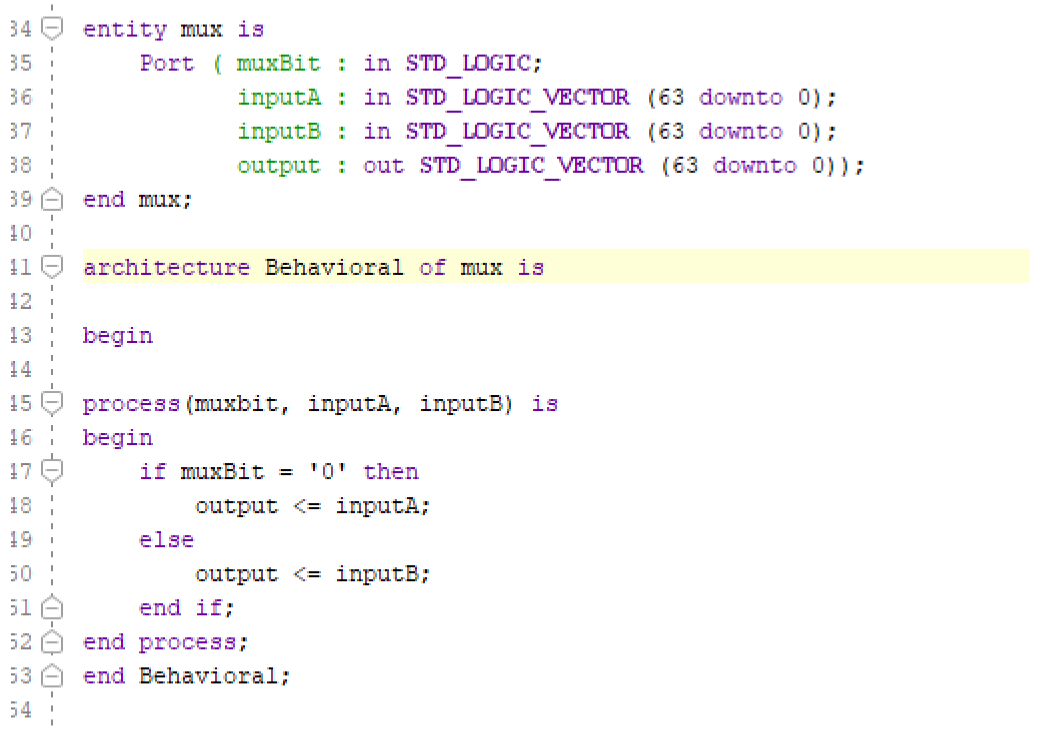
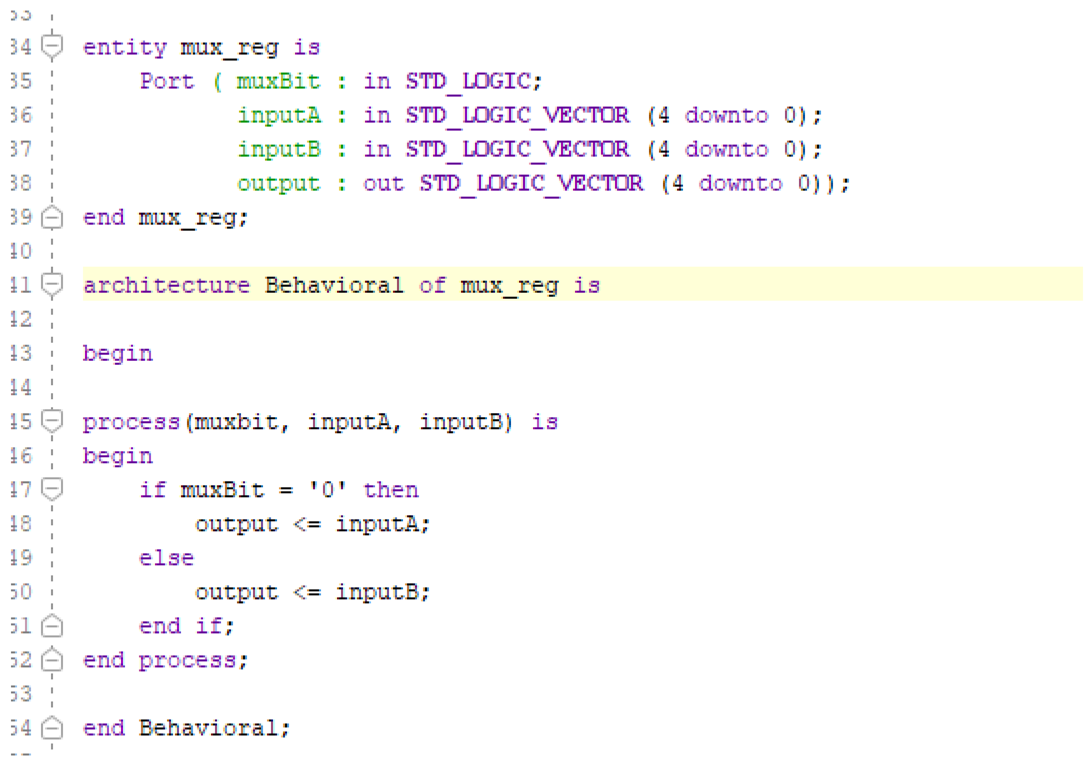
Austen Brooks
ECGR 3183 - Single Cycle Processor
Due 5/5/21

Mux
To simplify the design, I created 2 mux components. They are simple design files that take a 1 bit mux input and either 2 64 bit inputs and 1 64 bit output, or 2 5 bit inputs and 1 5 bit output.

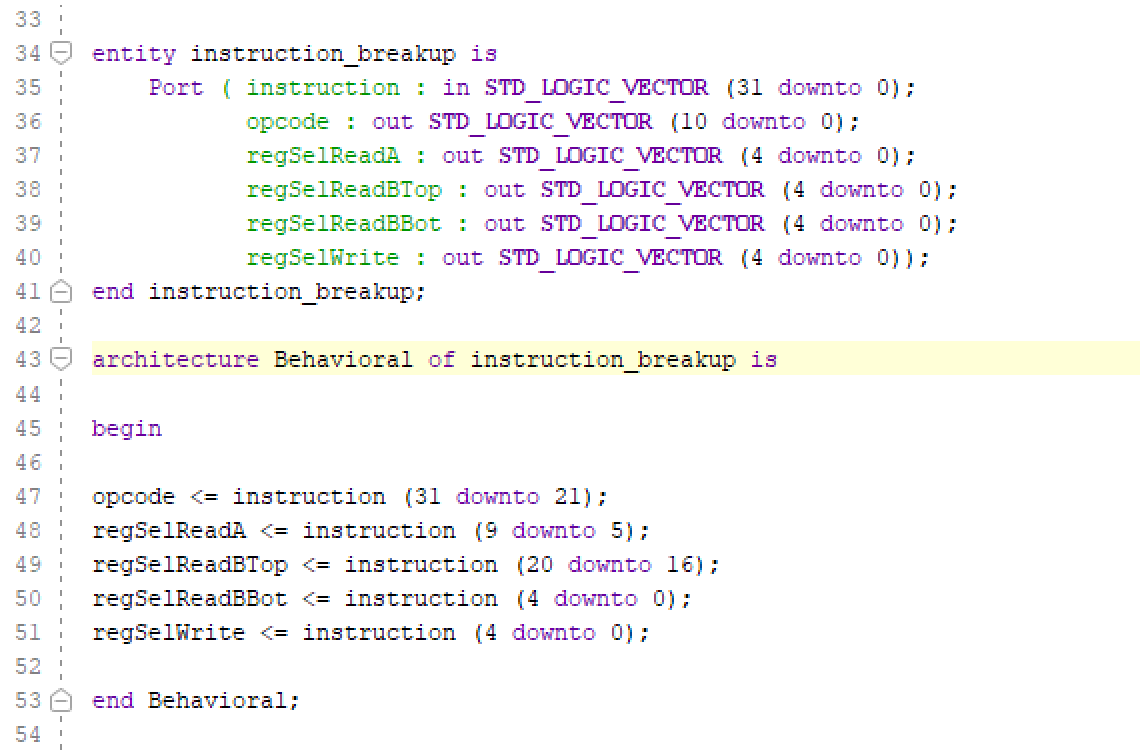


```vhdl
entity mux is
    Port ( muxBit : in STD_LOGIC;
           inputA : in STD_LOGIC_VECTOR (63 downto 0);
           inputB : in STD_LOGIC_VECTOR (63 downto 0);
           output : out STD_LOGIC_VECTOR (63 downto 0));
end mux;

architecture Behavioral of mux is

begin

process(muxbit, inputA, inputB) is
begin
    if muxBit = '0' then
        output <= inputA;
    else
        output <= inputB;
    end if;
end process;
end Behavioral;
```

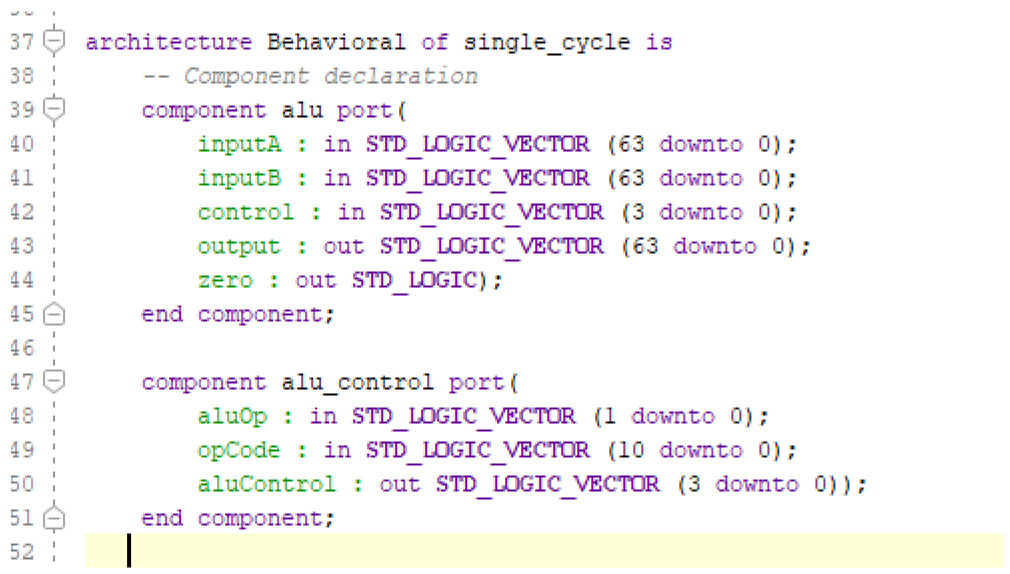


```vhdl
entity mux_reg is
    Port ( muxBit : in STD_LOGIC;
           inputA : in STD_LOGIC_VECTOR (4 downto 0);
           inputB : in STD_LOGIC_VECTOR (4 downto 0);
           output : out STD_LOGIC_VECTOR (4 downto 0));
end mux_reg;

architecture Behavioral of mux_reg is

begin

process(muxbit, inputA, inputB) is
begin
    if muxBit = '0' then
        output <= inputA;
    else
        output <= inputB;
    end if;
end process;

end Behavioral;
```

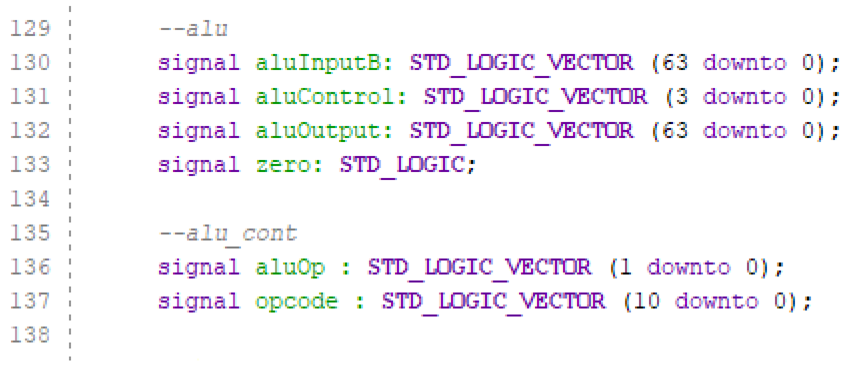
Instruction Breakup
Additionally I created another component that broke up the instruction into its relevant constant sections. This is unnecessary to make the processor run, but helped for labeling signals. Rather than just using instruction(31 downto 21) for the opcode inputs, you could just have it declared as a signal called opcode, which helps make the code a little more readable.


```vhdl

entity instruction_breakup is
    Port ( instruction : in STD_LOGIC_VECTOR (31 downto 0);
           opcode : out STD_LOGIC_VECTOR (10 downto 0);
           regSelReadA : out STD_LOGIC_VECTOR (4 downto 0);
           regSelReadBTop : out STD_LOGIC_VECTOR (4 downto 0);
           regSelReadBBot : out STD_LOGIC_VECTOR (4 downto 0);
           regSelWrite : out STD_LOGIC_VECTOR (4 downto 0));
end instruction_breakup;

architecture Behavioral of instruction_breakup is

begin

opcode <= instruction (31 downto 21);
regSelReadA <= instruction (9 downto 5);
regSelReadBTop <= instruction (20 downto 16);
regSelReadBBot <= instruction (4 downto 0);
regSelWrite <= instruction (4 downto 0);

end Behavioral;

```

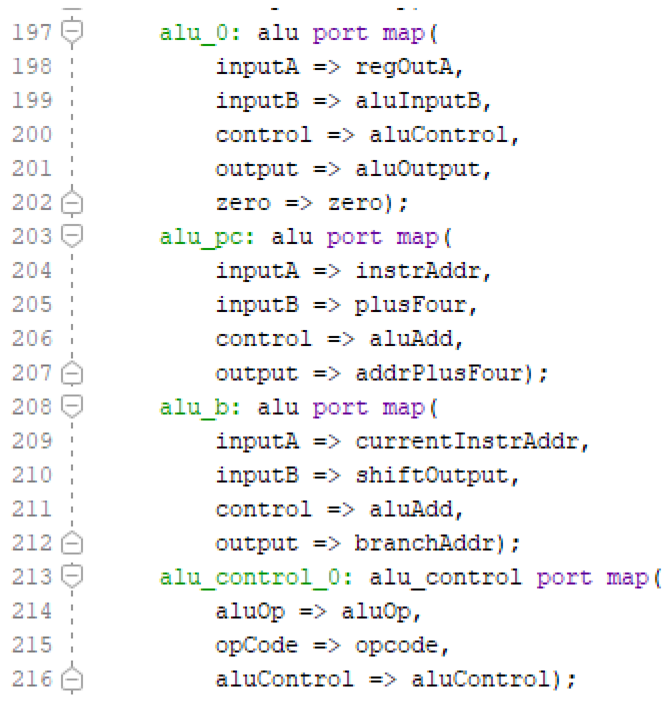
The Single Cycle Processor
The single cycle processor file essentially just declares all the components and connects the inputs and outputs of those components according to the diagram in the assignment instructions. Here is an example of the component declaration and connection using the alu and the alu control as an example:



```vhdl
architecture Behavioral of single_cycle is
    -- Component declaration
    component alu port(
        inputA : in STD_LOGIC_VECTOR (63 downto 0);
        inputB : in STD_LOGIC_VECTOR (63 downto 0);
        control : in STD_LOGIC_VECTOR (3 downto 0);
        output : out STD_LOGIC_VECTOR (63 downto 0);
        zero : out STD_LOGIC);
    end component;

    component alu_control port(
        aluOp : in STD_LOGIC_VECTOR (1 downto 0);
        opCode : in STD_LOGIC_VECTOR (10 downto 0);
        aluControl : out STD_LOGIC_VECTOR (3 downto 0));
    end component;

```



```vhdl
    --alu
    signal aluInputB: STD_LOGIC_VECTOR (63 downto 0);
    signal aluControl: STD_LOGIC_VECTOR (3 downto 0);
    signal aluOutput: STD_LOGIC_VECTOR (63 downto 0);
    signal zero: STD_LOGIC;

    --alu_cont
    signal aluOp : STD_LOGIC_VECTOR (1 downto 0);
    signal opcode : STD_LOGIC_VECTOR (10 downto 0);

```

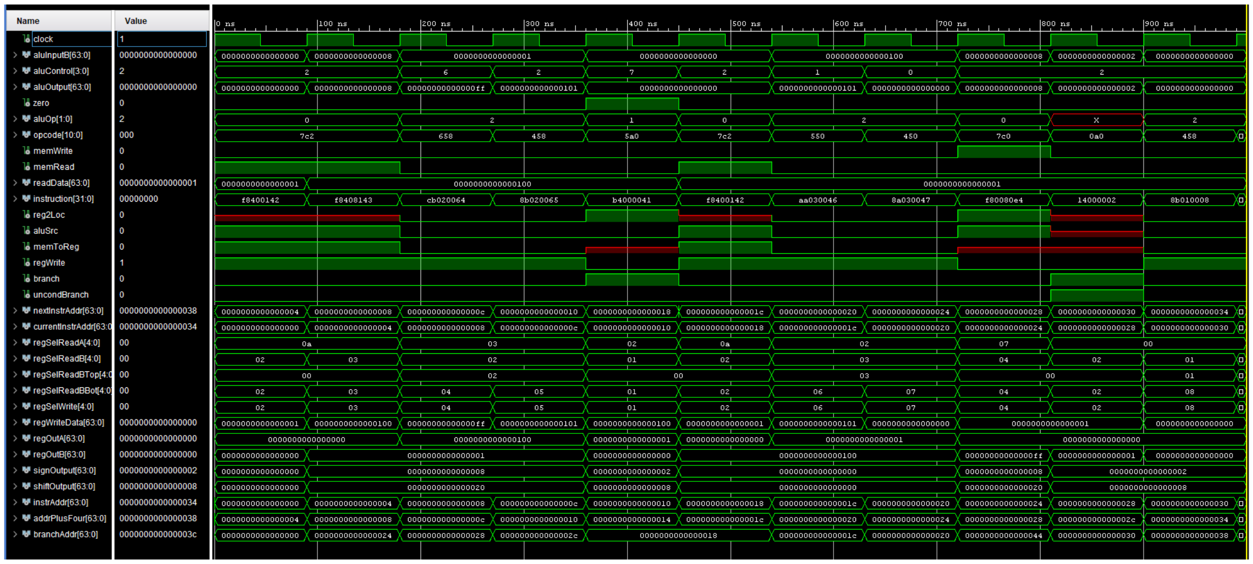


```vhdl
    alu_0: alu port map(
        inputA => regOutA,
        inputB => aluInputB,
        control => aluControl,
        output => aluOutput,
        zero => zero);
    alu_pc: alu port map(
        inputA => instrAddr,
        inputB => plusFour,
        control => aluAdd,
        output => addrPlusFour);
    alu_b: alu port map(
        inputA => currentInstrAddr,
        inputB => shiftOutput,
        control => aluAdd,
        output => branchAddr);
    alu_control_0: alu_control port map(
        aluOp => aluOp,
        opCode => opcode,
        aluControl => aluControl);
```

The alu and alu_control components are defined, their relevant signals are created and then they are instantiated as alu_0 and alu_control_0, where the relevant connections are made, for these two components the only connection is the output from the alu control called aluControl which is connected to the input called control on the alu. This process is repeated for every component in the single cycle processor until all connections have been made.

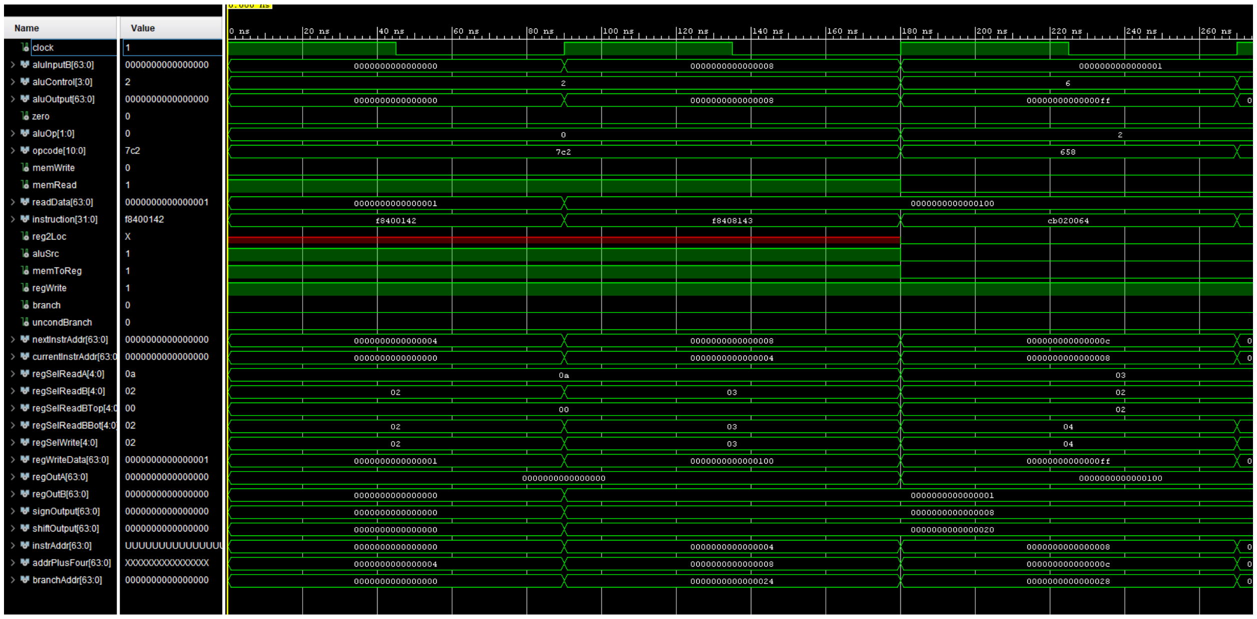
Testing the Processor
(All these waveforms are included in the single_cycle_behav.wcfg file in the zip file in order to see the values more clearly)

Running the processor with the data memory set for the first 64 bits = “0000….0001” and the second 64 bits set to “0000….0001 0000 0000”, results in this graph:



Name	Value
clock	1
aluInputB[63:0]	0000000000000000
aluControl[3:0]	2
aluOutput[63:0]	0000000000000000
zero	0
aluOp[1:0]	2
opcode[10:0]	000
memWrite	0
memRead	0
readData[63:0]	0000000000000001
instruction[31:0]	00000000
reg2Loc	0
aluSrc	0
memToReg	0
regWrite	1
branch	0
uncondBranch	0
nextInstrAddr[63:0]	0000000000000038
currentInstrAddr[63:0	0000000000000034
regSelReadA[4:0]	00
regSelReadB[4:0]	00
regSelReadBTop[4:0	00
regSelReadBBot[4:0]	00
regSelWrite[4:0]	00
regWriteData[63:0]	0000000000000000
regOutA[63:0]	0000000000000000
regOutB[63:0]	0000000000000000
signOutput[63:0]	0000000000000002
shiftOutput[63:0]	0000000000000008
instrAddr[63:0]	0000000000000034
addrPlusFour[63:0]	0000000000000038
branchAddr[63:0]	000000000000003c

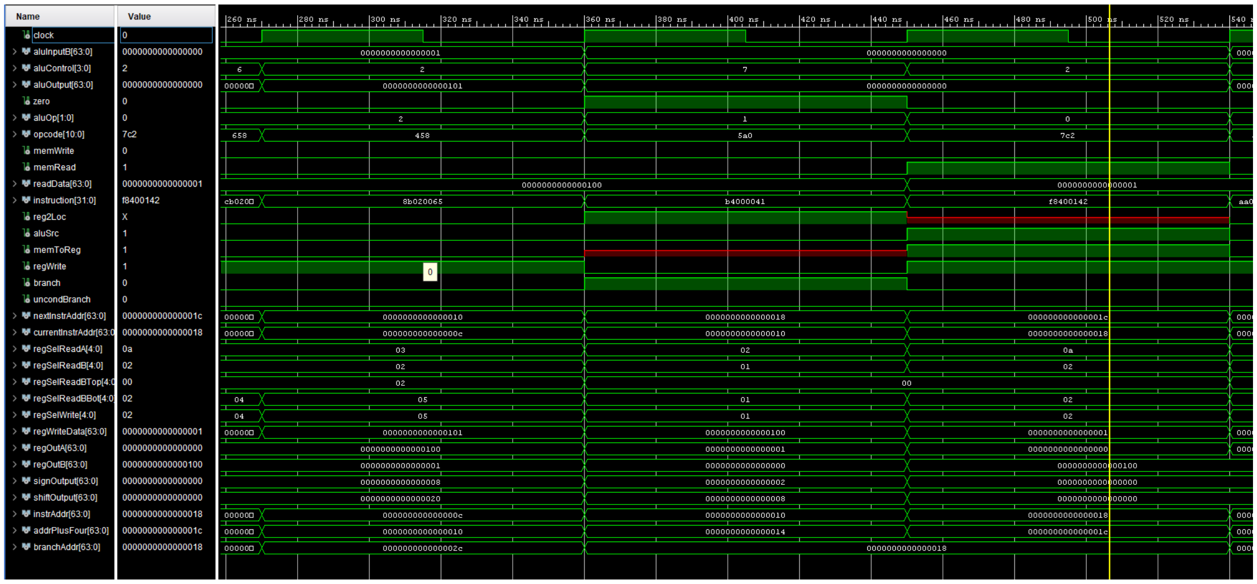
From this graph we can see the load and sub instructions working as intended in the first 3 instructions:



Name	Value
clock	1
aluInputB[63:0]	0000000000000000
aluControl[3:0]	2
aluOutput[63:0]	0000000000000000
zero	0
aluOp[1:0]	0
opcode[10:0]	7c2
memWrite	0
memRead	1
readData[63:0]	0000000000000001
instruction[31:0]	f8400142
reg2Loc	X
aluSrc	1
memToReg	1
regWrite	1
branch	0
uncondBranch	0
nextInstrAddr[63:0]	0000000000000000
currentInstrAddr[63:0	0000000000000000
regSelReadA[4:0]	0a
regSelReadB[4:0]	02
regSelReadBTop[4:0	00
regSelReadBBot[4:0]	02
regSelWrite[4:0]	02
regWriteData[63:0]	0000000000000001
regOutA[63:0]	0000000000000000
regOutB[63:0]	0000000000000000
signOutput[63:0]	0000000000000000
shiftOutput[63:0]	0000000000000000
instrAddr[63:0]	UUUUUUUUUUUUUUU
addrPlusFour[63:0]	XXXXXXXXXXXXXXXX
branchAddr[63:0]	0000000000000000

The first instruction loads the value 0x1 to register 2, the second instruction loads 0x100 into X3, this is verified by instruction 3, which runs x3 - x2 resulting in 0xff being stored into x4.

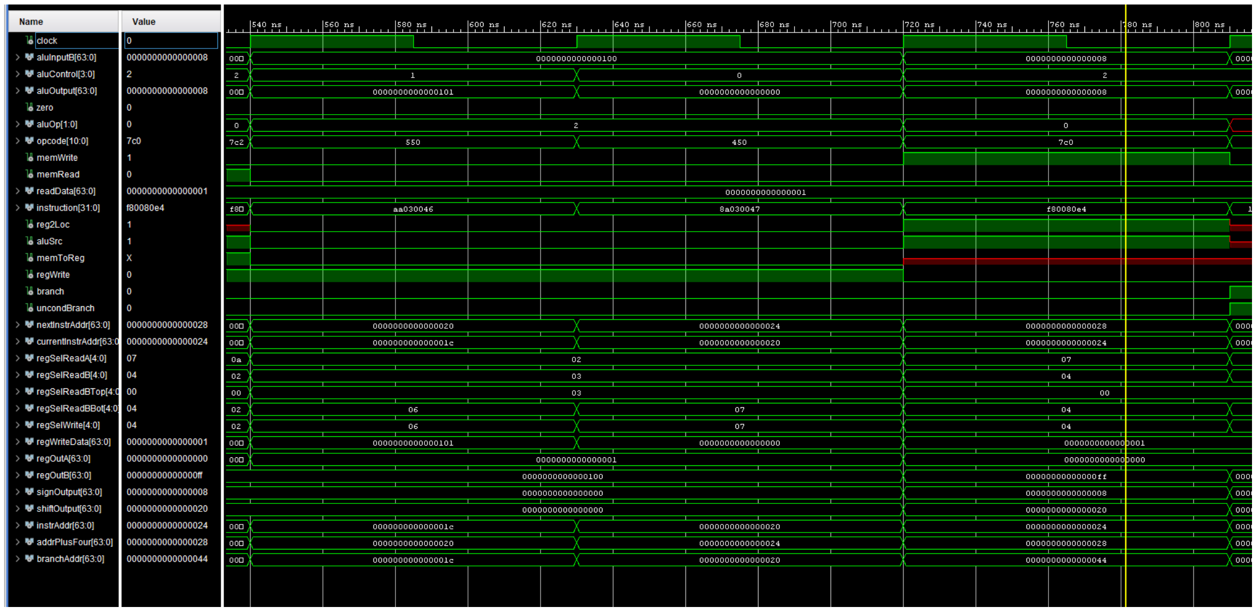
The next 3 instructions show the add, cbz and ldur instructions working:



Name	Value
clock	0
aluInputB[63:0]	0000000000000000
aluControl[3:0]	2
aluOutput[63:0]	0000000000000000
zero	0
aluOp[1:0]	0
opcode[10:0]	7c2
memWrite	0
memRead	1
readData[63:0]	0000000000000001
instruction[31:0]	f8400142
reg2Loc	X
aluSrc	1
memToReg	1
regWrite	1
branch	0
uncondBranch	0
nextInstrAddr[63:0]	000000000000001c
currentInstrAddr[63:0	0000000000000018
regSelReadA[4:0]	0a
regSelReadB[4:0]	02
regSelReadBTop[4:0	00
regSelReadBBot[4:0]	02
regSelWrite[4:0]	02
regWriteData[63:0]	0000000000000001
regOutA[63:0]	0000000000000000
regOutB[63:0]	0000000000000100
signOutput[63:0]	0000000000000000
shiftOutput[63:0]	0000000000000000
instrAddr[63:0]	0000000000000018
addrPlusFour[63:0]	000000000000001c
branchAddr[63:0]	0000000000000018

The add instruction adds the same registers as the sub instruction, resulting in a value of 0x101 being stored to x5. The CBZ instruction reads x1 which was left as zero resulting in the PC branching from address 0x10 to 0x18, where another ldur instruction is performed.

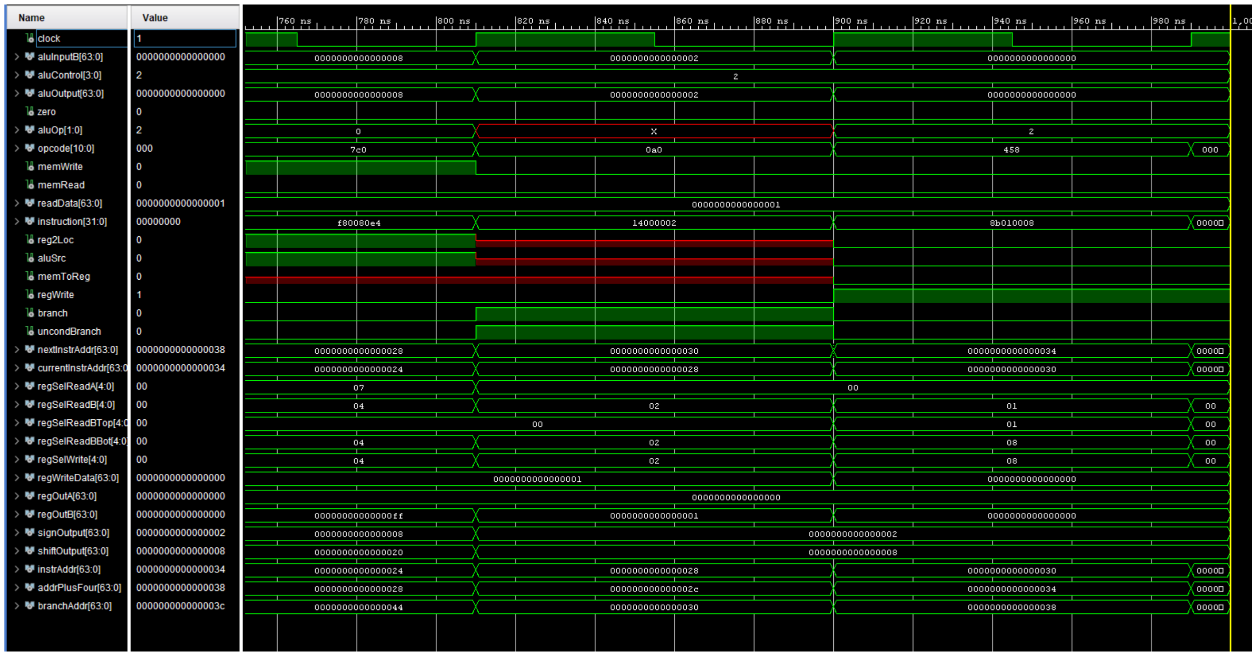
The next 3 instructions show the orr and, and stur instructions running:



Name	Value
clock	0
aluInputB[63:0]	0000000000000008
aluControl[3:0]	2
aluOutput[63:0]	0000000000000008
zero	0
aluOp[1:0]	0
opcode[10:0]	7c0
memWrite	1
memRead	0
readData[63:0]	0000000000000001
instruction[31:0]	f80080e4
reg2Loc	1
aluSrc	1
memToReg	X
regWrite	0
branch	0
uncondBranch	0
nextInstrAddr[63:0]	0000000000000028
currentInstrAddr[63:0	0000000000000024
regSelReadA[4:0]	07
regSelReadB[4:0]	04
regSelReadBTop[4:0	00
regSelReadBBot[4:0]	04
regSelWrite[4:0]	04
regWriteData[63:0]	0000000000000001
regOutA[63:0]	0000000000000000
regOutB[63:0]	00000000000000ff
signOutput[63:0]	0000000000000008
shiftOutput[63:0]	0000000000000020
instrAddr[63:0]	0000000000000024
addrPlusFour[63:0]	0000000000000028
branchAddr[63:0]	0000000000000044

The orr instruction performs bitwise or on 0x100 and 0x1, which stores the value 0x101 to x6. The and instruction ands together the same 2 registers, which stores the value 0x0 to x7. Finally the stur instruction stores the value of x4 (0xff) to the second doubleword (x7 = 0, x7 + 8 = second doubleword) in the data memory. This instruction technically isn't confirmed to be working because no instruction afterwards looks at that memory location, but all the correct flags and registers are set correctly so it should be working.

Finally the single cycle processor runs the b instruction, an and instruction, and then an added no op instruction:



Name	Value
clock	1
aluInputB[63:0]	0000000000000000
aluControl[3:0]	2
aluOutput[63:0]	0000000000000000
zero	0
aluOp[1:0]	2
opcode[10:0]	000
memWrite	0
memRead	0
readData[63:0]	0000000000000001
instruction[31:0]	00000000
reg2Loc	0
aluSrc	0
memToReg	0
regWrite	1
branch	0
uncondBranch	0
nextInstrAddr[63:0]	0000000000000038
currentInstrAddr[63:0	0000000000000034
regSelReadA[4:0]	00
regSelReadB[4:0]	00
regSelReadBTop[4:0	00
regSelReadBBot[4:0]	00
regSelWrite[4:0]	00
regWriteData[63:0]	0000000000000000
regOutA[63:0]	0000000000000000
regOutB[63:0]	0000000000000000
signOutput[63:0]	0000000000000002
shiftOutput[63:0]	0000000000000008
instrAddr[63:0]	0000000000000034
addrPlusFour[63:0]	0000000000000038
branchAddr[63:0]	000000000000003c

The branch instruction jumps the program counter from 0x28 to 0x30 (2 instructions). The and instructions ands x1 and x0, which are both left as all 0’s. And finally there is an additional noop command that does nothing, some of the flags and values remain the same during the noop because the components and controls aren’t actually set up to deal with noop commands, but the noop was needed to stop the program from having an error every time it ran, because the 11 instructions didn’t fit perfectly into 1000 ns.