ROYAL MELBOURNE INSTITUTE OF TECHNOLOGY

DEPARTMENT OF COMMUNICATION AND ELECTRONIC ENGINEERING

<u>TELECOM RESEARCH CONTRACT 68249</u>

SDL - VLSI PROJECT

REPORT No. 3

SEPTEMBER 1986

<u>RESEARCH AND DEVELOPMENT MEMORANDUM No. 112Ø46M</u>

BY THE

<u>MEMBERS OF THE SDL - VLSI PROJECT TEAM</u>

P. BECKETT, K. BOLCH, S. CLEARY,

G. EGAN, R. FERGUSON, L. JACKSON, J. PAU

SDL-VLSI TELECOM REPORT No. 3

INDEX

1.Ø Introduction

This is the third report from the SDL-VLSI team and its main aim is to outline a system for automatically generating VLSI circuits given an initial behavioral description specified in SDL and to perform some assessment of the effort required to implement such a system. With this in mind the report has been divided into six main sections:

(i) - a system overview and the rationale behind the approach taken;

(ii) - a guideline for the specification of the interfaces to (and between) each of the processes;

(iii) - an outline of how each of the processes work;

(iv) - an estimate of the time, cost and equipment required for the specification and implementation of the proposed system;

(v) - a summary of the options available for the major example to be given in the next report;

(vi) - conclusions and a list of research areas which require further study.

2.Ø System Overview

This report will endeavour to outline the major portions of a system for automatically translating CHILL code to a VLSI circuit. In conjunction with Melba, which translates SDL code to CHILL, a full SDL to VLSI system is proposed. At present elements of the system are still in a state of flux, with many secondary details remaining to be established. The current view of the system will be presented in this report - it will undoubtedly change as problems become apparent and new concepts are developed. An example application, to be presented in the next report, should prove to be useful for the development of the system.

2.1 The Rationale

Solutions to problems in such areas as placement and routing are computationally expensive to produce. It is generally impossible to derive optimal results, except in trivial examples, because the potential solution set is extremely large. The search for solutions can be constrained to the more productive paths using heuristics (or rules of thumb) producing good, although not necessarily optimal, results in reasonable time. In the past these heuristics have generally been embedded into the program generating the solutions but the current trend is to keep the rules separate from the program. Such systems are called **rule based** systems.

A silicon compilation system (or a system for the automatic production of the physical layout of a circuit from an initial high level behavioural description) usually contains several translation steps in which the solution set increases exponentially with search depth. The brute force approach to optimization in these areas is often not successful [63]. The use of rules (even very simple ones) can drastically reduce the search space. This is the major justification for using a rule based system.

2.2 Standard Cell vs. Cell Generators

The current investigation by the SDL-VLSI project team has been centered on a discussion of the various methods of silicon compilation, which were covered in the first 2 reports. The techniques covered to date are based on a 'standard cell' building block which uses either a library of standard logic cells or a 'generator' which draws on a description of the standard cell to derive the required logic function.

The library structure is a major stumbling block in the standard cell approach. The information stored has to be both created and maintained and as the data base becomes larger, and therefore more flexible, it becomes more difficult to use. Standard cells are usually technology dependent. Hence to cater for many different classes of technology there will be a need for large and varied data base.

The behavioural or structural description of the design is moulded to fit the available hardware structures that are described within the library. The solution limits the transformation of the description to only those structures available. The translation process is constrained by its cell base within the library, which automatically limits design flexibility.

Up to now the translation process has usually ignored the application of top-down design experience when creating the floorplan. Within the standard cell approach a common implementation involves the use of specialized compilers. These compilers have a fixed notion of floorplan in which the designer has limited freedom to explore the design space. Some are categorised as 'specialist' standard cell systems and take into account a particular architecture and floorplan style to automate the design process e.g. 'MacPitts' optimized for signal processing.

If there is no predefined floorplan structure, a great deal of effort can go into computing the placement and routing between cells. These tasks are further complicated by the lack of ability to manipulate the structure of the individual cells, e.g. making them long and thin or short and fat. These compilation systems, once constructed, are quite inflexible and do not allow large changes to the method used in determining the solution to be made easily. They have usually been created for a rigid environment, which is in itself what the library structure leads to.

One solution to the above problems is to employ a system that uses a set a rules to determine the path of translation. This is commonly known as a **knowledge based** system. The environment created by such a system would be much more flexible than a standard cell system. Instead of forcing the structure to use existing cells within the library, it would enable the description to help create the neccessary silicon structures. The rules would also enable the description of the module generators to utilize technology independent algorithms. The same rules would then be used to generate the specific technology structures of the modules thus reducing the physical size of the library.

Such a system would have a different rule base for the translation of behavioural or structural descriptions to silicon. This library of rules gives the system much more fexibility to handle variations in technology. A further advantage is that in the development stages of the system the designer can add rules which change approaches in order to solve problems that may occur. Thus, once the system has been developed, the solutions generated are then solely dependent upon the knowledge base, which is a dynamic environment.

The rule based system is not an expert system, it is simply a set a rules to tell the system how to go about the translation. The main advantage in the translation of SDL to VLSI is that the domain of each process within the system is well defined and thus should be able to support the neccessary translation. In this context, it will use the previously defined subset of CHILL and Zeus.

2.3 The Hardware/Software Interface

Although the main thrust of this project at present is to synthesize hardware components from a description in CHILL, it is envisaged that integration of hardware and software elements of a CHILL program will be necessary in a large system design.

One method for hardware and software to co-exist in a system is to attach the hardware units as I/O devices. However, as noted by Rattner [64], this approach was found to be unnatural and complicated by operating system issues especially when these hardware units were not meant to provide I/O functions. Therefore, it is more appropriate to view these units as hardware processes. These hardware processes can communicate with software processes by exchanging messages. Amongst the interprocess communication mechanisms available in CHILL, the SIGNAL would appear to be the most suitable in this case. The hardware processes would be constrained to use the same semantics and interfaces as the original software processes. In addition, each hardware process should not contain any globally accessible objects in order to preserve a 'clean' interface.

An example of such a system is the Intel 432 microprocessor. Its object-oriented architecture provides interprocess communications via 'communication ports' with all communications being based on messages. The use of this type of system would appear to reduce the complexity of the hardware/software interface.

2.4 The Translation System

Figure 2.4.1 presents an overview of the entire system. The main input is a segment of CHILL code which may consist of several CHILL processes. The code may be hand crafted or it may be obtained from the Melba system. The other input is a set of design constraints which physically limit the type of circuit that may be constructed. The principle constraints are power, area and speed but others are possible. The **F**low **C**ontrol and **C**onstraint **P**ropagation block (**FCCP**) manufactures more detailed constraints using a rule base (and perhaps, past experience). These are passed to the CHILL to Zeus compiler to be used as a basis for the architectural design of a piece of hardware capable of performing functions equivalent to those performed by the segment of CHILL code.

The architectural design is expressed in Zeus while the temporal behaviour of the design is specified in a separate control language. This language could be an extension of Zeus. When a trial design is completed, the CHILL to Zeus compiler communicates details about the design to the FCCP to allow it to fabricate detailed constraints to be used as guidelines by the lower levels of the translation process.

The Zeus description is expanded into an Intermediate **T**emporary **L**anguage (**ITL**) which effectively consists of a list of blocks and their interconnections. The ITL to CIF process subsequently uses this list of blocks and their corresponding constraints (obtained from the FCCP) to produce a trial physical design. This is the only technology dependent level.

System Overview

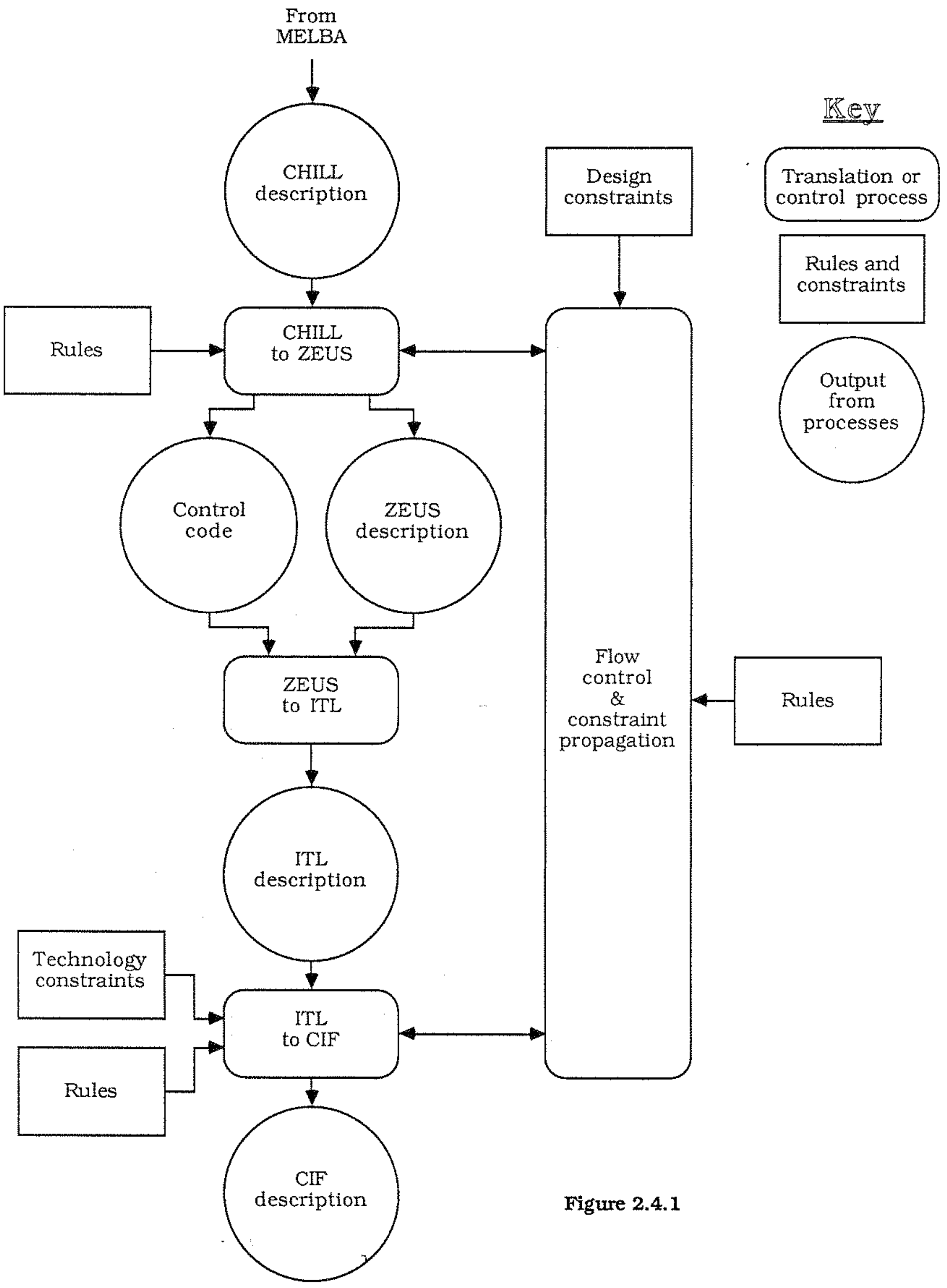


Figure 2.4.1

If the design does not meet specifications then rules within the FCCP are invoked to make power/area/speed tradeoffs. These tradeoffs are used to derive a new set of constraints which are used to guide the CHILL to Zeus compiler towards a design which has a greater likelihood of meeting specifications. This iterative process is repeated until the specifications are met or until all the design possibilities capable of being created by the system are exhausted. In the latter case the designer must reformulate the specifications or redesign the segment of CHILL code.

3.Ø Interface Specification

Any significant project without concrete specification and sufficient planning is unlikely to succeed. One authority estimates that a complex project would typically consist of one third planning and specification, one third implementation and one third testing and commissioning [46]. For this reason a chapter is presented with guidelines for the specification of the interfaces to and between each of the processes.

3.1 Design Constraints

The CCITT Specification and Description Language was designed to allow for the behavioural description of telecommunications systems. However, it does not contain constructs for the specification of physical constraints (such as power, cost and size) - presumably these attributes are specified by other means when contracts are tendered. Since SDL provides no method for specifying such constraints and guidelines are required to help the SDL-VLSI system generate reasonable solutions, then some method must be provided to enable their specification. There are at least two possible methods - the first is in the form of a specialized language and the second is in the form of an extension to SDL. Useful constraints which could be imposed are:

- the number of I/O LINES
- the POWER consumed
- the AREA used
- the CYCLE TIME of each control step

Each of these constraints could be specified for every process within a module. Other more specific constraints could also be imposed if greater control over the generation process was desired. Note that the last constraint is cycle time rather than absolute time. **Cycle time** relates to the maximum delay through any path in any given state. Assuming a synchronous clocked system, then this delay time determines the maximum frequency at which a process may be clocked. An exception to this is the case in which individual clock cycles can be shaved to the critical path delay during that cycle. It is not possible to determine the absolute time a process will take because, in general, this will depend on the specific data to be processed. It may be possible to run the process on some 'typical' data and thereby determine which sequence of operations is the most beneficial to optimize. However, this would be a major task in itself and is beyond the scope of this report.

3.1.1 Design Constraint Language

Of the two suggested methods for specifying design constraints, the specialized language is probably the easiest to implement. Such a design constraint language can be fairly simple. It should allow for selective optimization of parameters within specific modules and should be directly related to the SDL code. The syntax of one possible language is presented in appendix 1.1. Fig. 3.1.1 shows a typical use of this language for a hypothetical module which contains three processes - **Stack**, **In_Buffer** and **Out_Buffer**. In this figure the **process identifier** names (**Stack**, etc.) refer directly to the corresponding SDL process names. The **dimensions** are specified in some standard units i.e. AREA in lambda^2, POWER in mW and CYCLE TIME in nS. The MAXIMIZE/MINIMIZE statement is used to maximize/minimize some parameter given that all other constraints are not exceeded. Only one MAXIMIZE/MINIMIZE statement is allowed per **block**. Using the dimensions suggested the MAXIMIZE statement and the '>'

operator are nearly useless but were included for generality.

Design Constraint Language Stack Example

```
CONSTRAINT  Stack_Example;

        CONSTRAINT In_Buffer;
            POWER < 100;
              MINIMIZE CYCLE TIME;
        END In_Buffer;

        CONSTRAINT Out_Buffer;
            POWER < 100;
             MINIMIZE CYCLE TIME;
        END Out_Buffer;

        CONSTRAINT Stack;
            POWER < 50;
             MINIMIZE AREA;
        END Stack;

         AREA(Stack + In_Buffer + Out_Buffer) < 1000000;
        I/O LINES < 40;
         MINIMIZE CYCLE TIME;
END  Stack_Example.
```

Figure 3.1.1

3.2 CHILL Subset

The previous report [45] briefly described the limitations on the CHILL code which must be enforced to ensure that reasonably efficient hardware can be generated. It is important that this subset is formally specified so that the CHILL to Zeus compiler designer (and the Melba abstraction library designer) know exactly what CHILL constructs can be used. The CHILL subset should be specified using EBNF (**E**xtended **B**ackus **N**aur **F**ormat [62]) or preferably using railroad diagrams. The syntax should be formulated to specifically exclude such constructs as recursive procedure calls and other types disallowed under the guidelines produced in the SDL-VLSI Project Report No. 2 [45]. Redundant and esoteric features could also be excluded to greatly ease the burden on the compiler writer.

3.3 Rule Base

Three of the four processes of the system possess a rule base as an adjunct in their specified task. The rule base of each of these processes should be in the same format. A standard has not been fixed but a rule will generally be in the following form:

```
IF    condition
      boolean operator    condition
              .
              .
THEN
      consequence;
      consequence;
              .
              .
```

If the three rule bases are in a standard form then standard tools can be devised for checking the consistency of the rule base and for optimally ordering them to minimize condition evaluation. Tools could also be developed for updating the rule base. The rule interpreter for each of the processes would also be similar thus reducing the amount of original work required.

3.4 Zeus and the Control Code

3.4.1 The Control Code

Zeus is a modern structured HDL which allows the succint description of hardware. However, it is mainly a structural description language and although behavioural descriptions are possible they tend to be rather verbose. For example, a simple traffic light controller specification is given in figure 3.4.1 (for a full description see [8]). Much of this specification is simply a behavioural description of the hardware in each of its possible states and it does not explicitly describe what type of hardware to generate. If the description was literally translated into hardware then the result would be a rather large and complex controller.

It seems pointless to obtain explicit knowledge of the type of control needed (from the CHILL to Zeus compiler) and then to translate this into an ambiguous intermediate form. The result of such a translation would be either to lose information or to overly complicate the design of the Zeus to ITL translator. If the control of the hardware was to be implemented using a bit slice approach then the behaviour would be more easily described using a set of active control points. For example, the behavioural part of the traffic light controller could look like figure 3.4.2. The syntax for this control code language has still to be defined but it should allow the specification of the type of controller, a definition of the fields in the control words, the jump condition(s), the jump address(es) and the active control points. It may also be advantageous for the controller

Zeus Traffic Light Controller

```
TYPE
        trafficlightcontroller=
COMPONENT(IN caronfarmroad, timeoutshort, timeoutlong: logical;
        OUT highwaylight, farmroadlight: ARRAY[1..w_colors] OF logical;
        OUT starttimer:logical) IS
SIGNAL_CONST
        states=(.highwaygreen, highwayyellow, farmroadgreen,
                farmroadyellow.);
CONST w_states=WIDTH(states);
        yes=one;
HARDWARE state: ARRAY[1..w_states] OF REG;
CONNECT
        IF EQUAL(state.o, highwaygreen) THEN
                highwaylight:=green;
                farmroadlight:=red;
                IF ANDg(caronfarmroad, timeoutlong) THEN
                        state.i:=highwayyellow;
                        starttimer:=yes
                ELSE
                        state.i:=highwaygreen
                END;
        END;
        IF EQUAL(state.o, highwayyellow) THEN
                highwaylight:=yellow;
                farmroadlight:=red;
                IF timeoutshort THEN
                        state.i:=farmroadgreen;
                        starttimer:=yes
                ELSE
                        state.i:=highwayyellow
                END;
        END;
        IF EQUAL(state.o, farmroadgreen) THEN
                highwaylight:=red;
                farmroadlight:=green
                IF ORg(NOTg(caronfarmroad), timeoutlong) THEN
                        state.i:=farmroadyellow;
                        starttimer:=yes
                ELSE
                        state.i:=farmroadgreen
                END;
        END;
        IF EQUAL(state.o, farmroadyellow) THEN
                highwaylight:=red;
                farmroadlight:=yellow;
                IF timeoutshort THEN
                        state.i:=highwaygreen;
                        starttimer:=yes
                ELSE
                        state.i:=farmroadyellow;
                END;
        END;
END;
END;
END;
```

Figure 3.4.1

Control Code for Traffic Light Controller

```
BEGIN
     highwaygreen: SELECT green TO highwaylight,
                   SELECT red TO farmroadlight,
                   UNTIL caronfarmroad AND timeoutlong;

    highwayyellow: SELECT yellow TO highwaylight,
                   SELECT red TO farmroadlight,
                   UNTIL timeoutshort;


    farmroadgreen: SELECT red TO highwaylight,
                   SELECT green TO farmroadlight,
                    UNTIL NOT(caronfarmroad OR timeoutlong);

   farmroadyellow: SELECT red TO highwaylight,
                   SELECT yellow TO farmroadlight,
                   UNTIL timeoutshort;

                   GOTO highwaygreen;
END;
```

Figure 3.4.2

to have the option of using subroutines. The control code language could be a separate but it would be better to keep the hardware and its behaviour together. Thus the control code language could be defined as an extension to the Zeus language. Information on the preferred method of implementing extensions is given in the Zeus report [43].

3.4.2 Zeus

Since a direct translation from the Zeus language to ITL is not technically difficult it should be a straightforward task to implement the whole Zeus language. The CHILL to Zeus compiler will not use all the Zeus constructs but the Zeus language may be used manually to test the lower portions of the system and also to enter specialized designs. All the Zeus constructs will be required if manual entry of designs is permitted. It may also be desirable to implement all the features of the Zeus language in order to keep the design flexible. At present it is considered that the interface between Modula 2 and Zeus (as defined in [43]) is not necessary. However, its implementation would not be difficult, especially if a Zeus to ITL translator was to be constructed as suggested in this report.

3.5 The ITL

The **ITL** is used to describe the blocks at the silicon level along with their interconnection. It is a very simple language which describes the generic structure type of the block and its connection to other blocks. In essence it is a schematic net list which makes use of the inherent regularity achievable on silicon. Blocks are formed from multiples of bit slices of some basic structure e.g. an 8 bit adder is 8 times a one bit adder. ITL also enables the ITL to CIF translation process to select the form of the block structure. As in the example of the adder, the selection is made between a serial or parallel structure.

Thus the ITL language is a list of blocks and their interconnects. Generic types are adders, subtractors, comparators, multipliers, shift registers, RAM, PLA etc., including discrete logic (Ands, Ors, Nands etc.).

The inclusion of a BUS block structure as an extra block enables the large communication paths associated with buses to be avoided. For example, if four blocks are connected via bus there are six paths to describe and identify when translating to silicon. Clearly, it is much easier to stipulate a bus block with 4 connections.

4.0 Processes

This section will outline in more detail the four major parts of the SDL to VLSI project. They are:

- **CHILL to Zeus**, essentially a classic compiler design but with a rule base to help make sensible hardware decisions;
- **Zeus to ITL**, a straightforward translation of Zeus code into ITL;
- **ITL to CIF**, a complex suite of programs including generation, placement, routing and simulation controlled by a rule base system;
- **Flow Control & Constraint Propagation**, coordinating the efforts of the other three and making high level constraint limited decisions.

An estimate of the effort required to implement these processes is given in section five.

4.1 CHILL to Zeus

This transformation process takes as input an algorithm or sequence of operations defined in a subset of CHILL, and as output produces Zeus code and a control code for specifying the behaviour of the hardware. Other inputs are design constraints and a rule base.

4.1.1 Constraints

Many constraints could be placed on a hardware design produced by this system, some of which are listed below:

- number of states (or control steps);
- the length of the path in each state (i.e. how many functional blocks in a path);
- the number of temporary registers;
- the number of I/O lines connected;
- the number of buses;
- the number of multiplexers;
- the number of arithmetic/logical units;
- the number of specialized registers (i.e. incrementing, decrementing, shifting).

These are not generated by the designer but are formulated by the FCCP from the specified design constraints (given in the Design Constraint Language, for example) and the FCCP rule base. The tradeoff between power, area and speed is only indirectly affected by these high level constraints.

4.1 CHILL to Zeus cont.

4.1.2 Translation

Figure 4.1.1 shows an overview of the CHILL to Zeus compiler system. Given a segment of CHILL code the compiler will perform syntactic and semantic analysis (according to the predefined CHILL subset) and generate some form of intermediate code. This could be three address code [47] or some other intermediate form more suited to the unique properties of the target machine. Simultaneously, the compiler generates a list or tree of operations and data transfers that must be performed.

The next stage of the process is to bind the operations to specific functional units (or specialized registers) and to bind the data transfers to specific buses or multiplexers. To do this the rule interpreter looks at the design constraints and the intermediate code then applies the appropriate rules to determine the best physical implementation of the desired operations. For example, suppose a data transfer was required from one register to another. If a path already exists between the registers then no action is necessary. If the input to the target register is already bound then the other register must also be bound to the path at that input. If the input to the target register is unbound then a path must be created (either a bus, multplexor or direct connection). The rule below illustrates the case where the input to the target register already has a direct connection to another register.

CHILL to Zeus

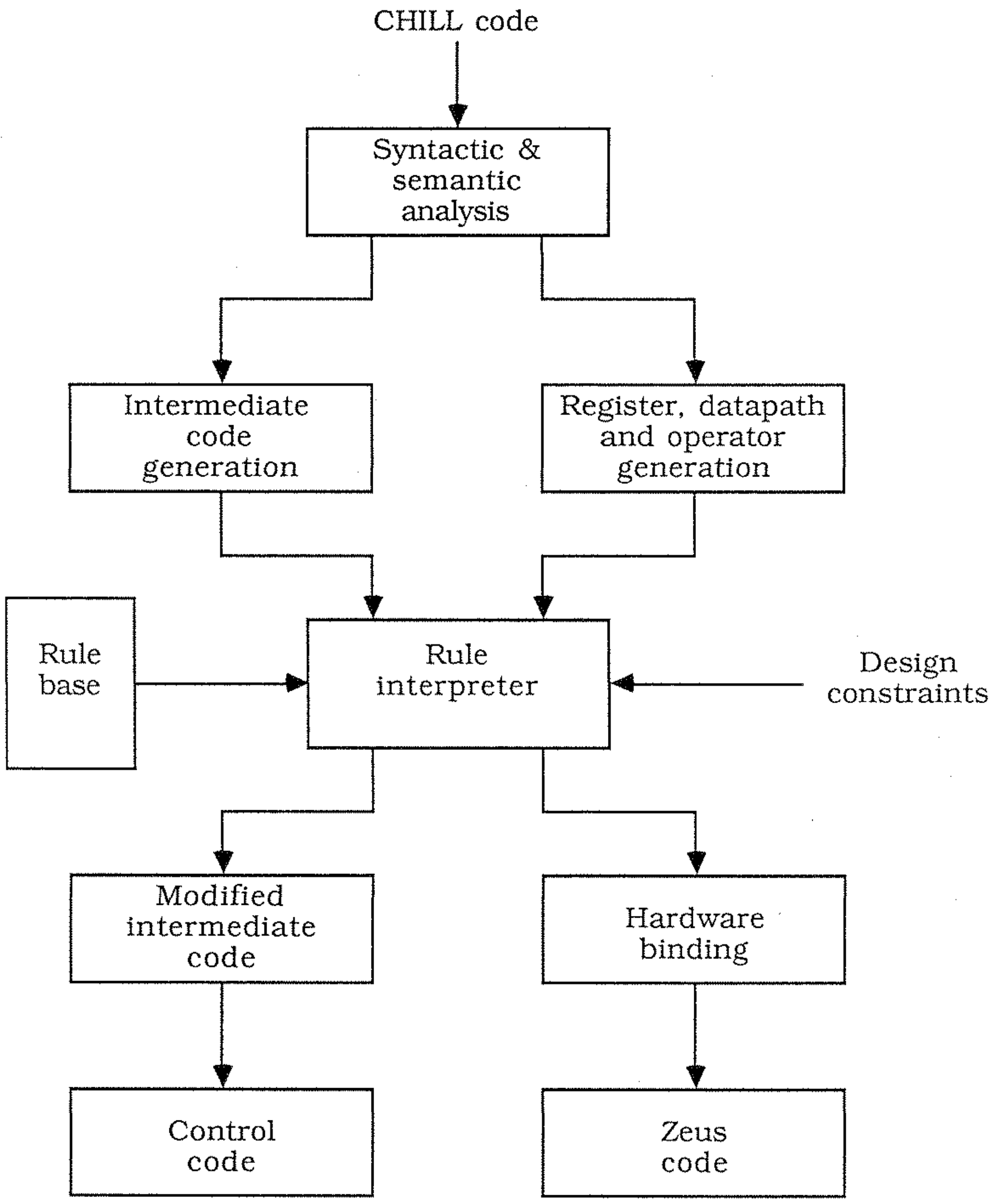

CHILL code
Syntactic & semantic analysis
Intermediate code generation
Register, datapath and operator generation
Rule base
Rule interpreter
Design constraints
Modified intermediate code
Hardware binding
Control code
Zeus code

Figure 4.1.1

```
IF data_path_required_to_register
   AND NOT path_exists(from_register,to_register)
   AND num_inputs(to_register) = 1
   AND input_type(to_register) <> multiplexer
   AND NOT max(num_of_multiplexers)
THEN
   RESET data_path_required_to_register;
   generate_multiplexer(from_register,to_register);
   increment(num_of_multiplexers);
END IF;
```

A large number of rules would be required to cater for all possibilities.

4.2 Zeus to ITL

This process is, perhaps, the simplest of the four. The syntax of the Zeus language is similar to Modula-2 and the lexical and semantic analysis should present no problem. Figure 4.2.1 shows a view of how the translation process would proceed. The Zeus language is slightly different to conventional languages in that it contains control variables and statements which control the generation of the hardware (i.e. statements similar to psuedo-ops and macros in an assembler). For this reason Zeus must first be translated into some intermediate code which is then run (or interpreted) to produce the final ITL code.

The simplest method of implementation would be to first translate the Zeus code into Modula-2 code (with all the control variables being translated into Modula-2 variables). This code would then be compiled and run (automatically) with the output of the program being ITL. Implementing a Modula-2 to Zeus interface (see [43]) has an added advantage in that it would be a simple matter to link user defined Modula-2 modules to the code produced by the Zeus translator. Predefined Zeus modules (previously translated to Modula-2) could also be linked at this stage. These predefined modules could be used to simplify the translation process.

Zeus to ITL Overview

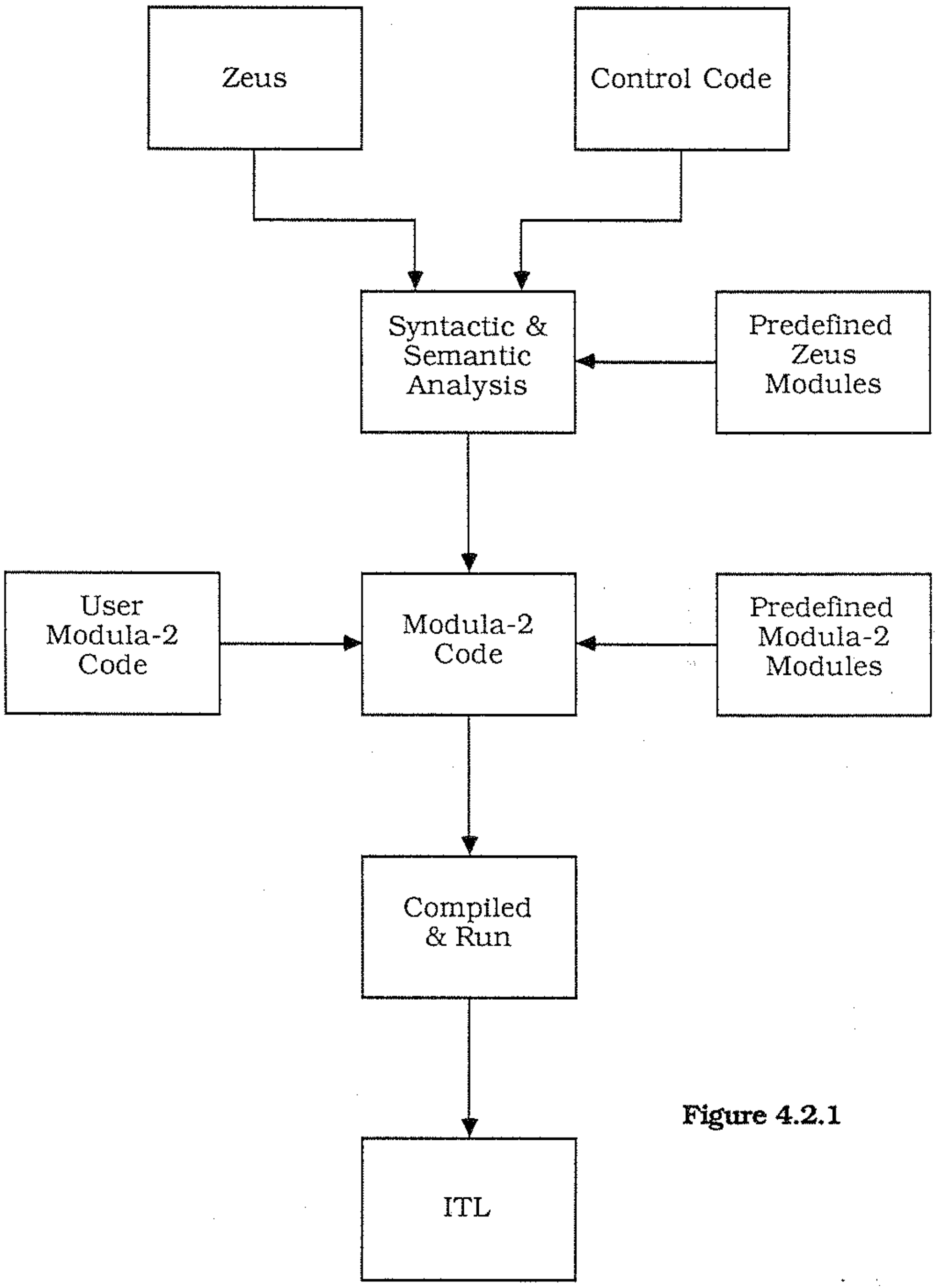


Figure 4.2.1

4.3 Flow Control and Constraint Propagation

This process oversees the operation of the whole of the translation system. With the help of general (technology independent) architectural knowledge, it is able to make decisions about what constraints should be placed on the various stages of the translation process. Figure 4.3.1 shows an overview of this process.

A typical translation would proceed through a number of steps as follows:

(i) The FCCP accepts the specified design constraints and from these generates 'reasonable' limits to be placed on the parameters controlling the generation of the architectural design produced by the CHILL to Zeus compiler. Such limits could include the number of multiplexers and other functional units (see section 4.1.1 for further details).

(2) Once the CHILL to Zeus compiler has accepted these constraints two results are possible:
- the compiler generates a trial design that meets the constraints given to it by the FCCP (i.e. the constraints were reasonable)
- the CHILL to Zeus compiler cannot meet one or more of the given constraints.

If the constraints placed on the CHILL to Zeus compiler can not be met then the FCCP must modify them. Generally this will involve a relaxation of the failed constraint and perhaps a corresponding tightening of other constraints. This process is repeated until the CHILL to Zeus compiler manages to produce a trial design. The estimation of good constraints from the design goals and the size and nature of the CHILL code would be an area with excellent potential for experiments in machine learning.

Flow Control & Constraint Propagation

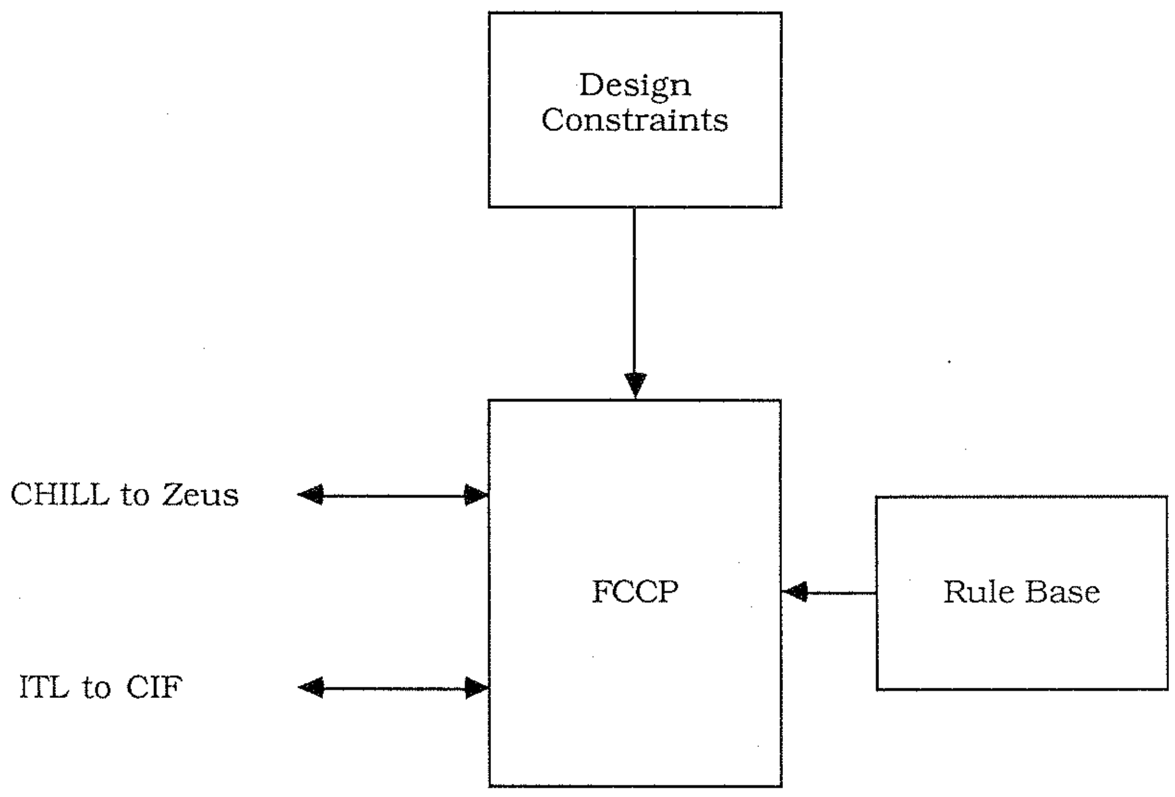


Figure 4.3.1

4.3 Flow Control and Constraint Propagation cont.

When the compiler has finished a trial design it passes back to the FCCP a list of paths in each state (eg. register -> multiplexer -> ALU -> multiplexer -> register). These paths are then combined with the timing constraints derived by the FCCP and, together with the area and power specifications, they form a set of constraints which are passed to the ITL to CIF process. Again, the ITL to CIF process may either generate a design which is within the given bounds (in which case we have a design completed to specifications) or some given constraint can not be met.

The failure may either be one of power, area or speed or perhaps some other criteria such as testability. As a results the FCCP will try to loosen the restriction on the failed constraint by tightening the bounds on other blocks within its field. This will allow some flexibilty for the critical block but keep the total limit over that field constant. For example, a path with five blocks in it may have a total path delay constraint of 100 nS. The FCCP may try to impose a 20 nS limit on each of the blocks but find that one block can not be kept within this bound. It may then relax the 20 nS limit on the failed block to 30 nS and confine the other blocks to 17.5 nS each. This type of procedure would continue until all specifications are met or it becomes obvious that the restrictions are too tight. In the latter case rules are applied within the FCCP which allow it to make power/area/speed tradeoffs at an architectural level.

These architectural level tradeoffs are expressed as new limits to be given to the CHILL to Zeus compiler. In this way a new design with a greater probability of meeting specifications is generated. The whole process is then repeated. If no power/area/speed tradeoffs can be made (i.e. all constraints are exceeded), the offending parameters can be identified. The designer would then be able either to relax the constraints or, using his superior design knowledge, guide the SDL-VLSI system with more explicit constraints. Hopefully, the designer will also be able to formally express his reasoning for these modified constraints to be incorporated into the FCCP rule base.

4.4 ITL to CIF

The **ITL** to **CIF** translation process is shown in figure 4.4.1. This process relies heavily on the top-down design strategies and an automatic floorplan generation such as is described in [60] & [61]. In the system overview it was suggested that to achieve best results with a silicon compiler, the translation process should assist the creation of silicon structures. Figure 4.4.1 shows the use of a hierarchical rule base to achieve this.

The ITL description has been defined as a netlist description between blocks. The ITL generates the floorplan graph by declaring the blocks as nodes and the interconnect as paths. From this floorplan graph and a process of selection a simulation description is generated.

The design constraints and specification in conjunction with the rules devised for floorplanning acts to select an available structure for each block. This block is then assigned some internal physical constraints, e.g. the general shape(such as long and thin), positionof signal and power connections etc. Included in this iterative selection and generation process are the results of any previous generation attempts. If all attempts to generate a solution which fits the design specifications fail, this information is passed back to the major controlling process as shown in the system overview.

Additional information generated by the above process can be used in conjunction with a predefined rule library to simulate the approximate behaviour of the blocks and their interconnects.

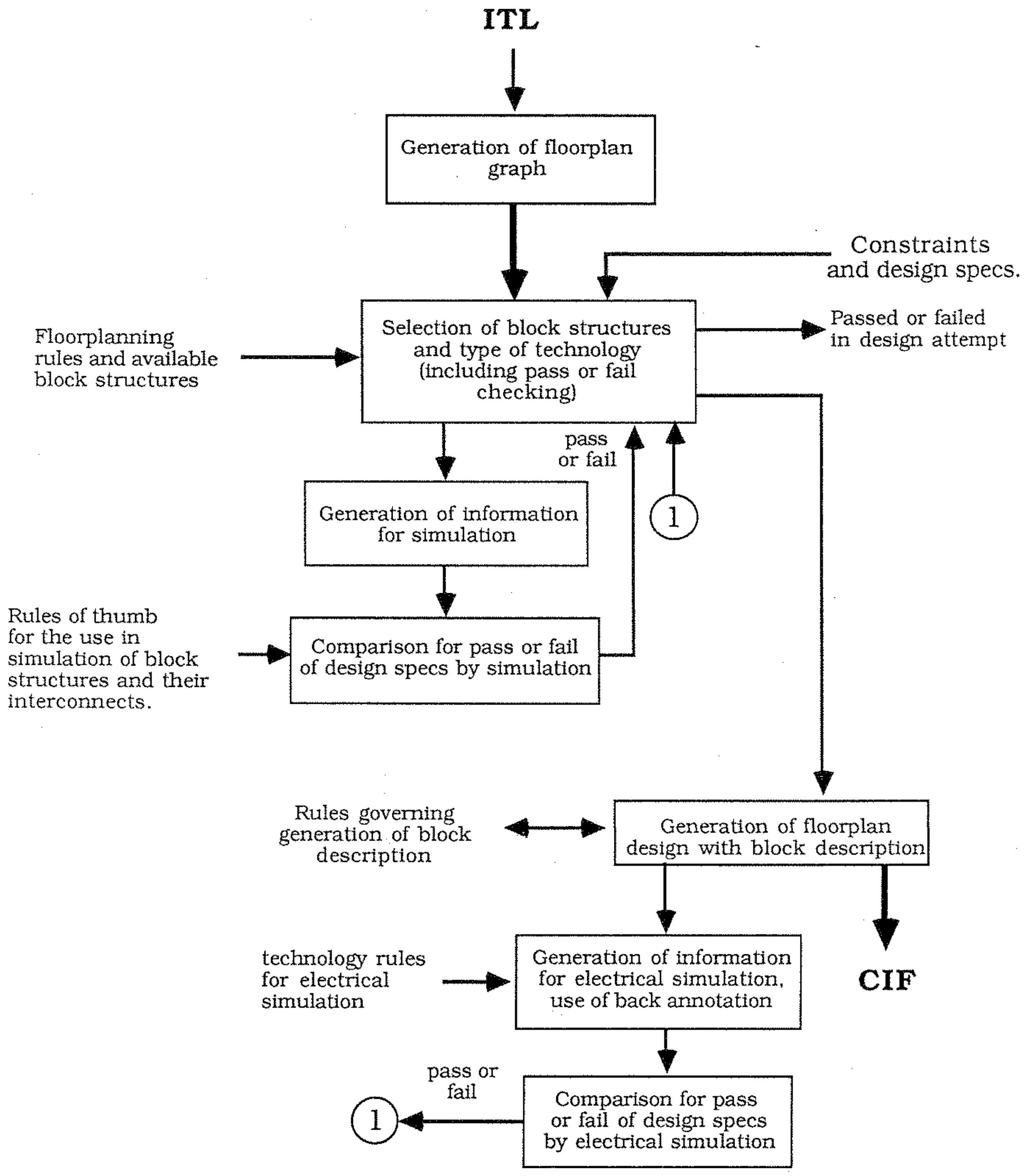

ITL
Generation of floorplan graph
Constraints and design specs.
Floorplanning rules and available block structures
Selection of block structures and type of technology (including pass or fail checking)
Passed or failed in design attempt
pass or fail
1
Generation of information for simulation
Rules of thumb for the use in simulation of block structures and their interconnects.
Comparison for pass or fail of design specs by simulation
Rules governing generation of block description
Generation of floorplan design with block description
CIF
technology rules for electrical simulation
Generation of information for electrical simulation, use of back annotation
pass or fail
1
Comparison for pass or fail of design specs by electrical simulation

Fig. 4.4.1

4.4 ITL to CIF cont.

This rule library is a list of 'rules of thumb' which is derived from the designer's knowledge. The simulation results achieved are then compared with the goals set for the simulation by the selection process. The results of the comparison are passed back to the selection process for evaluation.
When the selection process has chosen a design description that meets the design specifications it creates a description for the generation of actual hardware. Once the block modules have been created, placed and routed a full simulation is performed using geometric information from these processes. Simulation can be achieved in a hierarchical fashion if the modules are simulated individually followed by their interconnections. Failure to meet the given specifications at this point should affect only the module placement and routing. Optimization can proceed at this level until an acceptable result is obtained.

5.Ø Time & Cost Estimation

The estimation of time and costs at this stage will necessarally be somewhat imprecise , however, as planning and specification progress a refinement of these estimates will be possible.

5.1 Rules

The development times associated with construction of the rule library are still very difficult to estimate, for although there is some idea of what the rule base should look like, there has not been enough investigation to determine their exact nature. The rules for the ITL to CIF process would consist of:

- floorplanning and available block structures
- rules of thumb for blocks and interconnects
- rules governing generation of block descriptions.

For the CHILL to Zeus compiler:

- when to use buses or multiplexers
- combining of functional blocks
- when to use specialized registers.

For the FCCP:

- knowledge about how to make power/area/speed tradeoffs.

As noted in [63] the acquisition of rules is a major task. It was observed previously that the knowledge base is, by definition, a dynamic structure which will be updated continously.

5.2 Equipment & Support

A Unix environment would be useful for compiler development as it provides the **yacc** system for compiler design and **lex** for lexical analysis. A profiler would be useful for determining which parts of a program should be optimized if higher speed was required. Either Pascal, Modula-2 or C should be provided to write the compiler - with Pascal or Modula-2 being preferred unless speed was of paramount importance. If the full Zeus system was to be implemented then a Modula-2 compiler would be essential as Zeus has the option of importing Modula-2 blocks.

At the low level graphical capabilities are required to enable the display of placement and other information that is best presented pictorially. The languages used would be C, Pascal and Lisp and it would be neccessary to have a compiler for each language.

At the low level of fabrication there will be a requirement for the development tools aimed at the block generators, suchas a technology independent design rule checker and possibly an electrical rule checker. Also a graphical editor may be advisable for examination and modification of actual circuits.

The simulators used in the system would need to be considered very carefully for they are essential in the selection and design process. As the SDL-VLSI translation proceeds it would desirable to simulate at each level, especially when developing the compilation system. This implies that simulators would be required at each of the following levels: CHILL, Zeus, Logical, Switch Level and Electrical. It would be convenient if the simulation package could handle module and heirarchical description, as well as a graphical representation and be ableto support user interaction. A software package similiar to that provided by Mentor Graphics would be suitable as it is self documenting.

The hardware required to run this software should be based on a VM machine with at least 4 Mbytes of main memory to enable the use of LISP. It should also support reasonably high resolution graphics. A Sun-3 16Ø/C would be an appropriate machine.

6.Ø Major Example Proposal

Proposal of a subsystem as a trial application for the SDL/VLSI CAD system.

The IEEE 8Ø2 Token Ring Network is under consideration for evaluating the CAD system during the next phase of this project. This network is suggested because it is well documented and a specification is readily available. Examination of this token ring network, shows it to be too complicated to be implemented completely at this stage. A simplified version which consists of several interacting processes is under consideration. A design with multiple processes should provide interesting insights into the problems associated with interprocess communication.

7.Ø Areas of Further Research & Conclusions

Areas of Further Research

This report has identified at least five areas of interest that require major investigation to be fully explored. They are:

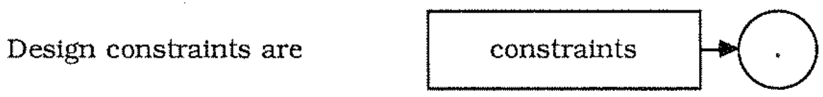
- The hardware/software interface, i.e. how do hardware and software processes interact and at what level?
- The type of hardware controller e.g. is it a fixed architecture and is the use of pipelining and subroutining helpful ?
- Whether pipelining of functional units can be implemented (the examination of compilers written for vector processers could be relevant).
- The possibility of providing automatic code profiling to determine which areas of the code are the most profitable to optimize.
- Whether machine learning could be usefully exploited (in the FCCP for example).

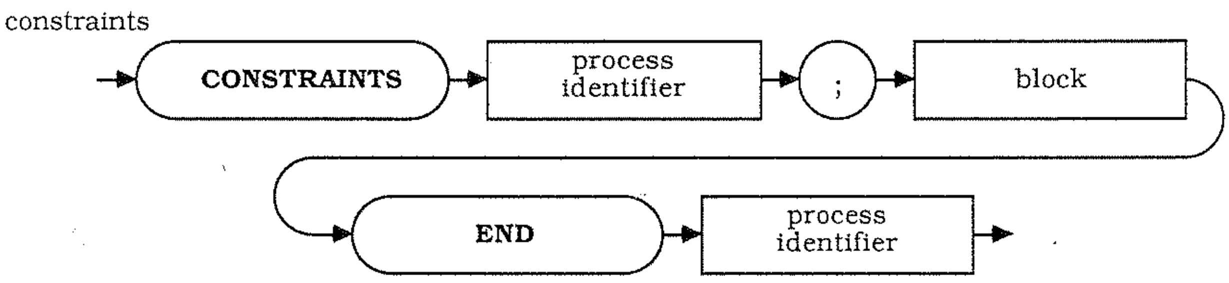
This report has outlined a system for the automatic generation of VLSI circuits from an initial SDL description. With further refinement it could form the basis of a practical implementation. Such an implementation would be not be trivial but the rewards produced by such an effort, not just in the form of a functional silicon compiler but also in the advancement of knowldege in this area, are felt to be too significant for such an attempt not to be made.

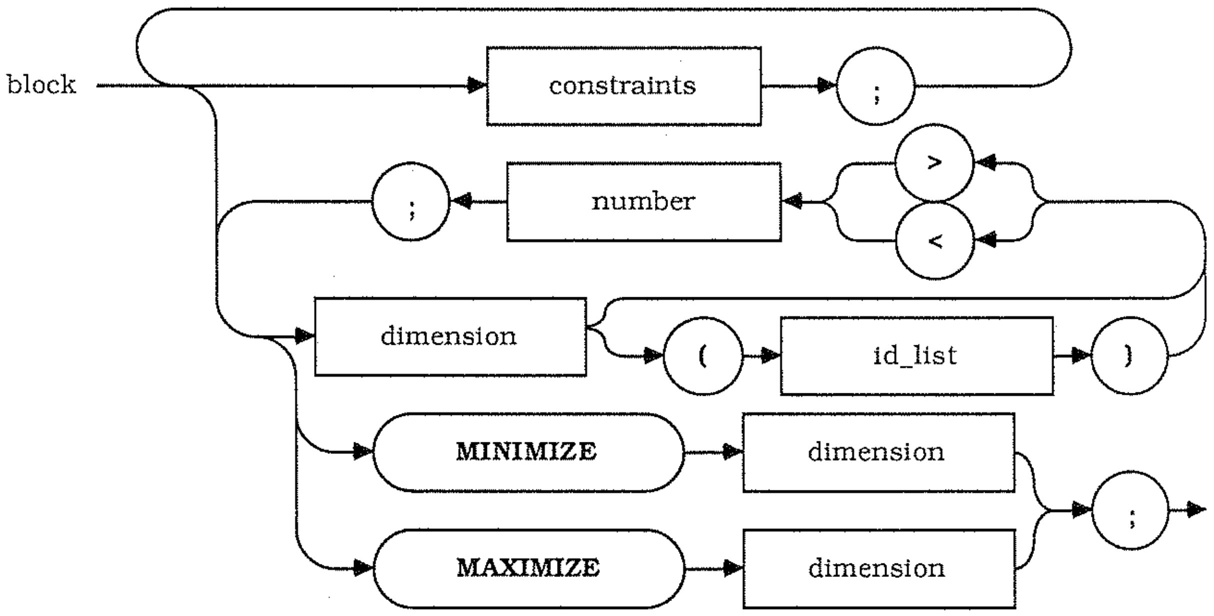
8.Ø Appendicies

APPENDIX 1.1 Railroad diagrams for the specification of a Design Constraint Language.

Syntax for Design Constraint Language

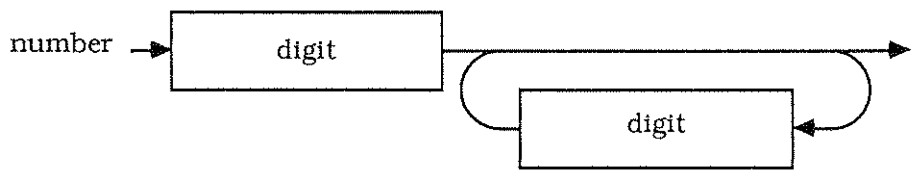


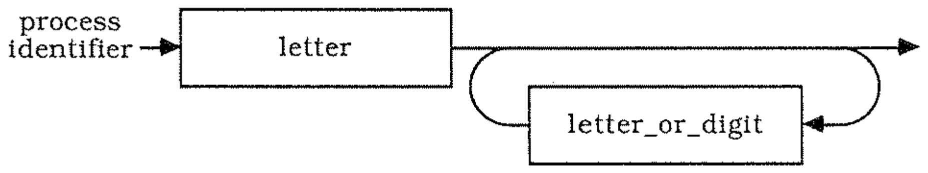


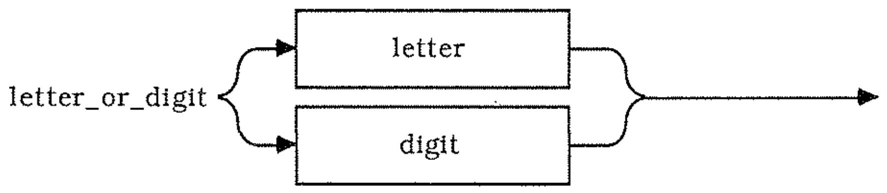


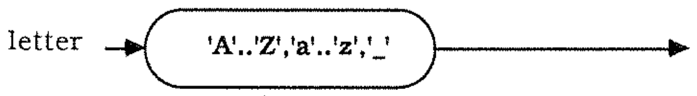
Appendix 1.1

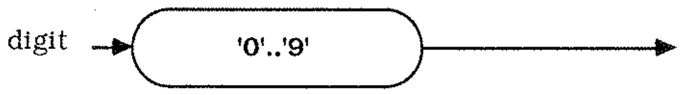
Syntax for Design Constraint Language cont.

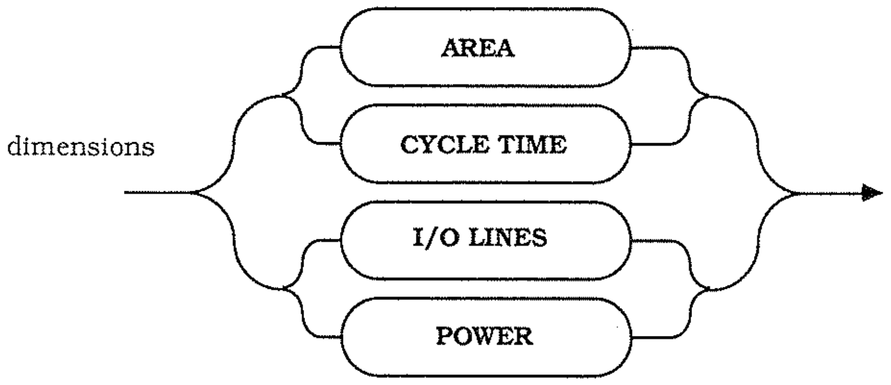












Appendix 1.1

9.Ø References

Note that the reference numbering system in this report continues from the previous two reports [41], [45] and references less than [39] can be found in [41] while references between [39] and [44] can be found in [45].

45. P. Beckett et al, "SDL - VLSI Project Report No. 2", Department of Communication and Electronic Engineering, RMIT, Research and Development Memorandum No. 112Ø4ØM, March 1986.
46. P.W. Metzger, Managing a Programming Project, 2nd Ed. Prentice-Hall Inc. 1981.
47. A.V Aho et al, Compilers - Principles, Techniques and Tools, Addison-Wesley Pub. Co., December 1985.
48. Lance A. Glasser & Daniel W. Dobberpuhl. The Design and Analysis of VLSI Circuits. Addison-Wesley Pub. CO.
49. Daniel D. Gajski, "Silicon Compilation", VLSI Systems Design, Nov 1985 pp 48-64.
5Ø. J.H. Kim, J. McDermott and D.P. Siewiorek, "Exploiting Domain Knowledge in IC Cell Layout", IEEE Design & Test, Aug 1984 pp 52-64.
51. Curtis Panasuk, "Silicon Compilers Make Sweeping Changes in the VLSI Design World", Electronic Design, Sept 2Ø 1984 pp 67-74.
52. Stephen C. Johnson, "Silicon Compiler lets System Makers Design their own VLSI Chips", Electronic Design, Oct 4 1984, pp 167-169.
53. S.C. Johnson, "One-stop VLSI Design Comes in a System that allows Tests, Refinements", Electronic Design, Oct 4 1984, pp 169-176.

9.Ø References

54. Stan Mazor, "Data Path Design Shows Worth of Silicon Compiler to VLSI System Makers", Electronic Design, Oct 4 1984, pp 176-181.

55. Max Schindler, "CAE", Electronic Design Nov 15 1984 pp 129-14Ø.

56. B. Lee, D. Ritzman and W. Snapp, "Silicon Compiler Teams with VLSI Workstation to Customize CMOSIC's", Electrnoic Design Nov 15 1984 pp 149-162.

57. Paul Wallich, "On the Horizon Fast Chips Quickly", IEEE Spectrum March 1984 pp 28-34.

58. J.R. Southard, "Silicon Compiler Demands No Hardware Expertize to Fashion Custom Chips", Electronic Design Nov 15 1984 pp 187-2ØØ.

59. Nicolas Maokhoff, "AI Techiniques Aim to Ease VLSI Design", Computer Design, March 1985, pp 33-38.

6Ø. M.A. Jabri & D.J. Skellern, "A Hybrid Rule-based/Algorithmic Approach to VlSI Circuit Floorplanning", The Fifth Aust. and Pacific Region MicroElectronics Conference., Adelaide Aust, May 1986. pp 225-231.

61. Alex Dickinson, "An Application of Domain Knowledge to VLSI CAD", The Fifth Aust. and Pacific Region MicroElectronics Conference., Adelaide Aust, May 1986. pp 225-231.

62. N. Wirth, Programming in Modula-2, Springer-Verlag, New York, 1982.

63. Thaddeus J. Kowalski, An Artificial Intelligence Approach to VLSI Design, Kluwe Academic Pub., Boston 1985.

9.Ø References

64. Justin R. Rattner, "Functional Extensibility: Making the World Safe for VLSI", VLSI Systems & Computations 1981 pp. 5Ø-51

65. G. W. Cox et al, "Interprocess Communications & Processor Dispatching on the Intel 432", ACM Trans. Comp. Sys., Vol. 1, No. 1, Feb. 1983, pp. 45-66