

Design of Digital Circuits and Systems

Memory I

Instructor: Vikram Iyer

Teaching Assistants:

Ariel Kao

Josh Wentzien

Selim Saridede

Jared Yoder

Derek Thorp

Relevant Course Information

- ❖ Lab 1 report due tomorrow (4/11)
- ❖ Lab 1 demo due by end of 4/18
 - Lab Demo Slot assignment coming tomorrow
- ❖ Lab 2 report due next Friday (4/18)
- ❖ Homework 2 due next Wednesday (4/16)
- ❖ Use Ed Discussion to ask course questions
 - If sensitive, can email *from a UW-associated email account*
 - Do NOT use Canvas messages

Lab 2 Notes

- ❖ Implementing a few **RAM variants** on the DE1-SoC
 - Using both a library catalog and user-specified RAM modules
- ❖ Learn how to create and use a **memory initialization file** (*.mif*) to initialize memory on your board
- ❖ Feel free to reuse other modules (*e.g.*, input, clock divider, 7-seg, counter) from 271/369
 - Simple modules don't need diagrams or simulations, but they should be shown in the block diagram and mentioned in your report

Synchronous Single-Port RAM (Review)

❖ Synchronous Inputs:

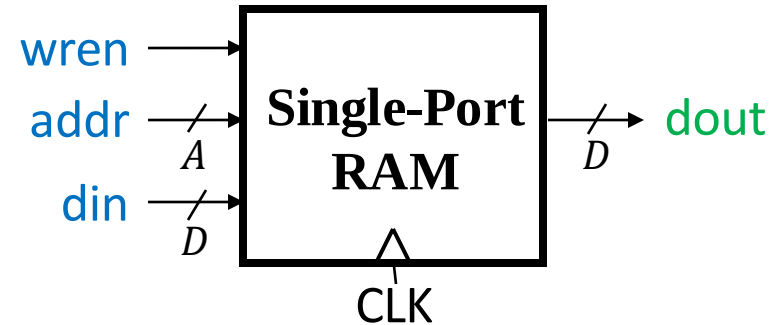
- wren (1 = write, 0 = read)
- addr (A -bit address)
- din (D -bit data)

❖ Synchronous Output:

- dout (D -bit data)

❖ Implementation hints:

- Will need an internal RAM array of what size?
- To synchronize, should update on clock triggers
- What should dout do when wren = 1?



Synchronous Single-Port RAM (Review)

```
module RAM_single #(parameter A, D)
    (clk, wren, addr, din, dout);

    input  logic clk, wren;
    input  logic [A-1:0] addr;
    input  logic [D-1:0] din;
    output logic [D-1:0] dout;

    logic [D-1:0] RAM [0:2**A-1];

    always_ff @(posedge clk) begin
        if (wren) begin
            RAM[addr] <= din;
            dout <= din;
        end
        else
            dout <= RAM[addr];
    end // always_ff
endmodule
```

Simplified Synchronous Dual-Port RAM

- ❖ 2 ports with 1 dedicated to writing and the other dedicated to reading

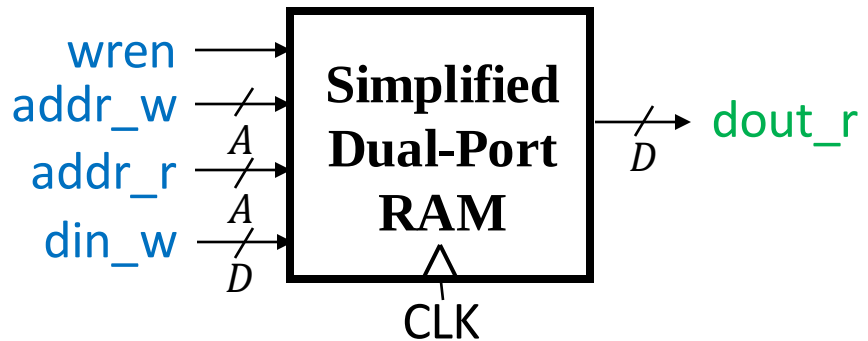
- ❖ Synchronous Inputs:

- `wren` (1 = write, 0 = read)
- `addr_w` (A -bit address)
- `addr_r` (A -bit address)
- `din_w` (D -bit data)

- ❖ Synchronous Output:

- `dout_r` (D -bit data)

- ❖ Differences in SystemVerilog?



Simplified Synchronous Dual-Port RAM (SV)

```
module RAM_dual_simple #(parameter A, D)
    (clk, wren, addr_w, addr_r, din_w, dout_r);

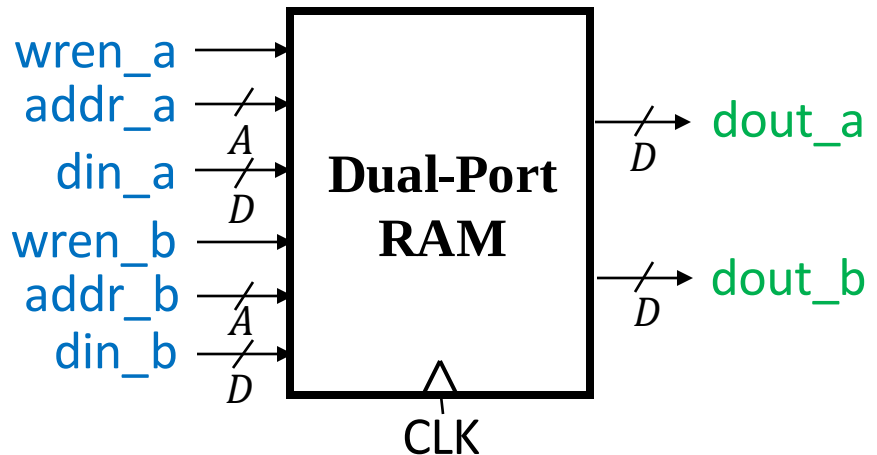
    input  logic clk, wren;
    input  logic [A-1:0] addr_w, addr_r;
    input  logic [D-1:0] din_w;
    output logic [D-1:0] dout_r;

    logic [D-1:0] RAM [0:2**A-1];

    always_ff @(posedge clk) begin
        if (wren) begin
            RAM[addr_w] <= din_w;
            dout_r <= (addr_r == addr_w) ? din_w : RAM[addr_r];
        end
        else
            dout_r <= RAM[addr_r];
        end // always_ff
    endmodule
```

Synchronous Dual-Port RAM

- ❖ The most general configuration – each port can either read or write
- ❖ Synchronous Inputs:
 - wren_a and wren_b
 - addr_a and addr_b
 - din_a and din_b
- ❖ Synchronous Output:
 - dout_a and dout_b
- ❖ Differences in SystemVerilog?



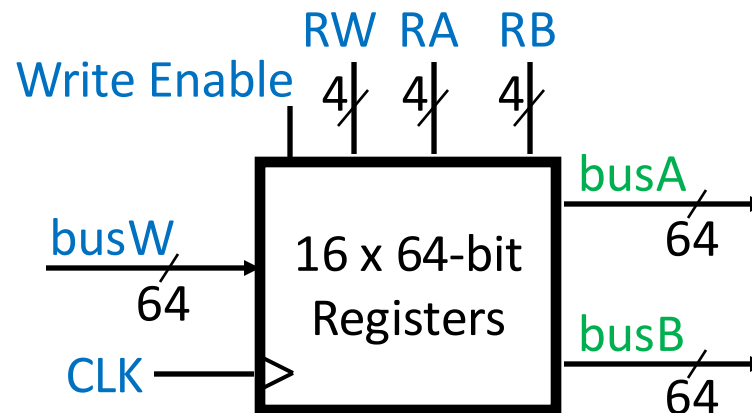
Memory

- ❖ Several forms of memory are available, which include:
 - ~~Secondary memory (e.g., hard disk, flash drive)~~
 - Read-only memory (ROM)
 - Random-access memory (RAM)
 - Register files
 - Small, fast, fixed-sized memory that hold CPU data state
 - First in, first out (FIFO) buffers

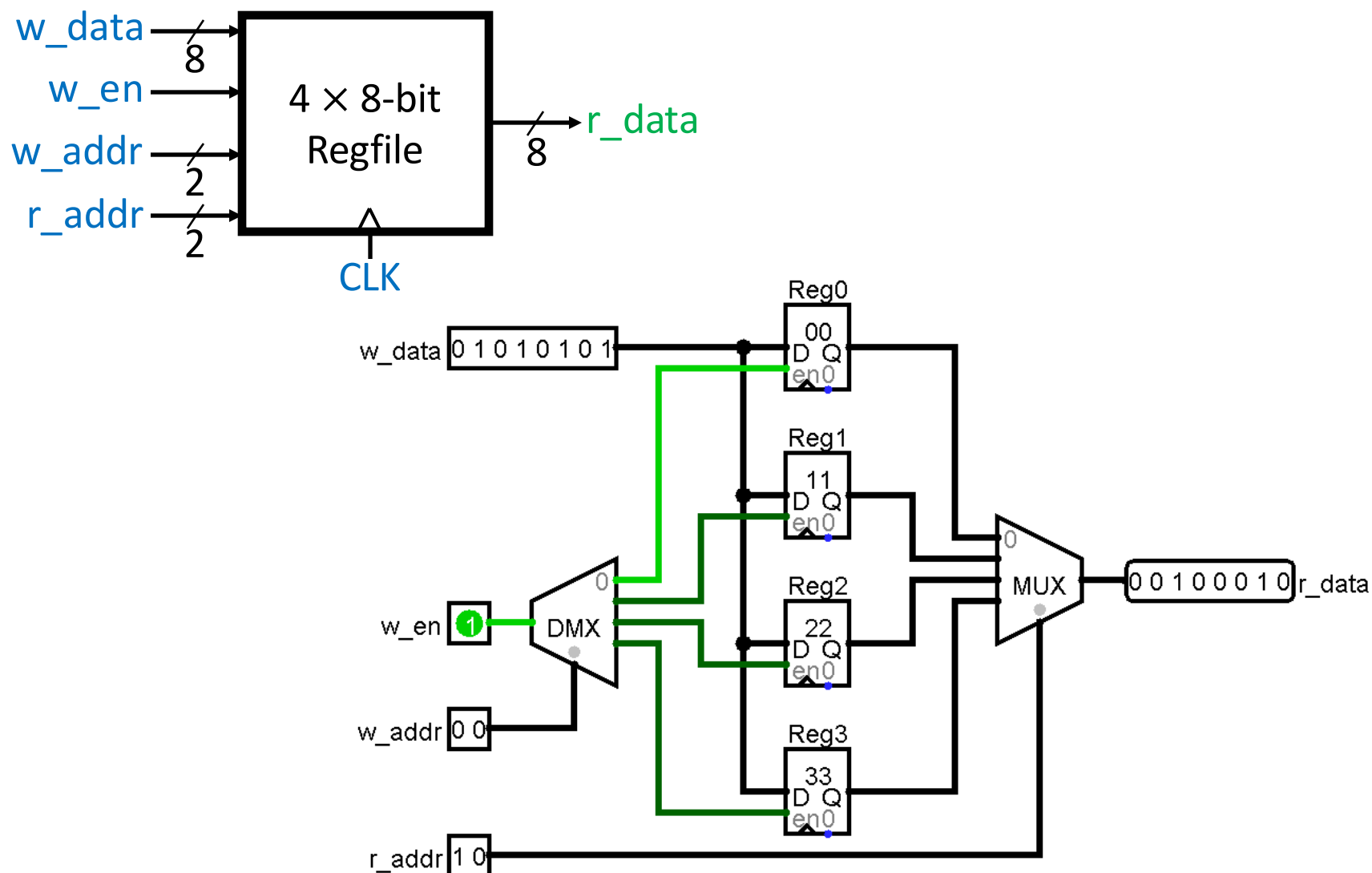
Memory Type #3: Register File

- ❖ Register File – a collection of registers
 - 1 input data port – can only write to 1 register at a time
 - 1+ output data ports – can read from 1+ register at a time
 - Address inputs to specify read/write targets
 - Write enable
- ❖ Frequently used in CPUs or as fast buffers

- ❖ Example:



Simple Register File (4 reg, 1 read port)



Memory Review

- ❖ Can think of reg file as a 2-D array of D flip-flops:
- ❖ The simple reg file was labeled 4×8
 - SystemVerilog array declaration:
- ❖ For a generic reg file with parameters D_WIDTH and A_WIDTH:
 - Depth:
 - Width:
 - SystemVerilog array declaration:

Register File with 1 Read Port (SV)

```
module reg_file #(parameter D_WIDTH=8, A_WIDTH=2)
    (clk, w_data, w_en, w_addr, r_addr, r_data);

    input  logic clk, w_en;
    input  logic [A_WIDTH-1:0] w_addr, r_addr;
    input  logic [D_WIDTH-1:0] w_data;
    output logic [D_WIDTH-1:0] r_data;

    // array declaration (registers)
    logic [D_WIDTH-1:0] array_reg [0:2**A_WIDTH-1];

    // write operation (synchronous)
    always_ff @(posedge clk)
        if (w_en)
            array_reg[w_addr] <= w_data;

    // read operation (asynchronous)
    assign r_data = array_reg[r_addr];

endmodule
```

Where's the Hardware?

```
module reg_file #(parameter D_WIDTH=8, A_WIDTH=2)
    (clk, w_data, w_en, w_addr, r_addr, r_data);

    input  logic clk, w_en;
    input  logic [A_WIDTH-1:0] w_addr, r_addr;
    input  logic [D_WIDTH-1:0] w_data;
    output logic [D_WIDTH-1:0] r_data;

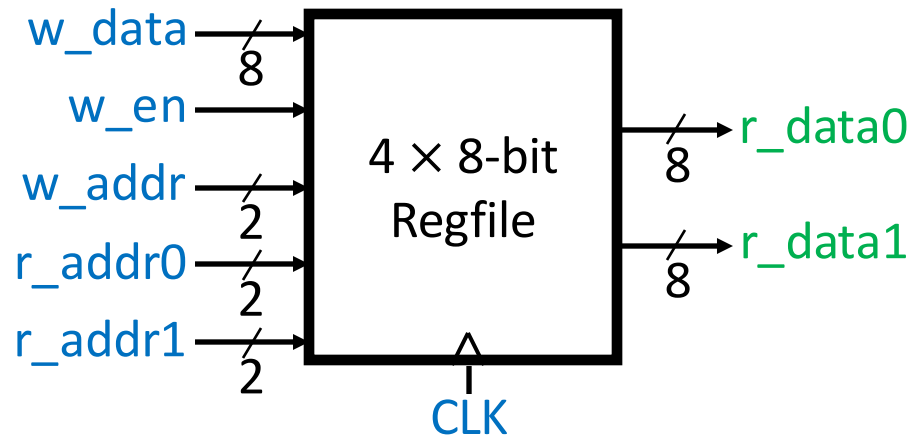
    // array declaration (registers)
    logic [D_WIDTH-1:0] array_reg [0:2**A_WIDTH-1];

    // write operation (synchronous)
    always_ff @(posedge clk)
        if (w_en)
            array_reg[w_addr] <= w_data;

    // read operation (asynchronous)
    assign r_data = array_reg[r_addr];

endmodule
```

Register File with 2 Read Ports



- ❖ What would change in hardware?
- ❖ What would change in SystemVerilog?

Register File with Synchronous Read

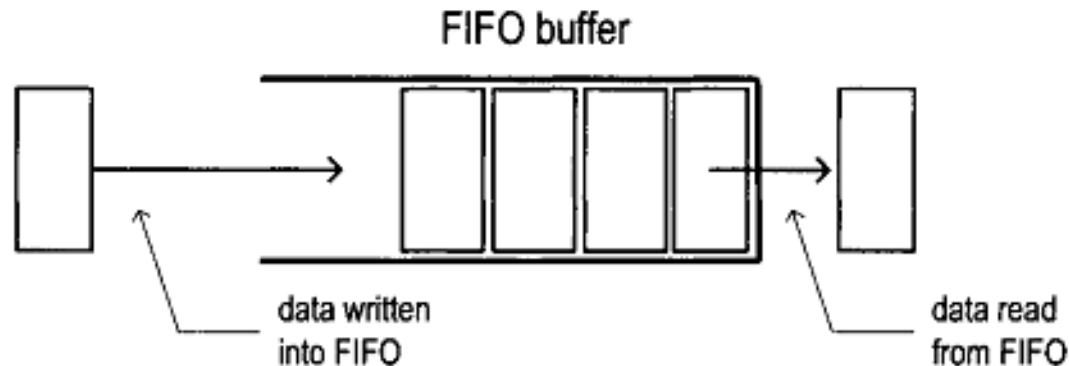
- ❖ Back to the 1 read port version, but now we want to make reading *synchronous*:
 - What would change in SystemVerilog?
 - What would change in hardware?

Short Tech Break

Memory Type #4: FIFO Buffer

❖ First-In First-Out (FIFO) Buffer

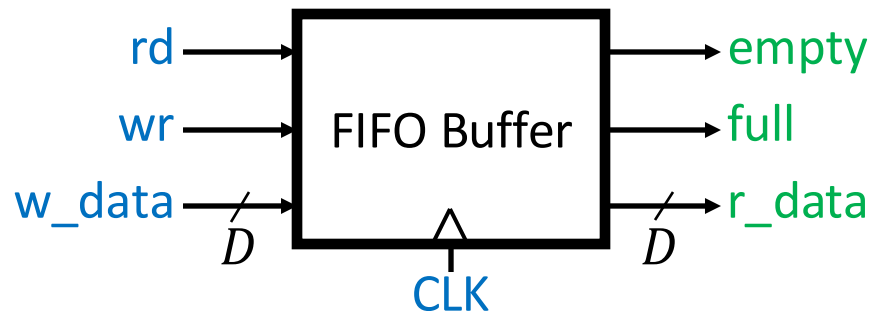
- Data storage such that elements that arrived earlier are accessed before elements that arrived later



- Has a limited capacity, so there is a notion of *fullness*
- Useful for synchronization, especially in communication (e.g., UART, disk, network)

FIFO Buffer Functionality

❖ Implementation we will work towards:



- **rd** signals to read the next element on **r_data**,
wr signals to write **w_data** into the buffer
- Outgoing data is read from the *front/head* of the buffer and incoming data is written to the *back/tail* of the buffer
- Can be implemented by wrapping a regular memory component with a special *controller*
 - However, the FIFO buffer has no visible notion of address!

FIFO Read Configurations

❖ First Word Fall Through (FWFT)

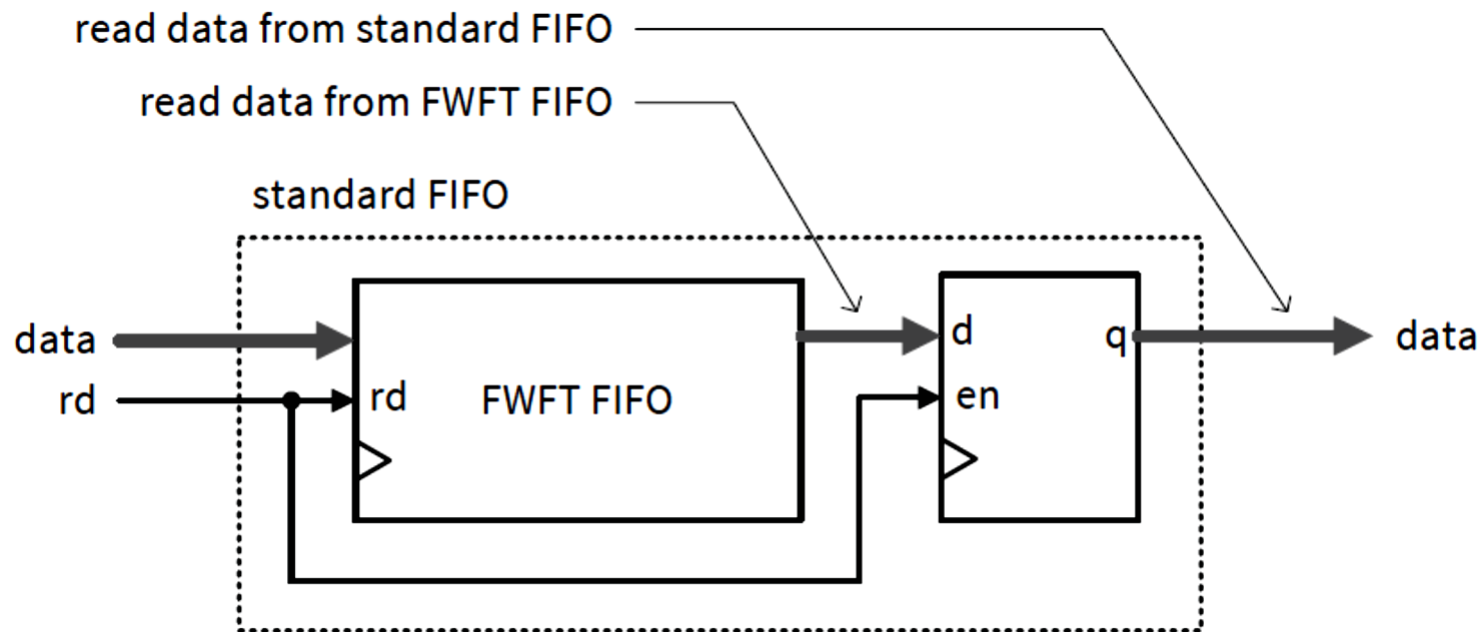
- *Asynchronous* read: front element of buffer always “falls through” and is immediately available on the output bus
 - Including when an element is written to an empty buffer!
- rd therefore acts more like a “remove” signal

❖ Standard

- *Synchronous* read: front element of buffer becomes available on next clock cycle after rd is asserted
- rd therefore acts more like a “request” signal

FIFO Read Configurations

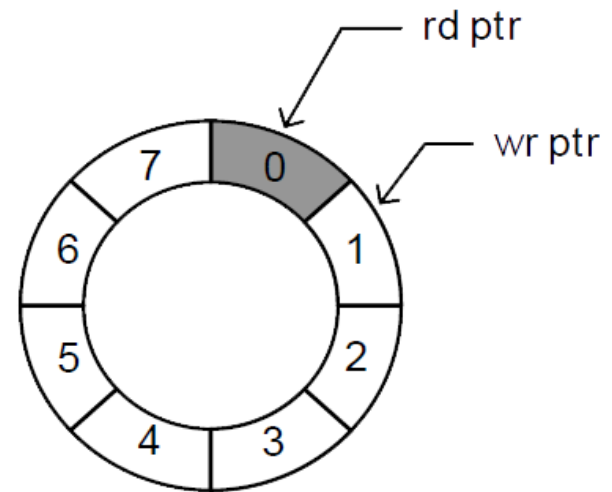
- ❖ Read configuration comparison
 - FWFT can be converted to standard by registering the output:



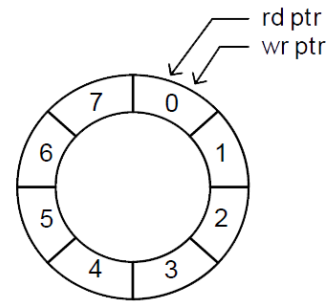
FIFO Implementation

❖ A FIFO buffer is often implemented as a *circular queue* with two pointers:

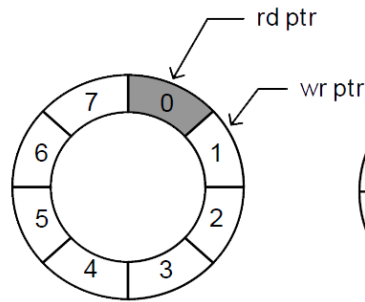
- **rd_ptr** indicates the location of the front/head (*i.e.*, the first valid data) and advances when rd is asserted
- **wr_ptr** indicates the location of the back/tail (*i.e.*, the first empty element) and advances when wr is asserted
- empty and full as buffer fullness status indicators
 - These are tricky because both situations have `rd_ptr == wr_ptr`



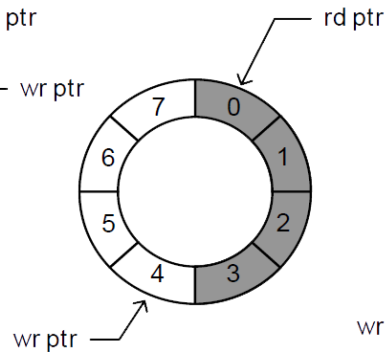
Circular Queue Example Operation



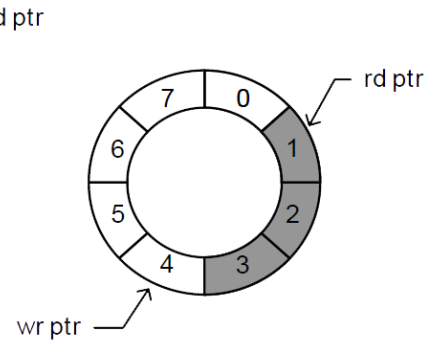
(a). initial (empty)



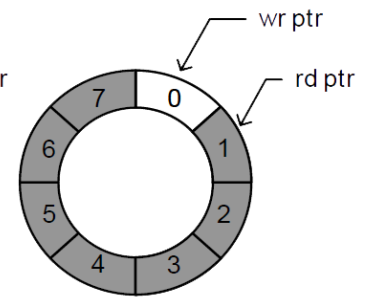
(b). after a write



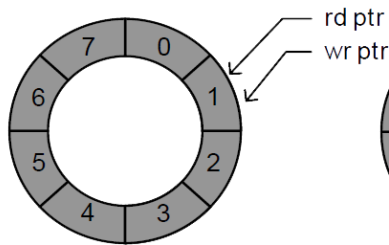
(c). 3 more writes



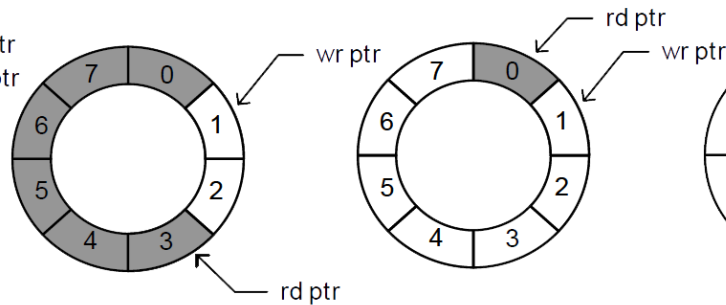
(d). after a read



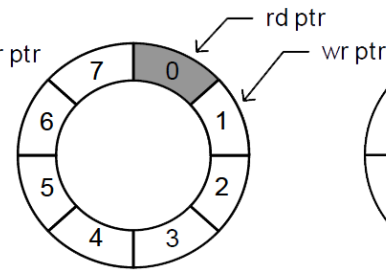
(e). 4 more writes



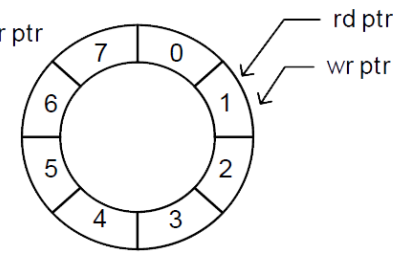
(f). 1 more write (full)



(g). 2 reads



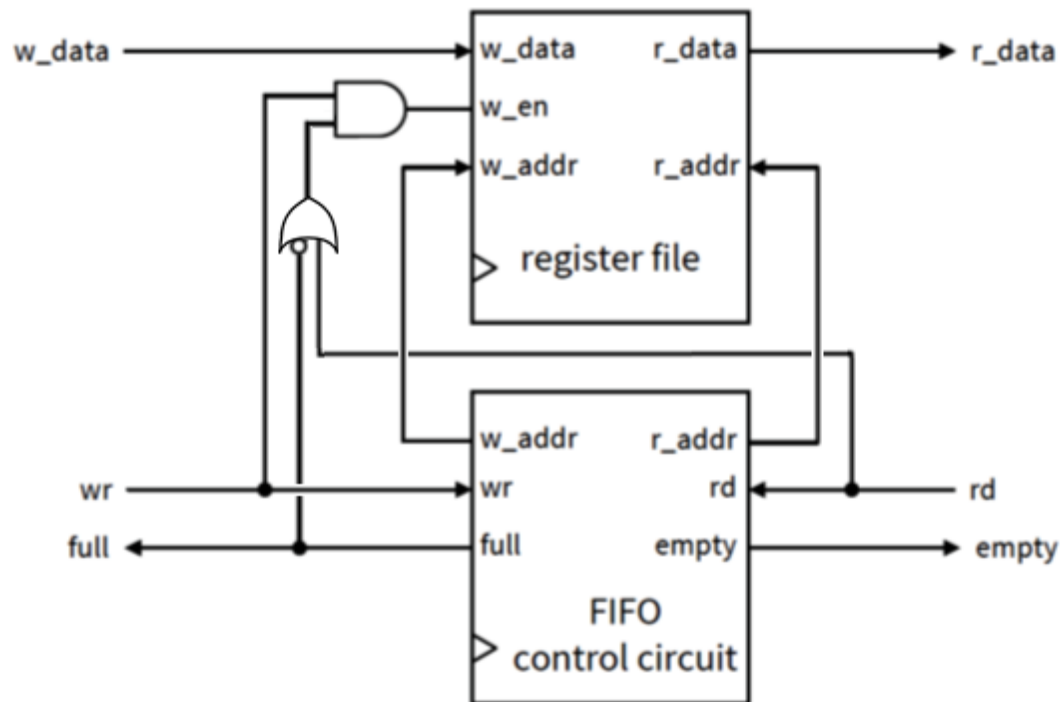
(h). 5 more reads



(i). 1 more read (empty)

Circular Queue Implementation

- ❖ A circular queue can be implemented using a RAM module and a *FIFO controller*
 - The controller handles the “arrangement” of the linear memory space into a circular queue



FIFO Controller

- ❖ FIFO controller internals:
 - rd_ptr and wr_ptr are *counters*
 - empty and full are *flip-flops*
 - Next state logic based on inputs rd and wr:

rd	wr	rd_ptr	wr_ptr	empty	full
0	0				
0	1				
1	0				
1	1				

FIFO Controller

❖ FIFO controller internals:

- rd_ptr and wr_ptr are *counters*
- empty and full are *flip-flops*
- Next state logic based on inputs rd and wr:
 - rw
 - 00 → no change
 - 11 → advance both rd_ptr and wr_ptr
full and empty don't change
 - 10 → if not empty: advance rd_ptr,
set full = 0,
set empty = 1 if rd_ptr == wr_ptr
 - 01 → if not full: advance wr_ptr,
set empty = 0,
set full = 1 if rd_ptr == wr_ptr

FIFO Controller (SV, 1/3)

```
module fifo_ctrl #(parameter A_WIDTH=4)
    (clk, reset, rd, wr, empty, full, w_addr, r_addr);

    input  logic clk, reset, rd, wr;
    output logic empty, full;
    output logic [A_WIDTH-1:0] w_addr, r_addr;

    // next state signal declarations
    logic [A_WIDTH-1:0] rd_ptr, rd_ptr_next;
    logic [A_WIDTH-1:0] wr_ptr, wr_ptr_next;
    logic empty_next, full_next;

    // output assignments
    assign w_addr = wr_ptr;
    assign r_addr = rd_ptr;

    // [continued on next slide...]
```

FIFO Controller (SV, 2/3)

```
// fifo controller logic
always_ff @(posedge clk) begin
    if (reset)
        begin
            wr_ptr <= 0;
            rd_ptr <= 0;
            full    <= 0;
            empty   <= 1;
        end
    else
        begin
            wr_ptr <= wr_ptr_next;
            rd_ptr <= rd_ptr_next;
            full    <= full_next;
            empty   <= empty_next;
        end
end

// [continued on next slide...]
```

FIFO Controller (SV, 3/3)

```
// next state logic
always_comb begin
    // default: keep current values
    rd_ptr_next = rd_ptr;
    wr_ptr_next = wr_ptr;
    empty_next = empty;
    full_next = full;

    // [continued in next box...]
```

```
case ({rd, wr})
2'b11: // read and write
    begin
        rd_ptr_next = rd_ptr + 1'b1;
        wr_ptr_next = wr_ptr + 1'b1;
    end
2'b10: // read
    if (~empty) begin
        rd_ptr_next = rd_ptr + 1'b1;
        if (rd_ptr_next == wr_ptr)
            empty_next = 1;
        full_next = 0;
    end
2'b01: // write
    if (~full) begin
        wr_ptr_next = wr_ptr + 1'b1;
        empty_next = 0;
        if (wr_ptr_next == rd_ptr)
            full_next = 1;
    end
2'b00: ; // no change
endcase
end // always_comb

endmodule
```

FIFO Buffer (SV)

```
module fifo #(parameter D_WIDTH=8, A_WIDTH=4)
    (clk, reset, rd, wr, empty, full, w_data, r_data);
```

```
    input logic clk, reset, rd, wr;
    output logic empty, full;
    input logic [D_WIDTH-1:0] w_data;
    output logic [D_WIDTH-1:0] r_data;
```

```
// signal declarations
```

```
    logic [A_WIDTH-1:0] w_addr, r_addr;
    logic w_en;
```

```
// enable write only when FIFO is not full
```

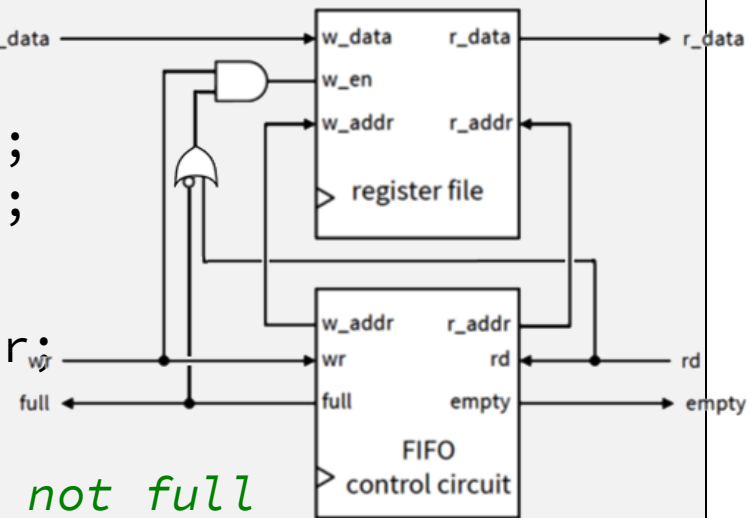
```
    assign w_en = wr & (~full | rd);
```

```
// instantiate FIFO controller and register file
```

```
    fifo_ctrl #(A_WIDTH) control (.*);
```

```
    reg_file #(D_WIDTH, A_WIDTH) mem (.*);
```

```
endmodule
```



Memory Controllers

- ❖ A *memory controller* is an interface circuit between user logic and the physical memory device
 - Abstracts away details of physical memory device while providing a consistent interface to the user
 - The FIFO controller we just discussed allows a user to interface with the register file we implemented on the FPGA's internal memory module
- ❖ Memory controllers are found with all kinds of memory
 - Your DE1-SoC contains memory controllers for SDRAM and DDR3 (and controllers for a bunch of other things like USB, VGA, PS/2, I2C)

DE1-SoC Memory Revisited

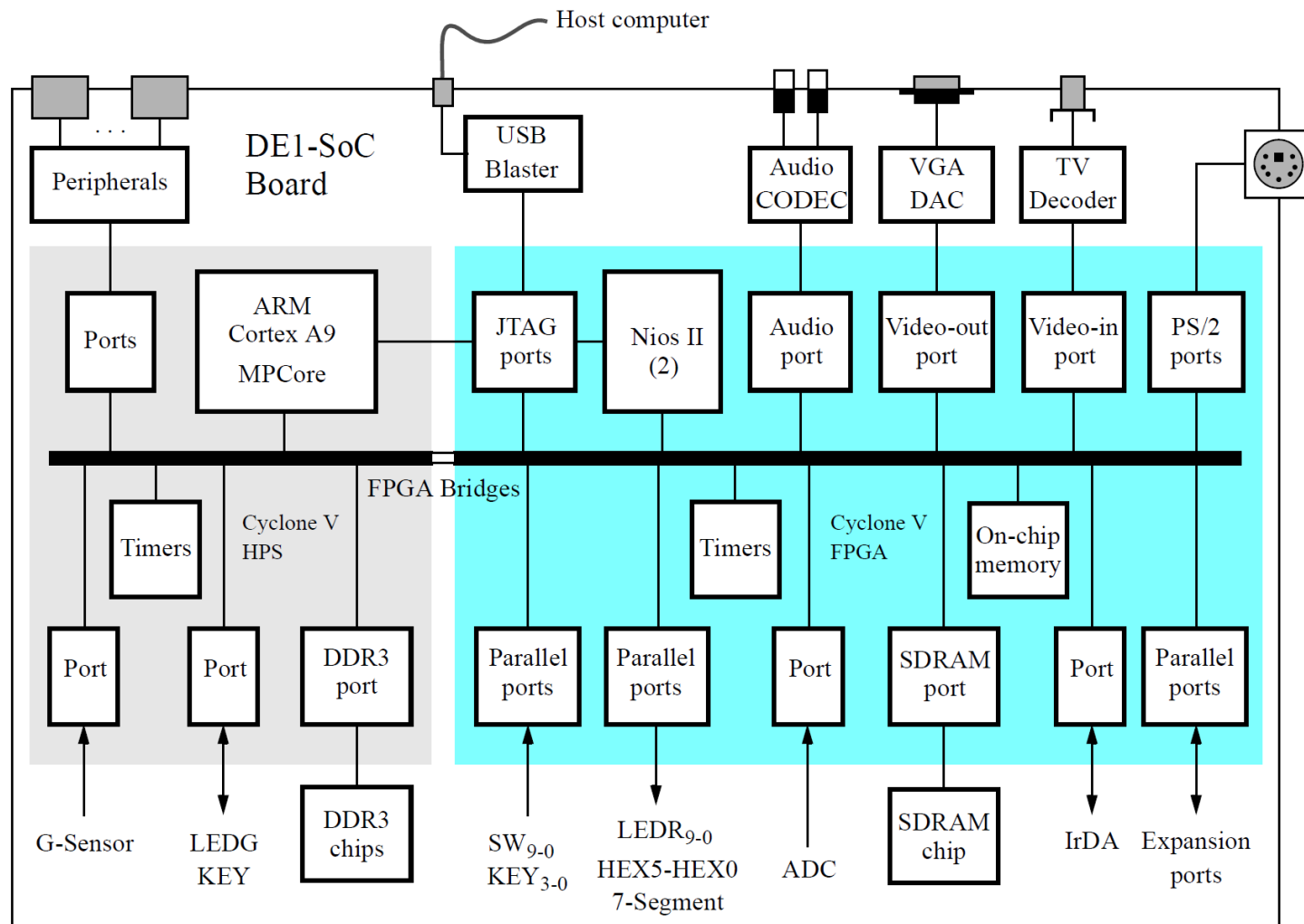
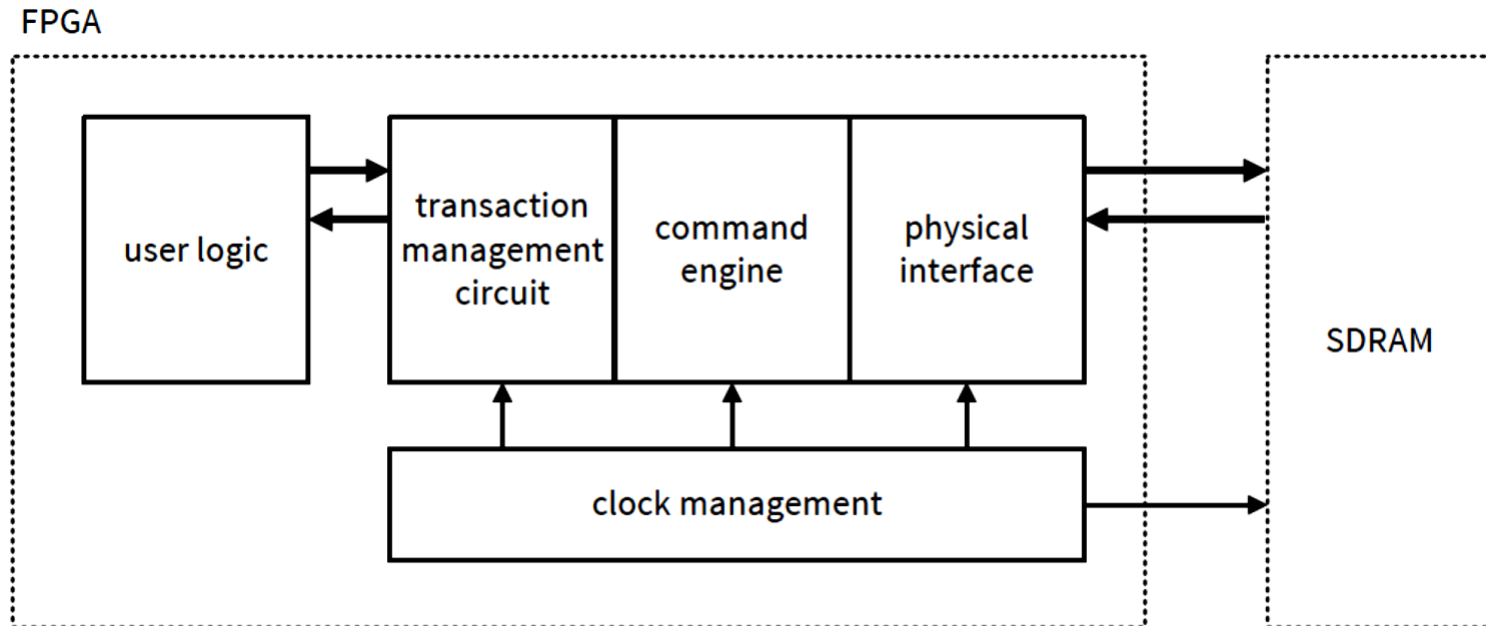


Figure 1. Block diagram of the DE1-SoC Computer.

SDRAM Controller



- ❖ High-performance controllers are very complex!
 - Design depends on individual FPGA and SDRAM devices
 - Usually constructed with vendor-supplied IP core