PICSIL:

Design and Synthesis of Digital ICs
from
Data Flow Diagrams

A dissertation presented
in partial fulfillment of the requirements
for the degree of
Doctor of Philosphy in Computer Science
at Massey University

Murray William Pearson

1992

Abstract

This thesis describes the background, development, and testing of PICSIL, a system for designing digital integrated circuits by structured decomposition.

PICSIL draws upon graphical and textual specification techniques; the first for high-level, architectural, system components, the second for more detailed, functional, specification.

Many graphical design paradigms already exist. Of these, DFDs (Data Flow Diagrams) best suit the assembly and intercommunication of abstract modules. With minor adaptations, DFDs were used as the high-level specification language. Lower-level functionality was described using a textual language based on HardwareC. Although HardwareC is not ideally suited to this use, and had to be extended in several areas it was adopted mainly for pragmatic reasons.

To accept a system defininition and subsequently determine the details of its synthesis, the PICSIL system had not only to capture and edit high-level specifications, but also to deliver these specifications to one of several possible synthesis paths. The practical part of the thesis therefore consisted of implementing a graphical editor and a synthesis compiler. These drive the lower level Olympus and Octtools synthesis packages to provide a complete path from PICSIL input to chip layout. A layout produced by following this path has been sent for fabrication.

Acknowledgements

I would like to thank Professor Mark Apperley and Paul Lyons for their guidance during this research. The encouragement to make the "supreme effort" necessary to complete this thesis is now greatly appreciated. Particular thanks must go to Paul Lyons for the suggestions made during the production of this thesis.

Thanks must also go to the members of the Computer Science department for their constant encouragement, and tolerating the lack of disk space in the later stages of this research. In particular thanks must go to Colin Eagle for installation and maintenance of software, and Peter Kay for proof reading the last version of the thesis.

Many thanks to Bruce Moore, and various people on internet, including Paul Cohen and Rajesh Gupta, for their technical expertise during the development of the PICSIL synthesis system.

I am also very grateful to both my parents and parents-inlaw for their encouragement during this time. Finally, I would like to thank my wife, Brenda, for her patience and support (despite an expanding deadline) which has been necessary for me to complete this thesis.

Contents

Chapter 1
Graphics and the Design of Silicon Architectures

Since integrated circuits were originally invented thirty years ago, improvements in fabrication technology have enabled the manufacture of vastly more complex chips. Circuits containing up to 100 million transistors are now being proposed (Cavin and Hilbert,1990), but the achieved rate of increase in circuit density - historically, a doubling every two years - has slowed in recent years (Dillinger, 1988).

This slowing rate of increase in complexity particularly applies to logic-intensive functions such as microprocessors and microcontrollers. Burger and Holton (1992) suggest that while the circuit density of memory chips has been increasing 60 percent a year, the circuit density of random logic chips is achieving an annual increase of only a 25 percent. Furthermore, while designer productivity has improved significantly in recent years, an additional 30-fold increase will be required in the next decade. The most significant reason for this difficulty is the complexity of the designs (Sequin, 1983).

What causes this difficulty with design? Are designs innately and unmanageably complicated and difficult to conceptualise?

If they were, then the design process could be expected to remain a bottleneck, hindering innovation forever. However, even casual experience shows us that, *in concept*, many "complex" designs are relatively simple. It is the unavailability of design tools at this abstract conceptual level, which forces us to muddy our abstractions with low-level implementation considerations, and causes difficulty.

An equivalent complexity problem was also encountered in the software industry when computer programs grew to lengths in excess of 10,000 lines of code (Sequin, 1983). This led to developments on two main fronts. The first of these was a development of high level programming languages and compilers. With these tools, programmers could program at higher levels of abstraction and have the program compiled automatically to a machine level representation.

The second main area of development was in the area of design methodologies, including top-down design (Sommerville, 1989), Structured Analysis (DeMarco, 1978), structure charts (Jackson, 1975) and structure diagrams (Doran and Tate, 1972). Most of these techniques use a

mixture of graphics and text. Graphics are particularly well suited to representing an overview of a complex system while the text-based notation has the complementary ability to represent the more detailed aspects. (Tse and Pong, 1991).

Until recently, design methodologies have achieved only partial success in the field of programming, as they have been a paper-based documentation tool and have proved difficult to integrate into the design expression. This has changed within the last seven years, however, as sufficient computing power and symbolic manipulation ability have become universally available in low cost work-stations with graphics capabilities. This has led to the development of a number of Computer Aided Software Engineering (CASE) tools based on formalised versions of these design methodologies, which support a system through design, implementation, documentation and maintenance.

IC designers also responded to the need to manage complexity in the early 1980s, starting with the introduction of the structured design methodology by Mead and Conway (1980). Since this time, a large number of CAD tools have been developed which support IC design at higher levels of abstraction, closer to a designer's conceptual model. Most of these CAD tools fall into two main groups: those that support design at a particular level of abstraction (e.g. simulators) and those that provide synthesis to lower levels of abstraction.

The principle input to these CAD tools is a representation of the IC device under design, using a hardware description language (HDL). The HDLs generally allow an IC to be described either structurally as an interconnected set of components, or behaviourally as a mapping of inputs onto outputs, or as a mixture of both. The levels of abstraction that these HDLs support vary widely, from the logic level, where a system is described as a set of logic gates, to high-level behavioural languages, where the behaviour of a system is described using constructs similar to those found in programming languages.

To reduce design complexity and the comprehension of the resulting design, HDLs which operate at the logic level commonly use graphics, to provide schematic capture. However almost all HDLs which work at the higher levels of abstraction rely solely on text-based descriptions.

To date, no equivalent to the software oriented CASE tools has been developed to support the design, documentation and maintenance of hardware. Text based HDLs are the highest level representation used to support these roles.

1.1 Context of the research

Motivation

The search for a powerful abstraction suitable for use in capturing digital IC designs has been the general motivation for the research described herein.

Thesis

A design environment should maximise the opportunity for creative input at the highest levels of abstraction, and minimise the need for input of automatically-derivable lower-level information. An ideal digital IC design environment would therefore:

- harness a designer's creativity effectively by concentrating on the system level of abstraction;

- recognise the natural vocabulary for system-level ideas by allowing direct graphical input;
- conserve the designers effort for creative work by automating the mechanical chore of layout synthesis.

Project specification

The highest level concepts used by most designers - both in producing designs and in documenting them - relate to the assembly and intercommunication of processing elements. In this thesis, the term *system* will be used to refer to designs at this level of abstraction. It is universal practice to develop systems from initial, diagrammatic representations of building blocks at this system level.

Earlier environments for design capture have been constrained by the historical impracticality of implementing design-capture at the diagrammatic stage. Now, however, three prerequisites are satisfied:

- graphics processing is widely available and cheap;
- guidelines have been established for ergonomic editing of graphical data;
- systems to assist in the building of graphics applications are available.

Accordingly, the specific goal of this research has been to apply graphical editing and capture techniques to the design of VLSI.

Sequin (1983) discusses a set of generalised requirements (based on those used in software engineering) necessary to manage complexity in VLSI design. These requirements can be used as a set of guidelines in developing an HDL:

- *Design Equals Documentation*: If the documentation necessary to understand a design forms part of the design description, then both remain consistent.
- *Use of Abstraction*: The use of abstraction can make a design representation more succinct, providing a large step toward managing complexity.
- *High level descriptions*: The higher the level of description, the shorter and more readable it will be, leading to designs with fewer errors.
- *Partitioning*: The partitioning of a design into smaller more manageable components reduces the level of complexity under consideration at any one time.
- *Structuring methods*: Choosing an appropriate structuring method to achieve partitioning is important.
- *Restriction to a limited number of constructs*: The understandabity of a design can be greatly improved if the number of available constructs is restricted.
- *Testing*: Testing is an integral part of the design process and should be kept in mind at all stages.
- *Tools and Design Methodologies*: Of crucial importance in the design of complex systems is a good set of tools and a suitable design method.

Project History

In this description, the progress of the research work has been divided into discrete phases, There has been a good deal of overlap between these phases, but is largely ignored, for the purpose of achieving a coherent portrayal.

Because HDLs are the current primary tool used by designers to perform design, a review of a number of HDLs was conducted. As this thesis is mainly concerned with designing at high levels of abstraction its main emphasis is on the constructs which support this area. This review was used to determine any strengths and weaknesses of current HDLs that could influence the development of a new design methodology.

As mentioned above, design methodologies already exist for managing software engineering complexity in software design. A study was conducted in which two of these methodologies (Structured Analysis and Structure Diagrams) were applied to design exercises to determine whether they could be used as the basis for an initial representation of a hardware design.

The Structured Analysis (SA) methodology, with a number of refinements, was identified as a suitable framework for a hardware design environment. The primary tool of SA is the data flow diagram (DFD), a simple graphical language for describing an application as a modular structure, in terms of the data flowing through it.

To refine the SA techniques into an appropriate language for hardware description, an iterative process of adding new constructs and evaluating them against the design exercises was pursued. This led to the development of the PICSIL (for PICtorial SILicon language) graphical and textual notation for representing complex designs digital logic.

If the PICSIL notation is to succeed in simplifying hardware design, then it needs to be integrated into the formal design process and not be only paper-based documentation. This consideration has led to the development of the two tools which comprise the PICSIL environment; an editor to support the capture of a PICSIL design, and a synthesis manager to drive a lower level synthesis path based on the captured design. Optimisation of these tools, whose behaviour necessarily depends upon the exact nature of the PICSIL notation, was given a lower priority than the notation. Consequently their current implementation could be characterised as functional rather than polished. Nevertheless, they are sufficiently well developed to have allowed the verification of the complete synthesis path, from initial capture of a design expressed in the PICSIL notation to automatic generation of the corresponding chip layout.

Although design testing is considered important to the production of functional devices, it was given still lower priority in order to keep the project to a manageable size. The incorporation of simulation tools to allow testing is discussed as a topic of future research in Chapter 8. Ten design exercises have been successfully undertaken using the editor, and three of these were chosen for synthesis, and successfully synthesised to a chip layout. A 2500 transistor design has been sent for fabrication.

Chapter 2
Hardware Description Languages for VLSI Design

Because of the complexity of VLSI design, the design process is usually divided into a number of stages representing the design at varying levels of abstraction. These levels of abstraction can typically be divided into system, algorithmic, register transfer, logic and circuit (Siewiorkek, Bell and Newell, 1982 and McFarland, Parker and Camposano, 1990).

The system level involves the initial partitioning of a design into high level functional units or modules and determines the relationships between them. For example the components of a computer system at this level are items such as processors, memories and controllers.

At the algorithmic level, the focus is on the processing performed by the system level modules, and the way these modules map sequences of inputs onto sequences of outputs.

Below the algorithmic level is the register transfer level where the design is viewed as a set of interconnected storage elements and low level functional blocks (e.g. ALUs and adders), with behaviour which is described as a series of data transforms using the low level functional components between the storage elements.

At the logic level the design is specified as a network of gates and flip-flops and the behaviour is specified by logic equations. Below that is the circuit level which views the system in terms of the geometry of which it is composed.

At each of the five levels of abstraction a circuit can also be represented in any of three domains: behavioural, structural and physical. Figure 2.1 shows a Y chart which is commonly used to show the varying levels of abstraction in each of the domains. The centre of the Y represents a design layout. The further a node is from the centre, the higher the level of design abstraction it represents.

In the behavioural domain some form of *input to output* mapping is used to describe what the design is to do. Details about the structure or implementation of the design are not present. In the structural domain, a design is specified in terms of its components and their interconnections. Information such as the location and size of the component on the chip is not included. The geometric domain binds the structures of a system to its physical layout space.

The structural domain can be seen as a bridge between the behavioural and geometric domains (Dutt and Gajski, 1990).

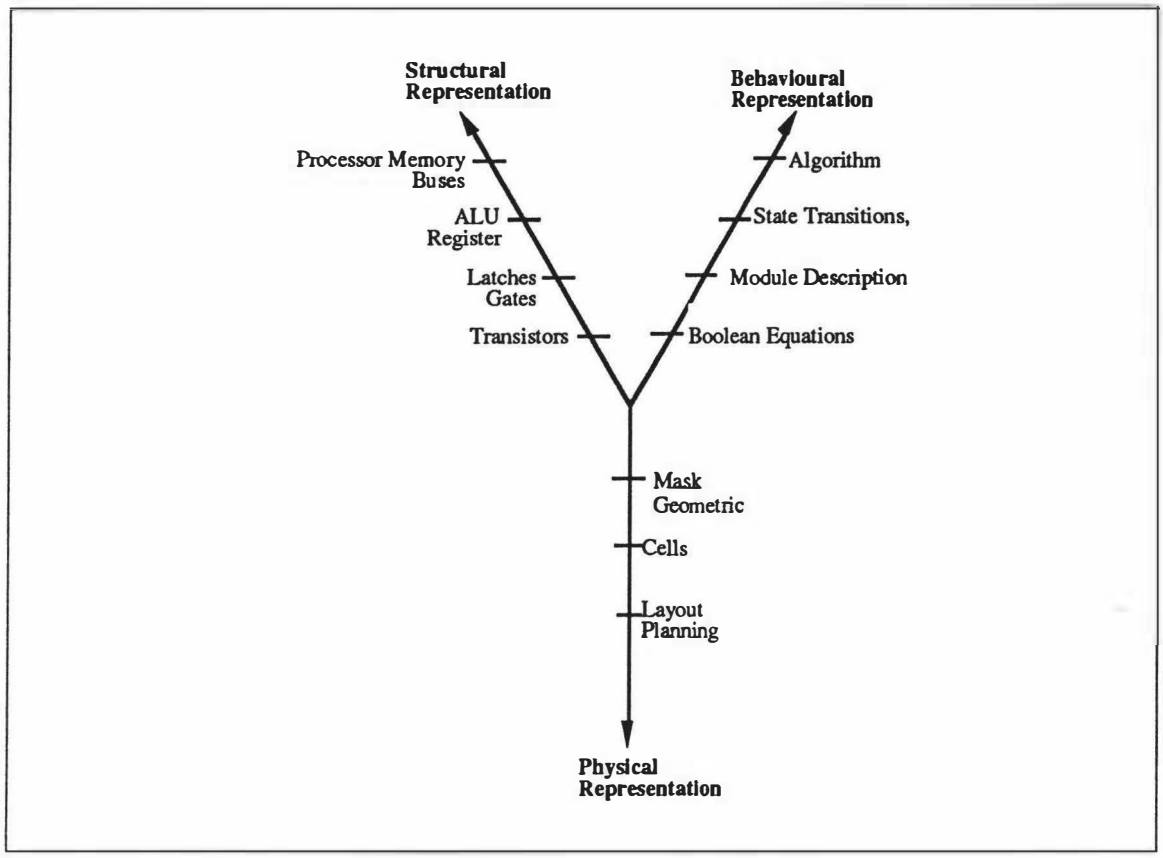


Figure 2.1 Y-chart showing the different levels of abstraction in each of the domains that can be used to describe a design. (after Gajski and Kuhn, 1983)

Converting a design to silicon requires that a path from the initial, highest level abstraction to the centre of the Y be traced. This involves augmenting the initial design with a huge amount of additional data. A mistake at any stage can cause an expensive loss of time and resources. To help manage design complexity there is an increasing desire to partition design work among several parties and reuse parts of previously verified designs (Yeng and Rees, 1991).

To formalise the data interchange required by this partitioning, HDLs (hardware description languages) have been developed. These HDLs also form the principle input to CAD tools which help to support design at varying levels of abstraction.

Some of the early HDLs were highly specialised, such as MacPitts (Southard, 1983) and FIRST (Denyer and Renshaw, 1985) and were oriented towards an implementation using a particular architecture. They allowed automatic compilation of efficient designs in very narrow application domains.

More recently there has been a trend to develop general purpose HDLs such as VHDL (Shahdad, Lipsett, Marschner, Sheehan, Cohen, Waxman and Ackley, 1985) and Verilog (Thomas and Moorby, 1991) which enable description of most types of hardware over a wide range of levels of abstraction. This allows a single HDL to support a design through most of the implementation process. As the implementation progresses, the designer successively replaces abstract descriptions by more concrete descriptions. These languages tend to contain constructs which support efficient simulation allowing the design to be verified before design proceeds to a lower level of abstraction.

To support design over a wide range of levels of abstraction these languages require large numbers of constructs making them very complex. However, they are intended more for design verification by simulation than for eventual translation into silicon, and implementations stop somewhat short of producing mask layouts.

Lately, dramatic development has been made in the area of *synthesis* (or *compilation*) tools which automatically compile a design to a lower level of abstraction. These automatic compilation tools allow designers to work at higher levels of abstraction without having to consider the finer details of IC design and processing technology. Hence they can devote more time to the creative aspects of the design at hand and explore different design alternatives. Another major advantage of this group of CAD tools is that they provide a *"correct by construction"* (Gajski, 1988) capability. That is, if the high level description is correct, then the layout generated will also be correct. Any design errors introduced by a designer are at a higher level and are therefore easier to detect and correct. Because designers work at a high level of abstraction, and they can produce correct designs first time, an improved competitiveness in the market place can be expected.

However automatic compilation tools are not without disadvantages. Generally designs produced by automatic compilation are larger and/or slower that the equivalent designs produced by a skilled human designer. Another disadvantage is that these tools are hard to integrate into current *well established* CAD environments.

However despite the problems of silicon compilation, it is expected that the complexity and performance of VLSI designs will continue to increase over the next decade (Cavin and Hilbert, 1990) to devices that have tens to hundreds of millions of transistors. Also, the competitive pressure to design, produce and market new VLSI products quickly will necessitate automatic compilation (silicon compilation) tools.

Currently silicon compilation tools accepting designs at the register transfer level are in common use (Walker and Camposano, 1991). Research into synthesis from higher levels of abstraction (high level synthesis) has been an active area of research for the last twenty years and a number of such systems exist, but they are not yet common in the market place. However this seems likely to change in the near future (De Micheli, 1990).

A number of HDLs such as HardwareC (Ku and De Micheli, 1990) have started to evolve which are oriented towards input into high level synthesis systems. As these HDLs only need to support the high levels of abstraction, fewer constructs are required, making the HDLs simpler. This allows HDLs to support the design process better and takes a step towards meeting Sequin's requirements (see page 3) for managing design complexity.

In the following sections, three hardware description languages will be presented in detail. The hardware description languages selected are either widely used in industry (VHDL and Verilog) or particularly suited to input to high level synthesis (HardwareC). Although VHDL and Verilog support description over a wide range of levels of abstraction, only the constructs which are oriented towards the higher levels of abstraction and which can be synthesised are presented.

To help consider the features/limitations of each of the HDLs, a common design example is represented in each of them. The device to be designed is the i8251 chip, a programmable communications interface (Intel, 1983).

The i8251 is used as a peripheral device and is programmed by the CPU to employ virtually any of the serial data transmission techniques which were in use at the time it was developed. The chip accepts data characters from the CPU in parallel format and converts them into a continuous serial data stream for transmission. Simultaneously, it can receive serial data

streams and convert them into parallel data characters for the CPU. The chip will signal to the CPU whenever it can accept a new character for transmission, or whenever it has received a character for the CPU. The CPU can read the complete status of the chip at any time.

The functionality of the chip can be broken into three main tasks, a bus interface, a serial receiver and a serial transmitter. The bus interface handles all communications between the CPU and the chip. It also communicates information to and from the serial receiver and the serial transmitter.

When the chip is first initialised it waits for a mode byte to be written to it. If synchronous operation is specified by the mode byte it reads one or two sync characters which it makes available to the serial receiver and transmitter. It continuously waits for a chip select signal and either a read or write command signal over the CPU interface. A read operation outputs data from the serial receiver or status information dependent on a chip data signal line. A write operation writes data to the serial transmitter or a control byte port also dependent on the chip data signal line.

2.1 VHDL

VHDL - VHSIC (Very High Speed Integrated Circuit) HDL was developed jointly by industry and the United States of America Department of Defence (DoD) (Shahdad, Lipsett, Marschner, Sheehan, Cohen, Waxman and Ackley, 1985, Ashenden, 1990). The language underwent many reviews to ensure any features that might possibly be needed were included in the language, and in 1987 it was adopted as IEEE Standard, 1076-1987 (cited in Yeng and Rees, 1991)

VHDL has been designed to support the design, documentation and simulation of digital hardware, and supports description from the system level down to the logic level. More recently it has been used as input to synthesis systems. However its complexity, the lack of a full (formal) definition of the semantics, and the impossibility of efficiently synthesising several constructs in the language hinder its use for synthesis (Walker and Camposano, 1991). This has led to a number of synthesis subsets of VHDL, which defeat the idea of a standard language.

Using VHDL a system is described as a hierarchical collection of modules or *design entities*, which form the languages primary abstraction mechanism. Each of these design entities is composed of an *interface* plus one or more *design bodies*.

The main purpose of a design entities interface is to define its ports (I/Os) which constitute its interface to the outside world. A design body (or architectural body) is used to describe an implementation of an entity's structure or behaviour. As the design of a system proceeds a design body is successively replaced by lower level descriptions, closer to the system's eventual implementation.

Figure 2.2 contains the interface of the i8251 example outlined earlier in this chapter, and declares each of the I/Os of the design entity as a list of named ports. Each of the I/O declarations (e.g. `data : inout word_size;`) defines the type and direction (i.e. `in`, `out` or `inout`) of the port. The different data types available in VHDL will be discussed later in this section.

The second part of a design entity is the design body, which describes an implementation of the entity, either structurally or behaviourally. In the description of a device's structure, component parts must be declared and instantiated within a (textual) architecture declaration. Figure 2.4 shows such an architectural body description for the i8251 interface chip, derived from the structure shown in the block diagram in Figure 2.3. The three main components of the i8251 are

represented by the modules bus_interface, serial_receiver and serial_transmitter. A fourth component maintain_status is used to synchronise the transfer of data between the other three modules.

```
entity i8251 is
        port(
            chip_data : in bit;
            RE : in bit;
            WE : in bit;
            data : inout word_size;
            CS : in bit;
            xdrdy : in bit;
            txd : out bit;
            rxd : in bit;
            drdy : in bit;
            syn_det : inout bit;
            rx_rdy : out bit;
            tx_rdy : out bit;
            tx_empty : out bit;
        );
end i8251;
```

Figure 2.2 VHDL interface for i8251 example

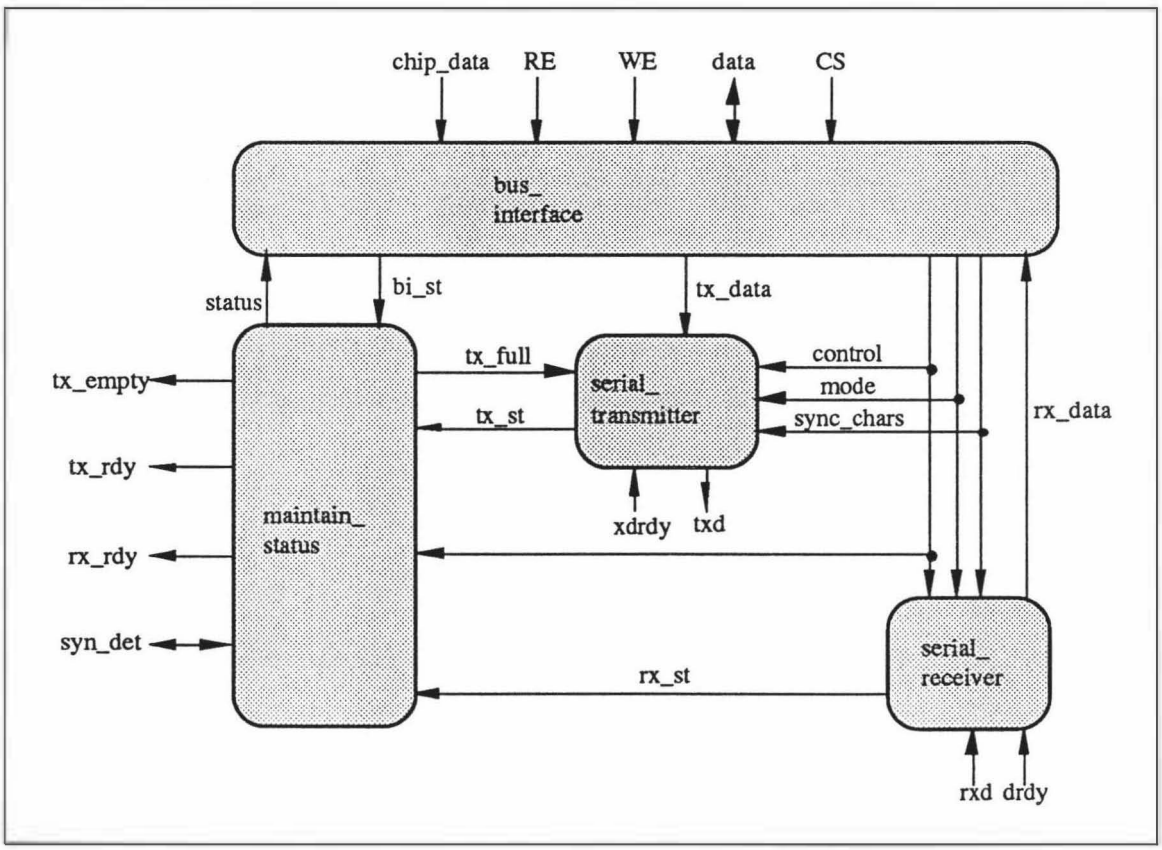


Figure 2.3 Block diagram for VHDL description of i8251 example

```
architecture system_level of i8251 is
   component bus_interface
      port(
           chip_data, RE, WE, CS : in Bit;
           status, rx_data : in word_size;
           tx_data, mode, control : out word_size;
           sync_chars : out two_chars;
           data : inout word_size;
           bi_st : out Bi_status_bits;);
   component serial_transmitter
      port(
           tx_full, xdrdy : in Bit;
           mode, control, tx_data: in word_size;
           sync_chars: in two_chars;
           txd : out Bit;
           tx_st : out Tx_status_bits;);
   component serial_receiver
      port(
           drdy, rxd  : in Bit;
           mode, control: in word_size;
           sync_chars: in two_chars;
           rx_data : out word_size;
           rx_st : out Rx_status_bits;);
   component maintain_status
      port(
           bi_st : in Bi_status_bits;
           tx_st : in Tx_status_bits;
           rx_st : in Rx_status_bits;
           status : out word_size;
           tx_empty, tx_rdy, rx_rdy, tx_full : out bit;
           syn_det : inout bit;
           control : in word_size;);
signal status, tx_data, rx_data, mode, control : word_size;
signal sync_chars : two_chars;
signal bi_st : Bi_status_bits;
signal rx_st : Rx_status_bits;
signal tx_st : Tx_status_bits;
signal tx_full : Bit;

begin
   BI : bus_interface port map (chip_data, RE, WE,CS, status, rx_data, tx_data, mode, control ,
                sync_chars , data, bi_st);
   ST : serial_transmitter  port map(tx_full, xdrdy, mode, control, tx_data, sync_chars, txd ,  tx_st);
   SR : serial_receiver port map (drdy, rxd, mode, control, sync_chars, rx_data, rx_st );
   MS : maintain_status port map(bi_st, tx_st, rx_st, status, tx_empty, tx_rdy, rx_rdy ,
                tx_full, syn_det, control);
end system_level;
```

(a) (b) (c)

Figure 2.4 VHDL top level Architectural description for i8251

Figure 2.4(a) shows the declarations of each of the components used within the structural definition of the i8251. Each component declaration should match its entity declaration defined elsewhere in the VHDL description. The main body of the definition (see Figure 2.4(c) contains the instantiation of each of the components used in the design. For example SR : serial_receiver port map (drdy, ..., rx_st); declares an instance of the component serial_receiver called SR. Each of the parameters (names enclosed in parentheses) in the instantiation defines which wires map onto the components instance. Each wire used for the connection must be either an I/O declared in the entities (i.e. i8251) interface or a locally declared signal as in Figure 2.4(b).

VHDL is the only language reviewed in this chapter which requires the declaration of components in a structural definition. An advantage of requiring component declarations in the structural body is that it is possible to determine the direction of data transfers between the

components. For example it is possible to see the direction of data transfer on rx_data is from the component serial_receiver to the component bus_interface.

Of the three languages reviewed in detail in this chapter, VHDL provides the most comprehensive set of data types. It provides two categories of types: scalar and composite. Scalar types include numeric types (integer and floating point), enumerated types (lists of values) and physical types such as time, length and voltage. The only example of a scalar type in Figure 2.2 is the predefined enumeration type bit which would be declared as:

```
type bit is ('0','1');
```

The composite types are array types and record types which are built up of elements that are any of scalar types or other composite types. Array types are built up of elements that are all the same. An example of an array type in Figure 2.2 is word_size which is declared as:

```
type word_size is array (0 to 7) of bit
```

Arrays can be extended to define memories and more complex structures. Record types are made up of elements that can be of different types. An example of a record type which can store the two sync characters in the i8251 example is:

```
type two_chars is
    record
        first : word_size;
        second : word_size;
    end record;
```

If an object sync_chars had been declared of type two_chars then the first element can be referenced using sync_chars.first. The use of record types provides a data abstraction mechanism, which can aid in the management design complexity. For example in the top level structural description, (see Figure 2.4) the two sync characters can be shown a single signal (i.e. syn_chars). The resulting description is less cluttered than if signals were declared for both characters, yet in lower level descriptions, a simple mechanism allows the individual characters to be referenced explicitly (i.e. sync_chars.first and sync_chars.second).

Because VLSI designs frequently contain regular structures (patterns repeated many times), VHDL includes the *generate* statement to facilitate their description. This statement essentially provides a macro capability within an architectural body. For example consider a four bit adder made up of four full adders. Figure 2.5 shows how the generate statement is used to create the four instances of the full adder. The *if* statement within the generate statement is used to describe an irregularity which occurs with the least significant bit.

```
architectural body PURE_STRUCTURE of FOUR_BIT_ADDER is
    signal c : bit_vector(3:0);
    component FULL_ADDER(Cin, I1, I2, in bit; Cout, RES, out bit);
begin
    for i = 3..0 generate
        if i = 0 generate     -- connect first stage
            FULL_ADDER(Cin, A(i), B(i), C(i), SUM(i));
        end generate;
        if i > 0 generate     -- connect other stages
            FULL_ADDER(C(i - 1), A(i), B(i), C(i), SUM(i));
        end generate;
    end generate;
    Cout <= C(3);
end PURE_STRUCTURE;
```

Figure 2.5 A VHDL architectural body of a four bit adder (Shahdad *et. al.*, 1985)

As well as being able to describe structure of a design entity in a design body, *process* statements may be used to describe behaviour. The constructs for describing behaviour in a process statement are similar to those found in conventional programming languages, with statements for assignment, iteration and selection. All statements within a process statement execute in sequence. Concurrency can be obtained by including multiple process statements, each executing in parallel with, and independently of, the other process statements.

Figure 2.6 shows an interface declaration and a behavioural body for the definition of the component bus_interface. One of the first noticeable differences from a conventional programming language is the use of the *signal assignment* operator (<=). A signal assignment allows a new value to be assigned to a port. For example mode <= temp assigns the value of the variable temp at the time the statement is executed to the output port mode, overwriting any existing value output on the port. When the name of an input port appears as part of an expression, the value used to evaluate the port is the value being input by the port at the time.

```
--** bus_interface interface **--
entity bus_interface is
    port(
            chip_data, RE, WE,CS : in Bit;
            status, rx_data : in word_size;
            tx_data, mode, control : out word_size;
            sync_chars : out two_chars;
            data : inout word_size bus;
            bi_st : out Bi_status_bits;
        );
end bus_interface;


--** bus interface body **--
architecture behaviour of bus_interface is

begin
  main : process

  variable temp : word_size;
  begin
    -- intialise signals
    bi_st <= '0';
    tx_data <= '0';
    control <= '0';

    wait until  (CS = '0' and WE = '0' and
                          chip_data = '1');
  -- receive mode word --
    temp := data;
    mode <= temp;
    if not(temp(0) = 0 and (temp(1) = '0') then
      -- sync mode --
wait until (CS = '0' and WE = '0' and
                          chip_data = '1');
    -- get first sync char --
    syn_chars.first <= data;
    if (temp(7) = '0')
      wait until (CS = '0' and WE = '0' and
                          chip_data = '1' );
      --get second sync char if necessary --
      syn_chars.second <= data;
    end if;
  end if;

  loop
    case (CS & WE & RE & chip_data)
      when B"0100"
        -- read data --
        data <= rx_data;
        bi_st <= B"01"
        bi_st <= B"00" after 1 ns
      when B"0101"
        -- read status --
        data <= status;
      when B"0010"
        -- write data --
        tx_data <= data;
        bi_st <= B"10"
        bi_st <= B"00" after 1 ns
      when B"0011"
        -- write control --
        temp := data;
        control <= temp;
    end case;
    wait CS = '1';
    data <= null; -- disconnect data bus
  end loop
end process
end
```

Figure 2.6 Interface and behavioural body for interface_bus entity in i8251 example

As multiple process statements, which may be contained in one or more design bodies, are executed independently of each other, synchronisation is required if processes transfer data between each other. For example when new bi_st values are transferred between the components bus_interface and maintain_status, both components must be ready for the transfer to take place. VHDL does not support constructs which allow data to be passed between entities using predefined communications protocols, requiring designers to implement their own. In this example it has been assumed that the maintain_status entity will continuously wait for new values on the 2-bit bi_st signal, and a value of 0 indicates the idle state. When a value is to be transferred, it is output on the bi_st port (e.g. bi_st <= "B01";). One clock cycle later the port's value

is returned back to the idle state (e.g. bi_st <= B"00" after 1 ns;). If it could not be assumed that the maintain_status entity was continuously waiting for new bi_st values, a more complex communications protocol involving extra signals to synchronise the transfer would be required.

To aid the designer in specifying the necessary synchronisation to transfer data between entities VHDL provides a *wait* statement (e.g. wait until (CS = '0' and WE = '0' and chip_data = '1');). When this statement is executed the process's execution is suspended until the condition becomes true.

Architectural bodies allow a design to be partitioned structurally. Sometimes it is necessary to be able to partition the behavioural description into more manageable components. Just as conventional programming languages provide functions and procedures to provide this type of partitioning, so does VHDL.

To illustrate the use of a procedure, the i8251 example will be expanded slightly. When the i8251 has been programmed to function in synchronous mode, the receiver must synchronise itself with the incoming bit stream on a command from the CPU. On receiving the command each received bit is loaded into a buffer until a match occurs with the first sync character received when the device was initialised. If the i8251 has been programmed for two sync chars the subsequent character is also compared. When both sync characters have been detected then the i8251 ends hunt mode and is in character synchronisation.

Figure 2.7 shows a skeleton definition of a procedure called hunt_mode which is called from within a behavioural description of the serial_receiver body. When the procedure call is executed the process suspends execution until the procedure finishes (i.e. when the necessary sync characters have been detected).

```
architecture behaviour of serial _receiver is
--port declarations

    procedure hunt_mode(
              rdx: in bit;
              drdy : in bit;
              sync1, sync2 : in word_size;
              mode : in word_size) is

       variable done : bit;
       variable data : word_size;
       variable ncount : bit_vector(2 downto 0)

    begin
       -- code to detect one or two sync characters
    end hunt_mode
```

```
receiver : process
    begin
       //...
       if (control(7))
          hunt_mode(rxd,drdy, sync_chars(7 downto 0),
                    sync_chars(15 downto 8) mode);
       //...
    end process
end behaviour
```

Figure 2.7 Skeleton body for serial_receiver entity and procedure hunt_mode

VHDL is a very comprehensive language that has been developed to allow a design to be defined using a wide range of levels of abstraction. The above discussion has ignored the constructs which support design at the lower levels of abstraction, and efficient simulation.

While the language has developed into a standard for hardware specification and simulation, it is not well suited for input into the synthesis process. The language is very complex with a large number of constructs, some of which lack a full definition of hardware semantics (De Micheli, 1990).

2.2 Verilog

Verilog was originally developed in the northern hemisphere winter of 1983/84 by Cadence design systems Inc. as a proprietary verification and simulation language (Thomas and Moorby, 1991). More recently it has become openly available for any CAD tool to read or write and is currently the most widely used HDL in industry for both simulation and synthesis.

Verilog, like VHDL, provides a digital system designer with a means of designing a system with a wide range of levels of abstraction. As the language was originally developed for simulation, a number of its constructs are not suited to synthesis.

The Verilog HDL describes a system as a set of *modules*. Each module is made up of two parts: an interface and a definition of its contents. A module represents a logical unit that can be described by specifying its internal logical structure, or describing its behaviour in a program-like manner. Modules can be connected by *nets* (wires) allowing them to communicate.

Figure 2.8 contains a top level module definition of the i8251 interface chip which uses the same structure as the VHDL description depicted in Figure 2.3 (page 9).

```
module i8251(chip_data, RE, WE, data, CS, xdrdy, txd, rxd, drdy);
    input       chip_data;
    input       RE;
    input       WE;
    inout [7:0] data;
    input       CS;                                                    (a)
    input       xdrdy;
    output      txd;
    input       rxd;
    input       drdy;
    inout       syn_det;
    output      rx_rdy;
    output      tx_rdy;
    output      tx_empty;

    wire        bi_st;
    wire [15:0] sync_chars;
    wire [7:0]  tx_data, rx_data, status, mode, control;               (b)
    wire [4:0]  rx_st;
    wire [3:0]  tx_st;

    bus_interface BI (chip_data, RE, WE,CS, status, rx_data, tx_data, mode, control , sync_chars,
                      data, bi_st);
    serial_transmitter  ST(tx_full, xdrdy, mode, control, tx_data, sync_chars, txd , tx_st);
    serial_receiver  SR(drdy, rxd, mode, control, sync_chars, rx_data, rx_st);                 (c)
    maintain_status MB(bi_st, tx_st, rx_st, status, tx_empty, tx_rdy, rx_rdy , tx_full, syn_det,
                      control);
end module
```

Figure 2.8 Top level Verilog module definition for i8251

The top level module definition in Figure 2.8 is made up of three parts: 2.8(a) definition of module interface, 2.8(b) declaration of local wires and 2.8(c) definition of modules contents. The first line is the opening module definition where the module *ports* are shown in parentheses. Within the module these ports must be declared to be *inputs*, *outputs* or bidirectional *inouts*. In the example the declaration inout [7:0] data declares a bidirectional port which is an 8-bit vector.

In Figure 2.8(b) all the *nets* which connect the modules of the definition are declared (e.g. wire [15:0] sync_chars;). Nets transmit only the values which are output on them, and do not store

values. Nets represent one of two major data types in the language. The other is the *register* type, which is an abstraction of a storage device. Registers are defined with the *reg* keyword and may be given a size. For example reg [7:0] mode defines an 8-bit register called mode. Memories can also be declared using the register declaration. For example reg [7:0] LUT [0:31]; declares an array of 32 8-bit words. *Integer* and *Time* types also exist in the language for simulation purposes.

Figure 2.8(c) contains the definition of the contents of the i8251 module. In this case the module is described structurally with a list of instantiations of the sub modules which make up the i8251. The Verilog definition in Figure 2.8 contains less information than the original, informal, diagram representing its structure (Figure 2.3). In particular, although wires are defined in the Verilog definition, the direction of the data flow along them has not been defined. This is an undesirable feature as lower level module definitions need to be consulted to understand a higher level definition.

Each of the modules instantiated, can be defined either structurally or behaviourally. The *always* statement provides a means for describing behaviour using similar constructs to those used in conventional programming languages with statements for assignment, iteration and selection. All statements within an *always* statement execute in sequence, although parallelism can be obtained within a module by using multiple *always* statements.

To allow a combinational logic function to be defined, an *assign* statement is provided. Figure 2.9 shows the definition of the bus_interface module using an *always* and an *assign* statement.

```
module bus_interface BI (chip_data, RE, WE,CS, status,
        rx_data, tx_data, mode, control , sync_chars,
         data, bi_st);
    input           chip_data, RE, WE, CS;
    input  [7:0]    status, rx_data;
    output [7:0]    tx_data, mode, control;
    output [15:0]   sync_chars;
    inout  [7:0]    data;
    output          bi_st;

    reg [7:0]   tx_data, mode, control, temp;
    reg[15:0]   sync_chars;
    reg[7:0]    data_out;
    reg         bi_st;
    `define DT_READ 2'b01
    `define DT_WRITE 2'b10
    `define DT_IDLE 2'b00

assign
    data = (!RE && !CS) ? data_out : 8'bz;

always
    begin
      /*intialise signals*/
      bi_st = 0;
      control = DT_IDLE;

      wait(!CS && !WE && chip_data);
      // receive mode word
      temp = data;
      mode = temp;
      if (!(temp[0] && temp[1]))
          begin  /* get sync characters */
            wait(!CS && !WE && chip_data);
            /* get first sync char */
            syn_chars[7:0] = data;
            if (!temp[7])
              begin
                /* get second sync char if necessary */
                wait(!CS && !WE && chip_data);
                syn_chars[15:8] = data;
              end
          end

      forever
          begin
            case {CS, WE, RE, chip_data)
              4'b0100 :  /* read data */
                begin
                  data_out = rx_data;
                  bi_st = DT_READ;
                  #1 bi_st = DT_IDLE;
                end
              4'b0101 :    /*read status*/
                  data_out = status;
              4'b0010 : /* write data*/
                begin
                  tx_data = data;
                  bi_st = DT_WRITE;
                  #1 bi_st = DT_IDLE;
                end
              4'b0011 : /* write control*/
                begin
                  temp = data;
                  control = temp;
                end
            endcase
            wait (CS);
          end
    end
endmodule
```

Figure 2.9 Verilog Module definition for bus_interface in i8251 example

This example also makes use of *`define* construct (e.g. `define DT_READ 2'b01) which is a compiler directive, providing a general textual substitution facility. It can be used to improve the readability of a Verilog description.

The assign statement in Figure 2.9 is used to drive the *inout* port data dependent on the RE and CS inputs. If the condition (!RE && !CS) is true then the data net is driven by the value following the ? (i.e. the output of the data_out register). If the condition evaluates to false then the net is driven to the high impedance state.

In Figure 2.9 each of the output ports (e.g. tx_data) also has a register declared with the same name, which implicitly connects the outputs of the registers to the ports. An output port changes value when a new value is assigned to the register within an always statement of the same name (e.g. tx_data = data;). When the name of an input port appears as part of an expression, the value used to evaluate the expression is the value of the port at the time. Verilog, like VHDL, requires the designer to implement any data transfers between modules.

As Verilog does not provide as comprehensive a set of data types as VHDL, the resulting description is more complex. For example sync_chars, declared as a single net to avoid clutter in the top level structural description, makes the lower level behavioural description more complex. To reference one of the characters its bit range has to be specified (i.e. sync_chars[7:0], and sync_chars[15:8]), which is less meaning full than sync_chars.first and sync_chars.second used in the VHDL case. If, however, two separate nets had been declared to simplify the behavioural description the top level structural definition would have been more cluttered.

To aid in the transfer of data, the language provides a wait statement (e.g. wait(!CS && !WE && chip_data);). If the condition (i.e. !CS && !WE && chip_data) evaluates to TRUE execution continues. If it is FALSE the process waits until it becomes TRUE.

To allow for a behavioural description to be broken into more manageable parts Verilog provides *functions* and *tasks*, which are the same as VHDL's functions and procedures.

2.3 HardwareC

HardwareC (Ku and DeMicheli, 1990) is an HDL that has been designed with synthesis in mind and allows both structure and behaviour to be specified at a high level. The language allows a wide variety of design styles to be represented, and includes features to allow a system to communicate with other systems, subject to a designer-defined handshaking protocol. HardwareC, is based as it name suggests, on the C programming language (Kernighan and Ritchie, 1978). The language has its own semantics, plus a number of added features which make it suitable for hardware description.

Both sequences of operations and structural interconnections of components can be used to describe designs in HardwareC. There are four fundamental design abstractions in HardwareC; *blocks, processes, procedures*, and *functions*. At the top-most level, a design is expressed in terms of a block. A block describes the structural relationships and physical connectivity of the various components of a design. A block definition is made up of instances of other blocks (the language supports such a hierarchy), processes and the interconnections between them. Figure 2.11 shows a block definition for the i8251 interface device using the structure given in figure 2.10. The structure is very similar to that used for the VHDL description (see figure 2.3 page 9) except that a new type of wire called a channel is introduced (shown as a dashed line). A channel allows components to communicate using a predefined communications protocol to be discussed below.

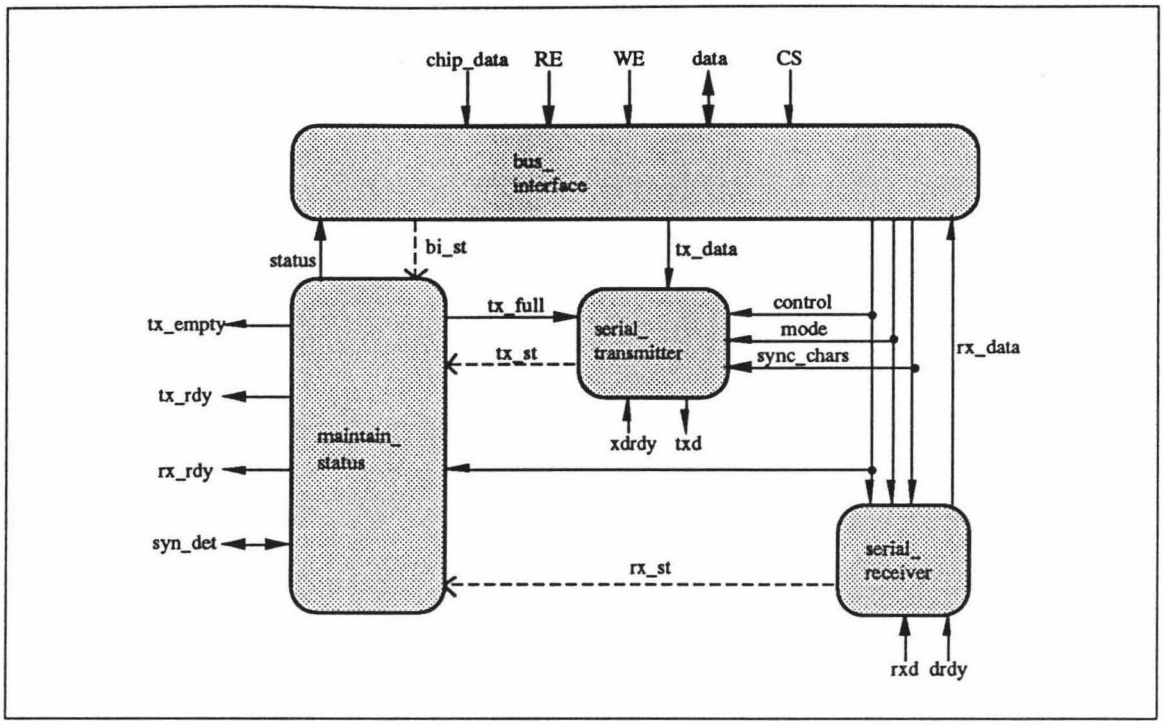


Figure 2.10 Block Diagram for HardwareC description of i8251 example

```
block i8251(chip_data, RE, WE, data, CS, xdrdy, txd, rxd, drdy, tx_empty, tx_rdy,
rx_rdy, syn_det)
    in port chip_data;
    in port RE;
    in port WE;
    inout port data[8];
    in port CS;
    in port xdrdy;
    in port txd;
    in port rxd;
    in port drdy;
    out port tx_empty;
    out port tx_rdy;
    out port rx_rdy;
    inout port syn_det;
<
    channel tx_st[5], rx_st[3], bi_st[1];
    boolean tx_data[8], rx_data[8], status[8], mode[8], control[8], sync_chars[16], tx_full;

    bus_interface(chip_data, RE, WE, data, CS, tx_data, status, bi_st, mode, control, sync_chars, rx_data);
    serial_transmitter(tx_rdy, xdrdy, mode, control, tx_data, sync_chars, txd, tx_st);
    serial_receiver(drdy, rxd, mode, control, sync_chars, rx_data, rx_st);
    maintain_status(bi_st, tx_st, rx_st, status, tx_empty, tx_rdy, rx_rdy, tx_full, syn_det, control);
>
```

Figure 2.11 HardwareC block definition for i8251

In contrast to blocks which define a structure, processes, procedures and functions define an algorithm as a set of operations sequenced in time (i.e. the behaviour). The algorithm consists of data-flow operations such as logic expressions and assignments to shared variables, and control flow constructs such as sequencing, branching and iteration. In addition, calls may be made to other procedures and functions. A process's semantics differ from procedures' and functions' semantics in that processes automatically start execution upon the completion of their last operation, whereas procedures and functions execute only when *called*. Figure 2.12 shows a process definition for the bus_interface module.

```
#define DT_READ 0b0;
#define DT_WRITE 0b01;

process bus_interface(chip_data, RE, WE, data, CS,
                      tx_data, status, bi_st, mode, control,
                      sync_chars, rx_data)
  in port chip_data;
  in port RE;
  in port WE;
  inout port data[8];
  in port CS;
  out port tx_data[8];
  in port status[8];
  out channel bi_st[1];
  out port  mode[8], control[8], sync_chars[16];
  in port rx_data[8];
{
  boolean temp[8], temp_2[8], flag;

  free data;
  while (CS | WE | !chip_data) ;
  /*receive mode word*/
  temp = read(data);
  write mode = temp;
  if (temp[0:1] == 0b01) {
     /*get syn chars */
     while (CS | WE | !chip_data) ;
     /*get first char*/
     temp_2 = read(data);
     write sync_chars[0:7] = temp_2;
     if (temp[7:7]) {
        while (CS | WE | !chip_data) ;
        temp_2 = read(data);
        write sync_chars[8:15] = temp_2;
     }
  }

  while (1) [
     switch (CS @ WE @ RE @ chip_data) {
     case 0b0100 :  /*read data*/
        write data = read(rx_data);
        flag = DT_READ;
        send(bi_st,flag);
        break;
     case 0b0101 :  /*read_status*/
        write data = read(status);
        break;
     case 0b0010 : /*write data*/
        write tx_data = read(data);
        flag = DT_WRITE;
        send(bi_st,flag);
        break;
     case 0b0011 : /*write control*/
        write control = read(data);
        break;
     }
     while (CS) ;
     free data;
  ]
}
```

Figure 2.12 Process definition for bus_interface module.

HardwareC has only three types of variable: *int*, *Boolean* and *static* which correspond to integer, Boolean, and register variables. Other types (e.g. structures, pointers, etc) are not supported. All variables' type and size must be declared before use. A Boolean variable is an array representing one or more binary signals. A number may be represented as a Boolean variable using the 2's complement convention. Boolean variables are initialised to zero, and their values are not saved across model invocations. In the mapping to final hardware implementation, they are synthesised either as wires or as registers. Static variables are similar to Boolean variables except that they are always implemented as registers and their values are retained across procedural invocations. They may be initialised to a given value.

In contrast, *int* variables are provided for the convenience of the description and are valid only if they can be resolved at compile time. Integer variables are often called *meta-variables* to emphasise the fact that they are not synthesised in the final hardware implementation.

A feature not present in VHDL or Verilog allows designers using HardwareC to adjust the degree of parallelism in a process, procedure or function. Operations can be grouped together in three ways: sequential ([...]), data-parallel ({...}) or parallel (<...>). In a data-parallel grouping, all operations execute in parallel unless data dependency requires them to execute sequentially (i.e. an expression in one statement is not dependent on an assignment in another, or vice versa). In a parallel grouping, all operations execute in parallel unconditionally. For example, swapping the values of two variables x and y can be represented as :

```
<
    x = y;
    y = x;
>
```

This will be executed in parallel so x and y will be swapped. If however a data parallel grouping (i.e. replacing <...> with {...}) was used, the transfer would not take place, as the value of x in y = x is dependent on the value assigned to it in the previous line. So both values would become equal to the initial value of y. Within a sequential grouping of statements, all statements are forced to execute in sequence.

To support communication between the different models in a HardwareC specification, HardwareC supports both port passing and message passing. In port passing, a shared medium such as a bus or memory allows ports of models to be interconnected. Values are read from and written to global ports using *read* and *write* statements respectively. It is left up to the designer to implement the communications protocol to ensure the correct transfer of data between models.

In message passing, a synchronous send/receive mechanism is used to synchronise or transfer data along a channel between processes. The corresponding hardware for communication, as well as its protocol, are synthesised automatically. *Send* and *receive* statements exist to allow messages to be passed along a channel. The message passing paradigm is synchronous and blocking, meaning that the sending process will wait until the corresponding receiving process has acknowledged the message. Likewise a receiving process will wait until the corresponding process sends the message.

HardwareC supports timing constraints by associating tags with operations and imposing lower and upper bounds on the time separation between the tags. In specifying interfaces, designers will find the support of timing constraints useful because they can constrain the time between I/O operations.

2.4 Other HDLs

ISPS

The ISPS language (Barbacci 1981) was one of the original HDLs developed to formalise the digital design process at the register transfer and algorithmic levels of abstraction. It was designed for a wide range of applications, rather than a wide range of design levels. The design philosophy of ISPS was guided by two principles, flexibility and simplicity.

ISPS allows a designer to describe the behaviour of hardware units and an interface between them. The interface describes the numbers and types of carriers used to store and transmit information between the units. The behaviour of a unit is described by procedures which specify the sequence of control and data operations.

With the advent of more recent languages such as VHDL and Verilog the ISPS language has been superseded.

YASC

The YASC (Yet Another Silicon Compiler) language (Jhon, Sobelman and Krekeberg,1985) is another example of an HDL designed with silicon compilation in mind and combines both structural and behavioural approaches. YASC is based on dataflow graphs, which were first suggested for use in silicon compilation by Keller, Linstrom and Patil (1980). The dataflow paradigm is based on an behavioural style of system specification, and the use of stream-based data types. Jhon *et. al.* (1985) suggest that this approach bears a natural relationship to the

behaviour of digital systems and allows rigorous design verification at a high level of abstraction to be performed.

YASC has two notable features worthy of mention. First YASC, like HardwareC, supports a message passing paradigm for communication between modules, with *send* and *receive* statements. Second, it supports various degrees of concurrency using three types of block: BEGIN/END, COBEGIN/COEND and PIBEGIN/PIEND. The BEGIN/END block indicates all the contained statements are executed in sequence. The COBEGIN/COEND block indicates all the contained statements are executed in simultaneously. Finally, the PIBEGIN/PIEND block indicates all the contained statements are executed in a pipelined manner.

2.5 Assessment of Existing HDLs

This section examines some of the useful conclusions that have been drawn from the above review of HDLs.

All of the languages use a mixture of structure and behaviour to represent a design. Structure is used at the top level to represent a system in terms of its components and their interconnections. Each of these components could be decomposed and described behaviourally. If a system is large and complex, all the languages allow further structural decomposition, representing each component by a set of sub components. Behaviour is always used to describe the lowest level components.

Verilog and HardwareC had the undesirable feature that a structural description of a design or component of a design did not contain all the information necessary to understand the description. In these languages no information is contained in a structural definition to determine the directions of data transfer between components, requiring the reader to go to lower level descriptions to obtain this information. VHDL did contain this information in its structural definitions. However the resulting descriptions were more verbose than the descriptions in Verilog and HardwareC.

The languages which supported more levels of abstraction had more constructs than those supporting fewer levels making them more complex and harder to learn. As the high level synthesis field matures there will be less need to design at the lower levels of abstraction allowing HDLs to be made simpler and better support design at the high levels.

Sequin (1983) stated that "Abstract terse symbols are crucial to making the *grand picture* visible". In all of the HDLs examined diagrams would greatly aid the designer in the specification and understanding of a design of more than trivial complexity. Diagrams do not however form part of any of the specifications and have to be stored as separate documentation. During the lifetime of a design it is likely that changes will be made to the description including the addition/removal of components and or connections between them. As the diagram does not form an integral part of the description, changes to the description may not be transferred to the documentation. If this documentation could be included as an integral part of a design representation, both would remain consistent, greatly aiding in the management of design complexity (i.e. the first technique listed in Chapter 1 (see page 3) for managing design complexity).

Further, tools for translating a diagrammatic representation of the design into lower-level descriptions would automatically incorporate the designer's top-level ideas with the necessary lower level support structures.

Many VLSI designs contain components that may be repeated many times. VHDL was the only language to facilitate the efficient representation of these repeated components with the inclusion of the generate statement. All the other languages require each instance of a repeated component to be instantiated explicitly which can lead to larger and more complex descriptions.

All the review languages had similar constructs to those found in conventional programming languages supporting assignment, iteration and selection. VHDL also has constructs to support data abstraction with enumeration and record types to aid in managing design complexity and produce more readable descriptions. For example, the two sync characters in the i8251 device were represented by a signal called sync_chars which was defined as a record containing two eight-bit quantities called first and second. In the behavioural description of the bus_interface module the two sync characters output on the sync_chars port could be referred to as sync_chars.first and sync_chars.second. In the Verilog and HardwareC descriptions, a single path was also declared to carry the two sync characters. However the bits representing each had to be specified to reference them (i.e. sync_chars[7:0] and sync_chars[15:8]). Declaring a path for each one would have simplified the behavioural descriptions, while adding more complexity to the high level structural descriptions as two paths would have to be declared instead of one.

For general purpose HDLs, it is important that a designer can specify a wide variety of communications protocols between the device and its external environment. All the languages presented meet this requirement and had constructs to support arbitrary communications protocols. However, generality of communication within a system is not as important as external communication, and it is preferable in most cases for a designer to specify a communications protocol by allowing all components of a design to communicate in a standard way. Both HardwareC and YASC support abstractions to support internal communication using message-passing paradigms, making it unnecessary for designers to have to define their own protocols for internal communication.

From the above discussion it can be seen that the presented languages go only part way to meeting the requirements outlined in Chapter 1 (see page 3) for managing design complexity. In particular, none of the languages described provided any features which allowed the documentation necessary to easily understand a design to be incorporated into the specification (i.e. design equals documentation).

To better meet the requirements outlined to manage design complexity an HDL would need to combine the use of graphics and the good features of current HDLs discussed above.

Chapter 3
Data Flow Diagrams for Hardware Description

Although existing HDLs were created to help minimise the complexity of hardware design, they are unlikely to be sufficiently powerful to deal with the large systems designed in the future.

In this respect, HDLs are going through the sort of evolutionary process which ordinary computer programming languages have undergone during the last 30 years or so. Based on experience gained from programming language development, a number of general conclusions may be reached.

High level programming languages bridge a "semantic gap" between the programmer's mental model of a problem's solution and its ultimate realisation in machine code. The data structures and procedures that compromise the vocabulary of the high-level solution pertain to that problem alone, and are ideally uninfluenced by the architecture of the target machine. By contrast, the machine code provides a set of low-level capabilities that are very general-purpose, but totally oriented towards a specific machine architecture.

Although appropriate high-level vocabularies, and techniques for translating them are well-known in the case of general-purpose programming languages, they are by no means as well understood for HDLs.

To extend the semantic gap analogy further, a programming language might be conceived of as providing piers for a bridge spanning the gap. The programming language primitives would be a pier located near the machine code; mainline procedure calls would constitute a pier nearer to the programmer's mental model of the solution. The compiler would span the gap between the primitives and the machine code; appropriate programming techniques such as top-down decomposition (Sommerville, 1989) would span the gap between the primitives and the procedure calls in the mainline, and there would remain only a small gap between these procedure calls and the programmer's mental model, which - it might theorised - is spanned by inspiration. See Figure 3.1.

In part, the gap exists because mental models tend to exist in a multi-dimensional space, with links in all directions, whereas machine code is unrelentingly linear. This particular aspect of the problem has been attacked by many design methodologies, which are often highly diagram-oriented. They are thus able to represent at least two-dimensional solutions to

problems. For some time, computer technology has been capable of capturing diagrammatic input, and visual programming languages have begun to achieve some prominence as tools for expressing solutions to programming problems.

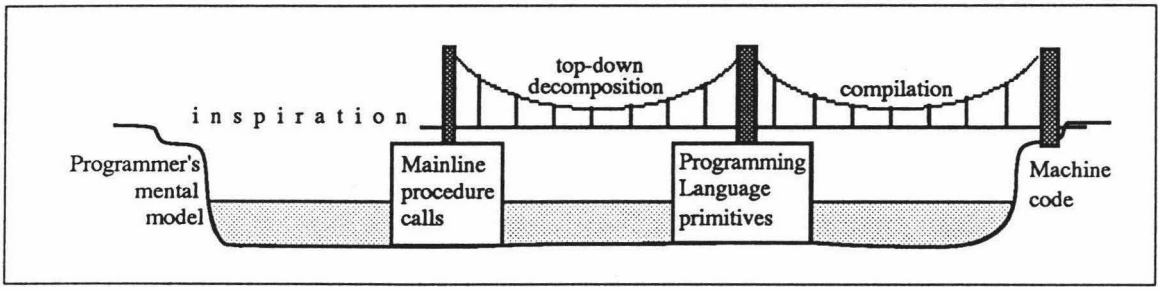


Figure 3.1 How programming languages bridge the semantic gap

Visual programming languages allow a user to specify a program in a two- (or more-) dimensional fashion (Myers, 1990), and providing an environment which enables them to be input and edited directly on a computer is very important for their successful use. Some examples of successful visual language environments include Visicalc (Shneiderman, 1983), LabVIEW (Labview, 1990) and Prograph (Szpakowski, 1989). Although there is a great deal of excitement about visual programming there is a lot scepticism about the success and prospects of the field (Myers, 1990). Much of this scepticism arises from the use of multiple dimensions to represent what must eventually be reduced to a single-dimensional, sequential, representation.

Whatever the validity of these criticisms in the case of general programming languages, hardware design is, by its nature, a two dimensional problem, so is admirably suited to a visual representation. Hardware is often designed in terms of hierarchical components which suit the sort of structured decomposition techniques easily provided by computer-based programming systems. However, personal experience leads the author to believe that the semantic gap is wider for IC design than for traditional programming languages. For example, many of the diagrammatic design aids (structure diagrams etc). which are intended for use as precursors in the design of computer programs are actually often produced only under duress as *post facto* documentation. High-level, *un*detailed, block diagrams, by contrast, are often drawn spontaneously by circuit designers in the early phases of system design, as a way of approaching the solution.

In the course of this work two visual languages have been investigated to establish their suitability for representing hardware designs.

3.1 Structure Diagrams

Structure diagrams (Doran & Tate, 1972) were investigated at the outset of this project. A structure diagram provides a hierarchal representation of a software system.

Figure 3.2 illustrates the components of a structure diagram: rectangles ▭ for operational statements and meta-operations, ovals ⬭ for iteration and hexagons ⬡ for conditional execution and links to denote the parentage of each box except for the root.

A structure diagram is drawn from the top down, starting at the root node (Parallel to serial Conversion in the above example) which is representative of the whole system. This is refined in to a series of subtasks (Read_inparallel_character and Output_serial_character). These subtasks have an implied order of execution from the left most box to the right most box. This refinement continues until the subtasks are simple enough to be readily expressed in terms of a programming language.

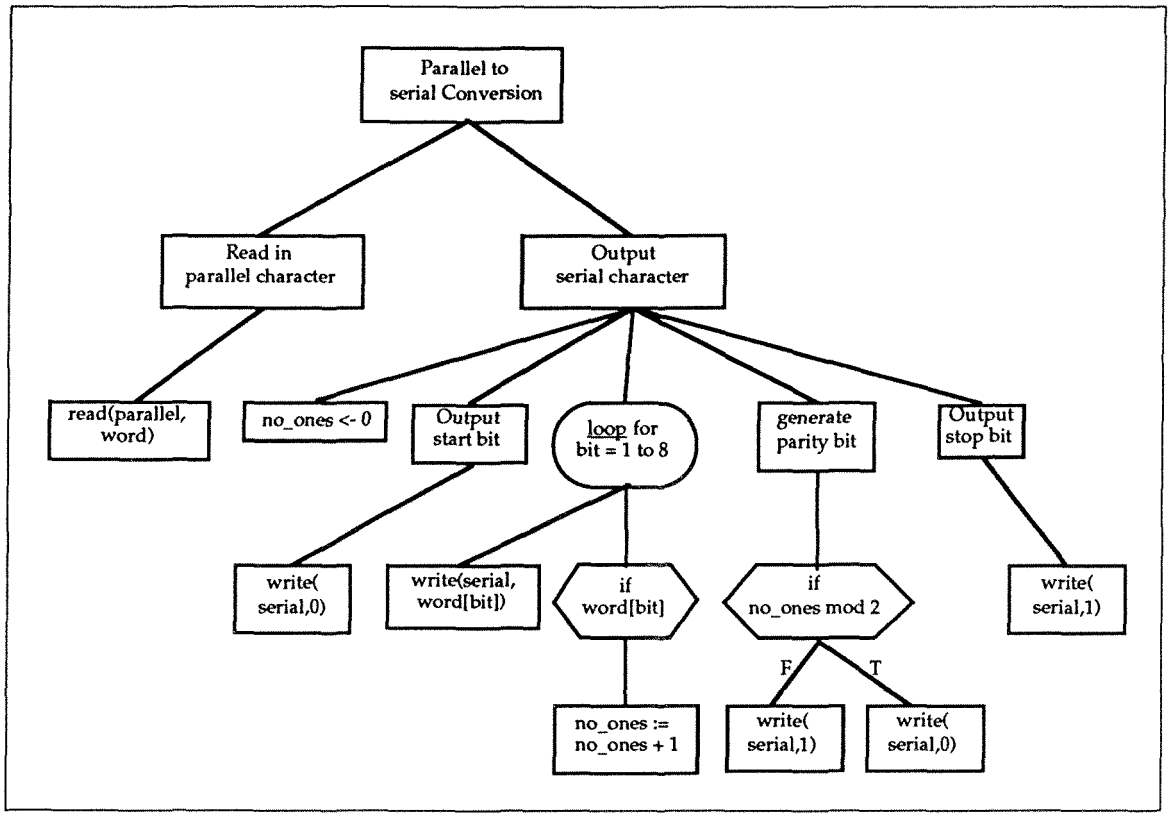


Figure 3.2 An example of a Structure Diagram for a parallel-to-serial converter

In fact, practical investigation of structure diagrams did not extend as far as implementation of an editor, as they were rejected on theoretical grounds. First, it would be difficult to invent an efficient interface for capturing structure diagrams which would be faster and less cumbersome than a text editor for capturing a text based representation of the hardware. Second, a number of hand simulations of hardware design exercises showed that, while structure diagrams represented serial devices well, they had no vocabulary for representing parallelism, nor could they be easily modified to do so.

Thus the search moved away from structure diagrams. Keller, Lindstrom and Patil (1980) and Jhon, Sobelman and Krekelberg (1985) have pointed to the data flow paradigm as one which is well suited to the specification of hardware. In the present work, consideration was given to the adaptation of the structured analysis tools which DeMarco (1978) and Gane and Sarson (1979) have developed for top-down system specification.

3.2 Structured Analysis

Software development is often broken into three main phases: analysis, design and implementation. As system complexity has increased, the analysis phase has become more important, and methodologies such as Structured Analysis have been developed to help add order and rigour.

The principle goal of structured analysis is to minimise the probability of system failure by detecting and rectifying any potential problems early in the project. Rectifying these problems at later stages requires considerably more effort and increases the likelihood of project failure.

A system which is defined in terms of this methodology is first subjected to a structured analysis which generates three types of item; data flow diagrams, data dictionaries and transform descriptions. These are translated to software in the design phase.

Data Flow Diagrams

Within the structured analysis methodology, a system's raw functionality is regarded as being divisible into component functions (processes), interconnected by a network of data flows. The overall representation of a system is in terms of a set of data flow diagrams, each containing processes and data flows. This allows a system to be portrayed as a set of sequential operations on its data, and control is excluded. Parallelism is automatically representable in such a notation. Data flow diagrams are made up of four types of symbol: the named vector (called a data flow) which portrays a data path, the bubble (called a process) which portrays transformation of data, the straight line which portrays a file or data base and the box (called a source or sink) which portrays a net originator or receiver of data. Figure 3.3 shows part of a data flow diagram for a wages system.

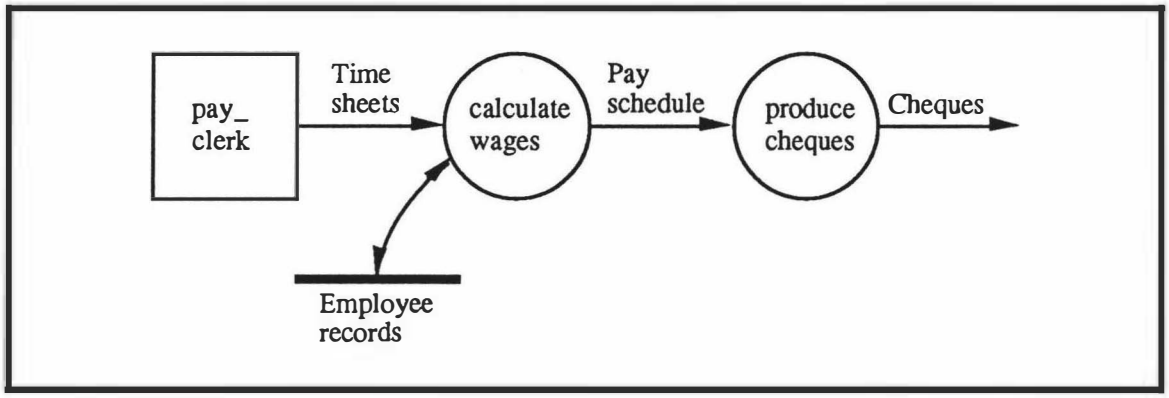


Figure 3.3 Example data flow diagram as used in structured analysis

Data Dictionary

The data dictionary contains a textual definition of each of the interface flows and data stores in any of the data flow diagrams.

Transform Descriptions

The Transform Descriptions specify the operations to be performed on the data by the processes, usually in structured English (DeMarco, 1978).

The suitability of using Structured Analysis tools, Data Flow Diagrams in particular, as a notation for the representation of digital hardware systems was tested by applying them to the design exercises described in Chapter 4. The tests showed that data flow diagrams, with a number of refinements were indeed a good representation for all the exercises. To refine the Structured Analysis tools into a suitable language for hardware description, an iterative process of adding new constructs, and evaluating them against the design exercises was pursued. Chapter 4 will describe in detail the refinements made to the Structured Analysis tools to make them suitable as an HDL. Here we will merely describe the three main groups in brief.

The first type of refinement was to formalise the notation for use as a data capture tool. Data flow diagrams are generally used as documentation rather than as machine input, so notational informalities acceptable in a documentary language were replaced by a more rigid syntax.

The second type of refinement was to augment the notation with constructs for specifying the conditions governing the activation and deactivation of processes. This capability was not present in early data flow diagram notations. However, a number of authors (Ward, 1986 and Hatley and Pirbhai, 1987) have produced extended data flow diagram notations to allow the

specification of process control in real time systems in particular. Their control extensions, all based on the use of finite state machines, are relevant to hardware specification, and have been used as the basis of process control in PICSIL.

Although DFDs have been identified as a suitable basis for a notation for representing hardware, there are some peculiarities of the hardware environment which a software-oriented notation is unable to represent elegantly. Therefore the third type of refinement was to add constructs for routing data in a complex design and for representing repeated instances of a construct.

While Chapter 4 describes the successive refinements made to the Structured Analysis tools, the rest of this chapter defines the PICSIL (for PICtorial SILicion language) HDL which resulted from these refinements[1]. Occasional references to the behaviour of the editor are indicated, to reflect the interactive data capture aspect of the system.

3.3 The PICSIL Notation

PICSIL is a hierarchical specification language for digital systems combining graphical (extended and modified data flow diagrams) and textual (HardwareC (Ku and De Micheli, 1990)) notations to allow the capture of specifications, and subsequent synthesis of IC layouts.

PICSIL allows the control of the data processing functions to be specified separately from the functions themselves. The data processing functions of a system are defined using functional specifications, including data flow diagrams and data dictionary entries. The data dictionary of the PICSIL notation represents both the data dictionary and the transform descriptions of structured analysis. The control of the data processing functions is defined using a controller specification, including state transition diagrams and process activation tables.

Functional Specification

The functional requirements of a system are shown in a hierarchy of data flow diagrams, which break a system down into component functions (*processes*) which are interconnected by a network of *data flows*. The data flow diagram shows at a high level how each process transforms its input (*import*) data flows into output (*export*) data flows and the communication between these processes.

In order to illustrate the definition of PICSIL, reference will be made to the design of a small image processing system which inputs two images as serial data, colour-processes each, and superimposes them. An updatable thresholding function may be applied to the resulting image. The problem has been chosen as its solution involves a large number of PICSIL's features. Figure 3.4 shows a PICSIL representation of this system, as it would appear on the designer's screen. The two processes perform_mapping_stream_1 and perform_mapping_stream_2 use the *data store* color_map to map the colours of the bits in input images stream_image_1 and stream_image_2 onto mapped_stream_1 and mapped_stream_2 respectively. These two mapped_stream flows are then superimposed and thresholded by the process combine_streams to produce the output image new_stream.

---

[1] The notation is presented before the evolution is described, in order to allow the reader to build a coherent picture of PICSIL, without the confusion of the various discarded constructs. Moreover a few of the constructs in the notation are not supported by the current implementation. These are noted at the appropriate places in the discussion

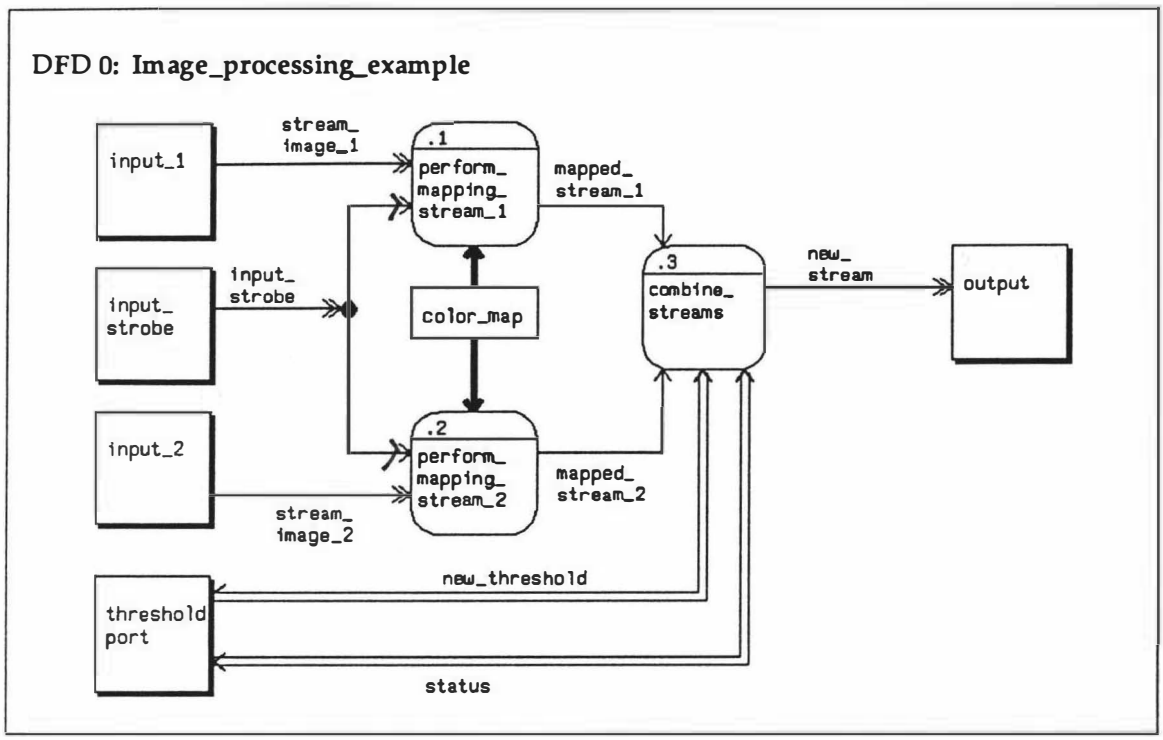


Figure 3.4 Top level data flow diagram for Image Analysis example.

Processes

A process transforms incoming flows into outgoing flows. In Figure 3.4 there are three processes (perform_mapping_stream_1, perform_mapping_stream_2 and combine_streams) which each accept various sorts of data from earlier in the system, alter it, then pass it on.

Each of the processes on this diagram is then decomposed and defined in more detail. If the level of abstraction of a process is low enough for its functionality to be defined briefly and concisely, it is defined in the data dictionary. Otherwise the process is defined as a lower level (or child) data flow diagram. Processes that are defined in the data dictionary are called *primitive processes* while processes that are defined as data flow diagrams are known as *non-primitive processes*.

Process decomposition

Figure 3.5 shows the decomposition of the non-primitive process combine_streams. The process has been decomposed into three sub-processes which in turn can be decomposed. This decomposition of data flow diagrams into increasingly detailed diagrams is called levelling and the resulting set of diagrams is known as a levelled set of diagrams[1].

It is important to note that decomposition of a process does not make new statements about the system, only more detailed ones. As a parent process and its decomposition represent the same information at different levels of abstraction, their inputs and outputs should be identical. The two mapped_stream flows appear as inputs and the new_stream flow appears as an output in both

[1] This somewhat confusing term is standard in the data flow field, and refers to a system whose definition comprises two or more levels, rather than a definition which has *been* levelled

the parent process combine_streams in Figure 3.4 and its child diagram in Figure 3.5. The last two flows in Figure 3.4, new_threshold and status, are group flows (see data flow section) which, in Figure 3.5, have been decomposed into their child flows. For example the flow new_threshold in Figure 3.4 has been decomposed into its three child flows new_threshold\value, new_threshold\ready and new_threshold\accept in the child diagram.

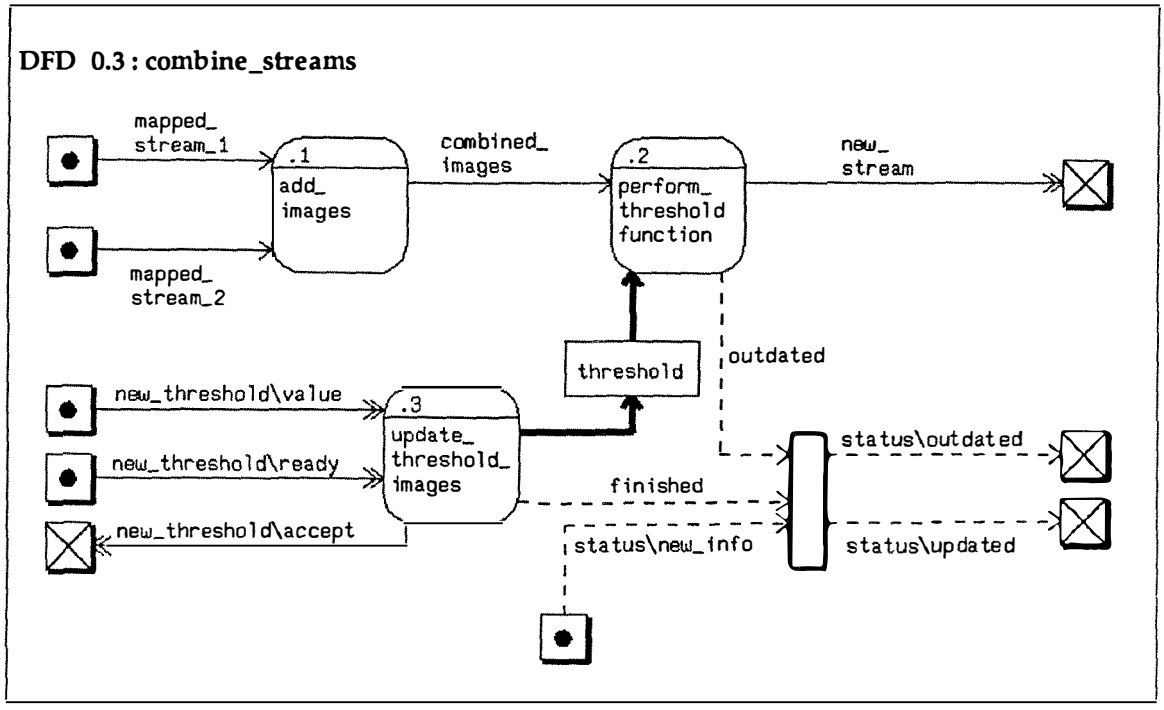


Figure 3.5 The refinement of process combine_streams

Import and Export Links



When the designer first decomposes a non primitive process, the editor will automatically show all the flows into and out of the parent process as import,  , export,  , and bidirectional,  , links. The automatic incorporation of these of these symbols into the child diagram ensures that the inputs and outputs of the child diagram match those of the parent process. If a link does not have a flow attached to it, then the designer can see immediately that the diagram is not balanced. No hardware is generated by the compiler for the links; their function is purely mnemonic.

Every process in a tree of PICSIL data flow diagrams is identified by a label comprising a tree address (automatically assigned) followed by the name of the parent process. As the top level diagram does not have a parent process it is given the name root. The root diagram has tree address 0, and its children are numbered 0.1, 0.2, 0.3, and so on (in the order of their creation). Processes at still lower levels have a correspondingly longer prefix, the children of process 0.2, for example, being numbered 0.2.1, 0.2.2, and so on.

As all processes within a particular diagram have the same prefix to their tree address (the address of the parent process), this is shown at the top left corner of the diagram, and the processes contain only their own unique extension of the prefix.

The actual information processing behaviour of a system is specified in the textual data dictionary entries of its primitive processes. All primitive processes in a system execute concurrently and, unless prevented from doing so by a controller, restart themselves on

completion. Figure 3.6 shows the data dictionary entry for the primitive process add_images. It can be seen that it resembles a C function definition, and can be interpreted in much the same way: the name of the process is specified first, followed by the declaration of its local variables, then statements for input processing and output. A more detailed description of the data dictionary entries for primitive processes is given in Appendix 1.

```
process add_images
seqbegin
   struct {
      boolean{0:8} red;
      boolean{0:8} green;
      boolean{0:8} blue;
   } pixel_1, pixel_2, resultant_pixel;
   parbegin
      receive(pixel_1, mapped_stream_1);
      receive(pixel_2, mapped_stream_2);
   parend
   parbegin
      resultant_pixel.red = ((pixel_1.red + pixel_2.red) >> 1);
      resultant_pixel.green = ((pixel_1.green + pixel_2.green) >> 1);
      resultant_pixel.blue = ((pixel_1.blue + pixel_2.blue) >> 1);
   parend
   send(combined_streams, resultant_pixel);
seqend
```

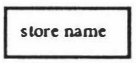
Figure 3.6 Data Dictionary entry for primitive process add_images

External Entities

An external entity is a symbol that represents the interface between the chip being designed and the outside world. It will eventually be synthesised into a set of I/O pads and their associated driver circuitry.

The name given to an external entity should be carefully chosen for mnemonic value of the interface it is representing. External entities may only appear in a top level diagram, meaning that all the inputs and outputs of a system must be shown on the top level diagram.

Data Stores



A data store is a random-access repository for data. Its contents may be accessed (i.e. read or written) by numerous processes. The readout operation is non-destructive. That is, when data is written to a store it may be read any number of times until new data is written in the same location in the store.

The name of a data store should reflect the data that is stored within it. In addition, every data store must have an entry in the data dictionary which defines the number of words and the size of each word the store is to contain. A data store definition for the store color_map is given in Figure 3.7 which defines a store with 32 eight bit words. As this store has no flows that write to it (i.e. all arcs lead from the store to processes), it is a ROM, it must be initialised as shown with the 32 hexadecimal constants enclosed between { and }. RAMs can be similarly initialised[1].

If a process wishes to read data from a data-store, process it and write the result back, and another process wishes to read the same data, a potential consistency problem arises. To

[1] In the current implementation of the PICSIL compiler, stores cannot be initialised.

prevent this, data stores enforce data locking. When a process reads data from a store in read/write mode, the store becomes locked, and no other processes may access it. The store remains locked until either the process finishes its current execution or a statement is executed by the process which explicitly unlocks the store. Any other process trying to access a store while it is locked is suspended until the store becomes available again. Note that processes that use a store in a read-only or write-only mode, only lock the data store for the time the process is accessing stores contents.

```
store {
  boolean{0:8} [32] } color_map =
                        {0X10, 0X64, 0XAF, 0X98, 0X3C, 0XB1, 0X19, 0XDD,
                         0X81, 0XC3, 0X88, 0X2C, 0X9B, 0X87, 0X12, 0XCD,
                         0XE1, 0X3F, 0XFF, 0X00, 0X77, 0XF3, 0X1E, 0X16,
                         0X71, 0X56, 0X5F, 0X8A, 0X23, 0X44, 0X00, 0XFF};
```

Figure 3.7 Data dictionary entry for the data store color_map

A given data store may only appear in one diagram. If its import or export store flows are to be used on other diagrams, they may be connected to non-primitive processes, allowing them to appear in lower level diagrams. In this case the diagram on which the data store should appear is the highest level diagram where the store is accessed by two or more processes.

Data Flows

A data flow is a communications channel carrying data between other components of the PICSIL design. A data flow can carry one or more bits. The flow arcs show the names, types and directions of data that flow between the various components of a design. Six types of flow exist in the PICSIL notation; Discrete, Continuous, Store, Group, Event and Continuous Event.

The first two flow types, discrete and continuous, are used to represent explicitly the different conditions applying respectively to data transfers between processes and to data transfers off-chip. Store flows are provided to indicate that a process uses the contents of a data store in transforming its inputs to outputs.

To help avoid clutter on data flow diagrams, group flows can be used to group two or more flows together and show them as a single flow. Lastly, event and continuous event flows allow control to be portrayed and they will be discussed in the section on the controller (page 41).

Discrete flows

——————→ Discrete flows such as combined_images (see Figure 3.5) are used to move information between processes within a DFD. An information transfer takes place when both the sending (exporting) process and the receiving (importing) process are ready. The designer is freed from having to design an inter-process synchronicity protocol, as a specification for hardware to implement the necessary data-driven communications protocol is automatically generated in the synthesis process.

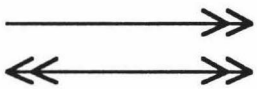
Send and *receive* statements are provided in the data dictionary language for specifying transfer of data over discrete flows. For example, when the process add_images wishes to send data on the discrete flow combined_images, a *send* statement is executed (e.g. send(combined_images, data);). The process perform_threshold_function receives new values off the flow by executing a receive statement (e.g. receive(combined_images, data);).

Every discrete flow must have an entry in the data dictionary which defines exactly the type of data the flow will carry. Figure 3.8 shows the data dictionary entry for the discrete flow mapped_stream_1. Note its similarity to a C declaration. A full definition of the syntax for the data dictionary entries for discrete flows can be found in Appendix 1.

```
Flow discrete struct{
        boolean{0:8} red;
        boolean{0:8} blue;
        boolean{0:8} green;
} mapped_stream_1;
```

Figure 3.8 Data Dictionary entry for the discrete flow mapped_stream_1

Continuous flows



Continuous flows such as stream_image_1, on the other hand, are provided to allow interfacing with other modules external to the design which do not necessarily conform to PICSIL's communications protocols. In such cases, synchronising the data transfer along continuous flows must be handled explicitly by the designer. In the image processing device, there are two different examples of communications protocols. The first example concerns reading in the image streams to the perform_mapping_stream processes. The continuous flow stream input_strobe provides a clock[1] which is pulsed every time there are new pixel values to be read from the stream_image continuous flows. The second example concerns the reading in of new threshold values by the combine_streams process and uses a special handshaking protocol. This example is covered in more detail in the section on group flows.

Values are read from, or written to, continuous flows using the data dictionary language's *read* and *write* statements. For example, when the statement write(new_stream, thres_value); is executed by the process perform_thresholdfunction, the value of the variable thres_value is output on the continuous flow new_stream. The value output remains on the flow until a new value is output. When a read statement is executed (e.g. read(stream_image_1,index);), the value input is the value of the continuous flow at the time the statement is executed.

A continuous flow with arrows at both ends, known as a bidirectional continuous flow, indicates that data can flow in both directions in a half duplex fashion. It is left to the designer to ensure that only one object outputs to a bidirectional continuous flow at a time.

All continuous flows must be defined in the data dictionary. The data dictionary entry for the flow stream_image_1 is shown in Figure 3.9. A full definition of continuous flows can be found in Appendix 1.

```
Flow continuous struct{
        boolean{0:8} red;
        boolean{0:8} blue;
        boolean{0:8} green;
} stream_image_1;
```

Figure 3.9 Data dictionary entry for stream_image_1

[1] This externally supplied clock is completely independent of any internal system clock that is used in the hardware realisation of the design.

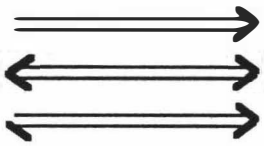
Store Flows

Store flows allow processes to access the contents of a data store. A store flow is made up of three components: an address, which specifies which location in the store is addressed; locking information, which is supplied by the PICSIL compiler to prevent deadlocks; and the data which is being read or written.

The data dictionary language's *stwrite* and *stread* statements allow a process to access the contents of a data store. *Stwrite* allows the results of an expression to be written to a particular location in the named store. For example stwrite(threshold[index],data) writes the value data into the data store threshold at the address given by index. *Stread* allows the contents of a particular location of the name store to be read into the named variable. For example stread(threshold[index],thres_value); reads the contents at the address index of the data store threshold into the variable thres_value.

Store flows do not require an entry in the data dictionary as their type (data, address and locking information fields) can be derived automatically during the synthesis process. Store flows are unnamed and are used to indicate that a process accesses and/or updates information in the connected data store. The direction of the arrow indicates the mode in which the process accesses the store: read only (arrow points in direction of process), write only (arrow points in direction of process) or read/write (arrow on both ends of the flow).

Group Flows



Group flows can be used to allow two or more data flows to be grouped together and shown as a single flow. To aid the readability of data flow diagrams the name given to a group flow should be representative of all the flows (child flows) that make it (the parent flow) up. Group flows are purely notational and do not imply any physical linking (such as synchronicity) of the flows that make up the group. They only exist to help condense and avoid cluttering a diagram.

Group flows can be decomposed into their child flows as they pass from a parent to a child diagram. In an editor to input PICSIL designs, when a process is first decomposed that has a group flow attached to it, a link is automatically shown for the flow in the child diagram. There is an operation provided in the editor to decompose this link into the appropriate number of child links. When a group flow is decomposed, each of the child flows is labelled with the name of the parent flow and the name of the child flow separated by "\". For example the group flow called new_threshold in Figure 3.4 is decomposed in Figure 3.5 to the three flows: new_threshold\value, new_threshold\ready and new_threshold\accept. When a group flow is decomposed, all its child flows are shown.

The flows new_threshold\ready and new_threshold\accept are responsible for synchronising the transfer of a new threshold value on the flow called new_threshold\value. In the top level diagram, these three flows are grouped, as their precise structures do not help the reader of the diagram to get an overview of the functionality of the system. The more important information at this level of abstraction is that new threshold values are imported by the combine_streams process.

The direction of the arrowheads on a group flow is purely notational and should be chosen to aid the reading of a data flow diagram, showing the directions of the *principal* flows that make up the group flow. For example, in the new_threshold group flow, the principle data flow is used for inputting a new threshold value so the group flow is shown with the same direction as the

flow new_threshold\value. A half arrow can be included on the other end of the group flow to indicate that minor data also flows in the opposite direction.

Group flows can contain any mixture of other group flows, discrete data flows, continuous data flows and event flows (to be discussed in the "Controller" section). A group flow may be imported or exported by a primitive process. A definition of a group flow is required in the data dictionary which names the child flows (which have their own data dictionary entries) and gives their directions. An example for the group flow dictionary is given in Figure 3.10.

```
Flow Group (threshold_port : combine_streams) {
    -> value,
    -> ready,
    <- accept;
} new_threshold;
```

Figure 3.10 Group flow data dictionary entry for new_threshold

Connectors

● Connectors allow the same data to be present on several continuous or store flows, much as a conventional computer bus allows the same signals to be presented to a number of devices simultaneously.

Connectors can be used with continuous data flows to allow data to be sent to multiple destinations or received from multiple sources or both. In Figure 3.4 the continuous flow input_strobe is replicated and sent to both the perform_mapping_stream processes. If data is received from multiple sources, it is up to the designer to ensure only one of the sources is outputting to the flow at a time. All continuous flows connected to a connector must have identical data dictionary definitions.

Connectors are mainly used on store flows to allow several processes in a child diagram to access a data store in a parent diagram. One store flow is brought into the child diagram and replicated for each process that accesses it. Figure 3.11 shows an example of the bidirectional store flow registers being replicated three times in the child diagram interface_serial.

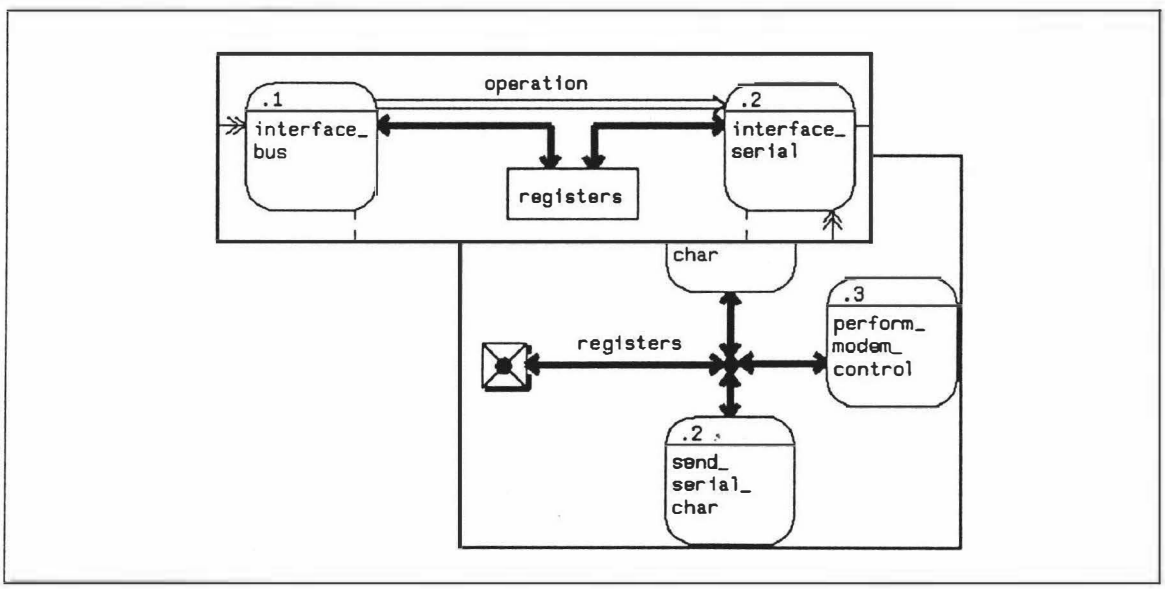


Figure 3.11 Example showing how a store flow can be replicated using a connector

A store flow may also be replicated in the same diagram if the designer feels that it will make the diagram less cluttered and easier to understand.

Routers

The router is provided as a tidy means to specify communication between more than two processes. It is a general purpose data steering device which allows an arbitrary switching of the data on its import flows to its export flows. Figure 3.12 is an example of a packet switch in which four processes send data to any of another four processes.

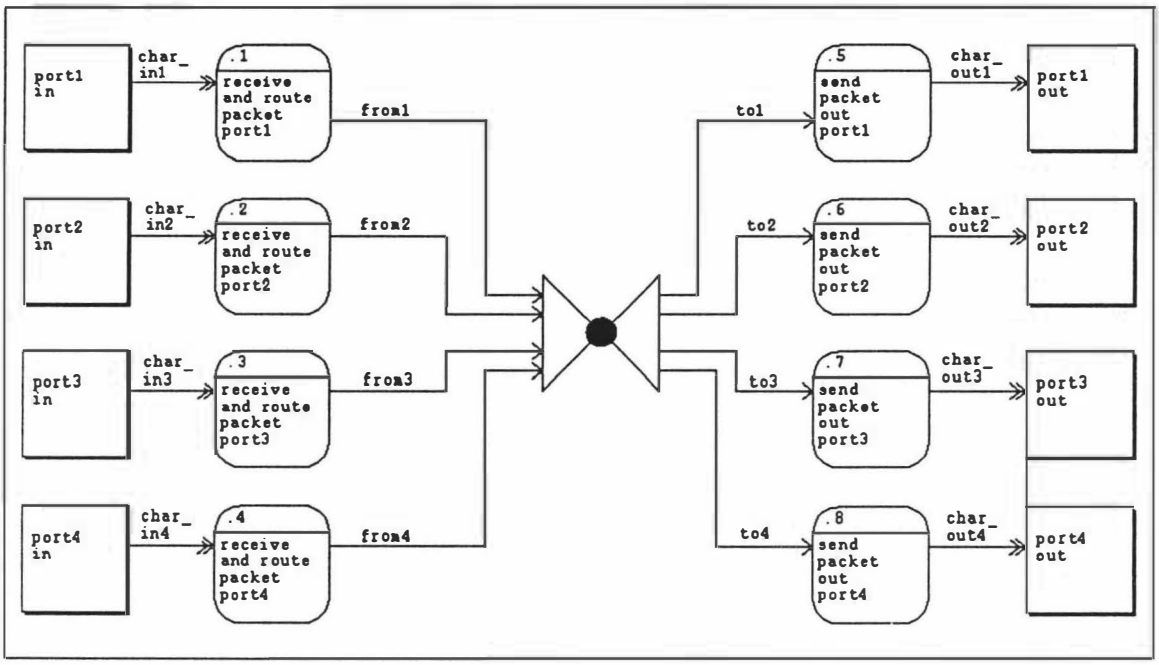


Figure 3.12 Packet switch using router

The route that data takes from an import flow (e.g. from1) to an export flow (e.g. to5) through a router is dependent on a connection being established by the exporting process (receive and route packet port1). Before an exporting process can send any data to a router it must lock a connection in the router from an import flow to an export flow using a lock statement (e.g. lock(from1,to5);). If another exporting process (e.g. receive and route packet port3) has already locked a connection to the same router export flow, the execution of the lock statement is suspended until the export flow becomes available.

Once a connection through a router is locked, it is available only to the process that locked it, until that process finishes its current execution, or it unlocks the connection using an *unlock* statement. When a path is locked from a router import flow to a router export flow, any data transfers along the import flow are automatically passed to the export flow with the data driven communications protocol between the two being maintained.

The maximum number of connections that can be locked through a router at any one time is the lesser of the number of import flows into the router and the number export flows from the router. All the flows connected to a router must be of the same type so that data on any import flow can be transferred to any export flow. No provision for any addressing or extra synchronising information has to be made in the definition of discrete flows attached to a router as it is automatically provided by the synthesis process.

Elements

VLSI designs often contain several, (sometimes many) identical sub-circuits. The PICSIL notation allows replicated sections of a design to be shown only once by including them in an *element*. (represented by a dashed box). A statement indicating the number of times the enclosed section is repeated in the top left corner of the element. Figure 3.13 shows a PICSIL diagram for an extended version of the earlier image processing example. This one reads in four images.

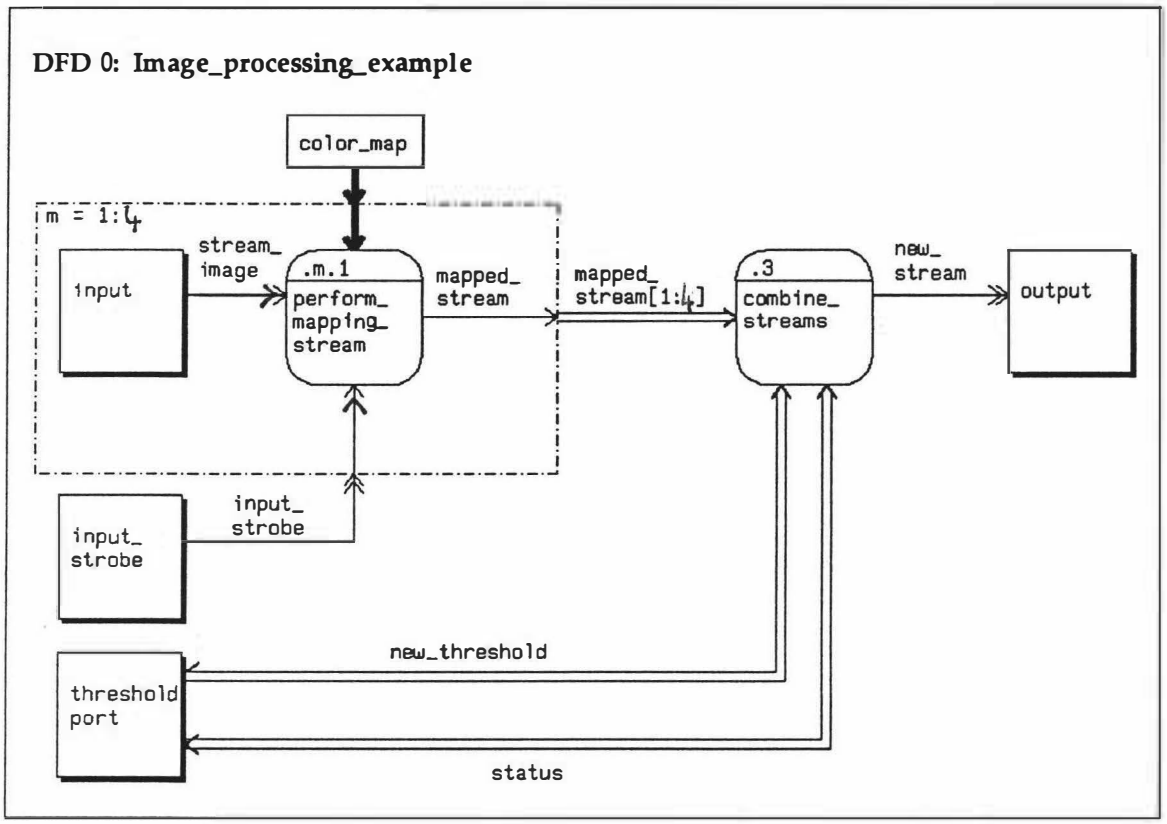
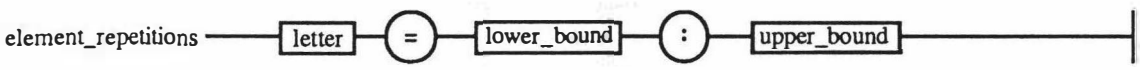


Figure 3.13 Top level data flow diagram for Image analysis example using an element.

The statement indicating the number of replicated occurrences has the syntax:



where letter can be any letter of the alphabet and both lower_bound and upper_bound are constant values. The number of occurrences of the element, m, is equal to ((upper_bound - lower_bound) + 1).

As an element represents n occurrences of part of a design, any flow that leaves the element becomes n flows outside its boundary. To represent this, any flows which cross an element boundary are shown as a group flow with n child flows. The child flows have the same name as the interior flow with a subscript comprising an underline character followed by the flow's occurrence number (e.g. mapped_stream_2). All the child flows have the same data dictionary entry as the interior flow.

Two exceptions to this rule exist, which are both shown in Figure 3.13. Both store flows and continuous flows may cross over an element's boundary, indicating that the data on each of the

occurrences of the interior flow is the same as that on the exterior flow. In the extended image processing example each of the perform_mapping_stream process occurrences can access the color_map store in a read-only fashion.

Data Dictionary

Data flow diagrams allow the designer to define the high level overview of a system, whereas the data dictionary is used to define the more detailed aspects of a system. The data dictionary in PICSIL combines both the data dictionary and transform descriptions of the Structured Analysis notation, allowing the definition of primitive processes, discrete and continuous flows and data stores.

The tools that have been traditionally used for defining primitive processes are *Structured English, Decision Tables* and *Decision Trees* (DeMarco 1978). These tools are however intended for the analysis and early phases of system design and do not have the formal semantics required for the implementation phase of a project development.

A number of programming environments based on data flow diagrams (Babb, 1982, Babb, 1984 Burns & Kirkman, 1986, Strong 1987 and Docker, 1989) have been developed to aid programmers deal with complexity of large software systems. Most of these environments use conventional procedural programming languages to define the behaviour of primitive processes. Docker, however, uses a functional language in which each of the export flows of a process is defined as a function of the import flows.

In selecting a language for defining primitive processes in the PICSIL notation, a class of programming languages called data flow languages (Ackerman, 1982) was considered. A data flow language is based entirely upon the notion of data flowing from one function entity to another. This flow concept gives data flow languages the advantage of allowing program definitions to be exclusively represented by graphs, which is a more natural representation than conventional languages.

A major problem found with data flow languages for describing primitive processes was their inability to specify the sequence of actions. For example, consider the transfer of a new value over the group flow new_threshold in the image processing example. When the external device connected to the threshold_port is ready to transfer a new value to the process update_threshold_images in Figure 3.5 (page 29) it places the new value on new_threshold\value and asserts the new_threshold\ready signal to true. When the update_threshold_images receives this signal and is ready to receive a new set of data, it sets new_threshold\accept to true, reads the new value and then sets new_threshold\accept back to false. Finally the external device removes the new value from new_threshold\value and reasserts new_threshold\ready back to false. Due to the nature of data flow languages, it is not possible to specify this sequence of events.

The choice of an appropriate graphical language has been predicated on the assumption of the ready availability of low-cost graphics-capable work stations and toolboxes for supporting interactive graphics applications. Consequently it has been possible to choose a graphical language taking into account only its suitability for the task.

However the selection of a textual language for defining the lower-level system functions has been more pragmatic than the choice of DFDs as a graphics language. The textual approaches discussed so far (Structured English, Data Flow Languages, Pascal-like languages) have all been rejected for linguistic reasons - appropriateness of the vocabulary and so on. The subsequent discussion has concluded that one of the currently available HDLs is most likely to provide the most appropriate tool to base the textual language on.

In selection of an HDL, consideration was given to the mapping of data flow diagram constructs to the HDL constructs, and the availability of a synthesis system which accepted the HDL as input. Using these more mundane considerations, HardwareC was chosen as the language upon which to base the PICSIL data dictionary language on (see Chapter 6).

As the PICSIL data dictionary language is based on HardwareC, the language for defining the data objects also has a C-like syntax. In addition to the structured analysis-style data dictionary and transform descriptions which the PICSIL data dictionary can contain, it can also include a third component, a data dictionary appendix which is used to define constants and functions which are used throughout a design. A number of examples of data dictionary and data dictionary appendix entries for the image processing device are shown in Figures 3.14 and 3.15 respectively.

```
/* Data dictionary appendix Entries*/
/* Definition of a constant called COLOUR_SIZE = 7*/
#define COLOUR_SIZE 7
/* Definition of a constant called THRESHOLD_TABLE_SIZE =256*/
#define THRESHOLD_TABLE_SIZE 256

/* Definition of a type THRESHOLD_ENTRY for subsequent use*/
typedef boolean{0:COLOUR_SIZE} THRESHOLD_ENTRY;

/* A function to read in a new value of an asynchronous link*/
function get_next_value(accept,ready,value) return THRESHOLD_ENTRY
      /* Declaration of parameters and their directions*/
      in continuous boolean accept;
      out continuous boolean ready;
      in continuous THRESHOLD_ENTRY value;

      /*statements between "seqbegin" and "seqend" are forced to execute in sequence*/
seqbegin
      /* Definition of a local variable to accept a new value of the*/
      /* asynchronous link*/
      THRESHOLD_ENTRY new_value;

            /* Function sits in null loop until ready = TRUE i.e. sender is */
            /* ready to send a new value*/
      while (!ready)    ;
            /* The current value of the continuous flow value is assigned to new_value*/
      read(value,new_value);
            /* accept line set to TRUE to indicate transfer taken place*/
      write(accept,1);
            /*Function sits in null loop until ready = FALSE i.e. sender has */
            /* received accept = TRUE*/
      while (ready)    ;
            /*asynchronous transfer completes with accept signal being set */
            /* to FALSE*/
      write(accept,0);
            /* the value assigned to "return_value" is the value returned by
            /* the function */
      return_value = new_value;
seqend
```

Figure 3.14 Example entries in the data dictionary appendix for the image processing example

In these examples, the definition of constants (using #define) and new data types (using typedef) are similar to those in the C programming language. The statements for defining the behaviour in both the function and process definition are similar to those found in HardwareC,

except for I/O statements, and different symbols used to mark the different types of block. The format of the *receive*, *read*, and *write* statements has been changed from the HardwareC language to allow a consistent format to be used for all I/O statements. In addition additional I/O statements have been added to the language to cater for discrete event (i.e. *report*) and store flows (*stwrite* and *stread*). The HardwareC symbols for marking the different types of blocks (i.e. { and } for data parallel, < and > for parallel and [and] for sequential) have been changed to more easily understood symbols (i.e. *begin* and *end* for data parallel, *parbegin* and *parend* for parallel and *seqbegin* and *seqend*, for sequential).

```
/*Store definition of threshold store in Figure 3.5.Contains*/
/*THRESHOLD_ENTRIES (i.e. 256) entries each of type THRESHOLD_ENTRY*/
store {
      THRESHOLD_ENTRY [THRESHOLD_TABLE_SIZE]
} threshold;

/* Data flow definition for the flow new_threshold\value in Figure 3.5*/
flow continuous THRESHOLD_ENTRY new_threshold\value;

/* process definition for the process "update_threshold_images" in Figure 3.5*/
process 0.3.3 : update_threshold_images
begin
   /* Definition of local variables*/
   NEW_THRESHOLD new_value;
   int current_threshold;                    /*integer variable used in a loop control*/
   seqbegin
                    /* for 0 to THRESHOLD_TABLE_SIZE - 1 (i.e. 256 iterations) */
                    /* read a value off the threshold flow and write it to the */
                    /* current position in the threshold data store*/
      for current_threshold = 1 to (THRESHOLD_TABLE_SIZE - 1) do seqbegin
                    /* call function get_next_value to read a new value off the threshold data*/
                    /*flow which is assign to new_value. process execution suspended until*/
                    /* function returns*/
         new_value = get_next_value(new_theshold\accept, new_threshold\ready,
                                    new_threshold\value);
                    /*new_value is then written to the threshold data store at the */
                    /*location current_threshold */
         stwrite(threshold[current_threshold], new_value);
      seqend
                /* The event finished is reported to the controller for the diagram */
                /*to indicate that the update of the threshold table has been completed*/
      report(finished);
   seqend
end
```

Figure 3.15 Example data dictionary examples for the image processing example

Appendix 1 describes the data dictionary language in detail. Although it is based on HardwareC, a number of extensions have been made to represent all the necessary constructs associated with data flow diagrams.

Controllers

Although functional specifications make it possible to illustrate the functions a system is to perform, they contain no provision for enabling or disabling a subset of those functions under particular conditions. The decisions that can be represented within a functional specification are restricted to the lowest level, with statements such as if-else and while. In the image processing example shown in Figure 3.5 we do not wish processing to continue (i.e. the

perform_threshold_function process should be turned off) while the thresholding table is being updated by the update_threshold_images process. Another situation where higher level control is required arises in the perform_threshold_function process. In addition to looking up values in the threshold table, it monitors its input values and on receiving some data condition (not important to the example), it requires that the threshold table be updated. When this condition occurs, it needs to be reported off-chip and thresholding should halt until the threshold table has been updated.

This is a higher level of control than can be specified in a functional specification, so a controller which allows this higher level control to be specified has been devised. The controller detects major changes in the systems operating mode and may turn on and off large groups of processes. It also receives information about the status of other components, both internally and externally, and transmits similar information about itself. The controller used in the PICSIL notation is based on extended DFD methodologies of Hatley & Pirbhai (1987) and Ward (1986).

Figure 3.16 shows the communication between the functional specification and controllers. The functional specification describes the data processing path of a system using its data inputs to produce data outputs. Some of the primitive processes produce control signals (data conditions) which are fed into the control structure. These internally generated control signals plus the externally generated control inputs are used to drive the controller to modify the response of the system according to their past and present values, by switching on and off groups of processes (using the process activators) and conveying this information to the outside world using the control outputs.

If a particular DFD requires no high level control, the controller is omitted and all processes in the diagram remain switched on (activated) all the time.

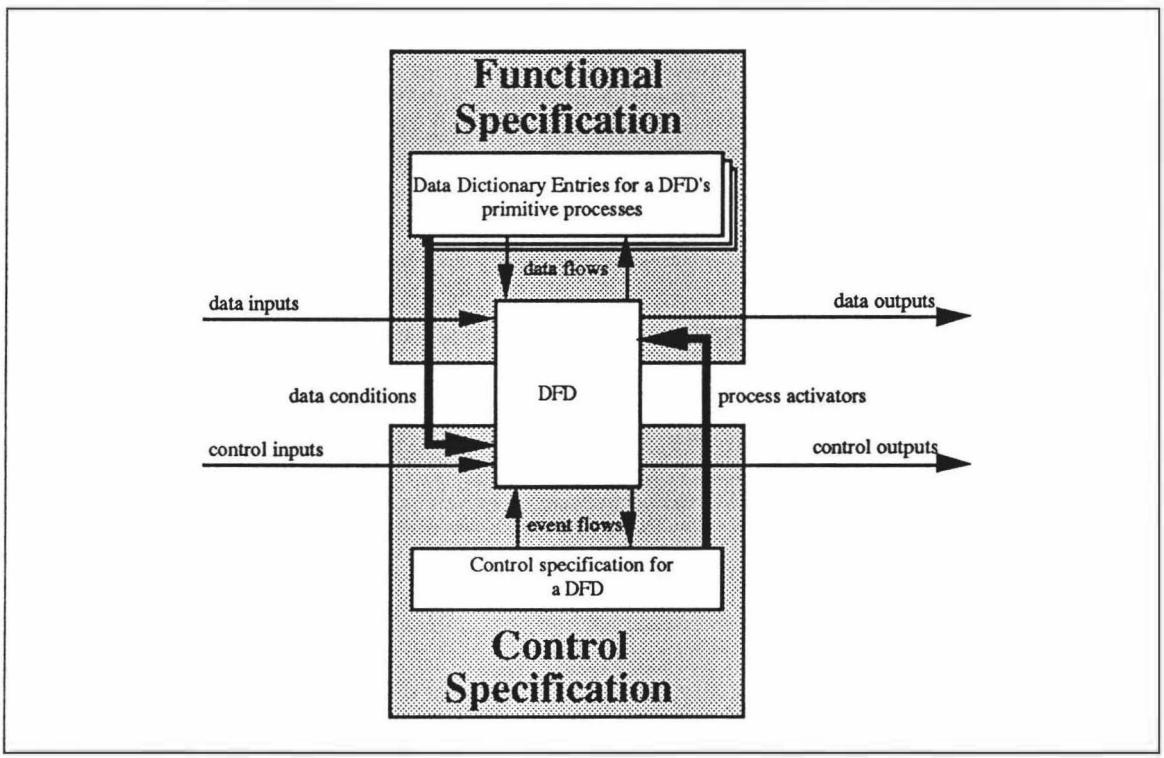


FIGURE 3.16 Relationship between the process model and controllers in the PICSIL notation

Two extra symbols are required in the PICSIL data flow diagram notation to allow control to be specified: *event flows* and *control processes*. Figure 3.5 shows the use of these two symbols to describe the necessary control in the image processing example.

Event Flows

An event flow reports, to the various types of object, events or data conditions occurring inside or outside a PICSIL design. Event flows are not used to activate or deactivate processes. This is done by process activators as discussed in the section on control processes. Two types of event flows exist: *discrete* and *continuous*. The name of an event flow should be representative of the condition it is used to report.

— — — — —> Discrete event flows are used to report events within a system at discrete times and have no other data associated with them. The only time a discrete event flow is valid is when an event is being reported. For example, consider the perfonn_threshold_function which, as part of its functionality, determines the data condition that renders the data in the threshold data store out of date and requires updating. When this data condition is determined it can be reported to a control process using the discrete event flow outdated. This event flow is defined from the time the event is reported by the perform_threshold_function until it is used by the importing control process.

Discrete event flows may be exported or imported by any of the following: control processes, non primitive processes, and import/export links. They may also be exported from primitive processes, but not imported by them. Designers need not concern themselves with the communications protocol used with discrete event flows.

If a discrete event flow such as outdated is exported by a primitive process, a report statement is used in the process definition to indicate that an event (data condition) has occurred. Thus when the perform_threshold_function process receives an internally generated data value (i.e. a properly synchronised signal originating within the design) indicating that the data store is to be updated, it conveys this information to the controller with a report statement:

```
report(outdated);.
```

— — — — ≫ Continuous events flows are used to convey a condition existing over a period of time, and represent a single bit asynchronous I/O channel. A continuous flow may be exported or imported by any of the following: control processes, external entities, export and import links and processes.

Because no synchronisation protocol applies to the continuous event flow, it can only be examined to determine its current value. For example, in the image processing example the continuous event flow status\new_info might be in an idle state when its input voltage is low. When its input voltage goes high, a new threshold value is about to be presented and some change of mode is required so that it can be accepted.

Control Processes

The control process is a representation on the data flow diagram of the interface between the functional specification and a controller. The event flows into and out of a control process are the inputs and outputs of the controller. Although only a single controller may exist for a single data flow diagram, several control processes may be shown on a diagram. Each of the control processes represents an interface to the same controller and prevents the diagrams from becoming too cluttered. As all control processes in a diagram refer to the same controller they do not need to be named.

Conversely, as more than one controller can appear within a PICSIL design (up to one for every DFD) every controller must be named. The name given to a controller is the same as the name given to the DFD that it is defined for except that the label "DFD" is replaced with "controller".

The definition of a controller takes the form of a state transition diagram. A state transition diagram represents a machine which generates outputs dependent on past and current events. Past events are encoded into the current state of the machine, and updated via the transition between those states; current events arrive at the machine as inputs; outputs may be generated whenever a state transition occurs. Figure 3.17 shows a state transition diagram for a controller for the DFD in Figure 3.5.

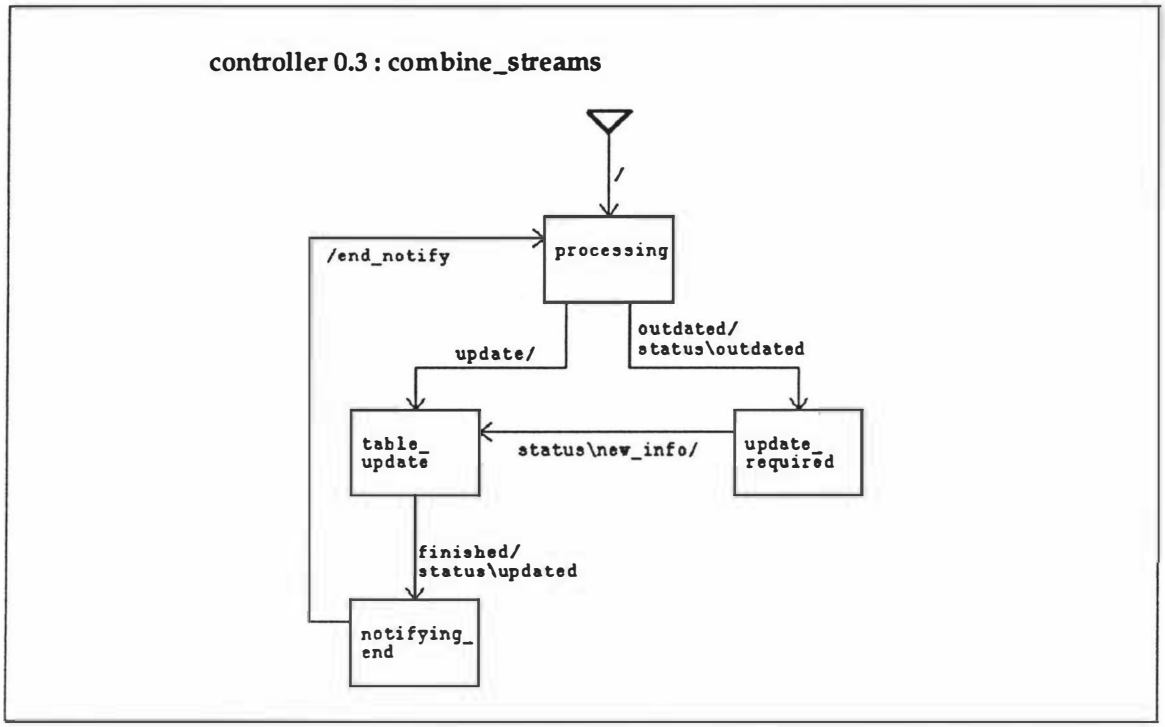
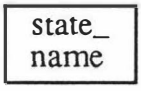


FIGURE 3.17 State transition diagram for controller of DFD in Figure 3.5

States



A state is used to represent a particular combination of past events. In addition to remembering the mode of the system, a state is used to define which processes in the particular diagram should be switched on (activated) or switched off (deactivated). In a given state a particular process can be activated continuously, activated once or deactivated. If activated continuously, it repeatedly performs its function until it is deactivated. If a process is activated once in a particular state, then when the state is entered, the process executes once then remains idle until it is activated again on entering another state. If a process is deactivated on entering a state, its current execution is terminated instantly[1], and it does not execute again until the controller enters a state that activates the process.

To represent this information, a process activation table[2] is used for each state showing the status of each process in that state. Figure 3.18 is a possible process activation table for the state table_update. The activation statuses for each of the processes are shown as icons in the right

[1] Implementation constraints in the current synthesis system make it impractical to terminate a process "midstream" as in the above discussion, so designs which are going to be synthesised need to assume that a process finishes its current execution before being terminated.

[2] Although a graphical form of process activation table is described in this chapter, the PICSIL editor is currently only capable of capturing a textual form of the process activation table.

column of the activation table. The icons are: |▪| for a deactivated process,  for an activated continuously process and  for a once activated process.

add_images	
perform_threshold_function	
update_threshold_function	

FIGURE 3.18 Process activation table for the update_table state

If a non-primitive process is deactivated, then all child processes of that process also become deactivated.

A controller remains in a particular state until an event occurs that causes a transition indicated by the transition arc labelled with that event, and will perform the action associated with the arc on entering the new state. If an event occurs that is not one of the output transitions of a state then nothing happens, and in the case of discrete events, the event is stored till it is used for a later transition. If more than one state transition is possible the controller will pick one of them in a non-deterministic fashion.

State Transitions

The transition arc is made up of three parts: the arc itself, an event and an action, where the event and the action are separated by a /. In a particular transition it is possible to omit the event, the action or both. If an event is omitted then the state transition occurs once all the activations and actions caused in the previous state transition have occurred. An example of this is the transition from the notifying_end state to the processing state in Figure 3.17. The same event may not cause more than one transition from a particular state, and a transition caused by the null event may not be attached to a state with other outward transitions.

Transitions sometimes return the controller to the state it was in before the transition, so that an event may cause an action, but not change the state. The converse can also occur: an event occurs which causes a change in state but no action is produced as occurs with the transition from the processing state to the table_update state.

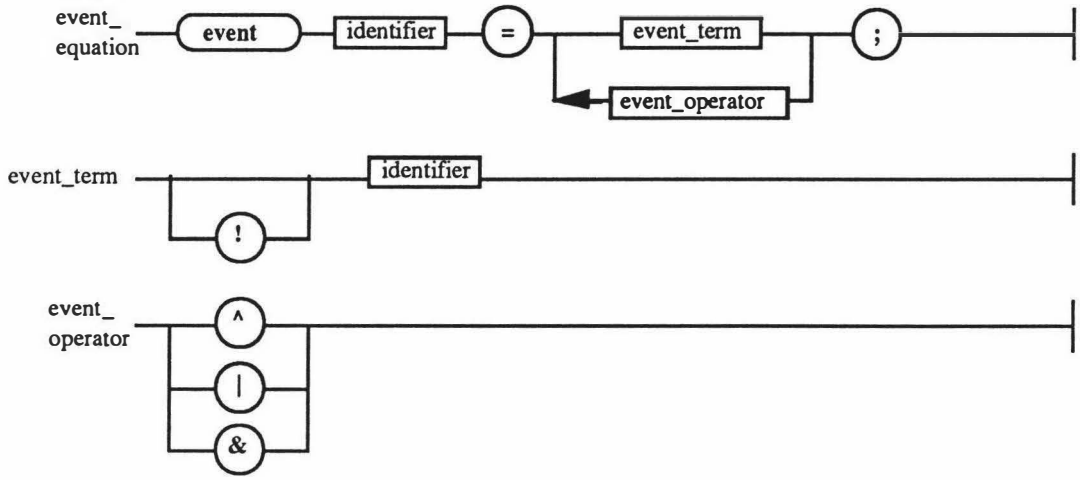
```
event update = !outdated & status\new_info;
action end_notify = status\updated & !status\outdated;
```

FIGURE 3.19 Event and action equations for the state transition diagram of Figure 3.17

An event which causes a state transition is defined as a Boolean expression including one or more event flows imported by a controller. Each discrete event flow which is imported into a controller is associated with a flag. When an event is reported to a controller, the flag associated with the event is set. The object (process or controller) which reported the event can then continue processing. The flag remains set until it is used to cause a state transition at which time it is cleared and the event is said to be consumed. If a subsequent event is reported on the same event flow before the previous one has been consumed it will be lost. No such flag is

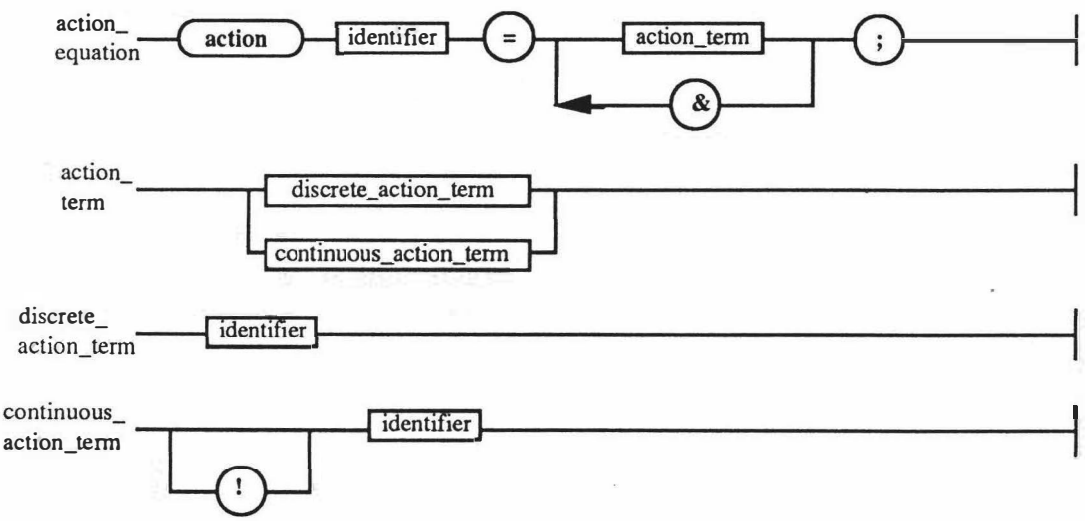
associated with a continuous event flow and the value used is the value that is presented at its input at the time of the transition.

An event label on a state transition arc can be made up of a single imported event flow, in which case the name of the event on the transition is the same as the event flow name, or of several import event flows, in which case the event name on the transition arc is representative of all the event flows that make it up. Such an event is then defined as a Boolean equation which forms part of the controller. Figure 3.19 shows an event equation for the event update from the state transition diagram shown in Figure 3.17. The syntax for an event equation is shown below.



Where !, ^, | and & are *not*, *exclusive or*, *or* and *and* operators respectively. Each term in the *event_equation* must have the same name as one of the event flows imported into the controller for the DFD. If a term which represents a discrete event flow is preceded by a *not* operator, the term will evaluate to true if the event is not present. If a Boolean event evaluates to true then the flags of all the discrete event flows that make up the equation are reset to zero. If a term which represents a continuous event flow is preceded by a *not*, the term will evaluate to true if the flow has a low voltage on it otherwise it will evaluate to true.

Actions report data conditions to other parts of the system or externally. An action label can represent a single event flow exported from a control process, in which case the action label has the same name as the exported event flow. If an action label represents several exported events then an *action_equation* is included in the control process dictionary defining each of the actions separated by &.



Each term in the *action_equation* must have the same name as one of the event flows exported from the controller to the DFD. If a term represents a discrete flow, then an event is reported along that flow. A term representing a continuous event flow will set the flow to false if the term is preceded by a *not*, otherwise it is set to true. Figure 3.19 shows an action equation for the action end_notify.

Power down causes an automatic transition (not recorded on the diagram) from any other state to a notional state represented by a triangle. This state must be present on all PICSIL state transition diagrams, as must a transition from it to an initial state which is automatically entered when power is applied to the system. The event part of the label of this transition must be null, but the action part may be present.

State transition diagrams become intractable when the number of states and/or transitions becomes large. To combat this problem the PICSIL notation allows state transition diagrams to be partitioned in the same fashion as PICSIL data flow diagrams[1]. A state is *primitive* if it can be represented using a process activation table, and *non-primitive* if it represents a number of sub-states.

The state transitions that arrive at and leave a non-primitive state are identical to the transitions that arrive and leave the child state transition diagram. To ensure that this consistency is maintained, the editor should automatically show a transitions into and out of the parent state as import, ⊡, and export, ⊠, links . Figure 3.20(a) shows how the state transition diagram of Figure 3.17 could have been represented as two states processing and update_threshold. The state update_threshold is a non-primitive state and its child state transition table is defined in Figure 3.20(b).

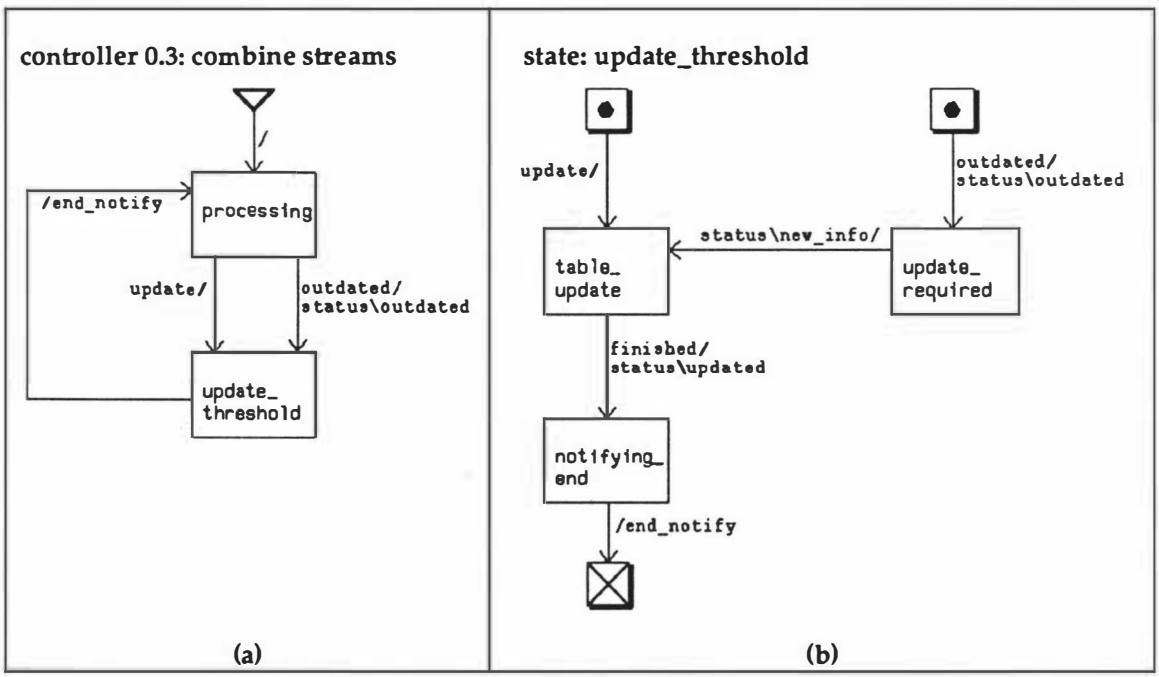


Figure 3.20 State transition diagrams for controller of DFD in Figure 3.5 using partitioning: (a) controller 0.3 : combine_streams and **(b)** Child state transition diagram for update_threshold

[1] Although Multi-level state transition diagrams are discussed in this chapter, the current version of the PICSIL editor is only capable of capturing single level state transition diagrams.

Chapter 4

Design Exercises

During the development of PICSIL, a set of ten design exercises were used as a benchmark against which the language's evolving constructs could be evaluated for generality, conciseness and an improvement of PICSIL's system-level vocabulary. These representative designs were:

- an *Ethernet controller*, which allows data to be sent and received on an Ethernet network;
- a *traffic light controller*, (Mead and Conway, 1980) used to control the lights at an intersection between a main highway and a quiet side road;
- a *packet switching node*, (Lyons and McGregor, 1990) part of the MasseyNet local area network;
- a *pipeline processor*, (Apperley, 1987) used as part of an image processing system;
- a *bifocal display*, (Apperley, 1982) allowing the mapping of a computer screen to be split into three viewports into larger images stored in memory;
- the *MC6850* (Motorola, 1983) *and i8251* (Intel, 1983) serial interface adaptor chips;
- a *parallel to serial converter*, which converts parallel data into a serial format and requires a buffer to accommodate the varying data rates;
- a *Kalman Filter*, a core computational building block in many modern control theory applications;
- an *edge detector*, (Lyons, 91) used to output a pulse for each detected on its two input lines.

Three of the exercises are described in detail below. These are sufficient to document the shortcomings which were found in the early versions of PICSIL, and the resulting developments.

These developments fell into three main groups; formalisation of syntax, introduction of high level controllers and development of a "vocabulary" more attuned to hardware systems.

Although the design exercises proceeded largely in parallel and consequently went through identical evolutionary phases, they are presented here as though their development was serial. That is, the introduction of controllers, the formalisation of syntax and some of the hardware-oriented vocabulary are described only in the first example; later examples are used to describe the evolution of other concepts; top-down design partitioning, and routers (the final item of hardware-oriented vocabulary).

The final PICSIL realisation of each example is also compared with a design for an identical device executed using a conventional HDL such as VHDL, Verilog or HardwareC.

4.1 Ethernet Receiver: Evolution of the basic notation

The Ethernet receiver discussed in this section is a subset of the full Ethernet controller represented as one of the design exercises. An abbreviated version of the design is described because space does not permit the description of the full design. The receiver receives data from an Ethernet connection and stores it in a global memory (see Figure 4.1). Its functionality can be split into three main components: receive character, receive frame and store frame.

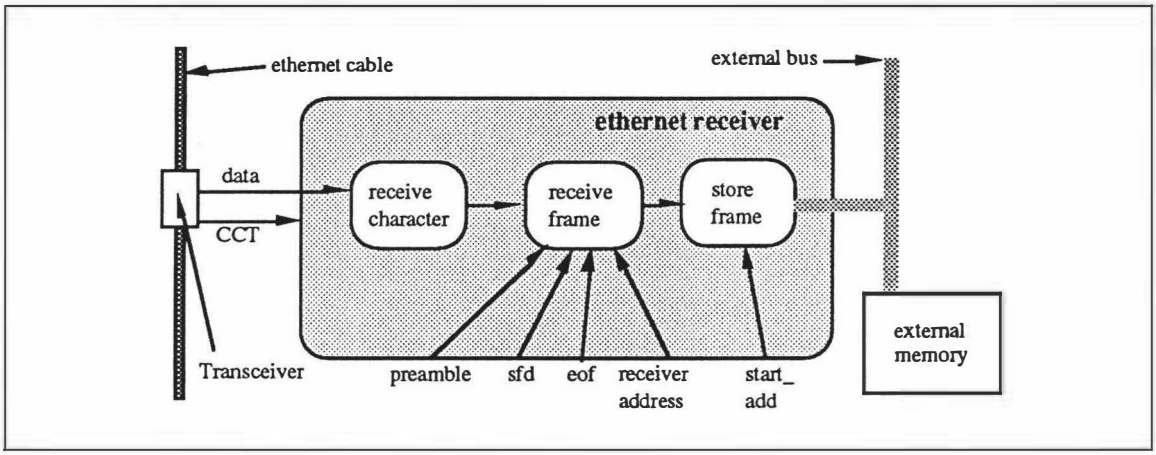


Figure 4.1 Block diagram of Ethernet receiver

When the CCT (collision detect) signal is detected, the receive character component reads groups of eight bits from the data line and groups them into characters to be sent to the receive frame component. The receive frame component receives the characters from the receive character component and builds them up into an Ethernet frame (see Figure 4.2) for processing. To allow the receive frame component to process a frame, lines defining valid values of preamble, sfd (start of frame definition), eof (end of frame) and the receiver address are input into the component. The preamble characters at the start of the frame are for synchronisation and are discarded by the receive frame component. The sfd byte denotes the actual start of the frame.

When the Destination bytes are received they are compared with the receiver address, and if they are different, the rest of the frame is discarded, as it is intended for another node in the Ethernet network. If the receiver address and the destination bytes are the same, the source address, the frame length and the data are extracted from the frame. If the CCT line becomes false at any stage while the frame is being received, the rest of the frame is discarded.

If a horizontal parity byte, generated from the values in the source address, length and data fields, is different from the parity byte in the frame, the frame is discarded. Otherwise a correct eof byte must be received before the frame can be considered valid. The source address, length and data bytes cannot be sent to the store frame component until the frame is validated, so buffering is required.

- p bytes : preamble, used for synchronisation
- 1 byte : start frame delimiter (sfd)
- 2 bytes : destination address (node for which frame intended)
- 2 bytes : source address (node from which frame sent)
- 1 byte : number (n) of data bytes in frame
- n bytes : data
- 1 byte : parity byte
- 1 byte : end of frame (eof)

Figure 4.2 Format of Ethernet frame

Once a frame has been validated, the store frame component stores the source address, length and data bytes in the external memory starting at the address start_add input into the component. However, before this can happen, the component must request and be granted access to the external bus.

The complete Ethernet receiver is a complex piece of circuitry, and to represent it completely using the original DFD-based notation would have led to a design in which the basic functionality was obscured by excessive detail.

Accordingly, a divide-and-rule policy was adopted; during the early phases of PICSIL's evolution, only a subset of the receiver was represented. In this simplified environment, it was possible to devote significant effort to achieving a level of abstraction in the language which represents a balance between information content and design complexity. Consequently, when the more complex full design was implemented, the necessary extra constructs could be developed in a comparatively simple pre-existing framework.

It had earlier been hypothesised (see Chapter 3) that Data Flow Diagrams would offer an appropriate vocabulary for representing high-level architecture. Consequently, the first representation of the Ethernet receiver used conventional data flow diagrams (see Figure 4.3).

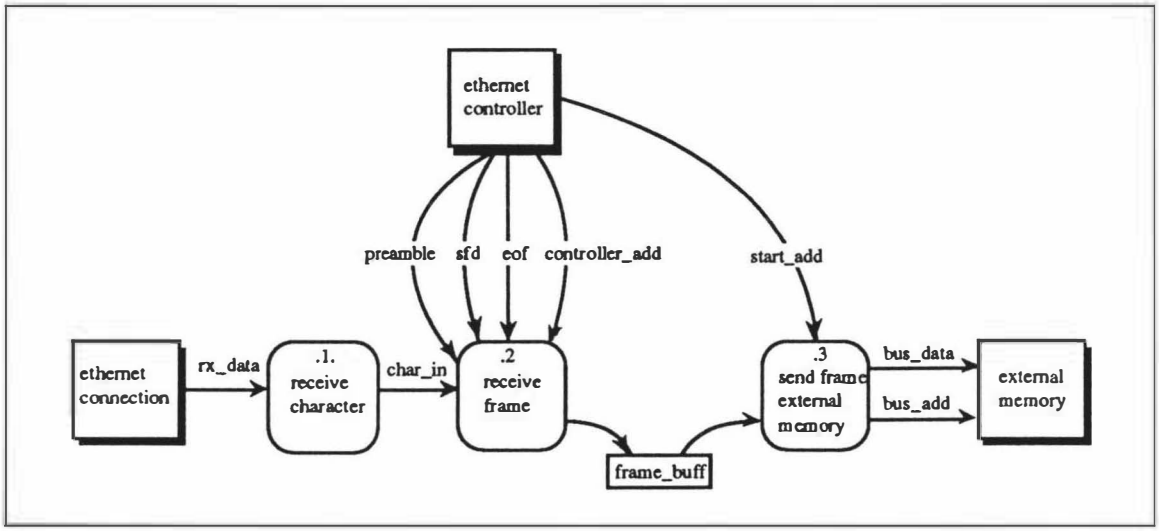


Figure 4.3 Initial representation of Ethernet receiver

As expected, this initial representation provided a natural-seeming architectural-level overview of the operations which the Ethernet receiver performs. The only aspect of the device which it failed to represent was inter-process communication.

Discrete and Continuous Flows

When several processes exist in a system and data is to be passed among them, the processes must be synchronised before data transfers between them can take place. The reason for this is that processes execute independently of one another, and do not necessarily know what state another is in. For example if a process is ready to send data to another process which is not ready to receive it, then the data may be lost if no synchronisation is provided. Without synchronisation it is not possible for the sending process to determine when the receiving process has read the data, and hence the sending process may remove the data before it has been used.

In this initial representation, only one type of data flow existed, and the usual DFD data driven paradigm for information transfer was adopted. That is, it took place when both the exporting and importing objects were ready.

While this model proved adequate for data transfers between processes (see Figure 4.4 in which mutually synchronised send and receive statements are used to implement data transfer) it is insufficient for external communications. For example, in the definition of the Ethernet receiver, a handshaking protocol is required within the device. The communication protocol referred to above is acceptable, as -within the device- the PICSIL system can be expected to generate the necessary synchronisation automatically. However, the receiver must also communicate with external devices and it is unreasonable to impose the overhead of this data-driven communications protocol on all I/O channels such as those to store the received frame in an external memory.

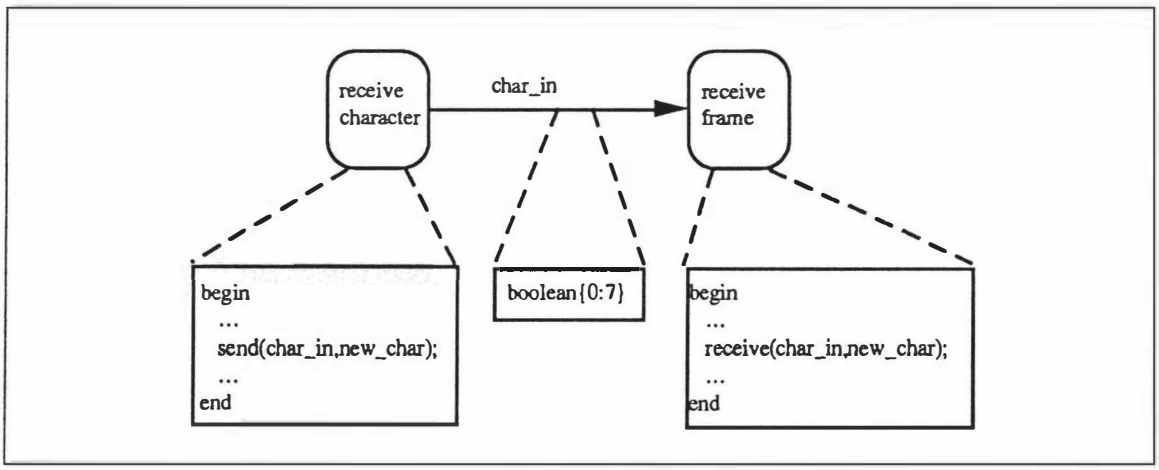


Figure 4.4 Definitions required to define data transfers between processes using a data-driven communications protocol

In order to investigate the desirability of a system which did not automatically provide any data driven synchronisation, a second version of the Ethernet receiver was defined, in terms of a notation which the designer was required to synchronise all communications, both internal and external. While this approach allows far more generality, an extra, unnecessary level of complexity was added both to the diagrams and the definitions of inter-process communication. Figure 4.5 shows the synchronisation implementation in terms of handshaking signals char_in_ak and char_in_rt. The associated overhead would be very common and was felt to be unacceptable.

In order to avoid the drawbacks of both the above approaches, the Ethernet receiver was again redesigned, with a notation which incorporated discrete flows (for which PICSIL provides a data-driven communications protocol) for interprocess communication and continuous flows

(no communications protocol) for external communication. This allowed interprocess communications to be defined simply, yet without restricting the external communications ability, and was the notation adopted.

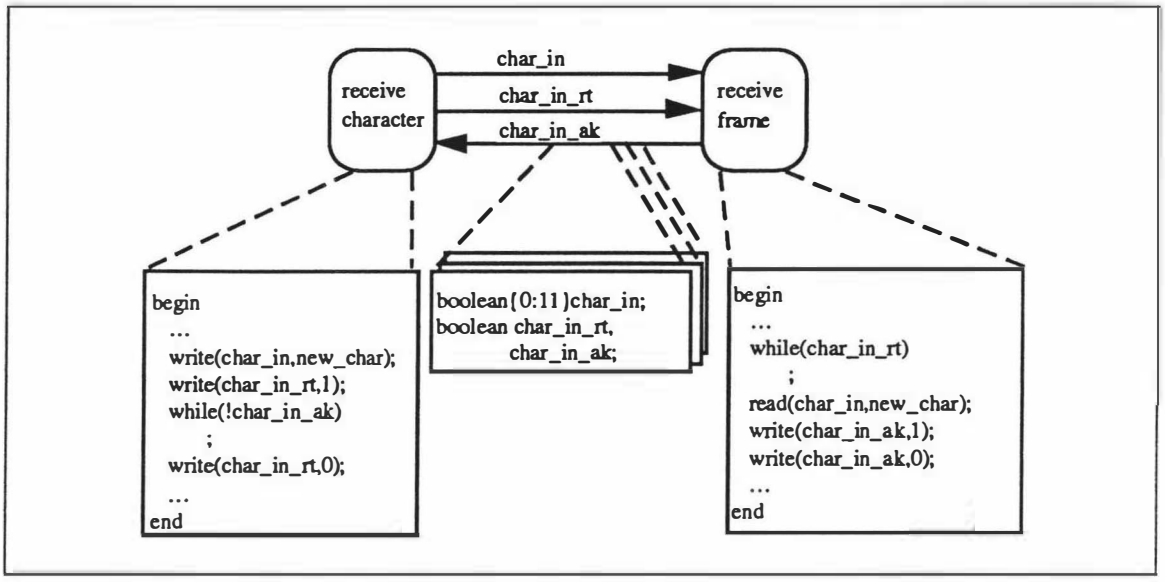


Figure 4.5 Definitions required to define data transfers between processes not using a communications protocol

Group Flows

As the notation just described restricts the use of an automatically supplied data-driven protocol to intra-design communication, designers were obliged to supply a significant number of synchronisation signals for off-chip communication. Figure 4.6 shows part of the data flow diagram which is associated with writing a received frame into an external memory by the process send frame external memory.

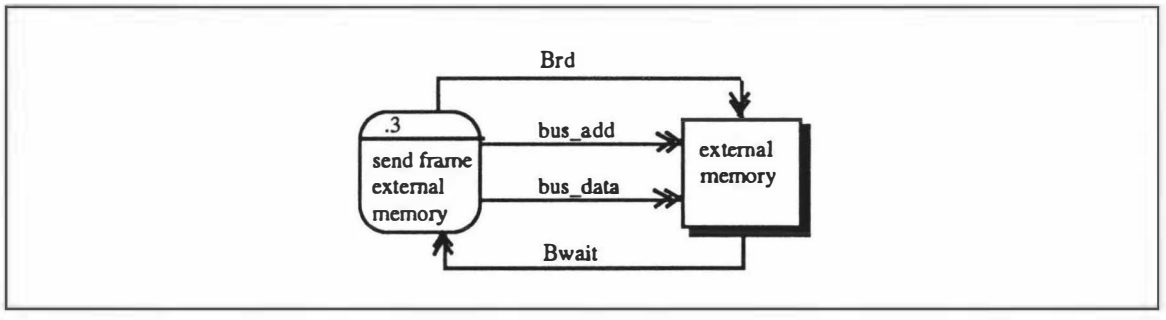


Figure 4.6 Continuous flows required for communication with external memory in Ethernet example

The flows Brd and Bwait are responsible for synchronising the transfer of data on the flow called bus_data. Showing these three flows in full clutters the DFD, without adding to the architectural overview which this top level diagram is intended to provide. What is important at this level of abstraction is that data is transferred between the send frame external memory process and the external memory external entity. To allow this information to be condensed and shown as a single flow, a third flow type, the group flow representing two or more discrete or continuous flows, was introduced to the notation (see Figure 4.7). Groups flows have also been found useful in

several of the other design exercises in circumstances where multiple flows carrying related data appear between two objects.

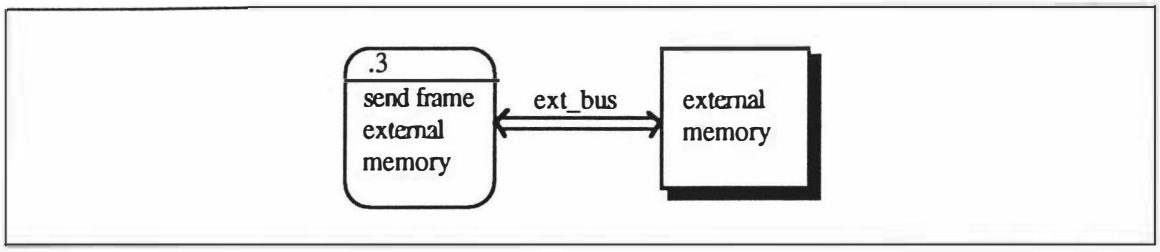


Figure 4.7 Group used to hide continuous flows used for communication with external memory

Store Flows

Figure 4.3, the initial representation for the Ethernet receiver, follows conventional, data-flow practice of using a normal data flow to represent communication with a data store. This informal representation does not include any component for specifying the location of the data being accessed. Less obviously, it also omits the administrative information which locks stores when multiple accesses are initiated simultaneously. The discrete data flows used by PICSIL for interprocess communication do not include any form of addressing component and to include one would overload discrete flows for normal (interprocess communication) use. A concise representation for a channel of communication with a data store was clearly necessary.

An elegant representation of the general case, communication between one data store and several processes, was achieved after several attempts. Figure 4.8 shows the first approach considered, in which each process which accessed a particular data store had a copy of the data store (represented by a named, low, wide rectangle) shown attached to it.

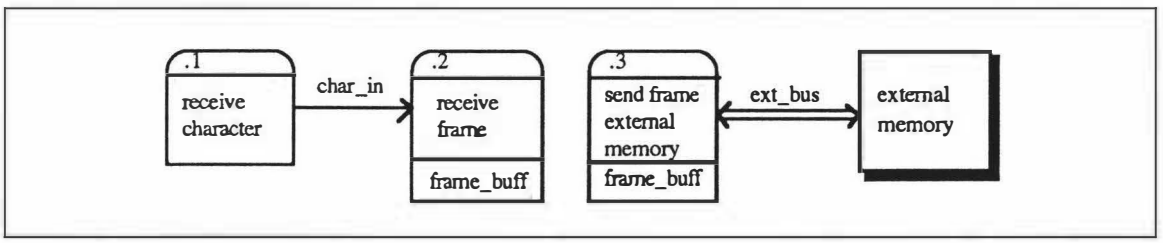


Figure 4.8 Data stores attached to processes

This approach provided a poor representation when many processes require access to the same data store. For example in Figure 4.8 the processes receive frame and send frame external memory both access the data store frame_buff. In omitting any visual link between processes accessing a common store, this notation de-emphasises an important relationship between them.

The second approach was to attach special processes known as data-store-access-processes (DSAPs) to data stores. A DSAP was created for each type of operation performed on a data store (see Figure 4.9). Although this representation was adequate for the Ethernet receiver, it was found confusing for other designs where more than two DSAPs existed and had to be stacked on top of one another. By contrast with the previous representation, which concealed relationships, this overemphasised them by approximating all processes communicating with a data store. Also, when multiple DSAPs were attached to a data store the diagrams became very congested around the data store.

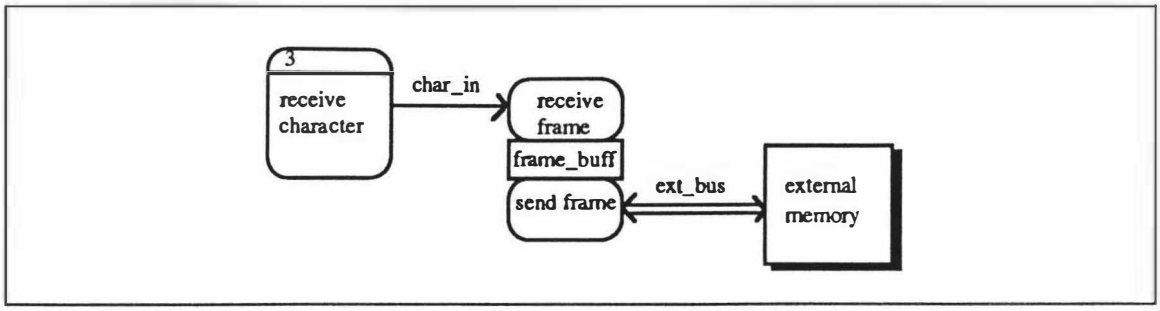


Figure 4.9 Two processes attached to the data store frame_buff

Figure 4.10 shows the final approach considered, and the one adopted. Here the store is shown as a separate object in the diagram, with a fourth type of flow, the store flow (represented by a heavy arrow), representing both the addressing and data transfer operations. Note that the arrow heads show the direction of data transfer; flow direction of addresses was not felt important enough to warrant explicit representation.

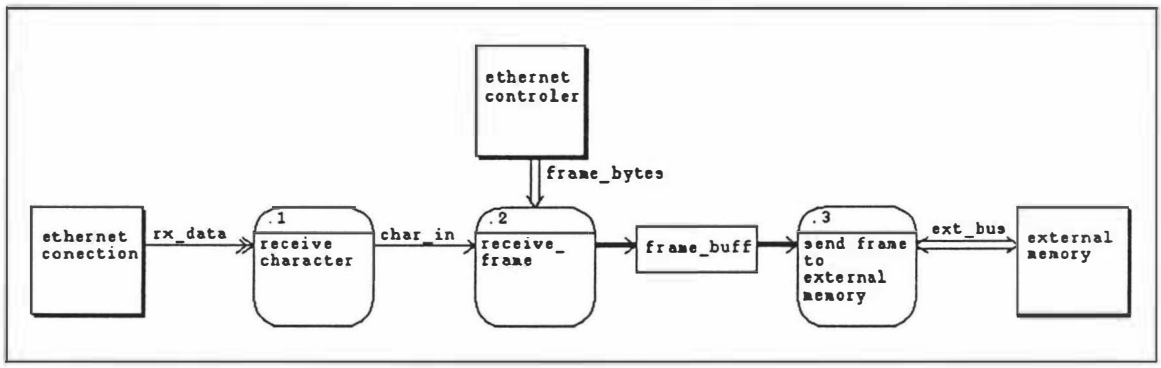


Figure 4.10 PICSIL representation of Ethernet receiver excluding control aspects

Development of the Controller

At this stage of PICSIL's development, the basic form of the Data Flow Diagrams had been determined. Several designs, including the partial Ethernet Receiver, had been represented in the notation, and it was generally satisfactory for most of them.

Some areas required further work, however. In particular, the pure DFD notation being employed was unable to activate and deactivate individual processes when stimulated by external or internal events. Individual components of a process could be controlled using *if-else* and *while* statements, but this approach buries high-level functionality in low-level code. It would be preferable to allow process activation and deactivation to be an system-level function, and PICSIL was augmented to allow this. Of course, the lower-level control components (the *if* and *while* statements) were left intact.

In the full Ethernet receiver, this type of control is necessary, because when the collision detect signal, CCT, goes false while a packet is being received, or an invalid parity byte is found in the received frame, then the frame is discarded and not sent to the external memory. To provide for this, the receive frame process must be able to be deactivated.

Accordingly, the development of PICSIL's representation for controllers will be described by working through a design for the full Ethernet receiver, noting, as before, that the language developments were applied in parallel to several of the other designs (i.e. image processing

device, MasseyNet device and the MC6850). It is appropriate to start with a description of the complete receiver. Part of the specification of the Ethernet receiver was omitted from the initial representation in Figure 4.3.

The PICSIL notation was extended to allow the definition of a controller based on extended DFD methodologies introduced by Hatley and Pirbhai (1987) and Ward (1986). Figure 4.11 shows a PICSIL representation of the receiver. It includes two extra symbols to allow control to be specified: event flows (represented by dashed lines with arrow heads) and control processes (represented by tall, thin rectangles) which form an interface to a single controller for the diagram. Figure 4.12 shows a definition of the controller in the original notation. The definition of a controller is made up of three parts: a state transition diagram (Figure 4.12(a)), a set of event action equations (Figure 4.12(b)) and a process activation table (Figure 4.12(c)).

Discrete and Continuous Event Flows

Initially a single type of event flow was used. It represented a single bit asynchronous channel. However in Figure 4.11 two different types of event can be identified. The first type is used to convey a data condition existing over a period of time, as occurs on the event flows new_frame, bhold, CCT, rx_enable, and Back. For example a new frame can be received while the event flow CCT is TRUE. If however CCT is set to FALSE while a frame is being received, it should be discarded.

The second type of event reports the occurrence of data conditions at a discrete point in time as occurs on the event flows frame_ok, frame_fault, and DMA_done. For example the DMA_done event flow is used by the process send frame to external memory to report that a DMA transfer has finished. Initially, these discrete data conditions were reported to the controller by setting the event flow to TRUE, followed by FALSE (e.g. write(DMA_done, TRUE); write(DMA_done, FALSE);). However it cannot be assumed that the controller will execute infinitely quickly, and hence it could not be guaranteed that the controller would receive the notification of the DMA_done event, as no synchronisation was provided. This led to the inclusion of two event flow types (discrete and continuous) in the notation.

Continuous event flows, shown as a dashed line with a double arrowhead, are used to report conditions which exist over a period of time, and represent a single bit asynchronous I/O channel.

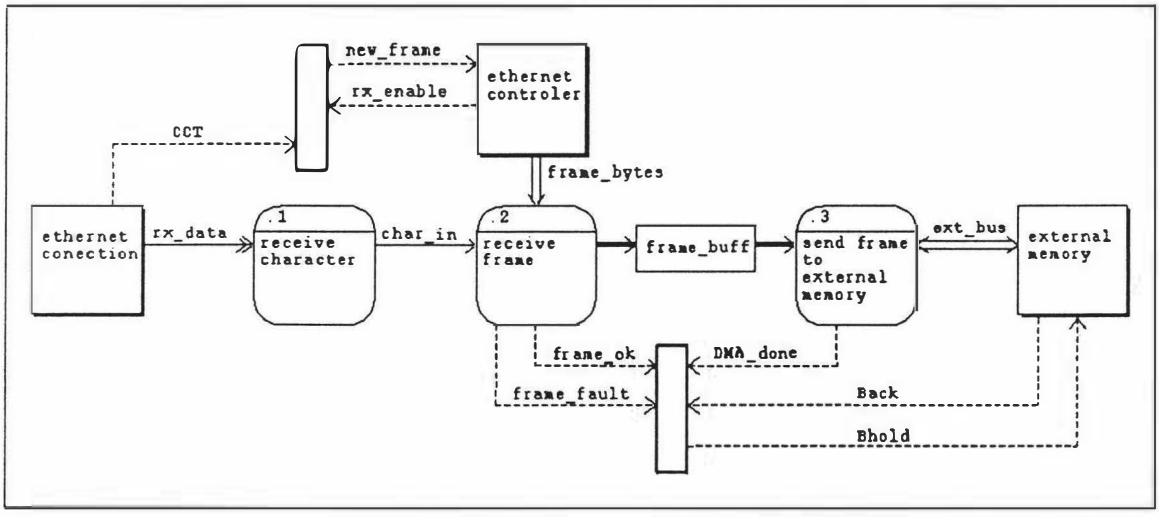


Figure 4.11 Initial PICSIL representation of ethernet controller including control aspects

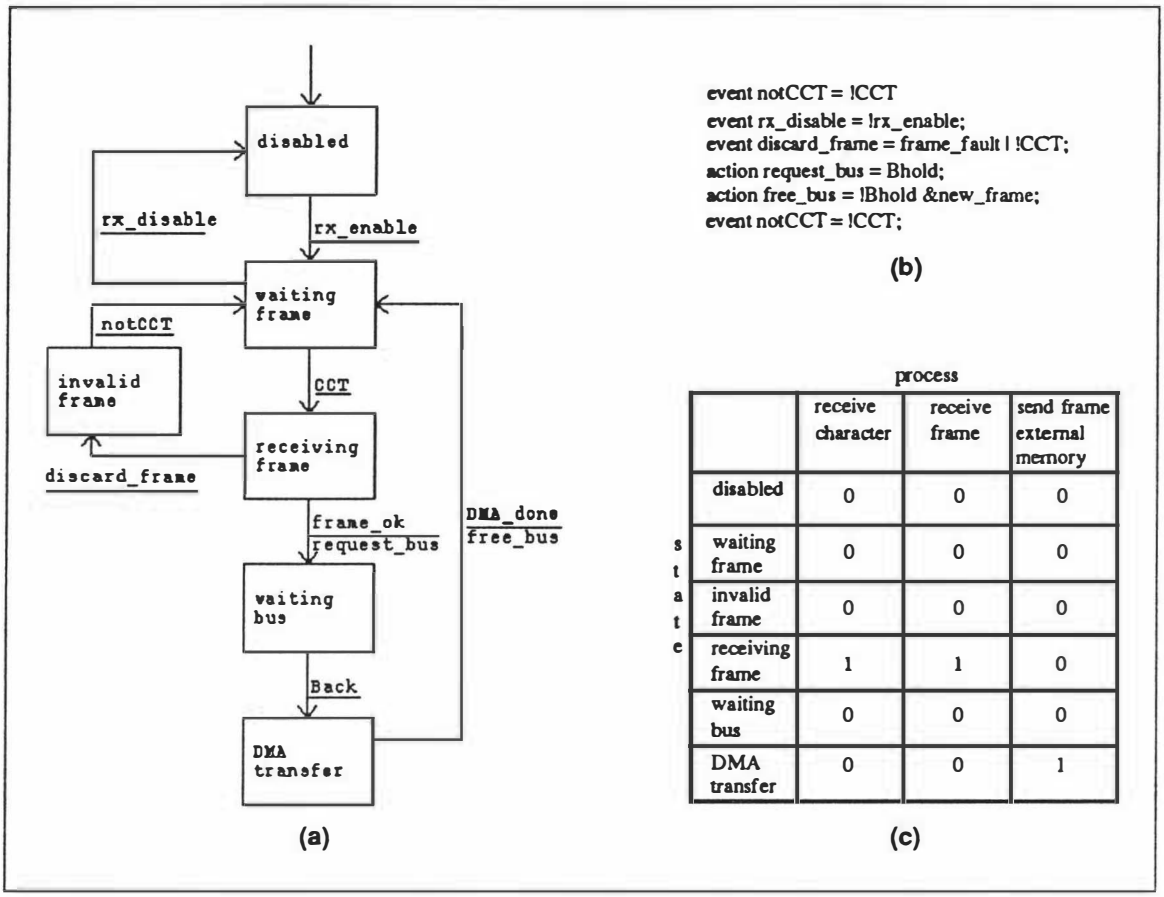


Figure 4.12 Initial representation of the controller definition for the ethernet receiver including (a) state transition diagram, (b) event action equations and (c) process activation table

Discrete event flows, shown as a dashed lines with a single arrow head, are used to report the occurrence of data conditions in a system which occur at discrete points in time. A report statement in a primitive process's data dictionary entry (e.g. report(DMA_done);) is used to report a discrete data condition to a controller which remembers it until it executes a state change conditional upon the event. Figure 4.13 shows the PICSIL representation of the receiver using both types of event flow.

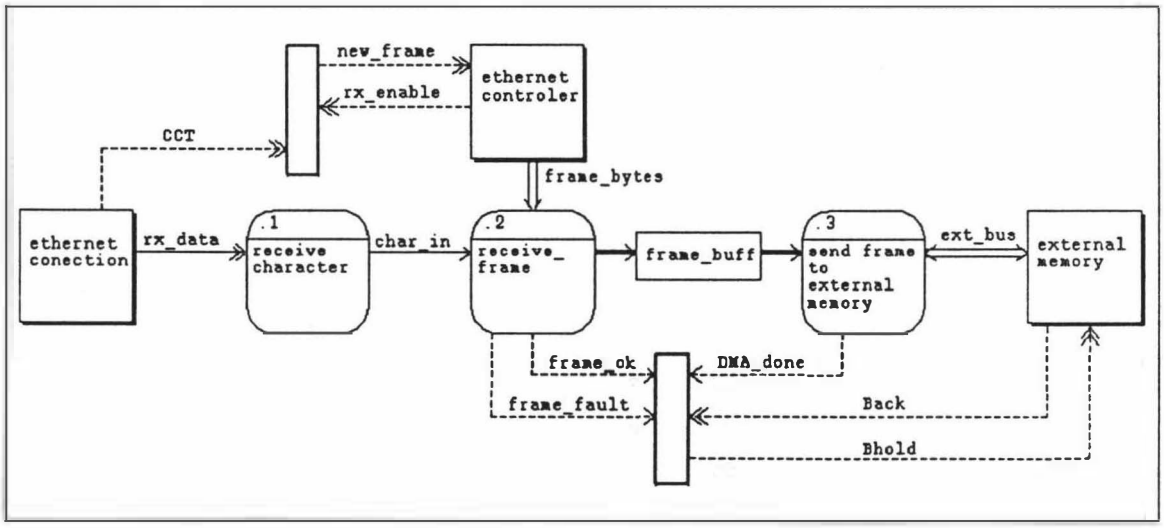


Figure 4.13 PICSIL representation of ethernet controller including control aspects

Controller Definitions

The second symbol included in the PICSIL notation to allow high level control to be specified was the control process, representing an interface to a controller for the diagram.

The representation of the state transition diagram (see Figure 4.12a) is based on those used in the Hatley and Pirbhai (1987) notation, and comprises states, represented by named rectangles, and transitions, represented by labelled arcs. The label on a state transition may contain an event causing the transition (shown above the horizontal line in the label) and/or action performed on the transition (shown below the horizontal line in the label). Synchronous transitions can be generated merely by leaving the corresponding arc unlabelled.

For example, on the transition from the DMA_transfer state, DMA_done represents the event and free_bus represents the action. When the system is initialised, the disabled state is entered (indicated by the transition attached to a single state.

When the controller is in the receivingframe state, the processes receive character and receive_frame are activated, allowing a new frame to be received. If a frame_ok event is reported, the controller makes a request for the external bus by setting Bhold TRUE. When Back becomes TRUE the DMA transfer state is entered, activating the send frame to external memory process to allow the contents of the frame to be transferred to external memory. If CCT goes FALSE or a frame_fault event occurs while the controller is in the receive frame state, a transition to the waiting frame state occurs, terminating the execution of the processes receive character and receive frame.

To accommodate the capture of state transition diagrams using an editor developed in parallel with the PICSIL language, a number of changes in appearance of the state transition diagram have been required. First, the editor developed requires that a flow (or transition in this case) be connected to an object at both ends. However in Figure 4.12 the transition into the initial state (i.e. disabled) has an object connected at only one end. This led to the development of a notational entry state represented by a triangle (see Figure 4.14) which the controller enters on being reset.

The second change in appearance of the state transition diagram is associated with the labels on the state transitions. Using the original notation two text fields were required to enter an event and an action. However, the editor developed only allowed a single text field to be associated with a state transition. To meet this requirement the notation was modified so that the event and action were separated by a '/' character instead of a horizontal line (see Figure 4.14a).

In addition to a state transition diagram, a process activation table is used in a controller's definition to define the execution state of each of the processes in a diagram for each state in the state transition diagram. In the original representation of the process activation table (see Figure 4.12c), entries in the table with a "1" represented activated processes while "0" entries represented deactivated processes. Under this arrangement, activated processes execute repeatedly, automatically restarting themselves on completion.

In some cases it was found that instead of a process executing continuously while the controller was in a particular state, only one execution was required. For example the process send frame to external memory should only be activated once for each time the controller is in the DMA_transfer state. When the PICSIL notation was applied to other designs (the ACIA and the Kalman filter) the ability to activate processes once only in a particular state was also required. This led to the inclusion of a third process activation type allowing processes to be activated once only in a particular state. At the same time new symbols were introduced to represent the three types of

process activation: |▪| for deactivated, |▶| for activated once, and ⟳ for activated continuously[1].

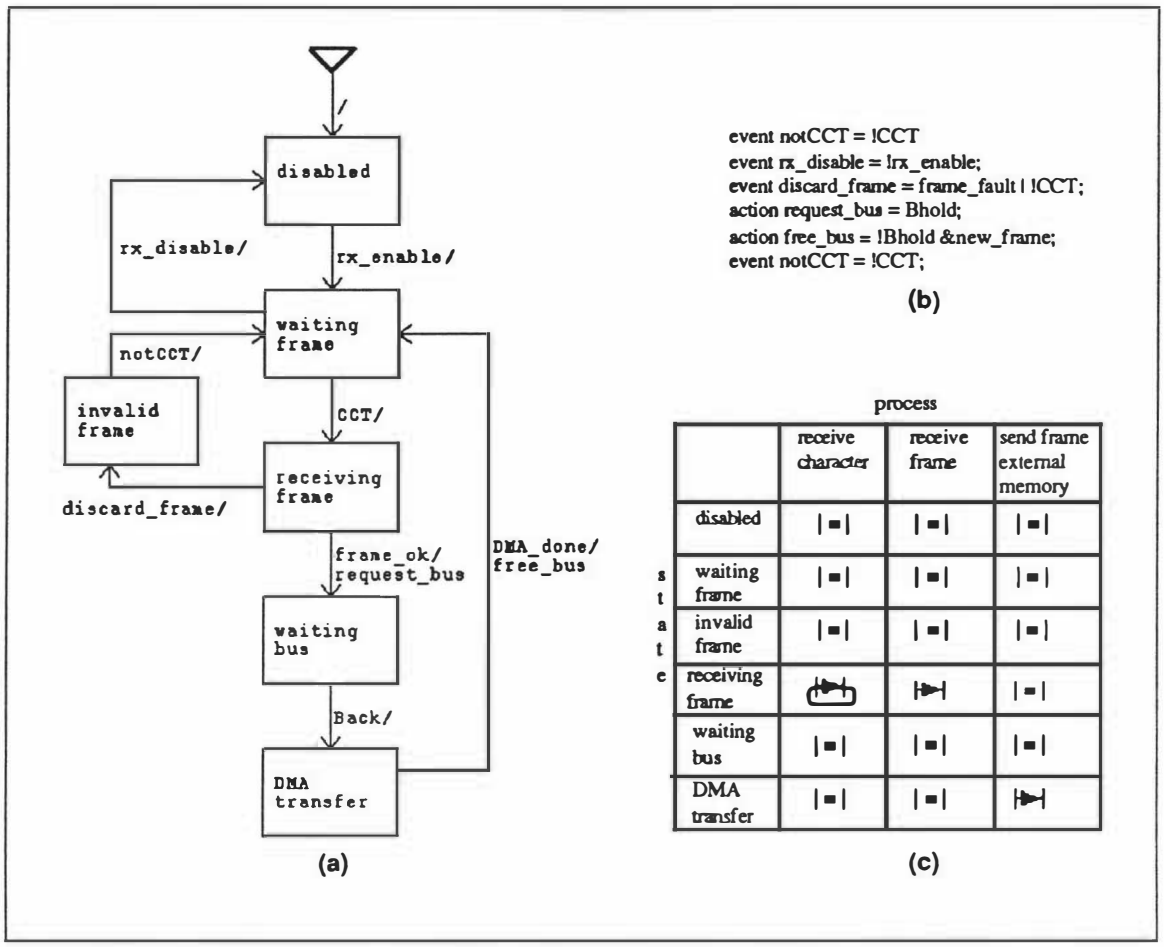


Figure 4.14 Definition of controller in Figure 4.13 including (a) state transition diagram with entry state, (b) event action equations and (c) process activation table

Inclusion of the controller into the design had minimal effects on the DFD/algorithmic aspects of a design. Other than the inclusion of the control process and a number of event flows (e.g. frame_ok), the only other changes required to the PICSIL representation shown in Figure 4.10 (i.e. excluding high level control) was the inclusion of report statements into the processes receive frame and send frame to external memory. The use of a separate controller allows the data processing functions of a system to be defined almost totally independently of the high level control aspects, reducing the amount of detail the designer has to deal with at any one time.

Figure 4.15 shows the process definition for the process receive frame. The four report statements are used to indicate to the controller that the frame has been accepted correctly (i.e. report(frame_ok)) or that the frame is to be discarded (i.e. report(frame_fault);). It is then left to the controller to terminate the process if required.

---

[1] Because of time constraints on this project, it has not been possible to implement features in the editor to allow graphical capture of process activation tables. Instead a textual form of the process activation table has been defined (see Chapter 5) allowing the process activations for each state to be specified.

```
process receive_frame;
seqbegin
   boolean{0:8}    buff_start,
                   new_char,
                   length,
                   counter,
                   parity;
   boolean{0:15}   destination;
   parity = 0xFF;
   receive(char_in, new_char);
   while(new_char != sfd)   /*discard preamble*/
      receive(char_in, new_char);
   receive(char_in, destination{15:8});
   parity = parity ^ destination{15:8};
   receive(char_in, destination{7:0});
   parity = parity ^ destination{7:0};
            /** check if frame is for this node **/
   if (destination != controller_add)
      report(frame_fault); /** not for this node **/
   receive(char_in, new_char); /**source add 1**/
   parity = parity ^ new_char;
   stwrite(frame_buff[buff_start],new_char);
   buff_start++;
   receive(char_in,new_char); /**source add 2**/
   parity = parity ^ new_char;
   stwrite(frame_buff[buff_start],new_char);
   buff_start++;
   receive(char_in, length);              /**length**/
   parity = parity ^ length;
   stwrite(frame_buff[buff_start],length);
   buff_start++;
   counter = 1;
   while (counter <= length) seqbegin
      receive(char_in,new_char); /** data byte **/
      parity = parity ^ parity;
      stwrite(frame_buff[buff_start],new_char);
      buff_start++; counter++;
   seqend
   receive(char_in,new_char) ;  /** parity byte**/
   parity = parity ^ new_char;
                              /** Do parity check **/
   if (parity != new_char)
      report(frame_fault);          /**parity error**/
   receive(char_in,new_char);     /**frame end**/
   if (new_char == eof)
      report(frame_ok);          /** frame received
                                          correctly**/
   else
      report(frame_fault);
seqend;
```

Figure 4.15 PICSIL Process definition for process receive frame in Ethernet receiver

Comparison with conventional HDL representation

The development of the controller seemed to exhaust the Ethernet receiver design exercise as a source of inspiration for language design. The next activity was to evaluate the language so developed.

Aptitude of Graphics for Architectural Expression

Before dealing with specific areas in which PICSIL's approach to hardware definition differs from conventional text-based HDLs (exemplified here by HardwareC) it is appropriate to consider the more general, graphical advantages of PICSIL's top-level vocabulary.

Designing with pictures on paper allows a designer to produce something very close to his or her mental model of the target device's architecture. However, designers using paper often take advantage of their freedom to modify or add to the visual vocabulary when expressing particular requirements, and successfully rely on the intelligence of their colleagues to understand the resulting idiomatic notation.

However personal experience has shown that too large a vocabulary in a computer-based graphical interface is confusing. Icons which would make sense in a complete design, where their contextual clues to their meaning are readily available, prove difficult to identify when presented out of context in a menu. Consequently, new icons were added only reluctantly to the PICSIL notation. Despite the limited vocabulary, the resulting PICSIL notation has been found to be sufficiently powerful to represent designs from wide variety of application domains. Furthermore, provided meaningful names are given to named objects (i.e. processes, data flows and data stores), the resulting language provides a natural representation of the use of a target device's architecture.

Earlier, it was stated that "it was possible to devote significant effort to achieving a level of abstraction in the language which represents a balance between information content and design complexity." If this goal were to be successfully achieved, a PICSIL design would be easier to produce, and more comprehensible than a design using a conventional HDL. To test this - though with the designer doing the testing, objectivity was clearly impossible - the PICSIL representation for the Ethernet controller was compared with its HardwareC representation (Coelho, 1990). Part of the HardwareC representation (approximately 1/3) is shown in Figure 4.17 .

This description contains a block definition (shown in the right most column) instantiating the components making up the Ethernet receiver, and a definition of one of its components, the rcvd_frame process. The components that make up the description are similar to the PICSIL representation (see Figure 4.10) except that the buffering stage is added before the rcvd_frame process, and there is no explicit controller. Because of the absence of a controller, extra signals between the components are required to allow them to communicate state information to each other.

Clarity of Relationships

The top level PICSIL representation gives a complete high level description of the design, and explicitly shows the components that make up the Ethernet controller and their inter-relationships in a clear and concise manner.

The top level HardwareC representation also explicitly states the components that make up the design; however it is left to the designer to infer their inter-relationships. In other than trivial cases this means drawing diagrams. The direction of data flow between the components, although an important aspect of the device, is not included in the HardwareC definition, and this information can be determined only by examining the lower levels of the description.

At the top level, the HardwareC description contains a declaration of each I/O and local communications channel, which is omitted from the equivalent level of the PICSIL description. At the level of detail presented in this top level abstraction these declarations do not add meaning to the "overall picture" being presented, so have been moved down one level in the PICSIL definition.

Because of the textual nature of the HardwareC representation and the information density of the resulting design, a significant portion of the designer's ingenuity is side-tracked into producing a syntactically correct design. For example the exclusion of a parameter (e.g. eof in the from the rcvd_frame instantiation) would not be easy to detect from the block description. In contrast, the graphical nature of PICSIL makes this type of error very easy to detect and allows the designer to concentrate on producing correct semantics.

Separation of Functions

While there are statements for describing behaviour in both the PICSIL and HardwareC languages, the receive_frame process in PICSIL and rcvd_frame process in HardwareC are very different in appearance. As HardwareC does not have constructs to model high level control separately, the process definition has to incorporate both the data processing and control functions. As a consequence, the HardwareC representation is divided into a series of states using a switch statement to select the current state. Each state has a small number of statements, which receive the next character from the buffer, process it and determine the next state. If a condition has occurred within or outside the process, the processing of the frame can be terminated if necessary.

By contrast, the PICSIL representation allows the data-processing function of the process to be considered independently of any control aspects, so that the designer can concentrate on defining the semantics of the process. The controller/data processing interface is minimal, and the resulting description is more readable.

This example has given rise to a number of points of comparison which, in general, also relate to the other design exercises described in this chapter. In order to avoid repetition they are called PICSIL Positive Points :

- Graphical designs can express system level ideas effectively and concisely;
- PICSIL diagrams explicitly state inter-relationships between components in the diagram;
- The directions of data flows are explicitly stated on PICSIL diagrams;
- The graphical nature of PICSIL prevents syntax errors;
- PICSIL separates control and data processing sections of a design.

While these Positive Points have been established in this comparison with HardwareC, they also generally hold true when PICSIL designs are compared with the VHDL and Verilog descriptions.

4.2 Traffic Light Controller: a control-only design

The traffic light controller (Mead and Conway, 1980) is a simple control-oriented example. Consider a busy highway which is intersected by a little-used farm road. A controller is designed to control a set of traffic lights at this intersection. The highway traffic lights remain green until a car is detected on the farm road, in which case they cycle through yellow to red and then the farm road lights go green. The farmroad lights remain green only while cars are detected on the farmroad, but never longer than determined by a short delay timer. The farmroad lights then cycle through yellow to red and the highway lights turn green again. The highway lights are not interruptable again until some longer period of time, determined by a second timer, has lapsed. Figure 4.16 shows a finite state machine to control the lights at the intersection, where car is used to indicate a car on the side road, and TimerS and TimerL represent the two timers.

Figure 4.18 shows the top level PICSIL diagram for the device and Figure 4.19 shows the definition of the controller.

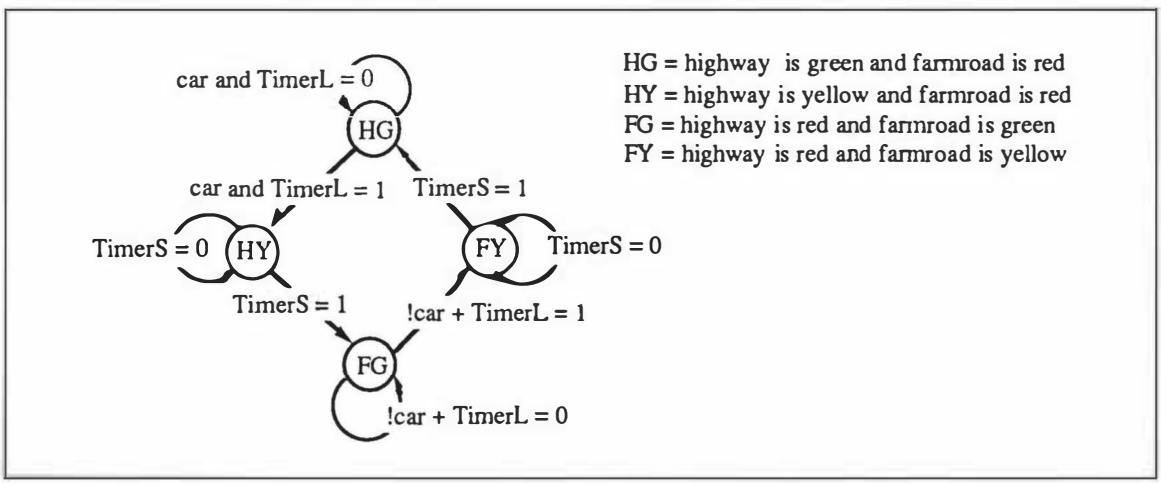


Figure 4.16 Traffic light controller state diagram (after Mead and Conway, 1980)

```
#define START       0
#define PREAMBLE    1
#define SFD         2
#define DEST1       3
#define DEST2       4
#define SOURCE1     5
#define SOURCE2     6
#define LENGTH      7
#define DATA        8
#define DATAEND     9
#define CRC         10
#define EOF         11
#define END         12

process rcvd_frame (preamble, sfd, eof, etherRcvd, recBeg,
                    controllerAddr,rcvdInh, bout, boutsync, enableFrame,
                    receptionOK)

in port preamble[8];          /* preamble code */
in port sfd[8];               /* start of frame delimiter code */
in port eof[8];               /* end of frame delimiter */
in channel etherRcvd[8];      /* frame being received */
in port rcvdInh;              /* inhibits reception of frame */
in port controllerAddr[16];   /* controller address */
in port enableFrame;          /* valid frame being received */


out port recBeg;          /* indicates frame being received */
out port bout[8];         /* byte received */
out port boutsync;        /* a one in this signal indicates bout valid */
out port receptionOK;     /* frame received without errors */
[
   static destination[16]; /* temporarily stores the destination address */
   static length[8], counter[8]; /* stores and counts the length of a frame */
   static b[8];
   static parity[8];      /* stores the parity computed for a frame */
   static state[5];       /* state of de-framing protocol */
   static recok;          /* stores the status of frame reception */

   <
      write receptionOK = 0;
      oad recok = 0;
      oad state = PREAMBLE;
      oad parity = 0xff;
      write boutsync = 0;
      write recBeg = 0;
      hile (rcvdInh);       /* wait until frame reception is enabled */
      hile (!enableFrame);  /* wait for a frame */
   >

   while (enableFrame) [     /* while a frame is being received */
         /* wait for a new data or an interruption in the reception */
      while (enableFrame & !(msgwait(etherRcvd)));

         /* is there a new data available ? */
      if (msgwait(etherRcvd)) [
         load b = receive(etherRcvd); /* receives data */
         witch (state) {
         case PREAMBLE    :
            load state = PREAMBLE; /* discard preamble bytes */
            if (b == sfd)      /* if sfd, receives  destination address */
               load state = DEST1;
            write recBeg = 0;
            break;
         case DEST1       : write recBeg = 1;        /* frame begins */
            write bout=b;
            oad destination[15:8] = b;
            oad state = DEST2;
            parity = parity ^ b;
            write boutsync = 1;
            break;
         case DEST2       : write bout=b;
            oad destination[7:0] = b;
            oad state = SOURCE1;
            parity = parity ^ b;
            write boutsync = 1;
            break;
         case SOURCE1     : parity = parity ^ b;
             /* if destination address is not the same as
              * the controller address, disregard the rest
              * of the frame */
            f (destination == controllerAddr) {
               write boutsync = 1;
               write bout=b;
               load state = SOURCE2;
            }
            else
               load state = END;
            break;
         case SOURCE2     : write bout=b;          /* receive source address */
            load state = LENGTH;
            parity = parity ^ b;
            write boutsync = 1;
            break;
         case LENGTH      : write bout=b;          /* receive length of frame */
            load length = b;
            load counter = 1;
            parity = parity ^ b;
            if (b == 0)
               oad state = CRC;
            else
               load state = DATA;
            write boutsync = 1;
            break;
         case DATA        : write bout=b;    /* receive data */
            parity = parity ^ b;
            if (length == counter)
               load state = CRC;
            else
               counter++;
            write boutsync = 1;
            break;
         case CRC             : load state = EOF;     /* check parity */
            f (parity == b)
               oad recok = 1;
            else
               oad recok = 0;
            break;
         case EOF         : load state = END;
            if (b == eof)   /* check eof delimiter */
               write receptionOK = recok;
            else {
               write receptionOK = 0;
               oad recok = 0;
            }
            write recBeg = 0;    /* frame ends */
            break;
         case END         : /* disregard data if this portion is executed.
            * This means a frame has already being received
            * and any other incoming data should not be
            * considered or destination address is different
            * from the controller address, and the frame
            * should not be de-framed */
            wwrite receptionOK = recok;
            break;
         }
      ]
    /* disables bout */
    write boutsync = 0;
     ]
  ]
```

```
block receiver(rx_data,
CCT,rx_disable,start_add,bus_add,bus_data,Brd,Bwait,
   Back,Bhold, preamble, sfd, eof, controller_addr, new_frame)

in port rx_data[8];
in port CCT;
in port rx_disable;
in port start_add[16];
in port bus_add[16];
in port bus_data[16];
in port Brd;
out port Bwait;
out port Back;
in port Bhold;
in port preamble[8];
in port sfd[8];
in port eof[8];
in port controller_addr[16];
out port new_frame;
<
   boolean bytercvd[8], pulsercvd;
   boolean receiving;
   channel etherRcvd[8];
   boolean enableFrame;
   boolean recBeg;
   boolean bout[8], boutsync;
   boolean receptionOK;

rcvd_bit (rx_data, CCT, bytercvd, pulsercvd, receiving);

 rcvd_buffer (bytercvd, pulsercvd, etherRcvd, rx_disable, receiving);


 rcvd_frame (preamble, sfd, eof, etherRcvd, recBeg, controller_addr,
         rx_disable, bout, boutsync, enableFrame, receptionOK);

 DMA_rcvd (bout, boutsync,bus_add,bus_data,Brd,Bwait, receptionOK,
         start_add, recBeg,new_frame,Back,Bhold);
>
```

Figure 4.17 Part of HardwareC description for Ethernet receiver

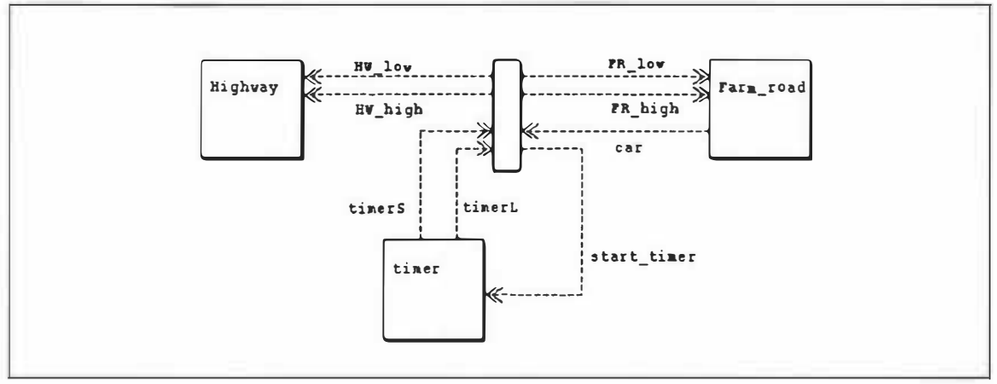


Figure 4.18 Top level PICSIL diagram for traffic light controller

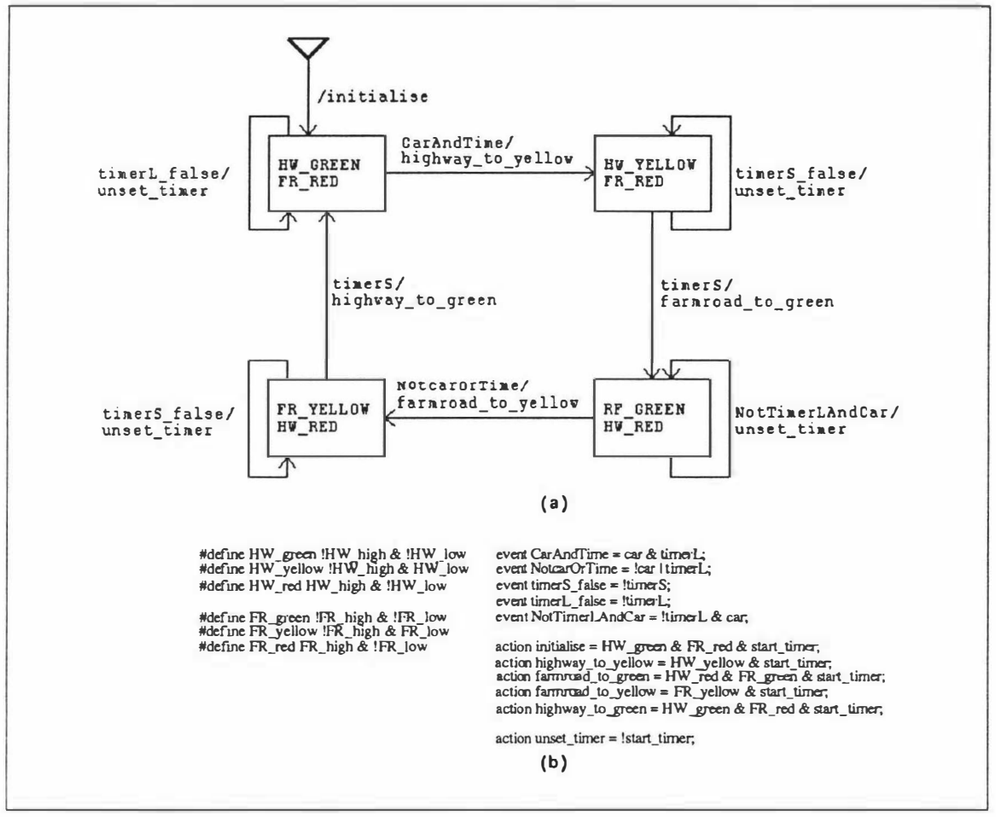


Figure 4.19 (a) State transition diagram and
(b) event/action equations for controller definition for traffic light controller

In this example, the top level data flow diagram (see Figure 4.18) defines only the inputs and outputs to the system, which could be represented more effectively using an alternative representation. However this example was not developed until the later stages of the research discussed in this thesis, by which time the present version of PICSIL had been finalised.

The edge detector design exercise is also represented using a single controller, and potential was seen to represent a number of other problems in the same way. Hence future versions of

PICSIL need to include more appropriate constructs to define the I/Os for designs consisting of a single controller.

Comparison with conventional HDL representation

Figure 4.20 shows the representation of the traffic light controller in VHDL using a single process.

```
package traffic_light is
  subtype Light is BIT_VECTOR(0 to 1);
  constant Red    : Light := B"00";
  constant Green  : Light := B"01";
  constant Yellow : Light := B"10";
end traffic_light;

use work.traffic_light.all;
entity tlc is
  port (sidecar        : in bit;
        TimerL         : in bit;
        TimerS         : in bit;
        hili           : out Light := Green;
        sideli         : out Light := Red
        StartTimer     : out bit := '1');

end tlc;
architecture behavior of tlc is
begin
  process
  begin
      -- highway is green, sidestreet is red
      --
      StartTimer <= '0'
      wait until (TimerS = '1' and sidecar = '1');
      start_timer = '1';
      hili <= Yellow;
      --
      -- highway is yellow, sidestreet is red
      start_timer = '0';
      wait until (TimerL = '1');
      start_timer = '1';
      hili <= Red;
      sideli <= Green;
      --
       -- highway is red, sidestreet is green
      start_timer <= '0';
      wait (sidecar = '0' or TimerS = '0');
      start_timer <= '1';
      sideli <= Yellow;
      --
      -- highway is red, sidestreet is yellow
      start_timer = '0';
      wait(TimerL = '1');
      start_timer = '1';
      sideli <= Red;
      hili <= Green;
  end process;
end behavior;
```

Figure 4.20 VHDL representation of traffic light controller

Both the VHDL entity tlc in Figure 4.20 and the top data flow diagram in Figure 4.18 define all the inputs and outputs to the traffic light controller. For this example the VHDL description provides a more concise representation than does the PICSIL representation.

In describing the functionality of the controller, the PICSIL representation is similar to the FSM shown in Figure 4.16 with each of the states and the possible state transitions being shown explicitly. The VHDL description, however, uses a linear sequence of instructions with the possible states and their transitions having to be inferred.

4.3 Packet Switch: Routers and Structured Decomposition

The third and final design exercise is a simplified version of the MasseyNet (Lyons and McGregor, 1990) device, called a packet switch. The example illustrates the use of top down design partitioning, and describes the evolution of elements and routers.

The packet switch directs packets of data (in the format shown in Figure 4.21) arriving at one of its four input ports to one of its four output ports. The input virtual circuit number and the input port are used to index a virtual circuit table to generate an output virtual circuit number and output port. Apart from the virtual circuit number the rest of the packet remains unchanged when it is output.

- 1 byte : STB character (start of packet character)
- 1 byte : VC number (virtual circuit number packet belongs to)
- 1 byte : STX character (data follows this character)
- n bytes : data in packet
- 1 byte : ETB character (represents end of packet)

Figure 4.21 Format of packets input and out by packet switch

If the outport port number found in the routing table is zero, then the packet is a control packet, and is intended for the nodes switch manager which maintains the routing table. The switch manager can also send packets out of any of the node's output ports.

It is also possible for packets destined for the same output port to arrive at two input ports simultaneously, so a buffer is required on the input. Figure 4.22 shows the initial representation used for the packet switch. Because of its complexity the routing table and switch manager have been omitted.

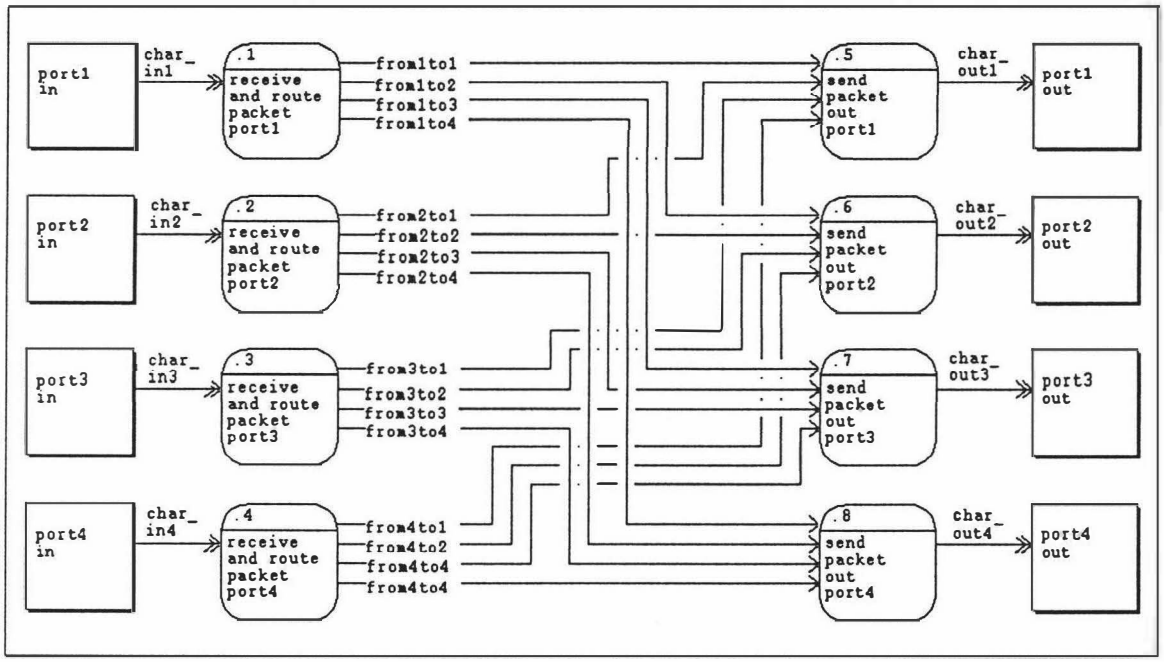


Figure 4.22 Initial PICSIL representation of packet switch

This initial representation is very cluttered as each of the receive and route packet processes has to be able to send data to each of the send packet out port processes, requiring sixteen data flows. If the number of input and/or output ports on the packet switch had been increased, the problem would have been compounded. This representation also imposed an unnecessary level of complexity onto the parts of the process definitions associated with the routing of data from one of the receive and route packet processes to one of the send packet out port processes. Figure 4.23 shows the process definitions for the processes receive and route packet port1 and send packet out port1, both of which require two multiway branch statements.

Routers

While this was the only design exercise which involved m processes sending data to each of n processes, a number of other types of design, particularly in the data communications area,

were seen to pose similar problems. To simplify PICSIL representations involving multiple processes sending data to multiple other processes, the *router* was introduced to the PICSIL vocabulary. Figure 4.24 shows the PICSIL representation of the packet switch using the router.

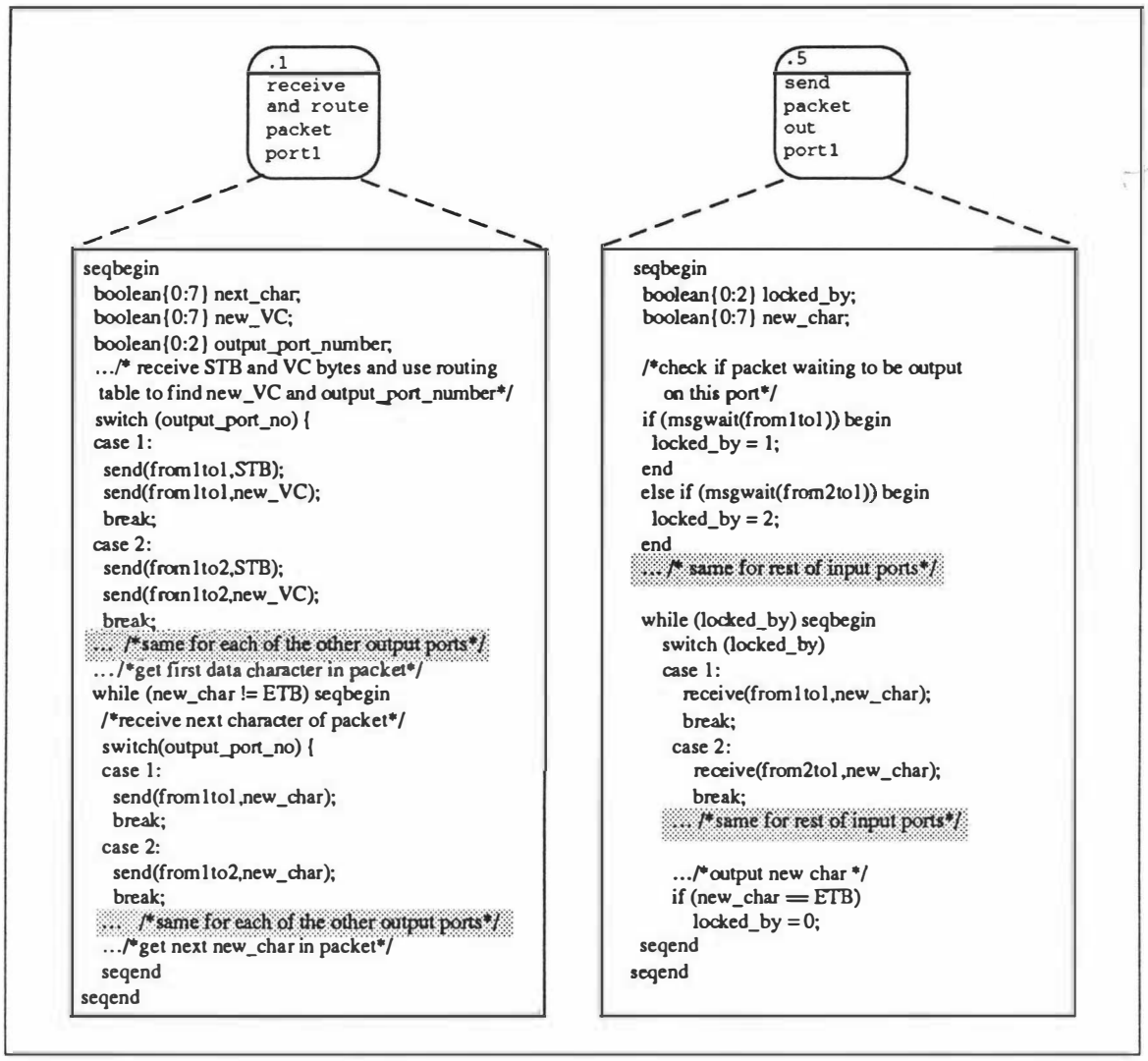


Figure 4.23 Definitions of the processes receive and route packet port1 and send packet out port1

In this representation the number of flows between the two processes has been reduced from sixteen (i.e. $m \times n$) to eight ($m + n$), leaving the diagram significantly less cluttered.

Introducing the router also allowed simplification of both the receive and route packet and receive packet process definitions (see Figure 4.25). In the case of the receive and route packet process, a switch statement is still required so that the appropriate channel through the router can be locked (e.g. lock(from1, to2)). Once the channel has been locked, conventional send statements can be used (e.g. send(from1, next_char)) automatically sending the packet to the correct destination and removing the need for the second switch as required in Figure 4.23.

Another advantage of locking channels through a router is shown in the send packet out port1 process. In the initial representation (see Figure 4.23) the variable locked_by is required, allowing the process to remember which receive and route process it is currently receiving a packet from. The process also has to check each character to determine the end of a packet, so the locked_by variable can be reset, allowing packets to be received from the other receive and route packet processes. With the introduction of the router, the send packet out port1 process is greatly

simplified (see Figure 4.25) as it imports only one data flow and the router ensures that the packets do not become intermingled.

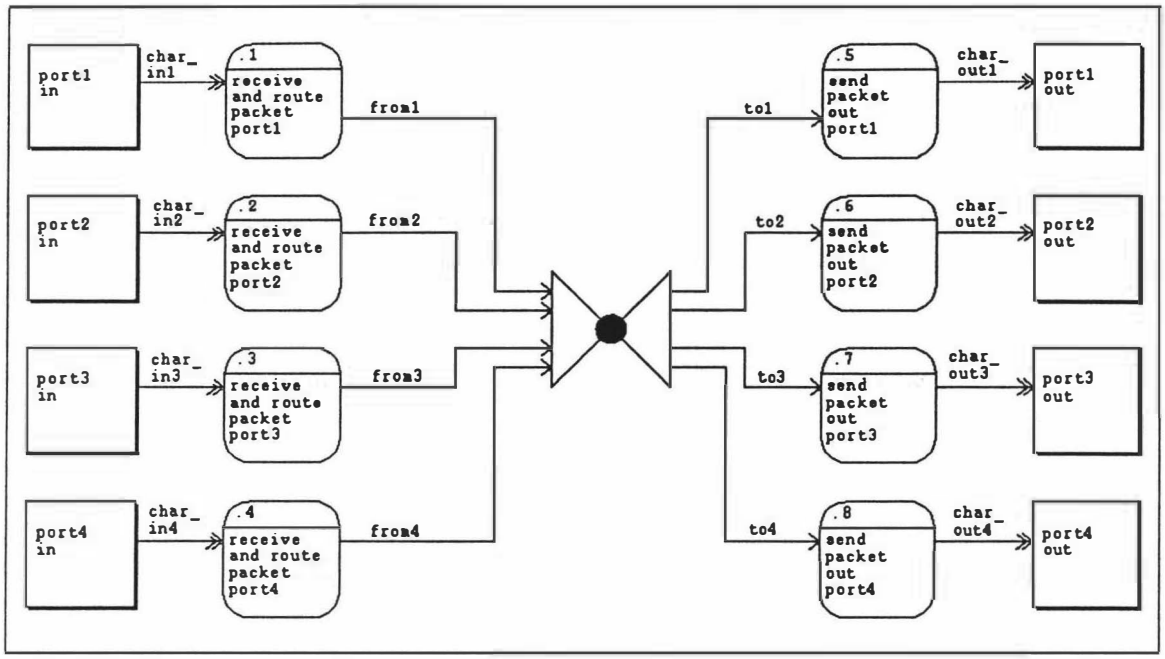


Figure 4.24 Representation of packet switch using router

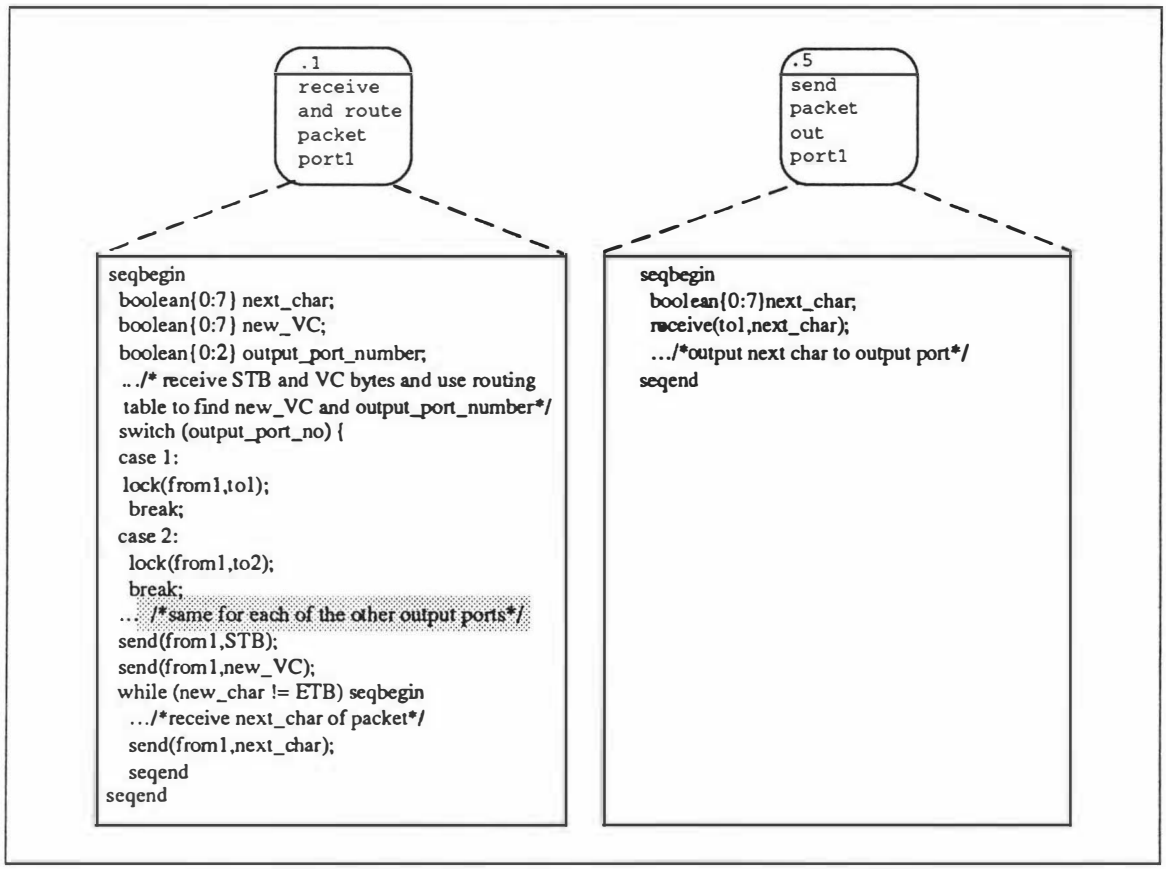


Figure 4.25 Definitions of the processes receive and route packet port1 and send packet out port1 after router added to PICSIL notation

Elements

The packet switch is made up of four identical input processes and four identical output processes, each of which is represented explicitly in the data flow diagram in Figure 4.24. Conventional data flow diagrams do not have any features to exploit this repetition, as a separate definition is required for each process in a set of data flow diagrams (DeMarco, 1978).

Hatley and Pirbhai (1987) have described a notation which allows repeated parts of a design to be defined only once. Figure 4.26 shows a PICSIL diagram for the packet switch in which repeated items are abbreviated according to Hatley and Pirbhai's notation. The number of identical receive and route packet processes can be inferred from the association of the variable n with one of them, and receive and route packet_1 is explicitly represented and decomposed: the reader must assume that the decomposition of the others is identical to that of number 1. The same applies to the send packet out port processes where m defines the number of identical processes. This diagram is now simple enough overall to allow inclusion of representations for the route table (as the datastore route_tble) and the switch manager (as the process process control packets) which were excluded from the initial representations of the packet switch.

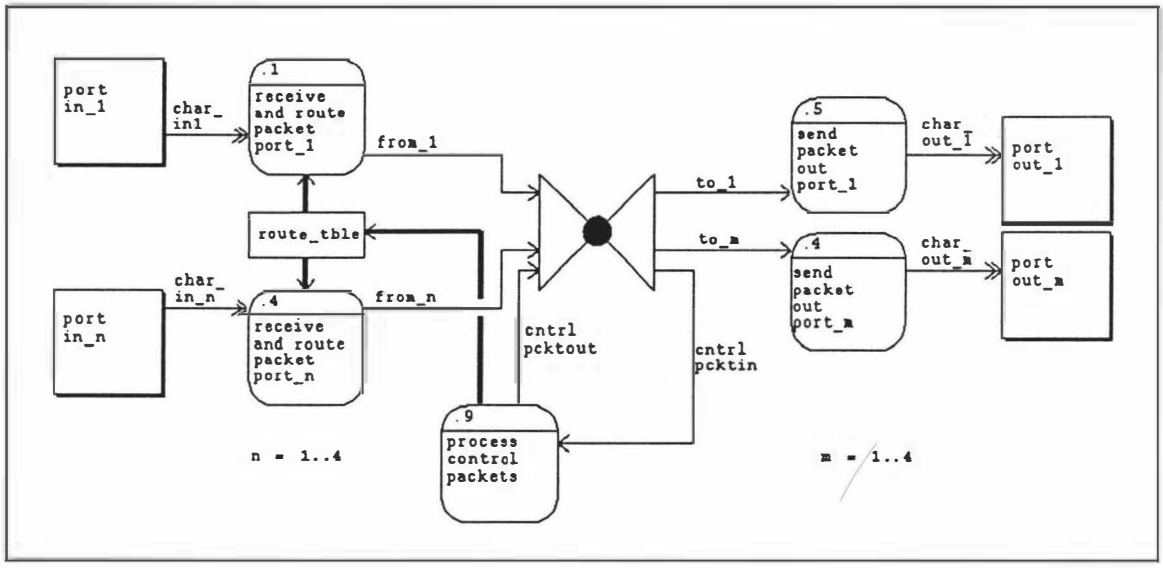


Figure 4.26 Hatley notation representation of simple packet switch device with four input streams.

While the Hatley & Pirbhai notation allows parts of a data flow diagram to be duplicated, the informality of the notation, particularly the boundary between the duplicated and non-duplicated parts of the diagram, and its redundancy, make it inappropriate for use in an automatically compiled language. The PICSIL notation has formalised the replication approach of Hatley & Pirbhai with the introduction of the *element* (by analogy with an element of an array). The dashed boxes explicitly represent the boundary between the replicated and non-replicated parts of the diagram. The label n = 1:4 defines the number of instances (four in this case) of the components contained within the element.

Flows which cross the boundary change from a single instance of the flow, inside the element, to a number of instances (4 in this case), outside the element. To reflect this change, a group flow (e.g. from[1:4]) is used to represent the multiple instances of the flow outside the element.

In the case of the single store flow attached to the route_table data store, the element boundary represents a connector, with a copy of the store flow being duplicated in each of the element's instances. The only other flow type that can have a single instance both inside and outside an

element is the continuous flow. A continuous flow may also change from a single flow to a group flow when it crosses an element's boundary indicating there is an instance of the continuous flow for each instance of the element.

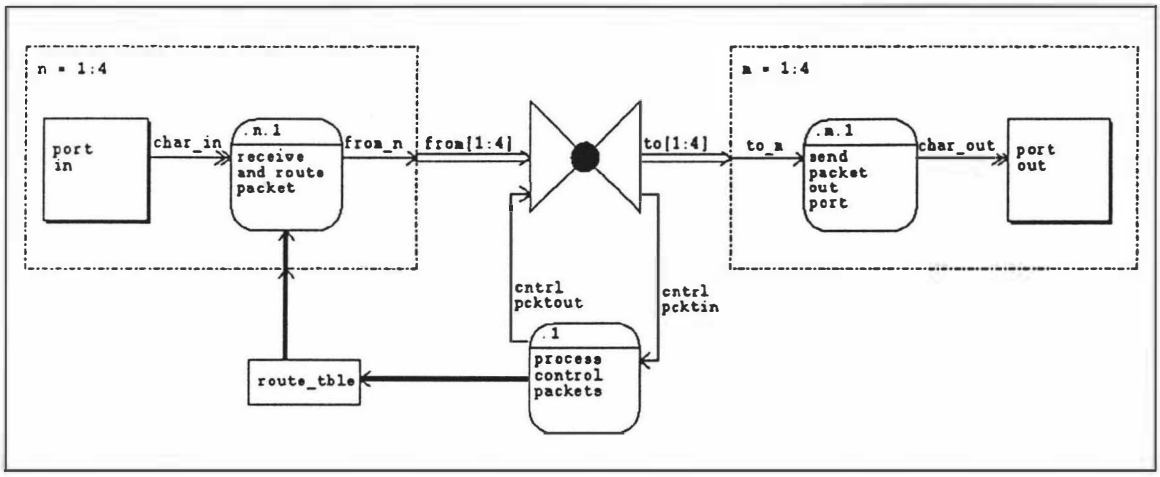


Figure 4.27 Final representation of packet switch using router and element

Figure 4.27 is the final representation of the packet switch. Although the process definition for receive and reroute packet has been given in Figures 4.23 and 4.25, these definitions do not represent the full functionality of the process. For example, each of the ports is serial, so characters making a packet are read in one bit at a time. If top-down design principles are used, the definition of this process should be broken into several sub problems. To accommodate this the PICSIL notation allows a process (non primitive process) to be decomposed as another data flow diagram (see Figure 4.28). As the PICSIL language can be input and edited directly on a computer, *links* have been added to the notation to aid the designer in keeping the set of data flow diagrams balanced.

A controller is included in this diagram to allow the two processes receive character and route packet to synchronise the transfer of packets into and out of the buff_pckt data store.

Comparison with conventional HDL representation

Figure 4.29 shows the top level representation of the packet switch in Verilog. Each of the InputPort instantiations creates an instance of an input port which performs the same function as the receive and reroute packet processes in the PICSIL representation. Likewise, each of the OutputPort instantiations creates an instance of an output port with the same functionality as the send packet out port processes in the PICSIL representation. The routing table and the packet manager are represented by the instantiation of the module PacketManager in the Verilog representation.

PICSIL's Positive Points also apply to this example. In addition, the comparison of the PICSIL and Verilog representations of the packet switch highlights three advantages of the PICSIL notation.

The first of these is that the constructor (i.e. the element) provides an efficient representation for replicated components. In comparison the Verilog representation requires explicit instantiation for each of the repeated components, making the resulting description larger and more difficult to read. Also, as each instance of a component has to be explicitly defined, more effort is required by the designer to avoid syntax and semantic errors in the description.

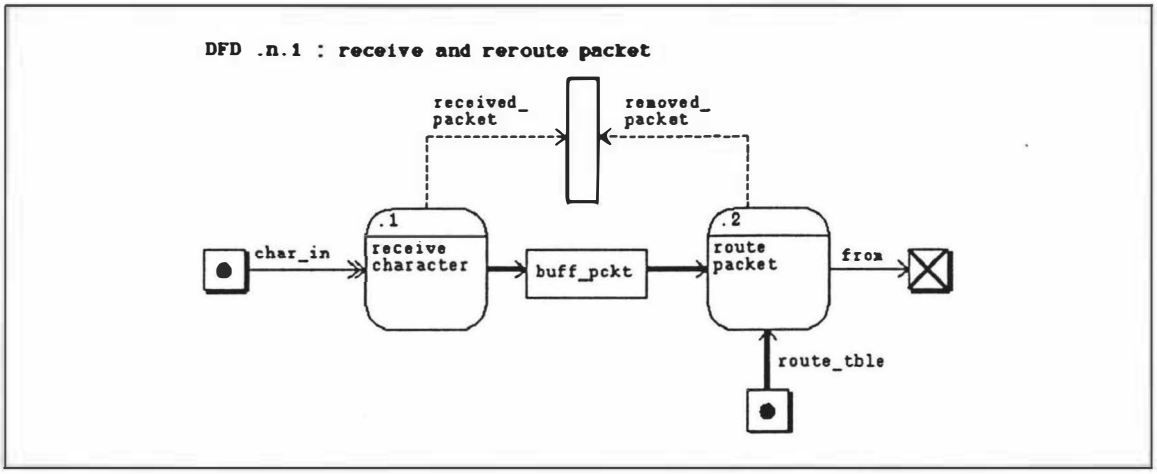


Figure 4.28 Decomposition of the process receive and reroute packet as a child data flow diagram

```
module packetSwitch(char_in1,char_in2,char_in3,char_in4,char_out1,char_out2,char_out3,char_out4)
        input char_in1, char_in2, char_in3, char_in4;
        output char_out1, char_out2, char_out3, char_out4;

        wire[7:0] from1, from2, from3, from4, fromcntrl, from1_data, from2_data, from3_data, from4_data,
                from1_add, from2_add, from3_add, from4_add;
        wire from1to1_rd, from1to2_rd, from1to3_rd, from1to4_rd, from1tocntrl_rt, from2to1_rd,
                from2to2_rd, from2to3_rd, from2to4_rd, from2tocntrl_rt, from3to1_rd, from3to2_rd,
                from3to3_rd, from3to4_rd, from3tocntrl_rt, from4to1_rd, from4to2_rd, from4to3_rd,
                from4to4_rd, from4tocntrl_rt, fromcntrlto1_rd, fromcntrlto2_rd, fromcntrlto3_rd,
                fromcntrlto4_rd, from1to1_ac, from1to2_ac, from1to3_ac, from1to4_ac, from1tocntrl_rt,
                from2to1_ac, from2to2_ac, from2to3_ac, from2to4_ac, from2tocntrl_rt, from3to1_ac,
                from3to2_ac, from3to3_ac, from3to4_ac, from3tocntrl_rt, from4to1_ac, from4to2_ac,
                from4to3_ac, from4to4_ac, from4tocntrl_rt, fromcntrlto1_ac, fromcntrlto2_ac,
                fromcntrlto3_ac, fromcntrlto4_ac,

InputPort ReceivePacket1(char_in1, from1, from1to1_rd, from1to1_ac, from1to2_rd, from1to2_ac,
    from1to3_rd, from1to3_ac, from1to4_rd,from1to1_ac, from1tocntrl_rd, from1tocntrl_ac,
    from1_data,from1_add,from1tort_rd,from1tort_ac);
InputPort ReceivePacket2(char_in2, from2, from2to1_rd, from2to1_ac, from2to2_rd, from2to2_ac,
    from2to3_rd, from2to3_ac, from2to4_rd,from2to1_ac, from2tocntrl_rd, from2tocntrl_ac,
    from2_data,from2_add,from2tort_rd,from2tort_ac);
InputPort ReceivePacket3(char_in3, from3, from3to1_rd, from3to1_ac, from3to2_rd, from3to2_ac,
    from3to3_rd, from3to3_ac, from3to4_rd,from3to1_ac, from3tocntrl_rd, from3tocntrl_ac,
    from3_data,from3_add,from3tort_rd,from3tort_ac);
InputPort ReceivePacket4(char_in4, from4, from4to1_rd, from4to1_ac, from4to2_rd, from4to2_ac,
    from4to3_rd, from4to3_ac, from4to4_rd,from4to1_ac, from4tocntrl_rd, from4tocntrl_ac,
    from4_data,from4_add,from4tort_rd,from4tort_ac);
OutputPort SendPacket1(from1, from1to1_rd, from1to1_ac, from2, from2to1_rd, from2to1_ac, from3,
    from3to1_rd,from3to1_ac,from4, from4to1_rd,from4to1_ac,fromcntrl, fromcntrlto1_rd, fromcntrlto1_ac,char_out1);
OutputPort SendPacket2(from1, from1to2_rd, from1to2_ac, from2, from2to2_rd, from2to2_ac, from3,
    from3to2_rd,from3to2_ac,from4, from4to2_rd,from4to2_ac,fromcntrl, fromcntrlto2_rd, fromcntrlto2_ac,char_out2);
OutputPort SendPacket3(from1, from1to3_rd, from1to3_ac, from2, from2to3_rd, from2to3_ac, from3,
    from3to3_rd,from3to3_ac,from4, from4to3_rd,from4to3_ac,fromcntrl, fromcntrlto3_rd, fromcntrlto3_ac,char_out3);
OutputPort SendPacket4(from1, from1to4_rd, from1to4_ac, from2, from2to4_rd, from2to4_ac, from3,
    from3to4_rd,from3to4_ac,from4, from4to4_rd,from4to4_ac,fromcntrl, fromcntrlto4_rd, fromcntrlto4_ac, char_out4);
PacketManager RouteTableManager(from1, from1tocntrl_rd, from1tocntrl_ac, from2, from2tocntrl_rd,
    from2tocntrl_ac, from3, from3tocntrl_rd, from3tocntrl_ac, from4, from4tocntrl_rd, from4tocntrl_ac,
    fromcntrl, fromcntrlto1_rd, fromcntrlto1_ac, fromcntrlto2_rd, fromcntrlto2_ac, fromcntrlto3_rd,
    fromcntrlto3_ac, fromcntrlto4_rd, fromcntrlto4_ac, from1_data, from1_add, from1tort_rd, from1tort_ac,
     from2_data, from2_add, from2tort_rd, from2tort_ac, from3_data, from3_add, from3tort_rd,
     from3tort_ac, from4_data, from4_add, from4tort_rd, from4tort_ac);
end module
```

Figure 4.29 Top level Verilog definition for the packet switch

Second, by contrast with Verilog, PICSIL provides constructs to allow data to be passed between modules using predefined interprocess communications protocols. While a designer can define his or her own protocols using Verilog functions or tasks, each of the handshaking signals used to implement the protocol has to be explicitly defined. In the case of the packet switch this accounts for over half of signals defined in the top-level Verilog representation shown in Figure 4.29.

Third, Verilog does not have an equivalent to PICSIL's router, and this requires the designer to define connections explicitly between each of the two sets of modules, adding an extra level of complexity to the description both in the top level and lower level descriptions. Figure 4.30 shows the Verilog representation of InputPort, which is similar in nature to the initial PICSIL representation before routers were added to the vocabulary. The main difference is that the definition also has to include the inter-module communications protocol for data transfers to occur.

```
module InputPort(char_in, from, fromto1_rd,
   fromto1_ac, fromto2_rd, fromto2_ac, fromto3_rd,
    fromto3_ac, fromto4_rd, fromto4_ac,
   fromtocntrl_rd, fromtocntrl_ac, from_data,
   from_add, fromtort_rd, fromtort_ac);

   input char_in, fromto1_ac,fromto2_ac,fromto3_ac,
       fromto4_ac, fromtocntrl_ac, fromtort_ac;
   output fromto1_rd, fromto2_rd, fromto3_rd,
       fromto4_rd, fromtocntrl_rd, fromtort_rd;
   output [7:0] from, from_add;
   input [7:0] from_data;

   reg[7:0] input_buffer [0:255];

   reg[7:0] new_VC, output_port_number, new_char;

      /** Defintion of process to receive a new
          character from input port**/
   always
      begin
         .../* code to read in new character on serial
              port and write to input_buffer*/
      end;

      /** Defintion of route packet process **/
   always
      begin
         .../* receive STB and VC bytes and use
            routing table to find new_VC and
            output_port_number*/
         case (output_port_number)
         1 : begin
               from = STB;
               fromto1_rt = 1;
               wait(fromto1_ac);
               fromto1_rt = 0;
               wait(~fromto1_ac);
               from = new_VC;
               fromto1_rt = 1;
               wait(fromto1_ac);
               fromto1_rt = 0;
            end;
         2 : begin
               from = STB;
               fromto2_rt = 1;
               wait(fromto1_ac);
               fromto2_rt = 0;
               wait(~fromto2_ac);
               from = new_VC;
               fromto2_rt = 1;
               wait(fromto2_ac);
               fromto1_rt = 0;
            end;
            /* same for other input ports */
         endcase
         .../*get first new_char from buffer*/
         while (new_char != ETB) begin
            case (output_port_number) :
            1 : begin
                  from = new_char;
                  fromto1_rt = 1;
                  wait(fromto1_ac);
                  fromto1_rt = 0;
                  wait(~fromto1_ac);
               end
            2 : begin
                  from = new_char;
                  fromto2_rt = 1;
                  wait(fromto2_ac);
                  fromto2_rt = 0;
                  wait(~fromto2_ac);
               end
               ... /* same for each of the other output
               ports*/
            endcase
            .../* get next new_char from buffer */
         end
      end
endmodule
```

Figure 4.30 Verilog definition of module InputPort in packet switch example

4.4 Conclusions

The design exercises discussed in this chapter have shown the evolution of the PICSIL language, demonstrated its sufficiency to describe hardware, and revealed a number of positive points:

- Graphical designs can express system level ideas effectively and concisely;
- PICSIL diagrams explicitly state inter-relationships between components in the diagram;
- The directions of data flows are explicitly stated on PICSIL diagrams;
- The graphical nature of PICSIL prevents syntax errors;
- PICSIL separates control and data processing sections of a design;
- The element provides an efficient representation for components that are repeated in a design;
- Discrete flows provide an efficient means for defining inter-process communication;
- The router provides an efficient mechanism to allow one set of processes to communicate with another.

While both PICSIL and conventional HDLs were able to represent each of the design exercises, apart from control-only designs (e.g. traffic light controller), the definition of the design and its readability were simpler in PICSIL in each case. The use of a graphical language allowed the components in a design and their inter-relationships to be represented explicitly rather than implicitly as with the textual languages.

The constructs for describing behaviour in PICSIL are similar to those for describing behaviour in conventional HDLs. However, in recognition of commonly occurring situations in behavioural descriptions, some graphical constructs (e.g. routers and controllers) have been added, and these simplify the resulting PICSIL descriptions. For example the use of a controller in the Ethernet receiver example allowed the receive frame process definition to be separated from the high level control aspects, with a very small communications overhead.

Two of constructs in the PICSIL language have been found to place an unnecessary burden on a designer. The first of these is that a controller has to "belong" to a data flow diagram, even though that data flow diagram may have no other function. The second (not mentioned previously) is the use of the data store I/O statements (stread and stwrite) which do not seem necessary in cases where a process wishes to make multiple accesses to the same datastore. However implementation constraints in the current version of the PICSIL synthesis system (see Chapter 6) prevent the format of these statements being changed.

The design exercises described in this chapter have shown that, despite having a limited vocabulary, the PICSIL notation is sufficiently powerful to represent designs from a wide variety of application domains. This has been achieved in part by the base notation, allowing a system to be decomposed into a set of processes which are interconnected by a network of data flows. The functionality of each process, and the data carried by data flows are described briefly in its name and together form an abstraction which, in most cases, achieves an ideal mix between conciseness and meaning. This limited set os additional constructs does not overload the language, but increases its power and conciseness, and help to manage design complexity.

Chapter 5

The PICSIL Editor

This chapter describes the form and implementation of the prototype PICSIL editor. The implementation is in the C language, with graphic support provided by Xview (Xview, 1990), one of Sun's proprietary graphic design toolboxes. Parts of the PICSIL editor which are concerned with driving the lower level synthesis path are discussed in Chapter 6.

In order to prevent this first implementation of an editor from delaying project completion unduly, some previously described features of PICSIL are not currently supported. These are multiple level state transition diagrams and graphical process activation tables. All state transition diagrams are currently constrained to one level, so a state cannot be defined by another state transition diagram. The graphical process activation tables outlined in Chapter 3 are currently implemented as a textual description which specifies the set of processes which are active in a particular state. Figure 5.1 shows syntax diagrams for the textual process activation table.

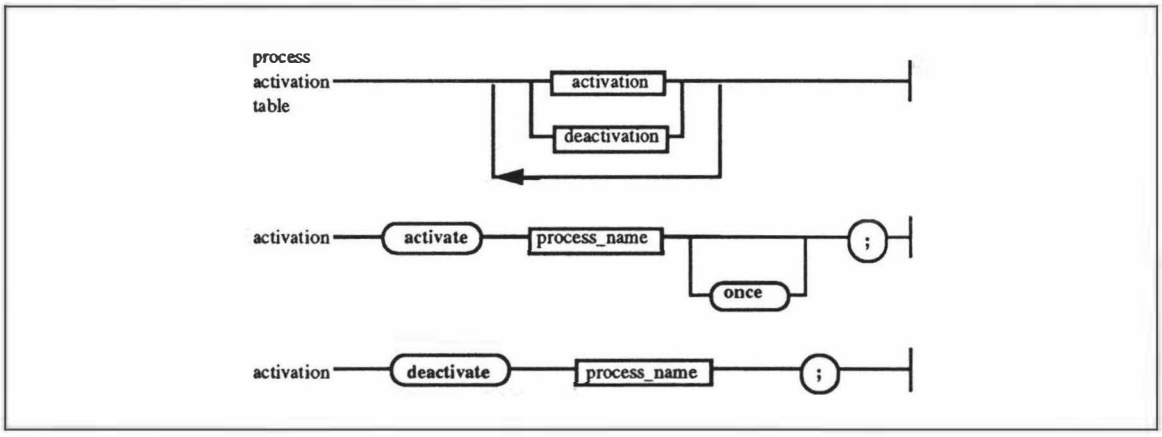


Figure 5.1 Syntax for textual process activation table

In the textual process activation table, an activation or deactivation should appear in each state for each process in the data flow diagram which the controller is describing. If the word *once* follows after a process name then it indicates that the process is activated once on entering the state.

5.1 An Overview of the PICSIL Editor

The development of an appropriate editing environment is clearly important to a research project which seeks to demonstrate that a direct-input graphical language is ideal for hardware design. However, fully optimising a graphical editor can be a large task, and the restricted duration of the project constrained development of comparatively standardised editor features to be offset against development of new and appropriate language constructs. Consequently, the PICSIL editing environment is adequate but unpolished. Suggestions for further development are included in Chapter 8.

The PICSIL editor has been designed for simplicity and transparency. It uses a direct manipulation interface to allow diagrams to be input and edited graphically. The editor supports multiple windows, and a new window is opened for each object that is refined. Figure 5.2 shows the various window types, which include the following:

- a menu bar gives the designer access to PICSIL system commands;
- a drawing window allows data flow and state transition diagrams to be captured;
- a text window allows textual specifications to be input and edited;
- a group flow window allows the components of a group flow to be specified;
- a set of iconised windows, which allow a number of relevant windows to remain on the screen in reduced form.

All input operations performed within the PICSIL editor use the standard Sun three-button mouse and a keyboard. The term pointer is used in this chapter to define the cursor displayed on the screen that tracks the mouse movements. Although a three-button mouse is used, PICSIL editing operations are exclusively based on a two-button paradigm. The right button activates a context-sensitive menu (i.e., one which alters to display objects which can occur in the current window), and the left mouse button selects and manipulates existing objects. The left and right buttons therefore are sometimes refereed to as the select/manipulate button and the menu button respectively.

The Drawing Window

Both data flow diagrams and state transition diagrams are input and edited *via* the drawing window. Figure 5.3(a) shows a blank-area menu which lists the items which can be added to a data flow drawing window. This menu appears when the menu (right) button is depressed while the screen cursor is in a blank area of the drawing window. Figure 5.3(b) shows the result of depressing the menu button while the screen cursor is over an existing process icon. In either case, dragging the cursor over a particular menu item and releasing the mouse button causes an action appropriate to the cursor's position to occur; instantiation of an object in the case of the blank-area menu, and one of the listed actions in the case of the process menu. As the type of mouse-based interface behaviour just described is common practice, future discussions of menu activation will be brief.

The objects have been divided into two groups: flow objects (flows and state transitions) and nodes (all other objects, such as processes, data stores, states, etc.) When the user selects a node, the PICSIL editor fills it with grey and attaches a number of handles (■) at points on its perimeter (see Figure 5.4(a)). These handles define the points at which flows can start or end. Clicking the select/manipulate (left) mouse button while the pointer is on a handle starts the addition of a flow from that point. An example of a selected process is shown in Figure 5.4(a).

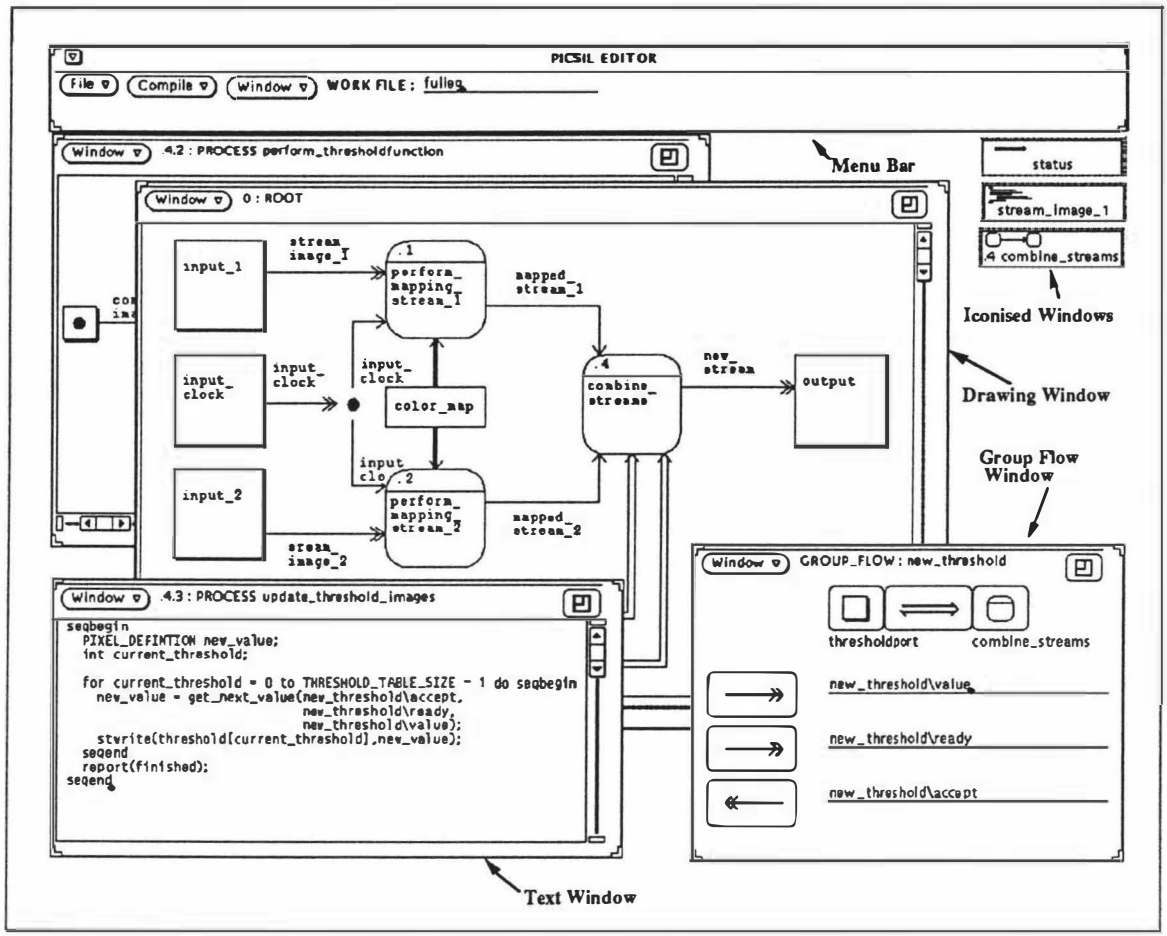


Figure 5.2 PICSIL editor's various windows

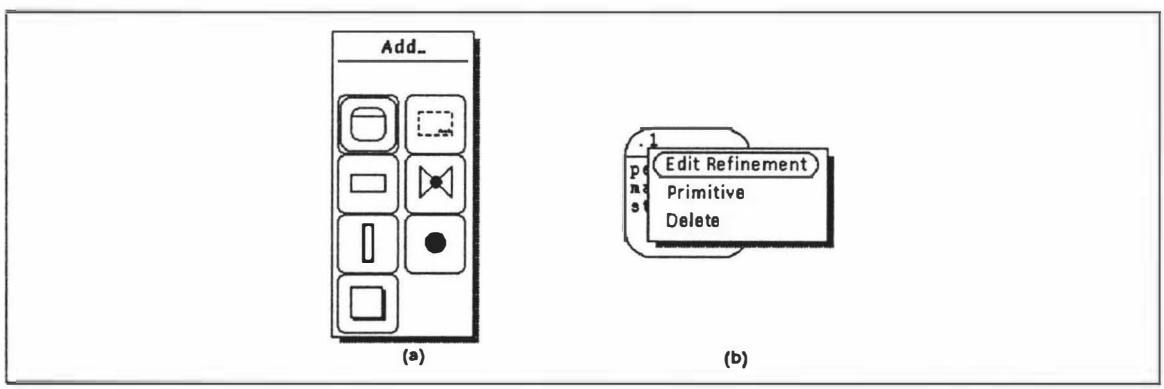


Figure 5.3 Context Sensitive menus used in the PICSIL Editor
a Blank area menu, and **b** process menu

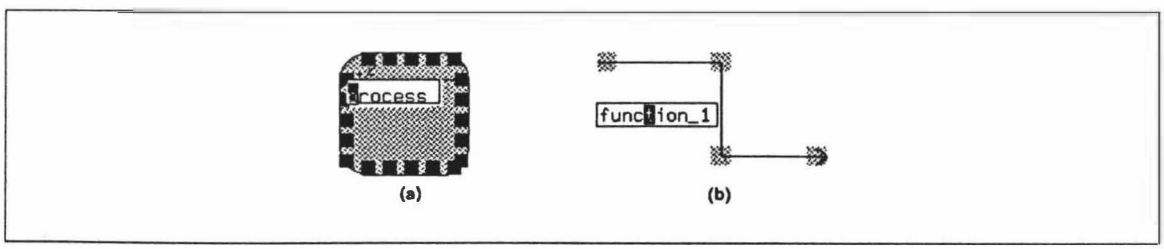


Figure 5.4 Examples of selected objects in the PICSIL editor
(a) Selected process and (b) selected flow

A flow is represented by a set of horizontal and vertical line segments connecting two nodes. The user creates the route of the data flow by clicking the select/manipulate mouse button with the cursor at successive vertex locations and terminates with a select/manipulate mouse button click when the pointer is over another node. To indicate that a flow is selected, a grey shaded box (▒) is drawn around each of its vertices (see Figure 5.4(b)). These grey shaded boxes also form handles which allow the path of the flow to be changed.

Processes, data stores, external entities and flows can be named. The names can be edited directly in the drawing window when the object they belong to is selected. When the object is deselected the name is updated in any other windows in which the object appears to reflect the editing changes.

The Text Window

The text window is used to edit textual definitions of primitive processes, discrete and continuous data flows, states, state transitions and data stores. PICSIL uses the standard text editing functions provided with the Xview toolbox (see below), and is directly involved only in the opening and closing of text windows.

The Group Flow Window

The group flow window allows the child flows that make up a group flow to be specified. Each child flow in a group flow is shown in the window by an icon representing the child flow and a name field. If the menu (right) mouse button is depressed when the cursor is positioned over a flow icon, the user is presented with a menu listing the operations which can be performed on the flow (such as editing its refinement and changing its type or direction) To allow the designer to determine the direction of a child flow relative to its parent, a representation of the parent flow and the two objects to which it is attached are shown at the top of the window.

Iconised Windows

Each of the different types of window may be turned into a small icon to prevent the screen becoming cluttered, while still allowing the designer easy access to the window. A double click using the select/manipulate mouse button, while the pointer is over the icon, opens the window again at its position when it was iconised.

As a device is designed using PICSIL, the editor generates a large data structure to represent it. This contains all the information needed to support three distinct types of operation on the design, capturing it, saving and restoring it, and presenting it to the layout synthesis software. The first two of these operations are described later in this chapter; the third involves a consideration of the requirements of the synthesis software and is left to the next chapter.

5.2 Xview

When the implementation of PICSIL began, the Sun workstation and the Macintosh Plus were the two platforms available for the author's use. The Macintosh Plus was considered too small in terms of screen size and processor speed to implement the PICSIL editor, leaving only the Sun workstation.

At the time of the implementation, Sunview (Sunview, 1988) was considered to be the best graphics toolkit available. More recently the PICSIL editor has been converted to use the Xview graphics toolkit offering two main advantages. First, Xview has been designed to run under the

X window system, which has implementations on most types of UNIX workstation currently available, and is thus not restricted to Sun workstations. Second, tools used in the lower level synthesis path used the X window system to display their results. However, using Sunview it was not possible to display these windows.

Xview is a user interface toolkit for supporting interactive, graphics based applications running within windows. It includes a run time system which relates input events to the window in which they occur, and performs window overlap housekeeping, and also an output handler which allows the applications to generate their output in terms of a number of high-level metaphors:

- canvases (on which programs can draw),
- text subwindows (with in-built editing capabilities),
- panels (containing items such as buttons and choice items and menus which can pop-up anywhere on the screen)

Xview is an object oriented system in which the objects are the building blocks, including menus, buttons, canvases and windows. Objects can be created and destroyed as required. When an object is created it is given a unique *handle* to identify it. An object can also be associated with a user-supplied function which is called to perform the desired task when an input event on the object occurs. When the function is called, the handle and event type are passed to the function.

5.3 Windows

There was a major change in the approach to drawing objects within the PICSIL editor during its evolution. The first implementation was mode-oriented; the interpretation of mouse click events depended upon which of several modes (see Figure 5.5) the user had selected.

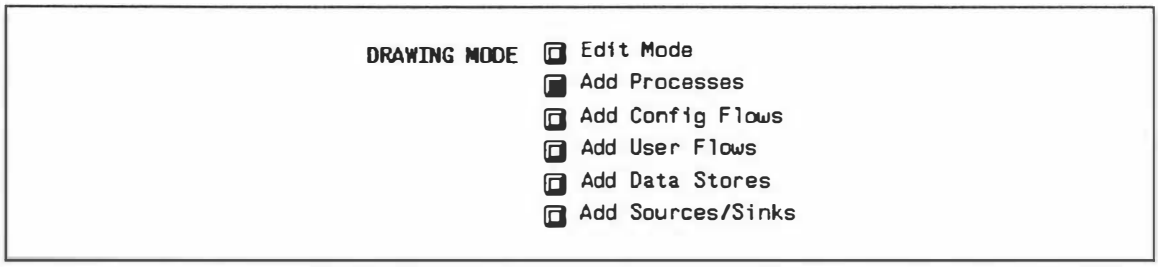


Figure 5.5 Drawing modes in the initial implementation of the PICSIL editor

In the current implementation, explicit mode selection has been dispensed with and the interpretation of mouse event depends on the position of the mouse. For example new nodes are added by selecting an item from a pop-up menu displayed by depressing the right mouse button while the pointer is not over another object. If however the pointer is over an object when the right mouse button is depressed then a pop-up menu defining operations that can be performed on that object appears.

Each of the different types of window has two parts: a title bar and a description part. The title bar has a label to identify the type and name of its contents (e.g. DFD 0 : root). A zoom box (回) also appears at the right hand of the title bar, and is used to toggle the window's size between a predefined value and a user-defined value.

A number of window operations are provided, each accessed from the *Window* pull-down menu shown in Figure 5.6.

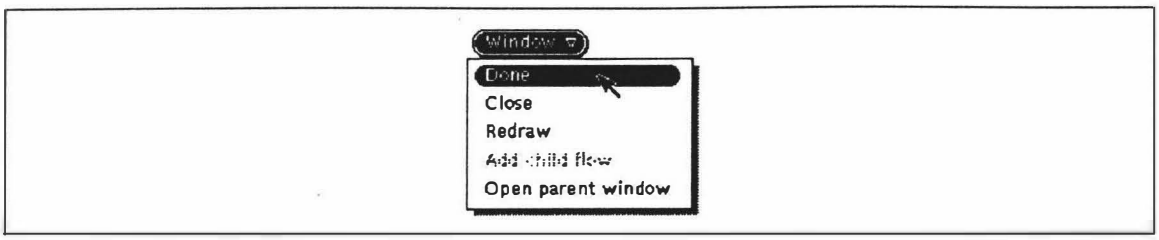


Figure 5.6 The menu which appears when the right mouse button is depressed with the pointer over the window button

When a window is closed by selecting the Done option from the window menu, it is hidden from view. When the close option is selected, the window is turned into an icon, which can be opened again by double clicking with the left mouse button while the pointer is over the icon.

If the Window pull-down menu is displayed in a drawing window the redraw option is active and allows the contents of that window to be redrawn. This has simplified implementation of the editor, as it obviates the necessity for the system to perform a total display consistency check after every editing operation. Any inconsistencies are very obvious to the user, and disappear when the window is completely redrawn using information from the PICSIL data structure. In text and group flow windows, this option is unnecessary, so it is greyed out and may not be selected by the designer.

The Add child flow option on the Window menu is active only in a group flow window and allows new child flows to be added to the group flow. The Open parent window option will open or bring to the front a window containing the parent diagram of the window.

The *window data structure* is a dynamic structure used by the PICSIL editor to keep track of the currently open windows and contains a linked list of instances of each type of window. Each entry in the linked list represents a single window and has the following information :

- an Xview window handle;
- a pointer to the parent object in the PICSIL data structure;
- a pointer to the diagram header in the case of a drawing window;
- the current size and position of the window;
- a zoom status flag;
- a next window pointer.

Operations on windows such as resizing and moving are handled by the Xview package, although after a window has been resized, a PICSIL function is called to ensure the zoom box is placed at the right hand end of the title bar.

When a designer selects the *Window* menu's done option, the window is removed from the screen and its local data structure is deallocated. For text windows, PICSIL's global data structure must be updated before the deallocation of the local data structure occurs, as, in by contrast with editing operations in drawing and group flow windows, text window edits do not cause immediate updating of the PICSIL data structure.

The Menu Bar

The menu bar is present whenever the PICSIL editor is running. It has two parts: a file name area and a menu area. The file name area contains the name of the PICSIL design that will be used by operations on the file menu in the menu area. The file name area is implemented as a Xview panel and can be edited so that a new design can be created or an old one can be opened or saved.

The menu area is made up of three pull-down menus corresponding to the three window buttons file, compile and window. Items selected from these menus allow general operations to be performed. Figure 5.7 shows the appearance of the three menus.

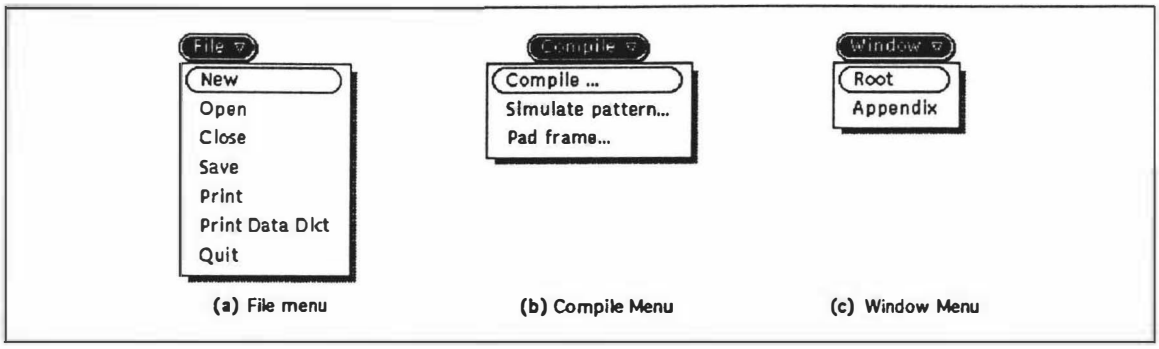


Figure 5.7 The three menus that appear on the menu bar

The File Menu

The operations that can be performed from the file menu can be broken into two groups: those involving disk I/O and those involved in hardcopy output of a design.

The file operations Close, Save and Quit do not need detailed explanation; they do what they say, and involve everyday housekeeping such as checks to prevent editing operations from being lost when files are closed. Open and New cause a window to be opened to display the root diagram after a design has been loaded or created.

Print saves a screen dump - of a part of the screen selected by the designer - to a file. Print Data Dict prints all object definitions associated with an object_description.

The Compile Menu

The compile menu has items to initiate and drive the synthesis process. As the synthesis process is discussed in the next chapter, discussion of the compile menu is also deferred to then.

The Window Menu

The last type of menu, the window menu allows certain windows to be opened. Root opens a window to display the root diagram in it. If the root window is already open then it will be bought to the front. Appendix opens a text window for the appendix, so that constants, procedures and functions can be defined.

5.4 Using the PICSIL Editor

In this section, the basic editing operations used in creating and editing a design are introduced, using the image processing example described in Chapter 3 (see page 27).

To clarify the drawing sequences shown in this section, icons representing particular mouse and keyboard events are shown in most frames. Figure 5.8 shows the icons used and their interpretations. Where more than one icon appears in a frame, the left-most icon represents the first event in a sequence, and the right-most icon represents the last.

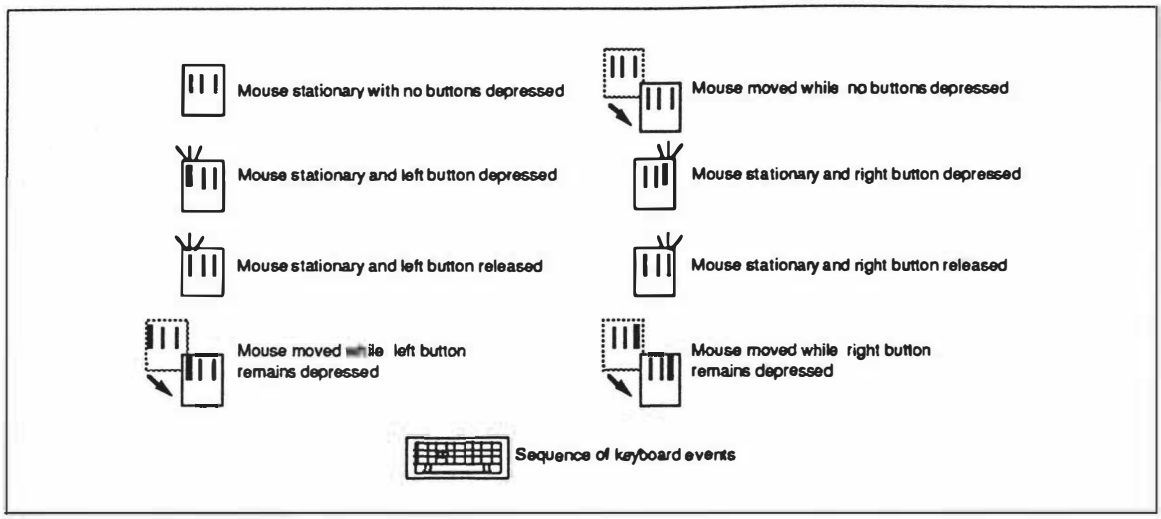


Figure 5.8 Icons used to represent events in sequences

When PICSIL is invoked from the UNIX command line, the menu bar and a blank drawing window representing the top level PICSIL diagram are shown. A name is given to the design by typing a name in the name field in the menu bar.

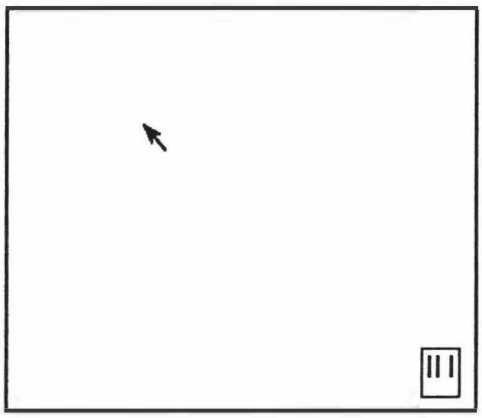
The first stage in creating a PICSIL diagram is to add the nodes that make up the diagram. New nodes are added by selecting items from a pop-up menu displayed when the menu button is pressed while the pointer is over a blank area in the drawing window. Figure 5.9 shows the sequence of operations required to add the process perform_mapping_stream_1 to the window. The other nodes in the diagram are added using the same sequence of operations. The editor will not allow a node to overlap another object.

Once two or more nodes have been added to a diagram, the data flows between them can then be added using a sequence of mouse clicks. Figure 5.10 shows the sequence of operations necessary to add a flow called mapped_stream_2 between the processes perform_mapping_stream_2 and combine_streams.

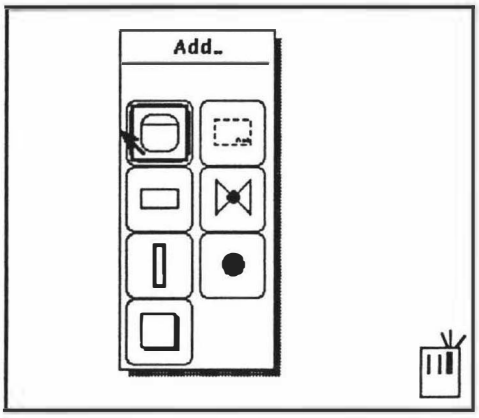
The PICSIL data structure is referred to repeatedly during the addition of a flow to allow the editor to:

- check that the flow originates from a free handle
- delete the redundant vertex if a new line segment is an extension of the previous one
- prevent the path of a flow segment from intersecting an existing node
- detect when a flow segment has been terminated over an existing node, indicating that the flow is destined for that node, and subsequently attach the flow to a free handle, update the links within the data structure, and perform consistency checking on the data transferred by the flow.

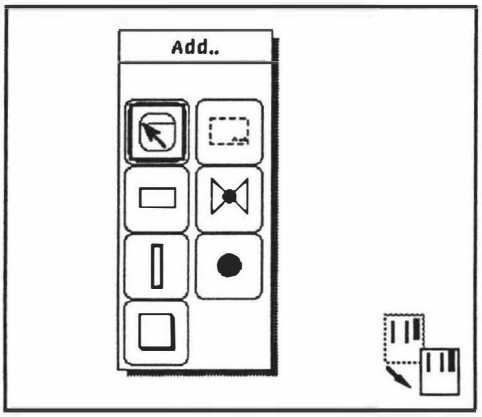
Figure 5.11 shows the sequence of operations involved in the changing type of the flow new_threshold from continuous to group. Only flow types which are valid for the data in the flow are selectable.



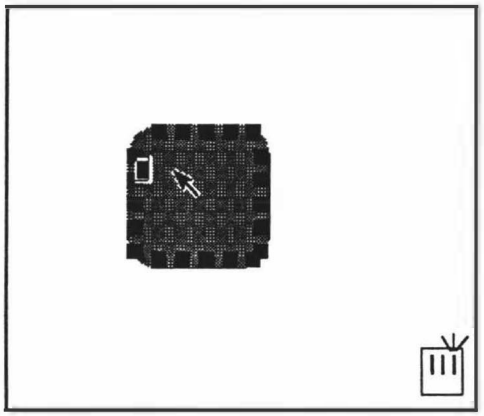
(a) Pointer moved to position (x,y) at which process is to be added.



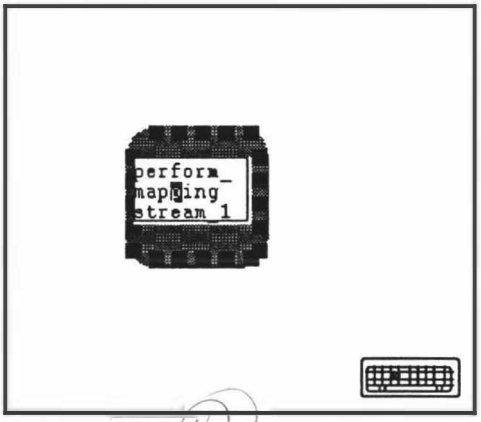
(b) (menu down, blank area) displays blank area menu.



(c) (move, to process icon)

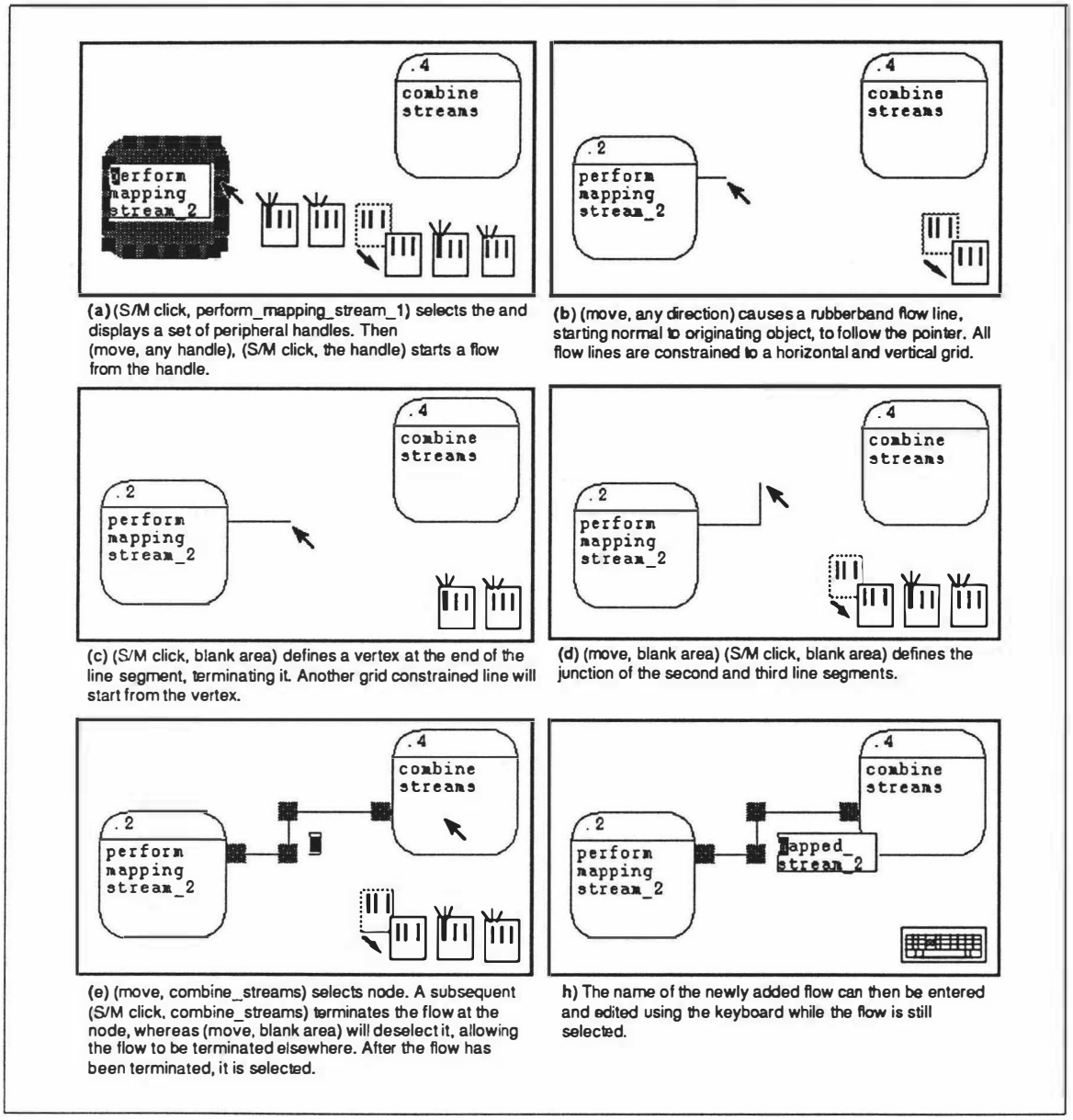


(d) (Menu up, process icon) causes process to be added with its top left corner at (x,y). The process is automatically selected



(e) The process's name can be entered and edited using the keyboard while the process is still selected.

Figure 5.9 Sequence of events to add a node



(a) (S/M click, perform_mapping_stream_1) selects the and displays a set of peripheral handles. Then (move, any handle), (S/M click, the handle) starts a flow from the handle.

(b) (move, any direction) causes a rubberband flow line, starting normal to originating object, to follow the pointer. All flow lines are constrained to a horizontal and vertical grid.

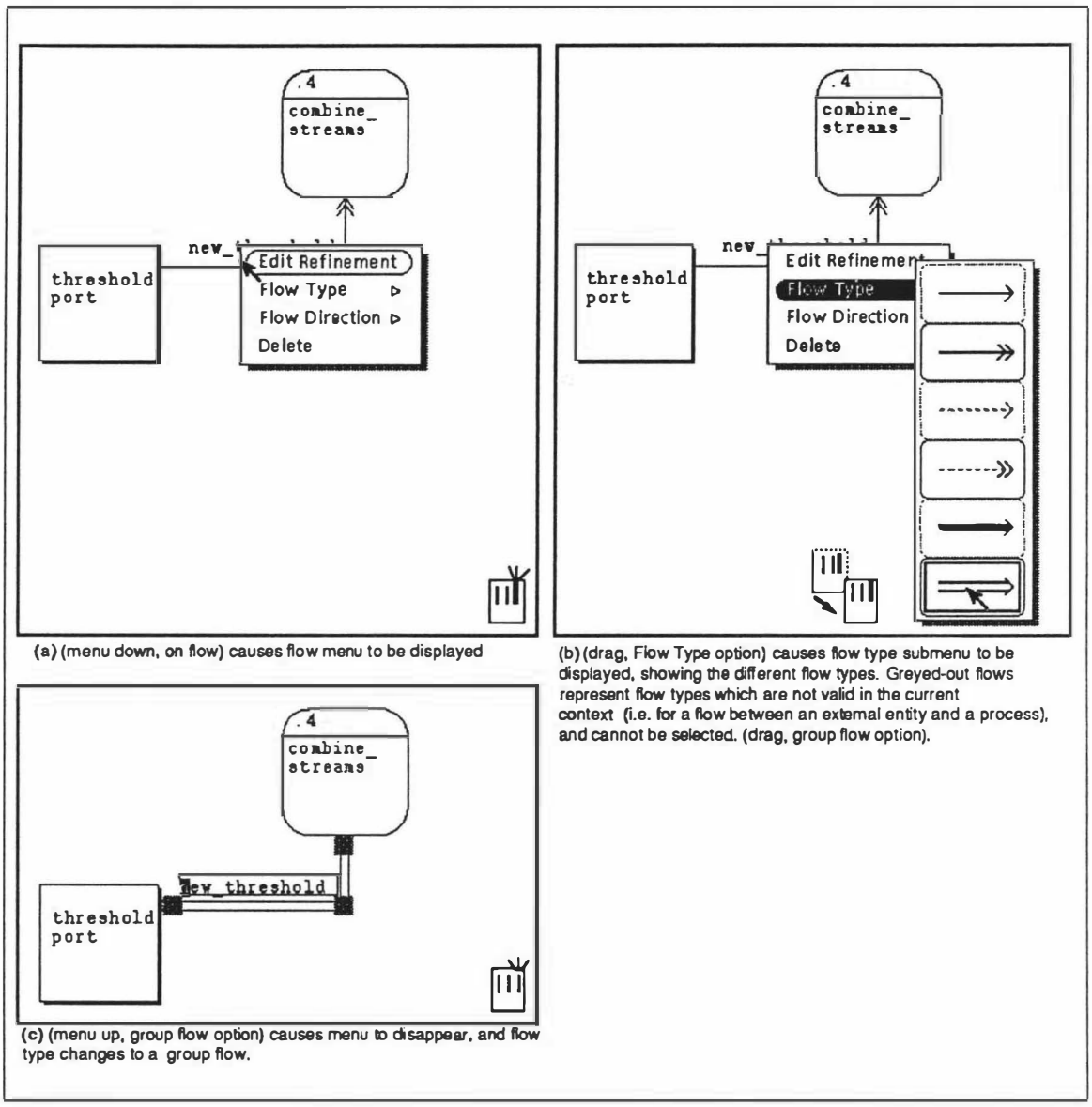
(c) (S/M click, blank area) defines a vertex at the end of the line segment, terminating it. Another grid constrained line will start from the vertex.

(d) (move, blank area) (S/M click, blank area) defines the junction of the second and third line segments.

(e) (move, combine_streams) selects node. A subsequent (S/M click, combine_streams) terminates the flow at the node, whereas (move, blank area) will deselect it, allowing the flow to be terminated elsewhere. After the flow has been terminated, it is selected.

h) The name of the newly added flow can then be entered and edited using the keyboard while the flow is still selected.

Figure 5.10 Operations involved in adding new data flow



(a) (menu down, on flow) causes flow menu to be displayed

(b) (drag, Flow Type option) causes flow type submenu to be displayed, showing the different flow types. Greyed-out flows represent flow types which are not valid in the current context (i.e. for a flow between an external entity and a process), and cannot be selected. (drag, group flow option).

(c) (menu up, group flow option) causes menu to disappear, and flow type changes to a group flow.

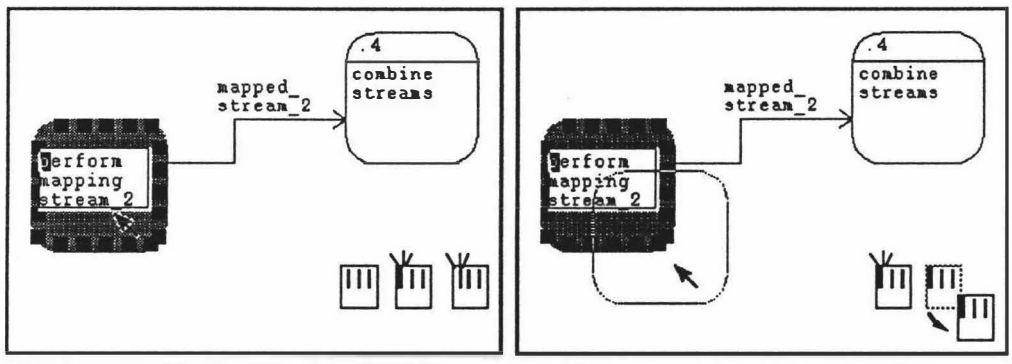
Figure 5.11 Using the flow menu to change a flow's type

Editing a DFD

A number of non-destructive DFD modification operations are supported by the editor. Figures 5.12, 5.13, and 5.14 show the sequences of operations involved, respectively, in:

- shifting a node, the process perform_mapping_stream_2;
- changing the destination of a flow, mapped_stream_2;
- changing the path of a flow, mapped_stream_1, without altering its destination.

The editor prevents the new diagram from containing overlaps where it is simple to do so. The main exception to this may occur when a node is shifted. Its attached flows move with it, and may then end up overlapping other objects. However, subsequent editing of the flow will reestablish the ban on overlaps.

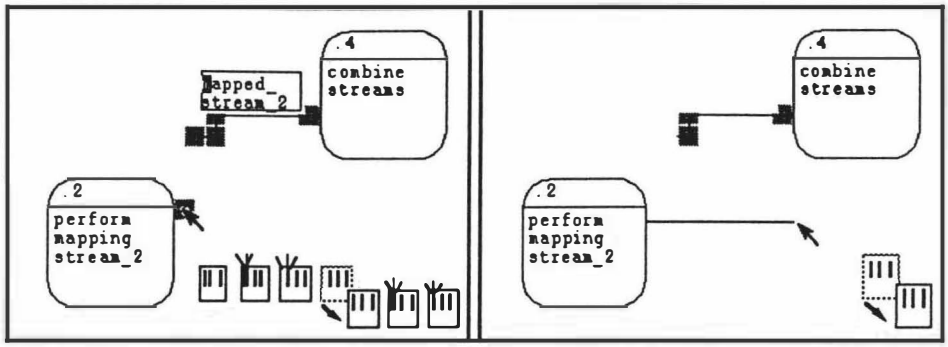


(a) (S/M click, object to be moved) will select object

(b) (S/M down, selected object) (drag, selected object) causes an outline of the object to track its movements

(c) (S/M up, objects new position) causes object to be moved to new position. Extra line segments are added to the flows attached to the object before it was moved, so they remain attached after the move

Figure 5.12 Operations involved in moving a node



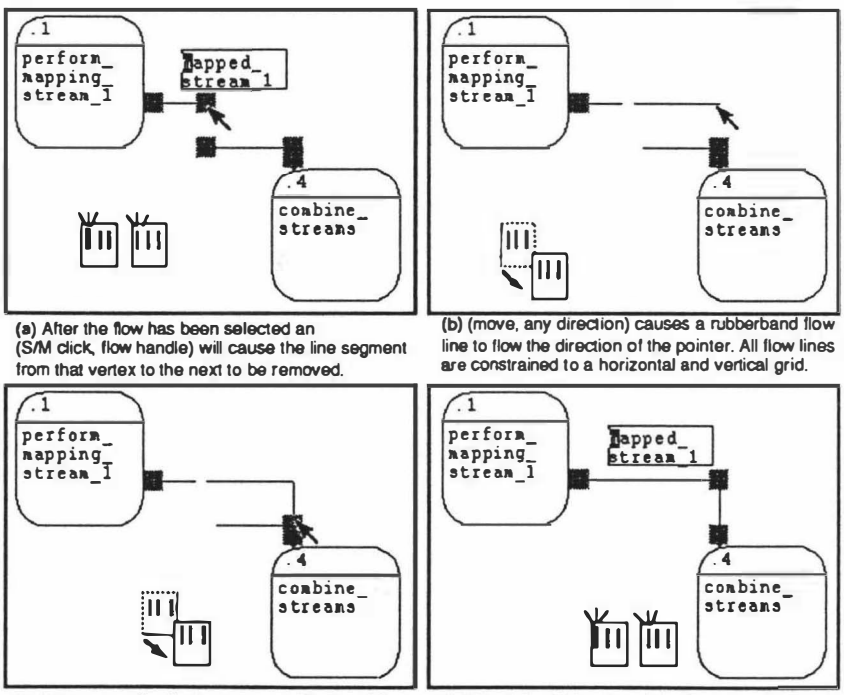
(a) (S/M click, mapped_stream_2) will select flow. (S/M click , flow handle) causes the line segment from that vertex to the next to be removed from the screen.

(b) (move, any direction) causes a rubberband flow line to flow the direction of the pointer. All flow lines are constrained to a horizontal and vertical grid.

(c) (S/M click, blank area) defines a vertex at the end of the current line segment, terminating it. Another grid constrained line segment will start from the vertex.

(d) (move, object) selects the object. A subsequent (S/M click, object) terminates the flow at the node, and selects it. The disconnected part of the flow is deleted. The user may then drag the flow name to an appropriate position using a similar technique to that used for moving primary objects

Figure 5.13 Changing the path of a flow to end on a new node handle



(a) After the flow has been selected an (S/M click, flow handle) will cause the line segment from that vertex to the next to be removed.

(b) (move, any direction) causes a rubberband flow line to flow the direction of the pointer. All flow lines are constrained to a horizontal and vertical grid.

(c) (move, any direction) causes a rubberband flow line to flow the direction of the pointer. All flow lines are constrained to a horizontal and vertical grid.

(d) (S/M click, old flow handle) will cause new flow path to join the old flow path. The unused part of the old flow path is deleted.

Figure 5.14 Changing the path of a flow which finishes on part of the old flow

To allow a designer to explore a design space, and experiment with different configurations, it is necessary to be able to delete objects from a diagram. Apart from links, the designer can delete any object on a diagram by selecting the delete option from the objects pop-up menu. Deleting an object may also cause other objects to be deleted so as not to leave other parts of the diagram in an inconsistent state. For example if a process is deleted then all of its attached flows are also deleted, so as not to leave flows with only one end attached in a diagram. If more than one object will to be deleted as a consequence of deleting another object, the PICSIL editor will notify the designer of the consequences of performing the deletion (see Figure 5.16).

Figure 5.15 shows the completed top-level PICSIL diagram for the image processing example. All objects in this diagram other than external entities, the connector and store flows require further definition. Using the PICSIL editor they can be further defined by selecting the Edit Refinement option from the objects pop-up menu (displayed by depressing the right mouse button while the pointer is over the object).

If the object's window is currently open, but obscured, it will be brought to the front of the display. If it has been previously opened, it will be reopened with its earlier size and position restored. Otherwise, a new window of the correct type (drawing or flow), and of standard size and position will be created.

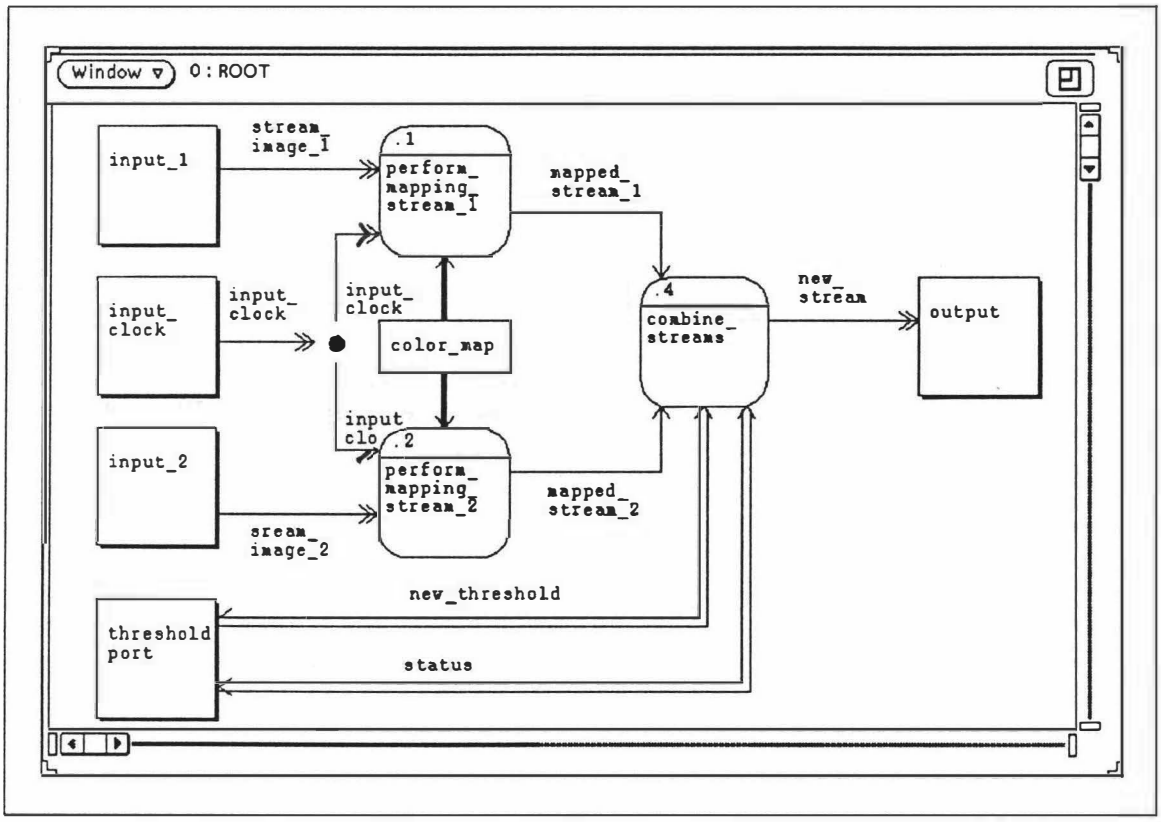


Figure 5.15 Top level data flow diagram for image processing example created using the PICSIL editor

Text windows are used to further define data stores, discrete and continuous flows, and primitive processes. Figure 5.17 shows a window being opened to define the store color_map. If the store has previously been defined, its definition is copied from the data structure into the text window. All editing operations are performed under the control of the Xview package. Any edits made on the text window are copied back to the data structure when either the design is saved, or the text window is closed by selecting the done option from the window's window menu.

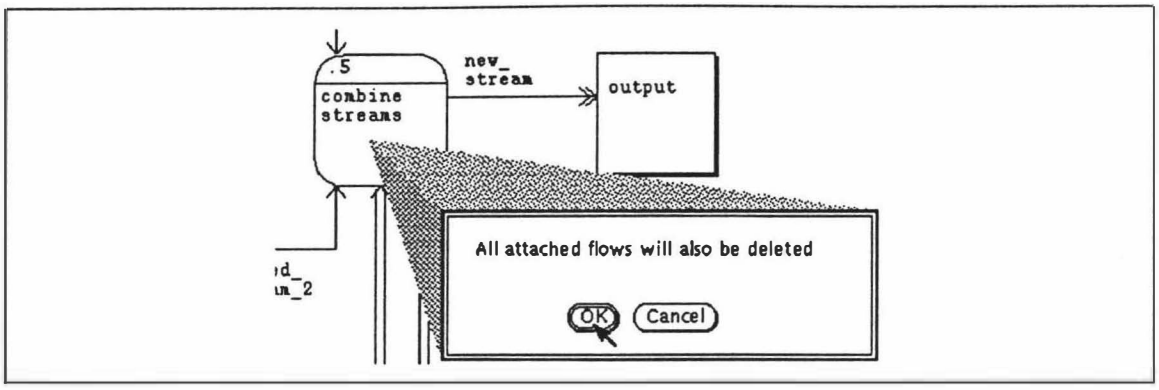


Figure 5.16 Dialogue box shown when a designer attempts to delete a process with flows attached

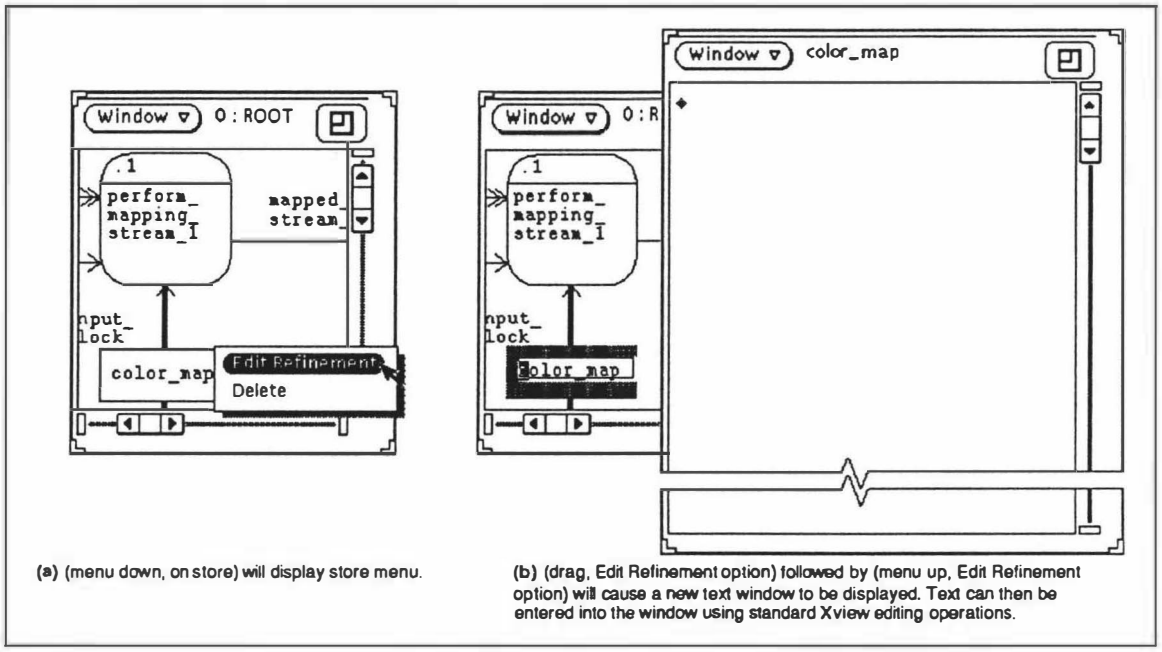


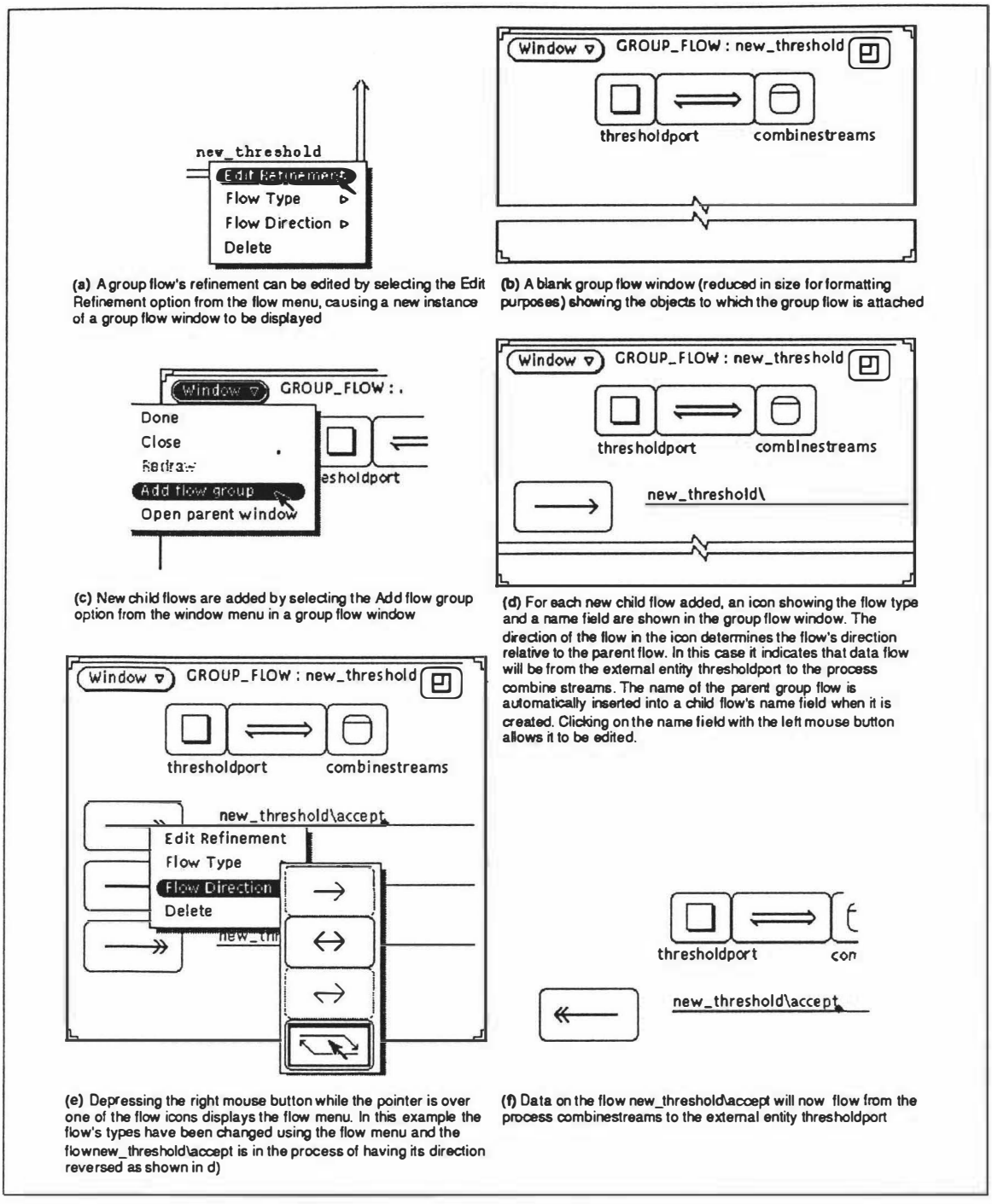
Figure 5.17 Opening a text window to define a data store's contents

Group flow windows are used to define the child flows that make up a group flow. Figure 5.18 shows a summary of the operations that can be performed in a group flow window. When new child flows are added, or an existing child flow's type or direction is changed, the data structure is updated immediately. However changes made to names of child flows are copied back to the data structure until the group flow window is closed or the design is saved.

Drawing windows are used to augment the definition of controllers and non-primitive processes. Figure 5.19 shows the sequence of operations used to define the non-primitive process combine_streams further.

When a definition window for a non-primitive process is first opened, its import and export flows are automatically displayed as links on the left and right edges, respectively, of the diagram. Subsequently, the ordinary PICSIL node shift operation can be applied to them. They will disappear if their parent flows in the parent diagram are deleted (see below), but they cannot be deleted explicitly.

As external entities can be added only in the root diagram, they are greyed-out on the blank area menu for child diagrams. The PICSIL editor also allows group flows to be decomposed when they enter a child diagram (see Figure 5.20).



(a) A group flow's refinement can be edited by selecting the Edit Refinement option from the flow menu, causing a new instance of a group flow window to be displayed

(b) A blank group flow window (reduced in size for formatting purposes) showing the objects to which the group flow is attached

(c) New child flows are added by selecting the Add flow group option from the window menu in a group flow window

(d) For each new child flow added, an icon showing the flow type and a name field are shown in the group flow window. The direction of the flow in the icon determines the flow's direction relative to the parent flow. In this case it indicates that data flow will be from the external entity thresholdport to the process combine streams. The name of the parent group flow is automatically inserted into a child flow's name field when it is created. Clicking on the name field with the left mouse button allows it to be edited.

(e) Depressing the right mouse button while the pointer is over one of the flow icons displays the flow menu. In this example the flow's types have been changed using the flow menu and the flownew_threshold\accept is in the process of having its direction reversed as shown in d)

(f) Data on the flow new_threshold\accept will now flow from the process combinestreams to the external entity thresholdport

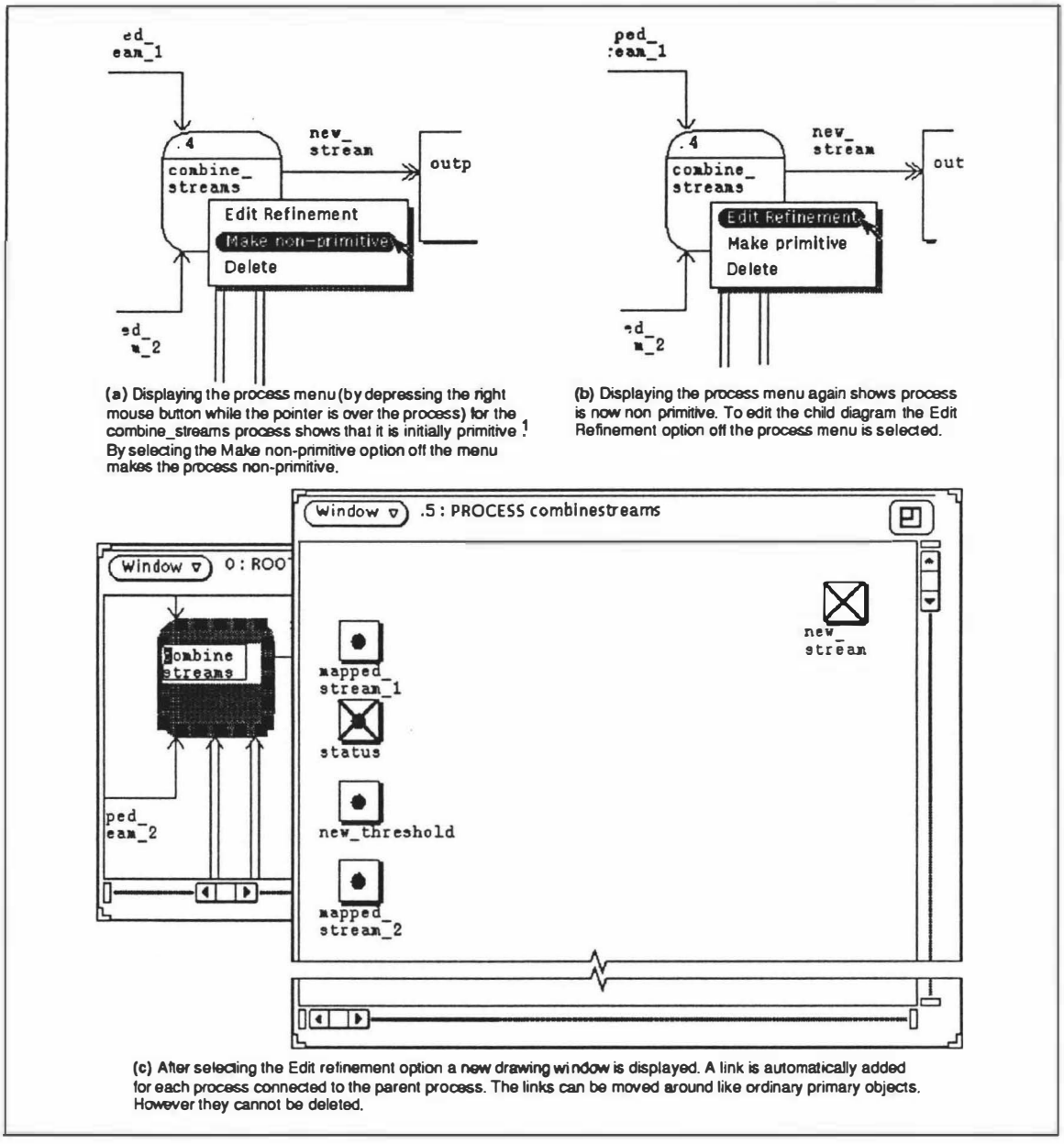
Figure 5.18 Summary of operations that can be performed in a group flow window

As multiple windows can be open simultaneously, it is possible that updates made in one window will affect the contents of other windows. In the development of the editor attempts have been made to ensure any changes made in a window are reflected immediately in other windows. When flows are attached to a non-primitive process by adding a new flow or changing the path of an existing one, links are automatically added to the child diagram, and the new link is drawn if a window is open for the child diagram. Conversely if a flow is detached from a process by deleting the flow or changing its path, its links and attached flows are deleted from the data structure in all the child diagrams of the non-primitive processes.

As controller (state) diagrams are similar to Data Flow Diagrams (both are directed graphs), editing a controller diagram is very similar to editing a DFD. At the visible level, the menus

differ, reflecting the different vocabulary of controller diagrams. Moreover, but invisible to the designer, all of a controller's state transitions have a common data dictionary entry.

Selecting the Edit Refinement option from the pop-up menu for a state causes a text window to be opened in which a textual version of the process activation table (discussed at the beginning of the chapter) can be entered.



(a) Displaying the process menu (by depressing the right mouse button while the pointer is over the process) for the combine_streams process shows that it is initially primitive.[1] By selecting the Make non-primitive option off the menu makes the process non-primitive.

(b) Displaying the process menu again shows process is now non primitive. To edit the child diagram the Edit Refinement option off the process menu is selected.

(c) After selecting the Edit refinement option a new drawing window is displayed. A link is automatically added for each process connected to the parent process. The links can be moved around like ordinary primary objects. However they cannot be deleted.

Figure 5.19 Sequence of operations to define a non-primitive process further

[1] In the implementation of the PICSIL editor discussed in this chapter, the only way to tell whether a process is primitive or not is to display its menu. If the menu contains the item Make non-primitive, then the process is currently primitive, by contrast, if the menu contains the item Make primitive the process is currently non-primitive. In the final stages of this work the PICSIL language has been changed so that there is a visual difference between the symbols used for primitive and non-primitive processes (see figure 5.28).

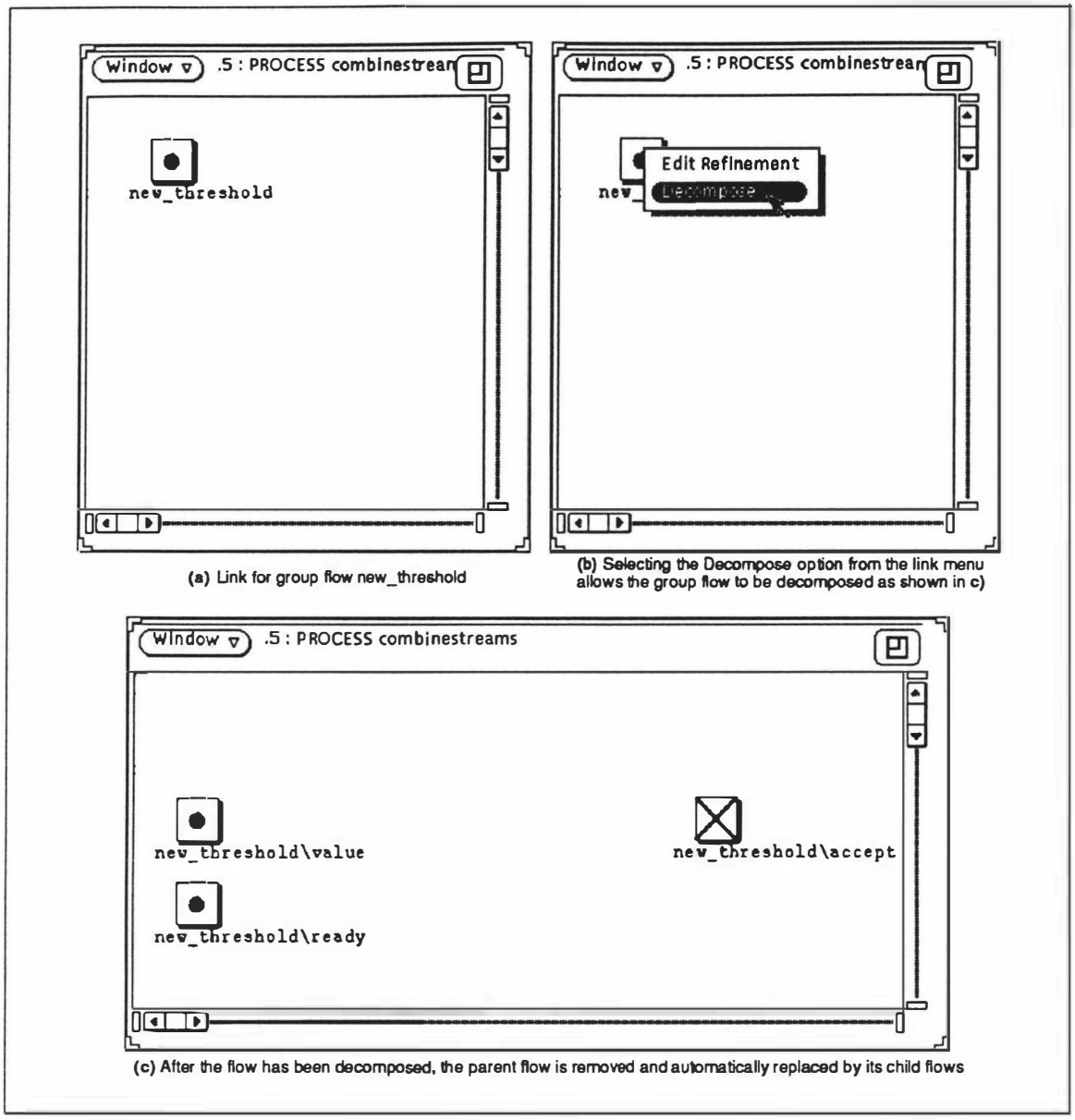


Figure 5.20 Refining a link representing a primitive process using the PICSIL editor

As the PICSIL editor can have multiple windows open at one time it is possible that updates made in one window will affect the contents of other windows.

5.5 The PICSIL Data Structure

The PICSIL data structure is at the centre of the PICSIL editor. It must satisfy three main requirements:

- it must completely represent the PICSIL language in the editor;
- it must be useable as input to the synthesis process;
- it must be able to be written to and retrieved from disk.

The data structure is based on the different record types shown in Figure 5.21. Each diagram in a PICSIL design has a diagram header (DIAGRAM) associated with it. The objects (processes, flows, stores etc) that make up the diagrams each have a record (OBJECT) to define their main details. These object records are stored in one of two linked lists: one for flows and one for nodes. Figure 5.22 shows the basic format of the diagram header and the two linked lists.

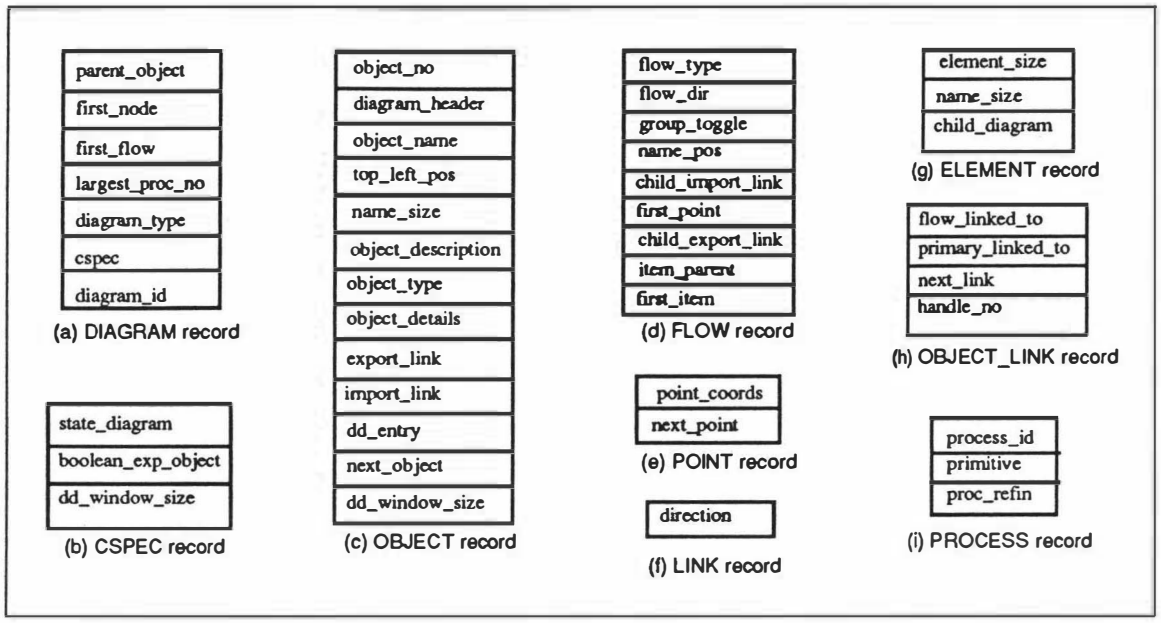


Figure 5.21 The record types used in the PICSIL data structure

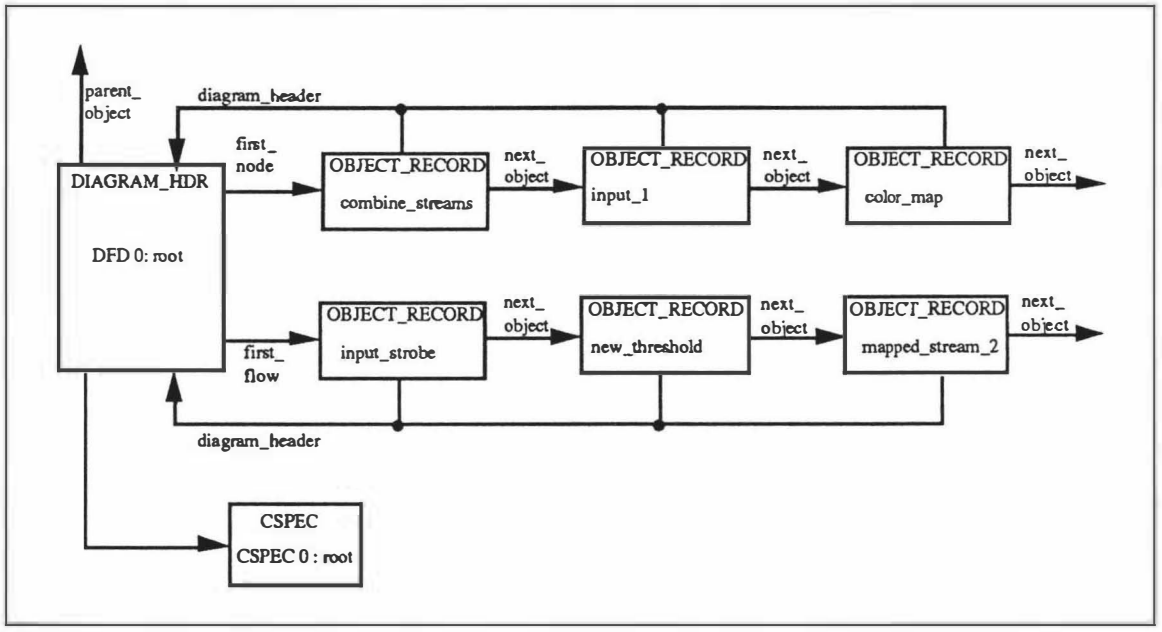


Figure 5.22 Basic format of PICSIL data structure

In the initial implementation, the OBJECT record represented only the details common to all objects. All other details were stored in a record appropriate to the object type (e.g. PROCESS, STORE). For example all but one object type required a single pair of coordinates to describe the object's position, and the positional information was put into each of the object type records. While this created a structure which required minimal space, it led to the replication of large sections of code.

The final structure still has individual record types for some objects. However all details common to more than one object type have been moved into the OBJECT type record. For example, object coordinates have been moved to the main object record even though not all object types use this field. This greatly reduced the complexity of the coding and maintenance task. Figure 5.23 shows how various objects are stored.

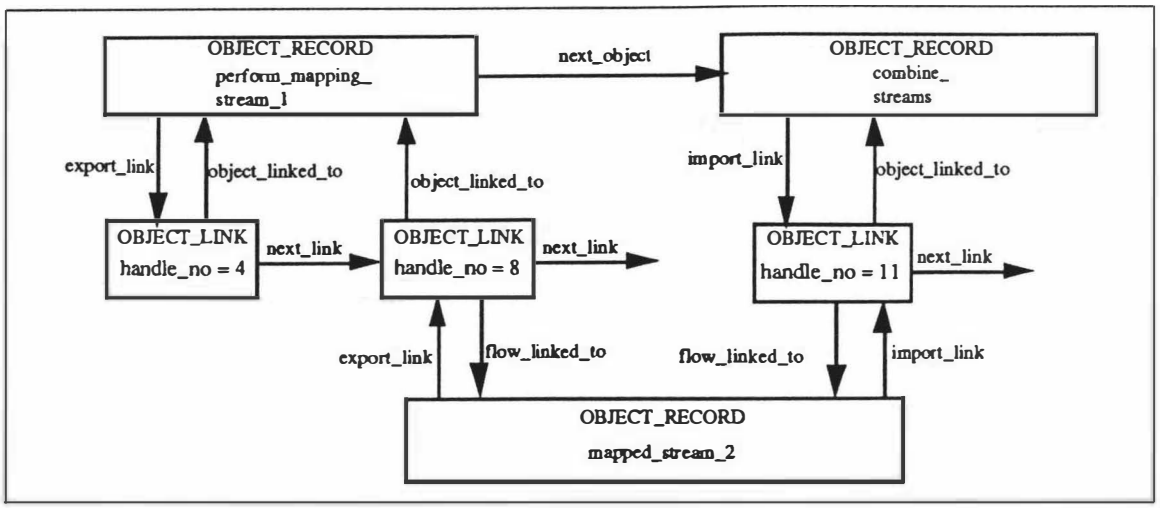


Figure 5.24 Linking between nodes and flows

Figure 5.25 also shows that details which are specific to each of the diagrams in which the flow appears, such as its path within the diagram, are stored in the child diagram's flow_record.

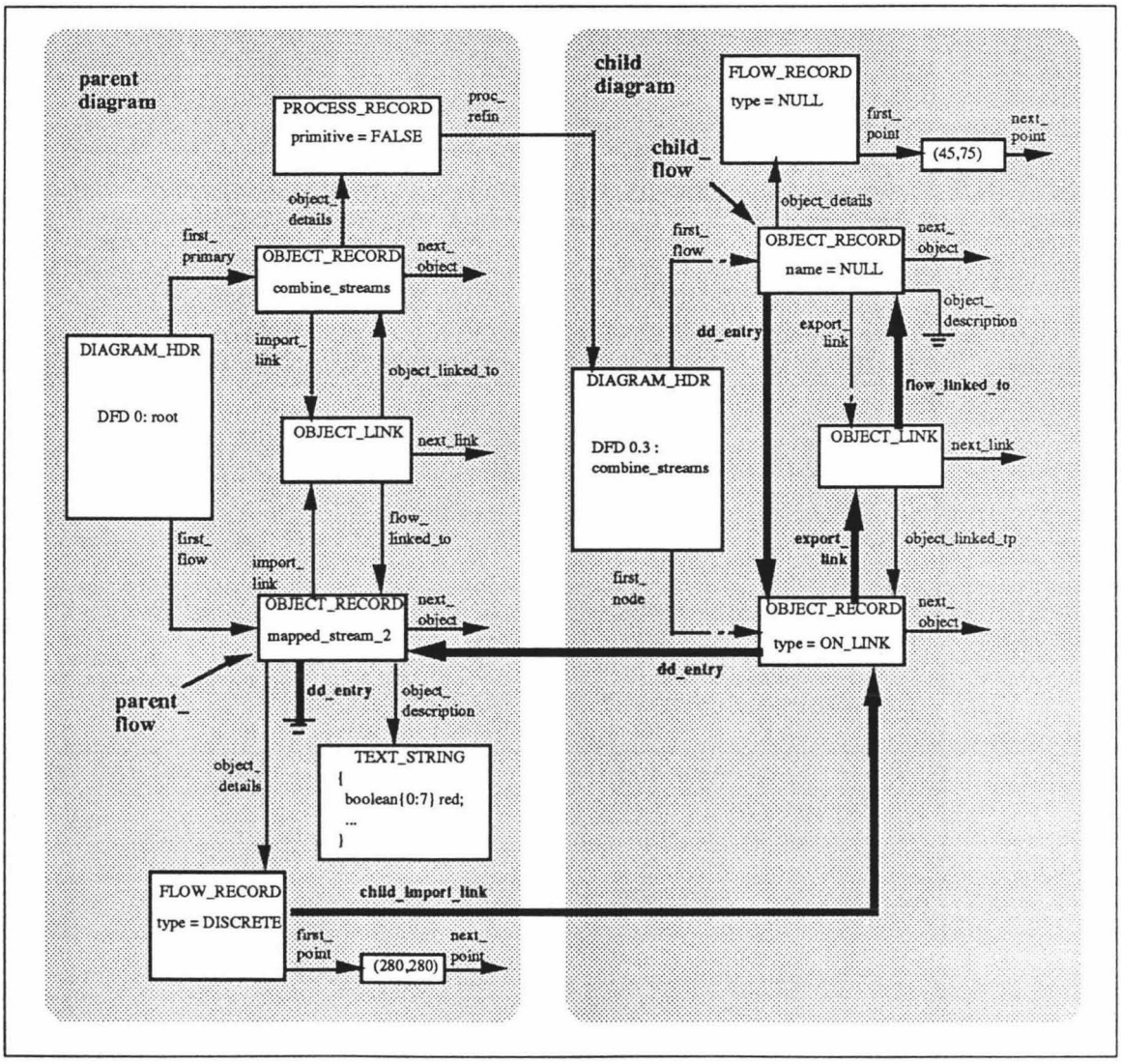


Figure 5.25 Linking of parent and child flows between diagrams

The PICSIL editor has *save* and *load* routines for interface to disk files. As the PICSIL data structure is dynamic, and large areas within its overall memory map may be deallocated space, it is not desirable to store a complete memory map when the data structure is saved to file.

In the case of simple linked lists, the items are stored in the file in the order in which they are appear in the linked list, making it possible to rebuild the linked list when the file is read from disk. However some pointer fields of OBJECT type do not form part of a linked list, so a representation of the pointer needs to be stored to disk. Instead of saving the absolute value of a pointer to disk, a unique number given to the object the pointer points to is saved instead. When the data structure is read from memory, a lookup table is used to replace the unique numbers with the memory address allocated when the object the unique number represents was read from disk.

The structure of the save and load routines closely resembles the record types that they work with. Both routines have a function for each record type which transfers that record type to or from disk. Each field of a record has a statement or statements to read or write the field.

Representation of Objects in Drawing Windows

Most of the effort in implementing the editor has gone into the routines associated with the drawing windows. As the drawing window is used for capturing both data flow diagrams and state transition diagrams, two major modes exist: a DFD mode for editing data flow diagrams and a state transition mode for editing state transition diagrams.

As the PICSIL editor has been developed in parallel with the PICSIL notation the objects to be manipulated have been changed from time to time. The graphical representation of the objects has been standardised to make these changes easier. Apart from the element, which may itself contain a sub-diagram of arbitrary complexity, all representations of nodal objects are a fixed size and shape, so only one set graphical representation exists for each. The positional information stored for each node in the data structure comprises the x and y coordinates of its top left corner. Each object type then has the following set of attributes to describe its representation within the drawing window :

- a set of vertices relative to its top left corner to define the object outline;
- the number of vertices;
- the object width;
- the object height;
- an array of handle positions;
- the number of handle positions.

An array is present for each attribute, and each object type has an entry in each of the arrays. Each of the object types has an integer to identify it so that the integer can be used as an index into an array. Most of the routines that deal with nodes use entries in this array, so adding a new object or editing an existing one quite often means that only small changes to the code are required. Figure 5.26 shows the set of attributes used to represent a data store on a drawing window. The only graphical information stored in the PICSIL data structure about each object is its type and the coordinates of its top left corner.

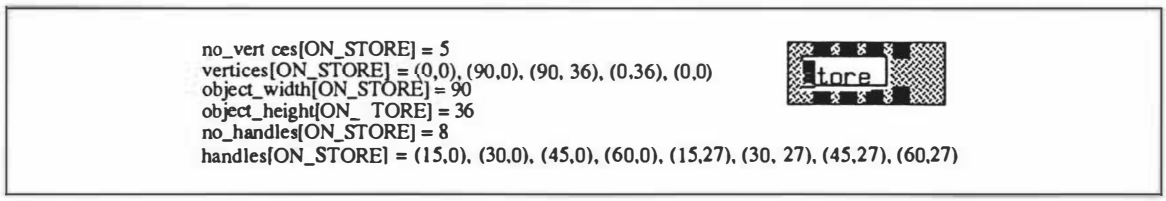


Figure 5.26 Attributes used to represent a data store on a drawing window

Each of the handles displayed on an object has a number, so that when a flow is attached, the handle number is recorded in the OBJECT_LINK record (see Figures 5.21 and 5.24) between the object and the flow. This gives an easy means of ensuring that two flows are not connected to the same handle.

As the element may contain a number of other objects, it does not have a fixed size, so it is treated differently from the other nodes. The width and height are stored in the ELEMENT record and all the other attributes are calculated dynamically as required. For example, the positions and numbers of handles are calculated each time they are to be displayed.

Elements may contain other objects within their boundaries. The interior of an element is treated by the editor as a separate diagram with its own diagram header. The position data stored for all the objects within the element is relative to the top left hand corner of the element. Thus when an element is moved, all the coordinates of its internal objects do not have to be updated.

Two sets of handles exist for an element as shown in Figure 5.27 The interior set allows flows to be connected between objects within the element and the element border. The external set allows flows to be connected between the element border and other objects exterior to the element. Abutting interior and exterior handles have the same handle number. In the current implementation, no connection is made in the data structure between the internal and external handle of the same number so the designer must provide a data dictionary entry for both flows.

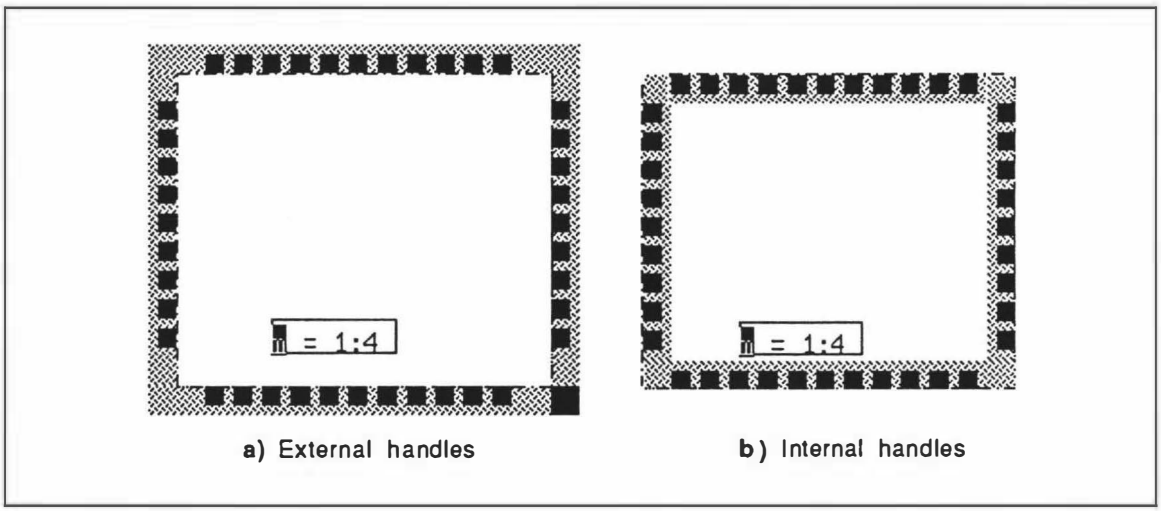


Figure 5.27 Internal and external handles for a element

To aid the designer in drawing and editing diagrams, all objects are automatically aligned to a grid. In the current implementation, the grid pitch is set at fifteen pixels

5.6 Conclusions

This chapter has discussed the development of a prototype editor which allows the direct capture and editing of the PICSIL HDL. The purpose of the implementation has been to gain experience in the use of such an editor and demonstrate the ability to capture a PICSIL design (including data flow diagrams and data dictionary entries) directly. The development of a commercial system, which would be expected to be free from errors, has not been a consideration in the development of the editor.

To test the editor, the ten design exercises discussed in Chapter 4 have been input using the editor. To ensure that capturing a design using the PICSIL editor achieves the goal of being faster than by hand, three main problem areas need to be addressed.

First, using the current version of the PICSIL editor, the designer is required to open a separate window for each data dictionary entry. In the case of data flows this has been found to be very time consuming, particularly when a lot of the data flows have the same definition. A more integrated data dictionary, allowing a single definition to be associated with multiple objects, could greatly improve a designer's efficiency.

Second, the editing operations provided by the editor are not powerful enough to allow a designer to explore a design space effectively. For example, if a designer wishes to add a new object on the left hand side of a drawing and there is not enough space, then all the rest of the objects must be moved to the right one at a time. The addition of more powerful editing operations such as block moves (allowing any number of objects to be selected and moved at the same time) and cut, paste and copy operations, could greatly aid the designer in editing PICSIL diagrams.

Third, a number of cases were found where parts of one design could be used in another. For example, a process which received characters in a parallel format and output them in a serial format was required in four of the design exercises. However no facility allowed a process to be defined in one design and be included in other designs. In its simplest form this problem could be solved by allowing cut, copy and paste operations between different designs.

In a more comprehensive form, the editor would include a library sub-system allowing frequently used components (e.g. primitive and non-primitive processes) to be stored for easy retrieval. This would require a means of mapping the library components data flow names to the working design names to be developed.

The editor does, however, offer a number of advantages over creating PICSIL designs by hand. First, the data structure used to represent the PICSIL language in the editor is used as input to the synthesis process (see next chapter). This allows the overall design time to be reduced, as no manual translation to a purely textual representation is required, which is the case with designs created by hand.

Second, the PICSIL editor has features to ensure consistency is maintained through a levelled set of diagrams. For example when a flow is added to a non-primitive process a link is automatically created in the child diagram. If a designer examines a diagram and sees a link with no flow attached, it is immediately obvious that a flow is attached to the parent process and not used in the child diagram. Also any changes made a flow name, type or direction are immediately updated in all diagrams where the flow appears.

Development of the PICSIL editor has been an ongoing process, with improvements being made up to the time of the submission of this dissertation. One improvement made too late to be included throughout the body of the dissertation is the marking of non-primitive processes. Using the editor in the form previously described it is not immediately obvious whether a process is primitive or non-primitive. To help designers differentiate between the two types of process, non-primitive processes are now automatically drawn with a double width outline. For example Figure 5.28 shows a recent image of a drawing window, which contains one non-primitive (i.e. combine_streams) and two primitive (i.e. perform_mapping_stream_1 and perform_mapping_stream_2) processes.

While the current implementation of the PICSIL editor is not of commercial quality, its development has shown that it is possible to capture PICSIL designs directly in a relatively simple and easy to use manner.

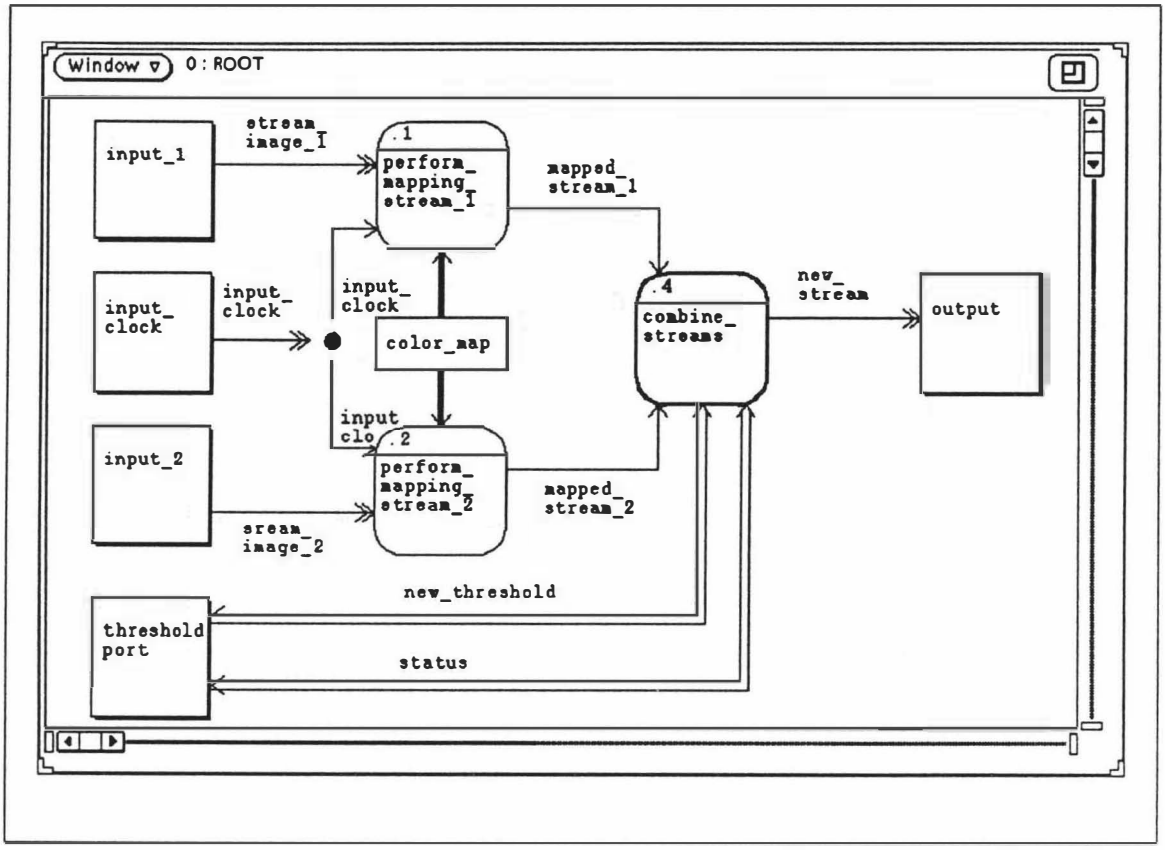


Figure 5.28 Latest Version of PICSIL editor which uses a double width outline to represent non-primitive processes.

Chapter 6

From PICSIL to Hardware

This chapter discusses the development of an experimental synthesis system (known as the PICSIL synthesis system) which allows PICSIL designs to be realised in hardware. The PICSIL synthesis system bridges the gap (see Figure 6.1) between the high-level behavioural specification output by the PICSIL editor and a description of a chip layout.

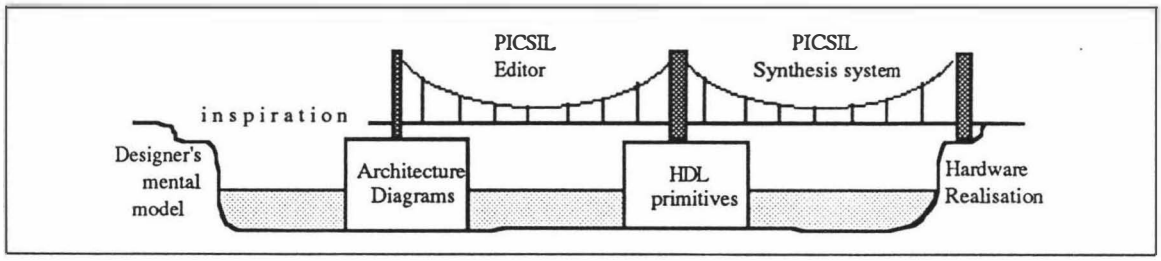


Figure 6.1 How the PICSIL system bridges the semantic gap

Production of a chip layout from high level behavioural descriptions is a complex process and is traditionally split into three stages: behavioural synthesis, logic synthesis and physical synthesis (De Micheli, 1990).

High level synthesis involves the translation of a high-level behavioural description into a structural description of logic components and their interconnections. In general, high-level synthesis is technology-independent. It first optimises the design's behaviour and then converts the behavioural description into structure generally using area and time constraints.

Logic synthesis follows high-level synthesis and is used to optimise the logic description output by high-level synthesis. It generally involves two phases. The first is technology-independent circuit optimisation, while the second is mapping of the logic description into technology dependent library parts.

The third stage, physical synthesis, deals with the geometric design of a chip and involves the placement of the library components and routing the interconnections between them.

A number of synthesis systems have been produced both in academia and in industry which deal with one or more stages of the synthesis process. Tools based on logic and physical techniques are mature enough to be used in production systems, but whereas the development

of high level systems has so far been restricted to research projects, and they generally do not deal with lower level aspects such as physical synthesis.

The primary objective of the PICSIL synthesis system is to provide an automated path from the output of the PICSIL editor to description of a chip layout. The development of new synthesis algorithms is not an objective. To assess any potential problems with using the system, an added objective was to have a chip fabricated from the output description.

To aid in the development of the PICSIL synthesis system, existing tools have been used where possible. As the expected user of the PICSIL system would be a computer engineer with little or no VLSI design experience, the overhead of having to learn to control one or more synthesis tools in addition to PICSIL was considered prohibitive. The PICSIL synthesis system controls the synthesis process and requires the designer to set only a handful of options at the start.

The PICSIL language does not have facilities for simulation. However this is an important part of the design process, so some form of interface needs to be provided to drive simulation.

The rest of this chapter discusses the selection of a synthesis path, the mapping of PICSIL constructs suitable for synthesis through the chosen path and automated driving of the synthesis process using the PICSIL Synthesis Manager.

6.1 Selection of Synthesis Path

Existing synthesis tools vary greatly in the portion of the synthesis process that they support and the types of system that they can synthesise. For example the FIRST system (Denyer and Renshaw, 1985) is limited to synthesis of designs in the digital signal processing domain, but provides a complete synthesis path. Other systems such as Olympus (De Micheli, Ku, Mailhot and Truong, 1990) can accept designs from a wide set of application domains, but only support high level and logic level synthesis.

Because of the limited budget associated with the research reported in this thesis, the selection of synthesis tools was essentially limited to those available in the public domain. A limited review identified the following synthesis tools available in the public domain: the FIRST system (Denyer and Renshaw, 1985), the VAST system (Kam and Hellestrand, 1990), a silicon compiler based on Asynchronous Architecture (Hirayama, 1986), the System Architect's Workbench (Thomas, Dirkes, Walker, Rajan, Nestor and Blackburn, 1988), the Olympus system (De Micheli, Ku, Mailhot and Truong, 1990) and Berkeley Octtools (Octtools, 1991).

None of the above tools was considered suitable alone for use in the PICSIL synthesis system as either the PICSIL constructs did not map well onto its input HDL, or it did not support the full synthesis path or both. A second approach was to consider the use of a combination of tools for use in the PICSIL synthesis system. Starting with high level synthesis, the Hirayama system, Olympus and the Systems Srchitect's Workbench were identified as being the most suitable, as most PICSIL constructs mapped well onto their input HDLs.

Because of the US Department of Defence rules, the System Architect's Work bench was not available outside the United States of America, and the Hirayama system would run only on a DEC 2060, not available at this institution. The Olympus system was the only remaining option.

Olympus is an integrated set of tools (see Figure 6.2) supporting high-level and logic-level synthesis starting from behavioural descriptions written in HardwareC. Most PICSIL constructs map well onto HardwareC constructs which allowed the magnitude of the PICSIL system development to be reduced enormously. However, data stores and routers do not, and have to be compiled directly to SLIF, one of the two intermediate languages used by the Olympus system.

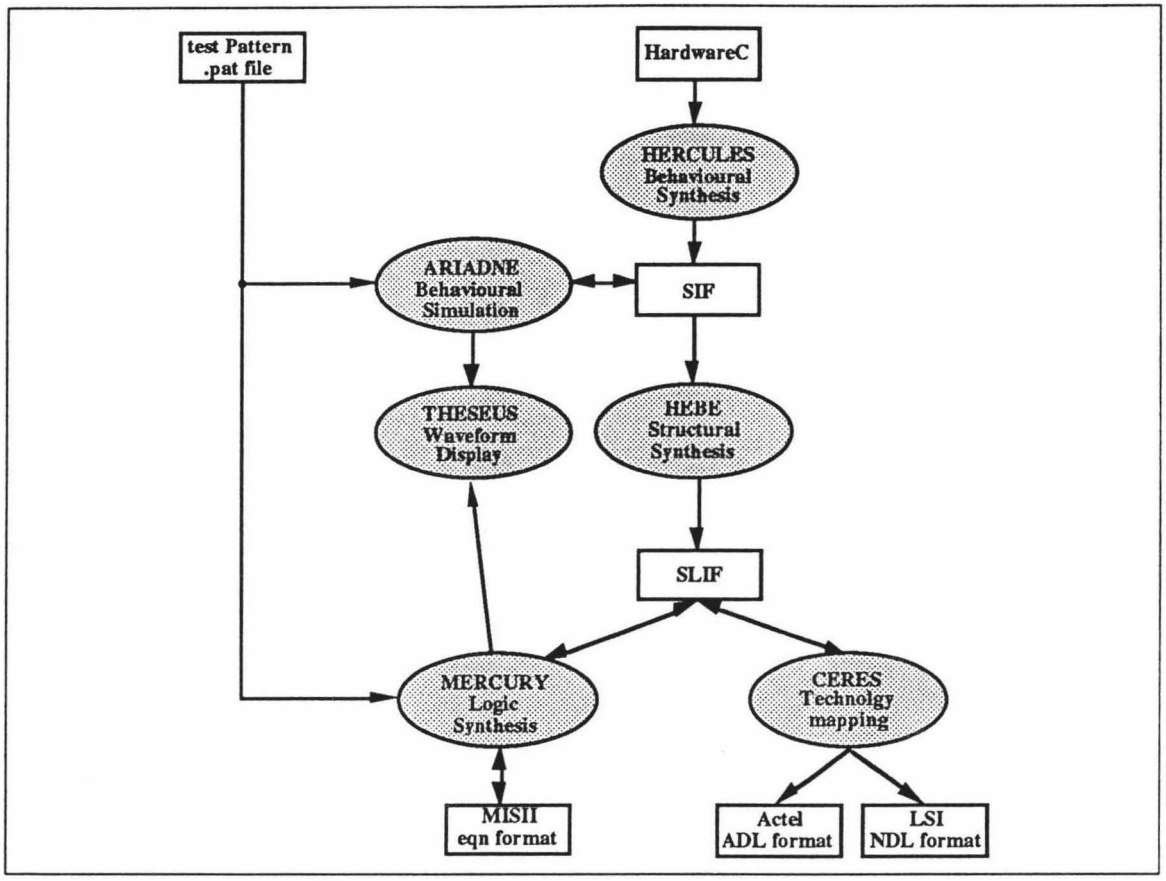


Figure 6.2 Block Diagram of the Olympus Synthesis System

The SLIF (*Structural/Logic Intermediate format*) format is used at the structural and logic levels and is a hierarchical net list language. An example SLIF description for the daisy chain arbiter in Figure 6.15 is shown in Figure 6.3.

```
.model arbiter;                                          # definition of model arbiter
   .inputs CLK pr1_rqt, pr2_rqt, pr3_rqt;                # inputs list
   .outputs pr1_lck, pr2_lck, pr3_lck, not_locked;       #outputs list
   .search pcell_file;                                   #file pcell_file may contain needed models

   pr1_freein = pr1_lck' pr2_lck' pr3_lck';              # logic equations: ' = NOT, + = OR and * =AND
                                                         # * may be omitted ie. a' * b <=> a' b
   .call  inst0 pcell(CLK,pr1_rqt, pr1_freein; ; pr1_lck, pr2_freein);  # instance definition. pcell described externally
   .call  inst1 pcell(CLK,pr2_rqt, pr2_freein; ; pr2_lck, pr3_freein);  # instance definition. pcell described externally
   .call  inst2 pcell(CLK,pr3_rqt, pr3_freein; ; pr3_lck, not_locked);  # instance definition. pcell described
externally
.endmodel arbiter;                                       # end definition (model arbiter)

file pcell_file
.model pcell;          #Externally called model. Calling model must have argument lists of correct size and order
   .inputs CLK rqt freein;
   .outputs lck freeout;

   lck = @D(lck_in, CLK);                                #register definition where Q = lck, D = lck_in and
                                                         # clock = CLK

   lck_in = (lck rqt) + (freein rqt);
   freeout = freein rqt';
.endmodel pcell;
```

Figure 6.3 Example SLIF description for daisy chain arbiter in Figure 6.15

In this example *logic statements* have the form var = expression (e.g. freeout = freein rqt';) where expression is a Boolean expression which defines the conditions for which var is true. An expression may also be used to define a latch @D(lck_in, clk) which has an input lck_in and is clocked by clk. *Call* commands (e.g .call inst0 pcell(CLK,pr1_rqt, pr1_freein; ; pr1_lck, pr2_freein);) are used to create an instance of the named cell (i.e. pcell) which may be described in the same file or in a file specified by a *search* statement (e.g. pcell_file).

Each of the tools within Olympus has been designed to be used either interactively, allowing the designer to experiment with the effects of different synthesis options, or in batch mode, allowing the synthesis process to be automated.

While no physical synthesis tools are provided within Olympus, one of the tools, Mercury, produces the input description of some commercial logic synthesis tools (De Micheli, Ku, Mailhot and Truong, 1990); LSI Logic's NDL format for "sea of gates" implementations and Actel's ADL format for electrically programmable gate array implementations. It also supports an interface to Berkeley's MISII combinational synthesis program (Brayton, Rudel, Sangiovanni-Vincentelli, and Wang, 1987) providing a partial entry point to UC Berkeley's Octtools design system.

The cost of both the software and production of a chip using the LSI or Actel logic design system prevented them from being considered for use in the PICSIL synthesis system (Anisimoff, 1992).

While the Mercury tool provided only a partial entry point into Octtools, Octtools was available almost free of charge and the fabrication of a chip from the output of Octtools was within the budget of this research, using the TINY chip in Orbit Semiconductors' FORESIGHT program (Orbit, 1991). Hence the Octtools suite of programs was selected to perform PICSIL's logic and physical synthesis.

To allow a full entry point into Octtools, parts of the Mercury program code used for producing a MISII description were used to create a program called to_oct (see Figure 6.4). This program reads in a SLIF description and outputs two files, root.eqn (describing the designs combinatorial logic) and root.bdnet (describing the design's registers) providing a full path into Octtools system.

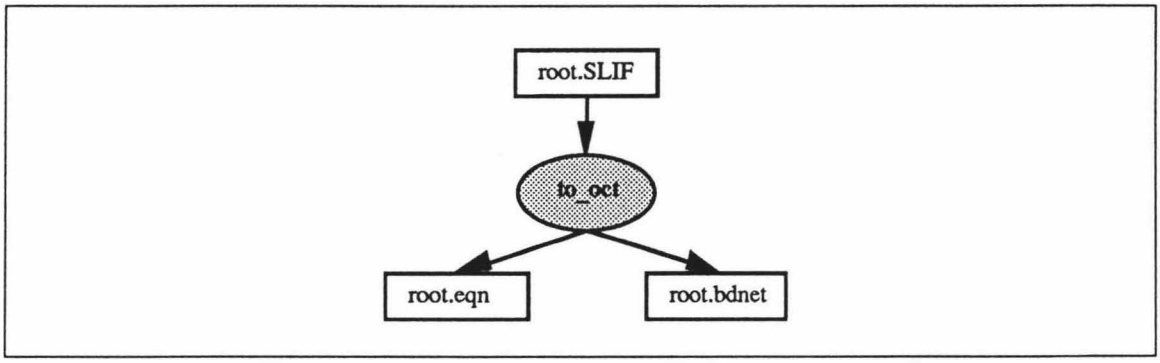


Figure 6.4 Conversion of a SLIF description in to descriptions suitable for input into Octtools using the to_oct program

Octtools (Octtools, 1991) is a collection of programs and libraries that form an integrated system for IC design. The system includes tools for PLA and multiple-level logic synthesis, state assignment, standard-cell, gate-matrix and macro-cell placement and routing, custom-cell design, circuit, switch and logic-level simulation, and a variety of utility programs for manipulating schematic, symbolic, and geometric design data. Only a subset of these tools is required to synthesise a chip layout from a SLIF description and a possible synthesis path is shown in Figure 6.5.

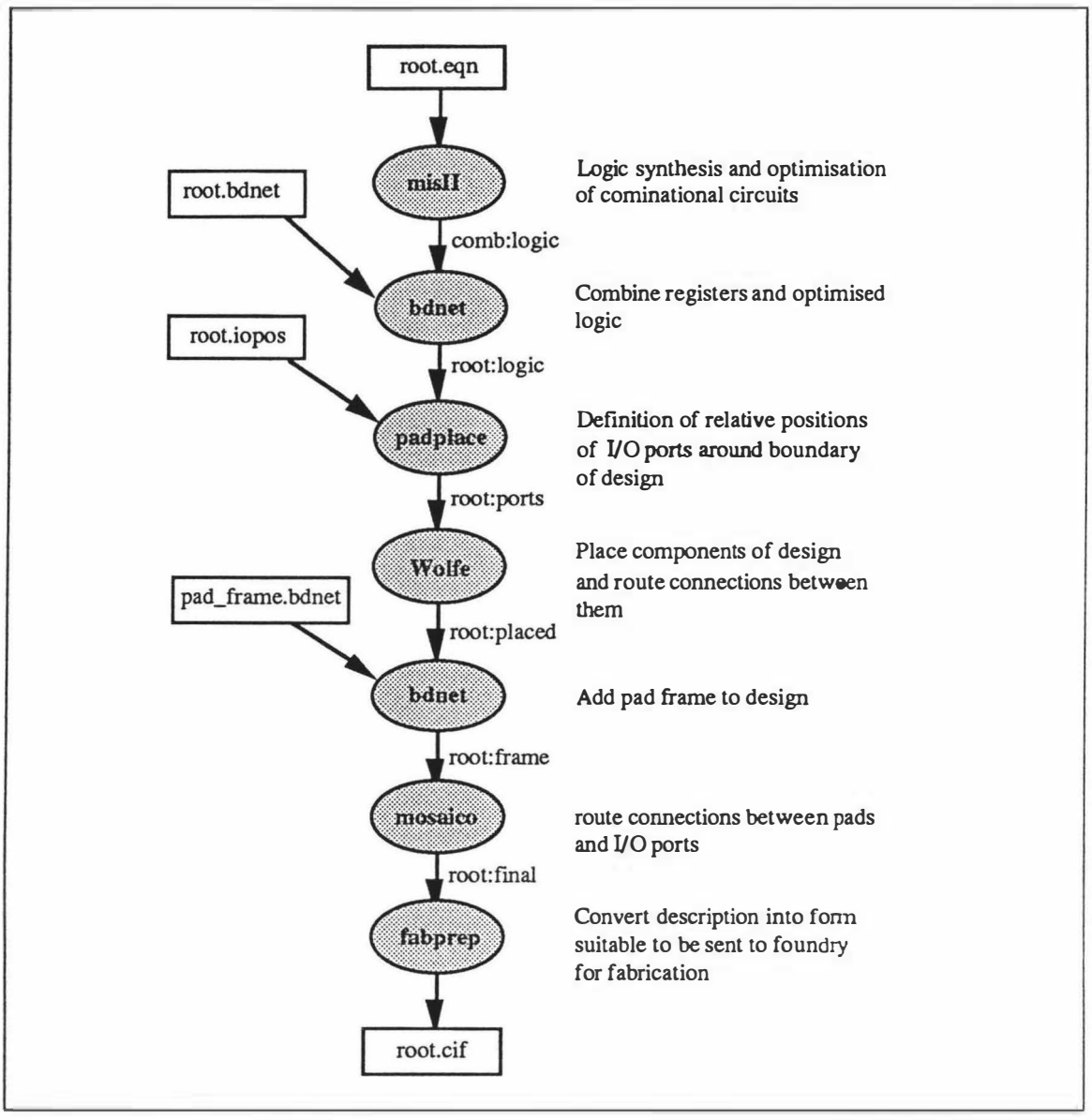


Figure 6.5 Logic and physical synthesis using Octtools

Each of the tools in the synthesis path in Figure 6.5 can be invoked with a single batch mode command.

To complete the physical synthesis of a design, bonding pads must be added so that connections may be made between the design's I/Os on the chip and the leads of the IC package. Octtools can either automatically place pads around the outside of the chip, or use a predefined fixed pad frame. As Orbit's TINY chip, the target implementation for PICSIL designs, is oriented towards a fixed pad frame, the PICSIL synthesis system follows this path.

6.2 Translation of PICSIL to Olympus descriptions

Before a PICSIL design can be synthesised by the Olympus and Octtools systems, the PICSIL data structure, which includes information derived from both DFDs and the Data Dictionary Entries (see Appendix 1), must be compiled into Olympus' input language, HardwareC. To simplify this process, PICSIL constructs have been chosen to map directly onto HardwareC constructs where appropriate.

At the top level both PICSIL (using data flow diagrams) and HardwareC (using block definitions - see page 16) represent a design as a set of concurrent component functions interconnected by communications channels. The functional components in PICSIL are primitive and non-primitive processes, data stores, control processes and routing devices, while in HardwareC they are processes and blocks. Most of the component types in PICSIL have satisfactory direct mappings onto HardwareC (see Table 1).

Data stores and routers are not directly representable in HardwareC, but experiments with the implementation of a variety of circuits (see Chapter 4) established their desirability. Accordingly, a program to compile them directly into SLIF was written, and its output is linked into the SLIF produced for the other types of component by the Olympus system.

PICSIL Object	Olympus representation
Non primitive process (DFD)	HardwareC block
Primitive Process	HardwareC process
Control Model	HardwareC Process
Data store	SLIF
Router	SLIF

Table 1 Mapping of PICSIL objects onto HardwareC models

To allow conventional techniques to be used in the compilation of PICSIL components into Olympus descriptions, the multi-dimensional PICSIL data structure representation is first translated into a linear textual form called LinearP.

Translation of the PICSIL Data Structure into LinearP

The translation of a PICSIL design into a linear form involves a traversal of its data structure representation, to extract all the relevant information into an intermediate language. This language, LinearP, comprises a set of objects exactly corresponding to the nodes in the PICSIL Data Structure. Each object is a self-contained textual item with a body comprising the corresponding node's textual data dictionary entry (primitive nodes) or links to its subcomponents (non-primitive nodes), and parameters to represent the flows which were attached to it in the Data Flow Diagram. No parsing of the textual data dictionary entries is performed during this stage. Figure 6.6 shows a user-level definition of an object and its LinearP representation.

As each node in the PICSIL data structure is visited, a check is made to ensure that all the details for the object the node represents have been entered. For example, a primitive process requires both a name and data dictionary entry to be present. If any of the details required for an object have been left out then an error message is reported back to the PICSIL editor so the user can be notified (see section 6.3).

Three of the items in PICSIL's diagrammatic vocabulary (external entities, links, and connectors) are tools which facilitate the construction of DFDs. They are redundant in a completed diagram, and can therefore be omitted from the LinearP representation. A fourth item, the group flow, is merely an abbreviation of a set of other flows, and is replaced by these in the translation into LinearP.

The non-primitive nodes in the PICSIL data structure are non-primitive processes, elements and controllers. In the case of non-primitive processes and elements, the body of the LinearP

description lists the components of the non primitive processes and their interconnections (see Figure 6.7(a)). It also contains declarations of the I/O channels (see Figure 6.7 (b)) which form the interface to the source DFD, and then the internal communications channels (see Figure 6.7 (c)).

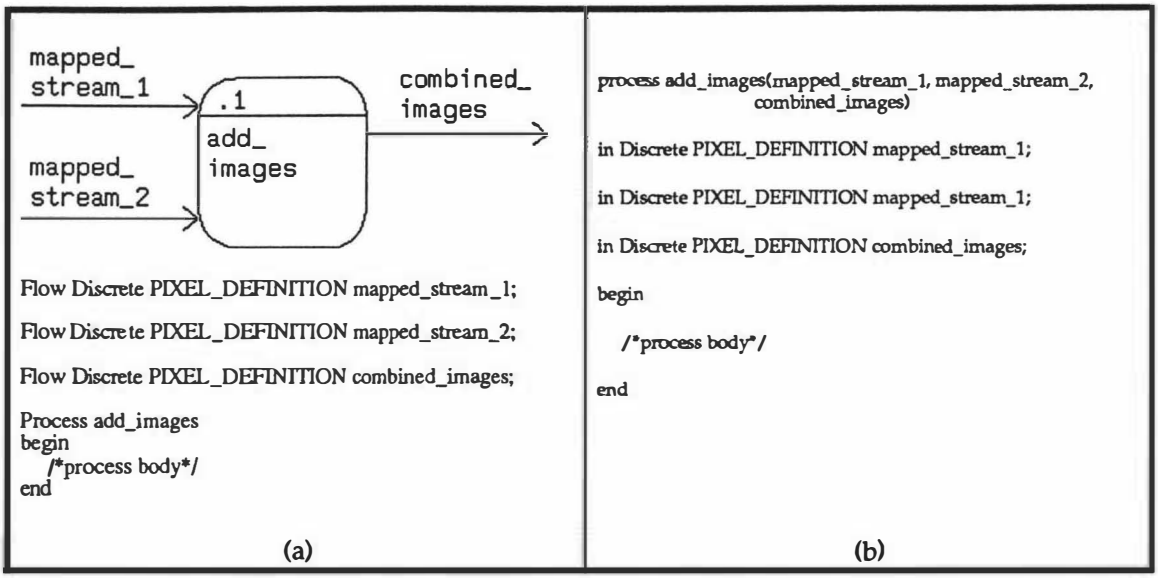


Figure 6.6 (a) User level definition of a process and attached flows and (b) its LinearP representation

```
dfd root(stream_image_1,input_clock,stream_image_2,new_threshold\value,new_threshold\ready,
        new_threshold\accept,status\outdated,status\updated,status\new_value, new_stream)
in continuous PIXEL_DEFINTION stream_image_1;
in continuous boolean input_clock;
in continuous PIXEL_DEFINTION stream_image_2;
in continuous PIXEL_DEFINTION new_threshold\value;
in continuous boolean new_threshold\ready;
out continuous boolean new_threshold\accept;
out event status\outdated;
out event status\updated;
in event status\new_value;
out continuous PIXEL_DEFINTION new_stream;                                        (b)

<
    discrete PIXEL_DEFINTION mapped_stream_1;
    discrete PIXEL_DEFINTION mapped_stream_2;                                     (c)
    /*declaration of flows associated with data store*/

    color_map(/*parameters associated with data store*/);
    perform_mapping_stream_1(stream_image_1,input_clock,mapped_stream_1,/*store parameters*/);
    perform_mapping_stream_2(stream_image_2,input_clock,mapped_stream_2,/*store parameters*/);
    combine_images(mapped_stream_1,mapped_stream_2,new_threshold\value,new_threshold\ready,
                   new_threshold\accept,status\outdated,status\updated,status\new_value,
                   new_stream);                                                   (a)
>
```

Figure 6.7 LinearP description for root DFD in image processing system described in Chapter 3.

A controller which is a non-primitive node in the PICSIL data structure is turned into a primitive object in its LinearP description. Each state in the state transition diagram that forms the definition of a controller is assigned a unique number. The states are then visited in turn

and all their details (including the process activations, and all the transitions to the next state including the event/action pair causing the transition) are extracted.

The compilation of LinearP into HardwareC is greatly simplified if object definitions (the size of data stores for example) occur in the LinearP description before their use. Figure 6.8 shows two examples of an adapted form of the Nassi-Shneiderman (Nassi and Shneiderman, 1973) chart representation to describe the algorithm used to extract objects from the PICSIL data structure in the correct order. In this adapted form of Nassi-Shneiderman chart, statements in a box are executed from top to bottom. A grey shaded area at the top of a box indicates the condition for which the box's contents are processed.

The first of the Nassi-Shneiderman charts shows that the first item extracted from the PICSIL data structure for inclusion into the LinearP description is the data dictionary appendix. Following that, LinearP descriptions are generated for each of the data stores and routers in the data structure. Lastly the rest of the objects in the data structure are extracted using the algorithm defined in the Nassi-Shneiderman diagram titled Diagram-definition. This algorithm recursively calls itself for each of the elements and primitive processes in the diagram, to create the LinearP definitions of all the object's children. After LinearP descriptions have been created for all the child diagrams, a LinearP description is created for the diagram's controller if it has one. LinearP descriptions are then created for each of the primitive processes in the diagram, followed by a definition of all of the components in the diagram and their interconnections.

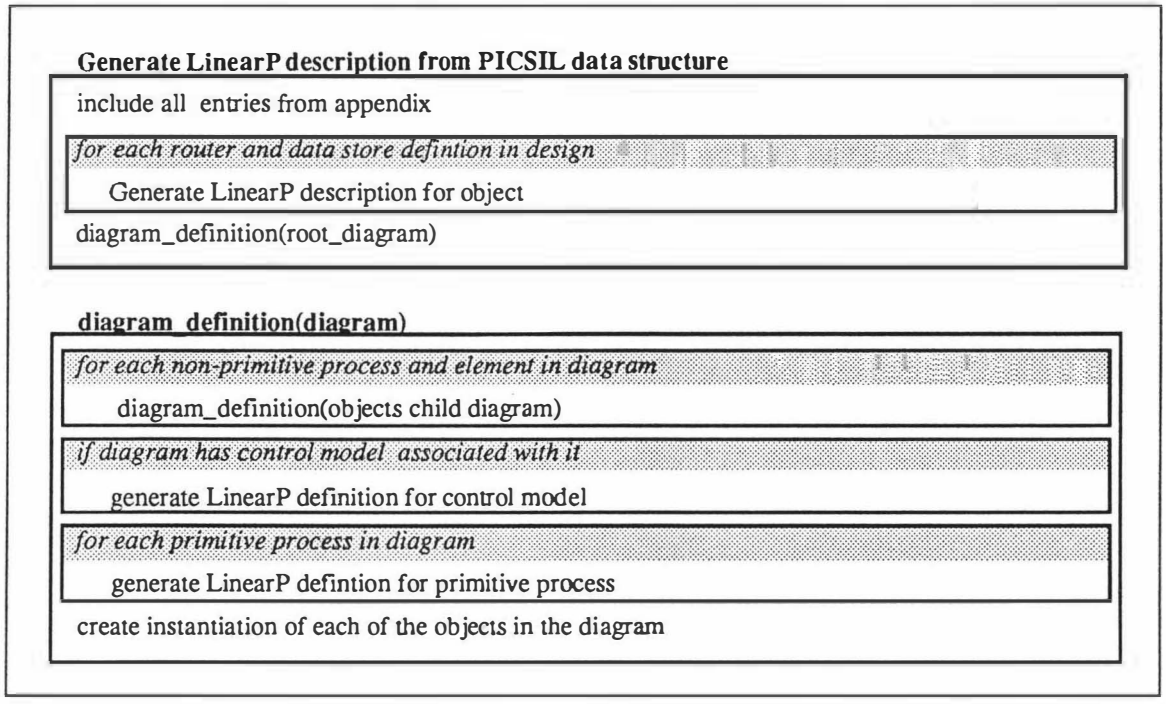


Figure 6.8 Algorithm used to generate LinearP from PICSIL data structure

Compilation into HardwareC

Once a design's LinearP description has been generated, and passed through the C pre-processor to remove comments and to expand #includes and #defines, the PICSIL compiler (created with lex (Lesk, 1975) and yacc (Johnson, 1975) translates it into HardwareC and SLIF.

The PICSIL compiler parses LinearP objects one at a time and generates the HardwareC code and any SLIF for each object before moving onto the next one. A symbol table is maintained in

which each entry contains the type, size and direction of a parameter. After a LinearP object has been compiled, only the information necessary to compile other objects is retained in the symbol table. Any syntax errors found by the compiler are reported back to the PICSIL editor for notification to the user (see section 6.3 page 121).

The following subsections describe the compilation of each of the different LinearP objects into their Olympus representation(s).

Data Declarations and Use

The basic data types supported in both LinearP and HardwareC are similar, although the format used is slightly different. The declaration and use of integer variables is identical in both LinearP and HardwareC so no conversion is required.

Boolean and static variables in LinearP and HardwareC are the similar, although their representation is different. LinearP Boolean and static variable declarations in the format type {lower_bound : upper_bound} name, are converted into HardwareC variables with the format type name[size], where size is defined as upper_bound - lower_bound + 1. If no bounds are given the size is assumed to be 1.

The way variables are referenced in LinearP is also slightly different from that in HardwareC. The addresses of the individual bits in a LinearP variable are in the range lower_bound to upper_bound, whereas the addresses of the individual bits in a HardwareC variable have a range between 0 and its size minus 1. To accommodate these differences, any LinearP references specifying a range of bits have the value of the lower_bound in the variable's declaration subtracted from them. Figure 6.9(a) shows a simple LinearP process which declares and uses a variable with a non zero lower_bound, and 6.9(b) the HardwareC code generated for it.

```
begin
   boolean{8:15} colour_1;
   boolean{16:23} colour_2;
      ...
   colour_1 {8:11} = colour_1{8:11} +
                     colour_2{20:23};
      ...
end
```

(a)

```
{
   boolean colour_1[8];
   boolean colour_2[8];
      ...
   colour_1[0:3] = colour_1[0:3] + colour_2[4:8];
      ...
}
```

(b)

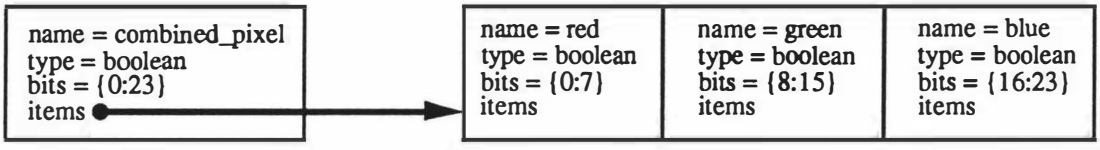
Figure 6.9 Example of variable declarations and references in (a) LinearP and (b) HardwareC

In addition to the basic data types, LinearP also supports the definition of new and more complex types through its *typedef* and *struct* constructs, which require mapping into HardwareC constructs.

This mapping makes use of HardwareC's ability to allow sub-ranges of a variable to be referenced (e.g. variable[3:5]). When the PICSIL compiler encounters a variable as a structure, it generates a HardwareC variable large enough to contain all the components in the structure. Extra information is stored in the symbol table to relate the components of the HardwareC variable to the corresponding items in the LinearP structure (see Figure 6.10(b)). Figure 6.10(c) shows the HardwareC generated by the PICSIL compiler for the PICSIL block in Figure 6.10(a).

```
seqbegin
    struct {
        boolean{0:7} red;
        boolean{0:7} green;
        boolean{0:7} blue;
    } pixel_1, pixel_2, combined_pixel;
        ...
    combined_pixel.red = pixel_1.red + pixel_2.red;
    combine_pixel.green = pixel_1.green + pixel_2.green;
    combined_pixel.blue = pixel_1.blue + pixel_2.blue;
        ...
seqend
```

(a)



(b)

```
[
    boolean pixel_1[24];
    boolean pixel_2[24];
    boolean combined_pixel[24]
        ...
    combined_pixel.[0:7] = pixel_1[0:7] + pixel_2[0:7];
    combine_pixel.[8:15] = pixel_1[8:15] + pixel_2[8:15];
    combined_pixel[16:23] = pixel_1[16:23] + pixel_2[16:23];
]
```

(c)

Figure 6.10 Representations of Structures in (a) LinearP, (b) symbol table and (c) HardwareC

When the declaration of a typedef is encountered, an entry is placed in a separate symbol table set aside for new type definitions. As each identifier is read in by the lexical analyser, the type definition symbol table is checked to see if the identifier has been declared as a type. If the identifier is a user-defined type and the parser is parsing a type, then the type definition found in the type symbol table is copied into the new variable's entry.

Primitive and Non primitive processes

LinearP non-primitive processes (DFDs), primitive processes, functions and procedures have an almost direct mapping onto HardwareC blocks, processes, functions and procedures respectively.

The only differences between LinearP non-primitive processes and HardwareC blocks is the format of the declarations, as discussed in the previous section.

The main difference between PICSIL primitive processes, functions and procedures, and their HardwareC equivalents is that different symbols in PICSIL are used to indicate the beginning and end of blocks and the I/O statements.

The HardwareC generated for I/O statements associated with data stores, routers and controllers is discussed in the section dealing with those objects.

Incorporating Data Stores into an Olympus design

Although it was necessary to generate SLIF for data stores independently of the Olympus system, some aspects of their interaction with other components of a design could be designed in HardwareC. In particular, the array of memory modules in a data store and the arbitration circuitry necessary to prevent conflict were generated independently; store I/O statements and data, address, control and arbitration flows, were incorporated into the standard HardwareC definition.

Data, address, and control flows to data stores would be most naturally represented by the bidirectional tri-state communications channels available within Olympus. However, Olympus does not handle tri-state buffers correctly when generating the SLIF for multi-process systems. Instead, data store communication has been incorporated into the Olympus paradigm using unidirectional links (see Figure 6.11). Note that the data_out flow, although sent to two destinations, is written to only by the store, and is read only by one process at a time, so despite its appearance it is not truly bidirectional.

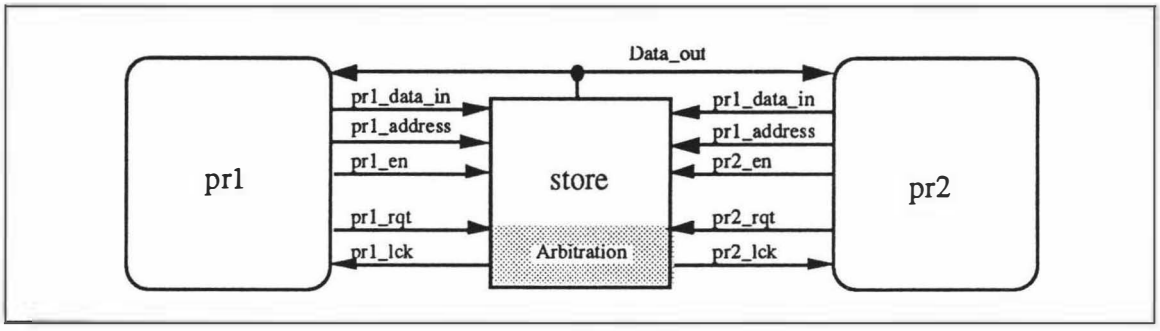


Figure 6.11 Block diagram for store representation not using bidirectional links

In the absence of tri-state buffering, well-defined data input to the store is achieved by ORing the store flows from all the processes which write to it (see Figure 6.12). The output of each OR gate will follow the input of the process that has access to the data store.

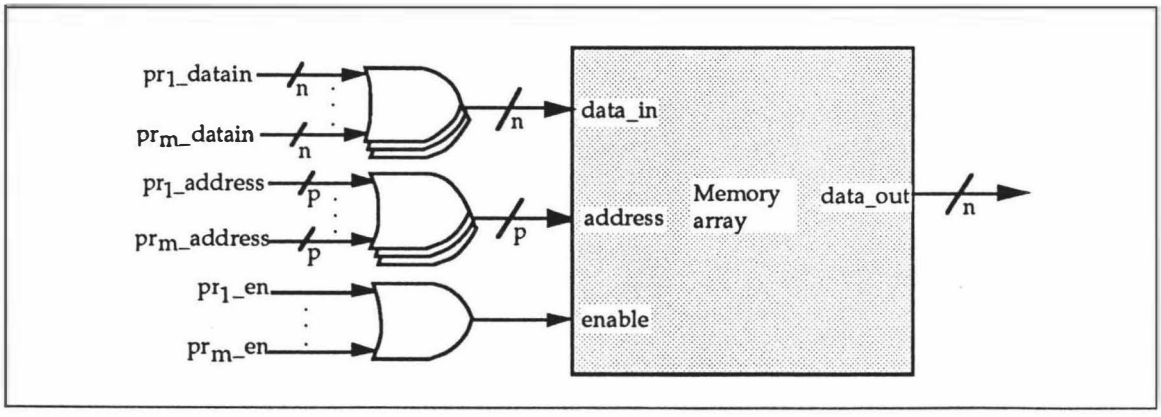


Figure 6.12 Corresponding bits of each store flow ORed together to produce a single store flow input to the memory array

As HardwareC does not have any constructs to represent memories or arrays, no natural representation for a data store exists in the language. One possible representation of a store in HardwareC is a process with a static variable declared for each memory location. The different memory locations can then be accessed using two switch statements, one for reading and one

for writing. Figure 6.13 shows a HardwareC process definition for a 4 word by 3 bit memory based on this model.

```
process store(address, data_in, data_out,en)
    in port data_in[3], address[2], en;
    out port data_out[3];
[
 static val_0[3], val_1[3], val_2[3], val_3[3];
 boolean new_add[2],new_dir,
          new_data_in[3],new_data_out[3];
 new_add = read(address);
 <
  if (en) [
   new_data_in = read(data_in);
   switch (new_add) {
   case 0b00:val_0 = new_data_in;break;
   case 0b01:val_1 = new_data_in;break;
   case 0b10:val_2 = new_data_in;break;
   case 0b11:val_3 = new_data_in;break;
   }
  ]
  [
   switch (new_add) {
   case 0b00:new_data_out = val_0;break;
   case 0b01:new_data_out = val_1;break;
   case 0b10:new_data_out = val_2;break;
   case 0b11:new_data_out = val_3;break;
   }
   write data_out = new_data_out;
  ]
 >
]
```

Figure 6.13 HardwareC representation of four word three bit memory

To ascertain the practicality of representing a data store this way, the HardwareC description of the 4X3 memory was synthesised using Olympus and Octtools to obtain a layout which was 0.9 mm by 1.1 mm using a 2 micron feature size. Because of the very large size for a very small memory this approach was considered to be unacceptable.

The second approach considered was to use an independent memory module generator. A number of RAM/ROM module generators exist (Li and Hellestrand, 1990 and Cheng and Mazor, 1988) which take as input a number of parameters such as the number of words, and the number of bits per word. The memory module generator discussed by Li and Hellestrand shows a 1K bit DRAM requiring 1.8 mm by 1.6 mm in a 2µm cmos process, which represents a significant improvement over a HardwareC representation. A major disadvantage with the use of memory module generators in the PICSIL synthesis system is that simulation is not possible until the physical synthesis stage is reached, which was considered unacceptable. Also no memory module generators could be found that could be integrated directly into Olympus or Octtools.

A compromise between the two approaches is to represent memory using registers, and AND, OR and NOT gates, which corresponds to the level of description provided by the SLIF language in Olympus. Figure 6.14 show a representation of a memory described using these gates. In this diagram each row represents one of the four three-bit words. An address must be provided at least one system clock cycle before a read or write operation can occur. A read operation occurs on the positive edge of the en signal. Synthesis of the 4 X 3 memory resulted in a layout 0.5 by 0.6 mm using a 2 micron feature size.

This representation is very regular and expands to large memories, making it possible to automatically generate a SLIF description of an arbitrary memory given the number and size of the words. While it does not produce an efficient implementation of memories, it is practical for small memories and could be incorporated into the chosen synthesis path.

An arbitration device prevents more than one process accessing a data store a time. Two lines for the purposes of arbitration are provided in the implementation of store flows: request (<process_name>_rqt) and lock (<process_name>_lck). Once a process wishes to read from or write data to a data store it sets its request line to true. The process then waits for the lock line

to go true, giving it control of the data store. When the process finishes accessing the data store it sets the request line false again. Any other process waiting to access the data store can then be granted permission to do so. A suitable arbitration device based on (Mano, 1982) for a data store is shown in Figure 6.15.

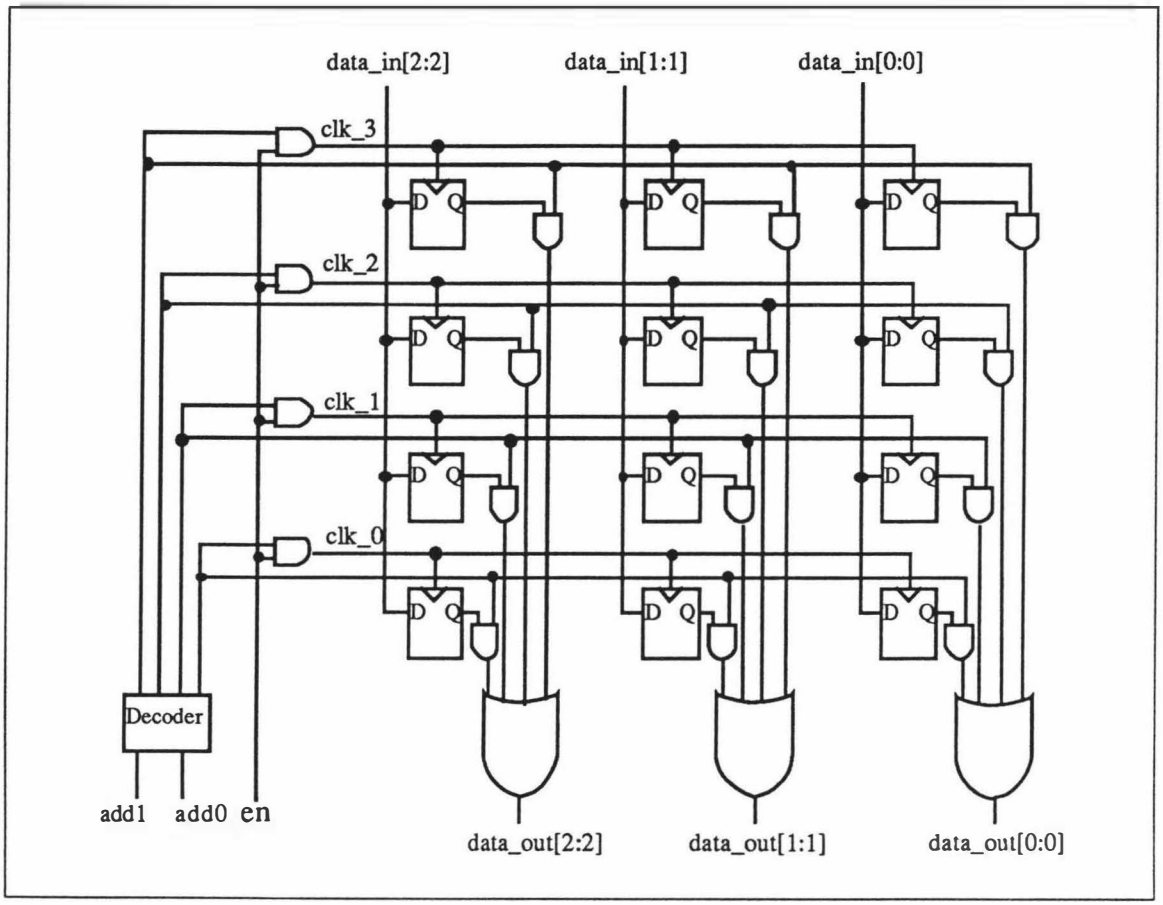


Figure 6.14 Logic Diagram for 4 X 3 Memory (after Tanenbaum, 1990)

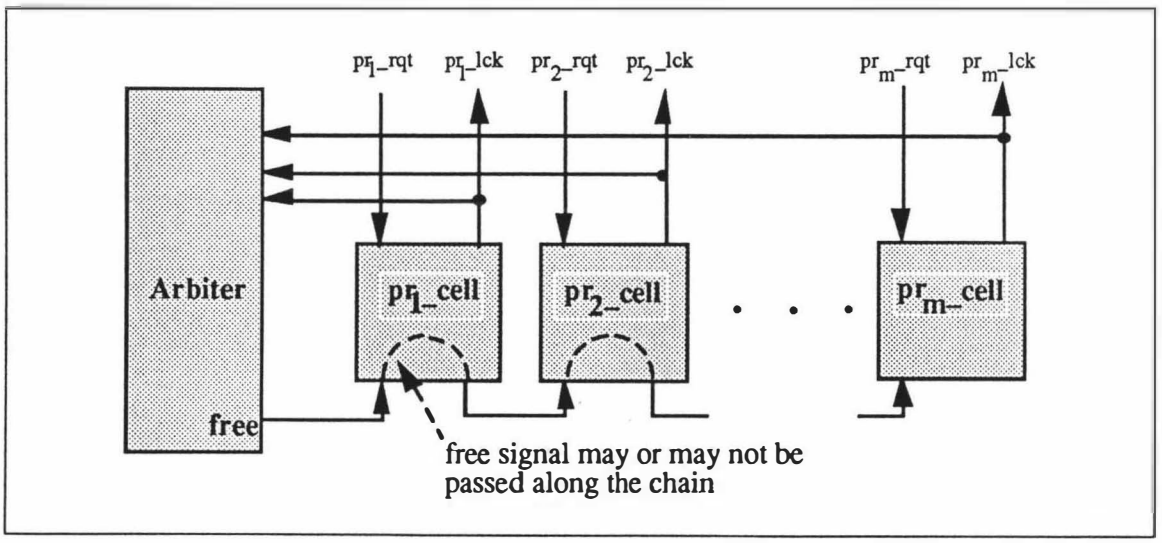


Figure 6.15 Data store arbiter using daisy chaining

When each of the lock lines fed into the arbiter is false, the arbiter outputs a true (or active) free signal. The free line is wired through all each of the cells. If both a request and free input to a

cell are active, the cell becomes the active cell and sets its lock output to true causing the arbiter to set its free output false. The cell remains active until its request input goes false again, at which time its lock output goes false allowing the arbiter to make its free output active. When two or more cells receive active request inputs at the same time, the one closer to the arbiter will become active, as an active cell prevents the free signal from being passed further down the chain. To prevent races, the values of the lock outputs can only change on the positive edge of the system clock, and it is assumed that the request inputs will only change soon after the positive edge. Figure 6.16 shows the circuits used for the arbiter and the each of the cells and Figure 6.17 shows the algorithm used to generate the arbitration device for a data store.

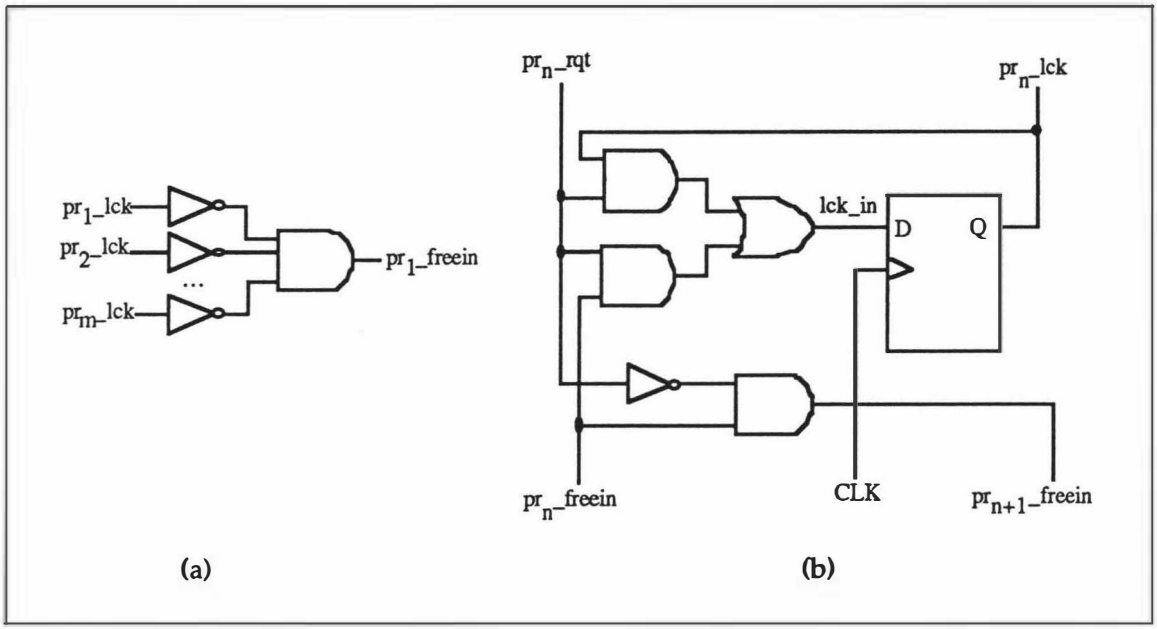


Figure 6.16 Circuits generated for (a) arbiter and (b) each instance of a process cell for the store arbitration device

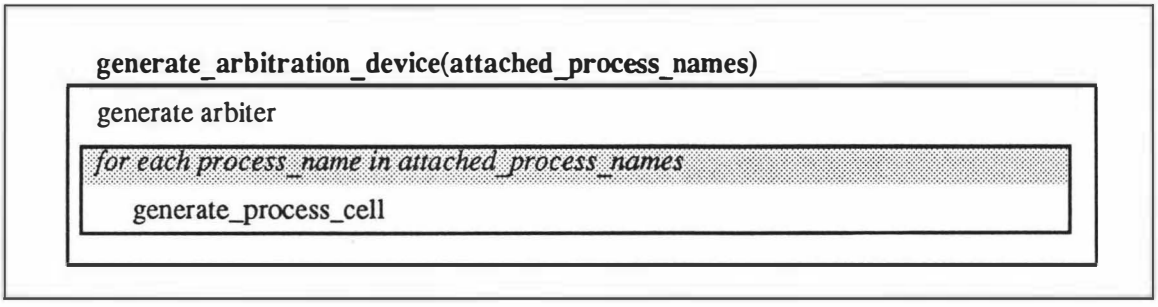


Figure 6.17 Algorithm used to generate arbitration device for data store

A single SLIF description is produced for each data store in a design including the memory element and arbitration device (see Figure 6.18). To complete the description, the inputs and outputs (parameters) of the data store description must be defined. If a process only uses a data store in read-only mode (ie. has a unidirectional store flow from the data store to the process), no inputs are generated for data_in and enable. Lastly the description requires a single set of inputs for the memory array generated by ORing the corresponding bits of the address, data_in and en inputs from each process attached to the data store.

The SLIF description is stored in a file called "<store_name>_Cnone.SLIF" for incorporation into the SLIF description generated by the Olympus system for the rest of a design.

Unfortunately, the HardwareC language does not provide features to allow a SLIF description to be included as part of a specification. To overcome this problem, the PICSIL synthesis manager generates a dummy process for the data store with the same name, inputs and outputs as its SLIF counterpart generated by the PICSIL synthesis. When the dummy process is synthesised, the SLIF generated for it by the OLYMPUS system is replaced by the SLIF generated by the PICSIL synthesis system for the data store. This will be discussed in more detail in Section 6.3.

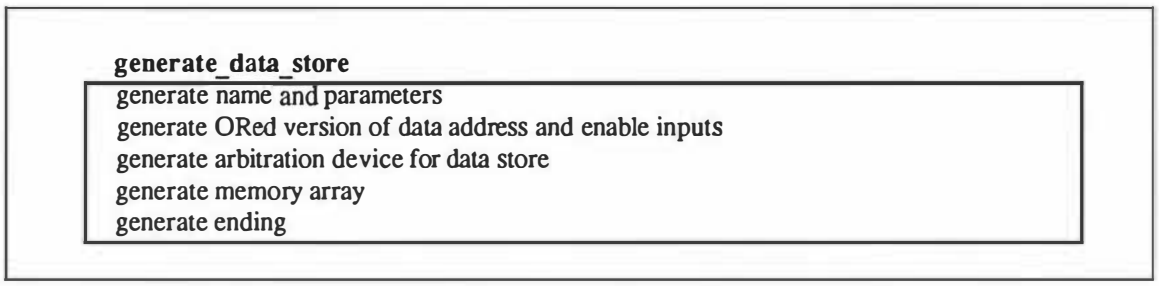


Figure 6.18 Algorithm used to generate data store

The last phase of incorporating a data store into an Olympus design is the compilation of store flows and store I/O statements into HardwareC. In HardwareC, a store flow is represented as a number of ports (sets of lines) dependent on the direction of the store flow. A bidirectional store flow (ie. read and write) between a process and data store is represented by the ports data_in, data_out, address, en, request and lock. A unidirectional store flow from a process to a data store (ie. write-only) is the same with the data_out port is omitted, while a unidirectional store flow from a data store to a process (ie. read only) has the data_in and en ports omitted.

The PICSIL data dictionary language has three store I/O statements: *stread*, *stwrite* and *unlock*. Each of these statements is compiled into a series of HardwareC statements to implement their functionality and drive the I/O lines associated with the store flow.

Before a data transfer between a process and a data store can occur using a *stwrite* or *stread* statement, the process must first lock the data store for it sole use. After the data store has been locked, the process then writes the address of the store location that data is to be read from or written to, and one clock cycle later the transfer can take place. If the store flow between the process and data store indicates that the process can use the data store in read-only or write-only mode, then the data store is freed by asserting all its outputs including setting the rqt signal false. If the process uses the data store in read/write mode then the data store is not freed until either the process finishes its current execution or an explicit unlock statement is executed. Figure 6.19(a) shows an example of a LinearP process definition containing both stread and stwrite statements. Figure 6.19(b) shows the HardwareC code generated by the PICSIL compiler for the LinearP process, with the code generated for the stread statement in (i), and the code generated for the stwrite in (ii).

When a process using a data store in read-write mode finishes its current execution, it sets all of its outputs which are attached to the data store to false, allowing another process to gain access to it. A process using a data store in this mode may also release the data store before the end of the current execution by executing a *free* statement which also causes all outputs attached the store to be set to false.

```
PROCESS .1 countvalues
seqbegin
  boolean{0:7} current_count;
  boolean{0:7} index;

  receive(value,index);
  stread(counts[index],current_count);
  current_count++;
  stwrite(counts[index],current_count);
seqend
```

(a)

```
process countvalues (counts_data, counts_dout, counts_dir,
        counts_add, counts_rqt, counts_ack, counts_en, value)
in port counts_data[8];
out port counts_dout[8];
out port counts_dir;
out port counts_add[8];
out port counts_rqt;
in port counts_ack;
out port counts_en;
in channel value[8];
[
[
boolean current_count[8];
boolean index[8];
index = receive(value);
[
write counts_rqt = 1;
while(!counts_ack);
<
write counts_add = index;                    (i)
write counts_dir = 0;
>
write counts_en = 1;
current_count = read(counts_data);
write counts_en = 0;
]
current_count++;
[
write counts_rqt = 1;
while(!counts_ack);
<
write counts_add = index;
write counts_dir = 1;                        (ii)
write counts_dout = current_count;
>
 write counts_en = 1;
 write counts_en = 0;
]
]
<
write counts_rqt = 0;
write counts_add = 0;
write counts_dir = 0;
write counts_dout = 0;
>
]
```

(b)

Figure 6.19 (a) LinearP process containing *stread* and *stwrite* statements and (b) the HardwareC generated by the PICSIL compiler for the process.

Routers

Although several attempts were made to implement a router in HardwareC, no satisfactory representation was developed, leading to an implementation directly in SLIF. However, a satisfactory representation in HardwareC was found for aspects of a router's interaction with other components of a design.

A router is a "programmable" switch which provides connections, from its import flows to its export flows. A controller sets up and maintains the "programming". Although it can provide multiple simultaneous connections, each import and export flow can be a part of only a single

connection. The switch has to be capable of switching discrete flows' handshaking signals as well as their data signals.

When a process wishes to send data to another process through a router, it first makes a request to the router's controller for a connection from an export flow from the router to the other process. If the requested export flow is part of another connection, the requesting process's execution is suspended until the export flow becomes free. Once the requested export flow becomes available, the controller will direct the switch to set up the connection and then acknowledge the process's request. Thereafter, the process may send data through the router. The connection is maintained until either the process finishes its current execution or it executes an unlock statement.

In a PICSIL design, the discrete flows imported by a router do not include any lines for allowing connections to be set up and broken, so these are supplied by the PICSIL compiler. Extra lines are required for exchanging request, grant, and destination information between a process and a router (see Figure 6.20 which shows a router with *i* import and *e* export flows).

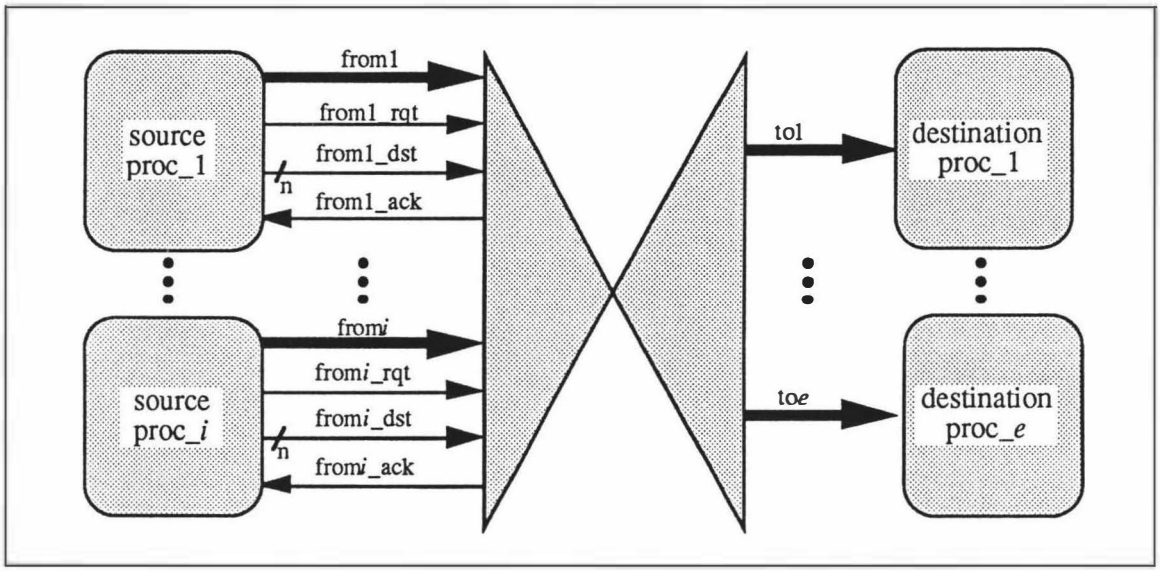


Figure 6.20 Block diagram showing extra lines that have to be added to allow a process to make connections through a router

Each of the flows exported from a router is given an id by the PICSIL compiler. When a lock statement is encountered by the PICSIL compiler, it generates HardwareC code to output the id of the destination export flow on the _dst lines. The code generated then causes the _rqt line to be set to true, and suspends the process's execution until a true _ack signal is received from the router indicating that the connection has been set up. When a unlock statement is encountered, HardwareC code is generated to break the connection by setting the _rqt line to false.

A suitable circuit for the controller (see Figure 6.21) is a two dimensional version of the arbitration device used for data stores. Each row represents one of the flows exported by the router, while each column represents one of the flows imported by the router. The arbiter and from*n*_to*m* cells respectively are the same as the arbiter and process cell respectively shown in Figure 6.16. The algorithm used to generate SLIF for a router is shown in Figure 6.22.

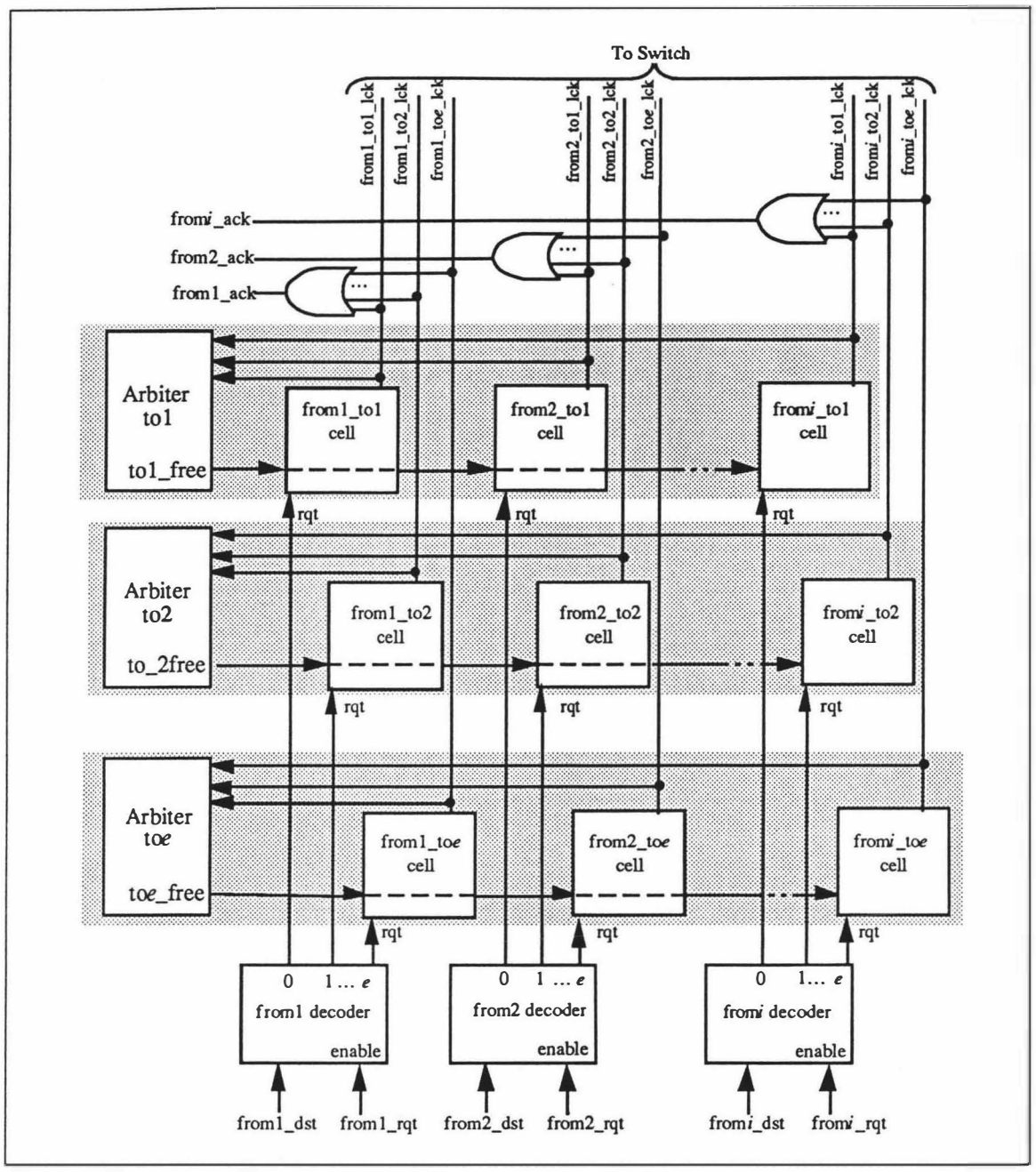


Figure 6.21 Controller for a router with *i* import and *e* export discrete flows attached.

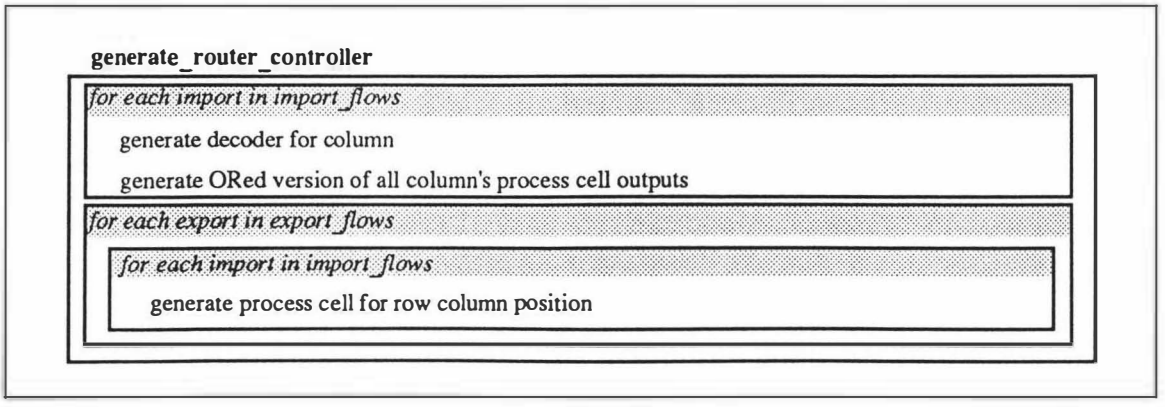


Figure 6.22 Algorithm used to generate SLIF for a routers controller

The second part of the router is the switch to direct any of the import discrete flows onto export discrete flows. The maximum number of connections that can exist in the switch at a time is the smaller of the number of import or export flows.

In addition to switching the data part of a discrete flow, any handshaking lines added during the synthesis of discrete flows into SLIF must also be handled by the switch. Discrete flows are mapped onto HardwareC channels, which, when synthesised by Olympus, have two extra lines added for handshaking. These lines are <flow_name>_rq in the same direction, and <flow_name>_ak in the opposite direction as the data flow. Figure 6.23 shows a circuit to implement the switching of import flows, including handshaking lines to export flows .

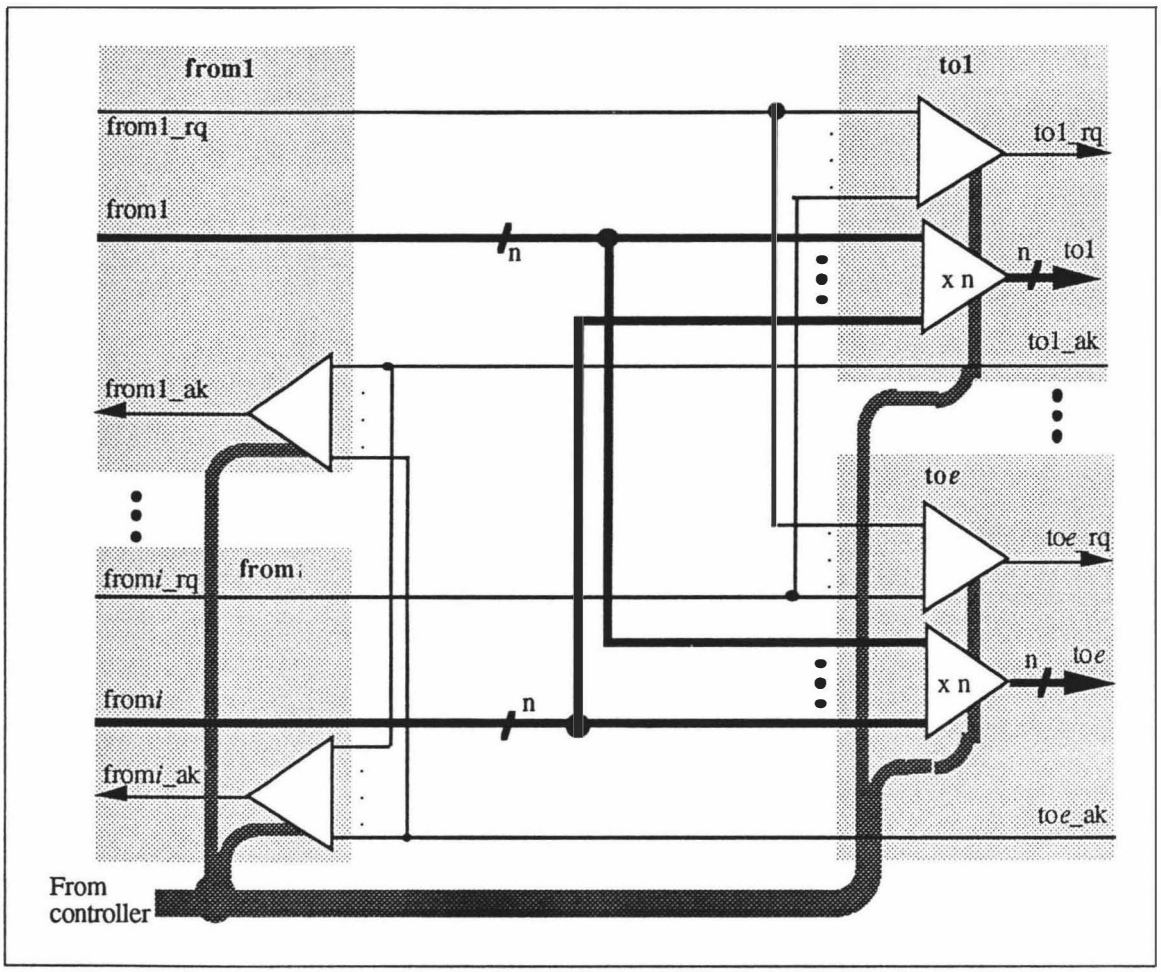


Figure 6.23 Circuit to implement switching part of a router

The switch is made up of a series of blocks (represented by the shaded areas), one for each import flow down the left hand side of the diagram, and one for each export flow down the right hand side of the diagram. Each of the output lines of a block (the ak lines in the import blocks and the data and rq lines in the export blocks) uses a multiplexer to select the value to output.

The multiplexer select lines are outputs of the router's controller. Each multiplexer has a select line for each of its signal inputs and the output of the multiplexer follows the value of the selected input signal. When a connection is made from an import block to an export block, the same select line will be active on all multiplexers in both the blocks so that the two flows will be connected. Figure 6.24 shows the circuits and naming conventions used in the generation of the multiplexers in both the import and export blocks and Figure 6.25 shows the algorithm used to generate SLIF for the router switch.

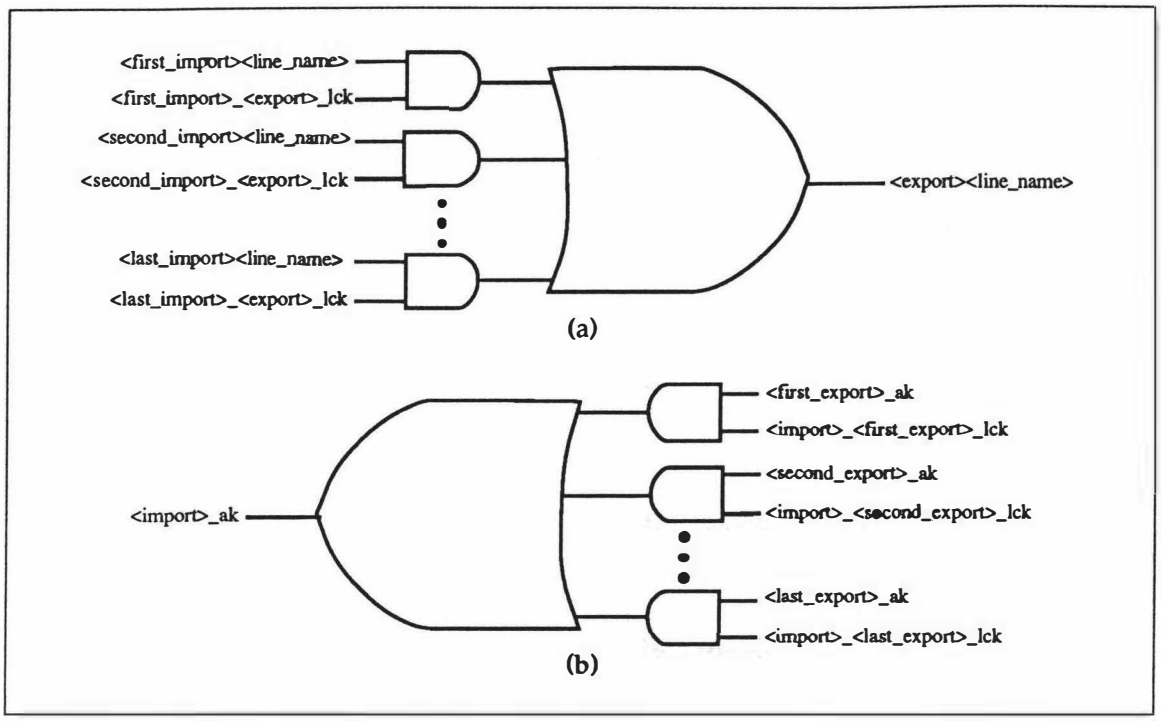


Figure 6.24 Circuits and naming conventions for (a) export and (b) import Multiplexers in Router switch

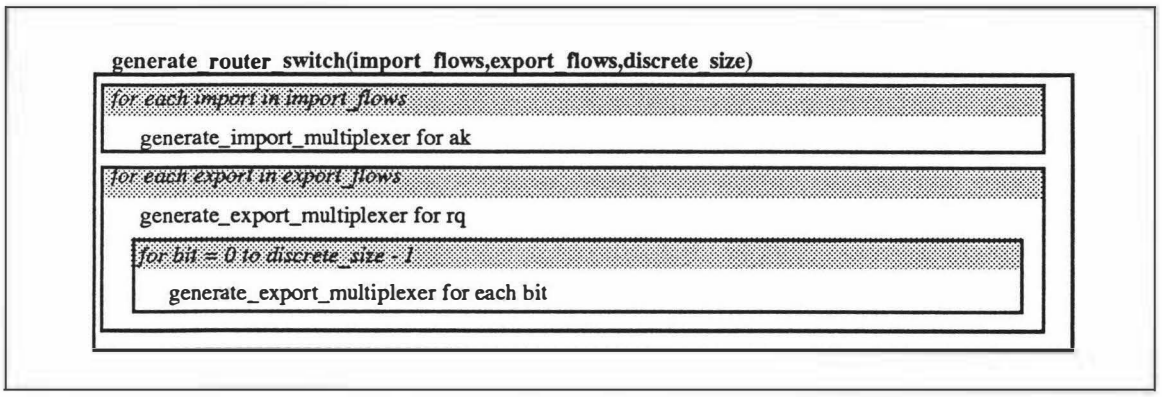


Figure 6.25 Algorithm used to generate SLIF for a router switch

Just as with data stores, a dummy process is included in the HardwareC description for each router in a design to allow the SLIF description to be incorporated into Olympus' output. A single SLIF description is produced for each router in a design including the parameters, the controller, and the switch. This SLIF description is stored in a file called <router_name>_Cnone.slif for incorporation into the SLIF description generated by the Olympus system.

Processes which export discrete flows for import by routers are modified to allow them to interact with routers in the Olympus representation. First parameters are added to the process to provide the lines necessary for establishing a connection through a router. For each discrete flow (<flow_name>) exported by a process and imported by a router, three parameters (HardwareC ports) called <flow_name>_rqt and <flow_name>_dst, in the same direction as the discrete flow, and <flow_name>_ak, in the opposite direction, are added.

The second modification is the compilation of the two PICSIL data dictionary statements for making and breaking connections with a router into HardwareC statements to implement their

functionality. The lock statement (ie. lock(<export_flow>,<import_flow>);) requests a connection from the router and then suspends the process's execution until the connection becomes available.

To break a connection through a router using an unlock statement (ie. unlock(export_flow>);) a process sets its _rqt and _dst lines to zero indicating to the router's controller that the process wishes to break the connection. For each process that uses router lock and unlock statements, the functionality of the unlock statement is inserted by the PICSIL compiler as the last statements executed by the process. Figure 6.26(a) shows an example PICSIL process and Figure 6.26(b) shows the HardwareC process generated by the PICSIL compiler for a process connecting to a router defined by the designer. Figure 6.26(b)(i) shows the HardwareC code generated for the PICSIL lock statement in Figure 6.26(a)(i). Figure 6.26(b)(ii) shows the statement inserted at the end of the process definition to break an outstanding connection to the router.

```
process receive_2
seqbegin
  boolean{0:1} new_char;
  while ( !clk_2);
  new_char = read(in_2);
  switch (new_char) {
  case 0b01 :
    lock(from_2, to_1);
    break;
  case 0b10 :
    lock(from_2,to_2);
    break;
  }
  do seqbegin
    while (clk_2);
    while ( !clk_2);
    read(in_2, new_char);
    send (from_2, new_char);
  seqend while (new_char != 0b11) ;
]
```

(a) PICSIL process

```
process receive_2 (from_2, from_2_rqst, from_2_add,
        from_2_ackn, in_2, clk_2)
out channel from_2[2];
out port from_2_rqst;
out port from_2_add[1];
in port from_2_ackn;
in port in_2[2];
in port clk_2;
[
boolean new_char[2];
while ( !clk_2);
new_char = read(in_2);
switch (new_char) {
case 0b01 :
< write from_2_add = 1; write from_2_rqst = 1;
  while (!from_2_ackn); >                    (i)
break;
case 0b10 :
< write from_2_add = 0; write from_2_rqst = 1;
  while (!from_2_ackn); >
break;
}
do {
while (clk_2);
while ( !clk_2);
new_char = read(in_2);
send (from_2, new_char);
} while (new_char != 0b11) ;
write from_2_rqst = 0;                       (ii)
]
```

(b) HardwareC process

Figure 6.26 (a) Example PICSIL process which makes connections to a router and (b) HardwareC generated by PICSIL compiler for PICSIL process in a

Incorporating Controllers into an Olympus design

While the PICSIL controller does not have a parallel in HardwareC, state transition diagrams and aspects of their interaction with other components of a design can be represented in HardwareC.

Event flows and the report statement in the PICSIL data dictionary language allow a controller to interact with other components of a design and require a representation in HardwareC. Continuous and discrete event flows are represented as one-bit ports and one-bit channels respectively. The *report* statement which reports an event along a discrete event flow can be

replaced by send(<discrete_event_flow>, zz_one); where zz_one is declared as a boolean variable and assigned a value of one.

A PICSIL controller can be implemented in HardwareC as a process specifying four main operations: monitoring input events, performing state transitions, performing actions when state transitions occur, and activating processes. These four main operations remained constant through a number of development iterations.

The monitoring of inputs events involves receiving any new discrete events and storing them for later use in a state transition, and evaluating the event equations defined for the controller. To allow discrete events to be stored, each discrete event flow imported by a controller has a one-bit static variable (zz_<event_flow_name>) associated with it, which is set when an event occurs on the discrete event flow. The value of the variable is not cleared until it is consumed in a state transition.

The event equations defined for a controller are evaluated using the stored values of the discrete event flows and the current values of the continuous event flows. These event equations are evaluated in parallel and are in the same form as defined in the PICSIL definition except that the names of the discrete events are replaced by the names of the static variables associated with them.

The current state of a controller is stored in a static variable called state which is updated every time a state transition occurs. For each state a nested multiway branch is used to test the values of the events associated with each transition. If a transition can occur, then the current state and process activations are updated, discrete events causing the transition are consumed by clearing the value of the static variable associated with them and the necessary actions are performed.

Continuous actions are performed by writing the value of the action (0 or 1) to the port associated with the action, while discrete actions are performed by sending a message on the channel associated with the action. A number of optimisations in the implementation of controllers were required to decrease the cost parameter (a relative estimate of the area of the circuit produced) output by Olympus during high level synthesis.

When a new control state is entered, the process activation table for the state defines the current activation for each of the processes in the data flow diagram. Any process deactivated on entry into a new state should have its current execution terminated immediately. HardwareC does not allow a process to be explicitly disabled, and this can only be achieved with extra control lines between the controller and the process, and extra code in the process. Because it is not practical to insert extra code between each of the process's statements a process's execution can only be terminated as a process begins a new execution. When the process starts a new execution, the inserted code follows a predefined protocol to determine when it can continue execution of the rest of its body.

In the initial representation, the type of control line used between a controller and process depended on the type of the process to be activated. If a process was both deactivated and activated continuously in different states a single bit port was used. When the control line (say zz_<process_name>) was active, the process was enabled:

```
[
    while (! zz_process_name) ; /*loop while false*/
    ...original process body...
]
```

If a process had once-only activation, a single bit channel was used. When a state was entered in which a process was activated once only, a message was sent on the channel. When a process started an execution, it would wait until it received a message on the channel (say zz_<process_name>) before continuing to execute the rest of the process body:

```
[
    boolean zz_idle;
    zz_idle = receive(zz_process_name); /* hang till message arrives*/
    ...original process body...
]
```

This representation for process activations proved unsatisfactory both for processes activated once and for processes activated continuously. In the case of processes with once-only activations, the process did not use the value of the variable zz_idle and the optimisations performed by the Olympus system would remove some of the handshaking signals required for correct operation. In the case of processes which had continuous activations, a potential race condition existed. For example, consider a process which reported an event to a controller which caused a state transition to a state in which the process was switched off. Because the controller takes finite time to execute it would be possible for the process to begin its next execution before the controller could clear the control line to switch the process off.

Because of these problems, an alternative representation of process activations was designed. In the new representation, two control lines were added between a controller and each process to be switched off. Each time a process started a new activation, it would set its *request* line true and wait for the *acknowledge* line from the controller to go true before continuing to execute the rest of the process body:

```
[
    write zz_process_name_rqt = 1;
    while (! zz_process_name_ack) ; /*loop while false*/
    write zz_process_name_rqt = 0;
    ...original process body...
]
```

In the HardwareC representation of the controller a static variable was added for each of the processes in the diagram which was deactivated in any state. When the controller started an execution, it would check each process's request line and static variable. If these were respectively true and set, the process would be activated by setting the acknowledge line true for one clock cycle. The static variable would be cleared for processes which were activated once only.

6.3 The PICSIL Synthesis Manager

The synthesis path discussed in the previous two sections involves several software tools which have to be invoked in the correct sequence. Each has its own set of commands, presenting a large learning curve when all the tools are combined. Within the PICSIL environment, the alternative synthesis options are well-defined; consequently PICSIL includes a Synthesis Manager (see Figure 6.27) which coordinates the operation of the lower-level software. Thus system designers do not have to learn about the peculiarities of controlling the tools employed in the synthesis process.

This Synthesis Manager is fully integrated into the PICSIL editor and allows the designer to define, initiate, monitor and terminate synthesis.

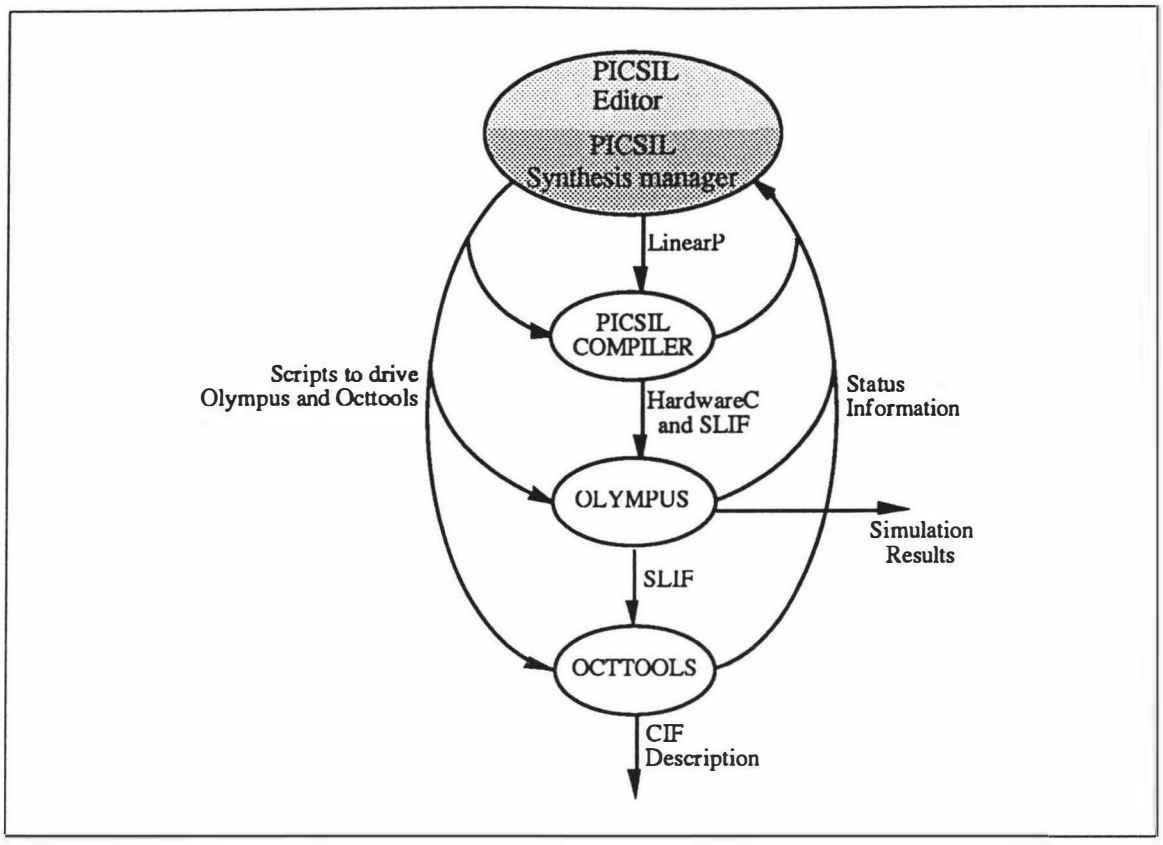


Figure 6.27 Components of a PICSIL synthesis

The Synthesis Manager is invoked from within the PICSIL editor by selecting the Synthesis Manager option in the *synthesis* pull down menu, causing the synthesis dialogue box (see Figure 6.28) to be displayed. Part of the dialogue box is made up of a control panel allowing the designer to select a limited number of options, used to guide synthesis. The options presented on the control panel have been kept as general as possible so that the designer need not know specific details about each of the tools in the synthesis path.

Once the designer has selected the desired options, the synthesis process is initiated by clicking on the start button. The Synthesis Manager using the selected options in the control panel then generates and issues the necessary commands to drive the synthesis process.

The Synthesis Manager also captures the output of the currently invoked tool which it monitors, so that progress reports on the status of synthesis can be displayed in the progress window. The designer can click on the STOP button at any time to terminate the synthesis process.

The first stage is the conversion of the PICSIL definition into a LinearP format suitable for translation into HardwareC and subsequent high level synthesis using Olympus. As the LinearP description is generated, each object is checked to ensure that it has a name and/or a description if required. If one of these is absent when required, the PICSIL editor opens a window for the diagram in which the object appears (if not already open), selects the object and displays an error message.

If the LinearP description is generated correctly, the Synthesis Manager invokes the PICSIL compiler with the LinearP description. The compiler sends messages back to the Synthesis Manager either when it starts processing a new object so the progress window can be updated, or when a syntax error is detected. The latter event causes a window to be opened in which the

line of the data dictionary entry in which the syntax error occurs is highlighted. A syntax error message is also displayed in a pop-up dialogue as shown in Figure 6.29. Once a syntax error is detected, the Synthesis Manager terminates the synthesis process.

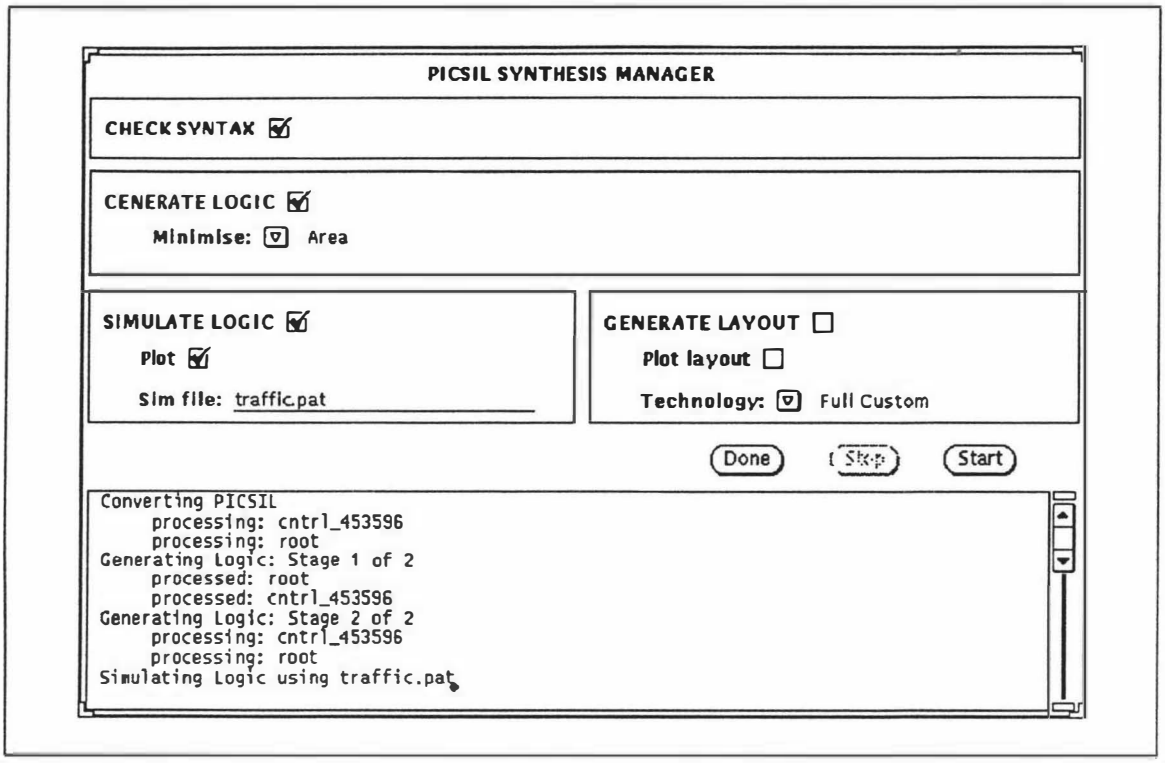


Figure 6.28 Synthesis Dialogue pop-up window

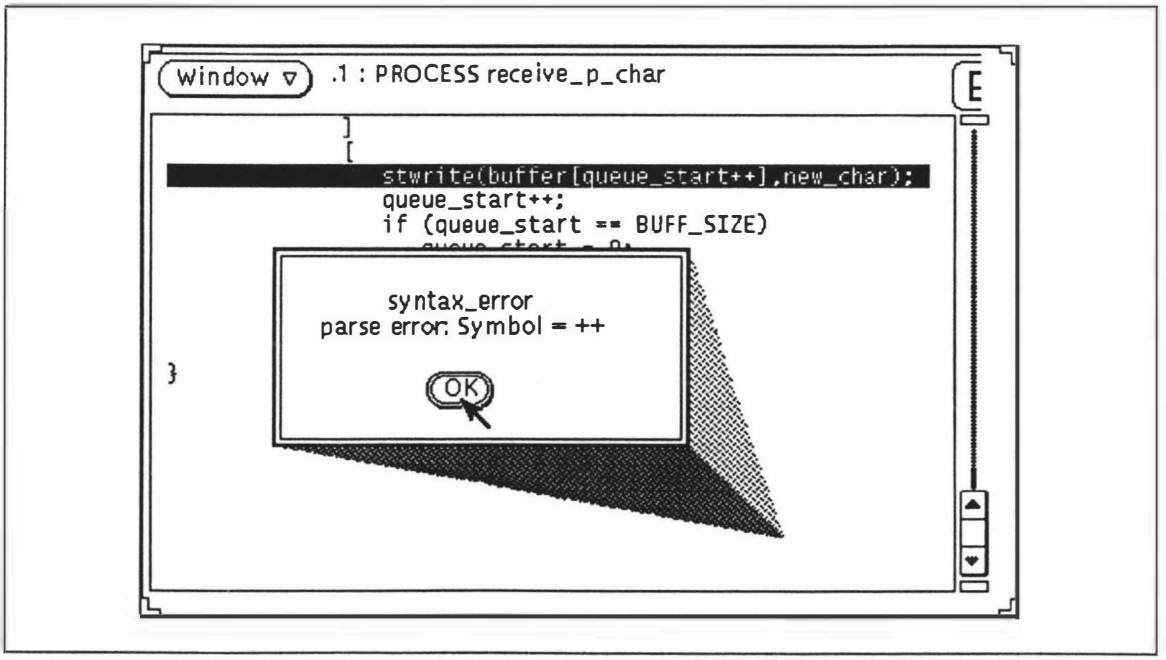


Figure 6.29 Reporting of syntax errors back to PICSIL editor

Once suitable input for Olympus has been generated, the Synthesis Manager produces scripts to make Olympus perform high level synthesis on the design. High level synthesis using Olympus is split into two parts: behavioural synthesis using Hercules and structural synthesis using Hebe. The script for behavioural synthesis initiates Hercules, supplies it with the

HardwareC description generated by the PICSIL compiler, and instructs it to perform an automated synthesis to produce a SIF description for each HardwareC model (process, procedure, function or block) which it produces. Routers and data stores are represented by dummy SIF descriptions. The Synthesis Manager also displays status messages which occur in the Hercules output.

The script which the Synthesis Manager produces to drive structural synthesis first generates dummy SLIF definitions for the routers and data stores and SLIF definitions for procedures and functions used in the design. Subsequent sections of the script traverse the tree of PICSIL input diagrams in bottom-up order, generating a SLIF definition of each, including the SLIF definitions of any child diagrams, functions, procedures, routers, or data stores referred to in the diagram. In order that the real SLIF definitions of routers and data stores are included, their dummy definitions are first replaced by the real definitions generated by the PICSIL compiler (see section 6.2).

Once high level synthesis has been completed, the SLIF description generated can be used to simulate the design and/or used as input to logic synthesis.

If the SIMULATE LOGIC LEVEL DESCRIPTION box is selected in the control panel, then the design will be simulated using Mercury within Olympus. In addition to the SLIF description of the design Mercury also requires a file containing an input test pattern to perform a simulation. This file is made up of two parts, a list of the inputs, and a set of input vectors. While the list of inputs can be generated automatically by PICSIL, the set of input vectors cannot, and these have to be supplied by the designer.

To prevent the designer having to leave the PICSIL environment to generate the file containing the test pattern, a text editing window (see Figure 6.30) can be opened by selecting the Define test pattern option of the synthesis menu to enter a test pattern. If the PICSIL design to be simulated has already been compiled using the PICSIL compiler, then the list of inputs can be automatically inserted into the text window by selecting the Insert input vector option of the File pull down menu on the window.

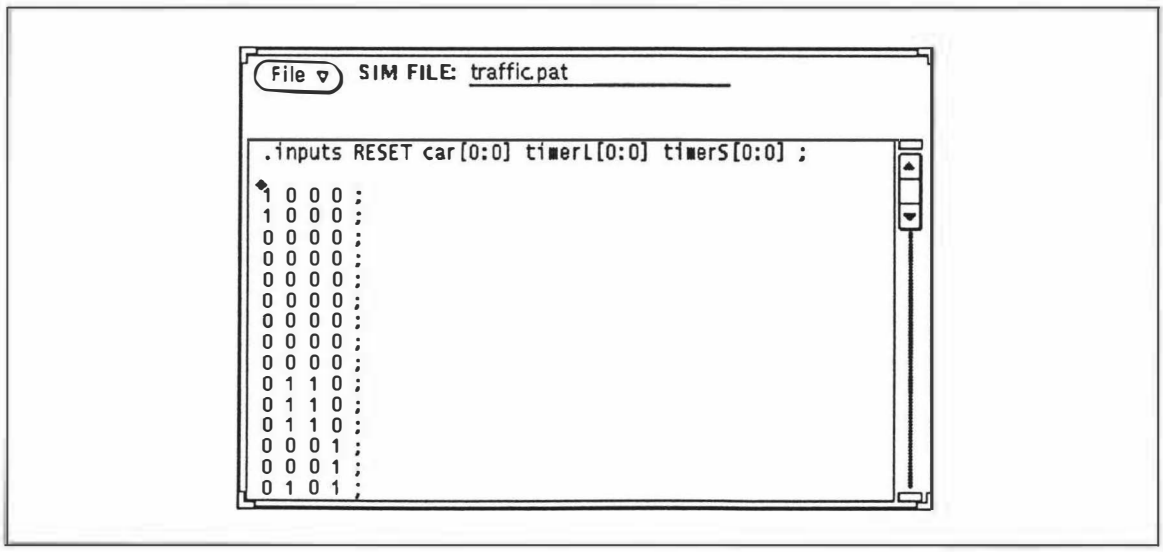


Figure 6.30 Window to allow test vectors to be defined for use in simulation

To perform simulation using Mercury, the design manager produces a script which initiates Mercury, supplies it with the SLIF description of the design and the name of the file containing the input test pattern provided on the control panel, and instructs it to perform the simulation. If the Plot box is selected, a command is also inserted in the script to instruct Mercury to

display a plot of the simulation results. Otherwise the simulation results are displayed in the progress window. An example simulation plot for a traffic light controller to be discussed in the next Chapter is shown in Figure 6.31.

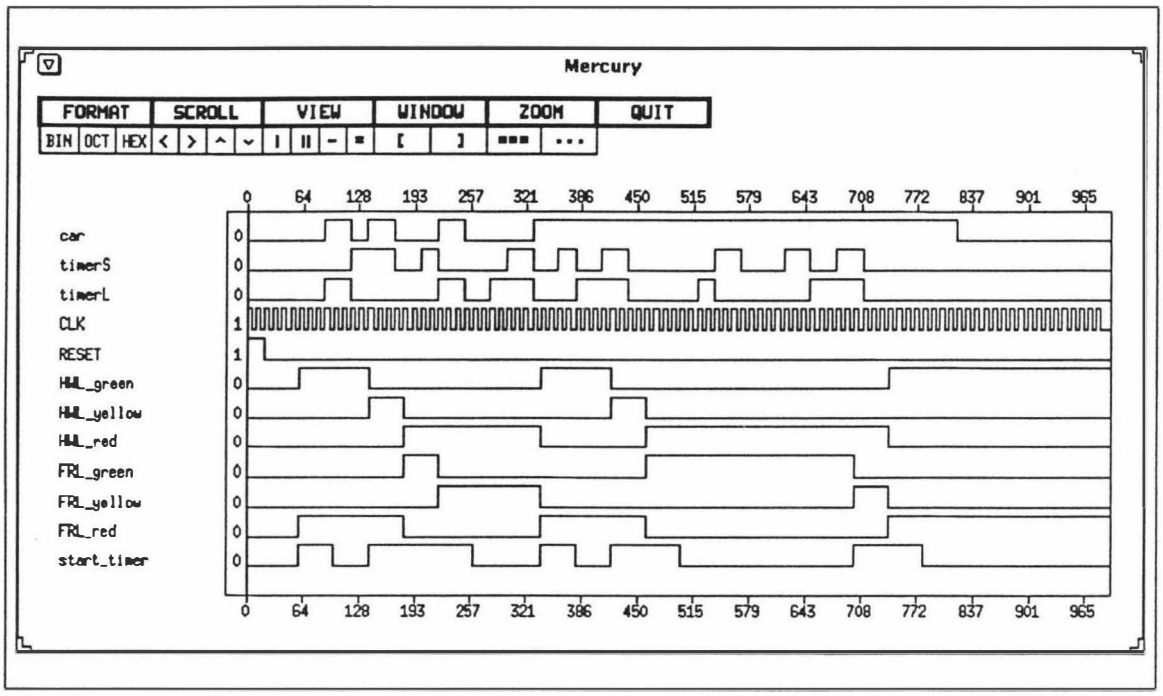


Figure 6.31 Results of a simulation shown as a plot

The last stage in the synthesis path is the generation of a chip layout from the SLIF description generated during the high level synthesis stage using Octtools. If the GENERATE LAYOUT box is selected in the control panel, the design manager produces a script which invokes each of the tools in the chain shown in Figure 6.5 (see page 103). The design manager monitors the output of each of the tools so that progress can be displayed in the progress window.

The generation of the input files root.iopos and padframe.bdnet shown in Figure 6.5, requires details about the desired connections between the design's I/Os and the pads in the padframe, which cannot be derived from a PICSIL design. Facilities have been included into the PICSIL environment to allow these details to be defined. By selecting the Define pads option from the synthesis menu, a pop-up window is displayed (see Figure 6.32). This window shows all the I/Os for a design (e.g char_in_data{0}) in the centre of the window, and the pads (e.g. 27) in the padframe displayed around the circumference of the window. Different padframes can be selected from a menu in the windows menu bar. By clicking on an I/O name then a pad (or vice versa) the designer can define a connection between the I/O and the pad represented by showing the pad number next to the I/O.

Once the designer has finished defining pad connectivity, the done option from the window menu can be selected and synthesis can proceed. The design manager then uses the defined connections and template files to generate the root.iopos and padframe.bdnet files.

The PICSIL synthesis system has been tested successfully on a number of designs and the results are presented in the next chapter.

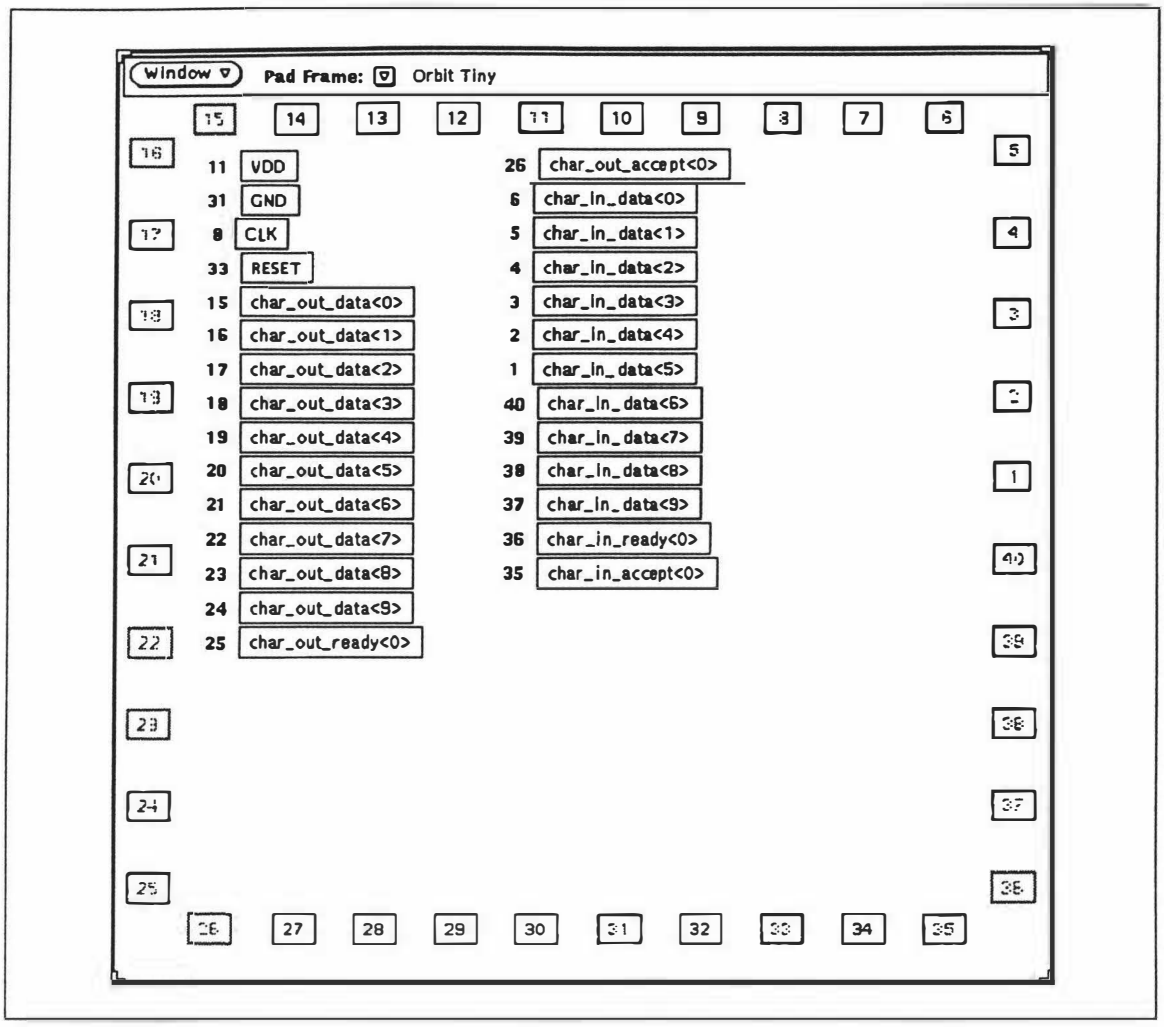


Figure 6.32 Window to allow designer to define which pads each of the I/Os are attached to

Chapter 7

Testing the PICSIL Synthesis System

To test the PICSIL synthesis system, three designs have been synthesised. They were selected to test the algorithms used to map PICSIL components to hardware, and the synthesis manager's ability to fully automate the synthesis process. The exercises were not designed to test the Olympus and Octtools synthesis algorithms.

The devices used to validate the synthesis software were:

- a *parallel buffer*, containing a four-word buffer which receives and transmits words using a ready/accept handshaking protocol;
- a *simplified packet switch*, which allows packets of data to be switched from either of its two inputs to either of its two outputs depending on an address contained in the first bit of the packet;
- a *traffic light controller* used to control the lights at an intersection between a main highway and a quiet side road.

It should be noted that the PICSIL descriptions for each of the examples may seem somewhat artificial, as the representations have been selected to test as many features as possible while keeping the eventual layout size as small as possible. In the current implementation of the PICSIL system, the padframe that can produced automatically is the MOSIS TINY padframe[1], which allows a logic core of approximately 2 mm x 2 mm to be enclosed. Because of this restriction, the discussion of the first two examples is limited to the earlier stages of the synthesis path as their eventual layouts are too large to fit within the MOSIS TINY pad frame. The last example is smaller and has been included to illustrate the later stages of the synthesis path.

---

[1]MOSIS (MOS Implementation System) was setup into 1980 to provide fabrication services to U.S. goverment contractors, agencies and university classes. The MOSIS tiny padframe is made up of 40 pads (10 along each side) which are at fixed locations inside a 2.2 mm by 2.4 mm box. Orbit Semiconductor also accept designs for fabrication which use the MOSIS tiny padframe (Orbit, 1991).

7.1 Parallel Buffer

The parallel buffer provides buffering for up to four words which are received and transmitted using ready/accept handshaking protocols. Figure 7.1 shows the top level PICSIL diagram for the design.

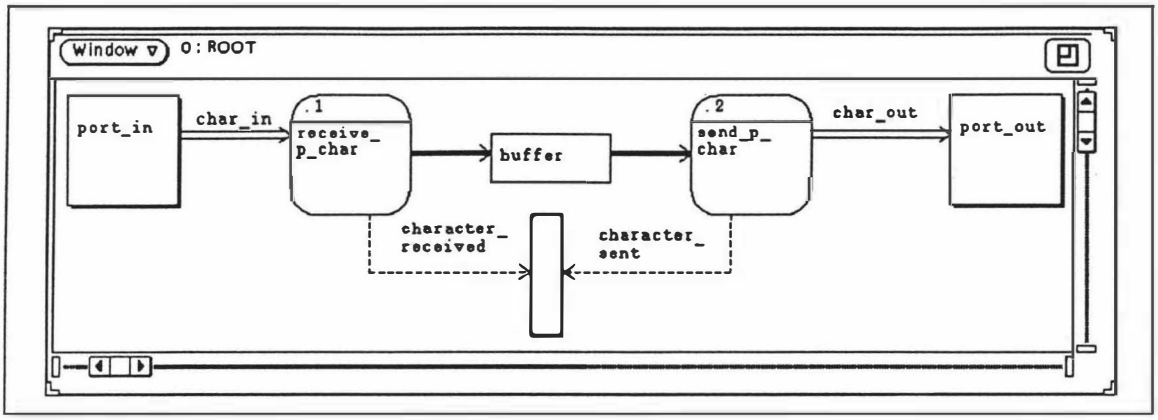


Figure 7.1 PICSIL diagram for Parallel Buffer Example

Figure 7.2(e) shows the definition of the process receive_p_char which reads in and synchronises the input of new values from the group flow char_in (defined in Figure 7.2(d)), and stores them in the data store buffer (defined in Figure 7.2(c)). Subsequently the queue_start variable is incremented and the addition of the character is reported to the controller.

```
#define UPPER_WORD 3
#define BUFF_SIZE 4
STORE boolean[0:UPPER_WORD][BUFF_SIZE] buffer;
```
c

```
FLOW Continuous boolean[0:UPPER_WORD]
char_out\data;

FLOW Continuous boolean char_out\ready;

FLOW Continuous boolean char_out\accept;

FLOW Group (send_p_char:port_out) {
   ->char_out\data;
   ->char_out\ready;
   <-char_out\accept;
} char_out;
```
a

```
FLOW Continuous boolean[0:UPPER_WORD]
char_in\data;

FLOW Continuous boolean char_in\ready;

FLOW Continuous boolean char_in\accept;

FLOW Group (port_in:receive_p_char) {
   ->char_in\data;
   ->char_in\ready;
   <-char_in\accept;
} char_in;
```
d

```
PROCESS .2 send_p_char
begin
  static[0:1] queue_end = 0;
  boolean[0:UPPER_WORD] out_char;
  write(char_out\ready,0);
  seqbegin
   stread(buffer[queue_end],out_char);
   parbegin
     load queue_end = queue_end + 1;
     seqbegin
       parbegin
         write(char_out\ready,1);
         write(char_out\data,out_char);
       parend
       while (!char_out\accept);
       write(char_out\ready,0);
     seqend
   parend
   report(character_sent);
  seqend
end
```
b

```
PROCESS .1 receive_p_char
begin
   static[0:1] queue_start = 0;
   boolean[0:UPPER_WORD] new_char;
   write(char_in\accept,0);
   seqbegin
    while(!char_in\ready);
    read(char_in\data,new_char);
    parbegin
      seqbegin
        write(char_in\accept,1);
        write(char_in\accept,0);
      seqend
      seqbegin
        stwrite(buffer[queue_start],new_char);
        queue_start++;
      seqend
    parend
    report(character_received);
   seqend
end
```
e

Figure 7.2 Data dictionary entries for parallel buffer exercises

The send_p_char process defined in Figure 7.2(b) reads values out of the data store buffer, and synchronises their output on the group flow char_out. Queue_end is then incremented and the controller is notified that a value has been removed from the buffer.

The controller is used to turn the receive_p_char and send_p_char processes on and off according to how full the data store is, preventing values being overwritten before transmission, or being transmitted more than once. This information is derived from events reported on the character_received and character_sent event flows. Figure 7.3 shows the state transition diagram for the controller in the DFD shown in Figure 7.1. Each state represents the number of items in the buffer with the empty state indicating zero items and the full state indicating four items. A character_received event causes a state transition towards the full state, in which the receive_p_char process is deactivated. Conversely a character_sent event causes a state transition back towards the empty state in which the send_p_char process is deactivated. In the other three states, both processes are activated.

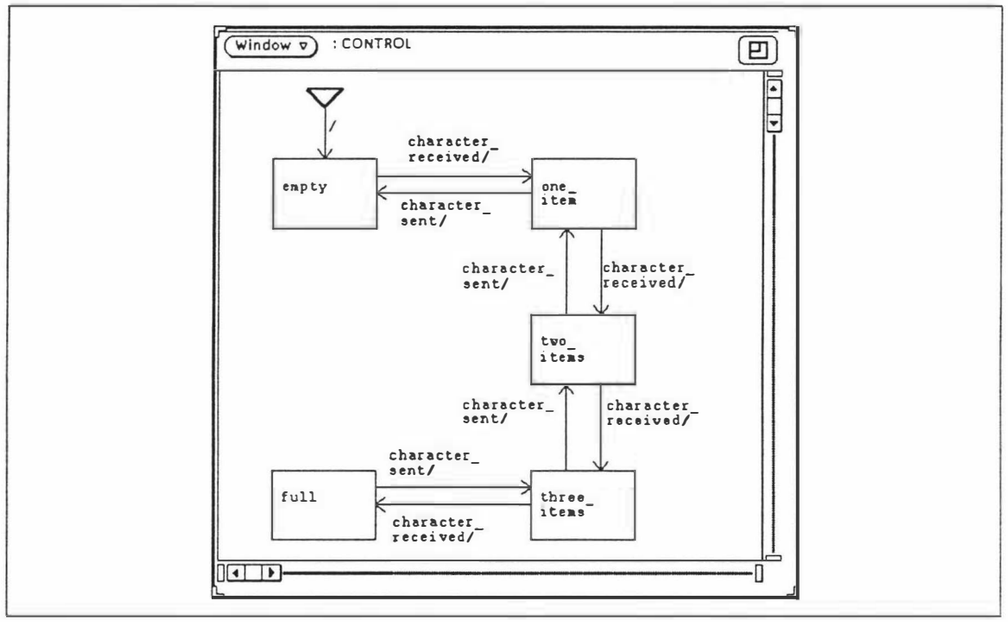


Figure 7.3 State transition diagram for controller in Figure 7.1

This design has been correctly synthesised, thus providing partial verification of the PICSIL synthesis system. The design allowed the following features of the PICSIL synthesis system to be tested:

- the implementation of PICSIL diagrams as HardwareC blocks;
- the implementation of PICSIL processes as HardwareC processes;
- the implementation and integration of data stores and store flows in SLIF and HardwareC;
- the implementation of PICSIL store I/O statements in HardwareC;
- the implementation of PICSIL controllers as HardwareC processes;
- the implementation of process activation and deactivation;
- the correctness of Synthesis Manager's algorithms used to automate synthesis.

The correctness of the design has been checked at various stages by direct inspection, by the syntax-checking routines included in all the intermediate-level tools, and by inspection of simulations at three progressively more concrete levels of implementation.

Figure 7.5 shows the intermediate representations that were generated during the synthesis of the parallel buffer design. The numbered eyes represent test points at which the inspections referred to above occurred. Direct inspections of a representation are indicated by an eye adjacent to the representation's filename; inspections of simulation output are indicated by an eye adjacent to a waveform representing that output. Syntax checks occurred at every phase.

The development of the PICSIL editor and PICSIL compiler have involved the same sorts of visual inspection as those at test points 1, 2, and 3. Thus by the time this test of the complete path was performed, the earlier phases of the process were fairly reliable, and only the most trivial errors were detected at these test points. However, at this stage, it became apparent that some syntax errors were not being detected by the PICSIL software. Although they were subsequently detected by Hercules, the HardwareC compiler, information about their position in the original PICSIL design was no longer available at that phase in the synthesis, as they had been omitted from the PICSIL data structure. Thus locating them was not simple.

It was decided to deal with this problem during the rewrite of the synthesis path which will be suggested in Chapter 8, in view of its comparatively minor relevance to establishing the validity of the current approach, and the disproportionate amount of effort which would be involved in modifying the current PICSIL compiler to detect the errors.

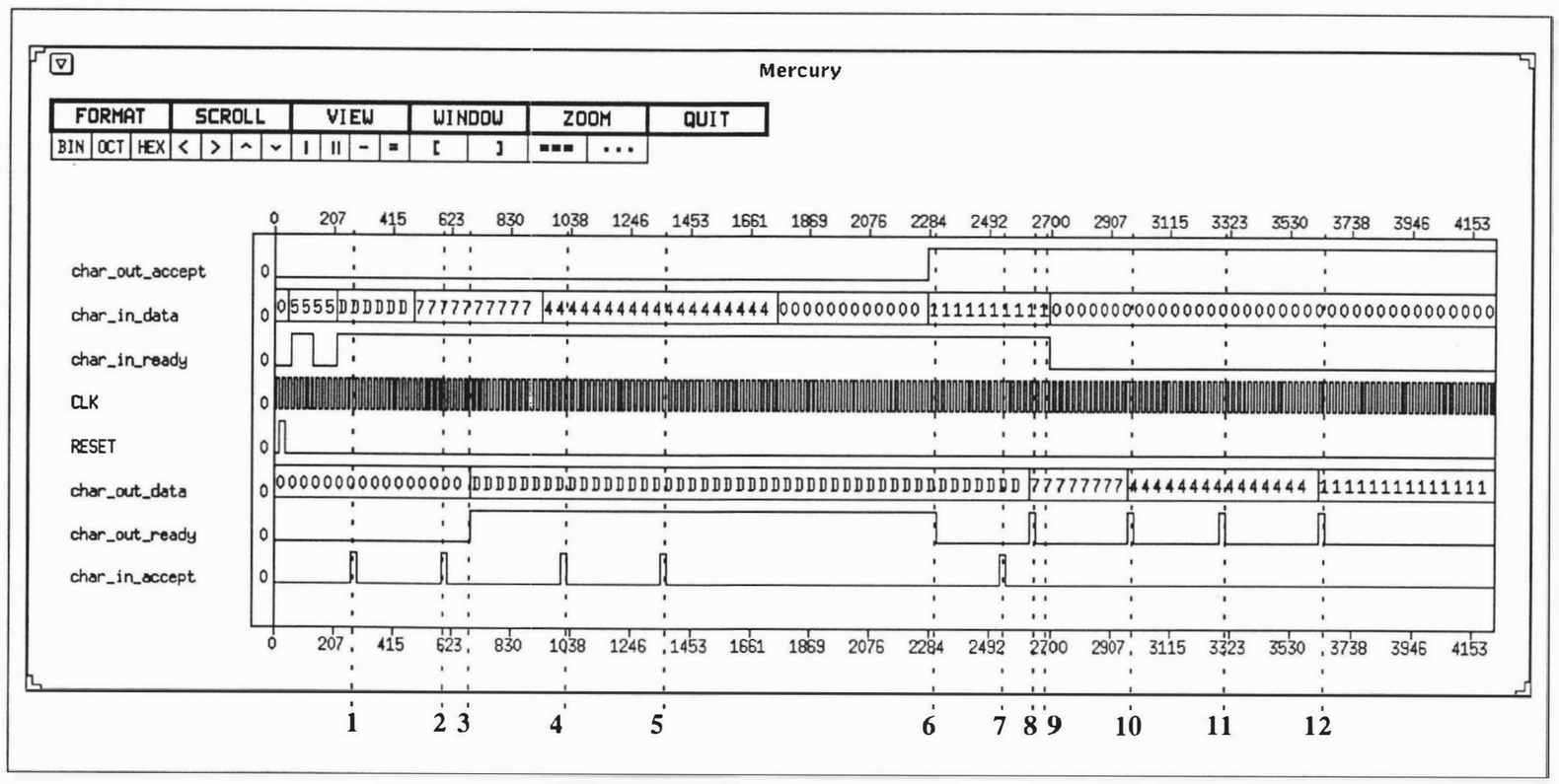


Figure 7.4 Simulation results produced at test point 4 for Parallel Buffer

Test point 4 has been used to verify the behaviour of the complete design at the logic level. Figure 7.4 shows a plot of one of the simulations performed at this level using Mercury within

the Olympus system, and shows that the resulting circuit exhibits the desired behaviour. The waveforms shown for char_out_accept, char_in_data, char_in_ready, CLK and RESET, represent inputs whose values were derived from a set of input vectors. The char_out, char_out_ready and char_in_accept waveforms represent the outputs of the system. The numbers at the bottom of the plot indicate important instants in the simulation.

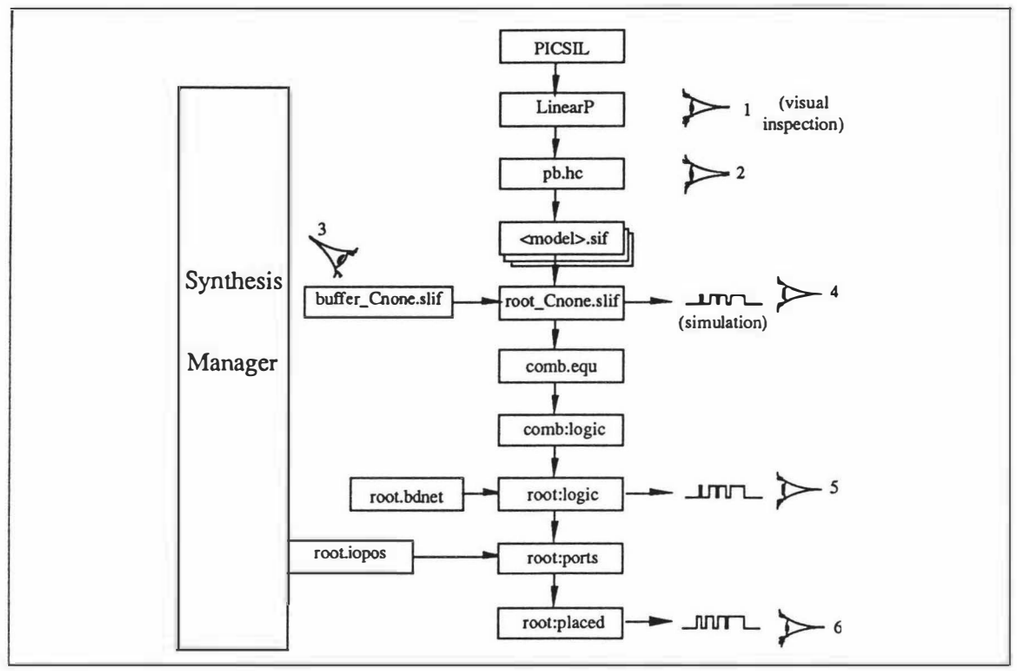


Figure 7.5 Parallel Buffer : Test Points

A pulse on char_in_accept (i.e. instants 1, 2, 4, 5 and 7) indicates the instants at which the process receive_p_char reads in new values. Likewise, a positive pulse on char_out_ready (i.e. instants 3, 8, 10, 11 and 12) indicates that a new value is ready to be output on char_out_data, by process send_p_char. A value is not considered to be output until a positive value is detected on the external input char_out_accept. The data values output on char_out_data at instants 3, 8, 10, 11 and 12 are the same as the data accepted on the input char_in_data at instants 1, 2, 4, 5 and 8, indicating that data is being correctly stored into and retrieved from the data store.

The char_out_accept line has been held low (under external control) until instant 6, to prevent any characters from being output until then. Four characters are however read in before this time, (at instants 1, 2, 4 and 5) so that the buffer becomes full, and the process receive_p_char should be deactivated. It can be seen that this has occurred, as no characters are accepted from instant 5 until instant 7.

At instant 9 the input char_in_ready is set low, meaning no new characters can be input. The buffer becomes empty at instant 12 when the last character is output and the send_p_char process should then become deactivated. Again this can be verified, as the output char_out_ready line does not go high again.

The PICSIL environment was used to enter the input test vectors and initiate the simulation discussed above. This simulation made verification of the design difficult, as the designer could not interactively alter input values in response to particular output values. For example char_out should only be true when char_out_ready is true. However when the designer is creating the input test vectors, it is not possible to determine the instants when char_out_ready will be

true, and so they must be predicted. As testing has been given a low priority in this research, no attempt to provide a better interface for simulation was made, and is discussed as future work in Chapter 8.

The PICSIL synthesis system has been used to take the parallel buffer design through to a chip layout (i.e. test point 6 in Figure 7.5). The size of the resulting layout was 2.025 mm x 3.159 mm, which is too large to fit inside the MOSIS TINY frame, so synthesis could not proceed past this point.

Simulation has also been performed using Musa (Octtools, 1991) at test points 5 and 6 to ensure that software tools in the synthesis path had not changed the behaviour of the circuit. The set of input vectors used at test point 4 were used again at points 5 and 6, allowing the three set of results to be compared. Figure 7.6 shows a plot of the results produced for test point 6. To provide some assistance in reading the plot in Figure 7.6, the hexadecimal values of the four input char_in_data lines, at the time each new value is read, have been included above the plot. Likewise, the hexadecimal values of the four char_out_data outputs at the times when values are output have been shown below the plot. The simulation results in Figure 7.6 can be directly related to the simulation results in Figure 7.4, meaning that the behaviour of the design has been preserved during synthesis between test points 4 and 6.

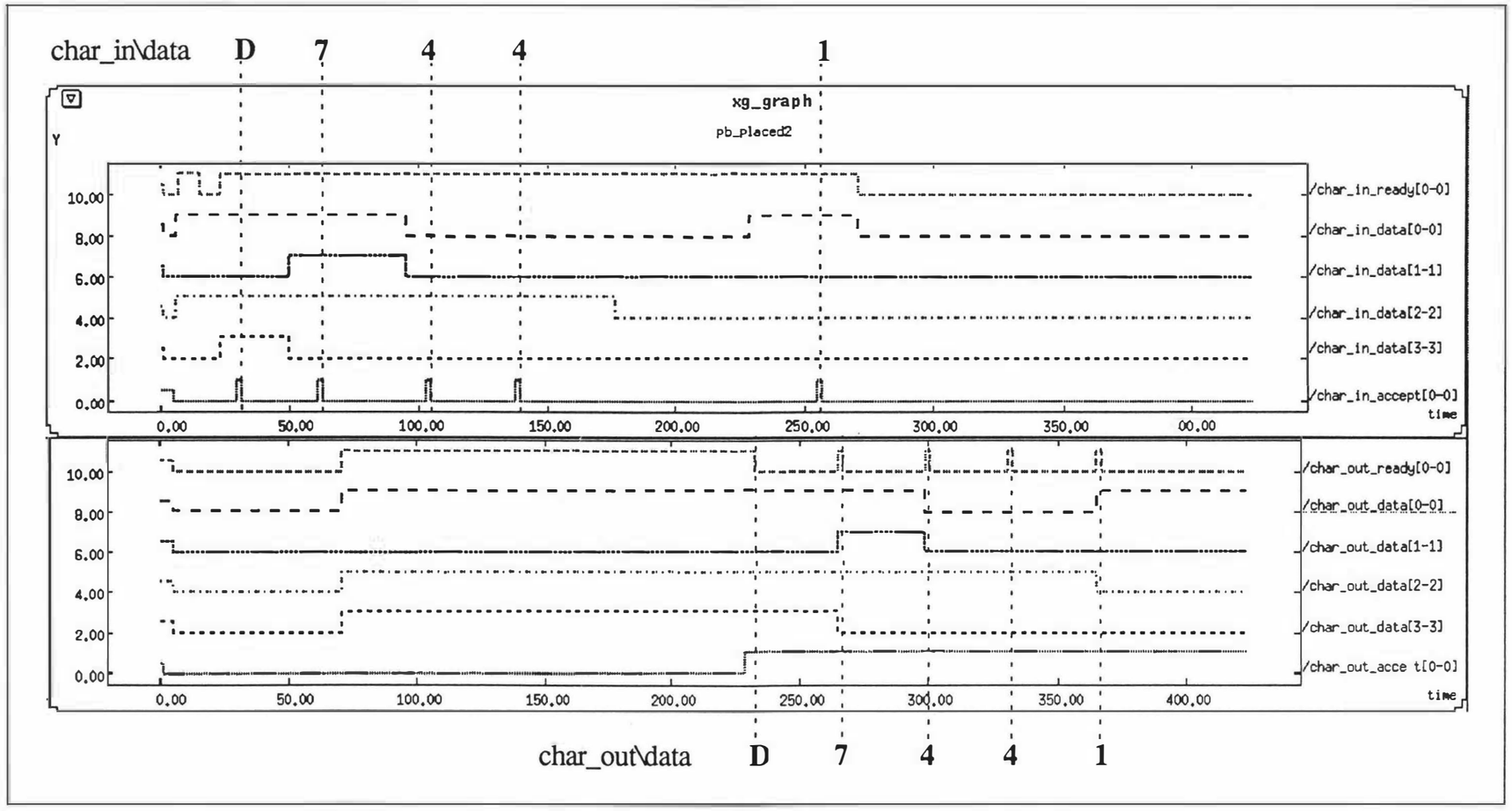


Figure 7.6 Simulation results produced at test point 6 for Parallel Buffer

From the results of these simulations and visual checks of the intermediate representations, it can be seen that all tools required to synthesise a logic level description of the parallel buffer have functioned correctly. Furthermore, traversal of the entire synthesis path was directed automatically from within the PICSIL environment. (The simulations at test points 5 and 6 are not part of the standard synthesis path).

7.2 Packet Switch

The packet switch reads packets of data from either of its two inputs and switches them to either of its outputs depending on an address in the first bit of the packet.

Figure 7.7 shows the PICSIL diagram used to describe the packet switch device. The process receive_1 (see definition in Figure 7.9(e)) reads in values from the group flow in_1 using the same protocol as the one used by receive_p_char in the parallel buffer. An extra flow in_1\locked has however been included in the in_1 group flow to help verify the design during simulation. Each value read in represents a whole packet, comprising a one-bit address and two three-bit data words (defined by the typedef in Figure 7.9(a)). To determine which port a packet should be switched to, a switch statement uses new_char.address as the selector. If the address bit is 0, then the statement lock(from_1,to_1); locks the path to the process send_1, otherwise the path to the process send_2 is locked. Once a path through the router has been locked, the output in_1\locked is set to high and the two data words in the packet are sent to the router. After the two words have been sent, in_1\locked is cleared, and the statement unlock(from_1); frees the path through the router. The group flow in_2 and the process receive_2 have the same structure and behaviour respectively as the group flow in_1 and the receive_1, so their definitions have not been included in Figure 7.9. The processes send_1 (see definition in Figure 7.9(d)) and send_2 both transfer the values received from their to_ discrete flows to their out_ continuous flows.

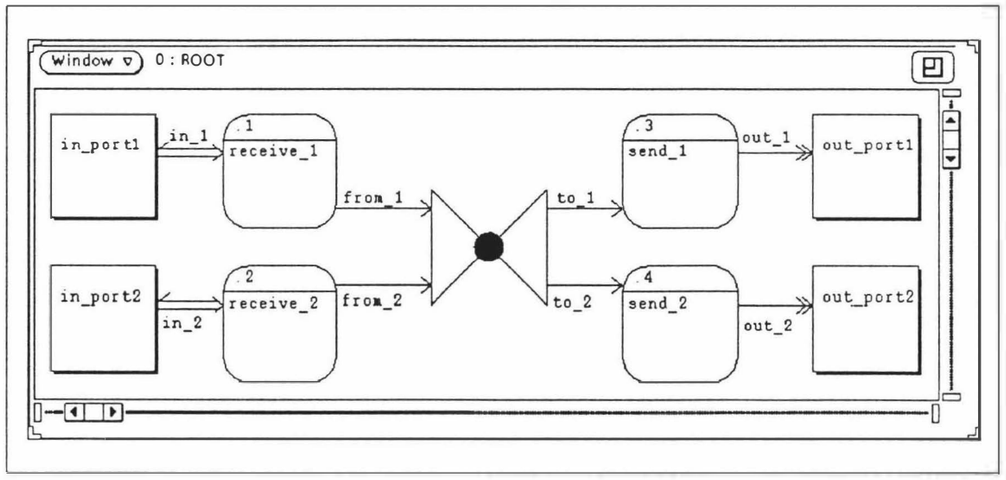


Figure 7.7 PICSIL diagram for Packet Switch example

This packet switch has been synthesised, thus providing partial verification of the PICSIL synthesis system. The design allowed the following features of the PICSIL system to be tested:

- the implementation of *struct* types in HardwareC;
- the implementation of routers in HardwareC and SLIF;
- the correctness of the algorithms used by the Synthesis Manager to automate the synthesis of designs containing routers.

The same three techniques for verifying the parallel buffer design have been used to verify the correctness of the synthesis system for this design. Figure 7.10 shows the intermediate representations generated during the synthesis of the packet switch design, and the test points used to verify the design.

The design has been visually inspected at test points 1, 2 and 3 to verify the correctness of the intermediate descriptions. Each of these descriptions is correctly read by the synthesis tools that follow them in the synthesis path, confirming that their syntax is correct.

Test point 4 has been used to verify the behaviour of the complete Packet Switch design at the logic level. Figure 7.8 shows a plot of one of the simulations performed at the logic level using Mercury to verify the behaviour of the design. On this plot, the waveforms shown for in_1_data, in_1_ready, in_2_data, in_2_ready, CLK and RESET represent inputs whose values were derived from a set of input vectors. The waveforms shown for in_1_data and in_2_data do not clearly show which bits map onto which components of the struct type for which the flows have been declared in PICSIL. To aid in the readability of the diagrams, the values of each component in both the in_1_data and in_2_data flows, when read by their importing processes, have been shown above the plot.

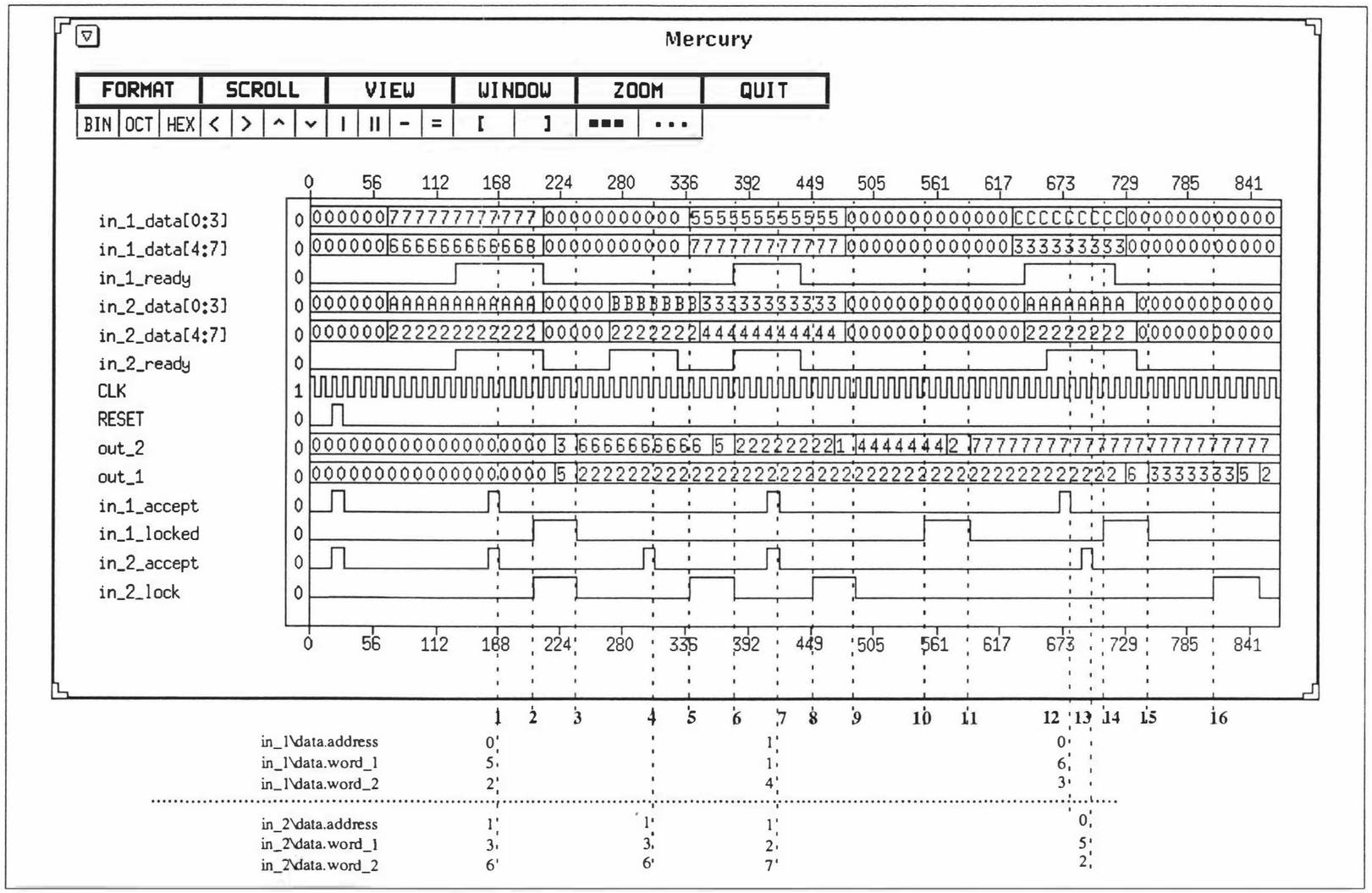


Figure 7.8 Simulation results produced at test point 4, for Packet Switch

```
#define WORD_SIZE 2

typedef struct{
  boolean address;
  boolean[0:WORD_SIZE] word_1;
  boolean[0:WORD_SIZE] word_2;
} WORD_IN;
```
a

```
FLOW Discrete boolean[0:WORD_SIZE] from_1;

FLOW Continuous boolean[0:WORD_SIZE] out_1;

FLOW Discrete boolean[0:WORD_SIZE] to_1;
```
b

```
FLOW Continuous WORD_IN in_1\data;

FLOW Continuous boolean in_1\ready;

FLOW Continuous boolean in_1\accept;

FLOW Continuous boolean in_1\locked;

FLOW Bidirectional Group (in_port1 : receive_1) {
    -> in_1\data;
    -> in_1\ready;
    <- in_1\accept;
    <- in_1\locked;
} in_1;
```
c

```
PROCESS .3 send_1
seqbegin
  boolean[0:WORD_SIZE] new_char;

  receive(to_1, new_char);
  write(out_1,new_char);
seqend
```
d

```
PROCESS .1 receive_1
seqbegin
  WORD_IN new_char;
  while(in_1\ready);
  read(in_1\data,new_char);
  parbegin
   seqbegin
     write(in_1\accept,1);
     write(in_1\accept,0);
   seqend
   seqbegin
     switch(new_char.address){
     case 0b0:
       lock(from_1,to_1);
       break;
     case 0b1:
       lock(from_1,to_2);
       break;
     }
     write(in_1\locked,1);
         send(from_1,new_char.word_1);
         send(from_1,new_char.word_2);
     write(in_1\locked,0);
     unlock(from_1);
   seqend
  parend
seqend
```
e

Figure 7.9 Data dictionary entries for packet switch

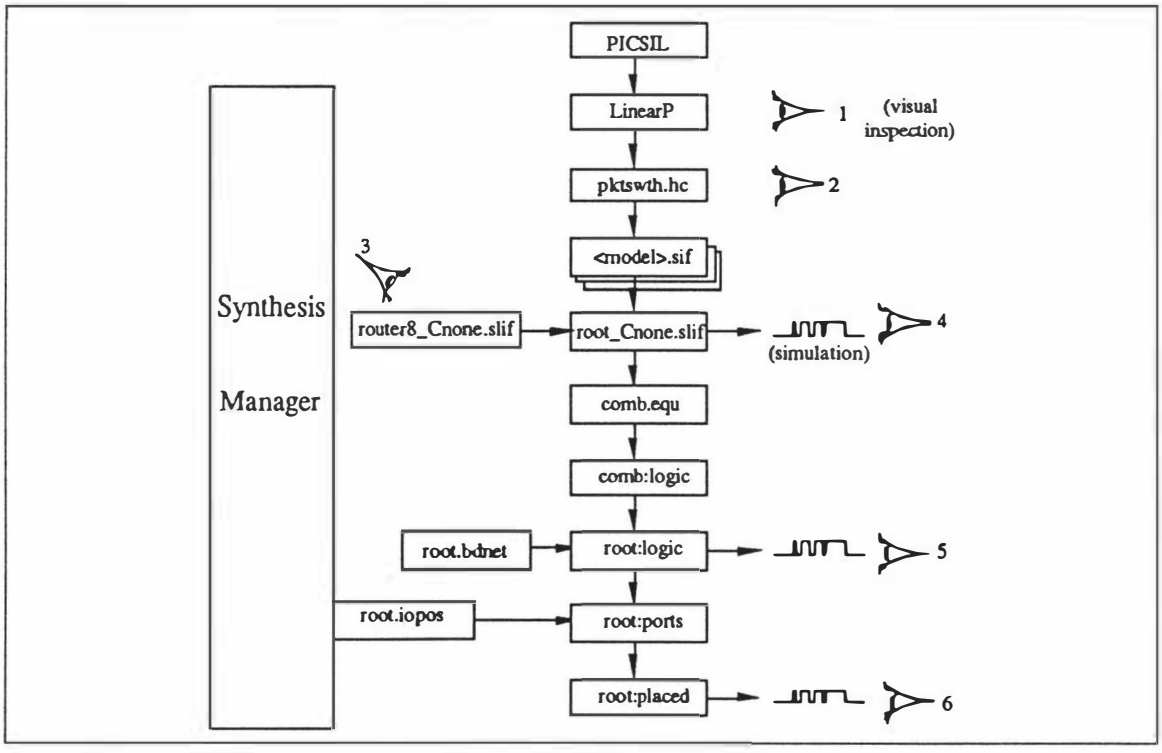


Figure 7.10 Packet Switch : Test Points

The waveforms out_2, out_1, in_1_accept, in_1_locked, in_2_accept and in_2_lock represent the outputs of the system. To clarify the discussion of the plot, the numbers below the plot indicate instants that are of interest.

Pulses on in_1_accept (at instants 1, 7 and 11) and in_2_accept (at instants 1, 4, 7 and 12) indicate the instants when the processes receive_1 and receive_2 respectively have read in new packets. The pulses on in_1_locked and in_2_lock indicate the instants at which the processes receive_1 and receive_2 have locked a path on their from_ export flows through the router to a to_ flow.

At instant 1, both receive_ processes read in new values. In this first case the address fields received by the two processes are different (i.e. receive_1 is sending its packet to send_1, while receive_2 is sending its packet to send_2) meaning that both processes can make a connection through the router at the same time. This is the case, as both the in_1_locked and in_2_lock lines are high between instants 2 and 3 indicating that both processes have a connection through the router at the same time. A short time later the two data words of each packet are output one after the other, on the correct out_ lines, indicating the router has performed the correct switching and allowed data to be sent through the router in parallel.

At instant 4 the process receive_2 receives a second packet, which is sent through the router to the process send_1 between instants 4 and 5. This case has been included to insure that the process receive_1 correctly releases the path to the process send_1 after it sends its first packet through the router. As the pulse in_2_lock appears soon after the process receives the packet, and the two words appear on the out_1 lines a short time later, it shows that the connection through the router has been released properly by the receive_1 process.

The third case tested, starting at instant 7, involves both processes receiving packets with the same destination address at the same time. This means one process has to wait while the other process makes a connection through the router to send its two words. From the simulation plot it can be seen that the process receive_2 locks a path through the router first (between instants 8 and 9) and the two data words appear on the out_ lines. At instant 11, after the process receive_2 has released its connection the process receive_1 locks a connection through the router to send_2 and sends it two words which are output on the out_2 lines.

In the previous case when both processes tried to make a connection through the router at the same time, the process receive_2 was given priority and allocated the channel first. The last case tested, starting at instant 12, ensures that if the process receive_1 has a connection through the router, and receive_2 requests a connection to the same destination, it does not deprive receive_1 of the connection. At instant 12, the process receive_1 receives a packet and makes a connection through the router between instants 14 and 15. Receive_2 receives a new packet at instant 13, but does not get the connection through the router until instant 16, after which receive_1 has finished with it, and indicates that the locking mechanism in the router is functioning correctly.

Synthesis to a chip layout has been performed from the PICSIL environment (i.e. test point 5). The resulting layout was 1.665 mm x 2.683 mm, which is still too large in one direction to fit inside the MOSIS TINY frame. Simulations have also been perform at test points 5 and 6, with the results agreeing with those performed at the logic level (test point 4).

From the results of this simulation and the visual checks made on the intermediate representations, it has been established that *struct* definitions and routers are correctly implemented by the tools that make up the PICSIL synthesis system. As with the synthesis of the parallel buffer, it was not necessary for the designer to operate outside the PICSIL environment at any stage during the synthesis process.

7.3 Traffic Light Controller

The traffic light controller discussed in this section is a modified version of the one discussed in chapter 4 (page 60). Instead of using two continuous event flows to output an encoded version

of the light switched on for each set of lights, a continuous event flow exists for each traffic light. Figure 7.11 shows the top level PICSIL diagram for the modified traffic light controller in which a group flow is used to represent the three continuous event flows for each set of lights. The controller definition for the modified traffic light controller is shown in Figure 7.12. The state transition diagram is the same as that used Chapter 4, while the event/action equations have been modified to control the three continuous event flows for each set of lights.

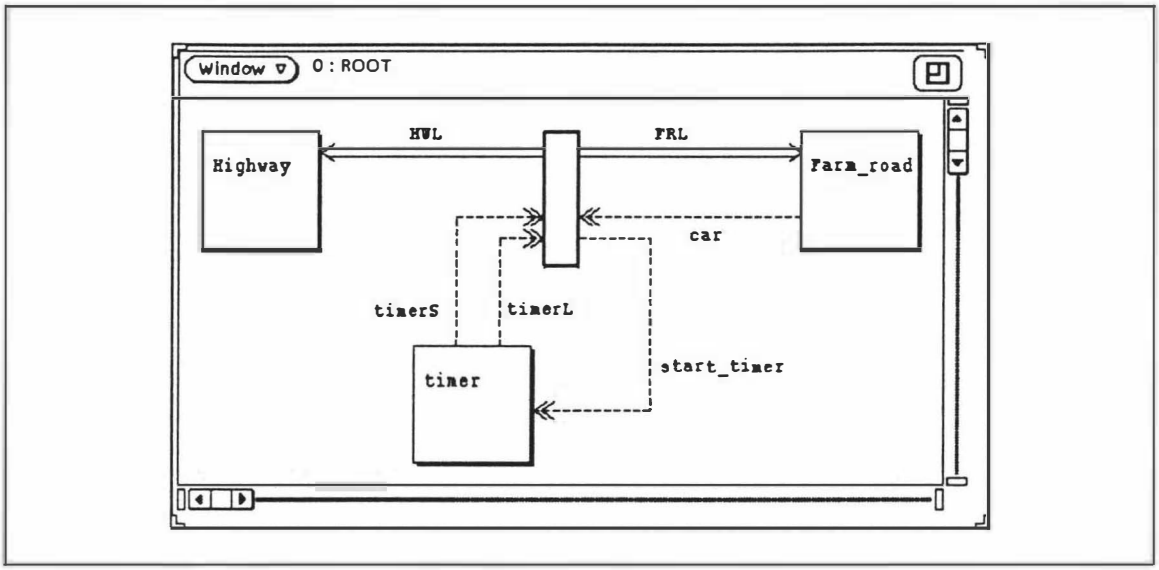


Figure 7.11 PICSIL diagram for traffic light controller

This design is significantly simpler that the parallel buffer and packet switch so that the resulting layout will fit within the area available within the MOSIS TINY frame, allowing the last stages of the synthesis process to be verified. The only construct not tested in the previous two designs is the continuous event flow. Figure 7.13 shows the intermediate descriptions generated in the synthesis of the traffic light controller and the test points used to verify the synthesis process. Stick representations are used to indicate where layouts were generated.

The traffic light controller was the first design used to test the later stages of the synthesis path. Because of this, extra visual checks have been included to verify the structure of the intermediate representations (see test points 3, 6 and 8) generated by the PICSIL software. In each case, the representation has the correct format and is read correctly by the tool that follows it in the synthesis path.

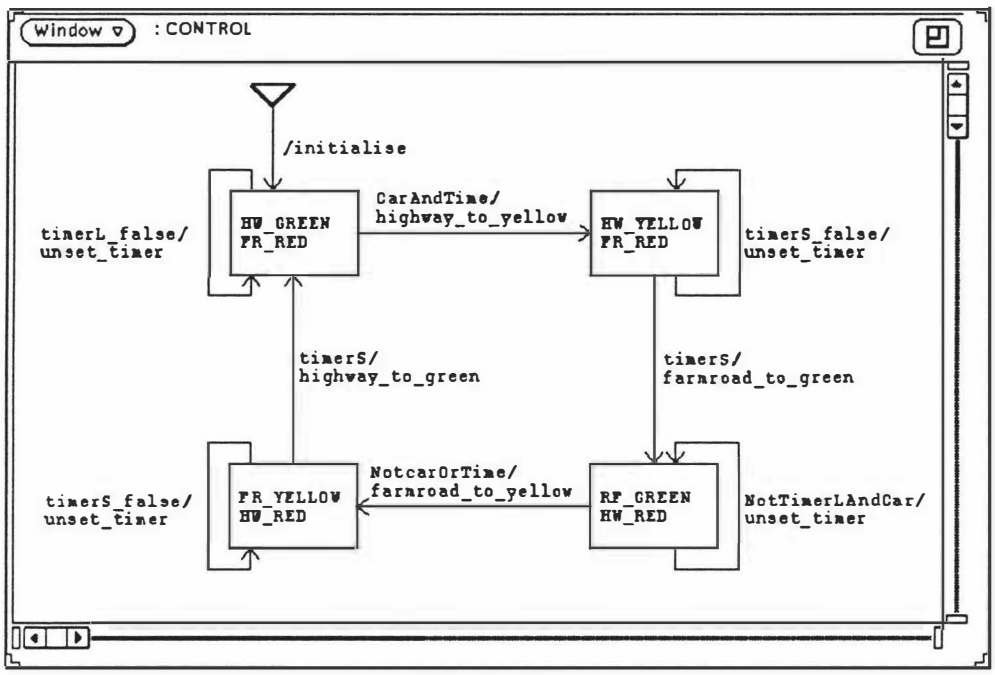
The behaviour of the traffic light controller has been verified by performing simulations at test points 1, 4 and 5.

The last stages (test points 7, 8 and 10) of the testing for this device have been visual checks on the layout generated to verify that the padframe is correct and that the logic has been placed correctly within it. Figure 7.14 shows a plot of the layout generated at test point 10. This plot shows that the logic block (layout in the centre), which contains 2550 transistors, has been positioned correctly at the centre of the pad frame.

A close inspection of this plot has shown that each of the pads in the padframe is at the correct location and is of the correct type (i.e. input, output, Vdd, GND or unused), and further, that the correct connections have been produced between the terminals on the logic block and the terminals of the pads in the pad frame.

The final stage in the testing has been to submit the final description of the design (i.e. root.cif) for fabrication using Orbit Semiconductors Foresight program (Orbit, 1991). At the time of submission of this thesis, the fabricated chips had not been received back from Orbit Semiconductor to allow the final testing to be performed.

The three designs discussed in this chapter have been successfully synthesised using the PICSIL synthesis system. While the layouts produced are almost certainly much larger than those generated by hand, it has been shown that it is possible to synthesise designs produced using the PICSIL editor into a hardware representation. Furthermore, the synthesis of these designs using the PICSIL system has shown that the possibility of fully automating the synthesis process, eliminating the creative distraction of manual synthesis.



(a)

```
#define HW_green HWL\green & !HWL\yellow & !HWL\red
#define HW_yellow !HWL\green & HWL\yellow & !HWL\red
#define HW_red !HWL\green & !HWL\yellow & HWL\red

#define FR_green FRL\green & !FRL\yellow & !FRL\red
#define FR_yellow !FRL\green & FRL\yellow & !FRL\red
#define FR_red !FRL\green & !FRL\yellow & FRL\red
```

```
event CarAndTime = car & timerL;
event NotcarOrTime = !car | timerL;
event timerS_false = !timerS;
event timerL_false = !timerL;
event NotTimerLAndCar = !timerL & car;
action initialise = HW_green & FR_red & start_timer;
action highway_to_yellow = HW_yellow & start_timer;
action farmroad_to_green = HW_red & FR_green & start_timer;
action farmroad_to_yellow = FR_yellow & start_timer;
action highway_to_green = HW_green & FR_red & start_timer;
action unset timer = !start timer;
```

(b)

Figure 7.12 Controller for the traffic light controller

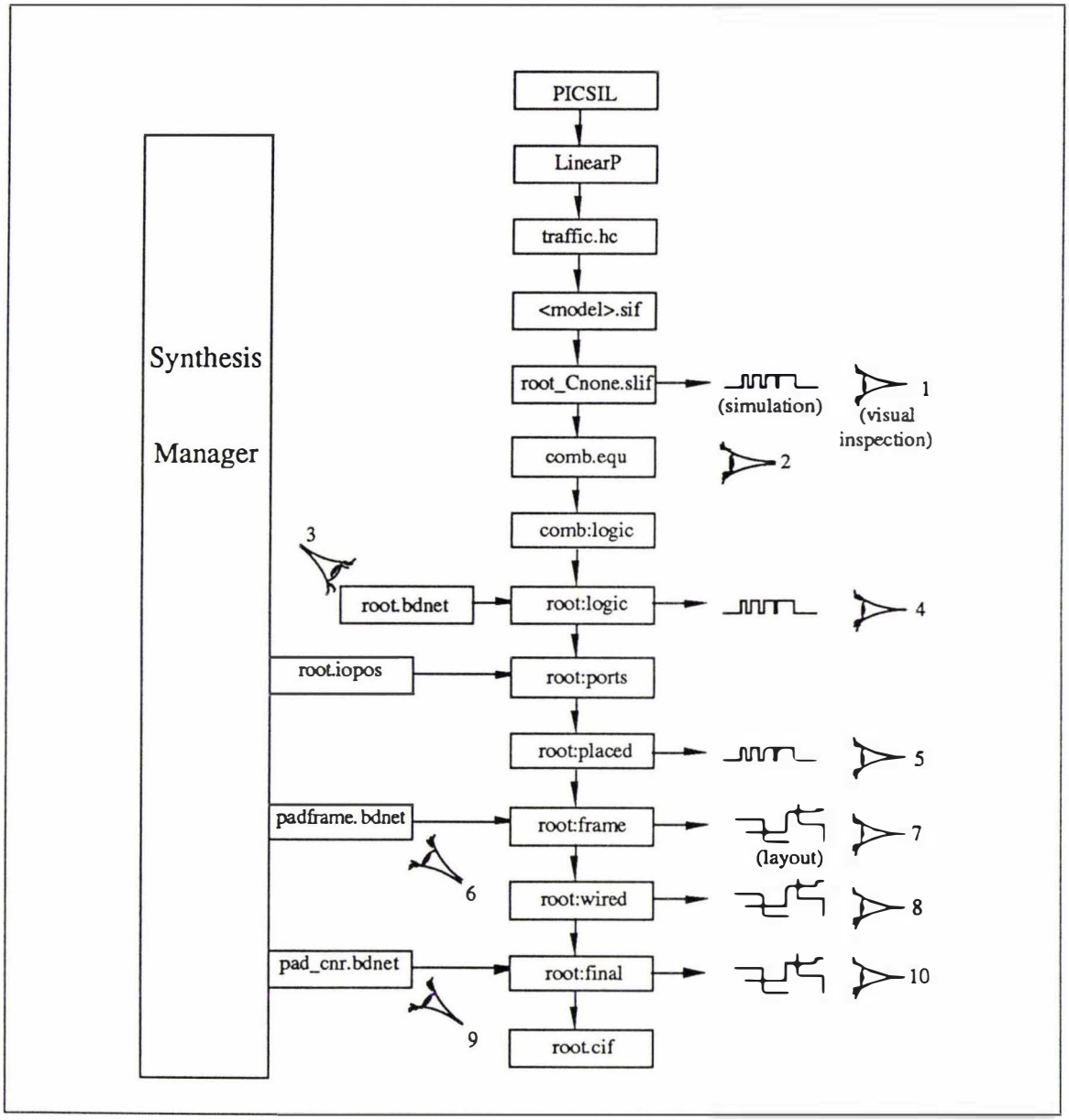


Figure 7.13 Traffic light controller : Test Points

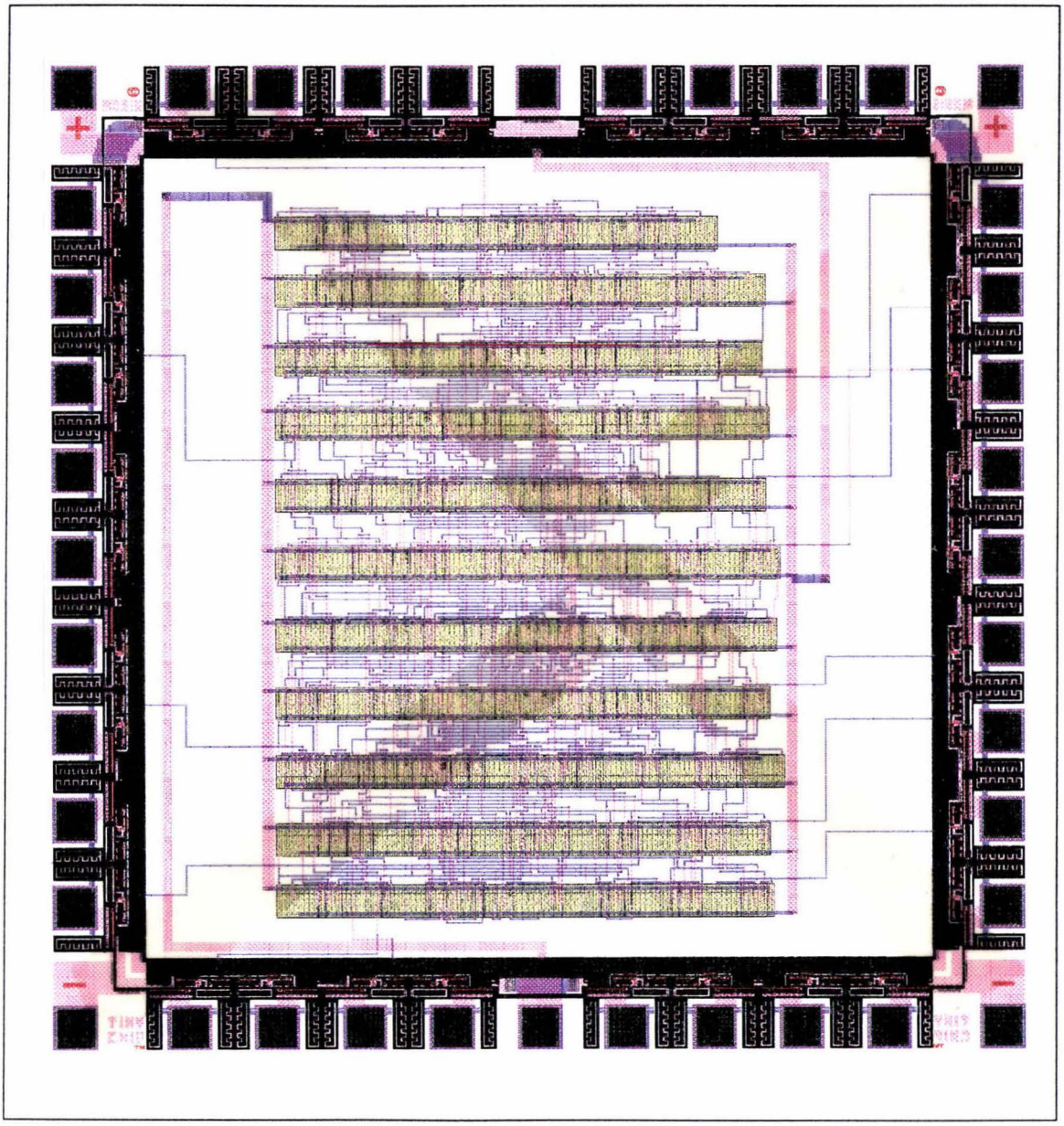


Figure 7.14 Final chip layout produced for traffic light controller

Chapter 8

Conclusions

This research has explored the application of Software Engineering's Structured Analysis (SA) techniques to complexity management in VLSI design. The primary tool of SA, the data flow diagram, is a simple graphical language which can greatly increase the understanding of a design description.

To ascertain the suitability of using SA techniques for VLSI design, a number of design exercises have been conducted. The initial design exercises revealed that, with a number refinements, DFD's were well suited to support VLSI design. The process of making the refinements was iterative, involving adding new constructs and evaluating them against the design exercises, eventually resulting in the current form of a notation called PICSIL. The notation was integrated into the design process, using tools for capturing PICSIL representations (the PICSIL editor) and generating a layout from the captured design (the PICSIL synthesis manager).

The PICSIL editor provides for the direct input and modification of a design expressed in a notation combining graphics and text. It has been designed to be relatively simple to use, and is based on a consistent graphical user interface, which makes use of multiple windows, menus and buttons.

The PICSIL synthesis system allows designs specified using the PICSIL editor to be interfaced to a lower level synthesis path using the Olympus system for high level synthesis and the Octtools system for logic and physical level synthesis. A Synthesis Manager has also been developed which automatically issues commands to drive both Olympus and Octtools allowing the synthesis path to be fully automated.

8.1 Conclusions

In the first Chapter, the general problem of designing complex digital devices was explored, and characterisation of an ideal design environment developed:

> *A design environment should maximise the opportunity for creative input at the highest levels of abstraction, and minimise the need for input of automatically-derivable lower-level information. An ideal digital IC design environment would therefore:*
>
> - *harness a designer's creativity effectively by concentrating on the system level of abstraction;*
> - *recognise the natural vocabulary for system-level ideas by allowing direct graphical input;*
> - *conserve the designer's effort for creative work, by automating the mechanical chore of layout synthesis.*

How has PICSIL system produced in this project lived up to this set of ideals?

Designing at the System level

In practical terms, designing at the system level was perhaps the easiest goal to achieve. Existing HDLs include many constructs which allow designers to work in terms of abstract building blocks, and the necessity to design a chip in terms of its layout has long been superseded in design systems which have allowed the synthesis of progressively higher-level computational modules, from Boolean equations, to state machines, to algorithms. Similarly, the arrangement of these computational building blocks has been facilitated by systems like Olympus which provide tools for functional decomposition, and for communication synchronisation.

The PICSIL project was therefore approached from the viewpoint that this prior development provides a rich seam of concepts to be mined; there is little point in inventing a complete structure, different only in detail from existing Hardware Description Languages. Familiar concepts adopted in the PICSIL system are functional decomposition, data flows, algorithmic design, and automatic synchronisation of building blocks. These concepts are common to many Hardware Description Languages, but are based in particular on those available in HardwareC.

Practical experience with a number of design examples has shown that the resulting language offers, with some small extensions, a sufficiently powerful vocabulary. In order to prevent PICSIL from growing inordinately large, extensions were only adopted if they offered

- generality,
- conciseness,
- system level abstraction.

The added constructs are: addressable data stores accessible from a number of other devices, a general-purpose data routing device, a generator construct which allows easy representation of repeated subcircuits, and a properly formalised representation of control definition.

Complex digital designs are naturally graphic

In order to demonstrate that a mainly graphic representation forms a natural input representation, a graphic language was designed, an editor for capturing designs in the language was implemented, and a variety of designs were created in the resulting environment.

The development of a suitable graphical language started with the selection of a suitable base notation. Data flow diagrams, the primary tool of structured analysis, with a number of refinements, were identified as a suitable framework for a graphical Hardware Description Language. The necessary refinements were made in an iterative manner, adding new constructs and evaluating them against a set of design exercises. The original view that everything could be represented most naturally in a graphical format was revised, and algorithmic aspects of designs are represented textually, using a language based closely on HardwareC.

Once PICSIL had been sufficiently well developed, the PICSIL representations of three of the design exercises were compared with equivalent representations in a conventional HDL. These comparisons have shown that the resulting PICSIL representations are shorter, more informative and easier to produce than those expressed in a conventional text-based HDL.

Implementation of the editor acted as a test of the completeness and orthogonality of the constructs in the language, and proceeded partly in parallel with their development. Consequently, some parts of it were developed several times. However, it was felt that developing an optimum set of constructs should necessarily precede refining the editor, and the current editor is functional - even omitting some designed language constructs, - rather than optimum.

Implementing the editor generated most of the practical development work of the PICSIL project; an editor was successfully implemented, and the ten design exercises were successfully and simply captured.

Even though it lacks superficial gloss, the PICSIL editor is capable of capturing designs expressed in a more natural vocabulary than any other Hardware Description Language currently available. Also, when evaluated against Sequin's (Sequin, 1983) guidelines, it has been found that PICSIL does provide the necessary features to effectively manage design complexity. PICSIL has mixed graphics and text, forming the basis of a new generation of Hardware Description Languages which will become the designer's natural choice for unambiguous representation of complex digital designs.

Automating synthesis

The third property of an ideal system was that it would conserve the designer's effort for creative work, by automating the mechanical chore of layout synthesis. PICSIL has achieved this by the implementation of a synthesis manager which:

- converts PICSIL to Hardware C, where possible;
- synthesises SLIF code for the remaining constructs;
- drives the Olympus synthesis system to produce SLIF intermediate code, incorporating the outputs of both previous steps;
- provides a simple interface to allow batch mode simulation to be performed on SLIF;
- translates SLIF into input for the OCTTOOLS layout synthesis system;
- drives OCTTOOLS to generate a CIF layout which can be supplied to a silicon foundry for fabrication.

The layouts generated by the PICSIL synthesis system are almost certainly less efficient, in terms of space and execution speed than those which could be generated by hand, or even possibly by other synthesis systems. In particular, the mapping of controllers and datastores to a layout is seen as being particularly inefficient.

The implementation of the PICSIL synthesis system has, however, demonstrated that it is possible to map a design expressed in system- and algorithmic-level vocabulary into a chip layout automatically, and making the synthesis more efficient is seen as future work.

Three designs have been successfully synthesised using the PICSIL synthesis system. During the synthesis of these designs, a number of test points (including visual checks and simulations) were inserted into the synthesis path, which has shown that all the tools in the synthesis path have functioned correctly. A chip designed and synthesised using PICSIL System has been sent for fabrication.

8.2 Further Research

During the project, four areas for further research were identified: the PICSIL notation, the PICSIL editor, the PICSIL synthesis system, and the provision of facilities for testing designs.

PICSIL HDL

Use of graphics at the Algorithmic Level

The development of the current version of the PICSIL HDL has focused on the system level of design with the development of a graphical language based on data flow diagrams. A purely textual language has been used to define the algorithmic level aspects of a system and the use of graphics was not considered. It is possible that graphics may also be used to help manage design complexity at the Algorithmic level of abstraction.

One possible graphical representation that could help manage design complexity is the state chart (Levi and Agrawala, 1990). One HDL based on state charts called Envision-VHDL (Toomajanian, 1992) already exists, it does not however consider the system level aspects of a design.

Designs containing a Single Controller or Process

Several of the designs used as exercises during the evaluation of PICSIL could be described using a single controller. In these cases, a separate DFD is still required to define the inputs and outputs for the controller, which places an unnecessary burden on the designer. Future refinements to the PICSIL HDL would need to allow more natural definition of the I/Os of designs that are made up of a single controller or process.

Data Stores

Data stores can be defined only as linear blocks of memory in the current version of PICSIL. Allowing data stores to be defined using more complex structures such as queues and stacks would also provide useful abstractions to aid the designer.

PICSIL EDITOR

Editing Operations

Capturing designs using the current version of the PICSIL editor is marginally slower than creating hand drawn designs. To allow a designer to explore the design space of a problem more effectively, the input and editing operations performed by the editor should be faster than those involving pencil and paper. This will require more powerful editing operations such as block moves, and automatic rerouting of data flows between processes, to be incorporated into the editor environment.

Undo Operations

Exploring the design space for a problem is an iterative process of making changes to a design and then evaluating them. To support this iterative design process, features need to be included into the editor environment to allow a designer to resurrect a previous version of a design. Such features could range from a simple undo operation, to an interface to a version control system such as RCS (Tichy, 1985).

Reuse of Design Sections

Many designs contain sections which are identical to sections in other designs. To allow a designer to reuse these sections of old designs, and speed up the design process, the PICSIL editor would need to be extended. In the simplest form this would involve the addition of cut, copy and paste operations to allow a section of PICSIL diagram including any child diagrams and data dictionary entries to be moved from one design to another.

In a more comprehensive form, the editor would also need to include a library sub-system, allowing frequently used components to be stored for easy retrieval as required. In such a library sub-system data flow names connected to the included library module would need to be able to be mapped onto the names within the module.

PICSIL SYNTHESIS SYSTEM

In the present version of the PICSIL synthesis system, HardwareC is the main target language of the PICSIL complier. The only high level synthesis system to use HardwareC as input is the Olympus system, which currently has a number of limitations such as not supporting memory sub-systems.

As VHDL is the only standardised HDL a wide variety of tools have been developed which accept it as input. However, not all VHDL constructs are suitable for synthesis and a number of different subsets have been defined, making it difficult to develop tools which produce VHDL as output. This is likely to change in the near future, as the VHDL standard is currently under review.

Once a standard subset of the VHDL language has been defined for synthesis, the PICSIL compiler could be modified to produce VHDL, allowing the PICSIL environment to be interfaced to a wide variety of CAD tools to support different aspects of design, including synthesis. The modification of the PICSIL compiler should also introduce more type checking to the parser to ensure all syntax errors can be detected and reported within the PICSIL environment.

TESTING

Simulation is the most widely used means of verifying the functions of VLSI designs at the various levels of abstraction. The PICSIL environment currently provides a simple interface to allow simulation to be performed in batch mode. To assist a designer in verifying a design, an interactive environment is required. The environment should provide:

- an interface to allow a designer to generate input values interactively and/or automatically, dependent on previous and current signal values as in (Maurer, 1990).

- an interface to allow the results of a simulation step to be displayed. These results could be displayed using timing diagrams such as those used in the current version of PICSIL, or they could be displayed directly on the data flows of the PICSIL diagrams.

- a means to allow either a whole system to be simulated or a part of it to be simulated in isolation.

8.3 Concluding Remarks

To conclude, further research and development are required in order to make practical use of the PICSIL environment for the complete design of large and complex hardware systems. However, this research has shown the potential for a new generation of HDLs which allow a designer to concentrate on representing designs at the system level of abstraction, in the most natural vocabulary (graphics for systems aspects, text for algorithms), and to sidestep distractions from creativity by automating layout synthesis.

Bibliography

ACKERMAN, W.B. (1982): Data Flow Languages, *IEEE Computer* **15**, 2, pp. 15 - 25.

ANISIMOFF, D. (1992): *Personal Communication* .

APPERLEY, M.D., TZAVARAS, I., and SPENCE, R. (1982): A Bifocal Display technique for Data Representation. In *Eurographics '82*, Greenway, D.S. and Warmans, E.A. (Eds), North-Holland Publishing Company, pp. 27 - 43.

APPERLEY, M.D. (1987): *Personal Communication.*

ASHENDEN, P.J.(1990): *The VHDL Cookbook*, Department of Computer Science, University of Adelaide, South Australia.

BABB, R.G. (1982): Data-Driven Implementation of Data Flow Diagrams. In *Proceedings Sixth International Conference on Software Engineering*, pp. 309 - 318.

BABB, R.G. (1984): Parallel Processing with Large-Grain Data Flow Techniques, *IEEE Computer* **17**, 7, pp. 55 - 61.

BARBACCI, M.R. (1981): Instruction Set Processor Specifications (ISPS) : The Notation and its Applications, *IEEE transactions on computers* **C30**, 1, pp 24 -40.

BRAYTON, R.K., RUDEL, R., SANGIOVANNI-VINCENTELLI, A., and WANG, A.R. (1987): MIS: A Multiple Level Logic Optimisation System, *IEEE Transactions on Computer Aided Design* **CAD6**, 6, pp. 1062 - 1081.

BURGER, R.M. andHOLTON, W.C. (1992): Reshaping the Microchip, *Byte* **17**, 2, pp. 137 - 148.

BURNS, A. and KIRKHAM, J.A. (1986): The Construction of Information Management System Prototypes in Ada, *Software - Practice and Experience* **16**, 4, pp. 341 - 350.

CAVIN, R. K., and HILBERT, J.L. (1990): Design of Integrated Circuits : Directions and Challenges, *Proceedings of the IEEE* **78**, 2, pp. 418 - 435.

CHENG, E.K. and MAZOR, S. (1988): The Gensil Silicon Compiler, *Silicon Compilation*, Gajski, D.D. (Eds), Addison Wesley, pp. 361 - 405.

COELHO, C.N. (1991): *Personal Communication.*

DE MICHELI, G. (1990): Guest Editorial: High-Level Synthesis of Digital Circuits, *IEEE Design & Test of Computers* **7**, 5, pp. 6-7.

DE MICHELI, G., KU, D., MAILHOT, F., and TRUONG, T. (1990): The Olympus Synthesis System, *IEEE Design & Test of Computers Magazine* **7**, 5, pp. 37 - 53.

DEMARCO, T. (1978): *Structured Analysis and System Specification*, Prentice-Hall.

DENYER, P.B. and RENSHAW, D. (1985): *VLSI Signal Processing: A Bit Serial Approach*, Addison Wesley.

DILLINGER, T.E. (1988): *VLSI Engineering*, Prentice Hall.

DOCKER, T.W.G. (1989): *SAME : Structured Analysis Modelling Environment The Design of an Executable Data Flow Diagram and Dictionary system*, Ph.D. dissertation, Department of Computer Science, Massey University, Palmerston North, New Zealand.

DORAN, B. and TATE, G. (1972): "An Approach to Structured Programming", Department of Computer Science, Massey University, Publication, no. 6.

DUTT, N.D. and GAJSKI, D.D. (1990): Design Synthesis and Silicon Compilation, *IEEE Design & Test of Computers* **7**, 6, pp. 8 - 23.

GAJSKI, D.D. and KUHN, R.H. (1983): New VLSI Tools, *IEEE Computer* **16**, 12, pp. 11 - 14.

GAJSKI, D. and THOMAS, D. (1988): Introduction to Silicon Compilation, *Silicon Compilation*, Gajski, D. (Eds), Addison Wesley pp. 1 - 48.

GANE, C.P. and SARSON, T. (1979): *Structured Systems Analysis: Tools and Techniques*, Prentice Hall.

HATLEY, D.J. and PIRBHAI, I.A. (1987): *Strategies for Real-Time System Specification*, Dorset House.

HIRAYAMA, M. (1986): VLSI Oriented Asynchronous Architecture, *13th Annual International Symposium on Computer Architecture*, pp. 290 - 296.

INTEL, (1983): *Microprocessor and Peripherals*, Intel Corporation.

JACKSON, M.A. (1975): *Principles of Program Design*, Academic Press.

JHON, C.S., SOBELMAN, G.E., and KREKELBERG, D.E. (1985): Silicon Compilation Based on a Data Flow Paradigm, *IEEE Circuits and Devices Magazine* **1**, 5, pp. 21 - 28.

JOHNSON, S.C. (1975): "Yacc - Yet another Compiler Compiler", Technical Report, Computer Science, Bell Laboratories, no. 32, New Jersey.

KAM, M.C. and HELLESTRAND, G.R. (1990): The Vast VLSI Architecture and Design Environment. In *Proceedings of the 9th Australian Micro-electronics Conference*, The Institution of Radio and Electronics Engineers Australia, pp. 145 - 250

KELLER, R.M., LINDSTROM, G., and PATIL, S.S. (1980): Data-Flow Concepts for Hardware Design. In *COMPCON 80*, pp. 105 - 111.

KERNIGHAN, B.W. and RITCHIE, D.M. (1978): The C Programming Language, *Prentice Hall.*

KU, D. and DE MICHELI, G. (1990): "HardwareC - A Language for Hardware Design Version 2.0", Computer Systems Laboratory, Stanford University, no. CSL-TR-90-419.

LABVIEW, (1990): *Lab View 2 User Manual*, Part Number 320244-01, National Instruments Corporation.

LESK, M.E. and SCHMIDT, E. (1975): "Lex - A Lexical Analyzer Generator", Technical Report, Computer Science, Bell Laboratories, no. 39, New Jersey.

LEVI, S. and AGRAWALA, A.K. (1990): *Real Time System Design*, McGraw-Hill.

LI, E.L.H. and HELLESTRAND, G.R. (1990): Design and Generation Issues of VLSI Memory Sub-systems. In *Proceedings of the 9th Australian Microelectronics Conference*, The Institution of Radio and Electronics Engineers Australia, pp. 65 - 68.

LYONS, P.J. and MCGREGOR, A.J. (1990): *Multipath Local Area Network*, U. S. Patent 4953162.

LYONS, P.J. (1991): *Personal Communication.*

MANO, M.M. (1982): *Computer System Architecture*, Prentice Hall.

MAURER, P.M. (1990): Dynamic Functional Testing for VLSI Circuits, *IEEE Design and Test of Computers 7*, 6, pp. 42 - 49.

MCFARLAND. M.C., PARKER, A.C., and CAMPOSANO, P. (1990): The High-Level Synthesis of Digital Systems, *Proceedings of the IEEE 78*, 2, pp. 301 - 318.

MEAD, M. and CONWAY, L. (1980): *Introduction to VLSI Systems*, Addison Wesley.

MOTOROLA INC, (1983): *8-Bit Microprocessor and Peripheral Data.*

MYERS, B.A. (1990): Taxonomies of Visual Programming and Program Visualisation, *Journal of Visual Languages and Computing 1*, 1, pp. 97 - 123.

NASSI, I. and SHNEIDERMAN, B. (1973): Flowchart Techniques for Structured Programming, *ACM SIGPLAN Notices 8*, 8, pp. 12 - 26.

OCTTOOLS, (1991): *Tool User Guides and Tutorials, Octtools version 5.0*, Electronics Research Laboratory, University of California, Berkeley.

ORBIT, (1991): *Foresight Users Manual*, Orbit Semiconductor, 1230 Bordeaux Dr., Sunnyvale, California 94089, 1.4.

SEQUIN, C.H. (1983): Managing VLSI Complexity : An Outlook, *Proceedings of the IEEE 71*, 1, pp. 149-435.

SHAHDAD, M., LIPSETT, R., MARSCHNER, E., SHEEHAN, K., COHEN, H., WAXMAN, R., and ACKLEY, D. (1985): VHSIC Hardware Description Language, *IEEE Computer 18*, 2, pp. 94 - 103.

SHNEIDERMAN, B. (1983): Direct Manipulation: A Step Beyond Programming Languages, *IEEE Computer 16*, 8, pp. 57 - 69.

SIEWIOREK, D.P., BELL, C.G., and NEWELL, A. (1982): *Computer Structures: Principles and Examples*, McGraw-Hill.

SOMMERVILLE, I. (1989): *Software Engineering*, Addison-Wesley, Third edition.

SOUTHARD, J.R. (1983): MacPitts : An Approach to Silicon Compilation, *IEEE Computer 16*, 12, pp. 74 - 82.

STRONG, D. (1987): DataLink - Running Data Flow Diagrams. In *Proceedings 10th New Zealand Computer Conference ('Putting Computers to Work')*, pp. 26 - 28.

SUNVIEW, (1988): *Sunview1 Programmers Guide*, Sun Micro Systems, Inc., Version 4.

SZPAKOWSKI, M. (1989): Prograph Raises OOP To New Height, *MacTutor 5*, 7, pp. 66 - 75.

TANENBAUM, A.S. (1990): *Structured Computer Organisation*, Prentice-Hall.

THOMAS, D.E., DIRKES, E.M., WALKER, R.A., RAJAN, J.V., NESTOR, J.A., and BLACKBURN, R.L. (1988): The System Architect's Workbench. In *Proceedings of the 25th Design Automation Conference.*

THOMAS, D.E. and MOORBY, P. (1991): *The Verilog Hardware Description Language*, Kluwer Academic Publishers.

TICHY, W.F. (1985): RCS - A System for Version Control, *Software - Practice and Experience* **15, 7**, pp. 637 - 654.

TOOMAJANIAN, G. (1992): *Graphical Behaviour Capture to VHDL*, Personal Communication.

TSE, T.H. and PONG, L. (1991): An Examination of Requirements Specification Languages, *The Computer Journal* **34**, 2, pp. 143 - 152.

WALKER, R.A. and CAMPOSANO, R. (1991): *A Survey of High-Level Synthesis Systems*, Kluwer Academic Publishers.

WARD, P.T. (1986): The Transformation Schema: An Extension of the Data Flow Diagram to Represent Control and Timing, *IEEE Transactions on Software Engineering* **SE12**, 2, pp. 198 - 210.

XVIEW, (1990): *Xview Programming Mannual*, O'Reilly and Associates Inc.

YENG, P. and REES, D. (1991): "A Comparison of Hardware Description Languages", Department of Computer Science, University of Edinburgh, Internal Report, no. CSR-13-91.

Appendix 1
PICSIL
Data Dictionary Language

This appendix describes the PICSIL Data dictionary language in detail. To illustrate a number of constructs in the language the image processing example introduced in Chapter 3 (page 27) is used.

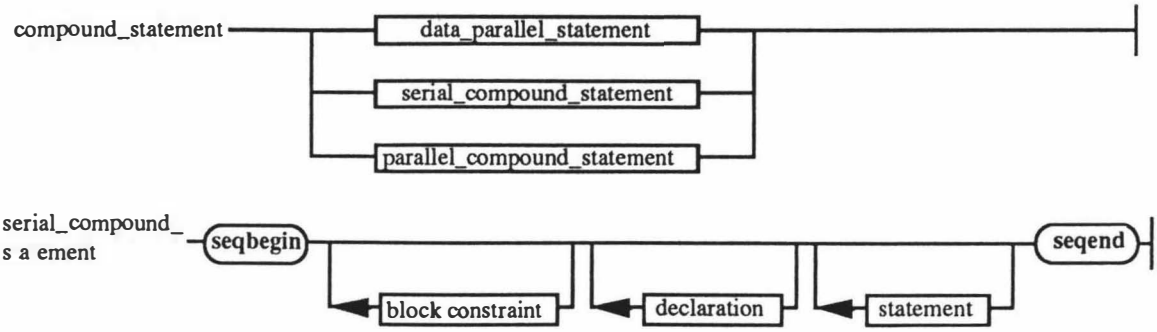
A1.1 Primitive Process Definitions (PSPECs)

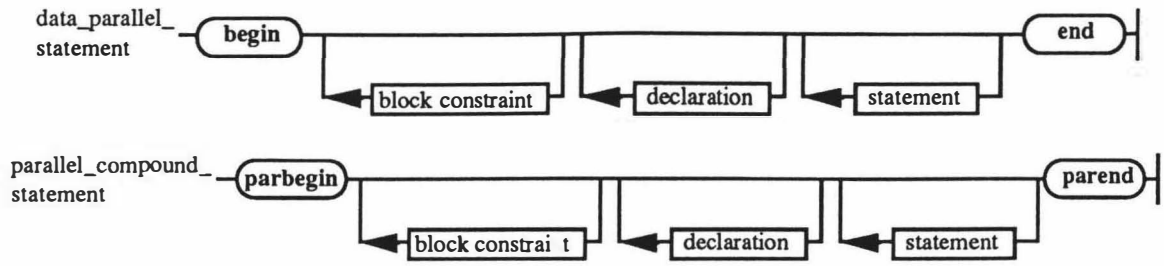
Every primitive process in a PICSIL definition requires a data dictionary entry with the following syntax:

where process_id and name correspond to the id and name of the process in the parent data flow diagram. The body is made up of a compound statement which is used to group a series of declarations and statements together to define the behaviour of the process.

Compound Statements

Three types of compound statements exist in the language to support varying degrees of parallelism. The three types are *serial, data-parallel* and *parallel* compound statements. The syntax of the compound statement is :





A serial compound statement is enclosed between the statements **seqbegin** and **seqend**, and generates hardware which performs the operations specified by the enclosed statements in the order in which those statements are written. For example consider the following serial compound statement :

```
seqbegin
   a = b + c;
   c = d + e;
seqend
```

The first statement will finish execution before the second one starts. A serial compound statement completes execution when the last statement in the compound statement completes execution.

A data-parallel statement is enclosed between the statements **begin** and **end**, and generates hardware which may perform the enclosed statements in parallel, provided no data-dependencies exist between them. For example consider the following data parallel statement:

```
begin
   a = b + c;
   d = b + f;
   g = a + d;
end
```

The first and second statements can execute in parallel as the second statement does depend on the result of the first statement. The third statement depends on the results produced from both the first and second statements so it cannot be executed until they have both finished. The synthesis system detects sets of statements with data dependencies and generates hardware to prevent conflict from this cause. In addition it is capable of choosing alternative degrees of parallelism for the other statements according to space. A data parallel statement completes execution when the last statement in the compound statement completes execution.

A parallel compound statement is enclosed between the statements **parbegin** and **parend** and generates hardware which forces the enclosed statements to be executed in parallel. The values of the variables that are referenced inside a parallel compound statement are the values they had just before it was entered. Consider the following parallel compound statement:

```
parbegin
   a = b;
   b = a;
parend
```

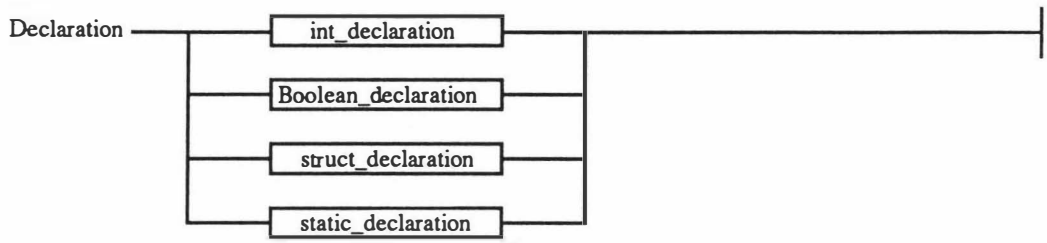
Both statements are executed concurrently and the effect is to swap the values of "a" and "b". As all statements are executed concurrently a single assignment convention is applied. I.e. a variable may only have one value assigned to it in a parallel compound statement. A parallel compound statement finishes execution when all its statements have completed execution.

Variables and Constants

There are two types of data entity that can be used in the PSPECs - constants and variables. Constants can be decimal, hexadecimal or binary depending on their prefix. A sequence of digits is taken to be hexadecimal if it is prefixed by **0X** or **0x**. A binary constant is prefixed by **0b** or **0B** and a decimal constant has no prefix.

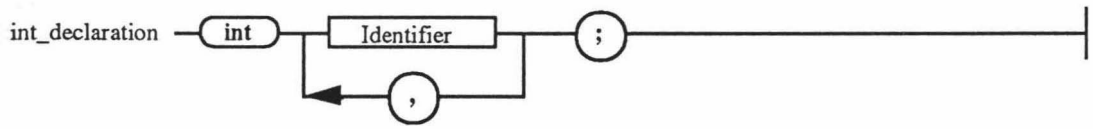
Variables are named via *identifiers* and are used to store the result of a computation or use the results of previous computations. The value of a variable is defined to be the value most recently assigned to it, where data is defined as the value returned by a procedure call, or a value supplied by a continuous, discrete or store flow or a binary or unary expression.

There are three major variable types in the language: **boolean, int** and **static.**

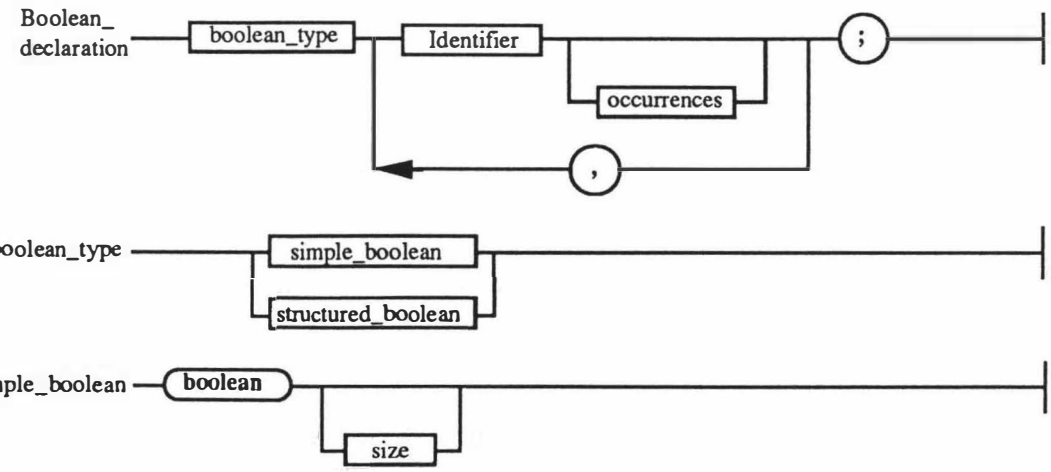


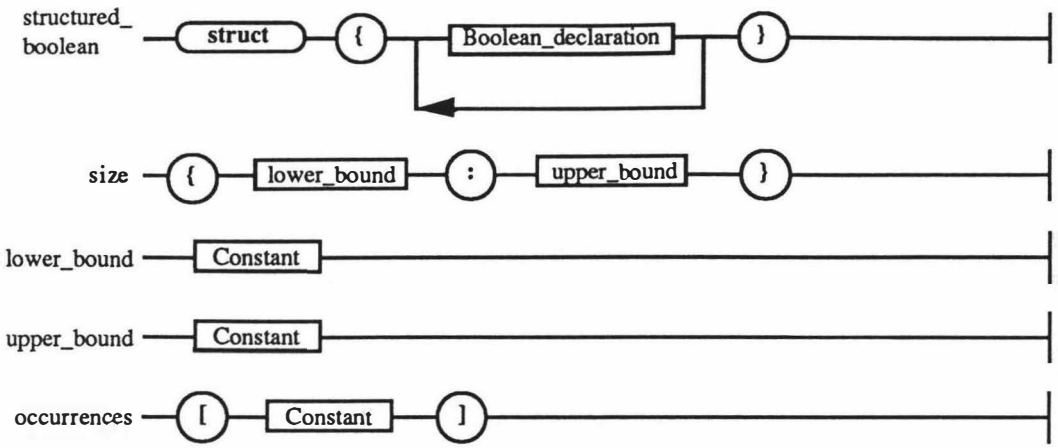
All variables must be declared before use, and can be declared within any compound statement in the description. Following the semantics of block structured languages, a variable is only visible within the compound block within which it is declared. A variable with the same name at a deeper nesting block level will override any current definition of the variable. No global variables are allowed, as they allow side effects that are not explicitly identified. If data is to be shared between two processes then the data should be explicitly specified as parameters to the two processes or stored in a data store.

Integer variables are scalar quantities which are only used for the convenience of the description as they are not synthesised into hardware. Thus integer variables are only valid when their values can be resolved at compile time. They are mainly used as indices to constant iteration loops (for-loops), and as indices for accessing Boolean vectors.



Boolean variables are the principle data type in the language and can represent one or more signals, where each bit of the variable can be independently set. Boolean variables do not retain their values between successive process executions. A variable of Boolean type may represent a single two's complement number with one or more bits.





The size of each word in a Boolean object can be given by the optional size expression. For example boolean{0:7} value defines an eight bit Boolean word named value. If the size of the Boolean word is not specified then as default, it is taken to be one. The range of values a Boolean can represent depends on its size. For example, a scalar (single bit) Boolean can assume the values 0 and 1, and a vector (n bits) can assume the values ranging from $-2^{n/2}$ ... $2^{(n/2)-1}$.

It is also possible to specify an array of Boolean variables using the *array_construct*. For example boolean{0:7} mark[15] declares 15 occurrences of the variable mark addressed from 0 to 14 (i.e. mark[10] references the eleventh item in the array).

To allow more complex Boolean objects to be defined a **struct** type also exists. An object of struct type allows one or more Boolean objects (including other **struct** types), to be grouped together under a single name.

Each pixel in our image processing example comprises a linear combination of red, green and blue intensities, which can be stored in a **struct**.

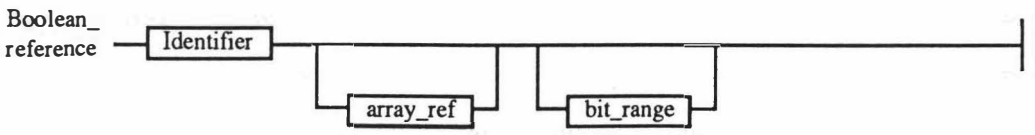
```
struct {
    boolean { 0:7 } red;
    boolean { 0:7 } green;
    boolean { 0:7 } blue;
} colour_pixel
```

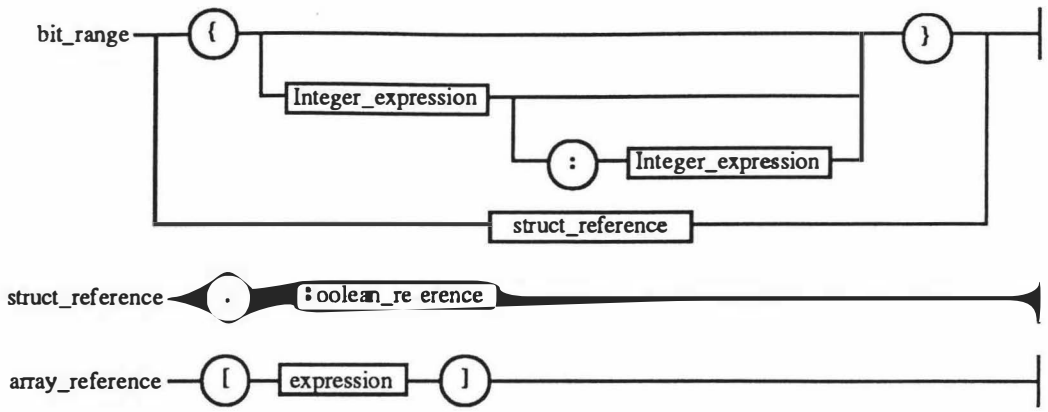
The resulting variable, colour_pixel, is 24 bits wide, where the red component is contained in the first eight bits (i.e. 0 : 7), the green component in the next eight bits (i.e. 8 : 15) and the blue component in the last eight bits (i.e. 16 : 23).

If more complex groupings of Boolean objects are required, then subsequent **struct** declarations can appear within a **struct** declaration. For example if one of the processes in the image processing example worked on a group of four pixel's then a variable quad could be defined as:

```
struct {
    struct {
            boolean {0:7} red, green, blue;
    } top_left, top_right, bottom_left, bottom_right;
} quad
```

The following syntax is used for referencing Boolean variables including **structs** and their component parts:





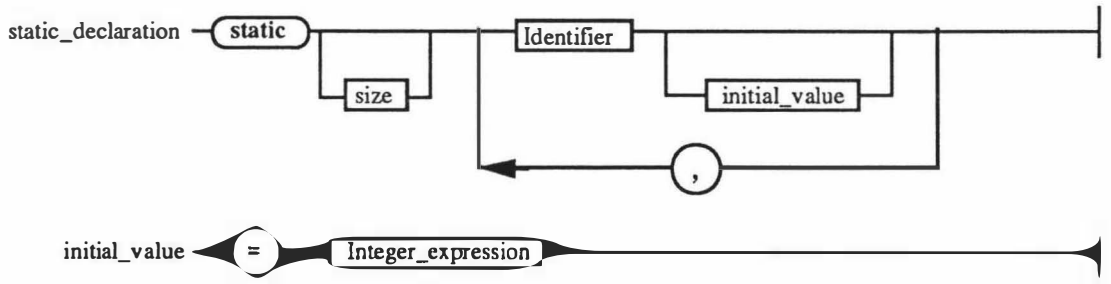
If the bit range is omitted or an empty bit range is specified (i.e. {}) then the whole variable is referenced. If only a single integer_expression is specified in the bit range then only the specified bit will be accessed. For example value{0} will reference the first or least significant bit of value while value{4} will reference the fifth bit.

If two integer expressions are given in the bit_range then all the bits between and including the two integer expressions will be referenced For example value{0:3} will reference the four least significant bits of value. The order of the integer expressions does not influence the result as value{0:3} is equivalent to value{3:0}. If a Boolean or struct variable is declared as an array then a location within the array must be specified, as only a single item can be referenced at a single time.

Any range of bits can also be specified with a struct variable. For example colour_pixel{12 : 19} would reference the four most significant bits of the green component and the four least significant components of the blue one.

If a member of a struct variable is to be referenced by name, then the struct_reference construct is used, where the identifier before the "." or dot operator specifies the struct name and Boolean reference after the dot specifies the member to be referenced. For example pixel.red{0:3} will reference the four least significant bits of the member red in the complex Boolean variable pixel. If more complex groupings are used then any member or part of a member can be referenced by using several occurrences of the dot operator. For example quad.bottom_left.red will reference the whole of the red component of the bottom_left pixel of the complex Boolean quad.

Static variables are similar to simple Boolean variables, with the semantic difference that their values are retained across process executions and procedural invocations.



Identifier represents the name of the variable. Static variables may be initialised to a given value which is assigned to the variable when the system is reset. If an initial value is not specified, then it is set to zero. The syntax for referencing a static variable is the same as that for referencing a Boolean variable.

Expressions

Constants and variables may be combined using binary and unary operators to form expressions. Figure A1.1 shows the four major types of expression in the language: arithmetic, logical, relational and auto.

rithmetic Operators

The binary arithmetic operators are "+", "-", "*" and "/". There is a unary "-" but no unary "+". The "+" and binary "-" have the same precedence, which is lower than the precedence of "*" and "/" which in turn is lower than unary minus.

Logical Operators

The binary logical operators are "&", "|" and **xor** (or "^") corresponding to bitwise AND, bitwise OR and bitwise XOR operations. The unary "!" complement operator produces the bitwise complement of a given value or variable.

A value or variable may be shifted by the shift ("<<" and ">>") and rotate (**rl** and **rr**) binary operators. For example value >> 3 will shift all the bits of value three places, losing the three least significant bits and placing 0s in the most significant bits. The binary concatenation operator "@" concatenates the left operand as the most significant portion with the right operand as the least significant portion. For example, if x{0:3} = 0100, y {0:3} = 0111, then b{0:7} = x{0:3} @ y{0:3}; will give b{0:7} the value 01000111.

Relational Operators

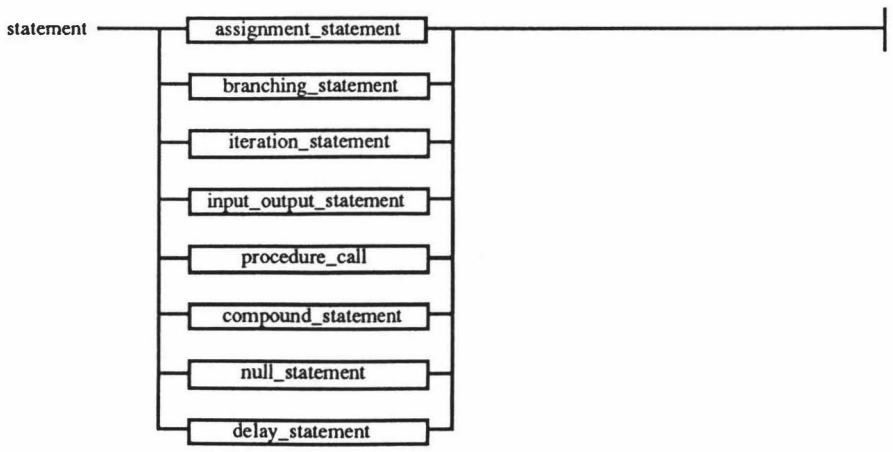
The relational operators consist of ">", ">=", "<" and "<=" all with the same precedence. Just below them in precedence are the equality operators "==" and "!=" which have the same precedence. The relational operators have lower precedence the arithmetic operators.

Auto Increment and Decrement

The auto increment ("++") and auto decrement ("--") operators are similar to those in C, and add 1 or subtract 1 from a variable respectively. The major difference from the auto increment and decrement operators of C is that the auto increment or decrement expression cannot be referenced. That is, b = a++; is illegal.

Statements

The language has a number of different types of statement:



TYPE	OPERATOR	DESCRIPTION
Arithmetic	+	binary addition
	-	binary subtraction or unary minus
	*	binary multiplication
	/	binary division
Logical	&	bitwise AND
	|	bitwise OR
	xor *or* ^	bitwise XOR
	!	unary bitwise NOT
	@	binary concatenate
	rr	rotate right
	rl	rotate left
	>>	shift right
	<<	shift left
Relational	>	greater than
	<	less than
	>=	greater equal
	<=	less equal
	!=	not equal
	==	equal
Auto	++	auto-increment
	- -	auto-decrement

Figure A1.1 Valid operators in the PICSIL notation (after Ku & De Micheli, 1990)

Assignment Statements

An assignment to a variable modifies the value of the variable. Both variables (either int, Boolean, and static) and continuous data flows can have values assigned to them.

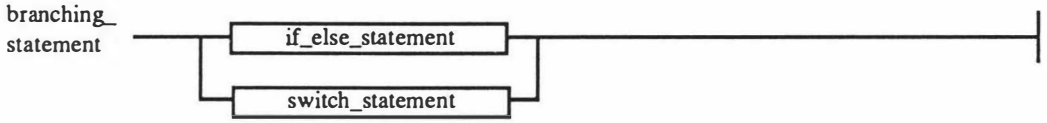
Identifier can be either a local variable or an exported continuous flow. Expression is an arithmetic, logical, or relational expression, or a function call.

Only constants or integer expressions can be assigned to integer variables. There is no restriction on the values that can be assigned to Boolean variables.

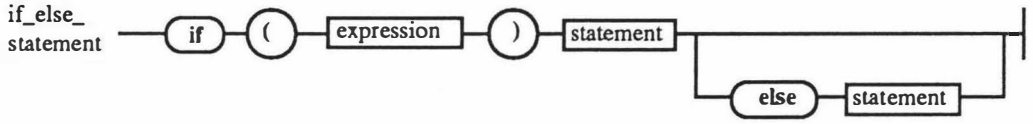
Both assignment statements and write statements can be used to update the value of a continuous flow. However, if a continuous flow is updated by a write statement, all assignments are ignored. If there is no write statement, the last assignment to the continuous flow during execution of the process is the one which takes effect.

Branching Statements

The *if-else* and *switch* branching statements allow conditional execution of sections of code depending on the value a conditional expression.



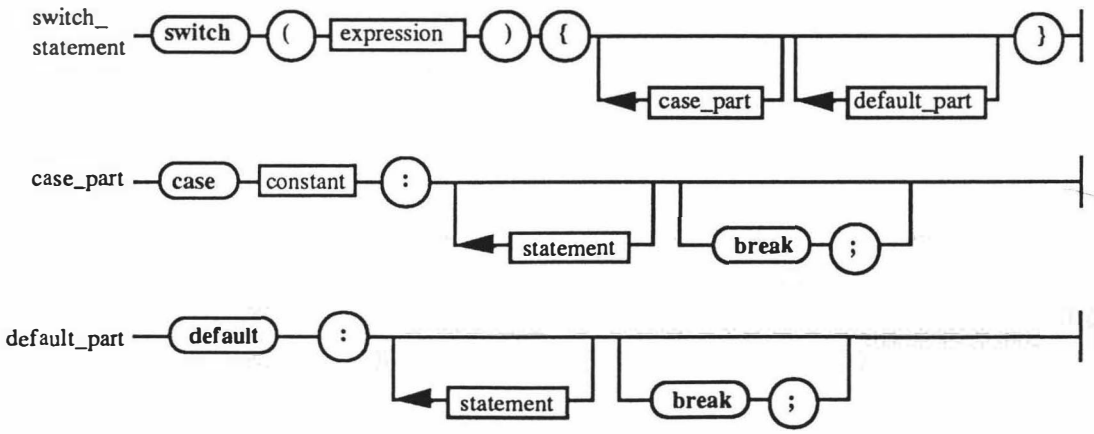
If-Else Statement



If the expression evaluates to a nonzero ("true") value, then statement-1 is executed. Otherwise, if the else part is specified, statement-2 is executed instead. The expression must evaluate to a single bit value, and can be either a variable, or any arithmetic, logical or relational expression. An example *if-else* statement:

```
if ( current_pixel. red > current_pixel.green)
    biggest_value = current_pixel.red;
else
    biggest_value = current_pixel.green;
```

Switch Statement



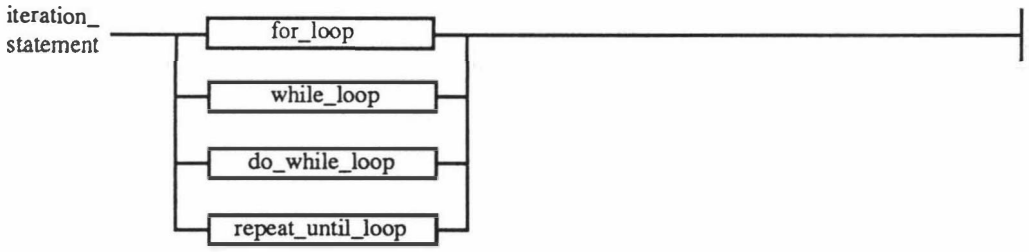
The switch statement chooses between several alternative groups of statements. The expression is evaluated and compared to the case constants. The group of statements located between the matching case constant and either the first subsequent break or the end of the switch statement

is executed. However if the expression does not match any of the case constants and a default part exists then the statements following the default are executed. If no expression matches and a default part is absent, none of the statements of the switch is executed. The following example shows the use of switch statement to select among a set of operations based on the variable opcode.

```
switch(opcode) {
case 0b00:
    result = a + b;
    break;
case 0b01:
    result = a - b;
    break;
case 0b10:
case 0b11:
    result =a & b;
    break;
default:
    result = 0;
    break;
}
```

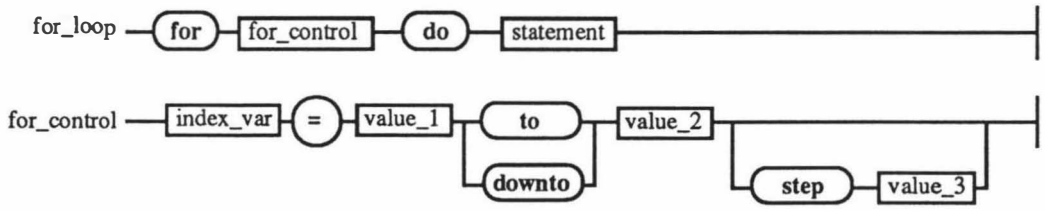
Iteration statements

There are two types of iterative loop construct in the language, *for-loops* and (three variations of) the *while-loop*.



For Loop

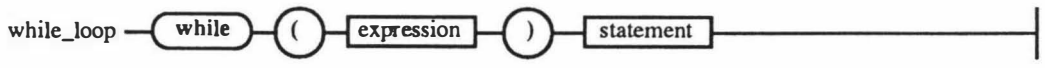
The *for-loop* is a constant bound iteration on a given integer variable (i.e. the number of iterations is bound at compile time).



Value1, value2 and value3 can be any constant or integer expression. The optional step clause defaults to one if not specified. The variable index_var must be an integer variable.

While Loop

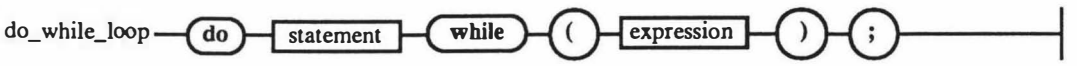
The while loop is a data-dependent iteration on a given Boolean expression which is evaluated before each iteration of the loop.



The loop body executes until *expression* (any single bit quantity) evaluates to 0.

Do while Loop

In addition, a variant of the while loop executes the body first prior to evaluating the loop exit condition.

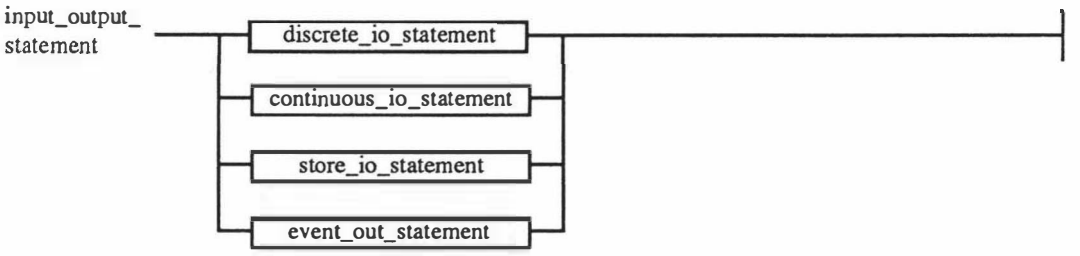


Repeat Until Loop

Finally a third variant of the data-dependent loop is the repeat-until loop, where the loop body executes repeatedly until the expression evaluates to non-zero.

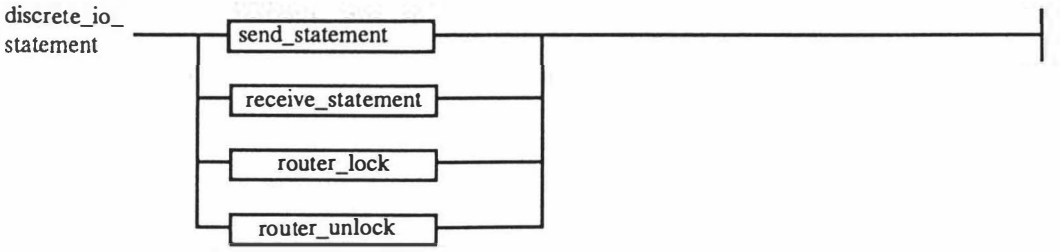
Input/Output Statements

The way in which data is transferred to and from processes is dependent on the flow type over which it is being transferred. Each of the four flow types discrete, continuous, store and event has its own set of I/O statements.

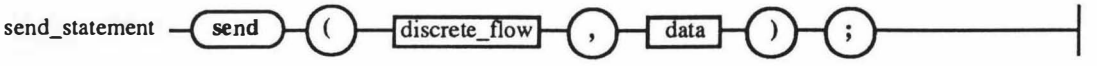


discrete flows

Data transfer takes place on a discrete flow using a predefined communications protocol (implemented in the synthesis of the design). This protocol is based on a send-receive message passing scheme. When data is to be sent to or received from a discrete flow the message passing protocol suspends the execution of the process sending or receiving it until the transfer (message) is acknowledged or received from the flow. There are three primitive I/O operations associated with discrete flows: send, receive and msgwait.

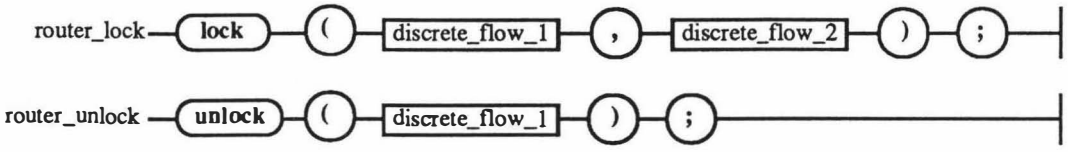


The send statement transmits a message on a given discrete flow. The sending process will wait until the receiving process issues a receive, whereupon the data transfer takes place. The syntax is :



where *data* is the message to be sent over *discrete_flow*. The definition of the discrete flow must match the definition of data, as sending data on a portion of a data flow is not allowed. An example of a send statement in the process perform_mapping_stream_1 (Figure 3.4, page 28) to send the value of mapped_pixel over the discrete flow mapped_stream_1 is:

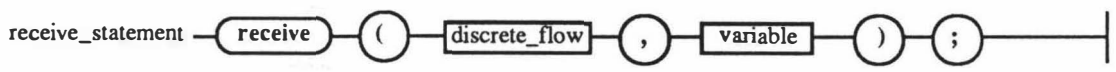
```
send(mapped_stream_1, mapped_picsil);
```



Before data can be sent along a discrete flow which is imported by a router a channel through the router, to one of its export flows must first be locked using the router lock statement in the sending process. If a process executes a router lock statement and the channel is currently locked by another process then the first processes execution is suspended until the channel becomes available. Once a channel through a router has been locked then a process can use a send statement to send data along the discrete flow to the router. The channel through the router remains locked until either a router unlock statement is executed or the process finishes its current execution. An example process definition for the process receive_char_port_1 in Figure 3.12 (page 35) is :

```
process receive_char_port_1
{
    boolean{0:7} address
    boolean{0:7}current_char;
    receive(inport_1,address);
    switch (address{0:1}) {
        case 0b01 :
            lock(from_1,to_4);
            break;
        case 0b10 :
            lock(from_1, to_5);
            break;
        case 0b11 :
            lock(from_1,to_6);
            break;
    }
    do {
        receive(inport_1,current_char);
        send(from_1,current_char);
    until (current_char == ETX);
    unlock(from_1);
}
```

A receive statement accepts a message off a discrete flow, and will wait until a message is sent on the discrete flow. The syntax of the receive statement is:



where variable is the name of the variable the message will be assigned to, and discrete_flow is the name of the flow the message will be received from. The definitions of the discrete_flow and buffer must be the same size, as a receive from a portion of a channel is not allowed. For the process add_images to receive values off the discrete flows mapped_stream_1 and mapped_stream_2 the following receive statements are used:

```
pixel_1 = receive(mapped_stream_1);
pixel_2 = receive(mapped_stream_2);
```

The msgwait command is a query that returns a single bit Boolean flag indicating whether the specified input discrete flow has data waiting to be received. The syntax of the msgwait command is:

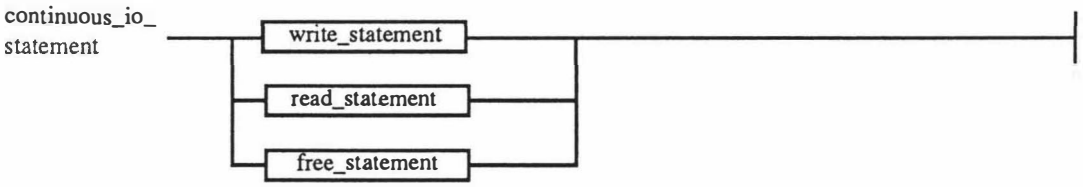
which returns a scalar flag which can be assigned to a Boolean variable or used as an expression within one of the conditional or looping constructs. The flag will be true if there is a message pending and false otherwise. The following example show the use of the msgwait statement.

```
if (msgwait(line_1)
    current_data = receive(line_1);
else if (msgwait(line_2)
    current_data = receive(line_2);
/*process current_data*/
```

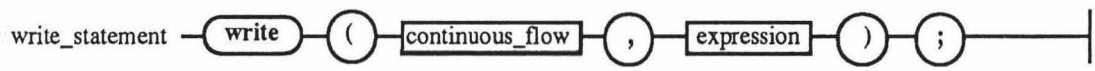
Line_1 and line_2 are both discrete flows. If a message is waiting on line_1 then it is received an assigned to the variable current_data. If no message is waiting on flow_1 a check is made to see if data is present on flow_2, in which case it is received.

Continuous Flows

No communications protocol is implemented by the synthesis system for continuous flows and the I/O statements access the values on the flows directly. The three I/O statements associated with the continuous flow are the write, read and free statements as outlined below.



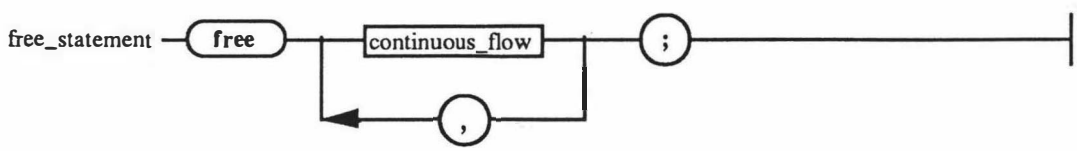
A write statement writes a given value to the corresponding continuous flow. The syntax of the write statement is



where continuous_flow can be either the entire continuous_flow or specific subranges of it. Any value written to a continuous flow is output immediately and remains on it until either the next write or free statement. For example consider a single bit continuous flow called pulse exported from a primitive process. The statements:

```
write(pulse, 0);
write(pulse, 1);
write(pulse, 0);
```

will cause a pulse to be generated on the pulse continuous flow. A free statement sets the corresponding bidirectional continuous flow to a high impedance float state. The syntax of the free statement is :



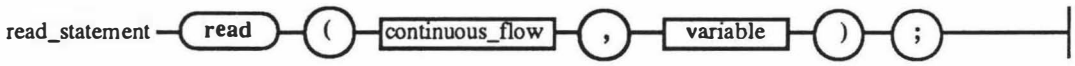
where continuous_flow can be either the entire continuous_flow or specific subranges of it. Any write to a bidirectional continuous flow that has been set to a float state will assign the

new value. For example consider a process which interfaces to a data bus in a microprocessor system through the bidirectional continuous flow data. The following statements write data to the bus then allow data to be received from it:

```
write data = result;
/*Another process may read data from the bus*/
free data;
/*Another process may write data from the bus.
new_data = read(data);
```

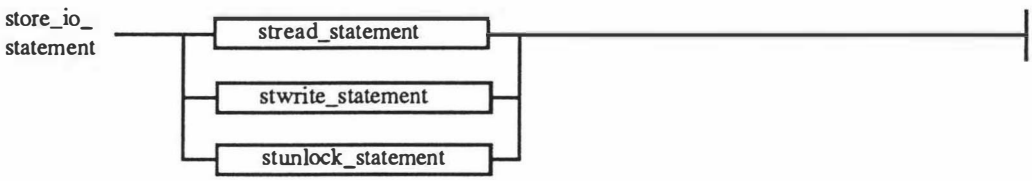
It is left up to the designer to ensure that only one process writes to bidirectional flow at a time.

The read statement samples the corresponding import continuous flow and store the result in the specified variable. The syntax of the read statement is:

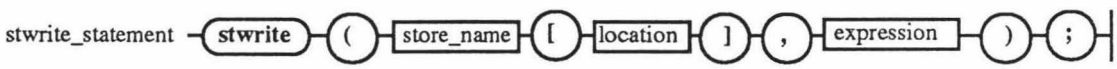


Store flows

The I/O statements associated with store flows allow a process to access the contents of a data store. Three I/O statements are associated with store flows: stwrite, stread and unlock.



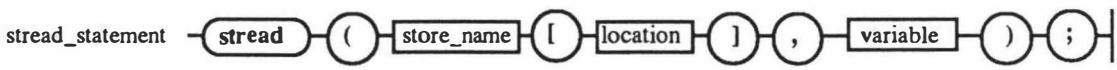
The stwrite statement allows the results of an expression to be written to a particular location of the named data store. The syntax of the stwrite statement is :



where store_name is the name of the store the expression is to be written to and location specifies the address within the store it is to be written to. If the store is locked by another process when the stwrite statement is executed then the process execution is suspended until the write can take place. An example of a store write operation in the process update_threshold_images to update the data store threshold is:

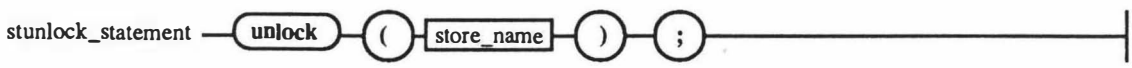
```
stwrite(threshold[new_value.location], new_value.data);
```

The stread statement allows the contents of a particular location of the named store to be read into the named variable.



If the process accesses the store in read/write mode (i.e. the store flow linking the process and the store has an arrow at each end) then a process performing a read operation will lock the store for its sole use. If the store is locked by another process when the stread statement is executed then the process execution is suspended until the read can take place.

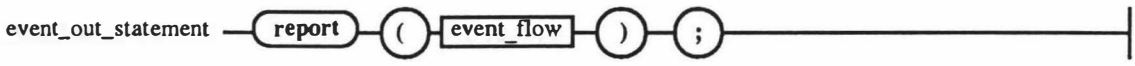
When a store is locked by a process it remains so until the process finishes its current execution or it is explicitly unlocked using an unlock statement.



This prevents a process from having to complete its execution before unlocking a store if the designer knows that no potential consistency problem exists.

Event Flows

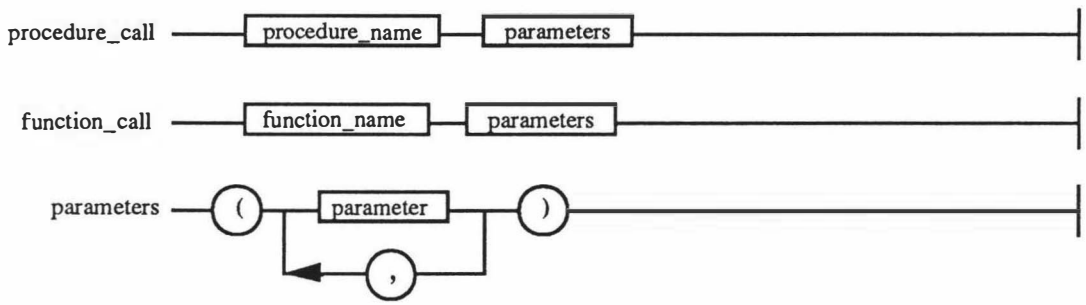
Event flows allow a single data condition to be reported to other parts of a system and are covered in more detail in the section on the control model. Processes can report the occurrence of a data condition using the report statement :



Procedure and Function Calls

To help with functional partitioning the PICSIL notation allows procedures and functions to be defined in an data dictionary appendix (see section on appendix). A process definition may include calls to procedures or functions, which indicate a request by the process to execute the functionality defined in the procedure or function. A function call may appear wherever *expression* occurs in the syntax diagrams.

These procedure and function calls are checked against the procedure and function definitions to ensure the formal and actual parameters are compatible in size, type and direction. This will be covered in more detail in the section on the appendix. Procedure and function calls have the syntax:



An example procedure call to the procedure pixel_add2 is shown below. See the data dictionary appendix section for the definition of the procedure pixel_add2.

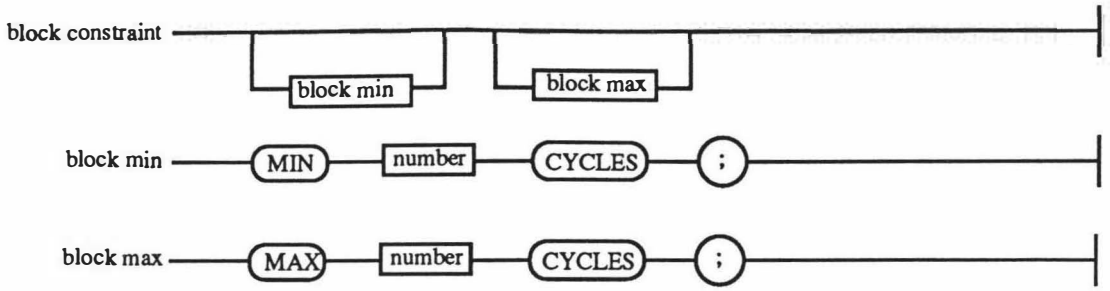
```
pixel_add2(mapped_stream_1, mapped_stream_2, resultant_pixel);
```

A function call to the function pixel_add2 is shown below.

```
resultant_pixel = pixel_add(mapped_stream_1,mapped_stream_2);
```

Temporal Constraints

There are two forms of temporal constraint, one to define *minimum* and *maximum* execution times for a block of statements, and the second which delays the execution of a single statement.

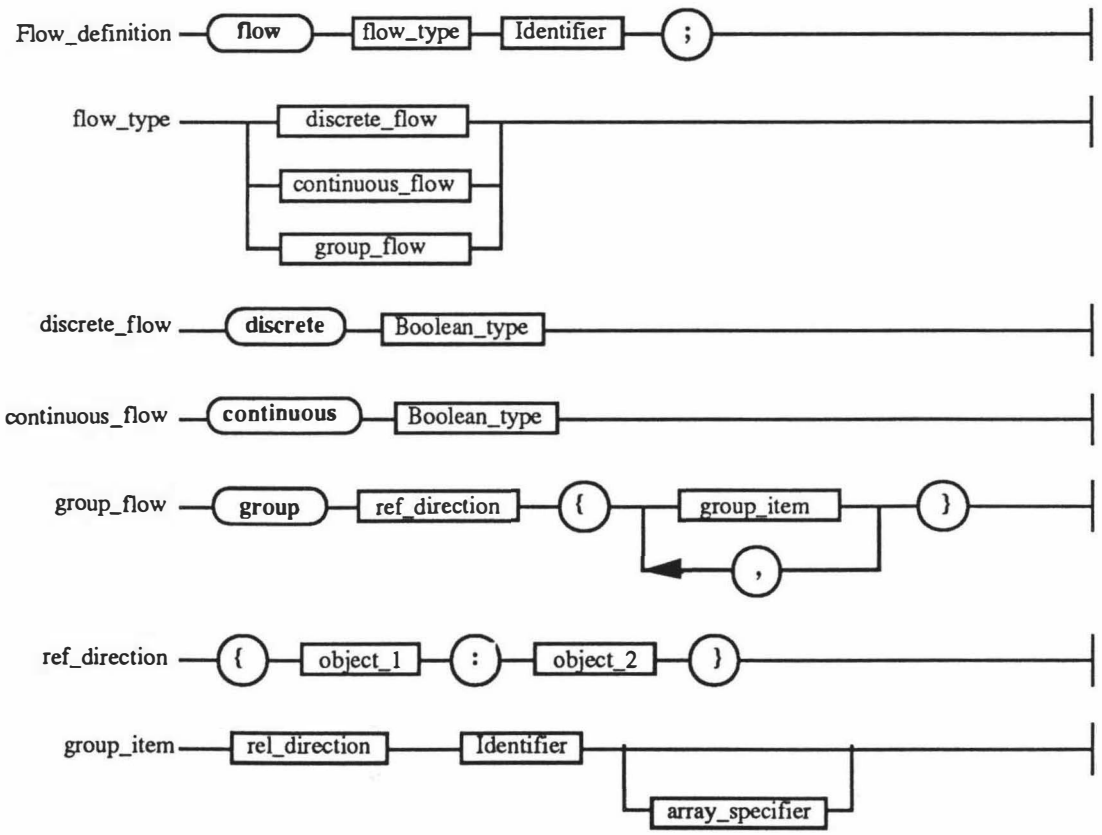


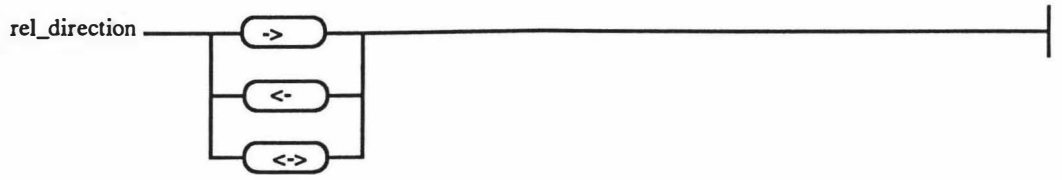
The first statement of a compound statement may be a block constraint statement which allows limits to be specified on minimum time, maximum time or both for the execution of the block. If a *block min* statement exists then the block cannot complete execution until at least *number* of cycles after it starts. The existence of a *block max* statement specifies that the block must complete within *number* of cycles from starting.

The second type of constraint allows the start of a statement to be delayed by *number* of cycles

A1.2 Data Flow Definitions

The definition of a data flow in the data dictionary specifies the exact format of the data to be communicated along the data flow. The types of flow that require entries in the data dictionary are discrete, continuous and group. As the size of a store flow can be calculated from the definition of the data store it is attached to store flows do not required entries in the data dictionary. Likewise event flows (to be discussed in Control Model section) do not require a data dictionary entry as all event flows are the same and have no data associated with them.





Both discrete and continuous data flow definitions are relatively straightforward and several examples are shown in Figures 3.8 and 3.9 (page 32).

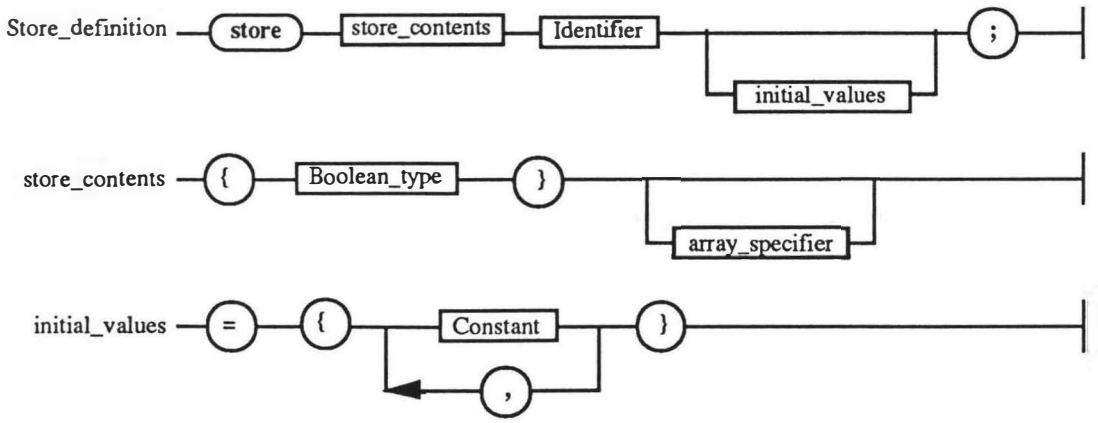
The group flow definition is made up of zero or more group items. Each *group_item* defines one of the child flows that make up the group. Only the child flow name is included and its definition is included elsewhere in the data dictionary. Each *group_item* may have also have an array specifier associated with it, indicating a number of occurrences of that child flow. The array specifier is mainly used when a group flow is imported or exported by an element.

It is possible that a group flow is never decomposed into its child flows so a means of determining the direction of the child flows is required. The *ref_direction* construct defines a reference direction from *object_1* to *object_2*. of the group flow. *Object_1* and *object_2* are the names of the objects that the group flow is connected to in the highest level diagram that it appears. The *rel_direction* construct is then used to define the relative direction of each *group_item* compared to the *ref_direction* where -> indicates the same direction as the *ref_direction*, <- indicates the opposite direction from the *ref_direction* and <-> indicates the group item is a bidirectional flow. The definition for the group flow status in the image processing example (Figure 3.4 page 28) is:

```
flow group (threshold_port:combine_streams) {
   -> status\new_info;
   <- status\outdated;
   <- status\updated;
} status;
```

In this example the child flow status\new_info flows from the external entity threshold_port to the process combine_streams and the flows status\outdated and status\updated flow from the process combine_streams to external entity threshold_port.

A1.3 Data Store Definitions



Every data store in a PICSIL design requires an entry in the data dictionary which defines the size (*Boolean_type)* and number of items (*array_specifier*) the store may contain. If no *array_specifer* is given then the store size is set to one. The following data store definition defines a store called floating_point_store which contains 256 eight bit floating point numbers addressed from 0 to 255:

```
store {
   struct{
      boolean sign;
      boolean{0:3} mantissa;
      boolean{0:2}exponent;
   } [256]
} floating_point_store;
```

A store may also have its contents initialised to predefined values on power-up using the intial_value construct. If the *intial_value* is not specified then the store values cannot be assumed on system is power-up. A definition for a four word data store called routing table where the contents have been initialised is:

```
store { boolean {0:7} [4]} routing_table = {1, 2, 3,4};
```

A1.4 Data Dictionary Appendix

The specification of a design using PICSIL may contain items that have global significance. The image processing example defined in this chapter has several such items. For example the definition of a pixel is required through out the example. Also several processes may require to implement an algorithm to add two pixel values together.

The PICSIL notation has a data dictionary appendix to allow objects with global significance to be defined once. Each time the item is required in the body of the design a reference to its definition in the data dictionary appendix can be made saving multiple definitions. The data dictionary appendix allows constants, structures, procedures and functions to be defined.

Constants

The PICSIL notation allows the use of the **#define** construct of the C programming language to allow a symbolic name to be assigned to be a particular string of characters. When a design is compiled the synthesis system will then replace any occurrences of the symbolic name with its string of characters. For example if a word size in a design is 16 bits the a constant called word_size can be defined using:

```
#define WORD_SIZE 16
```

User Defined Types

The PICSIL data dictionary allows for user defined types to be defined and is based on the C programming language.

For example a pixel type could be declared:

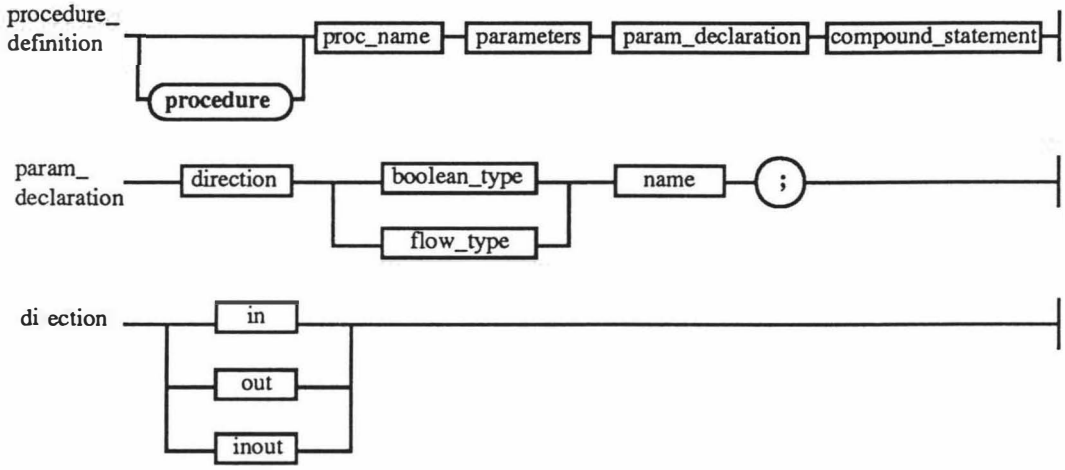
```
typedef struct {
   boolean <0:7> red;
   boolean <0:7> green;
   boolean<0:7>  blue;
} pixel_defintion;
```

Once defined, the new type can be used in the place of *boolean_type* in the declaration of other objects. For example to define the continuous flow stream_image_1 the following data dictionary entry could be used:

```
flow continuous pixel_defintion stream_image_1;
```

Procedures

A procedure is an encapsulation of operations, and may consist of calls to other procedures or functions. A procedure executes whenever it is called by another process, procedure, or function, whereupon the control flow is temporarily transferred to the called process, procedure, or function. Upon completion, the control flow returns to the calling process. No recursion is permitted in the language. The formal syntax of the procedure definition is:



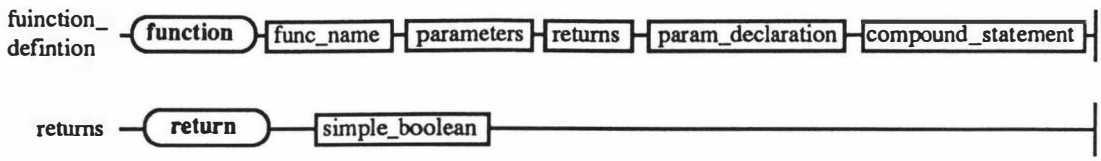
The parameters is a list of identifiers separated by commas, representing the formal parameters of the procedure or function. The direction, size and type of the formal parameters are declared in *parameter_declaration*. An example procedure which takes as input pixels pixel_1 and pixel_2, and adds then together returning the result in the parameter resultant_picsil is shown below.

```
procedure  pixel_add2(pixel_1,pixel_2,resultant_picsil)
    in discrete pixel_type pixel_1, pixel_2;
    out pixel_type resultant_picsil;
{
    resultant_pixel.red = pixel_1.red + pixel_2.red;
    resulant_pixel.green = pixel_1.green + pixel_2.green;
    resulant_pixel.blue = pixel_1.blue + pixel_2.blue;
}
```

When a procedure or function is called, the actual parameters must match the formal parameters in direction type and size. For example in the procedure pixel_add2 the parameters pixel_1 and pixel_2 are declared as import discrete flows of type pixel_type. This means that the names of the first two actual parameters in any calls must be import discrete flows of type pixel_type.

Functions

A function is semantically equivalent to a procedure, with the difference that a function returns a scalar or vector of Boolean values to the calling model. The formal syntax of the function definition is:



where **return** *simple_boolean* indicates the type of the value returned by the function. To return values to the calling model, explicit assignments are made to the keyword **return_value** in the body of the function. The last value assigned to **return_value** is the value that is returned. The following example is a function which takes two input pixels pixel_1 and pixel_2, and adds them together, returning the result as the value of the function.

```
function pixel_add(pixel_1,pixel_2) return picsil_type
    in discrete picsil_type pixel_1, pixel_2;
{
    pixel_type resultant_pixel;
    resultant_pixel.red = pixel_1.red + pixel_2.red;

    resultant_pixel.green = pixel_1.green + pixel_2.green;
    resultant_pixel.blue = pixel_1.blue + pixel_2.blue;
    return_value = resultant_pixel;
}
```