Lab report due by 11:45PM, May 11

ENGR-AD 201: Lab Assignment 4

8-bit Microprocessor

Objectives:

1. Learn about microprocessors.
2. Implement the NYUAD processor in VHDL and on the FPGA board. The processor has the following key features: a set of 16 instructions, eight 8-bit registers, 256-entries instruction memory.
3. Synthesize programs in binary language and execute them on the processor.

You may work in groups of two, whereupon you'll get the same grades.

Introduction

In this lab, you will implement an 8-bit processor, the NYUAD processor, which is capable of executing more or less simple microprograms. This processor supports an instruction set of 16 instructions, which can all operate within one clock cycle. It also has eight 8-bit registers, R0 through R7.

Terminology Q&A

1. **What is a processor?**

   A processor is a hardware component 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 input numbers.

3. **What is an instruction?**

   An instruction is an operation to be performed on a number of operand (sources from the registers and, if applicable, so-called immediate values) with the result to be written into the destination register also defined by the instruction.

   Example: *ADD R3, R1, R2* is an instruction that adds the content of two registers R1 and R2 (operands: R1 and R2), and writes the result into the destination register R3.

   Example: *ADDI R2, R5, 1* is an instruction that adds the content of the register R5 with immediate value of 1 (operands: R5 and 1), and writes the result into the register R2. ADDI stands for "Add Immediate".

   Example: *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 (operands: R1 and R2). BNE stands for "Branch if Not Equal to".

4. **How does the processor execute a program?**

   The binary code (see below) of the program is loaded into the instruction memory, whereupon the instructions are fetched (that is, read and decoded) one at a time, in sequential order. The program flow can be redirect by branch and jump instructions. 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 content of the PC must be updated to point to the next instruction to execute. For regular sequential instructions, the PC is incremented; for branch/jump instructions, the PC is updated to point to the branch/jump target.

5. **What is binary code?**

   The binary code of an instruction is the binary representation of the instruction. In our processor, every instruction has a corresponding 16-bit binary code. 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 values.

Example: *SUB R3*, *R2*, *R1* translates to the following 16-bit binary code:

0010 011 010 001 000

The first 4 bits are the operation code (SUB), the next 3 bits the destination register (R3), the next 3 bits the first source register (R2), the next 3 bits the second source register (R1), and the last bits are so-called tail bits (here not used, only for some instructions). All the encoding specifics are given further below.

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.

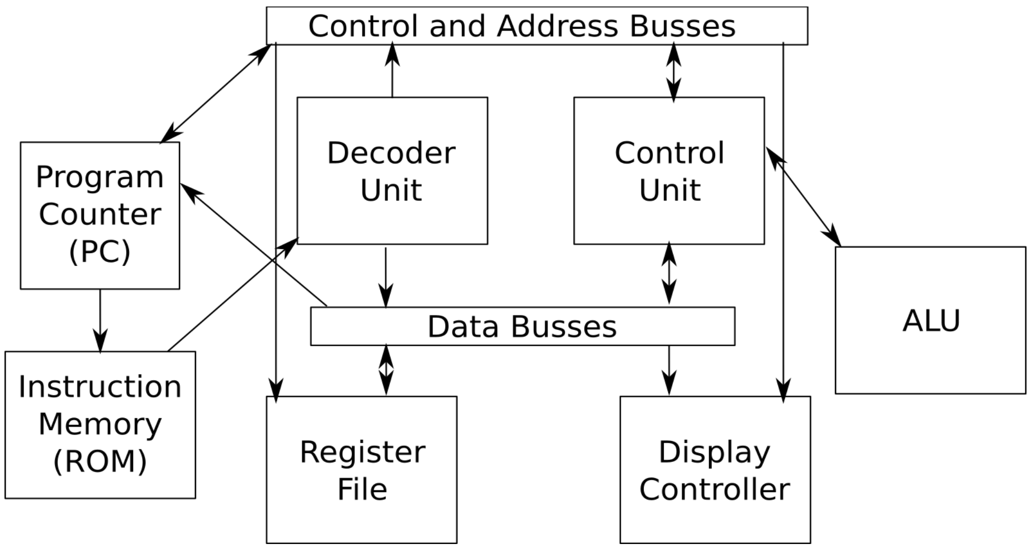
   b. PC Update: The content of the Program Counter is updated to point to the next instruction in the instruction memory; see also (c).

   c. 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. For other instructions such as branch, jump and halt, the content of the Program Counter is updated depending on the conditions.

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

   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 reset to “0” in order to point to that first instruction. Now, the processor starts, executing one instruction at a time. The last instruction shall always be the Halt instruction which, upon execution, terminates the program by preventing the update of the PC. As a result, the processor “stalls”.

Processor Components



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 calculated and then used to update the PC. This register can be reset to all 0's, thereafter pointing to the first entry in the instruction memory.

This component will be provided to you.

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

This component will be provided to you. You will load your binary code by adapting the main part of this module.

Decoder unit: This block evaluates all 16 bits of the instruction, and extracts the different parts as separate signals: the two source-register addresses, the destination-register address, the opcode, and (if applicable) the immediate value.

You will implement this block.

Control unit: This block works in conjunction with the decoder unit. Depending on the opcode, the proper control signals and inputs for the other processor blocks (ALU, register file, PC) are generated.

You will implement this block, which can be considered as the "brain" of the processor, as it orchestrates the operations. You may use VHDL operators for comparisons (instead of delegating that to the ALU). You may also combine this block with the decoder in case you deem doing so more practical.

ALU: This block can be considered as the “muscles” of the processor. It does all the hard work, such as addition, subtraction, etc. It uses the opcode generated by the Decoder/Control Unit, as well as the operands (also derived by the Decoder/Control unit, from the registers or the immediate values). It computes data that can be written into one of the registers (including PC).

You will implement this block by referring to the instruction set.

Register file: This block contains 8 registers which can hold 8 bits each.

The content of two selected data registers is mapped onto two 8-bit output ports, data_out1 and data_out2. The addresses to select the registers have to be provided to the two 3-bit input ports of the block, Addr_Rs1 and Addr_Rs2.

The selected destination register is update with the data on the 8-bit input port, data_in, but only if the 1-bit write input is active. The address to select the register has to be provided to the 3-bit input port Addr_Rd.

This component will be provided to you.

Top-level modules: You will implement two versions of a top-level module as follows.

1. Basic version: This version connects all the processor modules mentioned above. It takes a clock and reset signals as inputs, and it provides the two current operands, the corresponding ALU result, as well as the current opcode as outputs. This version is to test your processor implementation via simulation, based on the instructions you programmed into the instruction memory.
2. FPGA version: This version is to connect the processor on the FPGA with to the seven-segment displays. Therefore, the ports are revised as follows: the master clock, a custom clock, and the reset signals are inputs, whereas only the two display-controller signals are provided as 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 FPGA button (thus, whenever you push that button, the processor executes one cycle).

Seven-segment displays: Finally, for the FPGA version, you will implement and make use of all four seven-segment displays. The first two displays show the signed operands, the third display the opcode (in hex), and the last display the signed result. For the halt instruction, display “H” on all four displays.

You may reuse the display and clock-divider modules from your prior labs as starting point.

Processor Instructions

Every instruction has 16 bits that define the type of the instruction (also called opcode) as well as the operands and the destination of the result. These instructions are stored in the instruction memory from where they are fetched one at a time, and are executed by the ALU. The first instruction is stored at the memory address 0000 0000. Hence, the PC register's reset towards 0000 0000 is to point to this instruction.

The instruction format is:

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

The opcode defines the operation to be performed, Rd denotes the address of the destination register in which the result of the operation will be written, and Rs1 and Rs2 denote the addresses of the two registers that contain the two operands. The Tail field is used by some of the instructions.

The instruction set supported by the NYUAD processor is defined below. All operations are performed assuming two-complement notation for the operands and the result, unless otherwise specified.

Opcode	Mnemonic	Description	Operation
0000	ADD	Add	Rd = Rs1 + Rs2
0001	ADDI	Add Immediate	Rd = Rs1 + Value (6 bits from both Rs2 and tail, sign-extended into 8 bits; padding two sign bits to the left)
0010	SUB	Subtract	Rd = Rs1 - Rs2
0011	SUBI	Subtract Immediate	Rd = Rs1 - Value (6 bits from both Rs2 and tail, sign-extended into 8 bits; padding two sign bits to the left)
0100	AND	Bitwise And	Rd = R1 & R2 (two 8-bit numbers are bitwise ANDed)
0101	ANDI	Bitwise And Immediate	Rd = R1 & Value (6 bits from both Rs2 and tail, unsigned extended into 8 bits; padding "00" to the left)
0110	OR	Bitwise OR	Rd = R1 | R2 (two 8-bit numbers are bitwise ORed)
0111	ORI	Bitwise OR Immediate	Rd = R1 | Value (6 bits from both Rs2 and tail, unsigned extended into 8 bits; by padding "00" to the left)

1000	SLL	Shift Left Logical	Rd = Rs1 << Value (tail, unsigned extended into 8 bits; by padding “00000” to the left)
1001	SRL	Shift Right Logical	Rd = Rs1 >> Value (tail, unsigned extended into 8 bits; by padding “00000” to the left)
1010	HAL	Halt	PC = PC (“Stalling” processor in infinite loop)
1011	*Unused*		*To be defined by you (bonus)*
1100	JMP	Jump	PC = PC + Value (6 bits from both Rs2 and tail, sign-extended into 8 bits; padding two sign bits to the left)
1101	BLT	Branch If Less Than	If (Rs1 < Rs2) then PC = PC + Value (6 bits from both Rd and tail, sign-extended into 8 bits; padding two s. bits)
1110	BE	Branch If Equal To	If (Rs1 = Rs2) then PC = PC + Value (6 bits from both Rd and tail, sign-extended into 8 bits; padding two s. bits)
1111	BNE	Branch If Not Equal To	If (Rs1 /= Rs2) then PC = PC + Value (6 bits from both Rd and tail, sign-extended into 8 bits; padding two s. bits)

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 increment).

As the register file has eight registers, 3 bits are used to select the registers as follows:

Rd/Rs1/Rs2	Register Selected
000	R0 = 0000 0000
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.

Lab Tasks Description

Make sure to provide your project files as ZIP folder. Other deliverables are mentioned in the task instructions.

TASK 1: Implement the NYUAD processor in VHDL.

Instructions and deliverables:

1) Start with the individual modules, which all have to be implemented as separate VHDL modules (entities).

2) Implement the basic top-level module, without the seven-segment display.

3) Report the performance (max frequency), the resources utilized, and the warnings (if any) occurred during synthesis. Look into the design summary for that information.

Also 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 the simulation runs (see below) to derive that information.

4) Implement a simple testbench for that top-level module, which triggers the reset signal in the beginning, and then (with reset set to 0 again), the processor should run by itself according to the program.

Make use of that testbench for the remaining tasks. You may also make use of the testbench for debugging of your processor implementation. You may additionally implement testbenches for the individual modules, but they are not considered for grading.

5) Implement the FPGA version for the top-level module, with the seven-segment display. Make sure to provide the master clock only for the display and the clock divider, whereas a custom clock is used for the processor execution and is mapped to an FPGA button. Implement a UCF.

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

```
ADDI R1, R0, 7 // R1 = R0 + 7 = 7
ADDI R2, R0, 8 // R2 = R0 + 8 = 8
ADD R3, R1, R2 // R3 = R1 + R2
```

The 8-byte binary code for the instruction sequence above is already pre-loaded as binary code in the instruction ROM:

```
00010010 00000111

00010100 00001000

00000110 01010000
```

Instructions and deliverables:

1) Provide simulation snapshots. Use the testbench from Task 1 to start the simulation. Make sure to show all signals of your top-level module; to do so, in ISIM, unfold the testbench topmodule, and drag&drop the "uut" to the waveform window.

2) Provide a short video with the program running on the FPGA. You may also show the program running on the FPGA in person to the instructor.

TASK 3: Test your processor by executing the following binary program (already provided as comment in the instruction memory):

```
0001 001 000 000 101
0001 010 000 000 000
0001 010 010 000 010
0011 001 001 000 001
1111 111 001 000 110
```

Instructions and deliverables:

1) Identify the instructions and the registers/immediate values for the binary program. Provide pseudo-code for the program, that is translate the binary program to some "regular" code (the code format is your choice).

2) Outline the expected program behaviour and the final results.

3) Provide simulation snapshots. Use the testbench from Task 1 to start the simulation. Make sure to show all signals of your top-level module; to do so, in ISIM, unfold the testbench topmodule, and drag&drop the "uut" to the waveform window.

4) Provide a short video with the program running on the FPGA. You may also show the program running on the FPGA in person to the instructor.

TASK 4: Write your own program for the NYUAD processor.

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.

2) Provide the code and the expected behaviour. You may provide pseudo-code and/or directly binary code; in case you provide only binary code make sure to properly label it regarding the opcode, registers, immediate values, results.

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

4) Provide simulation snapshots. Use the testbench from Task 1 to start the simulation. Make sure to show all signals of your top-level module; to do so, in ISIM, unfold the testbench topmodule, and drag&drop the “uut” to the waveform window.

5) Provide a short video with the program running on the FPGA. You may also show the program running on the FPGA in person to the instructor.

BONUS TASK 1: Implement a “tracing mode” to display the register content on the FPGA. Towards this end, extend the top-level module.

Instructions and deliverables:

1) Implement a mode switch as follows.

   a) For the regular mode, the operands, opcode, and ALU results are displayed as usual, whereas for the tracing mode, the content of one register (addressed via switches) is displayed on all four displays.

   b) In the tracing mode, both the reset as as well as the custom clock (to advance the processor) must be disabled.

2) Provide a short video with a program of your choice running on the FPGA. Switch between regular mode and tracing mode. In tracing mode, also show that resetting or advancing the processor is disabled. You may also show the program running on the FPGA in person to the instructor.

BONUS TASK 2: Opcode 1011 is left unused intentionally. This is an opportunity for you to extend this processor.

Instructions and deliverables:

1) What instruction would you like to add to this processor? Consider an instruction that would be quite handy to have. Justify this by re-writing some program above while using your new instruction.

2) Implement the new instruction in your VHDL code.

3) Is this new instruction worth having? To answer this:

   a) Compare the performance (max frequency) of this extended version with your original evaluation from Task 1.

   b) Look into the total runtimes, obtained by simulation for both versions of a program of your choice.

   c) State the difference in resources utilized for your new design.