**ΠΑΝΕΠΙΣΤΗΜΙΟ ΔΥΤΙΚΗΣ ΜΑΚΕΔΟΝΙΑΣ**

**ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ**

**ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ**

# UPGRADE OF A C TO ADA REASEARCH COMPILER

**ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ**

του

**ΜΑΡΓΑΡΙΤΗ ΔΟΥΛΑΚΗ**

**(MARGARITIS DOULAKIS)**

(AEM: 2228 )

**Επιβλέπων :** **Δρ. Μιχαήλ Δόσης (Dr. Michael Dossis)**

**Καθηγητής (Professor)**

Καστοριά April - 2023 (παρουσίασης της εργασίας)

Η παρούσα σελίδα σκοπίμως παραμένει λευκή

**ΠΑΝΕΠΙΣΤΗΜΙΟ ΔΥΤΙΚΗΣ ΜΑΚΕΔΟΝΙΑΣ**

**ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ**

**ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ**

# ΑΝΑΒΑΘΜΙΣΗ ΕΡΕΥΝΗΤΙΚΟΥ ΜΕΤΑΦΡΑΣΤΗ C ΣΕ ADA

**ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ**

του

**ΜΑΡΓΑΡΙΤΗ ΔΟΥΛΑΚΗ**

(ΑΕΜ: 2228 )

**Επιβλέπων :** **Δρ. Μιχαήλ Δόσης**

**Καθηγητής**

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 25/04/2023

| | | |
|---|---|---|
| ..... | ..... | ..... |
| Δόσης Μιχαήλ | Δημόκας Νικόλαος | Ευάγγελος Καρβούνης |
| Ιδίοτητα Μέλους | Ιδίοτητα Μέλους | Ιδίοτητα Μέλους |

Καστοριά Απρίλιος - 2023 (παρουσίασης της εργασίας

# Ευχαριστίες (Acknowledgments)

Especially to Dr. Michael Dosis and Dr. Georgios Dimitriou for providing me with support and an opportunity to work on this project.

To all my professors, people whose material I used and everyone who has supported me throughout the undergraduate course and whose shoulders I have been standing on, including Dr. Dosis and Dr. Dimitriou as well as collaborators such as Apostolos (Tolis) Tsakiridis and Georgios (Giorgos) Chatzianastasiou (among others).

# Περίληψη

Η επίτευξη παραλληλισμού σε επίπεδο εντολών (ΠσΕΕ, ILP) σε υλικό που έχει την δυνατότητα να τον εκμεταλλευτεί μπορεί, θεωρητικά, να βελτιώσει την απόδοση σε μεγάλο βαθμό. Μερικές βελτιστοποιήσεις, συμπεριλαμβανομένης της “ξεδιπλώματος βρόγχων” (loop unrolling) έχουν την δυνατότητα βελτίωσης του ΠσΕΕ και με τον συνδυασμό της με την βελτιστοποίηση των “συμπιεσμένων εκφράσεων” (expression compression) μπορεί να γίνει πιο αποτελεσματική χρήση πόρων ενός ολοκληρωμένου κυκλώματος. Για την εργασία αυτή τροποποιήθηκε ένας υπάρχων μεταγλωττιστής της γλώσσας C στη γλώσσα Ada (που ονομάζεται csense, μέρος των εργαλείων CCC (”Custom Coprocessor Complier suite”)) ώστε να υποστηρίζει πλήρες ξεδίπλωμα βρόγχων με διαφύλλωση και αναδιάταξη εντολών (που μπορεί να βελτιώσει την διαθεσιμότητα δεδομένων στην κρυφή μνήμη). Επίσης ολοκληρώθηκε η υλοποίηση συμπιεσμένων εκφράσεων με την δυνατότητα συνδυασμού και επιλογής επιπέδων ακέραιων και Boolean εκφράσεων.

***Λέξεις Κλειδιά: μεταφραστής/μεταγλωττιστής πηγαίου προς πηγαίου κώδικα, μεταγλωττιστής βελτιστοποίησης, C, Ada, πλήρες ξεδίπλωμα βρόγχων, διαφύλλωση/αναδιάταξη εντολών, παραλληλισμός σε επίπεδο εντολών (ΠσΕΕ/ΠΕΕ – ILP), διαθεσιμότητα δεδομένων στην κρυφή μνήμη, συμπίεση εκφράσεων, σύνθεση υψηλού επιπέδου.***

# Abstract

Achieving Instruction level parallelism (ILP) on hardware capable of exploiting it can greatly improve performance, in theory. Some optimizations, including loop unrolling, can improve ILP, and combined with the expression compression optimization can improve efficiency in resource usage in an integrated circuit. For this paper an existing C to Ada compiler (csense, part of the CCC (”Custom Coprocessor Complier”) suite) was modified to implement full loop unrolling in its output with a capability to interleave and reorder instructions (which can potentially improve data locality). Additionally, the expression compression implementation with combined and controllable levels of integer and Boolean expressions was completed.

***Key Words: source to source compiler, transpiler, optimizing compiler, C, Ada, full loop unrolling, instruction interleaving/reordering, instruction level parallelism (ILP), data locality, expression compression, high level synthesis (HLS)***

# Πίνακας Περιεχομένων (Table of contents)

*ΠΡΟΣΟΧΗ: Ο Πίνακας Περιεχομένων θα πρέπει να δημιουργείται αυτόματα (από το πρότυπο του επεξεργαστή Κειμένου με παράθεση όλων των Στυλ Επικεφαλίδων που χρησιμοποιήσατε (με εμφάνιση των αριθμών σελίδων δεξιά, διαχωριζόμενες με στηλοθέτη ….. από τον τίτλο έκαστης Επικεφαλίδας)*

# Λίστα Εικόνων (List of images)

*ΠΡΟΣΟΧΗ: Η Λίστα Εικόνων (ή Λίστα Σχημάτων) θα πρέπει να δημιουργείται αυτόματα (από το πρότυπο εισαγωγής Πίνακα Εικόνων του Επεξεργαστή Κειμένου με παράθεση όλων των Λεζαντών Εικόνων (ή Σχημάτων) που δημιουργήσατε **κάτω από** καθεμία Εικόνα (ή Σχήμα) της εργασίας σας.*

*Στη Λίστα Εικόνων (ή Λίστα Σχημάτων) παρατίθενται όλες οι Λεζάντες Εικόνων (ή Σχημάτων) με εμφάνιση των αριθμών σελίδων δεξιά, διαχωριζόμενες με στηλοθέτη ….. από τον τίτλο έκαστης Λεζάντας*

# Λίστα Πινάκων (List of Tables)

*ΠΡΟΣΟΧΗ: Η Λίστα Πινάκων θα πρέπει να δημιουργείται αυτόματα (από το πρότυπο εισαγωγής Πίνακα του Επεξεργαστή Κειμένου με παράθεση όλων των Λεζαντών Πινάκων που δημιουργήσατε,* ***πάνω από*** *καθένα Πίνακα της εργασίας σας.*

*Στη Λίστα Πινάκων παρατίθενται όλες οι Λεζάντες Πινάκων με εμφάνιση των αριθμών σελίδων δεξιά, διαχωριζόμενες με στηλοθέτη ….. από τον τίτλο έκαστης Λεζάντας.*

# Introduction

The first chapter gives an outline of what the subject of this paper is.

The second chapter explains key concepts needed to understand what was done for this paper, and what we are trying to accomplish. Section 2.1 expands on the structure of a typical compiler and the concepts one needs to know about in order to understand some terminology. Section 2.2 describes the C and Ada languages. Section 2.3 describes some optimizations, including loop unrolling and expression compression, which are the focus of this paper. Section 2.4 contains some theory on parallel computing and performance; what can theoretically be achieved by applying some of the optimizations.

The third chapter describes the csense C to Ada compiler, its capabilities, and an analysis of the relevant function which was modified to implement full loop unrolling and interleaving of instructions , as well as expression compression.

The fourth chapter contains benchmarks, including a description of the setup and the test code.

The fifth chapter contains related work, other projects and papers trying to solve problems similar to the ones in this paper.

# 1. The subject of this paper

High level synthesis (HLS) tools are important tools for the design of integrated circuits (IC) such as ASICs, or for programming FPGAs. The performance of such ICs strongly depends on optimizations of the HDL code that generates them. The benefits of optimizing range from more efficient memory usage and instruction processing time to power consumption efficiency, which might be achieved, for instance, by parallelization and elimination of redundant circuit generation respectively.

Depending on the structure of a HLS tool, it might consist of a frontend, which translates from a high level language, and a backend which generates the HDL from the output of the frontend. The CCC compiler suite falls into this category. The backend of the CCC compiler suite is used to produce optimized VHDL or Verilog code. The input for the backend is, conventionally, Ada code which must however adhere to a certain set of standards and syntax. These standards are defined in [1]. The frontend can translate C code to Ada that adheres to the said standards and is ideal for the backend. C code acceptable by the frontend is defined in [2]. The frontend can also apply optimizations to the Ada code which are expected to improve the ability of the backend to generate HDL code with its own optimizations and desired traits of the output.

Two such optimizations are full loop unrolling and expression compression (described in section 2.3).

The subject of this thesis is the effects of applying the full loop unrolling and expression compression optimizations to the CCC frontend's Ada output which in turn affects the backend's VHDL output. Although unrolling increases the states in the VHDL code produced by the CCC backend, when combined with expression compression we expect to see a decrease with the help of the PARCS optimizer which compresses the states in the output [3] [4]. This results in greater opportunities for parallelism.

When true full unrolling is applied, the loop is eliminated entirely, effectively removing the cost associated with the loop control instruction and loop test instructions.

By controlling the level of expression depth we can fine-tune the output of the HDL for the target implementation constraints i.e. capability for expressions of integer, boolean or a combination of the two's simultaneous execution.

The implementation of the full loop unrolling optimization required modifying the loop unrolling function to alter the abstract syntax tree (AST) appropriately to

remove the loop structure and reorder the instructions in the candidate loops, with all appropriate checks to make sure the requested optimizations can be applied (if they could not be, then fallback measures are taken in an attempt to apply the most possible). Options for combinations of loop unrolling optimizations (simple, full, full with reordering of instructions or simple with reordering of instructions) for specified candidate loops were also added, with a possibility to apply a final loop unrolling option to all loop candidates that were not specified.

The implementation of the expression compression optimization for combined boolean and integer instructions of a requested level of depth for each, required modifying the appropriate functions and altering the way the AST of one instruction is decomposed into separate instructions as requested by the user. Modifications to the print_expression() function which outputs the final code (for the frontend) were also required, as well as generating the appropriate cc files as required by the standards of Dr. Dossis ([3]) as of 09/2022.

# 2. Key Concepts

This section will attempt to give a briefing of the concepts dealt with throughout this paper.

## 2.1 Compilers

Given by [5], a compiler is "computer software that translates (compiles) source code written in a high-level language (e.g., C++) into a set of machine-language instructions that can be understood by a digital computer's CPU. Compilers are very large programs, with error-checking and other abilities. Some compilers translate high-level language into an intermediate assembly language, which is then translated (assembled) into machine code by an assembly program or assembler. Other compilers generate machine language directly. The term compiler was coined by American computer scientist Grace Hopper, who designed one of the first compilers in the early 1950s."

A more general definition is given by [6]: "A compiler is the software whose goal is to translate a program from one language to another. We say that the program is written in source language in its initial form and the compiler produces the equivalent program in the target language"

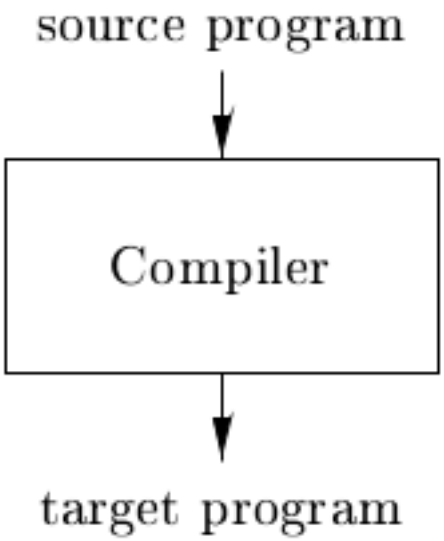


**Image 1. A compiler translates the source program to a target program**

Πηγή: **[7]**

### 2.1.1 Programming language

[8] gives a set of definitions, that help us understand where the term “programming language” fits between the written representation of an algorithm and its relation to following a set of rules that comply to a spec:

1. A “computational model” is a collection of values and operations.
2. A “computation” is the application of a sequence of operations to a value to yield another value.
3. A “program” is a specification of a computation.
4. A “programming language” is a notation for writing programs.
5. The “syntax” of a programming language refers to the structure or form of programs.
6. The “semantics” of a programming language describe the relationship between a program and the model of computation.
7. The “pragmatics” of a programming language describe the degree of success with which a programming language meets its goals both in its faithfulness to the underlying model of computation and in its utility for human programmers.

#### 2.1.1.1 Algorithms and their representation throughout history (leading up to high level programming languages and compilers)

“The earliest known written algorithms come from ancient Mesopotamia, about 2000 B.C. In this case the written descriptions contained only sequences of calculations on particular sets of data, not an abstract statement of the procedure; it is clear that strict procedures were being followed but they never seem to have been written down. By the time of Greek civilization, several nontrivial abstract algorithms had been studied rather thoroughly. The description of algorithms was always informal however, rendered in natural language.

Programs written for early computing devices such as those for Babbage’s Calculating Engine were naturally presented in machine language rather than a true programming language. Thus, the three-address code for Babbage’s machine was to consist of instructions such as “$V_4 \times V_0 = V_{10}$” where operation signs like “x” would appear on an Operation-card, and subscript numbers like (4, 0, 10) would appear on a separate Variable-card.”

The first “high level” programming language actually to be implemented was the Short Code, originally suggested by John W. Mauchly in 1949 for the UNIVAC.

Corrado Böhm developed the first practical compiler for his PhD thesis in 1951.

The first implemented compiler (in the definition we know today) was Alick E. Glennie's AUTOCODE, which came into use in 1952 for the Manchester Mark I.

The term "compiler" itself was introduced by Grace Murray Hopper in 1951: "To compile means to compose out of materials of other documents. Therefore, the compiler method of automatic programming consists of assembling and organizing a program from programs or routines or in general from sequences of computer code which have been made up previously.", although this definition is not used today. [9]

### 2.1.2 A compilers' general structure

A compiler maps a source program into a semantically equivalent target program. There are two parts to this mapping; – analysis and synthesis:

"The analysis part breaks up the source program into constituent pieces and imposes a grammatical structure on them. It then uses this structure to create an intermediate representation of the source program. If the analysis part detects that the source program is either syntactically ill formed or semantically unsound, then it must provide informative messages, so the user can take corrective action. The analysis part also collects information about the source program and stores it in a data structure called a symbol table , which is passed along with the intermediate representation to the synthesis part. The synthesis part constructs the desired target program from the intermediate representation and the information in the symbol table. The analysis part is often called the front end of the compiler; the synthesis part is the back end. If we examine the compilation process in more detail, we see that it operates as a sequence of phases , each of which transforms one representation of the source program to another. A typical decomposition of a compiler into phases is shown in image 2.

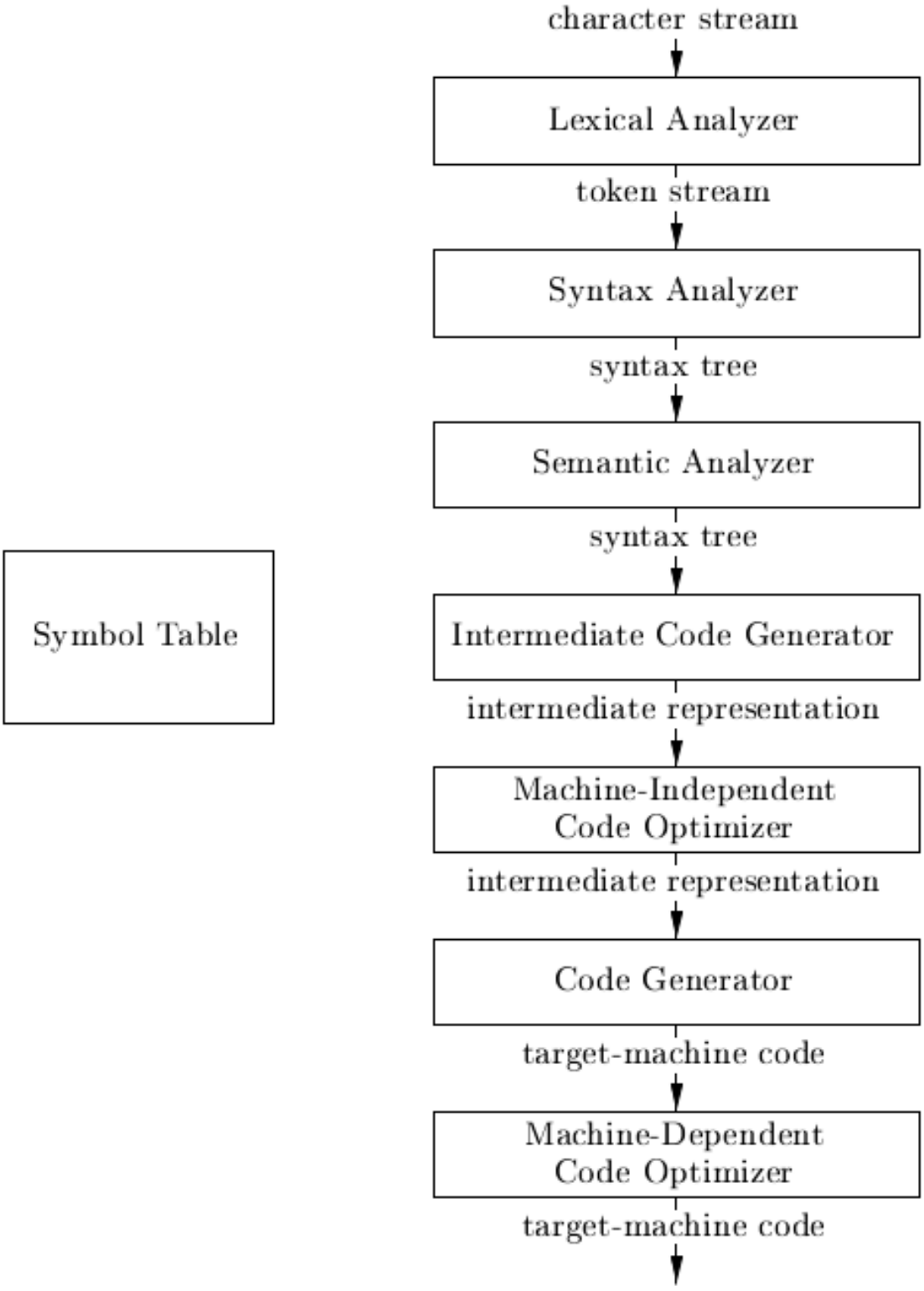


**Image 2. The phases of a compiler**

Πηγή: **[7]**

In practice, several phases may be grouped together, and the intermediate representations between the grouped phases need not be constructed explicitly. The symbol table, which stores information about the entire source program, is used by all phases of the compiler.

Some compilers have a machine-independent optimization phase between the front end and the back end. The purpose of this optimization phase is to perform transformations on the intermediate representation, so that the back end can produce a better target program than it would have otherwise produced from an unoptimized intermediate representation. Since optimization is optional, one or the other of the two optimization phases shown in image 2 may be missing.” [7]

Below is an elaboration of some of the phases that are relevant to this document.

#### 2.1.2.1 Lexical analysis (The first part of which is called “Scanning”)

“In this phase, the compiler receives a stream of characters as input: the source code. The goal of lexical analysis is to categorize the characters into groups with a particular meaning in accordance with the definition of the source language. These groups of characters are called “Lexemes” and each one of these corresponds to some token.

Examples of lexemes are reserved words (“if”, “while”, etc), identifiers (alphanumeric character strings provided by the user), special symbols that might consist of one or more characters (such as “<”, “<=”, etc), among others.”

Lexemes can be captured using regular expressions and finite automata. [6]Error: Reference source not found

#### 2.1.2.2 Syntax analysis (parsing)

In this phase the compiler receives the lexemes as input and executes the syntax analysis, where the structure of the program is checked. Error checking is also performed at this stage. The structural components of the program are located and their relations are set. The syntax of a language is described using a set of context free grammar rules. The result of the syntax analysis is a tree, called a syntax tree. [6]

“A grammar naturally describes the hierarchical structure of most programming language constructs.”

“A context-free grammar has four components:

1. A set of terminal symbols, sometimes referred to as "tokens." The terminals are the elementary symbols of the language defined by the grammar.

2. A set of non-terminals, sometimes called "syntactic variables." Each non-terminal represents a set of strings of terminals.

3. A set of productions, where each production consists of a non-terminal, called the head or left side of the production, an arrow, and a sequence of terminals and/or non-terminals, called the body or right side of the production. The intuitive intent of a production is to specify one of the written forms of a construct; if the head non-terminal represents a construct, then the body represents a written form of the construct.

4. A designation of one of the non-terminals as the start symbol.” [7]

#### 2.1.2.2.1 Backus Naur Form (BNF)

"The syntax rules for the language can be written down in Backus-Naur Form (BNF) or a similar notation."

"A BNF grammar is a list of syntax rules."

"Each rule defines one "nonterminal symbol", which appears at the left of a ":=" sign in the rule." "(this makes BNF grammars which are called "context free" grammars)"

"Alternative definitions of the nonterminal symbol appear to the right of the ":=" sign, separated by "|" signs."

"Often the definition of a nonterminal in a BNF grammar is recursive: it defines the nonterminal in terms of itself."

"The nonterminal symbol defined in rule listed first in the grammar is called the "start" symbol of the grammar."

"A symbol not defined by a rule in the grammar is a "terminal symbol", and is usually taken literally."

"If a string satisfies the definition of the "start" symbol, it is in the language defined by the BNF grammar; otherwise not." [10]

Below is an example of BNF for a very simple calculator:

"

<exp> ::= <exp> + <term> | <exp> - <term> | <term>

<term> ::= <term> * <power> | <term> / <power> | <power>

<power> ::= <factor> ^ <power> | <factor>

<factor> ::= ( <exp> ) | <int>

<int> ::= <digit> <int> | <digit>

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

" [11]

#### 2.1.2.2.2 LR Parsing (“bottom up” parsing)

LR parsing is a method and order by which grammar rules are matched to a given input.

"A bottom-up parse corresponds to the construction of a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top)." [7] [p233]

"The most prevalent type of bottom-up parser today is based on a concept called LR(k) parsing; the "L" is for left-to-right scanning of the input, the "R" for constructing a rightmost derivation in reverse, and the k for the number of input symbols of lookahead that are used in making parsing decisions." [7] [p241]

The Basic idea is that the LR parser has a stack (LIFO (last in first out) data structure) and input. Given contents of stack and k tokens, a look-ahead parser does one of following operations:

Shift: move the first input token to the top of the stack

or

Reduce: the top of the stack matches a rule. [12]

Given the example input "(3+4)+(5+6)" and the rules

"

E := int

E := (E)

E := E + E

"

LR parsing would conduct these steps (in the order they are presented here):

Shift ( on to stack ["("]

Shift 3 on to stack ["(3"]

Reduce using rule E := int ["(E"]

Shift + on to stack ["(E+"]

Shift 4 on to stack ["(E+4"]

Reduce using rule E := int ["(E+E"]

Reduce using rule E := E + E ["(E"]

Shift ) on to stack ["(E)"]

Reduce using rule E := (E) ["E"]

Shift + on to stack ["E+"]

Shift ( on to stack ["E+("]

Shift 5 on to stack ["E+(5"]

Reduce using rule E := int ["E+(E"]

Shift + on to stack ["E+(E+"]

Shift 6 on to stack ["E+(E+6"]

Reduce using rule E := int ["E+(E+E"]

Reduce using rule E := E + E ["E+(E"]

Shift ) on to stack ["E+(E)"]

Reduce using rule E := (E) ["E+E"]

Reduce using rule E := E + E ["E"]

[12]

#### 2.1.2.3 Semantic analysis

This stage is for the calculation of additional information and to conduct checks that can't be defined in the scope of a context free grammar. The processing here conforms to the to the rules that ensure the correctness and the seamless execution of the source code, but do not describe the syntax of the language. It is based on the information that is available during the compilation, and due to this we say it is related to the static semantics of the program. [6]

"The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition. It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation. An important part of semantic analysis is type checking, where the compiler checks that each operator has matching operands. For example, many programming language definitions require an array index to be an integer; the compiler must report an error if a floating-point number is used to index an array. The language specification may permit some type conversions called coercions. For example, a binary arithmetic operator may be applied to either a pair of integers or to a pair of floating-point numbers. If the operator is applied to a floating-point number and an integer, the compiler may convert or coerce the integer into a floating-point number." [7]

The most important function for languages with static analysis demands (such as C) is the symbol table. This table records information that is related to the meaning of names and type checking, among other things (such as scope rules and visibility, flow control checks). [6]

#### 2.1.2.4 Intermediate code generation

The term "Intermediate code" refers to the data structure that preserves the source code during its translation. Generally, the intermediate code can express the functionality of the commands on a level as high as a syntax tree, or it can be more resemblant of the code of the target language. It may also use the characteristics of the

machine or the execution environment of the program (for example the size of the data types, the positions of the variables and the availability of registers.) Moreover, the intermediate code may or may not integrate information stemming from the symbol table such as scopes and field nesting levels, among others.

The intermediate code is especially useful for the optimization of the final program. For this particular processing the requirement of additional data structure use is likely, with information stemming from other processing phases. Optimizations are easier to be implemented in the intermediate code level. The intermediate code might also be the key to developing a portable compiler. This is achieved when the intermediate code does not depend on the executing machine. [6]

#### 2.1.2.4.1 Abstract Syntax Trees (ASTs)

“In an abstract syntax tree for an expression, each interior node represents an operator; the children of the node represent the operands of the operator. More generally, any programming construct can be handled by making up an operator for the construct and treating as operands the semantically meaningful components of that construct.

Abstract syntax trees, or simply syntax trees, resemble parse trees to an extent. However, in the syntax tree, interior nodes represent programming constructs while in the parse tree, the interior nodes represent non-terminals. Many non-terminals of a grammar represent programming constructs, but others are "helpers" of one sort of another, such as those representing terms, factors, or other variations of expressions. In the syntax tree, these helpers typically are not needed and are hence dropped.” [7]Error: Reference source not found

"ASTs represent only semantically meaningful aspects of input program, unlike concrete syntax trees which record the complete textual form of the input. There’s no need to record keywords or punctuation like (), ;, else." [13]

"A syntax-tree node representing an expression E1 + E2 has label + and two children representing the subexpressions E1 and E2." [7]

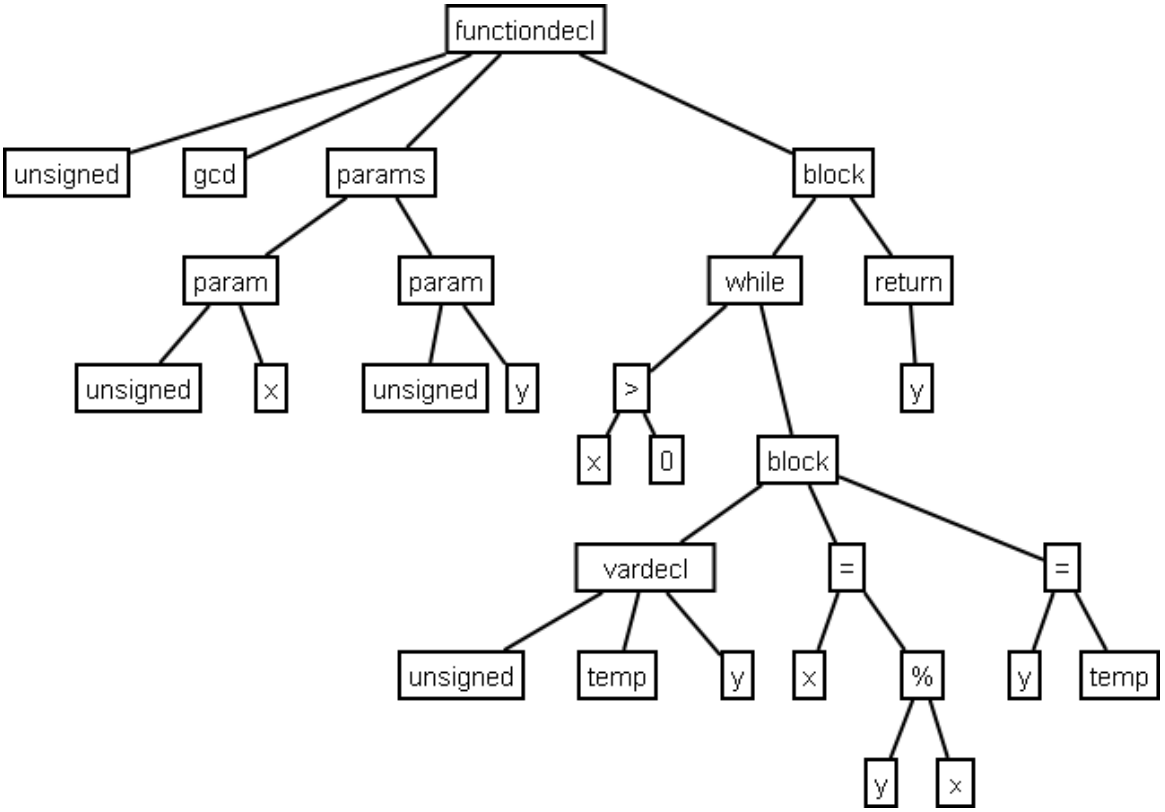


**Image 3.** An example of an AST

Πηγή: [14]

#### 2.1.2.4.2 Directed Acyclic Graphs (DAGs)

"A directed acyclic graph (DAG) for an expression identifies the common subexpressions (subexpressions that occur more than once) of the expression." [7]

"Like the syntax tree for an expression, a DAG has leaves corresponding to atomic operands and interior nodes corresponding to operators. The difference is that a node N in a DAG has more than one parent if N represents a common subexpression; in a syntax tree, the tree for the common subexpression would be replicated as many times as the subexpression appears in the original expression. Thus, a DAG not only represents expressions more succinctly, it gives the compiler important clues regarding the generation of efficient code to evaluate the expressions." [7]

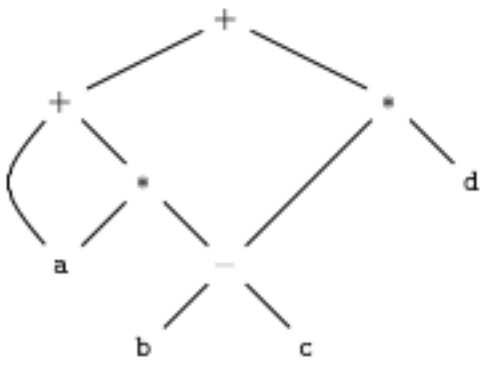


Figure 6.3: Dag for the expression a + a * (b - c) + (b - c) * d

**Image 4. An example of a DAG**

Πηγή: [7]

#### 2.1.2.5 Code optimization

Each compiler is different not only in relation to the optimizing capabilities it has, but also the compiler phases in which it conducts them. It is clear that the first optimizations are implemented after the semantic analysis and are concerned with improvements in the level of the source code. [6]

“The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result. Usually better means faster, but other objectives may be desired, such as shorter code, or target code that consumes less power.” [7]

One such case is constant collapsing, where operations on constants are replaced with their final value. Optimizations of this type can be immediately integrated in the syntax tree, with the collapsing of a section to just one node. Other optimizations that can be implemented at the intermediate code level are the propagation of assignment operations to the rest of the program, removal of inaccessible sections of the code, the replacement of computationally expensive expressions by others, renaming of variables, the elimination of common expressions, the movement of code outside of loops wherever possible and the simplification of variables whose values change within loops. [6]

“There is a great variation in the amount of code optimization different compilers perform. In those that do the most, the so-called "optimizing compilers," a significant amount of time is spent on this phase. There are simple optimizations that significantly improve the running time of the target program without slowing down compilation too much.”Error: Reference source not found [7]

#### 2.1.2.6 Code generation

In this phase the intermediate representation of the source code which has resulted from the processing of the previous phases is used to create code in the target language. [6]

“The code generator takes as input an intermediate representation of the source program and maps it into the target language. If the target language is machine code, registers or memory locations are selected for each of the variables used by the program. Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task. A crucial aspect of code generation is the judicious assignment of registers to hold variables.” [7]

“The requirements imposed on a code generator are severe. The target program must preserve the semantic meaning of the source program and be of high quality; that is, it must make effective use of the available resources of the target machine. Moreover, the code generator itself must run efficiently.” [7]

“Compilers that need to produce efficient target programs, include an optimization phase prior to code generation. The optimizer maps the IR into IR from which more efficient code can be generated. In general, the code-optimization and code-generation phases of a compiler, often referred to as the back end, may make multiple passes over the IR before generating the target program.” [7]

The immediate synthesis of machine code is time consuming and a procedure very prone to errors, as it depends on the accurate determination of a large number of digital forms. As such, the symbolic language of the target processor for which the code will be generated is usually selected as the target language. When a program in its final form is formulated in a symbolic language, then an appropriate symbol translator can be used to turn it into an executable. Languages such as C and C++ are sometimes used as target languages due to the broad dissemination of their modern specification compliant compilers. [6]Error: Reference source not found

### 2.1.3 Source-to-source Compilers (or “Transpilers”)

“A conventional compiler consumes source code and produces binaries. A source-to-source compiler produces transformed source code from the original source. This transformation can be e.g. refactoring, parallelization or translation to a different language. The advantage is that the resulting code can be modified by the programmer if desired and compiled with a compiler of choice.” [15]

“Suppose you've written a program in one language but wish to convert this to another language, then you would invoke what's called a transpiler. The programming language at the input to the transpiler may be called the source language whereas the language at the output may be called the target language. A transpiler is sometimes called a source-to-source compiler.

For example, converting C++ code to C code will involve a transpiler. Converting Python code to Ruby code will involve a transpiler. Let's note that in these examples both source and target languages are at the same level of abstraction. But let's say we convert Java code to bytecode, or C code to assembly code, then this is not called transpilation. Transpilers don't change the level of abstraction such as translating from a high-level language to assembly code, bytecode or machine code.

[...]

A transpiler [...] usually works at the abstraction of high-level languages. The output code is still human readable. It cannot be executed directly unless its own compiler or interpreter is invoked. For example, a transpiler can convert Java code to C++ code.

Programmer will still need to invoke a C++ compiler before executing the resultant machine code.

It's acceptable to call translations at the same level of abstraction as transpilation, such as from .ASM assembly code to .A86 assembly code.

ISO/IEC/IEEE 24765 defines five generations of languages that can be related to abstractions described above.

[...]

Given this understanding (of the syntax rules of the input language), the transpiler builds what is called an Abstract Syntax Tree (AST).

The next step is to transform the AST to suit the target language. This is then used to generate code in the target language.” [16]

## 2.2 The C and Ada languages

In December 2021 the TIOBE index ranked C as #2 and Ada as #30. [17]

Redmonk Q3(June) 2021 rankings rank C as #10. Ada is not on the list. [18]

### 2.2.1 C

Applications of C include Operating systems (Unix, Windows, Android), Embedded systems and drivers, Graphical user interface (GUI) applications such as Adobe Photoshop, Illustrator and Premiere, programming platforms such as Matlab and Mathematica, Google Chromium, Mozilla Firefox and Thunderbird, and MySQL. Other uses include compilers such as MINGW and Clang C and some gaming and animation related projects. [19] [20]

#### 2.2.1.1 History and development of C

“C is a general-purpose programming language developed by Dennis Ritchie at Bell Laboratories in 1972. Since then, it has become a major language not only at Bell Labs but also throughout the world. C was originally developed for use with the Unix [a trademark of Bell Labs] operating system, which is largely written in C, so part of the success of the language is due to the acceptance of Unix. C, however, has spread far beyond Unix systems in the past few years, and a booming compiler industry has sprung up around it.

C was originally designed for "systems programming," that is, for writing programs like compilers, operating systems, and text editors. It has proven satisfactory for other applications as well, including data base systems, telephone switching systems,

numerical analysis, engineering programs, and a great deal of text-processing software." [21]

"C has its roots in the language BCPL, which is a "typeless" language that operates only on a single data type, the machine word. As such, BCPL is an excellent match to the hardware of word-oriented machines such as the PDP-10. In 1970, Ken Thompson designed a stripped-down version of BCPL for use with the first Unix system on the PDP-7. This stripped-down version is called B, and like BCPL is typeless.

With the advent of the PDP-11, upon which the next version of Unix was written, it became clear that a typeless language did not match the hardware's capabilities. The PDP-11 provided several fundamental objects of different sizes— 1-byte characters, 2-byte integers, and 4-byte floating point numbers. B provided no way to talk about these different sizes, let alone operators to manipulate them.

The C language was an attempt to deal with a variety of types, which it did by adding the notion of data type to the B language. In C, as in most languages, each object has a type as well as a value. The type determines what kinds of machine operations can be applied to the value and how much storage is occupied.

Although C was originally implemented for a PDP-11, it is not particularly tied to that machine, and around 1975 work began at Bell Labs on C compilers for other machines. In particular, a "portable compiler" was implemented, which made it relatively easy to modify the compiler to generate code for different machines.

As the Unix operating system spread, the technology of the portable compiler made it possible to move the operating system and its programs from one kind of hardware to another with little work. This meant that C became more widely available and used.

In the late 1970s, C became available from commercial sources, as well as from Bell Labs, for microprocessors like the Z80. This marked the beginning of C's commercial success. Since then, it has been implemented on many computers, from the smallest microprocessors to machines as large as the Cray-1; compilers are available from dozens of suppliers. The C language is sufficiently well standardized that with some care, it is possible to write C programs that will run without change on any machine that supports the standard language and the standard run-time environment." [21]

#### 2.2.1.2 C's syntax

For an elaboration of C's syntax see appendix A.

### 2.2.2 Ada

"Ada's philosophy is different from most other languages. Underlying Ada's design are principles that include the following:

Readability is more important than conciseness. Syntactically this shows through the fact that keywords are preferred to symbols, that no keyword is an abbreviation, etc.

Very strong typing. It is very easy to introduce new types in Ada, with the benefit of preventing data usage errors.

It is similar to many functional languages in that regard, except that the programmer has to be much more explicit about typing in Ada, because there is almost no type inference.

Explicit is better than implicit. Although this is a Python7 commandment, Ada takes it way further than any language we know of:

– There is mostly no structural typing, and most types need to be explicitly named by the programmer.

– As previously said, there is mostly no type inference.

– Semantics are very well defined, and undefined behavior is limited to an absolute minimum.

– The programmer can generally give a lot of information about what their program means to the compiler (and other programmers). This allows the compiler to be extremely helpful (read: strict) with the programmer." [22]

#### 2.2.2.1 Ada's history and use-cases

"In the 1970s the United States Department of Defense (DOD) suffered from an explosion of the number of programming languages, with different projects using different and non-standard dialects or language subsets / supersets. The DOD decided to solve this problem by issuing a request for proposals for a common, modern programming language. The winning proposal was one submitted by Jean Ichbiah from CII Honeywell-Bull.

The first Ada standard was issued in 1983; it was subsequently revised and enhanced in 1995, 2005 and 2012, with each revision bringing useful new features." [22]

"Today, Ada is heavily used in embedded real-time systems, many of which are safety critical. While Ada is and can be used as a general-purpose language, it will really shine in low-level applications:

Embedded systems with low memory requirements (no garbage collector allowed).

Direct interfacing with hardware.

Soft or hard real-time systems.

Low-level systems programming.

Specific domains seeing Ada usage include Aerospace & Defense, civil aviation, rail, and many others. These applications require a high degree of safety: a software defect is not just an annoyance, but may have severe consequences. Ada provides safety features that detect defects at an early stage — usually at compilation time or using static

analysis tools. Ada can also be used to create applications in a variety of other areas, such as:

Video game programming

Real-time audio

Kernel modules” [22]

#### 2.2.2.2 Ada’s syntax

Ada’s syntax is elaborated in appendix A.

## 2.3 Compiler optimizations

Programmers generally write clean, high level programs and usually do not write optimal code. Optimal code can depend on features not expressed to the programmer. Modern architectures assume optimization. [23] "[Optimization] is the concept of program transformation, to make it consume fewer resources such as CPU and memory. This will result in more efficient machine code and therefore a better performing program.

Generally, we can give our compiler optimization the following rules:

The optimization must not in any way change the function or meaning of the program

The optimization should increase the performance of the program

The compilation time must remain reasonable

The optimization must not delay compilation

Since our code optimization reduces code readability, it is often conducted at the end of development." [24]

"Different kinds of optimizations [can affect]:

Time: improve execution speed

Space: reduce amount of memory needed

Power: lower power consumption (e.g. to extend battery life)" [23]

It should be noted that in some cases optimizations might lead to security flaws (although this arguably violates rule #1 from [24]). [25] So, ideally, optimizations should go through some program analysis to determine if the transformation really is safe, but also to determine whether the transformation is cost effective. Typically there is no

guarantee that the transformations will improve performance, nor that compilation will produce optimal code. [23]

"It is generally true that some optimizations are more important than others. Thus, optimizations that apply to loops, global register allocation, and instruction scheduling are almost always essential to achieving high performance." [26]

"For almost every optimization or set of optimizations, we can easily construct a program for which they have significant value and only they apply." [26]

[26] categorizes "the intraprocedural (or global) optimizations covered in [its] Chapters 12 through 18 (excluding trace and percolation scheduling) into four groups, numbered I through IV, with group I being the most important and group IV the least.

Group I consists mostly of optimizations that operate on loops, but also includes

several that are important for almost all programs on most systems, such as constant folding, global register allocation, and instruction scheduling. Group I consists of

1. constant folding;

2. algebraic simplifications and reassociation;

3. global value numbering;

4. sparse conditional constant propagation;

5. the pair consisting of common-subexpression elimination and loop-invariant code motion or the single method of partial-redundancy elimination;

6. strength reduction;

7. removal of induction variables and linear-function test replacement;

8. dead-code elimination;

9. unreachable-code elimination (a control-flow optimization);

10. graph-coloring register allocation;

11. software pipelining, with loop unrolling, variable expansion, register renaming, and hierarchical reduction; and

12. branch and basic-block (list) scheduling.

Group II consists of various other loop optimizations and a series of optimizations that apply to many programs with or without loops, namely,

1. local and global copy propagation,

2. leaf-routine optimization,

3. machine idioms and instruction combining,

4. branch optimizations and loop inversion,

5. unnecessary bounds-checking elimination, and

6. branch prediction.

Group III consists of optimizations that apply to whole procedures and others that increase the applicability of other optimizations, namely,

1. procedure integration,

2. tail-call optimization and tail-recursion elimination,

3. in-line expansion,

4. shrink wrapping,

5. scalar replacement of aggregates, and

6. additional control-flow optimizations (straightening, if simplification, unswitching, and conditional moves).

Finally, group IV consists of optimizations that save code space but generally do not save time, namely,

1. code hoisting and

2. tail merging." [26]

A list of compiler optimizations and a brief description of them can be found at [27]. There are too many to include in this document in detail, so only some of the most popular/noteworthy/important ones are expanded on in appendix B (Most from group I). The ones that the CCC frontend applies are expanded on in section 2.3.2.

### 2.3.1 Data flow analysis

Some optimizations require data flow analysis, and others do not.

""Data-flow analysis" refers to a body of techniques that derive information about the flow of data along program execution paths. For example, one way to implement global common subexpression elimination requires us to determine whether two textually identical expressions evaluate to the same value along any possible execution path of the program." [7]

"The execution of a program can be viewed as a series of transformations of the program state, which consists of the values of all the variables in the

program, including those associated with stack frames below the top of the run-time stack. Each execution of an intermediate-code statement transforms an input state to a new output state. The input state is associated with the program point before the statement and the output state is associated with the program point after the statement.

When we analyze the behavior of a program, we must consider all the possible sequences of program points ("paths") through a flow graph that the program execution can take. We then extract, from the possible program states at each point, the information we need for the particular data-flow analysis problem we want to solve. In more complex analyses, we must consider paths that jump among the flow graphs for various procedures, as calls and returns are executed." [7]

"The purpose of data-flow analysis is to provide global information about how a procedure (or a larger segment of a program) manipulates its data.

For example, constant-propagation analysis seeks to determine whether all assignments to a particular variable that may provide the value of that variable at some particular point necessarily give it the same constant value. If so, a use of the variable at that point can be replaced by the constant.

The spectrum of possible data-flow analyses ranges from abstract execution of a procedure, which might determine, for example, that it computes the factorial function, to much simpler and easier analyses such as the reaching definitions problem.

In all cases, we must be certain that a data-flow analysis gives us information that does not misrepresent what the procedure being analyzed does, in the sense that it must not tell us that a transformation of the code is safe to perform that, in fact, is not safe. We must guarantee this by careful design of the data-flow equations and by being sure that the solution to them that we compute is, if not an exact representation of the procedure's manipulation of its data, at least a conservative approximation of it. For example, for the reaching definitions problem, where we determine what definitions of variables may reach a particular use, the analysis must not tell us that no definitions reach a particular use if there are some that may. The analysis is conservative if it may give us a larger set of reaching definitions than it might if it could produce the minimal result." [26]

[28] provides a simple example for a constant propagation data flow analysis. The example algorithm to be analyzed is:

"

```
k = 2;
if (...) {
  a = k + 2;
  x = 5;
} else {
  a = k * 2;
  x = 8;
}
k = a;
while (...) {
  b = 2;
  x = a + k;
  y = a * b;
  k++;
}
print(a+x);
```

" [28]

"The goal of constant propagation is to determine where in the program variables are guaranteed to have constant values. More specifically, the information computed for each CFG (control flow graph) node n is a set of pairs, each of the form (variable, value). If we have the pair (x, v) at node n, that means that x is guaranteed to have value v whenever n is reached during program execution.

Below is the CFG for the example program, annotated with constant-propagation information. " [28]

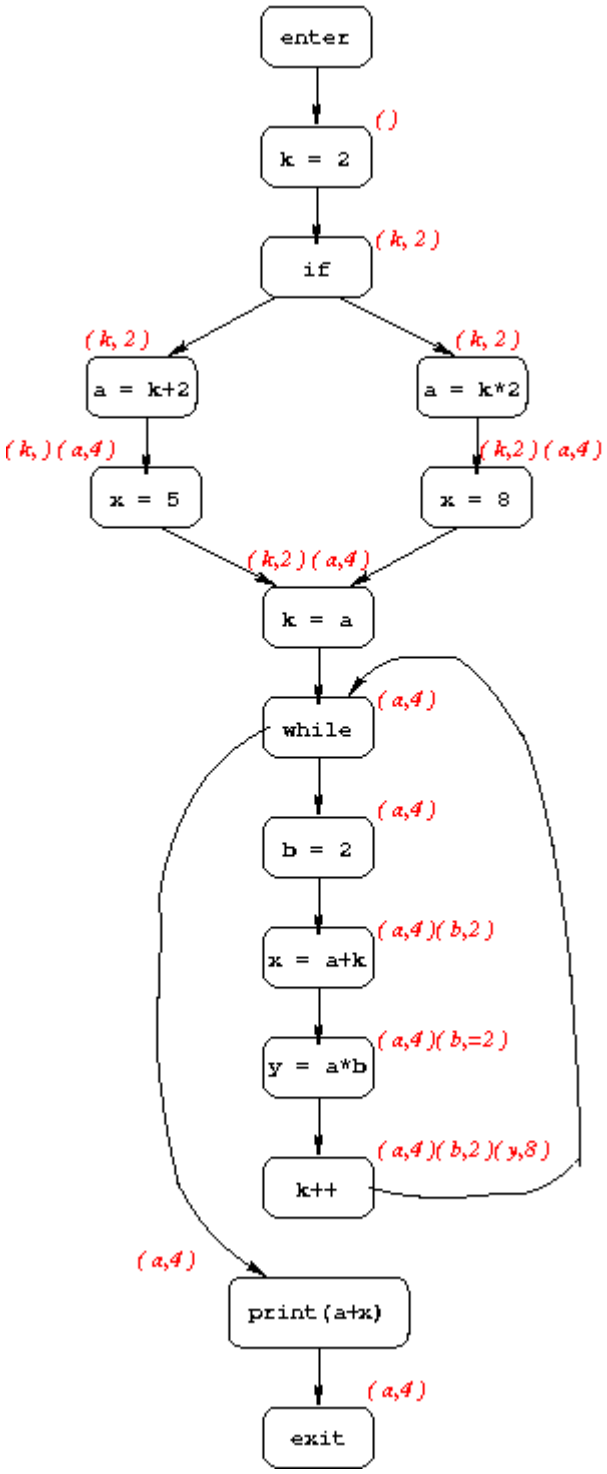


**Image 5.** **Data flow analysis constant propagation control flow graph example**

Πηγή: **[28]**

### 2.3.2 Description of optimizations relevant to the CCC frontend

(For other important optimizations not implemented by the CCC frontend see appendix B.)

#### 2.3.2.1 Constant folding

"Constant-expression evaluation, or constant folding, refers to the evaluation at compile time of expressions whose operands are known to be constant. It is a relatively simple transformation to perform, in most cases. In its simplest form, constant-expression evaluation involves determining that all the operands in an expression are constant-valued, performing the evaluation of the expression at compile time, and replacing the expression by its value. For Boolean values, this optimization is always applicable.

For integers, it is almost always applicable—the exceptions are cases that would produce run-time exceptions if they were executed, such as divisions by zero and overflows in languages whose semantics require overflow detection. Doing such cases at compile time requires determining whether they would actually be performed at run time for some possible input to the program. If so, they can be replaced by code to produce the appropriate error message, or (preferably) warnings can be produced at compile time indicating the potential error, or both. For the special case of addressing arithmetic, constant-expression evaluation is always worthwhile and safe—overflows do not matter." [26]

"For floating-point values, the situation is more complicated. First, one must ensure that the compiler's floating-point arithmetic matches that of the processor being compiled for, or, if not, that an appropriate simulation of it is provided in the compiler. Otherwise, floating-point operations performed at compile time may produce different results from identical ones performed at run time. Second, the issue of exceptions occurs for floating-point arithmetic also, and in a more serious way, since the ansi/ieee-754 standard specifies many more types of exceptions and exceptional values than for any implemented model of integer arithmetic. The possible cases—including infinities, NaNs, denormalized values, and the various exceptions that may occur—need to be taken into account. Anyone considering implementing constant-expression evaluation for floating-point values in an optimizer would be well advised to read the ansi/ieee-754 1985 standard and Goldberg's explication of it very carefully ( [29])" [26]

Example:

input:

"

int x = (2+3) * y;

b & false;

" [19]

output:

"

int x = 5 * y;

false;

" [23]

#### 2.3.2.2 Algebraic Simplification

Algebraic simplification can be seen as a more general form of constant folding. It takes advantage of mathematically sound simplification rules. [23]

"Algebraic simplifications use algebraic properties of operators or particular operator-operand combinations to simplify expressions. Reassociation refers to using specific algebraic properties—namely, associativity, commutativity, and distributivity—to divide an expression into parts that are constant, loop-invariant (i.e., have the same value for each iteration of a loop), and variable." [26]

"The most obvious algebraic simplifications involve combining a binary operator with an operand that is the algebraic identity element for the operator or with an operand that always yields a constant, independent of the value of the other operand.

For example, for any integer-valued constant or variable i, the following are always true:

i + 0 = 0 + i = i - 0 = i

0 - i = -i

i * 1 = 1 * i = i / 1 = i

i * 0 = 0 * i = 0

There are also simplifications that apply to unary operators, or to combinations of unary and binary operators, such as

- (- i) = i

i + (-j) = i - j

Similar simplifications apply to Boolean and bit-field types. For b, a Boolean valued constant or variable, we have

b V true = true V b = true

b V false = false V b = b

and corresponding rules for &." [26]

"Another class of simplifications involves the use of commutativity and associativity. For example, for integer variables i and j ,

(i - j) + (i - j) + (i - j) + (i - j) = 4 * i - 4 * j

except that we may incur spurious overflows in the simplified form. For example, on a 32-bit system, if i = 230 = 0x40000000 and j = 230 - 1 = 0x3fffffff , then the expression on the left evaluates to 4 without incurring any overflows, while that on the right also evaluates to 4, but incurs two overflows, one for each multiplication. Whether the overflows matter or not depends on the source language—in C or Fortran 77 they don't, while in Ada they do. It is essential that the optimizer implementer be aware of such issues." [26]

#### 2.3.2.3 Common Subexpression Elimination

"An occurrence of an expression E is called a common subexpression if E was previously computed and the values of the variables in E have not changed since the previous computation. We avoid recomputing E if we can use its previously computed value; that is, the variable x to which the previous computation of E was assigned has not changed in the interim. If x has changed, it may still be possible to reuse the computation of E if we assign its value to a new variable y, as well as to x, and use the value of y in place of a recomputation of E." [7]

The idea is to replace an expression with previously stored evaluations of that expression. [23]

Example:

input:

"

i = x + y + 1;

j = x + y;

" [27]

output:

"

t1 = x + y;

i = t1 + 1;

j = t1;

" [27]

For safety, we must be sure that the shared expression always has the same value in both places. [23]

"[CSE Elimination] almost always improves performance. But sometimes, it might be less expensive to recompute an expression, rather than to allocate another register to hold its value (or to store it in memory and later reload it)." [23]

#### 2.3.2.4 Code Motion (Hoisting)

"Loops are a very important place for optimizations, especially the inner loops where programs tend to spend the bulk of their time. The running time of a program may be improved if we decrease the number of instructions in an inner loop, even if we increase the amount of code outside that loop." [7]

"Loop-invariant code motion recognizes computations in loops that produce the same value on every iteration of the loop and moves them out of the loop. Note that if a computation occurs inside a nested loop, it may produce the same value for every iteration of the inner loop(s) for each particular iteration of the outer loop(s), but different values for different iterations of the outer loop(s). Such a computation will be moved out of the inner loop(s), but not the outer one(s).

Many, but not all, of the most important instances of loop-invariant code are addressing computations that access elements of arrays, which are not exposed to view and, hence, to optimization until we have translated a program to an intermediate code or to a lower-level one." [26]

Example 1:

input:

"

while (b) {

z = y/x;

… // y, x not updated

}

" [23]

output:

"

z = y/x;

while (b) {

… // y, x not updated

}

" [23]

This is not limited only to the body of the loop, but its expression as well.

Example 2:

input:

"

while (i <= limit-2)

    /* statement does not change limit */

" [7]

output:

"

t = limit-2

while (i <= t)

    /* statement does not change limit or t */

" [7]

#### 2.3.2.5 Loop Unrolling

"Loop unrolling replaces the body of a loop by several copies of the body and adjusts the loop-control code accordingly. The number of copies is called the unrolling factor and the original loop is called the rolled loop." [26]

"Unrolling reduces the overhead of executing an indexed loop and may improve the effectiveness of other optimizations, such as common-subexpression elimination, induction-variable optimizations, instruction scheduling, and software pipelining." [26]

Example:

input:

"

```
for(int i=0; i<100; i=i+1) {
    s = s + a[i];
}
```

" [23]

output:

"

```
for(int i=0; i<99; i=i+2){
    s = s + a[i];
    s = s + a[i+1];
}
```

" [23]

"The unrolled loop executes the loop-closing test and branch half as many times as the original loop and may increase the effectiveness of instruction scheduling by, e.g., making two loads of a[i] values available to be scheduled in each iteration. On the other hand, the unrolled version is larger than the rolled version, so it may impact the effectiveness of the instruction cache, possibly negating the improvement gained from unrolling. Such concerns dictate that we exercise caution in deciding which loops to unroll and by what unrolling factors." [26]

"Also, notice that we have oversimplified the unrolling transformation in the example: we have assumed that the loop bounds are known constants and that the unrolling factor divides the number of iterations evenly. In general, these conditions are, of course, not satisfied. However, loops with general bounds can still be unrolled. What we need to do is to keep a rolled copy of the loop, exit the unrolled copy when the number of iterations remaining is less than the unrolling factor, and then use the rolled copy to execute the remaining iterations. We take this approach rather than testing in each unrolled copy of the loop body for early termination because one reason for unrolling loops is to allow instruction scheduling more latitude, in particular to allow it to interleave instructions from the copies of the body, which it cannot do as effectively if there is conditional control flow between the copies." [26]

"The most important issues in loop unrolling are deciding which loops to unroll and by what factor. This has two aspects: one involves architectural characteristics, such as the number of registers available, the available overlap among, for example, floating-point operations and memory references, and the size and organization of the instruction cache; and the other is the selection of particular loops in a program to unroll and the unrolling factors to use for them. The impact of some of the architectural characteristics is often best determined by experimentation. The result is usually a heuristic, but unrolling decisions for individual loops can benefit significantly from feedback from profiled runs of the program that contains the loops.

The result of a set of such experiments would be rules that can be used to decide what loops to unroll, which might depend, for example, on the following types of characteristics of loops:

1.those that contain only a single basic block (i.e., straight-line code),

2.those that contain a certain balance of floating-point and memory operations or a certain balance of integer memory operations,

3.those that generate a small number of intermediate-code instructions, and

4.those that have simple loop control.

The first and second criteria restrict unrolling to loops that are most likely to benefit from instruction scheduling. The third keeps the unrolled blocks of code short, so as not to adversely impact cache performance. The last criterion keeps the compiler from unrolling loops for which it is difficult to determine when to take the early exit to the rolled copy for the final iterations, such as when traversing a linked list. The unrolling factor may be anywhere from two up, depending on the specific contents of the loop body, but will usually not be more than four and almost never more than eight, although further development of VLIW machines may provide good use for larger unrolling factors.

Loop unrolling generally increases the available instruction-level parallelism, especially if several other transformations are performed on the copies of the loop body to remove unnecessary dependences. Such transformations include software register renaming, variable expansion, and instruction combining. Using dependence testing to disambiguate potential aliases between memory addresses can also remove dependences. Thus, unrolling has the potential of significant benefit for most implementations and particularly for superscalar and VLIW ones." [26]

("VLIW" stands for "Very Large Instruction Word", an instruction set architecture ("ISA") designed for instruction level parallelism ("ILP"). ILP is the parallel or simultaneous execution of a sequence of instructions in a computer program. See [30] for VLIW.)

#### 2.3.2.5.1 Full loop unrolling

It is possible to fully unroll a loop. If an index is used, it can be replaced by the actual constant values it will have for every instruction. The loop itself is removed, only the body is kept.

Example:

input:

"

```
for(i = 0; i < 5; i++)
{
    a[i] = b[i] + c[i];
}
```

"

output:

"

```
a[0] = b[0] + c[0];
a[1] = b[1] + c[1];
a[2] = b[2] + c[2];
a[3] = b[3] + c[3];
a[4] = b[4] + c[4];
i = 5;
```

"

### 2.3.2.6 Expression simplification and compression

Expression simplification is the decomposition of one instruction which consists of multiple embedded expressions into multiple instructions consisting of less or only one expression. Compression can be applied to control the level of depth of the expressions before being split into more instructions.

The benefit of this optimization is, if a boolean instruction takes 2ns to execute and the clock speed of the target hardware is 6ns, 4ns have been wasted waiting for the next instruction if the expression depth is the minimum. A boolean expression depth (compression) of three would fit the clock cycle perfectly, and so no time would be wasted. If, however, we incorrectly set a compression level of 4, then 4ns would be wasted again as two clock cycles would be necessary (all of the first ones used, but only 1/3 of the second one). It is thus important that these constraints are able to be declared properly.

Input:

"

```
i = (c < (b - 1)) || (a && (b > (c + 5)));
```

"

Output (compression level of 2 for both boolean and integer operations):

"

```
i1 = c < (b - 1);
i2 = b > (c + 5);
i3 = i1 || (a && i2);
```

"

#### 2.3.2.7 Loop Pipelining/Software Pipelining

"Software pipelining operates specifically on loop bodies and, since loops are where most programs spend most of their execution time, can result in large improvements in performance, often a factor of two or more." [26]

"In a compiler that does software pipelining, it is crucial to making it as effective as possible to have loop unrolling, variable expansion, and register renaming available to be performed on the loop bodies that are being pipelined." [26]

"Software pipelining is an optimization that can improve the loop-executing performance of any system that allows instruction-level parallelism, including VLIW and superscalar systems, but also one-scalar implementations that allow, e.g., integer and floating-point instructions to be executing at the same time but not to be initiated at the same time. It works by allowing parts of several iterations of a loop

to be in process at the same time, so as to take advantage of the parallelism available in the loop body." [26]

"Since software pipelining moves a fixed number of iterations of a loop out of the loop body, we must either know in advance that the loop is repeated at least that many times, or we must generate code that tests this, if possible, at run time and that chooses to execute either a software-pipelined version of the loop or one that is not. Of course, for some loops it is not possible to determine the number of iterations they take without executing them, and software pipelining cannot be applied to such loops." [26]

For the example below assume S3 depends on S2 and S2 depends on S1:

input:

"

```
for(i1 = 1; i1 < 2; i1++)
{
    for(i2 = 1; i2 < 10; i2++)
    {
        S1();
        S2();
        S3();
    }
}
```

" (C appropriation of an example from [31])

output:

"

```
for(i1 = 1; i1 < 2; i1++)
{
    S1();
    S2(); S1(); // These will run in parallel
    for(i2 = 3; i2 < 10; i2++)
    {
        S3(); S2(); S1(); // These will run in parallel
    }
```

```
    S3(); S2(); // These will run in parallel
    S3();
}
```

" (C appropriation of an example from [31])

#### 2.3.2.7.1 Recurrence duplication

With recurrence duplication a recurrent assignment might be repeated through a temporary variable, in order to eliminate dependence cycles and thus allow pipelining with a minimal initiation interval between consecutive iterations (II). [4] It can be combined with code motion.

Example input:

```
"
for (i = 0; i < 1000; i++) {
  curr[x] = clp[block[i] + curr[x]];
  x++;
}
" [4]
```

Example (expression simplified):

```
"
i = 0;
temp = i < 1000;
while (temp) {
  t1 = block[i];
  t2 = curr[x];
  t3 = t1 + t2;
  t4 = clp[t3];
  curr[x] = t4;
  x++;
  i++;
  temp = i < 1000;
}
" [4]
```

Example (expression simplified and pipelined with II = 4cc) output:

"

```
i = 0;
t1 = block[i]; // Pipeline prologue
t2 = curr[x];
t3 = t1 + t2;
t4 = clp[t3];
i++;
temp = i < 1000;
while (temp) { // Pipeline body
  curr[x] = t4; x++; t1 = block[i]; // These will run in parallel
  t2 = curr[x];
  t3 = t1 + t2; i++; // These will run in parallel
  t4 = clp[t3]; temp = i < 1000; // These will run in parallel
}
curr[x] = t4; // Pipeline epilogue
x++;
```

" [4]

Example (expression simplified and pipelined with II = 1cc) output:

"

```
i = 0;
t_x = x; // Temporary for recurrence duplication
t1 = block[i]; // Pipeline prologue
t2 = curr[t_x];
t_x++;
i++;
t3 = t1 + t2;
t1 = block[i];
t2 = curr[t_x];
t_x++;
```

```
i++;
t4 = clp[t3];
t3 = t1 + t2;
t1 = block[i];
t2 = curr[t_x];
t_x++;
i++;
temp = i < 1000;
while (temp) { // Pipeline body
  curr[x] = t4; x++; t4 = clp[t3]; t3 = t1 + t2; t1 = block[i]; t2 = curr[t_x]; t_x++;
        temp = i < 999; i++; // All will run in parallel
}
curr[x] = t4; // Pipeline epilogue
x++;
t4 = clp[t3];
t3 = t1 + t2;
curr[x] = t4;
x++;
t4 = clp[t3];
curr[x] = t4;
x++;
```

" [4]

## 2.4 High Level Synthesis (HLS)

"HLS is an automated design process that takes as input an algorithmic description in order to create the digital hardware that implements the desired function. Typically, the control algorithms are written in a high-level programming language such as C or variants (SystemC, OpenCL framework, among others), and the automated tool provides the register transfer level (RTL) hardware description. " [32]

"The HLST design process workflow is summarized in image 6. First, the desired control algorithm is described using a high-level programming language, typically C." [32]

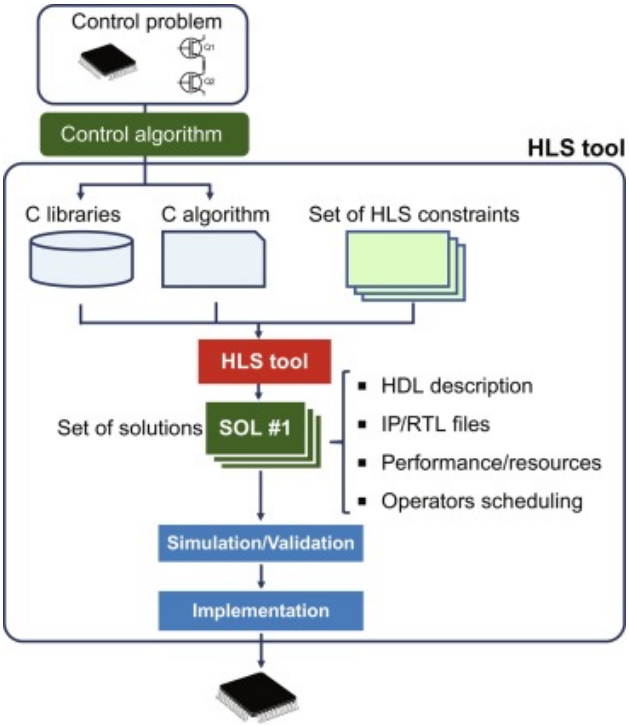


**Image 6. HLST Design process**

Πηγή: **[32]**

"Together with the algorithm to implement, the HLST takes as an input a set of constraints defined by the designer in order to perform the RTL synthesis. This is the most critical point in the design flow, since the defined constraints will define the final performance, resource usage, and energy consumption, leading to optimum or subpar implementations. Typically, when using HLS tools for C-to-VHDL translation the most important constraints are those related to loop unrolling, pipelining, and memories partition. By combining correctly both constraints optimum implementations can be achieved considering the required performance and available resources/cost." [32]

"The output of the HLST includes the set of solutions created as a result of the set of constraints. For each solution, the HDL description (either VHDL

or Verilog), operator scheduling graphs, and performance reports are provided." [32]

"One of the main benefits of the HLST approach is that a set of algorithms and HLS constraints can be directly evaluated. This enables new design possibilities and opens the window to wide design space exploration leading to unprecedented optimization. It is important to note that, for complex algorithms, it is unfeasible to test a high number of implementations as well as unrolling/pipelining possibilities by using hand-coded solutions." [32]

## 2.5 Parallelization and performance

"Given the complexities of computer architecture, and the fact that different computers can vary significantly, how can you optimize code for performance across a range of computer architectures? The trick is to realize that modern computer architectures are designed around two key assumptions: data locality and the availability of parallel operations. Get these right and good performance can be achieved on a wide range of machines, although perhaps after some per-machine tuning. However, if you violate these assumptions, you cannot expect good performance no matter how much low-level tuning you do." [33] [p50]

### 2.5.1 Data locality

"Good use of memory bandwidth and good use of cache depends on good data locality, which is the reuse of data from nearby locations in time or space. Therefore, you should design your algorithms to have good data locality by using one or more of the following strategies:

Break work up into chunks that can fit in cache. If the working set for a chunk of work does not fit in cache, it will not run efficiently.

Organize data structures and memory accesses to reuse data locally when possible. Avoid unnecessary accesses far apart in memory and especially simultaneous access to multiple memory locations located a power of two apart. The last consideration is to avoid cache conflicts on caches with low associativity.

To avoid unnecessary TLB misses, avoid accessing too many pages at once.

Align data with cache line boundaries. Avoid having unrelated data accesses from different cores access the same cache lines, to avoid false sharing." [33][p50]

(TLB stands for "Translation Lookaside Buffer (TLB). The TLB is a specialized cache that translates logical addresses to physical addresses for a small set of active pages. Like ordinary caches, it may have hierarchical levels and may be split for instructions versus data." [33][p47] The working set is "the total amount of physical memory that needs to be accessed within a time period that is short relative to the disk access time". [33][p47])

"Another issue that affects the achievable performance of an algorithm is arithmetic intensity. This is the ratio of computation to communication. Given the fact that on-chip compute performance is still rising with the number of transistors, but off-chip bandwidth is not rising as fast, in order to achieve scalability approaches to parallelism should be sought that give high arithmetic intensity. This ideally means that a large number of on-chip compute operations should be performed for every off-chip memory access." [33][p50-51]

### 2.5.2 Parallel slack

"Parallel slack is the amount of “extra” parallelism available (Section 2.5.6) above the minimum necessary to use the parallel hardware resources. Specifying a significant amount of potential parallelism higher than the actual parallelism of the hardware gives the underlying software and hardware schedulers more flexibility to exploit machine resources.

Normally you want to choose the smallest work units possible that reasonably amortize the overhead of scheduling them and give good arithmetic intensity. Breaking down a problem into exactly as many chunks of work as there are cores available on the machine is tempting, but not necessarily optimal, even if you know the number of cores on the machine. If you only have one or a few tasks on each core, then a delay on one core (perhaps due to an operating system interrupt) is likely to delay the entire program." [33][p51]

### 2.5.3 Performance theory

The primary purpose of parallelization, is performance. Performance is usually about one of the following:

"Reducing the total time it takes to compute a single result (latency; Section 2.5.1)

Increasing the rate at which a series of results can be computed (throughput; Section 2.5.1)

Reducing the power consumption of a computation" [33][p54]

#### 2.5.3.1 Latency and throughput

"The time it takes to complete a task is called latency. It has units of time. The scale can be anywhere from nanoseconds to days. Lower latency is better.

The rate a which a series of tasks can be completed is called throughput. This has units of work per unit time. Larger throughput is better. A related term is bandwidth, which refers to throughput rates that have a frequency-domain interpretation, particularly when referring to memory or communication transactions.

Some optimizations that improve throughput may increase the latency. For example, processing of a series of tasks can be parallelized by pipelining, which overlaps different stages of processing. However, pipelining adds overhead since the stages must now synchronize and communicate, so the time it takes to get one complete task through the whole pipeline may take longer than with a simple serial implementation." [33][p55]

#### 2.5.3.2 Speedup, Efficiency, and Scalability

"Two important metrics related to performance and parallelism are speedup and efficiency. Speedup compares the latency for solving the identical computational problem on one hardware unit (“worker”) versus on P hardware units:" [33][p56]

$$\text{speedup} = S_P = \frac{T_1}{T_P}$$

**Image 7.** Speedup

Πηγή: **[33]**[p56]

"where T1 is the latency of the program with one worker and TP is the latency on P workers. Efficiency is speedup divided by the number of workers:" [33][p56]

$$\text{efficiency} = \frac{S_P}{P} = \frac{T_1}{PT_P}.$$

**Image 8.** Efficiency

Πηγή: **[33]**[p56]

"Efficiency measures return on hardware investment. Ideal efficiency is 1 (often reported as 100%), which corresponds to a linear speedup, but many factors can reduce efficiency below this ideal.

If T1 is the latency of the parallel program running with a single worker," [33][p56] image 7 (“speedup”) "is sometimes called relative speedup, because it shows relative improvement from using P workers. This uses a serialization of the parallel algorithm as the baseline. However, sometimes there is a better serial algorithm that does not parallelize well. If so, it is fairer to use that algorithm for T1 , and report absolute speedup, as long as both algorithms are solving an identical computational problem. Otherwise, using an unnecessarily poor baseline artificially inflates speedup and efficiency." [33][p56]

"An algorithm that runs P times faster on P processors is said to exhibit linear speedup. Linear speedup is rare in practice, since there is extra work involved in distributing work to processors and coordinating them. In addition, an optimal serial algorithm may be able to do less work overall than an optimal parallel algorithm for certain problems, so the achievable speedup may be sublinear in P, even on theoretical ideal machines. Linear speedup is usually considered optimal since we can serialize the parallel algorithm, as noted above, and run it on a serial machine with a linear slowdown as a worst-case baseline.

However, as exceptions that prove the rule, an occasional program will exhibit superlinear speedup—an efficiency greater than 100%. Some common causes of superlinear speedup include:

Restructuring a program for parallel execution can cause it to use cache memory better, even when run on with a single worker! But if T1 from the old program is still used for the speedup calculation, the speedup can appear to be superlinear." [33] [p56-57]

"The program’s performance is strongly dependent on having a sufficient amount of cache memory, and no single worker has access to that amount. If multiple workers bring that amount to bear, because they do not all share the same cache, absolute speedup really can be superlinear.

The parallel algorithm may be more efficient than the equivalent serial algorithm, since it may be able to avoid work that its serialization would be forced to do. For example, in search tree problems, searching multiple branches in parallel sometimes permits chopping off branches (by using results computed in sibling branches) sooner than would occur in the serial code.

However, for the most part, sublinear speedup is the norm." [33][p57]

#### 2.5.3.3 Power

"Parallelization can reduce power consumption. CMOS is the dominant circuit technology for current computer hardware. CMOS power consumption is the sum of dynamic power consumption and static power consumption". [33][p57]

"Suppose that parallelization speeds up an application by 1.5× on two cores. You can use this speedup either to reduce latency or reduce power. If your latency requirement is already met, then reducing the clock rate of the cores by 1.5× will save a significant amount of power. Let P1 be the power consumed by one core running the serial version of the application. Then the power consumed by two cores running the parallel version of the application will be given by:" [33][p57-58]

$$\mathcal{P}_2 = 2\left(\frac{1}{1.5}\right)^3 \mathcal{P}_1$$

$$\approx 0.6\mathcal{P}_1,$$

**Image 9.** **Power: two core example**

Πηγή: **[33]**[p58]

"where the factor of 2 arises from having two cores. Using two cores running the parallelized version of the application at the lower clock rate has the same latency but uses (in this case) 40% less power.

Unfortunately, reality is not so simple. Current chips have so many transistors that frequency and voltage are already scaled down to near the lower limit just to avoid overheating, so there is not much leeway for raising the frequency." [33][p58]

"Especially in mobile devices, parallelism can be used to reduce latency. This reduces the time the device, including its display and other components, is powered up. This not only improves the user experience but also reduces the overall power consumption for performing a user's task: a win-win." [33][p58]

#### 2.5.3.4 Amdahl's law

"Amdahl argued that the execution time T1 of a program falls into two categories:

Time spent doing non-parallelizable serial work

Time spent doing parallelizable work

Call these Wser and Wpar , respectively. Given P workers available to do the parallelizable work, the times for sequential execution and parallel execution are:" [33][p59]

$$T_1 = W_{\text{ser}} + W_{\text{par}},$$
$$T_P \geq W_{\text{ser}} + W_{\text{par}}/P.$$

**Image 10.** **Times for sequential and parallel execution**

Πηγή: **[33]**[p59]

"The bound on TP assumes no superlinear speedup, and is an exact equality only if the parallelizable work can be perfectly parallelized. Plugging these relations into the definition of speedup yields Amdahl's Law:" [33][p59]

$$S_P \leq \frac{W_{\text{ser}} + W_{\text{par}}}{W_{\text{ser}} + W_{\text{par}}/P}.$$

**Image 11.** **Amdahl's law**

Πηγή: **[33]**[p59]

"Amdahl's Law has an important corollary. Let f be the non-parallelizable serial fraction of the total work. Then the following equalities hold:" [33][p59]

$$W_{\text{ser}} = f T_1,$$
$$W_{\text{par}} = (1-f) T_1.$$

**Image 12.** **Amdahl's law corollary equalities**

Πηγή: **[33]**[p59]

"Substitute these into " [33][p59] image 11 ("Amdahl's law") " and simplify to get:" [33][p59]

$$S_P \leq \frac{1}{f + (1-f)/P}.$$

**Image 13.** **Amdahl's law simplification**

Πηγή: **[33]**[p59]

"Now consider what happens when P tends to infinity:" [33][p60]

$$S_\infty \leq \frac{1}{f}.$$

**Image 14.** **Amdahl's law: P tending to infinity**

Πηγή: **[33]**[p60]

"Speedup is limited by the fraction of the work that is not parallelizable, even using an infinite number of processors. If 10% of the application cannot be parallelized, then the maximum speedup is 10×. If 1% of the application cannot be parallelized, then the maximum speedup is 100×. In practice, an infinite number of processors is not available. With fewer processors, the speedup may be reduced, which gives an upper bound on the speedup." [33][p60]

#### 2.5.3.5 Gustafson-Barsis' Law

"Amdahl's Law views programs as fixed and the computer as changeable, but experience indicates that as computers get new capabilities, applications change to exploit these features. Most of today's applications would not run on computers from 10 years ago, and many would run poorly on machines that are just 5 years old." [33][p60-61]

"More than two decades after the appearance of Amdahl's Law, John Gustafson2 noted that several programs at Sandia National Labs were speeding up by over 1000×. Clearly, Amdahl's Law could be evaded.

Gustafson noted that problem sizes grow as computers become more powerful. As the problem size grows, the work required for the parallel part of the problem frequently grows much faster than the serial part. If this is true for a given application, then as the problem size grows the serial fraction decreases and speedup improves." [33][p61]

"Both Amdahl's and Gustafson-Barsis' Laws are correct. It is a matter of "glass half empty" or "glass half full." The difference lies in whether you want to make a program run faster with the same workload or run in the same time with a larger workload. History clearly favors programs getting more complex and solving larger problems, so Gustafson's observations fit the historical trend. Nevertheless, Amdahl's Law still

haunts you when you need to make an application run faster on the same workload to meet some latency target.

Furthermore, Gustafson-Barsis' observation is not a license for carelessness. In order for it to hold it is critical to ensure that serial work grows much more slowly than parallel work, and that synchronization and other forms of overhead are scalable." [33][p62]

#### 2.5.3.6 Work-Span Model

"The work-span model is much more useful than Amdahl's law for estimating program running times, because it takes into account imperfect parallelization. Furthermore, it is not just an upper bound as it also provides a lower bound.

It lets you estimate TP from just two numbers: T1 and T∞.

In the work-span model, tasks form a directed acyclic graph. A task is ready to run if all of its predecessors in the graph are done. The basic work-span model ignores communication and memory access costs. It also assumes task scheduling is greedy, which means the scheduler never lets a hardware worker sit idle while there is a task ready to run.

The extreme times for P = 1 and P = ∞ are so important that they have names. Time T1 is called the work of an algorithm. It is the time that a serialization of the algorithm would take and is simply the total time it would take to complete all tasks. Time T∞ is called the span of an algorithm. The span is the time a parallel algorithm would take on an ideal machine with an infinite number of processors. Span is equivalent to the length of the critical path. The critical path is the longest chain of tasks that must be executed one after each other. Synonyms for span in the literature are step complexity or depth." [33][p62]

"Work and span each put a limit on speedup. Superlinear speedup is impossible in the work-span model:" [33][p63]

$$S_P = \frac{T_1}{T_P} \leq \frac{T_1}{T_1/P} = P.$$

**Image 15.** **Work-Span speedup can not be superlinear**

Πηγή: **[33]**[p63]

"On an ideal machine with greedy scheduling, adding processors never slows down an algorithm:" [33][p63]

$$S_P = \frac{T_1}{T_P} \leq \frac{T_1}{T_\infty}.$$

**Image 16.** **Adding processors does not slow an algorithm down**

Πηγή: **[33]**[p63]

"Or more colloquially:" [33][p63]

$$\text{speedup} \leq \frac{\text{work}}{\text{span}}.$$

**Image 17.** **Speedup is less than or equal to the work by the span**

Πηγή: **[33]**[p63]

"The span provides more than just an upper bound on speedup. It can also be used to estimate a lower bound on speedup for an ideal machine. An inequality known as Brent's Lemma [Bre74] bounds TP in terms of the work T1 and the span T∞ :" [33][p63]

$$T_P \leq (T_1 - T_\infty)/P + T_\infty.$$

**Image 18.** **Brent's lemma**

Πηγή: **[33]**[p63]

"Here is the argument behind the lemma. The total work T1 can be divided into two categories: perfectly parallelizable work and imperfectly parallelizable work. The imperfectly parallelizable work takes time T∞ no matter how many workers there are. The perfectly parallelizable work remaining takes time T1 − T∞ with a single worker, and since it is perfectly parallelizable it speeds up by P if all P workers are working on it. But if not all P workers are working on it, then at least one worker is working on the T∞ component. The argument resembles Amdahl's argument, but generalizes the notion of an inherently serial portion of work to imperfectly parallelizable work.

Though the argument resembles Amdahl's argument, it proves something quite different. Amdahl's argument put a lower bound on TP and is exact

only if the parallelizable portion of a program is perfectly parallelizable. Brent's Lemma puts an upper bound on TP . It says what happens if the worst possible assignment of tasks to workers is chosen.

In general, work-span analysis is a far better guide than Amdahl's Law, because it usually provides a tighter upper bound and also provides a lower bound." [33][p63-64]

"Brent's Lemma leads to a useful formula for estimating TP from the work T1 and span T∞ . To get much speedup, T1 must be significantly larger than T∞ , In this case, T1 − T∞ ≈ T1 and the right side of " [33][p64] image 18 ("Brent's lemma") " also turns out to be a good lower bound estimate on TP . So the following approximation works well in practice for estimating running time:" [33][p64]

$$T_P \approx T_1/P + T_\infty \quad \text{if } T_\infty \ll T_1.$$

**Image 19.** **Approximation to estimate running time**

Πηγή: **[33]**[p64]

"The approximation says a lot:

Increasing the total work T1 hurts parallel execution proportionately.

The span T∞ impacts scalability, even when P is finite.

When designing a parallel algorithm, avoid creating significantly more work for the sake of parallelization, and focus on reducing the span, because the span is the fundamental asymptotic limit on scalability. Increase the work only if it enables a drastic decrease in span." [33][p65]

"Brent's Lemma also leads to a formal motivation for overdecomposition. From Equation" [33][p65] image 18 ("Brent's lemma") "the following condition can be derived:" [33][p65]

$$S_P = T_1/T_P \approx P \quad \text{if } T_1/T_\infty \gg P.$$

**Image 20.** **Overdecomposition**

Πηγή: **[33]**[p65]

"It says that greedy scheduling achieves linear speedup if a problem is overdecomposed to create much more potential parallelism than the hardware can use. The excess parallelism is called the parallel slack, and is defined by:" [33][p65]

$$\text{parallel slack} = \frac{S_\infty}{P} = \frac{T_1}{PT_\infty}$$

**Image 21.** **Parallel slack**

Πηγή: **[33]**[p65]

"In practice, a parallel slack of at least 8 works well.

If you remember only one thing about time estimates for parallel programs, remember" [37][p65] image 19 (“Approximation to estimate running time”). "From it, you can derive performance estimates just by knowing the work T1 and span T∞ of an algorithm. However, this formula assumes the following three important qualifications:

Memory bandwidth is not a limiting resource.

There is no speculative work. In other words, the parallel code is doing T1 total work, period.

The scheduler is greedy." [33][p65]

# 3. The csense compiler

This chapter describes the csense compiler, its capabilities and details on the functions that perform the loop unrolling optimization and expression simplification/compression.

## 3.1 csense overview

csense is a source to source compiler by Michael Dosis and Georgios Dimitriou, written in C that translates from C to Ada (the implementation is due to Georgios Dimitriou). It uses flex for the lexical analysis and bison for parsing. The intermediate representation (IR) is in the form of an abstract syntax tree (AST). The AST is constructed by a set of structs forming trees with linked lists sometimes connected to them (for example: a "for" loop and its body are connected via a linked list.).

### 3.1.1 flex

flex stands for "the fast lexical analyser generator". [34]

flex files usually end in ".l".

"'flex' is a tool for generating "scanners". A scanner is a program which recognizes lexical patterns in text. The 'flex' program reads the given input files, or its standard input if no file names are given, for a description of a scanner to generate. The description is in the form of pairs of regular expressions and C code, called "rules". 'flex' generates as output a C source file, 'lex.yy.c' by default, which defines a routine 'yylex()'. This file can be compiled and linked with the flex runtime library to produce an executable. When the executable is run, it analyzes its input for occurrences of the regular expressions. Whenever it finds one, it executes the corresponding C code." [35]

"The 'flex' input file consists of three sections, separated by a line containing only '%%'.

definitions

%%

rules

%%

user code

" [35]

#### 3.1.1.1 Definitions section

"The "definitions section" contains declarations of simple "name" definitions to simplify the scanner specification, and declarations of "start conditions", which are explained in a later section.

Name definitions have the form:

name definition

The 'name' is a word beginning with a letter or an underscore ('_') followed by zero or more letters, digits, '_', or '-' (dash). The definition is taken to begin at the first non-whitespace character following the name and continuing to the end of the line. The definition can subsequently be referred to using '{name}', which will expand to '(definition)'. For example,

```
DIGIT    [0-9]
ID       [a-z][a-z0-9]*
```

Defines 'DIGIT' to be a regular expression which matches a single digit, and 'ID' to be a regular expression which matches a letter followed by zero-or-more letters-or-digits. A subsequent reference to

```
{DIGIT}+"."{DIGIT}*
```

is identical to

```
([0-9])+"."([0-9])*
```

and matches one-or-more digits followed by a '.' followed by zero-or-more digits.

An unindented comment (i.e., a line beginning with '/*') is copied verbatim to the output up to the next '*/'.

Any _indented_ text or text enclosed in '%{' and '%}' is also copied verbatim to the output (with the %{ and %} symbols removed). The %{ and %} symbols must appear unindented on lines by themselves.

A '%top' block is similar to a '%{' ... '%}' block, except that the code in a '%top' block is relocated to the _top_ of the generated file, before any flex definitions (1). The '%top' block is useful when you want certain preprocessor macros to be defined or certain files to be included before the generated code. The single characters, '{' and '}' are used to delimit the '%top' block, as show in the example below:

```
%top{
  /* This code goes at the "top" of the generated file. */
  #include <stdint.h>
  #include <inttypes.h>
}
```

Multiple '%top' blocks are allowed, and their order is preserved.

" [35]

#### 3.1.1.2 Rules section

"The "rules" section of the 'flex' input contains a series of rules of the form:

```
pattern   action
```

where the pattern must be unindented and the action must begin on the same line. " [35]

"

In the rules section, any indented or %{ %} enclosed text appearing before the first rule may be used to declare variables which are local to the scanning routine and (after the declarations) code which is to be executed whenever the scanning routine is entered. Other indented or %{ %} text in the rule section is still copied to the output,

but its meaning is not well-defined and it may well cause compile-time errors (this feature is present for POSIX compliance. *Note Lex and Posix::, for other such features).

Any _indented_ text or text enclosed in '%{' and '%}' is copied verbatim to the output (with the %{ and %} symbols removed). The %{ and %} symbols must appear unindented on lines by themselves." [35]

#### 3.1.1.3 User code section

"The user code section is simply copied to 'lex.yy.c' verbatim. It is used for companion routines which call or are called by the scanner. The presence of this section is optional; if it is missing, the second '%%' in the input file may be skipped, too." [35]

### 3.1.2 bison

The "NAME" section in the bison "man" page reads "GNU Project parser generator (yacc replacement)". [36]

A bison file usually has a ".y" extension.

"

"Bison" is a general-purpose parser generator that converts an annotated context-free grammar into a deterministic LR or generalized LR (GLR) parser employing LALR(1), IELR(1) or canonical LR(1) parser tables. Once you are proficient with Bison, you can use it to develop a wide range of language parsers, from those used in simple desk calculators to complex programming languages.

" [37]

#### 3.1.2.1 General information about bison

"Bison was written originally by Robert Corbett. Richard Stallman made it Yacc-compatible. Wilfred Hansen of Carnegie Mellon University added multi-character string literals and other features. Since then, Bison has grown more robust and evolved many other new features thanks to the hard work of a long list of volunteers." [37]

"In order for Bison to parse a language, it must be described by a "context-free grammar". This means that you specify one or more "syntactic groupings" and give rules for constructing them from their parts. For example, in the C language, one kind of grouping is called an 'expression'. One rule for making an expression might be, "An expression can be made of a minus sign and another expression". Another would be, "An expression can be an integer". As you can see, rules are often recursive, but there must be at least one rule which leads out of the recursion.

The most common formal system for presenting such rules for humans to read is “Backus-Naur Form” or “BNF”, which was developed in order to specify the language Algol 60. Any grammar expressed in BNF is a context-free grammar. The input to Bison is essentially machine-readable BNF." [37]

"Although it can handle almost all context-free grammars, Bison is optimized for what are called LR(1) grammars. In brief, in these grammars, it must be possible to tell how to parse any portion of an input string with just a single token of lookahead." [37]

"Parsers for LR(1) grammars are “deterministic”, meaning roughly that the next grammar rule to apply at any point in the input is uniquely determined by the preceding input and a fixed, finite portion (called a “lookahead”) of the remaining input." [37]

#### 3.1.2.2 bison token interpretation

"In the formal grammatical rules for a language, each kind of syntactic unit or grouping is named by a “symbol”. Those which are built by grouping smaller constructs according to grammatical rules are called “nonterminal symbols”; those which can’t be subdivided are called “terminal symbols” or “token kinds”. We call a piece of input corresponding to a single terminal symbol a “token”, and a piece corresponding to a single nonterminal symbol a “grouping”.

We can use the C language as an example of what symbols, terminal and nonterminal, mean. The tokens of C are identifiers, constants (numeric and string), and the various keywords, arithmetic operators and punctuation marks. So the terminal symbols of a grammar for C include ‘identifier’, ‘number’, ‘string’, plus one symbol for each keyword, operator or punctuation mark: ‘if’, ‘return’, ‘const’, ‘static’, ‘int’, ‘char’, ‘plus-sign’, ‘open-brace’, ‘close-brace’, ‘comma’ and many more. (These tokens can be subdivided into characters, but that is a matter of lexicography, not grammar.)

Here is a simple C function subdivided into tokens:

```
int           /* keyword ‘int’ */
square (int x)  /* identifier, open-paren, keyword ‘int’,
                   identifier, close-paren */
{             /* open-brace */
  return x * x; /* keyword ‘return’, identifier, asterisk,
                   identifier, semicolon */
}             /* close-brace */
```

" [37]

"The Bison parser reads a sequence of tokens as its input, and groups the tokens using the grammar rules. If the input is valid, the end result is that the entire token sequence reduces to a single grouping whose symbol is the grammar's start symbol. If we use a grammar for C, the entire input must be a 'sequence of definitions and declarations'. If not, the parser reports a syntax error." [37]

"A nonterminal symbol in the formal grammar is represented in Bison input as an identifier, like an identifier in C. By convention, it should be in lower case, such as 'expr', 'stmt' or 'declaration'.

The Bison representation for a terminal symbol is also called a "token kind". Token kinds as well can be represented as C-like identifiers. By convention, these identifiers should be upper case to distinguish them from nonterminals: for example, 'INTEGER', 'IDENTIFIER', 'IF' or 'RETURN'. A terminal symbol that stands for a particular keyword in the language should be named after that keyword converted to upper case. The terminal symbol 'error' is reserved for error recovery." [37]

"The grammar rules also have an expression in Bison syntax. For example, here is the Bison rule for a C 'return' statement. The semicolon in quotes is a literal character token, representing part of the C syntax for the statement; the naked semicolon, and the colon, are Bison punctuation used in every rule.

stmt: RETURN expr ';' ;" [37]

"The token kind is a terminal symbol defined in the grammar, such as 'INTEGER', 'IDENTIFIER' or ‘','’. It tells everything you need to know to decide where the token may validly appear and how to group it with other tokens. The grammar rules know nothing about tokens except their kinds.

The semantic value has all the rest of the information about the meaning of the token, such as the value of an integer, or the name of an identifier. (A token such as ‘','’ which is just punctuation doesn't need to have any semantic value.)

For example, an input token might be classified as token kind 'INTEGER' and have the semantic value 4. Another input token might have the same token kind 'INTEGER' but value 3989. When a grammar rule says that 'INTEGER' is allowed, either of these tokens is acceptable because each is an 'INTEGER'. When the parser accepts the token, it keeps track of the token's semantic value." [37]

#### 3.1.2.3 bison rules

"In order to be useful, a program must do more than parse input; it must also produce some output based on the input. In a Bison grammar, a grammar rule can have an “action” made up of C statements. Each time the parser recognizes a match for that rule, the action is executed.

Most of the time, the purpose of an action is to compute the semantic value of the whole construct from the semantic values of its parts. For example, suppose we have a rule which says an expression can be the sum of two expressions. When the parser recognizes such a sum, each of the subexpressions has a semantic value which describes how it was built up. The action for this rule should create a similar sort of value for the newly recognized larger expression.

For example, here is a rule that says an expression can be the sum of two subexpressions:

```
expr: expr '+' expr   { $$ = $1 + $3; } ;
```

The action says how to produce the semantic value of the sum expression from the values of the two subexpressions." [37]

"Each token has a semantic value. In a similar fashion, each token has an associated location, but the type of locations is the same for all tokens and groupings. Moreover, the output parser is equipped with a default data structure for storing locations." [37]

"Like semantic values, locations can be reached in actions using a dedicated set of constructs. In the example above, the location of the whole grouping is ‘@$’, while the locations of the subexpressions are ‘@1’ and ‘@3’.

When a rule is matched, a default action is used to compute the semantic value of its left hand side. In the same way, another default action is used for locations. However, the action for locations is general enough for most cases, meaning there is usually no need to describe for each rule how ‘@$’ should be formed. When building a new location for a given grouping, the default behavior of the output parser is to take the beginning of the first symbol, and the end of the last symbol" [37]

"When you run Bison, you give it a Bison grammar file as input. The most important output is a C source file that implements a parser for the language described by the grammar. This parser is called a “Bison parser”, and this file is called a “Bison parser

implementation file". Keep in mind that the Bison utility and the Bison parser are two distinct programs: the Bison utility is a program whose output is the Bison parser implementation file that becomes part of your program.

The job of the Bison parser is to group tokens into groupings according to the grammar rules—for example, to build identifiers and operators into expressions. As it does this, it runs the actions for the grammar rules it uses.

The tokens come from a function called the "lexical analyzer" that you must supply in some fashion (such as by writing it in C). The Bison parser calls the lexical analyzer each time it wants a new token. It doesn't know what is "inside" the tokens (though their semantic values may reflect this). Typically the lexical analyzer makes the tokens by parsing characters of text, but Bison does not depend on this.

The Bison parser implementation file is C code which defines a function named 'yyparse' which implements that grammar. This function does not make a complete C program: you must supply some additional functions. One is the lexical analyzer. Another is an error-reporting function which the parser calls to report an error. In addition, a complete C program must start with a function called 'main'; you have to provide this, and arrange for it to call 'yyparse' or the parser will never run.

Aside from the token kind names and the symbols in the actions you write, all symbols defined in the Bison parser implementation file itself begin with 'yy' or 'YY'. This includes interface functions such as the lexical analyzer function 'yylex', the error reporting function'yyerror' and the parser function 'yyparse' itself. This also includes numerous identifiers used for internal purposes." [37]

#### 3.1.2.4 Language design process using bison

"The actual language-design process using Bison, from grammar specification to a working compiler or interpreter, has these parts:

1. Formally specify the grammar in a form recognized by Bison. For each grammatical rule in the language, describe the action that is to be taken when an instance of that rule is recognized. The action is described by a sequence of C statements.

2. Write a lexical analyzer to process input and pass tokens to the parser. The lexical analyzer may be written by hand in C. It could also be produced using Lex, but the use of Lex is not discussed in this manual.

3. Write a controlling function that calls the Bison-produced parser.

4. Write error-reporting routines.

To turn this source code as written into a runnable program, you must follow these steps:

1. Run Bison on the grammar to produce the parser.

2. Compile the code output by Bison, as well as any other source files.

3. Link the object files to produce the finished product.

" [37]

#### 3.1.2.5 bison grammar file structure

"The input file for the Bison utility is a “Bison grammar file”. The general form of a Bison grammar file is as follows:

```
%{
PROLOGUE
%}

BISON DECLARATIONS

%%
GRAMMAR RULES
%%
```

EPILOGUE

The '%%', '%{' and '%}' are punctuation that appears in every Bison grammar file to separate the sections.

The prologue may define types and variables used in the actions. You can also use preprocessor commands to define macros used there, and use '#include' to include header files that do any of these things. You need to declare the lexical analyzer 'yylex' and the error printer 'yyerror' here, along with any other global identifiers used by the actions in the grammar rules.

The Bison declarations declare the names of the terminal and nonterminal symbols, and may also describe operator precedence and the data types of semantic values of various symbols.

The grammar rules define how to construct each nonterminal symbol from its parts.

The epilogue can contain any code you want to use. Often the definitions of functions declared in the prologue go here. In a simple program, all the rest of the program can go here." [37]

#### 3.1.2.6 An example bison file (infix calculator)

The example that follows is an Infix notation calculator.

"Here is the Bison code for 'calc.y', an infix desk-top calculator.

```
/* Infix notation calculator. */

%{
  #include <math.h>
  #include <stdio.h>
  int yylex (void);
  void yyerror (char const *);
%}
```

```
/* Bison declarations. */
%define api.value.type {double}
%token NUM
%left '-' '+'
%left '*' '/'
%precedence NEG   /* negation--unary minus */
%right '^'       /* exponentiation */

%% /* The grammar follows. */
input:
  %empty
| input line
;

line:
  '\n'
| exp '\n'  { printf ("\t%.10g\n", $1); }
;

exp:
  NUM
| exp '+' exp       { $$ = $1 + $3;     }
| exp '-' exp       { $$ = $1 - $3;     }
| exp '*' exp       { $$ = $1 * $3;     }
| exp '/' exp       { $$ = $1 / $3;     }
| '-' exp  %prec NEG { $$ = -$2;         }
| exp '^' exp       { $$ = pow ($1, $3); }
| '(' exp ')'       { $$ = $2;          }
;
```

```
%%
```

In the second section (Bison declarations), ‘%left’ declares token kinds and says they are left-associative operators. The declarations ‘%left’ and ‘%right’ (right associativity) take the place of ‘%token’ which is used to declare a token kind name without associativity/precedence. (These tokens are single-character literals, which ordinarily do not need to be declared. We declare them here to specify the associativity/precedence.)

Operator precedence is determined by the line ordering of the declarations; the higher the line number of the declaration (lower on the page or screen), the higher the precedence. Hence, exponentiation has the highest precedence, unary minus (‘NEG’) is next, followed by ‘*’ and ‘/’, and so on. Unary minus is not associative, only precedence matters (‘%precedence’.

The other important new feature is the ‘%prec’ in the grammar section for the unary minus operator. The ‘%prec’ simply instructs Bison that the rule ‘| '-' exp’ has the same precedence as ‘NEG’—in this case the next-to-highest.

Here is a sample run of ‘calc.y’:

```
$ calc
4 + 4.5 - (34/(8*3+-3))
6.880952381
-56 + 2
-54
3 ^ 2
9
" [37]
```

### 3.1.3 The files csense consists of

The files the project consists of and their dependencies are shown in the image below.

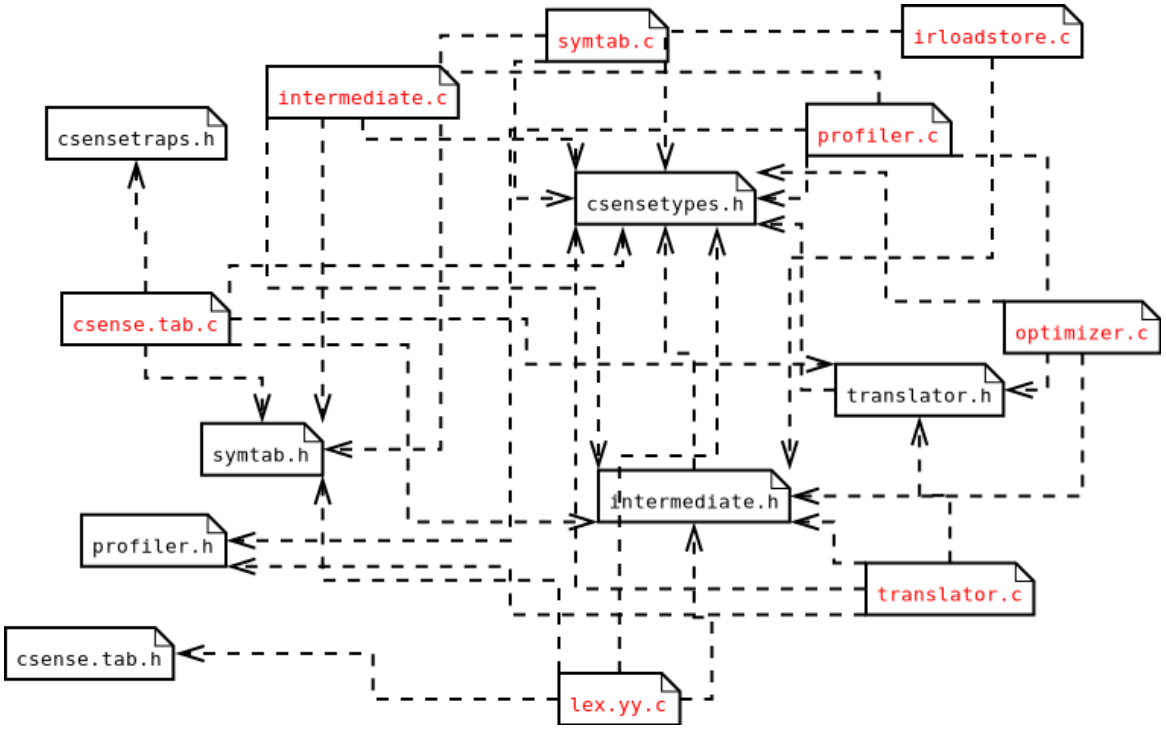


**Image 22. csense "include" dependencies**

The "main" function is in translator.c.

translator.c is where the IR (AST) gets converted to Ada code (making appropriate checks along the way.)

csensetypes.h defines types identified by the compiler.

intermediate.h defines expressions, instructions and the like.

translator.h is where Ada types are defined.

profiler.h is where command "statistics" profiling types are defined.

profiler.c is where functions for the profiling (mentioned in profiler.h's description) are implemented.

symtab.h is where the types for the symbol table are defined.

symtab.c contains functions for interacting with the symbol table.

intermediate.c contains functions for creating, deleting and modifying the IR (instructions, expressions, etc).

irloadstore.c contains functions for storing/loading the IR into/from a file.

optimizer.c contains functions mainly for applying optimizations to the IR. A lot of functions are used for checks and tasks used in the optimization functions that are repeated throughout multiple optimizations.

csensetraps.h (This file does not do anything relevant or noteworthy in respect to this document)

csense.tab.c and csense.tab.h are generated by running "bison -d csense.y".

lex.yy.c is generated by running "flex csense.l"

To compile the project with gcc (the gnu c compiler), run "gcc *.c -lfl" in the project folder.

### 3.1.4 csense limitations

Not all C and Ada features are supported by csense. The features that are NOT supported are listed below:

- Function pointers.
- Unnamed functions.
- Varargs.
- Unions.
- Strings (including string constants).
- Non-null pointer constants with non-pointer values.
- Shift operations.
- Type casting.
- Subprogram access types.
- Unnamed subprograms.
- Indexing an access object.
- The "goto" statement.
- Any of the '#' directives ('#' is not even recognized as a token, hence no "#include", "#ifndef", etc is considered legal.)

### 3.1.5 csense optimizations

csense supports several optimization options:

"

optimization options:

-ose : simplify and compress expressions

-ocbe=n : compress boolean expressions of up to n levels

-ocie=m : compress integer expressions of up to m levels

-ouil l1 l2 […] : unroll inner loop

-ofuil l1 l2 […]: fully unroll inner loop

-ouilr l1 l2 […]: unroll inner loop and reorder lines

-ofuilr l1 l2 […]: fully unroll inner loop and reorder lines

-ocm : code motion

-olp : loop pipeline

-olprcm : loop pipeline with code motion and recurrence duplication

-olpm : loop pipeline with minimal prologue/epilogue

-olpmrcm : loop pipeline with minimal prologue/epilogue, code motion and recurrence duplication

"

Where “n” and “m” can be constants and “l1”, “l2” loop candidate indexes (which can be omitted. In such a case the unrolling option in question is implemented on all the candidates that have not been mentioned already).

(Since code motion and loop pipelining have already been described in brief in the second chapter of this document and are not its focus, no examples will be provided in this section concerning these.)

#### 3.1.5.1 The “simplify and compress expressions” optimization

The -ose ("simplify and compress expressions") option divides expressions into ones with no more than two operands if no other option is specified. A temporary variable is generated for reassignment. An example is provided to illustrate the difference:

C input:

"

```
int main(void)
{
    int a;

    a = 5 - 8 * 4 + 3;
    a = 4 + a + 5;
```

```
    return 0;
}
"

Ada output without "-ose":
"
------------------------------------------------------------
------------------  C front-end translator  ------------------
--------------------- CCC Compiler Group ----------------------
------------------------------------------------------------

package from_c_program1 is

  function main
        return INTEGER;

end from_c_program1;

------------------------------------------------------------

package body from_c_program1 is

  function main
        return INTEGER is
    V001_a: INTEGER;
  begin
    V001_a := -24;
    V001_a := (4 + V001_a) + 5;
    return 0;
  end main;
```

```
end from_c_program1;
"

Ada output with "-ose":
"
------------------------------------------------------------
------------------  C front-end translator  -------------------
---------------------- CCC Compiler Group ----------------------
------------------------------------------------------------

------------------------------------------------------------
-------------------  C front-end optimizer  -------------------
---------------------- CCC Compiler Group ----------------------
------------------------------------------------------------
----    option -ose     : simplify expressions  ---------------
------------------------------------------------------------

package from_c_program1 is

  function main
        return INTEGER;

end from_c_program1;

------------------------------------------------------------

package body from_c_program1 is

  function main
        return INTEGER is
```

```
      V001_a: INTEGER;
      OPT_TEMP_000: INTEGER;
    begin
      V001_a := -24;
      OPT_TEMP_000 := 4 + V001_a;
      V001_a := OPT_TEMP_000 + 5;
      return 0;
    end main;
  end from_c_program1;
  "
```

(Note that constant folding has been automatically applied to the first assignment to "a" in both cases, even though no other optimization option has been requested.)

The -ocbe=n (“compress boolean expressions of up to n levels”) and -ocie=m (“compress integer expressions of up to m levels”) options control the amount of instructions of the respective types that can appear in one instruction. The depth level can be the same or independent and can apply to expressions with mixed instruction types.

Below are a couple of examples:

C input:

```
"
int main(int argc)
{
    int a = 5, b = 10, c = 15;
    int bool_expr, int_expr;

    bool_expr = (2 && (!b)) || (c && (a || 0));
    int_expr = 1 == ((-a > (b + c - 5)) != (7 <= c));

    return 0;
}
"
```

Ada output with -ocbe=3 and -ocie=2:

"

```
------------------------------------------------------------
------------------  C front-end translator  -------------------
---------------------- CCC Compiler Group ----------------------
------------------------------------------------------------
----    option -noshort : turn off short-circuit logic mode  --
----    option -msize=10000   : set external memory size to  --
----                    10000   byte elements  --------------
------------------------------------------------------------


------------------------------------------------------------
-------------------  C front-end optimizer  -------------------
---------------------- CCC Compiler Group ----------------------
------------------------------------------------------------
----    option -ose     : simplify and compress expressions  --
----    option -ocbe=3  : compress boolean expressions of up --
----                    to 3 levels
----    option -ocie=2  : compress integer expressions of up --
----                    to 2 levels
------------------------------------------------------------


package test1_separate_bool_int_28092022 is

  function main (
           P01_argc: in INTEGER)
         return INTEGER;

end test1_separate_bool_int_28092022;
```

```
------------------------------------------------------------

package body test1_separate_bool_int_28092022 is

  function main (
          P01_argc: in INTEGER)
        return INTEGER is
    V002_a: INTEGER;
    V003_b: INTEGER;
    V004_c: INTEGER;
    V005_bool_expr: BOOLEAN;
    V006_int_expr: BOOLEAN;
    OPT_TEMP_000: INTEGER;
    OPT_TEMP_001: INTEGER;
    OPT_TEMP_002: BOOLEAN;
    OPT_TEMP_003: BOOLEAN;
    TEMPORARY0000: INTEGER;
    TEMPORARY0001: INTEGER;
    TEMPORARY0002: BOOLEAN;
    TEMPORARY0003: BOOLEAN;
  begin
    V002_a := 5;
    TEMPORARY0000 := 5;
    V003_b := 10;
    TEMPORARY0001 := 10;
    V004_c := 15;
    TEMPORARY0002 := false;
    V005_bool_expr := (2 and (not V003_b)) or (V004_c and (V002_a or
TEMPORARY0002));
    OPT_TEMP_000 := -V002_a;
    OPT_TEMP_001 := (V003_b + V004_c) - 5;
```

```
      OPT_TEMP_002 := OPT_TEMP_000 > OPT_TEMP_001;
      OPT_TEMP_003 := 7 <= V004_c;
      TEMPORARY0003 := true;
      V006_int_expr := TEMPORARY0003 = (OPT_TEMP_002 /= OPT_TEMP_003);
      return 0;
    end main;
  end test1_separate_bool_int_28092022;
  "
  C input:
  "
  int main(int argc)
  {
      int a = 5, b = 10, c = 15;
      int d, e;

      d = ((!((32 + a - 89) == -(17 - c)) && ((24 + 27) < (c - a))) == !((-b) + 9)) || !(!((5 >=
(3 - (-4))) != 8) && ((12 < (15 - 2)) == ((10 - 9) + b)));
      e = ((75 == c) > ((95 || a) != b)) && ((57 < a) != (b > ((c - a) > 43)));

      return 0;
  }
  "

  Ada output with -ocbe=2 and -ocie=3:
  "
  ------------------------------------------------------------
  ------------------  C front-end translator  -------------------
  ---------------------- CCC Compiler Group ----------------------
  ------------------------------------------------------------
  ----    option -noshort : turn off short-circuit logic mode  --
  ----    option -msize=10000   : set external memory size to  --
```

```
----                  10000   byte elements  ----------
-------------------------------------------------------------


-------------------------------------------------------------
------------------  C front-end optimizer  ------------------
--------------------- CCC Compiler Group ---------------------
-------------------------------------------------------------
----    option -ose     : simplify and compress expressions  --
----    option -ocbe=2  : compress boolean expressions of up --
----                  to 2 levels
----    option -ocie=3  : compress integer expressions of up --
----                  to 3 levels
-------------------------------------------------------------

package test3_combined_consts_30092022 is

  function main (
           P01_argc: in INTEGER)
         return INTEGER;

end test3_combined_consts_30092022;

-------------------------------------------------------------

package body test3_combined_consts_30092022 is

  function main (
           P01_argc: in INTEGER)
         return INTEGER is
    V002_a: INTEGER;
```

```
   V003_b: INTEGER;
   V004_c: INTEGER;
   V005_d: BOOLEAN;
   V006_e: BOOLEAN;
   OPT_TEMP_000: BOOLEAN;
   OPT_TEMP_001: BOOLEAN;
   OPT_TEMP_002: INTEGER;
   OPT_TEMP_003: BOOLEAN;
   OPT_TEMP_004: BOOLEAN;
   OPT_TEMP_005: BOOLEAN;
   OPT_TEMP_006: BOOLEAN;
   OPT_TEMP_007: BOOLEAN;
   OPT_TEMP_008: BOOLEAN;
   TEMPORARY0000: INTEGER;
   TEMPORARY0001: INTEGER;
   TEMPORARY0002: BOOLEAN;
begin
   V002_a := 5;
   TEMPORARY0000 := 5;
   V003_b := 10;
   TEMPORARY0001 := 10;
   V004_c := 15;
   OPT_TEMP_000 := ((32 + V002_a) - 89) = (- (17 - V004_c));
   OPT_TEMP_001 := (not OPT_TEMP_000) and (51 < (V004_c - V002_a));
   OPT_TEMP_002 := (- V003_b) + 9;
   OPT_TEMP_003 := OPT_TEMP_002 = 0;
   OPT_TEMP_004 := 1 = (1 + V003_b);
   TEMPORARY0002 := false;
   OPT_TEMP_005 := TEMPORARY0002 and OPT_TEMP_004;
   V005_d := (OPT_TEMP_001 = OPT_TEMP_003) or (not OPT_TEMP_005);
```

```
        OPT_TEMP_006 := 75 = V004_c;
        OPT_TEMP_007 := (95 or V002_a) /= V003_b;
        OPT_TEMP_008 := (V004_c - V002_a) > 43;
        V006_e := (OPT_TEMP_006 > OPT_TEMP_007) and ((57 < V002_a) /= (V003_b >
OPT_TEMP_008));
        return 0;
      end main;
    end test3_combined_consts_30092022;
    "
```

#### 3.1.5.2 The loop unrolling optimizations

There are four options for loop unrolling given by csense's optimizer as of the writing of this text.

The first (-ouil) is the simplest, it will make up to four copies of the loop body.

The second (-ofuil) will unroll the body up to ten times. If the iterations happen to be less, the loop structure will be eradicated and modifications of the index will be replaced by their final value (this is especially desirable if the code is to be translated to a language such as VHDL and printed to a circuit board (a PCB) as this will mean less inessential circuitry on the board).

The third (-ouilr) will reorder the instructions in the unrolled body so the scheme looks like "a1, a2, a3, b1, b2, b3" instead of the default "a1, b1, a2, b2, a3, b3", as long as the index is not used. (This is still a work in progress; checks to confirm that there are no write-after-read(WAR), read-after-write(RAW) or write-after-write(WAW) dependencies within the body have not been implemented yet.)

The final option (-ofuilr) combines reordering and full loop unrolling, although the same restrictions on the third (-ouilr) options apply (so if the index is used at all, it falls back to the second option (-ofuil)).

Below are a few demonstrations:

Given C input:

```
"
int main(void)
```

```
{
    int a[50][50], b[50], i, j;
    for (i = 4; i < 50; i++ ) {
            b[i] = 1;
            for (j = 2; j < 9; j++ ) {
                    a[i][j] = 5;
            }
    }
    return 0;
}
"
```

Ada output (no optimizations):

```
"
------------------------------------------------------------
------------------  C front-end translator  -------------------
---------------------- CCC Compiler Group ----------------------
------------------------------------------------------------

package from_c_program3 is

  type TYPE000 is array (0..49) of INTEGER;
  type TYPE001 is array (0..49) of TYPE000;

  function main
        return INTEGER;

end from_c_program3;

------------------------------------------------------------
```

```
package body from_c_program3 is

  function main
        return INTEGER is
    V003_i: INTEGER;
    V004_j: INTEGER;
    GV000_V001_a: TYPE001;
    GV001_V002_b: TYPE000;
    INDEX000: INTEGER;
    TEMPINT000: INTEGER;
    INDEX001: INTEGER;
    TEMPINT001: INTEGER;
  begin
    V003_i := 4;
    TEMPINT000 := 45;
    for INDEX000 in 0..TEMPINT000 loop
      GV001_V002_b(V003_i) := 1;
      V004_j := 2;
      TEMPINT001 := 6;
      for INDEX001 in 0..TEMPINT001 loop
        GV000_V001_a(V003_i)(V004_j) := 5;
        V004_j := V004_j + 1;
      end loop;
      V003_i := V003_i + 1;
    end loop;
    return 0;
  end main;
end from_c_program3;
"
```

Ada output (-ouil, ordinary loop unrolling):

"

```
------------------------------------------------------------
------------------  C front-end translator  -------------------
---------------------- CCC Compiler Group ----------------------
------------------------------------------------------------


------------------------------------------------------------
-------------------  C front-end optimizer  -------------------
---------------------- CCC Compiler Group ----------------------
------------------------------------------------------------
----    option -ouil    : unroll inner loop  ------------------
------------------------------------------------------------

package from_c_program3_ouil is

  type TYPE000 is array (0..49) of INTEGER;
  type TYPE001 is array (0..49) of TYPE000;

  function main
        return INTEGER;

end from_c_program3_ouil;

------------------------------------------------------------

package body from_c_program3_ouil is

  function main
```

```
      return INTEGER is
   V003_i: INTEGER;
   V004_j: INTEGER;
   OPT_TEMP_000: INTEGER;
   GV000_V001_a: TYPE001;
   GV001_V002_b: TYPE000;
   INDEX000: INTEGER;
   TEMPINT000: INTEGER;
   INDEX001: INTEGER;
   TEMPINT001: INTEGER;
begin
   V003_i := 4;
   TEMPINT000 := 45;
   for INDEX000 in 0..TEMPINT000 loop
      GV001_V002_b(V003_i) := 1;
      V004_j := 2;
      TEMPINT001 := 0;
      for INDEX001 in 0..TEMPINT001 loop
         OPT_TEMP_000 := V004_j;
         GV000_V001_a(V003_i)(OPT_TEMP_000) := 5;
         OPT_TEMP_000 := V004_j + 1;
         GV000_V001_a(V003_i)(OPT_TEMP_000) := 5;
         OPT_TEMP_000 := V004_j + 2;
         GV000_V001_a(V003_i)(OPT_TEMP_000) := 5;
         OPT_TEMP_000 := V004_j + 3;
         GV000_V001_a(V003_i)(OPT_TEMP_000) := 5;
         V004_j := V004_j + 4;
      end loop;
      TEMPINT001 := 8 - V004_j;
      for INDEX001 in 0..TEMPINT001 loop
```

```
          GV000_V001_a(V003_i)(V004_j) := 5;
          V004_j := V004_j + 1;
        end loop;
        V003_i := V003_i + 1;
      end loop;
      return 0;
    end main;
end from_c_program3_ouil;
"

Ada output (-ofuil, full loop unrolling):
"
------------------------------------------------------------
------------------  C front-end translator  -------------------
---------------------- CCC Compiler Group ----------------------
------------------------------------------------------------

------------------------------------------------------------
-------------------  C front-end optimizer  -------------------
---------------------- CCC Compiler Group ----------------------
------------------------------------------------------------
----    option -ouil    : unroll inner loop  ------------------
----    option -ofuil   : fully unroll inner loop  ------------
------------------------------------------------------------

package from_c_program3_ofuil is

  type TYPE000 is array (0..49) of INTEGER;
  type TYPE001 is array (0..49) of TYPE000;
```

```
   function main
         return INTEGER;

end from_c_program3_ofuil;

------------------------------------------------------------

package body from_c_program3_ofuil is

   function main
         return INTEGER is
      V003_i: INTEGER;
      V004_j: INTEGER;
      GV000_V001_a: TYPE001;
      GV001_V002_b: TYPE000;
      INDEX000: INTEGER;
      TEMPINT000: INTEGER;
   begin
      V003_i := 4;
      TEMPINT000 := 45;
      for INDEX000 in 0..TEMPINT000 loop
         GV001_V002_b(V003_i) := 1;
         V004_j := 2;
         GV000_V001_a(V003_i)(2) := 5;
         GV000_V001_a(V003_i)(3) := 5;
         GV000_V001_a(V003_i)(4) := 5;
         GV000_V001_a(V003_i)(5) := 5;
         GV000_V001_a(V003_i)(6) := 5;
         GV000_V001_a(V003_i)(7) := 5;
         GV000_V001_a(V003_i)(8) := 5;
```

```
        V004_j := V004_j + 7;
        V003_i := V003_i + 1;
      end loop;
      return 0;
    end main;
end from_c_program3_ofuil;
"
```

To demonstrate the reordering options (-ouilr and -ofuilr) a different input file is going to be used, since the use of the index variable in the previous example would cause the options to fall back to keeping the standard order of the unrolled loop body.

Given C input:

```
"
int main(void)
{
    int x, y, z, i, j;
    for (i = 4; i < 50; i++ ) {
            for (j = 2; j < 19; j++ ) {
                    x = 1;
                    y = 2;
                    z = 3;
            }
    }
    return 0;
}
"
```

Ada output (-ouil, ordinary loop unrolling):

```
"
------------------------------------------------------------
```

```
------------------  C front-end translator  --------------------
---------------------- CCC Compiler Group ----------------------
-------------------------------------------------------------


-------------------------------------------------------------
------------------  C front-end optimizer  --------------------
---------------------- CCC Compiler Group ----------------------
-------------------------------------------------------------
----    option -ouil    : unroll inner loop  ------------------
-------------------------------------------------------------

package from_c_program5_ouil is

  function main
        return INTEGER;

end from_c_program5_ouil;

-------------------------------------------------------------

package body from_c_program5_ouil is

  function main
        return INTEGER is
    V001_x: INTEGER;
    V002_y: INTEGER;
    V003_z: INTEGER;
    V004_i: INTEGER;
    V005_j: INTEGER;
    INDEX000: INTEGER;
```

```
   TEMPINT000: INTEGER;
   INDEX001: INTEGER;
   TEMPINT001: INTEGER;
 begin
   V004_i := 4;
   TEMPINT000 := 45;
   for INDEX000 in 0..TEMPINT000 loop
     V005_j := 2;
     TEMPINT001 := 3;
     for INDEX001 in 0..TEMPINT001 loop
       V001_x := 1;
       V002_y := 2;
       V003_z := 3;
       V001_x := 1;
       V002_y := 2;
       V003_z := 3;
       V001_x := 1;
       V002_y := 2;
       V003_z := 3;
       V001_x := 1;
       V002_y := 2;
       V003_z := 3;
       V005_j := V005_j + 4;
     end loop;
     TEMPINT001 := 18 - V005_j;
     for INDEX001 in 0..TEMPINT001 loop
       V001_x := 1;
       V002_y := 2;
       V003_z := 3;
       V005_j := V005_j + 1;
```

```
        end loop;
        V004_i := V004_i + 1;
      end loop;
      return 0;
   end main;
end from_c_program5_ouil;
"

Ada output (-ouilr, unroll inner loops and reorder instructions in the body):
"
------------------------------------------------------------
------------------  C front-end translator  ------------------
---------------------- CCC Compiler Group ----------------------
------------------------------------------------------------

------------------------------------------------------------
------------------  C front-end optimizer  ------------------
---------------------- CCC Compiler Group ----------------------
------------------------------------------------------------
----    option -ouil    : unroll inner loop  ------------------
----    option -ouilr   : unroll inner loop and reorder  ------
------------------------------------------------------------

package from_c_program5_ouilr is

   function main
         return INTEGER;

end from_c_program5_ouilr;
```

```
------------------------------------------------------------

package body from_c_program5_ouilr is

  function main
        return INTEGER is
    V001_x: INTEGER;
    V002_y: INTEGER;
    V003_z: INTEGER;
    V004_i: INTEGER;
    V005_j: INTEGER;
    INDEX000: INTEGER;
    TEMPINT000: INTEGER;
    INDEX001: INTEGER;
    TEMPINT001: INTEGER;
  begin
    V004_i := 4;
    TEMPINT000 := 45;
    for INDEX000 in 0..TEMPINT000 loop
      V005_j := 2;
      TEMPINT001 := 3;
      for INDEX001 in 0..TEMPINT001 loop
        V001_x := 1;
        V001_x := 1;
        V001_x := 1;
        V001_x := 1;
        V002_y := 2;
        V002_y := 2;
        V002_y := 2;
        V002_y := 2;
```

```
          V003_z := 3;
          V003_z := 3;
          V003_z := 3;
          V003_z := 3;
          V005_j := V005_j + 4;
        end loop;
        TEMPINT001 := 18 - V005_j;
        for INDEX001 in 0..TEMPINT001 loop
          V001_x := 1;
          V002_y := 2;
          V003_z := 3;
          V005_j := V005_j + 1;
        end loop;
        V004_i := V004_i + 1;
      end loop;
      return 0;
    end main;
end from_c_program5_ouilr;
"
```

Ada output (-ofuilr, fully unroll inner loops and reorder instructions in the body):

```
"
------------------------------------------------------------
------------------  C front-end translator  -------------------
---------------------- CCC Compiler Group ----------------------
------------------------------------------------------------

------------------------------------------------------------
------------------  C front-end optimizer  -------------------
---------------------- CCC Compiler Group ----------------------
```

```
-------------------------------------------------------------
----    option -ouil    : unroll inner loop  ------------------
----    option -ouilr   : unroll inner loop and reorder  ------
----    option -ofuil   : fully unroll inner loop  ------------
-------------------------------------------------------------

package from_c_program5_ofuilr is

  function main
        return INTEGER;

end from_c_program5_ofuilr;

-------------------------------------------------------------

package body from_c_program5_ofuilr is

  function main
        return INTEGER is
    V001_x: INTEGER;
    V002_y: INTEGER;
    V003_z: INTEGER;
    V004_i: INTEGER;
    V005_j: INTEGER;
    INDEX000: INTEGER;
    TEMPINT000: INTEGER;
    INDEX001: INTEGER;
    TEMPINT001: INTEGER;
  begin
    V004_i := 4;
```

```
TEMPINT000 := 45;
for INDEX000 in 0..TEMPINT000 loop
  V005_j := 2;
  V001_x := 1;
  V001_x := 1;
  V001_x := 1;
  V001_x := 1;
  V001_x := 1;
  V001_x := 1;
  V001_x := 1;
  V001_x := 1;
  V001_x := 1;
  V001_x := 1;
  V002_y := 2;
  V002_y := 2;
  V002_y := 2;
  V002_y := 2;
  V002_y := 2;
  V002_y := 2;
  V002_y := 2;
  V002_y := 2;
  V002_y := 2;
  V002_y := 2;
  V002_y := 2;
  V003_z := 3;
  V003_z := 3;
  V003_z := 3;
  V003_z := 3;
  V003_z := 3;
  V003_z := 3;
  V003_z := 3;
```

```
        V003_z := 3;
        V003_z := 3;
        V003_z := 3;
        V005_j := V005_j + 10;
        TEMPINT001 := 18 - V005_j;
        for INDEX001 in 0..TEMPINT001 loop
          V001_x := 1;
          V002_y := 2;
          V003_z := 3;
          V005_j := V005_j + 1;
        end loop;
        V004_i := V004_i + 1;
      end loop;
      return 0;
    end main;
  end from_c_program5_ofuilr;
  "
```

## 3.2 csense's loop unrolling function

(Most work has been done by Tolis Tsakiridis & Giorgos Chatzianastasiou. The reordering section and the modifications to implement full unrolling have been done by the author of this document.)

The first thing the loop unrolling function does when called is count the number of inner "for" loops that might be candidates for unrolling. For every candidate it finds, it first checks the second expression: if the expression is not of a form similar to "variable < INTEGER" it skips the candidate and moves on to the next one. The same action is taken if the subsequent first expression check indicates that an assignment to the index variable is missing, or if the third expression check is not of the form "++index_variable" or "index_variable++". A fourth check will skip to the next candidate if the index variable is modified within the loop body.

After this, the unroll factor is determined. If the assignment in the first expression of the for loop to the index variable is not a constant value, the unroll factor is set to 4 if full unrolling is not requested or to the constant "MAX_UNROLL_FACTOR" (10 as of the writing of this document) otherwise. If it is, however, there will be a few more checks following. The candidate might still be skipped if there are less than or equal to one

iteration in the loop. If the iterations are less than four, then the full unrolling flag is set regardless of whether it was requested or not. A "true full unroll" flag will also be set to "on". The "remain" flag will be set to "off", so no second rolled loop will be appended after the unrolled loop for the iterations that remain (the loop in its initial form minus the amount of iterations that were run unrolled). Else, if full unrolling has not been requested, the unrolling factor is set to four and the remain flag is set to "off" if the iteration count modulus the unroll factor results to zero. Else, if the iterations are more than the "MAX_UNROLL_FACTOR" constant, then there is a nested check: if the "MAX_UNROLL_FACTOR" constant is larger than the iterations minus itself (the "MAX_UNROLL_FACTOR") then the "true full unroll" flag is set to "on". After this nested check but within the same "else" (there is no "else" after the nested check) the unroll factor is set to "MAX_UNROLL_FACTOR" and the remain flag is set to "off" if the iteration count modulus the unroll factor results to zero. Else, finally, we have the true full unrolling case: the "true full unroll" flag is set to "on", the unroll factor is set to the iterations, and the "remain" flag is set to "off".

The next check is whether the "expression simplification" flag is "on"; if it is not, then nested and compound writes as well as comma expressions are simplified.

After this, a check is performed in case the second expression in the loop does not have a constant as a right child (e.g. it is "var < other_var" instead of "var < CONST"); dependencies are analyzed for the loop candidate and if the "abort" flag is set along the way, the loop candidate is skipped. This is the final check where the loop candidate might be skipped, so beyond this point the AST begins to get modified.

If the "remain" flag is "on", the loop and its body in their original form are copied in order to be appended to the unrolled loop (or the last instruction of the fully unrolled "loop"). The first expression is set to null for this copy.

The first expression to be modified in the original loop is the second one: the right child of the operator is set to what it is minus the unroll factor (minus one).

The third expression is then modified to become an "assign add" expression (e.g. "i++" becomes "i += unroll_factor").

Before implementing the actual unrolling of the body a check is made in case the index variable is used within it; if it is, a temporary variable is generated. Any modification to the index will first become an assignment to this variable, and this temporary variable will subsequently replace any appearance of the actual index variable within the body.

There are two different sections run depending on if reordering is requested (-ouilr or -ofuilr). If it is and the index is not used in the body, then the instructions in the body are counted. Following this, the first instruction in the body is copied to each place of an array for as many iterations there will be in the loop (the unroll factor). Then, for the amount of instructions in the body (that were counted earlier), for as many times as the

unroll factor the pointer to the current instruction is set to the array (mentioned in the previous sentence) the element of which is the index (used in the latest mentioned loop). The current pointer is then set to its next element (as it is a linked list) for as many times up to the current index of the instruction count loop. Outside of this last loop but still within the one up to the unroll factor, another two dimensional array, the first dimension of which is set to the instruction count loop index and the second of which is set to the unroll factor loop index, is set to the current pointer. Once the instruction count loop has finished, the current pointer is set to the very first element of the two dimensional array. To finish the reordering section, for as many times as the instruction count, for as many times as the unroll factor, the next element of the current pointer is set to point to the two dimensional array, the first dimension of which is set to the index of the instruction count loop and the second of which is set to the unroll factor loop index. The current pointer will subsequently be set to its next element (which was just set in the previous step). The reordering section has now finished, and the instructions should now be reordered.

The second section runs if either no reordering has been requested, or if the index variable is used. Thus, since we can not be certain that we are here solely because reordering has not been requested, there is another check in case it has been. If true, a warning is printed notifying us that reordering is not possible and that the simple versions (-ouil or -ofuil) will be implemented instead. After appropriate pointer variable initializations, the first instruction in the body is copied to each place of an array for as many iterations there will be in the loop (the unroll factor), much like in the reordering section. The current pointer is then set to the first instruction in the body (not the array with the duplicated instructions). If the "true full unroll" flag is "on" and the index is used, every appearance of the index variable is replaced by its initialization value (the right hand side of the first expression in the unroll candidate loop), a constant. (This is achieved by a function that recursively calls itself until it finds a variable type with a specific ID in the instruction; it then proceeds to make appropriate changes to the node.) Then, for as many times as the unroll factor (minus one, because the original already exists and does not need to be modified), a copy of the first instruction is assigned to a pointer variable. If the index is not used, the next element of what the current pointer points to is set to the pointer variable (mentioned in the previous sentence). The current pointer is then set to the last instruction and the unroll factor loop continues to its next iteration. So, if the index is used, if the "true full unroll" flag is set to "off", then an additional instruction is appended to the next element of the current pointer: an assignment of the index variable plus the constant that represents the index of the unroll factor loop to the temporary variable that was generated earlier (right before starting the unrolling). The next element of the current pointer is then set to the copy of the first instruction, and on every iteration along the way to setting the current pointer to the last element in the list a check and modification is made to each instruction to replace every appearance of the index variable with the temporary variable. Otherwise,

the case is that the index is used and the "true full unroll" flag is "on". A "current temporary" variable is set to the assignment to the temporary much like in the non true full unrolling case mentioned earlier. The index variable (which is of course on the right hand side of the assignment to the temporary) is then replaced by the constant in its initialization instruction, much like before entering the unroll factor loop. The assignment now has the form "temp = 1 + 2", two constants on the right hand side of the assignment, so they are simplified to simply one: the final value. (This is achieved by a function that works similar to the one used for variable-to-constant transformation, the main difference being that it replaces the sum node with a new constant node of the sum of the two constants it finds.) The current pointer (not the temporary current pointer) is then set to the next element of the array with copies of the first instructions of the loop body. Then, for every next element of the current pointer (as long as one exists) every appearance of the index variable in the instruction is replaced by the temporary variable (not the pointer to the current temporary variable). Finally, every appearance of the temporary variable is replaced by the right hand side of the assignment to the temporary variable, which should hold the constant. (Note that the actual assignment to the temporary variable instruction is never actually connected to the AST; it is skipped entirely and free'd, therefore not appearing in the final output.)

The loop body has now been unrolled, but there are a few more things that might be done before we exit the function: if the index variable is used within the body and the "true full unroll" flag is set to "off", an assignment instruction of the form "temp = index_variable" is generated and appended to the start of the loop body. For a final time, all appearances of the index variable within the body are replaced by the temporary variable. If these conditions are not true, the variables for the first and for the current pointer are set to the first instruction in the loop body.

Lastly, the "true full unroll" flag is checked again. If it is "on", a copy of the first and the third section of the "for" loop candidate is made (e.g. "for(<first>;<second>;<third>)"). After the first section has been copied, the pointer to the first instruction in the loop body is set to the new (copied "first") instruction. Its next element is then set to the first instruction in the for loop. Next, the current pointer is set to the (new) first instruction, and subsequently to the last instruction in the loop body. After this, the third section is copied. Once done, a check is made in case the "remain" flag is "on" (recall that this is set if there are any remaining iterations; that is, when the iterations divided with the unroll factor, do not have a remainder of zero). If it is, the newly copied third expression is set to the loop candidate's next element (that was set before we started the unrolling itself). Regardless of the previous check, the next element of the current pointer (which at this point points to the last instruction in the loop body) is set to the copied third expression. Finally, the pointer to the first instruction in the loop is copied onto the loop candidate's node itself, thus eliminating

the loop structure entirely (which results in the fully unrolled loop body – without the actual loop – that can be seen in the examples above).

The following is a pseudocode generalization for the entire loop unrolling function:

"

for each loop candidate {

declare and initialize variables;

if the first or second or third expression is invalid or the index variable is modified within the body {

skip to the next candidate;

}

<determine unroll factor, if there are any iterations that will remain and if we have a case of true full unrolling>

divide compound instructions into one operation per instruction;

if the right hand side of the middle expression is not a constant and is modified within the body {

skip to the next candidate;

}

if any iterations will remain {

connect a copy of the loop with the appropriately modified iteration count as the next instruction after the original loop;

}

modify the limit of the second expression of the loop to be what it was minus the unroll factor minus one;

modify the third expression of the loop to become an assign-add (+=) expression with the unroll factor;

if the index variable is used within the loop body {

```
            generate a temporary variable;
        }

        if the index is not used and reordering has been requested {
            initialize variables;
            unroll the loop;
            reorder its instructions;
            reattach the instructions as the loop's new body;
        } else {
            if reordering has been requested {
                print warning that no reordering will be performed;
            }

            initialize variables;

            for each local index up to the unroll factor {
                if the index is not used {
                    unroll the loop and continue to the next local iteration;
                }

                if no true full unrolling is to be done {
                    unroll the loop while attaching an assignment of the
modification of the index variable to the temporary variable before the current copy of
the loop body;
                    replace every appearance of the index variable in the loop
body with the temporary variable;
                } else {
                    generate an assignment of the modification of the index
variable to the temporary variable;
                    replace the index variable in the instruction generated by
the above line with its initialization value;
```

replace the sum in the instruction modified in the above line with its result;

unroll the loop while replacing each appearance of the temporary variable with the right hand side of the instruction modified in the above three lines;

}

}

}

if the index variable is used within the body and we are not in the case of true full unrolling {

generate and attach an assignment instruction of the index variable to the temporary variable before the beginning of the loop body;

reinitialize local variables;

} else {

reinitialize local variables (in a different order);

}

if in the case of true full unrolling {

copy the first expression of the loop to the beginning of the loop body;

if there is another loop for the remaining iterations attached to the original loop, attach the (copied) third expression of the loop before it;

attach the last instruction in the loop body before the copied third expression;

copy the first instruction in the loop body (which is now the copied first expression) to the loop node (effectively keeping the body but eradicating the loop components. Whatever pointed to the loop node now points to the first instruction in the remaining body.);

}

}

"

Here is some more analytic pseudocode for the initialization of the two dimensional array of the reordering section:

"

```
for (k = 0; k <= commands_in_loop_count; k++) {
    for (j = 0; j < unroll_factor; j++) {
            set the current_pointer to the j'th element of the array of the copied first instructions;
            for (kk = 0; kk < k; kk++) { // This loop is to find the index of the instruction to be copied.
                    set the current_pointer to its next element;
            }
            reordered_instruction_table[k][j] = current_pointer;
    }
}
```

"

## 3.3 csense's expression compression functions

(Most work has been done by Dr. Georgios Dimitriou, finding a way to implement mixed integer and Boolean expressions was done by the compiler of this document.)

The parent function for the expression simplification function calls a function to simplify the instruction sequence for each subroutine it finds. This latter function then runs through a switch statement to check the type (expression, if, while, etc.) of the current instruction being processed and subsequently calls a function to simplify the expression tree with relevant checks and calls to instruction connection functions as required for each type case. It is this expression tree simplification function where most of the spadework is done. This function checks the type of the expression in an instruction with a switch statement and then, according to what type of node (operators, identifier, etc) it is dealing with, mostly interconnects instructions with the instruction returned by recursive calls to itself.

Specifically for the integer and Boolean operators: a call is made to a function to calculate the expression's full and original depth level (or "height"). The operators are then checked again, and if the calculated height is the same as the sole integer or Boolean height calculated by another similar function, we proceed to the actual simplification (the calls to connect instructions to recursive calls of the function being described) depending on the conditions which mostly in turn depend on the requested expression depth level by the user and the remaining (current) height of the function's

iteration. This latter height gets passed in and is decreased on every recursive call. At some point, we either return the current instruction generated and connected from a condition or we return a new, further simplified instruction generated once the switch statement ends.

For mixed expressions there are two parameters decreased on every call: integer and Boolean height. For expressions not involving one of the two types the non-relevant one is not decreased.

# 4. Results in the backend (VHDL generation from Ada input)

This chapter contains a few benchmarks with results from applying a range of combinations of loop unrolling and expression compression. The resulting .ada files were used as input to the CCC backend. Results were obtained between December 2022 and January 2023.

The benchmarking is done by measuring the time taken to run the post-implementation functional simulation in Xilinx Vivado 2018.3 in project mode with the Zedboard, an evaluation and development board based on the Xilinx Zynq-7000, as the target.

The constraints file (.xdc) used was a variation of the following for all tests:

"

create_clock -period 100.000 [get_ports clock]

#set_property IOSTANDARD LVDS_25 [get_ports *]

#set_property IOSTANDARD LVCMOS33 [get_ports *]

set_property IOSTANDARD LVTTL [get_ports *]

# clock, reset, start, results_read, busy, done  location constraints

set_property PACKAGE_PIN R18 [get_ports reset]

set_property PACKAGE_PIN R16 [get_ports start]

set_property PACKAGE_PIN P16 [get_ports results_read]

set_property PACKAGE_PIN T22 [get_ports busy]

set_property PACKAGE_PIN T21 [get_ports done]

set_property PACKAGE_PIN Y9 [get_ports clock]

set_property PACKAGE_PIN U22 [get_ports our_main]

set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets reset_IBUF]

set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets start_IBUF]

set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets results_read_IBUF]

set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets clock_IBUF]

"

Package pin Y9 is a clock pulse at 33.33MHz, according to the Zedboard Hardware User Guide, and so was the most sensible choice to map the clock to. Had the clock been mapped to a button, for example, we would need to repeatedly press the button to advance each clock cycle; which is infeasible for a count above 500.

Reset, start and results_read have been mapped to buttons to make testing on the hardware easier.

Busy, done and our_main have been mapped to LEDs.

## 4.1 First test

### 4.1.1 State count reduction

The PARCS optimizer in the backend reduces the amount of states in the original VHDL output. A theoretical advantage of expression compression is that states can be reduced further by the PARCS optimizer.

To test the output, the following c input was used (while the contents are the same, the filenames indicate the optimizations used to preserve results and prevent overwriting):

"

```
int main(int argc)
{
        int m2[50], a = 5, b = 10, c = 15, i, j;

        for (i = 4; i < 50; i++ ) {
                for (j = 2; j < 19; j++ ) {
                        m2[j] = a || ((b + c - (5 + a) > (a - c)) && (b || !c));
```

```
                }
        }

        return 0;
}
"
```

The unoptimized (by the backend) result of the output of running "./a.exe input_files/f_c_p7_loop1_no_opts.c" generates 40 states:

"

[...]

```
  SIGNAL done_int : std_logic;

  TYPE states_type IS (state_40,
                       state_39, state_38, state_37, state_36, state_35,
                       state_34, state_33, state_32, state_31, state_30,
                       state_29, state_28, state_27, state_26, state_25,
                       state_24, state_23, state_22, state_21, state_20,
                       state_19, state_18, state_17, state_16, state_15,
                       state_14, state_13, state_12, state_11, state_10,
                       state_9, state_8, state_7, state_6, state_5,
                       state_4, state_3, state_2, state_1, state_0);
  SIGNAL state : states_type;  -- this stores the current and next state of the circuit

  SIGNAL v002_m2 : Std_logic_vector(49 DOWNTO 0) ;
```

[...]

"

The backend PARCS optimized result of the output of running "./a.exe input_files/f_c_p7_loop1_no_opts.c" generates 18 states:

"

[...]

```
 SIGNAL done_int : std_logic;


   TYPE states_type IS (state_18,
                        state_17, state_16, state_15, state_14, state_13,
                        state_12, state_11, state_10, state_9, state_8,
                        state_7, state_6, state_5, state_4, state_3,
                        state_2, state_1, state_0);
   SIGNAL state : states_type;  -- this stores the current and next state of the circuit


   SIGNAL v002_m2 : Std_logic_vector(49 DOWNTO 0) ;
```

[...]

"

This is a 55% decrease.

We will now focus only on the PARCS optimized results' state increase/decrease in relation to the unoptimized input results shown above (18 states).

The PARCS optimized result of the output of running "./a.exe -noshort -ofuil input_files/f_c_p7_loop1_ofuil.c" generates 73 states: a 305.55% increase from the non optimized version. For reference, the simple non PARCS optimized result has 153 states.

The PARCS optimized result of the output of running "./a.exe -noshort -ocbe=2 -ocie=3 input_files/f_c_p7_loop1_ocbe2_ocie3.c" generates 11 states: a 38.88% decrease from the non optimized version. For reference, the simple non PARCS optimized result has 30 states.

The PARCS optimized result of the output of running "./a.exe -noshort -ofuil -ocbe=2 -ocie=3 input_files/f_c_p7_loop1_ofuil_ocbe2_ocie3.c" generates 62 states: a 244.44% increase from the non optimized version and 15.06% decrease from the compressionless version. For reference, the simple non PARCS optimized result has 131 states.

We can thus conclude that expression compression is quite an important aid to decreasing the amount of states in the VHDL output, which could prove crucial when combined with loop unrolling (which increases the amount of states).

### 4.1.2 Modification to the .vhd output

The output of each .vhd file was modified in order to accommodate for the fact that the entity generated returns a "main[31:0]" variable (OUT  std_logic_vector(31 DOWNTO 0)) that needs a pin mapping for the Zedboard. There are only 8 LEDs on the Zedboard and all outputs must be mapped to a pin. Normally we would output the final value to a multiplexer to solve this issue. However, since this "main[31:0]" variable which originated from the "return 0;" line in the .c file (which is required for it to compile properly with gcc) is only set in the "reset" section of the .vhd and never gets any other value, we can safely replace the "main" variable with an "our_main" variable (OUT std_logic) which turns on after the first run has finished. The changes are the following (output of "diff <original>.vhd <modified>.vhd"):

"

< main : OUT  std_logic_vector(31 DOWNTO 0);

---

> --main : OUT  std_logic_vector(31 DOWNTO 0); -- This is the original line

> our_main : OUT  std_logic; -- This line was added for pin mapping

163c164,165

< main <= (OTHERS => '0');

---

> --main <= (OTHERS => '0'); -- Original line

> our_main <= '0'; -- Line added for pin mapping

192c194

< IF results_read = '1' THEN done_int <= '0'; END IF;

---

> IF results_read = '1' THEN done_int <= '0'; our_main <= '1'; END IF;

"

This change is reflected in all the .vhd files presented.

### 4.1.3 Vivado post-implementation functional simulation timing results

#### 4.1.3.1 No optimizations

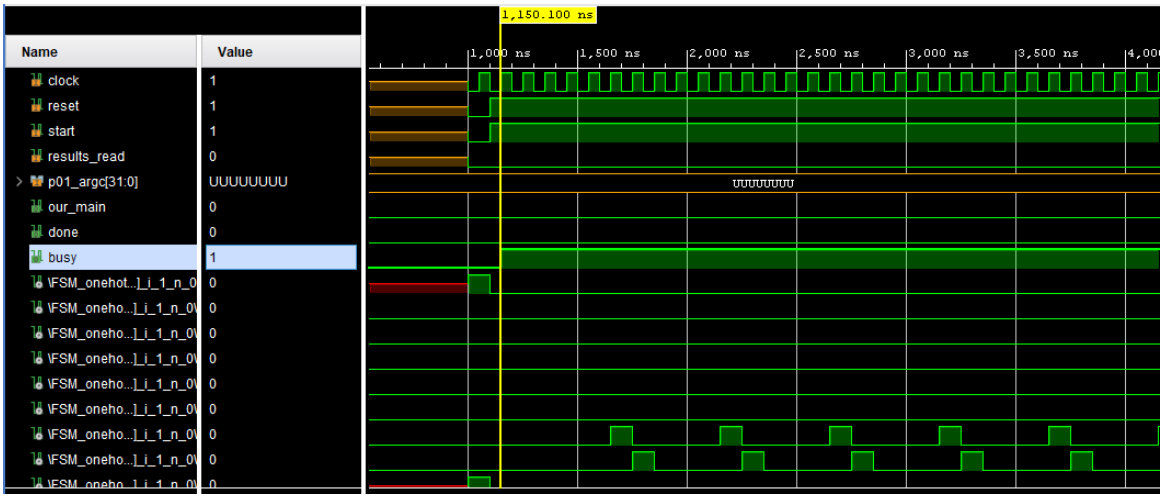


**Image 23.** First test no opts post-imp. func. simulation start

Πηγή: **(none)**

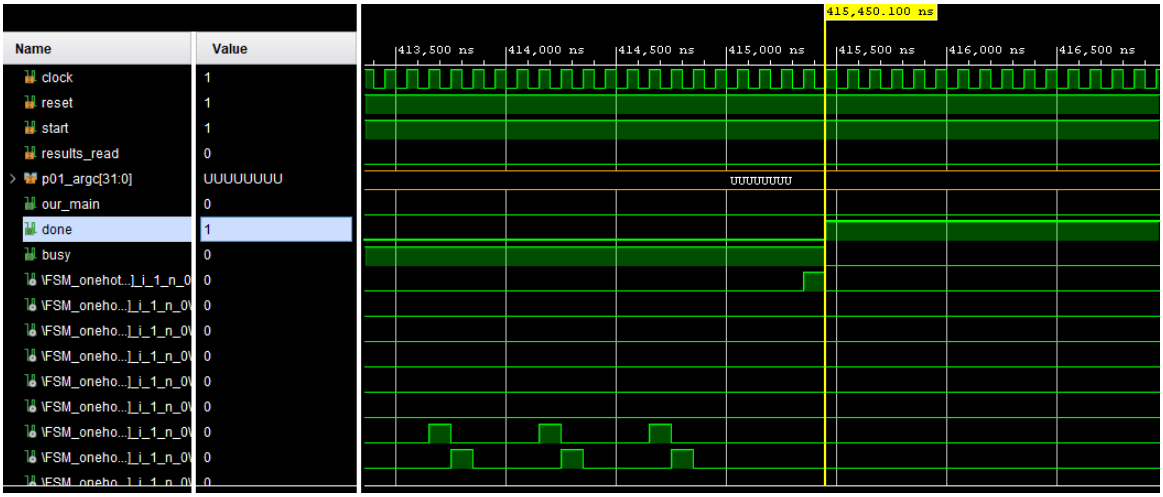


**Image 24.** First test no opts post-imp. func. simulation end

Πηγή: **(none)**

The backend PARCS optimized result of the output of running "./a.exe input_files/f_c_p7_loop1_no_opts.c" finished in 414300.0ns with a clock cycle of 100.0ns.

The tcl console commands used for the simulation were the following:

"

add_force clock {0 0ns} {1 50ns} -repeat_every 100ns

add_force reset {0 0ns} {1 100ns}

add_force start {0 0ns} {1 100ns}

add_force results_read {0 0ns}

run 300ns

run 400000ns

run 20000ns

add_force results_read {1 0ns}

"

(As can be seen, "reset" is active low.)

**Table 1. First test no opts. hardware utilization**

| Total Power | Failed Routes | LUT | FF | BRAMs | URAM | DSP |
|---|---|---|---|---|---|---|
| | | 115 | 107 | 0.00 | 0 | 0 |
| 0.11 | 0 | 115 | 107 | 0.00 | 0 | 0 |

Πηγή: (None)

| | |
|---|---|
| Worst Negative Slack (WNS): | 95.143 ns |
| Total Negative Slack (TNS): | 0 ns |
| Number of Failing Endpoints: | 0 |
| Total Number of Endpoints: | 201 |

**Image 25.** First test no opts. timing summary

Πηγή: **(none)**

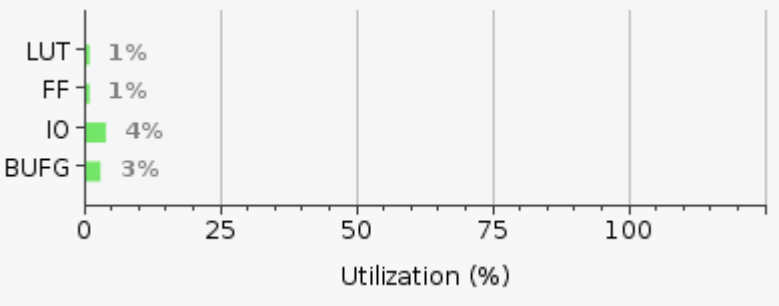


**Image 26.** First test no opts. utilization summary

Πηγή: **(none)**

### 4.1.3.2 With full unrolling

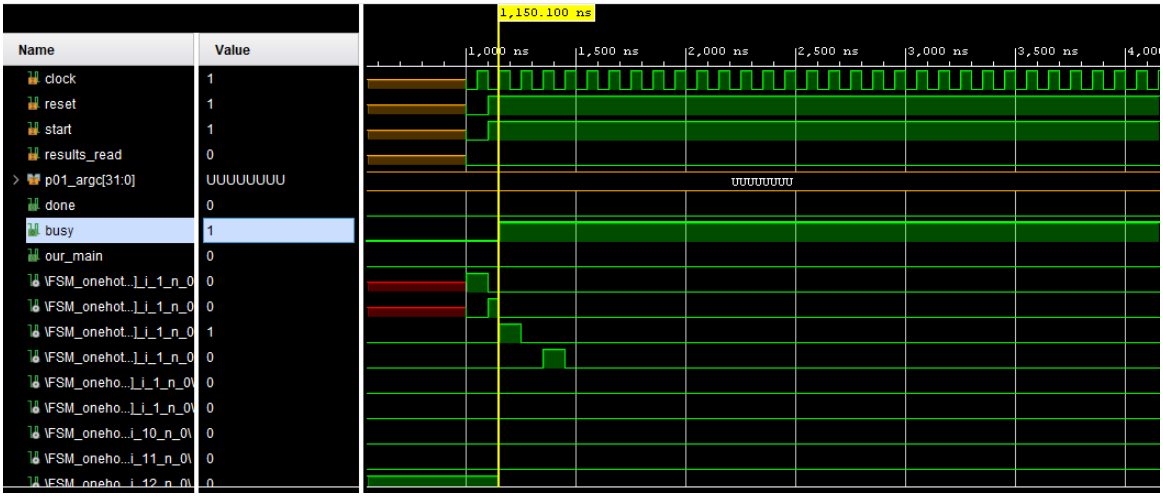


**Image 27.** First test full unr. post-imp. func. simulation start

Πηγή: **(none)**

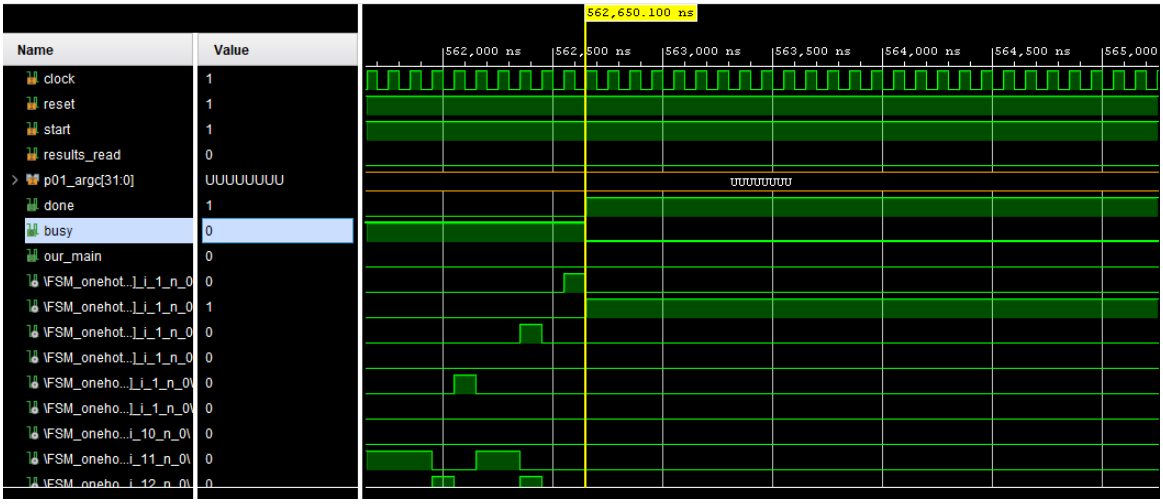


**Image 28.** First test full unr. post-imp. func. simulation end

Πηγή: **(none)**

The backend PARCS optimized result of the output of running "./a.exe -noshort -ofuil input_files/f_c_p7_loop1_ofuil.c" finished in 561500.0ns with a clock cycle of 100.0ns. This is a x0.7378 speedup from the original (so actually a slowdown). The tcl console commands used were the same as the ones in the non-optimized section bar the "run" commands which used different ns values.

**Table 2. First test ofuil hardware utilization**

| Total Power | Failed Routes | LUT | FF | BRAMs | URAM | DSP |
|---|---|---|---|---|---|---|
| | | 144 | 208 | 0.00 | 0 | 0 |
| 0.11 | 0 | 144 | 208 | 0.00 | 0 | 0 |

Πηγή: (None)

| | |
|---|---|
| Worst Negative Slack (WNS): | 94.234 ns |
| Total Negative Slack (TNS): | 0 ns |
| Number of Failing Endpoints: | 0 |
| Total Number of Endpoints: | 411 |

**Image 29.** First test ofuil timing summary

Πηγή: **(none)**

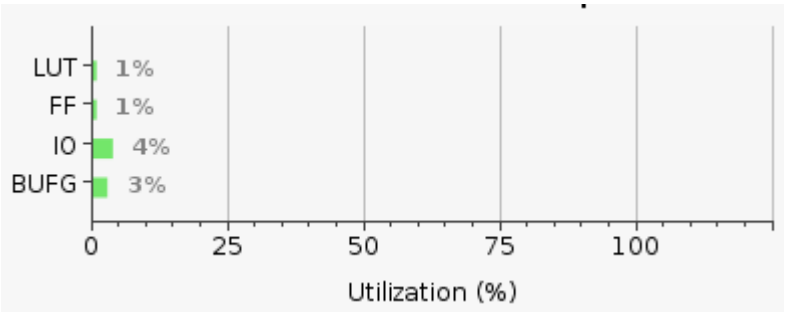


**Image 30.** First test ofuil utilization summary

Πηγή: **(none)**

#### 4.1.3.3 Plain expression compression (with a boolean depth of 2 and an integer depth of 3)

There were a few errors in the output of the CCC backend, according to Vivado, concerning assignments of std_logic types to std_logic_vectors and vice versa:

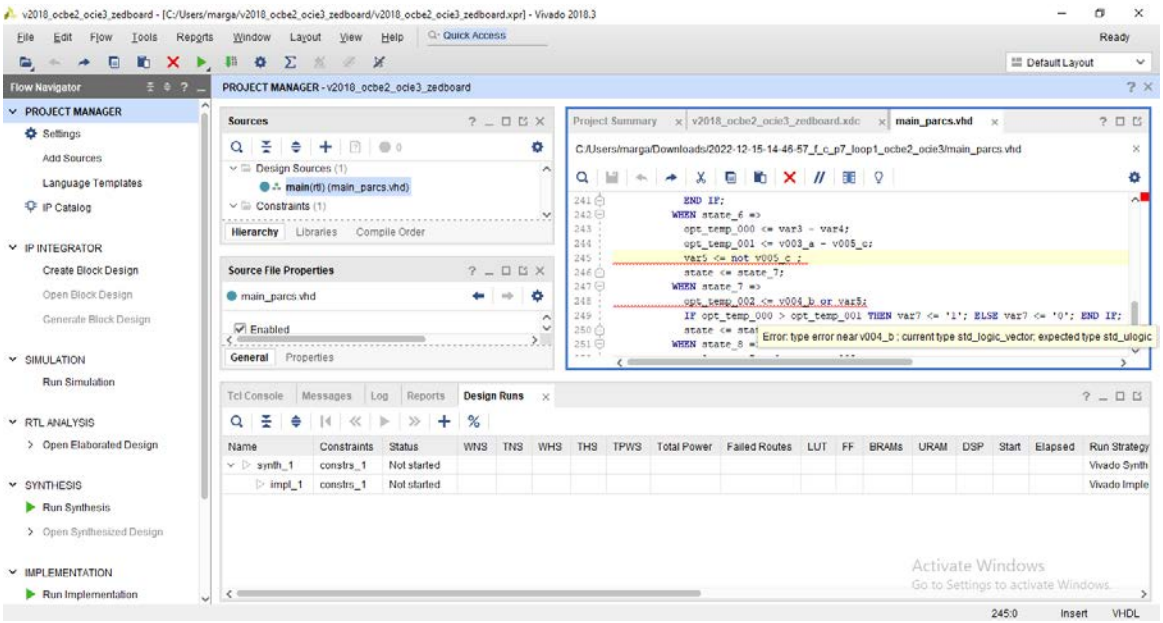


**Image 31.** Ocbe2 ocie3 std_logic to vector assignment error

Πηγή: **(none)**

Therefore a few modifications needed to be made for Vivado's version of the .vhd input (output of "diff <original>.vhd <modified>.vhd):

"

< main : OUT std_logic_vector(31 DOWNTO 0);

< done, busy : OUT std_logic

---

> --main : OUT std_logic_vector(31 DOWNTO 0);

> done, busy, our_main : OUT std_logic

122,123c122,123

< SIGNAL opt_temp_002 : std_logic;

< SIGNAL opt_temp_003 : std_logic;

---

> SIGNAL opt_temp_002 : std_logic_vector(31 DOWNTO 0);

> SIGNAL opt_temp_003 : std_logic_vector(31 DOWNTO 0);

```
140c140
<    SIGNAL var5 : std_logic;
---
>    SIGNAL var5 : std_logic_vector(31 DOWNTO 0);
142,143c142,143
<    SIGNAL var7 : std_logic;
<    SIGNAL var8 : std_logic;
---
>    SIGNAL var7 : std_logic_vector(31 DOWNTO 0);
>    SIGNAL var8 : std_logic_vector(31 DOWNTO 0);
161c161,162
<        main <= (OTHERS => '0');
---
>        --main <= (OTHERS => '0');
>        our_main <= '0';
170,171c171,172
<        opt_temp_002 <= '0';
<        opt_temp_003 <= '0';
---
>        opt_temp_002 <= (OTHERS => '0');
>        opt_temp_003 <= (OTHERS => '0');
180c181
<        var5 <= '0';
---
>        var5 <= (OTHERS => '0');
182,183c183,184
```

```
<       var7 <= '0';
<       var8 <= '0';
---
>       var7 <= (OTHERS => '0');
>       var8 <= (OTHERS => '0');
192c193
<           IF results_read = '1' THEN done_int <= '0'; END IF;
---
>           IF results_read = '1' THEN done_int <= '0'; our_main <= '0'; END IF;
249c250
<           IF opt_temp_000 > opt_temp_001 THEN var7 <= '1'; ELSE var7 <= '0'; END IF;
---
>           IF opt_temp_000 > opt_temp_001 THEN var7 <= (OTHERS => '1'); ELSE var7 <=
(OTHERS => '0'); END IF;
258c259
<           v002_m2(CONV_INTEGER(v007_j)) <= opt_temp_003;
---
>           v002_m2(CONV_INTEGER(v007_j)) <= opt_temp_003(0);
"
```

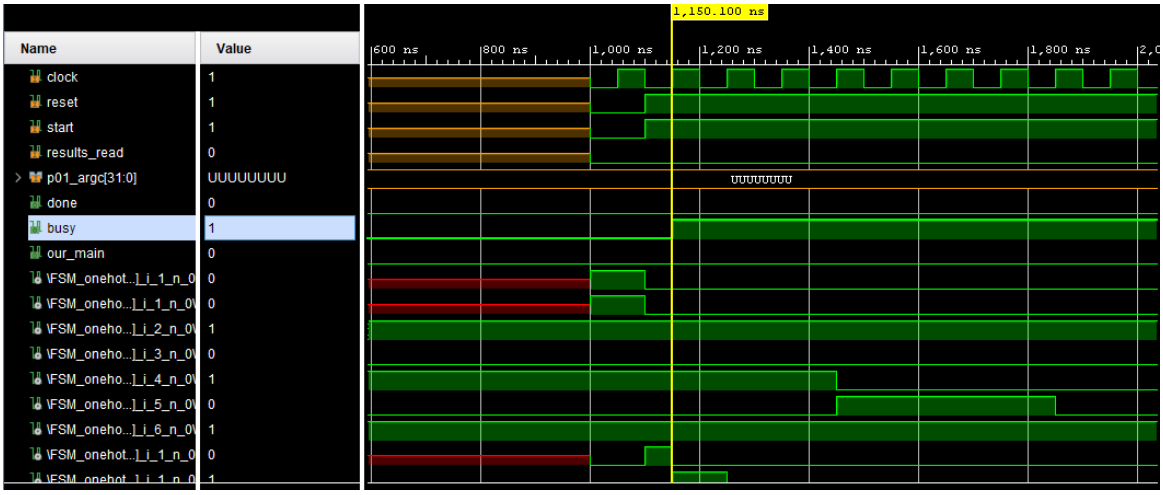


**Image 32.** **First test ocbe2 ocie3 post-imp. func. simulation start**

Πηγή: **(none)**

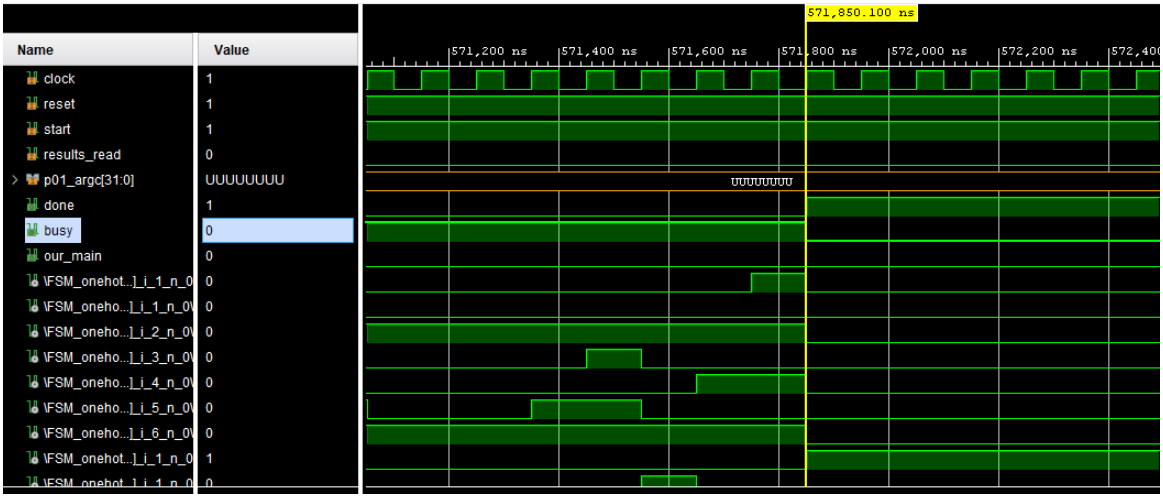


**Image 33.** **First test ocbe2 ocie3 post-imp. func. simulation end**

Πηγή: **(none)**

The backend PARCS optimized result of the output of running "./a.exe -noshort -ocbe=2 -ocie=3 input_files/f_c_p7_loop1_ocbe2_ocie3.c" finished in 570700.0ns with a clock cycle of 100.0ns. This is a x0.7260 speedup from the original (so actually a slowdown) and a x0.9839 speedup from the fully unrolled version (also a slowdown). The tcl console commands used were the same as the ones in the non-optimized section bar the "run" commands which used different ns values and varied in count.

**Table 3. First test ocbe2 ocie3 hardware utilization**

| Total Power | Failed Routes | LUT | FF | BRAMs | URAM | DSP |
|---|---|---|---|---|---|---|
| | | 79 | 82 | 0.00 | 0 | 0 |
| 0.11 | 0 | 79 | 82 | 0.00 | 0 | 0 |

Πηγή: (None)

| | |
|---|---|
| Worst Negative Slack (WNS): | 95.87 ns |
| Total Negative Slack (TNS): | 0 ns |
| Number of Failing Endpoints: | 0 |
| Total Number of Endpoints: | 159 |

**Image 34.** First test ocbe2 ocie3 timing summary

Πηγή: **(none)**

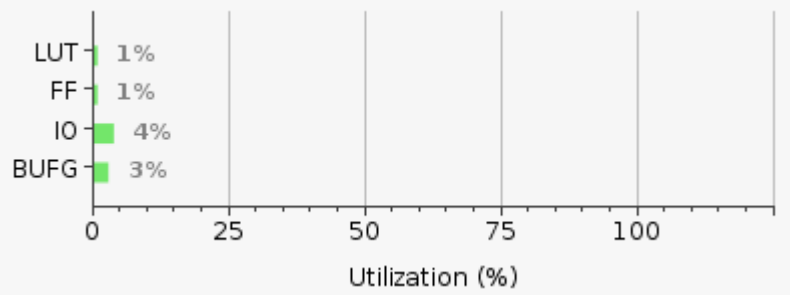


**Image 35.** First test ocbe2 ocie3 utilization summary

Πηγή: **(none)**

### 4.1.3.4 With full unrolling and expression compression (with a boolean depth of 2 and an integer depth of 3)

The same problem with the plain expression compression problem was encountered when combining the two optimizations. The changes that were required were therefore similar to the ones in the previous example: std_logic variables with mixed assignments were converted to std_logic_vector, while some assignments were modified to use the least significant bits (LSBs) for the operations. The following shows a glimpse of the changes (tail of the output of "diff <original>.vhd <modified>.vhd):

```
"
640c640
<          v002_m2(CONV_INTEGER(const2)) <= opt_temp_035;
---
>          v002_m2(CONV_INTEGER(const2)) <= opt_temp_035(0);
654c654
<          var70 <= var69 and opt_temp_038;
---
>          var70(0) <= var69 and opt_temp_038(0);
660c660
<          v002_m2(CONV_INTEGER(const13)) <= opt_temp_039;
---
>          v002_m2(CONV_INTEGER(const13)) <= opt_temp_039(0);
691c691
<          var78 <= var77 and opt_temp_042;
---
>          var78(0) <= var77 and opt_temp_042(0);
697c697
<          v002_m2(CONV_INTEGER(v007_j)) <= opt_temp_043;
---
>          v002_m2(CONV_INTEGER(v007_j)) <= opt_temp_043(0);
"
```

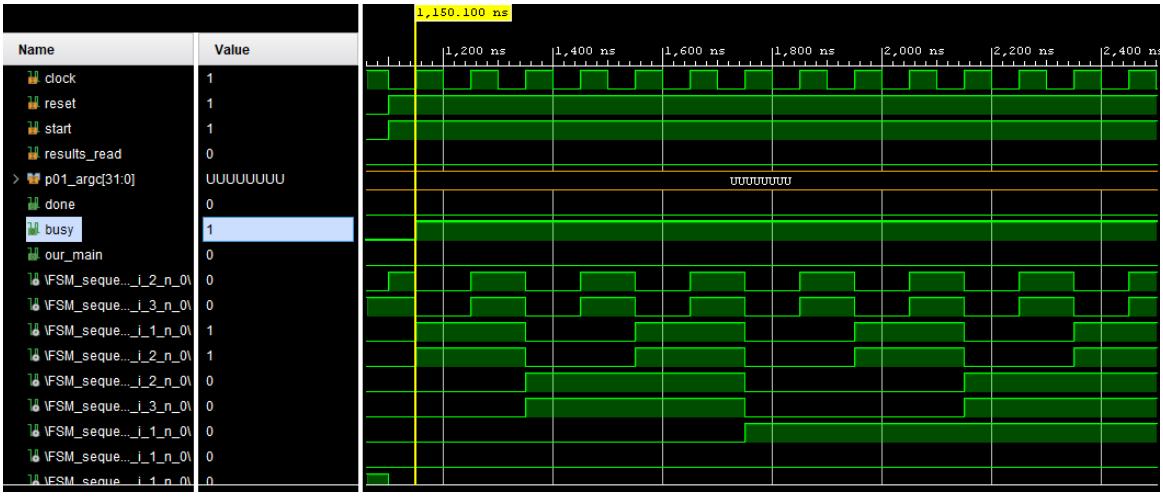


**Image 36.** First test full unr. ocbe2 ocie3 post-imp. func. simulation start

Πηγή: **(none)**

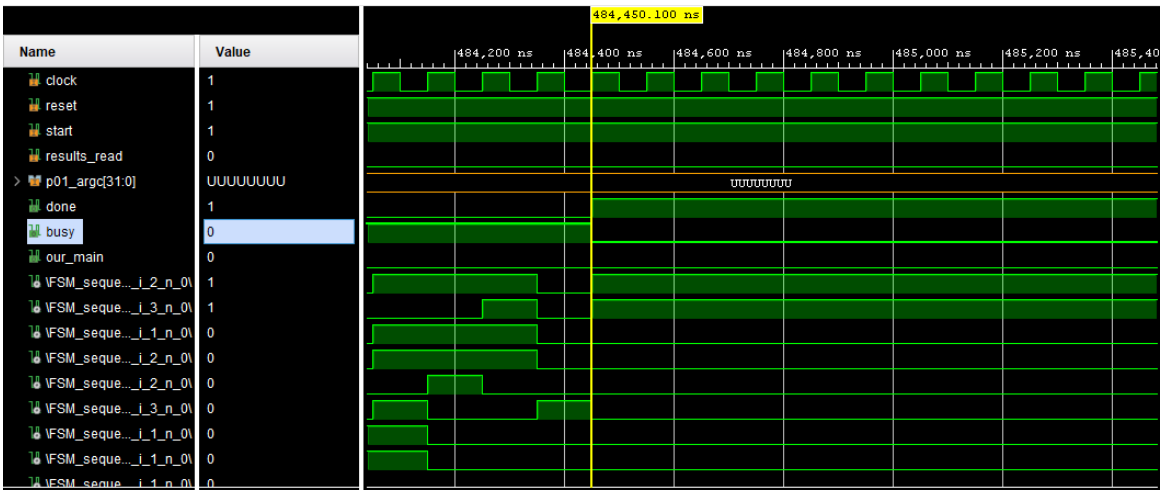


**Image 37.** First test full unr. ocbe2 ocie3 post-imp. func. simulation end

Πηγή: **(none)**

The backend PARCS optimized result of the output of running "./a.exe -noshort -ofuil -ocbe=2 -ocie=3 input_files/f_c_p7_loop1_ofuil_ocbe2_ocie3.c" finished in 483300.0ns with a clock cycle of 100.0ns. This is a x0.8572 speedup from the original (so actually a slowdown), a x1.1618 speedup from the plain fully unrolled version, and a x1.1808 speedup from the plain expression compression version. The tcl console commands used were the same as the ones in the non-optimized section bar the "run" commands which used different ns values and varied in count.

**Table 4. First test ofuil ocbe2 ocie3 hardware utilization**

| Total Power | Failed Routes | LUT | FF | BRAMs | URAM | DSP |
|---|---|---|---|---|---|---|
| | | 142 | 140 | 0.00 | 0 | 0 |
| 0.11 | 0 | 142 | 140 | 0.00 | 0 | 0 |

Πηγή: (None)

| | |
|---|---|
| Worst Negative Slack (WNS): | 94.697 ns |
| Total Negative Slack (TNS): | 0 ns |
| Number of Failing Endpoints: | 0 |
| Total Number of Endpoints: | 275 |

**Image 38.** First test ofuil ocbe2 ocie3 timing summary

Πηγή: **(none)**

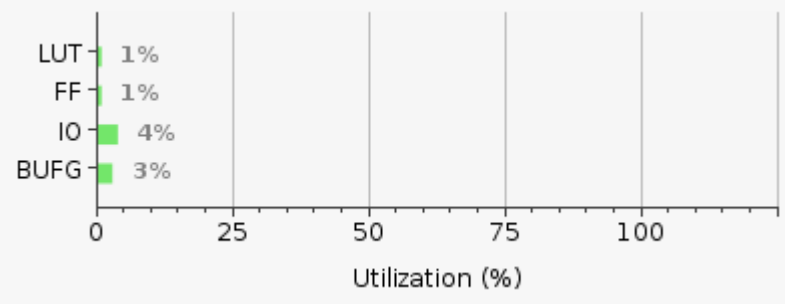


**Image 39.** First test ofuil ocbe2 ocie3 utilization summary

Πηγή: **(none)**

The results show a slowdown from the original unoptimized version for every modification applied. The timings of the example with full unrolling combined with expression compression are similar to the unoptimized version, however, with a difference of about ~70000ns.

**Table 5. First test benchmark results (running times in nanoseconds)**

| | **No optimization** | **Fully unrolled** | **Bool=2, integer=3** | **Fully unrolled, bool=2, integer=3** |
|---|---|---|---|---|
| **Run 1** | 414300.0ns | 561500.0ns | 570700.0ns | 483300.0ns |

Πηγή: (None)

## 4.2 Second test

The second test will focus on loop unrolling and instruction reordering. To test the output, the following .c input was used (while the contents are the same, the filenames indicate the optimizations used to preserve results and prevent overwriting):

"

```
int main(int argc)
{
	int a = 5, b = 10, c = 15, i, j, x, y, z;

	for (i = 4; i < 50; i++ ) {
		for (j = 2; j < 19; j++ ) {
			x = (2 && (!b)) || (c && (a || 0));
			y = 1 == ((-a > (b + c - 5)) != (7 <= c));
			z = ((c && a) == 0) && (((a + c) > (c - 4)) || (b != 10));
		}
	}

	return 0;
}
```

"

### 4.2.1 State count reduction

The unoptimized (by the backend) result of the output of running "./a.exe -noshort input_files/f_c_p7_loop4_2_no_opts.c" generates 45 states.

The backend PARCS optimized result of the output of running "./a.exe -noshort input_files/f_c_p7_loop4_2_no_opts.c" generates 17 states.

This is a 62.2% decrease.

We will now focus on the PARCS optimized results' state increase/decrease in relation to the unoptimized input results shown above (17 states).

The PARCS optimized result of the output of running "./a.exe -noshort -ouil input_files/f_c_p7_loop4_2_ouil.c" generates 64 states: a 276.47% increase from the non optimized version. For reference, the simple non PARCS optimized result has 152 states.

The PARCS optimized result of the output of running "./a.exe -noshort -ofuil input_files/f_c_p7_loop4_2_ofuil.c" generates 128 states: a 652.94% increase from the non optimized version. For reference, the simple non PARCS optimized result has 296 states.

The PARCS optimized result of the output of running "./a.exe -noshort -ofuilr input_files/f_c_p7_loop4_2_ofuilr.c" generates 128 states: a 652.94% increase from the non optimized version. For reference, the simple non PARCS optimized result has 296 states.

The higher the unrolling factor, the more dramatic the increase of states seems to be. Nevertheless, the PARCS optimizer is evidently capable of decreasing the states to less than half the amount of the original .vhd file in every case. Reordering doesn’t seem to cause any deviation in the state count of the output whatsoever.

### 4.2.2 Vivado post-implementation functional simulation timing results

As with the first test, the "main" stc_logic_vector was replaced with a std_logic "our_main" in all input files for Vivado.

#### 4.2.2.1 No optimizations

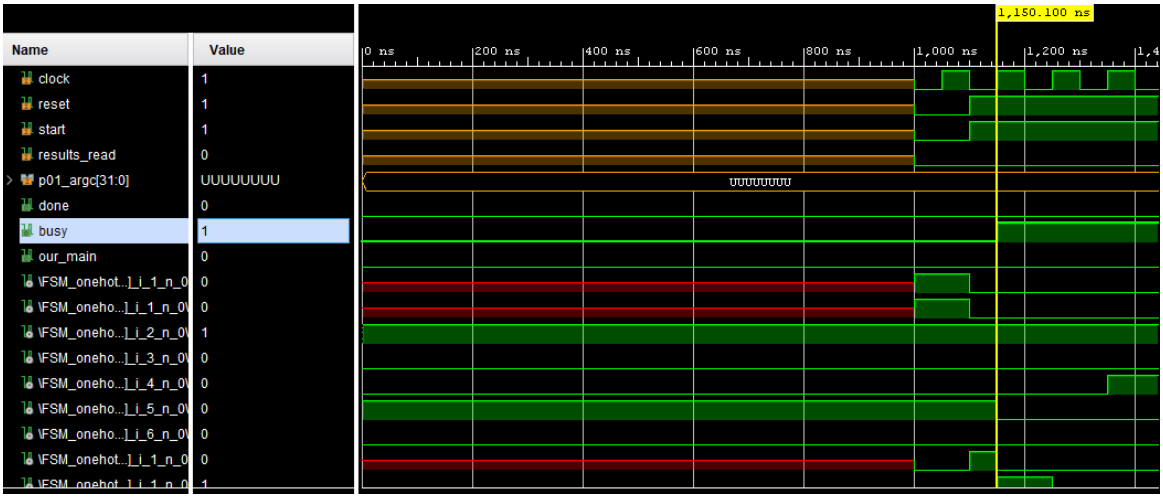


**Image 40.** Second test no opts. post-imp. func. simulation start

Πηγή: **(none)**

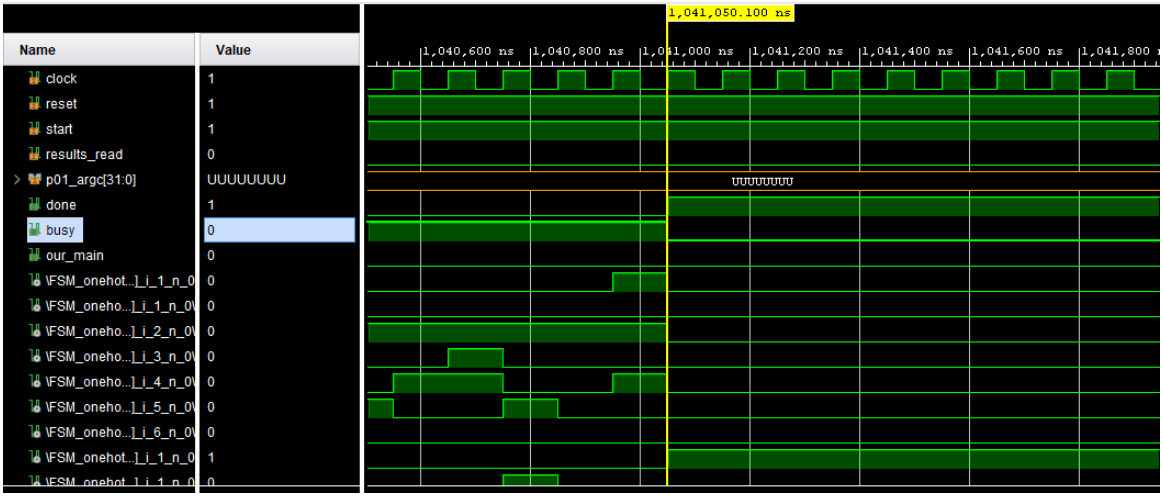


**Image 41.** Second test no opts. post-imp. func. simulation end

Πηγή: **(none)**

The backend PARCS optimized result of the output of running "./a.exe -noshort input_files/f_c_p7_loop4_2_no_opts.c" finished in 1039900.0ns with a clock cycle of 100.0ns. The tcl console commands used were the same as the ones in the first test bar the "run" commands which used different ns values and varied in count. They are repeated below for convenience:

"

add_force clock {0 0ns} {1 50ns} -repeat_every 100ns

add_force reset {0 0ns} {1 100ns}

add_force start {0 0ns} {1 100ns}

add_force results_read {0 0ns}

run 300ns

run 1000000ns

run 50000ns

add_force results_read {1 0ns} {0 100ns}

run 1000ns

"

**Table 6. Second test no opts. hardware utilization**

| Total Power | Failed Routes | LUT | FF | BRAMs | URAM | DSP |
|---|---|---|---|---|---|---|
| | | 80 | 89 | 0.00 | 0 | 0 |
| 0.11 | 0 | 80 | 89 | 0.00 | 0 | 0 |

Πηγή: (None)

| | |
|---|---|
| Worst Negative Slack (WNS): | 95.414 ns |
| Total Negative Slack (TNS): | 0 ns |
| Number of Failing Endpoints: | 0 |
| Total Number of Endpoints: | 172 |

**Image 42.** **Second test no opts. timing summary**

Πηγή: **(none)**

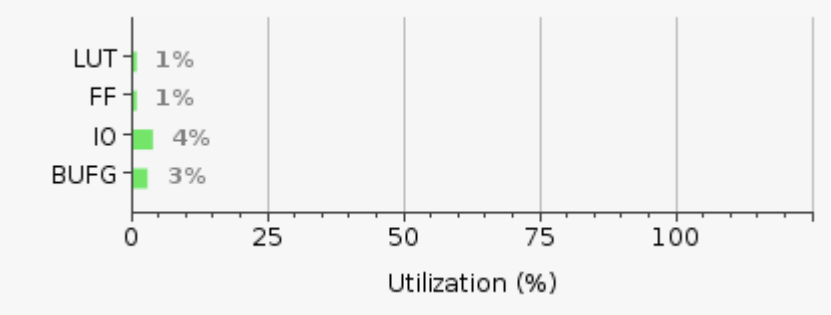


**Image 43.** Second test no opts. utilization summary

Πηγή: **(none)**

### 4.2.2.2 Simple unrolling (4 times)

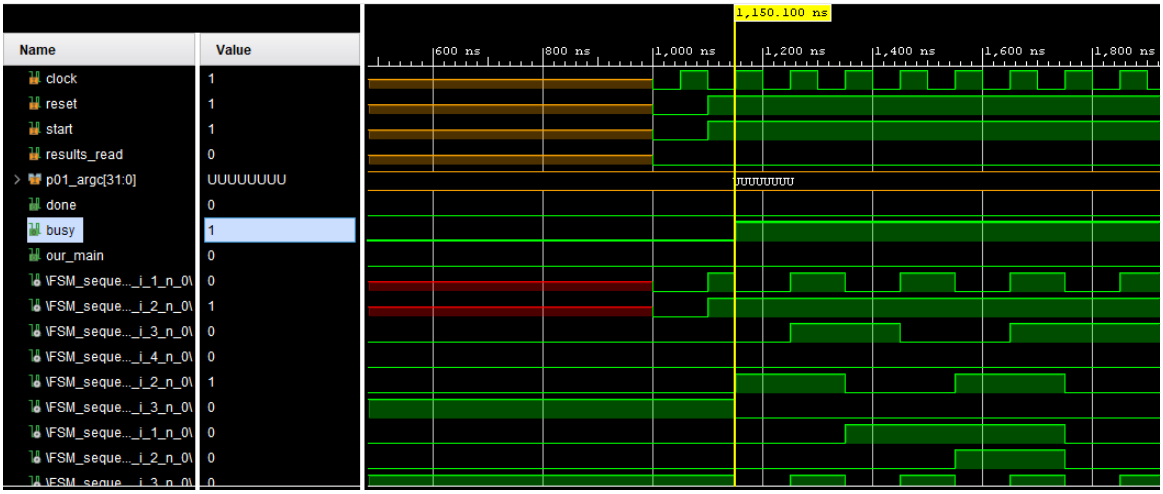


**Image 44.** Second test ouil post-imp. func. simulation start

Πηγή: **(none)**


**Image 45.** Second test no ouil post-imp. func. simulation end

Πηγή: **(none)**

The backend PARCS optimized result of the output of running "./a.exe -noshort -ouil input_files/f_c_p7_loop4_2_ouil.c" finished in 943300.0ns with a clock cycle of 100.0ns. This is a speedup of x1.1024 in relation to the unoptimized version. The tcl console commands used were the same as the ones in the first test bar the "run" commands which used different ns values and varied in count.

**Table 7. Second test ouil hardware utilization**

| Total Power | Failed Routes | LUT | FF | BRAMs | URAM | DSP |
|---|---|---|---|---|---|---|
| | | 168 | 142 | 0.00 | 0 | 0 |
| 0.11 | 0 | 168 | 142 | 0.00 | 0 | 0 |

Πηγή: (None)

| | |
|---|---|
| Worst Negative Slack (WNS): | 94.982 ns |
| Total Negative Slack (TNS): | 0 ns |
| Number of Failing Endpoints: | 0 |
| Total Number of Endpoints: | 278 |

**Image 46.** Second test ouil timing summary

Πηγή: **(none)**

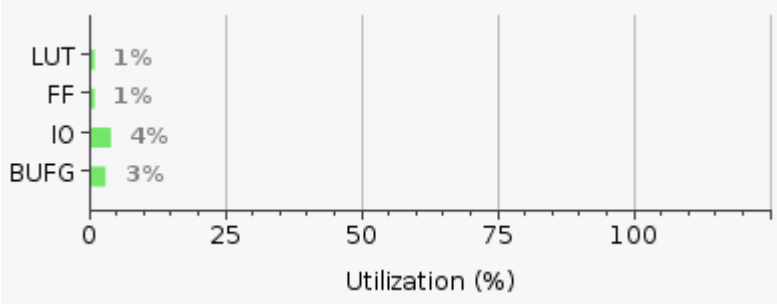


**Image 47.** **Second test ouil utilization summary**

Πηγή: **(none)**

### 4.2.2.3 Full unrolling (maximum of 10, no reordering)

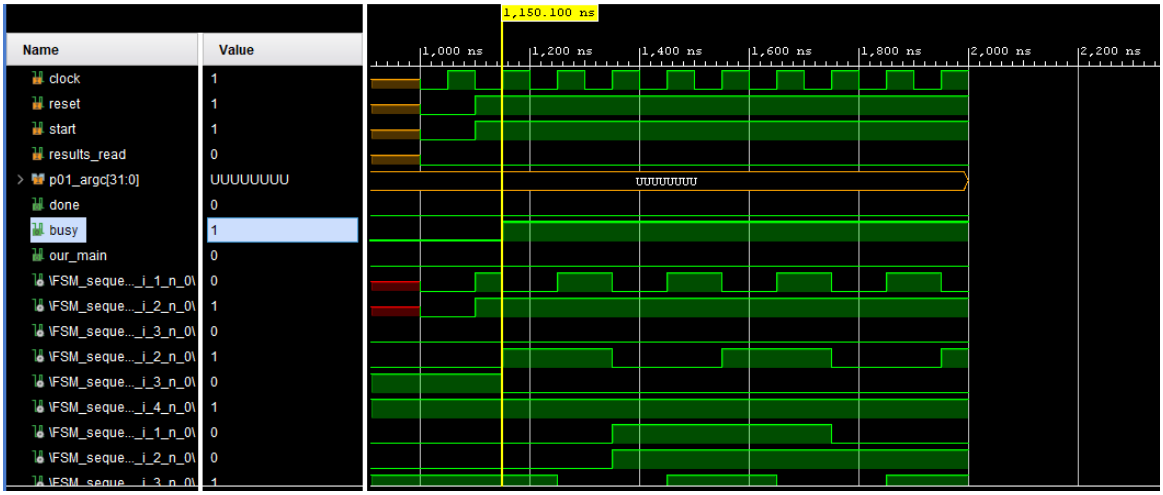


**Image 48.** **Second test ofuil post-imp. func. simulation start**

Πηγή: **(none)**

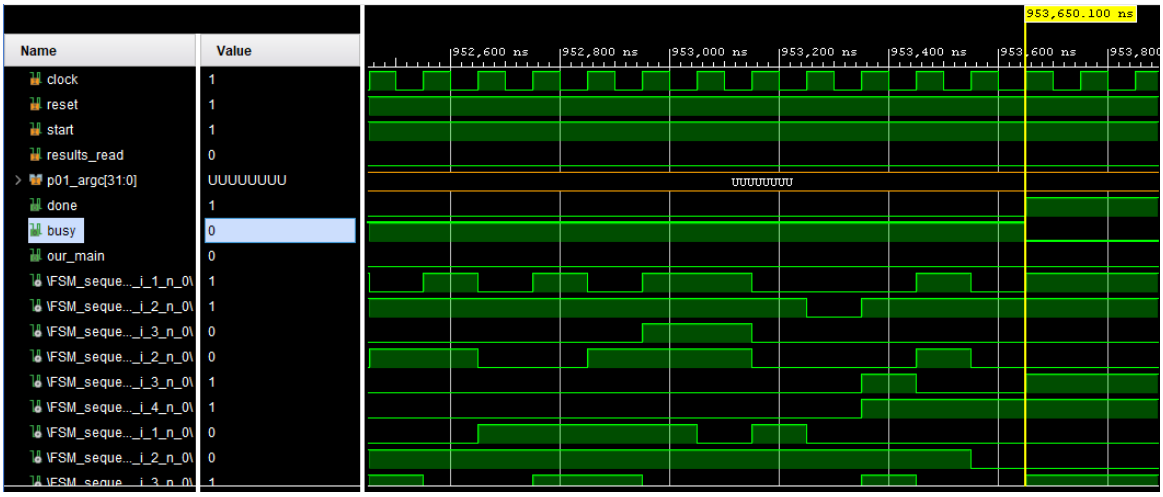


**Image 49.** **Second test ofuil post-imp. func. simulation end**

Πηγή: **(none)**

The backend PARCS optimized result of the output of running "./a.exe -noshort -ofuil input_files/f_c_p7_loop4_2_ofuil.c" finished in 952500.0ns with a clock cycle of 100.0ns. This is a speedup of x1.0918 in relation to the unoptimized version and a x0.9903 speedup (so actually a slowdown) in relation to the simple unrolled version. The tcl console commands used were the same as the ones in the first test bar the "run" commands which used different ns values and varied in count.

**Table 8. Second test ofuil hardware utilization**

| Total Power | Failed Routes | LUT | FF | BRAMs | URAM | DSP |
|---|---|---|---|---|---|---|
| | | 156 | 142 | 0.00 | 0 | 0 |
| 0.11 | 0 | 156 | 142 | 0.00 | 0 | 0 |

Πηγή: (None)

| | |
|---|---|
| Worst Negative Slack (WNS): | 94.422 ns |
| Total Negative Slack (TNS): | 0 ns |
| Number of Failing Endpoints: | 0 |
| Total Number of Endpoints: | 279 |

**Image 50.** **Second test ofuil timing summary**

Πηγή: **(none)**

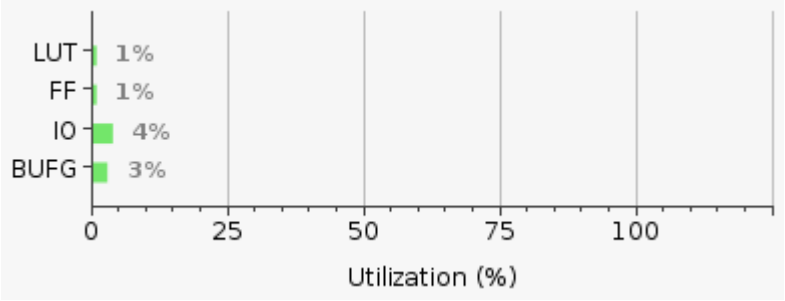


**Image 51.** Second test ofuil utilization summary

Πηγή: **(none)**

### 4.2.2.4 Full unrolling (maximum of 10) with instruction reordering

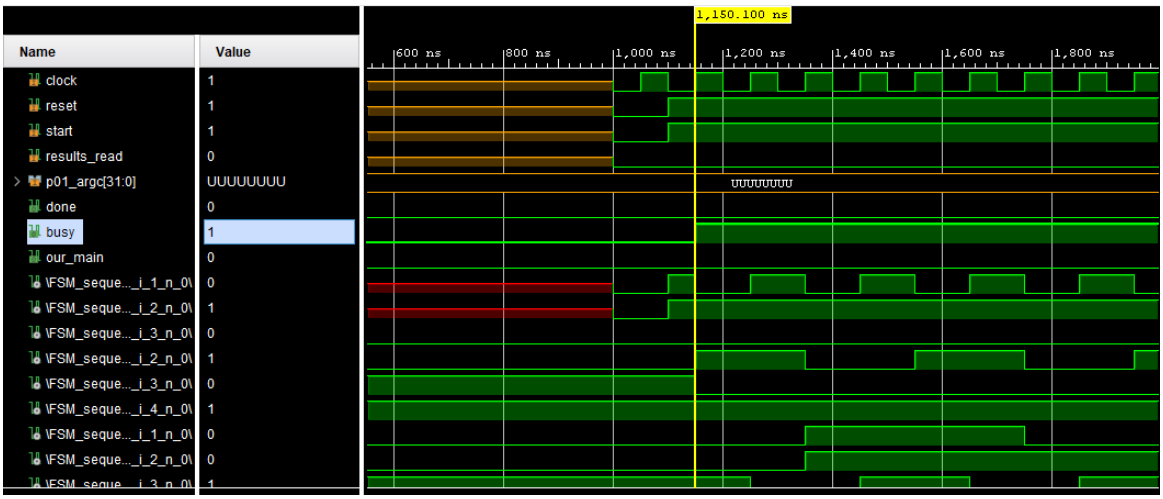


**Image 52.** Second test ofuilr post-imp. func. simulation start

Πηγή: **(none)**

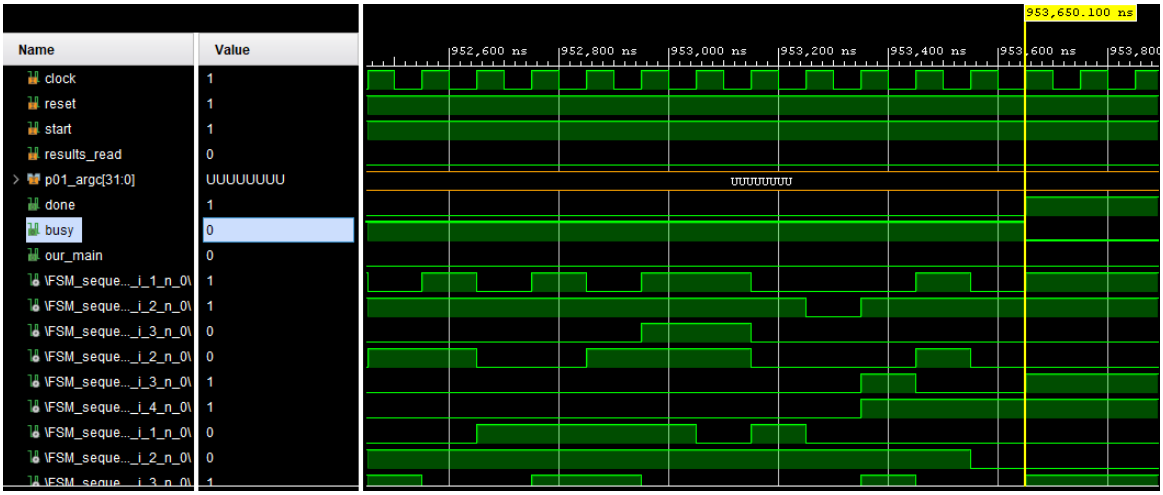


**Image 53.** Second test ofuilr. post-imp. func. simulation end

Πηγή: **(none)**

The backend PARCS optimized result of the output of running "./a.exe -noshort -ofuilr input_files/f_c_p7_loop4_2_ofuilr.c" finished in 952500.0ns with a clock cycle of 100.0ns. This seems to present no change whatsoever in relation to the version of full unrolling without reordering. So this is still a speedup of x1.0918 in relation to the unoptimized version and a x0.9903 speedup (so actually a slowdown) in relation to the simple unrolled version. The tcl console commands used were the same as the ones in the first test bar the "run" commands which used different ns values and varied in count.

**Table 9. Second test ofuilr hardware utilization**

| Total Power | Failed Routes | LUT | FF | BRAMs | URAM | DSP |
|---|---|---|---|---|---|---|
| | | 156 | 142 | 0.00 | 0 | 0 |
| 0.11 | 0 | 156 | 142 | 0.00 | 0 | 0 |

Πηγή: (None)

| | |
|---|---|
| Worst Negative Slack (WNS): | 94.422 ns |
| Total Negative Slack (TNS): | 0 ns |
| Number of Failing Endpoints: | 0 |
| Total Number of Endpoints: | 279 |

**Image 54.** Second test ofuilr timing summary

Πηγή: **(none)**

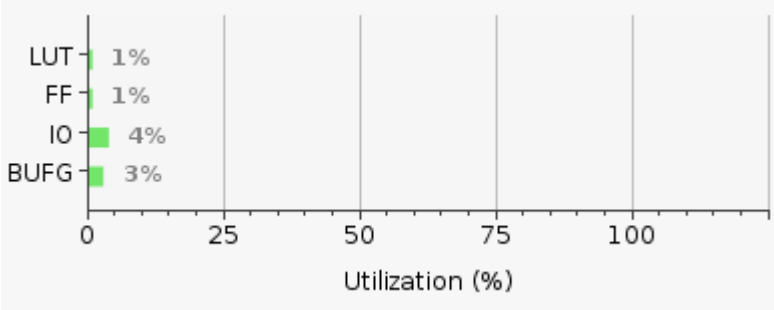


**Image 55. Second test ofuilr utilization summary**

Πηγή: **(none)**

This second test reflected no improvements in latency by applying full unrolling over simple unrolling, whether the instructions are reordered or not.

**Table 10. Second test benchmark results (running times in nanoseconds)**

| | **No optimization** | **Unrolled 4 times** | **Fully unrolled (10 times)** | **Fully unrolled (10 times) and reordered** |
|---|---|---|---|---|
| **Run 1** | 1039900.0ns | 943300.0ns | 952500.0ns | 952500.0ns |

Πηγή: (None)

## 4.3 Third test

The third test is meant to resemble a section of Twofish's decryption algorithm.

"

/* Code taken and adjusted from:

* https://github.com/bheesham/applied-cryptography-accompanying-source-code/blob/master/TWOFISH

* It is meant to resemble a step in decryption.

*/

int pow(int base, int exp)

```
{
	int result = 1;

	for( ;exp > 0; exp--) result *= base;

	return result;
}

int rotr(int x, int n)
{
	//return ((x >> (n)) | (x << (32 - n)));
	return ((x / pow(2, n)) | (x * pow(2, (32 - n))));
}

int rotl(int x, int n)
{
	//return ((x << (n)) | (x >> (32 - n)));
	return ((x * pow(2, n)) | (x / pow(2, (32 - n))));
}

int main(int argc)
{
	int t0 = 1, t1 = 2, i, j, blk[4], l_key[45];

	for(i = 0; i < 45; i++) l_key[i] = i;

	for (j = 0; j < 8; j++ ) {
		blk[0] = 5;
		blk[1] = 9;
		blk[2] = 10;
```

```
            blk[3] = 7;
            for (i = 0; i < 8; i++ ) {
        if (i % 2) {
          if ((i % 4) >= 2)
            blk[i % 4] = rotr(blk[i % 4] ^ (t0 + 2 * t1 + l_key[4 * (i) + 11]), 1);
          else
            blk[i % 4] = rotr(blk[i % 4] ^ (t0 + 2 * t1 + l_key[4 * (i) + 9]), 1);
        } else {
          if ((i % 4) >= 2)
            blk[i % 4] = rotl(blk[i % 4], 1) ^ (t0 + t1 + l_key[4 * (i) + 10]);
          else
            blk[i % 4] = rotl(blk[i % 4], 1) ^ (t0 + t1 + l_key[4 * (i) + 8]);
        }

        t0 = blk[0] ^ blk[3];
        t1 = blk[1] ^ blk[2];
            }
    }

    return 0;
}
"
```

It consists of four functions: "pow" which calculates the power given a base and an exponent, "rotl" which rotates an integer to the left (this calls "pow" twice), "rotr" which rotates an integer to the right (this also calls "pow" twice), and the main function which calls "rotl" and "rotr" along with some other calculations in a couple of conditions in a loop. Due to these conditions, however, csense could not apply loop unrolling to this particular loop. It was applied to an array initialization loop earlier in the "main" function. Expression compression could be and was applied to the calculations in "main".

### 4.3.1 State count reduction

The unoptimized (by the backend) result of the output of running "./a.exe -noshort input_files/tf_decr_part_no_opts.c" generates 96 states for main, 12 states for pow 8 states for rotl and 8 states for rotr.

The backend PARCS optimized result of the output of running "./a.exe -noshort input_files/tf_decr_part_no_opts.c" generates 48 states for main, 6 states for pow, 5 states for rotl and 5 states for rotr.

This is a total decrease of 48.38%.

We will now focus on the PARCS optimized results' state increase/decrease in relation to the unoptimized input results shown above (48 states for main, 6 states for pow, 5 states for rotl and 5 states for rotr; a total state count of 64).

The PARCS optimized result of the output of running "./a.exe -noshort -ofuil -ocbe=3 -ocie=3 input_files/tf_decr_part_ofuil_ocbe3_ocie3.c" generates a total of 75 states (59 states for main, 6 states for pow, 5 states for rotl and 5 states for rotr): a 17.19% increase from the non optimized version. For reference, the simple non PARCS optimized result has 123 states for main, 12 states for pow, 8 states for rotl and 8 states for rotr; a total state count of 151.

### 4.3.2 Vivado post-implementation functional simulation timing results

For the simulation a top level wrapper was appended to the project in order to avoid mapping "main"'s "rotl" and "rotr" I/O variables to pins on the Zedboard. For all simulations (including behavioral) a testbench was required to be written declaring two separate instances of the "pow" component; one for "rotl" and one for "rotr". As with the first and second tests, the "main" stc_logic_vector was replaced with a std_logic "our_main" in all input files for Vivado.

#### 4.3.2.1 No optimizations

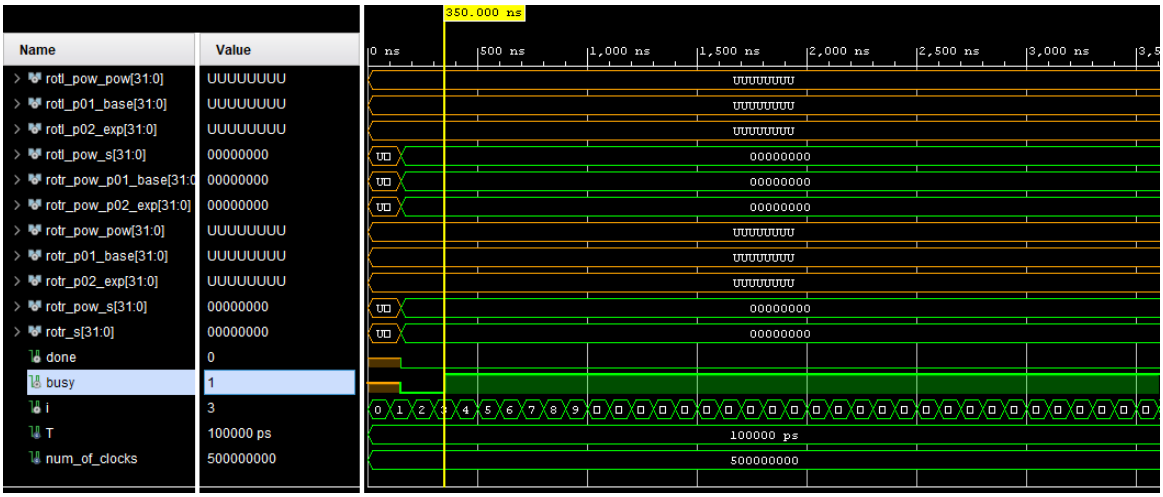


**Image 56.** Third test no opts. post-imp. func. simulation start

Πηγή: **(none)**

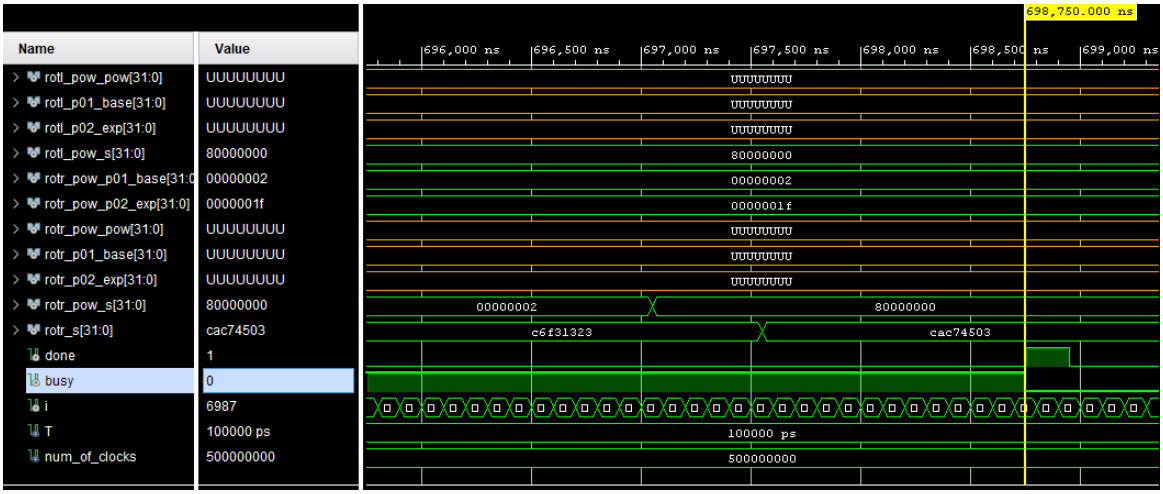


**Image 57.** Third test no opts. post-imp. func. simulation end

Πηγή: **(none)**

The backend PARCS optimized result of the output of running "./a.exe -noshort input_files/tf_decr_part_no_opts.c" finished in 698400.0ns with a clock cycle of 100.0ns. Since we used a testbench, only a "run 700000ns" tcl command was required to run the simulation.

**Table 11. Third test no opts. hardware utilization**

| Total Power | Failed Routes | LUT | FF | BRAMs | URAM | DSP |
|---|---|---|---|---|---|---|
| | | 120 | 87 | 0.00 | 0 | 0 |
| 0.11 | 0 | 120 | 87 | 0.00 | 0 | 0 |

Πηγή: (None)

| | |
|---|---|
| Worst Negative Slack (WNS): | 94.651 ns |
| Total Negative Slack (TNS): | 0 ns |
| Number of Failing Endpoints: | 0 |
| Total Number of Endpoints: | 158 |

**Image 58. Third test no opts. timing summary**

Πηγή: **(none)**

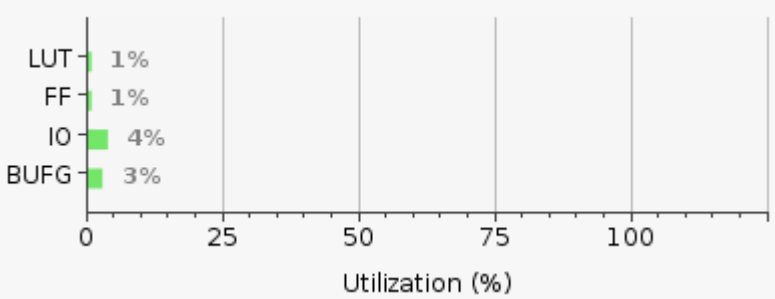


**Image 59. Third test no opts. utilization summary**

Πηγή: **(none)**

#### 4.3.2.2 With full unrolling and expression compression (with a boolean depth of 3 and an integer depth of 3)


**Image 60.** **Third test ofuil ocbe3 ocie3 post-imp. func. simulation start**

Πηγή: **(none)**

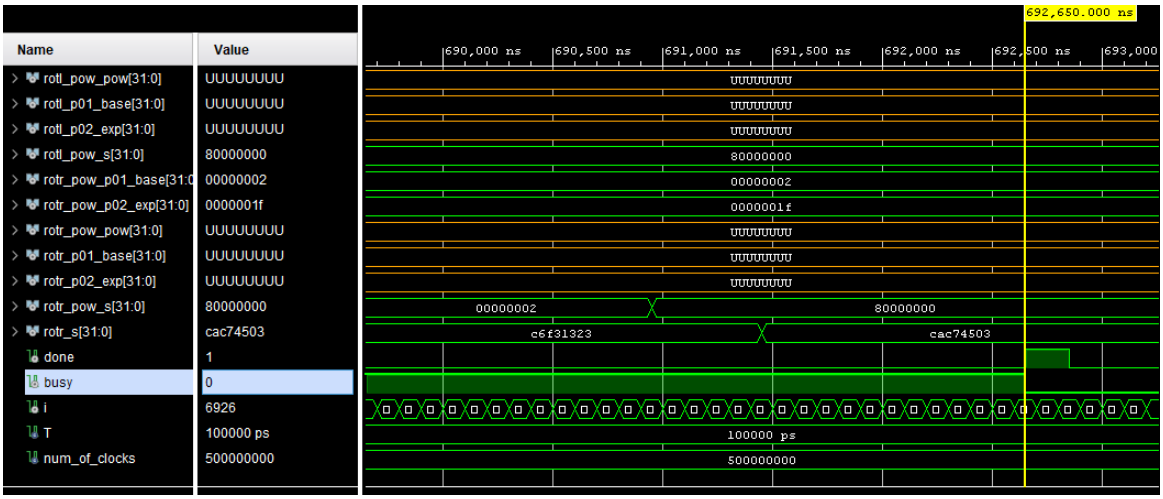


**Image 61.** **Third test ofuil ocbe3 ocie3 post-imp. func. simulation end**

Πηγή: **(none)**

The backend PARCS optimized result of the output of running "./a.exe -noshort -ofuil -ocbe=3 -ocie=3 input_files/tf_decr_part_ofuil_ocbe3_ocie3.c" finished in 692300.0ns with a clock cycle of 100.0ns. This is a x1.0088 speedup in relation to the unoptimized version. Since we used a testbench, only a "run 700000ns" tcl command was required to run the simulation.

**Table 12. Third test ofuil ocbe3 ocie3 hardware utilization**

| Total Power | Failed Routes | LUT | FF | BRAMs | URAM | DSP |
|---|---|---|---|---|---|---|
| 0.11 | 0 | 269 | 178 | 0.00 | 0 | 0 |

Πηγή: (None)

| | |
|---|---|
| Worst Negative Slack (WNS): | 93.421 ns |
| Total Negative Slack (TNS): | 0 ns |
| Number of Failing Endpoints: | 0 |
| Total Number of Endpoints: | 345 |

**Image 62. Third test ofuil ocbe3 ocie3 timing summary**

Πηγή: **(none)**

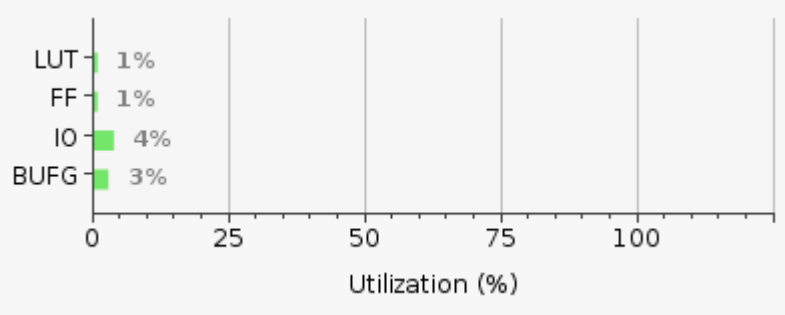


**Image 63. Third test ofuil ocbe3 ocie3 utilization summary**

Πηγή: **(none)**

**Table 13. Third test benchmark results (running times in nanoseconds)**

| | **No optimization** | **Fully unrolled (10 times) Bool=3, integer=3** |
|---|---|---|
| **Run 1** | 698400.0ns | 692300.0ns |

Πηγή: (None)

As can be seen, we got a very slight increase in performance in the third test with our optimizations.

## 4.4 Fourth test

The fourth test is an implementation of the sha1 hashing algorithm. Its implementation in the form used for this project involved Y. Hara, H. Tomiyama, S. Honda, H. Takada, K. Ishii, Uwe Hollerbach, Peter C. Gutmann and Bruce Schneier.

```
"
/*
+--------------------------------------------------------------------+
| CHStone : a suite of benchmark programs for C-based High-Level Synthesis |
|
=====================================================================
|
|                                                    |
| * Collected and Modified : Y. Hara, H. Tomiyama, S. Honda,            |
|                      H. Takada and K. Ishii                  |
|                      Nagoya University, Japan                |
|                                                    |
| * Remark :                                                 |
|    1. This source code is modified to unify the formats of the benchmark |
|       programs in CHStone.                                 |
|    2. Test vectors are added for CHStone.                        |
|    3. If "main_result" is 0 at the end of the program, the program is    |
|       correctly executed.                                  |
|    4. Please follow the copyright of each benchmark program.           |
+--------------------------------------------------------------------+
*/
/* NIST Secure Hash Algorithm */
/* heavily modified by Uwe Hollerbach uh@alumni.caltech edu */
```

```
/* from Peter C. Gutmann's implementation as found in */
/* Applied Cryptography by Bruce Schneier */

/* NIST's proposed modification to SHA of 7/11/94 may be */
/* activated by defining USE_MODIFIED_SHA */

/* Tools */

void
local_memset (unsigned int s[16], int c, int n, int e)
{
  unsigned int uc;
  int i;
  int m;

  m = n / 4;
  uc = (unsigned int) c;
  i = 0;
  while (e-- > 0)
    {
      i++;
    }
  while (m-- > 0)
    {
      s[i++] = uc;
    }
}

void
local_memcpy (unsigned int s1[16], const unsigned char s2[8192], int ps2, int n)
```

```
{
  int i1, i2;
  unsigned int tmp;
  int m;
  m = n / 4;
  i1 = 0;
  i2 = ps2;

  while (m-- > 0)
    {
      s1[i1] =
          0xFF && s2[i2++] || (0xFF && (s2[i2++] * 256)) ||
                (0xFF & (s2[i2++] * (256*256))) ||
                          (0xFF & (s2[i2++] * (256*256*256)));
      i1++;
    }
}
"
```

For the tests, only the “local_memcpy” function was compiled.

### 4.4.1 State count reduction

The unoptimized (by the backend) result of the output of running "./a.exe -noshort input_files/sha1_no_opts.c" generates 43 states for local_memcpy.

The backend PARCS optimized result of the output of running "./a.exe -noshort input_files/sha1_no_opts.c" generates 23 states for local_memcpy.

This is a 46.51% decrease.

We will now focus on the PARCS optimized results' state increase/decrease in relation to the unoptimized input results shown above (23 states).

The PARCS optimized result of the output of running "./a.exe -noshort -ofuilr -ocbe=3 -ocie=3 input_files/sha1_ofuilr_ocbe3_ocie3.c" generates 25 states: an 8.7% increase from the non optimized version. For reference, the simple non PARCS optimized result has 55 states.

This time a very slight increase was observed for the state count where only expression compression was applied.

### 4.4.2 Vivado post-implementation functional simulation timing results

#### 4.4.2.1 No optimizations

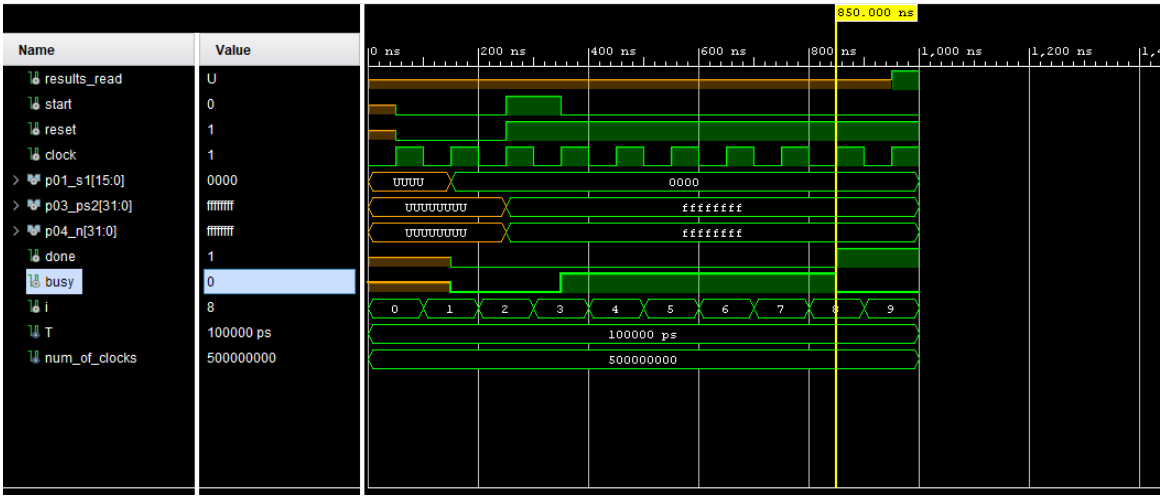


**Image 64.** **Fourth test no opt. post-imp. func. simulation end**

Πηγή: **(none)**

The backend PARCS optimized result of the output of running "./a.exe -noshort input_files/sha1_no_opts.c" finished in 500.0ns with a clock cycle of 100.0ns. No other commands were required for the simulation to run.

**Table 14. Fourth test no opts. hardware utilization**

| Total Power | Failed Routes | LUT | FF | BRAMs | URAM | DSP |
|---|---|---|---|---|---|---|
| | | 79 | 72 | 0.00 | 0 | 0 |
| 0.11 | 0 | 79 | 72 | 0.00 | 0 | 0 |

Πηγή: (None)

| | |
|---|---|
| Worst Negative Slack (WNS): | 95.195 ns |
| Total Negative Slack (TNS): | 0 ns |
| Number of Failing Endpoints: | 0 |
| Total Number of Endpoints: | 142 |

**Image 65. Fourth test no opts. timing summary**

Πηγή: **(none)**

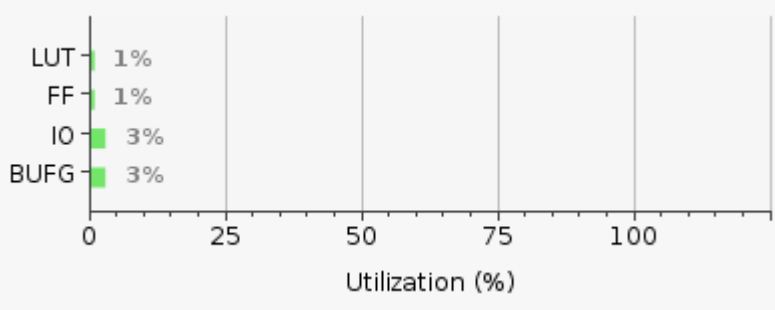


**Image 66. Fourth test no opts. utilization summary**

Πηγή: **(none)**

### 4.4.2.2 Full loop unrolling with instruction reordering and a boolean and integer depth of 3

(No unrolling was available for this test, so only the expression compression was actually applied.)

There were errors while compiling related to std_logic_vectors being assigned to std_logic, so the LSB modification was required in the main source:

"

[...]

```
            WHEN state_23 =>
          var7 <= var6 or opt_temp_011(0);
          state <= state_24;
         WHEN state_24 =>
          opt_temp_016 <= var7 or opt_temp_015(0);
          state <= state_25;
```

[...]

"

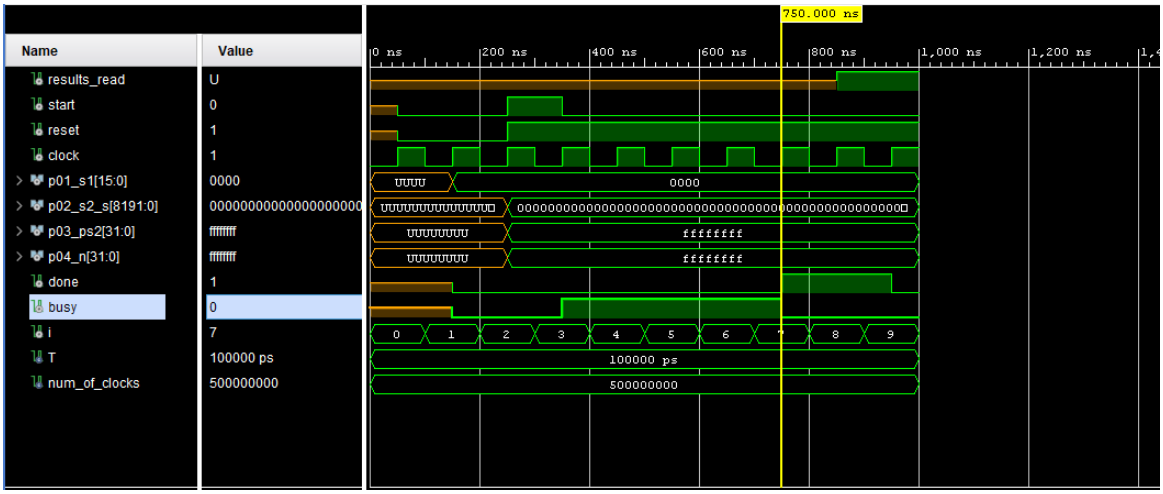


**Image 67.** **Fourth test ofuilr ocbe3 ocie3 post-imp. func. simulation end**

Πηγή: **(none)**

The backend PARCS optimized result of the output of running "./a.exe -noshort -ofuilr -ocbe=3 -ocie=3 input_files/sha1_ofuilr_ocbe3_ocie3.c" finished in 400.0ns with a clock cycle of 100.0ns. This is a x1.25 speedup in relation to the unoptimized version. No other commands were required for the simulation to run.

**Table 15. Fourth test ofuilr ocbe3 ocie3 hardware utilization**

| Total Power | Failed Routes | LUT | FF | BRAMs | URAM | DSP |
|---|---|---|---|---|---|---|
| | | 81 | 72 | 0.00 | 0 | 0 |
| 0.11 | 0 | 81 | 72 | 0.00 | 0 | 0 |

Πηγή: (None)

| | |
|---|---|
| Worst Negative Slack (WNS): | 95.885 ns |
| Total Negative Slack (TNS): | 0 ns |
| Number of Failing Endpoints: | 0 |
| Total Number of Endpoints: | 142 |

**Image 68.** **Fourth test ofuilr ocbe3 ocie3 timing summary**

Πηγή: **(none)**

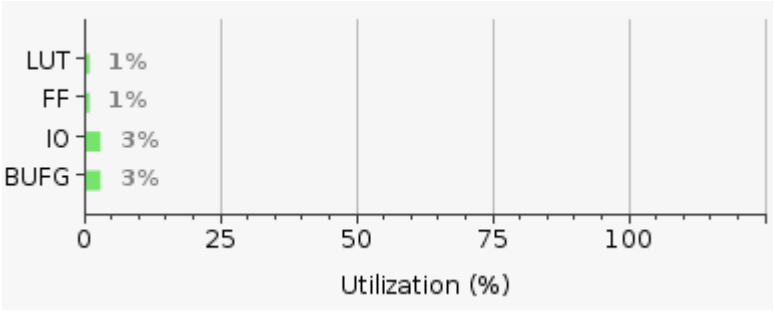


**Image 69.** **Fourth test ofuilr ocbe3 ocie3 utilization summary**

Πηγή: **(none)**

**Table 16. Fourth test benchmark results (running times in nanoseconds)**

| | **No optimization** | **Ex. Comp.: Bool=3, integer=3** |
|---|---|---|
| **Run 1** | 500.0ns | 400.0ns |

Πηγή: (None)

The fourth test did show a slight increase in performance.

# 5. Related Work

This chapter briefly summarizes some work related (admittedly quite loosely) to the subjects of this paper.

## 5.1 Loop unrolling

[38] use a neural network to find the optimal unrolling factor for TIRAMISU programs.

[39] modify LLVM's DAG for loop optimizations.

[40] describes loop optimizations (including loop unrolling) of a code generator written in python.

[41] apply loop unrolling on an ASIP (Application Specific Instruction-set Processor) to improve ILP and performance.

[42] benchmark several versions of matrix multiplication methods (including some that are loop-unrolled).

[43] traversed a linked list in an unrolled loop and discovered a slight improvement in energy usage and power consumption.

[44] measures performance of unrolled and pipelined inner loops on an FPGA (field programmable gate array) to simulate specialized hardware for (particular) programs.

[45] transform a shadow AST (as the original is immutable) of clang to implement OpenMP's loop transformations (including loop unrolling).

[46] show a less favorable result of applying loop unrolling to AES (advanced encryption standard) encryption algorithms, compared to other optimizations.

[47] present a reduction technique using loop unrolling (among others) to increase parallelisim and performance on GPUs.

## 5.2 Source to source compilers/transpilers

[48] present a python to NumPy/CuPy source to source optimizing (parallelizing) compiler.

[49] present a C++ to FHE (fully homomorphic encryption) C++ transpiler.

[50] present a new code generation framework for optimizing DSL (domain specific language) code.

[51] present an AMR (Abstract Meaning Representation: rooted, labeled, directed, acyclic graphs, comprising whole sentences) to SPARQL (a semantic query language for databases) transpilation method for KBQA (Knowledge Base Question Answering).

## 5.3 C to Ada compilers

[52], [53] is a public domain/MIT license C to Ada compiler that supports the C '#' directive (although explicit parameters might need to be set when calling the executable to aid in correct translation).

[54] describes another C to Ada translator. The '#' directive is also supported.

(None of the two projects mentioned in this section indicate a support of optimizations.)

## 5.4 High Level Synthesis (HLS)

[55] present a HLS tool (called “CaT”) with source-to-source (high level P4-16 to lower level P4-14) compilation to generate P4 output optimized for ALUs. (P4 is a language for packet switches.)

[56] present an algorithm (called “Iris”) to automatically create an efficient data layout to maximize the use of available bandwidth (intended for High Bandwidth Memory (HBM) architectures). This could aid or improve on HLS output.

[57] present a HLS library for explainable artificial intelligence (XAI, specifically convolutional neural networks (CNNs)) and show results on FPGAs.

[58] present a hardware/software (including a HLS) implementation of BIKE (“Bit Flipping Key Encapsulation”. From the website at bikesuite.org: “BIKE is a code-based key encapsulation mechanism based on QC-MDPC (Quasi-Cyclic Moderate Density Parity-Check) codes submitted to the NIST Post-Quantum Cryptography Standardization Process”) for embedded platforms.

[59] present a Graph Neural Network (GNN) design for particle detection (called “JEDI-net”) with reduced latency and improved efficiency compared to previous approaches oriented for HLS.

[60] present a programming abstraction that leverages HLS, Dynamic Partial Reconfiguration and synchronization mechanisms to use an FPGA as a multi-tasking server with preempting scheduling and priority queues.

[61] present a scalable and automatic stencil acceleration framework on modern HBM-based FPGAs (called "SASA") which employs a multi-PE (processing element) approach to exploit temporal and spatial parallelisms (generates the optimized FPGA design with the best parallelism configuration in TAPA high-level synthesis C++) for better scalability.

[62] present "a novel design and optimization methodology for the compilation and mapping of fixed function neural networks to digital signal processors (DSPs) on the FPGAs employing high-level synthesis flow."

[63] present an automated design space exploration tool for applying HLS optimization directives, (called "Chimera"), which significantly reduces the human effort and expertise needed for creating high-performance HLS designs.

## 5.5 Expression simplification/compression

[64] present an MBA (mixed Boolean and arithmetic) simplifier, called “SSLEM,” based on a new concept of “semi-linear” MBA expression transformation which could be used for obfuscated malware detection.

# Conclusion

This paper accompanies the modifications to implement full loop unrolling, full loop unrolling with instruction reordering/interleaving (for cases where the index is not used in the body), and expression compression for mixed Boolean and integer expressions in the csense optimizing source to source compiler. The modifications aim to improve ILP and data locality, as well as resource use efficiency.

Benchmarks run on the post implementation functional simulations in Vivado with the Zedboard as the target hardware and a clock cycle of 100ns did not show a deviation greater than 0.25 in speedup or slowdown. For two tests which combined full loop unrolling with expression compression, there was a very faint increase in performance, but the decrease seen in one of the other two tests dwarfed any semblance of an improvement due to full loop unrolling specifically. A slight increase in performance was seen in the simple unrolled version compared to the completely plain version in one test, with a slightly smaller improvement seen in the full loop unrolled versions. This simply reflects on the well known fact that the unrolling factor must be chosen carefully for each algorithm and target hardware combination (a dynamic unrolling factor is not available in csense as of the time of this writing). Nevertheless, benchmarks in the related work section do show improvements by applying the same optimization, and it is expected that systems using a VLIW architecture or ASICs based on the output code would benefit the most out of it.

The state count in the VHDL output by the backend PARCS optimizer using our expression compressed Ada input, was reduced by more than 38.88% in one case while an increase of 8.7% was observed in another. In all cases, the loop unrolled versions increased the state count sharply. The minimum state count increase observed involving loop unrolling was 17.19% (which was combined with expression compression).

Γράψτε Προτάσεις μελλοντικής επέκτασης της εργασίας σας

# Βιβλιογραφία (References)

*ΠΡΟΣΟΧΗ: Η λίστα Βιβλιογραφικών Αναφορών θα πρέπει να δημιουργείται αυτόματα (από το πρότυπο εισαγωγής Βιβλιογραφίας του Επεξεργαστή Κειμένου με παράθεση όλων των προελεύσεων--πηγών των Αναφορών σας Για τη Διαχείριση των προελεύσεων μπορείτε να χρησιμοποιήσετε το στυλ IEEE2006 (κατά προτίμηση, εκτός και αν ο Επιβλέπων καθηγητής σας προτείνει να χρησιμοποιήσετε άλλο πρότυπο).*

*Εφιστάται η προσοχή σας στην ορθή καταχώρηση καθεμίας αναφοράς (μέσω του εργαλείου «Εισαγωγή Αναφοράς» από το μενού «Αναφορές»)*

*Παρακάτω παρατίθεται υπόδειγμα βιβλιογραφικών αναφορών που εισήχθησαν από το μενού: Αναφορές → βιβλιογραφία → Εισαγωγή Βιβλιογραφίας*


[1] Dr. M.F. Dossis: “CUSTOM COPROCESSOR COMPILATIONS USER GUIDE”, February 2020

[2] CCC C Front End User Guide (March 2022)

[3] Dr. Michael Dossis (Μιχαήλ Δόσης)

[4] Dr. Georgios Dimitriou (Γεώργιος Δημητρίου)

[5] https://www.britannica.com/technology/compiler (18/11/2021)

[8] https://web.archive.org/web/20121108043216/http://www.emu.edu.tr/aelci/Courses/D-318/D-318-Files/plbook/intro.htm (18/11/2021)

[6] Κ. Λάζος, Π. Κατσαρός, Ζ. Καραίσκος: “Μεταγλωττιστές Γλωσσών Προγραμματισμού: Θεωρία και Πράξη”, 3rd edition, Θεσσαλονίκη 2004

[9] D. Knuth, Luis Trabb Pardo: “The early development of programming languages” (STAN-CS-76-562), August 1976

[7] A. Aho, M. Lam, R. Sethi, J. Ullman: “Compilers: principles, techniques and tools”, 2nd edition, Pearson Addison Wesley 2007

[15] W. Vanderbauwhede, G. Davidson: “Domain-Specific Acceleration and Auto-Parallelization of Legacy Scientific Code in FORTRAN 77 using Source-to-Source Compilation” (arXiv:1711.04471v1), November 2018

[16] https://devopedia.org/transpiler (22/12/2021)

[17] https://tiobe.com/tiobe-index/ (28/12/2021)

[18] https://redmonk.com/sogrady/2021/08/05/language-rankings-6-21/ (28/12/2021)

[19] https://data-flair.training/blogs/applications-of-c/ (28/12/2021)

[20] https://www.codeavail.com/blog/uses-of-c-programming-language/ (28/12/2021)

[21] B. Kernighan: “A programming language called C” (DOI: 10.1109/mp.1983.6499601, DOI owner: Institute of Electrical and Electronics Engineers), December 1983

[65] B. Kernighan, D. Ritchie: “The C programming Language”, Prentice-Hall 1988

[66] https://www.geeksforgeeks.org (11/01/2022)

[22] R. Amiard, G. Hoffmann: “Introduction to Ada” (learn.adacore.com), June 2021

[25] https://www.redhat.com/en/blog/security-flaws-caused-compiler-optimizations (01/02/2022)

[27] https://compileroptimizations.com/ (01/02/2022)

[24] https://iq.opengenus.org/code-optimizations-in-compiler-design/ (01/02/2022)

[23] https://groups.seas.harvard.edu/courses/cs153/2019fa/lectures/Lec19-Optimization.pdf (Stephen Chong: "CS153: Compilers Lecture 19: Optimization") (01/02/2022)

[26] S. Muchnick: "Advanced compiler design and implementation", 1st edition, Morgan Kaufmann, 1997

[28] https://pages.cs.wisc.edu/~horwitz/CS704-NOTES/2.DATAFLOW.html (01/02/2022)

[29] "Goldberg, David. What Every Computer Scientist Should Know About Floating-Point Arithmetic, ACM Computing Surveys, Vol. 23, No. 1, Mar. 1991, pp. 5-48."

[67] https://www.math.uci.edu/~chenlong/226/FDM.pdf (Long Chen: "FINITE DIFFERENCE METHODS FOR POISSON EQUATION") (01/02/2022)

[68] "Allen, Frances E., John Cocke, and Ken Kennedy. Reduction of Operator Strength, in [Muchnick, Steven S. and Neil D. Jones (eds.). Program Flow Analysis: Theory

and Applications, Prentice-Hall, Englewood Cliffs, NJ, 1981], pp. 79-101."

[30] https://www.isi.edu/~youngcho/cse560m/vliw.pdf (Philips Semiconductors: "An Introduction To Very-Long Instruction Word (VLIW) Computer Architecture") (01/02/2022)

[31] https://www3.nd.edu/~zxu2/acms60212-40212/ic981219-pipelining.pdf (Jian Wang, Bogong Su: "SOFTWARE PIPELINING OF NESTED LOOPS FOR REAL-TIME DSP APPLICATIONS") (01/02/2022)

[10] https://cseweb.ucsd.edu/~kube/cls/12.s13/Lectures/lec16/lec16.pdf (12/02/2022)

[13] https://courses.cs.washington.edu/courses/cse401/08wi/lecture/AST.pdf (12/02/2022)

[14] https://lh5.googleusercontent.com/proxy/KGUHIURTgec23e9X2iXv2NuwFtbqSviWb9lWTOo3oSpo_u_JyI4X6xW4eS0kT6MLwt_Vy4lQwSnj6xM5qiU=w1200-h630-p-k-no-nu (12/02/2022)

[11] https://athena.ecs.csus.edu/~gordonvs/135/resources/04bnfParseTrees.pdf (12/02/2022)

[34] "man flex" command, flex version 2.6.4 (24/02/2022)

[36] "man bison" command, bison (GNU Bison) version 3.7.6 (24/02/2022)

[35] "info flex" command, flex version 2.6.4 (24/02/2022)

[37] "info bison" command, bison (GNU Bison) version 3.7.6 (24/02/2022) (can be found online at https://www.gnu.org/software/bison/manual/bison.pdf )

[12] https://groups.seas.harvard.edu/courses/cs153/2018fa/lectures/Lec06-LR-Parsing.pdf (Stephen Chong: "CS153: Compilers Lecture 6: LR Parsing") (07/03/2022)

[69] E. Dijkstra: "Go To Statement Considered Harmful", Communications of the ACM, Vol. 11, No. 3, March 1968, pp. 147-148.

[33] M. McCool, A. Robison, J. Reinders: "Structured Parallel Programming Patterns for Efficient Computation", Morgan Kaufmann 2012

[70] https://www.javatpoint.com/matrix-multiplication-in-c (11/03/2022)

[38] A. Balamane, Z. Taklit, R. Baghdadi: "Using Deep Neural Networks for Estimating Loop Unrolling Factor", (arXiv: 1911.03991v1) November 2019

[39] M. Kruse, H. Finkel: "Loop Optimization Framework", (arXiv:1811.00632v1) November 2018

[40] A. Kl ̈ockner: "Loo.py: transformation-based code generation for GPUs and CPUs", (arXiv:1405.7470v1) May 2014

[41] R. Navarathna, S. Radhakrishnan and R. Ragel: "Loop Unrolling in Multi-pipeline ASIP Design", (arXiv:1402.0671)

[42] I. HEDTKE: "METHODS OF MATRIX MULTIPLICATION (AN OVERVIEW OF SEVERAL METHODS AND THEIR IMPLEMENTATION)", (arXiv:1106.1347v1) June 2021

[43] M. Booshehri, A. Malekpour, P. Luksch: "An Improving Method for Loop Unrolling", (IJCSIS) International Journal of Computer Science and Information Security, Vol. 11, No. 5, May 2013

[44] M. Desai: "Inner loop optimizations in mapping single-threaded programs to hardware", (arXiv:1411.0863v1) November 2014

[45] M. Kruse: "Loop Transformations using Clang's Abstract Syntax Tree", (arXiv:2107.08132v1) July 2021

[46] K. Lata, S. Chhabra, S. Sain: "Hardware Software Co-design framework for Data Encryption in Image Processing Systems for the Internet of Things Environment", (arXiv:2111.14370v1) November 2021

[47] W. Jradia, H. do Nascimento W. Martins: "A Fast and Generic GPU-Based Parallel Reduction Implementation", (arXiv:1710.07358v1) October 2017

[48] J. Shirako, A. Hayashi, S. Raj Paul, A. Tumanov, V. Sarkar: "Automatic Parallelization of Python Programs for Distributed Heterogeneous Computing", (arXiv:2203.06233v1) March 2022

[49] S. Gorantala, R. Springer, S. Purser-Haskell, W. Lam, R. Wilson, A. Ali, E. P. Astor, I. Zukerman, S. Ruth, C. Dibak, P. Schoppmann, S. Kulankhina, A. Forget, D. Marn, C. Tew, R. Misoczki, B. Guillen, X. Ye, D. Kraft, D. Desfontaines, A. Krishnamurthy, M. Guevara, I. M. Perera, Y. Sushko, B. Gipson: "A General Purpose Transpiler for Fully Homomorphic Encryption", (arXiv:2106.07893v1) June 2021

[50] P. Kumbhar, O. Awile, L. Keegan, J. Blanco Alonso, J. King, M. Hines, F. Schürmann: "An optimizing multi-platform source-to-source compiler framework for the NEURON MODeling Language", (arXiv:1905.02241v1) May 2019

[51] M. Bornea , R. Fernandez Astudillo, T. Naseem, N. Mihindukulasooriya, I. Abdelaziz, P. Kapanipathi, R. Florian, S. Roukos: "Learning to Transpile AMR into SPARQL",

(arXiv:2112.07877v1) December 2021

[52] http://c2ada.sourceforge.net/c2ada.html (17/03/2022)

[53] https://github.com/mikequentel/c2ada (17/03/2022)

[54] http://www.knosof.co.uk/ctoa.html (17/03/2022)

[32] Óscar Lucía, Eric Monmasson, Denis Navarro, Luis A.Barragán, Isidro Urriza, José I.Artigas: “Control of Power Electronic Converters and Systems”, Academic Press, 2018

[55] Xiangyu Gao, Divya Raghunathan, Ruijie Fang, Tao Wang, Xiaotong Zhu, Anirudh Sivaraman, Srinivas Narayana, Aarti Gupta: "High-Level Synthesis for Packet-Processing Pipelines", (arXiv:2211.06475v1) November 2022

[56] Stephanie Soldavini, Donatella Sciuto, Christian Pilato: "Iris: Automatic Generation of Efficient Data Layouts for High Bandwidth Utilization", (arXiv:2211.04361v1) November 2022

[57] Ashwin Bhat, Adou Sangbone Assoa, Arijit Raychowdhury: "Gradient Backpropagation based Feature Attribution to Enable Explainable-AI on the Edge", (arXiv:2210.10922v1) October 2022

[58] Gabriele Montanaro, Andrea Galimberti, Ernesto Colizzi, Davide Zoni: "Hardware-Software Co-Design of BIKE with HLS-Generated Accelerators", (arXiv:2209.03830v1) September 2022

[59] Zhiqiang Que, Hongxiang Fan, Marcus Loo, Michaela Blott, Maurizio Pierini, Alexander D Tapper, Wayne Luk: "LL-GNN: Low Latency Graph Neural Networks on FPGAs for Particle Detectors", (arXiv:2209.14065v3) October 2022

[60] Gabriel Rodriguez-Canal, Nick Brown, Yuri Torres, Arturo Gonzalez-Escribano: "Programming abstractions for preemptive scheduling in FPGAs using partial reconfiguration", (arXiv:2209.04410v1) August 2022

[61] Xingyu Tian, Zhifan Ye, Alec Lu, Licheng Guo, Yuze Chi, Zhenman Fang: "SASA: A Scalable and Automatic Stencil Acceleration Framework for Optimized Hybrid Spatial and Temporal Parallelism on HBM-based FPGAs", ACM Trans. Reconfig. Technol. Syst. 1, 1, Article 1 (arXiv:2208.10770v1) January 2022

[62] SOHEIL NAZAR SHAHSAVANI, ARASH FAYYAZI, MAHDI NAZEMI, MASSOUD PEDRAM: "Efficient Compilation and Mapping of Fixed Function Combinational Logic onto Digital Signal Processors Targeting Neural Network Inference and Utilizing High-level Synthesis", (arXiv:2208.00302) 2022

[63] Mang Yu, Sitao Huang, Deming Chen: "Chimera: A Hybrid Machine Learning Driven Multi-Objective Design Space Exploration Tool for FPGA High-Level Synthesis", (arXiv:2207.07917v1) July 2022

[64] Seong-Kyun Mok, Seoyeon Kang, Jeongwoo Kim, Eun-Sun Cho, Seokwoo Choi: "SSLEM: A Simplifier for MBA Expressions based on Semi-linear MBA Expressions and Program Synthesis", (arXiv:2208.05612v2) August 2022

# 6. Παράρτημα A (Appendix A: C and Ada syntax)

Below is an elaboration of the syntax of the C and Ada languages.

## 6.1 C's syntax

A C source code file is usually given a ".c" extension.

Everything is case sensitive.

### 6.1.1 Comments

Comments traditionally begin with "/*" and end with "*/". This type can span multiple lines until the ending is found. A comment can be inserted anywhere within the .c file and anything within it will be completely ignored. Another way to comment is by starting with "//" on a line. This type turns anything after it into a comment and spans only one line.

### 6.1.2 The '#' directive and string parameters in standard library functions

The "#" directives give commands to the compiler and are usually placed at the start of the .c file.

The "#include" directive is used to include functions from libraries or "header" files, usually with a ".h" extension. By convention, standard libraries are included by giving their filename between a less than ("<") and a greater than (">") character, and custom header files included in the same directory as the .c file are enclosed between two double quotes ('"'). A header file can in turn include other header files as well.

The "#define" directive declares constants; replacements of a string of characters (usually in upper case to differentiate them from variable and function names), numbers and underscores by a value (usually numeric). E.g. in "#define MAX_UNROLL_FACTOR 10" the value "10" is going to replace every appearance of the string "MAX_UNROLL_FACTOR" unless it's within double quotes (e.g. a string within a "printf" function. ("printf" is one of the functions provided by including the <stdio.h> library. It accepts input to print to stdout (usually the screen or terminal window) as the first argument, and values in the second, third, and so on. The values in a string (what the input is) replace special combinations of characters such as "%d" ("decimal", for ints), "%f" (for floats), "%c" (for single characters) and "%s" (for strings).)).

### 6.1.3 Value and type management, functions, naming and scope

Values all have a type in C (although coercions might change them) and variables are used to store values of a specified type. Variable and function types can be coerced to a different type by "casting" (e.g. .

"

float a, b = 1.2;

int c;

a = b + 1;  // Coercion: "1" will automatically be converted to "1.0".

c = (int)b;  // Casting: "c" will get "b"'s rounded value: 1.

"

) but unlike immediate value usage must be explicit; otherwise, an error is thrown. A C program consists of at least one function called "main", where execution always starts and that may call other functions (either defined in a .h file or within the same source file above or below the "main" function. If the function is defined below the "main" function, its declaration must be declared above "main".). Valid variable and function names can be described by the regex "[a-zA-Z_][a-zA-Z0-9_]*" excluding some reserved words such as "else" and "while" (among others).

A basic function declaration (the first line) and its body (the rest of it) looks something like the example below:

"

int squareInt (int i) // this line is the declaration of the function

{

    return i*i;

}

"

Functions all return a value of a type (the first word in the example), be it custom (such as a struct or union) or standard, including the "void" type, where the return statement can be omitted (as "void" indicates a lack of a value). "void" can also be used when casting. The input that the function expects is declared between a "(" and a ")" which should always be present, even if there is no input. If there are multiple input variables they are separated by a ",". A function's body begins with a "{" and ends with a "}" (everything within these characters is in a separate scope). Once the return statement is found and executed in the function no further code within the function call

is run. To run or call a function (within another, such as "main") type its name along with input (if any) as an argument where the declared input existed omitting the type and replacing the variable name with a value or a valid initialized variable of the same type (eg. "squareInt(2);" or "int j; j = 1; squareInt(j);"). The ";" character separates commands from each other and indicate their end and a new beginning after their appearance.

### 6.1.4 Standard types and sizes

The standard types of values are the following:

"

int

float

double

char

"

"int" is an integer, "float" and "double" are decimals (separated by a ".") and "char" is for characters. All of these support negative values, although all printable "char" characters are above 0. Additionally, "short", when included before "int" or on its own declares an integer with a smaller value than plain "int". Likewise, "long", which can also be applied to "double" (unlike "short") signifies an "int" or "double" with a size bigger than the default. When "short" and "long" appear on their own they by default declare integers. Although each machine/compiler might assign different sizes for the amount of memory each declared variable holds, it is guaranteed that "short" <= "int" <= "long" and "float" <= "double" <= "long double" in size. "unsigned" can be applied to integers to double their size at the expense of not supporting negative values. If a char takes up one byte, an unsigned char (which is also considered an integer) has valid values in the range of 0 to 255, enough for every character in the extended ASCII table. Special characters are usually "escaped" (they consist of a "\" followed by another character), for example '\t' is one character and indicates a tab. '\n' is a new line. To refer to a single character, it should be enclosed between two "'"s (single quotes).

### 6.1.5 Variable declaration and scope

Variables can be declared outside functions (called "globals") or anywhere within a function, although putting all declarations at the beginning of the function's body is considered good practice. When a variable is declared with the same name within a different scope, the newest declaration is valid until exiting the scope of the latest

declaration. The scope immediately before the one that ended then takes validity, as shown in the example below.

```
"
#include <stdio.h>

int a = 1;

void printA()
{
    int a = 2;

    printf("A = %d\n", a);
}

int main()
{
    printA(); // Prints "A = 2".

    printf("A = %d\n", a); // Prints "A = 1".

    return 0;
}
"
```

It is not possible to re-declare a variable with the same name within the same scope (regardless of type); an error is thrown. A variable can not be used before its declaration. Variables can be initialized with a value on declaration using the "=" ("assignment") operator, and multiple declarations of variables of the same type can be done in the same command as long as they are separated by a  "," (e.g. "float a, b = 1.2;"). The assignment is optional in the declaration. Uninitialized variables contain random and generally unpredictable values ("garbage").

### 6.1.6 Compound or derived data types

In addition to ordinary type variables, derived data types exist, such as arrays, structures, unions and pointers. It should be noted that constants can be declared by appending the keyword “const” in front of a declaration that would otherwise have been a variable (e.g. “int I = 0;” declares a variable, “const int i = 0;” declares a constant; the value can not be changed once it is set, so the second “i” is immutable).

### 6.1.7 Pointers and addresses

A pointer is declared by appending a "*" to the left of a variable name. The "*" symbol is called the "dereference" operator and can be used when declaring any type (e.g. a pointer array of a structure type). When using a variable that was declared as a pointer without the "*" operator, it refers to an address. When the "*" is present, it refers to the contents of where the pointer variable points to. It is possible to have pointers to pointers (and pointers to pointers to pointers and so on). The "reference" operator, "&", can be applied anywhere the dereference operator can be applied excluding variable declaration (something like "int b; int *p = &b;" is legal, however) and works in the opposite way. A "&" to the left of an ordinary operand refers to its address rather than its contents. Trying to use an uninitialized pointer will cause a segmentation fault (crash) during runtime, and referencing a variable too many times will cause an error during compilation. An example is provided to demonstrate pointer use:

```
"
#include <stdio.h>


int main()
{
    int a;
    int *p = &a, **p2 = &p;


    printf("a = %d\n", a);
    printf("&a = %d\n", &a);
    printf("p = %d\n", p);
    printf("*p = %d\n", *p);
    printf("*&*p = %d\n", *&*p);
```

```
        printf("&p = %d\n", &p);
        printf("p2 = %d\n", p2);
        printf("*p2 = %d\n", *p2);
        printf("**p2 = %d\n", **p2);
        printf("&p2 = %d\n", &p2);

        return 0;
}

/*
output:

a = 0
&a = 1811352364
p = 1811352364
*p = 0
*&*p = 0
&p = 1811352352
p2 = 1811352352
*p2 = 1811352364
**p2 = 0
&p2 = 1811352344
*/
"
```

### 6.1.8 Arrays, dimensions and indexing

An array is declared by appending a "[", the size (always an integer or an integer constant, variable use is not allowed) and a "]" (e.g. "int a[5];"). Arrays always start from 0 and end at <whatever their size was declared> -1 (so in the previous example "a[4]" is the last place). Arrays can also hold a structure, union, or a pointer to any type. Although

arrays aren't pointers, they can be accessed by using the pointer form as well (for the previous example, "a[4]" is equivalent to "*(a+4)"). It is also possible to have multiple dimensions (e.g. "a[4][2][8]", which can also be written as "*(*(*(a+4)+2)+8)" provided this is not its declaration). Arrays can also be initialized using a form similar to "{{1,4}, {2,8,16}}", omitting places will initialize only the first places provided with a value.

### 6.1.9 Strings (or char arrays)

Strings don't exist in C, but char arrays can fulfill this purpose. The '\0' character indicates the end of a string and is required at the end of one for the correct function of standard library functions; it is automatically appended to the end when using '"' (multiple chars, or "string") notation. It is also possible to set a char pointer to a string value. The example below demonstrates valid string usage:

```
"
#include <stdio.h>

int main()
{
    char straasn[] = "test0";
    char strfasn[20] = "test1";
    char straaan[] = {'t', 'e', 's', 't', '2', '\0'};
    char strfaan[20] = {'t', 'e', 's', 't', '3', '\0'};
    char *ptrstr = "test4";

    printf("straasn = %s\n", straasn);
    printf("strfasn = %s\n", strfasn);
    printf("straaan = %s\n", straaan);
    printf("strfaan = %s\n", strfaan);
    printf("ptrstr = %s\n", ptrstr);

    return 0;
}
```

```
/*
output:
straasn = test0
strfasn = test1
straaan = test2
strfaan = test3
ptrstr = test4

*/
"
```

### 6.1.10 Structs, component referral, dynamic memory management and typedef

Structures are types that can hold more variables within them accessed by appending a "." to the right of their name followed by the name of the variable. An example initialization and use is shown:

```
"
#include <stdio.h>

int main()
{
    struct
    {
        int i1, i2, ai[2][2];
        float f1;
        struct { int si1, si2; } iis;
        char c;
    } s1, s2 = {1, 2, {{6, 7}, {8, 9}}, 3.14, {4, 5}, 'c'};

    s1.iis.si2 = 3;
```

```
        printf("s2.ai[0][0] = %d\n", s2.ai[0][0]);
        printf("s2.ai[0][1] = %d\n", s2.ai[0][1]);
        printf("s1.iis = %d\n", s1.iis);
        printf("s1.iis.si2 = %d\n\n", s1.iis.si2);

        return 0;
}
/*
output:

s2.ai[0][0] = 6
s2.ai[0][1] = 7
s1.iis = 32568
s1.iis.si2 = 3

*/
"
```

Structs can also be assigned names to be used in a later statement and can be ended without an immediate declaration. The "->" operator is used instead of "." to access elements in pointers to structs, although it is also possible to access a pointer to a struct's element in the form "(*sp).element;".

The "sizeof()" operator returns the size of a type in bytes. "malloc()" is a function provided by stdlib.h; it allocates memory for a complex type. The "free()" function, also provided by stdlib.h, deallocates the memory. The following example includes some of the elements mentioned:

```
"
#include <stdio.h>
#include <stdlib.h>

struct ourStruct
{
```

```
    int v;
    struct ourStruct *next;
};

int main()
{
    struct ourStruct *os, *osprev;

    os = osprev = (struct ourStruct*)malloc(sizeof(struct ourStruct));
    os->next = (struct ourStruct*)malloc(sizeof(struct ourStruct));
    os->next->v = 2;
    os = (*os).next; // Equivalent to "os = os->next;"

    printf("os->v = %d\n", os->v); // os->v = 2

    free(osprev);
    free(os);

    return 0;
}
"
```

"typedef" can be used to set names for complex types such as structs, so instead of declaring "struct sn { ... }; struct sn si;" one could instead type "typedef struct { ... } sn; sn si;". In the second example "sn" is the second argument of typedef and tells it what name to associate the struct with; it is not an instantiation of the struct. The struct can still be given a name though, right after the keyword "struct", as in the previous example.

### 6.1.11 Enums

Enums are similar to constants. Each value in an enum has an ascending int value related to its index within it by default, but custom values can be inserted explicitly

instead as well. The next value in an enum is what the one before it is +1. An enum instance is not necessary for a value to be used. Enums can not have values named the same way they are in another enum within the same scope. The example below provides a demonstration:

```
"
#include <stdio.h>

typedef enum // Anonymous enum
{
    ZERO,
    ONE
} etd; // typedef value given here

enum eStartAt1 // Named enum
{
    TWO = 2,
    THREE,
    FOUR
};

enum eCustomVals
{
    VAL00 = 4,
    VAL01 = 4,
    VALA,
    VAL10 = 2,
    VAL11 = 2,
    VAL12 = 2,
    VALB,
    VAL2 = 'a'
} ecvi; // enum instance declared here
```

```
int main()
{
    enum { ONE = 8 };
    etd etdi;
    enum eStartAt1 esa1 = THREE;

    etdi = TWO;
    ecvi = VALA;

    printf("ZERO = %d\n", ZERO);
    printf("ONE = %d\n", ONE);
    printf("esa1 = THREE = %d\n", esa1);
    printf("etdi = TWO = %d\n", etdi);
    printf("ecvi = VALA = %d\n", ecvi);

    ecvi = VALB;

    printf("ecvi = VALB = %d\n", ecvi);

    ecvi = VAL2;

    printf("ecvi = VAL2 = %d\n", ecvi);

    return 0;
}

/*
output:

ZERO = 0
```

```
ONE = 8
esa1 = THREE = 3
etdi = TWO = 2
ecvi = VALA = 5
ecvi = VALB = 3
ecvi = VAL2 = 97
*/
"
```

### 6.1.12 Unions and memory allocation

Unions are similar to structs in syntax. The difference is in how memory is handled. In structs, each member (variable) gets its own separate storage of the size necessary. That is, the collective memory allocated for a struct instance is equal to the size of all of its members combined. A union only allocates memory for the largest member in its declaration. This is, of course, enough for the smaller members to operate, so only one member can be used at a time without being corrupted.

The example below is provided to clarify:

```
"
#include <stdio.h>

union ourUnion
{
    char c1;
    char c2;
    int i;
};

int main()
{
    union ourUnion ou;
```

```
    ou.c1 = 'x';

    printf("ou.c1 = %c\n", ou.c1); // ou.c1 = x

    ou.c2 = 'T';

    printf("ou.c1 = %c\n", ou.c1); // ou.c1 = T

    ou.i = 2736757;

    printf("ou.c1 = %c\n", ou.c1); // ou.c1 = u
    printf("ou.i = %d\n", ou.i); // ou.i = 2736757

    return 0;
}
"
```

### 6.1.13 Operators and precedence

There are multiple assignment operations in C, usually involving the operand to its left as an implied right operand immediately after the assignment operator itself (excluding the ordinary "=" operator). The "=" operator assigns the variable to its left to the result of the expression on its right. These can be cascaded (provided the assignments are not attempted on variable declaration, where only one is allowed per variable declaration). As shown in the example below, the rightmost value gets assigned first:

```
"
#include <stdio.h>

int main()
{
    int a, b, c, d;
```

```
    c = 2;
    b = 100;

    a = b += c = d = 8 / 2;

    printf("a = %d\n", a); // a = 104
    printf("b = %d\n", b); // b = 104
    printf("c = %d\n", c); // c = 4
    printf("d = %d\n", d); // d = 4

    return 0;
}
```

"

Arithmetic operators include "+" for addition, "-" for negation (so it can be used without a numeric left operand) or subtraction, "*" for multiplication, "/" for division, "%" for modulus (division remainder). "++" to increment and "--" to decrement by one apply to variables, when on the right of them they are in/decremented after the statement they are in has been executed. When on the left, the addition/subtraction happens before the rest of the statement. The example below should clarify the difference:

"

```
#include <stdio.h>

int main()
{
    int a, b, c;

    a = 1;
    b = 5;

    c = b++ + ++a;
```

```
    printf("a = %d\n", a); // a = 2
    printf("b = %d\n", b); // b = 6
    printf("c = %d\n", c); // c = 7
    printf("c = %d\n", ++c); // c = 8
    printf("c = %d\n", c++); // c = 8

    return 0;
}
"
```

Bitwise operators can only be used on integer and character types, whether signed or unsigned, and are called so because they operate on the binary value of the ints or characters. These include "&": AND, "|": OR (inclusive), "^": XOR (exclusive), "~": one's complement (inversion), "<<": left shift, ">>": right shift. The shift operators do not rotate bits, and shifting bits out of range will not throw an error, but simply replace the empty space with zeroes, as shown in the example below:

```
"
#include <stdio.h>

int main()
{
    int b, a = 1;

    b = a << 4;

    printf("b = %d\n", b); // b = 16

    b = a >> 5;

    printf("b = %d\n", b); // b = 0

    return 0;
```

```
}
"
```

In C there are numeric/assignment combo operators such as "+=", "-=", "*=", "/=", "%=", "<<=", ">>=", "&=", "|=" and "^=". As an example "i += 2;" is equivalent to "i = i + 2;".

The last sets of non-unary operators are the relational, equality and logical operators. The relational operators are ">" (is greater than), ">=" (is greater than or equal to), "<" (is less than), "<=" (is less than or equal to). Equality operators include "==" (is equal to) and "!=" (is not equal to). Logical operators include "&&": AND and "||": OR. There is also the unary negation operator "!". By definition, the numeric value of a relational or logical expression is 1 if the relation is true, and 0 if the relation is false. Expressions connected by "&&" or "||" are evaluated left to right, and evaluation stops as soon as the truth or falsehood of the result is known. The example below is provided to help clarify:

```
"
#include <stdio.h>

int main()
{
    int a = 2, b = 1;
    int c = ((!a) != b);

    printf("a == b = %d\n", a == b);
    printf("c = %d\n", c);
    printf("(!a) = %d\n", !a);
    printf("!(a == b) = %d\n", !(a == b));
    printf("(!a) > b = %d\n", (!a) > b);
    printf("a > b = %d\n", a > b);
    printf("a && b = %d\n", a && b);
    printf("(!a) && b = %d\n", (!a) && b);
    printf("1 || (!a) && b = %d\n", 1 || (!a) && b);

    return 0;
```

```
}

/*
output:

a == b = 0
c = 1
(!a) = 0
!(a == b) = 1
(!a) > b = 0
a > b = 1
a && b = 1
(!a) && b = 0
1 || (!a) && b = 1
*/
"
```

Using "(" and ")" we can change the default order of operations.

The ternary operators "?" and ":", also sometimes called "select" always go together. If the value to the left of "?" is anything but 0, then the value to the left of ":" is run, otherwise the value to the right of ":" is run instead (only one of the two will be run). Example:

```
"
#include <stdio.h>

int main()
{
    int a = -2;

    4 ? printf("True\n") : printf("False\n");
    0 ? printf("True\n") : printf("False\n");
    a ? printf("a is not 0\n") : printf("a is 0\n");
```

```
    return 0;
}
/*
output:

True
False
a is not 0
*/
"
```

Note the lack of a ";" after the statement to the right of "?". It isn't possible to have a "return" statement with the ternary operators (something like "a ? return 1 : return 0;" would be illegal).

Operator precedence is shown in the following image:

| Operators | Associativity |
|---|---|
| `() [] -> .` | left to right |
| `! ~ ++ -- + - *(type) sizeof` | right to left |
| `* / %` | left to right |
| `+ -` | left to right |
| `<< >>` | left to right |
| `< <= > >=` | left to right |
| `== !=` | left to right |
| `&` | left to right |
| `^` | left to right |
| `\|` | left to right |
| `&&` | left to right |
| `\|\|` | left to right |
| `?:` | right to left |
| `= += -= *= /= %= &= ^= \|= <<= >>=` | right to left |
| `,` | left to right |

**Image 70. C operator precedence**

Πηγή: **[65]**

In words,

"() [] -> ." have the highest precedence and are evaluated in the left to right order (hereon "LtR" for brevity),

next "! ~ ++ -- + - *(type) sizeof": right to left ("RtL"),

"* / %": LtR,

"+ -": LtR,

"<< >>": LtR,

"< <= > >=": LtR,

"== !=": LtR,

"&": LtR,

"^": LtR,

"|": LtR,

"&&": LtR,

"||": LtR,

"? :": RtL,

"= += -= *= /= %= &= ^= |= <<= >>=": RtL,

and last, ",": LtR.

Unary "&", "+", "-" and "*" have higher precedence than the binary forms.

### 6.1.14 Keywords, loops and conditions

Inside functions it is possible to use "if()", "if() ... else", "switch" and loops including "while", "do ... while" and "for(...;...;...)".

The "if" clause runs the code within "{" and "}" immediately after it (or only the statement following it, if the braces are excluded) only if the value inside the parenthesis following the "if" keyword evaluates to a "true" (non-zero) value. It is possible to trail it with the "else" keyword that will run if the statement evaluates to "false" (zero). "if"

statements can be nested, but the "else" keyword associates with the "if" immediately before it. The example below demonstrates the "if" clause:

```
"
#include <stdio.h>

int main()
{
    int a = 0, b = 1;

    if(a)
        printf("a is true\n");
    else
        if(b >= 1) // An "if ... else" clause counts as one statement, so no braces are needed for the "else" of "if(a)".
        { // "if(b >= 1)" has more than one statement though, so braces must be used.
            printf("b >= 1\n");

            if (b == 1)
                printf("b == 1\n");

            b = 2;
        }
        else
        {
            printf("b < 1\n");
            b = 1;
        }
    // This line and everything that follows it is outside the scope of "if(a)"'s "else".

    return 0;
```

```
}

/*
output:

b >= 1
b == 1
*/
"
```

The switch statement will run the code below it when a "case"'s value is matched with the input. Without a "break;" keyword, however, all code below it will run, including "case"'s that do not match the input. The "default" keyword can be used to run a section of code if there was no match within the entire "switch" statement. Switch statements are demonstrated in the following example:

```
"
#include <stdio.h>

int main()
{
    char c1 = 'b';
    int i1 = 5;

    switch(c1)
    {
            case 'a':
                    printf("c1 = a\n");
                    break;
            case 'b': // this case is true
            case 'c':
                    switch(i1)
                    {
```

```
                        case 1: case 2: case 3:
                                printf("i1 < 4\n");
                        default:
                                printf("i1 >= 4\n");
                }
        case 'd':
                printf("End of fallthrough due to a break keyword\n");
                break;
        case 'e':
                printf("This will not be printed\n");
    }

    return 0;
}

/*
output:

i1 >= 4
End of fallthrough due to a break keyword
*/
"
```

"break" can be used to stop execution of a loop, as will be shown. The "continue" keyword can only be used inside loops. Any code beyond these within the same scope will not be run. "continue;" stops execution at a point and starts the next iteration of the loop instead of exiting altogether, contrary to what "break;" does.

The scope for loops is similar to the "if" clause. The simplest loop in C is the "while" loop, with the syntax being "while() ..." or "while() { ... }". The statement(s) following it will only run for as long as what is between the "(" and the ")" evaluates to non-zero. If the while clause evaluates to true, the statements after it are run. When they finish they return to the clause again, and if it is still true, they run again. It is possible to have an infinite loop. A "do ... while()" or "do { ... } while()" loop is the same as a simple "while" loop, with the difference that the clause is evaluated after it has run at least once.

This example demonstrates some of the keywords explained:

"

```
#include <stdio.h>

int main()
{
    int i = 0, j = 0, k = 0;

    while(-1) // This will (theoretically) never end
    {
            while(i < 2)
            {
                    while(j < 2)
                            printf("k = %d, j = %d, i = %d\n", k, j++, i);
                    j = 0;
                    i++;
            }

            k++;
            i = j = 0;

            if(k <= 1)
                    continue;

            printf("This will not run the first time\n");

            if(k >= 5)
                    break; // This will end the infinite loop
    }
```

```
    while(0)
            printf("This will never run\n");

    do
            printf("This will run at least once\n");
    while(0);

    return 0;
}

/*
output:

k = 0, j = 0, i = 0
k = 0, j = 1, i = 0
k = 0, j = 0, i = 1
k = 0, j = 1, i = 1
k = 1, j = 0, i = 0
k = 1, j = 1, i = 0
k = 1, j = 0, i = 1
k = 1, j = 1, i = 1
This won't run the first time
k = 2, j = 0, i = 0
k = 2, j = 1, i = 0
k = 2, j = 0, i = 1
k = 2, j = 1, i = 1
This won't run the first time
k = 3, j = 0, i = 0
k = 3, j = 1, i = 0
k = 3, j = 0, i = 1
```

k = 3, j = 1, i = 1

This won't run the first time

k = 4, j = 0, i = 0

k = 4, j = 1, i = 0

k = 4, j = 0, i = 1

k = 4, j = 1, i = 1

This won't run the first time

This will run at least once

*/

"

The last type of loop is the "for" loop. It always has two ";"s within its' parentheses. The syntax is "for(...;...;...)". The first argument is usually used for variable initializations, the second is the condition to check (like what is within the parenthesis in a "while" loop) and the third runs at the end of every iteration. The example below shows a for loop and a while that is equivalent to it:

"

```
#include <stdio.h>


int main()
{
    int i, j, k, a[5][5];


    j = 0;
    while(j < 5)
    {
            i = 0; // first statement in a for loop
            while(i < 5) // second statement in a for loop
            {
                    a[i][j] = i + j;
                    i++; // third statement in a for loop
            }
```

```
            j++;
    }

    for(j = 0; j < 5; j++)
            for(k = 0, i = 0; i < 5; i++, k++)
                    printf("a[%d][%d] = %d\n", i, j, a[i][j]);

    printf("k = %d\n", k);

    return 0;
}

/*
output:

a[0][0] = 0
a[1][0] = 1
a[2][0] = 2
a[3][0] = 3
a[4][0] = 4
a[0][1] = 1
a[1][1] = 2
a[2][1] = 3
a[3][1] = 4
a[4][1] = 5
a[0][2] = 2
a[1][2] = 3
a[2][2] = 4
a[3][2] = 5
a[4][2] = 6
```

```
a[0][3] = 3
a[1][3] = 4
a[2][3] = 5
a[3][3] = 6
a[4][3] = 7
a[0][4] = 4
a[1][4] = 5
a[2][4] = 6
a[3][4] = 7
a[4][4] = 8
k = 5
*/
"
```

As shown, there can be more than one statement in the first and third section of "for", separated by a ",".

The “goto” statement works much like jump instructions (usually “JMP”) in assembly: it will jump to the line of code annotated with the label stated to the right of the “goto” statement. The label should be followed by a colon (“:”), but not when being referred to. This example should help clarify:

```
"
#include <stdio.h>

int main() {
    int i, j;

    j = 0;
here1:      i = 0; // "here1" is a label.
    printf("i has been reinitialized to %d\n", i);

here2:      ++i;
    ++j;
    printf("i is now %d\n", i);
```

```
    if (i < 5) {
            goto here2;
    } else {
            if (j > 15)
                    goto exit;
    }

    goto here1;

exit:
    return 0;
}

/*
output:
i has been reinitialized to 0
i is now 1
i is now 2
i is now 3
i is now 4
i is now 5
i has been reinitialized to 0
i is now 1
i is now 2
i is now 3
i is now 4
i is now 5
i has been reinitialized to 0
i is now 1
i is now 2
i is now 3
```

```
i is now 4
i is now 5
i has been reinitialized to 0
i is now 1
i is now 2
i is now 3
i is now 4
i is now 5

*/
"
```

(It should be noted that “goto” usage is discouraged and should be avoided in most conventional cases; see [69]. )

### 6.1.15 Function pointers, recursion

Functions can be recursive (call an instance of themselves), as shown in the following example:

```
"
#include <stdio.h>

int factorialFun(int n)
{
    if(n <= 1)
        return n;
    else
        return n * factorialFun(n-1);
}

int main()
{
```

```
    printf("The factorial of 5 is %d\n", factorialFun(5)); // The factorial of 5 is 120


    return 0;
}
"
```

Recursion can also lead to infinite calls; the equivalent to an infinite loop if used incorrectly/appropriately.

It is also possible to declare pointers to functions. This way, it is possible to call different functions by the same name (provided their input is the same, as well as what they return). The demonstration below is a slightly modified example from [66]:

```
"
#include <stdio.h>


void add(int a, int b)
{
    printf("a + b =  %d\n", a + b);
}


void subtract(int a, int b)
{
    printf("a - b = %d\n", a - b);
}


void multiply(int a, int b)
{
    printf("a * b = %d\n", a * b);
}


int main()
{
    // fun_ptr_arr is an array of function pointers
```

```
        void (*fun_ptr_arr[])(int, int) = {add, subtract, multiply};
        int a = 15, b = 10;

        // 0 for add, 1 for subtract and 2 for multiply
        (*fun_ptr_arr[1])(a, b);
        (*fun_ptr_arr[0])(a, b);
        (*fun_ptr_arr[2])(a, b);

        return 0;
    }
    /*
    output:

    a - b = 5
    a + b =  25
    a * b = 150
    */
"
```

## 6.2 Ada's syntax

A Ada source code file is given an ".ada" extension if the package declaration and body are in the same file. (This is what the csense compiler of this paper's project outputs. The GNAT compiler, however, requires the declarations to be in a ".ads" file, and the body in a separate".adb" file.)

Only one line comments exist in Ada. They start with a "--". Everything after encountering these two characters together will be ignored until the next line.

Everything that follows is either based on or directly quotes [22].

### 6.2.1 Package and library inclusion, procedures

The "with" clause is Ada's equivalent of C's "#include" clause. With it libraries and packages can be included in the project. Unlike C, an Ada program can consist of "procedures" (equivalent to a C function that returns "void") with any given name. The example below demonstrates a simple Ada program:

"

```
with Ada.Text_IO;


procedure Greet is
begin
   -- Print "Hello, World!" to the screen
   Ada.Text_IO.Put_Line ("Hello, World!");
end Greet;
```

"

"Ada.Text_IO" is a system library. "Greet" is the procedure's name and could be any valid variable name. From "begin" and until "end <procedure_name>;" is encountered, the body of the procedure is defined. "Put_Line" is similar to C's "printf", with the difference that it adds a new line at the end of its input string. By adding a "use" clause after the "with" clause ("use Ada.Text_IO;"), the necessity to type "Ada.Text_IO." in front of "Put_Line" is removed.

"The type "Integer" is a predefined signed type, and its range depends on the computer architecture. On typical current processors "Integer" is 32-bit signed." Variables are declared after the "is" and before the "begin" keywords in a procedure.

### 6.2.2 The “if” clause and basic input/output

"The if statement minimally consists of the reserved word "if", a condition (which must be a Boolean value), the reserved word "then" and a non-empty sequence of statements (the then part) which is executed if the condition evaluates to True, and a terminating "end if;"." Before the "end if;" multiple "elsif" and/or one "else" section can be included (optionally). The "Get" function reads input from "stdin" (equivalent to C's "scanf"). The "Put" function is the same as "Put_Line" with the difference that it does not add a new line at the end.

"

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;


procedure Check_Direction is
   N : Integer;
begin
   Put ("Enter an integer value: "); -- Puts a String
   Get (N); -- Reads an Integer
   Put (N); -- Puts an Integer
   if N = 0 or N = 360 then
         Put_Line (" is due east");
   elsif N in 1 .. 89 then
         Put_Line (" is in the northeast quadrant");
   elsif N = 90 then
         Put_Line (" is due north");
   elsif N in 91 .. 179 then
         Put_Line (" is in the northwest quadrant");
   elsif N = 180 then
         Put_Line (" is due west")
   elsif N in 181 .. 269 then
         Put_Line (" is in the southwest quadrant");
```

```
    elsif N = 270 then
        Put_Line (" is due south");
    elsif N in 271 .. 359 then
        Put_Line (" is in the southeast quadrant");
    else
        Put_Line (" is not in the range 0..360");
    end if;
end Check_Direction;
"
```

"This example expects the user to input an integer between 0 and 360 inclusive, and displays which quadrant or axis the value corresponds to. The in operator in Ada tests whether a scalar value is within a specified range and returns a Boolean result."

### 6.2.3 Loops, range, parameters and variables

"Ada has three ways of specifying loops. They differ from the C / Java / Javascript for-loop, however, with simpler syntax and semantics in line with Ada's philosophy."

"The first kind of loop is the "for" loop, which allows iteration through a discrete range."

```
"
with Ada.Text_IO; use Ada.Text_IO;

procedure Greet_5a is
begin
    for I in 1 .. 5 loop
        Put_Line ("Hello, World!" & Integer'Image (I)); --
        --                                  ^ Procedure parameter
    end loop;
end Greet_5a;
"
"
```

Executing this procedure yields the following output:

Hello, World! 1

Hello, World! 2

Hello, World! 3

Hello, World! 4

Hello, World! 5

"

"A few things to note:

1 .. 5 is a discrete range, from 1 to 5 inclusive.

The loop parameter I (the name is arbitrary) in the body of the loop has a value within this range.

I is local to the loop, so you cannot refer to I outside the loop.

Although the value of I' is incremented at each iteration, from the program's perspective it is constant. An attempt to modify its value is illegal; the compiler would reject the program.

Integer'Image is a function that takes an Integer and converts it to a String. It is an example of a language construct known as an attribute, indicated by the "'" syntax.

The & symbol is the concatenation operator for String values

The end loop marks the end of the loop

The "step" of the loop is limited to 1 (forward direction) and -1 (backward). To iterate backwards

over a range, use the reverse keyword:

"

"

```
with Ada.Text_IO; use Ada.Text_IO;


procedure Greet_5a_Reverse is
begin
    for I in reverse 1 .. 5 loop
            Put_Line ("Hello, World!" & Integer'Image (I));
    end loop;
end Greet_5a_Reverse;
```

"

"

Executing this procedure yields the following output:

Hello, World! 5

Hello, World! 4

Hello, World! 3

Hello, World! 2

Hello, World! 1

"

"

The bounds of a for loop may be computed at run-time; they are evaluated once, before the loop body is executed. If the value of the upper bound is less than the value of the lower bound, then the loop is not executed at all. This is the case also for reverse loops. Thus no output is produced in the following example:

"

"

```
with Ada.Text_IO; use Ada.Text_IO;


procedure Greet_No_Op is
begin
    for I in reverse 5 .. 1 loop
            Put_Line ("Hello, World!" & Integer'Image (I));
    end loop;
end Greet_No_Op;
```

"

"The simplest loop in Ada is the bare loop, which forms the foundation of the other kinds of Ada loops."

"

```
with Ada.Text_IO; use Ada.Text_IO;


procedure Greet_5b is
    I : Integer := 1; -- Variable declaration
```

```
   -- ^ Type
   --              ^ Initial value
begin
   loop
         Put_Line ("Hello, World!" & Integer'Image (I));
         exit when I = 5; -- Exit statement
         --              ^ Boolean condition
         -- Assignment
         I := I + 1; -- There is no I++ short form to increment a variable
   end loop;
end Greet_5b;
"
"
```

This example has the same effect as Greet_5a shown earlier.

It illustrates several concepts:

We have declared a variable named I between the is and the begin. This constitutes a declarative region. Ada clearly separates the declarative region from the statement part of a subprogram. A declaration can appear in a declarative region but is not allowed as a statement.

The bare loop statement is introduced by the keyword loop on its own and, like every kind of loop statement, is terminated by the combination of keywords end loop. On its own, it is an infinite loop. You can break out of it with an exit statement.

The syntax for assignment is ":=", and the one for equality is "=". There is no way to confuse them, because as previously noted, in Ada, statements and expressions are distinct, and expressions are not valid statements.

"

So ":=" in Ada is the equivalent to "=" in C, and "=" in Ada is equivalent to "==" in C.

"The last kind of loop in Ada is the while loop."

"

```
with Ada.Text_IO; use Ada.Text_IO;


procedure Greet_5c is
```

```
    I : Integer := 1;
begin
    -- Condition must be a Boolean value (no Integers).
    -- Operator "<=" returns a Boolean
    while I <= 5 loop
            Put_Line ("Hello, World!" & Integer'Image (I));
            I := I + 1;
    end loop;
end Greet_5c;
```

"

"The condition is evaluated before each iteration. If the result is false, then the loop is terminated.

This program has the same effect as the previous examples.

Note that Ada has different semantics than C-based languages with respect to the condition in a while loop. In Ada the condition has to be a Boolean value or the compiler will reject the program; the condition is not an integer that is treated as either True or False depending on whether it is non-zero or zero."

### 6.2.4 The case statement

"Ada's case statement is similar to the C and C++ switch statement, but with some important differences.

Here's an example, a variation of a program that was shown earlier with an if statement:"

"

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Check_Direction is
    N : Integer;
begin
    loop
```

```
            Put ("Enter an integer value: "); --
            Get (N); -- Reads an Integer
            Put (N); -- Puts an Integer
            case N is
                  when 0 | 360 =>
                        Put_Line (" is due east");
                  when 1 .. 89 =>
                        Put_Line (" is in the northeast
                  when 90 =>
                        Put_Line (" is due north");
                  when 91 .. 179 =>
                        Put_Line (" is in the northwest
                  when 180 =>
                        Put_Line (" is due west");
                  when 181 .. 269 =>
                        Put_Line (" is in the southwest
                  when 270 =>
                        Put_Line (" is due south");
                  when 271 .. 359 =>
                        Put_Line (" is in the southeast
                  when others =>
                        Put_Line (" Au revoir");
                        exit;
            end case;
      end loop;
end Check_Direction;
"
```

"This program repeatedly prompts for an integer value and then, if the value is in the range 0..360, displays the associated quadrant or axis. If the value is an Integer outside this range, the loop (and the program) terminate after outputting a farewell message.

The effect of the case statement is similar to the if statement in an earlier example, but the case statement can be more efficient because it does not involve multiple range tests.

Notable points about Ada's case statement:

The case expression (here the variable N) must be of a discrete type, i.e. either an integer type or an enumeration type.

Every possible value for the case expression needs to be covered by a unique branch of the case statement. This will be checked at compile time.

A branch can specify a single value, such as 0; a range of values, such as 1 .. 89; or any combination of the two (separated by a |).

As a special case, an optional final branch can specify others, which covers all values not included in the earlier branches.

Execution consists of the evaluation of the case expression and then a transfer of control to the statement sequence in the unique branch that covers that value.

When execution of the statements in the selected branch has completed, control resumes after the end case. Unlike C, execution does not fall through to the next branch. So Ada doesn't need (and doesn't have) a "break" statement."

### 6.2.5 Nested procedures, expressions verses statements

"Let's look at an example of a nested procedure:"

"

```
with Ada.Text_IO; use Ada.Text_IO;


procedure Main is
    procedure Nested is
    begin
            Put_Line ("Hello World");
    end Nested;
begin
    Nested;
    -- Call to Nested
end Main;
```

"

"A declaration cannot appear as a statement. If you need to declare a local variable amidst the statements, you can introduce a new declarative region with a block statement:"

"

```
with Ada.Text_IO; use Ada.Text_IO;


procedure Greet is
begin
    loop
            Put_Line ("Please enter your name: ");
            declare
                    Name : String := Get_Line;
                    --                  ^ Call to the Get_Line function
            begin
                    exit when Name = "";
                    Put_Line ("Hi " & Name & "!");
            end;
            -- Name is undefined here
    end loop;
    Put_Line ("Bye!");
end Greet;
```

"

"The Get_Line function allows you to receive input from the user, and get the result as a string. It is more or less equivalent to the scanf C function.

It returns a String, which, is an Unconstrained array type. For now we simply note that, if you wish to declare a String variable and do not know its size in advance, then you need to initialize the variable during its declaration."

"Ada 2012 introduced an expression analog for conditional statements (if and case).

Here's an alternative version of an example we saw earlier; the if statement has been replaced by an if expression:"

"

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;


procedure Check_Positive is
   N : Integer;
begin
   Put ("Enter an integer value: "); -- Put a String
   Get (N); -- Reads in an integer value
   Put (N); -- Put an Integer
   declare
         S : String :=
               (if N > 0 then " is a positive number"
               else " is not a positive number");
   begin
         Put_Line (S);
   end;
end Check_Positive;
"
```

"The if expression evaluates to one of the two Strings depending on N, and assigns that value to the local variable S.

Ada's if expressions are similar to if statements. However, there are a few differences that stem from the fact that it is an expression:

All branches' expressions must be of the same type

It must be surrounded by parentheses if the surrounding expression does not already contain them

An else branch is mandatory unless the expression following then has a Boolean value. In that case an else branch is optional and, if not present, defaults to "else True"."

"Analogous to if expressions, Ada also has case expressions. They work just as you would expect."

"

```
with Ada.Text_IO; use Ada.Text_IO;
```

```
procedure Main is
begin
    for I in 1 .. 10 loop
            Put_Line (case I is
                    when 1 | 3 | 5 | 7 | 9 => "Odd",
                    when 2 | 4 | 6 | 8 | 10 => "Even");
    end loop;
end Main;
"
```

"This program produces 10 lines of output, alternating between ”Odd” and ”Even”."

"The syntax differs from case statements, with branches separated by commas" (",").

### 6.2.6 Type declarations, attributes, overflow

"A nice feature of Ada is that you can define your own integer types, based on the requirements of your program (i.e., the range of values that makes sense). In fact, the definitional mechanism that Ada provides forms the semantic basis for the predefined integer types. There is no ”magical” built-in type in that regard, which is unlike most languages."

```
"
with Ada.Text_IO; use Ada.Text_IO;


procedure Integer_Type_Example is
    -- Declare a signed integer type, and give the bounds
    type My_Int is range -1 .. 20;
    --                          ^ High bound
    --                    ^ Low bound
    -- Like variables, type declarations can only appear in
    -- declarative regions
begin
    for I in My_Int loop
```

```
            Put_Line (My_Int'Image (I));
            --                  ^ 'Image attribute, converts a value to a String
        end loop;
    end Integer_Type_Example;
    "
```

"This example illustrates the declaration of a signed integer type, and several things we can do with them. Every type declaration in Ada starts with the type keyword."

"After the type, we can see a range that looks a lot like the ranges that we use in for loops, that defines the low and high bound of the type. Every integer in the inclusive range of the bounds is a valid value for the type."

"In Ada, an integer type is not specified in terms of its machine representation, but rather by its range. The compiler will then choose the most appropriate representation."

"Another point to note in the above example is the My_Int'Image (I) expression. The "Name'Attribute (optional params)" notation is used for what is called an attribute in Ada.

An attribute is a built-in operation on a type, a value, or some other program entity. It is accessed by using a "'" symbol (the ASCII apostrophe).

Ada has several types available as ”built-ins”; Integer is one of them. Here is how Integer might be defined for a typical processor:

"type Integer is range -(2 ** 31) .. +(2 ** 31 - 1);"

"**" is the exponent operator, which means that the first valid value for Integer is −2^31 , and the last valid value is 2^31 − 1.

Ada does not mandate the range of the built-in type Integer."

"Unlike some other languages, Ada requires that operations on integers should be checked for overflow."

"

```
procedure Main is
    A : Integer := Integer'Last;
    B : Integer;
begin
    B := A + 5;
    -- This operation will overflow, eg. it will
    -- raise an exception at run time.
```

```
end Main;
"
```

"There are two types of overflow checks:

Machine-level overflow, when the result of an operation exceeds the maximum value (or is less than the minimum value) that can be represented in the storage reserved for an object of the type, and

Type-level overflow, when the result of an operation is outside the range defined for the type.

Mainly for efficiency reasons, while machine level overflow always results in an exception, type level overflows will only be checked at specific boundaries, like assignment:"

```
"
with Ada.Text_IO; use Ada.Text_IO;


procedure Main is
    type My_Int is range 1 .. 20;
    A : My_Int := 12;
    B : My_Int := 15;
    M : My_Int := (A + B) / 2;
    -- No overflow here, overflow checks are done at
    -- specific boundaries.
begin
    for I in 1 .. M loop
            Put_Line ("Hello, World!");
    end loop;
    -- Loop body executed 13 times
end Main;
"
```

"Type level overflow will only be checked at specific points in the execution. The result, as we see above, is that you might have an operation that overflows in an intermediate computation, but no exception will be raised because the final result does not overflow."

"Ada also features unsigned Integer types. They’re called modular types in Ada parlance. The reason for this designation is due to their behavior in case of overflow: They simply ”wrap around”, as if a modulo operation was applied.

For machine sized modular types, for example a modulus of 2**32, this mimics the most common implementation behavior of unsigned types. However, an advantage of Ada is that the modulus is more general:"

"

```
with Ada.Text_IO; use Ada.Text_IO;


procedure Main is
    type Mod_Int is mod 2 ** 5;
    --              ^ Range is 0 .. 31
    A : Mod_Int := 20;
    B : Mod_Int := 15;
    M : Mod_Int := A + B;
    -- No overflow here, M = (20 + 15) mod 32 = 3
begin
    for I in 1 .. M loop
            Put_Line ("Hello, World!");
    end loop;
end Main;
```

"

"The modulus does not need to be a power of 2."

"Enumeration types are another nicety of Ada’s type system. Unlike C’s enums, they are not integers,

and each new enumeration type is incompatible with other enumeration types. Enumeration types are part of the bigger family of discrete types, which makes them usable in certain situations [...] one context that we have already seen is a case statement."

"

"

```
with Ada.Text_IO; use Ada.Text_IO;
```

```
procedure Enumeration_Example is
    type Days is (Monday, Tuesday, Wednesday,
            Thursday, Friday, Saturday, Sunday);
    -- An enumeration type
begin
    for I in Days loop
            case I is
                    when Saturday .. Sunday =>
                            Put_Line ("Week end!");
                    when Monday .. Friday =>
                            Put_Line ("Hello on " & Days'Image (I));
                            -- 'Image attribute, works on enums too
            end case;
    end loop;
end Enumeration_Example;
"
```

"Enumeration types are powerful enough that, unlike in most languages, they’re used to define the

standard Boolean type:

"type Boolean is (False, True);"

As mentioned previously, every ”built-in” type in Ada is defined with facilities generally available to the user."

"Like most languages, Ada supports floating-point types. The most commonly used floating-point type is Float."

"The Ada language does not specify the precision (number of decimal digits in the mantissa) for Float; on a typical 32-bit machine the precision will be 6.

All common operations that could be expected for floating-point types are available, including absolute value and exponentiation. For example:"

```
"
with Ada.Text_IO; use Ada.Text_IO;


procedure Floating_Point_Operations is
```

```
    A : Float := 2.5;
begin
    A := abs (A - 4.5);
    Put_Line ("The value of A is " & Float'Image (A));
    A := A ** 2 + 1.0;
    Put_Line ("The value of A is " & Float'Image (A));
end Floating_Point_Operations;
"
```

"The value of A is 2.0 after the first operation and 5.0 after the second operation.

In addition to Float, an Ada implementation may offer data types with higher precision such as Long_Float and Long_Long_Float. Like Float, the standard does not indicate the exact precision of these types: it only guarantees that the type Long_Float, for example, has at least the precision of Float. In order to guarantee that a certain precision requirement is met, we can define custom floating-point types."

"Ada allows the user to specify the precision for a floating-point type, expressed in terms of decimal digits. Operations on these custom types will then have at least the specified precision. The syntax for a simple floating-point type declaration is:

"type T is digits <number_of_decimal_digits>;"

The compiler will choose a floating-point representation that supports the required precision. For example:"

```
"
with Ada.Text_IO; use Ada.Text_IO;


procedure Custom_Floating_Types is
    type T3 is digits 3;
    type T15 is digits 15;
    type T18 is digits 18;
begin
    Put_Line ("T3 requires " & Integer'Image (T3'Size) & " bits");
    Put_Line ("T15 requires " & Integer'Image (T15'Size) & " bits");
    Put_Line ("T18 requires " & Integer'Image (T18'Size) & " bits");
end Custom_Floating_Types;
```

"

"In this example, the attribute 'Size is used to retrieve the number of bits used for the specified data type. As we can see by running this example, the compiler allocates 32 bits for T3, 64 bits for T15 and 128 bits for T18. This includes both the mantissa and the exponent."

"The number of digits specified in the data type is also used in the format when displaying floating-point variables. For example:"

"

```
with Ada.Text_IO; use Ada.Text_IO;


procedure Display_Custom_Floating_Types is
    type T3 is digits 3;
    type T18 is digits 18;


    C1 : constant := 1.0e-4;
    A : T3 := 1.0 + C1;
    B : T18 := 1.0 + C1;
begin
    Put_Line ("The value of A is " & T3'Image (A));
    Put_Line ("The value of B is " & T18'Image (B));
end Display_Custom_Floating_Types;
```

"

"As expected, the application will display the variables according to specified precision (1.00E+00 and 1.00010000000000000E+00)."

"In addition to the precision, a range can also be specified for a floating-point type. The syntax is similar to the one used for integer data types — using the "range" keyword."

"The application is responsible for ensuring that variables of this type stay within this range; otherwise an exception is raised. In this example, the exception Constraint_Error is raised when assigning 2.0 to the variable A:"

"

```
with Ada.Text_IO; use Ada.Text_IO;
```

```
procedure Floating_Point_Range_Exception is
   type T_Norm is new Float range -1.0 .. 1.0;
   A : T_Norm;
begin
   A := 2.0;
   Put_Line ("The value of A is " & T_Norm'Image (A));
end Floating_Point_Range_Exception;
"
```

"Ada is strongly typed. As a result, different types of the same family are incompatible with each other; a value of one type cannot be assigned to a variable from the other type. For example:"

```
"
with Ada.Text_IO; use Ada.Text_IO;


procedure Illegal_Example is
   -- Declare two different floating point types
   type Meters is new Float;
   type Miles is new Float;
   Dist_Imperial : Miles;
   -- Declare a constant
   Dist_Metric : constant Meters := 1000.0;
begin
   -- Not correct: types mismatch
   Dist_Imperial := Dist_Metric * 621.371e-6;
   Put_Line (Miles'Image (Dist_Imperial));
end Illegal_Example;
"
```

"A consequence of these rules is that, in the general case, a ”mixed mode” expression like 2 * 3.0 will trigger a compilation error. In a language like C or Python, such expressions are made valid by implicit conversions. In Ada, such conversions must be made explicit:"

```
"
```

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Conv is
    type Meters is new Float;
    type Miles is new Float;
    Dist_Imperial : Miles;
    Dist_Metric : constant Meters := 1000.0;
begin
    Dist_Imperial := Miles (Dist_Metric) * 621.371e-6;
    --                   ^ Type conversion, from Meters to Miles
    -- Now the code is correct
    Put_Line (Miles'Image (Dist_Imperial));
end Conv;
"
```

"Of course, we probably do not want to write the conversion code every time we convert from meters to miles. The idiomatic Ada way in that case would be to introduce conversion functions along with the types."

```
"
with Ada.Text_IO; use Ada.Text_IO;

procedure Conv is
    type Meters is new Float;
    type Miles is new Float;

    -- Function declaration, like procedure but returns a value.
    function To_Miles (M : Meters) return Miles is
        --                                ^ Return type
    begin
        return Miles (M) * 621.371e-6;
    end To_Miles;
```

```
    Dist_Imperial : Miles;
    Dist_Metric : constant Meters := 1000.0;
begin
    Dist_Imperial := To_Miles (Dist_Metric);
    Put_Line (Miles'Image (Dist_Imperial));
end Conv;
```

"The Ada compiler will always reject code that mixes floating-point and integer variables without explicit conversion. The following Ada code will not compile:"

```
procedure Main is
    A : Integer := 3;
    B : Integer := 2;
    F : Float;
begin
    F := A / B;
end Main;
```

"The offending line must be changed to "F := Float (A) / Float (B);" in order to be accepted by the compiler."

"In Ada you can create new types based on existing ones. This is very useful: you get a type that has the same properties as some existing type but is treated as a distinct type in the interest of strong typing."

```
procedure Main is
    -- ID card number type, incompatible with Integer.
    type Social_Security_Number
    is new Integer range 0 .. 999_99_9999;
    --                  ^ Since a SSN has 9 digits max, and cannot be negative, we enforce a validity constraint.
    SSN : Social_Security_Number := 555_55_5555;
```

```
    --                                  ^ You can put underscores as formatting in
any number.
    I : Integer;
    Invalid : Social_Security_Number := -1;
    --                                  ^ This will cause a runtime error
    -- (and a compile time warning with GNAT)
begin
    I := SSN;          -- Illegal, they have different types
    SSN := I;          -- Likewise illegal
    I := Integer (SSN);        -- OK with explicit conversion
    SSN := Social_Security_Number (I);    -- Likewise OK
end Main;
"
```

"The type Social_Security is said to be a derived type; its parent type is Integer."

### 6.2.7 Subtypes, aliases (type synonyms)

"types may be used in Ada to enforce constraints on the valid range of values. However, we sometimes want to enforce constraints on some values while staying within a single type. This is where subtypes come into play. A subtype does not introduce a new type."

"Several subtypes are predefined in the standard package in Ada, and are automatically available to you:"

```
"
subtype Natural is Integer range 0 .. Integer'Last;
subtype Positive is Integer range 1 .. Integer'Last;
"
```

"While subtypes of a type are statically compatible with each other, constraints are enforced at run time: if you violate a subtype constraint, an exception will be raised."

```
"
with Ada.Text_IO; use Ada.Text_IO;


procedure Greet is
```

```
    type Days is (Monday, Tuesday, Wednesday, Thursday,
                  Friday, Saturday, Sunday);
    subtype Weekend_Days is Days range Saturday .. Sunday;
    Day : Days := Saturday;
    Weekend : Weekend_Days;
begin
    Weekend := Day;
    --      ^ Correct: Same type, subtype constraints are respected
    Weekend := Monday;
    --      ^ Wrong value for the subtype
    -- Compiles, but exception at runtime
end Greet;
"
```

"We could also create type aliases, which generate alternative names — aliases — for known types. Note that type aliases are sometimes called type synonyms.

We achieve this in Ada by using subtypes without new constraints. In this case, however, we don’t get all of the benefits of Ada’s strong type checking. Let’s rewrite an example using type aliases:"

```
"
with Ada.Text_IO; use Ada.Text_IO;


procedure Undetected_Imperial_Metric_Error is
    -- Declare two type aliases
    subtype Meters is Float;
    subtype Miles is Float;
    Dist_Imperial : Miles;
    -- Declare a constant
    Dist_Metric : constant Meters := 100.0;
begin
    -- No conversion to Miles type required:
    Dist_Imperial := (Dist_Metric * 1609.0) / 1000.0;
```

```
    -- Not correct, but undetected:
    Dist_Imperial := Dist_Metric;
    Put_Line (Miles'Image (Dist_Imperial));
end Undetected_Imperial_Metric_Error;
```

"

"In the example above, the fact that both Meters and Miles are subtypes of Float allows us to mix variables of both types without type conversion."

"Subtypes in Ada correspond to type aliases if, and only if, they don’t have new constraints. Thus, if we add a new constraint to a subtype declaration, we don’t have a type alias anymore. For example, the following declaration can’t be considered a type alias of Float:"

"

```
subtype Meters is Float range 0.0 .. 1_000_000.0;
```

"

### 6.2.8 Subprograms, parameters and default values, modes

"Procedures are one kind of subprogram.

There are two kinds of subprograms in Ada, functions and procedures. The distinction between the two is that a function returns a value, and a procedure does not."

"This example shows the definition of a function:"

"

```
-- We define the Increment function
function Increment (I : Integer) return Integer is
begin
    return I + 1;
end Increment;
```

"

"Subprograms in Ada can, of course, have parameters. One syntactically important note is that a subprogram which has no parameters does not have a parameter section at all, for example:"

"

```
procedure Proc;
function Func return Integer;
"
```

"In this example, we see that parameters can have default values. When calling the subprogram, you can then omit parameters if they have a default value. Unlike C/C++, a call to a subprogram without parameters does not include parentheses."

```
"
function Increment_By
   (I : Integer := 0;
   Incr : Integer := 1) return Integer is
begin
   return I + Incr;
end Increment_By;
"
"We can then call our subprogram this way:"
"
with Ada.Text_IO; use Ada.Text_IO;
with Increment_By;

procedure Show_Increment is
   A, B, C : Integer;
begin
   C := Increment_By;
   --  ^ Parameterless call, value of I is 0 and Incr is 1

   Put_Line ("Using defaults for Increment_By is "
             & Integer'Image (C));

   A := 10;
   B := 3;
   C := Increment_By (A, B);
```

```
        --                      ^ Regular parameter passing

        Put_Line ("Increment of " & Integer'Image (A)
                & " with " & Integer'Image (B)
                & " is " & Integer'Image (C));

        A := 20;
        B := 5;
        C := Increment_By (I => A,
                Incr => B);
        --      ^ Named parameter passing

        Put_Line ("Increment of " & Integer'Image (A)
                & " with " & Integer'Image (B)
                & " is " & Integer'Image (C));
    end Show_Increment;
    "
```

"Ada allows you to name the parameters when you pass them, whether they have a default or not.

There are some rules:

Positional parameters come first.

A positional parameter cannot follow a named parameter."

"As briefly mentioned earlier, Ada allows you to declare one subprogram inside of another."

"For the previous example, we can move the duplicated code (call to Put_Line) to a separate procedure. This is a shortened version with the nested Display_Result procedure."

```
"
with Ada.Text_IO; use Ada.Text_IO;
with Increment_By;
```

```
procedure Show_Increment is
   A, B, C : Integer;

   procedure Display_Result is
   begin
         Put_Line ("Increment of " & Integer'Image (A)
               & " with " & Integer'Image (B)
               & " is " & Integer'Image (C));
   end Display_Result;
begin
   A := 10;
   B := 3;
   C := Increment_By (A, B);
   Display_Result;
end Show_Increment;
"
```

"An important feature of function calls in Ada is that the return value at a call cannot be ignored; that is, a function call cannot be used as a statement.

If you want to call a function and do not need its result, you will still need to explicitly store it in a local variable."

```
"
function Quadruple (I : Integer) return Integer is
   function Double (I : Integer) return Integer is
   begin
         return I * 2;
   end Double;

   Res : Integer := Double (Double (I));
   --                          ^ Calling the double function
begin
   Double (I);
```

```
    -- ERROR: cannot use call to function "Double" as a statement


    return Res;
end Quadruple;
"
```

"A subprogram parameter can be specified with a mode, which is one of the following:

in -- Parameter can only be read, not written

out -- Parameter can be written to, then read

in out -- Parameter can be both read and written

The default mode for parameters is in; so far, most of the examples have been using in parameters."

"The first mode for parameters is the one we have been implicitly using so far. Parameters passed using this mode cannot be modified, so that the following program will cause an error:"

```
"
procedure Swap (A, B : Integer) is
    Tmp : Integer;
begin
    Tmp := A;


    -- Error: assignment to "in" mode parameter not allowed
    A := B;
    -- Error: assignment to "in" mode parameter not allowed
    B := Tmp;
end Swap;
"
```

"To correct our code above, we can use an ”in out” parameter."

```
"
with Ada.Text_IO; use Ada.Text_IO;
```

```
procedure In_Out_Params is
    procedure Swap (A, B : in out Integer) is
            Tmp : Integer;
    begin
            Tmp := A;
            A := B;
            B := Tmp;
    end Swap;

    A : Integer := 12;
    B : Integer := 44;
begin
    Swap (A, B);
    Put_Line (Integer'Image (A)); -- Prints 44
end In_Out_Params;
"
```

"An in out parameter will allow read and write access to the object passed as parameter, so in the example above, we can see that A is modified after the call to Swap."

"The ”out” mode applies when the subprogram needs to write to a parameter that might be uninitialized at the point of call. Reading the value of an out parameter is permitted, but it should only be done after the subprogram has assigned a value to the parameter. Out parameters behave a bit like return values for functions. When the subprogram returns, the actual parameter (a variable) will have the value of the out parameter at the point of return."

"While reading an out variable before writing to it should, ideally, trigger an error, imposing that as a rule would cause either inefficient run-time checks or complex compile-time rules. So from the user’s perspective an out parameter acts like an uninitialized variable when the subprogram is invoked."

"GNAT will detect simple cases of incorrect use of out parameters. For example, the compiler will emit a warning for the following program:"

```
"
procedure Outp is
```

```
    procedure Foo (A : out Integer) is
        B : Integer := A; -- Warning on reference to uninitialized A
    begin
        A := B;
    end Foo;
begin
    null;
end Outp;
"
```

"A subprogram can be declared without being fully defined, This is possible in general, and can be useful if you need subprograms to be mutually recursive, as in the example below:"

```
"
procedure Mutually_Recursive_Subprograms is
    procedure Compute_A (V : Natural);
    -- Forward declaration of Compute_A

    procedure Compute_B (V : Natural) is
    begin
        if V > 5 then
            Compute_A (V - 1);
            -- Call to Compute_A
        end if;
    end Compute_B;

    procedure Compute_A (V : Natural) is
    begin
        if V > 2 then
            Compute_B (V - 1);
            -- Call to Compute_B
        end if;
```

```
    end Compute_A;
begin
    Compute_A (15);
end Mutually_Recursive_Subprograms;
"
```

### 6.2.9 Subprogram renaming

"Subprograms can be renamed by using the renames keyword and declaring a new name for a subprogram:"

```
"
procedure New_Proc renames Original_Proc;
"
```

"This can be useful, for example, to improve the readability of your application when you're using code from external sources that cannot be changed in your system. Let's look at an example:"

```
"
with Ada.Text_IO; use Ada.Text_IO;

procedure A_Procedure_With_Very_Long_Name_That_Cannot_Be_Changed
(A_Message : String) is
begin
    Put_Line (A_Message);
end A_Procedure_With_Very_Long_Name_That_Cannot_Be_Changed;
"
```

"As the wording in the name of procedure above implies, we cannot change its name. We can, however, rename it to something like Show in our test application and use this shorter name. Note that we also have to declare all parameters of the original subprogram — we may rename them, too, in the declaration. For example:"

```
"
with A_Procedure_With_Very_Long_Name_That_Cannot_Be_Changed;

procedure Show_Renaming is
```

```
        procedure Show (S : String) renames
A_Procedure_With_Very_Long_Name_That_Cannot_Be_Changed;
    begin
        Show ("Hello World!");
    end Show_Renaming;
    "
    "Note that the original name
(A_Procedure_With_Very_Long_Name_That_Cannot_Be_Changed) is still visible after
the declaration of the Show procedure. We may also rename subprograms from the
standard library. For example, we may rename Integer'Image to Img:"
    "
    with Ada.Text_IO; use Ada.Text_IO;


    procedure Show_Image_Renaming is
        function Img (I : Integer) return String renames Integer'Image;
    begin
        Put_Line (Img (2));
        Put_Line (Img (3));
    end Show_Image_Renaming;
    "
    "Renaming also allows us to introduce default expressions that were not available in
the original declaration. For example, we may specify "Hello World!" as the default for
the String parameter of the Show procedure:"
    "
    with A_Procedure_With_Very_Long_Name_That_Cannot_Be_Changed;


    procedure Show_Renaming_Defaults is
        procedure Show (S : String := "Hello World!") renames
A_Procedure_With_Very_Long_Name_That_Cannot_Be_Changed;
    begin
        Show;
    end Show_Renaming_Defaults;
```

"

### 6.2.10 Records, aggregates

"Records allow composing a value out of instances of other types. Each of those instances will be given a name. The pair consisting of a name and an instance of a specific type is called a field, or a component."

"Fields look a lot like variable declarations, except that they are inside of a record definition. And as with variable declarations, you can specify additional constraints when supplying the subtype of the field."

"

```
type Date is record
    Day : Integer range 1 .. 31;
    Month : Months := January;
    -- This component has a default value
    Year : Integer range 1 .. 3000 := 2012;
    --                                  ^ Default value
end record;
```

"

"Record components can have default values. When a variable having the record type is declared, a field with a default initialization will be automatically set to this value. The value can be any expression of the component type, and may be run-time computable."

"Records have a convenient notation for expressing values. This notation is called aggregate notation, and the literals are called aggregates. They can be used in a variety of contexts that we will see throughout the course, one of which is to initialize records. An aggregate is a list of values separated by commas and enclosed in parentheses. It is allowed in any context where a value of the record is expected."

"

```
Ada_Birthday : Date := (10, December, 1815);
Leap_Day_2020 : Date := (Day => 29, Month => February, Year => 2020);
--                                  ^ By name
```

"

"To access components of a record instance, you use an operation that is called component selection. This is achieved by using the dot notation. For example, if we declare a variable Some_Day of the Date record type mentioned above, we can access the Year component by writing "Some_Day.Year". Let's look at an example:"

"

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Record_Selection is
   type Months is
         (January, February, March, April, May, June, July,
         August, September, October, November, December);
   type Date is record
         Day : Integer range 1 .. 31;
         Month : Months;
         Year : Integer range 1 .. 3000 := 2032;
   end record;

   procedure Display_Date (D : Date) is
   begin
         Put_Line ("Day:" & Integer'Image (D.Day)
               & ", Month: " & Months'Image (D.Month)
               & ", Year:" & Integer'Image (D.Year));
   end Display_Date;

   Some_Day : Date := (1, January, 2000);
begin
   Display_Date (Some_Day);

   Put_Line ("Changing year...");
   Some_Day.Year := 2001;
```

```
    Display_Date (Some_Day);
end Record_Selection;
"
```

"When we use D.Year in the call to Put_Line, we're retrieving the information stored in that component. When we write Some_Day.Year := 2001, we're overwriting the information that was previously stored in the Year component of Some_Day."

"We can rename record components as well. Instead of writing the full component selection using the dot notation, we can declare an alias that allows us to access the same component. We can rename record components by using the renames keyword in a variable declaration. For example:"

```
"
Some_Day : Date
Y : Integer renames Some_Day.Year;
"
```

"Here, Y is an alias, so that every time we using Y, we are really using the Year component of Some_Day."

### 6.2.11 Arrays: indexing, size/range, the <> wildcard

"Arrays in Ada are used to define contiguous collections of elements that can be selected by indexing. Here's a simple example:"

```
"
with Ada.Text_IO; use Ada.Text_IO;


procedure Greet is
    type My_Int is range 0 .. 1000;
    type Index is range 1 .. 5;
    type My_Int_Array is array (Index) of My_Int;
    --                                    ^ Type of elements
    --                              ^ Bounds of the array


    Arr : My_Int_Array := (2, 3, 5, 7, 11);
    --                          ^ Array literal, called aggregate in Ada
```

```
begin
    for I in Index loop
            Put (My_Int'Image (Arr (I)));
            --                          ^ Take the Ith element
    end loop;
    New_Line;
end Greet;
"
```

"The first point to note is that we specify the index type for the array, rather than its size. Here we declared an integer type named Index ranging from 1 to 5, so each array instance will have 5 elements, with the initial element at index 1 and the last element at index 5. Although this example used an integer type for the index, Ada is more general: any discrete type is permitted to index an array, including Enum types."

"Another point to note is that querying an element of the array at a given index uses the same syntax as for function calls: that is, the array object followed by the index in parentheses. Thus when you see an expression such as A (B), whether it is a function call or an array subscript depends on what A refers to. Notice how we initialize the array with the (2, 3, 5, 7, 11) expression. This is another kind of aggregate in Ada, and is in a sense a literal expression for an array, in the same way that 3 is a literal expression for an integer."

"New_Line outputs an end of line."

"Semantically, an array object in Ada is the entire data structure, and not simply a handle or pointer. Unlike C and C++, there is no implicit equivalence between an array and a pointer to its initial element."

```
"
with Ada.Text_IO; use Ada.Text_IO;


procedure Array_Bounds_Example is
    type My_Int is range 0 .. 1000;
    type Index is range 11 .. 15;
    --                      ^ Low bound can be any value
    type My_Int_Array is array (Index) of My_Int;
    Tab : My_Int_Array := (2, 3, 5, 7, 11);
begin
```

```
    for I in Index loop
        Put (My_Int'Image (Tab (I)));
    end loop;
    New_Line;
end Array_Bounds_Example;
"
```

"The bounds of an array can be any values. In the first example we constructed an array type whose first index is 1, but in the example above we declare an array type whose first index is 11. That's perfectly fine in Ada, and moreover since we use the index type as a range to iterate over the array indices, the code using the array does not need to change."

"Since you can use any discrete type to index an array, enumeration types are permitted."

```
"
with Ada.Text_IO; use Ada.Text_IO;

procedure Month_Example is
    type Month_Duration is range 1 .. 31;
    type Month is (Jan, Feb, Mar, Apr, May, Jun,
        Jul, Aug, Sep, Oct, Nov, Dec);
    type My_Int_Array is array (Month) of Month_Duration;
    --                              ^ Can use an enumeration type as the index

    Tab : constant My_Int_Array := (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31);
    --      ^ constant is like a variable but cannot be modified
    -- Maps months to number of days (ignoring leap years)

    Feb_Days : Month_Duration := Tab (Feb);
    -- Number of days in February
begin
    for M in Month loop
        Put_Line(Month'Image (M) & " has "
```

```
                & Month_Duration'Image (Tab (M))
        end loop;
    end Month_Example;
    "
    "In the example above, we are:
     Creating an array type mapping months to month durations in days.
     Creating an array, and instantiating it with an aggregate mapping months to their actual durations in days.
     Iterating over the array, printing out the months, and the number of days for each."
    "As is true in general in Ada, the indexing operation is strongly typed. If you use a value of the wrong type to index the array, you will get a compile-time error."
    "
    with Ada.Text_IO; use Ada.Text_IO;


    procedure Greet is
        type My_Int is range 0 .. 1000;
        type My_Index is range 1 .. 5;
        type Your_Index is range 1 .. 5;

        type My_Int_Array is array (My_Index) of My_Int;
        Tab : My_Int_Array := (2, 3, 5, 7, 11);
    begin
        for I in Your_Index loop
                Put (My_Int'Image (Tab (I)));
                --                  ^ Compile time error
        end loop;
        New_Line;
    end Greet;
    "
```

"Arrays in Ada are bounds checked. This means that if you try to access an element outside of the bounds of the array, you will get a run-time error instead of accessing random memory as in unsafe languages."

"

```
with Ada.Text_IO; use Ada.Text_IO;


procedure Greet is
    type My_Int is range 0 .. 1000;
    type Index is range 1 .. 5;


    type My_Int_Array is array (Index) of My_Int;
    Tab : My_Int_Array := (2, 3, 5, 7, 11);
begin
    for I in Index range 2 .. 6 loop
            Put (My_Int'Image (Tab (I)));
            --              ^ Will raise an exception when I = 6
    end loop;
    New_Line;
end Greet;
```

"

When the index type of the array isn't known, it is possible to use "'Range":

"

```
with Ada.Text_IO; use Ada.Text_IO;


procedure Range_Example is
    type My_Int is range 0 .. 1000;
    type My_Int_Array is array (1 .. 5) of My_Int;
    Tab : My_Int_Array := (2, 3, 5, 7, 11);
begin
    for I in Tab'Range loop
    --          ^ Gets the range of Tab
```

```
            Put (My_Int'Image (Tab (I)));
        end loop;
        New_Line;
    end Range_Example;
    "
    Or "'First" and "'Last" using the ".." notation:
    "
    with Ada.Text_IO; use Ada.Text_IO;

    procedure Array_Attributes_Example is
        type My_Int is range 0 .. 1000;
        type My_Int_Array is array (1 .. 5) of My_Int;
        Tab : My_Int_Array := (2, 3, 5, 7, 11);
    begin
        for I in Tab'First .. Tab'Last - 1 loop
        --                  ^ Iterate on every index except the last
            Put (My_Int'Image (Tab (I)));
        end loop;
        New_Line;
    end Array_Attributes_Example;
    "
```

"The 'Range, 'First and 'Last attributes in these examples could also have been applied to the array type name, and not just the array instances.

Although not illustrated in the above examples, another useful attribute for an array instance A is A'Length, which is the number of elements that A contains."

"Ada also allows you to declare array types whose bounds are not fixed: in that case, the bounds will need to be provided when creating instances of the type."

```
    "
    with Ada.Text_IO; use Ada.Text_IO;

    procedure Unconstrained_Array_Example is
```

```
type Days is (Monday, Tuesday, Wednesday,
        Thursday, Friday, Saturday, Sunday);
type Workload_Type is array (Days range <>) of Natural;
-- Indefinite array type
--                                  ^ Bounds are of type Days, but not known

Workload : constant Workload_Type (Monday .. Friday) := (Friday => 7, others =>
8);
--                                      ^ Specify the bounds when declaring
--                                                                  ^ Default
value
--                                                      ^ Specify element by
name of index
begin
   for I in Workload'Range loop
        Put_Line (Integer'Image (Workload (I)));
   end loop;
end Unconstrained_Array_Example;
"
```

"The fact that the bounds of the array are not known is indicated by the Days range <> syntax. Given a discrete type Discrete_Type, if we use Discrete_Type for the index in an array type then Discrete_Type serves as the type of the index and comprises the range of index values for each array instance. If we define the index as Discrete_Type range <> then Discrete_Type serves as the type of the index, but different array instances may have different bounds from this type. An array type that is defined with the Discrete_Type range <> syntax for its index is referred to as an unconstrained array type, and, as illustrated above, the bounds need to be provided when an instance is created.

The above example also shows other forms of the aggregate syntax. You can specify associations by name, by giving the value of the index on the left side of an arrow association. 1 => 2 thus means ”assign value 2 to the element at index 1 in my array”. others => 8 means ”assign value 8 to every element that wasn’t previously assigned in this aggregate”."

"The so-called ”box” notation (<>) is commonly used as a wildcard or placeholder in Ada. You will often see it when the meaning is ”what is expected here can be anything”."

"While unconstrained arrays in Ada might seem similar to variable length arrays in C, they are in reality much more powerful, because they're truly first-class values in the language. You can pass them as parameters to subprograms or return them from functions, and they implicitly contain their bounds as part of their value. This means that it is useless to pass the bounds or length of an array explictly along with the array, because they are accessible via the 'First, 'Last, 'Range and 'Length attributes explained earlier.

Although different instances of the same unconstrained array type can have different bounds, a specific instance has the same bounds throughout its lifetime."

"The String type in Ada is a simple array. Here is how the string type is defined in Ada:"

"

type String is array (Positive range <>) of Character;

"

"The only built-in feature Ada adds to make strings more ergonomic is custom literals, as we can see in the example below.

String literals are a syntactic sugar for aggregates, so that in the following example, A and B have the same value."

"

```
package String_Literals is
    -- These two declarations are equivalent
    A : String (1 .. 11) := "Hello World";
    B : String (1 .. 11) := ('H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd');
end String_Literals;
```

"

"You can omit the bounds when creating an instance of an unconstrained array type if you supply an initialization, since the bounds can be deduced from the initialization expression."

"

```
with Ada.Text_IO; use Ada.Text_IO;


procedure Greet is
    Message : constant String := "dlroW olleH";
    --                  ^ Bounds are automatically computed from initialization value
```

```
begin
    for I in reverse Message'Range loop
            Put (Message (I));
    end loop;
    New_Line;
end Greet;
"
```

"A String value is stack allocated, it is accessed efficiently, and its bounds are immutable."

"A very important point about arrays: bounds have to be known when instances are created. It is for example illegal to do the following."

```
"
declare
    A : String;
begin
    A := "World";
end;
"
```

"Also, while you of course can change the values of elements in an array, you cannot change the array's bounds (and therefore its size) after it has been initialized. So this is also illegal:"

```
"
declare
    A : String := "Hello";
begin
    A := "World"; -- OK: Same size
    A := "Hello World"; -- Not OK: Different size
end;
"
```

"Also, while you can expect a warning for this kind of error in very simple cases like this one, it is impossible for a compiler to know in the general case if you are assigning a value of the correct length, so this violation will generally result in a run-time error."

"It is important to know that arrays are not the only types whose instances might be of unknown size at compile-time.

Such objects are said to be of an indefinite subtype, which means that the subtype size is not known at compile time, but is dynamically computed (at run time)."

"

```
with Ada.Text_IO; use Ada.Text_IO;


procedure Indefinite_Subtypes is
   function Get_Number return Integer is
   begin
            return Integer'Value (Get_Line);
   end Get_Number;

   A : String := "Hello";
   -- Indefinite subtype

   B : String (1 .. 5) := "Hello";
   -- Definite subtype

   C : String (1 .. Get_Number);
   -- Indefinite subtype (Get_Number's value is computed at run-time)
begin
   null;
end Indefinite_Subtypes;
```

"

"The return type of a function can be any type; a function can return a value whose size is unknown at compile time. Likewise, the parameters can be of any type. For example, this is a function that returns an unconstrained String:"

"

```
with Ada.Text_IO; use Ada.Text_IO;
```

```
procedure Main is
    type Days is (Monday, Tuesday, Wednesday,
            Thursday, Friday, Saturday, Sunday);

    function Get_Day_Name (Day : Days := Monday) return String is
    begin
            return
                    (case Day is
                            when Monday => "Monday",
                            when Tuesday => "Tuesday",
                            when Wednesday => "Wednesday",
                            when Thursday => "Thursday",
                            when Friday => "Friday",
                            when Saturday => "Saturday",
                            when Sunday => "Sunday");
    end Get_Day_Name;
begin
    Put_Line ("First day is " & Get_Day_Name (Days'First));
end Main;
"
```

"While we can have array types whose size and bounds are determined at run time, the array's component type needs to be of a definite and constrained type.

Thus, if you need to declare, for example, an array of Strings, the String subtype used as component will need to have a fixed size."

```
"
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Days is
    type Days is (Monday, Tuesday, Wednesday,
            Thursday, Friday, Saturday, Sunday);
```

```
    subtype Day_Name is String (1 .. 2);
    -- Subtype of string with known size

    type Days_Name_Type is array (Days) of Day_Name;
    --                                     ^ Type of the index
    --                                           ^ Type of the element. Must be definite

    Names : constant Days_Name_Type := ("Mo", "Tu", "We", "Th", "Fr", "Sa", "Su");
    -- Initial value given by aggregate
begin
    for I in Names'Range loop
        Put_Line (Names (I));
    end loop;
end Show_Days;
"
"It is possible to take and use a slice of an array (a contiguous sequence of elements) as a name or a value."
"
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
    Buf : String := "Hello ...";
    Full_Name : String := "John Smith";
begin
    Buf (7 .. 9) := "Bob";
    -- Careful! This works because the string on the right side is the
    -- same length as the replaced slice!
    Put_Line (Buf);
    -- Prints "Hello Bob"
    Put_Line ("Hi " & Full_Name (1 .. 4)); -- Prints "Hi John"
end Main;
```

"

"As we can see above, you can use a slice on the left side of an assignment, to replace only part of an array.

A slice of an array is of the same type as the array, but has a different subtype, constrained by the bounds of the slice."

"Slices will only work on one dimensional arrays."

"An Ada aggregate is, in effect, a literal value for a composite type. It's a very powerful notation that helps you to avoid writing procedural code for the initialization of your data structures in many cases. A basic rule when writing aggregates is that every component of the array or record has to be specified, even components that have a default value."

"There are a few shortcuts that you can use to make the notation more convenient:

To specify the default value for a component, you can use the <> notation.

You can use the | symbol to give several components the same value.

You can use the others choice to refer to every component that has not yet been specified, provided all those fields have the same type.

You can use the range notation .. to refer to specify a contiguous sequence of indices in an array.

However, note that as soon as you used a named association, all subsequent components likewise need to be specified with names associations."

"

```
package Points is
    type Point is record
            X, Y : Integer := 0;
    end record;

    type Point_Array is array (Positive range <>) of Point;

    Origin : Point := (X | Y => <>); -- use the default values
    Origin_2 : Point := (others => <>); -- likewise use the defaults

    Points_1 : Point_Array := ((1, 2), (3, 4));
    Points_2 : Point_Array := (1 => (1, 2), 2 => (3, 4), 3 .. 20 => <>);
```

```
end Points;
"
```

### 6.2.12 Overloading

"It is possible in Ada to have functions that have the same name, but different types for their parameters."

"This is a common concept in programming languages, called overloading , or name overloading. One of the novel aspects of Ada's overloading facility is the ability to resolve overloading based on the return type of a function."

"However, sometimes an ambiguity makes it impossible to resolve which declaration of an overloaded name a given occurrence of the name refers to. This is where a qualified expression becomes useful."

```
"
package Pkg is
   type SSID is new Integer;

   function Convert (Self : SSID) return Integer;
   function Convert (Self : SSID) return String;
   function Convert (Self : Integer) return String;
end Pkg;
"
"
with Ada.Text_IO; use Ada.Text_IO;
with Pkg; use Pkg;

procedure Main is
   S : String := Convert (123_145_299);
   --                ^ Invalid, which convert should we call?
   S2 : String := Convert (SSID'(123_145_299));
   --                            ^ We specify that the type of the expression is SSID.
   -- We could also have declared a temporary
   I : SSID := 123_145_299;
```

```
    S3 : String := Convert (I);
begin
    Put_Line (S);
end Main;
"
```

### 6.2.13 Access types, discriminants

```
"Here is how you declare a simple pointer type, or access type, in Ada:"
"
package Dates is
    type Months is (January, February, March, April, May, June, July,
            August, September, October, November, December);

    type Date is record
            Day : Integer range 1 .. 31;
            Month : Months;
            Year : Integer;
    end record;
end Dates;
"
"
with Dates; use Dates;

package Access_Types is
    -- Declare an access type
    type Date_Acc is access Date;
    --                      ^ "Designated type"
    --                      ^ Date_Acc values point to Date objects

    D : Date_Acc := null;
```

```
    --                  ^ Literal for "access to nothing"
    --  ^ Access to date
end Access_Types;
"
```

"In line with Ada's strong typing philosophy, if you declare a second access type whose designated type is Date, the two access types will be incompatible with each other, and you will need an explicit type conversion to convert from one to the other:"

```
"
with Dates; use Dates;

package Access_Types is
    -- Declare an access type
    type Date_Acc is access Date;
    type Date_Acc_2 is access Date;

    D : Date_Acc := null;
    D2 : Date_Acc_2 := D;
    --              ^ Invalid! Different types
    D3 : Date_Acc_2 := Date_Acc_2 (D);
    --              ^ Valid with type conversion
end Access_Types;
"
```

"Once we have declared an access type, we need a way to give variables of the types a meaningful value! You can allocate a value of an access type with the new keyword in Ada."

```
"
with Dates; use Dates;

package Access_Types is
    type Date_Acc is access Date;
```

```
    D : Date_Acc := new Date;
    --                  ^ Allocate a new Date record
end Access_Types;
"
```

"If the type you want to allocate needs constraints, you can put them in the subtype indication, just as you would do in a variable declaration:"

```
"
with Dates; use Dates;

package Access_Types is
    type String_Acc is access String;
    --              ^ Access to unconstrained array type

    Msg : String_Acc;
    --    ^ Default value is null

    Buffer : String_Acc := new String (1 .. 10);
    --                                 ^ Constraint required
end Access_Types;
"
```

"Ada also allows you to initialize along with the allocation. This is done via the qualified expression syntax:"

```
"
with Dates; use Dates;

package Access_Types is
    type Date_Acc is access Date;
    type String_Acc is access String;

    D : Date_Acc := new Date'(30, November, 2011);
    Msg : String_Acc := new String'("Hello");
```

```
end Access_Types;
"
```

"Dereferencing a pointer uses the .all syntax in Ada, but is often not needed - in many cases, the access value will be implicitly dereferenced for you:"

```
"
with Dates; use Dates;


package Access_Types is
    type Date_Acc is access Date;


    D : Date_Acc := new Date'(30, November, 2011);
    Today : Date := D.all;
    --                  ^ Access value dereference
    J : Integer := D.Day;
    --                ^ Implicit dereference for record and array components
    -- Equivalent to D.all.day
end Access_Types;
"
```

"As you might know if you have used pointers in C or C++, we are still missing features that are considered fundamental to the use of pointers, such as:

Pointer arithmetic (being able to increment or decrement a pointer in order to point to the next or previous object)

Manual deallocation - what is called free or delete in C. This is a potentially unsafe operation. To keep within the realm of safe Ada, you need to never deallocate manually.

Those features exist in Ada, but are only available through specific standard library APIs.

The guideline in Ada is that most of the time you can avoid manual allocation, and you should."

"Discriminants can be used to obtain the functionality of what are sometimes called ”variant records”: records that can contain different sets of fields."

```
"
package Variant_Record is
```

```
    type Expr; -- Forward declaration of Expr
    type Expr_Access is access Expr; -- Access to a Expr
    type Expr_Kind_Type is (Bin_Op_Plus, Bin_Op_Minus, Num);
    -- A regular enumeration type
    type Expr (Kind : Expr_Kind_Type) is record
    --              ^ The discriminant is an enumeration value
            case Kind is
                    when Bin_Op_Plus | Bin_Op_Minus =>
                            Left, Right : Expr_Access;
                    when Num =>
                            Val : Integer;
            end case;
    -- Variant part. Only one, at the end of the record
    -- definition, but can be nested
    end record;
end Variant_Record;
"
```

"The fields that are in a when branch will be only available when the value of the discriminant is covered by the branch. In the example above, you will only be able to access the fields Left and Right when the Kind is Bin_Op_Plus or Bin_Op_Minus. If you try to access a field that is not valid for your record, a Constraint_Error will be raised."

```
"
with Variant_Record; use Variant_Record;


procedure Main is
    E : Expr := (Num, 12);
begin
    E.Left := new Expr'(Num, 15);
    -- Will compile but fail at runtime
end Main;
"
```

### 6.2.14 Encapsulation and visibility, generics

"One of the main principles of modular programming, as well as object oriented programming, is encapsulation.

Encapsulation, briefly, is the concept that the implementer of a piece of software will distinguish between the code's public interface and its private implementation. This is not only applicable to software libraries but wherever abstraction is used.

In Ada, the granularity of encapsulation is a bit different from most object-oriented languages, because privacy is generally specified at the package level."

"

```
package Encapsulate is
    procedure Hello;
private
    procedure Hello2;
    -- Not visible from external units
end Encapsulate;
```

"

"

```
with Encapsulate;


procedure Main is
begin
    Encapsulate.Hello;
    Encapsulate.Hello2;
    -- Invalid: Hello2 is not visible
end Main;
```

"

"Ada's limited type facility allows you to declare a type for which assignment and comparison operations are not automatically provided."

"

```
package Stacks is
    type Stack is limited private;
```

```
        -- Limited type. Cannot assign nor compare.

        procedure Push (S : in out Stack; Val : Integer);
        procedure Pop (S : in out Stack; Val : out Integer);
    private
        subtype Stack_Index is Natural range 1 .. 10;
        type Content_Type is array (Stack_Index) of Natural;

        type Stack is limited record
                Top : Stack_Index;
                Content : Content_Type;
        end record;
    end Stacks;
    "
    "
    with Stacks; use Stacks;

    procedure Main is
        S, S2 : Stack;
    begin
        S := S2;
        -- Illegal: S is limited.
    end Main;
    "
```

"Generics are used for metaprogramming in Ada. They are useful for abstract algorithms that share common properties with each other.

Either a subprogram or a package can be generic. A generic is declared by using the keyword "generic"."

"Formal types are abstractions of a specific type. For example, we may want to create an algorithm that works on any integer type, or even on any type at all, whether a numeric type or not. The following example declares a formal type T for the Set procedure."

"

generic

type T is private;

-- T is a formal type that indicates that any type can be used,

-- possibly a numeric type or possibly even a record type.

procedure Set (Dummy : T);

"

"

procedure Set (Dummy : T) is

begin

null;

end Set;

"

"The declaration of T as private indicates that you can map any type to it. But you can also restrict the declaration to allow only some types to be mapped to that formal type. Here are some examples:"

"

type T is private; -- Any type

type T is (<>); -- Any discrete type

type T is digits <>; -- Any floating-point type

"

"We don’t repeat the generic keyword for the body declaration of a generic subprogram or package. Instead, we start with the actual declaration and use the generic types and objects we declared. For example:"

"

generic

type T is private;

X : in out T;

procedure Set (E : T);

"

"

```
procedure Set (E : T) is
   -- Body definition: "generic" keyword is not used
begin
   X := E;
end Set;
"
```

"Generic subprograms or packages can’t be used directly. Instead, they need to be instantiated, which we do using the new keyword, as shown in the following example:"

```
"
with Ada.Text_IO; use Ada.Text_IO;
with Set;

procedure Show_Generic_Instantiation is
   Main : Integer := 0;
   Current : Integer;

   procedure Set_Main is new Set (T => Integer, X => Main);
   -- Here, we map the formal parameters to actual types and objects.
   --
   -- The same approach can be used to instantiate functions or
   -- packages, e.g.:
   -- function Get_Main is new ...
   -- package Integer_Queue is new ...
begin
   Current := 10;
   Set_Main (Current);
   Put_Line ("Value of Main is " & Integer'Image (Main));
end Show_Generic_Instantiation;
"
```

### 6.2.15 Exceptions

"Ada uses exceptions for error handling. Unlike many other languages, Ada speaks about raising, not throwing, an exception and handling, not catching, an exception."

"Ada exceptions are not types, but instead objects."

"Here’s how you declare an exception:"

"

```
package Exceptions is
    My_Except : exception;
    -- Like an object. *NOT* a type !
end Exceptions;
```

"

"Ada does not require that a subprogram declare every exception it can potentially raise."

"

```
with Exceptions; use Exceptions;

procedure Main is
begin
    raise My_Except;
    -- Execution of current control flow abandoned; an exception of kind
    -- "My_Except" will bubble up until it is caught.

    raise My_Except with "My exception message";
    -- Execution of current control flow abandoned; an exception of
    -- kind "My_Except" with associated string will bubble up until it is caught.
end Main;
```

"

"The neat thing in Ada is that you can add an exception handler to any statement block as follows:"

"

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Exceptions; use Ada.Exceptions;

procedure Open_File is
   File : File_Type;
begin
   -- Block (sequence of statements)
   begin
         Open (File, In_File, "input.txt");
   exception
         when E : Name_Error =>
         --      ^ Exception to be handled
                  Put ("Cannot open input file : ");
                  Put_Line (Exception_Message (E));
                  raise;
                  -- Reraise current occurence
   end;
end Open_File;
"
```

"In the example above, we're using the Exception_Message function from the Ada.Exceptions package. This function returns the message associated with the exception as a string.

You don't need to introduce a block just to handle an exception: you can add it to the statements block of your current subprogram:"

```
"
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Exceptions; use Ada.Exceptions;

procedure Open_File is
   File : File_Type;
begin
```

```
    Open (File, In_File, "input.txt");
    -- Exception block can be added to any block
exception
    when Name_Error =>
        Put ("Cannot open input file");
end Open_File;
"
```

"Exception handlers have an important restriction that you need to be careful about:

Exceptions raised in the declarative section are not caught by the handlers of that block. So for example, in the following code, the exception will not be caught."

```
"
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Exceptions; use Ada.Exceptions;


procedure Be_Careful is
    function Dangerous return Integer is
    begin
        raise Constraint_Error;
        return 42;
    end Dangerous;
begin
    declare
        A : Integer := Dangerous;
    begin
        Put_Line (Integer'Image (A));
    exception
        when Constraint_Error => Put_Line ("error!");
    end;
end Be_Careful;
"
```

"Ada has a very small number of predefined exceptions:

Constraint_Error is the main one you might see. It's raised:

– When bounds don't match or, in general, any violation of constraints.

– In case of overflow

– In case of null dereferences

– In case of division by 0

Program_Error might appear, but probably less often. It's raised in more arcane situations, such as for order of elaboration issues and some cases of detectable erroneous execution.

Storage_Error will happen because of memory issues, such as:

– Not enough memory (allocator)

– Not enough stack

Tasking_Error will happen with task related errors, such as any error happening during task activation." [22]

### 6.2.16 Topics not covered

Tasking (for concurrent programming) and protected objects/types are not covered in this document, among other features of Ada. These include: packages, contracts and object oriented programming. The standard library is also not covered here. It should be noted that it is possible to use C functions and variables in Ada and vice versa.

# 7. Παράρτημα Β (Appendix B: Other optimizations)

This appendix contains explanations of some other important optimizations.

## 7.1 Strength Reduction

"Strength reduction replaces expensive operations, such as multiplications and divisions, by less expensive ones, such as additions and subtractions. It is a special case of the method of finite differences applied to computer programs." [26]

"The best well known method, finite differences, consists of replacing each derivative by a difference quotient in the classic formulation." [67]

"For example, the sequence

0 , 3, 6, 9, 12, ...

has first differences (i.e., differences between successive elements) that consist of all 3s. Thus, it can be written as s_i = 3 * i for i = 0, 1, 2, ... or as s_(i+1) = s_i + 3 with s_0 = 0.

The second form is the strength-reduced version—instead of doing multiplications, we do additions. Similarly, the sequence

0 , 1, 4, 9, 16, 25, ...

has first differences

1 , 3, 5, 7, 9, ...

and second differences that consist of all 2s. It can be written as s_i = i^2 for i = 0, 1, 2, 3, ..., or as s_(i+1) = s_i + 2 * i + 1 for s_0 = 0, or as s_(i+1) = s_i + t_i where t_(i+1) = t_i + 2, s_0 = 0, and t_0 = 1. Here, after two finite differencing operations, we have reduced computing a sequence of squares to two additions for each square.

Strength reduction is not limited to replacing multiplication by additions and replacing addition by increment operations. Allen and Cocke ( [68]) discuss a series of applications for it, such as replacing exponentiation by multiplications, division and modulo by subtractions, and continuous differentiable functions by quadratic

interpolations." [26]

A few examples:

input:

"

a * 4;

a * 7;

a / 32767;

" [19]

output:

"

a << 2;

(a << 3) – a;

(a >> 15) + (a >> 30);

" [23]

## 7.2 Constant Propagation

"Constant propagation is a transformation that, given an assignment x = c for a variable x and a constant c, replaces later uses of x with uses of c as long as intervening assignments have not changed the value of x." [26]

"More generally, constant propagation reduces the number of registers needed by a procedure and increases the effectiveness of several other optimizations, such as constant-expression evaluation, induction-variable optimizations, and dependence-analysis-based transformations." [26]

Example:

input:

"

int x = 5;

int y = x * 2;

" [23]

output:

"

int y = 5 * 2;

" [23]

"To be most effective, constant propagation can be interleaved with constant folding. For safety, it requires a data-flow analysis." [23]

## 7.3 Copy Propagation

The equivalent of constant propagation for variables.

"Copy propagation is a transformation that, given an assignment x = y for some variables x and y, replaces later uses of x with uses of y, as long as intervening instructions have not changed the value of either x or y." [26]

This interacts with the scoping rules of the language. [23]

Example:

input:

"

```
x = y;
if (x > 1) {
    x = x * f(x – 1);
}
```

" [23]

output:

"

```
x = y;
if (y > 1) {
    x = y * f(y – 1);
}
```

" [23]

"One advantage of copy propagation is that it often turns the copy statement into dead code." [7]

## 7.4 Dead Code Elimination

"A variable is dead if it is not used on any path from the location in the code where it is defined to the exit point of the routine in question. An instruction is dead if it computes only values that are not used on any executable path leading from the instruction. If a dead variable's value is assigned to a local variable, the variable and the instruction that assigns to it are dead if the variable is not used on any executable path to the procedure's exit (including its being returned as the value of the procedure). If it is assigned to a variable with wider visibility, it

generally requires interprocedural analysis to determine whether it is dead, unless there is another instruction that assigns to the same variable on every executable path from its point of computation.

Programs may include dead code before optimization, but such code is much more likely to arise from optimization; strength reduction is an example of an optimization that produces dead code, and there are many others. Many optimizations create dead code as part of a division of labor principle: keep each optimization phase as simple as possible so as make it easy to implement and maintain, leaving it to other phases to clean up after it." [26]

Example:

input:

"

x = y * y; // x is dead!

... // x never used

x = z * z;

" [23]

output:

"

...

x = z * z;

" [23]

It is only applicable if the code in question is pure (i.e. it has no externally visible side effects (such as raising an exception, modifying a global variable or going into an infinite loop.)) [23]

## 7.5 Unreachable Code Elimination

"Unreachable code is code that cannot possibly be executed, regardless of the input data. It may never have been executable for any input data to begin with, or it may have achieved that status as a result of other optimizations. Its elimination has no direct effect on the execution speed of a program but obviously decreases the space the program occupies, and so may have secondary effects on its speed, particularly by improving its instruction-cache utilization.

Note that elimination of unreachable code is one of two transformations that are occasionally confused with each other. The other is dead-code

elimination, which removes code that is executable but that has no effect on the result of the computation being performed." [26]

## 7.6 Function Inlining

"Procedure integration, also called automatic inlining, replaces calls to procedures with copies of their bodies. It can be a very useful optimization, because it changes calls from opaque objects that may have unknown effects on aliased variables and parameters to local code that not only exposes its effects but that can be optimized as part of the calling procedure.

Some languages provide the programmer with a degree of control over inlining. C++, for example, provides an explicit inline attribute that may be specified for a procedure. Ada provides a similar facility. Both are characteristics of the procedure, not of the call site. While this is a desirable option to provide, it is significantly less powerful and discriminating than automatic procedure integration can be. An automatic procedure integrator can differentiate among call sites and can select the procedures to integrate according to machine-specific and performance-related criteria, rather than by depending on the user's intuition.

The opportunity to optimize inlined procedure bodies can be especially valuable if it enables loop transformations that were originally inhibited by having procedure calls embedded in loops or if it turns a loop that calls a procedure, whose body is itself a loop, into a nested loop." [26]

"If procedures are invoked indirectly through a pointer or via the method-dispatch mechanism prevalent in object-oriented programming, analysis of the program's pointers or references can in some cases determine the targets of the indirect invocations. If there is a unique target, inlining can be applied." [7]

Inlining is best done at the AST or relatively high-level IR. [23]

Example:

input:

"

```
int add (int x, int y)
{
  return x + y;
}
```

```
int sub (int x, int y)
{
  return add (x, -y);
}
```

" [27]

output:

"

```
int sub (int x, int y)
{
  return x + -y;
}
```

" [27]

"Code inlining might increase the code size." [23]

## 7.7 Tail Call Elimination

"Tail-call optimization and its special case, tail-recursion elimination, are transformations that apply to calls. They often reduce or eliminate a significant amount of procedure-call overhead and, in the case of tail-recursion elimination, enable loop optimizations that would otherwise not apply.

A call from procedure f() to procedure g() is a tail call if the only thing f() does, after g() returns to it, is itself return. The call is tail-recursive if f() and g() are the same procedure." [26]

"Combined with inlining, a recursive function can become as cheap as a while loop." [23]

"A tail-recursive call can be replaced with a goto, which avoids the overhead of the call and return and can also reduce stack space usage." [27]

Example:

input:

"

```
int f (int i)
{
```

```
 if (i > 0)
   {
     g (i);
     return f (i - 1);
   }
 else
   return 0;
}
" [27]
output:
"
int f (int i)
{

 entry:

 if (i > 0)
   {
     g (i);
     i--;
     goto entry;
   }
 else
   return 0;
}
" [27]
```

## 7.8 Loop fusion and Loop fission/distribution

"The fusion transform is characterized by mapping multiple loop indexes in the original program to the same loop index. The new loop fuses statements from different loops." [7]

"Fission is the inverse of fusion. It maps the same loop index for different statements to different loop indexes in the transformed code. This splits the original loop into multiple loops." [7]

"Although loop fusion reduces loop overhead, it does not always improve run-time performance, and may reduce run-time performance. For example, the memory architecture may provide better performance if two arrays are initialized in separate loops, rather than initializing both arrays simultaneously in one loop." [27]

Example:

Fusion input, fission output:

"

```
for (i = 0; i < 300; i++)
  a[i] = a[i] + 3;


for (i = 0; i < 300; i++)
  b[i] = b[i] + 4;
```

" [27]

Fusion output, fission input:

"

```
for (i = 0; i < 300; i++)
{
  a[i] = a[i] + 3;
  b[i] = b[i] + 4;
}
```

" [27]

## 7.9 Loop Collapsing

"Some nested loops can be collapsed into a single-nested loop to reduce loop overhead and improve run-time performance." [27]

"Loop collapsing can improve the opportunities for other optimizations, such as loop unrolling." [27]

Example:

input:

"

```
int a[100][300];


for (i = 0; i < 300; i++)
  for (j = 0; j < 100; j++)
    a[j][i] = 0;
```

" [27]

output:

"

```
int a[100][300];
int *p = &a[0][0];


for (i = 0; i < 30000; i++)
  *p++ = 0;
```

" [27]

## 7.10 Variable expansion

"Variable expansion in the body of an unrolled loop that has an unrolling factor of n selects variables that can be expanded into n separate copies, one for each copy of the loop body, and that can be combined at the loop's exits to produce the values that the original variables would have had. The expansion has the desirable property of decreasing the number of dependences in the loop, thus making instruction scheduling likely to be more effective when applied to it." [26]

Example:

input (rolled loop):

"

```
acc = 10;
max = 0;
imax = 0;
for(i = 1; i <= 100; i++)
{
```

```
    acc = acc + a[i] * b[i];
    if(a[i] > max)
    {
            max = a[i];
            imax = i;
    }
}
" (C appropriation of an example from [26])
output (unrolled loop):
"
acc = 10;
acc1 = 0;
max = 0;
max1 = 0;
imax = 0;
imax1 = 0;
i1 = 2;
for(i = 1; i <= 99; i += 2)
{
    acc = acc + a[i] * b[i];
    if(a[i] > max)
    {
            max = a[i];
            imax = i;
    }
    acc1 = acc1 + a[i1] * b[i1];
    if(a[i1] > max1)
    {
            max1 = a[i1];
            imax1 = i1;
```

```
    }
    i1 += 2;
}
acc += acc1;
if(max1 > max)
{
    max = max1;
    imax = imax1;
}
```

" (C appropriation of an example from [26])

## 7.11 Register renaming

"Register renaming is a transformation that may increase the flexibility available to code scheduling. It can be applied to low-level code to remove unnecessary dependences between instructions that use the same register by replacing some of the uses by other registers." [26]

## 7.12 Instruction combining

"Machine idioms are instructions or instruction sequences for a particular architecture that provide a more efficient way of performing a computation than one might use if one were compiling for a more generic architecture. Many machine idioms are instances of instruction combining, i.e., the replacement of a sequence of instructions by a single one that achieves the same effect." [26]

"The primary technique for recognizing opportunities to use machine idioms is pattern matching. The search has two main parts. The first part is looking for instructions whose purpose can be achieved by faster, more specialized instructions. The second part begins by looking for an instruction that may be the first of a group that can be combined into a shorter or faster sequence; finding one triggers a search for the other instruction(s) that are needed to form the appropriate group. Unless the target architecture allows functionally independent instructions to be combined into one (as, in some cases, was true for the Stanford mips architecture), the searching can be done most efficiently and effectively on the dependence DAG, rather than on straight-line code." [26]

"An instruction combiner may increase the number of superscalar instruction groups that are required to execute a sequence of instructions." [26]

# 8. Παράρτημα Γ (Appendix C: Benchmarking)

This chapter contains a benchmark to measure the improvement in execution time by applying the loop unrolling optimizations of csense. Although no hardware that csense optimizations target was at our disposal, a small benchmark was still attempted on one ordinary PC (desktop) machine.

## 8.1 Hardware the benchmark was performed on (specs)

The relevant part of the "sudo lshw" command shows the hardware this benchmark was performed on:

"

*-core

description: Motherboard

product: GA-A75M-UD2H

vendor: Gigabyte Technology Co., Ltd.

physical id: 0

*-firmware

description: BIOS

vendor: Award Software International, Inc.

physical id: 0

version: F2

date: 06/08/2011

size: 128KiB

capacity: 4MiB

capabilities: isa pci pnp apm upgrade shadowing cdboot bootselect socketedrom edd int13floppy360 int13floppy1200 int13floppy720 int13floppy2880 int5printscreen int9keyboard int14serial int17printer int10video acpi usb ls120boot zipboot biosbootspecification

*-cpu

description: CPU

product: AMD A8-3850 APU with Radeon(tm) HD Graphics

vendor: Advanced Micro Devices [AMD]

physical id: 4

bus info: cpu@0

version: AMD A8-3850 APU with Radeon(tm) HD Graphics

slot: Socket M2

size: 2899MHz

capacity: 3GHz

width: 64 bits

clock: 100MHz

capabilities: fpu fpu_exception wp vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ht syscall nx mmxext fxsr_opt pdpe1gb rdtscp x86-64 3dnowext 3dnow constant_tsc rep_good nopl nonstop_tsc cpuid extd_apicid aperfmperf pni monitor cx16 popcnt lahf_lm cmp_legacy svm extapic cr8_legacy abm sse4a misalignsse 3dnowprefetch osvw ibs skinit wdt hw_pstate vmmcall arat npt lbrv svm_lock nrip_save pausefilter cpufreq

*-cache:0

description: L1 cache

physical id: a

slot: Internal Cache

size: 128KiB

capacity: 128KiB

capabilities: synchronous internal write-back

configuration: level=1

*-cache:1

description: L3 cache

physical id: c

slot: External Cache

size: 1MiB

capacity: 1MiB

capabilities: synchronous internal write-back

configuration: level=3

*-cache

description: L1 cache

```
       physical id: b
       slot: Internal Cache
       size: 128KiB
       capacity: 128KiB
       capabilities: synchronous internal write-back
       configuration: level=1
    *-memory
       description: System Memory
       physical id: 27
       slot: System board or motherboard
       size: 8GiB
     *-bank:0
          description: DIMM 1333 MHz (0.8 ns) [empty]
          product: None
          vendor: None
          physical id: 0
          serial: None
          slot: A0
          width: 64 bits
          clock: 1333MHz (0.8ns)
     *-bank:1
          description: DIMM 1333 MHz (0.8 ns) [empty]
          product: None
          vendor: None
          physical id: 1
          serial: None
          slot: A1
          width: 64 bits
          clock: 1333MHz (0.8ns)
     *-bank:2
```

description: DIMM 1333 MHz (0.8 ns)

product: None

vendor: None

physical id: 2

serial: None

slot: A2

size: 4GiB

width: 64 bits

clock: 1333MHz (0.8ns)

*-bank:3

description: DIMM 1333 MHz (0.8 ns)

product: None

vendor: None

physical id: 3

serial: None

slot: A3

size: 4GiB

width: 64 bits

clock: 1333MHz (0.8ns)

"

## 8.2 The code used for the benchmark

The input C file was a simple matrix multiplication algorithm (found in and adjusted from [70]) which was split into two parts.

Part 1:

"

```
int main() {

    long a[50][50],b[50][50],mul[50][50],i,j,k;
```

"

Part 2:

"

```
    for(i=0;i<50;i++) {
        for(j=0;j<50;j++) {
            mul[i][j]=0;
            for(k=0;k<50;k++) {
                mul[i][j]+=a[i][k]*b[k][j];
            }
        }
    }

    return 0;
}
```

"

The initializations were generated by the following two commands:

"

```
for i in $(seq 0 49); do for j in $(seq 0 49); do echo "a[$i][$j] = $(( $RANDOM % 1000 ));" >> matMulabInit.txt; done done

for i in $(seq 0 49); do for j in $(seq 0 49); do echo "b[$i][$j] = $(( $RANDOM % 1000 ));" >> matMulabInit.txt; done done
```

"

(Larger array sizes were also tried but 10000x10000 caused the terminal to crash when a compilation was attempted by gcc, 1000x1000 caused the executable a segmentation fault when run, and 500x500 and 100x100 resulted in csense printing the

following error: "Global memory capacity exceeded; try to decrease global storage requirements!". 50x50 did not present any errors anywhere, and hence was selected for the benchmark.)

The final .c file was then generated using "cat":

"

cat matrixMultiplicationforcsense_noInit.c_firstHalf.txt matMulabInit.txt matrixMultiplicationforcsense_noInit.c_secondHalf.txt > matMulForCsense_50x50.c

"

The resulting Ada code had to be divided into an .adb and an .ads file to be compiled by gcc-gnat, and a third .adb needed to be created to call the function "main" in the first .adb. The files also needed to be renamed, as the GNAT compiler wanted them to be all lowercase letters.

The "run" or calling .adbs have a form similar to the following:

"

```
with matMulForCsense_50x50;
with Ada.Text_IO; use Ada.Text_IO;

procedure run_matMulForCsense_50x50 is
begin
  Put_Line("Execution returned: " & Integer'Image(matMulForCsense_50x50.main));
end run_matMulForCsense_50x50;
```

"

The corresponding .ads file:

"

```
package matMulForCsense_50x50 is

  type TYPE000 is array (0..49) of INTEGER;
  type TYPE001 is array (0..49) of TYPE000;
```

```
  function main
        return INTEGER;

end matMulForCsense_50x50;
"
```

The first 40 lines of the corresponding .adb (the result of the command "head -40 matmulforcsense_50x50.adb"):

```
"
------------------------------------------------------------
------------------  C front-end translator  ------------------
---------------------- CCC Compiler Group ----------------------
------------------------------------------------------------



------------------------------------------------------------

package body matMulForCsense_50x50 is

  function main
        return INTEGER is
    V004_i: INTEGER;
    V005_j: INTEGER;
    V006_k: INTEGER;
    GV000_V001_a: TYPE001;
    GV001_V002_b: TYPE001;
    GV002_V003_mul: TYPE001;
    TEMPORARY000: INTEGER;
    TEMPORARY001: INTEGER;
    TEMPORARY002: INTEGER;
```

```
    INDEX000: INTEGER;
    TEMPINT000: INTEGER;
    INDEX001: INTEGER;
    TEMPINT001: INTEGER;
    INDEX002: INTEGER;
    TEMPINT002: INTEGER;
  begin
    GV000_V001_a(0)(0) := 325;
    GV000_V001_a(0)(1) := 355;
    GV000_V001_a(0)(2) := 158;
    GV000_V001_a(0)(3) := 396;
    GV000_V001_a(0)(4) := 755;
    GV000_V001_a(0)(5) := 838;
    GV000_V001_a(0)(6) := 543;
    GV000_V001_a(0)(7) := 407;
    GV000_V001_a(0)(8) := 99;
    GV000_V001_a(0)(9) := 314;
    GV000_V001_a(0)(10) := 126;
    GV000_V001_a(0)(11) := 462;
"
```

The last 30 lines of the same .adb (the result of the command "tail -30 matmulforcsense_50x50.adb"):

```
"
    GV001_V002_b(49)(44) := 78;
    GV001_V002_b(49)(45) := 482;
    GV001_V002_b(49)(46) := 41;
    GV001_V002_b(49)(47) := 154;
    GV001_V002_b(49)(48) := 850;
    GV001_V002_b(49)(49) := 888;
    V004_i := 0;
```

```
      TEMPINT000 := 49;
      for INDEX000 in 0..TEMPINT000 loop
        V005_j := 0;
        TEMPINT001 := 49;
        for INDEX001 in 0..TEMPINT001 loop
          GV002_V003_mul(V004_i)(V005_j) := 0;
          V006_k := 0;
          TEMPINT002 := 49;
          for INDEX002 in 0..TEMPINT002 loop
            TEMPORARY000 := GV000_V001_a(V004_i)(V006_k);
            TEMPORARY001 := GV001_V002_b(V006_k)(V005_j);
            TEMPORARY002 := GV002_V003_mul(V004_i)(V005_j);
            TEMPORARY002 := TEMPORARY002 + (TEMPORARY000 * TEMPORARY001);
            GV002_V003_mul(V004_i)(V005_j) := TEMPORARY002;
            V006_k := V006_k + 1;
          end loop;
          V005_j := V005_j + 1;
        end loop;
        V004_i := V004_i + 1;
      end loop;
      return 0;
    end main;
  end matMulForCsense_50x50;
  "
```

The last 55 lines of the "-ouil" (ordinary unrolling) .adb (the result of the command "tail -55 matmulforcsense_50x50_ouil.adb"):

```
  "
      GV001_V002_b(49)(47) := 154;
      GV001_V002_b(49)(48) := 850;
      GV001_V002_b(49)(49) := 888;
```

```
V004_i := 0;
TEMPINT000 := 49;
for INDEX000 in 0..TEMPINT000 loop
  V005_j := 0;
  TEMPINT001 := 49;
  for INDEX001 in 0..TEMPINT001 loop
    GV002_V003_mul(V004_i)(V005_j) := 0;
    V006_k := 0;
    TEMPINT002 := 11;
    for INDEX002 in 0..TEMPINT002 loop
      OPT_TEMP_000 := V006_k;
      TEMPORARY000 := GV002_V003_mul(V004_i)(V005_j);
      TEMPORARY001 := GV000_V001_a(V004_i)(OPT_TEMP_000);
      TEMPORARY002 := GV001_V002_b(OPT_TEMP_000)(V005_j);
      TEMPORARY003 := TEMPORARY000 + (TEMPORARY001 * TEMPORARY002);
      GV002_V003_mul(V004_i)(V005_j) := TEMPORARY003;
      OPT_TEMP_000 := V006_k + 1;
      TEMPORARY004 := GV002_V003_mul(V004_i)(V005_j);
      TEMPORARY005 := GV000_V001_a(V004_i)(OPT_TEMP_000);
      TEMPORARY006 := GV001_V002_b(OPT_TEMP_000)(V005_j);
      TEMPORARY007 := TEMPORARY004 + (TEMPORARY005 * TEMPORARY006);
      GV002_V003_mul(V004_i)(V005_j) := TEMPORARY007;
      OPT_TEMP_000 := V006_k + 2;
      TEMPORARY008 := GV002_V003_mul(V004_i)(V005_j);
      TEMPORARY009 := GV000_V001_a(V004_i)(OPT_TEMP_000);
      TEMPORARY010 := GV001_V002_b(OPT_TEMP_000)(V005_j);
      TEMPORARY011 := TEMPORARY008 + (TEMPORARY009 * TEMPORARY010);
      GV002_V003_mul(V004_i)(V005_j) := TEMPORARY011;
      OPT_TEMP_000 := V006_k + 3;
      TEMPORARY012 := GV002_V003_mul(V004_i)(V005_j);
```

```
            TEMPORARY013 := GV000_V001_a(V004_i)(OPT_TEMP_000);
            TEMPORARY014 := GV001_V002_b(OPT_TEMP_000)(V005_j);
            TEMPORARY015 := TEMPORARY012 + (TEMPORARY013 * TEMPORARY014);
            GV002_V003_mul(V004_i)(V005_j) := TEMPORARY015;
            V006_k := V006_k + 4;
          end loop;
          TEMPINT002 := 49 - V006_k;
          for INDEX002 in 0..TEMPINT002 loop
            TEMPORARY016 := GV002_V003_mul(V004_i)(V005_j);
            TEMPORARY017 := GV000_V001_a(V004_i)(V006_k);
            TEMPORARY018 := GV001_V002_b(V006_k)(V005_j);
            TEMPORARY019 := TEMPORARY016 + (TEMPORARY017 * TEMPORARY018);
            GV002_V003_mul(V004_i)(V005_j) := TEMPORARY019;
            V006_k := V006_k + 1;
          end loop;
          V005_j := V005_j + 1;
        end loop;
        V004_i := V004_i + 1;
      end loop;
      return 0;
    end main;
  end matMulForCsense_50x50_ouil;
  "
```

The 25 last corresponding "-ofuil" (full loop unrolling) .adb lines (the result of the command "tail -25 matmulforcsense_50x50_ofuil.adb") were the following:

```
  "
            TEMPORARY029 := GV000_V001_a(V004_i)(OPT_TEMP_000);
            TEMPORARY030 := GV001_V002_b(OPT_TEMP_000)(V005_j);
            TEMPORARY031 := TEMPORARY028 + (TEMPORARY029 * TEMPORARY030);
            GV002_V003_mul(V004_i)(V005_j) := TEMPORARY031;
```

```
            OPT_TEMP_000 := V006_k + 8;
            TEMPORARY032 := GV002_V003_mul(V004_i)(V005_j);
            TEMPORARY033 := GV000_V001_a(V004_i)(OPT_TEMP_000);
            TEMPORARY034 := GV001_V002_b(OPT_TEMP_000)(V005_j);
            TEMPORARY035 := TEMPORARY032 + (TEMPORARY033 * TEMPORARY034);
            GV002_V003_mul(V004_i)(V005_j) := TEMPORARY035;
            OPT_TEMP_000 := V006_k + 9;
            TEMPORARY036 := GV002_V003_mul(V004_i)(V005_j);
            TEMPORARY037 := GV000_V001_a(V004_i)(OPT_TEMP_000);
            TEMPORARY038 := GV001_V002_b(OPT_TEMP_000)(V005_j);
            TEMPORARY039 := TEMPORARY036 + (TEMPORARY037 * TEMPORARY038);
            GV002_V003_mul(V004_i)(V005_j) := TEMPORARY039;
            V006_k := V006_k + 10;
          end loop;
          V005_j := V005_j + 1;
        end loop;
        V004_i := V004_i + 1;
      end loop;
      return 0;
    end main;
  end matMulForCsense_50x50_ofuil;
  "
```

Each running .adb was then compiled using gnatmake:

```
"
$ gnatmake run_matmulforcsense_50x50_ofuil.adb
gcc -c run_matmulforcsense_50x50_ofuil.adb
gcc -c matmulforcsense_50x50_ofuil.adb
matmulforcsense_50x50_ofuil.adb:69:07: warning: variable "INDEX000" is never
read and never assigned
```

matmulforcsense_50x50_ofuil.adb:71:07: warning: variable "INDEX001" is never read and never assigned

matmulforcsense_50x50_ofuil.adb:73:07: warning: variable "INDEX002" is never read and never assigned

matmulforcsense_50x50_ofuil.adb:5078:11: warning: for loop implicitly declares loop variable

matmulforcsense_50x50_ofuil.adb:5078:11: warning: declaration hides "INDEX000" declared at line 69

matmulforcsense_50x50_ofuil.adb:5081:14: warning: for loop implicitly declares loop variable

matmulforcsense_50x50_ofuil.adb:5081:14: warning: declaration hides "INDEX001" declared at line 71

matmulforcsense_50x50_ofuil.adb:5085:17: warning: for loop implicitly declares loop variable

matmulforcsense_50x50_ofuil.adb:5085:17: warning: declaration hides "INDEX002" declared at line 73

gnatbind -x run_matmulforcsense_50x50_ofuil.ali

gnatlink run_matmulforcsense_50x50_ofuil.ali

"

The resulting executables run despite the warnings.

## 8.3 Benchmark results

The results from running each executable three times are shown below:

"

$ time ./run_matmulforcsense_50x50

Execution returned: 0

real 0m0.020s

user 0m0.013s

sys0m0.003s

$ time ./run_matmulforcsense_50x50

Execution returned: 0

real 0m0.017s

user 0m0.012s

sys0m0.004s

$ time ./run_matmulforcsense_50x50

Execution returned: 0

real 0m0.018s

user 0m0.014s

sys0m0.003s

$ time ./run_matmulforcsense_50x50_ouil

Execution returned: 0

real 0m0.018s

user 0m0.011s

sys0m0.005s

$ time ./run_matmulforcsense_50x50_ouil

Execution returned: 0

real 0m0.016s

user 0m0.010s

sys0m0.005s

$ time ./run_matmulforcsense_50x50_ouil

Execution returned: 0

real 0m0.029s

user 0m0.012s

sys0m0.004s

$ time ./run_matmulforcsense_50x50_ofuil

Execution returned: 0

real 0m0.018s

user 0m0.014s

sys0m0.002s

$ time ./run_matmulforcsense_50x50_ofuil

Execution returned: 0

real 0m0.017s

user 0m0.013s

sys0m0.004s

$ time ./run_matmulforcsense_50x50_ofuil

Execution returned: 0

real 0m0.022s

user 0m0.010s

sys0m0.007s

"

**Table 17. Benchmark results (50x50 matrix multiplication running times in seconds)**

| | No optimization | Unrolled 4 times | Unrolled 10 times |
|---|---|---|---|
| **Run 1** | 0.020 | 0.018 | 0.018 |
| **Run 2** | 0.017 | 0.016 | 0.017 |
| **Run 3** | 0.018 | 0.029 | 0.022 |
| **Average** | 0.018 | 0.021 | 0.019 |

Πηγή: (None)

The results do not seem to suggest an improvement in execution time from the optimizations performed on the code running on this hardware. It is expected that the results would be different for a VLIW system, however.

# Παράρτημα Κώδικα (Code Appendix)

If you want the code shown in this paper (excluding the appendices), please ask for the accompanying .zip file that was provided to the library along with this document.