Austen Brooks
ECGR 3183
Datapath Elements

Program Counter
The program counter is relatively simple and will simply output the passed input on the rising clock edge. This is done by calling a process based on when the clock input changes, and then seeing if the clock is currently a 1, if it is, pass the input.

```
entity program_counter is
    Port ( clock : in STD_LOGIC;
           nextInstrAddr : in STD_LOGIC_VECTOR (63 downto 0);
           currentInstrAddr : out STD_LOGIC_VECTOR (63 downto 0));
end program_counter;

architecture Behavioral of program_counter is

begin
    process(clock) is
        begin
        if clock'event and clock = '1' then
            currentInstrAddr <= nextInstrAddr;
        end if;
    end process;


end Behavioral;
```

Instruction Memory
The instruction memory contains the machine code of all our instructions, each instruction is broken into 4 chunks of 8 bits. Each byte is stored in a constant array that can hold 16 instructions. The input that is passed is converted into an integer that is used to grab the 4 bytes from the instruction memory array. It is assumed that the instructions start at address 0x0000000000000000. The last 12 bytes are left all zeros, because there are only 13 instructions and I made the array hold 16 instructions just to have a nice number of bytes.
The data is stored in big endian format, where the most significant bit is in the lowest memory locations.

```
34  entity instruction_memory is
35      Port ( instrAddr : in STD_LOGIC_VECTOR (63 downto 0);
36             instrData : out STD_LOGIC_VECTOR (31 downto 0));
37  end instruction_memory;
38
39  architecture Behavioral of instruction_memory is
40
41      -- memory containing the instructions in binary, holds 64 bytes of data:
42      type ROM is array (0 to 63) of std_logic_vector(7 downto 0);
43      constant INSTR_DATA : ROM := (
44      "11111000", "01000000", "00000001", "01000010",
45      "11111000", "01000000", "10000001", "01000011",
46      "11001011", "00000010", "00000000", "01100100",
47      "10001011", "00000010", "00000000", "01100101",
48      "10110100", "00000000", "00000000", "01000001",
49      "10110100", "00000000", "00000000", "01000000",
50      "11111000", "01000000", "00000001", "01000010",
51      "10101010", "00000011", "00000000", "01000110",
52      "10001010", "00000011", "00000000", "01000111",
53      "11111000", "00000000", "10000000", "11100100",
54      "00010100", "00000000", "00000000", "00000010",
55      "11111000", "01000000", "10000001", "01000011",
56      "10001011", "00000001", "00000000", "00001000",
57      "00000000", "00000000", "00000000", "00000000",
58      "00000000", "00000000", "00000000", "00000000",
59      "00000000", "00000000", "00000000", "00000000");
60
61  begin
62      instrData(31 downto 24) <= INSTR_DATA(to_integer(unsigned(instrAddr)));
63      instrData(23 downto 16) <= INSTR_DATA(to_integer(unsigned(instrAddr) + 1));
64      instrData(15 downto 8) <= INSTR_DATA(to_integer(unsigned(instrAddr) + 2));
65      instrData(7 downto 0) <= INSTR_DATA(to_integer(unsigned(instrAddr) + 3));
66
67  end Behavioral;
```

Register File

The register file holds an array of 32 64 bit registers that can be written to and read from. The reads will always occur, but the writes will only occur when the regWrite bit is set to 1 from the control. Registers that would normally function differently like XZR or LR are not implemented, so they can be written to the same as any other register.

```
33
34  entity register_file is
35      Port ( regSelReadA : in STD_LOGIC_VECTOR (4 downto 0);
36             regSelReadB : in STD_LOGIC_VECTOR (4 downto 0);
37             regSelWrite : in STD_LOGIC_VECTOR (4 downto 0);
38             writeData : in STD_LOGIC_VECTOR (63 downto 0);
39             regWrite : in STD_LOGIC;
40             outA : out STD_LOGIC_VECTOR (63 downto 0);
41             outB : out STD_LOGIC_VECTOR (63 downto 0));
42  end register_file;
43
44  architecture Behavioral of register_file is
45  type REG is array (0 to 31) of std_logic_vector(63 downto 0);
46      signal registers : REG := (
47      "0000000000000000000000000000000000000000000000000000000000000000", --X0
48      "0000000000000000000000000000000000000000000000000000000000000000",
49      "0000000000000000000000000000000000000000000000000000000000000000",
50      "0000000000000000000000000000000000000000000000000000000000000000",
51      "0000000000000000000000000000000000000000000000000000000000000000",
52      "0000000000000000000000000000000000000000000000000000000000000000",
53      "0000000000000000000000000000000000000000000000000000000000000000",
54      "0000000000000000000000000000000000000000000000000000000000000000",
55      "0000000000000000000000000000000000000000000000000000000000000000",
56      "0000000000000000000000000000000000000000000000000000000000000000",
57      "0000000000000000000000000000000000000000000000000000000000000000", --X10
        "--------------------------------------------------------------------"
```

```
66          "0000000000000000000000000000000000000000000000000000000000000000",
67          "0000000000000000000000000000000000000000000000000000000000000000", --X20
68          "0000000000000000000000000000000000000000000000000000000000000000",
69          "0000000000000000000000000000000000000000000000000000000000000000",
70          "0000000000000000000000000000000000000000000000000000000000000000",
71          "0000000000000000000000000000000000000000000000000000000000000000",
72          "0000000000000000000000000000000000000000000000000000000000000000",
73          "0000000000000000000000000000000000000000000000000000000000000000",
74          "0000000000000000000000000000000000000000000000000000000000000000",
75          "0000000000000000000000000000000000000000000000000000000000000000",
76          "0000000000000000000000000000000000000000000000000000000000000000",
77          "0000000000000000000000000000000000000000000000000000000000000000", --X30
78          "0000000000000000000000000000000000000000000000000000000000000000");
79      begin
80          process(regSelReadA, regSelReadB, regSelWrite, writeData, regWrite) is
81              begin
82
83              outA <= registers(to_integer(unsigned(regSelReadA)));
84              outB <= registers(to_integer(unsigned(regSelReadB)));
85
86              if regWrite = '1' then
87                  registers(to_integer(unsigned(regSelWrite))) <= writeData;
88              end if;
89          end process;
90
91
92      end Behavioral;
```

ALU

The ALU is very straightforward, it takes 2 64 bit inputs, and will perform an instruction decided by the assigned 4 bit ALU control. It sets the output based on the operation and the inputs, and it will output a 1 on the zero flag if the operation results in a zero. Because the zero flag is put through an AND gate with a flag from the control, there is no risk in setting the zero flag for all operations, but you could just set it for the operation used for CBZ if you wanted. The zero flag is also reset to 0 at the beginning every time the ALU runs.

```
35  entity alu is
36      Port ( inputA : in STD_LOGIC_VECTOR (63 downto 0);
37             inputB : in STD_LOGIC_VECTOR (63 downto 0);
38             control : in STD_LOGIC_VECTOR (3 downto 0);
39             output : out STD_LOGIC_VECTOR (63 downto 0);
40             zero : out STD_LOGIC);
41  end alu;
42
```

```
41  end alu;
42
43  architecture Behavioral of alu is
44
45  begin
46      process(inputA, inputB, control) is
47          begin
48          zero <= '0';
49          if control = "0000" then
50              output <= inputA and inputB;
51              if (inputA and inputB) = "00000000000000000000000000000000000000000000000000000000000000000" then
52                  zero <= '1';
53              end if;
54          elsif control = "0001" then
55              output <= inputA or inputB;
56              if (inputA or inputB) = "00000000000000000000000000000000000000000000000000000000000000000" then
57                  zero <= '1';
58              end if;
59          elsif control = "0010" then
60              output <= inputA + inputB;
61              if (inputA + inputB) = "00000000000000000000000000000000000000000000000000000000000000000" then
62                  zero <= '1';
63              end if;
64          elsif control = "0110" then
65              output <= inputA - inputB;
66              if (inputA - inputB) = "00000000000000000000000000000000000000000000000000000000000000000" then
67                  zero <= '1';
68              end if;
69          elsif control = "0111" then
70              output <= inputB;
71              if inputB = "00000000000000000000000000000000000000000000000000000000000000000" then
72                  zero <= '1';
73              end if;
74          elsif control = "1100" then
75              output <= inputA nor inputB;
76              if (inputA nor inputB) = "00000000000000000000000000000000000000000000000000000000000000000" then
77                  zero <= '1';
78              end if;
79          end if;
80
81      end process;
82
83  end Behavioral;
```

Data Memory

The data memory holds an array of 256 bytes that can be written to or read from, the reads and writes are controlled by the memWrite and memRead bits that will be set by the control. The data is stored in big endian format, where the most significant bit is in the lowest memory locations, same as the instruction memory.

```
34  entity data_memory is
35      Port ( addrIn : in STD_LOGIC_VECTOR (63 downto 0);
36             writeData : in STD_LOGIC_VECTOR (63 downto 0);
37             memWrite : in STD_LOGIC;
38             memRead : in STD_LOGIC;
39             readData : out STD_LOGIC_VECTOR (63 downto 0));
40  end data_memory;
41
42  architecture Behavioral of data_memory is
43  type ROM is array (0 to 255) of std_logic_vector(7 downto 0);
44      signal  memData : ROM := (
45      "00000000", "00000000", "00000000", "00000000", "00000000", "00000000", "00000000", "00000000",
46      "00000000", "00000000", "00000000", "00000000", "00000000", "00000000", "00000000", "00000000",
47      "00000000", "00000000", "00000000", "00000000", "00000000", "00000000", "00000000", "00000000",
48      "00000000", "00000000", "00000000", "00000000", "00000000", "00000000", "00000000", "00000000",
49      "00000000", "00000000", "00000000", "00000000", "00000000", "00000000", "00000000", "00000000",
50      "00000000", "00000000", "00000000", "00000000", "00000000", "00000000", "00000000", "00000000",
51      "00000000", "00000000", "00000000", "00000000", "00000000", "00000000", "00000000", "00000000",

73      "00000000", "00000000", "00000000", "00000000", "00000000", "00000000", "00000000", "00000000",
74      "00000000", "00000000", "00000000", "00000000", "00000000", "00000000", "00000000", "00000000",
75      "00000000", "00000000", "00000000", "00000000", "00000000", "00000000", "00000000", "00000000",
76      "00000000", "00000000", "00000000", "00000000", "00000000", "00000000", "00000000", "00000000");
77  begin
78
79      process(addrIn, writeData, memWrite, memRead) is
80          begin
81          if memRead = '1' then
82              readData(63 downto 56) <= memData(to_integer(unsigned(addrIn)));
83              readData(55 downto 48) <= memData(to_integer(unsigned(addrIn) + 1));
84              readData(47 downto 40) <= memData(to_integer(unsigned(addrIn) + 2));
85              readData(39 downto 32) <= memData(to_integer(unsigned(addrIn) + 3));
86              readData(31 downto 24) <= memData(to_integer(unsigned(addrIn) + 4));
87              readData(23 downto 16) <= memData(to_integer(unsigned(addrIn) + 5));
88              readData(15 downto 8) <= memData(to_integer(unsigned(addrIn) + 6));
89              readData(7 downto 0) <= memData(to_integer(unsigned(addrIn) + 7));
90          end if;
91
92          if memWrite = '1' then
93              memData(to_integer(unsigned(addrIn))) <= writeData(63 downto 56);
94              memData(to_integer(unsigned(addrIn) + 1)) <= writeData(55 downto 48);
95              memData(to_integer(unsigned(addrIn) + 2)) <= writeData(47 downto 40);
96              memData(to_integer(unsigned(addrIn) + 3)) <= writeData(39 downto 32);
97              memData(to_integer(unsigned(addrIn) + 4)) <= writeData(31 downto 24);
98              memData(to_integer(unsigned(addrIn) + 5)) <= writeData(23 downto 16);
99              memData(to_integer(unsigned(addrIn) + 6)) <= writeData(15 downto 8);
100             memData(to_integer(unsigned(addrIn) + 7)) <= writeData(7 downto 0);
101         end if;
102         end process;
103 end Behavioral;
```

Sign Extend

The sign extend element works by reading the instruction given and finding the immediate value's location based on the opcode. With the location, it will look at the msb of the immediate and if it is a 0, fill the output with 0s, otherwise fill it with 1s. Then it will replace the end bits with whatever value was in the immediate section of the instruction.

```
34  entity sign_extend is
35      Port ( instruction : in STD_LOGIC_VECTOR (31 downto 0);
36             output : out STD_LOGIC_VECTOR (63 downto 0));
37  end sign_extend;
38
39  architecture Behavioral of sign_extend is
40
41  begin
42
43      process(instruction) is
44          begin
45          if instruction (31 downto 26) = "000101" then --if B
46              if instruction (25 downto 25) = "0" then
47                  output <= "00000000000000000000000000000000000000000000000000000000000000000";
48              else
49                  output <= "11111111111111111111111111111111111111111111111111111111111111111";
50              end if;
51              output (25 downto 0) <= instruction (25 downto 0);
52
53          elsif instruction (31 downto 27) = "11111" then --if LDUR or STUR
54              if instruction (20 downto 20) = "0" then
55                  output <= "00000000000000000000000000000000000000000000000000000000000000000";
56              else
57                  output <= "11111111111111111111111111111111111111111111111111111111111111111";
58              end if;
59              output (8 downto 0) <= instruction (20 downto 12);
60
61          elsif instruction (31 downto 25) = "1011010" then --if CBZ or CBNZ
62              if instruction (23 downto 23) = "0" then
63                  output <= "00000000000000000000000000000000000000000000000000000000000000000";
64              else
65                  output <= "11111111111111111111111111111111111111111111111111111111111111111";
66              end if;
67              output (18 downto 0) <= instruction (23 downto 5);
68
69          end if;
70
71      end process;
72
73
74  end Behavioral;
```