Final FPGA and HDL

The objective is to load these instructions into memory and execute them on the FPGA board:

1. LW 5 201 1001_0101_1100_1001
2. LW 6 202 1001_0110_1100_1010
3. ADD 7 5 6 0000_0111_0101_0110
4. SW 7 203 1010_0111_1100_1011
5. LI 8 250 1000_1000_1111_1010
6. SUB 4 8 5 0001_0100_1000_0101
7. SW 4 204 1010_0100_1100_1100
8. SRA 3 7 0111_0010_0111_0000
9. XOR 2 3 4 0100_0010_0011_0100
10. SW 2 205 1010_0010_1100_1101

Store suitable initial values at memory locations 201 and 202

Output: Show the values in memory locations 203, 204 and 205 in a 7-segment display once all instructions are executed.

At the memory locations 201 and 202 I have stored the integers 32 and 64 respectfully so our output should be:

1. LW r5 <- x0020
2. LW r6 <- x0040
3. ADD r7 r5 r6 == r7 <- x0060
4. SW r7 -> MEM[203] x0060 -> MEM[203]
5. LI r8 <- 250 // REMEBER THIS IS SIGNED EXTENDED IMMEDIATE MEANING xFFFA
6. SUB r4 r8 r5 == r4 <- xFFFA - x20 goes to xFFDA
7. SW r4 -> MEM[204] xFFDA -> MEM[204]
8. SRA r3 r7 == 0011 0000 x30
9. XOR r2 r3 r4 == 0000 0000 0011 0000 XOR 1111 1111 1101 1010//1111 1111 1110 1010 xFFEA
10. SW r2 -> MEM[205]

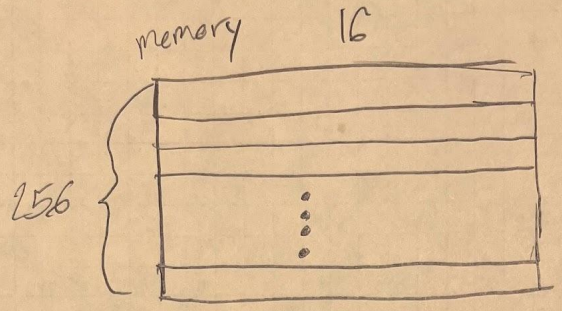
Starting off, for ALU control I made a 3 bit wide signal to execute the following:

ADD = 000;
SUB = 001;
AND = 010;
OR = 011;
XOR = 100;
NOT = 101;
SLA = 110;
SRA = 111;

Next all I had to do was build each of the modules and thoroughly understand the controller to assert correct signals:

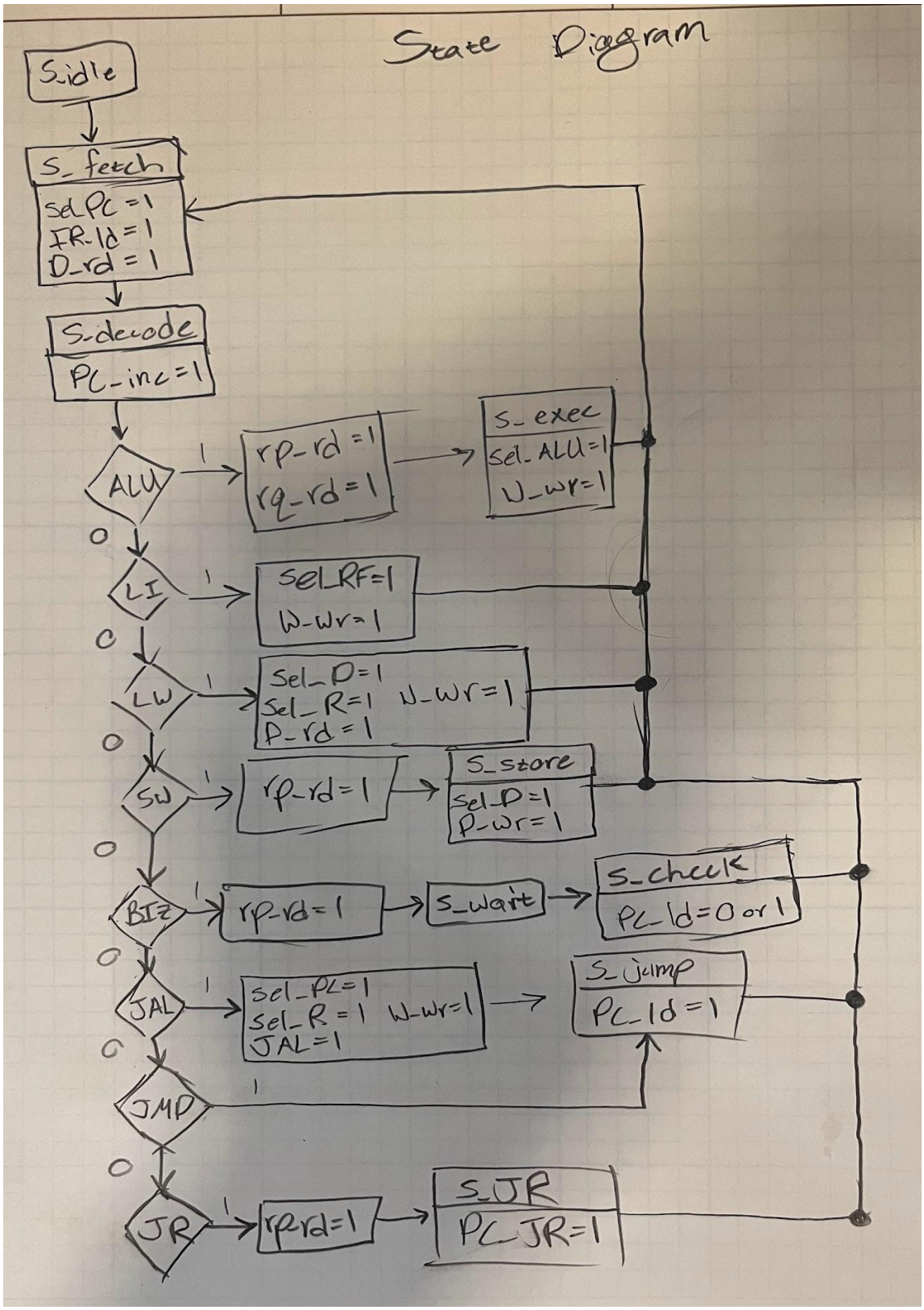
ALU Operations

a+b
a-b
a+b
a|b
a^b
~a
a<<1
a>>1



Control signals	Action
W_wr	loads destination register address
Rp_rd	loads Source reg add
Rq_rd	loads target reg add
alu_s	Conords ALU functioning
RF_s	MUX selects RF_W_Data, R_data, or ALU output to RF
IR_ld	loads R_data to IR
PC_ld	adds IR to PC
PC_clr	reset PC
PC_inc	increment PC by 2?
D_addr	MUX selects PC or IR Dir
D_r	read memory
D_wr	Write to memory
RF_RP_zero	zero flag

After understanding ALU and control signals I created a diagram showing the progression of states in StateTransitions.xlsx and here:



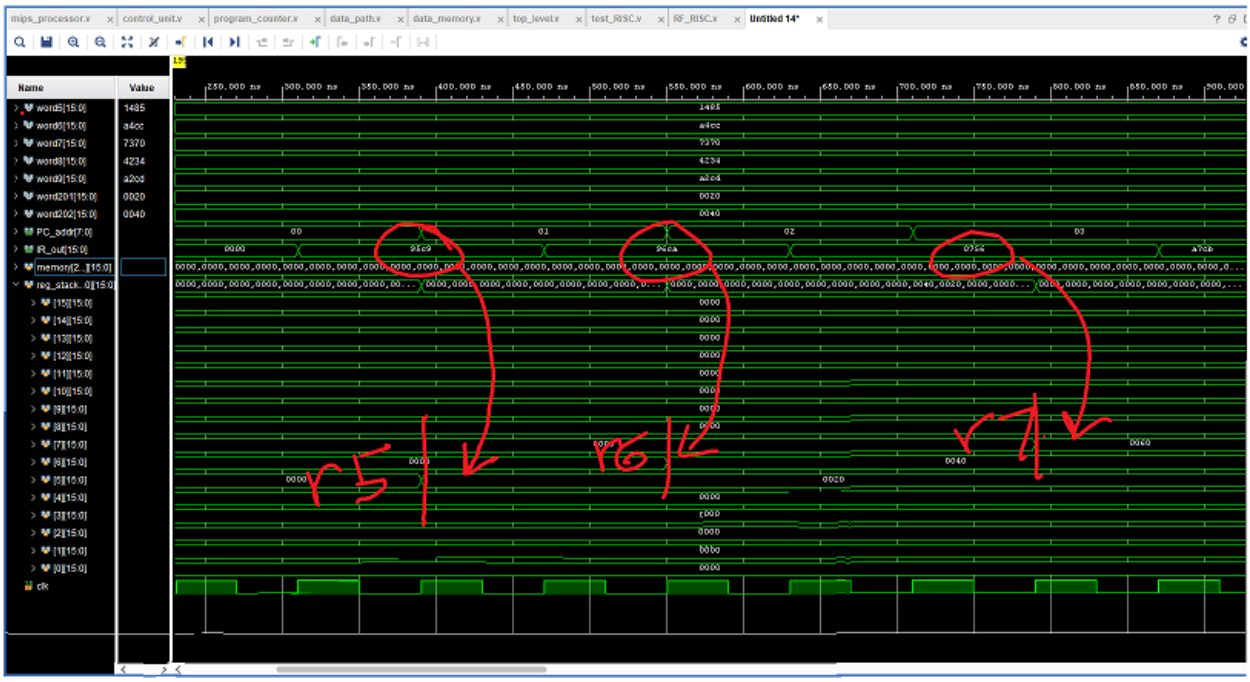
These are the corresponding clock cycles for each instruction
Green signifies start of instruction and red signifies end of instruction returning to FETCH:

	1 cc	2 cc	3 cc	4 cc
ALU (ADD, SUB, AND, OR, XOR, NOT, SLA, SRA)	FETCH Fetches instruction and loads into IR	DECODE Increments PC and loads Rp and Rq buses, at same time opcode is sent to ALU for computation	EXECUTION Selects ALU datapath from MUX and writes to RF[W_addr]	
LI	FETCH Fetches instruction and loads into IR	DECODE Increments PC and selects RF_W_data sign extended on MUX to write to RF[W_addr]		
LW	FETCH Fetches instruction and loads into IR	DECODE Increments PC and writes MEM[D_addr] to RF[W_addr] register		
SW	FETCH Fetches instruction and loads into IR	DECODE Increments PC and loads our DEST part of instruction code into Rp address bus	EXECUTION Writes the input W_data into MEM[D_addr]	
BIZ/BNZ	FETCH Fetches instruction and loads into IR	DECODE Increments PC and loads RF[RF_Rp_addr]	WAIT RF_Rp_zero MUST be asserted during this time to check if Rp output is 0 or not	CHECK Load offset into PC if RF_Rp output is zero or not; else return to FETCH
JAL	FETCH Fetches instruction and loads into IR	DECODE Increments PC and loads PC into RF[RF_Rp_addr]	EXECUTION Writes to RF and offsets PC	
JMP	FETCH Fetches instruction and loads into IR	DECODE Increments PC and decodes instruction	EXECUTION offsets PC	
JR	FETCH Fetches	DECODE Increments PC	EXECUTION Loads the	

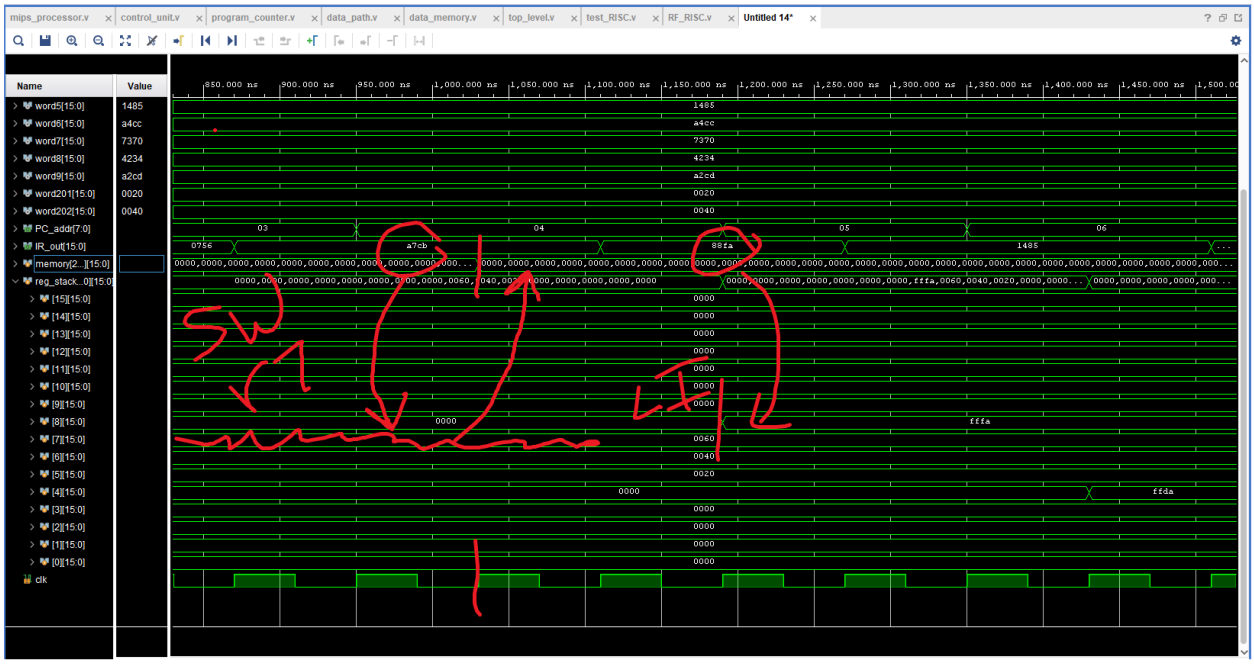
	instruction and loads into IR	and loads RF[RF_Rp_addr]	registers contents into PC without adding offset	

Given all previous information + tables I coded in the states and asserted signals to get results shown below. If you wish to see my code it is archived in RISC_Processor.xpr.zip.

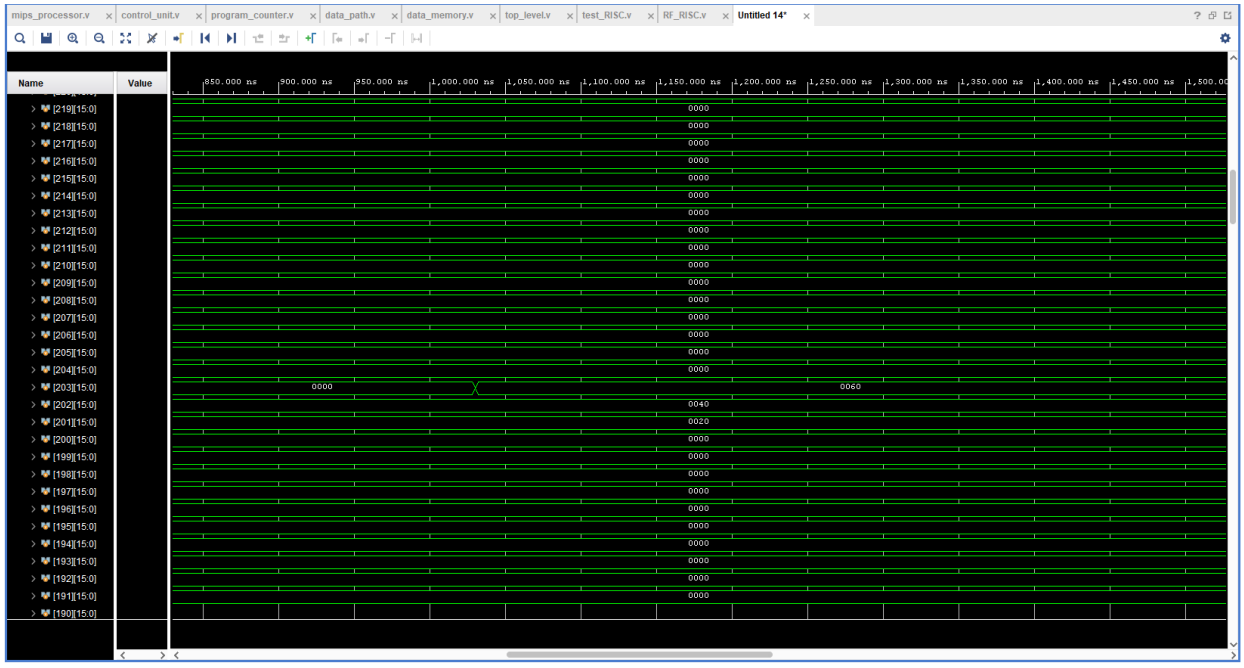
Now here is a test bench of the instructions to be demonstrated:
Here you can see the changes instructions 1 through 3 make on the RF register stack.



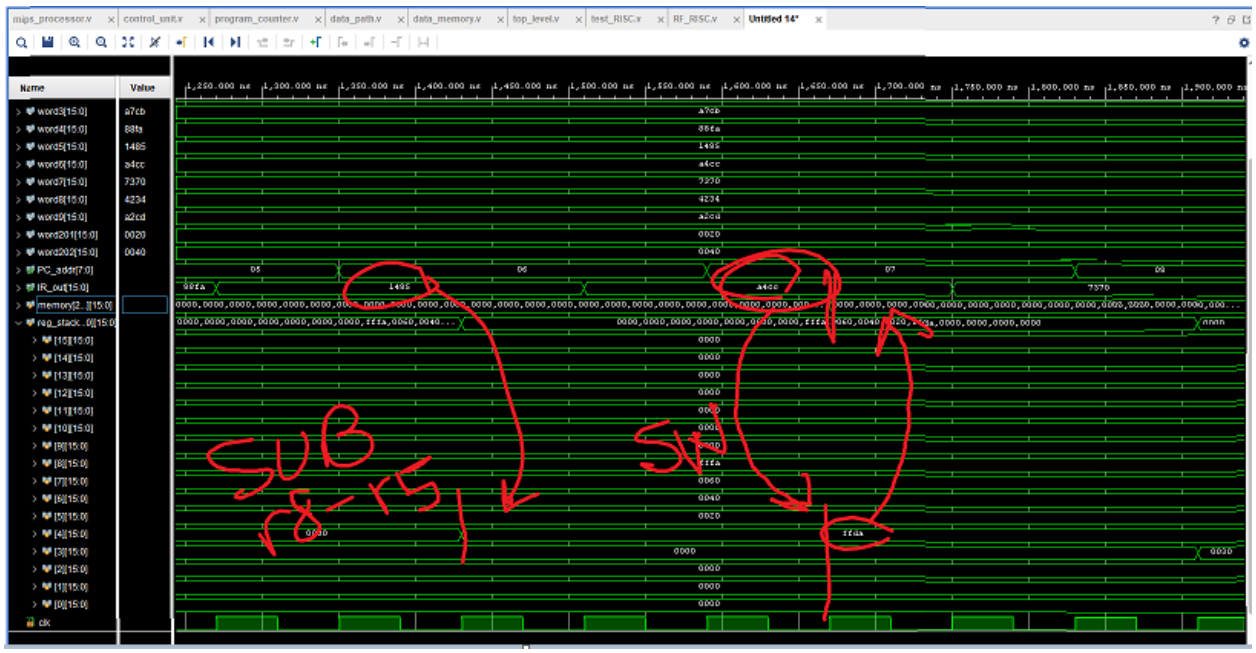
Next we can see the SW instruction and LI instruction



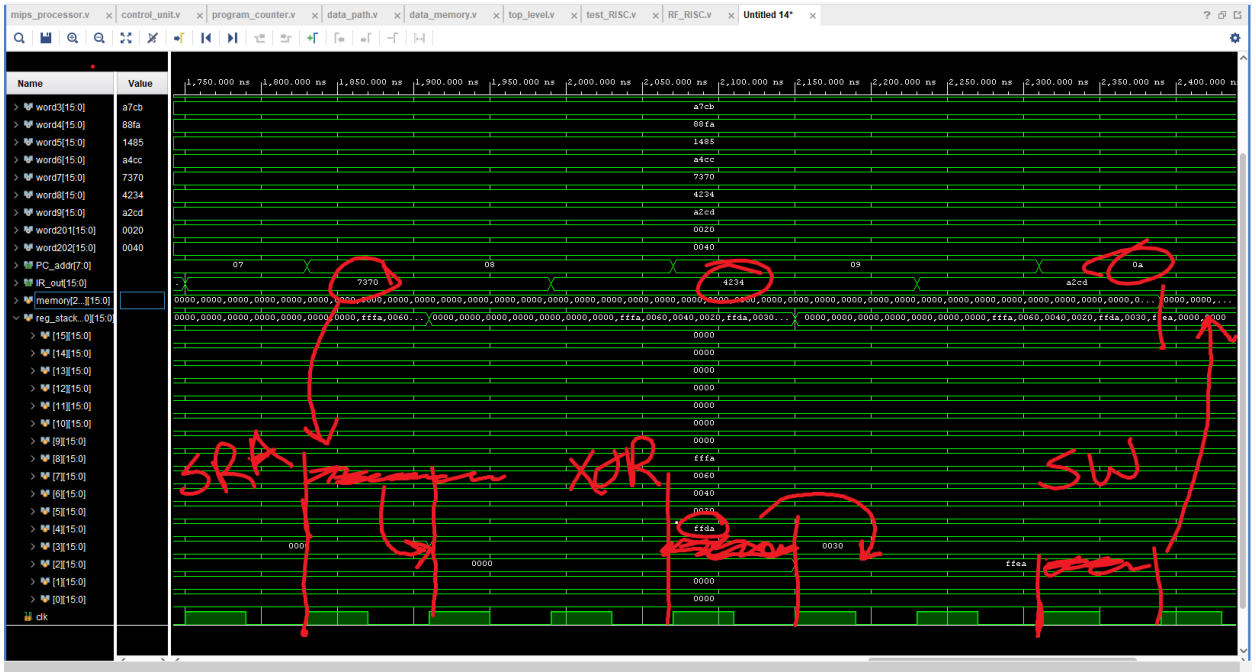
Update memory stack now shows x060 in MEM[203]:



SUB and SW instruction:

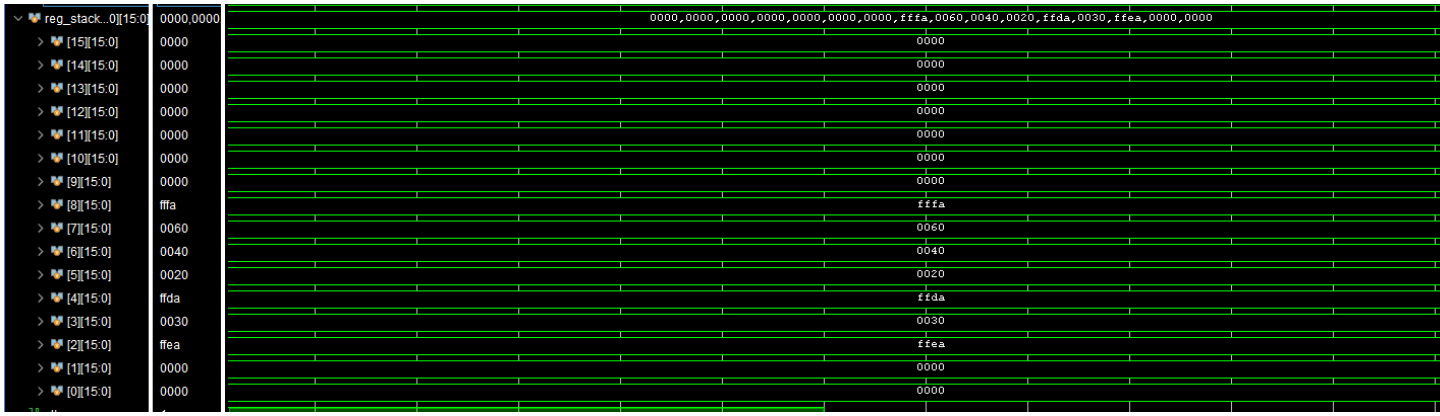


8,9,10 instructions

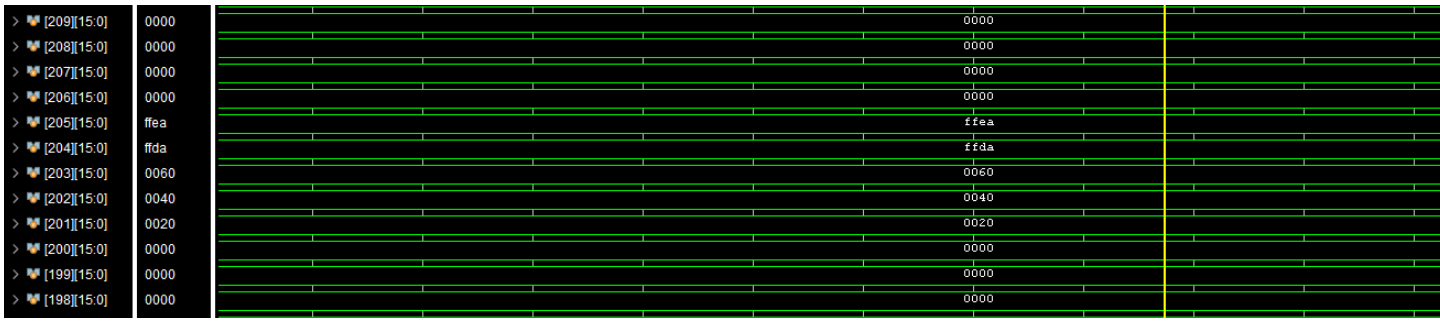


Final updated registers:

RF

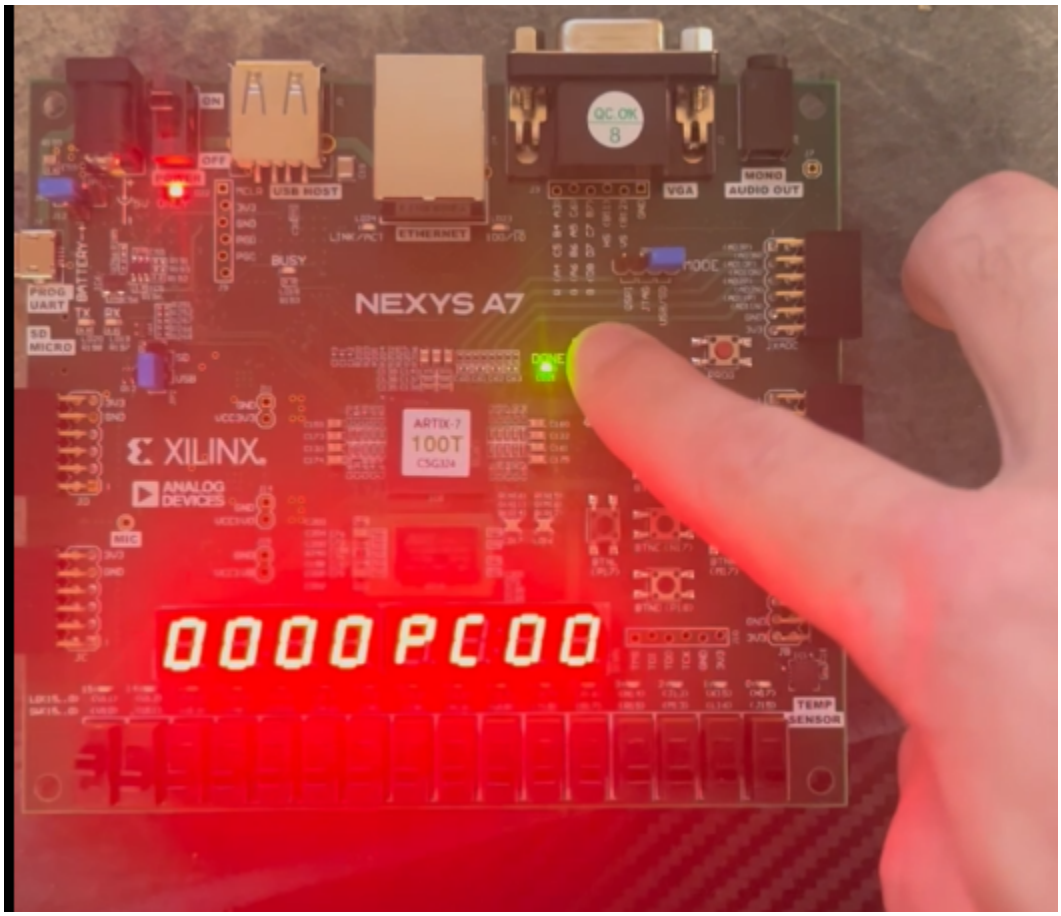


MEM

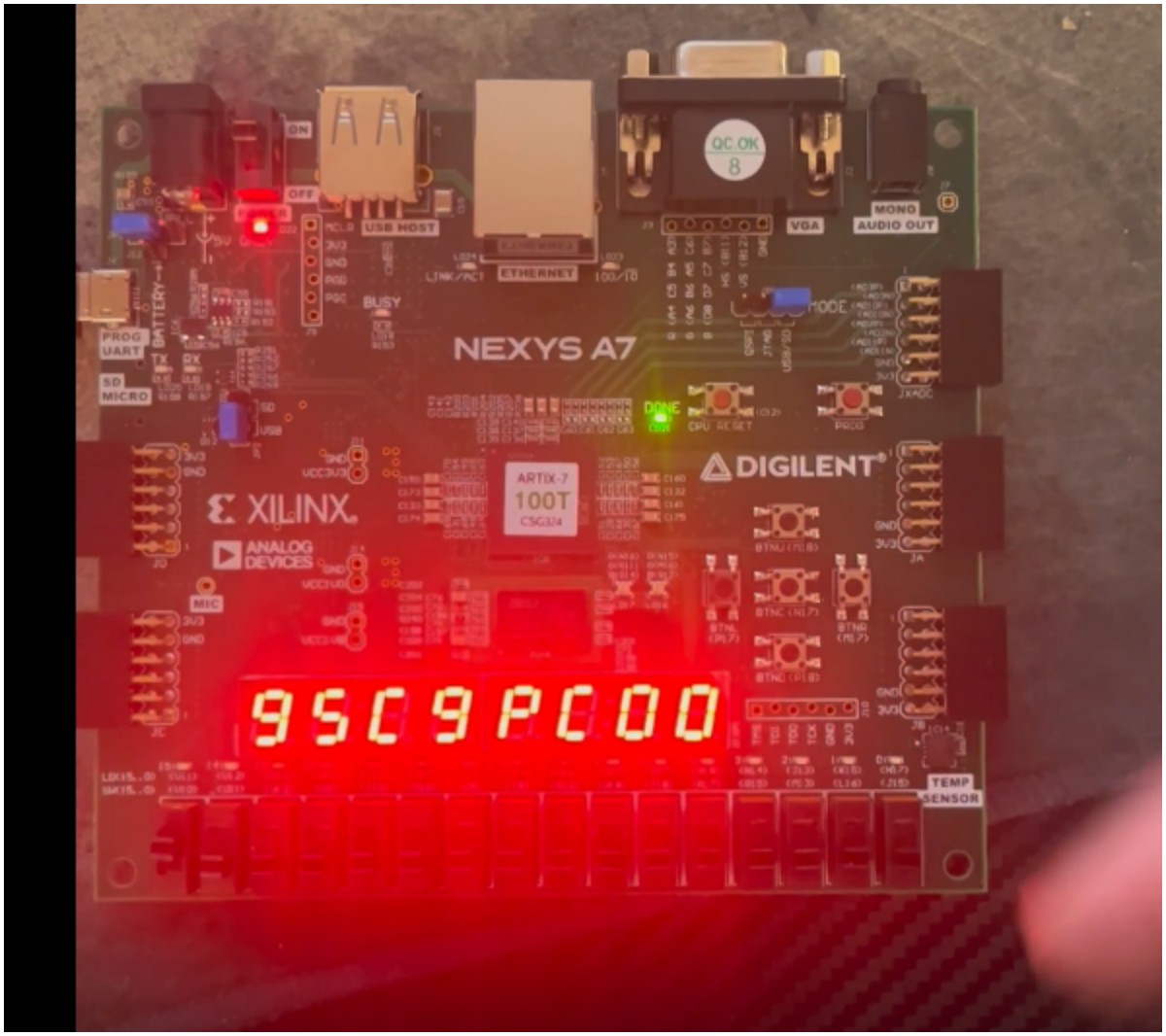


Now a Demo on the FPGA board:

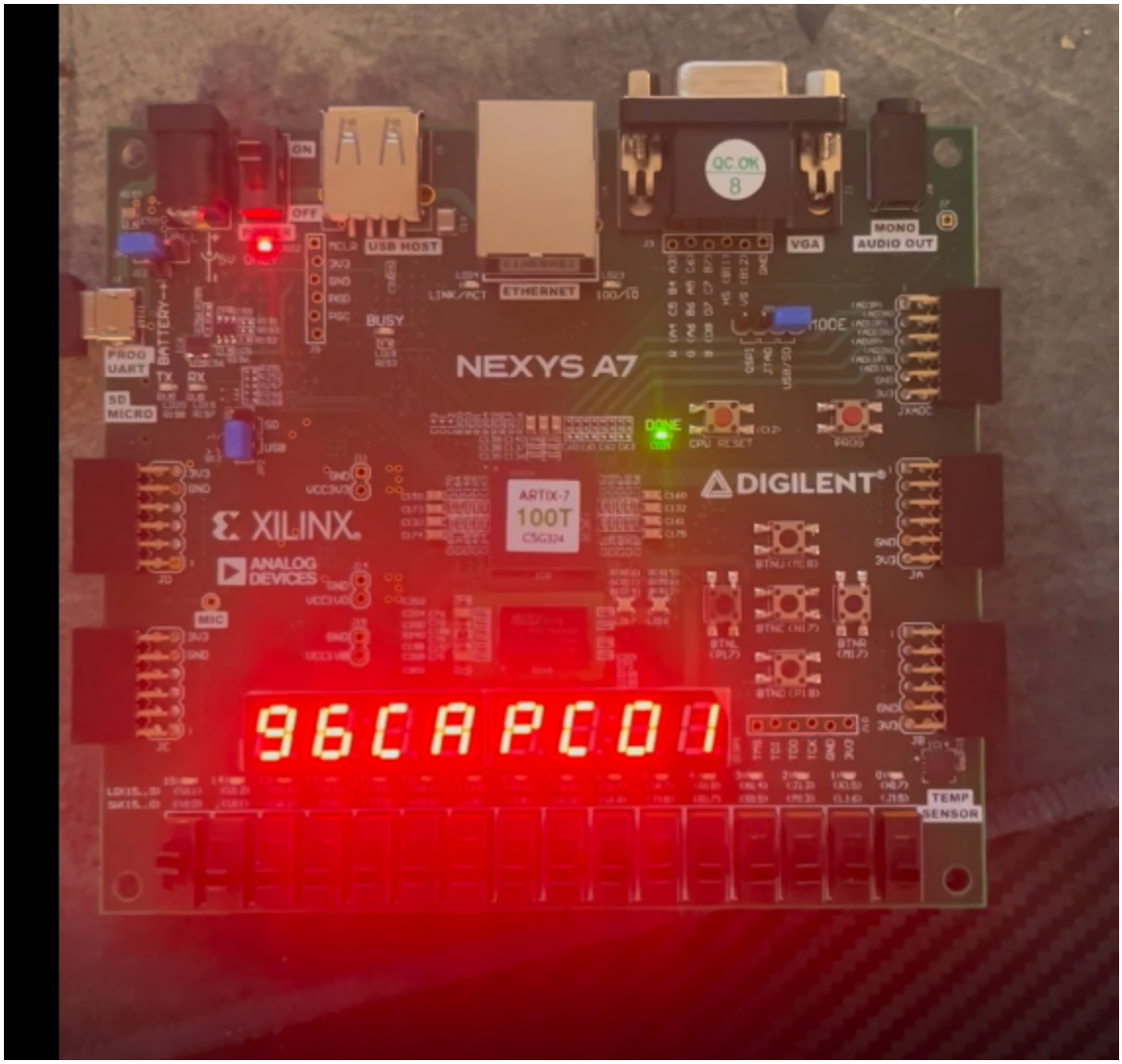
First we can see the PC start at 00 after reset and show the IR (to the left of PC) being 0000 because this is before the first fetch.



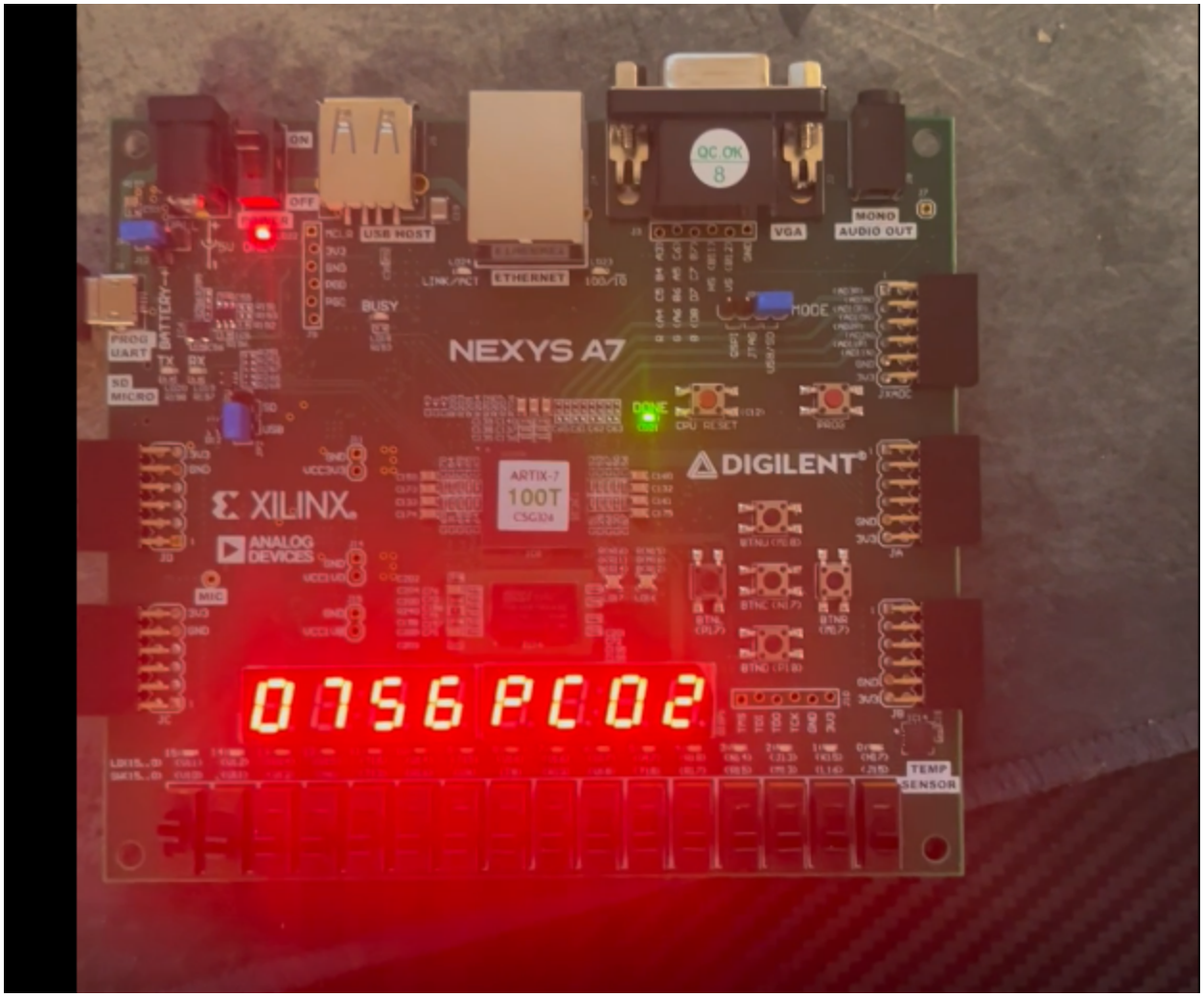
Next it fetches from MEM[x00] returning first instruction:



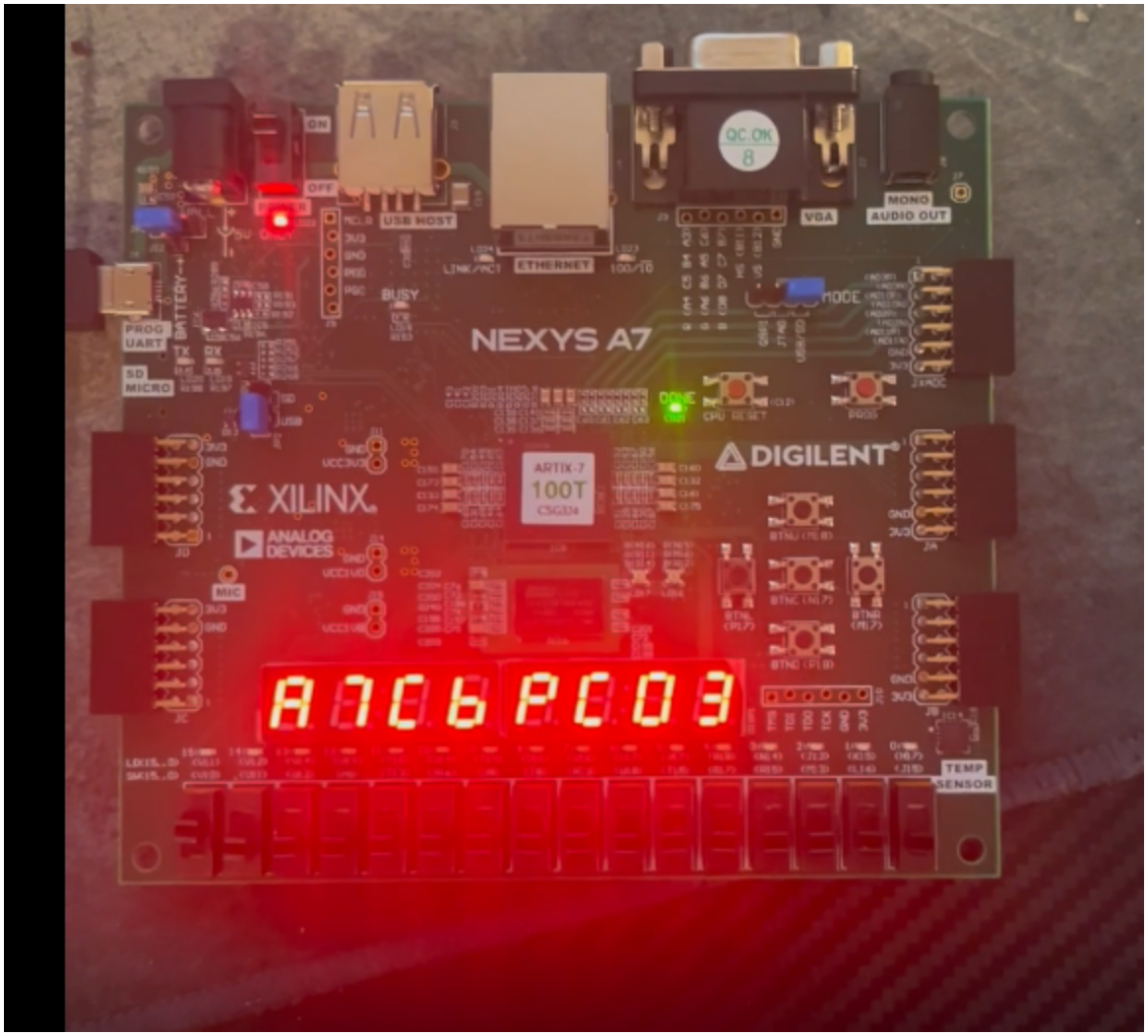
Increments > Decodes > then fetches next



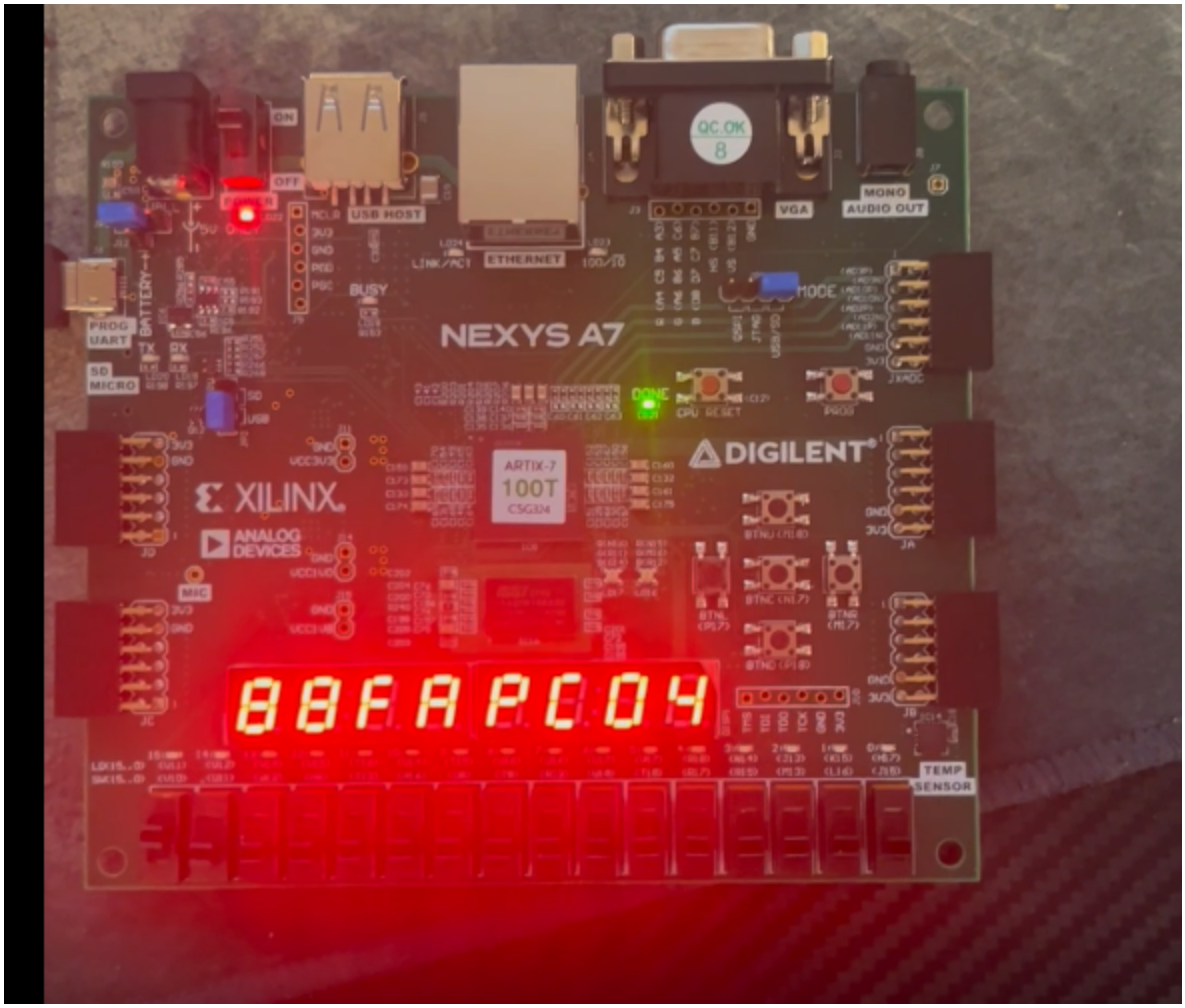
Next MEM address:



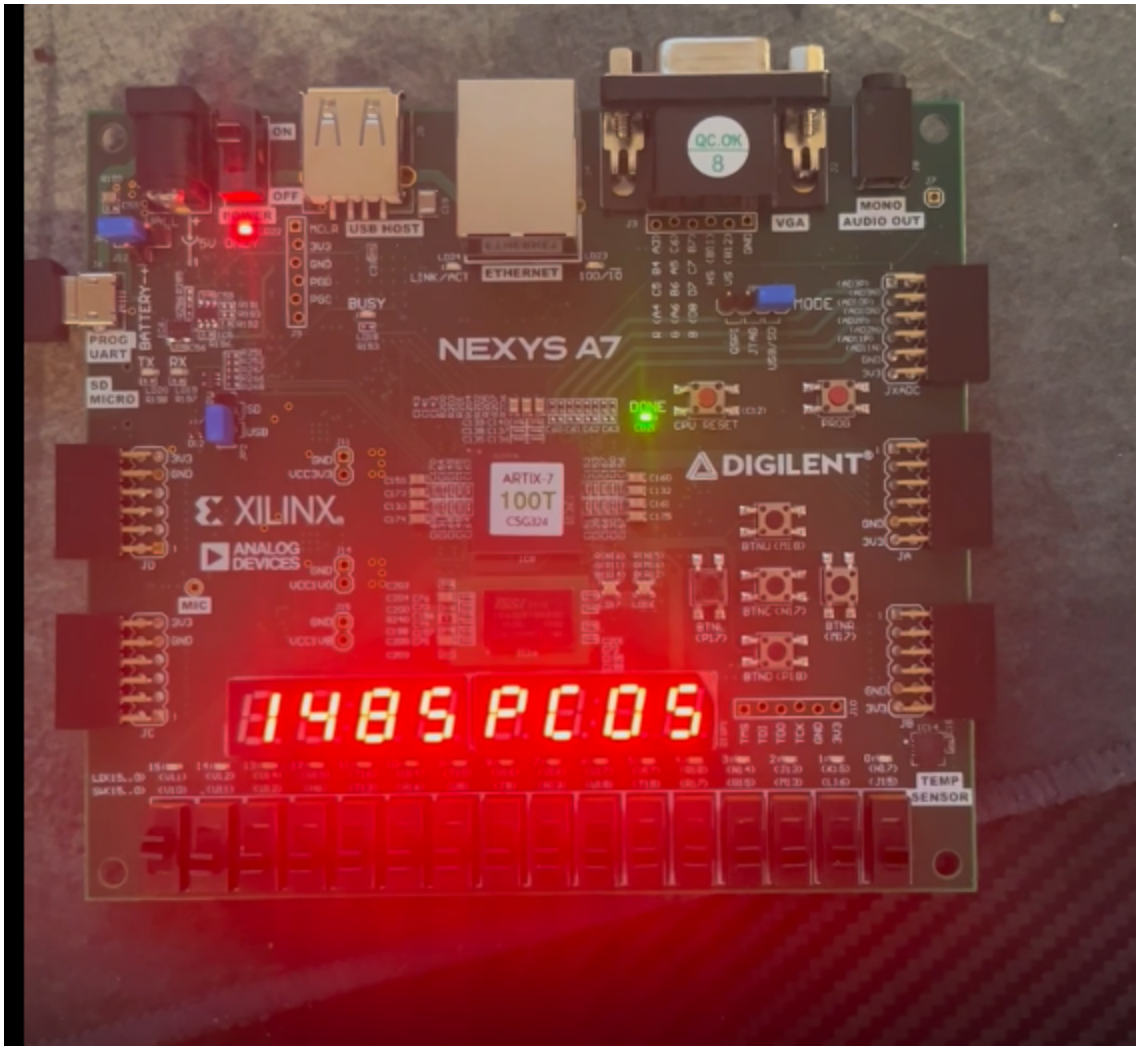
Next MEM address:



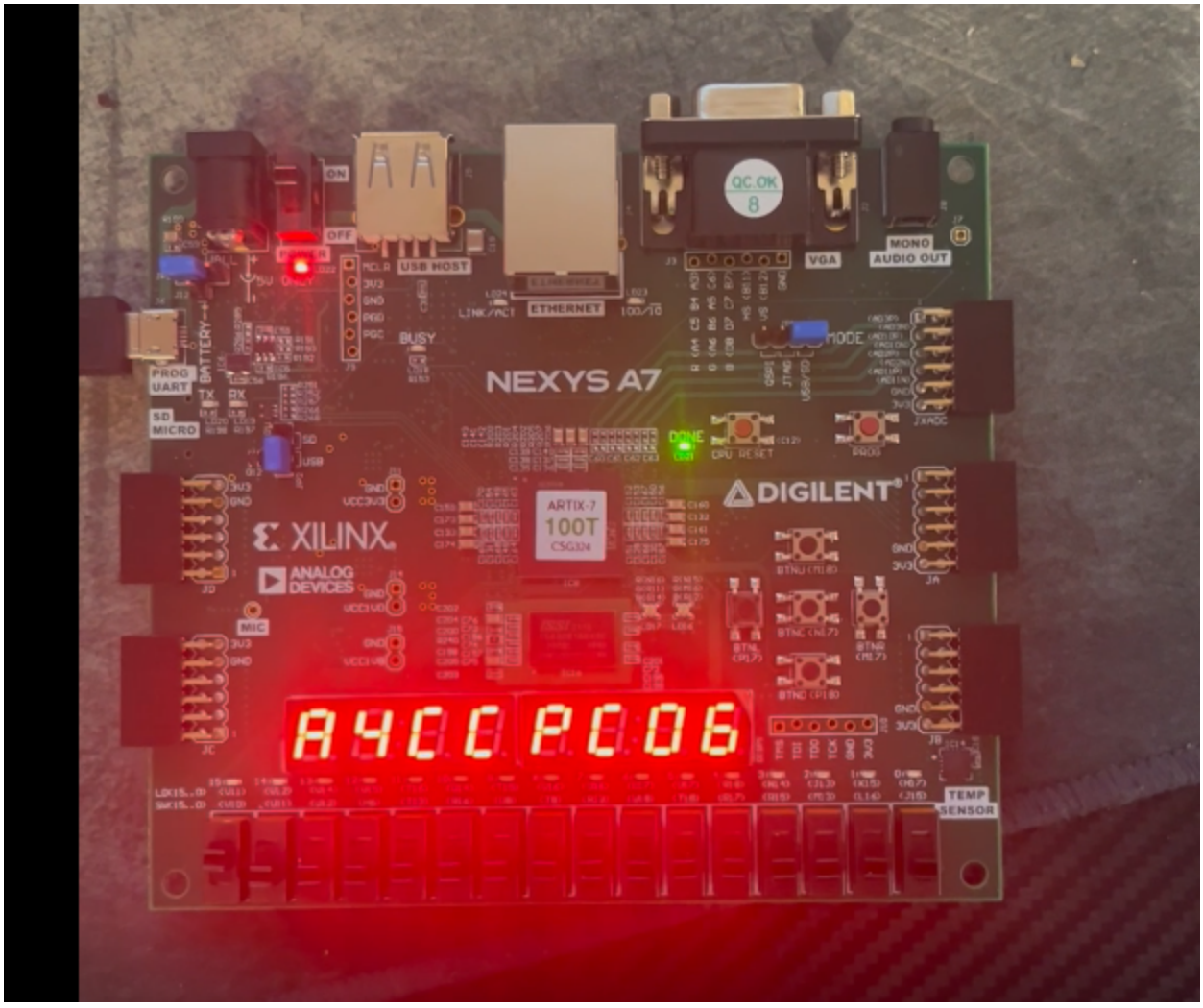
Next MEM address:



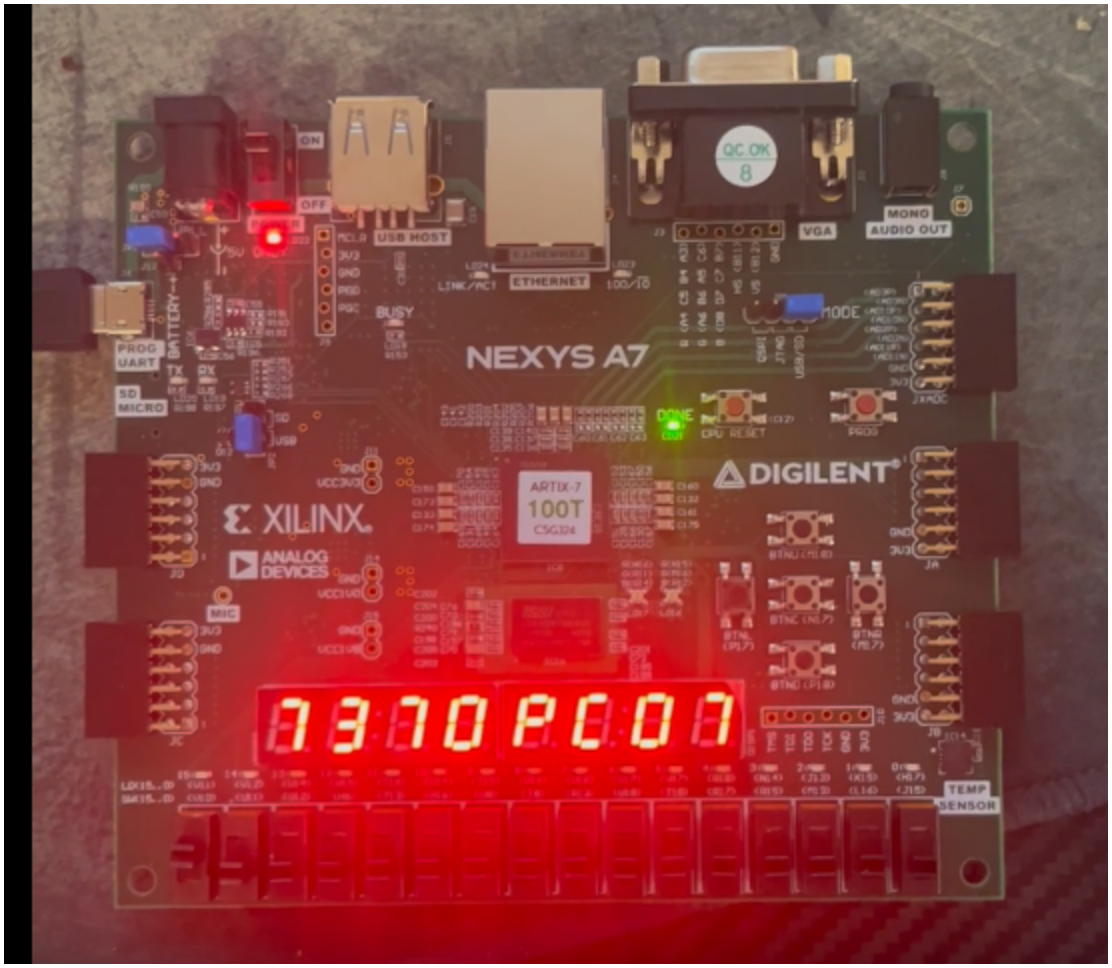
Next MEM address:



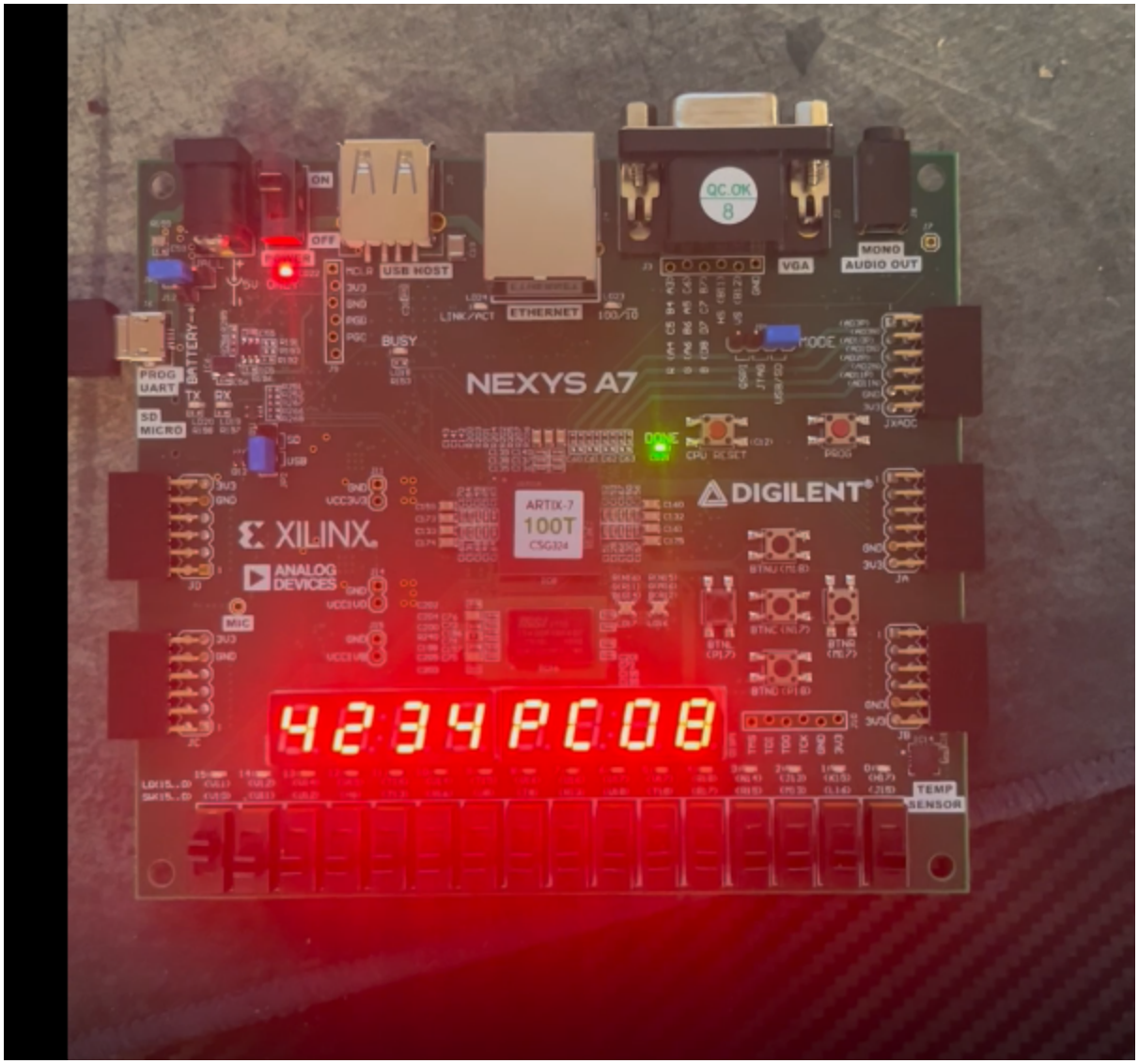
Next MEM address:



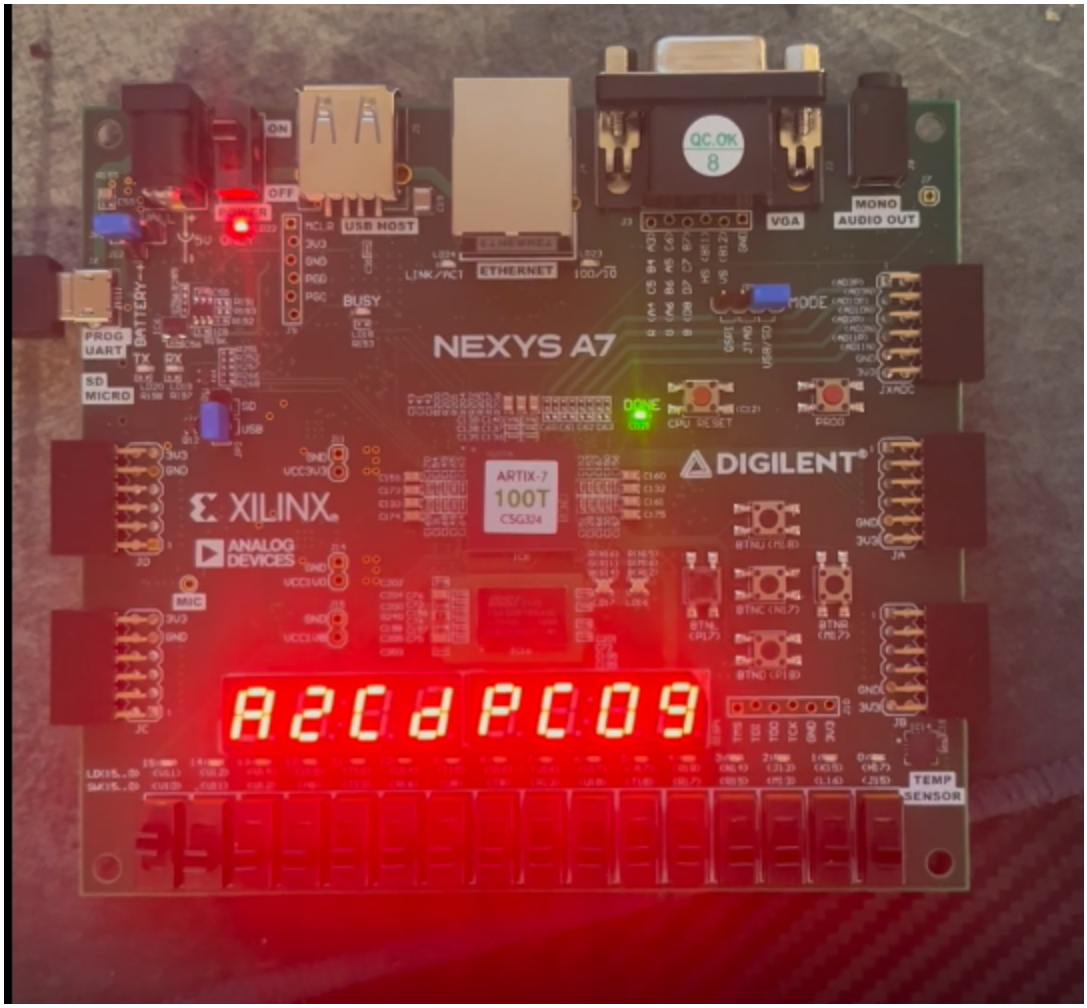
Next MEM address:



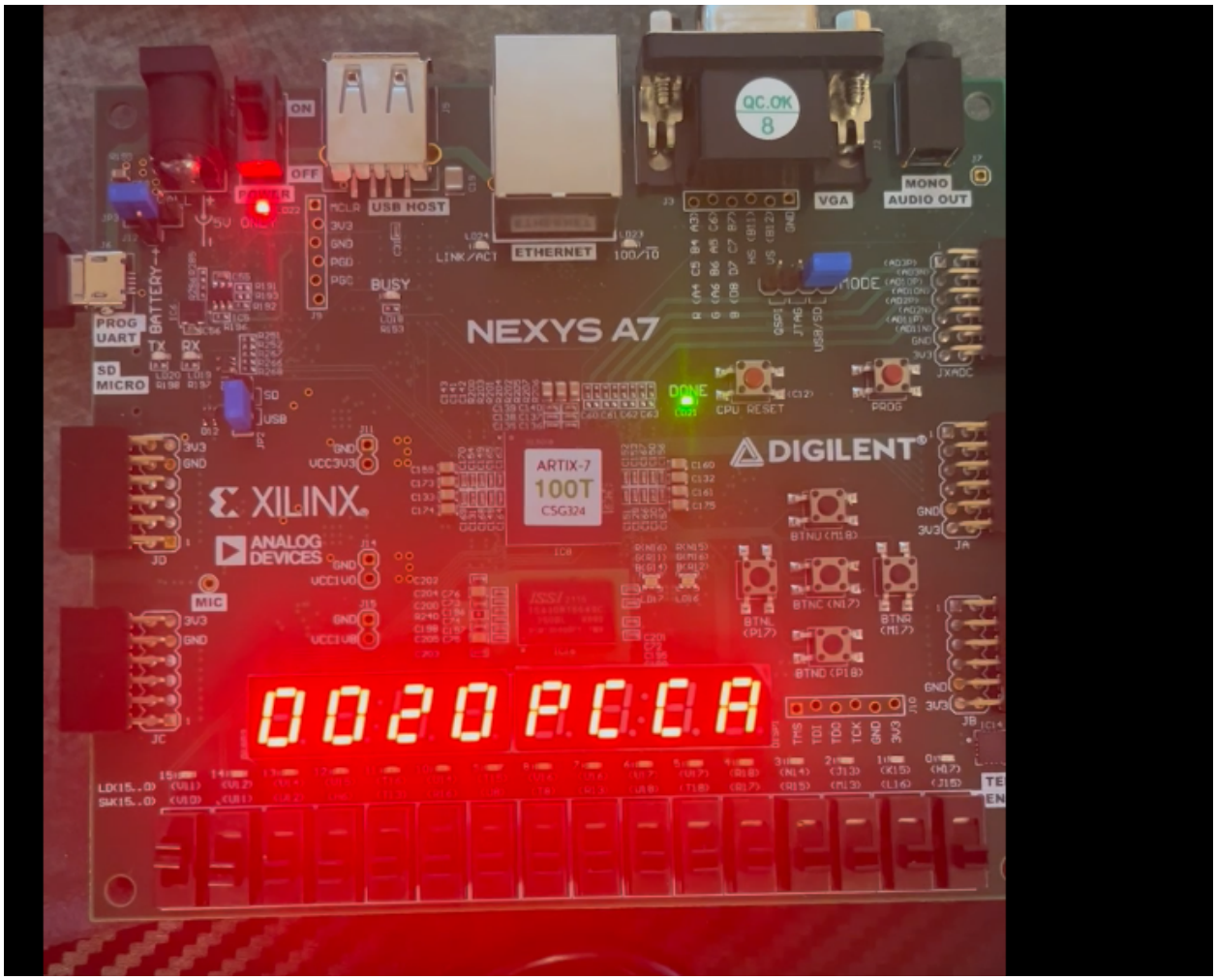
Next MEM address:



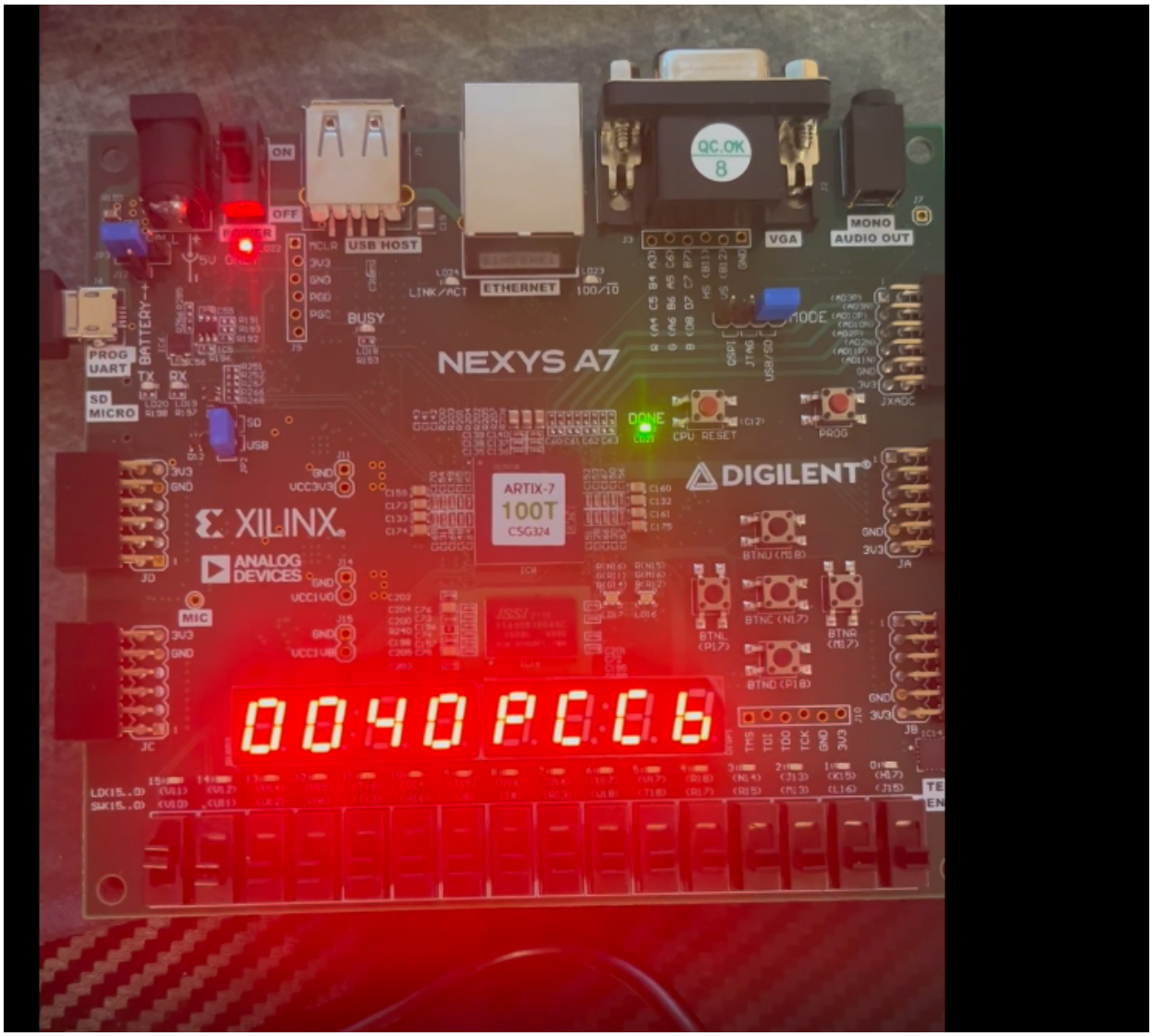
Next MEM address:



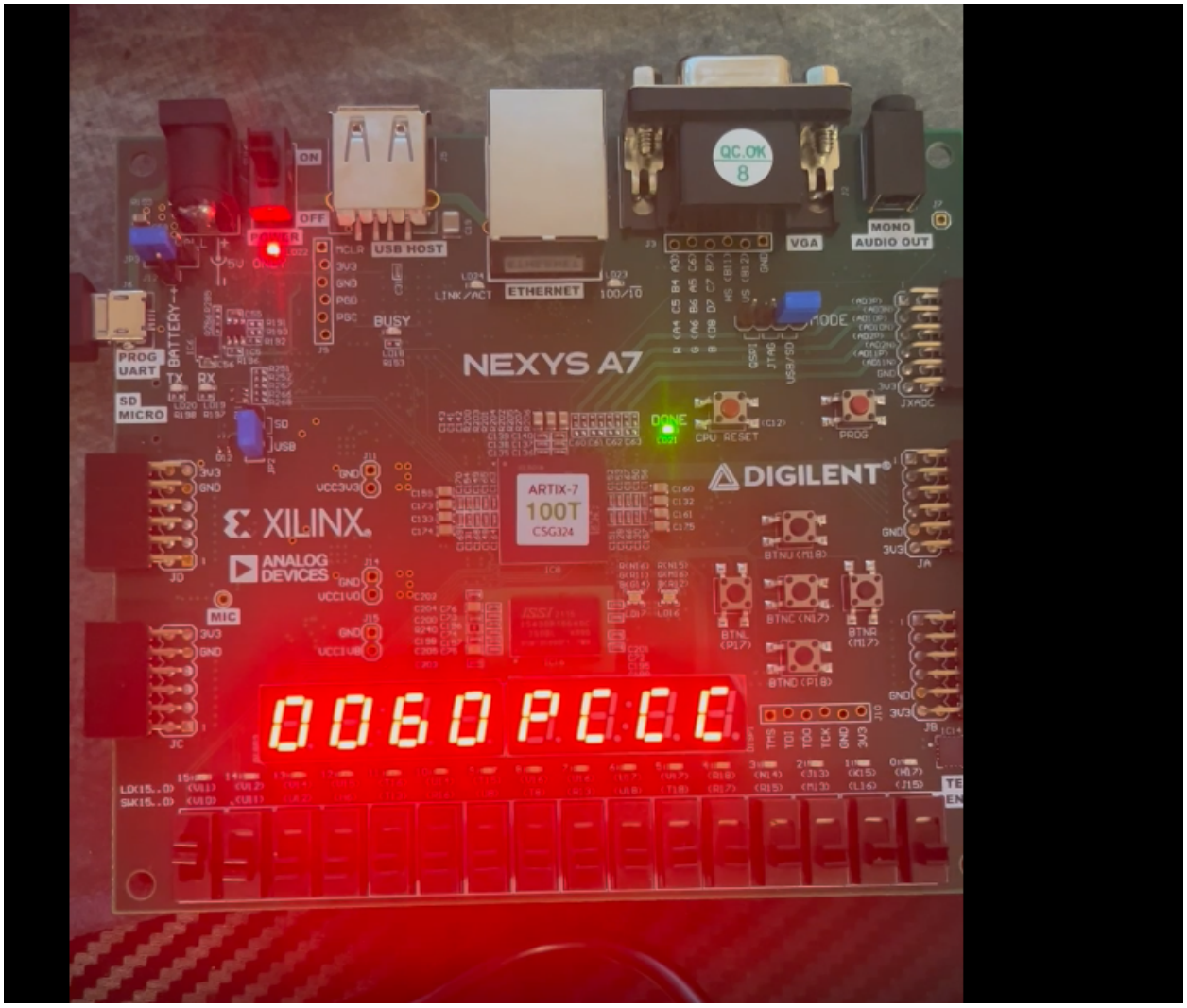
MEM[201] address:



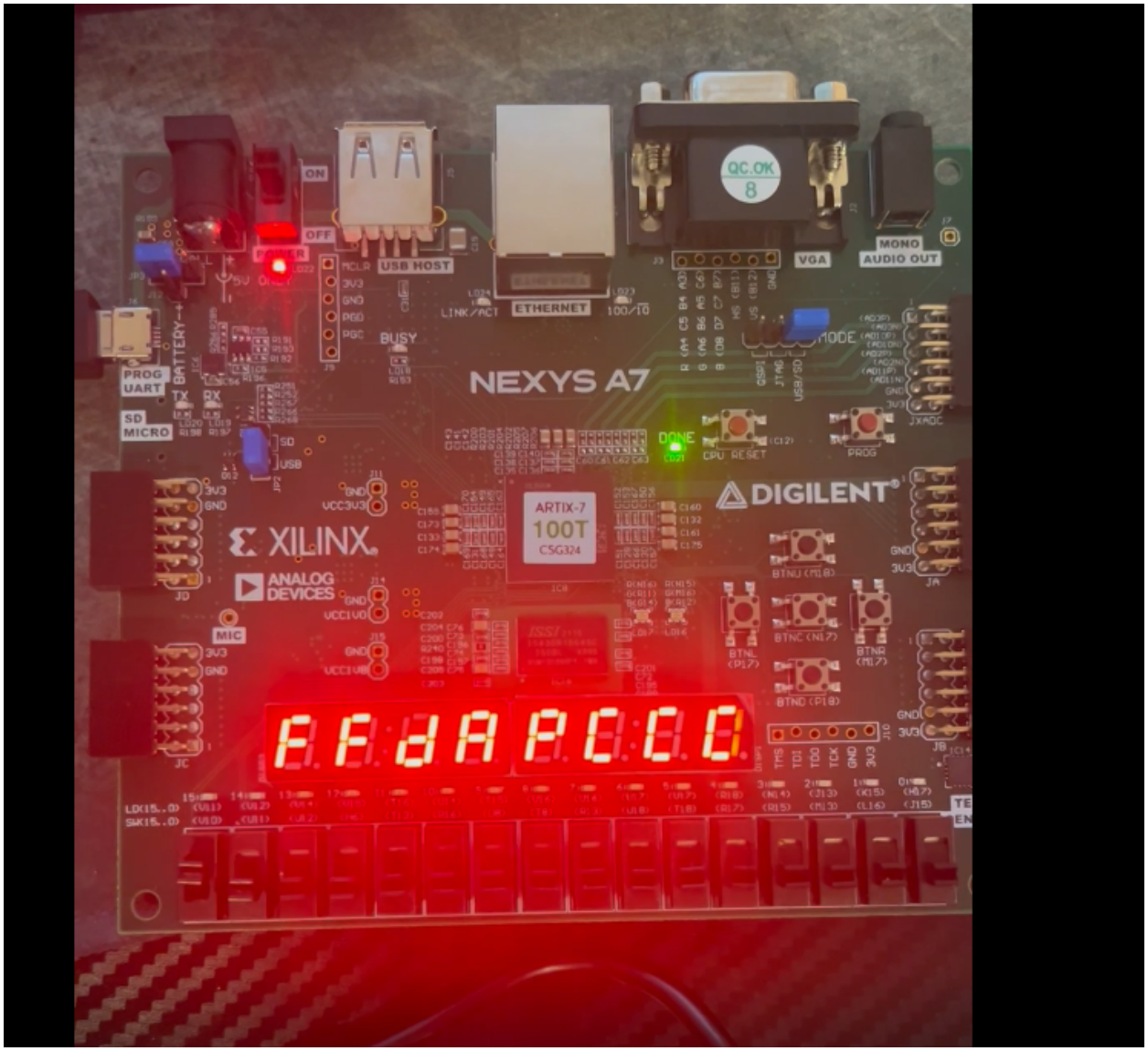
MEM[202] address:



MEM[203] address:

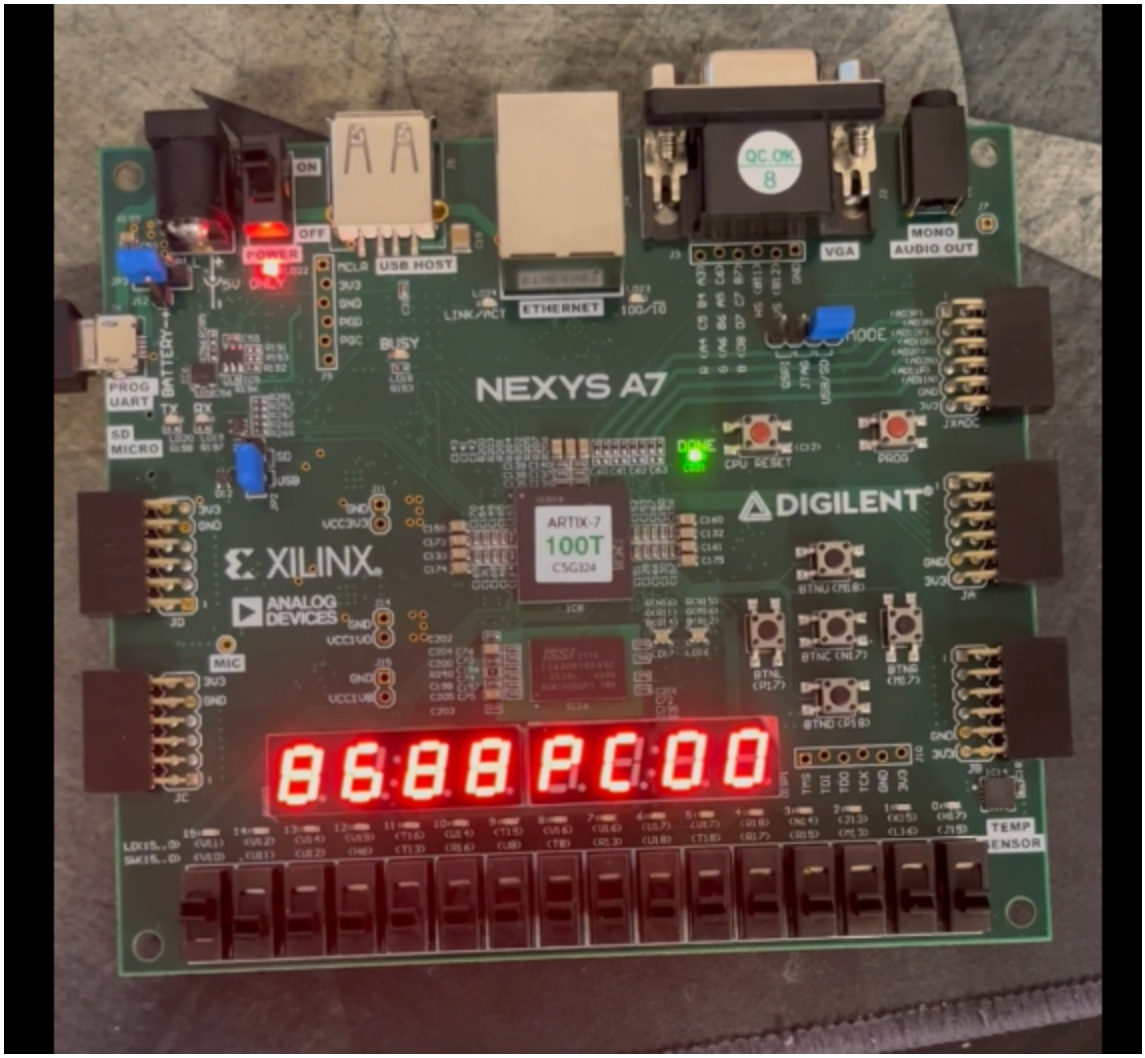


MEM[204] address:



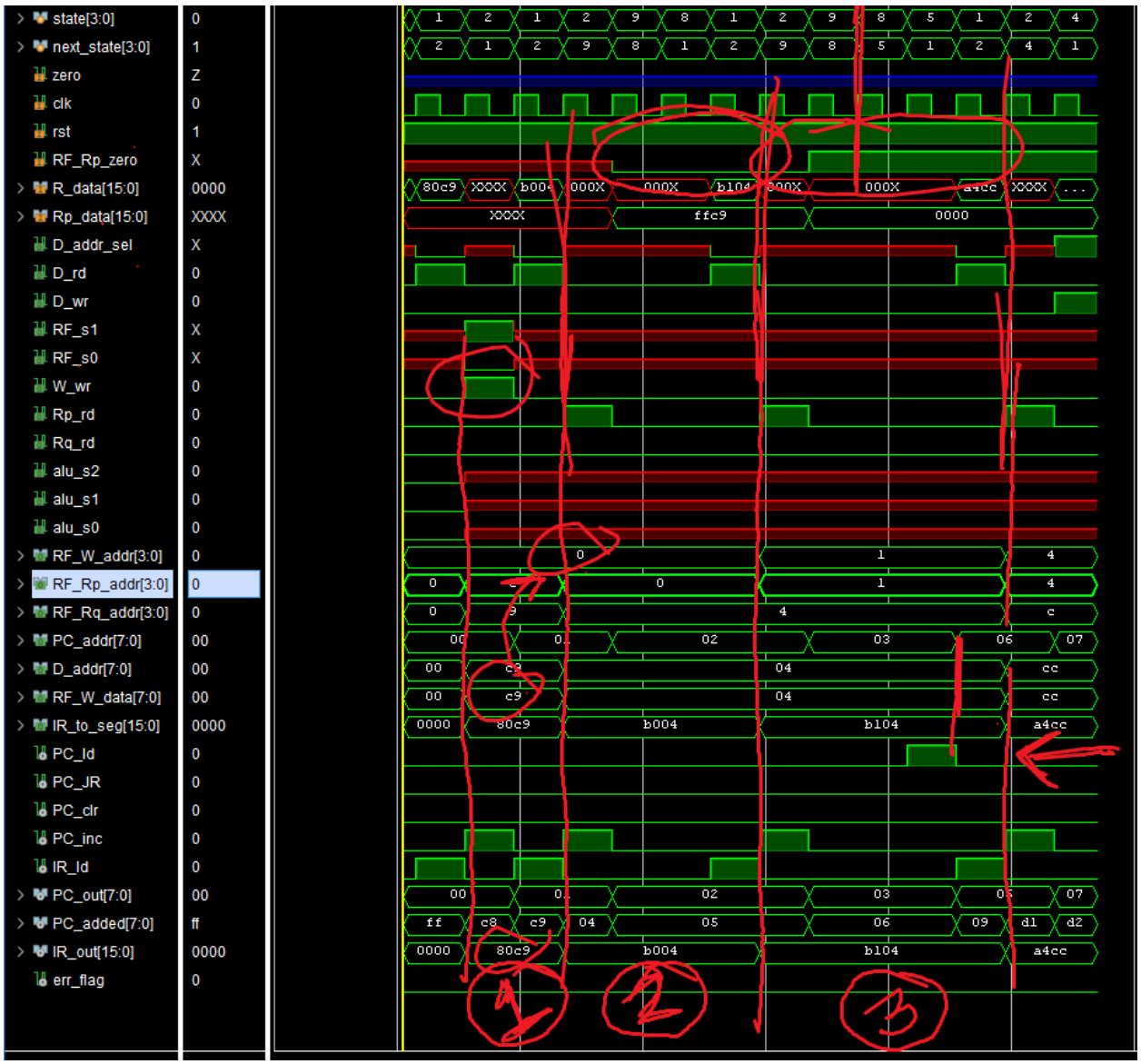
NOTE:

When PC gets to xFFDA this MEM address is loaded into IR causing it to jump due to 1111 opcode being a JR command. The JR command is pointing to register xD meaning 1101 or r13. In r13 is x0000 resulting in PC resetting to x00 so we don't get to see the PC pull out MEM[205] into the IR:



Reset to 0, working as intended.

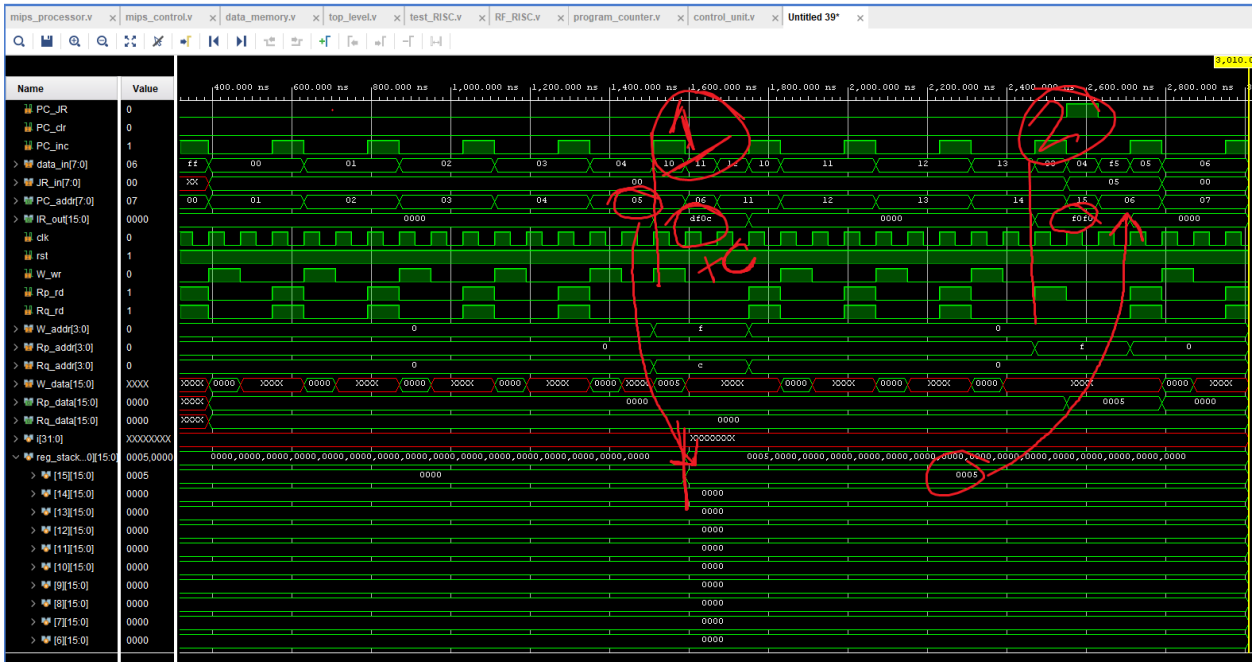
So as the implementation given covers most instructions (ALU operations, JR, LW(I), and SW) here is the rest of instructions done on a test bench:
BIZ/BNZ:



For these sections:

1. Uses instruction LI (x80c9) to load xC9 Immediate into r0
2. Uses instruction BIZ (xB004) to check if the output of r0 is zero, since it is not 0 the PC increments instead of offsetting
3. Uses instruction BIZ (xB104) to check if the output of r1 is zero, since r1 == 0 the PC then increments by current PC (x03) + offset (x04) - (x01) resulting in final PC being x06

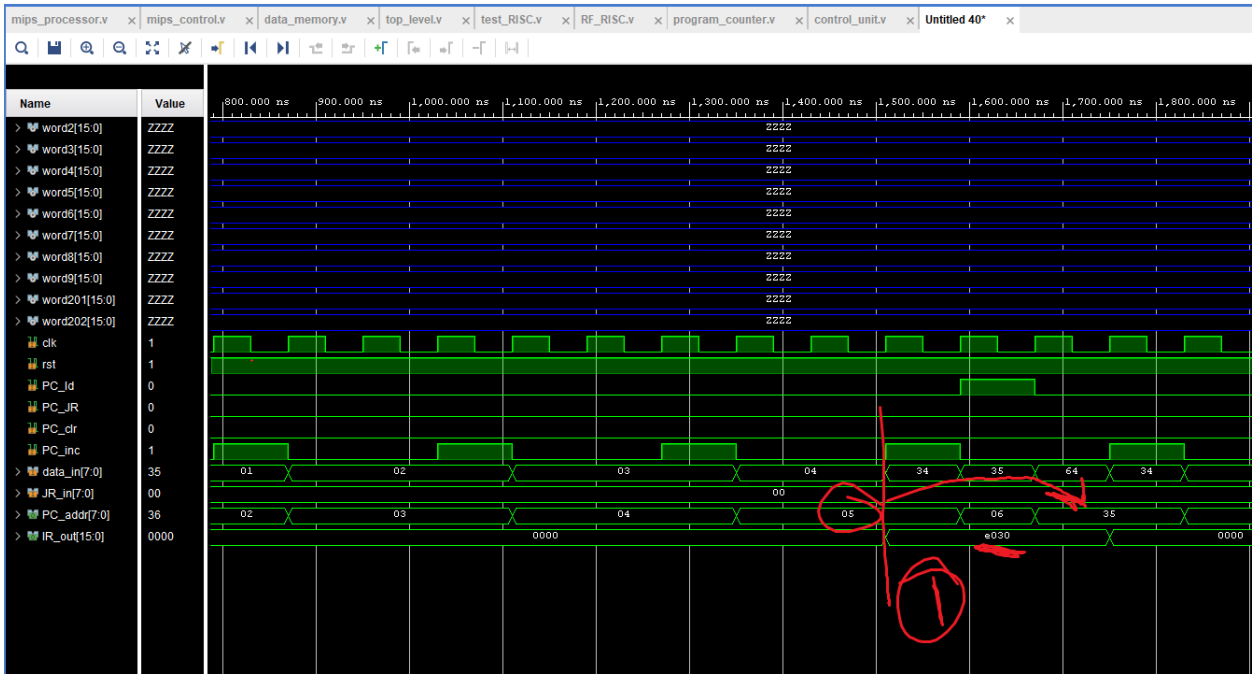
JAL/JR:



For this section:

1. Receives JAL (xDF0C) instruction to offset by +x0C and store return address in r15, offset: x06 + x0C - x01 = x11
2. Receives JR (xF0F0) instruction to return to register address stored in r15, loads register into PC and increments it by 1 to avoid recursion

JMP



1. Receives instruction to JMP (xE030) offset by x30:
   Offset = x06 + x30 - x01 = x35