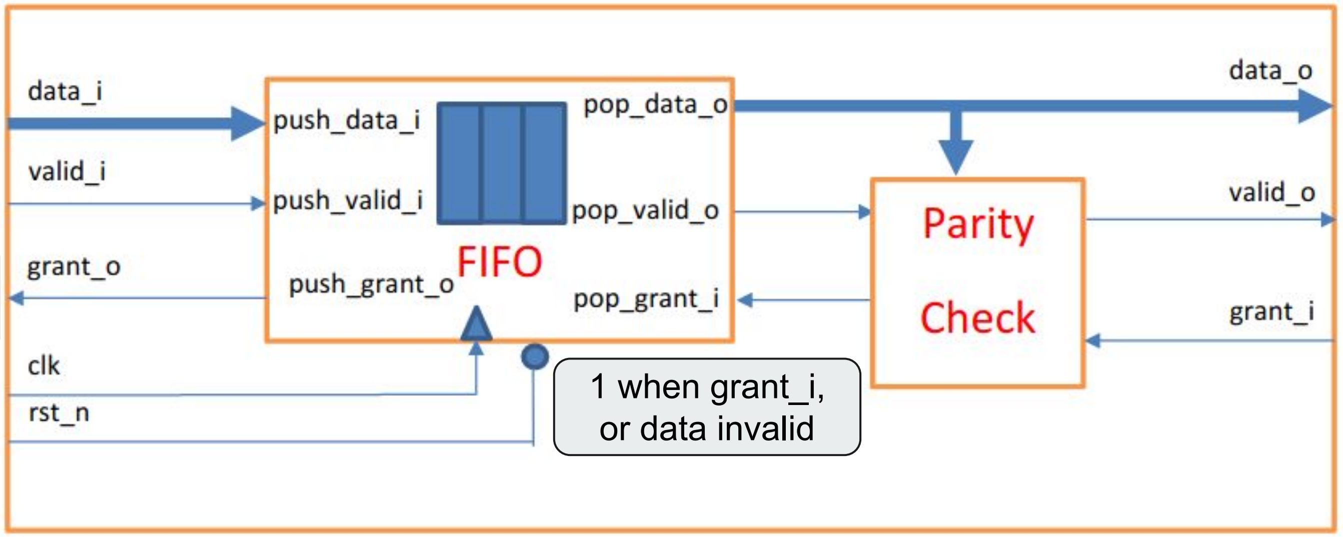
FIFO & GVSOC

1. HW
2. SW

https://github.com/CyrilKoe/random_name/

Presentation of the problem

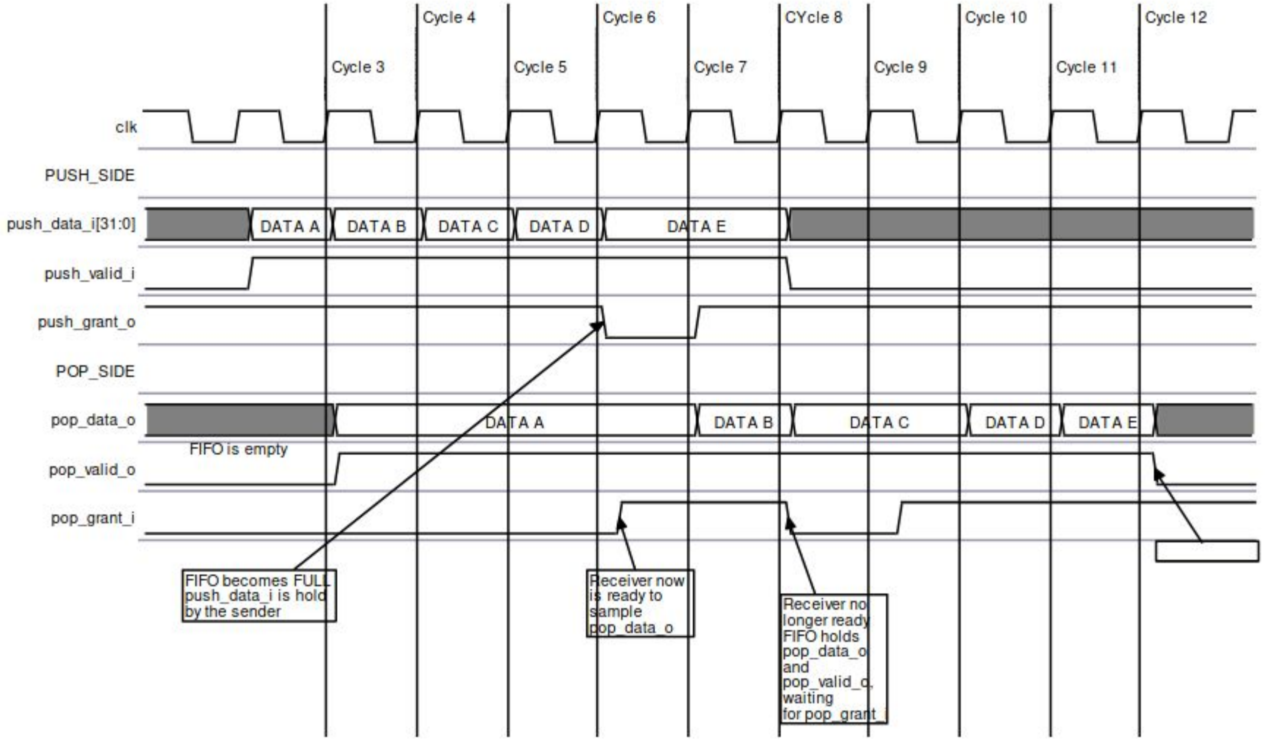


1 when fifo is not full

1 when grant_i, or data invalid

1 when output data and valid bits

Desired timings



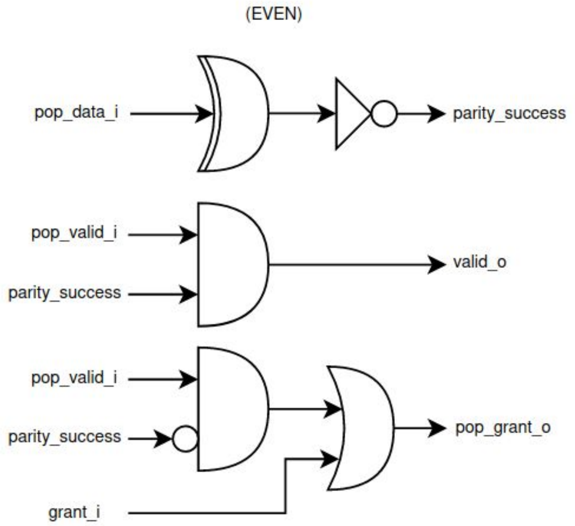
<u>Note :</u>
If pop_valid_o is available directly after write sampling, then read is not sampled
-> Synchronous write
-> Asynchronous read
This requires using of register files on an FPGA if it doesn't have asynchronous read RAM

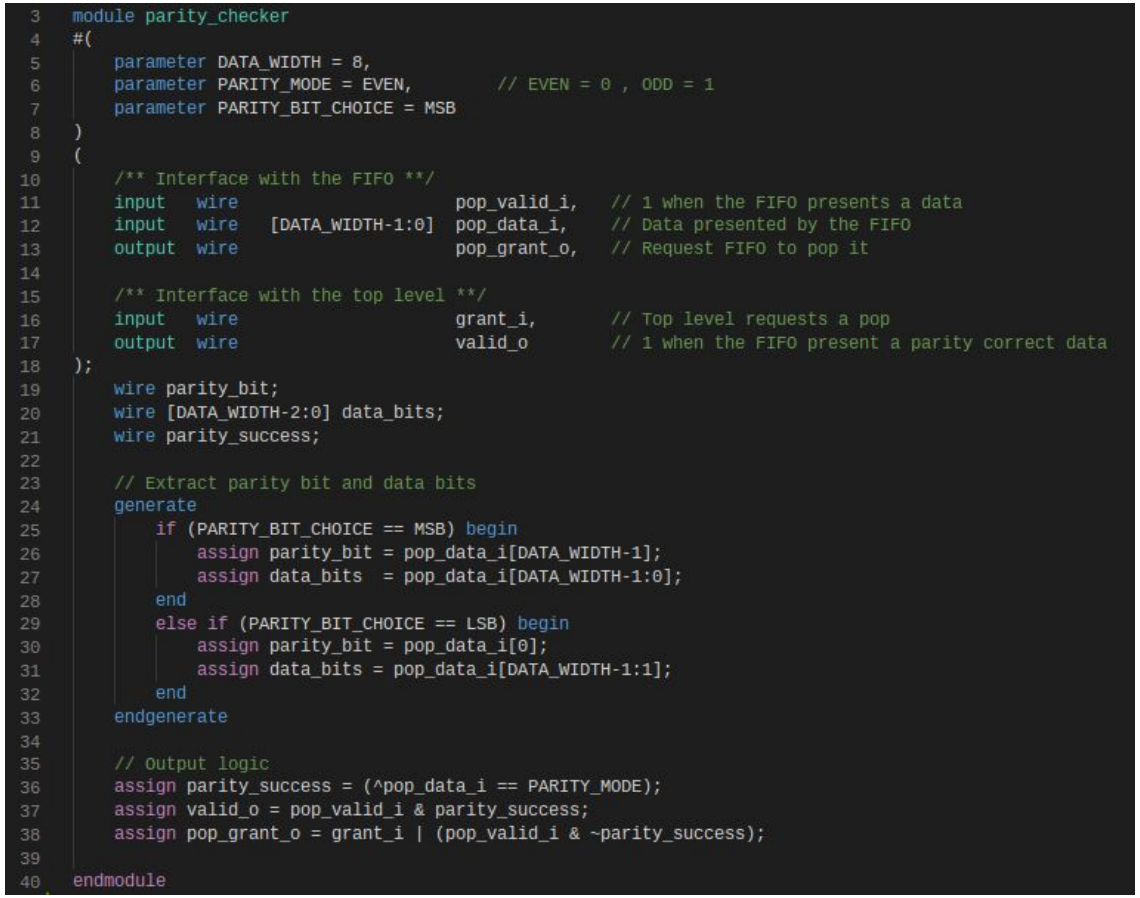
<u>Outputs :</u>
push_grant_o = 1 if FIFO not full
pop_valid_o = 1 if FIFO not empty and parity check is good

HW

Parity checker : logic & code

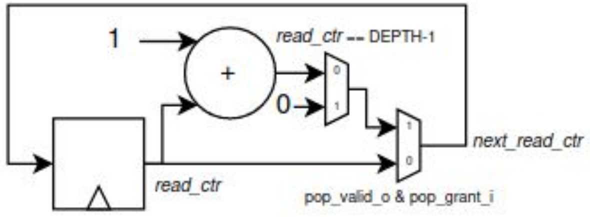
<u>Note :</u>
No need to know which bit is the parity bit.

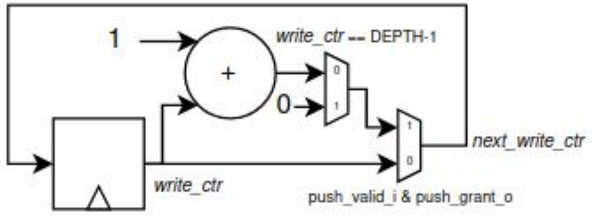


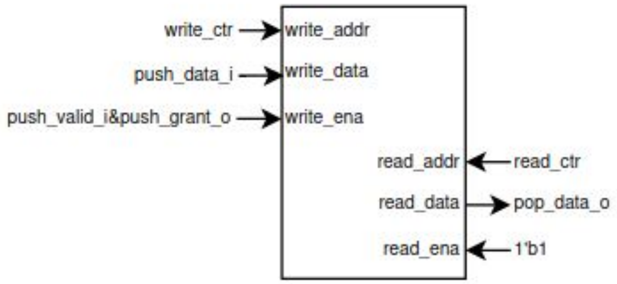

```verilog
module parity_checker
#(
    parameter DATA_WIDTH = 8,
    parameter PARITY_MODE = EVEN,        // EVEN = 0 , ODD = 1
    parameter PARITY_BIT_CHOICE = MSB
)
(
    /** Interface with the FIFO **/
    input   wire                     pop_valid_i,   // 1 when the FIFO presents a data
    input   wire   [DATA_WIDTH-1:0]  pop_data_i,    // Data presented by the FIFO
    output  wire                     pop_grant_o,   // Request FIFO to pop it

    /** Interface with the top level **/
    input   wire                     grant_i,       // Top level requests a pop
    output  wire                     valid_o        // 1 when the FIFO present a parity correct data
);
    wire parity_bit;
    wire [DATA_WIDTH-2:0] data_bits;
    wire parity_success;

    // Extract parity bit and data bits
    generate
        if (PARITY_BIT_CHOICE == MSB) begin
            assign parity_bit = pop_data_i[DATA_WIDTH-1];
            assign data_bits  = pop_data_i[DATA_WIDTH-1:0];
        end
        else if (PARITY_BIT_CHOICE == LSB) begin
            assign parity_bit = pop_data_i[0];
            assign data_bits = pop_data_i[DATA_WIDTH-1:1];
        end
    endgenerate

    // Output logic
    assign parity_success = (^pop_data_i == PARITY_MODE);
    assign valid_o = pop_valid_i & parity_success;
    assign pop_grant_o = grant_i | (pop_valid_i & ~parity_success);

endmodule
```

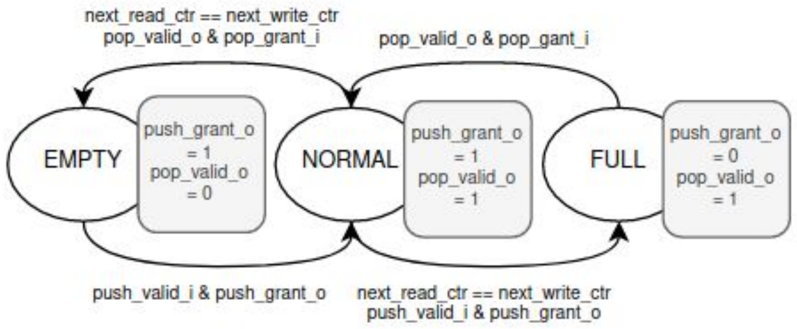
HW

FIFO : FSM and logic





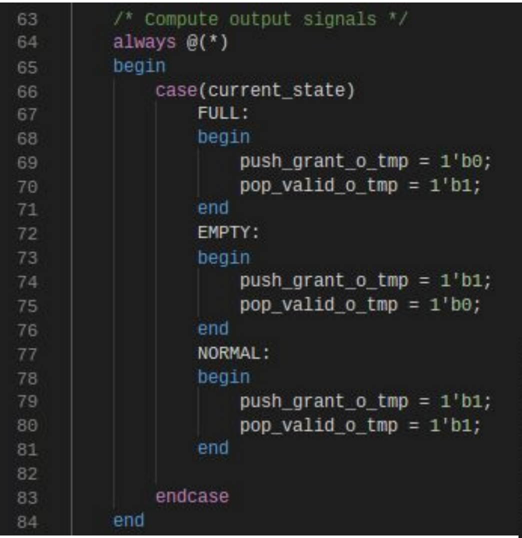



next_read_ctr == next_write_ctr
pop_valid_o & pop_grant_i
pop_valid_o & pop_gant_i
EMPTY
push_grant_o = 1
pop_valid_o = 0
NORMAL
push_grant_o = 1
pop_valid_o = 1
FULL
push_grant_o = 0
pop_valid_o = 1
push_valid_i & push_grant_o
next_read_ctr == next_write_ctr
push_valid_i & push_grant_o

HW

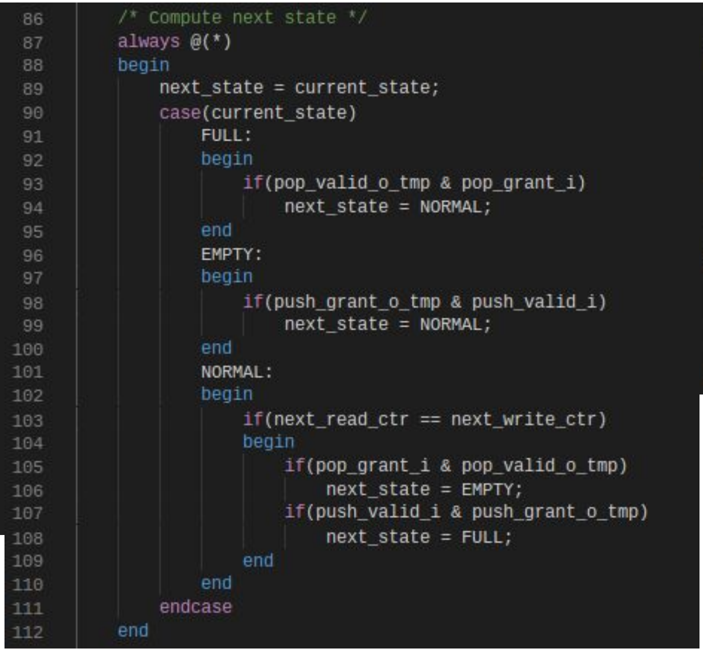
FIFO : Code

FSM outputs



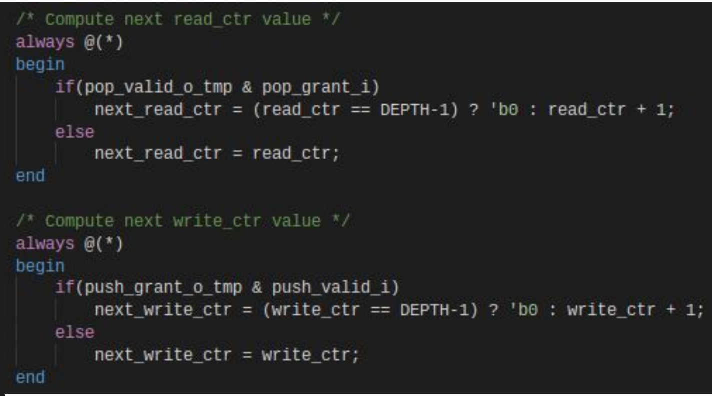
```
/* Compute output signals */
always @(*)
begin
    case(current_state)
        FULL:
        begin
            push_grant_o_tmp = 1'b0;
            pop_valid_o_tmp = 1'b1;
        end
        EMPTY:
        begin
            push_grant_o_tmp = 1'b1;
            pop_valid_o_tmp = 1'b0;
        end
        NORMAL:
        begin
            push_grant_o_tmp = 1'b1;
            pop_valid_o_tmp = 1'b1;
        end

    endcase
end
```

FSM states



```
/* Compute next state */
always @(*)
begin
    next_state = current_state;
    case(current_state)
        FULL:
        begin
            if(pop_valid_o_tmp & pop_grant_i)
                next_state = NORMAL;
        end
        EMPTY:
        begin
            if(push_grant_o_tmp & push_valid_i)
                next_state = NORMAL;
        end
        NORMAL:
        begin
            if(next_read_ctr == next_write_ctr)
            begin
                if(pop_grant_i & pop_valid_o_tmp)
                    next_state = EMPTY;
                if(push_valid_i & push_grant_o_tmp)
                    next_state = FULL;
            end
        end
    endcase
end
```

Counters computation



```
/* Compute next read_ctr value */
always @(*)
begin
    if(pop_valid_o_tmp & pop_grant_i)
        next_read_ctr = (read_ctr == DEPTH-1) ? 'b0 : read_ctr + 1;
    else
        next_read_ctr = read_ctr;
end

/* Compute next write_ctr value */
always @(*)
begin
    if(push_grant_o_tmp & push_valid_i)
        next_write_ctr = (write_ctr == DEPTH-1) ? 'b0 : write_ctr + 1;
    else
        next_write_ctr = write_ctr;
end
```

Verification : Python model

Idea :

Use a python model of the FIFO to output expected test input vectors and outputs.

→ System Verilog testbench can easily be used to verify input and output data

→ Checking expected timing behavior requires more advances models (ex SystemC).

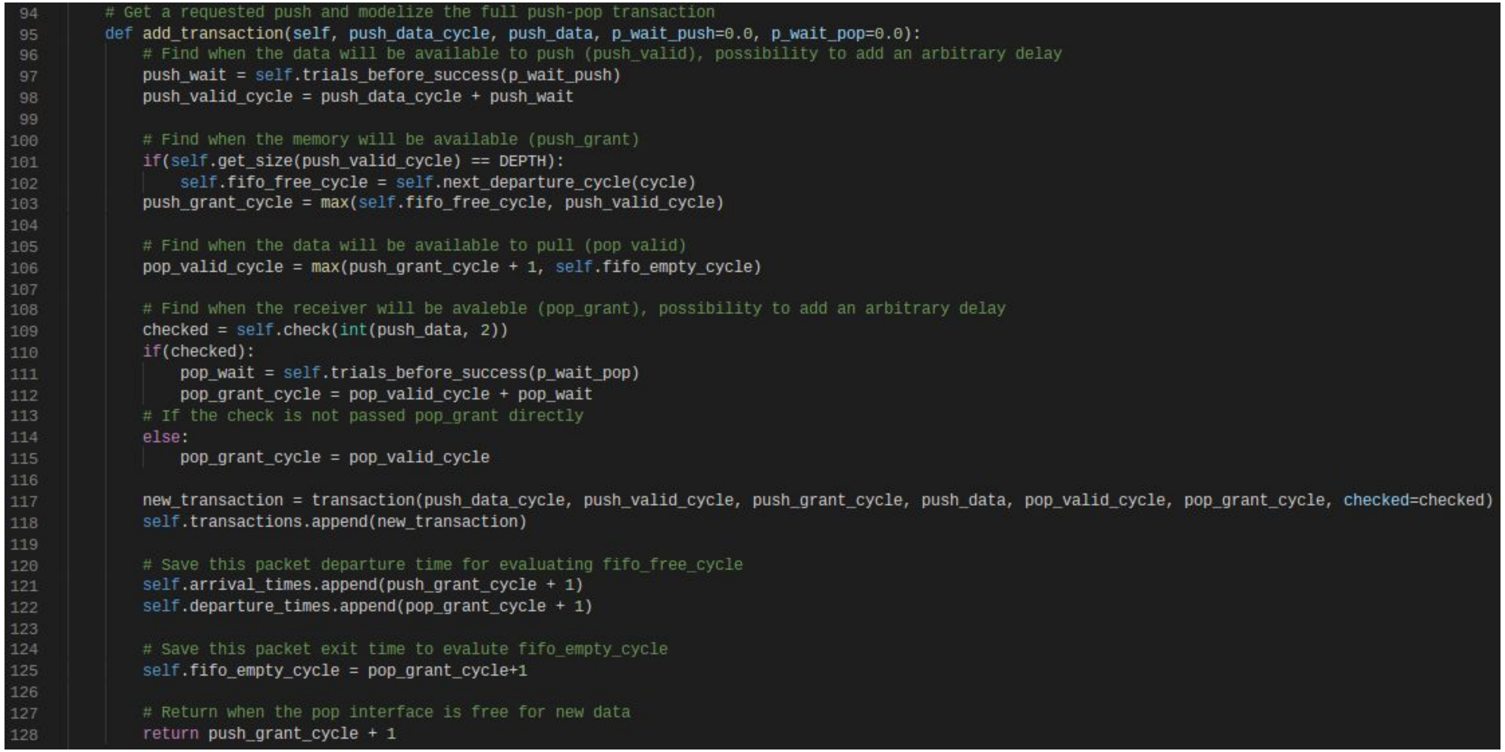
→ Here a Python model write the input events and expected output events in a file and SV testbench asserts its

Program :

For each cycle :

- Generate a data (invalid with p_error_bit)
- Ask to push the data at this cycle
- If FIFO full wait until next departure
- When a random amount of time (p_wait_push)
- Push and update the size of the FIFO
- Go to the cycle when the FIFO will be empty
- Wait a random amount of time (p_wait_pull)
- Grant the pop out, update the next time FIFO will be empty, update the size

Verification : Python model code



```python
# Get a requested push and modelize the full push-pop transaction
def add_transaction(self, push_data_cycle, push_data, p_wait_push=0.0, p_wait_pop=0.0):
    # Find when the data will be available to push (push_valid), possibility to add an arbitrary delay
    push_wait = self.trials_before_success(p_wait_push)
    push_valid_cycle = push_data_cycle + push_wait

    # Find when the memory will be available (push_grant)
    if(self.get_size(push_valid_cycle) == DEPTH):
        self.fifo_free_cycle = self.next_departure_cycle(cycle)
    push_grant_cycle = max(self.fifo_free_cycle, push_valid_cycle)

    # Find when the data will be available to pull (pop valid)
    pop_valid_cycle = max(push_grant_cycle + 1, self.fifo_empty_cycle)

    # Find when the receiver will be avaleble (pop_grant), possibility to add an arbitrary delay
    checked = self.check(int(push_data, 2))
    if(checked):
        pop_wait = self.trials_before_success(p_wait_pop)
        pop_grant_cycle = pop_valid_cycle + pop_wait
    # If the check is not passed pop_grant directly
    else:
        pop_grant_cycle = pop_valid_cycle

    new_transaction = transaction(push_data_cycle, push_valid_cycle, push_grant_cycle, push_data, pop_valid_cycle, pop_grant_cycle, checked=checked)
    self.transactions.append(new_transaction)

    # Save this packet departure time for evaluating fifo_free_cycle
    self.arrival_times.append(push_grant_cycle + 1)
    self.departure_times.append(pop_grant_cycle + 1)

    # Save this packet exit time to evalute fifo_empty_cycle
    self.fifo_empty_cycle = pop_grant_cycle+1

    # Return when the pop interface is free for new data
    return push_grant_cycle + 1
```

HW

Verification : Output

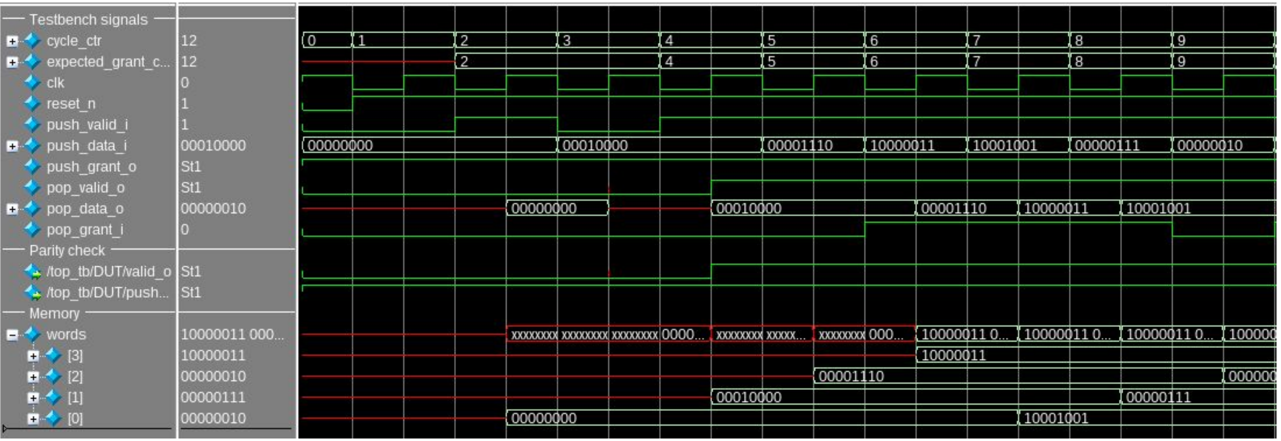
Example test vector

```
# DATA_WIDTH=8 DEPTH=4 PARITY_MODE=ODD PARITY_BIT_CHOICE=MSB
# push_data_cycle : push_valid_cycle : push_grant_cycle : data : pop_valid_cycle : pop_grant_cycle
2 : 2 : 2 : 00000000 : -1 : -1
3 : 4 : 4 : 00010000 : 5 : 6
5 : 5 : 5 : 00001110 : 7 : 7
6 : 6 : 6 : 10000011 : 8 : 8
7 : 7 : 7 : 10001001 : 9 : 10
8 : 8 : 8 : 00000111 : 11 : 11
9 : 9 : 9 : 00000010 : 12 : 13
```

```
make compile_with_args PARITY_MODE=ODD TRANSACTION_FOLDER=test_03
```

Example test log

```
[2] (00000000) arrived at the push interface
[2] (00000000) ready to be pushed
[2] (00000000) will be pushed
[2] (00001110) End of cycle [2] size is 1

[3] (00000000) ready to be popped
[3] (00000000) will be popped
[3] (00010000) arrived at the push interface
[3] (00001110) End of cycle [3] size is 0

[4] (00010000) ready to be pushed
[4] (00010000) will be pushed
[4] (00001110) End of cycle [4] size is 1

[5] (00010000) ready to be popped
[5] (00001110) arrived at the push interface
[5] (00001110) ready to be pushed
[5] (00001110) will be pushed
[5] (00001110) End of cycle [5] size is 2
```



Example simulation result

Presentation of the problem