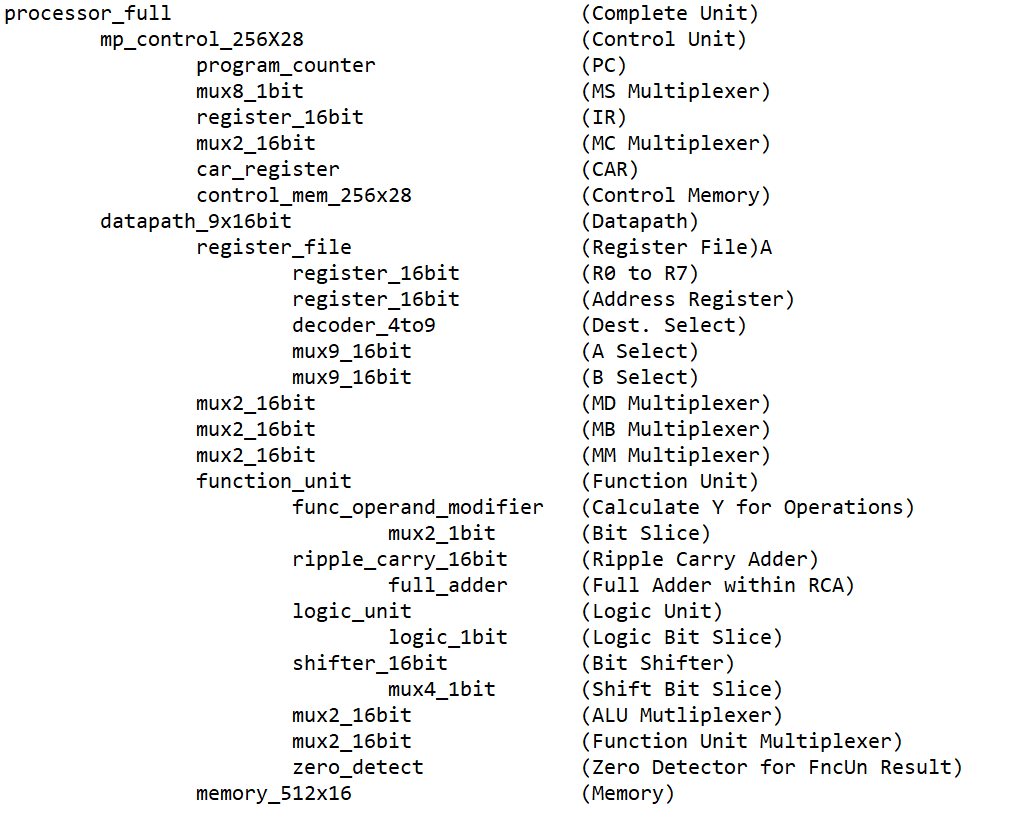
Computer Arch Assignment 3

# Stephen Rowe ID 14319662

The hierarchy of my VHDL implementation of the Micro-coded Instruction Set Processor is as follows:



The **Microcode** used in this program is:

*Address Binary Hex Info*

@C0/192 1100000100001100000000000010 C10C002 // Instruction Fetch

@C1/193 0000000000110000000000000000 0030000 // Execute

@00/000 1100000000100000001000100100 C020224 // Add Immediate

@01/001 1100000000100000000000001100 C02000C // Load

@02/002 1100000000100000000000000001 C020001 // Store

@03/003 1100000000100000000000010100 C020014 // Increment

@04/004 1100000000100000000011100100 C0200E4 // NOT

@05/005 1100000000100000000000100100 C020024 // Add (two registers)

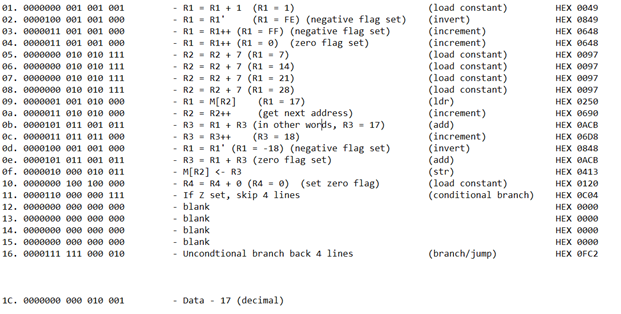
@06/006 1100000011100000000000000000 C0E0000 //branch if zero

@07/007 1100000000100010000000000000 C022000 //unconditional

branch/jump

Microcode in blue was included in the lecture slides, microcode in green is code I wrote myself. This is stored in *control\_mem\_256x28.*

The **machine code** I wrote to demonstrate this system is as follows:



This is stored in *mem\_512x16*. The purpose of this program is as follows:

* Every operand in this Instruction Set is used
* The first four lines are demonstrating that the ADI, NOT, INC instructions work, and that they modify the condition bits depending on the results from the Function Unit
* Lines 5 to 8 is to put the value ‘28’ into R2
* We load the value of MEM[28] into R1 (using R2 as an address reference), then incrementing the address in R2 to get the next sequential address
* Copy this value into R3, then we modify both registers (R1 and R3) so that they are the two’s compliment of each other, then we add them together.
* Store this value into MEM[R2]
* We set the Zero flag with R4 = R4 + 0 (i.e. 0 + 0), so that we can use the conditional branch operation, “If Z set, branch to 0x16”
* At 0x16, we have an instruction that unconditionally branches/jumps, but it jumps back to 0x11
* The program then remains in this loop of jumping between 0x16 and 0x11

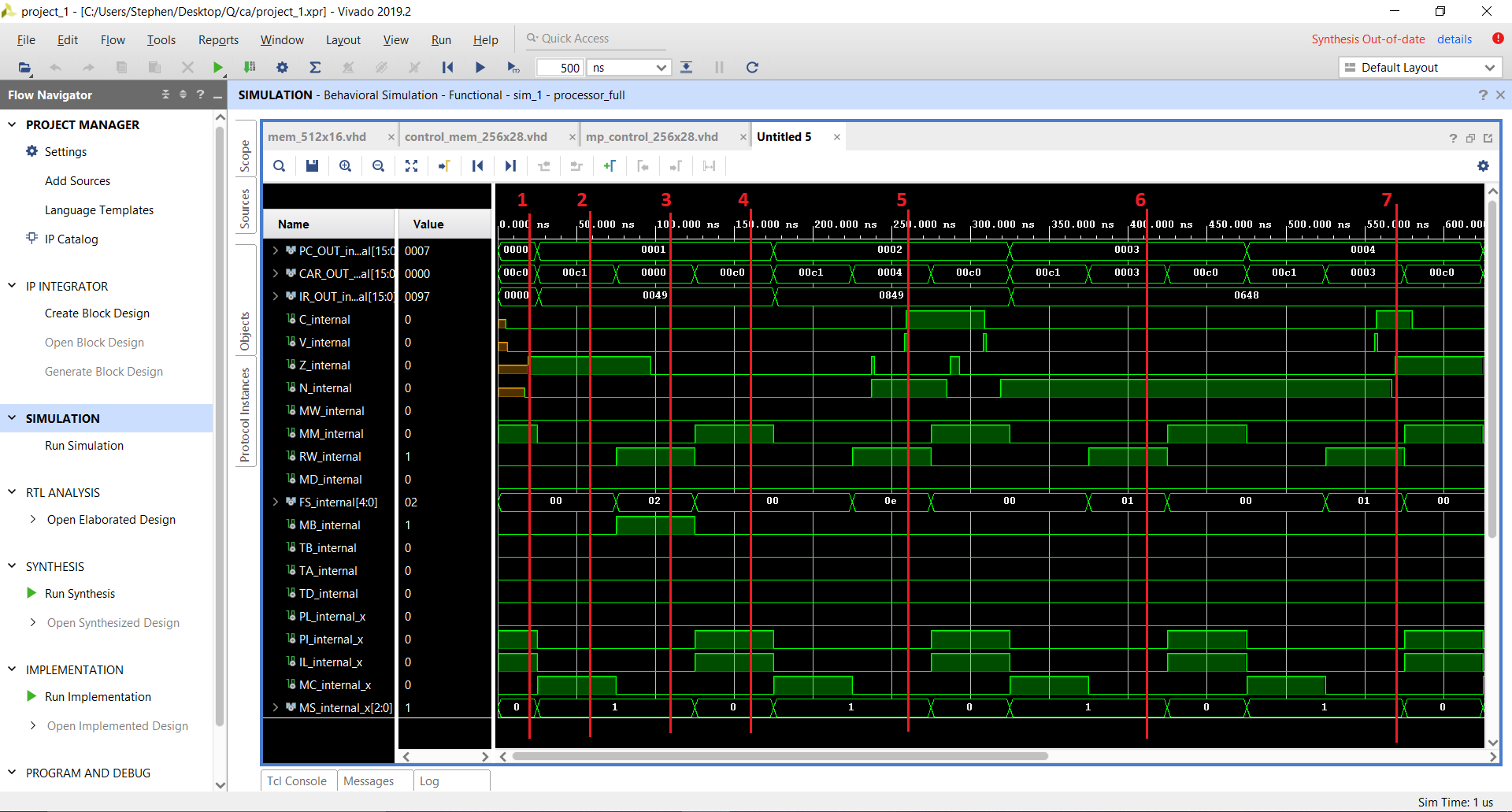
For this assignment, I went back and modified the datapath, so that it could accept the Control Bits produced by the Control Memory entity. This involved changing the output and input ports, but also renaming many of the signals within components.

For the PC and CAR registers, I created two new entities that were based on the register files from earlier assignments, but the main difference is that these new entities can increment their value on their own, depending on the input signals from Control Memory.

Resetting the processor was achieved by initialising the CAR register to 0xC0 (control address for Instruction Fetch), initialising PC as 0x0000 (first address in Memory). I also initialized all the register values as 0x0000 so that we could run, for example, ADD or INC operations without having to load the value 0 into the register first.

The processor\_full file serves as the combination of the Control Unit, Datapath, and Memory. Since the processor does not accept external signals (all the instructions and control code is included within memory and control memory respectively), then I could use this file as its own Testbench, which is why there is no processor\_full\_tb. In the processor\_full file, there are XX\_internal singals, which are essentially the control bits as they are being both inputted and outputted, this was the best way of showing what these values are at any given moment on the Waveform. The images you will see below are from processor\_full.vhd.

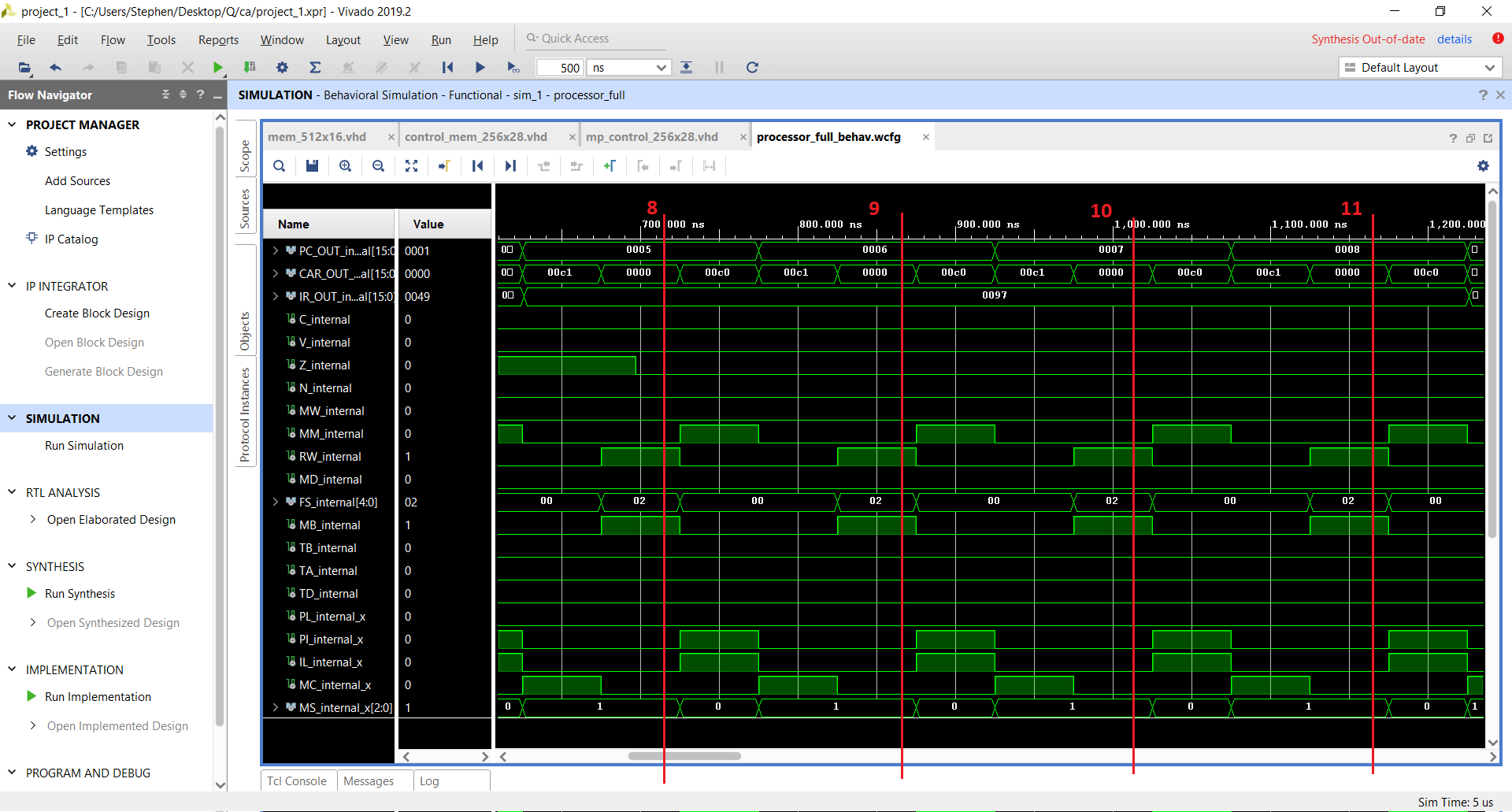
The following is the result of the first 4 instructions. Since there were so many signals in this file, then I’m not able to show them all in one screenshot, and the signals missing from these screenshots include DR, SA, SB, Address Out and Data Out.



At line:

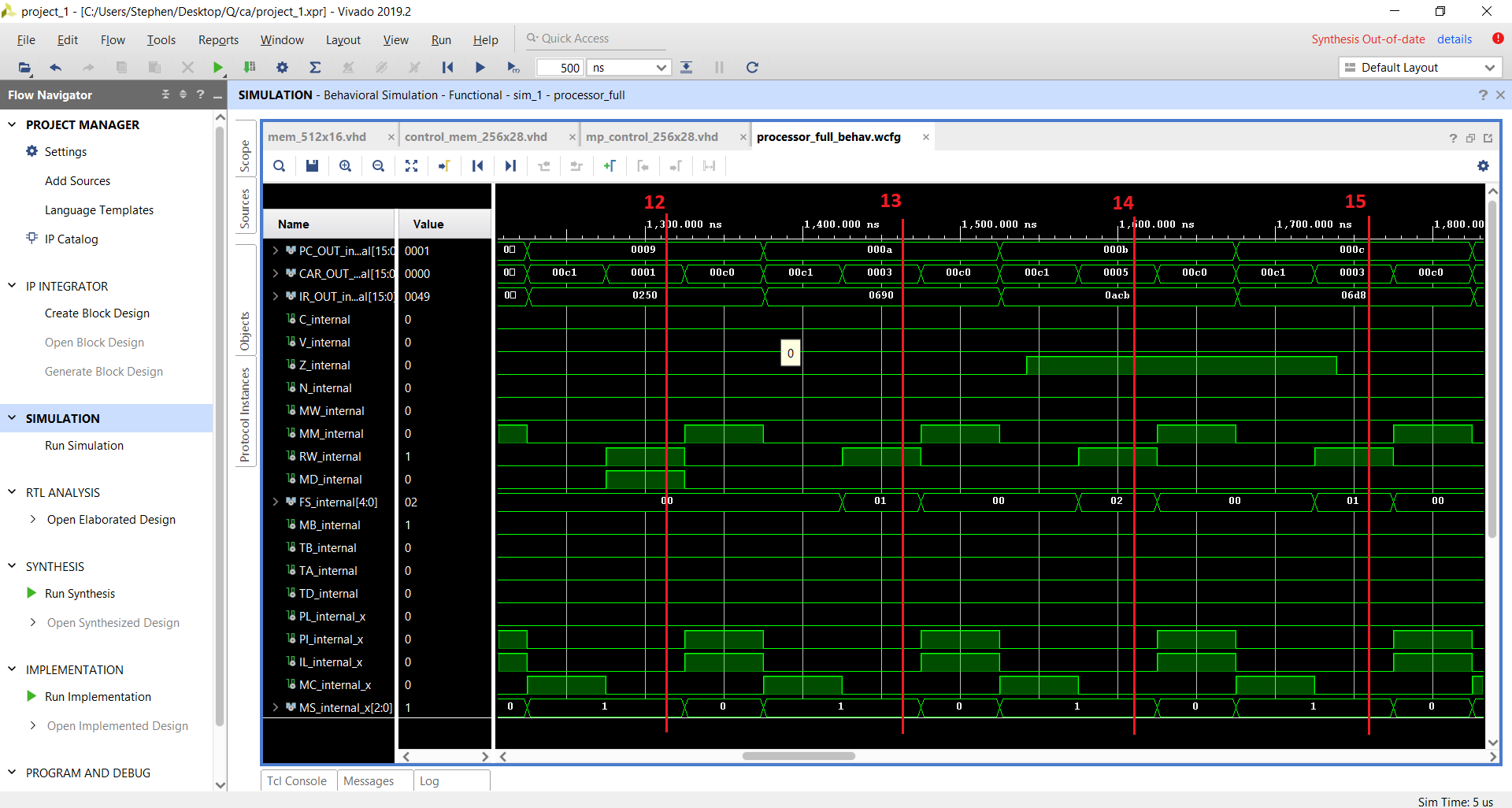
1. The PC and CAR have been initialised to 0x0000 and 0x00C0 respectively. As We can see that MM, PI and PL are set in this state, as intended.
2. We then advance to Execute State, with MC set and MS set to 1. The instruction is now being read from IR to CAR.
3. This is when the first Instruction we have written is actually executed (R1 = R1 + 1). We have entered state 0x0000, which is add immediate. None of the CVZN flags are set, as R1 is now 1.
4. We have returned to IF, then to go to EXO (0xC0, 0xC1).
5. The next instruction is R1 = R1', so our state is 0x04. The value in the register is now FFFE, so the N flag is set. (-> IF -> EXO).
6. R1 = R1++, so R1 now equals 0xFFFF. Negative flag is still set.
7. Again, the instruction R1 = R1++ is carried out, so R1 = 0. Negative flag is no longer set, and Zero flag is set.

*Please note that the PC is incrementing every time PI is set.*

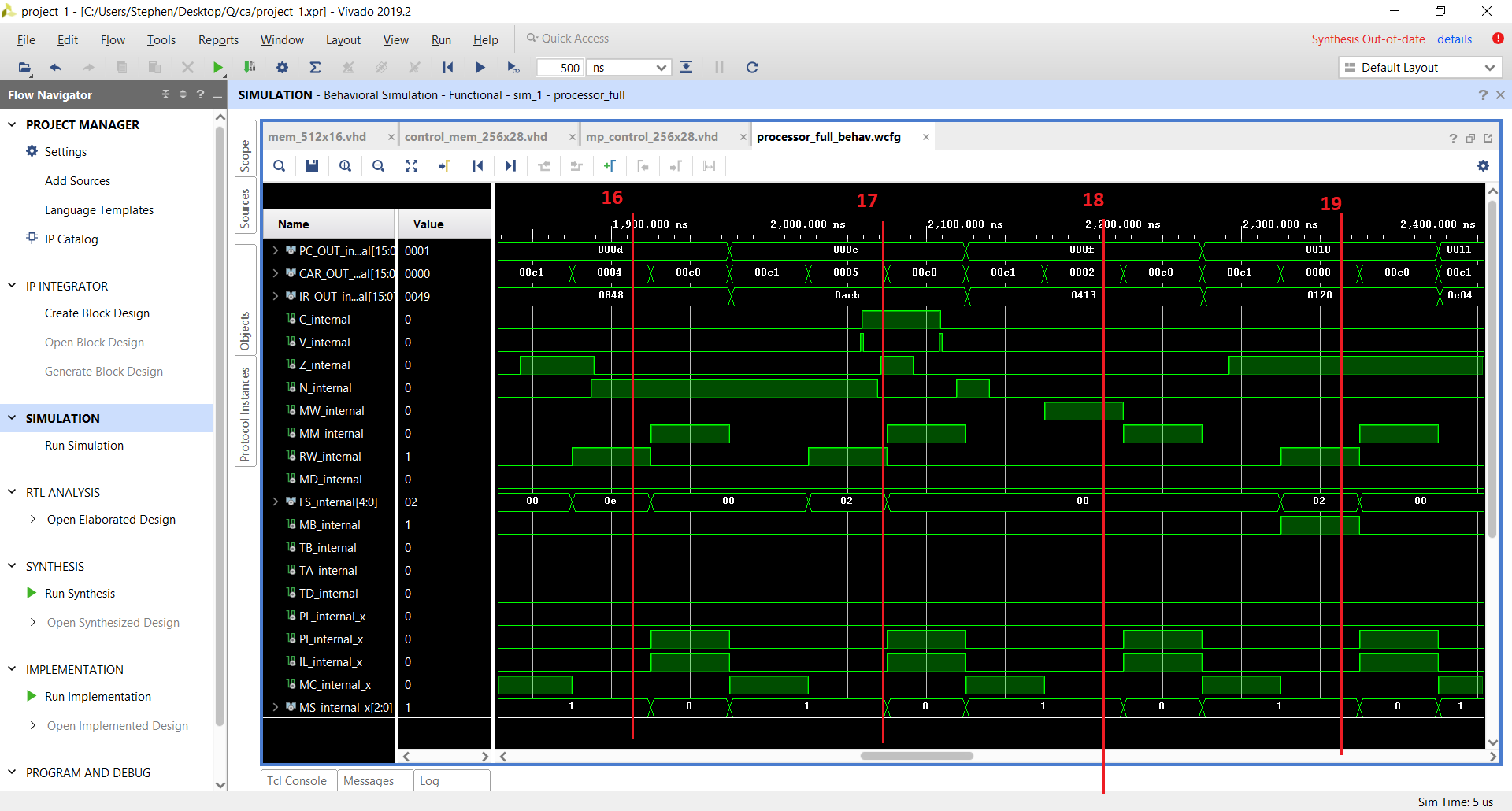


1. R2 = R2 + 7
2. R2 = R2 + 7
3. R2 = R2 + 7
4. R2 = R2 + 7

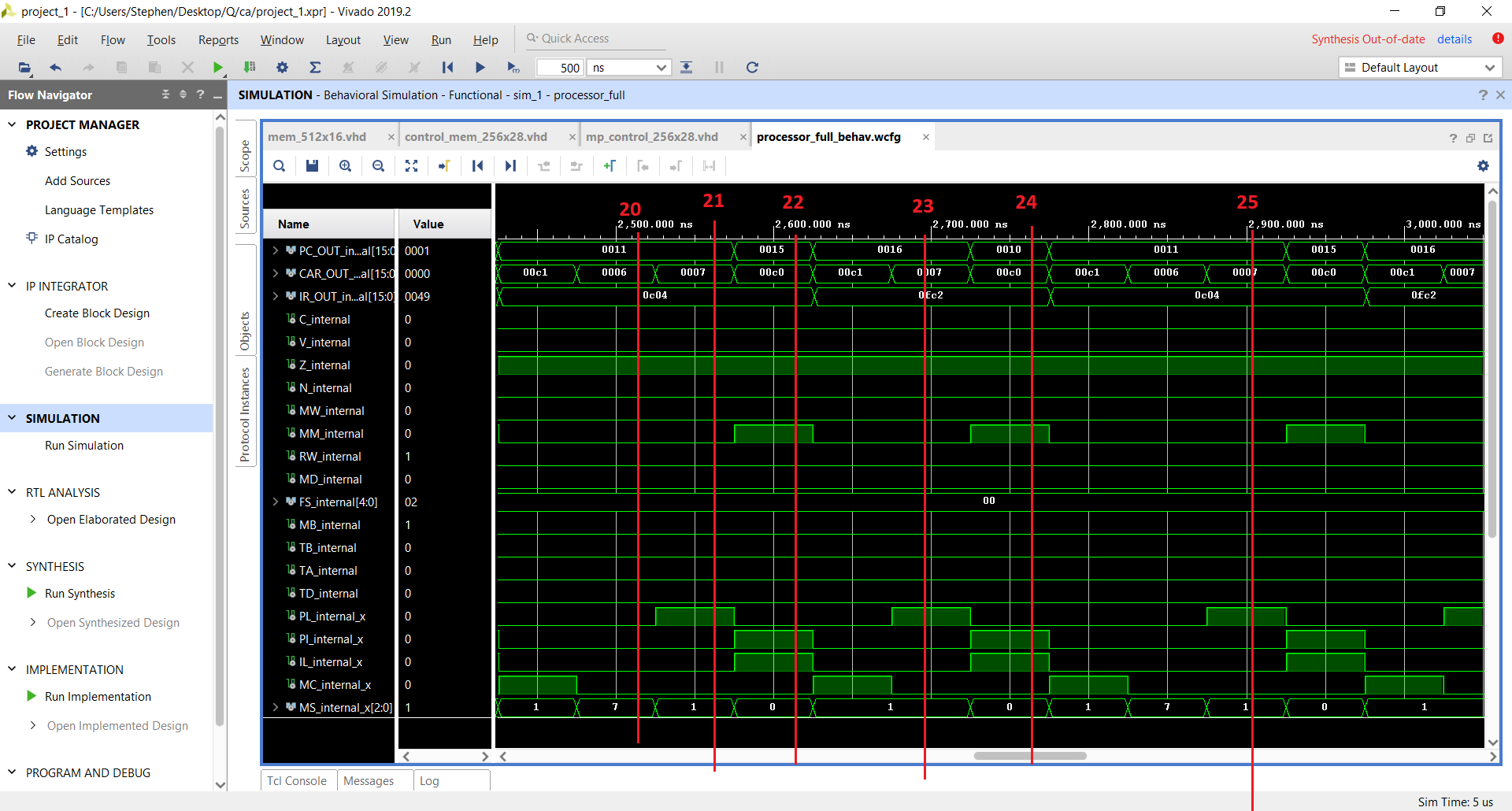
Unfortunately, I’m not really able to show the values actually stored within the Registers, so I have to rely on the CVNZ bits to see that the operations I’m expecting are actually being carried out within the registers.



1. R1 = M[R2], R1 now contains the value that I stored in memory address 0x1C, which is 17.
2. R2 = R2++ (Increment address stored in R2)
3. R3 = R1 + R3 (in other words, R3 = 17)
4. R3 = R3++ (R3 = 18)



1. R1 = R1' (R1 = -18). The negative flag is set. At this point, R1 is -18, R3 is 18, they are the two’s compliment of each other.
2. R3 = R1 + R3. The result is 0, and we can see that the Z flag is set. From this we can conclude that (i) loading from memory is successful, (ii) register addition is successful.
3. M[R2] <- R3, we are storing the result back into Memory.
4. R4 = R4 + 0, this operation is just to set the Z flag again for good, to make sure that the Branch If Zero instruction works correctly.

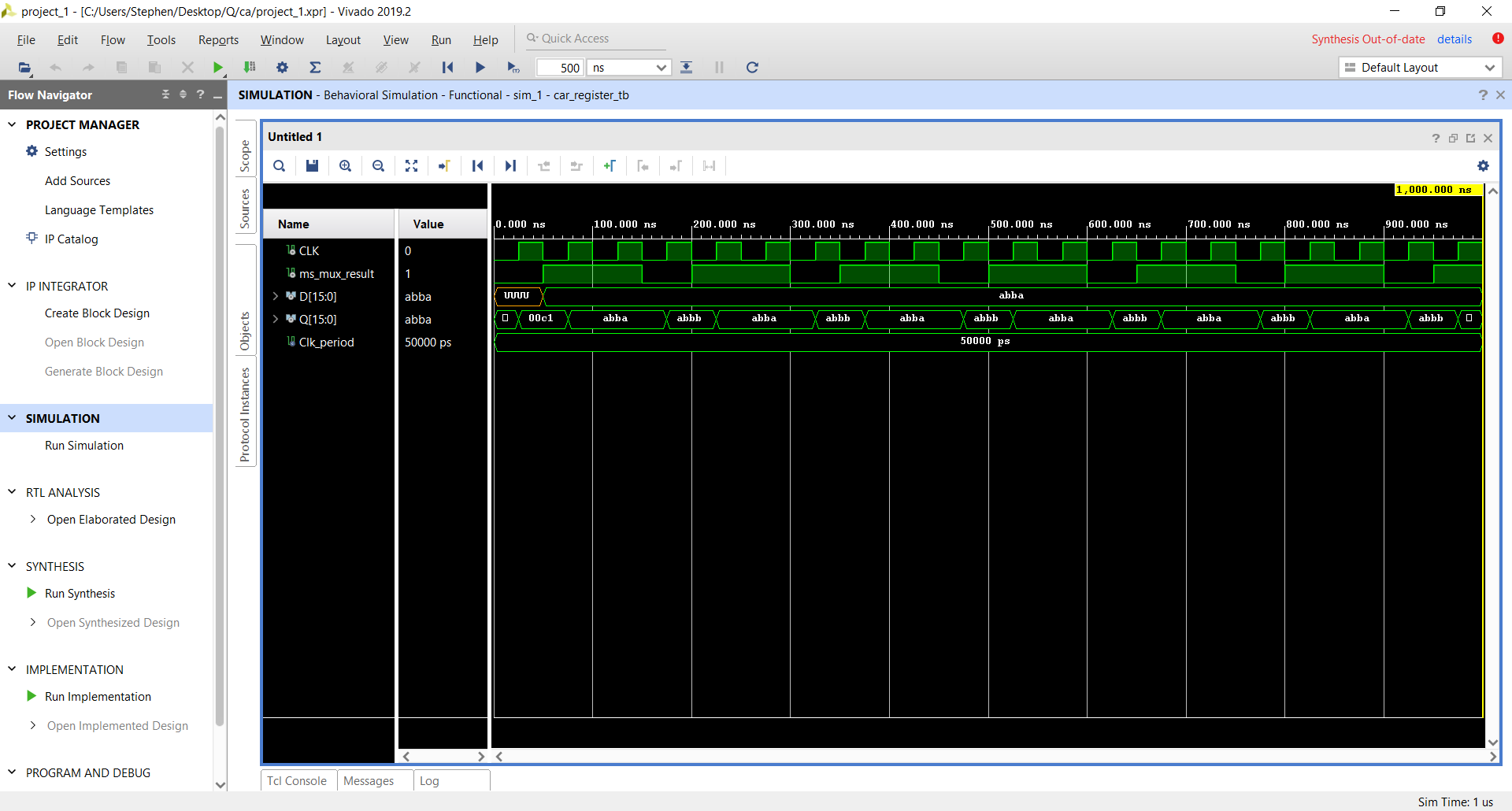


1. The actual instruction for “Branch If Zero” is in memory address 0x10, but because the PC is incremented after this instruction was received, then we can see that PC is at 0x0011. We are current in State 6, which is checking that, if NOT Z is set, then we go to the normal Instruction Fetch state. Else, we move to state 7, which is responsible for Branching.
2. We have moved to state 7, so this means that Z has been recognised as being set.
3. The instruction that we used for calling “Branch if Zero” specified that if it is zero, then we jump forward 4 lines in memory. We can see that this was successful, as PC has now jumped forward to 0x0015.
4. At 0x0015, we have an instruction that is an unconditional branch. This will jump back 5 lines, back to the “Branch if Zero” line.
5. We can see that this will be an infinite loop.

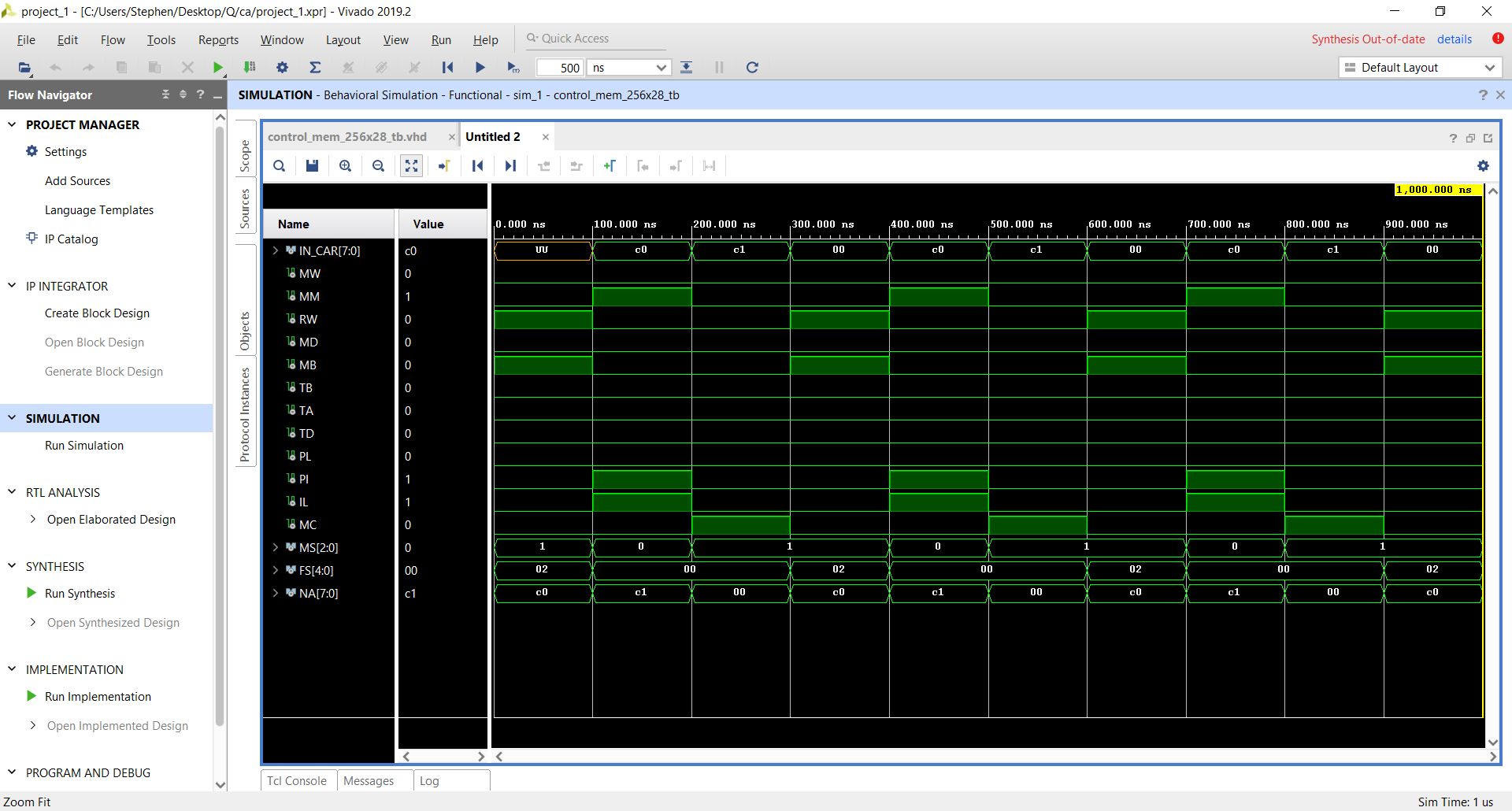
## Other Test Benches

***Below are the screenshots of all the other testbenches I created for this system. In the testbenches files themselves, there are comments fully explaining what each signal change is demonstrating in terms of entity function. Please take a look at the comments in these files for explanations for the following waveforms.***

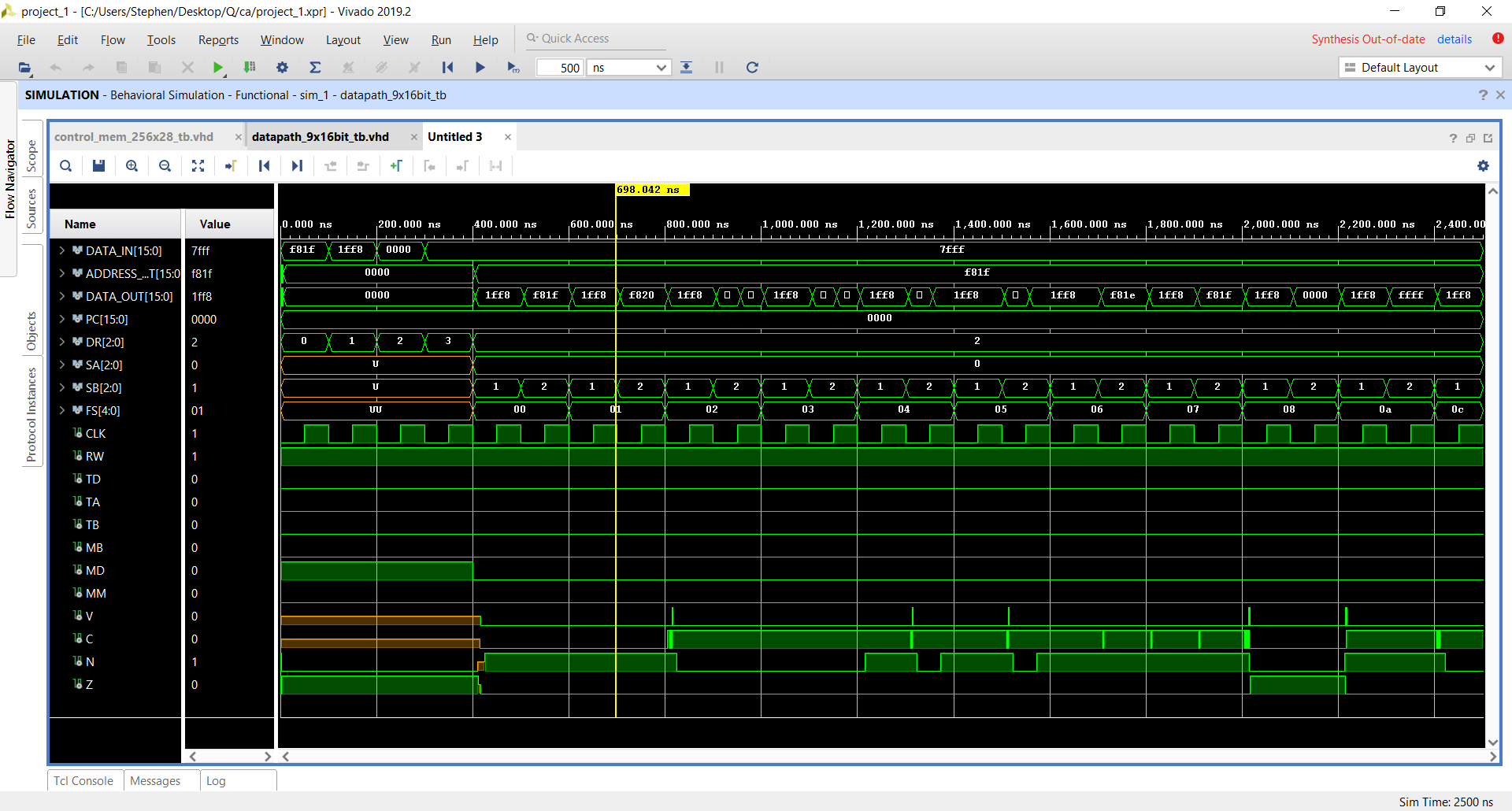
**CAR Register:**

****

**control\_mem\_256x28\_tb:**

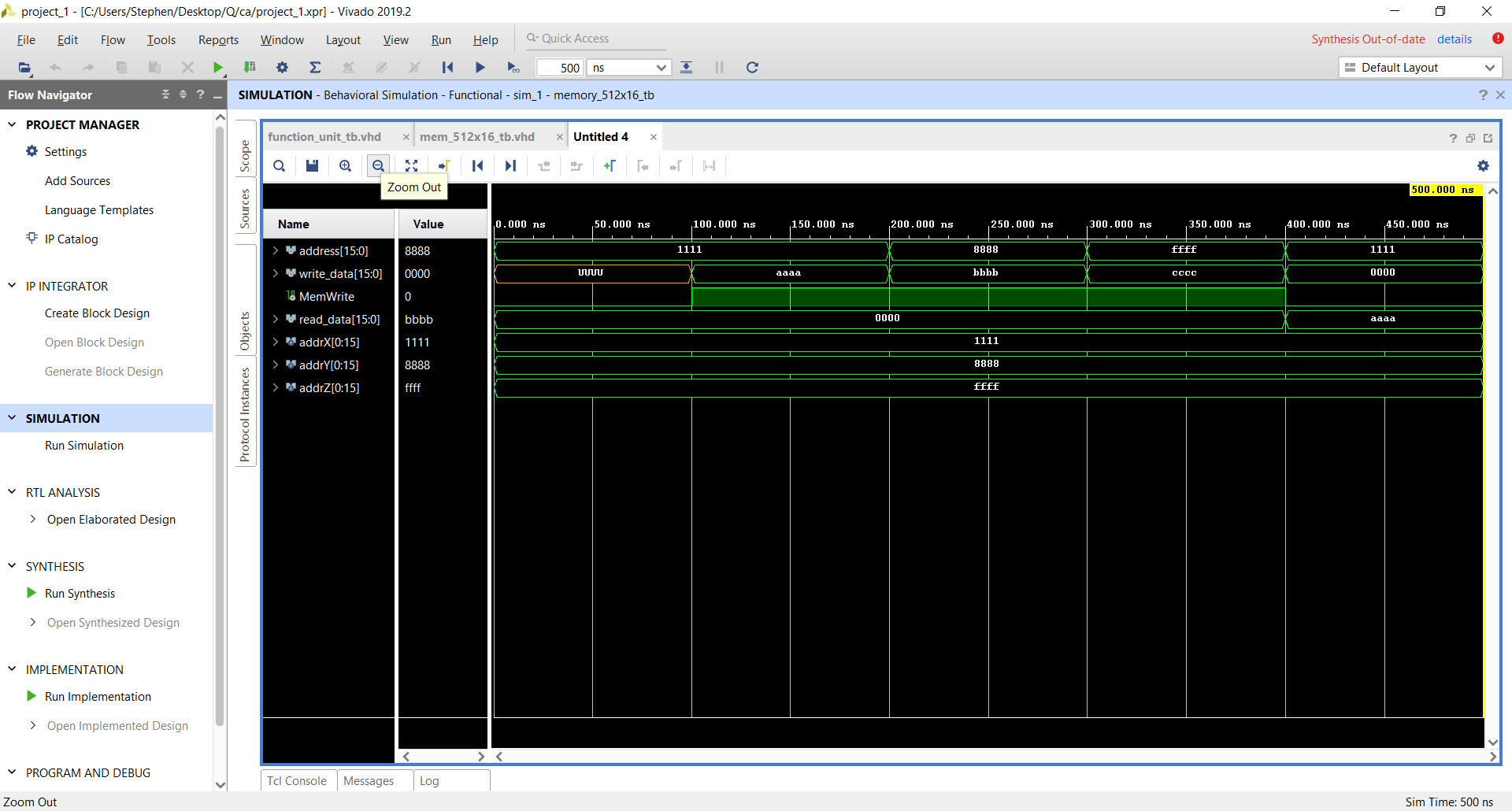
****

**Datapath (Explanation for this is included in the last assignment, the difference with this version is that there are more input values from the Control Unit)**

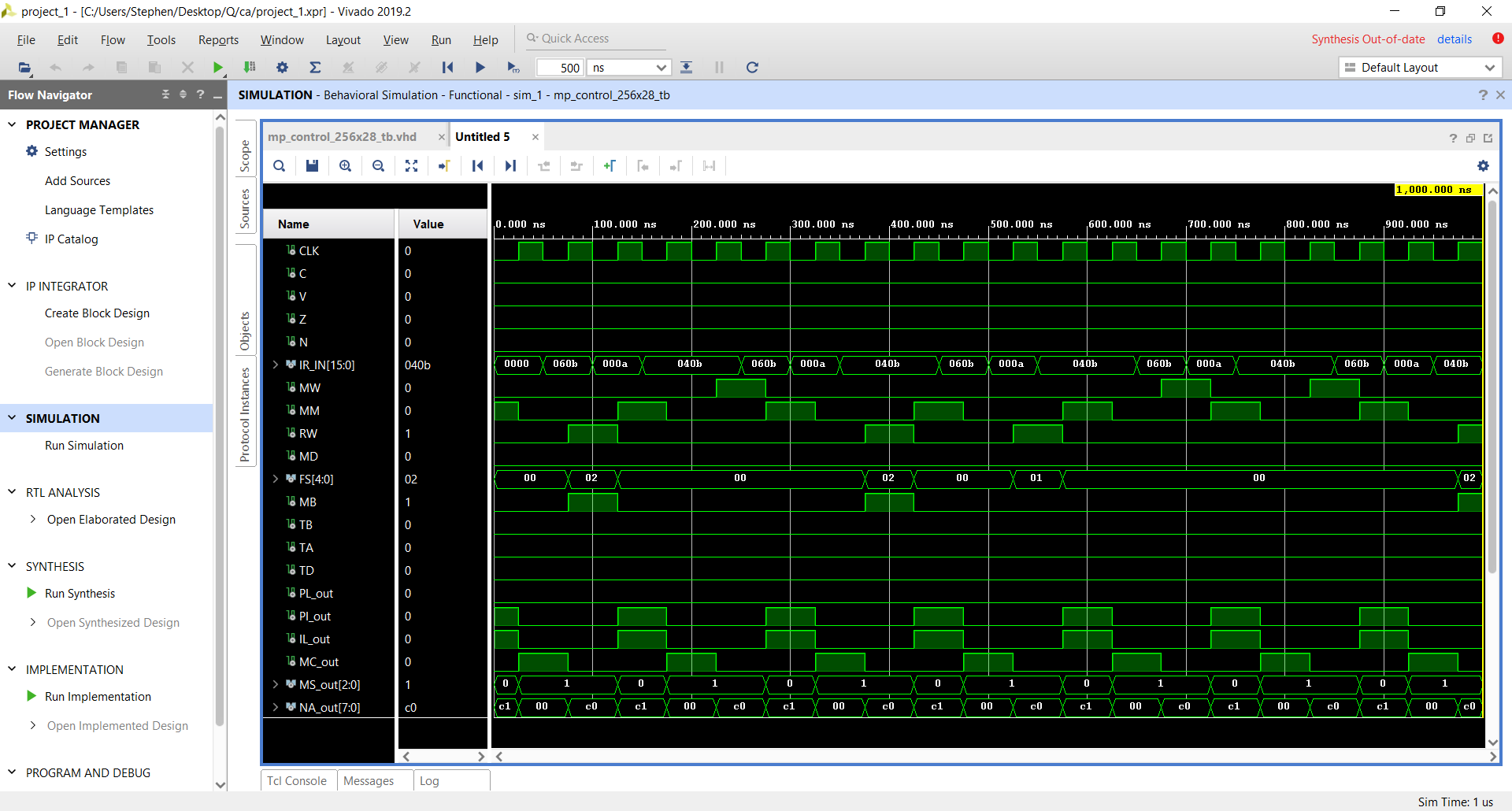
****

**Function Unit** and **Logic Unit** are mostly unchanged from the last assignment too, so I won’t include their testbenches here.

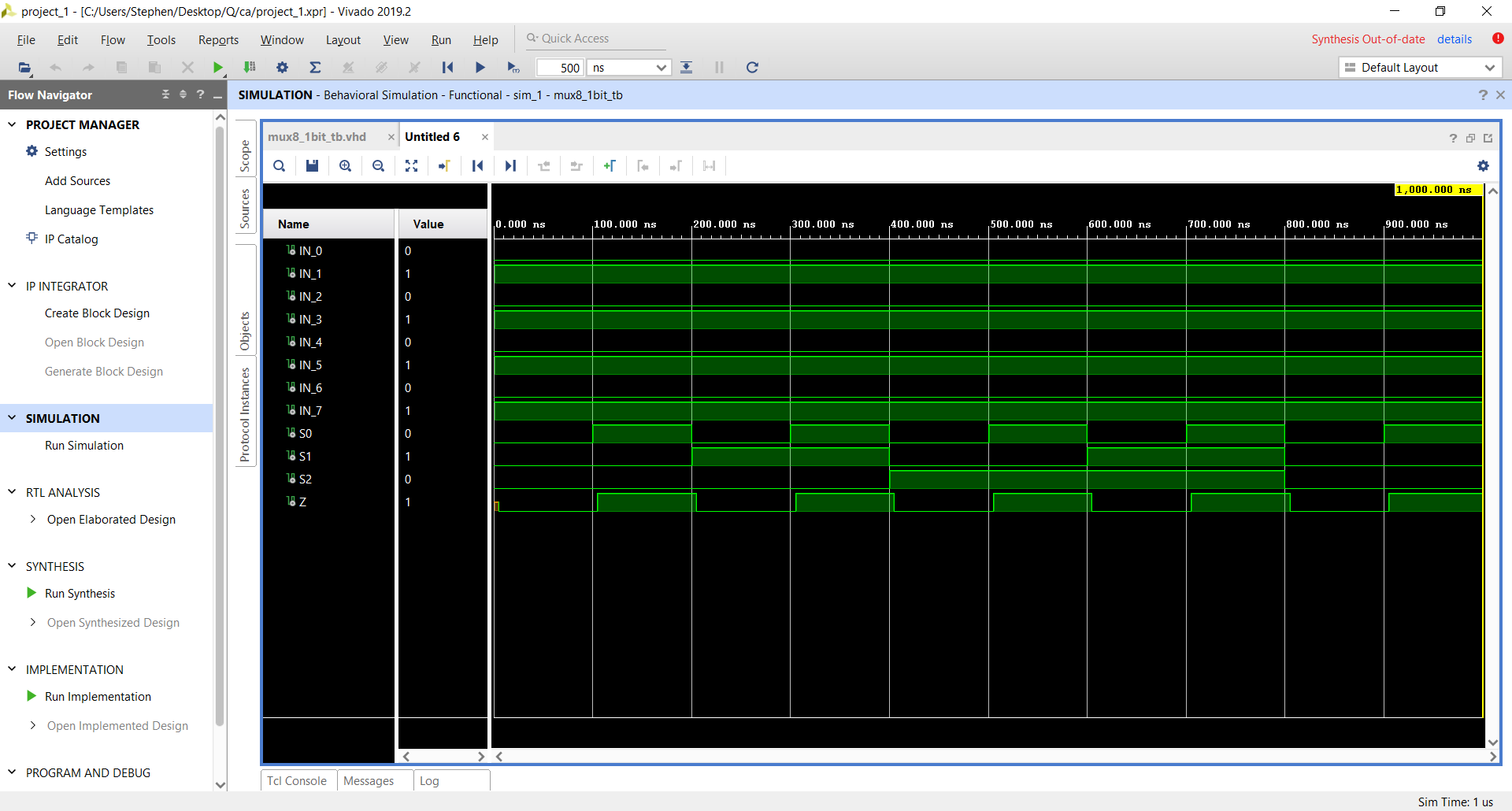
**Memory:**

****

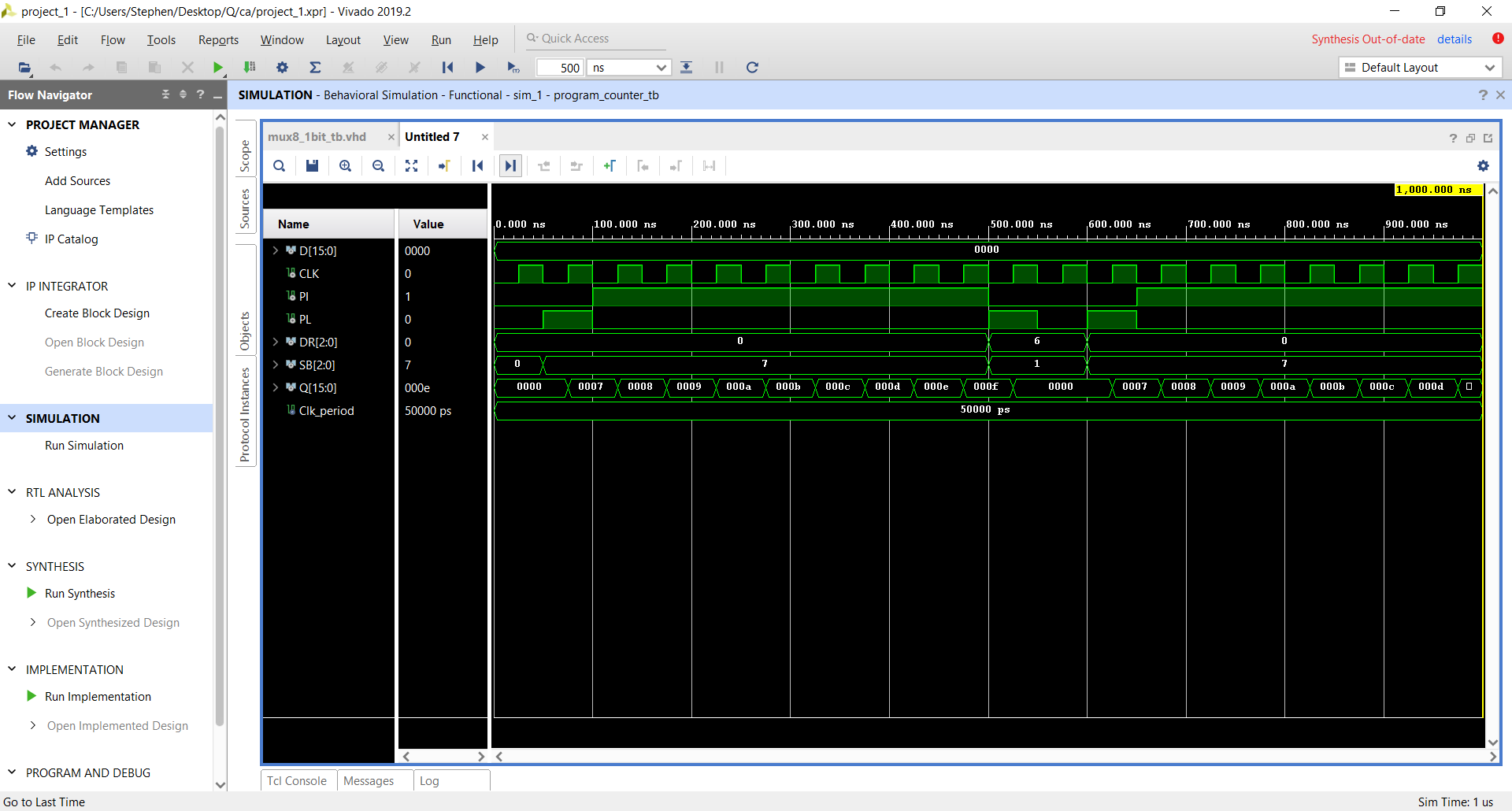
**Control Unit:**

****

**Mux 8 1-bit**

****

**Program Counter**

****

**Register File** and **16-bit Register**, **RCA** and **Shifter** were included in previous assignments, so I’ll leave it out here.