CS2310 – Digital Logic Design Lab
July-Nov. 2014, Prof. Krishna Sivalingam
Lab 2 – A "Bit" of Combinatorial Logic Design
Lab Date: Aug. 12, 2014
Record Book Due Date: Aug. 18, 2014, 12 PM

Problem Statement

In this lab, you will be designing and implementing circuits to solve the following two problems. You can use any of the basic 2-input logic gates (AND-OR-NOT-NAND-NOR-XOR) in your implementation, but you can use at most one IC chip of a certain type (eg., you can't use two IC7402's for Problem 1), for each problem.

1. Determinant of a binary matrix

Design a circuit that evaluates the determinant of a 2×2 *binary* matrix.
(Note : State any assumptions made about input and output representations)

2. 2-bit Comparator

Design a circuit that takes two unsigned 2-bit numbers (a and b), and displays one of *greater* ($a > b$), *lesser* ($a < b$) or *equal* ($a == b$) signals.

Tip :- You are not required to implement the above in the most efficient way for full credit, but doing so definitely helps with wiring.

Pre-Lab Requirement

Please come prepared on 12th August with the truth tables and circuit design (in rough) for the above two problems.

CS2310 – Digital Logic Design Lab
July-Nov. 2014, Prof. Krishna Sivalingam
Lab 3 – The Dark Adder
Lab Date: Aug. 19, 2014
Record Book Due Date: Aug. 25, 2014, 12 PM

Problem Statement

Adders form a core component of the Arithmetic Logic Unit (ALU) and play a major role in calculating memory addresses, table indices etc., in Computer Processors. In this lab, you will be learning and implementing Adder circuits for unsigned numbers.

The *Half adder* takes in two input bits and produces two output bits, the *sum* and the *carry,* the *XOR* and *AND* of the two bits respectively.

The *Full adder* takes in two input bits and a third bit (*carry-in*). It also produces two output bits, the *sum* and the *carry-out*. Their truth tables are given below.

Truth table for Half adder

INPUTS		OUTPUTS	
A	B	SUM	CARRY
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Truth Table of Full Adder

Inputs			Outputs	
X	Y	Z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Figure 1: Truth tables for Half Adders and Full Adders

1. Half Adders and Full Adders

You will first implement a Half Adder using basic logic gates, and then implement a Full Adder using Half Adders and basic logic gates.

2. Ripple-carry Adders

Connect multiple adders (half/full) to display the 4-bit sum of two 3-bit numbers.

3. The Dark Adder

The Dark Adder ($\widetilde{+}$) is a devious adder that *inverts* the carry bit before rippling it to the next stage. For example, $111 \widetilde{+} 111 = 0000$. Make suitable changes to convert the Ripple-carry adder

implemented in (2) into a 3-bit dark adder.

Bonus Question

A student implemented a standard 3-bit Ripple Carry Adder (Half, full, full) using basic logic gates. But, to her surprise, she found that the circuit performed Dark Addition instead. It turns out that one of the (naughty) TAs had altered the labels on some of the ICs before she took them to build her circuit. Help her figure out what ICs she ended up using instead.

Hint : Use this to simplify your implementation of the dark adder

4. Introducing Multipliers

Design a circuit that multiplies two 2-bit numbers. You are allowed to use IC7483 in this section.

Pre-Lab Requirement

Please come to lab with the rough circuit diagrams and expressions for the above sections.

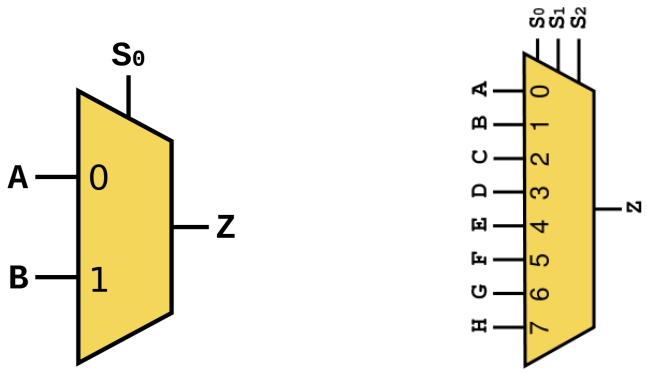
References

Refer Chapter 5.2 of the book by Brown and Vranesic for additional reading on adder circuits and unsigned addition. Refer Chapter 2 in the same book to learn to derive circuits from truth tables, and Chapter 4 for advanced gate minimization techniques (optional).

CS2310 – Digital Logic Design Lab
July-Nov. 2014, Prof. Krishna Sivalingam
Lab 4 - Mux and Match
Lab Date: Aug. 26, 2014
Record Book Due Date: Sept. 9, 2014, 2 PM

Problem Statement

A *multiplexer* (or *mux*) is a device that selects one of several input signals as the output, based on a separate selection input. For example, for the 2:1 mux shown in Figure 1a, the main inputs are A and B, and the selection input is S_0(Let's call it S for now). The output Z is A if S is 0, B if S is 1. The corresponding boolean expression can be worked out from the truth table of the mux to be $Z = \overline{S}A + SB$. The selection line S essentially chooses one of the inputs to forward to the output line and can be thought of as the *tie-breaker* for competing inputs. This notion extends to a larger number of input lines and multiple selection lines. For example, in a 8:1 mux, we have eight inputs and three selection lines S_0, S_1, S_2. A selection signal combination of say 101 chooses the 5th input line as the output.



(a) 2:1 Mux, $Z = \overline{S}A + SB$ (b) 8:1 Mux

Figure 1: Multiplexers

In this lab, you will be reading up on Multiplexers (see References) and using them to implement logic circuits.

1. Binary to One-Hot

The *one-hot* representation of a n-bit binary number (equivalent decimal number i) is a sequence of 2^n bits where the ith bit (from LSB) is 1, and the rest are 0's. For example, the one-hot representation for 01 would be 0010, and that for 11 would be 1000. Given a **2-bit** binary number as input, design a circuit using only **basic logic gates** to convert it into its *one-hot* representation. (*Clarification* : No muxes allowed yet).

2. Railway Crossing

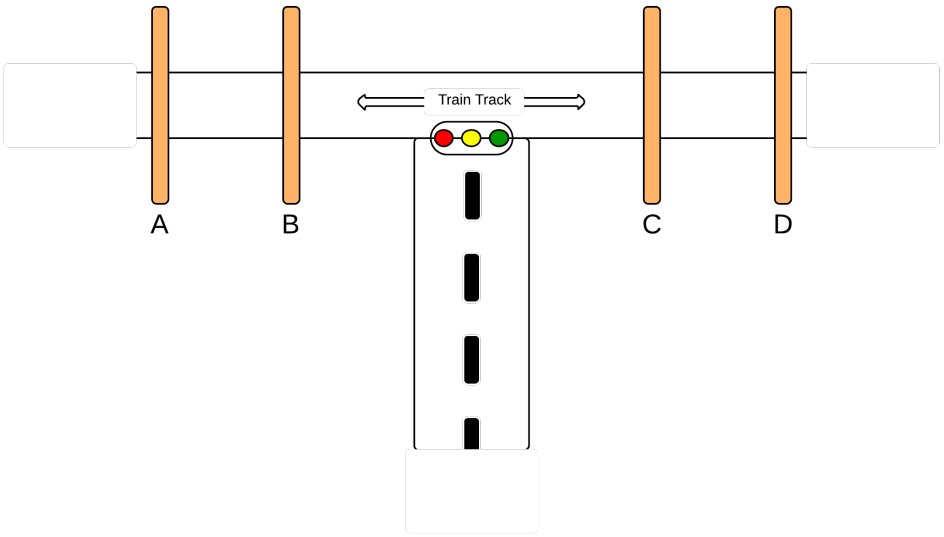


Figure 2: Railway Crossing

At the railway crossing, there is a traffic signal that shows one of *red, yellow* or *green* lights. Four sensors, A, B, C and D are on the track on both sides as shown. Sensors B and C are equidistant from the road, and are distance of 200 meters apart from each other. Sensors A and D are at a distance of 100 meters from B and C respectively. Each sensor gets active when a train is passing through it, and is inactive otherwise. Trains that operate on this track are of length 250 meters and operate in both directions. In this section, you will be designing the logic for the traffic signal using **only** multiplexers (2:1 or 4:1), that adheres to the following rules.

1. At any point of time, exactly one of the lights is on.
2. If any portion of the train is between B and C, then the signal shows red.
3. When none of the sensors are active, the signal shows green.
4. In all other **valid** cases, the signal shows yellow.

   You are expected to have **three** different LEDs for the output: one for each red, green and yellow. You are **allowed** to use the circuit implemented in (1), along with your muxes.

   Hint: By using the circuit implemented in (1), can you reduce the work needed to implement (2) ?

Pre-Lab Requirement

Please come to lab with the rough circuit diagrams and expressions for the above sections.

References

For further reading on Multiplexers and on implementing truth tables with multiplexers, refer **Chapter 6** of the book by Brown and Vranesic.

CS2310 – Digital Logic Design Lab
July-Nov. 2014, Prof. Krishna Sivalingam
Lab 5 - Combinatorial Counting
Lab Date: Sept. 9, 2014
Record Book Due Date: Sept. 15, 2014, 2 PM

Problem Statement

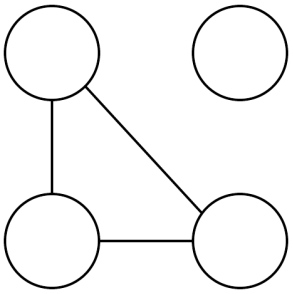
Consider a *simple, undirected* graph with **four** vertices. You will design circuits that count the following :-

1. The total number of *cycles* in the graph.
2. The number of *connected components* in the graph.
   A *connected component* of an undirected graph is a subgraph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices in the supergraph.

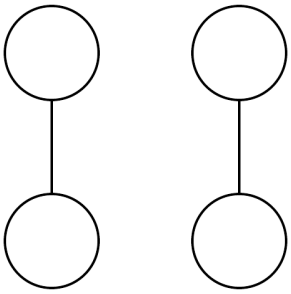
For both problems, you will be presenting the output by displaying the corresponding number on a 7-segment display.

You are allowed to use any of the basic logic gates, adders, muxes, decoders and encoders for this assignment.

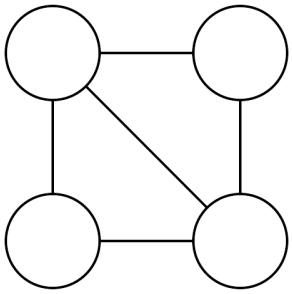
Hint : Reduce and reuse.



(a) One cycle and two connected components



(b) Zero cycles and two connected components



(c) Three cycles and one connected component

Figure 1: Sample undirected graphs with four vertices

Pre-Lab

Please come prepared with your circuit designs in rough for the lab.

References

- Chapters 2, 4, 5, and 6 in Brown and Vranesic.
- *http://en.wikipedia.org/wiki/Graph_theory*

CS2310 – Digital Logic Design Lab
July-Nov. 2014, Prof. Krishna Sivalingam
Lab 7 - NOTs and Crosses
Lab Date: Sept. 22, 2014
Due Date: Oct 14, 2014

Problem Statement

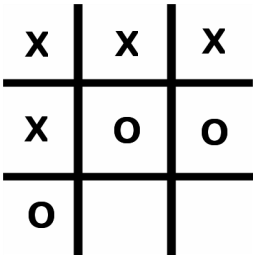
Congratulations! You have been selected to be the judge for a Tic-Tac-Toe contest. Given the snapshot of a Tic-Tac-Toe board, it is your task to determine the state of the board, which can be one of the following *four* states:

1. Player X wins.
2. Player O wins.
3. Invalid board snapshot - An invalid snapshot is a snapshot that can *never* arise via *any* game-play sequence (Assume that X always starts first).
4. Tie or game-incomplete.

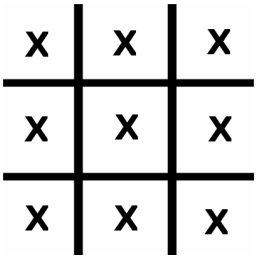
Since you will be busy doing other assignments and have no time to attend the contest, you will summon your knowledge of digital logic circuits to design a combinatorial logic circuit to do the judging on your behalf.

You can use any of the basic logic gates, muxes, decoders, encoders. Write verilog code to implement your circuit design and verify it using the Xilinx simulator. *Hint* : Write your own custom modules and re-use them effectively.

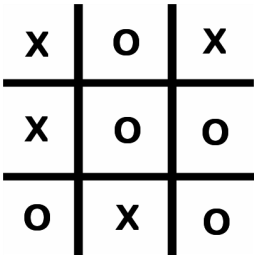
NOTE: Your group will show circuit designs and working code to your respective TAs on or before the due date. Extra credit will be provided for intelligent usage of circuit elements.



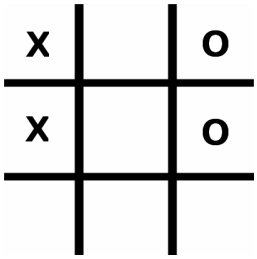
(a) X wins



(b) Invalid



(c) Another invalid position (X starts the game)



(d) Tie or game-incomplete

Figure 1: Sample game snapshots

References

1. Video Tutorial on Xilinx Simulations : *Link to Video*
2. Refer the pdfs on moodle.

Honor Code

You are only allowed to collaborate with students *of your own group* for this assignment. Any form of code-sharing across groups is not allowed. Please approach the TAs if your group needs any help.

CS2310 – Digital Logic Design Lab
July-Nov. 2014, Prof. Krishna Sivalingam
Lab 8 - Latch on
Lab Date: Oct 14,, 2014
Due Date: Oct 21, 2014

Problem Statement

This assignment introduces you to sequential circuits through gate-level construction of the basic elements that define them. Up until the previous lab, you designed and implemented combinatorial circuits i.e circuits in which the output was a function of *only* the inputs. Sequential circuits extend this by introducing the notion of time. They allow the output at time *t* to be a function of not *only* the input at time *t*, but also the output at time *t-1* (the *previous state*). This can be achieved by carefully feeding back outputs of certain elements in the circuit as inputs to other elements in the same circuit. In this lab, you will be implementing the basic circuits that can achieve this functionality.

1. Lazy Lamps

Your aim is to design a circuit to control a light bulb using two push-button switches. If we push the first switch (A), the light should turn on. If we release A, the light should stay on. If we push the second switch (B), the light should turn off if it was on, and remain off if it was off. The light must also stay off after releasing B. Assuming that A and B are not pushed at the same time, design a circuit to implement this functionality. Write verilog code for the same and verify it on the Xilinx simulator or the FPGA board. You can only use basic logic gates (AND-OR-NOT-NAND-NOR-XOR). Assume that a push is 1 and release is 0 or vice versa.

- What does your circuit output when both the buttons are pushed?
- Suppose we assume that all the six gates given have a common input–to–output delay of δ. What is the total time that your circuit takes for a change in the input to get reflected in the output? Assume that the output from the previous state is readily available (wherever needed in your circuit). Redesign your circuit to minimize the maximum delay that your circuit can incur for a change in A or B to get reflected in the output. How does this circuit behave when both A and B are pressed at the same time?

2. Master Control

In the previous circuit, suppose there existed another push-button switch called the *master switch.* If the master switch is in the release state, then the other push buttons must have no effect on the circuit's output. If this master switch is in the push state, then the other two push buttons

must function as described in the previous circuit. Extend the previous circuit to achieve this functionality.

3. Stabilizing

One solution to Lazy Lamps is to use cross–coupled NOR gates i.e an S-R latch. But, the S-R latch enters into an unstable state when both the inputs are 1. One way to handle this problem is to enforce $A = \overline{B}$ which is equivalent to taking only one input from the user and feeding its complement as the other input. Implement the S-R latch with this condition imposed, assuming that master control exists. What does the circuit simplify to if the master control for this circuit is removed? What is the reason for this bizarre observation?

NOTE: For all the above problems, you will write verilog code and test it on the Xilinx simulator/FPGA board. You will also document the truth tables, circuit diagrams, and boolean expressions for these circuits in your record notebooks along with answers to the questions given in the problem statement. You are expected to get a thorough understanding of the concepts involved in this assignment.

Honor Code

You are only allowed to collaborate with students *of your own group* for this assignment. Any form of code-sharing across groups is not allowed. Please approach the TAs if your group needs any help.

CS2310 – Digital Logic Design Lab
July-Nov. 2014, Prof. Krishna Sivalingam
Lab 9 - Sleep Sort
Lab Date: Oct 21, 2014
Due Date: Oct 29, 2014

Introducing Sleep Sort

Sleep sort is an interesting sorting algorithm which exploits the notion of time. Given n numbers $a_1, ...a_n$, the algorithm first creates a separate task for each number. Each task i prints the number a_i after waiting (sleeping) for a_i time units (nanoseconds, for example). All tasks start at the same time, run in parallel, and print to a common output screen. This ensures that the output is sorted.

Eg. To sort the sequence $\{7, 3, 4, 1, 2\}$, sleep sort creates 5 separate parallel tasks i.e one for each of the numbers in the sequence. Task 1 waits for 7 nanoseconds before printing 7 onto the screen. Task 2 waits for 3 nanoseconds before printing 3 onto the screen, and so on.

Problem Statement

Your aim is to design a sequential circuit to sleep–sort **eight distinct** 4-bit numbers. To be more precise, your circuit will print the number a_i at the a_ith time unit (in seconds, say). You will use Verilog and the Xilinx Simulator to implement your design. You will keep track of time by maintaining a clock bit that toggles value every half a second, so that one clock cycle represents one second.

You are allowed to use only the basic logic gate modules. You are allowed to write behavorial code only to toggle the clock bit every half a second. The rest of your code should only be structural verilog.

Hint: First implement a 4-bit **counter circuit** (synchonous/asynchronous) using flip-flops of your choice (which you will also implement from scratch using basic logic gate modules) as a starting point.

You are also free to implement your own approach to the problem, if different from the above.

Challenge Version

This section is purely optional and significant extra credit will be rewarded if implemented. Extend the circuit implemented above to support duplicate numbers in the input sequence of 8 numbers. It is not necessary to print the duplicate numbers at the same physical time instant. You can print them in successive clock cycles.

Hint: Find a way to freeze the counter for a certain number of clock cycles based on some external input (count of duplicates).

Honor Code

You are only allowed to collaborate with students *of your own group* for this assignment. Any form of code-sharing across groups is not allowed. Please approach the TAs if your group needs any help.

Reference & Reading

For further reading on Flip flops and on implementing counters, refer **Chapter 7** in the book by Brown and Vranesic.

CS2310 – Digital Logic Design Lab
July-Nov. 2014, Prof. Krishna Sivalingam
Final Project
The Ghost Processor
Verilog Code Upload: Nov. 13, 2014, 5PM via Moodle
Demo dates: Nov. 14, 9am – Nov. 17th, 6pm
Record Book Due Date: Scheduled Viva Voce Exam Time

1 Introduction

CS-13 is an abandoned classroom in the ground floor of the Computer Science and Engineering department at IIT Madras. It is believed that the room is haunted by the ghosts of dismantled computers! The old and broken computers are envious of the powers that modern computers are being built with and have planned to take revenge on all Computer scientists. It is up to you, the CS-13 batch, to live up to your legacy and save the department from the wrath of the CS-13 ghosts! To show that Computer scientists still care for them, you will design and build a basic microprocessor (similar to those in the old computers) to convince the ghosts that people haven't forgotten about them! The specification of the microprocessors in the ghost computers has been given in this document. You will design and build a consolidated digital logic circuit to replicate its functionality.

2 The Ghost Processor: CPU Details

The legacy ghost microprocessor is a 4-bit CPU that can support the instructions specified below. Verilog will be used for the implementation; the FPGA will be programmed with this implementation, and used for the program execution. You will be given a small program in binary (a sequence of instructions from the pool of instructions described below) . Your CPU must be capable of executing the program ensuring correctness of output and must also correctly simulate the time needed for each of the instructions.

Each CPU instruction contains 12 bits: of this, the first 4 bits specify the specific operation to be done (add, move, etc.); The CPU has four 4-bit registers for storage; register R0 is represented by 00, R1 by 01, etc. Signed integers are used and represented with two's-complement representation. The CPU also includes Carry and overflow flags (single bits), that are set depending upon the output of an instruction's execution. For instructions that need less than 12 bits, you can assume that the remaining bits are don't cares. The time taken (in clock cycles) for the execution of each of the instructions is also given.

The circuitry for all the instructions should be realized using **basic** Logic gates. The implementation has to be purely in **structural verilog** (except for the implementation of the clock, for which you can use behavioral code).

3 CPU Instructions

The different CPU instructions are listed below.

1. MOVK R1, K

Instr (4)	Regr. (2)	Constant (4)	

   R1 is destination register; K is a 4-bit signed integer; This instruction stores the value of K into R1. The instruction bits are: 0000. This instruction takes 2 clock cycles i.e the result is reflected in the destination register(s) two clock cycles after the instruction begins execution.

2. MOVR R1, R2

Instr (4)	Regr. (2)	Regr. (2)	

   R1 is destination register; R2 is source register; R2's value is copied to R1. The instruction bits are: 0001 (for all subsequent instructions, the list number equals the instruction code). This instruction also takes 2 clock cycles.

3. INC R1

Instr (4)	Regr. (2)		

   R1 is incremented by one. This instruction takes 1 clock cycle.

4. DEC R1

Instr (4)	Regr. (2)		

   R1 is decremented by one. This instruction takes 1 clock cycle.

5. INC R1

Instr (4)	Regr. (2)		

   R1 is incremented by one. This instruction takes 1 clock cycle.

6. ADD R1, R2, R3

Instr (4)	Regr. (2)	Regr. (2)	Regr. (2)

   R1 is destination register; $R1 = R2 + R3$. Note: the destination register can also be one of the two inputs. You will be implementing a **Carry-lookahead adder** for this section (and also for subtraction). Refer Chapter 5.4 in Brown and Vranesic for details. This instruction takes 3 clock cycles.

7. SUB R1, R2, R3

Instr (4)	Regr. (2)	Regr. (2)	Regr. (2)

   R1 is destination register; $R1 = R2 - R3$. Note: the destination register can also be one of the two inputs.

   This instruction takes 3 clock cycles.

8. MUL R1, R2, R3

Instr (4)	Regr. (2)	Regr. (2)	Regr. (2)

R1 is destination register; $R1 = R2 * R3$. Note: the destination register can also be one of the two inputs. Since you are dealing with signed multiplication in two's complement, you will implement **Booth's algorithm** for this section.

This instruction takes 3 clock cycles.

9. AND R1, R2, R3

Instr (4)	Regr. (2)	Regr. (2)	Regr. (2)

R1 is destination register; $R1 = R2 \& R3$ (bit-wise AND). Note: the destination register can also be one of the two inputs.

This instruction takes 2 clock cycles.

10. OR R1, R2, R3

Instr (4)	Regr. (2)	Regr. (2)	Regr. (2)

R1 is destination register; $R1 = R2 | R3$ (bit-wise OR). Note: the destination register can also be one of the two inputs.

This instruction takes 2 clock cycles.

11. NOT R1

Instr (4)	Regr. (2)		

The bit-wise NOT of R1 is stored back in R1.

This instruction takes 1 clock cycle.

12. SL R1, K

Instr (4)	Regr. (2)	Constant (K)	

R1 is left-shifted by K-bits ($K \leq 3$). This instruction takes 1 clock cycle.

13. RL R1, K

Instr (4)	Regr. (2)	Constant (K)	

R1 is right-shifted by K-bits ($K \leq 3$). This instruction takes 1 clock cycle.

14. SLC R1, K

Instr (4)	Regr. (2)	Constant (K)	

R1 is circular left-shifted by K-bits ($K \leq 3$). This instruction takes 1 clock cycle.

15. RLC R1, K

Instr (4)	Regr. (2)	Constant (K)	

R1 is circular right-shifted by K-bits ($K \leq 3$). This instruction takes 1 clock cycle.

4 Flag Details

http://teaching.idallen.com/dat2343/10f/notes/040_overflow.txt

4.1 The Carry Flag

The rules for turning on the carry flag are:

- The carry flag is set if the addition of two numbers causes a carry out of the most significant (leftmost) bits added.

  1111 + 0001 = 0000 (carry flag is turned on)

- The carry (borrow) flag is also set if the subtraction of two numbers requires a borrow into the most significant (leftmost) bits subtracted.

  0000 - 0001 = 1111 (carry flag is turned on)

Otherwise, the carry flag is turned off.
0111 + 0001 = 1000 (carry flag is turned off)
1000 - 0001 = 0111 (carry flag is turned off)

4.2 The Overflow Flag

The rules for turning on the overflow flag are:

- If the sum of two numbers with the sign bits off yields a result number with the sign bit on, the overflow flag is turned on.

  0100 + 0100 = 1000 (overflow flag is turned on)

- If the sum of two numbers with the sign bits on yields a result number with the sign bit off, the overflow flag is turned on.

  1000 + 1000 = 0000 (overflow flag is turned on)

Otherwise, the overflow flag is turned off. Examples :
0100 + 0001 = 0101 (overflow flag is turned off)
0110 + 1001 = 1111 (overflow flag is turned off)
1000 + 0001 = 1001 (overflow flag is turned off)
1100 + 1100 = 1000 (overflow flag is turned off)

5 Notes

- You may choose to do this as a GROUP assignment, with AT MOST 3 students per group. The submitted CODE should include ALL students' names in the Comments sections.

- Please fill in the group members' names along with their email addresses, before Oct. 28, 2014, 8pm at `http://goo.gl/forms/N5dPDChkBK`.
- The Academic Honor Policy must be strictly followed. The objective is to thoroughly learn the concepts taught during this semester by implementing them. The objective is NOT to implement "something" and get "some decent" grade.

  Groups should not interact with each other and/or exchange or share code. Code downloaded from the Internet (if any) or any other source (e.g., DC++, Moodle, etc.) should not be used or submitted. Any such violations will result in failing the course, followed by reporting to the Institute Committee that handles academic code violations.
- Please protect your Moodle password, your computers, notebooks, etc. If someone pilfers your intellectual property because of your carelessness, you will also be held responsible.
- NO Extensions allowed.
- A report **per group** (in pdf or handwritten) documenting relevant high-level circuit/module design.
- **What to Submit:** (a) On Moodle, submit a single tar-gzipped file, containing roll number(s) of students in the group as: CS10B805-CS10B810-CS10B900.tgz; (b) Include a brief README explaning the structure of the tar file. (c) Do not submit .bit files

6 Grading

- Verilog Implementation: 60%
- Demo and Viva Voce Exam: 40%

During the demo, you will be given a small program written in assembly using the instructions specified above. Your task is to code and execute these instructions using your Verilog program.

Individual (NOT Group-wise) Viva Voce Exam slots will be scheduled with the assigned TA from Nov. 14, 9am – Nov. 17th, 6pm.

Sample program

A sample 5-line program is shown below:

- MOVK R2, 2
- MOVK R3, 3
- ADD R1, R2, R3
- MUL R1, R2, R1
- OR R4, R2, R3