

Putting it All Together: Building the Computer

In this lab you will put the parts of the computer together to build a functional computer. This is just a matter of including all the modules in your processor file, and defining the inputs to and the outputs from the blocks in this file. You will also define the other combinational circuits used by the computer, such as a few multiplexers and a decoder.

You could do the following:

1. For the computer, you may included all the modules required for the computer. You do not have to implement each of the elements of the computer in separate verilog files. You can instantiate two copies of the MUX1.v file and used one as MUXIN (this multiplexes the accumulator and the data input into the bus), and one as MUXMEM (this multiplexes the input to the bus from the registers X, PC, and MAR). You could also instantiate two copies of the file COUNTER.v and used them as the X and PC registers/counters, two copies of the REGISTER.v and used them as MAR and OUTR registers, two copies of the REGISTER1BIT and used them in ZR and CR registers.

```
// Decoder and Muxes
DECODER DECODER1 (ADDR, ADDR_FF, ADDR_NOTFF);
MUX1 MUXIN (ACCAR, 8'h00, INPUT_PORT, 8'h00, {!INMUX1, !STORE}, OUTMUX);
MUX1 MUXMEM (EXT_ADDR, XREG, PCREG, MARREG, MEM_SEL, ADDR);

// Tristat Buffer
TRISTATE TRIS1 (!E, OUTMUX, DATA);

// Counters
COUNTER XR (DATA, CLK, !X_L, !X_I, GND, XREG);
COUNTER PCR (DATA, CLK, !PC_L, !PC_I, !RESN, PCREG);

// Register
REGISTER MAR (DATA, CLK, !MAR_L, MARREG);
REGISTER OUTRG (DATA, CLK, !OUTPUT_ENA, OUTPUT_PORT);
REGISTER1BIT ZR (ZALU, CLK, !Z_L, ZREG);
REGISTER1BIT CR (CALU, CLK, !C_L, CREG);
REGISTER ACCR (ALUR, CLK, !ACCA_L, ACCAR);
LATCH1 INSL (DATA, CLK, !INST_L, INSTR);
```

2. Create the file for the final computer design. Here are some lines from my file processor.v — Include the parts which define the inputs to the ALU and ACCA:

```

module PROCESSOR (CLK, RESN, EXT_W, EXT_R, INPUT_PORT, EXT_ADDR, MEM_CS, M_R, M_W,
                 DATA, ADDR, OUTPUT_PORT, TIN, TOUT, E, OUTMUX, INMUX1, STORE);

    input CLK, RESN, EXT_W, EXT_R;
    input [7:0] INPUT_PORT, EXT_ADDR;
    output MEM_CS, M_R, M_W, E, INMUX1, STORE;
    inout [7:0] DATA;
    output [7:0] ADDR, OUTPUT_PORT, OUTMUX;
    output [1:0] TIN, TOUT;

    wire OUTPUT_ENA, READ;
    wire ZREG, CREG, ZALU, CALU, C_L, Z_L, ADDR_FF, ADDR_NOTFF;
    wire X_I, X_L, INST_L, PC_I, PC_L, ACCA_L, MAR_L;
    wire [1:0] MEM_SEL;
    wire [5:0] ALU_CTL, DONE;
    wire [7:0] XREG, PCREG, MARREG, INSTR, ACCAR, ALUR;

    parameter VCC = 1'b1, GND = 1'b0;

    assign TIN = TOUT;
    assign E = (!(INMUX1 | !STORE));
    assign MEM_CS = ADDR_NOTFF;

    // Decoder and Muxes
    DECODER DECODER1 (ADDR, ADDR_FF, ADDR_NOTFF);
    MUX1 MUXIN (ACCAR, 8'h00, INPUT_PORT, 8'h00, {!INMUX1, !STORE}, OUTMUX);
    MUX1 MUXMEM (EXT_ADDR, XREG, PCREG, MARREG, MEM_SEL, ADDR);

    // Tristat Buffer
    TRISTATE TRIS1 (!E, OUTMUX, DATA);

    // Counters
    COUNTER XR (DATA, CLK, !X_L, !X_I, GND, XREG);
    COUNTER PCR (DATA, CLK, !PC_L, !PC_I, !RESN, PCREG);

    // Register
    REGISTER MAR (DATA, CLK, !MAR_L, MARREG);
    REGISTER OUTRG (DATA, CLK, !OUTPUT_ENA, OUTPUT_PORT);
    REGISTER1BIT ZR (ZALU, CLK, !Z_L, ZREG);
    REGISTER1BIT CR (CALU, CLK, !C_L, CREG);
    REGISTER ACCR (ALUR, CLK, !ACCA_L, ACCAR);
    LATCH1 INSL (DATA, CLK, !INST_L, INSTR);

    // ALU
    ALU ALU (ALU_CTL, XREG, DATA, ACCAR, ZALU, CALU, ALUR);

    // Control
    CONTROL CONTROL (INSTR, TIN, TOUT, CLK, !RESN, ZREG, CREG, ALU_CTL, MEM_SEL, INST_L, PC_I, PC_L,
                   ACCA_L, MAR_L, C_L, Z_L, X_I, X_L, READ, STORE, OUTPUT_ENA);

    // Combinatorial
    COMBINA COMBINA (!RESN, !EXT_W, !STORE, !ADDR_FF, !EXT_R, !READ, M_W, M_R, INMUX1);

endmodule

```

You should write this file in pieces – for example, start with the control unit, and make sure that part compiles. (The control unit uses inputs from the instruction register, the carry and zero registers, which have not yet been defined).

Also, it is possible to simplify the circuit by including the gates used to generate EXT_ENA, M_W and M_R in a Verilog file:

```

module COMBINA (RESN, EXT_W, STORE, ADDR_FF, EXT_R, READ, M_W, M_R, INMUX1);
  input RESN, EXT_W, STORE, ADDR_FF, EXT_R, READ;
  output reg M_W, M_R, INMUX1;
  reg EXT_ENA, INPUT_ENA;

  always @(*)
  begin
    EXT_ENA = !(RESN & EXT_W);
    M_W = (!(EXT_ENA | STORE));
    M_R = (!(!(RESN & EXT_R)) | READ);
    INPUT_ENA = !(ADDR_FF & !M_R);
    INMUX1 = (!(EXT_ENA | !INPUT_ENA));
    // OUTPUT_ENA = !(!ADDR_FF & !M_W);
  end

endmodule

```

4. Simulate the function of the computer. In your simulation, include the inputs to and the outputs from the computer, and the values of the internal registers. You need to supply the values for the data from memory. You may start with the reset input high, and made sure the control lines from the control state machine were in the proper states. I then brought reset low, and reran the simulation. In order to get data from the memory, both MEM_CS and MEM_R need to be low. When MEM_CS and MEM_R were both low, put the appropriate values for the address on the DATA lines. You may assume that the following program is in the memory:

```

LDAA #0x2A
ADAA 0xF0
STAA 0xF1

```

The following values should be in memory:

Address	Data
0x00	0x02
0x01	0x2A
0x02	0x04
0x03	0xF0
0x04	0x03
0x05	0xF1
.	.
.	.
0xF0	0xEB
0xF1	0x00

At the start of the simulation, when MEMM_CS and MEM_R are both low, and ADDR is at 0x00, put an 0x02 on the DATA input lines. Then reran the simulation and in the next cycle, when the ADDR was at 0x01, put an 0x2A on the DATA input lines. The instruction LDAA #0x2A is a two-cycle instruction. At the end to the second cycle, verify that 0x2A was loaded into ACCA.

To simulate the ADDA 0xF0 instruction, you should verify that the ACCA contained a 0x15 (0x2A + 0xEB), and that the carry is set. Then, simulate the STAA 0xF1 instruction to verify that the value in ACCA was put on the DATA output lines. Check that that MEM_CS and MEM_W are both low, so that the value in ACCA would be written into memory address 0xF1. Continue until you verify that all the instructions worked properly.

5. You may then test the input and output ports. Simulate the instructions LDAA 0xFF to verify that the system would load data from the input port, and simulated the STAA 0xFF instruction to verify that the system would write data to the output port.

After the simulation works, it is time to program the computer into an Altera chip. Follow the process you have developed in previous labs.

A Test Program for Computer

Your computer should be able to run a program when the RESET input is switched to HIGH. The program you need to demonstrate is as follows:

```

J1:   LDX #0A
J2   LDAA 0,X
     INX
     CPX #10
     JEQ J1
     JMP J2

```

This program consists on the following instructions:

1. LDAX #0A Loads the X register with the address value 0x0A.
2. LDAA 0,X Loads ACCA with the value in memory at the address in the X register (the contents of address 0x0A).
3. INX Increments the X register (the X register now points to the next memory address).
4. CPX #10 Compares the contents of X to the number 0x0F (to see if it has reached the last element in the table).
5. JEQ J1 Conditional jump to address J1. The next line to execute will be address J1 IF the result of the previous operation is zero: Z = 1, otherwise it will continue executing the program in a normal way.
6. JMP J2 Unconditional jump to address J2. If the conditional jump is not true (Z ≠ 1), then the program will continue reading the next element in memory.

The data values to be read by this program are stored in the following addresses:

0x0A	0x81
0x0B	0x42
0x0C	0x24
0x0D	0x18
0x0E	0x24
0x0F	0x42

You need to generate a signal to enable the OUTPUT register to be able to display these values on the LEDs or use the logic analyzer to show that you are accessing these addresses.

A Verilog Program for writing to RAM memory

You may use the following Verilog program to program your memory. You need to create a project and include the `memory.v` file and a counter (similar to the PC of your computer) which in the following source code is named `cnter` and instantiated with the name `cnter1`. The counter will be your clock, and every time you need to output the address (`ADDR`) and what needs to be saved at that address (`DATA`).

The way this program saves to memory is by bringing both Chip Select (CS) and Write Enable (WE) low to be able to save the information into the RAM. When these two signals are low, it then provides the address and the data. It then brings CS and WE high. This process is repeated until your entire program is loaded into memory. The last statement in this code sets the variable `DONE` high to indicate that it has finished saving to memory. You may connect this output of the FPGA to a LED to show you that programming the memory is done.

```
module memory (CLK, CS, OE, WE, ADDR, DATA, DONE);
  input CLK;
  output reg CS, OE, WE, DONE;
  wire [7:0] COUNT;
  output reg [7:0] ADDR, DATA;
  reg RESET;

  cnter cnter1 (CLK, RESET, COUNT);
  always @(*)
  begin
    CS = 1'h1; OE = 1'h1; WE = 1'h1; RESET = 1'h0; DONE = 1'h0;
    ADDR = 8'hzz; DATA = 8'hzz;
    case(COUNT)
    0: ;
    1: begin
        CS = 1'h0; WE = 1'h0; ADDR = 8'h00; DATA = 8'h09;
      end
    2: ;
    3: begin
        CS = 1'h0; WE = 1'h0; ADDR = 8'h01; DATA = 8'h0A;
      end
    4: ;
    5: begin
        CS = 1'h0; WE = 1'h0; ADDR = 8'h02; DATA = 8'h02;
      end
    6: ;
    7: begin
        CS = 1'h0; WE = 1'h0; ADDR = 8'h03; DATA = 8'h0A;
      end
    8: ;

    . . . .

    21: begin
        CS = 1'h0; WE = 1'h0; ADDR = 8'h0A; DATA = 8'h18;
      end
    22: ;
    23: begin
        CS = 1'h0; WE = 1'h0; ADDR = 8'h0B; DATA = 8'h24;
      end
    24: ;
    25: begin
        CS = 1'h0; WE = 1'h0; ADDR = 8'h0C; DATA = 8'h42;
      end
    26: ;
    27: begin
        CS = 1'h0; WE = 1'h0; ADDR = 8'h0D; DATA = 8'h81;
      end
    28: ;
    29: begin
        CS = 1'h0; WE = 1'h0; ADDR = 8'h0E; DATA = 8'h42;
      end
    30: ;
    31: begin
        CS = 1'h0; WE = 1'h0; ADDR = 8'h0F; DATA = 8'h24;
      end
    32: ;
    33: begin
        CS = 1'h0; WE = 1'h0; ADDR = 8'h10; DATA = 8'h00;
      end
    34: ;
    default: begin
        DONE = 1'h1;
      end
    endcase
  end
endmodule
```