Order Number 1360408

Hardware specific optimization on RTL descriptions

Al-Mulhem, Abdulaziz Sultan, M.S.

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

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

Hardware Specific Optimization on RTL Descriptions

BY

Abdulaziz Sultan Al-Mulhem

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

June 1994

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS
DHAHRAN 31261, SAUDI ARABIA

COLLEGE OF GRADUATE STUDIES

This thesis, written by

ABDULAZIZ SULTAN AL-MULHEM

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:

7th June 94
Sadiq Sait M.

Dr. Sadiq M. Sait (Chairman)

Dr. Mohammad S.T. Benten (Co – Chairman)

7/6/1994

Dr. Habib Youssef (Member)

Department Chairman

Dean, College of Graduate Studies

7/6/94

Date

كلية الدراسات العليا
COLLEGE OF GRADUATE STUDIES

To the memories of my father

and mother whose support and prayers

led to this achievement.

To my dear wife.

Acknowledgments

All praise be to Allah for his limitless help and guidance. Peace and blessings of Allah be upon His prophet Muhammad.

Acknowledgement is due to King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia, for the generous help and support for this research.

I would like to express my profound and heart-felt gratitude and appreciation to my advisor, Dr. Sadiq M. Sait, Associate Professor of Computer Engineering, for his guidance and patience throughout this thesis. His continuous support and encouragement can never be forgotten.

I would also like to thank co-chairman Dr. Mohammad S.T. Benten, Dean CCSE and Associate Professor of Computer Engineering, whose continuous encouragement can never be forgotten, and Dr. Habib Yousef, Assistant Professor of Computer Engineering for his consistent support 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 Dr. Samir Abdul-Jauwad.

Finally, special thanks must be given to my family and my wife, for their encouragement and moral support.

Contents

List of Figures

List of Tables

Abstract

Name: ABDULAZIZ SULTAN AL-MULHEM

Title: Hardware Specific Optimization on RTL Descriptions

Major Field: Computer Engineering

Date of Degree: June 1994

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

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 (IFs).

Existing HLS systems incorporate IFs that are used in programming language compilers. In this work we present the stack intermediate form. Stacks are used in language interpreters. We refer to this IF as **Blocked Reverse Polish Notation** *(BRPN). It is the first time in the field of HLS to use stack representation. The behavioral specification of a digital system is first interpreted into BRPN. Due to the complication involved, BRPN is translated into a graph based IF. We refer to this IF as the* **Canonical RTL** *(CRTL).*

Due to the ample code generated from BRPN, optimization techniques are exercised diligently on CRTL code. Unlike the traditional compilers optimization techniques used in HLS systems, hardware specific optimization techniques are applied. These techniques utilize hardware specific traits.

The major contributions of this work are the introduction of the stack IF in HLS and the exploitation of the hardware specific features in optimizing RTL descriptions.

Master of Science Degree
King Fahd University of Petroleum and Minerals
Dhahran, Saudi Arabia
1994

خلاصة الرسالة

اسم الطالـــب : عبدالعزيز سلطان الملحم

عنوان الرسالة : الإختزالات الخاصة بالتصميمات التركيبية في مستويات الوصف التركيبي .

التخصـــص : هندسة الحاسب الآلي

تاريخ الشهادة : يونية ١٩٩٤ م

التصميم العام هو عملية تحويل المواصفات العامة لسلوكيات الدوائر المتكاملة الى التصميم التركيبي لتلك الدوائر . وتنتج هذة العملية شبكة من العناصر المتصلة على مستوى السجلات (مستوى الوصف التركيبي) ومن أمثلة هذه العناصر : السجلات ووحدات المعالجات المنطقية والحسابية ووحدات إختيار المدخلات .

ولما كانت عملية التصميم العام عملية معقدة . فإنة يتم تقسيم هذة العملية الى عدد من المراحل وتهدف كل مرحلة الى إنجاز جزء من التصميم . والهدف من ذلك هو نقل مواصفات التصميم من مستوى الوصف العام الى المواصفات المطلوبة فى مستوى السجلات (مستوى الوصف التركيبي) مروراً بعدد من الأشكال المتوسطة .

تَستخدمُ أنظمة التصميم العام الحالية بنفس الاشكال المتوسطة المستخدمة فى المولفات (المنسقات) البرمجية . في هذا البحث تم عرض الشكل المتوسط الكومي . ان هذة المكومات كثيراً ماتُستخدمُ فى المترجمات اللغوية ، ولكنها المرة الأولى التى يتم فيها استخدام المكومات في عمليات التصميم العام . حيث يتم ترجمة المواصفات العامة لسلوكيات الدوائر المتكاملة الى الشكل المتوسط الكومي . ومن ثم يتم تحويلها الى شكل متوسط بسيط هو ماندعوة «مستوى الوصف التركيبي البسيط» وذلك لما يحويه الوصف الكومي من تعقيدات .

ونظراً لحجم الوصف التركيبي الناتج فقد تم تطبيق طرق إختزالية تختلف عن الطرق الإختزالية المستخدمة في المنسقات البرمجية حيث تَستغل هذه الطرق الصفات الخاصة والموجودة في التصميمات التركيبية للدوائر .

بالإضافة الى ما تقدم فإن أهم المساهمات العلمية لهذا البحث هي إستخدام الأشكال المتوسطة الكومية فى التصميم العام وبناء طرق إختزالية تعتمد فى عملها على الصفات الخاصة للتصميمات التركيبية .

درجة الماجستير في العلوم
جامعة الملك فهد للبترول والمعادن
الظهران ، المملكة العربية السعودية
يونية ١٩٩٤م

Chapter 1

Introduction

Digital circuits can be described at different levels of abstraction: *behavioral*, *structural*, or *physical*. Digital systems described at the physical level are ready for fabrication. Structural descriptions are at the next higher level of the hierarchy. In the structural domain, the components of the digital system (gates, flip-flops, registers, functional units...etc), and their interconnections are described as a netlist. Physical design has been a topic of research for over three decades and several efficient physical synthesis CAD tools are available that translate structural descriptions to layouts [Sai92].

Digital systems can also be described in the behavioral domain. The concern at this level, is on how the digital system interacts with its environment. Usually, the digital system structure is hidden. The actions or behavior of the system outputs in

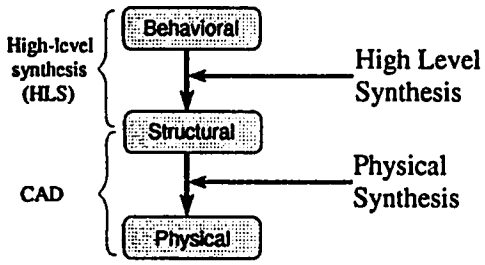


Figure 1.1: Different levels of abstraction.

response to different inputs is the only required information to describe the systems functionality. Basically, this is the merit of describing digital systems behaviorally. Figure 1.1 shows the different levels of abstraction. The scope of this work is to synthesize structural level descriptions from behavioral models. This process is called High-level Synthesis (HLS).

1.1 High-level Synthesis

High-level Synthesis (HLS) is the process of translating an input behavioral description of a digital system into a structural description. Behavioral specifications of digital systems can be described using high level programming languages (HLLs), Hardware Description Languages (HDL), Algorithmic pseudo codes, etc [TKK89, Tri87]. Among the different ways available for describing digital circuits, HLL provides the most abstract level. In addition, simplicity and conciseness in the description of the circuit functions characterize modeling in HLL.

The synthesis process requires different tasks to be accomplished. The main issue here is to preserve the functionality of the digital system while generating its hardware. This process is complex and consists of analysis, transformation, and manipulation. Due to this complexity, the process is divided into tasks. In each task, a partial refinement towards the target structure is introduced. The behavioral description is first translated to an intermediate form (IF) from which other IFs can be generated if necessary. This structure should be functionally equivalent to the system behavioral model. Most commonly used IFs are graph-based such as data flow graphs (DAG) and control flow graphs (CFG) [PG87].

1.2 Objectives

The objective of this thesis is to develop a HLS system that accepts an input description in a HLL and generates the appropriate structural description. The behavioral language selected is a subset of the C programming language; and we will refer to as C-like. The structural description is in A Hardware Programming Language (AHPL). This work has two distinctive features:

1. A stack data structure is used as the primary intermediate form. To our knowledge this is the first time that techniques and data structures used by programming language interpreters are employed in the HLS of hardware.

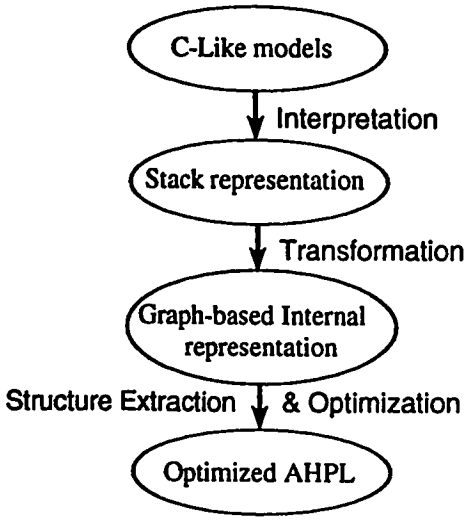


Figure 1.2: Main tasks in the HLS system.

2. The input behavioral specification is interpreted as a hardware description. Thus, besides the usual software optimization techniques applied during the synthesis tasks, hardware specific optimization is also performed.

Figure 1.2 gives the main steps followed by the HLS system. First, the behavioral description of the digital system modeled in C-like language is interpreted into a stack representation. Second, a graph-based IF is generated from this stack representation. The structure is then extracted from the graph-based model. Finally, the extracted structure is optimized.

Stacks are commonly used in interpreters. They are efficient data structures in evaluating arithmetic expressions. In stack machines, expressions are simplified into a group of basic operations. These basic operations are executed sequentially.

The advantages of using a stack representation as an IF are:

1. Simple, yet powerful data structure,

2. Easy to generate and manipulate, and

3. Flexibility of grouping stack operations.

In this stack IF representation, we will use the notion of *blocking*. By blocking we mean that a set of consecutive stack operations are grouped in a labeled block. This blocking is useful in interpreting behavioral control structures into a stack representation.

1.3 Organization

Details on the implementation of the various tasks described above are covered in six chapters. Following the introduction, Chapter 2 presents some background on High-level Synthesis (HLS), a brief review of Intermediate Forms (IFs) and some literature review on scheduling and target design hardware. In Chapter 3, the C-like models of digital circuits are translated into a stack representation. Complex control structures such as *if-then-else* and *case* are also converted into the stack representation. In Chapter 4, we describe the procedure used to transform a primary intermediate form (stack) into another internal representation more suitable for the

remaining tasks of HLS. This secondary IF is a Control-Data Flow Graph (CDFG) where each node holds a stack operation and edges represents the flow of control between the nodes. The generated IF is a Canonical Register Transfer Language (CRTL). The algorithm to do this transformation and its complexity are also discussed in Chapter 4. From CRTL the target structure of the digital system in a known RTL description is extracted. The target RTL descriptions in this work is *A Hardware Programming Language* (AHPL). Hardware specific optimization is discussed in Chapter 5. Initially software optimization techniques are used to eliminate redundant code and unconditional branching. The algorithms to perform software optimization and their complexities are described. In the remaining part of the chapter, hardware specific optimization techniques are presented. Switch transformation and loop transformation are the two techniques discussed. The algorithms used to implement these optimization techniques are also presented and explained. The complexity of these algorithms is analyzed. Different examples on applying these techniques are illustrated.

Analysis of the developed HLS system and comparison of obtained results with other systems using some benchmark test cases and other circuits are given in Chapter 6. Finally, conclusions and future work are found in Chapter 7.

Chapter 2

Literature review

Intensive effort has been put on the synthesis of digital systems. The development of methodologies for automating the synthesis process has been a research topic for the last 20 years [TS86].

There are several tasks in the automatic synthesis of digital systems [Par84]: definition of the circuit function, translation into an intermediate form, operation scheduling, hardware module allocation, and data path synthesis. High level programming languages (HLL) which are behavioral descriptors, are used to define the functionality of digital systems. Structural descriptions are hard to generate from HLL. Therefore, intermediate forms (IFs) are used to translate the input specification from behavioral level down to structural level. Transformations and optimization techniques are applied on these IFs. Scheduling of operations to control steps

and allocation of operators to functional units make the next tasks. Finally, the data path is synthesized corresponding to the required digital system.

2.1 Behavioral description

An important issue is the choice of a behavioral description language. The main advantages of designing systems using a behavioral level description [CR89] are: reduction in design time, correctness, semantic check, and availability of IC technology to non-experts. Moreover, as much parallelism as possible can be identified from behavioral descriptions [KM91]. Several high level synthesis (HLS) systems are reported using a wide variety of input specification languages which range from PASCAL-like to ISPS [CR89, MPC90]. ISPS is a hardware description language which stands for Instruction Set Processor Specification. The proposed languages are classified into three categories: programming languages, special programming languages, and hardware description languages. In the first category, existing programming languages are used to describe the behavior of digital systems. Using programming languages in HLS is characterized by having a huge implementation space as a single behavioral description can have many structural descriptions [Par84]. On the other hand, hardware description languages are among the first languages used in designing digital systems [MPC90]. Digital system specification language (DSL) [CR89], structured function description language (SFL) [NON91], HardwareC [KM91], and

A Hardware Programming Language (AHPL) [SBK93] are some proposed hardware description languages that are in the third category.

Defining a special programming language rather than using an existing programming language offers the advantage that it can be designed according to the special needs of the application [CR89]. One reported language is Architectural Behavioral Description Language (ABDL) which is a C-like language that falls in this category.

PUBSS is a HLS system developed at Princeton University. The intended hardware is described as a set of communicating behavior finite state machines (BFSMs) [WTL91]. Each BFSM is an automaton whose inputs and outputs are only partially scheduled [WTL91]. The BFSM network model combines scheduling information with a state-based description of control [WTL91].

2.2 Intermediate forms

The behavioral description of digital systems is compiled into an intermediate form [TS86]. This intermediate form (IF) abstracts the input behavioral description specification. Two generally used types of IFs [Par84] are: parse trees and graphs. The most common IFs are graph-based models [PG87, TS86]. The behavioral description is compiled into a graph which is broken into directed acyclic graphs (DAGs). These DAGs are attractive IFs because many compiler optimization techniques can be ap-

plied on them [PG87]. Data flow and control flow graphs are the most widely used types of DAGs.

One reported HLS called HARP translates Fortran programs into data flow graphs [TS86]. In DFG, the data dependencies of the input specification is translated into operation ordering. High-level IBM synthesis System (HIS) is another HLS that uses control flow graphs (CFG) as IF where nodes holds operations and edges presents the precedence relation [CBH+91]. In CFG, the execution dependencies among the statements of the input specification is translated into states ordering. In another HLS called STAR [TH86], relation networks are used. A relation network is a weighted graph where nodes represent objects and edges represent correlation between objects [TH86].

Another type of DAGs used as an IF, is control-data flow graph (CDFG). CDFG which is a graph representation that captures both data flow and control dependencies among the operators, is used in THEDA [PK89b]. Flamel is another HLS that translates Pascal programs into *dacon* (the data flow/control flow). *Dacon* is a block graph whose nodes represent blocks and edges represent the "transfer-control-to" relation [Tri87]. Value trace (VT) is an intermediate form used in the *facet* HLS [TS86]. In VT, DFG and CFG are integrated into one structure.

A tree-structured control flow graph (tCFG) which is an implementation independent description of hardware representation in tree structured diagrams is used

as an IF in Cyber [Wak91]. tCFG, which results from compiling Behavioral Description Language (BDL) [Wak91], is transformed into CDFG.

Hercules/Hebe [KM91] generates an implementation independent description of behavior in a graph-based representation called Sequence Intermediate Form (SIF). SIF which is modeled as a polar DAG, is a sequencing graph that preserves the partial order among a set of operations [KM91].

2.3 Tasks in HLS

The two major tasks in HLS are scheduling and allocation. In scheduling, operations are assigned to control steps which are fundamental sequencing units in synchronous systems [Par84]. While assigning operators to hardware is called allocation. Several scheduling and allocation approaches have been reported [CR89]: direct compilation, graph transformation, data flow analysis, mixed-integer linear programming, clique partitioning, and force directed.

2.3.1 Scheduling

In scheduling, the propagation delay of every operation is determined [PK89a]. Then each operation is assigned to a specific control step [PK89a, Par84]. The target in scheduling is to minimize the time required for program completion [Par84].

The algorithmic approaches used in scheduling fall in one of two basic classes: transformational and iterative/constructive algorithms. An iterative/constructive algorithm schedules one operation at a time until all operations are scheduled. Algorithms of the other class requires an initial schedule. This initial schedule is usually either maximally serial or maximally parallel. Then the function of the algorithm is to transform the schedule from serial to parallel or vice versa. One reported scheduling algorithm which is graph-based, is used in MAHA. MAHA schedules the critical path. Then, the remaining operations are scheduled in order of increasing freedom [Wak91]. Another scheme is Force Directed scheduling (FDS) [PK89a]. In FDS, the probability distribution of operations is determined to balance the required hardware amount in each control step [Wak91]. A global time constraint is specified and the algorithm attempts to minimize the resources required to meet the stated constraint [PK89a]. FDS is used in HAL [Pau91, Wak91] HLS system. Critical path scheduling and FDS exploit potential parallelism but they do not deal with branches and loops [Wak91].

List scheduling (LS) is another scheduling scheme where hardware constraints are specified. The LS algorithm attempts to minimize the total execution time by using a local priority function to defer operations when resource conflicts occur [Pau91, PK89a]. A combination of LS and FDS is the *force-directed list scheduling* (FDLS) [PK89a]. The goal in FDLS is to reduce the resources by balancing the concurrency of the operations assigned to hardware without increasing the total

execution time [PK89a].

In HIS, As Fast As Possible (AFAP) scheduling scheme is applied on the CFG [CBH+91]. AFAP emphasizes conditional branching rather than potential parallelism like in LS or FDS [CBH+91]. Cyber uses As Soon As Possible scheduling algorithm [Wak91].

ADAM system [WTL91] synthesizes circuits interfaces by transforming its internal representation into finite state machines (FSM). The scheduling scheme used in ADAM is state scheduling which is used to minimize the number of states in a controller [WTL91]. State scheduling uses a process called unzapping. The states of a basic block are unzapped by creating equivalent states. A basic block is a linear sequence of operations having one entry point and one exit point. Then, the state scheduling takes advantage of state equivalence to create larger subproblems which allow more states to be combined [WTL91].

An algorithm called zone scheduling [HL91] is used in THEDA. It is a heuristic method for solving the resource constraint scheduling of a large basic block; and several control steps (forming a zone) are solved at a time [HL91].

Unified System Construction (USC) uses 3D scheduling [PKPW91]. 3D scheduling is a module allocation and assignment scheduling technique [PKPW91]. During the high level synthesis process, 3D scheduling incorporates interconnection delays obtained from floorplanning [PKPW91].

O'Brien *et al* [ORJ92] proposed the Dynamic Loop Scheduling (DLS) algorithm. DLS is used for the treatment of control-flow dominated descriptions written in VHDL [ORJ92]. Path-based scheduling (PBS) is another scheduling scheme [Cam91]. This scheme uses directed graphs that represent the precedence relation among operators. In PBS, parallelism is emphasized on conditional branching; that is to schedule the paths of a condition branch in the minimum number of control steps [Cam91]. A heuristic scheduling scheme that is based on PBS is the loop-based scheduling (LBS) [AS94]. In LBS, the input specification is mapped into directed graphs. Loop entrance nodes are scheduled in different control steps [AS94].

2.3.2 Allocation

Allocation consists of assigning the operations to hardware. The allocation goal is to minimize the number of functional units, total storage, and the total connection path [Par84]. The main allocation techniques in HLS are [CBH+91]: heuristic or greedy techniques, linear programming formulation, and clique covering or coloring based allocation.

The allocation scheme in *Facet* [TS86] is to transform the minimization of the number of storage elements, data operators, and interconnection units problems into clique partitioning problems. The clique partitioning technique is used to bind variables to the minimum number of registers, and operators to the minimum number

of functional units.

In Mimola [TS86], a minimum implementation/cost subject to hardware availability is reached by means of statistical analysis. A similar approach is followed in HARP. HARP uses a First Come First Served (FCFS) strategy in allocating functional units (FUs) [TKK89]. The merging criteria of FUs is determined by calculating the minimum mutual correlation between FUs.

Other allocation techniques are applied. In HIS, a modified data-flow analysis technique which is used in compiler construction is adopted [NON91]. However, HAL uses FDLS in allocation [Pau91]. Finally, Devadas and Newton [DN89] proposed a simulated-annealing based algorithm for hardware allocation.

There is no consensus as to which task to perform first, scheduling or allocation [KM92]. Most of the HLS systems perform scheduling first [KM92]. While Caddy/DSL is reported to perform allocation first [KM92]. MAHA and HAL combine scheduling and allocation [KM92].

2.4 Target architecture

The final step in HLS is data path synthesis. The target in data path synthesis is to produce Register Transfer level hardware designs (structural description) [DN89]. Structural description specifies a set of components and their interconnection. Each

component is retrieved from a cell library. The design is finished once components are placed and routed. Designers have to do evaluation and modification to meet constraints [PG87]. Usually this is done with available Design Automation (DA) analysis tools.

Different target architectures are followed. In HARP, the data path is synthesized in RTL like description. Then, a micro-programmed controller is loaded with the data path control information.

EXPL [TS86] uses the DEC PDP-11 Register Transfer Modules (RTM) as a module set for the implementation of the design. In Camposano *et al*, the synthesized structure is given in STRUDEL (STRUcture DEscription Language) which consists of hierarchical netlists [CR89].

In STAR, data path construction (DPC) and data path refinement (DPR) are the phases where the physical design is generated from the relation networks as defined earlier [TH86].

HIS synthesizes synchronous digital systems. The output design is an FSM describing the control and a netlist for the data path [CBH+91]. In Hercules/Hebe, a logic-level implementation consisting of data path and control is described in Structural/Logic Intermediate Form (SLIF) [KM91]. Finally the logic design in PARTHENON is described in netlists [NON91].

2.5 Conclusion

From this literature review, it can be concluded that most of the HLS systems use graph based IFs. Moreover, the optimization techniques followed are software oriented. Basically they manipulate the specification as a software program rather than a description of a hardware system. Therefore, The optimization carried is similar to that used in compiler design. However, as will be demonstrated in a later chapter, hardware has some distinctive features which allow us to perform hardware specific optimization techniques, leading to more efficient/optimized RTL descriptions.

In the next chapter, we describe the intermediate form used by our HLS system.

Chapter 3

The Primary Intermediate Form

Automating the synthesis of digital systems as described earlier requires the compilation of behavioral descriptions into structural descriptions such as register transfer language (RTL). This compilation process is implemented as a sequence of translation steps. At each step, an intermediate form (IF) that preserves the original functionality is generated. IFs are internal representation that are introduced for the purpose of ease of manipulation as required by the various tasks of HLS. Different intermediate forms have been used. From the existing literature, most common IFs are Graph-based representations.

In this chapter a new IF in the context of HLS will be presented. Translation to this proposed IF will also be presented. This IF is based on stack internal representation. Stack representation is commonly used in interpreters. Moreover,

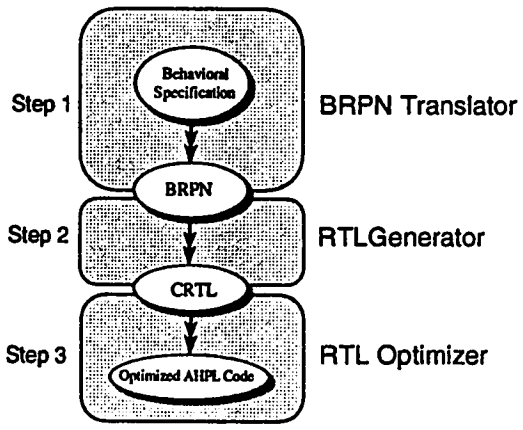


Figure 3.1: Major tasks in the new HLS system.

stacks are very efficient in evaluating arithmetic expressions, especially when used in conjunction with Reverse Polish Notation (RPN).

From this stack IF, a Canonical Register Transfer Language (CRTL) description is generated. CRTL is a control and data flow graph structure (CDFG). The CRTL description is further optimized and translated into the target RTL. Figure 3.1 gives an overview of the main tasks in the synthesis system. Step 1 in Figure 3.1 is the subject of this chapter. While, Step 2 is presented in the next chapter. Step 3 will be described in Chapter 5.

3.1 Blocked Reverse Polish Notation (BRPN)

To evaluate mathematical expressions on stack machines, prefix notation is the most appropriate ordering of the expression operators and operands. This prefix notation

is known as Reverse Polish Notation (RPN).

Usually, RPN is used for a single expression. In order to extend the scope of RPN, other control operations can be added to RPN. This adds more features to the mathematical expressions. Sub-expressions can be executed and transfer of control between them can also be achieved. Therefore, each sub-expression is converted into RPN separately. The control is then added to the RPN code to link up the different sub-expressions.

The mathematical expressions (arithmetic and logical) in this context correspond to a behavioral description of some digital system written in a HLL. To establish the connection between a HLL description and its corresponding RPN code, consider the stack operations shown in Figure 3.2. These operations contain data (Dopr) and control (Copr) operations.

The translation of C-like descriptions to stack notation is a language to language translation. The mapping of C-like to stack notation is illustrated with the following example. Consider the following HLL description of a *While* construct:

$i = 1;$
while $(i \leq k)$ do
    ⋮
    body
    ⋮
    $i = i + 1$
end-while.

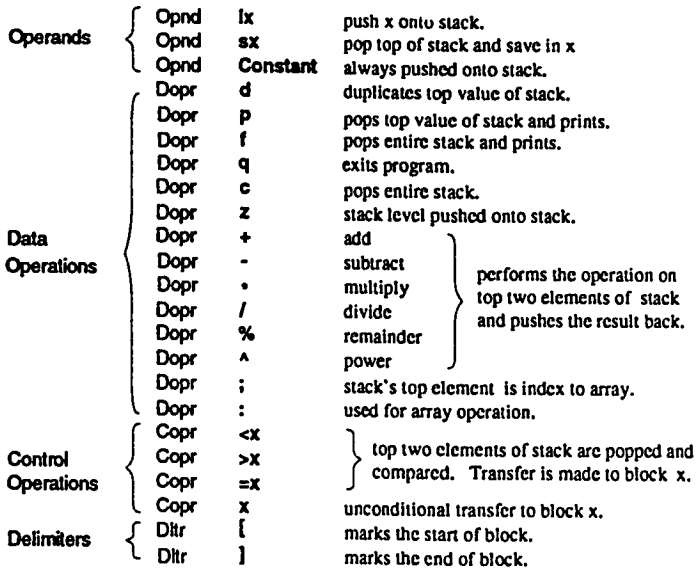


Figure 3.2: Some stack operations and operands used in BRPN.

The first line in the description is an assignment. This assignment is translated into pushing the constant '1' onto the stack followed by a pop operation. Then the popped value is loaded into register i. The corresponding stack representation is '1si'. In order to execute the loop, the condition 'i' is evaluated. The corresponding stack representation is as follows: the first operand is pushed onto the stack 'li', then the second operand is pushed 'lk'. Now both operands are on top of the stack. These operands are popped and the condition expression is formed. The evaluation of this condition will determine whether to execute the body of the loop or not. The stack representation corresponding to this is '$lilk > 0$', where '0' is the address of the code segment containing the body. The mapping of the rest of the code into stack representation produces the following stack code:

$1si$
$lilk$ >0
begin *(code segment 0)*
$li1 + si$
$lilk > 0$
end *(code segment 0)* .

The first line in the above code is an assignment instruction where register i is assigned the value 1. The second line is to check the starting of the loop, if the condition ($k > i$) is satisfied a branch to *code segment 0* takes place. In *code segment 0*, the first line is the adjusting step and the second line is the continuity step. The loop will be executed as long as the condition in the continuity step is true.

We define a block to be a sequence of stack operations with the restriction that transfer of control from other blocks can only take place at the first statement of the block. *Blocks* are delimited with square brackets and labeled. Each block label is prefixed with the capital letter *'S'*. Thus the above stack representation is as follows:

$1si$
$lilk > 0$
$[li1 + sililk > 0]S0.$

Because of this blocking in the stack representation, the RPN code generated by interpreting HLL will be called *blocked RPN* (BRPN) to differentiate it from the traditional way of expressing stack operations as a sequence of statements.

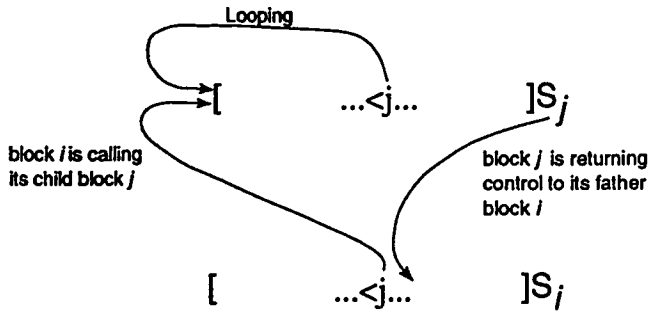


Figure 3.3: Father-children relation in BRPN.

In BRPN, the relation between the blocks is a *father-children* relationship. That is, during the execution of a set of stack instructions in a block, control may transfer to another block (calling block is termed as *father* and the called block as *child*). After a successful execution of the *child* block, control returns to the *father*. For the above example, the body of the *while* loop can be interpreted in the same block $S0$. Another way is to interpret the body in another block $S1$ where $S1$ is called from $S0$ every time the loop condition is evaluated true. This *father-child* relationship is illustrated in Figure 3.3. Recall that as a rule, the blocking strategy used here satisfy the following condition that: *control can only transfer to the beginning of a block.*

The translation of HLL descriptions to BRPN is a language to language translation. The grammar of the input behavioral language is given in Appendix A.

In the next section, the translation of different HLL constructs to BRPN will be shown. In addition, the translation of logical operators such as *and* and *or* to

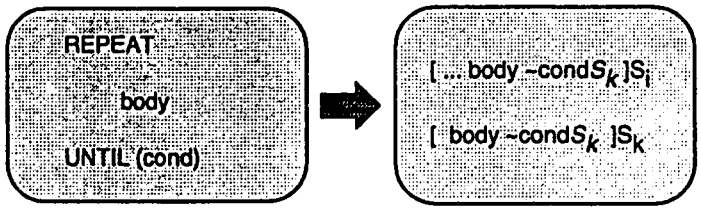


Figure 3.4: BRPN translation of *Repeat-Until* control statement.

BRPN will also be illustrated. Then a comparison between BRPN and graph-based representations like those described in the literature, is conducted in Section 3.3

3.2 Translating different HLL constructs into BRPN

In the previous section, the conversion or translation process from HLL descriptions into BRPN was shown. The *While-do* construct was used as an example. Other control constructs such as *Repeat-Until*, *If-Then-Else*, *Case*, ...etc can also be translated into their corresponding BRPN. This section shows the different HLL control structures with their corresponding BRPN.

Repeat-Until: This construct differs from *While-do* in two basic characteristics: the condition is evaluated at the end, and the loop is executed as long as the condition is false. The translation of this construct is shown in Figure 3.4

Here the body is executed before evaluating the condition. Therefore, the body is translated into stack operations as in block 'S_i' (Figure 3.4). Then, the condition

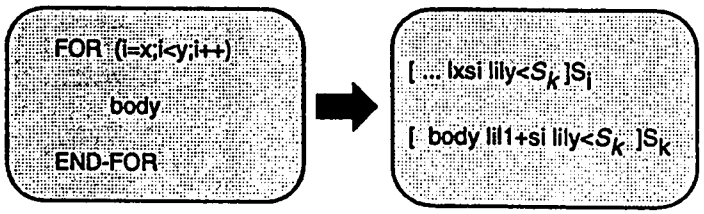


Figure 3.5: BRPN translation of *For* statement.

is translated to ($\sim$ *cond*S_k) in block 'S_i'. The stack operation ($\sim$ *cond*S_k) means that the control is transferred to the beginning of block 'S_k' if the condition (*cond*) is false. In block 'S_k', the body is executed as long as the condition (*cond*) is false.

For: Unlike *While-do* and *Repeat-Until*, the *For* control construct is used when the number of iterations is known. Counters are used to control the execution instead of conditional expressions. The *For* construct and its corresponding BRPN are shown in Figure 3.5.

The initialization of the loop counter is translated into stack operations as in block 'S_i' (Figure 3.5). The counter is checked whether its limit value is reached. If this is not the case, the control is transferred to block 'S_k'. The body is executed and the loop counter is advanced by the step value. Thereafter, the body is executed as long as the loop counter is within the range.

If-Then-Else: In this construct, the code execution is controlled by a condition. The *If-Then-Else* control construct and its corresponding stack representation are illustrated in Figure 3.6. The graphical representation of the resulting BRPN is

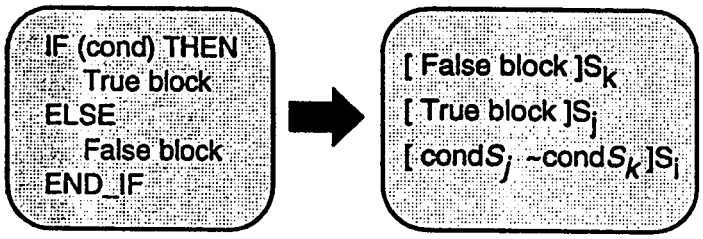


Figure 3.6: BRPN translation of *If-Then-Else* statement.

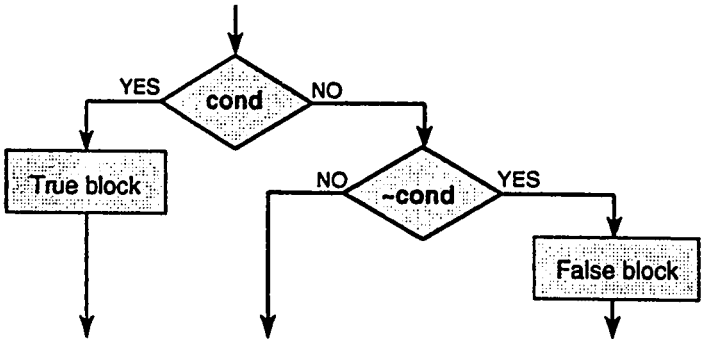


Figure 3.7: Graphical representation of BRPN *If-Then-Else* code.

illustrated in Figure 3.7.

Case: These statements are multi-branch control statements. This means that they are viewed as multiple *if-then-else* statements. The *Case* statement and its corresponding BRPN are illustrated in Figure 3.8. The BRPN generated is graphically illustrated in Figure 3.9.

Logical operators: These operators are widely used in high-level programming languages. *AND* and *OR* are among the widely used logical operators. The corresponding translation of both is shown in Figure 3.10(a) and (b) respectively. Statements

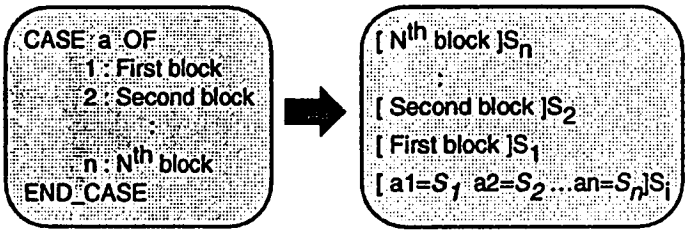


Figure 3.8: BRPN translation of *Case* statement.

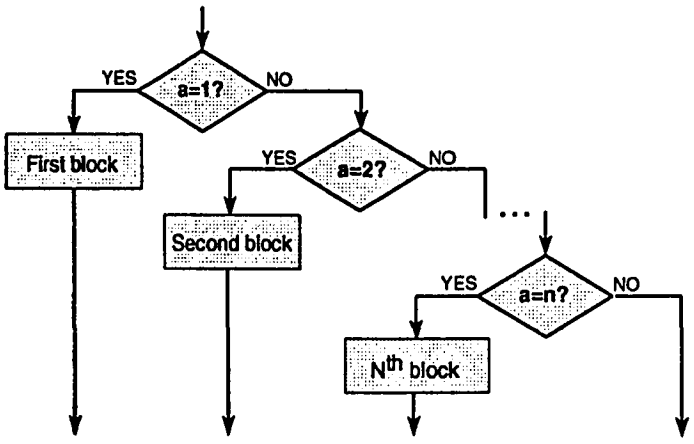


Figure 3.9: Graphical representation of BRPN *Case* code.

with multiple logical operators can also be translated to BRPN as illustrated in Figure 3.10(c). The *If-Then-Else* construct with logical operators can also be translated into BRPN. Figure 3.11(a) shows the *If-Then-Else* construct including the *AND* operator and its corresponding BRPN. While the same construct with the *OR* operator is illustrated in Figure 3.11(b).

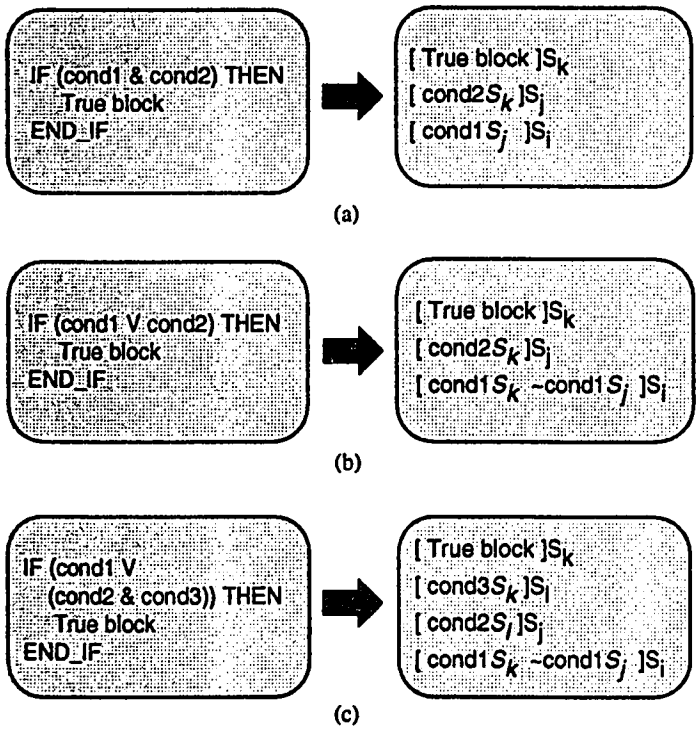


Figure 3.10: BRPN translation of different logical operators.

3.3 BRPN versus graph-based representation

The representation in BRPN is different from graph-based representations such as DFG and CFG. In this section, we will discus the key differences.

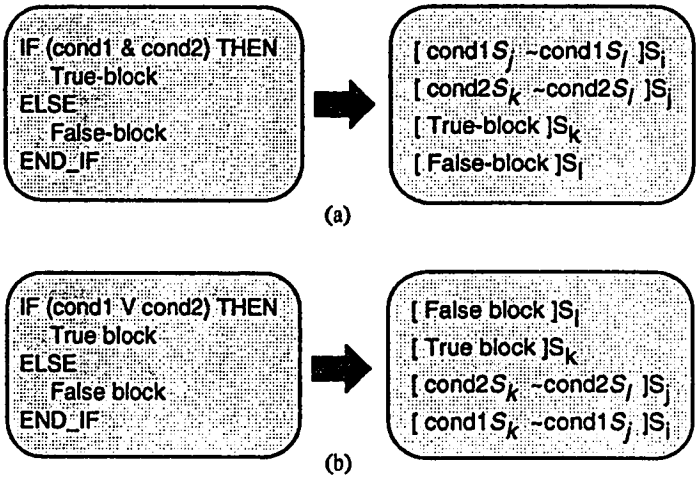


Figure 3.11: BRPN translation of Different logical operators in *If-Then-Else* statement.

A DFG models the data dependencies among the operators and variables. Furthermore, a DFG may be disconnected. The precedence relations on these data operations cannot be shown in a DFG alone. These relations are modeled by a CFG. The DFG and CFG are linked to resemble the behavioral model.

Flamel [Tri87] uses *Dacons*, which are CDFG. Each node represents a basic blocks, where a basic block is the largest number of statements in the code that does not contain any type of control construct and has one entry point and one exit point. The edges in *dacons* represent the control transfer among the basic blocks.

On the other hand, the structure in BRPN resembles the structure of HLL models. The basic entity in BRPN is a block. Blocks are not basic blocks as defined above. Instead they may contain control statements and loops. The definition of a

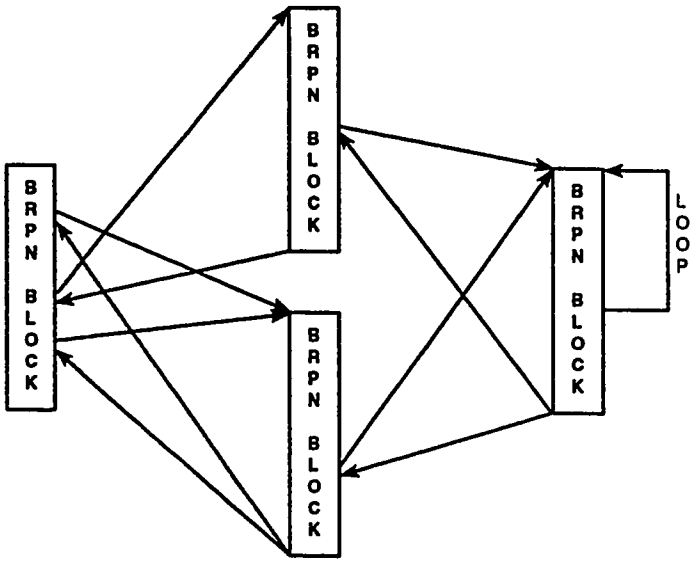


Figure 3.12: Blocking in BRPN.

block here is the largest number of stack operations that satisfy the following condition: "*control can only be transferred to the beginning of a block*". For this, blocks cannot be merged because this might violate the transfer of control condition. Moreover, optimization cannot be carried out on BRPN code. For example, a common sub-expression in different blocks cannot be eliminated. As each sub-expression is located in a different block and invoked at different instances. Therefore, BRPN code is translated into another IF for optimization. The precedence relation among the blocks is more like the strategy used in HLL subroutine calling. This is illustrated in Figure 3.12.

3.4 Conclusion

A new IF is presented. The translation of different HLL constructs into this IF (BRPN) is also presented. A comparison between the proposed IF and the known graph-based representation is conducted. The structure in BRPN resembles HLL subroutine calling mechanism. This is unlike DFG and CFG.

The translation of BRPN into the second intermediate form will be explained in the following chapter. The structural description can be extracted from the second IF. In Chapter 5, the developed hardware specific optimization will be explained.

Chapter 4

The Secondary Intermediate Form

In the previous chapter, the behavioral model of a digital system is translated into BRPN. As stated earlier the BRPN code resembles HLL description. The stack representation form does not allow the efficient implementation of the optimization tasks required to generate a quality RTL output. Graph-based IFs are most suitable in this case. Therefore, before attempting to carry any optimization step, we first transform the BRPN into a CDFG. This transformation is accomplished in a systematic way and with the same ease as the interpretation of the initial behavioral specification into BRPN.

This chapter presents the methodology and the algorithm developed to generate

the second IF from BRPN. This IF is a Canonical Register Transfer Language (CRTL). CRTL is a graph based IF.

Section 4.1 discusses the generation process of CRTL from BRPN. Section 4.2 presents the algorithm to perform this task. The algorithm builds up an internal data structure which is discussed in Section 4.3. One important issue is how to generate CRTL code for BRPN control operations. This issue is discussed in Section 4.4. The different HLL constructs shown in the previous chapter are translated into CRTL in Section 4.5. Some illustrative examples are given in Section 4.6. In Section 4.7 the target RTL is presented. We conclude in Section 4.8.

4.1 Generation of CRTL models from BRPN

After translating HLL descriptions into BRPNs, the CRTL models are extracted from these BRPNs. CRTL is a graph based IF. It is simpler to extract the hardware structural description from such IFs. Recall from Chapter 3 that the general block format in BRPN is:

[stack operations]Sx

In the above format, x is a distinct label given to the block of stack operations. These stack operations may either be operations on data or program control. Fig-

ure 3.2 shows some of the stack operations where *Oprnd* stands for operand, *Dopr* stands for data operation, *Copr* stands for control operation, and *Dltr* stands for a delimiter. More details are given in Section 4.2.

The stack operations in the above format are executed in sequence. This means that these operations are ordered. The location of each stack operation, whether data or control, inside the code is unique. As a result, the above format can be divided into a sequence of statements where each statement holds either a data or control operation and has a unique tag. Moreover, all branches to successor statements are defined. In this context, the concern is with the operations that cause popping from the stack.

The above description has similarities to some known RTL formats such as A Hardware Programming Language (AHPL) except that it is not as compact as AHPL description. Thus, the generated code is referred to as Canonical RTL (CRTL). It is called canonical because each statement holds either a data or a control operation.

There are two steps involved in generating the CRTL statements:

Step 1 Determine the CRTL statement number; this number is formed as a pair of block number and sequence number (BN,SN), where BN is the block number uniquely defined for each block of the stack code and SN is the sequence number associated with any operation within the block that causes popping

from the top of stack. SN is the relative position of the operation from the left of the BRPN block.

Step 2 Indicate whether the stack operation is a control or a data statement; if the stack operation is a data operation then the RTL statement is to hold the operation and the address to the next statement. Otherwise, the statement is to hold the condition of the operation and the addresses of the true and the false branches. The mechanism used to determine the true and false branches is described below.

The format of a CRTL statement consists of:

stmnt no.	data operation	control operation	true branch	false branch

Using this format, each statement in the code obtained from BRPN holds either (1) a data operation and an unconditional branch or (2) a conditional branch.

The two steps defined to generate CRTL statements are graphically illustrated in Figure 4.1.

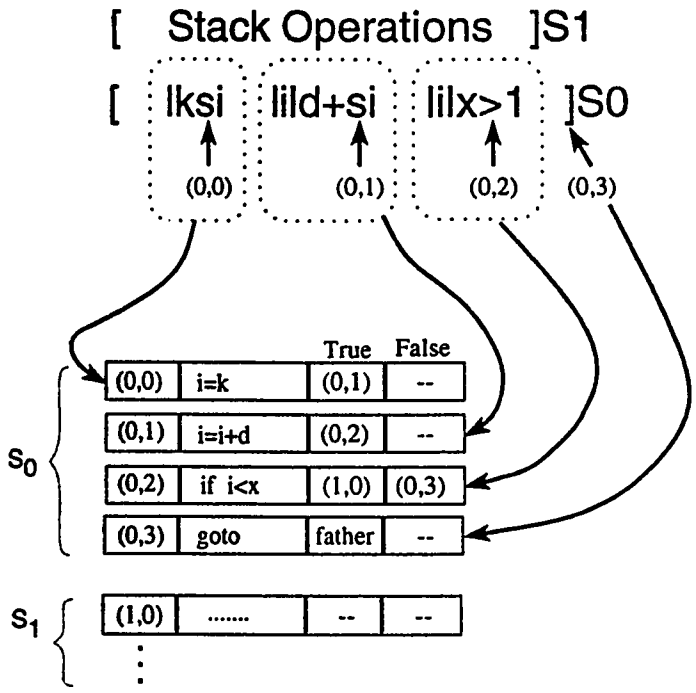


Figure 4.1: The translation of BRPN into CRTL.

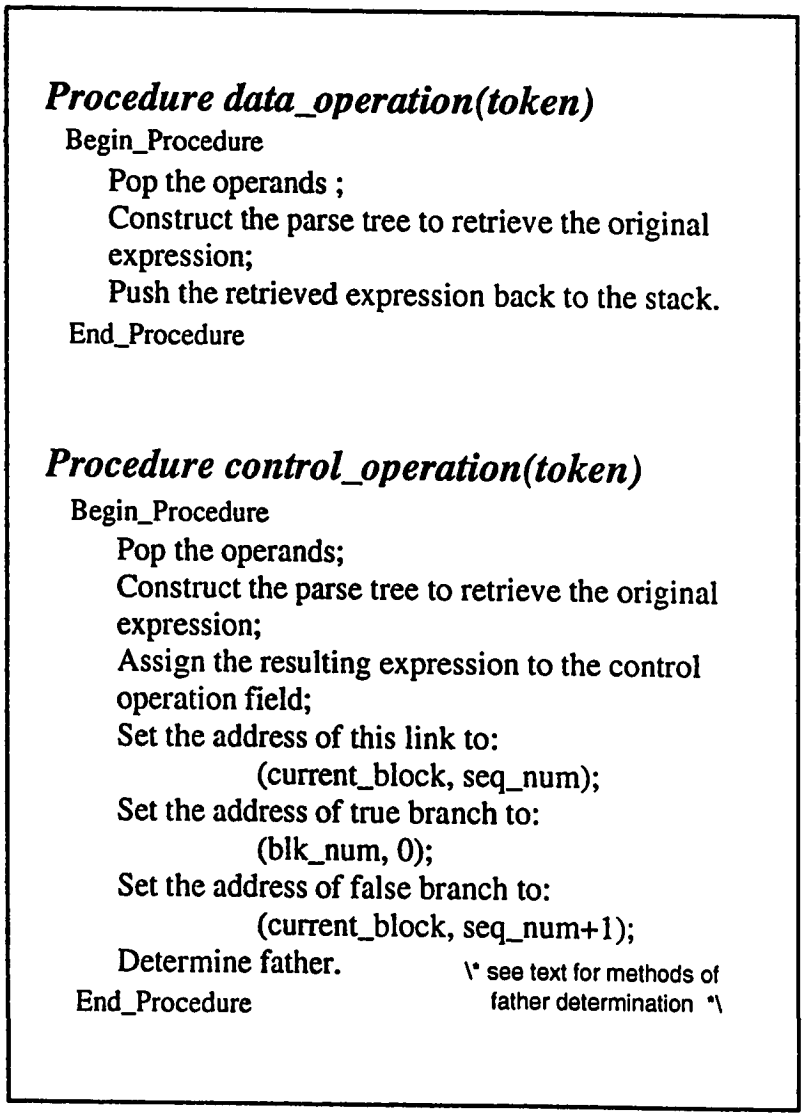
4.2 BRPN-to-CRTL Algorithm

The algorithm that accomplishes the conversion process from BRPN operation format to CRTL format is presented in Figure 4.2.

The main task of the algorithm is to extract the data and control parts. The input to the algorithm is the BRPN code translated from behavioral description. The algorithm outputs a generalized CDFG where nodes represent CRTL statements and edges represent the order of execution of these statements. The algorithm begins by reading a token. A *token* may be a delimiter (*Dltr*), an operand (*Opnd*), a data operation (*Dopr*), or a control operation (*Copr*). Delimiters are used to signal special events in the BRPN code, e.g. '[' and ']' signal the beginning and end of a block respectively. Operands may either be numeric constants or variables. Constant operands can be of any length and are always pushed onto the stack. Each variable operand starts with either 'l' for pushing the variable onto stack or 's' for popping the value on top of the stack and assigning it to the variable. Data operator characters are one character tokens. On the other hand a control operator consists of one or two characters. Figure 3.2 shows some of the operators and operands used by BRPN. Details on the translation algorithm are given in Section 4.6.

```
Current_block = -1
input(token)
WHILE  (not eof)
  CASE  token :
    token = '['                    \* The starting of a block *\
      Initialize a new block with seq_num=0;
      Current_block=block label.
    token = push operation
      Push operand to the stack.
    token = pop operation
      Pop operand from the stack;
      Form the expression:    operand=expression
       and put it in a new link;
      set the address of this link to:
                (current_block, seq_num)
      Increment seq_num.
    token = data operation     \* construct expression *\
      data_operation(token).
    token = control operation
      control_operation(token).
      Increment seq_num.
    token = ']'                    \* The ending of a block *\
      Increment seq_num;
      Current_block=-1.
  END_CASE
  input(token)
END_WHILE
```

Figure 4.2: The algorithm that generates CRTL code from BRPN.



Procedure data_operation(token)

```
Begin_Procedure
    Pop the operands ;
    Construct the parse tree to retrieve the original
    expression;
    Push the retrieved expression back to the stack.
End_Procedure
```

Procedure control_operation(token)

```
Begin_Procedure
    Pop the operands;
    Construct the parse tree to retrieve the original
    expression;
    Assign the resulting expression to the control
    operation field;
    Set the address of this link to:
            (current_block, seq_num);
    Set the address of true branch to:
            (blk_num, 0);
    Set the address of false branch to:
            (current_block, seq_num+1);
    Determine father.          \* see text for methods of
                                  father determination *\
End_Procedure
```

Figure 4.3: Procedures used by the algorithm in Figure 4.2.

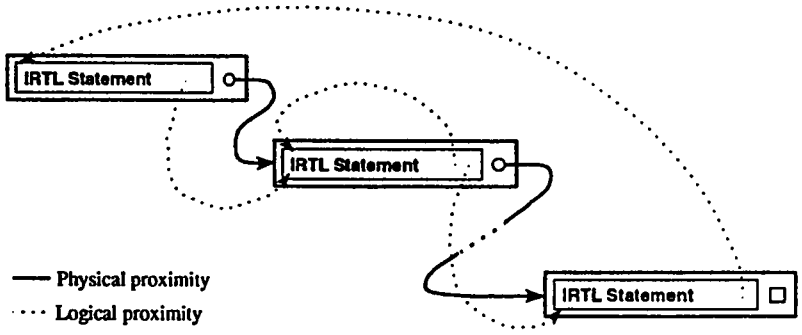


Figure 4.4: The internal data structure used by the algorithm.

4.3 Internal data structure

The data structure used to generate the CRTL code is a linear linked list. Each element of the linked list consists of a body that holds a single CRTL statement with the format defined in Section 4.1 and a pointer to the next element in the list.

This internal structure is not efficient because two types of links are used. One type is used to keep track of the physical proximity of the statements. The other type is for proper transfer of control as illustrated in Figure 4.4. A more efficient internal structure can be implemented where only one type of links is used.

The CRTL code generated from BRPN is stored in a single linked list for further processing. Therefore, each statement holds either a data or a control operation as defined previously. This sequence of statements is subject to further compaction in the following optimization steps.

4.4 True and false branching mechanism

The formation of true and false branches is only needed for control statements. To illustrate the mechanism used in forming these branches, let us consider the general format of a condition operation in a BRPN block:

$$[\ ...\mathbf{l}Opnd_1\ \mathbf{l}Opnd_2\ \mathbf{Rop}\ \mathbf{BN}\ ...\ ]S_i$$

In this statement, 'l' is the push operation, $Opnd_1$ and $Opnd_2$ are operands to be pushed on top of the stack, **Rop** is a relational operation on the two top values of the stack, and **BN** is the block number to branch to on true. Now, if the condition holds (true) then a branch takes place to the statement that has a block number equals BN and a sequence number equals to zero (BN,0), i.e. to the first statement in block BN. While on false, a branch takes place to the next operation in sequence in the current block (current block number, current sequence number +1). This is shown in Figure 4.1.

The father of the child block can be determined in two ways depending on the order of the blocks. In the case where the blocks are processed in descending order of their number, the children blocks are processed before their parents. Here, the return address of the child block is left empty till its father is defined. Once the control operation that calls the child block is located, the return address of the child block is set to the address of the next stack operation in the father block.

On the other hand, when the blocks are processed in ascending order, the father blocks are processed first. Each time a stack control operation is processed, the address of the next stack operation in sequence is saved in a table. This address is basically the return address of the child block. This table contains the branch to block (child) number and the return address. Once the child is processed, its return address is fetched from the table.

4.5 Generating CRTL for different HLL constructs

The translation of various HLL constructs into BRPN was discussed in the previous chapter. In this section we describe how we generate the CRTL model for each of the corresponding HLL constructs.

4.5.1 The *Repeat-Until* construct

Recalling Figure 3.4, the BRPN code of *Repeat-Until* construct is processed to generate the corresponding CRTL description. In block i, the *body* is translated into CRTL statements. Assuming the sequence number of the body is h, then the CRTL statement is '(i, h) *body*'. The next stack operation in sequence is a control operation. The transfer to block k takes place only when the condition *cond* is false. Hence, the corresponding CRTL statement is '$(i, h+1)$ $if \sim cond$ *then* $(k, 0)$'.

Block k is processed in a similar way. The generated CRTL code from the BRPN description is given in Figure 4.5.

```
(i,h)     body
(i,h+1)   if ~cond then (k,0)
   :
(k,0)     body
(k,1)     if ~cond then (k,0)
(k,2)     goto (i,h+2)
```

Figure 4.5: CRTL description of the *Repeat-Until* construct.

4.5.2 The *For* construct

Recalling Figure 3.5, the CRTL description is extracted from the BRPN code as follows. In block i, the first stack operation is translated into '$(i,h)\ i = x$' where h is the current sequence number. Then the stack control operation is translated into '$(i,h+1)\ if\ \ i < y\ \ then\ (k,0)$'.

The complete CRTL model for the *For* construct shown in Figure 3.5, is given in Figure 4.6.

4.5.3 The *If-Then-Else* construct

The BRPN description on *If-Then-Else* construct is given in Figure 3.6. There are two children blocks: the true block j and the false block k. The control transfer to

```
(i,h)     i=x
(i,h+1)   if i<y then (k,0)
   :
(k,0)     body
(k,1)     i=i+1
(k,2)     if i<y then (k,0)
(k,3)     goto (i,h+2)
```

Figure 4.6: CRTL description of the For construct.

one of these block takes place depending on the condition. In the father block i, the first control operation is translated into the corresponding CRTL description '(i,h) *if cond then* $(j,0)$' assuming h is the current sequence number. Similarly, the statement '$(i,h+1)$ *if* $\sim$ *cond then* $(k,0)$' is generated. The *If-Then-Else* CRTL description generated from the corresponding BRPN is given in Figure 4.7.

```
(i,h)     if cond then (j,0)
(i,h+1)   if ~cond then (k,0)
   :
(j,0)     true block
(j,1)     goto (i,h+1)
   :
(k,0)     false block
(k,1)     goto (i,h+2)
```

Figure 4.7: CRTL description of the *If-Then-Else* construct.

4.5.4 The *Case* construct

The *Case* construct is a multi-branch construct. The BRPN code of the *Case* construct is given in Figure 3.8. Processing this code as described above generates the

CRTL code shown in Figure 4.8.

```
(i,k)       if a=1 then (1,0)
(i,k+1)     if a=2 then (2,0)
   :
(i,k+n-1)   if a=n then (n,0)
   :
(1,0)       first block
(1,1)       goto (i,k+1)
(2,0)       second block
(2,1)      goto (i,k+2)
 :
(n,0)      n-th block
(n,1)      goto (i,k+n)
```

Figure 4.8: CRTL description of the Case construct.

4.6 Illustrative examples

Example 1

The first example shows the generation phases of a structural description from a behavioral description. The first phase is to translate the behavioral model of a digital system into BRPN. To illustrate the process of translating algorithmic (behavioral) descriptions into BRPN, we use the bubble sort algorithm shown in Figure 4.9.

Using the stack operations given in Figure 3.2, the BRPN code corresponding to the above algorithm is given in Figure 4.10.

```
a[1]=21
a[2]=19
a[3]=13
a[4]=12
a[5]=1
for (i=5; i>1; i--) {
   for (j=1; j<i); j++) {
      if (a[j]>a[j+1]) {
         t=a[j]
         a[j]=a[j+1]
         a[j+1]=t
      }
   }
}
```

Figure 4.9: The C-like description of Bubble Sort Algorithm.

```
21  1:!
19  2:!
13  3:!
12  4:!
 1  5:!
[lj;!st lj 1+;!lj:1ltlj 1+:!]S2
[lj;!lj 1+;!<2 ljd1+sj ljli>1]S1
[1dsj ljli>1 lid1-si li 1<0]S0
5dsi li 1<0
q
```

Figure 4.10: The BRPN description of Bubble Sort Algorithm.

The second phase is to apply the algorithm presented in Section 4.2 in order to generate CRTL from this BRPN. To illustrate the steps of the algorithm, consider the following BRPN code segment of the bubble sort algorithm taken from 4.10:

```
[lj;!st lj 1+;!lj:!ltlj 1+:!]S2
[lj;!lj 1+;!<2 ljd1+sj ljli>1]S1.
```

In the first line of the code '[' indicates the start of the block. The block number (blk_num) is derived from the block label. In this case, blk_num equals 2 and the sequence number (seq_num) is reset to zero. The next input token is 'lj;', causing the variable 'j' to be pushed into the stack. However, since 'j' is followed by ';', the variable is interpreted as an array index and the name of the array follows the ';'. The variable name is found by taking the character after ';' (which is !) and adding 64 to its ASCII value to get the ASCII value of the actual variable (in this case 'a'). This coding is used to avoid directly using the alphabetical characters as variable names in the BRPN output thus avoiding conflicts with the stack operations. Therefore, 'a[j]' is pushed into the stack. The next token is 'st' which means pop the top of the stack to the variable 't', form the expression 't=a[j]', and assign it to the data operation field of the current link (the first link). The address field of the link is also created: (blk_num,seq_num)=(2,0). The seq_num is then incremented to 1.

The next token is 'lj', and the resulting action is push 'j' into stack, followed by 1, which is also pushed into the stack. Next the operator '+' will cause the top two elements of the stack to be popped, and the expression 'j+1' is pushed back

into the stack. The ';' indicates that the expression 'j+1' on top of the stack is an array index. The next token 'lj' causes 'j' to be pushed in the stack, the ':' indicates an array operation that pops the top element of the stack (in this case 'j'), uses it as an array index and assigns to it the next top element of the stack ('a[j+1]' in this example). The variable or array name (which is 'a') is also found as mentioned previously. The expression thus formed is 'a[j]=a[j+1]'. Note that ':' is of the pop-operand type and thus the resulting expression is assigned to the statement in the current link, and the address formed is '(2,1)'. The seq_num is then incremented.

The algorithm continues in the same way to form the cell 'a[j+1]=t', and the delimiter ']' indicates the end of the block. Since the father of this block is not yet defined, it is left temporarily empty.

Now the next block, given below, is read:

```
[lj; !lj 1+;! <2ljd1 +sjs. ljli >1]S1
```

The blk_num is 1 and the seq_num is reset to 0. The statements 'lj; ! lj 1 +;' will result in forming the expressions 'a[j]' and 'a[j+1]' as the top two elements of the stack. The token '<2' will form the condition 'a[j] < a[j+1]' and assign it to the control operation field of the link. The address of this link is (current blk_num, current seq_num) which is (1,0). The true branch field gets (2,0) and the false branch field gets (1,1). The "Father" of the called block (which is block 2) is (1,1). Since the

father of block 2 is now defined, the generated code is updated accordingly. Finally, the generated CRTL code is:

```
(1,0)  if a[j+1]<a[j] then (2,0)
(1,1)  j<=j+1
(1,2)  if i>j then (1,0)
(1,3)  goto 'father'
(2,0)  t<=a[j]
(2,1)  a[j]<=a[j+1]
(2,2)  a[j+1]<=t
(2,3)  goto (1,1)
```

The complete bubble sort CRTL model generated from BRPN is given in Figure 4.11. Note that the stack operations outside the blocks are grouped into block number -1.

Example 2

Consider the following HLL of *If-then-else* construct with logical operators:

```
a=1
b=3
if (a==1 & b==2) then
   s=a+b
else
   s=b-a
end_if
t=2*s
```

The first step is the translation of the above behavioral description into BRPN.

```
     (-1,0)   a[1]=21.
     (-1,1)   a[2]=19.
     (-1,2)   a[3]=13.
     (-1,3)   a[4]=12.
     (-1,4)   a[5]=1.
 1.  (-1,5)   i<=5.
 2.  (-1,6)   if i>1 then (0,0).
 3.  (-1,7)   End.
 4.   (0,0)   j<=1.
 5.   (0,1)   if i>j then (1,0).
 6.   (0,2)   i<=i-1.
 7.   (0,3)   if i>1 then (0,0).
 8.   (0,4)   goto (-1,7).
 9.   (1,0)   if a[j+1]<a[j] then (2,0).
10.   (1,1)   j<=j+1.
11.   (1,2)   if i>j then (1,0).
12.   (1,3)   goto (0,2).
13.   (2,0)   t<=a[j].
14.   (2,1)   a[j]<=a[j+1].
15.   (2,2)   a[j+1]<=t.
16.   (2,3)   goto (1,1).
```

Figure 4.11: CRTL description of Bubble Sort Algorithm.

The resulting BRPN code is:

```
1 sa
2 sb
[3 la+ su]S3
[1alb+ su]S2
[2 lb=2 2 lb!=3]S1
[1 la=1 1 la!=3]S0
2 lu* st
q
```

Applying the algorithm given in this chapter, in the same manner described earlier, the CRTL code is generated.

In Block '0', the first condition ($a == 1$) is evaluated. If the condition is true, the control is transferred to Block '1' where the other condition ($b == 1$) is evaluated. Otherwise, control is transferred to the block that holds the BRPN translation of the false code (Block '3'). Similarly, the condition ($b == 1$) is evaluated in Block '1'. If the condition is true, the control is transferred to Block '2' which holds the BRPN translation of the true code. Else, control is transferred to Block '3'. The complete CRTL code is given in Figure 4.12.

4.7 AHPL as a target RTL

The generation of AHPL code from CRTL models is described in this section. The language AHPL uses the convention that any digital system can be divided into a

```
(-1,0)  a<=1
(-1,1)  b<=2
(-1,2)  t<=2*s
(-1,3)  End.
( 0,0)  if (a==1) then (1,0)
( 0,1)  if (a<>1) then (3,0)
( 0,2)  goto (-1,2)
( 1,0)  if (b==2) then (2,0)
( 1,1)  if (b<>2) then (3,0)
( 1,2)  goto (0,1)
( 2,0)  s<=a+b
( 2,1)  goto (1,1)
( 3.0)  s<=3+a
( 3,1)  goto (0,2)
```

Figure 4.12: CRTL description of the algorithm in Example 2.

data part and a control part. Recall from Section 4.3 that the information about the data transfer and the control flow is stored in the form of a linked list. Then the corresponding RTL specification in AHPL can easily be obtained from this linked list. In this section, a very brief overview of the AHPL language is presented, following which, the generation procedure of the AHPL code from CRTL is discussed.

This section is not intended to detail the entire AHPL language but to mention only the basic constructs. We also restrict the discussion to the *SEQUENCE section* of the AHPL model. This is the section that models the finite state automaton.

The AHPL SEQUENCE section consists of a sequence of numbered steps, each step representing a state of the finite state machine. A step may have transfer statements '$\Leftarrow$', connections to buses '$=$', or conditional/unconditional branches. Con-

ditional branches are expressed as '$\Rightarrow (f_1, f_2, \ldots, f_n)/(S_1, S_2, \ldots S_n)$', which reads as, if condition f_i is true, then, in the next clock pulse, transfer control to Step S_i; else transfer to the following step in sequence. Unconditional branching to Step S_i is expressed as '$\Rightarrow (S_i)$'. Conventionally, in AHPL, all transfers to registers, and state transitions, are assumed to take place at the trailing edge of the clock pulse, whereas transfer to buses are active during the entire duration of the clock. In addition, an abbreviated form for expressing combinational functions called CLUs (combinational logic units or functional units) is available. Complex combinational logic circuits can be modeled separately as CLUs, and then invoked in the sequential part of the description when needed. Examples of combinational logic units are comparators (COM), bus_functions (BUSFN), binary decoders (DCD), binary incrementers (INC), and binary decrementers (DEC). The comparator is used to compare two vectors, and has 3 outputs. For example, in COM(I;J) where I and J are binary codes, the third output bit of the CLU, that is COM{2} is high if condition '$I < J$' is true. Reading from memory is done through the decode (DCD) and the bus_function (BUSFN) CLUs. The DCD unit decodes the given address and enables the corresponding memory location value. CLU BUSFN then routes the value of the enabled location to the destination. Thus a memory read is accomplished as follows,

```
destination_register<=BUSFN(MEMORY,DCD(address_register)).
```

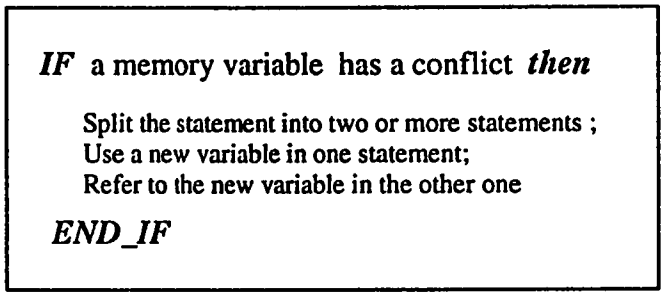


Figure 4.13: Algorithm for memory conflict resolution.

Writing to the memory is done in a similar way. The address is decoded using DCD and the corresponding memory location is enabled for writing. The instruction is

```
MEMORY*DCD(address_register)<=source.
```

For a more detailed overview of AHPL the reader may refer to [SBK93]. The complete documentation on the language and its grammar are available in [Mas81].

With this introduction to AHPL, let us consider the memory conflict issue which is a by-product of interpreting HLL descriptions. Statements such as '$a[j] < a[j+1]$' are possible in high level descriptions. However, in AHPL it is not possible to read two locations of the memory bank simultaneously and compare their contents. Similarly, it is not possible to simultaneously read and write, i.e. move the contents of one memory location into another in a single step. To resolve this conflict, the action is divided into two steps and temporary registers are introduced as described in Figure 4.13. For the bubble sort example in Figure 4.11, there are two steps where we have memory conflicts: steps 9 and 14. They are resolved by splitting

Step 9 into 9.1 and 9.2 and introducing temporary registers YY1. While Step 14 is split into 14.1 and 14.2 and register YY2 is used. The new generated AHPL model for the bubble sort is given in Figure 4.14.

```
MODULE   : SORT.
MEMORY   : A{8}<8>;I{3};J{3};T{8}.
MEMORY   : YY1{8}; YY2{8}.
EXINPUTS : RESET;CLOCK.
CLUNITS  : DCD{6};INC{3};BUSFN{12};COM3{3};COM8;DEC{3}.
BODY SEQUENCE:CLOCK.
  (-1,5)  1     I<=\1,0,1\.
  (-1,6)  2     =>COM{2}(\0,0,1\;I)/(4).
  (-1,7)  3     DEADEND.
  ( 0,0)  4     J<=\0,0,1\.
  ( 0,1)  5     =>COM3{2}(J;I)/(9.1).
  ( 0,2)  6     I<=DEC(I).
  ( 0,3)  7     =>COM3{2}(\0,0,1\;I)/(4).
  ( 0,4)  8     => (3).
  ( 1,0)  9.1   YY1 <= BUSFN(A;DCD(J)).
  ( 1,0)  9.2   => COM8{2}(BUSFN(A;DCD(INC(J)));YY1)/(13).
  ( 1,1) 10     J<=INC(J).
  ( 1,2) 11     =>COM3{2}(J;I)/(9.1).
  ( 1,3) 12     => (6).
  ( 2,0) 13     T<=BUSFN(A;DCD(J)).
  ( 2,1) 14.1   YY2<=BUSFN(A;DCD(INC(J))).
  ( 2,1) 14.2   A*DCD(J)<=YY2.
  ( 2,2) 15     A*DCD(INC(J))<= T.
  ( 2,3) 16     =>(10).
ENDSEQUENCE
CONTROLRESET(RESET)/(1).
END.
```

Figure 4.14: The bubble sort example after resolving memory conflicts.

4.8 Conclusion

A general methodology to generate CRTL from BRPN is proposed. The algorithm to accomplish this is developed and presented. The internal data structure generated by this algorithm is discussed. One major issue in generating CRTL is the proper processing of control constructs. In this chapter, we described algorithms that are used to obtain CRTL code from the primary IF (BRPN). Several examples are used to illustrate the translation.

The target RTL description in this work is AHPL and the extraction of AHPL from CRTL is also discussed. The issue of memory conflicts which are inherently present in interpreting HLL descriptions is addressed. A methodology to resolve such conflicts is proposed.

As described earlier in this chapter, the RTL code is not optimized. In the next chapter, software as well as hardware specific optimization techniques will be presented in order to further reduce the number of control steps in the RTL code.

Chapter 5

Optimization of AHPL Descriptions

In Chapters 3 and 4, the synthesis of an AHPL description from a behavioral description of a digital system was discussed. The generated AHPL code contains statements holding either a single data operation and an unconditional branch, or a conditional branch operation.

Scheduling is an essential step of HLS. Besides deciding the correct ordering of the operations, it also determines the number of various hardware resources required by the following allocation step. The outcome from the scheduling step depends

entirely on two main features:

1. the scheduling objective, and

2. the existence of constraints, such as limited resources.

In this work we assume unlimited resources, and our objective is to obtain an RTL description with the smallest number of control steps (CSteps). Two types of optimization are considered: software optimization and hardware optimization.

Section 5.1 discusses the software optimization techniques used. While hardware specific optimization is addressed in Section 5.2. Hardware specific optimization is based on two transformations: loop transformation and switch transformation. The application of these techniques is shown in Section 5.3. We conclude in Section 5.4.

5.1 Software Optimization

In this optimization, compiler-like optimization techniques are applied. Unconditional branch elimination and code factorization are the two major types of optimization applied.

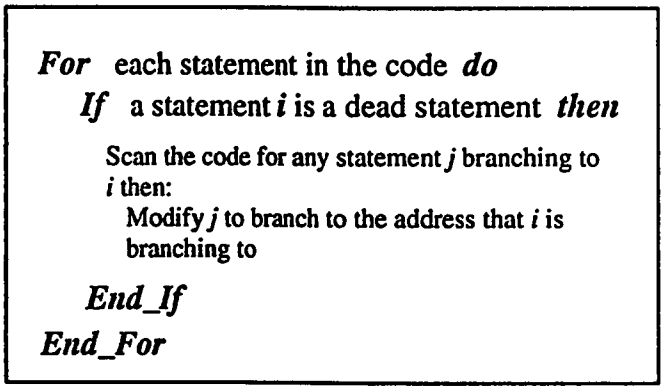


Figure 5.1: The unconditional branch elimination algorithm.

5.1.1 Unconditional branch elimination

Each transfer of control from a child block to its father is an unconditional branch statement. These statements are extra and therefore are deleted. Their true branch addresses are copied into the true or the false branches of the calling statements. The algorithm used for unconditional branch elimination is shown in Figure 5.1 and has an $O(n^2)$ complexity; where n is the number of statements in the AHPL code. The algorithm initially scans every statement looking for unconditional branches. A second scan is done by the algorithm for branch adjustment. For example, consider the following code segment:

```
10      J<=INC(J); => (5).
 :
15      A*DCD(INC(J)) <= T.
16      => (10).
```

Statement 16 is an unconditional branch. It is deleted and Statement 15 is

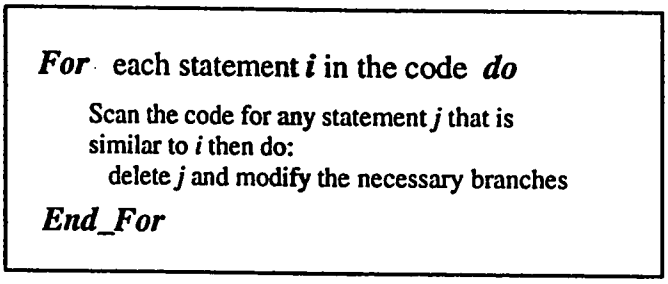


Figure 5.2: The code factorization algorithm.

adjusted as follows:

```
15  A*DCD(INC(J))<= T;    => (10).
```

5.1.2 Code factorization

Similar expressions can be found in different blocks. These redundant expressions are easy to identify and factored out. The algorithm to do this is given in Figure 5.2 and has an $O(n^2)$ complexity; where n is the number of statements in the AHPL code. Two scans are carried out by the algorithm. One to pick the statement and the other to check for identical statements and delete them. The branches of the calling statements are adjusted accordingly. For example consider the following code segment:

```
2       =>COM{2}(\0,0,1\;I)/(4).
:
6       I<=DEC(I).
7       =>COM{2}(\0,0,1\;I)/(4).
```

Statements 2 and 7 have similar expressions. Thus, one of them is kept to represent the other. In here Statement 7 is deleted. Now the branch out of Statement 6 is adjusted to point to Statement 2 as follows:

```
6  I<=DEC(I)  =>  (2).
```

5.1.3 Example

To illustrate the optimization of this phase, consider the AHPL code of the bubble sort given at the end of Section 4.7. The resulting code after software optimization is given in Figure 5.3. The order of applying these optimization techniques may give different results. But for this work (in Figure 4.14), unconditional branch elimination is applied first. Steps 8, 12 and 16 are unconditional branches (GOTO). Thus they are deleted. The branchings in Steps 7, 11 and 15 are modified to include the transfers to Steps 3, 6 and 10 respectively. Applying code factorization, Steps 2 and 7 are found redundant as they have similar expressions. Hence Step 7 is deleted and Step 6 is modified to branch to Step 2. Moreover, as Step 3 is the successor of Step 2, it is not necessary to include transfer of control to Step 3 in the set of output branch addresses of Step 2. Similarly, Steps 5 and 11 are redundant and Step 11 is deleted. Modification in the code is done in a similar manner. The control steps in the resulting code has been reduced by five control steps. The optimized AHPL is given in Figure 5.3.

```
MODULE   : SORT.
MEMORY   : A{8}<8>;I{3};J{3};T{8}.
MEMORY   : YY1{8}; YY2{8}.
EXINPUTS : RESET;CLOCK.
CLUNITS  : DCD{6};INC{3};BUSFN{12};COM3{3};COM8;DEC{3}.
BODY SEQUENCE:CLOCK.
  (-1,5)  1     I<=\1,0,1\.
  (-1,6)  2     =>COM{2}(\0,0,1\;I)/(4).
  (-1,7)  3     DEADEND.
  ( 0,0)  4     J<=\0,0,1\.
  ( 0,1)  5     =>COM3{2}(J;I)/(9.1).
  ( 0,2)  6     I<=DEC(I); => (2).
  ( 1,0)  9.1   YY1 <= BUSFN(A;DCD(J)).
  ( 1,0)  9.2   => COM8{2}(BUSFN(A;DCD(INC(J)));YY1)/(13).
  ( 1,1) 10     J<=INC(J); => (5).
  ( 2,0) 13     T<=BUSFN(A;DCD(J)).
  ( 2,1) 14.1   YY2<=BUSFN(A;DCD(INC(J))).
  ( 2,1) 14.2   A*DCD(J)<=YY2.
  ( 2,2) 15     A*DCD(INC(J)) <= T; => (10).
ENDSEQUENCE
CONTROLRESET(RESET)/(1).
END.
```

Figure 5.3: AHPL description of Bubble Sort Algorithm after applying software optimization.

5.2 Hardware Specific Optimization

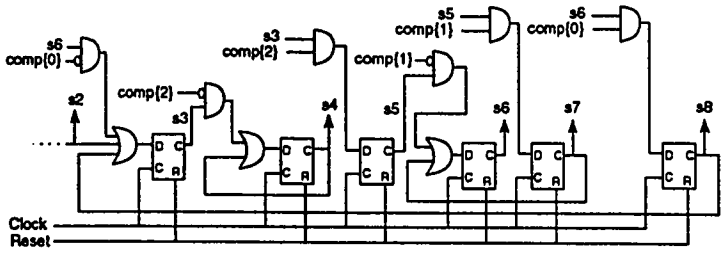
In the previous section, superfluous CSteps due to unconditional branches and duplicate statements are detected and eliminated. To produce optimized AHPL code, further optimization techniques should be applied. The techniques used here are *optimization by successive transformations* that are applied for the purpose of identifying mergable CSteps. The goal here is to further minimize the number of CSteps in the AHPL model.

Since we assume unlimited resources, minimizing the number of CSteps is equivalent to maximizing parallelism, i.e., assign as many statements as possible to the same CStep. This will achieve other objectives as well: (1) elimination of multiplexers, (2) obtain smaller controller, and (3) provide more opportunity for logic minimization. In Figure 5.4(a), an AHPL model of a digital system is given. The control part of this AHPL model is shown in Figure 5.4(b). Merging different CSteps (see Figure 5.4(c)) produces a smaller control circuit, as shown in Figure 5.4(d).

Hence, as far as the logic in the AHPL data part is concerned, the chance to eliminate redundant logic increases. Silicon compilers like AHPL silicon compiler run logic optimization on their input models. The optimization is only applied to the logic within a CStep rather than the logic in the whole model. By clustering as many statements as possible in a single CStep, we increase the chance for logic optimization. Thus, leading to the minimization of overall silicon area. This has

```
3    => (comp{2}(x,y),~comp{2}(x,y)) / (5,4).
4    => (4).
5    => (comp{1}(x,y),~comp{1}(x,y)) / (7,6).
6    => (comp{0}(x,y),~comp{0}(x,y)) / (8,3).
7    y<= INC(ADD(y,~x)); => (6).
8    x<= INC(ADD(x,~y)); => (3).
```

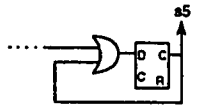
(a) AHPL model



(b) Control unit of above model

```
5    y*comp{1}(x,y)<= INC(ADD(y,~x));
     x*comp{0}(x,y)<= INC(ADD(x,~y));    => (5).
```

(c) AHPL model after merging CSteps (Optimized model)



(d) Control unit of optimized model

Figure 5.4: Example of merging different CSteps.

been confirmed by experimental results as will be discussed in Section 6.2.

Hardware specific optimization algorithms are developed to accomplish the above task. Central to these algorithms is a template which is the subject of the following paragraphs.

5.2.1 Template

Recalling the general format of CRTL statements described in Section 4.1, each statement in the CRTL code is mapped into the following template and tagged with its reference number:

$$< Bin_i , O_i , I_i , Bout_i >$$

where,

Bin_i is the set of all predecessor statement numbers of statement i,

O_i is the set of all output variables in statement i,

I_i is the set of all input (used) variables in statement i, and

$Bout_i$ is the set of all successor statement numbers ($|Bout_i| > 0$). If $|Bout_i| = 0$ then statement i is a **dead end.**

The sets O_i and I_i for the i^{th} statement are formed from the variables found either in the data or the condition operations.

Using this template, two optimization techniques are proposed: loop transformation and switch transformation.

5.2.2 Loop transformation

Loops can be: predicate loops or counter loops. For a predicate loop, the execution is controlled by a boolean expression. Changes in the logic value of this expression are subject to the body of the loop. The optimization involved in these loops is similar to the switch transformation which is discussed in the next section.

A counter loop, in general, consists of an initialization step, a compare step, a body, and an adjust step. The number of executions in a loop is controlled by a variable (*cv*). Every time an iteration is completed, *cv* is adjusted accordingly. In AHPL, the adjust and compare steps can be performed in the same CStep. The reason is that in AHPL, register transfers occur at the trailing edge of the clock. This means that the new value of a variable is set at the trailing edge. During the clock, the old value is unchanged. We refer to the operation of merging the adjust and compare steps as *loop transformation.* Figure 5.5 illustrates this merging. However, one has to be careful because this merging requires that *cv* be adjusted inside as well as outside the loop. The adjustment is essential as *cv* holds the next value at

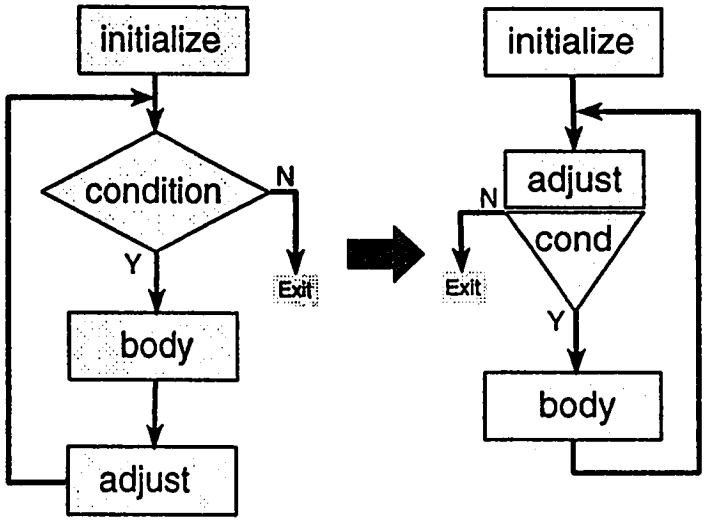


Figure 5.5: The definition of loop transformation.

the beginning of the loop rather than at the end.

The variable *cv* appears in both the data and the control expressions after this merging. Because of the aforementioned property of AHPL register transfers, *cv* in the control part of the compare step stays unchanged. The new value of *cv* is set in the next clock cycle.

To adjust the value of *cv* inside and outside the loop, we need to backtrack to all locations where *cv* is invoked. The expressions that use *cv* are replaced with new expressions containing the adjusted value of *cv*. That is, outside the loop, the effect of the adjustment step is undone. For example, assume that the adjust step for a loop contains '$cv = cv + x$'. Then *cv* in any expression inside or outside the loop is replaced with '$cv - x$'. We shall call this adjustment step as *ADJUST(cv)*.

```
IF  (O_i ∩ I_i ≠ Ø & |O_i ∩ I_i| = 1)  THEN
    cv = O_i ∩ I_i          \* cv is the controlling variable *\
    Search for S_j | cv ∈ Cond(S_j)
    IF  S_j is located  THEN
        BackTrack(S_i, ADJUST(cv))
        Merge S_i into S_j
    END_IF
END_IF
```

Figure 5.6: Algorithm for loop transformation.

The loop transformation algorithm is given in Figure 5.6. The algorithm is executed when the adjust step is located. Let S_i be a particular statement and I_i and O_i be respectively, the set of input variables and the set of output variables of statement S_i.

Definition 1 *A statement S_i is a loop adjustment step if and only if $I_i \cap O_i \neq \emptyset$ and $I_i \cap O_i = \{cv\}$ where cv is the loop controlling variable.*

Once an adjust step is identified, a search for the compare step is initiated. Once the compare step is located, a backtrack procedure is invoked to find the places where *cv* is used, and the variable *cv* is adjusted as described above.

To illustrate the algorithm, consider the AHPL model for the bubble sort example of Section 5.1. The flowchart representing the partially optimized AHPL state

```
Procedure BackTrack(S_i , ADJUST(cv))
  Begin_Procedure
      FOR  each S_k ∈ Bin(S_i)
          BackTrack(S_k , ADJUST(cv))
          IF  cv ∈ O_i ∪ I_i  THEN
              Replace cv with ADJUST(cv)
          END_IF
      END_FOR
  End_Procedure
```

Figure 5.7: Back Track Procedure.

machine is shown in Figure 5.8.

Loop transformation is illustrated in Figure 5.9. The variable 'j' is *cv*. The shaded region in this figure represents the body of the loop, and is executed if the condition '$i > j$' is true (Step 5). If the condition is false then control branches to Step 6. The adjustment of *cv* is done in Step 10 '$j <= j + 1$'. Figure 5.9(b) illustrates the merging in Steps 5 and 10, and the adjustment of the value of j both inside the loop and at Step 6 where j is decremented.

Lemma 1 *A loop transformation reduces the number of Control Steps by at least one.*

Proof: A loop has a compare step and an adjust step. Merging the adjust step with the compare step results in reducing the total number of CSteps by one. The

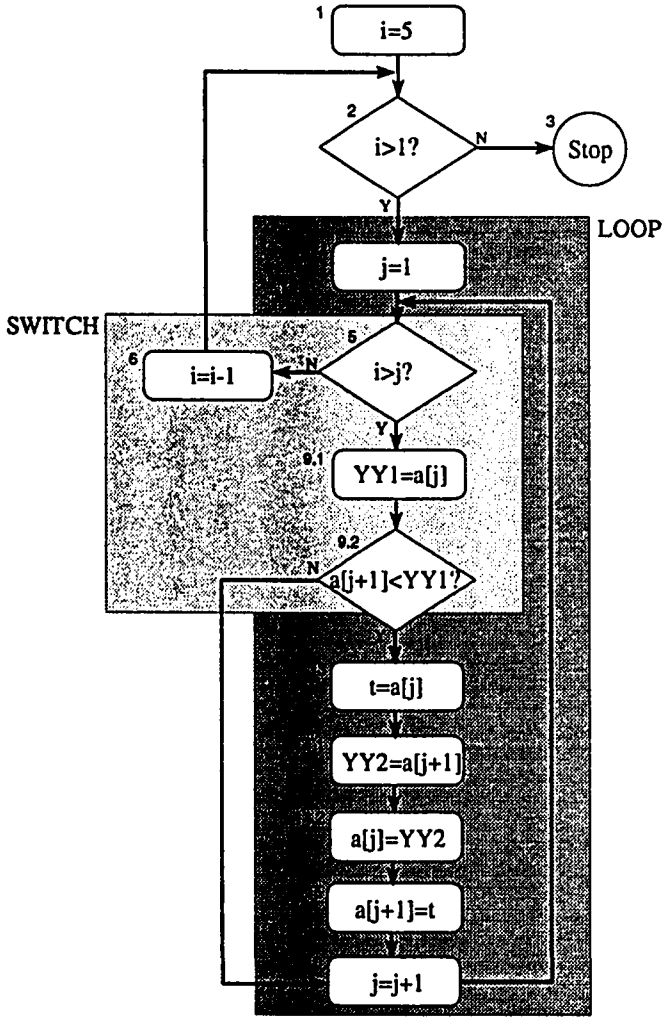


Figure 5.8: Partially optimized AHPL model for bubble sort example.

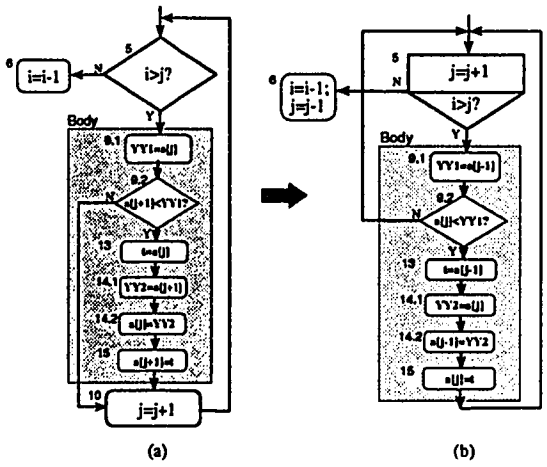


Figure 5.9: Flow chart illustrating loop transformation.

adjustments that may be required to statements where *cv* is used outside the loop do not cause any increase in the number of CSteps. ∎

To derive the complexity of this transformation, let us consider the extreme case where we have n statements with $n/2$ nested loops as shown in Figure 5.10. The loop

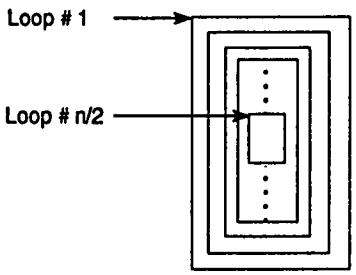


Figure 5.10: Nested loop used in calculating the complexity of loop transformation.

transformation algorithm will find the adjust step for the inner loop (Loop# $n/2$) in $n/2$ comparisons while the continuity step will be located in one comparison. Backtracking requires at most n comparisons.

In the second iteration for the algorithm, $n/2$ comparisons are required to find the adjust step, two to locate the continuity step and at most $n-1$ for backtracking. This continues till identifying the last loop (Loop# 1). Here the transformation will locate the adjust step in $n/2$ comparisons and the continuity step in $n/2$ comparisons too. Therefore, the total number of comparisons required to find all the adjust and continuity steps is $(n^2+2n)/2$. While backtracking requires a total of $n^2/4$ comparisons. Then the overall number of comparisons is $(3n^2+2n)/4$. Hence, the loop transformation algorithm has an $O(n^2)$ complexity, where n is the number of statements in the code.

5.2.3 Switch transformation

A statement is a *switch* if it branches to more than one statement. A condition is associated with each branch. The branch is taken (executed) if the corresponding condition is true.

If a statement is switching to a number of statements, then these statements can be merged into one statement. The condition can either be inclusive or mutually exclusive. In mutually exclusive conditions, one and only one condition is true. Whereas, in inclusive conditions, none or at least one condition is true.

Mutually exclusive conditions

In the case of mutually exclusive conditions, the switch forks to at most 2^n branches; where n is the number of boolean variables forming the conditions. To illustrate this transformation on AHPL descriptions, consider the following general AHPL code:

Step	Data transfers	Control transfers
$k.$	$stmt_k;$	$\Rightarrow (m_1, m_2, \ldots, m_{2^n})/(s_1, s_2, \ldots, s_{2^n}).$
$\vdots$	$\vdots$	$\vdots$
$s_1.$	$stmt_{s_1};$	$\Rightarrow (x_1, x_2, \ldots, x_l)/(t_1, t_2, \ldots, t_l).$
$s_2.$	$stmt_{s_2};$	$\Rightarrow (y_1, y_2, \ldots, y_p)/(u_1, u_2, \ldots, u_p).$
$\vdots$	$\vdots$	$\vdots$
$s_{2^n}.$	$stmt_{s_{2^n}};$	$\Rightarrow (z_1, z_2, \ldots, z_q)/(v_1, v_2, \ldots, v_q).$

In AHPL, more than one register transfer operation can take place in a single control step. Moreover, these register transfers can be conditionally controlled. Thus, a data transfer operation will only be executed if the corresponding condition is true. Therefore, the above code can be transformed to a more compact form as follows,

$$k. \quad stmt_k; stmt_{s_1} * m_1; stmt_{s_2} * m_2; \ldots; stmt_{s_{2^n}} * m_{2^n};$$
$$\Rightarrow (x_1 m_1, \ldots, x_l m_1, y_1 m_2, \ldots, y_p m_2, \ldots, z_1 m_{2^n}, \ldots, z_q m_{2^n})/$$
$$(t_1, \ldots, t_l, u_1, \ldots, u_p, \ldots, v_1, \ldots, v_q).$$

The conditional branch in the transformed statement is the *and* operation between the switch conditions and the conditions of the branches. Figure 5.11 shows

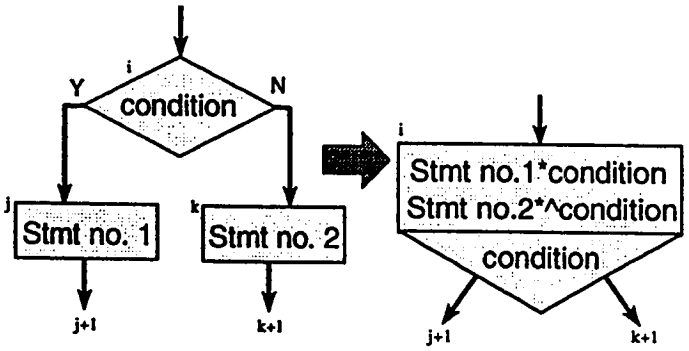


Figure 5.11: Switch transformation of an *If-Then-Else* statement.

the switch transformation technique applied to *If-Then-Else* statement.

Lemma 2 *A memory conflict free switch with n-boolean variables forming* 2^n *mutually exclusive conditions is reduced by* 2^n *Csteps.*

Proof: The total number of CSteps forming this switch is $2^n + 1$ (the condition statement + 2^n branches). If there is no memory conflicts between the branches and the switch, then these branches are merged into the switch. This results in a code reduction of 2^n CSteps. ∎

Lemma 3 *A switch with n-boolean variables forming* 2^n *mutually exclusive conditions is reduced by at least* $2^n - 1$ *CSteps.*

Proof: As in the above lemma, 2^n branches are formed. In the case of memory conflict free condition we achieve a reduction by 2^n CSteps. In the presence of

memory conflicts, two CSteps are needed, one to evaluate the conditions and the other to perform the corresponding register transfer operation. A total of $2^n - 1$ CSteps are saved. ■

Inclusive conditions

Unlike mutual exclusive conditions, inclusive conditions are independent. This is to say that the variables forming the conditions are not related. Moreover, the number of conditions forming the conditional branch in an AHPL statement is not limited. Thus, if none of the conditions is true, the next AHPL statement in sequence is executed. To show the optimization involved in this case, consider the AHPL statements below:

Step	Data transfers	control transfers
$k.$	$stmt_k;$	$\Rightarrow (m_1, m_2, \ldots, m_i)/(s_1, s_2, \ldots, s_i).$
$k+1.$	$stmt_{k+1};$	$\Rightarrow (\ldots).$
$\vdots$	$\vdots$	$\vdots$
$s_1.$	$stmt_{s_1};$	$\Rightarrow (x_1, x_2, \ldots, x_l)/(t_1, t_2, \ldots, t_l).$
$s_2.$	$stmt_{s_2};$	$\Rightarrow (y_1, y_2, \ldots, y_p)/(u_1, u_2, \ldots, u_p).$
$\vdots$	$\vdots$	$\vdots$
$s_i.$	$stmt_{s_i};$	$\Rightarrow (z_1, z_2, \ldots, z_q)/(v_1, v_2, \ldots, v_q).$

Because $m_1, m_2, \ldots, m_i$ are independent conditions, there is a probability that none of them is true. And therefore, a correct transformation of the above $i + 2$ CSteps is as follows,

k. $stmt_k; stmt_{s_1} * m_1; stmt_{s_2} * m_2; \ldots; stmt_{s_i} * m_i$
$\Rightarrow (x_1 m_1, \ldots, x_l m_1, y_1 m_2, \ldots, y_p m_2, z_1 m_i, \ldots, z_q m_i, \overline{(m_1 + \ldots + m_i)})/$
$(t_1, \ldots, t_l, u_1, \ldots, u_p, v_1, \ldots, v_q, k+1)$.

Now, statement $k+1$ is part of the control transfer code. If none of the conditions is true, control branches to $k+1$.

Lemma 4 *A memory conflict free switch with k inclusive conditions is reduced by $k+1$ Csteps.*

Proof: For a switch with k independent conditions, k different statements can be branched to. If none of the conditions is true then a branch to the following statement takes place. A total of $k+1$ branches are available. If none of the branches has memory conflict, then $k+1$ branches are merged into the switch. This results in reducing the code by $k+1$ CSteps. ∎

Lemma 5 *A switch with k inclusive conditions is reduced by at least 1 Cstep.*

Proof: As the conditions are independent, one or more branches can be taken at the same time. Moreover, all the branches can be taken when all conditions are true. In the extreme case of memory conflicts between all conditions, none of the branches is mergable. The reason is that conditions may all be true. If none of the conditions is true, the following statement is executed. As the condition of this statement is

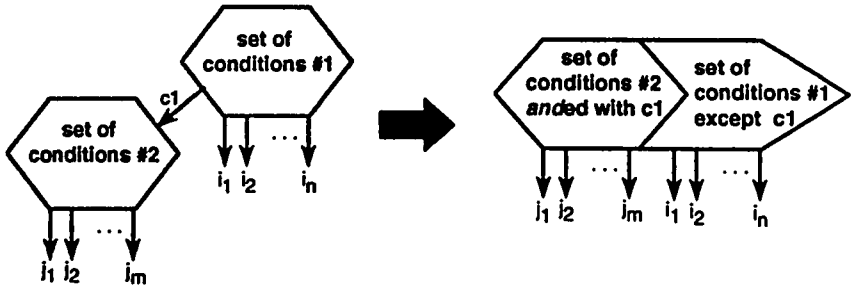


Figure 5.12: Cascade transformations at two level switches.

disjoint with the conditions of the branches, this statement is merged with one of the branches. Therefore, at least one CStep is saved. ∎

The switch transformation, in the case of inclusive conditions is more general than the case of mutual exclusive conditions. In fact, mutual exclusiveness is a special case of inclusive conditions.

The switch transformation is applied recursively. Therefore, a branch of a switch statement can also be another switch statement. For example, in the case of two-level switching, the branches in the second level are merged into the switch of the first level. A new condition set is formed from the two sets. The first set is the set of conditions in the first level switch excluding the condition that causes branching to the second level switch. The other set is formed by *and*ing the excluded condition with the set of conditions in that branch. This is illustrated in Figure 5.12.

The switch transformation algorithm is given in Figure 5.13. The algorithm is called by the main program once a statement with at least one conditional expression

```
Procedure Switch_Transformation(S_i)
  Begin_Procedure
     Form the set of conditions C
           C={c_1,c_2,...,c_j}; j≥ |Bout_i|
     FOR  each c_k ∈ C
       IF (S_k has no memory conflict with S_i) THEN
          Switch_Transformation(S_k)
          Merge S_k * c_k into S_i
       END_IF
     END_FOR
  End_Procedure
```

Figure 5.13: Algorithm for switch transformation.

is found. The set of conditions (C) in this statement is formed. For every condition $c_k \in C$, if the corresponding statement has no memory conflict, then the statement can be merged into the switch statement. The execution of the child statement is controlled by the truth value of c_k. The algorithm continues scanning AHPL statements and merging them whenever possible.

To illustrate how the algorithm works, consider the AHPL model for the bubble sort example after loop transformation. Two forks are identified in the flow chart of Figure 5.14(a). One in Step 2 and the other in Step 5. From Step 2, C contains $\{i > j, \sim (i > j)\}$. If the condition '$i > j$' is true, we proceed with Step 4. Step 3 is called when the condition '$\sim (i > j)$' is true. Step 4 has no memory conflict with Step 2. The procedure is recursively called until all conditional expressions in Step 4 are processed. Thus, Step 4 is merged into Step 2, conditional on '$i > j$'. Then

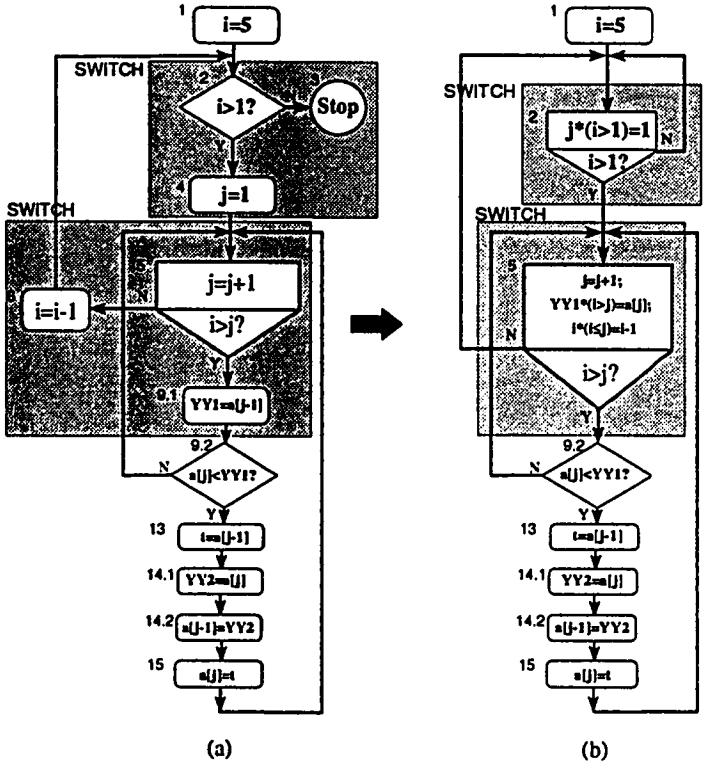


Figure 5.14: Flow chart illustrating switch transformation in bubble sort.

Step 3 is merged into Step 2. A dead end (stop) is achieved by a transfer back to the same state.

Applying the algorithm to the switch at Step 5, a conditional branch forks to Steps 6 and 9.1. Data transfers in these steps are converted to destination controlled conditional transfers in Step 5 of Figure 5.14(b). Therefore, Step 6 which contains the transfer '$i \Leftarrow i - 1$' is moved to Step 5 and is executed if '$i \leq j$'. The same transfer is applied to the transfer in Step 9.1. Figure 5.14(b) shows the flow chart of the bubble sort AHPL model after switch transformation.

To find the complexity of this transformation, let us assume a code with s switches. In the case of disjoint switches, the algorithm processes each switch individually and merges the corresponding branches if there is no memory conflict. Therefore, the transformation will be applied s times. Hence, the complexity is $O(s)$.

5.3 Illustrative examples

The transformation algorithms were explained and illustrated with the bubble sort example. In this section, we shall illustrate the entire synthesis process on two new examples. Initially the behavioral model is given; then, the translation step into BRPN and the generation of CRTL are shown. The first example is the greatest common divisor (gcd) benchmark test [DR92]. The algorithm for *gcd* is:

```
x=4
y=5
while(x! = y) {
      if(x < y) { y = y − x }
      if(x > y) { x = x − y }
}
```

The translation of the above *gcd* behavioral description will result in the following

BRPN intermediate form.

4sx
5sy
$[lylx - sy]S1$
$[lxly - sx]S2$
$[lxly > 1lxly < 2lxly! = 0]S0$
$lxly! = 0$
q.

The second step is to obtain the CRTL code from the BRPN description. The outcome of this step is the following:

```
(-1,0) 1: x=4
(-1,1) 2: y=5
(-1,2) 3: if y!=x then 5
(-1,3) 4: Deadend.
(0, 0) 5: if y>x then 9
(0, 1) 6: if y<x then 11
(0, 2)  7: if y!=x then 5
(0, 3)  8: goto 4
(1, 0)  9: y=y-x
(1, 1) 10: goto 6
(1, 2) 11: x=x-y
(1, 3) 12: goto 7
```

Then, following the application of software optimization, Steps 8, 10 and 12 which are unconditional branches are deleted. Steps 7, 9 and 11 are modified to branch to Steps 4, 6 and 7 respectively. Steps 3 and 7 are redundant. Thus, Step 7 is deleted and the code is modified. The partially optimized code is as follows:

```
(-1,0) 1: x=4
(-1,1) 2: y=5
(-1,2) 3: if y!=x then 5
```

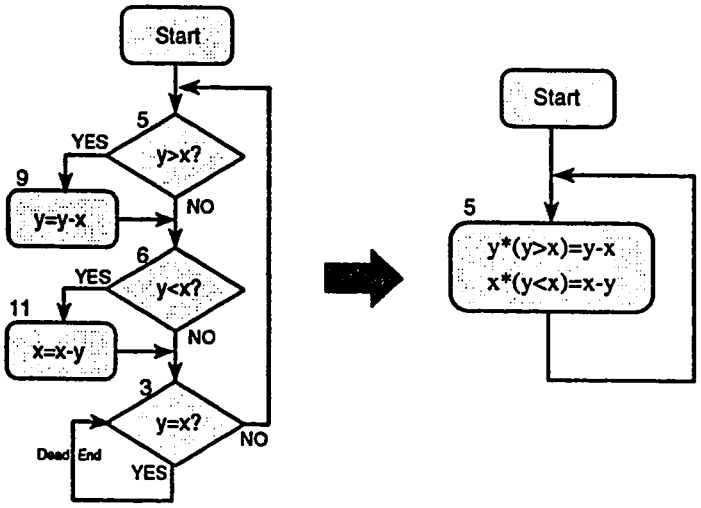


Figure 5.15: Switch transformation in the *gcd* example.

```
(-1,3) 4: Deadend.
(0, 0) 5: if y>x then 9
(0, 1) 6: if y<x then 11 else 3
(1, 0) 9: y=y-x; goto 6
(1, 2) 11: x=x-y; goto 3
```

The repetitive application of the switch transformation leads to the merging of Steps 3, 5 and 6 into one CStep. Meanwhile, the register transfer operation in Step 9 is translated into a destination controlled conditional transfer '$y*(y>x)=y-x$'. Similarly, Step 11 is transformed into '$x*(y<x)=x-y$'. Since there is no memory conflict in Steps 1 and 2, they are merged. Figure 5.15 shows these transformations. The *gcd* algorithm finishes when y and x are equal. Therefore, a *finish* signal is introduced. This signal is true only when $y=x$.

The optimized AHPL model for the *gcd* example is given below. All conditional

transfers are destination controlled transfers.

$$
\begin{array}{ll}
1 & x = 4; y = 5; => (5). \\
5 & y * (x > y) = y - x; x * (x < y) = x - y; \\
 & finish * (y = x) = \backslash 1 \backslash; => (5).
\end{array}
$$

A more complex example is the traffic light controller (*tlc*) benchmark test [DR92]. The high level description of *tlc* is:

```
current_state=newstate
case current_state of
     0 :  newHL=4; newFL=6;
          if (cars==1)and(timeoutL==1) then
             newstate=4; newST=1;
          else
             newstate=0; newST=0;
          end_if
     4 :  newHL=2; newFL=6;
          if (timeoutS==1) then
             newstate=0; newST=0;
          else
             newstate=6; newST=0;
          end_if
     2 :  newHL=6; newFL=4;
          if (cars==0)and(timeoutL==1) then
             newstate=6; newST=0;
          else
             newstate=2; newST=0;
          end_if
     6 :  newHL=6; newFL=2;
          if (timeoutS==1) then
             newstate=0; newST=1;
          else
             newstate=6; newST=0;
          end_if
     7 :  newstate=0; newHL=6;
           newFL=0; newST=0;
end case
```

The corresponding BRPN code is:

```
ln sc
[6 sn 0 st]S15
[0 sn 1 st]S14
[2 sn 0 st]S13
[6 sn 1 st]S12
[1 ll=12 1 ll<>13]S11
[6 sn 0 st]S10
[2 sn 1 st]S9
[0 sn 0 st]S8
[4 sn 1 st]S7
[1 ll=7 1 ll<>8]S6
[0 sn 0 sh 0 sf 0 st]S5
[6 sh 2 sf 1 ls=14 1 ls<>15]S4
[6 sh 4 sf 0 lr=12 0 lr<>11]S3
[2 sh 6 sf 1 ls=9  1 ls<>10]S2
[4 sh 6 sf 1 lr=6 1 lr<>8]S1
[0 lc=1 4 lc=2 2 lc=3 6 lc=4 7 lc=5]S0
q
```

The AHPL model extracted from the above BRPN code is a 63 AHPL state model. Due to the length of the AHPL code, its flow chart is illustrated in Figure 5.16. The switch transformation algorithm is run on the *tlc* AHPL model. A two CStep AHPL model is accomplished and is given in Figure 5.17.

5.4 Conclusion

In this chapter we described the optimization techniques that are used to produce an RTL description with a minimum number of states. Software optimization techniques such as common sub-expression and unconditional branch elimination, are

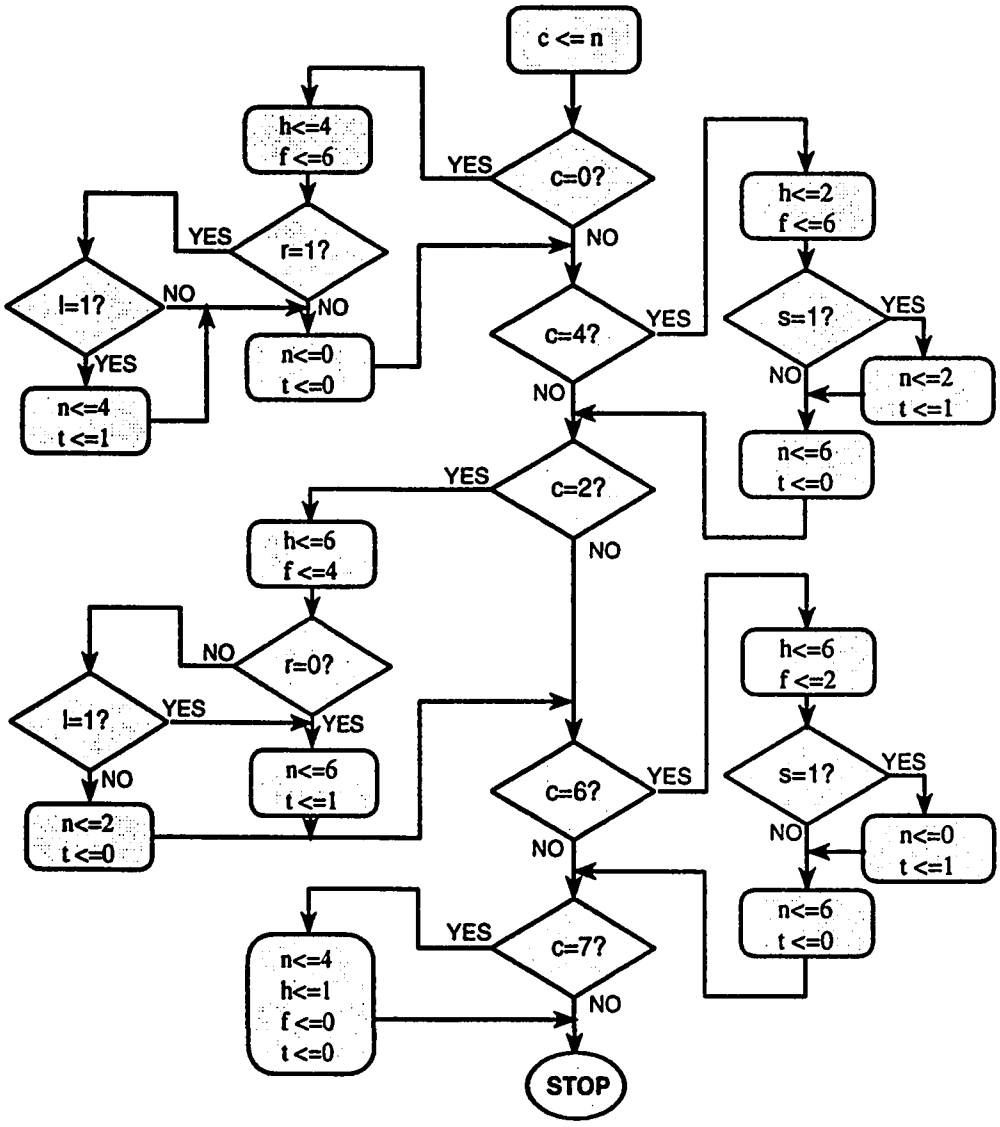


Figure 5.16: Flow chart representing the partially optimized AHPL machine of *tlc*.

used. The algorithm for these techniques are given. The application of these techniques was also demonstrated on several examples.

The novelty of this work is hardware specific optimization techniques. Two such techniques namely: loop transformation and switch transformation, are proposed. The algorithms developed for these transformations and their complexities are also presented.

The techniques described are illustrated with three examples, bubble sort, *gcd* and *tlc*. Optimized AHPL models are generated.

```
MODULE   : TLC.
MEMORY   : H{3}; F{3}; N{3}; C{8}; T; R; L; S.
BUSES  : C0; C2; C4; C6; C7.
EXINPUTS : RESET; CLK.
BODY SEQUENCE:CLK.
       1     C<=N.
       2     C0=~C{0}&~C{1}&~C{2};
     H<=\1,0,0\*C0; F<=\1,0,0\*C0;
     N<=\1,0,0\*(C0&R&L); T<=\1\*(C0&R&L);
     N<=\0,0,0\*(C0&~(R&L)); T<=\0\*(C0&~(R&L));

             C4=C{0}&~C{1}&~C{2};
     H<=\0,1,0\*C4; F<=\1,1,0\*C4;
     N<=\0,1,0\*(C4&S); S<=\1\*(C4&S);
     N<=\1,1,0\*(C4&~S); S<=\0\*(C4&~S);

             C2=~C{0}&C{1}&~C{2};
     H<=\1,1,0\*C2; F<=\1,0,0\*C2;
     N<=\1,1,0\*(C2&~R&L); T<=\1\*(C2&~R&L);
     N<=\0,1,0\*(C2&~(~R&L)); T<=\0\*(C2&~(~R&L));

             C6=C{0}&C{1}&~C{2};
     H<=\1,1,0\*C6; F<=\0,1,0\*C6;
     N<=\0,0,0\*(C6&S); T<=\1\*(C6&S);
     N<=\1,1,0\*(C6&~S); T<=\0\*(C6&~S);

             C7=C{0}&C{1}&C{2};
     H<=\0,0,0\*C7; F<=\0,0,0\*C7;
     T<=\0\*C7; N<=\0,0,0\*C7;
     => (1).
ENDSEQUENCE
CONTROLRESET(RESET)/(1).
END.
```

Figure 5.17: AHPL description of *tlc* after optimization.

Chapter 6

Analysis and Comparison

This chapter describes, analyzes, and compares the HLS methodology developed in this thesis with previously reported systems. This comparison will be carried using some benchmark tests from the 1992 High-level Synthesis Workshop [DR92] and other circuits. The rest of the chapter is organized as follows: Section 6.1 analyzes the different tasks in the HLS system presented in this work. Details of the comparison is given in Section 6.2. Section 6.3 concludes the chapter.

6.1 Analysis of HLS system

The HLS system proposed and developed in this work consists of three major tasks: BRPN translation, CRTL generation, and CRTL optimization. These tasks are

integrated as illustrated in Figure 3.1. The system was implemented in the C programming language.

The behavioral description of a digital system is first translated into BRPN. This behavior is modeled in a C-like description. An extended version of the **UNIX** *bc* utility is used to compile the input behavioral specification into BRPN. *bc* is an interactive arithmetic language processor that outputs an RPN representation. Other constructs such as *repeat-until*, *if-then-else*, and *case* are also translated into BRPN. Logical operators like *and* and *or* are mapped into BRPN. Systematic procedures to map these constructs and logical operators are implemented. The behavioral models of these constructs and their corresponding BRPN were presented in Figures 3.4, 3.5, 3.6, 3.8, 3.10 and 3.11.

The next task accomplished by this work is the generation of RTL from BRPN. The generated structural description is the Canonical RTL (CRTL), where each step is limited to one statement. This CRTL is a graph-based representation. A procedure to generate CRTL descriptions from BRPN descriptions has been implemented. The algorithm used in this task is presented in Figure 4.2. The complexity of the algorithm is $O(n)$; where n is the number of tokens in the BRPN code. The algorithm builds up a linear linked list data structure. The data structure created is shown in Figure 4.4. This data structure is developed for experimental purposes. More efficient data structures can be investigated.

The format of the CRTL code has a simple structure. It simply consists of statements. A statement can either be a data operation and an unconditional branch, or a conditional branch. Then several optimization techniques are performed in order to reduce the RTL description to a minimum number of control steps. The optimization strategy followed in this work consists of: *software optimization techniques* and *hardware specific optimization techniques.* Initially, compiler like optimization techniques are used. Two commonly used techniques are applied: unconditional branch elimination and common sub-expression (redundant code) elimination. The algorithms developed to accomplish these optimizations have a quadratic complexity in the number of AHPL statements.

In hardware specific optimization, two techniques are developed. These techniques rely on the hardware specific features. They are *loop transformation* and *switch transformation.* The algorithms developed for these transformations are shown in Figures 5.6 and 5.13. The complexity of the loop transformation algorithm is $O(n^2)$ where n is the number of statements in the code. While the complexity of the switch transformation algorithm is $O(s)$ where s is the number of switches in the code.

The final step in our HLS system is the generation of the AHPL code from the optimized CRTL. This AHPL code is fed to an AHPL silicon compiler to generate the physical description (layout). The VLSI layout of the control unit of the bubble sort circuit is given in Figure 6.1. The transistor level circuit is extracted from the

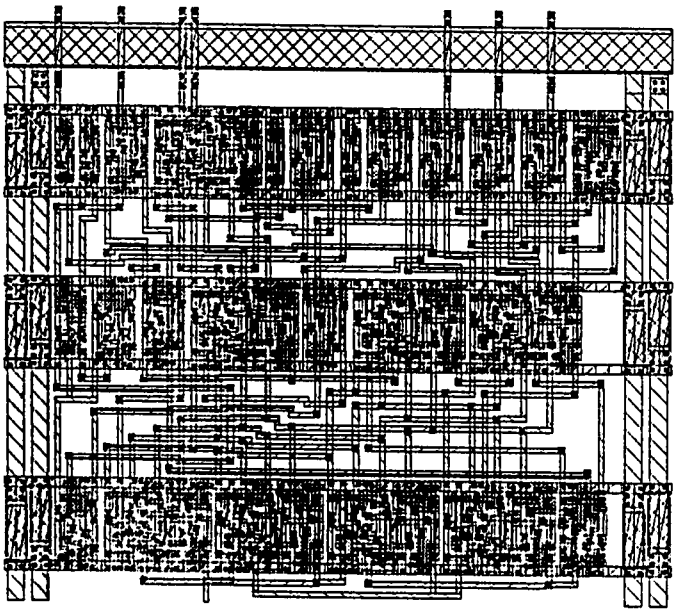


Figure 6.1: Layout of bubble sort control circuit.

layout and simulated to verify the correctness at layout level. The output of the circuit level simulator is shown in Figure 6.2.

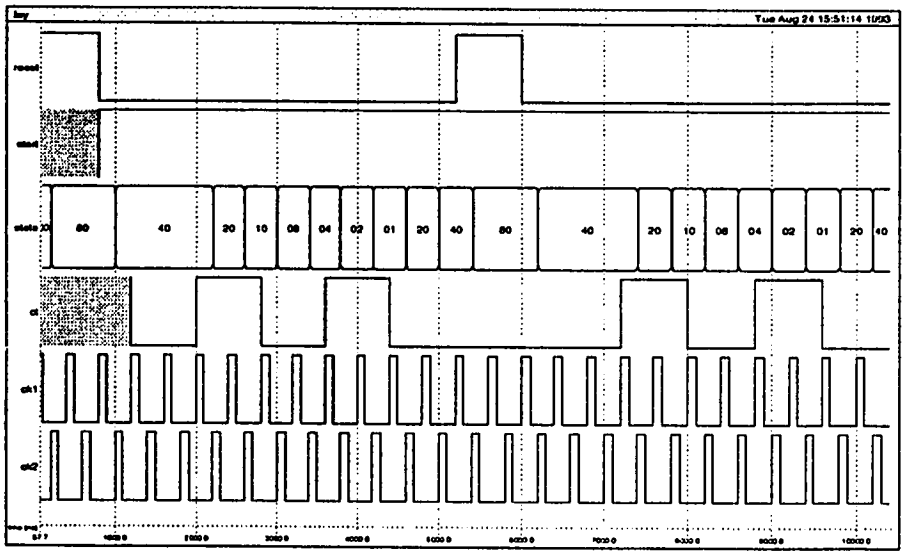


Figure 6.2: Transistor-level simulation results of bubble sort control circuit.

6.2 Comparison

The results of generating concise RTL code of digital systems modeled in HLL are presented here. Benchmarks from [DR92] are used. The benchmark circuits used are traffic light controller (*tlc*), greatest common divisor (*gcd*), and differential equation (*DiffEq*). The bubble sort (*BubSrt*) model is also used to study the situation of having memory conflicts in the AHPL code. The generated circuits are compared with those in [Cam91], [ORJ92], and [AS94]. The comparison is with respect to the number of CSteps required.

A digital system can be modelled as a finite state machine (FSM). While RTL models consist of control steps (CSteps). Each CStep may hold one or more states in the corresponding FSM. For example, a 3-bit incrementer is modeled with eight FSM states; while only one control step is required to model it in AHPL.

During optimization, transformation algorithms search for possible merging of CSteps among the unoptimized RTL statements. This optimization process reduces the number of CSteps as well as the number of logic components used in the control logic; the overall logic may also be minimized. In Table 6.1, the first column (*Before optimization*) shows the hardware resources consumed by the control logic in the unoptimized RTL description. The columns titled (*CSteps*) hold the number of control steps which correspond to the number of *flip-flops*. The consumed logic is given in the columns titled *Logic Units*. The logic components used are: 4-bit comparator (Comp), and some logic gates (And, Or). The second column (*After optimization*) shows the resources allocated based on the optimized code.

Circuit	*Before optimization*		*After optimization*	
	CSteps	Logic Units	CSteps	Logic Units
TLC	63	36And,17Or,3comp	2	1Or
GCD	9	8And,4Or,4comp	2	1Or
DiffEq	19	4And,2Or,2Comp	2	1Or
BubSrt	18	10And,4or,5comp	6	2And,3Or,3Comp

Table 6.1: Control logic in unoptimized and optimized RTL.

Software as well as hardware optimization techniques are performed on each

circuit. Exploiting hardware special features has resulted in more optimized models. This is illustrated in Table 6.2.

Circuit	*Unoptimized*	*Software*	*Hardware*
TLC	63	47	2
GCD	9	7	2
DiffEq	19	17	2
BubSrt	18	14	6

Table 6.2: The number of CSteps at different stages of the HLS system.

In the *tlc* benchmark test, the unoptimized code has 63 CSteps. The application of software optimization had reduced the number of CSteps in the code by 16 CSteps. While the application of hardware specific optimization had resulted in a 2-CStep AHPL model.

The results obtained in this work are compared with those reported in literature. In Table 6.3, the columns titled *Path*, *DLS*, and *LBS* give the number of states obtained by the scheduling algorithms reported in [Cam91], [ORJ92], and [AS94] respectively. The last column titled HSO in the table, holds the number of states obtained by the application of the optimization techniques developed in this work. The experimental results show a reduction in the number of states ranging from 50% for DiffEq to 75% for TLC. This reduction was mainly a consequence of hardware specific optimization.

Another measure to test the quality of the optimization techniques is to calculate

Circuit	*Path*	*DLS*	*LBS*	*HSO*
TLC	8	7	5	2
GCD	2	2	2	2
DiffEq	4	-	3	2
BubSrt	-	-	-	6

Table 6.3: Experimental vs literature results.

how much reduction is achieved in the total area of the chip. Table 6.4 illustrates this. The numbers in the first three columns are in λ^2. The figures in the first column

Circuit	*Unoptimized*	*Software*	*Hardware*	*-%*
TLC	653,632	562,832	223,644	65.4
GCD	971,152	960,480	598,096	38.4
DiffEq	2,712,544	2,602,112	2,486,112	8.3
BubSrt	1,813,312	1,586,848	1,427,728	21.3

Table 6.4: The areas of test circuits at different stages of the HLS system.

are the areas of the test circuits before any optimization is performed. The second column titled *Software* holds the areas after the application of software optimization. The (Hardware) column shows the areas of the optimized circuits. The relative percentage reduction in each model is shown in the last column (-%). For example, the total area of the optimized AHPL code for the TLC benchmark test is 223,644 λ^2. The area of the unoptimized model is 653,632 λ^2. This gives a percentage decrease in the total area by 65.8%.

In Section 5.2, we stated that the chance for the silicon compiler to perform logic optimization increases as more logic is combined in a single CStep. Table 6.5 gives

the percentage decrease in area at different stages of our HLS system. The figures

Circuit	*Unoptimized*	*Software*	*Hardware*
TLC	57.2%	57.7%	74.8%
GCD	13.6%	13.8%	16.8%
DiffEq	20.2%	20.1%	20.6%
BubSrt	13.1%	19.8%	22.8%

Table 6.5: The logic optimization performed by AHPL silicon compiler.

in each column present the percentage decrease in area when the logic optimization of AHPL silicon compiler is performed. For example, in the unoptimized *tlc* model, the AHPL silicon compiler accomplished 57.2% reduction in the total area. This reduction is due to logic optimization only. The application of software optimization has reduced the number of CSteps as discussed earlier. Consequently, the logic per CStep is increased. Therefore, the chance for the AHPL silicon compiler to reduce the circuit area also increased. As seen in the second column (*Software*) a 57.7% reduction is accomplished.

Finally, the application of hardware specific optimization has further reduced the number of CSteps. Thus more logic is combined into a single CStep. The AHPL silicon compiler was able to perform more logic optimization. A reduction of 74.8% in the circuit area was reached. This is mainly due to combining more logic in one CStep.

6.3 Conclusion

The experimental results were collected by running the proposed optimization techniques on some benchmarks. A reduction upto 75% in the number of control steps was reached. This is compared to the results of others work. The situation of having memory conflicts in AHPL code was also studied. The optimization techniques developed were able to resolve memory conflicts and reduce the number of control steps in the initial AHPL model.

Another quality measure of our HLS system was to calculate the reduction in the total area of the chip. Experimental results showed that a reduction in the chip area ranging from 8.3% to 65.4% was achieved. This is mainly due to exploiting hardware special features.

Moreover, we observe that as more logic is combined in a single CStep, the silicon compiler has a greater chance to perform logic optimization. Table 6.5 showed that the percentage reduction in the area increases as more CSteps were combined. And the most reduction was obtained after the application of hardware specific optimization.

Chapter 7

Conclusion and Future Work

High-level synthesis (HLS) is the process of translating an input behavioral specification of a digital system into hardware. In this work we presented a high-level synthesis system which accepts as input a behavioral specification in a subset of the C language and generates optimized RTL descriptions in the AHPL language. We introduced a new stack intermediate form expressed in blocked reverse polish notation (BRPN). BRPN serves as a primary intermediate form from which other internal representations are extracted. The simplicity of the stack abstract data type made the generation to/from BRPN very efficient and easy. The translation of behavioral models into BRPN is described in Chapter 3.

The target AHPL description is derived in two steps. First a Canonical RTL (CRTL) description, where each step is limited to one statement, is extracted from

the BRPN. Then, in the second step, the CRTL is optimized to produce a compact AHPL description with minimum number of CSteps. In Chapter 4, the generation of CRTL from BRPN and the extraction of AHPL models were described.

The main tasks in HLS systems are: scheduling and allocation. In scheduling, operations are assigned to control steps. The objective in scheduling is to minimize the time required for program completion.

In this work, software optimization techniques are used. Unconditional branch elimination and code factorization are two techniques applied. Section 5.1 discussed these software optimization techniques.

Besides the well known compiler-like optimization techniques, our scheduler exploits the hardware specific features of the AHPL language to perform suitable optimization, yielding a schedule with minimum number of CSteps. Experiments on benchmark tests show sizeable reduction in the number of CSteps compared to other reported systems.

The optimization techniques developed in this work were tested. The comparison was carried using benchmark circuits. The comparison results were given in Chapter 6.

In this research, we were concerned more with the quality of results rather than the quality of tools and techniques developed. Though all algorithms have low

polynomial complexity, they can still be improved and refined. The places where improvement can take place are discussed next.

The interface between the different steps in the proposed HLS system is not yet automated. Therefore, these steps can be interfaced together to automate the propagation from one step to the other.

The behavioral description language used is an extended version of *UNIX* utility called *bc*. *bc* resembles C language but it does not define some HLL constructs such as *Case* construct. Therefore, a new behavioral description language can be developed for this purpose. Otherwise, it is sufficient to enhance *bc* to define other HLL constructs and logical operators.

As shown in Chapter 4, the internal data structure developed in this work uses two types of links. One type is used to keep track of the physical proximity of the statements. The other type is for proper transfer of control. This is because our intention was to develop a data structure for experimental purposes. More efficient data structures can be implemented using one type of link.

In this work, we exploited some specific hardware characteristics. Other hardware characteristics can be investigated and exploited. The algorithms proposed for hardware specific optimization can be enhanced to provide global optimization. Moreover, the goal of the techniques developed is to minimize the number of control steps. However, new techniques exploiting the hardware special attributes such as

parallelism and pipelining, can be developed to optimize the logic within the control step (data path logic).

In current implementation, we assume unlimited hardware resources and a single port memory. Upgrades to allow the specification of resource constraints can easily be accommodated; instead of checking for memory conflicts only, one also has to check for other resource conflicts.

Appendix A

Grammar of the C-like HLL

The translation of HLL descriptions to BRPN is a language to language translation. The HLL description language used in this HLS system is a C-like language. This language is a subset of C language. The definition of the language in BNF is given in this appendix.

```
program     :  statement

statement   :  expression
            |  IF '(' expression ')' statement
            |  IF '(' expression ')' statement ELSE statement
            |  WHILE '(' expression ')' statement
            |  FOR '(' expression ';' expression ';' expression ')'
               statement
            |  REPEAT statement UNTIL '(' expression ')' ';'
            |  CASE '(' expression ')' ':' statement
            |  identifier ':' statement

expression  :  binary { ';' binary}

binary      :  identifier '=' binary
            |  identifier '+=' binary
            |  identifier '-=' binary
            |  identifier '*=' binary
            |  identifier '/=' binary
            |  identifier '%=' binary
            |  identifier '^=' binary
            |  binary '==' binary
```

```
                | binary '<=' binary
                | binary '>=' binary
                | binary '!=' binary
                | binary '<' binary
                | binary '>' binary
                | binary '+' binary
                | binary '-' binary
                | binary '*' binary
                | binary '/' binary
                | binary '%' binary
                | binary '^' binary
                | unary

unary           : '++' identifier
                | '--' identifier
                | primary

primary         : identifier
                | constant
                | '(' expression ')'
```

Bibliography

[AS94] Hassan F. Al-Sukhni. A C-based High-Level Synthesis System. *Master Thesis, KFUPM*, January 1994.

[Cam91] Raul Camposano. Path-Based Scheduling for Synthesis. *IEEE Transaction on CAD*, 10(1):85–93, January 1991.

[CBH+91] R. Camposano, R.A. Bergamaschi, C.E. Haynes, M. Payer, and S.M. Wu. The IBM High-level Synthesis System. *In High-Level VLSI Synthesis, Editors Raul Camposano and Wayne Wolf, Kluwer Academic Publishers*, pages 79–104, 1991.

[CR89] Raul Camposano and Wolfgang Rosenstiel. Synthesizing Circuits From Behavioral Descriptions. *IEEE Transaction on CAD*, 8(2):171–180, February 1989.

[DN89] Srinivas Devadas and Richard Newton. Algorithms for Hardware Allocation in Data Path Synthesis. *IEEE Transaction on CAD*, 8(7):768–781, July 1989.

[DR92] Nikil Dutt and Champaka Ramachandran. Benchmarks for the 1992 High Level Synthesis Workshop. *Technical Report no. 92-107*, October 1992.

[HL91] Y. Hsu and Y. Lin. High Level Synthesis in the THEDA System. *In High-Level VLSI Synthesis, Editors Raul Camposano and Wayne Wolf, Kluwer Academic Publishers*, pages 283–306, 1991.

[KM91] David Ku and Giovanni De Micheli. Synthesis of ASICs with Hercules and Hebe. *In High-Level VLSI Synthesis, Editors Raul Camposano and Wayne Wolf, Kluwer Academic Publishers*, pages 177–203, 1991.

[KM92] D. Ku and G. De Micheli. Relative Scheduling Under Timing Constraints: Algorithms for High-Level Synthesis of Digital Circuits. *IEEE Transaction on CAD*, 11(6):696–718, June 1992.

[Mas81] M. Masud. Modular Implementation of a Digital Hardware Automation System. *Ph.D dissertation*, Department of EE, University of Arizona, 1981.

[MPC90] Michael C. McFarland, Alice C. Parker, and Paul Camposano. The High-Level Synthesis of Digital Systems. *IEEE proc.*, 78(2):301–318, February 1990.

[NON91] Y. Nakamura, K. Oguri, and A. Nagoya. Synthesis from Pure Behavioral Descriptions. *In High-Level VLSI Synthesis, Editors Raul Camposano and Wayne Wolf, Kluwer Academic Publishers*, pages 205–229, 1991.

[ORJ92] K. O'Brien, M. Rahmouni, and A. A. Jerraya. A VHDL-Based Scheduling Algorithm For Control-Flow Dominated Circuits. *IMAG/TIM3 Technical Report*, 1992.

[Par84] Alice C. Parker. Automated Synthesis of Digital Systems. *IEEE Design and Test*, pages 75–81, November 1984.

[Pau91] P. Paulin. Global Scheduling and Allocation Algorithms in the HAL System. *In High-Level VLSI Synthesis, Editors Raul Camposano and Wayne Wolf, Kluwer Academic Publishers*, pages 255–281, 1991.

[PG87] Barry Michel Pangrle and Daniel D. Gajski. Design Tools for Intelligent Silicon Compilation. *IEEE Transaction on CAD*, CAD-6(6):1098–1112, November 1987.

[PK89a] Pierre G. Paulin and John P. Knight. Algorithms for High-Level Synthesis. *IEEE Design and Test of Computers*, pages 18–31, December 1989.

[PK89b] Pierre G. Paulin and John P. Knight. Force-Directed Scheduling for the Behavioral synthesis of ASIC's. *IEEE Transaction on CAD*, 8(6):661–679, June 1989.

[PKPW91] A. Parker, K. Kucukcakar, S. Prakash, and J. Weng. Unified System Construction (USC). *In High-Level VLSI Synthesis, Editors Raul Camposano and Wayne Wolf, Kluwer Academic Publishers*, pages 331–354, 1991.

[Sai92] Sadiq M. Sait. Integrating UAHPL-DA System with VLSI Design Tools to Support VLSI DA Courses. *IEEE Transaction on Education*, 35(4):321–330, November 1992.

[SBK93] Sadiq M. Sait, Muhammad S. Benten, and Asjad M. Khan. ASIC Design from UAHPL Models. *ICM proceedings*, pages 237–144, December 1993.

[TH86] Fur-Shing Tsai and Yu-Chin Hsu. STAR: An Automatic Data Path Allocator. *IEEE Transaction on CAD*, 11(9):1053–1064, September 1986.

[TKK89] Toshiaki Tanka, Tsutomu Kobayashi, and Osamu Karatsu. HARP: Fortran to Silicon. *IEEE Transaction on CAD*, 8(6):649–660, June 1989.

[Tri87] Howard Trickey. Flamel: A High-Level Hardware Compiler. *IEEE Transaction on CAD*, CAD-6(2):259–269, March 1987.

[TS86] Chia-Jeng Tseng and Daniel P. Siewoirk. Automated Synthesis of Data path in Digital Systems. *IEEE Transaction on CAD*, CAD-5(3):379–395, July 1986.

[Wak91] Kazutoshi Wakabayashi. Cyber: High-level Synthesis System from Software into ASIC. *In High-Level VLSI Synthesis, Editors Raul Camposano and Wayne Wolf, Kluwer Academic Publishers*, pages 127–151, 1991.

[WTL91] W. Wolf, A. Takach, and T. Lee. Archetictural Optimization Methods for Control-Dominated Machines. *In High-Level VLSI Synthesis, Editors Raul Camposano and Wayne Wolf, Kluwer Academic Publishers*, pages 231–254, 1991.

Vita

Abdulaziz Sultan Al-Mulhem was born in Muharaq, Bahrain, on December 16, 1967. After graduating from Na'eem Secondary School, Bahrain, he attended King Fahd University of Petroleum and Minerals at Dhahran, Saudi Arabia. He received the degree of Bachelor of Science with highest honors in Computer Engineering in June 1990. In 1990, he was employed in the Computer Engineering Department of the College of Computer Science and Engineering at the same university as a Graduate Assistant. He received the degree of Master of Science in Computer Engineering from the same university in June 1994.