Register Transfer Level Design with Verilog

Adapted from Z. Navabi
Portions Copyright Z. Navabi, 2006

2.1 RT Level Design
   2.1.1 Control/data partitioning
   2.1.2 Data part
   2.1.3 Control part

2.2 Elements of Verilog
   2.2.1 Hardware modules
   2.2.2 Primitive instantiations
   2.2.3 Assign statements
   2.2.4 Condition expression
   2.2.5 Procedural blocks
   2.2.6 Module instantiations
Register Transfer Level Design with Verilog

2.3 Component Description in Verilog
   2.3.1 Data components
   2.3.2 Controllers

2.4 Testbenches
   2.4.1 A simple tester
   2.4.2 Tasks and functions

2.5 Summary

RT Level Design

- **RT level design:**
  - Taking a high level description of a design
  - Partitioning
  - Coming up with an architecture
  - Designing the bussing structure
  - Describing and implementing various components of the architecture

- **Steps in RT level design:**
  - Control/Data Partitioning
  - Data Part Design
  - Control Part Design
Control/Data Partitioning

RT Level Design

Data Part

Control/data Partitioning

Data Part

Control Part
Data Part

module DataPath
    (DataInput, DataOutput, Flags, Opcodes, ControlSignals);

    input [15:0] DataInputs;
    output [15:0] DataOutputs;
    output Flags, ...;
    output Opcodes, ...;
    input ControlSignals, ...;

    // instantiation of data components
    // ...
    // interconnection of data components
    // bussing specification
endmodule

* DataPath Module

Output Signals: Going to the control part, provide flags and status of the data

Control Signals: Inputs to data part, sent to the data components and busses

Control Signals for the busses: Select the sources and routing of data from one data component to another
**Data Part**

```
module DataComponent
    (DataIn, DataOut, ControlSignals);

    input [7:0] DataIn;
    output [7:0] DataOut;
    input ControlSignals;
    // Depending on ControlSignals
    // Operate on DataIn and
    // Produce DataOut
endmodule
```

- Partial Verilog Code of a Data Component

---

**Control Part**

RT Level Design

Control/data Partitioning

Data Part

Control Part
Control Part

Consists of one or more state machines to keep the state of the circuit.

Makes decisions as to when and what control signals to issue depending on its state.

---

**module** ControlUnit
(Flags, Opcodes, ExternalControls, ControlSignals);

**input** Flags, ...;
**input** Opcodes, ...;
**Input** ExternalControls, ...;
**output** ControlSignals;
// Based on inputs decide:
// What control signals to issue,
// and what next state to take
**endmodule**

* Outline of a Controller
Elements of Verilog

- We discuss basic constructs of Verilog language for describing a hardware module.
Hardware Modules

Module Instantiations

Primitive Instantiations
Assign Statements
Condition Expression
Procedural Blocks

* Module Specifications
Hardware Modules

- There is more than one way to describe a Module in Verilog.
- May correspond to descriptions at various levels of abstraction or to various levels of detail of the functionality of a module.
- Descriptions of the same module need not behave in exactly the same way nor is it required that all descriptions describe a behavior correctly.
- We discuss basic constructs of Verilog language for a hardware module description.
- We show a small example and several alternative ways to describe it in Verilog.

Primitive Instantiations

- Hardware Modules
- Assign Statements
- Procedural Blocks
- Condition Expression
- Module Instantiations
- Primitive Instantiations
Primitive Instantiations

* A Multiplexer Using Basic Logic Gates

```
module MultiplexerA (input a, b, s, output w);
  wire a_sel, b_sel, s_bar;
  not U1 (s_bar, s);
  and U2 (a_sel, a, s_bar);
  and U3 (b_sel, b, s);
  or  U4 (w, a_sel, b_sel);
endmodule
```

* Primitive Instantiations
Assign Statements

Hardware Modules

Primitive Instantiations

Assign Statements

Condition Expression

Procedural Blocks

Module Instantiations

Assign Statements

Assign Statement and Boolean Expressions to describe the logic

Using Boolean expressions to describe the logic

Continuously drives \( w \) with the right hand side expression

```
module multiplexerB (input a, b, s, output w);
  assign w = (a & ~s) | (b & s);
endmodule
```

` Assign Statement and Boolean `
**Condition Expression**

1. **Hardware Modules**
2. **Primitive Instantiations**
3. **Assign Statements**
4. **Procedural Blocks**
5. **Module Instantiations**

---

**Module Instantiations**

```
module MultiplexerC (input a, b, s, output w);
    assign w = s ? b : a;
endmodule
```

- **Assign Statement and Condition Operator**
  - Can be used when the operation of a unit is too complex to be described by Boolean expressions
  - Very Effective in describing complex functionalities
  - Useful in describing a behavior in a very compact way
**Procedural Blocks**

- **Hardware Modules**
- **Primitive Instantiations**
- **Assign Statements**
- **Condition Expression**
- **Procedural Blocks**
- **Module Instantiations**

---

**Procedural Block Example**

```verilog
module MultiplexerD (input a, b, s, output w);
  reg w;
  always @(a, b, s) begin
    if (s) w = b;
    else w = a;
  end
endmodule
```

- **always statement**
- **Sensitivity list**
- **if-else statement**

**Remarks:**

- Can be used when the operation of a unit is too complex to be described by Boolean or conditional expressions.
Module Instantiations

**Hardware Modules**

- **Primitive Instantiations**
- **Assign Statements**
- **Condition Expression**
- **Procedural Blocks**

**Module Instantiations**

```verilog
module ANDOR (input i1, i2, i3, i4, output y);
    assign y = (i1 & i2) | (i3 & i4);
endmodule

//
module MultiplexerE (input a, b, s, output w);
    wire s_bar;
    not U1 (s_bar, s);
    ANDOR U2 (a, s_bar, s, b, w);
endmodule
```

* Module Instantiation
Module Instantiations

* Multiplexer Using ANDOR

Component Description in Verilog

- Component Description
- Data Components
- Controllers
Data Components

Component Description

Data Components

Controllers

Data Components

Multiplexer

Flip-Flop

Counter

Full-Adder

Shift-Register

ALU

Interconnections
**Multiplexer**

- **Data Components**
  - Multiplexer
  - Flip-Flop
  - Counter
  - Full-Adder
  - Shift-Register
  - ALU

---

```verilog
// Defines a Time Unit of 1 ns and Time Precision of 100 ps.
	timescale 1ns/100ps

module Mux8 (input sel, input [7:0] data1, data0, output [7:0] bus1);
  assign #6 bus1 = (sel ? data1 : data0); // Selects its 8-bit data0 or data1 depending on its sel input.
endmodule

* Octal 2-to-1 MUX

A 6-ns Delay is specified for all values assigned to bus1

---

- **Interconnections**

---
Flip-Flop

Data Components

- Multiplexer
- Counter
- Shift-Register
- Flip-Flop
- Full-Adder
- ALU

Interconnections

---

module Flop (reset, din, clk, qout);
input reset, din, clk;
output qout;
reg qout;
always @ (negedge clk) begin
  if (reset) qout <= #8 1'b0;
  else qout <= #8 din;
end
endmodule

* Flip-Flop Description

- Synchronous Input
- A Non-blocking Assignment
- An 8-ns Delay
- The Body of `always` statement is executed at the negative edge of the `clk` signal
- Flip-Flop triggers on the falling edge of `clk` Input
- A Software-Like Procedural Coding Style
Counters are used in data part for registering data, accessing memory or queues and register stacks.

A 4-bit modulo-16 Counter

A 4-bit Register

Counter Verilog Code

```
module Counter4 (input reset, clk,
output [3:0] count);

reg [3:0] count;

always @(negedge clk) begin
  if (reset) count <= #3 4'1000_00;
  else count <= #5 count + 1;
end
endmodule
```

* Counter Verilog Code

When `count` reaches 1111, the next count taken is 10000

Constant Definition

`timescale 1ns/100ps`
Full-Adder

Data Components

- Multiplexer
- Flip-Flop
- Counter
- Full-Adder
- Shift-Register
- ALU

Interconnections

Full-Adder Verilog Code

```verilog
`timescale 1ns/100ps
module fulladder (input a, b, cin, output sum, cout);
assign sum = a ^ b ^ cin;
assign #3 cout = (a & b) | (a & cin) | (b & cin);
endmodule
```

- Full-Adders are used in data part for building Carry-Chain adders
- A combinational circuit
- All Changes Occur after 5 ns
- One delay for every output: tPLH and tPHL
- All Changes Occur after 3 ns

Full-Adders are used in data part for building Carry-Chain adders
Shift-Register

Data Components

- Multiplexer
- Flip-Flop
- Counter
- Full-Adder
- Shift-Register
- ALU
- Interconnections

### Verilog Code

```verilog
module ShiftRegister8
(input sl, sr, clk,
 input [1:0] m,
 input [7:0] ParIn,
 output [7:0] ParOut);

always @(negedge clk)
begin
  case (m)
    0: ParOut <= ParOut;
    1: ParOut <= {sl, ParOut [7:1]};
    2: ParOut <= {ParOut [6:0], sr};
    3: ParOut <= ParIn;
    default: ParOut <= 8'bX;
  endcase
end
endmodule
```

2 Mode inputs $m[1:0]$ form a 2-bit number

- $m=0$: Does Nothing
- $m=1,2$: Shifts Right and Left
- $m=3$: Loads its Parallel input into the register

An 8-bit Universal Shift Register

Case Statement With 4 case-alternatives and default Value
Shift-Register (Continued)

```verilog
	-timescale 1ns/100ps

module ShiftRegister8
  (input sl, sr, clk, input [7:0] ParIn,
   input [1:0] m, output reg [7:0] ParOut);

always @(negedge clk) begin
  case (m)
    0: ParOut <= ParOut;
    1: ParOut <= {sl, ParOut [7:1]};
    2: ParOut <= {ParOut [6:0], sr};
    3: ParOut <= ParIn;
    default: ParOut <= 8'bX;
  endcase
end
endmodule
```

Shift Right: The $SL$ input is concatenated to the left of $ParOut$.

Shift the $ParOut$ to the left.

---

**ALU**

**Data Components**

- Multiplexer
- Flip-Flop
- Counter
- Full-Adder
- Shift-Register
- **ALU**

**Interconnections**
ALU

```verilog
`timescale 1ns/100ps

module ALU8 (input [7:0] left, right,
             input [1:0] mode,
             output reg [7:0] ALUout);
always @(left, right, mode) begin
    case (mode)
    0: ALUout = left + right;
    1: ALUout = left - right;
    2: ALUout = left & right;
    3: ALUout = left | right;
    default: ALUout = 8'bX;
endcase
end
endmodule
```

- An 8-bit ALU

ALU (Continued)

```verilog
`timescale 1ns/100ps

module ALU8 (input [7:0] left, right,
             input [1:0] mode,
             output reg [7:0] ALUout);
always @(left, right, mode) begin
    case (mode)
    0: ALUout = left + right;
    1: ALUout = left - right;
    2: ALUout = left & right;
    3: ALUout = left | right;
    default: ALUout = 8'bX;
endcase
end
endmodule
```

- An 8-bit ALU
Interconnections

Data Components

- Multiplexer
- Flip-Flop
- Counter
- Full-Adder
- Shift-Register
- ALU

Interconnections

Partial Hardware Using MUX8 and ALU examples forming a Partial Hardware

* Partial Hardware Using MUX8 and ALU
Verilog Code of The Partial Hardware Example

```
Instantiation of ALU8 and MUX8

ALU8 $1 ($left(Inbus), $right(ABinput),
        $mode(function), $ALUout(Outbus));
Mux8 $2 ($sel(select_source), $data1(Aside),
        $data0(Bside), $bus1(ABinput));
```

A Set of parenthesis enclose port connections to the instantiated modules

- $1$ and $2$: Instance Names

Ordered Port Connection

```
ALU8 U1 ( Inbus, ABinput, function, Outbus );
Mux8 U2 ( select_source, Aside, Bside, ABinput );
```

- An Alternative format of port connection

- The actual ports of the instantiated components are excluded

- The list of local signals in the same order as their connecting ports
Controllers

Data Components

Controllers

Component Description

* Controller Outline

- Decisions Based on Inputs, Outputs, State
  - Issue Control Signal
  - Set Next State
  - Go to Next State
Controllers

- Controller:
  - Is wired into data part to control its flow of data.
  - The inputs to it controller determine its next states and outputs.
  - Monitors its inputs and makes decisions as to when and what output signals to assert.
  - Keeps the history of circuit data by switching to appropriate states.
- Two examples to illustrate the features of Verilog for describing state machines:
  - Synchronizer
  - Sequence Detector
Synchronizer

Controllers

Synchronizer

Sequence Detector

- Synchronizing adata

Synchronizer

Clk

adata

synched

* Synchronizing adata
**Synchronizer**

```verilog
`timescale 1ns/100ps

module Synchronizer (input clk, adata,
                     output reg synched);

always @(posedge clk)
  if (adata == 0) synched <= 0;
  else synched <= 1;

endmodule
```

* A Simple Synchronization Circuit

---

**Sequence Detector**

- **Controllers**
- **Synthesizer**
- **Sequence Detector**

If a 1 is Detected on `adata` on the rising edge of clock, `synched` becomes 1 and remains 1 for at least one clock period.
Sequence Detector

Searches on its input for the 110 Sequence

If 110 is detected on a, then w gets 1, else w gets 0.

When the sequence is detected, the w output becomes 1 and stays 1 for a complete clock cycle.

* State Machine Description

Sequence Detector State Machine

A Moore Machine Sequence Detector

States are named: S0, S1, S2, S3

The State in which the 110 sequence is detected.

It Takes at least 3 clock periods to get to the S3 state.

* Sequence Detector State Machine
module Detector110 (input a, clk, reset, output w);
    parameter [1:0] s0=2’b00, s1=2’b01, s2=2’b10, s3=2’b11;
    reg [1:0] current;
    always @(posedge clk)
    if (reset) current = s0;
    else
        case (current)
            s0: if (a) current <= s1; else current <= s0;
            s1: if (a) current <= s2; else current <= s0;
            s2: if (a) current <= s2; else current <= s3;
            s3: if (a) current <= s1; else current <= s0;
        endcase
    assign w = (current == s3) ? 1 : 0;
endmodule

* Verilog Code for 110 Detector
**Sequence Detector**

```verilog
always @(posedge clk) begin
    if (reset) current = s0;
    else
        case (current)
            s0: if (a) current <= s1; else current <= s0;
            s1: if (a) current <= s2; else current <= s0;
            s2: if (a) current <= s2; else current <= s3;
            s3: if (a) current <= s1; else current <= s0;
        endcase;
end
```

* Verilog Code for 110 Detector

At the Absence of a 1 on reset, if-else statement checks for reset.

The 4 Case-alternatives each correspond to a state of state machine.

**Sequence Detector**

- **s1**: if (a) current <= s2; else current <= s0;
- **s2**: if (a) current <= s2; else current <= s3;
- **s3**: if (a) current <= s1; else current <= s0;

* State Transitions on Corresponding Verilog Code
Sequence Detector

```verilog
endmodule

---

* Verilog Code for 110 Detector

Outside of the `always` Block: A combinational circuit

Assigns a 1 to \( w \) output when Machine Reaches to \( s3 \) State

Testbenches

A Simple Tester

Tasks And Functions
A Simple Tester

Testbenches

A Simple Tester

Tasks And Functions

A Simple Tester

 sandals 1ns/100ps

module Detector110Tester;

reg aa, clock, rst;

wire ww;

Detector110 UUT (aa, clock, rst, ww);

initial begin

  aa = 0; clock = 0; rst = 1;

end

initial repeat (44) #7 clock = ~clock;

initial repeat (15) #23 aa = ~aa;

initial begin

  #31 rst = 1;
  #23 rst = 0;

end

always @(ww) if (ww == 1)

$display("A 1 was detected on w at time = %t", $time);

endmodule

* Testbench for Detector110
A Simple Tester

```
module Detector110Tester;
  reg aa, clock, rst;
  wire ww;
  Detector110 UUT (aa, clock, rst, ww);
  ...................
initial begin
  aa = 0; clock = 0; rst = 1;
end
initial repeat (44) #7 clock = ~clock;
initial repeat (15) #23 aa = ~aa;
initial begin
  #31 rst = 1;
  #23 rst = 0;
end
```

* Testbench for Detector110

---

Unsure whether to include this? Discuss.

```
initial begin
  #31 rst = 1;
  #23 rst = 0;
end
```

* Testbench for Detector110

---

Once

```
initial begin
  #31 rst = 1;
  #23 rst = 0;
end
```

* Testbench for Detector110

---

Unsure whether to include this? Discuss.

```
initial begin
  #31 rst = 1;
  #23 rst = 0;
end
```

* Testbench for Detector110

---

Unsure whether to include this? Discuss.

```
initial begin
  #31 rst = 1;
  #23 rst = 0;
end
```

* Testbench for Detector110

---

Unsure whether to include this? Discuss.

```
initial begin
  #31 rst = 1;
  #23 rst = 0;
end
```

* Testbench for Detector110
A Simple Tester

initial begin
  aa = 0; clock = 0; rst = 1;
end

initial repeat (44) #7 clock = ~clock;
initial repeat (25) #23 aa = ~aa;
initial begin
  #31 rst = 1;
  #23 rst = 0;
end

always @((#(w) w == 1))
  $display("A 1 was detected on w at time = %t", $time);

* Testbench for Detector110

Repeats 44 times of complementing the clock input with 7ns delay, generates a periodic signal on clock

For Initial the Input Signals

Signal aa is also assigned a periodic signal, with a different frequency

Waits 31 ns before assigning 1 to rst

This Note Will Appear in the Simulation Environment's Window: "Console" or "Transcript"

A Verilog System Task

Reports the Times at which the w Variable becomes 1

always Block Wakes up when w Changes

A Simple Tester
Tasks And Functions

Testbenches

A Simple Tester

Tasks And Functions

- Verilog Tasks and Functions:
  - System tasks for Input, Output, Display, and Timing Checks
  - User defined tasks and functions
- **Tasks:**
  - Can represent a sub module within a Verilog module
  - Begins with a `task` keyword
  - Its body can only consist of sequential statements like `if-else` and `case`
- **Functions:**
  - Can be used for corresponding to hardware entities
  - May be used for writing structured codes
  - Applications: Representation of Boolean functions, data and code conversion, and input and output formatting
Funky parallelism

- Hardware is inherently parallel
- FPGA = Fine-grained massively parallel computer
- Verilog = Funky parallel programming language

Verilog tips and traps
**Constants: 32 bits, decimal**

- wire [7:0] foo = 127; // synthesis warning!
- wire [7:0] foo = 8’d127;
- wire [7:0] foo = 8’b11111111;
- wire [7:0] foo = 8’hff;
- wire [7:0] foo = 8’hFF;
- watch out: 1010 looks like 4’b1010!

**Truncation**

- wire [7:0] a = 8’hAB;
- wire b; // oops! forgot width
- wire [7:0] c;
- assign b = a; // synthesis warning if lucky.
- assign c = a;
**reg vs. wire**

```verilog
wire f; reg g, h;
assign f = a & b;

always @(posedge clk)
g <= a & b;

always @(*)
h = a & b;
```

---

**blocking) = vs. <= (non-blocking)**

- **Simple rule:**
  - **If you want sequential logic, use**
    ```verilog```
    always @(posedge clk) with <=. (non-blocking)
  ```verilog```
  - **If you want combinational logic, use**
    ```verilog```
    always @(*) with =. (blocking)
(blocking) = vs. <= (non-blocking)

- always @ (posedge clk)
  begin
  f <= a + b;
  g <= f + c;
  end

- always @ (posedge clk)
  begin
  f = a + b;
  g = f + c; // a + b + c
  end

- always @ (posedge clk)
  begin
  f2 <= f1;
  f3 <= f2;
  f4 = f3;
  f5 = f4; // f5 != f3!!
  f7 = f6;
  f6 = f5;
  end

Verilog Stratified Event Queue [1]

- Region 1: Active Events
  - Most events except those explicitly in other regions
  - Includes $display system tasks
- Region 2: Inactive Events
  - Processed after all active events
  - #0 delay events (bad)
- Region 3: Non-blocking Assign Update Events
  - Evaluation previously performed
  - Update is after all active and inactive events complete
- Region 4: Monitor Events
  - Caused by $monitor and $strobe system tasks
- Region 5: Future Events
  - Occurs at some future simulation time
  - Includes future events of other regions
  - Other regions only contain events for CURRENT simulation time
Verilog Stratified Event Queue [2]

- **Active Events**
  - Blocking assignments
  - Evaluate RHS of nonblocking assignments
  - Continuous assignments
  - `#display` command execution
  - Evaluate inputs and change outputs of primitives

- **Inactive Events**
  - #0 blocking assignments

- **Nonblocking Events**
  - Update LHS of nonblocking assignments

- **Monitor Events**
  - `$monitor` command execution
  - `$strobe` command execution
  - Other specific PLI commands

These events may be scheduled in any order within a block, blocking assignments, are in order

---

Incomplete sensitivity lists

- • always @(a or b) // it’s or, not ||
  
  \[
  f = a \& b;
  \]

- • always @(a)
  
  \[
  f = a \& b;
  \]

- • always
  
  \[
  f = a \& b;
  \]

- • Just use always@(*) for combinational logic
Blocking and non-blocking assignments

- **Blocking assignments** (Q = A)
  - Variable is assigned immediately
  - New value is used by subsequent statements
- **Non-blocking assignments** (Q <= A)
  - Variable is assigned after all scheduled statements are executed
  - Value to be assigned is computed but saved for later
- **Example: Swap**

```verilog
always @(posedge CLK) begin
    temp = B;
    B = A;
    A = temp;
end
```

```verilog
reg B, C, D;
always @(posedge clk) begin
    B = A;
    C = B;
    D = C;
end
```
Swap

- The following code executes incorrectly
  - One block executes first
  - Loses previous value of variable

```verilog
always @(posedge CLK) begin
  A = B;
end

always @(posedge CLK) begin
  B = A;
end
```

- Non-blocking assignment fixes this
  - Both blocks are scheduled by posedge CLK

```verilog
always @(posedge CLK) begin
  A <= B;
end

always @(posedge CLK) begin
  B <= A;
end
```

Parallel versus serial execution

- `assign` statements are implicitly parallel
  - “=” means continuous assignment
  - Example
    ```verilog
    assign E = A & D;
    assign A = B & C;
    ```
  - `A` and `E` change if `B` changes

- `always` blocks execute in parallel
  - `always @(posedge clock)`

- Procedural block internals not necessarily parallel
  - “=” is a blocking assignment (sequential)
  - “<=” is a nonblocking assignment (parallel)
  - Examples of procedures: `always, function, etc.`