Exercise 6: Memory Architectures

Prof. Dr. Sven Simon

1 Division ROM

Expensive math operations such as divisions can be replaced by a ROM that is initialized by precalculated values. Implement a module that calculates the **quotient** and **remainder** of the division of an input by a constant divisor. Make sure the design is parametrizable. The parameters are the data width of the input, the divisor value, and whether the design should be sequential or combinational. Hint: *It might be convenient to use a package in which you define constants, functions to calculate data widths, and functions to intialize the ROMs.*

2 Dual-port RAM

Implement a simple dual-port RAM with single clock (synchronous write and synchronous read). Avoid using shared variables for that. The entity is given below (also in `dual_port_ram.vhd`):

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.math_real.all;

entity dual_port_ram is

  generic (
    ADDR_WIDTH : natural := 6;
    DATA_WIDTH : natural := 8);

  port (
    clk   : in  std_logic;
    wr    : in  std_logic;
    addra : in  std_logic_vector(ADDR_WIDTH-1 downto 0);
    dia   : in  std_logic_vector(DATA_WIDTH-1 downto 0);
    rd    : in  std_logic;
    addrb : in  std_logic_vector(ADDR_WIDTH-1 downto 0);
    dib   : out std_logic_vector(DATA_WIDTH-1 downto 0));

end entity dual_port_ram;
```

When does the tool synthesize the memory as a distributed RAM. When does it synthesize the memory as a block RAM? *Optional:* What if the entries are not 2^{DATA_WIDTH}? Pass the number of entries instead of `DATA_WIDTH` as a generic.

3 First-In-First-Out

First-in-first-out (FIFO) buffers act as storage between two subsystems. One subsystem stores data into the buffer, and the other sybsystem retrieves the data and removes it from the buffer. The data is retrieved in the same order it is stored, and hence the name. One common way to construct a FIFO buffer is by deploying a dual port RAM and some control logic

around it (see Figure 1). The RAM is arranged as a circular queue and two pointers are used to mark the beginning and end of the FIFO buffer. The first pointer, known the *write pointer* points to the first empty slot in front of the buffer. During a write operation, the data is stored in this designated slot, and write pointer is incremented. The second pointer, known as *read pointer* points to the end of the buffer. During a read operation, the data is retieved and the read pointer is incremented. Initially the two pointers are at zero. In order to ensure correct operation, a FIFO buffer must include a *full* and *empty* signals. The external subsystems should check these signals in order to avoid overflow and underflow. Initially, the FIFO buffer is empty and not full. When the FIFO buffer is empty, the read pointer is the same as the write pointer. Unfortunately, this is also the case when the FIFO is full!
One way to implement the full and empty logic is described here. Two extra flip-flops are needed to record the empty and full status. Initially the full FF is set to '0' and the empty FF is set to '1'. After initialization, the `wr` and `rd` are examined at the rising edge of the clock, and the pointers and the FFs are modified according to the following rules:

- When `wr` and `rd` = "00":
  pointers and FFs remain unchanged.
- When `wr` and `rd` = "11":
  write and read operations are performed simultaneously. Apparently in this case the empty and full FFs will not change. The two pointers are incremented.
- When `wr` and `rd` = "10":
  In this case we must first make sure that the buffer is not full. If it is not full, the write pointer advances one position and the empty FF should be deasserted. The advancement may make the buffer full. This happens if the *next value* of the write pointer is equal to the current value of the read pointer. If this is the case, the full FF will be set to '1'.
- When `wr` and `rd` = "01":
  In this case we must first make sure that the buffer is not empty. If it is not empty, the read pointer advances one position and the full FF should be deasserted. The advancement may make the buffer empty. This happens if the *next value* of the read pointer is equal to the current value of the write pointer. If this is the case, the empty FF will be set to '1'.

Implement a fifo, given the entity declaration below (also in `fifo.vhd`):

```
entity fifo is
  generic (
    ADDR_WIDTH : natural := 10;
    DATA_WIDTH : natural := 8);

  port (
    clk   : in  std_logic;
    wr    : in  std_logic;
    di    : in  std_logic_vector(DATA_WIDTH-1 downto 0);
    rd    : in  std_logic;
    do    : out std_logic_vector(DATA_WIDTH-1 downto 0);
    empty : out std_logic;
    full  : out std_logic);

end entity fifo;
```

4 Single-port RAM

Implement a single-port RAM with single clock (synchronous write and synchronous read). The module should be able to generate a *write-first* or a *read-firs*t RAM. Write-first means that the new content is immediately made available for reading, whereas read-first means old content is read before the new content is loaded. Avoid using shared variables for that. The entity declaration is given below (also in `single_port_ram.vhd`):

```
entity single_port_ram is
```

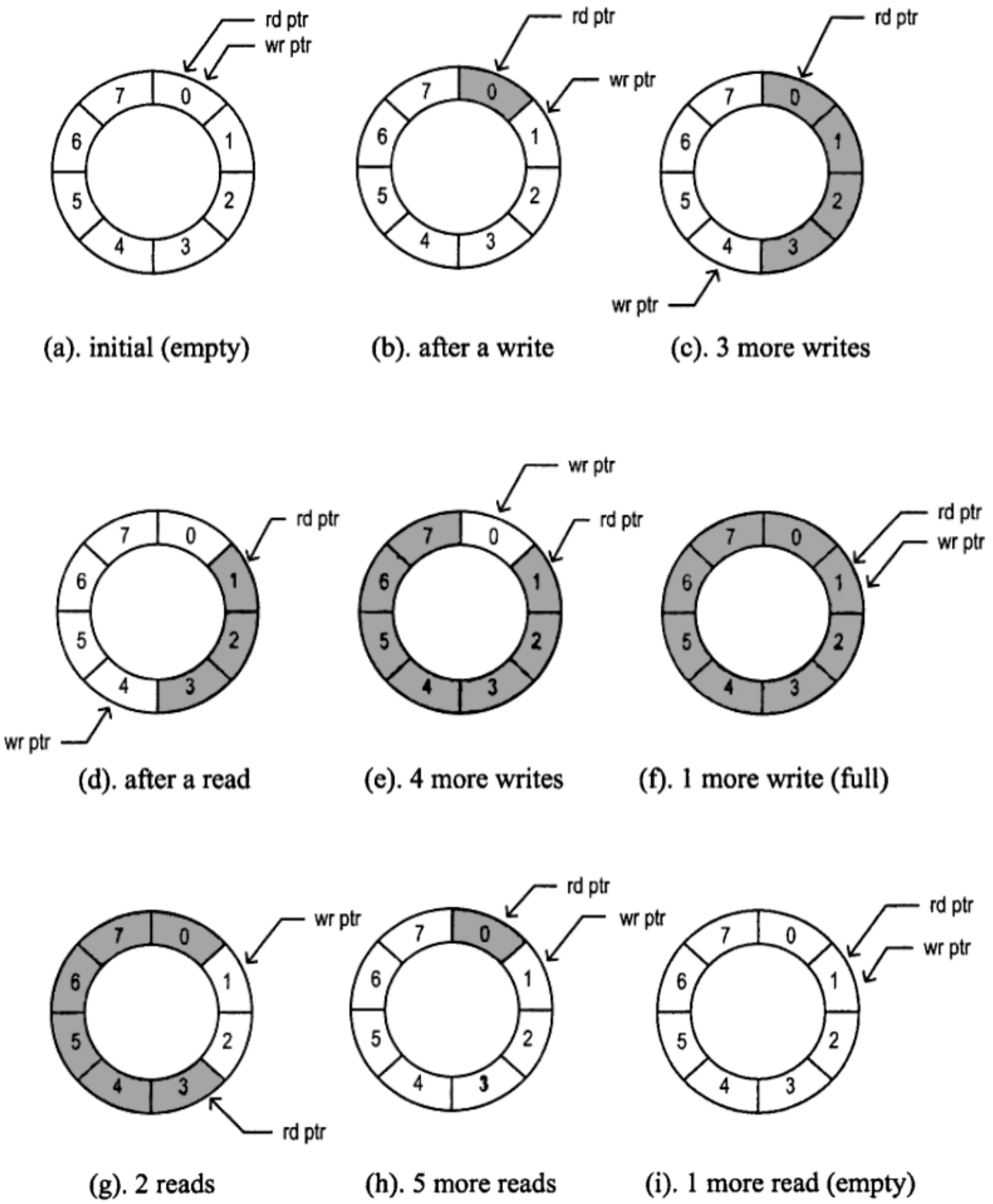

Figure 1: Circular-queue implementation of a FIFO buffer.

```
generic (
  MODE        : string  := "write_first";
```

```
    ADDR_WIDTH : natural := 10;
    DATA_WIDTH : natural := 8);

  port (
    clk  : in  std_logic;
    ce   : in  std_logic;
    wr   : in  std_logic;
    rd   : in  std_logic;
    addr : in  std_logic_vector(ADDR_WIDTH-1 downto 0);
    di   : in  std_logic_vector(DATA_WIDTH-1 downto 0);
    do   : out std_logic_vector(DATA_WIDTH-1 downto 0));

end entity single_port_ram;
```

5 Stack

A stack is a memory structure that allows the data items to be visited twice, although, contrary to a FIFO buffer, the second time the items are accessed in the reverse order. Items are *pushed* onto a stack by storing them in successive memory locations. When an item is *popped* from the stack, the most recently saved item is retrieved first. A stack is typically implemented around a read-first single port ram and some control logic.
The block diagram of a stack in Figure 2. In the figure, a multiplexer is used to add either 1 or -1 to *SP* depending on whether a *push* or a *pop* is performed, respectively. A push uses SP directly as the RAM address, whereas a pop uses the decremented SP as the address. The stack is empty if the address is zero. The stack is full after the last memory element is written. Incrementing *SP* will wrap it past the end. This is prevented by making *SP* one bit longer than needed to address the memory. The stack is then full when the most significant bit of SP is set.
Describe the stack in VHDL. The entity declaration is given below (also in `stack.vhd`):

```
entity stack is

  generic (
    ADDR_WIDTH : natural := 8;
    DATA_WIDTH : natural := 5);

  port (
    clk   : in  std_logic;
    push  : in  std_logic;
    empty : out std_logic;
    di    : in  std_logic_vector(DATA_WIDTH-1 downto 0);
    pop   : in  std_logic;
    full  : out std_logic;
    do    : out std_logic_vector(DATA_WIDTH-1 downto 0));

end entity stack;
```

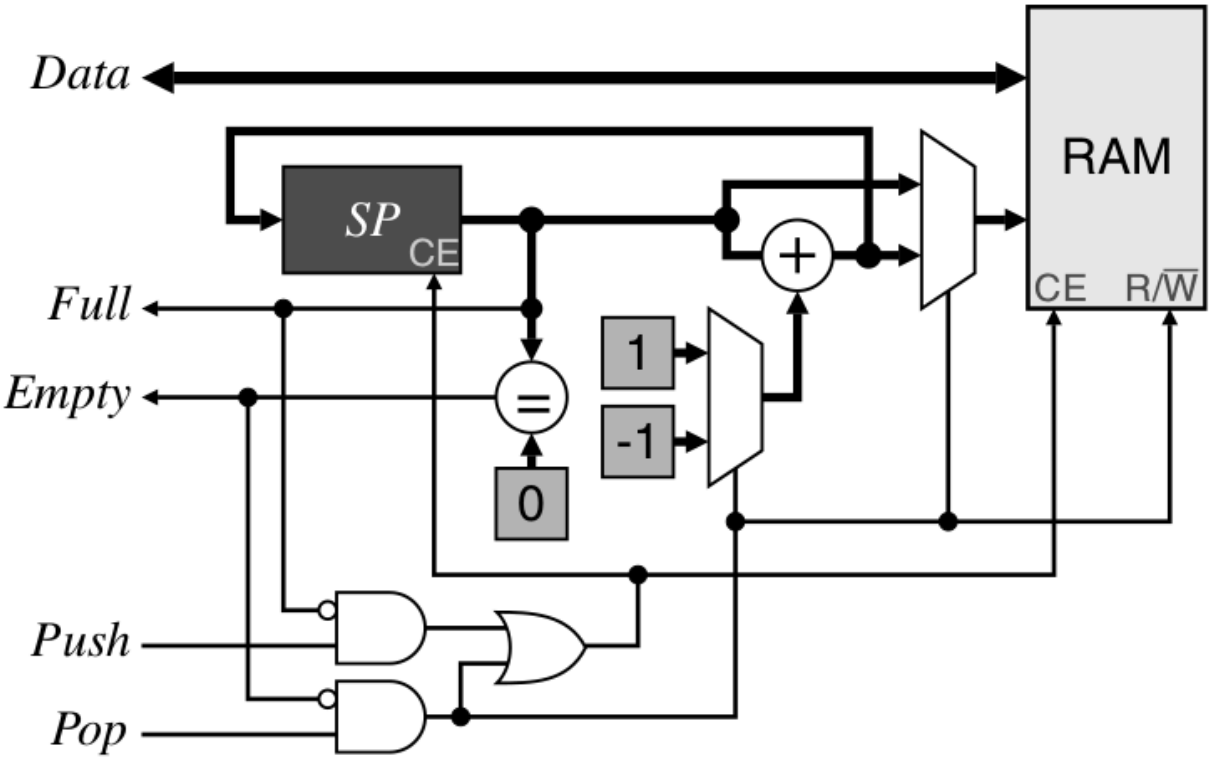

Figure 2: Block diagram of a stack.