EECS 151/251A Homework 2

Due Friday , September 20th, 2019

Problem 1: Building the MIPS ALU

MIPS (Microprocessor without Interlocked Pipelined Stages) is a RISC (Reduced Instruction Set Computer) ISA (Instruction Set Architecture) developed by by several Stanford researchers in the mid 1980s. In this problem, you will be asked to implement a simplified 32 bit ALU of the MIPS processor and test its functionality.

Function Code	Operation
000	Signed Add
001	Unsigned Add
010	Signed subtract
011	Unsigned Subtract
100	And
101	Or
110	Shift Right Arithmetic
111	Shift Right Logical

(a) Write a Verilog module named alu to implement the above listed operations.

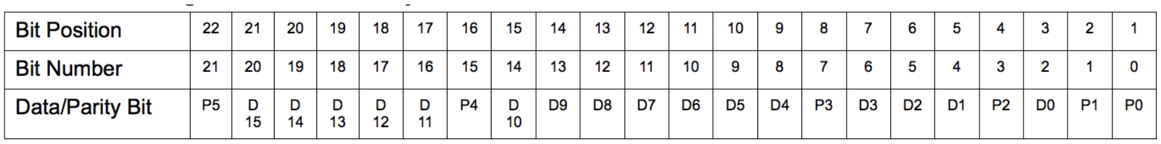
(b) Write a testbench (alu_testbench) to verify every arithmetic operation. Your testbench should check the results and output a pass/fail indicator. Make sure you show the difference between different flavors of operations (signed vs. unsigned, arithmetic vs. logical).

Problem 2: Hamming Code

Bit Error Rate (BER) is an important measure for chip designers to decide how reliable a memory block is. Therefore, it is beneficial to be able to identify and correct bit errors. Hamming code is a family of linear error correcting code that can detect up to two-bit errors or correct one-bit errors . Review the wikipedia page on Hamming Code, especially the section on SECDED, before proceeding with the problem: https://en.wikipedia.org/wiki/Hamming_code.

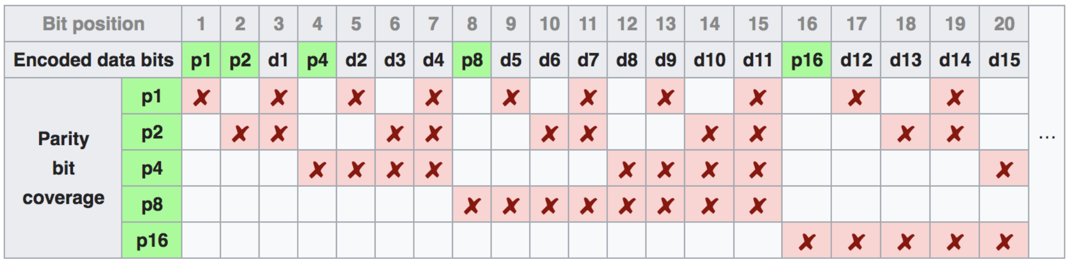
Assume there is a databus that contains 16-bit data, the data written into memory is a 22-bit data word. When the data is read back from the memory device, the stored parity bits are compared with a newly created set of parity bits from the read data. The result of this comparison, called the syndrome, will indicate the incorrect bit position in a single data error. The following table illustrates how the 16-bit data word and parity bits are stored in memory.

The parity bits P0-P4 are created for single error detection and correction by XOR-ing highlighted bits in figure 2. One additional parity bit (P5) detects double errors that are not correctable. This is calculated by XOR-ing all of the data bits and parity bits.



Bit Position	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
Bit Number	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Data/Parity Bit	P5	D 15	D 14	D 13	D 12	D 11	P4	D 10	D9	D8	D7	D6	D5	D4	P3	D3	D2	D1	P2	D0	P1	P0

Figure 1: Hamming Code Data and Parity Bits



Bit position		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	
Encoded data bits		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8	d9	d10	d11	p16	d12	d13	d14	d15	
Parity bit coverage	p1	✗		✗		✗		✗		✗		✗		✗		✗		✗		✗		
	p2		✗	✗			✗	✗			✗	✗			✗	✗			✗	✗		...
	p4				✗	✗	✗	✗					✗	✗	✗	✗					✗	
	p8								✗	✗	✗	✗	✗	✗	✗	✗						
	p16																✗	✗	✗	✗	✗	

Figure 2: Hamming Code Illustration

The syndrome is created by XOR-ing the parity bits read out of memory with the newly created set of parity bits from the data stored in memory. The value of the syndrome will indicate the bit position if a single bit error has occurred. The table below illustrates how the value of syndrome corresponds to the overall parity bit in detecting errors.

Syndrome	Overall Parity	Error type	Notes
0	0	No Error	
$\neq 0$	1	Single Error	Correctable
$\neq 0$	0	Double Error	Not correctable
0	1	Parity Error	Overall parity is in error and can be corrected

Given a 22-bit input memory data, implement the SECDED algorithm to generate a 2-bit error code that encodes the error information, and submit the following items:

(a) A table indicating how your outputs map to each error case.

(b) A hierarchical block diagram illustrating how you implemented the algorithm. Be specific about the inputs and outputs of each block, as well as the bit widths.

(c) RTL and testbench of your design. Your testbench should check the results and output a pass/fail indicator. Make sure you run enough test cases to cover each possible error condition.

Problem 3: Counters

1. Write Verilog for a 16-bit up/down counter that takes 1-bit inputs *en, up* and *reset*, and outputs a 16-bit result. Assume all inputs are synchronous. The counter must follow the following rules:

   (a) Resets to 0 when reset is 1

   (b) Only counts when enabled
   (c) Counts up when up = 1, otherwise counts down.

2. Write a testbench to verify that your counter works. Your testbench should check the results and output a pass/fail indicator.