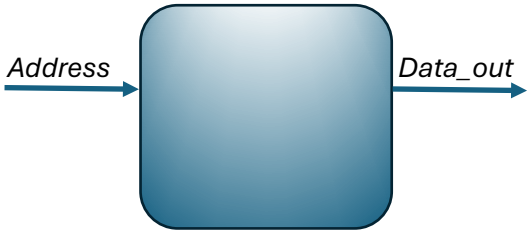
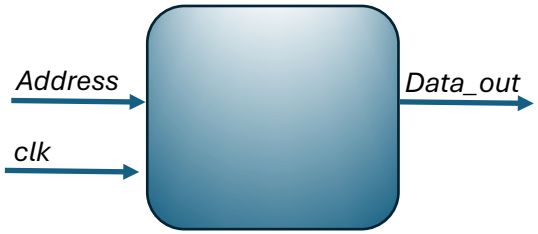
General Types of Memories in Hardware Design

ROM
Read Only Memory



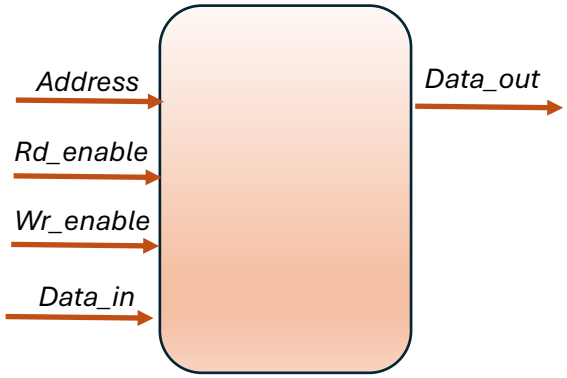
Asynchronous ROM design – no clock



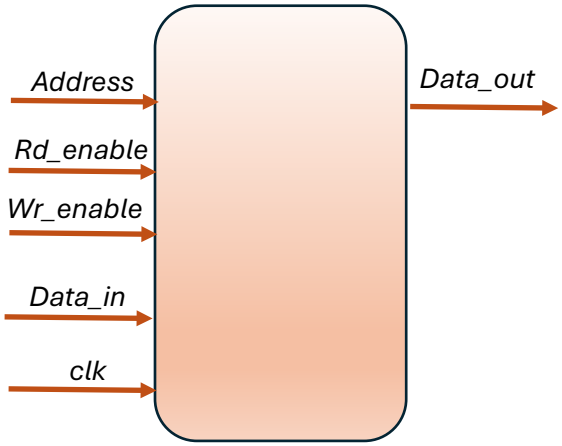
Synchronous ROM design – clocked

Any ROM design has an input address to determine the location of the memory register to be accessed. ROMs do not have data to be written into the memory, as they are ***Read ONLY.*** All ROMs have an output data port.

RAM
Read And Write Memory



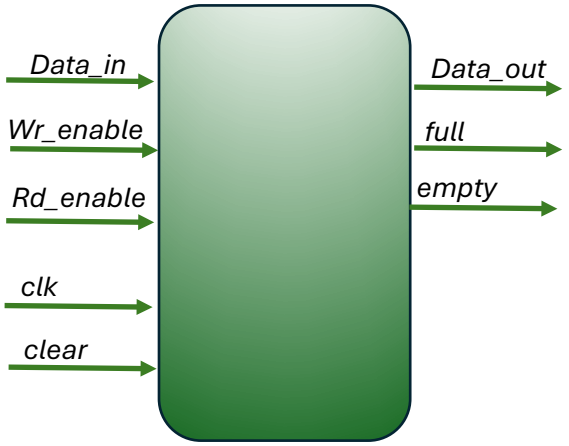
Asynchronous RAM design



Synchronous RAM design

The RAM may have one clock for the read and one clock for the write, or just a single clock for both. Some RAMs can read and write at the same time, and will have different read and write address lines.

FIFO
First In First Out



Synchronous FIFO design

Notice that the FIFO ***does not have address input ports***. The FIFO internal pointers decide where to write the incoming data, and what data to read based on a First_in First_out sequence. If the stack of FIFO memory is full, and a wr_enable request is placed at the input, the incoming data will be ignored because the stack is full. Only after some data are read and the full flag is low, more data can be written in the stack of memory. Also, if all the data has been read, the empty flag will be raised. The full and empty flags are internally managed based on the read and write pointers and the difference between them.

ROM Design Examples:

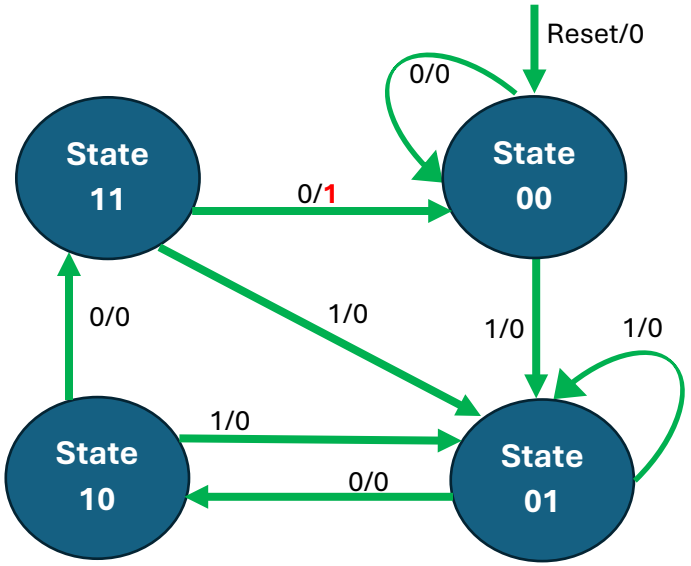
1. **Look-up table for data conversion: Compare the conversion code using the case structure versus the ROM**

```
module Binary_7segment (BCD, SevenSegment);
input       [3:0]  BCD;
output reg  [6:0]  SevenSegment;

parameter BLANK = 7'b111_1111;  // 127
parameter ZERO  = 7'b100_0000;  //  64
parameter ONE   = 7'b111_1001;  // 121
parameter TWO   = 7'b010_0100;  //  36
parameter THREE = 7'b011_0000;  //  48
parameter FOUR  = 7'b001_1001;  //  25
parameter FIVE  = 7'b001_0010;  //  18
parameter SIX   = 7'b000_0001;  //   1
parameter SEVEN = 7'b111_1000;  // 120
parameter EIGHT = 7'b000_0000;  //   0
parameter NINE  = 7'b001_1000;  //  24

always @ *
case (BCD)
   4'b0000 : SevenSegment = ZERO;
   4'b0001 : SevenSegment = ONE;
   4'b0010 : SevenSegment = TWO;
   4'b0011 : SevenSegment = THREE;
   4'b0100 : SevenSegment = FOUR;
   4'b0101 : SevenSegment = FIVE;
   4'b0110 : SevenSegment = SIX;
   4'b0111 : SevenSegment = SEVEN;
   4'b1000 : SevenSegment = EIGHT;
   4'b1001 : SevenSegment = NINE;
   default : SevenSegment = BLANK;
   endcase

endmodule
```

```
module Binary_7segment_converter (BCD, SevenSegment);
  input       [3:0]  BCD;
  output      [6:0]  SevenSegment;

  // Instantiation of the ROM look-up-table
  Binary_7segment_ROM   lookUpTable  (.ROM_address (BCD),
                                      .ROM_data    (SevenSegment) );

  endmodule

  module Binary_7segment_ROM  (ROM_address, ROM_data);

    input       [3:0]  ROM_address;
    output      [6:0]  ROM_data;

    reg  [6:0] ROM  [15:0];     // width is 7 bits and depth is 16 words

    parameter BLANK = 7'b111_1111;  // 127
    parameter ZERO  = 7'b100_0000;  //  64
    parameter ONE   = 7'b111_1001;  // 121
    parameter TWO   = 7'b010_0100;  //  36
    parameter THREE = 7'b011_0000;  //  48
    parameter FOUR  = 7'b001_1001;  //  25
    parameter FIVE  = 7'b001_0010;  //  18
    parameter SIX   = 7'b000_0001;  //   1
    parameter SEVEN = 7'b111_1000;  // 120
    parameter EIGHT = 7'b000_0000;  //   0
    parameter NINE  = 7'b001_1000;  //  24

    initial
    begin
       ROM [0]  = ZERO;
       ROM [1]  = ONE;
       ROM [2]  = TWO;
       ROM [3]  = THREE;
       ROM [4]  = FOUR;
       ROM [5]  = FIVE;
       ROM [6]  = SIX;
       ROM [7]  = SEVEN;
       ROM [8]  = EIGHT;
       ROM [9]  = NINE;
       ROM [10] = BLANK;
       ROM [11] = BLANK;
       ROM [12] = BLANK;
       ROM [13] = BLANK;
       ROM [14] = BLANK;
       ROM [15] = BLANK;
    end

    endmodule
```

2. **Look-up table to find the next_state and output value for a FSM:**

Mealy FSM to detect a 1000 sequence



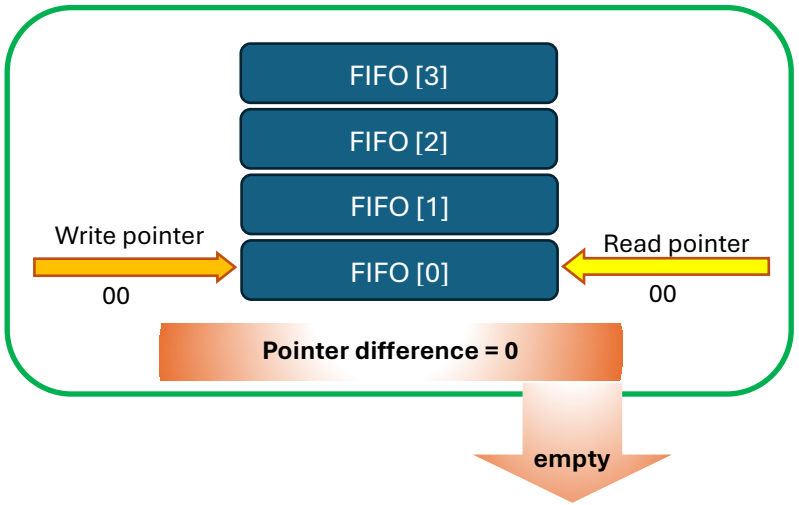
state	x_in	next_state	z_out
00	0	00	0
00	1	01	0
01	0	10	0
01	1	01	0
10	0	11	0
10	1	01	0
11	0	00	1
11	1	01	0

```
module ROM_FSM (clk, reset, z_out, x_in);

input          clk, reset, x_in;
output  reg    z_out;

reg    [1:0]   state;
wire   [1:0]   next_state;
wire   [2:0]   address, data_out;


always @ (posedge clk)
  if (reset)  begin
        state <= 2'b00;
        z_out <= 1'b0; end
   else begin
        state <= next_state;
        z_out <= z;    end

// Combinational logic to find next_state and z

assign    address    = {state, x_in};
assign    next_state =  data_out [2:1];
assign    z          =  data_out [0];

ROM lookUpTable  (address, data_out);

endmodule


 module ROM (address, data_out);
 input       [2:0]   address;
 output reg  [2:0]  data_out;

 reg   [2:0]  ROM_reg   [7:0] ;

 initial
 begin
 ROM_reg [0] = 3'b000;
 ROM_reg [1] = 3'b010;
 ROM_reg [2] = 3'b100;
 ROM_reg [3] = 3'b010;
 ROM_reg [4] = 3'b110;
 ROM_reg [5] = 3'b010;
 ROM_reg [6] = 3'b001;
 ROM_reg [7] = 3'b010;
 end

 endmodule
```

Example designs for two RAM modules:

```
module RAM_ONE_clk (
  input          clk,          // one clk for read and write
  input  [ 5:0] address,       // one address for read and write     (log2(memory depth) log2(64) = 6
  input          mem_read,     // read enable
  input          mem_write,    // write enable
  input  [15:0] data_in,
  output [15:0] data_out
  );

 reg [15:0] memory [63:0];        // The memory has a depth of 64 words (addresses from 0 to 63)

 always @(posedge clk)
       if(mem_write)   memory[address] <= data_in;    // synchronized write


 assign data_out = (mem_read == 1'b1) ? memory[address] : 16'd0;  // combinational logic for data_out

 endmodule



 module RAM_clear (
  input               clk,              // one clk for read and write
  input       [ 5:0] address,           // (log2(memory depth) log2(64) = 6
  input              read_or_write,     // read = 0, write = 1
  input              clear,             // clear memory option
  input       [15:0] data_in,
  output  reg [15:0] data_out           // registered output
  );

 reg     [15:0] memory [63:0];        // The memory has a depth of 64 words (addresses from 0 to 63)
 integer         k;

 always @(posedge clk)

          if (clear)
               for (k = 0 ; k < 64 ; k = k+1)
                    memory[k] <= 16'd0;
    else  if (read_or_write)
                    memory[address] <= data_in;    // priority to write
    else  if (~read_or_write)
                    data_out        <= memory[address];

 endmodule
```

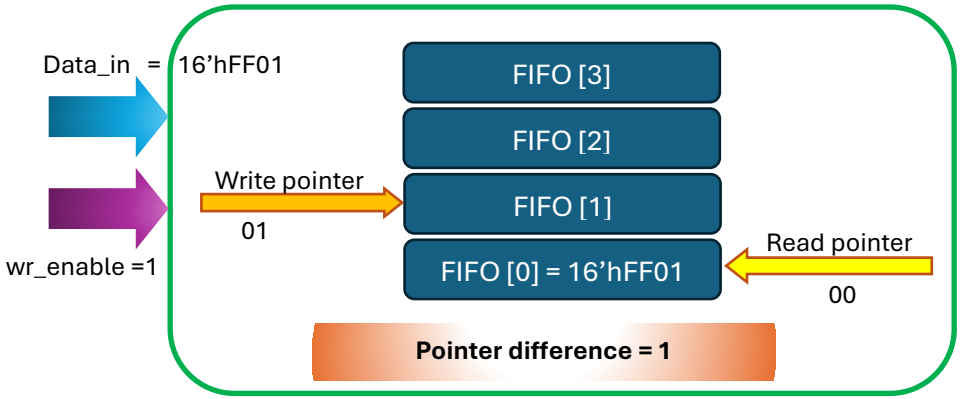
FIFO Explanation:



A FIFO with a memory stack of just 4 memory locations (words). The width of each word is 16 bits [15:0]. As we have 4 memory locations (FIFO depth is 4), we need 2 bits to address the memory locations, each by a unique address.

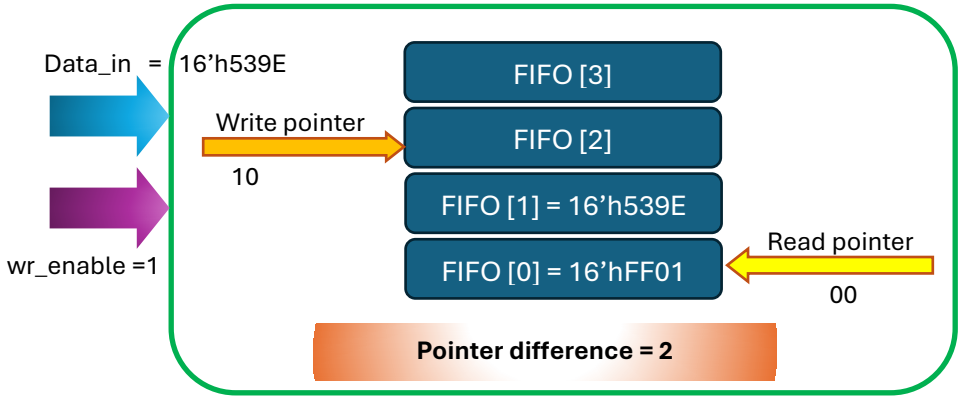
Before writing anything to the FIFO, both the read and write pointers will be pointing at address 00.

The pointer difference will be 0 indicating that the FIFO is empty. No reading will be allowed as there is no saved data.



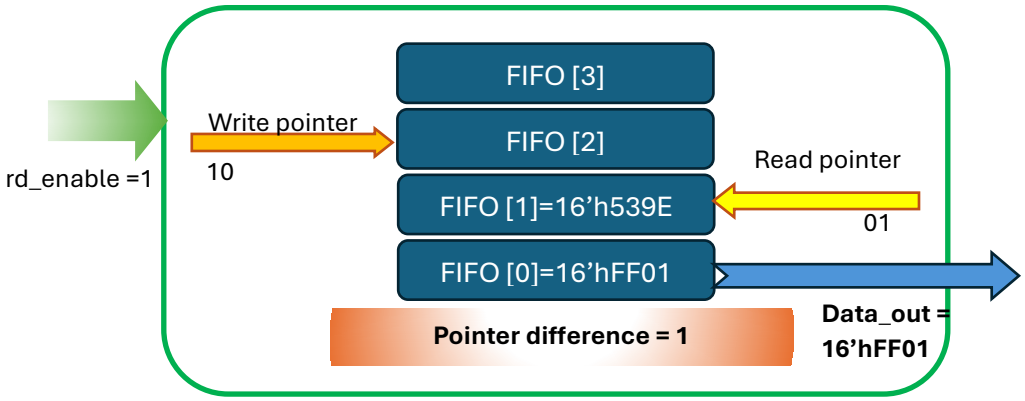
When a wr_enable is activated, the value on the Data_in will be passed to the FIFO and written in the memory location pointed at by the write pointer. Then, the write pointer will be incremented by 1 (00+1=01 in binary), and the pointer difference will be incremented by 1 (000+1 = 001 in binary). The empty flag will be now 0. The FIFO is not empty anymore. It stores a value that was not read yet.

The pointers and memory locations will be updated by the assigned values at the positive clock edge.

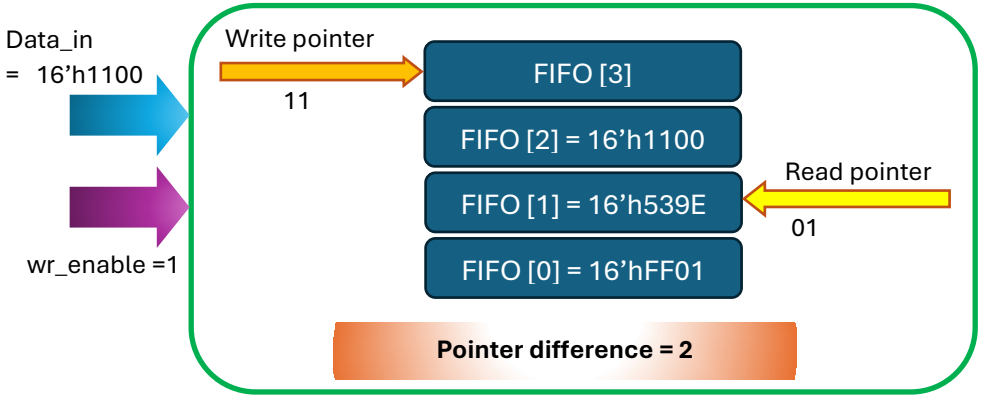


If the wr_enable is active again, the value on the Data_in will be passed to the FIFO and written in the memory location pointed at by the write pointer. Then, the write pointer will be incremented by 1 (01+1=10 in binary), and the pointer difference will be incremented by 1 (001+1 = 010 in binary). The FIFO is not empty and it stores two values that were not read yet.

The pointers and memory locations are updated by the assigned values at the positive clock edge.

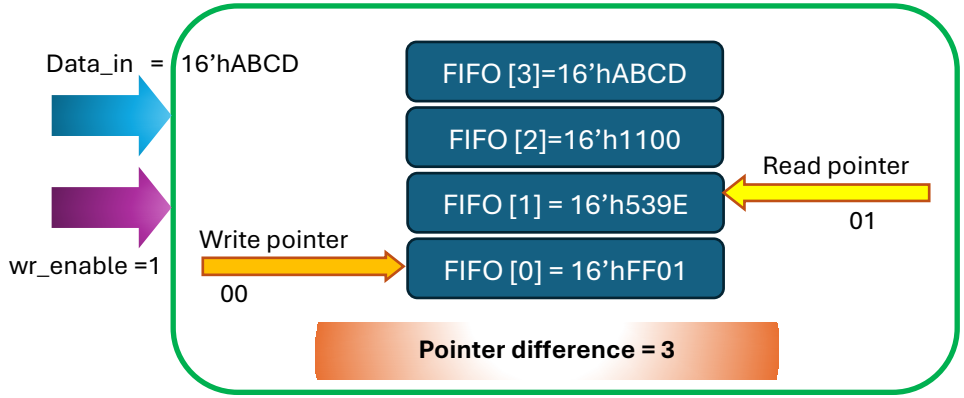


If the read enable is activated, then the data content saved in the memory location pointed at by the read pointer will be sent to the output data port. Then Read pointer will be incremented by 1 (00+1=01 in binary). The pointer difference will be decremented by 1 (10 – 1=01 in binary) This indicates that there is eaxatly one data saved in the FIFO that was written and not read yet.(That is the 16'h539E saved in the memory location 01). Now what about the 16'hFF01 value saved in memory location 00? This value has been already read by now, it is OK to overwrite it! We will not delete it, but we can overwrite in when the memory space is needed for new incoming values.



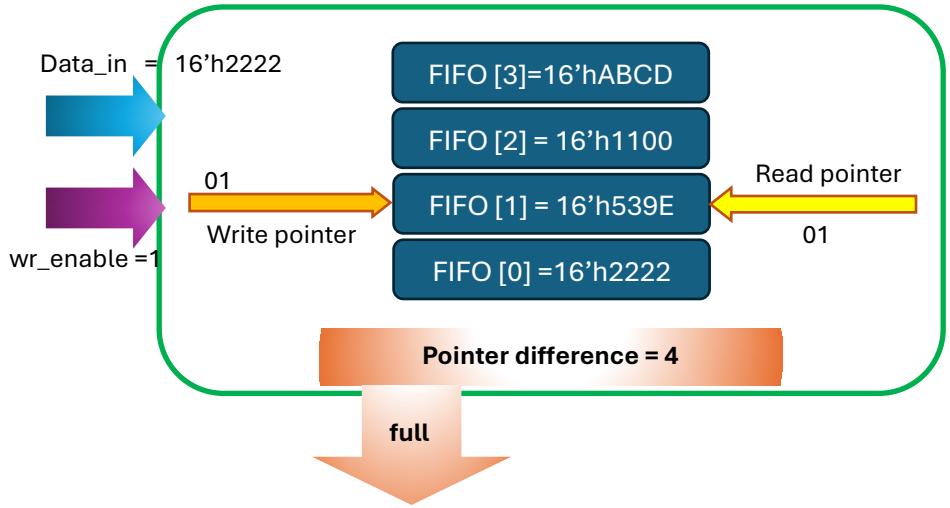
When a wr_enable is activated, the value on the Data_in will be passed to the FIFO and written in the memory location pointed at by the write pointer. Then, the write pointer will be incremented by 1 (10+1=11 in binary), and the pointer difference will be incremented by 1 (001+1 = 010 in binary). The FIFO is not empty, it stores 2 values that were not read yet.

The pointers and memory locations will be updated by the assigned values at the positive clock edge.

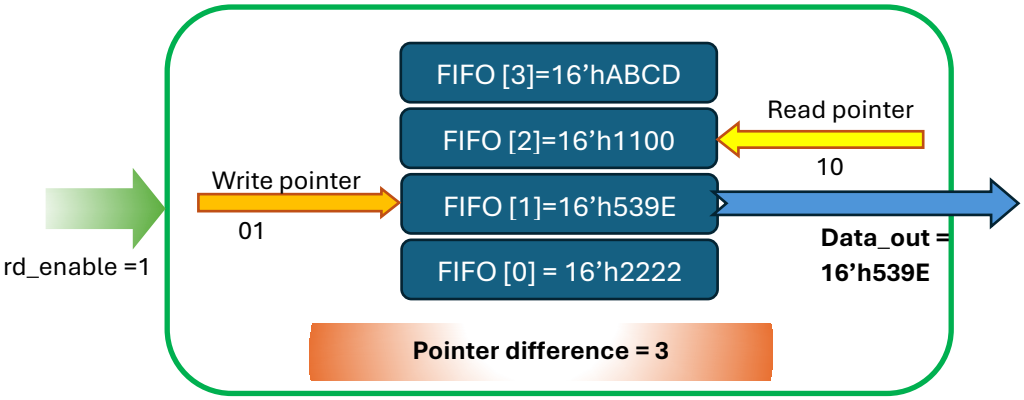


If the wr_enable is active again, the value on the Data_in will be passed to the FIFO and written in the memory location pointed at by the write pointer. Then, the write pointer will be incremented by 1 (11+1=00 in binary, as the overflow bit does not exist 100 with the size of the address not large enough), and the pointer difference will be incremented by 1 (010+1 = 011 in binary). The FIFO is not empty and it stores 3 values that were not read yet. This number is saved by the pointer difference.

The pointers and memory locations are updated by the assigned values at the positive clock edge.



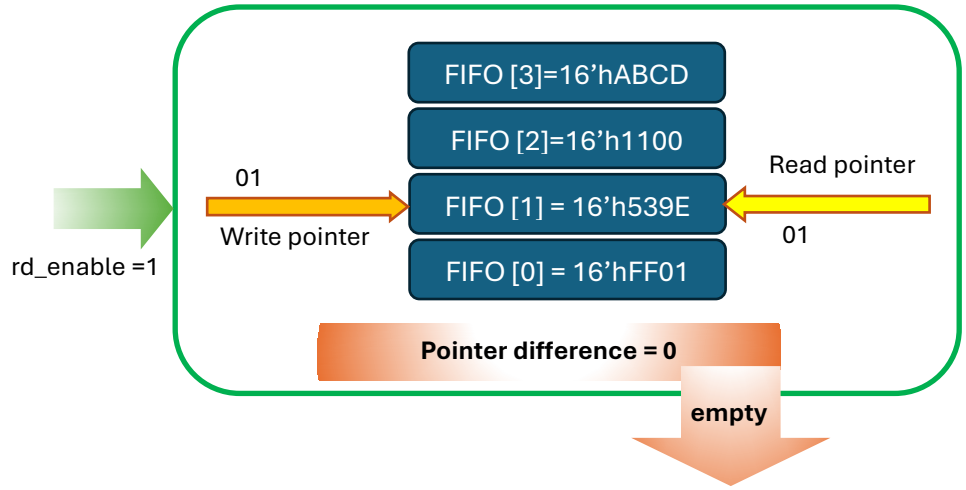
If the wr_enable is active again, the value on the Data_in will be passed to the FIFO and written in the memory location pointed at by the write pointer. Then, the write pointer will be incremented by 1 (01+1=10 in binary), and the pointer difference will be incremented by 1 (011+1 = 100 in binary). The FIFO has now 4 values that were not read yet. 4 is the depth of the FIFO memory. This means that the FIFO is full. No more data can be written. One or more words need to be read before anymore data is accepted. The pointer difference register will be always wider by 1 bit compared to the read and write pointer registers.



If the read enable is activated, then the data content saved in the memory location pointed at by the read pointer will be sent to the output data port. Then the Read pointer will be incremented by 1 (01+1=10 in binary). The pointer difference will be decremented by 1 (100 – 1= 011 in binary) This indicates that there is exactly 3 data saved in the FIFO that was written and not read yet. Now the 16'h539E value saved in memory location 0 has been already, and it is fine to overwrite it! We will not delete it, but we can overwrite in when the memory space is needed for new incoming values. Now the FIFO is not completely full. Data can be accepted to be written

....

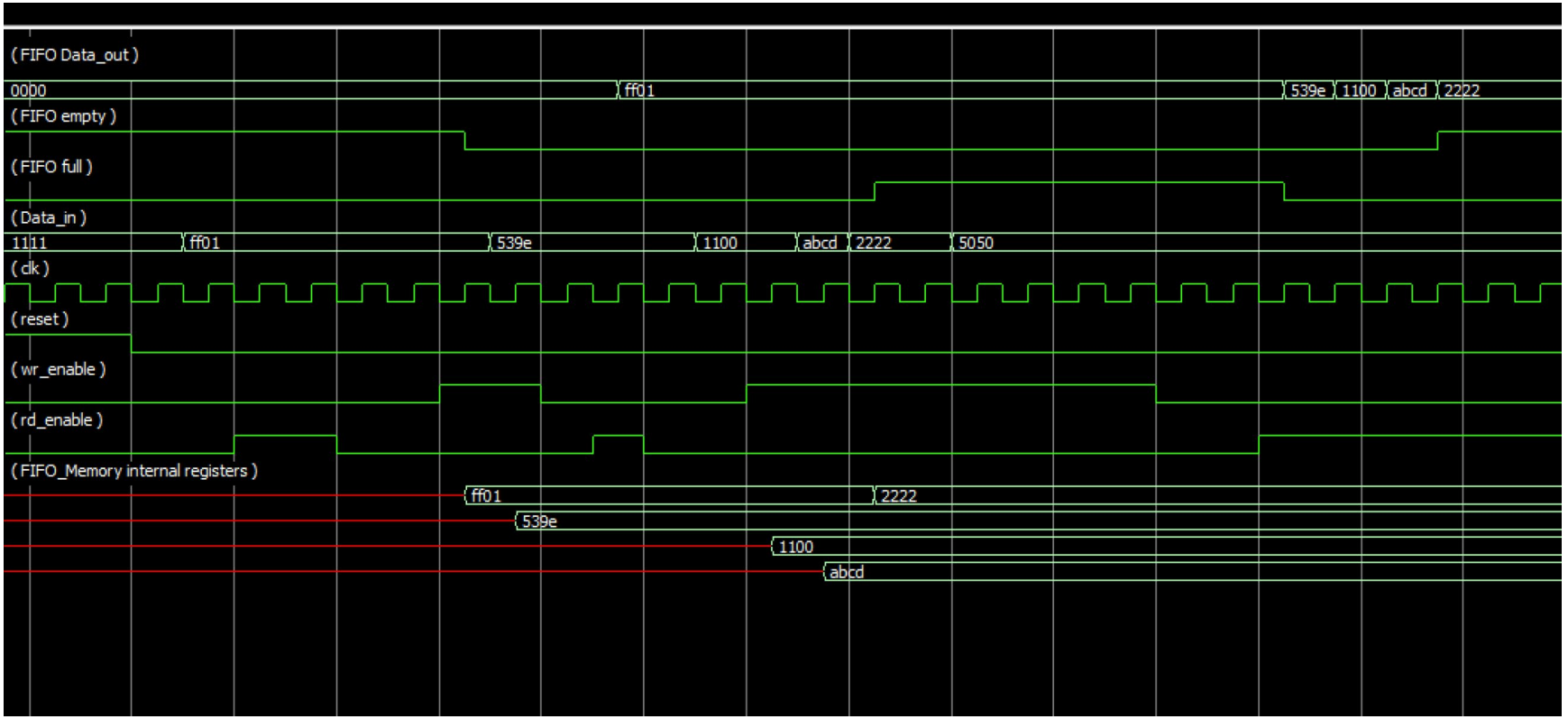
....



... If the rd_enable is active again over 3 more clock cycles, every time the data is read, the read pointer increments by 1 and the pointer difference decreases by 1. When the read pointer reaches the value on the write pointer (01) in this case, the pointer difference would have reached 0 indicating that there is no unread data in the FIFO anymore. The FIFO is empty.

```
module FIFO (   Data_out,               // Data path from FIFO
                empty,                  // Flag asserted high for empty stack
                full,                   // Flag asserted high for full stack
                Data_in,                // Data path into FIFO
                wr_enable,              // Input controlling a write to FIFO memory stack
                rd_enable,              // Input controlling a read to stack
                clk,                    // clock
                rst                     // reset
                    );

parameter   FIFO_width  = 16;          // width of stack and data paths
parameter   FIFO_height = 4;           // height of stack (# of words)
parameter   ptr_width   = 2;           // width of read and write pointers to address the FIFO words

output  reg    [FIFO_width -1 : 0]   Data_out;
output                               empty, full;
input          [FIFO_width -1 : 0]   Data_in;
input                                clk, rst;
input                                wr_enable, rd_enable;

// Pointers for reading and writing
reg  [ptr_width -1  : 0]  read_ptr, write_ptr;
reg  [ptr_width     : 0]  ptr_diff;

// Memory FIFO registers
reg  [FIFO_width -1 : 0]  FIFO_Mem  [FIFO_height -1 : 0];     // memory array

// empty and full flags
assign     empty = (ptr_diff == 0)           ? 1'b1 : 1'b0;
assign     full  = (ptr_diff == FIFO_height) ? 1'b1 : 1'b0;


always  @ (posedge clk or posedge rst)
   begin
        if (rst) begin
                Data_out    <= 0;
                read_ptr    <= 0;
                write_ptr   <= 0;
                ptr_diff    <= 0;
                end
          else  if ( (rd_enable) && (!empty)  )    // read request and FIFO not empty
                   begin
                      Data_out   <= FIFO_Mem [read_ptr];
                      read_ptr   <= read_ptr + 1;
                      ptr_diff   <= ptr_diff - 1;
                      end
          else   if ( (wr_enable) && (!full)   )    // write request and FIFO not full
                   begin
                      FIFO_Mem [write_ptr]   <= Data_in;
                      write_ptr              <= write_ptr + 1;
                      ptr_diff               <= ptr_diff  + 1;
                      end
   end

endmodule
```

```
module FIFO_tb ();
   parameter   FIFO_width  = 16;          // width of stack and data paths
   parameter   FIFO_height = 4;           // height of stack (# of words)
   parameter   ptr_width   = 2;           // width of pointer to address stack

   wire     [FIFO_width -1 : 0]   Data_out;
   wire                           empty, full;
   reg      [FIFO_width -1 : 0]   Data_in;
   reg                            clk, rst;
   reg                            wr_enable, rd_enable;

   wire     [FIFO_width-1: 0]     fifo_0, fifo_1, fifo_2, fifo_3;

   FIFO  UUT   ( Data_out, empty, full, Data_in, wr_enable, rd_enable, clk, rst );

   assign   fifo_0  = UUT.FIFO_Mem [0];
   assign   fifo_1  = UUT.FIFO_Mem [1];
   assign   fifo_2  = UUT.FIFO_Mem [2];
   assign   fifo_3  = UUT.FIFO_Mem [3];

   initial
   begin
   clk = 0;
   rst = 1;
   rd_enable = 0;
   wr_enable = 0;
   Data_in = 16'h1111;
   #350 $stop;
   end

   initial
   begin  forever
   #5 clk = ~clk;   end

  initial
  fork
  #40   rst       = 1'b0;
  #50   Data_in   = 16'hFF01;
  #60   rd_enable = 1'b1;
  #80   rd_enable = 1'b0;
  #100  wr_enable = 1'b1;
  #110  Data_in   = 16'h539E;
  #120  wr_enable = 1'b0;
  #130  rd_enable = 1'b1;
  #140  rd_enable = 1'b0;
  #150  Data_in   = 16'h1100;
  #160  wr_enable = 1'b1;
  #170  Data_in   = 16'hABCD;
  #180  Data_in   = 16'h2222;
  #200  Data_in   = 16'h5050;
  #240  wr_enable = 1'b0;
  #260  rd_enable = 1'b1;
  join

  endmodule
```



If the FIFO is empty, a read enable signal will not result in accessing any memory value. When empty, the Data_out will keep its value.