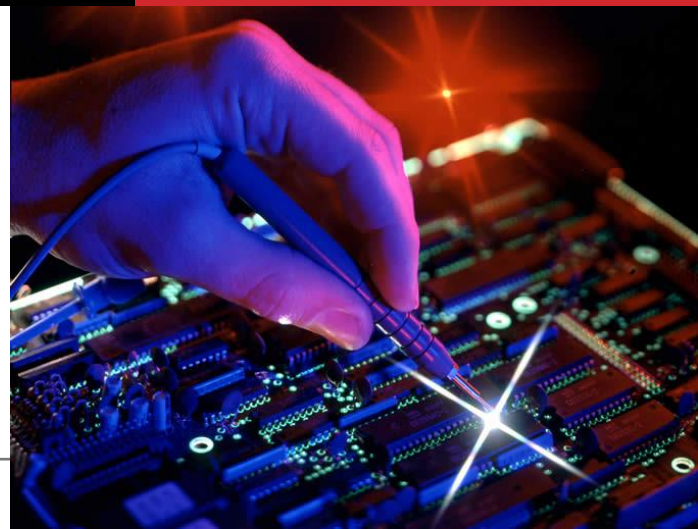
Computer Architecture & Microprocessor System

DIGITAL CIRCUIT VERIFICATION

Dennis A. N. Gookyi

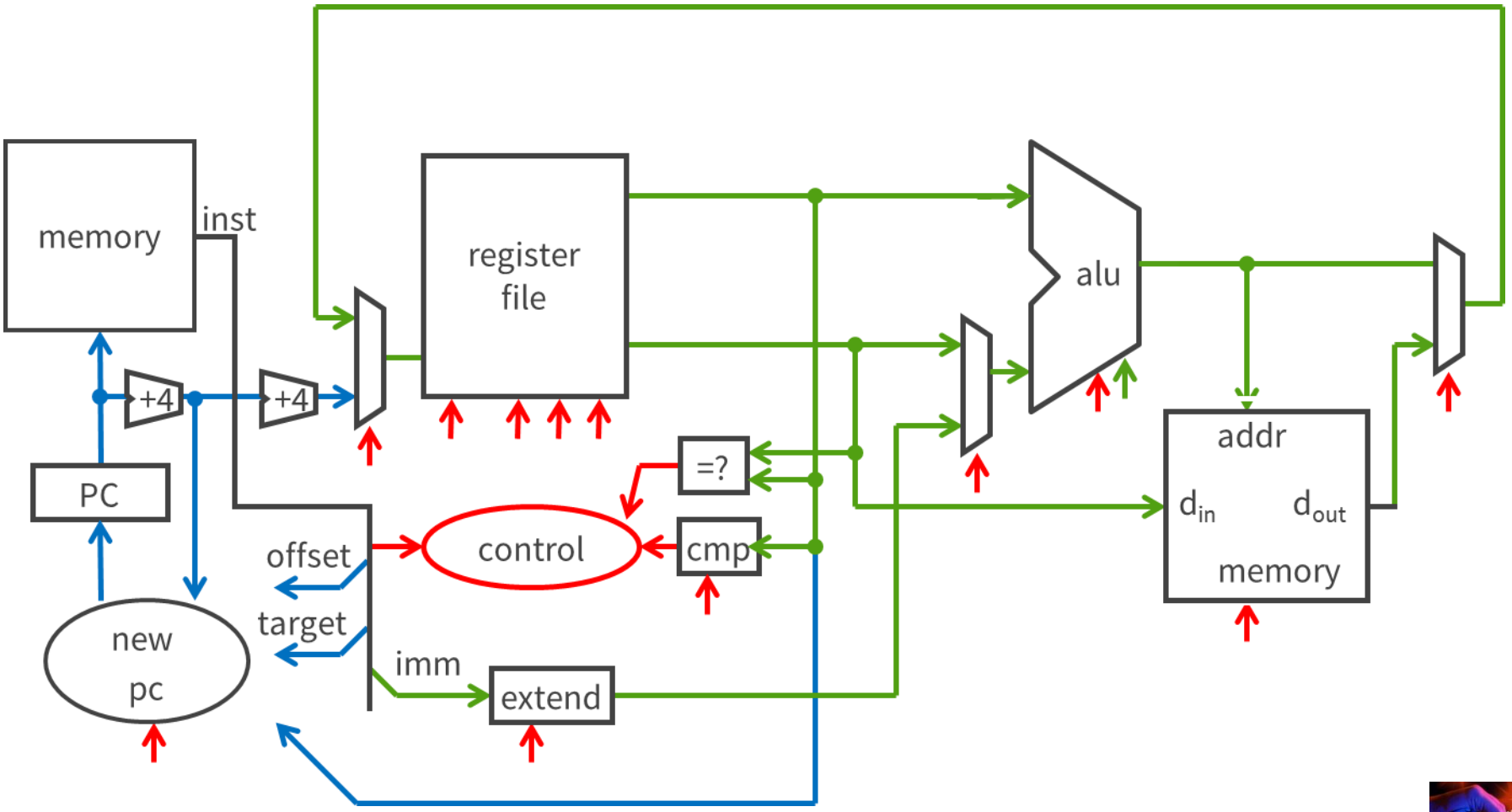


CONTENTS

- DIGITAL CIRCUIT VERIFICATION

BIG PICTURE: BUILDING A PROCESSOR

- Single cycle processor



INTRODUCTION

- How Do You Know That A Circuit Works?
  - You have designed a circuit
    - Is it functionally correct?
    - Even if it is logically correct, does the hardware meet all timing constraints?
  - How can you test for:
    - Functionality?
    - Timing?
  - Answer: simulation tools!
    - Formal verification tools (e.g., SAT solvers)
    - HDL timing simulation (e.g., Vivado)
    - Circuit simulation (e.g., SPICE)

TESTING LARGE DIGITAL DESIGNS

- Testing can be the most time consuming design stage
  - Functional correctness of all logic paths
  - Timing, power, etc. of all circuit elements
- Unfortunately, low-level (e.g., circuit) simulation is much slower than high-level (e.g., HDL, C) simulation
- Solution: we split responsibilities:
  - Check only functionality at a high level (e.g., C, HDL)
    - (Relatively) fast simulation time allows high code coverage
    - Easy to write and run tests
  - Check only timing, power, etc. at low level (e.g., circuit)
    - No functional testing of low-level model
    - Instead, test functional equivalence to high-level model
      - Hard, but easier than testing logical functionality at this level

TESTING LARGE DIGITAL DESIGNS

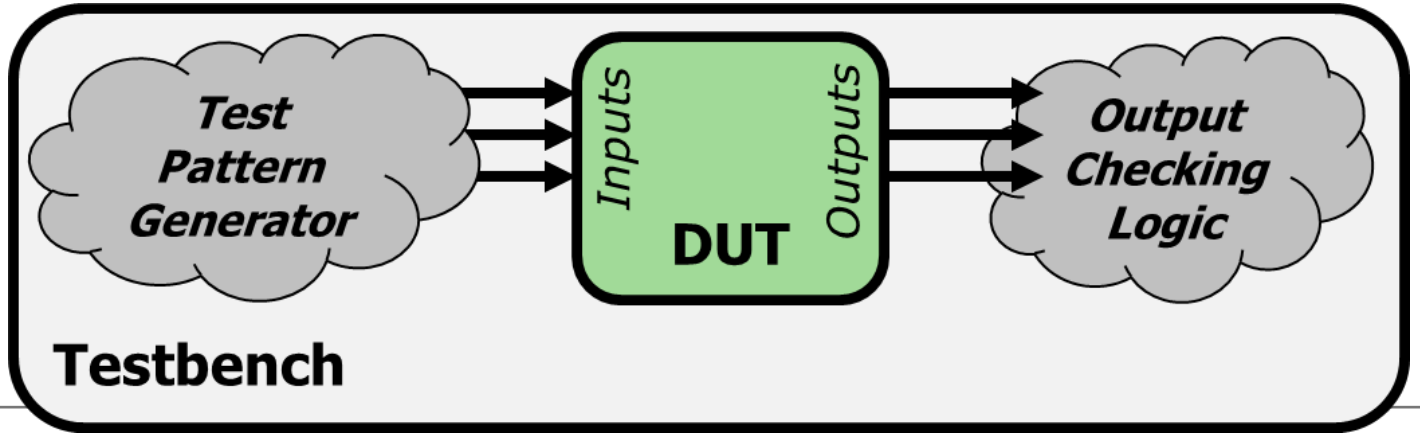
- We have tools to handle different levels of verification
  - Logic synthesis tools guarantee equivalence of high-level logic and synthesized circuit-level description
  - Timing verification tools check all circuit timings
  - Design rule checks ensure that physical circuits are buildable
- The task of a logic designer is to:
  - Provide functional tests for logical correctness of the design
  - Provide timing constraints (e.g., desired operating frequency)
- Tools and/or circuit engineers will decide if it can be built!

FUNCTIONAL VERIFICATION

- Goal: check logical correctness of the design
- Physical circuit timing (e.g., t_{setup}/t_{hold}) is typically ignored
  - May implement simple checks to catch obvious bugs
  - We'll discuss timing verification later in this lecture
- There are two primary approaches
  - Logic simulation (e.g., C/C++/Verilog test routines)
  - Formal verification techniques
- In this course, we will use Verilog for functional verification

TESTBENCH-BASED FUNCTIONAL TESTING

- Testbench: a module created specifically to test a design
  - Tested design is called the "device under test (DUT)"
- Testbench provides inputs (test patterns) to the DUT
  - Hand-crafted values
  - Automatically generated (e.g., sequential or random values)
- Testbench checks outputs of the DUT against:
  - Hand-crafted values
  - A "golden design" that is known to be bug-free



TESTBENCH-BASED FUNCTIONAL TESTING

- A testbench can be:
  - HDL code written to test other HDL modules
  - Circuit schematic used to test other circuit designs

- The testbench is not designed for hardware synthesis
  - Runs in simulation only
    - HDL simulator (e.g., Vivado simulator)
    - SPICE circuit simulation
  - Testbench uses simulation-only constructs
    - E.g., "wait 10ns"
    - E.g., ideal voltage/current source
    - Not suitable to be physically built

TESTBENCH-BASED FUNCTIONAL TESTING

- Example DUT
- We will walk through different types of testbenches to test a module that implements the logic function:

$$y = (\overline{b} \cdot \overline{c}) + (a \cdot \overline{b})$$

```
// performs y = ~b & ~c | a & ~b
module sillyfunction(input  a, b, c,
                     output y);
      wire b_n, c_n;
      wire m1, m2;

      not not_b(b_n, b);
      not not_c(c_n, c);

      and minterm1(m1, b_n, c_n);
      and minterm2(m2, a, b_n);
      or  out_func(y, m1, m2);
endmodule
```

TESTBENCH-BASED FUNCTIONAL TESTING

- Useful Verilog Syntax for Testbenching

```
module example_syntax();
  reg  a;

  // like "always" block, but runs only once at sim start
  initial
  begin
      a = 0; // set value of reg: use blocking assignments
      #10;   // wait (do nothing) for 10 ns
      a = 1;
      $display("printf() style message!"); // print message
   end
endmodule
```


TESTBENCH-BASED FUNCTIONAL TESTING

- Simple Testbench

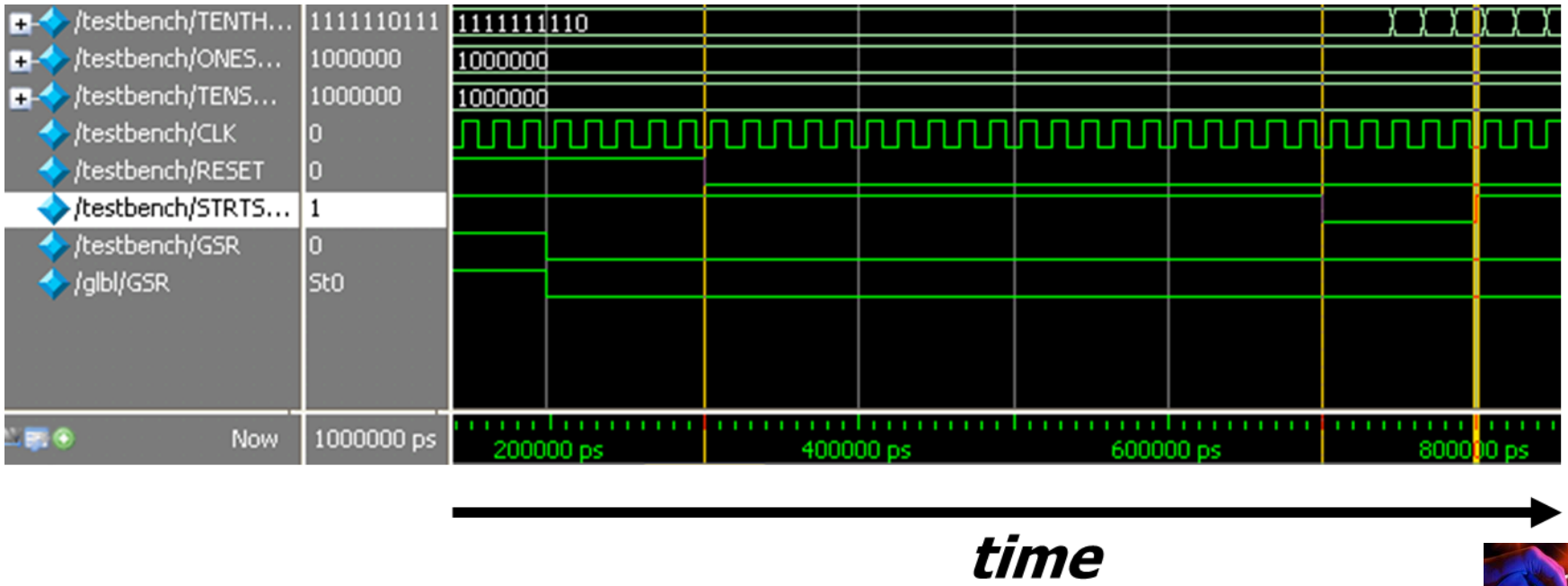
```
module testbench1(); // No inputs, outputs
  reg  a, b, c;      // Manually assigned
  wire y;            // Manually checked

  // instantiate device under test
  sillyfunction dut (.a(a), .b(b), .c(c), .y(y) );

  // apply hardcoded inputs one at a time
  initial begin
    a = 0; b = 0; c = 0; #10; // apply inputs, wait 10ns
    c = 1; #10;               // apply inputs, wait 10ns
    b = 1; c = 0; #10;        // etc .. etc..
    c = 1; #10;
    a = 1; b = 0; c = 0; #10;
  end
endmodule
```


TESTBENCH-BASED FUNCTIONAL TESTING

- Simple Testbench: Output checking
  - Most common method is to look at waveform diagrams
    - Thousands of signals over millions of clock cycles
    - Too many to just printf()
  - Manually check that output is correct at all times



TESTBENCH-BASED FUNCTIONAL TESTING

- Simple Testbench
  - Pros:
    - Easy to design
    - Can easily test a few, specific inputs (e.g., corner cases)
  - Cons:
    - Not scalable to many test cases
    - Outputs must be checked manually outside of the simulation
      - E.g., inspecting dumped waveform signals
      - E.g., printf() style debugging

TESTBENCH-BASED FUNCTIONAL TESTING

- Self-Checking Testbench

```
module testbench2();
  reg  a, b, c;
  wire y;

   sillyfunction dut(.a(a), .b(b), .c(c), .y(y));

   initial begin
      a = 0; b = 0; c = 0; #10; // apply input, wait 10ns
      if (y !== 1) $display("000 failed."); // check result
      c = 1; #10;
      if (y !== 0) $display("001 failed.");
      b = 1; c = 0; #10;
      if (y !== 0) $display("010 failed.");
  end
endmodule
```

TESTBENCH-BASED FUNCTIONAL TESTING

- Self-Checking Testbench
  - Pros:
    - Still easy to design
    - Still easy to test a few, specific inputs (e.g., corner cases)
    - Simulator will print whenever an error occurs
  - Cons:
    - Still not scalable to millions of test cases
    - Easy to make an error in hardcoded values
    - You make just as many errors writing a testbench as actual code
    - Hard to debug whether an issue is in the testbench or in the DUT