# ECSE 548 (VLSI) Project: MIPS Processor On-Chip Cache Design and Integration

***Abstract*—In this paper, we present an on-chip cache design and integration for 8-bit MIPS processor. The cache that we adopt is a direct-mapped cache with 1 byte cache blocks. It has a size of 16 bytes and is used for both instructions and data. We apply a top-down design methodology, i.e. we start from the architecture level and go down to its layout, with testing on each level. To integrate the cache, MIPS Finite State Machine (FSM) is modified accordingly, which results in controller architecture modifications. Assembly code is also rewritten to test cache functionality by SystemVerilog simulation. Upon completing functionality verification, cache components are implemented and integrated in Electric, and then their testing methodology is explained in detail. In addition, design quality is tested using SPICE simulations.**

## I. Introduction

Cache is a small memory that is used to speed up the processor memory-related operation. Due to **Locality of Reference**, where a process at any given time accesses only a small region of memory, a cache helps significantly reduce processor I/O overhead by loading the small region that can be accessed by the processor with a much faster rate. For example, the typical memory access time of 60 ns of a Pentium processor can be reduced to 15 ns with the presence of a cache [1]. Usually data from memory as placed using *modulo placement* policy, but other policies can also be applied, such as *random placement* proposed in [2].

According to cache replacement policies, they can be classified as direct-mapped cache, set-associative cache, fully-associative cache, random cache [3] etc. In this paper, we propose a design for a simple direct-mapped cache for MIPS processor using SRAM memory elements. The paper is organized as follows. Section II and Section III provides detailed descriptions of proposed design for cache architecture and modifications on the existing MIPS processor for the integration between the two parts; then Section IV presents the process of implementing the design. In Sections V and VI, both functional and performance testing on the design is discussed. Section VII concludes the design process.

## II. Cache Architecture

The MIPS processor presented in the course takes an 8-bit address line and 8-bit data line every time it reads data from external memory; to maintain a manageable project scale while ensuring basic functionality of the cache, the cache was designed to split up the address line into a 4-bit tag line and a 4-bit set line, where set signals are used for row selection in cache whereas tag signals are used for comparison to generate cache hits. The designed configuration results in a 16 8-bit data row in cache, making it capable of storing 16 bytes (4 distinctive assembly instructions) for MIPS.

The resulting design of the cache is shown in Figure 1. Set signals are used to decode data rows inside SRAM arrays, and tag signals are used to compare the tags stored in cache to generate cache hit signals. Write enable (*we* as shown in figure) signal ensures the cache is writeable whenever it set to high. Additionally, for better stability, a validation bit ($v$) was added to each row of SRAM array inside the cache to implement *Compulsory Miss Strategy*, where the first read from cache is always guaranteed to generate a cache miss signal.

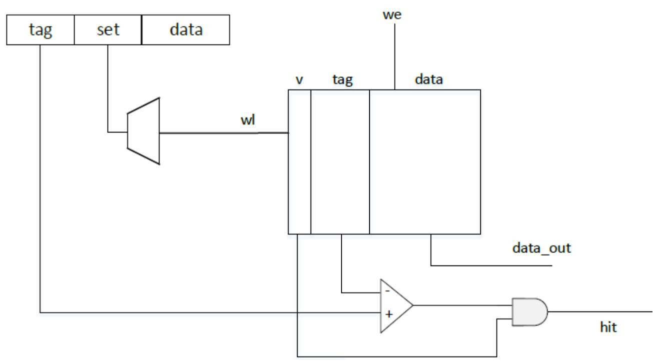


Fig. 1. Cache Microarchitecture

## III. MIPS Integration Architecture

Since our cache is used for both instructions and data, in following sections, when we say data from memory, we refer to both instructions and data unless specified otherwise.

### A. Modified FSM

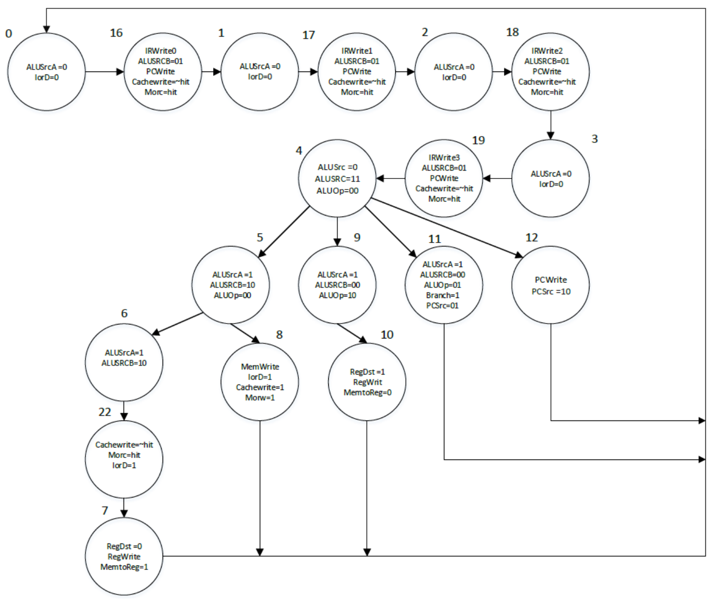


Fig. 2. Modified FSM of Controller

We adopt hierarchical top-down design methodology. We go through architecture level, schematic level and layout level. At architecture level, the Finite State Machine (FSM) is modified and the controller is changed accordingly, see Figure 2. This is implemented in SystemVerilog and the functional simulation has been verified successfully before going down to schematic level.

In Figure 2, the modified FSM is similar to the original MIPS processor FSM. But in order to integrate our on-chip cache into the processor, we added some states for cache accesses and some signals to control additional multiplexers.

In total, we added 5 states. Previously, there are 13 states, i.e. from state 0 to state 12. We added an extra state bit and make it 5-bit FSM. The states we added are state 16 to state 19, and state 22 for cache read access. Besides, we modified state 8 for cache write access.

State 16 to state 19 are added for instruction fetch. Take state 0 as an example. Instead of fetching off-chip memory directly, first it checks the status of the cache at state 0. Then it goes to state 16. If there is a cache hit, the multiplexer selects instruction from cache and puts it into instruction register. Otherwise it reads the instruction from off-chip memory and at the same time, *cachewrite* signal is enabled and corresponding multiplexer select signal is set to store the instruction to the cache simultaneously.

For state 6, it reads data instead of instruction. It works in a similar way as it does for an instruction fetch. First it checks cache status at state 6 and then it reads data at state 22. Apart from *cachewrite* signal and multiplexer select signal, the signal *IorD* is set to 1 to read data.

For state 8, this is a memory write access. For memory write access, we still use only 1 state instead of 2. This is because we have chosen write-through policy for cache, which means whenever a write access happens, the data is written back to both cache and off-chip memory simultaneously. Therefore there is no need to add an extra state. To write data into cache, we need to enable *cachewrite* signal and select the data from datapath using the multiplexer.

## B. MIPS Microarchitecture

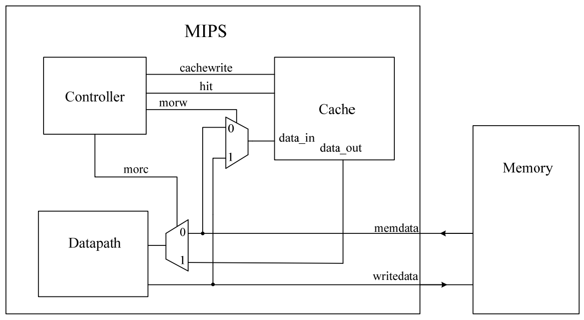


Fig. 3. MIPS Microarchitecture

The MIPS microarchitecture with on-chip cache is shown in Figure 3. In Figure 3, we can see that our cache is integrated. To make it work. two additional multiplexers are added. Besides, three output signals (*cachewrite, morw, morc*) and one input signal (*hit*) are added to the controller due to modified FSM, and they are explained as follows:

- *cachewrite* is used to enable write/read access to the cache with a value of 1/0.
- *morw* is used to select data from memory or datapath. For a read access, if a cache miss happens, it is set to 0 and the data from memory is written into the cache. For a write access, it is set to 1 to send the datapath data to cache.
- *morc* is used to select data from memory or from cache. If there is a cache hit, it is set to 1 and the data from cache is sent to MIPS. Otherwise it is set to 0 and memory data is used.
- *hit* is used to show status of cache. 1 represents a cache hit and 0 represents a cache miss. A compulsory miss strategy was implemented for an ensured miss for a first-time read from cache.

## IV. Implementaion

This section discusses the implementation process of cache components and integration in Electric.

### A. Comparator

A 4-bit comparator was implemented to compare the *tag* bits from the memory and cache to generate corresponding *cache hit* signals. The designed schematic and layout can be found in Figure 4.

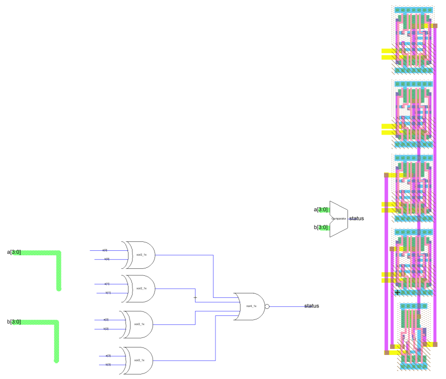
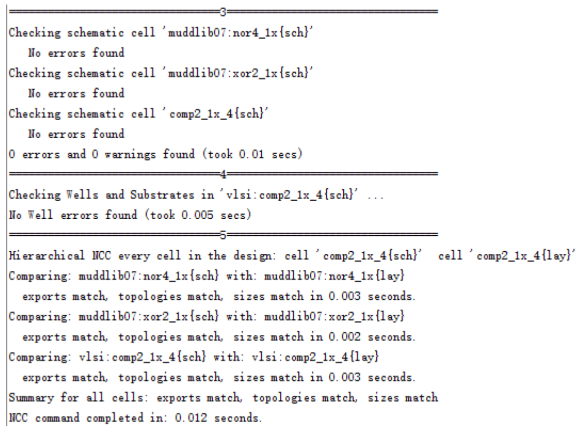


Fig. 4. The Schematic and Layout of Comparator Design

Figure 5 indicates that the comparator design was valid, passing all three design rule checks.

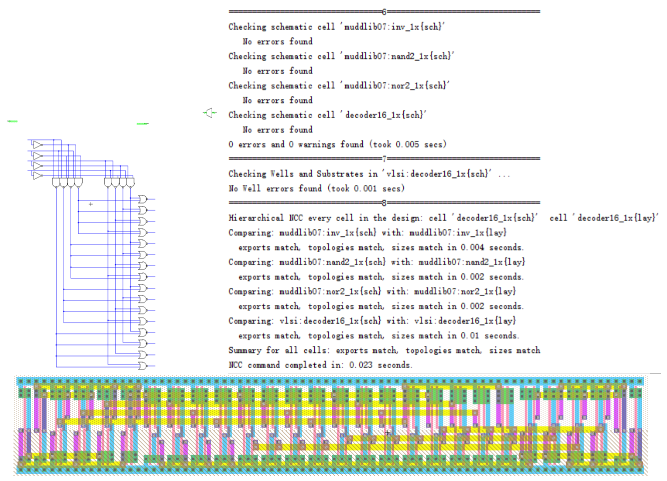


```
================================3==================================
Checking schematic cell 'muddlib07:nor4_1x{sch}'
   No errors found
Checking schematic cell 'muddlib07:xor2_1x{sch}'
   No errors found
Checking schematic cell 'comp2_1x_4{sch}'
   No errors found
0 errors and 0 warnings found (took 0.01 secs)
================================4==================================
Checking Wells and Substrates in 'vlsi:comp2_1x_4{sch}' ...
No Well errors found (took 0.005 secs)
================================5==================================
Hierarchical NCC every cell in the design: cell 'comp2_1x_4{sch}'  cell 'comp2_1x_4{lay}'
Comparing: muddlib07:nor4_1x{sch} with: muddlib07:nor4_1x{lay}
  exports match, topologies match, sizes match in 0.003 seconds.
Comparing: muddlib07:xor2_1x{sch} with: muddlib07:xor2_1x{lay}
  exports match, topologies match, sizes match in 0.002 seconds.
Comparing: vlsi:comp2_1x_4{sch} with: vlsi:comp2_1x_4{lay}
  exports match, topologies match, sizes match in 0.003 seconds.
Summary for all cells: exports match, topologies match, sizes match
NCC command completed in: 0.012 seconds.
```

Fig. 5. Deisgn Rule Checks for Comparator

### B. 4-to-16 Row Decoder

In order for the cache to make use of the 4-bit set signals, a row decoder was applied to decode the set signal and select 1 out of 16 8-bit data lines lying inside the cache. In order to reduce the delay and space usage of the decoder, *pre-decoding* technology was used, resulting in a design shown in Figure 6. It was worth noting that the layout of the decoder was deliberately designed to be wide with a minimum height in order to pitch-match with the SRAM array inside the cache.



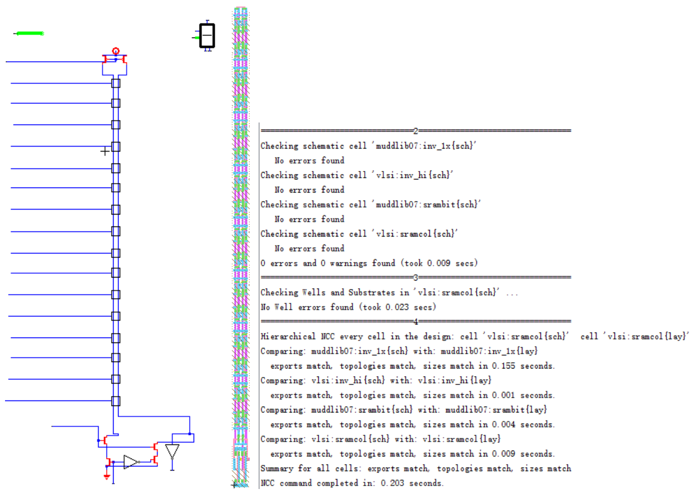
```
=================================6=================================
Checking schematic cell 'muddlib07:inv_1x{sch}'
   No errors found
Checking schematic cell 'muddlib07:nand2_1x{sch}'
   No errors found
Checking schematic cell 'muddlib07:nor2_1x{sch}'
   No errors found
Checking schematic cell 'decoder16_1x{sch}'
   No errors found
0 errors and 0 warnings found (took 0.005 secs)
=================================7=================================
Checking Wells and Substrates in 'vlsi:decoder16_1x{sch}' ...
No Well errors found (took 0.001 secs)
=================================8=================================
Hierarchical NCC every cell in the design: cell 'decoder16_1x{sch}'  cell 'decoder16_1x{lay}'
Comparing: muddlib07:inv_1x{sch} with: muddlib07:inv_1x{lay}
  exports match, topologies match, sizes match in 0.004 seconds.
Comparing: muddlib07:nand2_1x{sch} with: muddlib07:nand2_1x{lay}
  exports match, topologies match, sizes match in 0.002 seconds.
Comparing: muddlib07:nor2_1x{sch} with: muddlib07:nor2_1x{lay}
  exports match, topologies match, sizes match in 0.002 seconds.
Comparing: vlsi:decoder16_1x{sch} with: vlsi:decoder16_1x{lay}
  exports match, topologies match, sizes match in 0.01 seconds.
Summary for all cells: exports match, topologies match, sizes match
NCC command completed in: 0.023 seconds.
```

Fig. 6. The Schematic and Layout of Decoder Design With Design Rule Check Results

It can be shown from the Figure 6 that the decoder design passed all three design rule checks and was ready for the integration into the cache deisgn.

## C. SRAM Column

An SRAM column represents a column of SRAM bits inside the SRAM array of the cache, consisting of 16 SRAM bits sharing bit and inverting bit lines. The design was presented in Figure 7.



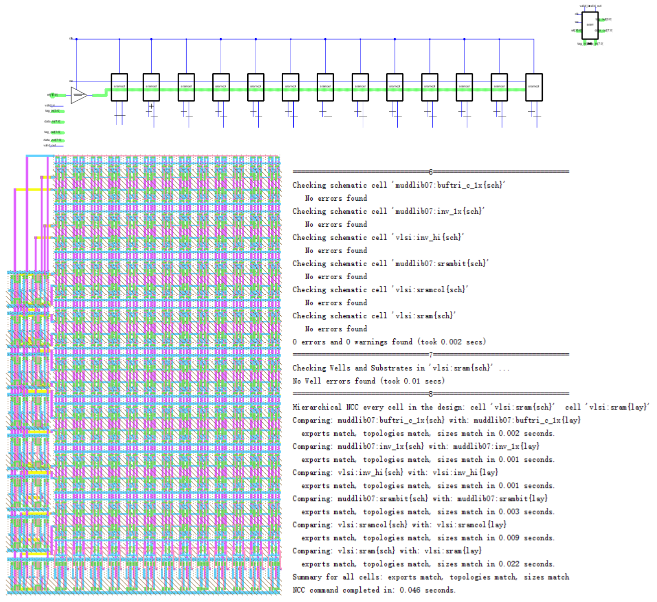
```
=================================2=================================
Checking schematic cell 'muddlib07:inv_1x{sch}'
   No errors found
Checking schematic cell 'vlsi:inv_hi{sch}'
   No errors found
Checking schematic cell 'muddlib07:srambit{sch}'
   No errors found
Checking schematic cell 'vlsi:sramcol{sch}'
   No errors found
0 errors and 0 warnings found (took 0.009 secs)
=================================3=================================
Checking Wells and Substrates in 'vlsi:sramcol{sch}' ...
No Well errors found (took 0.023 secs)
=================================4=================================
Hierarchical NCC every cell in the design: cell 'vlsi:sramcol{sch}'  cell 'vlsi:sramcol{lay}'
Comparing: muddlib07:inv_1x{sch} with: muddlib07:inv_1x{lay}
  exports match, topologies match, sizes match in 0.155 seconds.
Comparing: vlsi:inv_hi{sch} with: vlsi:inv_hi{lay}
  exports match, topologies match, sizes match in 0.001 seconds.
Comparing: muddlib07:srambit{sch} with: muddlib07:srambit{lay}
  exports match, topologies match, sizes match in 0.004 seconds.
Comparing: vlsi:sramcol{sch} with: vlsi:sramcol{lay}
  exports match, topologies match, sizes match in 0.009 seconds.
Summary for all cells: exports match, topologies match, sizes match
NCC command completed in: 0.203 seconds.
```

Fig. 7. The Schematic and Layout of SRAM Column With Design Rule Check Results

The design rule check results can be found in Figure 7. It can be shown from the Figure that all three of the design rule requirements were satisfied by the design.

## D. SRAM Array

The SRAM array is the most essential part of the cache, consisting of 16 *13-bit* SRAM bit rows, with each row having 1 bit for validation (compulsory miss), 4 bits for tag line and 8 bits for data storage. Each row also has a wordline buffer for a stable charging-up. The design is presented in Figure 8; the design rule check results can be found in the same figure as well.



```
=================================6=================================
Checking schematic cell 'muddlib07:buftri_c_1x{sch}'
   No errors found
Checking schematic cell 'muddlib07:inv_1x{sch}'
   No errors found
Checking schematic cell 'vlsi:inv_hi{sch}'
   No errors found
Checking schematic cell 'muddlib07:srambit{sch}'
   No errors found
Checking schematic cell 'vlsi:sramcol{sch}'
   No errors found
Checking schematic cell 'vlsi:sram{sch}'
   No errors found
0 errors and 0 warnings found (took 0.002 secs)
=================================7=================================
Checking Wells and Substrates in 'vlsi:sram{sch}' ...
No Well errors found (took 0.01 secs)
=================================8=================================
Hierarchical NCC every cell in the design: cell 'vlsi:sram{sch}'  cell 'vlsi:sram{lay}'
Comparing: muddlib07:buftri_c_1x{sch} with: muddlib07:buftri_c_1x{lay}
  exports match, topologies match, sizes match in 0.002 seconds.
Comparing: muddlib07:inv_1x{sch} with: muddlib07:inv_1x{lay}
  exports match, topologies match, sizes match in 0.001 seconds.
Comparing: vlsi:inv_hi{sch} with: vlsi:inv_hi{lay}
  exports match, topologies match, sizes match in 0.001 seconds.
Comparing: muddlib07:srambit{sch} with: muddlib07:srambit{lay}
  exports match, topologies match, sizes match in 0.003 seconds.
Comparing: vlsi:sramcol{sch} with: vlsi:sramcol{lay}
  exports match, topologies match, sizes match in 0.009 seconds.
Comparing: vlsi:sram{sch} with: vlsi:sram{lay}
  exports match, topologies match, sizes match in 0.022 seconds.
Summary for all cells: exports match, topologies match, sizes match
NCC command completed in: 0.046 seconds.
```

Fig. 8. The Schematic and Layout of SRAM Array With Design Rule Check Results

## E. Cache

After the implementation of all circuit components, the cache was built based on the microarchitecture of the proposed cache circuit design. For easy integrations with the MIPS processor, in the layout, universal *vdd* and *gnd* tracks were added that connects *vdd* and *gnd* of sub-components together. The cache circuit design, along with the evidence that it passes all three design checks, is presented in Figure 9.

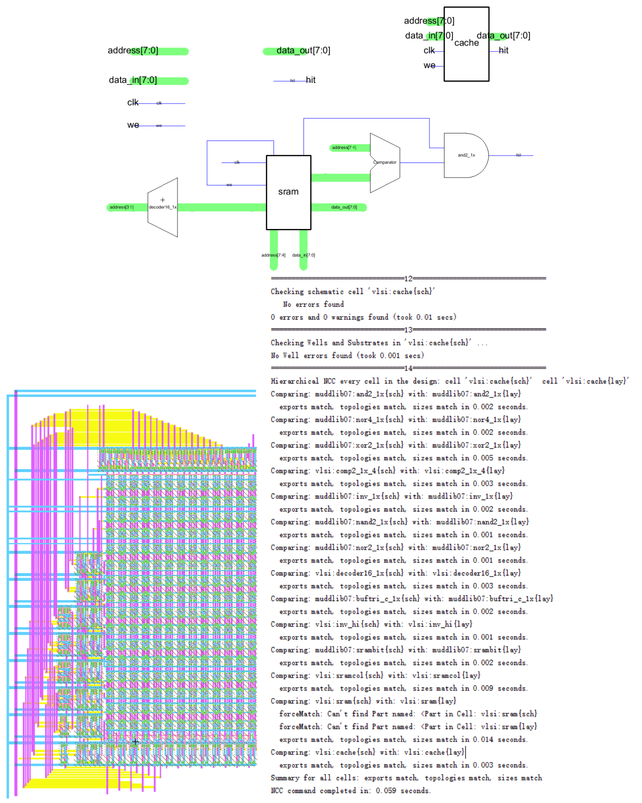
## F. PLA Controller & Controller

As indicated in Section III, with updated FSM for the controller, the PLA controller which represents the FSM logic has been modified accordingly. With an updated FSM description file, a new PLA controller was generated and re-integrated into the controller circuit of MIPS processor. New exports corresponding to cache signals have been added to the controller circuit as well. The resulting circuit is presented in the next subsection.

## G. Integration

Upon the completion and tests of each component, the cache was then integrated with MIPS processor (with updated controller component), resulting in a circuit presented in Figure 10.

The pad frame was then generated from the new MIPS processor with cache implemented. To contain the much-expanded size of the MIPS, the size of the pad frame was extended from *40* pads to *64* pads. The resulting chip, along with error-free design rule check results, is shown in Figure 11. Due to page limits, a truncated version of design rule checks for the chip is presented in this paper; the full check has been presented during the demo with the professor.



```
===============================12===============================
Checking schematic cell 'vlsi:cache{sch}'
   No errors found
0 errors and 0 warnings found (took 0.01 secs)
===============================13===============================
Checking Wells and Substrates in 'vlsi:cache{sch}' ...
No Well errors found (took 0.001 secs)
===============================14===============================
Hierarchical NCC every cell in the design: cell 'vlsi:cache{sch}'  cell 'vlsi:cache{lay}'
Comparing: muddlib07:and2_1x{sch} with: muddlib07:and2_1x{lay}
  exports match, topologies match, sizes match in 0.002 seconds.
Comparing: muddlib07:nor4_1x{sch} with: muddlib07:nor4_1x{lay}
  exports match, topologies match, sizes match in 0.002 seconds.
Comparing: muddlib07:xor2_1x{sch} with: muddlib07:xor2_1x{lay}
  exports match, topologies match, sizes match in 0.005 seconds.
Comparing: vlsi:comp2_1x_4{sch} with: vlsi:comp2_1x_4{lay}
  exports match, topologies match, sizes match in 0.003 seconds.
Comparing: muddlib07:inv_1x{sch} with: muddlib07:inv_1x{lay}
  exports match, topologies match, sizes match in 0.002 seconds.
Comparing: muddlib07:nand2_1x{sch} with: muddlib07:nand2_1x{lay}
  exports match, topologies match, sizes match in 0.001 seconds.
Comparing: muddlib07:nor2_1x{sch} with: muddlib07:nor2_1x{lay}
  exports match, topologies match, sizes match in 0.001 seconds.
Comparing: vlsi:decoder16_1x{sch} with: vlsi:decoder16_1x{lay}
  exports match, topologies match, sizes match in 0.003 seconds.
Comparing: muddlib07:buftri_c_1x{sch} with: muddlib07:buftri_c_1x{lay}
  exports match, topologies match, sizes match in 0.002 seconds.
Comparing: vlsi:inv_hi{sch} with: vlsi:inv_hi{lay}
  exports match, topologies match, sizes match in 0.001 seconds.
Comparing: muddlib07:srambit{sch} with: muddlib07:srambit{lay}
  exports match, topologies match, sizes match in 0.002 seconds.
Comparing: vlsi:sramcol{sch} with: vlsi:sramcol{lay}
  exports match, topologies match, sizes match in 0.009 seconds.
Comparing: vlsi:sram{sch} with: vlsi:sram{lay}
  forceMatch: Can't find Part named: <Part in Cell: vlsi:sram{sch}
  forceMatch: Can't find Part named: <Part in Cell: vlsi:sram{lay}
  exports match, topologies match, sizes match in 0.014 seconds.
Comparing: vlsi:cache{sch} with: vlsi:cache{lay}
  exports match, topologies match, sizes match in 0.003 seconds.
Summary for all cells: exports match, topologies match, sizes match
NCC command completed in: 0.059 seconds.
```

Fig. 9. The Schematic and Layout of Cache With Design Rule Check Results

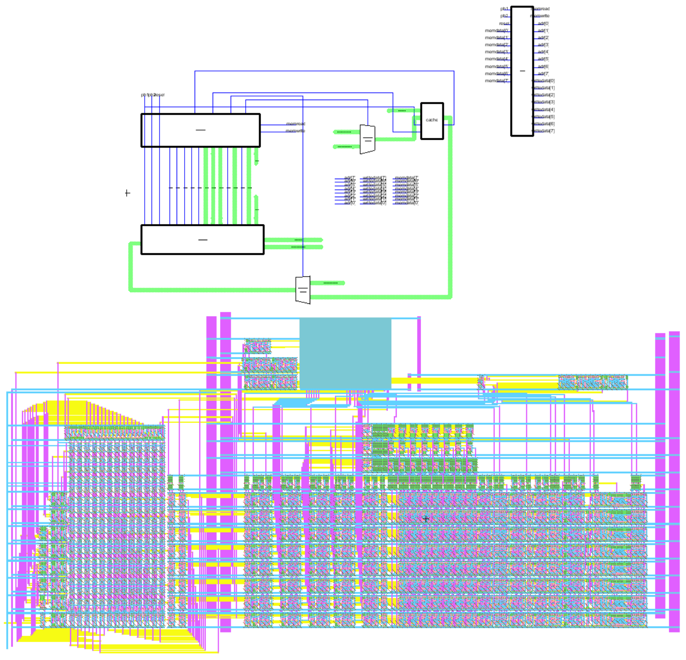


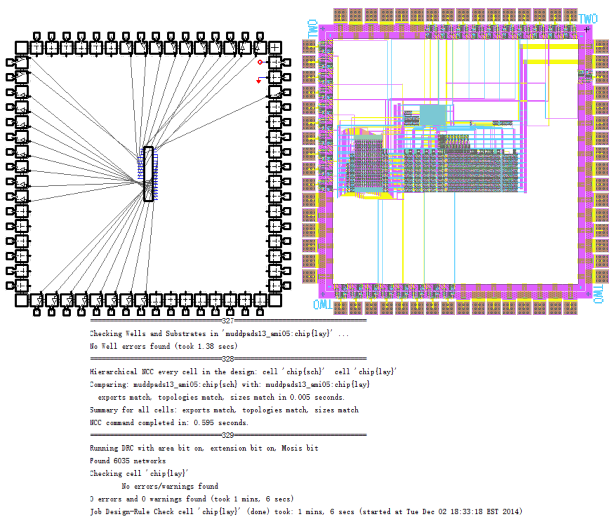
Fig. 10. The Schematic and Layout of MIPS integrated with Cache

## V. Testing

In this section, we explain our testing methodology. We tested components individually in Section V-A. After assembly, the SystemVerilog MIPS processor is simulated for functional testing in Section V-B.

### A. Components Testing

The main components to be tested are shown below:

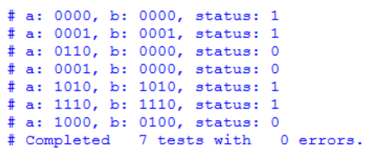


```
===============================327===============================
Checking Wells and Substrates in 'muddpads13_ami05:chip{lay}' ...
No Well errors found (took 1.38 secs)
===============================328===============================
Hierarchical NCC every cell in the design: cell 'chip{sch}'  cell 'chip{lay}'
Comparing: muddpads13_ami05:chip{sch} with: muddpads13_ami05:chip{lay}
  exports match, topologies match, sizes match in 0.005 seconds.
Summary for all cells: exports match, topologies match, sizes match
NCC command completed in: 0.595 seconds.
===============================329===============================
Running DRC with area bit on, extension bit on, Mosis bit
Found 6035 networks
Checking cell 'chip{lay}'
        No errors/warnings found
0 errors and 0 warnings found (took 1 mins, 6 secs)
Job Design-Rule Check cell 'chip{lay}' (done) took: 1 mins, 6 secs (started at Tue Dec 02 18:33:18 EST 2014)
```

Fig. 11. The Schematic and Layout of Chip Integration With Design Rule Check Results

- Comparator

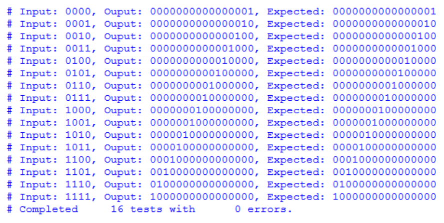
  The comparator was tested in ModelSim with 7 test cases; Figure 12 indicates that the comparator was able to operate correctly, passing all test cases.



```
# a: 0000, b: 0000, status: 1
# a: 0001, b: 0001, status: 1
# a: 0110, b: 0000, status: 0
# a: 0001, b: 0000, status: 0
# a: 1010, b: 1010, status: 1
# a: 1110, b: 1110, status: 1
# a: 1000, b: 0100, status: 0
# Completed    7 tests with    0 errors.
```

Fig. 12. ModelSim Test Results for Comparator

- Decoder
  The 4-to-16 decoder was tested in ModelSim with all 16 test cases; the results showing that the design was operating properly is presented in Figure 13.

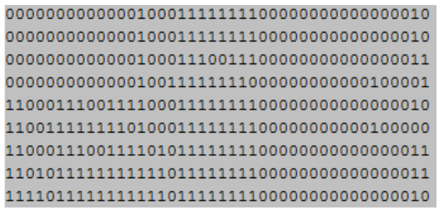


```
# Input: 0000, Ouput: 0000000000000001, Expected: 0000000000000001
# Input: 0001, Ouput: 0000000000000010, Expected: 0000000000000010
# Input: 0010, Ouput: 0000000000000100, Expected: 0000000000000100
# Input: 0011, Ouput: 0000000000001000, Expected: 0000000000001000
# Input: 0100, Ouput: 0000000000010000, Expected: 0000000000010000
# Input: 0101, Ouput: 0000000000100000, Expected: 0000000000100000
# Input: 0110, Ouput: 0000000001000000, Expected: 0000000001000000
# Input: 0111, Ouput: 0000000010000000, Expected: 0000000010000000
# Input: 1000, Ouput: 0000000100000000, Expected: 0000000100000000
# Input: 1001, Ouput: 0000001000000000, Expected: 0000001000000000
# Input: 1010, Ouput: 0000010000000000, Expected: 0000010000000000
# Input: 1011, Ouput: 0000100000000000, Expected: 0000100000000000
# Input: 1100, Ouput: 0001000000000000, Expected: 0001000000000000
# Input: 1101, Ouput: 0010000000000000, Expected: 0010000000000000
# Input: 1110, Ouput: 0100000000000000, Expected: 0100000000000000
# Input: 1111, Ouput: 1000000000000000, Expected: 1000000000000000
# Completed     16 tests with      0 errors.
```

Fig. 13. ModelSim Test Results for decoder

- SRAM

  This is the storage elements in cache. As shown in Figure 1, it has 16 sets and each set consists of 1 *valid* bit, 4 *tag* bits and 8 *data* bits. The input signal *we* is set to 0/1 for data read/write and the signal *wl* is used to select the set for access. Initially, the *valid* bit is 0. Once a cache set is written, its corresponding *valid* becomes 1. *valid* is used to indicate if the data has been loaded or not. We produce some random inputs for testing. Due to the complexity of memory accesses, we generate test vectors from the inputs using our *Python* script, as shown in Figure 14. The signal representation is listed in Table I. The functionality of SRAM is verified successfully according to test vectors.



```
0000000000000100011111111000000000000000010
0000000000000100011111111000000000000000010
0000000000000100011100111000000000000000011
0000000000000100111111111000000000000100001
1100011100111100011111111000000000000000010
1100111111111010001111111100000000000100000
1100011100111101011111111000000000000000011
1101011111111111011111111000000000000000011
1111011111111111011111111000000000000000010
```

Fig. 14. SRAM Test Vectors

TABLE I. SRAM TEST VECTOR REPRESENTATION

| bit position | signal |
|---|---|
| 0 | we |
| 16:1 | wl |
| 24:17 | data_in |
| 28:25 | tag_in |
| 36:29 | expected_data |
| 40:37 | expected_tag |
| 41 | expected_valid |

## B. Functional Testing

In this section, the functional testing of MIPS SystemVerilog model is demonstrated. For this purpose, assembly code is written, as shown in Listing 1.

```
main:    # assembly code        # effect
         lb  $2, 80($0)         # $2 = 5
         lb  $7, 76($0)         # $7 = 3
         lb  $3, 81($7)         # $3 = 12
         or  $4, $7, $2         # $4 <=3 or 5 = 7
         and $5, $3, $4         # $5 <=12 and 7 = 4
         add $5, $5, $4         # $5 <= 4+7 = 11
         beq $5, $7, end        # not taken
         slt $6, $3, $4         # $6 <= 12 < 7 = 0
         beq $6, $0, around     # not taken
         lb  $5, 0($0)          # not taken
around:  slt $6, $7, $2         # $6 <= 3 < 5 = 1
         add $7, $6, $5         # $7 <= 1 + 11 = 12
         sub $7, $7, $2         # $7 <= 12 − 5 = 7
         add $3, $7, $0         # $3 = 7
loop:    beq $3, $0, end        # taken if $3 = 0
         sub $3, $3, $6         # $3 = $3 − 1
         j loop                 # taken
         lb  $7, 0($0)          # not taken
end:     sb  $7, 71($2)         # write adr 76 <=7
         .dw 3
         .dw 5
         .dw 12
```

Listing 1. Assembly Code

At the beginning, instructions are fetched, and there are compulsory cache misses. The assembly code contains a loop to test cache hits. When dealing with this loop, there are cache hits. In Figure 15, we can see that there is a cache miss. This happens because the instructions are used for the first time here. During program execution, cache hits occur for the loop. This is illustrated in Figure 16. The instructions in the loop are put in the cache and then fetched soon. As a result, cache hits happen here.

Apart from cache read access, we also checked cache write access, as shown in Figure 17. In this case, signal *morw* is set to 1 to select data and *cachewrite* is set to 1 to write the data from datapath into cache.

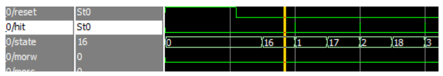


Fig. 15. Cache Read Miss

The functionality verification result is shown in Figure 17. If cache is absent, the write address *adr* is 76 and write data *writedata* is 7. After we integrated on-chip cache, the result is still correct. Our cache functionality is thus verified.

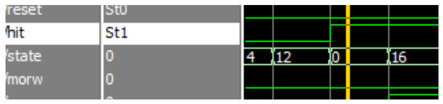


Fig. 16. Cache Read Hit

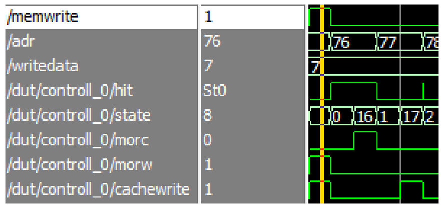


Fig. 17. Cache Write

## VI. DESIGN QUALITY

In this section, performances of certain components, such as regular inverter, high-skew inverter as well as and gate, are evaluated using CAD tool(i.e.PSPICE) since we are not capable to perform evaluations directly on integrated circuits blocks due to the circuit size limitation from CAD tool. The corresponding results are shown in Figure 18 for the setup time analysis of each component respectively. As can be seen clearly from the output waveform, the high-skew inverter pulls up faster than the regular inverter as expected. As for the power dissipation, the corresponding total power consumption of a and gate, inverter and high-skew inverter are $18.9\mu W$, $22.2\mu W$ and $12.7\mu W$, respectively. Therefore, skewed logic is clearly one of the high performance and low power-carry logic components available currently.

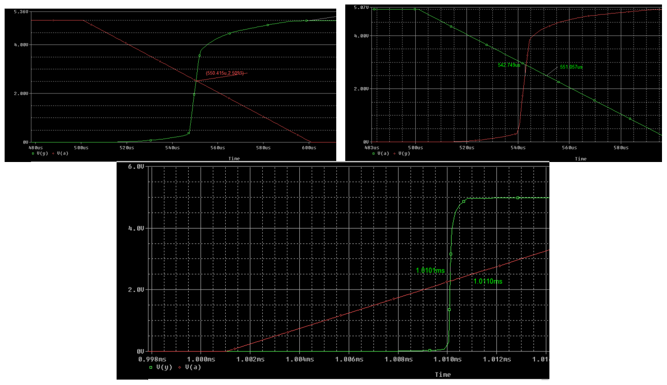


Fig. 18. SPICE simulation of INV, high-skew INV and AND gate

## VII. CONCLUSION

Cache can be implemented in a number of ways. By using different placement and replacement policies, there are different access time and design complexities. In this paper, we successfully designed a simple direct-mapped cache for an 8-bit MIPS processor. It is then implemented as schematic and layout respectively in Electric, and DRC, ERC and NCC tests all passed. In addition, we integrated the cache into the processor by modifying the controller FSM and verified the functionality by SystemVerilog testbench in ModelSim. Finally, we employed SPICE simulation for design quality test. Thus we have finished the project successfully by applying our knowledge from VLSI course and processor cache.

## References

[1] http://download.intel.com/design/intarch/papers/cache6.pdf

[2] Kosmidis L, Abella J, Quinones E, Cazorla FJ (2013a) A cache design for probabilistically analysable real-time systems. In: Proceedings of the Conference on Design, Automation and Test in Europe, EDA Consortium, San Jose, CA, USA, DATE 13, pp 513518, URL http://dl.acm.org/citation.cfm?id=2485288.2485416

[3] Quinones E, Berger E, Bernat G, Cazorla F (2009) Using randomized caches in probabilistic real-time systems. In: Real-Time Systems, 2009. ECRTS 09. 21st Euromicro Conference on, pp 129138, DOI 10.1109/ECRTS.2009.30