Mechtron 3TB4: Lab5

Stepper Motor Controller ASIP Design

Reports Due:
Pre-Lab Part : At the start of the tutorial session.
Lab Report: April 9th.

1 Goals

- To practice building, debugging and testing a complex digital system
- To learn how to build circuits based on datapath and control unit descriptions
- To learn how to build FSMs using Verilog
- To gain practice using the on-chip memory modules available on the FPGA
- To learn how a simple processor operates on a cycle-by-cycle basis
- To learn how assembler code gets converted into machine code

Note: This lab consists of four activities. Marks allocated to each are indicated in square brackets.

2 Introduction

In this lab you will build an Application Specific Instruction Set Processor (ASIP) for controlling a stepper motor. The processor will support instructions that can move the motor by a number of steps or half steps. The processor will also contain special purpose registers that will keep track of the motor's current position, and allow the programmer to precisely control the time delay between steps. Based on the skeleton code that will be provided to you and the instructions provided in class, you will design circuitry (using Verilog) that implements the datapath and control unit for the processor. Once the processor design is done, you will write a few simple programs in machine language for the newly built processor to test its functionality. The processor you are designing is mimicking a real product from Pontech (see STP100/STP101 Users Manual). The original product was stripped of many features, to make it suitable for a lab. The above link is provided as a reference only, and will not be otherwise useful for completing this lab.

In this lab, you will continue to use the Cyclone V FPGA available on the DE1-SoC development board. You will build your processor fully on the FPGA, taking advantage of the on-chip memory available in the FPGA. You will use Quartus Prime to write the Verilog code, simulate and test the resulting circuitry. You should use Signal Tap Logic Analyzer to debug your circuit once it is implemented in the FPGA.

2.1 System Description

The system you are building is shown in Figure 1. The FPGA implementing the processor will be connected to the motor through one of the expansion headers on the DE1-SoC board.

Figure 2 shows the GPIO pins layout of the expansion headers of the DE1-SoC boards. In Figure 2, the VCC5 pin (for 5V), VCC3 pin(for 3.3V) and GND pins do not have FPGA pin assignment. In the pin assignment table in the user manual, these pins are not listed.

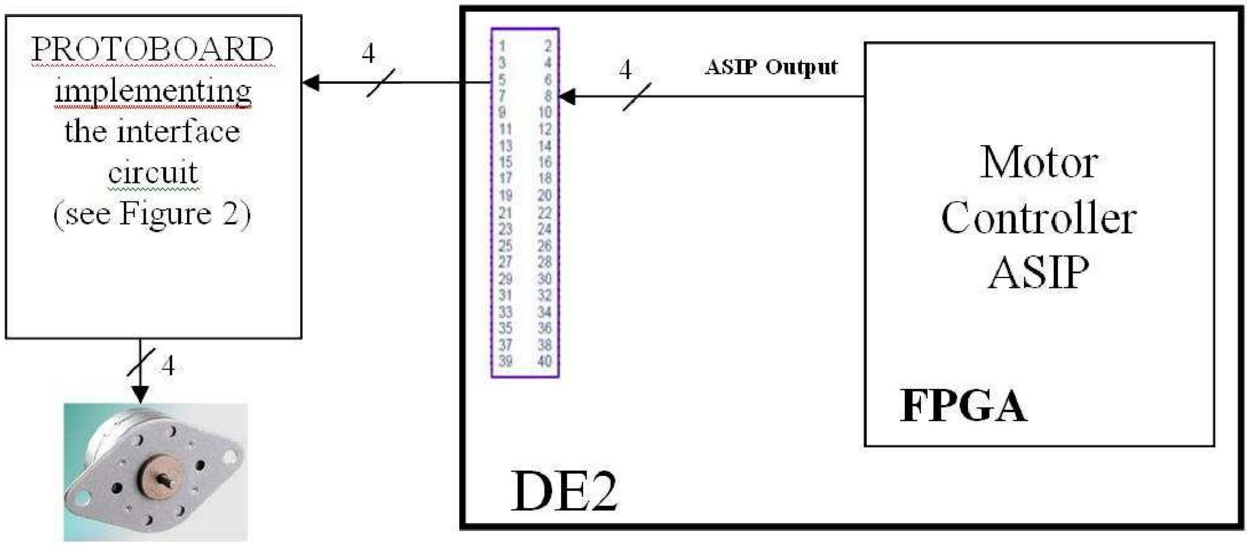

PROTOBOARD implementing the interface circuit (see Figure 2)
4
4
ASIP Output
Motor Controller ASIP
FPGA
DE2

Figure 1: System Block Diagram

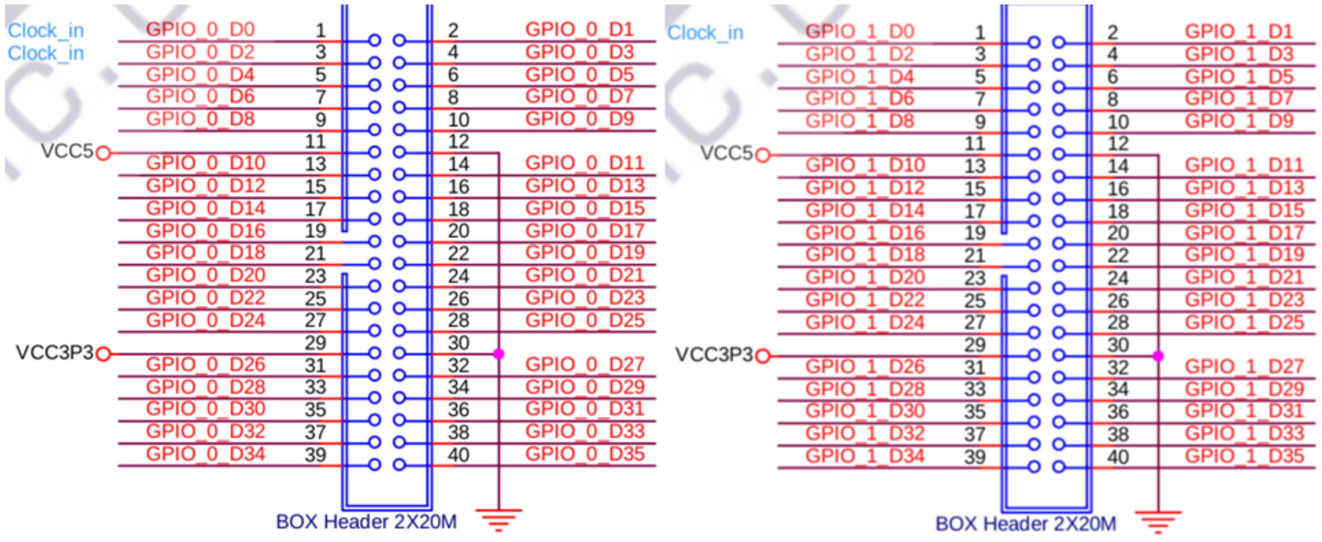


Figure 2: DE1-SoC GPIO Pins Layout

Since the board does not have enough power to directly drive the motor, you will build a motor driver interface circuit. Figure 3 shows a schematic diagram of the interface circuit using the SN754410NE driver chip.

You will be designing an ASIP processor that has four registers in the register file, and implements 12 instructions. Both instructions and data in this processor are 8 bits wide. Some instructions are specific to the target application, while others are generic and provide support for basic loop operations.

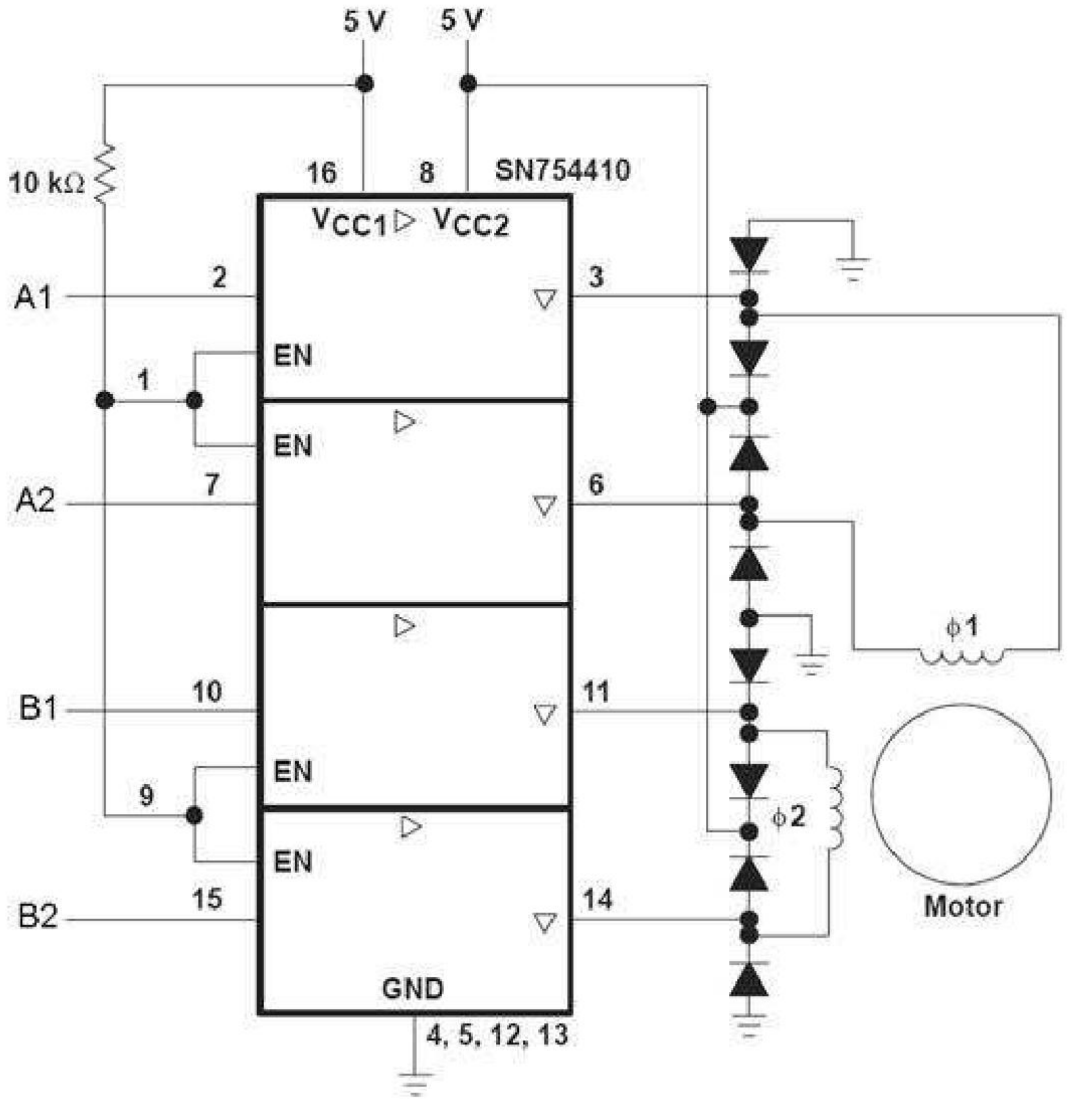


Figure 3: Motor Driver Interface Circuit

2.2 Register File

The register file consists of four 8-bit registers, labelled R0 through R3. Registers R0 and R1 are general-purpose registers, meaning that they can be used for general computation, while registers R2 and R3 are special purpose.

Register R2 stores the position of the stepper motor. The position is an 8-bit signed number, and it contains the number of half steps the motor has moved since the last time the processor has been reset. The behaviour of the position register is best understood through the following simple rules.

1. When the processor is reset, the position register contains 0. This is the starting position.

2. When the processor moves in the clockwise direction, the value of the position register is increased by an appropriate value, as follows. For each half-step the motor makes, the value in the position

register is increased by 1. For each full step the motor makes the value in the position register is increased by 2.

3. When the processor moves in the counter-clockwise direction, the value of the position register is decreased by an appropriate value, as follows. For each half-step the motor makes, the value in the position register is decreased by 1. For each full-step the motor makes the value in the position register is decreased by 2.

Register R3 contains a value that determines the delay between individual steps of the stepper motor. The instructions controlling the motor will automatically pause between individual steps for an amount of time specified in this register. The time is specified in hundredths of a second, and the same register is used for full- and half-stepping.

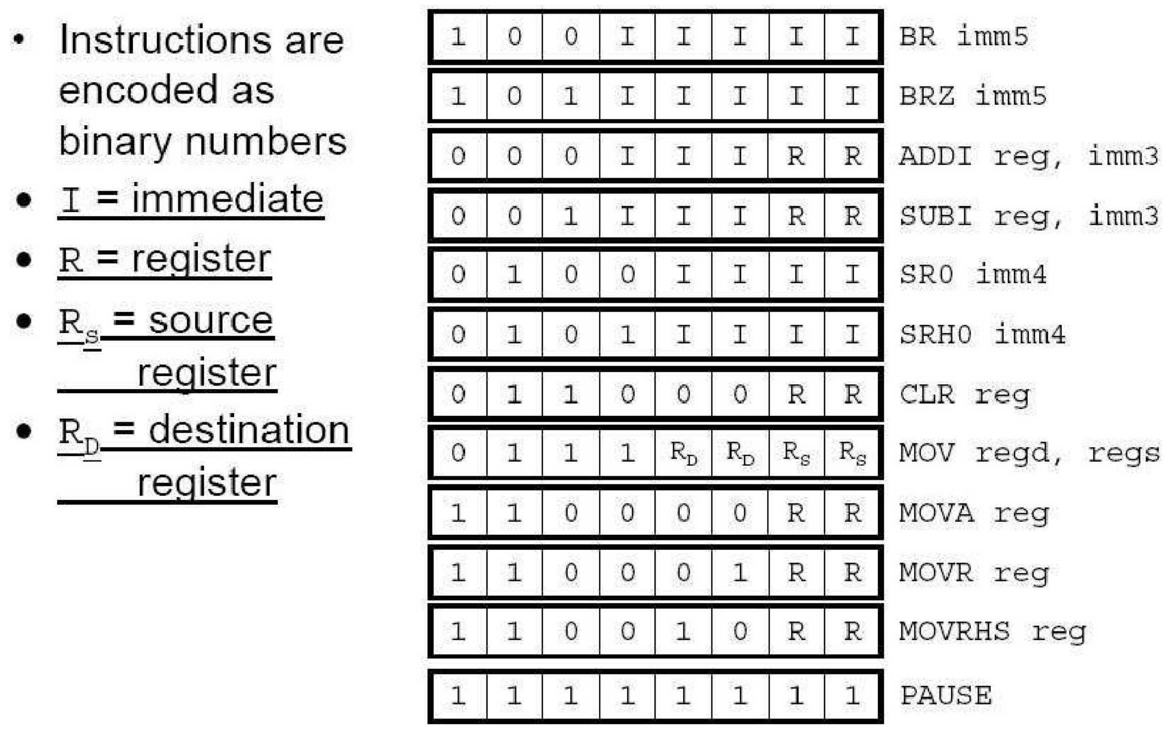
2.3 Instruction Set

- MOVR reg
  - Move the motor by a number of full steps specified in the register
  - The number in the register is signed, so the motor can move in either direction
  - After each step, wait for the amount of time specified in the delay register before proceeding
- MOVRHS reg
  - Move the motor by a number of half-steps specified in the register
  - All other requirements are the same as for MOVR
- PAUSE
  - Wait for the amount of time specified by the delay register
- CLR reg
  - Clear (i.e., set to 0) the specified register
- ADDI reg, imm3
  - Add the 3-bit unsigned immediate operand to the specified register; store the result into the register
- SUBI reg, imm3
  - Subtract the 3-bit unsigned immediate operand from the specified register; store the result into the register
- SR0 imm4
  - Set (fill) the 4 lower bits of register 0 with the 4-bit immediate operand
- SRH0 imm4
  - Set (fill) the 4 higher bits of register 0 with the 4-bit immediate operand

- MOV regd, regs
  - Move the content of the regs (source) register into the regd (destination) register
- BR imm5
  - Branch (unconditionally) to PC+imm5, where PC is the current program counter and imm5 is a signed 5-bit immediate value
- BRZ imm5
  - If register R0 is zero, branch to PC+imm5, where PC is current program counter and imm5 is a signed 5-bit immediate value

2.4 Instruction Encoding

The assembly mnemonics used in the above instruction descriptions are used to make it easier for humans to understand the operations actually performed by the processor. In reality, the instructions are actually encoded as binary numbers. The encoding of our instruction set is shown in Figure 4. Letters "I" in this figure refers to bits denoting an immediate constant. "R" refers to bits denoting the register operand to be used by the instruction. The "MOV" instruction uses two registers, and hence uses labels "R_S" and "R_D" for the bits of the source and destination registers, respectively. The figure also contains encoding for the MOVA instruction.



- Instructions are encoded as binary numbers
- I = immediate
- R = register
- R_S = source register
- R_D = destination register

1	0	0	I	I	I	I	I	BR imm5
1	0	1	I	I	I	I	I	BRZ imm5
0	0	0	I	I	I	R	R	ADDI reg, imm3
0	0	1	I	I	I	R	R	SUBI reg, imm3
0	1	0	0	I	I	I	I	SR0 imm4
0	1	0	1	I	I	I	I	SRH0 imm4
0	1	1	0	0	0	R	R	CLR reg
0	1	1	1	R_D	R_D	R_S	R_S	MOV regd, regs
1	1	0	0	0	0	R	R	MOVA reg
1	1	0	0	0	1	R	R	MOVR reg
1	1	0	0	1	0	R	R	MOVRHS reg
1	1	1	1	1	1	1	1	PAUSE

Figure 4: Instruction Encoding for the ASIP

2.5 System Behaviour

The behaviour of the ASIP you are building is specified by the FSMD diagram depicted in Appendix A. As discussed in class, each instruction has a dedicated state for execution, while the MOVR, MOVRHS, and PAUSE instructions have more than one dedicated state, because their operation is more complex.

Based on this FSMD, you will build a datapath and the control FSM that implements the functionality specified by the FSMD. The datapath that implements this FSMD is depicted in Appendix B. In this figure, ports of the various datapath elements are denoted in **blue**, control signals are denoted in **dark red**, and status signals are denoted in **dark green**. Control signals will be driven by the control FSM that you will build. Status signals will be forwarded to the same control FSM, so that it can make appropriate decisions. Finally, the **orange** text on top of the diagram denotes three basic stages in the execution of any instruction.

The elements of the datapath are briefly described below.

- Program Counter(PC)

  PC is a register that contains the program counter. It contains the memory address of the instruction currently being executed. The PC can be incremented by asserting the *increment_pc* control signal. The PC can also be loaded with a new value (on branch) by asserting the *commit_branch* control signal.

- Instruction Memory

  Contains the program to be executed by the processor. The program is specified as a sequence of machine instructions. First create a new file of type "Memory Initialization File (mif)" using the "`File`" menu of Quartus Prime and save it as "`instruction_rom.mif`" and add it to the project. Next you should create this memory module using `Tools | IP Catalog`, and name it "`instruction_rom`", because that is the name used in the existing code. This is a 256 x 8 synchronous memory implemented on-chip. The module type is "ROM: 1-PORT" (under the category "`Library > Basic Functions > On Chip Memory`"). If by default there is a check to register the output port "q", uncheck it. Click "Yes, use this file for memory content data" and specify "`instruction_rom.mif`" as the file name. Also uncheck any default checks to generate files other than "`instruction_rom.v`". After you create this module, make sure to add it to the project (`Project | Add/Remove Files in Project`).

- Decoder

  Decoder is a combinational logic block that analyzes the instruction currently being output by the instruction memory and asserts one of the signals on its output, depending on the instruction actually being read from memory.

- Register File

  The register file contains four registers. R0 through R3. Register file constantly outputs the values in the registers R0, R2 and R3. (means this module has output ports for register R0, R2 and R3) In addition, the register file contains two outputs: `selected0` and `selected1`. Each of these can output the value stored in any of the four registers. Which register is actually output depends on the 2-bit select signal: `select0` for output `selected0` and `select1` for output `selected1`. Data on the data input can be written to any of the four registers when the `write_reg_file` control signal is asserted. Which register gets written depends on the value of the `write_select` input.

- Write Address Select

  This is a multiplexer that selects one of the four possible register addresses to be written. The register to be written can be either register 0 (i.e., R0), which is useful for the SR0 and SRH0 instructions; register 2, which is useful for the MOVR and MOVRHS instructions; or it can be one of the registers specified by the instruction itself. The last two choices are labelled as `reg_field0` and `reg_field1`. `reg_field0` refers to the lowest two bits of the instruction (i.e., bits 1 and 0), while `reg_field1` refers to the next two bits (i.e., bits 3 and 2). `reg_field0` is used for writing results of instructions with only one operand, while `reg_field1` is used for writing results of the MOV instruction. The behaviour of this multiplexer is controlled by the `select_write_address` 2-bit control signal.

- Result Mux

  This is a multiplexer that selects one of the two possible values to be written to the register file. The value is either a constant 0 (for the CLR instruction), or an output of the ALU, which usually drives the result of a computation performed by the current instruction. The behaviour of this multiplexer is controlled by the `result_mux_select` control signal.

- Immediate Extractor

  This is a combinational logic block that extracts appropriate bits from the instruction to form an immediate operand, and extends the operand to 8 bits. Its behaviour can be summarized as follows:

  - For instructions that use a 3-bit immediate operand, this module extracts bits 4 through 2 of the instruction and pads the remaining bits with zeros to form an 8-bit output.
  - For instructions that use a 4-bit immediate operand, this module extracts bits 3 through 0 of the instruction and pads the remaining bits with zeros to form an 8-bit output.
  - For instructions that use a 5-bit immediate operand, this module extracts bits 4 through 0 of the instruction and sign-extends the number (i.e., pads the remaining bits with replicas of bit 4) to form an 8-bit output.
  - Finally, for the MOV instruction, the number output by this module is 0.

  One way to understand what this module does is to look at it as a multiplexer and bit extractor. Which of the four numbers that is output is determined by the `select_immediate` 2-bit control signal. This signal (just like all other control signals) is set by the control unit based on the instruction currently being executed.

- ALU

  This is an arithmetic and logic unit. It can compute one of four operations on the two 8-bit input operands. If the control signal `alu_set_low` is asserted, the ALU concatenates the highest four bits of `operanda` and the lowest four bits of `operandb` to produce an 8-bit output {`operanda[7:4], operandb[3:0]`}. If the control signal `alu_set_high` is asserted, the ALU concatenates the lowest four bits of `operandb` and the lowest four bits of `operanda` to produce an 8-bit output {`operandb[3:0], operanda[3:0]`}. These two operations implement functionality of the SR0 and SRH0 instructions. If neither of these two signals is asserted, the ALU performs addition, if the control signal `alu_add_sub` is 0, or subtraction, if this control signal is 1.

- Operand 1 Mux

  This is a multiplexer that selects one of four possible inputs for `operanda` of the ALU. Possible inputs are PC (useful for the BR and BRZ instructions), a register specified by the instruction (useful for ADDI, SUBI, and similar instructions), register R2, which contains the motor position (useful for MOVR and MOVRHS), or register R0 (useful for SR0 and SRH0). The behaviour of this multiplexer is controlled by the `op1_mux_select` control signal.

- Operand 2 Mux

  This is a multiplexer that selects one of four possible inputs for `operandb` of the ALU. Possible inputs include a register specified by the instruction (useful for the MOVA instruction), immediate operand (useful for ADDI, SUBI, and other instructions using an immediate operand), constant value 1 (useful for MOVRHS), or constant value 2 (useful for MOVR). The behaviour of this multiplexer is controlled by the `op2_mux_select` control signal.

- Stepper ROM

  This is an 8 x 4 synchronous memory implemented on-chip, which contains the sequence of signals that need to be driven to move the stepper motor. The binary numbers stored in this ROM are the 8 binary combinations required to drive the stepper using half-stepping, as discussed in lecture. You should create this module using the IP Catalog tool, and name it "`stepper_rom`", because that is the name used in the existing code. The module type is "ROM: 1-PORT" (under the category `Library>Basic Functions>On Chip Memory`). After you create this module, make sure to add it to the project (`Project | Add/Remove Files in Project`). Also make sure to create an initialization file `stepper_rom.mif`.

- Branch Logic

  This is a combinational logic block that outputs 1 when the register R0 equals 0, and outputs 0 otherwise. This block facilitates implementation of the BRZ instruction. Its output is forwarded to the control unit, which can set other control signals accordingly.

- Delay Counter

  This is probably the most complicated module in the system. It implements the functionality to delay the next step of the motor for an amount of time specified in hundredths of a second. The module behaves as follows.

  When the `start_delay_counter` control signal is asserted, the module loads (i.e., reads) the 8-bit value on its input and stores it into an internal register. This number indicates the time period that the delay counter should measure, in hundredths of a second. This loading has to be performed every time before the delay counter is enabled (see below).

  When the `enable_delay_counter` control signal is asserted, the module measures the time period that was loaded in the previous step. Once the specified amount of time has expired, the module asserts the `delay_done` signal, which is forwarded to the control unit. The control unit can take appropriate steps accordingly. The `delay_done` signal should be asserted only for as long as the `enable_delay_counter` control signal is asserted.

  There are many ways a delay counter like this could be implemented. Probably the simplest way is to design two counters. One of them measures the period of 1/100 of a second. The other counter can initially be loaded with the delay value. Every time 1/100 of a second expires, this other counter can be decremented. Once the counter reaches 0, the desired amount of time has expired and the `delay_done` signal can be asserted.

To measure the time period of 1/100 of a second, a counter has to be designed that counts to a certain value. The exact value is based on the clock speed, which is 50MHz. You should make this value parameterizable. This will help you debug the module in case it does not operate properly. If there is a problem with your circuit, simulating your design for long periods of time would take too long (keep in mind that the Quartus simulator takes a few seconds to simulate a few nanoseconds of circuit behaviour). Instead, this parameter can be set to a lower value for the purpose of simulation. Once the problem has been identified and fixed, the parameter can be set back to its original value. More details on this module construction will be given in class.

- TEMP Register

  This is a simple register/counter which is used to facilitate implementation of the MOVR and MOVRHS instructions. The module behaves as follows:

  - When the `load_temp` control signal is asserted, the module loads (i.e., reads) the 8-bit value on its input and stores it into an internal register. This number indicates the number of (half-)steps to be performed.
  - When the `increment_temp` control signal is asserted, the module increments (i.e., increases by 1) the internal counter.
  - When the `decrement_temp` control signal is asserted, the module decrements the internal counter (i.e., decreases it by 1). The module also contains three status outputs `temp_is_positive`, `temp_is_negative`, and `temp_is_zero`, whose meanings should be self-evident. These signals are forwarded to the control unit for decision-making purposes.

2.6 Debugging and Testing Tips and Tricks

There are many ways a circuit can be tested and debugged. In this section, you can find some tips that may help you in this lab, and in the future.

- You should test (typically using simulation) each module as you build them. Simulate with "representative" inputs and try to cover as many input combinations as reasonably possible. "Representative" means that the stimulus you apply during simulation has to correspond to actual values of signals that the module will see on its inputs once the module becomes a part of the system.

- Check for all warnings while building individual modules. Make sure to remove all critical warnings. Preferably, you should also remove as many non-critical warnings as you can, because they can distract you from new warnings that may be critical.

- Once you believe that every module works according to specification, integrate several modules and test the resulting circuit. Then add more modules and test that circuit. Repeat the process until the circuit is complete and operational.

2.7 How to deal with bugs

When you encounter bugs, the first thing to do is to isolate the problem. It is not sufficient to know that you have a problem; you need to know where the problem is, to be able to fix it. This is, of course, easier said than done. A useful strategy is to first identify when (under which conditions) the circuit fails to perform according to specification. These conditions are the first step towards finding the source of the problem.

Once you know the conditions under which the circuit fails, try to reproduce that behaviour (either in simulation, or in the circuit), and observe various values in the circuit (using either the simulator or the Signal Tap). Of course, this requires that you understand clearly what the circuit should be doing, which requires understanding of the lecture and lab material.

Once you find a value in the circuit that is incorrect, trace the source of the signal, i.e., find the module that is driving this signal, and observe its inputs. If all of its inputs are correct, the problem is in that module. If one of the inputs is incorrect, trace its source, and repeat the procedure. To be able to effectively do this tracing procedure, you should have a block diagram of the circuit printed. Alt-Tab-ing between Quartus and a PDF viewer tends not to work well, because you cannot make notes about the things you observed, and it is slower than using paper. Since you are working in pairs, one of you can look things up on the block diagram while the other one is reading the observed values off the screen.

It is important to remember that sometimes all the modules may be implemented correctly, but there is an incorrect/missing connection between the modules. Always double-check your connections as you are tracing the problems.

2.7.1 Simplify the problem

If you have a large circuit, try to reduce it to a smaller size, because a smaller circuit is easier to manage. This may be especially effective if the circuit is parameterizable. Even if the circuit is not parameterizable, it may be a good investment of your time to try to solve a smaller version of the same problem. In the process you may realize what you have been doing wrong.

2.7.2 Debugging tools

Don't forget to use all debugging tools that stand at your disposal:

1. Simulation is probably the easiest to use. Unfortunately, it is not always applicable, because it takes a long time to simulate realistic intervals (i.e., anything longer than 1ms). However, it may be possible to reduce simulation time by changing something in the circuit. For instance, our ASIP contains a delay circuit that implements a relatively long delay. The delay can be shortened for simulation purposes; if you find that the circuit works for a short delay, it is likely (although not guaranteed) that it also works for the larger delay. An example when the smaller delay might work, but the larger would not is if you did not provide a register that is large enough to contain the full value of the (large) delay that you are trying to implement.

   Another potential problem with simulation may occur because Quartus optimizes your circuit, so you may not be able to observe the value of some part of the circuit that was optimized. There are two solutions to this problem:

   (a) Make the signal of interest be an output of the top-level module. Quartus never optimizes outputs of modules. To make a signal an output of the top-level module, you may have to add wires at several levels of hierarchy, which may not always be practical.

(b) A better choice is to instruct Quartus not to optimize the part of the circuit you are interested in. For an example of how this can be done, look at the `datapath.v` file provided in the starter kit. Look for /*synthesis keep*/ (for wires) or /*synthesis preserve*/ (for regs) and accompanying explanation. This approach is limited to wires declared in your circuit. **However, note that for Quartus 17.1 and for the simulator Modelsim-Altera, /*synthesis preserve*/ and /*synthesis noprune*/ do no work for simulting a reg type variable which may otherwise be optimized out.** Therefore, the safest way to simuate a reg is to make the signal of interest be an output of the top-level module.

2. If the simulation is too slow, you should implement your circuit in the FPGA and use the Signal Tap Logic Analyzer. The analyzer can be configured to detect certain conditions using triggering. You can then observe values of various signals around the time of the event of interest. When adding nodes to be observed in Signal Tap, it is important to select the filter "Signal Tap:pre-synthesis", which gives you the best choice of signals that will be observable, and which will not be optimized. Signal Tap also has a limitation that it can only display a limited number of "samples" (i.e., values of signals of interest). One can circumvent these limitations by configuring and changing the triggering conditions in a clever way.

3. Sometimes it may be useful to implement the circuit in the FPGA, and map inputs and outputs to switches and lights, respectively. It is then possible to control the circuit and observe its outputs. Seven-segment displays may be used to display values of registers in the system. Since the number of seven-segment displays is limited, you could add a multiplexer to your circuit that could choose to display one of many registers in the circuit. If such a multiplexer is controlled by switches, you could manually choose which register to display, and you could thus observe values of many registers on a limited number of displays. A problem with this method is that designs usually run at high speeds (e.g. 50MHz), so it is not possible to make any meaningful observations on LEDs and displays. This can be circumvented by temporarily connecting the clock signal to one of the push-buttons on the board, like you did in the last tutorial. Each key press then represents one clock edge, so the circuit can be run slowly, step by step, and the outputs may be observed. This, of course, limits you to a small number of steps (i.e., clock cycles) that you can keep track of.

As you might have observed, each of the above methods has its advantages and drawbacks. Together, they form a powerful alliance, and virtually all problems can be solved by using a combination of these approaches,. One thing to warn you against is the following:

```
When a bug is encountered, do NOT go through your code, changing random things, hoping
that it will start working.  This method has a very low chance of success, and should
only be applied if you are confident that you know where the problem is.  If your first
attempt at this does not succeed, resort to the procedure outlined above to identify
the problem.  Also, the TAs are more than happy to assist you!
```

2.8 Updating memories

In this lab you will be using on-chip memories, (removed "modules", since students may be confused with the Verilog "modules") whose contents will be specified using `.mif` files. Every time you update one of these `.mif` files, a new programming file has to be generated. This can be done by recompiling the whole project, but there is also a faster way, by using the following three steps:

1. Select `Processing | Update Memory Initialization File`
2. Select `Processing | Start | Start Assembler`
3. Run the programmer and program the FPGA device

You will typically update the `instruction_rom.mif` to change the program the processor should execute.

Keep in mind that the above procedure only updates the memories in the system. If you made any changes to any Verilog files, you have to recompile the design.

2.9 Test Program

The final test of your processor will be performed using the following test program (depending upon available time in the lab). The program turns the motor 10 turns clockwise, pauses for 8/100s of a second, turns the motor 10 turns counter-clockwise, pauses for 8/100s of a second, and finally turns the motor half a turn counter-clockwise, but this time using half-stepping. The program code is provided below:

```
CLR R0  # R0 <-  0
SR0 1  # R0 <-  1
MOV R3,R0 # R3 <-  1 (define delay to be 1/100 of a second)
SR0 0  # R0 <-  0
SRH0 3  # R0 <-  30(hex)=48(dec)
MOV R1,R0 # R1 <-  30(hex)=48(dec)
CLR R0  # R0 <-  0 (set-up loop iterator)
SR0 10  # R0 <-  10(decimal) - (loop repeats 10 times)
MOVR R1  # Move the motor 48 steps (1 full turn) clockwise
SUBI R0,1 # R0 <-  R0-1
BRZ 2  # Branch 2 instructions forward if R0 has reached 0
BR -3  # Branch 3 locations backwards
PAUSE  # pause 1/100 of a second
...  # 8 times for a total of 8/100 of a second
SR0 0  # R0 <-  0
SRH0 13  # R0 <-  D0(hex)=-48(dec)
MOV R1,R0 # R1 <-  D0(hex)=-48(dec)
CLR R0  # R0 <-  0
SR0 10  # R0 <-  10(decimal)
MOVR R1  # Move the motor 48 steps (1 full turn) counter-clockwise
SUBI R0,1 # R0 <-  R0-1
BRZ 2  # Branch 2 instructions forward if R0 has reached 0
BR -3  # Branch 3 locations backwards
PAUSE  # pause 1/100 of a second
...  # 8 times for a total of 8/100 of a second
MOVRHS R1 # Move the motor 48 half-steps (1/2 a turn) counter-clockwise
BR 0  # infinite loop (branches to PC+0, i.e., itself)
```

3 Activities

3.1 Pre-lab [30]

The following activities must be completed and submitted as part 1 of the lab report in class at the time given at the start of this handout.

1. Download the starter kit for lab 5 and open it in Quartus Prime. When you open it, the top-level module should be `datapath.v`. Open the `datapath.v` file. In the file you will find declarations of all of the modules that comprise the datapath, but they will NOT be connected.

2. Label all wires that are not already labelled in the block diagram in Appendix B of this lab handout. Bring this labelled diagram with you to the lab (you do not have to submit this labelled diagram electronically).

3. Declare all the wires in the `datapath.v` file. Pay special attention to the widths of wires that have more than 1 bit. Use the wires you have just declared to interconnect the modules into a system as depicted in Appendix B. Each partner should do this independently.

4. Each of the modules is specified in its own file. Open each of the files, and implement each of the modules using techniques discussed in class. Each partner should do this independently.

5. Simulate each of the modules individually by applying representative inputs. Use the following guidelines for doing this:

   - You do not have to simulate the Instruction Memory, Stepper ROM or the four multiplexer modules.
   - Split the remaining eight modules into two equal groups of four between you and your partner. Each of you should simulate only half the modules to reduce the time required to complete this preparation. However, each of you should write Verilog code for ALL modules.
   - Each simulation should have at least four (preferably more) different combinations of inputs to test behaviour. For modules that require a clock signal, you should use a clock of 50MHz.
   - If any simulation uncovers a problem, fix the corresponding module and repeat the simulation until the problem disappears.
   - Take screenshots for EACH of the four simulations, and submit them as one component of preparation. To be able to simulate individual modules, you will have to make each module (one by one) a top-level module, generate functional simulation netlists, and create new waveforms for stimulus.

6. Compile the system with `datapath.v` module as a top-level module, and ensure that there are no errors or critical warnings.
   Complete the following tasks after your tutorial sessions but before the lab sessions. (You do not need to submit any work relating to these tasks with your pre-lab reports, however past history has shown that the lab is only feasible to complete if you have the machine code ready to be used in the lab.)

7. Write an assembly-level program that puts the number consisting of the last two digits of your student number into register R1, then adds 5 to this number and moves the result into register R0. Keep in mind that your student number is a decimal number, so take care to convert it into binary correctly!

8. Convert the above program into machine code. You will find this useful for testing basic functionality of your processor.

9. Write an assembly level program that spins the motor three steps clockwise, then moves it four half-steps counter-clockwise. The delay between the steps should be 1/2 of a second. The program should repeat these movements forever.

10. Convert the above program into machine code. You will find this useful for testing advanced functionality of your processor.

11. In class you have seen how control signals have to be set in different states of the FSMD to implement several different instructions. Determine the values of the control signals to implement the remaining instructions.

12. Based on the values determined in the previous step and in class, describe an FSM that implements the control circuit for your processor in Verilog. Compile it and simulate it to ensure that it produces correct signals for at least three different instructions. Save the results of these simulations by taking screenshots, and submit them as part of your preparation. You should also submit your Verilog code for your FSM.

3.2 In the tutorial session [30]

1. Open the project you worked on for preparation. You should first merge your and your partner's systems. Since each of you has verified half the modules using simulation, you should create a system that uses all of the verified modules.

2. Work on simulation of modules as described in item 5 of the pre-lab part. Show the results of your simulation to one of the TAs.

3. Open the project you worked on during the tutorial session.

4. In class we described how control signals should be set to implement three different instructions. You should compile your system with `datapath.v` being the top-level module. Then, you should create a waveform that asserts the signals to implement the three instructions described in class.

5. Simulate your datapath using the stimulus waveform derived in step 2. If the results are not as expected, find the bugs in your code and fix them. To help you simulate the instructions, the datapath module has registers for position, delay and R0 already connected using wires. The wires are declared in a manner that prevents Quartus from optimizing them, so that you can observe them in simulation. By observing the values in the registers, you can determine whether the instructions have performed the expected operations or not. For debugging purposes, you may choose to use the /*synthesis keep*/ attribute to protect other wires from being optimized.

6. Once the instructions are working as expected, demonstrate the correct behaviour to one of the TAs.

3.3 In the lab session [30]

1. Open the project you worked on for preparation. Make the lab5 module the top-level module. Compile the system.

2. Download the system to the board and verify that it performs as expected. Use both of the programs you developed for preparation to test and debug your processor. At this stage, you should use the Signal Tap Logic Analyzer for debugging. For your convenience, the outputs of the processor that drive the stepper motor are also connected to four red LEDs on the board, so that you can observe how they change. If the problem seems to be in your delay circuit, you should adjust the duration of delay by changing the delay circuit parameter, to be able to observe its behaviour using Signal Tap. This is necessary because Signal Tap can only capture a certain (finite) number of samples of each signal.

3. Once the system is working, demonstrate its behaviour to one of the TAs.

3.4 Lab Report: [10]

Since we shall be approaching end of the term, the lab report for this lab will be very simple. Just describe what you did in this lab and include the code used. **However, you should keep notes for your own record, because any of the material covered in the lab may appear on the final.**