Register Transfer Level in Verilog: Part I

Lan-Da Van (范倫達), Ph. D.
Department of Computer Science
National Chiao Tung University
Taiwan, R.O.C.
Fall, 2014

ldvan@cs.nctu.edu.tw
http://www.cs.nctu.edu.tw/~ldvan/

Introduction

A digital system is a sequential logic system constructed with flip-flops and gates.

- To specify a large digital system with a state table is very difficult.
- Modular subsystems
- Registers, decoders, multiplexers, arithmetic elements and control logic.
- They are interconnected with datapaths and control signals.

A digital system is represented at the register transfer level (RTL) when it is specified by the following three components:

- The set of registers in the system.
- The operations that are performed on the data stored in the registers.
- The control that supervises the sequence of operations in the system.
R2 ← R1
denotes a transfer of the contents of register R1 into register R2.

A conditional statement governing a register transfer operation is symbolized with an if-then statement such as

\[ \text{If (} T1 = 1 \text{) then (} R2 \leftarrow R1 \text{)} \]

where T1 is a control signal generated in the control section.

\begin{align*}
R1 &\leftarrow R1 + R2 & \text{Add contents of R2 to R1} \\
R3 &\leftarrow R3 + 1 & \text{Increment R3 by 1} \\
R4 &\leftarrow \text{shr} \ R4 & \text{Shift right R4} \\
R5 &\leftarrow 0 & \text{Clear R5 to 0}
\end{align*}
Type of Operations Most Often Encountered in Digital System

- Transfer operations, which transfer data from one register to another.
- Arithmetic operations, which perform arithmetic on data in registers.
- Logic operations, which perform bit manipulation of nonnumeric data in registers.
- Shift operations, which shift data between registers.
Register Transfer Level in HDL

In Verilog, descriptions of **RTL operations** use a combination of **behavioral** and **dataflow** constructs and are employed to specify:

- **register operations** — *procedural* assignment statements within an *edge-sensitive* cycle behavior

- **combinational logic functions** — *continuous* assignment statement or by *procedural* assignment statements within a *level-sensitive* cycle behavior
The following examples show the various ways to specify a register transfer operation in Verilog:

(a) **assign** $S = A + B$;  // **continuous** assignment for addition operation

(b) **always** @ (A, B)    // **level-sensitive** cyclic behavior
    $S = A + B$;  // **combinational logic** for addition operation

(c) **always** @ (negedge clock) // **edge-sensitive** cyclic behavior
    begin
    RA <= RA + RB; // **nonblocking** procedural assignment for addition
    RD <= RA; // **register transfer** operation
    end

The **target operand** in a **continuous** assignment statement (**assign** $S = A + B$) cannot be a register data type, but must be a type of net, for example, **wire**.

There are two kinds of procedural assignments: **blocking** and **nonblocking**.
# HDL Operators

**Table 8.1**

*Verilog 2001 HDL Operators*

<table>
<thead>
<tr>
<th>Operator Type</th>
<th>Symbol</th>
<th>Operation Performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>+</td>
<td>addition</td>
</tr>
<tr>
<td></td>
<td>−</td>
<td>subtraction</td>
</tr>
<tr>
<td></td>
<td>∗</td>
<td>multiplication</td>
</tr>
<tr>
<td></td>
<td>/</td>
<td>division</td>
</tr>
<tr>
<td></td>
<td>%</td>
<td>modulus</td>
</tr>
<tr>
<td></td>
<td>∗∗</td>
<td>exponentiation</td>
</tr>
<tr>
<td>Logic (bitwise reduction)</td>
<td>~</td>
<td>negation (complement)</td>
</tr>
<tr>
<td></td>
<td>&amp;</td>
<td>AND</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Logical</td>
<td>!</td>
<td>negation</td>
</tr>
<tr>
<td></td>
<td>&amp;&amp;</td>
<td>AND</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## HDL Operators

<table>
<thead>
<tr>
<th>Shift</th>
<th>Logical Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;&gt;</td>
<td>logical right shift</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>logical left shift</td>
</tr>
<tr>
<td>&gt;&gt;&gt;</td>
<td>arithmetic right shift</td>
</tr>
<tr>
<td>&lt;&lt;&lt;</td>
<td>arithmetic left shift</td>
</tr>
<tr>
<td>{ , }</td>
<td>concatenation</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Relational</th>
<th>Comparison Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
</tr>
<tr>
<td>==</td>
<td>equality</td>
</tr>
<tr>
<td>!=</td>
<td>inequality</td>
</tr>
<tr>
<td>===</td>
<td>case equality</td>
</tr>
<tr>
<td>!==</td>
<td>case inequality</td>
</tr>
<tr>
<td>=&gt;</td>
<td>greater than or equal</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal</td>
</tr>
</tbody>
</table>
HDL Operators

- The exponentiation operator (***) was added to the language in 2001 and forms a double-precision floating-point value from a base and exponent having a real, integer, or signed value.

- Two types of logic operators for binary words:
  - **bitwise** — a bit-by-bit operation on two vector operands to form a vector result.
  - **reduction** — acting on a single operand and producing a scalar (one-bit) result.
    - the reduction NOR (~|) results in 0 with operand and 00101 and in 1 with operand 00000.
    - Negation: bitwise only.
HDL Operators

- The logical and relational operators are used to form Boolean expressions and can take variables or expressions as operands.
- An operand that is variable evaluates to 0 if the value of the variable is equal to zero and to 1 if the value is *not equal to zero*. 
For example, if $A = \text{1010}$, $B = \text{0000}$, then $A$ has the Boolean value $1$, $B$ has Boolean value $0$. Results of other operation with these value:

- $A \&\& B = 0$ // logical AND
- $A | | B = 1$ // logical OR
- $!A = 0$ // logical complement
- $!B = 1$ // logical complement
- $(A > B) = 1$ // is greater than
- $(A == B) = 0$ // identity (equality)
### Table 8.2
**Verilog Operator Precedence**

<table>
<thead>
<tr>
<th>Operator(s)</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ − ! ~ &amp; ~ &amp;</td>
<td>~</td>
</tr>
<tr>
<td>**</td>
<td></td>
</tr>
<tr>
<td>*/%</td>
<td></td>
</tr>
<tr>
<td>+ − (binary)</td>
<td></td>
</tr>
<tr>
<td>&lt;&lt; &gt;&gt; &lt;&lt;&lt; &gt;&gt;&gt;&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt; &lt;= &gt; &gt;=</td>
<td></td>
</tr>
<tr>
<td>== != === !==</td>
<td></td>
</tr>
<tr>
<td>&amp; (binary)</td>
<td></td>
</tr>
<tr>
<td>^ ^~ ^~ (binary)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(binary)</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>?: (conditional operator)</td>
<td></td>
</tr>
<tr>
<td>{} {} {{}}</td>
<td>Lowest precedence</td>
</tr>
</tbody>
</table>
Loop Statements

- **repeat loop**
  
  ```
  initial
  begin
  clock = 1'b0;
  repeat (16)
    #5 clock =~clock;
  end
  ```

- **forever loop**
  
  ```
  initial
  begin
  clock = 1'b0;
  forever
    #10 clock =~clock;
  end
  ```

- **while loop**
  
  ```
  integer count;
  initial
  begin
  count = 0;
  while (count < 64)
    #5 count = count + 1;
  end
  ```

- **for loop**
  
  ```
  for (j = 0; j < 8; j = j + 1)
  begin
    //procedural statement go here
  end
  ```
Although it is possible to use a `reg` variable to index a loop, sometimes it is more convenient to declare an `integer` variable, rather than a `reg`, for counting purposes.

Variables declared as data type `reg` are stored as unsigned numbers. Those declared as data type `integer` are stored as signed numbers in 2’s-complement format. The default width of an integer is a minimum of 32 bits.
HDL Example 8.1

// Description of 2 × 4 decoder using a for loop statement
module decoder (IN, Y);

input [1: 0] IN; // Two binary inputs
output [3: 0] Y; // Four binary outputs
reg [3: 0] Y; // Control (index) variable for loop
integer k; // Control (index) variable for loop

always @ (IN)
  for (k = 0; k <= 3; k = k + 1)
    if (IN == k) Y[k] = 1;
    else Y[k] = 0;

endmodule
Logic Synthesis

Logic synthesis is the automatic process by which a computer-based program (i.e., a synthesis tool) transforms an HDL model of a logic circuit into an optimized netlist of gates ...

The types of ICs that implement the design may be:
- an application-specific integrated circuit (ASIC),
- a programmable logic device (PLD),
- a field-programmable gate array (FPGA).

Logic synthesis is widely used in industry to design and implement large circuits efficiently, correctly, and rapidly.
The continuous assignment (assign) statement is used to describe combinational circuits.

- (+) → a binary adder with full-adder circuits.
- (−) → a gate-level subtractor consisting of full adders and exclusive-OR gates (Fig. 4.13).
- A statement with a conditional operator such as

  \[
  \text{assign } Y = S \ ? \ In_1 : In_0;
  \]

  translates into a two-to-one-line multiplexer with control input \(S\) and data input \(In_1\) and \(In_0\).
Logic Synthesis

A cyclic behavior (always ...) may imply a combinational or sequential circuit, depending on whether the event control expression is level sensitive or edge sensitive.

- For example,

```
always @ (ln_1 or ln_0 or S)
if (S) Y = ln _ 1;
else Y = ln _ 0;
```

translates into a two-to-one-line multiplexer.

- An edge-sensitive cyclic behavior (e.g., always @ (posedge clock)) specifies a synchronous (clocked) sequential circuit. Examples of such circuits are registers and counters.
Logic Synthesis

- A sequential circuit description with a **case** statement translates into a control circuit with **D flip-flops** and **gates** that form the inputs to the flip-flops.
  
  each statement ↔ a gate/flip-flop circuit

- For **synthesizable** sequential circuits, the event control expression must be sensitive to the **positive or negative edge** of the clock (synchronizing signal), but not to both.
Fig. 8.1 A simplified flowchart for HDL-based modeling, verification, and synthesis
The logic design of digital system can be divided into two distinct parts.

- One is concerned with the design of the digital circuits that perform the data-processing operations.
- The other is concerned with the design of the control circuits that determine the sequence in which the various actions are performed.
Algorithmic State Machines (ASMs)

The control logic that generates the signals for sequencing the operations in the data path unit is a finite state machine (FSM).

A flowchart that has been developed specifically to define digital hardware algorithm is called an algorithmic state machine (ASM) chart.

The ASM chart is composed of three basic elements:

- State box
- Decision box
- Conditional box

They connected by directed edges indicating the sequential precedence and evolution of the states as the machine operates.
ASM Chart State Box

(a) Binary code

State name
Moore-type
output signals, register operations

(b) 0101

$S_{\text{pause}}$
$R \leftarrow 0$

Start.OP.A

FIGURE 8.3 ASM chart state box

FIGURE 8.4 ASM chart decision box
ASM Chart Conditional Box

- **State name**: Moore-type output signals
- **Unconditional register operations**
- **Condition**: Conditional (Mealy) outputs and register operations
- **Binary code**

(a) Diagram:
- **Reset_b**: 001
- **S_1**: Start
- **Flag**: 0
- **R**: 0
- **F**: G
- **S_2**: Load_F_G
- **S_3**: 100

(b) Diagram:
- **Reset_b**: 001
- **S_1**: Start
- **Flag**: 0
- **R**: 0
- **S_3**: 100

(c) Diagram:
- **Reset_b**: 001
- **S_1**: Start
- **Flag**: 0
- **Flush_R**: 0
- **S_3**: 100
FIGURE 8.6 ASM block
Simplifications

\[ EF = 00 \]

\[ E = 1 \]

\[ EF = 01 \]

\[ 010 \]

\[ 011 \]

\[ 100 \]

**FIGURE 8.7**
State diagram equivalent to the ASM chart of Fig. 8.6
Timing Considerations

The timing for all register and flip-flop in digital system is controlled by a master-clock generator.

**FIGURE 8.8 Transition between states**
ASMD Chart

Contrasted between Algorithmic State Machine and Datapath (ASMD) charts & ASM charts.

- An ASMD chart does not list register operations within a state box.
- The edges of an ASMD charts are annotated with register operations that are concurrent with the state transition indicated by the edge.
- An ASMD chart includes conditional boxes identifying the signals which control the register operations that annotate the edges of the chart.
- An ASMD chart associates register operations with state transitions rather than with state.
Three-step To Design an ASMD Chart

- Step 1: Form an ASM chart displaying only how the inputs to the controller determine its state transitions.
- Step 2: Convert the ASM chart to an ASMD chart by annotating the edges of ASM chart to indicate to the concurrent register operations of the datapath unit.
- Step 3: Modify the ASMD chart to identify the control signals that are generated by the controller and use the indicated register operations in the datapath unit.
Design Example

- **Block diagram:**
  - **A**: a 4-bit binary counter
  - **E** and **F**: JK flip-flops

- **Sequence of operations:**
  - If $A_2 = 0$, $E$ is cleared to 0 and the count continues.
  - If $A_2 = 1$, $E$ is set to 1; then if $A_3 = 0$, then the count continues, but if $A_3 = 1$, $F$ is set to 1 on the next clock pulse and the system stops counting.
  - Then, if $Start = 0$, the system remains in the initial state, but if $Start = 1$, the operation cycle repeats.
Design Example

Status signals

Controller

Datapath

Start

reset_b

clock

A3

A2

clr_E

set_E

set_F

clr_A_F

incr_A

Datapath

A

E

F

Digital Circuit Lab
Lecture 10
Digital Circuits
Design Example

### Design Example Diagram

- **Reset_b**
- **Start**
  - \( A \leq 0 \)
  - \( F \leq 0 \)
- **clr_A_F**
- **S_idle**
- **incr_A**
  - \( A \leq A + 1 \)
  - \( E \leq 1 \)
- **set_E**
- **A2**
  - \( A \leq A + 1 \)
  - \( E \leq 1 \)
- **set_F**

### Table 8.3: Sequence of Operations for Design Example

<table>
<thead>
<tr>
<th>State</th>
<th>A2</th>
<th>A3</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_idle</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>S_2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>S_1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A2 = 1, A3 = 1 )</td>
</tr>
<tr>
<td>( A2 = 0, A3 = 0 )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>A3</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Flip-Flops</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
</tr>
<tr>
<td>----</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

Digital Circuits
Design Example

Datapath and controller
Design Example

\[
\begin{align*}
S_{\text{idle}} & \rightarrow S_1, \text{clr}_A F: \\
S_1 & \rightarrow S_1, \text{incr}_A: \\
& \quad \text{if } (A_2 = 1) \text{ then set}_E: \\
& \quad \text{if } (A_2 = 0) \text{ then clr}_E: \\
S_2 & \rightarrow S_{\text{idle}}, \text{set}_F:
\end{align*}
\]

Start = 0

\begin{align*}
S_{\text{idle}} & \rightarrow S_1, \text{clr}_A F: \\
S_1 & \rightarrow S_1, \text{incr}_A: \\
& \quad \text{if } (A_2 = 1) \text{ then set}_E: \\
& \quad \text{if } (A_2 = 0) \text{ then clr}_E: \\
S_2 & \rightarrow S_{\text{idle}}, \text{set}_F:
\end{align*}

Start = 1

\begin{align*}
A_2 & = 0 \\
A_2A_3 & = 11
\end{align*}

(a)

\begin{align*}
A_2A_3 & = 10 \\
A & \leftarrow 0, F \leftarrow 0 \\
A & \leftarrow A + 1 \\
E & \leftarrow 1 \\
E & \leftarrow 0 \\
F & \leftarrow 1
\end{align*}

(b)

(ASMD chart)

(state diagram)
## Design Example

### State Table for the Controller of Fig. 8.10

<table>
<thead>
<tr>
<th>Present-State Symbol</th>
<th>Present State</th>
<th>Inputs</th>
<th>Next State</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>set_E</td>
</tr>
<tr>
<td>S_idle</td>
<td>G1 0, G0 0</td>
<td>Start 0, A2 X, A3 X</td>
<td>G1 0, G0 0</td>
<td>0 0</td>
</tr>
<tr>
<td>S_idle</td>
<td>G1 0, G0 0</td>
<td>Start 1, A2 X, A3 X</td>
<td>G1 0, G0 1</td>
<td>0 0</td>
</tr>
<tr>
<td>S_1</td>
<td>G1 0, G0 1</td>
<td>Start X, A2 0, A3 X</td>
<td>G1 0, G0 1</td>
<td>0 0</td>
</tr>
<tr>
<td>S_1</td>
<td>G1 0, G0 1</td>
<td>Start X, A2 1, A3 0</td>
<td>G1 0, G0 1</td>
<td>0 0</td>
</tr>
<tr>
<td>S_1</td>
<td>G1 0, G0 1</td>
<td>Start X, A2 1, A3 1</td>
<td>G1 1, G0 1</td>
<td>0 0</td>
</tr>
<tr>
<td>S_2</td>
<td>G1 1, G0 1</td>
<td>Start X, A2 X, A3 X</td>
<td>G1 0, G0 0</td>
<td>0 0</td>
</tr>
</tbody>
</table>

- **Start** indicates the initial state.
- **A2A3** represent the next state conditions.
- **G1** and **G0** are the outputs for the controller.
- **set_E**, **clr_E**, **set_F**, **clr_A_F**, and **incr_A** are the output controls for the controller.
Design Example

- By inspection:

\[ D_{G0} = \text{Start } S_{\text{idle}} + S_{\text{1}} \]
\[ D_{G1} = S_{\text{1}} A_2 A_3 = \]

**State Table for the Controller of Fig. 8.10**

<table>
<thead>
<tr>
<th>Present-State Symbol</th>
<th>Present State</th>
<th>Inputs</th>
<th>Next State</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>set_E</td>
</tr>
<tr>
<td>( S_{\text{idle}} )</td>
<td>0 0 0</td>
<td>0 X X</td>
<td>0 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>( S_{\text{idle}} )</td>
<td>0 0 0</td>
<td>1 X X</td>
<td>0 1</td>
<td>0 0 0</td>
</tr>
<tr>
<td>( S_{\text{1}} )</td>
<td>0 1 1</td>
<td>X 0 X</td>
<td>0 1</td>
<td>0 0 1</td>
</tr>
<tr>
<td>( S_{\text{1}} )</td>
<td>0 1 1</td>
<td>X 1 0</td>
<td>0 1</td>
<td>0 0 1</td>
</tr>
<tr>
<td>( S_{\text{2}} )</td>
<td>0 1 1</td>
<td>X 1 1</td>
<td>1 1</td>
<td>0 0 1</td>
</tr>
</tbody>
</table>
Design Example

- By inspection:

(State 10 is not used) →

\[
egin{align*}
set_E &= S_1 A_2 \\
clr_E &= S_1 A_2' \\
set_F &= S_2 = G_1 \\
clr_A_F &= Start S_{idle} = Start G_0 \\
incr_A &= S_1
\end{align*}
\]

State Table for the Controller of Fig. 8.10

<table>
<thead>
<tr>
<th>Present-State Symbol</th>
<th>Present State</th>
<th>Inputs</th>
<th>Next State</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S_idle</td>
<td>0 0</td>
<td>0</td>
<td>X X</td>
<td>G1 0</td>
</tr>
<tr>
<td>S_idle</td>
<td>0 0</td>
<td>1</td>
<td>X X</td>
<td>G1 1</td>
</tr>
<tr>
<td>S_1</td>
<td>0 1</td>
<td>X</td>
<td>0 X</td>
<td>G1 1</td>
</tr>
<tr>
<td>S_1</td>
<td>0 1</td>
<td>X</td>
<td>1 0</td>
<td>G1 1</td>
</tr>
<tr>
<td>S_1</td>
<td>0 1</td>
<td>X</td>
<td>1 X</td>
<td>G1 1</td>
</tr>
<tr>
<td>S_2</td>
<td>1 1</td>
<td>X</td>
<td>X X</td>
<td>G1 0</td>
</tr>
</tbody>
</table>
Design Example

logic diagram of control unit

\[ \text{set}_E = S_1 \ A_2 \]
\[ \text{clr}_E = S_1 \ A_2' \]
\[ \text{set}_F = S_2 = G_1 \]
\[ \text{clr}_A_F = \text{Start} \ S_{idle} = \text{Start} \ G_0 \]
\[ \text{incr}_A = S_1 \]
HDL Description of Design Example

- **Structural** ← **sufficient experience**
  - The lowest and most detailed level
  - Specified in terms of physical components and their interconnection

- **RTL**
  - Imply a certain hardware configuration
  - Specified in terms of the registers, operations performed, and control that sequences the operations.

- **Algorithmic-based behavioral**
  - The most abstract
  - Most appropriate for simulating complex systems to verify design ideas and explore tradeoffs
The manual method of design developed

- A block diagram (Fig. 8.9(a)) showing the interface between the datapath and the controller.
- An ASMD chart for the system. (Fig. 8.9(d))
- The logic equations for the inputs to the flip-flops of the controller.
- A circuit that implements the controller (Fig. 8.12).

In contrast, an RTL model describes (1) state transitions of the controller and (2) operations of the datapath as a step towards automatically synthesizing the circuit that implements them.
RTL Description

HDL Example 8.2

// RTL description of design example (see Fig. 8.11)
module Design_Example_RTL (A, E, F, Start, clock, reset_b);

// Specify ports of the top-level module of the design
// See block diagram, Fig. 8.10
output [3: 0] A;
output E, F;
input Start, clock, reset_b;

// Instantiate controller and datapath units
Controller_RTL M0 (set_E, clr_E, set_F, clr_A_F, incr_A, A[2], A[3], Start, clock, reset_b);
Datapath_RTL M1 (A, E, F, set_E, clr_E, set_F, clr_A_F, incr_A, clock);
endmodule

module Controller_RTL (set_E, clr_E, set_F, clr_A_F, incr_A, A2, A3, Start, clock, reset_b);
output reg set_E, clr_E, set_F, clr_A_F, incr_A;
input Start, A2, A3, clock, reset_b;
reg [1: 0] state, next_state;
parameter S_idle = 2'b00, S_1 = 2'b01, S_2 = 2'b11; // State codes
RTL Description

```vhls
always @ (posedge clock or negedge reset_b) // State transitions (edge sensitive)
  if (reset_b == 0) state <= S_idle;
  else state <= next_state;

// Code next-state logic directly from ASM chart (Fig. 8.9d)
always @ (state, Start, A2, A3) begin // Next-state logic (level sensitive)
  next_state = S_idle; ← (forget to make an assignment is OK)
  case (state)
    S_idle: if (Start) next_state = S_1; else next_state = S_idle;
    S_1: if (A2 & A3) next_state = S_2; else next_state = S_1;
    S_2: next_state = S_idle;
    default: next_state = S_idle; ← (one of the above three not detected)
  endcase
end

// Code output logic directly from ASM chart (Fig. 8.9d)
alwayss @ (state, Start, A2) begin
  set_E = 0; // default assignments; assign by exception
  clr_E = 0;
  set_F = 0;
  clr_A_F = 0;
  incr_A = 0;
```

Digital Circuit Lab
RTL Description

```verbatim
case (state)
    S_idle:    if (Start) next_state = S_1; else next_state = S_idle;
    S_1:      if (A2 & A3) next_state = S_2; else next_state = S_1;
    S_2:      next_state = S_idle;
    default:  next_state = S_idle;
endcase
end

// Code output logic directly from ASM chart (Fig. 8.9d)
always @ (state, Start, A2) begin
    set_E   = 0;  // default assignments; assign by exception
    clr_E   = 0;
    set_F   = 0;
    clr_A_F = 0;
    incr_A  = 0;
    case (state)
        S_idle:    if (Start) clr_A_F = 1;
                    begin incr_A = 1; if (A2) set_E = 1; else clr_E = 1; end
        S_1:      set_F = 1;
        S_2:      endcase
    end
endmodule
```
module Datapath_RTL (A, E, F, set_E, clr_E, set_F, clr_A_F, incr_A, clock);
    output reg [3: 0] A; // register for counter
    output reg E, F; // flags
    input set_E, clr_E, set_F, clr_A_F, incr_A, clock;

    // Code register transfer operations directly from ASMD chart (Fig. 8.9(d))
    always @ (posedge clock) begin
        if (set_E) E <= 1;
        if (clr_E) E <= 0;
        if (set_F) F <= 1;
        if (clr_A_F) begin A <= 0; F <= 0; end
        if (incr_A) A <= A + 1;
    end
endmodule
Testing the Design Description

HDL Example 8.3

// Test bench for design example
module t_Design_ExampleRTL;
  reg Start, clock, reset_b;
  wire [3: 0] A;
  wire E, F;
// Instantiate design example
Design_ExampleRTL M0 (A, E, F, Start, clock, reset_b);
// Describe stimulus waveforms
initial #500 $finish; // Stopwatch
initial
  begin
    reset_b = 0;
    Start = 0;
    clock = 0;
    #5 reset_b = 1; Start = 1;
    repeat (32)
      begin
        #5 clock = ~ clock; // Clock generator
      end 
  end
Testing the Design Description

```
initial
$monitor("A = %b E = %b F = %b time = %0d", A, E, F, $time);
endmodule
```

Simulation log:
A = xxxx E = x F = x time = 0
A = 0000 E = x F = 0 time = 10
A = 0001 E = 0 F = 0 time = 20
A = 0010 E = 0 F = 0 time = 30
A = 0011 E = 0 F = 0 time = 40
A = 0100 E = 0 F = 0 time = 50
A = 0101 E = 1 F = 0 time = 60
A = 0110 E = 1 F = 0 time = 70
A = 0111 E = 1 F = 0 time = 80
A = 1000 E = 1 F = 0 time = 90
A = 1001 E = 0 F = 0 time = 100
A = 1010 E = 0 F = 0 time = 110
A = 1011 E = 0 F = 0 time = 120
A = 1100 E = 0 F = 0 time = 130
A = 1101 E = 1 F = 0 time = 140
A = 1101 E = 1 F = 1 time = 150
A = 0000 E = 1 F = 0 time = 160
Testing the Design Description

Name (in groups) 0 50 100 150

clock
reset_b
Start
A2
A3
state[1: 0] (recommended)
cla_A_F
set_E
clear_E
set_F
incr_A
A[3: 0] x 0 1 2 3 4 5 6 7 8 9 a b c d 0
E
F

(Table 8.3)
HDL Example 8.4 presents the structural description of the design example. It consists of a nested hierarchy of modules and gates describing:

- The top-level module, *Design_example_STR*
- The modules describing the controller and the datapath
- The modules describing the flip-flops and counters
- **Gates** implementing the logic of controller
The **top-level** module (see Fig. 8.10) encapsulates the entire design by:

- Instantiating the *controller* and the *datapath* modules
- Declaring the *primary (external) input signals*
- Declaring the *output signals*
- Declaring the *control signals* generated by the controller and connected to the datapath unit
- Declaring the *status signals* generated by the datapath unit and connected to the controller
HDL Example 8.4

```verilog
module Design_Example_STR
   input Start, clock, reset_b);

Controller_STR M0 (clr_A_F, set_E, clr_E, set_F, incr_A, Start, A[2], A[3], clock,
  reset_b);
Datapath_STR M1 (A, E, F, clr_A_F, set_E, clr_E, set_F, incr_A, clock);
endmodule

module Controller_STR
  (output clr_A_F, set_E, clr_E, set_F, incr_A,
   input Start, A2, A3, clock, reset_b);

wire G0, G1;
parameter S_idle = 2'b00, S_1 = 2'b01, S_2 = 2'b11;
wire w1, w2, w3;
```
not (G0_b, G0);
not (G1_b, G1);
buf (incr_A, w2);
buf (set_F, G1);
not (A2_b, A2);
or (D_G0, w1, w2);
and (w1, Start, G0_b);
and (clr_A_F, G0_b, Start);
and (w2, G0, G1_b);
and (set_E, w2, A2);
and (clr_E, w2, A2_b);
and (D_G1, w3, w2);
and (w3, A2, A3);
D_flip_flop_AR M0 (G0, D_G0, clock, reset_b);
D_flip_flop_AR M1 (G1, D_G1, clock, reset_b);
endmodule

// datapath unit

module Datapath_STR
(output [3: 0] A,
 output E, F,
 input clr_A_F, set_E, clr_E, set_F, incr_A, clock
);

Digital Circuit Lab

Digital Circuits
JK_flip_flop_2 M0 (E, E_b, set_E, clr_E, clock);
JK_flip_flop_2 M1 (F, F_b, set_F, clr_A_F, clock);
Counter_4 M2 (A, incr_A, clr_A_F, clock);
endmodule

// Counter with synchronous clear

module Counter_4 (output reg [3: 0] A, input incr, clear, clock);
    always @ (posedge clock)
        if (clear)  A <= 0; else if (incr) A <= A + 1;
endmodule

module D_flip_flop_AR (Q, D, CLK, RST);
    output Q;
    input D, CLK, RST;
    reg Q;

    always @ (posedge CLK, negedge RST)
        if (RST == 0) Q <= 1'b0;
        else Q <= D;
endmodule
// Description of JK flip-flop

module JK_flip_flop_2 (Q, Q_not, J, K, CLK);
output Q, Q_not;
input J, K, CLK;
reg Q;
assign Q_not = ~Q;
always @ (posedge CLK)
  case (J, K)
    2'b00: Q <= Q;
    2'b01: Q <= 1'b0;
    2'b10: Q <= 1'b1;
    2'b11: Q <= ~Q;
  endcase
endmodule

module t_Design_Example_STR;
  reg Start, clock, reset_b;
  wire [3: 0] A;
  wire E, F;

  // Instantiate design example
Design_Example_STR M0 (A, E, F, Start, clock, reset_b);

// Describe stimulus waveforms

initial #500 $finish;               // Stopwatch
initial
begin
  reset_b = 0;
  Start = 0;
  clock = 0;
  #5 reset_b = 1; Start = 1;
repeat (32)
begin
  #5 clock = ~ clock;               // Clock generator
end
end
initial
$monitor ("A = %b E = %b F = %b time = %0d", A, E, F, $time);
endmodule