Cache Replacement Algorithms in Hardware

Trilok Acharya, Meggie Ladlow

May 2008

Abstract

This paper describes the implementation and evaluates the performance of several cache block replacement policies. All of the policies were initially implemented in C using the SimpleScalar cache simulator.

By default, the SimpleScalar cache simulator includes a Least Recently Used (LRU) policy, a First-In, First-Out (FIFO) policy, and a Random policy. These options were expanded to include a page-replacement LRU approximation algorithm, Clock[1]. A simplified version of a more complex page-replacement algorithm, ClockPro[2] was also implemented. Finally, two versions of a replacement policy based on spatial locality were added.

1 Introduction

1.1 Motivation

Memory accesses are an essential part of the processor pipeline, and because they are very frequent, they should be as fast as possible. The fastest kind of memory is the cache. Unfortunately, caches are also very small. In order for the cache to be most useful, it should contain data that is most likely to be used again in the near future. Also, the contents of the cache should be dynamically updated as likely data changes. Data in the cache is stored in sections called data **blocks**. Typically, the cache is divided into **sets**, each set having the capacity to store n blocks where n is the **associativity** of the cache. Cache replacement policies determine which data blocks should be removed from the cache when a new data block is added.

The SimpleScalar cache simulator includes LRU, FIFO, and Random replacement policies. Random is the most simple of these policies; when a cache data block is to be replaced, it simply selects a block from the appropriate set to replace randomly. LRU replacement decides that the block that was least recently used is unlikely to be used again in the near future and replaces that block. FIFO takes a slightly different approach, and decides that the oldest block (the one that entered the cache first) will not be used again soon, and replaces that block.

All of these approaches are flawed. Random is unpredictable, and is likely to perform poorly. Furthermore, there is not necessarily a correlation between how old the block is and whether or not it will be used again soon. The lack of direct correlation here makes FIFO less than optimal. LRU is generally a good choice for block replacement. Unfortunately, it is not feasible to implement in hardware. LRU requires either maintaining an ordered list of when data blocks were accessed (which requires too much space in hardware) or keeping timestamp information about when the block was last accessed (too much maintenence and comparison).

1.2 Alternatives

There are a variety of ways to address the flaws inherent to the three algorithms included in SimpleScalar. The Clock algorithm attempts to make LRU simpler, perhaps even feasible in hardware. ClockPro attempts to improve upon both LRU and Clock by evaluating frequency of access in addition to recency of access. In addition, algorithms based on examining **spatial locality**, or how close instructions are to the currently executing instruction in the code, may provide additional insight into possible performance enhancements.

2 Alternative Cache Block Replacement Algorithms

2.1 Clock

The simple clock hand algorithm is a simple LRU estimation algorithm that provides a reasonably good LRU estimation with low costs in added hardware

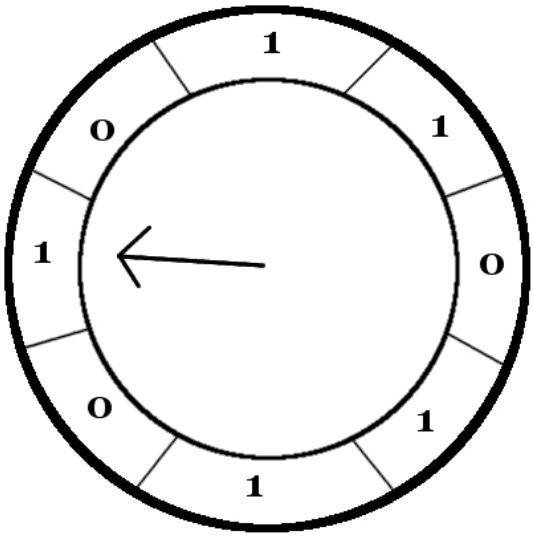

Figure 1: A diagram of state in the Clock algorithm. The clock hand is currently pointing a a block with its clockbit set, so the clockbit will change to 0 and the clock hand will advance to the next block. It will replace the next block because that clockbit is already 0.

and processing time. We implemented two variations of the simple clock hand algorithm. **The 1 bit clock hand algorithm** and **The 2 bit clock hand algorithm**. Since these have tradeoffs in efficiency vs. complexity and size on cache, we decided to implement and investigate both these algorithms despite their apparent similarity.

2.1.1 Description of 1 bit Clock

The Clock algorithm attempts to approximate LRU. Originally intended for page-replacement inside an operating system, it has been adapted to fit SimpleScalar. In a cache, each data block stores some information. It has data, a tag letting the hardware know what data is stored in the block, and a bit that represents whether or not the block is valid. Clock adds two pieces of data. One is a bit called the **clockbit** which represents whether or not the block has been used recently. Each block must have its own clockbit. Additionally, each set must have a **clockhand** which keeps track of what block has been examined most recently.

The Clock Hand LRU algorithm

```
On Start():
  clock_block = first block

On Memory Lookup():
  If clock_block is in cache:
    clock_block.clockbit = 1
  Else:
    curr_block = NULL
    While curr_block == NULL:
      If clock_block.clockbit == 0:
        curr_block = clock_block
      Else:
        clock_block.clockbit = 0
      clock_block = clock_block.next

    If curr_block is dirty: write
    Find accessed block in memory
    Return fetched block to CPU
    Replace curr_block with fetched one
```

2.1.2 Analysis of the Clock Hand algorithm

The clock hand algorithm works on the principle of keeping track of the most recent access of blocks. Since the clock hand initially starts at the beginning, it works as a FIFO implementation, however, by adding a clock bit to keep track of which cache block was accessed before the hand made it all the way around the set, it adds LRU estimation. The best case scenario for this algorithm is when the block the hand is currently pointing to is fit for replacement. In such a case, the algorithm is O(1). In the worst case scenario, all the clock bits would be set and therefore, the clock hand would have to move through every block once, making it O(n). However, such cases would be expected to occur rarely. The most significant weakness of this algorithm is that it keeps track of a history that lasts only as long as one revolution of the clock hand.

Strengths

- Simple to implement
- Only 1 extra bit per block and 1 address pointer per set is required
- A better estimation algorithm than FIFO

Weaknesses

- Weak LRU estimation since it has a short history
- Does not account for frequency of access

2.2 The 2 bit Clock Hand Algorithm

This algorithm has the advantage of being able to keep track of a longer history compared to the 1 bit clock hand algorithm. The functionality of this algorithm is essentially the same as the 1 bit Clock Hand algorithm, however with a 2 clock bits rather than 1 clock bit.

2.2.1 A description of the algorithm

There is only one difference between this algorithm and the 1 bit clock hand algorithm. There are two clock bits, and they are both set and unset one at a time during access and clock rotation respectively. The first time a cache block is accessed, the first clock bit is set. The second time, the second clock bit is set. In this way, this is analogous to having a 2 bit saturating counter keeping track of block accesses. When seeking a block to replace, the algorithm checks to see how many clock bits are set. If two bits are set, it unsets one and advances to the next block. If one bit is set, it unsets it and progresses to the next block. When no bits are set, the algorithm picks the block for replacement.

2.2.2 Analysis of the algorithm

As mentioned earlier, the purpose of this algorithm is to extend on the 1 bit clock hand algorithm to keep a longer history. The history kept by the two bits increases to two revolutions of the clock hand compared to one. Similarly, we could keep adding bits and increasing the history tracked by the clock hand. However, with increasing number of bits being dedicated to storing things other than data blocks, the cache utilization decreases. The worst case scenario for this algorithm is O(2n), although again, this is very rare, more so than for the 1 bit clock.

Strengths

- Better LRU estimation than 1 bit Clock Hand due to longer history
- More sensitive to frequency of access of blocks than the 1 bit clock
- Simplicity of the algorithm

Weaknesses

- Requires two bits to keep track of history, decrease cache utilization

2.3 ClockPro

2.3.1 Description of ClockPro

ClockPro attempts to improve upon both LRU and Clock by considering frequency of access in addition to recency of access. To do so, it categorizes a block as either hot or cold. Hot blocks are accessed frequently, and cold blocks are not. The categorization is done live, and blocks can switch between the two categories. ClockPro also uses the same basic clockbit and clockhand structure of the Clock algorithm. It also uses a test period. The test period tracks how old the block is in the cache.

The full clockhand algorithm includes three clockhands as well as implementation for **nonresident** blocks, or blocks whose history remains cached while the data stored by the block is no longer stored in the cache. In SimpleScalar, a simplified version of this algorithm using only two of the clock hands and not including support for nonresident blocks was implemented.

The ClockPro Algorithm

```
On Start():
  cold_block = first block
  hot_block = first block

On Memory Lookup():
  curr_block = NULL
  If block is in cache:
    Set clockbit
    Return block to CPU
  Else:
    While curr_block == NULL:
      If cold_block.clockbit == 0:
        curr_block = cold_block
      Else if cold_block.test == 1:
        Turn cold hand block hot
        Unset the clockbit
        Run Hot Hand Algorithm
      Else:
        cold_block.clockbit = 0
      cold_block = cold_block.next

    If curr_block is dirty: write
    Find accessed block in memory
    Return fetched block to the CPU
    Replace curr_block with fetched one

Hot Hand Algorithm():
  curr_block = NULL
  While curr_block == NULL:
    If hot_block is cold:
      hot_block.text = 0
    Else if hot_block.clockbit == 0:
```

```
    Turn the block cold
  Else:
    hot_block.clockbit = 0
  hot_block = hot_block.next
```

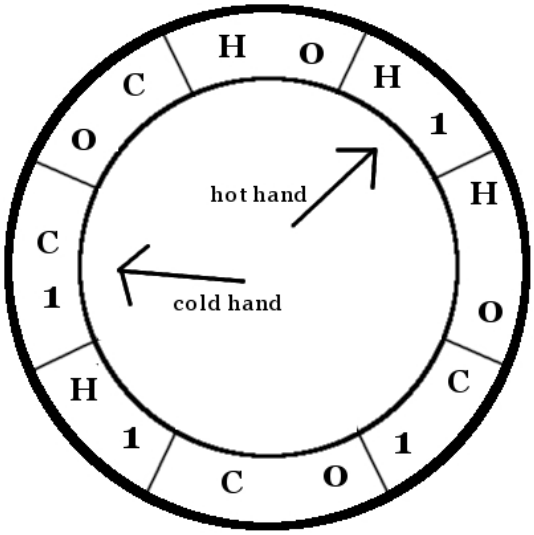

Figure 2: A diagram of state in the ClockPro algorithm. The cold hand is searching for a block to replace. The clockbit of the first block is set, so it becomes a hot block and a hot block must change into a cold block. The hot hand finds the hot block to replace. In this case, it will have to unset the current bit, but move to the next block before finding a suitable replacement. Then the cold hand will replace the next cold block with an unset bit for the new incoming block.

2.3.2 Analysis of the ClockPro Algorithm

The ClockPro algorithm expands on Clock because it tracks the frequency of use in addition to the recency of use. The clock bit of each block tracks how recently it was used. The ability to turn blocks hot or cold represents whether they are accessed frequently. The best case scenario in this algorithm is when the clockbit of the block pointed to by the cold hand is unset. In that case, it is O(1) because that block is simply replaced. In the worst case scenario here, all of the clockbits are set and the first cold block is in its test period. In this case, the cold block turns hot, the hot hand must circle all the way around the clock to find a replacement, then the cold hand must circle all the way around to finally find a cold block to replace. This is still O(n). Such a case would be rare. This version of ClockPro could be improved by keeping more state information about the nonresident blocks.

Strengths

- Considers frequency as well as recency
- Easier to implement than LRU
- Only 3 extra bits (clockbit, test, hot) per block and 2 (or 3 in the full version) extra address pointers per set is required
- Performs better than Clock

Weaknesses

- More complicated than Clock
- While adding support for non-resident blocks could theoretically perform better than LRU in some cases, it would take up more space per set

2.4 Spatial Locality Algorithms

LRU algorithms are effective because of temporal locality of data accesses. However, they do not account for the spatial locality of data accesses, which is another important factor in determining which data blocks are most likely to be accessed next. Therefore, we implemented two spatial locality algorithms to gauge the effectiveness of spatial locality based algorithms. The spatial locality algorithms we implemented used spatial locality exclusively to decide which block on cache was to be replaced.

2.4.1 Description of first Spatial Locality algorithm

This algorithm calculated the distance between the address of the block being requested currently and all the blocks on the set. It then picked the block that was furthest from current block as the candidate for replacement.

Spatial Locality 1

```
On memory lookup(mem_addr):
 max_distance=0
 furthest=NULL
 for each block in set:
    dist=abs(this_block.addr-mem_addr)
    if dist> max_distance:
       max_distance = dist
       furthest=this_block
    if furthest.dirty=true:
       write furthest to memory
 Fetch mem_addr from memory
 Replace furthest with fetched block
```

2.4.2 Analysis of Algorithm

The sole purpose of this algorithm is to find the block in the set which is furthest away from the currently accessed block. This algorithm is very simplistic in its assumption that memory address distance is the sole factor in determining the likelihood of future access. For example, it does not consider the fact that the an address that comes after the currently accessed memory address is likelier to be accessed than one that came before. This algorithm is always O(n), however, this is only because it was implemented in software. On hardware, all distance comparisons could be done simultaneously.

Strengths

- Simple implementation

Strengths

- Simplistic assumptions regarding memory accesses.
- Slow on software, requires N comparatos per set of N in hardware

2.4.3 Description of Second Spatial Locality algorithm

After running tests, we found the first spatial locality algorithm to perform much worse than expected, as we will see in the results section. We therefore decided to address one significant problem with the first algorithm, the fact that it doesn't treat address that come after the current one differently from the addresses that come before.

Therefore, we modified the first spatial locality algorithm so that it would pick the block whose address precedes the address of the current access rather than whichever one is furthest. However, we introduced a threshold factor to balance the distance and the direction of the address.

Spatial Locality 2

```
On Memory lookup(mem_addr):
   max_pos_dist=0
   max_neg_dist=0
   furthest_pos=NULL
   furthest_neg=NULL
   for each block in set:
      dist=this_block.addr-mem_addr
      if (dist < 0 &&
          dist < max_neg_dist):
         max_neg_dist=dist
         furthest_neg=this_block
      else if distance > max_pos_dist:
         max_pos_dist=dist
         furthest_pos=this_block
   if max_neg_dist < -1 *THRESHOLD:
      to_replace=furthest_neg
   else if max_pos_dist > THRESHOL:
      to_replace=furthest_pos
   else:
      to_replace=furthest_neg
```

2.4.4 Analysis of Algorithm

This algorithm improves on the previous one by weighting negative distance more heavily than positive distance. It also uses a threshold so that if the positive distance is much greater than the negative distance, it picks the positive distance as the appropriate candidate for replacement. However, after implementing this algorithm we found that finding this threshold was difficult without a lot of statistical analysis of addresses accesses, and also that this algorithm didn't improve much over the poor performance of the previous algorithm.

3 Hardware Implementation

One of the goals of our project was to implement algorithms that were more feasible to implement in hardware than a perfect LRU would be. We chose the clock hand algorithms because we believed that they were implementable in hardware as well as being good LRU approximators.Therefore, we implemented the clock hand algorithm in Verilog and tested its functionality to test our assumption that it was fairly easy to implement in hardware. We implemented the 1 bit clock hand algorithm in hardware because it was the simplest to implement. Since the 2 bit clock hand algorithm and the clock pro algorithm have very similar implementational details, and therefore, if the 1 bit clock hand algorithm can be implemented in hardware, the other two could also be implemented in hardware.

Our implementation for the 1 bit clock hand algorithm in Verilog worked and was simple to implement. The Verilog code and the simulation showing it's correct functionality are in the Appendix of this paper. The algorithm we implemented was the same as described in the Clock Hand Algorithm section of this paper. For simplicity we used 1 bit data and each block in the cache was made up of the 1 bit clock bit and the 1 bit data. The clock hand algorithm

stores the cache address of the block it is pointing to and progresses through the addresses, jumping back to the starting address after the last address. We did not simulate an entire cache and decided to simulate only one four way associative set.

The implementation has shown us that it is in fact possible to implement the clock hand algorithm for cache replacement. It requires the clock hand implementational hardware to be added for each set, assuming the cache is set associative. This does take up extra space on the cache, but it is much better than implementing a perfect LRU. We do not know what algorithm real cache's use and how complex their implementations are, so we cannot comment on the relative complexity or efficiency of our implementation. However, we have shown that it is in fact implementable in hardware.

4 Results

The general trend with results in order of lowest to highest miss rates is LRU, ClockPro, Clock, FIFO, then Slocal(Spatial locality). Usually, algorithms perform better with 8-way set associativity than with 4-way set associativity. Note that we include only one of our Slocal algorithms because both algorithms tended to perform very similarly with rarely any difference between them.

Some notable exceptions to the above general rule are the floating point Art benchmark and the integer GCC benchmark. In these benchmarks, ClockPro was the best performer, followed by Clock, then LRU and FIFO. Of some interest is the exception shown by Twolf, the only case where Clock and ClockPro have a higher miss rate than FIFO. Worth noting, however, is that the miss rate difference is 0.0001.

In general, changing the set associativity makes more of a difference in the integer benchmarks; the only one that it does not noticeably change is Twolf.

The trend of 8-way set associativity having better performance than 4-way set associativity was expected. Additionally, the trend of Clock's miss rate being between FIFO and LRU was also expected. ClockPro was a less certain element. Theoretically, if the support for non-resident blocks was included, it should perform better than LRU. Since non-resident block support was not implemented, it is more of an LRU approximation algorithm with more state and history than Clock, so one might expected it to perform in between Clock and LRU.

Note that Slocal results were not included in the integer benchmark tests. Trends with high miss rates continued, and not including Slocal results allows for greater precision in graph display. The Slocal results were surprising in some ways since the policy performed so poorly. We expected Spatial Locality and Temporal locality to have equal effect on which cache blocks were going to be accessed soon, however, this is clearly not the case. Our second spatial locality algorithm should have done much better, however it was hard to come up with a good weighting factor between positive and negative direction memory address blocks and with a lot more experimentation, we could probably improve on the results we have using a very similar algorithm. However, we are lead believe spatial locality replacement algorithms by themselves will not perform as well as temporal locality, i.e, LRU replacement algorithms.

Finally, our attempt to simulate implementing Clock in the Quartus simulator shows that Clock can be implemented fairly easily in hardware, unlike LRU. Also, since ClockPro needs a slight increase in state over Clock, ClockPro is also implementable in hardware.

5 Conclusion

It is important to have a good cache replacement policy so that the CPU can minimize cache miss rates. LRU is a very good policy, but is unfortunately impossible to implement in hardware. FIFO is a bad policy, but easy to implement in hardware. Fortunately, the Clock and ClockPro algorithms provide performance similar to LRU and are possible to implement in hardware. While ClockPro is slightly more complicated than Clock and requires more hardware, it also performs better. Spatial Locality is a cool concept, but unfortunately does not seem to work very well on its own.

6 Future Work

There are three main interesting possibilities for future work. One is implementing Clock or ClockPro in hardware with a real cache for testing purposes. This would allow us to more completely evaluate the performance trade off between the extra hardware and the better miss rates.

Additionally, it would be interesting to implement support for non-resident blocks in the ClockPro algorithm and see if the increased data helps it perform

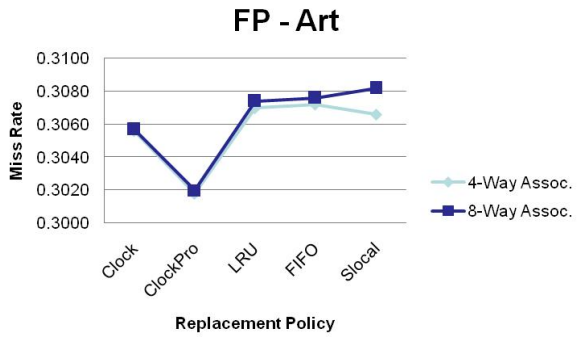

Figure 3: Algorithm results on the floating point Art benchmark.

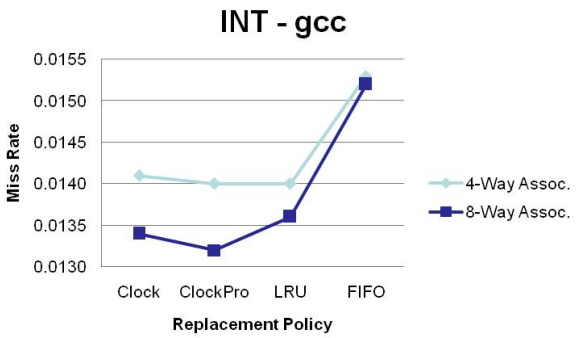

Figure 6: Algorithm results on the integer gcc benchmark.

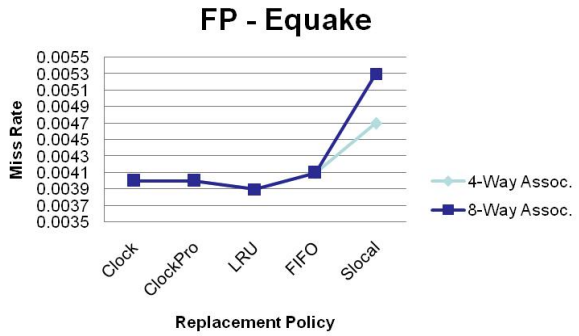

Figure 4: Algorithm results on the floating point Equake benchmark.

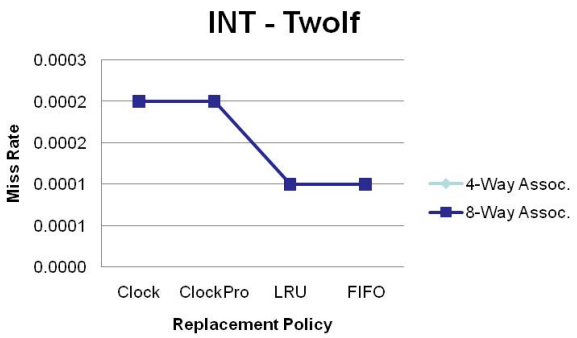

Figure 7: Algorithm results on the integer Twolf benchmark.

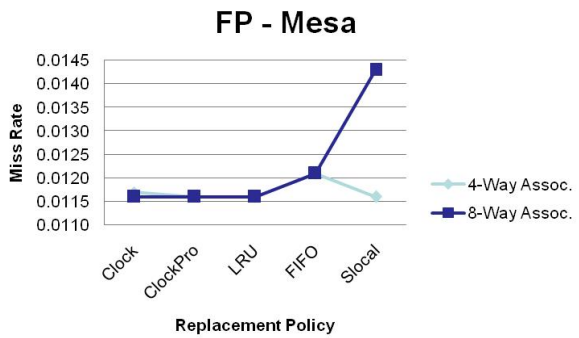

Figure 5: Algorithm results on the floating point Mesa benchmark.

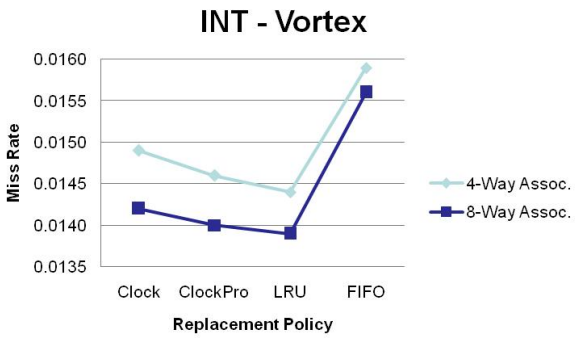

Figure 8: Algorithm results on the integer Vortex benchmark.

better than LRU more consistently.

Finally, combining an algorithm that examines spatial locality with an algorithm that examines temporal locality would also be an interesting experiment. This may be especially relevant for an instruction cache. However, because the spatial locality performs so badly, it seems that taking it into additional consideration would not be helpful.

References

[1] F. J. Corbato, "A Paging Experiment with the Multics System", *MIT Project MAC Report MAC-M-384*, May, 1968.

[2] Feng Chen, Song Jiang, Xiaodong Zhang, *CLOCK-Pro: An Effective Improvement of the CLOCK Replacement*, USENIX Annual Technical Conference, 2005, pp. 323-336.

Appendix A: Quartus Code for Clock Hardware Implementation

```
module ClockHand(Clk,block,write,
                 access,addr,show);
  input Clk,write,block,access;
  input [0:1] addr; // address of access
  output [0:7] show; // show the cache
  // data is 1 bit
  // two bit data storage.
  //  First bit is clock bit
  //  second bit is data.
  //wire [0:1] d_0,d_1,d_2,d_3;
  reg [0:1] d0,d1,d2,d3;
  reg [0:2] hand;
  // hand starts out pointing to 00

  always @ (posedge Clk)
    begin
    if(write)
      begin
      if (hand==2'b00)
        begin
        if (d0[0]==1)
          begin
          d0<=d0&2'b01;
          hand<=2'b01;
          end
        else
          begin
          d0<={1'b0,block};
          hand<=2'b01;
          end
        end
      else if (hand==2'b01)
        begin
        if (d1[0]==1)
          begin
          d1<=d1&2'b01;
          hand<=2'b10;
          end
        else
          begin
          d1<={1'b0,block};
          hand<=2'b10;
          end
        end
      else if (hand==2'b10)
        begin
        if (d2[0]==1)
          begin
          d2<=d2&2'b01;
          hand<=2'b11;
          end
        else
          begin
          d2<={1'b0,block};
          hand<=2'b11;
          end
        end
      else if (hand==2'b11)
        begin
        if (d3[0]==1)
          begin
          d3<=d3&2'b01;
          hand<=2'b00;
          end
        else
          begin
          d3<={1'b0,block};
          hand<=2'b00;
          end
        end
      end
      if(access)
      // on access, set clock bit
        begin
        if(addr==2'b00)
          d0[0]<=1'b1; // set
        else if(addr==2'b01)
          d1[0]<=1'b1; // set
        else if(addr==2'b10)
          d2[0]<=1'b1; // set
        else
          d3[0]<=1'b1; // set
        end
    end
  assign show={d0,d1,d2,d3};
```

```
endmodule
```

Appendix B: Quartus Waveform File for Clock Hardware Implementation