3TB4 Postlab

Yukun Chen
Chris Peabody

To make this stepper motor ASIP, we first created and tested the individual modules, integrating them into the datapath as we create them. We tested each module individually to make sure they were working correctly according to the specifications. Afterwards, generated modules using the Quartus 2 MegaWizard Plugin to take advantage of the on-chip memory of the FPGA. We used the memory to hold machine code instructions used to control the motor, as well as binary used to control what voltage should be outputted to the motor, converting a counter number to actual motor position, showing which coils should be active to achieve that position.

The different modules are controlled with an control FSM, which sends various control signals at different states, corresponding to a specific instruction which is read from the instruction memory.

The motor is connected with GPIO pins on the FPGA board which we had to configure to be GPIO-0.

Throughout the creation of the ASIP, many small errors came up. Errors such as a wrong delay counter and improper control signals were isolated by analyzing a functional simulation of the top level module, using a top down approach.

Individual Modules:

Arithmetic Logic Unit:

```
module alu (input add_sub, set_low, set_high, input [7:0] operanda , operandb, output reg [7:0] result);
   always@(*)
          begin
                 case ({set_low,set_high})
                        2'b10: result = {operanda[7:4], operandb[3:0]};
                        2'b01: result = {operandb[3:0], operanda[3:0]};
                        2'b00:
                               begin
                                      if(~add_sub) result = operanda + operandb;
                                      else result = operanda - operandb;
                               end
                        2'b11: result = 8'b0;
                 endcase
          end
endmodule
```

This module performs addition, subtraction, set high and set low operations on two inputted operands from the operation multiplexers to be later stored in a register or the program counter. The set high and set low operation are usually used to insert data to a register.

Branch Logic:

```
module branch_logic (input [7:0] register0, output branch);
   assign branch = (register0==8'b0)?1'b1:1'b0;
endmodule
```

This branch logic is specifically for the BRZ instruction, acting as an if statement. It will send a signal to the control if register 0 is equal to 0.

Decoder:

```
module decoder (input [5:0] instruction,
                                output br, brz, addi, subi, sr0, srh0, clr, mov, mova, movr, movrhs, pause
);

   assign br = (instruction[5:3] == 3'b100);
   assign clr = (instruction[5:0] == 6'b011000);
   assign brz = (instruction[5:3] == 3'b101);
   assign addi = (instruction[5:3] == 3'b000);
   assign subi = (instruction[5:3] == 3'b001);
   assign sr0 = (instruction[5:2] == 4'b0100);
   assign srh0 = (instruction[5:2] == 4'b0101);
   assign mov = (instruction[5:2] == 4'b0111);
   assign mova = (instruction[5:0] == 6'b110000);
   assign movr = (instruction[5:0] == 6'b110001);
   assign movrhs = (instruction[5:0] == 6'b110010);
   assign pause = (instruction[5:0] == 6'b111111);
endmodule
```

The Decoder figures out what instruction is inputted and sends it to the control FSM

Delay Counter:

```
module delay_counter (input clk, reset_n, start, enable, input [7:0] delay, output reg done);
parameter BASIC_PERIOD=19'd500000;
   reg [7:0] delay_saved;
   reg [18:0] item;
   reg [7:0] item2;
   reg divided_clk;
   always @(posedge clk)//clock divider
   begin
         if(start) delay_saved = delay;
         if(!enable)
         begin
               done = 0;
               item2 = 0;
         end
         if (item == BASIC_PERIOD) //if item reached the requested value
               begin//run the counter
                     if(enable)
                           begin
                                 if(item2 == delay_saved)
                                       begin
                                             done = 1;
                                             item2 = 0;
                                       end
                                 else
                                       begin
                                             //done = 0;
                                             item2 = item2 + 1;
                                       end
                           end
                     item = 0;
               end
         else
               begin
                     item = item+1; //otherwise, during normal times, increment the item
               end
   end
endmodule
```

The delay counter would wait for a specified time, then send out a delay done signal. Start loads the delay increment in (amount in terms of 1/100th of a second). Then enable signal will start the counter.

Immediate Extractor:

```
module immediate_extractor (input [4:0] instruction, input [1:0] select, output reg [7:0] immediate);
   always@(*)
   begin
         case (select)
               2'b00: immediate = {{5{instruction[4]}},instruction[4:2]};//3items
               2'b01: immediate = {{4{instruction[3]}},instruction[3:0]};//4items
               2'b10: immediate = {{3{instruction[4]}},instruction[4:0]};//5items
               2'b11: immediate = 8'b0;
         endcase
   end
endmodule
```

The immediate extractor will extract a value from the instruction depending on the select control signal. This is a value that is written with the instruction.

Operation 1 Multiplexer

```
module op1_mux (input [1:0] select, input [7:0] pc, register, register0, position,
                        output reg [7:0] result);
   always@(*)
        begin
                case (select)
                        2'b00: result = pc;
                        2'b01: result = register;
                        2'b10: result = position;
                        2'b11: result = register0;
                endcase
        end
endmodule
```

The operation 1 mux outputs specified data as an operand to the ALU. The data could be from the Program Counter, a selected register from the register file, the position of the motor, and a general purpose register.

Operation 2 Multiplexer

```
module op2_mux (input [1:0] select, input [7:0] register, immediate,
                        output reg [7:0] result);
   always@(*)
        begin
                case (select)
                        2'b00: result = register;
                        2'b01: result = immediate;
                        2'b10: result = 8'b1;
                        2'b11: result = 8'd2;
                endcase
        end
endmodule
```

The operation 2 mux outputs specified data to the ALU, the data could be from a selected register from the register file (though this isn’t used due to a simplification of the system), the immediate value from the operand, an 8 bit data value of 1 and an 8 bit data value of 2.

Program Counter

```
module pc (input clk, reset_n, branch, increment, input [7:0] newpc,
                output reg [7:0] pc);
parameter RESET_LOCATION = 8'h00;
   always@(posedge clk)
        begin
                if(!reset_n) pc = RESET_LOCATION;
                else
                begin
                        if(increment) pc = pc+1;
                        if(branch) pc = newpc;
                end
        end
endmodule
```

The program counter holds the address of the instruction that is to be, or being executed. It is possible to branch forward or backward to a specific instruction, or just increment from the current value.

Register File

```
// This module implements the register file
module regfile (input clk, reset_n, write, input [7:0] data, input [1:0] select0, select1, wr_select,
                        output reg [7:0] selected0, selected1, output [7:0] delay, position, register0);
   // select1 and selected1 are not used as they ae part of a bigger systems
   //write_select chooses to write the data
   // The comment /* synthesis preserve */ after the declaration of a register
   // prevents Quartus from optimizing it, so that it can be observed in simulation
   // It is important that the comment appear before the semicolon
   reg [7:0] reg0 /* synthesis preserve */;
   reg [7:0] reg1 /* synthesis preserve */;
   reg [7:0] reg2 /* synthesis preserve */;
   reg [7:0] reg3 /* synthesis preserve */;
   assign delay = reg3;
   assign position = reg2;
   assign register0 = reg0;
   always @(posedge clk)
   begin
         if(!reset_n)
         begin
               reg0 = 8'b0;
               reg1 = 8'b0;
               reg2 = 8'b0;
               reg3 = 8'b0;
               selected0 = 8'b0;
         end
         else
         begin
               if(write)
                     begin
                           case(wr_select)
                                 2'b00: reg0 = data;
                                 2'b01: reg1 = data;
                                 2'b10: reg2 = data;
                                 2'b11: reg3 = data;
                           endcase
                     end
               case(select0)
                     2'b00: selected0 = reg0;
                     2'b01: selected0 = reg1;
                     2'b10: selected0 = reg2;
                     2'b11: selected0 = reg3;
               endcase
         end
   end
endmodule
```

The Register file holds relevent data of the motor. This includes position (reg2), delay (reg3), and two general purpose registers (reg0 and reg1). New data can be stored into the register, or data could be read from the registers as select0. The register to be written is selected with wr_select and the data to be read is select0.

Result Multiplexer

```
module result_mux (
    input select_result,
    input [7:0] alu_result,
    output [7:0] result
);
    assign result = (select_result)?alu_result:8'b0;
endmodule
```

The result mux sends out data to the register file. It select_result is high, then it will output data from the ALU, if select_result is low, then it will output a value of 0. The result mux is connected to the data port in the register file.

Temporary Register

```
module temp_register (input clk, reset_n, load, increment, decrement, input [7:0] data,
                                        output reg [7:0] counter,//test
                                        output negative, positive, zero
                                        );
   //reg [7:0] counter;
   assign negative = counter[7];
   assign zero = (counter == 8'b0);
   assign positive =(~zero&~negative);
   always@(posedge clk)
          begin
                 if(!reset_n) counter = 8'b0;
                 else
                 begin
                        if(load)
                               begin
                                      counter = data;
                               end
                        if(increment) counter = counter + 1;
                        if(decrement) counter = counter - 1;
                 end
          end
endmodule
```

This is a register used to determine which direction a motor should spin based on the value inputted into the data port. The motor will turn in a specified direction for a specified amount of steps. The motor will turn counter clockwise if the value is negative, clockwise if the value is positive. If the value is zero, the motor won't turn. For example, if -3 is inputted, the motor will spin 3 steps counter clockwise.

Write Address Select

```
module write_address_select (input [1:0] select, input [1:0] reg_field0, reg_field1, output reg [1:0]
write_address);
    always@(*)
    begin
            case (select)
                    2'b00: write_address = 2'b0;
                    2'b01: write_address = reg_field0;
                    2'b10: write_address = reg_field1;
                    2'b11: write_address = 2'b10;
            endcase
    end


endmodule
```

This module takes in input from a control signal from the control FSM to determine which register in the regfile to be written to. The selected output is to the register file. Possible selections include, register 0 (a general purpose register), a register value parsed from the last two bits of the instruction ([1:0]), the second last pair of bits parsed from the instruction ([3:2]) or, register 2 (position).

Memory:

Instruction ROM instruction file:

```
-- Copyright (C) 1991-2013 Altera Corporation
-- Your use of Altera Corporation's design tools, logic functions
-- and other software and tools, and its AMPP partner logic
-- functions, and any output files from any of the foregoing
-- (including device programming or simulation files), and any
-- associated documentation or information are expressly subject
-- to the terms and conditions of the Altera Program License
-- Subscription Agreement, Altera MegaCore Function License
-- Agreement, or other applicable license agreement, including,
-- without limitation, that your use is for the sole purpose of
-- programming logic devices manufactured by Altera and sold by
-- Altera or its authorized distributors.  Please refer to the
-- applicable agreement for further details.

-- Quartus II generated Memory Initialization File (.mif)

WIDTH=8;
DEPTH=256;

ADDRESS_RADIX=UNS;
DATA_RADIX=BIN;

CONTENT BEGIN
    0      :  01100000;
    1      :  01000001;
    2      :  01111100;
    3      :  01000000;
    4      :  01010011;
    5      :  01110100;
    6      :  01100000;
    7      :  01001010;
    8      :  11000101;
    9      :  00100100;
    10     :  10100010;
    11     :  10011101;
    [12..19]  :  11111111;
    20     :  01000000;
    21     :  01011101;
    22     :  01110100;
    23     :  01100000;
    24     :  01001010;
    25     :  11000101;
    26     :  00100100;
    27     :  10100010;
    28     :  10011101;
```

```
    [29..37]  :   11111111;
    38   :   11001001;
    39   :   10000000;
    [40..255]  :   00000000;
END;
```

Thes are the instructions to be executed, written in machine code. Specifically, this code will turn the motor 10 full turns clockwise using half stepping, 10 full turn counter clockwise using full stepping, and finally, half a turn counter clockwise using half stepping

Instruction ROM Module

```
// megafunction wizard: %ROM: 1-PORT%
// GENERATION: STANDARD
// VERSION: WM1.0
// MODULE: altsyncram

// ============================================================
// File Name: instruction_rom.v
// Megafunction Name(s):
//                     altsyncram
//
// Simulation Library Files(s):
//                     altera_mf
// ============================================================
// ************************************************************
// THIS IS A WIZARD-GENERATED FILE. DO NOT EDIT THIS FILE!
//
// 13.0.1 Build 232 06/12/2013 SP 1 SJ Full Version
// ************************************************************


//Copyright (C) 1991-2013 Altera Corporation
//Your use of Altera Corporation's design tools, logic functions
//and other software and tools, and its AMPP partner logic
//functions, and any output files from any of the foregoing
//(including device programming or simulation files), and any
//associated documentation or information are expressly subject
//to the terms and conditions of the Altera Program License
//Subscription Agreement, Altera MegaCore Function License
//Agreement, or other applicable license agreement, including,
//without limitation, that your use is for the sole purpose of
//programming logic devices manufactured by Altera and sold by
//Altera or its authorized distributors.  Please refer to the
//applicable agreement for further details.
```

```
// synopsys translate_off
`timescale 1 ps / 1 ps
// synopsys translate_on
module instruction_rom (
    address,
    clock,
    q);

    input    [7:0]  address;
    input      clock;
    output    [7:0]  q;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_off
`endif
    tri1      clock;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_on
`endif

    wire [7:0] sub_wire0;
    wire [7:0] q = sub_wire0[7:0];

    altsyncram    altsyncram_component (
                .address_a (address),
                .clock0 (clock),
                .q_a (sub_wire0),
                .aclr0 (1'b0),
                .aclr1 (1'b0),
                .address_b (1'b1),
                .addressstall_a (1'b0),
                .addressstall_b (1'b0),
                .byteena_a (1'b1),
                .byteena_b (1'b1),
                .clock1 (1'b1),
                .clocken0 (1'b1),
                .clocken1 (1'b1),
                .clocken2 (1'b1),
                .clocken3 (1'b1),
                .data_a ({8{1'b1}}),
                .data_b (1'b1),
                .eccstatus (),
                .q_b (),
                .rden_a (1'b1),
                .rden_b (1'b1),
                .wren_a (1'b0),
                .wren_b (1'b0));
    defparam
```

```
        altsyncram_component.clock_enable_input_a = "BYPASS",
        altsyncram_component.clock_enable_output_a = "BYPASS",
        altsyncram_component.init_file = "instruction_rom.mif",
        altsyncram_component.intended_device_family = "Cyclone II",
        altsyncram_component.lpm_hint = "ENABLE_RUNTIME_MOD=NO",
        altsyncram_component.lpm_type = "altsyncram",
        altsyncram_component.numwords_a = 256,
        altsyncram_component.operation_mode = "ROM",
        altsyncram_component.outdata_aclr_a = "NONE",
        altsyncram_component.outdata_reg_a = "UNREGISTERED",
        altsyncram_component.widthad_a = 8,
        altsyncram_component.width_a = 8,
        altsyncram_component.width_byteena_a = 1;


endmodule

// ============================================================
// CNX file retrieval info
// ============================================================
// Retrieval info: PRIVATE: ADDRESSSTALL_A NUMERIC "0"
// Retrieval info: PRIVATE: AclrAddr NUMERIC "0"
// Retrieval info: PRIVATE: AclrByte NUMERIC "0"
// Retrieval info: PRIVATE: AclrOutput NUMERIC "0"
// Retrieval info: PRIVATE: BYTE_ENABLE NUMERIC "0"
// Retrieval info: PRIVATE: BYTE_SIZE NUMERIC "8"
// Retrieval info: PRIVATE: BlankMemory NUMERIC "0"
// Retrieval info: PRIVATE: CLOCK_ENABLE_INPUT_A NUMERIC "0"
// Retrieval info: PRIVATE: CLOCK_ENABLE_OUTPUT_A NUMERIC "0"
// Retrieval info: PRIVATE: Clken NUMERIC "0"
// Retrieval info: PRIVATE: IMPLEMENT_IN_LES NUMERIC "0"
// Retrieval info: PRIVATE: INIT_FILE_LAYOUT STRING "PORT_A"
// Retrieval info: PRIVATE: INIT_TO_SIM_X NUMERIC "0"
// Retrieval info: PRIVATE: INTENDED_DEVICE_FAMILY STRING "Cyclone II"
// Retrieval info: PRIVATE: JTAG_ENABLED NUMERIC "0"
// Retrieval info: PRIVATE: JTAG_ID STRING "NONE"
// Retrieval info: PRIVATE: MAXIMUM_DEPTH NUMERIC "0"
// Retrieval info: PRIVATE: MIFfilename STRING "instruction_rom.mif"
// Retrieval info: PRIVATE: NUMWORDS_A NUMERIC "256"
// Retrieval info: PRIVATE: RAM_BLOCK_TYPE NUMERIC "0"
// Retrieval info: PRIVATE: RegAddr NUMERIC "1"
// Retrieval info: PRIVATE: RegOutput NUMERIC "0"
// Retrieval info: PRIVATE: SYNTH_WRAPPER_GEN_POSTFIX STRING "0"
// Retrieval info: PRIVATE: SingleClock NUMERIC "1"
// Retrieval info: PRIVATE: UseDQRAM NUMERIC "0"
// Retrieval info: PRIVATE: WidthAddr NUMERIC "8"
// Retrieval info: PRIVATE: WidthData NUMERIC "8"
// Retrieval info: PRIVATE: rden NUMERIC "0"
```

```
// Retrieval info: LIBRARY: altera_mf altera_mf.altera_mf_components.all
// Retrieval info: CONSTANT: CLOCK_ENABLE_INPUT_A STRING "BYPASS"
// Retrieval info: CONSTANT: CLOCK_ENABLE_OUTPUT_A STRING "BYPASS"
// Retrieval info: CONSTANT: INIT_FILE STRING "instruction_rom.mif"
// Retrieval info: CONSTANT: INTENDED_DEVICE_FAMILY STRING "Cyclone II"
// Retrieval info: CONSTANT: LPM_HINT STRING "ENABLE_RUNTIME_MOD=NO"
// Retrieval info: CONSTANT: LPM_TYPE STRING "altsyncram"
// Retrieval info: CONSTANT: NUMWORDS_A NUMERIC "256"
// Retrieval info: CONSTANT: OPERATION_MODE STRING "ROM"
// Retrieval info: CONSTANT: OUTDATA_ACLR_A STRING "NONE"
// Retrieval info: CONSTANT: OUTDATA_REG_A STRING "UNREGISTERED"
// Retrieval info: CONSTANT: WIDTHAD_A NUMERIC "8"
// Retrieval info: CONSTANT: WIDTH_A NUMERIC "8"
// Retrieval info: CONSTANT: WIDTH_BYTEENA_A NUMERIC "1"
// Retrieval info: USED_PORT: address 0 0 8 0 INPUT NODEFVAL "address[7..0]"
// Retrieval info: USED_PORT: clock 0 0 0 0 INPUT VCC "clock"
// Retrieval info: USED_PORT: q 0 0 8 0 OUTPUT NODEFVAL "q[7..0]"
// Retrieval info: CONNECT: @address_a 0 0 8 0 address 0 0 8 0
// Retrieval info: CONNECT: @clock0 0 0 0 0 clock 0 0 0 0
// Retrieval info: CONNECT: q 0 0 8 0 @q_a 0 0 8 0
// Retrieval info: GEN_FILE: TYPE_NORMAL instruction_rom.v TRUE
// Retrieval info: GEN_FILE: TYPE_NORMAL instruction_rom.inc FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL instruction_rom.cmp FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL instruction_rom.bsf FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL instruction_rom_inst.v FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL instruction_rom_bb.v FALSE
// Retrieval info: LIB_FILE: altera_mf
```

The Instruction ROM Module is used to interface other modules with the instructions to be executed, stored on the on chip memory of the Altera DE2 board. Generated by the MegaWizard Plugin on the DE2 board.

Stepper ROM Instruction set:

-- Quartus II generated Memory Initialization File (.mif)

```
WIDTH=4;
DEPTH=8;

ADDRESS_RADIX=UNS;
DATA_RADIX=BIN;

CONTENT BEGIN
   0  :  0010;
   1  :  0110;
   2  :  0100;
   3  :  0101;
   4  :  0001;
   5  :  1001;
   6  :  1000;
   7  :  1010;
END;
```

This is the sequence of states needed to control the motor to move a certain direction. The position register (reg2) in the register file is used to choose what which state is read from this ROM.

Stepper ROM Module

```
// megafunction wizard: %ROM: 1-PORT%
// GENERATION: STANDARD
// VERSION: WM1.0
// MODULE: altsyncram

// ============================================================
// File Name: stepper_rom.v
// Megafunction Name(s):
//                      altsyncram
//
// Simulation Library Files(s):
//                      altera_mf
// ============================================================
// ************************************************************
// THIS IS A WIZARD-GENERATED FILE. DO NOT EDIT THIS FILE!
//
// 13.0.1 Build 232 06/12/2013 SP 1 SJ Full Version
// ************************************************************


//Copyright (C) 1991-2013 Altera Corporation
//Your use of Altera Corporation's design tools, logic functions
//and other software and tools, and its AMPP partner logic
//functions, and any output files from any of the foregoing
//(including device programming or simulation files), and any
//associated documentation or information are expressly subject
//to the terms and conditions of the Altera Program License
//Subscription Agreement, Altera MegaCore Function License
//Agreement, or other applicable license agreement, including,
//without limitation, that your use is for the sole purpose of
//programming logic devices manufactured by Altera and sold by
//Altera or its authorized distributors.  Please refer to the
//applicable agreement for further details.


// synopsys translate_off
`timescale 1 ps / 1 ps
// synopsys translate_on
module stepper_rom (
    address,
    clock,
    q);

    input    [2:0]  address;
    input      clock;
    output    [3:0]  q;
```

```
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_off
`endif
    tri1      clock;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_on
`endif

    wire [3:0] sub_wire0;
    wire [3:0] q = sub_wire0[3:0];

    altsyncram    altsyncram_component (
                .address_a (address),
                .clock0 (clock),
                .q_a (sub_wire0),
                .aclr0 (1'b0),
                .aclr1 (1'b0),
                .address_b (1'b1),
                .addressstall_a (1'b0),
                .addressstall_b (1'b0),
                .byteena_a (1'b1),
                .byteena_b (1'b1),
                .clock1 (1'b1),
                .clocken0 (1'b1),
                .clocken1 (1'b1),
                .clocken2 (1'b1),
                .clocken3 (1'b1),
                .data_a ({4{1'b1}}),
                .data_b (1'b1),
                .eccstatus (),
                .q_b (),
                .rden_a (1'b1),
                .rden_b (1'b1),
                .wren_a (1'b0),
                .wren_b (1'b0));
    defparam
        altsyncram_component.clock_enable_input_a = "BYPASS",
        altsyncram_component.clock_enable_output_a = "BYPASS",
        altsyncram_component.init_file = "stepper_rom.mif",
        altsyncram_component.intended_device_family = "Cyclone II",
        altsyncram_component.lpm_hint = "ENABLE_RUNTIME_MOD=NO",
        altsyncram_component.lpm_type = "altsyncram",
        altsyncram_component.numwords_a = 8,
        altsyncram_component.operation_mode = "ROM",
        altsyncram_component.outdata_aclr_a = "NONE",
        altsyncram_component.outdata_reg_a = "UNREGISTERED",
        altsyncram_component.widthad_a = 3,
        altsyncram_component.width_a = 4,
```

```
        altsyncram_component.width_byteena_a = 1;


endmodule

// ============================================================
// CNX file retrieval info
// ============================================================
// Retrieval info: PRIVATE: ADDRESSSTALL_A NUMERIC "0"
// Retrieval info: PRIVATE: AclrAddr NUMERIC "0"
// Retrieval info: PRIVATE: AclrByte NUMERIC "0"
// Retrieval info: PRIVATE: AclrOutput NUMERIC "0"
// Retrieval info: PRIVATE: BYTE_ENABLE NUMERIC "0"
// Retrieval info: PRIVATE: BYTE_SIZE NUMERIC "8"
// Retrieval info: PRIVATE: BlankMemory NUMERIC "0"
// Retrieval info: PRIVATE: CLOCK_ENABLE_INPUT_A NUMERIC "0"
// Retrieval info: PRIVATE: CLOCK_ENABLE_OUTPUT_A NUMERIC "0"
// Retrieval info: PRIVATE: Clken NUMERIC "0"
// Retrieval info: PRIVATE: IMPLEMENT_IN_LES NUMERIC "0"
// Retrieval info: PRIVATE: INIT_FILE_LAYOUT STRING "PORT_A"
// Retrieval info: PRIVATE: INIT_TO_SIM_X NUMERIC "0"
// Retrieval info: PRIVATE: INTENDED_DEVICE_FAMILY STRING "Cyclone II"
// Retrieval info: PRIVATE: JTAG_ENABLED NUMERIC "0"
// Retrieval info: PRIVATE: JTAG_ID STRING "NONE"
// Retrieval info: PRIVATE: MAXIMUM_DEPTH NUMERIC "0"
// Retrieval info: PRIVATE: MIFfilename STRING "stepper_rom.mif"
// Retrieval info: PRIVATE: NUMWORDS_A NUMERIC "8"
// Retrieval info: PRIVATE: RAM_BLOCK_TYPE NUMERIC "0"
// Retrieval info: PRIVATE: RegAddr NUMERIC "1"
// Retrieval info: PRIVATE: RegOutput NUMERIC "0"
// Retrieval info: PRIVATE: SYNTH_WRAPPER_GEN_POSTFIX STRING "0"
// Retrieval info: PRIVATE: SingleClock NUMERIC "1"
// Retrieval info: PRIVATE: UseDQRAM NUMERIC "0"
// Retrieval info: PRIVATE: WidthAddr NUMERIC "3"
// Retrieval info: PRIVATE: WidthData NUMERIC "4"
// Retrieval info: PRIVATE: rden NUMERIC "0"
// Retrieval info: LIBRARY: altera_mf altera_mf.altera_mf_components.all
// Retrieval info: CONSTANT: CLOCK_ENABLE_INPUT_A STRING "BYPASS"
// Retrieval info: CONSTANT: CLOCK_ENABLE_OUTPUT_A STRING "BYPASS"
// Retrieval info: CONSTANT: INIT_FILE STRING "stepper_rom.mif"
// Retrieval info: CONSTANT: INTENDED_DEVICE_FAMILY STRING "Cyclone II"
// Retrieval info: CONSTANT: LPM_HINT STRING "ENABLE_RUNTIME_MOD=NO"
// Retrieval info: CONSTANT: LPM_TYPE STRING "altsyncram"
// Retrieval info: CONSTANT: NUMWORDS_A NUMERIC "8"
// Retrieval info: CONSTANT: OPERATION_MODE STRING "ROM"
// Retrieval info: CONSTANT: OUTDATA_ACLR_A STRING "NONE"
// Retrieval info: CONSTANT: OUTDATA_REG_A STRING "UNREGISTERED"
// Retrieval info: CONSTANT: WIDTHAD_A NUMERIC "3"
```

```
// Retrieval info: CONSTANT: WIDTH_A NUMERIC "4"
// Retrieval info: CONSTANT: WIDTH_BYTEENA_A NUMERIC "1"
// Retrieval info: USED_PORT: address 0 0 3 0 INPUT NODEFVAL "address[2..0]"
// Retrieval info: USED_PORT: clock 0 0 0 0 INPUT VCC "clock"
// Retrieval info: USED_PORT: q 0 0 4 0 OUTPUT NODEFVAL "q[3..0]"
// Retrieval info: CONNECT: @address_a 0 0 3 0 address 0 0 3 0
// Retrieval info: CONNECT: @clock0 0 0 0 0 clock 0 0 0 0
// Retrieval info: CONNECT: q 0 0 4 0 @q_a 0 0 4 0
// Retrieval info: GEN_FILE: TYPE_NORMAL stepper_rom.v TRUE
// Retrieval info: GEN_FILE: TYPE_NORMAL stepper_rom.inc FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL stepper_rom.cmp FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL stepper_rom.bsf FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL stepper_rom_inst.v FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL stepper_rom_bb.v FALSE
// Retrieval info: LIB_FILE: altera_mf
```

The Stepper ROM Module is used to interface between the stepper ROM, in the on chip memory of the Altera DE2 board. Generated by the MegaWizard Plugin on the DE2 board.

Datapath:

```
module datapath (input clk, reset_n,
                                // Control signals
                                input write_reg_file, result_mux_select,
                                input [1:0] op1_mux_select, op2_mux_select,
                                input start_delay_counter, enable_delay_counter,
                                input commit_branch, increment_pc,
                                input alu_add_sub, alu_set_low, alu_set_high,
                                input load_temp, increment_temp, decrement_temp,
                                input [1:0] select_immediate,
                                input [1:0] select_write_address,
                                // Status outputs
                                output br, brz, addi, subi, sr0, srh0, clr, mov, mova, movr, movrhs, pause,
                                output delay_done,
                                output temp_is_positive, temp_is_negative, temp_is_zero,
                                output register0_is_zero,
                                //test signals
                                input [7:0] test_in,
                                output [7:0] test_out,
                                output [7:0] test_out1,
                                output [7:0] test_out2,
                                output [7:0] test_out3,
                                output [7:0] test_out4,
                                output [7:0] test_out5,
                                // Motor control outputs
                                output [3:0] stepper_signals

);
// The comment /*synthesis keep*/ after the declaration of a wire
// prevents Quartus from optimizing it, so that it can be observed in simulation
// It is important that the comment appear before the semicolon
wire [7:0] position /*synthesis keep*/;
wire [7:0] delay /*synthesis keep*/;
wire [7:0] register0 /*synthesis keep*/;
wire [7:0] PC /*synthesis keep*/;
wire [7:0] data;
wire [7:0] selected0;
wire [7:0] selected1;
wire [7:0] immediate_operand;
wire [7:0] ALU_out /*synthesis keep*/;
wire [7:0] instruction,operanda,operandb;
wire [1:0] write_select;

assign test_out = instruction;       //test
assign test_out1 = delay;            //test
//assign test_out2 = temp_register;                  //test
assign test_out3 = register0;        //test
```

```
assign test_out4 = position;    //test
assign test_out5 = data;    //test

decoder the_decoder (
    // Inputs
    .instruction (instruction[7:2]),
    // Outputs
    .br (br),
    .brz (brz),
    .addi (addi),
    .subi (subi),
    .sr0 (sr0),
    .srh0 (srh0),
    .clr (clr),
    .mov (mov),
    .mova (mova),
    .movr (movr),
    .movrhs (movrhs),
    .pause (pause)
);
regfile the_regfile(
    // Inputs
    .clk (clk),
    .reset_n (reset_n),
    .write (write_reg_file),
    .data (data),
    .select0 (instruction[1:0]),
    .select1 (instruction[3:2]),
    .wr_select (write_select),
    // Outputs
    .selected0 (selected0),
    .selected1 (selected1),
    .delay (delay),
    .position (position),
    .register0 (register0)
    //.selected0 (test_out),    //test
    //.selected1 (test_out1),    //test
    //.delay (test_out2),     //test
    //.position (test_out3),    //test
    //.register0 (test_out4)    //test
);

op1_mux the_op1_mux(
    // Inputs
    .select (op1_mux_select),
    .pc (PC),
    .register (selected0),
    .register0 (register0),
```

```
    .position (position),
    // Outputs
    .result(operanda)
);

op2_mux the_op2_mux(
    // Inputs
    .select (op2_mux_select),
    .register (selected1),
    .immediate (immediate_operand),
    // Outputs
    .result (operandb)
);

delay_counter the_delay_counter(
    // Inputs
    .clk(clk),
    .reset_n (reset_n),
    .start (start_delay_counter),
    .enable (enable_delay_counter),
    .delay (delay),
    // Outputs
    .done (delay_done)
);

stepper_rom the_stepper_rom(
    // Inputs
    .address (position[2:0]),
    .clock (clk),
    // Outputs
    .q (stepper_signals)
);

pc the_pc(
    // Inputs
    .clk (clk),
    .reset_n (reset_n),
    .branch (commit_branch),
    .increment (increment_pc),
    .newpc (ALU_out),
    //.newpc(test_in),    //test
    // Outputs
    //.pc (test_out),    //test
    .pc (PC)
);

instruction_rom the_instruction_rom(
    // Inputs
```

```
    .address (PC),
    .clock (clk),
    // Outputs
    .q (instruction)
    //.q(test_out)              //test
);
//assign instruction = test_in;       //test
alu the_alu(
    // Inputs
    .add_sub (alu_add_sub),
    .set_low (alu_set_low),
    .set_high (alu_set_high),
    .operanda (operanda),
    .operandb (operandb),
    // Outputs
    .result (ALU_out)
);

temp_register the_temp_register(
    // Inputs
    .clk (clk),
    .reset_n (reset_n),
    .load (load_temp),
    .increment (increment_temp),
    .decrement (decrement_temp),
    .data (selected0),
    // Outputs
    .negative (temp_is_negative),
    .positive (temp_is_positive),
    .counter(test_out2),    //test
    .zero (temp_is_zero)
);

immediate_extractor the_immediate_extractor(
    // Inputs
    .instruction (instruction),
    .select (select_immediate),
    // Outputs
    .immediate (immediate_operand)
);

write_address_select the_write_address_select(
    // Inputs
    .select (select_write_address),
    .reg_field0 (instruction[1:0]),
    .reg_field1 (instruction[3:2]),
    // Outputs
    .write_address(write_select)
```

```
);

result_mux the_result_mux (
    .select_result (result_mux_select),
    .alu_result (ALU_out),
    .result (data)
);

branch_logic the_branch_logic(
    // Inputs
    .register0 (register0),
    // Outputs
    .branch (register0_is_zero)
);

endmodule
```

The datapath connects all the different modules together. It requires control signals from the FSM to correctly run an instruction.

Control FSM:

```
module control_fsm (
    input clk, reset_n,
    // Status inputs
    input br, brz, addi, subi, sr0, srh0, clr, mov, mova, movr, movrhs, pause,
    input delay_done,
    input temp_is_positive, temp_is_negative, temp_is_zero,
    input register0_is_zero,
    // Control signal outputs
    output reg write_reg_file,
    output reg result_mux_select,
    output reg [1:0] op1_mux_select, op2_mux_select,
    output reg start_delay_counter, enable_delay_counter,
    output reg commit_branch, increment_pc,
    output reg alu_add_sub, alu_set_low, alu_set_high,
    output reg load_temp_register, increment_temp_register, decrement_temp_register,
    output reg [1:0] select_immediate,
    output reg [5:0] state, //for testing
    output reg [1:0] select_write_address

);
parameter RESET=5'b00000, FETCH=5'b00001, DECODE=5'b00010,
                        BR=5'b00011, BRZ=5'b00100, ADDI=5'b00101, SUBI=5'b00110, SR0=5'b00111,
                        SRH0=5'b01000, CLR=5'b01001, MOV=5'b01010, MOVA=5'b01011,
                        MOVR=5'b01100, MOVRHS=5'b01101, PAUSE=5'b01110, MOVR_STAGE2=5'b01111,
                        MOVR_DELAY=5'b10000, MOVRHS_STAGE2=5'b10001,
MOVRHS_DELAY=5'b10010,
                        PAUSE_DELAY=5'b10011;

//reg [5:0] state;
reg [5:0] next_state_logic; // NOT REALLY A REGISTER!!!

// Next state logic
always@(*)
begin
    case(state)
            RESET: next_state_logic = FETCH;
            FETCH: next_state_logic = DECODE;
            DECODE:
            begin
                    if(br) next_state_logic = BR;
                    else if(brz) next_state_logic = BRZ;
                    else if(addi) next_state_logic = ADDI;
                    else if(subi) next_state_logic = SUBI;
                    else if(sr0) next_state_logic = SR0;
                    else if(srh0) next_state_logic = SRH0;
                    else if(clr) next_state_logic = CLR;
```

```
                else if(mov) next_state_logic = MOV;
                else if(movr) next_state_logic = MOVR;
                else if(movrhs) next_state_logic = MOVRHS;
                else if(pause) next_state_logic = PAUSE;
                else next_state_logic = RESET;
            end
            BR: next_state_logic = FETCH;
            BRZ: next_state_logic = FETCH;
            ADDI: next_state_logic = FETCH;
            SUBI: next_state_logic = FETCH;
            SR0: next_state_logic = FETCH;
            SRH0: next_state_logic = FETCH;
            CLR: next_state_logic = FETCH;
            MOV: next_state_logic = FETCH;
            MOVR: next_state_logic = MOVR_STAGE2;
            MOVRHS: next_state_logic = MOVRHS_STAGE2;
            PAUSE: next_state_logic = PAUSE_DELAY;
            MOVR_STAGE2:
                begin
                    if(temp_is_zero) next_state_logic = FETCH;
                    else next_state_logic = MOVR_DELAY;
                end
            MOVR_DELAY:
                begin
                    if(delay_done) next_state_logic = MOVR_STAGE2;
                    else next_state_logic = MOVR_DELAY;
                end
            MOVRHS_STAGE2:
                 begin
                    if(temp_is_zero) next_state_logic = FETCH;
                    else next_state_logic = MOVRHS_DELAY;
                end
            MOVRHS_DELAY:
                 begin
                    if(delay_done) next_state_logic = MOVRHS_STAGE2;
                    else next_state_logic = MOVRHS_DELAY;
                end
            PAUSE_DELAY:
                 begin
                    if(delay_done) next_state_logic = FETCH;
                    else next_state_logic = PAUSE_DELAY;
                end
            default:next_state_logic=RESET;
       endcase
    end

    // State register
    always@(posedge clk)
```

```
    begin
            if(!reset_n) state = RESET;
            else state = next_state_logic;
    end
// Output logic
always@(*)
    begin
    write_reg_file = 1'b0;
    result_mux_select = 1'b0;
    op1_mux_select = 2'b00;
    op2_mux_select = 2'b00;
    start_delay_counter = 1'b0;
    enable_delay_counter = 1'b0;
    commit_branch = 1'b0;
    increment_pc = 1'b0;
    alu_add_sub = 1'b0;
    alu_set_low = 1'b0;
    alu_set_high = 1'b0;
    load_temp_register = 1'b0;
    increment_temp_register = 1'b0;
    decrement_temp_register = 1'b0;
    select_immediate = 2'b00;
    select_write_address = 2'b00;
    case(state)
            ADDI:
            begin
                        write_reg_file = 1'b1;
                        result_mux_select = 1'b1;
                        op1_mux_select = 2'b01;
                        op2_mux_select = 2'b01;
                        increment_pc = 1'b1;
                        select_write_address = 2'b01;
            end
            SUBI:
            begin
              write_reg_file = 1'b1;
              result_mux_select = 1'b1;
              op1_mux_select = 2'b01;
              op2_mux_select = 2'b01;
              increment_pc = 1'b1;
              alu_add_sub = 1'b1;
              select_write_address = 2'b01;
            end
            MOV:
            begin
                    write_reg_file = 1'b1;
                    result_mux_select = 1'b1;
                    op1_mux_select = 2'b01;
```

```
		op2_mux_select = 2'b01;
		increment_pc = 1'b1;
		alu_add_sub = 1'b0;
		alu_set_low = 1'b0;
		alu_set_high = 1'b0;
		select_immediate = 2'b11;
		select_write_address = 2'b10;
end
SR0:
begin
		write_reg_file = 1'b1;
		result_mux_select = 1'b1; //write data;
		op1_mux_select = 2'b11;//select register0;
		op2_mux_select = 2'b01; //select immediate;
		increment_pc = 1'b1; //increment the PC;
		alu_set_low = 1'b1;
		select_immediate = 2'b01;
end
SRH0:
begin
		write_reg_file = 1'b1;
		result_mux_select = 1'b1; //write data;
		op1_mux_select = 2'b11; //select register0;
		op2_mux_select = 2'b01; //select immediate;
		increment_pc = 1'b1; //increment the PC;
		alu_set_high = 1'b1;
		select_immediate = 2'b01;
end
CLR:
begin
		write_reg_file = 1'b1;
		increment_pc = 1'b1;
		select_write_address = 2'b01;
end
BR:
begin
		op2_mux_select = 2'b01;
		select_immediate = 2'b10;
		commit_branch = 1'b1;
end
BRZ:
begin
		op2_mux_select = 2'b01;
		select_immediate = 2'b10;
		if(register0_is_zero)
			begin
				commit_branch = 1'b1;
			end
```

```
		else
			begin
				increment_pc = 1'b1;
			end
end
MOVR:load_temp_register = 1'b1;
MOVR_STAGE2:
begin//begin stage 2
		if (temp_is_zero)
			begin
				increment_pc = 1'b1;
				//start_delay_counter = 1'b1;
				//enable_delay_counter = 1'b0;
			end
		else if(temp_is_positive)
			begin
				start_delay_counter = 1'b1;
				decrement_temp_register = 1'b1;
				op1_mux_select = 2'b10;
				op2_mux_select = 2'b11;//full step;
				write_reg_file = 1'b1;
				result_mux_select = 1'b1;
				select_write_address = 2'b11;
			end
		else if (temp_is_negative)
			begin
				start_delay_counter = 1'b1;
				op1_mux_select = 2'b10;
				op2_mux_select = 2'b11; //full step;
				alu_add_sub = 1'b1;
				write_reg_file = 1'b1;
				result_mux_select = 1'b1;
				increment_temp_register = 1'b1;
				select_write_address = 2'b11;
			end
		else increment_pc = 1'b0;
end
MOVR_DELAY:
begin
		enable_delay_counter = 1'b1;
		start_delay_counter = 1'b1;
end
MOVRHS: load_temp_register = 1'b1;
MOVRHS_STAGE2:
begin//begin stage 2
		if (temp_is_zero)
			begin
				increment_pc = 1'b1;
```

```
                        start_delay_counter = 1'b1;
                        enable_delay_counter = 1'b0;
                    end
                else if(temp_is_positive)
                    begin
                        start_delay_counter = 1'b1;
                        enable_delay_counter = 1'b0;
                        decrement_temp_register = 1'b1;
                        op1_mux_select = 2'b10;
                        op2_mux_select = 2'b10;//half step;
                        write_reg_file = 1'b1;
                        result_mux_select = 1'b1;
                        increment_temp_register = 1'b0;
                        select_write_address = 2'b11;
                    end
                else if (temp_is_negative)
                    begin
                        increment_pc = 1'b0;
                        start_delay_counter = 1'b1;
                        enable_delay_counter = 1'b0;
                        decrement_temp_register = 1'b0;
                        op1_mux_select = 2'b10;
                        op2_mux_select = 2'b10; //half step;
                        alu_add_sub = 1'b1;
                        write_reg_file = 1'b1;
                        result_mux_select = 1'b1;
                        increment_temp_register = 1'b1;
                        select_write_address = 2'b11;
                    end
                else increment_pc = 1'b0;
            end
            MOVRHS_DELAY:
            begin
                enable_delay_counter = 1'b1;
                start_delay_counter = 1'b1;
            end
            PAUSE: start_delay_counter = 1'b1;
            PAUSE_DELAY:
                begin
                    //start_delay_counter = 1'b1;
                    enable_delay_counter = 1'b1;
                    if (delay_done) increment_pc = 1'b1;
                end
            default: increment_pc = 1'b0;
    endcase
end

endmodule
```

The control FSM sends control signals to the datapath modules. It activates the necessary modules, and sends them the signals needed to run an instruction.

Lab5:

```
// This is the top-level file for lab 5 that connects the datapath and the control unit
module lab5 (input clk, reset_n, output [3:0] stepper_signals, output [3:0] LEDG,
output [5:0] state, // test
output [7:0] test_out,
output [7:0] test_out1,
output [7:0] test_out2,
output [7:0] test_out3,
output [7:0] test_out4,
output [7:0] test_out5
);

wire br, brz, addi, subi, sr0, srh0, clr, mov, mova, movr, movrhs, pause;
wire delay_done;
wire temp_is_positive, temp_is_negative, temp_is_zero;
wire register0_is_zero;

wire write_reg_file, result_mux_select;
wire [1:0] op1_mux_select, op2_mux_select;
wire start_delay_counter, enable_delay_counter;
wire commit_branch, increment_pc;
wire alu_add_sub, alu_set_low, alu_set_high;
wire load_temp_register, increment_temp_register, decrement_temp_register;
wire [1:0] select_immediate;
wire [1:0] select_write_address;

// connect stepper signals to the green LEDs, so that we can observe them visually
assign LEDG = stepper_signals;

control_fsm the_control_fsm (
    .state(state), //test
    .clk (clk),
    .reset_n (reset_n),
    // Status inputs
    .br (br),
    .brz (brz),
    .addi (addi),
    .subi (subi),
    .sr0 (sr0),
    .srh0 (srh0),
    .clr (clr),
    .mov (mov),
    .mova (mova),
    .movr (movr),
```

```
    .movrhs (movrhs),
    .pause (pause),
    .delay_done (delay_done),
    .temp_is_positive (temp_is_positive),
    .temp_is_negative (temp_is_negative),
    .temp_is_zero (temp_is_zero),
    .register0_is_zero (register0_is_zero),
    // Control signal outputs
    .write_reg_file (write_reg_file),
    .result_mux_select (result_mux_select),
    .op1_mux_select (op1_mux_select),
    .op2_mux_select (op2_mux_select),
    .start_delay_counter (start_delay_counter),
    .enable_delay_counter (enable_delay_counter),
    .commit_branch (commit_branch),
    .increment_pc (increment_pc),
    .alu_add_sub (alu_add_sub),
    .alu_set_low (alu_set_low),
    .alu_set_high (alu_set_high),
    .load_temp_register (load_temp_register),
    .increment_temp_register (increment_temp_register),
    .decrement_temp_register (decrement_temp_register),
    .select_immediate (select_immediate),
    .select_write_address (select_write_address)
);

datapath the_datapath (
    .clk (clk),
    .reset_n (reset_n),
    // Control signals
    .write_reg_file (write_reg_file),
    .result_mux_select (result_mux_select),
    .op1_mux_select (op1_mux_select),
    .op2_mux_select (op2_mux_select),
    .start_delay_counter (start_delay_counter),
    .enable_delay_counter (enable_delay_counter),
    .commit_branch (commit_branch),
    .increment_pc (increment_pc),
    .alu_add_sub (alu_add_sub),
    .alu_set_low (alu_set_low),
    .alu_set_high (alu_set_high),
    .load_temp (load_temp_register),
    .increment_temp (increment_temp_register),
    .decrement_temp (decrement_temp_register),
    .select_immediate (select_immediate),
    .select_write_address (select_write_address),
    // Status outputs
    .br (br),
```

```
    .brz (brz),
    .addi (addi),
    .subi (subi),
    .sr0 (sr0),
    .srh0 (srh0),
    .clr (clr),
    .mov (mov),
    .mova (mova),
    .movr (movr),
    .movrhs (movrhs),
    .pause (pause),
    .delay_done (delay_done),
    .temp_is_positive (temp_is_positive),
    .temp_is_negative (temp_is_negative),
    .temp_is_zero (temp_is_zero),
    .register0_is_zero (register0_is_zero),
    // Temporary outputs for debugging purposes
    .test_out(test_out),
    .test_out1(test_out1),
    .test_out2(test_out2),
    .test_out3(test_out3),
    .test_out4(test_out4),
    .test_out5(test_out5),
    // Motor control outputs
    .stepper_signals (stepper_signals)
);


endmodule
```

The lab 5 module connects the datapath and the control FSM together. This is the top level module to synthesize.