Lab 9-10 – Nanoprocessor Design Competition

CS1050 Computer Organization and Digital Design

Dept. of Computer Science and Engineering, University of Moratuwa

Learning Outcomes

In this lab, we will design a 4-bit processor capable of executing 4 instructions. After completing the lab, you will be able to:

- design and develop a 4-bit arithmetic unit that can add and subtract signed integers
- decode instructions to activate necessary components on the processor
- design and develop *k*-way *b*-bit multiplexers or tri-state busses
- verify their functionality via simulation and on the development board

This is a team project, and each team consists of 5 students. Therefore, you will also practice team-working skills such as communication, coordination, sharing responsibilities, and integrating components developed by different team members. You are free to select lab buddies.

Introduction

We will design a very simple microprocessor (hence, called a *nanoprocessor*) capable of executing a simple set of instructions. Block diagram of the nanoprocessor is given in Fig. 1. Set of instructions supported by the nanoprocessor is given in Table 1. To build this circuit, we need to develop/extend several components/modules:

- 4-bit Add/Subtract unit
  - This unit should be capable of adding and subtracting numbers represented using 2's complement
  - You may implement this component by modifying your 4-bit RCA from Lab 3
- 3-bit adder
  - This unit is used to increment the Program Counter
  - You may implement this component by modifying 4-bit RCA from Lab 3
- 3-bit Program Counter (PC)
  - Program counter needs to be reset to 0 when required. Hence build it using D Flip Flops with a clear/reset input. You may use the D Flip-Flop from Lab 5
- *k*-way *b*-bit multiplexers
  - A *k*-way *b*-bit multiplexer can take in *k*-inputs, each with *b*-bits, rather than a single bit, and the output is a group of *b*-bits. There are $\log_2 k$ control bits, and these control bits are used to select one of the *k* groups of *b* bits rather than a single bit.
  - Build a 2-way 3-bit multiplexer
  - Build a 2-way 4-bit multiplexer
  - Build a 8-way 4-bit multiplexer
  - You may implement the component using 8-to-1 multiplexer developed in Lab 4
  - Alternatively, instead of multiplexers, you may do the same using tri-state buffers. You need to research how to build a tri-state buffer in VHDL.

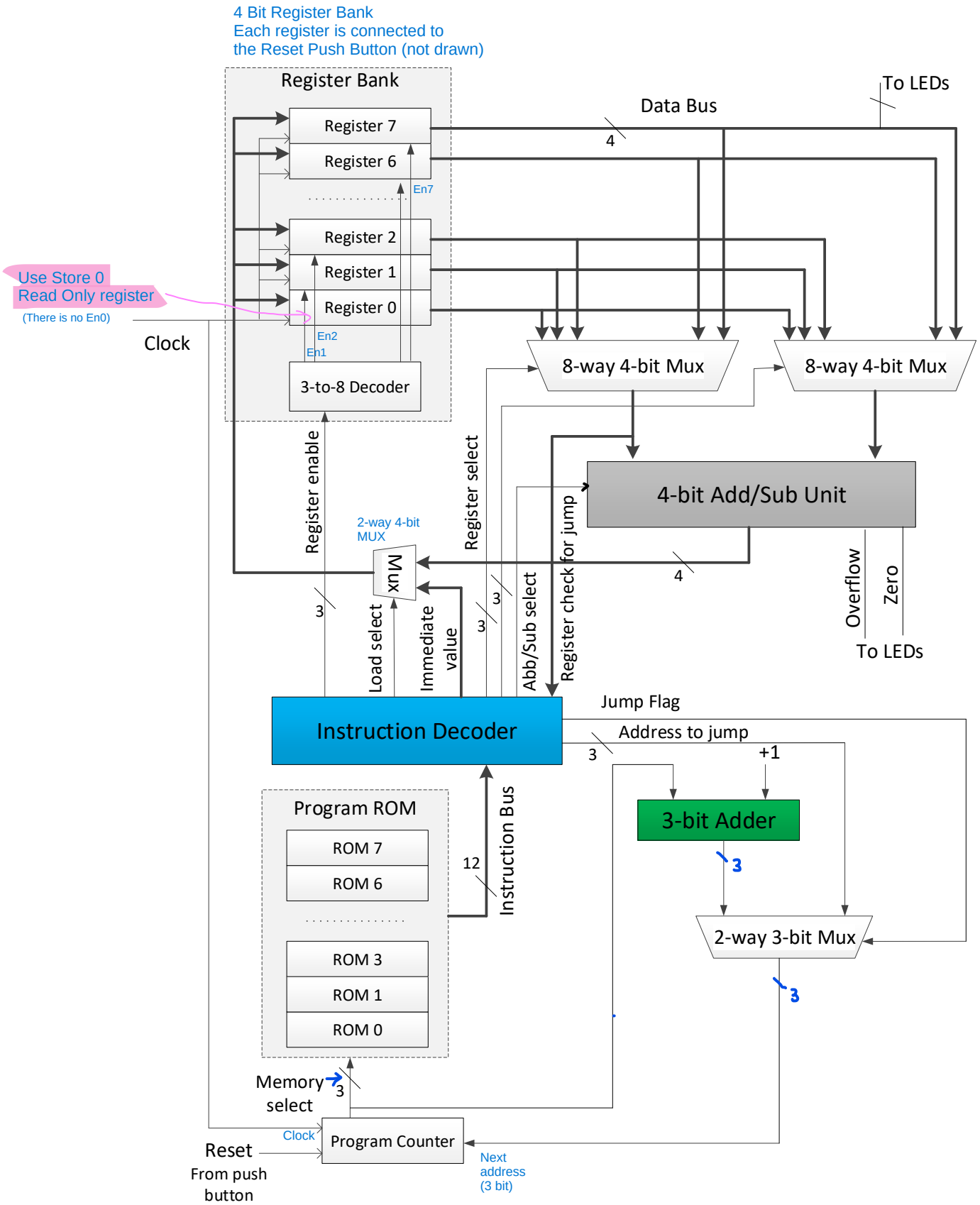

Figure 1 – High-level diagram of the nanoprocessor.

Table 1 – Instruction Set.

Instruction	Description	Format (12-bit instruction)
MOVI R, *d*	Move immediate value *d* to register R, i.e., *R* ← *d* R ∈ [0, 7], *d* ∈ [0, 15]	1 0 R R R 0 0 0 d d d d
ADD Ra, Rb	Add values in registers Ra and Rb and store the result in Ra, i.e., Ra ← Ra + Rb Ra, Rb ∈ [0, 7]	0 0 Ra Ra Ra Rb Rb Rb 0 0 0 0
NEG R	2's complement of registers R, i.e., R ← – R R ∈ [0, 7]	0 1 R R R 0 0 0 0 0 0 0 0
JZR R, d	Jump if value in register R is 0, i.e., If R == 0 PC ← d; Else PC ← PC + 1; R ∈ [0, 7], *d* ∈ [0, 7]	1 1 R R R 0 0 0 0 d d d

- Register Bank
  - Contains 8, 4-bit registers (named R0 to R7)
  - Hardcode value of R0 to all 0s
  - You may use 3-to-8 decoder developed in Lab 4
  - As we do not have a separate instruction to reset a register, we will use D Flip Flops with a reset input and connect the reset input to Reset button (not shown in Fig. 1 for simplicity). This could be connected to the same pushbutton connected to the Reset input of Program Counter (see Fig. 1)
- Program ROM
  - This stores our Assembly program
  - This can be built by extending the ROM-based LUT developed in Lab 7
- Buses
  - Use 3, 4, and 12-bit buses to connect components. This will greatly simplify your design rather than running so many wires around. Use may use labels such as **D(3 downto 0)**, **I(11 downto 0)**, **M(3 downto 0)**, and **R(3 downto 0)**.
- Instruction Decoder
  - We need to design and build the Instruction Decoder circuit to activate necessary components based on the instructions we wish to execute
  - First design the internal logic such that instructions in Table 1 will execute properly
  - Be careful to activate only the necessary modules. For example, while executing MOVI instruction only the required register should be enabled, and the immediate value needs to be placed on the data bus. For ADD and NEG instructions, relevant inputs should be selected from the multiplexers and output should be sent to the correct register (register needs to be enabled). To simplify the implementation of NEG instruction, we can hardwire register R0 to be 0 (i.e., this will be a read-only register). Moreover, NEG require setting Add/Sub select

We wish to execute an Assembly program like the following in our nanoprocessor:

```
MOVI   R1, 10       ; R1 ← 10
MOVI   R2, 1        ; R2 ← 1
NEG    R2           ; R2 ← -R2
```

```
ADD    R1, R2       ; R1 ← R1 + R2
JZR    R1, 7        ; If R1 = 0 jump to line 7
JZR    R0, 3        ; If R0 = 0 jump to line 3
```

As the microprocessor only understands machine language, we need to provide those instructions as a binary value. We will hard code our program to ROM. One of the pushbuttons should be used to reset the PC and Register Bank (this enables us to restart the program at any time).

We will use the slow-clock to drive our nanoprocessor. Therefore, to be able to see the changes as our program executes reduce the clock rate such that it ticks every 2 or 3 seconds.

Students are encouraged to share the workload and work on different PCs in the lab while building different components. For example, while one extends the Register Bank other can work on the 8-way 4-bit multiplexer.

Building the Circuits

Step 1: Design the internal structure of the Instruction Decoder. Clearly identify the role of each of the output pins and how to activate them when necessary.

Step 2: Build the necessary sub-components. Test each component using simulation. As in previous labs, test inputs should be derived from team members' index numbers.

Step 3: Build the top-level design and test using simulation.

Step 4: Write an Assembly program to calculate the total of all integers between 1 and 3 (unfortunately, we cannot work on a larger problem as our processor is only 4-bit wide). Make sure the final answer is stored in Register R7. Remember we can use only the instructions in Table 1 and our PC is only 3-bits long. Convert the Assembly program to machine code and hard code it to ROM.

Step 5: Connecting inputs and outputs.

Output of R7 should be connected to a set of LEDs (**LD0** – **LD3** outputs) and 7-segment display, as our result will be stored on R7. Connect **LD14** and **LD15** for zero and carry flags, respectively.

Step 6: Test on BASYS 3 and verify the functionality of your nanoprocessor.

Demonstrate the circuit to the instructor and get the evaluation done. Be ready to explain how your design work and each team member's contribution.

Step 7: Lab Report

You need to submit a report for this lab. Your report should include the following:

- Student names and index numbers. Do not attach a separate front page
- State the assigned lab task in a few sentences
- Assembly program and its machine code representation
- All VHDL codes
- All timing diagrams
- Conclusions from the lab
- Clearly describe the contribution of each team member to project and number of hours spent

Extra Credit

You may claim extra credits under the following categories:

- Design with least number of basic logic gates:
  - 1st place – 3 marks
  - 2nd place – 2 marks
  - 3rd place – 1 mark
- Other creative designs – up to 3 marks
  - You may consider opportunities to support large registers, extra flags, more instructions, hardware optimizations, etc.
  - Students need to justify the advantage(s) of their creative design.
- These extra features need to be demonstrated during the demo to claim the extra credits.

Bibliography

- Mihir Kedia and Aseem Kishore, "Optional: Building a processor from scratch", MIT Open Courseware, 2008

Prepared By

- Dilum Bandara, PhD – Apr 29, 2014
- Updated on Nov 01, 2018