DESIGN OF DIGITAL CIRCUITS USING PROTOTYPICAL DESCRIPTIONS

by

William Stephen Adolph

B.A., Simon Fraser University, 1982

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in the School

of

Computing Science

© William Stephen Adolph 1987

SIMON FRASER UNIVERSITY

May 1987

All rights reserved. This work may not be
reproduced in whole or in part, by photocopy
or other means, without permission of the author.

APPROVAL

Name: W. Stephen Adolph

Degree: Master of Science

Title of THESIS:

Design of Digital Circuits Using Prototypical Descriptions

Examining Committee:

Chairman: Dr. Binay Bhattacharya

Prof. H.K. Reghbati, Senior Supervisor

Dr. Rick Hobson

Dr. Nick Cercone

Dr. James Delgrande, External Examiner

Date Approved: August, 26, 1986

PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis, project or extended essay (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted by me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis/Project/Extended Essay

Design of Digital Circuits Using Prototypical Descriptions

Author:

(signature)

Stephen Adolph

(name)

July 27th/87

(date)

ABSTRACT

Engineers rarely create new circuit designs from scratch. New designs are often composed of old designs whose specifications are similar to those of the proposed design. A designer may begin by comparing the specifications of a known design to those of the proposed design. If the specifications for both circuits are similar, then the existing design can be used as a prototype for the new circuit. To automate this process, designs must be described such that the details that implement a specific set of specifications in a particular design can be easily identified and modified.

This thesis develops a model of a circuit design system which designs new circuits by using an existing circuit as a prototype. The architecture of the prototype is represented by a frame. The frame is used like a language grammar production rule to expand a prototype circuit into its constituent components.

The knowledge required to use a prototype as a design guide is represented using IF-THEN rules. These rules are used to identify the differences between the behavioural specifications of the prototype and the behavioural specifications of the proposed circuit and to modify the prototype's architecture such that it can implement the proposed behaviour. A dependency network is used to relate the behaviour of the prototype to its architecture.

ACKNOWLEDGEMENTS

The author is grateful for the patience and supervision of Prof. Hassan Reghbati. Nobody could have provided more assistance and inspiration in this research effort.

The author would like to express his thanks to Dr. Cercone and extend a special thanks to Dr. Hobson for his patience and help in numerous problems.

Thanks to Amar Shan for helping me develop some of the earlier concepts this research was based on. I would like to thank Steve Lang for leading me by the hand through behavioural models. I would like to thank my employer, Microtel Pacific Research Ltd. for their financial and moral support. I would also like to thank the members of the coffee break club, Willie, Wilf, Mo, Pat, Shiv, Don and Pierre who did their best to keep up the lighter side of life.

Lastly, to my wife Fariba who put up with this for two years, its finally over love.

This research was supported by the Canadian Natural Sciences and Engineering Research Council under grant No. A0743.

DEDICATION

In memory of my grandparents,

Beatrice and Eugene Zonta

who made it all possible

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The VLSI revolution has dramatically increased the complexity of digital circuit design. Technological advancement in the fabrication of VLSI systems has superseded the capabilities of the design technology. Even with today's advanced CAD systems, the lead time required to develop a new circuit is considerable. Some circuits can require over two years to bring to the market place. This occurs in an industry where a circuit may have less than an eighteen month product window in the market place. We would therefore like to create a set of design tool which would reduce the lead time required to develop a new circuit.

1.1 The Conventional Approach to Dealing with Design Complexity

There has been significant work done in both industry and academia to provide the designer with better tools for coping with the complexity VLSI technology has imposed on digital circuit design [Gam85].

An example of these tools are the Mentor Graphics Corporation's IDEA Electronic Design System CAE tools [IDE84]. The objective of these tools is to shorten the time required to produce electronic components. Mentor Graphics breaks the design process down into four steps:

1. designing the component;
2. analyzing the component;
3. laying out of the component; and
4. documenting the component.

The focus of this thesis will be only on the first step, however, the concepts proposed in this thesis could also be applied to step 3.

The Mentor Graphic's design tools are called SYMED and NETED. SYMED is a component symbol editor. Once a symbol is created to represent a component, the user can then specify the behaviour of a component by either using the network editor NETED to create an underlying schematic, or by supplying a program that describes the behaviour of the component. A program that is used to describe the behaviour of a circuit is called a behavioural model.

If the user decides to construct a behavioural model of their component, then the user must specify an architecture that will implement the behaviour. This requires the user to relate the circuit's behaviour to the circuit's architecture, and for a VLSI system this may prove to be a rather onerous task.

Silicon compilers have been proposed and developed as solutions to the complexity associated with circuit design. Examples of silicon compilers are Macpitts [Sou83], Bristle Blocks [Joh79], and Capri [Anc83]. Silicon compilers permit the user to describe a circuit in behavioural terms with description languages similar to high level programming languages. The user is thus relieved of the tedious design details of developing a design and ultimately the layout for a circuit. However, the design generated by these systems is usually inferior to the design produced by an expert designer. For example, a layout produced by Silicon Compilers Inc. of Los Gatos, California, for the Micro Vax-1 data path chip required fifteen percent more area for individual transistors and wiring than a regular hand crafted design [Wal84] In addition many compilers are optimized for a particular class of circuits. For example a compiler which is reasonably good at producing a layout for a microprocessor may be quite

inferior for producing a memory array. This begs the question, why is a design produced by an expert designer superior to that which can be produced by automated synthesis tools?

1.2 Expertise and Design

What distinguishes the expert designer from automatic synthesis tools, such as silicon compilers, is that expert designers possess large bodies of knowledge which guide their design decisions in producing an optimal design. Development of the discipline of artificial intelligence makes it possible to consider constructing systems which can represent and exploit such design knowledge.

To develop a knowledge based CAD system we need to acquire and codify the knowledge of circuit design an expert designer possesses. We can start by asking the question "How do engineers design circuits?"
A general answer to this question is provided by Polya, in his book "How to Solve It" [Pol57]

> Have you seen it before? Or have you seen the same problem in a slightly different form? Do you know of a related problem?

In his doctoral thesis Andrew Ressller [Res84] applied this principle to circuit design by observing that

> "...circuit designers seldom create really new topologies or use old ones in a novel way. Most designs are known combinations of common configurations tailored for the particular problem at hand".

This would seem to be intuitively obvious; no one can afford to waste time inventing new circuits when an adequate design may be readily available. Often, if the existing circuit is inadequate for the task at hand, then it may be easier to modify

that circuit rather than to design a new circuit from scratch.

To support this argument Ressler developed CIROP, a system which models the behaviour of a designer engaged in the ordinary design of operational amplifiers. The objective of this thesis is to extend CIROP's model of circuit design to digital circuits.

1.3 Thesis Contributions and Roadmap

Circuit design is the process of converting a behavioural description of a circuit into an architecture that will implement the behaviour. This thesis develops a model of a CAD system which can design circuits by using an existing circuit as a prototype. The prototype for a circuit consists of a representation of its behaviour and a representation of the architecture that implements the behaviour. A designer would design a new circuit by describing its behavioural characteristics to the system. The system would then select a prototype circuit whose behavioural characteristics are similar to those of the proposed circuit. The prototype's architecture is modified so that it will correctly implement the new behaviour. Prototypes may represent abstract models of circuits where their subcomponents are themselves represented only as a behavioural description. For this reason prototypes are implemented using frames where the slots represent the individual subcomponents of the circuit [Ado86].

A set of rules are created which define the concept of circuit similarity. These rules are used to transform a prototype circuit's architecture into an architecture which can implement the new behavioural specifications. Together, the prototype frames and the similarity rules make up the knowledge base of the CAD system.

Chapter two is a survey of the related work in the application of artificial intelligence techniques to design automation which influenced this research. Chapter three is an example of how the Mentor Graphics tools may be used to construct a simple multiplier. This example is then expanded to show how frames can be used to support the hierarchical design of digital circuits. The rules of circuit similarity used for refining a prototype design are defined in this chapter. Chapter four is a detailed description of the model's architecture. Chapter five demonstrates the capabilities and limitations of the model as presented in chapter four by describing the model's design of a multiplier. Chapter six concludes by suggesting how this model can be improved and extended.

A preliminary account of the research efforts leading to the successful completion of this thesis was presented at the 23rd ACM-IEEE Design Automation Conference [Ado86].

CHAPTER 2

RELATED WORK

There have been several systems written which tackle the thorny issues associated with the design of artifacts. Of particular note, and influence on this work were CIROP, MOLGEN, Redesign, and the VLSI Design Automation Assistant.

2.1 CIROP

CIROP [Res84] served as the fundamental model for this research. CIROP is a system which can design operational amplifiers from a set of user supplied specifications. Input for CIROP consists of the specifications for the op–amp required by the user (eg. gain, frequency, slew rate). Output would be either a completed op–amp or an indication that the specifications as given cannot be simultaneously satisfied by the circuits CIROP knows how to design.

Circuit components are modeled as a set of abstract hierarchical objects which can be successively refined in order to meet user supplied specifications. Rules for expanding objects into their constituent components are modeled as grammar rules similar to those used by parsers for language compilers. For example, an operational amplifier can be regarded as an abstract object which can be expanded into three subcomponents: input stage, gain stage, and output stage. Each of these subcomponents could then be subsequently expanded into their constituent components with respect to the user supplied constraints. An example of this expansion is shown in figure 2.1.

Each of these components and subcomponents are represented by a grammar rule which states how to build that object. The circuit grammar rules have two main parts,

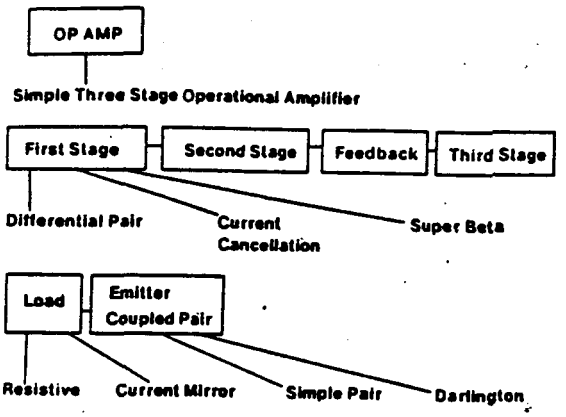


Figure 2.1 Expansion of an Operational Amplifier

the *pattern* and the *body*. The pattern describes the main characteristics of the abstract object built by the rule and is used to determine if the rule is applicable when building an object. The pattern includes three components:

1. the *type property* which specifies the generic circuit type the grammar rule constructs.
2. the *has property* which lists the behavioural properties the circuit can have.
3. and the *specifications tradeoffs* which list the design tradeoffs for that configuration.

The body of a grammar rule is a list of assertions describing how the object is to be built and analyzed. There are three types of assertions in a grammar rule:

1. The *new_part* assertion which describes the behaviour of the subcomponents.
2. The *connection* assertion which creates the circuit's architecture by connecting the internal components to each other and to the abstract object.
3. The *constraint equations* which creates constraints between the components

and the abstract object.

An example of a grammar rule is shown in figure 2.2. This rule shows how to construct a Darlington Pair transistor. This rule defines an abstract object because the NEW-PART assertions in the body of this rule declares that this component is built from two other subcomponents, namely transistors q1 and q2. The connection assertions shows how these subcomponents are to be connected together. The constraint equations are used during the analysis of the design to determine if the candidate grammar rule can be used to implement an op-amp that will meet the user's specifications.

```
(to-make-a double-darlington-transistor-bjt
     ;;
     ;; Pattern to match
     ;;
     (where(type virtual-bjt-transistor)
          (has sign sign)))
     ;;
     ;; New Parts are
     ;;
     (NEW-PART q1 ((type bjt)(has(sign sign))
     (NEW-PART q2 ((type bjt)(has(sign sign))
     ;;
     ;; Connections are
     ;;
     (connect (base) (base q1)
     (connect (collector) (collector q1) (collector q2))
     (connect (emitter) (emitter q2))
     (connect (emitter q1) (base q2))
     ;;
     ;; Constraint Equations
     ;;
     (= (rpi) (* 2 (beta q1) (rpi q2)))
     (= (ro) (ro q2))
     (= (beta)(* (beta q1)(beta q2)))
     (= (gm) (* (/ 1 2) (gm q2))))
```

Figure 2.2 Grammar Rule for Building a Darlington Pair

The rule shown in figure 2.3 defines a simple NPN type transistor that could be used to build the Darlington Pair transistor. This is a primitive object because the body does not have any NEW-PART assertions and therefore it cannot be divided further into subcomponents.

Search is the problem solving technique used to refine abstract objects. Once an abstract object is proposed, each of its NEW-PART assertions will be expanded into their constituent components by searching for a grammar rule which will satisfy the requirements associated with the NEW-PART assertion.

The search process is a negotiation between the requirements expressed by the NEW-PART's pattern and the capabilities advertised in a grammar rule's pattern. Only

```
(to-make-an bjt simple-npn-bjt
;; Pattern to match
(where(type bjt bjt)
      (has bjt (sign sign))
      (terminal-device bjt)
      (device-parameter (beta bjt))
      (device-parameter (gm bjt))
      (has bjt (priority 1))
      (three-terminal-device(base bjt)
            (base bjt)
            (emitter bjt)
            (collector bjt)))
      (equation-with-variable-priority
      (= (current (collector bjt))
            (* (beta bjt) (current (base bjt))))
      (= (beta bjt) npn-beta)
      (= (gm bjt) (* q/kt (current (collector bjt))))
      (= (rpi bjt)
            (// (beta bjt) (* q/kt (current (collector bjt)))))
      (equation-with-variable-priority
      (= (ro bjt) (// 200 (current (collector bjt))))
      (= 0 (+ current (collector bjt)) (current (base bjt))
            (current (emitter bjt)))))
```

Figure 2.3 Grammar Rule for an NPN BJT

grammar rules of the right **type** are even considered. Next any grammar rule which does not have the required specialities, the **has** properties, required for the particular part is eliminated. The remaining candidates are then sorted from simplest to most complex. Assuming the simplest design is the best, the simplest candidate which satisfies the requirements is chosen.

2.2 MOLGEN

MOLGEN is a knowledge based system which assists molecular geneticists in planning experiments. There are actually two MOLGENs, one developed by Mark Stefik [Ste81a] [Ste81b] which employs a layered planning mechanism and one developed by Friedland [Fri79] which designs experiments by refining *skeletal plans*. The focus here will be on Friedland's planner.

An observation made by Friedland during the development of MOLGEN was that scientists rarely invent the plan for an experiment from scratch. Usually they begin with an abstract, or *skeletal* plan that contains the basic steps. Then they instantiate each of the plan steps by a method that will work within the environment of the particular problem.

MOLGEN's knowledge base consists of a selection of skeletal plans and the objective and procedural knowledge necessary to instantiate the plans completely. The two major steps in plan refinement are *plan selection* and *plan instantiation*.

The problem of selecting a suitable plan is finding a plan which provides a satisfactory solution and will require the least refinement. Finding a skeletal plan can be reduced to a simple look up when exactly the same problem has been solved

before, but becomes more difficult when only related problems have been solved. The task then becomes a decision as whether to choose a detailed plan for a related problem or to choose a more general plan for the class of problems.

A search starts by looking for a skeletal plan which exactly matches the experimental design goal. If several matches are found,then all are tried. If none match, then a more general goal is chosen and the search process is repeated.

Refinement of the skeletal plan is the process of selecting an appropriate ground-level instantiation for each one of the steps in the abstract plan. This is a recursive application of the search process to the individual steps of the abstract plan.

The strategy is to avoid re-inventing plans and to use plan outlines that have worked in the past on related problems. Obviously it is important to know what constitutes a related problem.

2.3 Redesign

Redesign [Stein84] [Van84] was developed by L. Steinberg and T. Mitchell at Rutgers as a prototype knowledge based system in which AI techniques are used to interactively aid in the functional redesign of digital circuits. Redesign when provided with an existing circuit's functional specification and a desired change to those specifications, helps the user determine a change to the structure of the circuit which will allow it to satisfy the new specifications. Redesign was the forerunner of VEXED, the VLSI EXpert EDitor [Mit84].

The process of redesigning a circuit is a analogous to defusing a bomb, one should know how the mechanism works before tampering with it. It is important to

understand both the structure of the circuit and the reasoning that went into that structure.

The structure of the circuit is represented by a network of *modules* and *data paths*. A module represents either a single component (ie gate) or a cluster of components being viewed as a single functional block. Figure 2.4 is a representation of a code converter module. Modules are wired together by connecting their ports to the ports of other modules.

A data path represents either a single wire or a group of wires. The data flowing on a data path is represented by a *data-stream*. A data-stream gives the entire history of a signal as an infinite sequence of data elements. The behaviour of a data-stream is represented by an equation formula.

Each module description consists of a set of *operating conditions* and a set of *input/output mappings*. Operating conditions are performance specifications. Input/output mappings provide an equation for each feature of the output in terms of features of the inputs.

In addition to modelling the existing circuit's structure and behaviour, Redesign also represents the original designer's justifications for his design choices. Information of

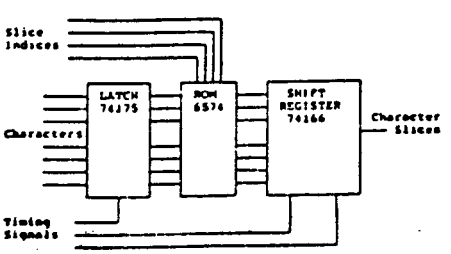


Figure 2.4 A Code Converter Module

a teleological nature is recorded in a data structure called the *design plan*.

The design plan is a data structure which shows how the circuit specification's were decomposed and implemented in the circuit. In addition, the design plan reveals the conflicts and subgoals which arose during the initial design. The design plan is a recording of the reasoning which went into the original design of the circuit. Figure 2.5 is an example of a design plan for the code converter module.

The design plan can be liken to a labelled graph where the nodes represent a high level circuit module. Each high level node can be decomposed into a design plan which contains the reasoning for that module's internal implementation. The nodes are linked by two types of edges. The first type of edge (solid lines) corresponds to some implementation choice in the design. The second type of edge (broken lines) represents conflicts arising from implementation choices. A conflict can arise in a circuit for example, when a decision is made to use a ROM with a parallel output when a serial signal is required. The conflict is resolved by creating a subgoal in the design plan to accommodate a parallel to serial conversion.

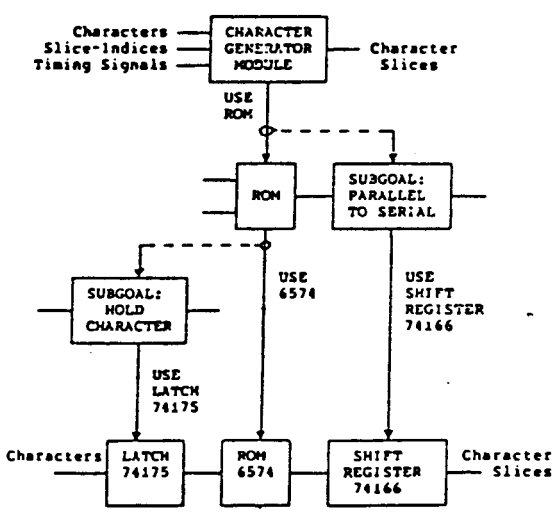


Figure 2.5 Design Plan Example

By examining the design plan of a circuit, Redesign can reason about the purposes of various circuit modules and the way the circuit specifications are implemented.

The functional redesign of a circuit starts by limiting the size of the solution space. Redesign first focuses attention onto the appropriate sections of the circuit. Redesign replays the Design Plan and compares the purpose of each new module with the corresponding module in the original Design Plan. If the purpose is unchanged then the module will be re–used without change in the new design. If the new module has a different purpose then Redesign stops expanding this portion of the circuit and marks it for further redesign.

Once the appropriate section of the circuit requiring redesign has been identified, then redesign options are generated for those sections. Options considered are:

1. breaking the data path at the focal point and inserting a module which will transform values on the data path to values which will satisfy the new specifications.
2. alter the module immediately up stream and make it provide the required signal.

Redesign employs a set of heuristics for ranking redesign options based on

1. the estimated difficulty of implementing the redesign option.
2. the likely impact of the redesign option on global circuit parameters.
3. the likelihood and severity of the side effects that might be associated with the redesign.

The final step of the redesign process implements the selected redesign option and evaluates its side effects.

2.4 The VLSI Design Automation Assistant

The Design Automation Assistant (or DAA) performs data path synthesis for the CMU/DA [Thom83] It was written as a knowledge based alternative to the algorithmically based program EMUCS which performed the same task.

The register transfer level behaviour of a circuit is expressed in ISPS which is then compiled into a data flow like representation called the Value Trace [Snow78]. The DAA will take the Value Trace description of the circuit and produce a functional block description of that circuit.

The DAA is a rule based production system consisting of a working memory, a rule memory, and a rule interpreter. The working memory describes the current design state of the circuit. The rule memory consists of a collection of conditional statements which operate on the elements stored in the rule memory. Figure 2.6 is an example of a rule.

Rule selection is data driven. The rule interpreter looks through the rule memory for a rule whose conditions are all true. If more than one rule is applicable, then the

```
IF the most current active context is to create a link
     AND the link should go from a source to a destination port
     AND the module of the source port is not a mux or a bus
     AND there is a line from another module to the same destination
     AND this other module is not on a mux or bus
THEN create a mux module
     AND connect the mux to the destination
     AND connect the source port and destination to the mux.
```

Figure 2.6 A Sample DAA Rule

rule dealing with the working memory element most recently modified is selected first. If there are still multiple rules applicable then the most specific rule is selected.

DAA rules are grouped into a set of temporally ordered subtasks. Synthesis begins by assigning hardware storage modules such as registers and memories to all Value Trace values declared in the ISPS description. A Value Trace body is then selected and control steps are allocated. Next the synthesis step maps all Value Trace operator outputs not assigned to storage modules. Finally it maps each Value Trace operator to processor modules, connecting links and supplying multiplexers where necessary.

The DAA has been applied to the design of circuits varying from a MCS6502 to an IBM/370. Figure 2.7 is an example of a circuit the DAA designed to implement the architecture of a MCS6502.

2.5 Using Prototypical Knowledge for Design

The DAA, unlike CIROP and MOLGEN, does not take advantage of the very large pool of knowledge represented by existing designs. Systems such as MOLGEN and CIROP demonstrate that prototypes represented as skeletal plans or as abstract designs can be used to develop new plans or new circuits. Chapter three will show how prototypical descriptions of existing digital circuits can be used to design new digital circuits.

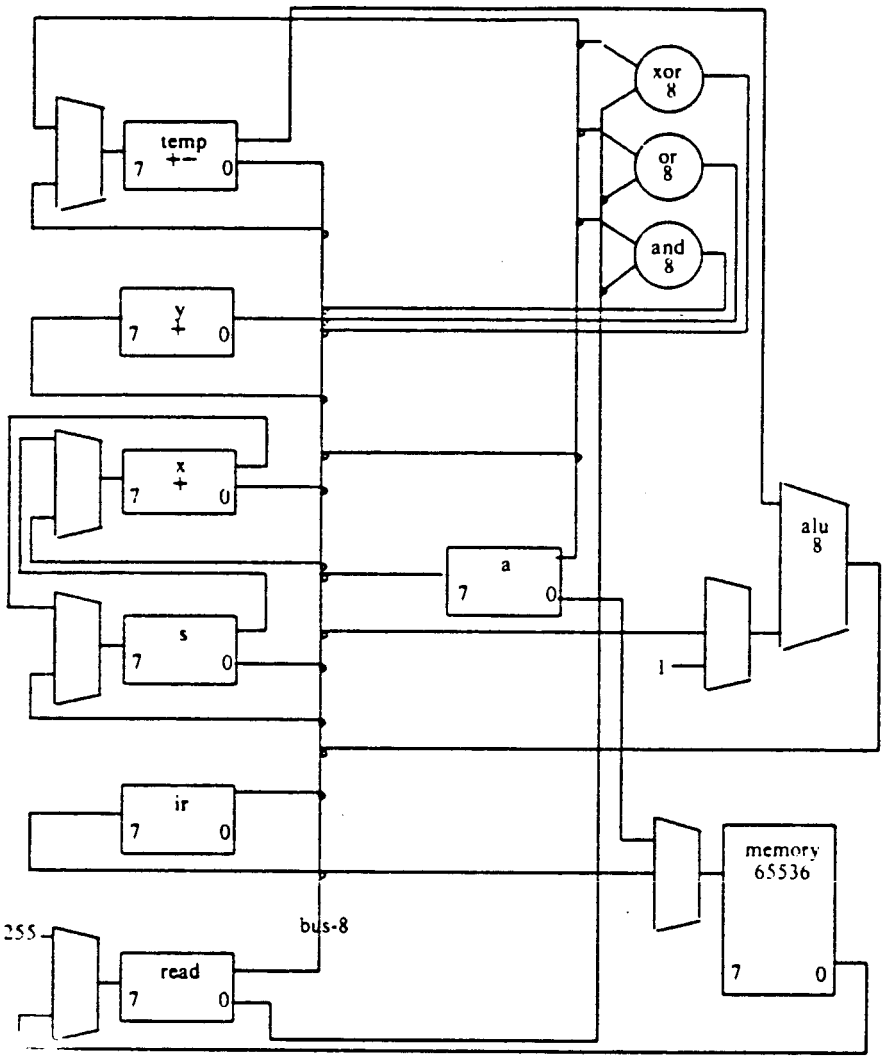


Figure 2.7 MCS6502 Architecture as Designed by the DAA

CHAPTER 3

CIRCUIT DESIGN USING PROTOTYPICAL DESCRIPTIONS

We suggested in chapter one that engineers often design circuits by recognizing that a circuit they have designed previously may satisfy the specifications for a new circuit. This would require the designer be able to recognize that the behaviour of an existing circuit is similar to the proposed behaviour of the new circuit. Specifically, the designer must be able to recognize which of the architectural attributes of the existing circuit support behaviour that is identical to that of the new circuit, and which support behaviour that is unique to the existing circuit. The designer must then be able to replace those architectural features with new features that will support the behaviour that is unique to the new circuit.

We call this process *design using prototypical descriptions.* The prototypical description for a new circuit can be any circuit which has a behaviour that is similar to that of the proposed circuit.

Automation of this process requires a representation of a circuit's behaviour and of its architecture. There must exist a means for relating the circuit's behaviour to its architecture. There also must be a means for recognizing which circuits have similar specifications. Finally, there must be a means for resolving the differences between the behaviour of the existing circuit and the new circuit.

To develop a representation for digital circuits, we examine the structures a designer might develop to represent a design using the Mentor Graphics IDEA system.

3.1 Representing Prototype Circuits

The behavioural specification for a circuit is often expressed using some form of algorithmic description. Figure 3.1 is a PASCAL translation of a program that appears in John Haye's text on digital circuit design [Hay84] which describes the behaviour of a multiplier.

A circuit designer will use his knowledge of circuit design to transform this specification into a circuit. For example, a circuit designer "knows" that the storage and shifting requirements of the accumulator can be satisfied using a shift register. The circuit designer "knows" that an adder will be required and that there must be a data path from the multiplicand and multiplier registers to the adder. The circuit designer also "knows" that he will need a multiplexer to gate the sixteen bit output of the accumulator to the eight bit output port. This is the type of knowledge the DAA uses to design circuits.

The circuit designer would start the implementation of the multiplier's behaviour by using SYMED to create a symbol for each subcomponent of the multiplier. The general procedure followed in SYMED is to:

1. draw the body of the symbol.
2. place pins on the symbol body.
3. add property information about the symbol body.
4. add property information about the pins,
5. check for correctness.

Figure 3.2 is the schematic symbol drawn for the multiplier's accumulator using SYMED.

```
PROCEDURE mult$$allocate;

  VAR
        creg                        : REGISTER;
        areg                        : REGISTER;
        qreg                        : REGISTER;
        mreg                        : REGISTER;

        data_in                     : PORT;
        data_out                    : PORT;
        start                       : PORT;
        stop                        : PORT;

PROCEDURE mult_function;

  VAR
    j                               : integer16;

  BEGIN
    IF ( start = sim_$one) THEN
      BEGIN
        bus_in (data_in, qreg);
        init_reg  (sp.areg, sim_$zero);
        init_reg  (sp.creg, sim_$zero);

        bus_in (data_in, mreg);

        WHILE (decode_reg (creg) < 8) DO
           BEGIN
             IF (qreg.reg_val[0] = sim_$one) THEN
                    add_reg ( sp.areg, 8, 15,
                              sp.mreg, 0, 7,
                              sp.areg, 8, 15,
                              sp.carry_out ) ;

             shift_reg ( areg, 0, 15, RIGHT, carry_out);
             shift_reg ( qreg, 0, 7,  RIGHT, sim_$zero);
             incr_reg ( creg);
           END;

       bus_out (data_out, areg, 0, 7);
       signal_out (sp.stop, sim_$one);

       bus_out (data_out, areg, 8, 15);
    END;
  END;
```

Figure 3.1 PASCAL Listing of Multiplier

This schematic contains very little information about the accumulator other than its pin outs. Nothing is revealed about the behaviour of the subcomponent or about the signals carried by its pins. By attaching properties to the symbol the designer can describe the behaviour of the symbol and describe the signals carried by the pins.

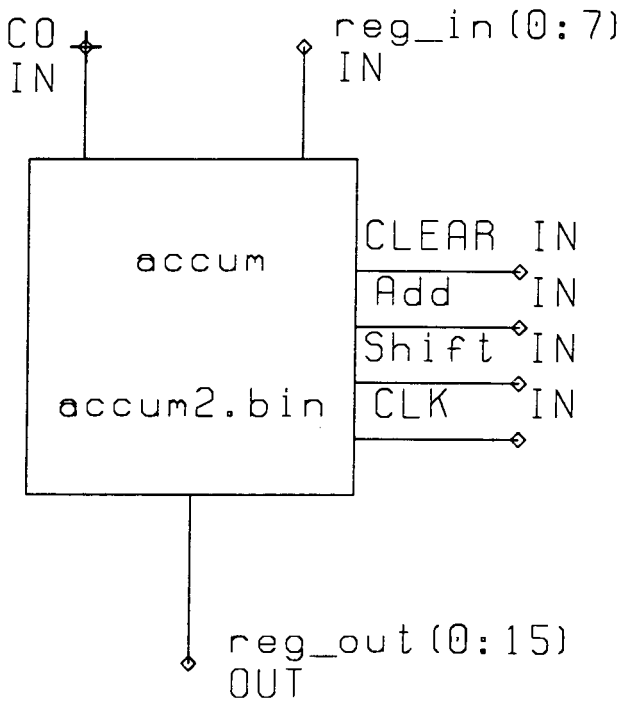


Figure 3.2 Schematic Drawn with SYMED

The behaviour of a subcomponent is described by associating a MODELCODE property with it. The MODELCODE property is a PASCAL program which describes the subcomponent's behaviour. Signals are described by attaching the PINTYPE property to the pins.

After the multiplier's subcomponents have been defined, the designer would use NETED to specify the data and control path between the individual subcomponents. Each of the subcomponents created using SYMED are placed on the NETED sheet and connected. The final result is the schematic shown in figure 3.3. The designer would then use the QUICKSIM circuit simulator to verify the correctness of the proposed design.

By using the Mentor tools, the designer has expanded the original PASCAL description of the multiplier's behaviour into an architecture that will implement the behaviour. The behaviour of each subcomponent is now described by the PASCAL

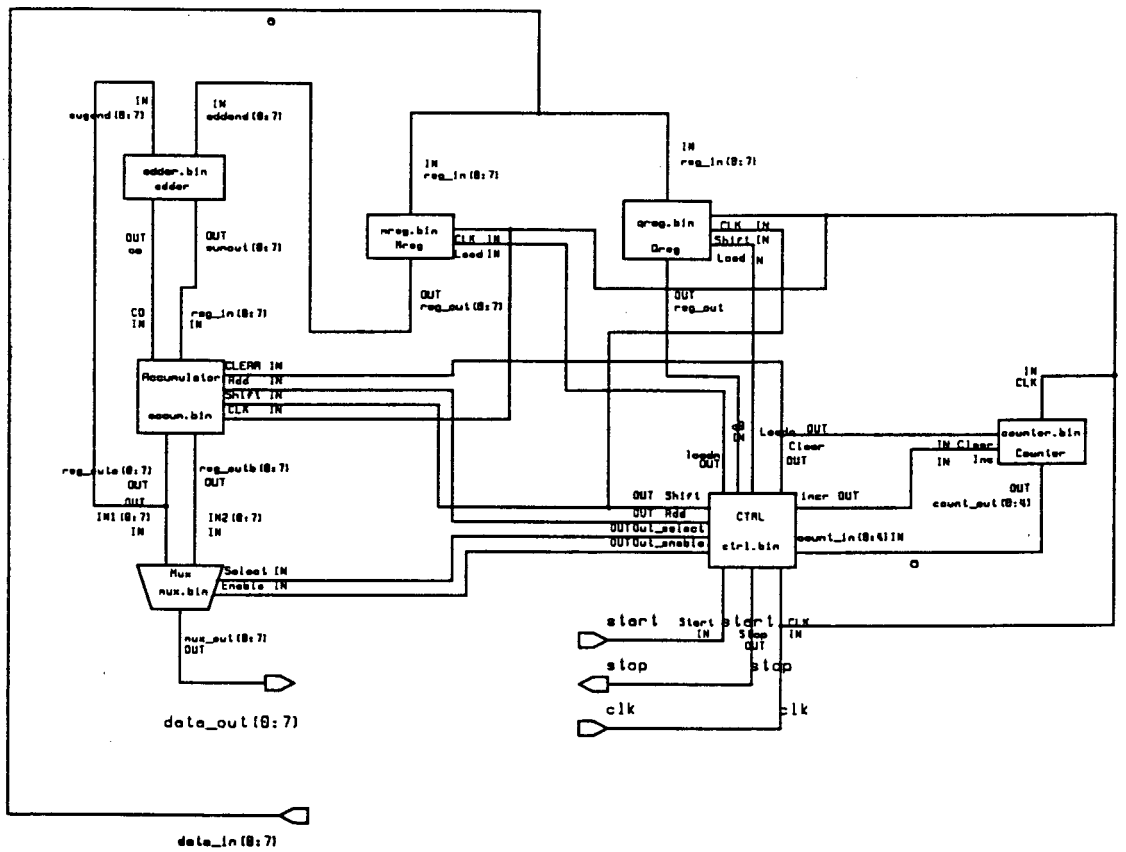


Figure 3.3 Multiplier Architecture

program that is associated with the subcomponent's MODELCODE property. The multiplier's architecture is represented as a network of communicating black boxes with each black box having its behaviour described by a PASCAL program.

The designer now has two options, he can either assign primitive hardware to the subcomponents or he can expand the PASCAL descriptions into their respective architectures until primitive hardware can be assigned to their subcomponents. Design is therefore a hierarchical process where abstract components are successively refined into their constituent components. [1]

[1] It should be noted that design is also an iterative process where lower level design

The schematic produced by the designer in this example, represents two types of information about the multiplier circuit:

1. *Circuit Architecture.* If we ignore the MODELCODE properties associated with the subcomponents in the schematic then we obtain a list of the different functional blocks required to implement the circuit's behaviour and their interconnection.
2. *Component function.* The MODELCODE property associated with each functional block reveals the intended behaviour of the module.

The architectural information contained within the schematic can be represented using a frame.

3.1.1 Frame Based Representation

Frames were originally proposed by Minsky [Min75] as a basis for understanding complex behavior such as visual perception and natural language understanding, but have recently found their way to representing design knowledge [Abad85] [Sait81] [Giam85]. Frames provide a structure within which new data can be interpreted in terms of concepts acquired through previous experience [Barr81]. A frame could be loosely modelled as a template used to describe an object. The template contains a number of *slots* which must be filled in for the template to be considered complete. Often the slots have associated with them information which will constrain the set of values which can be used to fill the slot. Procedures can be attached to the slot in order to compute a value for the slot when a value is not readily available. Once the template is completed, it represents a specific instance of the typical object.

---

[1](cont'd) failures may result in redesign of higher level objects.

Frames are a good implementation for representing prototypes for circuit design because they can aggregate all the individual properties that are associated with a circuit design into one representation structure. Each subcomponent can be represented by a slot where the behaviour of the subcomponent specifies the constraints that must be satisfied by the object used to fill in the slot.

The feature that makes a frame a better representation for circuit designs than other techniques is the frame rule of inference *instantiation*. This rule when applied to a frame representing a circuit, implements circuit design by top-down refinement. Each slot in the frame represents a *black box* description of a subcomponent. When a slot is filled in, the black box description of the subcomponent is replaced by an architecture which will implement its black box behaviour. Another frame can be used as a slot filler resulting in a recursive application of the design process. Instantiation permits a frame to be used as a production rule for constructing a digital circuit in a manner similar to CIROP's [Res84] circuit grammar rules. A frame can therefore be used to support hierarchical design.

Minsky stated in his original paper that his ideas were not refined enough to serve as the basis for a working system and therefore a number of different frame based languages have been developed over the past few years. Typical of the early frame languages was KRL developed by Bobrow and Winograd [Bob75] which provided constructs for describing individual entities and classes of entities. Each individual or class is represented as a frame. Frames can be organized into taxonomies using links that that represent class membership. For example, one frame can be used to represent the general class automobile and another frame can be used to represent the specific instance of the class such as Ford Mustang. The Ford Mustang will inherit the properties associated with the superordinate frame "automobile". This is an example of

the IS_A relationship between frames where Ford Mustang IS_A automobile.

We have taken a more restrictive view of the links between frames by having the links between frames represent subcomponent composition. For example, the shift register of a multiplier is not a type of multiplier, rather, it is a subcomponent or IS_PART_OF the multiplier. It is this relationship between a prototype circuit and its subcomponents that we wish to express using frames.

We opted not to use an existing frame language because we did not want to obscure our concepts within the idiosyncratic syntax of any given language. Rather we chose to present our system as a logical model which could be adapted to a specific implementation.

3.1.2 A Frame Based Representation for Digital Circuit Prototypes

We used CIROP's circuit grammars as a model for creating a frame to represent digital circuits prototypes. The body of a CIROP grammar rule lists assertions describing how the circuit is to be built. Each of the NEW-PART assertions in the body of the grammar rule describes the behaviour a component must implement if it is to be used in the design of an op-amp. The behavioural requirements for the abstract components of a digital circuit are represented by a DESIGN slot. The name DESIGN slot is derived from this slot's usage. A DESIGN slot describes the behaviour a component must have if it is to be used as part of the circuit.

The CONNECTION assertions in a CIROP grammar rule specifies how the terminals of subcomponents are to be connected. This assertion specifies the architecture of the op-amp. The architecture of a digital circuit is represented by a list of PLACE and CONNECT instructions associated with the prototype frame called the

ARCHITECTURAL-PROPERTIES list. Figure 3.4 is a skeletal model of a prototype frame.

The grammar rule's pattern is used to describe the function the circuit implements. CIROP designs circuits by matching the requirements expressed in a NEW-PARTS assertion to the pattern of a grammar rule. Similarly, each frame has associated with it a *header* which describes the behaviour of the circuit it represents. Circuit behaviour is expressed using the PASCAL language. Like CIROP, digital circuit design can also be implemented as search where an attempt is made to find a prototype with a frame header that matches the behavioural requirements expressed in a DESIGN slot.

This search process requires that two algorithms be compared to each other to see if they are equivalent. This is a computationally difficult problem if we try to solve it algorithmically, especially if we are not looking for exact matches. We therefore take a knowledge based approach to this problem because we can tailor the

```
(PROTOTYPE (
(HEADER (algorithm describing circuit behaviour))
(DESIGN (
     (COMPONENT-1 (component-1 algorithm))
     (COMPONENT-2 (component-2 algorithm))
     :
     :
     (COMPONENT-n (component-n algorithm)))
(ARCHITECTURAL-PROPERITES (
     (LAYOUT-COMMAND-1)
     (LAYOUT-COMMAND-2)
     :
     :
     (LAYOUT-COMMAND-n)
```

Figure 3.4 Model of a Prototype Frame

search process to the comparison of algorithms describing digital circuit behaviour rather that trying to solve a general match problem.

3.2 Using Prototypes to Design Circuits

In CIROP, the *has* properties specified in a NEW-PART assertion must exactly match the *has* properties in a pattern of a grammar rule before that grammar rule can be used to instantiate the NEW-PART. For digital circuits this match criteria is too restrictive.

If it were to be required that the behavioural requirements expressed in the DESIGN slot must identically match the capabilities expressed in the frame header then the probability of finding a suitable component in the data base with which to fill a DESIGN slot is very low. Yet each day engineers design circuits using either standard TTL components or a library of standard cells to implement their designs. How is it that engineers can assign hardware structures to their preliminary designs?

We can assume that during the process of detailed design engineers do not look for an exact match between the behavioural requirements of their abstract subcomponents and the capabilities of the hardware they have available to them. Instead they look for hardware which closely matches the behavioural requirements of their design and then modify or re-configure the hardware such that it satisfies the design requirements. Consider a register whose behavioural requires that when it receives a load signal that it must load only its upper half word and leave the lower half word intact. There are few, if any, standard registers which can be controlled in this manner. An engineer would resolve this problem by linking two registers, each a half word in width, to construct the full word register. The upper half word will be

connected to the load line while the lower half will not be.

It is quite clear that knowledge is required to use a prototype as a design guide because even the largest library of prototype circuits will not help a designer if he does not know how to use them. The solution proposed here is to define the concept of a similar circuit and to be able to use similar circuits as a prototype to guide the development of a new circuit.

3.2.1 Defining Circuit Similarity

What characteristics of a circuit's function establish it as being different or similar to another circuit's function? Why, for example, can it be said that a left shift register is similar to a bi-directional shift register but not to a multiplier?

One criteria which can be established for determining whether two circuits are similar is if the differences between the behavioural specifications of the two circuits can be explained by using a limited set of rules. These rules would define how circuits differ but yet can still be considered similar.

For this research, four classes of rules define what differences between the behavioural specifications of two circuits are explainable and therefore when two circuits can be considered to be similar.

1. *Serial/Parallel IO Conversion Rules:* Many circuits which implement a behaviour may read their operands or write their results either serially or in parallel. An example of this may be seen in computer families, where low end processors which implement the same instruction set as their high performance siblings, but read information from their bus byte serially, while the higher end machines will have increasing degrees of parallelism.

2. *Encoded/Decode Decision Rules:* There can be a wide variation in how circuits implementing the same function receive control information from the external environment. Control information can be fully encoded, where both the low and high signal on a line initiate active responses or control information may be decoded where only one signal state on a line will initiate an active response in the circuit. This is analogous to the difference between the use of CASE statements and successive IFs in a programming language.
3. *Subset Function Rules:* Some circuits may perform only a subset of the functions that another circuit performs. For example consider a universal shift register which performs bidirectional shifting and is presettable. A shift register which can only right shift and is presettable implements a subset of the functions of the universal shift register.
4. *Pre/Post Processing Rule:* Some circuits implement the same function as other circuits however they may pre process input or post process output. An example of this may be where a circuit inverts its inputs or outputs.

If the differences between the behavioural specifications of two circuits can be explained by the repeated application of one or more rules from this set of rules, then the circuits are considered to be similar.

A circuit which implements a behaviour similar to that of a circuit proposed by the designer, can be used as a prototype to implement the proposed circuit. The prototype can then be modified to implement the behaviour of the new circuit.

To modify a circuit such that it will implement a new behaviour requires that the designer know how the existing circuit implements its behaviour. We therefore require a means for relating a circuit's behaviour to its architecture.

3.3 Relating Circuit Architecture to Circuit Behaviour

Redesign used a circuit's Design Plan to reason about the purpose of circuit's components. By examining the Design Plan of a circuit, Redesign could reason about the purpose of various circuit components and the way the circuit specifications are implemented.

We have created a comparable structure called the *Dependency Network* which shows how a circuit's architectural features implement its behaviour. Whereas the Design Plan recorded the designer's justifications for an implementation, the Dependency Network records the architectural resources, data paths, control paths, and subcomponent behaviour, required to implement the circuit's behaviour. For each step in the algorithm describing a circuit's behaviour, the dependency network will record:

1. the steps in the algorithms describing the behaviour of the subcomponents that must be carried out.
2. the control paths that are used to transport the control signals required for this step.
3. the data paths that are used to transport the operands and results of the step.

The information represented in the dependency network is used by the system to determine which architectural features of the circuit must be altered when the behavioural specifications of the prototype is modified.

3.4 Components of the Model

This chapter has developed the components of our model system for designing digital circuits using prototypical descriptions. We have used frames to represent the relationships between a circuit and its subcomponents and to support hierarchical design. We have created a set of rules to define circuit similarity. These rules permit the use of circuits implementing a behaviour similar to that of a proposed circuit to be used as prototypes for that new circuit. Finally, we created the dependency network which relates a circuit's behaviour to its architecture.

Chapter four provides a description of how the components of this model could be implemented.

CHAPTER 4

REPRESENTING AND USING PROTOTYPICAL DESCRIPTIONS

This chapter will describe the detailed architecture of the components of our model for designing digital circuits from prototypical descriptions. The first section of this chapter describes our language for specifying the behaviour of a digital circuit. The second section describes our frame implementation for representing the architecture of a circuit. The third section describes the frame header and the dependency network which relates the architectural features of the circuit to its behaviour. The fourth section describes our representation of the knowledge required to use an existing circuit as a prototype for a new circuits. Finally, the last section describes the mechanisms used to control the implementation of design process.

4.1 Representing Circuit Behaviour

The behaviour of a digital circuit can be described by an algorithm written in some procedural language. For this research, we used Mentor Graphic's PASCAL language to express the behaviour of a digital circuit. The basic PASCAL language was augmented with a subroutine library and a set of data structures to support its use as a hardware description language.

4.1.1 Instructions

The most basic unit of the language is the instruction. An instruction represents one primitive functional transform the circuit can perform on an operand in one clock cycle. There are currently thirteen instructions defined in three instruction classes:

1. **IO–Group**: These instructions read or write data into or out from the circuit.

There are six instructions in this group:

a. **SIGNAL_IN** (*signal*): This operator samples the current value on the port specified by *signal*. The sampled value is not stored and is lost after the call.

b. **SIGNAL_OUT** (*signal-port, value*): This operator drives output port *signal-port* to the value given by *value*.

c. **BUS_IN** (*input-src, input-src-width, input-dest, input-dest-width*): This operator samples the port specified by *input-src* and stores the value into *input-dest*. The *input-src-width* and *input-dest-width* specify the address of the first and last bits in the port or register affected by the operation.

d. **BUS_OUT** (*output-dest, output-dest-width, output-src, output-src-width*): This operator writes the contents of *output-src* to *output-dest*.

e. **GET_ARG** (*input-src, input-src-width, input-dest, input-dest-width*): This operator is identical to BUS_IN except that the target *input-dest* is not a true register. This input operator is used when the operand is not to be stored within the circuit.

f. **PUT_ARG** (*output-dest, output-dest-width, output-src, output-src-width*): This operator is identical to BUS_OUT except that the source *output-src* is not a true register. This output operator is used when the output source is from the result of an operator and not from a storage element within the circuit.

2. **Functional Group**: These instructions perform transformations on their operands. There are five instructions in this group:

a. **SHIFT_REG** (*shift-src, dir-spec, shift-in-src*): This instruction shifts *shift-src* one bit to the direction specified in *dir-spec* with *shift-in-src* being shifted in.

b. **ADD_REG** (*result, result-width, addend, addend-width, augend, augend-width, carry-out*): This instruction adds *addend* to *augend* to produce *augend*. All operands must be registers. The carry out is coded into the output signal *carry-out*.

c. **INIT_REG** (*register, value*): This instruction sets *register* to the predefined value given by *value*.

d. **INCR_REG** (*register*): This instruction increments *register*.

e. **MOVE** (*src-reg, src-reg-width, dest-reg, dest-reg-width*): This instruction moves the contents of the *src-reg* to the *dest-reg*. Both operands must be registers and both operands can be masked.

3. **Predicate Group**: This group of instructions is used to test an event for flow control. There are two instructions in this group:

a. **EQUAL** (*register1, value*): This instruction is used to compare a register to a value.

b. **NOT_ZERO** (*register*): This instruction returns true if the contents of *register* are zero.

4.1.2 Operands

An instruction operand can be either a port or a register. A port is the boundary through which the circuit communicates information to the external environment and has no storage associated with it. A port is defined by declaring an identifier to be of type PORT. Figure 4.1 shows the PASCAL type definition for the port record. The WIDTH of the port specifies the width of the port in bits. The IOTYPE of a port is used to specify whether the designer intended the port to be part of the circuit's data path or its control network. The IO of a port is the direction of data flow through the port. A port can be strictly input, output, or it can

```
CONST
  RIGHT                    = 0;
  LEFT                     = 1;

TYPE
  port_type                = (ctrl_port, data_port);
  port_dir                 = (input_port, output_port, io_port);
  port_name_ord            = (clk_sig, start_sig, stop_sig, data_out_bus);
  reg_name_ord             = (creg_reg, areg_reg, mreg_reg, qreg_reg);
  clock_enum               = ( clock1, clocki, clock2i, clocknl, clockn);

  port                     = RECORD
      port_name                : port_name_ord;
      iotype                   : port_type;
      iodir                    : port_dir;
      port_width               : INTEGER16;
      port_val                 : ARRAY [0..15] OF INTEGER16;
                             END;
```

Figure 4.1 PASCAL Type Record for Ports

be an input-output port. These values are set in the allocation procedure of the PASCAL program.

A register defines a unit of storage within the circuit and is defined by declaring an identifier to be of type REGISTER.

The values associated with registers and ports in a circuit describe the circuit's state. Non volatile storage is allocated within the PASCAL program for the ports and registers which describe a circuit's state. Declarations for registers which are not in the state record are for psuedo registers which do not contribute to the state of circuit and are only used for notational convenience within the PASCAL program. The PASCAL type record for REGISTER and an example state record is shown in figure 4.2. The WIDTH of the port specifies the width of the register in bits.

```
register                    = RECORD
    reg_name                    : reg_name_ord;
    reg_width                   : INTEGER16;
    reg_val                     : ARRAY [0..15] OF INTEGER16;
                                END;

mult_state_rec              = RECORD
    data_in                     : port;
    data_out                    : port;
    start                       : port;
    stop                        : port;
    clk                         : port;
    areg                        : register;
    creg                        : register;
    mreg                        : register;
    qreg                        : register;
    carry_out                   : INTEGER16;
    clock_cycle                 : clock_enum;
                                END;
```

Figure 4.2 PASCAL Type Record for Registers and Sample State Record

4.1.3 Statements

Instructions are aggregated into *statements*. A statement is a list of instructions that are executed concurrently. A statement has the format shown in figure 4.3. Statements do not correspond to PASCAL grammatical constructs. They are implemented as comments and are used to designate groupings of instructions which can be executed concurrently in one clock cycle. This structure was created because there is no suitable feature within the PASCAL syntax to represent concurrent execution of

```
{   (statement_type, statement_number }
        instruction_list
{   (statement_type, statement_number }
        instruction_list
```

Figure 4.3 Statement Format

instructions.

The *statement_type* serves as a statement separator and identifies the type of instructions following in the *instruction_list*. There are five types of statements:

1. **clock-begin:** This is always the first statement in a behavioural specification. It has a null *instruction_list* and implicitly branches to the statement physically succeeding it.
2. **clock-end:** This is always the last statement in a behavioural description. It has a null *instruction_list* and terminates a description.
3. **clock-step:** This statement is used to specify clock periods in a behavioural specification. It implicitly branches to the statement physically succeeding it.
4. **dec:** This indicates that all the instructions in the *instruction_list* are decision instructions, that is, IF, WHILE, and REPEAT. The next statement for this statement will be determined by the execution of the individual instructions.
5. **act:** This indicates that all the instructions in the *instruction_list* perform some action or transformation upon their operands.

The *statement_number* is used to uniquely identify the statement.

The *instruction_list* may be a list of zero or more instructions. Each instruction is prefixed by an instruction identifier INST*i*, where *i* is an integer used to identify the instruction. Figure 4.4 shows two examples of statements.

Statement 1 in this example is an action statement with three instructions which are executed in parallel. The instruction labelled INST1 moves the value at "data_in" to "multiplier". INST2 and INST3 set "accumulator" and "counter" to zero. The next statement to be executed is statement number 2 which is a clock_step.

```
{ ACT, 1 }
    { INST1 } bus_in (data_in, 0, 7, multiplier, 0, 7);
    { INST2 } init_reg (accumulator, sim_$zero);
    { INST3 } init_reg (counter, sim_$zero);
{ CLOCK_STEP, 2 }


{ DEC, 3 }
    { INST1 } IF (multiplier.reg_val[0]= sim$_one) THEN
{ ACT, 4 }
```

Figure 4.4 Example of Decision and Action Statements

Statement 3 in this example is a decision statement with only one instruction in its instruction list, INST1 tests bit 0 of "multiplier".

After establishing a notation for specifying and describing circuit behaviour we need to be able to describe the architectures that can implement the behaviour.

4.2 Representing Circuit Architecture

A circuit schematic represents a circuit's architecture as a collection of interconnected data processing elements and controllers. Some of these elements may be considered primitive because they can be immediately implemented using available hardware. Some elements may be abstract because they implement a complex function and will require more refinement before they can be implemented in hardware. A schematic may therefore possess a hierarchical structure because the internal structure of a component appearing in the top sheet may be described in the lower sheets.

To support the representation of both abstract and primitive components, a prototype frame has two representational structures, DESIGN slots and the ARCHITECTURAL-PROPERTIES list. Abstract circuit components are represented using DESIGN slots which provide a black box like representation of abstract circuit components. The ARCHITECTURAL-PROPERTIES list is a procedural representation of the circuit's schematic which lists all the circuit's components and interconnects.

4.2.1 Representing Black Boxes: Design Slots

Our frame model was influenced by Patrick Winston's text on LISP [Win84] where frame slots are partitioned into facets. A DESIGN slot is partitioned into four facets. The facets express the properties the subcomponent must possess if it is to support the circuit's specified behaviour.

The four facets of a DESIGN slot are:

1. PORT-SYMBOLS: The port symbols define the component's interfaces to the external world. These are symbols that can be referenced for component interconnection.
2. REG-SYMBOLS: The register symbols are names which are internal to the component. These symbols are defined such that they can be used in the algorithm to describe the behaviour of the component. If the component has storage associated with it, then it is defined as a REG-SYMBOL.
3. RT-TYPE: The RT-TYPE classifies the type of register-transfer operation the device supports. This is analogous to the TYPE assertion used in the pattern of a CIROP grammar rule. John Hayes identified four types of MSI components that are required to implement register-transfer operations [Hay84]:
   a. STORAGE: any device such as a register or latch which has a memory

property.

b. PROCESSOR: devices which performs a functional mapping between its inputs or outputs.

c. ROUTER: devices used to route data through the system.

d. CONTROLLER: devices which are used to generate the control signals necessary to ensure that the desired register transfer operation is carried out at the proper times and in the proper sequence.

Many devices could be labelled as having more than one type classification. For example a shift register could be classified as storage device or as a processing device. The RT-TYPE assigned to the component therefore indicates the designer's intended primary function for the component.

4. FUNCTIONAL-REQUIREMENTS: This is the specification of the behaviour the component must implement if it is to support the circuit's behaviour.

4.2.2 Representing Primitives: The Architectural-Properties List

Primitive components and wires are represented procedurally within a prototype frame. We assumed the existence of a layout language with three keywords, EXPAND, PLACE and CONNECT.

EXPAND causes the behavioural description of a component expressed in a DESIGN slot to be expanded into an architecture. The syntax of an EXPAND is

EXPAND (*design-slot-name*).

PLACE causes a specific instance of a primitive component to be created. The syntax of a PLACE is

PLACE (*component-id, placement-name, instance-number*).

The *component-id* is the name of a primitive component or the name of a DESIGN slot. The *placement-name* and the *instance-number* are used to uniquely identify the placement instance of *component-id* in subsequent CONNECT statements. The *instance-number* is used to support the automatic placement of multiple instances of the same *component-id*.

CONNECT creates a conductive link between the terminals of two components. The syntax of a CONNECT statement is

CONNECT (*terminal-spec-1*, *terminal-spec-2*)

where *terminal-spec* identifies the terminals of the components to be interconnected. A *terminal-spec* consists of a component's *placement-name* and *instance-number* pair, the name of a component port, and a number representing the bit address of a terminal in that port. Figure 4.5 is a set of layout instructions which place an adder and register on a logical sheet and then connects them.

These procedural descriptions of the circuit's architecture make up the ARCHITECTURAL-PROPERTIES list. The instructions in the list must be causally

```
EXPAND ( adder );
EXPAND ( store_reg )
PLACE ( adder, sysadd, 1);
PLACE ( store_reg, augend, 1 );
FOR i := 1 to 7 DO
CONNECT ( augend, 1, reg_out, 1,
sysadd, 1, augend_in, i);
```

Figure 4.5 Example Layout Instruction

ordered, that is, a component must be PLACEd before it can be CONNECTed.

We have described how the behaviour of circuits are represented using the PASCAL language and how the architecture of the circuit is represented using DESIGN slots and the ARCHITECTURAL-PROPERTIES list. What must be shown now is the linkage between the circuit's architecture and its specified behaviour.

4.3 Relating Circuit Architecture to Behaviour - The Frame Header and the Dependency Network

The frame header is similar to the CIROP grammar pattern rule because both describe the behaviour the circuit can implement. The CIROP circuit grammar pattern does not show how each of the subcomponents implements to the overall behaviour of the circuit. For CIROP this was not necessary because design modification is not one of CIROP's design strategies. The frame header, on the other hand, must not only describe the behaviour the circuit implements, but also show how the circuit's architecture implements the behaviour.

The structure of the header is quite similar to that of a DESIGN slot. There are four facets to the header, PORT-SYMBOLS, REG-SYMBOLS, RT-TYPE, and FUNCTIONAL-CAPABILITIES each of which corresponds to a facet of a DESIGN slot. The FUNCTIONAL-CAPABILITIES describes the behaviour the circuit implements and corresponds to the FUNCTIONAL-REQUIREMENTS facet of the DESIGN slot. Each instruction in the FUNCTIONAL-CAPABILITIES has a *dependency record* associated with it. Each dependency record has four types of dependency pointers, the INSTRUCTION, IS-CARRIED-OUT-BY, CTRL, and DATA-PATH pointers. These pointers form the basis of the dependency network and record the architectural

resources that are required to support a specific feature of the circuit's behaviour. Figure 4.6 shows the format of a dependency record.

The INSTRUCTION pointer points to the header instruction the dependency record is associated with. There is only one INSTRUCTION pointer associated with a dependency record. An instruction in a dependency record is identified by the tuple

(*statement_number, instruction_number*).

The IS-CARRIED-OUT-BY pointer is a list of pointers which points to the instructions that subcomponents must carry out to implement the behaviour specified in the header instruction. Each pointer is tagged with the RT-TYPE of the instruction's supporting component. Each pointer of the *carry-out-support-ptr-list* is identified by the quadruple

(*RT-TYPE, component-id, statement_number, instruction_number*)

The CTRL pointers point to the CONNECT statements in the ARCHITECTURAL-PROPERTIES list which generate the control paths that must be in place for each of the subcomponents supporting the instruction pointed to by the INSTRUCTION pointers. Each signal pointed to in the *path-list* is described by three pointers, one which points to the signal source, one which points to the signal destination, and one which points to the layout statement in the

```
(INSTi (( INSTRUCTION header-instruction-ptr)
                (IS-CARRIED-OUT-BY carry-out-support-ptr-list)
                (CTRL path-list)
                (DATA-PATH path-list)
```

Figure 4.6 Format of a Dependency Record

ARCHITECTURAL-PROPERTIES list that creates the path. A signal source or destination is identified by the triple

(*placement_name, instance_number, port_name*).

The layout statement which generates the path is identified by the arbitrary name that is assigned to it in the ARCHITECTURAL-PROPERTIES list. Figure 4.7 shows an example of a *path-list* for a CTRL pointer.

The DATA-PATH pointer is identical to the CTRL pointer except that it points to the paths which must be in place for the instruction operands to be made available. Figure 4.8 is an example of a dependency record.

This instruction sets a register named "areg" to zero. For this instruction to be carried out, a component of RT-TYPE controller must execute instruction

(controller, ctrl, 20, INST1)

and a component of RT-TYPE "storage" must execute instructions

(storage, areg, 10, INST1) and (storage, areg, 20, INST1).

A CTRL path from the "clear" port of ctrl to the "clear" port of the "areg" provides the only control signal required. No data paths are required.

```
((SRC   ctrl, 1, shift))
 (DEST areg, 1, shift))
 (PATH c013)) ))
```

Figure 4.7 Path List for a CTRL Pointer.

```
init_reg ( areg, sim_$zero )


(DEPENDENCY (
            (INSTRUCTION ( 20, INST2) )
            (IS-CARRIED-OUT-BY ((controller, ctrl, 20, INST1)
                                (storage, areg, 10, INST1)
                                (storage, areg, 20, INST1) ))
            ((CTRL (
                   ((SRC ctrl, 1, clear)
                    (DEST areg, 1, clear)
                    (PATH c002)) ))
            (DATA-PATH nil) ))
```

Figure 4.8 Example of a Dependency Record

4.3.1 The DEPENDENCY Attribute

The IS-CARRIED-OUT-BY, CTRL and DATA-PATH are pointers from an instruction describing a behavioural feature to the supporting architectural feature. The DEPENDENCY attribute associated with each of the supporting architectural features within the DESIGN slots and the ARCHITECTURAL-PROPERTIES list records how many of these pointers fall onto that architectural feature and thereby justify its existence. The number associated with the DEPENDENCY attribute indicates how many pointers point to that feature. During a transformation, if the DEPENDENCY number for a port, wire, subcomponent or statement falls to zero then there is no justification for feature's existence and it is removed.

One DEPENDENCY is recorded for an instruction in the FUNCTIONAL-REQUIREMENTS of a DESIGN slot for each reference to that instruction that is made in an IS-CARRIED-OUT-BY slot of the dependency record.

One DEPENDENCY is recorded for a CONNECT statement which is referenced in the PATH attribute of a CTRL or DATA-PATH slot in the dependency record. One DEPENDENCY is recorded between an EXPAND statement and its associated DESIGN slot. Finally, one DEPENDENCY is recorded for each PORT and REGISTER for each time it is referenced in a header instruction.

The dependency network lists the architectural resources required to support a feature of a circuit's behaviour. This is the type of information that is required by the rules which use the prototype to design a new circuit.

4.4 Representing the Knowledge Required to Use Prototypes

The knowledge required to use the circuit prototypes is represented as a collection of IF-THEN rules. IF-THEN rules are more suitable than frames because we wish to represent "how to" knowledge rather than knowledge about an object's properties.

The rules used in the system range from very simple rules, which look for identical matches, to rules with complex antecedents which must make reasonably sophisticated modifications to a circuit prototype. The modifications which can be made to a circuit are restricted by the four rule classes defining a similar circuit given in chapter three. This restriction is imposed because we really do not want to be in a position where we are bashing multiplexers into counters or multipliers into adders.

4.4.1 Rule Base Architecture

Rules are organized into a production system that consists of a working memory, a rule base and an interpreter. A description of the interpreter is provided in the section on control.

The working memory consists of three major data structures:

1. A list of the FUNCTIONAL-REQUIREMENTS, PORT-SYMBOLS and REG-SYMBOLS from the DESIGN slot requiring instantiation.
2. A list of the FUNCTIONAL-CAPABILITIES, PORT-SYMBOLS and REG-SYMBOLS from the frame header of the candidate prototype.
3. A symbol table which is constructed dynamically during the design process.

The rules are an explicit, declarative representation of the knowledge an engineer may use to determine if the behavioural specifications for the prototype is similar to that of the proposed circuit. The IF portion of the rule is a conjugation of conditional clauses that must be met if the THEN part of the rule is to be executed. Figure 4.9 is an example of a rule.

```
IF object-type of <port-1 is PORT
   AND object-type of <port-2 is PORT
    AND source-type of <port-1 is user-spec
   AND source-type of <port-2 is prototype
   AND width of <port-1 & <port-2 is the SAME
   AND io of <port-1 or <port-2 is input-output
   AND info-type of <port-1 & <port-2 is the SAME
   AND binding of <port-1 & <port-2 is NOT weak-bound
THEN
   weak-bind ( <port-1, <port-2 )
```

Figure 4.9 Example Rule

Each of the conditional clauses in the IF part or antecedent tests if a property of an object has a specific value or set of values. An object can be one of the following entities:

1. Instructions,
2. Statements,
3. Ports, or
4. Registers.

Objects may be from either the prototype or user specification. If the object is part of the prototype then its name will be prefixed by "prototype" in the symbol table. If an object is part of the user specification then its name will be prefixed by "user-spec".

Each object has a number of properties associated with it. All objects have at least three properties:

1. name,
2. source type (prototype or user-spec); and
3. match type (no match, weak match, strong match).

In addition to these properties an object may have a number of object specific properties. For example, registers have a width property, instructions have control-path and data-path properties, and ports have IO and INFO-TYPE properties. Instruction operands are considered to be properties of the instruction. For example, a BUS_IN instruction will have an *input-src* property, a *input-dest* property, a *input-src-width* property, and a *input-dest-width* property. These properties can be identified in a rule.

Objects can be represented by variables in a rule. The left angle bracket "<" is used to designate symbols in a rule that are variables. Variables are bound to values

in the order in which they appear in a rule. The first occurrence of a variable in a rule will cause the system to search for an object that can have the property described by the conditional clause the variable occurs in. Variables are local to the rule in which they appear.

If all the conditional clauses of a rule can be satisfied then the rule *fires* and executes the action items listed in the THEN or consequent portion of the rule. The action items of a rule consequent may:

1. Add an object to the symbol table.
2. Update the property attributes of an object in the symbol table.
3. Declare a match between a user-spec and prototype object.
4. Declare a mismatch between objects appearing in the working memory is explainable and modify the prototype.

The type of action items that make up a rule's consequent divide the rules into three broad categories, the *symbol table construction* rules, the *matching* rules and the *transformational* rules. The symbol table construction rules install objects into the symbol table. Initially, this is a very simple task as registers and ports from both the prototype and user-spec are placed into the symbol table. During the application of other rules, a transformational rule may create new registers and ports which must be subsequently installed.

The matching rules establish bindings between the objects of the working memory by linking them in the symbol table. The matches are used to find the behavioural features of the prototype that are identical to those of the user specification. There is some overlap between the symbol table construction rules and the matching rules because instructions and statements are not installed into the symbol

table until there is a match established for them. Figure 4.10 is an example of a matching rule.

There are two levels of matches defined within our system, *weak* and *strong*. A weak match occurs when the characteristic attributes of two objects fall into the same class or satisfy some predefined set of constraints. For example, the instructions

move (a, 0, 7, b, 0, 7)

move (x, 8, 15, y, 0, 7)

will weak match if "a" and "x" are both registers or ports and if "b" and "y" are both registers or ports. These two instructions will not weak match if one of the sources, "a" or "x", is a port and the other is a register. A strong match occurs when the characteristic attributes of two objects are identical.

The transformational rules modify the prototype's ARCHITECTURAL-PROPERTIES and DESIGN slots such that the prototype will implement the behaviour specified by the user. A transformational rule may have to:

1. Modify the prototype's header instructions.

```
IF object-type of <instr-1 is INSTRUCTION
    AND object-type of <instr-2 is INSTRUCTION
    AND source-type of <instr-1 is prototype
    AND source-type of <instr-2 is user-spec
    AND binding of <instr-1 & <instr-2 is strong-bound
    AND binding of operand(<instr-1) & operand(<instr-2) is
        strong-bound
THEN
    strong-bind ( <instr-1, <instr-2 )
```

Figure 4.10 Strong Instruction Matching Rule.

2. Modify the prototype data paths supporting the instruction.

3. Modify the prototype control net supporting the instruction.

4. Modify the functional requirements of the components supporting the header's instructions.

Figure 4.11 is a transformational rule which detects the case where the difference between the specification for the prototype and the user-spec requirements for a shift register is the shift in source.

There is a strong assumption in the transformational rules that circuits are designed with respect to some generally accepted principles of circuit design. The action items within the rules operate on an expected general organization of the circuit. If a

```
IF object-type of <instr-1 is INSTRUCTION
    AND object-type of <instr-2 is INSTRUCTION
    AND source-type of <instr-1 is prototype
    AND source-type of <instr-2 is user-spec
    AND object-type of <port-1 is PORT
    AND source-type of <port-1 is prototype
    AND width of <port-1 is 1
    AND object-type of <reg-1 is REGISTER
    AND source-type of <reg-1 is prototype
    AND object-type of <reg-2 is REGISTER
    AND source-type of <reg-2 is user-spec
    AND operator-type of <instr-1 & <instr-2 is SHIFTREG
    AND shift-in-src of <instr-1 is <port-1
    AND shift-in-src of <instr-2 is CONSTANT
THEN
    RM-SRC   ( <instr-1, shift-in-src, <port-1 )
    ADD-SRC  ( <instr-1, shift-in-src, CONSTANT )
    RM-PATH  ( <instr-1, shift-in-src, <port-1 )
    ADD-PATH   ( <instr-1, shift-in-src, CONSTANT )
    RPL-INSTR-SRC ( <instr-1, shift-in-src, CONSTANT )
```

Figure 4.11 Modified Shift Input Source Rule

design is a radical departure from conventional circuit architectures then the rules will fail to transform the circuit properly.

4.5 Control of the Design Process

Control of the design process is divided between three independent mechanisms, the *instantiator*, the *designer* and the *prototype manager*. Communication between these three mechanisms is supported by a message system.

4.5.1 Instantiator Control

The instantiator's role in the design task is to expand a frame into its constituent components. Design progresses in the order in which the prototype's ARCHITECTURAL-PROPERTIES are listed. The ARCHITECTURAL-PROPERTIES list is therefore not only a procedural description of the prototype's architecture, but is also an agenda that specifies when the prototype's DESIGN slots should be expanded and when the ARCHITECTURAL-PROPERTIES should be instantiated.

The order in which the EXPAND commands appear within the ARCHITECTURAL-PROPERTIES is important. The objects of the EXPAND commands should be listed by decreasing difficulty. In the multiplier design for example, it is believed that the accumulator may be the most difficult element to find or to develop a satisfactory architecture for and therefore may be the design task which is most likely to fail. If a subcomponent cannot be designed, then the global design will fail. In the interests of efficiency we would like to abandon a design early if it is unrealizable.

The instantiator uses stack based control. When it picks up a new prototype it will push the ARCHITECTURAL-PROPERTIES list onto its stack and then begin executing the command which appears at the top of the stack. The first element of the ARCHITECTURAL-PROPERTIES list will appear as the top element of the stack.

An EXPAND command requires the instantiator to request an architecture from the designer. If the designer is successful, it will return an architecture that will implement the behaviour specified in the DESIGN slot. If the architecture is not a primitive structure then its DESIGN slots will have to be expanded. The ARCHITECTURAL-PROPERTIES list from this new architecture will be pushed onto the instantiator's stack which results in design being carried out in a top down *depth first* fashion.

Design is complete when the instantiator's stack is empty.

4.5.2 Designer Control

The designer's task is to develop an architecture that will implement the behaviour requested by the instantiator.

There are two processing phases in the designer. The first phase is a heuristic search carried out by the prototype manager for prototype circuits which may satisfy the behavioural requirements. The purpose of this search is to limit the number of potential circuits the designer must consider in its more computationally complex second stage.

Prototypes are stored in a data base and are indexed by their RT-TYPE. If the instantiator is requesting a design for a circuit with an RT-TYPE of "processor", then only circuits which have an RT-TYPE of "processor" in the data base are considered.

Of these circuits, the data base manager compares the *relative complexity* of the requested design to those having the same RT-TYPE. The relative complexity is a simple minded count of the number of instructions appearing in an algorithm describing the circuit's behaviour.

There are two heuristics used to limit the potential number of candidate prototypes. The first is that the relative complexity of the candidate cannot be less than 75% of the relative complexity of the FUNCTIONAL-REQUIREMENTS. There is no point in using a prototype that is obviously too simple to implement a circuit behaviour. The second pruning hueristic states that the relative complexity of a candidate prototype cannot exceed the relative complexity of the FUNCTIONAL-REQUIREMENTS by a factor of two. The remaining prototypes are then sorted by their relative complexity and the prototype which has the lowest relative complexity is chosen as the candidate prototype. The header of the frame representing the candidate prototype is placed into the prototype side of the system's working memory.

The rule matcher works by forward chaining. Each object in the working memory is compared to the conditional clauses of the rules. When a rule is complete then it will fire and add new objects and relationships to the working memory which may cause other rules to fire. The rules appearing in the rule base are ordered by priority. Those rules appearing first in the rule base have the highest priority. The symbol table construction rules have the highest priority and the transformational rules have the lowest priority. The rule matcher steps through the rules in the rule base, querying each rule to see if it is ready to fire. If a rule can fire, then the action items listed in its consequent are applied to the working memory and then control returns to the first rule. If none of the rules can fire during a cycle then either

there is a completed match or a failure. If all the memory elements of the user-spec are matched then the prototype implements the behaviour of the user-spec, otherwise there is failure. The rule matcher follows the algorithm shown in figure 4.12.

The algorithm given in figure 4.12 hides a bit of what happens under the surface of the SEARCH step. When a rule is first selected the designer will sequentially search the working memory for an object that has the properties that will satisfy the conditional clause. Once an object has been found the rule matcher does not immediately proceed to find a match for the next conditional clause in the rule's antecedent. Rather, it will generate another copy of the rule and continue the search

```
START : -get first rule from rule base

CONT   : -get first conditional clause of rule
SEARCH: -search for an element in working memory
          which satisfies the conditional clause.

         -IF the conditional clause is satisfied THEN
            IF all rule conditions are now satisfied THEN
               -execute the rule's consequent.
               -GOTO START
            ELSE
               -get conditional clause of rule
               -GOTO SEARCH
          ELSE IF last rule THEN
            -GOTO HALT
          ELSE
            -get next rule
            -GOTO CONT

HALT   :  -stop
```

Figure 4.12 Rule Matcher Algorithm

for a working memory object for the first conditional clause of that rule. This match and copy process stops when there are no more objects in the working memory which satisfy the first conditional clause in the rule. The algorithm then steps back to the SEARCH step of the algorithm with the next conditional clause of the rule.

Satisfying the next conditional clause within the context of the match for the first conditional clause may prove impossible, and those instances of the rule for which no match can be found are deleted. If there are no instances of a rule left then there is no applicable case for this rule and the next rule is tried.

When all the conditional clauses of a rule are satisfied then the rule is complete and the rule can fire. If there is more than one rule which can fire, then the first rule on the list is fired and has its consequent actions applied to the working memory.

The remaining rules cannot be fired because the consequent of the rule which has just fired may have altered the working memory and enabled a higher priority rule to fire. There are two options for dealing with completed instances of unfired rules:

1. Erase the unfired rule instances. This is the simplest method for dealing with the problem but is inefficient because the information gained in the rule bindings is lost.
2. Maintain the unfired rule instances until they can be tested again. This strategy preserves the matches, but requires an extensive dependency network to track the changes to working memory objects bound to the rule's conditional clauses.

4.5.3 Message Communications

In order to decouple the control mechanisms, a message based communications system is used. There are six message types defined:

1. DESIGN-REQUEST is sent by the instantiator to the designer when the instantiator requires expansion of a DESIGN slot. The DESIGN-REQUEST message carries all the information contained in the facets of the DESIGN slot to the designer. The format of a DESIGN-REQUEST message is

   (DESIGN-REQUEST ((ID ...) (PORT-SYMBOLS ...) (REG-SYMBOLS ...)
   (RT-TYPE)(FUNCTIONAL-REQUIREMENTS ...)))

2. POLL messages are sent by the designer to the prototype manager when it is looking for potential candidates with which to satisfy the DESIGN-REQUEST from the instantiator. The designer computes the relative complexity of the requested design and places this along with the RT-TYPE of the design into the POLL message. The POLL message is then sent to the prototype manager. The prototype manager will compare the relative complexity and RT-TYPE in the message to the corresponding values appearing in potential candidates. The format of a POLL message is

   (POLL ((RT-TYPE ...) (REL-CMPLX ...)))

3. BID messages are sent by the prototype manager in response to POLLs when candidate frames have been selected which meet the requirements specified in the POLL.

   (BID ((FRAME-ID ...) (REL-CMPL ...)))

4. POLL-FAIL is sent by the prototype manager if no candidate designs can be found in the data base in response to a POLL. The format of a POLL-FAIL message is

(POLL-FAIL)

5. DESIGN-COMPLETE message is sent by the designer to the instantiator if a successful match is found. The format of a DESIGN-COMPLETE message is

   (DESIGN-COMPLETE (FRAME-ID))

6. DESIGN-FAIL message is sent by the designer to the instantiator if none of the candidate designs could be made to satisfy the requirements. The format of the DESIGN-FAIL message is

   (DESIGN-FAIL)

4.6 Revisiting the Model

This chapter has provided a description of how the model components identified in chapter three could be implemented. We have shown how we use an augmented PASCAL to specify and represent the behaviour of digital circuits. The architecture of a circuit is represented by the DESIGN slots and ARCHITECTURAL-PROPERTIES list of a frame. The dependency network is used to relate the individual instructions that describe a circuit's behaviour to the architectural resources required to implement the behaviour. Finally, we have encoded the knowledge required to use a circuit as a design prototype in a set of IF-THEN rules.

The next chapter is an example of how a circuit may be designed using this model.

CHAPTER 5

THE DESIGN OF AN UNSIGNED MULTIPLIER

A design begins with the user writing a PASCAL program to specify the behaviour of the proposed circuit. For this example we will design a circuit which implements the behaviour described by the program given in figure 5.1.

The design process is started by pushing on the instantiator's stack the command EXPAND (multiplier).

The instantiator begins its design cycle by popping the top of its stack and executing the command. The EXPAND command tells the instantiator to format a DESIGN-REQUEST message and mail it to the designer.

The designer pre-processes the DESIGN-REQUEST by counting the number of instructions appearing in the FUNCTIONAL-REQUIREMENTS to obtain the relative complexity of the requested circuit. The relative complexity of the circuit in this example is thirteen. The designer then formats the RT-TYPE of the circuit and its relative complexity into a POLL message and sends the message to the prototype manager.

When the prototype manager receives a POLL message, it searches for all prototypes having the same RT-TYPE as the the requested design. In our assumed data base there are four prototypes which have an RT-TYPE of processor as required by this design. They are RDIV, a restoring divider, UMULT, an unsigned multiplier, AND, an and gate, and OR, an or gate. The prototype manager then compares the relative complexity of the candidates to that of the request. RDIV has a relative complexity of 20 which does not exceed the complexity cutoff limit and UMULT has

```
{ FUNCTIONAL-CAPABILITIES }

PROCEDURE mult_function;

  VAR
    j                              : integer16;
    state_ptr                      : mult_state_ptr_type;
    k                              : integer16;

  BEGIN

    WITH simbase_$instance_ptr^:i DO
      state_ptr := i.user_data_area;
    WITH state_ptr^:sp DO
      BEGIN

{ CLOCK_BEGIN, 0 }

{ DEC, 1 }
{ INST1 } IF (signal_in (sp.start) = sim_$one) THEN
            BEGIN
{ ACT, 2 }
{ INST1 }     bus_in (sp.data_port, 0, 7, sp.multiplier, 0, 7);
{ INST2 }     init_reg  (sp.accumulator, sim_$zero);
{ INST3 }     init_reg  (sp.counter, sim_$zero);
{ INST4 }     bus_in (sp.data_port, 8, 15, sp.multicand, 0, 7);
              sp.clock_cycle := clock1;
            END
          ELSE CASE sp.clock_cycle OF

{ CLOCK_STEP, 3 }

        clock1: BEGIN
{ DEC, 4 }
{ INST1 }   IF (decode_reg (sp.counter) < 8) THEN
              BEGIN
{ DEC, 5 }
{ INST1 }       IF (sp.multiplier.reg_val[0] = sim_$one) THEN
{ ACT, 6 }
{ INST1 }         add_reg ( sp.accumulator, 8, 15,
                            sp.multicand, 0, 7,
                            sp.accumulator, 8, 15,
                            sp.carry_out ) ;
                sp.clock_cycle := clock2i ;
              END
            ELSE
              sp.clock_cycle := clockn;
          END;

{ CLOCK_STEP, 7 }

        clock2i: BEGIN
{ ACT, 8 }
{ INST1 }       shift_reg ( sp.accumulator, 0, 15, RIGHT, sp.carry_out);
{ INST2 }       shift_reg ( sp.multiplier, 0, 7,  RIGHT, sim_$zero);
{ INST3 }       incr_reg ( sp.counter);
            sp.clock_cycle := clock1;
          END;

{ CLOCK_STEP, 9 }

        clockn: BEGIN
{ ACT, 10 }
{ INST1 }       signal_out (sp.stop, sim_$one);
{ INST1 }       bus_out (sp.data_port, 0, 15, sp.accumulator, 0, 15);
          END;

      END;

{ CLOCK_END, 11 }

    END;
```

Figure 5.1 Specification for an Unsigned Multiplier

a relative complexity of 14. Both prototypes are chosen as potential candidates with which to satisfy the DESIGN-REQUEST. AND and OR both have a relative complexity of 1 which cuts them off from further consideration. Appendix one is a listing for the multiplier frame used as the prototype in this chapter. Appendix two is

a partial description of a possible prototype for a restoring divider.

The prototype manager formats two BID messages, one for each candidate prototype and then mails them to the designer.

When the designer receives the BID messages, it sorts the responses by their relative complexity. The prototype with the lowest relative complexity is used. The designer uses the FRAME-ID in the BID message to access the data base for the candidate prototype and then copies the prototype's frame header into the prototype's side of the working memory. The REG-SYMBOLS, PORT-SYMBOLS and FUNCTIONAL-REQUIREMENTS appearing in the DESIGN-REQUEST message are copied into the user-spec side of the working memory.

5.1 An Overview of the Multiplier Design Process

The flowcharts shown in figure 5.2 represent the structures of the algorithms used to describe the behaviour of the prototype multiplier and the user-spec multiplier. Each node in the graph represents one instruction (bubbles are used to represent the clock-step statements) and crosses appearing on the arcs connecting the nodes indicate statement boundaries. All instruction nodes between the boundaries are members of one statement. This graphical representation of behaviour will be used to illustrate the design process in this chapter.

Inspection of the two flowcharts show that the behaviour of these two circuit's is almost identical except for the input and output of data. The prototype multiplier reads and writes data half word serially while the user-spec multiplier reads and writes data word parallel. If this difference between the behaviour of the prototype and the

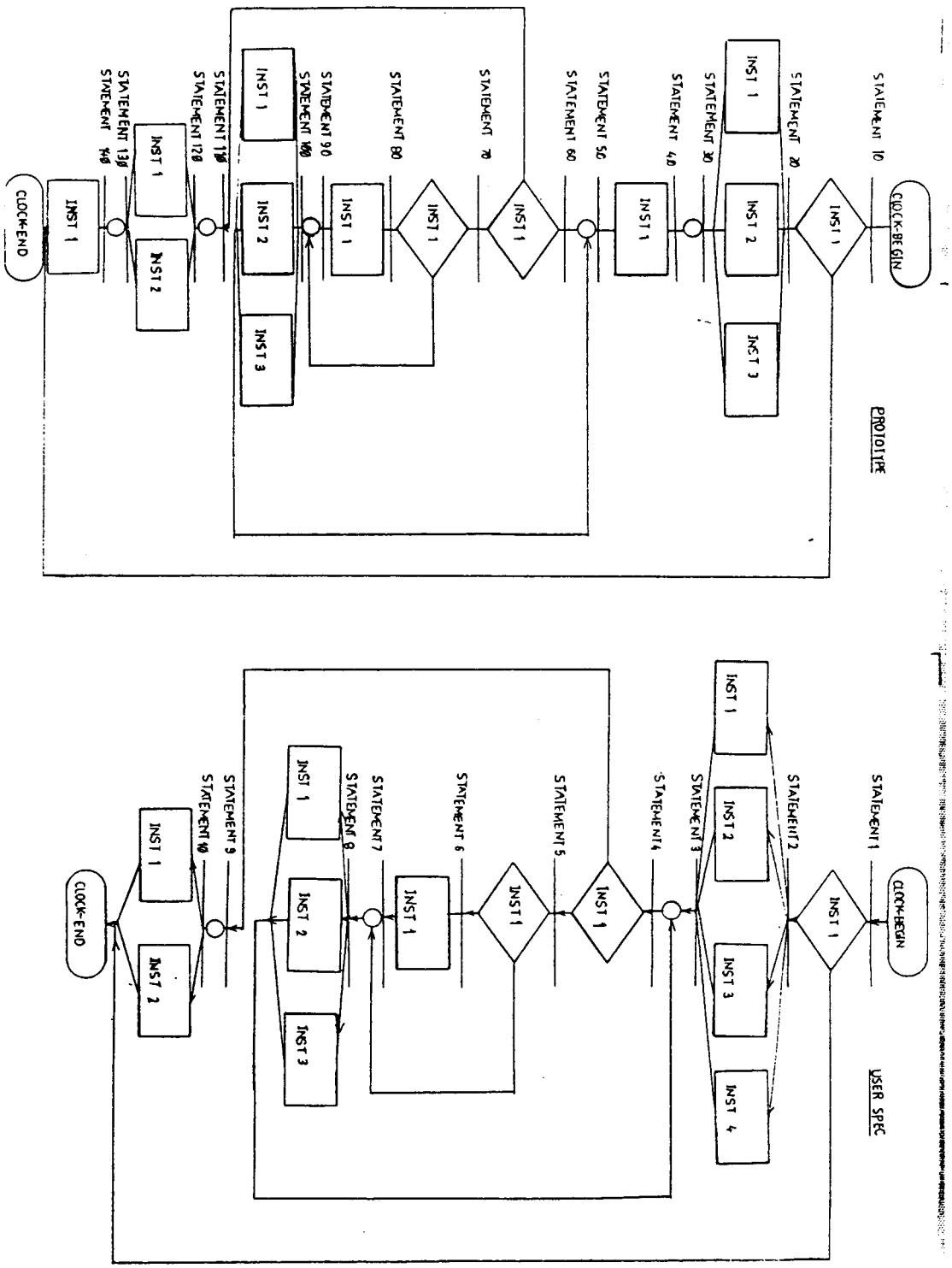


Figure 5.2 Comparative structure of the prototype and user-spec algorithms

user–spec can be resolved then the architecture of the prototype can be used to implement the user–spec.

This is the task of the designer, to resolve the differences between the behaviour of the prototype and of the user–spec, and to return to the instantiator an architecture that will implement the user–spec. The designer will start by installing the register and port symbols from the prototype and user–spec into the symbol table. Any bindings that can be made between the ports of the prototype and the ports of the user–spec will be established.

When non of the symbol table constructor rules can fire, then the designer will poll the matching rules to see if they are ready to fire. Bindings will be established between the instructions and statements of the prototype and the user–spec. These bindings may also result in bindings being established between the registers of the prototype and the registers of the user–spec.

These matches and their resultant bindings identify those portions of the prototype's behaviour that are the same as the user–spec's. This implies that the architectural features of the prototype that implement the matched behavioural features can be directly used to implement the behaviour of the user–spec.

Some of the behavioural features of user–spec cannot be matched to the behavioural features of the prototype. This is the case between the input–output behaviour of the prototype and the user–spec. The transformational rules are then used to modify the behaviour of the prototype and the architecture that supports its behaviour. The transformational rules will be applied until the behaviour of prototype can be matched to the user–spec or until non of the transformational rules can explain the differences. This latter case is a design failure and indicates that the

designer does not know enough to use the prototype as a design guide for this case.

5.2 Matching the Prototype Multiplier to the User-Spec Multiplier

5.2.1 Building the Symbol Table

The first rules to fire are the symbol table construction rules. The repeated application of these rules eventually leads to the construction of the symbol table shown in figure 5.3.

```
(OBJECT=port, NAME=data_in, SOURCE_TYPE=prototype, WIDTH=*n,
                IO=input-output, INFO_TYPE=data)
(OBJECT=port, NAME=data_out, SOURCE_TYPE=user-spec, WIDTH=*n,
                IO=output, INFO_TYPE=data)
(OBJECT=port, NAME=start, SOURCE_TYPE=prototype, WIDTH=1,
                IO=input, INFO_TYPE=ctrl)
(OBJECT=port, NAME=stop, SOURCE_TYPE=prototype, WIDTH=1,
                IO=output, INFO_TYPE=ctrl)
(OBJECT=port, NAME=data_port, SOURCE_TYPE=user-spec, WIDTH=16,
                IO=input, INFO_TYPE=data)
(OBJECT=port, NAME=start, SOURCE_TYPE=user-spec, WIDTH=1,
                IO=input, INFO_TYPE=ctrl)
(OBJECT=port, NAME=stop, SOURCE_TYPE=user-spec, WIDTH=1,
                IO=output, INFO_TYPE=ctrl)
(OBJECT=register, NAME=qreg, SOURCE_TYPE=prototype, WIDTH=*n)
(OBJECT=register, NAME=mreg, SOURCE_TYPE=prototype, WIDTH=*n)
(OBJECT=register, NAME=areg, SOURCE_TYPE=prototype, WIDTH=*2n)
(OBJECT=register, NAME=creg, SOURCE_TYPE=prototype, WIDTH=*m)
(OBJECT=register, NAME=multiplier, SOURCE_TYPE=user-spec, WIDTH=8)
(OBJECT=register, NAME=multicand,  SOURCE_TYPE=user-spec, WIDTH=8)
(OBJECT=register, NAME=accumulator, SOURCE_TYPE=user-spec, WIDTH=16)
(OBJECT=register, NAME=counter, SOURCE_TYPE=user-spec, WIDTH=4)
```

Figure 5.3 Initial Symbol Table Configuration

It can be observed that the width of data paths in the prototype are parameterized. During the matching process these parameters will be bound to values.

The next rule which becomes enabled after the symbol table is constructed is the Strong_Port_Match rule. The prototype ports "Start" and "Stop", and the user-spec ports "Start" and "Stop " respectively have identical attributes. There is strong evidence then that there may be a one to one correspondence between "Start" and "Stop" in the user-spec and "Start" and "Stop" in the prototype. This correspondence is tagged as a STRONG binding. Only ports and registers that have identical attributes are assigned a strong binding.

A weak binding can be established between the "data_port" port appearing in the user-spec and the "data_in" and "data_out" ports in the prototype. The Weak_Port_Match rule fires because the input-output specification of user-spec port "data_port" can matched to both the input-only and output-only properties of "data_in" and "data_out". The bit width of the "data_port" port still remains parametric however. After the application of these rules the symbol table appears as shown in figure 5.4.

5.2.2 Matching Behaviour

The next rule which can fire is the Clock_Begin_Match rule. The first statement in a user-spec is the Clock-Begin statement. This match is obtained for free because the designer knows every algorithm must start with CLOCK-BEGIN and end with CLOCK-END.

Th next match is between user-spec INST1 of statement 1,

{ INST1 } IF (signal_in (sp.Start = sim$_one)) THEN

```
(OBJECT=port, NAME=data_port, SOURCE_TYPE=user-spec, WIDTH=16,
                IO=input-output, INFO_TYPE=data,
                BINDING=(WEAK data_in, data_out))
(OBJECT=port, NAME=start, SOURCE_TYPE=user-spec, WIDTH=1,
                IO=input, INFO_TYPE=ctrl,
                BINDING=(STRONG start))
(OBJECT=port, NAME=stop_sig, SOURCE_TYPE=user-spec, WIDTH=1,
                IO=output, INFO_TYPE=ctrl,
                BINDING=(STRONG stop))
(OBJECT=port, NAME=data_in, SOURCE_TYPE=prototype, WIDTH=*n,
                IO=input, INFO_TYPE=data,
                BINDING=(WEAK data_port))
(OBJECT=port, NAME=data_out, SOURCE_TYPE=prototype, WIDTH=*n,
                IO=output, INFO_TYPE=data,
                BINDING=(WEAK data_port))
(OBJECT=port, NAME=start, SOURCE_TYPE=prototype, WIDTH=1,
                IO=input, INFO_TYPE=ctrl,
                BINDING=(STRONG start))
(OBJECT=port, NAME=stop, SOURCE_TYPE=prototype, WIDTH=1,
                IO=output, INFO_TYPE=ctrl
                BINDING=(STRONG, stop))
```

Figure 5.4 Symbol Table after Application of Port Matching Rules

and prototype statement 1 INST1,

{ INST1 } IF (signal_in (sp.Start = sim$_one)) THEN

These two instructions are matched by the Strong_Instruction_Match Rule. A strong match occurs here because the formats of the instructions are identical. Both instructions apply the same operator to operands which are strongly bound. Figure 5.5 shows the entries which are recorded in the symbol table from this match.

This instruction match triggers another rule. Both of these instructions occur in statements where they are the only instruction appearing in that statement. If all the instructions appearing in a statement in the user-spec are bound to all of the instructions appearing in a prototype statement then the Weak_Statement_Match Rule establishes a weak binding between the two statements. This weak binding implies the

```
(OBJECT=instruction, NAME=(1, INST1), SOURCE_TYPE=user-spec,
                          BINDING=(STRONG (10, INST1))
(OBJECT=instruction, NAME=(10, INST1), SOURCE_TYPE=prototype,
                          BINDING=(STRONG (1, INST1))

(OBJECT=statement, NAME=1, SOURCE_TYPE=user-spec,
                          BINDING=(STRONG 10 ) )
```

Figure 5.5 Symbol Table Entries Made after Instruction Match

two statements are considered to be identical when considered on their own.

Before two statements can be considered to be strongly matched, their preceding statement must be strongly matched and their succeeding statements must be at least weakly matched. A preceding statement is the first statement encountered which will branch to the current statement and not necessarily the physically preceding statement. The succeeding statement is the statement the current statement may branch to. For decision statements all possible branch targets must be at least weakly matched for the decision statement to be considered strongly matched.

No special processing for loop structures is required because only one predecessor must be matched for a strong statement match to be declared. The matching of a loop structure will start with the matching of the entry test (for DO-WHILE type loops) and then attempt to strong match the individual statements within the loop. If the branch target of the last statement in the loop is the strongly matched loop entry in both the user-spec and prototype, then the loop is matched. If there is no match between the branch targets of the prototype statement and user-spec statement then there cannot be a strong match.

The rule strategy is to find matches tentatively and progressively. Matches are first made between ports and registers and then between instructions. Matches are then

found between groups of instructions appearing in statements. Finally, the Strong_Statement_Match rule is used to match to the program structure.

The next possible match candidate is between INST1 of statement 2 in the user-spec

{ INST1 } BUS_IN (sp.data_port, 0, 7, sp.multiplier, 0, 7);

and INST1 of statement 20 in the prototype

{ INST1 } BUS_IN (sp.data_in, (*dfst, *dlst), sp.qreg, (*qfst, *qlst));

These two instructions weak match because both have the same general format for BUS_IN instruction. They cannot strong match because the ports are only weak matched and because there are no matchings established for the registers. These statements can also only weak match because their specific formats are different. The user-spec port source only reads from half of the port, whereas the prototype statement reads from the full width of the port.

After firing a rule, the designer goes back to the top of the rule list and starts sampling the rules over again because the application of the last rule will have altered contents of the working memory. This application of the Weak_Instruction_Match rule alters more than just the working memory's symbol table. An attempt is made to establish a binding between the registers in the instruction. A part of the action items of the Weak_Instruction_Match rule would recognize that the width of the "qreg" register referred to in the prototype is still parametrically defined,

(OBJECT=register name=qreg, width=(*qfst, *qlst))

and therefore cannot be bound to any of the user-spec registers. The target register in the user-spec instruction has WIDTH of 8 bits. Therefore, it is tentatively asserted that the value of parameter *qfst should be 0 and the value of parameter *qlst should be 7. The general strategy of the designer is if bindings for ports or registers

cannot be immediately established by matching their attributes, then matchings can be determined by their usage.

The bindings that have been established between the prototype and the user–spec are summarized in figure 5.6

5.2.3 Establishing Bindings for Registers and Ports

The application of the Weak_Instruction_Match rule made an assertion that the width of the prototypes's "qreg" register was eight bits. The prototype defines the "qreg", "mreg", and the "areg" registers to be multiples of each other. This permits the matcher to infer the width of the "mreg" register to be eight bits, and the width of the "areg" register to be sixteen bits. These assertions will pre–empt any further weak bindings between instructions because the Strong_Register_Match rule has a higher priority than the Weak_Instruction_Match rule.

The next five design cycles will result in the following bindings:

1. the user–spec "multiplier" register is strongly bound to the prototype "qreg" register.
2. the user–spec "multiplier" register is strongly bound to the prototype "mreg" register.
3. the user–spec "multicand" register is strongly bound to the prototype "qreg" register.
4. the user–spec "multicand" register is strongly bound to the prototype "mreg" register.
5. the user–spec "accumulator" register is strongly bound to the prototype "areg" register.

There are also potential weak bindings between the user–spec "accumulator" register

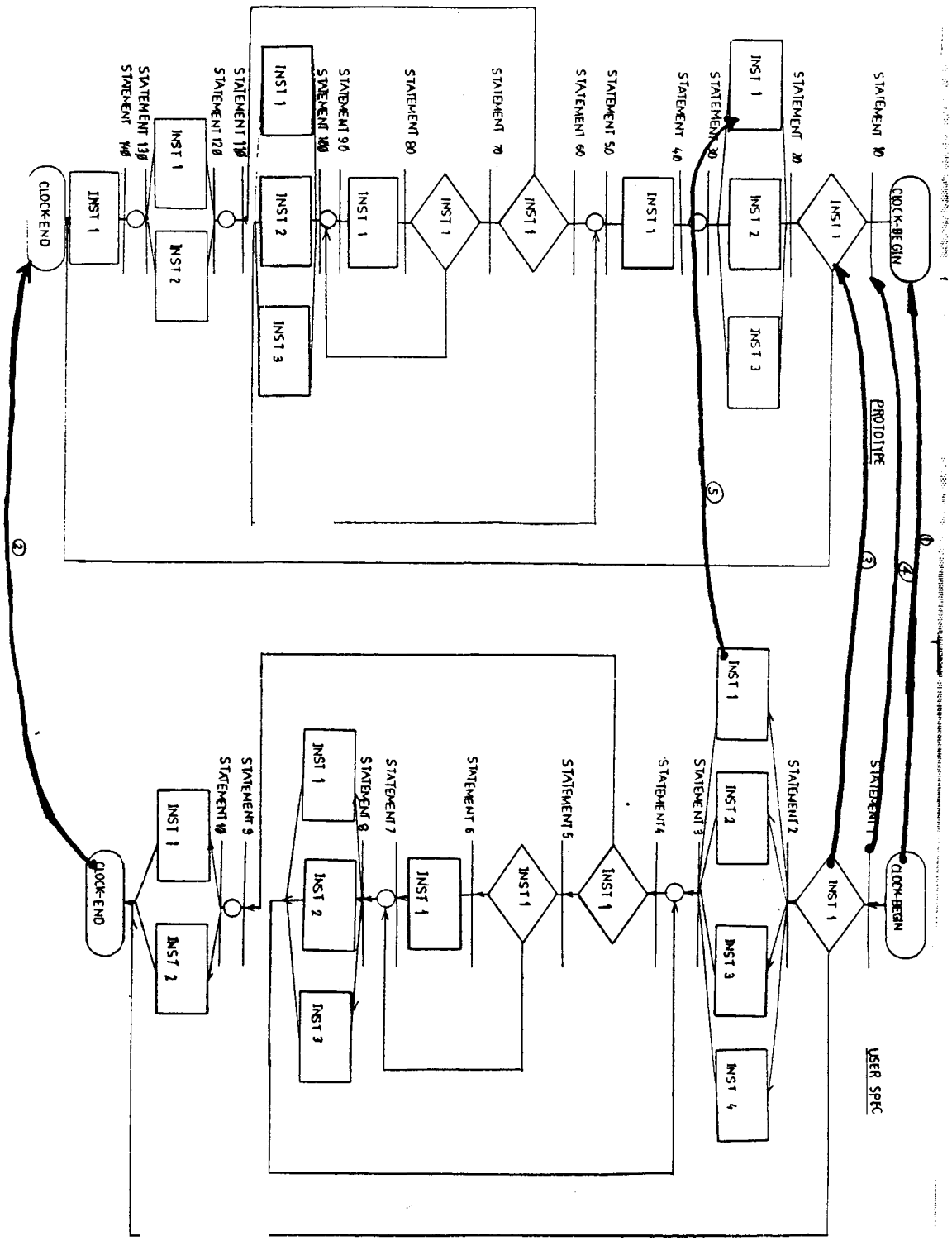


Figure 5.6 summary of the bindings established between the prototype and user-spec

and the prototype "qreg" and "mreg" registers because they satisfy the criteria of a weak match. However, a strong match will lock out a weak match. The registers' symbol table entries are shown in figure 5.7. There is some ambiguity within the bindings because there are multiple bindings between the user–spec registers and the prototype's registers. User spec registers "multicand" and "multiplier" have both been strong bound to prototype registers "mreg" and "qreg". There are three possible outcomes for multiple strong bindings:

1. Each register or port in the user–spec will have a unique binding to a register or port in the prototype. This is the most desirable outcome.
2. One or more of the user–spec register or ports will not strong bind to a prototype register or port. This is regarded as a match failure and the proposed design cannot be used to implement the user–spec.
3. One or more of the user–spec register or ports will have multiple bindings. This implies that each of the alternative bindings remained equally consistent during and the matching process and therefore it really does not make any difference how the user–spec register or ports are mapped to the prototype register or

```
(OBJECT=register, NAME=qreg, SOURCE_TYPE=prototype, WIDTH=8,
                    BINDING=(STRONG multiplier, multicand))
(OBJECT=register, NAME=mreg, SOURCE_TYPE=prototype, WIDTH=8,
                    BINDING=(STRONG multiplier, multicand))
(OBJECT=register, NAME=areg, SOURCE_TYPE=prototype, WIDTH=16,
                    BINDING=(STRONG accumulator))
```

Figure 5.7 Symbol Table after Initial Register Binding

ports because the register or ports have the same operations performed on them. This would be the case for example for individual register in a register file. The resolution of these ambiguites will be left to the strong matching rules which will determine register or port mapping by usage.

The next design cycle finds that INST2 of user-spec statement 2

{ INST2 } INIT_REG (sp.accumulator, sim_$zero)

strong binds to INST2 of prototype statement 20

{ INST2 } INIT_REG (sp.areg, sim_$zero).

It might seem that the user-spec may also bind to INST3 of prototype statement 20

{ INST3 } INIT_REG (sp.creg, sim_$zero).

This will not occur because there is only one strong binding for register "accumulator" and that is to prototype register "areg". In addition, the width of the prototype register "creg" has not been established yet. The best match that could be hoped for here would be a weak match, and a weak matches will only be made if there are no strong matches which can be made first.

The next match to occur is a strong instruction match between INST1 of prototype statement 70

IF (sp.qreg.reg_val[0] = sim_$one) THEN

and INST1 of user-spec statement 5

IF (sp.multiplier.reg_val[0] = sim_$one) THEN.

This match will be followed by a weak statement match between prototype statement 70 and user-spec statement 5.

The next match to occur is a strong instruction match between INST1 of user-spec statement 6

```
{ INST1 } ADD_REG (sp.accumulator, 8, 15,
                   sp.multicand, 0, 7
                   sp.accumulator, 8, 15,
                   carry_out)
```

and INST1 of prototype statement 80

```
{ INST1 } ADD_REG (sp.areg, 8, 15,
                   sp.mreg, 0, 7
                   sp.areg, 8, 15,
                   carry_out )
```

We can note here that the matching can only be achieved when the prototype register name "mreg" is substituted for references to the "multicand" register in the user–spec. If a register or a port has multiple strong bindings, then when that register or port is referenced in the user–spec instruction during a match attempt, each one of the prototype names will be substituted for the user–spec name. If a match cannot be found for that substitution then the assertion that the prototype name and the user–spec name are strongly bound is dropped. There are no statements of the form

```
{ INST1 } add_reg (sp.areg, 8, 15,
                   sp.qreg, 0, 7
                   sp.areg, 8, 15,
                   carry_out )
```

appearing in the prototype. Therefore, the only consistent binding for the user–spec register "multicand" is the prototype register "mreg".

The bindings that have been established between the prototype and user–spec are summarized in figure 5.8.

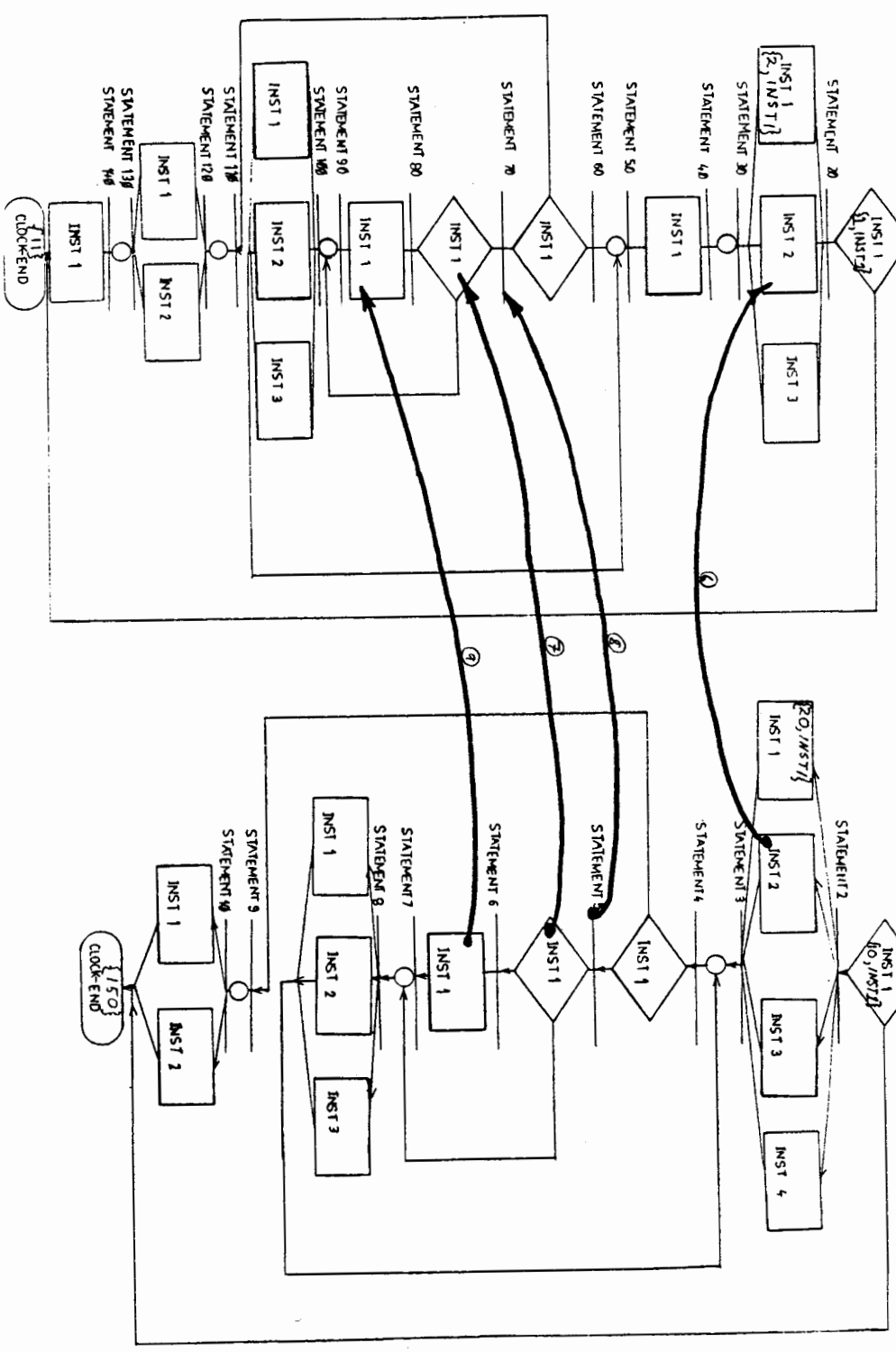


Figure 5.8 summary of bindings between the prototype and user-spec

5.2.4 Establishing Matches between Statements

The next design cycle will weak match these statements because all the instructions of user-spec statement 6 are strong matched to all the instructions appearing in prototype statement 80.

The next three design cycles will produce strong instructions matches between

1. The first instruction, INST1, of user-spec statement 8

   { INST1 } SHIFT_REG (sp.accumulator, 0, 15, RIGHT, carry_out)

   and the first instruction, INST1, of prototype statement 100

   { INST1 } SHIFT_REG (sp.areg, 0, 15, RIGHT, carry_out)

2. The second instruction, INST2, of user-spec statement 8

   { INST2 } SHIFT_REG (sp.multiplier, 0,7, RIGHT, sim_$zero)

   and the second instruction, INST2, of prototype statement 100

   { INST2 } SHIFT_REG (sp.qreg, 0, 7, RIGHT, sim_$zero)

3. The second instruction, INST2, of user-spec statement 10

   { INST2 } SIGNAL_OUT (sp.stop_sig, sim_$one)

   and the second instruction, INST2, of prototype statement 120

   { INST2 } SIGNAL_OUT (sp.end_sig, sim_$one).

No more strong matches are possible with the current state of working memory.

The next design cycle will weak match instruction INST1 of user-spec statement 2

{ INST1 } BUS_IN (sp.data_port, 8, 15, sp.multiplier, 0, 7)

to instruction INST1 of prototype statement 40

{ INST1 } BUS_IN (sp.data_in, 0, 7, sp.mreg, 0, 7).

This may seem to be a bit silly considering that bindings between the user-spec

"multiplier" register and the prototype "mreg" register have been definitely ruled out. However, it must be remembered that weak instruction matches are based on the format or structure of the instruction or statement and not the actual objects manipulated by the instruction or statement. The match here is based strictly on the fact that there is a transfer from a port to a register.

The next weak match is between the third instruction, INST3, of user-spec statement 2

{ INST3 } INIT_REG (sp.counter, sim_$zero)

and the third instruction, INST3, of prototype statement 20

{ INST3 } INIT_REG (sp.creg, sim_$zero).

Again the action items in the rule recognize that the prototype register "creg" is parametrized and binds the bit width of the user-spec "counter" register to the prototype "creg" register. This touches off another round of strong bindings.

On the next design cycle the Strong_Register_Match rule will fire establishing a strong binding between the user-spec "counter" register and the prototype "creg" register.

The next rule which will fire will be the Strong_Instruction_Match which will establish a strong binding between the third instruction, INST3, of user-spec statement 2 and the third instruction, INST3, of prototype statement 20.

The next rule which fires will be the Strong_Instruction_Match rule which will strongly match INST1 of user-spec statement 4

{ INST1 } IF (decode_reg (sp.counter) < 8) THEN

to INST1 of prototype statement 60

{ INST1 } IF (decode_reg (sp.creg) < *MAX-VAL) THEN.

The value of *MAX-VAL is automatically bound to 8. The strong binding of these instructions causes the statements to be weakly bound on the next design cycle.

The next rule to fire will strongly bind INST3 of user-spec statement 8

{ INST3 } INCR_REG (sp.counter)

to INST3 of prototype statement 100

{ INST3 } INCR_REG (sp.creg).

Because all three instructions of user-spec statement 8 are strongly bound to all three instruction of prototype statement 100, the two statements are declared to be weakly bound by the Weak_Statment_Match rule on the next design cycle.

There are four more weak matches that are established.

1. The fourth instruction, INST4, of user-spec statement 2

   { INST4 } BUS_IN (sp.data_port, 0, 7, sp.multicand, 0, 7))

   and the first instruction, INST1, of prototype statement 20

   { INST1 } BUS_IN (sp.data_in, 0, 7, sp.qreg, 0, 7)

2. The fourth instruction, INST4, of user-spec statement 2

   { INST4 } BUS_IN (sp.data_port, 0, 7, multicand, 0, 7)

   and the first instruction, INST1, of prototype statement 40

   { INST1 } BUS_IN (sp.data_port, 0, 7, mreg, 0, 7)

3. The first instruction, INST1, of user-spec statement 10

   { INST 1 } BUS_OUT (sp.data_port, 0, 15, sp.accumulator, 0, 15)

   and the first instruction, INST1, of prototype statement 120

   { INST1 } BUS_OUT (sp.data_out, 0, 7, sp.areg, 0, 7)

4. The first instruction, INST1, of user-spec statement 10

   { INST1 } BUS_OUT (sp.data_port, 0, 15, sp.accumulator, 0, 15)

   and the first instruction, INST1, of prototype statement 140

{ INST1 } BUS_OUT (sp.data_out, 0, 7, sp.areg, 8, 15)

The bindings that have been established between the prototype and user–spec are summarized in figure 5.9.

5.2.5 Match Exhaustion

With the current state of working memory there are no more possible matches, weak or strong. If we were expecting exact matches between the prototype and the user's specifications then we would have to reject the prototype even though its behaviour is quite similar to that requested by the user. The application of the transformational rules breaks this deadlock. No more matching rules can fire because there are differences between the behaviour the prototype implements and the behaviour requested by the user. Having identified these differences, it is now the task of the designer to determine if the differences are explainable. That is, can the transformational rules modify the circuit architecture such that it will implement the user's requested behaviour?

5.3 Transforming the Prototype Multiplier

5.3.1 Overview

The prototype circuit has two 8 bit ports, "data_in" for input and "data_out" for output. Reading the multiplier and the multicand operands requires two clock cycles and writing the result requires two clock cycle. The "qreg" and "mreg" register are tied directly to the "data_in" port and the "areg" writes its result to the "data_out" port through a multiplexer. Figure 5.10 is a schematic of the circuit generated by the prototype frame which implements the behaviour described in the header.

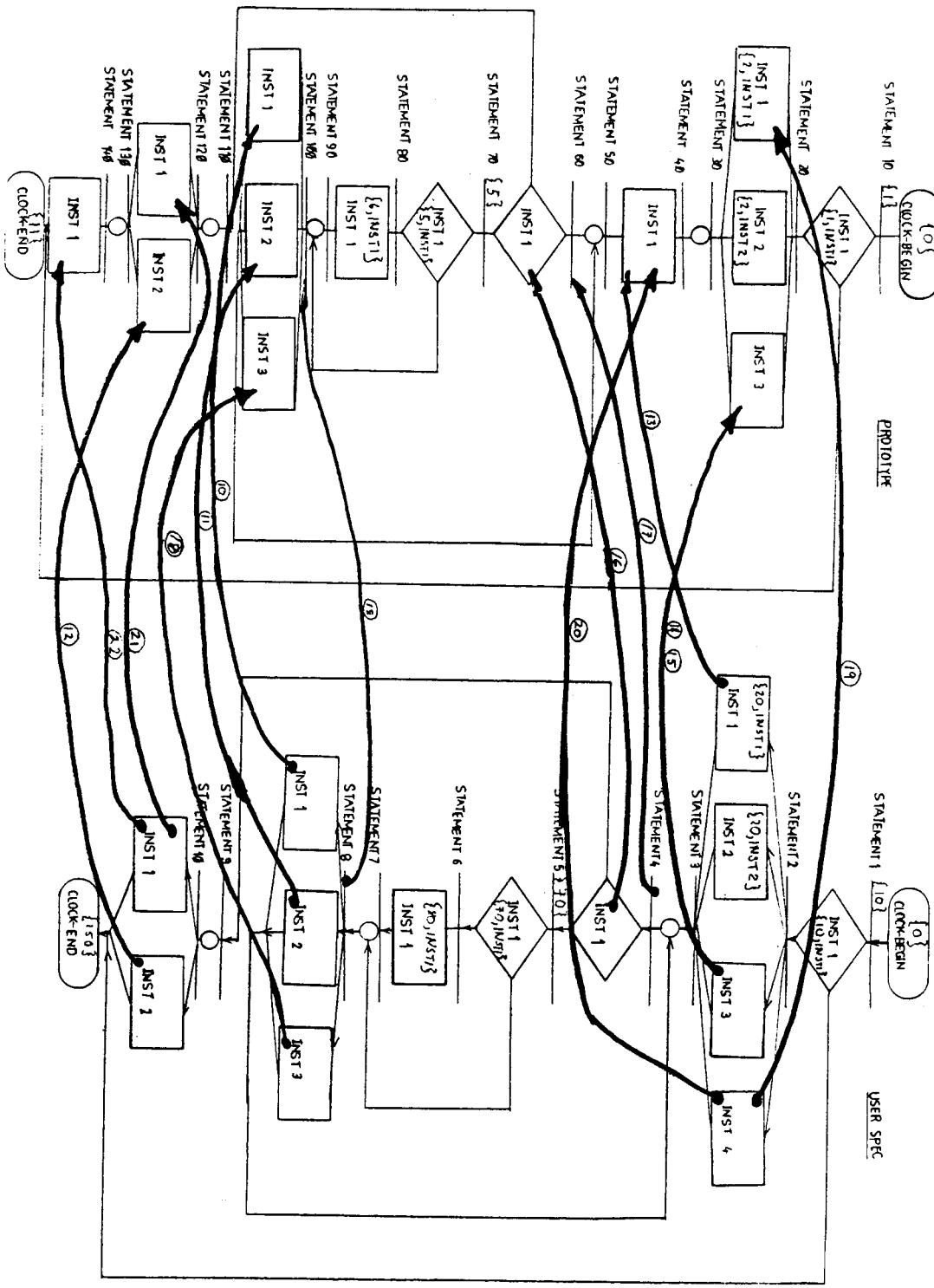


Figure 5.9 summary of bindings between the prototype and user-spec

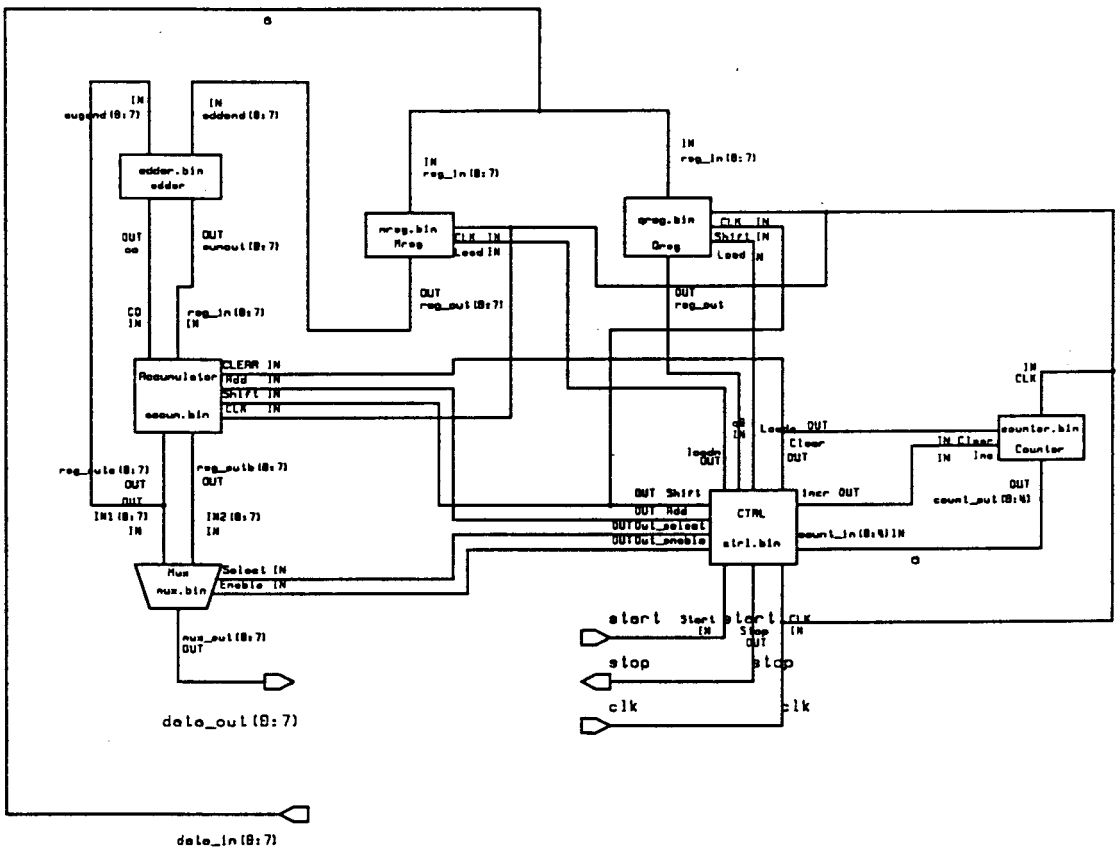


Figure 5.10 Schematic for Prototype Circuit.

What the user wishes to create is a circuit which will read two 8 bit operands, the "multiplier" and the "multicand", in one clock cycle and write the 16 bit result in one clock cycle. To implement this, the user has specified that the multiplier should have a 16 bit wide input/output port and a 16 bit internal bus off of which the "multiplier" and "multicand" registers accept their input and to which the "accumulator" register will dump its contents. Figure 5.11 is a schematic of a circuit which could implement the user's behaviour.

To make the circuit of figure 5.10 implement the user's behaviour, the designer would replace the data path from the "data_in" port to the "mreg" and "qreg"

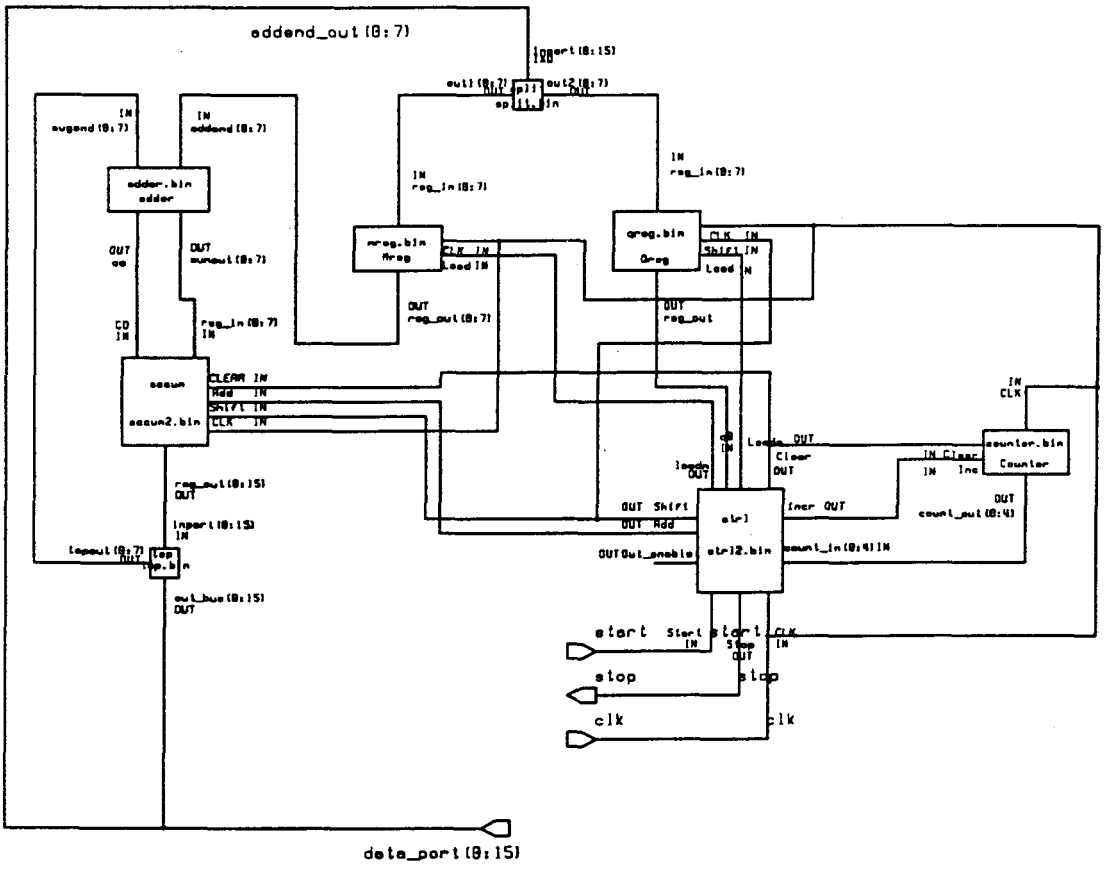


Figure 5.11 Schematic of a Possible Solution Circuit.

registers with a 16 bit data path. The output of the "areg" could be tied directly to this new port eliminating the requirements for the multiplexer. Naturally, with the "qreg" and "mreg" register reading from a port that is also used as an output port it is important that the output of the "areg" be tri-stated. The rest of this chapter shows how this transformation of the multiplier circuit is carried out.

5.3.2 A New Port

One of the most highly visible differences between the behaviour of the prototype and of the behaviour of the user-spec is that the user-spec circuit reads its operands and and writes its results to one port and the prototype circuit has two ports, one for reading the operands and one for writing its results to. These ports are somewhat similar and have been weak matched. However, a strong match is not possible because the ports have different attributes.

The Input_Output_Port_Merge rule recognizes this situation and fires. Its preconditions state there must be two ports in the prototype, an input and an output, and one input-output port in the user-spec which is the sum of the bit widths of the prototype ports. When the rule fires it will add a new port to the prototype design which will strong match with the user-spec port.

There is only one action item in the consequent of the Input_Output_Port_Merge rule. The action item

```
MAKE-PORT (NAME= <p-name-1, WIDTH=width of <port-3
                IO=input-output, INFO-TYPE=info-type of <port-3
```

creates a new prototype port with its name and attributes taken from the user-spec port <port-3. This rule does not install the new port into the symbol table. Rather, after this rule completes the next rule to fire will be the Install_Port rule which will place the new port into the symbol table. This will cause the Strong_Port_Match rule to fire which will create a strong binding between the newly created port and the "data_port" port in the user-spec. This match satisfies all of the conditional clauses of the Half_Word_to_Word_Input rule and causes it to fire.

5.3.3 Converting a Half Word Input to a Word Input

The user-spec "multicand" and "multiplier" registers are both eight bits wide and read from a sixteen bit bus in one clock cycle. In the prototype, the registers which are strongly bound to "multicand" and "multiplier", are "mreg" and "qreg" which are also eight bits wide but read from an eight bit bus in two successive clock cycles. This difference can be resolved by replacing the eight bit bus in the prototype with a 16 bit bus.

The action items RM-SRC and RM-PATH modify the dependency records that are associated with the prototype instructions bound to <instr-1 and <instr-3 in the rule antecedent. INST1 of statement 20 is bound to <instr-1 and INST1 of statement 40 is bound to <instr-3. The dependency record for INST1 of statement 20 is shown in figure 5.12.

```
{ INST1 } bus_in (data_in, 0, 7, multiplier, 0, 7)

(DEPENDENCY (
 (INSTRUCTION ( 20, INST1 ) )
  (IS-CARRIED-OUT-BY ((controller ctrl, 20, INST2) )
                          (storage qreg, 10, INST1)
                          (storage qreg, 20, INST1) ))
  (CTRL (
          ((SRC   ctrl, 1, loadq)
           (DEST qreg, 1, loadq)
           (PATH c001)) ))
  (DATA-PATH (
                ((SRC   mult, 1, data_in)
                 (DEST qreg, 1, reg_in)
                 (PATH d001)) )) ))
```

Figure 5.12 Prototype Statement 20, INST1 and Dependency Record.

These action item remove the SRC and PATH dependency pointers from the data path. The instruction will no longer have DEPENDENCY pointers to the port "data_in" and the CONNECT statements in the ARCHITECTURAL-PROPERTIES labelled "d001". This, however does not necessarily mean that these elements have been deleted, just that this instruction no longer requires these features.

No circuit feature is ever deleted outright because other entities within the circuit may depend on that feature. Therefore, the method used here to minimize the side effects of a modification is to remove the DEPENDENCY pointers as requirements for a feature are removed. Only when the DEPENDENCY attribute of a feature is zero is that feature actually deleted from the circuit. In this case the DEPENDENCY attribute for CONNECT statement d001 is zero and it is deleted. The DEPENDENCY count for the "data_in" port is not zero because some other instruction requires this circuit feature.

The register destination identified in this instruction, the "qreg" register, is connected to the new input source. The ADD-SRC transform identifies the newly created prototype port as the input source to be inserted into the SRC of the DATA-PATH. The next transform is ADD-PATH which adds a new layout command to the ARCHITECTURAL-PROPERTIES to create a link between the output ports of the "qreg" register and the new input port. The problem here becomes knowing which half of the input port to connect the "qreg" register to. This is easy to resolve since we can obtain this information from the instruction which uses the user-spec register which is strongly bound to the "qreg" register. The new layout command generated by the ADD-PATH action item is shown in figure 5.13. The action items that have been applied to the BUS_IN instruction of prototype statement 20 are also applied to the BUS_IN instruction of prototype statement 40.

```
{D$001 (DEPENDENCY 1)}

   FOR i := 0 to 7 DO
         CONNECT (mult, 1, data_port, i,
                                  multiplier, 1, reg_in, i);
```

Figure 5.13 Entry Added to the ARCHITECTURAL-PROPERTIES list by ADD-PATH Action Item

These changes to the input source must be reflected in the instruction itself. The RPL-INST-SRC transform replaces "data_in" as the source operand for the bus_in instruction with a masked source from the new port. The transformed instruction appears as shown in figure 5.14.

The transformations applied to the prototype by the Half_Word_to_Word_Input rule are summarized in figure 5.15.

```
{ INST1 } bus_in (data_port, 0, 7, qreg, 0, 7)

(DEPENDENCY (
 (INSTRUCTION ( 20, INST1 ) )
  (IS-CARRIED-OUT-BY ((controller ctrl, 20, INST2) )
                          (storage qreg, 10, INST1)
                          (storage qreg, 20, INST1) ))
  (CTRL (
          ((SRC   ctrl, 1, loadq)
           (DEST qreg, 1, loadq)
           (PATH c001)) ))
  (DATA-PATH (
                ((SRC   mult, 1, data_port)
                 (DEST qreg, 1, reg_in)
                 (PATH d$001)) )) ))
```

Figure 5.14 Prototype Statement 20, INST1 after Transformation.

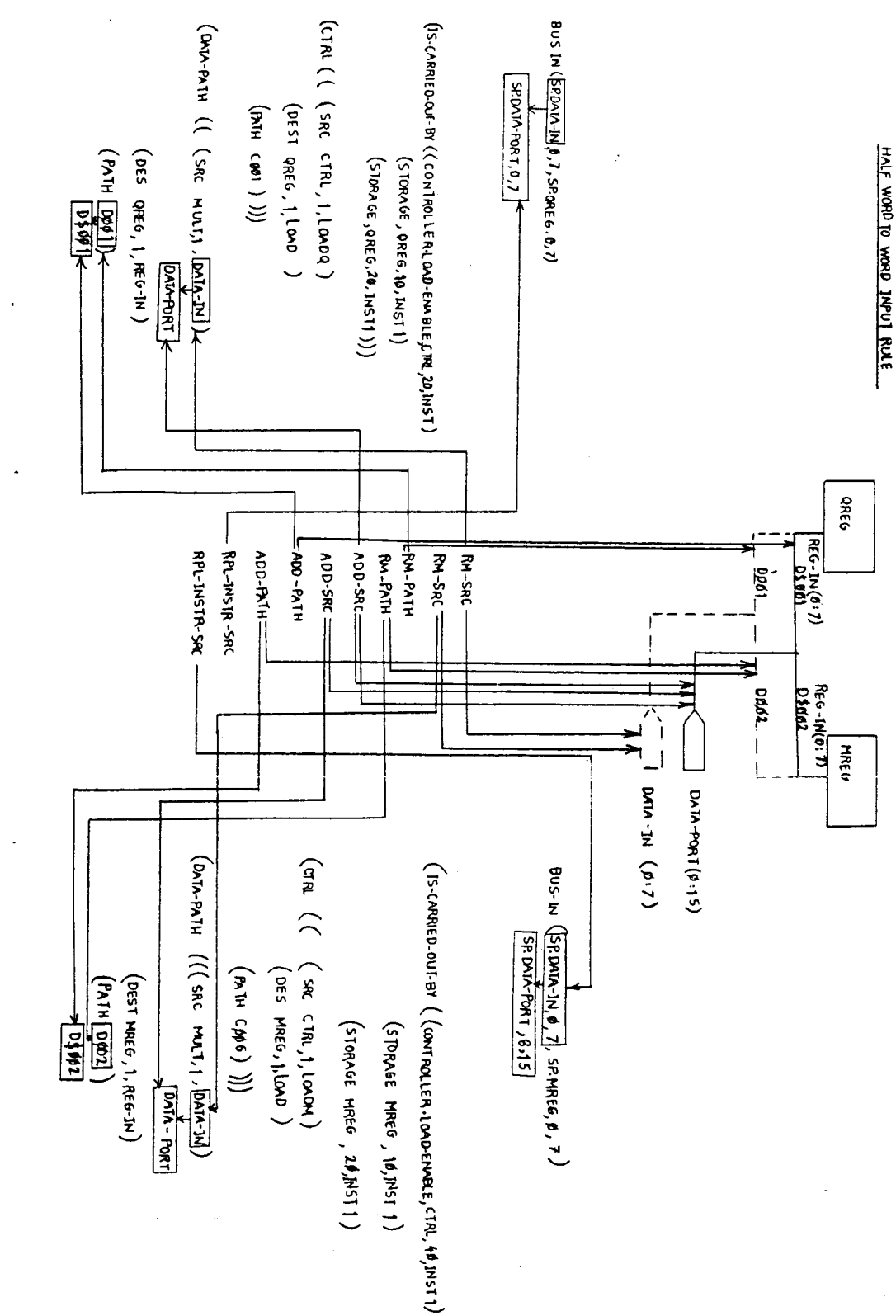


Figure 5.15 summary of transformations applied by HalfWordtoWordInput rule.

The application of the Half_Word_to_Word_Input rule sets up the Strong_Instr_Match rule which declares a strong match between prototype INST1 of statement 20

{ INST1 } BUS_IN (sp.data_port, 0, 7, sp.qreg, 0, 7)

and INST1 of user-spec statement 2

{ INST1 } BUS_IN (sp.data_port, 0, 7, multiplier, 0, 7)

A strong instruction match is also established between prototype statement 40 INST1

{ INST1 } BUS_IN (sp.data_port, 8, 15, sp.mreg, 0, 7)

and user-spec statement 2 INST4

{ INST1 } BUS_IN (sp.data_port, 8, 15, sp.multicand, 0, 7)

The bindings established by the application of these rules are summarized in figure 5.16.

The application of the Half_Word_to_Word_Input rule does not appear to have purchased much for us. Only two more strong instruction matches have been declared which means some of the differences between the user-spec and prototype have been reduced, however there are still sufficient differences between the two circuits to inhibit a full match. For example, while the prototype "qreg" and "mreg" registers have had their data paths reconstructed so as to resemble those of the user-spec, their control paths are different. The user-spec reads in its operands in parallel while the prototype reads in its operands sequentially. This difference in the control signals satisfies the conditional clauses of the Serial_Parallel_Input rule.

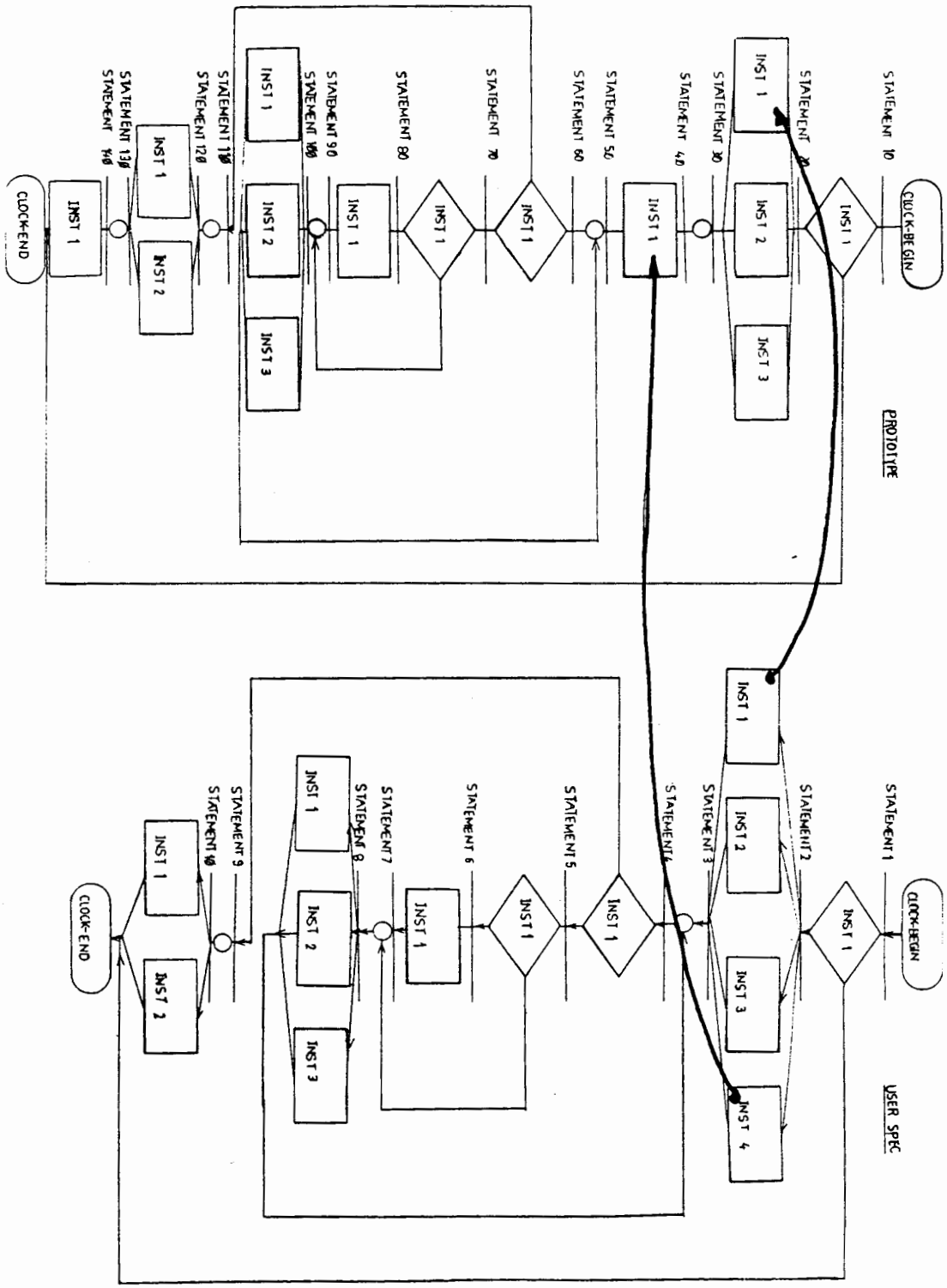


Figure 5.16 bindings established by design process

5.3.4 Increasing Parallelism in the Design

If two or more operations in a circuit are always performed together in the same clock cycle, then the components carrying out those operations can share the same control signal. The basis of the Serial_Parallel_Input rule is that serial operations can become parallel operations by having subcomponents share control signals.

The prototype registers "qreg" and "mreg" are loaded in two successive clock cycles because there was a space conflict for the input port "data_in" before the application of the Half_Word_to_Word_Input rule. To accommodate sequential operation, the "qreg" and "mreg" registers utilize different control signals. By merging the control signals we can force the registers to load in the same clock cycle.

The action item RM-CTRL-SRC, removes the "mreg" register's DEPENDENCY for the "loadm" port of "ctrl". The transform RM-CTRL-PATH, removes the DEPENDECY for layout command c006 which generated the path from the "loadm" port of "ctrl" to the "loadm" port of the "mreg" register.

After removing the "mreg" register's tie to the "loadm" signal from "ctrl", the next step is to link the "mreg" register to the "qreg" register's load signal, which is "loadq" from "ctrl". To rebuild the control path to the "mreg" register we must first obtain information about the "qreg" register's control path.

The action item ADD-CTRL-SRC takes the SRC identified in the CTRL path list for the prototype statement 20 INST1 and makes it the SRC also for the CTRL path of prototype statement 40 INST1. The transform ADD-CTRL-PATH adds a new layout command to the ARCHITECTURAL-PROPERTIES list for connecting the "loadq" output of "ctrl" to the "load" input of the "mreg" register. Both the "qreg"

and "mreg" registers now load off of the same control signal.

This modification of the circuit's architecture implies a need to alter the behaviour of supporting subcomponents such as "ctrl". The dependency pointers tagged as "controller" of the IS-CARRIED-OUT-BY pointer lists for both statement 20 INST1 and statement 40 INST1 identify the controller operations this instruction is dependent upon. The "controller" of the IS-CARRIED-OUT-BY pointer list slot shows that "ctrl" INST1 of statement 60

{ INST1 } signal_out (loadm, sim_$one)

must be executed in order for this instruction to be carried out. This is no longer true because the "mreg" register loads off of the "loadq" signal which is generated by "ctrl" INST2 of statement 20. The RM-CTRL-INST action item removes prototype statement 40 INST1's DEPENDENCY for "ctrl" statement 60 INST1. The ADD-CTRL-INST action item then adds a new "controller" pointer to statement 40 INST1's IS-CARRIED-OUT-BY pointer list which is the same as the "controller" pointer from the IS-CARRIED-OUT-BY pointer list of statement 20 INST1. The modified prototype instruction INST1 of statement 40 and its dependency record is shown in figure 5.17.

Since prototype INST1 of statement 40 will execute in parallel with INST1 of statement 20, it is move from statement 40 to statement 20. The transform MOVE-INST simply removes INST1 from the instruction list of statement 40 and appends it to the INSTRUCTION list of statement 20 and is renumbered as INST4. MOVE-INST also adjusts the symbol table name of the instruction so any bindings to the instruction are not lost. The transformations applied to the prototype by the Serial_Parallel rule are summarized in figure 5.18.

```
{ INST1 } bus_in (data_port, 8, 15, mreg, 0, 7)

(DEPENDENCY (
 (INSTRUCTION ( 20, INST1 ) )
  (IS-CARRIED-OUT-BY ((controller ctrl, 20, INST2) )
                      (storage mreg, 10, INST1)
                      (storage mreg, 20, INST1) ))
  (CTRL (
         ((SRC   ctrl, 1, loadq)
          (DEST mreg, 1, load)
          (PATH c$001)) ))
  (DATA-PATH (
              ((SRC   mult, 1, data_port)
               (DEST qreg, 1, reg_in)
               (PATH d$001)) )) ))
```

Figure 5.17 Statement 40, INST1 after Modification of the Controller.

5.3.5 Matching the New Input Structure

This adjustment of the input paths in the multiplier permits several new bindings to take place. The first is a weak statement match between user-spec statement 2 and the newly modified statement 20. The matching of these two statements now enables the strong matching of the user-spec statement 1 to prototype statement 10.

The strong statement match rule binds user-spec statement 1 to prototype statement 10. This rule fires because the preceding statement is strongly bound (by the Clock-Begin-Match rule) and all of the outcomes of the decision are at least weakly matched. In this case we have the clock-end statement as one outcome which has already been matched by the Clock-End-Match rule and statement 2 which we just

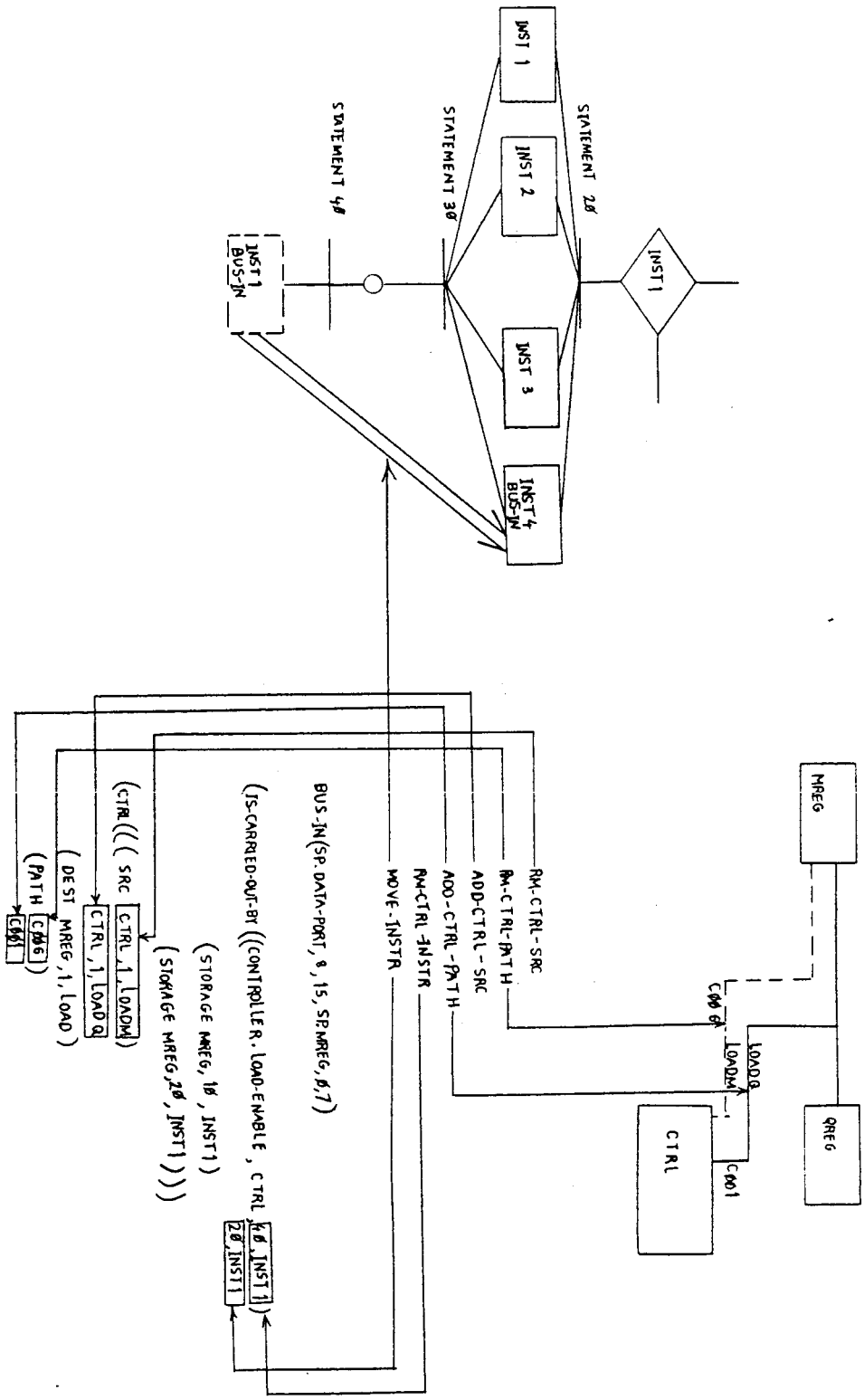


Figure 5.18 summary of the transformations applied by the serial-parallel rule

weakly matched. The new bindings that have been established between the user–spec and the prototype are summarized in figure 5.19.

Again, no more matches are possible because the prototype circuit writes its outputs halfword serially over two clock cycles and the user–spec writes its output word parallel in one clock cycle. The Half_Word_to_Word_Output rule recognizes this situation and fires.

5.3.6 Deserializing Output

The design represented by the prototype requires two clock cycles to write a 16 bit result. The behavioural requirements of the user–spec call for a design which can write a 16 bit result in one clock cycle. The width of the data that can be written out by the prototype in one clock cycle is constrained by the width of the output port "data_out". If a 16 bit output port was available within the prototype, then that port could be used to meet the behavioural requirements of the user–spec.

The prototype could take advantage of the 16 bit input–output port added to the prototype by the Input_Output_Port_Merge rule. The accumulator would then be able to write its 16 bit output in one clock cycle. The output of the accumulator should be tri–state such that it will not interfere with the "qreg" and "mreg" registers when they read their inputs from this port. The solution used here is to install a latch which has a tri–state capability between the "areg" and the "data_port" input–output port.

The first action item of the Half_Word_to_Word_Output rule is RM–INSTR which removes INST1 of statement 140. This is a BUS_OUT instruction

BUS_OUT (sp.data_out, 0, 7, sp.areg, 8, 15)

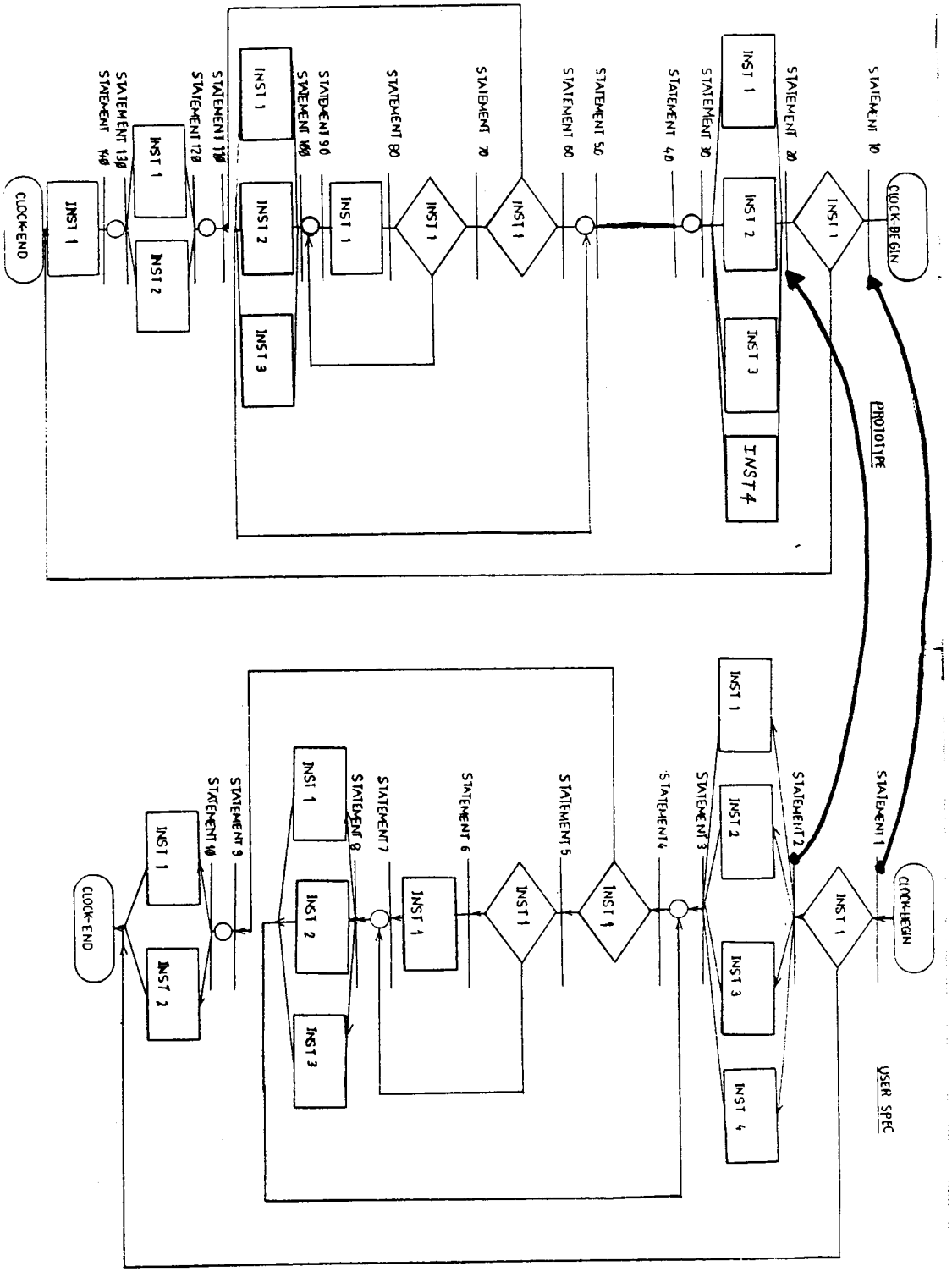


Figure 5.19 summary of the bindings established between the prototype and user-spec

which writes out the upper half of "areg". The removal of this instruction also causes its dependency record to be deleted from the dependency network. This removes the requirement this instruction had for the output multiplexer.

The second action item is the RM-DEST instruction which removes the destination dependency from the dependency record for INST1 of statement 120. This is a BUS_OUT instruction

BUS_OUT (sp.data_out, 0, 7, sp.areg, 0, 7)

which writes out the lower half of "areg". This causes the DEPENDENCY attribute for the multiplexer to fall to zero and therefore causes it to be removed from the architecture of the prototype multiplier.

The next action item is the RM-SRC instruction item which removes the source dependency from the dependency record of INST1 of statement 120. The following action item is ADD-SRC which re-installs the "areg"'s reg_out output as the source for this instruction. This removes the old source dependency for bits 0 through 7 of the "areg"'s reg_out output and replaces it with a dependency for the full output width of reg_out.

The next two action items are RPL-INSTR-SRC and RPL-INSTR-DEST which replace the source and destination operands of the instruction itself. After the application of these two action items, INSTR1 of statement 120 will appear as

BUS_OUT (sp.data_port, 0, 15, sp.areg, 0, 15)

where "data_port" was the port that was added by the Input_Output_Port_Merge rule.

The remainder of this rule adds statements to the ARCHITECTURAL-DEPENDENCIES list for the path from the "areg" register to the "data_port" input-output port. A check is made in this rule to determine what are

the requirements for the output path. If the destination port is a dedicated output port then a simple conductor path is created to tie the "areg" to the "data_port". However, as in this case, if the port is shared between input and output operations, then a tri-state buffer latch is inserted between the output source, the "areg", and output destination, the "data_port". The the path elements of the dependency record for this instruction will be updated by first linking the "areg" to the tri-state buffer latch, and then linking the latch to the "data_port".

The ADD-COMPONENT action item adds the requirements for the tri-state buffer latch to the prototype by creating a new DESIGN slot in the prototype frame to describe the behavioural requirements of the tri-state buffer latch. This new slot is called "buff" and is shown in figure 5.20. The latche's existence is justified by adding a ROUTER entry into the IS_CARRIED_OUT_BY list of the dependency record for INST1 of statement 120.

The next five action items add the latch to the DATA-PATH for INST1 of statement 120. The action item ADD-DEST creates an entry in the dependency record that sets the latche's "buff_in" port as the output destination of the "areg" register's "reg_out" port. The next two action items are ADD-SRC and ADD-DEST which sets the "buff_out" port of the latch as the output source for the "data_port" port. The two ADD-PATH action items add two layout commands to the ARCHITECTURAL-PROPERTIES list to create the paths linking the "areg" register to the latch and the latch to the "data_port" port. These new layout commands are shown in figure 5.21.

The latch requires an "out_enable" signal to pass the output of the "areg" register to the "data_port". This requires a signal from the controller to support the

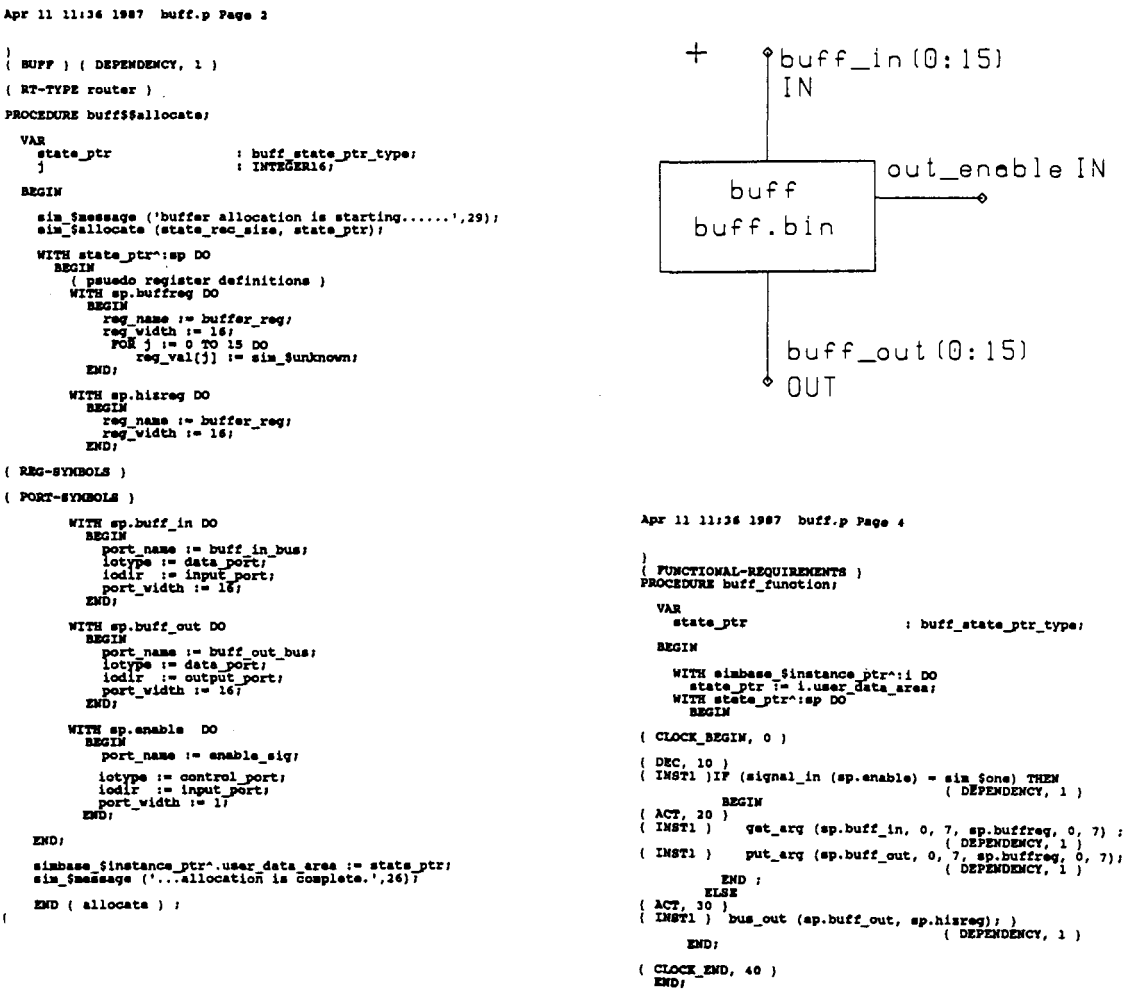


Figure 5.20 Buffer Latch Created for Multiplier.

behaviour of this instruction. The action item ADD-CTRL-SRC adds a SRC dependency to the CTRL list in the dependency record and adds that source to the behavioural requirements of the controller. The action item ADD-CTRL-DEST makes the latche's "out_enable" port the DEST of the CTRL list in the dependency record. The next action item ADD-CTRL-PATH adds a layout command to the ARCHITECTURAL-PROPERTIES list which links the control output of the controller

```
CONNECT (ctrl, out_enable, 1, buff, 1, out_enable, 1);

FOR i := 0 to 15 DO
     CONNECT (areg, 1, reg_out, i, buff, 1, buff_in, i);
FOR i := 0 to 15 DO
     CONNECT (buff, 1, buff_out, i, mult, 1, data_port, i);
```

Figure 5.21 Layout Commands for Data Path of Statement 120, INST1.

to the latch.

The final action item is ADD-CTRL-INSTR which adds an instruction to the behavioural requirements of the controller to drive the newly created signal path between the controller and the latch. The instruction

signal_out (out_enable, sim_$one)

is added to statement 120 in the controller. A controller entry is also added to the IS_CARRIED_OUT_BY list of the dependency record showing the requirement for this controller instruction. The new dependency record is shown in figure 5.22. Figure 5.23 summarizes the transformations that were applied to the prototype by the Half_Word_to_Word_Output rule and figure 5.24 shows a schematic for the modified architecture of the prototype.

5.3.7 Completing the Multiplier Design

On the next design cycle after the transformations the Port_Install rule fires and places the port symbol "data_port" into the symbol table. This causes the Strong_Port_Match rule to fire on the next cycle which strongly binds the user-spec "data_port" port to the prototype "data_port" port. The rules now fire in the following sequence:

1. The Strong_Statement_Match Rule strongly binds user-spec statement 4 to

```
(DEPENDENCY (
 (INSTRUCTION ( 120, INST1)
 (IS-CARRIED-OUT-BY ((controller ctrl, 130, INST2)
                     (controller ctrl, 130, INST3)
                     (controller ctrl, 30, INST1)
                     (controller ctrl, 50, INST1)
                     (controller ctrl, 130, INST4)
                     (controller counter, 30, INST1)
                     (controller counter, 40, INST1)
                     (controller counter, 50, INST1)
                     (storage areg, 80, INST1)
                     (router buff, 10, INST1)
                     (router buff, 20, INST1)
                     (router buff, 30, INST1) ))
(CTRL (
        ((SRC  ctrl, 1, outenable)
         (DEST buff, 1, outenable)
         (PATH c$021) ) ))
(DATA-PATH (
             ((SRC  areg, 1, regout)
              (DEST buff, 1, buffin)
              (PATH d$008))
             ((SRC  buff, 1, buffout)
              (DEST mult, 1, dataout)
              (PATH d$007)) ))  ))
```

Figure 5.22 New Dependency Record for INST1 of Statement 120.

prototype statement 60.

2. The Strong_Statement_Match rule strongly binds user-spec statement 5 to prototype statement 70. Note here that all clock-step statements are automatically assumed to be weakly bound.
3. The Strong_Statement_Match Rule strongly binds user-spec statement 6 to prototype statement 80.
4. The Strong_Statment_Match rule strongly binds user-spec clock-step statement 7 to prototype clock-step statement 90.
5. The Strong_Statement_Match rule stongly binds user-spec statement 8 to prototype statement 100.
6. The Strong_Statement_Match rule strongly binds user-spec clock-step statement 9 to prototype clock-step statement 110.
7. The Strong_Instruction_Match Rule strongly binds INST1 of user-spec statement 10

Figure 5.23 transformations applied to prototype by half-word-to-word-output rule

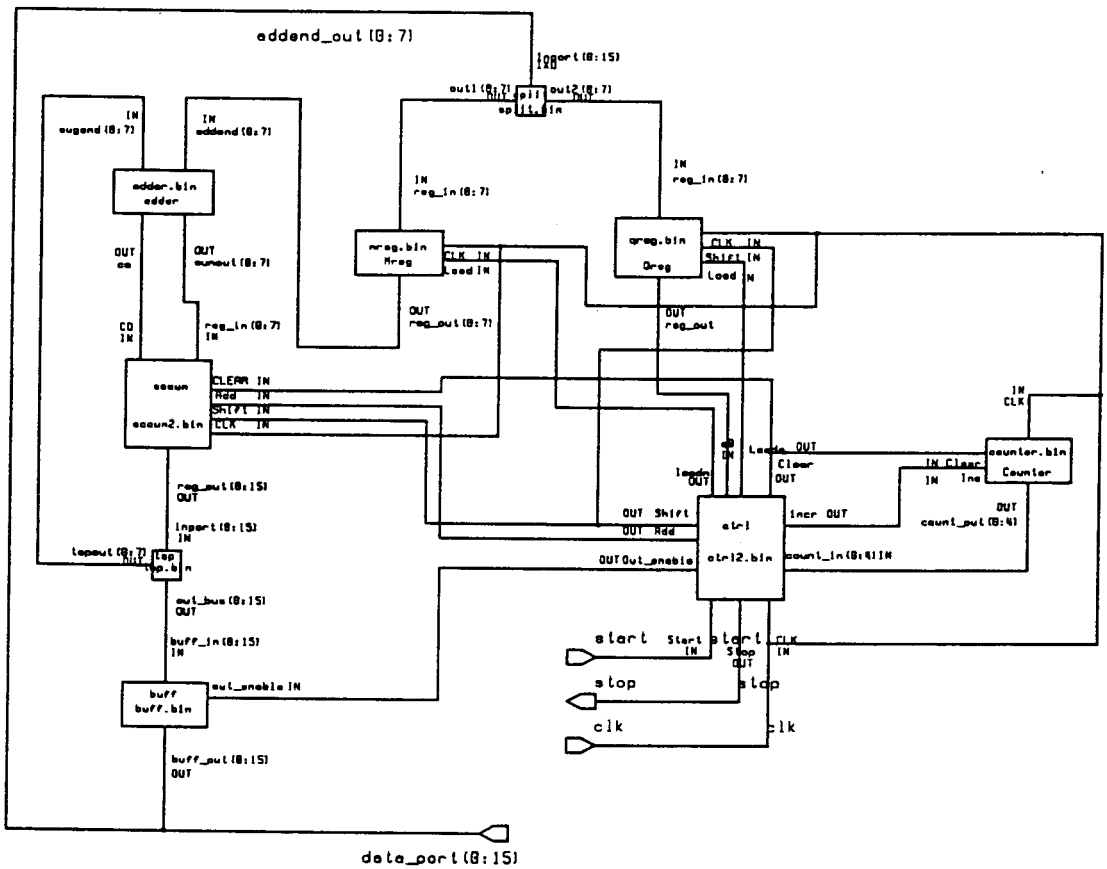


Figure 5.24 New Structure of Multiplier Circuit after Transforms

BUS_OUT (sp.data_port, 0, 15, sp.accumulator, 0, 15)

to INST1 of prototype statement 120

BUS_OUT (sp.data_port, 0, 15, sp.areg, 0, 15)

8. The Weak_Statement_Match Rule Weakly binds user-spec statement 10 to prototype statement 120.

9. The Strong_Statement_Match rule stongly binds user-spec statement 10 to

prototype statement 120.

With the firing of the last rule the two algorithms have been matched and we have a design which can implement the circuit desired by the user. The next pattern matcher cycle will find that no rule can be fired and that all user–spec statements are strongly matched and therefore a success is declared.

5.4 Expanding the Prototype

When the instantiator receives the DESIGN–COMPLETE message from the designer, it pushes the ARCHITECTURAL–PROPERTIES list of the design onto its stack and begins another design cycle.

This process will be repeated for each abstract component of the multiplier. This is how the frame inference mechanism of instantiation is used to support the top down design of digital circuits. This is how CIROP's [Res84] expansion of non terminals is implemented in this model.

5.5 An Example of a Designer Failure

If we were to assume that the prototype UMULT was unavailable then the designer would have used RDIV, a restoring divider, as a prototype. The schematic for RDIV as shown in figure 5.25 for the design of this circuit reveals a somewhat similar architecture to that of the unsigned multiplier. There are divider components which correspond to components in the unsigned multiplier. This correspondence is in fact born out by the behaviour of this circuit. All of the statements, port symbols and register symbols of the multiplier can be weak matched to the divider. This means

that the divider does provide features similar to those required to implement the multiplier. The divider reads values into registers from ports or writes values from registers to ports. The divider cycle is controlled by a counter and the divider also performs an accumulation function.

However few of these matches can be refined into strong matches and there does not exist within the rule base sufficient knowledge to modify the divider's design

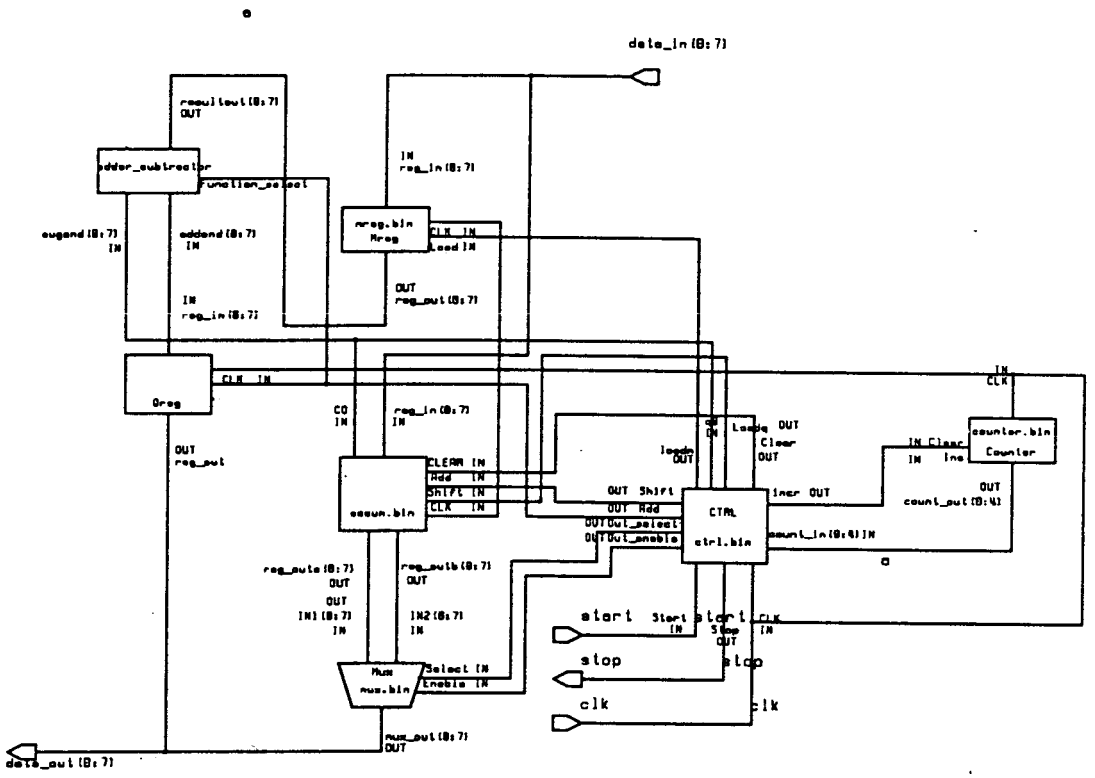


Figure 5.25 Schematic for a Restoring Divider.

such that it can satisfy the requirements of the unsigned multiplier. In short, the two circuits are not similar enough for the rules to be able to explain the differences between the circuits.

While our model cannot use this prototype to develop a design for the multiplier, it is interesting to note that an exercise appearing in Haye's [Hay84] textbook requires the student to develop a design for the divider using the multiplier as a prototype.

This example with the divider shows how the transformational rules can limit the capability of the proposed system. The transformational rules permit the system to use a similar circuit as prototype for a new circuit. If the rules are not powerful enough to recognize the similarities between the user–spec and the prototype then the design will fail, as was the case in this example with the divider.

The multiplier example given in this chapter uses the rule base defined in appendix three which was defined specifically to support this example. The rule base can be enhanced by adding more transformational rules to it such that it could be used to design of the multiplier using the divider as a prototype.

While adding more transformational rules to the rule base provides the system with more options for modifying circuits, it will also increase the complexity of the system. More rules in the rule base means more rules that have to be tested by the system before a suitable rule can be fired.

The addition of more transformational rules to the rule base would also permit the system to design new circuits from prototypes which would not normally be considered as being similar to the requested circuit. Taken to an extreme, with a large

enough rule base the system could create micro controllers from shift registers. This would violate the principle of using similar circuits as prototypes for the design of new circuits.

There is a limit on the number of transformational rules that can be added to the design system. If we regard the transformational rules as representing a unit of the fundemental knowledge of circuit design that a human circuit designer would possess, then this limit is intrinsic to the principle of using similar circuits as prototypes to guide new designs. If a human designer cannot recognize a set of circuit requirements as being similar to something that he has designed or has seen before, then the designer must draw more extensively on his fundemental knowledge of circuit design. However, if the designer knows of a similar circuit, then the knowledge the designer has of that circuit can be used to guide the design of the new circuit and reduce the designer's reliance on his fundemental knowledge of circuit design.

CHAPTER 6

SUMMARY AND CONCLUSION

This thesis has described a model system which can use existing circuits as prototypes to design new circuits. Prototypes were represented using frame structures which listed the necessary subcomponents, subcomponent behaviour, and interconnections required to implement a circuit behaviour. The frames are used as a production to build the circuit. A dependency network showed how each subcomponent and interconnection contributed to the circuit's behaviour.

Circuits are designed by first specifying an algorithm that describes the behaviour the circuit is required to implement. A search is made of the data base for a prototype which can implement behaviour. If an exact match is found then a circuit is considered to be an *off the shelf* part and the design is complete. The probability of this happening is low and most likely all that will be found in the data base will be prototypes that implement a similar behaviour. Therefore, the system has rules which can transform the behaviour of a similar prototype into one that can exactly implement the proposed behaviour.

Design proceeds in a top down depth first manner. Most prototypes represent abstract circuits which must themselves be refined into their constituent subcomponents. Design is complete when the algorithms specifying circuit behaviour are bound to primitive components.

The objective in this research was to show how the techniques used in CIROP might be used to design digital circuits. The assumption made was that digital circuits can have a more complicated behavioural description than that of operational-amplifiers

and therefore, maintaining the requirement of CIROP's pattern matcher that the capabilities of a candidate circuit must exactly match the requirements was too strong. This requirement was relaxed by permitting the use of similar circuits as design solutions. To support this, we created a set of rules which define a similar circuit and which could transform the behaviour of a circuit. To transform the behaviour of a circuit it was necessary to know how the behaviour was implemented by the circuit's architecture. This information was recorded in the dependency network.

Only part of a CIROP like system has been modelled in this thesis. The current model only views a circuit as an entity that implements a behaviour. There are no considerations made for technology and performance constraints.

6.1 Representing Technology Constraints

When technology constraints are added to a design then a designer is looking at the circuit from a different viewpoint. Throughout this thesis we have looked at a circuit as an architecture which implements a behaviour. We could also have modelled a circuit by using a timing diagram or by specifying its electrical requirements. We can therefore have multiple viewpoints of the same circuit.

Adopting a multiple viewpoint representation of a circuit opens up a Pandora's box of problems for the designer. Previously, the only prototype selection criteria used during the design of the multiplier was if the prototype implemented a behaviour that was similar to the required behaviour. Multiple viewpoints would require that a prototype be able to implement a behaviour subject to the constraints imposed by the other viewpoints. This would require a more sophisticated prototype selection mechanism than the one modelled here because the designer would have to deal with tradeoffs

between different features of the prototype.

A selection heuristic used by CIROP takes the form of the specification tradeoff. The differences between operational amplifiers are based on tradeoffs between their specifications. The priority of specifications for operational amplifiers will vary with the requirements for the operational amplifier. In some cases input bias will be of high importance while slew rate is not. The tradeoff made between input bias and slew rate will result in different operational amplifiers.

Similar tradeoffs can be made for digital circuits. For example, a fundamental tradeoff in any circuit design is the one between time and area. In general, faster circuits can be made at the expense of surface area. Another tradeoff occurs between time and power consumption.

A design process could start with the initial selection of prototypes which can implement the required behaviour. This selection phase is then followed by an analysis phase which uses the tradeoff specifications of the prototypes to determine if the circuit can meet the performance criteria. The transformational rules will become much more complicated because modifications to the circuit's behaviour by the removal or addition of a subcomponent will affect the circuit's performance.

The addition of performance criteria however can also shrink the size of the possible solution space. In the same way that the relative complexity is used to eliminate circuits which cannot obviously implement the required behaviour, heuristics could be used to eliminate circuits which obviously cannot meet the proposed circuit's performance requirements.

6.2 Recovery From Design Failure

In the current system model, when a design cannot be found in the data base the system has only two alternatives:

1. If the component which failed is a subcomponent of a higher level component then it may be possible to retract the design of that higher level component and see if there is an alternative design to the higher level component which does not require the subcomponent which cannot be designed.
2. If there is no higher level component then the design is declared a failure.

The first alternative is not realistic alternative because in a well designed data base all of the higher level component's subcomponents will be in the data base. The second alternative is not really an alternative.

In Peter Friedland's version of MOLGEN when the system failed to find a suitable plan for an experiment it looked for a closely related plan. Our system model already does this by looking for a prototype which can implement a behaviour similar to that of the proposed circuit. A second alternative proposed by Friedland was to look for a more general plan which could be refined into a specific plan.

In the current data base, the prototypes are at one level of abstraction. The only trace of a hierarchy comes in the form of the RT-TYPE which classifies circuits according to the specific register transfer behaviour they implement. Four RT-TYPEs, processor, storage, router, and controller were defined in this thesis because according to Hayes these constitute the minimal set of operations required to implement register transfer level circuits. These RT-TYPEs could be expanded into frames which could describe the *typical behaviour* of a circuit of that RT-TYPE. Then each of the RT-TYPEs could be expanded into the different abstract subclasses. For example,

processor could be expanded into arithmetic-processor, stored-program-processor, logical-processor, etc. Each of these subclasses would be represented by a frame describing the typical characteristics of an element in that class. The descendants of these classes would then be the frame types discussed in this thesis representing individual circuits.

If during design a prototype could not be found, the system could travel up to the next higher level of abstraction to find a more abstract prototype. The system would now require a library of refinement rules which could compare and refine the characteristics of the more abstract circuit to those of the requested circuit.

Such a system would require a more powerful planner because it would have to develop a circuit design strategy, knowing when to design a circuit using a similar circuit and when to use a more abstract design.

6.3 Side Effects of Transformations

Side effects which could be caused by the deletion of circuit subcomponents or interconnection is controlled by the DEPENDENCY network. No circuit feature is ever deleted outright until there is no more support for it within the DEPENDENCY network. The modification of existing subcomponents, interconnections, or subcomponent functions can lead to undesirable side effects if there is more than one DEPENDENCY for that circuit feature.

Consider the following example, during the transformation of the design of the unsigned multiplier, it was necessary add a tri-state output requirement to the accumulator. The solution implemented by the transformational rule was to install a

latch which had a tri-state capability between the accumulator and input-output port. This was only one possible solution to this design problem. An alternative solution would be to add a tri-state requirement to the FUNCTIONAL-REQUIREMENTS of the accumulator. The problem here, however, was that there is no efficient mechanism for determining the side effects caused by modifying a circuit function feature.

The frame representation for circuits presented in this thesis does not handle multiple dependencies well. For each instruction in the header of a frame, the subcomponent features and their interconnections that support the instruction are recorded. However, this is only a one way binding, the subcomponents do not know who uses them. For dealing with the side effects of a modification to a supporting feature, each supporting feature could have a pointers back to the header instruction which requires that features support. This again would require a more sophisticated planner than that presented here.

The planner would have to be capable of examining several alternate transformational rules and evaluating them within the context of their side effects. One model for this could be the Redesign system which generates a number of design modification alternatives and then evaluates the impact each alternative will have on the global circuit function. Redesign has a number of hueristics it uses to generate and select design alternatives.

6.4 Rule Design

The transformational rules demonstrated in this thesis often perform similar sets of functions on similar circuit structures. This results in many almost redundant transformational rules. This is particularly obvious between the

Half_Word_to_Word_Input and Half_Word_to_Word_Output rule. The reason for this was that it was required that when a rule transformed a circuit, the circuit must be left in a consistent state, that is, the header must still reflect the function the circuit structure implements. After a transformational rule is applied to a circuit, the circuit must still work, there cannot be any dead subcomponents or loose interconnections.

These criteria for rule design are too restrictive, and result in large coarse rules with many redundancies. Ideally we would like much more fine grained rules, which are more specific in their focus. The finer grain rules would have to leave state variables behind after their application such that they would forward chain in a consistent and useful manner.

Narrowing the focus of rules appears to improve the structure of the rule base. Early in this research very large coarse rules were tried for transforming designs with the results of the system becoming totally unwieldy. As the rules were refined, they became much more flexible. At some point there will be a minimum limit to how much knowledge must incorporated into a rule for that rule to be useful.

6.5 Notation

PASCAL was used to characterize this work because it was the language available under the Mentor Graphics design tools. The problem with using PASCAL to describe hardware behaviour is that it forces the user to combine the description of circuit's controller and data paths into one notation. This results in a description which is highly sensitive to the user's writing style. A data flow representation much like Snow's VT-diagrams would be more appropriate for representing the circuit data paths. Representation for control would remain a problem because of the necessity to assign

operations to specific clock periods.

6.6 The Final Word

It is very difficult to know when an expert type system is complete. In fact there is probably no satisfactory way in which any expert system can ever be declared complete. There is no theory which can prove the worst case complexity of the ill structured problems the system is designed to tackle. There is only the prospect that everytime the designers review their system they will find some feature that can be refined and improved. The system can never really be perfect. But then, like the personna from which it takes its name, it is not suppose to be, after all, it is an expert.

APPENDIX ONE - THE PROTOTYPE MULTIPLIER

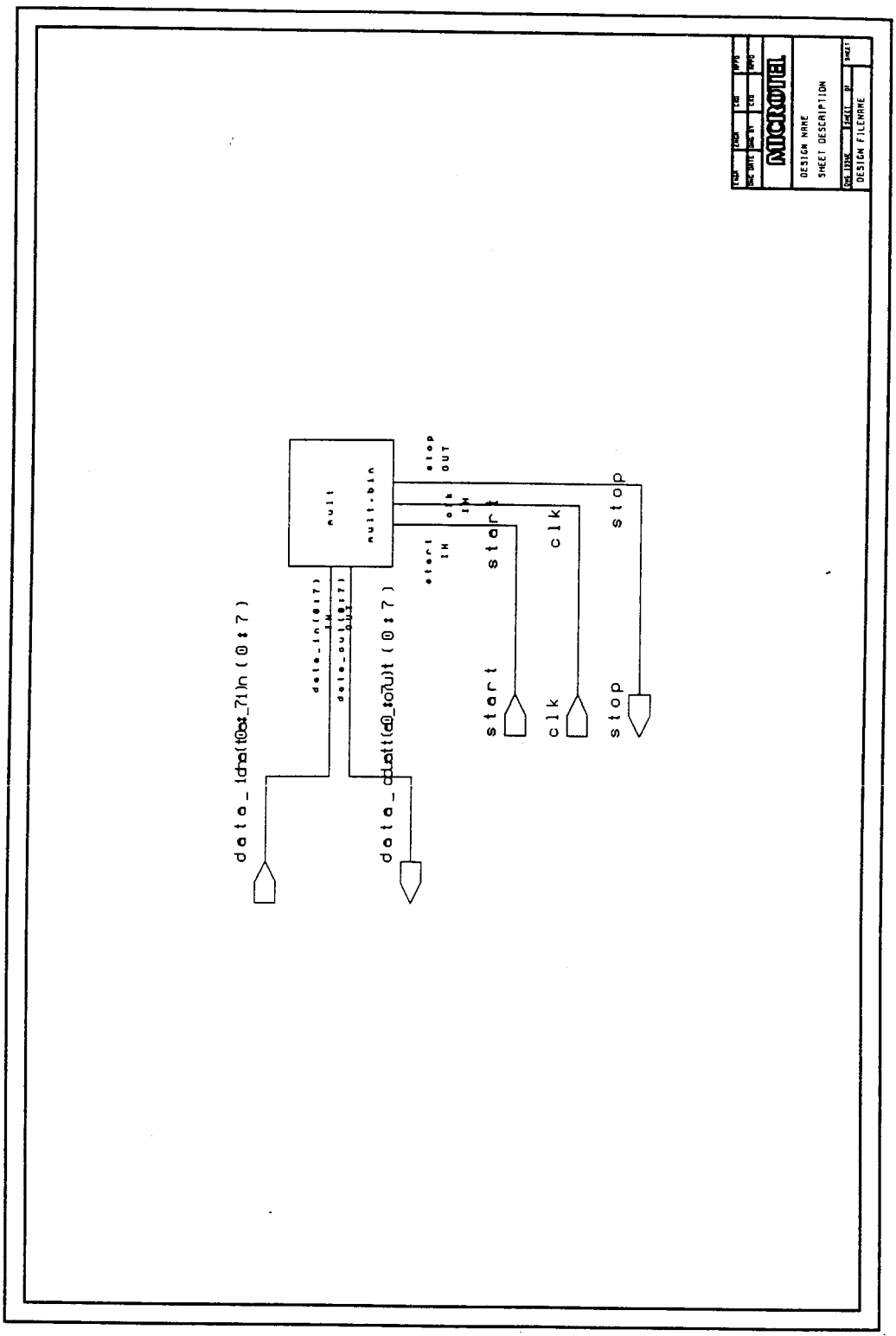
This appendix lists the set of behavioural models that were created to represent the prototype multiplier. These models have been organized into a prototype frame by augmenting them with an architectural properties list and a dependency network.

The contents of this appendix for the prototype multiplier are:

1. **The Frame Header** which includes the schematic for the multiplier and the program that describes the behaviour of the multiplier. The behaviour of this circuit is shown by the simulator traces included with it. The simulator traces are not part of the header.
2. **The Frame Body Design Slots** are the set of component models used to implement the multiplier behaviour described by the header. Each design slot consists of a component symbol and a listing of the program that implements the behaviour of the component.
3. **The Architectural-Properties List** includes the network schematic for the multiplier and the set of EXPAND, PLACE, and CONNECT commands which generate the network schematic. A simulator trace of the multiplier network is also included. The simulator traces are not part of the Architectural-Properties list.
4. **The Dependency Network** which shows how the architectural elements of the Design Slots and the Architectural-Properties list implement the multiplier behaviour described by the header.

Also included in this appendix are the elements of the multiplier frame that were modified to satisfy the user specifications given in the example of chapter five. These changes include:

1. A new schematic symbol for the multiplier.
2. The modified program describing the behaviour of the multiplier.
3. A simulator trace of the multiplier.
4. The modified design slot for the CTRL component.
5. The design slot that was added for the tri state buffer latch.
6. The modified network schematic for the multiplier.
7. A simulator trace of the modified multiplier network.



schematic for prototype multiplier

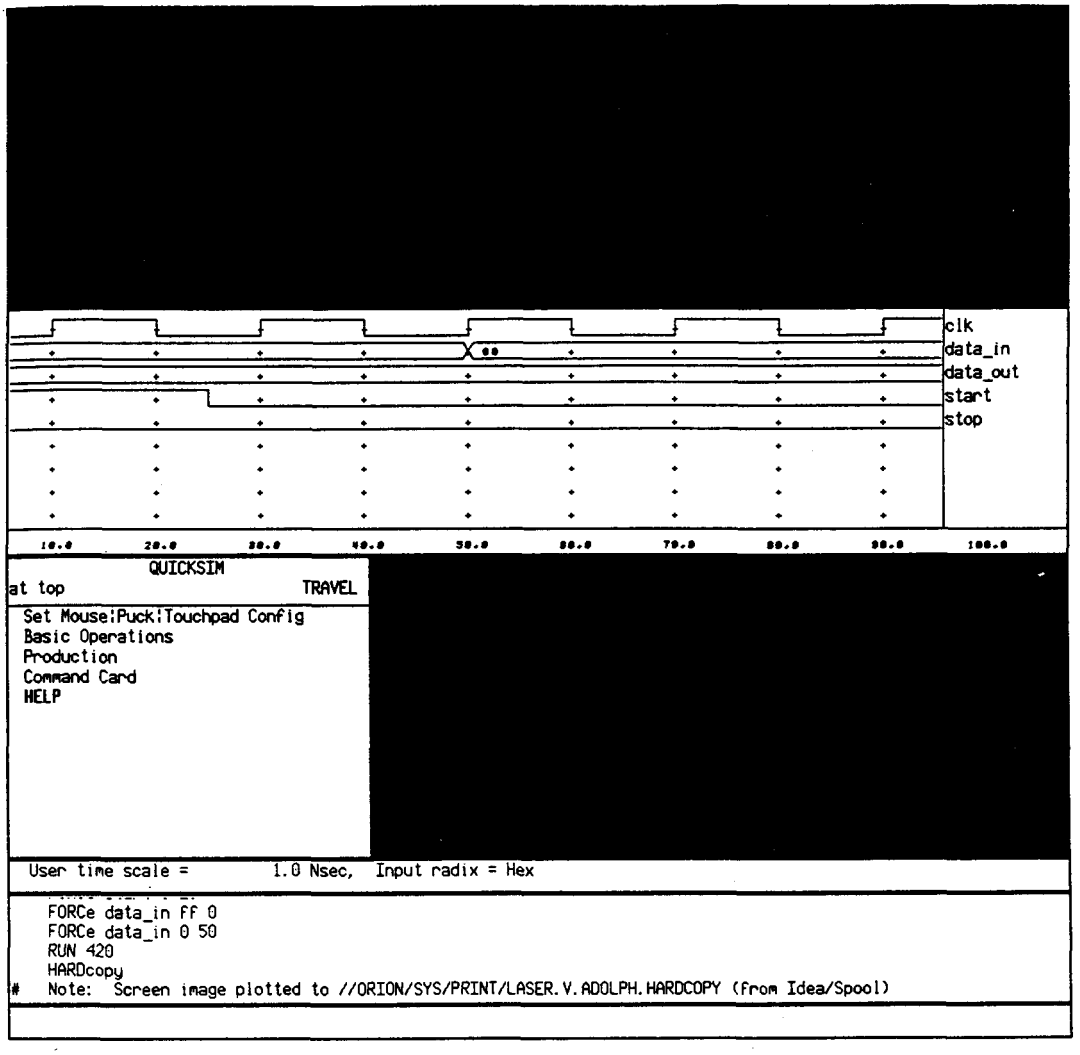
```
{ UMULT }
PROCEDURE mult$$allocate;
  VAR
    state_ptr                : mult_state_ptr_type;
    i                        : INTEGER16;

  BEGIN

    sim_$message ('allocation for mult is starting.....',45);
    sim_$allocate (state_rec_size, state_ptr);
    WITH state_ptr^:sp DO
      BEGIN
        sp.clock_cycle := clock1;
{ REG-SYMBOLS }
        WITH sp.counter DO
          BEGIN
            reg_name := counter_reg;
            reg_width := 5;
          END;
        WITH sp.accumulator DO
          BEGIN
            reg_name := accumulator_reg;
            reg_width := 16;
          END;
        WITH sp.multicand DO
          BEGIN
            reg_name := multicand_reg;
            reg_width := 8;
          END;
        WITH sp.multiplier DO
          BEGIN
            reg_name := multiplier_reg;
            reg_width := 8;
          END;
{ PORT-SYMBOLS }
        WITH sp.data_port DO
          BEGIN
            port_name := data_port_bus;
            iotype := data_port;
            iodir  := input_output_port;
            port_width := 16;
          END;
        WITH sp.start DO
          BEGIN
            port_name := start_sig;
            iotype := ctrl_port;
            iodir  := input_port;
            port_width := 1;
          END;
        WITH sp.stop  DO
          BEGIN
            port_name := stop_sig;
            iotype := ctrl_port;
            iodir  := output_port;
            port_width := 1;
          END;
        WITH sp.clk  DO
          BEGIN
            port_name := clk_sig;
```

```
            iotype := ctrl_port;
            iodir  := input_port;
            port_width := 1;
          END;
    END;

    simbase_$instance_ptr^.user_data_area := state_ptr;
    sim_$message ('...allocation is complete.',26);
    END { allocate } ;
```

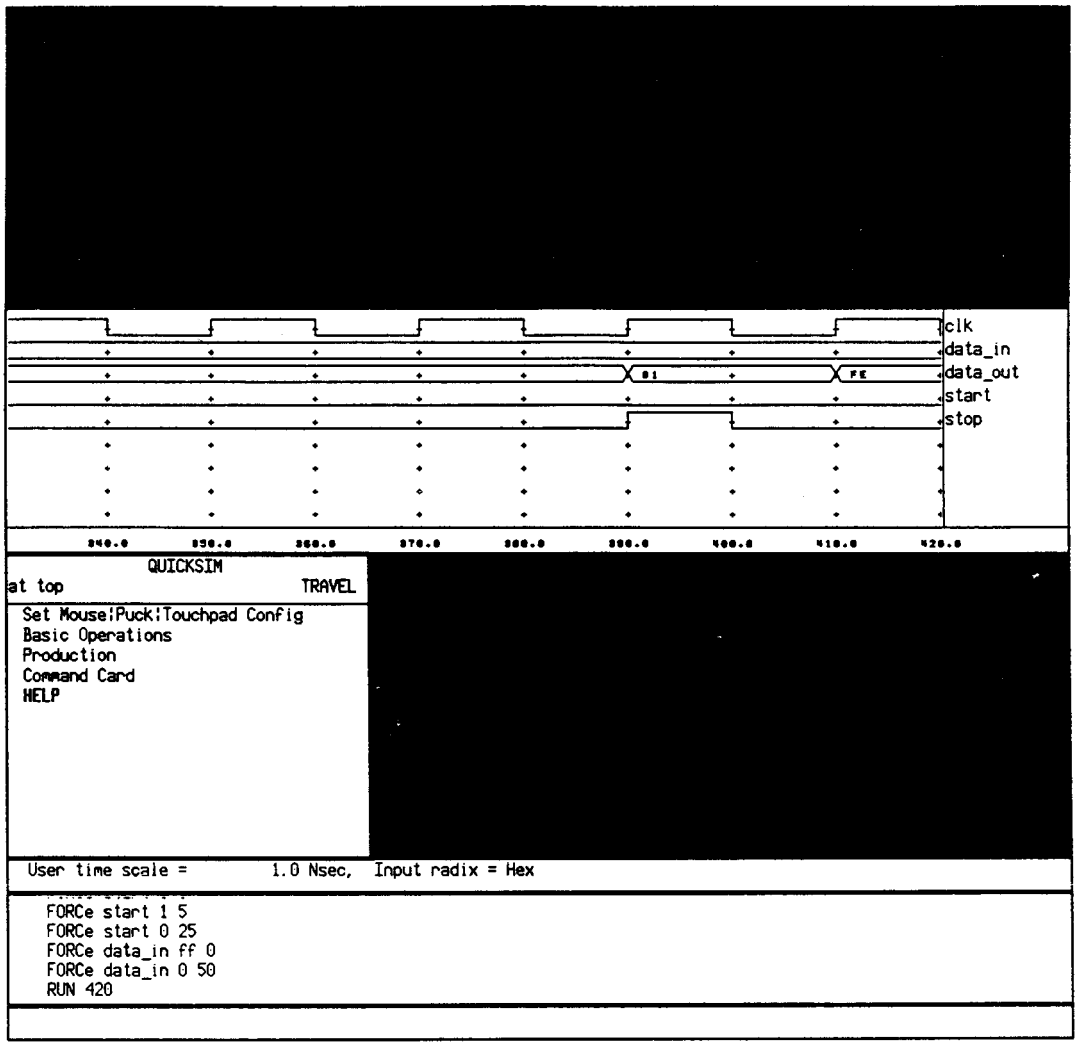
Behavioural model for user-spec multiplier

```
( FUNCTIONAL-CAPABILITIES )
PROCEDURE mult_function;
  VAR
    j                              : integer16;
    state_ptr                      : mult_state_ptr_type;
  BEGIN
    WITH simbase_$instance_ptr^:i DO
      state_ptr := i.user_data_area;
    WITH state_ptr^:sp DO
( CLOCK_BEGIN, 0 )
      BEGIN
( DEC, 10 )
( INST1 ) IF (signal_in (sp.start) = sim_$one) THEN
            BEGIN
( ACT, 20 )
(       1 )   bus_in (sp.data_in, 0, 7, sp.qreg, 0, 7);
(       2 )   init_reg  (sp.areg, sim_$zero);
(       3 )   init_reg  (sp.creg, sim_$zero);
              sp.clock_cycle := clock1;
            END
          ELSE CASE sp.clock_cycle OF
( CLOCK_STEP, 30 )
        clock1: BEGIN
( ACT, 40 )
( INST1 )   bus_in (sp.data_in, 0, 7, sp.mreg, 0, 7);
            sp.clock_cycle := clock1;
        END;
( CLOCK_STEP, 50 )
        clock1: BEGIN
( DEC, 60 )
( INST1 )   IF (decode_reg (sp.creg) < 8) THEN
              BEGIN
( DEC, 70 )
( INST1 )       IF (sp.qreg.reg_val[0] = sim_$one) THEN
( ACT, 80 )
                    add_reg ( sp.areg, 8, 15,
                              sp.mreg, 0, 7,
                              sp.areg, 8, 15,
                              sp.carry_out ) ;
                  sp.clock_cycle := clock2i ;
              END
            ELSE
              sp.clock_cycle := clockn1;
          END;
( CLOCK_STEP, 90 )
        clock2i: BEGIN
( ACT, 100 )
( INST1 )   shift_reg ( sp.areg, 0, 15, RIGHT, sp.carry_out);
( INST2 )   shift_reg ( sp.qreg, 0, 7,  RIGHT, sim_$zero);
( INST3 )   incr_reg ( sp.creg);
            sp.clock_cycle := clock1;
          END;
( CLOCK_STEP, 110 )
        clockn1: BEGIN
( ACT, 120 )
( INST1 )   bus_out (sp.data_out, 0, 7, sp.areg, 0, 7);
( INST2 )   signal_out (sp.stop, sim_$one);
            sp.clock_cycle := clockn;
        END;
```

```
( CLOCK_STEP, 130 )
        clockn: BEGIN
( ACT, 140 )
( INST1 )   bus_out (sp.data_out, 0, 7, sp.areg, 8, 15);
        END;
      END;
( CLOCK_END, 150 )
    END;
  END;
```



trace output for prototype multiplier

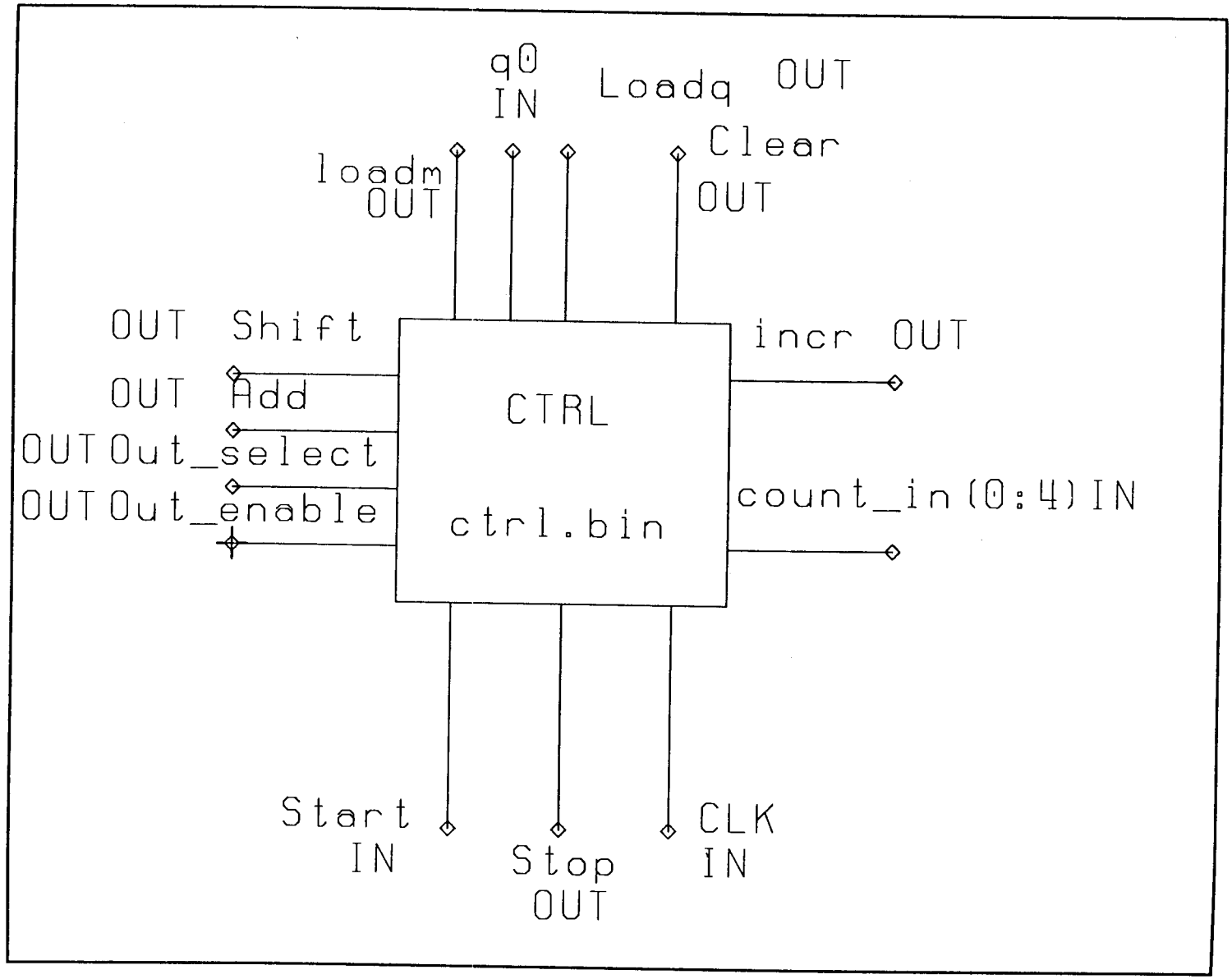

clk
data_in
data_out
start
stop
81
FE
340.0
350.0
360.0
370.0
380.0
390.0
400.0
410.0
420.0
QUICKSIM
at top
TRAVEL
Set Mouse!Puck!Touchpad Config
Basic Operations
Production
Command Card
HELP
User time scale = 1.0 Nsec, Input radix = Hex
FORCe start 1 5
FORCe start 0 25
FORCe data_in ff 0
FORCe data_in 0 50
RUN 420

augend(0:7)
IN
addend(0:7)
IN
adder
adder.bin
co
OUT
sumout(0:7)
OUT

```
{ ADDER } { DEPENDENCY, 1 }
{ RT-TYPE processor }
PROCEDURE adder$$allocate;
  VAR
    state_ptr                : adder_state_ptr_type;
    i                        : INTEGER16;

  BEGIN

    sim_$message ('allocation is starting.....',29);
    sim_$allocate (state_rec_size, state_ptr);
    WITH state_ptr^:sp DO
      BEGIN
        { psuedo register definitions }
        sp.reg1.reg_width := 8;
        sp.reg2.reg_width := 8;
        sp.reg3.reg_width := 8;
{ REG-SYMBOLS }
{ PORT-SYMBOLS }
        WITH sp.addend DO
          BEGIN
            port_name := addend_bus;
            iotype := data_port;
            iodir  := input_port;
            port_width := 8;
          END;
        WITH sp.augend DO
          BEGIN
            port_name := augend_bus;
            iotype := data_port;
            iodir  := input_port;
            port_width := 8;
          END;
        WITH sp.sumout  DO
          BEGIN
            port_name := sumout_bus;
            iotype := data_port;
            iodir  := output_port;
            port_width := 8;
          END;
        WITH sp.co  DO
          BEGIN
            port_name := co_sig;
            iotype := data_port;
            iodir  := output_port;
            port_width := 1;
            port_val[0] := sim_$zero;
          END;
        FOR i := 0 TO 15 DO
         BEGIN
          sp.augend.port_val[i] := sim_$zero;
          sp.addend.port_val[i] := sim_$zero;
          sp.sumout.port_val[i] := sim_$zero;
          sp.reg1.reg_val[i] := sim_$zero;
          sp.reg2.reg_val[i] := sim_$zero;
          sp.reg3.reg_val[i] := sim_$zero;
         END;
    END;
```

```
    simbase_$instance_ptr^.user_data_area := state_ptr;
    sim_$message ('...allocation is complete.',26);
    END{ allocate };
```

```
( FUNCTIONAL-REQUIREMENTS )
PROCEDURE adder_function;
  VAR
    state_ptr                    : adder_state_ptr_type;
    carry                        : INTEGER16;
  BEGIN
    WITH simbase_$instance_ptr^:i DO
      state_ptr := i.user_data_area;
    WITH state_ptr^:sp DO
      BEGIN
( CLOCK_BEGIN, 0 )
( ACT, 10 )
( INST1 ) get_arg (sp.addend, 0, 7, sp.reg1, 0, 7);
                                        ( DEPENDENCY, 1 )
( ACT, 20 )
( INST1 ) get_arg (sp.augend, 0, 7, sp.reg2, 0, 7);
                                        ( DEPENDENCY, 1 )
( ACT, 30 )
( INST1 ) add_reg ( sp.reg1, 0, 7,
                    sp.reg2, 0, 7,
                    sp.reg3, 0, 7,
                    carry );
                                        ( DEPENDENCY, 1 )
( ACT, 40 )
( INST1 ) put_arg (sp.sumout, 0, 7, sp.reg3, 0, 7);
                                        ( DEPENDENCY, 1 )
( ACT, 50 )
( INST1 ) signal_out ( sp.co, carry );
                                        ( DEPENDENCY, 1 )
      END;
( CLOCK_END, 60 )
  END;
```

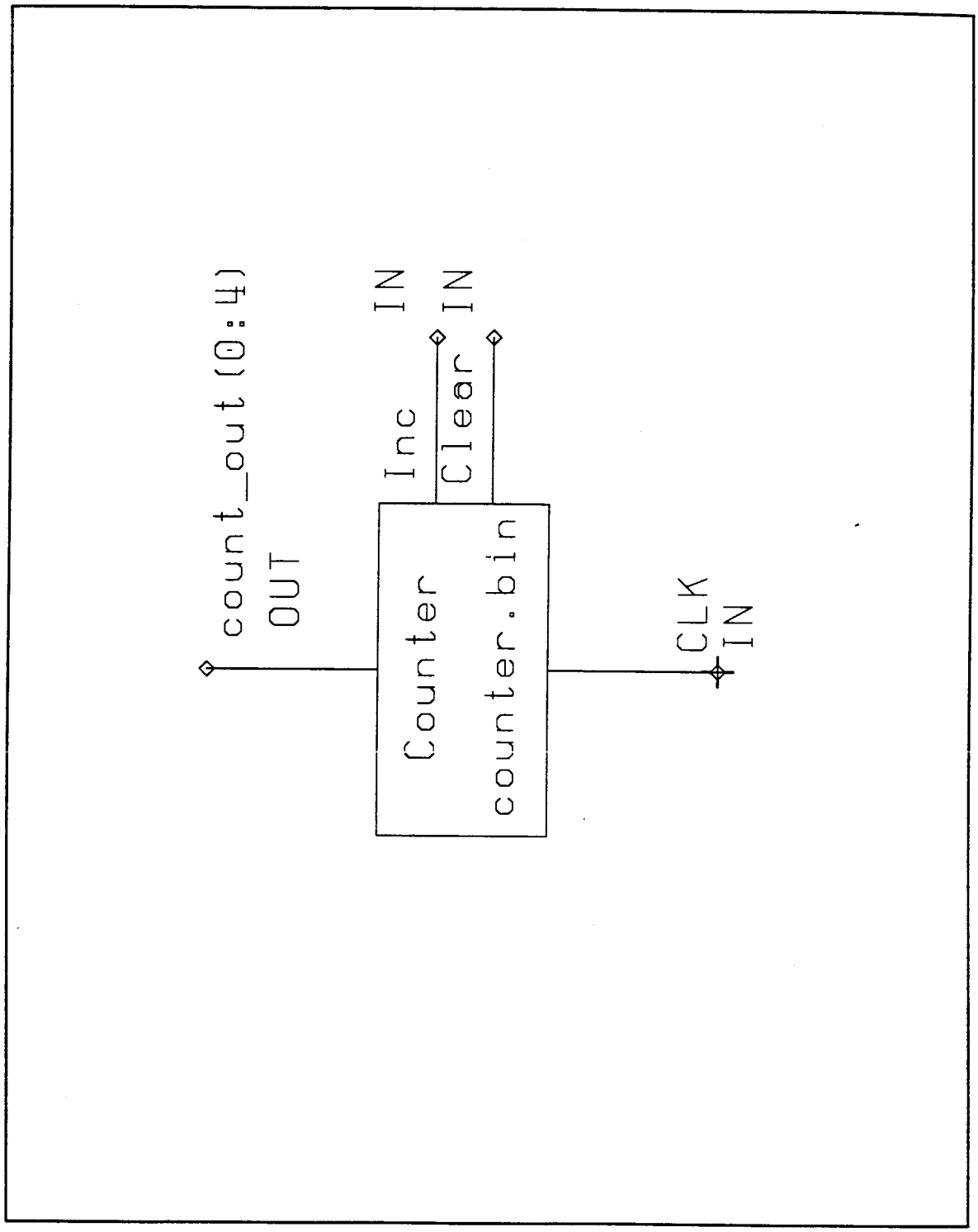

q0
IN
Loadq OUT
Clear
OUT
loadm
OUT
OUT Shift
OUT Add
OUT Out_select
OUT Out_enable
CTRL
ctrl.bin
incr OUT
count_in(0:4) IN
Start
IN
Stop
OUT
CLK
IN

```
{ CTRL } { DEPENDENCY, 1}
{ RT-TYPE controller }
PROCEDURE ctrl$$allocate;
  VAR
    state_ptr                : ctrl_state_ptr_type;
    i                        : INTEGER16;

  BEGIN

    sim_$message ('allocation for controller is starting.....',44);
    sim_$allocate (state_rec_size, state_ptr);
    WITH state_ptr↑:sp DO
      BEGIN
{ REG-SYMBOLS }
{ PORT-SYMBOLS }
        WITH sp.shift  DO
          BEGIN
            port_name := shift_sig;
            iotype := ctrl_port;
            iodir  := output_port;
            port_width := 1;
          END;
        WITH sp.add  DO
          BEGIN
            port_name := add_sig;
            iotype := ctrl_port;
            iodir  := output_port;
            port_width := 1;
          END;
        WITH sp.out_select  DO
          BEGIN
            port_name := out_select_sig;
            iotype := ctrl_port;
            iodir  := output_port;
            port_width := 1;
          END;
        WITH sp.out_enable  DO
          BEGIN
            port_name := out_enable_sig;
            iotype := ctrl_port;
            iodir  := output_port;
            port_width := 1;
          END;
        WITH sp.start  DO
          BEGIN
            port_name := start_sig;
            iotype := ctrl_port;
            iodir  := input_port;
            port_width := 1;
          END;
        WITH sp.stop  DO
          BEGIN
            port_name := stop_sig;
            iotype := ctrl_port;
            iodir  := output_port;
            port_width := 1;
          END;
        WITH sp.clk  DO
          BEGIN
```

```
            port_name := clk_sig;
            iotype := ctrl_port;
            iodir  := input_port;
            port_width := 1;
          END;
        WITH sp.count_in  DO
          BEGIN
            port_name := count_in_sig;
            iotype := ctrl_port;
            iodir  := input_port;
            port_width := 5;
          END;
        WITH sp.loadm  DO
          BEGIN
            port_name := loadm_sig;
            iotype := ctrl_port;
            iodir  := output_port;
            port_width := 1;
          END;
        WITH sp.clear  DO
          BEGIN
            port_name := clear_sig;
            iotype := ctrl_port;
            iodir  := output_port;
            port_width := 1;
          END;
        WITH sp.loadq  DO
          BEGIN
            port_name := loadq_sig;
            iotype := ctrl_port;
            iodir  := output_port;
            port_width := 1;
          END;
        WITH sp.incr  DO
          BEGIN
            port_name := incr_sig;
            iotype := ctrl_port;
            iodir  := output_port;
            port_width := 1;
          END;
        WITH sp.q0  DO
          BEGIN
            port_name := q0_sig;
            iotype := ctrl_port;
            iodir  := input_port;
            port_width := 1;
          END;
        WITH sp.count_state  DO
            reg_width := 5;
    END;

    simbase_$instance_ptr↑.user_data_area := state_ptr;
    sim_$message ('...allocation is complete.',26);
END { allocate } ;
```

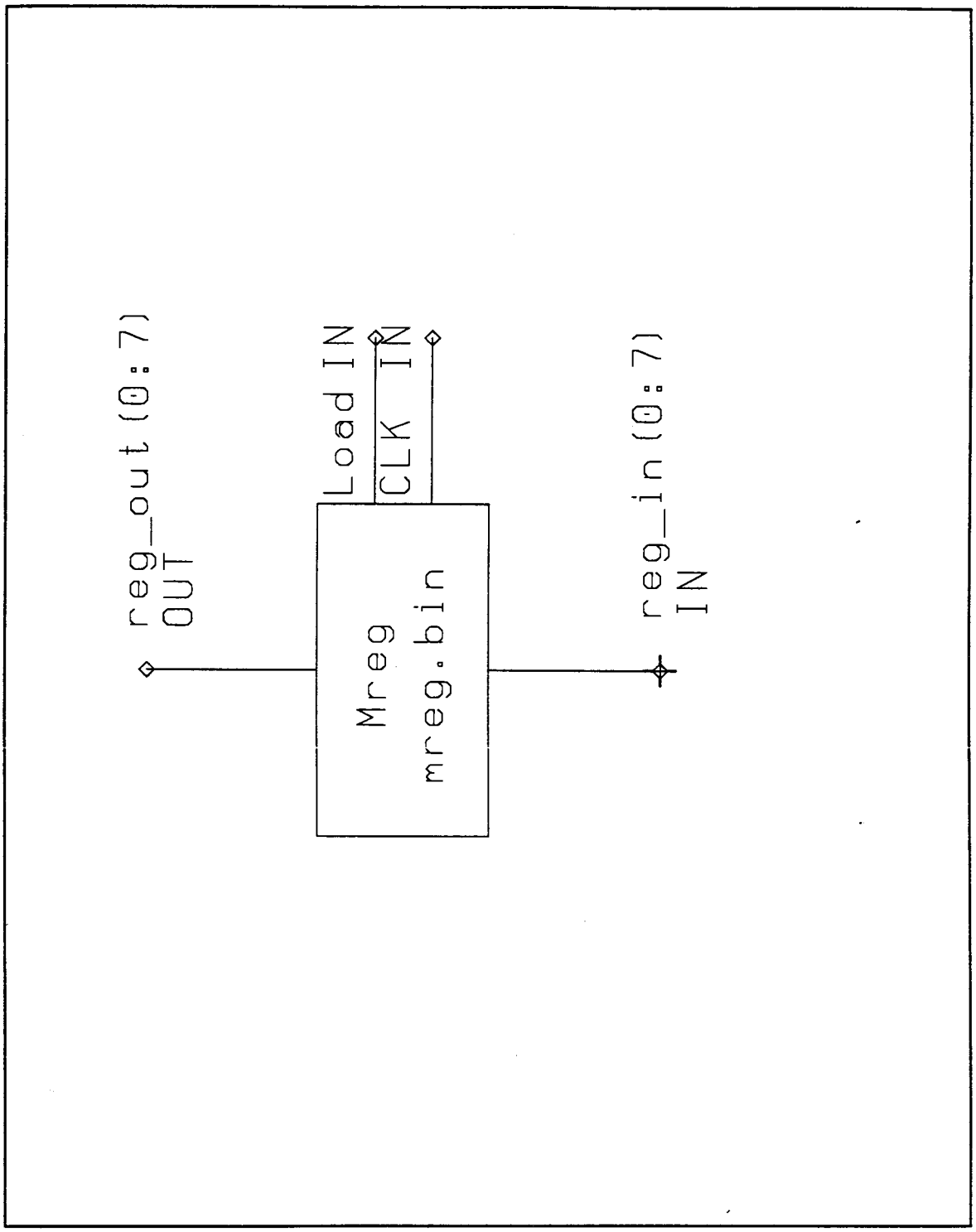
```
( FUNCTIONAL-REQUIREMENTS )
PROCEDURE ctrl_function;
  VAR
    state_ptr                    : ctrl_state_ptr_type;
    carry                        : INTEGER16;
  BEGIN
    WITH simbase_$instance_ptr^:1 DO
      state_ptr := 1.user_data_area;
    WITH state_ptr^:sp DO
      BEGIN

( CLOCK_BEGIN, 0 )
( DEC, 10 )
( INST1 ) IF (signal_in(sp.start) = sim_$one) THEN
                                 ( DEPENDENCY, 1 )
          BEGIN
( ACT, 20 )
( INST1 )   signal_out (sp.clear, sim_$one);
                                 ( DEPENDENCY, 2 )
( INST2 )   signal_out (sp.loadq, sim_$one);
                                 ( DEPENDENCY, 1 )
          END
        ELSE
( ACT, 30 )
( INST1 )   get_arg (sp.count_in, 0, 4, sp.count_state, 0, 4);
                                 ( DEPENDENCY, 7 )
( CLOCK_STEP, 40 )
( DEC, 50 )
( INST1 ) CASE decode_reg (sp.count_state) OF
                                 ( DEPENDENCY, 7 )
            0: BEGIN
( ACT, 60 )
( INST1 )       signal_out (sp.loadm, sim_$one);
                                 ( DEPENDENCY, 1 )
( INST2 )       signal_out (sp.incr, sim_$one);
                                 ( DEPENDENCY, 1 )
            END;
( CLOCK_STEP, 70 )
            1,3,5,7,9,11,13,15:
            BEGIN
( DEC, 80 )
( INST1 )       IF (signal_in (sp.q0) = sim_$one) THEN
                                 ( DEPENDENCY, 1 )
( ACT, 90 )
( INST1 )           signal_out (sp.add, sim_$one)
                                 ( DEPENDENCY, 1 )
                ELSE
( INST2 )           signal_out (sp.add, sim_$zero);
                                 ( DEPENDENCY, 1 )
( INST3 )           signal_out (sp.incr, sim_$one);
                                 ( DEPENDENCY, 1 )
            END;
( CLOCK_STEP, 100)
            2,4,6,8,10,12,14,16:
            BEGIN
( ACT, 110 )
( INST1 )       signal_out (sp.shift, sim_$one);
                                 ( DEPENDENCY, 1 )
( INST2 )       signal_out (sp.incr, sim_$one);
```

```
                                 ( DEPENDENCY, 1 )
            END;
( CLOCK_STEP, 120 )
          17: BEGIN
( ACT, 130 )
( INST1 )       signal_out (sp.stop, sim_$one);
                                 ( DEPENDENCY, 1 )
( INST2 )       signal_out (sp.out_enable, sim_$one);
                                 ( DEPENDENCY, 1 )
( INST3 )       signal_out (sp.out_select, sim_$zero);
                                 ( DEPENDENCY, 1 )
( INST4 )       signal_out (sp.incr, sim_$one);
                                 ( DEPENDENCY, 1 )
            END;
( CLOCK_STEP, 140 )

          18: BEGIN
( ACT, 150 )
( INST1 )       signal_out (sp.out_enable, sim_$one);
                                 ( DEPENDENCY, 1 )
( INST2 )       signal_out (sp.out_select, sim_$one);
                                 ( DEPENDENCY, 1 )
            END;
        END;
( CLOCK_END, 160 )
      END;
    END;
  END;
```


count_out(0:4)
OUT
Inc
IN
Clear
IN
Counter
counter.bin
CLK
IN

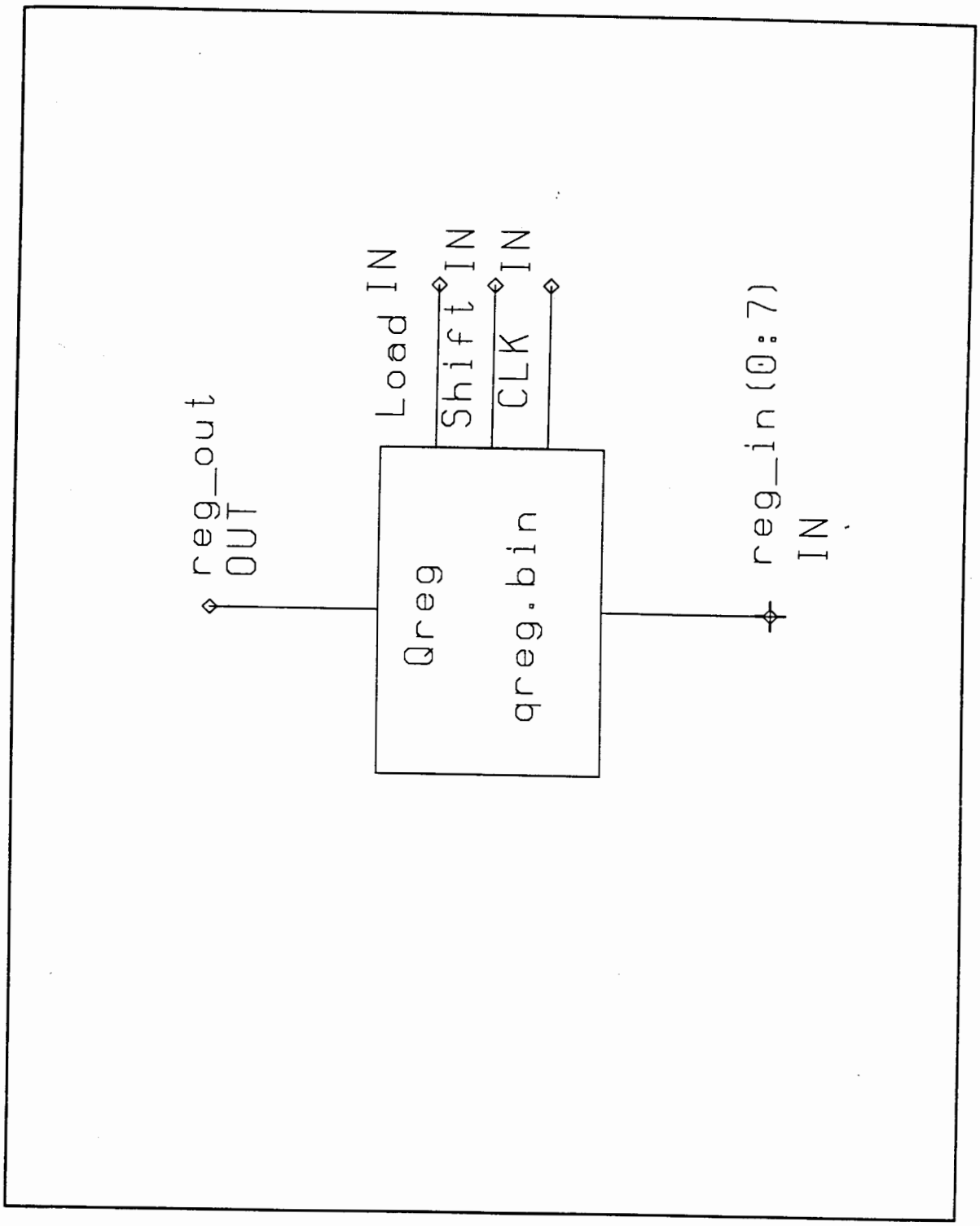
```
{ COUNTER } { DEPENDENCY, 1)
{ RT-TYPE controller }
PROCEDURE counter$$allocate;
  VAR
    state_ptr               : counter_state_ptr_type;
    i                       : INTEGER16;

  BEGIN

    sim_$message ('allocation for counter is starting.....',45);
    sim_$allocate (state_rec_size, state_ptr);
    WITH state_ptr↑:sp DO
      BEGIN
{ REG-SYMBOLS }
        WITH sp.count_reg DO
          BEGIN
            reg_name := count_reg_reg;
            reg_width := 5;
          END;
{ PORT-SYMBOLS }
        WITH sp.clear DO
          BEGIN
            port_name := clear_sig;
            iotype := ctrl_port;
            iodir  := input_port;
            port_width := 1;
          END;
        WITH sp.count_out DO
          BEGIN
            port_name := count_out_bus;
            iotype := data_port;
            iodir  := output_port;
            port_width := 5;
          END;
        WITH sp.inc  DO
          BEGIN
            port_name := inc_sig;
            iotype := ctrl_port;
            iodir  := input_port;
            port_width := 1;
          END;
        WITH sp.clk  DO
          BEGIN
            port_name := clk_sig;
            iotype := ctrl_port;
            iodir  := input_port;
            port_width := 1;
          END;
        FOR i := 0 TO 4 DO
         BEGIN
          sp.count_reg.reg_val[i] := sim_$zero;
         END;
    END;

    simbase_$instance_ptr↑.user_data_area := state_ptr;
    sim_$message ('...allocation is complete.',26);
    END { allocate } ;
```

```
{ FUNCTIONAL-REQUIREMENTS }
PROCEDURE counter_function;
  VAR
    state_ptr                   : counter_state_ptr_type;
  BEGIN
    WITH simbase_$instance_ptr↑:i DO
      state_ptr := i.user_data_area;
    WITH state_ptr↑:sp DO
      BEGIN
{ CLOCK_BEGIN, 0 }
{ DEC, 10 }
{ INST1 }  IF (signal_in (sp.clear) = sim_$one) THEN
                                { DEPENDENCY, 1}
{ ACT, 20 }
{ INST1 }     init_reg  (sp.count_reg, sim_$zero);
                                { DEPENDENCY, 1}
{ DEC, 30 }
{ INST1 }  IF (signal_in (sp.inc)= sim_$one) THEN
                                { DEPENDENCY, 7}
{ ACT, 40 }
{ INST1 }     incr_reg (sp.count_reg);
                                { DEPENDENCY, 7}
{ ACT, 50 }
{ INST1 }  bus_out (sp.count_out, 0, 4, sp.count_reg, 0, 4);
                                { DEPENDENCY, 7}
      END;
{ CLOCK_END, 60 }
  END;
```


reg_out(0:7)
OUT
Load IN
CLK IN
Mreg
mreg.bin
reg_in(0:7)
IN

```
{ MREG }  { DEPENDENCY, 1 }
PROCEDURE mreg$$allocate;
  VAR
    state_ptr                : mreg_state_ptr_type;
    i                        : INTEGER16;

  BEGIN

    sim_$message ('allocation is starting.....',29);
    sim_$allocate (state_rec_size, state_ptr);
    WITH state_ptr^:sp DO
      BEGIN
{ REG-SYMBOLS }
        WITH sp.m_reg DO
          BEGIN
            reg_name := mreg_reg;
            reg_width := 8;
          END;
{ PORT-SYMBOLS }
        WITH sp.reg_in DO
          BEGIN
            port_name := reg_in_bus;
            iotype := data_port;
            iodir  := input_port;
            port_width := 8;
          END;
        WITH sp.reg_out DO
          BEGIN
            port_name := reg_out_bus;
            iotype := data_port;
            iodir  := output_port;
            port_width := 8;
          END;
        WITH sp.load  DO
          BEGIN
            port_name := load_sig;
            iotype := control_port;
            iodir  := input_port;
            port_width := 1;
          END;
        WITH sp.clk  DO
          BEGIN
            port_name := clk_sig;
            iotype := control_port;
            iodir  := input_port;
            port_width := 1;
          END;
        FOR i := 0 TO 7 DO
          sp.m_reg.reg_val[i] := sim_$zero;
    END;

    simbase_$instance_ptr^.user_data_area := state_ptr;
    sim_$message ('...allocation is complete.',25);
    END { allocate } ;
```

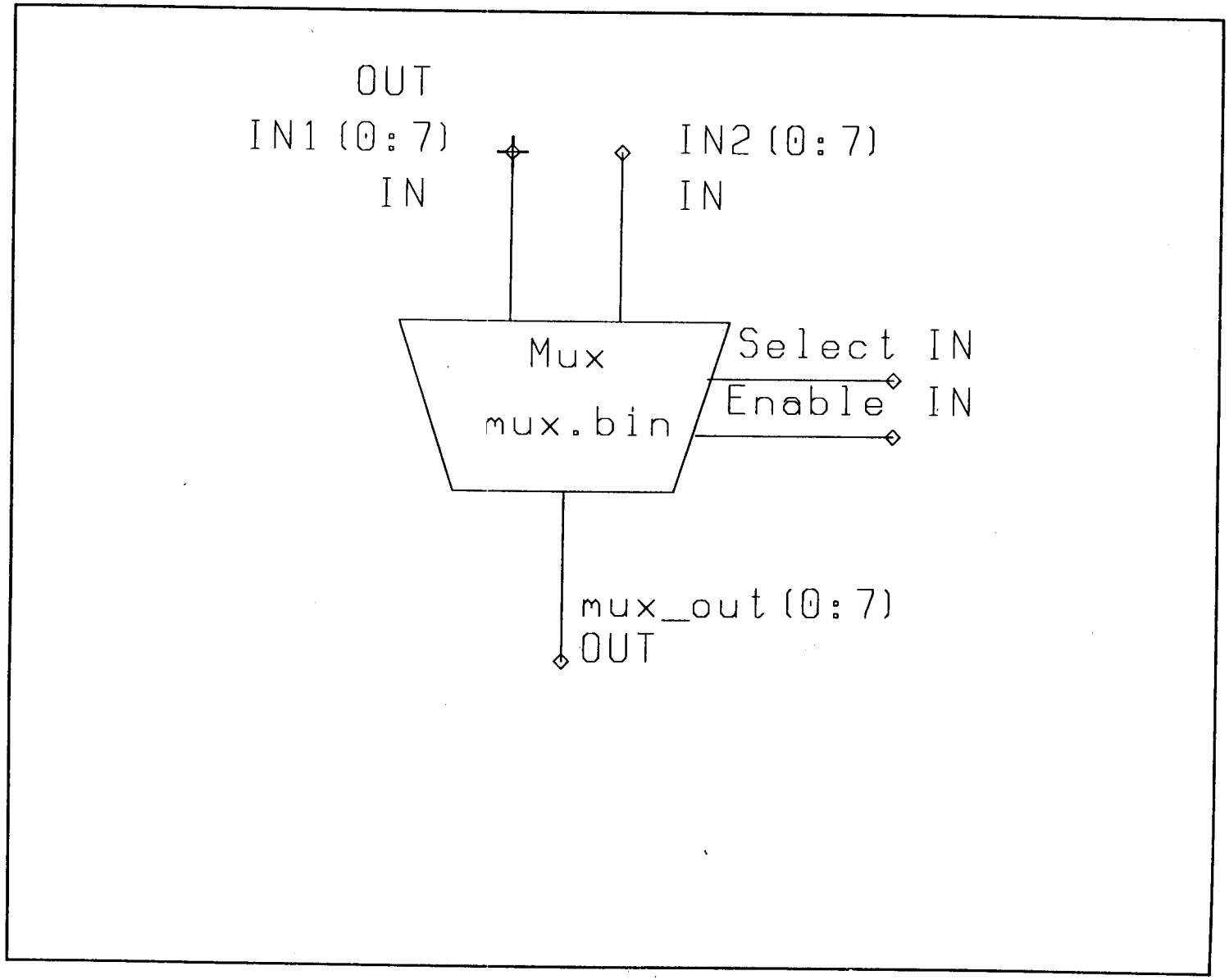
```
{ FUNCTIONAL-REQUIREMENTS }
PROCEDURE mreg_function;
  VAR
    state_ptr                : mreg_state_ptr_type;
  BEGIN
    WITH simbase_$instance_ptr^:i DO
      state_ptr := i.user_data_area;
    WITH state_ptr^:sp DO
      BEGIN
{ CLOCK_BEGIN, 0 }
{ DEC, 10 }
{ INST1 } IF (signal_in (sp.load) = sim_$one) THEN
                                              { DEPENDENCY, 1}
{ ACT, 20 }
{ INST1 }   bus_in (sp.reg_in, 0, 7, sp.m_reg, 0, 7) ;
                                              { DEPENDENCY, 1}
{ ACT, 3C }
{ INST1 } bus_out (sp.reg_out, 0, 7, sp.m_reg, 0, 7);
                                              { DEPENDENCY, 1}
      END;
{ CLOCK_END, 40 }
  END;
```


reg_out
OUT
Load IN
Shift IN
CLK IN
Qreg
qreg.bin
reg_in(0:7)
IN

```
{ QREG } { DEPENDENCY, 1 }
PROCEDURE Qreg$$allocate;
  VAR
    state_ptr                 : Qreg_state_ptr_type;
    i                         : INTEGER16;

  BEGIN

    sim_$message ('allocation is starting.....',29);
    sim_$allocate (state_rec_size, state_ptr);
    WITH state_ptr^:sp DO
      BEGIN
{ REG-SYMBOLS }

        WITH sp.qreg DO
          BEGIN
            reg_name := qreg_reg;
            reg_width := 8;
          END;
{ PORT-SYMBOLS }
        WITH sp.reg_in DO
          BEGIN
            port_name := reg_in_bus;
            iotype := data_port;
            iodir  := input_port;
            port_width := 8;
          END;
        WITH sp.reg_out DO
          BEGIN
            port_name := reg_out_bus;
            iotype := data_port;
            iodir  := output_port;
            port_width := 1;
          END;
        WITH sp.load  DO
          BEGIN
            port_name := load_sig;
            iotype := control_port;
            iodir  := input_port;
            port_width := 1;
          END;
        WITH sp.shift  DO
          BEGIN
            port_name := shift_sig;
            iotype := control_port;
            iodir  := input_port;
            port_width := 1;
          END;
        WITH sp.clk  DO
          BEGIN
            port_name := clk_sig;
            iotype := control_port;
            iodir  := input_port;
            port_width := 1;
          END;
        FOR i := 0 TO 7 DO
         BEGIN
          sp.reg_in.port_val[i] := sim_$zero;
          sp.reg_out.port_val[i] := sim_$zero;
```

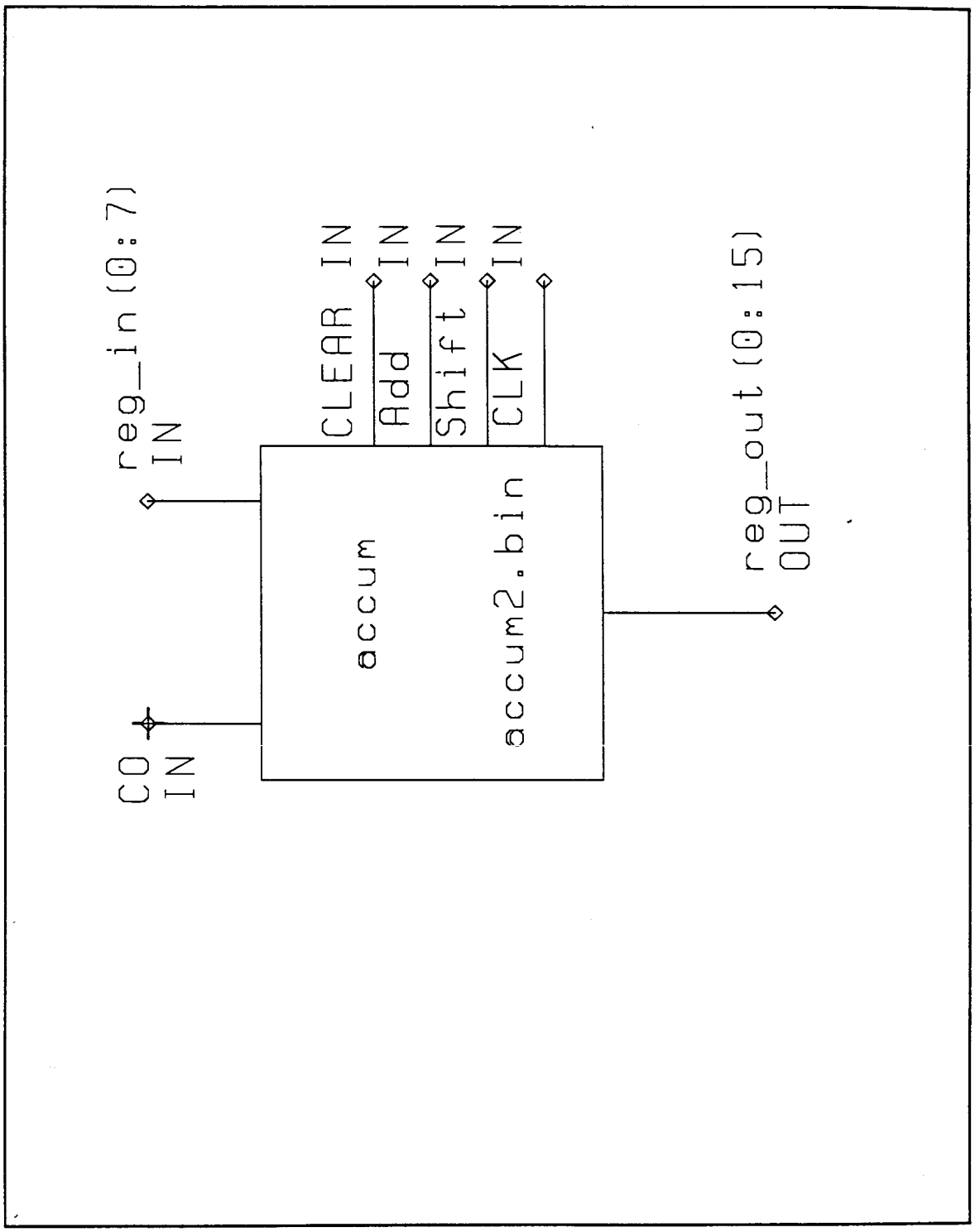
```
          sp.q_reg.reg_val[i] := sim_$zero;
        END;
    END;

    simbase_$instance_ptr^.user_data_area := state_ptr;
    sim_$message ('...allocation is complete.',26);
    END { allocate } ;
```

```
{ FUNCTIONAL-REQUIREMENTS }
PROCEDURE Qreg_function;
  VAR
    state_ptr                    : Qreg_state_ptr_type;
  BEGIN
    WITH simbase_$instance_ptr^.i DO
      state_ptr := i.user_data_area;
    WITH state_ptr^.sp DO
      BEGIN
{ CLOCK_BEGIN, 0 }
{ DEC, 10 }
{ INST1 }  IF (signal_in (sp.load) = sim_$one) THEN
                                                 { DEPENDENCY, 1 }
{ ACT, 20 }
{ INST1 }      bus_in (sp.reg_in, 0, 7, sp.q_reg, 0, 7)
                                                 { DEPENDENCY, 1 }
{ DEC, 30 }
{ INST1 }  ELSE IF (signal_in (sp.shift)= sim_$one) THEN
                                                 { DEPENDENCY, 1 }
{ ACT, 40 }
{ INST1 }      shift_reg (sp.q_reg, 0, 7, RIGHT, sim_$zero);
                                                 { DEPENDENCY, 1 }
{ ACT, 50 }
{ INST1 }  bus_out (sp.reg_out, 0, 7, sp.q_reg, 0, 7);
                                                 { DEPENDENCY, 1 }
      END;
{ CLOCK_END, 60 }
  END;
```


OUT
IN1(0:7)
IN
IN2(0:7)
IN
Mux
mux.bin
Select IN
Enable IN
mux_out(0:7)
OUT

```
{ MUX } { DEPENDENCY, 1 }
PROCEDURE mux$$allocate;
  VAR
    state_ptr                : mux_state_ptr_type;
    i                        : INTEGER16;

  BEGIN

    sim_$message ('allocation is starting.....',29);
    sim_$allocate (state_rec_size, state_ptr);
    WITH state_ptr↑:sp DO
      BEGIN
{ REG-SYMBOLS }
{ PORT-SYMBOLS }
        WITH sp.in1 DO
          BEGIN
            port_name := in1_bus;
            iotype := data_port;
            iodir  := input_port;
            port_width := 8;
          END;
        WITH sp.in2 DO
          BEGIN
            port_name := in2_bus;
            iotype := data_port;
            iodir  := input_port;
            port_width := 8;
          END;
        WITH sp.mux_out DO
          BEGIN
            port_name := mux_out_bus;
            iotype := data_port;
            iodir  := output_port;
            port_width := 8;
          END;
        WITH sp.select  DO
          BEGIN
            port_name := select_sig;
            iotype := control_port;
            iodir  := input_port;
            port_width := 1;
          END;
        WITH sp.enable  DO
          BEGIN
            port_name := enable_sig;
            iotype := control_port;
            iodir  := input_port;
            port_width := 1;
          END;
    END;

    simbase_$instance_ptr↑.user_data_area := state_ptr;
    sim_$message (' ..allocation is complete.',26);
    END { allocate } ;
```

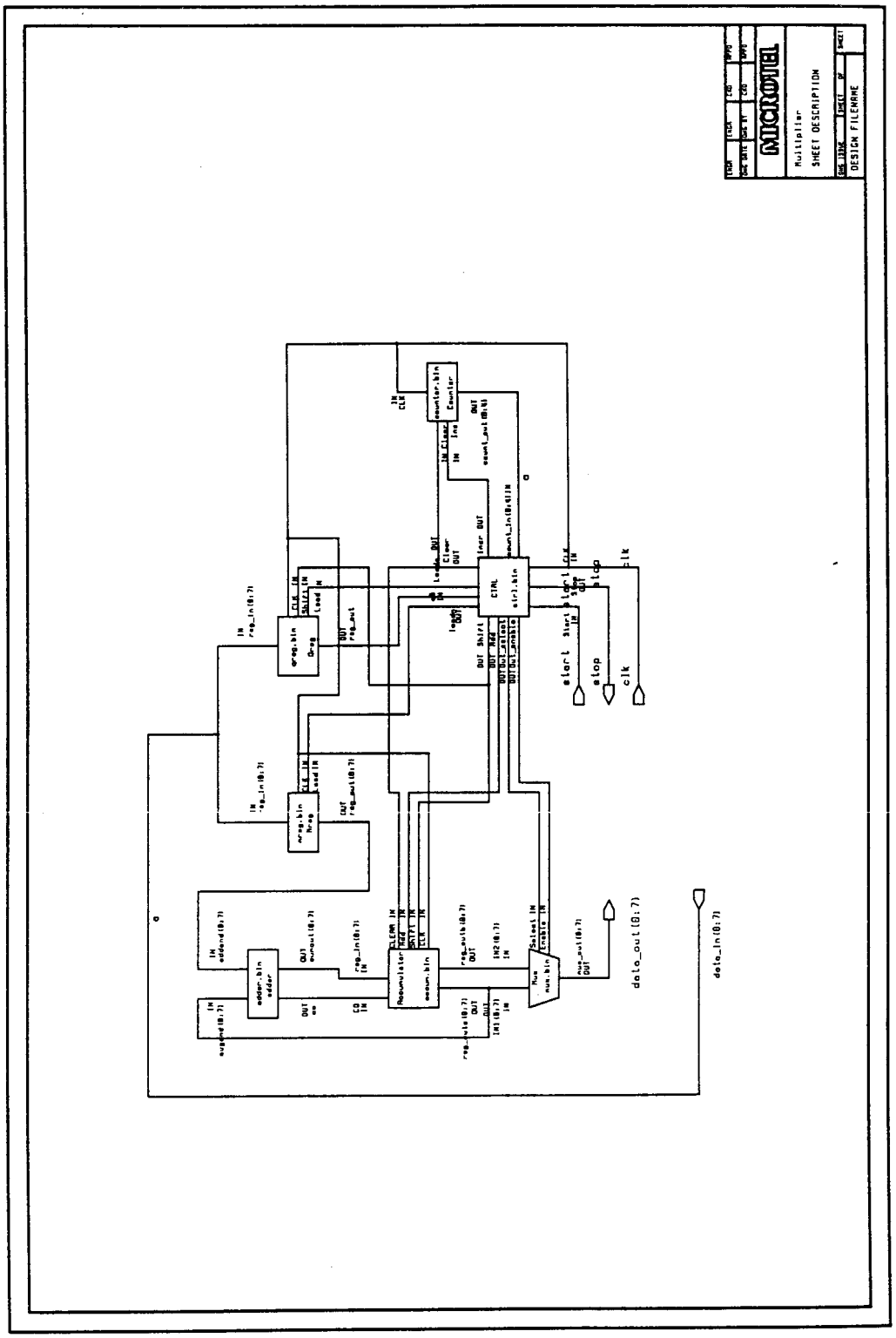
```
{ FUNCTIONAL-REQUIREMENTS }
PROCEDURE mux_function;
  VAR
    state_ptr                : mux_state_ptr_type;
  BEGIN
    WITH simbase_$instance_ptr↑:i DO
      state_ptr := i.user_data_area;
    WITH state_ptr↑:sp DO
      BEGIN
{ CLOCK_BEGIN, 0 }
{ DEC, 10 }
{ INST1 } IF (signal_in (sp.enable) = sim_$one) THEN
                                                { DEPENDENCY, 2 }
          BEGIN
{ DEC, 20 }
{ INST1 }    IF (signal_in (sp.select) = sim_$one) THEN
                                                { DEPENDENCY, 1 }
{ ACT, 30 }
{ INST1 }       get_arg (sp.in1, 0, 7, sp.muxreg, 0, 7) ;
                                                { DEPENDENCY, 1 }
{ DEC, 40 }
{ INST1 }    IF (signal_in (sp.select) = sim_$zero) THEN
                                                { DEPENDENCY, 1 }
{ ACT, 50 }
{ INST1 }       get_arg (sp.in2, 0, 7, sp.muxreg, 0, 7);
                                                { DEPENDENCY, 1 }
          END;
{ ACT, 60 }
{ INST1 } put_arg (sp.mux_out, 0, 7, sp.muxreg, 0, 7);
                                                { DEPENDENCY, 1 }
      END;
{ CLOCK_END, 70 }
  END;
```


reg_in(0:7)
IN
CLEAR IN
Add IN
Shift IN
CLK IN
reg_out(0:15)
OUT
accum
accum2.bin
CO
IN

```
{ AREG } { DEPENDENCY, 1)
{ RT-TYPE storage }
PROCEDURE accumulator$$allocate;
  VAR
    state_ptr                : accumulator_state_ptr_type;
    i                        : INTEGER16;

  BEGIN

    sim_$message ('allocation for the accumulator is starting.....',50);
    sim_$allocate (state_rec_size, state_ptr);
    WITH state_ptr↑:sp DO
      BEGIN
{ REG-SYMBOLS }

        WITH sp.a_reg DO
          BEGIN
            reg_name := a_reg_reg;
            a_reg.reg_width := 16;
          END;
{ PORT-SYMBOLS }
        WITH sp.reg_in DO
          BEGIN
            port_name := reg_in_bus;
            iotype := data_port;
            iodir  := input_port;
            port_width := 8;
          END;
        WITH sp.reg_out DO
          BEGIN
            port_name := reg_out_bus;
            iotype := data_port;
            iodir  := output_port;
            port_width := 8;
          END;
        WITH sp.co  DO
          BEGIN
            port_name := co_sig;
            iotype := data_port;
            iodir  := input_port;
            port_width := 1;
          END;
        WITH sp.add  DO
          BEGIN
            port_name := add_sig;
            iotype := control_port;
            iodir  := input_port;
            port_width := 1;
          END;
        WITH sp.shift  DO
          BEGIN
            port_name := shift_sig;
            iotype := control_port;
            iodir  := input_port;
            port_width := 1;
          END;
        WITH sp.clear  DO
          BEGIN
            port_name := clear_sig;
```

```
            iotype := control_port;
            iodir  := input_port;
            port_width := 1;
          END;
        WITH sp.clk  DO
          BEGIN
            port_name := clk_sig;
            iotype := control_port;
            iodir  := input_port;
            port_width := 1;
          END;
    END;

    simbase_$instance_ptr↑.user_data_area := state_ptr;
    sim_$message ('...allocation is complete.',26);
    END { allocate } ;
```

```
{ FUNCTIONAL-REQUIREMENTS }
PROCEDURE accumulator_function;
  VAR
    state_ptr                    : accumulator_state_ptr_type;
  BEGIN
    WITH simbase_$instance_ptr^:i DO
      state_ptr := i.user_data_area;
    WITH state_ptr^:sp DO
      BEGIN
{ CLOCK_BEGIN, 0 }
{ DEC, 10 }
{ INST1 } IF (signal_in (sp.clear) = sim_$one) THEN
                                  { DEPENDENCY, 1 }
{ ACT, 20 }
{ INST1 }   init_reg (sp.a_reg, sim_$zero)
                                  { DEPENDENCY, 1 }
        ELSE
          BEGIN
{ DEC, 30 }
{ INST1 }   IF (signal_in (sp.add) = sim_$one) THEN
                                  { DEPENDENCY, 1 }
              BEGIN
{ ACT, 40 }
{ INST1 }       sp.carry_in := signal_in (sp.co);
                                  { DEPENDENCY, 1 }
{ INST2 }       bus_in (sp.reg_in, 0, 7, sp.a_reg, 0, 7);
                                  { DEPENDENCY, 1 }
              END;
{ DEC, 50 }
                                  { DEPENDENCY, 1 }
{ INST1 }   IF (signal_in (sp.shift) = sim_$one) THEN
                                  { DEPENDENCY, 1 }
{ ACT, 60 }
{ INST1 }     shift_reg (sp.a_reg, 0, 15, RIGHT, sp.carry_in);
                                  { DEPENDENCY, 1 }
            END;
{ ACT, 70 }
{ INST1 } bus_out (sp.reg_out, 0, 15, sp.a_reg, 0, 15);
                                  { DEPENDENCY, 3 }
      END;
{ CLOCK_END, 80 }
  END;
```

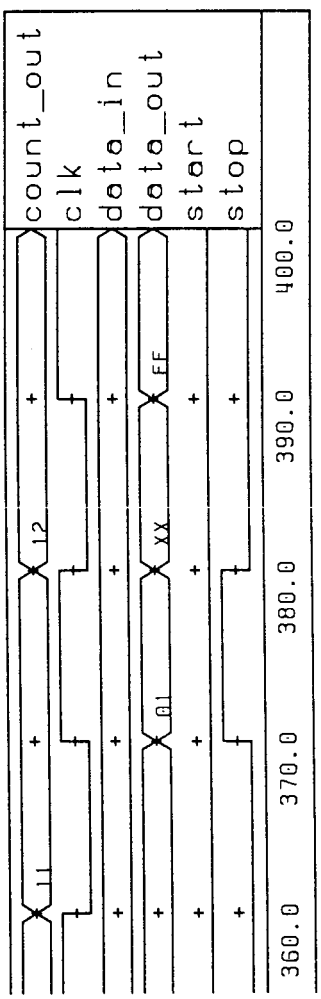


Network schematic for prototype multiplier

```
(ARCHITECTURAL_PROPERTIES (

e001 : EXPAND ( areg );      {DEPENDENCY, 1}
e002 : EXPAND ( qreg );      {DEPENDENCY, 1}
e003 : EXPAND ( counter );   {DEPENDENCY, 1}
e004 : EXPAND ( adder );     {DEPENDENCY, 1}
e005 : EXPAND ( mux );       {DEPENDENCY, 1}
e006 : EXPAND ( mreg );      {DEPENDENCY, 1}
e007 : EXPAND ( ctrl );      {DEPENDENCY, 1}
p001 : PLACE ( areg, areg, 1 );          {DEPENDENCY, 1}
p002 : PLACE ( qreg, qreg, 1 );          {DEPENDENCY, 1}
p003 : PLACE ( counter, counter, 1 );    {DEPENDENCY, 1}
p004 : PLACE ( adder, adder, 1 );        {DEPENDENCY, 1}
p005 : PLACE ( mux, mux, 1 );            {DEPENDENCY, 1}
p006 : PLACE ( mreg, mreg, 1 );          {DEPENDENCY, 1}
p007 : PLACE ( ctrl, ctrl, 1 );          {DEPENDENCY, 1}
c001 : CONNECT ( qreg, 1, load, 1, ctrl, 1, loadq, 1);  {DEPENDENCY, 1}

c002 : CONNECT ( areg, 1, clear, 1, ctrl, 1, clear, 1); {DEPENDENCY, 1}

d001 : FOR i := 0 TO 7 DO
          CONNECT ( qreg, 1, reg_in, i, mult, 1, data_in, i );
                                                  {DEPENDENCY, 1}

d002 : FOR i := 0 TO 7 DO
          CONNECT ( areg, 1, reg_in, i, mult, 1, data_in, i );
                                                  {DEPENDENCY, 1}

c005 : CONNECT ( counter, 1, clear, 1, ctrl, 1, clear, 1); {DEPENDENCY, 1}

c006 : CONNECT ( mreg, 1, load, 1, ctrl, 1, loadm, 1);  {DEPENDENCY, 1}

c007 : FOR i := 0 TO 3 DO
          CONNECT ( counter, 1, count_out, i,
                    ctrl, 1, count_in, i );           {DEPENDENCY, 1}

c008 : CONNECT ( qreg, 1, reg_out, 1, ctrl, 1, q0, 1);  {DEPENDENCY, 1}

c009 : CONNECT ( areg, 1, add, 1, ctrl, 1, add, 1);     {DEPENDENCY, 1}

d004 : FOR i := 0 TO 7 DO
          CONNECT ( areg, 1, reg_out, i, adder, 1, augend, i);
                                                  {DEPENDENCY, 1}

d005 : FOR i := 0 TO 7 DO
          CONNECT ( mreg, 1, reg_out, i, adder, 1, addend, i);
                                                  {DEPENDENCY, 1}

d006 : FOR i := 0 TO 7 DO
          CONNECT ( adder, 1, sum_out, i, areg, 1, reg_in, i+7);
                                                  {DEPENDENCY, 1}

c013 : CONNECT ( areg, 1, shift, 1, ctrl, 1, shift );   {DEPENDENCY, 1}

c014 : CONNECT ( qreg, 1, shift, 1, ctrl, 1, shift );   {DEPENDENCY, 1}

c015 : CONNECT ( counter, 1, incr, 1, ctrl, 1, inc );   {DEPENDENCY, 1}

d007 : FOR i := 0 TO 7 DO
```

```
          CONNECT ( mux, 1, mux_out, i, mult, 1, data_out, i); {DEPENDENCY, 2}

c017 : CONNECT ( mux, 1, select, 1, ctrl, 1, out_select, 1); {DEPENDENCY, 1}

c018 : CONNECT ( ctrl, 1, stop, 1, mult, 1, stop, 1);   {DEPENDENCY, 1}

d008 : FOR i := 0 TO 7 DO
          CONNECT ( areg, 1, reg_out, i+7, mux, 1, in1, i); {DEPENDENCY, 1}

d009 : FOR i := 0 TO 7 DO
          CONNECT ( areg, 1, reg_out, i, mux, 1, in2, i); {DEPENDENCY, 1}

c021 : CONNECT ( mux, 1, enable, 1, ctrl, 1, out_enable, 1); {DEPENDENCY, 1}

c022 : CONNECT ( ctrl, 1, start, 1, mult, 1, start, 1);   {DEPENDENCY, 1}

d010 : CONNECT ( adder, 1, co, 1, areg, 1, co, 1); {DEPENDENCY, 1}
```

architectural properties list for prototype multiplier



Trace output for network schematic of prototype multiplier

```
(DEPENDENCY (
 (INSTRUCTION (10, INST1) )
 (IS-CARRIED-OUT-BY ((controller, ctrl, 10, INST1)))
 (CTRL (
        ((SRC  mult, 1, start)
         (DEST ctrl, 1, start)
         (PATH c022))))
 (DATA-PATH nil) ))

(DEPENDENCY (
 (INSTRUCTION ( 20, INST1 ) )
  (IS-CARRIED-OUT-BY ((controller ctrl, 20, INST2) )
                      (storage qreg, 10, INST1)
                      (storage qreg, 20, INST1) ))
  (CTRL (
         ((SRC  ctrl, 1, loadq)
          (DEST qreg, 1, loadq)
          (PATH c001)) ))
  (DATA-PATH (
              ((SRC  mult, 1, data_in)
               (DEST qreg, 1, reg_in)
               (PATH d001)) )) ))

(DEPENDENCY (
 (INSTRUCTION ( 20, INST2 ) )
 (IS-CARRIED-OUT-BY ((controller ctrl, 20, INST1)
                     (storage areg, 10, INST1)
                     (storage areg, 20, INST1) ))
  (CTRL (
         ((SRC  ctrl, 1, clear)
          (DEST areg, 1, clear)
          (PATH c002)) ))
  (DATA-PATH nil) ))

(DEPENDENCY (
 (INSTRUCTION ( 20, INST3 ) )
 (IS-CARRIED-OUT-BY ((controller ctrl, 20, INST1)
                     (controller counter, 10, INST1)
                     (controller counter, 20, INST1) ))
  (CTRL (
         ((SRC  ctrl, 1, clear)
          (DEST counter, 1, clear)
          (PATH c005)) ))
  (DATA-PATH nil) ))

(DEPENDENCY (
 (INSTRUCTION ( 40, INST1 ))
 (IS-CARRIED-OUT-BY ((controller ctrl, 60, INST1)
                     (controller ctrl, 30, INST1)
                     (controller ctrl, 50, INST1)
                     (controller ctrl, 60, INST2)
                     (controller counter, 30, INST1)
                     (controller counter, 40, INST1)
                     (controller counter, 50, INST1)
                     (storage mreg, 10, INST1)
```

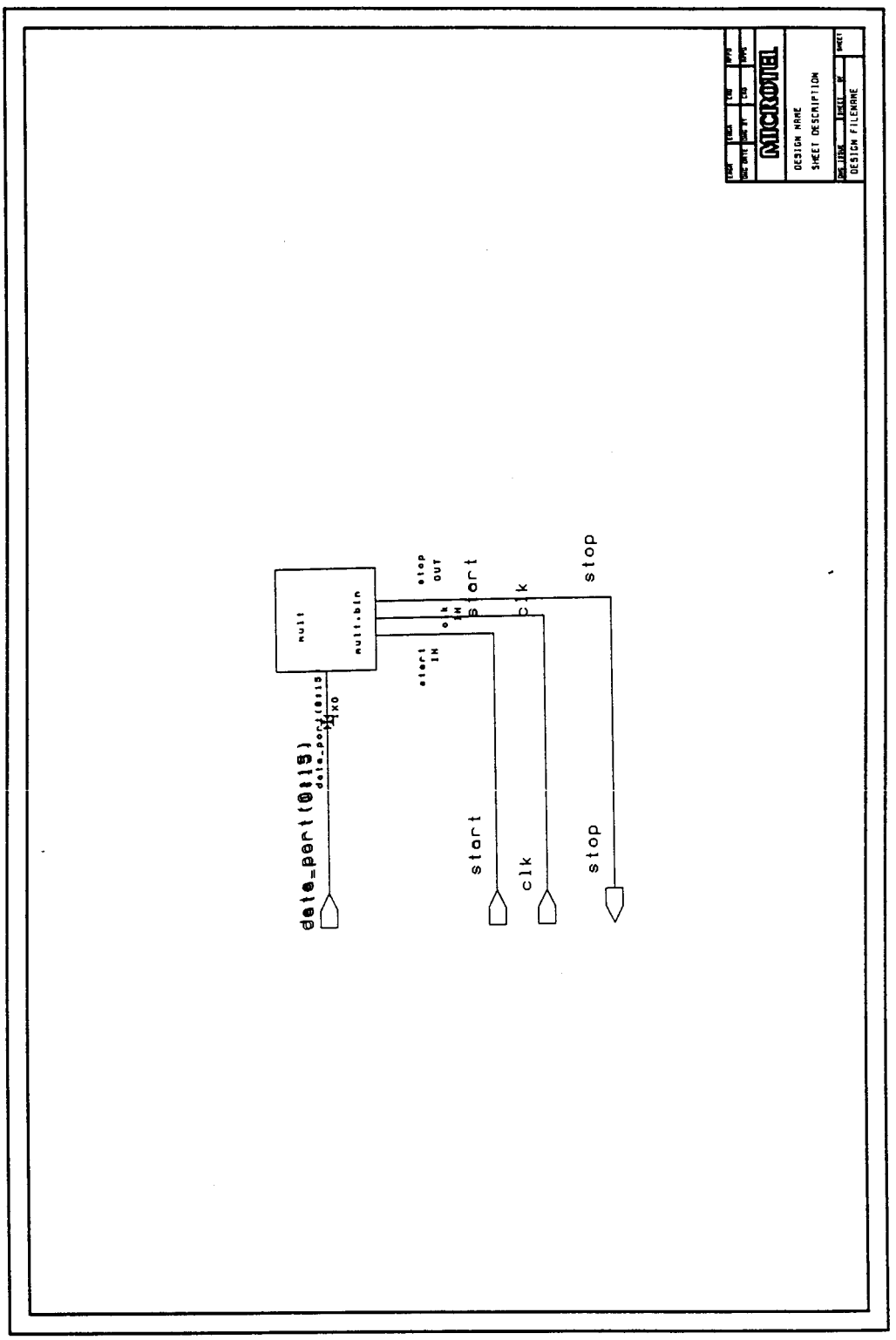
```
                     (storage mreg, 20, INST1) ))
 (CTRL (
        ((SRC  ctrl, 1, loadm)
         (DEST mreg, 1, load)
         (PATH c006)) ))
 (DATA-PATH (
             ((SRC  mult, 1, data_in)
              (DEST mreg, 1, reg_in)
              (PATH d002)) )) ))

(DEPENDENCY (
 (INSTRUCTION ( 60, INST1 ))
 (IS-CARRIED-OUT-BY ((controller ctrl, 80, INST1)
                     (controller ctrl, 30, INST1)
                     (controller ctrl, 50, INST1)
                     (controller ctrl, 80, INST3)
                     (controller counter, 30, INST1)
                     (controller counter, 40, INST1)
                     (controller counter, 50, INST1) ))
 (CTRL (
        ((SRC  counter, 1, count_out)
         (DEST ctrl, 1, count_in)
         (PATH c007)) ))
 (DATA-PATH nil)  ))

(DEPENDENCY (
 (INSTRUCTION ( 70, INST1 ))
  (IS-CARRIED-OUT-BY ((controller ctrl, 80, INST1)
                      (storage qreg, 50, INST1) ))
  (CTRL (
         ((SRC  qreg, 1, reg_val[0])
          (DEST ctrl, 1, q0)
          (PATH c008)) ))
  (DATA-PATH nil))      ))

(DEPENDENCY (
 (INSTRUCTION ( 80, INST1 ))
 (IS-CARRIED-OUT-BY ((controller ctrl, 90, INST1)
                     (controller ctrl, 30, INST1)
                     (controller ctrl, 50, INST1)
                     (controller ctrl, 90, INST2)
                     (controller counter, 30, INST1)
                     (controller counter, 40, INST1)
                     (controller counter, 50, INST1)
                     (storage areg, 30, INST1)
                     (storage areg, 40, INST1)
                     (storage areg, 40, INST2)
                     (storage areg, 70, INST1)
                     (storage mreg, 30, INST1)
                     (processor add, 10, INST1)
                     (processor add, 20, INST1)
                     (processor add, 30, INST1)
                     (processor add, 40, INST1)
                     (processor add, 50, INST1) ))
 (CTRL (
        ((SRC  ctrl, 1, add)
         (DEST areg, 1, add))
```

dependency network for prototype multiplier

```
        (PATH c009)) ))
 (DATA-PATH (
             ((SRC  areg, 1, reg_out[INDEXED])
              (DEST adder 1, augend)
              (PATH d004))
             ((SRC  breg, 1, reg_out)
              (DEST adder, 1, addend)
              (PATH d005))
             ((SRC  adder, 1, sumout)
              (DEST areg, 1, reg_in[INDEXED])
              (PATH d006))
             ((SRC  adder, 1, co)
              (DEST areg, 1, co)
              (PATH d010)) )) ))

(DEPENDENCY (
 (INSTRUCTION ( 100, INST1))
 (IS-CARRIED-OUT-BY ((controller ctrl, 110, INST1)
                     (controller ctrl, 30, INST1)
                     (controller ctrl, 50, INST1)
                     (controller ctrl, 110, INST2)
                     (controller counter, 30, INST1)
                     (controller counter, 40, INST1)
                     (controller counter, 50, INST1)
                     (storage areg, 50, INST2)
                     (storage areg, 60, INST1) ))
  (CTRL (
         ((SRC  ctrl, 1, shift))
          (DEST areg, 1, shift))
          (PATH c013)) ))
  (DATA-PATH nil) ))

(DEPENDENCY (
 (INSTRUCTION ( 100, INST2 ))
 (IS-CARRIED-OUT-BY ((controller ctrl, 110, INST1)
                     (controller ctrl, 30, INST1)
                     (controller ctrl, 80, INST1)
                     (controller ctrl, 110, INST2)
                     (controller counter, 30, INST1)
                     (controller counter, 40, INST1)
                     (controller counter, 50, INST1)
                     (storage qreg, 30, INST1)
                     (storage qreg, 40, INST1) ))
 (CTRL (
        ((SRC  ctrl, 1, shift)
         (DEST qreg, 1, shift)
         (PATH c014) ))
 (DATA-PATH nil) ))

(DEPENDENCY (
 (INSTRUCTION ( 100, INST3))
 (IS-CARRIED-OUT-BY ((controller ctrl, 110, INST1)
                     (controller ctrl, 30, INST1)
                     (controller ctrl, 50, INST1)
                     (controller ctrl, 60, INST2)
                     (controller ctrl, 90, INST3)
                     (controller ctrl, 110, INST2)
```

```
                     (controller ctrl, 130, INST4)
                     (controller counter, 30, INST1)
                     (controller counter, 50, INST1)
                     (controller counter, 40, INST1) )
 (CTRL (
        ((SRC ctrl, 1, incr)
         (DEST counter, 1, inc)
         (PATH c015)) ))
 (DATA-PATH nil)  ))

(DEPENDENCY (
 (INSTRUCTION ( 120, INST1)
 (IS-CARRIED-OUT-BY ((controller ctrl, 130, INST2)
                     (controller ctrl, 130, INST3)
                     (controller ctrl, 30, INST1)
                     (controller ctrl, 50, INST1)
                     (controller ctrl, 130, INST4)
                     (controller counter, 30, INST1)
                     (controller counter, 40, INST1)
                     (controller counter, 50, INST1)
                     (storage areg, 80, INST1)
                     (router mux, 10, INST1)
                     (router mux, 20, INST1)
                     (router mux, 30, INST1)
                     (router mux, 60, INST1) ))
 (CTRL (
        ((SRC  ctrl, 1, out_select)
         (DEST mux, 1, select)
         (PATH c017)
        ((SRC  ctrl, 1, out_enable)
         (DEST mux, 1, enable)
         (PATH c021) ) ))
 (DATA-PATH (
             ((SRC  areg, 1, reg_out[INDEXED])
              (DEST mux, 1, in1)
              (PATH d008))
             ((SRC  mux, 1, mux_out)
              (DEST mult, 1, data_out)
              (PATH d007)) ))  ))

(DEPENDENCY (
 (INSTRUCTION ( 120, INST2))
 (IS-CARRIED-OUT-BY ((controller ctrl, 130, INST1))
 (CTRL (
        ((SRC  ctrl, 1, stop)
         (DEST mult 1, stop)
         (PATH c018)) ))
 (DATA-PATH nil) ))

(DEPENDENCY (
 (INSTRUCTION ( 140, INST1))
 (IS-CARRIED-OUT-BY ((controller ctrl, 140, INST1)
                     (controller ctrl, 150, INST2)
                     (router mux, 10, INST1)
                     (router mux, 40, INST1)
                     (router mux, 50, INST1)
                     (router mux, 60, INST1)
```

```
                    (storage areg, 80, INST1)) )
(CTRL (
       ((SRC ctrl, 1, out_select)
        (DEST mux, 1, select)
        (PATH c017))
       ((SRC ctrl, 1, out_enable)
        (DEST mux, 1, enable)
        (PATH c021)) ))
(DATA-PATH (
            ((SRC  areg, 1, reg_out)
             (DEST mux, 1, in2)
             (PATH d009))
            ((SRC  mux, 1, muxout)
             (DEST mult, 1, data_out)
             (PATH d007)) )) ))
```



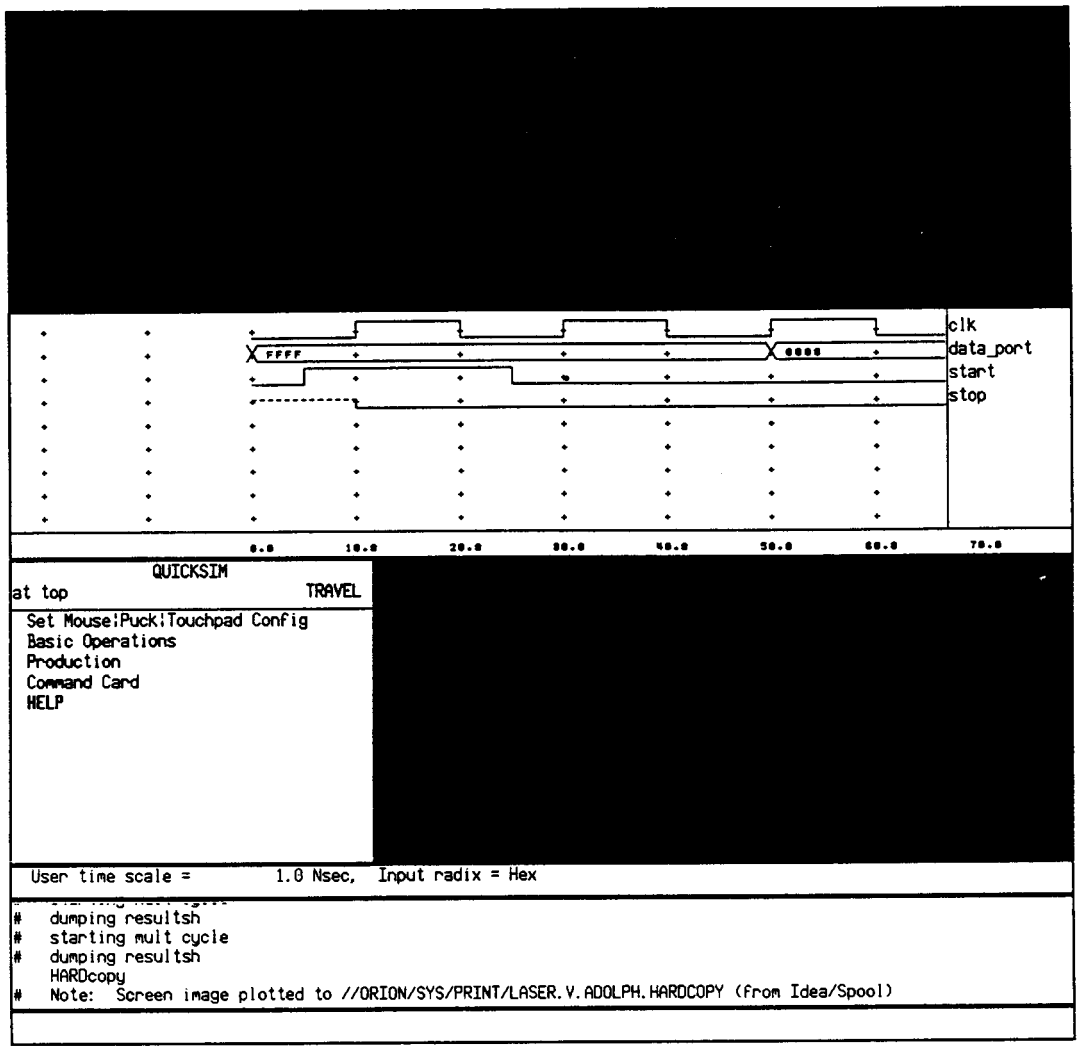
Schematic for user-spec multiplier

```
{ UMULT }
{ RT-TYPE processor }
PROCEDURE mult$$allocate;
  VAR
    state_ptr                : mult_state_ptr_type;
    i                        : INTEGER16;

  BEGIN

    sim_$message ('allocation for mult is starting.....',45);
    sim_$allocate (state_rec_size, state_ptr);
    WITH state_ptr^:sp DO
      BEGIN
{ REG-SYMBOLS }
        WITH sp.creg DO   { DEPENDENCY, 3 }
          BEGIN
            reg_name := creg_reg;
            reg_width := 5;
          END;
        WITH sp.areg DO   { DEPENDENCY, 6 }
          BEGIN
            reg_name := areg_reg;
            reg_width := 16;
          END;
        WITH sp.mreg DO   { DEPENDENCY, 2 }
          BEGIN
            reg_name := mreg_reg;
            reg_width := 8;
          END;
        WITH sp.qreg DO   { DEPENDENCY, 3 }
          BEGIN
            reg_name := qreg_reg;
            reg_width := 8;
          END;
{ PORT-SYMBOLS }
        WITH sp.data_in DO   { DEPENDENCY, 1 }
          BEGIN
            port_name := data_in_bus;
            iotype := data_port;
            iodir  := input_port;
            port_width := 8;
          END;
        WITH sp.data_out DO   { DEPENDENCY, 1 }
          BEGIN
            port_name := data_out_bus;
            iotype := data_port;
            iodir  := output_port;
            port_width := 8;
?   END;
        WITH sp.start DO   { DEPENDENCY, 1 }
          BEGIN
            port_name := start_sig;
            iotype := ctrl_port;
            iodir  := input_port;
            port_width := 1;
          END;
        WITH sp.stop  DO   { DEPENDENCY, 1 }
          BEGIN
            port_name := stop_sig;
```

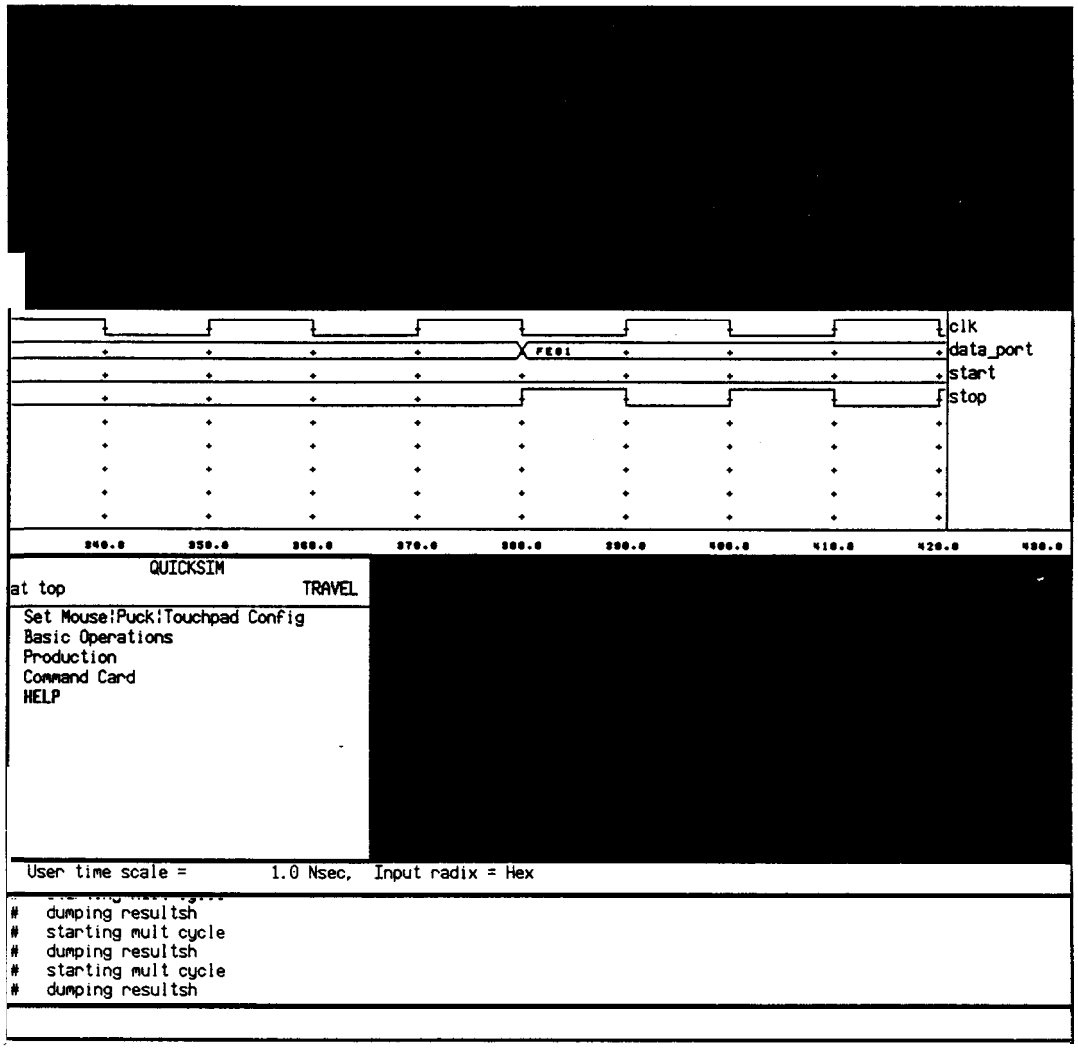
```
            iotype := ctrl_port;
            iodir  := output_port;
            port_width := 1;
          END;
    END;

    simbase_$instance_ptr^.user_data_area := state_ptr;
    sim_$message ('...allocation is complete.',26);
    END { allocate } ;
```

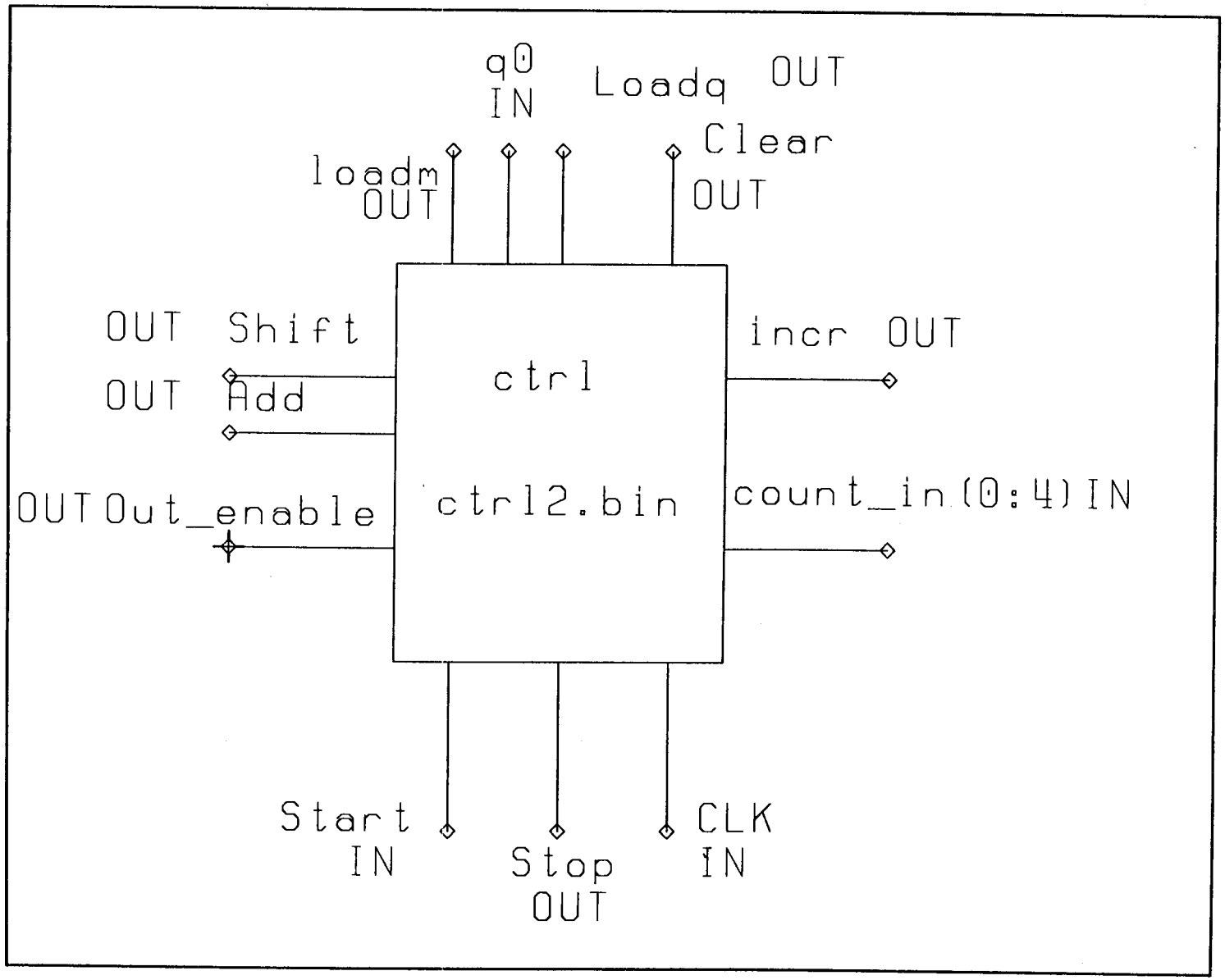
Behavioural model describing multiplier behaviour

```
{ FUNCTIONAL-CAPABILITIES }
PROCEDURE mult_function;
  VAR
    j                           : integer16;
    state_ptr                   : mult_state_ptr_type;
    k                           : integer16;
  BEGIN
    WITH simbase_$instance_ptr^:i DO
      state_ptr := i.user_data_area;
    WITH state_ptr^:sp DO
      BEGIN
{ CLOCK_BEGIN, 0 }
{ DEC,  1 }
{ INST1 } IF (signal_in (sp.start) = sim_$one) THEN
            BEGIN
{ ACT,  2 }
{ INST1 }     bus_in (sp.data_port, 0, 7, sp.multiplier, 0, 7);
{ INST2 }     init_reg  (sp.accumulator, sim_$zero);
{ INST3 }     init_reg  (sp.counter, sim_$zero);
{ INST4 }     bus_in (sp.data_port, 8, 15, sp.multicand, 0, 7);
              sp.clock_cycle := clock1;
            END
          ELSE CASE sp.clock_cycle OF
{ CLOCK_STEP  3 }
        clock1: BEGIN
{ DEC,  4 }
{ INST1 }   IF (decode_reg (sp.counter) < 8) THEN
              BEGIN
{ DEC,  5 }
{ INST1 }       IF (sp.multiplier.reg_val[0] = sim_$one) THEN
{ ACT,  6 }
{ INST1 }         add_reg ( sp.accumulator, 8, 15,
                            sp.multicand, 0, 7,
                            sp.accumulator, 8, 15,
                            sp.carry_out ) ;
                sp.clock_cycle := clock2i ;
              END
            ELSE
              sp.clock_cycle := clockn;
          END;
{ CLOCK_STEP,  7 }
        clock2i: BEGIN
{ ACT,  8 }
{ INST1 }       shift_reg ( sp.accumulator, 0, 15, RIGHT, sp.carry_out);
{ INST2 }       shift_reg ( sp.multiplier, 0, 7,  RIGHT, sim_$zero);
{ INST3 }       incr_reg ( sp.counter);
            sp.clock_cycle := clock1;
          END;
{ CLOCK_STEP,  9 }
        clockn: BEGIN
{ ACT,  10 }
{ INST1 }       signal_out (sp.stop, sim_$one);
{ INST1 }       bus_out (sp.data_port, 0, 15, sp.accumulator, 0, 15);
          END;
        END;
{ CLOCK_END, 11 }
      END;
  END;
```



Trace output for user-spec multiplier


clk
data_port
start
stop
FE01
QUICKSIM
at top
TRAVEL
Set Mouse!Puck!Touchpad Config
Basic Operations
Production
Command Card
HELP
User time scale = 1.0 Nsec, Input radix = Hex
dumping resultsh
starting mult cycle
dumping resultsh
starting mult cycle
dumping resultsh


q0
IN
Loadq
OUT
Clear
OUT
loadm
OUT
OUT Shift
OUT Add
OUT Out_enable
ctrl
ctrl2.bin
incr OUT
count_in(0:4) IN
Start
IN
Stop
OUT
CLK
IN

150

```
{ CTRL } { DEPENDENCY, 1 }
PROCEDURE ctrl$$allocate;
  VAR
    state_ptr                : ctrl_state_ptr_type;
    i                        : INTEGER16;

  BEGIN

    sim_$message ('allocation for  controller2 is starting.....',44);
    sim_$allocate (state_rec_size, state_ptr);
    WITH state_ptr↑:sp DO
      BEGIN
{ REG-SYMBOLS }
{ PORT-SYMBOLS }
        WITH sp.shift  DO
          BEGIN
            port_name := shift_sig;
            iotype := ctrl_port;
            iodir  := output_port;
            port_width := 1;
          END;
        WITH sp.add  DO
          BEGIN
            port_name := add_sig;
            iotype := ctrl_port;
            iodir  := output_port;
            port_width := 1;
          END;
        WITH sp.out_enable  DO
          BEGIN
            port_name := out_enable_sig;
            iotype := ctrl_port;
            iodir  := output_port;
            port_width := 1;
          END;
        WITH sp.start  DO
          BEGIN
            port_name := start_sig;
            iotype := ctrl_port;
            iodir  := input_port;
            port_width := 1;
          END;
        WITH sp.stop  DO
          BEGIN
            port_name := stop_sig;
            iotype := ctrl_port;
            iodir  := output_port;
            port_width := 1;
          END;
        WITH sp.clk  DO
          BEGIN
            port_name := clk_sig;
            iotype := ctrl_port;
            iodir  := input_port;
            port_width := 1;
          END;
        WITH sp.count_in  DO
          BEGIN
            port_name := count_in_sig;
```

```
            iotype := ctrl_port;
            iodir  := input_port;
            port_width := 5;
          END;
        WITH sp.loadm  DO
          BEGIN
            port_name := loadm_sig;
            iotype := ctrl_port;
            iodir  := output_port;
            port_width := 1;
          END;
        WITH sp.clear  DO
          BEGIN
            port_name := clear_sig;
            iotype := ctrl_port;
            iodir  := output_port;
            port_width := 1;
          END;
        WITH sp.loadq  DO
          BEGIN
            port_name := loadq_sig;
            iotype := ctrl_port;
            iodir  := output_port;
            port_width := 1;
          END;
        WITH sp.incr  DO
          BEGIN
            port_name := incr_sig;
            iotype := ctrl_port;
            iodir  := output_port;
            port_width := 1;
          END;
        WITH sp.q0  DO
          BEGIN
            port_name := q0_sig;
            iotype := ctrl_port;
            iodir  := input_port;
            port_width := 1;
          END;
        WITH sp.count_state  DO
            reg_width := 5;
    END;

    simbase_$instance_ptr↑.user_data_area := state_ptr;
    sim_$message (' ...allocation is complete.',26);
    END { allocate } ;
```

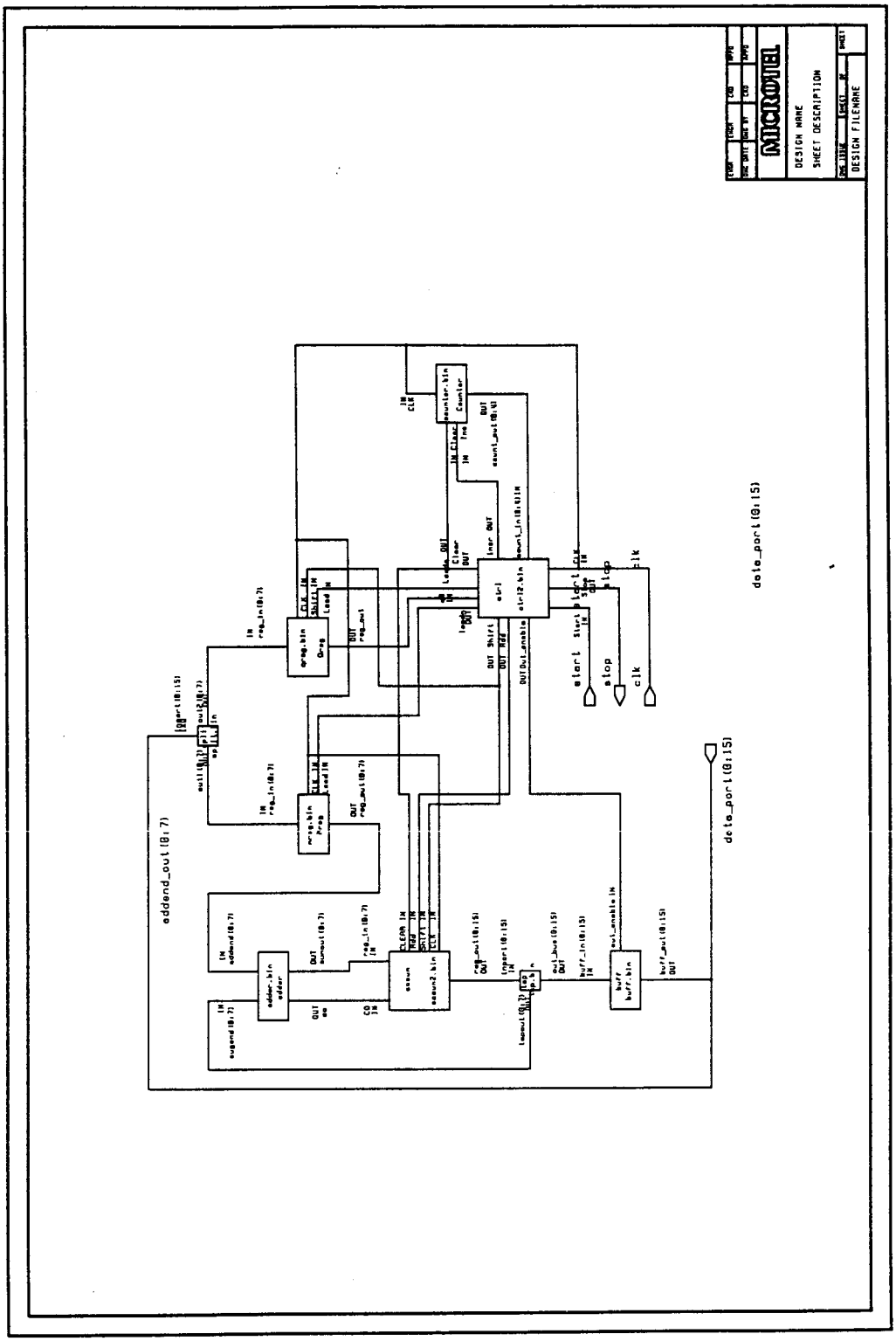
```
PROCEDURE ctrl_function;
  VAR
    state_ptr                   : ctrl_state_ptr_type;
    carry                       : INTEGER15;
  BEGIN
    WITH simbase_$instance_ptr^:1 DO
      state_ptr := i.user_data_area;
    WITH state_ptr^:sp DO
      BEGIN
{ CLOCK_BEGIN, 0 }
{ DEC, 10 }
{ INST1 } IF (signal_in(sp.start) = sim_$one) THEN
                                                { DEPENDENCY, 1}
          BEGIN
{ ACT, 20 }
{ INST1 }   signal_out (sp.clear, sim_$one);
                                                { DEPENDENCY, 2}
{ INST2 }   signal_out (sp.loadq, sim_$one);
                                                { DEPENDENCY, 1}
{ INST3 }   signal_out (sp.loadm, sim_$one);
                                                { DEPENDENCY, 1}
          END
        ELSE
          BEGIN
{ INST1 }   get_arg (sp.count_in, sp.count_state);
                                                { DEPENDENCY, 7}
{ CLOCK_STEP, 30 }
{ DEC, 40 }
{ INST1 }  CASE decode_reg (sp.count_state) OF
                                                { DEPENDENCY, 7}
            0,2,4,6,8,10,12,14:
              BEGIN
{ DEC, 50 }
{ INST1 }       IF (signal_in (sp.q0) = sim_$one) THEN
                                                { DEPENDENCY, 1}
{ ACT, 60 }
{ INST1 }         signal_out (sp.add, sim_$one)
                                                { DEPENDENCY, 1}
                ELSE
{ INST2 }         signal_out (sp.add, sim_$zero);
                                                { DEPENDENCY, 1}
{ INST3 }       signal_out (sp.incr, sim_$one);
                                                { DEPENDENCY, 1}
              END;
{ CLOCK_STEP, 70 }
            1,3,5,7,9,11,13,15: .
              BEGIN
{ ACT, 80 }
{ INST1 }       signal_out (sp.shift, sim_$one);
                                                { DEPENDENCY, 1}
{ INST2 }       signal_out (sp.incr, sim_$one);
                                                { DEPENDENCY, 1}
              END;
{ CLOCK_STEP, 90 }
            16: BEGIN
{ ACT, 100}
{ INST1 }       signal_out (sp.stop, sim_$one);
                                                { DEPENDENCY, 1}
{ INST2 }       signal_out (sp.out_enable, sim_$one);
```

```
                                                { DEPENDENCY, 1}
              END;

          END;
        END;
{ CLOCK_END, 110 }
      END;
  END;
```

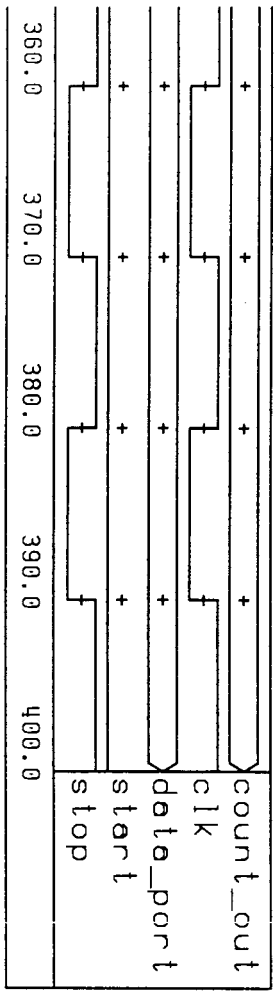
buff_in(0:15)
IN
out_enable IN
buff
buff.bin
buff_out(0:15)
OUT

```
{ BUFF } { DEPENDENCY, 1 }
{ RT-TYPE router }
PROCEDURE buff$$allocate;
  VAR
    state_ptr                : buff_state_ptr_type;
    j                        : INTEGER16;

  BEGIN

    sim_$message ('buffer allocation is starting.....',29);
    sim_$allocate (state_rec_size, state_ptr);
    WITH state_ptr↑:sp DO
      BEGIN
        { psuedo register definitions }
        WITH sp.buffreg DO
          BEGIN
            reg_name := buffer_reg;
            reg_width := 16;
            FOR j := 0 TO 16 DO
              reg_val[j] := sim_$unknown;
          END;
        WITH sp.hizreg DO
          BEGIN
            reg_name := buffer_reg;
            reg_width := 16;
          END;
{ REG-SYMBOLS }
{ PORT-SYMBOLS }
        WITH sp.buff_in DO
          BEGIN
            port_name := buff_in_bus;
            iotype := data_port;
            iodir  := input_port;
            port_width := 16;
          END;
        WITH sp.buff_out DO
          BEGIN
            port_name := buff_out_bus;
            iotype := data_port;
            iodir  := output_port;
            port_width := 16;
          END;
        WITH sp.enable  DO
          BEGIN
            port_name := enable_sig;
            iotype := control_port;
            iodir  := input_port;
            port_width := 1;
          END;
    END;

    simbase_$instance_ptr↑.user_data_area := state_ptr;
    sim_$message ('...allocation is complete.',26);
    END { allocate } ;
```

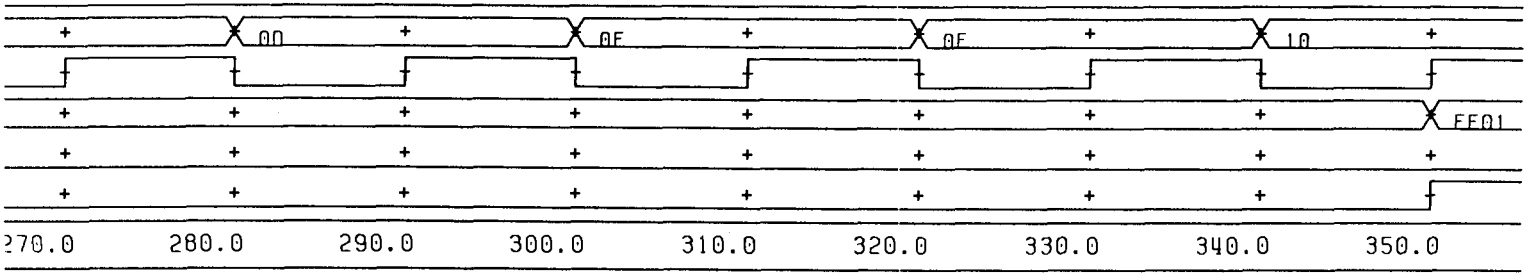
```
{ FUNCTIONAL-REQUIREMENTS }
PROCEDURE buff_function;
  VAR
    state_ptr                : buff_state_ptr_type;
  BEGIN
    WITH simbase_$instance_ptr↑:i DO
      state_ptr := i.user_data_area;
    WITH state_ptr↑:sp DO
      BEGIN
{ CLOCK_BEGIN, 0 }
{ DEC,  10 }
{ INST1 }IF (signal_in (sp.enable) = sim_$one) THEN
                                   { DEPENDENCY, 1 }
        BEGIN
{ ACT,  20 }
{ INST1 }     get_arg (sp.buff_in, 0, 7, sp.buffreg, 0, 7) ;
                                   { DEPENDENCY, 1 }
{ INST1 }     put_arg (sp.buff_out, 0, 7, sp.buffreg, 0, 7);
                                   { DEPENDENCY, 1 }
        END ;
      ELSE
{ ACT,  30 }
{ INST1 }  bus_out (sp.buff_out, sp.hizreg); }
                                   { DEPENDENCY, 1 }
      END;
{ CLOCK_END, 40 }
  END;
```



Network schematic for user-spec multiplier



Trace output for network schematic of user-spec multiplier


00
0F
0F
10
FF01
270.0
280.0
290.0
300.0
310.0
320.0
330.0
340.0
350.0

APPENDIX TWO - THE DIVIDER CIRCUIT

This appendix is a listing of the program that describes the behaviour of the restoring divider. This is an untested model and is included only to support the comparison between it and the prototype multiplier.

```
{ RDIV }
PROCEDURE div$$allocate;
  VAR
    state_ptr                  : div_state_ptr_type;
    i                          : INTEGER16;

  BEGIN

    sim_$message ('allocation for divider is starting.....',45);
    sim_$allocate (state_rec_size, state_ptr);
    WITH state_ptr^:sp DO
      BEGIN
        sp.clock_cycle := clock1;
{ REG-SYMBOLS }
        WITH sp.counter DO
          BEGIN
            reg_name := counter_reg;
            reg_width := 5;
          END;
        WITH sp.accumulator DO
          BEGIN
            reg_name := accumulator_reg;
            reg_width := 16;
          END;
        WITH sp.dividend DO
          BEGIN
            reg_name := dividend_reg;
            reg_width := 8;
          END;
        WITH sp.quotient DO
          BEGIN
            reg_name := quotient_reg;
            reg_width := 8;
          END;
{ PORT-SYMBOLS }
        WITH sp.start DO
          BEGIN
            port_name := start_sig;
            iotype := ctrl_port;
            iodir  := input_port;
            port_width := 1;
          END;
        WITH sp.stop  DO
          BEGIN
            port_name := stop_sig;
            iotype := ctrl_port;
            iodir  := output_port;
            port_width := 1;
          END;
        WITH sp.clk  DO
          BEGIN
            port_name := clk_sig;
            iotype := ctrl_port;
            iodir  := input_port;
            port_width := 1;
          END;
    END;

    simbase_$instance_ptr^.user_data_area := state_ptr;
```

```
    sim_$message ('...allocation is complete.',26);
    END { allocate } ;
```

```
{ FUNCTIONAL-CAPABILITIES }
PROCEDURE div_function;
  VAR
    }                                   integer16;
    state_ptr                           div_state_ptr_type;
    x                                   integer16;
  BEGIN
    WITH simbase_$instance_ptr^ DO
      state_ptr := i.user_data_area;
    WITH state_ptr^,sp DO
      BEGIN
{ CLOCK_BEGIN, 0 }
{ DEC, 10 }
{ INST1 } IF (signal_in (sp.start) = sim_$one) THEN
            BEGIN
{ ACT, 20 }
{ INST1 }     bus_in (sp.data_port, 0, 7, sp.accumulator, 8, 15);
{ INST2 }     init_reg  (sp.quotient,  sim_$zero);
{ INST3 }     init_reg  (sp.counter, sim_$zero);
              sp.clock_cycle := clock1;
            END
          ELSE CASE sp.clock_cycle OF
{ CLOCK_STEP, 30 }
        clock1: BEGIN
{ ACT, 40 }
{ INST1 }     bus_in (sp.data_in, 0, 7, sp.accumulator, 0, 7);
              sp.clock_cycle := clock2 ;
        END;
{ CLOCK_STEP, 50 }
        clock2: BEGIN
{ ACT, 60 }
{ INST1 }     bus_in (sp.data_in, 0, 7, sp.divisor, 0, 7);
              sp.clock_cycle := clock11 ;
        END;
{ CLOCK_STEP, 60 }
        clock11: BEGIN
{ DEC, 70 }
{ INST1 }   IF (decode_reg (sp.counter) < 8) THEN
              BEGIN
{ ACT, 80 }
{ INST1 }       shift_reg ( sp.accumulator, 0, 15, LEFT, sim_$zero);
{ INST2 }       shift_reg ( sp.quotient, 0, 7,  LEFT, sim_$zero);
{ INST3 }       incr_reg ( sp.counter);
                sp.clock_cycle := clock12;
            END
          ELSE
            sp.clock_cycle := clockn;
        END;
{ CLOCK-STEP, 90 }
        clock12: BEGIN
{ ACT, 100 }
{ INST1 }           sub_reg ( sp.accumulator, 8, 15,
                              sp.divisor, 0, 7
                              sp.accumulator, 8, 15,
                              sp.carry_out );
                    sp.clock_cycle := clock13;
        END;
{ CLOCK-STEP, 110 }
        clock13: BEGIN
```

```
{ DEC, 120 }
{ INST1 }       IF (sp.carry_out = sim_$one) THEN
{ ACT, 130 }
{ INST1 }         add_reg ( sp.accumulator, 8, 15,
                            sp.divisor, 0, 7
                            sp.accumulator, 8, 15,
                            sp.carry_out )
                ELSE
{ INST2 }         set_bit (sp.dividend.reg_val[0], sim_$one) ;
                sp.clock_cycle := clock11 ;
        END;
{ CLOCK_STEP, 140 }
        clockn: BEGIN
{ ACT, 150 }
{ INST1 }       signal_out (sp.stop, sim_$one);
{ INST2 }       bus_out (sp.data_out, 0, 7, sp.accumulator, 0, 7);
                sp.clock_cycle := clockn1;
        END;
{ CLOCK_STEP, 160 }
        clockn1: BEGIN
{ ACT, 170 }
{ INST1 }       bus_out (sp.data_out, 0, 7, sp.accumulator, 8, 15);
                sp.clock_cycle := clockn2;
        END;
{ CLOCK_STEP, 180 }
        clockn1: BEGIN
{ ACT, 190 }
{ INST1 }       bus_out (sp.data_out, 0, 7, sp.dividend, 0, 7);
        END;
      END;
{ CLOCK_END, 200 }
    END;
  END;
```

APPENDIX THREE - THE RULE BASE

This appendix lists the rules that were developed to support the example design task of chapter five.

```
/*****************************************************
/*
/*        Install Port Rule
/*
/*****************************************************
        IF object-type of <port is PORT
          AND sym-status of <port is NOT installed
        THEN
           install <port



/*****************************************************
/*
/*        Install Register Rule
/*
/*****************************************************
        IF object-type of <reg is REGISTER
          AND sym-status of <reg is NOT installed
        THEN
           install <port



/*****************************************************
/*
/*        Strong Port Match Rule
/*
/*****************************************************
       IF object-type of <port-1 is PORT
          AND object-type of <port-2 is PORT
           AND source-type of <port-1 is user-spec
          AND source-type of <port-2 is prototype
          AND width of <port-1 & <port-2 is the SAME
          AND io of <port-1 & <port-2 is the SAME
          AND info-type of <port-1 & <port-2 is the SAME
          AND binding of <port-1 & <port-2 is NOT strong-bound
       THEN
          strong-bind (<port-1, <port-2)
```

```
/*****************************************************
/*
/*        Weak Port Match Rule
/*
/*****************************************************
        IF object-type of <port-1 is PORT
          AND object-type of <port-2 is PORT
           AND source-type of <port-1 is user-spec
          AND source-type of <port-2 is prototype
          AND width of <port-1 & <port-2 is the SAME
          AND io of <port-1 or <port-2 is input-output
          AND info-type of <port-1 & <port-2 is the SAME
          AND binding of <port-1 & <port-2 is NOT weak-bound
       THEN
          weak-bind (<port-1, <port-2)



/*****************************************************
/*
/*        Strong Register Match Rule
/*
/*****************************************************
       IF object-type of <reg-1 is REGISTER
          AND object-type of <reg-2 is REGISTER
          AND source-type of <reg-1 is user-spec
          AND source-type of <reg-2 is prototype
          AND width of <reg-1 & <reg-2 is the SAME
          AND binding of <reg-1 & <reg-2 is NOT strong-bound
       THEN
          strong-bind (<reg-1, <reg-2)



/*****************************************************
/*
/*        Weak Register Match Rule
/*
/*****************************************************
       IF object-type of <reg-1 is REGISTER
          AND object-type of <reg-2 is REGISTER
          AND source-type of <reg-1 is user-spec
          AND source-type of <reg-2 is prototype
          AND width of <reg-1 & <reg-2 are MULTIPLES
          AND binding of <reg-1 & <reg-2 is NOT weak-bound
       THEN
          weak-bind (<reg-1, <reg-2)
```

```
/*******************************************************
/*
/*        Clock-Begin Match Rule
/*
/*******************************************************
       IF object-type of <statement-1 is STATEMENT
          AND object-type of <statement-2 is STATEMENT
          AND source-type of <statement-1 is prototype
          AND source-type of <statement-2 is user-spec
          AND statement-type of <statement-1 is CLOCK-BEGIN
          AND statement-type of <statement-2 is CLOCK-BEGIN
       THEN
          strong-bind (<statment-1, statement-2)



/*******************************************************
/*
/*        Clock-End Match Rule
/*
/*******************************************************
       IF object-type of <statement-1 is STATEMENT
          AND object-type of <statement-2 is STATEMENT
          AND source-type of <statement-1 is prototype
          AND source-type of <statement-2 is user-spec
          AND statement-type of <statement-1 is CLOCK-END
          AND statement-type of <statement-2 is CLOCK-END
       THEN
          strong-bind (<statment-1, statement-2)



/*******************************************************
/*
/*        Strong Statement Match Rule
/*
/*******************************************************
       IF object-type of <statement-1 is STATEMENT
          AND object-type of <statement-2 is STATEMENT
          AND source-type of <statement-1 is prototype
          AND source-type of <statement-2 is user-spec
          AND binding of <statement-1 & <statement-2 is weak-bound
          AND binding of successor(<statment-1) & successor(<statement-2) is strong-bound
       THEN
          strong-bind (<statment-1, statement-2)
```

```
/*******************************************************
/*
/*        Weak Statement Match Rule
/*
/*******************************************************
       IF object-type of <statement-1 is STATEMENT
          AND object-type of <statement-2 is STATEMENT
          AND source-type of <statement-1 is prototype
          AND source-type of <statement-2 is user-spec
          AND binding of <statement-1 & <statement-2 is NIL
          AND binding of instr(<statement-1) & instr(<statement-2) is strong-bound
       THEN
          weak-bind (<statement-1, <statement-2)
/*******************************************************
/*
/*        Strong Instruction Match Rule
/*
/*******************************************************
       IF object-type of <instr-1 is INSTRUCTION
          AND object-type of <instr-2 is INSTRUCTION
          AND source-type of <instr-1 is prototype
          AND source-type of <instr-2 is user-spec
          AND binding of <instr-1 & <instr-2 is strong-bound
          AND binding of operand(<instr-1) & operand(<instr-2) is strong-bound
       THEN
          strong-bind (<instr-1, <instr-2)
```

```
/*****************************************************
/*
/*        Weak Instruction Match Rule
/*
/*****************************************************
      IF object-type of <instr-1 is INSTRUCTION
         AND object-type of <instr-2 is INSTRUCTION
         AND source-type of <instr-1 is prototype
         AND source-type of <instr-2 is user-spec
         AND binding of <instr-1 & <instr-2 is nil
         AND operator-type of <instr-1 & <instr-2 is the SAME
         AND operand-type of <instr-1 & <instr-2 is the SAME
      THEN
         weak-bind (<instr-1, <instr-2)
         AND assign-width (operand of instr-1,
                           operand of instr-2)
```

```
/*****************************************************************************
/*
/*        Change Shift-In Source From Port to Constant Rule
/*
/*****************************************************************************

 DESCRIPTION: This is a very simple transformational rule. It compares the
              shift-in source of two shift registers. If the shift-in source
              for the prototype is a port, and the shift-in source for the
              user-spec is a constant (eg. zero or one) then this rule will
              fire. This rule begins by replacing the DATA-PATH source for the
              prototype's shift-in source with a constant source. It then
              removes the path for the port source from the DATA-PATH and
              replaces it with a path from the constant source. The shift-in
              source in the shift_reg instruction is then changed to reflect
              the constant source.


     IF object-type of <instr-1 is INSTRUCTION
        AND object-type of <instr-2 is INSTRUCTION
        AND source-type of <instr-1 is prototype
        AND source-type of <instr-2 is user-spec
        AND object-type of <port-1 is PORT
        AND source-type of <port-1 is prototype
        AND width of <port-1 is 1
        AND object-type of <reg-1 is REGISTER
        AND source-type of <reg-1 is prototype
        AND object-type of <reg-2 is REGISTER
        AND source-type of <reg-2 is user-spec
        AND operator-type of <instr-1 & <instr-2 is SHIFT_REG
        AND shift-in-src of <instr-1 is <port-1
        AND shift-in-src of <instr-2 is CONSTANT
     THEN
        RM-SRC  ( <instr-1, shift-in-src, <port-1 )
        ADD-SRC ( <instr-1, shift-in-src, CONSTANT )
        RM-PATH ( <instr-1, shift-in-src, <port-1 ))
        ADD-PATH ( <instr-1, shift-in-src, CONSTANT )
        RPL-INSTR-SRC ( <instr-1, shift-in-src, CONSTANT )
```

```
/*************************************************************************
/*
/*      Sample Bit from Word Output Rule
/*
/*************************************************************************

DESCRIPTION: This is another simple transformational rule. It compares the
             output bit width of two registers (storage or shift registers).
             If the width of the prototype register's output port is greater
              than one (eg. word parallel output) and the width  of the output
             port of the user-spec is one bit, then this rule will fire.
             This rule begins by creating a one bit output port and then
             replaces it for the DATA-PATH destination of the prototype's
             bus_out instruction with a bit port. It then changes the source
             for the DATA-PATH to only the bit which is referenced in the
             user-spec. A new path is then created to conduct this signal to
             the output port. The source in the bus_out instruction is then
             indexed to select the appropriate bit within the output word.


     IF object-type of <instr-1 is INSTRUCTION
        AND object-type of <instr-2 is INSTRUCTION
        AND source-type of <instr-1 is prototype
        AND source-type of <instr-2 is user-spec
        AND object-type of <port-1 is PORT
        AND source-type of <port-1 is prototype
        AND width of <port-2 is GREATER THAN 1
        AND object-type of <port-2 is PORT
        AND source-type of <port-2 is user-spec
        AND width of <port-2 is 1
        AND object-type of <reg-1 is REGISTER
        AND source-type of <reg-1 is prototype
        AND object-type of <reg-2 is REGISTER
        AND source-type of <reg-2 is user-spec
        AND operator-type of <instr-1 & <instr-2 is BUS_OUT
        AND output-src of <instr-1 is <reg-1
        AND output-src of <instr-2 is <reg-2
        AND binding of <reg-1 & <reg2 is strong-bound
        AND output-dest of <instr-1 is <port-1
        AND output-dest of <instr-2 is <port-2
     THEN
        MK-PORT ( NAME=$new_port, WIDTH=1, IO=output, INFO-TYPE=data)
        RM-DEST  ( <instr-1, output-dest, <port-1 )
        ADD-DEST ( <instr-1, output-dest, $new_port\
        RPL-SRC  ( <instr-1, output-dest, <reg-1[INDEX] )
        RM-PATH  ( <instr-1, output-dest, <port-1 ))
        ADD-PATH ( <instr-1, output-dest, $new_port)
        RPL-INSTR-SRC ( <instr-1, output-dest, <reg-1[INDEX])
```

```
/*************************************************************************
/*
/*      Input Output Port Merge Rule
/*
/*************************************************************************

DESCRIPTION: This is another simple transformational rule. It recognizes the
             situation when the prototype has both an input port and an
             port, but the user-spec calls for a single input/output port.
             This rule has only one action item, and that is to create the
             new port.


     IF object-type of <port-1 is PORT
        AND source-type of <port-1 is prototype
        AND IO of <port-1 is output
        AND object-type of <port-2 is PORT
        AND source-type of <port-2 is prototype
        AND IO of <port-2 is input
        AND object-type of <port-3 is PORT
        AND source-type of <port-3 is user-spec
        AND IO of <port-3 is input-output
        AND binding of <port-1 & <port-2 is weak-bound
        AND output-dest of <instr-1 is <port-1
        AND output-dest of <instr-2 is <port-2
     THEN
        MK-PORT ( NAME=$new_port, WIDTH=width of <port-3,
                  IO=input-output, INFO-TYPE=info-type of <port-3)
```

```
/*************************************************************************
/*
/*      Half word to word Input Rule
/*
/*************************************************************************

DESCRIPTION: This rule examines the input structure of the prototype to
             to determine if the prototype has two registers connected to
             the input source and if both of those registers have the same
             bitwidth as the input source. Then the input structure of the
             user-spec is examined to see if there are two registers which
             are connected to a port whose bitwidth is the sum of the two
             registers. If the prototype registers and the user-spec registers
             identified in this rule are strongly matched then this rule
             fires. The action items of this rule will connect the prototype
             registers to a port which is the sum of their widths. Often
             the new port will have been created by the Input Output Port
             Merge Rule.


    IF object-type of <port-1 is PORT
       AND source-type of <port-1 is prototype
       AND IO of <port-1 is input
       AND object-type of <port-2 is PORT
       AND source-type of <port-2 is user-spec
       AND IO of <port-2 is input or input-output
       AND object-type of <port-3 is PORT
       AND source-type of <port-3 is prototype
       AND IO of <port-3 is input or input-output
       AND binding of <port-1 & <port-2 is weak-bound
       AND binding of <port-2 & <port-3 is strong-bound
       AND object-type of <reg-1 is register
       AND source-type of <reg-1 is prototype
       AND object-type of <reg-2 is register
       AND source-type of <reg-2 is prototype
       AND object-type of <reg-3 is register
       AND source-type of <reg-3 is user-spec
       AND object-type of <reg-4 is register
       AND source-type of <reg-4 is user-spec
       AND binding of (<reg-1 & <reg-3) OR (<reg-1 & <reg4) is strong-bound
       AND binding of (<reg-2 & <reg-3) OR (<reg-2 & <reg4) is strong-bound
       AND object-type of <instr-1 is instruction
       AND source-type of <instr-1 is prototype
       AND operator-type of <instr-1 is BUS_IN
       AND input-src of <instr-1 is <port-1
       AND input-dest of <instr-1 is <reg-1 OR <reg-2
       AND object-type of <instr-2 is instruction
       AND source-type of <instr-2 is user-spec
       AND operator-type of <instr-2 is BUS_IN
       AND input-src of <instr-2 is <port-2
       AND input-dest of <instr-2 is <reg-3 OR <reg-4
       AND object-type of <instr-3 is instruction
       AND source-type of <instr-3 is prototype
       AND operator-type of <instr-3 is BUS_IN
       AND input-src of <instr-3 is <port-1
       AND input-dest of <instr-3 is <reg-1 OR <reg-2
       AND object-type of <instr-4 is instruction
```

```
       AND source-type of <instr-4 is user-spec
       AND operator-type of <instr-4 is BUS_IN
       AND input-src of <instr-4 is <port-2
       AND input-dest of <instr-4 is <reg-3 OR <reg-4
       AND binding of (<instr-1 & <instr-2) OR (<instr-1 & <instr-4)
                                          is strong-bound
       AND binding of (<instr-3 & <instr-2) OR (<instr-3 & <instr-4)
                                          is strong-bound
    THEN
       RM-SRC    ( <instr-1, input-src, <port-1 )
       RM-SRC    ( <instr-3, input-src, <port-1 )
       RM-PATH   ( <instr-1, input-src, <port-1 )
       RM-PATH   ( <instr-3, input-src, <port-1 )
       ADD-SRC   ( <instr-1, input-src, <port-3 )
       ADD-SRC   ( <instr-3, input-src, <port-3 )
       ADD-PATH  ( <instr-1, input-src, input-dest )
       ADD-PATH  ( <instr-3, input-src, input-dest )
       RPL-INSTR-SRC ( <instr-1, input-src, <port-3[INDEXED] )
       RPL-INSTR-SRC ( <instr-3, input-src, <port-3[INDEXED] )
```

```
/*************************************************************************
/*
/*      Serial to Parallel Input Rule
/*
/*************************************************************************

DESCRIPTION: This rule is closely related to the Half word to Word rule.
             Where the Half word to Word rule modified the input structure
             The Half word to Word rule modified the circuit's architecture
             such that two registers no longer had to contended for the same
             bits of an input port. This rule completes the modification by
             enabling both registers to read the input port during the same
             clock cycle.  This rule works by determining if the user-spec
             loads two registers during the same clock cycle from a shared
             input port. If the prototype has two registers sharing the same
             input port, but reading their input over successive clock cycles
             then this rule fires. The component identified as the controller
             is modified such that both input registers received their load
             signals during the same clock cycle.


    IF object-type of <port-1 is PORT
       AND source-type of <port-1 is prototype
       AND IO of <port-1 is input
       AND object-type of <port-2 is PORT
       AND source-type of <port-2 is user-spec
       AND IO of <port-2 is input or input-output
       AND binding of <port-1 & <port-2 is strong-bound
       AND object-type of <reg-1 is register
       AND source-type of <reg-1 is prototype
       AND object-type of <reg-2 is register
       AND source-type of <reg-2 is prototype
       AND object-type of <reg-3 is register
       AND source-type of <reg-3 is user-spec
       AND object-type of <reg-4 is register
       AND source-type of <reg-4 is user-spec
       AND binding of (<reg-1 & <reg-3) OR (<reg-1 & <reg4) is strong-bound
       AND binding of (<reg-2 & <reg-3) OR (<reg-2 & <reg4) is strong-bound
       AND object-type of <instr-1 is instruction
       AND source-type of <instr-1 is prototype
       AND operator-type of <instr-1 is BUS_IN
       AND input-src of <instr-1 is <port-1
       AND input-dest of <instr-1 is <reg-1 OR <reg-2
       AND object-type of <instr-2 is instruction
       AND source-type of <instr-2 is user-spec
       AND operator-type of <instr-2 is BUS_IN
       AND input-src of <instr-2 is <port-2
       AND input-dest of <instr-2 is <reg-3 OR <reg-4
       AND object-type of <instr-3 is instruction
       AND source-type of <instr-3 is prototype
       AND operator-type of <instr-3 is BUS_IN
       AND input-src of <instr-3 is <port-1
       AND input-dest of <instr-3 is <reg-1 OR <reg-2
       AND object-type of <instr-4 is instruction
       AND source-type of <instr-4 is user-spec
       AND operator-type of <instr-4 is BUS_IN
       AND input-src of <instr-4 is <port-2
```

```
       AND input-dest of <instr-4 is <reg-3 OR <reg-4
       AND object-type of <statement-1 is statement
       AND source-type of <statement-1 is prototype
       AND instruction of <statement-1 is <instr-1
       AND object-type of <statement-2 is statement
       AND source-type of <statement-2 is prototype
       AND instruction of <statement-2 is <instr-3
       AND object-type of <statement-3 is statement
       AND source-type of <statement-3 is user-spec
       AND instruction of <statement-3 is ( <instr-2 & <instr-4> )

       AND object-type of <statement-4 is statement
       AND source-type of <statement-4 is prototype
       AND statement-type of <statement-4 is clock-step
       AND successor of <statement-1 is <statement-4
       AND successor of <statement-4 is <statement-2
       AND binding of (<instr-3 & <instr-2) OR (<instr-3 & <instr-4)
                                              is strong-bound
       AND binding of (<instr-3 & <instr-2) OR (<instr-3 & <instr-4)
                                              is strong-bound
    THEN
       RW-CTRL-SRC    ( <instr-3, input-dest, LOAD-ENABLE )
       RW-CTRL-PATH   ( <instr-3, input-dest, LOAD-ENABLE )
       ADD-CTRL-SRC   ( <instr-3, input-dest,
                        GET-CTRL-SRC (<instr-1, input-dest, LOAD-ENABLE) )
       ADD-CTRL-PATH  ( <instr-3, input-dest, LOAD-ENABLE )
       RW-CTRL-INSTR  ( <instr-3, LOAD-ENABLE )
       ADD-CTRL-INSTR( <instr-3, LOAD-ENABLE,
                        GET-CTRL-INSTR (<instr-1, LOAD-ENABLE ) )
       MOVE-INSTR     ( <instr-3, <statement-1 )
```

```
/*************************************************************************
/*
/*      Half Word to Word Output Rule
/*
/*************************************************************************

DESCRIPTION: This rule examines the output structure of the prototype to
             to determine if the prototype has a register whose contents
             is serially gated to an output port over successive clock
             cycles. Then the output architecture of the user-spec is
             examined to see if the corresponding registers is output is
             dumped to the output port in one clock cycle. If this is the
             case, then this rule fires. The action items of this rule will
             connect the prototype register to an output port to which it
             can write its output in one clock cycle. This rule also has
             a consistency check in it, if the destination port is an
             input-output port, then a tri-state latch is installed.



    IF object-type of <port-1 is PORT
       AND source-type of <port-1 is prototype
       AND IO of <port-1 is output
       AND object-type of <port-2 is PORT
       AND source-type of <port-2 is user-spec
       AND IO of <port-2 is output
       AND object-type of <port-3 is PORT
       AND source-type of <port-3 is user-spec
       AND IO of <port-3 is output or input-output
       AND binding of <port-1 & <port-2 is weak-bound
       AND binding of <port-2 & <port-3 is strong-bound
       AND object-type of <reg-1 is register
       AND source-type of <reg-1 is prototype
       AND object-type of <reg-2 is register
       AND source-type of <reg-2 is prototype
       AND binding of <reg-1 & <reg-2 is strong-bound
       AND object-type of <instr-1 is instruction
       AND source-type of <instr-1 is prototype
       AND operator-type of <instr-1 is BUS_OUT
       AND output-dest of <instr-1 is <port-1
       AND output-src of <instr-1 is <reg-1[INDEX]
       AND object-type of <instr-2 is instruction
       AND source-type of <instr-2 is prototype
       AND operator-type of <instr-2 is BUS_OUT
       AND output-dest of <instr-2 is <port-1
       AND output-src of <instr-2 is <reg-1[INDEX]
       AND object-type of <instr-3 is instruction
       AND source-type of <instr-3 is user-spec
       AND operator-type of <instr-3 is BUS_OUT
       AND output-dest of <instr-3 is <port-2
       AND output-src of <instr-3 is <reg-2
       AND binding of (<instr-1 & <instr-3) is weak-bound
       AND binding of (<instr-2 & <instr-3) is weak-bound
       AND object-type of <statement-1 is statement
       AND source-type of <statement-1 is prototype
       AND instruction of <statement-1 is <instr-1
       AND object-type of <statement-2 is statement
       AND source-type of <statement-2 is prototype
```

```
       AND instruction of <statement-2 is <instr-3
       AND object-type of <statement-3 is statement
       AND source-type of <statement-3 is user-spec
       AND instruction of <statement-3 is <instr-3
       AND object-type of <statement-4 is statement
       AND source-type of <statement-4 is prototype
       AND statement-type of <statement-4 is clock-step
       AND successor of <statement-1 is <statement-4
       AND successor of <statement-4 is <statement-2
    THEN
       RM-INSTR ( <instr-2 )
       RM-DEST  ( <instr-1, output-dest, <port-1 )
       RM-PATH  ( <instr-1, output-dest, <port-1 )
       RM-SRC   ( <instr-1, output-src,  <reg-1[INDEX] )
       ADD-SRC  ( <instr-1, output-src,  <reg-1 )
       RPL-INSTR-SRC ( <instr-1, output-src, <reg-1)
       RPL-INSTR-DEST ( <instr-1, output-dest, <port-3)
        IF ( IO-TYPE of <port-3 is input-output) THEN
          BEGIN
            ADD-COMPONENT ( TRI-STATE-LATCH )
            ADD-DEST ( <instr-1, output-dest, INPUT_OF ( TRI-STATE-LATCH ))
            ADD-DEST ( <instr-1, output-dest, <port-3 )
            ADD-PATH ( <instr-1, output-src, INPUT_OF ( TRI-STATE-LATCH ))
            ADD-PATH ( <instr-1, output-dest, OUTPUT_OF ( TRI-STATE-LATCH ))
            ADD-CTRL-SRC  ( <instr-1, output-oest,
                            GET-CTRL-INSTR (<instr-1, LOAD-ENABLE ) )
            ADD-CTRL-DEST ( <instr-1, output-oest, TRI-STATE-LATCH ) )
            ADD-CTRL-PATH ( <instr-1, output-dest, FUNCTION-ENABLE ) )
            ADD-CTRL-INSTR (<instr-1, FUNCTION-ENABLE, "signal_out") )
          END
        ELSE
          BEGIN
            ADD-DEST ( <instr-1, output-dest, <port-3 )
            ADD-PATH ( <instr-1, output-src, output-oest )
          END
```

GLOSSARY

Architectural Properties List	A procedural representation of a circuit's architecture. The Architectural Properties List is a list of CONNECT, EXPAND, and PLACE statements which are used to produce the schematic for a circuit. The Architectural Properties List also serves as a program to the Instantiator which controls the design of a circuit.
Clock_Begin	Statement type which is used to begin a behavioural specification.
Clock_End	Statement type which is used to end a behavioural specification.
Clock_Step	Statement type which is used to specify clock periods in a behavioural specification.
CONNECT	A layout statement which is used to create a conductive link between the terminals of two components.
Dependency Attribute	A count of the number of behavioural features in the frame header that depend on a specific architectural feature.
Dependency Network	A network of pointers which is used to show which architectural features of a circuit are required to implement its behavioural features.
Designer	The rule based portion of the proposed design system which is used to compare circuit behaviour and modify circuit architecture.
Design Slot	A slot in the prototype frame that is used to represent the behaviour of a component that is required to implement the circuit behaviour described by the frame header.
EXPAND	A layout statement which is used to instruct the Instantiator to develop an architecture for a component that will implement the behaviour specified by its Design Slot.
Frame Body	The portion of a prototype frame that describes the components required to implement the behaviour described by the frame header (the design slots) and how those components are to be interconnected (the architectural properties list).
Frame Header	The portion of a prototype frame that describes the behaviour of the circuit the frame represents.
Instantiator	The stacked based portion of the proposed design system which controls the design process. The Instantiator follows the layout statements given in an Architectural Properties List.

Instruction	The basic unit of the augmented PASCAL language used to represent circuit behaviour. An instruction represents one primitive functional transform that the circuit can perform on a set of operands in one clock cycle.
Matching Rule	The set of rules which are used to find matches between the behavioural features of a prototype and user-spec. The matched features are bound by linking them in the symbol table.
Object	An entity which has properties that can be tested by the system rule base. An object can be an instruction, statement, port, or register.
Port	The boundary through which a circuit can exchange information with the external world. A port has no storage associated with it.
Port Symbols	The list of declarations for the ports of a circuit within the program that describes the behavior of the circuit.
Property	An attribute of an object. The properties that can be possessed by an object depend on the type of the object.
Prototype	Term used to designate the source type of the objects that were obtained from the prototype frame during the application of the matching and transformation rules.
Prototype Frame	A structure used to represent how a circuit's behaviour is implemented by the circuit's architecture.
Prototype Manager	The portion of the proposed circuit design system which manages the data base of prototype frames.
Register	A storage element within a circuit.
Reg Symbols	The list of declarations for the registers of a circuit within the program that describes the behaviour of the circuit.
RT-TYPE	Classifies the type of register transfer operations a circuit performs.
Source Type	Specifies whether an object is part of the user-spec or is part of the prototype frame.
Statement	An aggregation of instructions that can be executed concurrently.

Statement Type	A classification of statement by function. Statements whose instructions consist of predicate instructions are classified as DECISION statements. Statements whose instructions consist of IO group or Functional group instructions are classified as ACTION statements. Statements used to specify timing intervals are classified as CLOCK-STEPs.
Strong Match	A strong match is declared when there is an exact match between the properties of a user-spec object and a prototype object.
Symbol Table	A table in which all objects from the behavioural descriptions for the prototype and user-spec are stored. Bindings established by the rules are recorded in the symbol table.
Transformational Rule	A rule which can be used to transform the behaviour of the prototype such that it can implement the user-spec behaviour.
User Spec	Term used to designate the source type of the objects that were obtained from the program that specifies the behaviour of a new circuit.
Weak Match	A weak match is declared when the properties of a user-spec and prototype object satisfy a set of constraints defined in a matching rule.

REFERENCES

[Abad85] M.S. Abadir, and M.A. Breuer, "A Knowledge Based System for Designing Testable VLSI Chips", IEEE Design and Test of Computers, Aug 1985 pp 56–68.

[Ado86] W.S. Adolph, H.K. Reghbati, and A. Sanmugasunderam, "A Frame Based System for Representing Knowledge About VLSI Design: A Proposal", Proc. 23rd Design Automation Conference, 1986, pp 671–676.

[Anc83] F. Anceau, "CAPRI: A Design Methodology and a Silicon Compiler for VLSI Circuits Specified by Algorithms.", Third Caltech Conference on Very Large Scale Integration, Computer Science Press 1983, pp 30–45.

[Barr81] A. Barr, E. Feigenbaum, "The Handbook of Artificial Intelligence Vol 1", William Kaufmann, Los Altos California, 1981.

[Bob75] D.G. Bobrow, T. Winograd, "An Overview of KRL, A Knowledge Representation Language", Artificial Intelligence.

[Fre79] P.E. Freidland, "Knowledge-based Experiment Design in Molecular Genetics", Rep. No. 79–771, Computer Science Dept., Stanford University. (Doctorale disseration.)

[Gam85] D. Gamble, "Overview of Computer Aided Design of VLSI Layout", unpublished manuscript, 1985.

[Gia85] N. Giambiasi et. al, "An Adaptive Evolutive Tool for Describing General Hierarchical Models, Based on Frames and Demons", Proc. 22nd Design Automation Conference, (1985) pp 460–467.

[Hay84] John P. Hayes, "Digital System Design and Microprocessors", McGraw-Hill Book Company, New York, 1984.

[IDEA84] Mentor Graphics, "Getting Started with Your IDEA System", Mentor Graphics Corporation, Beaverton, Oregon. 1984

[Joh79] D. Johannsen, "Bristle Blocks: A Silicon Compiler", Proc. 16th Design Automation Conference, June 1979, pp 310–313.

[Min75] M. Minsky, "A Framework for Representing Knowledge", in The Psychology of Computer Vision, P. Winston (Ed.). New York, McGraw-Hill, 1975.

[Mit84] T. Mitchell, L. Steinberg, and J. Shulman, "VEXED: A Knowledge-Based VLSI Design Consultant.", Tech. Report LCSR-TR-57, Dept. of Computer Science and Laboratory for Computer Science Research, Rutgers University, New Brunswick, New Jersy, 1984.

[Pol57] Polya, G. "How to Solve It.", Doubleday Anchor Books, N.Y. 1957.

[Res84] A. L. Ressler, "A Circuit Grammar for Operational Amplifier Design", MIT Doctorale Thesis in EECS 1984.

[Sai81] T. Saito, T. Uehara, and N. Kawato, "A CAD System for Logic Design Based on Frames and Demons", Proc. 18th Design Automation Conference, (1981) pp 451-456.

[Sno78] E.A. Snow, "Automation of Module Set Independent Register-Transfer Level Design", PhD Thesis, Dept. Electrical Engineering Carnigie Mellon University, Dec. 1978.

[Sou83] J. Southard, "MacPitts: An Approach to Silicon Compilation", IEEE Computer, Dec. 1983, pp 74-82.

[Stef81a] M. Stefik, "Planning With Constraints: (MOLGEN: Part 1)", Artificial Intelligence 16, 1981, pp 123-140.

[Stef81b] M. Stefik, "Planning and Meta Planning: (MOLGEN: Part 2)", Artificial Intelligence 16, 1981, pp 141-170.

[Stein84] L. Steinberg, and T. Mitchell, "A Knowledge Based Approach to VLSI CAD, The REDESIGN System", Proc. 21st Design Automation Conference, 1984, pp 412-418.

[Thom83] Donald E. Thomas et. al., "Automatic Data Path Synthesis", Computer Dec. 1983, pp 59-70.

[Van84] Van E. Kelly, "The Critter System - Automated Critiquing of Digital Circuit Designs", Proc. 21st Design Automation Conference, 1984, pp 419-425.

[Wal84] P. Wallich, "On the Horizon: Fast Chips Quickly.", IEEE Spectrum, March 1984, pp 28-34.

[Win84] P. Winston, and B. K. P. Horn, "LISP Second Edition", Addison-Wesley, Reading Mass. 1984.