

Handling Exception Events in Behavioral VHDL Models

Janick Bergeron

AnalySYS Inc
25 Loiselle St, Suite 201, Embrun
Ontario, Canada, K0A 1W1
janick@analysys.com

Mario Dufresne

Bell-Northern Research Ltd
P.O. Box 3511, Station C, Ottawa
Ontario, Canada, K1Y 4H7
dufresne@bnr.ca

Abstract

This paper addresses the problems presented by the introduction of handling exception events in a behavioral VHDL model. The most common exception event is the hardware reset but similar problems and issues are encountered when introducing any asynchronous exceptions like interrupt service requests.

It will present various techniques for handling exception events in behavioral VHDL models, using reset as an example, and discuss their respective merits. It then looks at the solution available for behavioral models in Verilog and ADA programs then proposes similar mechanisms for VHDL.

Section 1. Introduction

In RTL models, the handling of reset is well established in the synthesizable subset of VHDL. The state of an RTL model being fully described by the inferred latches and flip-flops, resetting such a model is a simple matter of setting these inferred latches and flip-flops to a known state.

Figure 1 shows the style used in synthesizable VHDL to specify both a

synchronous and an asynchronous reset to inferred flip-flops while Figure 2 shows the equivalent style in synthesizable Verilog.

```
process (CLK, ASYNC)
begin
  if ASYNC = '1' then
    Q <= '0';
  elsif CLK'event and CLK = '1' then
    if SYNC = '1' then
      Q <= '0';
    else
      Q <= ...;
    end if;
  end if;
end process;
```

Figure 1. Resetting inferred flip-flops in synthesizable VHDL

```
always @ (posedge CLK or
         posedge ASYNC)
begin
  if (ASYNC == 1'b1)
    Q <= 1'b0;
  else if (CLK == 1'b1) begin
    if (SYNC == 1'b1) Q <= 1'b0;
    else
      Q <= ...;
    end
  end
end
```

Figure 2. Resetting inferred flip-flops in synthesizable Verilog

Resetting a model at this level of abstraction is performed in an efficient and

readable fashion and neither language offers a better solution. Handling other exception conditions at the RTL level is done through a similar process, with all state values being either saved, restored or set to a known value.

Architectural models, on the other hand, concern themselves with the processing of data and usually ignore the low-level exception conditions such as interrupts or reset signals.

However, the models at an abstraction level situated between the architectural and synthesizable levels, which must faithfully reproduce the complete behavior of the modelled device, have to respond to interrupt or reset signals. These models are typically used in board or system-level simulation in a prototype environment as drop-in replacements for the RTL model [1].

It is very important that the behavior of a component subjected to a reset condition be modelled accurately. Bringing a system back into a fully-functional state after a hardware reset is often a non-trivial task and it is often the first system-level simulation to be carried out. For example, in an ATM switching system, a normal flow of cells must be present in order to program the switch. Making sure that this normal flow establishes itself after a reset condition has occurred is a crucial step in its design.

In systems where real-time response is a critical characteristic, the time required to service an interrupt must be as close as possible to the actual implementation. It may be required that all normal processing activities be suspended at a level of granularity similar to the instruction-level offered by processors, not simply when it is convenient for the model to process interrupts.

Section 2. Detecting Exceptions

Since, in VHDL, processes cannot be affected by other processes other than through signals, each process must detect

any exception condition that applies to it. This implies that exceptions cannot be handled other than when a process is currently suspended on a wait statement or by explicitly checking if they are present.

In order to guarantee detection in all possible states of a process, every wait statement in the process in question must be explicitly sensitized to the exception signals and the exception conditions.

```
wait on ... until ... for ...;

wait on ..., RST
      until (...) or RST = '1'
      for ...;
```

Figure 3. Modification of a wait statement to detect reset

To catch the exception condition which may have caused the wait statement to complete, it is necessary to insert an if statement to detect the exception condition once more then act properly if it is indeed asserted.

```
wait on ..., RST
      until (...) or RST = '1'
      for ...;

if RST = '1' then
  ...
end if;
```

Figure 4. Handling of a reset condition

This modification of all wait statements in all processes of a model can be a significant task and definitely introduces clutter and reduces readability. Maintainability is also reduced since the exception conditions must be replicated many times. All the copies must be kept up-to-date whenever something as simple as the reset polarity changes.

An alternative modification may be applicable if all the wait statements use an identical expression and sensitivity list except for the signals used in each instances such as shown in Figure 5.

```
wait until CK1'event and CK1 = '1';
...
wait until CK2'event and CK2 = '1';
```

Figure 5. wait statements using identical expressions

These wait statements could be replaced by a call to a procedure which would monitor the exception condition internally and return a flag if the exception condition has occurred. Such a method does not impact readability as much and greatly improves maintainability, should the actual exception condition change, since only one section of the code will need to be modified.

```
procedure WAIT_RISING_EDGE(
  signal CK: in STD_LOGIC;
         RST: out BOOLEAN) is
begin
  wait until (CK'event and CK = '1')
            or RESET = '1';
  RST = RESET = '1';
end WAIT_RISING_EDGE
...
WAIT_RISING_EDGE(CK1, RST);
if RST then ...
...
WAIT_RISING_EDGE(CK2, RST);
if RST then ...
```

Figure 6. wait statements through a procedure

Some processes, such as those that model the software behavior, may use bus-functional models or access procedures and do not contain any explicit wait statements which can be modified to accommodate the exception detection. In such cases, polling must be used to detect if any exception condition has occurred and take appropriate action.

Figure 7 shows a software process using a PCI bus-functional model. To properly detect interrupts, the exception must be polled frequently enough to provide a granularity level similar to an instruction processor.

Section 3. Signalling Exceptions

An exception condition may be caused by several sources. In large ASICs, it is typical

```
process begin
  PCI.MEM_READ(...);
  if INTERRUPT = '1' then ...
  ...
  PCI.IO_WRITE(...);
  if INTERRUPT = '1' then ...
  ...
end process;
```

Figure 7. Polling exception conditions

to have a hardware reset signal coming from an external pin and a software reset activated by a processor-accessible register.

Figure 8 shows the additional VHDL code required to detect and handle a multiple reset condition. Since, in most cases, the reset behavior will be identical regardless of its source, code readability and maintainability would be greatly improved by creating a single internal reset signal raised by anyone of the possible sources.

```
wait on ..., HWRST, SWRST
  until ... or HWRST = '1'
         or SWRST = '1';

if HWRST = '1' or SWRST = '1' then
  ...
end if;
```

Figure 8. Detection and handling of multiple reset conditions

Furthermore, by making this internal reset signal of type BOOLEAN, the detection and handling expressions find themselves greatly simplified as shown in Figure 9.

```
signal RESET: BOOLEAN;
RESET <= HWRST = '1' or SWRST = '1';

wait on ..., RESET
  until ... or RESET;

if RESET then
  ...
end if;
```

Figure 9. Using a single reset signal

Section 4. Handling of Exceptions

There are several alternative methods to handle an exception condition once it has

been detected. The challenge is to make this handling as unobtrusive as possible in order not to reduce the readability and maintainability of the model.

The first method would be to faithfully adhere to the structured programming principles and gracefully exit of the process control structure.

```

process begin
  wait ...;
  BLOCK_OF_STATEMENT1;
  wait ...;
  BLOCK_OF_STATEMENT2;
  ... loop
    wait ...;
    BLOCK_OF_STATEMENT3;
  end loop;
  BLOCK_OF_STATEMENT4;
end process;

```

Figure 10. Original control structure

```

process begin
  wait ...;
  if not RESET then
    BLOCK_OF_STATEMENT1;
    wait ...;
    if not RESET then
      BLOCK_OF_STATEMENT2;
      ... loop
        wait ...;
        exit when RESET;
        BLOCK_OF_STATEMENT3;
      end loop;
      if not RESET then
        BLOCK_OF_STATEMENT4;
      end if;
    end if;
  end if;
  if RESET then
    ...
  end if;
end process;

```

Figure 11. Modified control structure

Figure 11 shows that such a modified structure quickly becomes hard to understand and to later modify without introducing bugs.

If there are only one or two wait statements imbedded in a deep control structure, it may be possible to keep all the state information in a set of variables, turn the control structure inside-out and use a

single wait statement at the top of the process.

```

process begin
  ...
  ILOOP: for I in 0 to 7 loop
    for J in 0 to 7 loop
      ...
      wait ...;
      exit ILOOP when RESET;
      ...
    end loop;
  end loop;
  if not RESET then
    ...
  end if;
end process;

```

Figure 12. Single wait statement in deep control structure

This methods, as illustrated in Figure 13, involves a significant modification of the original code and is only applicable to specific cases.

```

process
  variable I: INTEGER := 0;
  variable J: INTEGER := 0;
begin
  if I = 0 and J = 0 then
    ...
  end if;
  ...
  wait ...;
  if RESET then
    ...
  else
    ...
    I := I + 1 rem 8;
    J := J + 1 rem 8;
    if I = 8 and J = 8 then
      ...
    end if;
  end if;
end process;

```

Figure 13. Explicit state and single wait statement

The method found most useful and the least obtrusive by the authors is to introduce an explicit loop and the outset of the process which handles the normal processing. This outer loop can be easily exited from whenever an exception condition occurs and the implicit process loop is used to perform the exception

processing.

```
process begin
RESET_PROCESSING_STATEMENTS;
OUTER: loop
  wait ...;
  exit OUTER when RESET;
  BLOCK_OF_STATEMENT1;
  wait ...;
  exit OUTER when RESET;
  BLOCK_OF_STATEMENT2;
  ... loop
    wait ...;
    exit OUTER when RESET;
    BLOCK_OF_STATEMENT3;
  end loop;
  BLOCK_OF_STATEMENT4;
end loop OUTER;
end process;
```

Figure 14. Using explicit outer loop

The style shown in Figure 14, although not applicable to processes with sensitivity lists, further ensures that processes are started in a proper state at initialization.

Section 5. Exception Detection and Handling in Verilog

The disable statement in Verilog has the power of turning the hair of any structured programming language purist snowy white.

In its most benign use, it replaces VHDL's exit and next statements for loop control. It is a required evil to force the reconvergence of fork/join statements. But its probable *raison d'être* and most powerful usage is to force a new iteration of always blocks (Verilog's equivalent of a process statement) when an exception condition occurs.

In Verilog, sequential statements can be grouped into a single logical sequential statement using a Pascal-like begin/end construct. These begin/end blocks can be named to allow the declaration of local variables. The disable statement forces the termination of the named begin/end block and execution will resume at the statement following the end of the block. If the begin/end block in question is the body of an always block, this effectively causes the

always block to perform a new iteration.

```
always @ (...)
begin: LABEL
  ...
end

always @ (RST)
begin
  if (RST == 1'b1) disable LABEL;
end
```

Figure 15. disable statement in Verilog

Figure 15 illustrates how reset can be detected by a single always block and handled through the disable statement.

Section 6. Exception Detection and Handling in ADA

ADA has a more structured way of handling exception conditions than Verilog. It offers a new kind of declaration: exception. Users can thus add to the list of pre-defined exceptions by declaring new ones just as easily as declaring a variable. The raise statement is then use to explicitly raise any exception.

Any block of sequential statements can have an exception handler, similar to a case statement. When an exception is raised, execution of the sequential block is interrupted and resumes at the top of the exception handler section for that block.

```
OVERFLOW: exception;

begin
  ...
  ... raise OVERFLOW;
  ...
exception
  when OVERFLOW => ...
  when others => ...
end
```

Figure 16. Exception handling in ADA

Section 7. Suggestions for VHDL-98

Since there is an implicit loop around every process, it would be very useful if the next statement could be used to force a re-

initialization of the process.

Just like in its current use, a next statement without a label would cause a new iteration of the innermost loop, including the implicit process loop. It could be optionally followed by the process label to explicitly cause the next iteration of the process if embedded in explicit loops in the body of the process.

```
PLABEL: process begin
...
wait ...;
next PLABEL when RESET;
...
end process PLABEL;
```

Figure 17. next statement applied to implicit process loop

This previous suggestion would help in reducing the clutter introduced by the exception handling in the control structure but would not provide any significant improvement on the detection and handling of the reset proper.

A new disable statement, which would cause the named process to re-initialize, would greatly facilitate the modelling and handling of reset conditions.

```
PLABEL: process begin
...
wait ...;
...
end process PLABEL;

process (RST)
begin
  if RST = '1' then
    disable PLABEL;
  end if;
end process;
```

Figure 18. New disable statement

A modelling style, as illustrated in Figure 18, presents a minimum clutter level and maximizes maintainability by concentrating all reset detection to a single process. It also greatly facilitates the task of evolving an architectural model into a full functional behavioral model by not requiring any or little modification of the original code.

An exception handling mechanism, similar to the one available in ADA, might provide a cleaner language solution, although the parallel execution nature of VHDL will undoubtedly necessitate different semantics. It would also have a greater impact on the language itself by requiring more new reserved words and a new declaration.

```
PLABEL: process begin
...
wait ...;
...
exception
  when RESET => ...
end process PLABEL;

process (RST)
begin
  if RST = '1' then
    raise RESET;
  end if;
end process;
```

Figure 19. New exception handler

A modelling style, as illustrated in Figure 19, reduces clutter level and maximizes maintainability by concentrating all reset detection to a single process. More modifications to the original code is required when adding exception handling to a behavioral model than would be necessary using a disable statement but offers a better specification of the exception handling behavior.

It would be left to the language lawyers whether or not any pending assignment on the process's drivers would be removed when it is interrupted by a disable statement or by a raised exception.

Section 8. References

- [1] Allan Silburt and al., "Accelerating Concurrent Hardware Design with Behavioral Modeling and System Simulation". Design Automation Conference, April 1995
- [2] Karl A. Nyberg (Editor), "The Annotated Ada Reference Manual". 2nd Edition