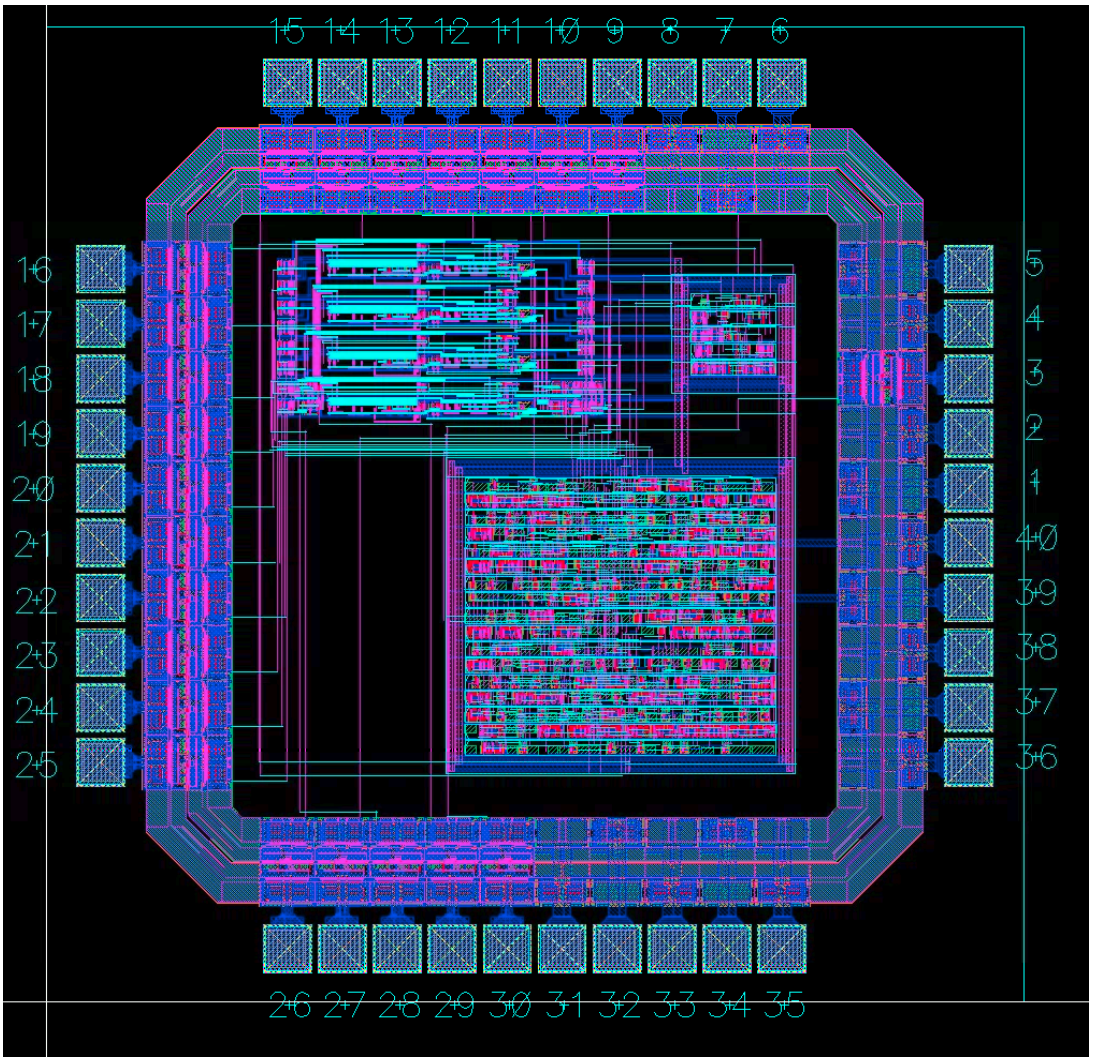
EECS 4612
Project 2: Final Report

Brandon Chan (214218861)
Maneesh Withanagamage (214251326)



1: Introduction

In modern computing, the combination of many page replacement algorithms has contributed to a significant increase in cache performance. The Least Recently Used (LRU) algorithm works by keeping track of how often a cache way is accessed and uses the least accessed way in a set for page replacement. Doing so allows heavily used pages that were used in the past few instructions to be available in the next few instructions as it is more likely that they will be needed.

The goal of our project is to create a chip that replicates the nature of the algorithm for a 4-way set-associative cache that has a total of 8 sets within a 600nM process. In doing so we will keep track of how often each way in each set is accessed by using a 2-bit counter where the value 11 represents the most recently accessed and 00 represents the least recently accessed.

2. Contribution Summary

Brandon: Verilog Synthesis and Verification

Maneesh: Schematic and Layout Construction

As established in the project proposal, Brandon led the design of the verilog simulation and the construction of the necessary modules while also ensuring that the simulation passed all tests in the verification process. At the same time, Maneesh was in charge of the schematic and layout construction ensuring that the modules and components that were being built based on the verilog code were electrically correct, thereby passing all DRC and LVS checks. While both team members were able to complete their tasks in their roles, the roles themselves were not mutually exclusive as each team member was able to help out the other and doing so ensured that both members were able to further improve their skills in VLSI design at each step in the process.

3. Specification

3.1 Chip

The chip is the highest level component of the system that encompasses the whole system as everything that is needed for normal operation is located within it. In addition to the padframe, the overall system consists of three main modules:

- Controller
- Single-port RAM
- Datapath

In a lower level perspective, each of these modules consist of smaller modules each containing their respective schematics and layouts.

3.1.1 Inputs and Outputs

Name	Direction	Bus Width	Description
set_index	input	3-bits	Denotes the set number that is being accessed for a given instruction.
cache_hit_index	input	2-bits	On a cache-hit this value denotes the way index (out of the four ways in a set) that identifies the way that is being accessed by the cache. On a cache miss this value does not need to be defined.
h_m	input	1-bit	Represents whether the current input is a cache hit (1) or cache miss (0)
reset_input	input	1-bit	When set to 1 this input acts a a flag to reset all 4 counters of each set to 00011011 During regular operation this input is set to 0.

clk	input	1-bit	Two phased clock
clk2	input	1-bit	Two phased clock
val_i	input	1-bit	When the bit is high, it ensures that the values that will be fed to the inputs of the chip will be valid.
rdy_i	input	1-bit	When rdy_i is high it indicates that sink is able to accept the output data sent from the chip
val_o	output	1-bit	Is an output from the controller that indicates that the output from the chip is valid and can be sent to the sink.
rdy_o	output	1-bit	Controller signal that lets the sink know that the input values can be accepted into the chip.
outputData	output	8-bits	Represents the resulting four 2-bit counters for the set being accessed
wayIndex	output	2-bits	On a cache miss this denotes the way index that was marked as least recently used and has had its counter updated upon replacement. On a cache hit this value is regarded as a ‘don’t care’’

Table 1: I/O table of Chip.

3.2 Controller

The purpose of the controller is to provide control signals to the rest of the system based on the current state of the FSM (finite state machine). Providing the correct control signals and setting up the correct proceeding states is crucial for the overall system to perform correctly

at the right time. The controller also uses a two phased clocking system to ensure the level sensitive flip flop (master-slave latches) is working as intended.

3.2.1 Verilog Description

As seen in section 9.1 Verilog code the verilog modules pertaining to the controller of the system are lru_controller_synth, statelogic, and outputlogic. In essence, the lru_controller_synth module calls the statelogic module and then sends the outputs of that module to the output logic module. The statelogic module can determine whether the system can transition to the next state or remain in the current state using an always_comb block. The states and their conditions are mentioned below in section 3.2.3. The outputlogic module is then called to assign the new control signals based on the next state defined in the statelogic module, the table that generates the output values are mentioned below in section 3.2.4. The lru_controller module then sends these newly generated signals to the datapath as shown in the top level lru_unit module. The verilog code described in this module is the first component that was synthesized.

3.2.2 Inputs and outputs

Name	Direction	Bus Width	Description
clk	input	1-bit	1st clock input
clk2	input	1-bit	2nd clock input
reset	input	1-bit	When set to 1 sends a signal to the single-port RAM to reset all counter values
val_i	input	1-bit	When the bit is high, it ensures that the values that will be fed to the inputs of the chip will be valid.
rdy_i	input	1-bit	When rdy_i is high it indicates that sink is able to accept the output data sent from the chip
read_done	input	1-bit	A signal from the single-port RAM
output_data_rdy	input	1-bit	Signal received from the datapath to show that the four 2-bit

			counters have been adjusted and are ready for output
output_wayindex_rdy	input	1-bit	Signal received from the datapath to show that the wayindex output is ready
write_done	input	1-bit	Signal received from the single_port_ram that any writing operations have finished
rdy_o	output	1-bit	lets the sink know that the input values can be accepted into the chip.
val_o	output	1-bit	indicates that the output from the chip is valid and can be sent to the sink.
we	output	1-bit	Determines whether the RAM is being read from (0) or written to (1)
output_rdy	output	1-bit	An output signal from the controller signifying that inputs outpu_data_rdy and output_wayindex_rdy are both 1

Table 2: I/O table of Controller.

3.2.3 Finite State Machine (FSM)

All five states of the FSM can be defined as follows:

STATE_RESET:
Defines the state to reset the system. In this state all counters in all sets are reset to a value of:__. No other computation or data transfer is performed

STATE_WAIT:
The system awaits for a new instruction to be sent to the system. This is where the input ready handshake is initiated on the val/rdy interface.

READ_MEMORY:
The system accesses the four 2-bit counters in the specified cache set from the set_index input. These four counters are then sent to the datapath.

CALCULATE_ALG:
Given the counters, the datapath will then calculate the new updated values that will then be re-inserted into the single-port RAM

WRITE_MEMORY:
Insert the counters with their updated values into the single-port RAM, thus overriding the old counter values.

STATE_DONE:
The operation on the current instruction is finished and the system will output the resulting counter values for the pertaining cache set as well as the way index that was replaced (on a cache miss) or no output at all (on a cache hit). The output handshake on the val/rdy interface is also initiated.

3.2.4 Controller Output Signals based on the State

A table of the control signals at each state is shown below:

	rdy_o	val_o	we	output_received
WAIT	1	0	0	0
READ_MEMORY	0	0	0	0
CALCULATE_ALG	0	0	0	1
WRITE_MEMORY	0	0	1	0
DONE	0	1	0	0

Table 3: State table of Controller Signals.

3.2.5 Sequence of operations

The following is a sequence of operations of the controller module

1. Controller will be initialized in the reset state to reset all counters of all cache sets to 00-01-10-11
2. The state will then change to the STATE_WAIT state until val_i and rdy_o are high.
3. The state will then change to READ_MEMORY and will stay until the read_done signal from the datapath is received
4. Once the read_done signal is received the state will change to the CALCULATE_ALG state

5. The state will stay in the CALCULATE_ALG state until the signals output_data_rdy and output_way_index_rdy from the datapath are high
6. The state will then change to WRITE_MEMORY and will remain in this state until the signal write_done is high which comes from the datapath
7. The state will then change to the final state STATE_DONE and will remain there until val_o and rdy_i are high.
8. The state will then change back to STATE_WAIT and the process will repeat

3.3 Single-port RAM

The purpose of the single-port RAM module is to hold all of the counter values of each way and each set of the cache. It is important to note that for this project the single-port RAM was constructed to be part of the overall chip of the system rather than being external to the chip. As such, this limited the amount of cache sets (rows) that the system could keep track of given the dimension constraints for this project. Therefore, the single-port RAM module is limited to 8 sets (or 8 rows) each containing a total of 8 bits to signify a 2-bit counter for each way in the 4-way set-associative cache.

3.3.1 Verilog Description

As shown in the singe_port_ram module in section 9.1 Verilog code the RAM is defined to have 8 rows and 8 columns as defined by the ‘ram’ logic. Given a clock input labelled as ‘clk’, the module has two edge triggers that are always blocks. On the positive edge the module will first check if the reset signal input is set to high as this signals the singe_port_ram to reset all 2-bit counters of all of the sets to the value 00-01-10-11 (or 27 in decimal). That is, in all sets, the counters for way_3 will be 00, the counters for way_2 will be 01, the counters for way_1 will be 10 and the counters for way_0 will be 11. Thus, this defines the second module that was used to synthesize a schematic and layout based on the verilog code defined here.

3.3.2 Inputs and outputs

Name	Direction	Bus Width	Description
clk	input	1-bit	Clock input
we	input	1-bit	Determines whether the RAM is being read from (0) or written to (1)

reset	input	1-bit	When set to 1 all counters in all sets are set to 00011011 otherwise 0.
updated_counter_set	input	8-bits	Denotes the four 2-bit counters that are being written to the RAM
set_index	input	3-bits	Denotes which set (or row) to perform the read or write operation to
read_done	output	1-bit	A signal that tells the Controller module that any reading from the RAM has finished
write_done	output	1-bit	A signal that tells the Controller module that any writing to the RAM has finished
q	output	8-bits	On a read operation, represents the four 2-bit counters being returned given the set_index input. On a write operation this output is not needed so a default value of 00000000

Table 4: I/O table of Single Port Ram.

3.3.3 Sequence of operations

1. When called, the single_port_ram will receive all the necessary inputs as described in the above table.
2. On the positive edge of the clock:
    a. if the reset input signal was 1, then each row of the RAM will have its 8-bits set to 00011011.
    b. If we = 1 (write): this indicates that the RAM is being written to and that the 8-bits that consist of the four 2-bit counters in the specified location defined by the set_index input will be overwritten with the 8-bit input defined as updated_counter_set.
3. On the negative edge of the clock:
    a. If we = 0 (read) this indicates that the RAM is being read and will send the current 8-bits in the specified set_index to the output q and will set the output signal read_done to 1.
    b. If we = 1 (write) this indicates that the RAM is being written to and will set the 8-bits in the specified cache set (row) from the set_index input to the new

8-bit value defined in the updated_counter_set input. The output q will be set to a default value of 00000000 and the write_done output signal will then be set to 1.

3.4 Datapath

The datapath will take the counter values as input, which was sent from the ram module and use a latch to store the values. The latch is enabled by a read_done signal which also comes from the ram module that signifies the counter values are valid and correct. The output of the latch would then be sent to the lru_arithmetic_unit_vector to be processed.

Once the lru_arithmetic_unit_vector updates all the counter values, it would output the updated counters and the index of the way that was updated to “11”. These two outputs will then go to their own latches to be stored. These latches are enabled by a controller signal, this ensures that the value is saved in the appropriate state of the controller which dictates the dataflow.

3.4.1 Inputs and outputs

Name	Direction	Bus Width	Description
cache_hit_index	input	2-bits	On a cache-hit this value denotes the way index (out of the four ways in a set) that identifies the way that is being accessed by the cache. On a cache miss this value does not need to be defined.
we	input	1-bit	Determines whether the RAM is being read from (0) or written to (1)
q	input	8-bits	Represents the resulting four 2-bit counters for the set being accessed
output_rdy	input	1-bit	The signal comes from the controller to notify that the datapath should have the output value ready
h_m	input	1-bit	Represents whether the current input is a cache hit

			(1) or cache miss (0)
outputData	output	8-bits	Represents the newly modified four 2-bit counters
wayIndex	output	2-bits	On a cache miss this denotes the way index that was marked as least recently used and has had its counter updated upon replacement. On a cache hit this value is regarded as a ‘don’t care’’
output_data_rdy	output	1-bit	Signal sent back to the controller to signify that the updated counter values are ready as outputs
output_wayindex_rdy	output	1-bit	Signal sent back to the controller to signify that the wayhit index is ready as output

Table 5: I/O table of Datapath.

3.4.2 Modules

As shown in section 9.4.7 the datapath module is made up of four distinct modules:

- Lru_arithmetic_unit_vector
- Latch_c_1x (3x)
- 2_to_1 mux
- inverter

Break down of each module is shown below starting from the lowest level module.

3.4.2.1 Module: lru_arithmatic_unit as shown in 9.3.4

Sub-modules :

- No2_1x (x2)
- comparator _wayhit_counter
- Half_subtractor (x2)
- 2_to_1 MUX
- 3_to_1 MUX
- 4_to_1 MUX

- Inveter_1x (x3)

The purpose of this module is perform all of the calculations that are needed when adjusting the counter values of each way counter. The output of these modules depends on whether the given instruction is a cache hit or a cache miss. As such a high level look at their behavior and output is described below

Regardless of a cache_hit or a cache_miss, there are three scenarios that could change the way the counter values are outputted.

1. The counter value is greater than the way-hit counter value:
    - The counter value is then decremented by one done by the half_substractor units
2. The counter value is less than the way-hit counter value:
    - The value of this counter remains the same
3. The counter value is equal to the way-hit counter value:
    - This counter IS the way that has a cache hit. Set the counter of this way to 3 or ‘11’ to designate that this way is the most-recently accessed.

3.4.2.1.1 Sequence of Operations

Multiple instances of the lru_arithmatic_unit are needed to create the lru_arithmatic_vector. Therefore, the functionality that this module provides is what lays the foundation on how the system computes the necessary calculations for the system. Below is a list of operations that show one of the four lru_arithmatic_units running in sequential order based on whether the current instruction is a cache-hit or a cache miss.

cache-miss

1. On a cache miss, the 2-bit counter value that was fed as input will be fed into the two nor gates. These two nor gates produce a high when the counter value = 0 or ‘00’. Essentially an equal_zero comparator.
2. If the value is equal to 0 it signifies this way consists of the LRU counter value. The reason there are two nor gates is because the output signal is a 2-bit signal and we need to produce 2-bits.
3. The outputs of these two nor gates are then fed to the d0 input of a 2_to_1 mux where the select value is the h_m signal in which a value of 0 signifies a cache miss.

cache-hit

1. On a cache hit each counter is passed into each lru_arithmatic_unit along with the value of the counter of the way that is designated as the cache hit which is determined by the cache_hit_index input.
2. The cache_hit_index input shows which out of the four ways has the cache hit.
3. These input values are fed to the comparator_wayhit_counter module.
4. This module has two sub modules, namely, the equal_0 module and the greater_than_wayhit_counter module.
    - The two submodules will produce 1-bit each which will become the 2-bit value that is fed into the d1 input of the 2_to_1 mux.

The output of the 2_to_1 mux mentioned in the two scenarios above is essentially the select signal for the 4_to_1 mux that produces the 2-bit updated counter value. The inputs for the 4_to_1 mux are the half_substractor unit output, the current counter value, and the bits '11'. Depending on the select value produced by the 3_to_1 mux the appropriate counter value update is sent to the output.

3.4.2.2 Module: lru_arithmatic_unit_vector as shown in section 9.4.6

Sub-modules :

- lru_arithmetic_unit (x4)
- 2_to_1 MUX
- 4_to_1 MUX (x2)
- Inveter_1x
- 3_to_2 Encoder
    - Composed of two Nor2_1x

3.4.2.2.1 Sequence of Operations

The purpose of this module is to include the lru_arithmetic_unit mentioned above and run 4 parallel instances as there are 4-ways per set.

1. The 4_to_1 mux produces valid values only when there is a cache hit. This mux essentially produces the actual 2-bit counter value of the way that was hit.
2. This value is then fed as an input to the lru_arithmetic_unit. If there is no cache hit, then the mux will produce a 00 which will be fed to the lru_arithmetic_unit, however, that value is not valid and will not be processed in the lru_arithmetic_unit. Each lru_arithmetic_unit will produce a 1-bit x-output and a 2-bit y-output (as seen on figure 43)
3. The x-output of each lru_arithmatic unit is then combined to form a 4-bit number that is then sent to a 3_to_2 encoder. Each bit represents a way and if that bit is high it will signify that counter value was updated to 3. By using a 3_to_2 encoder, the 4-bit outputWayIndex value can be encoded to produce the **wayIndex** output. The reason the encoder only accepts 3-inputs is due to the product of the k-maps where the least significant bit of the outputWayIndex is not needed.
4. The 2-bit y-outputs will then be combined to create the 8-bit **outputData**.

This all can be seen through the schematic in section 9.4 below in appendix.

4. Floorplan

4.1 Proposal floor plan

The initial floor plan, as seen on figure 1 below, submitted for the proposal was lacking in many ways. It did not include the correct dimensions for all the units inside the chip, the number of I/O pads and the size of the I/O pads. The initial structure that was planned during the proposal was also changed considerably as the project started.

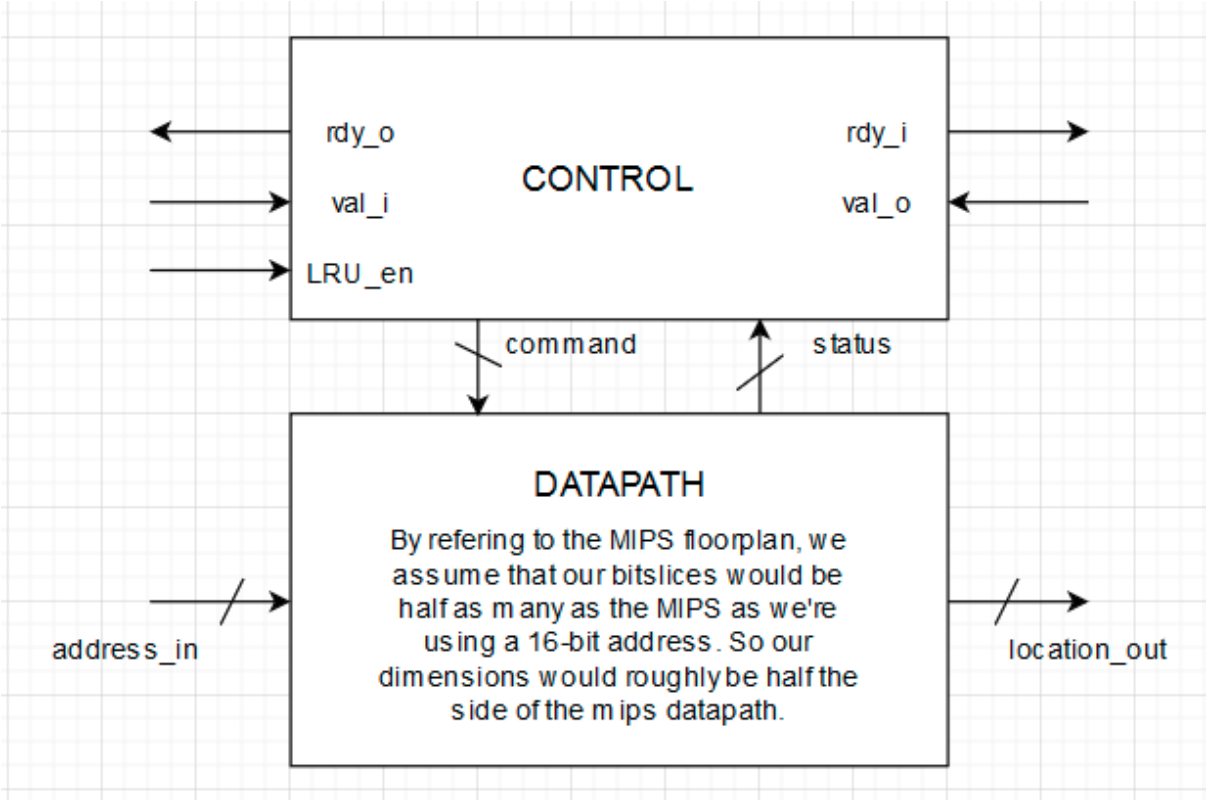


Figure 1: Initial floor plan of the lru chip.

4.2 Final floor plan

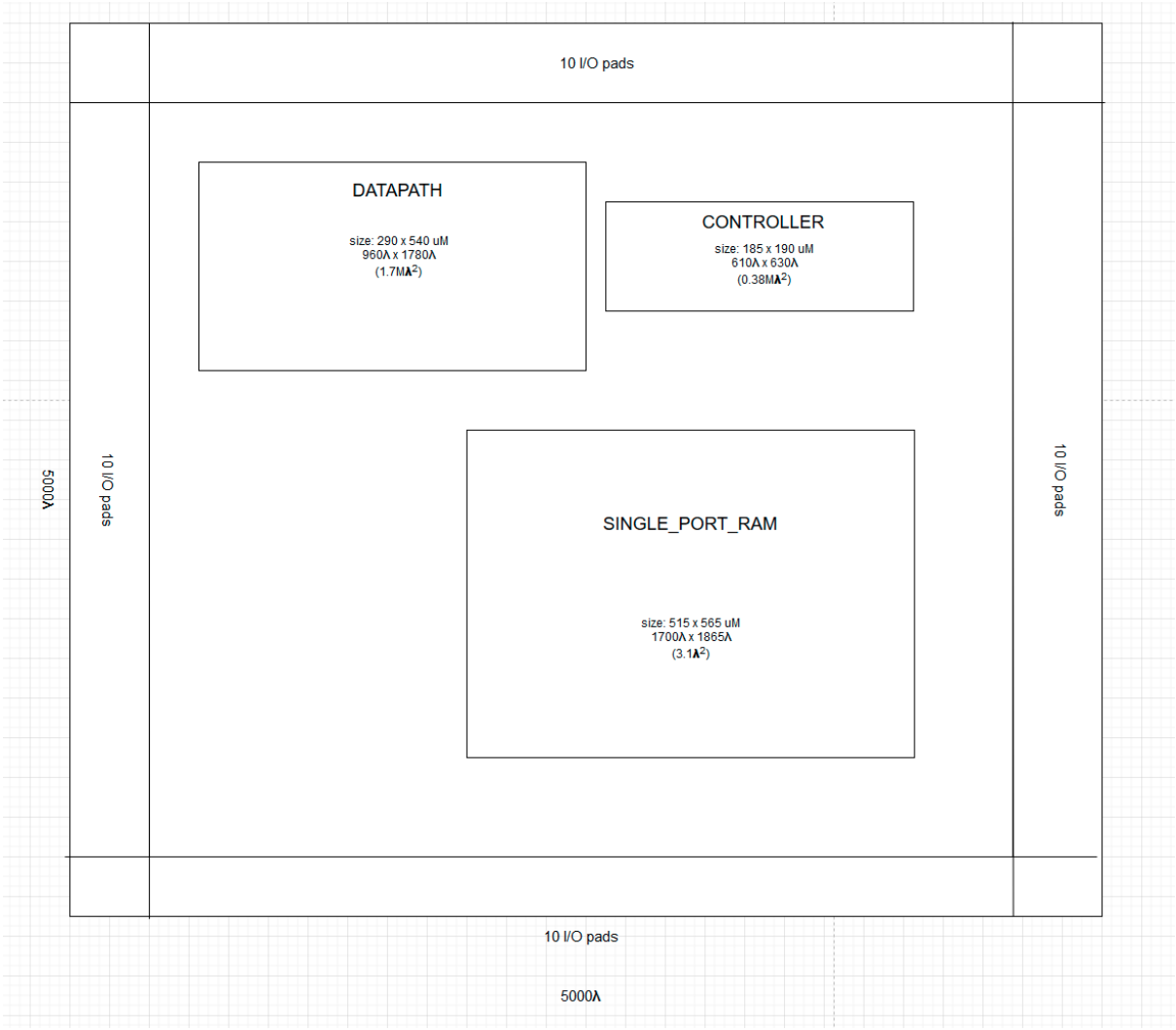


Figure 2: Final floorplan of the lru chip without pin arrangements.

4.3 Discrepancies:

- Single port ram was not included in the datapath in the final floor plan, it acted as an isolated unit instead.

- LRU_en was removed. The initial proposal mentioned using the val/rdy interface which already takes into account flow of data, so it was redundant to have that enable input.
- Additionally reset, clk and clk2 were introduced as inputs for the finalized floor plan.
- address_in was also removed and replaced with a multitude of different inputs as seen through figure 3 pinout diagram below. These new inputs were introduced due to changing our algorithms framework. The difference can be found below.
    - Initially the algorithm was to keep track of how many cycles ago was the address last accessed. Given a 4-bit counter if an andress has the count '1111', it would suggest that address not accessed for 16 clock cycles. This is assuming that the cache is a direct mapped cache.
    - The above structure was completely overhauled as the project progressed. Building a LRU algorithm for a direct-mapped cache proved to be irrelevant. Instead a 4-way set-associative cache was used for the project. Each way in a set would have a 2-bit saturating counter. '11' signified that specific way to be the most recently used (MRU) and '00' signified that specific way to be the LRU regardless of how many clock cycles have passed. This algorithm's framework was inherently different from the initial algorithm, where the counter values only change when that specific set was accessed, unlike the initial framework where the clock cycle would determine the counter value. A more in depth explanation of the final algorithm can be found in the specification section 3 above.
- The location_out output on the datapath remained the same on the finalized floor plan, however instead of providing the address itself as the output, the index of the way, which was to be replaced was outputted. The counter values of that set were also outputted as a means of proving that the counters were updated to its correct values.
- A better view of the command and status signals can be found on the final floor plan on figure 3 pinout diagram below.

4.4 Pinout diagram

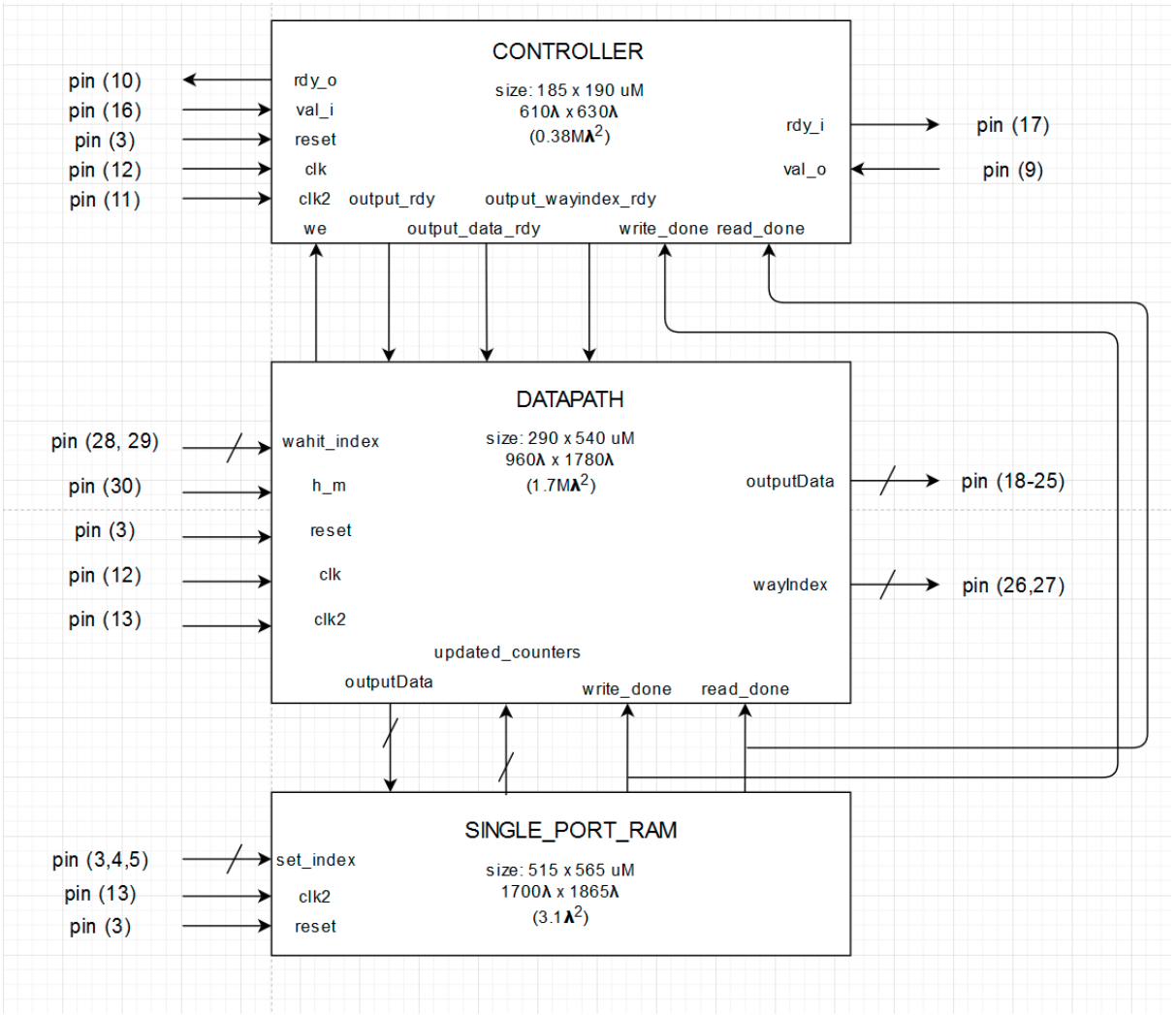


Figure 3: Pinout diagram of the chip.

4.5 LRU Chip Architecture

As seen in figure 4, the initial architecture of the LRU chip consisted of eight parallel pathways where four pathways were dedicated to cache-hit instructions and four pathways were dedicated to cache-miss instructions. This approach suggested that both scenarios would be calculated simultaneously and the correct output would be selected at the end by a 2 input MUX.

Figure 5 outlines the revised architecture that is now implemented in the LRU chip. Instead of having 8 parallel paths, each path now has the capability to perform cache-hit or cache-miss operations based on the h_m (hit-miss) signal provided by the input instruction. Additional registers were also added to save the state of the output values until they are needed.

Higher quality images can be found by referring to the corresponding JPEGs described in section 8

.

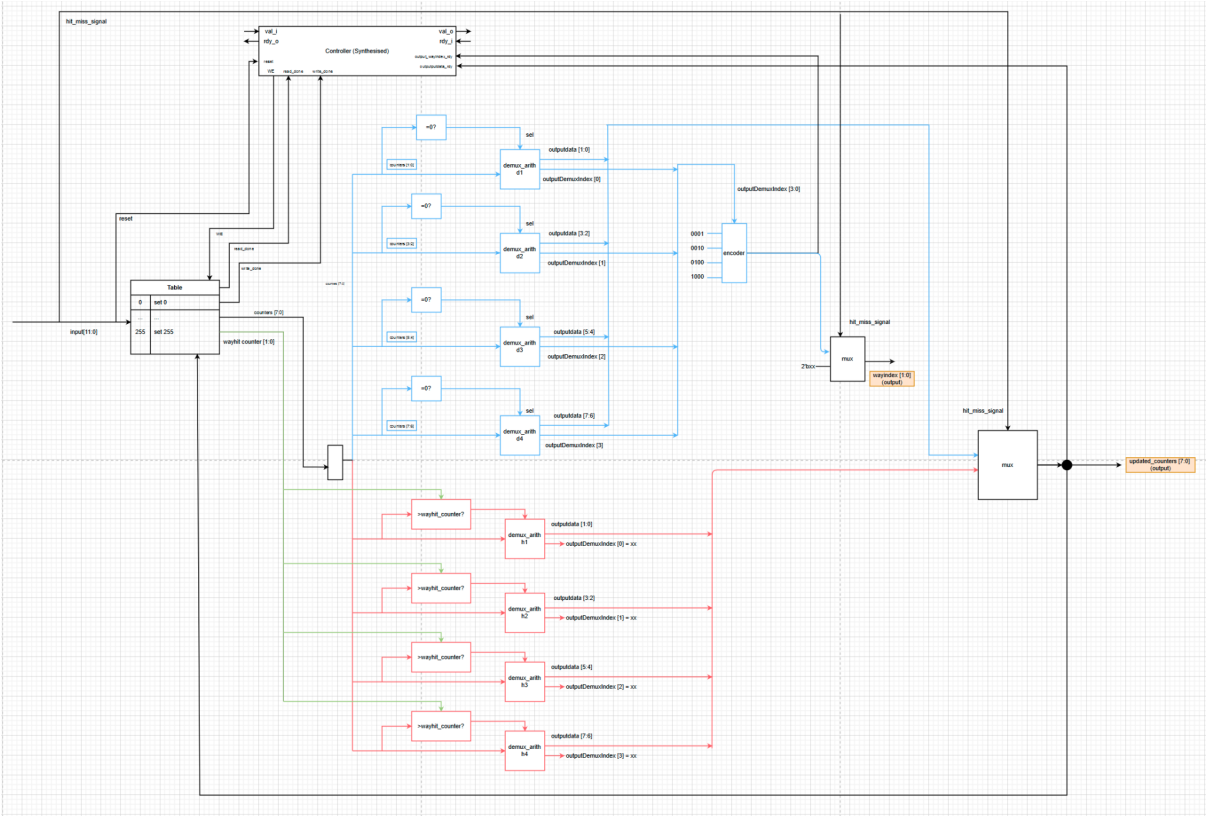


Figure 4: Initial architecture of the lru chip.

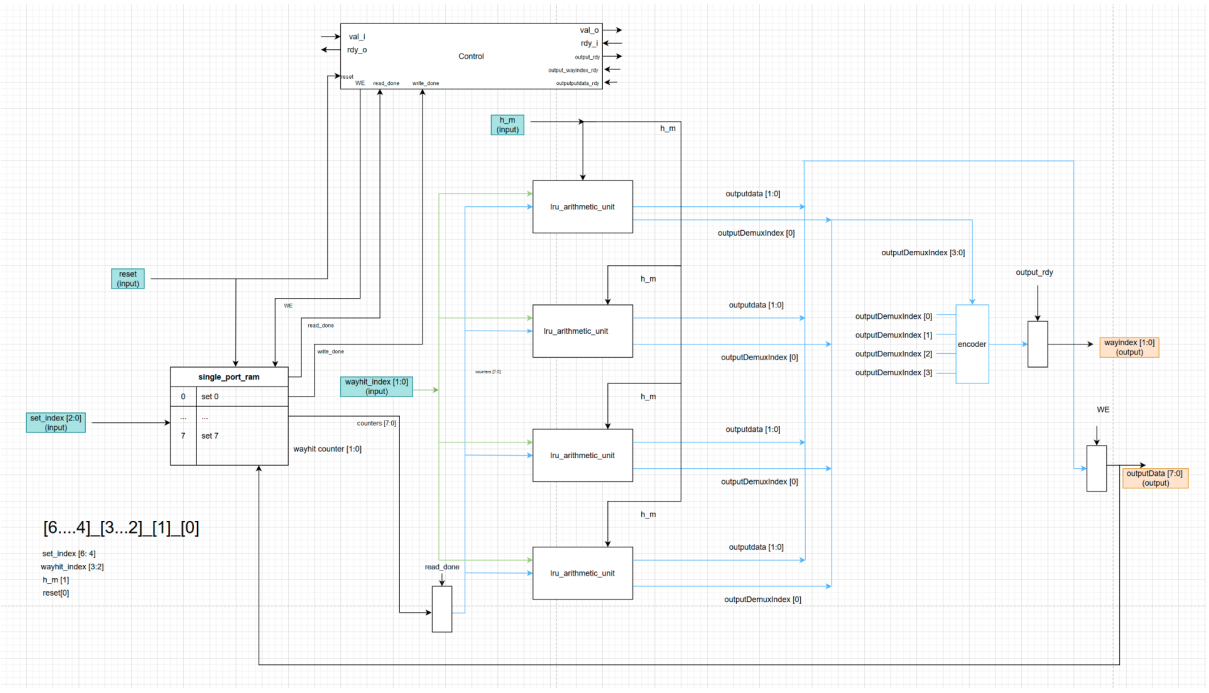


Figure 5: Final architecture of the lru chip.

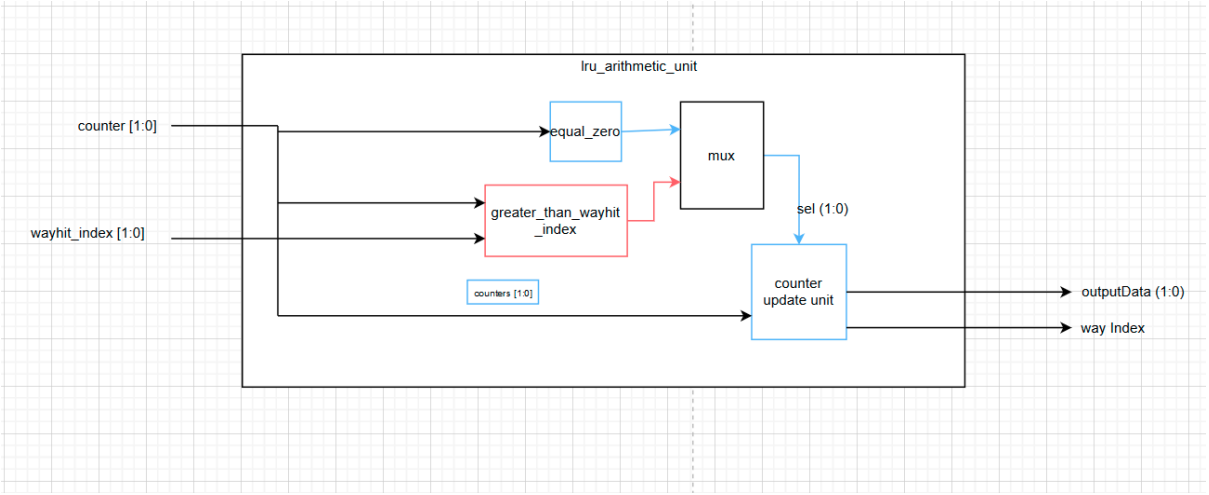


Figure 6: Internal view of the lru_arithmetic unit.

5. Verification

The following figures outline the various schematic, DRC and LVS checks where applicable. In order to remain concise, modules that consisted of many smaller leaf modules had schematic tests run at the higher level rather than creating test vectors for each leaf module. For example, running schematic tests (with respective test vectors and testfixtures) on the compare_wayhit schematic also tests the equal_0 module and greater_than_wayhit_counter module.

The synthesized modules like the controller and the single_port_ram were also tested within the core and chip components rather than by itself. A visual inspection of the waveform was done to ensure that the synthesised modules behave according to our specifications.

5.1 Verilog testbench

Using the code from section 9.1.

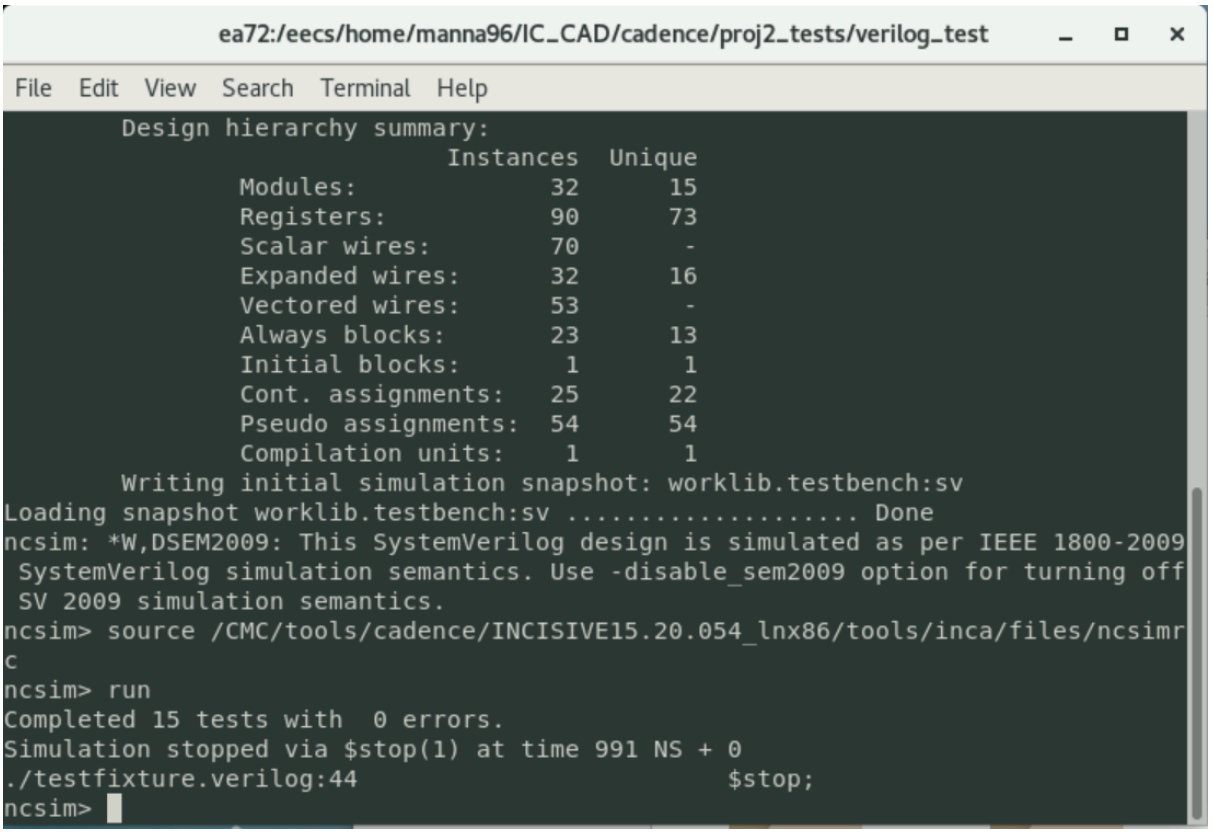


Figure 7: DRC check for Equal 0 module.

5.1 Datapath Schematic, DRC and LVS checks (Custom)

5.1.1 Equal 0 module

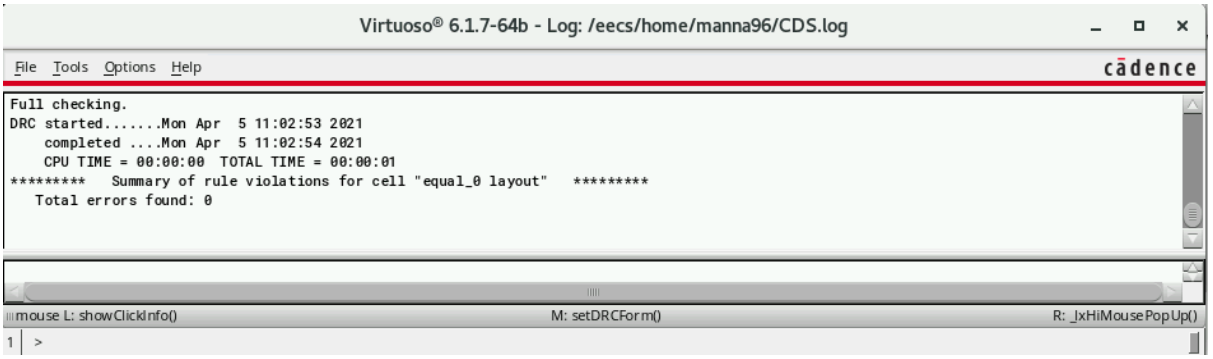


Figure 8: DRC check for Equal 0 module.

```
The net-lists match.

                          layout  schematic
                             instances
        un-matched              0       0
        rewired                 0       0
        size errors             0       0
        pruned                  0       0
        active                 38      38
        total                  38      38

                               nets
        un-matched              0       0
        merged                  0       0
        pruned                  0       0
        active                 25      25
        total                  25      25

                             terminals
        un-matched              0       0
        matched but
        different type          0       0
        total                   7       7
```

Figure 9: LVS check for Equal 0 module.: .

5.1.2 greater_than_wayhit_counter module

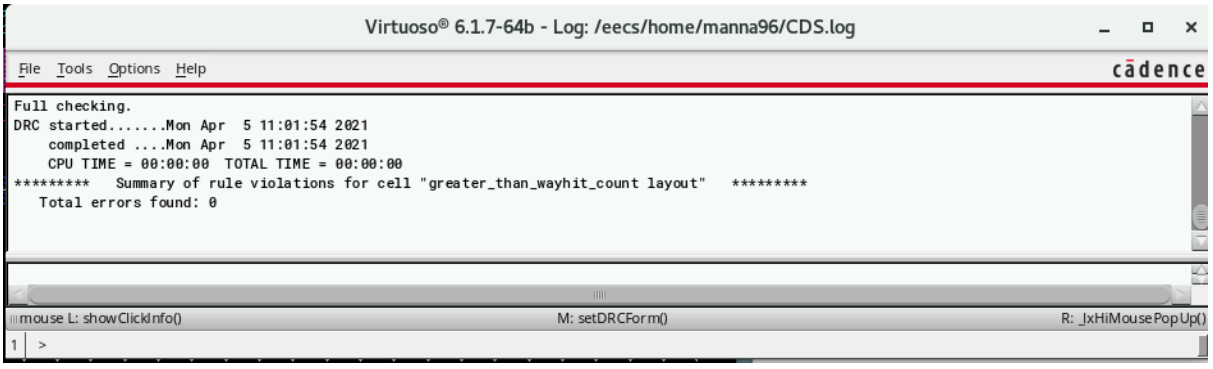


Figure 10: DRC check for greater_than_wayhit_counter module.

```
The net-lists match.

                              layout  schematic
                                 instances
        un-matched               0       0
        rewired                  0       0
        size errors              0       0
        pruned                   0       0
        active                   34      34
        total                    34      34

                                   nets
        un-matched               0       0
        merged                   0       0
        pruned                   0       0
        active                   23      23
        total                    23      23

                                 terminals
        un-matched               0       0
        matched but
        different type           0       0
        total                    7       7
```

Figure 11: LVS check for greater_than_wayhit_counter module.

5.1.3 compare_wayhit module

Below are screenshots of the comparator_wayhit_module passing schematic checks and DRC checks

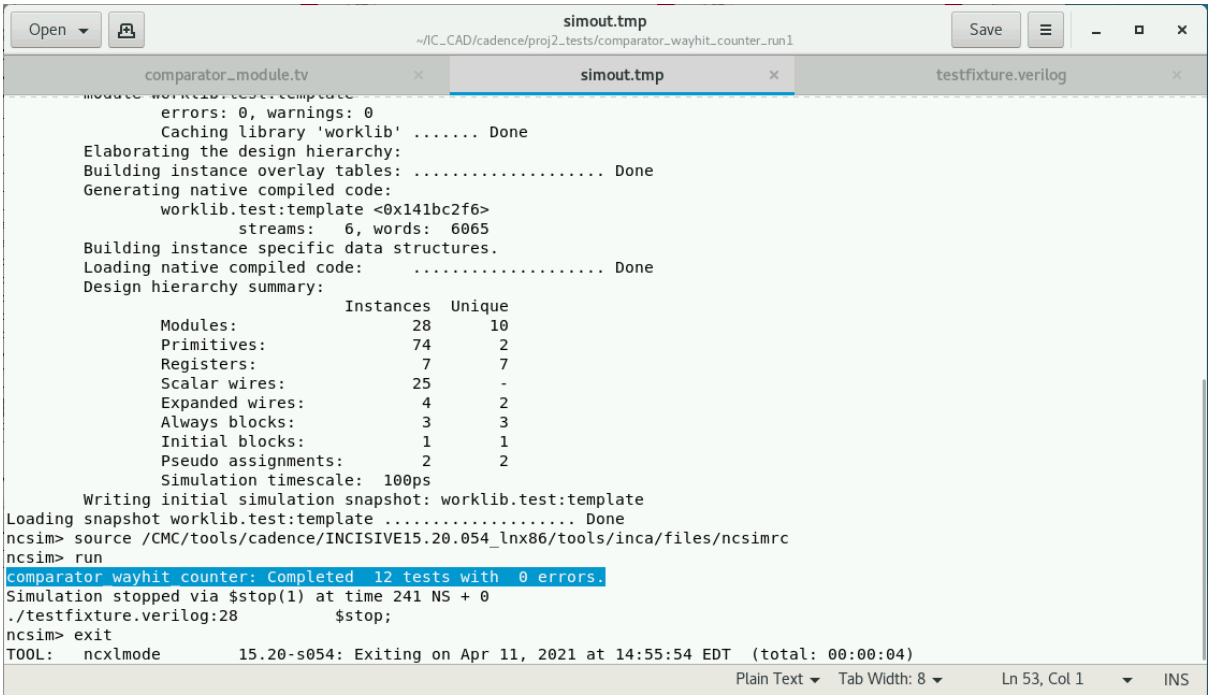


Figure 12: Schematic check for compare_wayhit module.

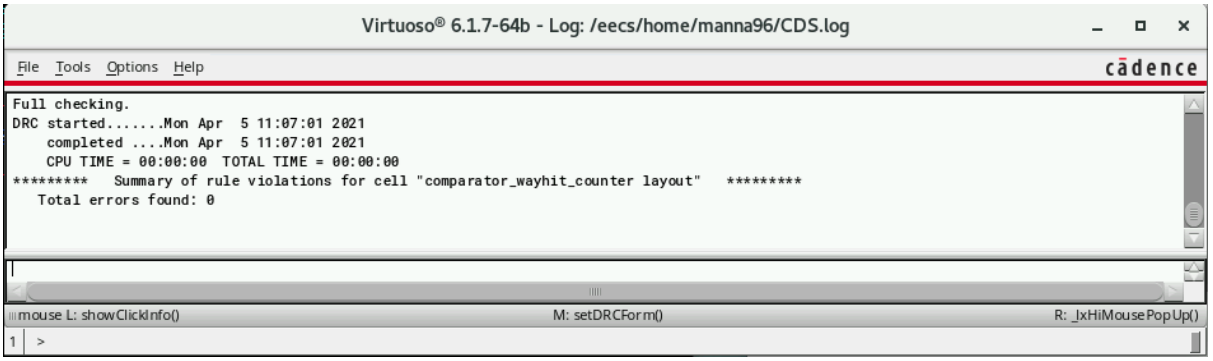
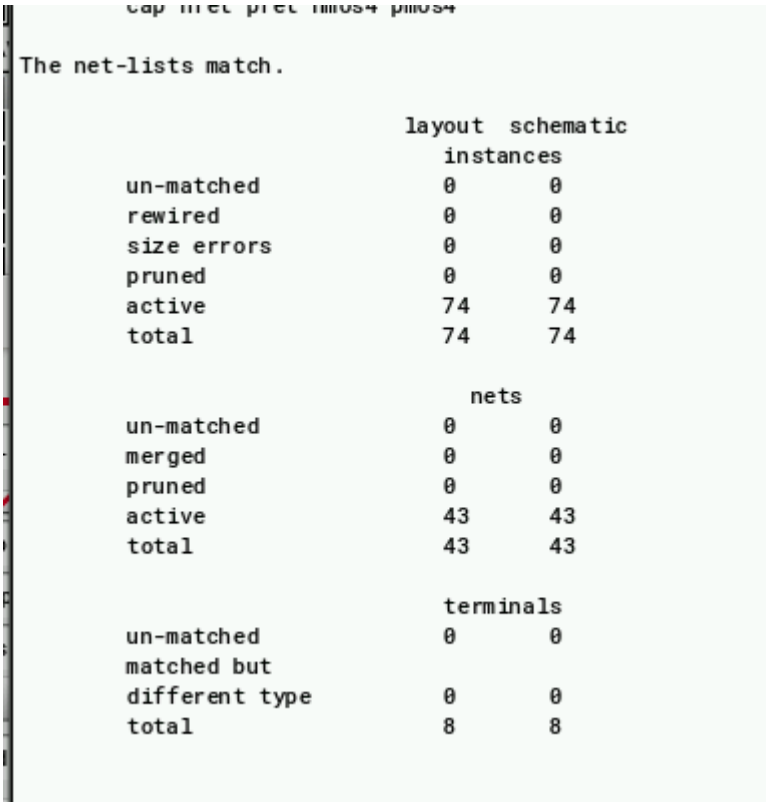


Figure 13: DRC check for the compare_wayhit module.



```
The net-lists match.

                          layout  schematic
                             instances
        un-matched             0       0
        rewired                0       0
        size errors            0       0
        pruned                 0       0
        active                74      74
        total                 74      74

                               nets
        un-matched             0       0
        merged                 0       0
        pruned                 0       0
        active                43      43
        total                 43      43

                             terminals
        un-matched             0       0
        matched but
        different type         0       0
        total                  8       8
```

Figure 14: LVS check for the compare_wayhit module.

5.1.4 lru_arithmetic_unit module

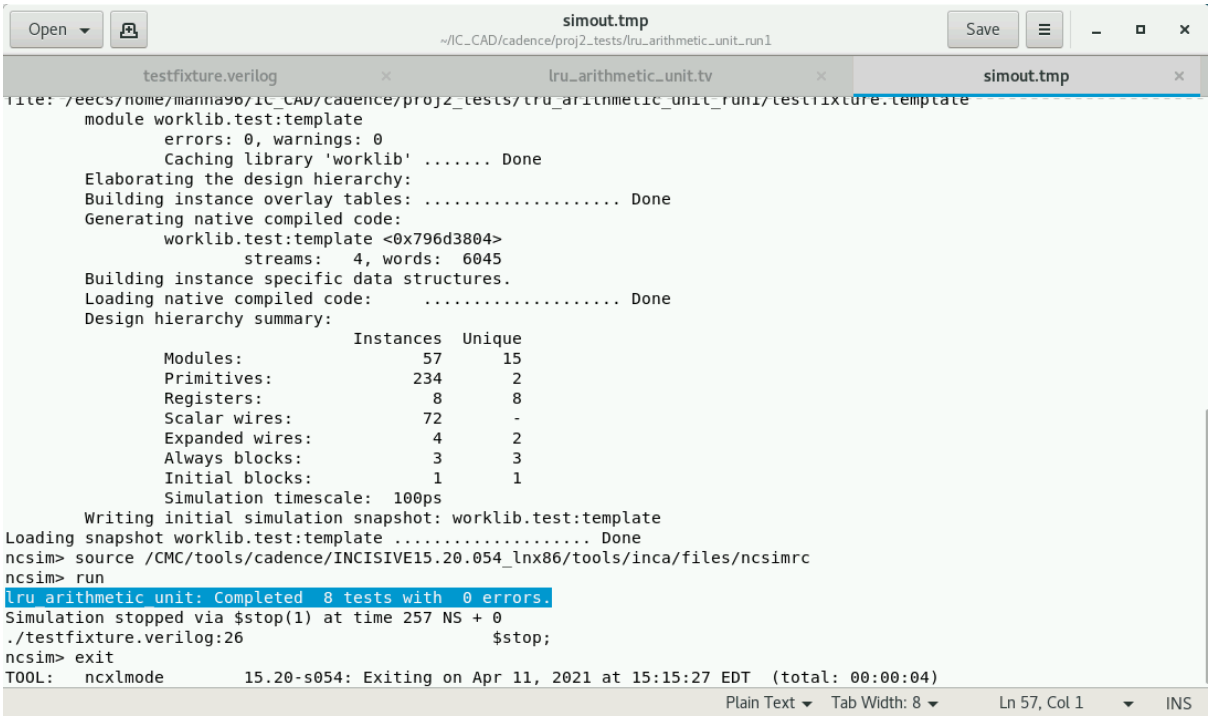


Figure 15: Schematic check for lru_arithmetic_unit module.

Figure 16: DRC check for the lru_arithmetic module.

```
The net-lists match.

                              layout  schematic
                                 instances
        un-matched                  0       0
        rewired                     0       0
        size errors                 0       0
        pruned                      0       0
        active                    234     234
        total                     234     234

                                   nets
        un-matched                  0       0
        merged                      0       0
        pruned                      0       0
        active                    138     138
        total                     138     138

                                 terminals
        un-matched                  0       0
        matched but
        different type              0       0
        total                      10      10
```

Figure 17: LVS check for the lru_arithmetic module.

5.1.5 lru_arithmetic_vector module

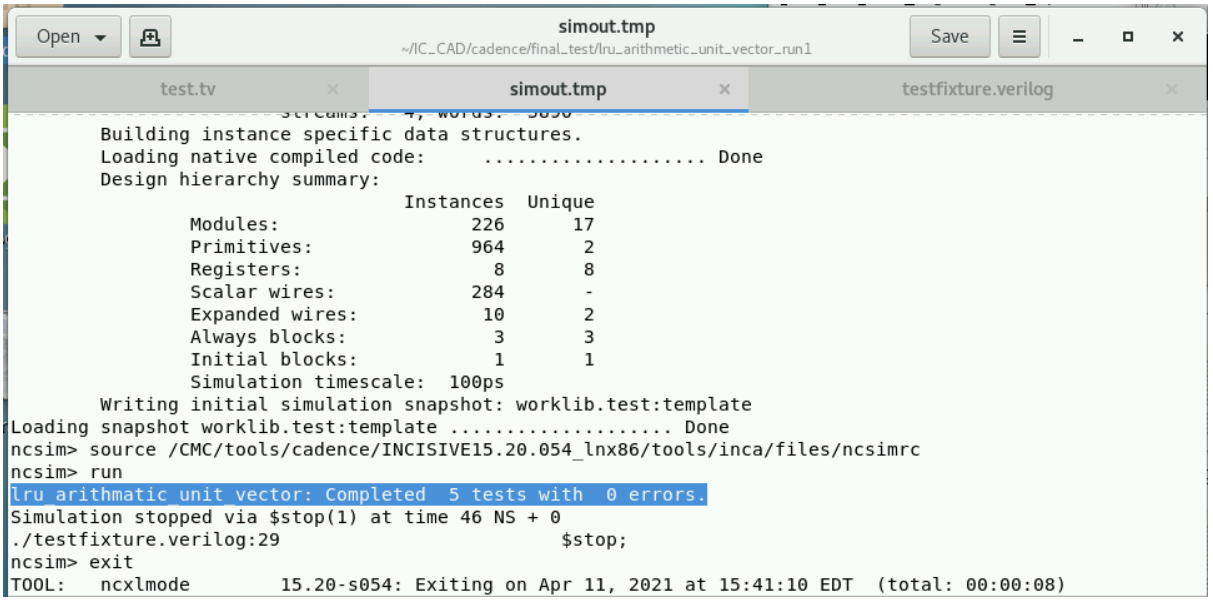


Figure 18: Schematic check for lru_artihmetic_vector module.

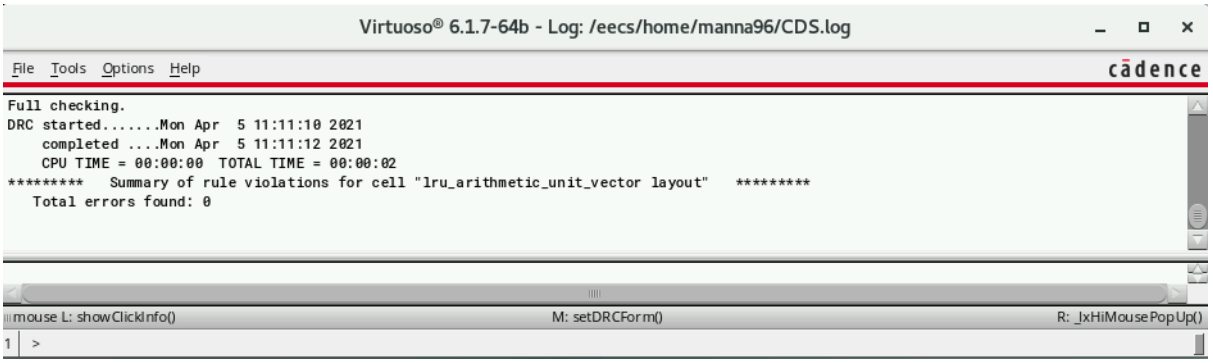


Figure 19: DRC check for the lru_arithmetic_vector module.

```
The net-lists match.

                          layout  schematic
                             instances
        un-matched           0       0
        rewired              0       0
        size errors          0       0
        pruned               0       0
        active               1020    1020
        total                1020    1020

                               nets
        un-matched           0       0
        merged               0       0
        pruned               0       0
        active               589     589
        total                589     589

                             terminals
        un-matched           0       0
        matched but
        different type       0       0
        total                23      23
```

Figure 20: LVS check for the lru_arithmetic_vector module.

5.1.6 datapath module

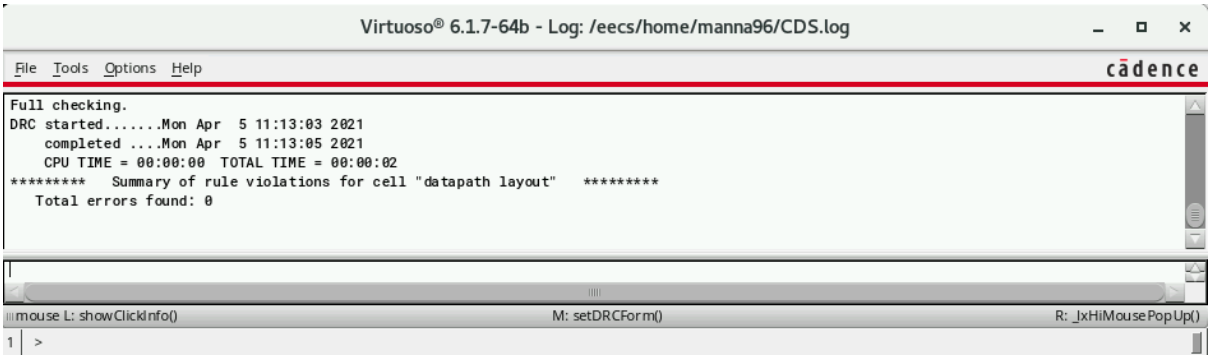


Figure 21: DRC check for the Datapath module.

```
The net-lists match.

                          layout  schematic
                             instances
        un-matched               0       0
        rewired                  0       0
        size errors              0       0
        pruned                   0       0
        active                1334    1334
        total                 1334    1334

                               nets
        un-matched               0       0
        merged                   0       0
        pruned                   0       0
        active                 751     751
        total                  751     751

                             terminals
        un-matched               0       0
        matched but
        different type           0       0
        total                   28      28
```

Figure 22: LVS check for the Datapath module.

5.2 Controller DRC and LVS checks

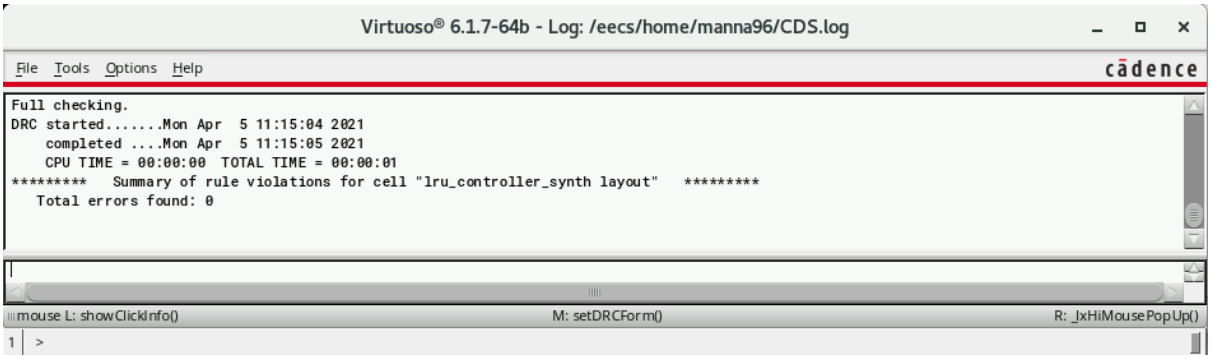


Figure 23: DRC check for the synthesised controller module.

```
The net-lists match.

                           layout  schematic
                              instances
        un-matched              0       0
        rewired                 0       0
        size errors             0       0
        pruned                  0       0
        active                288     288
        total                 288     288

                                nets
        un-matched              0       0
        merged                  0       0
        pruned                  0       0
        active                155     155
        total                 155     155

                              terminals
        un-matched              0       0
        matched but
        different type          0       0
        total                  13      15
```

Figure 24: LVS check for the synthesised controller module.

5.3 Single Port Ram DRC and LVS check

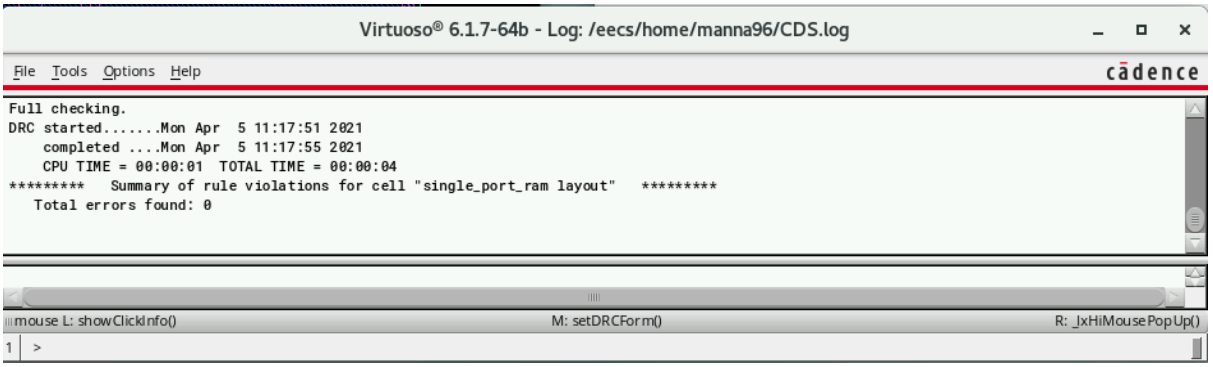


Figure 25: DRC check for the single port ram module.

```
The net-lists match.

                          layout  schematic
                             instances
        un-matched           0       0
        rewired              0       0
        size errors          0       0
        pruned               0       0
        active               3006    3006
        total                3006    3006

                               nets
        un-matched           0       0
        merged               0       0
        pruned               0       0
        active               1650    1650
        total                1650    1650

                             terminals
        un-matched           0       0
        matched but
        different type       0       0
        total                24      26
```

Figure 26: LVS check for the single port ram module.

5.4 Core Schematic check

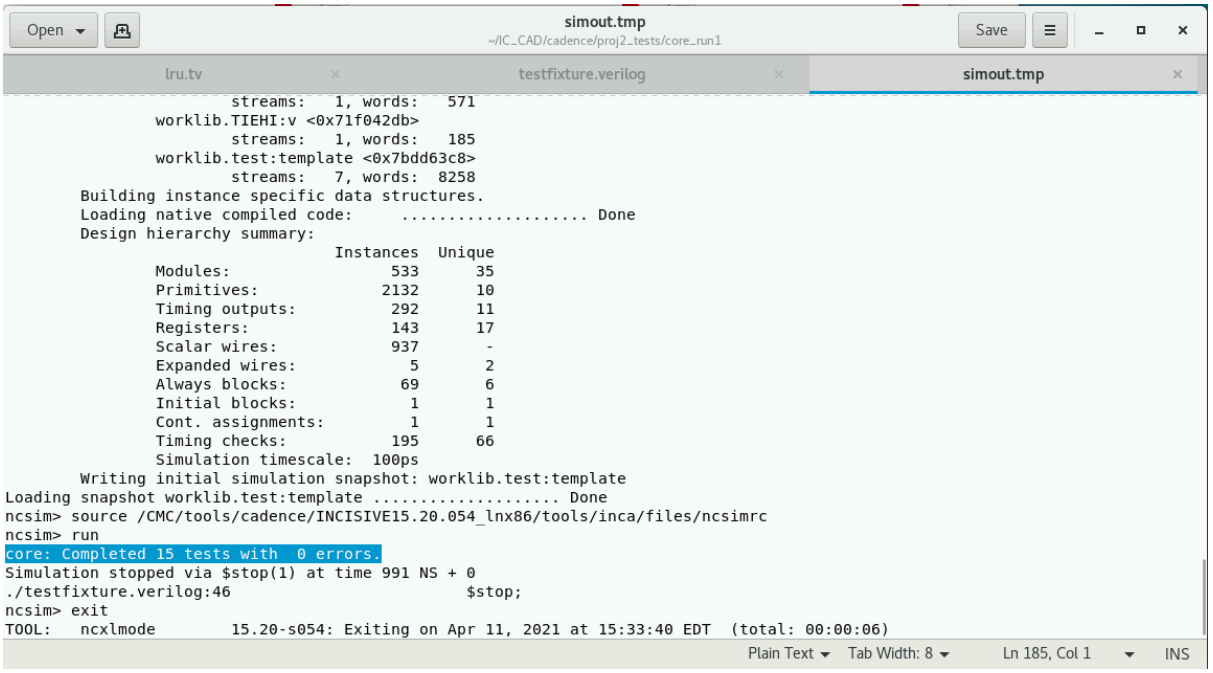


Figure 27: Schematic check for Core module.

5.5 Padframe Schematic, DRC and LVS check

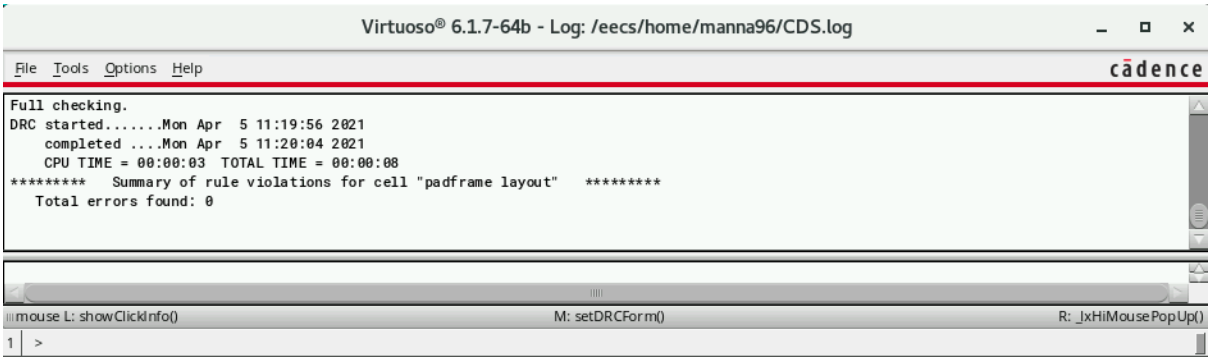


Figure 28: DRC check for the padframe.

```
The net-lists match.

                          layout  schematic
                             instances
        un-matched              0       0
        rewired                 0       0
        size errors             0       0
        pruned                  0       0
        active                  1792    402
        total                   1792    402

                               nets
        un-matched              0       0
        merged                  0       0
        pruned                  0       0
        active                  175     175
        total                   175     175

                             terminals
        un-matched              0       0
        matched but
        different type          6       6
        total                   46      48
```

Figure 29: LVS check for the padframe.

5.6 Chip DRC and LVS check

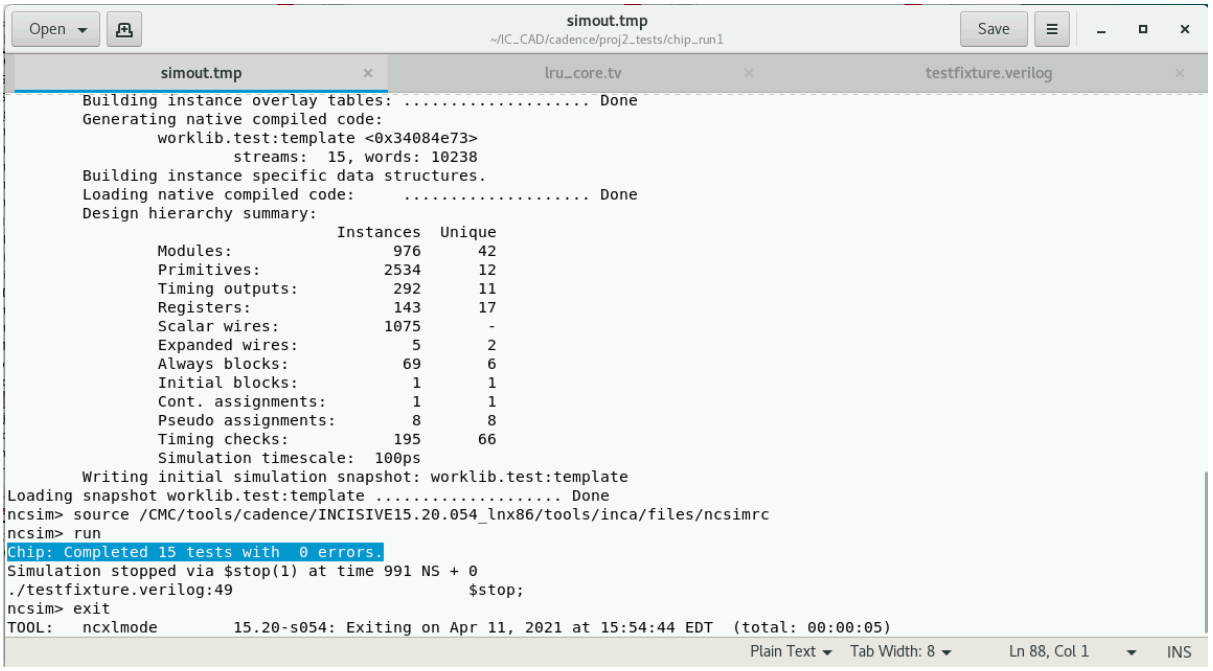


Figure 30: Schematic check for chip module.

Figure 31: DRC check for the chip.

```
The net-lists match.

                          layout  schematic
                             instances
        un-matched              0       0
        rewired                 0       0
        size errors             0       0
        pruned                  0       0
        active                  6420    5030
        total                   6420    5030

                               nets
        un-matched              0       0
        merged                  0       0
        pruned                  0       0
        active                  2676    2676
        total                   2676    2676

                             terminals
        un-matched              0       0
        matched but
        different type          0       0
        total                   0       25
```

Figure 32: LVC check for the Chip.

6. Post Fabrication test plan

Once the chip has been fabricated, it would be placed on a custom test board that would provide a socket for the chip. This board would have a source and sink where the source would provide the necessary input values as specified by the tester. Given the necessary inputs, the output pins of the board would then represent the output of the chip of which can be verified by an oscilloscope.

7. Design Time

7.1 Verilog Construction and Planning (13-15 hours):

The significance of the verilog code is to behaviorally construct what the final chip is supposed to do and simulate how it should handle its inputs and outputs. As such, it serves as the foundation on how the overall design is created as without it, schematics and layouts cannot be created or tested. Because of its importance, it took a considerable amount of time to create all the necessary modules and figure out how each of them are connected. Thus, this portion of the project took roughly 7-8 hours to complete.

Once the verilog code was constructed extensive testing needed to be done to ensure that the verilog code accurately simulated what we intended to build. Even though this testing phase was completed and it seemed that the verilog code initially passed all tests in the earlier stages of the project, it would continuously be revisited and modified as the later stages progressed in order to rectify any bugs or design changes. As such, additional testing of the verilog code needed to be done. Overall, the group dedicated roughly 6 hours over the course of the project

7.2 LRU_Arithmetic unit - custom (8-10 hours):

The LRU_Arithmetic unit was composed of several sub components, which was mentioned earlier in the 3.4 Datapath. The schematics and layouts for the subunits were relatively small thus taking only 1-2 hours to complete and test all. Since this was the first complex unit with multiple subunits, the team's inexperience in creating layouts affected the final layout of the unit. One of the biggest issues was not aligning the inputs for subunits that shared the same inputs. As such, multiple metals had to be used to connect the shared inputs together which caused a lot of overlapping issues and resulted in many DRC and LVS related design violations. The learning outcome was that the individual component layouts must be thoroughly preplanned from low level components to ensure that the hierarchical layouts of the proceeding components can be managed well.

Another issue that was faced was to create a constant high bit or a low bit that could be fed into some of the components, namely the half subtractor unit and the 2x1 mux that produced the 1-bit output (as seen on figure _xx). The half substactor required a constant high bit as the counter values were subtracted by 1 in certain cases. The 2x1 mux required a high bit as the output to indicate that unit updated the counter value to 3 and a low bit otherwise. Initially, VDD was assigned to the first half subtractor and d0 on the 2x1 mux while GND was assigned to d1 on the 2x1 mux. This method was successful in passing DRC and LVS testing in the schematic process but was proved to not be possible in the layout construction. Since the power rails are Input/output pins, it was impossible to create another vdd pin that was defined only as an input pin which caused our DRC and LVS to fail. To mitigate the issue of having a constant high bit without using VDD, a mux was used where the input

signal was the same as the select signals. By attaching an inverter to the d0 input of the mux, a high bit was always passed. This can be seen through section 9.4.4 figure 39 and 40.

The initial schematic design without the above mentioned update took 3-4 hours to complete as all the subunits needed to be created, tested and combined. Thorough testing took roughly less than an hour to ensure all test cases were passed.
The layout and the final design took a total of 5-6 hours to complete. Most of the time spent on figuring out a solution for the above mentioned issue of producing a constant high and low bit.

7.3 LRU_Arithmetic Vector - custom (8-10 hours):

The custom module layout that took the longest to design was the lru_arithmatic_vector as it required stacking 4 units of the LRU_Arithmetic modules mentioned above and adding additional latches. One of the biggest issues that was faced when designing this module was not pre-planning the design to match all the components. This caused a lot of the components to be misaligned which increased the complexity of the layout.The learning outcome from this unit was to not stack the two muxes on the LRU_Arithmentic unit vertically but rather lay them out horizontally. The vertical muxes can be seen through section 9.4.6 figure 44. Even though the immediate routing would have been complex the hierarchical layouts for the preceding layouts would have been much easier to route and would be symmetrical in nature, including this LRU_Arithmetic vector module.

The construction of the schematic took around 1-2 hours to ensure the I/O's are connected correctly. Testing took less than an hour to test the unit to ensure it passed all testbench cases.The construction of the layout took more than 7-8 hours in total to complete the unit and pass DRC and LVS.

7.4 Datapath (~ 3 hours):

The datapath module included the LRU_Arithmetic vector and a few latches. As you can see from the layout in section 9.4.7 figure 46, the latches on the two ends of the layout do not match with the 4 LRU_Aritnmentic units stacked in the middle. The offsets caused by the mismatch forced non symmetrical wires to be added to the I/O pins and power rails for the units. As previously mentioned the issue was that the pre-planning was done poorly. Due to time constraints and nearing deadlines, the decision was made to complete the design as it is instead of redesigning all the lower level components.

The schematic only a few other smaller components needed to be added to complete the datapath schematic so the process was roughly 1-2 hours. Testing took less than an hour to ensure all test cases passed.The layout took roughly 2-4 hours to complete as each I/O in was needed to be routed through a complex web of wires as can be seen through layout in section 9.4.7 figure 45.

7.5 Controller - synthesised (< 1 hour):

As mentioned in Section 3.2 the layout and schematic of the controller was synthesized using a design compiler and virtuoso. The synthesis was straightforward and only took roughly 30 mins to complete, schematic and layout. To test the controller, various test vectors were used to ensure that the states of the FSM were changing accordingly by looking at the resulting waveform simulations in which did not take long to complete.

7.6 RAM - synthesised (~ 1 week):

The ram synthesis was the component that took the longest to complete due to issues with importing into virtuoso. The cause of LVS errors was due to the abstract library names not being replaced by the actual library names during the import. This caused a huge set back in our time line. However, an alternative design, excluding the ram, was created in order to have a back-up plan. The backup plan was to directly send in the counter values to our algorithm to be processed instead of it coming from the ram model as initially intended. In this case, the 8-bits value consisting of the four 2-bit counter values would be sent to the system directly and keeping track of all counters would then be outside the scope of the system. However, since the issue was resolved, which took around a week, this backup approach was not used in the end.

While the verilog synthesis was rather simple and fast as there was only minimal configuration for Design Compiler. The layout however, took a lot longer to complete from innovus. This is due to the filler cells having blockages causing geometric failures despite these geometric failures not affecting the component once exported to virtuoso. Once the virtuoso import issues were resolved, it took roughly less than half an hour to complete the ram synthesis. Testing was conducted to ensure that given a specified set_index and a write-enable signal, the module would return the correct set of counters when being read and also overwrite the correct existing counter set while being written to.

7.7 Core (< 0.5 hours):

The core included the three major components in our chip, namely the datapath, controller and ram modules. This was a rather easy task; only the wire connections were missing that link each module. Once the connections were completed, the test vectors used for the verilog test were used for testing this. Overall the core took less than 30 minutes to complete

7.8 Padframe - custom (~ 1 hour):

The construction of the padframe did not take long as the padframe from the mips8 library was used as a template. The only modification that was required was to change the I/O pins and map all the necessary inputs and outputs. Since our design had less I/O pins, the unused I/O pins were replaced with GND and VDD pins. Overall, the schematic and layout

took roughly an hour to complete and testing the padframe would commence when the chip schematic and layout were finished.

7.9 Chip (~ 3 hours)

Since all the major components were completed (datapath, controller, and ram) only the module interconnections were left to complete. By using the core module as reference the padframe schematic was connected to the chip with minimal effort.

The layout had an issue where the auto routing would not fully route all the I/O pins. After several hours of debugging the only solution was to remove all the schematic I/O wires and connect only a handful of wires (maximum 10 wires) and then do an auto route on the layout. This ensured that auto routing was not overwhelmed with the number of I/O pins. While the schematic took less than an hour to complete, it took 1-2 hours to finish the layout due to the issues mentioned above. Once the layout passed all DRC and LVS checks, testing of the chip was straightforward as similar test vectors from the core and the verilog simulation were used.

8. File Locations

All project files are in a zip folder named *EECS4612_group1_proj2_files*.
The project library and all of its cells are in a sub-folder named *proj2_lib* in the submitted package.

The initial and final lru_chip architecture diagrams mentioned in section 4 are in two jpeg files in the submitted package named:

- *Initial_lru_chip_architecture.jpeg*
- *Final_lru_chip_architecture.jpeg.*

verilog_test is a sub-folder in the submitted package that contains the three files that are required to test the verilog code. Simply open a terminal on the verilog_test director and run sim-nc lru_unit.sv.

The chip plot can be found in the root folder of the submitted package named *Chip_plot.pdf.*

9. Appendix

9.1 Verilog code with testbench module

```
typedef enum logic [2:0] {STATE_RESET, STATE_WAIT, STATE_READ_MEMORY, STATE_CALCULATE_ALG,
STATE_WRITE_MEMORY, STATE_DONE} statetype;


module testbench();

    logic clk;
    logic clk2;
    logic reset;
    logic val_i;
    logic rdy_i;
    logic h_m;
    logic [2:0] set_index;
    logic [1:0] cache_hit_index;
    logic val_o;
    logic rdy_o;
    logic [1:0] wayindex;
    logic [7:0] outputdata;

        // The device under test
        //lru_datapath dut(inputData, hit_miss_signal, way_hit_index, clk, outputdata, wayindex);
    LRU_unit dut(clk, clk2, reset, val_i, rdy_i, h_m, set_index, cache_hit_index, val_o, rdy_o,
wayindex, outputdata);

    `include "testfixture.verilog"

endmodule


module LRU_unit #(parameter WIDTH=8)
        (input logic clk, clk2, reset,
        //input (request) side
        input logic val_i,
        input logic rdy_i,
        input logic h_m,
        input logic [2:0] set_index,
        input logic [1:0] cache_hit_index,

        //output (response) side
        output logic val_o,
        output logic rdy_o,
        output logic [1:0] wayindex,
        output logic [WIDTH-1:0] outputdata);

        //Command signals
        logic we; //write enable
        logic output_rdy; //output is done writing to the memory and is ready to output

        //Status signals
        logic read_done; //the data is read from the memory and is ready to go in to the alg
        logic output_data_rdy; //the data is ready to be outputted
        logic output_wayindex_rdy; //the way index is ready to be outputted
        logic write_done; //output data was written to memory


        //Control
        lru_controller_synth controller (.*);

        //Datapath
        lru_datapath datapath (.*);

endmodule
```

```
module lru_controller_synth  (input clk, clk2, reset,
                input val_i,
                output rdy_o,
        output val_o,
        input rdy_i,
                output logic we,
        output logic output_rdy,
        input logic read_done,
        input logic output_data_rdy,
        input logic output_wayindex_rdy,
        input logic write_done);

        statetype state;


        //statelogic controls the fms
        statelogic sl(clk, clk2, reset, val_i, rdy_o, val_o, rdy_i, read_done, output_data_rdy,
        output_wayindex_rdy, write_done, state);
        //outputlogic controls the output values of each state
        outputlogic ol(state, rdy_o, val_o, we, output_rdy);

endmodule

module statelogic (input logic clk, clk2, reset,
        input logic val_i,
        input logic rdy_o,
        input logic val_o,
        input logic rdy_i,
                input logic read_done,
                input logic output_data_rdy,
        input logic output_wayindex_rdy,
                input logic write_done,
        output statetype state);


        statetype nextstate;
        //logic req_go;
        logic is_read_dn;
        logic is_alg_dn;
        logic is_write_dn;
        logic resp_go;
        logic [2:0] ns, state_logic;


        //mux to select the next state when the reset bit is low
        mux2 #(3) resetmux(nextstate, STATE_RESET, reset, ns);

        //flip flop to put the nextstate into the register
        statereg #(3) sreg(clk, clk2, ns, state_logic);


        assign req_go = val_i && rdy_o; //val_i indicates that the LRU is ready to take in a value
and rdy_o indicates that its ready to receive a new output
        assign is_read_dn = read_done; //the data is read from the memory and is ready to be fed to
the lru algorithm
        assign is_alg_dn = output_data_rdy && output_wayindex_rdy; //output data and the wayindex
is ready from the algorithm
        assign is_write_dn = write_done; //is it done writing new data to the memory
        assign resp_go = val_o && rdy_i; //sink is ready to read the output and the lru is ready to
send the output
        assign state = statetype'(state_logic);

        always_comb
```

```
        begin
        case (state)
        STATE_RESET:    if(!reset)      nextstate = STATE_WAIT;
        STATE_WAIT:             if(req_go)      nextstate = STATE_READ_MEMORY;
                else    nextstate = STATE_WAIT;
        STATE_READ_MEMORY:      if(is_read_dn)  nextstate = STATE_CALCULATE_ALG;
                else    nextstate = STATE_READ_MEMORY;
        STATE_CALCULATE_ALG:    if(is_alg_dn)   nextstate = STATE_WRITE_MEMORY;
                else    nextstate = STATE_CALCULATE_ALG;
        STATE_WRITE_MEMORY:     if(is_write_dn) nextstate = STATE_DONE;
                else    nextstate = STATE_WRITE_MEMORY;
        STATE_DONE:             if(resp_go)     nextstate = STATE_WAIT;
                else    nextstate = STATE_DONE;
        endcase
        end

endmodule

module outputlogic (input statetype state_logic,
        output logic rdy_o,
        output logic val_o,
        output logic we,
        output logic output_rdy);

        task cs (input logic cs_rdy_o,
        input logic cs_val_o,
        input logic cs_we,
        input logic cs_output_rdy);

        begin
        rdy_o           = cs_rdy_o;
        val_o           = cs_val_o;
        we              = cs_we;
        output_rdy      = cs_output_rdy;
        end
        endtask

        //state input structure
        /*
                 rdy_o  val_o   we      output_recived
        WAIT            1       0       0       0
        READ_MEMORY   0         0       0       0
        CALC            0       0       0       1
        WRITE_MEMORY  0         0       1       0
        DONE            0       1       0       0
        */

        always @ (*)
        begin
        cs(0, 0, 0, 0); //default settings
        case (state_logic)
        STATE_WAIT:             cs(1, 0, 0, 0);
        STATE_READ_MEMORY:      cs(0, 0, 0, 0);
        STATE_CALCULATE_ALG:    cs(0, 0, 0, 1);
        STATE_WRITE_MEMORY:     cs(0, 0, 1, 0);
        STATE_DONE:             cs(0, 1, 0, 0);
        endcase
        end

endmodule


module statereg #(parameter WIDTH=4)
```

```
                (input logic clk,clk2,
                input  logic [WIDTH-1:0] d,
                output logic [WIDTH-1:0] q);

        logic [WIDTH-1:0] mid;

        latch #(WIDTH) master(clk2, d, mid);
        latch #(WIDTH) slave(clk, mid, q);

endmodule

module mux2 #(parameter WIDTH = 4)
        (input logic [WIDTH-1:0] d0, d1,
        input logic s,
        output logic [WIDTH-1:0] y);

        assign y = s ? d1 : d0;
endmodule

module latch #(parameter WIDTH = 8)
                (input  logic           ph,
                input  logic [WIDTH-1:0] d,
                output logic [WIDTH-1:0] q);

        always_latch
        if (ph) q <= d;
endmodule


module lru_datapath (input logic clk2, reset,
                input logic [2:0] set_index,
                input logic [1:0] cache_hit_index,
                input logic we,
                input logic output_rdy,
                input logic h_m,
                output logic output_data_rdy,
        output logic output_wayindex_rdy,
                output logic read_done,
                output logic write_done,
                output logic [1:0] wayindex,
                output logic [7:0] outputdata);


        logic [7:0] ram_output_counter_values; //store the counter values that are read from the
ram.
        logic [7:0] counter_values;  //counter value read from register after the ram stores it in
the register.
        logic [7:0] updated_counters; //counter values that were updated through the LRU arithmetic
unit.
        logic [7:0] updated_counters_post_register; //updated counter values that were saved in the
register
        logic [1:0] updated_wayindex; //way index that was outputted by the LRU arithmetic unit
        logic [1:0] updated_wayindex_post_register; //way index value that were saved in the
register to be outputted


        //read or write into memory
        single_port_ram spr(clk2, we, reset, updated_counters_post_register, set_index, read_done,
write_done, ram_output_counter_values);

        //call arithmetic unit with counter_values gotten from the register
        lru_arithmetic_unit lru_arith(counter_values, cache_hit_index, h_m, updated_counters,
updated_wayindex);
```

```
        //output latches
        latch #(8) output_reg_counter(~we, updated_counters, updated_counters_post_register);
        latch #(2) index(output_rdy, updated_wayindex, updated_wayindex_post_register);

        //input latches
        latch #(8) input_reg_counter(read_done, ram_output_counter_values, counter_values);


        assign output_data_rdy = output_rdy ? 1'b1 : 1'b0;
        assign outputdata = ram_output_counter_values;
        assign output_wayindex_rdy = output_rdy ? 1'b1 : 1'b0;
        assign wayindex = updated_wayindex_post_register;

endmodule

module single_port_ram (input logic clk,  we,
        input logic reset,
        input logic [7:0] updated_counter_set,
                input logic [2:0] set_index,
        output logic read_done,
                output logic write_done,
                output logic [7:0] q);

        // Declare the RAM variable
        logic [7:0] ram[7:0];


        //initialize all rows in memory to 00011011 when the reset flag is up.
        integer i;
        always @(posedge clk)
        begin
        if (reset)
        begin
                for(i=0; i<8; i=i+1)
                begin
                ram[i] <= 8'b00011011;
                end
        end
        else if (we)
        begin
                ram[set_index] <= updated_counter_set; //set the new data into the ram given the
row (set_index)
        end
        end

        //on the negative edge of the clock mark the read_done as high when write enable is low.
This is to ensure that ram[set_index] produced the               proper value in time
        always @ (negedge clk)
        begin
        read_done <= 1'b0;
        if (!we)
        begin
                read_done <= 1'b1;
        end
        else
        begin
                read_done <= 1'b0;
        end
        end


        assign q = (!we) ? ram[set_index] : 8'b00000000; //only output when write enable is low
```

```
        assign write_done = we ? 1'b1 : 1'b0; // if write enable is high then set the write_done
bit high that will be sent to the controller

endmodule

module lru_arithmetic_unit (input logic [7:0] counter_values,
                input logic [1:0] cache_hit_index,
                input logic h_m,
                output logic [7:0] outputData,
                output logic [1:0] wayIndex);


        //split the data into its 4 different ways
        wire [1:0] way_3 = counter_values[7:6];
        wire [1:0] way_2 = counter_values[5:4];
        wire [1:0] way_1 = counter_values[3:2];
        wire [1:0] way_0 = counter_values[1:0];

        wire[7:0] is_equal_zero;
        wire[7:0] outputMuxData;
        wire[3:0] outputMuxIndex;

        wire[7:0] select;

        //call the compare_to_zero module to see if the counter value is 0 or not for all for ways
        compare_to_zero equal1(.counterValue (way_0), .is_equal_zero (is_equal_zero[1:0]));
        compare_to_zero equal2(.counterValue  (way_1), .is_equal_zero (is_equal_zero[3:2]));
        compare_to_zero equal3(.counterValue  (way_2), .is_equal_zero (is_equal_zero[5:4]));
        compare_to_zero equal4(.counterValue (way_3), .is_equal_zero (is_equal_zero[7:6]));

        wire [1:0] way_hit_counter;
        wire [7:0] greater_than_way_hit_counter_select;

        //if its a cache hit, get the counter value of that way that was hit
        get_way_hit_counter g1(counter_values, cache_hit_index, way_hit_counter);

        //compare the cache hit way counter value with all other counters to determine if its
greater than the cache hit way counter value so they can          be decremented by -1.
        compare_counter_to_wayhit_counter compare1(way_0, way_hit_counter,
greater_than_way_hit_counter_select[1:0]);
        compare_counter_to_wayhit_counter compare2(way_1, way_hit_counter,
greater_than_way_hit_counter_select[3:2]);
        compare_counter_to_wayhit_counter compare3(way_2, way_hit_counter,
greater_than_way_hit_counter_select[5:4]);
        compare_counter_to_wayhit_counter compare4(way_3, way_hit_counter,
greater_than_way_hit_counter_select[7:6]);

        //2v1 mux to get the appropriate select value which will determine if its value is updated
to 3, -1 or stays the same.
        mux2 #(2) getway0sel(is_equal_zero[1:0], greater_than_way_hit_counter_select[1:0], h_m,
select[1:0]);
        mux2 #(2) getway1sel(is_equal_zero[3:2], greater_than_way_hit_counter_select[3:2], h_m,
select[3:2]);
        mux2 #(2) getway2sel(is_equal_zero[5:4], greater_than_way_hit_counter_select[5:4], h_m,
select[5:4]);
        mux2 #(2) getway3sel(is_equal_zero[7:6], greater_than_way_hit_counter_select[7:6], h_m,
select[7:6]);

        //3x1 mux that will update the counter values to their appropriate value given the select
from above.
        mux3 h1(way_0, select[1:0], outputMuxData[1:0], outputMuxIndex[0]);
        mux3 h2(way_1, select[3:2], outputMuxData[3:2], outputMuxIndex[1]);
        mux3 h3(way_2, select[5:4], outputMuxData[5:4], outputMuxIndex[2]);
```

```
        mux3 h4(way_3, select[7:6], outputMuxData[7:6], outputMuxIndex[3]);

        //4x2 encoder to output the way index that was updated to 3
        always @ (outputMuxIndex)
        begin
                case (outputMuxIndex)
                4'b0001: wayIndex = 2'b00;
                4'b0010: wayIndex = 2'b01;
                4'b0100: wayIndex = 2'b10;
                4'b1000: wayIndex = 2'b11;
                endcase
        end

        //output the new counter values
        assign outputData = outputMuxData;

endmodule

module get_way_hit_counter (input logic [7:0] inputData,
                input logic [1:0] way_hit_index,
                output logic [1:0] way_hit_counter);


        always @ (*)
        begin
        case(way_hit_index)
        2'b00 : way_hit_counter = inputData[1:0];
        2'b01 : way_hit_counter = inputData[3:2];
        2'b10 : way_hit_counter = inputData[5:4];
        2'b11 : way_hit_counter = inputData[7:6];
        endcase
        end


endmodule

module compare_counter_to_wayhit_counter(input logic [1:0] compare_inputData,
                        input logic [1:0] way_hit_counter,
                        output logic [1:0] greater_than_way_hit_counter );

        wire [1:0] select;

        assign select[0] = compare_inputData > way_hit_counter; //first bit represents if the
counter value is greater than the counter value that the cache was hit
        assign select[1] = compare_inputData == way_hit_counter; //second bit represents if the
counter value is equal to that of the counter value that was the cache hit

        //depending on the above scenario, the greate_than_way_hit_counter is assigned a bit that
will act as the select for the mux3.
        always @ (*)
        begin
        case(select)
        2'b00: greater_than_way_hit_counter = 2'b10;
        2'b01: greater_than_way_hit_counter = 2'b00;
        2'b10: greater_than_way_hit_counter = 2'b11;
        endcase
        end

endmodule

module compare_to_zero (
        input logic [1:0] counterValue,
        output wire [1:0] is_equal_zero);
```

```
        //check the counterValue if its equal to 0. Two bits are needed due to this acting as the
mux3 select.
        assign is_equal_zero[0] = ~| counterValue;
        assign is_equal_zero[1] = ~| counterValue;


endmodule

module mux2 #(parameter WIDTH = 4)
        (input logic [WIDTH-1:0] d0, d1,
        input logic s,
        output logic [WIDTH-1:0] y);

        assign y = s ? d1 : d0;
endmodule


module mux3 (input logic [1:0] countValue,
        input logic [1:0] select,
        output logic [1:0] outputMuxData,
        output logic outputMuxwayindex);



        always @ (*)
        begin
        case (select)
        2'b11:
        begin
                outputMuxData = 2'b11; //update the counter value to 3.
                outputMuxwayindex = 1'b1; //set the bits high for the way that was updated to 3
        end
        2'b00:
        begin
                outputMuxData = countValue - 1'b1; //decrement the counter value by -1
                outputMuxwayindex = 1'b0; //set the bits low for the way that was decremented by -1
        end
        2'b10:
        begin
                outputMuxData = countValue; //the counter value stays the same.
        end
        endcase
        end
endmodule
```

9.2 testfixture code for testing the above verilog code

```
logic [16:0] vectors[20:0], currentvec;
logic [5:0] vectornum, errors;
logic testbench_clk;

initial
begin
    $readmemb("lru.tv", vectors); // put your test vectors into the vectors variable
    vectornum = 0; errors = 0;
```

```
end

always begin
    testbench_clk = 0; #1; testbench_clk = 1; #16; testbench_clk = 0; #16;
end


always
begin
    clk <= 0; clk2 <= 0; #1;
    clk <= 1; # 4;
    clk <= 0; #2;
    clk2 <= 1; # 4;
end

always @(posedge testbench_clk)
begin
    currentvec = vectors[vectornum];
    reset = currentvec[10];

    if (rdy_o === 1'b1)
    begin
         val_i = 1;
         rdy_i = 0;
         set_index = currentvec[16:14];
         cache_hit_index = currentvec[13:12];
         h_m = currentvec[11];
         end
    else
    begin
         val_i = 1'b0;
    end

    if (currentvec[10] === 1'bx) begin // if the output is unknown (x), you have finished the test
         $display("Completed %d tests with %d errors.",
         vectornum, errors);
         $stop;
    end
end


always @(negedge testbench_clk) begin

    if(val_o === 1'b1)
    begin

         rdy_i = 1;
         val_i = 0;

         if((outputdata !== currentvec[9:2] && wayindex !== currentvec[1:0]))
         begin
                 $display("Error: inputs were set_index=%b", currentvec[16:14]);
                 $display("     updated_counters = %b wayindex =%b (%b %b expected)",
                 outputdata, wayindex,currentvec[9:2], currentvec[1:0]);
                 errors = errors + 1;
         end

         vectornum = vectornum + 1;

    end
    else
    begin
         rdy_i = 0;
```

```
    end

    if(vectornum === 6'b000000) vectornum = vectornum +1;

end
```

9.3 Test Vectors

```
xxx_xx_x_1____xxxxxxxx_xx
000_01_1_0____00011110_xx
000_xx_0_0____11001001_11
000_xx_0_0____10110100_10
001_11_1_0____11000110_xx
001_00_1_0____10000111_xx
010_00_1_0____00011011_xx
010_10_1_0____00110110_xx
010_xx_0_0____11100001_11
010_xx_0_0____10011100_01
010_01_1_0____10011100_xx
010_xx_0_0____01001011_00
010_xx_0_0____00110110_10
010_xx_0_0____11100001_11
010_10_1_0____10110001_xx
Xxx_xx_x_x____xxxxxxxx_xx
```

9.4 Datapath Schematics and Layouts (Custom)

9.4.1 equal_0 module

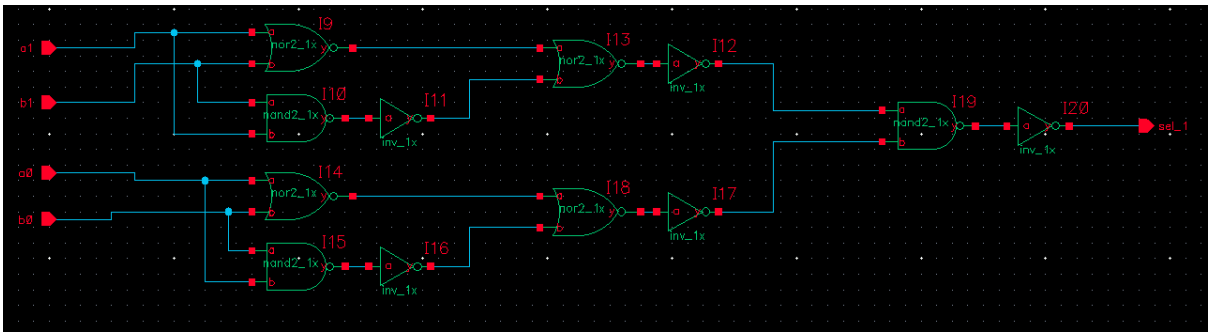


Figure 33: Schematic for Equal 0 module.

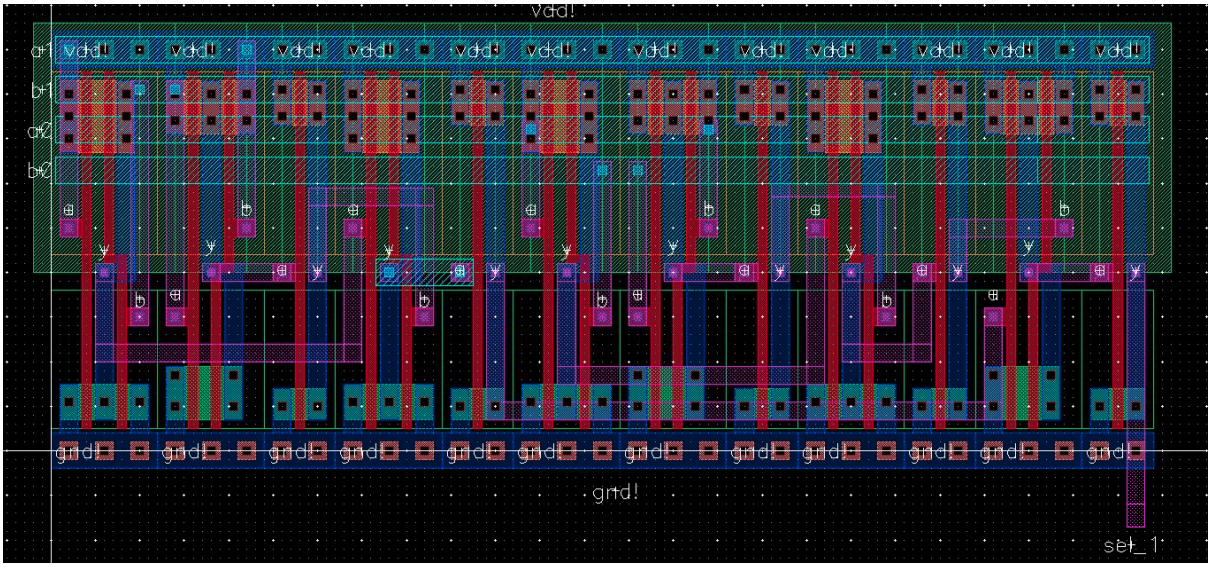


Figure 34: Layout for Equal 0 module.

9.4.2 greater_than_wayhit_counter module

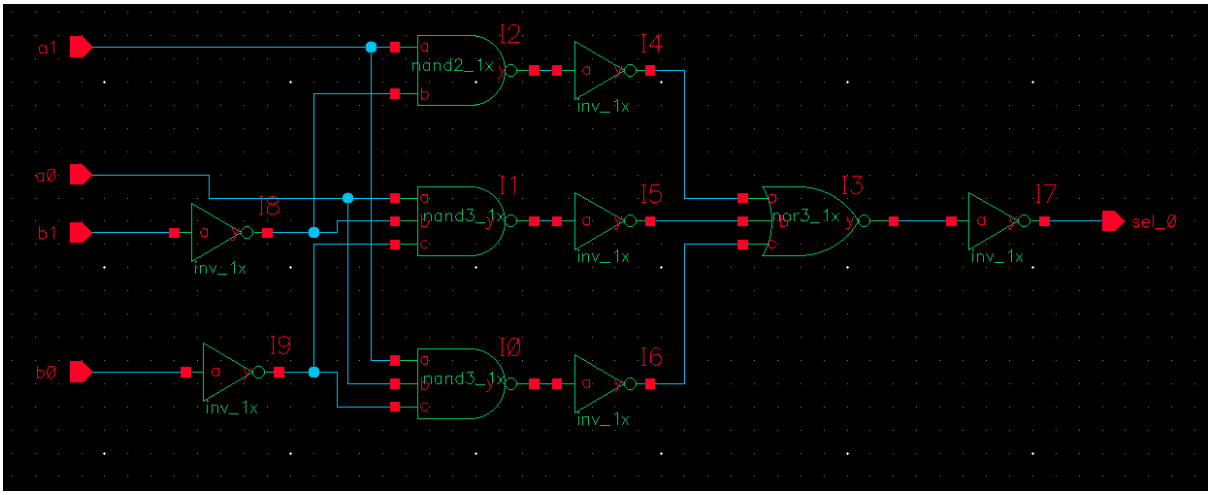


Figure 35: Schematic for greater_than_wayhit_counter module.

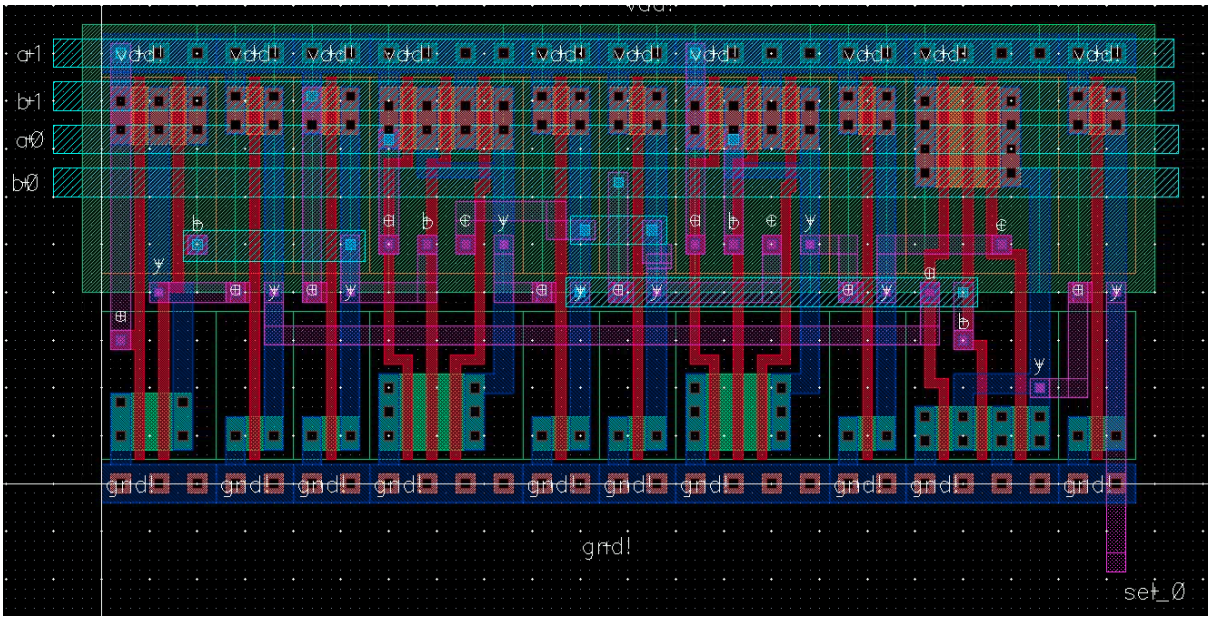


Figure 36: Layout for greater_than_wayhit_counter module.

9.4.3 compare_wayhit_module module

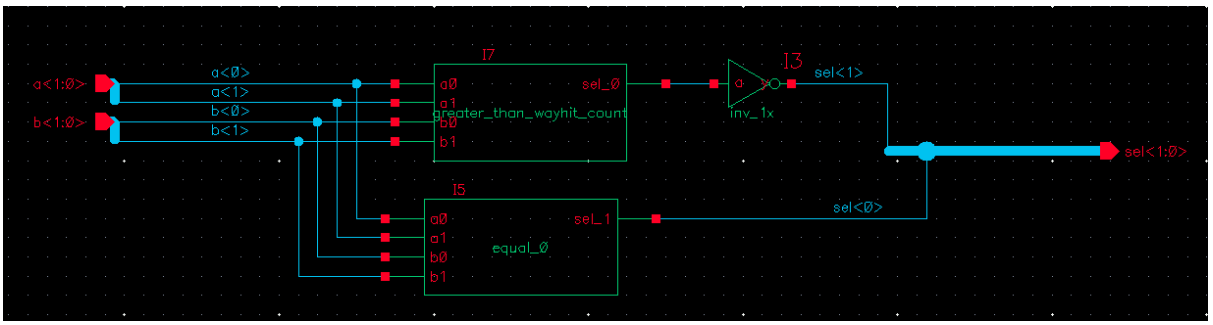


Figure 37: Schematic for the compare_wayhit module that encopasses both modules above.

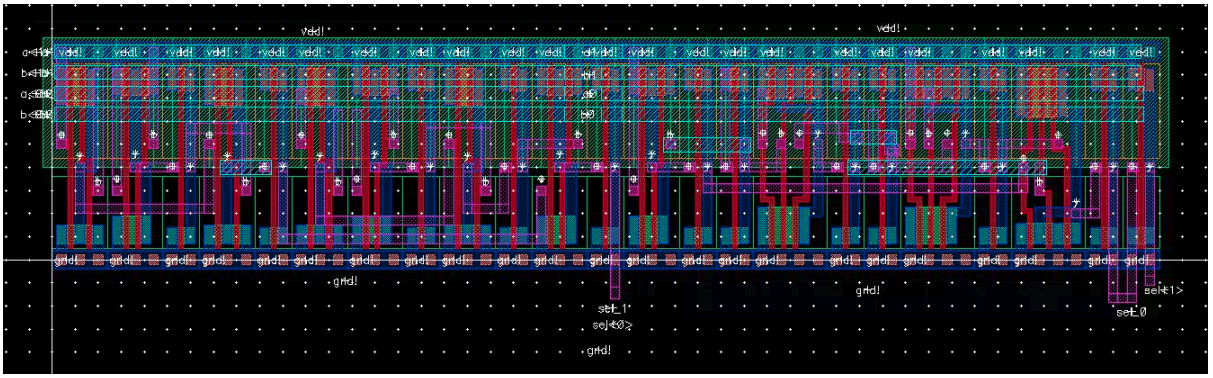


Figure 38: Layout for the compare_wayhit module.

9.4.4 lru_arithmetic_unit module

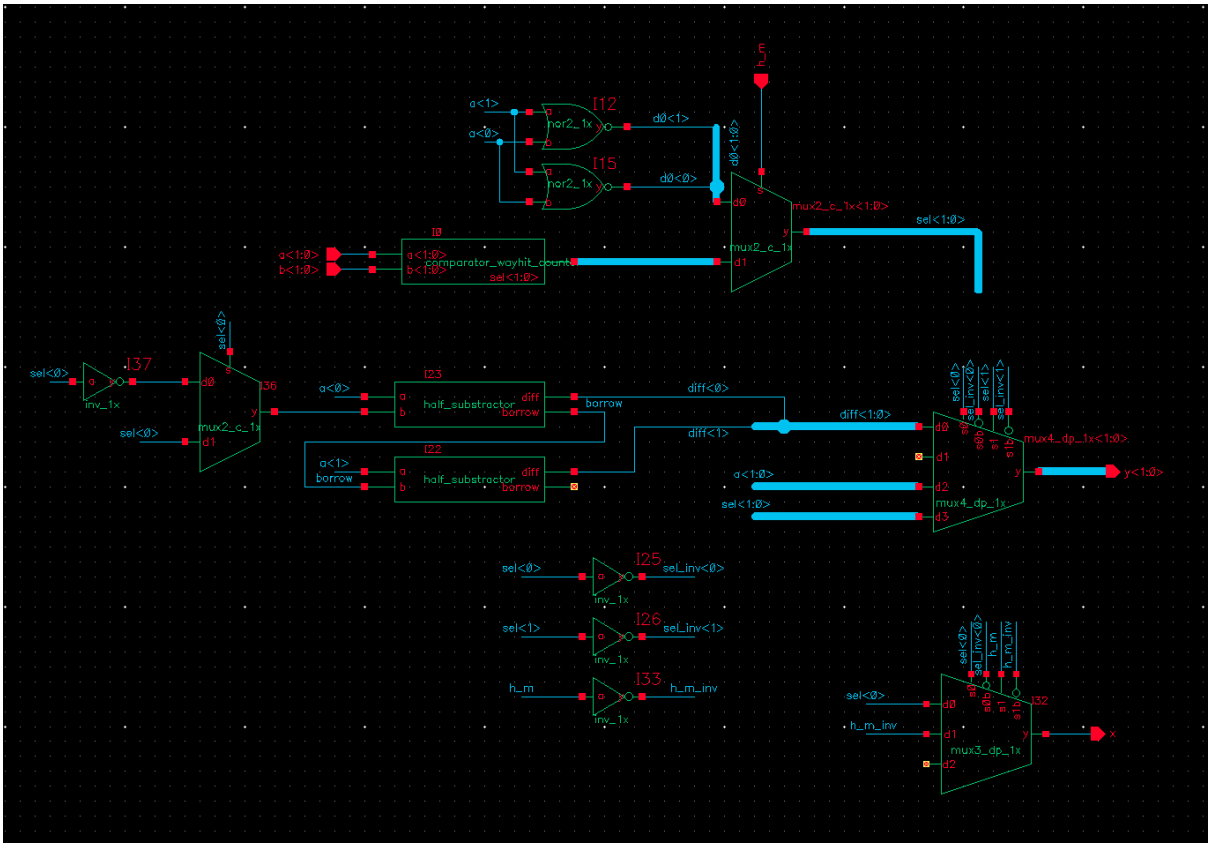


Figure 39: Schematic for the lru_arithmetic module.

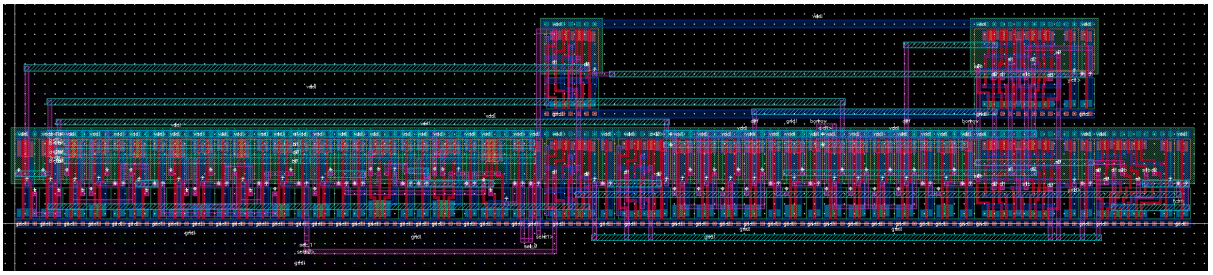


Figure 40: Layout for the LRU_arithmetic module.

9.4.5 4x2_encoder module

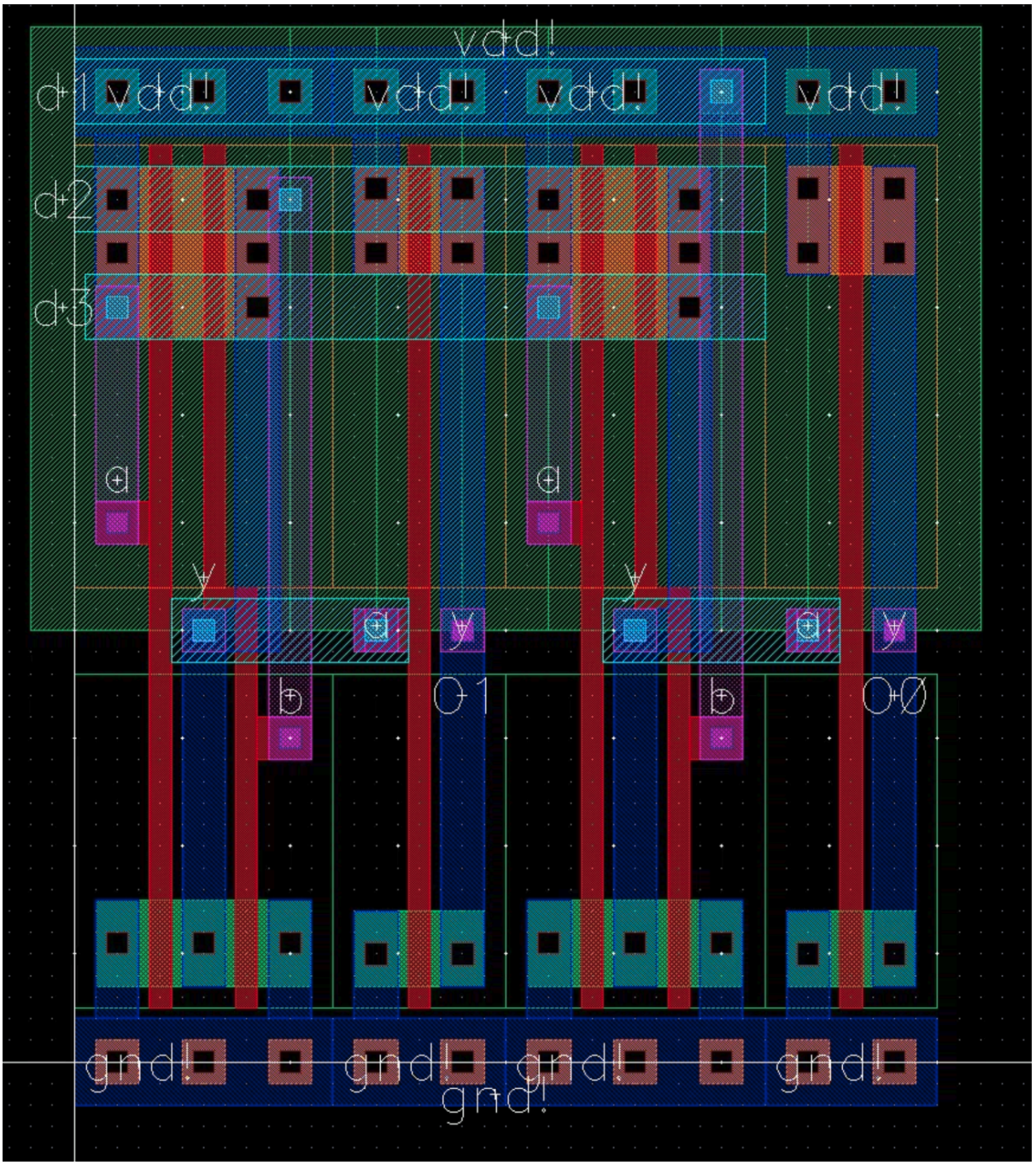


Figure 41: Schematic for the 4x2 Encoder.

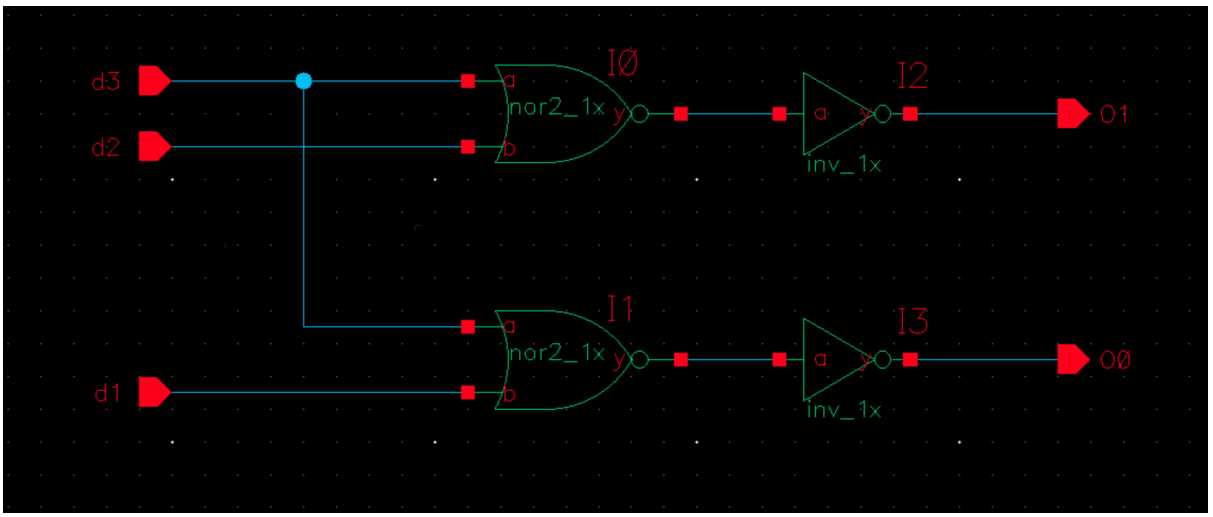


Figure 42: Layout for the 4x2 Encoder.

9.4.6 lru_arithmetic_vector module

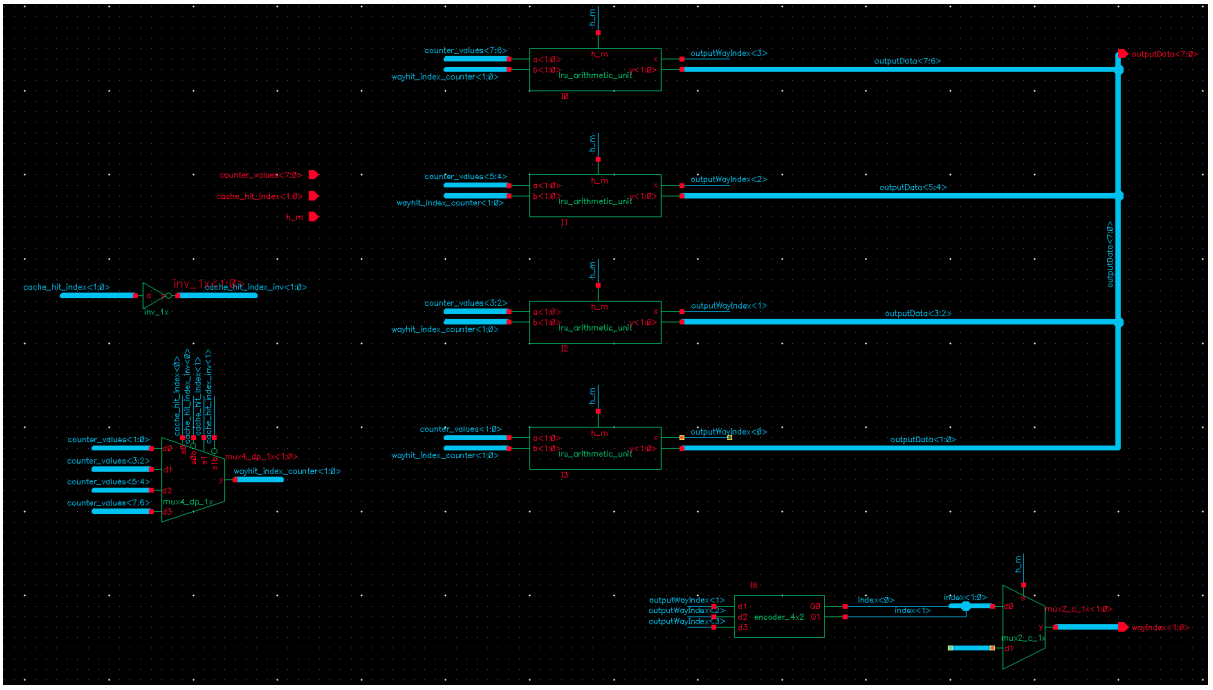


Figure 43: Schematic of the 4 LRU_arithemetic modules running in parallel, also included the encoder that produces the way that was updated to 3.

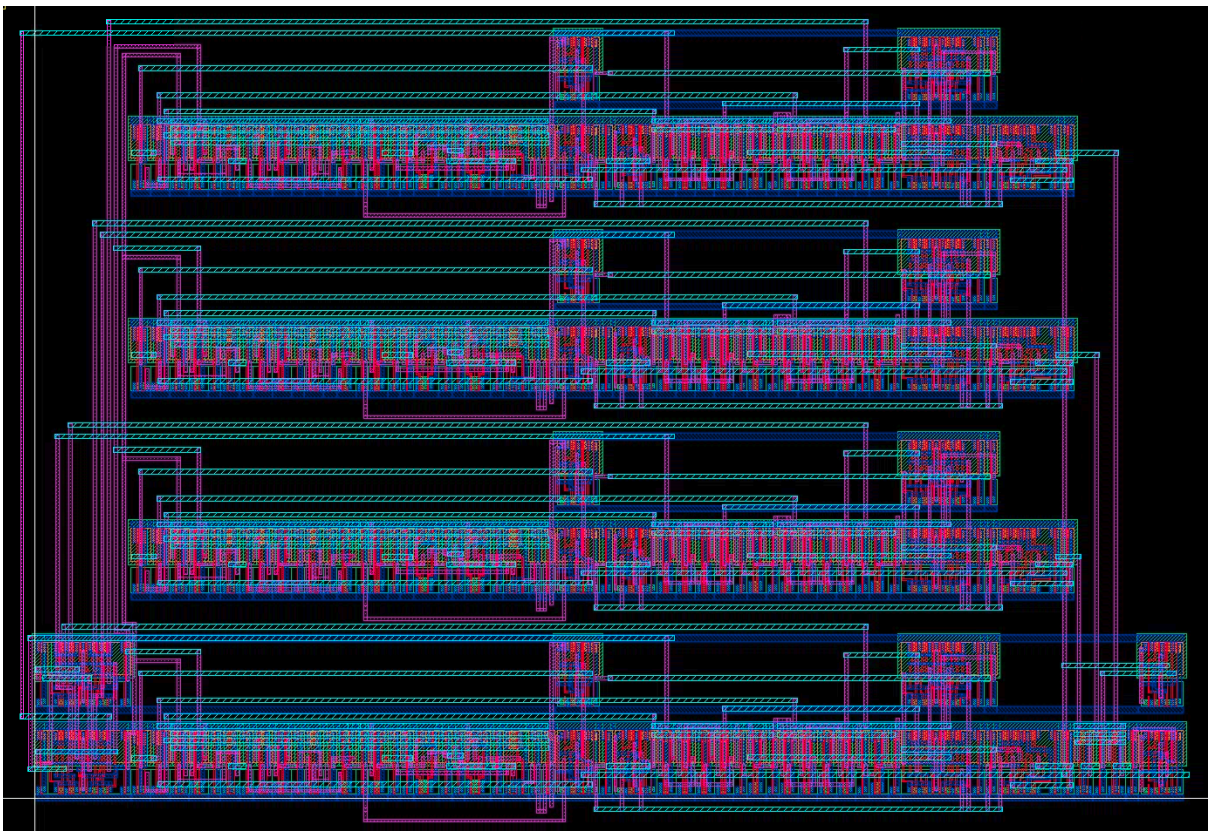


Figure 44: Layout of the 4 LRU_arithmetic module and the encoder mux combination that produces the way that was updated.

9.4.7 datapath module

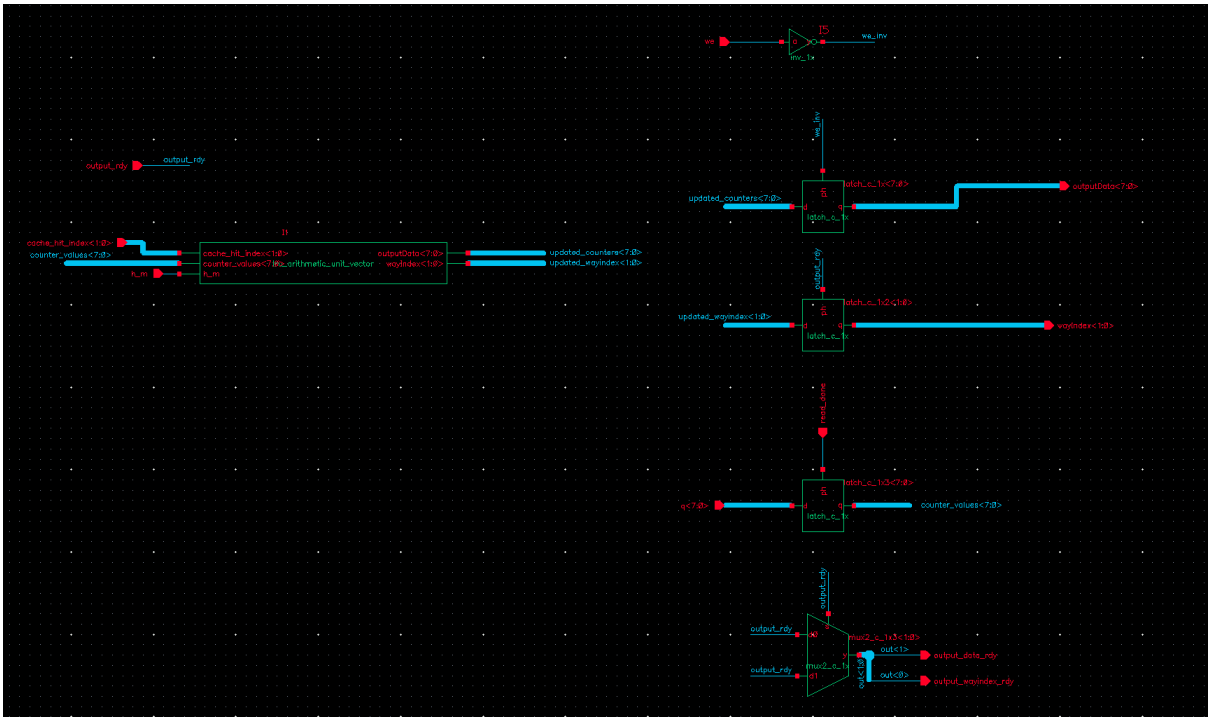


Figure 45: Schematic of the datapath that consists of the 3 latches, the LRU_arithmetic_vector and a mux2x1 module.

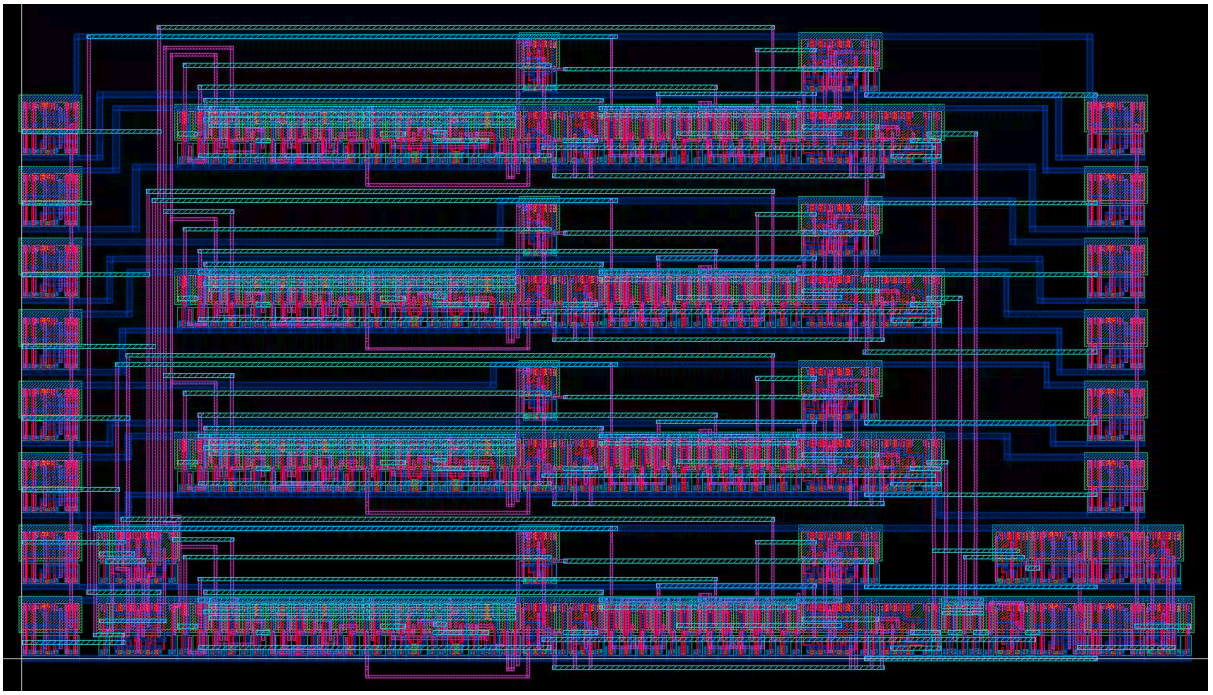


Figure 46: Layout of the datapath consisting of the 3 latches, the LRU_arithmetic_vector and the mux3x1 modules.

9.5 Core Schematics

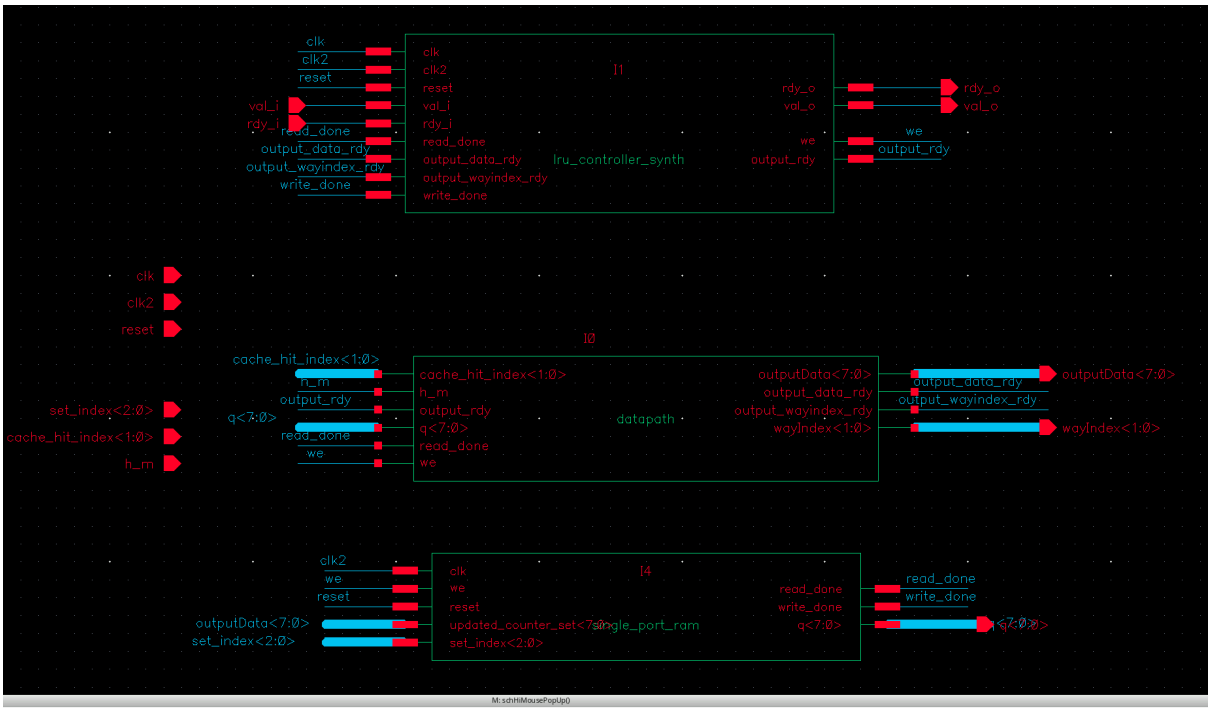


Figure 47: Schematic for the core unit which includes the controller, datapath and the single port ram modules.

9.6 Padframe Schematic and Layout

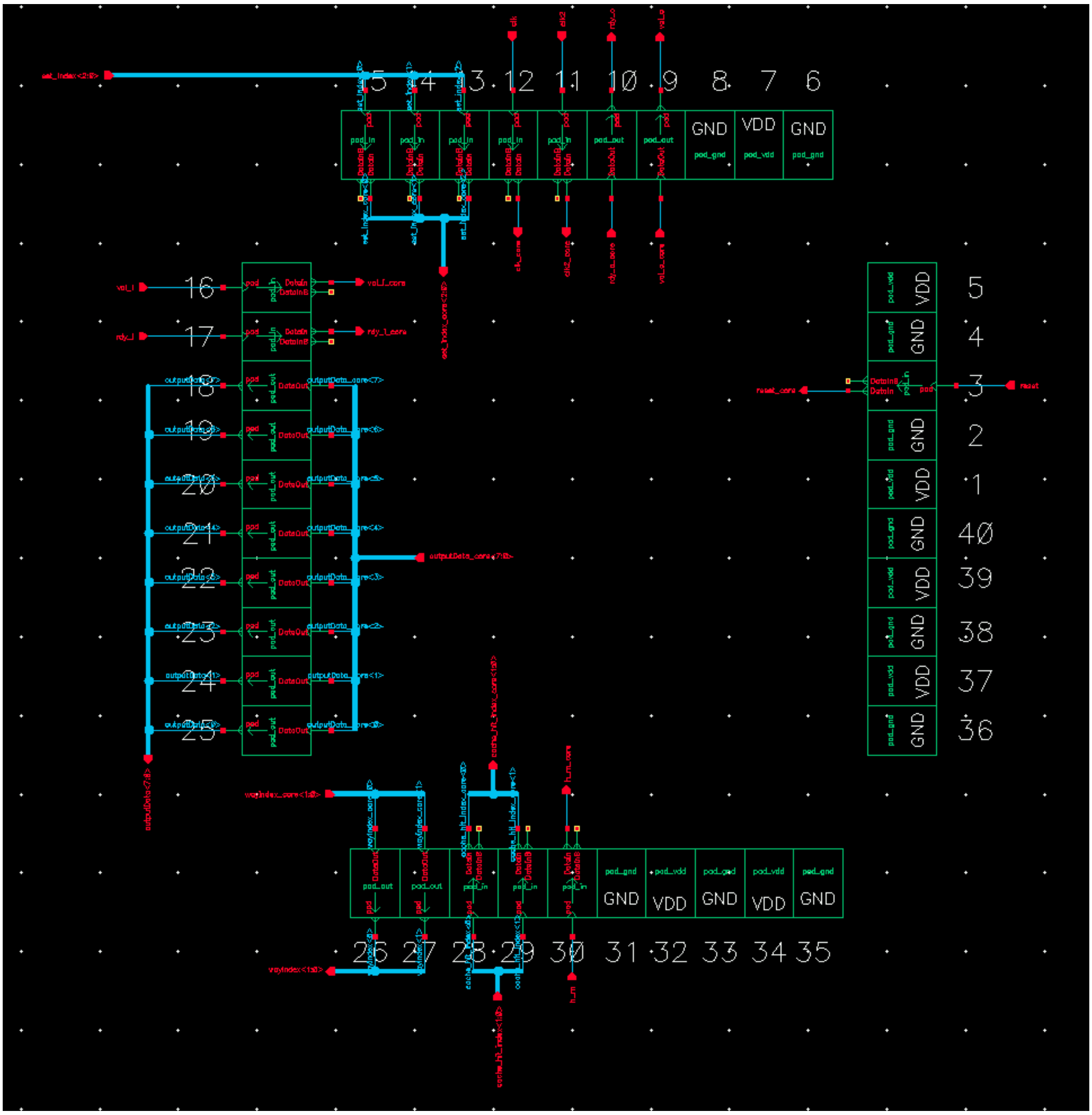


Figure 48: Schematic of the padframe.

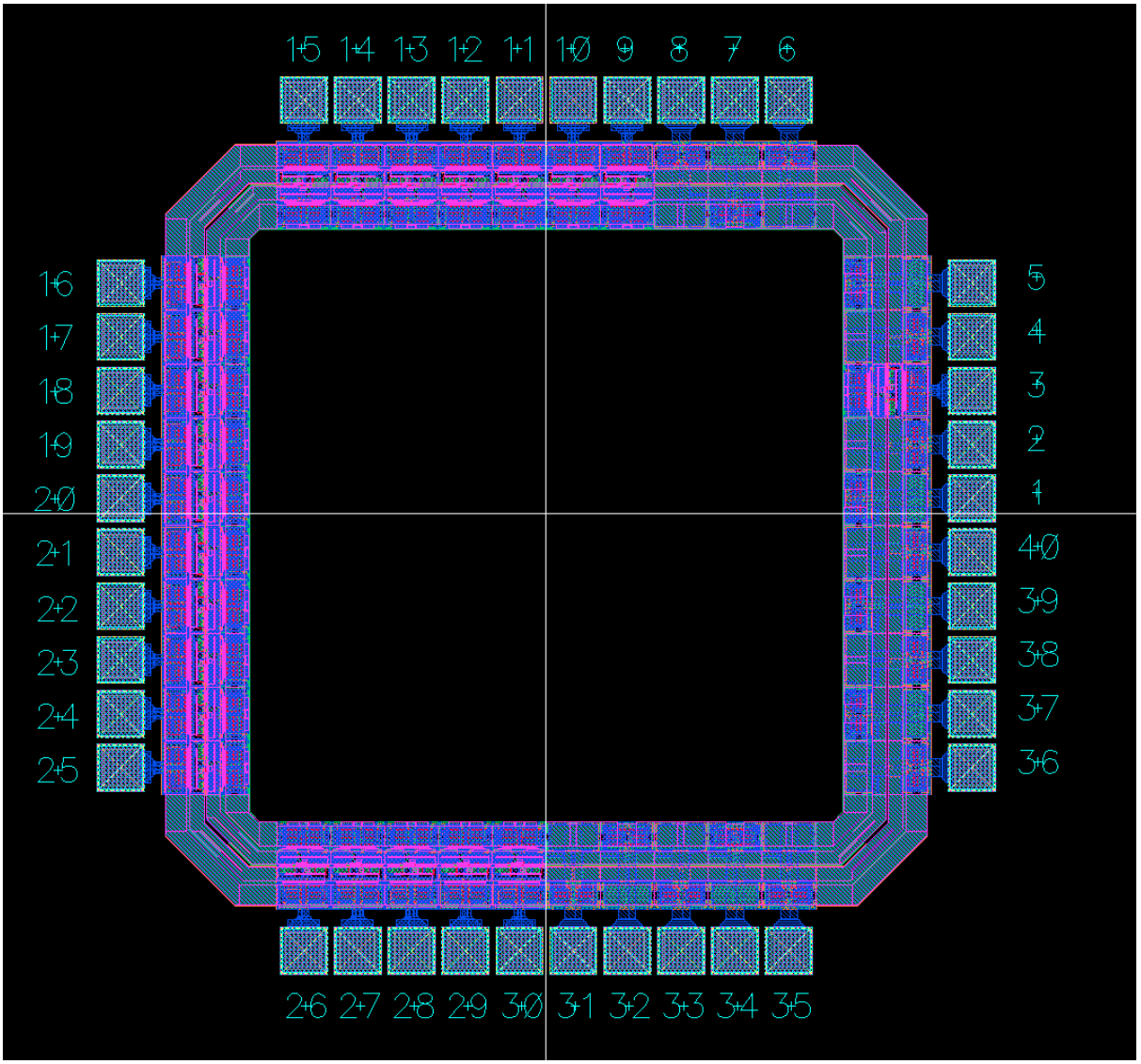


Figure 49: Layout of the padframe.

9.7 Chip Schematic and Layout

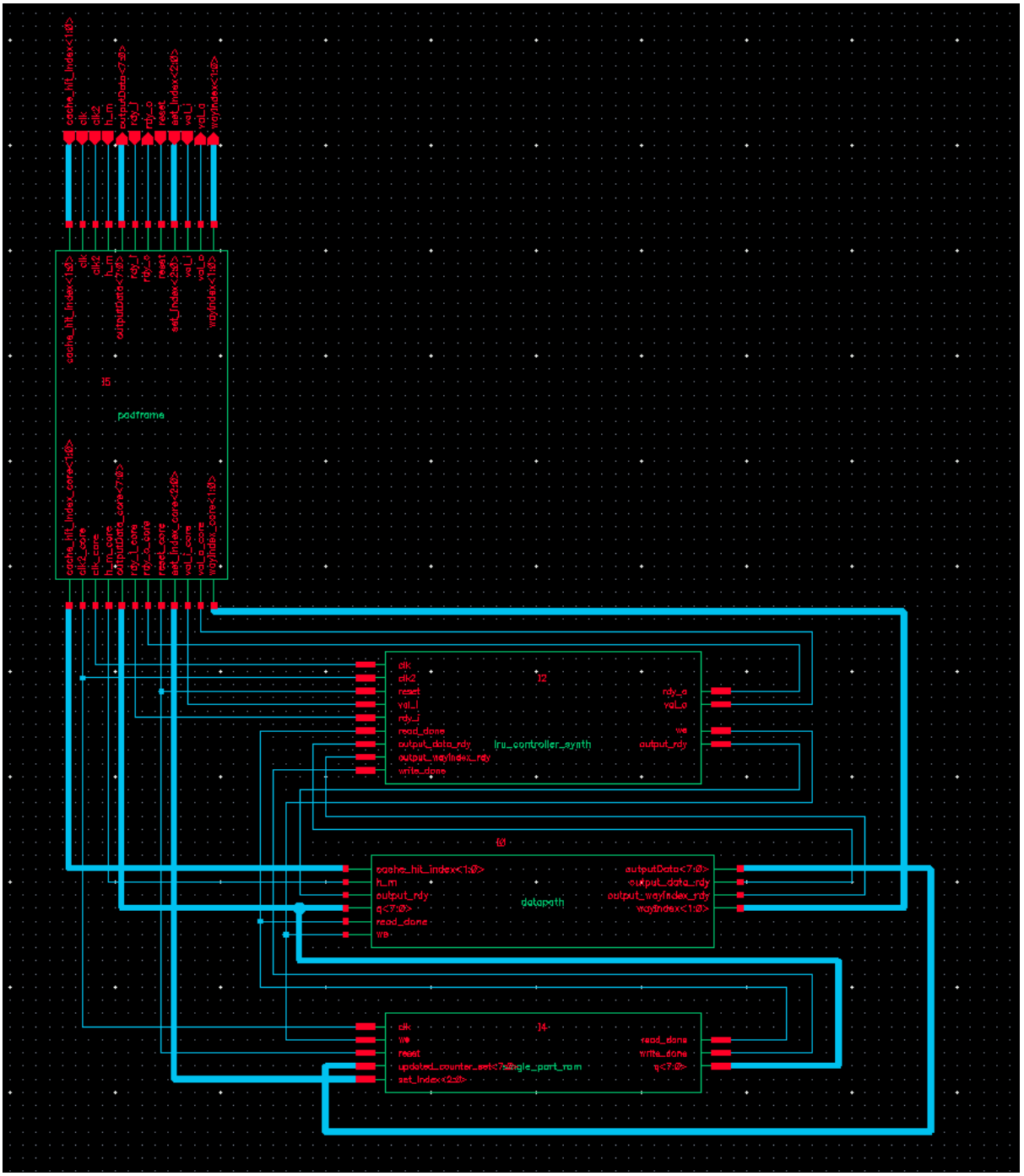


Figure 50:Schematic for the Chip module.

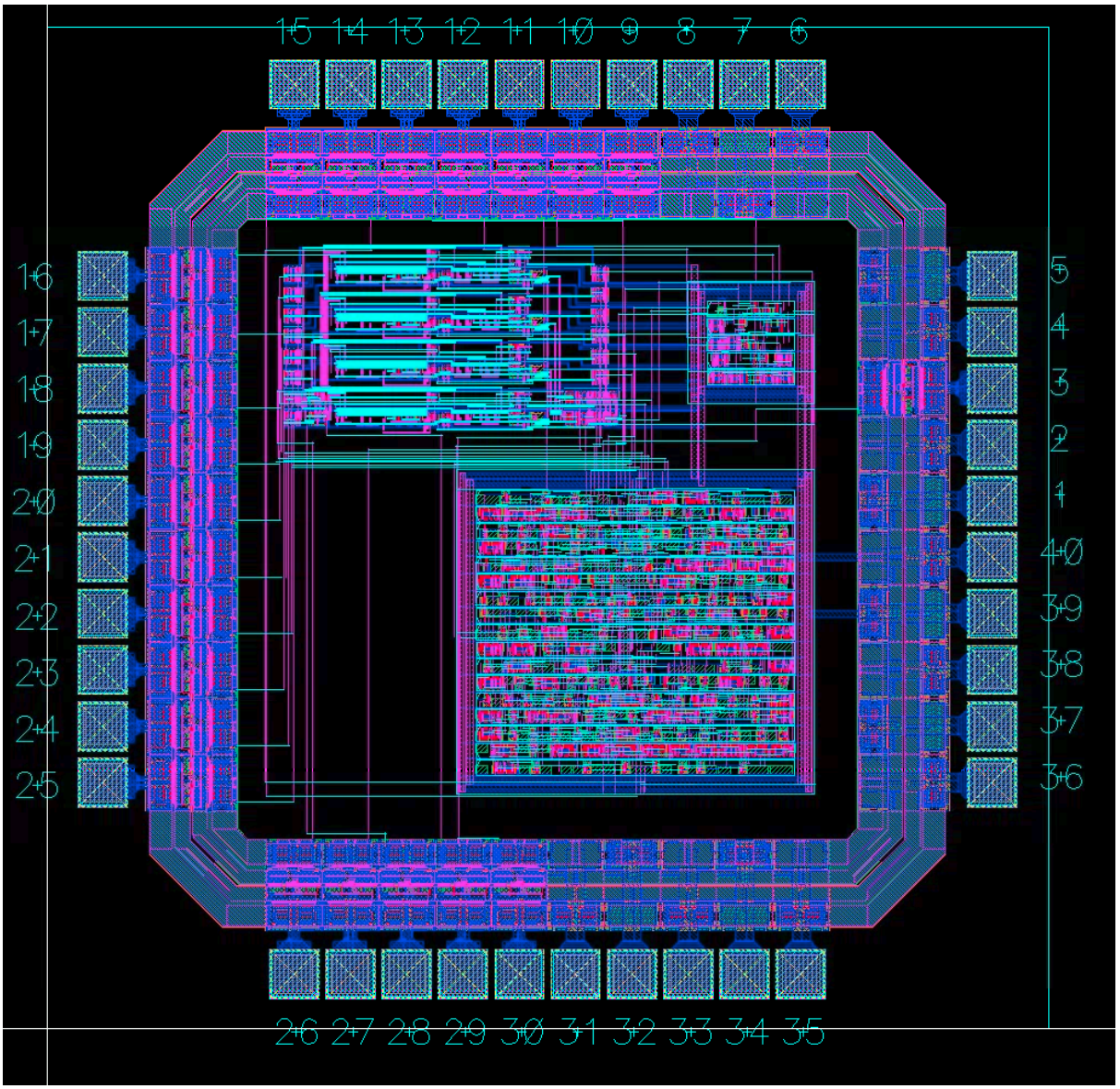


Figure 51: Layout for the Chip module (routed).