A C-Based High Level Synthesis System

by

Hassan Fakhri Al-Sukhni

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER ENGINEERING

January, 1994

Order Number 1360384

A C-based high level synthesis system

Al-Sukhni, Hassan Fakhri, M.S.

King Fahd University of Petroleum and Minerals (Saudi Arabia), 1994

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

A C-Based High Level Synthesis System

BY

Hassan Fakhri Al-Sukhni

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

Computer Engineering

January 1994

King Fahd University of Petroleum & Minerals
College of Graduate Studies

This thesis, written by **HASSAN FAKHRI AL-SUKHNI** under the direction of his Thesis Advisor and approved by his Thesis Committee, has been presented to and accepted by the Dean of the College of Graduate Studies, in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE** in **COMPUTER ENGINEERING.**

Thesis Committee

Thesis Chairman

Co-Chairman — 11th Jan '94

Member — 11 Jan. 1994

11.1.94
Department Chairman

Dean, College of Graduate Studies

11.1.94
Date

A C-Based High-Level Synthesis System

Hassan Fakhri Al-Sukhni

Computer Engineering

January 1994

Dedicated to

My Parents

Acknowledgment

Praise be to Allah for guiding and helping me in every aspect of this life. Peace and mercy be upon His last Prophet.

I wish to thank my thesis advisor Prof. Mohammad S.T. Benten for his continuous guidance, help, and encouragement, co-chairman Prof. Sadiq M. Sait who introduced me to the area of high-level synthesis, and Prof. Habib Youssef for their active support, help, and valuable suggestions. I also wish to thank faculty, research assistants, graduate assistants and the staff members of the Computer Engineering Department for their support, especially the chairman Prof. S. Abdul Jauwad.

I wish to thank Eng. Habib Mansour for his support and encouragement at the early stage of my study and Mr. Ibrahim Abushanab for his continuous help and encouragement.

Contents

List of Figures

Abstract

Name: HASSAN FAKHRI AL-SUKHNI
Title: A C-Based High-Level Synthesis System
Major Field: Computer Engineering
Date of Degree: January 1994

High-Level Synthesis (HLS) refers to the process of translating a high-level specification of the behavior of a circuit into a structural design. The outcome is a net-list of Register Transfer Level (RTL) components, such as ALUs, registers and multiplexers.

Because of its complexity, HLS is broken into several steps, where a subset of the overall problem is solved in each step. The steps move the source specification into a target specification, through several intermediate forms.

Existing HLS systems fall into two broad categories. The first category takes a hardware description language (HDL) as its source specification. Many structural biases are present in all existing HDLs. These biases lead to a restricted view of the design space. For the second category, a subset of some known high-level programming language is used to describe the behavior of the intended design. The identification of the subset to use is a problem by itself. Another problem is that the user must think within the boundary of the subset.

In this work we present a new approach to HLS, where the system takes its source specification in any high-level programming language, without any restrictions. This is facilitated by the novel idea of using the programming language compiler to produce the first intermediate form in the transformation process. This first intermediate form is the assembly language (AL). The use of the high-level language compiler serves two objectives. The first is the utilization of the optimization carried out by the compiler. The second is to avoid restricting the language to certain data types or control constructs.

Due to its machine dependency and complexity, AL is transformed into another form, which is machine independent and has simpler syntax and semantics. We refer to this form as **Pseudo Assembly Language** *(PAL). PAL descriptions are used by the system components to produce the intended hardware in an RTL description language. AHPL is used as an RTL description language in the system.*

The major contributions of this work are the introduction of this new methodology of tackling HLS problem and the introduction of a new heuristic scheduling algorithm based on the path-based scheduling originally presented by R. Camposano in [21].

Chapter 1

Introduction

High-Level Synthesis (HLS) refers to the process of translating a high-level specification of the behavior of a circuit into a structural design, in terms of an interconnected set of RTL components, such as ALUs, registers and multiplexers.

The main tasks performed by a general-purpose HLS system are:

1. Compilation of the source specification language into an internal representation, referred to as the Intermediate Form (IF). Usually the intermediate form consists of a data flow graph and/or a control flow graph. This step is a straightforward mapping of the behavioral description. It is very similar to the compilation task of programming languages.

2. Optimization of the internal representation. This step involves both compiler-like and hardware-specific optimization. Examples of compiler-like optimization include:

   - dead code elimination.
   - constant propagation.
   - common sub-expression elimination.
   - variable disambiguation by global flow analysis.
   - code motion.
   - in-line expansion of sub-programs.
   - loop unrolling.

   Hardware-specific optimization include:

   - life-time analysis to perform variable folding.
   - substituting multiplication and division by shifts.
   - increasing operator-level parallelism.
   - creating concurrent processes.

3. Scheduling and Allocation. Scheduling assigns each operation to a control-step. A control step is a measure of time. It is equivalent to a state in a finite state machine, or to a micro-program step in a micro-programmed

controller. Scheduling is sometimes called control synthesis or control step scheduling. Allocation or Data Path Synthesis, assigns each operation to a piece of hardware. Allocation involves both the selection of the type and quantity of hardware modules from a library (also called module assignment) and the mapping of each operation to the selected hardware (module binding). Scheduling and allocation are considered to be the heart of the HLS process.

4. Output generation produces the design in the desired format as required by a logic synthesis tool and/or a finite state machine synthesis tool. The resulting design is usually output in an RTL language, or as an interconnected set of RTL components.

HLS has drawn a lot of attention in the last decade. A large amount of work has already been reported [1]..[20]. The interest in HLS is due to several reasons, among these are:

1. Unprecedented circuit complexity that emerged as a result of developments in LSI and VLSI technologies. The design, modeling and implementation of such circuits have placed demands on highly structured and rigorous design methods as a means for mastering and managing this complexity.

2. The same developments have enhanced the feasibility of designing, building and studying computer hardware systems of enormous organizational complex-

ity. Again the need for structuring the design process and design descriptions seems imperative if designers are to produce well-behaved, understandable, and intellectually manageable systems.

3. Theoretical pressure was forcing itself on the designers, in the sense that designs' correctness should be verified and circuits should have fewer design errors.

4. ASICs are produced in smaller quantities and their life times are shorter. Therefore the design cost is becoming too expensive to be carried by expert humans. The ideal solution is to automate the process, making the design less expensive but obviously at the expense of design quality (performance).

5. Making the search through the design space more feasible.

6. To give an elegant way of design documentation.

7. High level descriptions constitute an excellent communication media.

8. To make the IC technology available to a wider slice of people.

High-level synthesis is a very involved process. There are several factors which account for this complexity:

- Optimization Problem: Even sub-problems of the synthesis problem such as scheduling or allocation have been proven to be NP-complete.

- Partial Evaluation Problem: The evaluation of partially completed designs is very complex, since the outcome could not be predicted before completing the design.

- Design strategies are problem dependent: To choose the proper design strategy that will best suit the problem is a very difficult task in an automated synthesis system. Usually such decisions are made by the user as in the System Architect's Workbench [2], [8].

- Design process steps may change with the application: The HLS process is broken down into steps, which take the source specifications (SS), through possibly several intermediate forms (IFs), into a target specification (TS), Figure 3.1. The order in which the steps are carried out is usually problem dependent.

The following chapter reviews the existing literature on HLS systems. Chapter 3 describes the HLS system implemented. The system is called KFUPM HLS, *KHLS* System. The chapter explains as well the defined *PAL*. Chapter 4 is devoted to the scheduling algorithm introduced and used in the system. In Chapter 5 the Internal Data Structure, IDS, of KFUPM HLS system is presented. Chapter 6 details the allocation algorithm used. Chapter 7 describes the translation from IDS into AHPL. Finally, conclusions are drawn in Chapter 8, and future work is suggested.

Chapter 2

Literature Review

The roots of HLS can be traced back to the late 60s [1]. In 1969 the ALERT system [3], might be one of the earliest HLS systems. By mid seventies the field gained more and more popularity, and several systems were reported [10]–[19]. Among them were the Carnegie-Mellon University's (CMU) Expl system [1], the CMU-DA system [9], [10], the University of Kiel's MIMOLA system [17], and the University of Karlsruhe's CADDY/DDL system [20]. In the last decade, work on HLS has proliferated and several systems were implemented. Neither the space, nor the scope of this work allows the description of all these systems. The interested reader is referred to [2] for an excellent survey. In this chapter a brief description of some of the well known systems that have been reported in the literature is presented.

One of the first HLS systems is the IBM's ALERT system [3], [4]. It takes as source specifications, descriptions in a language based on Iverson notation. The language supports arithmetic and logic operations, assignments, branching, arrays, etc. The behavioral description describes storage, registers, and any pre-defined piece of logic, such as adders or decoders. The intermediate form is a data-flow/control-flow design file, representing the initial structure of the design. This is the output of the compilation step. In the optimization step, macros are expanded, array referencing is replaced with logic to select the appropriate element, and common sub expression elimination is performed. The same intermediate form is used to perform scheduling and data path synthesis. In scheduling, the user is allowed to pre-specify all or part of the schedule. Initially, all operations are assigned to a single control step. This step is then split at statements that are destinations of **Goto** instructions, after conditional branching statements, and whenever a variable receives a second value in a given control step. In data path synthesis, a data flow analysis is used to assign values to flip-flops (or latches), storing those values that are produced in one control step and used in a later control step. The target specification of the system is an RTL design, specified in terms of Boolean equations. A complete IBM 1800 computer was synthesized automatically, requiring, however, more than twice as many components as used in an equivalent design that was produced manually (by a human designer) [4].

The System Architect's Workbench of Carnegie-Mellon University — [1], [2], [5]–[10] — is the result of an extensive work that started in the mid 70's [5]. Literature on the system is still being published [6]–[8]. Several synthesis tools were incorporated in the system for various target architectures. It supports three synthesis paths: a general synthesis path, a pipelined-instruction-set-processor-specific synthesis path, and a microprocessor-specific synthesis path. The system takes an extended IFPS description as its source specification. The IFPS extension supports processes and message-passing inter-process communication, and user definable operations. It can take its source specification in VHDL as well. It uses the Value Trace (VT) [34], a data-flow/control-flow graph, as its intermediate form. The system supports behavioral and structural transformations, some to improve the efficiency of the control structure, others to allow the user to explore algorithmic level design alternatives. Examples of structural transformations include: in-line expansion and formation of procedures, code motion, combination of nested decoding operations, etc. Examples of the algorithm-level transformations include adding concurrent processes to a design, pipelining a design, and structurally partitioning a design. Depending on the chosen synthesis path, the proper scheduling routine is invoked. The CSTEP scheduler used in the general synthesis path uses the list scheduling technique on a block-by-block basis, with timing constraint evaluation as the priority function. Operations are scheduled into control steps one basic block at a time, with blocks scheduled in execution order using a depth-first traversal of the control

flow graph. For each basic block, data ready operator are considered for placement into the current control step, using a priority function that reflects whether or not that placement will violate timing constraints. Resource limits may be applied to limit the number of operators of a particular type in any one control step, i.e., area restrictions. A data path synthesis routine may be invoked, again depending on the chosen synthesis path. The EMUCS data path allocator used in the general synthesis path attempts to bind data flow elements onto hardware elements in a step-by-step manner. In each step, all unbound data flow elements are considered. To decide which element to bind, EMUCS maintains a cost table, listing the cost of binding each data flow element onto each hardware element. For each unbound data flow element, EMUCS calculates the difference of the two lowest binding costs, then binds the data flow element with the highest difference to the hardware element with the lowest cost. This attempts to minimize the additional cost that would be incurred if that element were bound in a later step. The system outputs either a CMOS standard cell or TTL implementation [1]. Examples of hardware synthesized by the system include the Intel 8251, and the IBM System/370 [2].

Carleton's HAL (Hardware ALlocator) system [11]-[15] takes the behavioral source specification as a manually entered data-flow/control-flow graph. This graph is used for internal processing. The system combines the Force-Directed Scheduling (FDS) algorithm presented in [11] with list scheduling to produce a Force-Directed

List Scheduling (FDLS) algorithm. Whereas FDS constrains the length of the schedule, list scheduling constrains the number of functional units, or hardware. In FDLS, force is used as the priority function, where among the ready operations, the operation with the lowest force is deferred (not considered for the current control step). This process is continued until the constraint on the number of functional units is met. Data path synthesis is then performed, where functional units are allocated to perform the scheduled operations. Following that, a rule-based expert system, which takes into account the available cells, their area cost, and the timing constraints, is invoked to bind the operations to the allocated functional units. The target specification of the system is an interconnected set of RTL components. The system was used to synthesize a fifth-order digital elliptic wave filter and a pipelined 16-point digital FIR filter.

The above systems require that the behavioral specifications be described in a special functional specification language. Attempts to build Top-Down high-level synthesis systems that generate hardware from software algorithms written in ordinary programming languages have been reported in [2], [16]–[19]. There are several reasons in support of using ordinary programming languages for behavioral descriptions:

1. Special purpose structural specification languages are inconvenient. Their grammars require that the target architecture be almost completely defined

before using the language [19]. This requirement defeats the purpose of HLS, namely to describe a system behavior without any reference to structural or architectural details.

2. Wider slice of hardware designers would be able to use HLS systems. This is a direct result of the popularity of ordinary programming languages.

3. Transportability of the description to any HLS system that uses the same language as its source specification.

The YASC silicon compiler automatically synthesizes general cells from behavioral descriptions [16]. It transforms the behavioral descriptions into Boolean level descriptions, which are synthesized into custom cells. The MIMOLA system [17], takes input descriptions in the MIMOLA language - a Pascal-like language that includes recursive procedure calls and multi-dimensional arrays, and generates hardware automatically. The CADDY system [2] generates structural descriptions from the DSL language, which is also a Pascal-like language.

Stanford's Flamel System [18] takes as source specification a small subset of Pascal. This subset is limited to single, parameterless, non-recursive procedure, supporting only single-dimensional arrays, and not allowing multiplication, division and modulo operations except by constants that are powers of two. The source specification is compiled into a block graph that shows the transfers between basic

blocks of the algorithm, and to a Directed Acyclic Graph (DAG) that represents the data flow within each basic block. A graph combining both the data flow and control flow, called *dacon*, is used as an intermediate form. Block-level transformations are automatically applied to produce the dacon with the fastest implementation. The block-level transformations identified are :

(a) Line merge, that merges adjacent blocks.

(b) Alt merge, that combines a block followed by two or more conditionally executed blocks into a single block.

(c) Loop unrolling,

(d) Tat-to-tab, that moves test at top of a loop to test at bottom.

Then algebraic transformations are used to reduce the height of the dacon. Scheduling uses as soon as possible (ASAP) strategy with unlimited hardware. Then it attempts to fold together pairs of resources (functional units, registers, buses, etc.) that perform the same function or which can be generalized to perform the same function. If the cost exceeds the constraint, the schedule is lengthened. The target specification is a bit-slice architecture. Examples of hardware synthesized by Flamel include a bubble sorter, a string convertor, a hash table manager, and others that could be found in [18].

NTT's HARP (Hardware Architecture Ruling Processor) is another example of a top-down HLS system [19]. It takes as source specification a subset of ANSI FORTRAN77, which supports constant, integer, real, logical variables and arrays, subroutines and function calls, but does not support loops with indefinite iteration. It uses a data flow graph as its intermediate form. The scheduling strategy used is ASAP, together with a parallelism evaluator (PE), which does the binding of operations as well. The operations in the PE are bound to two types of functional units: the first type consists of functional units that perform a pre-specified set of operations, and the second type is defined automatically by the system, but limited by a database that specifies permissible combinations of operations in a functional unit. Variables are bound to registers by performing life-time analysis first. Then an algorithm similar to the left-edge algorithm used in channel routing is invoked. This algorithm attempts to minimize the number of registers. Buses and multiplexers are inserted wherever needed. The target specification is an RTL description that is fed to an RTL synthesizer. Examples of hardware synthesized by the system include a 4-points FFT, and a biquad filter.

The aforementioned systems are just examples of the large amount of work that has been done in HLS. These systems reflect the major contributions to the field. Needless to say that the reported work on sub-problems of the field is even larger, and including it here is out of the scope of this work.

Chapter 3

System Overview

3.1 General approach

To overcome the complexity of the HLS problem, a stepwise approach is adopted. The first step takes the source specification and transforms it into a first intermediate form, say IF_1, which is closer to the target specification. The k^{th} step transforms IF_{k-1} to IF_k. It is always the case that IF_k is closer to hardware (less abstract) than $IF_1, IF_2, .., IF_{k-1}$, see Figure 3.1.

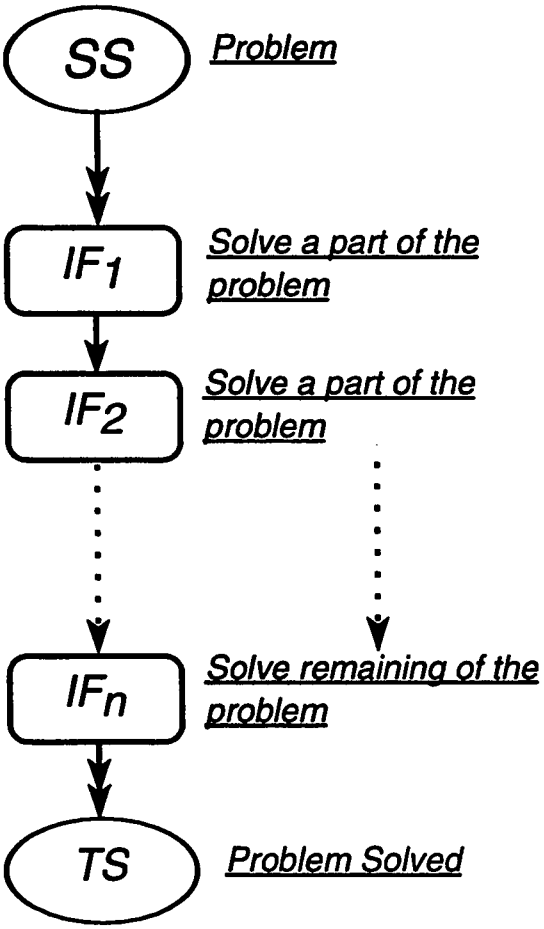

Figure 3.1: HLS formulated a series of transformations.

All existing solutions fall into two major classes. The first class performs HLS from Hardware Description Languages, *HDLs*. The second class performs HLS from a subset of a specific high level programming language. In this work a different approach is adopted. Synthesis is performed from any high level programming language, without restrictions on the language data types or control constructs that might be used, except for subroutines which are restricted at the time being to a certain way of usage. Figure 3.2 gives an overview of the approach. The salient features of this approach are the use of Assembly Language, *AL*, and the introduction of what we call Pseudo Assembly Language, *PAL*. The advantages of *AL* and *PAL* are discussed next.

The first step in our approach is the compilation of the source high-level specification into AL. This is performed by the language compiler. An assembly language translator is invoked next to produce PAL, which is the actual input specification to the HLS system. In this work the C-language is used as the input specification, however other high-level languages could easily be used as illustrated in Figure 3.2. The described HLS system could be used in any machine provided that the proper AL translator is built, Figure 3.3. PAL will be used to build the Internal Data Structure (IDS), of the system, which is similar to a Control/Data Flow Graph, *CDFG*, using the *IDS Extractor*, Figure 3.4. IDS is described in detail in a following section.

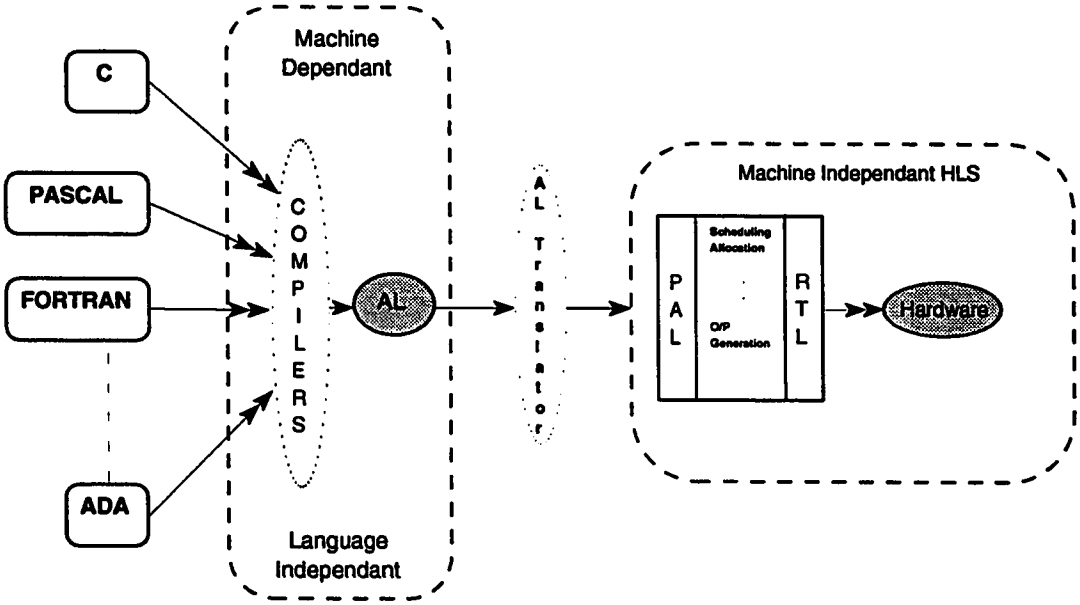

Figure 3.2: Proposed approach overview.

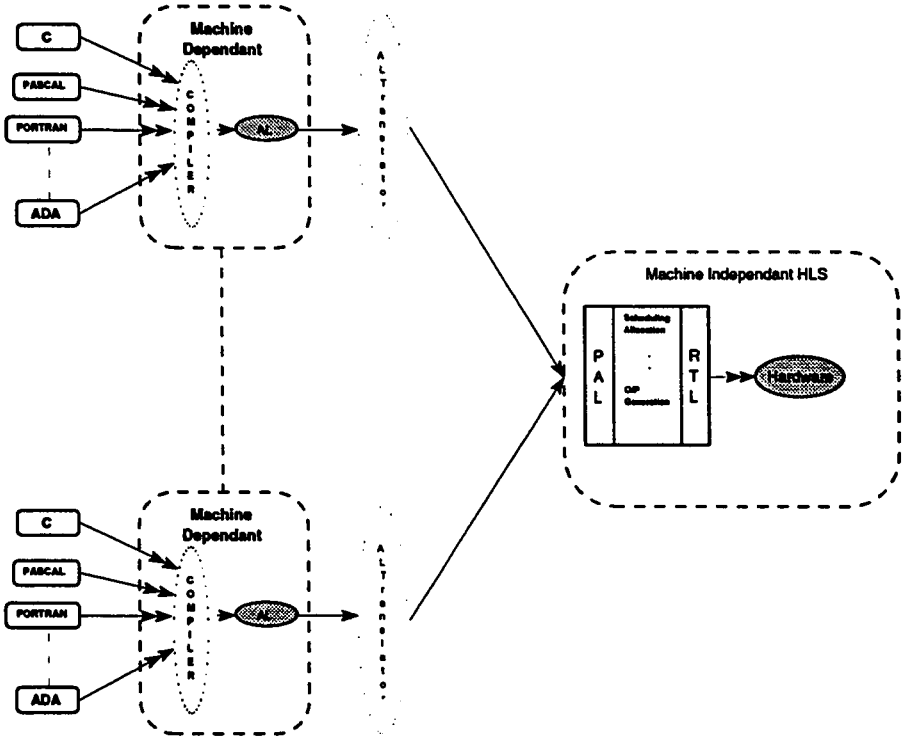

Figure 3.3: Machine independence of proposed approach.

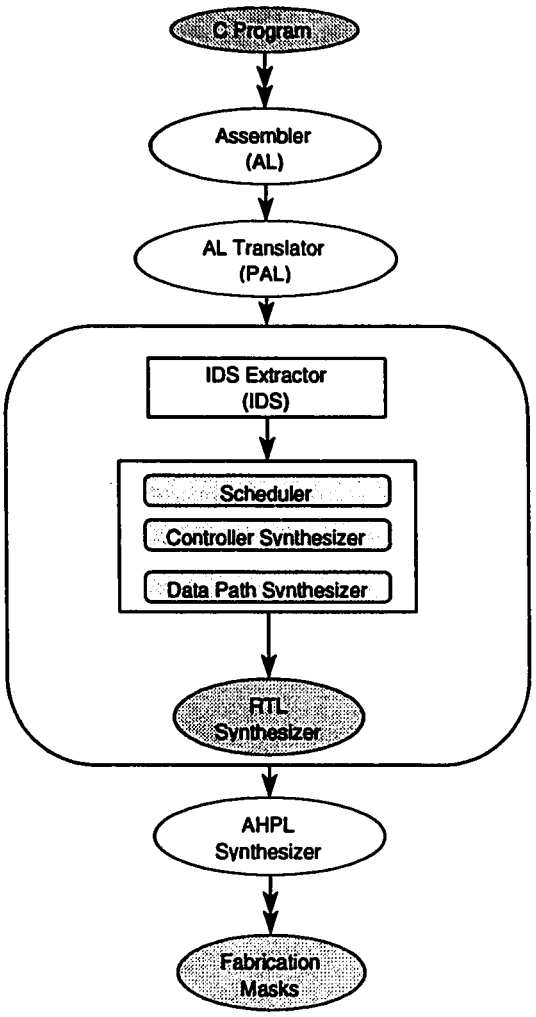

Figure 3.4: KFUPM HLS system components.

IDS is used by the system scheduler to synthesize a Finite State Machine (FSM) controller, satisfying, if any, user-specified constraints on the number of functional units and clock cycle, while minimizing the number of states of the FSM. The scheduler is the heart of the system. A heuristic algorithm is developed to carry out the scheduling task. The algorithm, called here *Loop-Based Scheduling* (LBS) is based upon the As-Fast-As-Possible (AFAP) path-based scheduling algorithm presented by R. Camposano [21]. A separate section in Chapter 4 is devoted for the LBS algorithm. Allocation is done alongside with scheduling, an As-Soon-As-Possible (ASAP) algorithm is used to allocate hardware. Allocation is described in a Chapter 5. RTL descriptions in AHPL are generated from IDS using the RTL synthesizer (Figure 3.4). Generated RTL descriptions are finally fed to the AHPL silicon compiler to generate the fabrication masks.

3.2 Assumptions on Input Specifications

As mentioned earlier, no restrictions are imposed on the input language. However, certain assumptions have to be made in order for the user to communicate with the system properly. The assumptions are quite simple, and not related to the constructs or types that might be used in the behavioral descriptions of the intended circuit, but rather to the behavior of the circuit. Behavioral description means the description

of the way the intended circuit behaves with the outside world. Formally stated, a behavioral description of a circuit is the description of how the circuit maps its inputs to its outputs. Thus, the following is assumed about the input specification.

1. Input/Output ports of the described circuit are assumed to be defined as "*external*" variables in the input specifications. The second assumption has already been introduced in the previous sub-section and is repeated here for completeness.

2. Subroutine calls are allowed. However they are assumed to have no parameters, and do not use local variables. Globally defined variables could be used freely.

   The rest of the assumptions are not related to the input specifications but rather to the system implementation.

3. Hardware modules are taken from a library where for each module the following information shall be supplied.

   - Time-delay for the hardware unit.
   - Maximum allowable number of units to be used in the implementation.
   - The name used to access the module from the AHPL Combinational Logic Units library. This is needed for the generation of AHPL output

description.

4. As for the generated hardware the following is assumed.

   - Single phase clocking is assumed.
   - Signals and their complements are assumed to be available.

3.3 Why AL ?

There are several motivations that prompted us to use assembly language as a first intermediate form.

1. To utilize the optimized assembly language output generated by the C-compiler. The C-compiler already performs several optimization tasks which include, but not limited to:

   (a) Dead Code Elimination.

   (b) Constant Propagation.

   (c) Common Sub-expression Elimination.

   (d) Code Motion.

   (e) In-line Expansion of Sub-programs.

   (f) Variable Disambiguation.

2. Other programming languages could be accommodated in the system, as illustrated in Figure 3.3.

3. The principal motivation for using AL is to avoid restricting the user to a subset of the original programming language. In existing HLS systems, programming languages were restricted to avoid handling complex constructs, such as multi-dimensional arrays, user-defined data types, subroutines, etc. Compilers already handle these problems and the generated AL does not have such constructs. An exception is subroutine calls which require special handling. Subroutines in the current implementation can have neither parameters, nor local variables. Subroutine calls are, for the time being, expanded in-line.

3.4 The Need for Pseudo Assembly Language

AL has several limitations that could be summarized as follows:

1. AL is bound to a specific machine, which makes the transportability of the HLS system limited to that class of machines which use the same AL instruction set.

2. AL has too many instructions and types of instructions.

3. AL usually introduces new dependencies, either in the control or the data flow. This is due to the fact that machines usually have a limited number of registers, and some machines do not have a memory-to-memory transfer instructions. An example for such introduced dependencies is illustrated in Figure 3.5. The AL code introduces "tempreg" as a third variable, which is not needed to carry the transfer operation in a hardware implementation.

C-Code	AL-Code
var1 = var2 ;	load tempreg,var2 stor tempreg,var1

Figure 3.5: Introduced dependencies in AL.

The solution to the aforementioned AL limitations is to translate the AL into an other form, called here Pseudo Assembly Language *PAL*. PAL is presented in the following section.

3.5 Pseudo Assembly Language (PAL)

In order to make the system portable, as well as to eliminate the limitations of AL discussed in the previous section, a new language, called here Pseudo Assembly

Language (PAL), is introduced. PAL was carefully designed to have the following characteristics:

1. Simplicity: PAL has simple syntactic and semantic constructs.
2. Completeness: It is limited to a set of basic types of instructions. This set includes:
   - Data Moves
   - Operations on data
   - Control constructs
3. Temporary-Register-Free: To make it machine independent, Figure 3.3.
4. Capable of sub-program handling. This characteristic requires further study. Currently, this capability is not included in the language.
5. Easily mapped from AL.

PAL is informally defined in "yacc" format, in Figure 3.6. PAL is intended to hide all insignificant details for a hardware implementation. By this, PAL moves the input specification a step toward the required target specification. Although PAL is very much a behavioral description, it does have hardware images in the way it is constructed, where hardware related information are kept from the assembly

```
line:
    / empty /
    |
    line statement '\n '
statement:
    /* empty */
    |
    instruction
    |
    label
    ;
label:
    address ':'
    ;
instruction:
    /* empty */
    |
    address '=' address REST
    |
    If condition Goto address
    |
    Goto address
    ;
REST1:
    address REST
    ;
REST:
    /*empty */
    |
    OPR address
    ;
OPR:
    /* To be defined according to the input assembly language */
    ;
address:
    /* any combination of characters */
    ;
condition:
    "flags"
    |
    "!flags"
```

Figure 3.6: PAL Definition in YACC format.

language. An example would be the translation of several assembly instructions into the same PAL instruction, since they all have the same hardware effect, say transferring a variable onto the other.

Simplicity of PAL, and the ease of mapping from AL are quite obvious from the way it is defined. The completeness of PAL is inherent to the way "address" and "OPR" are defined, where any assembly instruction could be easily included in PAL. Temporary registers are kept in PAL, they are eliminated in a following optimization phase. Subroutine handling is not incorporated in PAL for the time-being, since subroutines are expanded in-line in a pre-processing step applied to the input C-specification. Figure 3.7 shows a description of a Prefetch unit taken from [21]. The description in [21] is in VHDL, the figure is a direct translation to C. The AL output of the C-compiler for the Convex machine is shown in Figure 3.8. PAL output of the AL Translator is shown in Figure 4.1. The numbers that appear at the end of each line are added for clarity of explanation and not produced by the AL translator.

The AL Translator carries out some important tasks other than the translation into PAL. These tasks are:

1. Loop Identification: Loops are the key observation in the scheduling algorithm, LBS. Loop entrance points are marked during the translation process to be

used later by the scheduler.

2. Semantic Processing, where for each loop, the variables that are set to a new value within the loop body are identified, and passed to the scheduler.

```
extern int branchpc,ibus,ire;
extern int branch,ppc,popc,obus;
prefetch()
{
int pc,oldpc;
    while (1){
        ppc=pc;
        popc=oldpc;
        obus=ibus+4;
        if(branch)
            pc=branchpc;
        while(ire!=1);
        oldpc=pc;
        pc=pc+4;
    }
}
```

Figure 3.7: Prefetch Example: C-Code.

```
;NO_APP
gcc2_compiled.:
.text
.text
.align 2
.globl _prefetch
_prefetch:
        ld.w _ibus,s0
        add.w #4,s0
        ld.w _branch,s4
        ld.w _ire,s3
L1:
        st.w s1,_ppc
        st.w s2,_popc
        st.w s0,_obus
        eq.w #0,s4
        jbrs.t L4
        ld.w _branchpc,s1
L4:
        eq.w #1,s3
        jbrs.f L4
        mov.w s1,s2
        add.w #4,s1
        jbr _prefetch
```

Figure 3.8: Prefetch Example: Convex AL Code.

Chapter 4

Scheduling

4.1 Overview of Scheduling

The IDS extractor, Figure 3.4, transforms PAL into the IDS form, which is used by the various system components for decision making. Discussion on the IDS is delayed until Chapter 6. The IDS was designed to meet the requirements of the scheduling and allocation algorithms. Hence, presenting the IDS would be more convenient after presenting these algorithms.

Scheduling is defined in the context of HLS of synchronous digital systems as, the task of assigning operations to control steps so as to minimize an objective func-

tion while meeting certain user specified constraints [1]. Operations are the atomic components used to describe behavior, like arithmetic or boolean operations. A control step corresponds to one state in a finite state machine, or to a microprogram step. Control steps will be called *controlstates* or just *states* throughout the rest of this work. This shall reflect the fact that the controller synthesized to control the resulting hardware implementation is a Finite State Machine (FSM). Scheduling is an NP-hard problem [11]. Several heuristic algorithms have been developed to find a good solution rather than an optimal one [11], [12], [24], [25], [23].

Scheduling algorithms have been classified in [1] into two major classes. This classification is according to the way the algorithm develops its solution. The first class is identified as transformational, where a default solution is picked, then a series of transformations are applied to that solution to move it toward an acceptable solution, where the constraints are met and the objective function is satisfied. Examples of this class are the scheduling algorithms used in CAMAD [24] and in the YSC [25]. The second class is identified as iterative/constructive, where the operations are scheduled one at a time meeting the constraints, and improving the objective function all along. Examples of this class are the ASAP scheduling algorithm [1] and the different list scheduling algorithms, like the freedom-based list scheduling used in MAHA [26], and the force-directed list scheduling used in HAL [12].

Another possible classification of scheduling algorithms can be made on the basis of the resolution exploited in developing the solution. These are Operation Based Scheduling (OBS) and Path Based Scheduling (PBS). OBS visualizes the CFG as a set of operations that need be distributed among hardware resources. A schedule can be made faster by attempting to schedule as many operations as possible in the same control step. On the other hand operations can be assigned to control steps so as to maximize hardware resource sharing. The schedule resulting from such approach will be slower but more economical in terms of hardware usage. Most of the known scheduling algorithms fall into this class. Examples are the force-directed list scheduling in HAL [12] and the scheduling algorithm used in the YSC [25]. The other class (PBS) visualizes the CFG as a set of execution paths. This class considers mutual exclusion of different paths to produce its solution. Operations in mutually exclusive paths may share the same hardware resources in the same control step. The PBS algorithms attempt to minimize the schedule length, restricted by a set of constraints. The constraints are usually supplied by the user in order to limit the number of hardware units (i.e., area) and/or the timing (i.e., maximum clock period). Algorithms that fall in this class include the As-Fast-As-Possible (AFAP) [21], and the Dynamic Loop Scheduling (DLS) [23]. This classification is similar to that presented in [22].

Our scheduling algorithm, Loop Based Scheduling (LBS) is a PBS algorithm that

heuristically solves the scheduling problem. The technique is described in detail in a later section. In the following section we review the definitions required to formulate the problem as presented in [21]. In section 4.3 we present the formulation of [21] along with the AFAP scheduling algorithm. In section 4.4 DLS is presented. The rest of the chapter is devoted to the presentation of our approach (LBS), along with a comparison with the other two algorithms (AFAP and DLS).

4.2 Problem Formulation

In this section we recall the necessary terminology that will be used in the formulation of the scheduling problem. These definitions are taken from [21] and recalled here for the sake of completeness.

The input to the scheduling problem is a behavioral description in the form of a directed control-flow graph, CFG, $G = (V, E)$. The nodes $v \in V$ represent operations to be scheduled, and the edges give the precedence relation, i.e., $(v_i, v_j) \in E$ iff v_i is an immediate predecessor of v_j (v_j is called an immediate successor of v_i) . The interpretation of G is: an operation is executed if one of its predecessors is executed. If a node v has more than one successor, v is said to be a conditional branch. Only one of the successors will be executed. The decision of which successor is chosen is taken according to a condition predicate $cond(v_i, v_j)$ attached to the corresponding

edge. If $cond(v_i, v_j)$ is true, then v_j is executed after v_i. The conditions on outgoing edges from conditional branches must be all mutually exclusive. Conditions are arbitrary boolean functions that are derived from conditional constructs in the behavioral description language like IF, CASE, WHILE, etc. In PAL there is only one construct, namely "**If** condition **Goto** address ".

The control-flow graph has a unique first operation, v_1, at which execution starts. It should be possible to reach all other operations from v_1.

A longest path through the control-flow graph is a path starting at node v_1 and ending at an operation with no successor. The set of all longest paths is denoted as $\{P_l\}$. It represents all possible operation sequences that the specified behavior allows, excluding repetition of cycles.

The CFG is not directly obtained from PAL, but rather from the IDS. It is worth mentioning at this point that the C-compiler moves the constant assignments outside loops in an attempt to minimize the loop run time. This is acceptable except for external variables, which correspond to ports in our implementation. For a hardware implementation, the motion of such assignments is behaviorally incorrect because ports may change their values at any time. Hence, the value read at loop entrance time may change whilst loop execution. The solution to this problem is included in the IDS translator. In the translator, assignments of this type are put back inside

```
_prefetch:
   s0 = _ibus              (Read ibus port.)
   s0 = s0 + #4            (Add 4 to ibus.)
   s4 = _branch            (Read branch port.)
   s3 = _ire               (Read ire port. This instruction will go just after L4 label.)
L1:
   _ppc = s1               (Out s1 to ppc port.)
   _popc = s2              (Out s2 to popc port.)
   _obus = s0              (Out s0 to obus port.)
   flags = s4 == #0        (Compare branch port with 0.)
   If flags Goto L4        (If result is equal Goto L4.)
   s1 = _branchpc          (Otherwise s1 reads branchpc.)
L4:
   flags = s3 == #1        (Compare s3 (ire) with 1.)
   If !flags Goto L4       (If equal wait until reset to 0 (loop back to L4).)
   s2 = s1                 (Else store s1 for future need.)
   s1 = s1 + #4            (Increment s1 by 4.)
   Goto L1                 (Loop back to L1.)
```

Figure 4.1: Prefetch Example: PAL Code.

```
_prefetch:
   s0 = _ibus            —1    (Read ibus port.)
   s0 = s0 + #4          —2    (Add 4 to ibus.)
   s4 = _branch          —3    (Read branch port.)
L1:
   _ppc = s1             —4    (Out s1 to ppc port.)
   _popc = s2            —5    (Out s2 to popc port.)
   _obus = s0            —6    (Out s0 to obus port.)
   flags = s4 == #0      —7    (Compare branch port with 0.)
   If flags Goto L4      —8    (If result is equal Goto L4.)
   s1 = _branchpc        —9    (Otherwise s1 reads branchpc.)
L4:
   s3 = _ire             —10   (Read ire port.)
   flags = s3 == #1      —11   (Compare s3 (ire) with 1.)
   If !flags Goto L4     —12   (If equal wait until reset to 0 (loop back to L4).)
   s2 = s1               —13   (Else store s1 for future need.)
   s1 = s1 + #4          —14   (Increment s1 by 4.)
   Goto _prefetch        —15   (Loop back to _prefetch.)
```

Figure 4.2: Prefetch Example: IDS Code.

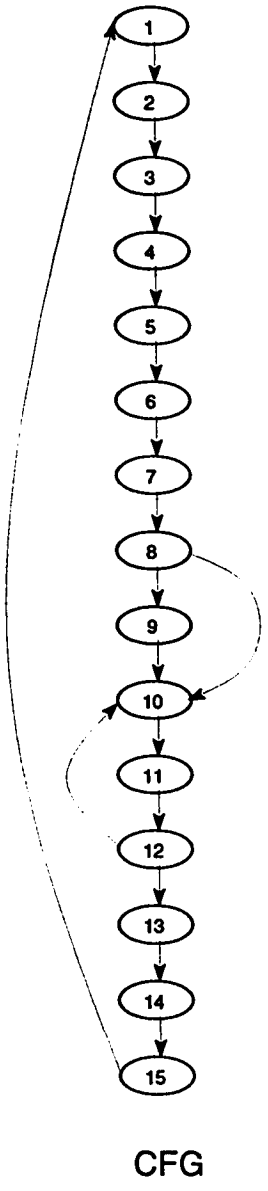

Figure 4.3: Prefetch Example: Control Flow Graph.

the loop. This is reflected in the IDS code of the prefetch example (see Figure 4.2) by changing the instruction "**Goto** L1" of the PAL code (Figure 4.1) to "**Goto** _prefetch" (instruction #15). Note also the migration of the instruction "$s3 = \_ire$" to the second loop after the label "$L4'$. Figure 4.3 shows the CFG of the prefetch example. The CFG nodes are numbered according to the statements of the IDS code given after the "—" in Figure 4.2. The CFG is a direct translation from the IDS code.

Operations that can be executed in parallel may be clustered in one node or ordered arbitrarily. If they are clustered in one node, they will always be scheduled in one control state (they will be treated as one large operation). If they are ordered, they may be scheduled in one or more control states.

In the remainder of this chapter we present several scheduling algorithms, including our algorithm, called Loop Based Scheduling.

4.3 Existing Scheduling Algorithms

In this section we present two Path Based Scheduling techniques. We start with the AFAP algorithm [21]. Then Dynamic Loop Scheduling (DLS) [23] algorithm is presented. In the following section (Section 4.4), we describe our scheduling algo-

rithm which adopts a loop based strategy and is called here Loop Based Scheduling (LBS).

4.3.1 As-Fast-As-Possible Scheduling

The AFAP scheduling problem was formulated in [21] as:

"Given $G = (V, E)$ and a set of constraints, schedule all operations $v \in V$ such that all possible longest paths $\{P_l\}$ execute in the minimum number of control states and all constraints are met."

The AFAP algorithm consists of four main steps:

1. Transform the control-flow graph into a directed acyclic graph (DAG) and keep lists of the loops.

2. All paths in the DAG are scheduled AFAP independently, according to the constraints in each path.

3. The schedules of Step 2 are overlapped while minimizing the number of control states.

4. Building the finite state machine controller.

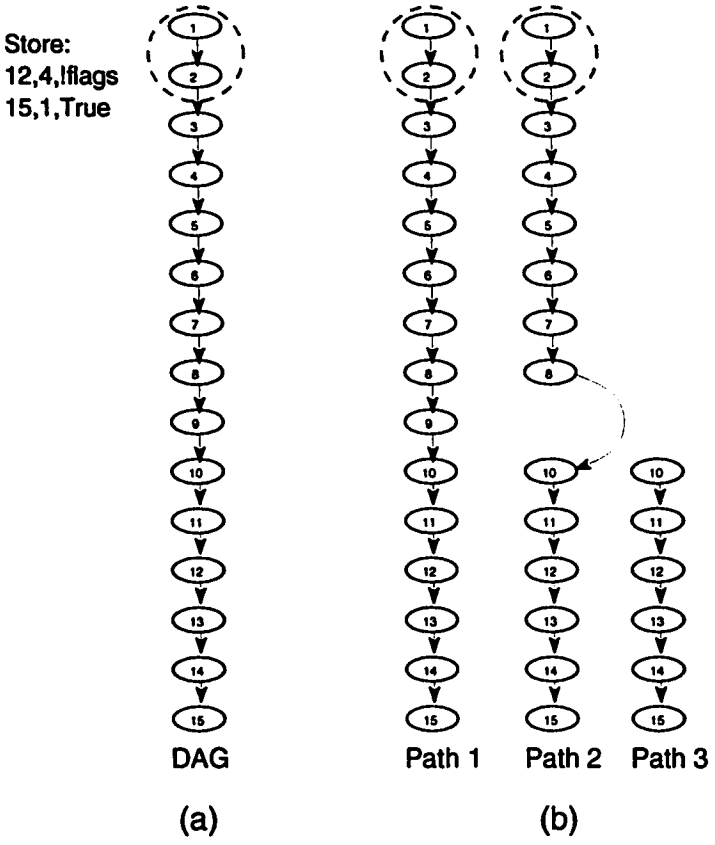

Figure 4.4: Prefetch Example: DAG and paths.

In the first step, the CFG is converted into a DAG by eliminating the loops, and keeping lists for such eliminations along with the conditions of transfer either back to the loop or out of it. This is illustrated in Figure 4.4(a), where the CFG of Figure 4.3 is shown as a DAG. Note that nodes 1 and 2 are combined together, This combination is a consequence of the optimization carried out in the optimization task within the scheduler. For the moment assume that the two operations, $s0 = \_ibus$ and $s0 = s0 + \#4$ are combined into one statement $s0 = \_ibus + \#4$. Optimization will be explained in chapter 5.

After this conversion, all longest paths ($\{P_l\}$) in the resulting DAG are identified (*path*1 and *path*2 in Figure 4.4(b)). Next, paths starting at loop entrance points are identified (*path*3 in Figure 4.4(b). Let this new set be called $\{P_e\}$. Then the set of all paths in the DAG, identified as $\{P\}$, is the union of these two sets. That is:

$$\{P\} = \{P_l\} \cup \{P_e\}$$

For the CFG of Figure 4.3, this set is shown in Figure 4.4(b). denoted the CFG of the

The next step is the calculation of all constraints for each path found in the previous step. The constraints consist of [21]:

1. Variables can only be assigned once in a control state.

2. I/O ports can be read or written only once in one control state.

3. Functional units can be used only once in a control state. This constraint is only relevant if the amount of hardware is constrained.

4. The maximal delay within one control state limits the number of operations that can be chained (i.e that feed data to each other and are executed in the same control state).

The constraints are kept as sets of operations, V, so that if any $v \in V$ is the first operation in the next state, the constraint is met. For one path, the nodes are totally ordered. Thus, each constraint (set of nodes) can be interpreted as an interval. Figure 4.5 illustrates the idea for path 1 of the prefetch example. "Constraint 1" is generated because variable $s1$ is written twice (constraint of type "1"). The constraint indicates that path 1 has to be "cut" between operations 9 and 14, so that the two assignments to $s1$ are not in the same control state. Assuming that only one adder is allowed to be used in the implementation will generate "constraint 2" between node 2 (the combined node) and node 14 (constraint of type "3"). "Constraint 2" implies that the path has to be "cut" between these two nodes, scheduling each operation in a different state, thus using a single adder in the implementation.

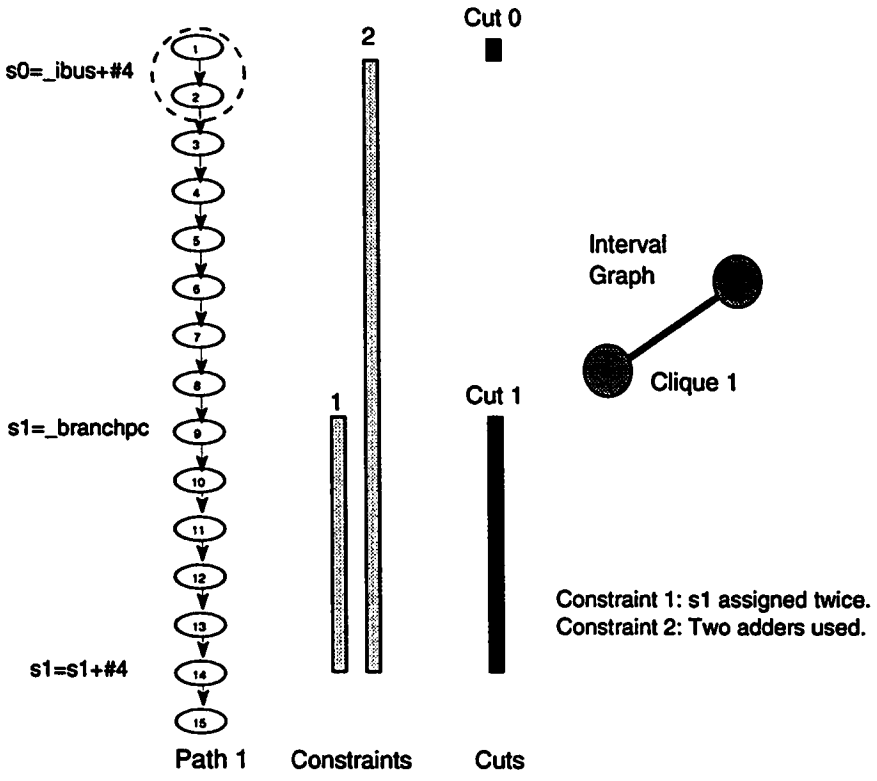

Figure 4.5: Prefetch Example: Constraints and interval graphs for path 1.

Once all constraints are calculated, an interval graph is formed (Figure 4.5). The nodes in the interval graph represent constraint intervals, and the edges join overlapping constraint intervals. A clique is a complete subgraph. A minimum clique covering is a minimal number of cliques, so that each node is in one clique. The solution to the minimum clique covering gives the minimum number of control states. A "cut" corresponds to each clique. Note that a "cut" is a set of nodes. Cutting the path at any of these nodes will satisfy the associated constraint(s). Cutting the path at different nodes of this set will produce different schedules. However, the number of states will be always the same, no matter where the cut took place. An additional cut, or state, is added for the first operation along the path, Cut0 in Figure 4.5 is of this type. In the above example, only one clique is generated, corresponding to a single cut, namely Cut1. By finishing this step the AFAP schedule is obtained for each path. The next step is the overlapping of all individual schedules to generate the final single schedule for the whole intended circuit.

To overlap schedules, another interval graph is generated in an attempt to overlap individual cuts while minimizing the total number of states. The nodes of the interval graph correspond to cuts (remember that a cut is a set of nodes, i.e., a set of operations). Edges join nodes corresponding to overlapping cuts. Again a minimum clique covering of this graph will generate the minimum set of cuts that fulfills the fastest schedule for all paths. Figure 4.6 shows the overlapping of cuts

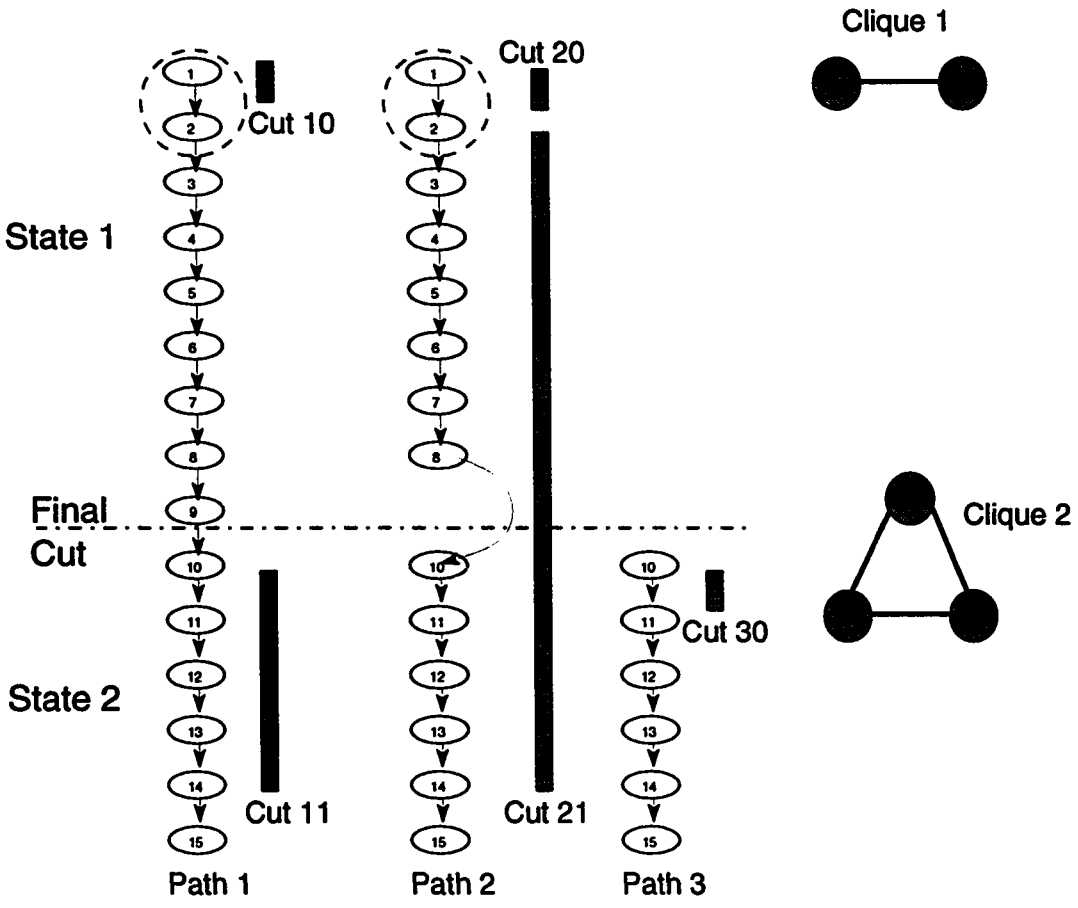

Figure 4.6: Prefetch Example: Cuts overlapping.

for the Prefetch example. Cuts are indexed with the path number first then with an increasing index starting at 0. Cut10 and Cut11 correspond to cuts discussed in Figure 4.5. Cut20 and Cut30 correspond to the initial operations in path1 and path2 respectively. Cut21 is generated by the area constraint along path2 and prevents the scheduling of nodes 2 and 14 in the same state. There are two cliques, the first corresponds to the cuts representing node 1. The second clique is formed by the cuts overlapping at operation 10. This clique indicates the starting of state 2 at operation 10.

Note that each initial operation in each path forces a cut that extends only to that node. This will force a state to be generated at each of these nodes. Recall that each loop entrance node starts at least one path, hence forcing a state in the final schedule at each of these nodes. This leads to the following Lemma.

Lemma 4.1 *The number of states generated by AFAP scheduling, denoted as* N_{AFAP} *is:*

$$N_{AFAP} \geq N_{loops}$$

Where N_{loops} *is the number of loop entrance nodes in the CFG of the input specification.*

Proof of Lemma 4.1 *The proof is obvious from the previous discussion.*

At this point all operations are scheduled. An FSM controller is then synthesized. Conditions are derived to control the transitions between states and which operations are executed within each state. Interested readers are referred to [21] for a detailed description on how conditions are derived.

4.3.2 Dynamic Loop Scheduling, DLS

O'brien et al., [23] noticed the excessive amount of processing required for the AFAP scheduling described in the previous section. The major limitation of AFAP is that, as the problem becomes complex, the number of paths generated quickly increases making the approach very cumbersome for large realistic cases. They suggested another heuristic scheduling algorithm, called Dynamic Loop Scheduling (DLS). Their problem formulation could be stated as follows:

"Given $G = (V, E)$ and a set of constraints, schedule all operations $v \in V$ such that all constraints are met, while trying to minimize the number of control states."

The formulation implies that the objective is not to produce the optimum solution in terms of the number of control states, neither does it guarantee a fastest execution time of the resulting schedule.

Dynamic Loop Scheduling works as follows. First, all longest paths are identified in the behavioral specification (paths are called *traces* in [23]). This is similar to AFAP first step. However, in AFAP each loop starts a path, while in DLS a *WAIT* statement is guaranteed to start a path, and any other loop starting nodes that do not include a *WAIT* statement. A *WAIT* statement is a VHDL construct which is actually a loop. DLS is tailored to VHDL input specifications. If the input specification were not in VHDL, the algorithm will produce the same number of paths as the AFAP.

The second step is to *cut* these paths. Cutting paths is performed ASAP, i.e., whenever a constraint is violated, the path is cut. Starting at the node after the cut, the path is scanned to check for any further violations of the given constraint.

The result is a set of *intervals*. These intervals are used to generate the FSM states. All intervals starting at the same node are combined into the same state. Note that each loop-starting-node generates a path of its own, assuming that the input specifications does not include "*WAIT*" statements. Accordingly, each of these nodes will have a respective state in the FSM. This leads us to state Lemma 4.2, which is similar to Lemma 4.1.

Lemma 4.2 *The number of states generated by DLS scheduling, denoted as* N_{DLS} *is:*

$$N_{DLS} \geq N_{loops}$$

Where N_{loops} *is the number of loop entrance nodes in the CFG of the input specification.*

Proof of Lemma 4.2 *The proof is obvious from the previous discussion.*

Conditions for transitions between states are generated from the branching conditions and the cutting points. Instructions are executed on the transitions between states, hence ensuring that the proper sequences of operations are executed.

The major advantage of DLS over AFAP, is that DLS does not perform the excessive calculations required for clique solving. However, it does not produce the same quality of solutions as in AFAP. DLS still processes all possible paths of the input specification. Figure 4.7 shows the application of DLS on the prefetch example.

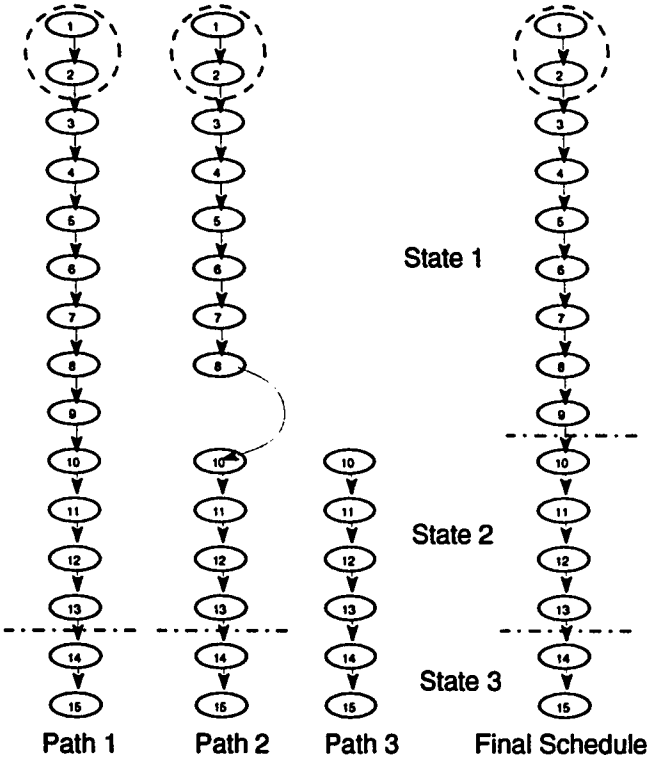

Figure 4.7: Prefetch Example: Cuts of the DLS.

4.4 Loop Based Scheduling: *a new approach*

When no constraints are present, both AFAP and DLS (assuming no WAIT statements) techniques generate schedules that are bound by the number of loops in the input specification (Lemma 4.1 and Lemma 4.2).

Above observation led us to the idea of *Loop Based Scheduling* (**LBS**). Both scheduling algorithms (AFAP and DLS) consider all paths of the CFG to produce their schedules. Loop entrance nodes are scheduled in different states of the resulting schedule. Our scheduling algorithm (LBS) uses this notion to cut the CFG into what we call *subgraphs*. "*Subgraphs*" are scheduled individually. Paths within each *subgraph* are considered to generate its corresponding schedule. This results in a considerable reduction in the number of paths to process. The technique is further explained in the remainder of this section.

The formulation of the problem tackled by *LBS* is the same as that of DLS.

"Given $G = (V, E)$ and a set of constraints, schedule all operations $v \in V$ such that all constraints are met, while trying to minimize the number of control states."

The LBS algorithm consists of the following steps:

1. Partitioning the CFG into subgraphs $\{sg_i\}$.

2. Scheduling each subgraph sg_i individually.

3. Combining individual schedules of subgraphs.

These steps are explained in detail in the remainder of this section.

4.4.1 Partitioning the CFG into Subgraphs

In *LBS*, the graph G is divided into subgraphs. A subgraph is a graph that contains exactly one loop entrance node. A loop entrance node is the first node in a loop body. Subgraphs divide the original graph such that each node of G is in exactly one subgraph. Let the first node in a loop body be identified as v_l^i where i is a running index starting at 1. "i" is incremented each time a loop is encountered in the graph. Hence v_l^1 is the first node in the first loop of the graph. Similarly, v_l^2 is the first node in the second loop of the graph. For example node 1 in the Prefetch example CFG is v_l^1, while node 10 is v_l^2. Subgraphs are constructed from G as follows. Starting at node $v1$, subgraph 1 (denoted as sg_1) is constructed by adding nodes until v_l^2 is reached. v_l^2 is the first node in sg_2. sg_2 is built the same way as sg_1 by adding nodes until v_l^3 is reached. This process is repeated until the last instruction in the input is reached. Note that v_l^1 is $v1$ assuming that the specification loops back indefinitely to $v1$ after a node with no successor is reached. An additional node is added at the end of each subgraph. This node is an unconditional branch to the first node

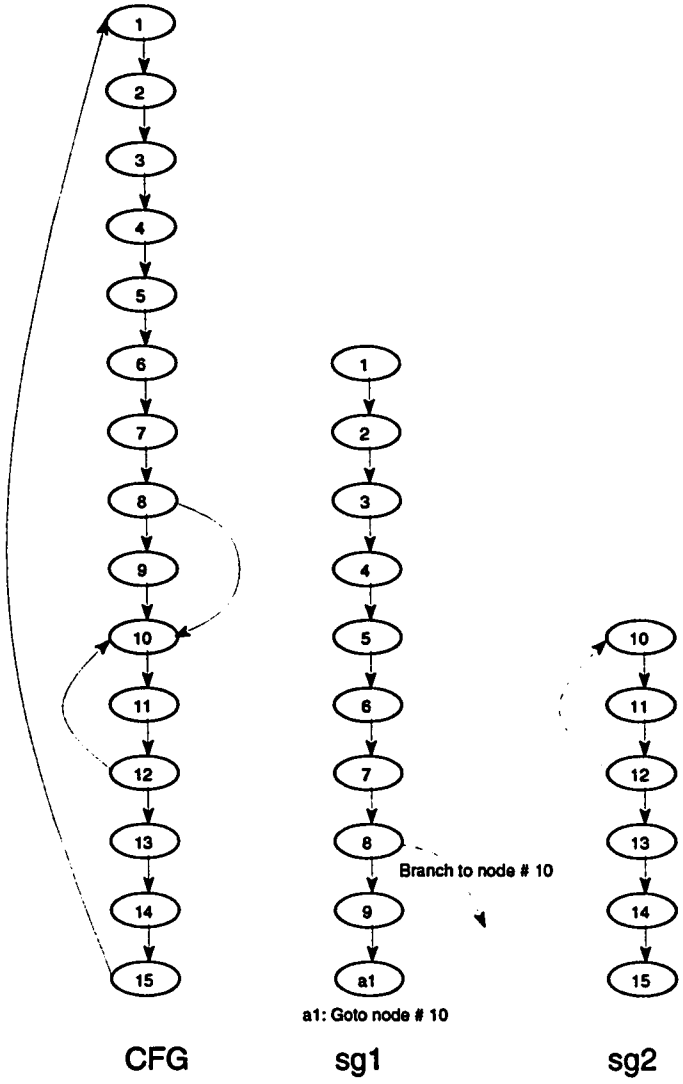

Figure 4.8: Prefetch Example: Partitioning the CFG into Subgraphs.

in the next subgraph. This is to keep the correct flow of control among subgraphs. Subgraphs will be processed independently in the following steps, thus keeping the flow of control in these branching nodes. Note the insertion of node $a1$ in subgraph sg_1 for the Prefetch example in Figure 4.8. Let the set of all subgraphs be denoted as $\{sg_i\}$. In the Prefech example, the number of subgraphs is two, subgraphs are shown in Figure 4.8. After dividing G into subgraphs, all branches of each subgraph are checked. Branches should satisfy one of the following two conditions:

(a) The branch is to a node within the subgraph.

(b) If the branch is to a node in another subgraph, then it should be to the first node in that subgraph.

If a branch does not satisfy any of the above conditions, then the subgraph being branched to is broken, ensuring that the branch satisfies condition (b).

In the example, note that node #8 in sg_1 is a branch to node #10, and node #10 is the first node in subgraph sg_2. On the other hand instruction #12 is a branch within subgraph $sg2$. To show the idea of breaking subgraphs, assume that node #5 in subgraph $sg1$ is a branch to instruction #13 in $sg2$, then $sg2$ should have been broken at instruction #13, forcing the nodes 13 to 15 to constitute a third subgraph, Figure 4.9.

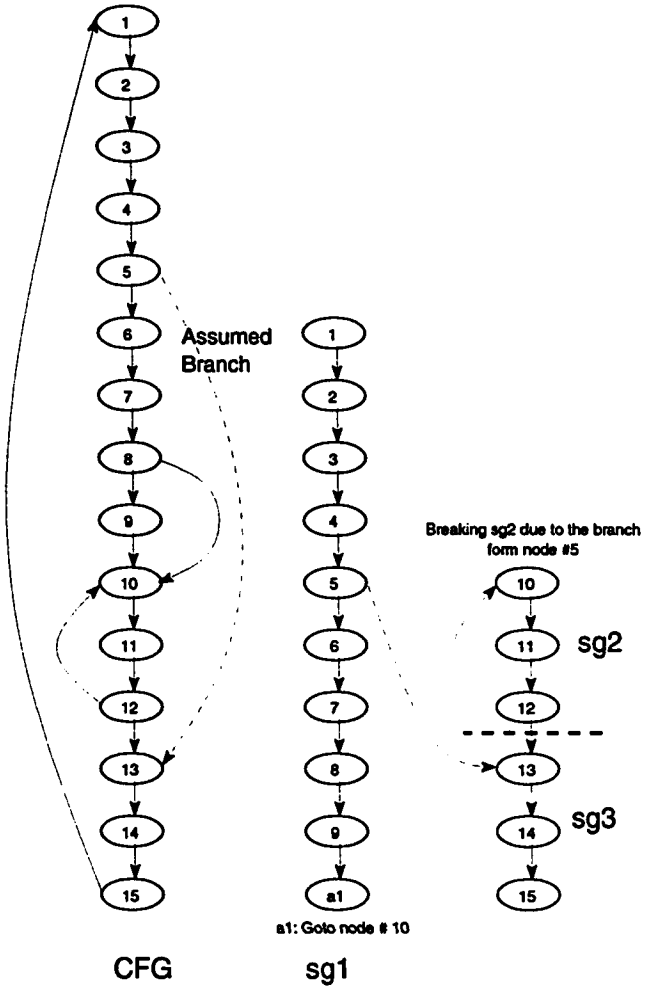

Figure 4.9: Subgraph Breaking.

4.4.2 Scheduling Individual Subgraphs

Each subgraph sg_i is scheduled separately. Scheduling subgraphs consists of the following steps:

1. Convert each subgraph into a DAGs
2. Generate paths of each subgraph.
3. Schedule individual paths.
4. Combine schedules of individual paths.

Converting subgraphs into DAGs

Each subgraph is converted into a Directed Acyclic Graph (DAG). This conversion is done by eliminating edges that branch to other subgraphs. Conditions for these branches are kept in lists associated with each subgraph. Loops back to the first node in each subgraph are eliminated and conditions for looping back are kept in lists as well. These conditions are to be used later for transitions among subgraphs. In the Prefetch example, the branch from node #8 to node #10 is eliminated in sg_1. The same is done for the loop back to node #10 from node #12 in sg_2, as depicted in Figure 4.8 and Figure 4.11.

Generating paths

Paths within a subgraph are generated the same way as in AFAP, or in DLS. An exception is that loops do not exist within a subgraph. Hence no special processing is required for loops.

The algorithm for generating paths is shown in Figure 4.10. The subroutine is called with the following arguments:

- *subgraph*, identifying the subgraph being processed.
- *start_node*, identifying the node in the subgraph from which paths are to be generated.
- *accumulated_cond* which is the accumulated condition for executing the nodes starting at *start_node*
- *accumulated_path* is the last argument in the arguments list, and is a copy of the accumulated path just before calling the routine.

When the routine is first called, the arguments are the subgraph to be processed, the first node in the subgraph, accumulated condition is passed as $TRUE$, and *accumulated_path* as empty. Then the subroutine calls itself when necessary to generate the appropriate set of paths. Note that the type of each node, or equivalently

instruction, is one of the three types of the PAL instructions, namely, conditional branch, unconditional branch and data transfer. A new path is generated only when a conditional branch instruction is encountered, given that the node branched to is within the subgraph being processed. That is, if the conditional branch is outside the subgraph, it is called a state transition. State transitions have been handled in the previous step.

Unconditional branches just ignore the nodes after the branch and continue the path as if the node to which the branch is done (the destination of the branch) comes immediately after the previous node. The nodes between the unconditional branching node and its destination, shall be reached by a branch from a previous node in the subgraph. Otherwise, these nodes would be unreachable, or equivalently dead code. Dead code should have been eliminated in the initial compilation process.

Note also that the conditions for executing each instruction along the path are collected in the variable *accumulated_cond*, and stored with each instruction. Note also that the conditions for executing the individual instructions, or nodes, are updated whenever a conditional branch is encountered. Each node is associated with a set of conditions. Each condition corresponds to a path and is found by ANDing the branching conditions along that path. Whenever any of these conditions is $TRUE$, that node is to be executed in the resulting schedule. This is discussed further in a following section.

```
Algorithm path_generation (subgraph,start_node, accumulated_cond,accumulated_path)
    cur_node=start_node;
    while (more_nodes in the subgraph){
        If (cur_node==conditional_branch){
            temp_cond= accumulated_cond;
            accumulated_cond=accumulated_cond AND branch_cond;
            If (branch_node ∈ subgraph){
                new_path=copy_path(accumulated_path);
                path_generation(subgraph,branch_node,accumulated_cond,new_ path);
            }
            Else {
                add_state_branch(branch_node,accumulated_condition);
            }
            accumulated_cond=(temp_cond AND !branch_cond);
        }
        Else If (cur_node == unconditional_branch){
            If (branch_node ∈ subgraph){
                skip_all_nodes_until_branch_node
            }
            Else {
            add_state_branch(branch_node,accumulated_condition)
            }
        }
        Else {
            add_node to accumulated_path;
            node_execution_cond=accumulated_cond;
        }
        cur_node=next_node_in_subgraph;
    }
    add accumulated_path to the list of paths generated;
}
```

Figure 4.10: Path Generation Algorithm.

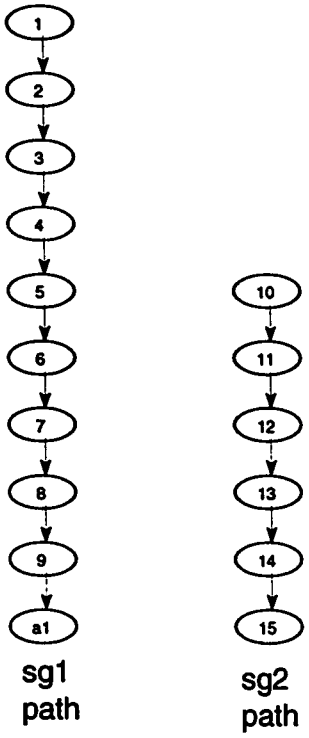

Figure 4.11: Prefetch Example: Paths' Generation in LBS.

The result of path generation is a set of paths, each path consisting of a set of ordered nodes. Each node is associated with an execution condition derived from the conditional branching nodes along the path in the corresponding subgraph. Figure 4.11 illustrates the path generation for the Prefetch example. Note that the total number of paths for the example is only two, compared to three for the AFAP. The number of paths (or traces) for this example is also three in the DLS algorithm. This reduction of the number of paths is due to the fact that the paths do not interact among subgraphs. That is, conditional branches are limited to the subgraph boundaries. This leads to a considerable reduction in the number of paths compared to the other two techniques, AFAP and DLS. This is stated in the following two lemmas.

Lemma 4.3 *The total number of paths (P_{LBS}) generated in LBS is:*

$$P_{LBS} = \sum_{i=1}^{N} p_i$$

Where p_i is the number of paths in subgraph sg_i and N is the total number of subgraphs of the CFG.

Proof of Lemma 4.3 *The proof is obvious from the previous discussion.*

The reduction in the number of paths in LBS compared to AFAP depends on

the CFG topology. LBS usually results in a sizeable reduction in the number of paths and never results in an increase. This is stated in the following lemma.

Lemma 4.4 *The number of paths generated in LBS denoted as N_{LBS} satisfies the following two inequalities:*

$$(a)\ N_{LBS} \leq N_{AFAP}$$

$$(b)\ N_{LBS} \leq N_{DLS}$$

Where N_{AFAP} and N_{DLS} are the number of paths generated in AFAP and DLS, respectively

Proof of Lemma 4.4 *The proof of parts (a) and (b) is identical since the number of paths generated in both DLS and AFAP is the same.*

(a) The equality occurs when the CFG is a single loop. In that case the number of paths generated in both scheduling algorithms will be the same. If the CFG consists of more than one loop, then the number of paths generated by LBS N_{LBS} will be the sum of the paths of these individual loops (refer to lemma 4.3). While in AFAP the number of paths (N_{AFAP}) will be N_{LBS} (since each loop starting node will contribute by the same number of paths in both algorithms) plus a number of paths that is generated due

to the interaction of the paths of different loops. The worst case reduction occurs when the CFG is a single loop, hence no subgraphs are formed and no reduction in the number of paths is gained.

(b) The same argument as in (a) applies to DLS.

Scheduling individual paths

Paths of each subgraph are then scheduled individually in the same way as in DLS. Each path is traced from its starting node and constraints are checked. Whenever a constraint is violated the path is *cut*. Constraints are defined the same way as in AFAP and DLS. An additional node is added after the last node in the cut path. This node is a branch to the first node in the rest of the path. This branch is associated with a condition equal to the condition of executing the node at which cutting took place. This is similar to the nodes' insertion in the subgraph formation step, and is done for the same reasons. Tracing is resumed from the cutting point until another cut is required or the path is consumed. This process will result in a set of *intervals*, each consisting of a set of ordered nodes. Each interval corresponds to a control state. Note that each node might exist in more than one interval, associated with different execution conditions in different intervals. For the prefetch example, no constraints are present in the individual paths, and hence, neither of the paths is cut. Cutting paths this way, i.e., As-Soon-As-Possible, is not as efficient as

AFAP scheduling, however it was shown in [22] that it compares quite favorably with other scheduling techniques. Furthermore, it was felt in [23] that the computational savings justify the slight loss in performance compared to AFAP scheduling. A further limitation is the fact that the subgraph is checked sequentially and no effort is made to optimize the order of operations. This is a classic problem with the path-based approach [23].

Combining schedules of individual paths

Cutting individual paths as in the previous step results in a set of intervals for each path. Intervals that start with the same node are collected together to form a control state. Thus scheduling the operations of the subgraph into control states. Each node is associated within each interval with a different execution condition. These execution conditions are ORed to produce the execution condition for the instruction, or node, within the state.

This results in scheduling each subgraph into a set of states. Transitions among states within each subgraph are taken from the nodes added at each cutting point in the "scheduling individual paths" step.

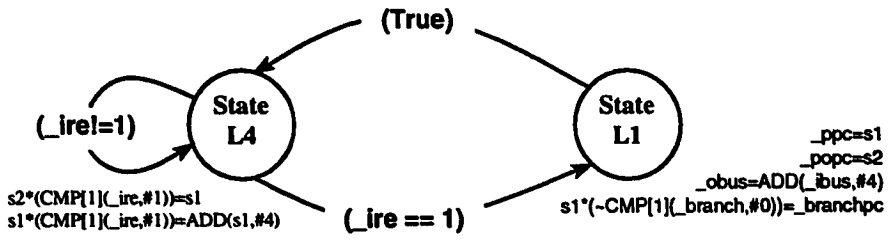

Figure 4.12: Prefetch Example: FSM Controller.

4.4.3 Combining schedules of subgraphs

At this point, each subgraph is scheduled in a set of states. Transitions among states within each subgraph have been decided in a previous step, namely in the nodes added at cut points. To combine the schedules of the individual subgraphs, each subgraph is processed as follows. Transitions outside the subgraph, collected while converting sg_s into $DAGs$, are checked to produce the transition to other subgraphs. Note that each of these transitions is associated with a condition produced in the path generation step. This condition is used as a transition condition to the first state in the subgraph being branched to.

By finishing this step the scheduling is complete, and the FSM controller is produced. Figure 4.12 shows the controller FSM of the prefetch example. The instructions in the figure are in an RTL language called AHPL [35].

Design	Method	States	Paths	State Transitions
Prefetch[1]	AFAP	2	3	3
	DLS	3	3	–
	LBS	2	2	3
Counter[1]	AFAP	1	3	1
	DLS	1	3	1
	LBS	1	3	1
GCD	AFAP	2	7	4
	DLS	2	7	4
	LBS	2	4	4
TLC	AFAP	8	19	18
	DLS	7	19	14
	LBS	5	19	14
DiffEq	AFAP	4	3	–
	DLS	–	–	–
	LBS	3	3	3

Table 4.1: WSHLS Benchmark Results

4.5 Experimental results

4.5.1 Benchmarks

The examples used in this section are shown in Table 4.1. The **Prefetch**[1] is an instruction fetch unit for a microprocessor. It is the example presented in [21] and used in this chapter. The second circuit is a modulo-8 **Counter**[1] with *clear* and *clock* inputs. The rest of the circuits are benchmarks from the 1992 Workshop on High Level Synthesis [33]. **GCD** is a greatest common divisor calculator. The traffic

[1]These are not WSHLS Benchmarks.

light controller **TLC** is specified excluding the required timer (the timer is called). **DiffEq** is a numerical solution to the differential equation $y'' + 3xy' + 3y = 0$. The designs were translated manually from VHDL to C. Neither area constraints nor timing constraints are specified in these implementations. Hence, all possible chaining of operations is utilized.

The three scheduling algorithms that were presented in this chapter are compared in Table 4.1 (namely AFAP, DLS and our algorithm LBS). Two major aspects are compared, the complexity of the algorithm and the quality of the solution they produced.

The complexity of the algorithm is a function of the number of paths processed. The more the number of paths, the more of processing time the algorithm requires to generate its schedule. complexity. Our scheduling algorithm generates the least number of paths compared with the other two. The reason is that our scheduling algorithm ignores the paths generated from the branches among different subgraphs.

The quality of the solution is a function of both the number of sates generated and the number of transitions among states. The number of states implies the complexity of the controller required to realize the schedule. The number of transitions is also an indication of the amount of hardware required to realize the controller. Our scheduling algorithm generated schedules with less number of states and less (or

equal) number of state transitions for the listed benchmarks.

It could be seen from the table that our scheduling algorithm does produce better schedules with less cost. The cost in terms of processing time (measured in the number of paths) is lesser and the quality (measured in number of states) is better. It is worth mentioning at this point that although the number of states of the schedule is less, this does not ensure a fastest execution time for all input sequences. This fastest execution time is guaranteed by the AFAP scheduling algorithm. However, the number of paths of AFAP explodes for realistic examples.

In chapter five, we give the complete specifications of the produced RTL descriptions of these examples.

4.5.2 How much reduction?

It was pointed out in the previous section that our scheduling algorithm, Loop Based Scheduling (LBS), is guaranteed to produce a number of paths that is less than or equal to that produced by As Fast As Possible (AFAP), or Dynamic Loop Scheduling (DLS), (refer to Lemma 4.4). In order to quantify how much less, a number of random graphs were generated. The number of paths processed by both AFAP and LBS for each graph was determined. The comparison results are presented in this subsection.

Generation of random graphs

The graphs processed by the system start at node v_1. Any node v_i in the graph should be reachable from v_1. A node v_i is one of two types:

(1) Data transfer operation, where the control is passed to the successor operation.

(2) Branching operation, where the control is transferred to an operation other than the successor operation if the condition is *true*.

A simulation program was written to generate random graphs that resemble the graphs processed by the system. For a given number of nodes (N) in the graph, if a node v_i is a branching node (i.e., of type 2) with a probability p and is a data transfer (i.e., of type 1) with a probability $1 - p$. A branch node could be one of two types:

(a) Forward Branch.

(b) Backward Branch, or a *Loop*.

For a branching node (i.e. of type 2) if the probability that it is a *Loop*ing node (i.e. of type (b)) is q, then it is a forward branch (i.e. of type (a)) with probability $1 - q$. Hence, starting at node v_1, each node among the N nodes of the graph is identified as one of three types according to p and q as follows:

$$NodeType = \begin{cases} DataTransfer & 1-p \\ ForwardBranch & p-pq \\ Loop & pq \end{cases}$$

Once a node v_i is determined to be a branching node (either forward or backward) the destination node v_d has to be determined. For a forward branch, v_d could be any of the nodes v_{i+2} to v_N (note that if v_{i+1} was the branch destination, then v_i will not be a branch but a data transfer). Hence, if the probability that node v_d is v_j where $(i+1) < j \leq N$, identified as p_f, j would be:

$$j = \lceil (p_f(N-i-2)) \rceil + i + 2$$

where $\lceil a \rceil$ is the smallest integer greater than a. Similarly for a backward branch (i.e. a loop), v_d may be any of the nodes v_1 to v_{i-1} (assuming that v_i will not branch to itself). Hence, if the probability that node v_d is v_j where $1 \leq j < i$ is p_b, then j would be:

$$j = \lceil (p_b(i-2)) \rceil + 1$$

All probabilities, p, q, p_f and p_b are generated using uniformly distributed random number generators.

Comparison results

We determined the number of paths processed by both AFAP and LBS for each random graph. The *Reduction* in the number of paths was plotted as a function of the number of nodes of the graph. The *Reduction* is defined as the ratio of the difference between the number of paths generated in both schedules to the number generated in AFAP. That is,

$$Reduction = \frac{P_{AFAP} - P_{LBS}}{P_{AFAP}}$$

Where P_{AFAP} and P_{LBS} are the number of paths generated by AFAP and LBS scheduling algorithms, respectively. Figures 4.13, 4.14 and 4.15 show some of these results.

In each figure the branching probability, p, was fixed, and the *Reduction* was plot for different values of looping probability. The following comments can be made about these figures.

- A sizeable *reduction* in the number of paths is noticed.
- The value of *Reduction* approaches 100% when the probability of looping increases. This is due to the fact that paths do not interact among different loop bodies in LBS.

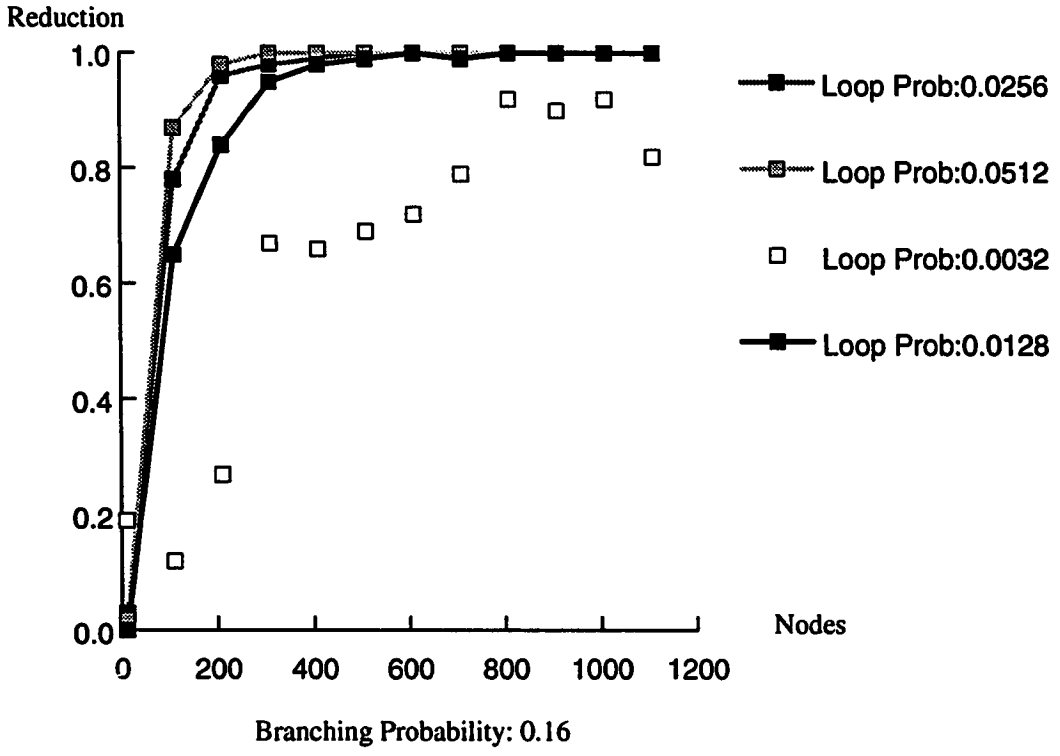

Figure 4.13: The reduction of the number of paths in LBS compared to AFAP.

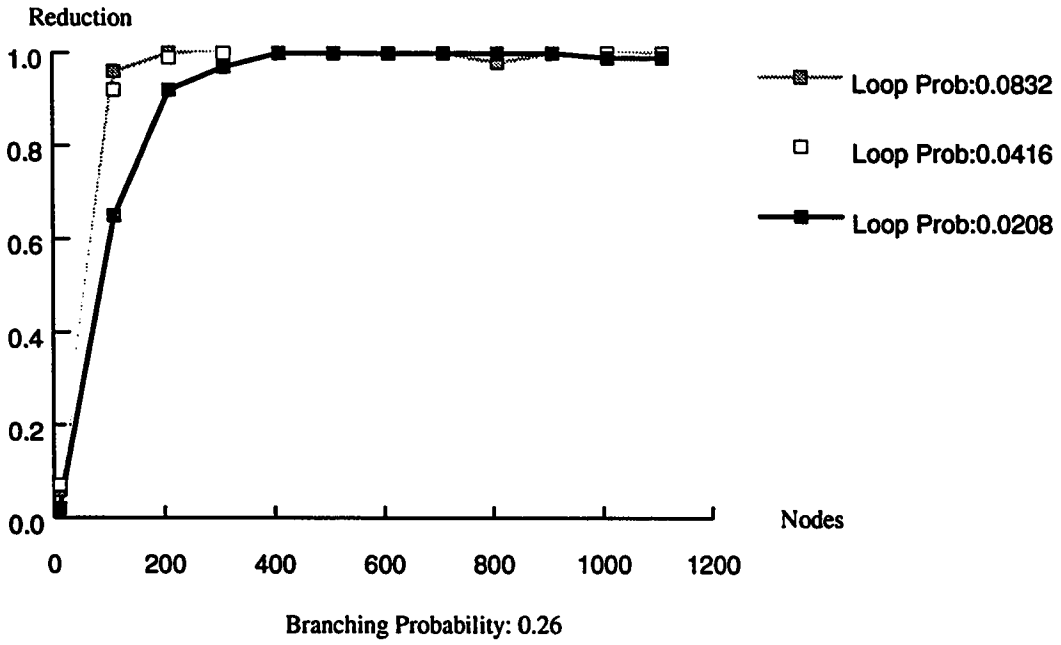

Figure 4.14: The reduction of the number of paths in LBS compared to AFAP.

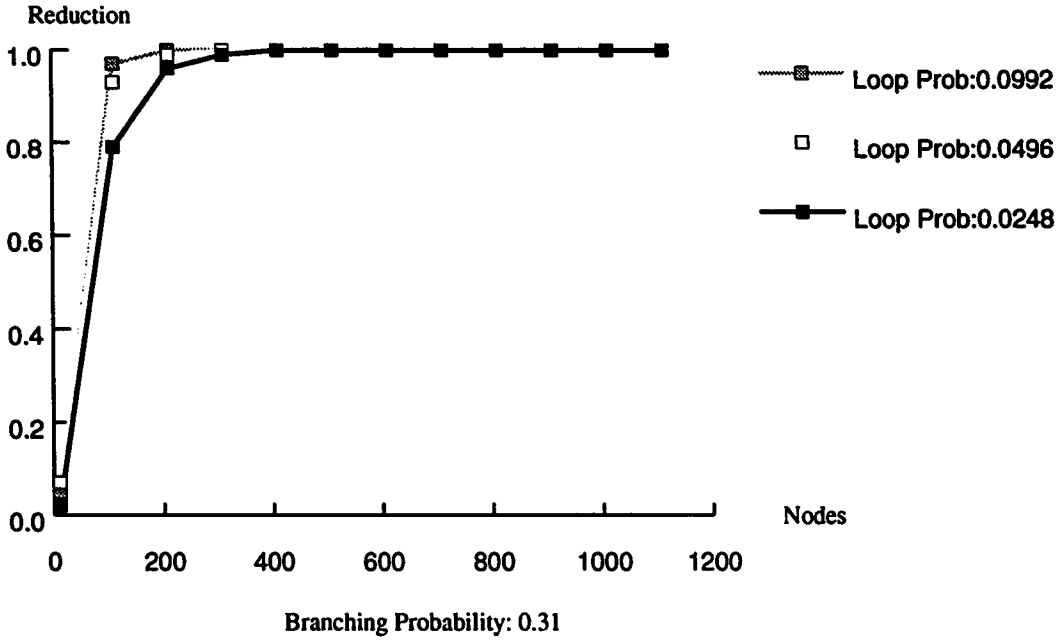

Figure 4.15: The reduction of the number of paths in LBS compared to AFAP.

- Even for small sized graphs with looping probability as low as 0.0032, and a branching probability of 0.16, our algorithm still performs quite well. (See Figure 4.13)

This experiment shows the amount of saving achieved by our scheduling algorithm compared to AFAP and DLS (recall that AFAP and DLS produce the same number of paths). On the other hand, we did not notice a loss in the quality of the solution for the in-house test examples. However a slight loss is expected for a certain type of graphs. We strongly believe that this loss is justifiable by the amount of savings in the calculation cost.

Chapter 5

Data Path Allocation

5.1 Introduction

Data path allocation deals with the problem of minimizing the amount of hardware needed to realize the data path input specification. This problem consists of two main tasks:

- *Module allocation* which is the task of determining the minimum number of hardware modules required to implement the data path.
- *Module binding* is the task of determining which instance of a hardware module is assigned to which operation, variable or data transfer operation.

Due to their inherent inter-dependencies, module allocation and module binding are usually combined in the same step and referred to as just allocation.

The hardware modules needed to realize the data path consist of:

- *Operators or Functional Units (FU)*: They are the hardware units that perform the operations of the data path, like addition, multiplication, etc.
- *Registers or storage elements*: They are used to hold variables or values needed in different control steps.
- *Communication paths or connections* over which the registers and FUs communicate.

Accordingly the allocation task is usually divided into three subtasks.

1. *Operation assignment*, which involves the assignment of the operations of the data path to operators or FUs while ensuring that no more than one operations within a same control step are assigned to the same FU. The objective of this task is usually to minimize the number of FUs used, or to meet a certain constraint on the number of FUs used.
2. *Register allocation*, which binds all variables into a number of registers such that the lifetimes of those variables bound to the same register do not overlap.

The primary objective of this subtask is to minimize the number of registers used.

3. *Inter-connection allocation*, which aims at connecting the FUs and registers in such a way, so as to minimize the interconnection cost.

Because these subtasks are tightly related, the order in which they are performed will affect the quality of the resulting allocation [32]. Decisions made in one subtask will significantly affect the other two.

5.2 Existing approaches to Allocation

Allocation and scheduling are the heart of any HLS system. In this section we describe briefly some of the techniques reported to solve the allocation problem.

FACET [31] solves all three subtasks of the allocation problem using a clique partitioning heuristic method. It does register allocation, then operation assignment and, finally, connection allocation. For register allocation, it builds a graph, where each vertex represents a variable and an edge exists between two vertices if and only if the two corresponding variables can share the same register (i.e., they have disjoint lifetimes). The graph is then partitioned into a number of cliques (a clique is a complete subgraph). The number of cliques partitioned is the number of registers

needed. A register is allocated for those variables corresponding to the vertices in each clique. For operation assignment, FACET again builds a graph where each vertex represents an operation in the input specification and an edge exists between two vertices if and only if the two corresponding operations will not be performed in the same control step. Again, the graph is partitioned into a number of cliques, and to each clique an FU is allocated. For connection allocation, FACET builds a graph where each vertex represents a point-to-point connection. An edge exists between two vertices if and only if the two corresponding connections will never be used simultaneously. After performing a clique partitioning, a bus is allocated for each clique. Because the clique partitioning problem is NP-complete, a heuristic algorithm has been developed in [31] to partition the graphs.

HAL [12] performs operation assignment, then register allocation and, finally, connection allocation. The register allocation is also modeled as a clique partitioning problem. However, weights are associated with the edges of the graph to reflect the preference of register sharing among variables [11].

REAL [28] is a subsystem of MAHA [26] which performs register allocation only. After MAHA has performed a scheduling, REAL allocates registers using the Left Edge Algorithm [27]. The lifetime of each variable is mapped into a net interval as in the channel routing problem. The number of tracks needed by the left-edge algorithm is equal to the number of registers allocated. Variables whose intervals

are in the same track are assigned to the same register. REAL does not handle the operation assignment nor the connection allocation.

Splicer [29] uses a branch-and-bound search to solve all three subtasks. Instead of solving one subtask at a time, it solves them all for each control step. It first assigns FUs to operations in a control step then allocates registers and connections. The system performs a look-ahead search for a user-specified number of steps to make its allocation decisions.

Raj [30] integrates allocation into a scheduler, which performs binding immediately after each control step is scheduled.

5.3 Allocation in our system

In our system, allocation is integrated into the scheduler. No FUs are used in the system, rather operators are assumed, where for each operation, an operator is assumed to be available in a hardware library. Associated with each operator the following information shall be supplied:

- The maximum number of units allowed to be used in the resulting hardware realization.

- The time delay for the operator to execute the corresponding operation.

Operator assignment and connection allocation is done in a First-Come-First-Served (FCFS) fashion. The allocation part of the scheduler is shown in Figure 5.1. The scheduler fetches one operation at-a-time. Assuming no constraints are violated, it checks whether the number of operators already used in the control step is equal to the maximum allowable number specified. If not, an operator is assigned to that operation. The inputs of the operation are connected to the inputs of the operator and the output is connected to the proper destination of the operation or left unconnected for further processing. If the number of operators used in the control step under consideration is equal to the maximum allowable number, the state is broken and the operation is assigned to an operator the same way as if there were no conflict. Then the number of hardware modules of the used operator is incremented in that control step.

Operator assignment and connection allocation done this way assume that different operators are used in different control steps, unless the inputs are the same in both control steps. This will result in a much larger number of operators in the final realization than required. In order to limit the hardware modules to the specified limit the following has to be done. Whenever a new state is formed, the operators used in previous states should be used for the operations of this state, introducing multiplexers at the inputs of those operators. The state flip-flops of the synthesized

```
Procedure Allocation(operation)
Begin
        If number_of_operators in current_state = max[operation] Then
        Begin
                Break_State;
                current_state = new_state;
                number_of_operators = 0;
        End
        Allocate_operator_for_operation;
        number_of_operators = number_of_operators + 1;
        Connect_inputs ;
        Connect_output;
End
```

Figure 5.1: Allocation Algorithm in KHLS.

FSM controller could be used as selection signals for the different multiplexers at the inputs of the operators. Hence a modification to the allocation algorithm has to be incorporated to come up with good allocation results. For the time being this is not yet done in the system, provisions are incorporated for this future work.

As mentioned earlier, the C-compiler does lifetime analysis and several optimization tasks, refer to Section 3.3. The assembly output is fairly optimum in terms of the number of variables used. However it was shown that it contains some redundant dependencies and further optimization is required (see Figure 3.5). This optimization is mainly concerned with the elimination of temporary variables introduced to hold the intermediate results. Register allocation deals with this problem.

Register allocation is again incorporated in the scheduler. The scheduler picks one instruction at a time. Each PAL instruction is either an operation that requires one or two operandi, or a control construct. Control constructs are used to find the execution conditions of operations as explained in Chapter 4. Each operand of an operation is either a variable or a port. Ports are distinguished by an underscore as the first character of their names in the PAL code (and IDS code). Ports need no register allocation and are directly fed to the inputs/output of the operator. Variables are processed independently by the *Register_Allocator*, Figure 5.2.

```
Procedure Register_Allocator;
Begin
    While more_vars in required_vars lists
    Begin
        For all states Do
        Begin
            For each required variable in the required_vars list Do
            Begin
                Schedule all assignments in all states reaching current state;
                Allocate a register for this variable;
                Remove the variable from the required_vars list;
            End
        End
    End
End
```

Figure 5.2: Register Allocation Algorithm.

As mentioned earlier, the scheduler processes paths independently. Each path is traced from the first node to the last node. When a node is picked in the path, each variable of the inputs to the operation is processed as follows. The last assignment to that variable along the path is checked (i.e., the last time the variable appeared at the left hand side of an instruction). In case the variable is not found in the path it is added to a list called *required_vars*. If it is found along the path, then the value it was assigned to is checked to make sure that the time delay of the path added to the current operation delay would not exceed a user specified maximum clock period. If that period is exceeded, then the path has to be cut and the operation has to be scheduled in a different control step, refer to *scheduling individual paths* in section 4.5.2. If the clock period is not exceeded then that last assignment of the variable is used as an input to the current operation. Unless that last assignment is needed some where else, it is a redundant assignment and could be eliminated and the variable need not be allocated to a register. After the scheduling of all paths, each variable in the *required_vars* list is allocated to a register.

Each operation has a destination variable on its left hand side. Once the inputs have been connected to the operator, the output is connected (virtually) to the variable at the left hand side of the instruction. If this variable is needed in a following instruction within the same control step, then the operator output is connected at that time to the proper destination, as explained in the previous paragraph.

```
_prefetch:
        s0 = _ibus              —1
        s0 = s0 + #4            —2
        s4 = _branch            —3
L1:
        _ppc = s1               —4
        _popc = s2              —5
        _obus = s0              —6
        flags = s4 == #0        —7
        if flags goto L4        —8
        s1 = _branchpc          —9
L4:
        s3 = _ire               —10
        flags = s3 == #1        —11
        if !flags goto L4       —12
        s2 = s1                 —13
        s1 = s1 + #4            —14
        goto _prefetch          —15
```

Figure 5.3: Prefetch Example: IDS Code.

Take the first two instructions of the prefetch example, Figure 5.3. When the scheduler picks the first instruction it assigns "*_ibus*" to "*s0*". This assignment is kept for future reference. The time delay for this assignment is assumed to be zero since it is a direct read. When the next instruction is processed, "*s0 = s0 + #4*", "*s0*" is traced along the path and the first assignment is found, its right hand side is fed to the adder input, that is, the input to the adder is assumed to be "*_ibus*" rather than "*s0*". The output of the adder is assigned to *s0* because it is the destination of the addition operation. Again when *s0* is required at instruction number 6 (see Figure 5.3) the same is done and the output of the adder is fed to the *_obus* port. "*s0*" is not needed anywhere else in the path. Neither is it required in the path of subgraph sg_2 (see Figure 4.11) and hence it is actually not needed in the hardware realization of this specification.

On the other hand take instruction #4 , "*_ppc = s1*" in Figure 5.3. Note that "*s1*" is not assigned on that path, hence it is added to the *required_vars* list. After scheduling is over and when this list is checked "*s1*" is found to be required, so it has to be allocated a register in all the states that might lead to the state in which "*s1*" is required. These states happen to be only one, namely the state into which the path of sg_2 is scheduled. Hence, the last assignment in that state for "*s1*" is checked and a register is allocated for that assignment. This is reflected in the instruction ("*s1*(CMP[1](_ire,#1)) = ADD(s1,#4)*" in Figure 4.12. The instruction is in AHPL

format and reads as " *s1 conditioned on the compare result of _ire with #1 gets the addition of s1 and #4*". This allocation will also force the allocation of a register to hold the value "*#4*".

To summarize, register allocation is done after scheduling. However the decision of which variables are to be allocated to which registers is done during scheduling. Variables used in the assembly output are taken as an initial allocation of variables. Redendant variables are eliminated in the scheduling process. The Register Allocator then reduces to allocating the required variables for a control step to registers in all control steps that could reach the control step under consideration.

5.4 Experimental Results

In this section we describe the experimental results of the examples introduced in Chapter 4. The examples are:

1. **Prefetch**
2. **GCD**
3. **Counter**
4. **TLC**
5. **DiffEq**

For each of these examples, the C-code, the PAL-code and the resulting RTL-code in AHPL are given.

In the figures of this chapter, note the following:

1. Ports are defined as *extern* in the C-code. They are identified by an underscore as the first character of the port name in both PAL and AHPL specifications (for example "*_ire*" in *prefetch*, Figure 5.6).

2. Variables are given different names in the internal representation. These names are taken from the Assembly Language produced by the C-compiler.

3. Further processing is required for the AHPL-code to make it acceptable for the KFUPM AHPL-based silicon compiler [36], in order to produce the fabrication masks. The processing needed consists of the following:

   (a) All variables and their sizes have to be defined.

   (b) The Combinational Logic Units (CLUs) library that implements the operators has to be loaded with the RTL-code. That is, CLUs have to be defined.

   (c) Under scores ("_") leading ports' names have to be deleted.

The **Prefetch** and the **GCD** examples have been fed to the AHPL-synthesis system at KFUPM. The specification file and the simulation results are presented

in *AppendixA*.

C-Code	PAL-Code	AHPL-Code

extern int
Xinport,Xoutport,DXport,Aport,Yin
port, Youtport;
extern int Uinport,Uoutport;
DiffEq()
{
int
x_var,y_var,u_var,a_var,dx_var;
int x1,y1,t1,t2,t3,t4,t5,t6;
L1:
x_var=xinport;
a_var=Aport;
dx_var=DXport;
y_var=Yinport;
u_var=Uinport;
while (x_var < a_var){
t1=u_var * dx_var;
t2= 3 * x_var;
t3 = 3 * y_var;
t4 = t1 * t2;
t5 = dx_var * t3;
t6 = u_var - t4;
u_var = t6 - t5;
y1 = u_var * dx_var;
y_var = y_var + y1;
x_var = x_var +
dx_var;
}
}
	_DiffEq:
	L1:
	s3 = _Xinport ;
	s7 = _Aport ;
	s6 = _DXport ;
	s5 = _Yinport ;
	s4 = _Uinport ;
	flags = (s7 < s3) ;
	if (!flags) goto L3;
	L4:
	s2 = s4 ;
	s2 = s2 * s6 ;
	s0 = s3 ;
	s0 = s0 + s0 ;
	s0 = s0 + s3 ;
	s1 = s5 ;
	s1 = s1 + s1 ;
	s1 = s1 + s5 ;
	s2 = s2 * s0 ;
	s1 = s1 * s6 ;
	s0 = s4 ;
	s0 = s0 - s2 ;
	s4 = s0 ;
	s4 = s4 - s1 ;
	s0 = s4 ;
	s0 = s0 * s6 ;
	s5 = s5 + s0 ;
	s3 = s3 + s6 ;
	flags = (s7 < s3) ;
	if (flags) goto L4;
	L3:
	_Xoutport = s3 ;
	_Youtport = s5 ;
	_Uoutport = s4 ;
	goto L1;
		STATE L1 :
		s3=_Xinport
		s7=_Aport
		s6=_DXport
		s5=_Yinport
		s4=_Uinport
		->(~CMP[0](_Aport,_Xinport))/(L3)
		->(~~CMP[0](_Aport,_Xinport))/(L4)
		STATE L4 :
		s4=SUB(SUB(s4,mul(mul(s4,s6),ADD(
		ADD(s3,s3),s3))),mul(ADD(ADD(s5,s5)
		,s5),s6))
		s5=ADD(s5,mul(SUB(SUB(s4,mul(mul(
		s4,s6),ADD(ADD(s3,s3),s3))),mul(ADD
		(ADD(s5,s5),s5),s6)),s6))
		s3=ADD(s3,s6)
		->(CMP[0](s7,ADD(s3,s6)))/(L4)
		->(~CMP[0](s7,ADD(s3,s6)))/(L3)
		STATE L3 :
		_Xoutport=s3
		_Youtport=s5
		_Uoutport=s4
		->(L1)

Figure 5.4: Differential Equation Example.

C-Code	PAL-Code	AHPL-Code

extern int clear,clock,out;
counter()
{
int clk1,clr,out1,clk;
while(1){
if (clear) out1=0;
else {
if ((clk1=clock)!=clk)
out1=out1+1;
clk=clk1;
if (out1==8) out1=0;
}
out=out1;
}
}
	_counter:
	s3 = _clear ;
	L1:
	flags = (s3 == #0) ;
	if (!flags) goto L7;
	s0 = _clock ;
	flags = (s2 == s0) ;
	if (flags) goto L5;
	s1 = s1 + #1 ;
	L5:
	s2 = s0 ;
	flags = (s1 == #8) ;
	if (!flags) goto L4;
	L7:
	s1 = #0 ;
	L4:
	_out = s1 ;
	goto L1;
		STATE L1 :
		s2*(~~CMP[1](_clear,#0))=_clock.
		s1*(OR(~CMP[1](_clear,#0),*AND(~~CMP[1]((ADD(s1,#1)!s1)*(AND(~CMP[1](s2,_clock),~~CMP[1](_clear,#0)),AND(CMP[1](s2,_clock),~~CMP[1](_clear,#0))),#8),~~CMP[1](_clear,#0))*))=#0.
		_out=(#0!s1!ADD(s1,#1))*(*OR(AND(~~CMP[1]((ADD(s1,#1)!s1)*(AND(~CMP[1](s2,_clock),~~CMP[1](_clear,#0)),AND(CMP[1](s2,_clock),~~CMP[1](_clear,#0))),#8),~~CMP[1](_clear,#0)),~CMP[1](_clear,#0))*,AND(CMP[1](s2,_clock),~~CMP[1](_clear,#0)),*AND(~CMP[1](s2,_clock),~~CMP[1](_clear,#0))*).
		->(L1)

Figure 5.5: Counter Example.

C-Code	PAL-Code	AHPL-Code

extern int
branchpc,ibus,branch,ire,ppc,pop
c,obus;
prefetch()
{
int pc,oldpc;
while (1){
ppc=pc;
popc=oldpc;
obus=ibus+4;
if(branch) pc=branchpc;
while(ire!=1);
oldpc=pc;
pc=pc+4;
}
}
	_prefetch:
	s0 = _ibus ;
	s0 = s0 + #4 ;
	s4 = _branch ;
	s3 = _ire ;
	L1:
	_ppc = s1 ;
	_popc = s2 ;
	_obus = s0 ;
	flags = s4 == #0 ;
	if flags goto L4 ;
	s1 = _branchpc ;
	L4:
	flags = s3 == #1 ;
	if !flags goto L4 ;
	s2 = s1 ;
	s1 = s1 + #4 ;
	goto L1 ;
		STATE L1 :
		_ppc=s1
		_popc=s2
		_obus=ADD(_ibus,#4)
		s1*(~CMP[1](_branch,#0))=_branchpc
		STATE L4 :
		s2*(~~CMP[1](_ire,#1))=s1
		s1*(~~CMP[1](_ire,#1))=ADD(s1,#4)
		->(~CMP[1](_ire,#1))/(L4)
		->(~~CMP[1](_ire,#1))/(L1)

Figure 5.6: Prefetch Example.

C-Code	PAL-Code	AHPL-Code

extern int xi,yi,rst,out;
gcd()
{
int x,y;
while (1){
while(!rst);
x=xi;y=yi;
while(x!=y){
if (x<y) y=y-x;
else x=x-y;
}
out=x;
}
}
	_gcd:
	s2 = _rst ;
	L3:
	flags = s2 == #0 ;
	if flags goto L3 ;
	s1 = _xi ;
	s0 = _yi ;
	L10:
	flags = s0 == s1 ;
	if flags goto L6 ;
	flags = s0 < s1 ;
	if !flags goto L7 ;
	s0 = s0 - s1 ;
	goto L10 ;
	L7:
	s1 = s1 - s0 ;
	goto L10 ;
	L6:
	_out = s1 ;
	goto L3 ;
		STATE L3 :
		s1*(~CMP[1](_rst,#0))=_xi
		s0*(~CMP[1](_rst,#0))=_yi
		->(CMP[1](_rst,#0))/(L3)
		STATE L10 :
		s0*(AND(~~CMP[0](s0,s1),~CMP[1](s0,s1)))=SUB(s0,s1)
		s1*(AND(~CMP[0](s0,s1),~CMP[1](s0,s1)))=SUB(s1,s0)
		_out*(CMP[1](s0,s1))=s1
		->(~CMP[1](s0,s1))/(L10)
		->(CMP[1](s0,s1))/(L3)

Figure 5.7: GCD Example.

C-Code

```
int extern Cars,TimeoutL,TimeoutS,StartTimer,HiWay,FarmL,state;
void TLC(){
int newstate,current_state,newHL,newFL,newST;
L1:
    current_state=newstate;
    if (current_state== 0){
        newHL=4;newFL=6;
        if (Cars && TimeoutL) {
            newstate=4;
            newST = 1;
        }
        else {
            newstate=0;
            newST=0;
        }
    }
    else if (current_state== 4){
        newHL=2;
        newFL=6;
        if (TimeoutS) {
            newstate=2;
            newST=1;
        }
        else{
            newstate=6;
            newST=0;
        }
    }
    else if (current_state== 2){
        newHL=6;newFL=4;
        if (!Cars || TimeoutL) {
            newstate=6;
            newST = 1;
        }
        else {
            newstate=2;
            newST=0;
        }
    }
    else if (current_state== 6){
        newHL=6;
        newFL=2;
        if (TimeoutS) {
            newstate=0;
            newST=1;
        }
        else{
            newstate=6;
            newST=0;
        }
    }
    else if (current_state== 7){
        newHL=0;
        newFL=0;
        newstate=0;
        newST=0;
    }
    state=newstate;
    HiWay=newHL;
    FarmL=newFL;
    StartTimer=newST;
    goto L1;
}
```

Figure 5.8: TLC Example: C-Code.

PAL-Code

```
_TLC:
L1:
 s0 = s2    ;
 flags = s2 == #0 ;
 if !flags goto L2 ;
 s4 = #4    ;
 s3 = #6    ;
 s0 = _Cars    ;
 flags = s0 == #0 ;
 if flags goto L20 ;
 s0 = _TimeoutL    ;
 flags = s0 == #0 ;
 if flags goto L20 ;
 s2 = #4    ;
 s1 = #1    ;
     goto L5 ;
L2:
 flags = s0 == #4 ;
 if !flags goto L6 ;
 s4 = #2    ;
 s3 = #6    ;
 s0 = _TimeoutS    ;
 flags = s0 == #0 ;
 if flags goto L7 ;
 s2 = #2    ;
 s1 = #1    ;
     goto L5 ;
L7:
 s2 = #6    ;
     goto L21 ;
L6:
 flags = s0 == #2 ;
 if !flags goto L10 ;
 s4 = #6    ;
 s3 = #4    ;
 s0 = _Cars    ;
 flags = s0 == #0 ;
 if flags goto L12 ;
 s0 = _TimeoutL    ;
 flags = s0 == #0 ;
 if flags goto L11 ;
L12:
 s2 = #6    ;
 s1 = #1    ;
     goto L5 ;
L11:
 s2 = #2    ;
     goto L21 ;
L10:
 flags = s0 == #6 ;
 if !flags goto L15 ;
 s4 = #6    ;
 s3 = #2    ;
 s0 = _TimeoutS    ;
 flags = s0 == #0 ;
 if flags goto L16 ;
 s2 = #0    ;
 s1 = #1    ;
     goto L5 ;
L16:
 s2 = #6    ;
     goto L21 ;
L15:
 flags = s0 == #7 ;
 if !flags goto L5 ;
 s4 = #0    ;
 s3 = #0    ;
L20:
 s2 = #0    ;
L21:
 s1 = #0    ;
L5:
 _state = s2    ;
 _HiWay = s4    ;
 _FarmL = s3    ;
 _StartTimer = s1    ;
     goto L1 ;
```

Figure 5.9: TLC Example: PAL-Code.

AHPL-Code

```
STATE L1 :
 s4*(~~CMP[1](s2,#0))=#4
 s3*(~~CMP[1](s2,#0))=#6
 s2*(AND(AND(~CMP[1](_TimeoutL,#0),~CMP[1](_Cars,#0)),~~CMP[1](s2,#0)))=#4
 s1*(AND(AND(~CMP[1](_TimeoutL,#0),~CMP[1](_Cars,#0)),~~CMP[1](s2,#0)))=#1
 s4*(AND(~~CMP[1](s2,#4),~CMP[1](s2,#0)))=#2
 s3*(AND(~~CMP[1](s2,#4),~CMP[1](s2,#0)))=#6
 s2*(AND(AND(~CMP[1](_TimeoutS,#0),~~CMP[1](s2,#4)),~CMP[1](s2,#0)))=#2
 s1*(AND(AND(~CMP[1](_TimeoutS,#0),~~CMP[1](s2,#4)),~CMP[1](s2,#0)))=#1
 s2*(AND(AND(CMP[1](_TimeoutS,#0),~~CMP[1](s2,#4)),~CMP[1](s2,#0)))=#6
 s4*(AND(AND(~~CMP[1](s2,#2),~CMP[1](s2,#4)),~CMP[1](s2,#0)))=#6
 s3*(AND(AND(~~CMP[1](s2,#2),~CMP[1](s2,#4)),~CMP[1](s2,#0)))=#4
 s2*(OR(AND(AND(AND(CMP[1](_Cars,#0),~~CMP[1](s2,#2)),~CMP[1](s2,#4)),~CMP[1](s2,#0)),AND(AND(AND(AND(
~CMP[1](_TimeoutL,#0),~CMP[1](_Cars,#0)),~~CMP[1](s2,#2)),~CMP[1](s2,#4)),~CMP[1](s2,#0))))=#6
 s1*(OR(AND(AND(AND(CMP[1](_Cars,#0),~~CMP[1](s2,#2)),~CMP[1](s2,#4)),~CMP[1](s2,#0)),AND(AND(AND(AND(
~CMP[1](_TimeoutL,#0),~CMP[1](_Cars,#0)),~~CMP[1](s2,#2)),~CMP[1](s2,#4)),~CMP[1](s2,#0))))=#1
 s2*(AND(AND(AND(AND(CMP[1](_TimeoutL,#0),~CMP[1](_Cars,#0)),~~CMP[1](s2,#2)),~CMP[1](s2,#4)),~CMP[1](s2,
#0)))=#2
 s4*(AND(AND(AND(~~CMP[1](s2,#6),~CMP[1](s2,#2)),~CMP[1](s2,#4)),~CMP[1](s2,#0)))=#6
 s3*(AND(AND(AND(~~CMP[1](s2,#6),~CMP[1](s2,#2)),~CMP[1](s2,#4)),~CMP[1](s2,#0)))=#2
 s2*(AND(AND(AND(AND(~CMP[1](_TimeoutS,#0),~~CMP[1](s2,#6)),~CMP[1](s2,#2)),~CMP[1](s2,#4)),~CMP[1](s2,#
0)))=#0
 s1*(AND(AND(AND(AND(~CMP[1](_TimeoutS,#0),~~CMP[1](s2,#6)),~CMP[1](s2,#2)),~CMP[1](s2,#4)),~CMP[1](s2,#
0)))=#1
 s2*(AND(AND(AND(AND(CMP[1](_TimeoutS,#0),~~CMP[1](s2,#6)),~CMP[1](s2,#2)),~CMP[1](s2,#4)),~CMP[1](s2,#0)
))=#6
->(AND(CMP[1](_Cars,#0),~~CMP[1](s2,#0)))/(L20)
->(AND(AND(CMP[1](_TimeoutL,#0),~CMP[1](_Cars,#0)),~~CMP[1](s2,#0)))/(L20)
->(AND(AND(~CMP[1](_TimeoutL,#0),~CMP[1](_Cars,#0)),~~CMP[1](s2,#0)))/(L5)
->(AND(AND(~CMP[1](_TimeoutS,#0),~~CMP[1](s2,#4)),~CMP[1](s2,#0)))/(L5)
->(AND(AND(CMP[1](_TimeoutS,#0),~~CMP[1](s2,#4)),~CMP[1](s2,#0)))/(L21)
->(OR(AND(AND(AND(CMP[1](_Cars,#0),~~CMP[1](s2,#2)),~CMP[1](s2,#4)),~CMP[1](s2,#0)),AND(AND(AND(AND(~
CMP[1](_TimeoutL,#0),~CMP[1](_Cars,#0)),~~CMP[1](s2,#2)),~CMP[1](s2,#4)),~CMP[1](s2,#0))))/(L5)
->(AND(AND(AND(AND(CMP[1](_TimeoutL,#0),~CMP[1](_Cars,#0)),~~CMP[1](s2,#2)),~CMP[1](s2,#4)),~CMP[1](s2,#0
)))/(L21)
->(AND(AND(AND(AND(~CMP[1](_TimeoutS,#0),~~CMP[1](s2,#6)),~CMP[1](s2,#2)),~CMP[1](s2,#4)),~CMP[1](s2,#0))
)/(L5)
->(AND(AND(AND(AND(CMP[1](_TimeoutS,#0),~~CMP[1](s2,#6)),~CMP[1](s2,#2)),~CMP[1](s2,#4)),~CMP[1](s2,#0)))/
(L21)
STATE L11 :
 s4*(~~CMP[1](s0,#7))=#0
 s3*(~~CMP[1](s0,#7))=#0
->(~CMP[1](s0,#7))/(L5)
->(~~CMP[1](s0,#7))/(L20)
STATE L20 :
s2=#0
->(L21)
STATE L21 :
s1=#0
->(L5)
STATE L5 :
_state=s2 , _HiWay=s4 , _FarmL=s3 , _StartTimer=s1
->(L1)
```

Figure 5.10: TLC Example: AHPL-Code.

Chapter 6

Internal Data Structure (IDS)

6.1 Introduction

The input specification to an HLS system as a behavioral description language is convenient for the human. This is not the case for the HLS system algorithms and routines such as the scheduling or allocation algorithms. For these algorithms to be more efficient, some of the hidden information in the constructs of the language has to be extracted and highlighted. On the other hand some information are not relevant to the algorithm and hiding it makes the processing less cumbersome and more efficient. An example of the latter is the use of names for variables, to make the specification meaningful for the human, is of no importance to an allocator;

an identification number will suffice and will be much more efficient in terms of processing time. Hence, the first task in HLS systems is usually the *compilation* of the input specification into an internal form. The internal form is designed to facilitate the tasks carried out by these algorithms.

The input specification includes two types of information, *control* and *data flow*. Control information is contained in the control constructs like GOTO, CASE, LOOP and other control statements. Control information is also included in the ordering of statements of the input specification. Data flow information is also included in the input specification along with the control, where data propagate through the different paths of the specification code. Thus, this information has to be extracted from the specification and made easily accessible to the system. Control information is usually represented by a *control flow graph*. On the other hand data flow information is mapped into a *data flow graph*.

The control flow graph (CFG) is a directed graph, where the nodes correspond to the operations of the specification and the edges link immediate predecessor-successor pairs. Conditional branching is indicated by more than one successor of a node. The data flow graph (DFG) is some sort of a directed bipartite graph. The nodes are either data or operation nodes. A directed edge between nodes A and node B, if node A puts data into node B.

The CFG and the DFG might be combined into a single graph that incorporates both information. This type of graphs is referred to as Control Data Flow Graph (CDFG). CDFGs are frequently used in HLS systems.

The HAL system [12] of Carleton University uses a CDFG as an internal form. The Carnegie Mellon University (CMU) [1, 5, 6, 7, 9, 10] HLS systems use the Value Trace (VT) as an internal form. The VT is a CDFG. The IBM's Yorktown Silicon Compiler (YSC) [25], uses separate CFG and DFG for the internal representation. Its internal form is called Yorktown Internal Format (YIF). Stanford's Flamel system [18] uses a CDFG, called *dacon*, as an internal representation.

In our system we use a combined control and data flow structure to represent the input specification internally. We call this form of a CDFG the *Internal Data Structure (IDS)*. IDS is explained in detail in the rest of this chapter.

6.2 IDS description

The IDS was carefully designed to meet the requirements of the system. It is imperative that the IDS provides an easy way to access the required data for the various system algorithms. Hence the requirements of each algorithm (i.e. the scheduler, allocator and the tasks carried alongside these modules) has been incorporated into

the IDS.

The first task carried by the IDS translator is to build the *cross reference table*, Figure 6.1. The variable names are converted into integers in this step. Integers are much more efficient in terms of processing, and storage. The variable name need be stored only once along with a corresponding integer. This information is stored in the cross reference table.

```
struct cross_ref_table{
    char                    name[VAR_LEN];
                            value;
    struct cross_ref        *next_item;
};
```

Figure 6.1: Cross reference table.

The heart of the system is the scheduler. The other parts carry their tasks from within the scheduler. Therefore the IDS was designed to optimize the various tasks carried out by the scheduler. The IDS primary construct is the *state*. A state in the IDS corresponds to a state in the attempted schedule. The scheduler starts from an initial schedule. This schedule is the subgraphs presented in Chapter 4. Hence the IDS translator carries out the task of subgraph generation (refer to Chapter 4 for more detail).

```
struct state{                       (* A state initially is a loop body *)
 char              label[VAR_LEN];(* A state identifier *)
 BLK_PTR           blks;           (* A pointer to the blocks of this loop body *)
 BRA_PTR           branches;       (* A pointer to the branching instructions out of this state *)
 REACH_PTR         reach;          (* A pointer to all branches among this state *)
 RCHR_PTR          rchrs;          (* A pointer to all states reaching this state *)
 HW_PTR            hw;             (* A pointer to the amount of Hardware used in this state *)
 VAR_PTR           reqs;           (* A list of all required variables for this state *)
 GOTO_PTR          gotos;          (* A list of all goto instructions of this state *)
 CND_REF_PTR       cnd_ref_table;  (* A table of each condition and its integer reference value *)
 struct state_list *next_state;    (* A pointer to the next state *)
};
```

Figure 6.2: *State* data structure.

The *state* data structure is shown in Figure 6.2. The variable *label* holds the identifying name of the state. This name is taken from the PAL code. The PAL code assigns a label for each loop entrance point. Remember that each *state* corresponds to a subgraph, which starts at a loop entrance point of the PAL specification. An exception for this is the first subgraph, which starts at the first instruction of PAL. The first instruction is always labeled in the PAL specification, whether it is a loop entrance instruction or not.

The prefetch example of Figure 4.1 is repeated in Figure 6.4 and used to explain the IDS translation. The translation of this specification into *states* is shown in Figure 6.3. The field *next_state* is a pointer to the next subgraph of the specification. This field corresponds to the nodes added after each subgraph in the *subgraph generation step*, presented in chapter 4.

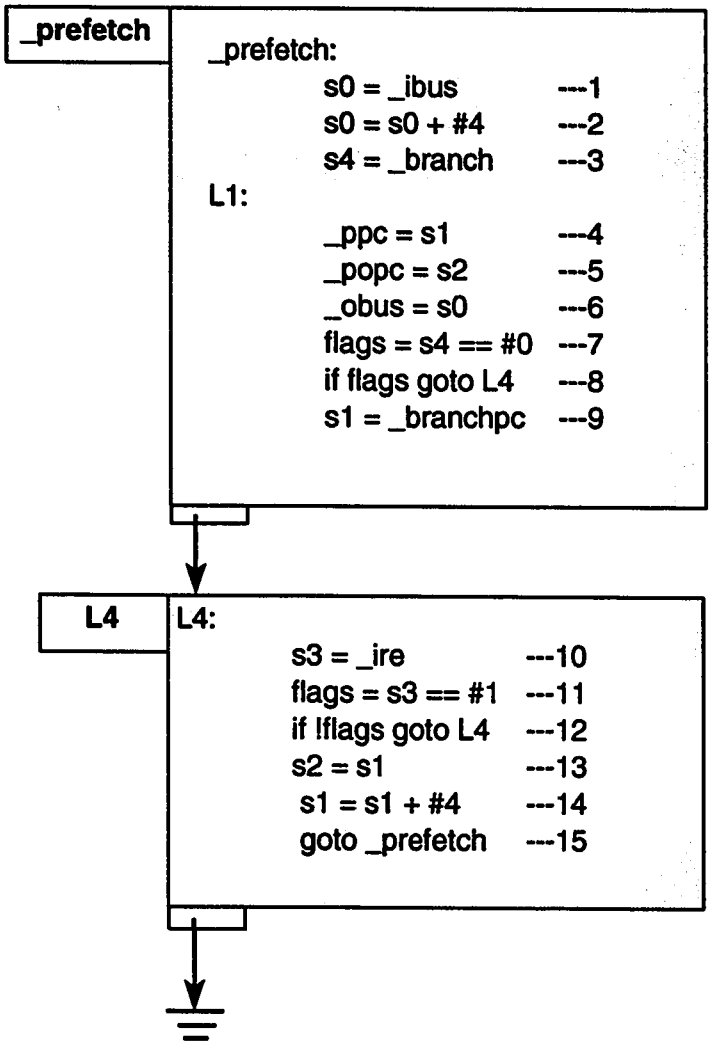

Figure 6.3: Prefetch Example: Translation into *states* of the IDS.

```
_prefetch:
        s0 = _ibus              —1
        s0 = s0 + #4            —2
        s4 = _branch            —3
L1:
        _ppc = s1               —4
        _popc = s2              —5
        _obus = s0              —6
        flags = s4 == #0        —7
        if flags goto L4        —8
        s1 = _branchpc          —9
L4:
        s3 = _ire               —10
        flags = s3 == #1        —11
        if !flags goto L4       —12
        s2 = s1                 —13
        s1 = s1 + #4            —14
        goto _prefetch          —15
```

Figure 6.4: Prefetch Example: PAL Code.

6.2.1 The field *cnd_ref_table*

Conditions generated in each state are mapped into integers. The cross reference table between conditions and corresponding integers is stored in the table pointed to by *cnd_ref_table*, Figure 6.5. A negative integer implies that the condition should be inverted.

```
struct cnd_ref_list{
    int                     cond_ref;
    char                    cond_value[INST_LEN];
    struct cnd_ref_list     *next_cr;
};
```

Figure 6.5: Conditions reference table.

6.2.2 The *reqs* field

This field holds a list of the required variables of the state from other states. These variables are identified by the scheduler and the register allocator, refer to Chapters 4 and 5. A required variable for a state is a variable that is used in the state before being set to a value, or a variable that is required for a state that is reached by the current state.

6.2.3 The *hw* field

This pointer points to a list of hardware units used in the state. This information is needed in the scheduler to decide whether a user specified limit on area is violated or not.

6.2.4 The *gotos* and *branches* fields

The *gotos* field points to a list of the *GOTO* instructions (both conditional and unconditional) in the state. Each goto is associated with a condition of its own and the ANDing of the conditions along the path leading to the execution of that instruction. This condition is used as an inter-state branching condition when the state is cut by the scheduler. Another field in the gotos data structure is the *st_num* (see Figure 6.6). This field holds the statement number from which the *goto* took place. Branching among the states is stored in the field *branches* with the proper condition for each branch. The data structure for these fields is shown in Figure 6.6

```
struct goto_list{
    char                label[VAR_LEN];
    CND_PTR             cond;
    int                 st_num;
    struct goto_list    *next_goto;
};
struct branches_list{
    char                inst[INST_LEN];
    struct bra_list     *next_bra;
};
```

Figure 6.6: *branches* and *gotos* data structure.

6.2.5 The *rchrs* field

```
struct rchr_list{
    char                label[VAR_LEN];
    ANDS_PTR            cnd;
    int                 st_num;
    struct rchr_list    *next_rchr;
};
```

Figure 6.7: The *rchrs* data structure.

The *rchrs* (reachers) pointer points to a list of states that reach this state. The record of each reacher, Figure 6.7, contains as well two more fields (*cnd* and *st_num*) which are not used in this context. These fields are used when the list is used in the *reach* data structure explained next.

6.2.6 The *reach* field

```
struct reach_list{
    char                label[VAR_LEN];
    struct reach_list   *next_reach;
    RCHR_PTR            reachers;
};
```

Figure 6.8: The *reach* data structure.

This field points to a list shown in Figure 6.8. The list holds the branching relationships among the blocks of the state. The field *label* holds the label of the block under consideration. The field reachers holds the information about all blocks reaching the one under consideration. For each reacher the following information is stored (refer to Figure 6.7).

- A *label* field identifying the reaching block.
- A *st_num* field identifying the statement in the reaching block where the branch took place.
- A *cnd* field that holds the condition for that branch.

6.2.7 The blocks field, *blks*

```
struct blk_list{
    char                label[VAR_LEN];
    STMTS_PTR           stmts;
    struct blk_list     *next_blk;
};
```

Figure 6.9: *Block* data structure.

In the PAL specification, there might be several segments of code each of them having a different label. These labels are used in the PAL for branching instructions. These segments are identified in the IDS as *blocks*. *Blocks* are the building units of *states* in the IDS. The *block* data structure is shown in Figure 6.9.

The *label* field is an identification field. It is given the label initially found in the PAL description. The *next_blk* field is a pointer pointing the following block of the PAL specification within the *state*. The *stmts* field stores the statements of the specification as they appear in the PAL specification. The *stmts* field is explained in the following subsection.

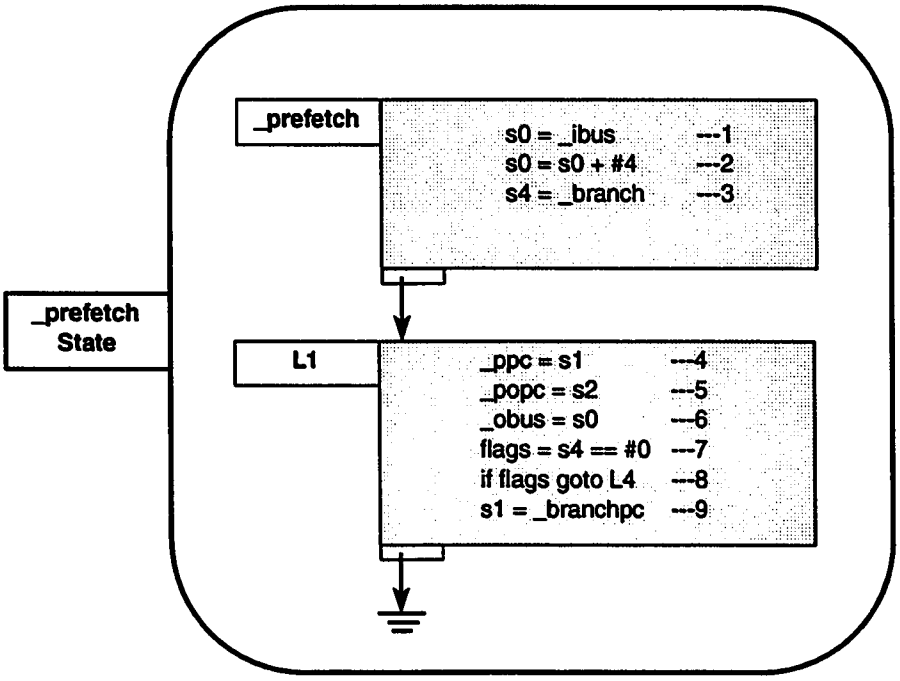

Figure 6.10: Prefetch Example: Blocks of the _prefetch state.

The state labeled *_prefetch* in Figure 6.3 is shown in Figure 6.10 as a set of blocks.

The *stmts* field

Each block consists of a number of statements. PAL statements are read from the PAL specification file one at a time as they appear in the code. Each PAL statement is analyzed as soon as it is read. This analysis converts the statement into the IDS form of the statement. The statement IDS form, *stmt* is shown in Figure 6.11. The fields of this data structure are explained next.

```
struct stmt{
    int             num;
    int             IF_cond,start;
    int             delay;
    int             LHS,RHS1,RHS2,OPR,type;
    char            label[VAR_LEN];
    CND_PTR         LHS_conds;
    SRC_PTR         RHS1_srcs,RHS2_srcs;
    struct stmt     *next_stmt;
};
```

Figure 6.11: The *stmt* data structure

The field *num*

Each statement in the PAL code is given a distinct identification number. This identification number is stored in the field *num*. All references to any statement use this identification number.

The field *type*

This field is an *integer* field that stores the type of the instruction. Recall that any PAL instruction is one of three types:

- Data transfer : The *type* value of which is 0.
- Conditional branch : The *type* value of which is 1.
- Unconditional branch: The *type* value of which is 2.

The field *IF_cond*

The conditional branch instruction in PAL has the form *"IF condition GOTO label"*. The *"condition"* takes only one of two values : *"flags"* or *"!flags"*. That is, the branch will take place if the result of a previous comparison operation is either *TRUE* or *FALSE*. The comparison operations always store their results in a variable called

flags. This is due to the fact that PAL is translated from assembly language, where a flags register is usually invoked to make the branching decisions.

The field *IF_cond* stores the value "1" if the *condition* is *flags* and the value "0" if the *condition* is *!flags*.

The field *label*

This field is associated with branching instructions. It stores the label into which the branch is done. It is a redundant field if the instruction is not a branch.

The field *LHS*

This field stores the integer reference value of the *L*eft *H*and *Side* variable of a data transfer instruction.

The field *RHS1*

This field stores the integer reference value of the first *R*ight *H*and *Side* variable of a data transfer instruction.

The field *RHS2*

This field stores the integer reference value of the second *R*ight *H*and *Side* variable of a data transfer instruction.

The field *OPR*

This field stores the integer reference value of the operation of a data transfer instruction.

The field *start*

This field is a Boolean variable that hold the value *TRUE* if the instruction is in the starting sequence of the PAL code, and *FALSE* otherwise. The starting sequence is the sequence that appears between the first instruction of the code and the first loop entrance instruction. This sequence of instructions will be executed only once, that is the first time the code is executed. The instructions in the block *_prefetch* are of this type of instructions. These instructions are usually the instructions moved outside loops by the compiler to minimize the run time of loops. These instructions are the candidate instructions for reinsertion inside loops. The instruction *s3 = _ire* is of this type. Remember that this instruction is moved inside the block labeled *L4*

in the CFG of this example (see Figure 4.3).

The field *delay*

This field is needed in the scheduling algorithm for the chaining of operations along a certain path. The scheduler takes as input the maximum clock period allowed in the final realization. Time delay of each operation is given in the hardware library. Hence this field is used to store the accumulated time delay of the data path realizing the associated instruction.

The fields *RHS1_srcs* and *RHS2_srcs*

These two fields are pointers to the operands of the operation. They are of type *SRC_PTR*. The type *SRC_PTR* is explained next.

The field *LHS_conds*

This field is a pointer to the different execution conditions of the statement. It is built by the scheduler. This pointer is of type *CND_PTR*. The type *CND_PTR* is explained later.

The type *SRC_PTR*

```
struct src_list{
    char            src[INST_LEN];
    CND_PTR         cond;
    struct src_list *next_src;
};
```

Figure 6.12: The *SRC_PTR* data structure record

The data structure pointed to by a *SRC_PTR* is shown in Figure 6.12. This list holds the different paths that the source of an operation might have followed. Each path is stored in a record of the list. The field that stores the path is called *src* and is of type *char*. The use of this type is due to the fact that the data path is converted into the output format in *AHPL* and stored in this variable. Each path is associated with a different execution condition stored in the list pointed to by the condition pointer *cond* of type *CND_PTR*.

The type *CND_PTR*

The *CND_PTR* is a pointer to a condition. The condition is the result of an AND-OR operations. Conditions are generated from the different paths reaching a certain instruction. Associated with each path is a set of conditions, the ANDing of which is the condition of executing the associated instruction. If any of the paths leading

```
struct cnd_list{
    ANDS_PTR        ands;
    struct cnd_list     *next_cnd;
};


struct ands_list{ /* pointed to by ANDS_PTR */
    int                 cond_ref;
    struct ands_list    *next_and;
};
```

Figure 6.13: The *CND_PTR* data structure records

to a certain statement is activated, that statement should be executed. Thus the ORing of the different paths conditions yields the execution condition for a specific statement. Figure 6.13 shows the data structure of *CND_PTR*. Figure 6.14 shows an example of how conditions are stored. In the figure the conditions "a –d" are stored as numbers and not characters.

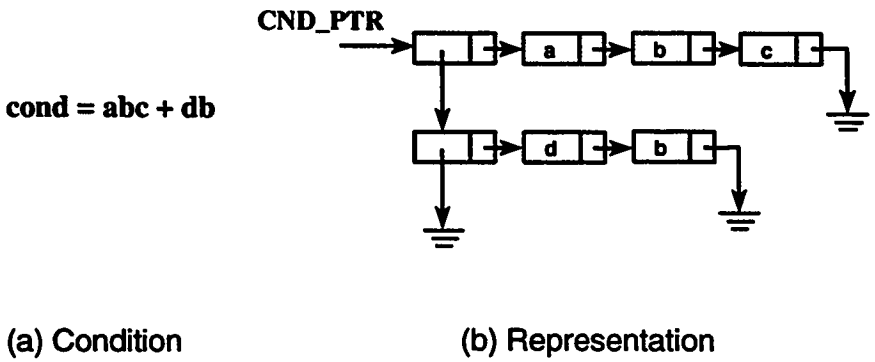

Figure 6.14: Storage of conditions.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

High Level Synthesis is a complex process. To overcome its complexity, a new step-wise approach is adopted. A series of transformations move the input specification into the target specification (RTL descriptions).

We have built an HLS system that incorporates the following features:

- The system can accomodate any ordinary high level programming language.
- The system produces RTL descriptions of input specifications written in the

C-language. It synthesizes these descriptions trying to minimize the schedule length restricted by a set of user supplied constraints. The constraints are either on area (represented by the number of hardware units to be used in the implementation) or on timing (represented by the minimum clock frequency) or on both.

- The system is machine independent.
- It incorporates a superior scheduling algorithm.
- An optimization register allocation algorithm is incorporated in the system.
- The system produced quality designs for several benchmarks.
- AHPL hardware description language is used as an RTL target specification language.
- The supported C-language is quite comprehensive.

The novel idea of this work is the use of the ordinary language compiler to produce the first intermediate form. This lead to the utilization of the tasks carried by the compiler such as:

1. Dead Code Elimination.
2. Constant Propagation

3. Common Sub-expression Elimination

4. Code Motion

5. Variable Disambiguation

The C-language is used in this work as a representative specification language. The supported C is quite comprehensive. However, some restrictions are still imposed on the input specification. All control constructs are allowed in the current implementation. Subroutine calls are allowed with some restrictions. Subroutines are currently expanded in line. Certain data types such as arrays and strings are supported but not handled efficiently.

This work introduced the Pseudo Assembly Language (PAL) as an actual input specification language for the system. PAL made the system machine independent and made it easier to carry out the various tasks of the HLS process.

Scheduling algorithms are classified in this work into two classes: Operation Based Scheduling algorithms and Path Based Scheduling algorithms. A new Path Based Scheduling algorithm is introduced. We call this algorithm Loop Based Scheduling (LBS). The experimental results show an exponential decrease in the number of paths processed by this algorithm compared to other reported Path Based Scheduling algorithms. The number of paths is a measure of the complexity of the

algorithm. The scheduler includes operation chaining as well.

The system includes an optimizing register allocator. The register allocator starts by the initial allocation performed by the language compiler. It then optimizes that solution by minimizing the number of registers used in the realization.

The operator assignment and interconnection allocation is not investigated in this work. A provision is made in the system design for accomodating these allocation techniques. For the time being an allocation technique like the ASAP is used.

The Internal Data Structure was carefully designed to make the tasks carried by the system modules easier. A comprehensive description of the IDS is presented.

Several benchmarks have been used in the system. The results are quite interesting both in the complexity of processing and the quality of the solution. Several examples have been simulated and fabrication masks were produced. However, no chips are yet fabricated.

7.2 Future Work

Several open problems need to be investigated within the context of this new synthesis methodology. Among these problems are the following.

- *Word Length.* At the time being all hardware units are assumed to have the same word length. This word length is supplied by the user. A more efficient and suitable way has to be found to determine the word length of the different components of the realization hardware. The Flamel system of Stanford University and the HARP system use simulation to determine the word length.

- *Arrays.* Arrays are handled in the system as separate registers. A better solution might be to use memory modules to realize arrays, or even multi-port memories. A further study is required for this issue.

- *Bit-wise Operations.* Bit-wise operations are not supported in the system. A way of supporting such operations is required.

- *Allocation.* Operator and interconnection allocation are not well handled in the system. A solution for this is proposed in Chapter 5. However, the solution was not implemented.

- *Subroutines and Stacks.* Subroutines are expanded inline in the system. A better way of implementing subroutines is needed. The possibility of modelling subroutines as asynchronous hardware modules has to be studied. Parameter passing is a limitation for such a solution. However, the stack used in the assembly languages might provide a good communication media that would overcome this problem.

- *User Interface.* Although an experienced user may be able to affect the decisions of the system by manipulating the PAL specification, no attention was paid for user interface. An interactive graphical interface and multi-cycle design style shall help producing more efficient solutions.

- The classification of scheduling algorithms into operation based and path based triggered the idea of trying to apply some operation based algorithms like Force-Directed Scheduling to the control flow graphs in a path based fashion. The idea would be probably to incorporate the mutual exclusion information of paths execution into the force measure. The force is an indication of the gain yielding from assigning an operation to a certain control step.

Appendix A1

This appendix shows the simulation results for two circuits: the *GCD* and the *Prefetch*. These circuits are explained in Chapter 5. The first is the GCD circuit, which calculates the Greatest Common Divisor of two numbers. The second is the prefetch circuitry explained in the chapters of this report. Given next are the AHPL model and the simulation results for both examples.

All necessary declarations are incorporated in the models. Combinational Logic Units are defined in the system.

The GCD Example

Simulation Results

```
MODULE    : GCD.
  MEMORY     : S0{8} ; S1{8}.
  EXINPUTS   : XI{8};YI{8};RST{1};CLOCK.
  CLUNITS    : ADD{8}  <: ADDER<. 8 .>.
  CLUNITS    : COM8{8}  <: COMPAR<. 8 .>.
  CLUNITS    : COM1{1}  <: COMPAR<. 1 .>.
  CLUNITS    : SUB{8}.
  OUTPUTS    : OUT{8}.
BODY
  SEQUENCE      : CLOCK.
  1     S1 * (~COM1{1}(RST;\0\)) <= XI;
        S0 * (~COM1{1}(RST;\0\)) <= YI;
        => (COM1{1}(RST;\0\)) / (1).

  2     S0 * ((~COM8{2}(S0;S1)) & (~COM8{1}(S0;S1)) )<= SUB(S0;S1);
        S1 * (( COM8{2}(S0;S1)) & (~COM8{1}(S0;S1)) )<= SUB(S1;S0);
        OUT * ( COM8{1}(S0;S1) ) <= S1;
        => (~COM8{1}(S0;S1),COM8{1}(S0;S1)) / (2,1).

   ENDSEQUENCE
   CONTROLRESET (~RST) / (1).
END.
```

```
1KING FAHD UNIV OF PETROLEUM & MINERALS, COLLEGE OF COMPUTER SCIENCE
 UNIVERSAL AHPL SIMULATOR OUTPUT       DATE: SAT 13 NOV., 1993       TIME:
12:20:50


       UNIVERSAL AHPL FUNCTIONAL LEVEL SIMULATOR OUTPUT:

            RST
            | S0
            | |  S1
            | |  |  XI
            | |  |  |  YI
            | |  |  |  |  OUT
 CLOCK #    | |  |  |  |  |
       0    0 00 00 00 00 00
            **** ACTIVE STEPS IN CLOCK (   1) : SYST/STEP ****
                                                  1/   1
       1    0 00 00 0F 14 00
            **** ACTIVE STEPS IN CLOCK (   2) : SYST/STEP ****
                                                  1/   1
       2    0 00 00 0F 14 00
            **** ACTIVE STEPS IN CLOCK (   3) : SYST/STEP ****
                                                  1/   1
       3    0 00 00 0F 14 00
            **** ACTIVE STEPS IN CLOCK (   4) : SYST/STEP ****
                                                  1/   1
       4    0 00 00 0F 14 00
            **** ACTIVE STEPS IN CLOCK (   5) : SYST/STEP ****
                                                  1/   1
       5    0 00 00 0F 14 00
            **** ACTIVE STEPS IN CLOCK (   6) : SYST/STEP ****
                                                  1/   1
       6    1 00 00 0F 14 00
            **** ACTIVE STEPS IN CLOCK (   7) : SYST/STEP ****
                                                  1/   2
       7    1 14 0F 0F 14 00
            **** ACTIVE STEPS IN CLOCK (   8) : SYST/STEP ****
                                                  1/   2
       8    1 05 0F 0F 14 00
            **** ACTIVE STEPS IN CLOCK (   9) : SYST/STEP ****
                                                  1/   2
       9    1 05 0A 0F 14 00
            **** ACTIVE STEPS IN CLOCK (  10) : SYST/STEP ****
                                                  1/   2
      10    1 05 05 0F 14 00
            **** ACTIVE STEPS IN CLOCK (  11) : SYST/STEP ****
                                                  1/   1
      11    1 05 05 04 08 05
            **** ACTIVE STEPS IN CLOCK (  12) : SYST/STEP ****
                                                  1/   2
      12    1 08 04 04 08 05
            **** ACTIVE STEPS IN CLOCK (  13) : SYST/STEP ****
                                                  1/   2
```

```
13    1 04 04 04 08 05
      **** ACTIVE STEPS IN CLOCK (  14) : SYST/STEP ****
                                            1/   1
14    1 04 04 04 08 04
      **** ACTIVE STEPS IN CLOCK (  15) : SYST/STEP ****
                                            1/   2
15    1 08 04 04 08 04
      **** ACTIVE STEPS IN CLOCK (  16) : SYST/STEP ****
                                            1/   2
16    1 04 04 04 08 04
      **** ACTIVE STEPS IN CLOCK (  17) : SYST/STEP ****
                                            1/   1
17    1 04 04 0A 14 04
      **** ACTIVE STEPS IN CLOCK (  18) : SYST/STEP ****
                                            1/   2
18    1 14 0A 0A 14 04
      **** ACTIVE STEPS IN CLOCK (  19) : SYST/STEP ****
                                            1/   2
19    1 0A 0A 0A 14 04
      **** ACTIVE STEPS IN CLOCK (  20) : SYST/STEP ****
                                            1/   1
20    1 0A 0A 0A 14 0A
      **** ACTIVE STEPS IN CLOCK (  21) : SYST/STEP ****
                                            1/   2
21    1 14 0A 0A 14 0A
      **** ACTIVE STEPS IN CLOCK (  22) : SYST/STEP ****
                                            1/   2
22    1 0A 0A 0A 14 0A
      **** ACTIVE STEPS IN CLOCK (  23) : SYST/STEP ****
                                            1/   1
23    1 0A 0A 0A 14 0A
      **** ACTIVE STEPS IN CLOCK (  24) : SYST/STEP ****
                                            1/   2
24    1 14 0A 0A 14 0A
      **** ACTIVE STEPS IN CLOCK (  25) : SYST/STEP ****
                                            1/   2
25    1 0A 0A 0A 14 0A
      **** ACTIVE STEPS IN CLOCK (  26) : SYST/STEP ****
                                            1/   1
26    1 0A 0A 0A 14 0A
      **** ACTIVE STEPS IN CLOCK (  27) : SYST/STEP ****
                                            1/   2
27    1 14 0A 14 1E 0A
      **** ACTIVE STEPS IN CLOCK (  28) : SYST/STEP ****
                                            1/   2
28    1 0A 0A 14 1E 0A
      **** ACTIVE STEPS IN CLOCK (  29) : SYST/STEP ****
                                            1/   1
29    1 0A 0A 14 1E 0A
      **** ACTIVE STEPS IN CLOCK (  30) : SYST/STEP ****
                                            1/   2
```

```
30    1 1E 14 14 1E 0A
      **** ACTIVE STEPS IN CLOCK (  31) : SYST/STEP ****
                                            1/   2
31    1 0A 14 14 1E 0A
      **** ACTIVE STEPS IN CLOCK (  32) : SYST/STEP ****
                                            1/   2
32    1 0A 0A 14 1E 0A
      **** ACTIVE STEPS IN CLOCK (  33) : SYST/STEP ****
                                            1/   1
33    1 0A 0A 14 1E 0A
      **** ACTIVE STEPS IN CLOCK (  34) : SYST/STEP ****
                                            1/   2
34    1 1E 14 14 1E 0A
      **** ACTIVE STEPS IN CLOCK (  35) : SYST/STEP ****
                                            1/   2
35    1 0A 14 14 1E 0A
      **** ACTIVE STEPS IN CLOCK (  36) : SYST/STEP ****
                                            1/   2
36    1 0A 0A 14 1E 0A
      **** ACTIVE STEPS IN CLOCK (  37) : SYST/STEP ****
                                            1/   1
37    1 0A 0A 14 1E 0A
      **** ACTIVE STEPS IN CLOCK (  38) : SYST/STEP ****
                                            1/   2
38    1 1E 14 14 1E 0A
      **** ACTIVE STEPS IN CLOCK (  39) : SYST/STEP ****
                                            1/   2
39    1 0A 14 14 1E 0A
      **** ACTIVE STEPS IN CLOCK (  40) : SYST/STEP ****
                                            1/   2
40    1 0A 0A 14 1E 0A
      **** ACTIVE STEPS IN CLOCK (  41) : SYST/STEP ****
                                            1/   1
::::: PROGRAM REACHED THE CLOCKLIMIT. AHPL SIMULATION STOPS. :::::
```

The Prefetch Example

Simulation Results

```
MODULE   :PREFETCH.
  MEMORY     : S1{8} ; S2{8}.
  EXINPUTS   : IRE{1};BRANCH{1};BRANCHPC{8};IBUS{8}.
  EXINPUTS   : CLOCK;RESET.
  CLUNITS    : ADD{8}  <: ADDER<. 8 .>.
  CLUNITS    : COM{8}  <: COMPAR<. 8 .>.
  OUTPUTS    : PPC{8};POPC{8};OBUS{8}.
BODY
  SEQUENCE      : CLOCK.
  1     PPC     <= S1;
        POPC    <= S2 ;
        OBUS    <= ADD(IBUS;\0,0,0,0,0,1,0,0\);
        S1 * (~COM{1}(BRANCH;\0\)) <= BRANCHPC.

  2     S2 * (COM{1}(IRE;\1\)) <= S1;
        S1 * (COM{1}(IRE;\1\)) <= ADD(S1;\0,0,0,0,0,1,0,0\);
        => (~COM{1}(IRE;\1\),COM{1}(IRE;\1\)) / (2,1).
   ENDSEQUENCE
   CONTROLRESET (RESET) / (1).
END.
```

```
1KING FAHD UNIV OF PETROLEUM & MINERALS, COLLEGE OF COMPUTER SCIENCE
 UNIVERSAL AHPL SIMULATOR OUTPUT       DATE: SAT 13 NOV., 1993       TIME:
12:16:17


       UNIVERSAL AHPL FUNCTIONAL LEVEL SIMULATOR OUTPUT:

            IRE
            | BRANCHPC
            | |  BRANCH
            | |  | IBUS
            | |  | | OBUS
            | |  | |  |  POPC
            | |  | |  |  |  PPC
 CLOCK #    | |  | |  |  |  |
       0    0 00 0 00 00 00 00
            **** ACTIVE STEPS IN CLOCK (   1) : SYST/STEP ****
                                                  1/   1
       1    1 04 0 01 00 00 00
            **** ACTIVE STEPS IN CLOCK (   2) : SYST/STEP ****
                                                  1/   2
       2    1 04 0 01 05 00 00
            **** ACTIVE STEPS IN CLOCK (   3) : SYST/STEP ****
                                                  1/   1
       3    0 04 0 01 05 00 00
            **** ACTIVE STEPS IN CLOCK (   4) : SYST/STEP ****
                                                  1/   2
       4    0 04 0 02 05 00 04
            **** ACTIVE STEPS IN CLOCK (   5) : SYST/STEP ****
                                                  1/   2
       5    1 04 0 02 05 00 04
            **** ACTIVE STEPS IN CLOCK (   6) : SYST/STEP ****
                                                  1/   1
       6    1 04 0 03 05 00 04
            **** ACTIVE STEPS IN CLOCK (   7) : SYST/STEP ****
                                                  1/   2
       7    0 04 0 03 07 04 08
            **** ACTIVE STEPS IN CLOCK (   8) : SYST/STEP ****
                                                  1/   2
       8    0 04 0 04 07 04 08
            **** ACTIVE STEPS IN CLOCK (   9) : SYST/STEP ****
                                                  1/   2
       9    1 04 0 04 07 04 08
            **** ACTIVE STEPS IN CLOCK (  10) : SYST/STEP ****
                                                  1/   1
      10    1 04 0 05 07 04 08
            **** ACTIVE STEPS IN CLOCK (  11) : SYST/STEP ****
                                                  1/   2
      11    0 04 1 05 09 08 0C
            **** ACTIVE STEPS IN CLOCK (  12) : SYST/STEP ****
                                                  1/   2
```

```
12    0 04 1 06 09 08 0C
      **** ACTIVE STEPS IN CLOCK (  13) : SYST/STEP ****
                                            1/   2
13    1 05 1 06 09 08 0C
      **** ACTIVE STEPS IN CLOCK (  14) : SYST/STEP ****
                                            1/   1
14    1 05 1 07 09 08 0C
      **** ACTIVE STEPS IN CLOCK (  15) : SYST/STEP ****
                                            1/   2
15    0 05 1 07 0B 0C 10
      **** ACTIVE STEPS IN CLOCK (  16) : SYST/STEP ****
                                            1/   2
16    0 06 1 08 0B 0C 10
      **** ACTIVE STEPS IN CLOCK (  17) : SYST/STEP ****
                                            1/   2
17    1 06 1 08 0B 0C 10
      **** ACTIVE STEPS IN CLOCK (  18) : SYST/STEP ****
                                            1/   1
18    1 06 1 01 0B 0C 10
      **** ACTIVE STEPS IN CLOCK (  19) : SYST/STEP ****
                                            1/   2
19    1 06 1 01 05 05 09
      **** ACTIVE STEPS IN CLOCK (  20) : SYST/STEP ****
                                            1/   1
20    1 07 1 01 05 05 09
      **** ACTIVE STEPS IN CLOCK (  21) : SYST/STEP ****
                                            1/   2
21    0 07 0 02 05 06 0A
      **** ACTIVE STEPS IN CLOCK (  22) : SYST/STEP ****
                                            1/   2
22    0 07 0 02 05 06 0A
      **** ACTIVE STEPS IN CLOCK (  23) : SYST/STEP ****
                                            1/   2
23    1 07 0 03 05 06 0A
      **** ACTIVE STEPS IN CLOCK (  24) : SYST/STEP ****
                                            1/   1
24    1 08 0 03 05 06 0A
      **** ACTIVE STEPS IN CLOCK (  25) : SYST/STEP ****
                                            1/   2
25    0 08 0 04 07 07 0B
      **** ACTIVE STEPS IN CLOCK (  26) : SYST/STEP ****
                                            1/   2
26    0 08 0 04 07 07 0B
      **** ACTIVE STEPS IN CLOCK (  27) : SYST/STEP ****
                                            1/   2
27    1 08 0 05 07 07 0B
      **** ACTIVE STEPS IN CLOCK (  28) : SYST/STEP ****
                                            1/   1
28    1 0A 0 05 07 07 0B
      **** ACTIVE STEPS IN CLOCK (  29) : SYST/STEP ****
                                            1/   2
```

```
29    0 0A 0 06 09 0B 0F
      **** ACTIVE STEPS IN CLOCK (  30) : SYST/STEP ****
                                            1/   2
30    0 0A 0 06 09 0B 0F
      **** ACTIVE STEPS IN CLOCK (  31) : SYST/STEP ****
                                            1/   2
31    1 0A 1 07 09 0B 0F
      **** ACTIVE STEPS IN CLOCK (  32) : SYST/STEP ****
                                            1/   1
32    1 04 1 07 09 0B 0F
      **** ACTIVE STEPS IN CLOCK (  33) : SYST/STEP ****
                                            1/   2
33    0 04 1 08 0B 0F 13
      **** ACTIVE STEPS IN CLOCK (  34) : SYST/STEP ****
                                            1/   2
34    0 04 1 08 0B 0F 13
      **** ACTIVE STEPS IN CLOCK (  35) : SYST/STEP ****
                                            1/   2
35    1 04 1 01 0B 0F 13
      **** ACTIVE STEPS IN CLOCK (  36) : SYST/STEP ****
                                            1/   1
36    1 05 1 01 0B 0F 13
      **** ACTIVE STEPS IN CLOCK (  37) : SYST/STEP ****
                                            1/   2
37    1 05 1 01 05 04 08
      **** ACTIVE STEPS IN CLOCK (  38) : SYST/STEP ****
                                            1/   1
38    1 05 1 02 05 04 08
      **** ACTIVE STEPS IN CLOCK (  39) : SYST/STEP ****
                                            1/   2
39    0 06 1 02 06 05 09
      **** ACTIVE STEPS IN CLOCK (  40) : SYST/STEP ****
                                            1/   2
40    0 06 1 03 06 05 09
      **** ACTIVE STEPS IN CLOCK (  41) : SYST/STEP ****
                                            1/   2
41    1 06 0 03 06 05 09
      **** ACTIVE STEPS IN CLOCK (  42) : SYST/STEP ****
                                            1/   1
42    1 06 0 04 06 05 09
      **** ACTIVE STEPS IN CLOCK (  43) : SYST/STEP ****
                                            1/   2
43    0 07 0 04 08 05 09
      **** ACTIVE STEPS IN CLOCK (  44) : SYST/STEP ****
                                            1/   2
44    0 07 0 05 08 05 09
      **** ACTIVE STEPS IN CLOCK (  45) : SYST/STEP ****
                                            1/   2
```

```
45    1 07 0 05 08 05 09
      **** ACTIVE STEPS IN CLOCK (  46) : SYST/STEP ****
                                            1/   2
46    1 07 0 06 08 05 09
      **** ACTIVE STEPS IN CLOCK (  47) : SYST/STEP ****
                                            1/   1

47    0 08 0 06 0A 09 0D
      **** ACTIVE STEPS IN CLOCK (  48) : SYST/STEP ****
                                            1/   2
48    0 08 0 07 0A 09 0D
      **** ACTIVE STEPS IN CLOCK (  49) : SYST/STEP ****
                                            1/   2
49    1 08 0 07 0A 09 0D
      **** ACTIVE STEPS IN CLOCK (  50) : SYST/STEP ****
                                            1/   1
50    1 08 0 08 0A 09 0D
      **** ACTIVE STEPS IN CLOCK (  51) : SYST/STEP ****
                                            1/   2

::::: PROGRAM REACHED THE CLOCKLIMIT. AHPL SIMULATION STOPS. :::::
```

Bibliography

[1] M. C. McFarland, A. C. Parker and R. Camposano "The High-Level Synthesis of Digital Systems," in proc. of the IEEE, vol. 78, No. 2, Feb. 1990

[2] R. A. Walker and R. Camposano "A Survey of High-Level Synthesis Systems," Kluwer Academic Publishers, 1991.

[3] T. D. Friedman and S. C. Yang, "Methods used in an Automatic Logic Design Generator (ALERT)," IEEE Trans. Computer, vol. C-18, 1969, pp. 593-614.

[4] T. D. Friedman and S. C. Yang, "Quality of Designs from an Automatic Logic Generator (ALERT)," Proc. of the 7th Design Automation Workshop, pp. 71-89, June 1970.

[5] D. E. Thomas and D. P. Siewiorek, "A Technology Relative Computer-Aided Design System: Abstract Representations, Transformations and Design Trade-offs," Proc. of the 14th DAC, June 1977.

[6] D. L. Springer and D. E. Thomas, "Exploiting the Special Structure of Conflict and Compatibility Graphs in High-Level Synthesis," Proc. of ICCAD'90, pp. 254-257, Nov. 1990.

[7] R. J. Cloutier and D. E. Thomas, "The Combination of Scheduling, Allocation and Mapping in a Single Algorithm," Proc. of the 27th DAC, pp. 71-76, June 1990.

[8] D. E. Thomas, E. D. Lagnese, R. A. Walker, J. A. Nestor, J. V. Rajan and R. L. Blackburn, "Algorithmic and Register-Transfer Level Synthesis: The System Architect's Workbench," Kluwer Academic Publishers, Boston, 1990.

[9] D. E. Thomas, C. Y. Hitchcock III, T. J. Kowalski, J. V. Rajan and R. A. Walker, "Methods of Automatic Data Path Synthesis," IEEE Computer, pp. 59-70, Dec. 1983.

[10] S. W. Director, A. C. Parker, D. P. Siework and D. E. Thomas, "A Design Methodology and Computer Aids for Digital VLSI Systems," IEEE Trans. on Circuits and Systems, pp. 634-635, July 1981.

[11] P. G. Paulin and J. P. Knight, "Force-Directed Scheduling for the Behavioral Synthesis of ASIC's," IEEE Trans. on CAD, pp. 661-679, June 1989.

[12] P. G. Paulin, J. P. Knight and E. F. Girczyc, "HAL: A Multi-Paradigm Approach to Automatic Data Path Synthesis," Proc. of the 23rd DAC, pp. 263-270, June 1986.

[13] P. G. Paulin and J. P. Knight, "Algorithms for High-Level Synthesis," IEEE Design and Test, pp. 18-31, Dec. 1989.

[14] P. G. Paulin and J. P. Knight, "Force-Directed Scheduling in Automatic Data Path Synthesis," Proc. of the 24th DAC, pp. 195-202, June 1987.

[15] P. G. Paulin and J. P. Knight, "Scheduling and Binding Algorithms for High-Level Synthesis," Proc. of the 26th DAC, pp. 1-6, June 1989.

[16] Krekelberg, E. Sharagowitz, G. E. Sobleman and L. S. Lin, "Automated Layout Synthesis in the YASC Silicon Compiler," Proc. of the 23rd DAC, pp. 447-453, 1986.

[17] P. Marwedel, "The MIMOLA Design System: Tools for the Design of Digital Processors," Proc. of the 21st DAC, pp. 587-593, June 1984.

[18] H. Trickey, "Flamel: A High-Level Hardware Compiler," IEEE Trans. on CAD, pp. 259-269, Mar. 1987.

[19] T. Tanaka, T. Kobayashi and O. Karatsu, "Harp: FORTRAN to Silicon," IEEE Trans. on CAD, vol. 8, No. 6, June 1989.

[20] H. Kramer and W. Rosenstiel, "System Synthesis using Behavioral Descriptions," Proc. of EDAC'90, pp. 277-282, Mar. 1990.

[21] R. Camposano, "Path-Based Scheduling for Synthesis," IEEE Trans. on CAD, vol. 10, No. 1 January 1991.

[22] R. Camposano and R. Bergamaschi, "Synthesis Using Path-Based Scheduling: Algorithms and Exercises," in Proc. of the 27th Design Automation Conference,ACM/IEEE, 1990, pp. 450-455.

[23] K. O'Brien, M. Rahmouni and A. A. Jerraya, "A VHDL-Based Scheduling Algorithm For Control-Flow Dominated Circuits," Technical report, Institut IMAG, Gernoble, France, 1992.

[24] Z. Peng, "Synthesis of VLSI Systems with the CAMAD Design Aid," in Proc. of the 23rd Design Automation Conference, New York, NY, ACM/IEEE June 1986, pp. 278-284.

[25] R. K. Brayton, R. Camposano, G. DeMichelo, R. Otten and J. vanEijindhoven, "The Yorktown Silicon Compiler," in *Silicon Compilation*, D. D. Gajski, Ed. Reading, MA: Addison-Wesley, 1988, pp. 204-311.

[26] A. C. Parker, J. Pizarro and M. Mlinar, "MAHA: A Program for Datapath Synthesis," in Proc. of the 23rd Design Automayion Conference, New York, NY, ACM/IEEE, June 1988, pp. 461-466.

[27] A. Hashimoto and M. Elmasry, "Automated Synthesis of a Multi-Bus Architecture for DSP," ICCAD-88, pp. 44-47, Nov. 1988.

[28] F. J. Kurdahi and A. C. Parker, "REAL: A Proogram for REgister ALlocation," 24th ACM/IEEE Design Automation Conference, pp. 210-215, Jun. 1987.

[29] B. M. Bangrle, "Splicer: A Heuristic Approach to Connectivity Binding," 25th ACM/IEEE Design Automation Conference, pp. 536-541, Jun. 1988.

[30] V. K. Raj, "Another Automated Data Path Designer," ICCAD-86, pp. 214-217, Nov. 1986.

[31] C. J. Tseng and D. P. Siewiorek, "Automated Synthesis of Data Path in Digital Systems," IEEE Trans. on CAD of ICAS, vol. CAD-5, no. 3, pp. 379-395, Jul. 1986.

[32] C. Y. Huang *et al.* "Data Path Allocation Based on Bipartite Weighted Matching," 27th ACM/IEEE Design Automation Conference, pp. 499-504, Jun. 1990.

[33] N. Dutt and C. Ramachandran, "Benchmarks for the 1992 High Level Synthesis Workshop," Technical Report #92-107, University of California, Irvine, Oct. 30, 1992.

[34] M. C. McFarland, "The Value Trace: A Data Base for Automated Digital Design," Dept. of Electrical Engineering, Carnegie-Mellon University, Dec. 1978.

[35] S. M. Sait, "Integrating UAHPL-DA System with VLSI Design Tools to Support VLSI DA Courses," IEEE Trans. on EDUCATION, vol. 35, no. 4, pp. 321-331, Nov. 1992.

[36] S. M. Sait and M.S.K. Tanvir, "VLSI Layout Generation of a programmable CRC Chip," IEEE Trans. on Consumer Electronics, vol. 39, no. 4, pp. 911-916, Nov. 1993.