CS450/CS650 Winter 2018 – Coding Project 1

Download p1.zip which contains counter.sv, cordic.sv, and plru.sv. Each source file contains a stub for the module that you must complete plus a testbench. p1.out shows the expected output for each testbench. The Makefile can be used to compile the source files and to simulate the testbenches.

Submit your completed counter.sv, cordic.sv, and plru.sv. "for" may only be used as part of generate statements. "while" and "repeat" loops are not permitted.

1. (3 marks) counter module

   Implement a counter that loads the count value from the `data` input on the rising clock edge when `load` is 1. On the rising clock edge, when `load` is 0, count down from the loaded value towards zero. Set `tc` (terminal count) to 1 when the count has reached zero.

2. (3 marks) cordic module

   After `go` transitions from 1 to 0, your cordic module should calculate the sine and cosine of the given input angle `a` and set `done` to 1 when finished. The angle and the sine and cosine are represented in signed fixed-point notation with 2 bits left of the decimal point and 14 bits right of the decimal point. Use the CORDIC algorithm which is as follows:

   Initial:

   $$\begin{aligned} x_0 &= 1/\prod_n \sqrt{1+2^{-2i}} \\ y_0 &= 0 \\ z_0 &= a \end{aligned}$$

   Iteration 0 to 15:

   $$\begin{aligned} x_{i+1} &= x_i - y_i * d_i * 2^{-i} \\ y_{i+1} &= y_i + x_i * d_i * 2^{-i} \\ z_{i+1} &= z_i - d_i * \operatorname{atan}(2^{-i}) \end{aligned}$$

   where

   $$d_i = -1 \text{if } z_i < 0, +1 \text{ otherwise}$$

   After 16 iterations, $x_{16} = \cos(a)$, $y_{16} = \sin(a)$.

   The precomputed values for $x_0$and $\operatorname{atan}(2^{-i})$ are provided in the module as `x_0` and `e[0:15]`. These are in fixed-point notation. If using a shift operator, be sure to use the signed shift operator ('<<<' or '>>>') and note that it has lower precedence that the binary '+' or '-' operators.

   Values of x_i, y_i, z_i that correspond to the testbench are provided in the cordic_debug.out file to help with debugging.

3. (4 marks)

Pseudo-LRU (pLRU) is a cache replacement policy. There is a short introduction to it at https://www.youtube.com/watch?v=8CjifA2yw7s. It applies to set-associative caches and once a cache set becomes full it determines which block will be removed from the set to make room for a new block. LRU is taught in second year but is expensive in terms of storage (for age registers) and energy use (updating age registers). pLRU approximates LRU using 1 'used' bit per block.

Consider for example a 2-way set-associative cache with only 2 sets. Each block has a tag to identify its location in memory, a valid flag, a used flag and storage for 16 bytes of data.

way 0

	tag	v	u	data
0				
1				

way 1

	tag	v	u	data
0				
1				

Set 0 occupies the first row of both ways and Set 1 occupies the second row of both ways. Initially all valid and used flags are set to zero.

When a CPU gives the cache controller, for example, a 16-bit address for data it wants, the cache controller breaks down the address as follows to determine if it has the data (a "hit") or must bring in the data (a "miss"):

15 … 5	4	3 … 0
Tag	index	Offset

- The number of offset bits = log2 block size (e..g log2(16)=4)
- The number of index bits = log2 number of sets (e.g. log2(2)=1)
- The number of tag bits = remaining bits (e.g. 16 – 4 – 1 = 11)

The cache controller uses the index to select a set. It then checks if any of the ways in that set is valid and has a matching tag. If so then this is a hit and the used bit is set to 1. If no match is found this is a miss and it selects the first way (starting from Way 0) in the set that has a used bit equal to 0. It replaces the data, sets the tag, and sets the valid bit and used bits to 1. In either case (hit or miss), if all used bits in the set are equal 1, then all used bits are set to 0 except the used bit that was just set.

Implement a 4-way set-associative cache that inputs a 16-bit address and outputs hit=1 if it has the data and hit=0 if it doesn't have the data. The cache doesn't actually store or return data – it just simulates the hit rate. The block size is 16 bytes so there are 4 offset bits in the address. The number of sets is configurable via the setBits parameter. On reset the valid and used flags should be cleared. Data structures are declared in the module stub for the tags, valid flags and user flags. You may configure them differently if you wish but the second monitor task (commented out) in the testbench uses them to produce the output provided in plru_debug.out which is provided to help with debugging.