Lab report due by 11:55PM, May 13

ENGR-UH 2310: Lab 4

Develop an 8-bit Microprocessor.

Objectives:

1. Learn about microprocessors.
2. Implement a simple microprocessor in VHDL and demonstrate it on the FPGA board. The processor has the following key features: 8-bit signed data handling, 16 operations, eight registers, and an 256-entries instruction memory.
3. Synthesize programs in binary language and execute them on the processor.
4. Demonstrate your microprocessor.

You may work in groups of two, but the two of will get the same grades then.

Also, no copy-paste between groups is allowed – we have to penalize any plagiarism with 0 points.

For this Lab 4, please don't use the Google Drive anymore to store your work-in-progress or your results – keep them private. Final submissions only to newclasses please.

Introduction

In this lab, you will implement an 8-bit processor which is capable of executing simple microprograms. This processor supports an instruction set of 16 operations. It also has eight 8-bit registers, R0 through R7.

Terminology Q&A

1. **What is a processor?**
   A processor is an integrated circuit that is capable of executing *programs*.

2. **What is a program?**
   A program is a sequence of *instructions* to accomplish a certain computational task. For example, a program can calculate the average of 10 numbers.

3. **What is an instruction?**

   An instruction is an operation, like addition. It is be performed on operands which are provided from the registers and/or from so-called immediate values. The result is to be written into the destination register which is also defined by the instruction.

   Examples:

   ADD R2, R0, R1 is an instruction that adds the content of two registers R0 and R1, and writes the result into the destination register R2.

   ADDI R2, R5, 1 *i*s an instruction that adds the content of the register R5 with immediate value of 1, and writes the result into the register R2. ADDI stands for “Add Immediate”.

   BNE R1, R2, -4 is an instruction that jumps to four instructions earlier in the program flow only if the content of register R1 is not equal to the content of register R2. Otherwise, if the contents of R1 and R2 are equal, the next instruction is executed. BNE stands for “Branch if Not Equal to”. Such an instruction is helpful, for example, to control the program flow for loops.

4. **What is binary code?**

   The binary code is the binary representation of all the instructions. In our processor, every instruction has a corresponding 16-bit binary code word. These 16 bits encode: 1) the type of the operation, also called *opcode*, 2) the operands, 3) the destination of the instruction, and, if applicable, 4) some immediate value.

   Example: *SUB R2, R1, R0* translates to the following 16-bit binary code:

   1010 010 001 000 000

   The first 4 bits represent the operation code (SUB), the next 3 bits the destination register (R2), the next 3 bits the first source register (R1), the next 3 bits the second source register (R0), and the last bits are so-called *tail bits* (not used here). All the specifics for other instructions are given further below.

5. How does the processor execute a program?

At the very beginning, the binary code of the program is to be loaded into the instruction memory. That is an initial stage which is separate from the actual program flow. Then, once the binary code is programmed, instructions are *fetched* one at a time, in sequential order. Fetching means to read and decode an instruction. A special register, called *Program Counter (PC)*, keeps track of the memory address of the next instruction to be executed. Thus, every time an instruction is executed, the PC must be updated to point to the next instruction to execute. For regular sequential instructions, the PC is simply incremented. For branch/jump instructions, the PC is updated to point to the branch/jump target. For conditional branch/jump instructions, in case the condition is not met, the PC is incremented as well.

6. What happens when the processor executes an instruction?

The operation specified by the instruction is carried out on the processor hardware. In our simple processor, the following steps are taken.

a) Instruction Fetch: The content of the instruction is read from the instruction memory and decoded, and related control signals are generated.

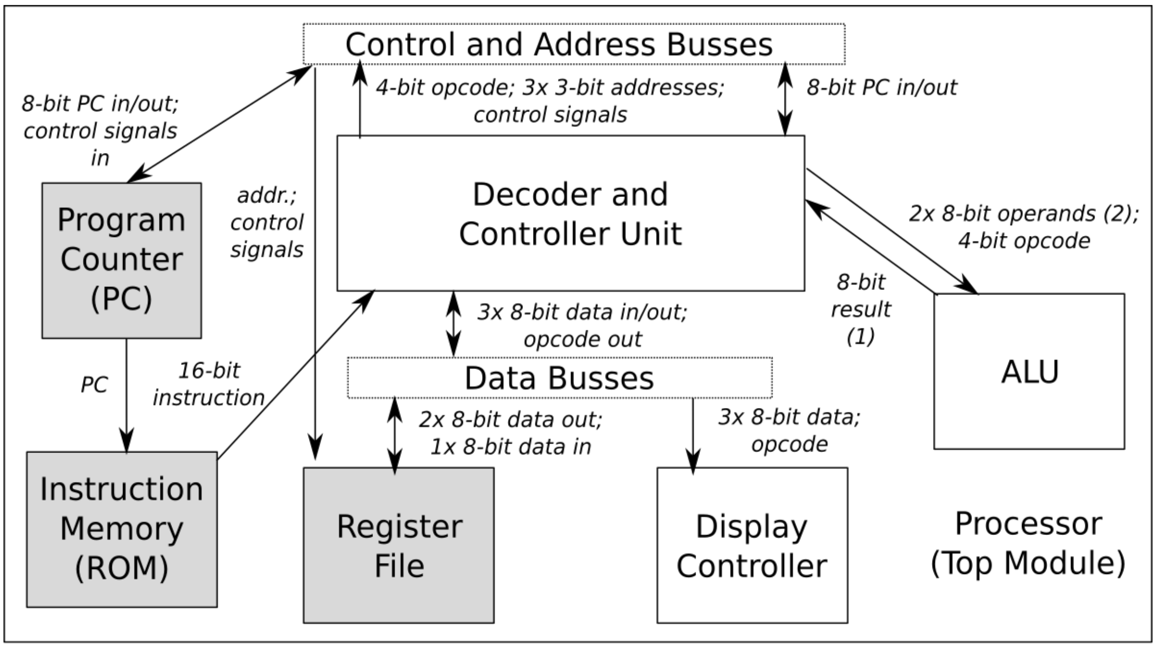
b) Execute and Write Back: For arithmetic and logic operations (add, subtract, and, or, shift), the operation is performed using the register data or immediate values, and the content of the destination register is updated with the result.

c) PC Update: The content of the Program Counter is typically incremented to point to the next instruction in the instruction memory. For instructions such as branch, jump and halt, the PC is re-written or incremented, depending on whether the conditions are met or not.

7. How does the execution of a program begin and end?

Initially the binary code of the entire program is loaded into the instruction memory, with the first instruction written into the first two bytes (16 bit). The PC is initially reset to “0” in order to point to that first instruction. Now, once the processor starts, it is executing one instruction at a time. The last instruction shall always be the *Halt* instruction which, upon execution, terminates the program by preventing an update of the PC. As a result, the processor “stalls”.

Processor Components


Control and Address Busses
8-bit PC in/out; control signals in
4-bit opcode; 3x 3-bit addresses; control signals
8-bit PC in/out
Program Counter (PC)
addr.; control signals
Decoder and Controller Unit
2x 8-bit operands (2); 4-bit opcode
8-bit result (1)
PC
16-bit instruction
3x 8-bit data in/out; opcode out
Data Busses
ALU
2x 8-bit data out; 1x 8-bit data in
3x 8-bit data; opcode
Instruction Memory (ROM)
Register File
Display Controller
Processor (Top Module)

(1) register data or new PC, depending on opcode

(2) register data, immediate value, or current PC, depending on opcode

Program counter (PC) register: This is an 8-bit register that contains the address of the next instruction to be executed by the processor. Every time a regular instruction is executed, this register is incremented so that it points to the subsequent instruction. Otherwise, for branch/jump instructions, the desired address is to be calculated by the ALU and used to update the PC, but only in case the branch conditions are met. This register can be reset to all 0’s, to point to the first entry in the instruction memory.

This component will be provided to you as “PC.vhd” via the project “Lab4_START.zip” Don’t change any of the code in there please.

Instruction memory: This memory module can hold up to 256 entries and contains the instructions to be executed. In your processor top-level module, the 8-bit output of the PC register is to be connected to the address inputs of this block, and its 16-bit data output (the instructions) is to be connected to the decoder and controller unit.

This component will be provided to you as “Instructions_ROM.vhd” via the project “Lab4_START.zip”. Both this and the PC module are already connected in your basic top-level module “top_processor.vhd” in “Lab4_START.zip”. Later on you load your binary codes by adapting the main part of this module, but leave the other parts as is please.

Decoder and controller unit: Depending on the opcode, the proper control signals and data inputs for the other components of the processor (ALU, register file, PC) are generated. Therefore, this module can be considered as the “brain” of the processor, as it orchestrates its working. This module also serves to extract the different address parts of an instruction: two addresses for the source registers, and one address for the destination register. This module should also extract the immediate value, if required.

You will have to implement this block. You may use additional logic here, but only as needed, for example for evaluating branch conditions. If you prefer, you can also implement the decoder and controller parts as separate modules.

ALU: This module can be considered as the “muscles” of the processor. It does all the computational work, such as addition, subtraction, etc. The actual work is dictated by the opcode, which is provided by the decoder and controller unit, and the data to work on is also provided by the decoder and controller unit (derived either from register contents or immediate values, depending on the opcode). The computation result is written back to some register, either a regular one or the PC special register.

You will have to implement this block. You can start with the ALU from prior labs, but here you also have to (a) make use of the custom type opcode_type, found in “common.vhd” in the project, and (b) detect but not correct overflows for signed addition and subtraction.

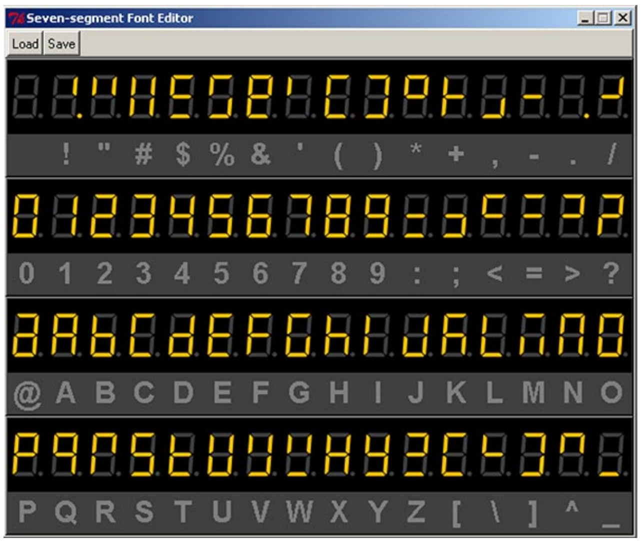
Register file: This module contains 8 registers which can hold 8-bit words each. The content of two selected data registers is simultaneously mapped onto two 8-bit output ports *Rs1_data_out* and *Rs2_data_out*. The addresses to select the registers have to be provided to the corresponding two 3-bit input ports *Rs1_addr_in* and *Rs2_addr_in*. The selected destination register is updated with the data on the 8-bit input port *Rd_data_in*, but only if the 1-bit control signal *Rd_we* is active. The address to select the register has to be provided to the 3-bit input port *Rd_addr_in*.

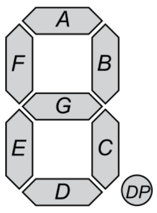
This component will be provided to you as “Registers.vhd” in “Lab4_START.zip”. Please leave the code as is. This module is already connected in the basic top-level module “top_processor.vhd” as well.

Display control unit: This module controls the seven-segment displays. It should display the operands, the opcode, and the result, one after another. Each value should be displayed for 1 s.

Negative numbers have to be indicated by “-” in front, e.g., “-128”. Detected overflows for the result have to be indicated by “OFL”.

The opcode mnemonic (abbreviation) has to be written out using alphabetic characters. You can use the guideline to the right for which segments to use for which character.


Seven-segment Font Editor
Load
Save
! " # $ % & ' ( ) * + , - . /
0 1 2 3 4 5 6 7 8 9 : ; < = > ?
@ A B C D E F G H I J K L M N O
P Q R S T U V W X Y Z [\ ] ^ _



A component “Display_Signed_BCD.vhd” will be provided to you. Please leave the code for that module as is. It works as follows: given an signed 8-bit number via “input”, it provides “sign” as std_logic output, and the number encoded as required for the seven-segment displays, already separated into hundreds, tens, and ones. Therefore, you would want to connect “sign” to the first leftmost display, and then “seg_dec_100”, “seg_dec_10”, and “seg_dec_1” to the second, third, and fourth display.

You will have to implement the actual display control unit, by making use of “Display_Signed_BCD” as component. In the display control unit, you need to handle the multiplexing of the four segments, also with alternation between operands, results (possibly as “OFL”), and the opcode characters.

Top-level modules: You will have to implement two top-level modules, in that order:

1. Basic version: This version connects all the processor modules mentioned above. A starting point is already provided to you via the “top_processor.vhd” module, but you have to extend it to connect all the modules of the processor.

   It takes the clock and reset signals as primary inputs, and provided the two current operands, the corresponding ALU result, as well as the current opcode as outputs.

   This version is to test your processor via simulation, based on some instructions given in the instruction memory. Use the provided “tb_top_processor.vhd” for starting simulations on this basic version.

2. FPGA version: This version is to run the processor on the FPGA along with the seven-segment displays.

   The ports are to be revised as follows: the master clock, a custom clock, and the reset signals are inputs, and the display signals are outputs.

   The master clock from the FPGA board is used for the display and its clock divider, whereas the custom clock is used for the processor execution and is mapped to an button – whenever you push that button, the processor executes one instruction. See also Task 1.

Processor Instructions

Every instruction has 16 bits that define the type of the instruction (opcode) as well as the operands and the destination of the result.

The instruction format is:

Opcode (4 bits)	Rd (3 bits)	Rs1 (3 bits)	Rs2 (3 bits)	Tail (3 bits)

The *opcode* defines the operation, *Rd* denotes the address of the destination register (into which the result of the operation will be written, unless otherwise specified), and *Rs1* and *Rs2* denote the addresses of the two source registers (which contain the two operands, unless otherwise specified). The *tail* contains additional data but only for some of the instructions.

The instruction set which has to be supported by your processor is specified below. All operations are to be performed on two-complement encoding for 8-bit signed numbers unless otherwise specified.

In case an immediate value (IV) is used, the register address bits are leveraged as actual data bits; they are not representing an address here, but they are readily part of the IV. The descriptions below specify how to derive the IV, if any, which varies for the different operations.

Opcode	Mnemonic	Description	Operation
0000	AND	Bitwise And	Rd = Rs1 and Rs2
0001	ANDI	Bitwise And Immediate	Rd = Rs1 and IV; IV derived from the address Rs2 and tail, in that order, and unsigned extended into 8 bits, by padding “11” to the left
0010	OR	Bitwise OR	Rd = Rs1 or Rs2
0011	ORI	Bitwise OR Immediate	Rd = Rs1 or IV; IV derived from the address Rs2 and tail, in that order, and unsigned extended into 8 bits, by padding “00” to the left
0100	SLL	Shift Left Logical	Rd = shift_left(Rs1, IV); IV derived from the tail and unsigned extended into 8 bits, by padding “00000” to the left
0101	SRL	Shift Right Logical	Rd = shift_right(Rs1, IV); IV derived from the tail and unsigned extended into 8 bits, by padding “00000” to the left
0110			*Bonus – to be defined by you*
0111	HLT	Halt	PC = PC; stalling processor in an infinite loop
1000	ADD	Add	Rd = Rs1 + Rs2
1001	ADDI	Add Immediate	Rd = Rs1 + IV; IV derived from the address Rs2 and tail, in that order, and sign-extended into 8 bits, by padding two sign bits to the left

1010	SUB	Subtract	Rd = Rs1 - Rs2
1011	SUBI	Subtract Immediate	Rd = Rs1 – IV; IV derived from the address Rs2 and tail, in that order, and sign-extended into 8 bits, by padding two sign bits to the left
1100	BLT	Branch If Less Than	If (Rs1 < Rs2) then PC = PC + IV, else PC = PC + 1; IV derived from the address Rd and tail, in that order, and sign-extended into 8 bits, by padding two sign bits to the left
1101	BE	Branch If Equal To	If (Rs1 = Rs2) then PC = PC + IV, else PC = PC + 1; IV derived from the address Rd and tail, in that order, and sign-extended into 8 bits, by padding two sign bits to the left
1110	BNE	Branch If Not Equal To	If (Rs1 /= Rs2) then PC = PC + IV, else PC = PC + 1; IV derived from the address Rd and tail, in that order, and sign-extended into 8 bits, by padding two sign bits to the left
1111	JMP	Jump	PC = PC + IV; IV derived from the address Rd and tail, in that order, and sign-extended into 8 bits, by padding two sign bits to the left

The jump and the branch instructions are to realize “loops” in your microprogram. This is accomplished by purposefully updating the PC depending on the loop conditions (otherwise, the PC is simply incremented, as with regular operations).

Since the register file has eight registers, 3 bits are used to address them as follows:

Rd/Rs1/Rs2	Register Selected
000	R0 (hard-wired word “0”)
001	R1
010	R2
011	R3
100	R4
101	R5
110	R6
111	R7

Note that R0 contains the constant value “0”, and also note that you cannot overwrite this special register.

Tasks and Description

Make sure to start from the provided project ZIP file. For submission, provide your whole project folder as ZIP file as well. Deliverables and important points are also highlighted in red for the task instructions given below.

TASK 1: Implement the processor in VHDL.

Instructions and deliverables:

1) Implement the missing VHDL modules decoder unit, control unit, and ALU. Make sure to include "use work.common.all;" in the VHDL headers. You may combine the decoder and the controller in case you deem it more practical. You may re-use the ALU from prior labs, but revise it to work on the custom type opcode_type and to detect (but not correct) overflows.

Make sure to leave the code for the provided modules "Display_Signed_BCD", "Registers", "PC", and "Instructions_ROM" as is.

2) Extend the basic top-level module, without the seven-segment display. You should start with the one provided to you. Use proper component instantiation; you are **not** allowed to implement any computation/processes directly in the top-level module (except for the tracing-mode bonus) but only within the modules.

Report how many clock cycles your implementation requires to fully execute one instruction, from loading the instruction to decoding it, to computing the result, and to writing back the result. Look into behavioral simulation runs to answer that.

4) Implement the FPGA top-level module. Implement the missing module display control unit for that. This module has to show signed numbers for the operands and result, show "OFL" in case the result overflows, and show the opcode mnemonic as characters. The display should transition every 1s between operand 1, operand 2, opcode, and result.

Make sure to provide the master clock "clk" only for the display controller and its clock divider(s), whereas a custom clock "clk_proc_in" is used for the processor execution and is mapped to an FPGA button. Use the related, already provided UCF.

You may also tackle this part only one you have started the behavioral simulations on the basic top-level module asked for in the other tasks, as this helps you to identify any issues with the working of your processor in general, without bothering about any possible issues with the FPGA implementation yet.

Report the resources utilized, and the warnings (if any) occurred during synthesis. Also report the maximum frequency and levels of logic of your implementation. Look into the design summary for that information. Also print the floorplan as PDF.

Points: 15.0

4.0 for VHDL code for decoder, controller unit

1.5 for VHDL code for ALU unit

1.5 for VHDL code basic top-level module

0.5 for report on clock cycles per instruction

1.0 for VHDL code FPGA top-level module

5.0 for VHDL code for display control unit

1.0 for report on resources, warnings, frequency, logic levels

0.5 for floorplan PDF

TASK 2: Test your processor by executing the following simple program:

```
ADDI R1, R0, -32        // R1 = R0 + -32 = -32
ADDI R2, R0, -32        // R2 = R0 + -32 = -32
ADD R3, R1, R2          // R3 = R1 + R2 = -64
ADD R3, R3, R3          // R3 = R3 + R3 = -128
ADDI R4, R3, -1         // R4 = R3 + -1 = 127; overflow, should be -129
```

Note that the binary code for the instruction sequence above is already pre-loaded as binary code in the instruction ROM.

Instructions and deliverables:

1) Provide snapshots for both the behavioral simulation of the basic top-level module and the post-route simulation of the FPGA top-level module.

a) For the behavioral simulation:
   - Use the provided testbench to start the behavioral simulation.
   - Make sure to show all signals of your top-level module. To do so, first unfold the testbench top-module in ISIM, and then drag & drop the "uut" to the waveform window.

b) For the post-route simulation:
   - Create a new testbench for the FPGA top-level module. Make sure to set the clock period in the testbench to some proper value which is above the minimum period / below the maximum frequency – otherwise your simulation results will be off. For that, look into the console output of the "Synthesize" step. There, toward the end of the output, look for a line like this: "Minimum period: 14.012ns (Maximum Frequency: 71.368MHz)". In this example, you would set the clock period to 15 ns in the testbench.
   - Make sure that you include the following signals as primary outputs: the two operands, the result, and the overflow. You don't have to show other internal signals, but you may if you want to.
   - In a separate zoomed-in snapshot, provide markers showing how long it takes the FPGA to settle the result for one computation step of your choice. To do so, use the rising edge of the clock as start, and the final transition for the result signal as stop.
   - When you start the simulation from ISE, make sure to select "Post-Route" from the simulation view. This has to reflect in the simulation option which you start, there it should read now "Simulate Post-Place & Route Model".

2) Provide a short video with the program running on the FPGA.

Points: 3.5

0.5 for behavioral simulation snapshot

0.5 for post-route testbench

1.5 for post-route simulation snapshots

1.0 for video

TASK 3: Evaluate the following binary program using your processor:

```
1001 001 000 000 111
1001 010 000 000 000
1001 010 010 000 011
1011 001 001 000 001
1110 111 001 000 110
```

Instructions and deliverables:

1) Describe the program behavior and the final result(s). To do so, you may either run a behavioral simluation, run the program on the FPGA board, or you may even "run" the program manually using pen and paper. Provide the program behavior either as simulation snapshot or as descriptive program flow; either way, make sure to provide all intermediate results for all related registers, all opcodes, and also the final result(s).

2) Implement the program as binary code in the instruction memory

3) Provide a short video with the program running on the FPGA.

Points: 3.0

1.5 for program behavior, intermediate results, final results

0.5 for implementation in instruction memory

1.0 for video

TASK 4: Write your own program for the microprocessor.

Instructions and deliverables:

1) Describe a task/procedure that can be run on the processor – get as creative as you can. You must use at least one jump or branch instruction, your program must be meaningful, and it must be at least 8 lines long.

2) Provide the program and the expected behaviour for meaningful inputs of your choice. You may provide pseudo-code or directly binary code; either way, make sure to properly comment it regarding the opcode, registers, immediate values, if any, and the result(s). You may also run behavioral simulations to do so.

3) Implement your program as binary code in the instruction memory.

4) Provide a short video with the program running on the FPGA.

Points: 4.0

1.0 for program, has to be meaningful and at least 8 lines long

1.5 for program behavior, intermediate results, final results

0.5 for implementation in instruction memory

1.0 for video

BONUS TASK 1: Implement a “tracing mode” to display the 8-bit signed register data on the FPGA. Toward this end, extend the top-level module.

Instructions and deliverables:

1) Implement a mode switch as follows. You may implement the related logic directly within the top-level module.

   a) For the regular mode, the operands, opcode, and result are displayed as usual.

   b) For the tracing mode, the content of one register (addressed via four switches) is to be displayed. Also, the custom clock (to advance the processor) must be disabled in the tracing mode.

2) Provide a short video with a program of your choice running on the FPGA. Switch between regular mode and tracing mode for key steps in your program. Show the relevant register data in trace mode.

Bonus points: 3.0

2.0 for implementation

1.0 for video

BONUS TASK 2: The opcode 0110 is left undefined so far. This is an opportunity for you to extend this processor.

Instructions and deliverables:

1) What instruction would you like to add to this processor? Motivate this by re-writing one of the programs you worked on (or any other program) while using your new instruction.

2) Implement the new instruction in your VHDL code. Make sure to consider all parts of the processor, from decoder/control unit, to ALU, and display control unit.

3) Is this new instruction making the processor more efficient? To answer this:

   a) Report the runtime, in terms of clock cycles, when running your considered program without and with your new instruction.

   b) Report the difference in resources utilized for your new design.

Bonus points: 4.5

1.0 for meaningful instruction and rewritten binary program
2.5 for VHDL implementation of new instruction
1.0 for report on cycles and resources