Modeling Custom Hardware in VHDL

Technical Report ICS-99-29
July 6, 1999

Heiko Lehr

Supervised by Prof. Daniel D. Gajski

Research Paper "Studienarbeit"

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

Abstract

This report focuses on models for describing Hardware at different refinement levels within High Level Synthesis flow: SFSMD, FSMD and FSM Controlling Datapath. Simple data interfaces are often needed in these models. The hardware description language VHDL is used and tested for implementing these models and interfaces. Problems inherent in the models as well as problems caused by VHDL are discussed. A model related main difficulty is the one-cycle delay of data processing, when introducing a datapath. VHDL descriptions of abstract models (SFSMD) can become complicated. Templates for general problems, examples and VHDL guidelines are provided in this report to help to minimize design errors significantly.

Contents

List of Figures

Modeling Custom Hardware in VHDL

Heiko Lehr

Abstract

This report focuses on models for describing Hardware at different refinement levels within High Level Synthesis flow: SFSMD, FSMD and FSM Controlling Datapath. Simple data interfaces are often needed in these models. The hardware description language VHDL is used and tested for implementing these models and interfaces. Problems inherent in the models as well as problems caused by VHDL are discussed. A model related main difficulty is the one-cycle delay of data processing, when introducing a datapath. VHDL descriptions of abstract models (SFSMD) can become complicated. Templates for general problems, examples and VHDL guidelines are provided in this report to help to minimize design errors significantly.

1 The meaning of the models

In order to discuss High Level Synthesis, different specification models are important.

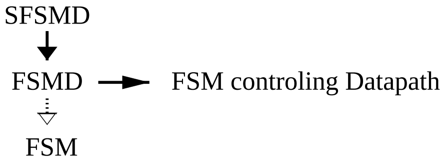


Figure 1: Models for HL-Synthesis

Figure 1 shows the refinement flow of the models. The models used for high level synthesis are connected by the solid arrows. The FSMD to FSM flow is not useful (indicated by a dotted arrow). The horizontal arrow indicates, that the connected models are on the same refinement level. They are related strongly, but their ideas and possibilities for usage are differently.

Theoretically, each model could be described behaviorally and structurally. However depending on the model, such a description may be impractical or even meaningless because of an exploding size or complexity of the description.

Figure 2 shows the system exploration chart (Y-Chart). It consists of refinement levels (circles) and description domains (lines). The significant description models are marked in the chart.

The following sections explain these models. Additionally, description models of little or no use are mentioned briefly.

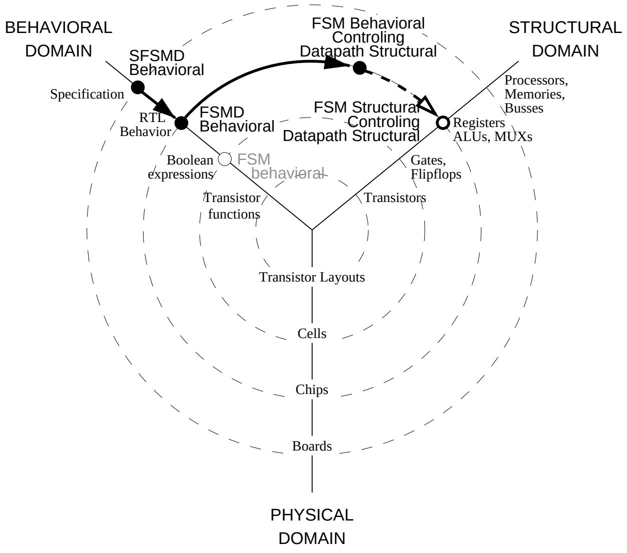


Figure 2: System Exploration Chart

1.1 SFSMD

The *Super Finite State Machine with Datapath (SFSMD)* is the most abstract level of description.

The whole algorithm can be considered to be a SFSMD with one superstate. But generally, the algorithm can be divided into any number of parts of any size. These parts are the superstates.

The algorithm parts consist of reading inputs (I), writing outputs (O) and executing expressions (EXP), which use intermediate storage variables (V). Using these elements, SFSMD can be described syntactically in the same way as FSMD. (See FSMD below.) The basic difference between the two models is, that SFSMD does not restrict the size of the algorithm-parts assigned to a state, whereas FSMD does. This is, because SFSMD does not correspond to hardware at all, whereas the states of FSMD correspond to clock cycles.

1.1.1 Behavioral Description

For *behavioral description*, the algorithm which is chosen to be implemented in hardware must be correct and executable on any platform, but no restrictions concerning a special structure apply. The whole algorithm could be assigned to one single big superstate (figure 3). But in most cases several superstates are recommended. When describing the algorithm in a high level programming language, it is typical to map the procedures to superstates.

Figure 4 shows an example with several superstates. Like in every state machine, in SFSMD a superstate can have several possible successors. The valid one is determined by transition conditions as shown in state *Part 3* of the figure. The left-hand side "cond" can be a variable, but it also could be a function.

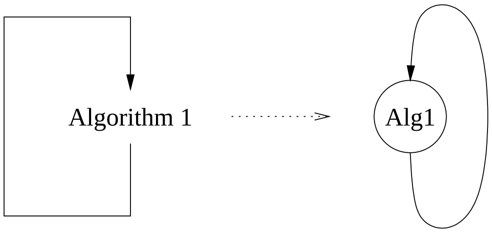


Figure 3: One-State-SFSMD

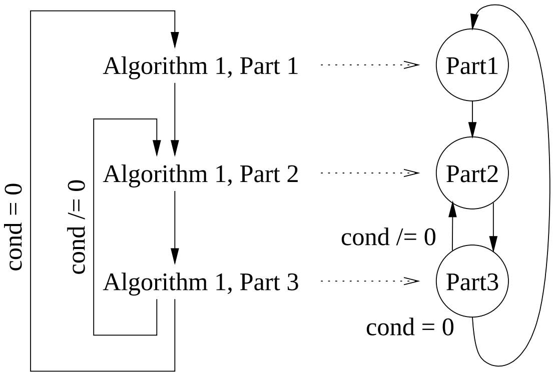


Figure 4: SFSMD

1.1.2 Structural Description

A *structural description* is *not used* for this model. Hardware allocation and scheduling are not done at this point (see section 2). Therefore a structural description just could be something like a general purpose computer which is capable of executing the appropriate program which is stored in some memory (Y-Chart, fig. 2: structural domain, outer circle). This is not helpful for implementing a specialized algorithm.

1.2 FSMD

In the model *Finite State Machine with Datapath (FSMD)* scheduling is performed (see section 2). That means, algorithm is divided into small pieces which are assigned to cycles using cycle based states. FSMD-cycles will be mapped to the cycles of the hardware clock. However hardware cycle time is not finally set yet. There may also be a multiplication factor in this mapping (e.g. each FSMD cycle lasts two clock cycles).
In each state two things are done:

- The operation scheduled to this state is executed within one clock cycle.
- The next state for the next clock cycle is determined.

For an exact description of FSMD, some sets have to be defined:

S: set of *states*
I: set of *inputs*
V: set of *storage variables*
O: set of *outputs*
EXP: set of *expressions*:
functions which give results depending
on *storage variables* V and *operators* OP:
$EXP = \{V, OP\} = \{f(x, y, z, \ldots) | x, y, z \in V\}$
OP: set of operators used in EXP
$STAT$: set of *status expressions*:
logic relations between
two expressions from the set EXP:
$STAT = \{Rel(a, b) | a, b \in EXP\}$

Referring to the above definitions, the data processing (first item in above list) is described by function h:

$$h : S \times (I \cup STAT \cup EXP) \rightarrow (V \cup O); \qquad V0, S0$$

There needs to be a set of initializing values $V0$ for the variables V when starting the FSMD, because the expressions in EXP read them. There also has to be

an initial value $S0$ for the state.
The next state (item 2 in above list) is determined in a similar way:

$$f : S \times (I \cup STAT) \rightarrow S; \qquad V0, S0$$

$V0$ and $S0$ are needed again, because $STAT$ depends on EXP, and EXP depends on V.
With these definitions, a FSMD is described by:

$$< S, S0, (I \cup STAT \cup OP), V, V0, O, h, f >$$

1.2.1 Behavioral Description

The *FSMD Behavioral description* is shown in figure 5. Behavioral description is preferred for this model, because of better readability. It clearly illustrates how the code pieces are scheduled in the states. It is located at the RTL Behavior point of the Y-Chart. The term *"RTL" (Register Transfer Level)* indicates the *cycle accuracy* of this description and the corresponding register transfers of the final hardware:
Each variable used in this description will be assigned to elements of the final hardware (register, memory, input, output, delay-wire). Several variables may be assigned to the same element as long as their lifetimes do not overlap. However, the assignment itself is not done in this description. It is just assumed that each variable will have a location somewhere in the final hardware.

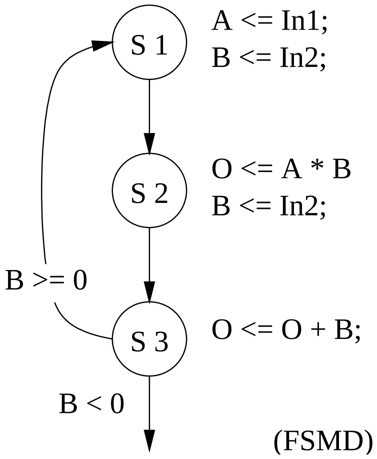


Figure 5: FSMD Behavioral

1.2.2 Structural Description

A *structural description* is *not used* for this model. The *structural description* would look like figure 6, except that control and datapath are not separated into distinct blocks.

The following model is more suitable for a structural description.

1.3 FSM Controlling Datapath

Here, the hardware is described on the same refinement level as the previous model. In contrast to FSMD, there is splitting into a *control*-part, described by a *Finite State Machine (FSM)* without datapath, and a separate datapath now.

The control block controls the execution of the operations while the datapath actually performs them. That means, compared to FSMD, the computation of the expression $STAT$ and EXP are done in the datapath. Storage variables V, inputs I and outputs O are also covered by the datapath.
For the control block, the set of states S remains. Additionally, there are connections between control block and datapath:

C: Control signals (output of control FSM)
M: Message signals (input of control FSM)

The functions for the datapath look similar to the FSMD model. The difference is, that message and control signals are used instead of the state:

$$h_M : C \times (I \cup STAT \cup EXP) \rightarrow (V \cup O); \qquad V0, C0$$

$$f_M : C \times (I \cup STAT) \rightarrow M; \qquad V0, C0$$

$$< C, C0, (I \cup STAT \cup OP), V, V0, O, h_M, f_M >$$

The control block defines the relationships among message signals M, control signals C and states S:

$$h_C : S \times M \rightarrow C$$

$$f_C : S \times M \rightarrow S$$

$$< S, M, C, h_C, f_C >$$

Depending on the current state, the control block sets control signals for the datapath. These signals tell the datapath which data are to be processed and how they are to be processed. The datapath can inform the control block about special results via message signals. See figures 6 and 8. Depending on the message signals, the control block determines the next state which becomes valid in the next clock cycle.

The separation into these two blocks offers new possibilities: Each can be independently described behaviorally or structurally.

1.3.1 Behavioral Description

A *pure behavioral* description of this model is *not used.* When trying to describe this, there would be a second state machine for the datapath. This would cause lot of trouble:
If NC is the number of control signals from the control block to the datapath, then $T = 2^{NC}$ is the number of different conditions for state transitions, which is an "exploding" expression.
Conclusion: A behavioral description of a separate datapath is not suitable. The model FSMD is used instead.

1.3.2 Behavioral-Structural Description

The big advantage of the split model is the possibility of still using a behavioral description for the control, while describing the datapath structurally. See figure 6. The first block illustrates the states in accordance with the Register Transfer Level while the second contains the data structure: memory, registers, ALU and so on, as well as their connections and data input- and output ports. For example, a multiplication is implemented by the ALU and the result is stored in its output register. The ALU consists of pure combinational logic. The command signals from the FSM control the datapath: `ALU = "11"` chooses multiplication and `ld_O = '1'` stores the result in register O.

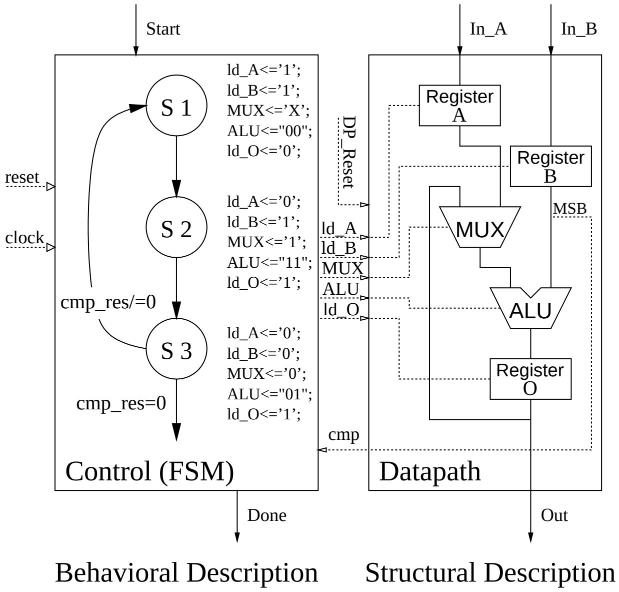


Figure 6: FSM Controlling Datapath (1)

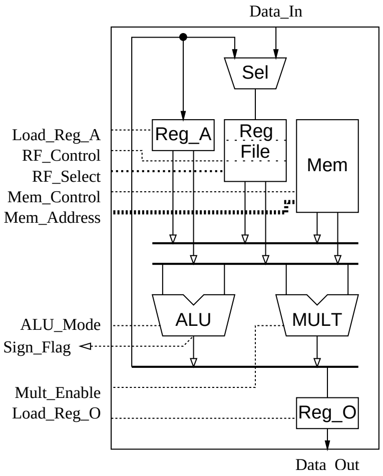


Figure 7: Multi purpose datapath

This structural representation also illustrates the possible data flows. In figure 6, variables A and B can be multiplied by the ALU and the result can be stored in variable O which resides in register O.

Instead of the very specific datapath in figure 6, also a more general datapath can be used, like the one in figure 7. A general datapath often avoids the necessity of creating or changing the datapath when the control FSM is replaced or changed. On the other hand, a general datapath is less optimized and may contain unused overhead.

The huge amount of combinational logic is represented well in a datapath picture. For example, multiplier and comparator are purely combinational. That means, they do not use the clock and they do not contain any latches.

This pure structural description grows just linearly when inserting additional components. The size of a *behavioral* datapath description would have exploded. There would also be complicated transition conditions due to the huge amount of combinational logic.

In the control block, the situation is different: There are no "control data" which are modified and then passed to other operations. There is exactly one variable and it has special meaning: the state variable. It needs to be *written and* to be *read.* As its value is the basis for cycle accurate control, it has to be defined by a *FSM* in each clock cycle. For pointing out the state transitions and the state based interaction with the datapath, the behavioral description on the left-hand side in figure 6 is the best representation.

It is to say, that the split model with the control part described behaviorally and the datapath described structurally is more difficult to read by humans than a behavioral description of the model FSMD. However, this approach to the final hardware has to be taken. Physical hardware finally is obtained by implementing a structural description. Therefore we finally need a pure structural description. The datapath, which is by far the largest part of the design, is structural already. The behavioral control description is simple and can be converted to a structural description by a simple tool.

Conclusion: The split model with behavioral control and structural datapath is a good basis for synthesis tools which have some intelligence about how to convert behavioral description into structure. The model is still readable by humans.

1.3.3 Structural Description

Synthesis tools with no intelligence may need a pure *structural description.* In figure 8, even the control part is structural. The control block shown here, implements the control FSM, too. However it reveals neither the states nor the associated actions any longer but hides them within the logic blocks. Even looking at the internal description of the logic blocks, pure structure is quite less readable.

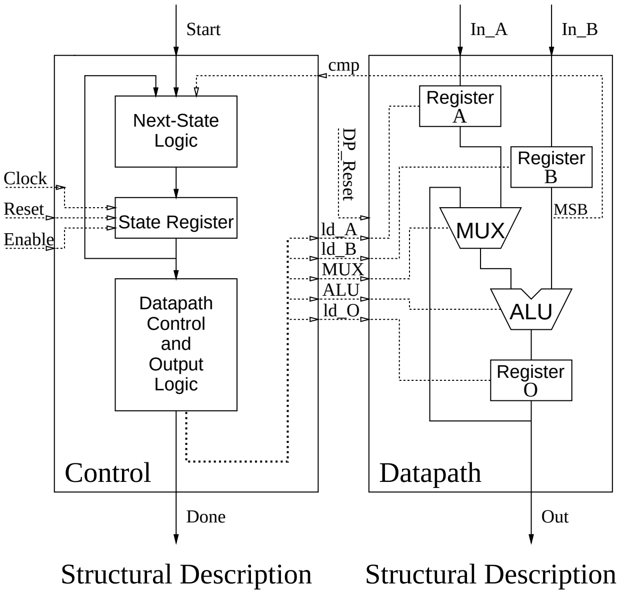


Figure 8: FSM Controlling Datapath (2)

Conclusion: A pure structural description should be used only if the synthesis tool or the input language does not permit a behavioral description of the control block.

1.4 FSM

A *Finite State Machine (FSM)* without Datapath for describing the whole design (and not just the control) causes an explosion of the number of states:
Using no datapath means that for all possible data a separate state is needed. For example, a 16-bit integer variable causes 65536 different states. This has to be done for all other variables, too. As there is no datapath, a formal description of this model is simple. There are no storage variables and expressions using them. The only value which is stored is the state.
The functions for the next state and output are simplified to:

$$h : S \times I \rightarrow O$$

$$f : S \times I \rightarrow S$$

The FSM defining tuple is:

$$< S, I, O, h, f >$$

It is obvious, that the model FSM can only be used for problems which are dominated by boolean variables. They should not use variables which are larger than about 3 or 4 bits. Thus, this model is only suited for control algorithms, which are a special case. The FSM within the model "FSM Controlling Datapath" is an example.
However, for the general case, which includes data processing, this model can not be used reasonably.
Note: Synthesis tools may use this model for a hidden, automatically generated intermediate format in order to further refine the synthesis flow. But this is only necessary if low-level IPs (Intellectual Properties) like adder, multiplier and so on can not be provided as plug-ins for the datapath (see previous model).

1.5 Conclusion

The above discussion points out the following models and descriptions. They are the best in most cases for performing High Level Synthesis with machine-human interaction.

1. SFSMD Behavioral

   (a) One-state SFSMD
   (general mathematical description or program)

(b) Multi-state SFSMD
(problem separation, structured program)

2. FSMD Behavioral

3. FSM Behavioral Controlling Datapath Structural

4. (FSM Structural Controlling Datapath Structural, if needed)

2 Scheduling, Allocation and Binding

The assignment of code slices to states is called *scheduling*. A slice of code from a FSMD Behavioral description should be as small to be executable within one cycle.

Exceptions can be made for some complex computations which cannot meet this constraint. They can also use multiple states.

Parallel execution of several incremental code pieces within one clock cycle can reduce the number of needed cycles and thus speed up the design. However there are two problems: One problem are the data dependencies. The simplest example is the impossibility of computing two values in parallel, if one is needed as an input for computing the other one. Another constraint is, that the datapath should not get too large exceeding restrictions in size or cost. For example, parallelizing two multiplications causes two multipliers in the datapath which significantly increases complexity.

So, when scheduling is performed for the behavioral description of the FSMD model, the *allocation* of data and actions partially should be planed already. It is accomplished in the structural description of the datapath however. The control block for the datapath points out the real requirements in a much more obvious way. The datapath has to be equipped with appropriate registers, memory and so on. This is the real execution of allocation.

Thereafter, *binding* has to be performed. The control block is adjusted to assign data to the right registers, correct memory addresses and other components.

3 VHDL Programming Structure

Generally, the "programming" language VHDL can be used for all the models and descriptions: Behavioral description of SFSMD, FSMD and for the description of a behavioral FSM controlling a structural datapath.

This section is a summary about the structure of VHDL programs. Those, who know VHDL well, may skip this section and proceed with section 4. For others, it helps to remember the structure of VHDL programs and to understand the following sections and examples better. However, this section does not teach VHDL.

Figure 9 shows a simple general template for the structure of VHDL-code from which the description of the models will be derived. Some optional parts are omitted, because they are seldom used and not needed here.

VHDL is block-oriented and in that way uses hierarchy. The hierarchy structure usually looks like the one shown in figure 7: A testbench forms the top level. It instantiates the design block which may have several subblocks. Subblocks can have subblocks themselves.

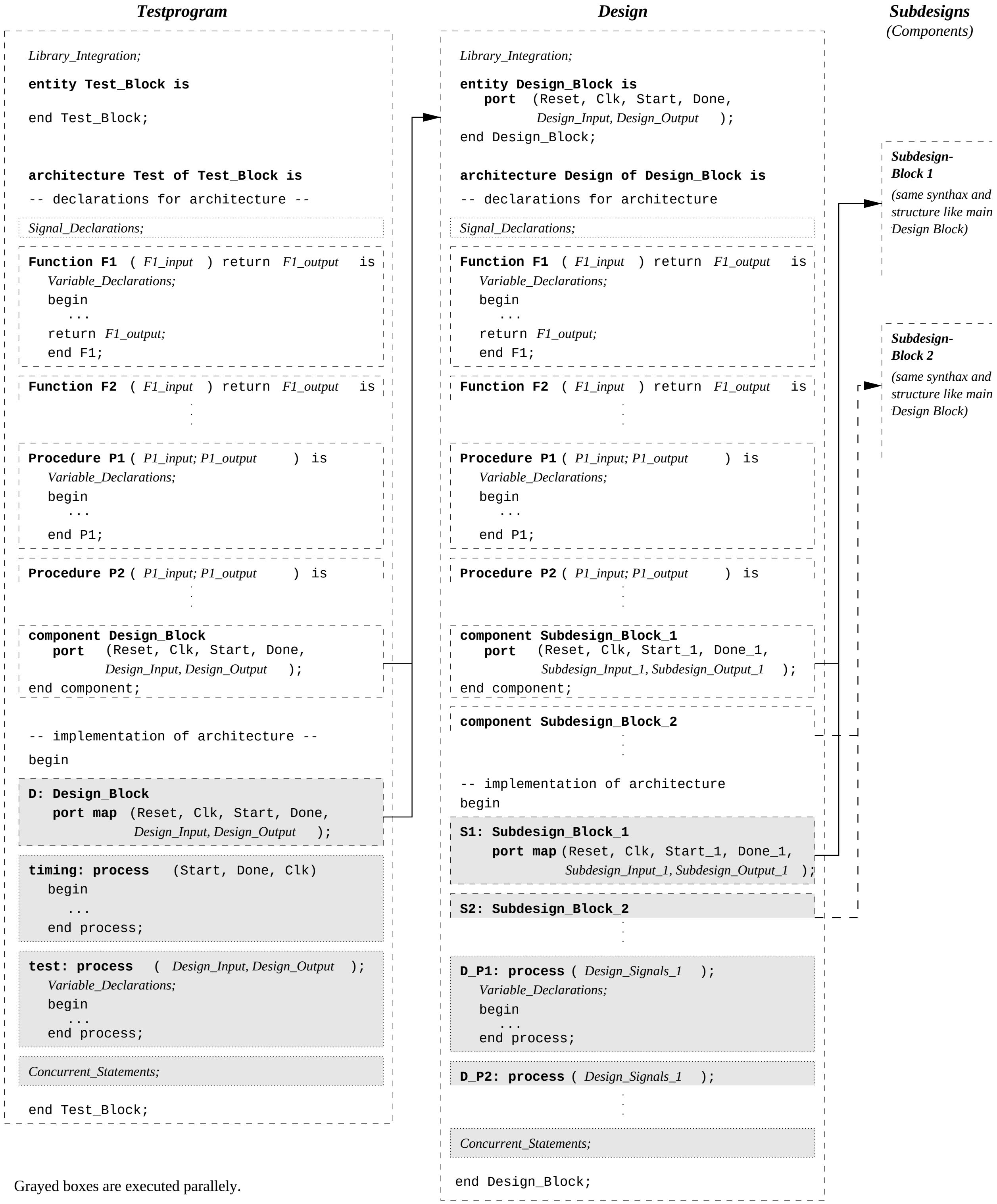
Except the testbench, the other blocks need to have data ports. The ports of the top level design block implement the inputs and outputs of the algorithm. The are independent of the currently used model and description. Therefore VHDL separates them: The *entity* covers the port declarations whereas the *architecture* implements the computation part.

The figure shows that the architecture consists of a declaration part and an implementation part. The latter one contains processes, subblock-instantiations and concurrent statements. They are executed parallelly, indicated by grayed boxes in the figure 9. Simple concurrent statements usually are a simple logic block. Processes behave like infinite loops of code. They temporarily have to be suspended before each run of the loop. This can be done by wait commands or by a so-called *sensitivity list*. Figure 9 uses sensitivity lists in the process heads, causing the process to wait on a change of the listed signals.

Subblock-components, functions, procedures and additional internal signals have to be declared in the declaration part before they can be used in the implementation part.

A process can also use its own local variables, declared inside the process which uses them (schematically indicated by *"Variable_Declaration"*).

This causes the question whether to declare a global signal or a local variable for holding a value.



Testprogram

```
Library_Integration;

entity Test_Block is

end Test_Block;


architecture Test of Test_Block is
-- declarations for architecture --
Signal_Declarations;
Function F1  ( F1_input  ) return  F1_output   is
   Variable_Declarations;
   begin
     ...
   return  F1_output;
   end F1;
Function F2  ( F1_input  ) return  F1_output   is
                          .
                          .
                          .
Procedure P1 ( P1_input; P1_output      )  is
   Variable_Declarations;
   begin
     ...
   end P1;
Procedure P2 ( P1_input; P1_output      )  is
                          .
                          .
                          .
component Design_Block
   port   (Reset, Clk, Start, Done,
          Design_Input, Design_Output    );
end component;

-- implementation of architecture --
begin
D: Design_Block
   port map  (Reset, Clk, Start, Done,
             Design_Input, Design_Output     );
timing: process   (Start, Done, Clk)
   begin
     ...
   end process;
test: process   (  Design_Input, Design_Output    );
   Variable_Declarations;
   begin
     ...
   end process;
Concurrent_Statements;

end Test_Block;
```

Design

```
Library_Integration;

entity Design_Block is
   port  (Reset, Clk, Start, Done,
          Design_Input, Design_Output    );
end Design_Block;

architecture Design of Design_Block is
-- declarations for architecture
Signal_Declarations;
Function F1  ( F1_input  ) return  F1_output   is
   Variable_Declarations;
   begin
     ...
   return  F1_output;
   end F1;
Function F2  ( F1_input  ) return  F1_output   is
                          .
                          .
                          .
Procedure P1 ( P1_input; P1_output      )  is
   Variable_Declarations;
   begin
     ...
   end P1;
Procedure P2 ( P1_input; P1_output      )  is
                          .
                          .
                          .
component Subdesign_Block_1
   port   (Reset, Clk, Start_1, Done_1,
          Subdesign_Input_1, Subdesign_Output_1   );
end component;
component Subdesign_Block_2
                          .
                          .
                          .
-- implementation of architecture
begin
S1: Subdesign_Block_1
    port map (Reset, Clk, Start_1, Done_1,
             Subdesign_Input_1, Subdesign_Output_1   );
S2: Subdesign_Block_2
                          .
                          .
                          .
D_P1: process ( Design_Signals_1    );
   Variable_Declarations;
   begin
     ...
   end process;
D_P2: process ( Design_Signals_1    );
                          .
                          .
                          .
Concurrent_Statements;

end Design_Block;
```

Subdesigns
(Components)

Subdesign-Block 1
(same synthax and structure like main Design Block)

Subdesign-Block 2
(same synthax and structure like main Design Block)

Grayed boxes are executed parallely.

Figure 9: General VHDL program structure

- If the value is needed outside the process, it can only be held by a signal.

- Sequential computations within one cycle need variables. Sequential computations means: A process *writes* a value. Thereafter, the same process *reads* this updated value in the same cycle.

  Signals own the property of having a *delta-delay*, which delays the update until the next cycle starts. Variables do not have this property and therefore can be used in this case.

- Otherwise:
  Declare a signal, if a physical equivalent (physical signal or wire) in the final hardware is expected.
  Declare a variable, if the description is too abstract to have an obvious correspondence to hardware (variables in SFSMD).

An exception to the rule of leaving the entity unchanged is needed: When switching the models for the algorithm to deeper refinement levels, some synchronization signals like *start*, *done* and *clk* (clock) need to be added in the entity. They are inevitable ports of the final, hardware but they are not *data* ports.

In the figure, the words *"Design_Input"* and *"Design_Output"* are abbreviations. They stand for the range of all inputs and outputs of the design.

4 Describing the Models in VHDL

Section 1 gave a view of the meaning of the hardware description models. The following sections explain how to model them in VHDL and discuss some significant problems.

4.1 Abstract and Introduction

4.1.1 The State Machines Graphically

The graphical representation of the state machines in figures 10 and 11 is according to the following rules:
The actions are attached to the states they are assigned or scheduled to. The kind and size of the actions is dependent on the model. The new data results caused by the actions are not available before the next state transition.

Among the arrows starting at the current state there must be exactly one with a true condition. Thisone determines the next state. All possibilities have to be covered by the conditions and ambiguity is not allowed.
Whereas state transitions are bound to conditions, actions are bound to states only but not to conditions. Such state machines are called *Moore Machines.*

4.1.2 Translation into VHDL

The right-hand side of figures 10 and 11 show possible VHDL descriptions, which implement the graphical representation of the right-hand side. Figure 10 shows how to implement a state machine inside a procedure. Figure 11 describes the implementation inside a process. Both versions use the VHDL-case statement for modeling the state machine.

In the figures, the typewriter font is used for direct VHDL code. Proportional font is used for parts which can either be a procedure (resp. function) or an abbreviation for a piece of code, that is:

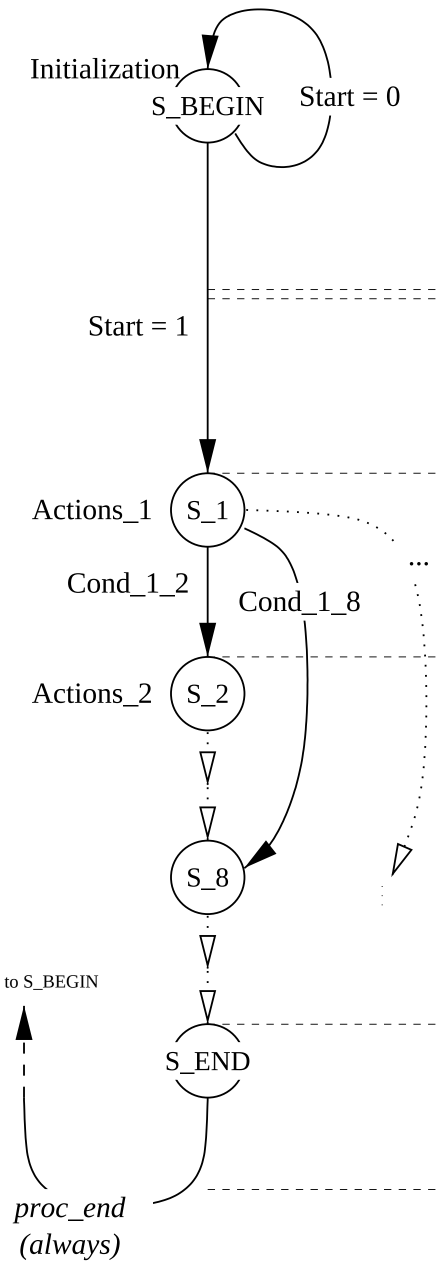
Actions_i:
Parts of the algorithm assigned to the state i.
Actions_i can be a piece of code or a procedure. It reads the signals/ variables represented by *IN_i* and writes the variables represented by *OUT_i.*

Cond_i_j:
Condition for the state transition from state i to j.
Cond_i_j may be either a function or a piece of code. It reads variables and signals from *IN_i_j.* The result value of the condition is represented by its name itself.

Next_State_i:
Sets the value of `state` after state i.
Next_State_i is an expression or a procedure.
In the used syntax, it has the following parameters:
inputs: list of conditions, list of possible next states;
output: next state (`STATE`).

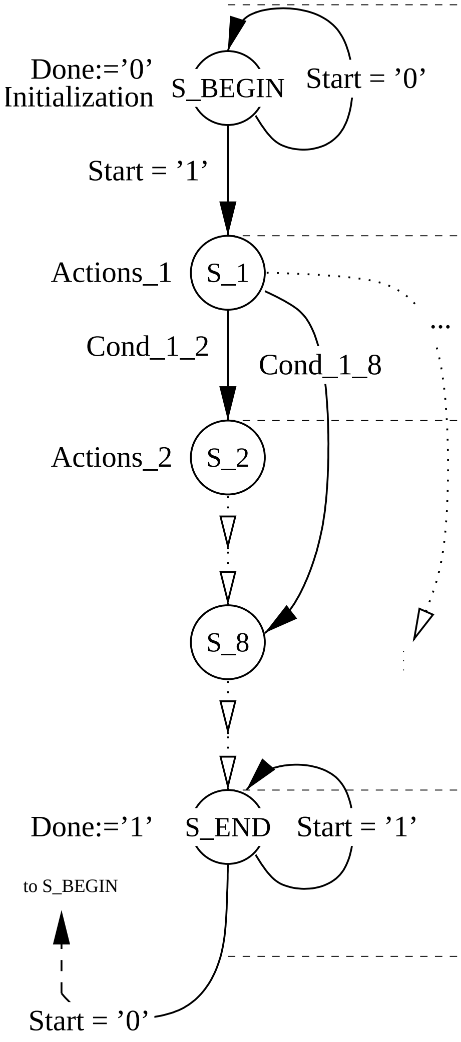
Procedure versus Process. *The "procedure version" should be used for models at a high level of abstraction (SFSMD) and for models which do not use timing (SFSMD, FSMD Without Time).*
Procedures are well-suited for sequential programming style with preferred use of variables instead of signals. This especially meets the requirements of SFSMD. Additionally, SFSMDs can be simply concatenated by calling one procedure after the other. This means, no allusions to a hardware protocol are needed.

A process on the other hand does not have these advantages. Additionally, a process is an endless loop which is developed for use with a clock.



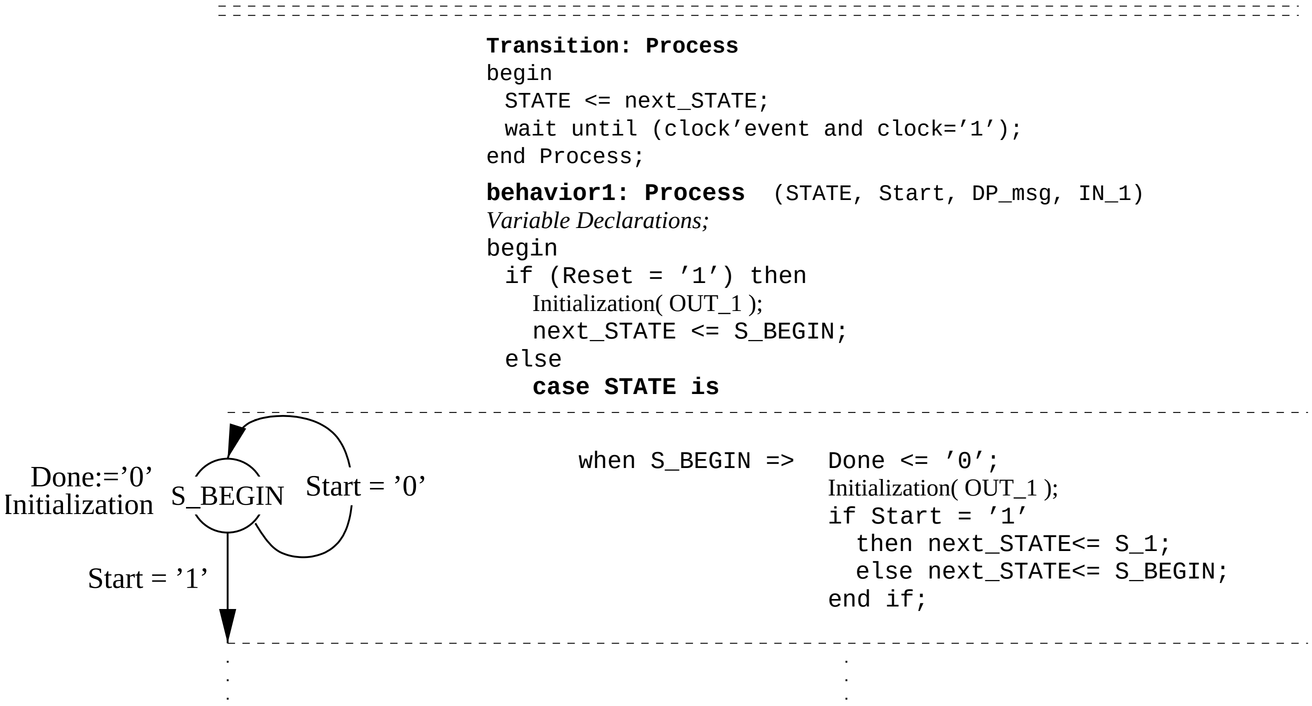
```
-- S_BEGIN == "procedure not running", managed by the calling process:
P1: Process;
Variable Declarations;
begin
   wait until (Reset=’1’ or Start=’1’)
   if (Reset=’1’) then
     Output1 <=... ;
   elsif (Start = ’1’) then
     behavior1(   Input1; Output1   );
   end if;
end process;

Procedure behavior1(  in Input1; inout Output1   );
Variable Declarations;
STATE := ’1’;
while (STATE /= S_END) and (Reset/=’1’) loop
   case STATE is

      when S_1 =>     Actions_1( IN_1, OUT_1 );
                      Next_State_1(  Cond_1_2(IN_1_2),
                                     S_1,                 S_8,
                                     STATE    );

      when S_2 =>     ...

      when S_END =>    -- nothing
                       -- S_END  -> end of "while"  -> end of procedure

   end case;
end loop
end behavior1;
```

Figure 10: A state machine inside a procedure



```
behavior1: Process(Clock)
Variable Declarations;
begin
 if (Clock’event and Clock=’1’) then
   if (Reset = ’1’) then
     Initialization( OUT_1 );
     next_STATE <= S_BEGIN;
   else
     case next_STATE is

        when S_BEGIN =>  Done <= ’0’;
                         Initialization( OUT_1 );
                         if Start = ’1’
                           then next_STATE<= S_1;
                           else next_STATE<= S_BEGIN;
                         end if;

        when S_1 =>      Actions_1( IN_1, OUT_1 );
                         Next_State_1(  Cond_1_2,   Cond_1_8,    ...
                                        S_2,        S_8,         ...
                                        next_STATE   );

        when S_2 =>      ...
            :
            :
            :

        when S_END =>  Done <= ’1’;
                       if Start = ’0’
                         then next_STATE<= S_BEGIN;
                         else next_STATE<= S_END;
                       end if;

      end case;
    end if;
  end if;
  end process;
```

Figure 11: A state machine inside a process



```
Transition: Process
begin
  STATE <= next_STATE;
  wait until (clock’event and clock=’1’);
end Process;
behavior1: Process  (STATE, Start, DP_msg, IN_1)
Variable Declarations;
begin
  if (Reset = ’1’) then
    Initialization( OUT_1 );
    next_STATE <= S_BEGIN;
  else
    case STATE is
        when S_BEGIN =>  Done <= ’0’;
                         Initialization( OUT_1 );
                         if Start = ’1’
                           then next_STATE<= S_1;
                           else next_STATE<= S_BEGIN;
                         end if;
  .
  .
  .
```

Figure 12: A state machine for controlling a separate datapath

The ”process version” is preferable if the model uses a clock (FSMD With Clock, FSM controlling datapath).

Although a procedure could be used with wait statements, this is not recommended. Such a bad style of hidden and scattered wait statements could confuse humans and synthesis machines. According to that, the even worse way of using wait statements inside a function is forbidden by the language already. Thus processes are better for use with a clocked description. The use of a sensitivity list in the process heads expose the wait conditions very well. In figure 11 the list has one entry: Clock.

Special Case: State Machine and Separate Datapath. Figure 12 shows some changes to figure 11: The activation of the next state is done by a newly introduced process. The second process, which contains the state machine, is made sensitive to the state and all input signals (`process(STATE,Start,DP_msg,IN_1)`). This modification is needed for a model which uses a separate datapath. It compensates delays which are caused by the datapath: In contrast to a single state machine with ”build-in” datapath, a separate control-FSM just gives commands and the actual computations are executed by the datapath one cycle delayed.

The modification in figure 12, however, compensates this: The clock has been removed from the state machine. Instead, datapath feedback messages now appear in the sensitivity list of the state machine’s process. Thus the state machine can react to changes on these signals without waiting for the next clock cycle. In this way the delay is ”compensated”. See section 4.2.6 for more.

4.2 Examples and Guidelines

This section shows how to use and code the models with VHDL with the help of examples. Examples are much clearer than the abstract descriptions of the previous section. The examples can be used like templates. Figures 13 to 17 show the descriptions of the state machines in either a procedure or a process. Figures 20 to 23 show their embedding into a complete VHDL description. In this latter set of figures, the declaration of all variables and signals, needed for the examples, can be seen too.

Important rules are generalized and described. Executable files of the examples are available and listed in the appendix.

4.2.1 SFSMD Behavioral

Figure 13 shows a very simple SFSMD: The superstates S_1 and S_2 perform expensive power operations each. In superstate S_3, the results are subtracted.
This is just one way of describing the algorithm. The abstract model SFSMD provides a lot of freedom:

- The three superstates in the example may be merged into a single one.
- Actions and conditions (conditions are not used here, but see figure 10) can be of any complexity.
- In this example even the loop and the case statement may be omitted, because there is just a sequential computation:

```
...
begin

-- State S_1:
-- S:= In1 ** In2;   -- (power)
S:= 1;
for i in In2 downto 1 loop
   S:= S * In1;
end loop;

-- State S_2:
-- R:= In3 ** In4;   -- (power)
R:=1;
for i in In4 downto 1 loop
   R:= R * In3;
end loop;

-- State S_3:
-- Output
Out1:= S - R;

end behavior1;
```

Rules. Almost all constructs of the programming language are allowed for use in the description of the SFSMD model, but the following rules have to be obeyed when modeling an SFSMD in VHDL:

- *All inputs and outputs have to be declared clearly.*

  In VHDL, do not introduce "unofficial" data ports by reading from or writing to a file within a procedure. All inputs and outputs should be declared in the entity.

- *Data ports have to be read and set accurately. Other ports (control, timing) should be avoided.*

  An abstract description never means, that abstract results are allowed. To be a valid model, it must occupy the data ports completely in accordance to the specifications, except that time is not handled. That means outputs are set without delay.

  Synchronization ports and other timing related ports (like the clock input) are implementation specific. They are omitted in this model because this one is abstract and non-timing accurate.

- *In this model parallelism and timing are not used.*

  This means, in VHDL use only one process and prefer to use variables instead of signals.

  Use of several parallel processes would mean switching to the model CHSFSMD *(Concurrent Hierarchic SFSMD)*. In that model hierarchically higher processes start other processes parallelly. But this is not discussed here.

  Signals should not be used because signal assignments used without appropriate timing delays are not executed in the desired sequential order (see section 3). It is difficult and dangerous to use signals in this model, because a large amount of sequential code is assigned to each superstate. The state is a variable, too.

4.2.2 FSMD Behavioral

There are two variants of description for the model FSMD: *FSMD Behavioral Without Time* (figure 14) and *FSMD Behavioral with clock* (figure 15).

Timing and synchronization originally do not belong to this model, but introducing clock at this level often is an advantage when using VHDL. (See section "Discussion" below.)

The VHDL-implementations of both behavioral variants look similar: In this example, figures 14 and 15 implement superstate S_1 of the SFSMD of figure 13. An FSMD also may implement several superstates.

Steps. The steps to convert a SFSMD into a FSMD are the following:

- *Divide the SFSMD into slices which are meant to become implemented by a FSMD each.*

  A FSMD can be the implementation of just one superstate up to the whole SFSMD. After testing and/or refining into the model "FSM Controlling Datapath", the state machines can be joined together again by concatenating the states (and merging the datapaths). Alternatively the state machines can remain separate when adding communication (see section 5).

- *Choose way of description: FSMD Without Time or FSMD With Clock.*

  Usually the description with clock is preferable, when modeling in VHDL (see section "Discussion" below).

  According to section 4.1.2, FSMD Without Time should be implemented in a procedure (figure 14). FSMD With Clock has to be implemented in a process (figure 15).

  Version without time in a procedure:

  - There doe not need to be a "when"-statement for the state `S_BEGIN` in this model.
    The reason is the sequential programming style: The state machine is alive only for the time it is needed, else it does not exist. If the code of the state machine is not executed, this means that it rests in the state S_BEGIN.
  - Inside the procedure only variables are used (assignment operator "`:=`").
    (However, signals become inevitable in communicating state machines.)
  - The simple data types are adopted from the SFSMD-model.
    The example uses "integer" for instance.

  Version using clock in a process:

  - The states `S_BEGIN` and `S_END` as well as "start"- and "done"-signals are needed for synchronization.
    In a process, a state machine is always alive. The state machine waits in state `S_BEGIN` for its "start"-signal. When the machine finishes, it reaches the state `S_END` and sets a "done" signal.
  - In the process, signals are used generally. (Variables may be used exceptionally, if they are not expected to have a corresponding wire or register in the final hardware, e.g. variables which model delay.)
  - The hardware related data type **`std_logic_vector`** is used in the example.
    This seems likely because this description uses clock which is close to hardware, too. Both aspects provide the possibility of easy translation into the model "FSM Controlling Datapath".

- Schedule the operations:
  A superstate is broken into several states for FSMD. Each state of a FSMD lasts exactly one cycle time. A state of a FSMD can contain as many computations as the data dependencies allow. However a lot of computations in a state cause the datapath to become large (see section scheduling).

  In the example, the superstate `S_1` in figure 13 is scheduled to the states `S_1` to `S_3` in figures 14 and 15. The "power"-operation of the superstate cannot be executed in one cycle. Hardware usually performs this operation by running a loop of multiplications. So does this FSMD example.

4.2.3 Discussion: FSMD Behavioral without time or with clock

The constraint of using variables for behavior input and output is not nice. Behavior input and outputs as well as many other value-holders[1] can be expected to have a physical equivalence (wire) in the final hardware. Therefore the use of signals would be more accurate.

A more unpleasant circumstance is the fact, that in a description without clock, a VHDL Simulator determines the execution time equal to zero. Therefore the possibilities for debugging the code are limited to the old fashioned way of running the code step by step. When simulating hardware, displaying the data and the clock cycles over the time may be desired. That is not possible, because there is no time. This disadvantage can be removed by using FSMD Behavioral with clock. In this description a clock accurate simulation shows much better how the design will work. (Example: The *"Waveform Viewer"* provided by *"Synopsys"* can be employed now.) Debugging becomes easier. Additionally, the designer already sees, where to pay attention to possible timing problems.

Using the appropriate subset of VHDL, the model FSMD Behavioral with clock can be even directly synthesizable already (with bad results though, compared to "FSM Controlling Datapath").

A theoretical disadvantage of this model is the fact, that hardware details like clock normally belong to a

[1] *Value-holder* is used as a generic term for both VHDL-*signal* and VHDL-*variable*.

structural description instead of to a behavioral one, because it is an implementation detail. However, this step has to be performed anyway and besides the advantages for simulation, this makes the step to the description "FSM Controlling Datapath" closer.

Finally, such allusions to hardware implementations are just *suggestions* and *not definite*. It still has to be regarded as being a *behavioral* description. The used clock time may change or become a multiple of the real system clock.

It also is possible to use both behavioral models, first the one without clock and then refine it to the one with clock.

4.2.4 FSM Behavioral Controlling Datapath Structural

After the FSMD Behavioral description with clock is finished, the step to a structural datapath controlled by a FSM is easily described, but tedious to perform.

- A VHDL datapath file has to be created, which binds the library components[2] and connects the entity signals (data in- and outputs, control inputs and message outputs). See figure 16.
- All computations and data assignments are removed from the state machines, including computations within conditions. Instead, the appropriate control signals, which cause the data processing in the datapath, are set. See example in figure 17.
- The state machine is notified of the result of conditions by message signals, only few bits wide ("`CMP`" in figure 16).
- A separate process for setting the next state has to be introduced.
  The remaining process for the FSM is not clock-triggered any longer, but sensitive to all inputs of the FSM. (See sensitivity list in the process's head.) Thus this process can react to results of the datapath *before* the next clock edge arrives and "compensate" the delay of the datapath (see section 4.2.6 below).
- A datapath structural description has to be got. This is a deep refinement level. All wires and hardware blocks are specified. Therefore only signals have to be used and the use of variables is forbidden here. Components of the datapath are instantiated as shown in figure 16 by an example.

4.2.5 FSM Structural Controlling Datapath Structural

In most cases, the step from behavioral FSM to structural FSM does not need to be supervised by humans. Figure 18 shows the conversion into a structural description. Each bit of the output of the FSM is a boolean function of all bits.

The example shows the unoptimized coding. The size of the boolean functions should be minimized before synthesis.

4.2.6 Compensation of Datapath Delay

An FSMD does computation directly. For example, in figure 14, the state `S_2` contains `Mult_temp :=O * A`. In contrast to that, a control-FSM just gives commands.

The actual computations are executed by the datapath one clock cycle delayed. As a result, a FSMD could not simply be translated into "FSM Controlling Datapath". The solution is a modification in figure 12 which is shown in figure 17. As soon as the datapath gives its result, the FSM is employed again within the same clock cycle because of the feedback messages appearing in the sensitivity list. In figure 17 there is just one feedback signal: `CMP`.

Thus the datapath delay is "compensated" and computation results can be used one cycle after they were ordered. The behavior of "FSM Controlling Datapath" is back to the expected behavior of a state machine. It is equivalent to the behavior of the FSMD, from which it is derived and in accordance with the FSM architecture on the left-hand side of figure 8 in section 1.3.3.

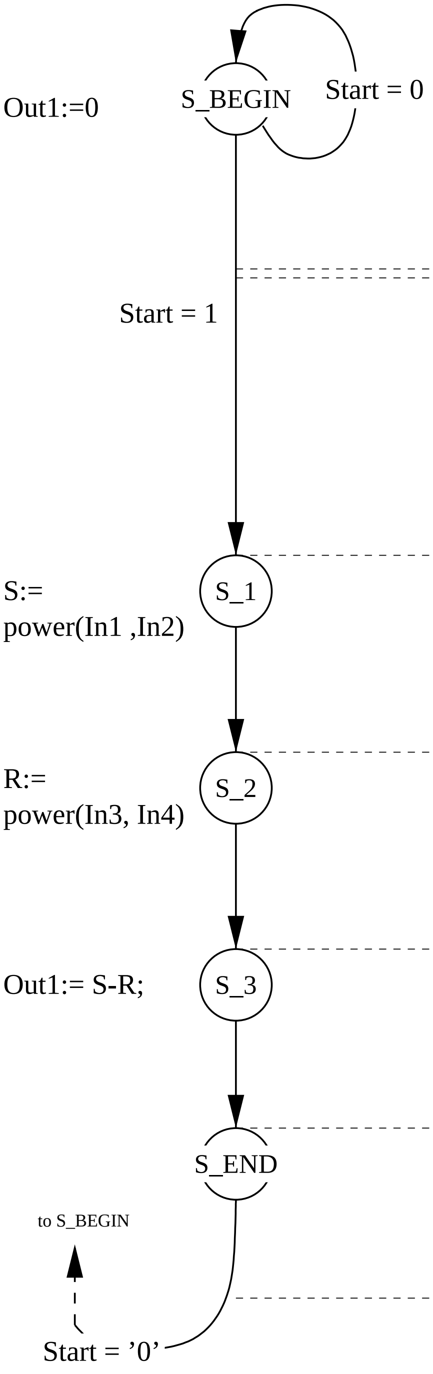
Note on "compensation":
When talking about "compensation" here, it does not mean a real removal of the delay. Delays and actions are just moved closer together. This causes a decrease of the allowed clock period time!

Warnings:

- Omitting the separation of next-state process and state-machine process means, the delay through the datapath will not be compensated. This is equivalent to introducing additional output latches as shown in figure 19. Doing this is possible, but discouraged. It means a change of common architecture agreements and the behavior is not equivalent to the FSMD-model.

[2] from user libraries and/or propriatary libraries

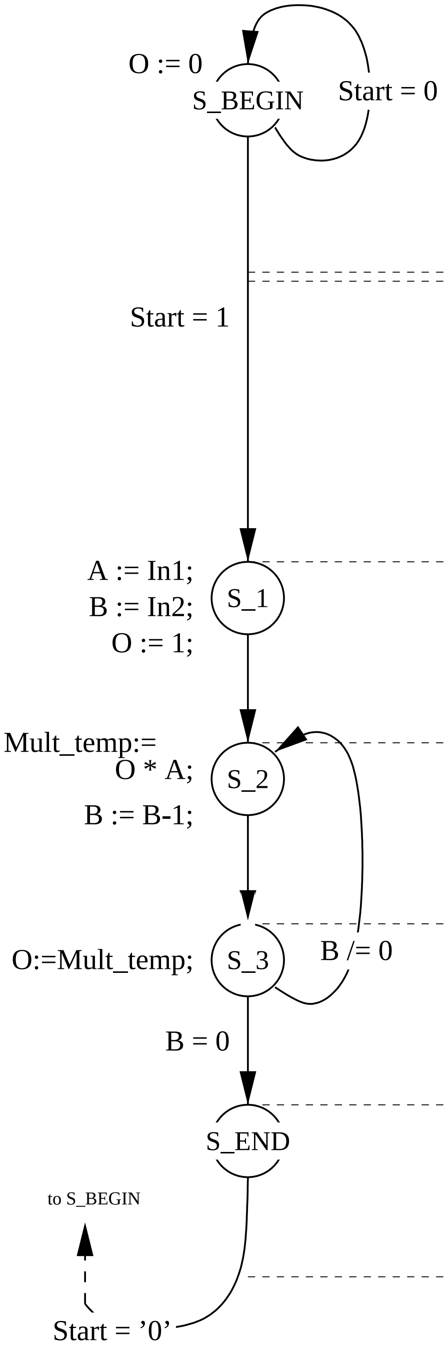
- Introducing a separate next-state process in the *FSMD*-model itself is *strictly* forbidden! Trying to compensate non-existent delays means removing necessary latches. Unlatched, indeterministic feedback loops may occur. Also, such a description can never be translated into "FSM Controlling Datapath" any more.



```
-- S_BEGIN == "procedure not running", managed by the calling process:
P1: Process;
begin
wait until (Reset = '1') or (Start = '1');
   if (Reset='1') then
     Out1 <= 0;
   elsif (Start = '1') then
     behavior1( In1, In2, In3, In4, Out1);
   end if;
end process;
-----------------------------------------------------------------------------
Procedure behavior1(  In1, In2, In3, In4: in integer;
                      signal Out1: out integer ) is
type State_Set is (S_1, S_2, S_3, S_END);
variable next_STATE: State_Set;
variable R, S: integer;

begin
next_STATE := S_1;
while (next_STATE /= S_END) and (Reset/='1') loop
   case STATE is
-----------------------------------------------------------------------------
    when S_1 =>   -- S:= In1 ** In2;-- (power)
                  S:= 1;
                  for i in In2 downto 1 loop
                     S:= S*In1;
                  end loop;
                  next_STATE:= S_2;
-----------------------------------------------------------------------------
    when S_2 =>   -- R:= In3 ** In4;  -- (power)
                  R:= 1;
                  for i in In4 downto 1 loop
                     R:= R*In3;
                  end loop;
                  next_STATE:= S_3;
-----------------------------------------------------------------------------
    when S_3 =>   Out1:= S-R;


                  next_STATE:= S_END;
-----------------------------------------------------------------------------
    when S_END => -- nothing



-----------------------------------------------------------------------------
  end case;
end loop;
end behavior1;
-----------------------------------------------------------------------------
```

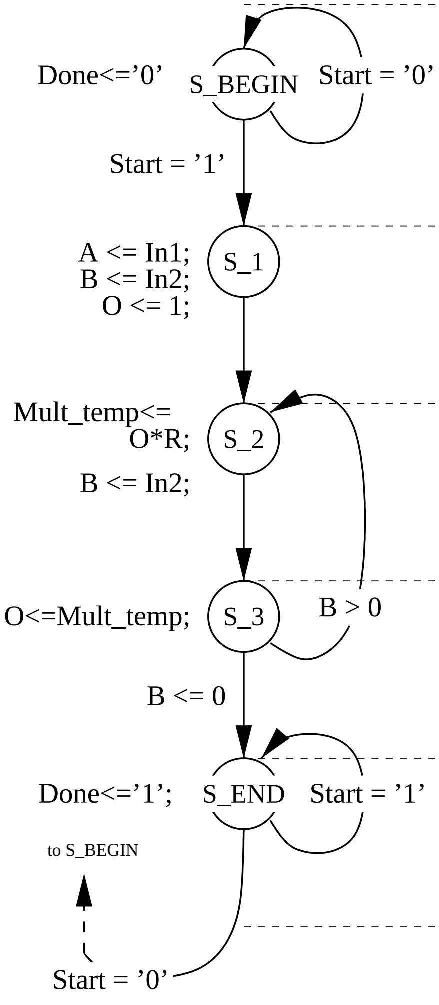
Example
for FSMD

Figure 13: Example: SFSMD Behavioral in VHDL



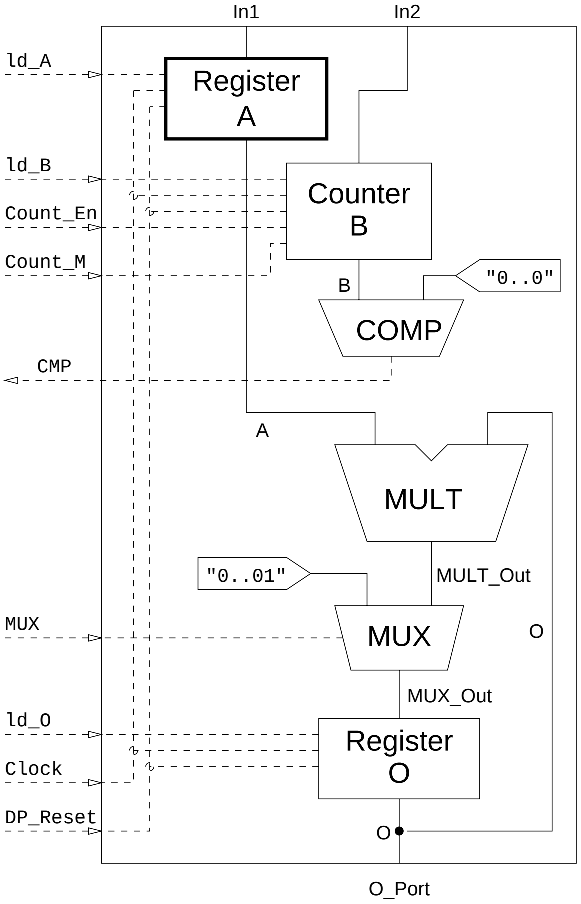
```
 -- S_BEGIN == "procedure not running", managed by the calling process:
 P1: Process;
 begin
 begin
 wait until (Reset = ’1’) or (Start = ’1’);
    if (Reset=’1’) then
      O_Port <= 0;
    elsif (Start = ’1’) then
      behavior1( In1, In2,  O);
    end if;
 end process;

 Procedure behavior1(  In1, In2: in natural;
                       O: inout natural ) is
 type State_Set is (S_1, S_2, S_3, S_END);
 variable next_STATE: State_Set;
 variable A, B, Mult_temp: natural;

 begin
 next_STATE := S_1;
 while (next_STATE /= S_END) and (Reset/=’1’) loop
    case next_STATE is

     when S_1 =>      A := In1;
                      B := In2;
                      O := 1;

                      next_STATE:= S_2;

     when S_2 =>      Mult_temp := O * A;
                      B:= B-1;

                      next_STATE:= S_3;

     when S_3 =>      O <= Mult_temp;

                      if (B > 0)
                        then next_STATE:= S_2;
                        else next_STATE:= S_END;
                      end if;

     when S_END =>     -- nothing

   end case;
end loop;
end behavior1;
```

Figure 14: Example: FSMD Behavioral Without Time in VHDL



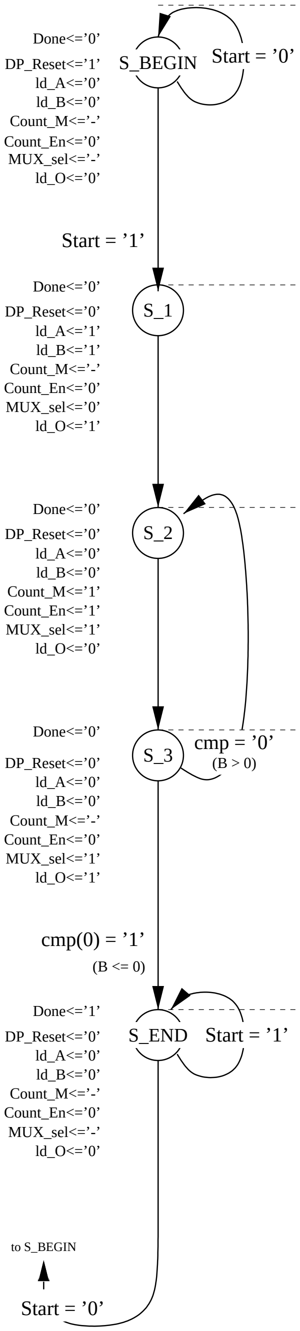
```
behavior1: Process(Clock)
begin
 if (Clock'event and Clock='1') then
   if (Reset = '1') then
     O <= (others=>'0');
     next_STATE <= S_BEGIN;
   else
     case next_STATE is
        when S_BEGIN =>  Done <= '0';

                         if Start = '1'
                           then next_STATE<= S_1;
                           else next_STATE<= S_BEGIN;
                         end if;
        when S_1 =>      A <= In1;
                         B <= In2;
                         O <= conv_std_logic_vector(1, 32);

                         next_STATE<= S_2;
        when S_2 =>      Mult_temp <= O(15 downto 0) * A;
                         B <= B - conv_std_logic_vector(1, 1

                         next_STATE<= S_3;
        when S_3 =>      O <= Mult_temp;

                         if (B > conv_std_logic_vector(0, 16
                           then next_STATE<= S_2;
                           else next_STATE<= S_END;
                         end if;
        when S_END =>    Done <= '1';

                         if Start = '0'
                           then next_STATE<= S_BEGIN;
                           else next_STATE<= S_END;
                         end if;
     end case;
   end if;
 end if;
end process;
```

Figure 15: Example: FSMD Behavioral With Clock in VHDL



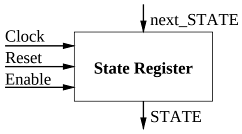
```
Entity DP is
    port (Clock:       in std_logic;
          DP_Reset:    in std_logic;
          ld_A, ld_B, Count_En, Count_M: in std_logic;
          CMP:        out std_logic_vector(1 downto 0);
          MUX_sel:     in std_logic;
          ld_O:        in std_logic;
          In1, In2:    in std_logic_vector(15 downto 0);
          O_Port:     out std_logic_vector(31 downto 0));
end DP;

Architecture DP_schematic of DP is
    -- connection signals
    signal A, B: std_logic_vector(15 downto 0);
    signal O: std_logic_vector(31 downto 0);
    signal MULT_Out, MUX_Out: std_logic_vector(31 downto 0);

    component Reg_16bit
       Port ( Clock:     in std_logic;
              Reset:     in std_logic;
              Load:      in std_logic;
              Data_In:   in std_logic_vector(15 downto 0);
              Data_Out: out std_logic_vector(15 downto 0));
    end component;
         ⋮

begin

Register_A: Reg_16bit
     Port Map ( Clock=>Clock, Reset=>DP_Reset, Load=>ld_A,
                Data_In=>In1, Data_Out=>A);
     ⋮

O <= O_Port;
end DP schematic:
```

Figure 16: Example: Datapath of "FSM Behavioral Controlling Datapath Structural"



```
Transition: Process
begin
 wait until(Clock’event and Clock = ’1’);
 STATE <= next_STATE;
end Process;

behavior1: Process  (STATE, CMP, Start)
begin
 if (Reset = ’1’) then
   ...             <
   next_STATE <= S_BEGIN;
 else
   case STATE is
      when S_BEGIN =>    Done<=’0’;
                         DP_Reset<=’1’;
                         ld_A<=’0’;
                         ld_B<=’0’;
                         Count_M<=’-’;
                         Count_En<=’0’
                         MUX_sel<=’0’;
                         ld_O<=’0’;

                         if Start = ’1’
                           then next_STATE<= S_1;
                           else next_STATE<= S_BEGIN;
                         end if;
      when S_1 =>        Done<=’0’;
                         DP_Reset<=’0’;
                         ld_A<=’1’;
                         ld_B<=’1’;
                         Count_M<=’-;’
                         Count_En<=’0’;
                         MUX_sel<=’0’;
                         ld_O<=’1’;

                         next_STATE<= S_2;
      when S_2 =>        Done<=’0’;
                         DP_Reset<=’0’;
                         ld_A<=’0’;
                         ld_B<=’0’;
                         Count_M<=’1’;
                         Count_En<=’1’;
                         MUX_sel<=’1’;
                         ld_O<=’0’;

                         next_STATE<= S_3;
      when S_3 =>        Done<=’0’;
                         DP_Reset<=’0’;
                         ld_A<=’0’;
                         ld_B<=’0’;
                         Count_M<=’-’;
                         Count_En<=’0’;
                         MUX_sel<=’1’;
                         ld_O<=’1’;

                         if (CMP(0) = ’1’
                           then next_STATE<= S_2;
                           else next_STATE<= S_END;
                         end if;
      when S_END =>      Done<=’1’;
                         DP_Reset<=’0’;
                         ld_A<=’0’;
                         ld_B<=’0’;
                         Count_M<=’-’;
                         Count_En<=’0’;
                         MUX_sel<=’-’;
                         ld_O<=’0’
                                  ;
                         if Start = ’0’
                           then next_STATE<= S_BEGIN;
                           else next_STATE<= S_END;
                         end if;
   end case;
 end if;
end process;
```

Figure 17: Example: FSM of ”FSM Behavioral Controlling Datapath Structural”



```
State_Register: Process(Clock, Enable)
begin
  if (Clock’event and Clock = ’1’) then
    if    (Reset = ’1’)  then STATE <= 000;
    elsif (Enable = ’1’) then STATE <= next_STATE;
    end if;
  end if;
end Process;
```

	DP-Control and Status Output Logic							
Input	DP-Control Output							Status Output
STATE	DP_Reset	ld_A	ld_B	Count_M	Count_En	MUX_sel	ld_O	Done
000	1	0	0	-	0	-	0	0
001	0	1	1	-	0	0	1	0
010	0	0	0	1	1	1	0	0
011	0	0	0	-	0	1	1	0
111	0	0	0	-	0	-	0	1
rest	don’t care							don’t care

```
-- Datapath-Control and Status Output Logic
-- Datapath Control Output
DP_Reset  <=   ( not STATE(2) and not STATE(1) and not STATE(0) );
ld_A      <=   ( not STATE(2) and not STATE(1) and     STATE(0) );
ld_B      <=   ( not STATE(2) and not STATE(1) and     STATE(0) );
Count_M   <=   ( not STATE(2) and     STATE(1) and not STATE(0) );
Count_En  <=   ( not STATE(2) and     STATE(1) and not STATE(0) );
MUX_sel   <=   ( not STATE(2) and     STATE(1) and not STATE(0) )
            OR ( not STATE(2) and     STATE(1) and     STATE(0) );
ld_O      <=   ( not STATE(2) and not STATE(1) and     STATE(0) )
            OR ( not STATE(2) and     STATE(1) and     STATE(0) );


-- Controller Status Output
Done      <=   (     STATE(2) and     STATE(1) and     STATE(0) );
```

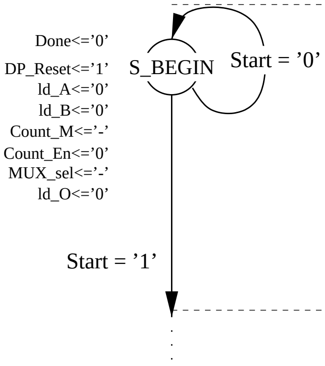
			Next-State Logic
Input			Output
STATE	CMP	Start	next_STATE
000	--	0	000
000	--	1	001
001	--	-	010
010	--	-	011
011	-1	-	010
011	-0	-	111
111	--	-	000
rest	--	-	don’t care

```
-- Next-State Logic
next_STATE(2) <=    ( not STATE(2) and     STATE(1) and     STATE(0) and not CMP(0) );
next_STATE(1) <=    ( not STATE(2) and not STATE(1) and     STATE(0) )
                 OR ( not STATE(2) and     STATE(1) and not STATE(0) )
                 OR ( not STATE(2) and     STATE(1) and     STATE(0) and     CMP(0) )
                 OR ( not STATE(2) and     STATE(1) and     STATE(0) and not CMP(0) );
next_STATE(0) <=    ( not STATE(2) and not STATE(1) and not STATE(0) and START )
                 OR ( not STATE(2) and     STATE(1) and not STATE(0) )
                 OR ( not STATE(2) and     STATE(1) and     STATE(0) and not CMP(0) );
```

Figure 18: Example: FSM of ”FSM Structural Controlling Datapath Structural”

a) Changes in FSM Behavioral Description when Using just One Process (Outputs Latched)

```
behavior1: Process (Clock)
begin
 if (Clock’event and Clock = ’1’) then
   if (Reset = ’1’) then
     ...
     next_STATE <= S_BEGIN;
   else
     case STATE is
```



```
       when S_BEGIN =>    Done<=’0’;
                          DP_Reset<=’1’;
                          ld_A<=’0’;
                          ld_B<=’0’;
                          Count_M<=’-’;
                          Count_En<=’0’
                          MUX_sel<=’0’;
                          ld_O<=’0’;

                          if Start = ’1’
                            then next_STATE<= S_1;
                            else next_STATE<= S_BEGIN;
                          end if;
                           .
                           .
                           .
```

b) Equvalent Changes in FSM Structural Description / Changes of the former Architecture (Outputs Latched)

```
Next_State_Register: Process(Clock, Enable)
begin
  if (Clock’event and Clock=’1’) then
    if    (Reset=’1’) then next_STATE <= "000";
    elsif (Enable=’1’) then next_STATE <= next_STATE_logic;
    end if;
  end if;
end Process;

Output_Registers: Process(Clock, Enable)
begin
  if (Clock’event and Clock=’1’) then
    if    (Reset=’1’) then
        DP_Reset <= ’1’;
        ld_A<=’0’;
        ld_B<=’0’;
        Count_M<=’0’;
        Count_En<=’0’;
        MUX_sel<=’0’;
        ld_O<=’0’;
        Done <= ’0’;
    else
        DP_Reset <= DP_Reset_Logic;
        ld_A <= ld_A_Logic;
        ld_B <= ld_B_logic;
        Count_M <= Count_M_Logic;
        Count_En <= Count_En_Logic;
        MUX_sel <= MUX_sel_Logic;
        ld_O <= ld_O_Logic;
        Done <= Done_Logic;
    end if;
  end if;
end Process;

-- Datapath-Control and Status Output Logic
DP_Reset_Logic <= ( not STATE(2) and not STATE(1) and not STATE(0) );
ld_A_Logic <= ...  ...
   ⋮

-- Next State Logic
next_STATE_Logic(2) <= (not STATE(2) and STATE(1) and STATE(0) and not CMP(0) );
next_STATE_Logic(1) <= ...  ...
   ⋮
```

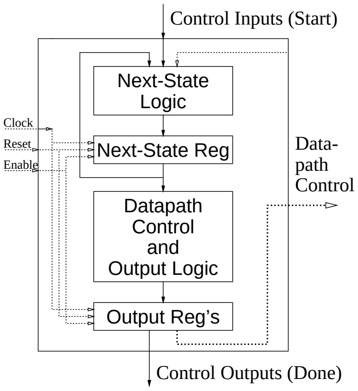
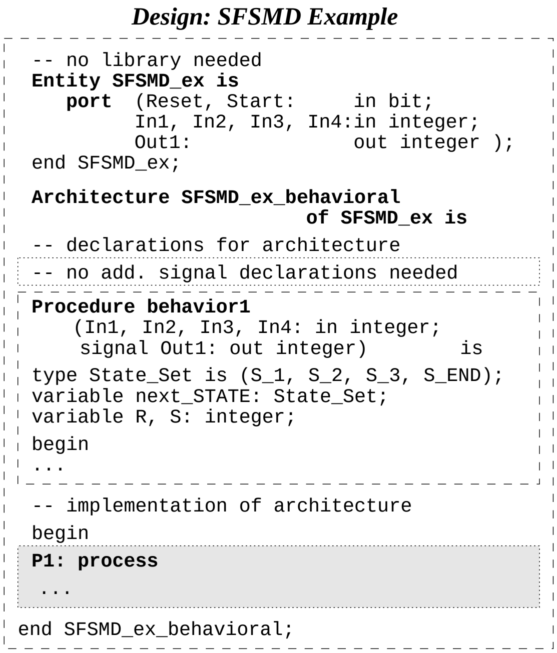


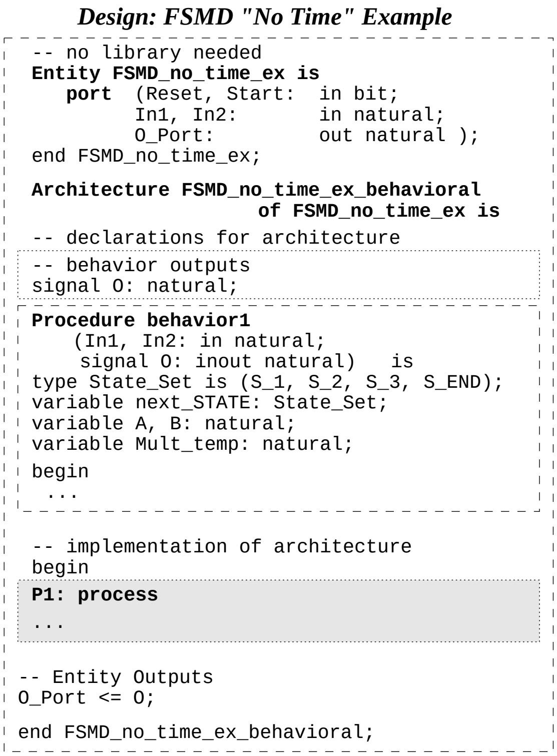
Figure 19: Warning: Omitting separation of next-state-process causes additional output latches



Design: SFSMD Example

```
-- no library needed
Entity SFSMD_ex is
   port  (Reset, Start:     in bit;
         In1, In2, In3, In4:in integer;
         Out1:              out integer );
end SFSMD_ex;

Architecture SFSMD_ex_behavioral
                        of SFSMD_ex is
-- declarations for architecture
-- no add. signal declarations needed
Procedure behavior1
    (In1, In2, In3, In4: in integer;
     signal Out1: out integer)       is
type State_Set is (S_1, S_2, S_3, S_END);
variable next_STATE: State_Set;
variable R, S: integer;
begin
...

-- implementation of architecture
begin
P1: process
 ...

end SFSMD_ex_behavioral;
```

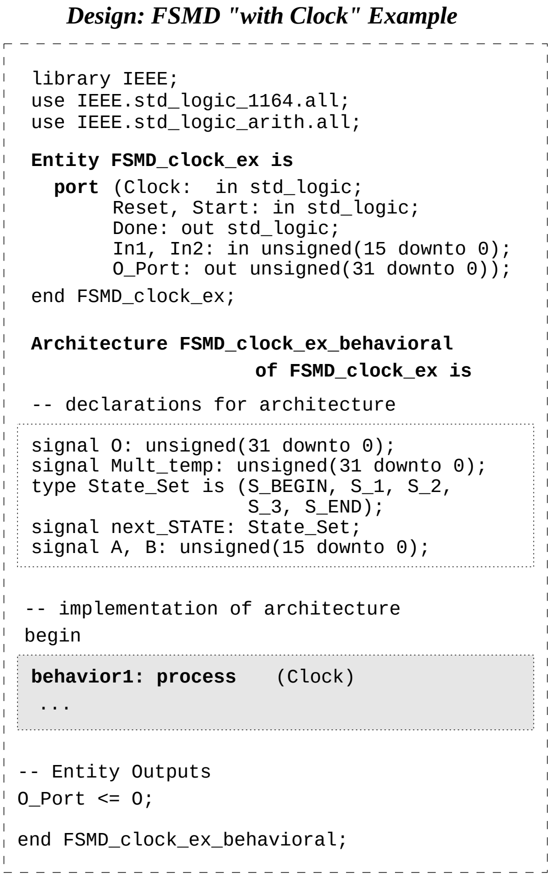
Figure 20: Embedding the example SFSMD (fig. 13) in VHDL



Design: FSMD "No Time" Example

```
-- no library needed
Entity FSMD_no_time_ex is
   port  (Reset, Start:  in bit;
         In1, In2:       in natural;
         O_Port:         out natural );
end FSMD_no_time_ex;

Architecture FSMD_no_time_ex_behavioral
                    of FSMD_no_time_ex is
-- declarations for architecture
-- behavior outputs
signal O: natural;
Procedure behavior1
    (In1, In2: in natural;
     signal O: inout natural)   is
type State_Set is (S_1, S_2, S_3, S_END);
variable next_STATE: State_Set;
variable A, B: natural;
variable Mult_temp: natural;
begin
  ...

-- implementation of architecture
begin
P1: process
...

-- Entity Outputs
O_Port <= O;

end FSMD_no_time_ex_behavioral;
```

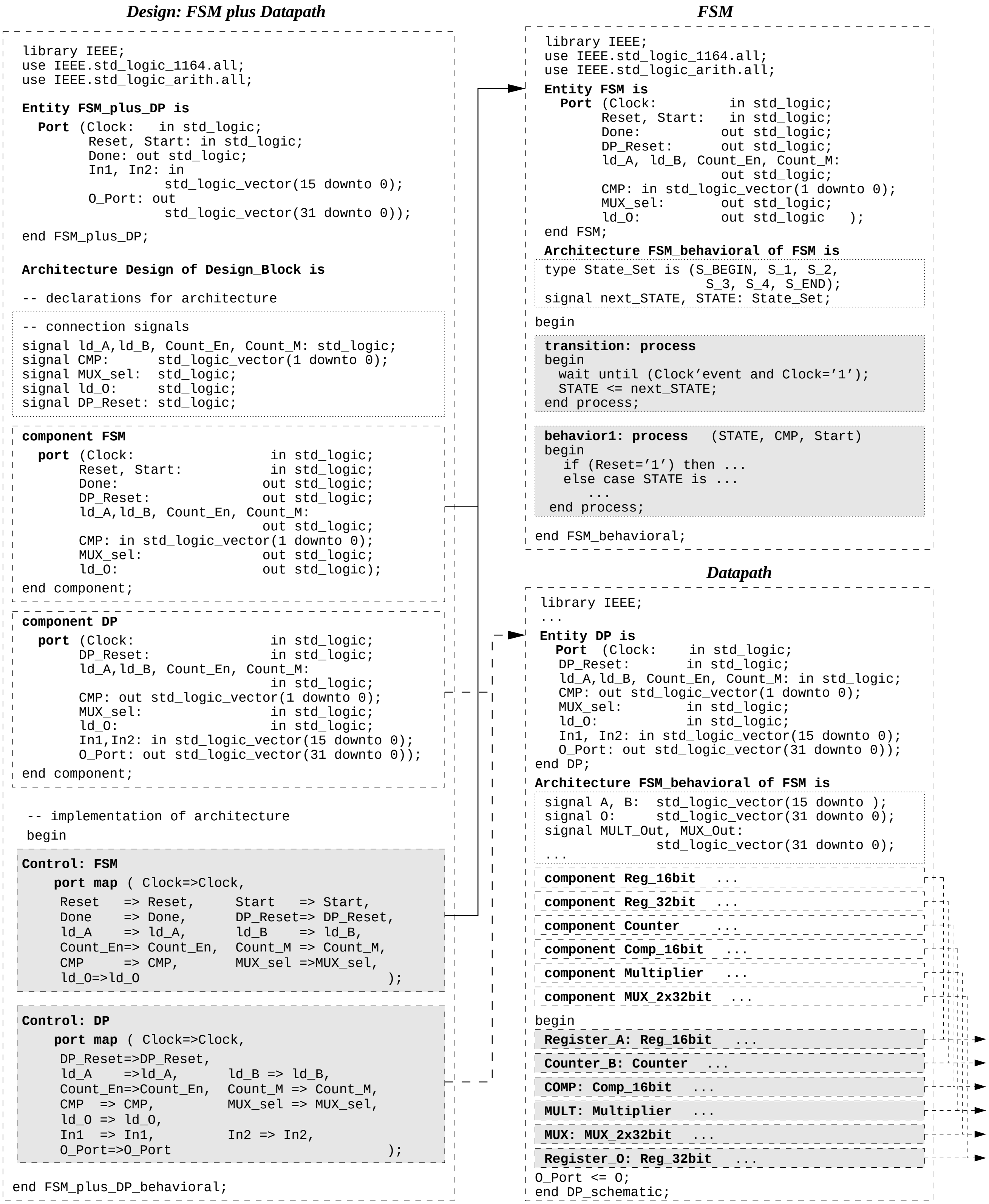
Figure 21: Embedding the example FSMD Without Time (fig. 14) in VHDL



Design: FSMD "with Clock" Example

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

Entity FSMD_clock_ex is
  port (Clock:  in std_logic;
        Reset, Start: in std_logic;
        Done: out std_logic;
        In1, In2: in unsigned(15 downto 0);
        O_Port: out unsigned(31 downto 0));
end FSMD_clock_ex;

Architecture FSMD_clock_ex_behavioral
                   of FSMD_clock_ex is
-- declarations for architecture

signal O: unsigned(31 downto 0);
signal Mult_temp: unsigned(31 downto 0);
type State_Set is (S_BEGIN, S_1, S_2,
                   S_3, S_END);
signal next_STATE: State_Set;
signal A, B: unsigned(15 downto 0);

-- implementation of architecture
begin

behavior1: process   (Clock)
 ...

-- Entity Outputs
O_Port <= O;

end FSMD_clock_ex_behavioral;
```

Figure 22: Embedding the example FSMD With Clock (fig. 15) in VHDL



Design: FSM plus Datapath

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

Entity FSM_plus_DP is
  Port (Clock:   in std_logic;
        Reset, Start: in std_logic;
        Done: out std_logic;
        In1, In2: in
                  std_logic_vector(15 downto 0);
        O_Port: out
                  std_logic_vector(31 downto 0));
end FSM_plus_DP;

Architecture Design of Design_Block is

-- declarations for architecture

-- connection signals
signal ld_A,ld_B, Count_En, Count_M: std_logic;
signal CMP:      std_logic_vector(1 downto 0);
signal MUX_sel:  std_logic;
signal ld_O:     std_logic;
signal DP_Reset: std_logic;

component FSM
  port (Clock:                 in std_logic;
       Reset, Start:           in std_logic;
       Done:                  out std_logic;
       DP_Reset:              out std_logic;
       ld_A,ld_B, Count_En, Count_M:
                              out std_logic;
       CMP: in std_logic_vector(1 downto 0);
       MUX_sel:               out std_logic;
       ld_O:                  out std_logic);
end component;

component DP
  port (Clock:                 in std_logic;
       DP_Reset:               in std_logic;
       ld_A,ld_B, Count_En, Count_M:
                               in std_logic;
       CMP: out std_logic_vector(1 downto 0);
       MUX_sel:                in std_logic;
       ld_O:                   in std_logic;
       In1,In2: in std_logic_vector(15 downto 0);
       O_Port: out std_logic_vector(31 downto 0));
end component;

 -- implementation of architecture
 begin

Control: FSM
    port map ( Clock=>Clock,
     Reset    => Reset,      Start    => Start,
     Done     => Done,       DP_Reset=> DP_Reset,
     ld_A     => ld_A,       ld_B     => ld_B,
     Count_En=> Count_En,    Count_M => Count_M,
     CMP      => CMP,        MUX_sel =>MUX_sel,
     ld_O=>ld_O                                 );

Control: DP
    port map ( Clock=>Clock,
     DP_Reset=>DP_Reset,
     ld_A     =>ld_A,       ld_B => ld_B,
     Count_En=>Count_En,    Count_M => Count_M,
     CMP  => CMP,           MUX_sel => MUX_sel,
     ld_O => ld_O,
     In1  => In1,           In2 => In2,
     O_Port=>O_Port                             );

end FSM_plus_DP_behavioral;
```

FSM

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
Entity FSM is
  Port (Clock:         in std_logic;
       Reset, Start:   in std_logic;
       Done:          out std_logic;
       DP_Reset:      out std_logic;
       ld_A, ld_B, Count_En, Count_M:
                      out std_logic;
       CMP: in std_logic_vector(1 downto 0);
       MUX_sel:       out std_logic;
       ld_O:          out std_logic   );
end FSM;
Architecture FSM_behavioral of FSM is
type State_Set is (S_BEGIN, S_1, S_2,
                    S_3, S_4, S_END);
signal next_STATE, STATE: State_Set;
begin
transition: process
begin
  wait until (Clock'event and Clock='1');
  STATE <= next_STATE;
end process;

behavior1: process   (STATE, CMP, Start)
begin
  if (Reset='1') then ...
  else case STATE is ...
     ...
 end process;

end FSM_behavioral;
```

Datapath

```
library IEEE;
...
Entity DP is
  Port  (Clock:    in std_logic;
   DP_Reset:       in std_logic;
   ld_A,ld_B, Count_En, Count_M: in std_logic;
   CMP: out std_logic_vector(1 downto 0);
   MUX_sel:        in std_logic;
   ld_O:           in std_logic;
   In1, In2: in std_logic_vector(15 downto 0);
   O_Port: out std_logic_vector(31 downto 0));
end DP;
Architecture FSM_behavioral of FSM is
 signal A, B:  std_logic_vector(15 downto );
 signal O:     std_logic_vector(31 downto 0);
 signal MULT_Out, MUX_Out:
               std_logic_vector(31 downto 0);
 ...
 component Reg_16bit   ...
 component Reg_32bit   ...
 component Counter    ...
 component Comp_16bit   ...
 component Multiplier   ...
 component MUX_2x32bit  ...
begin
 Register_A: Reg_16bit   ...
 Counter_B: Counter  ...
 COMP: Comp_16bit  ...
 MULT: Multiplier  ...
 MUX: MUX_2x32bit  ...
 Register_O: Reg_32bit   ...
O_Port <= O;
end DP_schematic;
```

Figure 23: Embedding the example "FSM Controlling Datapath" (fig. 16 and 17) in VHDL

5 Communication of State Machines

State machines may have to exchange data. There can be different reasons for having state machines which need to communicate:

- A large super state machine can be divided into smaller state machines in order to get smaller problems.
- Separate state machines can be designed, which solve common problems. They may be required for use in connection with other state machines like "plug-ins".
- A state machine may have to be divided, when it contains problems with different demands of the implementation platform. Each piece then can run on the platform which is best.

In all these cases, data exchange between the several state machines is necessary and an appropriate protocol needs to come along with it. In the following subsections, the simple method of handshaking is discussed, which is sufficient for many problems. *Single handshake* can be used, if the communicating state machines have the same cycle time. However *double handshake* is recommended, since it works with all odd cycle times and even if there is no assigned cycle time (SFSMD and FSMD Without Time).

A data sending machine may produce bursts of data with delays between the bursts. Also the receiving machine may consume the data irregularly. In these cases a buffer comes in handy, which can be a queue or a memory for instance. The design of state machines, exchanging data via a queue or memory, is shown in the later sections.

5.1 Single Handshake

A prerequisite for single handshake is that each state machine is either a FSMD, which uses clock, or a FSM, both working with the same clock.

Figure 24 shows the usage of signals for single handshake between two state machines. *Set-output* and *read* actions are adapted to FSMD or FSM according to the table below the figure. If the sender (A) is a FSMD, the Output is set in state `S_c` directly. If it is a FSM, the associated datapath sets the output one cycle delayed, in state `S_d`. The signal `A_Ready_B` is set to `'1'`, when the state machine has set the output or just sets it. This means, one cycle after `A_Ready_B` was set, the data can be read by the receiver (B). The receiver gives the acknowledge signal `B_Ackn_A` when it reads the data.

Note, that the receiver may reach state `S_w` earlier than data is send. Then it reads invalid data. But, the signal `A_Ready` is `'0'` as long as the data is invalid. This causes the receiver to read the data again and again until it finally reads valid data.

If the receiver is an FSM, reading is delayed by the datapath, too. However, reading the data in state `S_x` is early enough, since both state machines work synchronously with the same clock.

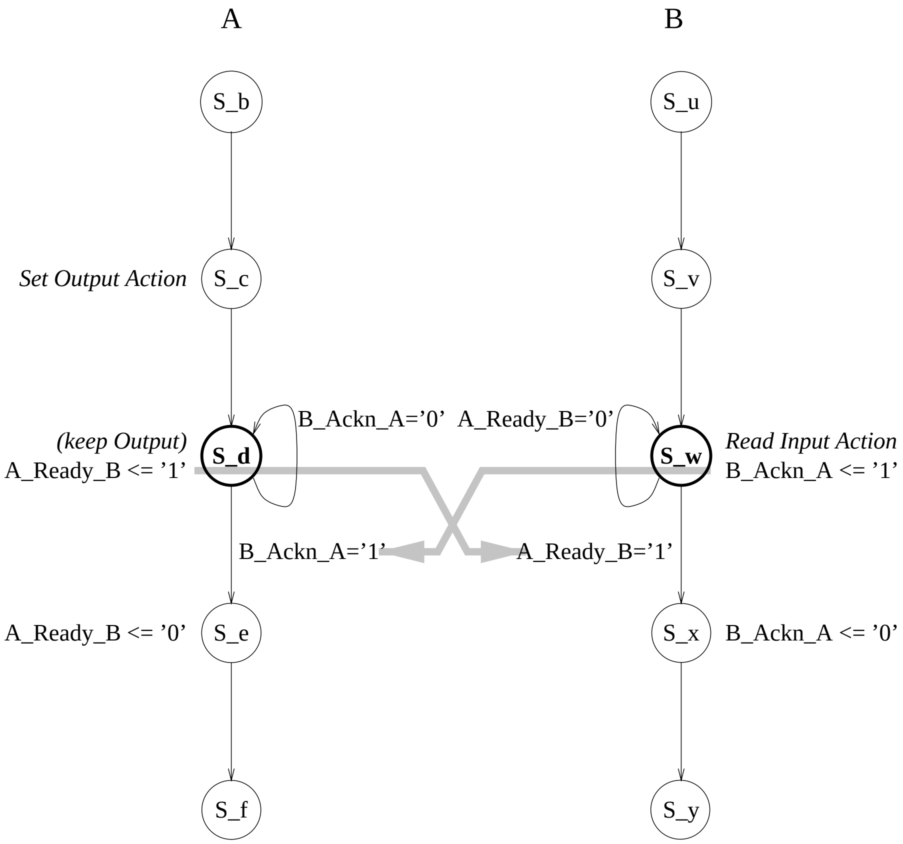
Figures 25 to 28 show the order of events and switching of states for all possibilities: Figures for FSMD-FSMD, FSM-FSM, FSM-FSMD and FSMD-FSM each contain diagrams for the following three cases: A is earlier than B, A and B reach the communication states at the same time, B is earlier than A (where A is sender and B is receiver).
Black arrows show how ready and acknowledge signals cause state transitions. A FSMD model cannot recognize the setting of a signal within the same cycle but only in the next cycle. This causes the large "scissors" in figure 25.
In the model "FSM Controlling Datapath", a side effect of compensating the datapath-delay is the early reaction to the input signals. The results are the smaller, narrow "scissors" in figure 26.
Figures 27 and 28 show the connection of FSM with FSMD, resulting in a mixture of the two types of "scissors".

In the FSMs, thin gray arrows indicate the wait period for setting and reading data by the datapath. The compensation of the datapath delay, stated in section 4.2.6, does not apply for synchronous data-transfer via the ports. The compensation just compensates a delayed reaction of the FSM *within* the same design.

Thick gray arrows point out, how many cycles after the actual setting of the data, it is read. This number must be greater than or equal to one, meaning the arrow must point downwards. A horizontal arrow is not allowed. If the thick arrows point downwards through several states, the output data has to be kept in these states.

The timing diagrams show the functionality of the construct in figure 24.

The appendix contains figures with statemachines transmitting data via a queue by single handshake. This is not discussed further, because the focus is set on the more robust and adaptive double handshake. For explanation about a queue, refer to section 5.3.2.



	FSMD (with clock)	***FSM (of FSM+Datapath)***
	Set / Read Ports	*Load Registers*
Set Output Action	Data_AB <= ...	Ld_Data_AB <= '1'
Read Output Action	B_Data_A <= Data_AB	Ld_B_Data_A <= '1'

Figure 24: Single Handshake

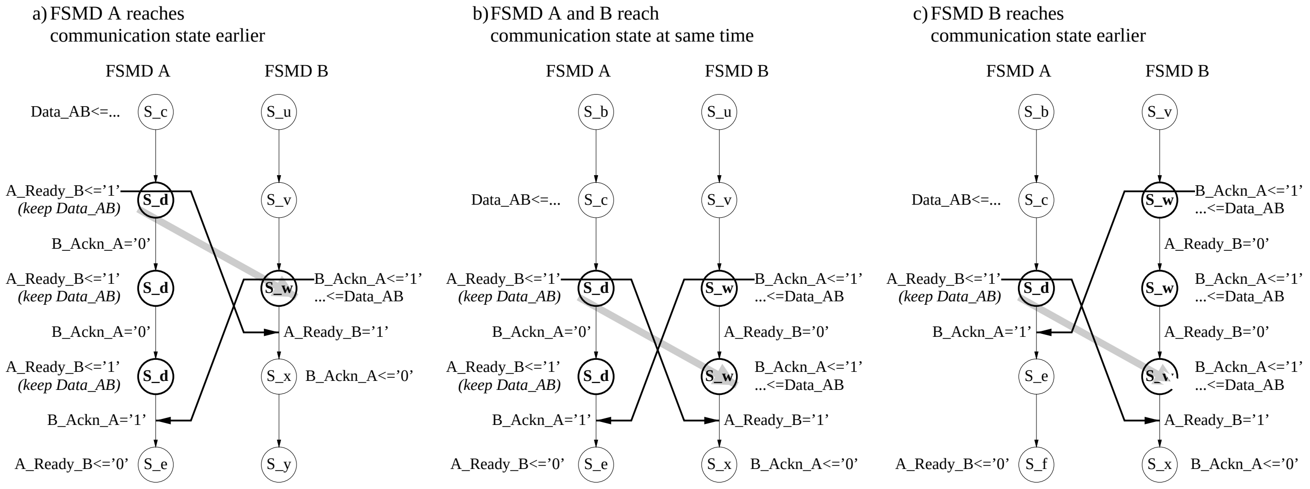


Figure 25: Cases, when using Single Handshake with two FSMD

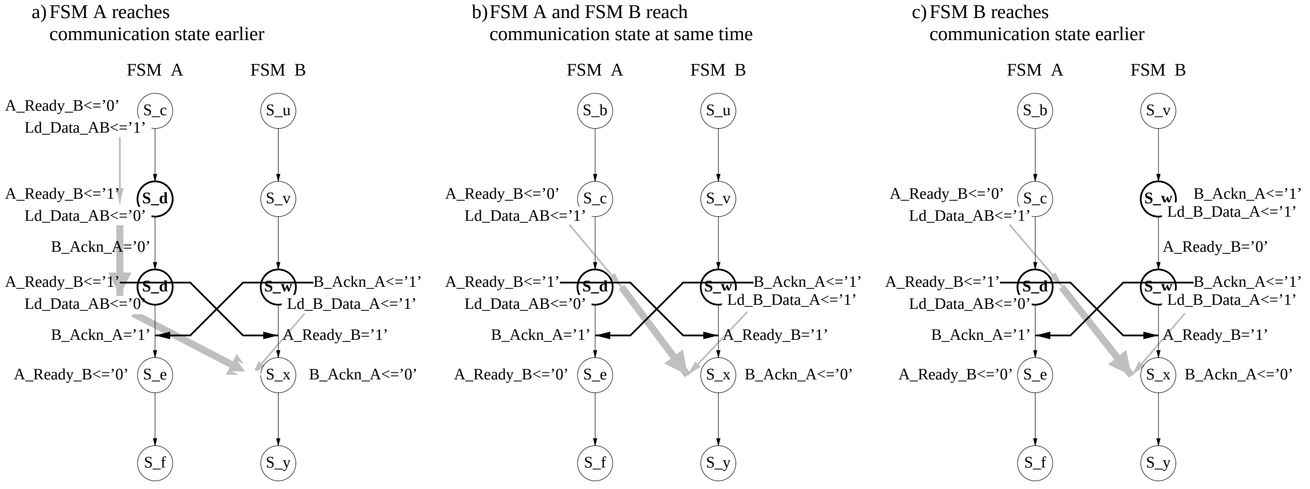


Figure 26: Cases, when using Single Handshake with two FSM

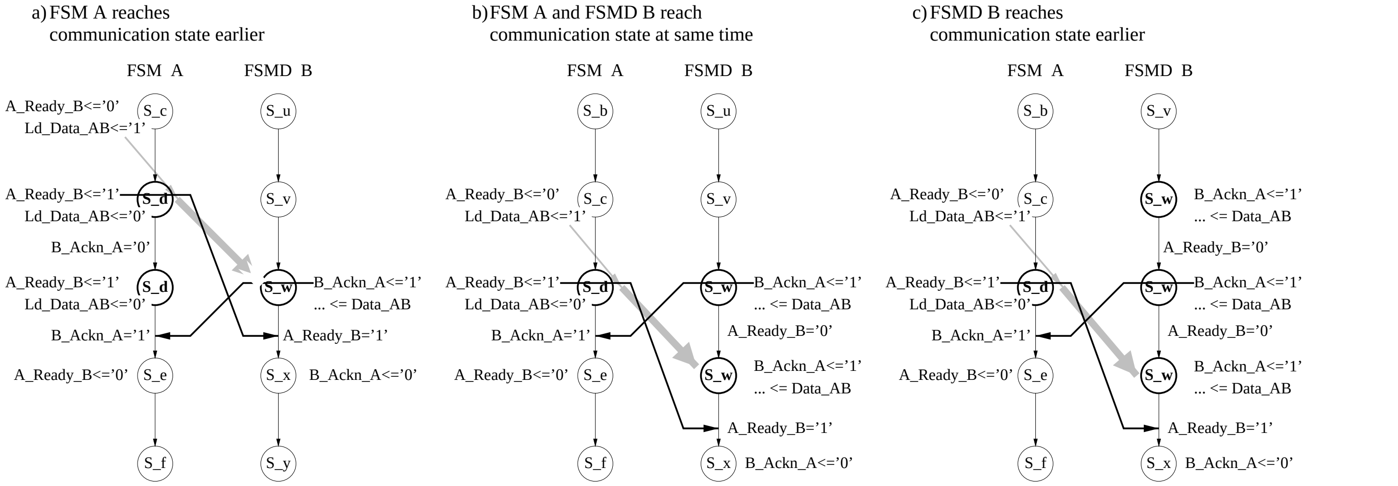


Figure 27: Cases, when using Single Handshake with a FSM transmitting to a FSMD

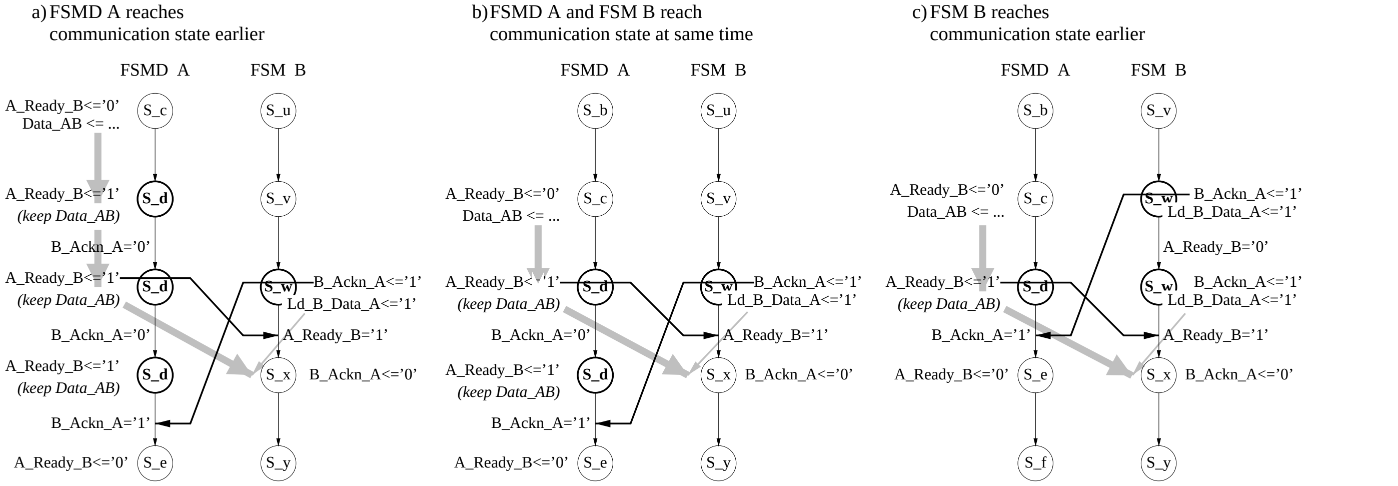


Figure 28: Cases, when using Single Handshake with a FSMD transmitting to a FSM

5.2 Double Handshake

Double handshaking makes the data exchange much more robust and works with all kinds of cycle times.

See figure 30. First, similar to the simple handshake, the sender puts data on its output port and sets the ready signal (A_Ready_B='1'). The receiver reads the data and acknowledges it by B_Ackn_A='1'. Then a second handshake follows: The sender sets A_Ready_B to '0' and waits for a B_Ackn_A='0' from the receiver. This assures that none of the state machines can run ahead of the other one:
If the receiver would not wait in state S_w for A_Ready_B to become '0' again, it would not be able to know when the sender has noticed that the data has been received. There is the danger of removing the received confirmation signal B_Ackn_A too early.
If the sender would not wait in state S_d for B_Ackn_A to be '0' again, it may start another data exchange in some other state and read the old B_Ackn_A='1'.

Figure 30 also assures, that the signal A_Ready_B='1' *at earliest* is set *one cycle after* the output signal has been set. Both signals cannot be set at the same time, because in an asynchronous communication, it is unknown which one would be recognized first by the receiver.
In the same way, the output signal must remain valid until the next receiver-cycle after the read operation. In figure 30, this is *not* done by moving B_Ackn_A <= '1' to a later cycle. Instead the output of A is held until B_Ackn_A = '0'. Doing so saves additional states.

The reaction delay of a datapath makes it necessary to move the *”set output operation”* within state machine A from state S_b to state S_a. State S_a can be used as a *”generalized set output state”*, which works for all models.
The delay of the datapath has to be considered in the receiving state machine, too. Here, a *”generalized”* state is impossible without introducing another state (figure 29). If no extra state is introduced, the output action has to be moved, depending on the used model (see figure 30):
SFSMD, FSMD: use state S_w: B_Data_A<=Data_AB;
FSM: use state S_v: ld_B_Data_A<='1'.

Figures 31 and 32 demonstrate the timing problems which may occur, if output action and input action are not scheduled at the right state.

Figure 32b also shows that a FSMD determines its next state when entering a state. An FSM, however, updates its ”next state”-value continuously during the the current cycle. Thus, communication between FSMs uses less clock cycles. Double handshake does not guarantee a specific number of cycles. It just assures a safe exchange of data.

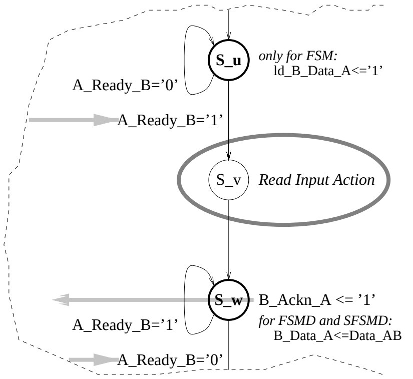
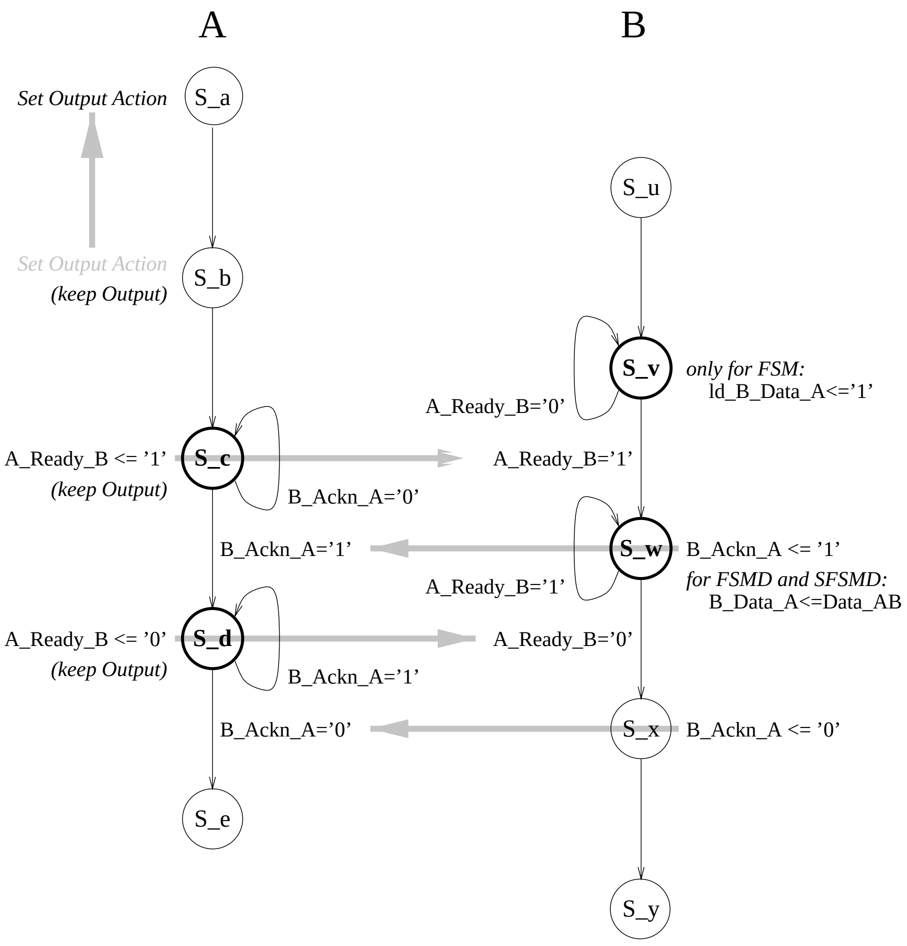


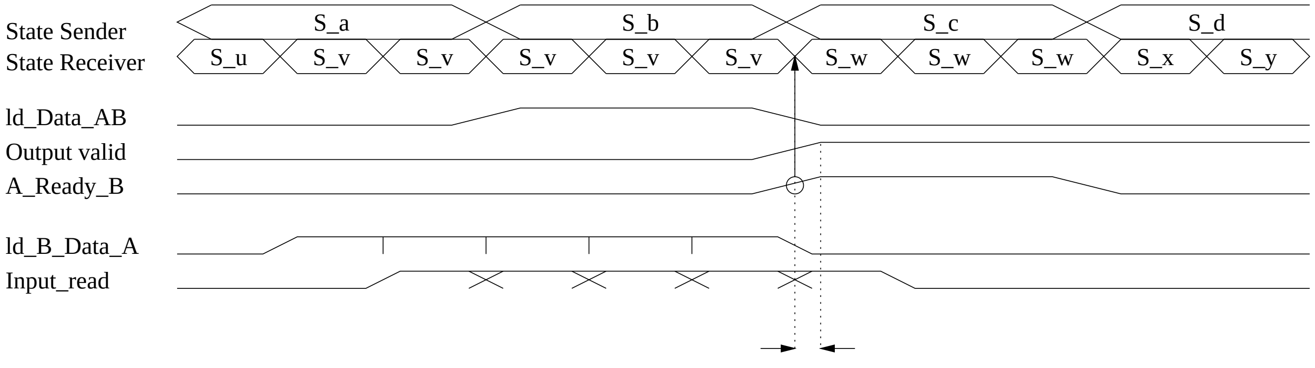
Figure 29: Insertion of a ”General State” in the receiving state machine of figure 30



	SFSMD / FSMD	***FSM (of FSM+D)***
	Set / Read Ports	*Load Registers*
Set Output Action	Data_AB <= ...	Ld_Data_AB <= ’1’
Read Output Action	B_Data_A <= Data_AB	Ld_B_Data_A <= ’1’

Figure 30: Double Handshake

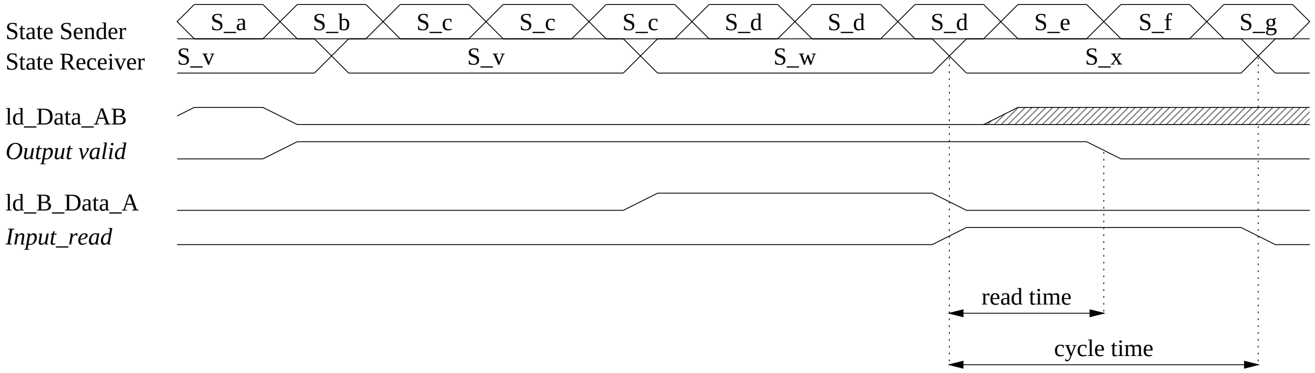
a) For the state machines being FSMs, assume the write command is given in state S_b instead of state S_a.
The following problem may occur, if the reading machine runs much faster than the writing machine:



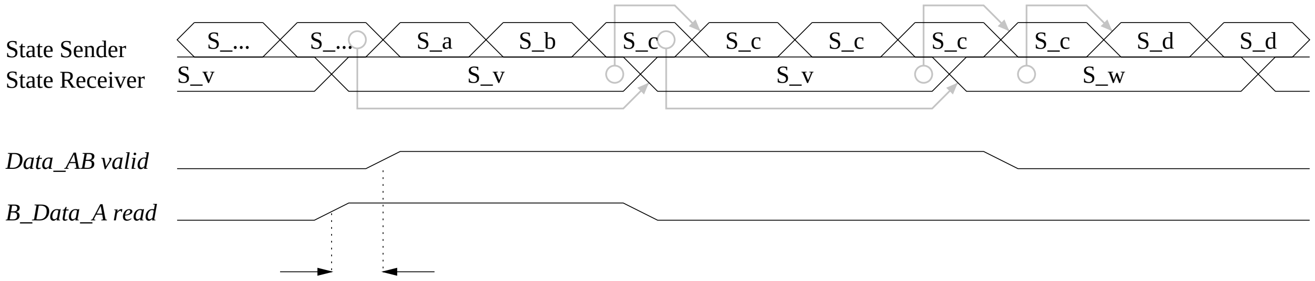
Output may not be valid when it is read again for the final time.

Figure 31: Timing problem when writing

a) For the state machines being FSMs, assume the read command is given in state S_w instead of state S_v.
The following problem is caused, if the writing machine runs much faster than the reading machine:



b) For the state machines being FSMDs assume reading is done in state S_v instead of S_w.
The following problem is caused, if the writing machine runs much faster than the reading machine again:



wrong data read
(and will not be read again in a new cycle)

Figure 32: Timing problems when receiving

5.3 Examples: Data-Transfer Constructions in VHDL

This section presents complete graphical descriptions of the three most important and simple constructions for data-transfer. For all these examples executable VHDL code is provided in the appendix.
In all cases, double handshake is used.

- Direct data-transfer:
  A construction of two state machines with data-transfer
  - Sender (FSMD)
    → Receiver (FSMD)
  - Sender (FSM Controlling Datapath)
    → Receiver (FSM Controlling Datapath)
- Data-transfer via queue:
  Data-transfer from sender to queue and from queue to receiver
  - Sender (FSMD)
    → Queue (FSMD)
    → Receiver (FSMD)
  - Sender (FSM Controlling Datapath)
    → Queue (FSM Controlling Datapath)
    → Receiver (FSM Controlling Datapath)
- Data-transfer via memory:
  An arbiter first grants sender to store data in a memory, then it grants the receiver to read the data.
  - Sender (FSM Controlling Datapath)
    → Memory || Arbiter (FSM) [3]
    → Receiver (FSM Controlling Datapath)

There are no examples of FSMD and "FSM Controlling Datapath" being mixed, just because giving examples for all possibilities would mean too many repetitions. The models can be mixed easily. They just have to be exchanged in the examples. For example refer to the example of sender, queue and receiver, all being FSMDs (figure 35 some pages forward). The FSMD-sender simply can be removed and replaced by the sender which is described by "FSM Controlling Datapath". There are no changes in the connection wires (e.g compare figure 35 and figure 37). In the figures, which show the state machines, the differences in FSMD and FSM are mentioned for sender and receiver.

For each FSMD, there are two versions of VHDL files in the appendix: FSMD described inside a procedure (meant for use without timing) and FSMD in a process (meant for use with clock). Refer to sections 4.1.2 and 4.2.3 for the meaning of these two versions.

The model SFSMD is not provided in the examples for data exchange, because double handshake for SFSMD is identical to double handshake for the procedure-version of FSMD.

Direct data-transfer (first item) implements the basic form of double handshake as described in section 5.2.

When transferring data via a queue, the described double handshake is modified on the receiver side. This is, because the queue controller is passive and does not give a write request or an "output data ready" signal to the receiver. Instead the receiver transmits a read request signal to the queue.

In the data-transfer via memory, double handshake is used in a different way. Double handshake does not supervise the transfer of each word of data any more. This is done by sender and receiver themselves now, when the memory access has been granted to them via double handshake by an arbiter. The arbiter talks to sender and receiver. There is no connection to the memory. Thus, the arbiter always is a simple FSM and there is no FSMD version of it.

The following conventions apply for correspondence between the figures and the VHDL code:
The figures show the logical hierarchy of the constructions, using block elements with solid thick borders. They are given an explanatory name, printed in big letters (e.g. in figure 33 "FSMD A"), except the outer block which covers everything. Each block element corresponds to a VHDL entity. Its name is given in small letters enclosed in parenthesis. Also some signal names are given in small letters. A name is attached to each wire in the figures. In VHDL, the same name is used in the entity-/architecture pair which corresponds to the block in the figure, which has the signal name printed in[4]. If a wire is labeled outside a block, it has the same or a similar name inside the block.
In the state machine figures, the real VHDL signal names are used, too.
Figure 40 uses thick wires. They represent busses.

[3] The arbiter is arranged parallely to the memory. It is connected to sender and receiver but not to the memory.

[4] Except sometimes the trailing word "...Port" is omitted in the signal names in the figures.

5.3.1 Direct Data-Transfer

The following two figures show the block diagrams of the implementation of simple direct data-transfer under cover of the double handshake as described in section 30.

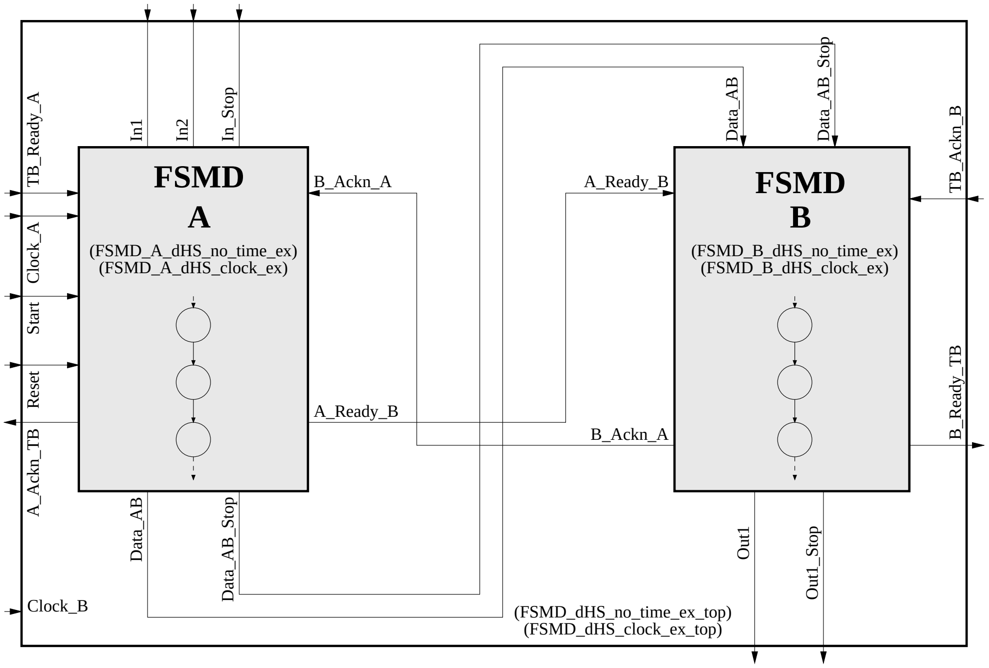


Figure 33: Direct data-transfer with Double Handshake: Block structure for FSMDs

State Machines: See figure 30 for the state machines for direct data-transfer via simple handshake.

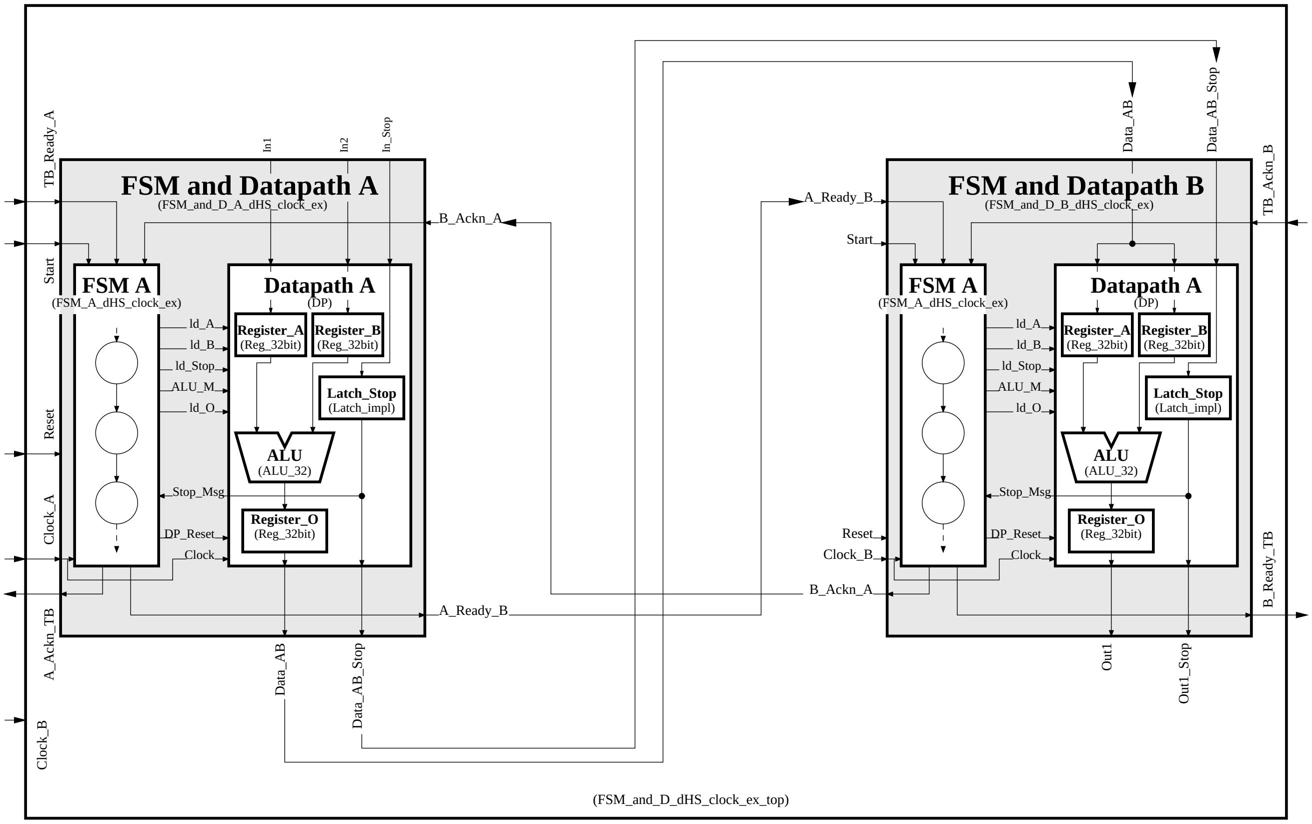


Figure 34: Direct data-transfer with Double Handshake: Refinement to FSMs and datapaths

5.3.2 Data-Transfer via Queue

Figure 35 shows how the queue is inserted in the transmission path. Between FSMD A and the queue, the familiar double handshake is used unchanged[5]. The handshake between queue and receiver (B) is modified however. It also is a double handshake, but it does not start with a "data ready" signal from the queue. As the queue is passive, the receiver has to start with a read request. The state machines in figure 36 show more details: The read request signal `B_Rd_Req_QC='1'` is set in receiver state `S_v`. When the data is ready to be read, the queue gives the signal `QC_Ready_B='1'`. The receiver reads the data in state `S_w` and acknowledges with `B_Rd_Req_QC='0'`, which causes the queue to release `QC_Ready_B` again.

It may appear to be dangerous, that the receiver gives the acknowledge signal (`B_Rd_Req_QC='0'`) at the same time it reads the data, which causes the queue to go to its idle state `0` again. Doing so in an asynchronous design looks dangerous, because the provider of the data may catch the acknowledge a delta time too early. This queue however, has an output buffer and will not discard the data. It keeps the data until the receiver reads another data word. It even keeps the data when the sender sends data to the queue in the meantime.

But: The contexts of the previous paragraph are *not* the reason for the introduction of an output buffer: Common FIFOs for queuing data, usually discard the read set of data with the next clock cycle. This does not meet the requirements in this asynchronous data exchange, because the receiver may be slower than the queue. Therefore the queue output has to be buffered (see also figure 38).

There needs to be a control, which avoids data writing when the queue is full and data reading when the queue is empty. An initial control is build into the queue control: When the queue is empty, the read cycle is not started and `QC_Ready_B='1'` will not be set. This avoids the receiver from reading data. It waits in state `S_v`. The queue control does not start a write cycle when the queue is full and `A_Ready_QC='1'` will not be signaled to the sender. This simple management, provided by the queue controller, allows a safe functionality, even if sender and receiver do not recognize "full" and "empty" signals. This makes it possible to insert a queue between two state machines in a late state of development by doing just minimal changes.

However, the state machines may be large and able to do other tasks instead of waiting in a state for data exchange. Then, they should ask for the signals "full" and "ready", which tell them the reason for their waiting. The signal "Q_Full" in figures 35 and 37 would be connected to the sender A and the signal "Q_Empty" would be connected to the receiver B.

Figure 37 shows the refinement into FSMs and datapaths.

In figure 39, the state machine for the queue is changed into a simple FSM for controlling a datapath. Figure 38 is the datapath for this refined model. It shows the buffering built around an original FIFO-queue. The original queue used here is taken from the book *"Gajski: Principles of Digital Design"* (see references)[6].

Figure 38 does not specify the data width. The depth is assumed to be four. The associated VHDL files of the queue datapath are fully generic, which means, depth and width can be specified by parameters. Thus, these VHDL files also provide an interesting example on how to write generic designs at low structural level. When the datapath is instantiated for connection with the FSM however, the depth is chosen to be four and the data width is 32.

[5] Except from renaming the signal names: The letter "B", which stood for the receiver B, is replaced by "Q" for "queue" or "QC" for "queue control", respectively.

[6] The author of this book says, this design is meant for small queues. He recommends an implementation of large queues by usage of a memory and address pointers.

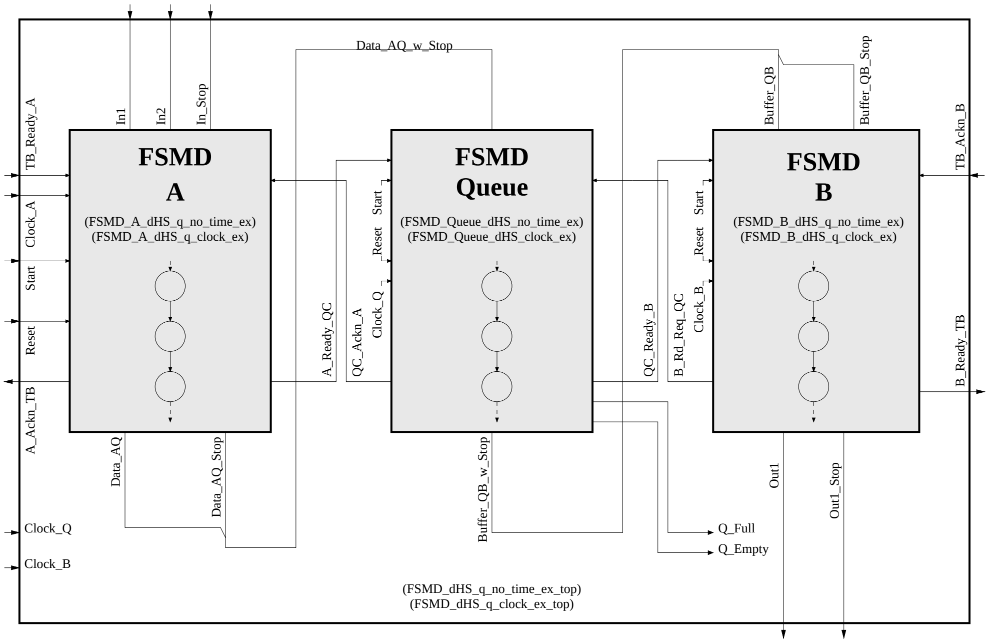


Figure 35: Data-transfer via queue with Double Handshake: Block structure for FSMDs

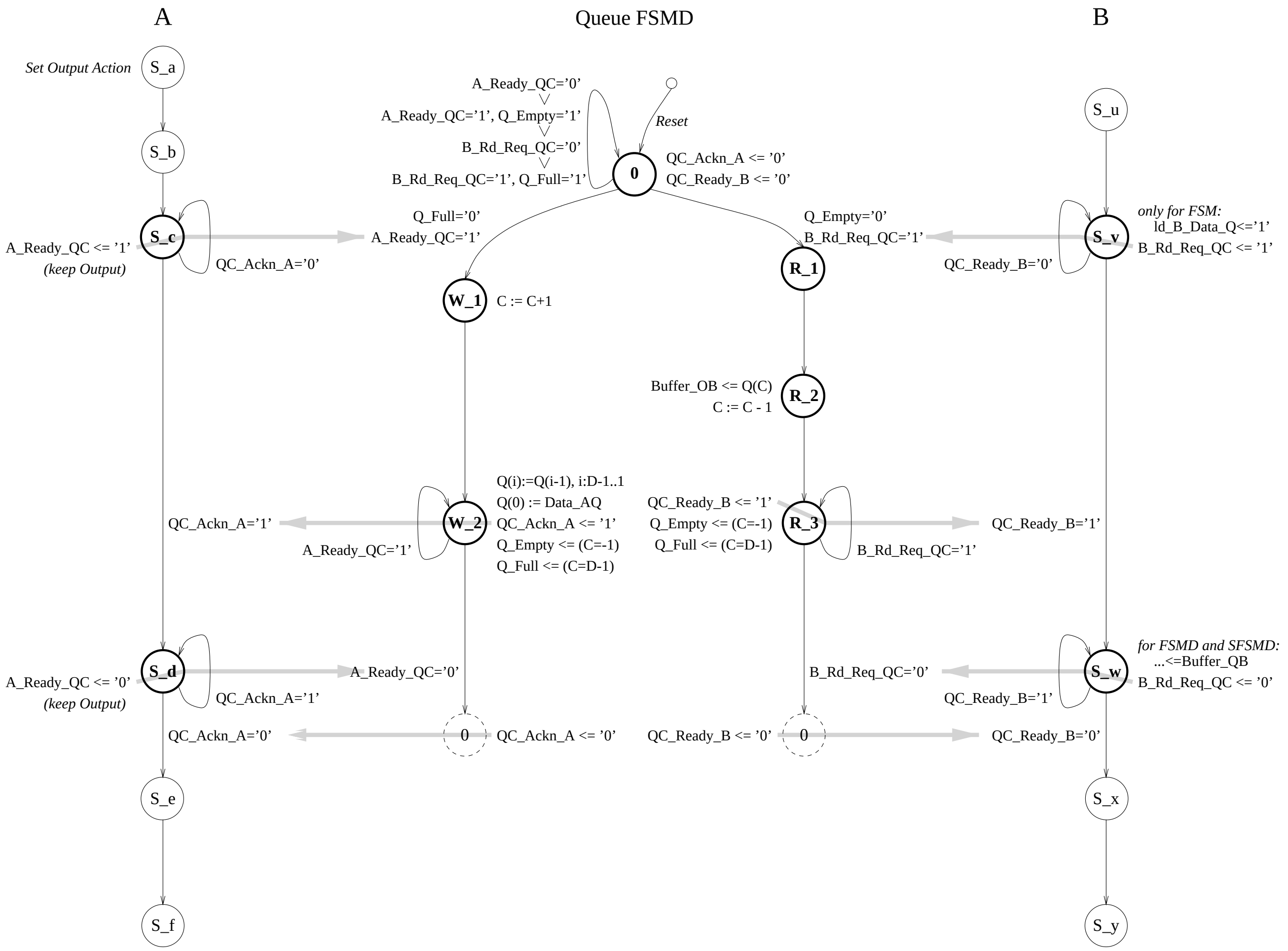


Figure 36: Data-transfer via queue with Double Handshake: FSMDs

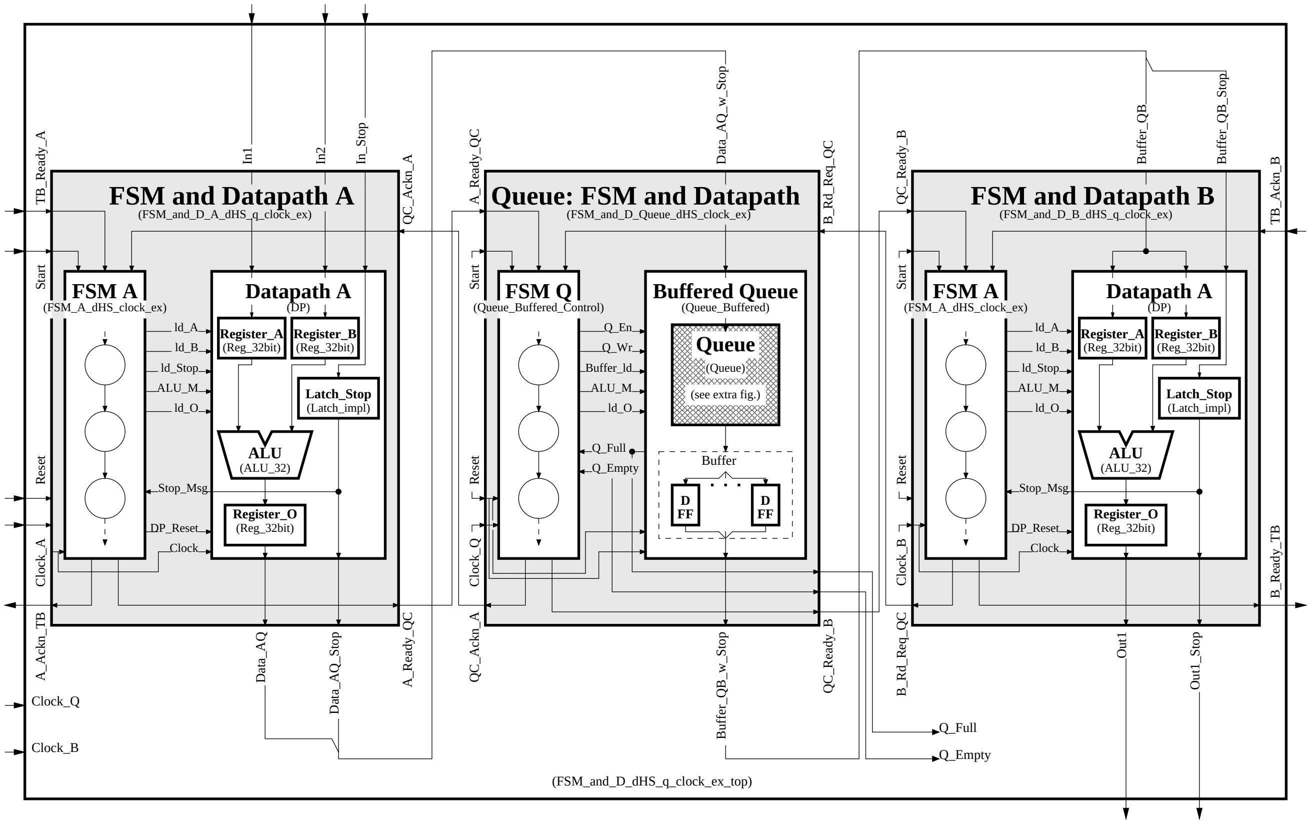
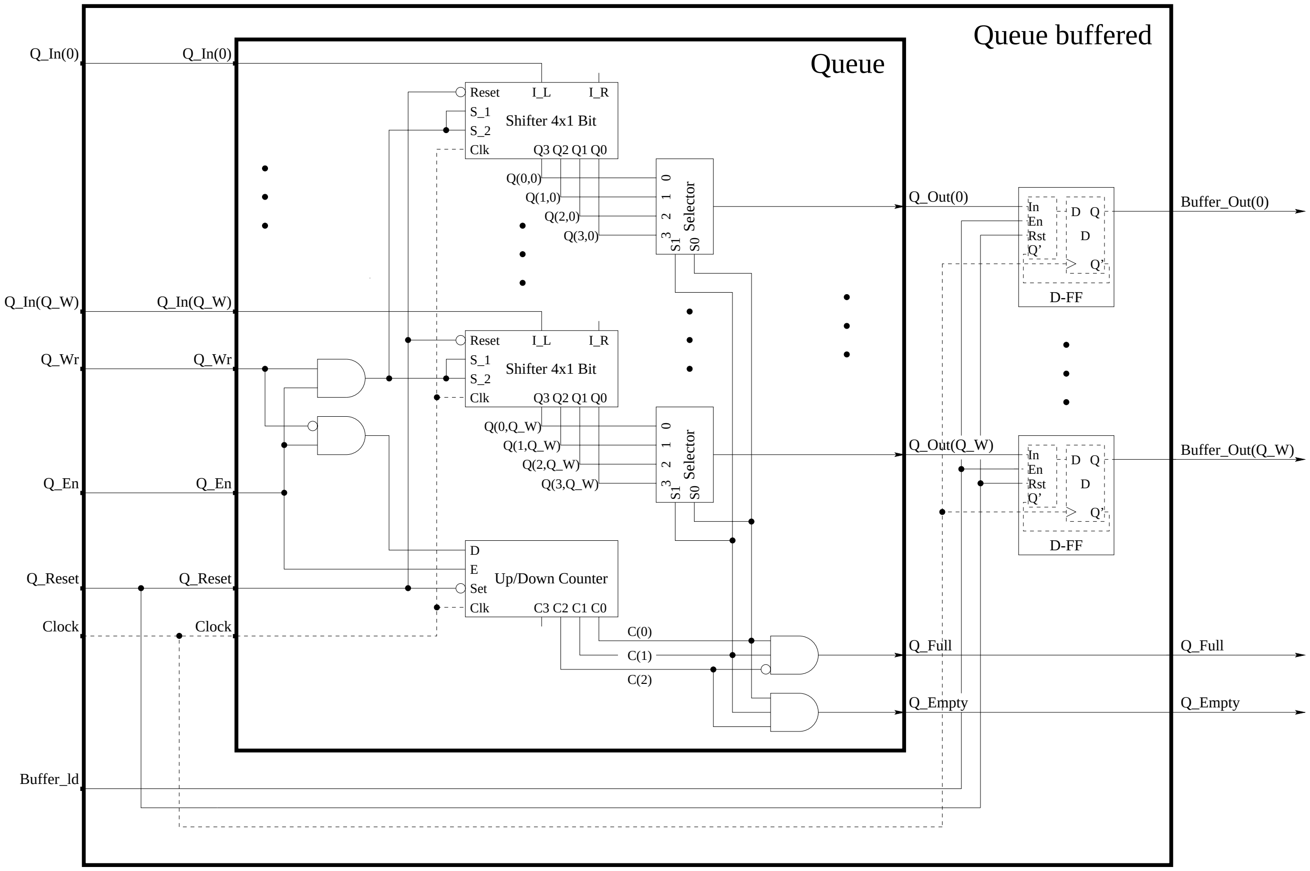


Figure 37: Data-transfer via queue with Double Handshake: Refinement to FSMs and Datapaths



The inner queue is taken from the book "Principles of Digital Design" by D. Gajski. The picture is slightly modified without changes in functionality or general structure. The following explanation of the queue is taken from that book:

"... Whenever data is queued, the shift register shifts data to the right and the counter is incremented. On the other hand, whenever data is read out, the data at the bottom of the queue is selected by the selector and the counter is decremented. Notice that data is not really discarded at that time, but rather is invalidated by decrementing the counter...."

[Write operation is selected by Q_Wr='1' and Q_En='1', Read operation is selected by Q_Wr='0' and Q_En='1']

"... During a read operation, the content of the shift register will not change and the counter will count down by 1. During the write operation, however, the shift register will shift one position to the right and the counter will count up by 1. The counter also controls selection of the proper data during the read operation. As a rule it would be set to 1111 during the initialization, so that its content will be zero when the first data is in the queue. This negative bias of 1 in counting is necessary to accommodate the selector control, which requires a value of 00, 01, 10, or 11 in order to select one of the shifter outputs....
[The signals] Full and Empty ... [are] indicating that the queue is empty whenever the counter content is all 1's, and that the queue is full whenever the counter content is equal to 011. ..."

Figure 38: Datapath of buffered queue

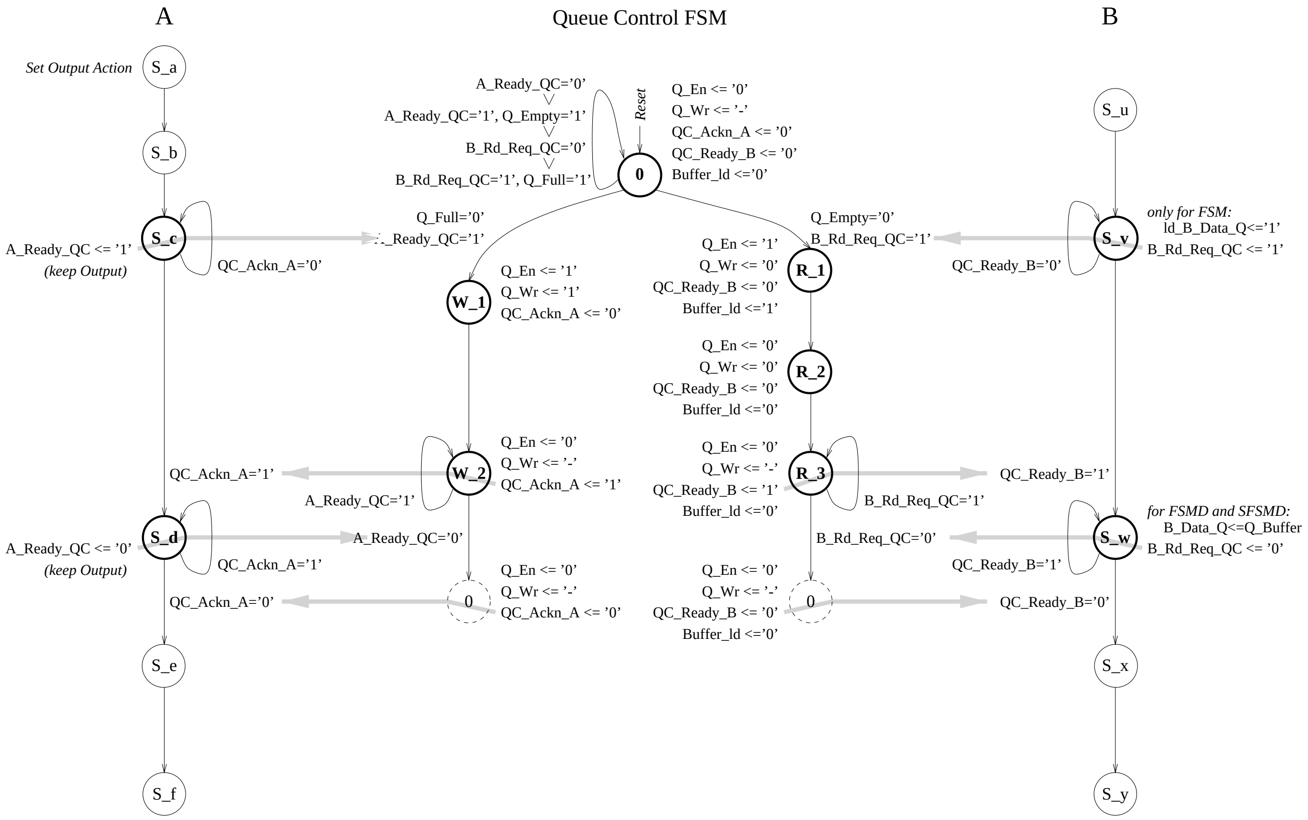


Figure 39: Data-transfer via queue with Double Handshake: FSMs

5.3.3 Data-Transfer via Memory

Two design possibilities come to mind:
One way of modeling a data-transfer via memory is to use a similar mechanism like for the queue design: Sender and receiver request write and read of each data word after selecting a memory address. This solution means using double handshake for each data word. As a result a data-transfer will have a lot of protocol overhead, when the sender sends several words of data continuously or when the receiver reads several words continuously.
Therefore a different solution is presented here: An arbiter grants either the sender or the receiver the permission to access the memory (figure 40). With the permission, the sender or receiver can access the memory directly without any additional protocol overhead. Sender and receiver are connected to the memory by busses (figure 40):

- Command bus: `nWE`, `nOE`
- Address bus: `Addr`
- Data bus: `Data_and_Stop`[7]

Outputs of the FSMs and datapaths, which are connected to a bus, are *tristate outputs*. When a state machine does not access the memory, the tristate driver sets the port to "high-impedance" (which is represented by `'Z'`'s in VHDL). As a result, a non-active state machine behaves indifferently on the bus and an active one can set the bus values. The chip select signal `nCS` (negative logic) is handled differently. It is driven by an AND-gate instead of being connected to a bus. As soon as `A_nCS` or `B_nCS` becomes `'0'`, chip select `nCS` will become `'0'` (= active), too. If `A_nCS` and `B_nCS` both are `'1'`, then `nCS` will become `'1'`, which switches off all memory ports. This could not be done by `nCS` being high-impedant (`'Z'`).

The implementation of data-transfer via memory as provided here also has a disadvantage: Sender and receiver are responsible for correct timing when accessing the memory. When increasing the sender's or the receiver's clock frequency, at some point the data exchange will fail. This can only be avoided with a lot of additional effort and protocol overhead.

Figure 45 in the appendix shows a graphical template, which is sufficient for simple memory types. In that figure, the memory access control state machine assures the correct timing. Many additional handshakes are needed to handle this. The signals printed bold had to be added. The grayed circles are the states where the data is transmitted to or from the memory, finally. For example, the sender starts writing the data by $\overline{\texttt{A\_WE}}$`<='0'`. This causes the access control machine to make the transition to the grayed dotted circle. The dotted circle stands for several states which are passed through for counting the necessary setup time for the memory. When the setup time is counted, the access control gives an OK to the receiver (`AC_nWE_OK='1'`). There are two more (ungrayed) dotted circles in the receive branch of the access control machine. They represent states which count for address and chip enable times (setup time and recovery time). The receiver side works respectively.
Using such an extended memory access management, the clock cycles of sender and receiver can be increased without trouble. No problems for the data-transfer timing will occur.

[7] this is called `Data_and_Stop`, because a stop signal is attached to the data, which signals whether this is the last data word to process or not (see datapath A in figure 40).

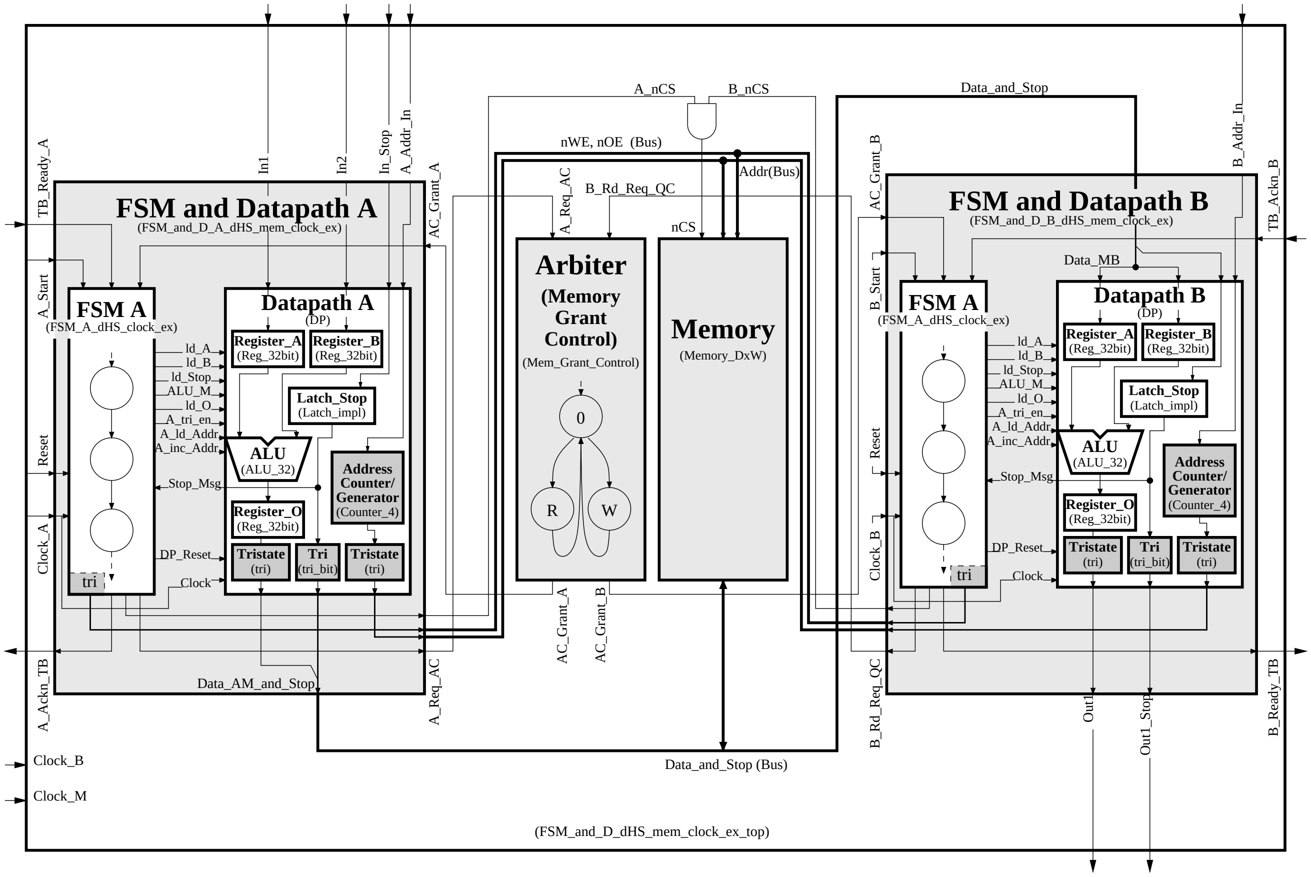


Figure 40: Data-transfer via memory with Double Handshake: Structure

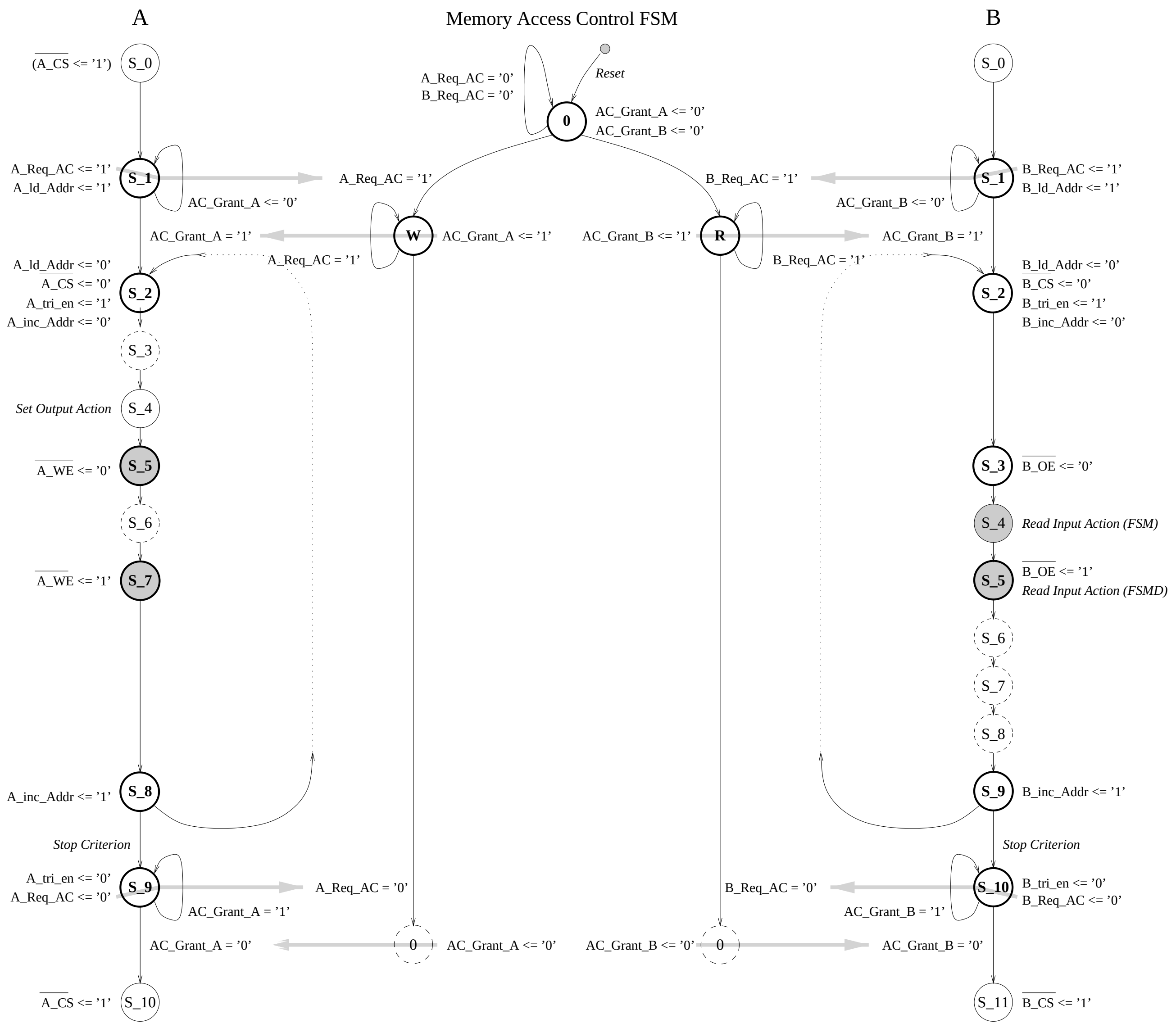


Figure 41: Data-transfer via memory with Double Handshake: FSMs

6 Describing the Models in VHDL – Summary of Problems and Conclusions

This section collects the advantages and disadvantages of using VHDL for describing the models for High Level Hardware Synthesis: SFSMD, FSMD and "FSM Controlling Datapath".

Some problems are not caused by the language VHDL but actually reside within the models themselves. However a VHDL description has to cope with these problems, too. For a *general* overview of the models, their usage and suitability in the High Level Synthesis design flow, refer to section 1.

6.1 Single Models in VHDL

6.1.1 SFSMD and FSMD Without Time

These models do not handle time. They are described in procedures instead of a processes in this report, because a process has its focus on designing with time.

The abstract model SFSMD usually is described by a structured executable high level program. The only relation to hardware is a declaration of input and output ports.

FSMD without time is the original FSMD-model. Scheduling is done, but the description is still executed in zero time.

Advantages of VHDL description. VHDL meets the above mentioned requirements for structured procedural programming. It provides high level programming language-constructs and data types. Compared to other programming languages, VHDL provides a nicer way to declare the inputs and outputs of the algorithm by using an entity.

Disadvantages of VHDL description. Using VHDL as a general purpose programming language, things may become a little complicated, because VHDL is quite unflexible: The programmer is forced to use many hardware-related constructs, even if he does not describe hardware. For example, he has to start with an "architecture" and at least one process in it.
For type conversions, often rigid library functions with varying syntax are needed.
Low level programming for FSMD is easier than high level for SFSMD, however.

In these single state machines, there are just minor difficulties. But when exchanging data between several of them, problems become severe (see below).

6.1.2 FSMD With Clock

The usage of "FSMD With Clock" is advantageous for VHDL descriptions and it helps handling the models in general, too. A model using time is described in a process. (A description in a procedure is possible but not recommended.)

Advantages of this model This clocked model can be used *instead* of the model without time, or, it can be used as a refinement after the model without time. Using FSMD with clock, some timing problems may become obvious in advance and the step towards the model "FSM Controlling Datapath" becomes easier (also see below).

Disadvantages of this model. This model is not the original FSMD model. Usually, time is not regarded at this point.

Advantages of VHDL description. In many VHDL systems, using a clock with a specific cycle time enables additional possibilities for debugging. A full simulation can be done, displaying signals over time and cycles. Also, using the right subset of VHDL, this model may be synthesizable already. Then, a preliminary implementation on a FPGA is possible for testing purposes.
Intelligent high level synthesis tools can turn this model into an optimized model "FSM Controlling Datapath", which would not be possible if the model is described without timing and in a procedure. (An example is the *"Behavioral Compiler"*[8] by *"Synopsys"*.)

On top of this, a data exchange between state machines causes less trouble, when they are described in a process and use a clock.

Disadvantages of VHDL description. There are no significant VHDL related disadvantages or problems.

6.1.3 FSM Controlling Datapath

In this model the state machine does not do the computation, but gives the appropriate command to a datapath.

[8] which can also do rescheduling

Problems of the model. After a command for the datapath is set, it needs one more cycle for the datapath to execute it. This delay is not present in the model FSMD. In order to retain behavioral equivalence between the models, the delay has to be compensated by an early reaction of the FSM to the datapath's results. A resulting problem is the decrease of the allowed clock time.

Once this model-related problem is solved, there is no problem describing it in VHDL. It is done by a separate next state process.

Advantages of VHDL description. VHDL provides a good possibility to block structure the design (FSM, datapath with subblocks) by use of separate descriptions with entities. Many helpful constructs are provided by the language to describe hardware at this refinement level.

Disadvantages of VHDL description. There are no problems.

6.2 Communicating Models in VHDL

In this report, communication means data exchange with hand shake.

6.2.1 SFSMD and FSMD Without Time

Problems of the model Here, handshaking is tried without using timing, because theoretically, handshaking just is based on causality, not timing.

Disadvantages of VHDL description. Only signals can handle ports and communication. Therefore, many of them have to be used in addition to variables. Variables are still needed for sequential computation. For communication, a procedure must be able to catch the change of a signal while it is running. There is also the problem, that a signal can not be passed to a procedure, if the parameters are declared as variables. Due to these facts, the procedures need to access several signals directly, like acessing a "global variable". Coexistence and mixing of signals and variables as well as their different meanings and properties is confusing at these levels of abstraction.
Additionally, VHDL fails to provide a description which really does not use timing. *Wait* statements become necessary because of the signal's delta-delay property. This confuses even more.

Conclusion: Describing handshake in VHDL without using timing is not recommended. For the model SFSMD, sequentialize the procedures of the state machines instead. For FSMD do the same, or switch over to "FSMD with clock".

6.2.2 FSMD With Clock

Introducing handshake into the model "FSMD with clock" is not difficult, because the model is simple: Each model consists of just one process and all actions of a state are executed by this one process. So, communication can be inserted easily. In the following model, there are problems, however.

6.3 FSM Controlling Datapath

When introducing a data exchange with handshaking between two of these models, the handshake signals are connected between the FSMs and the data is exchanged between the datapaths.

Problems of the model In this model, the FSM is used to compensate the delay of the datapath. However, firstly, the data which is exchanged, is not read by a FSM but by another datapath. So, a data exchange via the data ports is not compensated. Secondly, the FSMs compensate a non-existent delay of the handshake signals.

As of these facts, timing of double handshake is diffrent between "FSM Controlling Datapath" and FSMD. This makes the step from the first model to the second complicated and confusing. But these general problems are solved and the solutions can be reused.

Disadvantages of VHDL description. There are no additional VHDL related problems.

6.4 Final Conclusion

In general, for each model, a corresponding VHDL description can be written. However, VHDL is suited better for low level clocked models than for abstract ones. A communication between state machines, which do not use a clock, should be avoided in VHDL.

The delay of data processing, which comes along with the introduction of a datapath, causes many problems. These are solved by this report in a generic way. Templates and examples are provided.

References

[1] Daniel D. Gajski. *Principles of Digital Design*, Prentice Hall, August 1996.

[2] Shuquing Zhao, Daniel D. Gajski. *Communication Synthesis with Custom Hardware*, Technical Report, Department of Information and Computer Science, University of California, Irvine, *coming soon.*

[3] A. Gerstlauer, Shuquing Zhao, Daniel D. Gajski. *Design of a GSM Vocoder using SpecC Methodology*, Technical Report, Department of Information and Computer Science, University of California, Irvine, December 1998.

[4] Shuquing Zhao, Daniel D. Gajski. *Communication Synthesis Issues in SpecC Environment*, Technical Report, Department of Information and Computer Science, University of California, Irvine, June 1999.

A Queue with Single Handshake

A.1 FSMD

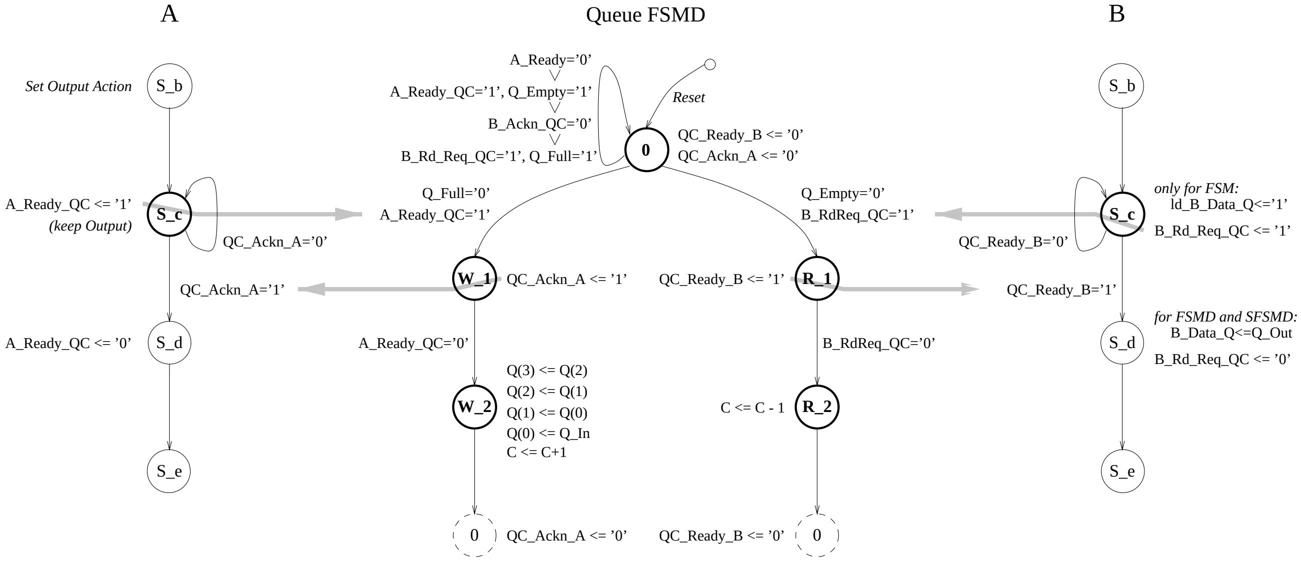


Figure 42: Statemachines for Single Handshake via Queue (FSMD)

A.2 FSM for "FSM Controlling Datapath"

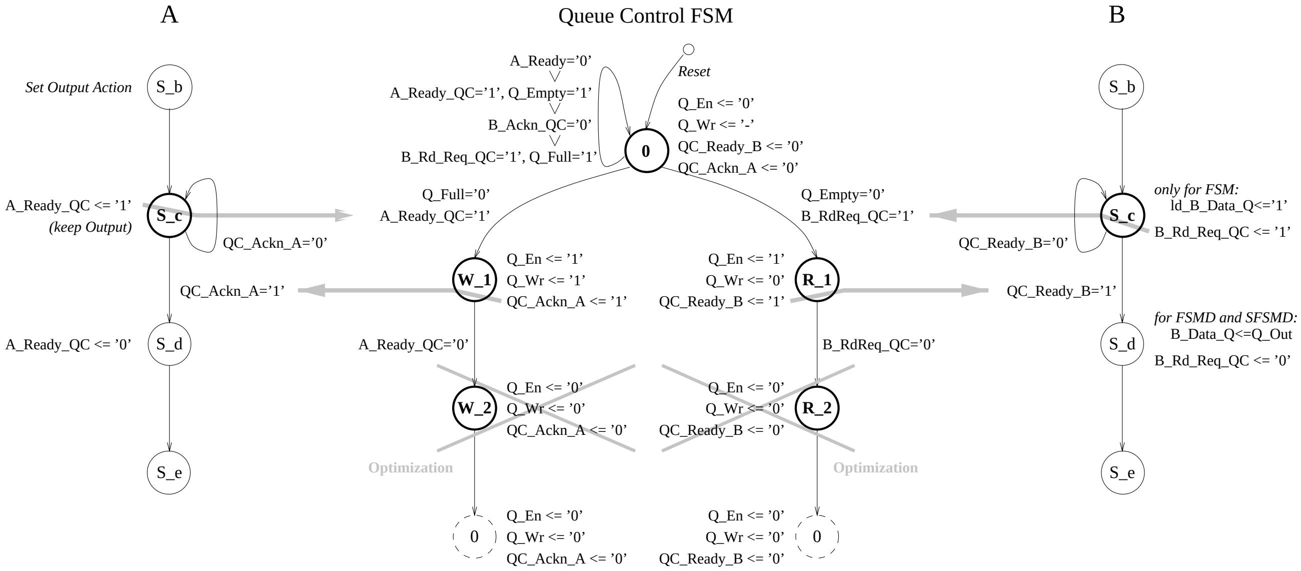


Figure 43: Statemachines for Single Handshake via Queue (FSM Controlling Datapath)

B Extended Ways of Buffered Datatransfer

B.1 FSM for Queue with Simultaneous Read/Write

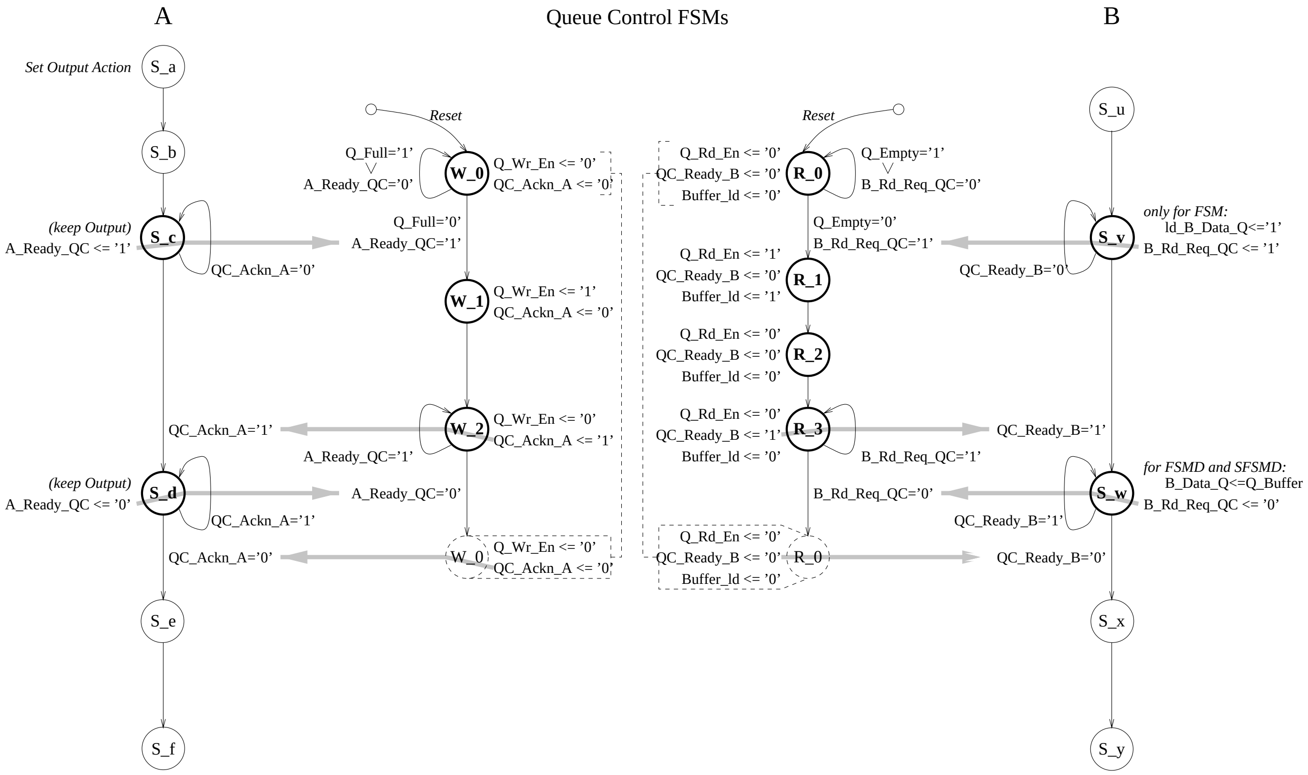


Figure 44: Statemachines for Doublehandshake via Queue with simultaneous Read/Write

B.2 Extended Timing for Memory

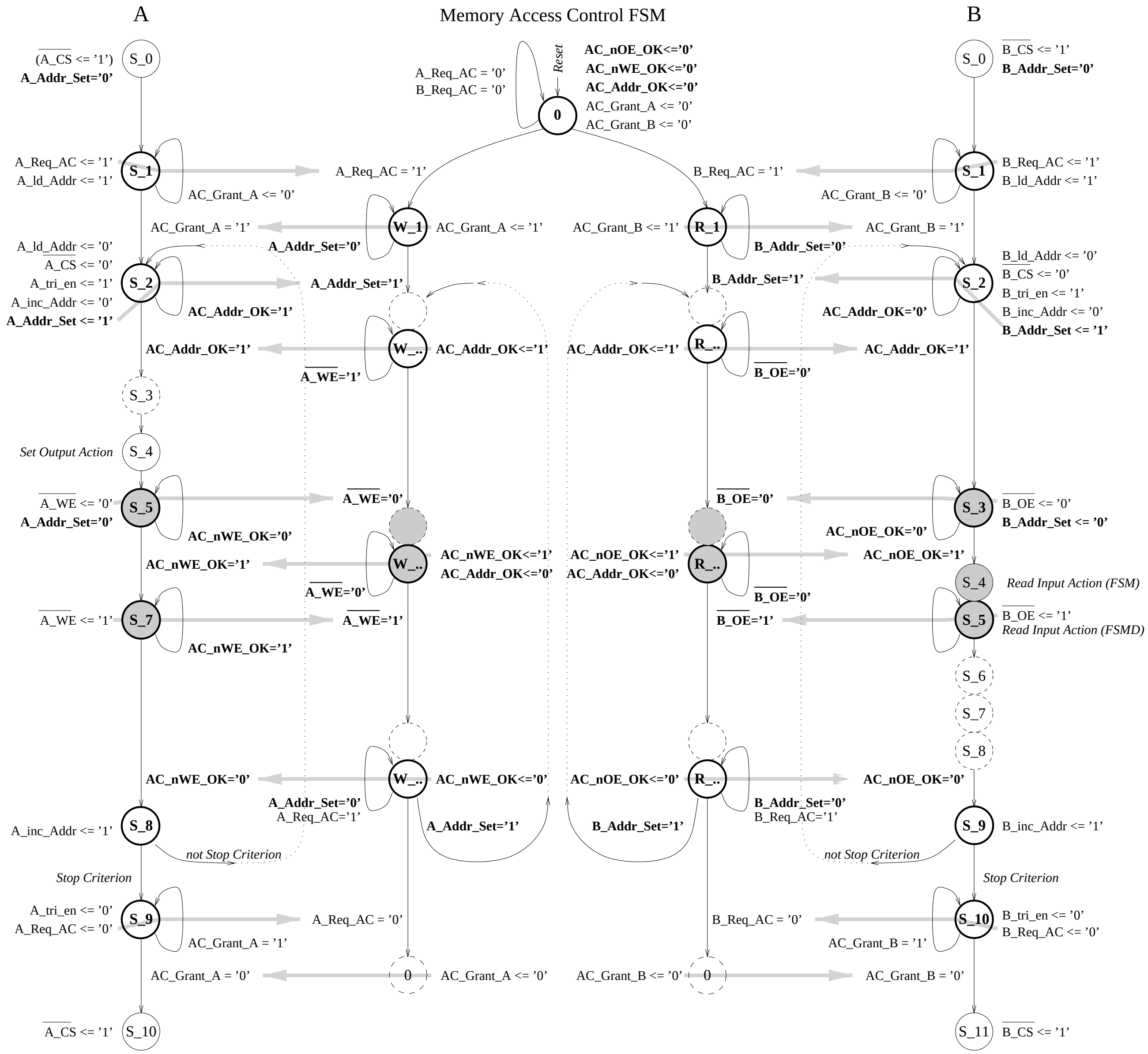


Figure 45: Statemachines with Extended Timing Properties for Doublehandshake via Memory (FSM)

C VHDL examples

This appendix lists only the main files.
Tesbench files and datapath components are not listed. A directory of computer files belongs to this report. There, all files can be found and the examples can be executed.

C.1 Single Finite State Machines

- Single SFSMD
- Single FSMD without time
- Single FSMD using clock
- Single FSM Controlling Datapath
  - FSM
  - Datapath

C.1.1 Single SFSMD

VHDL/plain_SMs/SFSMD/SFSMD_ex.vhd

```
-- example of a SFSMD inherent in a procedure

Entity SFSMD_ex is
   port ( Reset, Start:        in bit;
          In1, In2, In3, In4:  in integer;
          Out1:                out integer);
end SFSMD_ex;


Architecture SFSMD_ex_behavioral of SFSMD_ex is

Procedure behavior1(In1, In2, In3, In4: in integer;
                    signal Out1: out integer) is
type State_Set is (S_1, S_2, S_3, S_END);
-- S_BEGIN is represented by "Procedure not running".
variable next_STATE: State_Set;
variable R, S: integer;

begin
next_STATE:= S_1;
while (next_STATE/=S_END) and (Reset/='1') loop
  case next_STATE is

    when S_1 =>   -- S:= In1 ** In2;   -- (power)
                  S:= 1;
                  for i in In2 downto 1 loop
                    S:= S*In1;
                  end loop;

                  next_STATE:= S_2;

    when S_2 =>   -- R:= In3 ** In4;  -- (power)
                  R:= 1;
                  for i in In4 downto 1 loop
                    R:= R*In3;
                  end loop;

                  next_STATE:= S_3;

    when S_3 =>   Out1<= S-R;

                  next_STATE:= S_END;

    when S_END=>  -- nothing (Procedure quits)

  end case;
end loop;

end behavior1;


begin -- Architecture

P1: Process
begin
  wait until (Reset = '1') or (Start = '1');
  if (Reset = '1') then
    Out1 <= 0;
  elsif (Start = '1') then
    behavior1(In1, In2, In3, In4, Out1);
  end if;
end process;
end SFSMD_ex_behavioral;
```

C.1.2 Single FSMD without time

VHDL/plain_SMs/FSMD_no_time/FSMD_no_time_ex.vhd

```
-- example of a FSMD (Without time) inherent in a procedure

Entity FSMD_no_time_ex is
   port (Reset, Start:        in bit;
         In1, In2:            in integer;
         O_Port:          out integer);
end FSMD_no_time_ex;


Architecture FSMD_no_time_ex_behavioral of FSMD_no_time_ex is

Procedure behavior1(In1, In2: in integer;
                    O: inout integer) is
type State_Set is (S_1, S_2, S_3, S_END);
-- S_BEGIN is represented by "Procedure not running".
variable next_STATE: State_Set;
variable A, B: integer;
variable Mult_temp: integer;

begin
next_STATE:= S_1;
while (next_STATE/=S_END) and (Reset/='1') loop
  case next_STATE is

    when S_1 =>   A := In1;
                  B := In2;
                  O := 1;

                  next_STATE:= S_2;

    when S_2 =>   Mult_temp := O * A;
                  B:= B-1;

                  next_STATE:= S_3;

    when S_3 =>   O := Mult_temp;

                  if (B > 0)
                    then next_STATE:= S_2;
                    else next_STATE:= S_END;
                  end if;

    when S_END=>  -- nothing (Procedure quits)

  end case;
end loop;

end behavior1;


begin -- Architecture

P1: Process
variable O: integer;
begin
  wait until (Reset = '1') or (Start = '1');
  if (Reset = '1') then
    O_Port <= 0;
  elsif (Start = '1') then
    behavior1(In1, In2, O); O_Port <= O;
  end if;
end process;
end FSMD_no_time_ex_behavioral;
```

C.1.3 Single FSMD using clock

VHDL/plain_SMs/FSMD_clock/FSMD_clock_ex.vhd

```
-- example of a FSMD (using clock) inherent in a process

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

Entity FSMD_clock_ex is
   port (Clock:                  in std_logic;
         Reset, Start:           in std_logic;
         Done:                   out std_logic;
         In1, In2:               in std_logic_vector(15 downto 0);
         O_Port:             out std_logic_vector(31 downto 0));
end FSMD_clock_ex;


Architecture FSMD_clock_ex_behavioral of FSMD_clock_ex is

-- behavior outputs
signal O:  std_logic_vector(31 downto 0);
signal Mult_temp:  std_logic_vector(31 downto 0);
type State_Set is (S_BEGIN, S_1, S_2, S_3, S_END);
signal next_STATE: State_Set;
signal A, B: std_logic_vector(15 downto 0);

begin -- Architecture

behavior1: Process(Clock)
begin
  if (Clock'event and Clock = '1') then
    if (Reset = '1') then
      O <= (others=>'0');
      next_STATE <= S_BEGIN;
    else

      case next_STATE is
        when S_BEGIN => Done <= '0';

                         if Start = '1'
                           then next_STATE <= S_1;
                           else next_STATE <= S_BEGIN;
                         end if;

        when S_1 =>    A <= In1;
                       B <= In2;
                       O <= conv_std_logic_vector(1, 32);

                       next_STATE<= S_2;

        when S_2 =>    Mult_temp <= O(15 downto 0) * A;
                       B<= B-conv_std_logic_vector(1, 16);

                       next_STATE<= S_3;

        when S_3 =>    O <= Mult_temp;

                       if (B > conv_std_logic_vector(0, 16))
                         then next_STATE<= S_2;
                         else next_STATE<= S_END;
                       end if;

        when S_END=>   Done <= '1';
```

```
                      if Start = '0'
                        then next_STATE <= S_BEGIN;
                        else next_STATE <= S_END;
                      end if;

      end case;

    end if;
  end if;
end process;

O_Port <= O;

end FSMD_clock_ex_behavioral;
```

C.1.4 Single FSM Controlling Datapath

VHDL/plain_SMs/FSM_structural/FSM_and_D_ex.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

Entity FSM_plus_DP is
   Port ( Clock:          in std_logic;
         Reset, Start:   in std_logic;
         Done:           out std_logic;
         In1, In2:       in std_logic_vector(15 downto 0);
         O_Port:         out std_logic_vector(31 downto 0));
end FSM_plus_DP;

Architecture FSM_plus_DP_behavioral of FSM_plus_DP is

-- connection signals
signal ld_A, ld_B, Count_En, Count_M: std_logic;
signal CMP:          std_logic_vector(1 downto 0);
signal MUX_sel:      std_logic;
signal ld_O:         std_logic;
signal DP_Reset:     std_logic;

component FSM
   Port ( Clock:              in std_logic;
         Reset, Start:       in std_logic;
         Done:               out std_logic;
         DP_Reset:           out std_logic;
         ld_A, ld_B, Count_En, Count_M: out std_logic;
         CMP:                in std_logic_vector(1 downto 0);
         MUX_sel:            out std_logic;
         ld_O:               out std_logic );
end component;

component DP
   port ( Clock:          in std_logic;
         DP_Reset:       in std_logic;
         ld_A, ld_B, Count_En, Count_M: in std_logic;
         CMP:            out std_logic_vector(1 downto 0);
         MUX_sel:        in std_logic;
         ld_O:           in std_logic;
         In1, In2:       in std_logic_vector(15 downto 0);
         O_Port:         out std_logic_vector(31 downto 0));
end component;

begin

Control : FSM
   Port map ( Clock=>Clock,
              Reset=>Reset,
              Start=>Start,
              Done=>Done,
              DP_Reset=>DP_Reset,
              ld_A=>ld_A,
              ld_B=>ld_B,
              Count_En=>Count_En,
              Count_M=>Count_M,
              CMP=>CMP,
              MUX_sel=>MUX_sel,
              ld_O=>ld_O );

Datapath: DP
   Port map ( Clock=>Clock,
              DP_Reset=>DP_Reset,
              ld_A=>ld_A,
                 ld_B=>ld_B,
                 Count_En=>Count_En,
                 Count_M=>Count_M,
                 CMP=>CMP,
                 MUX_sel=>MUX_sel,
                 ld_O=>ld_O,
                 In1=>In1,
                 In2=>In2,
                 O_Port=>O_Port );

end FSM_plus_DP_behavioral;
```

C.1.5 FSM

VHDL/plain_SMs/FSM_structural/FSM_ex.vhd

```
-- example of a FSM inherent in a process

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

Entity FSM is
   port (Clock:             in std_logic;
         Reset, Start:      in std_logic;
         Done:              out std_logic;
         DP_Reset:          out std_logic;
         ld_A, ld_B, Count_En, Count_M: out std_logic;
         CMP:               in std_logic_vector(1 downto 0);
         MUX_sel:           out std_logic;
         ld_O:              out std_logic  );
end FSM;


Architecture FSM_behavioral of FSM is

-- behavior outputs
signal STATE, next_STATE: std_logic_vector(2 downto 0);
signal Enable: std_logic;

begin -- Architecture

Enable <= '1';

State_Register: Process(Clock, Enable)
begin
  if (Clock'event and Clock = '1') then
    if     (Reset = '1') then STATE <= "000";
    elsif  (Enable = '1') then STATE <= next_STATE;
    end if;
  end if;
end Process;


-- Datapath-Control and Status Output Logic

-- Datapath Control Output:
DP_Reset <= ( not STATE(2) and not STATE(1) and not STATE(0) );
ld_A     <= ( not STATE(2) and not STATE(1) and     STATE(0) );
ld_B     <= ( not STATE(2) and not STATE(1) and     STATE(0) );
Count_M  <= ( not STATE(2) and     STATE(1) and not STATE(0) );
Count_En <= ( not STATE(2) and     STATE(1) and not STATE(0) );
MUX_sel  <= ( not STATE(2) and     STATE(1) and not STATE(0) )
         OR ( not STATE(2) and     STATE(1) and     STATE(0) );
ld_O     <= ( not STATE(2) and not STATE(1) and     STATE(0) )
         OR ( not STATE(2) and     STATE(1) and     STATE(0) );

-- Controller Status Output
Done     <= (     STATE(2) and     STATE(1) and     STATE(0) );


-- Next State Logic

next_STATE(2)<=(not STATE(2) and     STATE(1) and     STATE(0) and not CMP(0));
next_STATE(1)<=(not STATE(2) and not STATE(1) and     STATE(0) )
             OR (not STATE(2) and     STATE(1) and not STATE(0) )
             OR (not STATE(2) and     STATE(1) and     STATE(0) and     CMP(0))
             OR (not STATE(2) and     STATE(1) and     STATE(0) and not CMP(0));
next_STATE(0)<=(not STATE(2) and not STATE(1) and not STATE(0) and START )
```

```
            OR (not STATE(2) and     STATE(1) and not STATE(0) )
            OR (not STATE(2) and     STATE(1) and     STATE(0) and not CMP(0));

  end FSM_behavioral;
```

C.1.6 Datapath

VHDL/plain_SMs/Datapath/DP.vhd

```
-- Datapath (DP) for FSM

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

Entity DP is
   port (Clock:           in std_logic;
         DP_Reset:        in std_logic;
         ld_A, ld_B, Count_En, Count_M: in std_logic;
         CMP:             out std_logic_vector(1 downto 0);
         MUX_sel:         in std_logic;
         ld_O:            in std_logic;
         In1, In2:        in std_logic_vector(15 downto 0);
         O_Port:          out std_logic_vector(31 downto 0));
end DP ;

Architecture DP_schematic of DP is

   -- connection signals
   signal A, B: std_logic_vector(15 downto 0);
   signal O: std_logic_vector(31 downto 0);
   signal MULT_Out, MUX_Out: std_logic_vector(31 downto 0);
   signal zero16: std_logic_vector(15 downto 0);
   signal one32: std_logic_vector(31 downto 0);


   component Reg_16bit
      Port ( Clock:    in std_logic;
             Reset:    in std_logic;
             Load:     in std_logic;
             Data_In:  in std_logic_vector (15 downto 0);
             Data_Out: out std_logic_vector (15 downto 0) );
   end component;

   component Reg_32bit
      Port ( Clock:    in std_logic;
             Reset:    in std_logic;
             Load:     in std_logic;
             Data_In:  in std_logic_vector (31 downto 0);
             Data_Out: out std_logic_vector (31 downto 0) );
   end component;

   component Counter
      Port ( Clock:    in std_logic;
             Reset:    in std_logic;
             Load:     in std_logic;
             Enable:   in std_logic;
             Mode:     in std_logic;
             Data_In:  in std_logic_vector (15 downto 0);
             Data_Out: out std_logic_vector (15 downto 0) );
   end component;

   component Comp_16bit
      Port ( Data_In1:   in std_logic_vector(15 downto 0);
             Data_In2:   in std_logic_vector(15 downto 0);
             Result:     out std_logic_vector (1 downto 0) );
   end component;

   component Multiplier
      Port ( Clock:      in std_logic;
             Data_In1:   in std_logic_vector (15 downto 0);
             Data_In2:   in std_logic_vector (15 downto 0);
```

```
                Data_Out:    out std_logic_vector (31 downto 0) );
    end component;

    component MUX_2x32bit
       Port ( Sel:           in std_logic;
                Data_In1:    in std_logic_vector (31 downto 0);
                Data_In2:    in std_logic_vector (31 downto 0);
                Data_Out:    out std_logic_vector (31 downto 0) );
    end component;

begin

zero16 <= (others=>'0');
one32 <= (0=>'1', others=>'0');

Register_A: Reg_16bit
    Port Map ( Clock=>Clock, Reset=>DP_Reset, Load=>ld_A,
                Data_In=>In1, Data_Out=>A);

Counter_B: Counter
    Port Map ( Clock=>Clock, Reset=>DP_Reset, Load=>ld_B,
                Enable=>Count_En, Mode=>Count_M,
                Data_In=>In2, Data_Out=>B);

COMP: Comp_16bit
    Port Map ( Data_In1=>zero16, Data_In2=>B, Result=>CMP);

MULT: Multiplier
    Port Map ( Clock=>Clock, Data_In1=>A, Data_In2=>O(15 downto 0),
                Data_Out=>MULT_Out);

MUX: MUX_2x32bit
    Port Map ( Sel=>MUX_sel, Data_In1=>one32,
                Data_In2=>MULT_Out,
                Data_Out=>MUX_Out);

Register_O: Reg_32bit
    Port Map ( Clock=>Clock, Reset=>DP_Reset, Load=>ld_O,
                Data_In=>MUX_Out, Data_Out=>O);

O_Port <= O;

end DP_schematic;
```

C.2 Double Handshake

- A (FSMD without time) → B (FSMD without time)
  - FSMD A
  - FSMD B
- A (FSMD using clock) → B (FSMD using clock)
  - FSMD A
  - FSMD B
- A (FSM Controlling Datapath) → B (FSM Controlling Datapath)
  - FSM Controlling Datapath A
    - FSM of A
    - Datapath of A (identical B)
  - FSM Controlling Datapath B
    - FSM of B
    - Datapath of B (see B, identical B)

C.2.1 A (FSMD without time) → B (FSMD without time)

VHDL/doubleHS_SMs/FSMD_dHS_no_time/FSMD_dHS_no_time_ex_top.vhd

```
-- File :          FSMD_dHS_no_time_ex_top.vhd
-- bounding the two parts of the example
-- for Example: FSMD (A) --> FSMD (B)
--                 using double handshake protocol

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use STD.textio.all;

entity FSMD_dHS_no_time_ex_top is
   port (Reset, Start:        in bit;
         TB_Ready_A, TB_Ackn_B: in bit;
         A_Ackn_TB, B_Ready_TB: out bit;
         In1, In2:            in integer;
         In_Stop:             in bit;
         Out1_Port:          out integer;
         Out1_Stop_Port:     out bit);
end FSMD_dHS_no_time_ex_top;

architecture FSMD_dHS_no_time_ex_top_arch of FSMD_dHS_no_time_ex_top is

Component FSMD_A_dHS_no_time_ex
   port (Reset, Start:        in bit;
         TB_Ready_A, B_Ackn_A: in bit;
         A_Ackn_TB, A_Ready_B: out bit;
         In1, In2:            in integer;
         In_Stop:             in bit;
         Data_AB_Port:          out integer;
         Data_AB_Stop_Port:     out bit);
end Component;

Component FSMD_B_dHS_no_time_ex
   port (Reset, Start:        in bit;
         A_Ready_B, TB_Ackn_B: in bit;
         B_Ackn_A, B_Ready_TB: out bit;
         Data_AB:           in integer;
         Data_AB_Stop:           in bit;
         Out1_Port:          out integer;
         Out1_Stop_Port:     out bit);
end Component;

signal B_Ackn_A, A_Ready_B: bit;
signal Data_AB:             integer;
signal Data_AB_Stop:        bit;

begin

FSMD_A_dHS_no_time_I: FSMD_A_dHS_no_time_ex
     port map ( Reset=>Reset, Start=>Start,
         TB_Ready_A=>TB_Ready_A, B_Ackn_A=>B_Ackn_A,
         A_Ackn_TB=>A_Ackn_TB, A_Ready_B=>A_Ready_B,
         In1=>In1, In2=>In2,
         In_Stop=>In_Stop,
         Data_AB_Port=>Data_AB,
         Data_AB_Stop_Port=>Data_AB_Stop );

FSMD_B_dHS_no_time_I: FSMD_B_dHS_no_time_ex
     port map ( Reset=>Reset, Start=>Start,
         A_Ready_B=>A_Ready_B, TB_Ackn_B=>TB_Ackn_B,
         B_Ackn_A=>B_Ackn_A, B_Ready_TB=>B_Ready_TB,
         Data_AB=>Data_AB,
         Data_AB_Stop=>Data_AB_Stop,
            Out1_Port=>Out1_Port,
            Out1_Stop_Port=>Out1_Stop_Port );

end FSMD_dHS_no_time_ex_top_arch;
```

C.2.2 FSMD A

VHDL/doubleHS_SMs/FSMD_dHS_no_time/FSMD_dHS_no_time_ex_A.vhd

```
-- File :          FSMD_dHS_no_time_ex_A.vhd
-- implements :    FSMD A
-- of Model:       FSMD (without using time)
-- for Example:    FSMD (A) --> FSMD (B)
--                 using double handshake protocol

Entity FSMD_A_dHS_no_time_ex is
   port (Reset, Start:          in bit;
         TB_Ready_A, B_Ackn_A:  in bit;
         A_Ackn_TB, A_Ready_B:  out bit;
         In1, In2:              in integer;
         In_Stop:               in bit;
         Data_AB_Port:            out integer;
         Data_AB_Stop_Port:            out bit);
end FSMD_A_dHS_no_time_ex;


Architecture FSMD_A_dHS_no_time_ex_behavioral of FSMD_A_dHS_no_time_ex is

-- behavior outputs
signal Data_AB:  integer;
signal Data_AB_Stop: bit;

Procedure behavior1(signal TB_Ready_A, B_Ackn_A: in bit;
                    signal A_Ackn_TB, A_Ready_B: out bit;
                    signal In1, In2: in integer;
                    signal In_Stop: in bit;
                    signal Data_AB: inout integer;
                    signal Data_AB_Stop: out bit) is
type State_Set is (S_1, S_2, S_3, S_4, S_5, S_6, S_END);
-- S_BEGIN is represented by "Procedure not running".
variable next_STATE: State_Set;
variable A, B: integer;
variable Stop: bit;

begin
next_STATE:= S_1;
Data_AB <= 0;
Data_AB_Stop <= '0';
A_Ackn_TB<='0';
A_Ready_B<='0';

while (next_STATE/=S_END) and (Reset/='1') loop
  case next_STATE is
    when S_1 =>

                 if (TB_Ready_A='1')
                   then next_STATE:= S_2;
                   else next_STATE:= S_1;
                 end if;

    when S_2 =>  A := In1;
                 B := In2;
                 Stop := In_Stop;
                 A_Ackn_TB<='1';

                 if (TB_Ready_A='0')
                   then next_STATE:= S_3;
                   else next_STATE:= S_2;
                 end if;

    when S_3 =>  Data_AB <= A - B;
                 Data_AB_Stop <= Stop;
```

```
                    A_Ackn_TB<='0';

                    next_STATE:= S_4;

    when S_4 =>

                    next_STATE:= S_5;

    when S_5 =>     A_Ready_B<='1';

                    if (B_Ackn_A='1')
                      then next_STATE:=S_6;
                      else next_STATE:=S_5;
                    end if;

    when S_6 =>     A_Ready_B<='0';

                    if (B_Ackn_A='0')
                      then if (Stop='1')
                             then next_STATE:=S_END;
                             else next_STATE:=S_1;
                           end if;
                      else next_STATE:=S_6;
                    end if;

    when S_END=>    -- nothing (Procedure quits)

  end case;
  wait for 1 ns; -- Synopsys needs wait > 0 ns
end loop;

end behavior1;


begin -- Architecture

P1: Process
begin
  wait until (Reset = '1') or (Start = '1');
  if (Reset = '1') then
    Data_AB <= 0;
    A_Ackn_TB<='0';
    A_Ready_B<='0';
  elsif (Start = '1') then
    behavior1(TB_Ready_A, B_Ackn_A, A_Ackn_TB, A_Ready_B,
              In1, In2, In_Stop, Data_AB, Data_AB_Stop);
  end if;
end process;

-- Entity Outputs
Data_AB_Port <= Data_AB;
Data_AB_Stop_Port <= Data_AB_Stop;

end FSMD_A_dHS_no_time_ex_behavioral;
```

C.2.3 FSMD B

`VHDL/doubleHS_SMs/FSMD_dHS_no_time/FSMD_dHS_no_time_ex_B.vhd`

```
-- File :          FSMD_dHS_no_time_ex_B.vhd
-- implements :    FSMD B
-- of Model:       FSMD (without using time)
-- for Example:    FSMD (A) --> FSMD (B)
--                 using double handshake protocol

Entity FSMD_B_dHS_no_time_ex is
   port (Reset, Start:          in bit;
         A_Ready_B, TB_Ackn_B: in bit;
         B_Ackn_A, B_Ready_TB: out bit;
         Data_AB:              in integer;
         Data_AB_Stop:              in bit;
         Out1_Port:             out integer;
         Out1_Stop_Port:        out bit);
end FSMD_B_dHS_no_time_ex;


Architecture FSMD_B_dHS_no_time_ex_behavioral of FSMD_B_dHS_no_time_ex is

-- behavior outputs
signal Out1: integer;
signal Out1_Stop: bit;

Procedure behavior1(signal A_Ready_B, TB_Ackn_B: in bit;
                    signal B_Ackn_A, B_Ready_TB: out bit;
                    signal Data_AB: in integer;
                    signal Data_AB_Stop: in bit;
                    signal Out1: inout integer;
                    signal Out1_Stop: out bit) is
type State_Set is (S_1, S_2, S_3, S_4, S_5, S_6, S_END);
-- S_BEGIN is represented by ”Procedure not running”.
variable next_STATE: State_Set;
variable C: integer;
variable Stop: bit;

begin
next_STATE:= S_1;
Out1 <= 0;
Out1_Stop <= '0';
B_Ackn_A<='0';
B_Ready_TB<='0';

while (next_STATE/=S_END) and (Reset/='1') loop
  case next_STATE is
    when S_1 =>

                 if (A_Ready_B='1')
                   then next_STATE:= S_2;
                   else next_STATE:= S_1;
                 end if;

    when S_2 =>  C := Data_AB;
                 Stop := Data_AB_Stop;
                 B_Ackn_A<='1';

                 if (A_Ready_B='0')
                   then next_STATE:= S_3;
                   else next_STATE:= S_2;
                 end if;

    when S_3 =>  Out1 <= 2 * C;
                 Out1_Stop <= Stop;
                 B_Ackn_A<='0';
```

```
                    next_STATE:= S_4;

    when S_4 =>

                    next_STATE:= S_5;

    when S_5 =>     B_Ready_TB<='1';

                    if (TB_Ackn_B='1')
                      then next_STATE:=S_6;
                      else next_STATE:=S_5;
                    end if;

    when S_6 =>     B_Ready_TB<='0';

                    if (TB_Ackn_B='0')
                      then if (Stop='1')
                             then next_STATE:=S_END;
                             else next_STATE:=S_1;
                           end if;
                      else next_STATE:=S_6;
                    end if;

    when S_END=>    -- nothing (Procedure quits)

   end case;
   wait for 1 ns; -- Synopsys needs wait > 0 ns
 end loop;

 end behavior1;


 begin -- Architecture

 P1: Process
 begin
   wait until (Reset = '1') or (Start = '1');
   if (Reset = '1') then
     Out1 <= 0;
     B_Ackn_A<='0';
     B_Ready_TB<='0';
   elsif (Start = '1') then
     behavior1(A_Ready_B, TB_Ackn_B, B_Ackn_A, B_Ready_TB,
               Data_AB, Data_AB_Stop, Out1, Out1_Stop);
   end if;
 end process;

 -- Entity Outputs
 Out1_Port <= Out1;
 Out1_Stop_Port <= Out1_Stop;

 end FSMD_B_dHS_no_time_ex_behavioral;
```

C.2.4 A (FSMD using clock) → B (FSMD using clock)

VHDL/doubleHS_SMs/FSMD_dHS_clock/FSMD_dHS_clock_ex_top.vhd

```
-- File :          FSMD_dHS_clock_ex_top.vhd
-- bounding the two parts of the example
-- for Example: FSMD (A) --> FSMD (B)
--                using double handshake protocol

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use STD.textio.all;

entity FSMD_dHS_clock_ex_top is
   port (Reset, Start:         in std_logic;
         Clock_A, Clock_B:     in std_logic;
         TB_Ready_A, TB_Ackn_B: in std_logic;
         A_Ackn_TB, B_Ready_TB: out std_logic;
         In1, In2:             in integer;
         In_Stop:              in std_logic;
         Out1_Port:           out integer;
         Out1_Stop_Port:      out std_logic);
end FSMD_dHS_clock_ex_top;

architecture FSMD_dHS_clock_ex_top_arch of FSMD_dHS_clock_ex_top is

Component FSMD_A_dHS_clock_ex
   port (Reset, Start:         in std_logic;
         Clock_A:             in std_logic;
         TB_Ready_A, B_Ackn_A: in std_logic;
         A_Ackn_TB, A_Ready_B: out std_logic;
         In1, In2:             in integer;
         In_Stop:              in std_logic;
         Data_AB_Port:            out integer;
         Data_AB_Stop_Port:       out std_logic);
end Component;

Component FSMD_B_dHS_clock_ex
   port (Reset, Start:         in std_logic;
         Clock_B:             in std_logic;
         A_Ready_B, TB_Ackn_B: in std_logic;
         B_Ackn_A, B_Ready_TB: out std_logic;
         Data_AB:            in integer;
         Data_AB_Stop:            in std_logic;
         Out1_Port:           out integer;
         Out1_Stop_Port:      out std_logic);
end Component;

signal B_Ackn_A, A_Ready_B: std_logic;
signal Data_AB:             integer;
signal Data_AB_Stop:        std_logic;

begin

FSMD_A_dHS_clock_I: FSMD_A_dHS_clock_ex
     port map ( Reset=>Reset, Start=>Start, Clock_A=>Clock_A,
         TB_Ready_A=>TB_Ready_A, B_Ackn_A=>B_Ackn_A,
         A_Ackn_TB=>A_Ackn_TB, A_Ready_B=>A_Ready_B,
         In1=>In1, In2=>In2,
         In_Stop=>In_Stop,
         Data_AB_Port=>Data_AB,
         Data_AB_Stop_Port=>Data_AB_Stop );

FSMD_B_dHS_clock_I: FSMD_B_dHS_clock_ex
     port map ( Reset=>Reset, Start=>Start, Clock_B=>Clock_B,
         A_Ready_B=>A_Ready_B, TB_Ackn_B=>TB_Ackn_B,
          B_Ackn_A=>B_Ackn_A, B_Ready_TB=>B_Ready_TB,
          Data_AB=>Data_AB,
          Data_AB_Stop=>Data_AB_Stop,
          Out1_Port=>Out1_Port,
          Out1_Stop_Port=>Out1_Stop_Port );

end FSMD_dHS_clock_ex_top_arch;
```

C.2.5 FSMD A

VHDL/doubleHS_SMs/FSMD_dHS_clock/FSMD_dHS_clock_ex_A.vhd

```
-- File :          FSMD_dHS_clock_ex_A.vhd
-- implements :    FSMD A
-- of Model:       FSMD (using clock)
-- for Example:    FSMD (A) --> FSMD (B)
--                 using double handshake protocol

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

-- FSMD A of example of two FSMD (Without time) exchanging data

Entity FSMD_A_dHS_clock_ex is
   port (Reset, Start:          in std_logic;
         Clock_A:              in std_logic;
         TB_Ready_A, B_Ackn_A: in std_logic;
         A_Ackn_TB, A_Ready_B: out std_logic;
         In1, In2:              in integer;
         In_Stop:               in std_logic;
         Data_AB_Port:            out integer;
         Data_AB_Stop_Port:            out std_logic);
end FSMD_A_dHS_clock_ex;


Architecture FSMD_A_dHS_clock_ex_behavioral of FSMD_A_dHS_clock_ex is
-- outputs
signal Data_AB:  integer;
signal Data_AB_Stop: std_logic;

-- internal signals
type State_Set is (S_BEGIN, S_1, S_2, S_3, S_4, S_5, S_6, S_END);
signal next_STATE: State_Set;
signal A, B: integer;
signal Stop: std_logic;

begin -- Architecture

behavior1: process(Clock_A)
begin
  if (Clock_A'event and Clock_A='1') then
    if (Reset = '1') then
      Data_AB <= 0;
      Data_AB_Stop <= '0';
      A_Ackn_TB<='0';
      A_Ready_B<='0';
      next_STATE <= S_BEGIN;
    else
      case next_STATE is

        when S_BEGIN => Data_AB <= 0;
                        Data_AB_Stop <= '0';
                        A_Ackn_TB<='0';
                        A_Ready_B<='0';

                        if Start = '1'
                          then next_STATE <= S_1;
                          else next_STATE <= S_BEGIN;
                        end if;

        when S_1 =>

                       if (TB_Ready_A='1')
                         then next_STATE<= S_2;
                         else next_STATE<= S_1;
                       end if;

         when S_2 =>   A <= In1;
                       B <= In2;
                       Stop <= In_Stop;
                       A_Ackn_TB<='1';

                       if (TB_Ready_A='0')
                         then next_STATE<= S_3;
                         else next_STATE<= S_2;
                       end if;

         when S_3 =>   Data_AB <= A - B;
                       Data_AB_Stop <= Stop;
                       A_Ackn_TB<='0';

                       next_STATE<= S_4;

         when S_4 =>

                       next_STATE<= S_5;

         when S_5 =>   A_Ready_B<='1';

                       if (B_Ackn_A='1')
                         then next_STATE<=S_6;
                         else next_STATE<=S_5;
                       end if;

         when S_6 =>   A_Ready_B<='0';

                       if (B_Ackn_A='0')
                         then if (Stop='1')
                                then next_STATE<=S_END;
                                else next_STATE<=S_1;
                              end if;
                         else next_STATE<=S_6;
                       end if;

         when S_END=>  -- nothing (Procedure quits)

       end case;
     end if;
   end if;

end process;


-- Entity Outputs
Data_AB_Port <= Data_AB;
Data_AB_Stop_Port <= Data_AB_Stop;

end FSMD_A_dHS_clock_ex_behavioral;
```

C.2.6 FSMD B

VHDL/doubleHS_SMs/FSMD_dHS_clock/FSMD_dHS_clock_ex_B.vhd

```
-- File :         FSMD_dHS_clock_ex_B.vhd
-- implements :   FSMD B
-- of Model:      FSMD (using clock)
-- for Example:   FSMD (A) --> FSMD (B)
--                using double handshake protocol

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

-- FSMD B of example of two FSMD (Without time) exchanging data

Entity FSMD_B_dHS_clock_ex is
   port (Reset, Start:         in std_logic;
         Clock_B:              in std_logic;
         A_Ready_B, TB_Ackn_B: in std_logic;
         B_Ackn_A, B_Ready_TB: out std_logic;
         Data_AB:              in integer;
         Data_AB_Stop:              in std_logic;
         Out1_Port:            out integer;
         Out1_Stop_Port:       out std_logic);
end FSMD_B_dHS_clock_ex;


Architecture FSMD_B_dHS_clock_ex_behavioral of FSMD_B_dHS_clock_ex is
-- outputs
signal Out1:  integer;
signal Out1_Stop: std_logic;

-- internal signals
type State_Set is (S_BEGIN, S_1, S_2, S_3, S_4, S_5, S_6, S_END);
-- S_BEGIN is represented by "Procedure not running".
signal next_STATE: State_Set;
signal C: integer;
signal Stop: std_logic;

begin -- Architecture

behavior1: process(Clock_B)
begin
  if (Clock_B'event and Clock_B='1') then
    if (Reset = '1') then
      Out1 <= 0;
      Out1_Stop <='0';
      B_Ackn_A<='0';
      B_Ready_TB<='0';
      next_STATE <= S_BEGIN;
    else
      case next_STATE is

        when S_BEGIN => Out1 <= 0;
                        Out1_Stop <='0';
                        B_Ackn_A<='0';
                        B_Ready_TB<='0';

                        if Start = '1'
                          then next_STATE <= S_1;
                          else next_STATE <= S_BEGIN;
                        end if;

        when S_1 =>

                      if (A_Ready_B='1')
```

```
                        then next_STATE<= S_2;
                        else next_STATE<= S_1;
                      end if;

        when S_2 =>   C <= Data_AB;
                      Stop <= Data_AB_Stop;
                      B_Ackn_A<='1';

                      if (A_Ready_B='0')
                        then next_STATE<= S_3;
                        else next_STATE<= S_2;
                      end if;

        when S_3 =>   Out1 <= 2 * C;
                      Out1_Stop <= Stop;
                      B_Ackn_A<='0';

                      next_STATE<= S_4;

        when S_4 =>

                      next_STATE<= S_5;

        when S_5 =>   B_Ready_TB<='1';

                      if (TB_Ackn_B='1')
                        then next_STATE<=S_6;
                        else next_STATE<=S_5;
                      end if;

        when S_6 =>   B_Ready_TB<='0';

                      if (TB_Ackn_B='0')
                        then if (Stop='1')
                               then next_STATE<=S_END;
                               else next_STATE<=S_1;
                             end if;
                        else next_STATE<=S_6;
                      end if;

        when S_END=>  -- nothing (Procedure quits)

      end case;
     end if;
   end if;

end process;


-- Entity Outputs
Out1_Port <= Out1;
Out1_Stop_Port <= Out1_Stop;

end FSMD_B_dHS_clock_ex_behavioral;
```

C.2.7 A (FSM Controlling Datapath) → B (FSM Controlling Datapath)

VHDL/doubleHS_SMs/FSM_dHS_clock/FSM_and_D_dHS_clock_ex_top.vhd

```
-- File :          FSM_and_D_dHS_clock_ex_top.vhd
-- bounding the two parts of the example
-- for Example: FSM,DP (A) --> FSM,DP (B)
--               using double handshake protocol

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use STD.textio.all;

entity FSM_and_D_dHS_clock_ex_top is
   port (Reset, Start:        in std_logic;
         Clock_A, Clock_B:    in std_logic;
         TB_Ready_A, TB_Ackn_B: in std_logic;
         A_Ackn_TB, B_Ready_TB: out std_logic;
         In1, In2:            in std_logic_vector(31 downto 0);
         In_Stop:             in std_logic;
         Out1_Port:          out std_logic_vector(31 downto 0);
         Out1_Stop_Port:     out std_logic);
end FSM_and_D_dHS_clock_ex_top;

architecture FSM_and_D_dHS_clock_ex_top_arch of FSM_and_D_dHS_clock_ex_top is

Component FSM_and_D_A_dHS_clock_ex
   port (Reset, Start:        in std_logic;
         Clock_A:            in std_logic;
         TB_Ready_A, B_Ackn_A: in std_logic;
         A_Ackn_TB, A_Ready_B: out std_logic;
         In1, In2:            in std_logic_vector(31 downto 0);
         In_Stop:             in std_logic;
         Data_AB_Port:          out std_logic_vector(31 downto 0);
         Data_AB_Stop_Port:     out std_logic);
end Component;

Component FSM_and_D_B_dHS_clock_ex
   port (Reset, Start:        in std_logic;
         Clock_B:            in std_logic;
         A_Ready_B, TB_Ackn_B: in std_logic;
         B_Ackn_A, B_Ready_TB: out std_logic;
         Data_AB:           in std_logic_vector(31 downto 0);
         Data_AB_Stop:          in std_logic;
         Out1_Port:          out std_logic_vector(31 downto 0);
         Out1_Stop_Port:     out std_logic);
end Component;

signal B_Ackn_A, A_Ready_B: std_logic;
signal Data_AB:             std_logic_vector(31 downto 0);
signal Data_AB_Stop:        std_logic;

begin

FSM_and_D_A_dHS_clock_I: FSM_and_D_A_dHS_clock_ex
     port map ( Reset=>Reset, Start=>Start, Clock_A=>Clock_A,
         TB_Ready_A=>TB_Ready_A, B_Ackn_A=>B_Ackn_A,
         A_Ackn_TB=>A_Ackn_TB, A_Ready_B=>A_Ready_B,
         In1=>In1, In2=>In2,
         In_Stop=>In_Stop,
         Data_AB_Port=>Data_AB,
         Data_AB_Stop_Port=>Data_AB_Stop );

FSM_and_D_B_dHS_clock_I: FSM_and_D_B_dHS_clock_ex
     port map ( Reset=>Reset, Start=>Start, Clock_B=>Clock_B,
         A_Ready_B=>A_Ready_B, TB_Ackn_B=>TB_Ackn_B,
            B_Ackn_A=>B_Ackn_A, B_Ready_TB=>B_Ready_TB,
            Data_AB=>Data_AB,
            Data_AB_Stop=>Data_AB_Stop,
            Out1_Port=>Out1_Port,
            Out1_Stop_Port=>Out1_Stop_Port );

end FSM_and_D_dHS_clock_ex_top_arch;
```

C.2.8 FSM Controlling Datapath A

VHDL/doubleHS_SMs/FSM_dHS_clock/FSM_and_D_dHS_clock_ex_A.vhd

```
-- File :          FSM_and_D_dHS_clock_ex_A.vhd
-- implements :    bounding of FSM A and Datapath A
-- of Model:       FSM and separate Datapath (FSM + D)
-- for Example:    FSM,DP (A) --> FSM,DP (B)
--                 using double handshake protocol

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

-- FSM_and_D A of example of two FSM_and_D (Without time) exchanging data

Entity FSM_and_D_A_dHS_clock_ex is
   port (Reset, Start:          in std_logic;
         Clock_A:               in std_logic;
         TB_Ready_A, B_Ackn_A:  in std_logic;
         A_Ackn_TB, A_Ready_B:  out std_logic;
         In1, In2:              in std_logic_vector (31 downto 0);
         In_Stop:               in std_logic;
         Data_AB_Port:          out std_logic_vector (31 downto 0);
         Data_AB_Stop_Port:          out std_logic);
end FSM_and_D_A_dHS_clock_ex;


Architecture FSM_and_D_A_dHS_clock_ex_structural of FSM_and_D_A_dHS_clock_ex is
-- outputs
signal Data_AB:  std_logic_vector (31 downto 0);
signal Data_AB_Stop: std_logic;

signal DP_Reset:     std_logic;
signal ld_A, ld_B, ld_Stop: std_logic;
signal StopMsg: std_logic;
signal ALU_M: std_logic_vector (1 downto 0);
signal ld_O: std_logic;
signal CMP: std_logic_vector(1 downto 0);
signal MUX_sel: std_logic;

Component DP
   port (Clock:            in std_logic;
         DP_Reset:         in std_logic;
         ld_A, ld_B, ld_Stop: in std_logic;
         StopMsg:          out std_logic;
         ALU_M:            in std_logic_vector (1 downto 0);
         ld_O:             in std_logic;
         In1, In2:         in std_logic_vector(31 downto 0);
         In_Stop:          in std_logic;
         O_Port:           out std_logic_vector(31 downto 0);
         Out_Stop:         out std_logic);
end Component;

Component FSM_A_dHS_clock_ex
   port (Reset, Start:          in std_logic;
         Clock_A:               in std_logic;
         TB_Ready_A, B_Ackn_A:  in std_logic;
         A_Ackn_TB, A_Ready_B:  out std_logic;
         DP_Reset:              out std_logic;
         ld_A, ld_B, ld_Stop:   out std_logic;
         StopMsg:               in std_logic;
         ALU_M:                 out std_logic_vector (1 downto 0);
         ld_O:                  out std_logic);
end Component;

begin -- Architecture
```

```
  DP_A_I: DP
      Port Map ( Clock=>Clock_A , DP_Reset=>DP_Reset ,
                 ld_A=>ld_A , ld_B=>ld_B , ld_Stop=>ld_Stop ,
                 StopMsg=>StopMsg, ALU_M=>ALU_M, ld_O=>ld_O ,
                 In1=>In1 , In2=>In2 ,
                 In_Stop=>In_Stop ,
                 O_Port=>Data_AB , Out_Stop=>Data_AB_Stop );

  FSM_A_dHS_clock_I: FSM_A_dHS_clock_ex
      Port Map ( Reset=>Reset , Start=>Start ,
                 Clock_A=>Clock_A ,
                 TB_Ready_A=>TB_Ready_A , B_Ackn_A=>B_Ackn_A ,
                 A_Ackn_TB=>A_Ackn_TB , A_Ready_B=>A_Ready_B ,
                 DP_Reset=>DP_Reset ,
                 ld_A=>ld_A , ld_B=>ld_B , ld_Stop=>ld_Stop ,
                 StopMsg=>StopMsg, ALU_M=>ALU_M, ld_O=>ld_O );

  -- Entity Outputs
  Data_AB_Port <= Data_AB;
  Data_AB_Stop_Port <= Data_AB_Stop;

end FSM_and_D_A_dHS_clock_ex_structural;
```

C.2.9 FSM of A

VHDL/doubleHS_SMs/FSM_dHS_clock/FSM_dHS_clock_ex_A.vhd

```
-- File :          FSM_dHS_clock_ex_A.vhd
-- implements :    FSM A
-- of Model:       FSM and separate Datapath
-- for Example:    FSM,DP (A) --> FSM,DP (B)
--                 using double handshake protocol

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

Entity FSM_A_dHS_clock_ex is
   port (Reset, Start:          in std_logic;
         Clock_A:               in std_logic;
         TB_Ready_A, B_Ackn_A:  in std_logic;
         A_Ackn_TB, A_Ready_B:  out std_logic;
         DP_Reset:              out std_logic;
         ld_A, ld_B, ld_Stop:   out std_logic;
         StopMsg:               in std_logic;
         ALU_M:                 out std_logic_vector (1 downto 0);
         ld_O:                  out std_logic);
end FSM_A_dHS_clock_ex;


Architecture FSM_A_dHS_clock_ex_behavioral of FSM_A_dHS_clock_ex is

type State_Set is (S_BEGIN, S_1, S_2, S_3, S_4, S_5, S_6, S_END);
signal next_STATE, STATE: State_Set;

begin -- Architecture

transition: Process
begin
  wait until (Clock_A'event and Clock_A = '1');
  STATE <= next_STATE;
end process;

behavior1: process(STATE, Start, TB_Ready_A, B_Ackn_A, StopMsg)
begin
    if (Reset = '1') then
         DP_Reset  <= '1';
         A_Ackn_TB <= '0';
         A_Ready_B <= '0';
         ld_A      <= '0';
         ld_B      <= '0';
         ld_Stop   <= '0';
         ALU_M     <= "--";
         ld_O      <= '0';
      next_STATE <= S_BEGIN;
    else
      case STATE is

        when S_BEGIN => DP_Reset  <= '1';
                        A_Ackn_TB <= '0';
                        A_Ready_B <= '0';
                        ld_A      <= '0';
                        ld_B      <= '0';
                        ld_Stop   <= '0';
                        ALU_M     <= "--";
                        ld_O      <= '0';

                       if Start = '1'
                         then next_STATE <= S_1;
                         else next_STATE <= S_BEGIN;
                        end if;

          when S_1 =>     DP_Reset   <= ’0’;
                          A_Ackn_TB  <= ’0’;
                          A_Ready_B  <= ’0’;
                          ld_A       <= ’0’;
                          ld_B       <= ’0’;
                          ld_Stop    <= ’0’;
                          ALU_M      <= ”--”;
                          ld_O       <= ’0’;

                        if (TB_Ready_A=’1’)
                          then next_STATE<= S_2;
                          else next_STATE<= S_1;
                        end if;

           when S_2 =>    DP_Reset   <= ’0’;
                          A_Ackn_TB  <= ’1’;
                          A_Ready_B  <= ’0’;
                          ld_A       <= ’1’;
                          ld_B       <= ’1’;
                          ld_Stop    <= ’1’;
                          ALU_M      <= ”11”;
                          ld_O       <= ’0’;

                        if (TB_Ready_A=’0’)
                          then next_STATE<= S_3;
                          else next_STATE<= S_2;
                        end if;

           when S_3 =>    DP_Reset   <= ’0’;
                          A_Ackn_TB  <= ’0’;
                          A_Ready_B  <= ’0’;
                          ld_A       <= ’0’;
                          ld_B       <= ’0’;
                          ld_Stop    <= ’0’;
                          ALU_M      <= ”11”;
                          ld_O       <= ’1’;

                        next_STATE<= S_4;

           when S_4 =>    DP_Reset   <= ’0’;
                          A_Ackn_TB  <= ’0’;
                          A_Ready_B  <= ’0’;
                          ld_A       <= ’0’;
                          ld_B       <= ’0’;
                          ld_Stop    <= ’0’;
                          ALU_M      <= ”--”;
                          ld_O       <= ’0’;

                        next_STATE<= S_5;

           when S_5 =>    DP_Reset   <= ’0’;
                          A_Ackn_TB  <= ’0’;
                          A_Ready_B  <= ’1’;
                          ld_A       <= ’0’;
                          ld_B       <= ’0’;
                          ld_Stop    <= ’0’;
                          ALU_M      <= ”--”;
                          ld_O       <= ’0’;

                        if (B_Ackn_A=’1’)
                          then next_STATE<=S_6;
                          else next_STATE<=S_5;
                        end if;
```

```
         when S_6 =>     DP_Reset   <= '0';
                         A_Ackn_TB  <= '0';
                         A_Ready_B  <= '0';
                         ld_A       <= '0';
                         ld_B       <= '0';
                         ld_Stop    <= '0';
                         ALU_M      <= "--";
                         ld_O       <= '0';

                        if (B_Ackn_A='0')
                          then if (StopMsg='1')
                                  then next_STATE<=S_END;
                                  else next_STATE<=S_1;
                               end if;
                          else next_STATE<=S_6;
                        end if;

         when S_END=>   -- nothing (Procedure quits)

       end case;
     end if;

end process;

end FSM_A_dHS_clock_ex_behavioral;
```

C.2.10 Datapath of A (identical B)

VHDL/doubleHS_SMs/Datapath/DP.vhd

```
-- File :          DP.vhd
-- implements :    Datapath (instanced twice: for A and for B)
-- for Example:    FSM,DP (A) --> FSM,DP (B)
--                 using double handshake protocol

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

Entity DP is
   port ( Clock:             in std_logic;
          DP_Reset:          in std_logic;
          ld_A, ld_B, ld_Stop: in std_logic;
          StopMsg:           out std_logic;
          ALU_M:             in std_logic_vector (1 downto 0);
          ld_O:              in std_logic;
          In1, In2:          in std_logic_vector(31 downto 0);
          In_Stop:           in std_logic;
          O_Port:            out std_logic_vector(31 downto 0);
          Out_Stop:          out std_logic);
end DP ;

Architecture DP_schematic of DP is

   -- connection signals
   signal A, B, O: std_logic_vector(31 downto 0);
   signal ALU_Out: std_logic_vector(31 downto 0);
   signal Stop:    std_logic;

   component Reg_32bit
      Port ( Clock:    in std_logic;
             Reset:    in std_logic;
             Load:     in std_logic;
             Data_In:  in std_logic_vector (31 downto 0);
             Data_Out: out std_logic_vector (31 downto 0) );
   end component;

   component Latch_impl
      Port ( Clock:     in std_logic;
             Reset:     in std_logic;
             Load:      in std_logic;
             Data_In:   in std_logic;
             Data_Out: out std_logic );
   end component;

   component ALU_32
      Port ( Mode:       in std_logic_vector (1 downto 0);
             Data_In1:   in std_logic_vector (31 downto 0);
             Data_In2:   in std_logic_vector (31 downto 0);
             Data_Out:   out std_logic_vector (31 downto 0) );
   end component;

begin

Register_A : Reg_32bit
   Port Map ( Clock=>Clock, Reset=>DP_Reset, Load=>ld_A,
              Data_In=>In1, Data_Out=>A);

Register_B : Reg_32bit
   Port Map ( Clock=>Clock, Reset=>DP_Reset, Load=>ld_B,
              Data_In=>In2, Data_Out=>B);

Latch_Stop : Latch_impl
```

```
    Port Map ( Clock=>Clock, Reset=>DP_Reset, Load=>ld_Stop,
               Data_In=>In_Stop, Data_Out=>Stop);

ALU: ALU_32
    Port Map ( Mode=>ALU_M, Data_In1=>A, Data_In2=>B,
               Data_Out=>ALU_Out);

Register_O : Reg_32bit
    Port Map ( Clock=>Clock, Reset=>DP_Reset, Load=>ld_O,
               Data_In=>ALU_Out, Data_Out=>O);

O_Port <= O;
Out_Stop <= Stop;
StopMsg <= Stop;

end DP_schematic;
```

C.2.11 FSM Controlling Datapath B

`VHDL/doubleHS_SMs/FSM_dHS_clock/FSM_and_D_dHS_clock_ex_B.vhd`

```
-- File :          FSM_and_D_dHS_clock_ex_B.vhd
-- implements :    bounding of FSM B and Datapath B
-- of Model:       FSM and separate Datapath (FSM + D)
-- for Example:    FSM,DP (A) --> FSM,DP (B)
--                 using double handshake protocol

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

Entity FSM_and_D_B_dHS_clock_ex is
   port (Reset, Start:          in std_logic;
         Clock_B:              in std_logic;
         A_Ready_B, TB_Ackn_B: in std_logic;
         B_Ackn_A, B_Ready_TB: out std_logic;
         Data_AB:            in std_logic_vector (31 downto 0);
         Data_AB_Stop:           in std_logic;
         Out1_Port:            out std_logic_vector (31 downto 0);
         Out1_Stop_Port:       out std_logic);
end FSM_and_D_B_dHS_clock_ex;


Architecture FSM_and_D_B_dHS_clock_ex_behavioral of FSM_and_D_B_dHS_clock_ex is
-- outputs
signal Out1:  std_logic_vector (31 downto 0);
signal Out1_Stop: std_logic;

signal DP_Reset:     std_logic;
signal ld_A, ld_B, ld_Stop: std_logic;
signal StopMsg: std_logic;
signal ALU_M: std_logic_vector (1 downto 0);
signal ld_O: std_logic;
signal CMP: std_logic_vector(1 downto 0);
signal MUX_sel: std_logic;

Component DP
   port (Clock:            in std_logic;
         DP_Reset:         in std_logic;
         ld_A, ld_B, ld_Stop: in std_logic;
         StopMsg:          out std_logic;
         ALU_M:            in std_logic_vector (1 downto 0);
         ld_O:             in std_logic;
         In1, In2:         in std_logic_vector(31 downto 0);
         In_Stop:          in std_logic;
         O_Port:           out std_logic_vector(31 downto 0);
         Out_Stop:         out std_logic);
end Component;

Component FSM_B_dHS_clock_ex
   port (Reset, Start:          in std_logic;
         Clock_B:               in std_logic;
         A_Ready_B, TB_Ackn_B:  in std_logic;
         B_Ackn_A, B_Ready_TB:  out std_logic;
         DP_Reset:              out std_logic;
         ld_A, ld_B, ld_Stop:   out std_logic;
         StopMsg:               in std_logic;
         ALU_M:                 out std_logic_vector (1 downto 0);
         ld_O:                  out std_logic);
end Component;

begin -- Architecture
DP_B_I: DP
     Port Map (Clock=>Clock_B, DP_Reset=>DP_Reset,
               ld_A=>ld_A, ld_B=>ld_B, ld_Stop=>ld_Stop,
               StopMsg=>StopMsg, ALU_M=>ALU_M, ld_O=>ld_O,
               In1=>Data_AB, In2=>Data_AB,
               In_Stop=>Data_AB_Stop,
               O_Port=>Out1, Out_Stop=>Out1_Stop);

FSM_B_dHS_clock_I: FSM_B_dHS_clock_ex
    Port Map ( Reset=>Reset, Start=>Start,
               Clock_B=>Clock_B,
               A_Ready_B=>A_Ready_B, TB_Ackn_B=>TB_Ackn_B,
               B_Ackn_A=>B_Ackn_A, B_Ready_TB=>B_Ready_TB,
               DP_Reset=>DP_Reset,
               ld_A=>ld_A, ld_B=>ld_B, ld_Stop=>ld_Stop,
               StopMsg=>StopMsg, ALU_M=>ALU_M, ld_O=>ld_O);

-- Entity Outputs
Out1_Port <= Out1;
Out1_Stop_Port <= Out1_Stop;

end FSM_and_D_B_dHS_clock_ex_behavioral;
```

C.2.12 FSM of B

VHDL/doubleHS_SMs/FSM_dHS_clock/FSM_dHS_clock_ex_B.vhd

```vhdl
-- File :          FSM_dHS_clock_ex_B.vhd
-- implements :    FSM B
-- of Model:       FSM and separate Datapath
-- for Example:    FSM,DP (A) --> FSM,DP (B)
--                 using double handshake protocol

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

Entity FSM_B_dHS_clock_ex is
   port (Reset, Start:        in std_logic;
         Clock_B:             in std_logic;
         A_Ready_B, TB_Ackn_B: in std_logic;
         B_Ackn_A, B_Ready_TB: out std_logic;
         DP_Reset:             out std_logic;
         ld_A, ld_B, ld_Stop:  out std_logic;
         StopMsg:              in std_logic;
         ALU_M:                out std_logic_vector (1 downto 0);
         ld_O:                 out std_logic);
end FSM_B_dHS_clock_ex;


Architecture FSM_B_dHS_clock_ex_behavioral of FSM_B_dHS_clock_ex is

type State_Set is (S_BEGIN, S_1, S_2, S_3, S_4, S_5, S_6, S_END);
signal next_STATE, STATE: State_Set;

begin -- Architecture

transition : Process
begin
  wait until (Clock_B'event and Clock_B = '1');
  STATE <= next_STATE;
end process;

behavior1: process(STATE, Start, A_Ready_B, TB_Ackn_B, StopMsg)
begin
    if (Reset = '1') then
         DP_Reset   <= '1';
         B_Ackn_A   <= '0';
         B_Ready_TB <= '0';
         ld_A       <= '0';
         ld_B       <= '0';
         ld_Stop    <= '0';
         ALU_M      <= "--";
         ld_O       <= '0';
      next_STATE <= S_BEGIN;
    else
      case STATE is

        when S_BEGIN => DP_Reset   <= '1';
                        B_Ackn_A   <= '0';
                        B_Ready_TB <= '0';
                        ld_A       <= '0';
                        ld_B       <= '0';
                        ld_Stop    <= '0';
                        ALU_M      <= "--";
                        ld_O       <= '0';

                        if Start = '1'
                          then next_STATE <= S_1;
                          else next_STATE <= S_BEGIN;
                          end if;

          when S_1 =>     DP_Reset   <= '0';
                          B_Ackn_A   <= '0';
                          B_Ready_TB <= '0';
                          ld_A       <= '1';
                          ld_B       <= '1';
                          ld_Stop    <= '1';
                          ALU_M      <= "--";
                          ld_O       <= '0';

                        if (A_Ready_B='1')
                          then next_STATE<= S_2;
                          else next_STATE<= S_1;
                        end if;

          when S_2 =>   DP_Reset   <= '0';
                        B_Ackn_A   <= '1';
                        B_Ready_TB <= '0';
                        ld_A       <= '0';
                        ld_B       <= '0';
                        ld_Stop    <= '0';
                        ALU_M      <= "10";
                        ld_O       <= '0';

                        if (A_Ready_B='0')
                          then next_STATE<= S_3;
                          else next_STATE<= S_2;
                        end if;

          when S_3 =>   DP_Reset   <= '0';
                        B_Ackn_A   <= '0';
                        B_Ready_TB <= '0';
                        ld_A       <= '0';
                        ld_B       <= '0';
                        ld_Stop    <= '0';
                        ALU_M      <= "10";
                        ld_O       <= '1';

                        next_STATE<= S_4;

          when S_4 =>   DP_Reset   <= '0';
                        B_Ackn_A   <= '0';
                        B_Ready_TB <= '0';
                        ld_A       <= '0';
                        ld_B       <= '0';
                        ld_Stop    <= '0';
                        ALU_M      <= "--";
                        ld_O       <= '0';

                        next_STATE<= S_5;

          when S_5 =>   DP_Reset   <= '0';
                        B_Ackn_A   <= '0';
                        B_Ready_TB <= '1';
                        ld_A       <= '0';
                        ld_B       <= '0';
                        ld_Stop    <= '0';
                        ALU_M      <= "--";
                        ld_O       <= '0';

                        if (TB_Ackn_B='1')
                          then next_STATE<=S_6;
                          else next_STATE<=S_5;
                        end if;
```

```
        when S_6 =>   DP_Reset   <= '0';
                      B_Ackn_A   <= '0';
                      B_Ready_TB <= '0';
                      ld_A       <= '0';
                      ld_B       <= '0';
                      ld_Stop    <= '0';
                      ALU_M      <= "--";
                      ld_O       <= '0';

                      if (TB_Ackn_B='0')
                        then if (StopMsg='1')
                               then next_STATE<=S_END;
                               else next_STATE<=S_1;
                             end if;
                        else next_STATE<=S_6;
                      end if;

        when S_END=>  -- nothing (Procedure quits)

      end case;
     end if;

end process;

end FSM_B_dHS_clock_ex_behavioral;
```

C.2.13 Datapath of B

The datapath for the receiver B is identical to the datapath of the sender A. → See datapath for sender A.

C.3 Transfer via Queue with Double Handshake

- A (FSMD without time) → Queue (FSMD without time) → B (FSMD without time)
  - FSMD A
  - FSMD Queue
  - FSMD B
- A (FSMD using clock) → Queue (FSMD using clock) → B (FSMD using clock)
  - FSMD A
  - FSMD Queue
  - FSMD B
- A (FSM Controlling Datapath) → Queue (FSM Controlling Datapath) → B (FSM Controlling Datapath)
  - FSM Controlling Datapath A
    - FSM of A
    - Datapath of A (identical B)
  - FSM Controlling Datapath Queue
    - FSM of Queue
    - Datapath of Queue (Buffered)
      - Datapath of Unbuffered Queue
  - FSM Controlling Datapath B
    - FSM of B
    - Datapath of B identical to A (see A)

C.3.1 A (FSMD without time) → Queue (FSMD without time) → B (FSMD without time)

VHDL/doubleHS_queued_SMs/FSMD_dHS_q_no_time/FSMD_dHS_q_no_time_ex_top.vhd

```
-- File :         FSMD_dHS_q_no_time_ex_top.vhd
-- bounding the three parts of the example
-- for Example: FSMD (A) --> FSMD(Queue) --> FSMD (B)
--                using double handshake protocol

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_signed.ALL;

entity FSMD_dHS_q_no_time_ex_top is
   port (Reset, Start:          in bit;
         TB_Ready_A, TB_Ackn_B: in bit;
         A_Ackn_TB, B_Ready_TB: out bit;
         In1, In2:              in integer;
         In_Stop:               in bit;
         Out1_Port:            out integer;
         Out1_Stop_Port:       out bit);
end FSMD_dHS_q_no_time_ex_top;

architecture FSMD_dHS_q_no_time_ex_top_arch of FSMD_dHS_q_no_time_ex_top is

Component FSMD_A_dHS_q_no_time_ex
   port (Reset, Start:          in bit;
         TB_Ready_A, QC_Ackn_A: in bit;
         A_Ackn_TB, A_Ready_QC: out bit;
         In1, In2:              in integer;
         In_Stop:               in bit;
         Data_AQ_Port:            out integer;
         Data_AQ_Stop_Port:       out bit);
end Component;

Component FSMD_Queue_dHS_no_time_ex
   generic (W, D: integer);
   port (Reset, Start:          in bit;
         A_Ready_QC, B_Rd_Req_QC: in bit;
         QC_Ackn_A, QC_Ready_B: out bit;
         Q_Empty_Port, Q_Full_Port: out bit;
         Data_AQ:              in bit_vector(W-1 downto 0);
         Buffer_QB_Port:             out bit_vector(W-1 downto 0));
end Component;

Component FSMD_B_dHS_q_no_time_ex
   port (Reset, Start:          in bit;
         QC_Ready_B, TB_Ackn_B: in bit;
         B_Rd_Req_QC, B_Ready_TB: out bit;
         Buffer_QB:               in integer;
         Buffer_QB_Stop:          in bit;
         Out1_Port:            out integer;
         Out1_Stop_Port:       out bit);
end Component;

signal A_Ready_QC, B_Rd_Req_QC: bit;
signal QC_Ackn_A, QC_Ready_B: bit;
signal Data_AQ:              integer;
signal Data_AQ_bit_v:        bit_vector(31 downto 0);
signal Data_AQ_Stop:         bit;
signal Buffer_QB:              integer;
signal Buffer_QB_bit_v:        bit_vector(31 downto 0);
signal Buffer_QB_Stop:         bit;

begin
```

```
FSMD_A_dHS_q_no_time_I: FSMD_A_dHS_q_no_time_ex
      port map ( Reset=>Reset, Start=>Start,
           TB_Ready_A=>TB_Ready_A, QC_Ackn_A=>QC_Ackn_A,
           A_Ackn_TB=>A_Ackn_TB, A_Ready_QC=>A_Ready_QC,
           In1=>In1, In2=>In2,
           In_Stop=>In_Stop,
           Data_AQ_Port=>Data_AQ,
           Data_AQ_Stop_Port=>Data_AQ_Stop );

FSMD_Queue_dHS_no_time_I: FSMD_Queue_dHS_no_time_ex
      generic map ( W=>33, D=>4)
      port map ( Reset=>Reset, Start=>Start,
           A_Ready_QC=>A_Ready_QC, B_Rd_Req_QC=>B_Rd_Req_QC,
           QC_Ackn_A=>QC_Ackn_A, QC_Ready_B=>QC_Ready_B,
           Q_Empty_Port=>open, Q_Full_Port=>open,
           Data_AQ(31 downto 0)=>Data_AQ_bit_v,
             Data_AQ(32)=>Data_AQ_Stop,
           Buffer_QB_Port(31 downto 0)=>Buffer_QB_bit_v,
             Buffer_QB_Port(32)=>Buffer_QB_Stop);

FSMD_B_dHS_q_no_time_I: FSMD_B_dHS_q_no_time_ex
      port map ( Reset=>Reset, Start=>Start,
           QC_Ready_B=>QC_Ready_B, TB_Ackn_B=>TB_Ackn_B,
           B_Rd_Req_QC=>B_Rd_Req_QC, B_Ready_TB=>B_Ready_TB,
           Buffer_QB=>Buffer_QB,
           Buffer_QB_Stop=>Buffer_QB_Stop,
           Out1_Port=>Out1_Port,
           Out1_Stop_Port=>Out1_Stop_Port );

Data_AQ_bit_v <= To_BitVector(conv_std_logic_vector( Data_AQ, 32   ));
Buffer_QB        <= conv_integer(To_StdLogicVector(      Buffer_QB_bit_v ));

end FSMD_dHS_q_no_time_ex_top_arch;
```

C.3.2 FSMD A

`VHDL/doubleHS_queued_SMs/FSMD_dHS_q_no_time/FSMD_dHS_q_no_time_ex_A.vhd`

```
-- File :          FSMD_dHS_q_no_time_ex_A.vhd
-- implements :    FSMD A
-- of Model:       FSMD (without using time)
-- for Example:    FSMD (A) --> FSMD(Queue) --> FSMD (B)
--                 using double handshake protocol

Entity FSMD_A_dHS_q_no_time_ex is
   port (Reset, Start:          in bit;
         TB_Ready_A, QC_Ackn_A: in bit;
         A_Ackn_TB, A_Ready_QC: out bit;
         In1, In2:              in integer;
         In_Stop:               in bit;
         Data_AQ_Port:            out integer;
         Data_AQ_Stop_Port:            out bit);
end FSMD_A_dHS_q_no_time_ex;


Architecture FSMD_A_dHS_q_no_time_ex_behavioral of FSMD_A_dHS_q_no_time_ex is

-- behavior outputs
signal Data_AQ:  integer;
signal Data_AQ_Stop: bit;

Procedure behavior1(signal TB_Ready_A, QC_Ackn_A: in bit;
                    signal A_Ackn_TB, A_Ready_QC: out bit;
                    signal In1, In2: in integer;
                    signal In_Stop: in bit;
                    signal Data_AQ: inout integer;
                    signal Data_AQ_Stop: out bit) is
type State_Set is (S_1, S_2, S_3, S_4, S_5, S_6, S_END);
-- S_BEGIN is represented by "Procedure not running".
variable next_STATE: State_Set;
variable A, B: integer;
variable Stop: bit;

begin
next_STATE:= S_1;
Data_AQ <= 0;
Data_AQ_Stop <= '0';
A_Ackn_TB<='0';
A_Ready_QC<='0';

while (next_STATE/=S_END) and (Reset/='1') loop
  case next_STATE is
    when S_1 =>

                   if (TB_Ready_A='1')
                     then next_STATE:= S_2;
                     else next_STATE:= S_1;
                   end if;

    when S_2 =>    A := In1;
                   B := In2;
                   Stop := In_Stop;
                   A_Ackn_TB<='1';

                   if (TB_Ready_A='0')
                     then next_STATE:= S_3;
                     else next_STATE:= S_2;
                   end if;

    when S_3 =>    Data_AQ <= A - B;
                   Data_AQ_Stop <= Stop;
```

```
                  A_Ackn_TB<='0';

                  next_STATE:= S_4;

    when S_4 =>

                  next_STATE:= S_5;

    when S_5 =>   A_Ready_QC<='1';

                  if (QC_Ackn_A='1')
                    then next_STATE:=S_6;
                    else next_STATE:=S_5;
                  end if;

    when S_6 =>   A_Ready_QC<='0';

                  if (QC_Ackn_A='0')
                    then if (Stop='1')
                           then next_STATE:=S_END;
                           else next_STATE:=S_1;
                         end if;
                    else next_STATE:=S_6;
                  end if;

    when S_END=>  -- nothing (Procedure quits)

  end case;
  wait for 1 ns; -- Synopsys needs wait > 0 ns
end loop;

end behavior1;


begin -- Architecture

P1: Process
begin
  wait until (Reset = '1') or (Start = '1');
  if (Reset = '1') then
    Data_AQ <= 0;
    Data_AQ_Stop <= '0';
    A_Ackn_TB<='0';
    A_Ready_QC<='0';
  elsif (Start = '1') then
    behavior1(TB_Ready_A, QC_Ackn_A, A_Ackn_TB, A_Ready_QC,
              In1, In2, In_Stop, Data_AQ, Data_AQ_Stop);
  end if;
end process;

-- Entity Outputs
Data_AQ_Port <= Data_AQ;
Data_AQ_Stop_Port <= Data_AQ_Stop;

end FSMD_A_dHS_q_no_time_ex_behavioral;
```

C.3.3 FSMD Queue

VHDL/doubleHS_queued_SMs/FSMD_dHS_q_no_time/FSMD_dHS_q_no_time_ex_Queue.vhd

```
-- File :          FSMD_dHS_q_no_time_ex_Queue.vhd
-- implements :    Queue
-- of Model:       FSMD (without using time)
-- for Example:  FSMD (A) --> FSMD(Queue) --> FSMD (B)
--                 using double handshake protocol

Entity FSMD_Queue_dHS_no_time_ex is
   generic (W, D: integer);
   port (Reset, Start:        in bit;
         A_Ready_QC, B_Rd_Req_QC: in bit;
         QC_Ackn_A, QC_Ready_B: out bit;
         Q_Empty_Port, Q_Full_Port: out bit;
         Data_AQ:             in bit_vector(W-1 downto 0);
         Buffer_QB_Port:           out bit_vector(W-1 downto 0));
end FSMD_Queue_dHS_no_time_ex;


Architecture FSMD_Queue_dHS_no_time_ex_behavioral of FSMD_Queue_dHS_no_time_ex
is

-- queue outputs
signal Q_Empty, Q_Full: bit;
signal Buffer_QB: bit_vector(W-1 downto 0);

Procedure Queue(signal Reset, Start: in bit;
                signal A_Ready_QC, B_Rd_Req_QC: in bit;
                signal QC_Ackn_A, QC_Ready_B: out bit;
                signal Q_Empty, Q_Full: inout bit;
                signal Data_AQ: in bit_vector(W-1 downto 0);
                signal Buffer_QB: out bit_vector(W-1 downto 0)) is
type State_Set is (S_0, W_1, W_2, R_1, R_2, R_3, S_END);
-- S_BEGIN is represented by "Procedure not running".
variable next_STATE: State_Set;
type fifo_array is ARRAY(D-1 downto 0) of bit_vector(W-1 downto 0);
variable Q: fifo_array;
variable C: integer range -1 to D-1;

begin
next_STATE:= S_0;
QC_Ackn_A <= '0';
QC_Ready_B <= '0';
C:= -1; -- empty;
Q_Empty <= '1';
Q_Full <= '0';

while (next_STATE/=S_END) and (Reset/='1') loop
  case next_STATE is
    when S_0 => QC_Ackn_A <= '0';
                QC_Ready_B <= '0';

                if (Q_Empty='0' and B_Rd_Req_QC='1')
                  then next_STATE:= R_1;
                elsif (Q_Full='0' and A_Ready_QC='1')
                  then next_STATE:= W_1;
                end if;

    when W_1 => C:= C+1;

                next_STATE:= W_2;

    when W_2 => for i in D-1 downto 1 loop
                  Q(i):= Q(i-1);
                end loop;
                Q(0):= Data_AQ;
                if (C=-1)
                  then Q_Empty <= '1';
                else
                  Q_Empty <= '0';
                end if;
                if (C=D-1)
                  then Q_Full <= '1';
                else
                  Q_Full <= '0';
                end if;
                QC_Ackn_A <= '1';

                if (A_Ready_QC='0')
                  then next_STATE:= S_0;
                else
                  next_STATE:= W_2;
                end if;

    when R_1 => -- needed for conversion into "FSM plus Datapath"

                next_STATE:= R_2;

    when R_2 => Buffer_QB <= Q(C);
                C:= C-1;

                next_STATE:= R_3;

    when R_3 => QC_Ready_B <= '1';
                if (C=-1)
                  then Q_Empty <= '1';
                else
                  Q_Empty <= '0';
                end if;
                if (C=D-1)
                  then Q_Full <= '1';
                else
                  Q_Full <= '0';
                end if;

                if (B_Rd_Req_QC='0')
                  then next_STATE:= S_0;
                else
                  next_STATE:= R_3;
                end if;

    when S_END=>  -- nothing (Procedure quits)
                  -- (will never happen though)

  end case;
  wait for 1 ns; -- Synopsys needs wait > 0 ns
end loop;

end Queue;


begin -- Architecture

Q1: Process
begin
  wait until (Reset = '1') or (Start = '1');
  if (Reset = '1') then
    QC_Ackn_A <= '0';
    QC_Ready_B <= '0';
    Q_Empty <= '0';
    Q_Full <= '0';
    elsif (Start = '1') then
      Queue(Reset, Start,
            A_Ready_QC, B_Rd_Req_QC,
            QC_Ackn_A, QC_Ready_B,
            Q_Empty, Q_Full,
            Data_AQ, Buffer_QB);
    end if;
  end process;

  -- Entity Outputs
  Q_Empty_Port <= Q_Empty;
  Q_Full_Port <= Q_Full;
  Buffer_QB_Port <= Buffer_QB;

  end FSMD_Queue_dHS_no_time_ex_behavioral;
```

C.3.4 FSMD B

VHDL/doubleHS_queued_SMs/FSMD_dHS_q_no_time/FSMD_dHS_q_no_time_ex_B.vhd

```
-- File :          FSMD_dHS_q_no_time_ex_B.vhd
-- implements :    FSMD B
-- of Model:       FSMD (without using time)
-- for Example:    FSMD (A) --> FSMD(Queue) --> FSMD (B)
--                 using double handshake protocol

Entity FSMD_B_dHS_q_no_time_ex is
   port (Reset, Start:          in bit;
         QC_Ready_B, TB_Ackn_B: in bit;
         B_Rd_Req_QC, B_Ready_TB: out bit;
         Buffer_QB:              in integer;
         Buffer_QB_Stop:         in bit;
         Out1_Port:            out integer;
         Out1_Stop_Port:       out bit);
end FSMD_B_dHS_q_no_time_ex;


Architecture FSMD_B_dHS_q_no_time_ex_behavioral of FSMD_B_dHS_q_no_time_ex is

-- behavior outputs
signal Out1:  integer;
signal Out1_Stop: bit;

Procedure behavior1(signal QC_Ready_B, TB_Ackn_B: in bit;
                    signal B_Rd_Req_QC, B_Ready_TB: out bit;
                    signal Buffer_QB: in integer;
                    signal Buffer_QB_Stop: in bit;
                    signal Out1: inout integer;
                    signal Out1_Stop: out bit) is
type State_Set is (S_1, S_2, S_3, S_4, S_5, S_6, S_END);
-- S_BEGIN is represented by "Procedure not running".
variable next_STATE: State_Set;
variable C: integer;
variable Stop: bit;

begin
next_STATE:= S_1;
Out1 <= 0;
Out1_Stop <= '0';
B_Rd_Req_QC<='0';
B_Ready_TB<='0';

while (next_STATE/=S_END) and (Reset/='1') loop
  case next_STATE is
    when S_1 =>   B_Rd_Req_QC <= '1';

                  if (QC_Ready_B='1')
                    then next_STATE:= S_2;
                    else next_STATE:= S_1;
                  end if;

    when S_2 =>   C := Buffer_QB;
                  Stop := Buffer_QB_Stop;
                  B_Rd_Req_QC<='0';

                  if (QC_Ready_B='0')
                    then next_STATE:= S_3;
                    else next_STATE:= S_2;
                  end if;

    when S_3 =>   Out1 <= 2 * C;
                  Out1_Stop <= Stop;
```

```
                    next_STATE:= S_4;

    when S_4 =>

                    next_STATE:= S_5;

    when S_5 =>     B_Ready_TB<='1';

                    if (TB_Ackn_B='1')
                      then next_STATE:=S_6;
                      else next_STATE:=S_5;
                    end if;

    when S_6 =>     B_Ready_TB<='0';

                    if (TB_Ackn_B='0')
                      then if (Stop='1')
                             then next_STATE:=S_END;
                             else next_STATE:=S_1;
                           end if;
                      else next_STATE:=S_6;
                    end if;

    when S_END=>    -- nothing (Procedure quits)

  end case;
  wait for 1 ns; -- Synopsys needs wait > 0 ns
end loop;

end behavior1;


begin -- Architecture

P1: Process
begin
  wait until (Reset = '1') or (Start = '1');
  if (Reset = '1') then
    Out1 <= 0;
    Out1_Stop <= '0';
    B_Rd_Req_QC<='0';
    B_Ready_TB<='0';
  elsif (Start = '1') then
    behavior1(QC_Ready_B, TB_Ackn_B, B_Rd_Req_QC, B_Ready_TB,
              Buffer_QB, Buffer_QB_Stop, Out1, Out1_Stop);
  end if;
end process;

-- Entity Outputs
Out1_Port <= Out1;
Out1_Stop_Port <= Out1_Stop;

end FSMD_B_dHS_q_no_time_ex_behavioral;
```

C.3.5 A (FSMD using clock) → Queue (FSMD using clock) → B (FSMD using clock)

VHDL/doubleHS_queued_SMs/FSMD_dHS_q_clock/FSMD_dHS_q_clock_ex_top.vhd

```
-- File :        FSMD_dHS_q_clock_ex_top.vhd
-- bounding the three parts of the example
-- for Example: FSMD (A) --> FSMD(Queue) --> FSMD (B)
--               using double handshake protocol

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use STD.textio.all;

entity FSMD_dHS_q_clock_ex_top is
   port (Reset, Start:        in std_logic;
         Clock_A, Clock_B, Clock_Q:   in std_logic;
         TB_Ready_A, TB_Ackn_B: in std_logic;
         A_Ackn_TB, B_Ready_TB: out std_logic;
         In1, In2:            in std_logic_vector(31 downto 0);
         In_Stop:             in std_logic;
         Out1_Port:          out std_logic_vector(31 downto 0);
         Out1_Stop_Port:     out std_logic);
end FSMD_dHS_q_clock_ex_top;

architecture FSMD_dHS_q_clock_ex_top_arch of FSMD_dHS_q_clock_ex_top is

Component FSMD_A_dHS_q_clock_ex
   port (Reset, Start:        in std_logic;
         Clock_A:            in std_logic;
         TB_Ready_A, QC_Ackn_A: in std_logic;
         A_Ackn_TB, A_Ready_QC: out std_logic;
         In1, In2:            in std_logic_vector(31 downto 0);
         In_Stop:             in std_logic;
         Data_AQ_Port:          out std_logic_vector(31 downto 0);
         Data_AQ_Stop_Port:     out std_logic);
end Component;

Component FSMD_Queue_dHS_clock_ex
   generic (W, D: integer);
   port (Reset, Start: in std_logic;
         Clock_Q:            in std_logic;
         A_Ready_QC, B_Rd_Req_QC: in std_logic;
         QC_Ackn_A, QC_Ready_B: out std_logic;
         Q_Empty_Port, Q_Full_Port: out std_logic;
         Data_AQ:         in std_logic_vector(W-1 downto 0);
         Buffer_QB_Port: out std_logic_vector(W-1 downto 0));
end Component;

Component FSMD_B_dHS_q_clock_ex
   port (Reset, Start:        in std_logic;
         Clock_B:            in std_logic;
         QC_Ready_B, TB_Ackn_B: in std_logic;
         B_Rd_Req_QC, B_Ready_TB: out std_logic;
         Buffer_QB:           in std_logic_vector(31 downto 0);
         Buffer_QB_Stop:           in std_logic;
         Out1_Port:          out std_logic_vector(31 downto 0);
         Out1_Stop_Port:     out std_logic);
end Component;

signal A_Ready_QC, B_Rd_Req_QC: std_logic;
signal QC_Ackn_A, QC_Ready_B: std_logic;
signal Data_AQ: std_logic_vector(31 downto 0);
signal Data_AQ_Stop: std_logic;
signal Buffer_QB: std_logic_vector(31 downto 0);
signal Buffer_QB_Stop: std_logic;
```

```
begin

FSMD_A_dHS_q_clock_I: FSMD_A_dHS_q_clock_ex
     port map ( Reset=>Reset, Start=>Start, Clock_A=>Clock_A,
         TB_Ready_A=>TB_Ready_A, QC_Ackn_A=>QC_Ackn_A,
         A_Ackn_TB=>A_Ackn_TB, A_Ready_QC=>A_Ready_QC,
         In1=>In1, In2=>In2,
         In_Stop=>In_Stop,
         Data_AQ_Port=>Data_AQ,
         Data_AQ_Stop_Port=>Data_AQ_Stop );

FSMD_Queue_dHS_clock_I: FSMD_Queue_dHS_clock_ex
     generic map (W=>33, D=>4)
     port map ( Reset=>Reset, Start=>Start, Clock_Q=>Clock_Q,
         A_Ready_QC=>A_Ready_QC, B_Rd_Req_QC=>B_Rd_Req_QC,
         QC_Ackn_A=>QC_Ackn_A, QC_Ready_B=>QC_Ready_B,
         Q_Empty_Port=>open, Q_Full_Port=>open,
         Data_AQ(31 downto 0)=>Data_AQ,
           Data_AQ(32)=>Data_AQ_Stop,
         Buffer_QB_Port(31 downto 0)=>Buffer_QB,
           Buffer_QB_Port(32)=>Buffer_QB_Stop);

FSMD_B_dHS_q_clock_I: FSMD_B_dHS_q_clock_ex
     port map ( Reset=>Reset, Start=>Start, Clock_B=>Clock_B,
         QC_Ready_B=>QC_Ready_B, TB_Ackn_B=>TB_Ackn_B,
         B_Rd_Req_QC=>B_Rd_Req_QC, B_Ready_TB=>B_Ready_TB,
         Buffer_QB=>Buffer_QB,
         Buffer_QB_Stop=>Buffer_QB_Stop,
         Out1_Port=>Out1_Port,
         Out1_Stop_Port=>Out1_Stop_Port );

end FSMD_dHS_q_clock_ex_top_arch;
```

C.3.6 FSMD A

VHDL/doubleHS_queued_SMs/FSMD_dHS_q_clock/FSMD_dHS_q_clock_ex_A.vhd

```
-- File :          FSMD_dHS_q_clock_ex_A.vhd
-- implements :   FSMD A
-- of Model:      FSMD (using time)
-- for Example:  FSMD (A) --> FSMD(Queue) --> FSMD (B)
--                using double handshake protocol

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_signed.all;

Entity FSMD_A_dHS_q_clock_ex is
   port (Reset, Start:         in std_logic;
         Clock_A:             in std_logic;
         TB_Ready_A, QC_Ackn_A: in std_logic;
         A_Ackn_TB, A_Ready_QC: out std_logic;
         In1, In2:             in std_logic_vector(31 downto 0);
         In_Stop:              in std_logic;
         Data_AQ_Port:          out std_logic_vector(31 downto 0);
         Data_AQ_Stop_Port:          out std_logic);
end FSMD_A_dHS_q_clock_ex;


Architecture FSMD_A_dHS_q_clock_ex_behavioral of FSMD_A_dHS_q_clock_ex is
-- outputs
signal Data_AQ:  std_logic_vector(31 downto 0);
signal Data_AQ_Stop: std_logic;

-- internal signals
type State_Set is (S_BEGIN, S_1, S_2, S_3, S_4, S_5, S_6, S_END);
signal next_STATE: State_Set;
signal A, B: std_logic_vector(31 downto 0);
signal Stop: std_logic;

begin -- Architecture

behavior1: process(Clock_A)
begin
  if (Clock_A'event and Clock_A='1') then
    if (Reset = '1') then
      Data_AQ <= (others=>'0');
      Data_AQ_Stop <= '0';
      A_Ackn_TB<='0';
      A_Ready_QC<='0';
      next_STATE <= S_BEGIN;
    else
      case next_STATE is

        when S_BEGIN => Data_AQ <= (others=>'0');
                        Data_AQ_Stop <= '0';
                        A_Ackn_TB<='0';
                        A_Ready_QC<='0';

                        if Start = '1'
                          then next_STATE <= S_1;
                          else next_STATE <= S_BEGIN;
                        end if;

        when S_1 =>

                       if (TB_Ready_A='1')
                         then next_STATE<= S_2;
                         else next_STATE<= S_1;
                         end if;

           when S_2 =>   A <= In1;
                         B <= In2;
                         Stop <= In_Stop;
                         A_Ackn_TB<='1';

                         if (TB_Ready_A='0')
                           then next_STATE<= S_3;
                           else next_STATE<= S_2;
                         end if;

           when S_3 =>   Data_AQ <= A - B;
                         Data_AQ_Stop <= Stop;
                         A_Ackn_TB<='0';

                         next_STATE<= S_4;

           when S_4 =>

                         next_STATE<= S_5;

           when S_5 =>   A_Ready_QC<='1';

                         if (QC_Ackn_A='1')
                           then next_STATE<=S_6;
                           else next_STATE<=S_5;
                         end if;

           when S_6 =>   A_Ready_QC<='0';

                         if (QC_Ackn_A='0')
                           then if (Stop='1')
                                  then next_STATE<=S_END;
                                  else next_STATE<=S_1;
                                end if;
                           else next_STATE<=S_6;
                         end if;

           when S_END=>  -- nothing (Procedure quits)

         end case;
       end if;
     end if;

  end process;

  -- Entity Outputs
  Data_AQ_Port <= Data_AQ;
  Data_AQ_Stop_Port <= Data_AQ_Stop;

  end FSMD_A_dHS_q_clock_ex_behavioral;
```

C.3.7 FSMD Queue

VHDL/doubleHS_queued_SMs/FSMD_dHS_q_clock/FSMD_dHS_q_clock_ex_Queue.vhd

```
-- File :          FSMD_dHS_q_clock_ex_Queue.vhd
-- implements :    Queue
-- of Model:       FSMD (using clock)
-- for Example:  FSMD (A) --> FSMD(Queue) --> FSMD (B)
--                 using double handshake protocol

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

Entity FSMD_Queue_dHS_clock_ex is
   -- pragma template
   generic (W, D: integer);
   port (Reset, Start:         in std_logic;
         Clock_Q:         in std_logic;
         A_Ready_QC, B_Rd_Req_QC: in std_logic;
         QC_Ackn_A, QC_Ready_B: out std_logic;
         Q_Empty_Port, Q_Full_Port: out std_logic;
         Data_AQ:              in std_logic_vector(W-1 downto 0);
         Buffer_QB_Port:            out std_logic_vector(W-1 downto 0));
end FSMD_Queue_dHS_clock_ex;


Architecture FSMD_Queue_dHS_clock_ex_behavioral of FSMD_Queue_dHS_clock_ex
is

-- queue outputs
signal Q_Empty, Q_Full: std_logic;
signal Buffer_QB: std_logic_vector(W-1 downto 0);
type State_Set is (S_0, W_1, W_2, R_1, R_2, R_3, S_END);
-- S_BEGIN is represented by "Procedure not running".
signal next_STATE: State_Set;
type fifo_array is ARRAY(D-1 downto 0) of std_logic_vector(W-1 downto 0);
signal Q: fifo_array;
signal C: integer range -1 to D-1;

begin

Queue: Process(Clock_Q)
begin
  if (Clock_Q'event and Clock_Q='1') then
    if (Reset = '1') then
      next_STATE<= S_0;
      QC_Ackn_A <= '0';
      QC_Ready_B <= '0';
      C<= -1; -- empty;
      Q_Empty <= '1';
      Q_Full <= '0';
    else
      case next_STATE is
        when S_0 => QC_Ackn_A <= '0';
                    QC_Ready_B <= '0';

                    if (Q_Empty='0' and B_Rd_Req_QC='1')
                      then next_STATE<= R_1;
                    elsif (Q_Full='0' and A_Ready_QC='1')
                      then next_STATE<= W_1;
                    end if;

        when W_1 => C<= C+1;
                    for i in D-1 downto 1 loop
                      Q(i)<= Q(i-1);
                    end loop;
                    next_STATE<= W_2;

        when W_2 => Q(0)<= Data_AQ;
                    if (C=-1)
                      then Q_Empty <= '1';
                    else
                      Q_Empty <= '0';
                    end if;
                    if (C=D-1)
                      then Q_Full <= '1';
                    else
                      Q_Full <= '0';
                    end if;
                    QC_Ackn_A <= '1';

                    if (A_Ready_QC='0')
                      then next_STATE<= S_0;
                    else
                      next_STATE<= W_2;
                    end if;

        when R_1 => -- needed for conversion into "FSM plus Datapath"

                    next_STATE<= R_2;

        when R_2 => Buffer_QB <= Q(C);
                    C<= C-1;

                    next_STATE<= R_3;

        when R_3 => QC_Ready_B <= '1';
                    if (C=-1)
                      then Q_Empty <= '1';
                    else
                      Q_Empty <= '0';
                    end if;
                    if (C=D-1)
                      then Q_Full <= '1';
                    else
                      Q_Full <= '0';
                    end if;

                    if (B_Rd_Req_QC='0')
                      then next_STATE<= S_0;
                    else
                      next_STATE<= R_3;
                    end if;

        when S_END=>  -- nothing (Procedure quits)
                      -- (will never happen though)

      end case;
    end if;
  end if;
end process;

Buffer_QB_Port <= Buffer_QB;

end FSMD_Queue_dHS_clock_ex_behavioral;
```

C.3.8 FSMD B

VHDL/doubleHS_queued_SMs/FSMD_dHS_q_clock/FSMD_dHS_q_clock_ex_B.vhd

```
-- File :          FSMD_dHS_q_clock_ex_B.vhd
-- implements :    FSMD B
-- of Model:       FSMD (using clock)
-- for Example:    FSMD (A) --> FSMD(Queue) --> FSMD (B)
--                 using double handshake protocol

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_signed.all;

-- FSMD B of example of two FSMD (with clock) exchanging data over a queue

Entity FSMD_B_dHS_q_clock_ex is
   port (Reset, Start:        in std_logic;
         Clock_B:            in std_logic;
         QC_Ready_B, TB_Ackn_B: in std_logic;
         B_Rd_Req_QC, B_Ready_TB: out std_logic;
         Buffer_QB:           in std_logic_vector(31 downto 0);
         Buffer_QB_Stop:            in std_logic;
         Out1_Port:          out std_logic_vector(31 downto 0);
         Out1_Stop_Port:     out std_logic);
end FSMD_B_dHS_q_clock_ex;


Architecture FSMD_B_dHS_q_clock_ex_behavioral of FSMD_B_dHS_q_clock_ex is
-- outputs
signal Out1:  std_logic_vector(31 downto 0);
signal Out1_Stop: std_logic;

-- internal signals
type State_Set is (S_BEGIN, S_1, S_2, S_3, S_4, S_5, S_6, S_END);
-- S_BEGIN is represented by "Procedure not running".
signal next_STATE: State_Set;
signal C: std_logic_vector(31 downto 0);
signal Stop: std_logic;

begin -- Architecture

behavior1: process(Clock_B)
begin
  if (Clock_B'event and Clock_B='1') then
    if (Reset = '1') then
      Out1 <= (others=>'0');
      Out1_Stop <='0';
      B_Rd_Req_QC<='0';
      B_Ready_TB<='0';
      next_STATE <= S_BEGIN;
    else
      case next_STATE is

        when S_BEGIN => Out1 <= (others=>'0');
                        Out1_Stop <='0';
                        B_Rd_Req_QC<='0';
                        B_Ready_TB<='0';

                        if Start = '1'
                          then next_STATE <= S_1;
                          else next_STATE <= S_BEGIN;
                        end if;

        when S_1 =>   B_Rd_Req_QC <='1';
```

```
                      if (QC_Ready_B='1')
                        then next_STATE<= S_2;
                        else next_STATE<= S_1;
                      end if;

        when S_2 =>   C <= Buffer_QB;
                      Stop <= Buffer_QB_Stop;
                      B_Rd_Req_QC<='0';

                      if (QC_Ready_B='0')
                        then next_STATE<= S_3;
                        else next_STATE<= S_2;
                      end if;

        when S_3 =>   Out1 <= C(28 downto 0) * "010";
                      Out1_Stop <= Stop;

                      next_STATE<= S_4;

        when S_4 =>

                      next_STATE<= S_5;

        when S_5 =>   B_Ready_TB<='1';

                      if (TB_Ackn_B='1')
                        then next_STATE<=S_6;
                        else next_STATE<=S_5;
                      end if;

        when S_6 =>   B_Ready_TB<='0';

                      if (TB_Ackn_B='0')
                        then if (Stop='1')
                               then next_STATE<=S_END;
                               else next_STATE<=S_1;
                             end if;
                        else next_STATE<=S_6;
                      end if;

        when S_END=>  -- nothing (Procedure quits)

      end case;
     end if;
   end if;

end process;


-- Entity Outputs
Out1_Port <= Out1;
Out1_Stop_Port <= Out1_Stop;

end FSMD_B_dHS_q_clock_ex_behavioral;
```

C.3.9 A (FSM Controlling Datapath) → Queue (FSM Controlling Datapath) → B (FSM Controlling Datapath)

VHDL/doubleHS_queued_SMs/FSM_dHS_q_clock/FSM_and_D_dHS_q_clock_ex_top.vhd

```
-- File :           FSM_and_D_dHS_q_clock_ex_top.vhd
-- bounding the three parts of the example
-- for Example: FSM,DP (A) --> FSM,DP(Queue) --> FSM,DP (B)
--                  using double handshake protocol

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity FSM_and_D_dHS_q_clock_ex_top is
   port (Reset, Start:        in std_logic;
         Clock_A, Clock_B, Clock_Q:   in std_logic;
         TB_Ready_A, TB_Ackn_B: in std_logic;
         A_Ackn_TB, B_Ready_TB: out std_logic;
         In1, In2:            in std_logic_vector(31 downto 0);
         In_Stop:             in std_logic;
         Out1_Port:          out std_logic_vector(31 downto 0);
         Out1_Stop_Port:     out std_logic);
end FSM_and_D_dHS_q_clock_ex_top;

architecture FSM_and_D_dHS_q_clock_ex_top_arch of FSM_and_D_dHS_q_clock_ex_top is

Component FSM_and_D_A_dHS_q_clock_ex
   port (Reset, Start:        in std_logic;
         Clock_A:            in std_logic;
         TB_Ready_A, QC_Ackn_A: in std_logic;
         A_Ackn_TB, A_Ready_QC: out std_logic;
         In1, In2:            in std_logic_vector(31 downto 0);
         In_Stop:             in std_logic;
         Data_AQ_Port:          out std_logic_vector(31 downto 0);
         Data_AQ_Stop_Port:     out std_logic);
end Component;

Component FSM_and_D_Queue_dHS_clock_ex
   generic (Q_W, Q_D, S_W: integer);
   port (Reset, Start: in std_logic;
         Clock_Q:            in std_logic;
         A_Ready_QC, B_Rd_Req_QC: in std_logic;
         QC_Ackn_A, QC_Ready_B: out std_logic;
         Q_Empty_Port, Q_Full_Port: out std_logic;
         Data_AQ:        in std_logic_vector(Q_W-1 downto 0);
         Buffer_QB_Port: out std_logic_vector(Q_W-1 downto 0));
end Component;

Component FSM_and_D_B_dHS_q_clock_ex
   port (Reset, Start:        in std_logic;
         Clock_B:            in std_logic;
         QC_Ready_B, TB_Ackn_B: in std_logic;
         B_Rd_Req_QC, B_Ready_TB: out std_logic;
         Buffer_QB:           in std_logic_vector(31 downto 0);
         Buffer_QB_Stop:           in std_logic;
         Out1_Port:          out std_logic_vector(31 downto 0);
         Out1_Stop_Port:     out std_logic);
end Component;

signal A_Ready_QC, B_Rd_Req_QC: std_logic;
signal QC_Ackn_A, QC_Ready_B: std_logic;
signal Data_AQ: std_logic_vector(31 downto 0);
signal Data_AQ_Stop: std_logic;
signal Data_AQ_w_Stop: std_logic_vector(32 downto 0);
signal Buffer_QB: std_logic_vector(31 downto 0);
signal Buffer_QB_Stop: std_logic;
signal Buffer_QB_w_Stop: std_logic_vector(32 downto 0);

begin

FSM_and_D_A_dHS_q_clock_I: FSM_and_D_A_dHS_q_clock_ex
      port map ( Reset=>Reset, Start=>Start, Clock_A=>Clock_A,
           TB_Ready_A=>TB_Ready_A, QC_Ackn_A=>QC_Ackn_A,
           A_Ackn_TB=>A_Ackn_TB, A_Ready_QC=>A_Ready_QC,
           In1=>In1, In2=>In2,
           In_Stop=>In_Stop,
           Data_AQ_Port=>Data_AQ,
           Data_AQ_Stop_Port=>Data_AQ_Stop );

FSM_and_D_Queue_dHS_clock_I: FSM_and_D_Queue_dHS_clock_ex
      generic map (Q_W=>33, Q_D=>4, S_W=>2)
      port map ( Reset=>Reset, Start=>Start, Clock_Q=>Clock_Q,
           A_Ready_QC=>A_Ready_QC, B_Rd_Req_QC=>B_Rd_Req_QC,
           QC_Ackn_A=>QC_Ackn_A, QC_Ready_B=>QC_Ready_B,
           Q_Empty_Port=>open, Q_Full_Port=>open,
           Data_AQ=>Data_AQ_w_Stop,
           Buffer_QB_Port=>Buffer_QB_w_Stop);

Data_AQ_w_Stop <= Data_AQ_Stop & Data_AQ;
Buffer_QB <= Buffer_QB_w_Stop(31 downto 0);
Buffer_QB_Stop <= Buffer_QB_w_Stop(32);

FSM_and_D_B_dHS_q_clock_I: FSM_and_D_B_dHS_q_clock_ex
      port map ( Reset=>Reset, Start=>Start, Clock_B=>Clock_B,
           QC_Ready_B=>QC_Ready_B, TB_Ackn_B=>TB_Ackn_B,
           B_Rd_Req_QC=>B_Rd_Req_QC, B_Ready_TB=>B_Ready_TB,
           Buffer_QB=>Buffer_QB,
           Buffer_QB_Stop=>Buffer_QB_Stop,
           Out1_Port=>Out1_Port,
           Out1_Stop_Port=>Out1_Stop_Port );

end FSM_and_D_dHS_q_clock_ex_top_arch;
```

C.3.10 FSM Controlling Datapath A

VHDL/doubleHS_queued_SMs/FSM_dHS_q_clock/FSM_and_D_dHS_q_clock_ex_A.vhd

```vhdl
-- File :           FSM_and_D_dHS_q_clock_ex_A.vhd
-- implements :     bounding of FSM A and Datapath A
-- of Model:        FSM and separate Datapath
-- for Example:     FSM,DP (A) --> FSM,DP(Queue) --> FSM,DP (B)
--                  using double handshake protocol

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

Entity FSM_and_D_A_dHS_q_clock_ex is
   port (Reset, Start:          in std_logic;
         Clock_A:              in std_logic;
         TB_Ready_A, QC_Ackn_A: in std_logic;
         A_Ackn_TB, A_Ready_QC: out std_logic;
         In1, In2:              in std_logic_vector (31 downto 0);
         In_Stop:               in std_logic;
         Data_AQ_Port:            out std_logic_vector (31 downto 0);
         Data_AQ_Stop_Port:            out std_logic);
end FSM_and_D_A_dHS_q_clock_ex;


Architecture FSM_and_D_A_dHS_q_clock_ex_structural of FSM_and_D_A_dHS_q_clock_ex is
-- outputs
signal Data_AQ:  std_logic_vector (31 downto 0);
signal Data_AQ_Stop: std_logic;

signal DP_Reset:     std_logic;
signal ld_A, ld_B, ld_Stop: std_logic;
signal StopMsg: std_logic;
signal ALU_M: std_logic_vector (1 downto 0);
signal ld_O: std_logic;
signal CMP: std_logic_vector(1 downto 0);
signal MUX_sel: std_logic;

Component DP
   port (Clock:           in std_logic;
         DP_Reset:        in std_logic;
         ld_A, ld_B, ld_Stop: in std_logic;
         StopMsg:         out std_logic;
         ALU_M:           in std_logic_vector (1 downto 0);
         ld_O:            in std_logic;
         In1, In2:        in std_logic_vector(31 downto 0);
         In_Stop:         in std_logic;
         O_Port:          out std_logic_vector(31 downto 0);
         Out_Stop:        out std_logic);
end Component;

Component FSM_A_dHS_q_clock_ex
   port (Reset, Start:          in std_logic;
         Clock_A:               in std_logic;
         TB_Ready_A, QC_Ackn_A: in std_logic;
         A_Ackn_TB, A_Ready_QC: out std_logic;
         DP_Reset:              out std_logic;
         ld_A, ld_B, ld_Stop:   out std_logic;
         StopMsg:               in std_logic;
         ALU_M:                 out std_logic_vector (1 downto 0);
         ld_O:                  out std_logic);
end Component;

begin -- Architecture
DP_A_I: DP
     Port Map (Clock=>Clock_A, DP_Reset=>DP_Reset,
                 ld_A=>ld_A, ld_B=>ld_B, ld_Stop=>ld_Stop,
                 StopMsg=>StopMsg, ALU_M=>ALU_M, ld_O=>ld_O,
                 In1=>In1, In2=>In2,
                 In_Stop=>In_Stop,
                 O_Port=>Data_AQ, Out_Stop=>Data_AQ_Stop);

FSM_A_dHS_q_clock_I: FSM_A_dHS_q_clock_ex
      Port Map ( Reset=>Reset, Start=>Start,
                 Clock_A=>Clock_A,
                 TB_Ready_A=>TB_Ready_A, QC_Ackn_A=>QC_Ackn_A,
                 A_Ackn_TB=>A_Ackn_TB, A_Ready_QC=>A_Ready_QC,
                 DP_Reset=>DP_Reset,
                 ld_A=>ld_A, ld_B=>ld_B, ld_Stop=>ld_Stop,
                 StopMsg=>StopMsg, ALU_M=>ALU_M, ld_O=>ld_O);

-- Entity Outputs
Data_AQ_Port <= Data_AQ;
Data_AQ_Stop_Port <= Data_AQ_Stop;

end FSM_and_D_A_dHS_q_clock_ex_structural;
```

C.3.11 FSM of A

VHDL/doubleHS_queued_SMs/FSM_dHS_q_clock/FSM_dHS_q_clock_ex_A.vhd

```
-- File :          FSM_dHS_q_clock_ex_A.vhd
-- implements :   FSM A
-- of Model:      FSM and separate Datapath
-- for Example:  FSM,DP (A) --> FSM,DP(Queue) --> FSM,DP (B)
--                using double handshake protocol

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

Entity FSM_A_dHS_q_clock_ex is
   port (Reset, Start:         in std_logic;
         Clock_A:                in std_logic;
         TB_Ready_A, QC_Ackn_A: in std_logic;
         A_Ackn_TB, A_Ready_QC: out std_logic;
         DP_Reset:               out std_logic;
         ld_A, ld_B, ld_Stop:   out std_logic;
         StopMsg:                in std_logic;
         ALU_M:                  out std_logic_vector (1 downto 0);
         ld_O:                   out std_logic);
end FSM_A_dHS_q_clock_ex;


Architecture FSM_A_dHS_q_clock_ex_behavioral of FSM_A_dHS_q_clock_ex is

type State_Set is (S_BEGIN, S_1, S_2, S_3, S_4, S_5, S_6, S_END);
signal next_STATE, STATE: State_Set;

begin -- Architecture

transition: Process
begin
  wait until (Clock_A'event and Clock_A = '1');
  STATE <= next_STATE;
end process;

behavior1: process(STATE, Start, TB_Ready_A, QC_Ackn_A, StopMsg)
begin
    if (Reset = '1') then
         DP_Reset   <= '1';
         A_Ackn_TB  <= '0';
         A_Ready_QC <= '0';
         ld_A       <= '0';
         ld_B       <= '0';
         ld_Stop    <= '0';
         ALU_M      <= "--";
         ld_O       <= '0';
      next_STATE <= S_BEGIN;
    else
      case STATE is

        when S_BEGIN => DP_Reset   <= '1';
                        A_Ackn_TB  <= '0';
                        A_Ready_QC <= '0';
                        ld_A       <= '0';
                        ld_B       <= '0';
                        ld_Stop    <= '0';
                        ALU_M      <= "--";
                        ld_O       <= '0';

                       if Start = '1'
                         then next_STATE <= S_1;
                         else next_STATE <= S_BEGIN;
```

```
                    end if;

          when S_1 =>    DP_Reset   <= '0';
                         A_Ackn_TB  <= '0';
                         A_Ready_QC <= '0';
                         ld_A       <= '0';
                         ld_B       <= '0';
                         ld_Stop    <= '0';
                         ALU_M      <= "--";
                         ld_O       <= '0';

                    if (TB_Ready_A='1')
                      then next_STATE<= S_2;
                      else next_STATE<= S_1;
                    end if;

          when S_2 =>    DP_Reset   <= '0';
                         A_Ackn_TB  <= '1';
                         A_Ready_QC <= '0';
                         ld_A       <= '1';
                         ld_B       <= '1';
                         ld_Stop    <= '1';
                         ALU_M      <= "11";
                         ld_O       <= '0';

                    if (TB_Ready_A='0')
                      then next_STATE<= S_3;
                      else next_STATE<= S_2;
                    end if;

          when S_3 =>    DP_Reset   <= '0';
                         A_Ackn_TB  <= '0';
                         A_Ready_QC <= '0';
                         ld_A       <= '0';
                         ld_B       <= '0';
                         ld_Stop    <= '0';
                         ALU_M      <= "11";
                         ld_O       <= '1';

                    next_STATE<= S_4;

          when S_4 =>    DP_Reset   <= '0';
                         A_Ackn_TB  <= '0';
                         A_Ready_QC <= '0';
                         ld_A       <= '0';
                         ld_B       <= '0';
                         ld_Stop    <= '0';
                         ALU_M      <= "--";
                         ld_O       <= '0';

                    next_STATE<= S_5;

          when S_5 =>    DP_Reset   <= '0';
                         A_Ackn_TB  <= '0';
                         A_Ready_QC <= '1';
                         ld_A       <= '0';
                         ld_B       <= '0';
                         ld_Stop    <= '0';
                         ALU_M      <= "--";
                         ld_O       <= '0';

                    if (QC_Ackn_A='1')
                      then next_STATE<=S_6;
                      else next_STATE<=S_5;
                    end if;
```

```
           when S_6 =>     DP_Reset   <= '0';
                           A_Ackn_TB  <= '0';
                           A_Ready_QC <= '0';
                           ld_A       <= '0';
                           ld_B       <= '0';
                           ld_Stop    <= '0';
                           ALU_M      <= "--";
                           ld_O       <= '0';

                          if (QC_Ackn_A='0')
                            then if (StopMsg='1')
                                   then next_STATE<=S_END;
                                   else next_STATE<=S_1;
                                 end if;
                            else next_STATE<=S_6;
                          end if;

           when S_END=>   -- nothing (Procedure quits)

         end case;
       end if;

  end process;

  end FSM_A_dHS_q_clock_ex_behavioral;
```

C.3.12 Datapath of A (identical B)

VHDL/doubleHS_queued_SMs/Datapath/DP.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

-- File :         DP.vhd
-- implements :   Datapath (instanced twice: for A and for B)
-- for Example: FSMD (A) --> Memory --> FSMD (B)
--                      <-->   FSM <-->
--                using double handshake protocol

Entity DP is
   port (Clock:            in std_logic;
         DP_Reset:         in std_logic;
         ld_A, ld_B, ld_Stop: in std_logic;
         StopMsg:          out std_logic;
         ALU_M:            in std_logic_vector (1 downto 0);
         ld_O:             in std_logic;
         In1, In2:         in std_logic_vector(31 downto 0);
         In_Stop:          in std_logic;
         O_Port:           out std_logic_vector(31 downto 0);
         Out_Stop:         out std_logic);
end DP;

Architecture DP_schematic of DP is

   -- connection signals
   signal A, B, O: std_logic_vector(31 downto 0);
   signal ALU_Out: std_logic_vector(31 downto 0);
   signal Stop:    std_logic;

   component Reg_32bit
      Port ( Clock:    in std_logic;
             Reset:    in std_logic;
             Load:     in std_logic;
             Data_In:  in std_logic_vector (31 downto 0);
             Data_Out: out std_logic_vector (31 downto 0) );
   end component;

   component Latch_impl
      Port ( Clock:     in std_logic;
             Reset:     in std_logic;
             Load:      in std_logic;
             Data_In:   in std_logic;
             Data_Out: out std_logic);
   end component;

   component ALU_32
      Port ( Mode:       in std_logic_vector (1 downto 0);
             Data_In1:   in std_logic_vector (31 downto 0);
             Data_In2:   in std_logic_vector (31 downto 0);
             Data_Out:   out std_logic_vector (31 downto 0) );
   end component;

begin

Register_A: Reg_32bit
   Port Map ( Clock=>Clock, Reset=>DP_Reset, Load=>ld_A,
              Data_In=>In1, Data_Out=>A);

Register_B: Reg_32bit
   Port Map ( Clock=>Clock, Reset=>DP_Reset, Load=>ld_B,
              Data_In=>In2, Data_Out=>B);
```

```
  Latch_Stop: Latch_impl
     Port Map ( Clock=>Clock, Reset=>DP_Reset, Load=>ld_Stop,
                Data_In=>In_Stop, Data_Out=>Stop);

  ALU: ALU_32
     Port Map ( Mode=>ALU_M, Data_In1=>A, Data_In2=>B,
                Data_Out=>ALU_Out);

  Register_O: Reg_32bit
     Port Map ( Clock=>Clock, Reset=>DP_Reset, Load=>ld_O,
                Data_In=>ALU_Out, Data_Out=>O);

  O_Port <= O;
  Out_Stop <= Stop;
  StopMsg <= Stop;

end DP_schematic;
```

C.3.13 FSM Controlling Datapath Queue

VHDL/doubleHS_queued_SMs/FSM_dHS_q_clock/FSM_and_D_dHS_q_clock_ex_Queue.vhd

```
-- File :          FSM_and_D_dHS_q_clock_ex_Queue.vhd
-- implements :    bounding of FSM and Datapath of Queue
-- of Model:       FSM and separate Datapath
-- for Example: FSM,DP (A) --> FSM,DP(Queue) --> FSM,DP (B)
--                 using double handshake protocol

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

Entity FSM_and_D_Queue_dHS_clock_ex is
   -- pragma template
   generic (Q_W, Q_D, S_W: integer);
   port (Reset, Start:          in std_logic;
         Clock_Q:          in std_logic;
         A_Ready_QC, B_Rd_Req_QC: in std_logic;
         QC_Ackn_A, QC_Ready_B: out std_logic;
         Q_Empty_Port, Q_Full_Port: out std_logic;
         Data_AQ:              in std_logic_vector(Q_W-1 downto 0);
         Buffer_QB_Port:            out std_logic_vector(Q_W-1 downto 0));
end FSM_and_D_Queue_dHS_clock_ex;


Architecture FSM_and_D_Queue_dHS_clock_ex_behavioral of FSM_and_D_Queue_dHS_clock_ex is

Component Queue_Buffered
   generic( Q_W, Q_D, S_W: integer);
   port ( Clock: in std_logic;
          Q_Reset: in std_logic;
          Q_En, Q_Wr: in std_logic;
          Buffer_ld: in std_logic;
          Q_In: in std_logic_vector(Q_W-1 downto 0);
          Buffer_Out: out std_logic_vector(Q_W-1 downto 0);
          Q_Full, Q_Empty: out std_logic);
   end Component;

Component Queue_Buffered_Control
   port ( Clock_Q: in std_logic;
          Q_Reset: in std_logic;
          Q_Start: in std_logic;
          Q_Wr, Q_En, Buffer_ld: out std_logic;
          Q_Full, Q_Empty: in std_logic;
          A_Ready_QC, B_Rd_Req_QC: in std_logic;
          QC_Ackn_A, QC_Ready_B: out std_logic);
   end Component;

-- queue outputs
signal Buffer_QB: std_logic_vector(Q_W-1 downto 0);
signal Buffer_ld: std_logic;
signal Q_Start: std_logic;
signal Q_En, Q_Wr: std_logic;
signal Q_Full, Q_Empty: std_logic;

begin

Queue_Buffered_I: Queue_Buffered
    generic Map (Q_W=>Q_W, Q_D=>Q_D, S_W=>S_W)
    Port Map ( Clock=>Clock_Q,
               Q_Reset=>Reset,
               Q_En=>Q_En, Q_Wr=>Q_Wr,
               Buffer_ld=>Buffer_ld,
               Q_In=>Data_AQ,
               Buffer_Out=>Buffer_QB,
                   Q_Full=>Q_Full , Q_Empty=>Q_Empty );

 Queue_Buffered_Control_I : Queue_Buffered_Control
      Port Map ( Clock_Q=>Clock_Q ,
                   Q_Reset=>Reset ,
                   Q_Start=>Start ,
                   Q_Wr=>Q_Wr, Q_En=>Q_En, Buffer_Id=>Buffer_Id ,
                   Q_Full=>Q_Full , Q_Empty=>Q_Empty,
                   A_Ready_QC=>A_Ready_QC , B_Rd_Req_QC=>B_Rd_Req_QC ,
                   QC_Ackn_A=>QC_Ackn_A , QC_Ready_B=>QC_Ready_B );

Buffer_QB_Port <= Buffer_QB ;
 Q_Full_Port <= Q_Full ;
 Q_Empty_Port <= Q_Empty;

end FSM_and_D_Queue_dHS_clock_ex_behavioral ;
```

C.3.14 FSM of Queue

VHDL/doubleHS_queued_SMs/FSM_dHS_q_clock/FSM_dHS_q_clock_ex_Queue.vhd

```
-- File :          FSM_dHS_q_clock_ex_Queue.vhd
-- implements :    FSM of Queue
-- of Model:       FSM and separate Datapath
-- for Example :   FSM,DP (A) --> FSM,DP(Queue) --> FSM,DP (B)
--                 using double handshake protocol

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

Entity Queue_Buffered_Control is
   Port ( Clock_Q: in std_logic;
          Q_Reset: in std_logic;
          Q_Start: in std_logic;
          Q_Wr, Q_En, Buffer_ld: out std_logic;
          Q_Full, Q_Empty: in std_logic;
          A_Ready_QC, B_Rd_Req_QC: in std_logic;
          QC_Ackn_A, QC_Ready_B: out std_logic);
end Queue_Buffered_Control;


Architecture Queue_Buffered_Control_beh of Queue_Buffered_Control is

type State_Set is (S_0, W_1, W_2, R_1, R_2, R_3, S_END);
signal next_STATE, STATE: State_Set;

begin

transition: Process
begin
  wait until (Clock_Q'event and Clock_Q = '1');
  STATE <= next_STATE;
end process;

Queue: Process(STATE, Q_Start, Q_Full, Q_Empty, A_Ready_QC, B_Rd_Req_QC)
begin
    if (Q_Reset = '1') then
      Q_En <= '0';
      Q_Wr <= '0';
      QC_Ackn_A <= '0';
      QC_Ready_B <= '0';
      Buffer_ld <=  '0';
      next_STATE<= S_0;
    else
      case STATE is
        when S_0 => Q_En <= '0';
                    Q_Wr <= '-';
                    QC_Ackn_A  <= '0';
                    QC_Ready_B <= '0';
                    Buffer_ld <=  '0';

                    if (Q_Empty='0' and B_Rd_Req_QC='1')
                      then next_STATE<= R_1;
                    elsif (Q_Full='0' and A_Ready_QC='1')
                      then next_STATE<= W_1;
                    end if;

        when W_1 => Q_En <= '1';
                    Q_Wr <= '1';
                    QC_Ackn_A  <= '0';
                    QC_Ready_B <= '0';
                    Buffer_ld <=  '0';
                    next_STATE<= W_2;

        when W_2 => Q_En <= '0';
                    Q_Wr <= '-';
                    QC_Ackn_A   <= '1';
                    QC_Ready_B <= '0';
                    Buffer_ld <=  '0';

                    if (A_Ready_QC='0')
                      then next_STATE<= S_0;
                    else
                      next_STATE<= W_2;
                    end if;

        when R_1 => Q_En <= '1';
                    Q_Wr <= '0';
                    QC_Ackn_A   <= '0';
                    QC_Ready_B <= '0';
                    Buffer_ld <=  '1';

                    next_STATE<= R_2;

        when R_2 => Q_En <= '0';
                    Q_Wr <= '0';
                    QC_Ackn_A   <= '0';
                    QC_Ready_B <= '0';
                    Buffer_ld <=  '0';

                    next_STATE<= R_3;

        when R_3 => Q_En <= '0';
                    Q_Wr <= '-';
                    QC_Ackn_A   <= '0';
                    QC_Ready_B <= '1';
                    Buffer_ld <=  '0';

                    if (B_Rd_Req_QC='0')
                      then next_STATE<= S_0;
                    else
                      next_STATE<= R_3;
                    end if;

        when S_END=>

      end case;
    end if;
end process;

end Queue_Buffered_Control_beh;
```

C.3.15 Datapath of Queue (Buffered)

VHDL/doubleHS_queued_SMs/Datapath_Queue/DP_all.vhd

```
-- File :         DP_all.vhd
-- implements :   Datapath of Buffered Queue
-- for Example : FSM,DP (A) --> FSM,DP (Queue) --> FSM,DP (B)
--                using double handshake protocol

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

Entity Queue_Buffered is
   generic( Q_W, Q_D, S_W: integer);
   port ( Clock: in std_logic;
          Q_Reset: in std_logic;
          Q_En, Q_Wr: in std_logic;
          Buffer_ld : in std_logic;
          Q_In: in std_logic_vector(Q_W-1 downto 0);
          Buffer_Out: out std_logic_vector(Q_W-1 downto 0);
          Q_Full, Q_Empty: out std_logic );
   end Queue_Buffered;

Architecture Queue_Buffered_Arch of Queue_Buffered is
signal Q_Out: std_logic_vector(Q_W-1 downto 0);
signal Buffer_In: std_logic_vector(Q_W-1 downto 0);
signal Q_N_Reset: std_logic;

Component Queue
   generic( Q_W, Q_D, S_W: integer);
   port ( Clock: in std_logic;
          Q_Reset: in std_logic;
          Q_En, Q_Wr: in std_logic;
          Q_In: in std_logic_vector(Q_W-1 downto 0);
          Q_Out: out std_logic_vector(Q_W-1 downto 0);
          Q_Full, Q_Empty: out std_logic );
   end Component;

Component D_FF
    port ( Clk: in std_logic;
           Rst, En: in std_logic;
           D: in std_logic;
           Q, Q_T: out std_logic  );
    end Component;

begin

Queue_I: Queue
    generic map(Q_W=>Q_W, Q_D=>Q_D, S_W=>S_W)
    port map ( Clock=>Clock,
          Q_Reset=>Q_Reset,
          Q_En=>Q_En, Q_Wr=>Q_Wr,
          Q_In=>Q_In,
          Q_Out=>Q_Out,
          Q_Full=>Q_Full, Q_Empty=>Q_Empty);

D_FFs: for i in Q_W-1 downto 0 generate
        FF: D_FF
            port map ( Clk=>Clock,
                       Rst=>Q_N_Reset, En=>Buffer_ld,
                       D=>Buffer_In(i),
                       Q=>Buffer_Out(i), Q_T=>open);
       end generate;

Buffer_In <= Q_Out;
Q_N_Reset <= not Q_Reset;
```

```
end Queue_Buffered_Arch ;
```

C.3.16 Datapath of Unbuffered Queue

VHDL/doubleHS_queued_SMs/Datapath_Queue/DP_Queue.vhd

```
-- File :          DP_all.vhd
-- implements :    Datapath of Queue (which will be wrapped by buffers)
-- for Example: FSM,DP (A) --> FSM,DP (Queue) --> FSM,DP (B)
--                 using double handshake protocol

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.Std_Logic_unsigned.ALL;

-- Queue

Entity Queue is
   generic( Q_W, Q_D, S_W: integer);
   port ( Clock: in std_logic;
          Q_Reset: in std_logic;
          Q_En, Q_Wr: in std_logic;
          Q_In: in std_logic_vector(Q_W-1 downto 0);
          Q_Out: out std_logic_vector(Q_W-1 downto 0);
          Q_Full, Q_Empty: out std_logic );
   end Queue; -- pragma template

Architecture Queue_arch of Queue is

type fifo_sh_transp is ARRAY(Q_W-1 downto 0) of std_logic_vector(Q_D-1 downto 0);
signal Q_Sh, Q_Sel: fifo_sh_transp;
type fifo_sh is ARRAY(Q_D-1 downto 0) of std_logic_vector(Q_W-1 downto 0);
signal Q: fifo_sh;

signal Sh_S: std_logic;
signal Sel_S: std_logic_vector(S_W-1 downto 0);
signal Coun_O: std_logic_vector(S_W downto 0);
signal Coun_D: std_logic;
signal Q_N_Reset: std_logic;
signal HL: std_logic;
signal LL: std_logic;

Component Shifter_Dx1
   generic( D: integer);
   port ( Reset: in std_logic;
          Clk: in std_logic;
          S_1, S_0: in std_logic;
          I_L, I_R: in std_logic;
          Q: out std_logic_vector(D-1 downto 0));
   end Component;

Component Sel_W
   generic (W, S_W: integer);
   -- W: Number of 1 Bit inputs joined in a std_logic_vector;
   -- S_W: Number of Select bits ( must be >= lb (W) );
   port ( I: in std_logic_vector(W-1 downto 0);
          S: in std_logic_vector(S_W-1 downto 0);
          O: out std_logic );
   end Component;

Component Counter_W
   generic(W: integer);
   port ( Reset, Set: in std_logic;
          Clk: in std_logic;
          D: in std_logic;
          E: in std_logic;
          C: out std_logic_vector(W-1 downto 0));
   end Component;
```

```
begin

Shifters : for i in Q_W-1 downto 0 generate
            Sh: Shifter_Dx1
                generic map (D=>Q_D)
                port map (  Reset=>Q_N_Reset,
                           Clk=>Clock,
                           S_1=>Sh_S, S_0=>Sh_S,
                           I_L=>Q_In(i), I_R=>LL,
                           Q=>Q_Sh(i)(Q_D-1 downto 0));
            Twist: for j in Q_D-1 downto 0 generate
                           Q(j)(i) <= Q_Sh(i)(Q_D-1 -j);
                   end generate;
          end generate;


Selectors: for i in Q_W-1 downto 0 generate
             Transp: for j in Q_D-1 downto 0 generate
                           Q_Sel(i)(j) <= Q(j)(i);
                     end generate;
             Sel: Sel_W
                generic map (W=>Q_D, S_W=>S_W)
                port map (  I=>Q_Sel(i)(Q_D-1 downto 0),
                           S=>Sel_S, O=>Q_Out(i));
           end generate;

Counter: Counter_W
           generic map (W=>S_W+1)
           port map (  Reset=>HL, Set=>Q_N_Reset,
                      Clk=>Clock,
                      D=>Coun_D, E=>Q_En, C=>Coun_O);


Sh_S <= (Q_Wr and Q_En);
Sel_S <= Coun_O(S_W-1 downto 0);
Coun_D <= ((not Q_Wr) and Q_En);


HL <= '1';
LL <= '0';
Q_N_Reset <= not Q_Reset;

Q_Empty <= '1' when (Coun_O = conv_std_logic_vector(-1, Coun_O'length))
               else '0';
Q_Full  <= '1' when (Coun_O = conv_std_logic_vector(Q_D-1, Coun_O'length))
               else '0';

end Queue_arch;
```

C.3.17 FSM Controlling Datapath B

`VHDL/doubleHS_queued_SMs/FSM_dHS_q_clock/FSM_and_D_dHS_q_clock_ex_B.vhd`

```
-- File :          FSM_and_D_dHS_q_clock_ex_B.vhd
-- implements :    bounding of FSM B and Datapath B
-- of Model:       FSM and separate Datapath
-- for Example:    FSM,DP (A) --> FSM,DP(Queue) --> FSM,DP (B)
--                 using double handshake protocol

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

Entity FSM_and_D_B_dHS_q_clock_ex is
   port (Reset, Start:         in std_logic;
         Clock_B:              in std_logic;
         QC_Ready_B, TB_Ackn_B: in std_logic;
         B_Rd_Req_QC, B_Ready_TB: out std_logic;
         Buffer_QB:             in std_logic_vector (31 downto 0);
         Buffer_QB_Stop:             in std_logic;
         Out1_Port:            out std_logic_vector (31 downto 0);
         Out1_Stop_Port:       out std_logic);
end FSM_and_D_B_dHS_q_clock_ex;


Architecture FSM_and_D_B_dHS_q_clock_ex_behavioral of FSM_and_D_B_dHS_q_clock_ex is
-- outputs
signal Out1:  std_logic_vector (31 downto 0);
signal Out1_Stop: std_logic;

signal DP_Reset:     std_logic;
signal ld_A, ld_B, ld_Stop: std_logic;
signal StopMsg: std_logic;
signal ALU_M: std_logic_vector (1 downto 0);
signal ld_O: std_logic;
signal CMP: std_logic_vector(1 downto 0);
signal MUX_sel: std_logic;

Component DP
   port (Clock:          in std_logic;
         DP_Reset:       in std_logic;
         ld_A, ld_B, ld_Stop: in std_logic;
         StopMsg:        out std_logic;
         ALU_M:          in std_logic_vector (1 downto 0);
         ld_O:           in std_logic;
         In1, In2:       in std_logic_vector(31 downto 0);
         In_Stop:        in std_logic;
         O_Port:         out std_logic_vector(31 downto 0);
         Out_Stop:       out std_logic);
end Component;

Component FSM_B_dHS_q_clock_ex
   port (Reset, Start:          in std_logic;
         Clock_B:               in std_logic;
         QC_Ready_B, TB_Ackn_B: in std_logic;
         B_Rd_Req_QC, B_Ready_TB: out std_logic;
         DP_Reset:              out std_logic;
         ld_A, ld_B, ld_Stop:   out std_logic;
         StopMsg:               in std_logic;
         ALU_M:                 out std_logic_vector (1 downto 0);
         ld_O:                  out std_logic);
end Component;

begin -- Architecture
DP_B_I: DP
    Port Map (Clock=>Clock_B, DP_Reset=>DP_Reset,
               ld_A=>ld_A, ld_B=>ld_B, ld_Stop=>ld_Stop,
               StopMsg=>StopMsg, ALU_M=>ALU_M, ld_O=>ld_O,
               In1=>Buffer_QB, In2=>Buffer_QB,
               In_Stop=>Buffer_QB_Stop,
               O_Port=>Out1, Out_Stop=>Out1_Stop);

FSM_B_dHS_q_clock_I: FSM_B_dHS_q_clock_ex
     Port Map (Reset=>Reset, Start=>Start,
               Clock_B=>Clock_B,
               QC_Ready_B=>QC_Ready_B, TB_Ackn_B=>TB_Ackn_B,
               B_Rd_Req_QC=>B_Rd_Req_QC, B_Ready_TB=>B_Ready_TB,
               DP_Reset=>DP_Reset,
               ld_A=>ld_A, ld_B=>ld_B, ld_Stop=>ld_Stop,
               StopMsg=>StopMsg, ALU_M=>ALU_M, ld_O=>ld_O);

-- Entity Outputs
Out1_Port <= Out1;
Out1_Stop_Port <= Out1_Stop;

end FSM_and_D_B_dHS_q_clock_ex_behavioral;
```

C.3.18 FSM of B

VHDL/doubleHS_queued_SMs/FSM_dHS_q_clock/FSM_dHS_q_clock_ex_B.vhd

```
-- File :          FSM_dHS_q_clock_ex_B.vhd
-- implements :    FSM B
-- of Model:       FSM and separate Datapath
-- for Example:    FSM,DP (A) --> FSM,DP(Queue) --> FSM,DP (B)
--                 using double handshake protocol

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

Entity FSM_B_dHS_q_clock_ex is
   port (Reset, Start:        in std_logic;
         Clock_B:             in std_logic;
         QC_Ready_B, TB_Ackn_B: in std_logic;
         B_Rd_Req_QC, B_Ready_TB: out std_logic;
         DP_Reset:              out std_logic;
         ld_A, ld_B, ld_Stop:   out std_logic;
         StopMsg:               in std_logic;
         ALU_M:                 out std_logic_vector (1 downto 0);
         ld_O:                  out std_logic);
end FSM_B_dHS_q_clock_ex;


Architecture FSM_B_dHS_q_clock_ex_behavioral of FSM_B_dHS_q_clock_ex is

type State_Set is (S_BEGIN, S_1, S_2, S_3, S_4, S_5, S_6, S_END);
signal next_STATE, STATE: State_Set;

begin -- Architecture

transition : Process
begin
  wait until (Clock_B'event and Clock_B = '1');
  STATE <= next_STATE;
end process;

behavior1: process(STATE, Start, QC_Ready_B, TB_Ackn_B, StopMsg)
begin
    if (Reset = '1') then
         DP_Reset    <= '1';
         B_Rd_Req_QC <= '0';
         B_Ready_TB  <= '0';
         ld_A        <= '0';
         ld_B        <= '0';
         ld_Stop     <= '0';
         ALU_M       <= "--";
         ld_O        <= '0';
      next_STATE <= S_BEGIN;
    else
      case next_STATE is

        when S_BEGIN => DP_Reset    <= '1';
                        B_Rd_Req_QC <= '0';
                        B_Ready_TB  <= '0';
                        ld_A        <= '0';
                        ld_B        <= '0';
                        ld_Stop     <= '0';
                        ALU_M       <= "--";
                        ld_O        <= '0';

                        if Start = '1'
                          then next_STATE <= S_1;
                          else next_STATE <= S_BEGIN;
```

```
                      end if;

        when S_1 =>     DP_Reset    <= '0';
                        B_Rd_Req_QC <= '1';
                        B_Ready_TB  <= '0';
                        ld_A        <= '1';
                        ld_B        <= '1';
                        ld_Stop     <= '1';
                        ALU_M       <= "--";
                        ld_O        <= '0';

                       if (QC_Ready_B='1')
                         then next_STATE<= S_2;
                         else next_STATE<= S_1;
                       end if;

        when S_2 =>    DP_Reset    <= '0';
                       B_Rd_Req_QC <= '0';
                       B_Ready_TB  <= '0';
                       ld_A        <= '0';
                       ld_B        <= '0';
                       ld_Stop     <= '0';
                       ALU_M       <= "10";
                       ld_O        <= '0';

                       if (QC_Ready_B='0')
                         then next_STATE<= S_3;
                         else next_STATE<= S_2;
                       end if;

        when S_3 =>    DP_Reset    <= '0';
                       B_Rd_Req_QC <= '0';
                       B_Ready_TB  <= '0';
                       ld_A        <= '0';
                       ld_B        <= '0';
                       ld_Stop     <= '0';
                       ALU_M       <= "10";
                       ld_O        <= '1';

                       next_STATE <= S_4;

        when S_4 =>    DP_Reset    <= '0';
                       B_Rd_Req_QC <= '0';
                       B_Ready_TB  <= '0';
                       ld_A        <= '0';
                       ld_B        <= '0';
                       ld_Stop     <= '0';
                       ALU_M       <= "--";
                       ld_O        <= '0';

                       next_STATE <= S_5;

        when S_5 =>    DP_Reset    <= '0';
                       B_Rd_Req_QC <= '0';
                       B_Ready_TB  <= '1';
                       ld_A        <= '0';
                       ld_B        <= '0';
                       ld_Stop     <= '0';
                       ALU_M       <= "--";
                       ld_O        <= '0';

                       if (TB_Ackn_B='1')
                         then next_STATE<=S_6;
                         else next_STATE<=S_5;
                       end if;
        when S_6 =>   DP_Reset    <= '0';
                      B_Rd_Req_QC <= '0';
                      B_Ready_TB  <= '0';
                      ld_A        <= '0';
                      ld_B        <= '0';
                      ld_Stop     <= '0';
                      ALU_M       <= "--";
                      ld_O        <= '0';

                      if (TB_Ackn_B='0')
                        then if (StopMsg='1')
                               then next_STATE<=S_END;
                               else next_STATE<=S_1;
                             end if;
                        else next_STATE<=S_6;
                      end if;

        when S_END=>  -- nothing (Procedure quits)

      end case;
     end if;

end process;

end FSM_B_dHS_q_clock_ex_behavioral;
```

C.3.19 Datapath of B identical to A (see A)

The datapath for the receiver B is identical to the datapath of the sender A. → See datapath for sender A.

C.4 Transfer via Memory with Double Handshake

- A (FSM Controlling Datapath) → Memory || Arbiter (FSM) → B (FSM Controlling Datapath)
  - FSM Controlling Datapath A
    - FSM of A
    - Datapath of A (identical B)
  - Memory
  - Arbiter (FSM)
  - FSM Controlling Datapath B
    - FSM of B
    - Datapath of B identical to A (see A)

C.4.1 A (FSM Controlling Datapath) → Memory || Arbiter (FSM) → B (FSM Controlling Datapath)

VHDL/doubleHS_mem_SMs/FSM_dHS_mem_clock/FSM_and_D_dHS_mem_clock_ex_top.vhd

```vhdl
-- File :          FSM_and_D_dHS_mem_clock_ex_top.vhd
-- bounding the three parts of the example
-- for Example: FSM, DP (A) --> Memory --> FSM, DP (B)
--                          <-->    FSM <-->
--                using double handshake protocol

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity FSM_and_D_dHS_mem_clock_ex_top is
   port (Reset, A_Start, B_Start:        in std_logic;
         Clock_A, Clock_B, Clock_M:    in std_logic;
         TB_Ready_A, TB_Ackn_B: in std_logic;
         A_Ackn_TB, B_Ready_TB: out std_logic;
         A_Addr_In, B_Addr_In: in std_logic_vector(3 downto 0);
         In1, In2:              in std_logic_vector(31 downto 0);
         In_Stop:               in std_logic;
         Out1_Port:            out std_logic_vector(31 downto 0);
         Out1_Stop_Port:       out std_logic);
end FSM_and_D_dHS_mem_clock_ex_top;

architecture FSM_and_D_dHS_mem_clock_ex_top_arch of FSM_and_D_dHS_mem_clock_ex_top is

Component FSM_and_D_A_dHS_mem_clock_ex
   port (Reset, A_Start:        in std_logic;
         Clock_A:              in std_logic;
         TB_Ready_A, AC_Grant_A: in std_logic;
         A_Ackn_TB, A_Req_AC: out std_logic;
         A_nCS, A_nWE, A_nOE:  out std_logic;
         Addr_In:               in std_logic_vector(3 downto 0);
         Addr_Out:              out std_logic_vector(3 downto 0);
         In1, In2:              in std_logic_vector(31 downto 0);
         In_Stop:               in std_logic;
         Data_AM_and_Stop_Port: out std_logic_vector(32 downto 0) );
end Component;

Component Mem_Grant_Control
   Port ( Clock_M: in std_logic;
          AC_Reset: in std_logic;
          AC_Start: in std_logic;
          A_Req_AC, B_Req_AC: in std_logic;
          AC_Grant_A, AC_Grant_B: out std_logic );
end Component;

Component Memory_DxW
   generic(D, W, A_W: integer);
   Port (nCS, nWE, nOE: in std_logic;
         Addr: in std_logic_vector(A_W-1 downto 0);
         Data: inout std_logic_vector(W-1 downto 0) );
end Component;

Component FSM_and_D_B_dHS_mem_clock_ex
   port (Reset, B_Start:        in std_logic;
         Clock_B:              in std_logic;
         AC_Grant_B, TB_Ackn_B: in std_logic;
         B_Req_AC, B_Ready_TB: out std_logic;
         B_nCS, B_nWE, B_nOE:  out std_logic;
         Addr_In:               in std_logic_vector(3 downto 0);
         Addr_Out:              out std_logic_vector(3 downto 0);
         Data_MB_and_Stop_Port: in std_logic_vector(32 downto 0);
         Out1_Port:            out std_logic_vector(31 downto 0);
         Out1_Stop_Port :      out std_logic );
end Component;

signal A_Req_AC, B_Req_AC: std_logic;
signal AC_Grant_A, AC_Grant_B: std_logic;
signal Data_and_Stop: std_logic_vector(32 downto 0);
signal Addr: std_logic_vector(3 downto 0);
signal nCS, A_nCS, B_nCS, nWE, nOE: std_logic;

begin

FSM_and_D_A_dHS_mem_clock_I: FSM_and_D_A_dHS_mem_clock_ex
     port map ( Reset=>Reset, A_Start=>A_Start, Clock_A=>Clock_A,
         TB_Ready_A=>TB_Ready_A, AC_Grant_A=>AC_Grant_A,
         A_Ackn_TB=>A_Ackn_TB, A_Req_AC=>A_Req_AC,
         A_nCS=>A_nCS, A_nWE=>nWE, A_nOE=>nOE,
         Addr_In=>A_Addr_In,
         Addr_Out=>Addr,
         In1=>In1, In2=>In2,
         In_Stop=>In_Stop,
         Data_AM_and_Stop_Port=>Data_and_Stop);

Mem_FSM_dHS_clock_I: Mem_Grant_Control
   Port map ( Clock_M=>Clock_M,
          AC_Reset=>Reset,
          AC_Start=>A_Start,
          A_Req_AC=>A_Req_AC, B_Req_AC=>B_Req_AC,
          AC_Grant_A=>AC_Grant_A, AC_Grant_B=>AC_Grant_B);

Memory_I: Memory_DxW
   generic map(D=>16, W=>33, A_W=>4)
   Port map (nCS=>nCS, nWE=>nWE, nOE=>nOE,
         Addr=>Addr,
         Data=>Data_and_Stop);

FSM_and_D_B_dHS_mem_clock_I: FSM_and_D_B_dHS_mem_clock_ex
     port map ( Reset=>Reset, B_Start=>B_Start, Clock_B=>Clock_B,
         AC_Grant_B=>AC_Grant_B, TB_Ackn_B=>TB_Ackn_B,
         B_Req_AC=>B_Req_AC, B_Ready_TB=>B_Ready_TB,
         B_nCS=>B_nCS, B_nWE=>nWE, B_nOE=>nOE,
         Addr_In=>B_Addr_In,
         Addr_Out=>Addr,
         Data_MB_and_Stop_Port=>Data_and_Stop,
         Out1_Port=>Out1_Port,
         Out1_Stop_Port=>Out1_Stop_Port  );

nCS <= A_nCS and B_nCS;

end FSM_and_D_dHS_mem_clock_ex_top_arch;
```

C.4.2 FSM Controlling Datapath A

`VHDL/doubleHS_mem_SMs/FSM_dHS_mem_clock/FSM_and_D_dHS_mem_clock_ex_A.vhd`

```
-- File :          FSM_and_D_dHS_mem_clock_ex_A.vhd
-- implements :    bounding of FSM A and Datapath A
-- of Model:       FSM and separate Datapath (FSM + D)
-- for Example:    FSM,DP (A) --> Memory --> FSM,DP (B)
--                         <-->   FSM <-->
--                 using double handshake protocol

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

Entity FSM_and_D_A_dHS_mem_clock_ex is
   port (Reset, A_Start:          in std_logic;
         Clock_A:               in std_logic;
         TB_Ready_A, AC_Grant_A: in std_logic;
         A_Ackn_TB, A_Req_AC: out std_logic;
         A_nCS, A_nWE, A_nOE:  out std_logic;
         Addr_In:               in std_logic_vector(3 downto 0);
         Addr_Out:              out std_logic_vector(3 downto 0);
         In1, In2:              in std_logic_vector(31 downto 0);
         In_Stop:               in std_logic;
         Data_AM_and_Stop_Port: out std_logic_vector(32 downto 0) );
end FSM_and_D_A_dHS_mem_clock_ex;


Architecture FSM_and_D_A_dHS_mem_clock_ex_structural of FSM_and_D_A_dHS_mem_clock_ex is
-- outputs
signal Data_AM:  std_logic_vector (31 downto 0);
signal Data_AM_Stop: std_logic;

signal DP_Reset:     std_logic;
signal ld_A, ld_B, ld_Stop: std_logic;
signal StopMsg: std_logic;
signal ALU_M: std_logic_vector (1 downto 0);
signal ld_O, A_tri_en: std_logic;
signal A_ld_Addr, A_inc_Addr: std_logic;
signal CMP: std_logic_vector(1 downto 0);
signal MUX_sel: std_logic;

Component DP
   port (Clock:           in std_logic;
         DP_Reset:        in std_logic;
         ld_A, ld_B, ld_Stop: in std_logic;
         StopMsg:         out std_logic;
         ALU_M:           in std_logic_vector (1 downto 0);
         ld_O, tri_en:    in std_logic;
         ld_Addr, inc_Addr: in std_logic;
         Addr_In:         in std_logic_vector(3 downto 0);
         Addr_Out_Port:   out std_logic_vector(3 downto 0);
         In1, In2:        in std_logic_vector(31 downto 0);
         In_Stop:         in std_logic;
         O_Port:          out std_logic_vector(31 downto 0);
         Out_Stop:        out std_logic);
end Component;

Component FSM_A_dHS_mem_clock_ex
   port (Reset, A_Start:          in std_logic;
         Clock_A:               in std_logic;
         TB_Ready_A, AC_Grant_A: in std_logic;
         A_Ackn_TB, A_Req_AC: out std_logic;
         A_nCS, A_nWE, A_nOE:  out std_logic;
         DP_Reset:              out std_logic;
         ld_A, ld_B, ld_Stop:  out std_logic;
```

```
          StopMsg:                 in std_logic;
          ALU_M:                   out std_logic_vector (1 downto 0);
          ld_O, A_tri_en:          out std_logic;
          A_ld_Addr, A_inc_Addr:       out std_logic);
end Component;

begin -- Architecture
DP_A_I: DP
      Port Map (Clock=>Clock_A, DP_Reset=>DP_Reset,
                ld_A=>ld_A, ld_B=>ld_B, ld_Stop=>ld_Stop,
                StopMsg=>StopMsg, ALU_M=>ALU_M, ld_O=>ld_O, tri_en=>A_tri_en,
                ld_Addr=>A_ld_Addr, inc_Addr=>A_inc_Addr,
                Addr_In=>Addr_In, Addr_Out_Port=>Addr_Out,
                In1=>In1, In2=>In2,
                In_Stop=>In_Stop,
                O_Port=>Data_AM, Out_Stop=>Data_AM_Stop);

FSM_A_dHS_mem_clock_I: FSM_A_dHS_mem_clock_ex
      Port Map (Reset=>Reset, A_Start=>A_Start,
                Clock_A=>Clock_A,
                TB_Ready_A=>TB_Ready_A, AC_Grant_A=>AC_Grant_A,
                A_Ackn_TB=>A_Ackn_TB, A_Req_AC=>A_Req_AC,
                A_nCS=>A_nCS, A_nWE=>A_nWE, A_nOE=>A_nOE,
                DP_Reset=>DP_Reset,
                ld_A=>ld_A, ld_B=>ld_B, ld_Stop=>ld_Stop,
                StopMsg=>StopMsg, ALU_M=>ALU_M, ld_O=>ld_O, A_tri_en=>A_tri_en,
                A_ld_Addr=>A_ld_Addr, A_inc_Addr=>A_inc_Addr);

-- Entity Outputs
Data_AM_and_Stop_Port <= Data_AM_Stop & Data_AM;

end FSM_and_D_A_dHS_mem_clock_ex_structural;
```

C.4.3 FSM of A

VHDL/doubleHS_mem_SMs/FSM_dHS_mem_clock/FSM_dHS_mem_clock_ex_A.vhd

```
-- File :          FSM_dHS_mem_clock_ex_A.vhd
-- implements :    FSM A
-- of Model:       FSM and separate Datapath (FSM + D)
-- for Example:    FSM,DP (A) --> Memory --> FSM,DP (B)
--                              <-->   FSM <-->
--                 using double handshake protocol

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

Entity FSM_A_dHS_mem_clock_ex is
   port (Reset, A_Start:        in std_logic;
         Clock_A:             in std_logic;
         TB_Ready_A, AC_Grant_A: in std_logic;
         A_Ackn_TB, A_Req_AC: out std_logic;
         A_nCS, A_nWE, A_nOE:     out std_logic;
         DP_Reset:              out std_logic;
         ld_A, ld_B, ld_Stop:   out std_logic;
         StopMsg:               in std_logic;
         ALU_M:                 out std_logic_vector (1 downto 0);
         ld_O, A_tri_en:        out std_logic;
         A_ld_Addr, A_inc_Addr: out std_logic);
end FSM_A_dHS_mem_clock_ex;


Architecture FSM_A_dHS_mem_clock_ex_behavioral of FSM_A_dHS_mem_clock_ex is

type State_Set is (S_BEGIN, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9, S_END);
signal next_STATE, STATE: State_Set;

begin -- Architecture

transition: Process
begin
  wait until (Clock_A'event and Clock_A = '1');
  STATE <= next_STATE;
end process;

behavior1: process(STATE, A_Start, TB_Ready_A, AC_Grant_A, StopMsg)
begin
    if (Reset = '1') then
         DP_Reset   <= '1';
         A_Ackn_TB  <= '0';
         A_Req_AC   <= '0';
         ld_A       <= '0';
         ld_B       <= '0';
         ld_Stop    <= '0';
         ALU_M      <= "--";
         ld_O       <= '0';
         A_ld_Addr  <= '0';
         A_inc_Addr <= '0';
         A_tri_en   <= '0';
         A_nCS   <= '1';
         A_nWE   <= 'Z';
         A_nOE   <= 'Z';
      next_STATE <= S_BEGIN;
    else
      case STATE is

        when S_BEGIN => DP_Reset   <= '1';
                        A_Ackn_TB  <= '0';
                        A_Req_AC   <= '0';
```

```
                         ld_A        <= '0';
                         ld_B        <= '0';
                         ld_Stop     <= '0';
                         ALU_M       <= "--";
                         ld_O        <= '0';
                         A_ld_Addr <= '0';
                         A_inc_Addr<= '0';
                         A_tri_en    <= '0';
                         A_nCS   <= '1';
                         A_nWE   <= 'Z';
                         A_nOE   <= 'Z';

                        if  A_Start  =  '1'
                          then next_STATE <= S_1;
                          else  next_STATE <= S_BEGIN;
                        end if;

        when S_1 =>      DP_Reset    <= '0';
                         A_Ackn_TB <= '0';
                         A_Req_AC    <= '1';
                         ld_A        <= '0';
                         ld_B        <= '0';
                         ld_Stop     <= '0';
                         ALU_M       <= "--";
                         ld_O        <= '0';
                         A_ld_Addr <= '1';
                         A_inc_Addr<= '0';
                         A_tri_en    <= '0';
                         A_nCS   <= '1';
                         A_nWE   <= 'Z';
                         A_nOE   <= 'Z';

                        if  (AC_Grant_A  =  '1')
                          then next_STATE <= S_2;
                          else  next_STATE <= S_1;
                        end if;

        when S_2 =>      DP_Reset    <= '0';
                         A_Ackn_TB <= '0';
                         A_Req_AC    <= '1';
                         ld_A        <= '0';
                         ld_B        <= '0';
                         ld_Stop     <= '0';
                         ALU_M       <= "--";
                         ld_O        <= '0';
                         A_ld_Addr <= '0';
                         A_inc_Addr<= '0';
                         A_tri_en    <= '1';
                         A_nCS   <= '0';
                         A_nWE   <= '1';
                         A_nOE   <= '1';

                        if  (TB_Ready_A='1')
                          then next_STATE<= S_3;
                          else  next_STATE<= S_2;
                        end if;

         when S_3 =>     DP_Reset    <= '0';
                         A_Ackn_TB <= '1';
                         A_Req_AC    <= '1';
                         ld_A        <= '1';
                         ld_B        <= '1';
                         ld_Stop     <= '1';
                         ALU_M       <= "11";
                         ld_O        <= '0';
                         A_ld_Addr <= '0';
                        A_inc_Addr<= '0';
                        A_tri_en   <= '1';
                        A_nCS  <= '0';
                        A_nWE  <= '1';
                        A_nOE  <= '1';

                      if (TB_Ready_A='0')
                        then next_STATE<= S_4;
                        else next_STATE<= S_3;
                      end if;

        when S_4 =>     DP_Reset   <= '0';
                        A_Ackn_TB  <= '0';
                        A_Req_AC   <= '1';
                        ld_A       <= '0';
                        ld_B       <= '0';
                        ld_Stop    <= '0';
                        ALU_M      <= "11";
                        ld_O       <= '1';
                        A_ld_Addr  <= '0';
                        A_inc_Addr<= '0';
                        A_tri_en   <= '1';
                        A_nCS  <= '0';
                        A_nWE  <= '1';
                        A_nOE  <= '1';

                      next_STATE<= S_5;

        when S_5 =>     DP_Reset   <= '0';
                        A_Ackn_TB  <= '0';
                        A_Req_AC   <= '1';
                        ld_A       <= '0';
                        ld_B       <= '0';
                        ld_Stop    <= '0';
                        ALU_M      <= "--";
                        ld_O       <= '0';
                        A_ld_Addr  <= '0';
                        A_inc_Addr<= '0';
                        A_tri_en   <= '1';
                        A_nCS  <= '0';
                        A_nWE  <= '0';
                        A_nOE  <= '1';

                      next_STATE<= S_6;

        when S_6 =>     DP_Reset   <= '0';
                        A_Ackn_TB  <= '0';
                        A_Req_AC   <= '1';
                        ld_A       <= '0';
                        ld_B       <= '0';
                        ld_Stop    <= '0';
                        ALU_M      <= "--";
                        ld_O       <= '0';
                        A_ld_Addr  <= '0';
                        A_inc_Addr<= '0';
                        A_tri_en   <= '1';
                        A_nCS  <= '0';
                        A_nWE  <= '0';
                        A_nOE  <= '1';

                      next_STATE<= S_7;

        when S_7 =>     DP_Reset   <= '0';
                        A_Ackn_TB  <= '0';
                        A_Req_AC   <= '1';
                        ld_A       <= '0';
                        ld_B       <= '0';
                        ld_Stop    <= '0';
                        ALU_M      <= "--";
                        ld_O       <= '0';
                        A_ld_Addr  <= '0';
                        A_inc_Addr <= '0';
                        A_tri_en   <= '1';
                        A_nCS    <= '0';
                        A_nWE    <= '1';
                        A_nOE    <= '1';

                       next_STATE<= S_8;

         when S_8 =>    DP_Reset   <= '0';
                        A_Ackn_TB  <= '0';
                        A_Req_AC   <= '1';
                        ld_A       <= '0';
                        ld_B       <= '0';
                        ld_Stop    <= '0';
                        ALU_M      <= "--";
                        ld_O       <= '0';
                        A_ld_Addr  <= '0';
                        A_inc_Addr <= '1';
                        A_tri_en   <= '1';
                        A_nCS    <= '0';
                        A_nWE    <= '1';
                        A_nOE    <= '1';

                        if (StopMsg='1')
                          then next_STATE<=S_9;
                          else next_STATE<=S_2;
                        end if;

         when S_9 =>    DP_Reset   <= '0';
                        A_Ackn_TB  <= '0';
                        A_Req_AC   <= '0';
                        ld_A       <= '0';
                        ld_B       <= '0';
                        ld_Stop    <= '0';
                        ALU_M      <= "--";
                        ld_O       <= '0';
                        A_ld_Addr  <= '0';
                        A_inc_Addr <= '0';
                        A_tri_en   <= '0';
                        A_nCS    <= '1';
                        A_nWE    <= 'Z';
                        A_nOE    <= 'Z';

                       if (AC_Grant_A='0')
                         then next_STATE<=S_END;
                         else next_STATE<=S_9;
                       end if;

         when S_END=>  -- nothing

       end case;
     end if;

  end process;

  end FSM_A_dHS_mem_clock_ex_behavioral;
```

C.4.4 Datapath of A (identical B)

VHDL/doubleHS_mem_SMs/Datapath/DP.vhd

```
-- File :         DP.vhd
-- implements :  Datapath (instanced twice: for A and for B)
-- for Example: FSM,DP (A) --> Memory --> FSM,DP (B)
--                           <-->   FSM <-->
--               using double handshake protocol

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

Entity DP is
   port (Clock:           in std_logic;
         DP_Reset:        in std_logic;
         ld_A, ld_B, ld_Stop: in std_logic;
         StopMsg:         out std_logic;
         ALU_M:           in std_logic_vector (1 downto 0);
         ld_O, tri_en:    in std_logic;
         ld_Addr, inc_Addr: in std_logic;
         Addr_In:         in std_logic_vector(3 downto 0);
         Addr_Out_Port:   out std_logic_vector(3 downto 0);
         In1, In2:        in std_logic_vector(31 downto 0);
         In_Stop:         in std_logic;
         O_Port:          out std_logic_vector(31 downto 0);
         Out_Stop:        out std_logic);
end DP ;

Architecture DP_schematic of DP is

   -- connection signals
   signal A, B, O:  std_logic_vector(31 downto 0);
   signal ALU_Out:  std_logic_vector(31 downto 0);
   signal Stop:     std_logic;
   signal Addr_Out: std_logic_vector(3 downto 0);
   signal HL, LL:   std_logic;

   component Reg_32bit
      Port ( Clock:    in std_logic;
             Reset:    in std_logic;
             Load:     in std_logic;
             Data_In:  in std_logic_vector (31 downto 0);
             Data_Out: out std_logic_vector (31 downto 0) );
   end component;

   component Latch_impl
      Port ( Clock:     in std_logic;
             Reset:     in std_logic;
             Load:      in std_logic;
             Data_In:   in std_logic;
             Data_Out: out std_logic );
   end component;

   component Counter_4
      Port ( Clk: in std_logic;
             Reset: in std_logic;
             Ld: in std_logic;
             C_In: std_logic_vector(3 downto 0);
             D: in std_logic;
             En: in std_logic;
             C: out std_logic_vector(3 downto 0));
   end component;

   component ALU_32
      Port ( Mode:       in std_logic_vector (1 downto 0);
              Data_In1 :   in std_logic_vector (31 downto 0);
              Data_In2 :   in std_logic_vector (31 downto 0);
              Data_Out :   out std_logic_vector (31 downto 0) );
    end component;

    component tri
       generic (W: integer);
       port ( Enable:   in std_logic;
              Data_In : in std_logic_vector(W-1 downto 0);
              Data_Out: out std_logic_vector(W-1 downto 0) );
    end component;

    component tri_bit
       port ( Enable:   in std_logic;
              Data_In : in std_logic;
              Data_Out: out std_logic );
    end component;

begin

Register_A : Reg_32bit
    Port Map ( Clock=>Clock, Reset=>DP_Reset, Load=>ld_A,
               Data_In=>In1, Data_Out=>A);

Register_B : Reg_32bit
    Port Map ( Clock=>Clock, Reset=>DP_Reset, Load=>ld_B,
               Data_In=>In2, Data_Out=>B);

Latch_Stop : Latch_impl
    Port Map ( Clock=>Clock, Reset=>DP_Reset, Load=>ld_Stop,
               Data_In=>In_Stop, Data_Out=>Stop);

Addr_Counter: Counter_4
    Port Map ( Clk=>Clock, Reset=>DP_Reset, Ld=>ld_Addr,
               C_In=>Addr_In, D=>LL, En=>inc_Addr,
               C=>Addr_Out);

ALU: ALU_32
    Port Map ( Mode=>ALU_M, Data_In1=>A, Data_In2=>B,
               Data_Out=>ALU_Out);

Register_O : Reg_32bit
    Port Map ( Clock=>Clock, Reset=>DP_Reset, Load=>ld_O,
               Data_In=>ALU_Out, Data_Out=>O);

O_tri : tri
    generic map ( W=>32 )
    Port map ( Enable=>tri_en, Data_In=>O, Data_Out=>O_Port );

Out_Stop_tri : tri_bit
    Port map ( Enable=>tri_en, Data_In=>Stop, Data_Out=>Out_Stop );

StopMsg_tri : tri_bit
    Port map ( Enable=>tri_en, Data_In=>Stop, Data_Out=>StopMsg );

Addr_Out_tri : tri
    generic map ( W=>4 )
    Port map ( Enable=>tri_en, Data_In=>Addr_Out, Data_Out=>Addr_Out_Port );

HL <= '1';
LL <= '0';

end DP_schematic;
```

C.4.5 Memory

VHDL/doubleHS_mem_SMs/FSM_dHS_mem_clock/Mem_ex.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

-- File :          Mem_ex.vhd
-- implements :    Memory
-- for Example:    FSM,DP (A) --> Memory --> FSM,DP (B)
--                           <-->   FSM <-->
--                 using double handshake protocol

entity Memory_DxW is
   generic(D, W, A_W: integer);
   Port (nCS, nWE, nOE: in std_logic;
         Addr: in std_logic_vector(A_W-1 downto 0);
         Data: inout std_logic_vector(W-1 downto 0) );
end Memory_DxW; -- pragma template

Architecture Memory_DxW_arch of Memory_DxW is

type mem_array is ARRAY(D-1 downto 0) of std_logic_vector(W-1 downto 0);

signal mem: mem_array;

begin

read_write : process(nCS, nWE, nOE, Addr)
begin
  if (nCS = '0') then
    if (nWE = '0') then
      mem(conv_integer(Addr)) <= Data;
    elsif (nOE = '0') then
      Data <= mem(conv_integer(Addr));
    else
      Data <= (others=>'Z');
    end if;
  end if;
end process;

end Memory_DxW_arch;
```

C.4.6 Arbiter (FSM)

VHDL/doubleHS_mem_SMs/FSM_dHS_mem_clock/Mem_FSM_dHS_clock_ex.vhd

```
-- File :            Mem_FSM_dHS_clock_ex.vhd
-- implements :     FSM for granting memory access
-- for Example:     FSM,DP (A) --> Memory --> FSM,DP (B)
--                           <-->    FSM <-->
--                  using double handshake protocol

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

Entity Mem_Grant_Control is
   Port ( Clock_M: in std_logic;
          AC_Reset: in std_logic;
          AC_Start: in std_logic;
          A_Req_AC, B_Req_AC: in std_logic;
          AC_Grant_A, AC_Grant_B: out std_logic);
end Mem_Grant_Control;


Architecture Mem_Grant_Control_beh of Mem_Grant_Control is

type State_Set is (S_0, W, R);
signal next_STATE, STATE: State_Set;

begin

transition: Process
begin
  wait until (Clock_M'event and Clock_M = '1');
  STATE <= next_STATE;
end process;

Control: Process(STATE, AC_Start, A_Req_AC, B_Req_AC)
begin
    if (AC_Reset = '1') then
      AC_Grant_A <= '0';
      AC_Grant_B <= '0';
      next_STATE<= S_0;
    else
      case STATE is
        when S_0 => AC_Grant_A  <= '0';
                    AC_Grant_B <= '0';

                    if (A_Req_AC = '1')
                      then next_STATE<=W;
                    elsif (B_Req_AC = '1')
                      then next_STATE<=R;
                    else
                      next_STATE<= S_0;
                    end if;

        when W =>   AC_Grant_A <= '1';
                    AC_Grant_B <= '0';

                    if (A_Req_AC = '0')
                      then next_STATE<= S_0;
                    else
                      next_STATE <= W;
                    end if;

        when R =>   AC_Grant_A  <= '0';
                    AC_Grant_B <= '1';
                    if (B_Req_AC = '0')
                      then next_STATE<= S_0;
                    else
                      next_STATE <= R;
                    end if;

      end case;
    end if;
end process;

end Mem_Grant_Control_beh;
```

C.4.7 FSM Controlling Datapath B

`VHDL/doubleHS_mem_SMs/FSM_dHS_mem_clock/FSM_and_D_dHS_mem_clock_ex_B.vhd`

```
-- File :          FSM_and_D_dHS_mem_clock_ex_B.vhd
-- implements :    bounding of FSM B and Datapath B
-- of Model:       FSM and separate Datapath (FSM + D)
-- for Example:    FSM,DP (A) --> Memory --> FSM,DP (B)
--                           <-->   FSM <-->
--                 using double handshake protocol

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

Entity FSM_and_D_B_dHS_mem_clock_ex is
   port (Reset, B_Start:          in std_logic;
         Clock_B:               in std_logic;
         AC_Grant_B, TB_Ackn_B: in std_logic;
         B_Req_AC, B_Ready_TB:  out std_logic;
         B_nCS, B_nWE, B_nOE:   out std_logic;
         Addr_In:                in std_logic_vector(3 downto 0);
         Addr_Out:               out std_logic_vector(3 downto 0);
         Data_MB_and_Stop_Port: in std_logic_vector(32 downto 0);
         Out1_Port:             out std_logic_vector (31 downto 0);
         Out1_Stop_Port:        out std_logic);
end FSM_and_D_B_dHS_mem_clock_ex;


Architecture FSM_and_D_B_dHS_mem_clock_ex_behavioral of FSM_and_D_B_dHS_mem_clock_ex is
-- outputs
signal Out1:  std_logic_vector (31 downto 0);
signal Out1_Stop: std_logic;

signal Data_MB: std_logic_vector (31 downto 0);
signal Data_MB_Stop: std_logic;

signal DP_Reset:    std_logic;
signal ld_A, ld_B, ld_Stop: std_logic;
signal StopMsg: std_logic;
signal ALU_M: std_logic_vector (1 downto 0);
signal ld_O, B_tri_en: std_logic;
signal B_ld_Addr, B_inc_Addr: std_logic;
signal CMP: std_logic_vector(1 downto 0);
signal MUX_sel: std_logic;

Component DP
   port (Clock:           in std_logic;
         DP_Reset:        in std_logic;
         ld_A, ld_B, ld_Stop: in std_logic;
         StopMsg:         out std_logic;
         ALU_M:           in std_logic_vector (1 downto 0);
         ld_O, tri_en:    in std_logic;
         ld_Addr, inc_Addr: in std_logic;
         Addr_In:         in std_logic_vector(3 downto 0);
         Addr_Out_Port:   out std_logic_vector(3 downto 0);
         In1, In2:        in std_logic_vector(31 downto 0);
         In_Stop:         in std_logic;
         O_Port:          out std_logic_vector(31 downto 0);
         Out_Stop:        out std_logic);
end Component;

Component FSM_B_dHS_mem_clock_ex
   port (Reset, B_Start:          in std_logic;
         Clock_B:               in std_logic;
         AC_Grant_B, TB_Ackn_B: in std_logic;
         B_Req_AC, B_Ready_TB:  out std_logic;
```

```
          B_nCS, B_nWE, B_nOE:  out std_logic;
          DP_Reset:             out std_logic;
          ld_A, ld_B, ld_Stop:  out std_logic;
          StopMsg:              in std_logic;
          ALU_M:                out std_logic_vector (1 downto 0);
          ld_O, B_tri_en:       out std_logic;
          B_ld_Addr, B_inc_Addr:  out std_logic);
end Component;

begin -- Architecture
DP_B_I: DP
     Port Map ( Clock=>Clock_B, DP_Reset=>DP_Reset,
                ld_A=>ld_A, ld_B=>ld_B, ld_Stop=>ld_Stop,
                StopMsg=>StopMsg, ALU_M=>ALU_M, ld_O=>ld_O, tri_en=>B_tri_en,
                ld_Addr=>B_ld_Addr, inc_Addr=>B_inc_Addr,
                Addr_In=>Addr_In, Addr_Out_Port=>Addr_Out,
                In1=>Data_MB, In2=>Data_MB,
                In_Stop=>Data_MB_Stop,
                O_Port=>Out1, Out_Stop=>Out1_Stop);

FSM_B_dHS_mem_clock_I: FSM_B_dHS_mem_clock_ex
     Port Map ( Reset=>Reset, B_Start=>B_Start,
                Clock_B=>Clock_B,
                AC_Grant_B=>AC_Grant_B, TB_Ackn_B=>TB_Ackn_B,
                B_Req_AC=>B_Req_AC, B_Ready_TB=>B_Ready_TB,
                B_nCS=>B_nCS, B_nWE=>B_nWE, B_nOE=>B_nOE,
                DP_Reset=>DP_Reset,
                ld_A=>ld_A, ld_B=>ld_B, ld_Stop=>ld_Stop,
                StopMsg=>StopMsg, ALU_M=>ALU_M, ld_O=>ld_O, B_tri_en=>B_tri_en,
                B_ld_Addr=>B_ld_Addr, B_inc_Addr=>B_inc_Addr);

-- Entity Outputs
Out1_Port <= Out1;
Out1_Stop_Port <= Out1_Stop;

Data_MB <= Data_MB_and_Stop_Port(31 downto 0);
Data_MB_Stop <= Data_MB_and_Stop_Port(32);

end FSM_and_D_B_dHS_mem_clock_ex_behavioral;
```

C.4.8 FSM of B

VHDL/doubleHS_mem_SMs/FSM_dHS_mem_clock/FSM_dHS_mem_clock_ex_B.vhd

```
-- File :          FSM_dHS_mem_clock_ex_B.vhd
-- implements :    FSM B
-- of Model:       FSM and separate Datapath (FSM + D)
-- for Example:    FSM,DP (A) --> Memory --> FSM,DP (B)
--                               <-->   FSM <-->
--                 using double handshake protocol

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

Entity FSM_B_dHS_mem_clock_ex is
   port (Reset, B_Start:         in std_logic;
         Clock_B:              in std_logic;
         AC_Grant_B, TB_Ackn_B: in std_logic;
         B_Req_AC, B_Ready_TB: out std_logic;
         B_nCS, B_nWE, B_nOE:     out std_logic;
         DP_Reset:             out std_logic;
         ld_A, ld_B, ld_Stop:  out std_logic;
         StopMsg:              in std_logic;
         ALU_M:                out std_logic_vector (1 downto 0);
         ld_O, B_tri_en:           out std_logic;
         B_ld_Addr, B_inc_Addr:    out std_logic);
end FSM_B_dHS_mem_clock_ex;


Architecture FSM_B_dHS_mem_clock_ex_behavioral of FSM_B_dHS_mem_clock_ex is

type State_Set is (S_BEGIN, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9, S_10, S_END);
signal next_STATE, STATE: State_Set;

begin -- Architecture

transition: Process
begin
  wait until (Clock_B'event and Clock_B = '1');
  STATE <= next_STATE;
end process;

behavior1: process(STATE, B_Start, AC_Grant_B, TB_Ackn_B, StopMsg)
begin
    if (Reset = '1') then
         DP_Reset    <= '1';
         B_Req_AC    <= '0';
         B_Ready_TB  <= '0';
         ld_A        <= '0';
         ld_B        <= '0';
         ld_Stop     <= '0';
         ALU_M       <= "--";
         ld_O        <= '0';
         B_ld_Addr   <= '0';
         B_inc_Addr  <= '0';
         B_tri_en    <= '0';
         B_nCS   <= '1';
         B_nWE   <= 'Z';
         B_nOE   <= 'Z';
      next_STATE <= S_BEGIN;
    else
      case STATE is

        when S_BEGIN => DP_Reset    <= '1';
                        B_Req_AC    <= '0';
                        B_Ready_TB  <= '0';
```

```
                          ld_A        <= '0';
                          ld_B        <= '0';
                          ld_Stop     <= '0';
                          ALU_M       <= "--";
                          ld_O        <= '0';
                          B_ld_Addr   <= '0';
                          B_inc_Addr  <= '0';
                          B_tri_en    <= '0';
                          B_nCS   <= '1';
                          B_nWE   <= 'Z';
                          B_nOE   <= 'Z';

                          if B_Start = '1'
                            then next_STATE <= S_1;
                            else next_STATE <= S_BEGIN;
                          end if;

        when S_1 =>      DP_Reset    <= '0';
                         B_Req_AC    <= '1';
                         B_Ready_TB  <= '0';
                         ld_A        <= '0';
                         ld_B        <= '0';
                         ld_Stop     <= '0';
                         ALU_M       <= "--";
                         ld_O        <= '0';
                         B_ld_Addr   <= '1';
                         B_inc_Addr  <= '0';
                         B_tri_en    <= '0';
                         B_nCS   <= '1';
                         B_nWE   <= 'Z';
                         B_nOE   <= 'Z';

                       if (AC_Grant_B='1')
                         then next_STATE<= S_2;
                         else next_STATE<= S_1;
                       end if;

        when S_2 =>     DP_Reset    <= '0';
                        B_Req_AC    <= '1';
                        B_Ready_TB  <= '0';
                        ld_A        <= '0';
                        ld_B        <= '0';
                        ld_Stop     <= '0';
                        ALU_M       <= "--";
                        ld_O        <= '0';
                        B_ld_Addr   <= '0';
                        B_inc_Addr  <= '0';
                        B_tri_en    <= '1';
                        B_nCS   <= '0';
                        B_nWE   <= '1';
                        B_nOE   <= '1';

                        next_STATE<= S_3;

        when S_3 =>     DP_Reset    <= '0';
                        B_Req_AC    <= '1';
                        B_Ready_TB  <= '0';
                        ld_A        <= '0';
                        ld_B        <= '0';
                        ld_Stop     <= '0';
                        ALU_M       <= "--";
                        ld_O        <= '0';
                        B_ld_Addr   <= '0';
                        B_inc_Addr  <= '0';
                        B_tri_en    <= '1';
                        B_nCS   <= '0';
                        B_nWE  <= '1';
                        B_nOE  <= '0';

                        next_STATE<= S_4;

        when S_4 =>     DP_Reset    <= '0';
                        B_Req_AC    <= '1';
                        B_Ready_TB  <= '0';
                        ld_A        <= '1';
                        ld_B        <= '1';
                        ld_Stop     <= '1';
                        ALU_M       <= "10";
                        ld_O        <= '0';
                        B_ld_Addr   <= '0';
                        B_inc_Addr  <= '0';
                        B_tri_en    <= '1';
                        B_nCS  <= '0';
                        B_nWE  <= '1';
                        B_nOE  <= '0';

                        next_STATE<= S_5;

        when S_5 =>     DP_Reset    <= '0';
                        B_Req_AC    <= '1';
                        B_Ready_TB  <= '0';
                        ld_A        <= '0';
                        ld_B        <= '0';
                        ld_Stop     <= '0';
                        ALU_M       <= "10";
                        ld_O        <= '1';
                        B_ld_Addr   <= '0';
                        B_inc_Addr  <= '0';
                        B_tri_en    <= '1';
                        B_nCS  <= '0';
                        B_nWE  <= '1';
                        B_nOE  <= '1';

                        next_STATE<= S_6;

        when S_6 =>     DP_Reset    <= '0';
                        B_Req_AC    <= '1';
                        B_Ready_TB  <= '0';
                        ld_A        <= '0';
                        ld_B        <= '0';
                        ld_Stop     <= '0';
                        ALU_M       <= "--";
                        ld_O        <= '0';
                        B_ld_Addr   <= '0';
                        B_inc_Addr  <= '0';
                        B_tri_en    <= '1';
                        B_nCS  <= '0';
                        B_nWE  <= '1';
                        B_nOE  <= '1';

                        next_STATE<= S_7;

        when S_7 =>     DP_Reset    <= '0';
                        B_Req_AC    <= '1';
                        B_Ready_TB  <= '1';
                        ld_A        <= '0';
                        ld_B        <= '0';
                        ld_Stop     <= '0';
                        ALU_M       <= "--";
                        ld_O        <= '0';
                        B_ld_Addr   <= '0';
                        B_inc_Addr  <= '0';
                        B_tri_en    <= '1';
                        B_nCS  <= '0';
                        B_nWE  <= '1';
                        B_nOE  <= '1';

                        if (TB_Ackn_B='1')
                          then next_STATE<=S_8;
                          else next_STATE<=S_7;
                        end if;

        when S_8 =>     DP_Reset    <= '0';
                        B_Req_AC    <= '1';
                        B_Ready_TB  <= '0';
                        ld_A        <= '0';
                        ld_B        <= '0';
                        ld_Stop     <= '0';
                        ALU_M       <= "--";
                        ld_O        <= '0';
                        B_ld_Addr   <= '0';
                        B_inc_Addr  <= '0';
                        B_tri_en    <= '1';
                        B_nCS  <= '0';
                        B_nWE  <= '1';
                        B_nOE  <= '1';

                        if (TB_Ackn_B='0')
                          then next_STATE<=S_9;
                          else next_STATE<=S_8;
                        end if;

        when S_9 =>     DP_Reset    <= '0';
                        B_Req_AC    <= '1';
                        B_Ready_TB  <= '0';
                        ld_A        <= '0';
                        ld_B        <= '0';
                        ld_Stop     <= '0';
                        ALU_M       <= "--";
                        ld_O        <= '0';
                        B_ld_Addr   <= '0';
                        B_inc_Addr  <= '1';
                        B_tri_en    <= '1';
                        B_nCS  <= '0';
                        B_nWE  <= '1';
                        B_nOE  <= '1';

                        if (StopMsg='1')
                          then next_STATE<=S_10;
                          else next_STATE<=S_2;
                        end if;

        when S_10 =>    DP_Reset    <= '0';
                        B_Req_AC    <= '0';
                        B_Ready_TB  <= '0';
                        ld_A        <= '0';
                        ld_B        <= '0';
                        ld_Stop     <= '0';
                        ALU_M       <= "--";
                        ld_O        <= '0';
                        B_ld_Addr   <= '0';
                        B_inc_Addr  <= '0';
                        B_tri_en    <= '0';
                        B_nCS  <= '1';
                        B_nWE  <= 'Z';
                        B_nOE  <= 'Z';

                        if (AC_Grant_B='0')
```

```
                                then next_STATE<=S_END;
                                else next_STATE<=S_10;
                              end if;

              when S_END=>  -- nothing (Procedure quits)

          end case;
        end if;

    end process;

    end FSM_B_dHS_mem_clock_ex_behavioral;
```

C.4.9 Datapath of B identical to A (see A)

The datapath for the receiver B is identical to the datapath of the sender A. → See datapath for sender A.