Design of Asynchronous FIFO

Presented by: Akash Pandey
2302102035
M.TECH. V.D.N.
Dept. of Electrical Engineering

Content

Asynchronous FIFO

- Asynchronous FIFOs are widely used to safely pass the data from one clock domain to another clock domain.

- FIFOs are used in designs to safely pass multi-bit data words from one clock domain to another.

- Data words are placed into a FIFO buffer memory array by control signals in one clock domain, and the data words are retrieved from another port of the same FIFO buffer memory array by control signals from a second clock domain.

- The difficulty associated with doing FIFO design is related to generating the FIFO pointers and finding a reliable way to determine full and empty status on the FIFO.

Block Diagram of Asynchronous FIFO

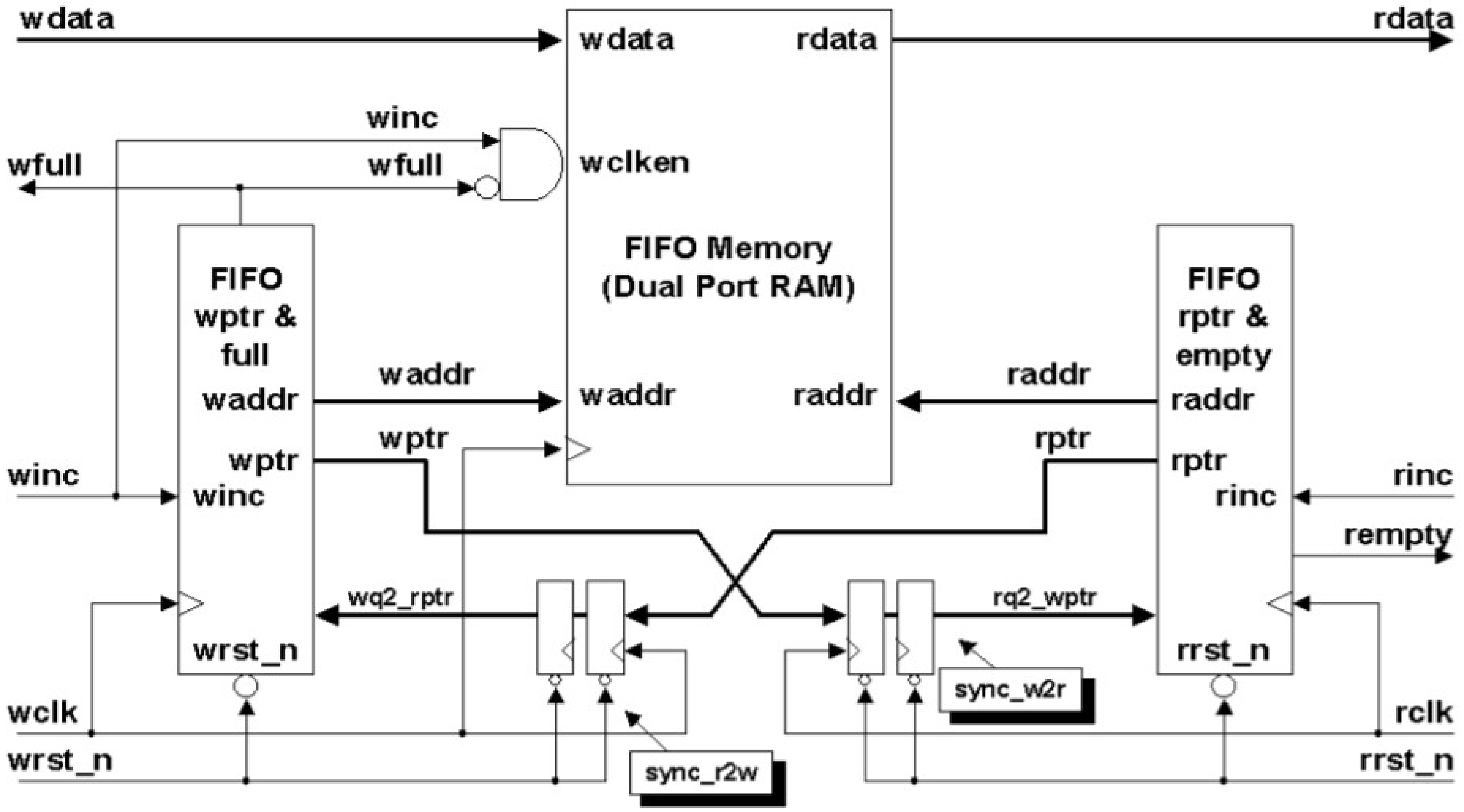


Figure 1: Block diagram of Asynchronous FIFO

Components of Asynchronous FIFO

- FIFO memory
- Binary & Gray Counter
- Synchronizer
- Full & Empty logic block

FIFO Memory

- **Data Storage:**
  - The FIFO memory module is the core component responsible for storing data temporarily. It's essentially a block of memory cells organized as a queue, where data is written to one end and read from the other.
- **Memory Depth:**
  - The depth of the FIFO memory determines its capacity to store data. It's crucial to choose an appropriate depth based on the expected data rates and burst lengths to avoid overflow and underflow conditions.

FIFO Memory

- **Memory Width:**
  - The width of the memory cells determines the size of data that can be stored in each location. It's typically matched to the data width of the system being interfaced.
- **Clock Domain Crossing (CDC):**
  - In an asynchronous FIFO, the memory module operates in both the write and read clock domains. Careful CDC techniques are necessary to ensure data integrity and prevent metastability issues when transferring data between the two domains

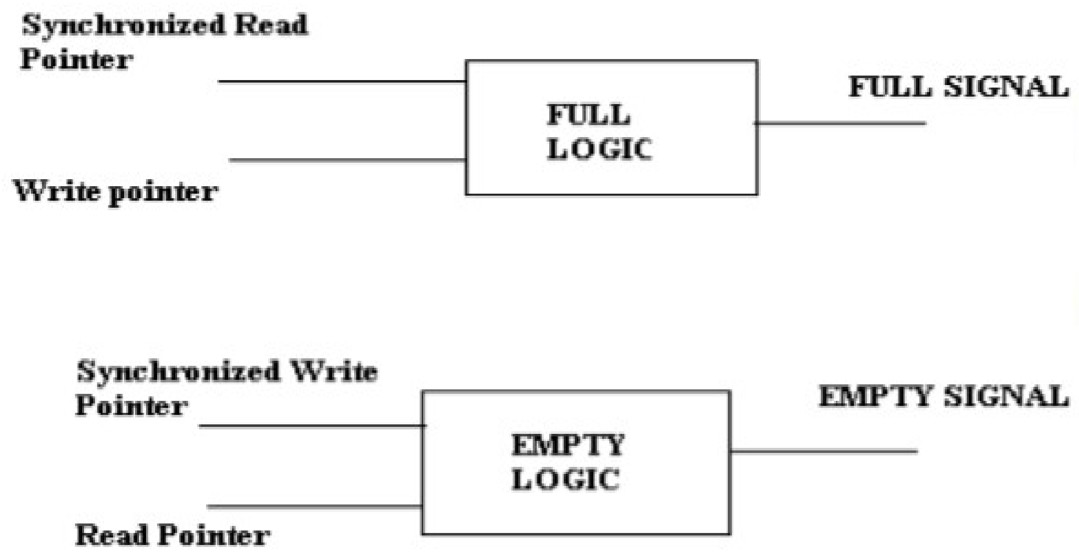
Binary and Gray Counter

- **Binary Pointers:**
  - Simple to implement.
  - Prone to metastability issues during comparison, especially when the pointers are close.
  - Requires additional logic to handle potential metastability.
- **Gray Code Pointers:**
  - More robust against metastability.
  - Only one bit changes between consecutive states, reducing the risk of multiple bits changing simultaneously.
  - Requires more complex logic to generate and compare Gray code values

2 Stage Synchronizer

- **Metastability Reduction:**
  - A 2-stage FF synchronizer is a common technique to reduce the risk of metastability when signals cross clock domain boundaries. By using two flip-flops in series, the probability of a metastable state propagating to the output is significantly reduced.
- **Synchronization Depth:**
  - While a 2-stage synchronizer is often sufficient, in critical applications or when dealing with high-frequency clock domains, a 3-stage or even 4-stage synchronizer may be necessary to further reduce the risk of metastability. However, increasing the number of stages can also introduce additional latency.

Full and Empty Logic

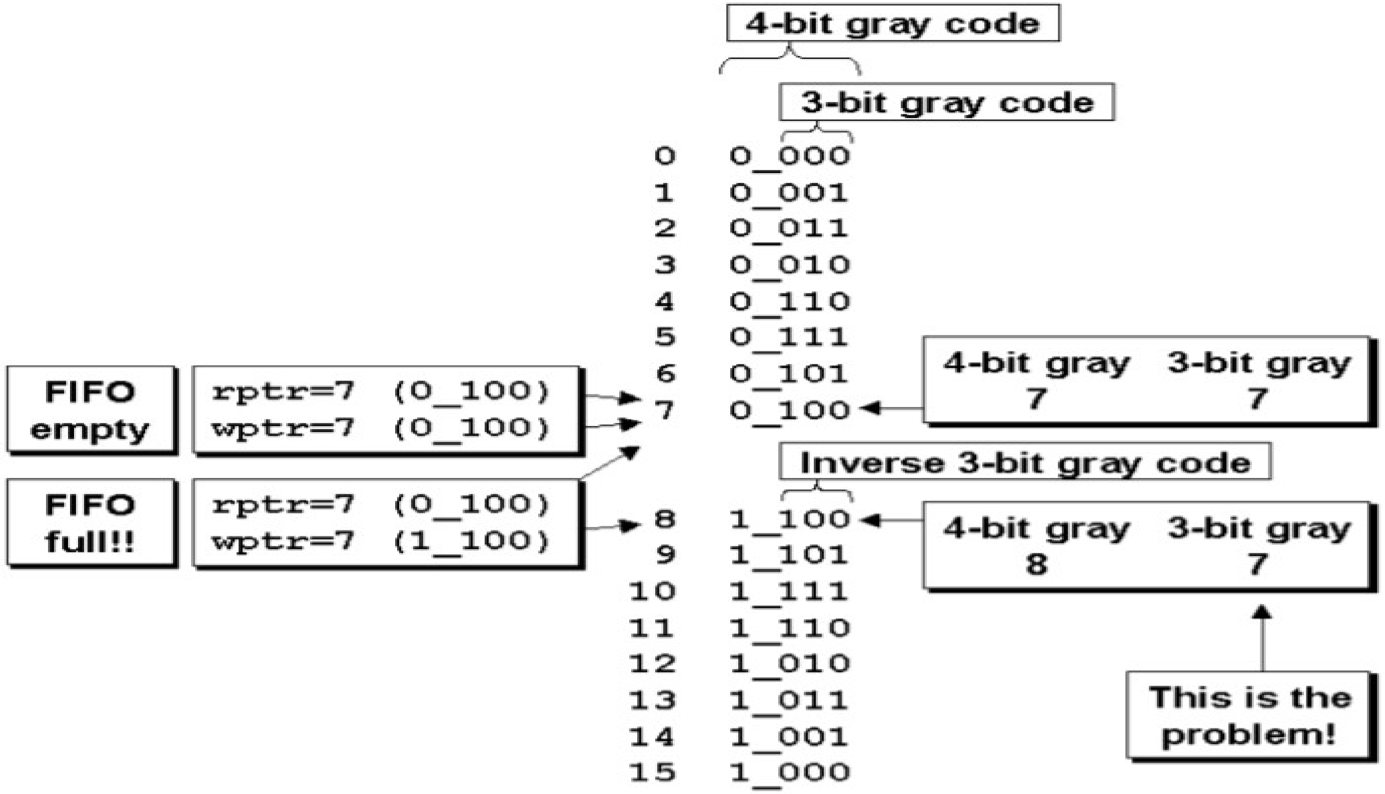


If ((synchronized Write pointer = = Read pointer) &&
(Synchronized Write pointer [3:0] = = Read pointer [3:0] then
Empty=1;
If (Write pointer= = {~ synchronized Read pointer [4:3], synchronized Read pointer [2:0]) then
Full=1;

Asynchronous FIFO pointers

- **Role:**
  - Pointers are essential for tracking the read and write positions within the FIFO memory. They determine where data is written and read.
- **Types:**
  - Binary Pointers: Simple to implement but susceptible to metastability.
  - Gray Code Pointers: More robust against metastability due to single-bit changes between consecutive states.
- **Comparison:**
  - Pointer comparison is a critical operation in FIFO design. It involves comparing the read and write pointers to determine the FIFO's fullness and emptiness status. Careful synchronization techniques are necessary to avoid metastability during pointer comparison

Full and Empty Condition



Full and Empty Condition

- If we use Gray code counters, then checking empty condition has no issue.
- But checking full condition is not as simple as we did in binary counters.
- If we take the same example of memory depth 8, w/r_add width will be 3 bits, w/r_ptr will be 4 bits.
- From the above figure , when write and read pointers point to 7th location it is empty condition with its Gray code value as 0100.
- Write and read address value will be 100.
- When we do next write, write pointer value will be 1100 and read pointer value will be 0100.
- If we use the same concept used for binary counters, then it is a full condition but here it is actually not.
- Also write address value not changing.

Verilog Code

```
module fifo1 #(parameter DSIZE = 8,
 parameter ASIZE = 4)
 (output [DSIZE-1:0] rdata,
 output wfull,
 output rempty,
 input [DSIZE-1:0] wdata,
 input winc, wclk, wrst_n,
 input rinc, rclk, rrst_n);
 wire [ASIZE-1:0] waddr, raddr;
 wire [ASIZE:0] wptr, rptr, wq2_rptr, rq2_wptr;


 sync_r2w sync_r2w (.wq2_rptr(wq2_rptr), .rptr(rptr),
 .wclk(wclk), .wrst_n(wrst_n));


 sync_w2r sync_w2r (.rq2_wptr(rq2_wptr), .wptr(wptr),
 .rclk(rclk), .rrst_n(rrst_n));
```

Verilog Code- Top Level Module

```
 fifomem #(DSIZE, ASIZE) fifomem
 (.rdata(rdata), .wdata(wdata),
 .waddr(waddr), .raddr(raddr),
 .wclken(winc), .wfull(wfull),
 .wclk(wclk));


 rptr_empty #(ASIZE) rptr_empty
 (.rempty(rempty),
 .raddr(raddr),
 .rptr(rptr), .rq2_wptr(rq2_wptr),
 .rinc(rinc), .rclk(rclk),
 .rrst_n(rrst_n));


 wptr_full #(ASIZE) wptr_full
 (.wfull(wfull), .waddr(waddr),
 .wptr(wptr), .wq2_rptr(wq2_rptr),
 .winc(winc), .wclk(wclk),
 .wrst_n(wrst_n));

endmodule
```

Verilog Code- Synchronizer

```
 module sync_r2w #(parameter ADDRSIZE = 4)
 (output reg [ADDRSIZE:0] wq2_rptr,
 input [ADDRSIZE:0] rptr,
 input wclk, wrst_n);

 reg [ADDRSIZE:0] wq1_rptr;

 always @(posedge wclk or negedge wrst_n)
 if (!wrst_n) {wq2_rptr,wq1_rptr} <= 0;
 else {wq2_rptr,wq1_rptr} <= {wq1_rptr,rptr};
endmodule

module sync_w2r #(parameter ADDRSIZE = 4)
 (output reg [ADDRSIZE:0] rq2_wptr,
 input [ADDRSIZE:0] wptr,
 input rclk, rrst_n);
 reg [ADDRSIZE:0] rq1_wptr;

 always @(posedge rclk or negedge rrst_n)
 if (!rrst_n) {rq2_wptr,rq1_wptr} <= 0;
 else {rq2_wptr,rq1_wptr} <= {rq1_wptr,wptr};
endmodule
```

Verilog Code- FIFO Memory

```
 module fifomem #(parameter DATASIZE = 8, // Memory data word width
 parameter ADDRSIZE = 4) // Number of mem address bits
 (output [DATASIZE-1:0] rdata,
 input [DATASIZE-1:0] wdata,
 input [ADDRSIZE-1:0] waddr, raddr,
 input wclken, wfull, wclk);

localparam DEPTH = 1<<ADDRSIZE;
 reg [DATASIZE-1:0] mem [0:DEPTH-1];
 assign rdata = mem[raddr];
 always @(posedge wclk)
 if (wclken && !wfull) mem[waddr] <= wdata;
endmodule
```

Verilog Code- Read pointer

```
 module rptr_empty #(parameter ADDRSIZE = 4)
 (output reg rempty,
 output [ADDRSIZE-1:0] raddr,
 output reg [ADDRSIZE :0] rptr,
 input [ADDRSIZE :0] rq2_wptr,
 input rinc, rclk, rrst_n);
 reg [ADDRSIZE:0] rbin;
 wire [ADDRSIZE:0] rgraynext, rbinnext;

always @(posedge rclk or negedge rrst_n)
 if (!rrst_n) {rbin, rptr} <= 0;
 else {rbin, rptr} <= {rbinnext, rgraynext};
 // Memory read-address pointer
assign raddr = rbin[ADDRSIZE-1:0];
 assign rbinnext = rbin + (rinc & ~rempty);
 assign rgraynext = (rbinnext>>1) ^ rbinnext;
assign rempty_val = (rgraynext == rq2_wptr);

 always @(posedge rclk or negedge rrst_n)
 if (!rrst_n) rempty <= 1'b1;
 else rempty <= rempty_val;
endmodule
```

Verilog Code- Write Pointer

```
 module wptr_full #(parameter ADDRSIZE = 4)
 (output reg wfull,
 output [ADDRSIZE-1:0] waddr,
 output reg [ADDRSIZE :0] wptr,
 input [ADDRSIZE :0] wq2_rptr,
 input winc, wclk, wrst_n);
 reg [ADDRSIZE:0] wbin;
 wire [ADDRSIZE:0] wgraynext, wbinnext;

always @(posedge wclk or negedge wrst_n)
 if (!wrst_n) {wbin, wptr} <= 0;
 else {wbin, wptr} <= {wbinnext, wgraynext};
 // Memory write-address pointer
assign waddr = wbin[ADDRSIZE-1:0];
 assign wbinnext = wbin + (winc & ~wfull);
 assign wgraynext = (wbinnext>>1) ^ wbinnext;
assign wfull_val = (wgraynext=={~wq2_rptr[ADDRSIZE:ADDRSIZE-1],
 wq2_rptr[ADDRSIZE-2:0]});

 always @(posedge wclk or negedge wrst_n)
 if (!wrst_n) wfull <= 1'b0;
 else wfull <= wfull_val;
endmodule
```

Verilog Code- Testbench

```
module fifo1_tb;

  // Parameters
  parameter DSIZE = 8;
  parameter ASIZE = 4;

  // Testbench Signals
  reg [DSIZE-1:0] wdata;
  reg winc, wclk, wrst_n;
  reg rinc, rclk, rrst_n;
  wire [DSIZE-1:0] rdata;
  wire wfull, rempty;

  // Instantiate the FIFO module
  fifo1 #(DSIZE, ASIZE) uut (
    .rdata(rdata),
    .wfull(wfull),
    .rempty(rempty),
    .wdata(wdata),
    .winc(winc),
    .wclk(wclk),
    .wrst_n(wrst_n),
    .rinc(rinc),
    .rclk(rclk),
    .rrst_n(rrst_n)
  );
```

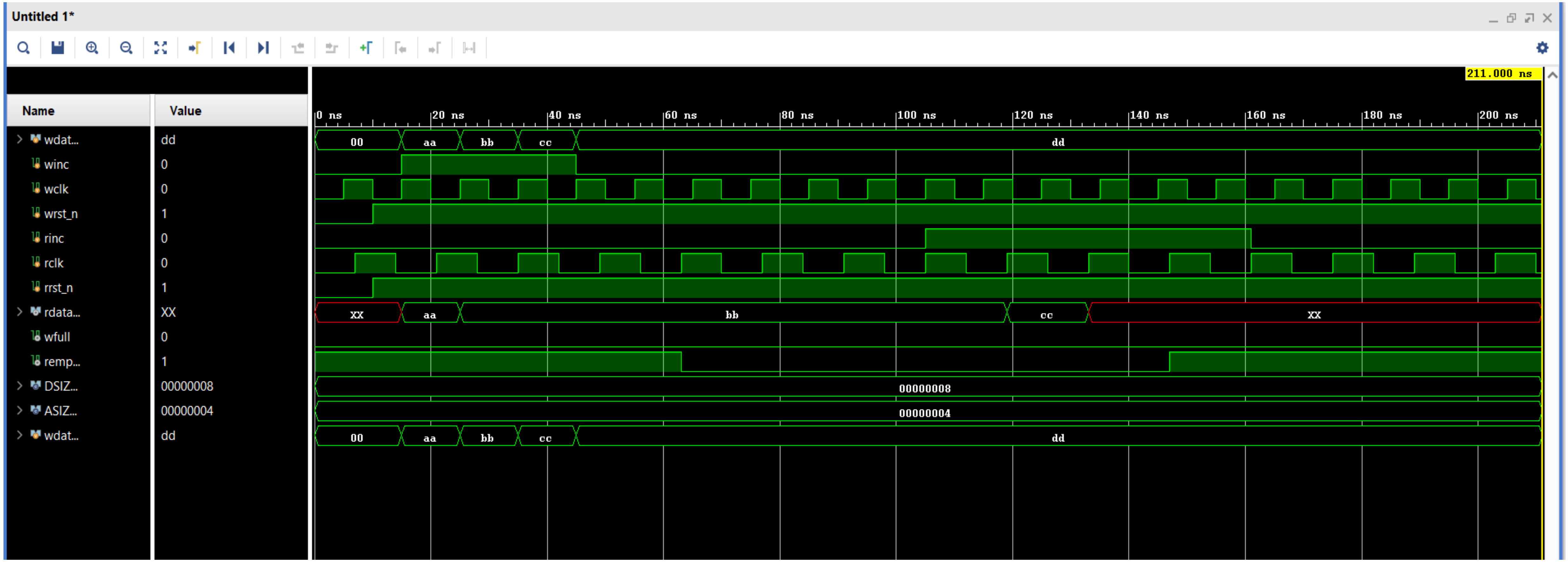
Verilog Code- Testbench

```
// Clock Generation
initial begin
  wclk = 0;
  forever #5 wclk = ~wclk;  // Write clock with 10ns period
end

initial begin
  rclk = 0;
  forever #7 rclk = ~rclk;  // Read clock with 14ns period
end

// Testbench Logic
initial begin
  // Initialize signals
  wdata = 0;
  winc = 0;
  rinc = 0;
  wrst_n = 0;
  rrst_n = 0;

  // Reset both write and read logic
  #10;
  wrst_n = 1;
  rrst_n = 1;
```

Verilog Code- Testbench

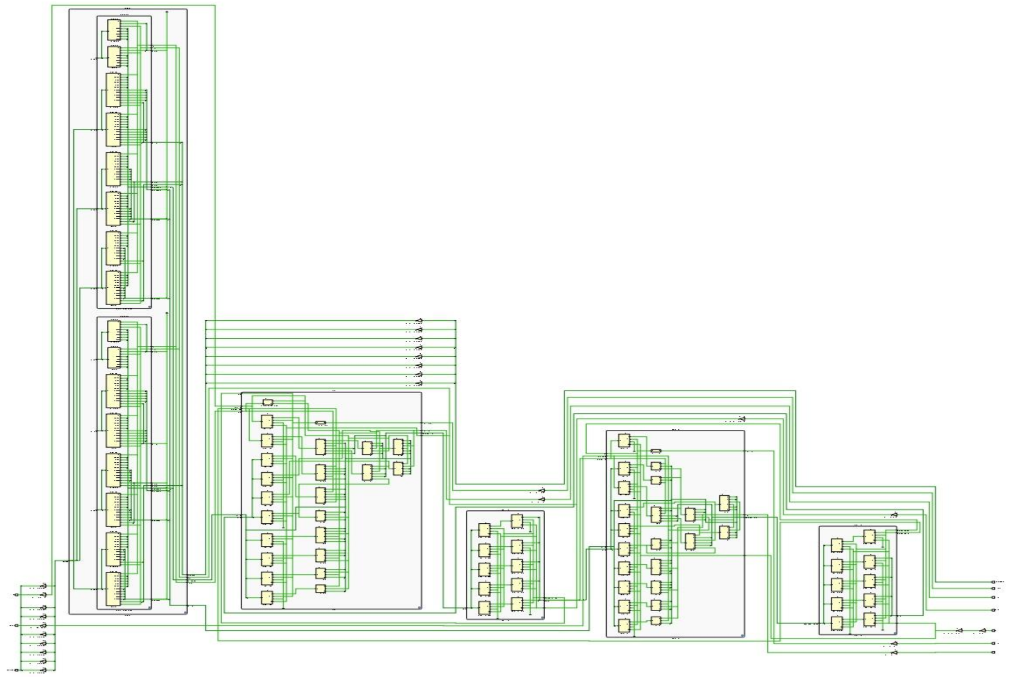
```
      // Test: Write data to FIFO
    @(posedge wclk);
    winc = 1;
    wdata = 8'hAA;
    @(posedge wclk);
    wdata = 8'hBB;
    @(posedge wclk);
    wdata = 8'hCC;
    @(posedge wclk);
    wdata = 8'hDD;
    winc = 0;

    // Wait for some cycles
    #50;

    // Test: Read data from FIFO
    @(posedge rclk);
    rinc = 1;
    @(posedge rclk);
    @(posedge rclk);
    @(posedge rclk);
    @(posedge rclk);
    rinc = 0;
#50;
    $finish;
  end
endmodule
```

Simulation

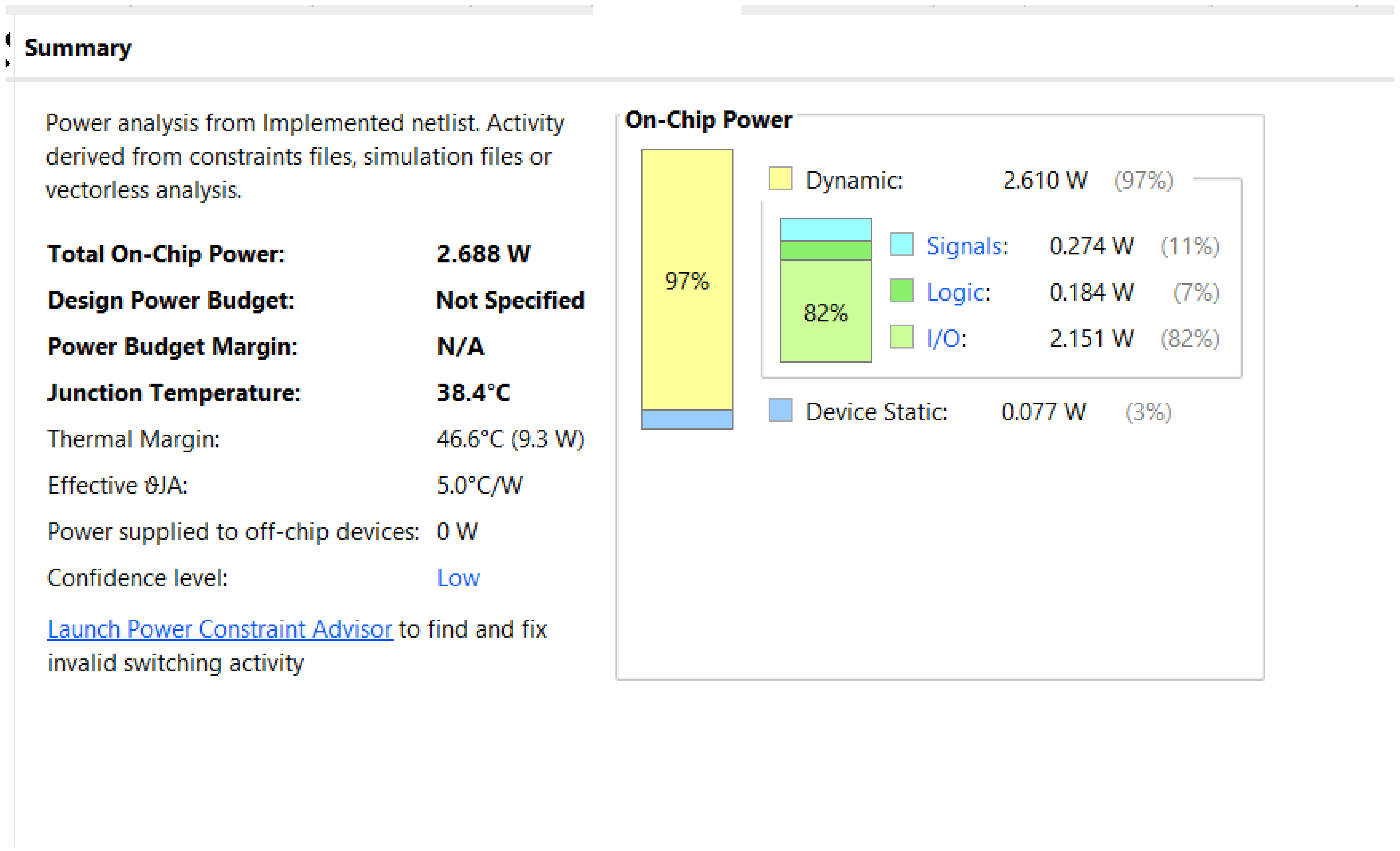


Name	Value
wdat...	dd
winc	0
wclk	0
wrst_n	1
rinc	0
rclk	0
rrst_n	1
rdata...	XX
wfull	0
remp...	1
DSIZ...	00000008
ASIZ...	00000004
wdat...	dd

Schematic



Power Report



Summary

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

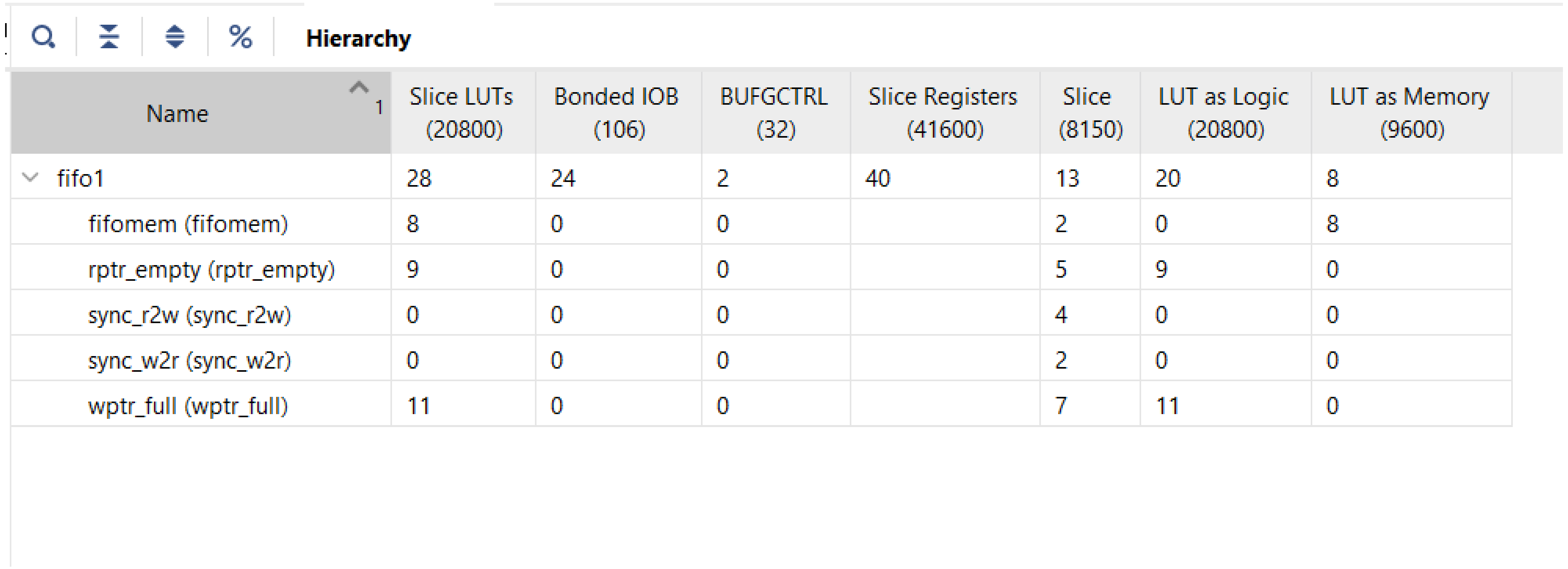
Total On-Chip Power:	**2.688 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**38.4°C**
Thermal Margin:	46.6°C (9.3 W)
Effective ϑJA:	5.0°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

🟨	Dynamic:	2.610 W	(97%)
🟦	Signals:	0.274 W	(11%)
🟩	Logic:	0.184 W	(7%)
🟩	I/O:	2.151 W	(82%)
🟦	Device Static:	0.077 W	(3%)

Utilization Report



Hierarchy

Name	Slice LUTs (20800)	Bonded IOB (106)	BUFGCTRL (32)	Slice Registers (41600)	Slice (8150)	LUT as Logic (20800)	LUT as Memory (9600)
fifo1	28	24	2	40	13	20	8
fifomem (fifomem)	8	0	0		2	0	8
rptr_empty (rptr_empty)	9	0	0		5	9	0
sync_r2w (sync_r2w)	0	0	0		4	0	0
sync_w2r (sync_w2r)	0	0	0		2	0	0
wptr_full (wptr_full)	11	0	0		7	11	0