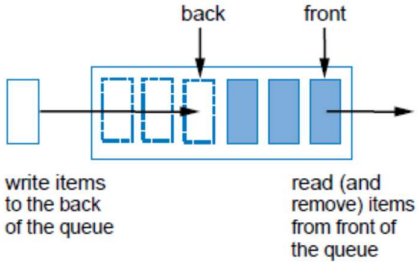
Lab session #10

Activity #1: Hardware FIFO

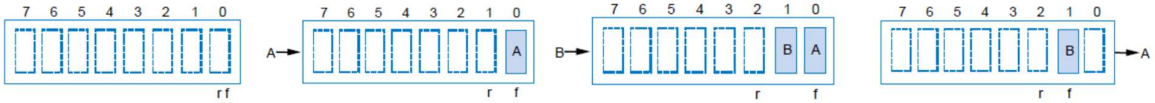
A queue is a component sometimes used during RTL design: it is written to at the back, it is read from the front. Writing is called an *enqueue*, reading is called a *dequeue*. As the first item written into a queue will be the first item read out, the queue is also called a *FIFO* (first-in-first-out):



Common use of queues includes:

- Computer keyboard: enqueues pressed keys onto queue, meanwhile dequeues and sends to computer
- Digital video recorder: enqueues captured frames, meanwhile dequeues frames, compresses them, and stores them
- Computer network routers: enqueue incoming packets onto queue, meanwhile dequeue packets, process destination information, and forward each packet out over appropriate port.

A queue is made of contiguous memory location, accessed by means of addresses and two pointers: *rear* and *front*, initially both pointing to 0. If the operation is an enqueue (write), the data is written to the address pointed to by *rear* and *rear* is incremented modulo the size of the queue. If the operation is a dequeue (read), the data is read from the address pointed to by *front*, *front* is incremented modulo the size of the queue.



Initially, then write A, followed by write B and by read

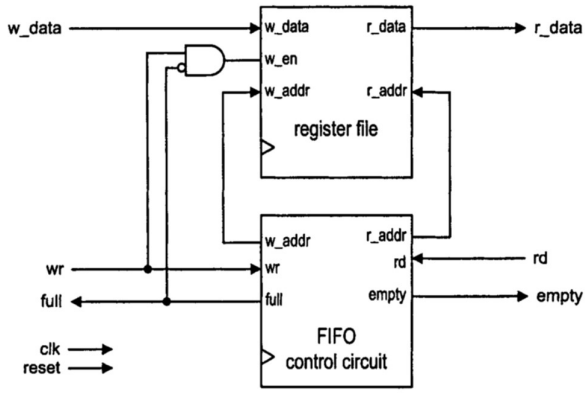
Two conditions are of interest:

- The queue is full: in an 8-entry queue, it means that 8 data are present and that no further enqueues are allowed until a dequeue occurs. The queue full condition causes front=rear
- The queue is empty: no dequeues are allowed until an enqueue occurs. The queue empty condition causes front=rear

As front=rear in both cases, the solution adopted in hardware to detect whether front=rear means full or empty resorts to a state machine that remembers if previous operation was enqueue or dequeue and accordingly sets full or empty output signal (respectively). Two flip-flops serve this purpose.

Design a 2^M-location FIFO for N-bit data as an FSM-D. Hint: according to the schematics in the following figure, use as **datapath** an 2^M x N register file with 1 write and 1 read port to store data. Input *wr* starts a synchronous enqueue operation, provided the FIFO is not full. Input *rd* starts a synchronous dequeue operation, provided the FIFO is not empty. Enqueue and dequeue operations may occur at the same time. The output data from the FIFO is always available. Outputs *full* and

empty signal the corresponding status of the FIFO. Reset is asynchronous. Design a **controller** FSM to properly handle enqueues/dequeues taking into account full/empty queue. Describe both datapath and controller behaviourally. Describe the FIFO in structural terms as a connection of datapath and controller.



Create a suitable testbench to test the FIFO on relevant values. Set N to 16 and M to 3.

Activity #2: Content-addressable memory

Content-addressable memory (**CAM**) is a special type of computer memory used in certain very-high-speed searching applications. It is also known as **associative memory** or **associative storage.** Unlike standard memory in which the user supplies a memory address and the RAM returns the data word stored at that address, a CAM is designed such that the user supplies a "key" and the CAM searches its entire memory to see if that key is stored anywhere in it. If the key is found, the CAM returns the data associated with the key.

Example: a network router that examines the 16-bit destination field of an incoming packet and routes it to one of the 8 output ports. A 4-word CAM stores information regarding the most frequently routed packets. The CAM includes four words, and each word is composed of a key-data pair. The **key** is the 16-bit destination field, the **data** is the 3-bit output port number.

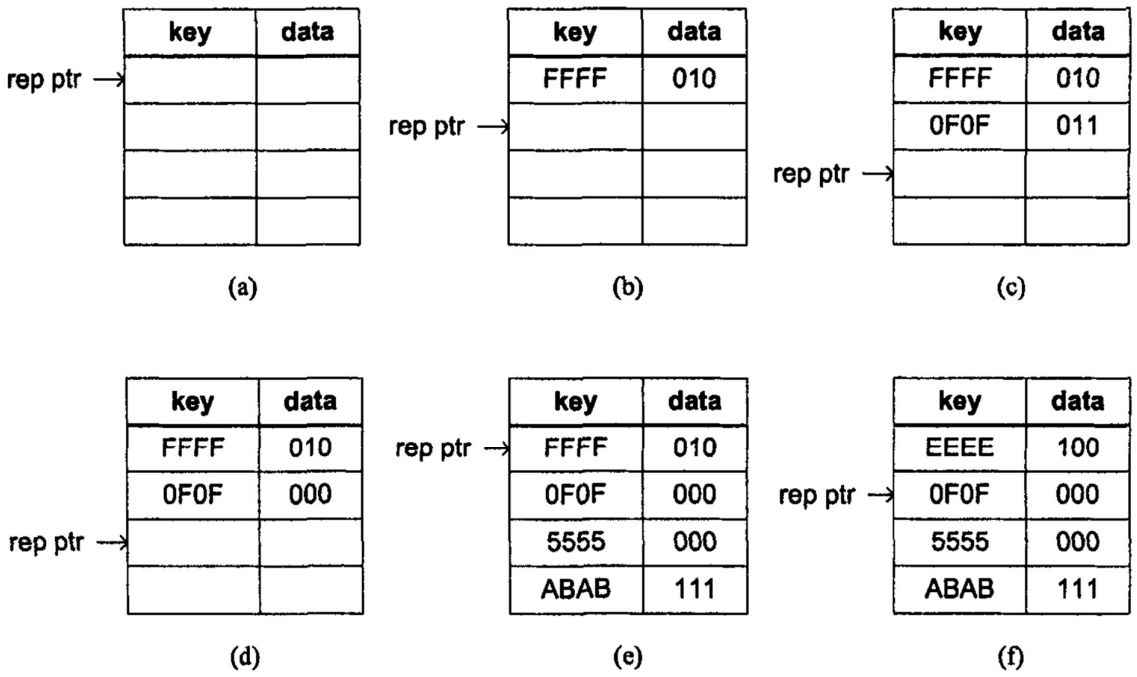
Two operations are supported:

- **Read**: the key is the input. If it is in the CAM, the data associated with the key will be routed to the output. The output signal *hit* indicates whether there is a match (search **hit**) or not (search **miss**)
- **Write**: the key is the input and the cases are as follows:
  - The input key matches an existing key in the CAM: the input data supersedes the existing data
  - The input key doesn't match an existing key in the CAM: the address for a write is stored in a replacement pointer *rep_ptr*. The term replacement comes from the need to remove an old key-data pair to make room for the new incoming pair, as the number of possible combinations for the key (2^{16}) is far greater than the 4-word capacity of the CAM. The replacement pointer may also point to an empty location.

The next figure reports the status of a CAM for some example cases:

a) initially empty CAM
b) write (FFFF, 010): no match, write occurs at location pointed to by *rep_ptr*, which is then incremented

c) write (0F0F, 011): no match, write occurs at location pointed to by *rep_ptr*, which is then incremented
d) write (0F0F, 000): match, replace 011 with 000, *rep_ptr* unchanged
e) write (5555, 000) then (ABAB, 111): no match, 2 successive write operations occur at location pointed to by *rep_ptr*, which is incremented each time
f) write (EEEE,100): no match, but CAM full, write occurs at location pointed to by *rep_ptr*. The content of the location pointed to by *rep_ptr* is discarded and overwritten with the new item.

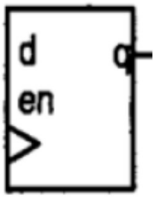


Design a 4-word, 16-bit CAM as an FSMD. Assumption: the stored key is unique, i.e. the number of matches is either 0 or 1. Hints: in the datapath separate the storage in 2: a register file for the data, accessed by means of its address and an array of 4 registers for the keys coupled with a matching circuit.

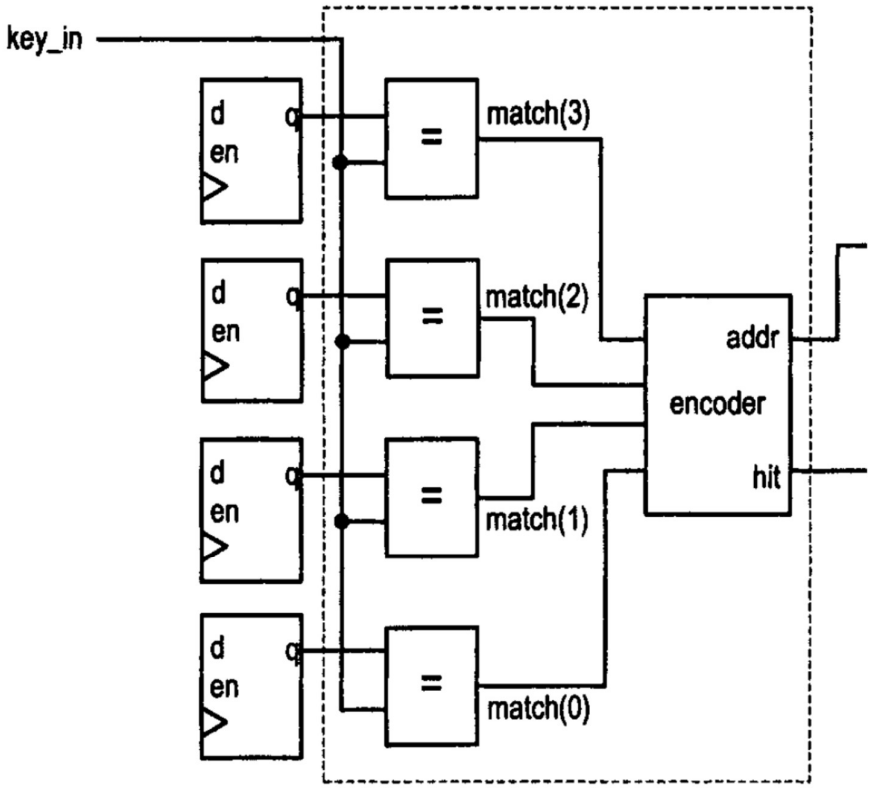
Signal *wr_en* selects whether is operation is write (*wr_en*=1) or read (*wr_en*=0).

Read: in the matching circuit, the output of each register of the key array is compared with the current value of the input key (i.e., the *key_in* signal) and a 1-bit matching signal (i.e., the *match(i)* signal) is generated accordingly. Since each stored key is unique, at most one can be matched. If for the *key_in* signal value there is a hit, a one bit *match* signal is asserted and a 2^2-to-2 binary encoder generates the binary code of the matched location *addr*, output of the matching circuit. Signal *addr* in this case is routed on signal *addr_out*, connected to the read port address *r_addr* of the data file, and the corresponding data item is routed to the output. The output is not valid if the *hit* signal is not asserted. Register *rep_ptr* is updated.

In the following



represents a 16-bit register used to store the key. The following schematics shows the matching unit:

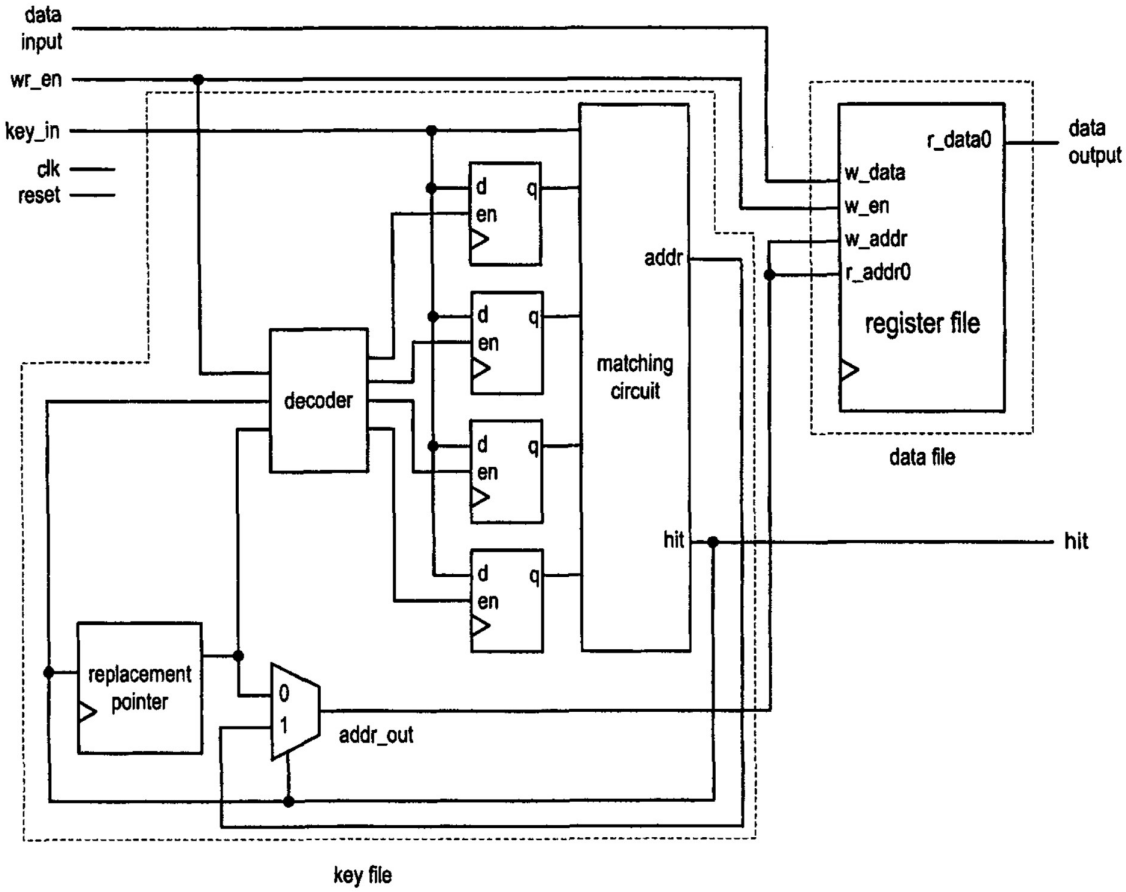


matching unit

Write: given the key-data pair, if the key is a hit, the matching circuit will generate *addr_out*, connected to the write port address *w_addr* of the data file and the input data will be stored into the corresponding location. If the key is a miss, several tasks must be performed:

- Find an available register in the key array resorting to *rep_ptr*
- Store the input key into this register and update *rep_ptr*
- Store the data into the corresponding address in the data file.

The following schematics represents the CAM:



Note that the schematics doesn't distinguish between control and status signals. Identify them , so as to comply with the FSMD model that has a datapath and a controller.

Design three entities: *RegFileGen* for the register file containing the data, *KeyFile* for the key registers and the matching unit and *ReplPtr* for the register containing the replacement pointer and then interconnect them in a structural description of the datapath. The architectures of the 3 entities follow the behavioral description style. Design the controller and interconnect it in a structural description with the datapath to yield the CAM. Create a suitable testbench to test the CAM on relevant values.