Implementation of LRU Replacement Policy for Cache Memory Using Hardware FPGA

Julie C. Kim
Department of Computer Engineering
California State University
Long Beach
CSULB
julie.c.kim@student.csulb.edu

Abstract—this paper is the repeat of the implementation of two LRU algorithms in HDL code. The repeat will be to prove the efficient of Tree Based Pseudo LRU over Conventional LRU in both design complexity and resource usage [1]. Cache memory is used to transfer data to and from main memory and CPU to improve the hit rate and access time. In set associative cache memory, when a cache set is full, replacing a cache line is a way to transfer new needed data from main memory. Study has shown that LRU is the most efficient cache replacement policy when implemented with n-way set associative cache. Although with the trade-off of the increase of power consumption, as n-way set increases, LRU algorithm is the replacement policy used with the n-way set associative cache architecture. In this paper, LRU (Least Recently Used) replacement police is implemented in SystemVerilog HDL. Conventional LRU and Tree Based Pseudo LRU algorithms are translated into hardware langue as part of the cache memory architecture. Conventional LRU has larger array size of 32 bits compared to Pseudo LRU has 7 bits of array size per cache line for the total of 4096 lines of cache memory. The result of the experiment shows Tree Based Pseudo LRU is implemented in less time and resources.

Keywords—Replacement policies, LRU, PLRU, FPGA, SystemVerilog.

I. INTRODUCTION

The components of a processor are instruction memory, register files, control unit to do arithmetic, and data cache. In 5-stage pipeline CPU, the main units are instruction, execution and storage unit. The first stage of the pipeline is sending out instruction, the ALU is responsible for execution, and small on chip memory storage is cache memory; it is the interface of CPU to the off chip slow main memory. Cache memory is used to minimize the access time of data off chip, retrieving data from main memory. The performance of cache memory in a computer is determined by high hit rate and fast main memory access time. To improve hit rate, the principle locality referencing is implemented in designing cache memory. There are two locality referencing, Temporal locality and Spatial locality. Temporal locality refers to referencing in a timing manner. If a memory location is referenced, it is most likely to be referenced again and again. It happens in loop, where small subset of instructions might be executed repeatedly. Spatial locality, if a memory location is referenced, the locations with nearby address will tend to be reference soon. It happens when a block of memory address might be accessed sequentially. Temporal locality is designed by keeping the most recent accessed data closed to the processor, Spatial locality is designed by move blocks consisting of contiguous word closer to the processor. The small cache memory store data/instruction that most likely to be used in the near future, or the data populated sequentially in the main memory, aligned with principle of Temporal and Spatial locality. An example of temporal locality is the For Loop to access sequential data from the main memory.

Cache memory is significantly small compare to large main memory. Three mapping techniques, including full associative mapping, direct mapping, and set associative mapping, are used to determine which cache line a block of memory content should be written to. Set associative has the highest hit rate as associative degree increases. However, there is an asymptotic limit, and increase cache size is the solution to maintain high hit rate and less access time [3]. As memory wall problem increase, computer architecture not only increase cache level but associative degree. When every line in all cache set are full, a new block of memory needs to replace or overwritten the existing cache line/set. When designing a cache, replacement policy is import to consider to design a fast and efficient cache memory. The replacement policy make decision which line of cache memory is to replace, which involve time delay and hardware resource to store past and future reference. Replacement policies is a factor to determine the effectiveness of a high degree of set associative cache. Replacement algorithms work better with set associative cache [4].

A. Overview of Cache Replacement Policies

As level of cache and degree of associativity increase, research of a best replacement policy is a hot topic and gain more significance. Based on the principle of spatial and temporal policy, and Optimal Replacement (OPT) algorithm should be able to store past cache line that has not been use for long time and/or guess which cache line will most likely to be used in the future. It requires more hardware to store extra information about cache line was used in the past, and hardware complexity increase as associative degree increase. It is impossible to accurately guess the unknown future; the best can be is approximation approach [4]. Computer CPU uses different replacement policies such as Random

Replacement, FIFO – Fist In First Out, LFU – Least Frequently Used, and LRU – Least Recently Used.

Random replacement does not take into account of order of which cache line was used. The algorithm generate a random cache line number, and overwritten the old data with the new data from the memory when cache miss.

FIFO policy does not take into consideration the principle of temporal locality. It replaces the cache line based on the time the data was written to the cache line compared to the last data written to the cache line. The first data was written is to be replaced with new data when cache miss.

LFU policy uses a counter to remember the cache line that has a fewest number of accessed. The new data replace the cache line that has the fewest number when cache miss [1].

LRU policy uses an array to remember the least recently accessed cache line (a cache line in a set that is not reference for the longest time). It is easy to know which cache line is the most recently use and will most likely to be reference a gain. The least recently used cache line will be replaced by new data when cache miss.

B. LRU – Least Recently Used Replacement Policy

Among all the replacement policy, LRU is the most effective algorithm because it is easy to implement. In modern process with high degree of set associative, Intel x86 and AMD processor use LRU in 16-way to 64-way set associative. LRU needs a number of status bits to maintain record of each cache line accessed [3]. When degree of associativity increase, the need of number of status bits also increase, so do the hardware resources increase. The LRU stack is maintain as below figure.

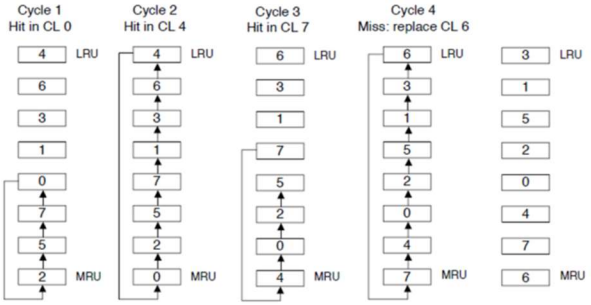


Fig. 1. An Illustration of LRU Mechanism

Fig. 1. Shows the Conventional LRU, which is quite easy to implement, but needs many number of bits to store history information. It store past reference and predict the future most likely will be used again based on the principle of temporal locality. The cost and resource of hardware increases as LRU array gets larger. The algorithm shift the status bits when both cache miss and hit to update the status of cache line reference; the operation requires time and power. It is a linear relationship between time, energy consumption and high degree of associativity.

Tree Base Pseudo LRU serves the same purpose of Conventional LRU, but reduces the amount of hardware resource usage, time and energy significantly. PLRU (Pseudo LRU) is a heuristics replacement algorithm. It approximates and find the approximate least recently used cache line. The cache line to be replace is not exactly the least recently use (not exactly the cache line that is not referenced for the longest time), but the cache line that needs to be replace is not the most likely will be reference again in the near future [6]. PLRU uses 1 bit to store status of one way in a cache set. The time to search is half less compared to the Conventional LRU, so the energy consumption is definitely more than half lesser than Conventional LRU.

Fig. 2. Shows how Bits work as a binary tree of 1-bit pointers that point to the less recently used subtree. Following the pointer chain to the leaf node identifies the replacement candidate. Upon an access all pointers in the chain from the accessed ways leaf node to the root node are set to point to subtree that does not contain the accessed way [5].

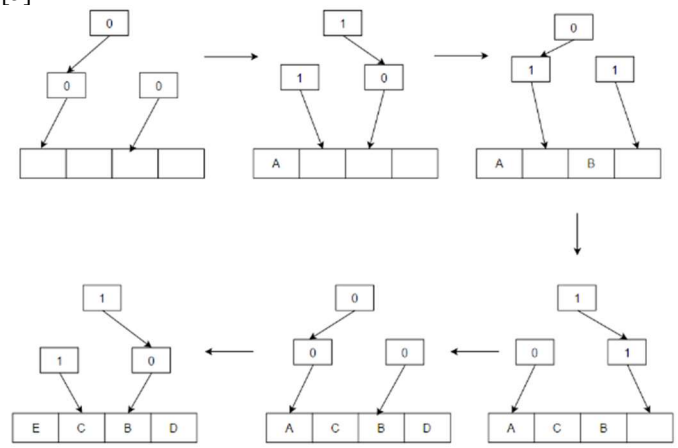


Fig. 2. Pseudo LRU access sequence A B C D E

II. CACHE MEMORY AND PIPE LINE PROCESSOR

The cache implementation in this paper is to use 5-stage pipeline MIPS CPU. The five stages are Instruction fetch, Instruction decode, Execution, Memory read, and Write back to 64-bits register files. Between each stage, there is a pipeline register to hold the instruction bits to pass down the stream. The number of bits decreases in the last pipeline stage as lesser decoded instructions need to be passed down. The pipeline registers are IF_IF_reg hold 300-bits, ID_EX_reg has 280-bits, Ex_Mem_reg hold 200-bits, and MEM_WB_reg hold 127-bits. In IF stage, instruction is fetch; ID stage, instruction is decoded; EX stage, ALU execution of arithmetic operation; MEM stage, load data from cache or main memory; WB stage, write the new data value to designated register files. The paper only implement data cache, which is 128-bits. In execution stage, data is loaded from the cache if present. If not data needs to be loaded from main memory. To accommodate the 64-bits word of register file, the 256 million lines main memory is designed to have two 64-bits words with the total of 128-bits for each line of memory. The total memory size is 4G. Data cache lines are design accordingly, each line consists of two 8-byte words. The lower 8-byte is word 0 (word_0), and the upper 8-byte is word 1 (word_1). For maximum design (this paper only) is 8 ways for data cache with the size of 512KB.

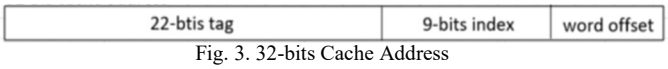


Fig. 3. 32-bits Cache Address

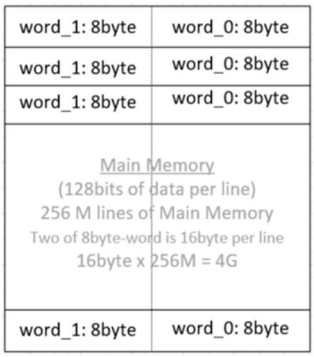


Fig. 4. 128bits Main Memory

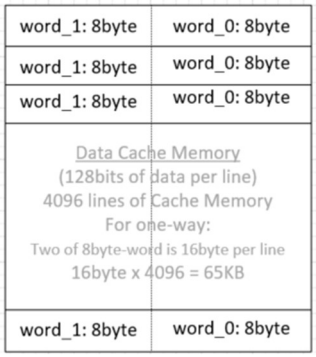


Fig. 5. 128bits Data Cache

There are 32 addresses in the 64-bits register file. It stores memory addresses offset or data cache addressd offeset to read from or write to, as well as data to do the arithmetic operation in ALU, the result of ALU or loaded data to be written back to the register file. Each MIPS instruction contains op code as command to do arithmethic operation or load from the memory, and address of register file to get data or address of data cache memory, and address to store the calculation result and loaded data from the data cache . The 128-bits data cache address contains 22-bits tage for comparing valid address in the data cache, 9-bits index to indicate the set number of the entire data cache line with 512 sets for 8 ways associativity. The word offset 0 indicate the lsb 8-byte word_0, and 1 indicate the msb 8-byte word_1.

III. IMPLEMENTATION OF LRU

In the 5-stage pipeline architecture, LRU controller is implemented as a combination of three 2-D arrays. There are 4096-lines of data array called data cache line, 4096-lines of tag array (store 22-bits tag), and 4096-line of LRU array. Each line of the arrays can accommodate up to 8 number of ways [1].

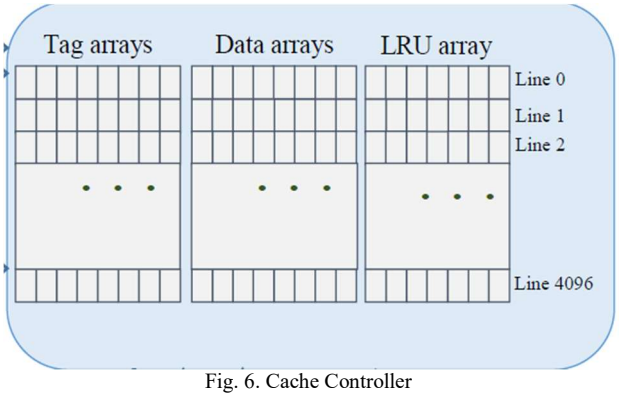


Fig. 6. Cache Controller

A. Conventional LRU

Conventional LRU replacement policy is implemented by using hexadecimal digits for each cache memory line as the maximum degree of associativity which can be selected. This makes the LRU array size 4096 lines each line have 32 bits in length and each line is implemented with initial value $(76543210)_H$ as shown in Fig. 7 [1].

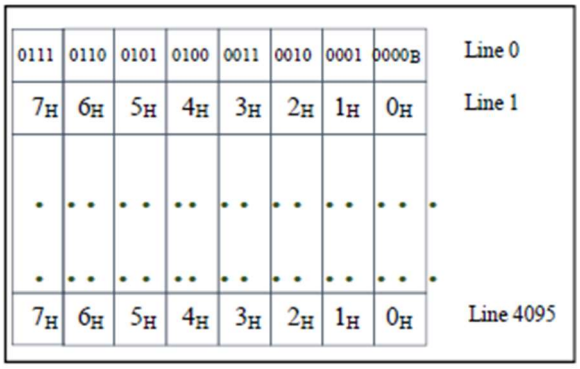


Fig. 7. LRU array of Conventional LRU design.

In this case there is only one unit for LRU array which is responsible for updating the LRU array's value in each cache memory access. In case of cache first miss occurs, the unit reads the 4 least significant bits which represents the first hexadecimal digit of LRU array for corresponding index of requested address and its initial value is 0H, the value of this digit is indicating which way have the least recently used block to be replaced. After writing the incoming block in the replaced block, these 4 least significant bits will be rotated to be 4 most significant bits and all other bits will be shifted right 4 bits, as shown in Fig. 8. [1]

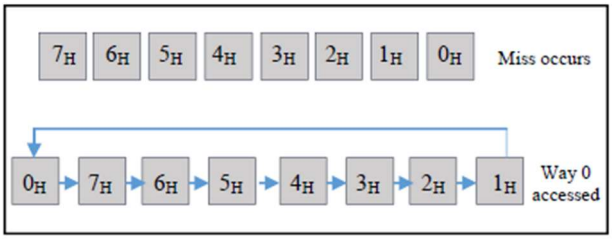


Fig. 8. First miss occurs

In case of second miss occurs, the 4 least significant bits now equal 1H, therefor, way 1 have the block which will be replaced and also the 4 least significant bits will be rotated to be 4 most significant bits and all other bits will be shifted right 4 bits, and the same process when third miss occurs. This procedure is continue when each new miss occurs. After the third miss occurs, if a his occurs with block in way_0, the LRU unit will search for the 4 bits carrying the value 0H, and rotate it to be 4 most significant bits and then shift right all others bits, as shown in Fig. 9 [1].

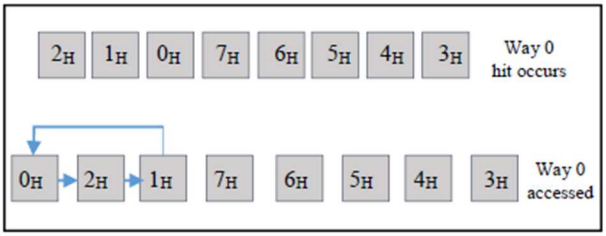
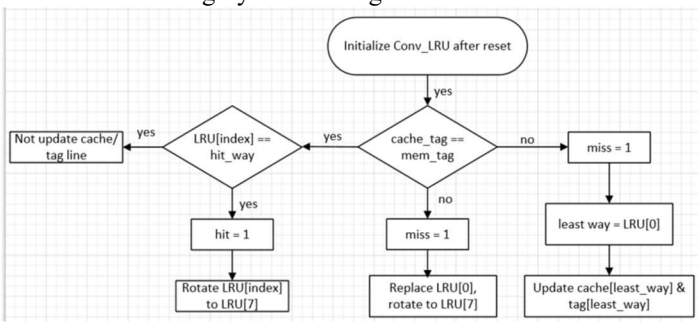


Fig. 9. Way_0 hit occurs

Following the flow chart below, Conventional LRU HDL is coded using SystemVerilog.



1. Initialize conv_lru array:

```
if (reset) begin
   for (shortint i=0; i<512; i++) begin
      for (shortint j=0; j<8; j++) begin
         lru_arr[i*8+j][j] <= j;
         for (shortint n0=0; n0<8; n0++) begin
            if (n0 != j) begin
               lru_arr[i*8+j][n0] <= '0;
            end
         end
      end
   end
end
```

2. Comparing tag and check for array index if cache hit:

```
// compare tag-array
for(shortint i=0; i<8; i=i+1)
begin
   //tag-index is corresponse to way_num of tag-array
   if (tag22bits[set_num_9bits*8+i][i] == mem28bits) begin
      hit      = 1'b1;
      //tag-index is corresponse to way_num of tag-array
      hit_way  = i;
      // find the lru index that is the hit way#
      for (shortint ih=0; ih<8; ih++) begin
         if (lru_arr[lru_set_num*8+ih][ih] == hit_way) begin
            lhit_idx =  ih;
         end
         break;
      end

      mru_way   = lru_arr[lru_set_num*8+7][7];
      way_num   = hit_way;
      cache_set = set_num_9bits;      //  2^5=512
      break;
   end
   else begin
      hit = 1'b0;
   end
end // end for loop
```

3. Update conv_lru array in miss or hit case:

```
else begin
   if (miss) begin // miss and hit is encoded inside mem_rd
      for (shortint i=lmiss_idx; i<7; i++) begin
         lru_arr[lru_set_num*8+7][7]  <= lest_use_way;
         lru_arr[lru_set_num*8+i][i]  <= lru_arr[lru_set_num*8+(i+1)][i+1];
      end
   end
   else if (hit) begin
      for (shortint i=lhit_idx; i<7; i++) begin
         lru_arr[lru_set_num*8+7][7]  <= hit_way;
         lru_arr[lru_set_num*8+i][i]  <= lru_arr[lru_set_num*8+(i+1)][i+1];
      end
   end
end
```

4. Update data cache / tag line if miss:

```
always_ff @(posedge clk)
begin
   if (reset) begin
      for (shortint i=0; i<4096; i++) begin
         for (shortint j=0; j<8; j++) begin
            cache_dat128_line[i][j] <= '0;
         end
      end
      for (shortint i=0; i<4096; i++) begin
         for (shortint j=0; j<8; j++) begin
            tag22bits[i][j] <= '0;
         end
      end
   end
   else begin
      if (miss) begin
         cache_dat128_line[cache_set*8+way_num][way_num] <= i_rddat128bits_mainmem[127:0]
         tag22bits[cache_set*8+way_num][way_num]         <= mem_addr;
      end
   end
end
```

B. Tree Based Pseudo LRU

Tree-based pseudo LRU technique required an algorithm to determine which block of each cache memory set is to be replaced. The algorithm requires number of bits for each cache memory line determined by the following equation:

LRU bits = Degree of associativity – 1 (1)

For example, if the cache is 2-way set associativity, then the number of LRU bits is (1), for 4-way set associative the number of LRU bits is (3), and for 8-way set associative the number of LRU bits is (7). Fig. 10. Shows 3-bits algorithm for 4-way set associative cache memory [1].

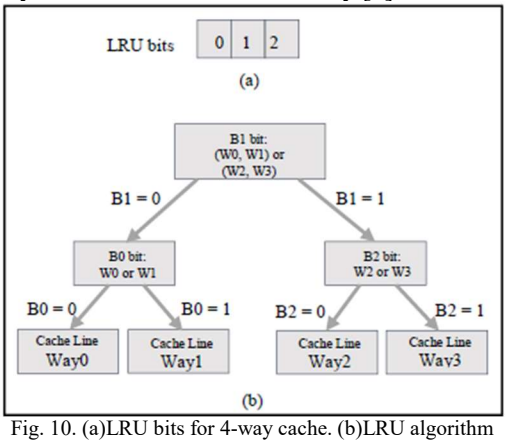


Fig. 10. (a)LRU bits for 4-way cache. (b)LRU algorithm

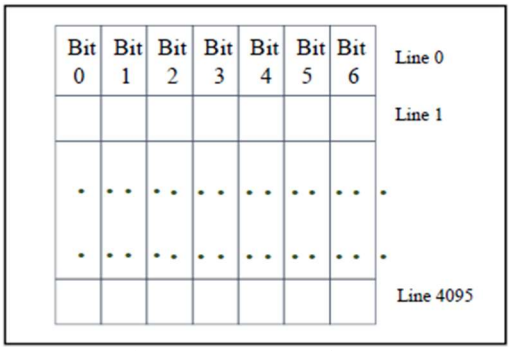


Fig. 11. Tree Base Pseudo LRU

Each line in the LRU array corresponds to line has the same index value in cache memory set arrays. This design requires a LRU controller unit in addition to LRU array to manage the LRU array and updating the LRU bits in each cache access [1].

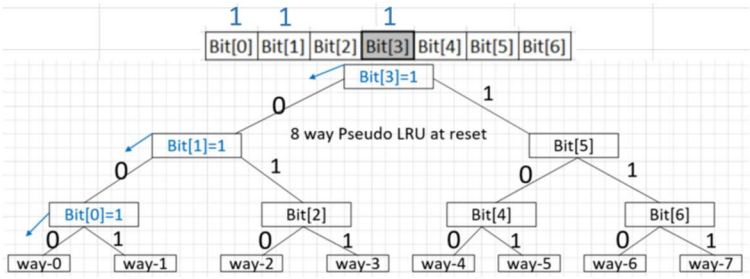
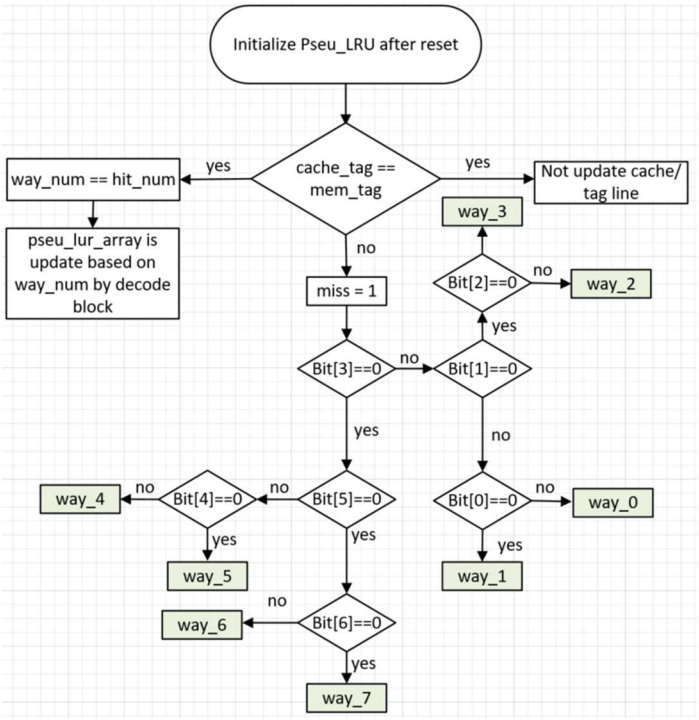


Fig. 12. Pseudo LRU bits for 8-ways cache and algorithm

The algorithm has only 7 bit for each line of 8-ways set associative data cache. After reset, the Pseudo_lur is initialized to (0001011) as shown in Fig. 12 to store data in cache line starting at way_0. When cache miss, the algorithm first search at Bit[3], "1" direct to search at Bit[5], "0" direct to search at Bit[1]. The idea is to find the least recently used way at the opposite side of the last used way number of the set. When cache hit, the Pseudo_lru controller update the array based on the way number indicated as hit. For example, if way-3 is a hit, the controller will update Bit[3]=0, Bit[1]=1, Bit[2]=1, the remaining bit retain the same value, therefor, the array at a line is (-- -- -- 0 1 1 --). If the second hit is way_0, the controller will update Bit[3]=0, Bit[1]=0, Bit[0]=0, the remaining bit retain the same value as (-- -- -- 0 1 0 0). If the third hit is way-7, Bit[3]=1, Bit[5]=1, Bit[6]=1, the remaining bit retain the same value as (1 1 – 1 1 0 0).

Following the flow chart below, Tree Based LRU HDL is coded using SystemVerilog.



1. Initialize the Tree Base Pseudo LRU array:

```
always_ff @(posedge clk)
begin
   if (reset)
   begin
      for (shortint i=0; i<512; i++) begin
         for (shortint j=0; j<8; j++) begin
            //-----------------------------------------
            // Pseido-lru 1bit-2d-array store initial way to us
            // way-0 is set to be the initial way
            //-----------------------------------------
            if (j!=0 && j!=1 && j!=3) begin
               pse_arr[i*8+j][j] <= 1'b0;
            end
            if (j==0 || j==1 || j==3) begin
               pse_arr[i*8+j][j] <= 1'b1;
            end

         end
      end
   end
```

2. Comparing the tag and check for hit way number:

```
if (mem_rd)
begin
   mem28bits        =  mem_addr[27:0];
   set_num_9bits    =  mem28bits % 512;
   pse_set_num      =  set_num_9bits;

   //--------------------
   // compare tag-array
   ///-------------------
   for(shortint i=0; i<8; i=i+1)
   begin
      //tag-index (0-7) corresponse to way_num# of tag-array[0:4095][0:7]
      if (tag22bits[set_num_9bits*8+i][i] == mem28bits) begin
         hit      = 1'b1;
         hit_way  = i;

         // the tag-line/cache-line = pse_set_num*8+way_num
         way_num   = hit_way;       //3bits 0-7
         cache_set = set_num_9bits; //2^5=512
         break;
      end
      else begin
         hit = 1'b0;
      end
   end // end for loop
```

3. Update Pseu_lru array if hit based on way number found:

```
else if (!reset)
begin
   //----------------------------------------
   // Pseido-lru 1bit-2d-array
   // store the currently used way_num#
   //----------------------------------------
   if (way_num==3'd0) begin
      pse_arr[set_num_9bits*8+ 3][3] <= 1'b0;
      pse_arr[set_num_9bits*8+ 1][1] <= 1'b0;
      pse_arr[set_num_9bits*8+ 0][0] <= 1'b0;
   end
   else if (way_num==3'd1) begin
      pse_arr[set_num_9bits*8+ 3][3] <= 1'b0;
      pse_arr[set_num_9bits*8+ 1][1] <= 1'b0;
      pse_arr[set_num_9bits*8+ 0][0] <= 1'b1;
   end
   else if (way_num==3'd2) begin
      pse_arr[set_num_9bits*8+ 3][3] <= 1'b0;
      pse_arr[set_num_9bits*8+ 1][1] <= 1'b1;
      pse_arr[set_num_9bits*8+ 2][2] <= 1'b0;
   end
   else if (way_num==3'd3) begin
      pse_arr[set_num_9bits*8+ 3][3] <= 1'b0;
      pse_arr[set_num_9bits*8+ 1][1] <= 1'b1;
      pse_arr[set_num_9bits*8+ 2][2] <= 1'b1;
   end
   else if (way_num==3'd4) begin
      pse_arr[set_num_9bits*8+ 3][3] <= 1'b1;
      pse_arr[set_num_9bits*8+ 5][5] <= 1'b0;
      pse_arr[set_num_9bits*8+ 4][4] <= 1'b0;
   end
      else if (way_num==3'd5) begin
         pse_arr[set_num_9bits*8+ 3][3] <= 1'b1;
         pse_arr[set_num_9bits*8+ 5][5] <= 1'b0;
         pse_arr[set_num_9bits*8+ 4][4] <= 1'b1;
      end
      else if (way_num==3'd6) begin
         pse_arr[set_num_9bits*8+ 3][3] <= 1'b1;
         pse_arr[set_num_9bits*8+ 5][5] <= 1'b1;
         pse_arr[set_num_9bits*8+ 6][6] <= 1'b0;
      end
      else if (way_num==3'd7) begin
         pse_arr[set_num_9bits*8+ 3][3] <= 1'b1;
         pse_arr[set_num_9bits*8+ 5][5] <= 1'b1;
         pse_arr[set_num_9bits*8+ 6][6] <= 1'b1;
      end
   end
end
```

4. Pseudo algorithm when miss to find approximate least recently used way number in a set to be replaced:

```
//-----------------------------------------------------
// pseudo-lru algorithm
// replacement policy (the least recently used way_num#)
//-----------------------------------------------------
if (miss) begin
   cache_set = set_num_9bits; //2^5=512
   if (pse_arr[set_num_9bits*8+3][3] == 1'b0) begin            //Bit[3]==0
      if (pse_arr[set_num_9bits*8+5][5] == 1'b0) begin         //Bit[5]==0
         if (pse_arr[set_num_9bits*8+6][6] == 1'b0) begin      //Bit[6]==0
            way_num = 3'd7;
         end
         else if (pse_arr[set_num_9bits*8+6][6] == 1'b1) begin //Bit[6]==1
            way_num = 3'd6;
         end
      end
      else if (pse_arr[set_num_9bits*8+5][5] == 1'b1) begin    //Bit[5]==1
         if(pse_arr[set_num_9bits*8+4][4] == 1'b0) begin       //Bit[4]==0
            way_num = 3'd5;
         end
         else if(pse_arr[set_num_9bits*8+4][4] == 1'b1) begin  //Bit[4]==1
            way_num = 3'd4;
         end
      end
   end
   else if (pse_arr[set_num_9bits*8+3][3] == 1'b1) begin       //Bit[3]==1
      if (pse_arr[set_num_9bits*8+1][1] == 1'b0) begin         //Bit[1]==0
         if (pse_arr[set_num_9bits*8+2][2] == 1'b0) begin      //Bit[2]==0
            way_num = 3'd3;
         end
         else if (pse_arr[set_num_9bits*8+2][2] == 1'b1) begin //Bit[2]==1
            way_num = 3'd2;
         end
      end
      else if (pse_arr[set_num_9bits*8+1][1] == 1'b1) begin    //Bit[1]==1
         if (pse_arr[set_num_9bits*8+0][0] == 1'b0) begin      //Bit[0]==0
            way_num = 3'd1;
         end
         else if (pse_arr[set_num_9bits*8+0][0] == 1'b1) begin //Bit[0]==1
            way_num = 3'd0;
         end
      end
   end
end // miss: pseudo-lru algorith
```

IV. EXPERIMENTAL METHODOLOGY

The top level design consist of 5-stage pipeline CPU, 8 way set associative data cache, cache controller, and 256 million lines of main memory. For the purpose experiment only, the access time of main memory compared to data cache is assumed to be the same. It is not feasible for the simulation tool to run with 4G of memory space. Initialization of all the 256M line of memory is required, but only 168 lines of memory that are initialized with value and to be use in the simulation and implementation. There are 9 sets (set-204, set-205, set-207, set-208, set-209, set-211, set-500, set-509, and set-511). A thorough calculation needs to be done to fine random memory lines that mapped to the above sets in order to do first load (initial miss), hit and miss again after all the cache lines are filled as shown in the excel spreadsheet below:

Table 1. Choosing main memory address, intention to be cache miss (compulsory miss)

line#	mem addr	cache line	set num	ways	mod	cal-line#	
1632	40963788	1,632	204	0	204	1,632	FIRST MISS
1633	40964300	1,633		1	204		
1634	40964812	1,634		2	204		
1635	40965324	1,635		3	204		
1636	40965836	1,636		4	204		
1637	40966348	1,637		5	204		
1638	40966860	1,638		6	204		
1639	40967372	1,639		7	204		

Table 2. Choose main memory address, intention to be cache hit after cache line is initially filled up

line#	mem addr	cache line	set num	ways	mod	cal-line#	
4072	241509885	4,072	509	0	509	4,072	HIT
4073	241510397	4,073		1	509		
4074	241511421	4,074		2	509		
4075	241512957	4,075		3	509		
4076	241515005	4,076		4	509		
4077	241517565	4,077		5	509		
4078	241520637	4,078		6	509		
4079	241524221	4,079		7	509		

Table 3. Choose main memory address, intention to be 2nd miss after all cache line is filled up

line#	mem addr	cache line	set num	ways	mod	cal-line#	
4000	218500596	4,000	500	0	500	4,000	MISS
4001	218501108	4,001		1	500		
4002	218502132	4,002		2	500		
4003	218503668	4,003		3	500		
4004	218505716	4,004		4	500		
4005	218508276	4,005		5	500		
4006	218511348	4,006		6	500		
4007	218514932	4,007		7	500		

The rest of the 120 lines of addresses are calculated the same way and initialize the value in the main memory module. Each line of the memory is 128-bits. The MSB 64-bit is word_1, and the LSB 64-bit is word_0. The offset bit of cache address determine the word to be chosen. For the purpose of convenience in verifying correct data is loaded properly from the cache line or main memory, each 64-bit word is decoded in a format as:

Word_1 = {20'hbbbb_b, 12bit of set#, 16bit-set#, 16bit-way#};

Word_0 = {20'haaaa_a, 12bit of set#, 16bit-set#, 16bit-way#};

When data values are written back to the register file, the way number (16 bit LSB of each 64-bits word) is used verify which way number is a miss or hit. When write back to the register file, the value is written in ascending order according to the instruction of loading data first, second accordingly. Below is the example of main memory data value that has been hardcoded according to the spreadsheet address calculation and word format. The first 8 lines of memory matches table 1, and the second 8 lines of memory matches table 2 accordingly.

```
//8 : 12'd-cache_line, 16'd-set# (set509)
mem128bits[241509885] <= {20'hbbbb_b, 12'd4072, 16'd509, 16'h0, 20'haaaa_a, 12'd4072, 16'd509, 16'h0};
mem128bits[241510397] <= {20'hbbbb_b, 12'd4073, 16'd509, 16'h1, 20'haaaa_a, 12'd4073, 16'd509, 16'h1};
mem128bits[241511421] <= {20'hbbbb_b, 12'd4074, 16'd509, 16'h2, 20'haaaa_a, 12'd4074, 16'd509, 16'h2};
mem128bits[241512957] <= {20'hbbbb_b, 12'd4075, 16'd509, 16'h3, 20'haaaa_a, 12'd4075, 16'd509, 16'h3};
mem128bits[241515005] <= {20'hbbbb_b, 12'd4076, 16'd509, 16'h4, 20'haaaa_a, 12'd4076, 16'd509, 16'h4};
mem128bits[241517565] <= {20'hbbbb_b, 12'd4077, 16'd509, 16'h5, 20'haaaa_a, 12'd4077, 16'd509, 16'h5};
mem128bits[241520637] <= {20'hbbbb_b, 12'd4078, 16'd509, 16'h6, 20'haaaa_a, 12'd4078, 16'd509, 16'h6};
mem128bits[241524221] <= {20'hbbbb_b, 12'd4079, 16'd509, 16'h7, 20'haaaa_a, 12'd4079, 16'd509, 16'h7};
//10 : 12'd-cache_line, 16'd-set# (set500)  // Miss
mem128bits[218500596] <= {20'hbbbb_b, 12'd4000, 16'd500, 16'h0, 20'haaaa_a, 12'd4000, 16'd500, 16'h0};
mem128bits[218501108] <= {20'hbbbb_b, 12'd4001, 16'd500, 16'h1, 20'haaaa_a, 12'd4001, 16'd500, 16'h1};
mem128bits[218502132] <= {20'hbbbb_b, 12'd4002, 16'd500, 16'h2, 20'haaaa_a, 12'd4002, 16'd500, 16'h2};
mem128bits[218503668] <= {20'hbbbb_b, 12'd4003, 16'd500, 16'h3, 20'haaaa_a, 12'd4003, 16'd500, 16'h3};
mem128bits[218505716] <= {20'hbbbb_b, 12'd4004, 16'd500, 16'h4, 20'haaaa_a, 12'd4004, 16'd500, 16'h4};
mem128bits[218508276] <= {20'hbbbb_b, 12'd4005, 16'd500, 16'h5, 20'haaaa_a, 12'd4005, 16'd500, 16'h5};
mem128bits[218511348] <= {20'hbbbb_b, 12'd4006, 16'd500, 16'h6, 20'haaaa_a, 12'd4006, 16'd500, 16'h6};
mem128bits[218514932] <= {20'hbbbb_b, 12'd4007, 16'd500, 16'h7, 20'haaaa_a, 12'd4007, 16'd500, 16'h7};
```

A. Simulation Set Up and Results

Number CPU = 1 (5 stage pipeline)
Main Memory size = 4G (only use 168 lines for experiment)
Number of Data Cache sets = 512 sets
Data Cache block size = 64KB
Mapping = 8 way set associative
Replacement policy = Conventional LRU, Pseudo LRU
First Miss Instructions = 72

First Hit Instructions = 24
Second Miss Instructions = 24

The test bench is written to simulate initial miss for 9 set of data cache line (204, 205, 207, 208, 209, 211, 500, 509, 511), the first hit for set-500, set-509, and set-511), another miss for set-500, set-509, and set-511. For each set, during initial miss the data is loaded sequential in ascending order of the memory address line, and loaded to register file according to ascending order as well. The first hit is for set-500, 509 and 511 with shuffled hit-way number order, indicating by the instruction memory. The total number of Instructions loading into pipeline CPU is 120.

Example of instruction first miss, first hit and second miss for set-204, set-205, set-500, set-511, and set509 are below:

First miss:

```
//-------------------------------------------------
//  32'b11111000010_0010010000_00_00000_01010;  80
//-------------------------------------------------
//5 : 12'd-cache_line, 16'd-set# (set204)
InstructionMem1.mem29msb[1] <= {1'b1, 28'd40963788};   //40963788
InstructionMem1.mem29msb[2] <= {1'b1, 28'd40964300};   //40964300
InstructionMem1.mem29msb[3] <= {1'b1, 28'd40964812};   //40964812
InstructionMem1.mem29msb[4] <= {1'b1, 28'd40965324};   //40965324
InstructionMem1.mem29msb[5] <= {1'b1, 28'd40965836};   //40965836
InstructionMem1.mem29msb[6] <= {1'b1, 28'd40966348};   //40966348
InstructionMem1.mem29msb[7] <= {1'b1, 28'd40966860};   //40966860
InstructionMem1.mem29msb[8] <= {1'b1, 28'd40967372};   //40967372
InstructionMem1.RF[1]       <= 32'b11111000010_000000000_00_00001_00001;
InstructionMem1.RF[2]       <= 32'b11111000010_000000000_00_00001_00010;
InstructionMem1.RF[3]       <= 32'b11111000010_000000000_00_00001_00011;
InstructionMem1.RF[4]       <= 32'b11111000010_000000000_00_00001_00100;
InstructionMem1.RF[5]       <= 32'b11111000010_000000000_00_00001_00101;
InstructionMem1.RF[6]       <= 32'b11111000010_000000000_00_00001_00110;
InstructionMem1.RF[7]       <= 32'b11111000010_000000000_00_00001_00111;
InstructionMem1.RF[8]       <= 32'b11111000010_000000000_00_00001_01000;
//6 : 12'd-cache_line, 16'd-set# (set205)
InstructionMem1.mem29msb[9]   <= {1'b1, 28'd61444301};   //61444301
InstructionMem1.mem29msb[10]  <= {1'b0, 28'd61444813};   //61444813
InstructionMem1.mem29msb[11]  <= {1'b1, 28'd61445325};   //61445325
InstructionMem1.mem29msb[12]  <= {1'b0, 28'd61445837};   //61445837
InstructionMem1.mem29msb[13]  <= {1'b1, 28'd61446349};   //61446349
InstructionMem1.mem29msb[14]  <= {1'b0, 28'd61448909};   //61448909
InstructionMem1.mem29msb[15]  <= {1'b1, 28'd61449421};   //61449421
InstructionMem1.mem29msb[16]  <= {1'b0, 28'd61449933};   //61449933
InstructionMem1.RF[9]       <= 32'b11111000010_000000000_00_00001_01001;
InstructionMem1.RF[10]      <= 32'b11111000010_000000000_00_00001_01010;
InstructionMem1.RF[11]      <= 32'b11111000010_000000000_00_00001_01011;
InstructionMem1.RF[12]      <= 32'b11111000010_000000000_00_00001_01100;
InstructionMem1.RF[13]      <= 32'b11111000010_000000000_00_00001_01101;
InstructionMem1.RF[14]      <= 32'b11111000010_000000000_00_00001_01110;
InstructionMem1.RF[15]      <= 32'b11111000010_000000000_00_00001_01111;
InstructionMem1.RF[16]      <= 32'b11111000010_000000000_00_00001_10000;
```

First hit:

```
// Instruction for cache hit
//2 : 12'd-cache_line, 16'd-set# (set500)
InstructionMem1.mem29msb[73] <= {1'b1, 28'd141515252}; //7-141515252   //141500916 :0
InstructionMem1.mem29msb[74] <= {1'b1, 28'd141511668}; //6-141511668   //141501428 :1
InstructionMem1.mem29msb[75] <= {1'b1, 28'd141508596}; //5-141508596   //141502452 :2
InstructionMem1.mem29msb[76] <= {1'b1, 28'd141506036}; //4-141506036   //141503988 :3
InstructionMem1.mem29msb[77] <= {1'b1, 28'd141503988}; //3-141503988   //141506036 :4
InstructionMem1.mem29msb[78] <= {1'b1, 28'd141502452}; //2-141502452   //141508596 :5
InstructionMem1.mem29msb[79] <= {1'b1, 28'd141501428}; //1-141501428   //141511668 :6
InstructionMem1.mem29msb[80] <= {1'b1, 28'd141500916}; //0-141500916   //141515252 :7
InstructionMem1.RF[73]       <= 32'b11111000010_000000000_00_00001_00001;
InstructionMem1.RF[74]       <= 32'b11111000010_000000000_00_00001_00010;
InstructionMem1.RF[75]       <= 32'b11111000010_000000000_00_00001_00011;
InstructionMem1.RF[76]       <= 32'b11111000010_000000000_00_00001_00100;
InstructionMem1.RF[77]       <= 32'b11111000010_000000000_00_00001_00101;
InstructionMem1.RF[78]       <= 32'b11111000010_000000000_00_00001_00110;
InstructionMem1.RF[79]       <= 32'b11111000010_000000000_00_00001_00111;
InstructionMem1.RF[80]       <= 32'b11111000010_000000000_00_00001_01000;
//8 : 12'd-cache_line, 16'd-set# (set509)
InstructionMem1.mem29msb[81]  <= {1'b1, 28'd241517565}; //5-241517565  //241509885  :0
InstructionMem1.mem29msb[82]  <= {1'b0, 28'd241520637}; //6-241520637  //241510397  :1
InstructionMem1.mem29msb[83]  <= {1'b1, 28'd241524221}; //7-241524221  //241511421  :2
InstructionMem1.mem29msb[84]  <= {1'b0, 28'd241511421}; //2-241511421  //241512957  :3
InstructionMem1.mem29msb[85]  <= {1'b1, 28'd241512957}; //3-241512957  //241515005  :4
InstructionMem1.mem29msb[86]  <= {1'b0, 28'd241515005}; //4-241515005  //241517565  :5
InstructionMem1.mem29msb[87]  <= {1'b1, 28'd241509885}; //0-241509885  //241520637  :6
InstructionMem1.mem29msb[88]  <= {1'b0, 28'd241510397}; //1-241510397  //241524221  :7
InstructionMem1.RF[81]        <= 32'b11111000010_000000000_00_00001_01001;
InstructionMem1.RF[82]        <= 32'b11111000010_000000000_00_00001_01010;
InstructionMem1.RF[83]        <= 32'b11111000010_000000000_00_00001_01011;
InstructionMem1.RF[84]        <= 32'b11111000010_000000000_00_00001_01100;
InstructionMem1.RF[85]        <= 32'b11111000010_000000000_00_00001_01101;
InstructionMem1.RF[86]        <= 32'b11111000010_000000000_00_00001_01110;
InstructionMem1.RF[87]        <= 32'b11111000010_000000000_00_00001_01111;
InstructionMem1.RF[88]        <= 32'b11111000010_000000000_00_00001_10000;
```

```
//9 : 12'd-cache_line, 16'd-set# (set511)
InstructionMem1.mem29msb[89]  <= {1'b1, 28'd253516799}; //4-253516799 //253511679 :0
InstructionMem1.mem29msb[90]  <= {1'b0, 28'd253514751}; //3-253514751 //253512191 :1
InstructionMem1.mem29msb[91]  <= {1'b1, 28'd253513215}; //2-253513215 //253513215 :2
InstructionMem1.mem29msb[92]  <= {1'b0, 28'd253526015}; //7-253526015 //253514751 :3
InstructionMem1.mem29msb[93]  <= {1'b1, 28'd253522431}; //6-253522431 //253516799 :4
InstructionMem1.mem29msb[94]  <= {1'b0, 28'd253519359}; //5-253519359 //253519359 :5
InstructionMem1.mem29msb[95]  <= {1'b1, 28'd253511679}; //0-253511679 //253522431 :6
InstructionMem1.mem29msb[96]  <= {1'b0, 28'd253512191}; //1-253512191 //253526015 :7
InstructionMem1.RF[89]        <= 32'b11111000010_000000000_00_00000_10001;
InstructionMem1.RF[90]        <= 32'b11111000010_000000000_00_00000_10010;
InstructionMem1.RF[91]        <= 32'b11111000010_000000000_00_00000_10011;
InstructionMem1.RF[92]        <= 32'b11111000010_000000000_00_00000_10100;
InstructionMem1.RF[93]        <= 32'b11111000010_000000000_00_00000_10101;
InstructionMem1.RF[94]        <= 32'b11111000010_000000000_00_00000_10110;
InstructionMem1.RF[95]        <= 32'b11111000010_000000000_00_00000_10111;
InstructionMem1.RF[96]        <= 32'b11111000010_000000000_00_00000_11000;
```

Second miss: new data is loaded into cache for set-500, set509, and set-511 (shown only set-511)

```
//9 : 12'd-cache_line, 16'd-set# (set511)
InstructionMem1.mem29msb[114]  <= {1'b1, 28'd249511935};  //249511935
InstructionMem1.mem29msb[115]  <= {1'b0, 28'd249512447};  //249512447
InstructionMem1.mem29msb[116]  <= {1'b1, 28'd249513471};  //249513471
InstructionMem1.mem29msb[117]  <= {1'b0, 28'd249515007};  //249515007
InstructionMem1.mem29msb[118]  <= {1'b1, 28'd249517055};  //249517055
InstructionMem1.mem29msb[119]  <= {1'b0, 28'd249519615};  //249519615
InstructionMem1.mem29msb[120]  <= {1'b1, 28'd249522687};  //249522687
InstructionMem1.mem29msb[121]  <= {1'b0, 28'd249526271};  //249526271
InstructionMem1.RF[114]        <= 32'b11111000010_000000000_00_00000_10001;
InstructionMem1.RF[115]        <= 32'b11111000010_000000000_00_00000_10010;
InstructionMem1.RF[116]        <= 32'b11111000010_000000000_00_00000_10011;
InstructionMem1.RF[117]        <= 32'b11111000010_000000000_00_00000_10100;
InstructionMem1.RF[118]        <= 32'b11111000010_000000000_00_00000_10101;
InstructionMem1.RF[119]        <= 32'b11111000010_000000000_00_00000_10110;
InstructionMem1.RF[120]        <= 32'b11111000010_000000000_00_00000_10111;
InstructionMem1.RF[121]        <= 32'b11111000010_000000000_00_00000_11000;
```

1. *Conventional LRU load in data and expected result:*

When filling up the cache line after the first miss, register file with address of 1 to 24 is expected to have value from main memory in ascending order.

```
Reg   1= bbbbbfa0_01f40000   Reg   9= bbbbbfe0_01fd0000   Reg  17= bbbbbff8_01ff0000
Reg   2= bbbbbfa1_01f40001   Reg  10= aaaaafe9_01fd0001   Reg  18= aaaaaff9_01ff0001
Reg   3= bbbbbfa2_01f40002   Reg  11= bbbbbfea_01fd0002   Reg  19= bbbbbffa_01ff0002
Reg   4= bbbbbfa3_01f40003   Reg  12= aaaaafeb_01fd0003   Reg  20= aaaaaffb_01ff0003
Reg   5= bbbbbfa4_01f40004   Reg  13= bbbbbfec_01fd0004   Reg  21= bbbbbffc_01ff0004
Reg   6= bbbbbfa5_01f40005   Reg  14= aaaaafed_01fd0005   Reg  22= aaaaaffd_01ff0005
Reg   7= bbbbbfa6_01f40006   Reg  15= bbbbbfee_01fd0006   Reg  23= bbbbbffe_01ff0006
Reg   8= bbbbbfa7_01f40007   Reg  16= aaaaafef_01fd0007   Reg  24= aaaaafff_01ff0007
```

Based on the word format encoded in the 64-bit word, it is confirmed that the three above data is loaded from cache set-500, set-509 and set-511. The lsb of each word is ascedning according to the register address which they are loaded into. This a way to verify the data is loaded in properly according to the instruction sent out to the CPU.

For the first hit, data is loaded from cache set in different order to test the shifting of hit way number to the last index of the LRU array. Below shows the order of hit way number of each set of the data cache line.

Set-500: hit-way7, hit-way6, hit-way5, hit-way4, hit-way3, hit-way2, hit-way1, hit-way0;

Set-509: hit-way5, hit-way6, hit-way7, hit-way2, hit-way3, hit-way4, hit-way0, hit-way1;

Set-511: hit-way4, hit-way3, hit-way2, hit-way7, hit-way6, hit-way5, hit-way0, hit-way1;

Below is the expected result of data cache value being written back to the register file, where the first hit value is loaded to the first regsiter address.

```
Reg   1= bbbbbfa7_01f40007   Reg   9= bbbbbfed_01fd0005   Reg  17= bbbbbffc_01ff0004
Reg   2= bbbbbfa6_01f40006   Reg  10= aaaaafee_01fd0006   Reg  18= aaaaaffb_01ff0003
Reg   3= bbbbbfa5_01f40005   Reg  11= bbbbbfef_01fd0007   Reg  19= bbbbbffa_01ff0002
Reg   4= bbbbbfa4_01f40004   Reg  12= aaaaafea_01fd0002   Reg  20= aaaaafff_01ff0007
Reg   5= bbbbbfa3_01f40003   Reg  13= bbbbbfeb_01fd0003   Reg  21= bbbbbffe_01ff0006
Reg   6= bbbbbfa2_01f40002   Reg  14= aaaaafec_01fd0004   Reg  22= aaaaaffd_01ff0005
Reg   7= bbbbbfa1_01f40001   Reg  15= bbbbbfe8_01fd0000   Reg  23= bbbbbff8_01ff0000
Reg   8= bbbbbfa0_01f40000   Reg  16= aaaaafe9_01fd0001   Reg  24= aaaaaff9_01ff0001
```

The LSB of each 64-bit word corresponse to the order of the hit-way of each set. The above three sets are the data loaded from data cache set-500(16'h1f4), set-509(16'h1fd), and set-511(16'h1ff).

For the second miss, data is loaded from the main memory, so each memory line of data is loaded into data

cache and write back into register file. The register file with address of 1 to 24 contain data in ascending order (indicate the data is loaded correctly)

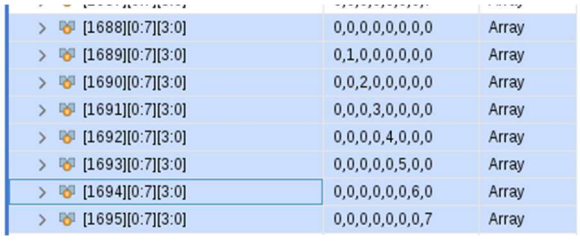
```
Reg     1=  bbbbbfa0_01f40000   Reg     9=  bbbbbfe8_01fd0000   Reg    17=  bbbbbff8_01ff0000
Reg     2=  bbbbbfa1_01f40001   Reg    10=  aaaaafe9_01fd0001   Reg    18=  aaaaaff9_01ff0001
Reg     3=  bbbbbfa2_01f40002   Reg    11=  bbbbbfea_01fd0002   Reg    19=  bbbbbffa_01ff0002
Reg     4=  bbbbbfa3_01f40003   Reg    12=  aaaaafeb_01fd0003   Reg    20=  aaaaaffb_01ff0003
Reg     5=  bbbbbfa4_01f40004   Reg    13=  bbbbbfec_01fd0004   Reg    21=  bbbbbffc_01ff0004
Reg     6=  bbbbbfa5_01f40005   Reg    14=  aaaaafed_01fd0005   Reg    22=  aaaaaffd_01ff0005
Reg     7=  bbbbbfa6_01f40006   Reg    15=  bbbbbfee_01fd0006   Reg    23=  bbbbbffe_01ff0006
Reg     8=  bbbbbfa7_01f40007   Reg    16=  aaaaafef_01fd0007   Reg    24=  aaaaaff_01ff0007
```

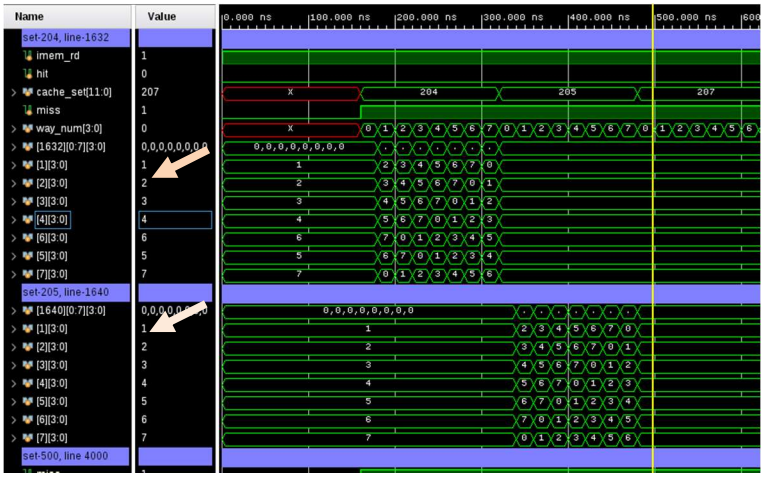
Based on Conventional LRU replacement algorithm, word in reg_1 is replaced in way_7 of set-500, word in reg_9 is replaced in way_5 of set 509, and word in reg_17 is replaced in way_4 of set 511. In set 511, once way_4 is replaced, it rotated to LRU[7] (the last index, indicating most recently used way number). Below is the result of conv_lru_array of set-511 at the end of loading all 8 words, way_4 is rotated back to LRU [0].

```
lru set num:  511
lru_arr[      4088][     0]=  4
lru set num:  511
lru_arr[      4089][     1]=  3
lru set num:  511
lru_arr[      4090][     2]=  2
lru set num:  511
lru_arr[      4091][     3]=  7
lru set num:  511
lru_arr[      4092][     4]=  6
lru set num:  511
lru_arr[      4093][     5]=  5
lru set num:  511
lru_arr[      4094][     6]=  0
lru set num:  511
lru_arr[      4095][     7]=  1
```

Below is the simulation result of all the 9 set of data cache including three instances (initial miss, first hit and second miss), when rd_mem signal is high, miss is high (hit is low), then hit is high (miss is low), and miss=1 (hit=0).

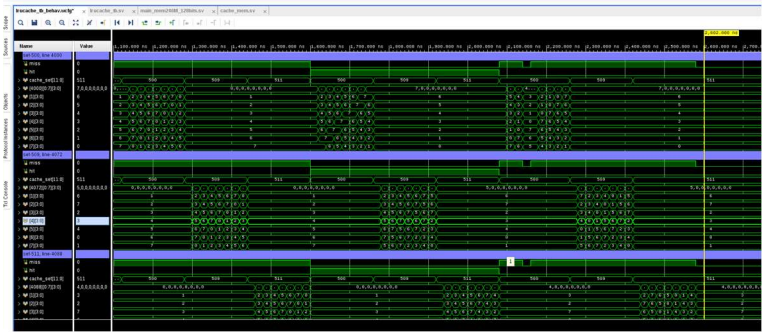
Each set of conv_lru_array has below initialization. For example of set-211 with the initial value way number is the same as the index of the array. It applies to all 512 set in the 4096 data cache line.



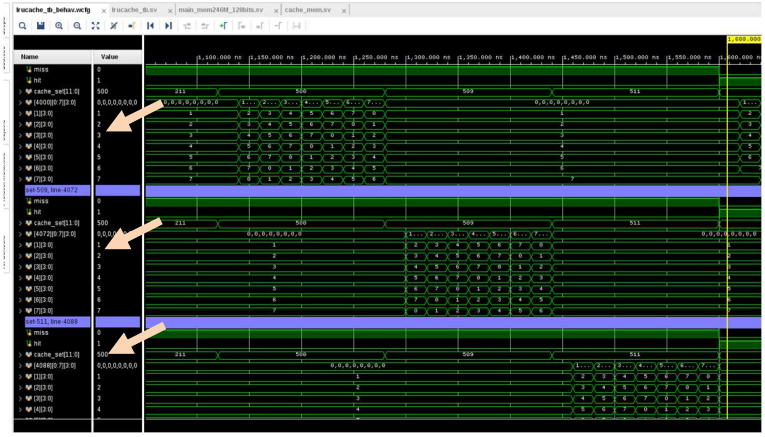


The above simulation result shows rotate the way number 0 to LRU[7], and way number 1 is shifted to LRU[0]. After word 7 (the last word) is written into way number 7, way_7 is rotated back to LRU[7]. The blue arrow shows the way number stored in the conv_lru_array after all 8 ways are written to in the data cache line of set 204. The above waveform also shows the result of set-205 for initial miss to load data from main memory into the data cache. The yellow vertical line indicate value in conv_lru_arr in time 500.00ns.

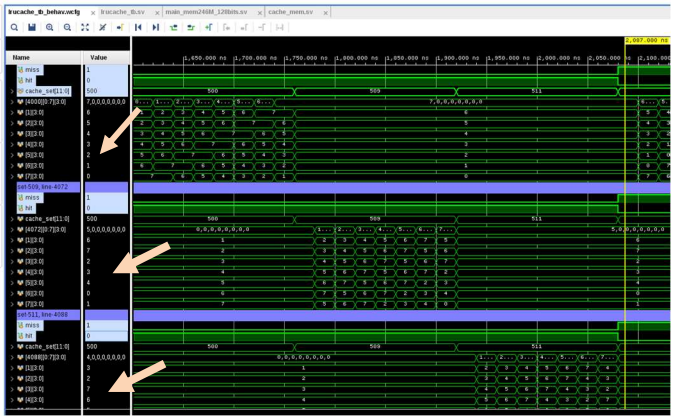
Below is the result of set-500, 509, and 511 for first miss, first hit and second hit.



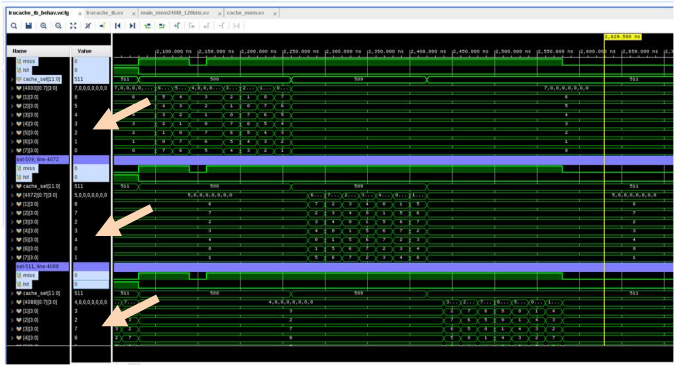
Below is the result of first miss of set-500, 509 and 511. The rotation and shift of way number from one index to the next during writing data word into data cache. When all way numbers are filled up, the least recently used way number is way_0 and most recently used way number is way_7.



Below is the result of first hit. The data words are loaded from data cache and written to register file in a different order as indicated previously. By applying the Conventional LRU algorithm, the hit-way number is rotated to LRU[7], and shift the shift down the way number to lower index. Way number of array in set-500 is expected to be 7, 6, 5, 4, 3, 2, 1, 0 of index0 to 7. Set-509 is expected to be 5, 6, 7, 2, 3, 4, 0, 1, of index0 to 7. Set-511 is expected to be 4, 3, 2, 7, 6, 5, 0, 1, of index0 to 7 accordingly (a clearer array value of set-511 is on previous page, result from the console).

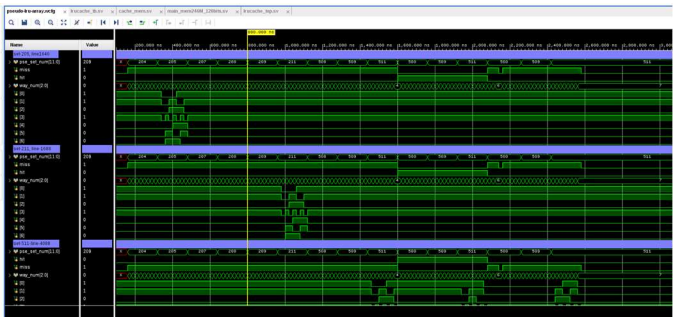


Below is the second miss. Data word is written to the cache set 500, 509, 511 using replacement policy algorithm. Since this is a miss, the replacement is the way number stored in the LRU [0], and rotate the way number to the LRU [7]. After writing 8 data word to each set of data cache, the way number store in the array is expected to be the same as above hit condition.



2. *Tree Based Pseudo LRU load in data and expected result:*

The same test bench and the same instruction sets is used to simulate the Pseudo LRU algorithm. The simulation is run for first miss, first hit, and second miss for 9 set of data cache line. Below is the overall simulation result for all three cases.



For convenient in checking result of the algorithm in simulation waveform, the first data word is being label as data_0, 2nd data word is data_1 and so on. During hit, the order of data word being loaded from the data cache is being shuffled the same order as simulation in Conventional LRU.

Set-500: data_7, data_6, data_5, data_4, data_3, data_2, data_1, data_0;

Set-509: data_5, data_6, data_7, data_2, data_3, data_4, data_0, data_1;

Set-511: data_4, data_3, data_2, data_7, data_6, data_5, data_0, data_1;

During second miss, new word data is loaded into cache using Tree Based Pseudo LRU replacement policy algorithm.

The initial load of data word from main memory to cache set-204, 205, 500, 509, and 511 is consistent for all 9 set, where data_0 is written to way number 0, data_1 is written to way number 7 and so on. Since this is a miss, the algorithm is used to choose the way number to write data word into data cache. Below is a visual representation of data word of a set is written into a way number in the set.

Initial miss, loading data into cache

set-204	dat_0	dat_1	dat_2	dat_3	dat_4	dat_5	dat_6	dat_7
way_num	w_0	w_7	w_3	w_5	w_1	w_6	w_2	w_4

set-205	dat_0	dat_1	dat_2	dat_3	dat_4	dat_5	dat_6	dat_7
way_num	w_0	w_7	w_3	w_5	w_1	w_6	w_2	w_4

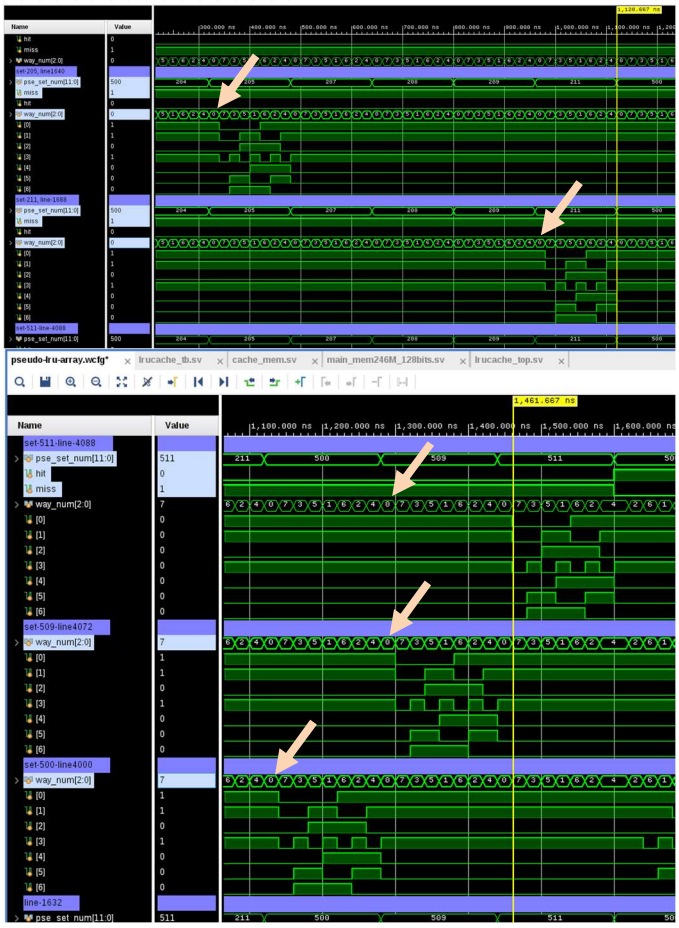
Initial miss, loading data into cache

set-500	dat_0	dat_1	dat_2	dat_3	dat_4	dat_5	dat_6	dat_7
way_num	w_0	w_7	w_3	w_5	w_1	w_6	w_2	w_4

set-509	dat_0	dat_1	dat_2	dat_3	dat_4	dat_5	dat_6	dat_7
way_num	w_0	w_7	w_3	w_5	w_1	w_6	w_2	w_4

set-511	dat_0	dat_1	dat_2	dat_3	dat_4	dat_5	dat_6	dat_7
way_num	w_0	w_7	w_3	w_5	w_1	w_6	w_2	w_4

Below is the first miss simulation result of set-205 and set-211, set-500, set-509, and set-511, which have the same data word order loaded into the same order of way number. The arrow shows the order of way number appear in wave form is in the same order of set-204 and set-205: w_0, w_7 …w_4. The arrow points to first way number of the set to write new data word to the data cache line.



During first hit, pseu_lru_array is update according to the way number of the hit way. Below is the order of data words that are cache hit and is mapped correctly to the way number of the hit data word in the cache set.

cache hit: loading data in the order as below to each cache set

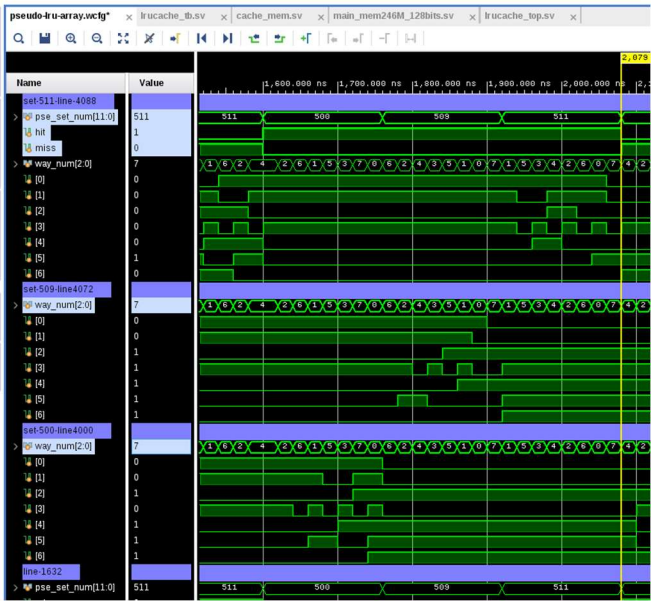
set-500	dat_7	dat_6	dat_5	dat_4	dat_3	dat_2	dat_1	dat_0
way_num	w_4	w_2	w_6	w_1	w_5	w_3	w_7	w_0

set-509	dat_5	dat_6	dat_7	dat_2	dat_3	dat_4	dat_0	dat_1
way_num	w_6	w_2	w_4	w_3	w_5	w_1	w_0	w_7

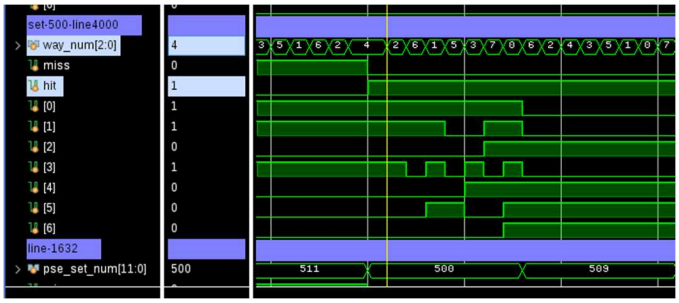
set-511	dat_4	dat_3	dat_2	dat_7	dat_6	dat_5	dat_0	dat_1
way_num	w_1	w_5	w_3	w_4	w_2	w_6	w_0	w_7

The algorithm updates pseudo_lru_array according to the way number being access instantaneously while other bits that is not related to the way number retain the same bit value. For example, in set-500, way number 4 is being accessed, so the pseu_lur_array of set-500 is : Bit[3]=1, Bit[5]=0, Bit[4]=0, and the rest of the bit is not changed. In set-509, way number 6 is being accessed, so the pseu_lru_array of set-509 is: Bit[6]=0, Bit[5]=1, Bit[3]=1, and the rest of the bit is not changed. In set 511, way number is being accessed, so the pseu_lru_array of set-511 is: Bit[3]=0, Bit[2]=0, Bit[0]=1, and the rest of the bit is not changed. The arrays of all the three set is updated according to the way numbers that are cache hit.

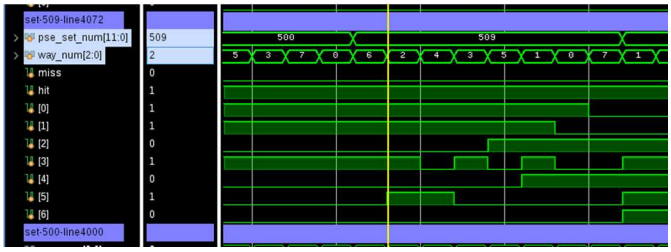
Below is the overview simulation result for first hit of set-511, set-500 and set-509. The simulation is the hit for all three sets. The 1-bit pseudo array value is updated one-clock after the way number is updated, or before the transition to the next new way number.



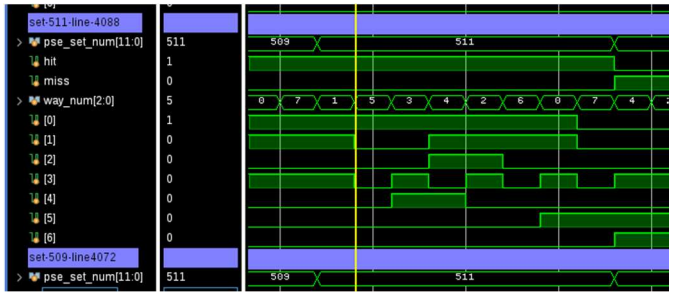
Below is set-500 with the first hit is way number 4: array value is 7'b0001011



Below is set-509 with the first hit is way number 6: array value is 7'b0101011

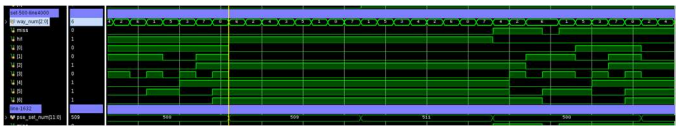


Below is set-511 with the first hit is way number 1; array value is 7'b0000001

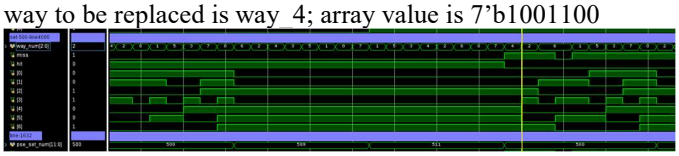


During second miss, pseu_lru_array algorithm searches for approximate least recently used way number to be replaced. The first waveform below is an overview of set-500, 509, and 511 during second miss.

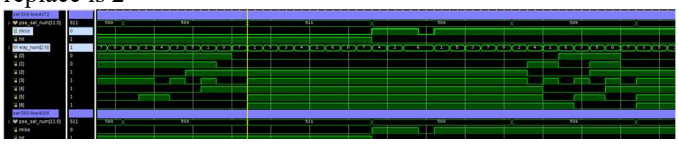
Below is set-500, the last hit way number is 0, and the array value is 7'b1110100. Bit[3]=0, the algorithm check Bit[5]=1, then check Bit[4]=1, so the way number to replace is way number 4.



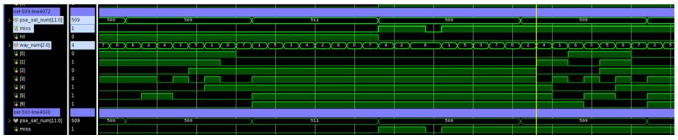
Below is set-500 with the approximate least recently used way to be replaced is way_4; array value is 7'b1001100



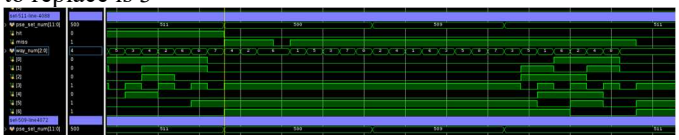
Below is set-509, the last hit way number is 7, and the array value is 7'b1111100. Bit[3]=1, the algorithm check Bit[1]=0, then check Bit[2]=1, so the approximate least way number to replace is 2



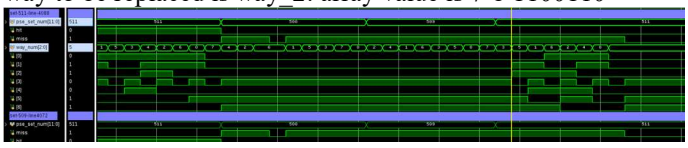
Below is set-509 with the approximate least recently used way to be replaced is way_2: array value is 7'b1110010



Below is set-511, the last hit way number is 7, and the array value is 7'b1101000 , Bit[3]=1 , the algorithm check Bit[1]=0, then check Bit[2]=0 , so the approximate least way number to replace is 3



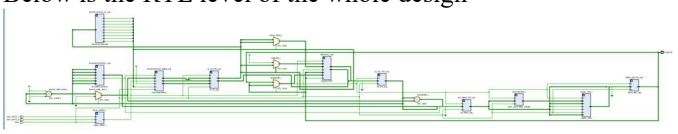
Below is set-511 with the approximate least recently used way to be replaced is way_2: array value is 7'b 1100110



B. Bit Stream Generation Set Up and Results

FPGA board part number = xczu9eg-ffvb1156-2-e
Clock frequency = 40 MHz
Output ports = 8 bits led
Below is an overview of Vivado tool setup to run implementation.

Below is the RTL level of the whole design



The top module of the design is a wrapper of a clock wizard IP (clock_40Mhz_i), 5-stage pipeline CPU, cache_mem_i, and mainmem256_i. The input ports are differential pair clock, which is converted to single ended clock by the clock wizard IP, the output port is 8bit led. To be able to run implementation and generate bit stream, there need to be an output port of the top module which connect to the board pin. For experimentation purpose, the 8bit led output port does not have any contribution the result, nor does it have any negative effect. Below is the top module with three input ports and a clock module to generate a single ended clock source for the rest of the modules.

```
module lrucache_top (
    input user_si570_p,
    input user_si570_n,
    input reset,
    output [7:0] led
);
    logic imem_rd = 1'b1;
    logic clk;

  clock_40Mhz clock_40Mhz_i (
    .clk_out1(clk),              // output clk_out1
    // Status and control signals
    .reset(reset),               // input reset
    .locked(locked),             // output locked
    // Clock in ports
    .clk_in1_p(user_si570_p),    // input clk_in1_p
    .clk_in1_n(user_si570_n)     // input clk_in1_n
  );
    //internal signal
    wire [31:0]  imem_out_wire;
    wire [63:0]  pc_out_wire;
    wire [63:0]  branch_addr_wire;
    wire         Zero_wire , Branch_wire ;
    //IFstg_1=====================================
    InstructionMem1  INSTRUCTION_MEM_unit(
        .clk(clk),
        .reset(reset),
        .imem_rd(imem_rd),
        .imem_out(imem_out_wire),
        .pc_out(pc_out_wire),
        .is_branch(Branch_wire),
        .branch_addr(branch_addr_wire),
```

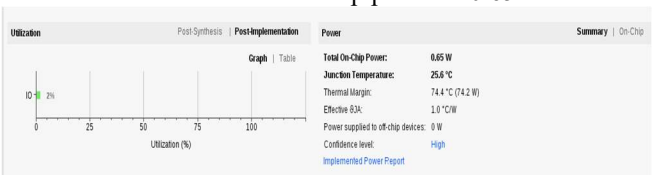
A constraint file is needed to specify the clock frequency for the design, and map the input / output ports to the pin location on the xczu9eg-ffvb1156-2-e FPGA board.

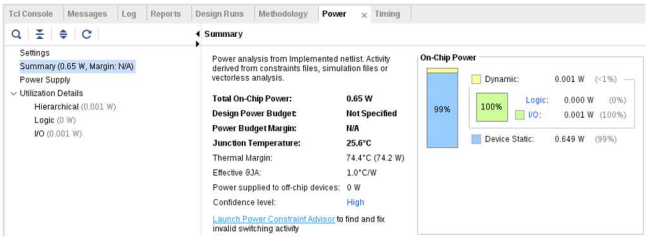
```
set_property PACKAGE_PIN AL8 [get_ports user_si570_p]
set_property PACKAGE_PIN AL7 [get_ports user_si570_n]

set_property IOSTANDARD DIFF_SSTL12 [get_ports user_si570_p]
set_property IOSTANDARD DIFF_SSTL12 [get_ports user_si570_n]
create_clock -period 25.000 -name user_si570_p [get_ports user_si570_p]

set_property PACKAGE_PIN AM13 [get_ports reset]
set_property IOSTANDARD LVCMOS33 [get_ports reset]

# leds:
set_property -dict {PACKAGE_PIN AG14 IOSTANDARD LVCMOS33} [get_ports {led[0]}]
set_property -dict {PACKAGE_PIN AF13 IOSTANDARD LVCMOS33} [get_ports {led[1]}]
set_property -dict {PACKAGE_PIN AE13 IOSTANDARD LVCMOS33} [get_ports {led[2]}]
set_property -dict {PACKAGE_PIN AJ14 IOSTANDARD LVCMOS33} [get_ports {led[3]}]
set_property -dict {PACKAGE_PIN AJ15 IOSTANDARD LVCMOS33} [get_ports {led[4]}]
set_property -dict {PACKAGE_PIN AH13 IOSTANDARD LVCMOS33} [get_ports {led[5]}]
set_property -dict {PACKAGE_PIN AH14 IOSTANDARD LVCMOS33} [get_ports {led[6]}]
set_property -dict {PACKAGE_PIN AL12 IOSTANDARD LVCMOS33} [get_ports {led[7]}]
```

Since the data cache array is larger than the default array size that the tool allow, the variable size need to be set to increase the size in order to run the implementation. Below blue line is the command to be entered in the Tcl console, and the black line is the echo indicating the variable size is set 327680128.

```
set_param synth.elaboration.rodinMoreOptions "rt::set_parameter var_size_limit 327680128"
rt::set_parameter var_size_limit 327680128
```

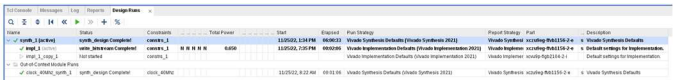
An overview of results after Implementation and Bit stream generation. Design of Conventional LRU and Pseudo LRU have the same utility percentage and power consumption. The two designs (Conv vs. Pseu) use the same 5-stage pipeline CPU; the only different is the replacement policies algorithm of Conventional LRU compared to Tree Based Pseudo LRU. The On-Chip power is 0.65W.



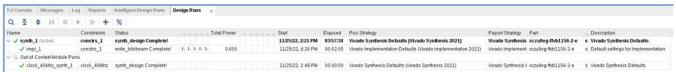


1. *Conventional LRU results after Implementation and Bit Stream generation:*

Below is 8 ways set associative cache memory with 4096 lines.



Below is 4 ways set associative cache memory with 4096 lines.



Below is one-line RTL level of cache memory.

Fig. 13 is one line lru_arr of eight-four-bits register.

```
logic     [3:0] lru_arr[0:0][0:7];
```

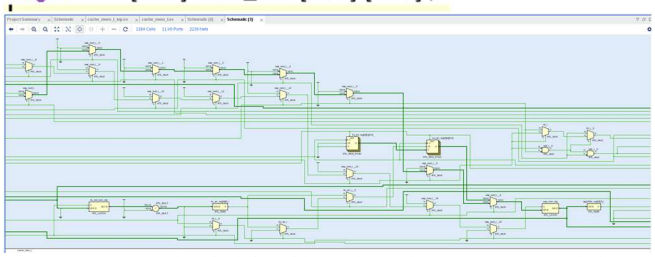
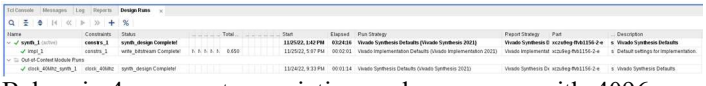


Fig. 13. Conventional Eight 4-bits registers for 8 ways set associative

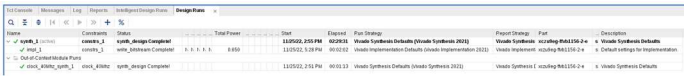
The 8 ways set associative cache mem takes 6-hours to complete bit stream generation, where 4 ways set associative cache mem takes only 4-hours.

2. *Pseudo LRU result after Implementation and Bit Stream Generation:*

Below is 8 ways set associative cache memory with 4096 lines.



Below is 4 ways set associative cache memory with 4096 lines.



Below is one-line RTL level of cache memory.

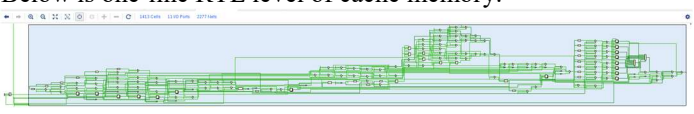


Fig. 14 is one line of pse_arr of one-seven-bits register

```
logic     pse_arr[0:0][0:6];
```

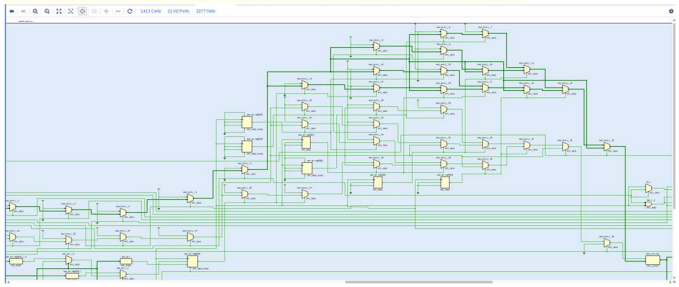


Fig. 14. Pseudo Seven 1-bits register for 8 ways set associative

The 8 ways set associative cache mem takes 3-hours and 30-minutesm to complete the bit stream generation, where 4 ways set associative cache mem takes only 2-hours and 30-minutes.

V. EVALUATING CACHE REPLACEMENT POLICIES

The decrease of way associative from 8 ways to 4 ways, time to implement Conventional LRU cache mem decreases by 40%((4/6)*100=66.6), and time to implement Tree Based Pseudo LRU cache mem also decreases by 40% ((2.5/3.5)*100 = 66.66). It shows the increase of way number increase the complexity of hardware design. However, the time completion of Pseudo cache memory takes 3-hours and 30-minutes, which is half time lesser than Conventional cache memory. As shown in Fig. 13. One cache memory line of 8 ways set associative needs four 8-bits registers to implement Conventional LRU algorithm, so for 4096 lines, the total 4-bits register for the design is 4*8*4096= 131,072 of 1-bits registers. As shown in Fig. 14. One cache memory line of 8 ways set associative needs seven 1-bits register to implement Pseudo LRU algorithm, so for 4096 lines, the total 1-bits register for the design is 7*4096=28,672 of 1-bit registers. Tree Based Pseudo LRU takes 75% less registers compared to Conventional LRU ((7/32)*100=22). Fig. 14 shows Pseudo LUR generates more 1413 cells and 2277 nets, the numbers is more than Conventional LRU, which shows the complexity of the Pseudo algorithm compared to Conventional LRU. The complexity of Pseudo algorithm also shows itself in the HDL SystemVerilog code. However the complex Pseudo algorithm speeds up the design and use less hardware resources. In Table 4 and Table 5, shows time elapse of 11 steps (read_checkpoint, Finished Cross Boundard … write_bitstream) during Synthesis, Implementation and Bit Stream Generation for Conventional LRU vs. Tree Based Pseudo LRU cache memory CPU designs. The Pseudo algorithm is either half lesser time than Conventional algorithm or equal elapse time for both 8 ways and 4 ways set associative. The memory usage of Pseudo is less and free virtual space is more according to the tables.

VI. CONCLUSION

Tree Based Pseudo LRU replacement policy algorithm is more efficient compared to Conventional LRU replacement policy algorithm in both time and hardware resource. The overview utilization is the same as 2% refers to utilization of a whole design. However, each cache memory of Conventional LRU uses half time more registers compared to Pseudo LRU. The registers utilization, which is core skeleton component of a design functionality, is extra hardware resource that can be reduced by implementing Pseudo algorithm instead. Conventional LRU algorithm stores the exact least recently used way number, while Pseudo LRU is the approximation and not the most recently used LRU. The lesser time and hardware resource to implement approximation algorithm, which is as precise as expected outweighs Conventional LRU algorithm.

Table 4. Synthesis and Implementation Result of 8ways Cache Mem of 5-stage Pipeline CPU using Vivado 2021.2

Conv_lru / Tree_Pseu_lru	Conv_lru Time (s): cpu	Tree_Pseu_lru Time (s): cpu	Conv_lru elapsed	Tree_Pseu_lru elapsed	Conv_lru Mem (MB): peak	Tree_Pseu_lru Mem (MB): peak	Conv_lru free virtual	Tree_Pseu_lru free virtual
read_checkpoint:	0:00:04	***0:00:03***	0:00:07	***0:00:05***	2854.836	***2854.801***	172573	***182025***
Finished Cross Boundary :	4:48:25	***2:21:08***	5:01:46	***2:31:56***	21322.738	***19016.305***	149575	***159649***
Finished Timing Optimization :	5:01:11	***2:37:48***	5:15:01	***2:49:05***	21326.32	***19016.305***	146080	***155853***
Finished Technology Mapping :	6:06:32	***3:37:01***	5:54:11	***3:24:43***	21330.242	***19016.305***	145818	***154151***
Synthesis Optimization Complete :	6:09:54	***3:40:14***	5:57:46	***3:28:10***	21330.242	***19016.305***	162648	***173542***
synth_design:	6:10:04	***3:40:23***	5:58:02	***3:28:25***	21330.242	***19016.305***	163823	***174537***
write_checkpoint:	0:02:00	***0:01:53***	0:02:04	***0:01:57***	21332.336	***19019.906***	163821	***174537***
link_design:	0:00:12	***0:00:12***	0:00:17	***0:00:16***	3166.23	***3165.105***	171385	***180847***
Cache Timing Information Task :	10us	***10us***	5ns	***4ns***	3189.707	***3185.613***	171374	***180832***
place_design completed	0:00:24	***0:00:20***	0:00:29	***0:00:29***	4661.129	***4661.066***	170321	***179774***
write_bitstream:	0:00:22	***0:00:22***	0:00:20	***0:00:20***	4000.254	***3999.316***	170914	***180391***

Table 5. Synthesis and Implementation Result of 4ways Cache Mem of 5-stage Pipeline CPU using Vivado 2021.2

Conv_lru / Tree_Pseu_lru	Conv_lru Time (s): cpu	Tree_Pseu_lru Time (s): cpu	Conv_lru elapsed	Tree_Pseu_lru elapsed	Conv_lru Mem (MB): peak	Tree_Pseu_lru Mem (MB): peak	Conv_lru free virtual	Tree_Pseu_lru free virtual
read_checkpoint:	0:00:16	***0:00:03***	0:0005	***0:00:05***	2854.832	***2854.801***	154768	***182025***
Finished Cross Boundary :	2:52:52	***1:55:05***	3:06:07	***1:59:00***	19000.164	***17850.73***	153973	***158475***
Finished Timing Optimization :	3:04:37	***2:15:10***	3:18:23	***2:19:25***	19003.746	***17850.73***	151937	***154055***
Finished Technology Mapping :	3:55:50	***2:21:05***	3:51:06	***2:26:15***	19007.668	***17850.73***	97793	***102721***
Synthesis Optimization Complete :	3:58:53	***2:22:11***	3:54:22	***2:27:22***	19007.668	***17850.73***	113052	***170158***
synth_design:	3:59:03	***2:22:18***	3:54:53	***2:27:35***	19007.668	***17850.73***	113706	***171213***
write_checkpoint:	0:01:53	***0:01:32***	0:01:56	***0:01:35***	19011.766	***17854.332***	115557	***171212***
link_design:	0:00:12	***0:00:12***	0:00:17	***0:00:16***	3165.074	***3165.199***	156610	***176777***
Cache Timing Information Task :	9us	***9us***	4ns	***4ns***	3188.551	***3188.676***	156539	***176753***
place_design completed	0:00:24	***0:00:20***	0:00:29	***0:00:29***	4660.066	***4660.129***	155236	***175707***
write_bitstream:	0:00:22	***0:00:22***	0:00:20	***0:00:20***	4000.258	***3999.316***	153870	***176304***

REFERENCES

[1] S. S. S. Omran and I. Amory, "Implementation of LRU Replacement Policy for Reconfigurable Cache Memory Using FPGA," 2018 International Conference on Advanced Science and Engineering (ICOASE), 2018, pp. 13-18.

[2] S. S. Omran and I. A. Amory, "Reconfigurable cache memory architecture design based on VHDL," 2017 International Conference on Electrical and Computing Technologies and Applications (ICECTA), 2017.

[3] Swadhesh Kumar And Dr. P K Singh. "An Overview of Modern Cache Memory and Performance Analysis of Replacement policies" In 2nd IEEE International Conference on Engineering and Technology, IEEE, 2016.

[4] Hussenin AI-Zoubi and Milena Mikenkovic, "Performance Evaluation of Cache Replacement

[5] Policies for the SPEC CP2000 Benchmark Suite," Proceedings of the 42nd annual Southeast regional conference, pp. 267-272, 2004

[6] https://en.wikipedia.org/wiki/Cache_replacement_policies

[7] Professor Zachary Kurmas at Grand Valley State University, CIS 351 Video 51: LUR and Pseudo LRU https://youtu.be/0qBrbVAJbfc