CMPE 315: Principles of VLSI Design
Project Cover Page

Final Project

Name: Brian Weber
Section: CMPE640 01

Date Submitted: 11/22/2017

TA / Grader Use Only:
Late Submission Deduction (20% per day):

Other Deductions:

Final Lab Grade:

Comments to student:

Contents

1 Current Status of Code (Read Me)

Before getting into the bulk of the report I would like to write a short summary of the status of my code. Based on my tests, everything works completely, and every layout LVS matches. Every entity that a layout was made for was optimized slightly for CMOS. State machine modules are still suboptimal, but work.

2 Entity Hierarchy

All entities are paired with architecture ”structural”.

- i) chip
  - a) Counter
    - 1) rd_wr_hit_miss_reg
      - (i) dff_reset
      - (ii) Dlatch_Reset
    - 2) SR18
      - (i) dff_reset_high
    - 3) dff_reset_high
    - 4) dff_reset
    - 5) srff
  - b) Cache_Block
    - 1) Cache_Cell_Row
      - (i) Cache_Cell_Valid
        - (a) SRlatch
        - (b) tx
      - (ii) Cache_Cell_Tag
        - (a) Cache_Cell
      - (iii) Cache_Cell_Data_Block
        - (a) Cache_Cell
  - c) Decoder
  - d) Hit_Miss
    - 1) Compare
  - e) Output_Enable
    - 1) tx
  - f) register8
    - 1) dff_reset

3 Chip

For the most part, this design sticks to the overall design shown in the prompt. However, this design only uses one decoder to select which cache cell will be operated on, rather than both a decoder and a multiplexor. The chip functions as follows. A read miss will take 19 clocks to complete. A read hit will take 19 clock cycles from the beginning of the busy signal, to when the cache stops outputting data. A read hit will take 2, and write hits and misses take 3. On a read miss, on the second clock cycle, the cache will enable the memory, and pass on the adress from the cpu. 7 clock cycles after enable goes low, the data will arrive on the memory data input. The cache gets 2 cycles to save the data, before the next value from the word of memory arrives. After all four values are written, the cache waits one more clock cycle before outputting the data that was asked for by the cpu. This process can be seen in Figure **??**. On a read hit, busy stays high for one clock before going low, and outputting the data. This process can be seen in **??**. On write miss, busy goes high for 2 cycles before going low. No data is read or edited. On a write hit, busy stays high for 2 cycles. The new value is written to the cache, and the new value should be available one cycle after busy goes low. Figures **??** and **??** show a write miss and a write hit respectively. An overall view of the chip is shown in Figure **??**.

I placed a lot of my logic that decides what and when to write within the Cache_Cells and rows. This increases size, but simplifies design as the output of the logic will have a much lower fanout, and therefore lower load. This allowed me to not need to worry about parasitic capacitances as much.

```vhdl
library STD;
library IEEE;
use IEEE.std_logic_1164.all;

entity chip is
    port(
        cpu_add     :  in      std_logic_vector(7 downto 0);
        cpu_data    :  inout   std_logic_vector(7 downto 0);
        cpu_rd_wrn  :  in      std_logic;
        start       :  in      std_logic;
        clk         :  in      std_logic;
        reset       :  in      std_logic;
        mem_data    :  in      std_logic_vector(7 downto 0);
        Vdd         :  in      std_logic;
        Gnd         :  in      std_logic;
        busy        :  out     std_logic;
        mem_en      :  out     std_logic;
        mem_add     :  out     std_logic_vector(7 downto 0)
    );
end chip;

architecture structural of chip is

    component Counter
    port(
        clk         :  in  std_logic;
        hit_miss    :  in  std_logic;  --1 hit 0 miss
        rd_wr       :  in  std_logic;  --1 read 0 write
        start       :  in  std_logic;
        Vdd         :  in  std_logic;
        Gnd         :  in  std_logic;
        reset       :  in  std_logic;
        busy        :  out std_logic;
        rd_wr_o     :  out std_logic;
        cache_write :  out std_logic;
        rm_wr_en    :  out std_logic;
        wr_hit      :  out std_logic;
        cpu_dout_en :  out std_logic;
        mem_enable  :  out std_logic;
        write_0     :  out std_logic;  --write word0 to cache
        write_1     :  out std_logic;  --write word1 to cache
        write_2     :  out std_logic;  --write word2 to cache
        write_3     :  out std_logic  --write word3 to cache
    );
    end component;

    component Cache_Block
    port(
        Data_In     :  in  std_logic_vector(7 downto 0);
        Tag_In      :  in  std_logic_vector(2 downto 0);
        Set_Valid   :  in  std_logic;
        Rd_Wr       :  in  std_logic;
        Cache_Write :  in  std_logic;
        Col_En      :  in  std_logic_vector(3 downto 0);
        Row_En      :  in  std_logic_vector(7 downto 0);
        Tag_Wr_En   :  in  std_logic;
        Gnd         :  in  std_logic;
        reset       :  in  std_logic;
        Data_Out    :  out std_logic_vector(7 downto 0);
        Tag_Out     :  out std_logic_vector(2 downto 0);
        Valid_Out   :  out std_logic
    );
    end component;

    component Decoder
    port(
        Mem_Add :   in  std_logic_vector(4 downto 0);
        Col_En  :   out std_logic_vector(3 downto 0);
        Row_En  :   out std_logic_vector(7 downto 0)
    );
    end component;

    component Hit_Miss
    port(
        tag1    :   in  std_logic_vector(2 downto 0);
        tag2    :   in  std_logic_vector(2 downto 0);
        Valid   :   in  std_logic;
        HitMiss :   out std_logic
    );
    end component;

    component Output_Enable
    port(
        in8     :   in  std_logic_vector(7 downto 0);
        enable  :   in  std_logic;
        out8    :   out std_logic_vector(7 downto 0)
    );
    end component;

    component register8
    port(
        d       :   in  std_logic_vector(7 downto 0);
        clk     :   in  std_logic;
        reset   :   in  std_logic;
        Gnd     :   in  std_logic;
        q       :   out std_logic_vector(7 downto 0)
    );
    end component;

    component or2
    port(
        in1     :   in  std_logic;
        in2     :   in  std_logic;
        out1    :   out std_logic
    );
    end component;

    component nor2
    port(
        in1     :   in  std_logic;
        in2     :   in  std_logic;
        out1    :   out std_logic
    );
    end component;

    component and2
    port(
        in1     :   in  std_logic;
        in2     :   in  std_logic;
        out1    :   out std_logic
    );
    end component;

    component nand2
```

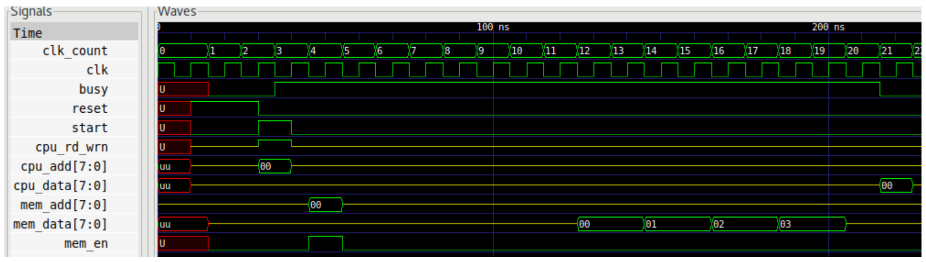

Figure 1: Timing of read miss.

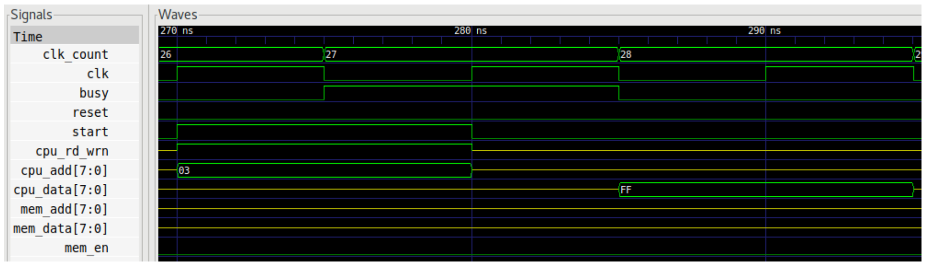

Figure 2: Timing of read hit.

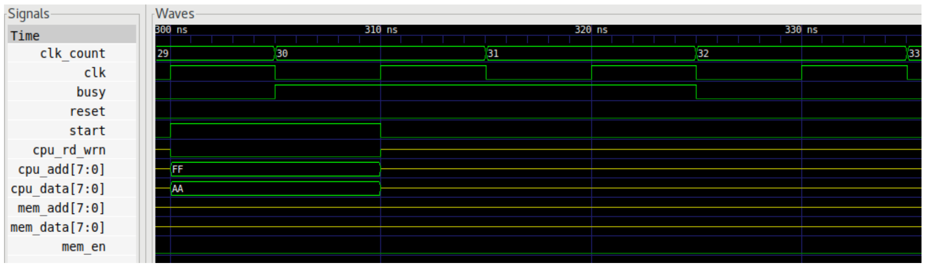

Figure 3: Timing of write miss.

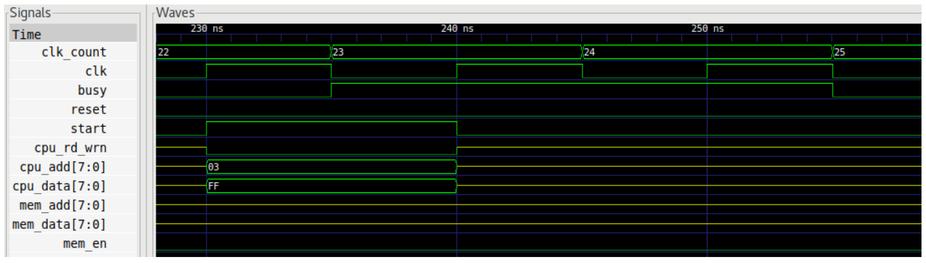

Figure 4: Timing of write hit.

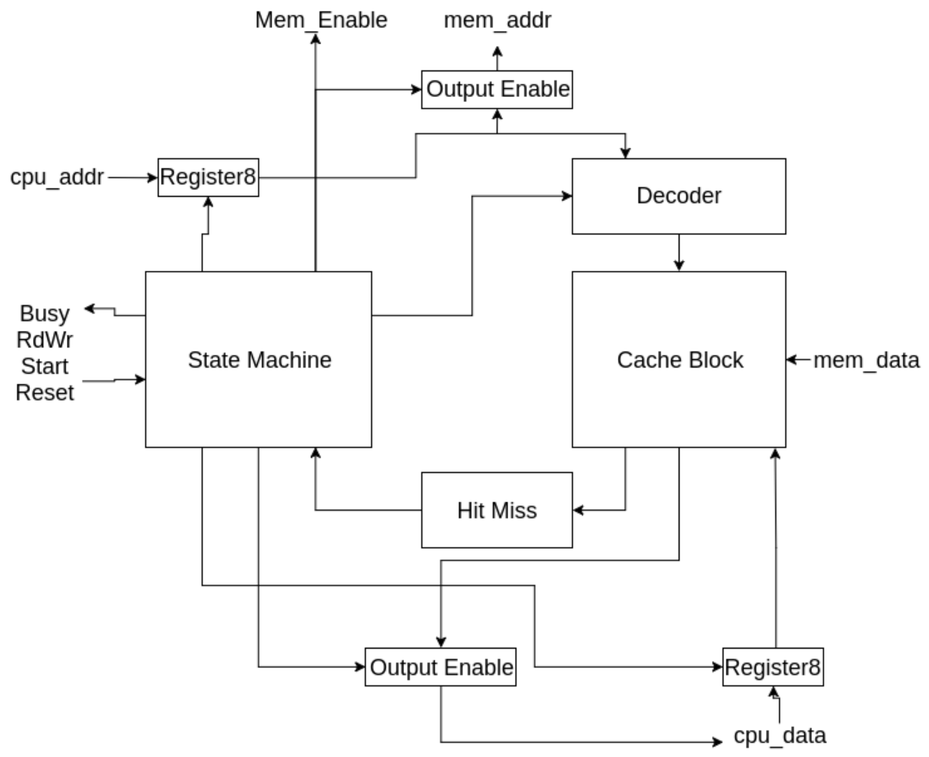

Figure 5: A high level view of the chip module.

4 Counter (State Machine)

The state machine is an entity called Counter, which is centered around a shift register that keeps track of the number of clocks that pass after the busy signal goes high. Depending on hit/miss status of each operation, different things are done depending on the clock count after busy is set high. Another key part of the state machine is a group of registers for storing both the inputs from the cpu, and whether or not there was a hit or a miss. There is an entity called rd_wr_hit_miss_reg that stores the operation (rd or write), as well as whether it was a hit or miss, and outputs 4 separate lines. Figures **??**, **??**, **??**, and **??** show input and output waveforms that were generated from the test bench Counter_Test. Figure **??** shows a state diagram of the state machine. Figure **??** shows the high level view of the hierarchy of the state machine. These waveform figures prove the correct operation of the state machine. The definitions of the signals are as follows:

- s_reset: (Input) The global reset signal.
- s_clk: (Input) The global clock signal.
- s_busy: (Output) The global busy signal.
- s_start: (Input) The start signal from the CPU.
- s_cpu_dout_en: (Output) The signal which enables the transmission gate from cache output to the CPU.
- s_cache_write: (Output) The signal which triggers all writes to the cache. It is connected to the cache block.
- s_hit_miss: (Input) The signal from the Hit_Miss module indicating whether there is a hit or a miss.
- s_mem_enable: (Output) The signal that enables external memory.
- s_rd_wr: (Input) The rd_wr signal given from the CPU.
- s_rd_wr_o: (Output) A rd_wr signal that is sent to other modules after the original rd_wr is latched.
- s_rm_wr_en: (Output) A signal that allows writing to the cache during a read miss.
- s_wr_hit: (Output) A signal that is sent to other modules so they know there was a write hit.
- s_write_[0:3]: (Output) Signals that select which block to be written to during read a read miss. When used with s_rm_wr_en, they override the which column the decoder is currently selecting.

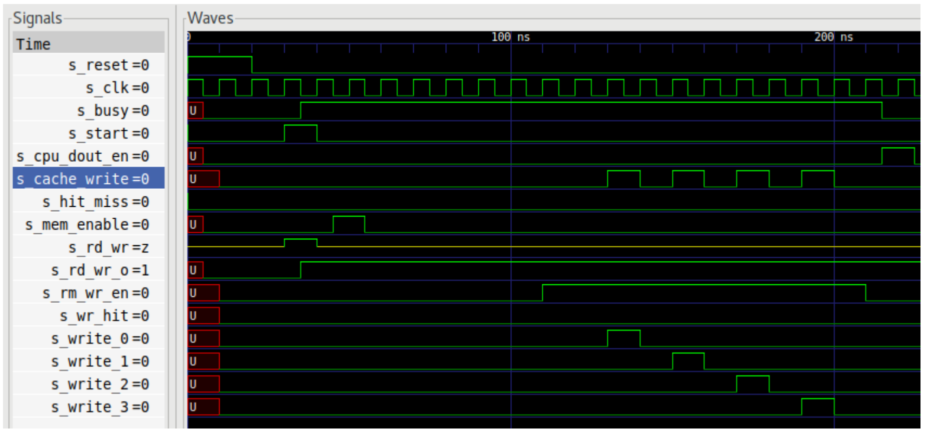

Figure 6: Counter module timing during read miss.

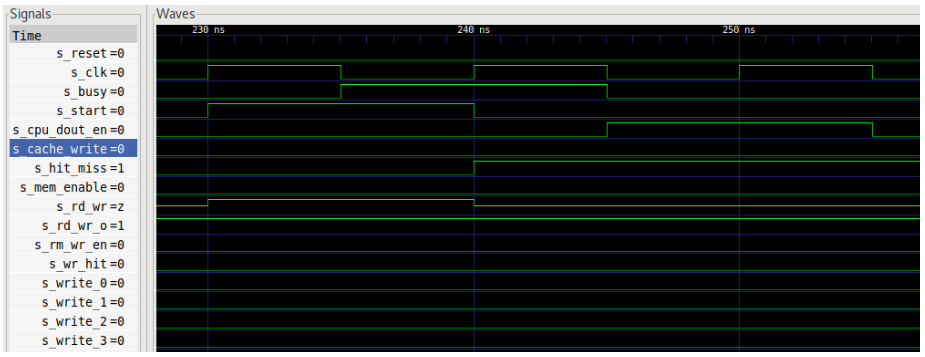

Figure 7: Counter module timing during read hit.

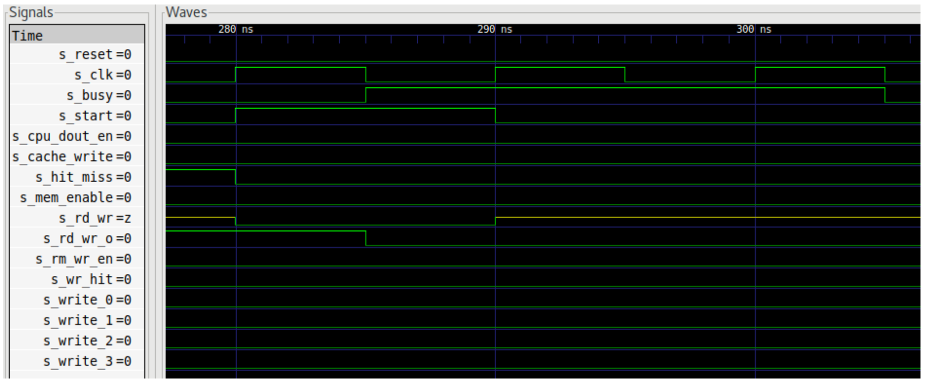

Figure 8: Counter module timing during write miss.

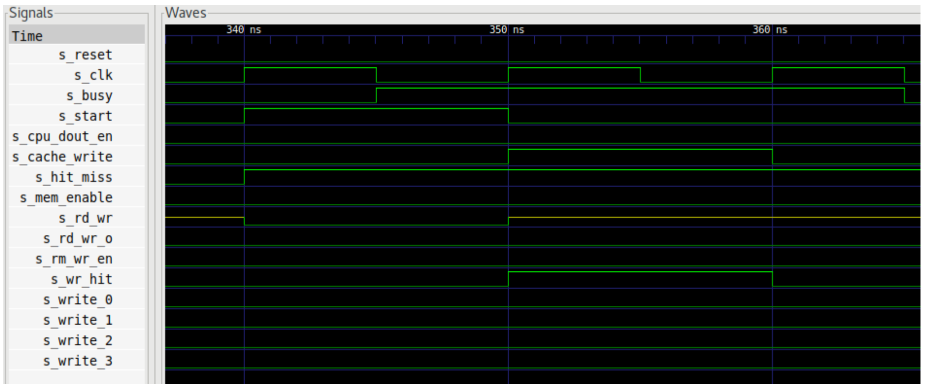

Figure 9: Counter module timing during write hit.

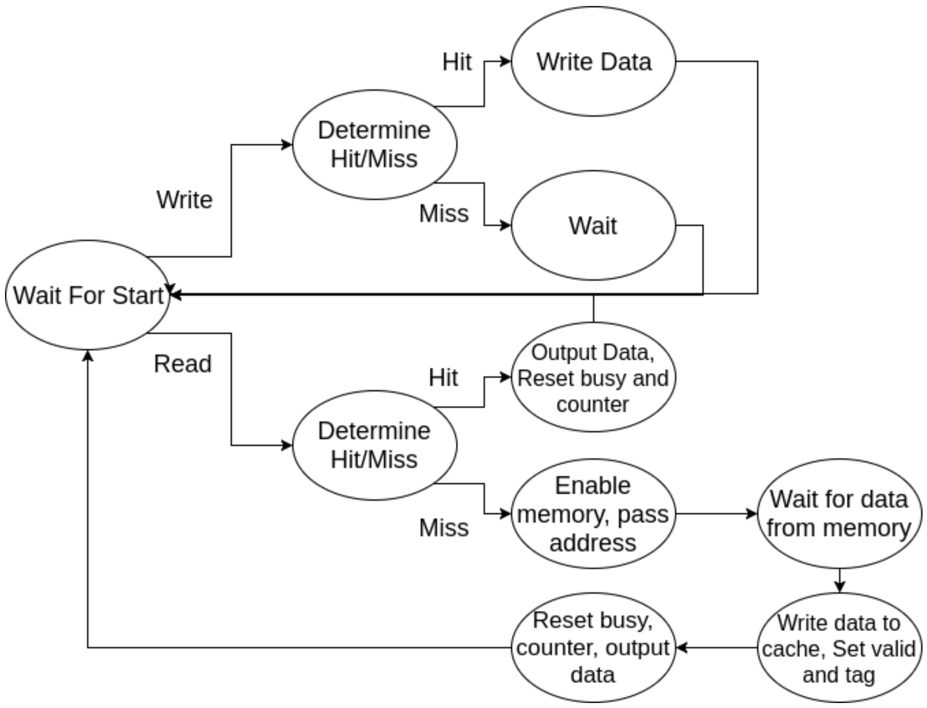
Wait For Start
Write
Read
Determine Hit/Miss
Hit
Miss
Write Data
Wait
Determine Hit/Miss
Hit
Miss
Output Data, Reset busy and counter
Enable memory, pass address
Wait for data from memory
Write data to cache, Set valid and tag
Reset busy, counter, output data

Figure 10: State diagram of the state machine.

4.1 rd_wr_hit_miss_reg

This module is responsible for storing the state of rd_wr, and hit_miss, as well as generating signals for the current combination. During operation, first rd_wr is stored, then once hit_miss is ready, that is latched as well. The rd_wr value is stored in a D flip flop, and the hit_miss value is stored in a D Latch. There

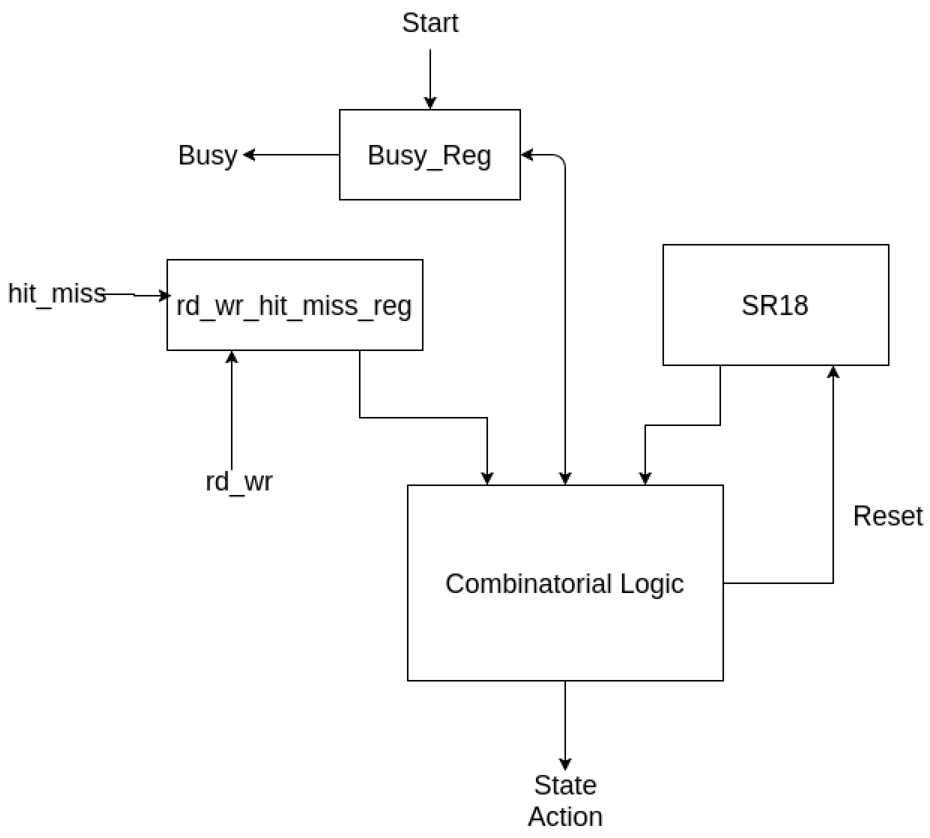

Figure 11: High level view of state machine hierarchy.

are other inputs and logic to enable the clock and latch of each. It then uses some simple logic to generate individual signals for each possible combination of rd_wr and hit_miss. As Figure **??** shows, after both, rd_wr and hit_miss are set high, read_hit is set high.

4.2 SR18

This module is just a simple shift register to keep track of how long the busy signal is high. Using this, the rd_wr_hit_miss_reg, and some combinational logic, it is relatively easy to determine when certain actions have to be performed. Figure **??** shows how this module counts clocks.

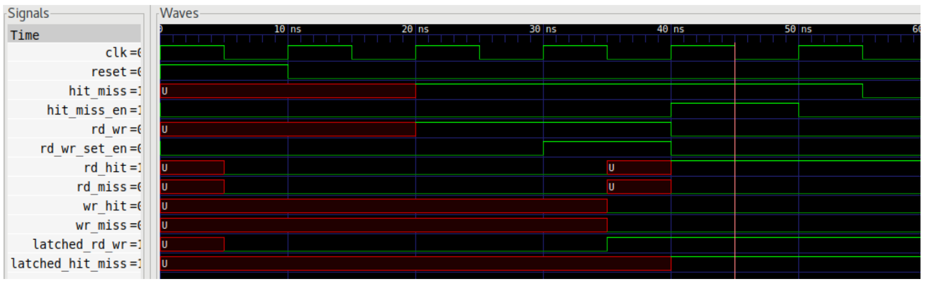

Figure 12: Functionality of the rd_wr_hit_miss_reg module.

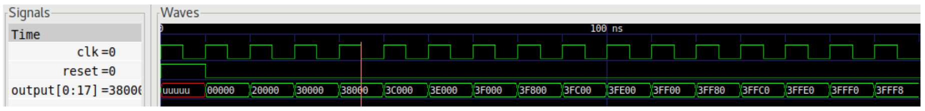

Figure 13: The SR18 module counting clock cycles.

5 Cache_Block

The cache block is a block of positive latch enabled Dlatches, indexed by the cpu address. In front of each Dlatch is a transmission gate. The inputs and outputs of each row all come from and lead to the same 8 bit bus, but only one row at a time can be selected due to the Decoder module. This prevents conflicts on the outputs as well as prevents more than one row from being written to at a time. The valid cells are asynchronous sr latches, and are set while data is being written from memory during a read miss. Tags are also set at this time. The valid and tag bits are output as soon as a row is selected, however, for a byte to be output, the internal rd_wr signal must be high, and a column must be selected. The Cache_Block was split into several modules for convenience. First it is split into rows, called Cache_Cell_Row. The rows are split into 4 8 bit data blocks, called Cache_Cell_Data_Block, a tag block called Cache_Cell_Tag, and a valid bit, called Cache_Cell_Valid. The Valid bit is an sr latch, while the the tag and data blocks are made up of 3 and 8 individual bit modules called Cache_Cell. This Cache_Cell is a resettable Dlatch with a transmission gate in front for enabling output. Figure **??** shows a waveform diagram of the cache block. At 10 ns, the data arrives at the input, the rd signal is turned on, and the row and column are selected. At 15 ns, cache_write is turned on, writing the input data. Since rd is on, the data shows on the output as well. At 15 ns, the tag and valid bits are also set. Notice that after the rd signal turns off at 20 ns, the data output becomes high impedence, however the tag and valid bits continue to output. At 30 ns, different data is set at a different position. At 40 ns, the old data is read, and at 45 ns, the new data is read. Here is a description of all the signals:

- data_in: (Input) Input data to the cache.

- data_out: (Output) Output data from the cache.
- rd_wr: (Input) Internal read write signal used for enabling data output.
- cache_write: (Input) Signal used to trigger writing of data to cache.
- row_en: (Input) Signal used for selecting which row to operate on.
- col_en: (Input) Signal used for selecting which column to operate on.
- tag_out: (Output) Output of tag of selected row.
- tag_in: (Input) Input tag to the cache.
- tag_wr_en: (Input) Triggers writing the tag to the row.
- set_valid: (Input) Sets the valid sr latch on the selected row.
- valid_out: (Output) Output of the selected row's valid bit.
- reset: (Input) Global reset.

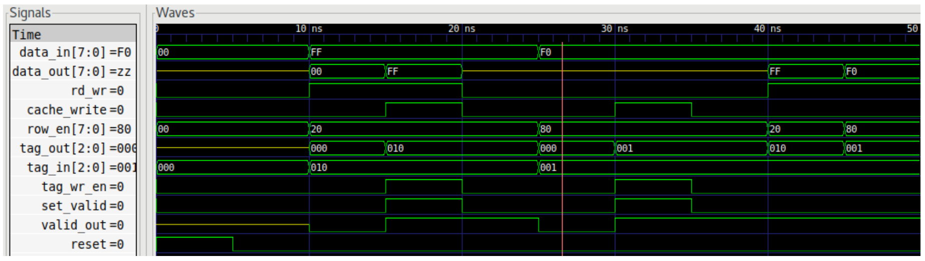

Figure 14: Timing diagram showing the functionality of the cache_block.

6 Decoder

The decoder selects which byte operations are to be performed on based on the memory address. It does this asynchronously. It still has to be optimized for cmos. Only one bit of each output is turned on at a time. The first Three bits of input index the row, and the last 2 index the column. Figure **??** shows example inputs and outputs.

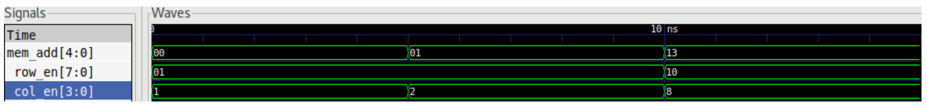

Figure 15: Functionality of the decoder module.

7 Hit Miss

This module is responsible for determining if there is a hit or a miss, based on the tag from the cache, and the tag from the memory address, as well as the valid bit from the selected row of the cache. There is a hit if both the tags match, and the valid bit is high. This module is asynchronous as well, and because the value of its output may change as the cache is written to (even within the same operation), its initial value is latched within the state machine in the beginning of each operation. There is a module called compare inside Hit_Miss which is reponsible for deterining whether the tags are equivalent. Figure **??** shows example inputs and outputs of this module.

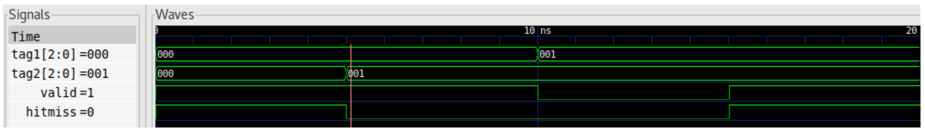

Figure 16: Functionality fo the Hit_Miss module.

8 Register8 and OutputEnable

These two modules are used to control IO to the cpu and memory. The registers ave the data that is sent from the cpu, and the Output_Enable module is used to mask output to the data bus until it is needed. The register8 module uses an array of negative edge trigger D flip flops, and the Output_Enable module uses an array of 8 transmission gates.