Design Verification

Ryan Hou

2025-02-18

Table of contents

Preface

Notes on Design Verification. The notes will mainly cover verification concepts, Verilog and SystemVerilog, and UVM. Pre-req: digital logic design fundamentals (see my Digital Logic Design notebook). Common interviews can be found in the Questions chapters.

Resources

Some relevant resources:

- ChipVerify
- Verification Guide
- Verification Academy
- HDLBits
- BSG SystemVerilog Coding Standards

UVM Specific Resources:

- ClueLogic

Textbooks:

- Book 1

For playing around with simulations:

- EDA Playground

List of interview questions: - Verification Guide - ASIC Verification Interview Questions - Verification Guide - SOC Verification Interview Questions - Verification Guide - UVM Interview Questions - Verification Guide - SystemVerilog Interview Questions - ChipVerify - Verilog Interview Set 1 - More can be found on their website - ChipVerify - SystemVerilog Interview Set 1 - More can be found on their website - ChipVerify - UVM Interview Set 1 - More can be found on their website - NAND LAND - FPGA Interview Questions

1 Introduction

1.1 Perspective

> Note 1: Definition - Some definition
>
> **Term** is defined as blah blah blah...

This note does ...

1.2 High Level Ideas

2 Verification Basics

Verification: testing and validating the correctness and functionality of a design

Code coverage tells us how much of the code in the DUT has been executed by the testbench, while **functional coverage** tells us how well the functionality of the DUT has been tested by the testbench.

- **functional coverage** measures how well the verification environment exercises the intended functionality of the design. It checks whether all important scenarios, features, and corner cases are tested. Functional coverage is user-defined
- **Code coverage** measures how much of the RTL code has been exercised by the testbench. It is automatically generated by EDA tools
  - Statement Coverage: Checks if every executable statement in the RTL has been executed.
  - Branch Coverage: Ensures all branches of an if or case statement are exercised.
  - Expression Coverage: Evaluates whether all boolean expressions have been evaluated to both true and false.
  - FSM (Finite State Machine) Coverage: Ensures that all states and transitions in an FSM are exercised.
  - Toggle Coverage: Monitors if all bits of a register or signal toggle from 0→1 and 1→0.
- **Assertion Coverage** measures whether all formal properties and assertions written in SystemVerilog Assertions (SVA) have been exercised during simulation

When can you stop verification?

- No fixed rule! Common criterias:
- Achieving code coverage goals
- Achieving functional coverage goals
- Meeting performance goals
  - timing constraints, power, area, etc
- Finding and fixing all critical bugs
- Available resources

2.1 ASIC Design Flow

1. Requirements
2. Specifications
3. Architecture
4. Digital Design
5. Verification
6. Logic Synthesis
7. Logic Equivalence
8. Placement & Routing
9. Validation

- Note that above steps a huge simplification and are often intertwined and fed back, as multiple iterations of the steps are gone through

2.2 Verification Techniques

- Functional Simulation: running design in simulation to validate its functionality, using various inputs, such as test vectors
- Formal verification: use mathematical proofs to verify the correctness of the design, often used for critical designs (eg safety-critical systems)
- Emulation: testing on specialized hardware that can emulate the behavior of the system, often for large, complex designs that can't be simulated on a computer
- Prototyping: building a physical prototype of the system to test its functionality in a real-world environment (often used for designs that require testing with real-world inputs and conditions)

2.3 Verification Stages

- Planning
- Testbench development
  - Develop testbench environment, provide inputs
- Test creation
  - Create tests based on spec
- Test execution
- Coverage analysis
- Debugging
- Closure

2.3.1 Levels of Verification

- **Unit Level**
  - lowest level, verify individual modules/blocks
- **Block Level**
  - Multiple modules/blocks are integrated and verified together
- **Subsystem Level**
- **Chip level / SOC level**

2.4 Directed Verification

- A type of functional verification where tests are created to test specific features/functions
- Done with different types of testbenches:
  - file-based
  - linear
  - state machine-based

Limitations:

- Limited test coverage
- Bias towards designer's assumptions
- Difficult to detect complex bugs
- Limited scalability
- Time-consuming to create

2.5 Constraint Random Verification

generating randomized test cases with specific constraints to ensure that the generated input stimuli meet certain design requirements.

limitations: The generated test cases may not cover all possible scenarios

2.6 Assertion Based Verification

assertions are used as the primary means of verifying the correctness of a digital design. Assertions are statements that describe a condition that must always be true within a design

basic idea: to use a combination of functional and formal verification techniques to verify the design

SystemVerilog Assertions. Example:

```
assert property ( @(posedge clk) disable iff (!rst_n)
    (in_fifo_wr && !in_fifo_full) ##[1:$] !in_fifo_empty && !in_fifo_wr );
```

2.7 Verification Plan

Contents of a verification plan:

- Overview
- Scope and Goal
- Methodology
- Testbench Architecture
- Test Cases
- Coverage Metrics
  - A description of the coverage metrics to be used, including functional coverage, code coverage, and assertion coverage
- Sign-off Criteria

2.7.1 Example Verification Plan

- Introduction
  - Description of the design and its purpose
  - Goals of the verification process
- Verification Environment
  - Description of the simulation environment
  - List of all verification components
  - Details of the simulation flow
- Features to be Verified
  - List of all design features to be verified

  - Description of the expected behavior of each feature

- Test Plan
  - List of all test cases
  - Description of the purpose and scope of each test case
  - Expected results for each test case
  - Pass/fail criteria for each test case

- Code/Functional/Assertion Coverage Plan
  - Description of the code coverage metrics to be used
  - Goal for each coverage metric
  - Methodology for measuring code coverage

- Bug Tracking
  - Methodology for tracking and reporting bugs
  - Reporting format for each bug
  - Bug severity levels and priority

- Sign-off Criteria
  - Criteria for determining when verification is complete
  - Criteria for determining if the design is ready for tapeout
  - Verification closure process

3 UVM Basics

UVM contains set of classes, methods for creating testbench components. Key components include:

- Testbench components
  - Base classes that can be extended to create tb components (drivers, monitors, scoreboards, agents, etc)
    - UVM classes denoted by `uvm_` prefix
- Transactions
  - Used to model communication between DUT and testbench
  - UVM transaction class - can be extended to carry info between DUT and TB
- Phases
  - set of simulation phases
- Messaging and Reporting
  - Output infrastructure
- Configuration
  - Configuration database
- Functional Coverage
  - UVM provides a mechanism for tracking functional coverage
- Register Abstraction Layer
  - RAL for creating and accessing register maps

In a testbench, **static** entities called **components** exist in a verification environment throughout a simulation, operating on data that flows around the environment. Data or transactions, called **objects** or **sequence items** are dynamic.

3.1 UVM Class Hierarchy

- uvm_object is the main class
- two branches in the hierarchy:
  - uvm_component: verification components
  - uvm_transaction: data objects operated on by verification components

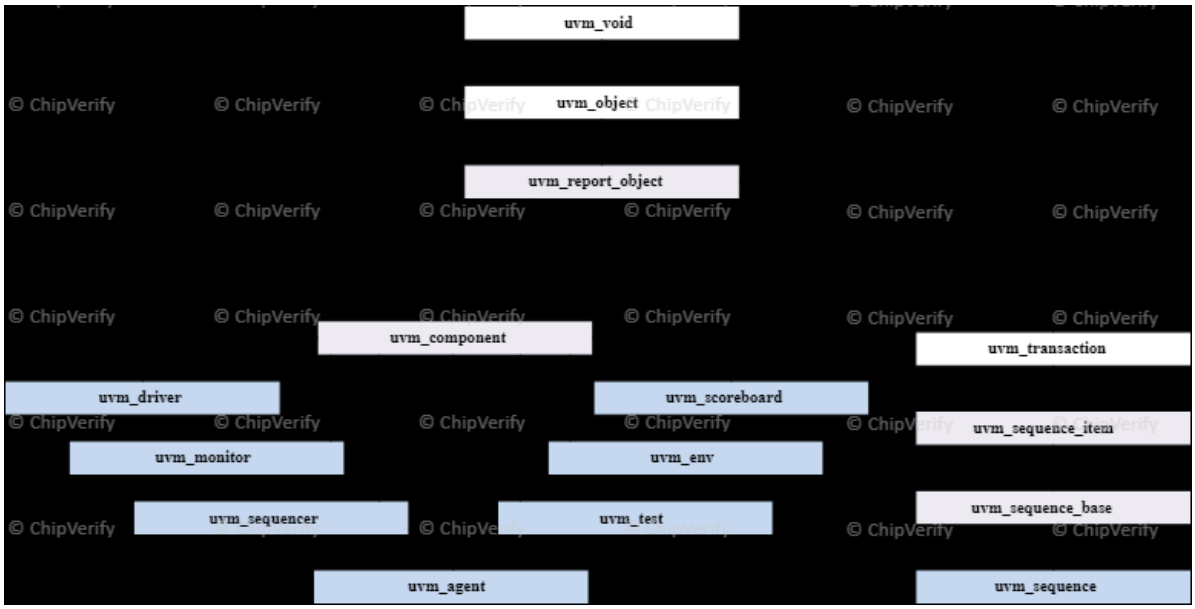


Figure 3.1: Source: ChipVerify

3.2 UVM Class Categories

- UVM Objects
  - configuration classes (define how tb environment is built)
- UVM Sequence
  - A container for the actual stimulus to the design
  - Sequence can be used inside other sequences
- UVM Sequence Items
  - Data objects that have to be driven to DUT
- UVM Components

  - uvm_driver - drives signal to DUT
  - uvm_monitor - monitor signals at DUT output
  - uvm_sequencer - create different test patterns
  - uvm_agent - contains the sequencer, driver, and monitor (an agent)
  - uvm_env - contains all other verification components
  - uvm_scoreboard - checker to determine test pass/fail
  - uvm_subscriber - subscribes to activites of other components

- Register Layer
  - set of classes to help with configuring control registers
-

3.3 TLM Connections

- UVM Phases
  - Phasing enables every component to sync with each other before proceeding to next phase
  - Every component goes through:
    * **build** phase - instantiation
    * **connect** phase - connect w/ other components
    * **run** phase - consume simulation time
    * **final** phase - stops

3.4 References:

- ChipVerify UVM Tutorial

4 Summary

In summary...

5 Questions - Fundamentals of Verification

Key concepts:

- How does verification fit in the product devlopment stack/timeline?
- What teams do verification engineers work with and in what way?
- Come up with a verification plan given a module and their specs/info
- Writing constraints

5.1 Verification Concepts

Q: What is the way to start verifying a design after finishing designing?

Q: What are the steps to verify a design?

Q: What conversations would you have while working with the designer?

Q: After functional simulation and no more bugs, will there be more bugs that will be caught in the emulation / gate level simulation stages?

Q: How would you verify an asynchronous design?

Q: CDC Question: What is the way to get a data input from an asynchronous signal?

Q: Imagine a case where your coverage events are not correct. Top bins are not being hit. Why might this be the case? Brainstorm possible reasons why top bins are being hit?

A: Possible reasons include driver issue, constraint being wrong

Q: Who do DV teams interact with?

Q: What are differences between SoC level verification vs IP/Component level verification?

Q: Design sign-off: when and how?

Q: There are always logical bugs found post-silicon. What to do about those?

Q: Coverage - FSM, toggle, branch, code, functional

5.2 Test Plans

Q: Given a DUT block diagram and timing diagram, try to come up with a verification plan to verify it

Q: Come up with a verification plan for priority arbiter with ack. What test cases would you run? What is your test plan? How would you structure your testbench?

Q: Given a round robin arbiter, where each input to the round robin arbiter is a FIFO, how would you verify this module?

Q: Verification plan - table of contents

5.3 Constraints

Q: You have 4 animal types: Dog, Cat, Sheep, Horse. Each dog can eat exactly 4 apples, each cat can eat exactly 3 apples, each sheep can eat exactly 1 apple, and each horse can eat exactly 10 apples. There are a total of 1024 apples. Write a constraint such that all animal types in total will eat more than 16 apples.

6 Questions - Logic Design & Basics of Digital Systems

Key concepts:

- Basics of logic design
- Basics of digital systems (bits/bytes, hex/bin/dec, throughput/latency, metrics, conversions, etc)
- FIFOs (synchronous and asynchronous)
- Arbiters
- Static timing analysis
- FSMs
- FPGA related roles will ask FPGA-specific optimizations/concepts as well

6.1 Digital System Basics

Q: Calculation of gigabits and speed

Q: How do you tell if your system is little endian or big endian?

Q: Constraint question: You have a total memory size of 4096 bytes. Each page can be 32, 64, 128 bytes. Each page must be aligned. Write a constraint / function that generates a random range that satisfy this constraint

Q: Gray codes, why? How to convert from binary?

1 bit flip between consecutive numbers. To convert from binary, keep the MSB, for subsequent bits, take the XOR of corresponding binary bit and the previous (more significant) binary bit. A Verilog conversion looks something like this:

```
module binary_to_gray #(parameter WIDTH = 4) (
    input  [WIDTH-1:0] binary,
    output [WIDTH-1:0] gray
);
    assign gray = binary ^ (binary >> 1);
endmodule
```

A gray code counter can be implemented as follows (Source: ChipVerify):

```
module gray_ctr
  # (parameter N = 4)

  ( input   clk,
    input   rstn,
    output reg [N-1:0] out);

    reg [N-1:0] q;

    always @ (posedge clk) begin
        if (!rstn) begin
        q <= 0;
            out <= 0;
      end else begin
        q <= q + 1;

`ifdef FOR_LOOP
        // For loop implementation
        for (int i = 0; i < N-1; i= i+1) begin
        out[i] <= q[i+1] ^ q[i];
        end
        out[N-1] <= q[N-1];
`else
            out <= {q[N-1], q[N-1:1] ^ q[N-2:0]};
`endif
    end
    end
endmodule
```

Key ideais to

Q: What is parity bit?

Even parity and odd parity, indicates whether the number of 1s in a binary string is even or odd. To compute the parity bit, take the XOR of all the bits, which gives us the even parity (0 = Even, 1 = Odd). To get odd parity, flip the even parity bit (e.g. by XORing it with 1).

6.2 FIFOs

Q: Asynchronous FIFO qustion (with dual port RAM): gray code? FIFO depth is 10, read/write pointers is 4 bits wide. What should the initial value of the

read/write pointers be?

Q: How would you implement this: An asynchronous FIFO, with 8bit read port and 16 bit write port, and the pointers are encoded in gray code

6.3 Static Timing Analysis

Q: Static timing analysis - given a logic circuit and some delay values, what will the setup and hold time be? How about the maximum clock frequency (given some constraints, circuit, etc)?

Q: (not really STA but timing): How do you delay a signal by N cycles?

6.4 Sequential Logic Design

Entries inserted into a buffer. You have to choose the oldest request among those that are ready. Optimize for power

Q: Regarray vs Flops. Indirect branches

6.5 Logic Design - General

Q: Design a circuit to divide the clock frequency by 4 (f/4) - in sychronous way and asynchronous way

Q: Design a circuit to divide the clock frequency by 3 - 50 % duty cycle

Q: How do you build a larger priority encoder from smaller priority encoders (16:4 from 4:1)?

Q: How do you design an edge detector? How about an edge detector that also detects glitched edges? (normal edge detector circuit does not detect if the transition is from x->x'->x)

Q: How are latches different from flip-flops?

Q: What are the different types of latches? What about flip-flops?

Q: Propagation of metastability...

Q: Design of hierarchical priority encoder?

6.6 FPGAs

Q: How do FPGAs work? How are they programmable?

Q: How to fix setup and hold time in an FPGA?

Q: What are some data structures used in an FPGA?

Q: How is coding Verilog on FPGA different from ASIC?

7 Questions - Computer Architecture

Key concepts:

- In-order 5 stage pipeline design
- Out-of-order CPU design
- Memory hierarchy - caches
- Memory coherence (MSI, MESI, MOESI, etc)
- Memory consistency ()

7.1 ISA

Q: ISA has a set of compressed instructions. Should find the instruction using LSB. Each inst can take one or two lines based on the type. The section selected can start from an inst or an half inst of an uncompressed.

7.2 In-Order CPU

Q: Explain how a 5-stage pipeline works

Q: Why don't we make in-order pipelines deep? Why not make pipelines of anything in general deep (general tradeoffs)?

Q: What are the hazards in an in-order pipeline? How are they detected? What causes them? How does moving towards OoO help with each?

7.3 OoO CPU Design

Q: LSQ. How does it work

Q: (in your project), how did you decoder work?

Q: (in your project), how did you try to optimize your design?

Q: (in your project), what are the sizes of your reservation station, reorder buffer, store queue, other buffers, etc? How did you find that it is the best size?

Q: (in your project) Explain the register path of your project, from the free list to back

Q: What are the differences/tradeoffs between a distributed reservation station and a centralized reservation state?

Q: Explain Tomasulo

Q: What are the different types of hazards in an OoO design? How are they detected and dealt with?

Q: FSM for a Hybrid Branch Predictor to choose between two BP using Global and Local Branch History

Q: Design a Simple ROB and Retirement logic

Q: Questions and Simple design on Issue Queue and Reservation station

7.4 Memory Architecture, Caches

Q: find the number of tag bits in the cache

Q: How would you determine if the requested data is available in the cache?

Q: What LRU policy do you use? (in project setting or for specific scenario)

Q: Assume a scenario where a cache line gets evicted and immediately the CPU Wants the same cache line. What should you do?

Q: What type of cache did you use? (for project setting or specific scenario)

Q: NUMA vs UMA. Differences and tradeoffs?

Q: Virtual memory, and TLBs. How do they work, what are they for?

Q: Aliasing of memory addresses in a cache. How to solve aliasing and conflict misses?

Q: What are the different types of cache misses?

Q: What are the different types of cache writeback policies and eviction policies?

Q: You have a 16-way cache. How would you speed it up?

Q: In cache design, explain the differences/tradeoffs between DM vs FA vs SA. What situations would you use them in?

Q: True LRU vs Pseudo LRU

Q: True LRU: How many bits to implement?

Q: Given an age vector for a LRU cache and a valid/invalid vector, how would you choose which way to evict?

Q: Prefetcher and I-cache logics

Q: (for your project) implementation details on your Load Store Queue

Q: Come up with cache access patterns in which FA would perform better than DM. In general: come up with a cache access pattern that performs better for each of DM, FA, SA

Q: Write back vs Write through, MRU vs LRU. (and what did you do in your project and how you went about choosing this?)

Q: Non-blocking D-cache and how you implement it?

Q: Virtual memory, VIPT. Explain

7.5 Branch Prediction

Q: Perceptron, what is it? Equation? How do you implement it in hardware?

7.6 Memory Coherency

Q: What is memory coherence and why is it needed?

Q: MSI protocol. What is it, how does it work?

Q: MSI, MESI, MOESI: why was each extra state added?

Q: Draw the state diagram for MESI

Q: Explain different cache coherency protocol.

7.7 Memory Consistency

Q: Consistency basics, what is memory consistency and why is it needed?

Q: Given a sequence of instructions on 2 cores, which values are possible/not possible - how would you check if a barrier is working?

7.8 Interconnect

Q: How do credits work in protocols

A:

- Credit Initialization:
  - The receiver starts with a certain number of credits, indicating how many messages or flits (flow control units) it can accept.
  - The sender maintains a credit counter that tracks how many credits it has available.
- Sending Data:
  - Every time the sender transmits a packet/flit, it decrements its available credits.
  - If the sender runs out of credits, it must stop sending until it receives more.
- Credit Return:
  - When the receiver processes or forwards a message, it returns a credit back to the sender, signaling that it has space for more data.
  - This ensures that the receiver's buffer never overflows.
- Use cases:
  - NoCs: Ensures flow control in mesh/toroidal interconnects to avoid congestion.
  - Coherent Interconnects (e.g., AMBA CHI, CXL, UPI): Used to manage request/response ordering and avoid buffer overflow.
- Why use credits:
  - Prevent buffer overflows at the receiver
  - Help load balance across links in NoC
  - Ensure efficient resource utilization

8 Questions - Coding (Non-HDL)

Common "software" coding questions, concepts, and ideas asked in a design verification interview:

- Bit manipulation
- String parsing/manipulation
- Basic recursive algorithms and their iterative counterparts
- Object-oriented design (polymorphism, inheretence, encapsulation, abstraction)
- Various algorithmic concepts (DFS, BFS, TSP)
- Parallelism - multi-threading (fork, join), writing parallelised algorithms
- Basic data structures and their implementation
- Basic array leetcode problems
- Mostly in C++ or Python (or another choice of scripting language)

8.1 Bit Manipulation

Q: Bit manipulation leetcode questions

8.2 Strings

Q: String matching regex

Q: Parse a string in Python: Given a .txt file, parse the file and print the frequency of each word, printed by ordering of most frequent first.

8.3 Data Structures

Q: Code up a class for a stack, including functions for push and pop

Q: Write C++ code to move value to the front of an array

Q: Write C++ code to find the second minimum of an array

Q: Two ints are represented as a linked list. Create a new linked list that is the sum of these two integers

Q: How do you build a queue from a stack? How do you build a stack from a queue?

Q: C program to find the length of a string

8.4 Recursion/Iteration

Q: Write a function to calculate the factorial of a given number. Recursively and iteratively

```
long long factorial_recursive(int n) {
    if (n == 0 || n == 1)
        return 1;
    return n * factorial_recursive(n - 1);
}

long long factorial_iterative(int n) {
    long long result = 1;
    for (int i = 2; i <= n; i++) {
        result *= i;
    }
    return result;
}
```

Q: Write a function to output the fibonacci sequence recursively. How about iteratively? What are the differences (especially in terms of complexity)?

```
long long fibonacci_recursive(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fibonacci_recursive(n - 1) + fibonacci_recursive(n - 2);
}

void fibonacci_iterative(int n) {
    long long a = 0, b = 1, c;
    cout << "Fibonacci (Iterative) sequence up to " << n << " terms: ";

    for (int i = 0; i < n; i++) {
        cout << a << " ";
```

```
        c = a + b;
        a = b;
        b = c;
    }
    cout << endl;
}
```

Q: Deleting objects at the end after execution (freeing up dynamic memory): use linked list, vector, or a counter

8.5 Object-Oriented Design

Q: What is polymorphism? What is inheritance?

- LinkedIn Post - Polymorphism in DV Interview

8.6 Others

Q: Write the pseudocode for how you would go about solving the traveling salesman problem

Q: Write code for a card game in C++: Black jack - shuffle, deal, and check the sum is equal to a value. If greater then lose. If rounds are complete, then highest wins

Q: Write code to perform matrix multiplication in C++

Q: Write C++ code to implement LRU. What is the time complexity?

Q: Loop interchange - to improve cache hit rate

Q: Compilers - register renaming

Q: How would you code Fibonacci in assembly?

8.7 Scripting

Q: In Python, parse a file line-by-line and finding the minimum time for a particular bucket id given this file structure: TASK_ID | STATUS | COMMAND_LINE | BUCKET_ID | TIME (HH:MM:SS)

9 Questions - Verilog

Key concepts

- Blocking vs Non-blocking
- How will given code synthesize
- Coding up FIFOs, arbiters, FSMs

Q: Given some Verilog code (blocking vs non-blocking assignment), what will these synthesize to?

Q: Code fizz buzz in Verilog

Q: How does polymorphism work in SystemVerilog?

Q: How do virtual interface work in SystemVerilog?

Q: What do clocking blocks do in SystemVerilog? What are they?

Q: Differences between Verilog and SystemVerilog?

Q: Code a sequence (pattern) detector state machine. How would you verify this state machine?

Q: SV data structures: queues, maps, associative array, dynamic array

Q: Write the code for a FIFO in SystemVerilog

Q: SystemVerilog Fork, Join(). Given some example code with these, what would happen to a task (e.g. a given task could take forever)? What could you do to fix it (e.g. by writing a task to print ERROR if a task doesn't return by TIMEOUT seconds)?

Q: Traffic light signaling problem in Verilog (state machine). Use counter and output color depending on range using assign statements

Q: Given an FSM state diagram, code it in SystemVerilog

Q: Logic vs wire vs bit in Verilog/SystemVerilog

Q: Write the RTL to generate a signal that is clock for 3 cycles then 0 for 3 cycles

Q: Given 3 enums, constrain 32 bits to be within these 3 enums. Then constrain their distribution

Q: SystemVerilog Assertions. Write assertions for Req and Ack with the given spec:

- Ack should assert within 5-10 cycles after Req asserts
- Ack should deassert within 5-10 cycles after Req desserts
- No Ack should assert before Req
- No Ack should deassert before Req deasserts

Q: CAM & TCAM in SystemVerilog. Modeling with 2D/3D array vs associative array (key as data, addr as value), and pros and cons of these approaches

Q: Write a module to convert binary to gray code (how about vice versa)?

Q: Write a pattern detector that detects input sequence of 101 from a binary stream. (Pattern 1101, 1011 are also common)

Q: Posedge reset vs negedge reset? Why?

Q: Write a stack in Verilog

9.1 Verilog Modules

9.1.1 Asynchronous FIFO

- Asynchronous FIFO: data read and writes use different clock frequencies (i.e. reads and writes are not synchronized)
- Usage: in systems with clock domain crossing to help synchronize data flow between systems in different clock domains. e.g. used to pass data from system A at 100MHz clock to system B at 125MHz clock

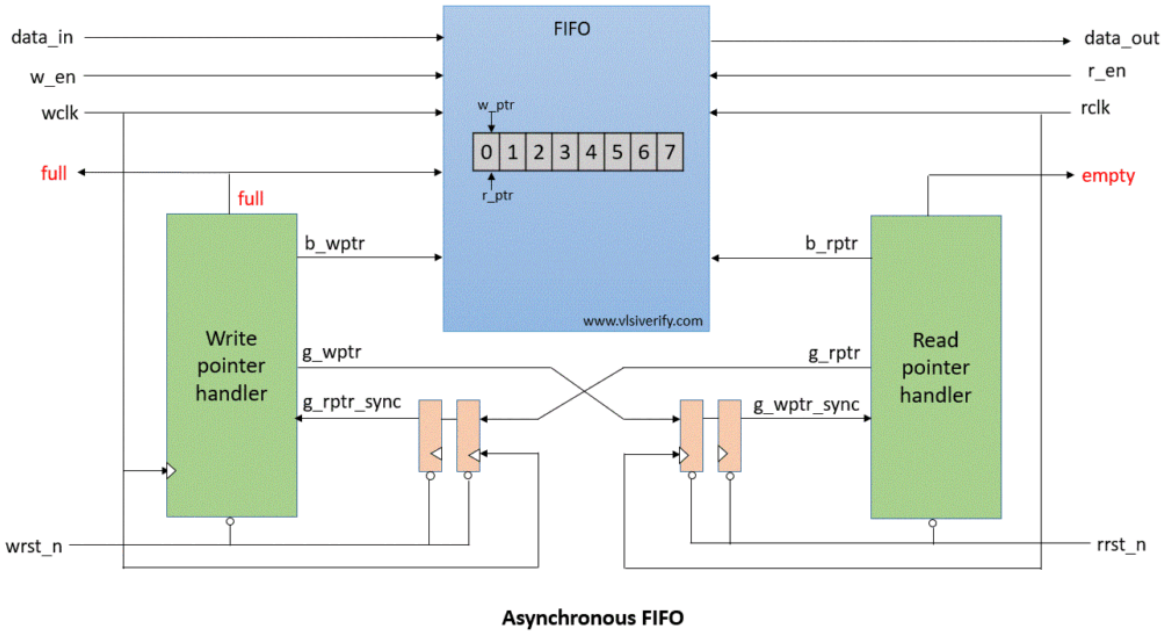


Figure 9.1: Source: VLSI Verify

•

9.2 Operation:

References:

- VLSI Verify

9.2.1 Synchronous FIFO

- Key parameters:
  - Depth: Number of entries in the FIFO
  - Width: Number of bits of each entry
- Synchronous FIFO: data read and write use the same clock frequency
- Operation:
  - When write enable is on, write the write data until full. Write pointer gets incremented on every data write
  -

9.2.1.1 Method 1 -

```
module synchronous_fifo #(parameter DEPTH=8, DATA_WIDTH=8) (
  input clk, rst_n,
  input w_en, r_en,
  input [DATA_WIDTH-1:0] data_in,
  output reg [DATA_WIDTH-1:0] data_out,
  output full, empty
);

  reg [$clog2(DEPTH)-1:0] w_ptr, r_ptr;
  reg [DATA_WIDTH-1:0] fifo[DEPTH];

  // Set Default values on reset.
  always@(posedge clk) begin
    if(!rst_n) begin
      w_ptr <= 0; r_ptr <= 0;
      data_out <= 0;
    end
  end

  // To write data to FIFO
  always@(posedge clk) begin
    if(w_en & !full)begin
      fifo[w_ptr] <= data_in;
      w_ptr <= w_ptr + 1;
    end
  end

  // To read data from FIFO
  always@(posedge clk) begin
    if(r_en & !empty) begin
      data_out <= fifo[r_ptr];
      r_ptr <= r_ptr + 1;
    end
  end

  assign full = ((w_ptr+1'b1) == r_ptr);
  assign empty = (w_ptr == r_ptr);
endmodule
```

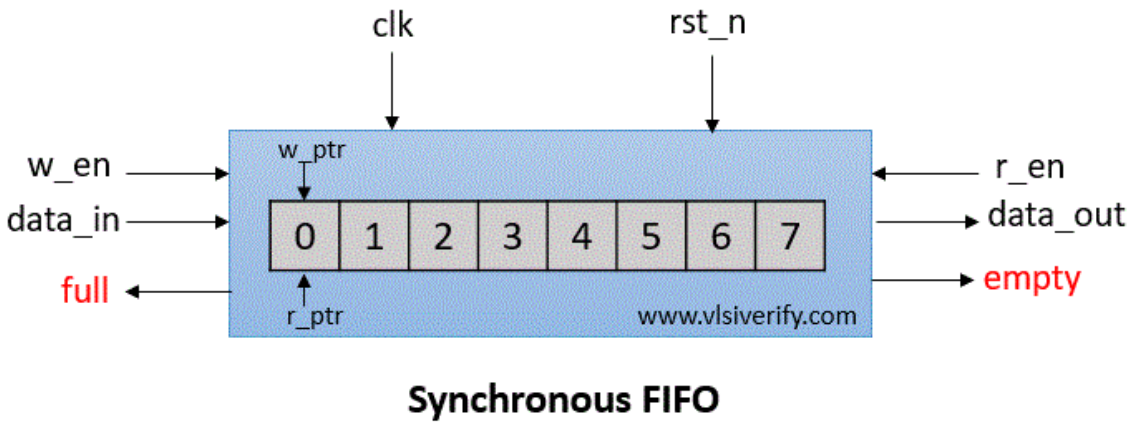


Figure 9.2: VLSI Verify

References:

- VLSI Verify

9.2.2 Arbiter

Case Statement Implementation

```
casez (req[3:0])
    4'b???1 : grant <= 4'b0001;
    4'b??10 : grant <= 4'b0010;
    4'b?100 : grant <= 4'b0100;
    4'b1000 : grant <= 4'b1000;
    4'b0000 : grant <= 4'b0000;
endcase
```

Assign Statement

```
grant[0] = req[0];
grant[1] = ~req[0] & req[1];
grant[2] = ~req[0] & ~req[1] & req[2];
// ...etc...
```

Parameterized Assign

```
parameter N = 16; // Number of requesters

// For example, higher_pri_reqs[3] = higher_pri_reqs[2] | req[2];
assign higher_pri_reqs[N-1:1] = higher_pri_reqs[N-2:0] | req[N-2:0];
assign higher_pri_reqs[0] = 1'b0;
assign grant[N-1:0] = req[N-1:0] & ~higher_pri_reqs[N-1:0];
```

References:

- Arbiter Design Ideas & Coding Styles

9.2.3 Round-Robin Arbiter

References:

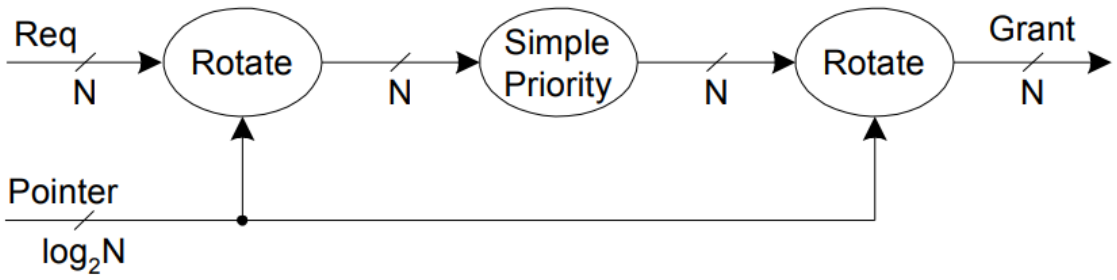
- Arbiter Design Ideas & Coding Styles

9.2.3.1 Naive

```
always_comb begin
    case (pointer_reg)
        2'b00 :
            if (req[0]) grant = 4'b0001;
            else if (req[1]) grant = 4'b0010;
            else if (req[2]) grant = 4'b0100;
            else if (req[3]) grant = 4'b1000;
            else grant = 4'b0000;
        2'b01 :
            if (req[1]) grant = 4'b0010;
            else if (req[2]) grant = 4'b0100;
            else if (req[3]) grant = 4'b1000;
            else if (req[0]) grant = 4'b0001;
            else grant = 4'b0000;
            2'b10 :
            if (req[2]) grant = 4'b0100;
            else if (req[3]) grant = 4'b1000;
            else if (req[0]) grant = 4'b0001;
            else if (req[1]) grant = 4'b0010;
            else grant = 4'b0000;
        2'b11 :
```

```
            if (req[3]) grant = 4'b1000;
            else if (req[0]) grant = 4'b0001;
            else if (req[1]) grant = 4'b0010;
            else if (req[2]) grant = 4'b0100;
            else grant = 4'b0000;
    endcase // case(req)
end

// Pointer Reg Logic:
logic [1:0] pointer_req, next_pointer_req;

  always @(posedge clock) begin
    if (reset) pointer_req <= '0;
    else       pointer_req <= next_pointer_req;
  end

  always_comb begin
    assign next_pointer_req = 2'b00;
    casez (gnt)
      4'b0001: next_pointer_req = 2'b01;
      4'b0010: next_pointer_req = 2'b10;
      4'b0100: next_pointer_req = 2'b11;
      4'b1000: next_pointer_req = 2'b00;
    endcase
  end
```

9.2.3.2 Rotate + Priority + Rotate



- Idea: rotate by amount specified by pointer, feed to simple arbiter, then rotate back

9.2.3.3 MUXed Parallel Priority Arbiter

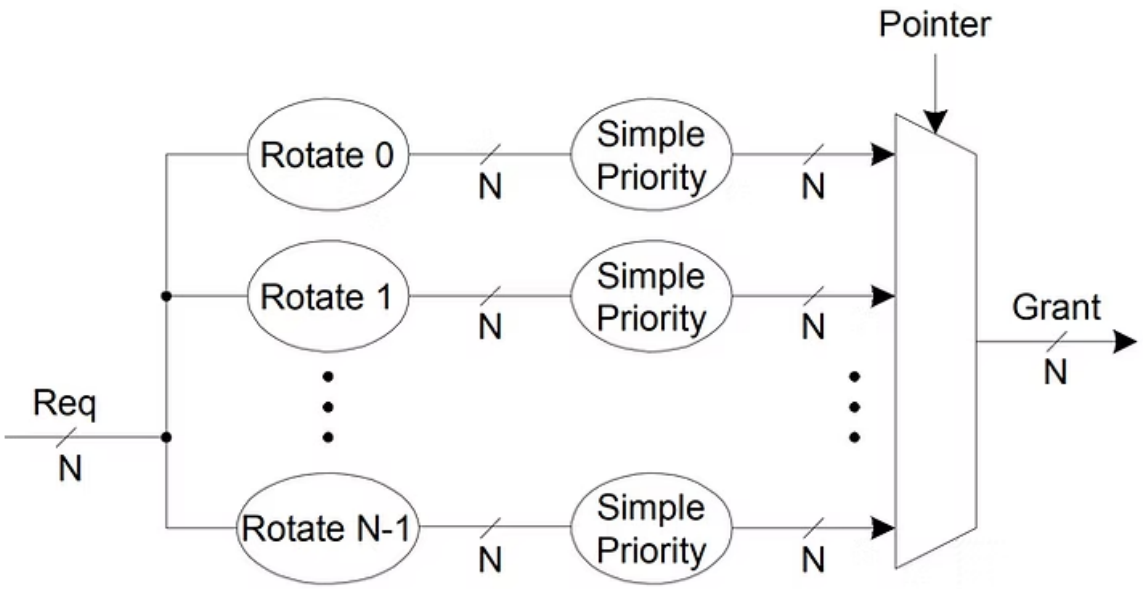


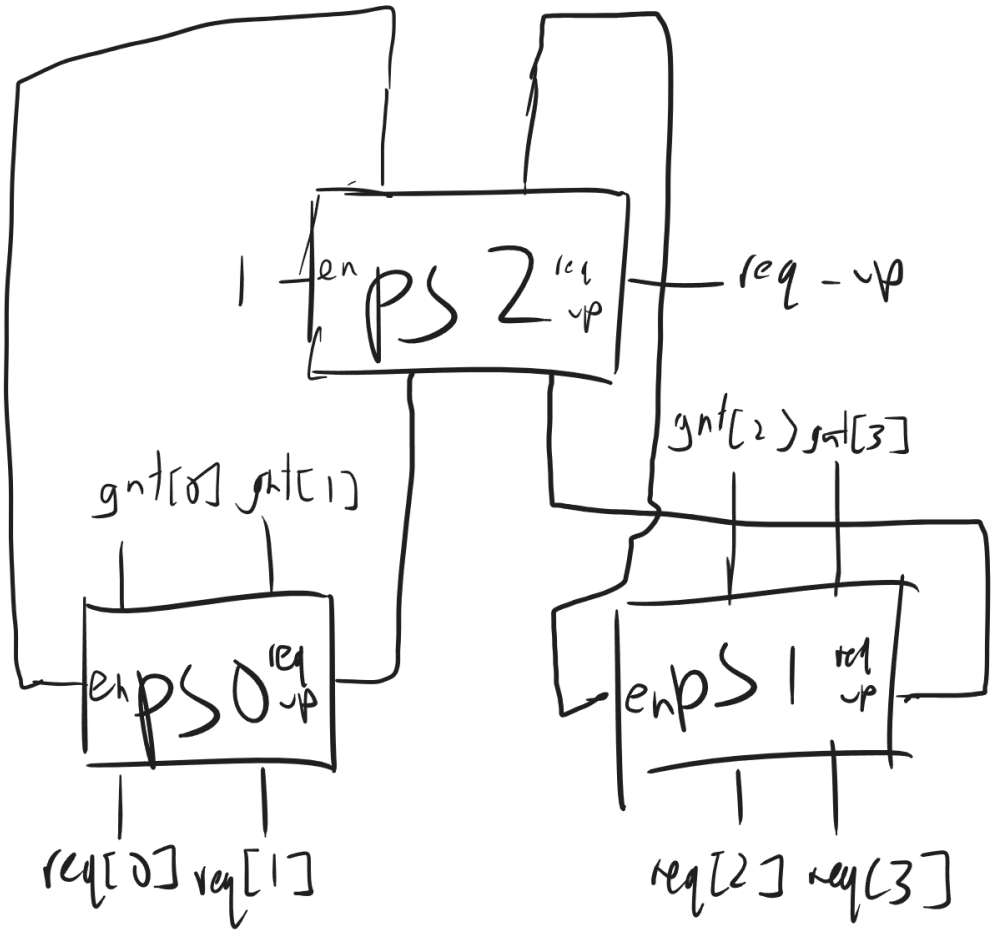
Figure 9.3: Source

- Rotate-i block rotates the request by i positions (just rewiring)
- Pointer selects the intermediate grant vectors to use
- Critical path: MUX tree (that can be built with 2:1 MUXes in logN complexity)
- Reference: Code Source

```
module muxed_rr_arb(
  input logic clock,
              reset,
  input logic [3:0] req,
  output logic [3:0] gnt
);

  logic [3:0] mux_ip0, mux_ip1, mux_ip2, mux_ip3;

  //Instantiate fixed priority arbiter and calculate the grant output for shifted priorities
  fixed_pri_arbiter inst0 (.req(req),    .gnt(mux_ip0));
  fixed_pri_arbiter inst1 (.req(req>>1), .gnt(mux_ip1));
  fixed_pri_arbiter inst2 (.req(req>>2), .gnt(mux_ip2));
```

```
  fixed_pri_arbiter inst3 (.req(req>>3), .gnt(mux_ip3));

  //Select line pointer calculation
  logic [1:0] pointer_req, next_pointer_req;

  always @(posedge clock) begin
    if (reset) pointer_req <= '0;
    else     pointer_req <= next_pointer_req;
  end

  always_comb begin
    assign next_pointer_req = 2'b00;
    casez (gnt)
      4'b0001: next_pointer_req = 2'b01;
      4'b0010: next_pointer_req = 2'b10;
      4'b0100: next_pointer_req = 2'b11;
      4'b1000: next_pointer_req = 2'b00;
    endcase
  end

  //Final output
  always_comb begin
    case (pointer_req)
      2'b00: gnt = mux_ip0;
      2'b01: gnt = mux_ip1;
      2'b10: gnt = mux_ip2;
      2'b11: gnt = mux_ip3;
    endcase
  end

endmodule
```

9.2.4 Rotating Priority Selector




```
// 2-Bit Rotating Priority Selector
module rps2(
    input           [1:0] req,
    input                 sel,
    input                  en,
    output logic       req_up,
    output logic    [1:0] gnt
);

    // Set grant lines
```

```
    assign gnt[1] = en && req[1] && (sel || (~sel && ~req[0]));
    assign gnt[0] = en && req[0] && (~sel || (sel && ~req[1]));

    // req_up is asserted if either request is asserted (no matter the value of the enable)
    assign req_up = req[1] || req[0];

endmodule

// ---- N-Bit Rotating Priority Selector ----
module rps #(parameter NUM_LINES = `NUM_LINES_DEFAULT) (
    input clock,
    input reset,
    input [NUM_LINES-1:0] req,
    input              en,

    output [NUM_LINES-1:0] gnt,
    output              req_up
  `ifdef DEBUG
  ,
  output logic [$clog2(NUM_LINES)-1:0] rps_priority
  `endif
);

    logic [$clog2(NUM_LINES)-1:0] count;
  assign rps_priority = count;
    rps_counter #(.NUM_LINES(NUM_LINES)) cnt(
        .clock(clock),
        .reset(reset),
        .count(count)
    );

  logic [NUM_LINES-2:0] [$clog2($clog2(NUM_LINES))-1:0] sel_idx;
    wire [NUM_LINES-2:0] req_ups;
    wire [NUM_LINES-2:0] enables;

    assign req_up = req_ups[NUM_LINES-2];
    assign enables[NUM_LINES-2] = en;

    genvar i,j;
    generate
        // not well-defined for NUM_LINES < 2 (what are you selecting between?)
        if (NUM_LINES == 2) begin
```

```
            rps2 single (
                .req    (req),
                .sel    (count),
                .en     (en),
                .req_up (req_up),
                .gnt    (gnt)
            );
        end else begin
            for(i = 0; i < NUM_LINES/2; i = i+1) begin
                rps2 base (
                    .req    (req[2*i+1:2*i]),
                    .sel    (count[0]),
                    .en     (enables[i]),
                    .req_up (req_ups[i]),
                    .gnt    (gnt[2*i+1:2*i])
                );
            end

            for(j = NUM_LINES/2; j <= NUM_LINES-2; j = j+1) begin
                assign sel_idx[j] = $ceil($ln(NUM_LINES)/$ln(2) - $ln(NUM_LINES-1-j)/$ln(2) -
                rps2 top (
                    .req    (req_ups[2*j-NUM_LINES+1:2*j-NUM_LINES]),
                    .sel    (count[sel_idx[j]]),
                    .en     (enables[j]),
                    .req_up (req_ups[j]),
                    .gnt    (enables[2*j-NUM_LINES+1:2*j-NUM_LINES])
                );
            end
        end
    endgenerate
endmodule
```

9.2.5 Counters

9.2.6 Gray Code Counter

Implementation with Binary Counter + Gray Code Converter

A gray code counter can be implemented as follows (Source: ChipVerify):

```
module gray_ctr
  # (parameter N = 4)

  ( input   clk,
    input   rstn,
    output reg [N-1:0] out);

    reg [N-1:0] q;

    always @ (posedge clk) begin
        if (!rstn) begin
        q <= 0;
            out <= 0;
      end else begin
        q <= q + 1;

`ifdef FOR_LOOP
        // For loop implementation
        for (int i = 0; i < N-1; i= i+1) begin
        out[i] <= q[i+1] ^ q[i];
        end
        out[N-1] <= q[N-1];
`else
            out <= {q[N-1], q[N-1:1] ^ q[N-2:0]};
`endif
    end
    end
endmodule
```

Direct Implementation (USC - Gray Counter Design)

Key idea: in a gray code up-counter, a bit toggles if all the bits to the right of it are 100… (1 with all 0s). However, after incrementing and flipping current bit, we still get 1 more count where bits to the right are 100…. So we also take into account the parity of the left side as well.

9.3 Online References

- Blind Post - Design Verification Interview

10 Questions - UVM

Q: What UVM phase is different from the others, and why?

Q: UVM Polymorphism question, child handle

References