UNIVERSITY OF BRITISH COLUMBIA
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

EECE 259: Introduction to Microcomputers
Assignment 1: Logic Gates, Binary Numbers and a Computational Datapath
Handed out Jan. 14, 2011

Overview

There are 2 parts to this homework!

1. The **Study Questions (SQ)** are not marked, but they form the basis of the Friday quiz. Solutions to the SQ will be provided.

2. The **Practical Assignment (PA)** is marked *out of 10* by a TA during your PA timeslot. The TA will ask you questions and expect you to demonstrate competence with the material. For example, the TA may ask you to repeat part of the practical with a small change and/or explain your steps. *Effective communication is a part of your grade.* You are graded individually, but you can only be marked **with your partner present.** If your partner cannot attend, s/he gets 0. (Please note the additional policies in the **Course Information** handout about this.)

Please also re-read the *Important Policies* section in the Course Information handout.

PART 0: Important TODO items **** DO THESE RIGHT AWAY ****.

1. Sign up for the course mailing list.

   https://lists.ece.ubc.ca/sympa/subscribe/eece259

2. With your partner, make a 15-minute TA appointment before Thursday, January 20.
   a. Note the timeslots for marking the PA are roughly Thurs 3-5pm and Fri 9-5pm. You must make the appointment at last ***12 hours in advance*** **and** ***no more than 7 days in advance.***
   b. If you still need a partner, use the mailing list mailto:eece259@ece.ubc.ca
3. Buy a DE1 board from ECE Stores in the basement of MCLD for **$100. CASH ONLY.**

PART 1: Study Questions

4. Fill in the following table by writing the required values in both binary and decimal. Why is the *minimum* unsigned value not included in the table?

	Maximum *unsigned*	**Maximum *signed***	**Minimum *signed*** (negative numbers)
4 bits	(binary) % 1111 (decimal) 15	(binary) % 0111 (decimal) 7	(binary) % 1000 (decimal) -8
8 bits			
16 bits			
32 bits			

5. Fill in the rest of the table. Use as many bits / digits as you need.

Decimal	**Binary (%)**	**Hexadecimal ($)**
2241_{10}	% 1000 0100 0001	$ 8 C 1
186_{10}		
41872_{10}		
	% 0111 1011 1000	
	% 1000 0100 0010 0001	
		$ A B C D
		$ E F 8

6. Convert directly from hexadecimal to decimal: $1111, $BEEF, $CAFE, $F00D. Verify your answers by converting directly from decimal back to hexadecimal.
7. Determine the decimal number that results if:
    a. –13 is stored in an 8-bit signed number format and then interpreted (mistakenly) as an 8-bit unsigned number.
    b. 253 is stored in memory as an 8-bit unsigned number and then interpreted (mistakenly) as an 8-bit signed number.
    c. 13 is stored in an 8-bit signed number format and just the lowest 4 bits are interpreted as a 4-bit signed value.
    d. –13 is stored in an 8-bit signed number format and just the lowest 4 bits are interpreted as a 4-bit signed value.
8. Perform *sign extension* for the following examples.
    a. Write the value of 4 in binary using 4 bits. Extend this to an 8-bit value.

b. Write the value of –4 in binary using 4 bits. Extend this to an 8-bit value.

9. To convert a positive number to a negative number in two's complement form, you invert all the bits and add 1. Try doing this for 8-bit values 0 and –128. What happens? Why?
10. Given a 4-bit adder that computes A+B, find a way of re-using that circuit by adding some additional logic so it will compute A–B.
11. The logic gates to produce carryout c_1 of a *full adder* were shown in class. The logic for the carryout C flag was also given. Verify that these two equations are the same. *Hint*: you can create a truth table and verify that both **C** and c_1 are the same, or you can manipulate the algebraic equations to show they are equivalent. Try it both ways!
12. In class, you were given the logic to detect overflows while *adding* two signed numbers ($V_{add}= \overline{a_7} \cdot \overline{b_7} \cdot r_7 + a_7 \cdot b_7 \cdot \overline{r_7}$). Using a similar approach, design overflow logic for *subtracting* two signed numbers, that is compute V_{sub} for the operation A–B.
13. Summarize the logic expression for each flag. Keep in mind that N and Z only look at the result bits, r_7 to r_0. Expressions for V_{add} and N are given as examples.
    a. C =
    b. B =
    c. $V_{add} = \overline{a_7} \cdot \overline{b_7} \cdot r_7 + a_7 \cdot b_7 \cdot \overline{r_7}$
    d. V_{sub} =
    e. N = r_7
    f. Z =
14. (**Long, but you need practice!**). Perform each of the operations in the table below. Express the answer in binary, decimal, and hexadecimal, and give ALL flags: Z, N, V_{sub}, V_{add}, B and C using the logic equations from problem 14. The first row is done for you.

    Hint: flags are **merely logic equations** that don't care *which* operation is performed.

Operation on 8-bit values	**Results**								
	Binary	**Decimal**	**Hex**	**Z**	**N**	**V_{sub}**	**V_{add}**	**B**	**C**
a. 114 + 24	% 1000 1010	138_{10}	$ 8 A	0	1	0	1	1	0
b. $37 + $34	% __ __ __ __ __ __ __ __	__ __ __	$__ __	__	__	__	__	__	__
c. $37 + $44	% __ __ __ __ __ __ __ __	__ __ __	$__ __	__	__	__	__	__	__
d. $13 + $EC	% __ __ __ __ __ __ __ __	__ __ __	$__ __	__	__	__	__	__	__
e. $13 + $ED	% __ __ __ __ __ __ __ __	__ __ __	$__ __	__	__	__	__	__	__
f. $13 + $EE	% __ __ __ __ __ __ __ __	__ __ __	$__ __	__	__	__	__	__	__
g. $83 + $96	% __ __ __ __ __ __ __ __	__ __ __	$__ __	__	__	__	__	__	__
h. $F0 + $02	% __ __ __ __ __ __ __ __	__ __ __	$__ __	__	__	__	__	__	__
i. $24 – $3B	% __ __ __ __ __ __ __ __	__ __ __	$__ __	__	__	__	__	__	__
j. $FD – $07	% __ __ __ __ __ __ __ __	__ __ __	$__ __	__	__	__	__	__	__

PART 2: Practical Assignment

There are 3 parts to this practical assignment, consisting of 3 different circuits containing:

A) basic logic gates,

B) register file, and

C) computational datapath.

Each part requires a **different** *bitstream file* to program your Altera DE1 board.

Installing Software, Opening Projects and Generating Bitstream Files

Visit the course web site for Altera installation and programming instructions:

http://courses.ece.ubc.ca/259/homework/installing.htm

http://courses.ece.ubc.ca/259/homework/programming.htm

Next, you must generate the **.pof** and **.sof** bitstream files for the homework. Download the appropriate ZIP file containing all of the circuits:

http://courses.ece.ubc.ca/259/homework/hw1/files/

After you unzip the files, there will be 3 folders. Enter the first folder, **hw1A_DE1**, and double-click the **.qpf** or Quartus project file. This is the main top-level file that describes all of the features, settings, and files needed for the circuit. It may take a minute or two for Quartus II to start up.

Go to **Processing → Start Compilation** to start generating the bitstream files. This step can take several minutes as the mapping process is quite involved. While it is running, look for the Status area to see the progress and browse some of the many information messages in the bottom area. You can ignore any warnings. When Quartus II finishes, press the OK button.

Go to **Tools → Programmer**. Make sure **USB-Blaster** is displayed beside the **Hardware Setup…**button. If is not displayed, check the *Altera programming* instructions link above.

In the programmer window area, check the box under **Program/Configure** and press the **Start** button. It should finish in 1 or 2 seconds.

Digging deeper – interesting, but not necessary for this course!

If you want to learn more about Quartus II and VHDL, go ahead. Start by double-clicking **hw1A** underneath **Entity** in the top-left corner of the window. This will open the main VHDL file that describes the circuit, **hw1A_DE1.vhd**. If you click the + symbol next to **hw1A** you can browse and open the subcircuits it uses as well. Subcircuits starting with **lpm_** are parts of Altera's own library, e.g. for the multiplier – **don't modify these**! You can learn about VHDL from your EECE256 textbook, or by reading about it on the web. Even without using the textbook, you should be able to understand some parts of the language and change it. For example, try to find the multiplier operator "*****" and change it to "**XOR**" instead!

A. Basic Logic Gates

15. A logic diagram for this circuit is shown in Figure A. Download the programming file and program your board. You probably want to use the *temporary programming method*. There are actually **4 distinct circuits** that are now programmed on your board. Try each:

    a. The first two circuits are two distinct 2-input logic gates, labeled **U1** and **U2**. Both gates share the same two inputs, switches SW9 and SW8. The outputs are the red LEDs: U1 drives LEDR9, while U2 drives LEDR8.

    Fill in the table below and **determine which gates** were implemented in A and B.

Inputs		**Outputs**	
SW9	**SW8**	**U1**	**U2**
0	0		
0	1		
1	0		
1	1		

Conclusions based on observations…

1. Logic Gate **U1** is an _____ gate.

2. Logic Gate **U2** is an _____ gate.

    b. The third circuit is an **Accumulator**. The outputs of the accumulator are shown in hexadecimal on HEX0, and in binary on the green LEDs labeled LEDG[3:0]. The accumulator has only two inputs: **reset** and **clock**. Find the switches that control these two signals. When you press **reset**, the register inside the accumulator is reset to 0, but you will see 1 on the output. Why? On the rising edge of the **clock**, the accumulator should increment by one.

    Q: When does the *rising edge* of the clock occur, on **PRESS** or **RELEASE**?

    Fill in the table below according to your observations.

Hex	**Binary**	**Hex**	**Binary**	**Hex**	**Binary**	**Hex**	**Binary**
$0	%____	$4	%____	$8	%____	$C	%____
$1	%____	$5	%____	$9	%____	$D	%____
$2	%____	$6	%____	$A	%____	$E	%____
$3	%____	$7	%____	$B	%____	$F	%____

    c. The fourth circuit is an **ALU**. The function calculated by the ALU is controlled by SW[1:0] according to the ALUop table in the figure. The answer is displayed on HEX1 as well as LEDG[7:4]. The inputs of the ALU are from two 4-bit registers, **A** and **B**. You can verify the register contents: **A** is displayed on HEX3 and LEDR[7:4], while **B** is displayed on HEX2 and LEDR[3:0]. After a **reset**, the **A** and **B** registers hold $0, and the ALU should be 0 for *all* functions.

    To load a new value into **A**, you must place the value on dataIn using SW[7:4] and set the enable signal for a, wrA on SW3, to 1. Of course, the value is not accepted in A until you send a clock pulse. Similarly, you can load a value in **B**.

    Q: How do you load the same value into **A** and **B** simultaneously? Try it!

Load the following values into **A** and **B** and observe the ALU calculations. The last few rows are left blank for you to try your own values for **A** and **B**. *Try to choose meaningful values that give interesting results for all (+ – * &) functions.*

A	B	A+B	A–B	A*B	A&B
%0001	%0010	$ __ % _ _ _ _	$ __ % _ _ _ _	$ __ % _ _ _ _	$ __ % _ _ _ _
$4	$8				
$8	$4				
$7	$F				
$3	$3				
$3	$4				
$A	$5				

B. Register File

16. The **Register File** circuit is shown in Figures B1 and B2. The purpose of this circuit is to experiment with reading and writing values to different registers. Navigate to the **hw1B** folder and open the **.qpf** project file. Create the programming file and program your board. Again, you probably want to use the *temporary programming method***.**

    a. Press the **reset**. To make things interesting, some of the flip-flops have this signal connected to their *preset* input instead, causing them to initialize to 1 instead of 0.

    b. Read all 4 registers to determine their initial value. Do this by selecting each one using selRd[1:0] and viewing dataOut[7:0]. You don't need to press **clock**. *Why?*

Register	selRd	Initial Value	Register	selRd	Initial Value
R0	%00	$ ___ ___	R2	%10	$ ___ ___
R1	%01	$ ___ ___	R3	%11	$ ___ ___

    c. Write the value $0F to register R2. Do this by setting dataIn[3:0] to all 1s, selWr[] to select R2, wrReg to 1, and sending a **clock** pulse. Verify that the other registers retain their original value and that **only** R2 has changed to $0F.

    Q: What happens if you press **clock** a second time? … a third time?

    d. Initialize all registers to 0. Verify they are all 0.

    e. Press **reset**. Verify that all registers have the correct initial values.

    f. Load the values 0, 1, 2 and 3 into registers R0, R1, R2, and R3 respectively. Verify they have the correct new contents.

C. Computational Datapath

17. The **Computational Datapath** is shown in Figure C2. Download the programming file and program your board. Since this circuit can be fun, you might want to try the *permanent programming method* so it is always available at power-up. Notice this circuit contains an 8-bit version of the **ALU** from Part A and the **Register File** from Part B.

    This circuit will teach you ***how to compute using a datapath***. You are the brains behind the circuit, so you must decide correct values of all **control signals** that are brought out to the switches. Also, you must figure out **when to send a clock pulse**. It is this combination of control signals and clock pulses that **govern the computation**. The logic gates are the minions doing the calculations, but you are the one instructing it!

    The switch settings you choose for each clock cycle are equivalent to one "machine language" instruction given to the computer.

    The most important thing for you to focus on is the **Bus**. This signal goes almost everywhere, so it is shown as a thick, bold line. The contents of the Bus are displayed on HEX3 and HEX2; since it is 8 bits, two hex digits are required. To control the minions, there are two things to ask about the Bus:

    a. Where does the Bus value come from? (from: **Register File**, **ALU**, constants)

    b. Where does the Bus value go? (writing to: Register File, A, or B)

    Important: there can be only one value on the Bus in each clock cycle.

    Where does the value on the Bus come from? The Bus value, determined from the settings of selBus[1:0], can be the constants **$00** or **$01** (settings %00 and %01, respectively), the **Register File** output (%10), or the **ALU** output (%11). If you select the **Register File**, you must also determine which register is to be output using selReg[1:0]. Similarly, if you select the **ALU**, you must set ALUop[1:0]. As the brain behind this computer, you must have first loaded meaningful values into **A** and **B** before using the **ALU**. Or, if you are using the **Register File**, you must have first stored a meaningful value in that register already.

    Where does the value on the Bus go? Of course, it is displayed on HEX3 and HEX2. However, this is just for your convenience! To do *useful* calculations, the value on the Bus must be written to *some register* inside the datapath. We say this is *changing the state* of the system, and it is the advancement of this changing state that helps us compute a final result. In this datapath, the Bus can go to registers **A** or **B** or *one* of registers in the **Register File** selected by selReg[1:0]. Although you can write to all 3 of these register destinations at the same time, it is more typical to write to only one destination.

18. After **reset**, verify whether the initial values of the **Register File** in your datapath are the same as you found earlier with the standalone **Register File** circuit from Part B.

19. Use your **Computational Datapath** to initialize the registers as follows:

    R0 = 1, R1 = 2, R2 = 3, R3 = 4.

    Do not rely upon the registers containing *any particular initial value*. The only reliable values you have are the **$00** and **$01** constants in the datapath.

To solve this problem, write out the sequence of steps required in **RTN**. Make sure all complex operations are broken down into a sequence of simple μ-ops. To make your job easier, you probably want to write out the control signal values in a worksheet

http://courses.ece.ubc.ca/259/homework/hw1/hw1worksheet.pdf

20. Use your **Computational Datapath** to compute 1+2+3+4. To solve this problem, write out the sequence of steps required in **RTN**. Do not rely upon the registers having any particular initial values – you can add to your solution from the previous step, or you can start from scratch. There are many possible ways of doing this – I have one solution that requires 10 μ-ops, and another that requires 25 μ-ops.

The problems below are "extra work" for the curious. **You do not need to do them for the PA**, but you may wish to do them to help you study.

21. (**Easy!)** Use your **Computational Datapath** to compute the Fibonacci sequence:

    $F_i = F_{i-1} + F_{i-2}$ where $F_2 = F_1 = 1$.

    If you can find the trick to get it started, this becomes very easy!

22. (**Not too hard, but long!**) Use your **Computational Datapath** to compute the following summation:

    $S = \sum_{i=1}^{4} F_i$, where $F_i = 2 \times F_{i-1} + F_{i-2}$ and $F_2 = F_1 = 1$.

    To perform this summation, you will need to use the registers to hold some intermediate values. *Hint*: to keep this easy, don't try to write a "loop" that stores *i* in a register. Instead, try the simple approach:

    R0 holds F_1, R1 holds F_2, R2 holds F_3, R3 holds F_4.

    Then, add the values R0+R1+R2+R3 (storing the final result in, say, R3).

23. (**A bit tricky!**) Repeat the last problem using a different series: $F_i = 8 \times F_{i-1} + F_{i-2}$.

24. (**Very difficult!**) Find a way to multiply two 4-bit numbers *without* using the multiply operation. You should use the *exact same* method for *any* two 4-bit numbers. That means a robot could repeat the exact same sequence of switch presses for any two 4-bit numbers and it would get the correct result. *Hint: I don't have one… but I think it is possible!*

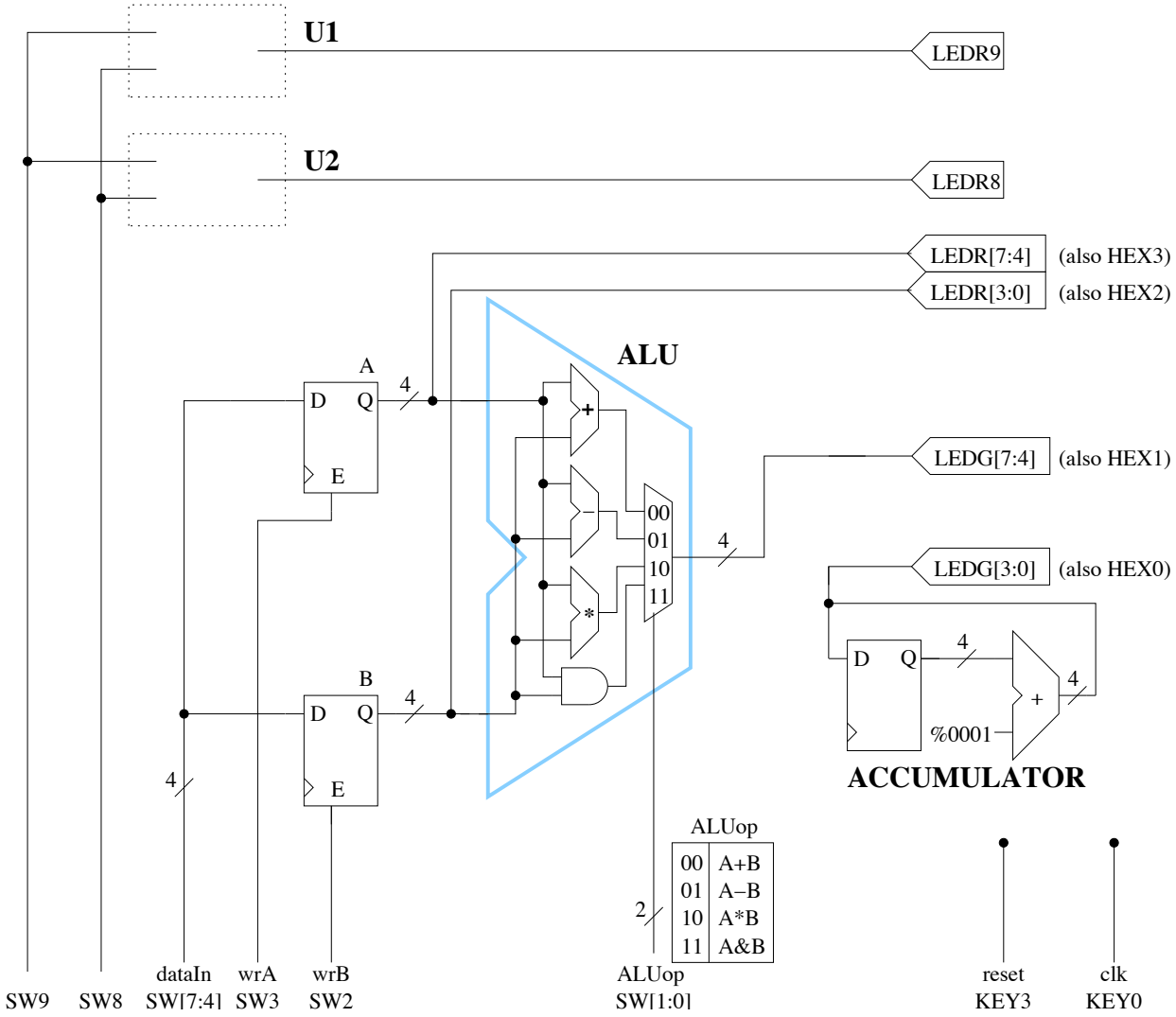


Figure A. Basic Logic Gates circuit.

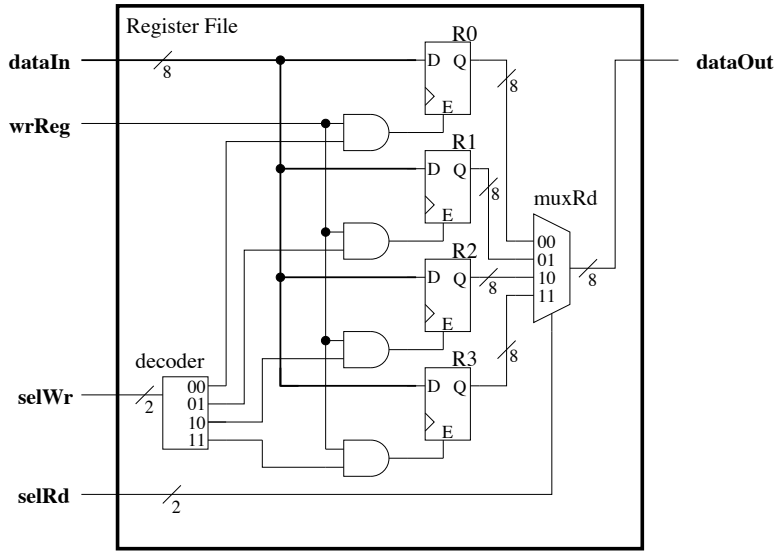


Figure B1. Register File details.

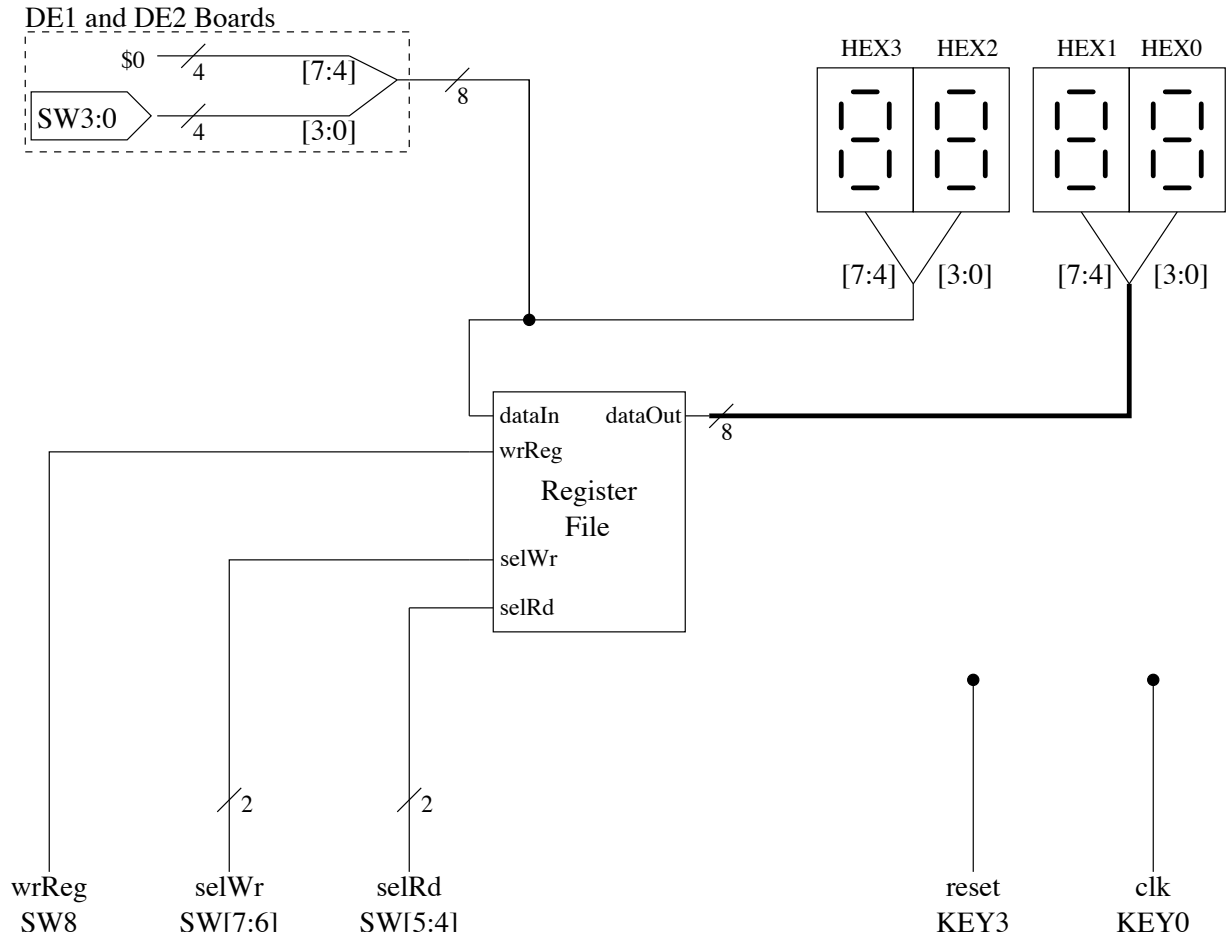


Figure B2. Register File circuit.

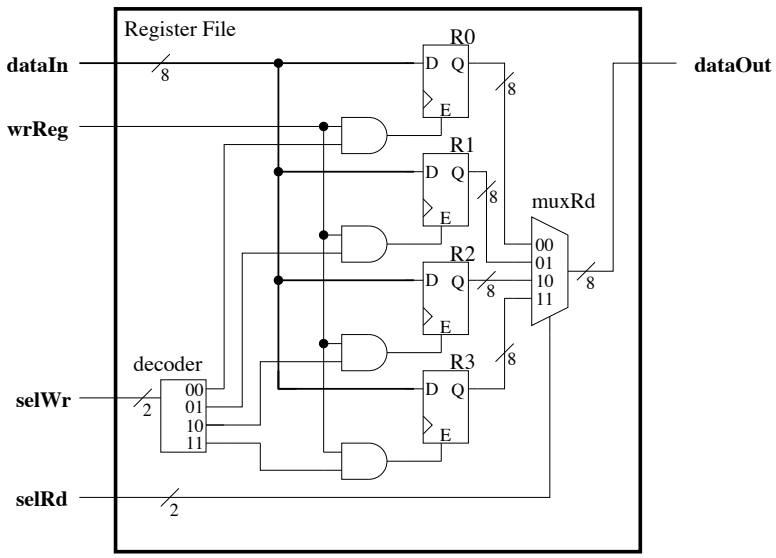


Figure C1. Register File details (same as Figure B1).

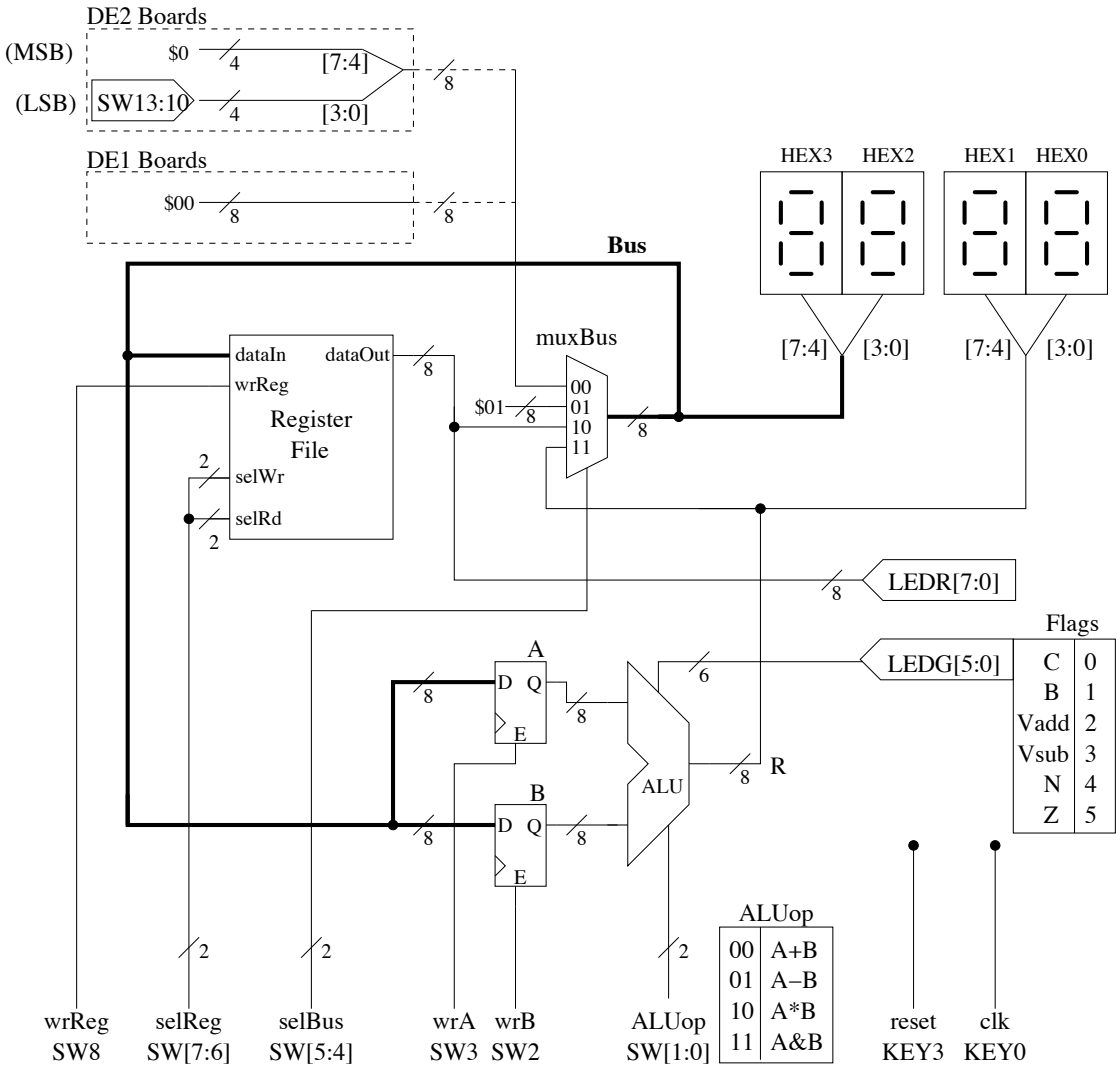
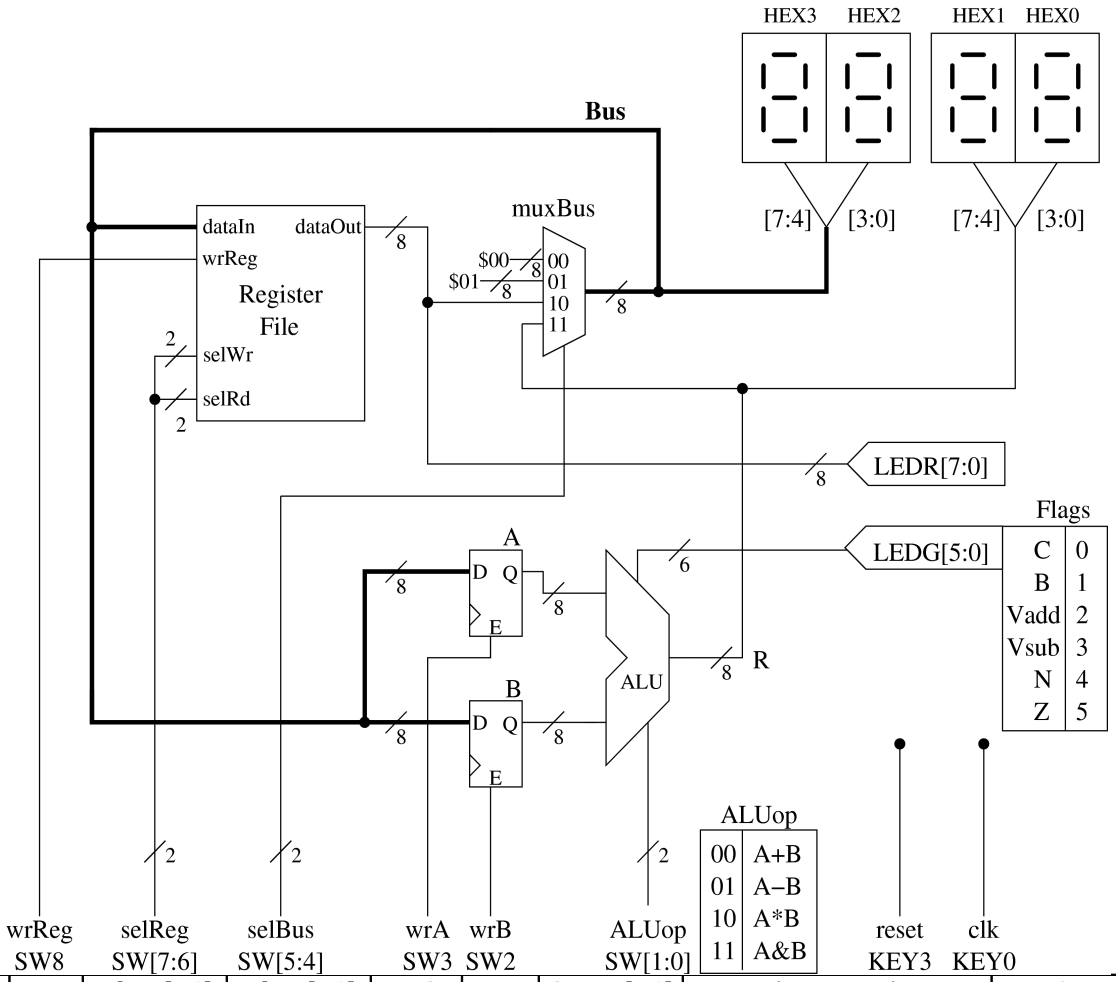


Figure C2. Computational Datapath circuit.



Cycle #	wrReg	selReg[1:0]	selBus[1:0]	wrA	wrB	ALUop[1:0]	Micro-operation	Overall Operation
1.								
2.								
3.								
4.								
5.								
6.								
7.								
8.								
9.								
10.								
11.								
12.								
13.								
14.								
15.								
16.								
17.								
18.								
19.								
20.								