Modeling with SpecCharts

Sanjiv Narayan
Frank Vahid
Daniel D. Gajski

Technical Report #90-20
July 25, 1990
Revised October 15, 1992

Dept. of Information and Computer Science
University of California, Irvine
Irvine, CA 92717
(714) 856-8059

narayan@ics.uci.edu
vahid@ics.uci.edu

Abstract

SpecCharts is a language intended for system level specification and synthesis. Its blending of hierarchical state-diagrams with programming constructs permits description of event-driven systems as a true hierarchy of behaviors, unlike existing languages. Through many examples, we show in this report how a language with an underlying model of behavioral hierarchy greatly aids concise and comprehensible specifications obtained with minimal effort.

Contents

List of Figures

1 Introduction

There is a need for a language that can specify the desired behavior of a system without indicating a structural design. Such a specification can serve as documentation, as a source for early system simulations, or as the starting point of synthesis. Current languages are insufficient since they require a designer to coerce a conceptualized system model into a rather different model provided by the language (e.g. concurrent processes). This in turn requires several design decisions to be made and often results in complex and confusing specifications.

The underlying model of the specification language should closely match the model with which designers conceptualize a system. The lack of such a language was one of the motivating factors for the development of the SpecCharts language [1, 2]. The SpecCharts language was developed by attempting to specify many real systems in the most natural possible manner, creating new constructs as they were needed. We found that a system's behavior, without any consideration of hardware, is commonly conceptualized as a *hierarchy of sequential and concurrent behaviors*.

We shall briefly summarize aspects of SpecCharts that are relevant to this report. The basic object in a SpecChart is a *behavior*. We found that there are three manners that are desirable for describing behaviors. (1) Some behaviors can be described as a simple program using sequential programming constructs, such as loop, if-then-else, assignment, and procedure call statements. In SpecCharts, VHDL sequential statements are used for such behaviors. (2) A behavior can be recursively defined as sequential subbehaviors that are sequenced by transition arcs, much like a state-transition diagram. There are two basic types of transitions that must be supported: (a) Transition immediately (TI) when the condition on the arc becomes true (an event). (b) Transition on completion (TOC) of the subbehavior from which the arc originates, if the arc condition is true. Note that the latter requires the concept of a behavior "completing". (3) A behavior can be composed of concurrent subbehaviors. Any behavior can contain declarations of storage, wires, types, functions, etc., with a scope of all descendant behaviors. SpecCharts use VHDL declarations for this purpose.

The above provides for what we call *behavioral hierarchy* [1]. Briefly, we say that if a language permits (1) behavioral decomposition (the ability to recursively define a behavior as sequential or concurrent subbehaviors), (2) hierarchical activation/deactivation (the ability of a behavior to activate/deactivate a subbehavior at any time) and (3) behavioral completion (the ability of a behavior to complete and of a parent behavior to detect this completion), then this language supports behavioral hierarchy. Behavioral hierarchy is a formalization of the conceptual model which we found was inherent in the designs that we modeled.

The underlying model of SpecCharts is one of behavioral hierarchy. Current languages have differing models. For example, most hardware description languages (such as VHDL or Verilog) have a model of concurrent behaviors, or processes. This is suitable for modeling *existing hardware*, since each module can be thought of as a process. However, processes cannot be decomposed, nor can they be deactivated simply (e.g. reset), nor is their completion defined.

Another example language is Statecharts [3]. The language is essentially a hierarchy of sequential and concurrent *finite-state machines* (FSMs). There is no "behavior" object in Statecharts, but instead "states". Simple "zero-time" behaviors (e.g. variable assignment) can occur when entering states, when leaving states, or when transitioning between states. This does not correspond to the conceptual behavior model we described above. In addition, the lack of programming constructs is a serious deficiency, as shall be seen in this report. Adding such constructs to FSMs is not a simple step, but instead requires major changes in the language. Such changes will lead one to a language which closely resembles SpecCharts.

Communicating Sequential Processes (CSP) [4] is another proposed specification language. However, it lacks the ability to deactivate a subbehavior since it uses only sequential statements (process activation is one such statement) rather than transition arcs. The ability to deactivate a behavior is *crucial* in system specification. In addition, state-transitions are a common conceptualization which are missing in CSP. These issues will be seen in many examples of this report.

While using SpecCharts simplifies specification, the fact remains that simulators, synthesizers, and various other design tools are available for more common languages such as the widely accepted VHDL standard. Rather than requiring the designer to use VHDL, we add translators which automatically convert SpecCharts to simulatable or synthesizable VHDL [5, 6, 7].

Not included in this report are several aspects of SpecCharts which greatly aid synthesis, i.e. the refining of a specification by the addition of structural details, such as chip-partitioning and communication structure. These aspects are extremely important when considering specification languages intended for synthesis, but are omitted since the purpose of this report is to focus on the modeling power of the language.

The examples in this report range from SpecCharts' lowest abstraction domain of description of hardware (i.e. an existing counter) to the description of abstract system behavior with no notion of hardware (a desired answering machine). Unfortunately, we haven't obtained large examples with much concurrency (e.g. a robot controller) which would be the highest abstraction domain.

For each example in this report we shall provide the following:

- *Source and Brief Description*: the origin and a brief description of the example in a few sentences.
- *English Specification*, in some cases being a partial specification since a complete one would require excess text or is simply too complex.
- *SpecChart Specification*, along with a description of the SpecChart.
- *Analysis*: a discussion of the modeling power of SpecCharts as demonstrated with the example, especially as compared to HDLs and Statecharts.

2 A Controlled Counter

Although SpecCharts are intended to specify complex systems at an abstract behavioral level, this example demonstrates the language's ability to model at the hardware level.

2.1 Source and Brief Description

The counter was described in [8], a book on VHDL modeling. It is a simple up/down counter with a variable count limit.

2.2 English Specification

The Controlled Counter can count up or down on each rising clock, until a specified limit is reached, and can be asynchronously cleared. It can be thought of as consisting of 3 main components (see figure 1). The value CON stored in CONREG, when decoded, determines the actions carried out by the counter (i.e. count up, count down, load the limit register LIM and clear counter). LIM is a register whose value is used by the counter to determine when to terminate counting (when the limit has been reached). COUNTER is the component that actually contains the current count value and performs the incrementing or decrementing based on CONREG, LIM, and the clock input. The output of this register becomes the controlled counter's count output.

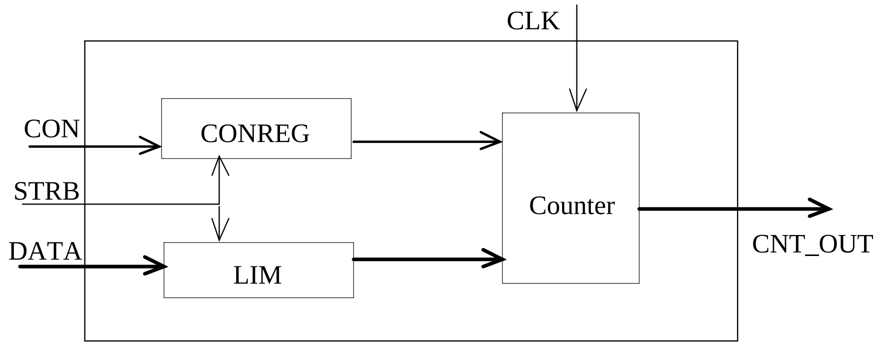


Figure 1: Block diagram of the Controlled Counter

When the STRB (strobe) input is rising, CON is stored. The controlled counter will then perform one of the following actions based on the stored value of CON:

- "00" : Clear (current count becomes 0)
- "01" : Load the limit with the value on the DATA input line on the falling edge of STRB
- "10" : Count up on rising clock unless limit is reached
- "11" : Count down on rising clock unless limit is reached

Note that any sequence and combination of count ups, count downs, limit loads, and clears are allowed, permitting rather strange sequences such as loading a limit of 7, counting from 0 to 3, changing the limit to 1, and then counting from 3 on up. An alternative design might require that the limit be reached before a new limit can be loaded; this might capture the true intention of usage and would result in a much simpler specification. However this design is not considered here as our purpose is to model the same design as in [8].

Several timing details of the counter's operation are included in the SpecChart. See [8, 9] for a description of those details and for another description of the controlled counter's operation.

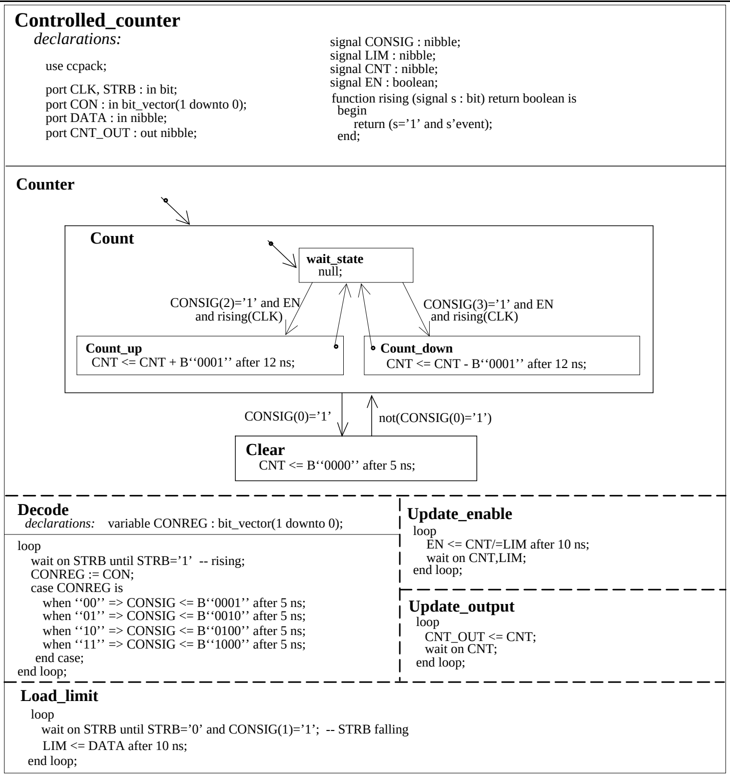


Figure 2: SpecChart of the Controlled Counter

2.3 SpecChart Specification

Figure 2 shows the SpecChart specification of the controlled counter. The description above gave three main functions that must be performed: loading and decoding CON, loading LIM, and counting. There are two other functions needed: updating the output ports to reflect the internal count value, and generating an internal enable signal which disables counting if the limit has been reached. This implies the controlled counter can be modeled by five concurrent subbehaviors, one for each function. Each of the five behaviors is now briefly discussed.

- *Decode*– Loads the condition register CONREG with CON, then generates a decoded signal CONSIG. It will do this whenever STRB is rising.
- *Load_limit*– Loads the limit register LIM when CONSIG is "0010" AND STRB is falling. The wait statement used is *wait on STRB until STRB='0' and CONSIG(1)='1'*. There is a VHDL intricacy to note here. The *on STRB* can not be omitted, since its omission implies *on STRB,CONSIG*, which is incorrect because a change on CONSIG should not trigger the loading of LIM.
- *Update_enable*– Generates an enable signal EN which disables counting if the current count value equals the limit. The code waits on a change of the count or the limit, which are the only two things that could change EN. The wait statement occurs at the end of the loop so that the EN signal is initially generated. This is how a concurrent signal assignment is done with SpecCharts, which only permits sequential code. Note that if there was no delay involved with the EN signal as there is now, then the EN signal would always be equal to $CNT/=LIM$ and thus could be replaced by this expression. Note that we *have* replaced the ENIT and CNT_CLR signals used in the [8, 9] VHDL models, since they do not simplify the SpecChart description.

Counter

```
loop
    wait until CONSIG(0)='1' or CLK='1';
    if CONSIG(0)='1' then
        CNT <= B“0000” after 5 ns;
    elsif CONSIG(2)='1' and EN then
        CNT <= CNT + B“0001” after 12 ns;
    elsif CONSIG(3)='1' and EN then
        CNT <= CNT - B“0001” after 12 ns;
    end if;
end loop;
```

Figure 3: Alternative SpecChart description of the Counter behavior

- *Update_output*– Ensures that the controlled counter's output CNT_OUT always reflects the internal count CNT (essentially a concurrent signal assignment).
- *Counter*– This is the most interesting behavior as it performs the actual counting. Counter can be thought of as always being in one of two states, either performing count operations or performing an asynchronous clear. Thus, it consists of two sequential subbehaviors, *Count* and *Clear*. The default initial behavior is *Count*; if at any time CONSIG is “0001”, CNT must be immediately cleared (actually after a small delay). This is accomplished by a TI arc flowing to *Clear*, which resets the count value to 0. The 'after 5 ns' clause in *Clear* not only delays the clearing for 5 ns, but also creates a 5 ns hold time for the clear to occur (otherwise by definition of a TI arc CNT would not get updated).

  When *Count* is active, it can be in one of three states, either counting up, counting down, or waiting for the next count operation. Note that *wait_state* demonstrates one use of a null behavior, a common occurrence in SpecCharts.

From the above description, it is clear that the signals CONSIG, LIM, CNT, and EN must be declared somewhere. They are declared along with the ports in the topmost behavior, *Controlled_counter*, whose declarations are global over all subbehaviors. The declarations assume the existence of a *nibble* subtype, declared as bit_vector(3 downto 0) in package *ccpack*.

2.3.1 Alternative Specification

A model was also developed which was identical to that given above except for the Counter behavior. The Counter behavior's functionality was described with code, rather than with sequential subbehaviors. The code was similar to that found in block CNT_UP_OR_DOWN in [8, 9] (see figure 3 for Counter's code). This resulted in less SpecChart code, but we feel using sequential subbehaviors and arcs enhances the understandability of the model.

2.4 Analysis

SpecCharts is not intended as another HDL. HDL constructs are sufficient to describe the controlled counter. The only advantage of using SpecCharts is the comprehension improvement of the state-transition based description as opposed to using only programming constructs. Here, SpecCharts can enhance documentation or serve as a convenient front-end to VHDL.

3 The DRACO Peripheral Interface

3.1 Source and Brief Description

This example is the 1781 discrete I/O backplane custom integrated circuit, DRACO, obtained from the Rockwell Corporation. The behavior of DRACO was determined from the data sheet [10], from the designing engineer, and from the production test vectors. DRACO interfaces 16 I/O ports to a microprocessor's 8-bit multiplexed address/data bus and control signals, with several security and error-checking features.

3.2 English Specification

Due to the proprietary nature of this example, we can only discuss general aspects of its behavior. However, we have modeled DRACO in complete detail, using the production test vectors to verify our SpecChart description.

A block diagram of the DRACO chip is shown in Figure 4.

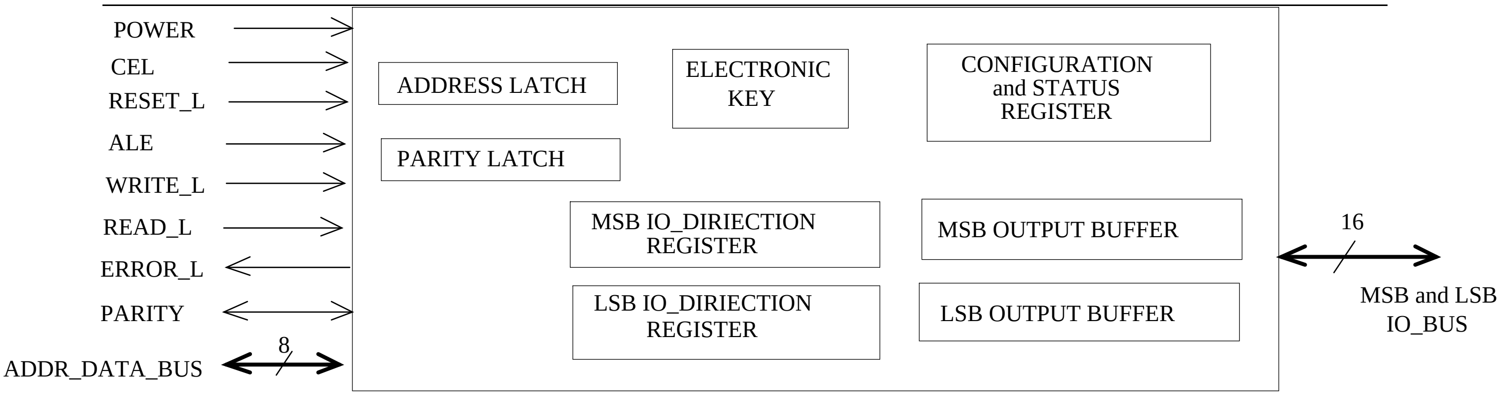


Figure 4: Block Diagram of DRACO

Upon power-up, DRACO initializes the values of several internal registers. DRACO then waits until it is instructed to perform an 8-bit address and parity-bit latch, a read, a write, or a reset.

A write first examines the latched address for a parity error if the address parity checking option has previously been enabled. Next, if the data parity checking option of DRACO is enabled, the parity of the data byte received is checked. Writes to a particular sequence of addresses will change the state of an internal key from "locked" to "unlocked". The key must be unlocked for any other writing or reading to take place. A write can be to an internal configuration register (which select error checking operations). Other addresses specify writes to one of two internal byte registers. Error checking can be done by using parity or by using a checksum. Parity errors, checksum errors, and invalid address errors all generate an error signal.

A read also first examines the address parity. Read addresses include: the checksum of the high and low bytes of the output buffer, the high byte or low byte of the output buffer, the internal status register, or the high or low byte of the 16-bit I/O port. A read will output the appropriate parity. Parity error on a read address or a read from an invalid address generates an error signal.

Further details of DRACO operation can be found in [10].

3.3 SpecChart Specification

The SpecChart is essentially a state-transition diagram, where each state is the power-up, reset, address-latch, read or write behavior. The details of each behavior are specified with VHDL sequential statements. Although we can't show the VHDL, we will mention that the read behavior contains 71 lines of VHDL, and the write behavior 184 lines. Each is essentially a set of irregularly nested if-then-else and case statements resulting in simple variable or port assignment statements. The other three behaviors contained less than 10 lines each.

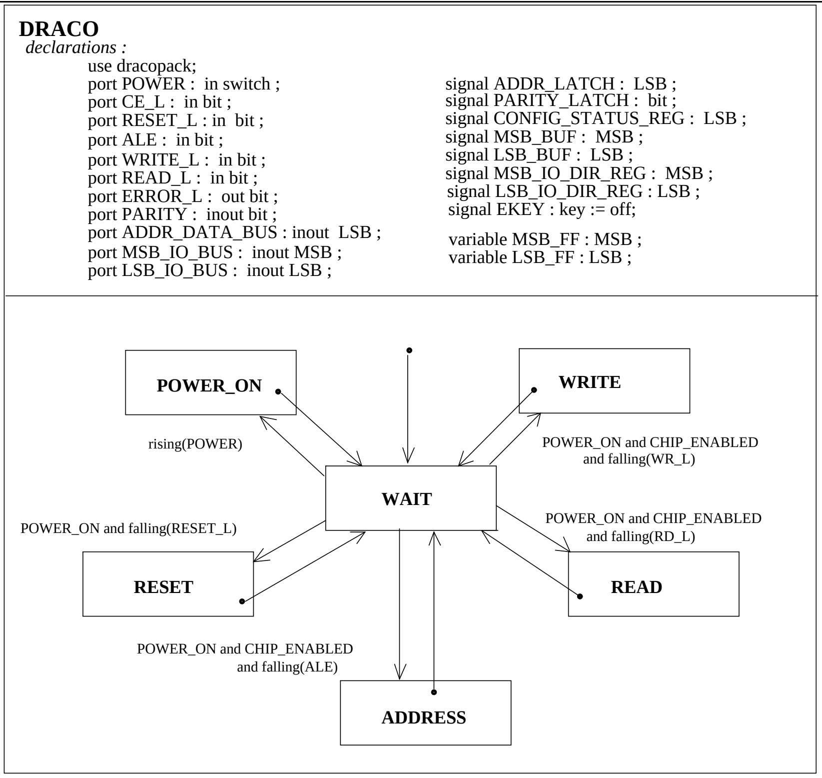


Figure 5: SpecChart of DRACO (top-level only)

3.4 Analysis

Due to the conditional branching nature of the read and write behaviors, nested if-then-else and case statements are the simplest constructs for capturing the behavior. Writing FSMs for these behaviors, as required in Statecharts, is tedious and makes the behavior difficult to comprehend.

SpecChart sequential subbehaviors are suitable for the state-transition relationship between the behaviors. Using VHDL procedures would require implementing state-transitions with programming constructs, such as a loop statement, case statement, and a state variable, as shown in Figure 6. This involves converting a two-dimensional state-transition diagram into a one-dimensional model of sequential programming constructs, which is cumbersome.

The blend of state-transition constructs and programming constructs makes the SpecChart specification the closest to the conceptual view of DRACO.

```
variable STATE_VAR : bit_vector(1 to 3);

loop
   case STATE_VAR is
     when "000" =>  -- DECODE
       wait until
       (
          (power='1' and not (power'stable))
          or (reset_l='0' and not (reset_l'stable) and power='1')
          or ((ale='0') and not(ale'stable) and (power='1') and (ce_l = '0'))
          or ((read_l = '0') and not(read_l'stable) and (power='1')
              and (ce_l='0'))
          or ((write_l='0') and not(write_l'stable) and (power='1')
              and (ce_l='0'))
       ) ;

       if (power='1' and not (power'stable)) then
         STATE_VAR := "001";
       elsif (reset_l='0' and not (reset_l'stable) and power='1') then
         STATE_VAR := "010";
       elsif ((ale='0') and not(ale'stable) and (power='1')
              and (ce_l='0')) then
         STATE_VAR := "011";
       elsif ((read_l='0') and not(read_l'stable)
               and (power='1') and (ce_l='0')) then
         STATE_VAR := "100";
       elsif ((write_l='0') and not(write_l'stable)
               and (power='1') and (ce_l='0')) then
         STATE_VAR := "101";
       end if;

     when "001" =>
       POWER_ON;
       STATE_VAR := "000";
     when "010" =>
       RESET;
       STATE_VAR := "000";
     when "011" =>
       ADDRESS;
       STATE_VAR := "000";
     when "100" =>
       READ;
       STATE_VAR := "000";
     when "101" =>
       WRITE;
       STATE_VAR := "000";
    when "110" =>
       STATE_VAR := "000";
    when "111" =>
       STATE_VAR := "000";
   end case;
end loop;
```

Figure 6: Specifiying DRACO's state-transitions with VHDL sequential statements is tedious

4 A Universal Asynchronous Receiver/Transmitter (UART)

4.1 Source and Brief Description

This example was provided by I-Logix Corporation at the Fall 1991 VHDL International User's Forum meeting in Newport Beach, California, in the form of a set of Statecharts [11]. The UART (Universal Asynchronous Receiver/Transmitter) converts a serial (or parallel) input of 8 bits and a parity bit to parallel (or serial) output.

4.2 English Specification

The UART interfaces an 8-bit processor with a 1-bit peripheral device. The UART has an 8-bit parallel input/output data port (DATA_IN) for communicating with the processor and a 1-bit serial input/output port (LINE) to communicate with the peripheral.

The UART has the capability of performing certain actions which depend on four externally specified bits - CSR(3), CSR(2), CSR(1) and CSR(0). These bits are set by the processor.

The processor can issue certain commands to the UART which are specified by bits CSR(0) and CSR(1) :

CSR(0)	CSR(1)	Command
0	0	——
0	1	Load 8-bit data into internal register from DATA_IN for serial transmission
1	0	Read 8-bit serial data which has been assembled in an internal register
1	1	Reset the UART, abandon all current transmitting/receive activities

Depending on bits CSR(2) and CSR(3), the UART can be in one of several *modes* :

CSR(2)	CSR(3)	Mode
0	0	Idle
0	1	Receive Mode : receive 8 bits of data serially
1	0	Transmit Mode : transmit 8 bits of data serially
1	1	Echo Mode : receive 8 bits of serial data and transmit it serially

In the *receive* mode, the UART waits until it detects a start bit (logic '1') on LINE, assembles the next 8 bits of serial data on LINE into an internal register and verifies that the following parity bit is correct for the byte received. If the parity is correct, it signals the processor which can then issue a read command (CSR(0..1) = '10') to access the received byte. In the receive mode, an odd parity is calculated for the data byte received.

In the *transmit* mode, the data byte that has to be transmitted is loaded into an internal register of the UART by issuing a load command (CSR(0..1) = '01'). A start bit (logic '1') is sent on LINE to start the serial transmission. This is followed by the transmission of the 8 bits serially and finally the parity bit. The parity bit generated for the transmitted data byte depends on an externally specified TX_PARITY_TYPE.

In the *echo* mode, the UART assembles a data byte received serially in a manner similar to that of the receive mode. The data byte received is then transmitted serially as is done for the transmit mode, except that the transmission commences automatically after the data byte has been received by the receiver (no transmit commands from the processor for loading the transmission data registers are needed).

4.3 SpecChart Specification

The top level Statechart and SpecChart for the UART is given in Figure 7. We must mention at this point that the SpecChart model was made as similar to the Statechart model as possible, so that we could compare the features of the two languages on the same modeling style (as opposed to comparing two different modeling styles of the UART).

The SpecChart model has three concurrent behaviors *transmitter, receiver* and *uart_mode*. The *transmitter* behavior consists of concurrent sub-behaviors: *tx_hold_reg* which latches the data byte to be transmitted in an internal register, and *tx_mode* which transmits the byte serially. The *receiver* behavior also consists of two concurrent sub-behaviors: *rx_mode* which assembles the serial data bits as they are received into an internal register, and *fifo* which transfers the byte received to the processor. The behavior *uart_mode* checks when the UART is in the echo mode and accordingly transfers the received serial data to the transmit register so that it can be then transmitted out serially.

All the arc conditions are implicitly ANDed with a *rising(clk)* event. The SpecCharts for the *receive* and *transmit* behaviors are given in Figure 8 and Figure 9. The complete SpecChart for the UART in the textual form is given in Appendix B.

4.4 Analysis

Using SpecChart or Statechart “substates” makes the specification easier to comprehend than is possible using an HDL.

A close examination of the StateChart and SpecChart of Figure 8 and Figure 9 will reveal that the SpecChart description is much easier to comprehend than the Statechart. We now discuss the reasons for this.

At the top level (Figure 7), both languages seem quite similar. The minor difference is that in SpecCharts all actions occur upon entering a state, never during an arc transition. On the other hand, Figure 8 and Figure 9 demonstrate major differences in the languages due to the lack of programming constructs in Statecharts. For example, the *receive* behavior detects a start bit (line=1) and then reads the next eight bits into a register. The reading of the eight bits is clearly shown using the for loop in *read_bits* of the SpecChart of Figure 8. Note that synthesis will likely unroll the for loop, so no index counter is needed in the implementation.

In the Statecharts language, no such for loop is possible since only FSMs can be described. One possible specification approach is to mimic a for loop by introducing a new variable i initialized to 0. An arc with the condition $i < 8$ and action $i := i + 1$ is added that loops back on *read_bits*. Another arc with condition $i = 8$ points to the C fork. Not only is this less readable than a for loop, “unrolling” such a configuration during synthesis is difficult. Hence it is likely that a register and incrementer will be synthesized for i, which is inefficient. A second specification possibility uses a trick in which the 8-bit register is loaded with a ‘1’ in its rightmost bit, ‘0’s in the other bits. Each input bit is shifted

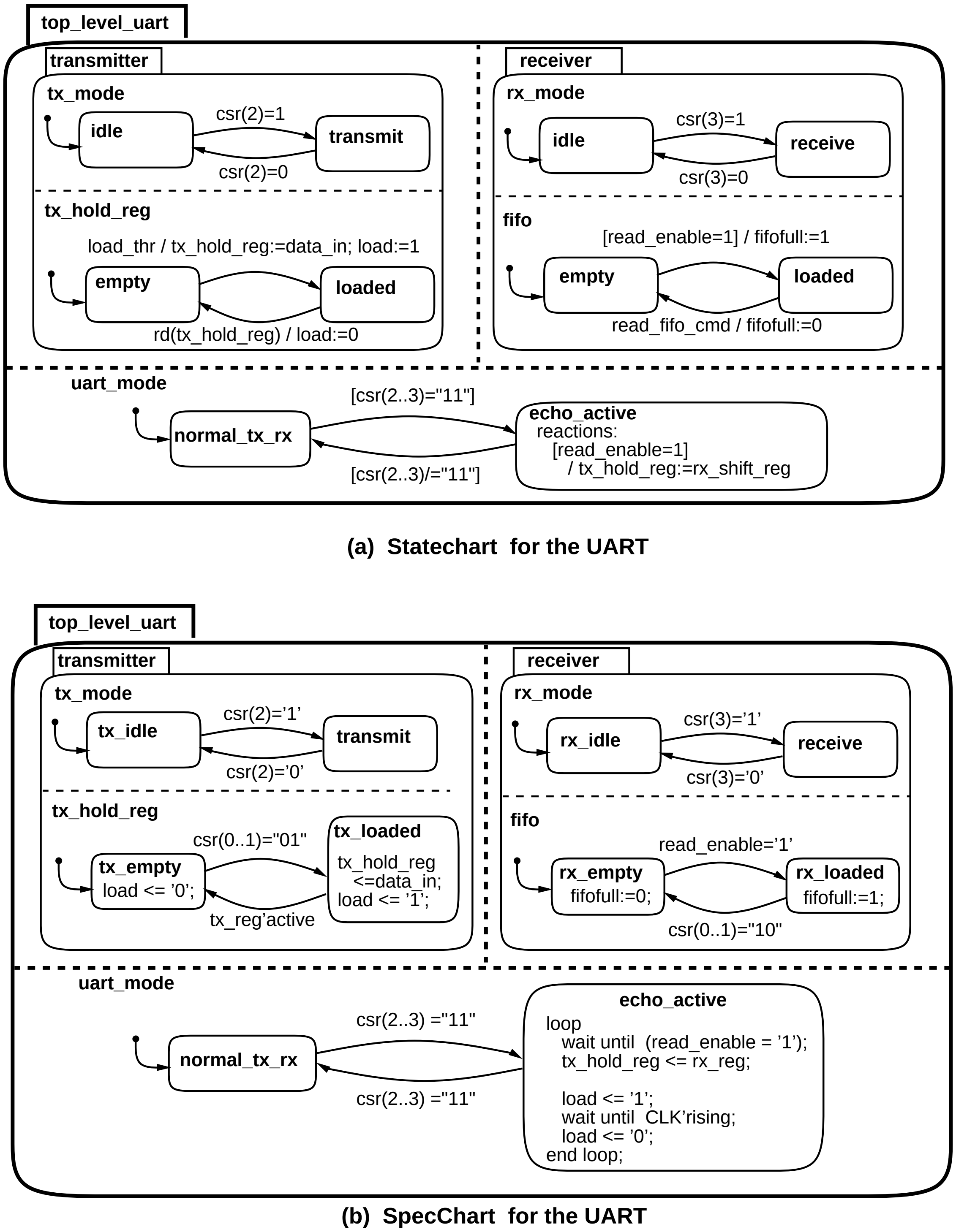
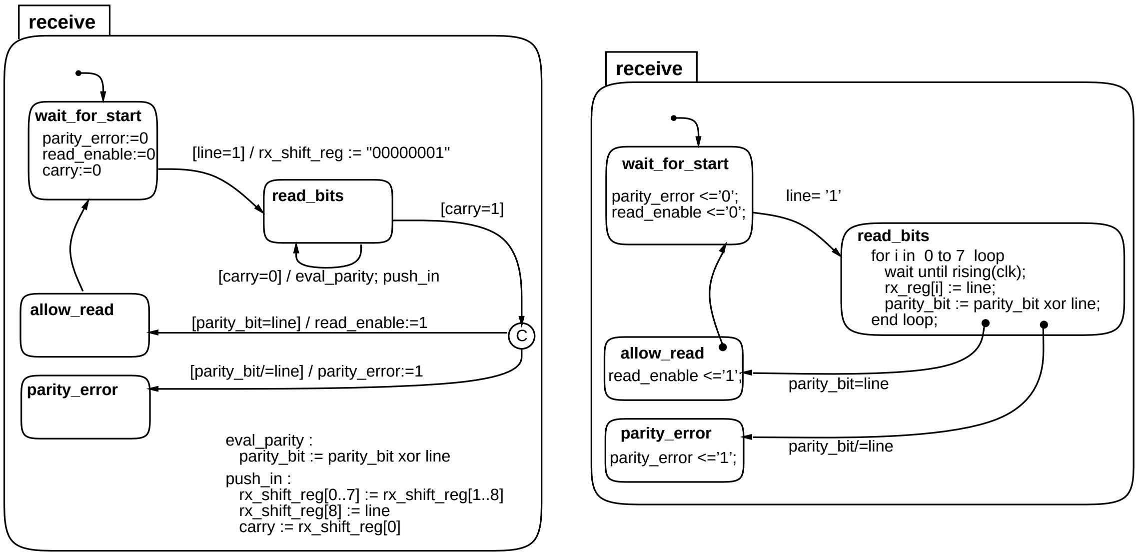
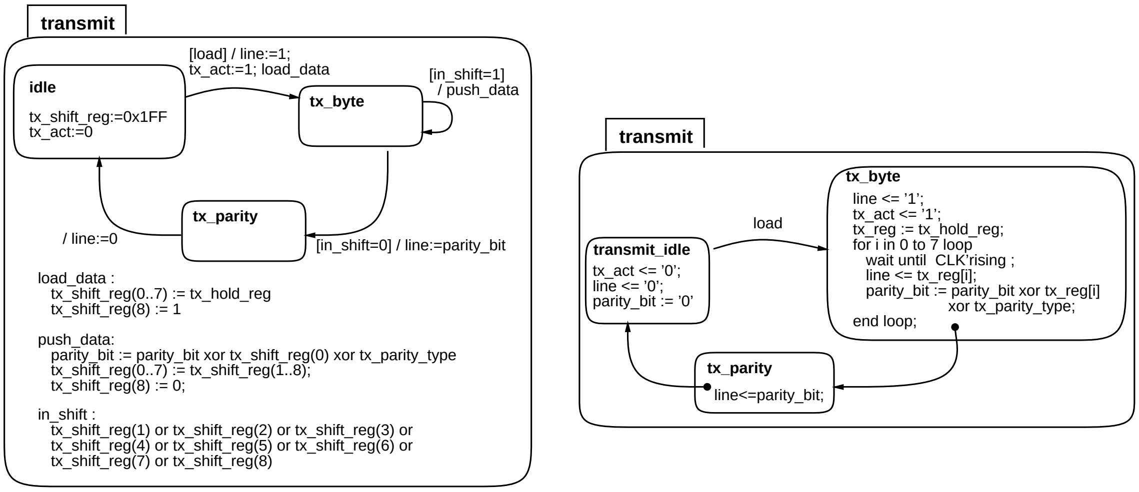


Figure 7: SpecChart of the UART (top levels)



(a) Statechart **(b) SpecChart**

Figure 8: SpecChart for the UART receive behavior



(a) Statechart **(b) SpecChart**

Figure 9: SpecChart for the UART transmit behavior

into this register. After each shift, the shift-out value (carry) is checked; a value of '1' signifies that all 8 bits have been shifted in. This is the trick used in the Statecharts model. A similar approach was used in the *transmit* behavior. Such tricks are made necessary when trying to model behavior using FSMs; the result is a specification that is very difficult to comprehend.

In Statecharts, the state names indicate that an action is being performed (e.g. *read_bits*), whereas the action is actually on the arc. This is awkward. In SpecCharts, there are only behaviors (not states). Arcs merely transition from one behavior to another. All actions are part of a behavior. Note this subtle but important difference between the models provided by the languages. Statecharts are essentially FSMs. SpecCharts provide for a complete hierarchy of behaviors.

4.5 UART Modeling Notes

As was mentioned earlier, it was necessary to model the UART in SpecCharts similar to the Statechart presented in [11] so as to facilitate a comparison of the two languages independent of the modeling style. However, there were a few issues in the Statechart model which we felt might lead to inaccuracies in the design.

For example, the *transmit* and *receive* behaviors are shown as concurrent even though only one of them can be active at any one time. This is due to the fact that there is only one single bit port LINE for serial transmitting/receiving. Since making these two behaviors concurrent was a fundamental modeling decision on part of the Statechart designer, we implemented the same in the SpecChart model.

Another inaccuracy is in the echo mode. After the received bits have been assembled into RX_SHIFT_REG, the byte is transfered to the TX_HOLD_REG, but the LOAD line is not set to logic level '1' to start the serial transmission (i.e., to cause the transition labeled *[LOAD]/LINE:= 1; TX_ACT := 1; LOAD_DATA* between states *idle* and *tx_byte*) in the Statechart *transmit*. Thus in the echo mode, the received byte can never be transmitted. We rectify this in the SpecChart by toggling the LOAD condition in behavior *echo_active*. Also, at the same time the byte is being 'echoed' or transmitted, since the *receive* state machine is enabled, it will be receiving the same bits as they are being transmitted out. This can be corrected by having separate serial data ports for transmitting and receiving, or by ensuring that the transmit and receive operations are not performed simultaneously.

In the Statechart the parity bits in the *receive* and *transmit* states are not initialized. Thus, for every time that a data byte is received or transmitted, the parity calculated could be wrong since it depends on the parity value calculated for the previous byte. We have corrected this by resetting the parity bit in the behaviors *receive_idle* and *transmit_idle*.

The Statechart also did not specify how the value TX_PARITY_TYPE is determined. We have assumed in the SpecChart that it is an external port.

5 A Pipelined Processor

5.1 Source and Brief Description

This example was obtained from a Verilog description of a pipelined processor in a book on modeling with Verilog [12]. It is a simple RISC processor with two-address instructions. The fetch, execute, and write-results stages of operation are pipelined.

5.2 English Specification

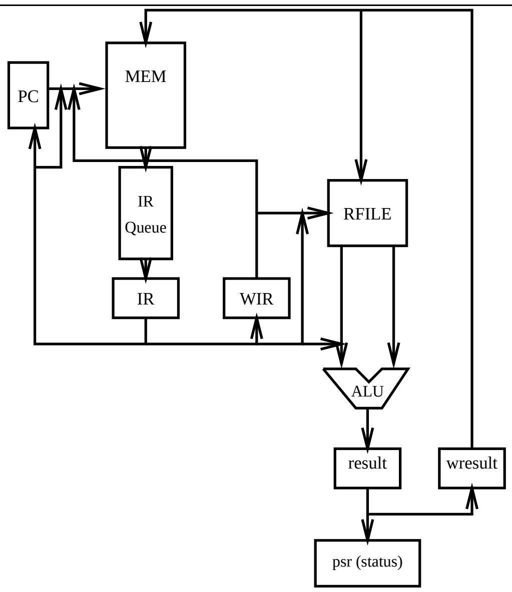


Figure 10: Block diagram of the pipelined processor

Figure 10 gives a block diagram of the pipelined processor. The processor supports two address instructions, where an instruction contains a register/register operation (thus a register file is present), or a register/memory transfer (load or store). Immediate operands can also be specified. The processor contains its own memory; this is merely to avoid detailing the memory to processor interface when instead we want to concentrate on the pipelining behavior. Other processor components include a program counter (PC), an instruction register (IR), an ALU with add, multiply, complement, shift, and rotate operations, and a status register (PSR). Momentarily ignoring any pipelining, an instruction is performed in three stages: *fetch, execute,* and *write result.* The fetch stage reads the instruction pointed to by the program counter into the instruction register. The execute stage either loads the PC with a new address (for a branch instruction), loads the result register with the result of an operation (for an ALU operation instruction), or moves data between a register and memory (for a load or store instruction). If the instruction was an ALU operation, then the *write results* stage writes the contents of the result register to the appropriate register or memory location.

To increase the efficiency of instruction execution, an instruction pipeline is implemented. The fetch, execute, and write result stages all perform in parallel in a single clock. *Fetch* fetches an instruction into an instruction queue. *Execute* pops an instruction off the instruction queue into the IR and then executes; if there was no instruction available, *execute* is idle. *Write results* stores the result of an operation instruction from the previous clock cycle to the appropriate location. Note that the destination location is needed from the IR, and the result is needed from the result register.

However, these resisters may get overwritten by the other stages, since they execute concurrently. Thus, after each clock, we need to store a copy of these registers for use by the subsequent *write results*. This is implemented by a two phase clock. On the first phase the three stages execute; on the second, backups are made of the two registers. It is also in the second phase that the queue is flushed if a branch was taken. In addition, the program counter is loaded with the next address in this second phase.

To model the latency of a typical memory, a memory access requires two clock cycles.

5.3 SpecChart Specification

(a)

```
event do_fetch, do_execute, do_write_results;

always @(posedge clock) begin : phase1_loop
    if (!reset) begin
        if (!queue_full && !mem_access) -> do_fetch;
        if ((qsize > 0) || mem_access) -> do_execute;
        if (result_ready) -> do_write_results;
    end
end

always @(negedge clock) begin : phase2_loop
    if (!reset) begin
        ...
    end
end

always @(do_fetch) begin : fetch_block
    fetch;
end;
always @(do_execute) begin : execute_block
    execute;
end;
always @(do_write_results) begin : write_block
    write_result;
end;

always @(posedge clock or negedge clock) begin
    if (reset=1) perform_reset;
end
```

(b)

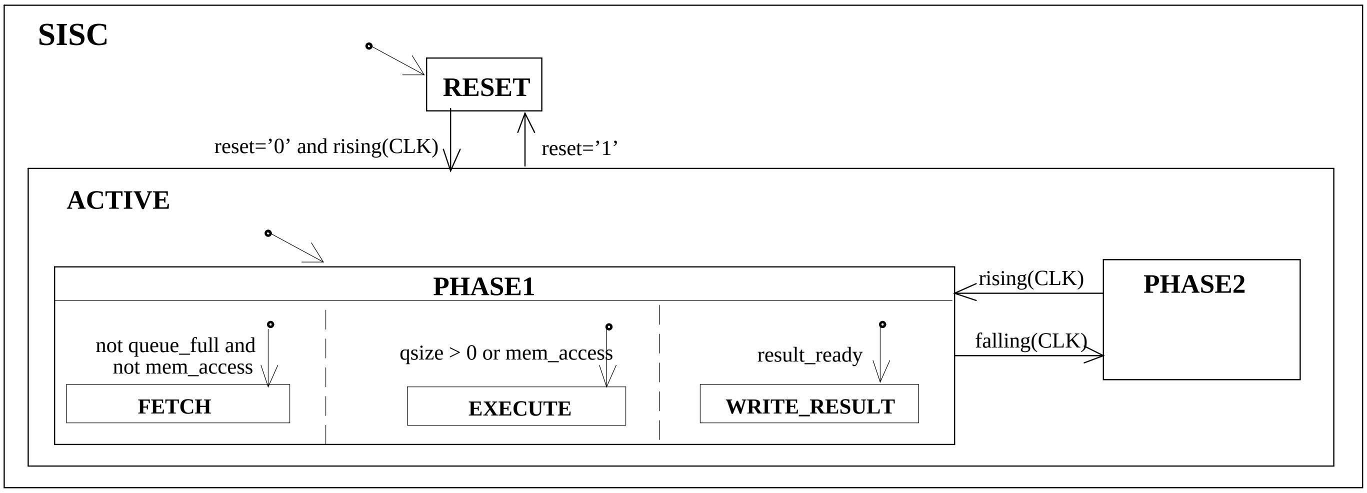


Figure 11: Overall control of the pipelined processor

Figure 11 provides the overall control model for the processor. Both a Verilog HDL description and a SpecChart description are shown.

5.4 Analysis

Note that it is much simpler to determine the functionality from the SpecChart model than from the HDL model. While the graphical representation plays a large role in this, it does not play the key role. Specifically, the underlying model of a SpecChart description is a hierarchy of sequential and concurrent behaviors, whereas that for a typical HDL is a set of concurrently executing processes. Thus, even a textual SpecChart of the processor is simpler to understand, since the hierarchy of functionality and the state transitions are explicit. Conversely, a HDL model must be visually scanned several times, and the hierarchy must be developed in one's mind after being extracted from the use of the activation control signals.

Another important difference to note between the two models is the specification of the reset. In the SpecChart model, it is clear that regardless of the execution status of the processor, the reset input becoming '1' *immediately* resets the processor, as indicated by the TI arc pointing from *Active* to *Reset*. On the other hand, specifying this immediate transition is almost impossible in the HDL description without hopelessly cluttering the model with checks for reset='1' after every statement during which time advances. Thus, the HDL model checks the reset only at the beginning of each phase. This is a strange way to specify a reset (it's essentially an edge-triggered reset rather than an asynchronous one), but it is the best that can easily be done in the HDL, since there are not abstractions for hierarchical behaviors.

6 The Intel 8237A DMA Controller

6.1 Source and Brief Description

This example was adapted from the Intel Microprocessor and Peripherals Handbook [13]. The 8237A Direct Memory Access Controller is a peripheral interface circuit for microprocessor systems, which allows external devices to transfer data to and from the memory directly. The 8237A offers a wide variety of of programmable control features, four independent DMA channels, memory-to-memory transfers and allows dynamic reconfiguration under program control.

6.2 English Specification

A full description of the 8237A is given in [13]. Briefly, the 8237A contains four independent DMA channels and may be expanded to any number of channels by cascading additional controller chips.

The DMA transfer can be performed in one of four modes. In the *Single Transfer Mode* (STM), the device requesting the DMA service is programmed to transfer one byte whenever its request is acknowledged. The *Block Transfer Mode* (BTM) allows the device to transfer the entire data until the word count register has been decremented to zero. The *Demand Transfer Mode* (DTM) the DMA is programmed to continue making transfers until either the word count has reached zero or the device requesting the service removes its request. In the *Cascade Transfer Mode* (CTM), more than one 8237As can be cascaded to enable more than the four devices to request and receive DMA service.

Each of the DMA transfers can be one of several types. *Read* transfers move data from the memory to the I/O devices, *write* transfers move data from I/O devices to the memory, *verify* transfers are pseudo transfers in that the appropriate addresses are generated but the I/O and memory control lines remain inactive. *Memory-to-memory* (MTM) transfer can achieve the transfer of data from one memory to another.

For each of the four channels, the 8237A has *base* and *current* word count and address registers. While the base registers retain the values written by the microprocessor during the DMA service, the current registers are constantly updated by incrementing/decrementing them during the data transfer. Each channel has an associated 6-bit *mode register* which determines the mode of operation (STM, BTM, DTM, or CTM) and the nature of the data transfer (read, write, verify or MTM) for the channel as described above. The mode register also indicates whether the autoinitialization is enabled for the channel, and whether the address has to be incremented or decremented.

The 8237A provides two rotating and fixed priority schemes to resolve contentions for DMA service by the four channels. The channels can be programmed to *autoinitialize* , i.e., restoring the current word count and address registers with the original values from the base registers at the completion of the DMA transfer by that channel.

The microprocessor configures the the 8237A by writing to the *command register, request register, mask register,* and the *mode registers* for each channel. In addition, the microprocessor may read the status register to examine the status of the devices at any point.

6.3 SpecChart Specification

Figure 12 shows the behavioral transitions for the DMA controller. All declarations as well as all leaf behavior details (i.e. VHDL sequential statements) are omitted for clarity. The complete SpecChart in its textual form can be found in Appendix E.

The SpecChart consists of two concurrent behaviors – *main* and *priority.* The behavior *main*

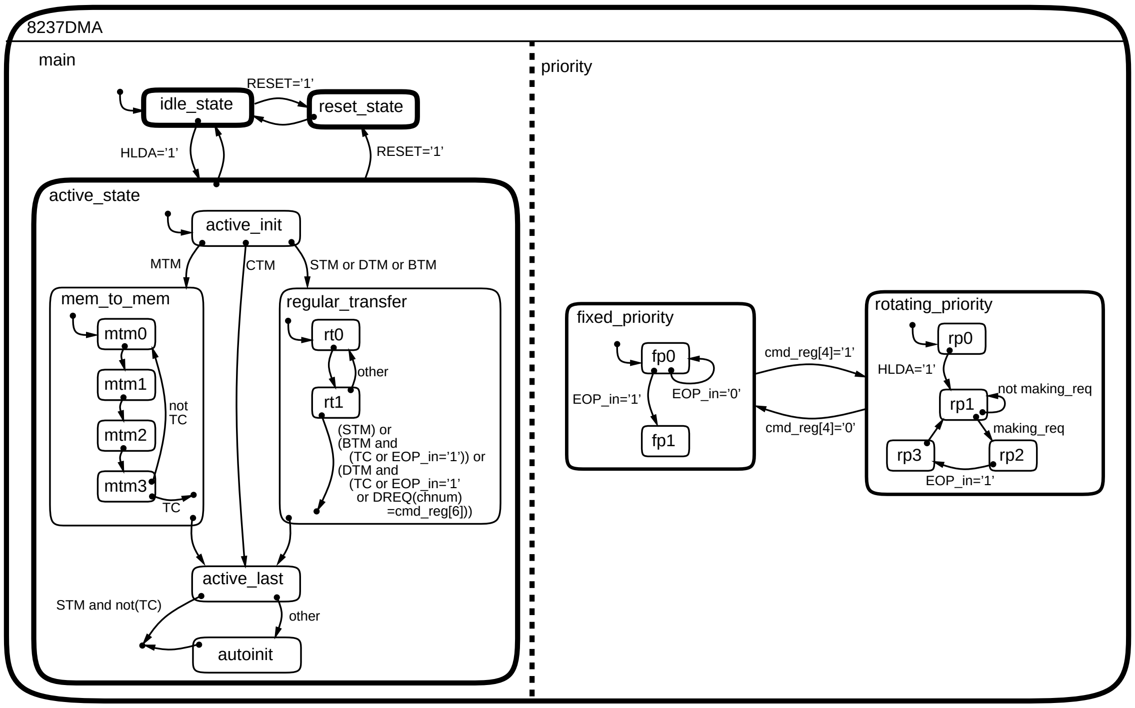


Figure 12: SpecChart of the 8237A DMA Controller

contains sub-behaviors which describe the microprocessor-8237A interface, the various DMA transfer modes (STM, DTM, BTM and CTM) and the data transfer types (read, write, verify or MTM). The behavior *priority* updates and decides the relative priority of the four channels requesting DMA service. This is used by the behavior *main* in reponding to simultaneous requests for service.

6.4 Analysis

Since the microprocessor's access of the 8237A depends on the values on the chip select (CS), the I/O read and write lines, the lower four bits of the address bus, the software commands to the 8237A were easily represented using a case statement and nested if-then-else statements in the *idle_state*. Writing FSMs for these behaviors, as would be required by a Statechart description, would not only be tedious but would clutter the specification.

The sequential sub-behaviors in SpecCharts are well suited for activating the appropriate behaviors based on the DMA transfer mode as shown in the behavior *active_state*. The concept of completion enables us to specify transitions between sequential behaviors even when there are no explicit events to initiate the transitions other than the knowledge that the previous behavior must complete before the next one is activated. (e.g. sequencing between *mtm0, mtm1, mtm2* and *mtm3*).

7 A Telephone Answering Machine

7.1 Source and Brief Description

This example was derived from the behavior of a telephone answering machine currently available on the market. In addition to standard answering machine functions, it has remote operation capability and tollsaver.

7.2 English Specification

Typical operation consists of answering a phone line after a required number of rings (see below), playing a prerecorded announcement, and recording a message onto a tape. A block diagram of the portion of the answering machine which we wish to specify is shown in Figure 13.

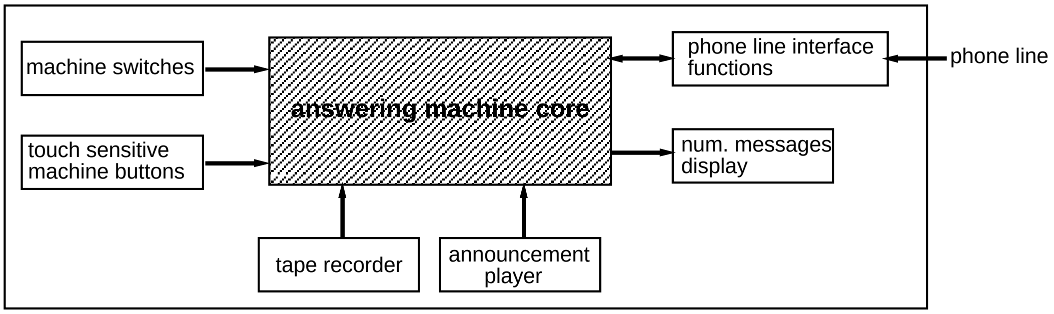


Figure 13: Block diagram of the answering machine (the shaded portion is to be specified)

The machine has several external buttons, switches, and a numerical display. The "system-on" switch is essentially the power switch; when in the "off" position, the machine ignores the phone line and all buttons. When "on", the display shows the current number of messages. Pressing the "play-messages" button will play all messages. Regular tape-player buttons are also available (play, forward, rewind, stop); pressing any of these will make the current number of messages zero. Pressing the "record-announcement" button causes a beep to sound, followed by another beep several seconds later. Anything spoken into the microphone between these two beeps becomes the prerecorded announcement. The announcement can be heard by pressing the "play-announcement" button. Pressing and holding the "memo" button allows one to record a message by speaking into the external microphone. The message is terminated when the button is released. "memo" is thus useful for leaving messages to other household members.

The "machine-on" switch denotes that the machine should answer the phone. When on, the number of rings that the machine will wait to answer is usually *four*. However, if the "tollsaver" switch is on and there is a message, the machine answers in *two* rings. Tollsaver allows the machine owner to determine via phone that no messages exist on the machine. If the phone rings three times, then there are no messages and the owner can hang up, thus avoiding any long-distance charges. Sometimes the owner will forget to turn on the machine before leaving the household. Hence, the machine answers the phone after 15 rings when not on. The owner can then remotely turn the machine on, as described below.

Once the machine has answered the phone, it plays the announcement. When completed, a beep is produced and the content of the phone line is recorded until a dialtone is detected or a maximum message time length is reached. Afterwards, the machine hangs up the phone, and again monitors the phone line for rings. If a dialtone was detected while playing the announcement, the machine immediately hangs up, and does not proceed to record a message. If the button-tone 1 is detected either while playing the announcement or while recording a message, the machine terminates either of these activities and enters remote operation mode.

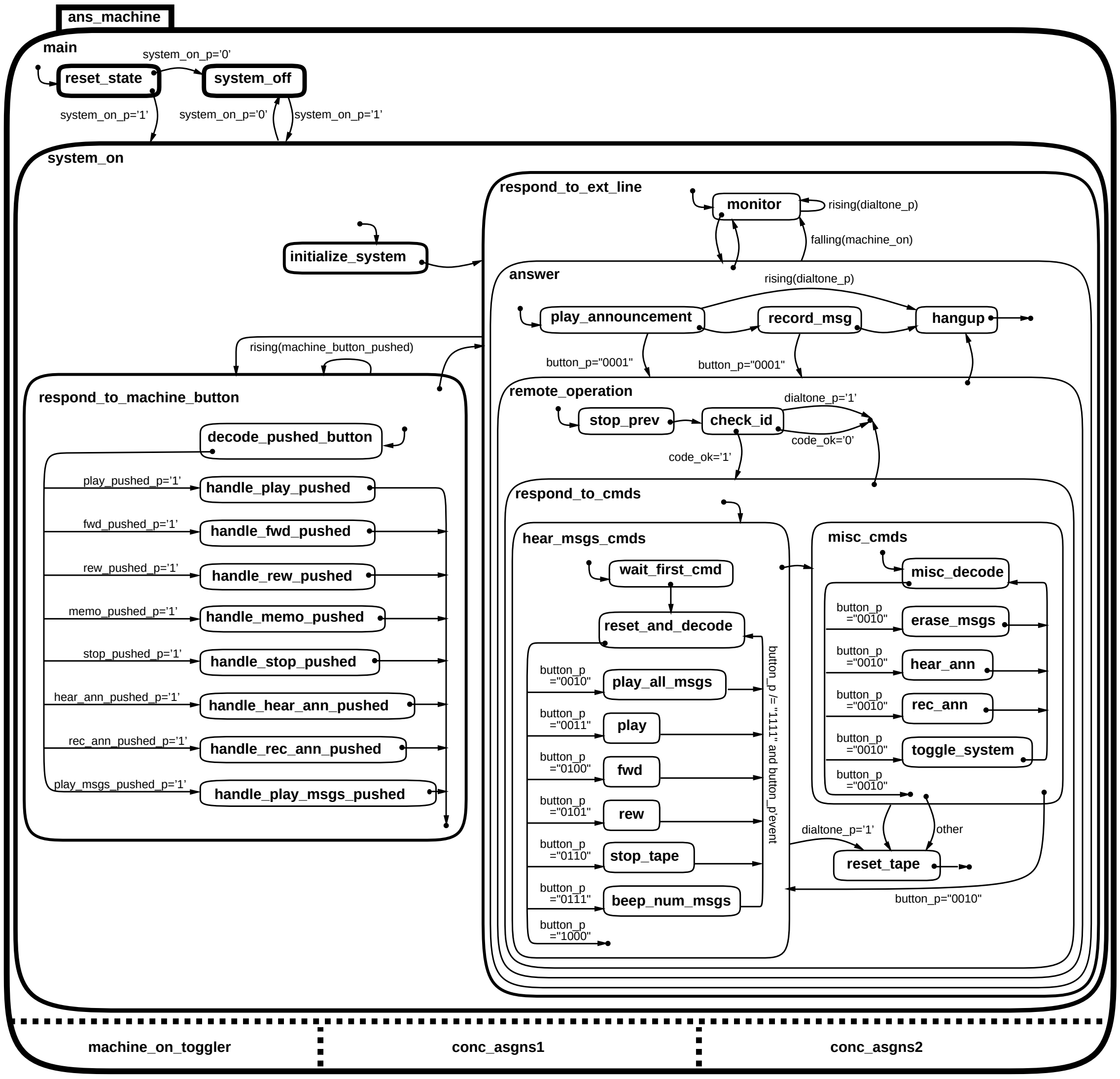


Figure 14: SpecChart of the answering machine (non-leaf levels)

The first step in remote operation mode is to check a user-identification number. The next four button-tone numbers that are pushed are compared to four numbers stored internally. If they do not match, the machine hangs up the phone. If they do match, the machine can be instructed to perform any of several basic commands. Button-tone 2 plays all messages. Button-tones 3 through 6 correspond to standard tape-player buttons play, forward, rewind, and stop. Button-tone 7 causes the number of messages to be indicated by a number of beeps. Button-tone 8 instructs the machine to leave the basic commands mode and enter the miscellaneous commands mode. In this mode, button-tone 3 "erases" all messages (i.e. rewinds to the beginning of the tape, sets the message counter to zero). Button-tone 4 plays the announcement. Button-tone 5 is identical to pressing the external "record-announcement" button. Button-tone 6 toggles the "machine-on" switch from off to on or vice-versa (upon hangup). Button-tone 2 returns the machine to the basic commands mode. If a dialtone occurs in either of these command modes, the tape is reset to the end of the last message (i.e. any existing messages are saved). This reset also occurs if button-tone 9 is pressed in basic command mode. After this reset, or if a dialtone occurs while checking the user id, the machine hangs up.

If the external "machine_on" button is set to off at any time after the machine has answered the phone, any current activity is immediately terminated and the machine goes back to monitoring the phone line, as above. This is useful for screening calls. External machine buttons have precedence over the machine answering the phone; hence pressing any external button will also terminate any current activity.

7.3 SpecChart Specification

Figure 14 shows the behavior transitions for the answering machine. All declarations as well as all leaf behavior details (i.e. VHDL sequential statements) are omitted for clarity. The details of the *monitor* leaf behavior are shown in Figure 15. The complete SpecChart in its textual form can be found in Appendix D.

Note that the answering machine is entirely event-driven; there is no notion of a clock. For example, at some time the *monitor* behavior may be active. If the *machine_button_pushed* event occurs (meaning that an external machine button was pushed), the *monitor* behavior is terminated and the *respond_to_machine_button* behavior is activated. In fact, this event terminates the *respond_to_ext_line* behavior regardless of which subbehavior is executing.

7.4 Analysis

Since the answering machine specification is essentially a hierarchy of modes with numerous events that cause immediate mode transitions, SpecCharts is an ideal language for capturing the specification.

There is no simple way to capture this system using HDL processes for each mode. Using procedures would also be difficult due to the number of state variables and nested case statements necessary. More importantly, handling the events in a procedural description would require modifying all "wait" statements to detect all appropriate events and then "jumping" to the appropriate next state; writing such HDL code is an arduous task. Essentially, it would be *extremely difficult* to write HDL code for the answering machine, and any such code would be *unreadable* for all practical purposes.

Note the extensive use of programming constructs (e.g. CHECK_ID). FSM languages such as Statecharts would require that such behaviors be described as a FSM, which unnecessarily obscures the functionality. Also note the extensive use of the completion concept in the SpecChart description (remember that Statecharts has no such concept). Note the elegance of completion being defined for any behavior, whether described as sequential sub-behaviors or as VHDL sequential statements.

In summary, the concept of behavioral hierarchy is essential for concise and accurate specification of the answering machine. Without it, specification is cumbersome, error-prone, and unreadable.

```
name {MONITOR}
declarations
{
   variable num_rings: integer range 1 to 20;
   variable i: integer range 0 to 20;
   procedure determine_num_rings( num_msgs : in integer ;
                                  tollsaver : in bit ;
                                  machine_on : in bit ;
                                  variable num_rings : out integer ) is
      begin
         if ((num_msgs > 0) and (tollsaver = '1') and (machine_on = '1')) then
            num_rings := 2;
         elsif (machine_on = '1') then
            num_rings := 4;
         else
            -- override in case forgot to turn machine on
            num_rings := 15;
         end if;
      end;
}

code
{
   determine_num_rings(num_msgs,tollsaver_p,machine_on,num_rings);
   i := 0;
   while (i < num_rings) loop
      wait  on ring_p,machine_on,tollsaver_p;
      if ring_p = '1' and ring_p'event then
         i := i + 1;
      else
         determine_num_rings(num_msgs,tollsaver_p,machine_on,num_rings);
      end if;
   end loop ;
   offhook_p <=  '1';
}
```

Figure 15: Example leaf behavior of the answering machine

8 A Microwave Transmitter Controller

8.1 Source and Brief Description

This example was designed by a San Diego consulting firm using SpecCharts as the specification language and starting point for software generation. It is a controller for a system which allows a television cameraman to transmit his/her video signal over any one of 45 different microwave frequencies, at one of three power levels. The cameraman may change frequencies or power levels at any time. Safety features are included to avoid accidental changes and to ensure certain frequencies are avoided near government installations.

8.2 English Specification

A block diagram of the microwave transmitter system is shown in Figure 16. A camera's video output is transmitted over a particular microwave frequency. Since several cameramen may be operating nearby, e.g. while covering a golf tournament, it is necessary for each cameraman to transmit over a unique frequency to avoid interference. A cameraman may also need to change frequencies to avoid other sources of interference, and in some cases may need to increase the power of the signal to overcome the interference. Thus each cameraman requires a hand-held unit for changing his microwave transmission frequency and power level.

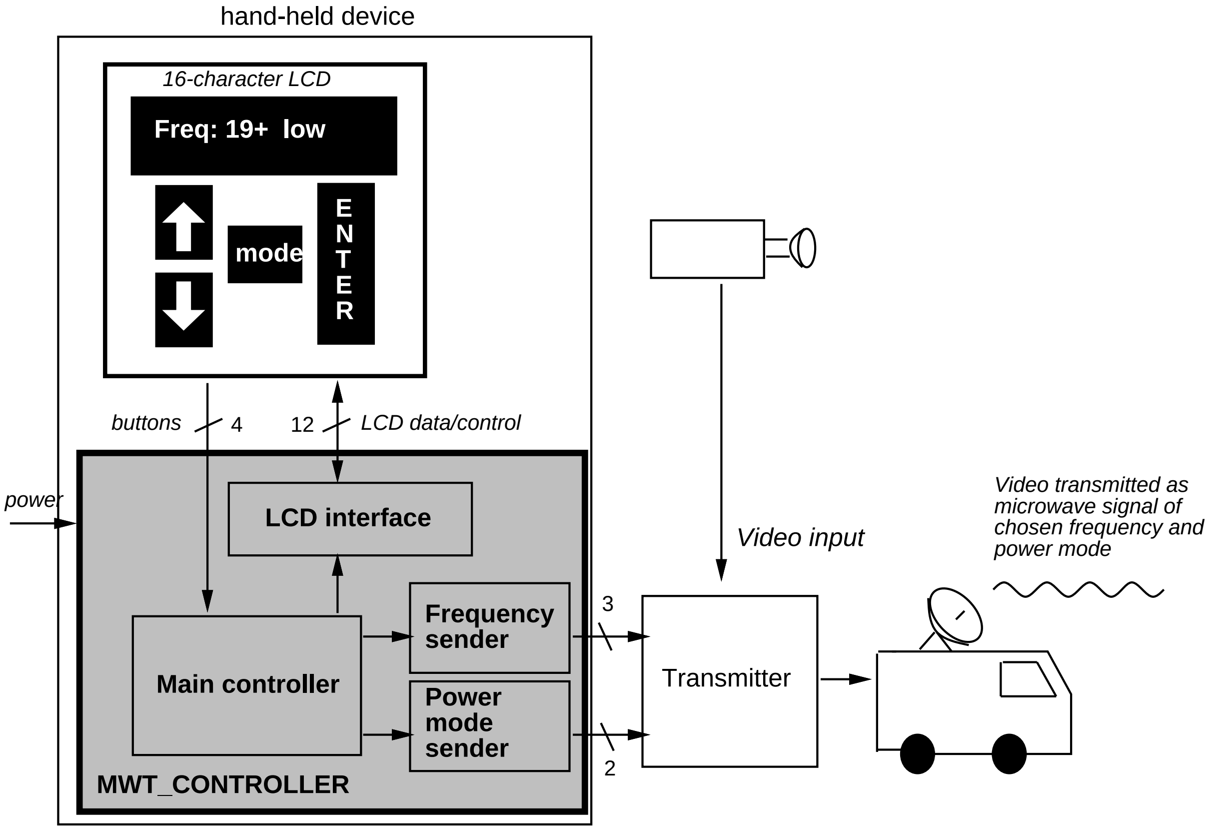


Figure 16: Block diagram of the microwave transmitter system (the shaded portion is to be specified and designed)

Figure 16 shows a hand-held unit consisting of an LCD and four buttons. The LCD displays the current frequency and power level. The frequency is displayed as a "channel", where there are 15 channels available for video transmission, each having a possible + and − offset, resulting in 45 possible transmission frequencies. There are 3 power levels, "Hi", "Low", and "Standby".

To change the power level, the *level* button is pressed, causing the LCD to display the next higher level, returning to "Standby" if the level was "Hi". To change the frequency, the up or down buttons are pressed, causing the LCD display to increment or decrement the channel, incrementing to the minimum channel after the maximum one, and vice-versa. For example, to change from channel 1 to channel 3+, one must step through the intermediate channels 1+, 2−, 2, 2+, 3−, and 3. However, we must not allow transmission to occur over the intermediate frequencies. The reason is that those frequencies may already be in use. Therefore, changes are only tentative and don't actually take effect until the *enter* key is pressed. Thus, when *up* or *down* is pressed, the LCD begins blinking to indicate that the displayed channel is merely a pending channel; likewise when *level* is pressed. When the enter key is pressed, the new frequency and power-level are sent to the transmitter, and the LCD stops blinking.

Since the unit will likely be strapped to the cameraman's belt, a button may be pressed accidentally. To avoid leaving the unit in a blinking state, it is required that *enter* must be pressed within ten seconds after the last pressing of *up*, *down*, or *level*; otherwise the LCD stops blinking and displays the actual transmission channel and power level.

The frequencies corresponding to channels 11-15 are commonly used by government installations. To provide extra assurance that one of these channels is not selected when near such installations, the unit should be easily configurable to make either channel 10 or channel 15 the largest channel. Holding the enter key for at least five seconds and then releasing it should put the unit in a state where the user can toggle between the two largest channel configurations by pressing either *up* or *down*. Pressing *enter* returns the unit to its normal state. If the current channel was between 11 and 15, and the unit was just configured with channel 10 as the largest channel, then the current channel should become 1, and the power-level should become "Standby".

When the unit is turned on, the company name and the serial number of the unit should be displayed for a few seconds. The initial channel and power level should then be the same as when the unit was last shut off.

8.3 SpecChart Specification

Figure 17 shows the behavior transitions for the microwave transmitter controller. All procedure bodies as well as all leaf behavior details (i.e. VHDL sequential statements) are omitted for clarity.

After choosing the specific LCD to use (Optrex's 16-character Dot Matrix LCD Module DMA Series), procedures were written to implement the protocols required to initialize the LCD and to send an instruction. Building on this, other procedures were then written to perform a clear, write a character, write a string of 16-characters, turn the display on or off (for blinking), and write the LCD with a frequency and power-level.

Procedures were then written to interface with the transmitter. One contained the protocol for sending a single bit to the transmitter, another for sending a new frequency value (by sending a series of bits), and a third for sending the power-level.

Variables were declared to maintain the maximum allowable frequency (*maxfreq*), the current frequency (*freq*), the current power-level (*level*), the pending frequency (*pending_freq*), and the pending level (*pending_level*). A frequency is an integer from 1 to 45; the *UpdateLcd* procedure converts the integer to the appropriate channel (1-10) and offset (−, none, +) by using division and mod opera-

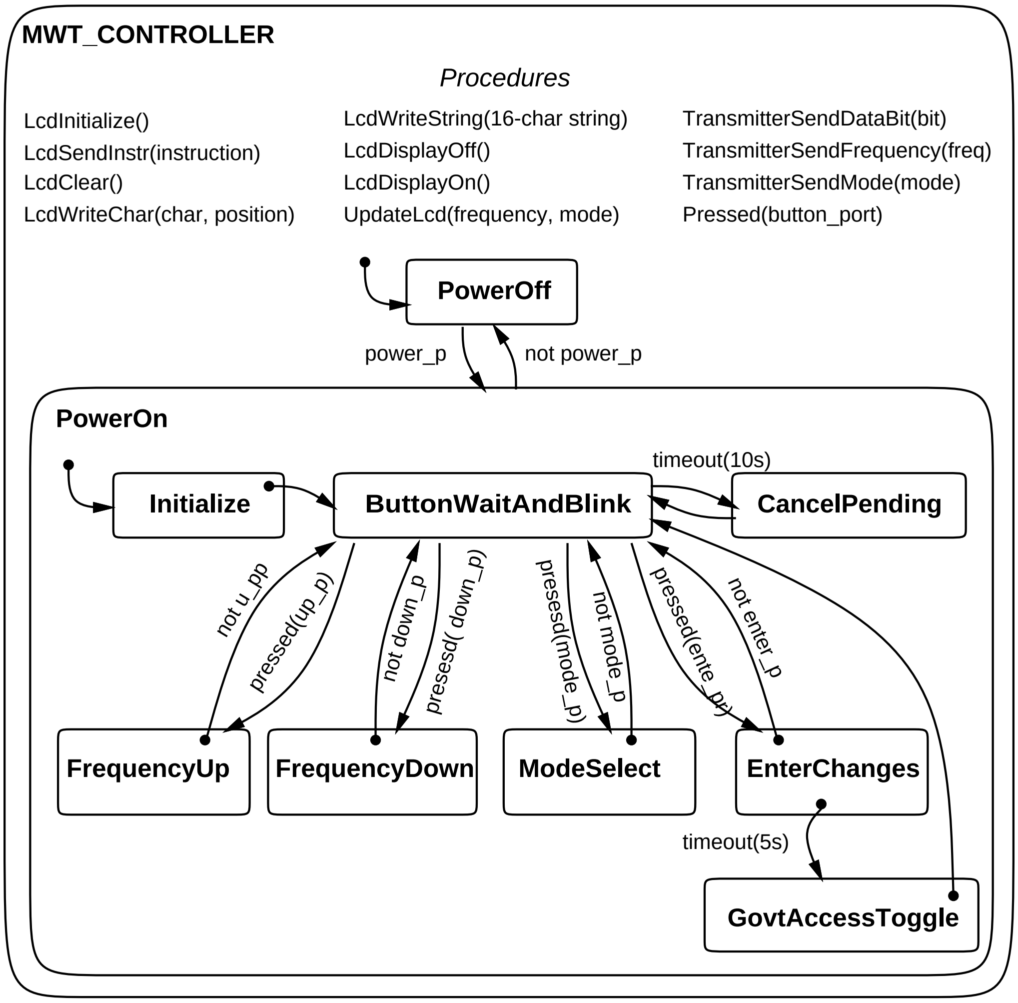


Figure 17: SpecChart of the microwave transmitter controller (state-transitions)

tions. An additional boolean variable *change_pending* is used to indicate whether a new frequency or power-level is pending.

The various modes and transitions for the controller were then determined. The first mode after power up is *Initialization*, in which the LCD is initialized, and initial values are determined for the frequency and power-level. The controller then waits for a button to be pressed while in the *ButtonWaitAndBlink* mode. If *up*, *down*, or *level* is pressed, the *FrequencyUp*, *FrequencyDown*, or *LevelSelect* mode is entered, respectively, to determine the appropriate next pending frequency or power-level. Each then calls *UpdateLcd* and sets *change_pending* to true. Afterwards, when the depressed button is released, control returns to the *ButtonWaitAndBlink* mode. Since *change_pending* is true, this mode turns the LCD on and off for one-half second each, thus achieving the desired blinking to indicate that the display shows the pending frequency and power-level, not the actual ones. If the controller stays in this mode for 10 seconds, the *CancelPending* mode is entered, which restores the LCD with the actual frequency and power-level, and sets *change_pending* to false to stop the blinking, after which *ButtonWaitAndBlink* is re-entered.

If *enter* is pressed while in the *ButtonWaitAndBlink* mode, the controller transitions to the *EnterChanges* mode, in which *freq* and *level* are updated with the pending values, *change_pending* is set to false, and the new values frequency and power-level values are sent to the transmitter by calling *TransmitterSendFrequency* and *TransmitterSendLevel*.

If *enter* is held down for at least 5 seconds, the *GovtAccessToggle* is entered. The LCD then displays whether or not channels 11-15 are enabled or disabled. After releasing *enter*, pressing *up* or *down* toggles between enabled or disabled. Pressing *enter* again returns the controller to the *ButtonWaitAndBlink* mode. Note that changes to the maximum channel are made immediately, i.e. there is no concept of a pending maximum channel.

8.4 Analysis

The state-transition constructs in SpecCharts are clearly useful in this example. Writing such a system in VHDL would require describing the states using a state variable and a case statement, which is cumbersome and more difficult to understand. Also note that the actions performed within each state are essentially sequential programs, thus the use of sequential statements in leaf states is very useful, whereas in Statecharts only simple zero-time actions are allowed. Especially useful is the procedure construct.

The engineer designing this system chose to implement it as software running on the Intel 8751 8-bit microcomputer. The SpecChart was translated into the C language, which was then compiled into the 8751 instruction set. The engineer stated that describing the system with SpecCharts and then translating (manually) reduced design time by approximately 20% (1 man-day) over describing the system with C initially, as is the standard practice. In addition, he stated that although the resulting C code was approximately the same size as to be expected if the system was initially described with C, it was far more structured and thus easier to understand and modify. When the SpecCharts to C translation is automated, he estimates a 40-50% speedup in design time. It should also be noted that the system's behavior was correct the first time.

9 An Ethernet Network Coprocessor

9.1 Source and Brief Description

This example was obtained from R. Gupta and C.N. Coelho at Stanford University. Their HardwareC behavioral specification was derived from the Ethernet 10b2 specifications, and was used as a hardware/software partitioning example in [14]. The HardwareC specification was manually rewritten as a SpecChart specification. The example is a network coprocessor that handles transmission and reception of data to and from an Ethernet link, including direct-memory access, thus relieving a host computer from these tasks. The host computer can send configuration or data-transmit instructions to the network coprocessor.

9.2 English Specification

A block diagram of the network coprocessor environment and a decomposition of the coprocessor into modules are shown in Figure 18. The host CPU sends instructions to the coprocessor's execution unit. Since such instructions may arrive faster than they can be executed, instructions are stored on a queue until executed.

For a transmit instruction, the execution unit sends the starting memory address to the transmit unit, which then directly accesses the host system's memory through a DMA protocol. Starting at the given address, the memory contains the size of the data to transmit, the data itself, and the destination network address to which the data should be sent. The data is sent over the network enclosed in a *frame*. The frame consists of a number of preamble byte patterns, a start-of-frame byte pattern, two bytes indicating the destination address, two bytes indicating the host system's address, a byte indicating the number of actual data bytes to follow, the data bytes themselves, a parity byte computed from the previous bytes starting with the destination address, and finally an end-of-frame byte pattern. The transmission unit sends the frame serially over the TXD line to the network, enabling each data bit using TXE. If a collision is detected, the unit transmits a number of jam bytes, waits for some amount of time, and then attempts to retransmit the frame. After a successful transmission, the unit waits for a period of time required between frames before attempting another transmission.

Separate instructions exist for the host CPU to configure the coprocessor by setting the preamble byte pattern and the number of such bytes to send per frame, by setting the start-of-frame and end-of-frame byte patterns, by setting the host system's network address, by setting the pattern and number of jam bytes sent after a collision, and by setting the interframe spacing time.

The reception unit continually receives data from the network until it detects a start-of-frame byte. It then receives the destination address and compares this with the host system's address. If they differ, the unit again waits for the next start-of-frame byte. However, if they are equal, the unit receives the source address, the data length, the data bytes themselves, and the parity byte. If the next byte is the end-of-frame byte, and the received parity equals the parity computed by the unit itself, then the data is stored in a local memory, otherwise an error has occured. Instructions exist for the host CPU to deactivate or reactivate the reception unit.

9.3 SpecChart Specification

Figure 19 shows the top-level concurrent behaviors (processes) for the network coprocessor. Several of the behaviors are decomposed into sequential substates, as shown. All other behaviors are described

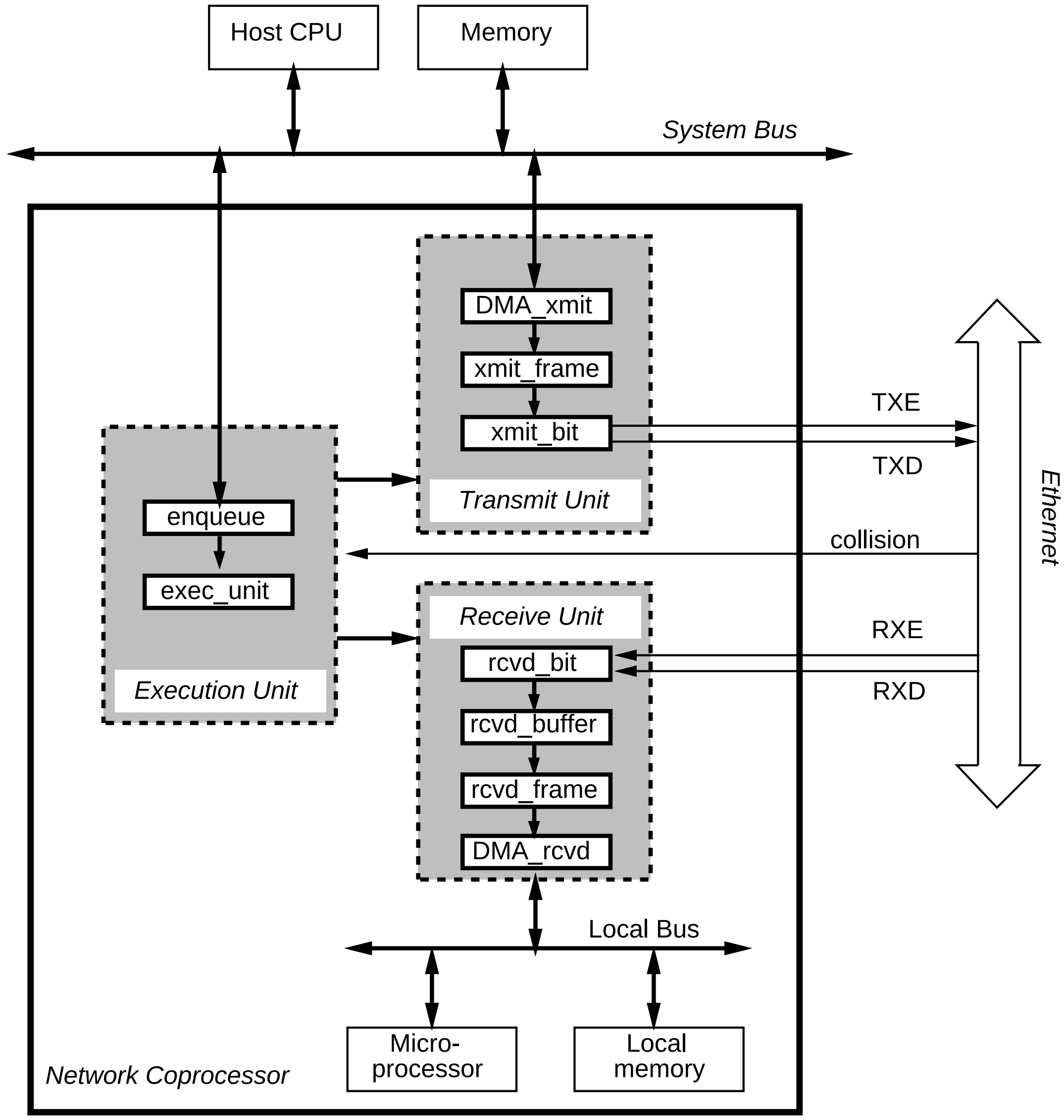


Figure 18: Block diagram of the network coprocessor (source: Gupta and De Micheli, EDAC'92)

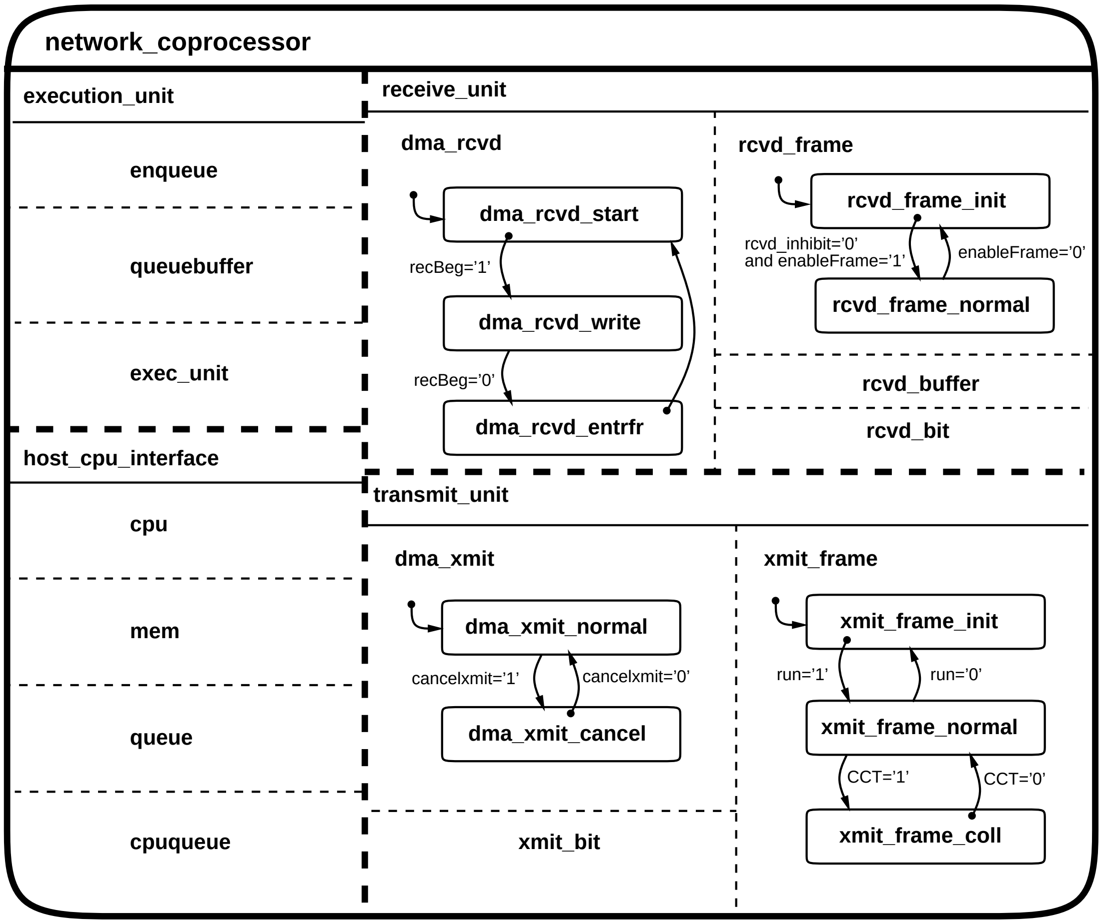


Figure 19: SpecChart of the network coprocessor (state-transitions)

using VHDL sequential statements. We have followed the modeling decision made in [14] to decompose the coprocessor into the ten concurrent behaviors shown, and have maintained the same behavior and signal names. The detailed data-flow between the processes is shown in Figure 20.

Included in the SpecChart is the interface behavior of the host CPU and memory. The *cpu* behavior grants bus access permission. *mem* is the system memory and contains several pieces of data to transmit. *queue* and *cpuqueue* mimick the action of a host CPU sending instructions to the coprocessor.

enqueue receives instructions from the CPU and writes them to *queuebuffer* which serves as an instruction queue. The CPU can also query *enqueue* to determine if the instruction queue is full.

exec_unit is the main controller for the coprocessor. It reads instructions from the instruction queue and then executes that instruction. It's code is simply a case statement with a branch for each possible instruction. Several instructions cause *exec_unit* to update signals which hold values for the various configuration values discussed above (e.g. preamble byte pattern). For a transmit instruction, the *exec_unit* activates *dma_xmit* with a particular starting memory address, and then waits until notified that the data was transmitted successfully before moving on to the next instruction.

When given the signal to start, *dma_xmit* first requests bus access from the host CPU, after which it directly reads from succesive memory locations (starting with the provided address) the destination address, data length, and actual data, sending each of these to *xmit_frame*. It then releases control of the bus to the CPU and waits for a new address from *exec_unit*. The above describes the normal mode of *dma_xmit* as captured with the *dma_xmit_normal* substate. Note that this substate is enclosed in a loop so repeats much like a process. While *dma_xmit* is still sending data to *xmit_frame*, *xmit_frame* may indicate that transmission failed. The normal mode is then terminated (via a TI arc), the CPU bus is released in the *dma_xmit_cancel* mode, and the normal mode is re-entered, initially waiting to be restarted by *xmit_frame*.

xmit_frame packages the data provided to it by *dma_xmit* in a frame, and then sends the frame as a series of bytes to *xmit_bit*. The structure of a frame was discussed in the previous section. The packaging and sending is done in the *xmit_frame_normal* substate; note that the code for this substate is enclosed in a loop, so it is repeated much like a process. However, if a collision is detected over the *CCT* port, *xmit_frame_normal* is immediately terminated (via a TI arc) and *xmit_frame_coll* is entered. The collision mode first indicates to *dma_xmit* that transmission failed. It then sends a series of jam bytes to *xmit_bit*, waits for some amount of time, waits until the network is available, and starts its normal mode of operation again.

xmit_bit simply receives bytes from *xmit_frame* and sends them serially over the serial network line *TXD*.

Concurrent to the above, *rcvd_bit* continually reads bits from the serial line of the network into a byte and sends that byte to *rcvd_buffer*. *rcvd_frame* takes bytes from *rcvd_buffer* and compares each with the pattern for start-of-frame. When the start-of-frame byte is detected, the next two bytes are interpreted as a destination address, which is then compared with the address of the host system (as configured by the CPU). If the addresses are equal, the rest of the bytes of the frame are read, and sent to *dma_rcvd* which in turn writes the data to a local processor's memory using a DMA protocol. The local DMA protocol has is not shown in the SpecChart in this report.

9.4 Analysis

The various modes or states of several of the coprocessor's processes are easily captured using the sequential subbehaviors of SpecCharts. Also, note that both TI and TOC arcs are used throughout the SpecChart. For example, the TOC arc in *dma_rcvd* is traversed only if *recBeg='1'* AND *dma_rcvd_start*

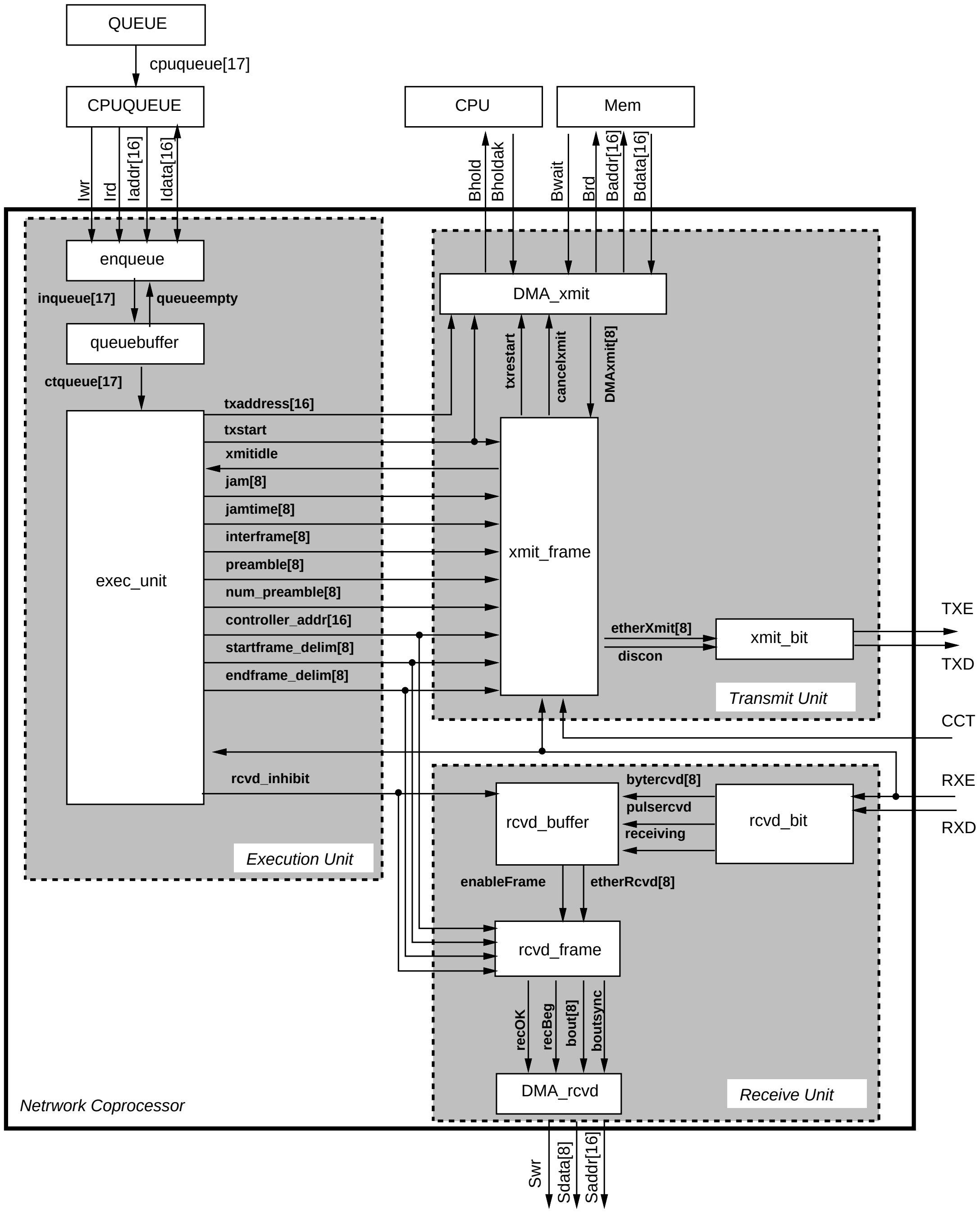


Figure 20: Data-flows between the concurrent behaviors

has completed execution, whereas the TI arc is traversed when *recBeg='0'* regardless of the execution status of *dma_rcvd_write*. The TOC arc in this case is a normal sequencing from one behavior to another. Such sequencing could also be captured by combining the two behaviors into one and inserting a wait statement between them, i.e. *wait until recBeg='1'*. Choosing between the two modeling alternatives is simply a matter of style. The TI arc, on the other hand, indicates an exceptional event which must be handled immediately, i.e. the reception of data by *dma_rcvd* should be terminated if *recBeg='0'*. Not handling this immediately would result in wasted bus cycles, since the system bus would be used to receive data that is eventually discarded.

The combination of sequential subbehaviors and TI/TOC arcs enabled us to write a specification that is much more explicit in its intended behavior than is the HDL specification from which we obtained the example. This improvement is by no means a result of the abilities of the specification writers, but instead of the specification languages. Most HDL's such as HardwareC and VHDL simply do not cater to mode-based systems. As an example, consider the *dma_xmit* behavior. In Figure 19, we see that the *dma_xmit_normal* mode is active unless *cancelxmit* becomes '1', in which case we immediately enter the *dma_xmit_cancel* mode. After performing whatever actions are necessary in this cancel mode, we again enter the normal mode. The contents of the normal and cancel modes are shown in Figure 21(a). The contents of *dma_xmit_normal* make it clear that the following actions are performed sequentially: (1) the starting memory address is stored after the appropriate activation signal is received, (2) the 16-bit destination address is received over the system bus and sent to another process as two bytes, (3) another 16-bit data is received over the system bus and the first byte of this, representing the data length, is sent to another process, (4) the appropriate number of data bytes are received and sent, and (5) the system bus is released. The contents of *dma_xmit_cancel* are also shown in the figure. Note that its statements are not enclosed in a loop, so after reaching the last statement, the behavior is complete so we traverse a TOC arc back to the beginning of *dma_xmit_normal*.

Trying to capture the same behavior with an HDL is difficult. The difficulty results from there being no constructs to easily capture the desired behavior of jumping to the cancel statements whenever *cancelxmit='0'*. One of two modeling methods is usually employed in an attempt to capture the behavior. In the first method, both the *normal* and *cancel* statements are placed in a single process. Each statement in the *normal* section of code is followed by a check for *cancelxmit='0'*; if true, a jump to the *cancel* code is executed. Each statement which waits on some event must also be modified to stop waiting if the *cancelxmit='0'* event occurs. Such modification of the *normal* statements obviously clutters the description and detracts from the readability of the code. (Actually, we need only add the additional statements after statements during which time can advance, which still clutters the description). To make matters worse, most HDL's support structured programming practices and thus do not contain a go-to statement. Therefore, one may have to enclose the *normal* statements in a loop, and implement a jump to the end of the statements by using a loop-exit statement.

A second modeling method, which is the method chosen by the HardwareC modelers, is shown in Figure 21(b). In this method, the code is first divided into blocks of statements, with each block becoming a branch in a case statement. At the end of each block, a variable is assigned a value indicating which is the next block to execute. The entire case statement is enclosed in a loop which exits when *cancelxmit='0'*. Thus *cancelxmit* is checked after executing each block, which is similar to the check accomplished by the first method described above. The advantage over the first method is that less code is needed to perform the check. However, there are several disadvantages to this model over the SpecChart model. One disadvantage is that the sequential series of actions being performed (get the starting address, receive the destination address, etc.) is embedded in a loop and case statement, making it necessary to mentally execute the model to understand the order of the actions. In the SpecChart model, the sequentiality of the actions is explicit; no loops, case branches,

```
behavior
{
  name { dma_xmit_normal}
  code
  {
    loop

       if (restart = '0') then
         -- normal start
         starting := '1';
         wait until txstart='1';
         startingpc := txaddress;
       else
         -- restart after cancel
         wait until txrestart='1';
         restart :='0';
       end if;
       pc := startingpc;
       -- get control of the system bus
       Bhold <= '1';
       wait until Bholdak='1';

       -- send DESTINATION
       starting := '0';
       Baddr <= pc; Brd <= '1'; rd := '1';
       wait until Bwait='0';
       data := Bdata; Brd <= '0'; rd := '0';
       send(data(15 downto 8), DMAxmit);
       send(data(7 downto 0), DMAxmit);

       -- send LENGTH
       pc := pc + 2;
       Baddr <= pc; Brd <= '1'; rd := '1';
       wait until Bwait='0';
       data := Bdata; Brd <= '0'; rd := '0';
       pc := pc + 2;
       len := data(15 downto 8);
       send(len, DMAxmit);

       -- send DATA
       for i in 1 to len loop
         if odd(i) then
           send(data(7 downto 0), DMAxmit);
         else
           Baddr <= pc; Brd <= '1'; rd := '1';
           wait until Bwait='0';
           data := Bdata; Brd <= '0'; rd := '0';
           pc := pc + 2;
           send(data(15 downto 8), DMAxmit);
         end if;
       end loop;

       -- END
       Bhold <= '0';
       wait until Bholdak='0';
       -- free the buses
       Baddr <= null; Brd <= null;

     end loop;
   }
 }


behavior
{
  name { dma_xmit_cancel }
  code
  {
    if (rd='1' and starting='0') then
      wait until Bwait='0';
      Brd <= '0'; rd := '0';
    end if;
    if (starting='0') then
      Bhold <= '0';
      wait until Bholdak='0';
    end if;
    Baddr <= null; Brd <= null;
    restart := '1';
  }
}
```

(a) SpecChart

```
dma_xmit : process
begin
  state := START;
  while (cancelxmit='0') loop
    case (state) is
      when START =>
          wait until txstart='1';
          startingpc := txaddress;
          pc := startingpc;
          Bhold <= '1';
          wait until Bholdak='1';
          Baddr <= pc; Brd <= '1'; rd := '1';
          state := DEST1;
      when RESTART
          wait until txrestart='1';
          pc := startingpc;
          Bhold <= '1';
          wait until Bholdak='1';
          Baddr <= pc; Brd <= '1'; rd := '1';
          state := DEST1;
      when DEST1 =>
          wait until Bwait='0';
          data := Bdata; Brd <= '0'; rd := '0';
          send(data(15 downto 8), DMAxmit);
          state := DEST2;
      when DEST2 =>
          send(data(15 downto 8), DMAxmit);
          state := LENGTH;
      when LENGTH =>
          -- send LENGTH
          pc := pc + 2;
          Baddr <= pc; Brd <= '1'; rd := '1';
          wait until Bwait='0';
          data := Bdata; Brd <= '0'; rd := '0';
          pc := pc + 2;
          len := data(15 downto 8);
          send(len, DMAxmit);
          counter := 0;
          if (len /= 0) then
            state := DATA;
          else
            state := END;
          end if;
      when DATA =>
          send(data(7 downto 0), DMAxmit);
          if (counter < len) then
            counter := counter + 1;
            state := DATA2;
          else
            state := END;
          end if;
      when DATA2 =>
          Baddr <= pc; Brd <= '1'; rd := '1';
          wait until Bwait='0';
          data := Bdata; Brd <= '0'; rd := '0';
          pc := pc + 2;
          send(data(15 downto 0), DMAxmit);
          if (counter < len) then
            counter := counter + 1;
            state := DATA;
          else
            state := END;
          end if;
      when END =>
          Bhold <= '0';
          wait until Bholdak='0';
          -- free the buses
          Baddr <= null; Brd <= null;
    end case;
  end loop
  if (rd='1' and state/=START) then
    wait until Bwait='0';
    Brd <= '0'; rd := '0';
  end if;
  if (state/=START) then
    Bhold <= '0';
    wait until Bholdak='0';
  end if;
  Baddr <= null; Brd <= null;
  state := RESTART;
end process;
```

(b) VHDL process

Figure 21: SpecChart and VHDL descriptions of *dma_xmit*

or state variables have been added. A second disadvantage is that when *cancelxmit='0'*, no action is taken until the beginning of the loop is reached. This means that, for example, we might continue waiting for *Bholdak='1'* even though bus access is no longer required, and only after access is granted do we then release the bus. In the SpecChart model, the waiting would be immediately terminated. These disadvantages arise when trying to describe an exceptional event using an HDL that provides only traditional sequential programming constructs.

10 Conclusion

The SpecCharts language was developed by modeling several systems. Through many examples, this report demonstrates the usefulness of the SpecCharts language for specifying system behavior while pointing out the weaknesses of existing languages. The language possesses an underlying model supporting behavioral hierarchy, which was shown to be extremely useful in many of the examples. In addition, its use of state-transition and programming constructs provide for concise and comprehensible descriptions. SpecChart specifications can serve as documentation, can be simulated to observe behavior, and can serve as a single format for representing a design throughout refinement/synthesis into a set of interconnected chips.

11 Acknowledgments

This work was supported by the National Science Foundation (grant #MIP-8922851) and the Semiconductor Research Corporation (grant #89-DJ-146). We are grateful for their support.

12 References

[1] S. Narayan, F. Vahid, and D. Gajski, "System Specification and Synthesis with the SpecCharts Language," in *Proc. of the International Conference on Computer-Aided Design*, 1991.

[2] F. Vahid, S. Narayan, D. Gajski, "SpecCharts : A Language for System Level Synthesis," in *Proc. of the 10th Intl. Symposium on Computer Hardware Decription Languages*, 1991.

[3] D. Harel, " Statecharts : A Visual Formalism for Complex Systems," *Science of Computer Programming 8*, 1987.

[4] C. Hoare, "Communicating Sequential Processes," *Comm. of the ACM*, August 1978.

[5] S. Narayan and F. Vahid, "Translating SpecCharts to VHDL." UC Irvine, TR 90-21, 1990.

[6] S. Narayan, F. Vahid, and D. Gajski, "Translating System Specifications to VHDL," in *Proc. of the European Conference on Design Automation*, 1991.

[7] F. Vahid and D. Gajski, "Obtaining Functionally Equivalent Simulations Using VHDL and a Time-shift Transformation," in *Proc. of the International Conference on Computer-Aided Design*, 1991.

[8] J. Armstrong, "Chip Level Modeling Using VHDL." Prentice-Hall, 1989.

[9] J. Lis and D. Gajski, "Structured Modeling for VHDL Synthesis." UC Irvine, TR 89-14, 1989.

[10] "DRACO Engineering Report." Rockwell International, 1989.

[11] I-Logix Corporation, "ExpressVHDL: UART Example." VHDL International Users' Forum, 1991.

[12] D. Thomas and P. Moorby, "The Verilog Hardware Description Language." Kluwer Academic Publishers, 1991.

[13] *Intel Microprocessor and Peripheral Handbook*, 1987.

[14] R. Gupta and G. D. Micheli, "System-level Synthesis using Re-programmable Components," in *Proc. of the European Conference on Design Automation*, 1992.

A Controlled Counter Appendix

A.1 Controlled Counter Textual SpecChart

```
state
{
  name {   Clear  }
  code
  {
    CNT <=  B"0000" after 5 ns;
  }
}
state
{
  name {   Controlled_counter  }
  declarations
  {
    use cctypes;
    use bit_functions;
    port CLK: in bit ;
    port STRB: in bit ;
    port CON: in bit_vector (1 downto 0) ;
    port DATA: in nibble ;
    port CNT_OUT: out nibble ;
    signal CONSIG : nibble ;
    signal LIM : nibble ;
    signal CNT : nibble ;
    signal EN : boolean ;
    function rising(signal s :  bit ) return boolean is
      begin
        return (s = '1' and s'event);
      end;
  }
  concurrent substates
  {
    Counter : ;
    Decode : ;
    Load_limit : ;
    Update_output : ;
    Update_enable : ;
  }
}
state
{
  name {   Count  }
  sequential substates
  {
   wait_state : (EI, CONSIG[2] = '1' and EN and rising(CLK), Count_up),
                (EI, CONSIG[3] = '1' and EN and rising(CLK), Count_down);
    Count_up : (EOC, true, wait_state);
    Count_down : (EOC, true, wait_state);
  }
}
state
{
  name {   Count_down  }
  code
  {
    CNT <=  SUB(CNT,B"0001") after 12 ns;
  }
}
state
{
  name {   Count_up  }
  code
  {
    CNT <=  ADD(CNT,B"0001") after 12 ns;
  }
}
state
{
  name {   Counter  }
  sequential substates
  {
    Count : (EI, CONSIG[0] = '1', Clear);
    Clear : (EI, not (CONSIG[0] = '1') , Count);
  }
}
state
{
  name {   Decode  }
  declarations
  {
    variable CONREG: bit_vector (1 downto 0);
  }
  code
  {
    loop
      wait  on STRB until STRB = '1';
      CONREG := CON;
      case CONREG is
        when "00" =>
          CONSIG <=  B"0001" after 5 ns;
        when "01" =>
          CONSIG <=  B"0010" after 5 ns;
        when "10" =>
          CONSIG <=  B"0100" after 5 ns;
        when "11" =>
          CONSIG <=  B"1000" after 5 ns;
      end case;
    end loop ;
  }
}
state
{
  name {   Load_limit }
  code
  {
    loop
      wait  on STRB until STRB = '0' and CONSIG[1] = '1';
      LIM <=  DATA after 10 ns;
    end loop ;
  }
}
state
{
  name {   Update_enable  }
  code
  {
    loop
      EN <=  CNT /= LIM after 10 ns;
      wait  on LIM,CNT;
    end loop ;
  }
}
state
{
  name {   Update_output  }
  code
  {
    loop
      CNT_OUT <=  CNT;
      wait  on CNT;
    end loop ;
  }
}
state
{
  name {   wait_state  }
  code
  {
    null;
  }
}
```

A.2 Controlled Counter Test Stimulus File

```
-----------------------------------------------------------
--file: sim.vhd
--authors: Frank Vahid, Sanjiv Narayan
--desc: provides simple functionality verification
--of the Controlled counter.  uses bit_vector
--inputs, requires bit functions package
--notes: * This file was originally written for models
--using integers instead of bit vectors.  It
--has been converted to work for bit vectors, but
--therefore looks a little funny since it tracks
--the integers and converts them, and vice-versa.
--* uses 'downto' bit vector direction, the agreed
--upon CADLAB standard.
--
--date: 6/20/90
-----------------------------------------------------------
use work.bit_functions.all;
entity E is
end;
architecture A of E is
  component Controlled_counterE
    port (
        CLK      : in bit;
        STRB     : in bit;
        CON      : in bit_vector(0 to 1);
        DATA     : in bit_vector(0 to 3);
        CNT_OUT  : out bit_vector(0 to 3)
  );
  end component;
  signal CLK       : bit;
  signal STRB      : bit;
  signal CON       : integer range 0 to 3;
  signal DATA      : integer range 0 to 15;
  signal CNT_OUT   : integer range 0 to 15;
  signal CONbv     : bit_vector(0 to 1);
  signal DATAbv    : bit_vector(0 to 3);
  signal CNT_OUTbv : bit_vector(0 to 3);
  for all : Controlled_counterE
     use entity work.Controlled_counterE(Controlled_counterA);
begin
   CC : Controlled_counterE port map (CLK, STRB, CONbv, DATAbv,
CNT_OUTbv);
  --track the integers/bit_vectors, convert to bit_vectors/integers
  CONbv <= INT_TO_BIN(CON,2);
  DATAbv <= INT_TO_BIN(DATA,4);
  CNT_OUT <= BIN_TO_INT(CNT_OUTbv);
  process
  begin
    wait for 1 ns;
    CLK <= transport '0';
    wait for 49 ns;
    CLK <= transport '1';
  end process;
  process
  begin
    wait for 30 ns;
 --start off with simple test of reset, count up, and count down, and limit
    --reset the counter
    CON <= 0;
    STRB <= '1' after 10 ns, '0' after 20 ns;
    wait for 50 ns;
    assert (CNT_OUT=0) report "ERROR1: CNT_OUT not reset to 0";
    --t=80
    --load the LIMIT
    DATA <= 2;
    CON <= 1;
    STRB <= '1' after 10 ns, '0' after 20 ns;
    wait for 50 ns;
    --t=130
    --count up
    CON <= 2;
    STRB <= '1' after 10 ns, '0' after 20 ns;
    wait for 50 ns;
    assert (CNT_OUT=1) report "ERROR2: CNT_OUT not incremented
to 1";
    --t=180
    --count up again
    wait for 50 ns;
    assert (CNT_OUT=2) report "ERROR3: CNT_OUT not incremented
to 2";
    --t=230
    --count up, should not increment since hit limit
    wait for 50 ns;
    assert (CNT_OUT=2) report "ERROR4: CNT_OUT should have hit
limit at 2";
    --t=280
    --count down, should not decrement since hit limit
    CON <= 3;
    STRB <= '1' after 10 ns, '0' after 20 ns;
    wait for 50 ns;
    assert (CNT_OUT=2) report "ERROR5: CNT_OUT at limit, shouldn't
change";
    --t=330
    --load the LIMIT
    DATA <= 0;
    CON <= 1;
    STRB <= '1' after 10 ns, '0' after 20 ns;
    wait for 50 ns;
    --t=380
    --count down
    CON <= 3;
    STRB <= '1' after 10 ns, '0' after 20 ns;
    wait for 50 ns;
    assert (CNT_OUT=1) report "ERROR6: CNT_OUT not decremented
to 1";
    --t=430
 --do some extensive testing of the counter's limit handling
    --set limit to 13
    DATA <= 13;
    CON <= 1;
    STRB <= '1' after 10 ns, '0' after 20 ns;
    wait for 50 ns;
    --t=480
    --reset the counter
    CON <= 0;
    STRB <= '1' after 10 ns, '0' after 20 ns;
    wait for 50 ns;
    assert (CNT_OUT=0) report "ERROR7: CNT_OUT not reset to 0";
    --t=530
    --count up to 13
    CON <= 2;
    STRB <= '1' after 10 ns, '0' after 20 ns;
    for i in 1 to 13 loop
      wait for 50 ns;
    end loop;
    --t=1180
    assert (CNT_OUT=13) report "ERROR8: CNT_OUT not up to 13";
    --count up, should not increment since hit limit
    wait for 50 ns;
    assert (CNT_OUT=13) report "ERROR9: CNT_OUT should have hit
limit at 13";
    --t=1230
    --count up, should not increment since hit limit
    wait for 50 ns;
    assert (CNT_OUT=13) report "ERROR10: CNT_OUT should have hit
limit at 13";
    --t=1280
    --change limit to 15
    DATA <= 15;
    CON <= 1;
    STRB <= '1' after 10 ns, '0' after 20 ns;
    wait for 50 ns;
    --t=1330
    --count up
    CON <= 2;
    STRB <= '1' after 10 ns, '0' after 20 ns;
```

```
    wait for 50 ns;
   assert (CNT_OUT=14) report "ERROR11: CNT_OUT didn't increment to 14";
    --t=1380
    --count up
    wait for 50 ns;
   assert (CNT_OUT=15) report "ERROR12: CNT_OUT didn't increment to 15";
    --t=1430
    --count up, should not increment since hit limit
    wait for 50 ns;
    assert (CNT_OUT=15) report "ERROR13:CNT_OUT should have hit limit at 15";
    --t=1480
    --change limit to 7
    DATA <= 7;
    CON <= 1;
    STRB <= '1' after 10 ns, '0' after 20 ns;
    wait for 50 ns;
    --t=1530
    --count down, try counting below 7
    CON <= 3;
    STRB <= '1' after 10 ns, '0' after 20 ns;
    for i in 1 to 10 loop
      wait for 50 ns;
    end loop;
    assert (CNT_OUT=7) report "ERROR14: CNT_OUT not equal to 7";
    --t=2030
    --change limit to 0
    DATA <= 0;
    CON <= 1;
    STRB <= '1' after 10 ns, '0' after 20 ns;
    wait for 50 ns;
    --t=2080
    --count down, try counting below 8
    CON <= 3;
    STRB <= '1' after 10 ns, '0' after 20 ns;
    for i in 1 to 8 loop
      wait for 50 ns;
    end loop;
    assert (CNT_OUT=0) report "ERROR15: CNT_OUT not equal to 0";
    --t=2480
    --count up, should not increment since hit limit
    CON <= 2;
    STRB <= '1' after 10 ns, '0' after 20 ns;
    wait for 50 ns;
      assert (CNT_OUT=0) report "ERROR16: CNT_OUT should have stayed at 0";
    --t=2530
 --try counting beyond the range, i.e. above 15 and below 0
    --reset the counter
    CON <= 0;
    STRB <= '1' after 10 ns, '0' after 20 ns;
    wait for 50 ns;
    assert (CNT_OUT=0) report "ERROR17: CNT_OUT not reset to 0";
    --t=2580
    --change limit to 7
    DATA <= 7;
    CON <= 1;
    STRB <= '1' after 10 ns, '0' after 20 ns;
    wait for 50 ns;
    --t=2630
    --count up 1
    CON <= 2;
    STRB <= '1' after 10 ns, '0' after 20 ns;
    wait for 50 ns;
   assert (CNT_OUT=1) report "ERROR18: CNT_OUT not incremented to 1";
    --t=2680
    --count down
    CON <= 3;
    STRB <= '1' after 10 ns, '0' after 20 ns;
    wait for 50 ns;
    assert (CNT_OUT=0) report "ERROR19:CNT_OUT not decremented to 0";
    --t=2730
    --count down
    wait for 50 ns;
   assert (CNT_OUT=15) report "ERROR20: CNT_OUT not decremented to 15";
    --t=2780
    --count down
    wait for 50 ns;
   assert (CNT_OUT=14) report "ERROR21: CNT_OUT not decremented to 14";
    --t=2830
    --count up
    CON <= 2;
    STRB <= '1' after 10 ns, '0' after 20 ns;
    wait for 50 ns;
   assert (CNT_OUT=15) report "ERROR22: CNT_OUT not incremented to 15";
    --t=2880
    --count up
    CON <= 2;
    STRB <= '1' after 10 ns, '0' after 20 ns;
    wait for 50 ns;
    assert (CNT_OUT=0) report "ERROR23: CNT_OUT not incremented to 0";
    --t=2930
  wait;
  end process;
end A;
```

A.3 Controlled Counter Simulation Output

```
0 NS
  SMON3:    ACTIVE /E/CNT_OUT (value = 0)
40 NS
   SMON:   ACTIVE /E/STRB (value = '1')
  SMON1:    ACTIVE /E/CON (value = 0)
80 NS
   SMON:   ACTIVE /E/STRB (value = '1')
  SMON1:    ACTIVE /E/CON (value = 1)
  SMON2:    ACTIVE /E/DATA (value = 2)
90 NS
   SMON:   ACTIVE /E/STRB (value = '0')
130 NS
   SMON:   ACTIVE /E/STRB (value = '1')
  SMON1:    ACTIVE /E/CON (value = 2)
162 NS
  SMON3:    ACTIVE /E/CNT_OUT (value = 1)
212 NS
  SMON3:    ACTIVE /E/CNT_OUT (value = 2)
280 NS
   SMON:   ACTIVE /E/STRB (value = '1')
  SMON1:    ACTIVE /E/CON (value = 3)
330 NS
   SMON:   ACTIVE /E/STRB (value = '1')
  SMON1:    ACTIVE /E/CON (value = 1)
  SMON2:    ACTIVE /E/DATA (value = 0)
340 NS
   SMON:   ACTIVE /E/STRB (value = '0')
380 NS
   SMON:   ACTIVE /E/STRB (value = '1')
  SMON1:    ACTIVE /E/CON (value = 3)
412 NS
  SMON3:    ACTIVE /E/CNT_OUT (value = 1)
430 NS
   SMON:   ACTIVE /E/STRB (value = '1')
  SMON1:    ACTIVE /E/CON (value = 1)
  SMON2:    ACTIVE /E/DATA (value = 13)
440 NS
   SMON:   ACTIVE /E/STRB (value = '0')
490 NS
   SMON:   ACTIVE /E/STRB (value = '1')
  SMON1:    ACTIVE /E/CON (value = 0)
500 NS
  SMON3:    ACTIVE /E/CNT_OUT (value = 0)
530 NS
```

```
  SMON:   ACTIVE /E/STRB (value = '1')
  SMON1:   ACTIVE /E/CON (value = 2)
562 NS
  SMON3:   ACTIVE /E/CNT_OUT (value = 1)
612 NS
  SMON3:   ACTIVE /E/CNT_OUT (value = 2)
662 NS
  SMON3:   ACTIVE /E/CNT_OUT (value = 3)
712 NS
  SMON3:   ACTIVE /E/CNT_OUT (value = 4)
762 NS
  SMON3:   ACTIVE /E/CNT_OUT (value = 5)
812 NS
  SMON3:   ACTIVE /E/CNT_OUT (value = 6)
862 NS
  SMON3:   ACTIVE /E/CNT_OUT (value = 7)
912 NS
  SMON3:   ACTIVE /E/CNT_OUT (value = 8)
962 NS
  SMON3:   ACTIVE /E/CNT_OUT (value = 9)
1012 NS
  SMON3:   ACTIVE /E/CNT_OUT (value = 10)
1062 NS
  SMON3:   ACTIVE /E/CNT_OUT (value = 11)
1112 NS
  SMON3:   ACTIVE /E/CNT_OUT (value = 12)
1162 NS
  SMON3:   ACTIVE /E/CNT_OUT (value = 13)
1280 NS
  SMON:   ACTIVE /E/STRB (value = '1')
  SMON1:   ACTIVE /E/CON (value = 1)
  SMON2:   ACTIVE /E/DATA (value = 15)
1290 NS
  SMON:   ACTIVE /E/STRB (value = '0')
1330 NS
  SMON:   ACTIVE /E/STRB (value = '1')
  SMON1:   ACTIVE /E/CON (value = 2)
1362 NS
  SMON3:   ACTIVE /E/CNT_OUT (value = 14)
1380 NS
  SMON:   ACTIVE /E/STRB (value = '0')
1412 NS
  SMON3:   ACTIVE /E/CNT_OUT (value = 15)
1480 NS
  SMON:   ACTIVE /E/STRB (value = '1')
  SMON1:   ACTIVE /E/CON (value = 1)
  SMON2:   ACTIVE /E/DATA (value = 7)
1490 NS
  SMON:   ACTIVE /E/STRB (value = '0')
1530 NS
  SMON:   ACTIVE /E/STRB (value = '1')
  SMON1:   ACTIVE /E/CON (value = 3)
1562 NS
  SMON3:   ACTIVE /E/CNT_OUT (value = 14)
1612 NS
  SMON3:   ACTIVE /E/CNT_OUT (value = 13)
1662 NS
  SMON3:   ACTIVE /E/CNT_OUT (value = 12)
1712 NS
  SMON3:   ACTIVE /E/CNT_OUT (value = 11)
1762 NS
  SMON3:   ACTIVE /E/CNT_OUT (value = 10)
1812 NS
  SMON3:   ACTIVE /E/CNT_OUT (value = 9)
1862 NS
  SMON3:   ACTIVE /E/CNT_OUT (value = 8)
1912 NS
  SMON3:   ACTIVE /E/CNT_OUT (value = 7)
2030 NS
  SMON:   ACTIVE /E/STRB (value = '1')
  SMON1:   ACTIVE /E/CON (value = 1)
  SMON2:   ACTIVE /E/DATA (value = 0)
2040 NS
  SMON:   ACTIVE /E/STRB (value = '0')
2080 NS
  SMON:   ACTIVE /E/STRB (value = '1')
  SMON1:   ACTIVE /E/CON (value = 3)
2112 NS
  SMON3:   ACTIVE /E/CNT_OUT (value = 6)
2162 NS
  SMON3:   ACTIVE /E/CNT_OUT (value = 5)
2212 NS
  SMON3:   ACTIVE /E/CNT_OUT (value = 4)
2262 NS
  SMON3:   ACTIVE /E/CNT_OUT (value = 3)
2312 NS
  SMON3:   ACTIVE /E/CNT_OUT (value = 2)
2362 NS
  SMON3:   ACTIVE /E/CNT_OUT (value = 1)
2412 NS
  SMON3:   ACTIVE /E/CNT_OUT (value = 0)
2480 NS
  SMON:   ACTIVE /E/STRB (value = '1')
  SMON1:   ACTIVE /E/CON (value = 2)
2540 NS
  SMON:   ACTIVE /E/STRB (value = '1')
  SMON1:   ACTIVE /E/CON (value = 0)
2580 NS
  SMON:   ACTIVE /E/STRB (value = '1')
  SMON1:   ACTIVE /E/CON (value = 1)
  SMON2:   ACTIVE /E/DATA (value = 7)
2590 NS
  SMON:   ACTIVE /E/STRB (value = '0')
2630 NS
  SMON:   ACTIVE /E/STRB (value = '1')
  SMON1:   ACTIVE /E/CON (value = 2)
2662 NS
  SMON3:   ACTIVE /E/CNT_OUT (value = 1)
2680 NS
  SMON:   ACTIVE /E/STRB (value = '1')
  SMON1:   ACTIVE /E/CON (value = 3)
2712 NS
  SMON3:   ACTIVE /E/CNT_OUT (value = 0)
2730 NS
  SMON:   ACTIVE /E/STRB (value = '1')
  SMON1:   ACTIVE /E/CON (value = 3)
2762 NS
  SMON3:   ACTIVE /E/CNT_OUT (value = 15)
2780 NS
  SMON:   ACTIVE /E/STRB (value = '1')
  SMON1:   ACTIVE /E/CON (value = 3)
2812 NS
  SMON3:   ACTIVE /E/CNT_OUT (value = 14)
2830 NS
  SMON:   ACTIVE /E/STRB (value = '1')
  SMON1:   ACTIVE /E/CON (value = 2)
2862 NS
  SMON3:   ACTIVE /E/CNT_OUT (value = 15)
2880 NS
  SMON:   ACTIVE /E/STRB (value = '1')
  SMON1:   ACTIVE /E/CON (value = 2)
2912 NS
  SMON3:   ACTIVE /E/CNT_OUT (value = 0)
2962 NS
  SMON3:   ACTIVE /E/CNT_OUT (value = 1)
3012 NS
  SMON3:   ACTIVE /E/CNT_OUT (value = 2)
3062 NS
  SMON3:   ACTIVE /E/CNT_OUT (value = 3)
3112 NS
  SMON3:   ACTIVE /E/CNT_OUT (value = 4)
3162 NS
  SMON3:   ACTIVE /E/CNT_OUT (value = 5)
3212 NS
  SMON3:   ACTIVE /E/CNT_OUT (value = 6)
3262 NS
  SMON3:   ACTIVE /E/CNT_OUT (value = 7)
10000 NS
```

A.4 Controlled Counter SpecChart to VHDL Translator Output

```
use work.cc_pack.all;
use work.cctypes.all;
use work.bit_functions.all;
entity ccE is
   port(CLK : in bit ; STRB : in bit ; CON : in bit_vector (1 downto 0) ; DATA : in nibble ; CNT_OUT : out nibble );
end;
Architecture ccA of ccE is
  signal incc   : boolean := false;
  signal donecc   : boolean := false;
  type cc_nibble_RES is array (natural range <>) of nibble;
  function cc_nibble_RESfct( INPUT :  cc_nibble_RES ) return nibble is
  begin
         assert (INPUT'length = 1) report "overdriven signal, type: cc_nibble_RES" severity warning;
     return INPUT(0);
  end;
  signal CONSIG : cc_nibble_RESfct nibble register;
  signal LIM : cc_nibble_RESfct nibble register;
  signal CNT : cc_nibble_RESfct nibble register;
  type cc_boolean_RES is array (natural range <>) of boolean;
  function cc_boolean_RESfct( INPUT :  cc_boolean_RES ) return boolean is
  begin
         assert (INPUT'length = 1) report "overdriven signal, type: cc_boolean_RES" severity warning;
     return INPUT(0);
  end;
  signal EN : cc_boolean_RESfct boolean register;
  function rising(signal s :  bit ) return boolean is
  begin
     return (s = '1' and s'event);
  end;
  signal CNT_OUT_sig : cc_nibble_RESfct nibble register;
  signal inCounter : boolean :=false;
  signal doneCounter : boolean :=false;
  signal inDecode : boolean :=false;
  signal doneDecode : boolean :=false;
  signal inLoad_limit : boolean :=false;
  signal doneLoad_limit : boolean :=false;
  signal inUpdate_output : boolean :=false;
  signal doneUpdate_output : boolean :=false;
  signal inUpdate_enable : boolean :=false;
  signal doneUpdate_enable : boolean :=false;
begin
  cc: block
  begin
      Counter: block
         signal inCount : boolean :=false;
         signal doneCount : boolean :=false;
         signal inClear : boolean :=false;
         signal doneClear : boolean :=false;
      begin
         Count: block
            signal inwait_state : boolean :=false;
            signal donewait_state : boolean :=false;
            signal inCount_up : boolean :=false;
            signal doneCount_up : boolean :=false;
            signal inCount_down : boolean :=false;
            signal doneCount_down : boolean :=false;
         begin
            wait_state: block (inwait_state and not(inwait_state'stable))
            begin
               code: process
               begin
               if guard then
                  wait_state_Loop : loop
                     null;
                     donewait_state <= transport true;
                     wait  until not (inwait_state) ;
                     donewait_state <= transport false;
                     exit wait_state_Loop;
                  end loop wait_state_Loop;
               end if;
               wait  on guard;
               end process code;
            end block wait_state;
            Count_up: block (inCount_up and not(inCount_up'stable))
            begin
               code: process
                  variable REMAIN_TIME: time;
                  variable GLOBAL_TIME: time;
               begin
               if guard then
                  Count_up_Loop : loop
                     REMAIN_TIME := 0 fs;
                     CNT <=  CNT;
                     REMAIN_TIME := MAX(REMAIN_TIME,12 ns);
                     CNT <=  ADD(CNT,B"0001") after 12 ns;
                     wait  until not (inCount_up)  for REMAIN_TIME;
                     if (not inCount_up ) then
                        exit Count_up_Loop;
                     end if;
                     doneCount_up <= transport true;
                     wait  until not (inCount_up) ;
                     doneCount_up <= transport false;
                     exit Count_up_Loop;
                  end loop Count_up_Loop;
               end if;
               CNT <= transport null;
               wait  on guard;
               end process code;
            end block Count_up;
         Count_down: block (inCount_down and not(inCount_down'stable))
            begin
               code: process
                  variable REMAIN_TIME: time;
                  variable GLOBAL_TIME: time;
               begin
               if guard then
                  Count_down_Loop : loop
                     REMAIN_TIME := 0 fs;
                     CNT <=  CNT;
                     REMAIN_TIME := MAX(REMAIN_TIME,12 ns);
                     CNT <=  SUB(CNT,B"0001") after 12 ns;
                     assert false report "Counting down" severity note;
                     wait  until not (inCount_down)  for REMAIN_TIME;
                     if (not inCount_down ) then
                        exit Count_down_Loop;
                     end if;
                     doneCount_down <= transport true;
                     wait  until not (inCount_down) ;
                     doneCount_down <= transport false;
                     exit Count_down_Loop;
                  end loop Count_down_Loop;
               end if;
               CNT <= transport null;
               wait  on guard;
               end process code;
            end block Count_down;
            control: process begin
                  if (inCount and not(inCount'stable)) then
                     inwait_state <= transport true;
                  elsif (inCount=false and not(inCount'stable)) then
                     inwait_state <= transport false;
                     inCount_up <= transport false;
                     inCount_down <= transport false;
                       elsif (inwait_state and (CONSIG(2) = '1' and EN and rising(CLK))) then
                     inwait_state <= transport false;
                     inCount_up <= transport true;
                       elsif (inwait_state and (CONSIG(3) = '1' and EN and rising(CLK))) then
                     inwait_state <= transport false;
```

```
          inCount_down <= transport true;
        elsif (doneCount_up and (true)) then
          inCount_up <= transport false;
          inwait_state <= transport true;
        elsif (doneCount_down and (true)) then
          inCount_down <= transport false;
          inwait_state <= transport true;
      end if;
    wait until (not inCount'stable) or (inwait_state and (CONSIG(2)
= '1' and EN and rising(CLK))) or (inwait_state and (CONSIG(3) = '1' and
EN and rising(CLK))) or (doneCount_up and (true)) or (doneCount_down
and (true));
    end process control;
  end block Count;
  Clear: block (inClear and not(inClear'stable))
  begin
    code: process
      variable REMAIN_TIME: time;
      variable GLOBAL_TIME: time;
    begin
    if guard then
      Clear_Loop : loop
        REMAIN_TIME := 0 fs;
        CNT <= CNT;
        REMAIN_TIME := MAX(REMAIN_TIME,5 ns);
        CNT <= B"0000" after 5 ns;
        wait until not (inClear) for REMAIN_TIME;
        if (not inClear ) then
          exit Clear_Loop;
        end if;
        doneClear <= transport true;
        wait until not (inClear) ;
        doneClear <= transport false;
        exit Clear_Loop;
      end loop Clear_Loop;
    end if;
    CNT <= transport null;
    wait on guard;
    end process code;
  end block Clear;
  control: process begin
      if (inCounter and not(inCounter'stable)) then
        inCount <= transport true;
      elsif (inCount and (CONSIG(0) = '1')) then
        inCount <= transport false;
        inClear <= transport true;
      elsif (inClear and (not (CONSIG(0) = '1') )) then
        inClear <= transport false;
        inCount <= transport true;
    end if;
    wait until (not inCounter'stable) or (inCount and (CONSIG(0)
= '1')) or (inClear and (not (CONSIG(0) = '1') ));
  end process control;
end block Counter;
Decode: block (inDecode and not(inDecode'stable))
begin
  code: process
    variable CONREG: bit_vector (1 downto 0);
    variable REMAIN_TIME: time;
    variable GLOBAL_TIME: time;
  begin
  if guard then
    REMAIN_TIME := 0 fs;
    CONSIG <= CONSIG;
    loop
      GLOBAL_TIME := now;
      wait on STRB until STRB = '1';
      GLOBAL_TIME := now - GLOBAL_TIME;
     REMAIN_TIME := MAX(REMAIN_TIME - GLOBAL_TIME,0
fs);

      CONREG := CON;
      case CONREG is
        when "00" =>
          REMAIN_TIME := MAX(REMAIN_TIME,5 ns);
          CONSIG <= B"0001" after 5 ns;
        when "01" =>
          REMAIN_TIME := MAX(REMAIN_TIME,5 ns);
          CONSIG <= B"0010" after 5 ns;
        when "10" =>
          REMAIN_TIME := MAX(REMAIN_TIME,5 ns);
          CONSIG <= B"0100" after 5 ns;
        when "11" =>
          REMAIN_TIME := MAX(REMAIN_TIME,5 ns);
          CONSIG <= B"1000" after 5 ns;
      end case;
    end loop ;
    wait for REMAIN_TIME;
    doneDecode <= transport true;
    wait until not (inDecode) ;
    doneDecode <= transport false;
  end if;
  CONSIG <= transport null;
  wait on guard;
  end process code;
end block Decode;
Load_limit: block (inLoad_limit and not(inLoad_limit'stable))
begin
  code: process
    variable REMAIN_TIME: time;
    variable GLOBAL_TIME: time;
  begin
  if guard then
    REMAIN_TIME := 0 fs;
    LIM <= LIM;
    loop
      GLOBAL_TIME := now;
      wait on STRB until STRB = '0' and CONSIG(1) = '1';
      GLOBAL_TIME := now - GLOBAL_TIME;
     REMAIN_TIME := MAX(REMAIN_TIME - GLOBAL_TIME,0
fs);

      REMAIN_TIME := MAX(REMAIN_TIME,10 ns);
      LIM <= DATA after 10 ns;
    end loop ;
    wait for REMAIN_TIME;
    doneLoad_limit <= transport true;
    wait until not (inLoad_limit) ;
    doneLoad_limit <= transport false;
  end if;
  LIM <= transport null;
  wait on guard;
  end process code;
end block Load_limit;
      Update_output:    block   (inUpdate_output   and
not(inUpdate_output'stable))
begin
  code: process
    variable REMAIN_TIME: time;
    variable GLOBAL_TIME: time;
  begin
  if guard then
    REMAIN_TIME := 0 fs;
    CNT_OUT_sig <= CNT_OUT_sig;
    loop
      CNT_OUT_sig <= CNT;
      GLOBAL_TIME := now;
      wait on CNT;
      GLOBAL_TIME := now - GLOBAL_TIME;
     REMAIN_TIME := MAX(REMAIN_TIME - GLOBAL_TIME,0
fs);

    end loop ;
    wait for REMAIN_TIME;
    doneUpdate_output <= transport true;
    wait until not (inUpdate_output) ;
    doneUpdate_output <= transport false;
  end if;
  CNT_OUT_sig <= transport null;
  wait on guard;
```

```
        end process code;
    end block Update_output;
                    Update_enable:       block     (inUpdate_enable     and
not(inUpdate_enable'stable))
    begin
        code: process
          variable REMAIN_TIME: time;
          variable GLOBAL_TIME: time;
        begin
        if guard then
          REMAIN_TIME := 0 fs;
          loop
            REMAIN_TIME := MAX(REMAIN_TIME,10 ns);
            EN <=  CNT /= LIM after 10 ns;
            GLOBAL_TIME := now;
            wait  on LIM,CNT;
            GLOBAL_TIME := now - GLOBAL_TIME;
           REMAIN_TIME := MAX(REMAIN_TIME - GLOBAL_TIME,0
fs);
          end loop ;
          wait  for REMAIN_TIME;
          doneUpdate_enable <= transport true;
          wait  until not (inUpdate_enable) ;
          doneUpdate_enable <= transport false;
        end if;
        EN <= transport null;
        wait  on guard;
        end process code;
    end block Update_enable;
    control: process begin
          if (incc and not(incc'stable)) then
            inCounter <= transport true;
            inDecode <= transport true;
            inLoad_limit <= transport true;
            inUpdate_output <= transport true;
            inUpdate_enable <= transport true;
          end if;
        wait until (not incc'stable);
    end process control;
  end block cc;
CNT_OUT <= transport CNT_OUT_sig;
start: process begin
  incc <= transport true;
  wait;
end process start;
end ccA;
```

B UART Textual SpecChart

```
state
{
  name {   ALLOW_READ  }
  code
  {
    READ_ENABLE <=  '1';
  }
}
state
{
  name {   ECHO_ACTIVE  }
  code
  {
    loop
      wait  until (READ_ENABLE = '1');
      TX_HOLD_REG <=  RX_REG;
      LOAD <=  '1';
      wait  until (CLK = '1') and (not CLK'stable );
      LOAD <=  '0';
    end loop ;
  }
}
state
{
  name {   FIFO  }
  sequential substates
  {
    RX_EMPTY : (EI, (READ_ENABLE = '1'), RX_LOADED);
    RX_LOADED : (EI, (CSR[1 downto 0] = "01"), RX_EMPTY);
  }
}
state
{
  name {   NORMAL_RX_TX  }
  code
  {
    null;
  }
}
state
{
  name {   PARITY_ERROR  }
  code
  {
    PARITY_ERROR <=  '1';
  }
}
state
{
  name {   READ_BITS }
  code
  {
    for I in 0 to 7 loop
      wait  until (CLK = '1') and (not CLK'stable );
      RX_REG[I] <=  LINE_IN;
      PARITY_BIT := PARITY_BIT xor LINE_IN;
    end loop ;
  }
}
state
{
  name {   RECEIVE }
  sequential substates
  {
    WAIT_FOR_START : (EI, (LINE_IN = '1'), READ_BITS);
    READ_BITS : (EOC, (PARITY_BIT = LINE_IN), ALLOW_READ),
(EOC, (PARITY_BIT /= LINE_IN), PARITY_ERROR);
    ALLOW_READ : (EOC, true, WAIT_FOR_START);
    PARITY_ERROR : ;
  }
}
state
{
  name {   RECEIVER }
  declarations
  {
    variable PARITY_BIT: bit;
  }
  concurrent substates
  {
    RX_MODE : ;
    FIFO : ;
  }
}
state
{
  name {   RX_EMPTY }
  code
  {
    FIFOFULL <=  '0';
  }
}
state
{
  name {   RX_IDLE }
  code
  {
    null;
  }
}
state
{
  name {   RX_LOADED  }
  code
  {
    FIFOFULL <=  '1';
  }
}
state
{
  name {   RX_MODE  }
  sequential substates
  {
    RX_IDLE : (EI, (CSR(3) = '1'), RECEIVE);
    RECEIVE : (EI, (CSR(3) = '0'), RX_IDLE);
  }
}
state
{
  name {   TOP_LEVEL_UART  }
  declarations
  {
    port LINE_IN: in bit ;
    port LINE_OUT: out bit ;
    port DATA_IN: in bit_vector (7 downto 0) ;
    port CLK: in bit ;
    port CSR: in bit_vector (3 downto 0) ;
    port TX_PARITY_TYPE: in bit ;
    port PARITY_ERROR: out bit ;
    port FIFOFULL: out bit ;
    port TX_ACT: out bit ;
    subtype BYTE is bit_vector (7 downto 0);
    signal TX_HOLD_REG : BYTE ;
    signal TX_REG : BYTE ;
    signal RX_REG : BYTE ;
    signal READ_ENABLE : bit ;
    signal LOAD : bit ;
  }
  concurrent substates
  {
    TRANSMITTER : ;
```

```
      RECEIVER : ;
      UART_MODE : ;
   }
}
state
{
   name {    TRANSMIT  }
   declarations
   {
      variable PARITY_BIT: bit;
   }
   sequential substates
   {
      TRANSMIT_IDLE : (EI, (LOAD = '1'), TX_BYTE);
      TX_BYTE : (EOC, true, TX_PARITY);
      TX_PARITY : (EOC, true, TRANSMIT_IDLE);
   }
}
state
{
   name {    TRANSMITTER  }
   concurrent substates
   {
      TX_MODE : ;
      TX_HOLD : ;
   }
}
state
{
   name {    TRANSMIT_IDLE  }
   code
   {
      TX_ACT <=  '0';
      LINE_OUT <=  '0';
      PARITY_BIT := '0';
   }
}
state
{
   name {    TX_BYTE  }
   code
   {
      LINE_OUT <=  '1';
      TX_ACT <=  '1';
      TX_REG <=  TX_HOLD_REG;
      for I in 0 to 7 loop
         wait  until (CLK = '1') and (not CLK'stable );
         LINE_OUT <=  TX_REG[I];
                  PARITY_BIT   :=   PARITY_BIT   xor   TX_REG[I]   xor
TX_PARITY_TYPE;
      end loop ;
   }
}
state
{
   name {    TX_EMPTY  }
   code
   {
      LOAD <=  '0';
   }
}
state
{
   name {    TX_HOLD  }
   sequential substates
   {
      TX_EMPTY : (EI, (CSR[1 downto 0] = "10"), TX_LOADED);
      TX_LOADED : (EI, TX_REG'active, TX_EMPTY);
   }
}
state
{
   name {    TX_IDLE  }
   code
   {
      null;
   }
}
state
{
   name {    TX_LOADED  }
   code
   {
      TX_HOLD_REG <=  DATA_IN;
      LOAD <=  '1';
   }
}
state
{
   name {    TX_MODE  }
   sequential substates
   {
      TX_IDLE : (EI, (CSR(2) = '1'), TRANSMIT);
      TRANSMIT : (EI, (CSR(2) = '0'), TX_IDLE);
   }
}
state
{
   name {    TX_PARITY  }
   code
   {
      LINE_OUT <=  PARITY_BIT;
   }
}
state
{
   name {    UART_MODE  }
   sequential substates
   {
     NORMAL_RX_TX : (EI, (CSR[3 downto 2] = "11"), ECHO_ACTIVE);
     ECHO_ACTIVE : (EI, (CSR[3 downto 2] /= "11"), NORMAL_RX_TX);
   }
}
state
{
   name {    WAIT_FOR_START  }
   code
   {
      PARITY_ERROR <=  '0';
      READ_ENABLE <=  '0';
   }
}
```

C Pipelined Processor Textual SpecChart

```
state
{
name {   active_state  }
sequential substates
{
  init_wait : (EI, rising(CLK),  phase1);
  phase1    : (EI, falling(CLK), phase2);
  phase2    : (EI, rising(CLK),  phase1);}
}
state
{
name {   execute  }
declarations
{  variable dest    : integer;
   variable source : integer;
   variable src1   : WORDTYPE;          --alu operation register
   variable src2   : WORDTYPE;          --alu operation register
   variable i      : integer;
   variable ir_empty  : boolean;
}
code
{MEM <= null;
RFILE <= null;
IRQ <= null;
ir_empty := Qempty(IRQ);
if (not ir_empty or mem_access) then
  dest := B2I(DST);
  source := B2I(SRC);
  --new IR required?
  if not(mem_access) then
    wait for 1 fs;  --prevent simultaneous write with fetch
    IRQ <= IRQ;
    Qpop(IRQ, ir);
    IRQ <= null;
  end if;
  case (OPCODE) is
   when NOP => assert false report "NOP" severity note;
            null ;
   when BRA => assert false report "BRA" severity note;
            if (checkcond(CCODE) = '1') then
              pc <= dest;
              branch_taken <= true;
              settle();
            end if;
   when LD  =>
            if (mem_access=false) then
              mem_access <= true; --reserve next cycle
              settle();
            else
              assert false report "LD" severity note;
              RFILE <= RFILE;
              clearcondcode(psr);
              if (SRCTYPE='1') then
                RFILE[dest] <= I2B(source,WIDTH);
              else
                RFILE[dest] <= MEM[source];
              end if;
              settle();
              setcondcode(psr, "0" &RFILE[dest]);
              mem_access <= false;
              settle();
            end if;
   when STR =>
            if (mem_access=false) then
              mem_access <= true; --reserve next cycle
              settle();
            else
              assert false report "STR" severity note;
              MEM <= MEM;
              clearcondcode(psr);
              if (SRCTYPE='1') then
                MEM[dest] <= I2B(source,WIDTH);
              else
                MEM[dest] <= RFILE[source];
              end if;
              mem_access <= false;
              settle();
            end if;
   when ADD => assert false report "ADD" severity note;
            clearcondcode(psr);
            src1 := getsrc();
            src2 := getdst();
            result <= CARRY_ADD(src1,src2);
            settle();
            setcondcode(psr, result);
   when MUL => assert false report "MUL" severity note;
            clearcondcode(psr);
            src1 := getsrc();
            src2 := getdst();
            result <= "0" &(src1 * src2);
            settle();
            setcondcode(psr, result);
   when CMP => assert false report "CMP" severity note;
            clearcondcode(psr);
            src1 := getsrc();
            result <= COMP(src1);
            settle();
            setcondcode(psr, result);
   when SHF => clearcondcode(psr);
            src1 := getsrc();
            src2 := getdst();
            --src1[ADDRSIZE-1:0] is the amount to shift by
               --amount is twos complement form; positive-shift right,
neg-left
            --check high bit to see if positive ('0') or negative ('1')
            if (src1[ADDRSIZE-1] = '0') then
              assert false report "SHF RIGHT" severity note;
             i := B2I(src1[ADDRSIZE-1 downto 0]); --amount to shift
by

              result <= "0" &SHR0(src2, i);
              settle();
            else
              assert false report "SHF LEFT" severity note;
              i := B2I(TWOs_COMP(src1[ADDRSIZE-1 downto 0]));
              result <= "0" &SHL0(src2, i);
              settle();
            end if;
            setcondcode(psr, result);
   when ROT => assert false report "ROT" severity note;
            clearcondcode(psr);
            src1 := getsrc();
            src2 := getdst();
            if src1[ADDRSIZE-1] = '0' then
              src2 := ROTR(src2, source);
            else
              src2 := ROTL(src2, source);
            end if;
            result <= PAD(src2, WIDTH+1);
            settle();
            setcondcode(psr, result);
   when HLT => assert false report "Halt..." severity note;
            halt_found <= true;
            settle();
    when others =>  assert (false) report "Error : Illegal opcode" severity
note;
  end case;
end if;
}
}
```

```
state
{
name {    fetch   }
declarations
{
   variable ir_full : boolean;
}
code
{
   IRQ <= null;
   ir_full := Qfull(IRQ);
   if not ir_full and not mem_access then
     IRQ <= IRQ;
     Qpush(IRQ, MEM[pc]);
     settle();
     IRQ <= null;
   end if;
}
}
state
{
name {    init_wait   }
code
{null;
}
}
state
{
name {    loadmem   }
code
{
   MEM[0]  <= "00101000000000000000000000000001"; --LD R1, #0
   MEM[1]  <= "00100000000000001001000000000000"; --LD R2, NMBR
  MEM[2]  <= "00010010000000000000000000000100"; --STRT: BRA L1,
EVEN
   MEM[3]  <= "01001000000000000001000000000001"; --ADD R1, #1
  MEM[4]  <= "01111000000000000001000000000000"; --L1: SHF R2, #1
  MEM[5]  <= "00010011000000000000000000000111"; --BRA L2, ZERO
    MEM[6]   <= "00010101000000000000000000000010"; --BRA STRT,
ALW
  MEM[7]  <= "00110000000000000001000000001010"; --L2: STR RSLT,
R2
   MEM[8]  <= "10011111111111111111111111111111"; --HLT
   MEM[9]  <= "01010101010101011010101010101010"; --NMBR:555aaa
  MEM[10] <= "00000000000000000000000000000000"; --RSLT:00000000
}
}
state
{
name {    phase1   }
concurrent substates
{fetch : ;
execute : ;
write_result : ;}
}
state
{
name {    phase2   }
declarations
{   variable op   : integer;
}
code
{
   op   := B2I(OPCODE);
   --copy results, set condition codes
   if ( not(mem_access) and not(branch_taken) ) then  --copy results
      if ((op >= B2I(ADD)) and (op <= B2I(HLT))) then
         setcondcode(psr,result);
         wresult <= result;
         wir <= ir;
         result_ready <= true;
         settle();
      end if;
   end if;
   if (branch_taken) then                              --Set next pc
      pc <= B2I(DST);
      settle();
   elsif not(mem_access) then                          --no change for idle cycle
      pc <= pc + 1;
      settle();
   end if;
   if (branch_taken or halt_found) then                --Flush queue
      Qflush(IRQ);
      branch_taken <= false;
      settle();
   end if;
   if (halt_found) then
      assert false report "Halt" severity failure;
      halt_found <= false;
      settle();
   end if;
}
}
state
{
name {    reset_state   }
code
   Qflush(IRQ);
   pc           <= 0;
   mem_access   <= false;
   branch_taken <= false;
   halt_found   <= false;
   result_ready <= false;
   wait for 1 fs;
}
}
state
{
name {    sisc   }
declarations
{   port   reset   : in bit;                  --system reset
   port   CLK     : in bit;
   constant WIDTH    : integer := 32;
   constant ADDRSIZE : integer := 12;
   constant MEMSIZE  : integer := 4096;
   constant MAXREGS  : integer := 16;
   constant SBITS    : integer := 5;
   subtype WORDTYPE    is bit_vector(WIDTH-1 downto 0);
   type    MEMTYPE     is array(natural range <>) of WORDTYPE;
   subtype RESULTTYPE  is bit_vector(WIDTH downto 0);
   subtype STATUSTYPE  is bit_vector(SBITS-1 downto 0);
   subtype ADDRESSTYPE is integer range 0 to 2**(ADDRSIZE)-1 ;
   --declare registers and memory
   signal MEM     : MEMTYPE(20 downto 0);   --memory
   signal RFILE   : MEMTYPE(MAXREGS-1 downto 0);   --register file
   signal ir      : WORDTYPE;               --instruction register
   signal result  : RESULTTYPE;             --alu result register
   signal pc      : ADDRESSTYPE;            --program counter
   signal dir     : bit;                    --rotate direction
   signal psr     : STATUSTYPE;             --processor status register
   --general definitions
   --true, false predefined
   alias OPCODE : bit_vector(3 downto 0) is ir[31 downto 28];
   alias SRC    : bit_vector(11 downto 0) is ir[23 downto 12];
   alias DST    : bit_vector(11 downto 0) is ir[11 downto 0];
    alias SRCTYPE: bit is ir[27];  --source type, 0=reg (mem for LD),
1=imm
   alias DSTTYPE: bit is ir[26];  --destination type, 0=reg , 1=imm
   alias CCODE  : bit_vector(3 downto 0) is ir[27 downto 24];
   alias SRCNT  : bit_vector(11 downto 0) is ir[23 downto 12];
      --shift,rotate count -=left, +=right
```

```
--operand types
constant REGTYPE : bit := '0';
constant IMMTYPE : bit := '1';
--opcode for each instruction
constant NOP : bit_vector(3 downto 0) :="0000";
constant BRA : bit_vector(3 downto 0) :="0001";
constant LD  : bit_vector(3 downto 0) :="0010";
constant STR : bit_vector(3 downto 0) :="0011";
constant ADD : bit_vector(3 downto 0) :="0100";
constant MUL : bit_vector(3 downto 0) :="0101";
constant CMP : bit_vector(3 downto 0) :="0110";
constant SHF : bit_vector(3 downto 0) :="0111";
constant ROT : bit_vector(3 downto 0) :="1000";
constant HLT : bit_vector(3 downto 0) :="1001";
--condition code fields
alias CARRY  : bit is psr[0];
alias EVEN   : bit is psr[1];
alias PARITY : bit is psr[2];
alias ZERO   : bit is psr[3];
alias NEG    : bit is psr[4];
constant CCC : integer :=0;
constant CCE : integer :=1;
constant CCP : integer :=2;
constant CCZ : integer :=3;
constant CCN : integer :=4;
constant CCA : integer :=5;
constant RIGHT : bit :='0';
constant LEFT  : bit :='1';
--functions for ALU operands and result
function getsrc() return bit_vector is
begin
  if SRCTYPE = REGTYPE then
    return(RFILE[B2I(SRC)]);
  else
    return(PAD(SRC,WIDTH));
  end if;
end;
function getdst() return bit_vector is
begin
  if DSTTYPE = REGTYPE then
    return(RFILE[B2I(DST)]);
  else
    assert false report "Error: Immediate data can not be destination";
  end if;
end;
--functions/procedures for condition codes
function checkcond (ccode : bit_vector) return bit is
begin
  case (B2I(ccode)) is
    when CCC => return(CARRY);
    when CCE => return(EVEN);
    when CCP => return(PARITY);
    when CCZ => return(ZERO);
    when CCN => return(NEG);
    when CCA => return('1');
    when others => assert false report "Error: invalid status code";
  end case;
end;
procedure clearcondcode(signal cond : inout STATUSTYPE) is
begin
  cond <= "00000";
  settle(); --wait delta time for value to settle
end;
procedure setcondcode (signal cond : inout STATUSTYPE;
                       res : in RESULTTYPE) is
begin
  cond[0] <= res[WIDTH];          --CARRY
  cond[1] <= not(res(0));         --EVEN
  cond[2] <= ODD_PARITY(res);     --PARITY
  if (SUM(res)=0) then
    cond[3] <= '1';               --ZERO
  else
    cond[3]<= '0';
  end if;
  cond[4] <= res[WIDTH-1];        --NEG
  settle();
end;
--pipeline decls
--Declare additional registers for pipeline control
signal wir     : WORDTYPE;              --Instr reg for write stage
signal wresult : RESULTTYPE;            --alu result register for write stage
--various flags
signal mem_access   : boolean;
signal branch_taken : boolean;
signal halt_found   : boolean;
signal result_ready : boolean;
signal  do_fetch         : boolean;
signal  do_execute       : boolean;
signal  do_write_results : boolean;
alias WOPCODE  : bit_vector(3 downto 0) is wir[31 downto 28];
alias WDST     : bit_vector(11 downto 0) is wir[11 downto 0];
alias WDSTTYPE : bit is wir[26];
type Qtype is queue(3) of WORDTYPE;
signal IRQ : Qtype;  --instruction queue declaration
}
sequential substates
{loadmem     : (EOC, true, reset_state);  /* read in the program */
reset_state  : (EI, reset='0', active_state);
active_state : (EI, reset='1', reset_state);}
}
state
{
name {   write_result   }
declarations
{  variable dest : integer;
   variable op   : integer;
}
code
{
MEM <= null;
RFILE <= null;
if result_ready then
  MEM <= MEM;
  RFILE <= RFILE;
  dest := B2I(WDST);
  op   := B2I(WOPCODE);
  if ((op >= B2I(ADD)) and (op <= B2I(HLT))) then
    if (WDSTTYPE = REGTYPE) then
      RFILE[dest] <= wresult[(result'high-1) downto 0];
    else
      MEM[dest] <= wresult[(result'high-1) downto 0];
    end if;
  end if;
end if;
}
}
```

D Answering Machine Textual SpecChart

```
state
{
  name {  ANSWER  }
  sequential substates
  {
      PLAY_ANNOUNCEMENT : (EI, button_p = "0001", REMOTE_OPERATION),
                          (EI, dialtone_p = '1' and dialtone_p'event, HANGUP),
                          (EOC, true, RECORD_MSG);
    REMOTE_OPERATION  : (EOC, true, HANGUP);
   RECORD_MSG       : (EI, button_p = "0001", REMOTE_OPERATION),
                          (EOC, true, HANGUP);
    HANGUP            : (EOC, true, stop);
  }
}
state
{
  name {  ANS_MACHINE  }
  declarations
  {
    --switches on the machine
    port system_on_p: in bit ;
    port tollsaver_p: in bit ;
    --touch sensitive buttons on the machine
    port machine_on_pushed_p: in bit ;
    port play_pushed_p: in bit ;
    port fwd_pushed_p: in bit ;
    port rew_pushed_p: in bit ;
    port memo_pushed_p: in bit ;
    port stop_pushed_p: in bit ;
    port play_msgs_pushed_p: in bit ;
    port hear_ann_pushed_p: in bit ;
    port rec_ann_pushed_p: in bit ;
    --interface to other devices internal to the machine
    port tape_count_p: in integer ;
    port ring_p: in bit ;
    port dialtone_p: in bit ;
    port button_p: in bit_vector (3 downto 0) ;
    port ann_done_p: in bit ;
    port produce_beep_p: out bit ;
    port offhook_p: out bit ;
    port ann_play_p: out bit ;
    port ann_rec_p: out bit ;
    port num_msgs_p: out integer range 0 to 31 ;
    port tape_play_p: out bit ;
    port tape_fwd_p: out bit ;
    port tape_rew_p: out bit ;
    port tape_rec_p: out bit ;
    type rf4 is array (1 to 4) of bit_vector (3 downto 0);
    signal user_code : rf4 ;
    signal machine_on : bit ;
    signal machine_on_toggle : bit ;
    signal num_msgs : integer range 0 to 31 ;
    signal machine_button_pushed : bit ;
  }
  constraints
  {
    area_per_chip <= 7500 sqmm (1);
    num_chips <= 3 chips (1);
    pins_per_chip <= 40 pins (1);
  }
  concurrent substates
  {
    MAIN : ;
    MACHINE_ON_TOGGLER : ;
    CONC_ASGNS1 : ;
    CONC_ASGNS2 : ;
  }
}
state
{
  name {  BEEP_NUM_MSGS  }
  declarations
  {
    variable msg_count: integer range 0 to 31;
  }
  code
  {
    msg_count := 0;
    wait  for 5 fs;
    while (msg_count < num_msgs) loop
      produce_beep_p <=  '1';
      wait  for 1 fs;
      produce_beep_p <=  '0';
      wait  for 1 fs;
      msg_count := msg_count + 1;
    end loop ;
  }
}
state
{
  name {  CHECK_ID  }
  declarations
  {
    variable entered_code: rf4;
    variable i: integer range 1 to 5;
  }
  code
  {
    i := 1;
    while i <= 4 loop
      wait  until button_p /= "1111" and button_p'event;
      assert false report "CHECK_ID: button pushed" severity note;
      entered_code[i] := button_p;
      i := i + 1;
    end loop ;
    if (entered_code[1] = user_code[1]) and
      (entered_code[2] = user_code[2]) and
      (entered_code[3] = user_code[3]) and
      (entered_code[4] = user_code[4]) then
      code_ok <=  '1';
    else
      code_ok <=  '0';
    end if;
  }
}
state
{
  name {  CONC_ASGNS1  }
  estimates
  {
    area = 100 sqmm;
  }
  code
  {
   --this is used to make up for the lack of a concurrent signal assignment
    --statement
    loop
      wait  on play_msgs_pushed_p,rec_ann_pushed_p,hear_ann_pushed_p,
            stop_pushed_p,memo_pushed_p,rew_pushed_p,fwd_pushed_p,
            play_pushed_p;
      if (play_pushed_p = '1' and play_pushed_p'event) or
        (fwd_pushed_p = '1' and fwd_pushed_p'event) or
        (rew_pushed_p = '1' and rew_pushed_p'event) or
        (memo_pushed_p = '1' and memo_pushed_p'event) or
        (stop_pushed_p = '1' and stop_pushed_p'event) or
        (hear_ann_pushed_p = '1' and hear_ann_pushed_p'event) or
        (rec_ann_pushed_p = '1' and rec_ann_pushed_p'event) or
        (play_msgs_pushed_p = '1' and play_msgs_pushed_p'event) then
        machine_button_pushed <=  '1';
      else
        machine_button_pushed <=  '0';
```

```
      end if;
    end loop ;
  }
}
state
{
  name {    CONC_ASGNS2   }
  estimates
  {
    area = 100 sqmm;
  }
  code
  {
   --this is used to make up for the lack of a concurrent signal assignment
    --statement
    loop
      wait  on num_msgs;
      if (num_msgs'event) then
        num_msgs_p <=  num_msgs;
      end if;
    end loop ;
  }
}
state
{
  name {    DECODE_PUSHED_BUTTON   }
  code
  {
    null;
  }
}
state
{
  name {    ERASE_MSGS   }
  code
  {
    tape_rew_p <=  '1';
    if not (tape_count_p = 0)  then
      wait  until tape_count_p = 0;
    end if;
    tape_rew_p <=  '0';
    terminal_tape_count <=  0;
  }
}
state
{
  name {    FWD   }
  code
  {
    tape_fwd_p <=  '1';
  }
}
state
{
  name {    HANDLE_FWD_PUSHED   }
  code
  {
    tape_fwd_p <=  '1';
    num_msgs <=  0;
  }
}
state
{
  name {    HANDLE_HEAR_ANN_PUSHED   }
  code
  {
    ann_play_p <=  '1';
    wait  until ann_done_p = '1';
    ann_play_p <=  '0';
  }
}
state
{
  name {    HANDLE_MEMO_PUSHED   }
  code
  {
    produce_beep_p <=  '1';
    wait  for 1 fs;
    produce_beep_p <=  '0';
    tape_rec_p <=  '1';
    wait  until memo_pushed_p = '0' for 1000 fs;
    --put a beep on the tape to indicate end of message
    produce_beep_p <=  '1';
    wait  for 1 fs;
    produce_beep_p <=  '0';
    num_msgs <=  num_msgs + 1;
    tape_rec_p <=  '0';
  }
}
state
{
  name {    HANDLE_PLAY_MSGS_PUSHED   }
  code
  {
    terminal_tape_count <=  tape_count_p;
    --rewind to beginning of tape
    tape_rew_p <=  '1';
    if not (tape_count_p = 0)  then
      wait  until tape_count_p = 0;
    end if;
    tape_rew_p <=  '0';
    --play until end of last message
    tape_play_p <=  '1';
    if (tape_count_p < terminal_tape_count) then
      wait  until tape_count_p = terminal_tape_count;
    end if;
    tape_play_p <=  '0';
  }
}
state
{
  name {    HANDLE_PLAY_PUSHED   }
  code
  {
    tape_play_p <=  '1';
    num_msgs <=  0;
  }
}
state
{
  name {    HANDLE_REC_ANN_PUSHED   }
  code
  {
    --give the user some time to prepare
    wait  for 50 fs;
    produce_beep_p <=  '1';
    wait  for 1 fs;
    produce_beep_p <=  '0';
    wait  for 0 fs;
    ann_rec_p <=  '1';
    wait  until ann_done_p = '1';
    ann_rec_p <=  '0';
    produce_beep_p <=  '1';
    wait  for 1 fs;
    produce_beep_p <=  '0';
  }
}
state
{
  name {    HANDLE_REW_PUSHED   }
  code
  {
    num_msgs <=  0;
    tape_rew_p <=  '1';
    if (tape_count_p /= 0) then
      wait  until tape_count_p = 0;
```

```
      end if;
      tape_rew_p <= '0';
    }
}
state
{
  name {   HANDLE_STOP_PUSHED   }
  code
  {
      num_msgs <= 0;
      tape_play_p <= '0';
      tape_fwd_p <= '0';
      tape_rew_p <= '0';
      tape_rec_p <= '0';
  }
}
state
{
  name {   HANGUP   }
  code
  {
      offhook_p <= '0';
      ann_play_p <= '0';
      if (toggle_on_hangup = '1') then
         toggle_on_hangup <= '0';
         machine_on_toggle <= '1';
         wait  for 1 fs;
         machine_on_toggle <= '0';
      end if;
  }
}
state
{
  name {   HEAR_ANN   }
  code
  {
      ann_play_p <= '1';
      wait  until ann_done_p = '1';
      ann_play_p <= '0';
  }
}
state
{
  name {   HEAR_MSGS_CMDS   }
  declarations
  {
      signal tape_end_count : bit_vector (7 downto 0) ;
  }
  sequential substates
  {
      WAIT_FIRST_CMD : (EOC, true, RESET_AND_DECODE);
               RESET_AND_DECODE: (EOC, button_p = "0010",
PLAY_ALL_MSGS),
                   (EOC, button_p = "0011", PLAY),
                   (EOC, button_p = "0100", FWD),
                   (EOC, button_p = "0101", REW),
                   (EOC, button_p = "0110", STOP_TAPE),
                   (EOC, button_p = "0111", BEEP_NUM_MSGS),
                   (EOC, button_p = "1000", stop);
      PLAY_ALL_MSGS
               :  (EI, button_p /= "1111" and button_p'event, RE-
SET_AND_DECODE);
      PLAY
               :  (EI, button_p /= "1111" and button_p'event, RE-
SET_AND_DECODE);
      FWD
               :  (EI, button_p /= "1111" and button_p'event, RE-
SET_AND_DECODE);
      REW
               :  (EI, button_p /= "1111" and button_p'event, RE-
SET_AND_DECODE);
      STOP_TAPE
               :  (EI, button_p /= "1111" and button_p'event, RE-
SET_AND_DECODE);
      BEEP_NUM_MSGS
               :  (EI, button_p /= "1111" and button_p'event, RE-
SET_AND_DECODE);
  }
}
state
{
  name {   INITIALIZE_SYSTEM   }
  code
  {
      num_msgs <= 0;
      --rewind tape to beginning
      tape_rew_p <= '1';
      if (tape_count_p /= 0) then
         wait  until tape_count_p = 0;
      end if;
      tape_rew_p <= '0';
      toggle_on_hangup <= '0';
  }
}
state
{
  name {   MACHINE_ON_TOGGLER   }
  code
  {
      machine_on <= '0';
      loop
         wait  until machine_on_pushed_p = '1' or machine_on_toggle = '1';
         --toggle
         if machine_on = '0' then
            machine_on <= '1';
         else
            machine_on <= '0';
         end if;
      end loop ;
  }
}
state
{
  name {   MAIN   }
  sequential substates
  {
      RESET_STATE : (EOC, system_on_p = '1', SYSTEM_ON),
                (EOC, system_on_p = '0', SYSTEM_OFF);
      SYSTEM_ON   : (EI, system_on_p = '0', SYSTEM_OFF);
      SYSTEM_OFF  : (EI, system_on_p = '1', SYSTEM_ON);
  }
}
state
{
  name {   MISC_CMDS   }
  declarations
  {
      signal first_entered : boolean :=true;
  }
  sequential substates
  {
      MISC_DECODE   : (EOC, button_p = "0010", stop),
                (EOC, button_p = "0011", ERASE_MSGS),
                (EOC, button_p = "0100", HEAR_ANN),
                (EOC, button_p = "0101", REC_ANN),
                (EOC, button_p = "0110", TOGGLE_SYSTEM);
      ERASE_MSGS    : (EOC, true, MISC_DECODE);
      HEAR_ANN      : (EOC, true, MISC_DECODE);
      REC_ANN       : (EOC, true, MISC_DECODE);
      TOGGLE_SYSTEM : (EOC, true, MISC_DECODE);
  }
}
state
{
  name {   MISC_DECODE   }
```

```
  code
  {
    if (first_entered) then
      produce_beep_p <= '1';
      wait for 1 fs;
      produce_beep_p <= '0';
      first_entered <= false;
    end if;
    wait until button_p /= "1111" and button_p'event;
  }
}
state
{
  name { MONITOR }
  declarations
  {
    variable num_rings: integer range 1 to 20;
    variable i: integer range 0 to 20;
    procedure determine_num_rings( num_msgs : in integer ;
                                   tollsaver : in bit ;
                                   machine_on : in bit ;
                                   variable num_rings : out integer ) is
    begin
     if ((num_msgs > 0) and (tollsaver = '1') and (machine_on = '1')) then
        num_rings := 2;
      elsif (machine_on = '1') then
        num_rings := 4;
      else
        --override in case forgot to turn machine on
        num_rings := 15;
      end if;
    end;
  }
  code
  {
    determine_num_rings(num_msgs,tollsaver_p,machine_on,num_rings);
    i := 0;
    while (i < num_rings) loop
      wait on ring_p,machine_on,tollsaver_p;
      if ring_p = '1' and ring_p'event then
        assert false report "MONITOR: Caught ring." severity NOTE;
        i := i + 1;
      else
       determine_num_rings(num_msgs,tollsaver_p,machine_on,num_rings);
      end if;
    end loop ;
    offhook_p <= '1';
  }
}
state
{
  name { PLAY }
  code
  {
    --play
    tape_play_p <= '1';
  }
}
state
{
  name { PLAY_ALL_MSGS }
  code
  {
    play_all_msgs_proc(terminal_tape_count,tape_count_p,tape_rew_p,
                       tape_play_p,produce_beep_p);
  }
}
state
{
  name { PLAY_ANNOUNCEMENT }
  code
  {
    ann_play_p <= '1';
    wait until ann_done_p = '1';
    ann_play_p <= '0';
  }
}
state
{
  name { RECORD_MSG }
  code
  {
    produce_beep_p <= '1';
    wait for 1 fs;
    produce_beep_p <= '0';
    tape_rec_p <= '1';
    if not (dialtone_p = '1') then
      --times out
      wait until dialtone_p = '1' for 1000 fs;
      --put a beep on the tape to indicate end of message
      produce_beep_p <= '1';
      wait for 1 fs;
      produce_beep_p <= '0';
      num_msgs <= num_msgs + 1;
    end if;
    tape_rec_p <= '0';
  }
}
state
{
  name { REC_ANN }
  code
  {
    --give the user some time to prepare
    wait for 50 fs;
    produce_beep_p <= '1';
    wait for 1 fs;
    produce_beep_p <= '0';
    wait for 0 fs;
    ann_rec_p <= '1';
    wait until ann_done_p = '1';
    ann_rec_p <= '0';
    produce_beep_p <= '1';
    wait for 1 fs;
    produce_beep_p <= '0';
  }
}
state
{
  name { REMOTE_OPERATION }
  declarations
  {
    signal code_ok : bit ;
  }
  sequential substates
  {
    STOP_PREV       : (EOC, true, CHECK_ID);
    CHECK_ID        : (EOC, code_ok = '1', RESPOND_TO_CMDS),
                      (EOC, code_ok = '0', stop),
                      (EI, dialtone_p = '1', stop);
    RESPOND_TO_CMDS : (EOC, true, stop);
  }
}
state
{
  name { RESET_AND_DECODE }
  code
  {
    tape_play_p <= '0';
    tape_fwd_p <= '0';
    tape_rew_p <= '0';
  }
}
state
{
```

```
  name {    RESET_STATE   }
  code
  {
    user_code[1] <= "0001";
    user_code[2] <= "0010";
    user_code[3] <= "0011";
    user_code[4] <= "0100";
    null;
  }
}
state
{
  name {    RESET_TAPE   }
  declarations
  {
    variable tape_count: integer;
  }
  code
  {
    if (tape_count_p > terminal_tape_count) then
      tape_rew_p <= '1';
      wait until not (tape_count_p > terminal_tape_count) ;
      tape_rew_p <= '0';
    elsif (tape_count_p < terminal_tape_count) then
      tape_fwd_p <= '1';
      wait until not (tape_count_p < terminal_tape_count) ;
      tape_fwd_p <= '0';
    end if;
  }
}
state
{
  name {    RESPOND_TO_CMDS   }
  sequential substates
  {
    HEAR_MSGS_CMDS: (EOC, true, MISC_CMDS),
                (EI, dialtone_p = '1', RESET_TAPE);
    MISC_CMDS     : (EOC, button_p = "0010", HEAR_MSGS_CMDS),
                (EOC, other, RESET_TAPE),
                (EI, dialtone_p = '1', RESET_TAPE);
    RESET_TAPE    : (EOC, true, stop);
  }
}
state
{
  name {    RESPOND_TO_EXT_LINE   }
  sequential substates
  {
    MONITOR : (EOC, true, ANSWER),
          (EI, dialtone_p = '1' and dialtone_p'event, MONITOR);
    ANSWER  : (EOC, true, MONITOR),
          (EI, machine_on = '0' and machine_on'event, MONITOR);
  }
}
state
{
  name {    RESPOND_TO_MACHINE_BUTTON   }
  sequential substates
  {
    DECODE_PUSHED_BUTTON
      : (EOC, play_pushed_p = '1', HANDLE_PLAY_PUSHED),
        (EOC, fwd_pushed_p = '1', HANDLE_FWD_PUSHED),
        (EOC, rew_pushed_p = '1', HANDLE_REW_PUSHED),
        (EOC, memo_pushed_p = '1', HANDLE_MEMO_PUSHED),
        (EOC, stop_pushed_p = '1', HANDLE_STOP_PUSHED),
       (EOC, hear_ann_pushed_p = '1', HANDLE_HEAR_ANN_PUSHED),
        (EOC, rec_ann_pushed_p = '1', HANDLE_REC_ANN_PUSHED),
      (EOC, play_msgs_pushed_p = '1', HANDLE_PLAY_MSGS_PUSHED);
    HANDLE_PLAY_PUSHED       : (EOC, true, stop);
    HANDLE_FWD_PUSHED        : (EOC, true, stop);
```

```
    HANDLE_REW_PUSHED        : (EOC, true, stop);
    HANDLE_MEMO_PUSHED       : (EOC, true, stop);
    HANDLE_STOP_PUSHED       : (EOC, true, stop);
    HANDLE_HEAR_ANN_PUSHED   : (EOC, true, stop);
    HANDLE_REC_ANN_PUSHED    : (EOC, true, stop);
    HANDLE_PLAY_MSGS_PUSHED : (EOC, true, stop);
  }
}
state
{
  name {    REW   }
  code
  {
    --note: user can terminate rewind at any time by pressing stop or
    --any other valid button
    tape_rew_p <= '1';
    if (tape_count_p /= 0) then
      wait until tape_count_p = 0;
    end if;
    tape_rew_p <= '0';
  }
}
state
{
  name {    STOP_PREV   }
  code
  {
   --shut off tape recording or announcement since could have interrupted
    tape_rec_p <= '0';
    ann_play_p <= '0';
    produce_beep_p <= '0';
  }
}
state
{
  name {    STOP_TAPE   }
  code
  {
    tape_play_p <= '0';
    tape_fwd_p <= '0';
    tape_rew_p <= '0';
  }
}
state
{
  name {    SYSTEM_OFF   }
  code
  {
    null;
  }
}
state
{
  name {    SYSTEM_ON   }
  declarations
  {
    signal terminal_tape_count : integer ;
    signal toggle_on_hangup : bit ;
   procedure play_all_msgs_proc(signal terminal_tape_count : inout integer
;
                        signal tape_count : in integer ;
                        signal tape_rew : out bit ;
                        signal tape_play : out bit ;
                        signal produce_beep : out bit ) is
    begin
      terminal_tape_count <= tape_count_p;
      --rewind to beginning of tape
      tape_rew <= '1';
        if not (tape_count = 0) then
          wait until tape_count = 0;
      end if;
      tape_rew <= '0';
```

```
      --play until end of last message
      tape_play <= '1';
      if (tape_count < terminal_tape_count) then
         wait until tape_count = terminal_tape_count;
      end if;
      tape_play <= '0';
      --produce a beep
      produce_beep <= '1';
      wait for 1 fs;
      produce_beep <= '0';
    end;
  }
  sequential substates
  {
    INITIALIZE_SYSTEM
      : (EOC, true, RESPOND_TO_EXT_LINE);
    RESPOND_TO_EXT_LINE
     : (EI, machine_button_pushed = '1' and machine_button_pushed'event,
        RESPOND_TO_MACHINE_BUTTON);
    RESPOND_TO_MACHINE_BUTTON
      : (EOC, true, RESPOND_TO_EXT_LINE),
      (EI, machine_button_pushed = '1' and machine_button_pushed'event,
         RESPOND_TO_MACHINE_BUTTON);
  }
}
state
{
  name {   TOGGLE_SYSTEM   }
  code
  {
    toggle_on_hangup <= '1';
  }
}
state
{
  name {   WAIT_FIRST_CMD   }
  code
  {
    wait until button_p /= "1111" and button_p'event;
  }
}
```

E Intel 8237 DMA Controller Textual SpecChart

Note: this description is known to contain minor errors. An error-free version will be provided in an upcoming revision of this report.

```
state
{
  name {   ACTIVE_INIT  }
  declarations
  {
    variable transfer_type: bit_vector (1 downto 0);
  }
  code
  {
    if (iDACK0) then
      chnum := 0;
    elsif (iDACK1) then
      chnum := 1;
    elsif (iDACK2) then
      chnum := 2;
    elsif (iDACK3) then
      chnum := 3;
    end if;
    MTM := (cmd_reg[0] = '1') and iDACK0;
    DTM := false;
    STM := false;
    BTM := false;
    CTM := false;
    if (not MTM ) then
      transfer_type := mode_regfile[chnum][5 downto 4];
      case (transfer_type) is
        when "00" =>
          DTM := true;
        when "01" =>
          STM := true;
        when "10" =>
          BTM := true;
        when "11" =>
          CTM := true;
      end case;
    end if;
  }
}
state
{
  name {   ACTIVE_LAST  }
  code
  {
    TC := false;
    EOP_out <=  '1';
    --make DACK inactive (could be either active high or low)
    DACK[chnum] <=  not (cmd_reg[6]) ;
    AEN <=  '0';
  }
}
state
{
  name {   ACTIVE_STATE  }
  declarations
  {
    procedure put_address_on_bus(signal AL_out : out nibble ;
  signal AH : out nibble ;
  constant curr_addr_regfile : in regfile4x16 ;
       constant chnum : in integer ;
  signal AEN : out bit ;
  constant DB_in : in byte ;
  signal DB_out : out byte ;
  signal ADSTB : out bit ) is
    begin
      AL_out <=  curr_addr_regfile[chnum][3 downto 0];
      AH <=  curr_addr_regfile[chnum][7 downto 4];
      AEN <=  '1';
      if not (DB_in = curr_addr_regfile[chnum][15 downto 8])  then
        ADSTB <=  '1';
        DB_out <=  curr_addr_regfile[chnum][15 downto 8];
        ADSTB <=  '0' after time_stb_high;
      end if;
    end;
    procedure inc_dec_address(constant MTM : in boolean ;
constant mode_regfile : in regfile4x6 ;
variable chnum : in integer ;
signal curr_addr_regfile : inout regfile4x16) is
    begin
      if not (MTM and (mode_regfile[0][1] = '1'))  then
        if (mode_regfile[chnum][3] = '1') then
          curr_addr_regfile[chnum] <=  DEC(curr_addr_regfile[chnum]);
        else
          curr_addr_regfile[chnum] <=  INC(curr_addr_regfile[chnum]);
        end if;
      end if;
    end;
    variable MTM: boolean;
    variable DTM: boolean;
    variable STM: boolean;
    variable CTM: boolean;
    variable BTM: boolean;
    variable TC: boolean;
  }
  sequential substates
  {
    ACTIVE_INIT :  (EOC, MTM, MEM_TO_MEM),
   (EOC, CTM, ACTIVE_LAST),
   (EOC, STM or DTM or BTM, REGULAR_TRANSFER);
    MEM_TO_MEM :  (EOC, true, ACTIVE_LAST);
    REGULAR_TRANSFER: (EOC, true, ACTIVE_LAST);
    ACTIVE_LAST :  (EOC, STM and not (TC) , stop),
(EOC, other, AUTOINIT);
    AUTOINIT :  (EOC, true, stop);
  }
}
state
{
  name {   AUTOINIT  }
  declarations
  {
    variable temp: reg16;
  }
  code
  {
    if (mode_regfile[chnum][2] = '1') then
      temp := curr_addr_regfile[chnum];
      curr_addr_regfile[chnum] <=  base_addr_regfile[chnum];
      base_addr_regfile[chnum] <=  temp;
      temp := curr_wc_regfile[chnum] <= base_wc_regfile[chnum];
      curr_wc_regfile[chnum] <=  base_wc_regfile[chnum];
      base_wc_regfile[chnum] <=  temp;
    end if;
  }
}
state
{
  name {   FIXED_PRIORITY  }
  sequential substates
  {
    fp0 : (EOC, EOP_in = '1', fp1), (EOC, EOP_in = '0', fp0);
    fp1 : ;
  }
}
state
{
  name {   IDLE_STATE  }
  declarations
```

```
  {
    variable cmd: bit_vector (6 downto 0);
    variable dbcmd: bit_vector (1 downto 0);
  }
  code
  {
      cmd := CS &IOR_in &IOW_in &AL_in[3] &AL_in[2] &AL_in[1]
&AL_in[0];
    case (cmd) is
      when "0101000" =>
        cmd_reg <= DB_in;
      when "0101011" =>
        dbcmd := DB_in[1] &DB_in[0];
        case dbcmd is
          when "00" =>
            mode_regfile[0] <= DB_in[7 downto 2];
          when "01" =>
            mode_regfile[1] <= DB_in[7 downto 2];
          when "10" =>
            mode_regfile[2] <= DB_in[7 downto 2];
          when "11" =>
            mode_regfile[3] <= DB_in[7 downto 2];
        end case;
      when "0101001" =>
        dbcmd := DB_in[1] &DB_in[0];
        case dbcmd is
          when "00" =>
            req_reg[0] <= DB_in[2];
          when "01" =>
            req_reg[1] <= DB_in[2];
          when "10" =>
            req_reg[2] <= DB_in[2];
          when "11" =>
            req_reg[3] <= DB_in[2];
        end case;
      when "0101010" =>
        dbcmd := DB_in[1] &DB_in[0];
        case dbcmd is
          when "00" =>
            mask_reg[0] <= DB_in[2];
          when "01" =>
            mask_reg[1] <= DB_in[2];
          when "10" =>
            mask_reg[2] <= DB_in[2];
          when "11" =>
            mask_reg[3] <= DB_in[2];
        end case;
      when "0101111" =>
        mask_reg[3 downto 0] <= DB_in[3 downto 0];
      when "0011101" =>
        DB_out <= tmp_reg;
      when "0011000" =>
        DB_out <= status_reg;
      when "0101100" =>
        int_FF <= '0';
      when "0101101" =>
        cmd_reg <= "00000000";
        status_reg <= "00000000";
        req_reg <= "0000";
        tmp_reg <= "00000000";
        mask_reg <= "1111";
      when "0100000" =>
        case (int_FF) is
          when '0' =>
            base_wc_regfile[0][7 downto 0] <= DB_in;
            curr_wc_regfile[0][7 downto 0] <= DB_in;
            int_FF <= '1';
          when '1' =>
            base_wc_regfile[0][15 downto 8] <= DB_in;
            curr_wc_regfile[0][15 downto 8] <= DB_in;
        end case;
      when "0010000" =>
        case (int_FF) is
          when '0' =>
            curr_addr_regfile[0][7 downto 0] <= DB_in;
            int_FF <= '1';
          when '1' =>
            curr_addr_regfile[0][15 downto 8] <= DB_in;
        end case;
      when "0100001" =>
        case (int_FF) is
          when '0' =>
            base_wc_regfile[0][7 downto 0] <= DB_in;
            curr_wc_regfile[0][7 downto 0] <= DB_in;
            int_FF <= '1';
          when '1' =>
            base_wc_regfile[0][15 downto 8] <= DB_in;
            curr_wc_regfile[0][15 downto 8] <= DB_in;
        end case;
      when "0010001" =>
        case (int_FF) is
          when '0' =>
            curr_wc_regfile[0][7 downto 0] <= DB_in;
            int_FF <= '1';
          when '1' =>
            curr_wc_regfile[0][15 downto 8] <= DB_in;
        end case;
      when "0100010" =>
        case (int_FF) is
          when '0' =>
            base_addr_regfile[1][7 downto 0] <= DB_in;
            curr_addr_regfile[1][7 downto 0] <= DB_in;
            int_FF <= '1';
          when '1' =>
            base_addr_regfile[1][15 downto 8] <= DB_in;
            curr_addr_regfile[1][15 downto 8] <= DB_in;
        end case;
      when "0010010" =>
        case (int_FF) is
          when '0' =>
            curr_addr_regfile[1][7 downto 0] <= DB_in;
            int_FF <= '1';
          when '1' =>
            curr_addr_regfile[1][15 downto 8] <= DB_in;
        end case;
      when "0100011" =>
        case (int_FF) is
          when '0' =>
            base_wc_regfile[1][7 downto 0] <= DB_in;
            curr_wc_regfile[1][7 downto 0] <= DB_in;
            int_FF <= '1';
          when '1' =>
            base_wc_regfile[1][15 downto 8] <= DB_in;
            curr_wc_regfile[1][15 downto 8] <= DB_in;
        end case;
      when "0010011" =>
        case (int_FF) is
          when '0' =>
            curr_wc_regfile[1][7 downto 0] <= DB_in;
            int_FF <= '1';
          when '1' =>
            curr_wc_regfile[1][15 downto 8] <= DB_in;
        end case;
      when "0100100" =>
        case (int_FF) is
          when '0' =>
            base_addr_regfile[2][7 downto 0] <= DB_in;
            curr_addr_regfile[2][7 downto 0] <= DB_in;
            int_FF <= '1';
          when '1' =>
            base_addr_regfile[2][15 downto 8] <= DB_in;
            curr_addr_regfile[2][15 downto 8] <= DB_in;
        end case;
      when "0010100" =>
        case (int_FF) is
          when '0' =>
            curr_addr_regfile[2][7 downto 0] <= DB_in;
            int_FF <= '1';
          when '1' =>
```

```
            curr_addr_regfile[2][15 downto 8] <= DB_in;
        end case;
      when "0100101" =>
        case (int_FF) is
          when '0' =>
            base_wc_regfile[2][7 downto 0] <= DB_in;
            curr_wc_regfile[2][7 downto 0] <= DB_in;
            int_FF <= '1';
          when '1' =>
            base_wc_regfile[2][15 downto 8] <= DB_in;
            curr_wc_regfile[2][15 downto 8] <= DB_in;
        end case;
      when "0010101" =>
        case (int_FF) is
          when '0' =>
            curr_wc_regfile[2][7 downto 0] <= DB_in;
            int_FF <= '1';
          when '1' =>
            curr_wc_regfile[2][15 downto 8] <= DB_in;
        end case;
      when "0100110" =>
        case (int_FF) is
          when '0' =>
            base_addr_regfile[3][7 downto 0] <= DB_in;
            curr_addr_regfile[3][7 downto 0] <= DB_in;
            int_FF <= '1';
          when '1' =>
            base_addr_regfile[3][15 downto 8] <= DB_in;
            curr_addr_regfile[3][15 downto 8] <= DB_in;
        end case;
      when "0010110" =>
        case (int_FF) is
          when '0' =>
            curr_addr_regfile[3][7 downto 0] <= DB_in;
            int_FF <= '1';
          when '1' =>
            curr_addr_regfile[3][15 downto 8] <= DB_in;
        end case;
      when "0100111" =>
        case (int_FF) is
          when '0' =>
            base_wc_regfile[3][7 downto 0] <= DB_in;
            curr_wc_regfile[3][7 downto 0] <= DB_in;
            int_FF <= '1';
          when '1' =>
            base_wc_regfile[3][15 downto 8] <= DB_in;
            curr_wc_regfile[3][15 downto 8] <= DB_in;
        end case;
      when "0010111" =>
        case (int_FF) is
          when '0' =>
            curr_wc_regfile[3][7 downto 0] <= DB_in;
            int_FF <= '1';
          when '1' =>
            curr_wc_regfile[3][15 downto 8] <= DB_in;
        end case;
      when others =>
        assert false report "Invalid programming mode" severity note;
    end case;
  }
}
state
{
  name { MAIN }
  declarations
  {
    signal base_addr_regfile : regfile4x16 ;
    signal curr_addr_regfile : regfile4x16 ;
    signal base_wc_regfile : regfile4x16 ;
    signal curr_wc_regfile : regfile4x16 ;
    signal mode_regfile : regfile4x6 ;
    signal tmp_reg : reg8 ;
    signal req_reg : reg4 ;
    signal int_FF : bit ;
    variable chnum: integer;
    constant time_db_stable: time:=200 ns;
    constant time_stb_high: time:=100 ns;
    constant time_clock_cycle: time:=320 ns;
  }
  sequential substates
  {
    IDLE_STATE : (EOC, HLDA = '1', ACTIVE_STATE),
   (EI, RESET = '1', RESET_STATE);
    ACTIVE_STATE : (EOC, true, IDLE_STATE),
   (EI, RESET = '1', RESET_STATE);
    RESET_STATE : (EOC, true, IDLE_STATE);
  }
}
state
{
  name { MEM_TO_MEM }
  sequential substates
  {
    mtm0 : (EOC, true, mtm1);
    mtm1 : (EOC, true, mtm2);
    mtm2 : (EOC, true, mtm3);
    mtm3 : (EOC, not TC , mtm0), (EOC, TC, stop);
  }
}
state
{
  name { PRIORITY }
  sequential substates
  {
    FIXED_PRIORITY : (EI, cmd_reg[4] = '1', ROTATING_PRIORITY);
    ROTATING_PRIORITY : (EI, cmd_reg[4] = '0', FIXED_PRIORITY);
  }
}
state
{
  name { REGULAR_TRANSFER }
  declarations
  {
    procedure get_rw( mode_regfile : regfile4x6 ;
chnum : integer ;
rd : out boolean ;
wr : out boolean ) is
    begin
     rd := (mode_regfile[chnum][1] = '1') and (mode_regfile[chnum][0] ='0');
        wr := (mode_regfile[chnum][1] = '0') and (mode_regfile[chnum][0]
='1');
    end;
    variable reading: boolean;
    variable writing: boolean;
  }
  sequential substates
  {
    rt0 : (EOC, true, rt1);
    rt1 : (EOC, STM or (BTM and (TC or EOP_in = '1')) or
 (DTM and (TC or EOP_in='1' or DREQ(chnum)=cmd_reg[6])),stop),
        (EOC, other, rt0);
  }
}
state
{
  name { RESET_STATE }
  code
  {
    cmd_reg <= "00000000";
    status_reg <= "00000000";
    req_reg <= "0000";
    tmp_reg <= "00000000";
    mask_reg <= "1111";
  }
```

```
}
state
{
  name {    ROTATING_PRIORITY  }
  declarations
  {
    type array_4 is array (0 to 3) of integer;
    variable plist: array_4;
    variable making_req: boolean;
    function calc_iDACK(plist : array_4 ; chnum : integer) return boolean
is
    begin
      if ((REQ(chnum) and (plist[0] = chnum)) or
        ((plist[1] = chnum) and (not (REQ[0]) )) or
        ((plist[2] = chnum) and (not (REQ[1]) ) and (not (REQ[2]) )) or
  ((plist[3] = chnum) and (not (REQ[2]) ) and (not (REQ[1]) )
          and (not (REQ[0]) ))) then
        return true;
      else
        return false;
      end if;
    end;
    function get_index_in_plist( plist :   array_4 ;
  chnum :   integer ) return integer is
      variable index: integer;
    begin
      index := 0;
      while not (plist[index] = chnum)  loop
        index := index + 1;
      end loop ;
      return index;
    end;
    procedure rotate_plist(variable plist : inout array_4 ;
   chnum :   integer ) is
      variable index: integer;
      variable tmp: integer;
    begin
      index := get_index_in_plist(plist,chnum);
      for i in 1 to index + 1 loop
        tmp := plist[3];
        plist[3] := plist[2];
        plist[2] := plist[1];
        plist[1] := plist[0];
        plist[0] := tmp;
      end loop ;
    end;
  }
  sequential substates
  {
    rp0 : (EOC, HLDA = '1', rp1);
    rp1 : (EOC, making_req, rp2), (EOC, not making_req , rp1);
    rp2 : (EOC, EOP_in = '1', rp3);
    rp3 : (EOC, true, rp1);
  }
}
state
{
  name {    dma  }
  declarations
  {
    use bit_functions;
    use i8237_types;
    port CLK: in bit ;
    port CS: in bit ;
    port RESET: in bit ;
    port READY: in bit ;
    port HLDA: in bit ;
    port DREQ: in nibble ;
    port DB_in: in byte ;
    port DB_out: out byte ;
    port IOR_in: in bit ;
    port IOR_out: out bit ;
    port IOW_in: in bit ;
    port IOW_out: out bit ;
    port EOP_in: in bit ;
    port EOP_out: out bit ;
    port AL_in: in nibble ;
    port AL_out: out nibble ;
    port AH: out nibble ;
    port HRQ: out bit ;
    port DACK: out nibble ;
    port AEN: out bit ;
    port ADSTB: out bit ;
    port MEMR: out bit ;
    port MEMW: out bit ;
    signal cmd_reg : byte ;
    signal status_reg : byte ;
    signal mask_reg : nibble ;
    signal iDACK0 : boolean ;
    signal iDACK1 : boolean ;
    signal iDACK2 : boolean ;
    signal iDACK3 : boolean ;
    subtype reg16 is bit_vector (15 downto 0);
    subtype reg8 is bit_vector (7 downto 0);
    subtype reg6 is bit_vector (5 downto 0);
    subtype reg4 is bit_vector (3 downto 0);
    type regfile4x16 is array (0 to 3) of reg16;
    type regfile4x6 is array (0 to 3) of reg6;
    function REQ( x :   integer ) return boolean is
    begin
      return ((DREQ[x] = cmd_reg[7]) and (mask_reg[x] = '0'));
    end;
  }
  constraints
  {
    area_per_chip <= 15000 sqmm (3);
    num_chips <= 4 chips (4);
    pins_per_chip <= 50 pins (3);
  }
  concurrent substates
  {
    MAIN : ;
    PRIORITY : ;
  }
}
state
{
  name {    fp0  }
  code
  {
    --internal DACK signals for our own use;
    --external DACK signals come later.
    iDACK0 <=  (HLDA = '1') and REQ(0);
    iDACK1 <=  (HLDA = '1') and not (REQ(0))  and REQ(1);
   iDACK2 <=  (HLDA = '1') and not (REQ(0))  and not (REQ(1))  and
REQ(2);
    iDACK3 <=  (HLDA = '1') and not (REQ(0))  and not (REQ(1))
          and not (REQ(2))  and REQ(3);
    if (REQ(0) or REQ(1) or REQ(2) or REQ(3)) then
      HRQ <=  '1';
    end if;
  }
}
state
{
  name {    fp1  }
  code
  {
    HRQ <=  '0';
  }
}
state
{
  name {    mtm0  }
```

```
  code
  {
    --we know channel is 0
    put_address_on_bus(AL_out,AH,curr_addr_regfile,chnum,AEN,
 DB_in,DB_out,ADSTB);
    MEMR <= '1';
    MEMW <= '1';
    wait for time_db_stable;
    tmp_reg <= DB_in;
    inc_dec_address(MTM,mode_regfile,chnum,curr_addr_regfile);
  }
}
state
{
  name { mtm1 }
  code
  {
    MEMR <= '0';
    chnum := 1;
    put_address_on_bus(AL_out,AH,curr_addr_regfile,chnum,
 AEN,DB_in,DB_out,ADSTB);
  }
}
state
{
  name { mtm2 }
  code
  {
    DB_out <= tmp_reg;
    if (curr_wc_regfile[1] = "0000000000000000") then
      TC := true;
    end if;
    curr_wc_regfile[1] <= DEC(curr_wc_regfile[1]);
    inc_dec_address(MTM,mode_regfile,chnum,curr_addr_regfile);
  }
}
state
{
  name { mtm3 }
  code
  {
    MEMW <= '0';
  }
}
state
{
  name { rp0 }
  code
  {
    plist[0] := 0;
    plist[1] := 1;
    plist[2] := 2;
    plist[3] := 3;
    making_req := false;
  }
}
state
{
  name { rp1 }
  code
  {
    iDACK0 <= calc_iDACK(plist,0);
    iDACK1 <= calc_iDACK(plist,1);
    iDACK2 <= calc_iDACK(plist,2);
    iDACK3 <= calc_iDACK(plist,3);
    if (REQ[0] or REQ[1] or REQ[2] or REQ[3]) then
      HRQ <= '1';
      making_req := true;
    end if;
  }
}
state
{
  name { rp2 }
  code
  {
    if iDACK0 then
      rotate_plist(plist,0);
    elsif iDACK1 then
      rotate_plist(plist,1);
    elsif iDACK2 then
      rotate_plist(plist,2);
    elsif iDACK3 then
      rotate_plist(plist,3);
    end if;
  }
}
state
{
  name { rp3 }
  code
  {
    HRQ <= '0';
    making_req := false;
  }
}
state
{
  name { rt0 }
  code
  {
    put_address_on_bus(AL_out,AH,curr_addr_regfile,chnum,
 AEN,DB_in,DB_out,ADSTB);
    get_rw(mode_regfile,chnum,reading,writing);
    if (writing) then
      IOR_out <= '1';
      MEMW <= '1';
    elsif (reading) then
      MEMR <= '1';
      IOW_out <= '1';
    end if;
    if READY = '1' then
      wait for 2 * time_clock_cycle -100 ns;
    else
      wait for time_clock_cycle -100 ns;
    end if;
  }
}
state
{
  name { rt1 }
  code
  {
    get_rw(mode_regfile,chnum,reading,writing);
    if (writing) then
      IOR_out <= '0';
      MEMW <= '0';
    elsif (reading) then
      MEMR <= '0';
      IOW_out <= '0';
    end if;
    DACK[chnum] <= cmd_reg[6];
    if (curr_wc_regfile[chnum] = "0000000000000000") then
      TC := true;
    end if;
    curr_wc_regfile[chnum] <= DEC(curr_wc_regfile[chnum]);
    inc_dec_address(MTM,mode_regfile,chnum,curr_addr_regfile);
  }
}
```