Testing and Continuous Integration

Emad Jacob Maroun

Technical University of Denmark
Embedded Systems Engineering

October 7, 2025

Agile Manifesto Focus

Individuals and interactions over processes and tools

Working software over comprehensive documentation

Customer collaboration over contract negotiation

Responding to change over following a plan [1]

[1] https://agilemanifesto.org/

Agile Manifesto Focus

Individuals and **interactions** over processes and tools

Working ~~software~~ **hardware** over comprehensive documentation

Customer collaboration over contract negotiation

Responding to change over following a plan [1]

[1] https://agilemanifesto.org/

Introduction

- What is testing?

Introduction

- What is testing?
  - Evaluating if a component/system behaves as expected

Introduction

- What is testing?
  - Evaluating if a component/system behaves as expected
  - Verification:

Introduction

- What is testing?
  - Evaluating if a component/system behaves as expected
  - Verification: Evaluating whether a system meets its specifications

Introduction

- What is testing?
  - Evaluating if a component/system behaves as expected **by the developer**
  - Verification: Evaluating whether a system meets its specifications
  - Less intensive & extensive for quick development

Introduction

- What is testing?
  - Evaluating if a component/system behaves as expected **by the developer**
  - Verification: Evaluating whether a system meets its specifications
  - Less intensive & extensive for quick development
- Why test?
  - Verify functionality
  - Catch errors & bugs

Introduction

- What is testing?
  - Evaluating if a component/system behaves as expected **by the developer**
  - Verification: Evaluating whether a system meets its specifications
  - Less intensive & extensive for quick development
- Why test?
  - Verify functionality
  - Catch errors & bugs
  - Document requirements
  - Catch regressions
  - Improve collaboration through iteration

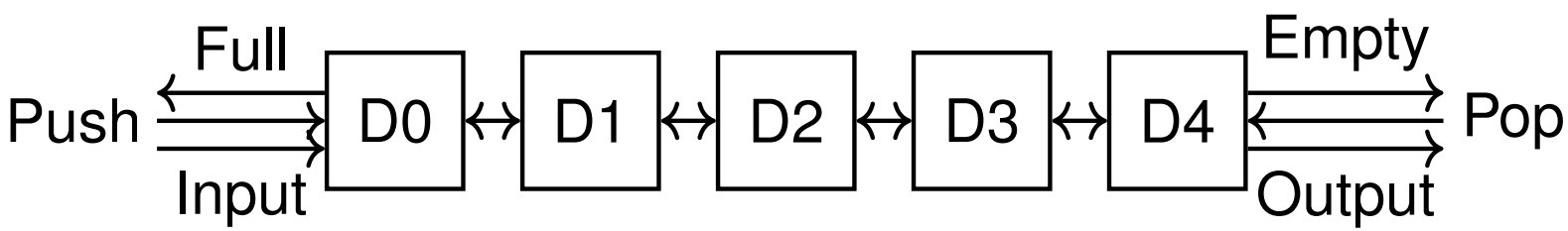
Introduction

- What is testing?
  - Evaluating if a component/system behaves as expected **by the developer**
  - Verification: Evaluating whether a system meets its specifications
  - Less intensive & extensive for quick development
- Why test?
  - Verify functionality
  - Catch errors & bugs
  - Document requirements
  - Catch regressions
  - Improve collaboration through iteration
- Continuous integration:
  - Centralized server as single source of truth
  - Runs all tests on every merge/pull request (PR)
  - Ensures project is always in a working state
  - Maybe: Manage deployments (continuous deployment)

Running Example: FIFO Queue

What is a FIFO Queue?

- First-In, First-Out
- First element added is the first one removed
- Commonly used for buffering and data flow control



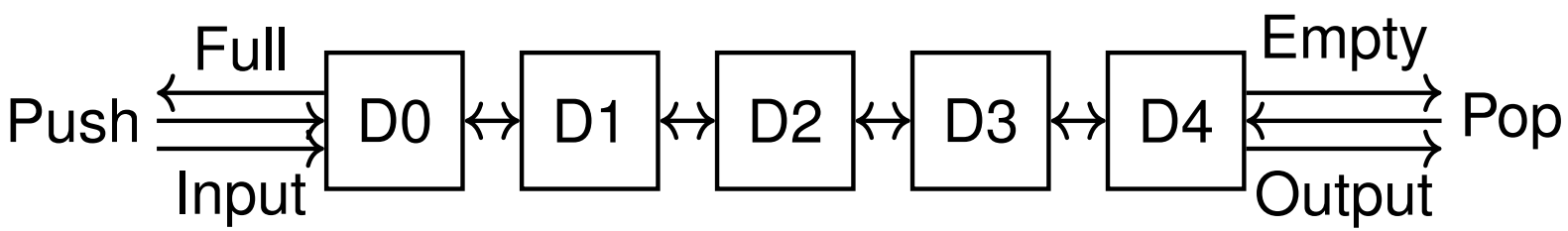
Running Example: FIFO Queue

What is a FIFO Queue?

- First-In, First-Out
- First element added is the first one removed
- Commonly used for buffering and data flow control

Applications

- Temporary storage between producer and consumer
- Clock domain crossing
- Managing data streams in communication



Testing Levels

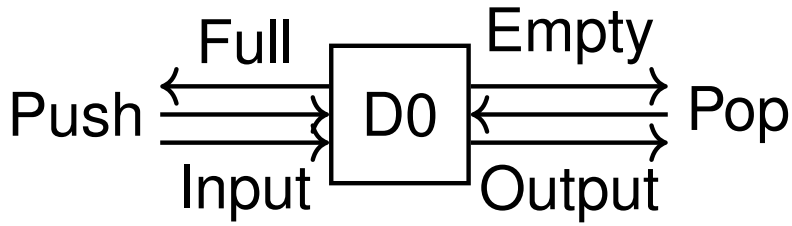
1. **Unit Tests**
   - Testing individual components in isolation
   - Exercise basic units of functionality
2. **Integration Tests**
   - Testing component interaction
   - Compound units of functionality
3. **System Tests**
   - Testing the complete system
   - Use-case/user story tests
4. **Acceptance Tests**
   - Testing by the client
5. Non-functional tests: Speed, size, documentation

Testing Levels - Unit Tests

- Identify limited functionality
- Create tests covering the functionality
- Run and verified locally by developers

Testing Levels - Unit Tests

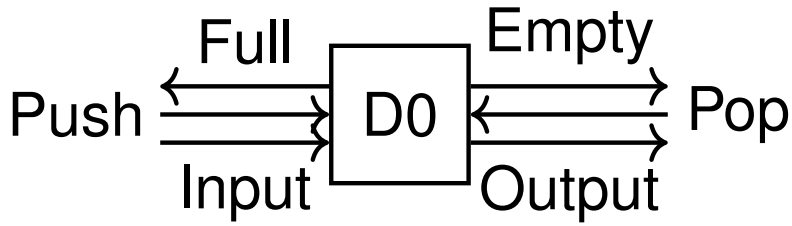
- Identify limited functionality
- Create tests covering the functionality
- Run and verified locally by developers



FIFO Unit Tests (10): Exercise: 3 mins

Testing Levels - Unit Tests

- Identify limited functionality
- Create tests covering the functionality
- Run and verified locally by developers



FIFO Unit Tests (10): Exercise: 3 mins

- Initialized to empty
- On push becomes full & !empty
- On pop becomes !full & empty
- If empty, pop does nothing (pops zero?)
- If empty, push updates next pop
- If full, can pop
- If full, push doesn't change next pop
- Values pushed will be popped next cycle
- Can store indefinitely
- Can push and pop in simultaneously (forwarding?)

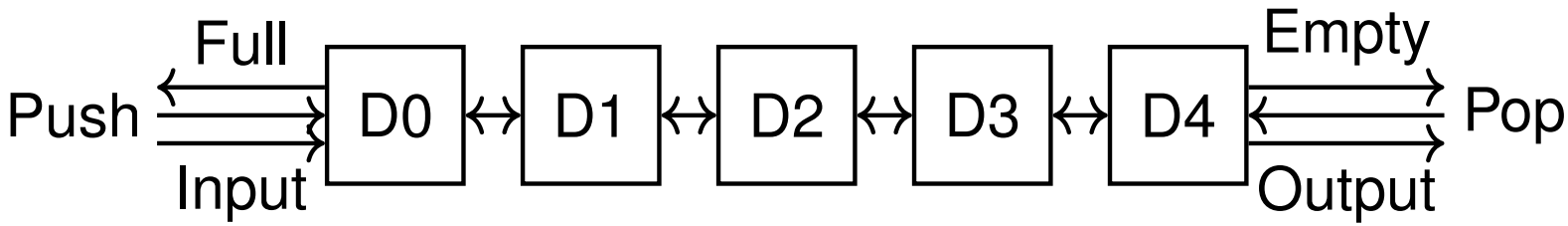
Testing Levels - Integration Tests

- Identify common module interactions
- Create tests covering the functionality
- Run by either developers or dedicated testers

Testing Levels - Integration Tests

- Identify common module interactions
- Create tests covering the functionality
- Run by either developers or dedicated testers

FIFO Tests (9): Exercise: 5 mins

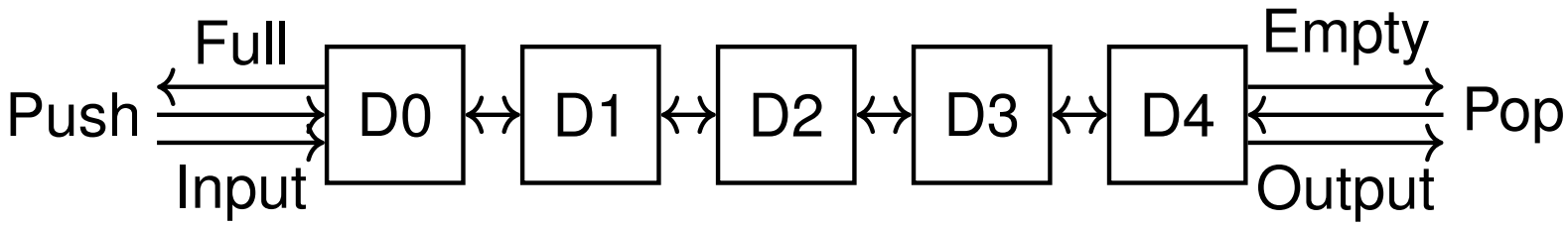


Testing Levels - Integration Tests

- Identify common module interactions
- Create tests covering the functionality
- Run by either developers or dedicated testers

FIFO Tests (9): Exercise: 5 mins

- Initialized to empty
- On push becomes !empty
- On pop becomes !full
- Pushed item will pop eventually
- List of items will pop in pushed order
- If full, can pop and push simultaneously
- Can store indefinitely
- Items get to the front without popping
- Two lists pushed eventually become one list



Test-Driven Development (TDD)

What is TDD?

- A software development approach where tests are written **before** the code
- Follows a short cycle: **Red** $\rightarrow$ **Green** $\rightarrow$ **Refactor**

Test-Driven Development (TDD)

What is TDD?

- A software development approach where tests are written **before** the code
- Follows a short cycle: **Red** $\rightarrow$ **Green** $\rightarrow$ **Refactor**

TDD Cycle

1. **Write a test** for a small piece of functionality
2. **Run the test** – it should fail (Red)
3. **Write the code** to make the test pass (Green)
4. **Refactor** the code while keeping the test passing

Test-Driven Development (TDD)

What is TDD?

- A software development approach where tests are written **before** the code
- Follows a short cycle: **Red** $\rightarrow$ **Green** $\rightarrow$ **Refactor**

TDD Cycle

1. **Write a test** for a small piece of functionality
2. **Run the test** – it should fail (Red)
3. **Write the code** to make the test pass (Green)
4. **Refactor** the code while keeping the test passing

Benefits in Agile Design

- Encourages modular, testable design
- Provides immediate feedback and confidence in changes
- Supports continuous integration and rapid iteration

Importance of Testable Design

Why Design for Testability?

- **Early Bug Detection:** Easier to identify and fix issues during development
- **Supports Automation:** Enables integration with CI
- **Improves Maintainability:** Modular and testable components are easier to update and refactor
- **Enables Agile Practices:** Prerequisite for iterative, feedback-driven development

Importance of Testable Design

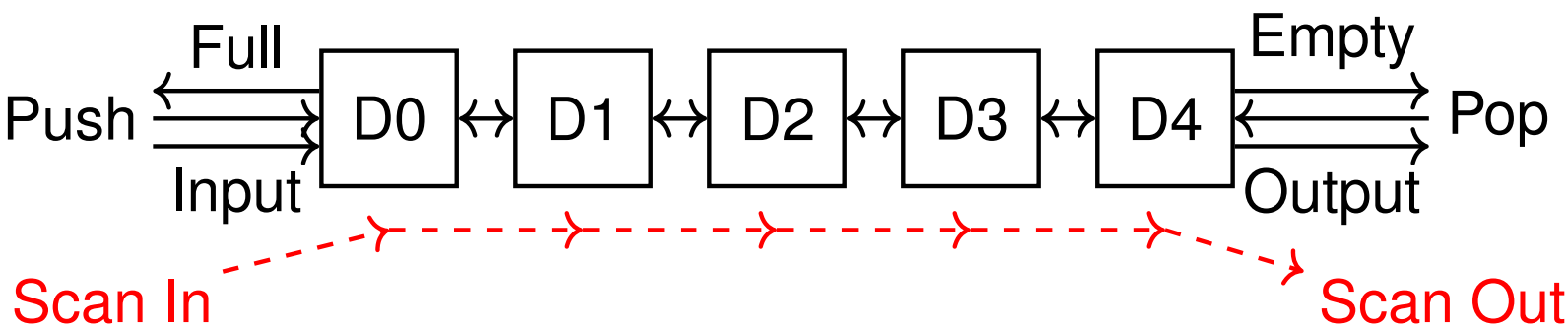
Why Design for Testability?

- **Early Bug Detection:** Easier to identify and fix issues during development
- **Supports Automation:** Enables integration with CI
- **Improves Maintainability:** Modular and testable components are easier to update and refactor
- **Enables Agile Practices:** Prerequisite for iterative, feedback-driven development

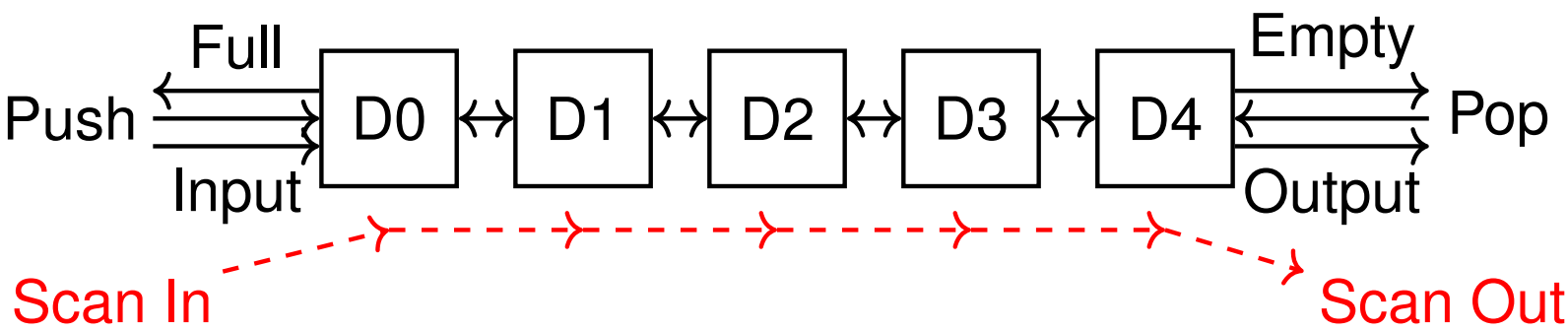
In Digital Design Context:

- Use of simulation testbenches for modules like FIFOs
- Design-for-Testability (DFT) features:
  - Scan chains for test setup and verification
  - Built-in self-test (BIST)
- Clear separation of control and data paths for easier verification

Scan Chain - FIFO



Scan Chain - FIFO



How does the scan chain change testing?

Property-Based Testing (PBT)

Why Property-Based Testing?

- Many systems have an **infinite input space** — it's impractical to test all possible cases manually
- PBT helps uncover **classes of bugs** by generating a wide range of random, valid inputs

Property-Based Testing (PBT)

Why Property-Based Testing?

- Many systems have an **infinite input space** — it's impractical to test all possible cases manually
- PBT helps uncover **classes of bugs** by generating a wide range of random, valid inputs

Core Strategy:

1. **Identify a property** that should always hold (e.g., "push followed by pop returns the same value")
2. **Define an input generator** that produces only valid inputs
3. **Implement a reference (golden) model** to compare expected behavior
4. **Run the test many times** with randomized inputs

Property-Based Testing (PBT)

Why Property-Based Testing?

- Many systems have an **infinite input space** — it's impractical to test all possible cases manually
- PBT helps uncover **classes of bugs** by generating a wide range of random, valid inputs

Core Strategy:

1. **Identify a property** that should always hold (e.g., "push followed by pop returns the same value")
2. **Define an input generator** that produces only valid inputs
3. **Implement a reference (golden) model** to compare expected behavior
4. **Run the test many times** with randomized inputs

When a bug is found:

- The failing input is **shrunk** to a minimal counterexample
- A new **unit or integration test** is created to capture and prevent regression

Property-Based Testing in Chisel

- Use of ScalaCheck for PBT
- Verifying hardware modules with randomized inputs

Example:

```
property("Push then pop") =
forAll { (in: UInt) =>
 test(new FifoBlock()) { c =>
  c.io.push.poke(true.B)
  c.io.inputs.poke(in)
  c.clock.step()
  c.io.push.valid.poke(false.B)
  c.io.pop.poke(true.B)
  c.io.output.expect(in)
  c.clock.step()
  c.io.output.expect(0)
 }
}
```

Key Points:

- `forAll` generates random valid inputs
- The property asserts that pop returns the pushed value
- Failing cases are minimized and can be turned into regression tests

Example: Push/pop list

```
property("FIFO preserves order of items") =
forAll { (depth: Int, inputs: List[Int]) =>
 whenever(depth > 0 && depth <= 32 && inputs.length <= depth) {
  test(new MyFifo(depth)) { c =>
   // Enqueue all elements
   for (in <- inputs) {
    c.io.push.poke(true.B)
    c.io.input.poke(in.U)
    c.clock.step()
   }
   while (c.io.empty.peek().litToBoolean) {
    c.clock.step()
   }

   // Dequeue and check order
   for (expected <- inputs) {
    c.io.pop.poke(true.B)
    c.io.output.expect(expected.U)
    c.clock.step()
   }
  }
 }
}
```

`Arbitrary` and Shrinking

Arbitrary

- A type class used to define how to generate random values
- Default `Arbitrary` instances for common types (e.g., `Int`, `List[Int]`)
- Define custom generators by creating your own `Arbitrary` instances

`Arbitrary` and Shrinking

Arbitrary

- A type class used to define how to generate random values
- Default `Arbitrary` instances for common types (e.g., `Int`, `List[Int]`)
- Define custom generators by creating your own `Arbitrary` instances

Example: Custom `Arbitrary` for Bounded Lists

```
implicit val smallListArb: Arbitrary[List[Int]] =
Arbitrary {
 Gen.listOf(Gen.choose(0, 255)).suchThat(_.length <= 8)
}
```

`Arbitrary` and Shrinking

Arbitrary

- A type class used to define how to generate random values
- Default `Arbitrary` instances for common types (e.g., `Int`, `List[Int]`)
- Define custom generators by creating your own `Arbitrary` instances

Example: Custom `Arbitrary` for Bounded Lists

```
implicit val smallListArb: Arbitrary[List[Int]] =
Arbitrary {
 Gen.listOf(Gen.choose(0, 255)).suchThat(_.length <= 8)
}
```

Shrinking

- Tries to find the **smallest failing input**
- Helps identify minimal failures

Tip: Override default shrinking using `Shrink`

Getting Started with ScalaCheck in Chisel

1. **Add ScalaCheck to `build.sbt`:**

```
libraryDependencies += "org.scalacheck" %% "scalacheck" % "1.19.0" % Test
```

2. **Import Required Packages:**

```
import org.scalacheck._
import org.scalacheck.Prop.forAll
```

3. **Create a ScalaCheck Property Class:**

```
class FifoProps extends Properties("FIFO") with ChiselScalatestTester {
 property("basic enqueue/dequeue") = forAll { (in: Int) =>
  test(new MyFifo(depth = 4)) { c =>
   ...
  }
 }
}
```

4. **Run with `sbt test`**

Continuous Integration (CI): Motivation

Tests are only useful if they are run!

The Problem:

- Developers may forget to run tests
- Different machines = inconsistent environments
- Tests can be time-consuming or complex
- As teams grow, ensuring everyone runs all tests becomes impractical

Continuous Integration (CI): Motivation

Tests are only useful if they are run!

The Problem:

- Developers may forget to run tests
- Different machines = inconsistent environments
- Tests can be time-consuming or complex
- As teams grow, ensuring everyone runs all tests becomes impractical

The Solution:

- Use a CI server to automatically run tests on every change
- Integrate with version control (e.g., GitHub Actions, GitLab CI)
- Enforce test success before merging pull requests

Continuous Integration (CI): Benefits and Extras

Key Benefits:

- Developers don't need to run all tests locally
- Tests run in a clean, consistent environment
- Regressions are caught **before** merging
- Ensures the `master` branch is always in a working state

Additional Advantages:

- Run performance or stress tests regularly
- Automate builds, documentation, and deployments
- Even useful for solo developers to avoid regressions

CI Tools:

- GitHub Actions, GitLab CI/CD, Jenkins, CircleCI, Travis CI

Setting Up GitHub Actions CI for Chisel

1. Create Workflow File:
In your repo, create the file: `.github/workflows/ci.yml`

Setting Up GitHub Actions CI for Chisel

1. Create Workflow File:
In your repo, create the file: `.github/workflows/ci.yml`
2. Example `ci.yml` for Chisel Project:

```
name: CI
  on:
    pull_request:
    push:
  jobs:
    test:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout
        uses: actions/checkout@v4
      - name: Setup JDK
        uses: actions/setup-java@v4
        with:
          distribution: temurin
          java-version: 11
      - name: Setup sbt launcher
        uses: sbt/setup-sbt@v1
      - name: Build and Test
        run: sbt test
```

Setting Up GitHub Actions CI for Chisel

1. Create Workflow File:
In your repo, create the file: `.github/workflows/ci.yml`
2. Example `ci.yml` for Chisel Project:

```
name: CI
  on:
    pull_request:
    push:
  jobs:
    test:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout
        uses: actions/checkout@v4
      - name: Setup JDK
        uses: actions/setup-java@v4
        with:
          distribution: temurin
          java-version: 11
      - name: Setup sbt launcher
        uses: sbt/setup-sbt@v1
      - name: Build and Test
        run: sbt test
```

3. Commit and Push:

- Automatically run on each push or PR
- Check the `Actions` tab for results

Lab session: Priority List

Systolic Array Priority List

- A **sorting structure** that maintains a list of key-value pairs in priority order

Lab session: Priority List

Systolic Array Priority List

- A **sorting structure** that maintains a list of key-value pairs in priority order
- **Push:** Insert a new key and payload at the front of the array
- The key propagates through the array, **bubbling up** to its correct position
- **Pop:** Remove the smallest key and its payload from the end of the array

Lab session: Priority List

Systolic Array Priority List

- A **sorting structure** that maintains a list of key-value pairs in priority order
- **Push:** Insert a new key and payload at the front of the array
- The key propagates through the array, **bubbling up** to its correct position
- **Pop:** Remove the smallest key and its payload from the end of the array
- **Throughput:** One pop every 2 cycles (after initial latency)
- **Applications:** Used in high-speed systems like **network routers**
- **Trade-offs:**
  - Very low latency
  - High hardware cost (area and power)
  - Highly parallel and pipelined

Lab session: Priority List

Systolic Array Priority List

- A **sorting structure** that maintains a list of key-value pairs in priority order
- **Push:** Insert a new key and payload at the front of the array
- The key propagates through the array, **bubbling up** to its correct position
- **Pop:** Remove the smallest key and its payload from the end of the array
- **Throughput:** One pop every 2 cycles (after initial latency)
- **Applications:** Used in high-speed systems like **network routers**
- **Trade-offs:**
  - Very low latency
  - High hardware cost (area and power)
  - Highly parallel and pipelined
- **Configurable:** Key size, payload size, and array length

Commands and Configuration

Supported Commands:

- `IDLE` – Do nothing
- `PUSH` – Insert a key/payload pair into the front of the array
- `POP` – Remove the smallest key/payload pair from the end
- `PUSH/POP` – Simultaneously insert a new pair and remove the smallest

Commands and Configuration

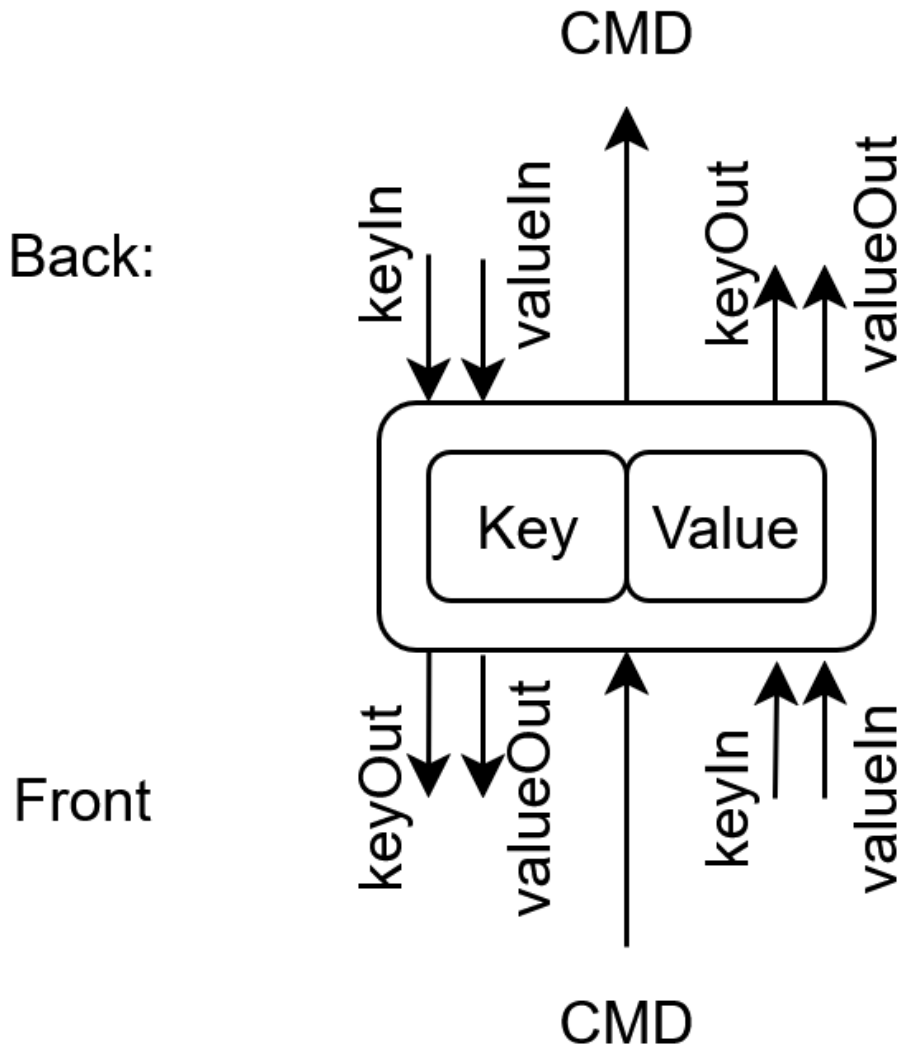
Supported Commands:

- `IDLE` – Do nothing
- `PUSH` – Insert a key/payload pair into the front of the array
- `POP` – Remove the smallest key/payload pair from the end
- `PUSH/POP` – Simultaneously insert a new pair and remove the smallest

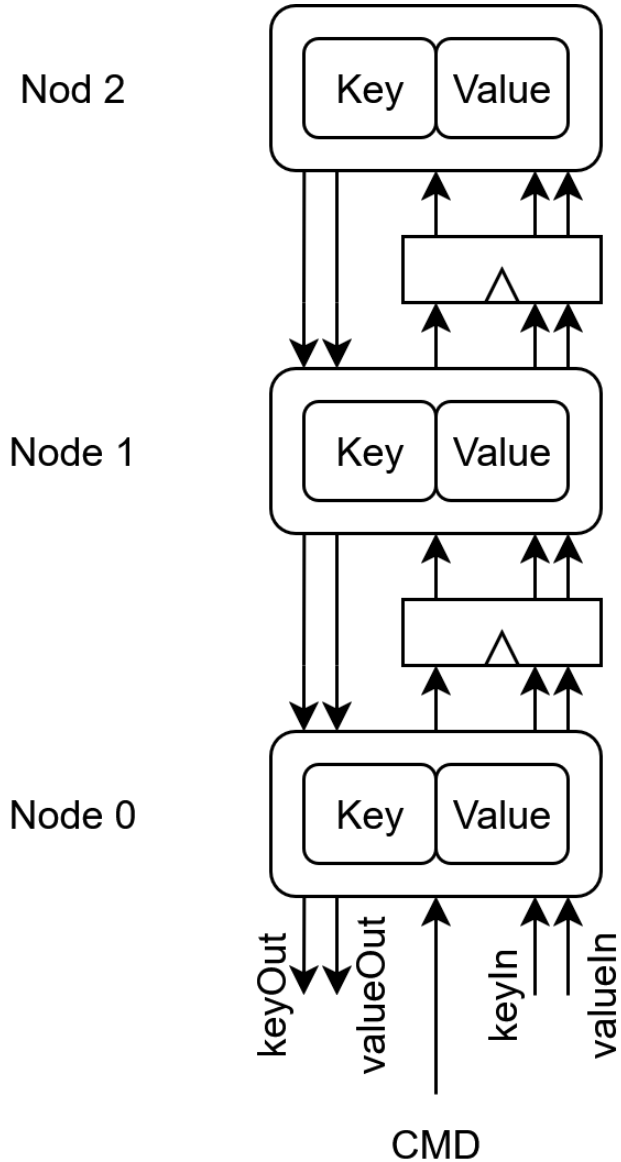
Design Requirements:

- **Configurable parameters:**
  - `keyWidth` – Bit width of the key (used for sorting)
  - `valueWidth` – Bit width of the payload
  - `depth` – Number of stages in the systolic array
- **Modular design:** Each stage compares and forwards data
- **Testable:** Design should support unit and integration testing

Systolic Array Priority List: Node



Systolic Array Priority List



Testing the Priority List

How should this be tested?

Testing the Priority List

How should this be tested?

- **Unit Tests:**
  - Test individual array elements (stages) for correct compare-and-forward behavior
- **Integration Tests:**
  - Validate full array behavior across PUSH/POP operations
  - Ensure keys bubble correctly and throughput matches expectations
- **Property-Based Tests:**
  - **Push-Pop Consistency:** Every pushed value should eventually be popped
  - **Sorted Output:** Popped values should be in ascending key order
  - **Idle Preservation:** IDLE cycles should not lose or corrupt data
  - **Stable Sorting:** Equal keys should preserve insertion order

Extension: Grouped Value Support

New Requirements:

- Values must be tracked in **groups**—each key may have multiple associated values
- Each `PUSH` command may insert a **variable number of values**
- Newly pushed values must **merge** with existing values for the same key
- System should be **configurable** in terms of maximum pair lengths

Extended Commands:

- `Push(0)` – No operation
- `Pop(0)` – Pop lowest key and its value group
- `Push(x)` – Push `x` values with a given key
- `Pop(x)` – Pop lowest key group, then push `x` values with a new key

Summary

- Testing is an integral part of agile development
- Test-driven development aids fast iteration and cooperation
- Property-based testing increases test quality
- Continuous integration ensures code quality is maintained
- Lab session: Priority List