



RTL Semantics

DRAFT SPECIFICATION

Written by the Accellera C/C++ Working Group of the Architectural Language Committee

Last Modification: February 13th, 2001

Revision: Version 0.8



Motivation

In the past, the EDA industry and designers have struggled with the issues of having multiple languages in use for describing, implementing and verifying their designs, such as Verilog and VHDL. This has led to gross inefficiencies in the industry with tool vendors needing to support multiple languages, which are often dissimilar, and in some cases contradictory, and with users having to deal with incompatible library issues. With the industry embarking on the search for new system level languages we already have several languages based on C or C++ that are emerging and the distinct possibility is arising that we will again be faced with language “wars”. In order to prevent this we need to ensure a minimum level of compatibility between them so that it can guaranteed that information could be moved from one language to another without loss of information. It is for this reason that this working group was formed and which the attached document is aimed.

All languages have two major parts. The first of these is the syntax. This is the actual language constructs that are presented to the user. For example this document is written using a specific syntax which defines the character set and the construction rules. This syntax is actually almost completely common to many languages such as English, French, German etc. However, this syntax does not tell us how to interpret what has been written. In order to do this we need the second component of the language, which is its semantics. Semantics contains the rules of interpretation. It allows us to know what is meant by the words. It is in fact possible to write many different syntaxes on top of a set of semantics and it is this fact that provides the basis for this work. In our language analogy, when we speak we are using the same semantic set as in the written form above, but have now substituted a different syntax, which is speech. What we set out to accomplish in this document is a set of semantic definitions that could be used to bind together all of the emerging system level languages. This still allows the language vendors to compete by creating the best tools and syntax for specific functions. To continue our analogy further, when we have a common semantic set and multiple syntaxes, we can choose which ever works best for a particular situation. For example in some cases it is better to distribute a document like this in text form. Other times speech works well, however, we know that we can always translate by reading the text out loud and there is no loss of content.

In the early formative stages of this group, we looked extensively at how these languages may be integrated into design flows. Several points were identified in these flows as being the points at which a user may need to transfer design information from one tool set to another. These are the target points where we need to ensure that the languages concur with the meaning of what is being represented. As the first step we decided to concentrate on the lowest level of transfer which is at the RTL level. This is a point where many existing tools are established in design flows and displacement of these may not be desired. This document is the result of that work and it is hoped will act as a point of unification amongst the providers of these languages. In fact this work is the result of many of those firms who are committed to making this happen.



Review Procedure:

This document is currently in an open review period where anybody is free to examine and to provide feedback on the document. We are doing this because of the importance of this work for the industry, and we want to make sure that everyone in the industry has the ability to ensure compatibility with the systems out there that we may not have considered. All feedback should be sent to brian_bailey@mentor.com and should be received no later than April 13th 2001.

Feedback should be limited to technical issues since it is known that there are many wording and grammatical errors that need to be fixed. This will be done during an extensive reformatting process before the formal release of the document. For all feedback to be considered, it must contain the following:

Full name, affiliation and valid contact information including postal address and phone number. A description of the identified problem and wherever possible the way that we should change the document in order for the reported problem to be solved.

We thank you for your help.



Contributors:

This document is the result of a lot of work, most of which was contributed by a few key members of the group. Those contributors are marked with an asterisk in member list below for the working group.

Co-Chairs

- * Brian Bailey - Mentor Graphics
- * Dan Gajski – UC Irvine

Members:

- * Martin Baynes – Previously of C Level Design
- Yuri Panchul - C Level Design
- Grant Martin - Cadence
- Simon Davidmann - CoDesign Automation
- Peter Flake - CoDesign Automation
- Paula Menzigian – Coware
- * John Sanguinetti - CynApps
- * Dave Springer – CynApps
- Guy Bois - Ecole Polytechnique de Montreal
- Duncan Gurley - Fujitsu WWSLT
- Kamal Hasmi – SpiraTech Ltd
- Vassilios Gerousis – Infineon
- * Asa Ben-Tzur - Intel Corp
- Dennis Brophy - Mentor Graphics
- Andrew Guyler - Mentor Graphics
- Shrenik Mehta - Sun Microsystems
- Charles Cheng - Sun Microsystems
- Wolfgang Nebel - OFFIS / Oldenburg University
- * Kevin Kranen – Synopsys
- Frederic Doucet - UC Irvine
- Filip Thoen – Virtio

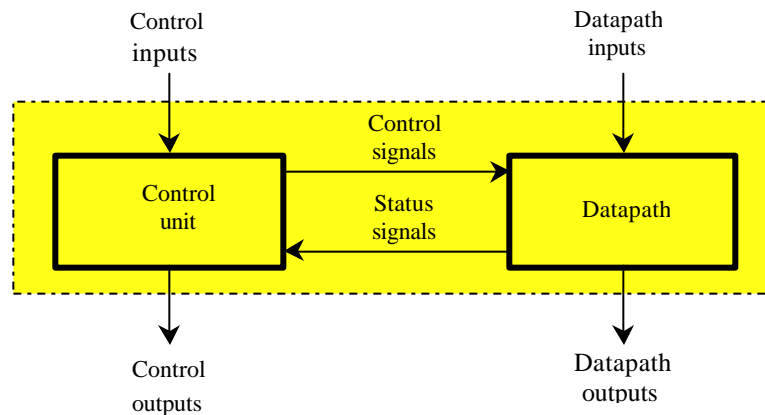
INTRODUCTION

This report is intended to relate RTL theory and practice with RTL modeling and synthesis/simulation issues. RTL semantics must be based on simple implementable models. Semantics defines what RTL model means, which in turn is defined by how RTL design is implemented. In this report we start with generic RTL implementation, called RTL processor, then derive the formal model for such RTL processor, and then describe how to build the systems out of RTL processors. This way we can define semantics for RTL modeling, simulation, and synthesis.

1 RTL PROCESSOR

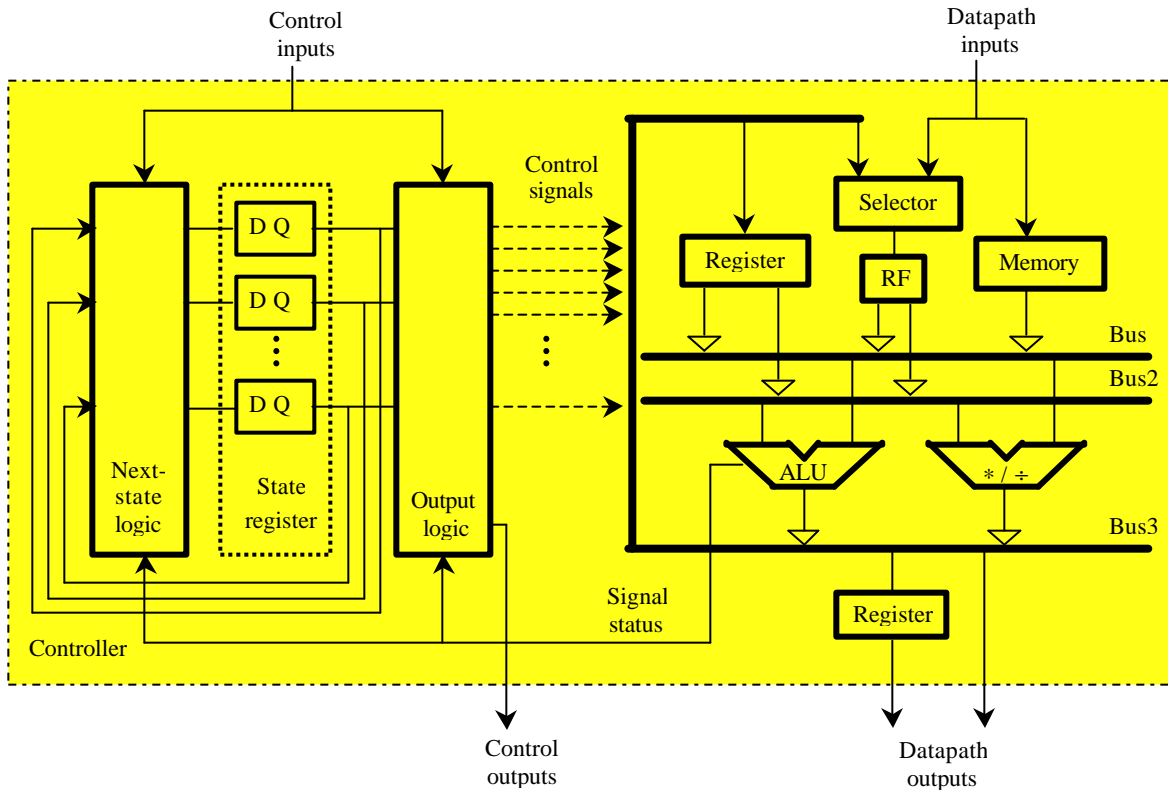
In order to define the RTL design flow we define the RTL-processor model. Such a model consists of a controller and a datapath. As shown in Figure 1(a), the model has two types of I/O ports. One type of I/O ports are data ports, which are used by the outside environment to send and receive data to and from the model. The data could be of type integer, floating point, characters, bit vector or any other type and any size. The data ports are usually 8, 16, 32 or 64 bits wide and have different types of attributes. The other types of I/O ports are control ports, which are used by the outside environment to receive the information about the status of the model and to send the information about the status of the environment. These two types of ports may be identified in the model definition so that the controller and datapath can be easily synthesized without complex inference procedure or they may be left untyped for synthesis to decide.

As shown in Figure 1(b), the datapath consist of storage units such as registers, register files, and memories, and combinatorial units such as ALUs, multipliers, shifters, and comparators. These units and the input and output ports are connected by

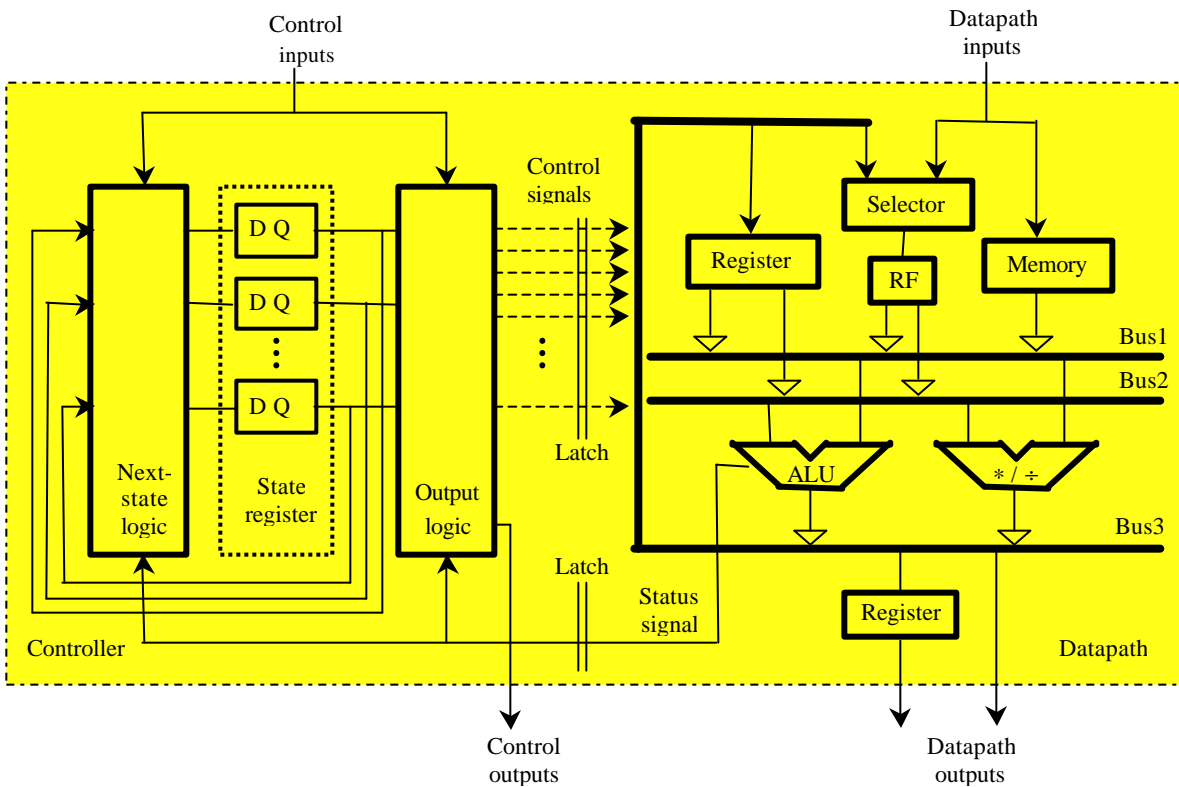


High-level block diagram

FIGURE 1(a) RTL Processor



(b) Register-transfer-level block diagram

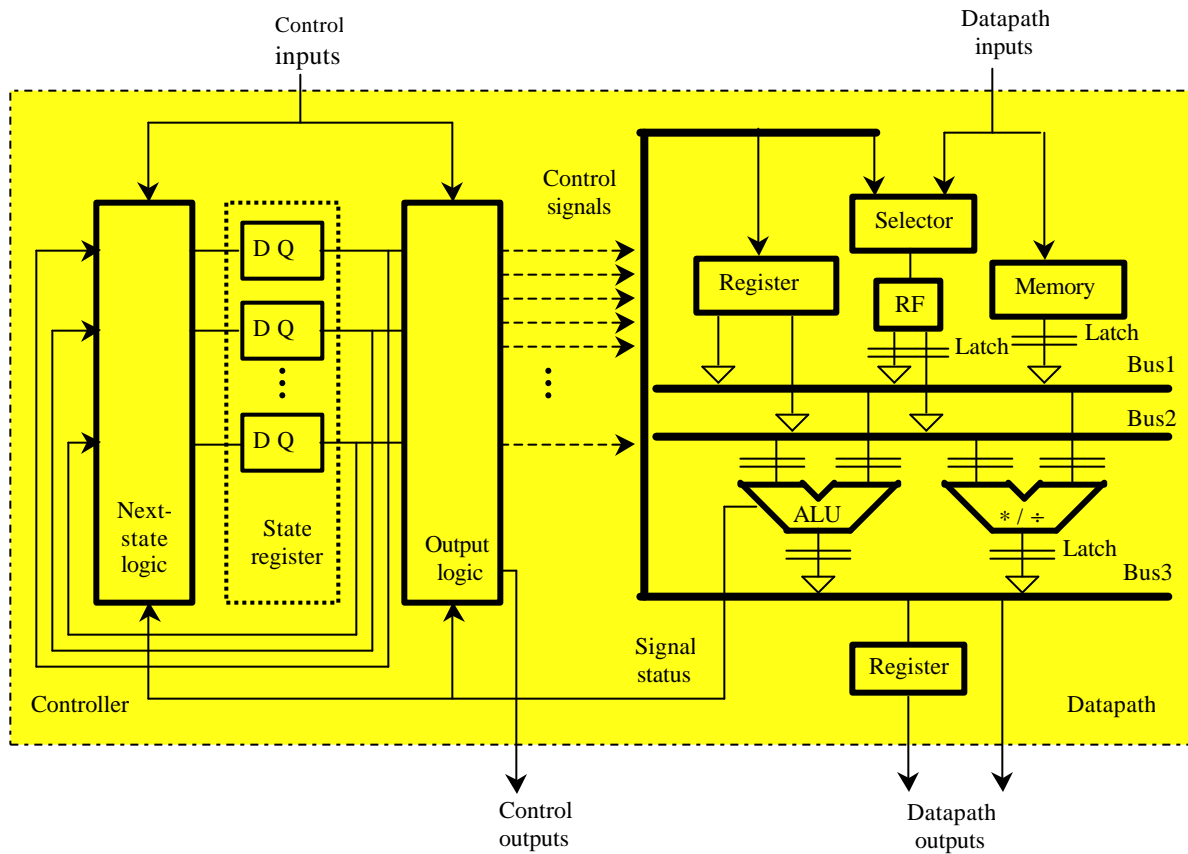


(c) Control-pipelined register-transfer-level diagram

FIGURE 1(b, c) RTL Processor.

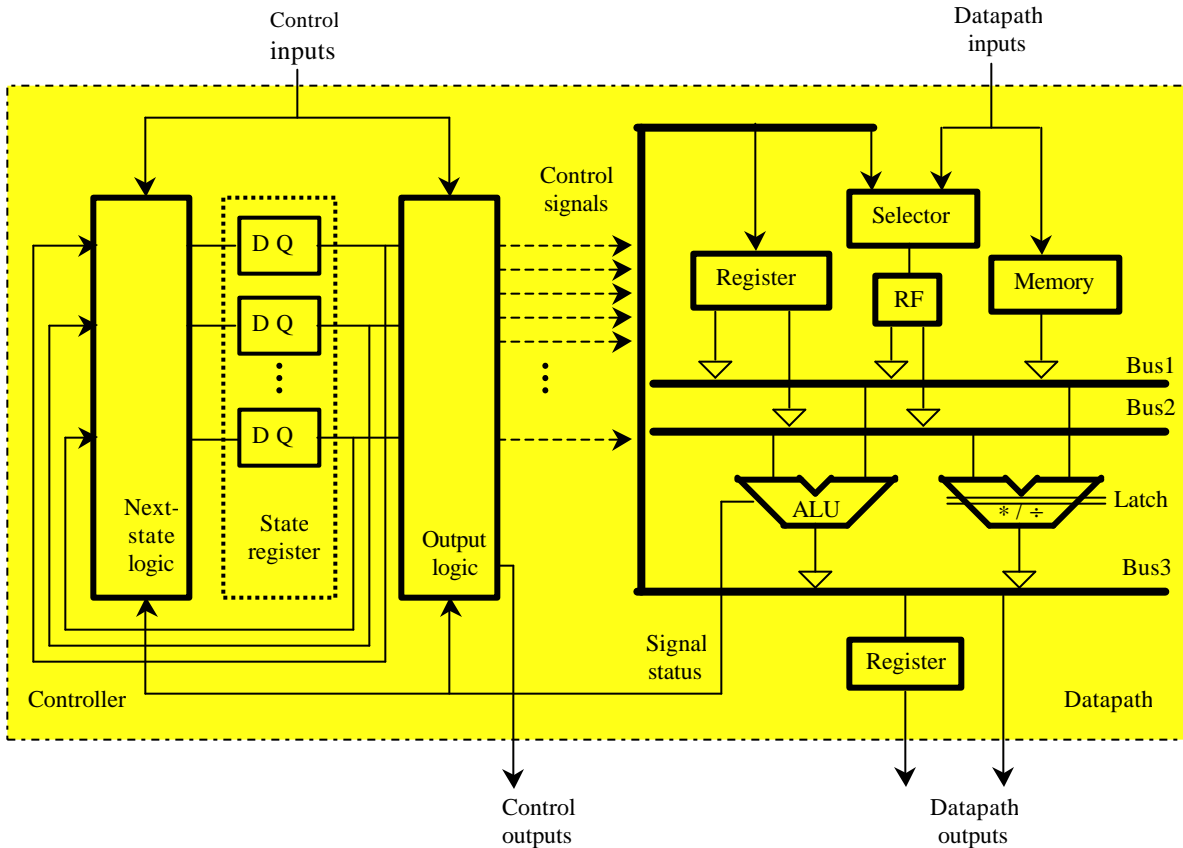
buses. The datapath takes the operand from storage units or input ports, performs the computation in the combinational units, and returns the results to storage units or output ports during each state, which is usually equal to one clock cycle.

The selection of operands, operations, and the destination of the result are controlled by the controller by setting proper values of datapath control signals. The datapath also indicates through status signals when a particular value is stored in a particular storage unit or when a particular relation between two data values stored in the datapath is satisfied. The input ports can be connected directly to register or storage units or to any other component in the datapath including the output ports. The output ports could be used for possible connections to other RTL processors through outside buses or directly through point-to-point connection.



(d) Datapath-pipelined register-transfer-level diagram

FIGURE 1(d) RTL Processor



(e) Function-pipelined register-transfer-level diagram

FIGURE 1(e) RTL Processor.

Similar to the datapath, a controller has a set of input and a set of output signals. Each signal is usually but not necessarily a Boolean variable. There are two types of input control signals: external signals and status signals. External signals represent the conditions in the external environment on which the model must respond. The *Start* signal in one's counter example in Figure 2(a), which starts the one's counter, is such an input signal. On the other hand, the status signals represent the state of the datapath. Their value is obtained by comparing values of selected variables stored in the datapath. For example, $Data = 0$ in one's counter example is such a signal whose value is equal to 1 when the value of *Data* is equal to 0 and 0 when *Data* value is not equal to 0.

There are also two types of output control signals: datapath control signals and external signals. The control signals select the operation for each component in the datapath, while the external signals identify to the environment that the model has reached a certain state or finished a particular computation. A controller consists of state register and next-state and output logic. Next-state logic generates the value for the state register in the next clock cycle while output logic generates the value of control and external signals. If the external control signals depend only on the state of the controller, the controller is called state-based or Moore type controller, and if they also depend on the input signals then the controller is called input-based or Mealy type controller.



Each RTL processor follows this general architecture, although two RTL processors may differ in the number and type of control units and datapaths, the number of components and connections in the datapath, the number of states in the control unit, and the number and type of I/O ports.

A RTL processor may also be pipelined in several different ways:

- (a) By inserting latches or registers on control signals and/or status signals, we obtain pipelined control as shown in Figure 1(c). Control registers are usually inserted in the last implementation stage, while status register is frequently used from the beginning. However, status register introduces at least one state delay. In other words, the condition evaluation must be performed one state before it is used, since it is loaded into status register in one state and used in the other. Similarly, the control register introduces one state delay in conditional evaluation.
- (b) Datapath can also be pipelined by inserting latches or registers on selected connections, such as after storage elements, before functional units, and after functional units as shown in Figure 1(d). With datapath pipelining the result of register transfers can be used only n states (clock cycles) later where n is the number of datapath stages.
- (c) Each function unit can be pipeline by dividing it into several stages and inserting latches between the stages as shown in Figure 1(e). The multiply/divide unit is divided into 2 stages. In the case of pipelined units, the result of the operation can be used only n states later, where n is the number of the pipelined stage in the functional unit.

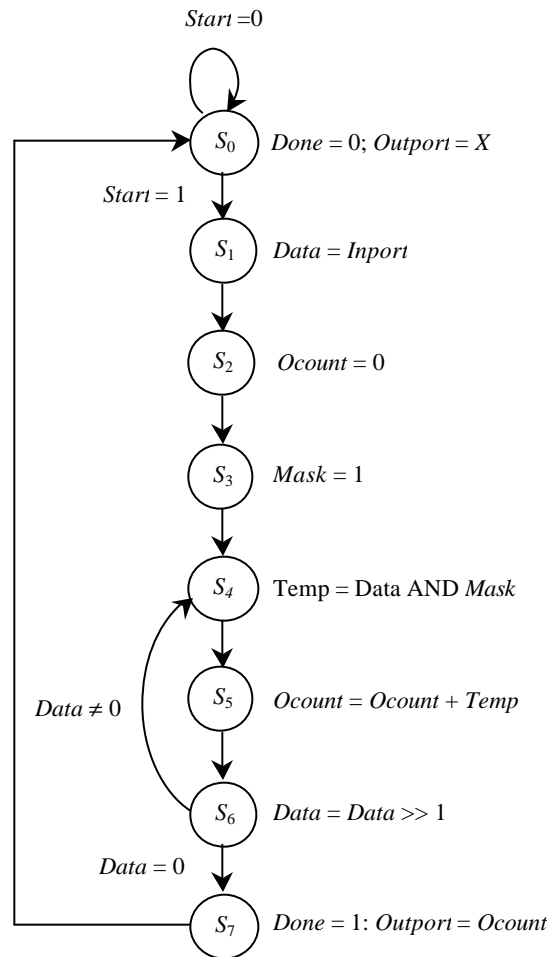
2 FSMD DEFINITION

In Section 1, we discussed in general terms the RTL processor model. In this section we discuss how to specify its functionality. We will introduce it on the example of a one's counter, which counts number of 1s in the word presented at the *Inport* as shown in Figure 2(a).

The one's counter is specified by an FSM, representing the control unit and a set of variable assignments representing register transfers in the datapath.

The FSM has eight states and transitions from one state to another under the control of the external signal *Start* and the status signal ($Data = 0$). In each state the FSM assigns values to a set of datapath control signal which completely specifies the behavior of the datapath. However, when there are too many control signals it is difficult to realize what and how the datapath will operate. To improve the comprehension of such a specification, we use variable assignment statements to indicate changes in variable values describe in the datapath operation.

A variable assignment statement gives an expression to be used for computation of the new variable value. In each state and for each variable assignment associated with that state the datapath evaluates the expression on the right-hand side of the assignment and assigns it to the variable on the left-hand side of the assignment. Generalizing from the one's counter specification, we may say that an FSM model with assignment statements added to each state, called an **FSM with data**, or FSMD, can completely specify the behavior of an arbitrary RTL processor.



(a) State diagram

FIGURE 2(a) One's counter specification

In order to define an FSMD formally, we must extend the definition of a FSM by introducing sets of datapath variables, inputs, and outputs that will complement the sets of FSM states, inputs, and outputs. As usually defined, an FSM as a quintuple

$$\langle S, I, O, f, h \rangle$$

where S is a set of states, I and O are the sets on input and output symbols, and f and h are functions that define the next state and the FSM output. More formally, f and h are defined as mapping

$$\begin{aligned} f &: S \times I \rightarrow S \\ h &: S \times I \rightarrow O \end{aligned}$$

they are usually specified by a table in which the next state and output symbols are given for each state and each input symbol. Each state, input, and output symbol is defines by a cross-product of variables. More precisely,



$$I = A_1 \times A_2 \times \dots \times A_k$$

$$S = Q_1 \times Q_2 \times \dots \times Q_m$$

$$O = Y_1 \times Y_2 \times \dots \times Y_n$$

where A_i , $1 \leq i \leq k$, is an input signal, Q_i , $1 \leq i \leq m$, is the flip-flop output, and Y_i , $1 \leq i \leq n$, is an output signal.

To include a datapath, we must extend the FSM definition above by adding the set of datapath variables, input and output ports. More formally we define a variables set

$$V = V_1 \times V_2 \times \dots \times V_q$$

which defines the state of the datapath by defining the values of all variables in each state. In the same fashion we can separate the set of FSM inputs into a set of FSM inputs I_C and a set of datapath inputs I_D . Thus

$$I = I_C \times I_D$$

where $I_C = A_1 \times A_2 \times \dots \times A_k$ as before and $I_D = B_1 \times B_2 \times \dots \times B_p$.

Similarly, the output set consists of FSM outputs O_C and datapath outputs O_D . In other words,

$$O = O_C \times O_D$$

where $O_C = Y_1 \times Y_2 \times \dots \times Y_n$ as before and $O_D = Z_1 \times Z_2 \times \dots \times Z_r$. However, note that A_i , Q_j and Y_k usually represent Boolean variables, while B_i , V_i , and Z_i represent bit-vectors, integers, floating-point numbers, characters and other data types.

Except for very trivial cases, the size of the data-path variables and ports makes specification of function f and h in tabular form very difficult. To be able to specify variable values in an efficient and understandable way in the definition of an FSM, we specify variable values using computable functions defined by mathematical expressions.

We define the set of all possible functions, $Func$, over the set of variables V to be the set of all constants K of the same type as variables in V , the set of variables V itself, and all the functions obtained by combining two functions with arithmetic, logic, or rearrangement operations. More formally,

$$Func(V) = K \cup V \cup \{(e_i * e_j) \mid e_i, e_j \in Func, * \text{ is an acceptable math operator or a computable function}\},$$

Using $Func(V)$, we can define the values of the status signals as well as transformations in the datapath. Let $STAT = \{stat_k = f_{B_i}(e_i, \Delta e_j) \mid e_i, e_j \in Func(V), \Delta \in \{\leq, <, =, \neq, >, \geq\}\}$ be the set of all status signals that are described as a Boolean function of one or more relations between variables or functions of variables. Examples of status signals are $Data = 0$, $(a-b) > (x+y)$, and $(counter = 0) \text{ AND } (x > 10)$. The relations defining status signals are either true, in which case the status signal has value 1, or false, in which case it has value 0.



With formal definition of functions and relations over a set of variables, we can simplify function $f : (S \times V) \times I \rightarrow S \times V$ by separating it into two parts: f_C and f_D . The function f_C defines the next state of control unit,

$$f_C : S \times I_C \times STAT \rightarrow S$$

while the function f_D defines the values of datapath variables in the next state.

$$f_D : S \times V \times I_D \rightarrow V$$

In other words, for each state $S_i \in S$ we compute a new value for each variable $V_j \in V$ in the datapath by evaluating a function $e_j \in Func(V)$. Thus the function f_D is represented by a set of simpler functions, in which each function in the set defines variables values for the state S_i :

$$f_D := \{f_{Di} : V \times I_D \rightarrow : \\ \{V_j = e_j \mid V_j \in V, e_j \in Expr(V \times I_D)\}\}$$

In other words, function f_D is decomposed into a set of functions f_{Di} , where each f_{Di} assigns one value e_j to each variable V_j in the datapath in state S_i . Therefore, new values for all variables in the datapath such that $1 \leq j \leq q$. are computed by evaluating functions e_j , for all j .

Similarly, we can decompose the output function $h : S \times V \times I \rightarrow O$ into two different functions, h_C and h_D defines the external control outputs O_C as in the definition of an FSM and h_D defines external datapath outputs. Therefore,

$$h_C : S \times I_C \times STAT \rightarrow O_C \\ h_D : S \times V \times I_D \rightarrow O_D$$

The above definition of a FSMD can be given in tabular form with a state and output table as shown in Figure 2(b) for the case of the one's counter. The first three columns define the present state, the next state, and external control outputs, whereas the next two columns define the datapath outputs and variable values. As usual, the symbol X is used for don't-care conditions. From the table in Figure 2(b) we see that a new value is assigned to each of the control outputs, datapath variables, and datapath outputs in each state. This kind of tabular definition may become awkward to comprehend and manipulate for large FSMDs with many states and hundreds of variables in the datapath.

Fortunately, many of these variables seldom change their values except when they represent pipeline registers or latches. Therefore, it would be more efficient if we assume that variables retain their old values if no new value is specified in a particular state. Therefore, the third, fourth and fifth column in Figure 2(b) could be written as a set of assignment statements, reminding us of straight-line code in most programming languages.

Similarly, the second column can be expressed as a set of conditions for transitions to particular states. Such a reduced table is called **state-action table** as shown in Figure 2(c). It is equivalent syntactically and semantically to the specification in Figure 2(a).

As an example, we show a state-action table for the one's counter in Figure 2(c). Such a table is easy to understand and provides all the necessary information for the implementation of a control unit and a datapath. It can be used to construct the state diagram for the control unit, synthesize next-state and output logic, and define the datapath components and their connections. It is also very easy to translate such table (Figure 2(c)) or state-diagram (Figure 2(a)) to any of the hardware description languages (such as VHDL or Verilog) or to any programming languages such as C or Java.



From our FSMD definition (tubular, graphic or language based) we see that in each state we compute the next state which depends on some condition from the outside environment or on some

| PRESENT STATE | NEXT STATE (Start, Data = 0) | | | | CONTROL OUTPUT <i>Done</i> | DATAPATH OUTPUT <i>Outport</i> | DATAPATH VARIABLES | | | |
|---------------|---------------------------------|-------|-------|-------|-------------------------------|-----------------------------------|--------------------|---------------|---------------|-------------|
| | 00 | 01 | 10 | 11 | | | <i>Data</i> | <i>Ocount</i> | <i>Temp</i> | <i>Mask</i> |
| S_0 | S_0 | S_0 | S_1 | S_1 | 0 | Z | X | X | X | X |
| S_1 | S_2 | S_2 | S_2 | S_2 | 0 | Z | Inport | X | X | X |
| S_2 | S_3 | S_3 | S_3 | S_3 | 0 | Z | Data | 0 | X | X |
| S_3 | S_4 | S_4 | S_4 | S_4 | 0 | Z | Data | Ocount | X | 1 |
| S_4 | S_5 | S_5 | S_5 | S_5 | 0 | Z | Data | Ocount | Data AND Mask | Mask |
| S_5 | S_6 | S_6 | S_6 | S_6 | 0 | Z | Data | Ocount+Temp | X | Mask |
| S_6 | S_4 | S_7 | S_4 | S_7 | 0 | Z | Data>>1 | Ocount | X | Mask |
| S_7 | S_0 | S_0 | S_0 | S_0 | 1 | Ocount | Data | Ocount | X | X |

(b) State and output table

| PRESENT STATE | NEXT STATE | | CONTROL AND DATAPATH ACTIONS | |
|---------------|-------------------------------------------------------------------------|-------|---------------------------------------------------------------------------|--------------------------|
| | CONDITION, | STATE | CONDITION, | ACTIONS |
| S_0 | $\left[\begin{array}{l} Start = 0, \\ Start = 1, \end{array} \right.$ | S_0 | $\left[\begin{array}{l} Done = 0 \\ Output = Z \end{array} \right.$ | |
| S_1 | | S_1 | | |
| S_2 | | S_2 | | $Data = Inport$ |
| S_3 | | S_3 | | $Ocount = 0$ |
| S_4 | | S_4 | | $Mask = 1$ |
| S_5 | S_5 | S_6 | $Temp = Data \text{ AND } Mask$ | |
| S_6 | $\left[\begin{array}{l} Data \neq 0, \\ Data = 0, \end{array} \right.$ | S_4 | $\left[\begin{array}{l} Done = 1 \\ Output = Ocount \end{array} \right.$ | $Ocount = Ocount + Temp$ |
| S_7 | | S_7 | | $Data = Data \gg 1$ |
| | | S_0 | | |

(c) State-action table

FIGURE 2(b, c) One's counter specification

status signal computed from values of FSMD variables. The FSMD definition also allows assignment of vales to any variable in each state. However, variables and functions in the definition of FSMD may have different interpretations which in turn defines several different styles of RTL semantics.

Figure 3(a) shows five (5) different RTL styles for a 4-state segment of a FSMD definition and necessary mappings to arrive to that particular style.

| STATE TRANSITIONS | UNMAPPED RTL (Style 1) | STORAGE MAPPED RTL (Style 2) | FUNCTION MAPPED RTL (Style 3) | CONNECTION MAPPED RTL (Style 5) | EXPOSE CONTROL RTL (Style 5) | STRUCTURAL RTL |
|-------------------|-------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------|----------------|
| | $a = d * e$ $b = f+()$ $c = \dots$ $f3(a,b,c)$ | $r1 = f*(r1,M(0))$ $r2 = f+(...)$ $M(0) = \dots$ | $r1 = FU2(*,r1,M(0))$ $r2 = \dots$ $M(0) = \dots$ | $Bus1 = r1$ $Bus2 = M(0)$ $Bus3 = FU2(*,Bus1,Bus2)$ $r1 = Bus3$ \dots | $C1 = 1$ $C2 = 0$ $C3 = 1$ $C4 = 0$ $C5 = 1$ $C6 = x$ $C7 = 0$ $C8 = 0$ $C9 = 0$ | |
| | $d = f4(a,b)$ $e = f5(c,d,e)$ | $r1 = \dots$ $M(0) = \dots$ | $r1 = \dots$ $M(0) = \dots$ | | | |
| | $f = a - b$ $g = f\#(\dots)$ $h = \dots$ | $r1 = f-(r1, r2)$ $r2 = \dots$ $M(1) = \dots$ | $r1 = FU1(-,r1,r2)$ $M(1) = r1$ | $Bus1 = r1$ $Bus2 = r2$ $Bus3 = FU1(-,Bus1,Bus2)$ $r1 = Bus3$ \dots | $C1=1$ $C2 = 1$ $C3 = 0$ $C4 = 1$ $C5 = 0$ $C6 = 1$ $C7 = x$ \dots | |
| | $i = \dots$ $j = \dots$ $k = \dots$ | $M(2) = \dots$ $r2 = \dots$ $r1 = \dots$ | $M(2) = \dots$ $r2 = \dots$ $r1 = \dots$ | | | |
| MAPPINGS | Storage mappings $\{a,d,f,k\} = r1$ $\{b,h,j\} = r2$ $\{c,e\} = M(0)$ $\{g\} = M(1)$ $\{i\} = M(2)$ \dots | Function mappings $\{f-, f+\} = FU1$ $\{f*, f\#\} = FU2$ $\{f3,f4,f5\} = FU3$ | Connection mappings $r1_to_FU1L = Bus1$ $r1_to_FU2L = Bus1$ $r2_to_FU1R = Bus2$ $M(0)_to_FU2R = Bus2$ $FU1_to_r1 = Bus3$ $FU2to_r1 = Bus3$ | Control mappings $r1_to_Bus1 = C1$ $r2_to_Bus2 = C2$ $M_to_Bus2 = C3$ $FU1_to_Bus3 = C4$ $FU2_to_Bus3 = C5$ $FU1 = C6$ $FU2 = C7$ $M(address) = C8$ $M(read/write) = C9$ | | |

(a) State diagram with different levels and mappings

FIGURE 3(a) An incomplete example



2.1 Unmapped RTL (Style 1)

In the Unmapped RTL the variables are divided into ports and internal variables, while ports are further divided into control and data ports, where each could be an input, output or input-output port. Unmapped RTL only specifies in each state the change of values for some variables. The order in which assignments are executed is determined by control dependencies, that is, the order written in the description. States, transitions and assignment statements are in no way related to any implementation. The variables do not represent registers or busses and functions or operations do not represent any functional units.

The Unmapped RTL is equivalent to the programming language code with exception that such code is divided into states, with conditional transition between states added to the code. For example, we see several assignment statements in each state in Figure 3(a). All statements assign values to uninterpreted variables. The values are computed by using standard language operators or functions if values require more complex computation. It is assumed that each operator or a function is computed in one clock cycle or less.

3 Mapped RTL

In the mapped RTL, the uninterpreted variables are mapped into storage units or wires/buses and computing functions or operators are assigned to functional units. Although this mapping can be performed in any order, it is convenient to map variables into storage first, followed by function mapping and then mapping of wires to buses. This way we can define three styles of mapped RTL.

3.1 Storage-mapped RTL (Style 2)

The variables in Style 1 can be of two types. One type are variables whose value is used in the same state in which that value is assigned. These variables represent wires. The other type are variables whose values are assigned in one state and used in another state. The states between the value assignment and its last usage define the lifetime of each variable. These variables must be mapped to storage units such as register, register files, and memories. Thus, storage-mapped RTL is a RTL description in which the second type of variables with non-overlapping lifetimes are grouped and assigned to storage units. In other words, a group of internal variables is replaced by a new variable of type storage.

In our example in Figure 3(a), we grouped variables a , d , f and k and assigned them to register $r1$ while b , h , and j were assigned to register $r2$. Similarly variables c and e were assigned to memory location $M(0)$ while g and i to memory locations $M(1)$ and $M(2)$. This assignment is shown in storage mapping table at the bottom of Style 1 in Figure 3(a). Note, that in Style 2 we used, functional notation for all the operators for uniformity sake.

Synthesis Note: Storage mapping consists of allocating some storage units, grouping variables with non-overlapping lifetimes, assigning them to storage units and replacing variables with storage unit names to which they have been assigned. Note that variables representing wires are not grouped in any way or mapped to any real wires or buses.



Simulation Note: While variables representing wires are assigned values in order written in the spec, variables representing storage units are assigned values on the next clock event (rising or falling edge of a clock). Since a clock event also represents transition between states, the value assigned in the present state can be used only in future states until it is reassigned again.

3.2 Function-mapped RTL (Style 3)

In Function-mapped RTL, the operators and/or functions with non-overlapping lifetimes are grouped into functional units, and a control encoding is assigned to each operation in the functional unit. Therefore, in Style 3 we must identify the operation performed by each function unit in each state. Style 3 is the same as Style 2 with functions replaced by multi-operation functional units. Note, that original functions and functions representing functional units use the same syntax.

As we see in Function mapping table in Figure 3(a), we have three functional units: *FU1* performing addition and subtraction, *FU2* performing multiplication and operation #, and *FU3* performing functions *f3*, *f4*, and *f5*. Thus, in Style 3 in Figure 3(a) operators and functions *f3*, *f4*, and *f5* are replaced by functions performed by functional units *FU1*, *FU2*, and *FU3*. If a functional unit needs more than one state to generate result then its input must be the stable through all the states while its output is loaded only in the last state. Functional unit may have several outputs, each of which must be clearly declared when assigning a new value to a variable.

Synthesis Note: Function mapping consists of allocating some functional units, grouping operations and functions with non-overlapping lifetimes into groups assigned to each functional unit and replacing operators and functions with functions representing functional units. If a functional unit takes more than one clock cycle to evaluate, then the FSM model must be adjusted. Function and storage mapping can be performed in any order.

Simulation Note: If a functional unit takes *n* states to execute, then the data and control inputs must hold the same value for *n* states, while result is loaded or used after *n* states.

3.3 Connection-mapped RTL (Style 4)

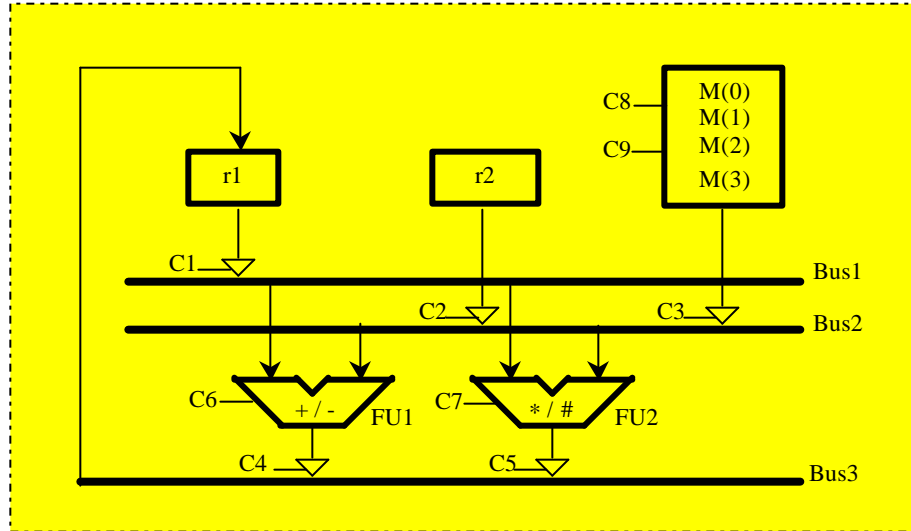
Similarly to Style 2, the variables, with non-overlapping life times, that represent wires as well as inputs and outputs to and from storage elements and functional units are grouped and assigned to busses. Syntactically, there is no difference between wires and buses. The only difference is in additional bus drivers that must be inserted in Style 5. Similarly, we can merge (multiplex) ports if they are not used at the same time.

We see from Connection mapping table in Figure 3(a), that connections from register *r1* to *FU1* and *FU2* are assigned to *Bus1*, connections from register *r2* to *FU1* and memory *M* to *FU2* to *Bus2*, while connections from *FU1* and *FU2* to register *r1* are assigned to *Bus3*. The refinement from Style 3 to Style 4 is similar to compiling a programming language into assembly language. Each assignment statement is decomposed into statements representing transfers from storage elements to functional units, functional units operation, and transfers back from functional units to storage elements. For demonstration purposes, this decomposition is performed only for the first assignment statement in states *S1* and *S3* in Figure 3(a).

Synthesis Note: Each register transfer (such as $x = a + b$) must be expanded with variables representing wires that connect storage units (such as *a*, *b*) to functional units (such as *+*) and output of functional units (such as *+*) to storage units (such as *x*). Once the wire variables are

introduced they can be grouped and assigned to buses. Bus is just another variable. Since only one source can be assigned to each variable in each state the problem of multiple drivers is avoided.

Simulation Note : No need for resolution function as per note above.



(b) Partial design

FIGURE 3(b) An incomplete example

3.4 Exposed-control RTL (Style 5)

In Exposed-control RTL, the FSM model consists of two parts: netlist of datapath components and a controller that assign a constant to each control variable in each state. The control variables specify the operation of each storage, functional or bus component in the datapath.

Control mappings to perform the Style 4 assignments in Figure 3(a) is shown in control mapping table. Thus, the transfers and operations are replaced by assignments to control signals for all storage, functional and bus units. The Style 5 column shows the control assignments for two statements given in Style 4. Also, the partial design corresponding to these two statements is shown in Figure 3(b).

The datapath netlist consists of declared components (storage, functional and connection) and two types of variable assignments, component ports to wires or buses and buses to ports. Note that more than two ports, can be assigned to each bus, but not in the same state. This situation requires that each port has a tristate driver that is allowed to drive the bus when its corresponding control signal is asserted. Similarly, two or more buses can be assigned to the same port requiring insertion of a selector. Such a selector (multiplexer) is controlled by corresponding control signal from the controller.



Synthesis Note : The refinement from Style 4 to Style 5 consists of extracting all the storage and functional units and connecting them with wires and busses from Style 4, thus forming the netlist for the datapath. Furthermore, all the register transfer statements in each state are omitted and replaced with assignments of constants to control variables that control the storage and functional units and busses. If control register is added, then all dependencies must be checked to accommodate an extra state delay. If that is not the case an extra states must be inserted to satisfy dependencies.

Simulation Note : Since control variables are wires, they are instantly assigned the values. The datapath is the netlist whose component models are run concurrently and sensitive to changes in control wires. In case of control register, they are sensitive to events on control register outputs.

General Note 1 : All styles have the same syntactic rules. Semantically they differ in types of variables:

- (a) uninterpreted (Style 1)
- (b) storage (Style 2, 3, 4)
- (c) wires (Style 2, 3, 4)
- (d) buses (Style 4)
- (e) control (Style 5)
- (f) special (clock, reset) (all Styles)

These types of variables are necessary so that proper implementation can be synthesized from any of the styles.

General Note 2 : All variable types may be mixed in any style. However, synthesis algorithms for the final implementation in this case may be more complex.

General Note 3 : All mappings can be performed in any order including also partial mappings at any step.

General Note 4 Control and Datapath pipelining introduces pipelined registers, which are considered to be equivalent to any other registers in the datapath. However, the RTL description must be checked (manually or automatically) that data and control dependencies are satisfied. In other words, the result of an operation or conditional evaluation can be used only n states later where n is the number of pipelined registers between the source and destination.

In case of pipelined functional units the pipelined registers do not have to be declared as long as it is understood that destination register is loaded n states after assignment state and the result used in the states after it is loaded into destination register.

The situation is the same in case of multicycle functional units. The only exception is that the assignment statement must be repeated in all state that functional unit is executing so that controller can keep operands at the input ports, functional unit executing the same operation and loading the destination in the last clock cycle.

4 COMMUNICATING FSMDS

As we defined in previous action, each FSMDS is uniquely defined by a set of ports and a description in any of the styles 1-5. The ports are defined by name: type, data, size, and attributes. Type is usually Boolean, bit-vector, integer, floating-point, character or other user defined type.

Attributes may include electrical, mechanical, simulation, test and synthesis requirements or metrics. The two necessary attributes for synthesis are **setup** and **hold** time for the input ports and **delay** time for the output ports. It is not possible to synthesize a design in which RTL processor is a component without these timing attributes for ports. With such attributes included in the FSM model we can combine two or more FSMs as shown in Figure 4 into communicating FSMs. The input ports can be connected to any component inside the datapath and output port can also be driven by any component in the datapath.

In such a structure of communicating FSMs, any output ports can be connected to any input ports as long as the type, size, and attributes match or connection is unambiguously specified for non-matching ports. Note that the above definition allows a datapath to be connected to a control port and vice versa. However, in most practical case control ports are connected to control ports and data ports to data ports as shown in Figure 4. Control ports are used for synchronization and data ports for data exchange.

In any FSM a input port may be connected to any component in the datapath or controller, that is to the input of any storage or functional unit. Similarly, any output port can be driven by the output of any unit in the RTL processor. The above definition allows for the existence of two types of IO paths in each FSM:

- Combinatorial IO in which the change at the input port will propagate with same delay to the output port
- Sequential path in which there is a storage element on the IO path and the input change will impact the output port in the future states but not in the present state.

A FSM with only sequential IO paths is called state-based (or Moore FSM since its controller is a Moore FSM), while FSM with one or more combinatorial IO paths is called input-based

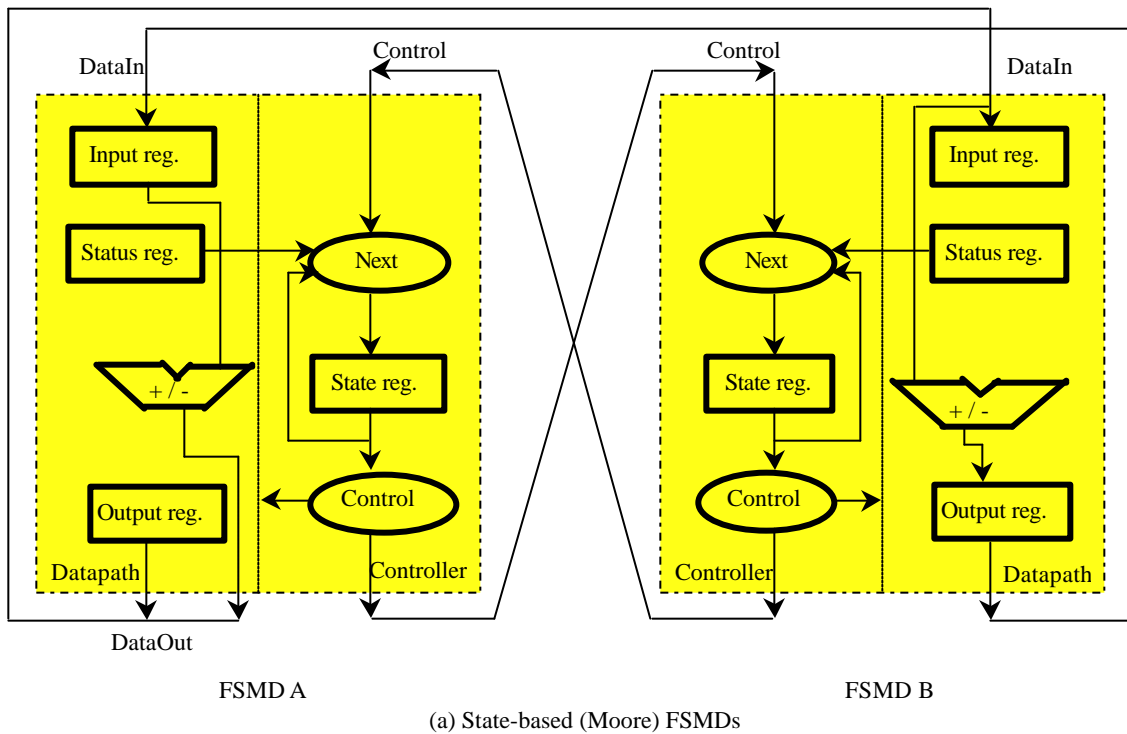


FIGURE 4(a) Communicating FSMs

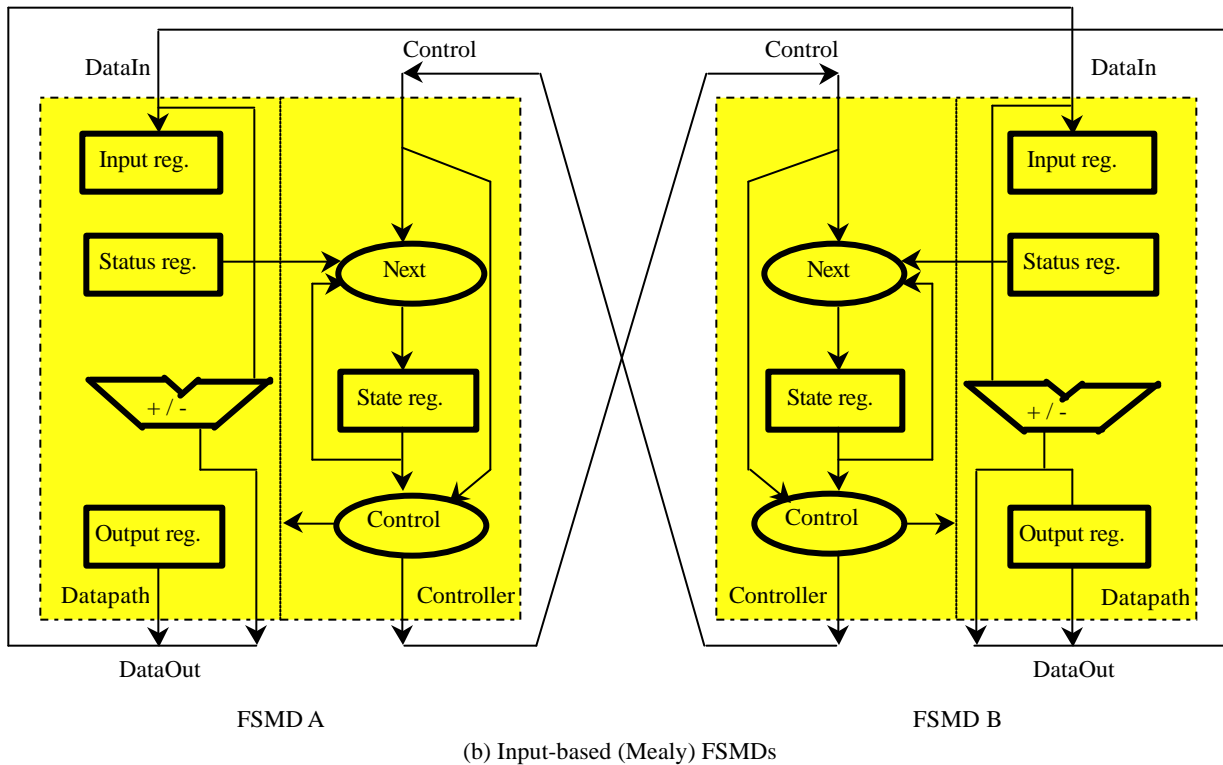


FIGURE 4(a) Communicating FSMDs

(or Mealy FSMD since its controller is a Mealy FSM). Figure 4(a) shows a connection of two state-based FSMDs while Figure 4(b) shows similar connection of two input-based FSMDs.

Unfortunately, the definition of communicating FSMDs allows creation of combinatorial loops by connecting two or more combinatorial IO paths in two or more different FSMDs serially in a loop. Such a combinatorial loop may lead to oscillation and should be avoided in good designs. A combinatorial loop can be avoided in three different ways:

- (a) Using only state-based FSMDs, which will guarantee that no output ports is driven from an input port in any of the FSMDs.
- (b) Having at least on register or storage unit in each loop but not necessarily in each IO path (in other words we may use input-based FSMDs in this case).
- (c) Having a combinatorial loop but making sure that it never gets used completely in any register transfer (fake loop).

Simulation Note: Communicating FSMDs require special care during simulation, since communicating FSMDs operate in parallel and simulator runs sequentially. In case (a) mentioned above, simulator must assign values to all output ports for all FSMDs before assigning values to all input ports. In this manner the proper operation of communicating FSMDs will be secured. Thus, for each FSMD described by a case statement the following order must be observed:

Case Statement

Compute all register transfers not dependent on input ports.

Assign values to all outputs ports.

Suspend process until all FSMs reach this point.
 Assign values to all input ports.
 Computer the register transfers dependent on input ports.
 Compute next state.

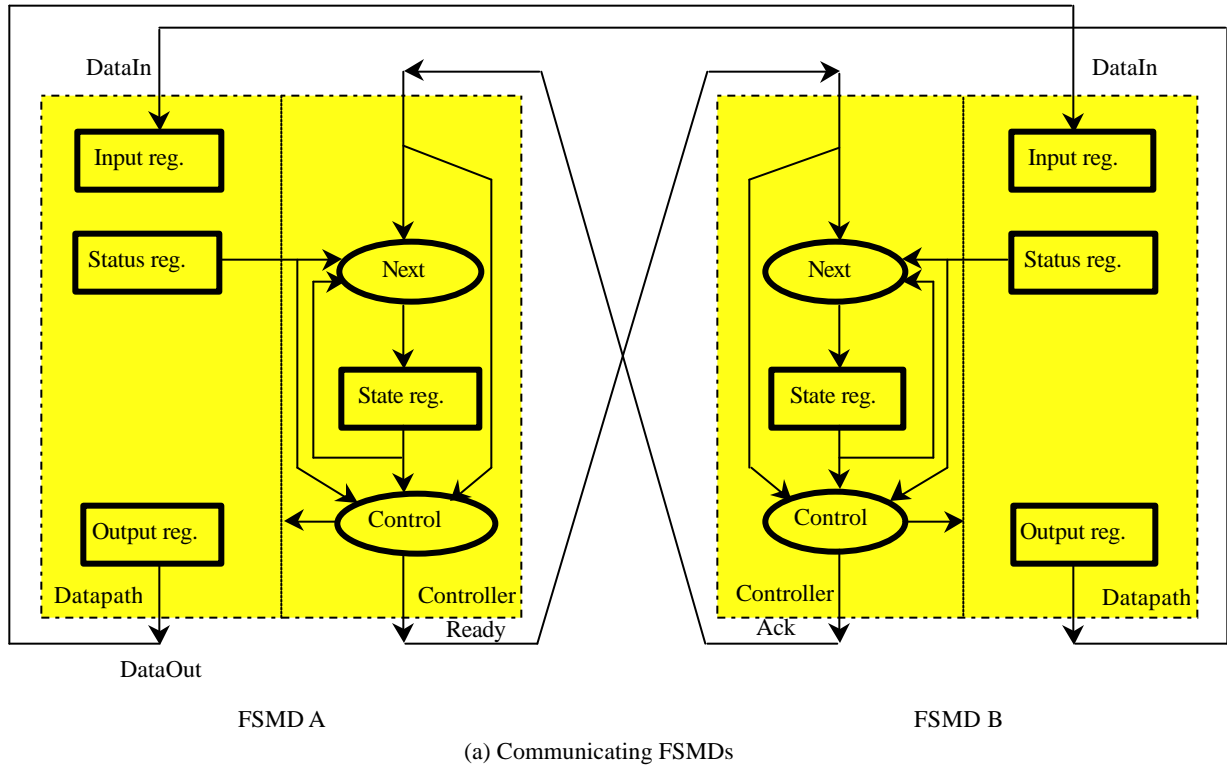
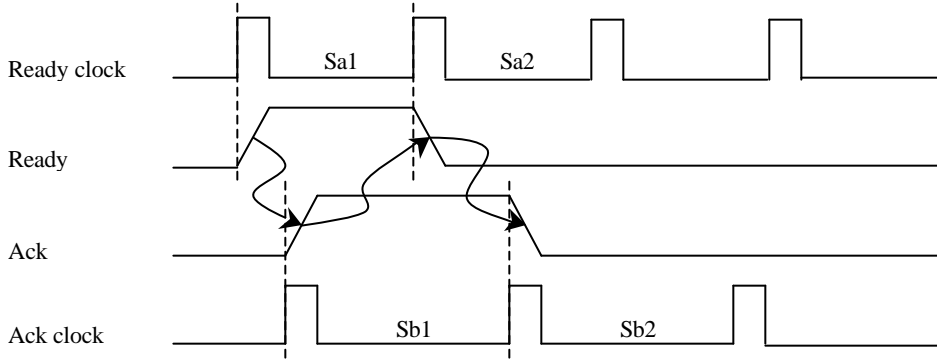
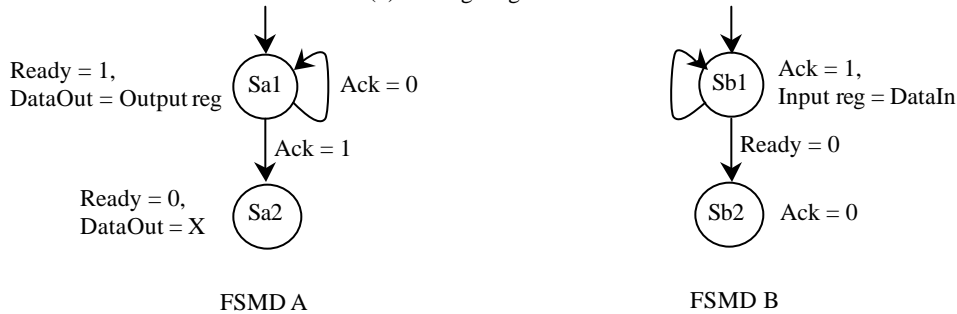


FIGURE 5(a) Synchronized FSMs

The suspension of simulation can be achieved in many different ways. In VHDL it is achieved by introduction of Δ delay and suspending the process by wait (Δ) statement which moves the simulation of the rest of the FSM description into the next Δ time slot. In case (b), simulation must be ordered in such a way that each loop gets evaluated starting from the register output and ends at the register input. This is similar to case (a) with an additional simulation order of FSMs required for each loop. This can be achieved for example by introducing a sensitivity list for all the input ports participating in the loop. In case (c) simulation is similar to case (b) with an additional check to make sure that the loop never gets exercised by not driving at least one port in the loop for example.



(b) Timing diagram



(c) State diagram

VHDL Code for FSM D A

```

process
begin
wait until clk'event and clk=1;
case State is
when Sa1 =>
... --reg. transfers
DataOut <= Output reg
Ready <= 1;
if(Ack=1) then
State <= Sa2;
else
State <= Sa1;
end if;
when Sa2 =>
... --reg. transfers
DataOut <= X
Ready <= 0;
State <= Sa3;
...
end process

```

VHDL Code for FSM D B

```

process
begin
wait until clk'event and clk=1;
case State is
when Sb0 =>
... --reg. transfers
Ack <= 0;
if(Ready=1) then
State <= Sb1;
else
State <= Sb0;
end if;
when Sb1 =>
... --reg. transfers
Input reg <= DataIn
Ack <= 1;
if(Ready=0) then
State <= Sb2;
else
State <= Sb1;
end if;
when Sb2 =>
... --reg. transfers
Ack <= 0;
State <= Sb3;
...
end process

```

(d) VHDL description

FIGURE 5 (b, c, d) Synchronized FSMs

4.1 CLOCKING OF COMMUNICATING FSMs

The same clock signal or different clock signals may drive the communicating FSMs. In the first case, the delay time of an output ports, the setup time of the input port, and delay of the connecting wires must be less than the clock period in case of state-based FSMs. Otherwise, the delay for any register to register transfer even through several FSM, must be less than a clock cycle. In the second case, we must make sure that two FSMs are synchronized during data exchange if the clocks are not multiples of each other. If they are, then the rules for FSMs with the same clock apply.

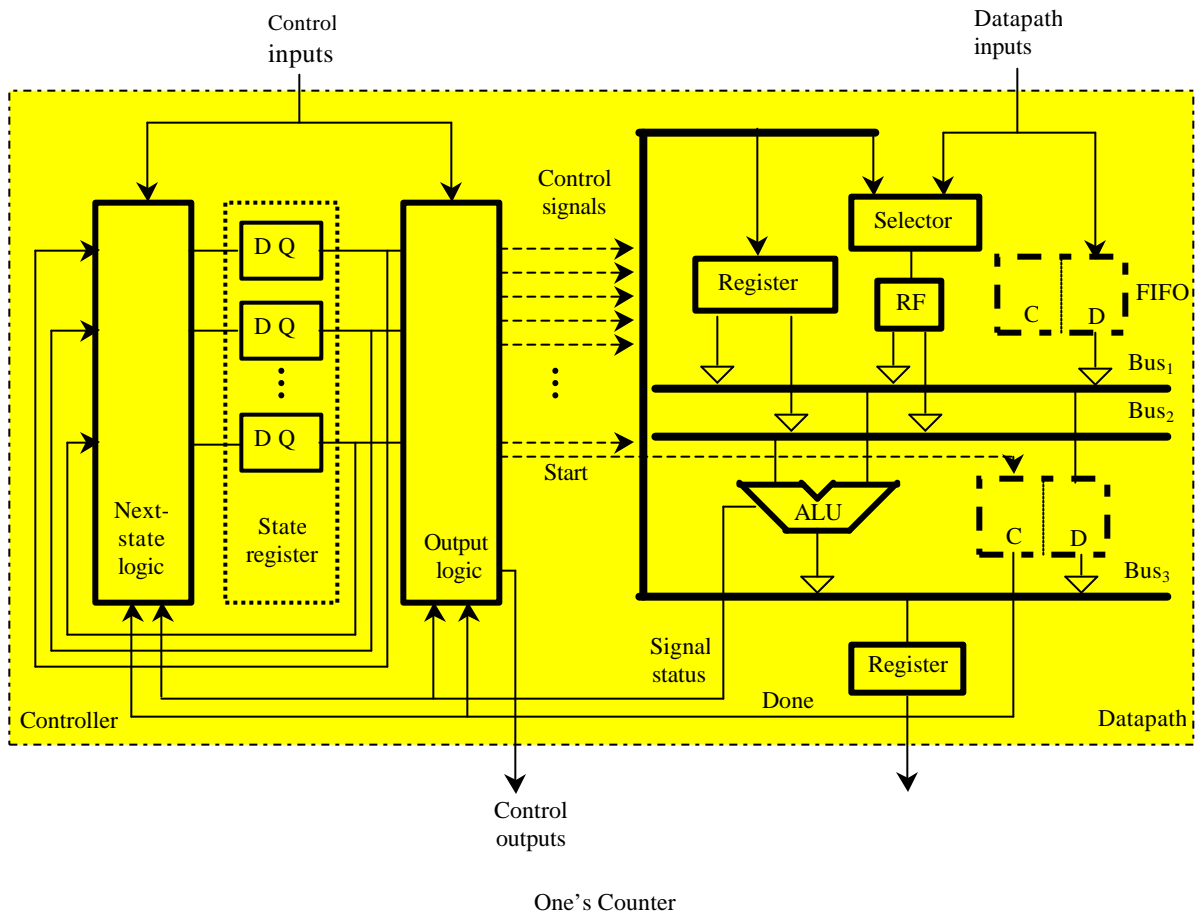


FIGURE 6 Embedded FSMs

In case of two FSMs with different clock signals (or two input-based FSMs) we can synchronize the data exchange by using *Ready* and *Ack* signals as shown in Figure 5(a). The *Ready* signal is asserted in state *Sa1* indicating that the data is ready at *DataOut* port as shown in timing and state diagrams in Figures 5(b) and 5(c). When FSM B recognizes that *Ready* signal is asserted it transitions to state *Sb1* in which it stores the data at *DataIn* port into *Input reg* and asserts *Ack* signal. Asserted *Ack*



removes the data from the *DataOut* port and deasserts *Ready* signal in state *Sa2*. After that, *Ack* gets deasserted in state *Sb2*.

The above protocol, in essence, transformed two input-based FSMs into two state-based FSMs for the duration of two states during data exchange, and thus interrupted the feedback loop. For completeness, VHDL code for this protocol is shown in Figure 5(d). Similarly approach is valid also for datapath loops.

5 HIERARCHICAL FSMs

The definition of an FSM in section 2 allows for hierarchical composition of FSMs. Each FSM can be a component in another FSM. In other words, an FSM may implement an arbitrary functional unit or a storage unit. In Figure 6 we show the *One's counter* used as a functional unit and a *FIFO* queue used as a storage unit. These component FSMs may need fixed number of states to finish such as a *FIFO* or a variable number of states such as *One's counter*.

In the first case the *FIFO* output may be used only fixed number of states later. In the second case the component FSM is synchronized with controller by use of *Start* and *Done* signals. The controller asserts the *Start* signal when the data at the input port is valid and component FSM asserts the *Done* signal when it is finished so that the data at its output port can be used. Note that *One's counter* asserts the *Done* signal and makes *Count* available at its output port for one clock cycle as shown in Figure 2. During that clock cycle the *Count* is loaded into *RF* or another storage element via *Bus₃*. In case that *Count* is loaded into *FIFO* and assuming *FIFO* is empty the *Count* value will be available several states later when *FIFO* becomes non-empty. At that time *FIFO* could be read and *Count* value processed further. In case the component and composite FSMs run at different clock cycle then synchronization with *Start* and *Done* signals must be used as shown in Figure 5. On the other hand any number of FSMs can be combined together in serial or parallel way to form larger FSMs as shown in Figure 7.

When connecting FSMs serially the control output *Done* of one FSM is connected to *Start* input of the other FSM. The *Start* input of the first becomes *Start* of the composite and the *Done* signal of the second becomes the *Done* of the composite. Obviously, other control and data ports can be connected arbitrarily, as the specification requires. When connecting FSMs in parallel fashion as shown in Figure 7(c), we may assume that one FSM takes the role of the master whose *Start* is the *Start* of the composite and whose *Done* is the *Done* of the composite. The rest of the ports can be connected as described in section 3. Note that *Start* and *Done* are not needed if both FSMs run the same clock and execution time is deterministic. In some cases the master FSM may be just reduced to a controller that synchronizes the other FSMs as shown in Figure 7(d). The FSM D gets started by the input *Start* signal and starts (not necessarily at the same time) the FSMs A and C. When C is finished, the data is transferred to FSM B that continues with its execution, which was started by FSM A. When B is finished it notifies D, which in turn asserts the *Done* signal for the composite.

6 SPECIAL CASES

6.1 FSM RESETING

Any register or storage element can be reset to any value. The reset is completely independent of clock and any other inputs and overrides the clocked or any other writing of the storage elements. In other words, if reset is asserted, the register or storage element is reset no matter what are the values on



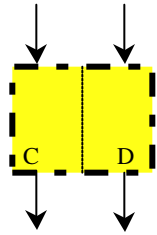
other inputs. Therefore, the FSM description must make sure that reset input is considered before any other input. There are two types of resets: synchronous and asynchronous.

Synchronous reset occurs on the clock edge and overrides the storage writing from any other input. The VHDL description for two communicating FSMs from Figure 5 with resetting feature is shown in Figure 8(a).

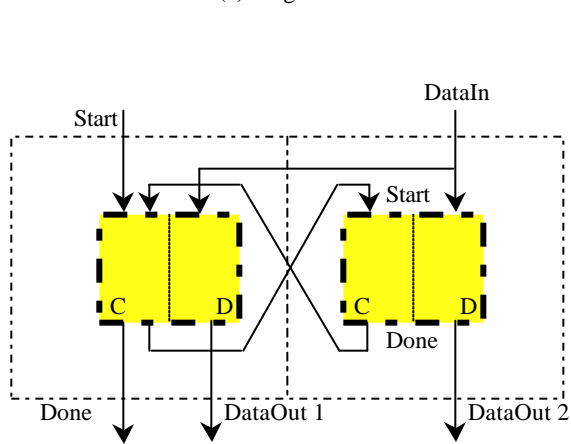
Asynchronous reset occurs at anytime and overrides the storage writing from any other input similarly to synchronous reset. Figure 8(b) shows VHDL description for two communicating FSMs for this case. In this case the description must be sensitive to reset as well as clock inputs.

Simulation Note: Since reset overrides anything else it must be considered always before any other FSM actions.

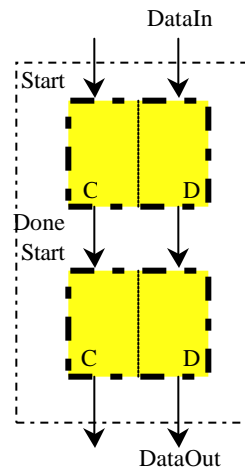
Synthesis Note: Reset input must be a special type of control input so that synthesis tools can distinguish it from other inputs and connect it properly to set/reset pins of a register or storage element (note, that VHDL does not type reset and thus it is only simulatable and not synthesizable).



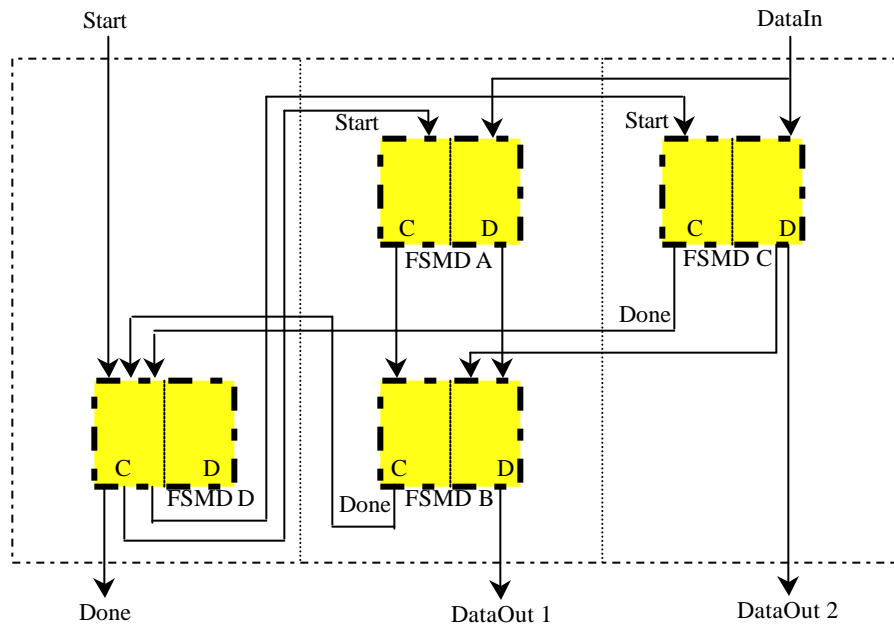
(a) Single FSM D



(c) Parallel master-slave connected FSM Ds



(b) Serially connected FSM Ds



(d) Serial-parallel connection of FSM Ds

FIGURE 7(a, b, c, d) Hierarchical FSM D

**VHDL Code for FSM D A**
(with synchronous reset)

```
process
begin
wait until clk'event and clk=1;
if(Reset=1) then
  Input_reg <= 0
  Output_reg <= 0
  Status_reg <= "011"
else
  case State is
    when Sa1 =>
      ... --reg. transfers
      DataOut <= Output_reg
      Ready <= 1;
      if(Ack=1) then
        State <= Sa2;
      else
        State <= Sa1;
      end if;
    when Sa2 =>
      ... --reg. transfers
      DataOut <= X
      Ready <= 0;
      State <= Sa3;
      ...
  end if
end process
```

VHDL Code for FSM D B
(with synchronous reset)

```
process
begin
wait until clk'event and clk=1;
if(Reset=1) then
  Input_reg <= 0
  Output_reg <= 0
  Status_reg <= "001"
else
  case State is
    when Sb0 =>
      ... --reg. transfers
      Ack <= 0;
      if(Ready=1) then
        State <= Sb1;
      else
        State <= Sb0;
      end if;
    when Sb1 =>
      ... --reg. transfers
      Input_reg <= DataIn
      Ack <= 1;
      if(Ready=0) then
        State <= Sb2;
      else
        State <= Sb1;
      end if;
    when Sb2 =>
      ... --reg. transfers
      Ack <= 0;
      State <= Sb3;
      ...
  end process
```

(a) with synchronous reset

FIGURE 8 VHDL model for FSM D resetting

**VHDL Code for FSM D A**
(with asynchronous reset)

```
process
begin
wait until (clk'event and clk=1) or (Reset'event and Reset=1);
if(Reset=1) then
  Input_reg <= 0
  Output_reg <= 0
  Status_reg <= "011"
else
  case State is
    when Sa1 =>
      ... --reg. transfers
      DataOut <= Output_reg
      Ready <= 1;
      if(Ack=1) then
        State <= Sa2;
      else
        State <= Sa1;
      end if;
    when Sa2 =>
      ... --reg. transfers
      DataOut <= X
      Ready <= 0;
      State <= Sa3;
      ...
  end if
end process
```

VHDL Code for FSM D B
(with asynchronous reset)

```
process
begin
wait until (clk'event and clk=1) or (Reset'event and Reset=1);
if(Reset=1) then
  Input_reg <= 0
  Output_reg <= 0
  Status_reg <= "001"
else
  case State is
    when Sb0 =>
      ... --reg. transfers
      Ack <= 0;
      if(Ready=1) then
        State <= Sb1;
      else
        State <= Sb0;
      end if;
    when Sb1 =>
      ... --reg. transfers
      Input_reg <= DataIn
      Ack <= 1;
      if(Ready=0) then
        State <= Sb2;
      else
        State <= Sb1;
      end if;
    when Sb2 =>
      ... --reg. transfers
      Ack <= 0;
      State <= Sb3;
      ...
  end process
```

(b) with asynchronous reset

FIGURE 8 VHDL model for FSM D resetting



7 CONCLUSION

This report presented basic concepts in RTL design and attempted to explain RTL methodology. It is the purpose of this report to familiarize readers with basic concepts explaining the pros and cons behind the devices and to serve as the accompanying document for the shorter, more formal document for RTL semantics standard.

8 ACKNOWLEDGEMENT

The authors would like to thank all the members of Accellera working group who contributed their comments to improve the quality of this report. Further we would like to thank graduate students at Center for Embedded Computer Systems who read the document and made many suggestion for its improvement. Particularly, we would like to thank Andreas Gerstlauer and Shuqing Zhao for developing and testing VHDL and SpecC models for examples included in this report.

REFERENCES:

1. John F. Wakerly; Digital Design: Principles and Practice, Prentice Hall, 2000
2. Randy H. Katz; Contemporary Logic Design, Benjamin/Cummings, 1994
3. Daniel Gajski; Principles of Digital Design, Prentice Hall, 1997