# EXTEND-L

# AN INPUT LANGUAGE FOR EXTENSIBLE REGISTER TRANSFER COMPILATION

BY

NIKIL D. DUTT
DANIEL D. GAJSKI

Technical Report 88-11

Information and Computer Science
University of California at Irvine
Irvine, CA 92717
(714) 856 7063

Abstract: This report discusses the model and input language for **EXTEND**, a synthesis system that permits extensible register transfer synthesis. **EXTEND-L** fills the need for a language that bridges the gap between existing behavioral input descriptions, which are too abstract, and structural schematics, which cannot capture the high-level behavior. The report first discusses previous work in behavioral synthesis and summarizes the deficiencies of these behavioral specifications. The report then describes the proposed language in detail, and concludes with a few examples that show its utility.

# TABLE OF CONTENTS

## LIST OF FIGURES

# CHAPTER 1.

# INTRODUCTION

## 1.1. Problem Description

The task of high level synthesis spans the continuum from fully automatic design, which starts from a purely behavioral description, down to compiling a fully specified structural design. Although automatic high level synthesis is the ultimate goal, several parts of the synthesis process are not completely understood. The design process spans many levels, and several inter-related synthesis tasks need to be performed at each level. These synthesis tasks are actively being researched at various universities and industries. Due to the complexity of this process, only a few tasks are examined at a time. Until we understand the synthesis task in its entirety, the user will play an important part in the design process. There is a need for a tool which allows the designer to give input to the iterative decision making loop. **EXTEND** is a new tool that attempts to meet these needs. This new tool

(1) *is powerful*: the user can specify any level of binding, from fully bound structural designs to purely behavioral specifications;

(2) *is interactive*: permits user interaction of compiled design; employs a mixed graphic/textual interface for ease of use;

(3) *is general*: allows combined specification of synchronous and asynchronous behavior;

(4) *simplifies compilation*: uses a small set of constructs for specifying timing and asynchronous behavior.

(5) *is extensible*: permits gradual incorporation of synthesis tools as they become well understood.

The user interacts with EXTEND through its input language **EXTEND-L** and the graphic user interface. This paper summarizes the model and features of EXTEND-L.

## 1.2. Existing Tools

### 1.2.1. Description level

Most of the existing synthesis tools are either at too high a level of description, or too low a level of description. Most research synthesis systems start with an algorithmic or instruction-set-processor-like description which is very abstract. With this very high level of description, no structural information is specified; the behavior is described as an algorithm which operates on abstract data carriers such as variables. On the other end of the spectrum, commercial schematic entry systems require the designer to enter a net-list of structures that are already designed; almost all of the high level synthesis is performed by the user. These low-level systems simply serve as a convenient entry point for simulation and layout.

### 1.2.2. Design Model

Existing high-level synthesis research has primarily concentrated on the synthesis of synchronous processes, where each process may be viewed as an FSM. Most high-level synthesis systems currently permit the synthesis of a single process at a time, with the designer manually specifying the inter-process communication and protocols. They also have no means of expressing simple asynchronous behavior, such as setting or clearing of registers within synchronous processes. Asynchronous designs are generally not considered.

These restrictions in the design model often arise from the fact that the existing high-level synthesis systems are targeted towards synthesizing instruction-set processors or microprocessor-like architectures. As a result, these synthesis systems cannot effectively cope with ASIC designs which have differing architectures and which often exhibit both synchronous and asynchronous behavior.

Another limitation of many existing systems is that most of them synthesize the data path alone, and generate a (symbolic) description of the control for the data path. This controller description is then sent to a control synthesizer as a post-synthesis task. Decoupling of the two synthesis tasks makes it hard to perform tradeoffs between the control and data parts of the design.

### 1.2.3. Interface and Timing Issues

Interface and timing issues are often not considered, although a few high level synthesis systems are starting to look into these issues. Several issues are involved in this regard; a few of the important ones are mentioned here.

*Communication between processes*: many systems cannot handle multiple processes that communicate with each other. The communication may involve a specified sequence of activities occurring at the process inputs and outputs, possibly with associated timing constraints. A simple example of this communication is the protocol for reading or writing of a memory that is external to a processor.

*Timing constraints between events*: within a process, operations may have to be performed in a particular order with respect to events occurring on the input ports of the process. For instance, a register may have to be loaded after a signal on the input port rises; minimum and maximum times for the loading of the register may need to be specified.

*Timing constraints on combinatorial paths*: if a designer knows that a certain section of the design is going to be critical, minimum and maximum delays may need to be specified on the critical path.

### 1.2.4. Interactivity

The process of automation is fraught with inertia and distrust, as history has shown. Designers who are used to the "hands-on" approach of existing CAD tools will not trust tools that suddenly elevate the abstraction level of the design and which perform automatic synthesis. To gain user acceptance, a smooth transition from existing commercial CAD tools to fully automatic synthesis systems is required. Initially, the user must be able to provide hints (or "pragmas") to the synthesis system and guide the design using the user's design knowledge. As the problems in synthesis become well understood and synthesis tools mature, they may be incorporated into the synthesis environment. These considerations underscore the importance of allowing user interaction in the synthesis tasks. Existing high level synthesis systems have not addressed this very important issue, and often permit little or no user interaction.

# CHAPTER 2.

# DESIGN PARADIGM AND MODEL

## 2.1. Design Paradigm

The task of design spans many levels and involves several inter-related steps, each of which are complex and often time-consuming for a human designer to perform. In order to control the complexity of the design process, the task of design has traditionally been split up into several hierarchical levels. At each level of the design process, the task of synthesis takes a specification which is an "abstract" description of the design, and produces a "completed" design, which is then passed on to the next level as a specification. The "abstract" design is often considered to be some kind of behavior at that level, while the "completed" design is a structural description for that level. For instance, high level synthesis takes an algorithmic (abstract) description of a design, and produces a register-transfer (structural) implementation of the description.

Existing tools operate at the ends of the design continuum: they either attempt to perform completely automatic design, or they require the user to perform the design manually. What is missing is a set of tools that permits a gradual move towards completely automatic synthesis. This will allow a smooth transition towards automatic synthesis by interfacing to the existing lower-level infrastructure.

The other important issue is the role of the user in the design paradigm. While design synthesis tasks are being researched and understood, the user has to play an active part in the design process. Automation of the synthesis tasks involves capture of the designer's knowledge into the synthesis tool in some form (rules/algorithms/meta-rules, etc.). When a generated design does not meet the design constraints, several options are available to the user and the synthesis system. Existing research tools often require the user to re-write the input in the hope that constraints are met in subsequent synthesis cycles. In this situation, the user has to have a good understanding of the design tool to be able to predict the output of the synthesis tool. This approach (although necessary sometimes), does not make efficient use of the compiled design and suffers from the inaccuracy of the user's predictions or the inability of the user to understand the assumptions behind and implementation of a particular design tool.

The other approach is to permit user modification of compiled design. This provides better prediction capability and allows a smoother transition towards automation.

EXTEND is a system that attempts to meet these needs. Its language, EXTEND-L, allows the user to provide structural and design "hints" in the input specification. These "hints" or "bindings" may be removed at a later time when a synthesis tool that performs the task is mature enough to be incorporated into the synthesis environment.

By permitting a gradual incorporation of synthesis tools, the design paradigm also alleviates the problem of *tool obsolescence.* As better synthesis tools become available, it is easier to incorporate them into EXTEND.

At a higher level of abstraction, the constraints may be managed by a simple "knobs-and-gauges" approach by the user [BrGa86]. This approach can also take advantage of the compiled design by modifying those parts that do not meet the constraints, or by performing tradeoffs between different parts of the design. The "knobs-and-gauges" paradigm also permits larger exploration of the design space.

## 2.2. Design Model

The design model assumes a partitioning of the intended design into a set of communicating processes, each of which is a finite state machine (FSM). Each process is described separately by specifying its behavior with respect to signals on its input and/or output ports. A process may exhibit synchronous, asynchronous, or combined behavior and hence the communication protocol between processes may also be synchronous, semi-synchronous, or asynchronous.

### 2.2.1. Synchronous Processes

For synchronous processes, the behavior is expressed is composed of two parts:

(1) State-by-state description of the synchronous behavior using a control graph (which captures sequencing information) and associated text (which describes the data operations performed in that state).

This description partitions the synchronous process into control and data parts, which are synthesized separately.

(2) Behavior which describes specific asynchronous activities with respect to structural components within the synchronous process (eg. asynchronous "clear" for a register).

### 2.2.2. Asynchronous Processes

The behavior of an asynchronous process is derived from a timing chart which describes the outputs of the process with respect to the inputs along the time axis. This timing chart is not part of the input, but merely a starting point in the user's mind for entering the behavior of the design into the system. An "event", in this context, is a change in the value of an input signal which causes some asynchronous activity within the process.

The behavior is then captured through an asynchronous "event-state chart", where each event-state has associated with it some text that describes the actions to be performed on that event occurring. Any conditional activity or looping is described textually within the event-state. This restricts the event-state chart to fixed sequence of events in time. This is similar to the concept of "event-graphs" in the WAVES system for transducer synthesis [BoKa87].

Although this may seem like a limitation in the model, it is quite appropriate for describing asynchronous protocols at the RT level. The variety of examples in this document support our belief that this model suffices for a large range of ASIC designs.

**2.2.3. Combined Synchronous and Asynchronous Behavior** When describing a design that exhibits a combination of asynchronous and synchronous behavior, the behavior naturally lends itself to a particular style of description. We have identified three ways of describing this mixed behavior; each is appropriate in certain cases. In general, a combination of the three specifications may be required.

#### 2.2.3.1. Synchronous Behavior within Asynchronous Specification

The asynchronous behavior is expressed as a sequence of *event-states*. Each event-state is triggered by a specified event (which is often a change in the value of an external signal). Within an event-state, the behavior is captured with a set of of operations on variables. If the behavior in a particular event-state requires a sequence of synchronous operations, this may be described by a "call" to a synchronous state chart. Semantically, this implies that the clock is the default "event" for the synchronous sub-chart; there is no overhead involved in "entering" or "leaving" the synchronous description.

As an example, consider the description of a count-down timer which is activated asynchronously, counts down from a specified value synchronously, and asserts the "DONE" line to signal completion. This type of description is similar to the synchronous subcontrol described in [Clar73] and [Holl81].

#### 2.2.3.2. Asynchronous Behavior Within Synchronous Behavior

In a particular state of a synchronous state chart, an asynchronous assignment may be made to a variable (which is bound to a register). This assignment implies that the storage element corresponding to the variable is enabled by the synchronous clock in the specified state, but is actually clocked in by the asynchronous event specified. If the value assigned to the variable is zero, this implies "clearing" of the storage element in that synchronous state. This is a feature which is missing in most of the existing behavioral synthesis systems.

#### 2.2.3.3. Separate Asynchronous and Synchronous Specification

When there is a fair amount of both kinds of behavior, and both behaviors are somewhat orthogonal, it is convenient to describe the asynchronous behavior separately from the synchronous behavior for the same process. Both descriptions may refer to the same set of variables and/or defined structures. Semantically, the asynchronous specification for a variable or structure overrides the synchronous behavior. For instance, if the reset line for a counter is enabled, the counter is cleared (no matter which synchronous state the counter was in). When the reset line is disabled, the counter continues its synchronous behavior. This style of description is similar to the "applicative" and "imperative" descriptions in DSL [Camp85].

### 2.2.4. Constraints

EXTEND performs design at the register transfer level. The outputs of the system consist of a net-list of register-transfer components such as registers, shifters, counters, RAM's, etc. These component types are stored in a generic component library. Abstraction of the constraints is achieved through the use of these generic components. Constraints such as time, area and power can be estimated by functions that return values based on the instantiations of generic components in the library. The generic component library itself is characterized by loading in a technology file at run time. Appendix B gives a description of the types and attributes of generic components in the library.

### 2.2.5. Uniform Representation

The design model treats both synchronous and asynchronous processes in a similar fashion. The default "event" which takes a synchronous process into a new state is the clock. Conversely, each "event" in an asynchronous description may be viewed as a "clock" which activates a new state and triggers data transfers and operations.

The use of a control/data representation permits the synthesis task to focus on data path synthesis first, generating a symbolic control table of activities in each state. For synchronous systems, this table may be implemented in a variety of different styles: PLA +

state register, sequencer + control output generator, ROM-based control, random logic, etc. The asynchronous description suits itself to a "one-hot" implementation, a counter-based controller, encoded control or even combinatorial logic. Several optimizations may be performed on the one-hot control. For instance, the flip-flop for a state may be replaced by combinatorial logic if that state is completely input dependent. The internal representation used in EXTEND is the subject of a forthcoming report.

# CHAPTER 3.

# INPUT LANGUAGE

## 3.1. Essential Issues

Several issues are involved in designing a behavioral representation for synthesis. In this section, we discuss these issues to provide a framework for comparing and evaluating different input languages.

### 3.1.1. Model

A behavioral description is written with a target architecture in mind, which we call the architectural *model*. In the most general case, the model consists of a set of communicating processes, where each process is a generalized finite state machine implemented as a control unit (which generates control outputs and the state of the machine), and a data path (which performs computations on data values on each state of the machine). For further details, the reader is directed to the tutorial described in [GaDP86]. The underlying architectural model for the representation determines the power of its expression.

### 3.1.2. Underlying Representation

The representation scheme describes the data structures used in capturing the behavior. This is most commonly a variant of the control/data flow graph. Other constraints such as timing and protocols are handled differently by each system. This report discusses the input language and model; representational issues will be discussed in a subsequent report.

### 3.1.3. Hierarchy

There are two types of hierarchy that the behavioral representation may allow. *Structural hierarchy* specifies the interconnection of the communicating processes via global signals and ports on the processes. *Procedural hierarchy* permits decomposition of the behavior in a structured fashion, thereby allowing a concise representation. In some representations, a process (or a sub-process) may be encapsulated into a structure, and may be used later in the structural hierarchy.

### 3.1.4. Data Types

Data typing in a system is characterized by the *formats* (eg. number of bits), *types* (eg. boolean, integer, floating point) and the *representation* (eg. signed/unsigned, 2's complement) of all data carriers in the behavioral description. This includes the data types of variables, components, ports, etc. *Strict data typing* permits consistency checks during the synthesis process, but also burdens the compiler with more tests. A minimal data type would consist of variables characterized by their bit widths, with the system assuming a default data representation.

### 3.1.5. Sequentiality and Parallelism

This issue focuses on how the language permits the user to specify sequentiality of operations and parallelism of operations. Most schemes enforce sequentiality through data dependence of operations, as well as through specific control sequencing constructs. Parallelism in the representation is most commonly implied between operations that have no dependencies (control or data) between them. Some languages have special constructs to indicate parallelism of activities within a block.

### 3.1.6. Delay Specification and Synchronization

This aspect deals with how delays may be specified in the behavior, and how synchronization of communication between different processes is achieved. Delays may be specified relative to a process clock, or relative to a specific event within a process. Synchronization between different processes is often achieved by means of global signals that indicate the status of the communicating processes.

### 3.1.7. Asynchrony

Another issue, closely related to timing specification, is how a system represents asynchronous events. Recall that the architectural model often consists of a set of communicating processes, each of which is synchronous with respect to a local clock. How can asynchronous events like RESETS and INTERRUPTS be handled? Most often, the behavioral specification indicates that a signal is asynchronous, in which case its value is asynchronously set in a register and is detected at the next clock cycle.

Many systems that have a synchronous model of processes have no language constructs to describe this asynchronous behavior. As a result, these systems are unable to deal with very simple designs which exhibit some amount of asynchrony.

### 3.1.8. Bindings

Behavioral synthesis is composed of four basic tasks: *state binding*, *register binding*, *function unit binding* and *connection binding.* State binding involves assigning each operation in the behavioral description to a state of the machine. Operation binding assigns each operation to a functional unit (component) that can perform the operation. Register binding assigns registers to those variables (or signals) which have to be stored across states. Finally connection binding refers to the task of synthesizing connections (wires, buses) between allocated functional units and registers.

Bindings come in three flavors: they may be implied in the model, they may specified by the user (in the behavioral description), or they may be synthesized from the behavior automatically. If a user specifies the bindings in the behavioral input, he or she is making a lot of the synthesis decisions. This burdens the user with the task of understanding the synthesis process. On the other hand, if the system has to perform the bindings automatically, the compiler has to incorporate the knowledge of the synthesis tasks. This defines a tradeoff between compiler complexity and the user's responsibility for the tasks of binding. Each system has a different mix of these bindings, which reflects the tradeoff between the user and compiler complexity .

### 3.1.9. Extensibility

This is closely related to the concept of evolutionary design. An extensible system permits the incorporation of synthesis tasks (eg. bindings) as these tasks become well understood. The user may specify a fully bound structural design in the input language. An extensible system permits gradual removal of these bindings from the input description and incorporation of automatic synthesis tools to perform the task.

## 3.2. Previous Work

Existing input languages fall into two major classes: *textual* languages, which are primarily used for synthesis and simulation; and *graphic* languages, which are primarily used as an aid for manual design.

### 3.2.1. Textual Languages

Textual languages are currently used by most high-level synthesis systems. These languages permit the user to describe the design as an algorithm (for general applications) or as an instruction set (for processors). Existing input languages have several limitations: they describe only synchronous processes driven by a single clock; they permit limited timing and protocol specification; they are hard to use when the design is fairly large. Two basic types of textual languages are currently used in high-level synthesis: *block-oriented* languages, and single-state languages.

#### 3.2.1.1. Block Oriented Languages

ISPS [Barb81] is representative of several block-oriented input languages for synthesis. These languages allow the user to specify the behavior of the design in a basic-blocks fashion, with languages control constructs (IF's, LOOP's) permitting transitions between blocks. As a result, operations within the basic blocks exhibit parallelism as constrained by the data dependencies between their inputs and outputs. Sequencing of different basic blocks is achieved via the language control constructs. Hence the underlying programming language model enforces meta-state binding through the sequencing of basic blocks. Most of the systems using this type of input language assume no state binding within the basic blocks, and use state synthesis algorithms for this purpose.

These languages are often suited towards the description of instruction-set processors, since control constructs like "DECODE" can be used to describe decoding of the processor instructions, without specifying state information for the operations that occur during the instruction. Other input languages like ADA [GiBK85] and Pascal [Pang87] [Tric87] are also used.

The CADDY system [Camp85] uses a Pascal-like input language, but extends it to permit the description of "applicative" behavior, which includes asynchronous behavior like sets, resets and interrupts.

#### 3.2.1.2. Single-state Languages

These languages require the user to specify the parallelism in the design explicitly. Hence state binding is done by the user in the input language. Examples of these languages include MacPitts [Sout83] and SILC [BlFR85]. These languages also have primitives for synchronization between processes.

### 3.2.2. Graphic Capture of Behavior

Existing graphic-based input languages are primarily used as an aid for manual design. The ASM [Clar73] methodology uses a graphic representation that is close to software flowcharting. This mimics the human designer's thought process and is thus a convenient form of input specification. However, ASM charts are used to specify the control for a data path that is almost completely designed: the user performs state, unit and register binding; only connections have to be synthesized. Tredennick [Tred81] describes a very similar flowcharting process for microprogramming.

Drongowski [Dron83] proposes the use of the "d-n" notation for providing a structured design methodology and notation in graphical environment. This methodology has been used to create a graphical hardware design language [Dron88]. At the time of writing, not much information was available about this effort, although it seems similar in spirit to that of EXTEND.

Graphical capture enhances user-interaction and permits a more natural means of design entry. It is also a good vehicle for describing the overall structure and partitioning of a design. However, the abstract behavior is more conveniently expressed in textual form (as expressions or statements).

## 3.3. EXTEND-L: The Input Language

### 3.3.1. Overview

EXTEND-L is a mixed graphic/textual language that permits the user to describe the design in a natural fashion. A design entity is described with the *input definitions* and the *behavior* as processes which operate on the defined structures and variables. Both the definitions and the behavior of a design to be synthesized are specified in a mixed graphic/textual input form. For each design, a set of declarations specify the inputs, outputs and variables to be used. Optionally, the user may specify structural information such as the type and number of units. The behavior of a process is specified using a graphical control flow format, along with textual expressions for operations. The control flow of the process is captured through an interconnection of graphic icons. This control flow specifies the states and their transitions for the process. Corresponding to each control flow template, data operations are expressed in a textual form.

For a detailed description of the input language syntax, the reader is referred to Appendix A which contains EXTEND-L's grammar in BNF notation.

### 3.3.2. Definitions

The definitions may be broadly categorized into four classes: *type*, *structural*, *behavioral* and *bindings*.

#### 3.3.2.1. Types

Type declarations allow the user to define new data types in the system. Each type definition can refer to previous type definitions. Primitive type definitions include bit-type, array-type, and component-instantiation type. Simple examples follow:

```
type
        SIXTEEN BIT = {15..0};
        1K_ARRAY    = [0..1023] of SIXTEEN_BIT;
        CMP_FOUR    = CMP(4,EQ,GT,LT);
```

The 1K_ARRAY type specifies 1024 locations of sixteen bits each, while CMP_FOUR specifies a comparator (CMP) instantiated with 4 bits and the functions "EQ", "GT" and "LT".

### 3.3.2.2. Structural Declarations

Structural declarations allow the user to prespecify structural components such as registers, function units and buses which are used in the design. This creates a partial structural design on which further synthesis is performed.

#### 3.3.2.2.1. Components

Appendix B gives a description of some generic components that may be used in the synthesis task. Each component is instantiated by specifying a call to the generic component name, along with its instantiation parameters. Components may be instantiated directly, or indirectly (through a previously defined "type"). For instance,

```
component
        COUNTER    = UP_DWN_CNT(4,UP,DOWN,LOAD,RESET,SET,ENABLE);
        COMP : CMP_FOUR;
```

specifies a four-bit up down counter named "COUNTER" which is instantiated here, while COMP is of type CMP_FOUR as described earlier.

#### 3.3.2.2.2. Ports

Ports specify the locations through which the process communicates with the other processes. A port is declared with the following attributes:

(1) mode = (input/output/input-output)

(2) gating = (tristate,wired).

(3) storage = (buffered/unbuffered)

A pre-defined type may be used to specify the nit-width of the port. A port specified with its mode only is assigned default values for gating (wired) and storage (unbuffered). The following is a sample port definition:

```
port
        APORT, ZPORT   : input of EIGHT_BIT;
        BPORT          : input wired of EIGHT_BIT;
        CPORT          : input tristate of TWO_BIT;
        DPORT          : output tristate buffered of EIGHT_BIT;
        EPORT          : input_output buffered of EIGHT_BIT;
```

### 3.3.2.3. Behavioral Declarations

The behavioral declarations allow the user to specify abstract data carriers such as variables and special types of operators. These declarations specify the data type, the size and representation of all data carriers. Behavioral elements (eg. variables) are mapped into structural components (eg. registers, buses, wires) during data path synthesis. This process is called *binding* of behavioral elements to structural elements. The binding process is effected in three ways:

(1) *Static binding* associates a structural component with the variable or operator at the time of declaration. Hence this binding is valid throughout the behavioral description.

(2) *Multiple binding* permits the user to change the structural bindings in the behavioral description. For instance, a variable 'A' may be bound to register 'R1' in one state of the process, and to bus 'B1' in another state of the behavioral description.

(3) *Automatic binding* is applied to variables and operators that have not been bound by the user. The synthesis system applies specialized rules and algorithms to perform this binding to meet the design constraints (also called *register and unit binding*).

#### 3.3.2.3.1. Variables

Each variable to be used must be declared with its size (number of bits), type (integer, floating point, etc.) and representation (unsigned, sign-magnitude, 2's complement, etc.). Often, only the size of a variable is specified in the language. The system assumes the defaults "integer" for its type, and unsigned for the representation. If the variable is an array, the lower and upper bounds of the array should be specified. A type-denoter defined earlier may be used for this purpose. Sample variable definitions are shown below:

```
var
        A, B, C     : EIGHT_BIT;
        D, E        : MEM_256;
        SIGNAL      : ONE_BIT;
```

### 3.3.2.3.2. Constants

Constants may be defined by the user; these may be merged into a constant register stack, or may be optimized (and some eliminated) by the synthesis tasks. Constants are defined by specifying the bit width and the value, as shown below:

```
const
        ZERO of EIGHT_BIT = 0;
        TEN  of EIGHT_BIT = 10;
```

### 3.3.2.3.3. Operators

Special types of operators may be defined by the user. Most of the primitive language operators are either binary or unary and generally produce a single output. The user may wish to define multiple input, multiple output operators in the language. These may simplify the task of binding to specific kinds of structural components. A typical example of such an operator is an add performed on an ALU. It takes three inputs (carry_in, A, B) and produces at least two outputs: (sum, carry_out, and status bits). Using the operational primitives in a Pascal-like language, it is impossible to describe such an add operation.

Since the user defines a *new* type of operator into the language, its input-output characteristics and its functionality has to be defined. A simple ALU add operator is defined below:

```
operator:

ALU_PLUS    (inputs:
```

```
                A of EIGHT_BIT;
                B of EIGHT_BIT;
                Cin of BOOLEAN;)
        (outputs:
                S of EIGHT_BIT;
                Cout of BOOLEAN;)
        (operation:
                Cout@S := A + B +  Cin;)
```

#### 3.3.2.3.4. Clocks

The clock for a synchronous system may either be defined internally, or as a set of input ports. In either case, the user must define the characteristics of the system clock by specifying the number of phases, duration of each phase, and the relative delay between successive phases. As with other parts of the definition, this could be entered using a query-like form system which prompts the user for various clock attributes.

### 3.3.2.4. Bindings

Bindings in the definitions permit "static" bindings of behavioral variables to structural components. For instance, a variable may be bound to a shift-register at definition time, or the ALU_PLUS operator could be bound to an instantiated ALU component as shown below:

```
   bind
        A to REG_A;
```

```
B to REG_B;

ALU_PLUS to ALU1;
```

### 3.3.3. Behavior

The description of a process in the design is expressed through graphic capture of sequencing (through control flow icons), and data operations (through textual assignments and expressions).

#### 3.3.3.1. Flow of Control

Control flow captures the sequencing of the design over time at the granularity of "states". In synchronous designs, the state length is fixed and is determined by the synchronous clock, while for asynchronous designs, a state may have a varying time length. The control flow chart specifies the execution of the design on a state-by-state basis, and is similar to flowcharting in conventional programming. Both synchronous and asynchronous behavior can be represented with the control flow chart. Although the basic concept is the same for both, we employ different symbols to represent synchronous and asynchronous control flow charts. Each of these is discussed separately.

##### 3.3.3.1.1. Synchronous Charts

Four symbols are used to specify the synchronous control flow chart: the unconditional box, the conditional test box, the conditional output box and the conditional join box as shown in Figure 1. The *unconditional box* specifies actions that are to occur unconditionally in that state. The *conditional test box* performs a test of some expression (writ-

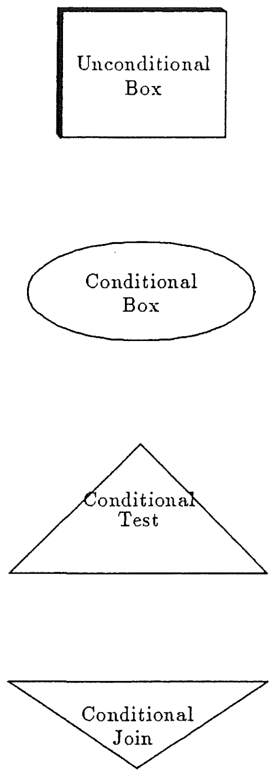


Figure 1. SYNCHRONOUS CONTROL FLOW ICONS

ten as a data flow sequence in the box), based on whose value one of the branches is executed. The *conditional output box* exists as an immediate output of a conditional test box. It specifies the actions to be performed only when the conditional value matches that of the output branch of the conditional test box. The *conditional join box* indicates a merging of several conditional paths.

These symbols are connected by the user in an unambiguous manner to specify the sequencing of the intended algorithm. The user partitions the control flow chart into *states* of the machine. In this version of the system, the user performs state binding by deciding which operations are to be executed in which state. This is done by designating state zones, each of which has unambiguous exits leading to the next state zone, as shown in Figure 2.

This representation can easily be extended to remove the state bindings by having the user describe activities in "macro-states", which are then "sliced" into states by performing state allocation [Pang87].

#### 3.3.3.1.2. Asynchronous Charts

In the asynchronous realm, two concepts are of importance: an *event* and an *event-state.* An event is defined by a change in an input signal (port), and forces the process to

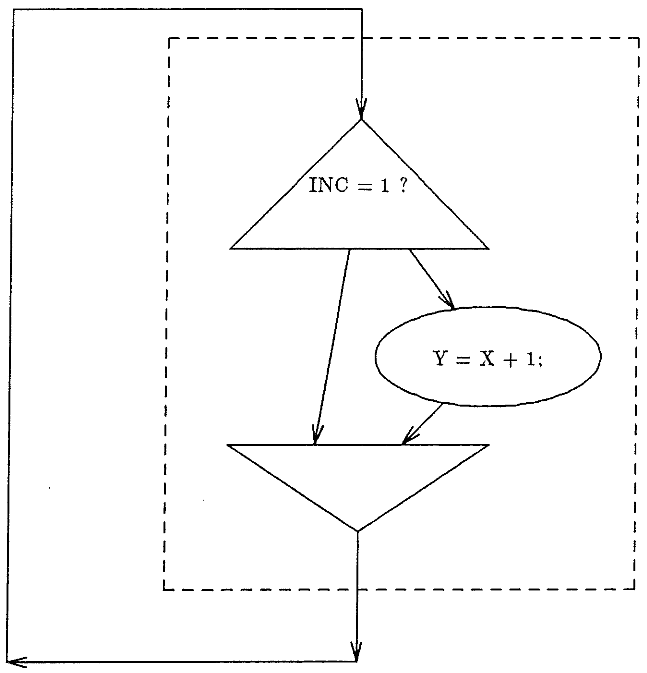


Figure 2. SYNCHRONOUS CONTROL FLOW EXAMPLE

enter a new event-state. An event-state lasts from the time the event occurs until the occurrence of the next event. The user must describe the asynchronous control flow chart from the behavior of the output signals (ports) with respect to the input signals (ports) as a function of increasing time. One way of achieving this is to have the user look at a timing diagram which shows the behavior of the input signals and the behavior of the process in terms of all signals (variables, ports, etc.) that are transformed with respect to the inputs. This is a natural starting point for the design. Figure 3 shows a sample timing diagram for

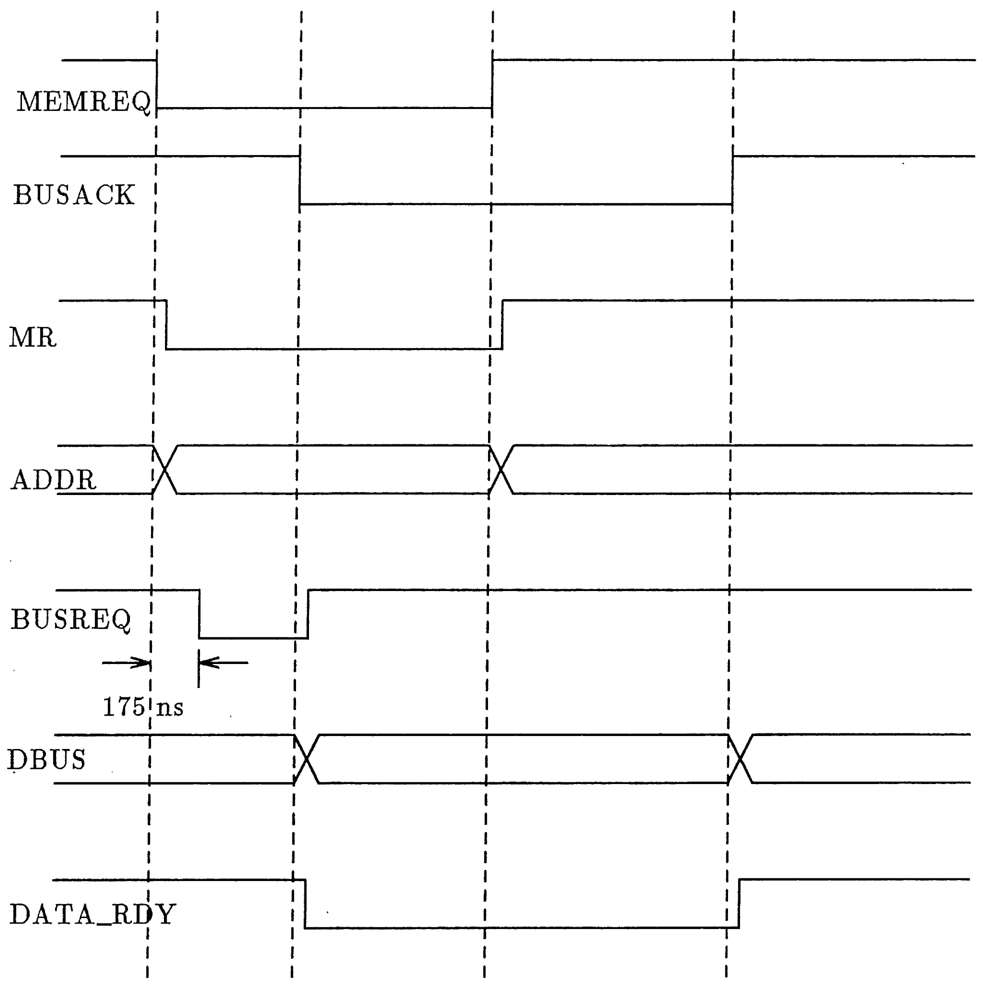


Figure 3. MEMORY CONTROLLER READ CYCLE TIMING DIAGRAM

the read cycle of a memory controller which is activated by two input signals, MEMREQ and BUSACK. Note how every event on the input signal causes a new state to be defined.

Once the states of the process have been determined, an asynchronous control flow chart is drawn. The chart is constructed by connecting event-nodes in the order of their appearance. Each event-node represents the event being tested. Figure 4 shows a sample symbol which indicates that state 5 is entered when the value on input port A rises. Figure 5 shows the asynchronous control flow chart for the sample timing diagram.

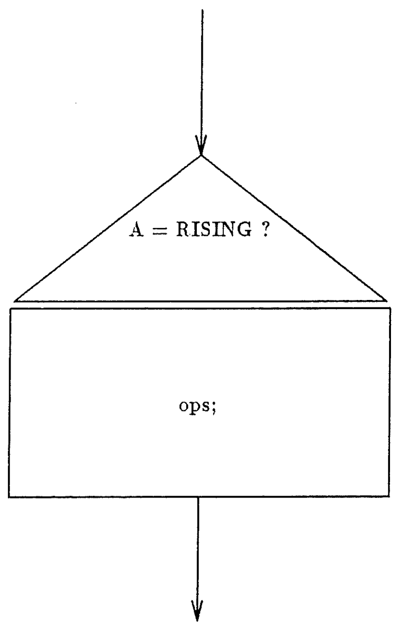


Figure 4. ASYNCHRONOUS CONTROL FLOW ICON

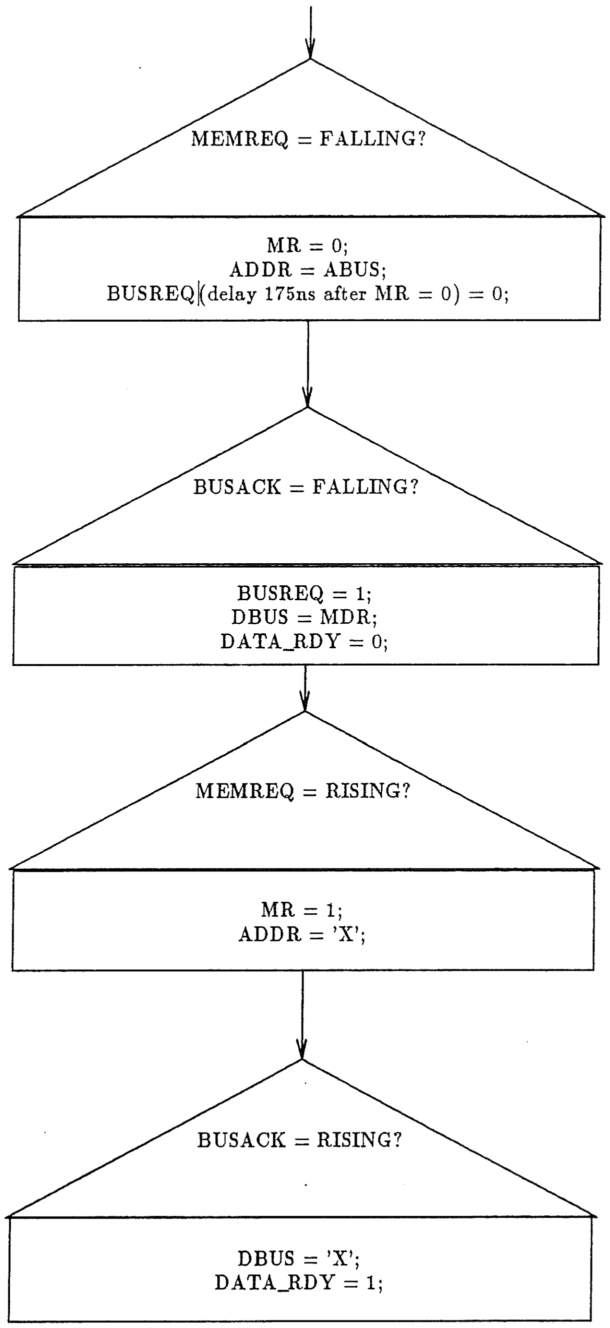


Figure 5. ASYNCHRONOUS CONTROL FLOW EXAMPLE

#### 3.3.3.2. Hierarchy

A control flow node may contain calls to other control flow nodes in the form of procedures and functions. This allows the user to express the design in a hierarchical fashion. For instance, in describing the behavior of a processor, the top level would have control nodes corresponding to the fetch/execute/store phases, while lower levels in the hierarchy would perform the actual data operations and transfers. The structural realization of the hierarchy can be "flat" or interpreted. The flattened implementation expands the control graph and uses a centralized controller for the operations. In the interpretive realization, each level of hierarchy in the control graph has a local controller which communicates with the levels above and below it [JVJC86]. For the first version of our system , we will assume a flattened implementation of the control hierarchy.

**3.3.3.2.1. Procedures and Functions**

Procedures and functions are used simply as a descriptive aid, providing a short-hand notation for repetitious segments of the description. The user may wish to define a main procedure with calls to sub-procedures and functions. This facility simplifies the specification of a process. Each procedure or function will be treated as a macro and will be expanded in-line during the compilation phase. This allows all process declarations to be visible within any procedure or function (these are treated like global variables).

The process window will then consist of a "page" for each procedure or function, with a label identifying the current "page" under examination.

### 3.3.3.3. Data Operations

Data transfers and transformations in the design are performed by various types of operators. Broadly, these may be classified into *arithmetic* operators, *comparison* operators, *shift/rotate* operators, *logical* operators, *bit manipulation* (concatenation/selection) operators, *array references* and *assignment* operators. Since each data carrier is strongly typed, it is not necessary to have special operators to be used with variables, ports and buses of different types. As described later, type mismatches are handled according to certain rules. When a mismatch cannot be resolved, or is erroneous, the system can flag an error to inform the user. Figure 6 summarizes sample data operators of each type.

#### 3.3.3.3.1. Arithmetic

The standard set of arithmetic operators for addition ('+'), subtraction ('-'), multiplication ('*'), and division ('/') may be used. These assume an inputs of the same type, and produce outputs of the same type as the inputs. The shorter inputs are right justified if they are all not of the same bit width.

#### 3.3.3.3.2. Comparison

Comparison operators ('==', '!=', '>', '<') may be used for comparison between variables. The user must make sure that the variables being tested are of the same type and have the appropriate bit widths.

Table Dataops.tbl

| Operator Type | Operator | Target Components |
|---|---|---|
| ARITHMETIC | + | ALU, Adder, Counter |
| | - | ALU, Subtractor, Counter |
| | * | Multiplier |
| | / | Divider |
| COMPARISON | <, <= | Comparator, ALU |
| | >, <= | Comparator, ALU |
| | =, != | Comparator, ALU |
| SHIFT/ROTATE | SHL{0/1} | Shifter, Shift-register |
| | SHR{0/1} | Shifter, Shift-register |
| | ASH{L/R} | Shifter, Shift-register |
| | ROT{L/R} | Shifter, Shift-register |
| BITWISE LOGICAL | & | And-gates, ALU |
| | \| | Or-gates, ALU |
| | ^ | XOR-gates, ALU |
| | ~ | Inverters, ALU |
| | ~& | Nand-gates, ALU |
| | ~\| | NOR-gates, ALU |
| | ^~ | XNOR-gates, ALU |
| BOOLEAN LOGICAL | LAND | And-gate, ALU |
| | LOR | Or-gate, ALU |
| | LNOT | Inverter, ALU |
| | LNAND | Nand-gate, ALU |
| | LXOR | XOR-gate, ALU |
| | LXNOR | XNOR-gate, ALU |
| CONCATENATE | @ | Switchbox |
| SELECT | {XX..YY} | Switchbox |
| ARRAY REFERENCE | [ ] | Memory, register file |
| ASSIGNMENT | := | |
| ASYNCHRONOUS ASSIGNMENT | <:= | |

Figure 6. DATA OPERATIONS AVAILABLE

#### 3.3.3.3.3. Logical

Logical operators include 'AND', 'OR', 'NAND', 'NOR', 'XOR' and 'XNOR'. These operators work on a bit-by-bit fashion on the operands.

#### 3.3.3.3.4. Boolean

These are unary operators that perform "concentrator" logic functions by subjecting all the bits of the single input to the logical operation and producing a boolean output.

#### 3.3.3.3.5. Shift/rotate

These operators perform arithmetic shifts, logical shifts, and rotations of the operator's first argument of the operator. The second argument specifies the distance for the shift or rotation. For example, A SHR0 2 performs a logical shift right of A by 2 bit positions, using a fill of 0.

#### 3.3.3.3.6. Bit manipulation

Bit extraction and insertion is achieved with the pair '{..}', with the high and low bit positions to be extracted specified within the curly braces. For example, A{8..5} appearing on the right hand side of an expression would extract bit positions 8 through 5 of the variable A. The construct A{4..2} appearing on the left hand side of an expression would replace the old value of positions 4 through 2 in A with the result of the computation on the right hand side.

The '@' operator is used to concatenate two bit strings to produce a new bit string whose length is the sum of the two input lengths. For instance, A{8..5}@B{2..0} would produce a carrier 7 bits long.

##### 3.3.3.3.7. Array references

Array are referenced with the '[' and ']' pair. Since an array most often gets bound to a memory or a register file, it may be necessary to qualify the access with the associated port of the structural module. For example, if array A[0..255] is bound to a single port memory, the reference A[5] accesses the memory with the address 5.

##### 3.3.3.3.8. Assignment

The '=' operator is used for assignment. The '=' operator assigns a value to a variable. The semantics of the assignment statement depends on the data types involved, so it is important to make sure that the operands are of similar types. Violations may be detected by the compiler and flagged as errors.

#### 3.3.3.4. Timing

##### 3.3.3.4.1. Types

Two types of timing specification are supported in this system. The first is a *path-relative delay* which expresses the delay from one point in the structural implementation to another. This delay is the sum of the transmission delays on the wires, the operation times for components and the set-up and hold times for registers that exist on the physical path between the two points.

The second is event-relative delay which expresses the delay for one event with respect to another. An *event* corresponds to the change in the value of particular signal (or of a set of signals). Since two events may be logically unrelated, event-relative timing specifies the exact sequencing of the two signals waveforms. This is often used in describing protocols which involve two or more signals that are not data dependent, but which must follow a particular behavior over time to ensure correct behavior.

In synchronous systems, the major event is the system clock. All actions are performed on a state-by-state basis where the rising or falling edge of the system clock initiates a new state. Hence event-relative delays in the synchronous case refer to delays specified with respect to the rising or falling edge of the system clock.

### 3.3.3.5. General Form and Semantics of the Assignment Statement

A general form of the assignment statement permits the user to express both kinds of timing constraints in a concise notation. The general form is:

carrier | event-constraint := expression | path-constraint;

where:

*expression* is a standard expression using the operators previously defined;

*path-constraint* is a delay specified from some input (of the expression or port) to the carrier on the left hand side;

and

*event-constraint* is a set of delays which specify when the signal on the left hand side should receive the computed value on the right hand side, with respect to the specified event.

Three notions are of importance here: **relativity, duration** and **event-cause.** Relativity specifies the change of one signal with respect to another. If the two signals are data dependent, we call this delay specification a path constraint and use the keyword **from** to indicate the relativity of the delay. If the two signals are data independent, we call this delay an event-relative delay and use the keywords **before** or **after** to specify event-relativity.

Delays may be if **minimum, maximum** or **nominal** duration. A nominal duration is an "average" value with a certain tolerance. A delay no specified too be of a particular duration type defaults to maximum for path-relative delays, and minimum for event-relative delays.

Event-cause specifies the characteristics of the event as being of type **rising, falling** or **changing**.

The versatility of this assignment construct permits the designer to specify timing at various levels: combinatorial delays, delays relative to clock phases, and asynchronous assignments. Each of these is illustrated with examples in the following sections.

#### 3.3.3.5.1. Path Constraints

A path constraint permits the user to specify the delay on a path from an input to the element on the left hand side. In this version of the system, since the user performs state-binding this delay is used to capture the combinatorial delay on the path from the input to the output of the expression. However, a general path constraint can specify delays from inputs to outputs over several assignments (and over several states). The syntax of each path constraint is:

**delay** *delay-value* **from** *input*

For instance, if A and B are register and INPORT is an input port, the statement

B := INPORT + A, delay 80 ns from INPORT, 40 - 60 ns from A;

specifies a delay of 80 ns maximum (by default) from the input port INPORT to the register B, and a delay of 40 ns minimum, 60 ns maximum from the output of register A to the register B. These delay constraints may be passed on to the module generator for the function '+'.

### 3.3.3.5.2. Event Constraints

An event constraint specifies a delay for the output of an expression with respect to a change in some signal (which is often data independent). The syntax of an event-constraint is:

**delay** *delay-value* {**after** or **before**} *event-cause*

where delay, as before, is a **minimum, maximum** or **nominal** delay, and event-cause is a signal **rising,, falling** or **changing**. This type of timing constraint is most often used to capture timing chains from a timing chart, which specifies the change of one signal with respect to another over time.

For example, if A is an output port and B is an input port, the statement

A | delay 100-1500 ns after B rising := X + 1;

specifies that the port A be assigned the value "X + 1" with a delay of 100 to 1500 ns after the value on port B rises.

Clock phase assignments are also achieved with this construct. For example, if R and Q are registers, and the system clock is 2-phase (with names phase-1 and phase-2), the statement:

R | (after phase-2 = rising) := Q;

assigns the value in register Q to register R in phase 2 of the system clock.

### 3.3.3.5.3. Asynchronous Assignments

When an asynchronous assignment to a variable has to be described, a special assignment operator, '<=', is used to indicate this. Semantically, the asynchronous assignment implies the use of an asynchronous input on the structure bound to the variable, to achieve the assignment. Most often, this type of assignment is used to clear or set a register asynchronously. For instance, if R is a variable bound to a register and RESET is defined to be an input port, the statement

R | (RESET = rising) <:= 0;

ties the RESET line to the 'clear' input of the register R.

# CHAPTER 4.

# EXAMPLES

This chapter illustrates the use of EXTEND-L to describe two designs: a simple controlled counter and a simple UART.

## 4.1. Controlled Counter

### 4.1.1. Principles of Operation

Figure 7 shows the block diagram of a process that we will call a controlled counter. This example was obtained by abstracting the behavior of the VHDL structure of the con-

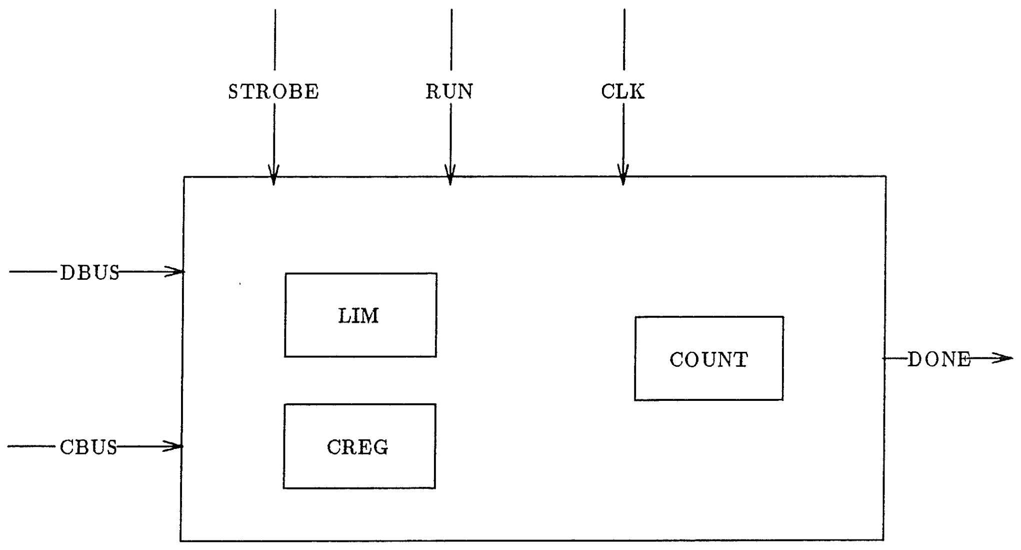


Figure 7. BLOCK DIAGRAM OF CONTROLLED COUNTER

trolled counter defined in [Arms88]. Its basic operation is sketched in Figure 8. On the rising edge of the signal STROBE, an internal control register is loaded with the value on CBUS. The value in the internal control register is decoded to perform one of four functions: clear the counter, load a limit register, count up till limit, or count down till limit. The counter runs synchronously under the input clock, and the counting functions are performed only when RUN is high.

#### 4.1.1.1. Declarations

Figure 9 shows the definitions for the process. Two registers, LIMIT and CREG, are defined. The input ports consist of STROBE, RUN, CLK, DBUS and CBUS, while the output port is DONE. The port definitions specify the width, type and direction of the ports for the process. For synchronous operation, the clock has to be defined explicitly. In this example, the system clock is defined to be 1 phase, with the source being the input port

```
When STROBE rises, load CREG with CBUS;
  while RUN is asserted,
  if CREG = '00', clear COUNT;
  if CREG = '01', load LIM with DBUS
                on falling edge of STROBE;
  if CREG = '01', count up until LIM reached;
  if CREG = '11', count down until LIM reached;
  set DONE to 1 when count is finished;
```

Figure 8. CONTROLLED COUNTER OPERATIONAL PRINCIPLES

```
type
        BOOLEAN      = {0};
        TWO_BIT      = {1..0};
        FOUR_BIT     = {3..0};
        REG_TWO             = REGISTER(2,LOAD,,,,RESET,ENABLE);
        REG_FOUR     = REGISTER(4,LOAD,,,,RESET,ENABLE);
        DEC_TWO             = DEC(2,4);
        CMP_FOUR     = CMP(4,,GT,LT);
        CNT_FOUR     = UP_DWN_CNT(4,UP,DOWN,LOAD,RESET,SET,ENABLE);

component
        CREG         : REG_TWO;
        LIMIT        : REG_FOUR;
        COUNT               : CNT_FOUR;
        COMP         : CMP_FOUR;
        DECODER             : DEC_TWO;

port
        CBUS         : input of TWO_BIT;
        STROBE, RUN         : input of BOOLEAN;
        DBUS         : input of FOUR_BIT;
        DONE         : output of BOOLEAN;

clock
        CLK          : port;

var
        LOAD_LIM, UP, DOWN      : BOOLEAN;

const
        ZERO of FOUR_BIT = 0;
        B_ONE of BOOLEAN = 1;
        ONE of  FOUR_BIT = 1;
```

Figure 9. CONTROLLED COUNTER DEFINITIONS

CLK. Two variables, COUNT and LIM_EN, are also defined.

### 4.1.1.2. Behavior

The behavior of this simple process can be expressed in many ways. For illustration, we choose to describe the behavior of the controlled counter with an asynchronous behavior chart. We will embed the synchronous behavior of the counter by explicitly specifying its clocking requirement. Figure 10 shows the asynchronous chart. Two main events can be recognized in this example: STROBE rising and STROBE falling. We therefore describe the behavior in each of these event-states.

When STROBE rises, CREG is asynchronously loaded with the value on CBUS. Next, based on the value in CREG, either COUNT is cleared, the limit register is enabled, or the count-up/count-down sequence is initiated by setting the signals UP or DOWN high.

When STROBE falls, if the LIM_EN signal is high, the LIMIT register is loaded asynchronously with the value on DBUS.

The synchronous behavior is described by the synchronous chart, shown in Figure 11. "COUNT_UP" is a one state loop with several synchronous control icons. First, based on the concatenated value of the signals RUN, UP and DOWN, one of three branches is taken: if RUN and UP are high, the counter counts up; if RUN and DOWN are high, the counter counts down; in all other cases, the counter busy-waits. When either the count-up or count-down sequence is completed, the DONE signal is set to 1 to indicate completion of the counting task.

### 4.1.1.3. Structure Generated

Figure 12 shows the structure that is generated after synthesis from the description. In this example, the synthesis task is quite straightforward as there are not many event-states synchronous states and variables (or defined structures).

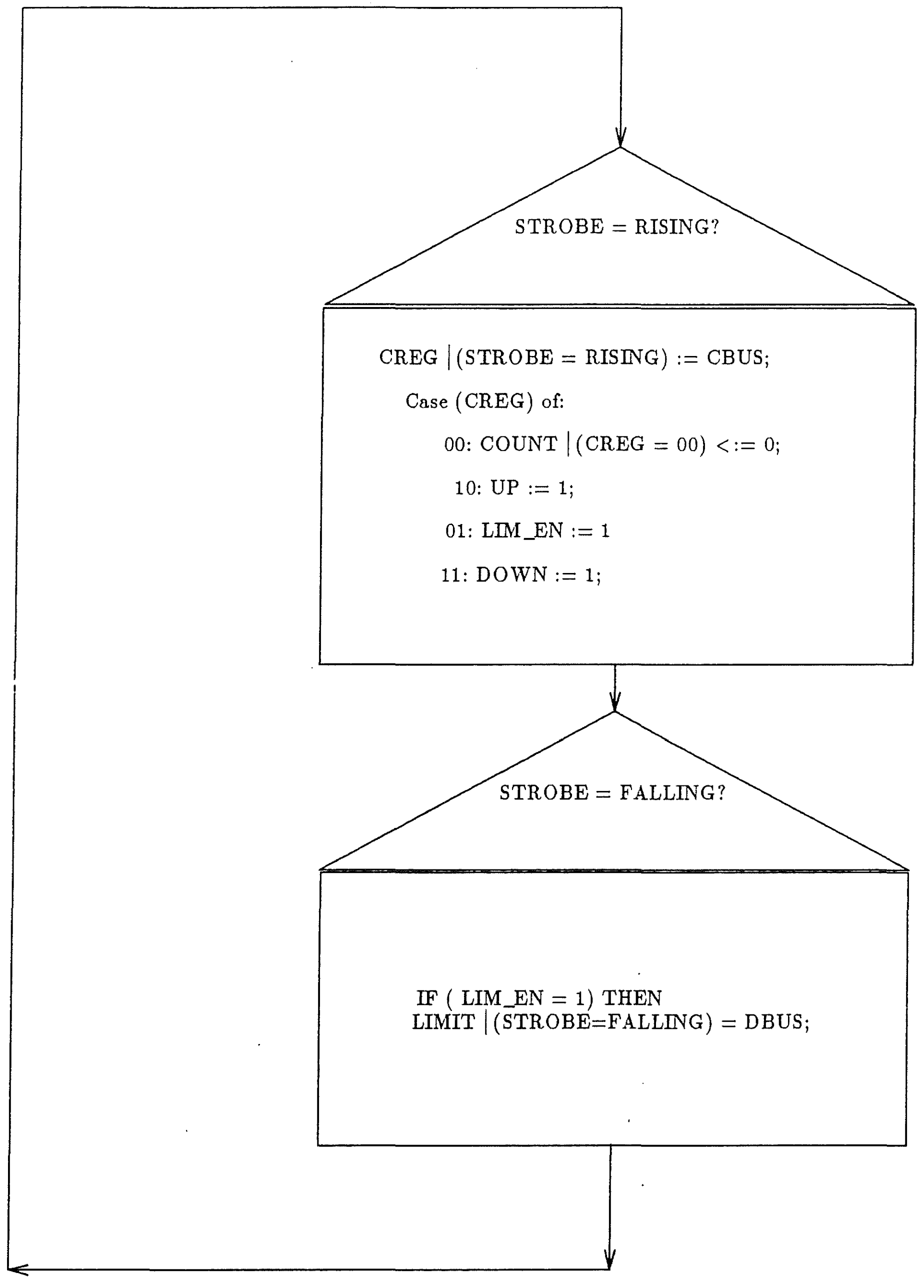


Figure 10. ASYNCHRONOUS CHART FOR CONTROLLED COUNTER

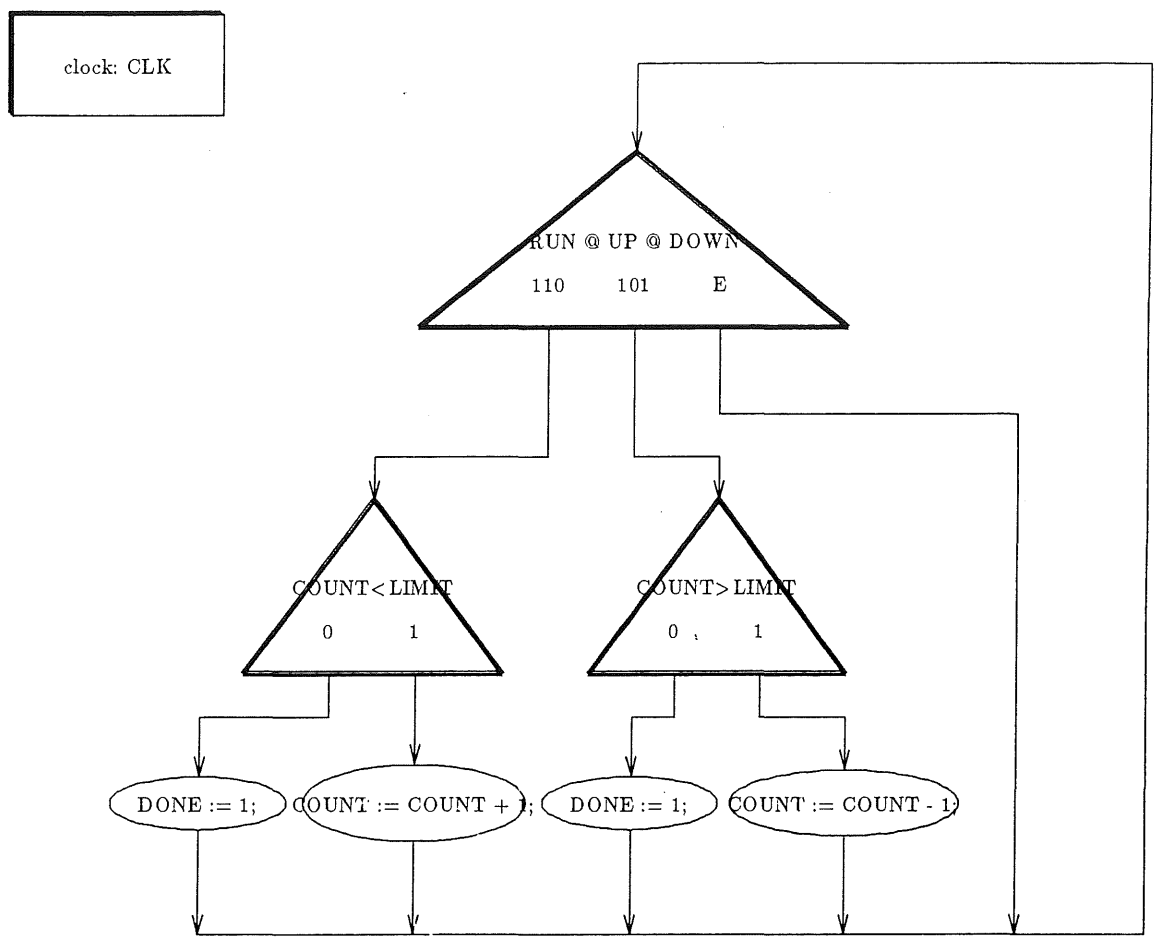


Figure 11. CONTROLLED COUNTER SYNCHRONOUS CHART

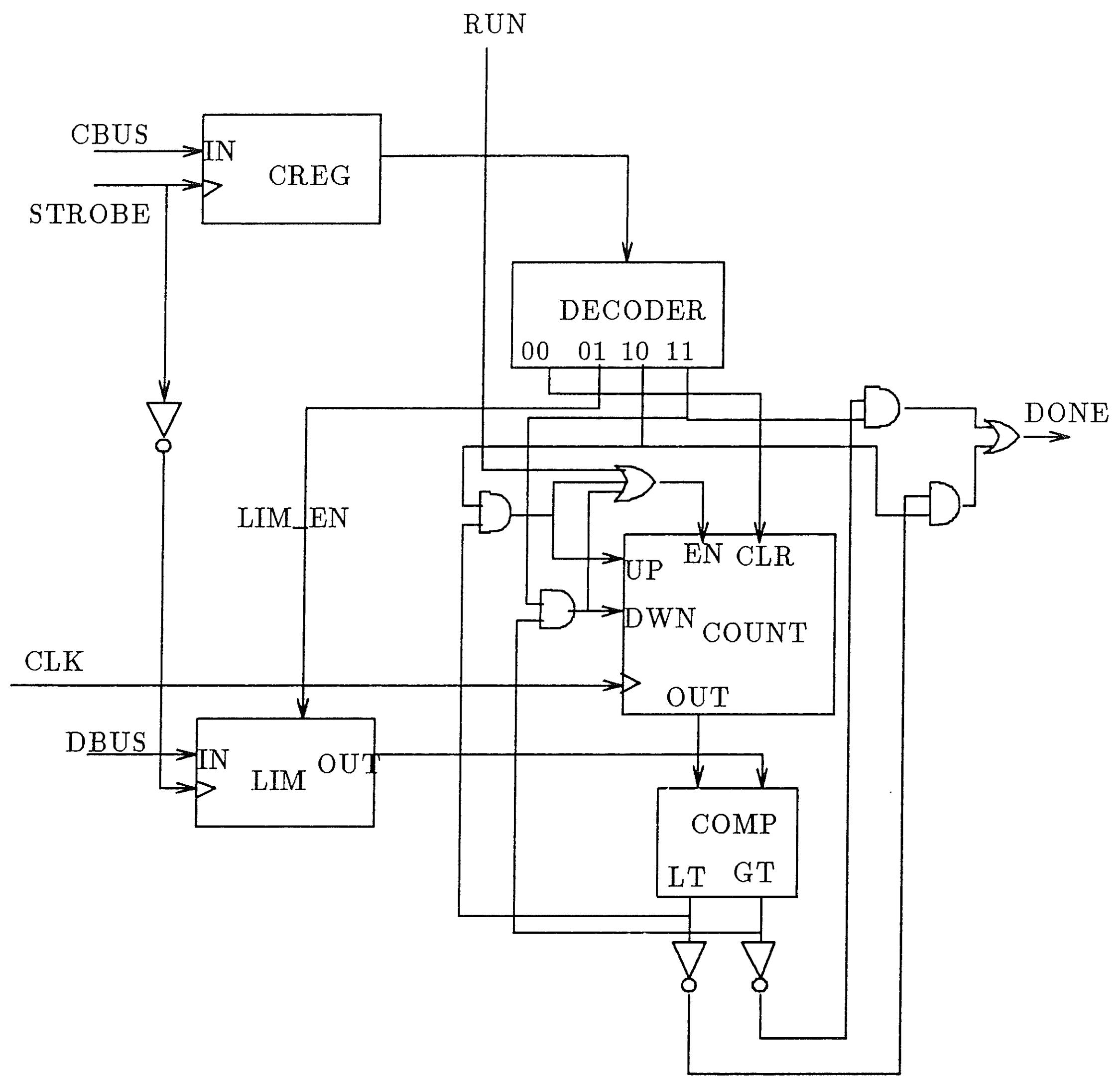


Figure 12. CONTROLLED COUNTER: GENERATED STRUCTURE

The case statement in the asynchronous chart gets compiled into a decoder, while the conditional test for LIM_EN in event-state 2 of the asynchronous chart gets compiled as the enable line for loading the limit register (on the falling edge of the STROBE line).

In the synchronous chart, the test for ">" and "<" get bound to a comparator, while the count-up and count-down functions are compiled into activating the control lines "UP" and "DOWN" for the counter. The counter is also enabled when one of the two procedures (count_up or count_down) is invoked.

#### 4.1.1.4. Comments

This example shows the power of the input description: synchronous and asynchronous behavior is described together in a natural fashion; the resulting description is quite concise and easy to compile. The user can see the behavior and make any modifications easily.

In contrast, the same example would require several pages of VHDL text to describe [VHDL87]. The VHDL description is bulky and cumbersome. The user does not have an immediate feel for the design just by looking at the VHDL description. Most of the existing high-level input languages are not capable of describing this kind of mixed behavior (synchronous and asynchronous), and do not permit mixed behavioral and structural description.

## 4.2. 6850 UART

### 4.2.1. Principles of Operation

Figure 13 shows the block diagram of the Motorola 6850 Asynchronous Communications Interface Adapter (ACIA), popularly referred to as a Universal Asynchronous Receiver-Transmitter (UART). Its basic function is to interface serial I/O devices to a microprocessor. The UART converts 8-bit parallel data from the processor into a serial data stream for the serial I/O device in "transmit" mode. In "receive" mode, the UART converts serial data from the I/O device into an 8-bit parallel word that the processor can understand.

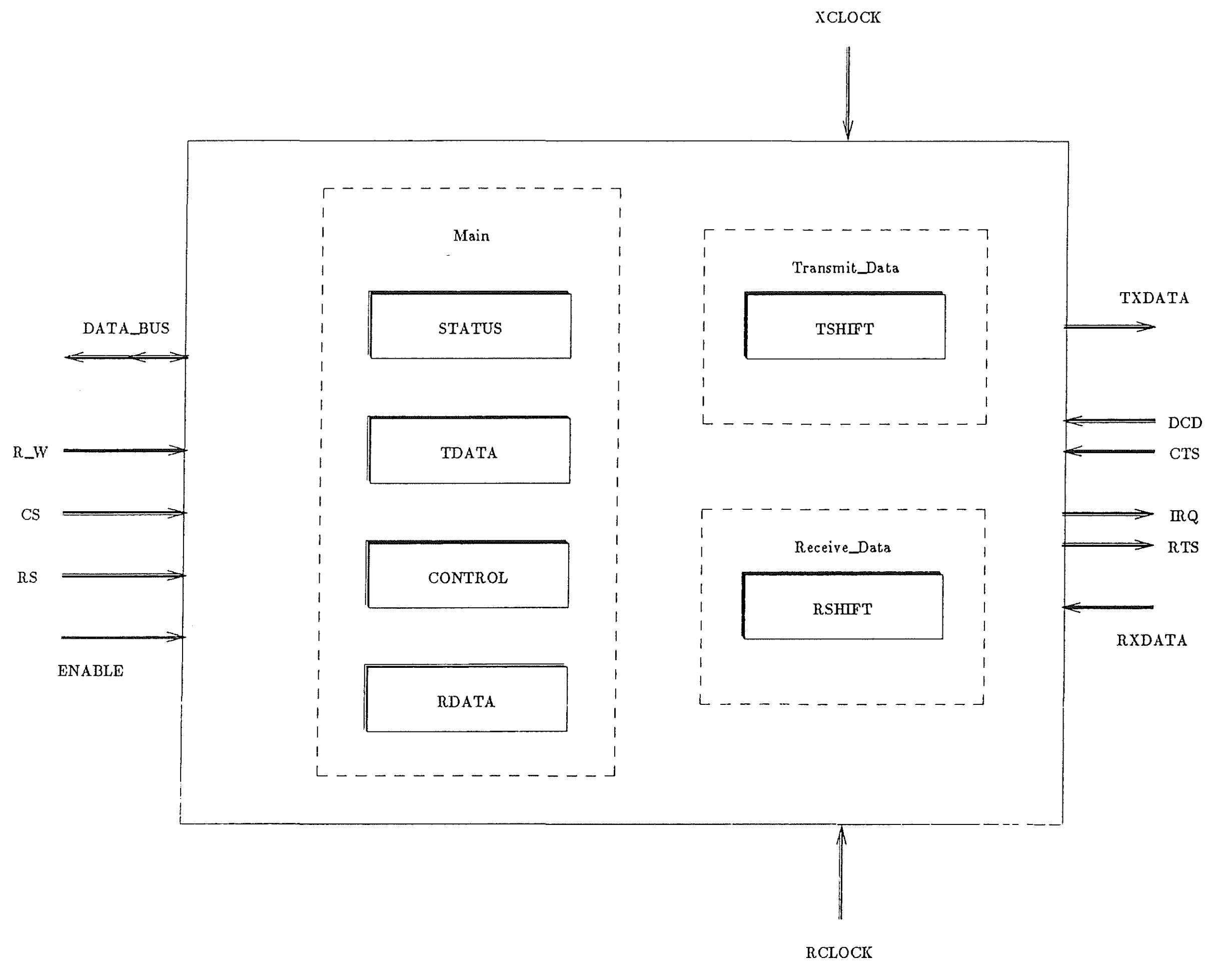


Figure 13. BLOCK DIAGRAM OF THE MOTOROLA 6850 UART

In this section, we describe a stripped down version of the 6850 that performs only the receive and transmit functions, without performing error checking. The description of the UART can be entered in several different forms, depending on how the user performs the initial system partitioning. In this example, we will treat the 6850 as three concurrent processes labelled "Main", "Receive_Data" and "Transmit_Data". The operation of "Main" is completely asynchronous, while "Transmit_Data" and "Receive_Data" are synchronous, clocked by TCLOCK and RCLOCK respectively. All the three processes operate on the structures and data carriers defined in the definitions section, and hence refer to the same variables and structures.

"Main" is the process that communicates with the processor and receives/sends data words in parallel. "Transmit_Data" is the transmitter process which converts a parallel word into a bit stream for the serial I/O device. "Receive_Data" is the receiver process which accepts a bit stream from a serial I/O device and converts into a parallel word for the processor.

### 4.2.2. Declarations

Figure 14 shows the definitions for the 6850. STATUS, CONTROL, TDATA and RDATA are four registers instantiated for the main process. TSHIFT and RSHIFT are eight bit shift registers, while TCNT and RCNT are four bit counters used for the transmit and receive functions. The STATUS register is eight bits wide, with each bit containing specific information about the status of the 6850. As shown in the "bind" section of the definitions, register RXRDF is connected to bit 0 of STATUS: this bit indicates if the receiver shift register, RSHIFT, is full (signalling start of the receive function). Likewise, register TXRDE is connected to bit 1 of STATUS; when TXRDE is high, it signals the main process to transfer another word from TDATA to TSHIFT and begin a new transmit cycle.

### 4.2.3. Behavior

The behavior of the UART is described using an asynchronous state chart and two synchronous charts. These correspond to the dotted partitions in the block diagram which

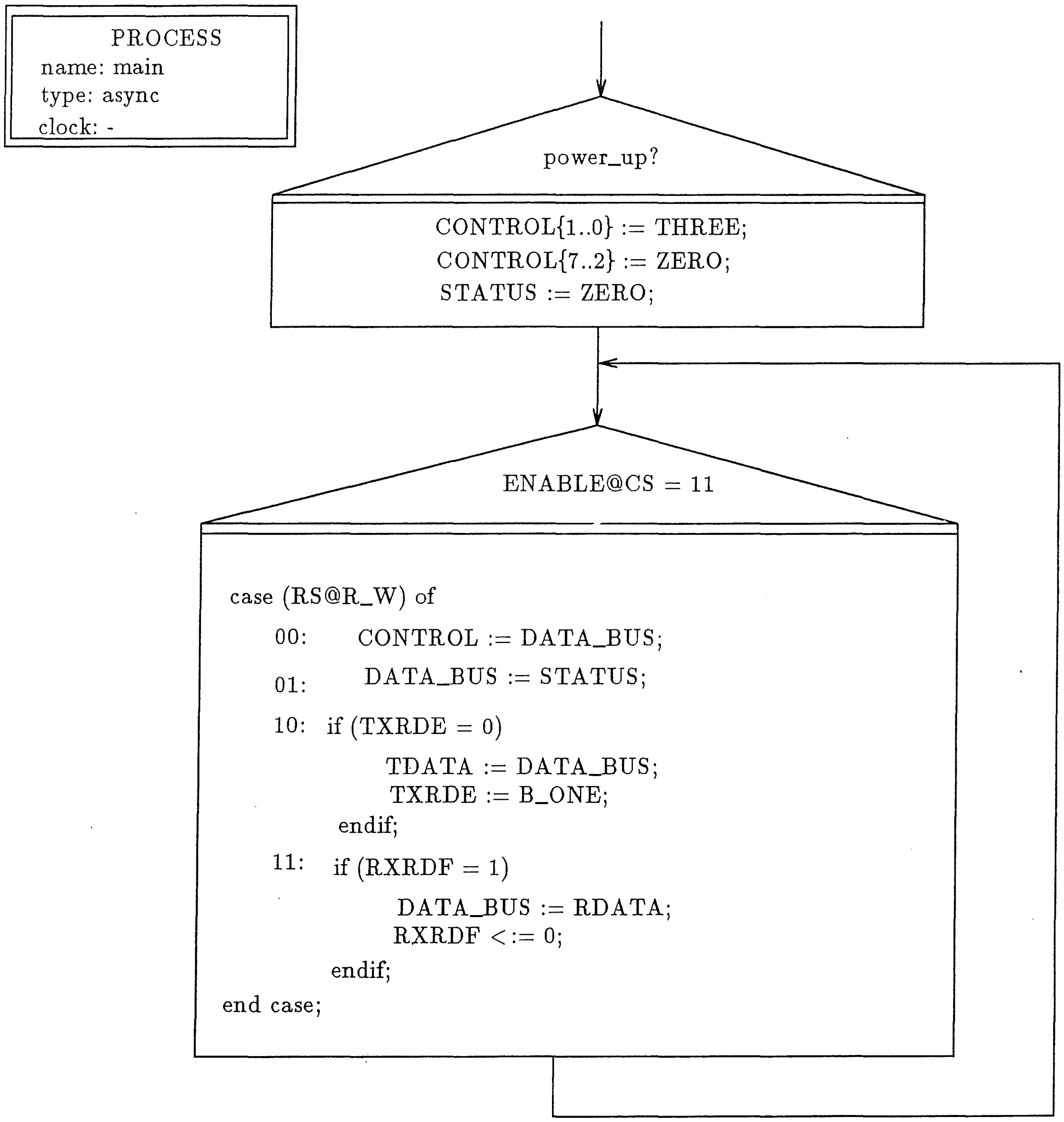
```
type
          BOOLEAN = {0};
          EIGHT_BIT = {7..0};
          TWO_BIT = {1..0};
          FOUR_BIT = {3..0};
          REG_1 = REGISTER(1,LOAD,CLEAR, , , ,ENABLE);
          REG_8 = REGISTER(8,LOAD, , , , ,ENABLE);
          SHIFT_8 = REGISTER(8,LOAD,SHL,SHR, , ,ENABLE);
          COUNT_4 = UP_DWN_CNT(4,UP,DOWN,LOAD, , ,ENABLE);
component
          STATUS, CONTROL, TDATA, RDATA  : REG_8;
          TSHIFT, RSHIFT  : SHIFT_8;
          TCNT, RCNT  :  COUNT_4;
          TXRDE, RXRDF :  REG_!;
port
          DATA_BUS  : input_output of EIGHT_BIT;
          R_W, CS, RS, ENABLE, XCLOCK, RCLOCK,
                    RXDATA, DCD, CTS  : input of BOOLEAN;
          TXDATA, IRQ, RTS  :  output of BOOLEAN;
clock
          XCLOCK, RCLOCK  :  port;
var
          START_FLAG :  BOOLEAN;
const
          B_ZERO of BOOLEAN  =  0;
          B_ONE of BOOLEAN  =  1;
          THREE of TWO_BIT  =  3;
          ONE of FOUR_BIT  =  1;
          EIGHT of FOUR_BIT  =  8;
bind
          TXRDE  to  STATUS{1};
          RXRDF  to  STATUS{0};
          IRQ    to  STATUS{7};
          CTS    to  STATUS{3};
          DCD    to  STATUS{2};
```

Figure 14.  6850 DEFINITIONS

are labelled "Main", "Transmit_Data" and "Receive_Data".

### 4.2.3.1. Main Process

Figure 15 shows the asynchronous chart for the 6850 main process. When the UART



```
case (RS@R_W) of
     00:      CONTROL := DATA_BUS;
     01:       DATA_BUS := STATUS;
     10:  if (TXRDE = 0)
               TDATA := DATA_BUS;
               TXRDE := B_ONE;
          endif;
     11:  if (RXRDF = 1)
               DATA_BUS := RDATA;
               RXRDF <:= 0;
          endif;
end case;
```

Figure 15. 6850 Main Process

is powered up, it enters the MASTER RESET state. In this state, all bits of the control and status register are set to zero, and the registers TXRDE and RXRDF are set to 1. These actions are described in the first event-state.

Subsequently, the UART has only one event-state: it functions only when both ENABLE and CS (Chip Select) are set high. Based on the value of the RS (Register Select) and R_W (Read/Write) lines, one of four sets of actions are performed.

When RS@R_W is '00', the DATA_BUS has the control word on it; this is loaded into the CONTROL register.

When RS@R_W is '01', the value in the STATUS register is put on the DATA_BUS.

When RS@R_W is '10', the UART is ready to execute a data-transmit cycle. If TXRDE is set to 1, the main process has to wait until the transmit process sends out the previous word. When TXRDE is set to 0, a new word can be transmitted. TDATA is loaded with the value on the DATA_BUS, and TXRDF is set to 1. This signals initiation of the transmit function.

When RS@R_W is '11', the UART is ready to execute a receive cycle. If RXRDF is set to 0, the reciever has not yet filled the receiver shift register with a word; hence the main process waits. After RXRDF is set to 1, the main process is ready to send out the received word. This word is sent out on DATA_BUS and RXDRF (connected to STATUS bit 0) is reset to indicate that the receiver buffer RSHIFT is empty.

### 4.2.3.2. The Transmit_Data Process

Figure 16 shows the synchronous process for data transmission. As indicated by the box on the upper left-hand corner, this is a synchronous process whose clock is XCLOCK.

The transmit sequence starts only when RS@RW equals "00" and TXRDE is "1" (indicating that the transmit register is loaded with a new word). The transmitter then moves

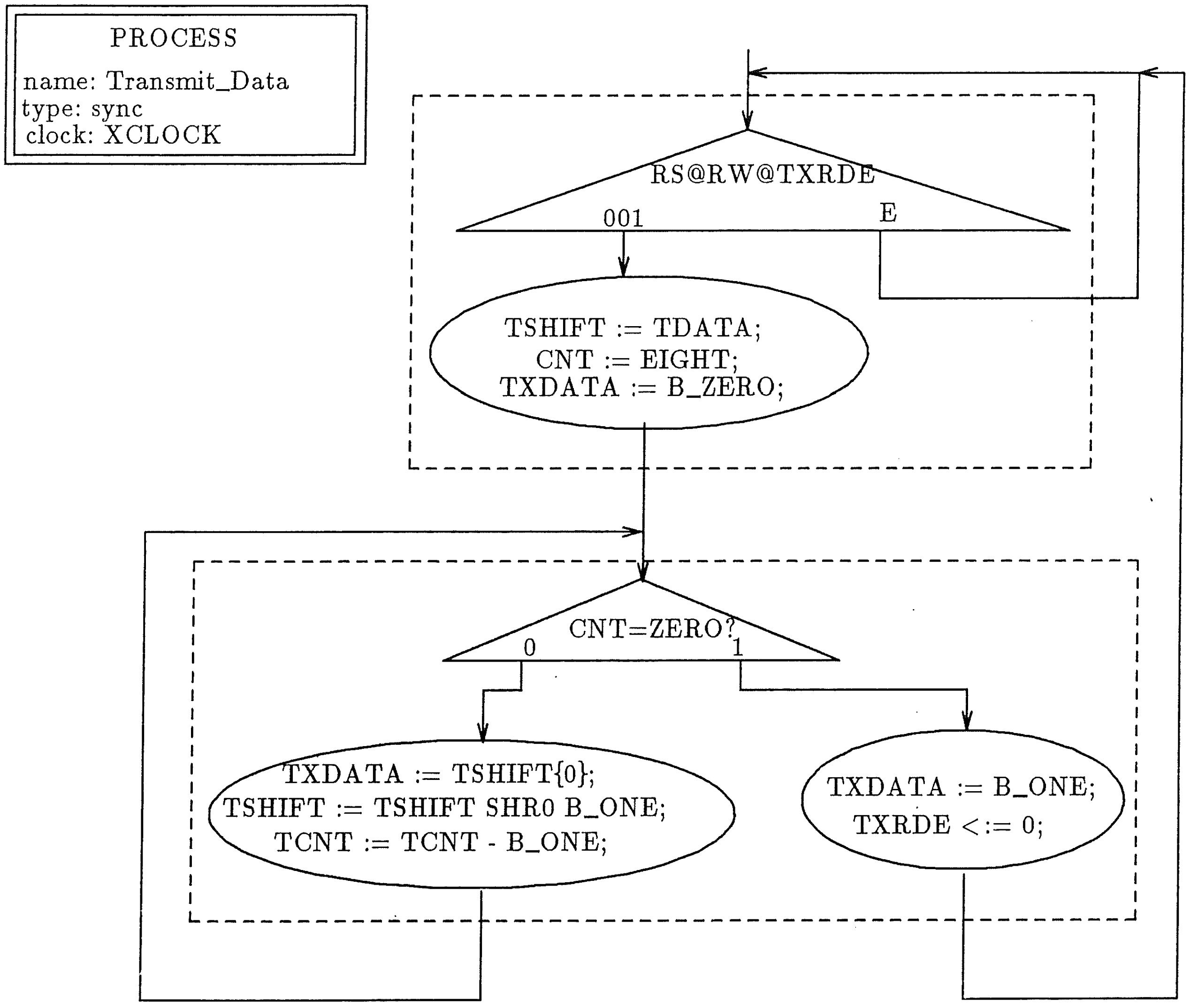


Figure 16. UART Transmit_Data PROCESS

the word to be transmitted from the register TDATA to the shift-register TSHIFT. The counter is set to eight, and a start bit is sent out on TDATA.

The next state is a loop in which the bits in TSHIFT are shifted out. While the value in the counter has not reached 0 (the body of the loop), the least significant bit of TSHIFT is sent out on the port TXDATA. When all the bits in TSHIFT have been shifted out (loop exit), a stop bit is sent out on TXDATA. The TXRDE flag is set to "0" to indicate completion of transmission.

#### 4.2.3.3. The Receive_Data Process

Figure 17 shows the synchronous process for Receive_Data. Receive_Data is clocked by RCLOCK, and RSHIFT is the shift-register used for the serial-to-parallel conversion.

The receiver first waits for the main process to finish transferring a previous word by checking for (RXRDF = 0) and (RS@RW = 01). At this time, it sets START_FLAG to 0. In the next state, the receiver loops until a start bit is detected in the serial input. The loop is terminated by setting START_FLAG to 1.

After the start flag is detected, the receiver proceeds to shift in 8 bits from RXDATA into RSHIFT. Finally, the newly-assembled word is moved from RSHIFT to RDATA and RXDRF is set high to indicate that a new word has been loaded into RDATA.

### 4.2.4. Structure Generated

Figure 18 shows the structure generated by the synthesis system for the UART. Each of the processes "Main", "Transmit_Data" and "Receive_Data" has its own control generated. They communicate with each other through the registers "TXRDE" and RXRDF"

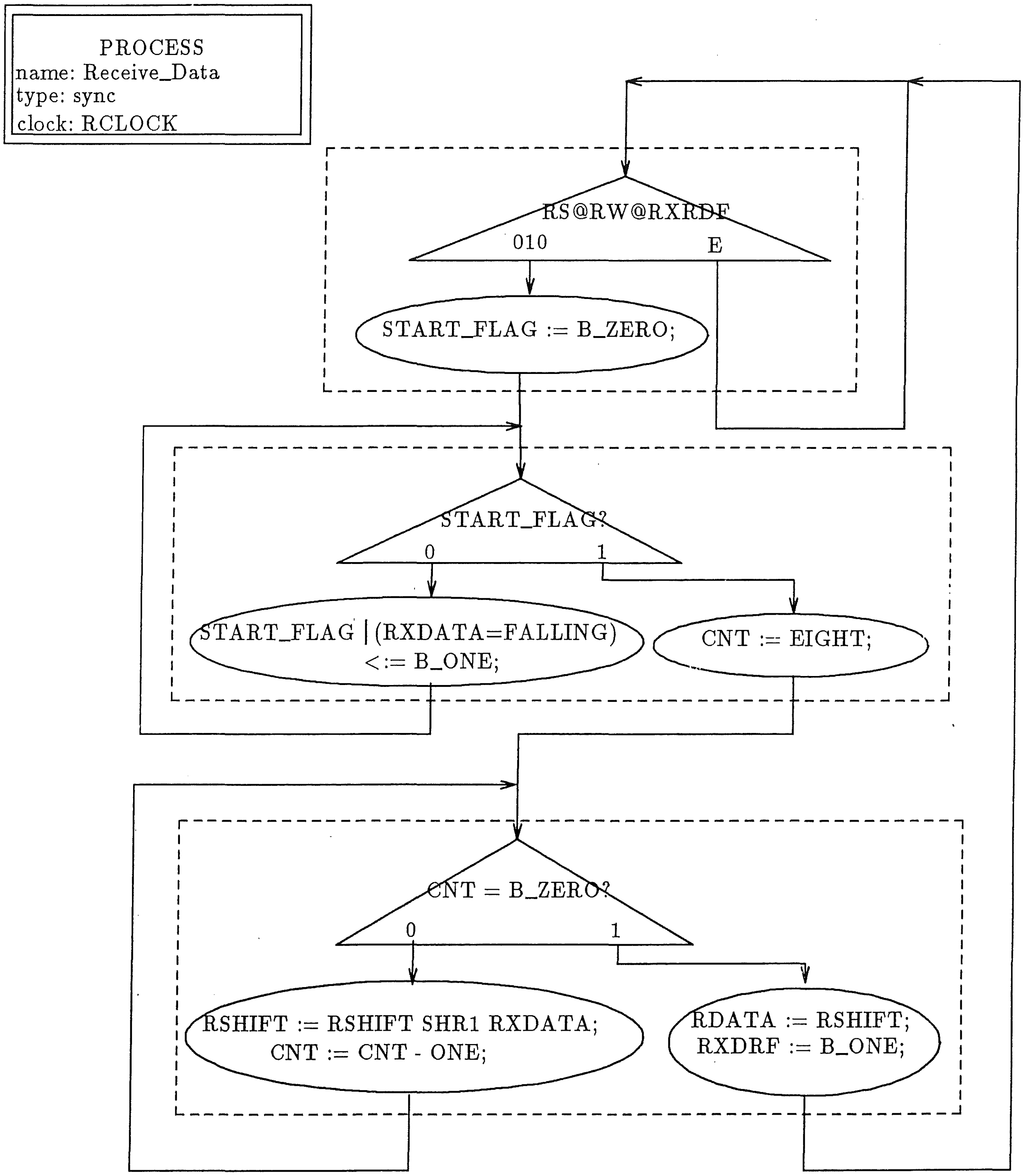


Figure 17. UART Receive_Data PROCESS

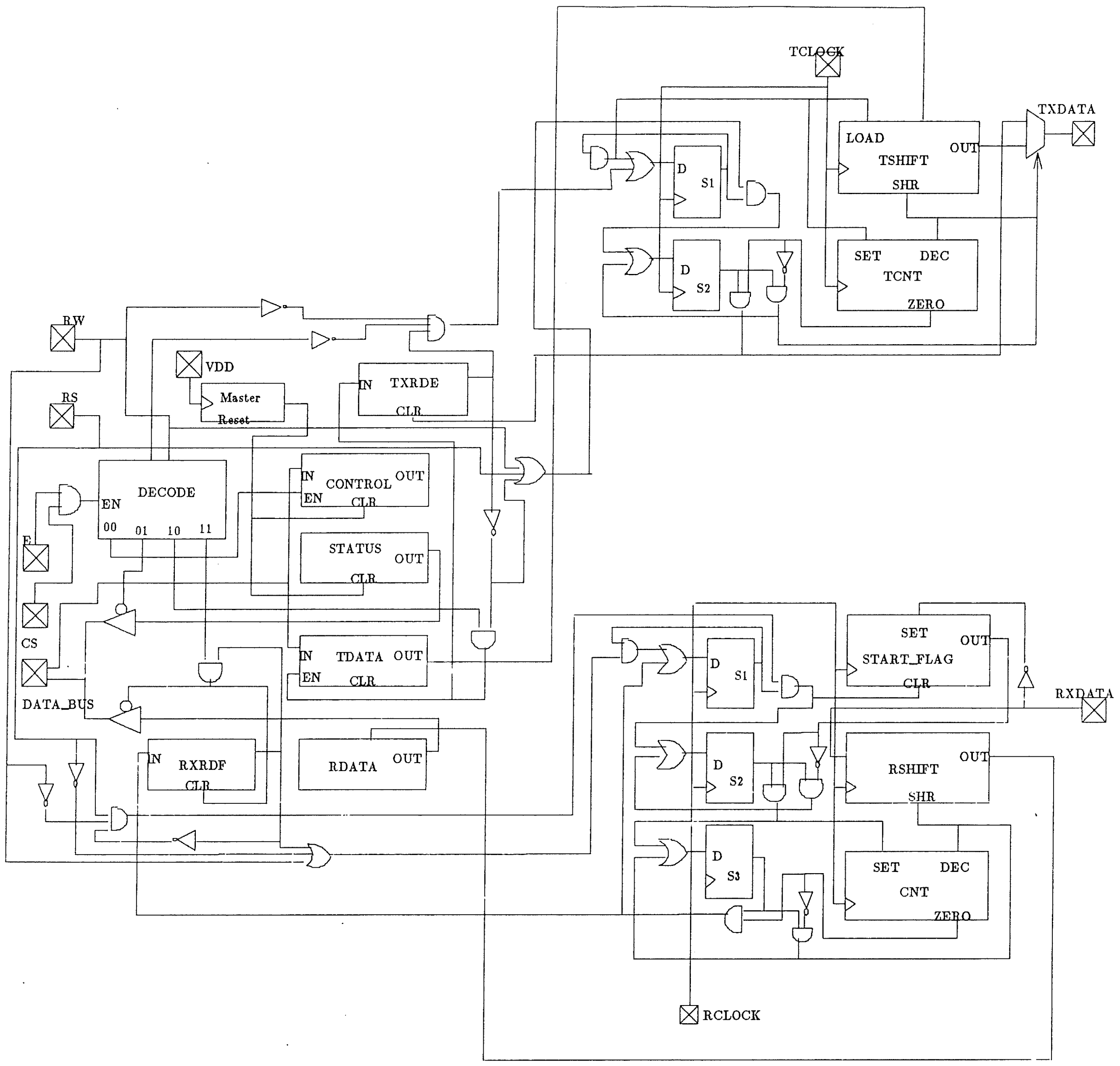


Figure 18. THE GENERATED STRUCTURE FOR THE 6850 UART

and the buses that connect TDATA with TSHIFT and RDATA with RSHIFT.

### 4.2.5. Comments

In this section, we described the use of synchronous sub-charts within an asynchronous process to capture the behavior of a 6850-like UART. Since a process, by definition, operates on a single clock, the design had to be partitioned into a a main (asynchronous) process, which calls two synchronous processes, each of which runs on a different clock.

This kind of design is hard to describe in a language like ISPS, which permits only one clocked process per description. For instance, Nestor [Nest88] has written a version of the Intel 8251 UART (similar to the 6850) which has separate processes running in parallel. The inability of ISPS to permit description of concurrent processes operating on the same set of variables and structures forces each process to be described separately with their own declarations and ports. Each process is then synthesized as a data path and controller, and they communicate through flip-flops. Hence the synthesis produces extra hardware for the interface and communication.

# CHAPTER 5.

# SUMMARY

In this document, we showed how existing input specifications are not powerful enough to capture several aspects of the design, including a combination of behavior and structure, synchronous and asynchronous functionality. We have developed a new language, EXTEND-L, which will be part of a synthesis system that will overcome many of these deficiencies. The document described the model and input language in detail. Several examples were used to illustrate the versatility of the input language in describing a variety of designs. Work on the synthesis system is in progress and forthcoming reports will describe the synthesis environment and the internal representation used in EXTEND.

## BIBLIOGRAPHY

[Arms88] J. R. Armstrong, "Chip Level Modeling with VHDL," *Prentice Hall*, 1988.

[Barb81] M. Barbacci, "Instruction Set Processor Specifications (ISPS): The Notation and its Applications" *IEEE Transactions on Computers* **30**(1)(Jan, 1981).

[BlFR85] T. Blackman, J. Fox, and C. Rosebrugh, "The SILC Silicon Compiler: Language and Features," *Proc. 22nd Design Automation Conf.*, June 1985.

[BoKa87] G. Borriello and R. H. Katz, "Synthesis and Optimization of Interface Transducer Logic," *Proc. ICCAD*, Nov. 1987.

[BrGa86] Forrest D. Brewer, Daniel D. Gajski, "An Expert System Paradigm for Design" *23rd IEEE Design Automation Conference* pp. 62-68, Las Vegas, NV (July, 1986).

[Camp85] R. Camposano, "Synthesis Techniques for Digital System Design," *Proc. 22nd Design Automation Conf.*, June, 1985.

[Clar73] C. R. Clare, "Designing Logic Systems using State Machines," *McGraw-Hill Inc.*, 1973.

[Dron83] P. J. Drongowski "A Graphical Engineering Aid for VLSI Systems," UMI Research Press, 1986.

[GaDP86] D. D. Gajski, N. D. Dutt, B. M. Pangrle, "Silicon Compilation (Tutorial)," *Custom Integrated Circuits Conference*, May, 1986.

[GiBK85] E. F. Girczyc, R. J. A. Buhr, J. P. Knight, "Applicability of a Subset of Ada as an Algorithmic Hardware Description Language for Graph-Based Hardware Compilation," *IEEE Transactions on Computer-Aided Design*, vol. CAD-4, no. 2, (April, 1985).

[JVJC86] A. A. Jerraya, P. Varniot, R. Jamier, B. Curtios, "Principles of the SYCO Compiler" *23rd Design Automation Conference* IEEE ACM, Las Vegas, NV, (July, 1986).

[Nest88] J. Nestor, "Notes on the i8251 ISPS Description," *ACM/IEEE Workshop on High-Level Synthesis*, Orcas Island, WA, Jan. 1988.

[Pang87] B. Pangrle, "A Behavioral Compiler for Intelligent Silicon Compilation" *PhD Dissertation*, *University of Illinois*, *Urbana-Champaign* (June, 1987).

[Sout83] J. R. Southard, "MacPitts: An Approach to Silicon Compilation," *IEEE Computer*, vol. 16, no. 12, (Dec, 1983).

[Tred81] N. Tredennick, "How to Flowchart for Hardware," *IEEE Computer*, December 1981.

[Tric87] Howard Trickey, "A High-Level Hardware Compiler" *IEEE TRAN. on Computer Aided Design* **CAD-6**(2), (March, 1987).

[VaGa88] N. VanderZanden and D. D. Gajski, "MILO: A Microarchitecture and Logic Optimizer," *Proc.* 25*th Design Automation Conference*, Anaheim, CA, June 1988.

[VHDL87] *VHDL Tutorial for IEEE Standard* 1076 *VHDL*, CAD Language Systems Inc., June 1987.

## APPENDIX A

### BNF syntax for definitions and statements

The syntax for the definition grammar and the statement grammar is shown separately.

### Definition Grammar

```
definition_part             :    definitions_header definition_block DOT
                            |    empty
                            ;

definitions_header          :    DEFINITIONS
                            ;

definition_block            :    TBEGIN
                                 type_definition_part
                                 component_definition_part
                                 port_definition_part
                                 clock_definition_part
                                 variable_definition_part
                                 constant_definition_part
                                 binding_definition_part
                                 TEND
                            ;

type_definition_part        :    TYPE type_definition_list
                            |    empty
                            ;

type_definition_list        :    type_definition_list type_definition
                            |    type_definition
                            ;

type_definition             :    tidentifier TEQUAL type_type semicolon
                            ;

tidentifier                 :    identifier
                            ;

type_denoter                :    identifier
                            ;

type_type                   :    bit_type
                            |    array_type
                            |    component_type
```

```
                    ;

bit_type            :     LCURL
                          subrange_type
                          RCURL
                    ;

subrange_type             :     constant
                          DOTDOT
                          constant

                    |     constant
                    ;

array_type          :     ARRAY LBRAC subrange_type RBRAC OF type_denoter
                    ;

component_type            :     generic_component_name  LPAREN parameter_list RPAREN
                    ;

generic_component_name    :     identifier
                          ;

parameter_list      :     parameter_list COMMA parameter
                    |     parameter
                    ;

parameter           :     identifier
                    |     DIGSEQ
                    |     empty
                    ;

component_definition_part :     COMPONENT component_definition_list semicolon
                          |     empty
                          ;

component_definition_list :     component_definition_list semicolon component_definition
                          |     component_definition
                          ;

component_definition :    component_id_list COLON type_denoter
                          ;

component_id_list   :     component_id_list comma identifier
                    |     identifier
                    ;
```

```
port_definition_part   :   PORT port_definition_list semicolon
                       |   empty
                       ;

port_definition_list   :   port_definition_list semicolon port_definition
                       |   port_definition
                       ;

port_definition            :       po_identifier_list
                           COLON
                           prt_mode
                           prt_gating
                           prt_storage
                           OF
                           type_denoter
                       ;

po_identifier_list     :   po_identifier_list comma identifier
                       |   identifier
                       ;

prt_mode               :   TINPUT
                       |   TOUTPUT
                       |   TINPUT_OUTPUT
                       ;

prt_gating             :   TWIRED
                       |   TTRISTATE
                       |   empty
                       ;

prt_storage            :   TUNBUFFERED
                       |   TBUFFERED
                       |   empty
                       ;

clock_definition_part  :   CLOCK clock_definition_list semicolon
                       |   empty
                       ;

clock_definition_list  :   clock_definition_list semicolon clock_definition
                       |   clock_definition
                       ;

clock_definition       :   clock_id_list COLON clock_source
                       ;
```

```
clock_id_list               :     clock_id_list comma identifier
                            |     identifier
                            ;

clock_source                :     PORT
                            |     VAR
                            ;

variable_definition_part        :     VAR variable_definition_list semicolon
                                |     empty
                                ;

variable_definition_list        :     variable_definition_list semicolon variable_definition
                                |     variable_definition
                                ;

variable_definition         :     videntifier_list COLON type_denoter
                                  ;

videntifier_list            :     videntifier_list comma identifier
                            |     identifier
                            ;

constant_definition_part        :     CONST constant_definition_list semicolon
                                |     empty
                                ;

constant_definition_list        :     constant_definition_list semicolon constant_definition
                                |     constant_definition
                                ;

constant_definition         :     cidentifier_list OF type_denoter TEQUAL cvalue
                            ;

cidentifier_list            :     cidentifier_list comma identifier
                            |     identifier
                            ;

cvalue                      :     constant
                            ;

binding_definition_part         :     BIND binding_definition_list semicolon
                            |     empty
                            ;

binding_definition_list         :     binding_definition_list semicolon binding_defintion
                            |     binding_defintion
```

```
                         ;

binding_defintion        :    var_name TO component_name
                         ;

var_name                 :    identifier
                         ;

component_name                :    identifier
                         ;
```

**Statement Syntax**

```
statement_part                :    compound_statement
                         ;

compound_statement            :    statement_sequence
                         ;

statement_sequence       :    statement_sequence semicolon statement
                         |    statement
                         ;

statement                :    assign_statement
                         |    async_assign_statement
                         |    empty
                         ;

assign_statement         :    l_h_s TASSIGN r_h_s
                         ;

async_assign_statement        :    l_h_s TASYNCASSIGN r_h_s
                         ;

r_h_s                    :    expression_part path_timing_part
                         ;

expression_part               :    expression
                         ;

path_timing_part         :    DELAY timing_list
                         |    empty
                         ;

timing_list              :    timing_list comma path_constraint
                         |    path_constraint
```

```
                            ;

path_constraint                 :     delay_range FROM timing_input
                            ;

delay_range                 :   min_delay max_delay
                            ;

min_delay                   :   MIN DIGSEQ
                            |   empty
                            ;

max_delay                   :   MAX DIGSEQ
                            |   empty
                            ;

timing_input                :   identifier
                            ;

l_h_s                       :   lhs_variable_access event_timing_part
                            ;

lhs_variable_access         :   lhs_identifier_part
                            |   lhs_indexed_variable
                            ;


lhs_identifier_part         :   identifier bit_field
                            |   identifier
                            ;

lhs_indexed_variable        :   lhs_a_identifier LBRAC lhs_index_expression RBRAC
                            ;

lhs_a_identifier            :   identifier
                            ;

lhs_index_expression        :   expression
                            ;

event_timing_part           :   TOR LPAREN event_timing LPAREN
                            |   empty
                            ;

event_timing                :   after_event
                            |   async_assign_event
                            ;
```

```
after_event              :   AFTER event_id event_cause
                         ;

event_id                 :   identifier
                         ;

event_cause              :   RISING
                         |   FALLING
                         ;

async_assign_event       :   port_name TEQUAL event_cause
                         ;

port_name                :   identifier
                         ;

expression               :   simple_expression
                         |   simple_expression or simple_expression
                         ;

simple_expression        :   logic_one
                         |   simple_expression xor logic_one
                         ;

logic_one                :   bit_one
                         |   logic_one and bit_one
                         ;

bit_one                  :   bit_two
                         |   bit_one bor bit_two
                         ;

bit_two                  :   bit_three
                         |   bit_two bxor bit_three
                         ;

bit_three                :   rel_one
                         |   bit_three band rel_one
                         ;

rel_one                  :   rel_two
                         |   rel_one relop_two rel_two
                         ;

rel_two                  :   shift_term
                         |   rel_two relop shift_term
                         ;
```

```
shift_term            :       add_term
                      |       shift_term shiftop add_term
                      ;

add_term              :       term
                      |       add_term addop term
                      ;

term                  :       factor
                      |       term mulop factor
                      ;

factor                :       sign factor
                      |       exponentiation
                      ;

exponentiation                :          base
                      |       base expop exponentiation
                      ;

base                  :       primary
                      |       base concatop primary
                      ;

primary               :       variable_access
                      |       unsigned_constant
                      |       LPAREN       expression RPAREN
                      |       not primary
                      |       complop
                              primary
                      ;

unsigned_constant     :       unsigned_number
                      ;

unsigned_number       :       unsigned_integer
                      ;

unsigned_integer      :       DIGSEQ
                      ;

expop          :      expop1 bind_op
               ;

expop1         :      TSTARSTAR
               ;
```

```
concatop        :       concatop1 bind_op
                ;

concatop1       :       TCONCAT
                ;

complop                 :       complop1 bind_op
                ;

complop1        :       TCOMPL
                ;

mulop           :       mulop1 bind_op
                ;

mulop1                  :       STAR
                |       SLASH
                ;

addop           :       addop1 bind_op
                ;

addop1                  :       TPLUS
                |       TMINUS
                ;

shiftop         :       shiftop1 bind_op
                ;

shiftop1        :       TASHL
                |       TASHR
                |       TSHL0
                |       TSHL1
                |       TSHR0
                |       TSHR1
                |       TROTL
                |       TROTR
                ;

relop           :       relop1 bind_op
                ;

relop1          :       TLT
                |       TGT
                |       TLE
                |       TGE
                ;
```

```
relop_two      :      relop_two1 bind_op
               ;

relop_two1     :      TEQUAL
               |      TNOTEQUAL
               ;

band           :      band1 bind_op
               ;

band1          :      TLAND
               |      TLNAND ;

bxor           :      bxor1 bind_op
               ;

bxor1          :      TLXOR
               |      TLXNOR
               ;

bor            :      bor1 bind_op
               ;

bor1           :      TLOR
               |      TLNOR
               ;

and            :      and1 bind_op
               ;

and1           :      TAND
               |      TNAND
               ;

xor            :      xor1 bind_op
               ;

xor1           :      TXOR
               |      TXNOR
               ;

or             :      or1 bind_op
               ;

or1            :      TOR
               |      TNOR
               ;
```

```
not                 :       not1 bind_op
                    ;

not1                :       TNOT
                    ;

bind_op             :       LCURL bound_component RCURL
                    |       empty
                    ;

bound_component     :       identifier
                    ;

variable_access     :       identifier_part
                    |       indexed_variable
                    ;

identifier_part     :       v_identifier bit_field   /* select (rhs) */
                    |       v_identifier
                    ;

v_identifier        :       identifier
                    ;

bit_field           :       LCURL subrange_type RCURL
                    ;

indexed_variable    :       a_identifier LBRAC index_expression RBRAC
                    ;

a_identifier        :       identifier
                    ;

index_expression    :       expression
                    ;

constant            :       non_string
                    |       sign non_string
                    ;

sign                :       TPLUS
                    |       TMINUS
                    ;

non_string          :       DIGSEQ
                    ;
```

```
identifier          :     IDENTIFIER
                    ;

semicolon           :     SEMICOLON
                    ;

comma                     :     COMMA
                    ;
```

# APPENDIX A

## GENERIC COMPONENT LIBRARY ELEMENTS

A brief description of the generic component library elements that are used in the synthesis task is given here. Abstract behavioral elements such as operations and variables in the language are mapped into instantiations of components drawn from this library.

### Storage Components

Various types of storage components may be declared by the user: registers, shift registers, counters, memories, register files, stacks and FIFO's. Storage elements are characterized by their type, the functions they perform, data inputs and outputs, control inputs, clock inputs, asynchronous inputs, and their attributes (#bits, size, etc.). Figure 19 shows these characteristics for storage components that are currently supported.

### Bus Components

Each bus component is declared with the following attributes:

(1) bit width

| TABLE Storage.tbl | | | | | |
|---|---|---|---|---|---|
| **Type** | **Functions** | **Data-i/o** | **control** | **async** | **attributes** |
| Register | load, clear | 1-inp, 1-outp | load, enable | set, clear | #bits, delay, power |
| Shift Register | load(par), clear,shr,shl | par-i/o, l/r-i/o, | load, shl, shr,enable | clear | #bits, delay, power |
| Counter | load(par), up, down, set, clear | 1-input, 1-output | enable, up, down | set, clear | #bits, delay, power |
| Memory | read, write | 1-input 1-output | enable,read, write,address | | size, delay, #bits, power |
| Register File | read, write | #i ports, #o ports, #i/o ports | enable, read,write & addr for each port | | size, delay, #bits, power |
| Stack or FIFO | push, pop | 1-input, 1-output | enable, push, pop | | size, delay, #bits, power |

Figure 19. GENERIC STORAGE COMPONENTS

(2) transfer mode = (unidirectional/bidirectional)

(3) storage = (buffered/unbuffered)

(4) gating = (tristate/wired).

**Functional Components**

Figure 20 lists some of the more commonly used functional components. In addition to the generic structural modules that are read into the library, specific function units may be declared by the user. This allows for an extensible library of structural modules, where declared function units are added to the library as new modules and are used later. The declaration should specify the number of data, control, clock and asynchronous inputs, the number of data outputs, and for each operation, the control code and function mapping the inputs to the outputs. The bit width and the representation of each input and output must also be specified.

**TABLE FUs.tbl**

| **Type** | **Functions** | **Data I/O** | **Control** | **Attributes** |
|---|---|---|---|---|
| ALU | ADD, SUB, AND, OR EQV, INC, ZRO, NOP | 2-inputs 1-output | 8 bits | #bits, delay, power |
| ADDER | ADD | 2-inputs 1-output | | #bits, delay, power |
| Comparator | GT, LT, EQ | 2-inputs | 3 bits | #bits, delay, power |
| Shifter | SHL, SHR, ROTL, ROTR | 1-input 1-output | sel, sh, rot l/r, a/l, fill | #bits, delay, power |
| Multiplier | MULT | 2-inputs 1-output | enable | #bits, delay, power |

Figure 20. GENERIC FUNCTIONAL COMPONENTS