Lab 07 - Generic Flip-Flop CAM Design Space Exploration

Pre-lab

You will need to submit your testbench on ilearn prior to coming to the lab, or whatever due date is given on Canvas. Your testbench should demonstrate that you have read through the lab specifications and understand the goal of this lab. You will need to consider the boundary cases. You do not need to begin designing yet, but this testbench will be helpful during the lab while you are designing.

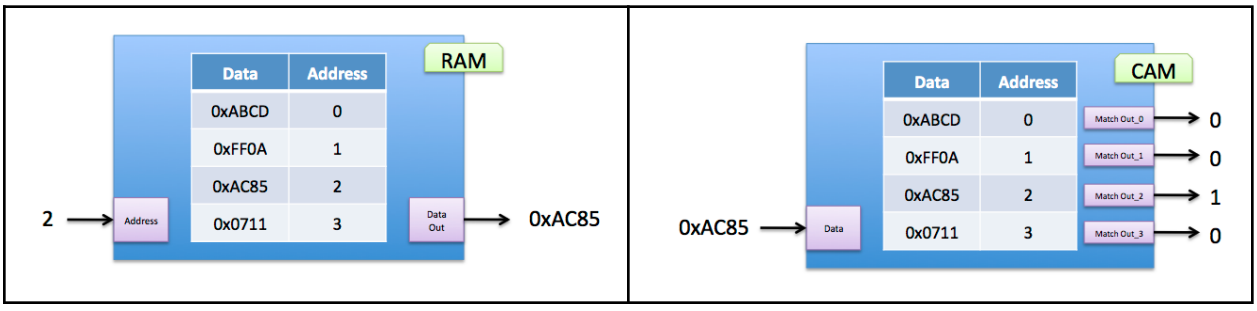
Introduction

In this lab you will be implementing a generic Content Addressable Memory (CAM). This CAM will have the following parameters:

- `M`, being the number of words stored in the CAM
- `N`, the width of each word in the CAM.
  (You can assume all words in the CAM will have the same width)

RAMs vs CAMs

Random Access Memories (RAMs) will read in one address. Find the data stored at that single point in its memory, and return the data it finds. A CAM will do something entirely different. A CAM will take a data point, search its **ENTIRE** memory for that data point, and return all address locations that match the data point. As can be seen in the following figures:



CAM Design

The CAMs you will be building for this lab can be thought of as having three levels.

1. The CAM Array (`CAM_Array.v`) which consists of
2. `M` CAM rows (`CAM_Row.v`, which you are given an implementation of) each of which consists of
3. `N` CAM cells (`BCAM_Cell.v`, `TCAM_Cell.v`, `STCAM_Cell.v`).

The top level module for your design, which the user will interact with, will be the CAM Wrapper (`CAM_Wrapper.v`).

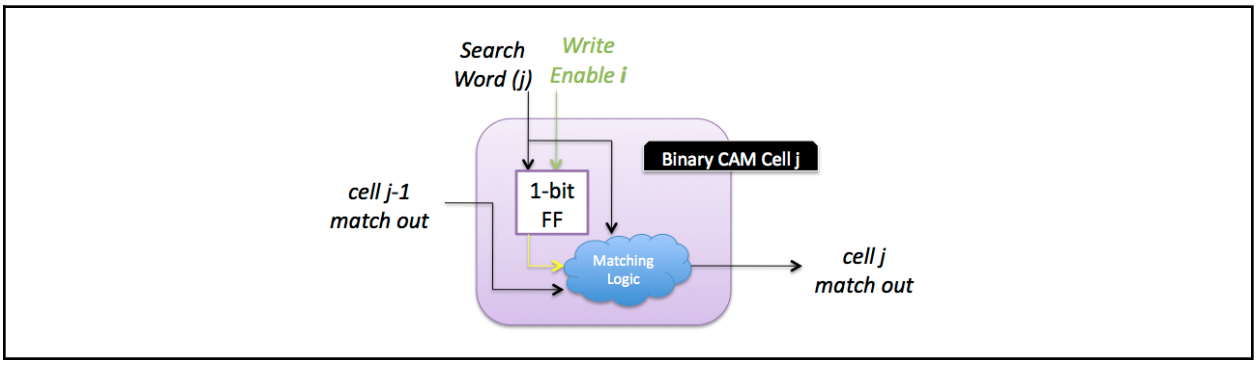
CAM Cell

The CAM cell will hold one bit of data in a Generic Flip Flop. This bit is continually being read (on the rising edge of the clock) unless the bit is being written. The searching (match) is done asynchronously. There are three different CAM cells you should implement for this lab.

Binary CAM Cell

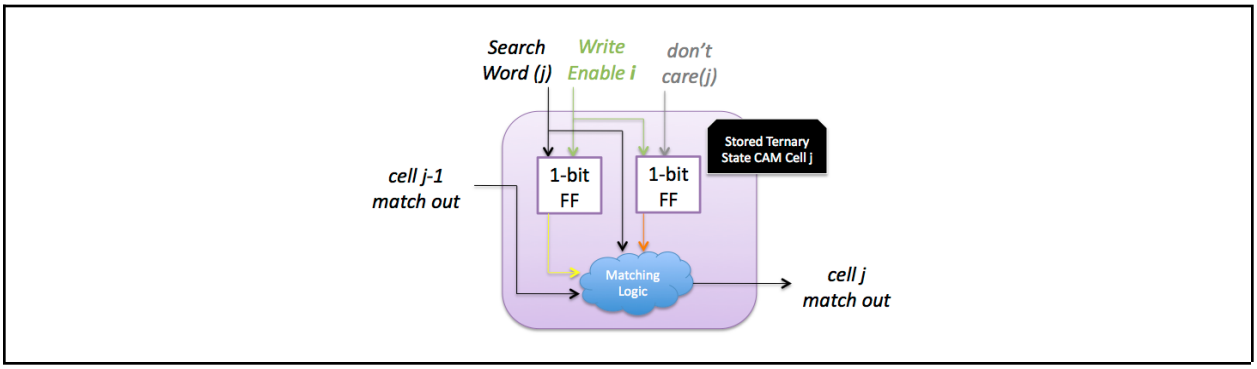
The first CAM cell, **Binary CAM Cell** (`BCAM_Cell.v`), is the simplest. This CAM cell will store only one bit of data.

- This bit of data can be changed by setting the '`search_word`' port to a value and setting '`write_enable`' high.
- The j^{th} binary CAM cell in a CAM row has an input from the $(j-1)^{th}$ cell. This input says whether the last j-1 cells have a match up to this point. The j^{th} cell determines if it has a match, and propagates its result to the $(j+1)^{th}$ cell.



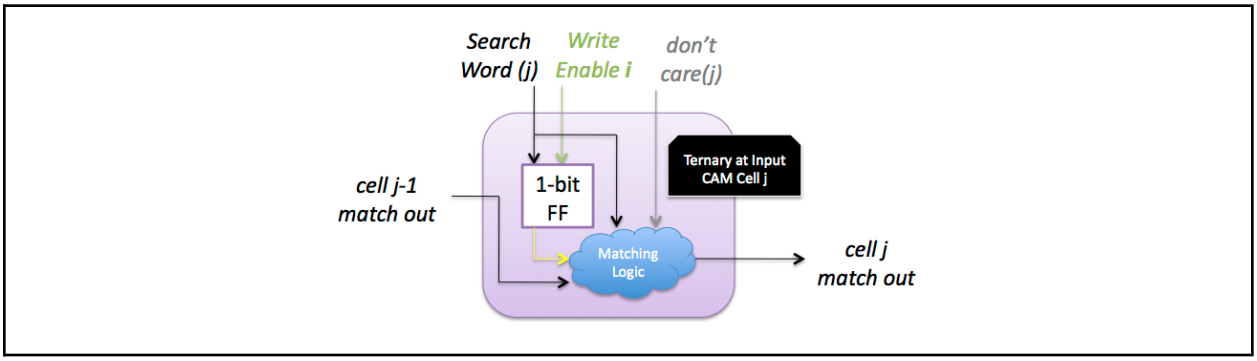
Stored Ternary State CAM Cell

The **Stored Ternary State CAM Cell** works similar to the binary CAM cell. The main difference being a '`don't care`' bit (X). The stored ternary state CAM cell will keep a second flip-flop that stores this '`don't care`' bit. When this bit is high, it doesn't matter what value is stored in the CAM register. The '`don't care`' bit is written in the same fashion as the CAM register.



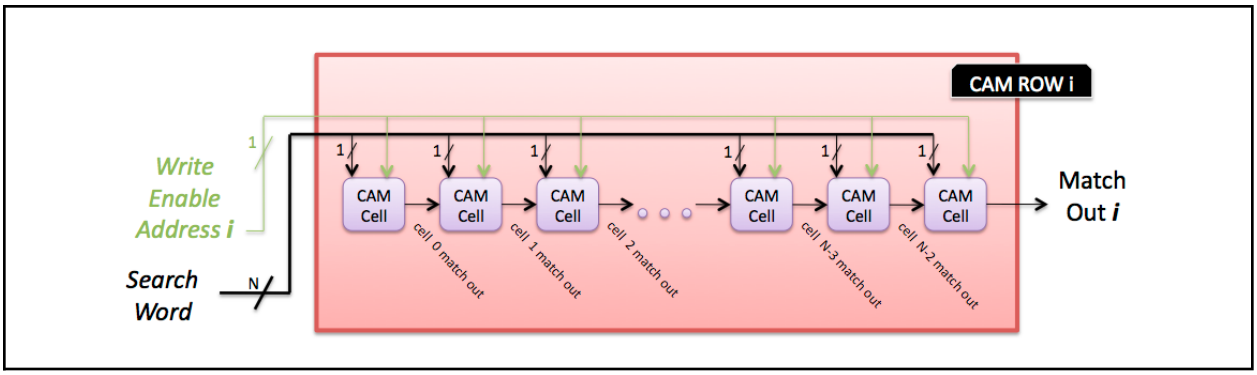
Ternary at Input CAM Cell

The **Ternary at Input CAM Cell** will not store a '`don't care`' bit. Instead, each search into the CAM is accompanied by a '`don't care`' bit. The match logic will consider the don't care bit on a search, by search basis.



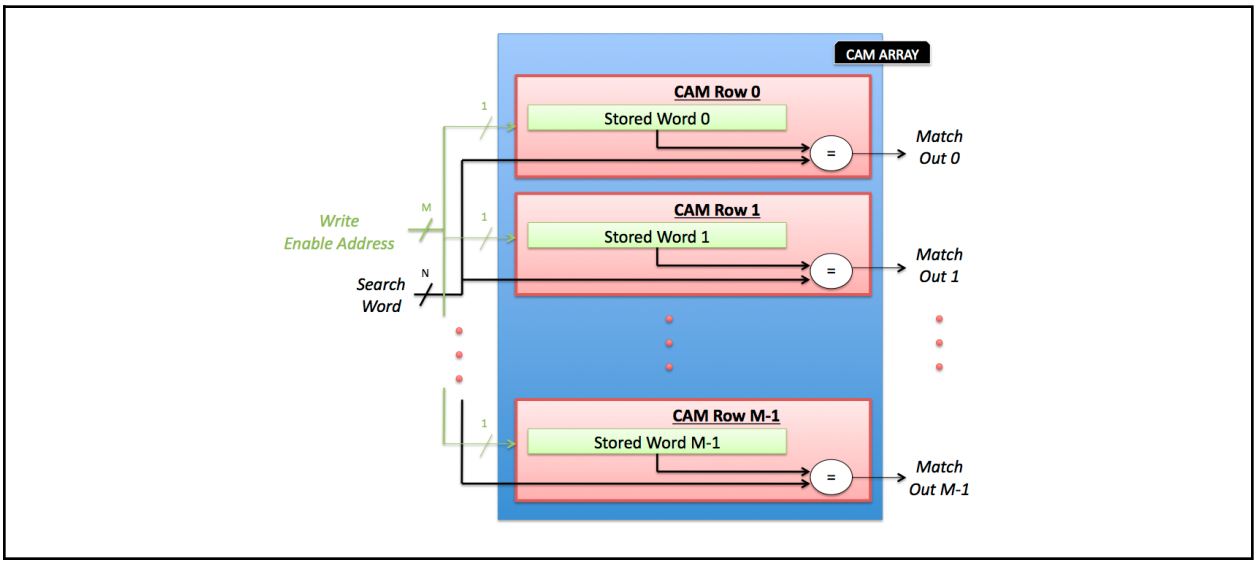
CAM Row

The CAM row consists of N CAM cells. The CAM row takes a search word, N-bits in width, and compares each of these N-bits to their corresponding CAM cells. If all the cells match then this CAM row sets '`match_out`' to high. The CAM row also has a 1-bit signal for '`write_enable`' when '`write_enable`' is set to high the CAM row writes the N-bits of '`search_word`' to the `N` CAM cells. The searching process should be asynchronous.



CAM Array

The CAM array will hold M CAM rows. The CAM array will read in one '`search_word`', N-bits in length, and send this N-bit '`search_word`' to all M CAM rows. The CAM array will also have an M-bit '`match_out`' port (one for each CAM row), to show which rows matched the search. The CAM array will have an M-bit '`write_enable_address`' that will be used to specify what CAM rows will be written to. Again this process should be asynchronous.



Lab Details

For this lab you are given skeleton code. These files are all you need to complete this lab. See the files in the zip file..

`BCAM_Cell.v`
`TCAM_Cell.v`
`STCAM_Cell.v`
	CAM Cells - Implement the 3 CAM cell types as described above. All should have the same input and output ports (Even if you don't use all of them in your design).

`CAM_Row.v`	You should create N CAM cell, based on the CAM_WIDTH parameter. You should also connect all N CAM cells together in this file. You should be able to change the CAM cells being used in your design from this file.
`CAM_Array.v`	This file you do not have to modify. The implementation creates M CAM rows, based off the CAM_DEPTH parameter. It connects all M CAM rows to the necessary CAM Array ports.
`CAM_Wrapper.v`	This file you do not have to modify, but all lower files should work with this file as the top level.

Some hints and specifications for this lab.

- When searching for a word it should take **two clock cycles** to return all of the correct rows that match.
- Not all parts of the CAM cell need to run based on the clock
- All flip flops will run based on the clock (note the matching logic is not in the flip flop)
- When generating N CAM cells and M CAM rows you will want to use a `generate` statement. This structure in Verilog allows you to synthesize a variable number of a module or subcircuit based on a parameter. Tutorial

Submission:

Each student **must** turn in one tar file to Gradescope. Your work can be (and should be) identical to the other members of your group. The contents of which should be:

- A README file with the group members names, and any incomplete or incorrect functionality
- All Verilog file(s) used in this lab (implementation and test benches).

If your file does not synthesize or simulate properly, you will receive a 0 on the lab.