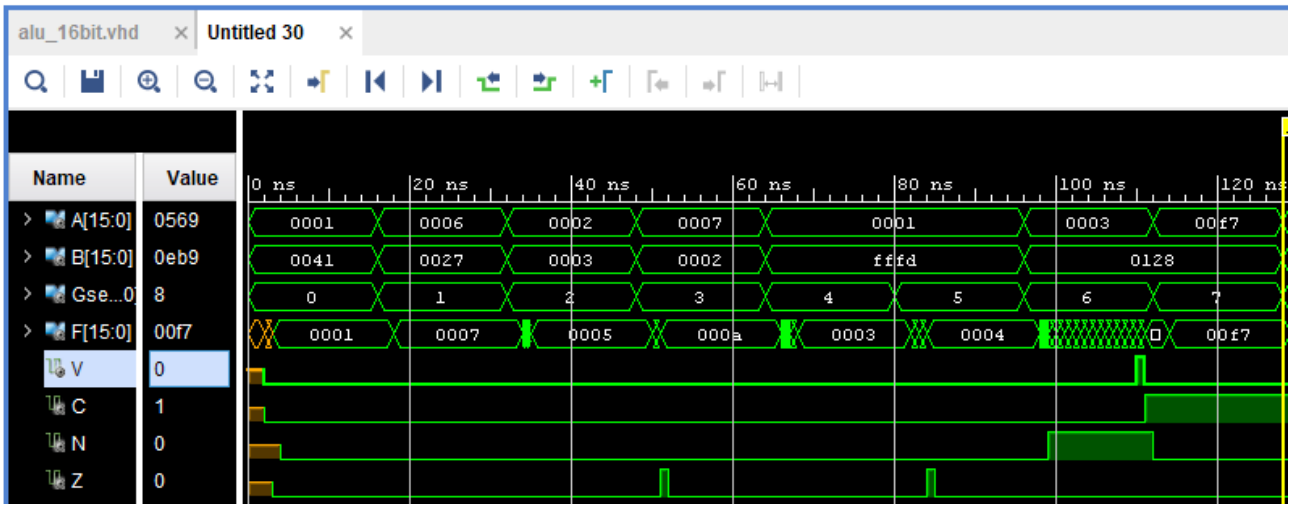
Function Unit

1. Arithmetic Logic Unit (1bit)

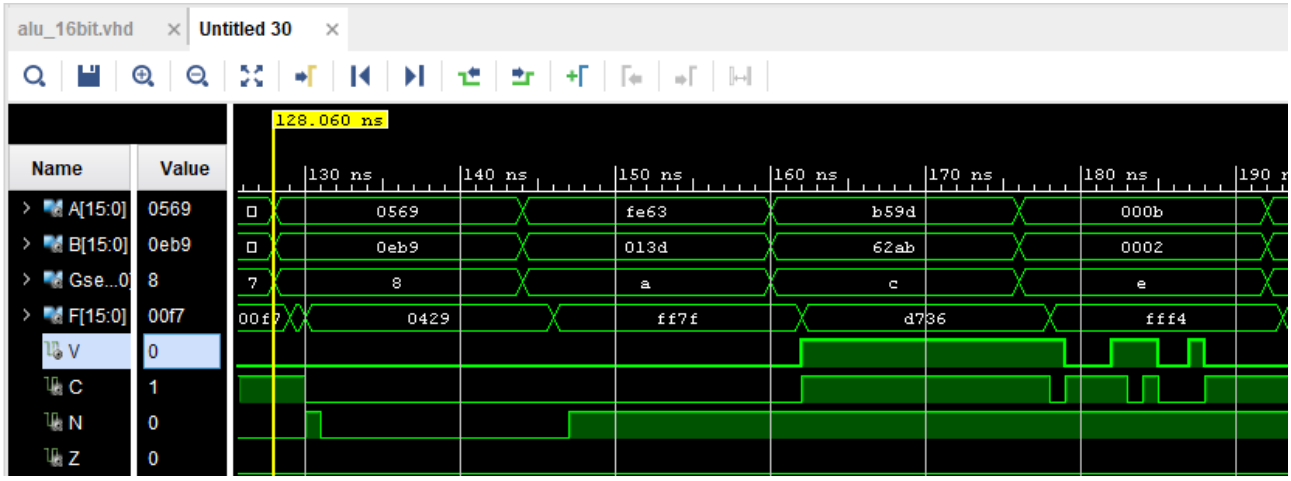


Every 2ns each of the different desired arithmetic operations are tested. The first of which can be seen as testing all zeros. The second test tests G = A which has the select code of S(000). When Cin = 0, the output G can be seen to be 1. When Cin changes to 1, G can be seen to change to 0, successfully performing G=A+1. A detailed description of the testbench can be found in alu_tb.vhd.

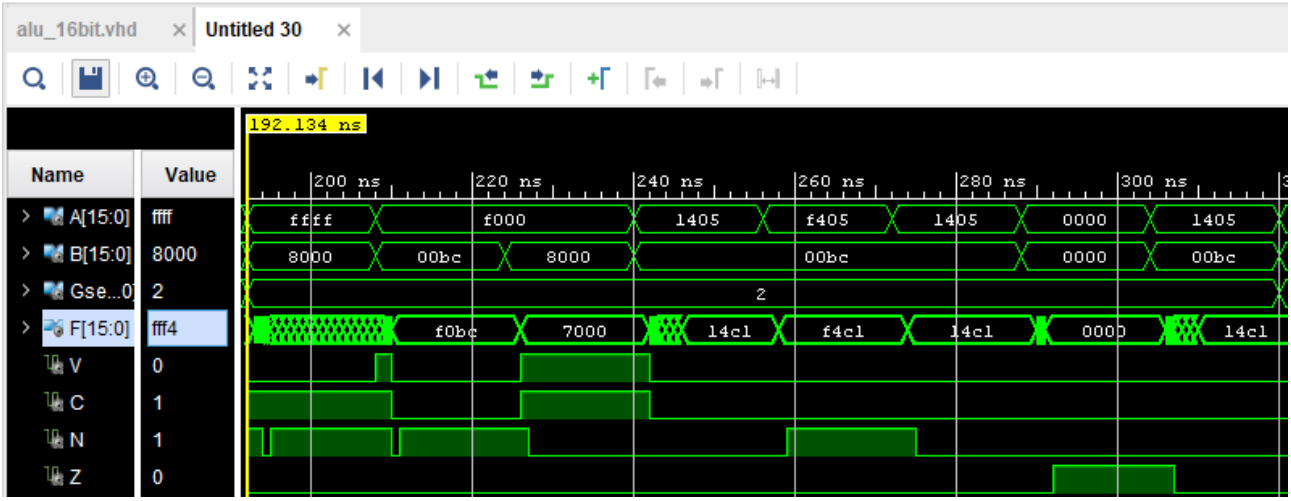
2. Arithmetic Logic Unit (16bit)



Arithmetic Tests: *Various arithmetic operations are performed and tested. The first of which is F = A, the output line can be seen to display the correct result. The second is F = A+1, once again the desired output is shown at F. A detailed description of the testbench can be found in alu_16bit_tb.vhd.*

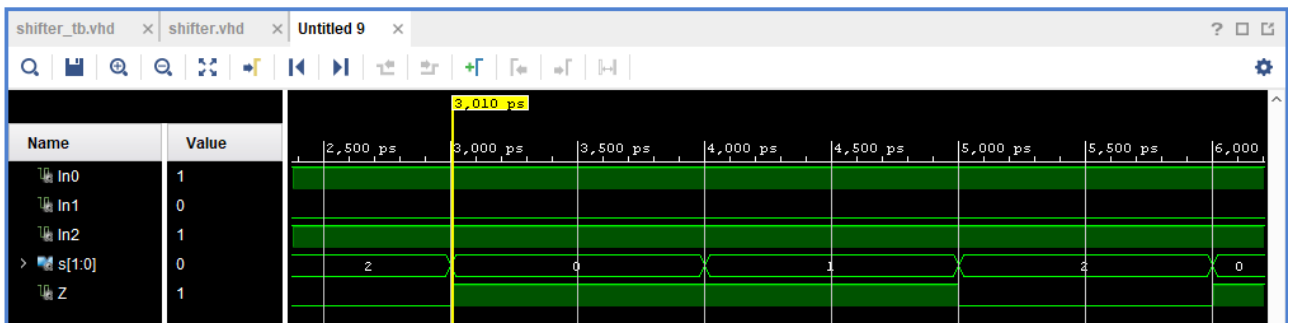


Logic Tests: *Various logic operations are performed and tested. The first of which is F = A^B, the output line can be seen to display the correct result. The second is F = A or B, once again the desired output is shown at F. A detailed description of the testbench can be found in alu_16bit_tb.vhd.*



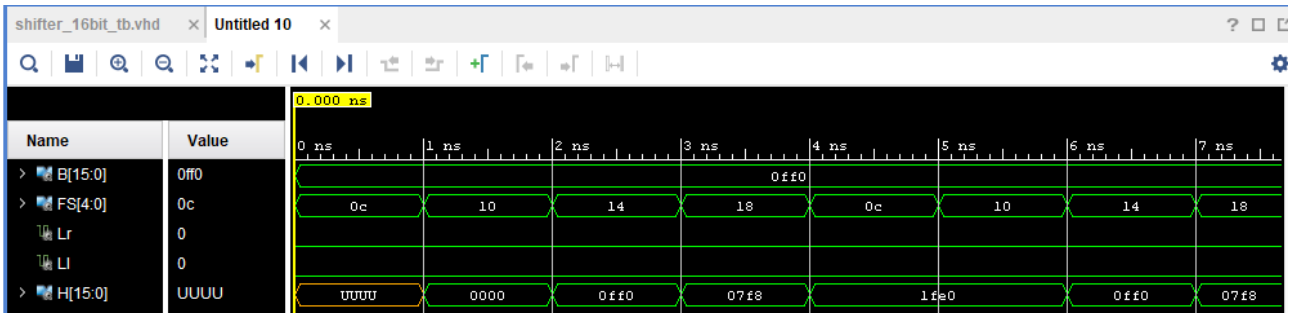
Flag Tests: *Various arithmetic operations are performed and tested in order to ensure the flags are working correctly. The first of which is the Overflow flag (V), the operation performed is 0xFFFF + 0x8000 = 0x7FFF -> V = 1. A detailed description of the testbench can be found in alu_16bit_tb.vhd.*

3. Shifter (1bit)



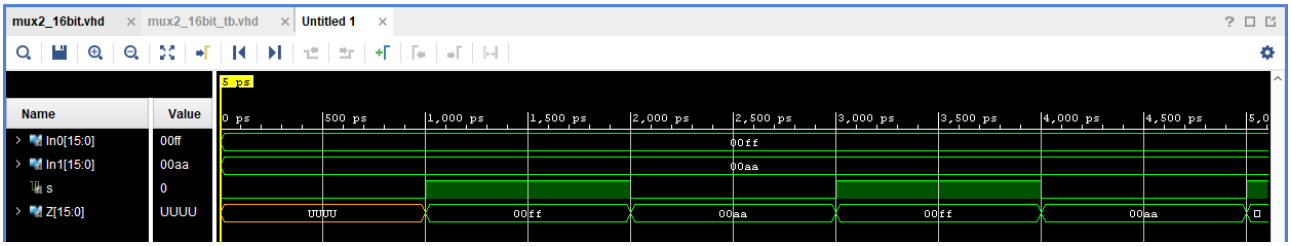
Various shift operations are performed and tested on a 1 bit vector. The input vector In0 was shifted twice as can be seen at the output line Z.

4. Shifter (16bit)



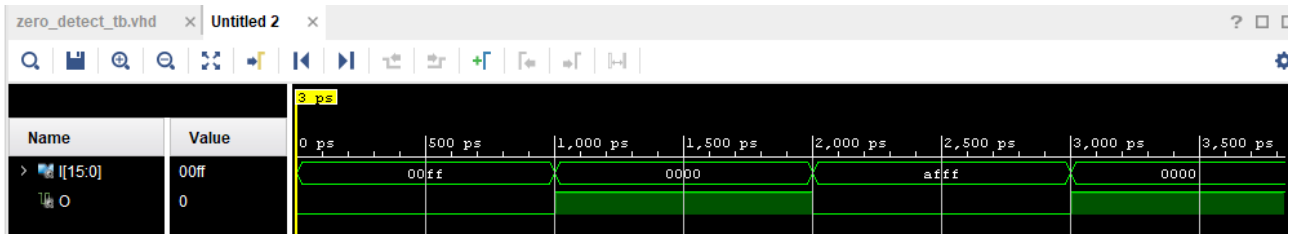
Various shift operations are performed and tested on a 16 bit input vector B. The first test ensures incorrect Opcode (FS) doesn't perform any operation. After this F=B is tested (shift of 0 bits). Followed by LSR, then LSL.

5. 2-to-1 MUX (16bit)



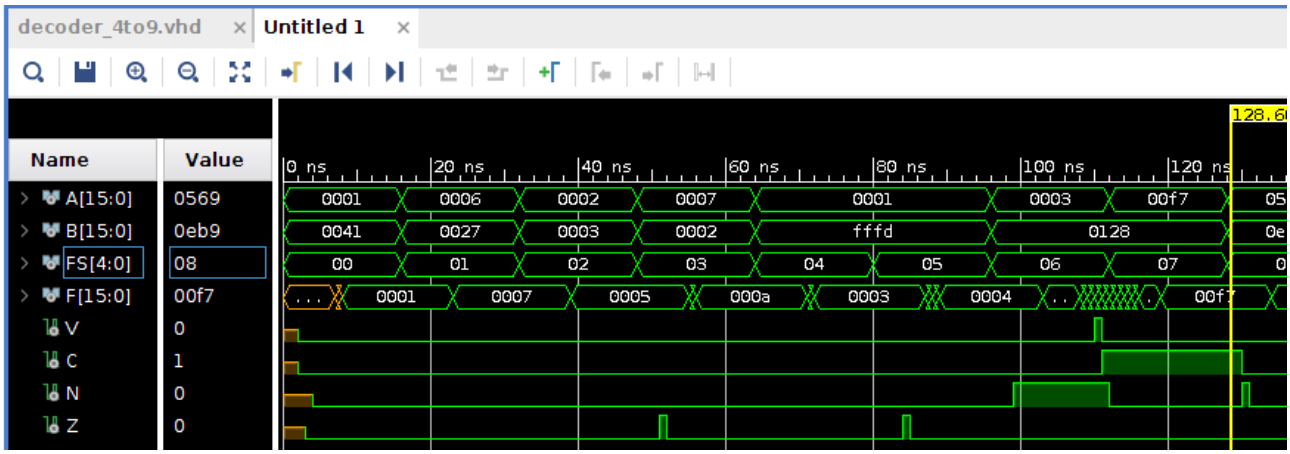
Each of the input lines are selected using the S vector and transferred through to the output line Z.

6. Zero Detect

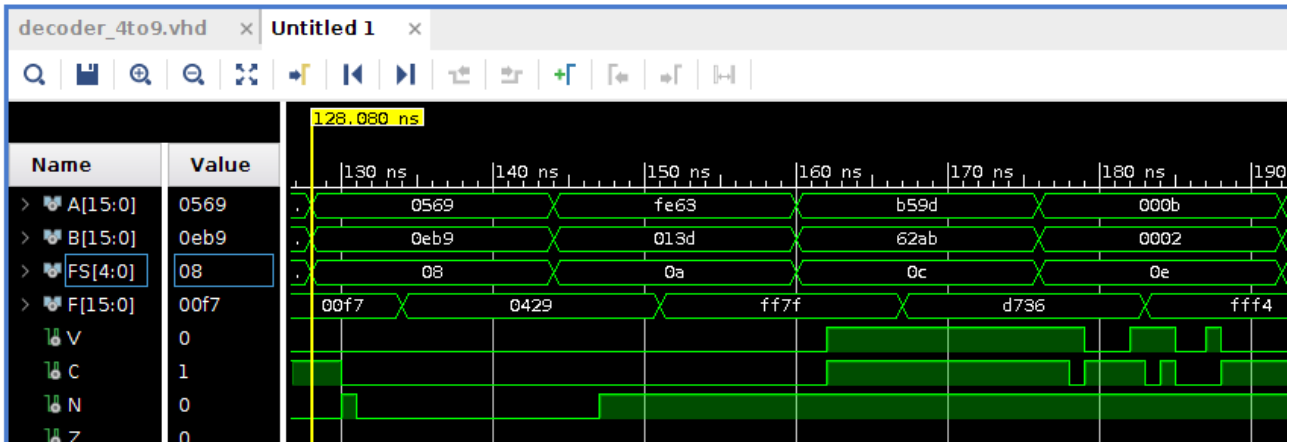


The Zero Detect device is tested to ensure that the Z flag will be set correctly upon a 0 value.

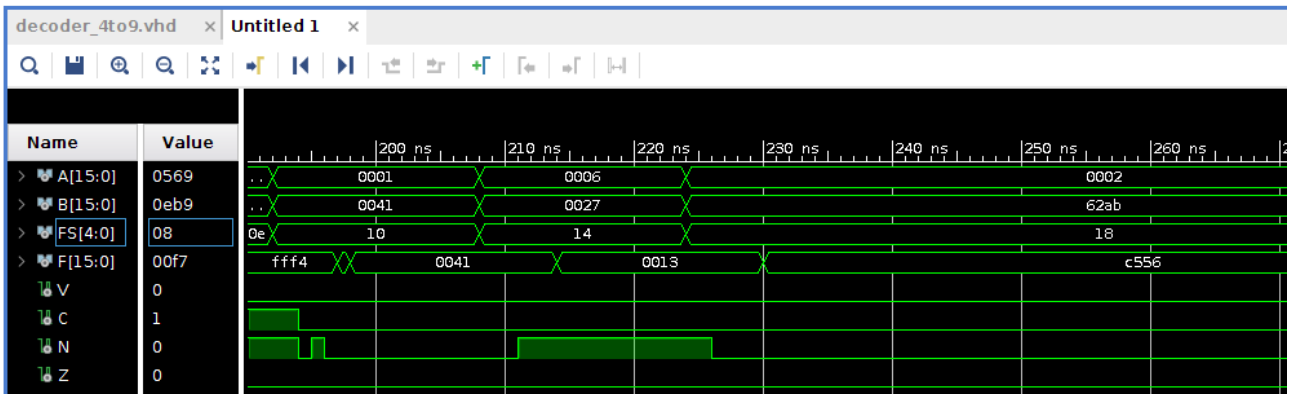
7. Function Unit (16 bit)



Arithmetic Tests: *Various arithmetic operations are performed and tested. The first of which is F = A, the output line can be seen to display the correct result. The second is F = A+1, once again the desired output is shown at F. A detailed description of the testbench can be found in function_unit_tb.vhd.*



Logic Tests: *Various logic operations are performed and tested. The first of which is F = A^B, the output line can be seen to display the correct result. The second is F = A or B, once again the desired output is shown at F. A detailed description of the testbench can be found in function_unit_tb.vhd.*

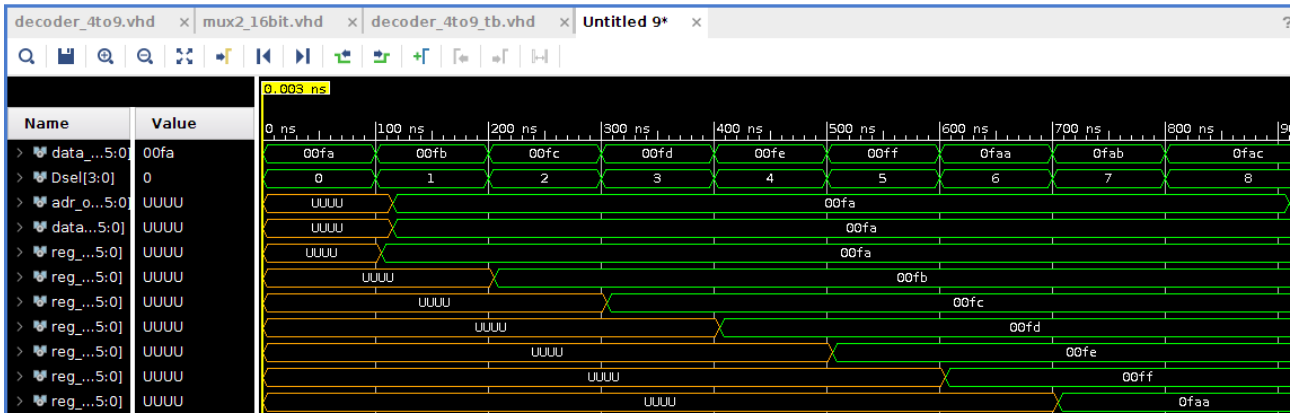


Shifter Tests: *Various shift operations are performed here including LSL and LSR. A detailed description of the testbench can be found in function_unit_tb.vhd.*

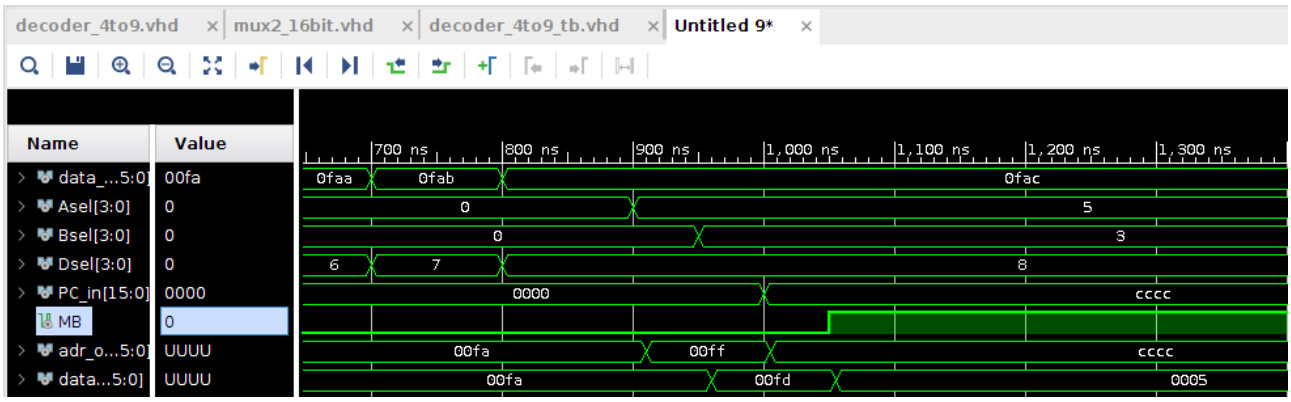
8. Zero Fill

A 3 bit vector (SB) is extended and filled with 0's and converted into a 16 bit vector of the same initial value as SB.

9. Datapath



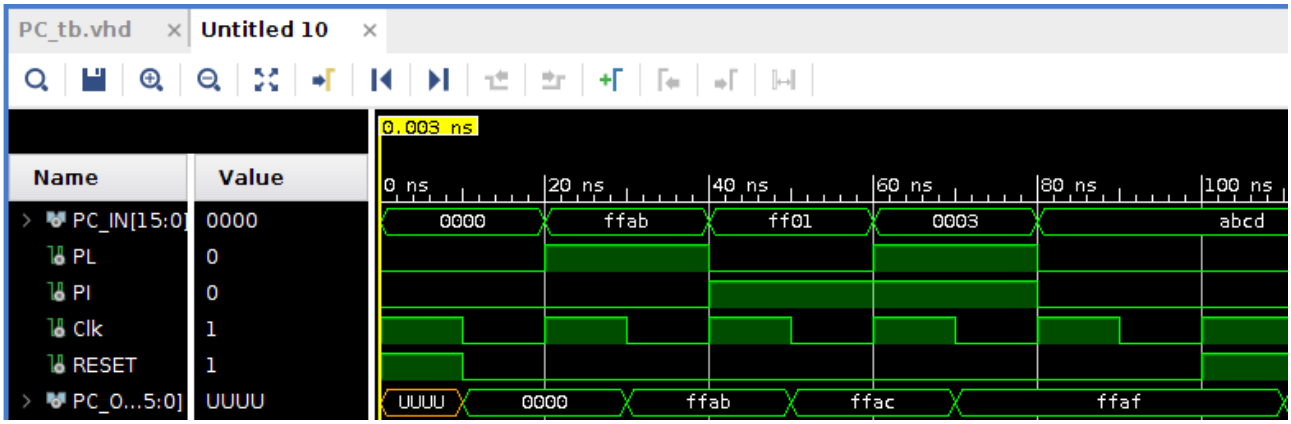
Register Loading: *Each register is first loaded with an arbitrary value to test register loading – including the temp register labeled reg8. Their values can be seen in the line regXout.*



Bus, PC Load and MuxB Tests: *First both Bus A and Bus B are tested, the Bus A selects register 3 (0xFD) and Bus B selects register 5 (0xFF). Their values can be seen outputted to adr_out and data_out. We then test loading in a value from PC_in and passing it through the datapath. This is seen when we pass 0xCCCC through the datapath to adr_out. We then test MUXB and test passing through an SB value of 101 (5).*

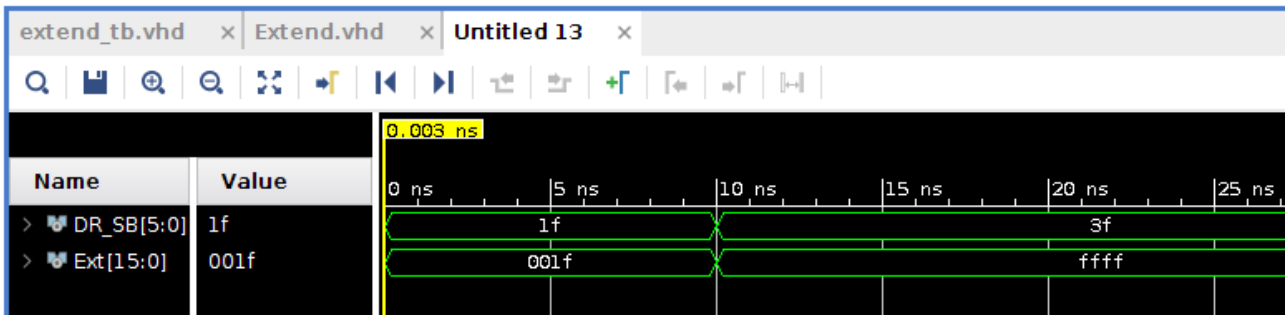
MICROPROGRAMMED CONTROL

Program Counter (PC)



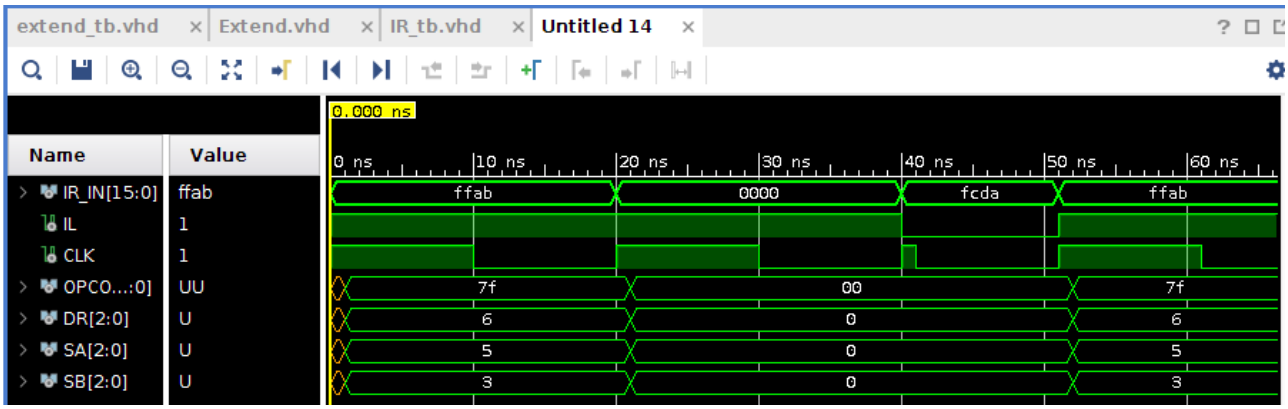
First the PC is reset by setting RESET and Clk to high. Then we test passing without increment PL=1 and PI =0. This can be seen as 0xFFAB is passed straight through. We then test PC increment PL=0 and PI =1. This can be seen as 0xFFAB becomes 0xFFAC. Then finally we test when PL=1 and PI=1, this is what happens to perform a branch operation. Here we add 0x0003 to 0xFFAC to give 0xFFAF.

Extend



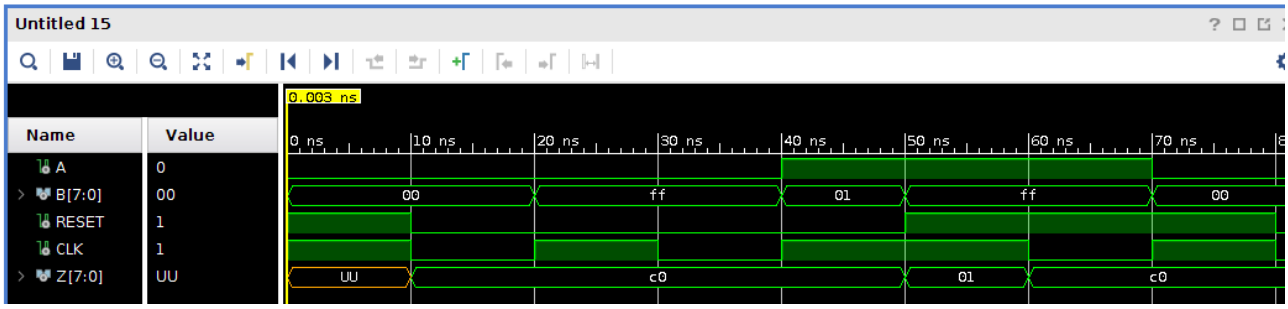
The extend takes in a 6bit value through the vector DR_SB and extends it to a 16bit vector whilst preserving its sign. The first value tested is a value where the MSB is 0, this is preserved. The second value tested has an MSB of 1, as a result it's extended with all 1's.

Instruction Register (IR)



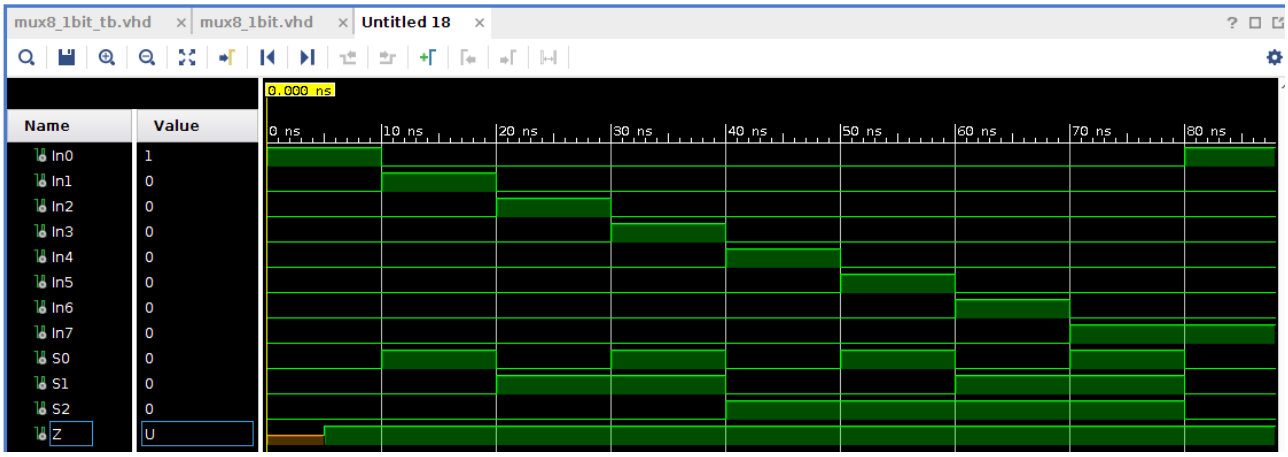
First a random IR_IN is passed through the IR (0xFFAB). This passes through the correct values to the respective output lines; OPCODE = 0x7F, DR =6, SA =5, SB =3. We then test passing through an IR_IN of all 0's. And finally we test that when IL is 0 the IR does not pass through the value at IR_IN.

Control Address Register (CAR)



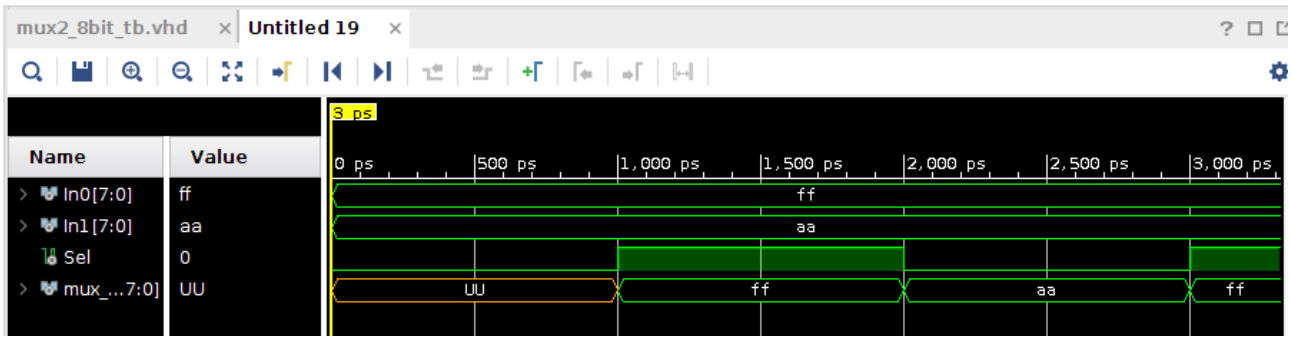
First of all the CAR is put in reset mode by setting RESET to 1 and Clk to 1. This sets the value of the CAR to 0xC0 which is the correct initial address for the process. We then test the CAR when the flags in (MUXS out) are low. This correctly doesn't allow the value of B to pass through. It is then tested that when the flags in are high that the input B is passed through to Z. The CAR is then finally reset.

MUXS 8-to-1 (1bit)



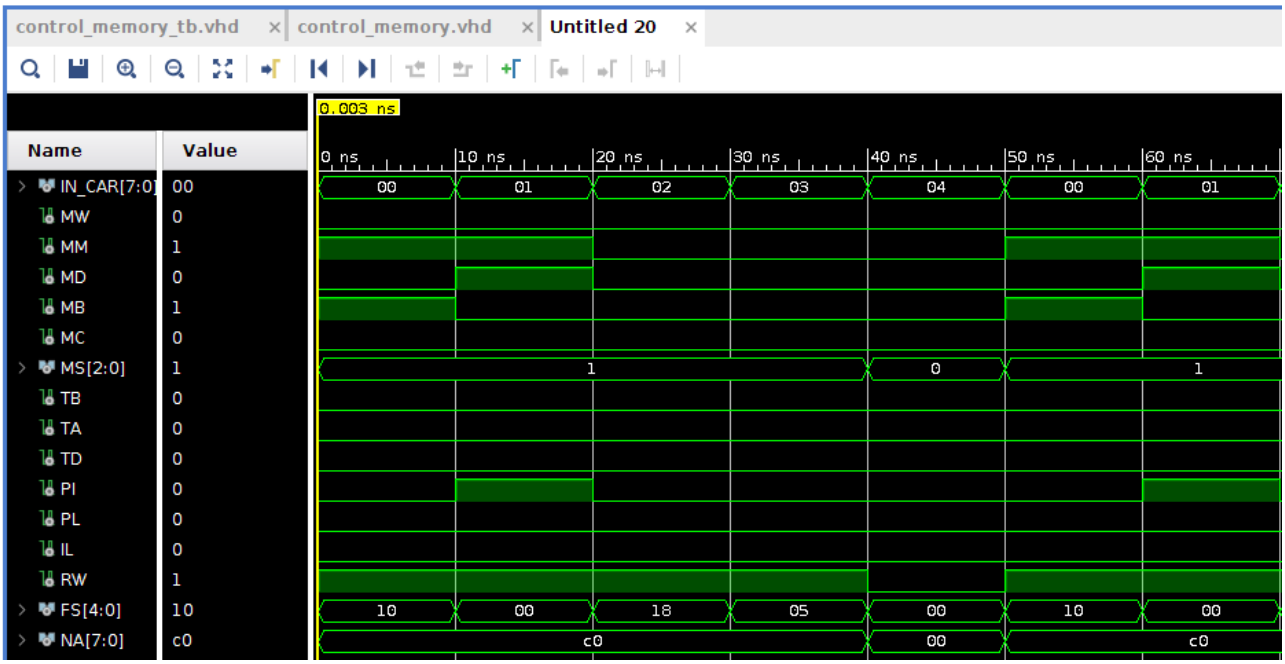
Throughout this test each input line is tested and sent to the output line Z to ensure the functionality of all possibilities.

MUXC 2-to-1 (8bit)



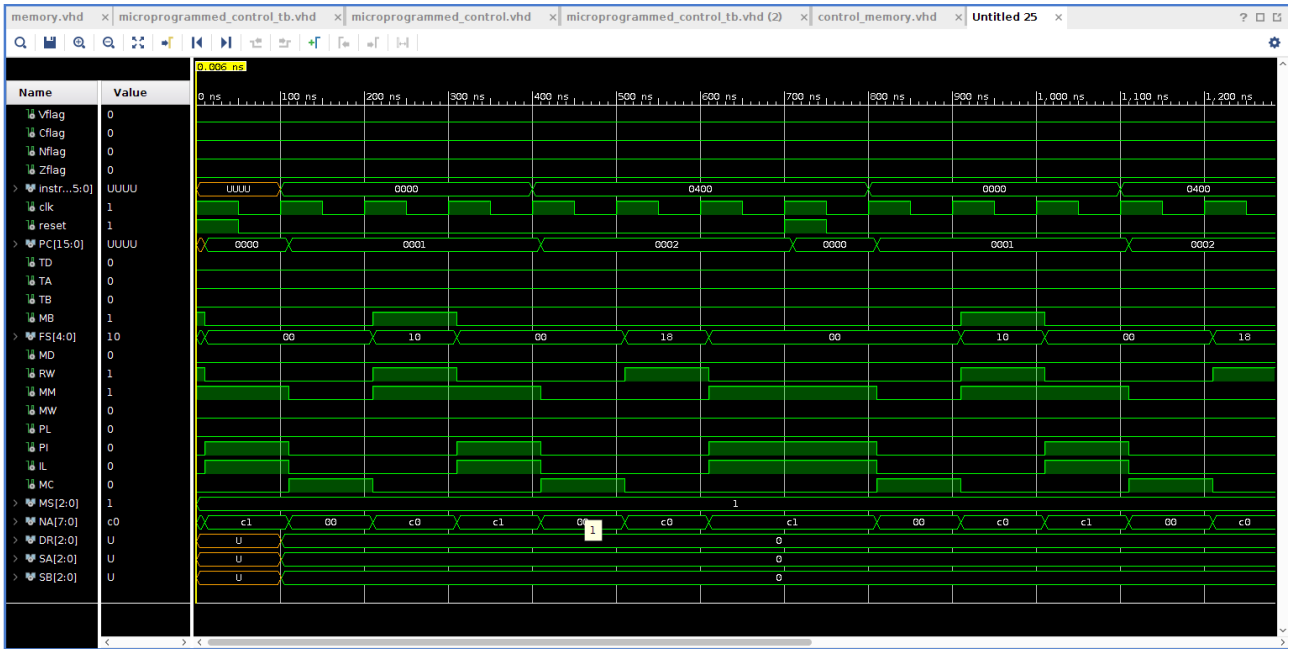
Each possible data input line is tested and sent through the multiplexer to the output line mux_out. The correct desired results are displayed throughout.

Control Memory



The first value passed into the Control Memory is the value 0x00 which points to the first address in memory. The value stored in this address is 0xC020306. This value is then decoded correctly by the control memory setting the appropriate lines to high and outputting the correct values. The desired changes for 0xC020306 are NA = C0, MS = 001, MB = 1, FS = 10000, RW = 1, MM =1. These can all be observed correct. The next 4 addresses in the control memory are then accessed. Further details of the contents of these addresses can be found within control_memory.vhd.

Microprogrammed Control



This testbench appears quite overwhelming but in essence it is extremely simple. What we see here is a total of 7 clock cycles which perform 1 reset operation and 2 whole fetch, decode, execute cycles of given instructions.

The first thing the process does is perform the RESET function by setting both Clk and Reset to high. This causes the system to point to the desired Reset Memory location which (although you can't see it) is defined as 0xC0. In the first clock cycle the system then fetches the instruction stored within the control memory at the address 0xC0 (0xC12C002). This causes the NA to be set to 0xC1, this will be the next address loaded from memory. You can see then that the system loads the value at 0xC1 (0x0030000) which causes ***MC to change to 1****. This means that the system* ***will now load the next address from the Instruction Register (IR)*** *and not from the given NA value.*

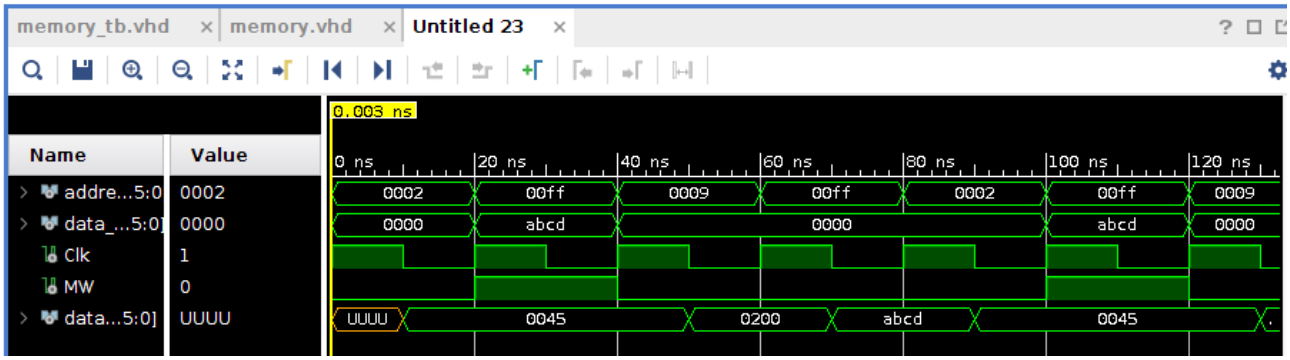
We then, through the testbench pass the system an instruction of 0x0000, which is an Opcode pointing to the instruction stored at the first address in the Control Memory. The instruction at this location (0xC020306) is then fetched and processed. You can see it successfully changes FS to 10000 and various other desired flags.

This process is repeated once more, passing in the instruction 0x0400, this translates to an Opcode of 0000 010, pointing to the second address within the control memory. The instruction stored at this address is then loaded from the Control Memory (0xC02400E) and decoded correctly. This can be seen as FS changes to 00000.

Further details on the contents of the Control Memory can be found within the control_memory.vhd file and details on the testbench can be found within the microprogrammed_control_tb.vhd.

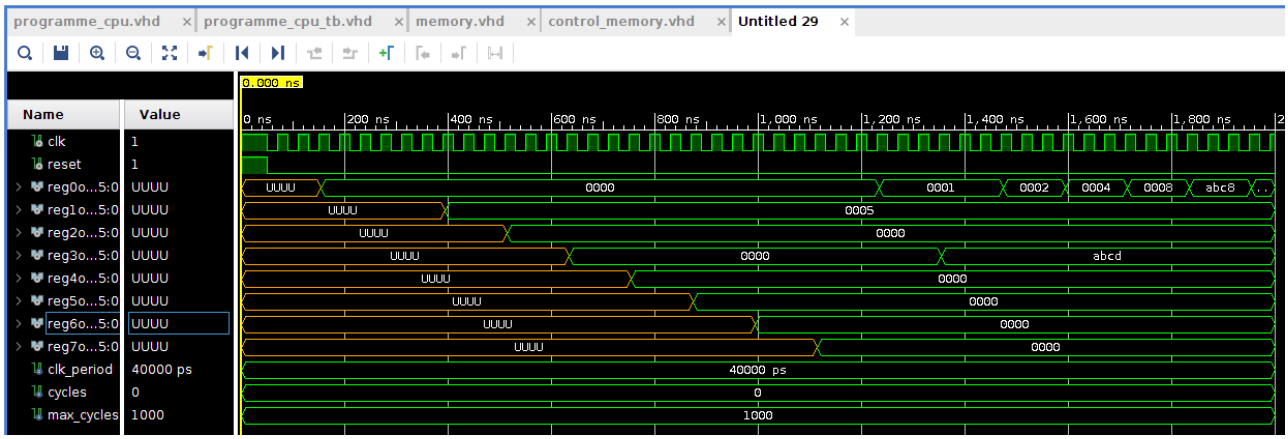
MEMORY

Memory (M)



First, we read the value stored within memory at the address 0x0002 (0x0045) and since MW = 0, when the Clk goes high the contents of address 0x0002 is outputted to data_out. We then test writing to an address in memory, setting MW = 1 and passing in the data 0xABCD and defining the desired address to write this value to as 0x00FF. To prove this successfully worked, we then read another random value from memory, this time at the address 0x0009 (0x0020). Then, finally the value that we previously wrote to the address 0x00FF is read from memory. This is done by setting MW = 1 again and defining address to be 0x00FF, we can see this was a success as 0xABCD is passed to the data_out line.

MULTICYCLE MICROPROGRAMMED CONTROL UNIT



Firstly, the system is put in RESET mode by setting both the Clk and Reset to high. This allows for the PC, IR and CAR to be reset to the appropriate starting values. This then means that the CAR points to the initial location in the Control Memory (0xC0) which contains the instruction 0xC12C002. This tells the system to set IL, PI and MM to high this causes the PC to be incremented and by setting MM to high allows the PC to point point to an address in Memory (0x0001).

It then sets NA to 0xC1 which contains the instruction 0x0030000, which itself sets MC to high allowing for the instruction read from memory to be passed through the IR and into the CAR allowing for program execution to begin. The CAR then points to the Control Memory address 0000000, containing the LDI instruction (0xC020306), this then repeatedly accesses memory incrementally loading pre-defined values into the registers as can be seen above.

The Control Memory is then eventually passed the Opcode 0000 010 as a result of the IR reading the value 0x0400 from Memory. This causes it to point to the second address in the Control Memory to be accessed which contains the instructions for the LSL operation (0xC020184). The program then over the duration of several clock cycles continues on performing the pre-programmed process of shifting the original value in R0 (0x0001) left three times, resulting in the value of R0 being 0x0008.

Finally, the Control Memory is then passed the Opcode 0000 011 as a result of the the IR reading the instruction 0x0619 from Memory. This causes the third address in the Control Memory to be accessed which contains the instruction set for the SUB operation (0xC020054). The program then executes the operation SUB R0, R0, R1 in other words SUB R0, 0xABCD, 0x0008 resulting in the end value 0xABC8 being stored in R0 as can be seen from the above testbench.

Unfortunately, I found it extremely difficult to make it this far and only managed to get the above three instructions coded. If I had more time I feel I would have been able to implement the remaining instructions.

-Brandon Dooley (#16327446)