ISTANBUL TECHNICAL UNIVERSITY

ELECTRICAL-ELECTRONICS FACULTY

SUPPORTING CUSTOM INSTRUCTIONS WITH THE LLVM COMPILER FOR RISC-V PROCESSOR

SENIOR DESIGN PROJECT

Mehmet Eymen ÜNAY

Bora İNAN

Emrecan YİĞİT

Electronics and Communication Engineering

June 2023

ISTANBUL TECHNICAL UNIVERSITY

ELECTRICAL-ELECTRONICS FACULTY

SUPPORTING CUSTOM INSTRUCTIONS WITH THE LLVM COMPILER FOR RISC-V PROCESSOR

SENIOR DESIGN PROJECT

Mehmet Eymen ÜNAY
(040190218)

Bora İNAN
(040190205)

Emrecan YİĞİT
(040190203)

Electronics and Communication Engineering

Thesis Advisor: Dr. Tankut AKGÜL

June 2023

İSTANBUL TEKNİK ÜNİVERSİTESİ

ELEKTRİK-ELEKTRONİK FAKÜLTESİ

LLVM DERLEYECİSİYLE RISC-V İŞLEMCİ İÇİN EK BUYRUKLARIN DESTEKLENMESİ

LİSANS BİTİRME TASARIM PROJESİ

Mehmet Eymen ÜNAY
(040190218)

Bora İNAN
(040190205)

Emrecan YİĞİT
(040190203)

Elektronik ve Haberleşme Mühendisliği

Tez Danışmanı: Dr. Tankut AKGÜL

Haziran 2023

We are submitting the Senior Design Project Interim Report entitled as “SUPPORTING CUSTOM INSTRUCTIONS WITH THE LLVM COMPILER FOR RISC-V PROCESSOR ”. The Senior Design Project Interim Report has been prepared as to fulfill the relevant regulations of the Electronics and Communication Engineering Department of Istanbul Technical University. We hereby confirm that we have realized all stages of the Senior Design Project Interim Report by ourselves, and we have abided by the ethical rules with respect to academic and professional integrity .

Mehmet Eymen ÜNAY
(040190218)

Bora İNAN
(040190205)

Emrecan YİĞİT
(040190203)

FOREWORD

We would like to thank our project advisor Dr. Tankut Akgül who helped and guided us during the project and assisted us to overcome the difficulties we encountered. We would like to express our gratitude to him for helping us make progress with the project. Also, we would like to thank our friends and families who did not spare us their support during our education life.

June 2023

Mehmet Eymen ÜNAY
Bora İNAN
Emrecan YİĞİT

TABLE OF CONTENTS

Page

ABBREVIATIONS

ABI	Application Binary Interface
ALU	Arithmetic Logic Unit
ASIP	Application Specific Integrated Processor
AST	Abstract Syntax Tree
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
DCE	Dead Code Elimination
DSE	Dead Store Elimination
FPGA	Field Programmable Gate Arrays
GCC	GNU Compiler Collection
IR	Intermediate Representation
ISA	Instruction Set Architecture
IoT	Internet of Things
LLVM	Low Level Virtual Machine
LSB	Least Significant Bit
MC	Machine Code
MSB	Most Significant Bit
RISC	Reduced Instruction Set Computer
SSA	Static Single Assignment

SYMBOLS

C	: Capacitance
H	: The amount of heat
$\mathbf{M_x}$, $\mathbf{M_y}$	: Torque Components
$\mathbf{N_x}$, $\mathbf{N_y}$, $\mathbf{N_z}$	: Normal Power Components
q	: Phase load
t	: Time
u, v	: Displacement Vector Components
w	: Angular velocity
XC	: Capacitive reactance
XL	: Inductive reactance
$\boldsymbol{\alpha}$	: Angle of deviation from the direction of the principal stresses
$\boldsymbol{\rho}$	: Density
$\boldsymbol{\sigma}_x$, $\boldsymbol{\sigma}_y$, $\boldsymbol{\sigma}_{xy}$	: Shell internal stresses

LIST OF TABLES

Page

LIST OF FIGURES

SUPPORTING CUSTOM INSTRUCTIONS WITH THE LLVM COMPILER FOR RISC-V PROCESSOR

SUMMARY

1 line spacing must be set for summaries. For theses in Turkish, the summary in Turkish must have 300 words minimum and span 1 to 3 pages, whereas the extended summary in English must span 3-5 pages.

For theses in English, the summary in English must have 300 words minimum and span 1-3 pages, whereas the extended summary in Turkish must span 3-5 pages.

A summary must briefly mention the subject of the thesis, the method(s) used and the conclusions derived. References, figures and tables must not be given in Summary.

Above the Summary, the thesis title in first level title format (i.e., 72 pt before and 18 pt after paragraph spacing, and 1 line spacing) must be placed. Below the title, the expression **ÖZET** (for summary in Turkish) and **SUMMARY** (for summary in English) must be written horizontally centered.

It is recommended that the summary in English is placed before the summary in Turkish.

LLVM DERLEYECİSİYLE RISC-V İŞLEMCİ İÇİN EK BUYRUKLARIN DESTEKLENMESİ

ÖZET

Özet hazırlanırken 1 satır boşluk bırakılır. Türkçe tezlerde, Türkçe özet 300 kelimeden az olmamak kaydıyla 1-3 sayfa, İngilizce genişletilmiş özet de 3-5 sayfa arasında olmalıdır.

İngilizce tezlerde ise, İngilizce özet 300 kelimeden az olmamak kaydıyla 1-3 sayfa, Türkçe genişletilmiş özet de 3-5 sayfa arasında olmalıdır.

Özetlerde tezde ele alınan konu kısaca tanıtılarak, kullanılan yöntemler ve ulaşılan sonuçlar belirtilir. Özetlerde kaynak, şekil, çizelge verilmez.

Özetlerin başında, birinci dereceden başlık formatında tezin adı (önce 72, sonra 18 punto aralık bırakılarak ve 1 satır aralıklı olarak) yazılacaktır. Başlığın altına büyük harflerle sayfa ortalanarak (Türkçe özet için) **ÖZET** ve (İngilizce özet için) SUMMARY yazılmalıdır.

Türkçe tezlerde Türkçe özetin İngilizce özetten önce olması önerilir.

1 line spacing must be set for summaries. For theses in Turkish, the summary in Turkish must have 300 words minimum and span 1 to 3 pages, whereas the extended summary in English must span 3-5 pages. For theses in English, the summary in English must have 300 words minimum and span 1-3 pages, whereas the extended summary in Turkish must span 3-5 pages. A summary must briefly mention the subject of the thesis, the method(s) used and the conclusions derived. References, figures and tables must not be given in Summary. Above the Summary, the thesis title in first level title format (i.e., 72 pt before and 18 pt after paragraph spacing, and 1 line spacing) must be placed. Below the title, the expression **ÖZET** (for summary in Turkish) and **SUMMARY** (for summary in English) must be written horizontally centered. It is recommended that the summary in English is placed before the summary in Turkish.

1. INTRODUCTION

1.1 Interim Report Explanations

For the last few months our main purpose is matching the S-BOX and ROT patterns. During our work, we found out that the LLVM RISC-V back-end includes the ROT instruction as an extension. We enabled its flag and set the custom opcode so we are able to use the ROT instruction. How to enable and use the ROT instruction is explained in this form. Implementing S-BOX was challenging for us since it is much more complicated than the other patterns we matched. TableGen was not sufficient for our algorithm so we decided to match the pattern using C++ during the instruction selection phase. We started to build the algorithm in the RISCVISelDAGToDAG.cpp file however we found out that matching such a complex pattern is not quite reasonable. To implement s-box instruction, using intrinsic functions is a preferred way because the s-box algorithm is not a common pattern and the user may use it in different ways. Under the supervision of our project advisor, we committed to matching the simpler pattern parts of the s-box. Progresses in coding and research are explained in the report.

1.2 Purpose of Project

Application-Specific Instruction Set Processors (ASIPs) are becoming more popular with the development of embedded systems. The specialization of the core causes a tradeoff between flexibility and performance. For special purposes, using ASIPs increases efficiency however, we can program a custom ASIP only by using assembly instructions that we defined. Programming custom processors with assembly language is not a preferred way of coding. We are also not able to use high-level languages because compiling tools are designed for common architectures with certain instructions. The ability to add custom instructions to compilers will enable us to make more use of custom hardware designs.

ASIPs are feasible for all application-specific embedded systems like consumer, industrial, automotive, home appliances, cryptology, medical, telecommunication, commercial, aerospace, and military applications. The custom back-end that we will design under the supervision of Dr. Tankut Akgül, is going to serve the processor designed by Prof. Dr. Sıddıka Berna Örs Yalçın's research team. When the project is completed, Prof. Yalçın is going to be able to produce the assembly codes that are compatible with the processor's extended instruction set in addition to RISC-V.

There are two ways to avoid designing a specific compiler for embedded microprocessors. The first one is using a common processor that already has compiling tools. The advantage of it is reducing the costs for both hardware and software designs. However, it reduces efficiency dramatically because the processor is not designed for a specific task, and hardware cost increases while the speed is decreasing. Another way is using an application-specific instruction set processor and programming with the assembly instructions that the hardware designer defined. Theoretically, all efficiency benefits can be achieved but programming with assembly instructions increases coding difficulty excessively.

Prof. Yalçın and her team are studying designing application-specific instruction set processors. The purpose of this project is to create a compiler back-end for a processor that supports custom instructions on top of RISC-V instructions. This compiler is going to help to program the custom processor by using high-level languages. Existing RISC-V compilers are not able to produce efficient assembly codes for ASIPs. Therefore a need arose for a compiler back-end. The main reason for choosing this project is that we wanted to meet an actual need for a critical existing problem. The project has the potential to be the bridge between hardware and software of custom hardware projects in research, enabling them to be candidates for critical applications. Our team has skills and experience in low-level programming and digital design. Our project advisor Dr. Tankut Akgül's lecture on microprocessors led us to work with him on the embedded systems. Our team member Mehmet Eymen Ünay is a double major student in the computer engineering department. Bora İnan has experience in software development in the defense industry and Emrecan Yiğit is working on low-level robotics programming. Emrecan and Bora are taking a digital system Design and application course from Prof. Yalçın to learn Verilog and get to know about hardware design. All of us have assembly, C, and FPGA programming experience. We also have worked with several open-source frameworks on different Linux distros. We consider that our skills and experiences match the project we will be doing.

2. BASICS OF A COMPILER

A compiler is a software that converts source code written in a high-level programming language into machine code appropriate for a particular computer architecture. There are different stages of a compiler but they can be grouped into two main parts such as “Front-End” and “Back-End”. These parts of the compiler are also called the analysis and synthesis parts of the compiler. The analysis stage separates the source program into its individual components and applies a grammatical structure to them. The source code is then represented in an intermediate stage using this structure. The synthesis phase creates the final target program by using the intermediate representation. We can think of the compilation process as a series of phases, each of which takes the source program and transforms it into another representation [8]. These phases can be seen in Figure 2.1.

2.1 Front-End

2.1.1 Lexical Analysis

The compiler breaks down the source code into smaller units called lexemes, which are pieces of code that correspond to specific patterns in the code. These lexemes are then converted into tokens that can be used for syntax and semantic analyses.

2.1.2 Syntax Analysis

The compiler checks that the code follows the proper syntax for the programming language it is written in. This process is also called parsing. As part of this step, the compiler often creates abstract syntax trees in order to represent the logical structure of different parts of the code.

2.1.3 Semantic Analysis

The compiler checks that the code makes logical sense, not just that it follows the syntax of the programming language. This step goes beyond syntax analysis by

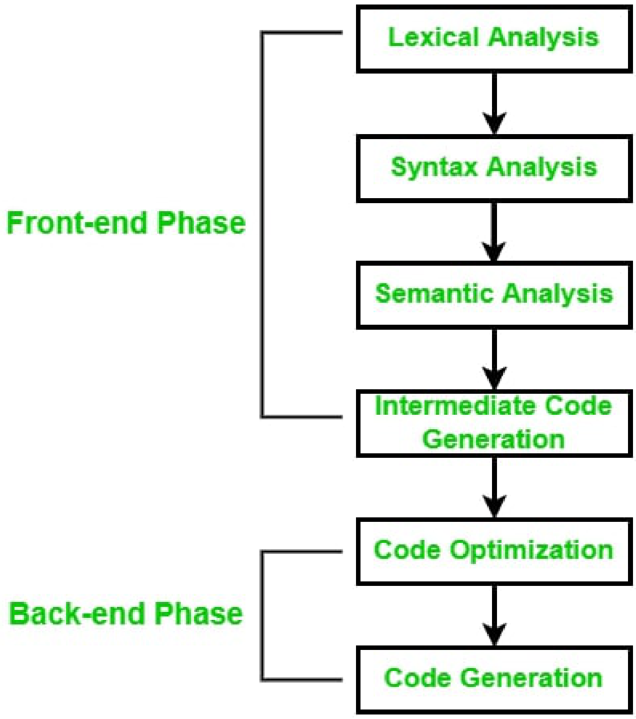


Figure 2.1 : Compiler Stages

ensuring that the code is correct. For example, the compiler might check that variables have been declared correctly and given the appropriate data types. This process is known as semantic analysis.

2.1.4 IR Code Generation

After the source code has been analyzed for lexemes, syntax, and semantics, the compiler creates an intermediate representation (IR) of the code. This intermediate code is going to be converted to machine code in the last two phases. These two phases are platform-dependent, meaning they are specific to a particular hardware architecture but the previous phases were not. Therefore, to create a new compiler, it is not necessary to start from scratch. Instead, it is possible to use the intermediate code from an existing compiler and build the final stages of the process for a specific platform. Because of that, we are interested in the back-end part for our project.

2.2 Back-End

2.2.1 Optimization

The intermediate code is prepared for the final code generation step. This process does not change the meaning or functionality of the code, but it can make the program run faster and more efficiently. A directed acyclic graph (DAG) used in the compiler design process might represent the dependencies between different instructions in the IR code, such as the order in which those instructions need to be executed or the data dependencies between them. The DAG can be used by the compiler to identify opportunities for optimization, such as removing unnecessary instructions, combining some of them, or rearranging the order of execution to reduce the number of resources required by the code.

2.2.2 Target Code Generation

The target code generator is the final stage of the compilation process, and its main function is to convert the optimized code into a form that the machine can understand. The optimized code is turned into a relocatable machine code. The relocatable machine code is the input to the linker and loader, which are responsible for combining the code with other necessary resources and preparing it for execution Target code generation can be divided into different parts:

2.2.3 Instruction Selection

IR is the input of the code generation step, and it maps the IR into the target machine's instruction set. There may be multiple ways for converting one representation, so the code generator tries to select the most suitable instructions.

2.2.4 Register Allocation

There may be many different variables/values in a program. The code generator decides which registers to use to keep these values.

2.2.5 Instruction Scheduling

The code generator determines the sequence in which instructions will be executed and creates schedules for the execution of those instructions.

3. THE LLVM COMPILER

LLVM is a collection of modular and flexible libraries and a toolchain software that can be used to build a wide variety of compilers and other tools. LLVM compilers are organized into a set of libraries that implement the parts of a compiler. There are different front-end libraries a for every language and different back-end libraries for every architecture. There is only one common intermediate representation optimizer that connects specific front-end and back-end.

LLVM IR is the common representation of languages. Various LLVM front-ends translate related languages into IR. Related back-end compiles IR into assembly according to the target hardware. This structure helps to increase flexibility between front-ends and back-ends. With this structure, we are able to have compilers for every combination of M source codes and N targets with M front-end and N back-end instead of M*N compilers. In our case, we do not have to struggle with the front-end also our back-end will be working for every source code of LLVM because of this benefit. The front-end we use in the developing process will be clang which is the LLVM C/C++ front-end [9].

The concept of instrinsic function is supported by LLVM. They are internal functions and they have their semantics directly defined by LLVM itself. LLVM provides both

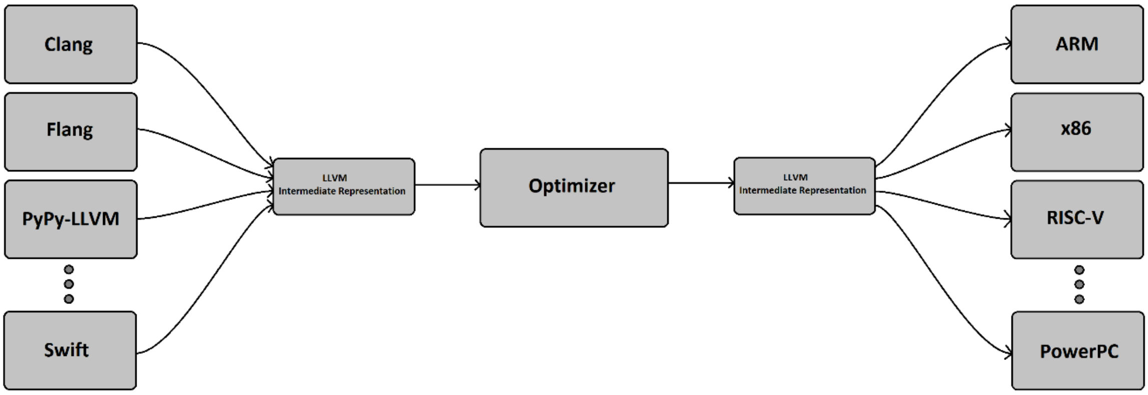

Clang
Flang
PyPy-LLVM
Swift
LLVM Intermediate Representation
Optimizer
LLVM Intermediate Representation
ARM
x86
RISC-V
PowerPC

Figure 3.1 : Front-end and Back-end libraries connected by LLVM

target-independent and target-specific intrinsics. [10] These intrinsics have well known semantics and names and they must adhere to certain restrictions. In general, these intrinsics serve as an expansion mechanism for the LLVM language that does not necessitate modifying all of the transformations in LLVM when introducing changes to the language. Intrinsic function names start with “llvm.” [11]

There are also builtin functions that are supported by Clang. Some of these have the same syntax as GCC. In addition to these, Clang supports other builtin functions that GCC doesn’t. Some of these are __builtin_shufflevector, __builtin_unreachable etc. As we can see, these builtin functions start with double underscores.

Intrinsic functions and builtin functions are two seperate things and should not be confused. Builtins are at C level (source code) while intrinsics are LLVM IR level and a builtin may or may not be expanded into intrinsic calls.

3.1 Parts of the Clang Front-end

3.1.1 Clang Lex Library

Clang Lex Library is a typical lexer implemented as finite state machines that read source code one character at a time and transition between different states based on the characters read. The Clang lexer, which is a front-end compiler for the C, C++, and Objective-C programming languages, uses this approach to filter out comments and white space, recognize and tokenize language elements such as keywords, identifiers, and operators, and handle escape sequences and string literals. The implementation files of the Clang lexer can be found in the llvm-project/clang/lib/lex directory within the LLVM infrastructure.

3.1.2 Clang Parse Library

Clang Parse Library is the parser that takes the tokens produced by the lexer and constructs an abstract syntax tree (AST) to represent the structure and meaning of the source code. The Clang parser checks the source code for proper syntax and resolves symbols and identifiers. It also performs type-checking to ensure the source code follows the rules of the programming language. It creates the AST, a tree-like structure, that represents the source code in a way that is easily processed by the compiler. The implementation files of the Clang lexer can be found in the llvm-project/clang/lib/parse directory within the LLVM infrastructure.

3.1.3 Clang Sema Library

Clang Sema Library is a semantic analyzer that involves examining the meaning and context of the source code in a program. In Clang, semantic analysis is a phase in the compilation process that analyzes the abstract syntax tree (AST) generated by the parser to verify that the source code conforms to the rules of the programming language and is properly constructed. Semantic analysis performs various checks and transformations on the AST to ensure the source code is correct. The implementation files of the Clang semantic analyzer can be found in the llvm-project/clang/lib/sema directory within the LLVM infrastructure.

3.1.4 Clang CodeGen Library

Clang CodeGen is the code generation library that takes the abstract syntax tree which is generated by the parser and corrected by the semantic analyzer as input. It generates the intermediate representation code and produces a .ll file which will be used in the back-end. The implementation files of the Clang lexer can be found in the llvm-project/clang/lib/CodeGen directory within the LLVM infrastructure.

3.2 LLVM IR Optimizer

LLVM IR is an intermediate representation which is which serves as a common ground for front-ends and back-ends. LLVM IR is not as high level as programming languages but it provides more information than assembly by having types or more expressive functions. LLVM IR instructions are stored in Basic Block structures which contain sequential IR instructions with an entry and exit.

LLVM Optimizer is a common optimization medium used for every possible source-target combination of a compiler. It takes the output file of CodeGen as input and runs three types of passes:

1. Analysis passes: These passes analyze the IR and collect information about the IR without modifying the IR.

2. Transformation passes: These passes modify the IR by using the information gathered from Analysis passes. The optimizations are the product of these transformations.

3. Utility passes: These passes are used to perform tasks such as printing the IR or verifying the IR.

The output of the optimizer becomes the input for the target back-end which lowers the LLVM IR to the target Assembly. As the generated LLVM IR at the end of the optimizations is the object of pattern matching and assembly support for any custom instruction, it is a critical part of the design process.

3.2.1 Analysis Passes

There are almost 40 analysis passes. The significant documented analysis passes are listed below:

- Exhaustive Alias Analysis Precision Evaluator
- Basic Alias Analysis (stateless AA impl)
- Basic CallGraph Construction
- Count Alias Analysis Query Responses
- Dependence Analysis
- AA use debugger
- Dominance Frontier Construction
- Dominator Tree Construction
- Simple mod/ref analysis for globals
- Counts the various types of Instructions
- Interval Partition Construction
- Induction Variable Users
- Lazy Value Information Analysis
- LibCall Alias Analysis
- Statically lint-checks LLVM IR
- Natural Loop Information
- Memory Dependence Analysis
- Decodes module-level debug info
- Post-Dominance Frontier Construction
- Post-Dominator Tree Construction
- Alias Set Printer
- Find Used Types
- Detect single entry single exit regions
- Scalar Evolution Analysis
- ScalarEvolution-based Alias Analysis
- Stack Safety Analysis
- Target Data Layout

3.2.2 Transformation Passes

There are almost 60 transformation passes. The documented transformation passes are listed below:

- Aggressive Dead Code Elimination
- Inliner for always_inline functions
- Promote 'by reference' arguments to scalars
- Basic-Block Vectorization
- Profile Guided Basic Block Placement
- Break critical edges in CFG
- Optimize for code generation
- Merge Duplicate Global Constants
- Dead Code Elimination
- Dead Argument Elimination
- Dead Type Elimination
- Dead Instruction Elimination

- Dead Store Elimination
- Deduce function attributes
- Dead Global Elimination
- Global Variable Optimizer
- Global Value Numbering
- Canonicalize Induction Variables
- Function Integration/Inlining
- Combine redundant instructions
- Combine expression patterns
- Internalize Global Symbols
- Interprocedural Sparse Conditional Constant Propagation
- Jump Threading
- Loop-Closed SSA Form Pass
- Loop Invariant Code Motion
- Delete dead loops
- Extract loops into new functions
- Extract at most one loop into a new function
- Loop Strength Reduction
- Rotate Loops
- Canonicalize natural loops
- Unroll loops
- Unroll and Jam loops
- Unswitch loops
- Lower global destructors
- Lower atomic intrinsics to non-atomic form
- Lower invokes to calls, for unwindless code generators
- Lower SwitchInsts to branches
- Promote Memory to Register
- MemCpy Optimization
- Merge Functions
- Unify function exit nodes
- Partial Inliner
- Remove unused exception handling info
- Reassociate expressions
- Relative lookup table converter
- Demote all values to stack slots
- Scalar Replacement of Aggregates
- Sparse Conditional Constant Propagation
- Simplify the CFG
- Code sinking
- Strip all symbols from a module
- Strip debug info for unused symbols
- Strip Unused Function Prototypes
- Strip all llvm.dbg.declare intrinsics
- Strip all symbols, except dbg symbols, from a module
- Tail Call Elimination

[12]

3.2.3 Case Study: Optimizations on S-box

One of the research topics of this study was to observe the changes to a function shown in Code 3.2.3 performing S-box with bitwise operations. As the pattern is large an enormous Assembly is generated without enabling optimizations. However, when the optimizations are enabled the final LLVM IR file shown in Code 3.1 and Assembly is significantly smaller.

```
typedef struct {
  int x[5];
} ascon_state_t;

void sbox(ascon_state_t state) {
    int t0, t1, t2, t3, t4;
    state.x[0] ^= state.x[4];
    state.x[4] ^= state.x[3];
    state.x[2] ^= state.x[1];
    t0  = state.x[0];
    t1  = state.x[1];
    t2  = state.x[2];
    t3  = state.x[3];
    t4  = state.x[4];
    t0 =~ t0;
    t1 =~ t1;
    t2 =~ t2;
    t3 =~ t3;
    t4 =~ t4;
    t0 &= state.x[1];
    t1 &= state.x[2];
    t2 &= state.x[3];
    t3 &= state.x[4];
    t4 &= state.x[0];
    state.x[0] ^= t1;
    state.x[1] ^= t2;
    state.x[2] ^= t3;
    state.x[3] ^= t4;
    state.x[4] ^= t0;
    state.x[1] ^= state.x[0];
    state.x[0] ^= state.x[4];
    state.x[3] ^= state.x[2];
    state.x[2] =~ state.x[2];

    return;
}
```

At the end of optimization passes the following IR will be generated:

```
; Function Attrs: mustprogress nofree norecurse nosync nounwind willreturn memory(argmem: readwrite) uwtable
define dso_local void @sbox(ptr nocapture noundef %state) local_unnamed_addr #0 {
entry:
  %arrayidx = getelementptr inbounds [5 x i32], ptr %state, i32 0, i32 4
  %0 = load i32, ptr %arrayidx, align 4, !tbaa !7
  %1 = load i32, ptr %state, align 4, !tbaa !7
  %xor = xor i32 %1, %0
  %arrayidx4 = getelementptr inbounds [5 x i32], ptr %state, i32 0, i32 3
  %2 = load i32, ptr %arrayidx4, align 4, !tbaa !7
  %xor7 = xor i32 %2, %0
  %arrayidx9 = getelementptr inbounds [5 x i32], ptr %state, i32 0, i32 1
  %3 = load i32, ptr %arrayidx9, align 4, !tbaa !7
  %arrayidx11 = getelementptr inbounds [5 x i32], ptr %state, i32 0, i32 2
  %4 = load i32, ptr %arrayidx11, align 4, !tbaa !7
  %xor12 = xor i32 %4, %3
  %not = xor i32 %xor, -1
  %not23 = xor i32 %3, -1
  %not24 = xor i32 %xor12, -1
  %not25 = xor i32 %2, -1
  %not26 = xor i32 %xor7, -1
  %and = and i32 %3, %not
  %and31 = and i32 %4, %not23
  %and34 = and i32 %2, %not24
  %and37 = and i32 %0, %not25
  %and40 = and i32 %xor, %not26
  %xor43 = xor i32 %and31, %xor
  %xor46 = xor i32 %and34, %3
  %xor49 = xor i32 %xor12, %and37
  %xor52 = xor i32 %and40, %2
  %xor55 = xor i32 %and, %xor7
  store i32 %xor55, ptr %arrayidx, align 4, !tbaa !7
  %xor60 = xor i32 %xor46, %xor43
  store i32 %xor60, ptr %arrayidx9, align 4, !tbaa !7
  %xor65 = xor i32 %xor43, %xor55
  store i32 %xor65, ptr %state, align 4, !tbaa !7
  %xor70 = xor i32 %xor49, %xor52
  store i32 %xor70, ptr %arrayidx4, align 4, !tbaa !7
  %not73 = xor i32 %xor49, -1
  store i32 %not73, ptr %arrayidx11, align 4, !tbaa !7
  ret void
}
```

Code 3.1 : Optimized S-box LLVM IR

LLVM optimization passes are responsible for the simplification of IR. In this case, the following passes were the passes changing the IR and were run sequentially.

1. InferFunctionAttrsPass

2. SROAPass

3. EarlyCSEPass
4. GlobalOptPass
5. InstCombinePass
6. EarlyCSEPass
7. InstCombinePass
8. ReassociatePass
9. InstCombinePass
10. DSEPass
11. PostOrderFunctionAttrsPass

As it can be observed some passes can run several times.

3.2.3.1 Infer Function Attributes - InferFunctionAttrsPass

This pass adds metadata to LLVM IR, by analyzing it. Function attributes are used to pass information about functions between LLVM passes.

```
; Function Attrs: nocallback nofree nosync nounwind willreturn ↩
   memory(argmem: readwrite)
declare void @llvm.lifetime.start.p0(i64 immarg, ptr nocapture) #1

; Function Attrs: nocallback nofree nosync nounwind willreturn ↩
   memory(argmem: readwrite)
declare void @llvm.lifetime.end.p0(i64 immarg, ptr nocapture) #1
```

Code 3.2 **:** LLVM IR Before InferFunctionAttrsPass

```
; Function Attrs: mustprogress nocallback nofree nosync nounwind ↩
   willreturn memory(argmem: readwrite)
declare void @llvm.lifetime.start.p0(i64 immarg, ptr nocapture) #1

; Function Attrs: mustprogress nocallback nofree nosync nounwind ↩
   willreturn memory(argmem: readwrite)
declare void @llvm.lifetime.end.p0(i64 immarg, ptr nocapture) #1
```

Code 3.3 **:** LLVM IR After InferFunctionAttrsPass

Function attribute is inferred as "mustprogress" as the lifetime starting function is interacting with its environment in an observable way making a memory access [13].

The lifetime function decides the accessibility of the pointer to the memory. When memory is allocated the the lifetime of pointer to the memory starts and ends when deallocated [14].

3.2.3.2 Scalar Replacement of Aggregates - SROAPass

Aggregate IR instructions such as "alloca" are promoted to registers. The promotion to registers also means the lifetime is under control and the explicit lifetime intrinsic calls can be removed.

```
%state.indirect_addr = alloca ptr, align 4
%t0 = alloca i32, align 4
%t1 = alloca i32, align 4
%t2 = alloca i32, align 4
%t3 = alloca i32, align 4
%t4 = alloca i32, align 4
store ptr %state, ptr %state.indirect_addr, align 4, !tbaa !7
call void @llvm.lifetime.start.p0(i64 4, ptr %t0) #2
call void @llvm.lifetime.start.p0(i64 4, ptr %t1) #2
call void @llvm.lifetime.start.p0(i64 4, ptr %t2) #2
call void @llvm.lifetime.start.p0(i64 4, ptr %t3) #2
call void @llvm.lifetime.start.p0(i64 4, ptr %t4) #2
```

Code 3.4 : Alloca and Lifetime Start Lines Removed From LLVM IR Before SROAPass

```
store i32 %not73, ptr %arrayidx75, align 4, !tbaa !11
call void @llvm.lifetime.end.p0(i64 4, ptr %t4) #2
call void @llvm.lifetime.end.p0(i64 4, ptr %t3) #2
call void @llvm.lifetime.end.p0(i64 4, ptr %t2) #2
call void @llvm.lifetime.end.p0(i64 4, ptr %t1) #2
call void @llvm.lifetime.end.p0(i64 4, ptr %t0) #2
```

Code 3.5 : Lifetime End Lines Removed From LLVM IR Before SROAPass

An important transformation SROA does is promoting the use of registers instead of using the stack for local variables and using "Load/Store" operations to use them in the unoptimized IR [15]. "Store/Load" operations are reduced significantly in this stage, especially for intermediate variables where the C code is not referring to the array directly.

SROA pass relies on the analysis passes of Alias Analysis through the collection of analysis passes for Loads.

```
    t0 =~ t0;
    t1 =~ t1;
    t2 =~ t2;
    t3 =~ t3;
    t4 =~ t4;
```

Code 3.6 : NOT operations between Intermediate Variables

```
store i32 %10, ptr %t4, align 4, !tbaa !11
%11 = load i32, ptr %t0, align 4, !tbaa !11
%not = xor i32 %11, -1
store i32 %not, ptr %t0, align 4, !tbaa !11
%12 = load i32, ptr %t1, align 4, !tbaa !11
%not23 = xor i32 %12, -1
store i32 %not23, ptr %t1, align 4, !tbaa !11
%13 = load i32, ptr %t2, align 4, !tbaa !11
%not24 = xor i32 %13, -1
store i32 %not24, ptr %t2, align 4, !tbaa !11
%14 = load i32, ptr %t3, align 4, !tbaa !11
%not25 = xor i32 %14, -1
store i32 %not25, ptr %t3, align 4, !tbaa !11
%15 = load i32, ptr %t4, align 4, !tbaa !11
%not26 = xor i32 %15, -1
store i32 %not26, ptr %t4, align 4, !tbaa !11
```

Code 3.7 : Intermediate Load and Stores in LLVM IR Before SROAPass

```
%not = xor i32 %6, -1
%not23 = xor i32 %7, -1
%not24 = xor i32 %8, -1
%not25 = xor i32 %9, -1
%not26 = xor i32 %10, -1
```

Code 3.8 : Load and Stores Promoted to Registers in LLVM IR After SROAPass

Similar to the dramatic change in the previous example, operations between the array elements and the intermediate variables are optimized so that the registers are used instead of the stack.

According to the statistics obtained from the "opt" tool of LLVM:

5 mem2reg - Number of alloca’s promoted within one block

1 mem2reg - Number of alloca’s promoted with a single store

1 sroa - Maximum number of partitions per alloca

8 sroa - Maximum number of uses of a partition

41 sroa - Number of alloca partition uses rewritten

6 sroa - Number of alloca partitions formed

6 sroa - Number of allocas analyzed for replacement

41 sroa - Number of instructions deleted

6 sroa - Number of allocas promoted to SSA values

3.2.3.3 Early Common Subexpression Elimination - EarlyCSEPass

Performs a simple dominator tree walk, eliminating trivially redundant instructions. Dominator tree is a type of tree where every parent node dominates the child node.

The definition of dominance from graph theory is that every path to the dominated node passes through the dominator node [16].

Early CSE pass relies on MemorySSA analysis which analyses by representing memory operations in SSA form [17, 18].

```
define dso_local void @sbox(ptr noundef %state) #0 {
entry:
  %x = getelementptr inbounds %struct.ascon_state_t, ptr %state, i32 0, i32 0
  %arrayidx = getelementptr inbounds [5 x i32], ptr %x, i32 0, i32 4
  %0 = load i32, ptr %arrayidx, align 4, !tbaa !7
  %x1 = getelementptr inbounds %struct.ascon_state_t, ptr %state, i32 0, i32 0
  %arrayidx2 = getelementptr inbounds [5 x i32], ptr %x1, i32 0, i32 0
  %1 = load i32, ptr %arrayidx2, align 4, !tbaa !7
```

Code 3.9 : Redundant Load Instructions in LLVM IR Before EarlyCSEPass

In Code 3.9 you can see that "%x1" and "%arrayidx2" are equal to the function argument "%state". EarlyCSE pass detects this redundant condition and uses the already present "%state" pointer in the output.

```
define dso_local void @sbox(ptr noundef %state) #0 {
entry:
  %arrayidx = getelementptr inbounds [5 x i32], ptr %state, i32 0, i32 4
  %0 = load i32, ptr %arrayidx, align 4, !tbaa !7
  %1 = load i32, ptr %state, align 4, !tbaa !7
```

Code 3.10 : Optimized Load Instructions in LLVM IR After EarlyCSEPass

Another remark from this example is that the pointer calculation is done in two instructions by "getelementptr" LLVM instruction which accesses the struct's address and then the element's address in it. EarlyCSE combines these two instructions outputting the offset calculated pointers.

A natural result of these simple optimizations is that the section which makes use of registers has increased. It can be seen in Code 3.11 that the recalculation of pointers by getelementptr is removed as they are used at the beginning of the function, as shown partly in Code 3.10.

```
  %x13 = getelementptr inbounds %struct.ascon_state_t, ptr %state, i32 0, i32 0
  %arrayidx14 = getelementptr inbounds [5 x i32], ptr %x13, i32 0, i32 0
  %6 = load i32, ptr %arrayidx14, align 4, !tbaa !7
  %x15 = getelementptr inbounds %struct.ascon_state_t, ptr %state,↩
    i32 0, i32 0
  %arrayidx16 = getelementptr inbounds [5 x i32], ptr %x15, i32 0,↩
    i32 1
  %7 = load i32, ptr %arrayidx16, align 4, !tbaa !7
  %x17 = getelementptr inbounds %struct.ascon_state_t, ptr %state,↩
    i32 0, i32 0
  %arrayidx18 = getelementptr inbounds [5 x i32], ptr %x17, i32 0,↩
    i32 2
  %8 = load i32, ptr %arrayidx18, align 4, !tbaa !7
  %x19 = getelementptr inbounds %struct.ascon_state_t, ptr %state,↩
    i32 0, i32 0
  %arrayidx20 = getelementptr inbounds [5 x i32], ptr %x19, i32 0,↩
    i32 3
  %9 = load i32, ptr %arrayidx20, align 4, !tbaa !7
  %x21 = getelementptr inbounds %struct.ascon_state_t, ptr %state,↩
    i32 0, i32 0
  %arrayidx22 = getelementptr inbounds [5 x i32], ptr %x21, i32 0,↩
    i32 4
  %10 = load i32, ptr %arrayidx22, align 4, !tbaa !7
  %not = xor i32 %6, -1
  %not23 = xor i32 %7, -1
  %not24 = xor i32 %8, -1
  %not25 = xor i32 %9, -1
  %not26 = xor i32 %10, -1
  %x27 = getelementptr inbounds %struct.ascon_state_t, ptr %state,↩
    i32 0, i32 0
  %arrayidx28 = getelementptr inbounds [5 x i32], ptr %x27, i32 0,↩
    i32 1
  %11 = load i32, ptr %arrayidx28, align 4, !tbaa !7
  %and = and i32 %not, %11
  %x29 = getelementptr inbounds %struct.ascon_state_t, ptr %state,↩
    i32 0, i32 0
  %arrayidx30 = getelementptr inbounds [5 x i32], ptr %x29, i32 0,↩
    i32 2
  %12 = load i32, ptr %arrayidx30, align 4, !tbaa !7
  %and31 = and i32 %not23, %12
  %x32 = getelementptr inbounds %struct.ascon_state_t, ptr %state,↩
    i32 0, i32 0
  %arrayidx33 = getelementptr inbounds [5 x i32], ptr %x32, i32 0,↩
    i32 3
  %13 = load i32, ptr %arrayidx33, align 4, !tbaa !7
  %and34 = and i32 %not24, %13
  %x35 = getelementptr inbounds %struct.ascon_state_t, ptr %state,↩
    i32 0, i32 0
  %arrayidx36 = getelementptr inbounds [5 x i32], ptr %x35, i32 0,↩
    i32 4
  %14 = load i32, ptr %arrayidx36, align 4, !tbaa !7
  %and37 = and i32 %not25, %14
  %x38 = getelementptr inbounds %struct.ascon_state_t, ptr %state,↩
    i32 0, i32 0
  %arrayidx39 = getelementptr inbounds [5 x i32], ptr %x38, i32 0,↩
    i32 0
  %15 = load i32, ptr %arrayidx39, align 4, !tbaa !7
  %and40 = and i32 %not26, %15
```

Code 3.11 : Redundant Instructions in LLVM IR Before EarlyCSEPass

```
  %6 = load i32, ptr %state, align 4, !tbaa !7
  %7 = load i32, ptr %arrayidx9, align 4, !tbaa !7
  %8 = load i32, ptr %arrayidx4, align 4, !tbaa !7
  %9 = load i32, ptr %arrayidx, align 4, !tbaa !7
  %not = xor i32 %6, -1
  %not23 = xor i32 %7, -1
  %not24 = xor i32 %xor12, -1
  %not25 = xor i32 %8, -1
  %not26 = xor i32 %9, -1
  %and = and i32 %not, %7
  %and31 = and i32 %not23, %xor12
  %and34 = and i32 %not24, %8
  %and37 = and i32 %not25, %9
  %and40 = and i32 %not26, %6
```

Code 3.12 : Optimized LLVM IR After EarlyCSEPass

Register-based operations increased because redundant load operations from the same pointers are removed. For example, in Code 3.11 to obtain "%and", register operation result "%not" and loaded value "%11" are used. In the output of EarlyCSE, Code 3.12, we can see that instead of reloading to register the loaded register is used, "%7" in this case.

According to the statistics obtained from the "opt" tool of LLVM:

19 early-cse - Number of instructions Common Subexpression Eliminated

7 early-cse - Number of load instructions Common Subexpression Eliminated

35 early-cse - Number of instructions simplified or Dead Code Eliminated

3.2.3.4 Optimize Global Variables - GlobalOpt

This pass aims to optimize global variables and transforms them to constants if necessary. This pass did not significantly change the IR. It only added an attribute to the function, "local_unnamed_addr" meaning that the address of the function is not significant in the module.

```
define dso_local void @sbox(ptr noundef %state) local_unnamed_addr↩
    #0 {
```

Code 3.13 : "local_unnamed_addr" Attribute Added to LLVM IR After GlobalOpt

3.2.3.5 Combine Redundant Instructions - InstCombinePass

Combines redundant instructions and canonicalizes them. Canonicalization is the form in which a single way of commutability is preferred. For example, if a binary operator has a constant operand it is moved to the right. Canonic instructions can then be used by other passes which can assume the instructions to be in the canonic form [19].

```
%0 = load i32, ptr %arrayidx, align 4, !tbaa !7
%1 = load i32, ptr %state, align 4, !tbaa !7
%xor = xor i32 %1, %0
store i32 %xor, ptr %state, align 4, !tbaa !7
%arrayidx4 = getelementptr inbounds [5 x i32], ptr %state, i32 0, i32 3
%2 = load i32, ptr %arrayidx4, align 4, !tbaa !7
%3 = load i32, ptr %arrayidx, align 4, !tbaa !7
%xor7 = xor i32 %3, %2
```

Code 3.14 **:** Redundant Load Instruction in LLVM IR Before InstCombine

```
%arrayidx = getelementptr inbounds [5 x i32], ptr %state, i32 0, i32 4
%0 = load i32, ptr %arrayidx, align 4, !tbaa !7
%1 = load i32, ptr %state, align 4, !tbaa !7
%xor = xor i32 %1, %0
store i32 %xor, ptr %state, align 4, !tbaa !7
%arrayidx4 = getelementptr inbounds [5 x i32], ptr %state, i32 0, i32 3
%2 = load i32, ptr %arrayidx4, align 4, !tbaa !7
%xor7 = xor i32 %0, %2
```

Code 3.15 **:** Removed Load Instruction in LLVM IR After InstCombine

According to the statistics obtained from the "opt" tool of LLVM:

5 aa - Number of NoAlias results

109 assume-queries - Number of Queries into an assume assume bundles

10 basicaa - Number of times a GEP is decomposed

11 instcombine - Number of insts combined

1 instcombine - Number of expansions

2 instcombine - Number of instruction combining iterations performed

3.2.3.6 Early Common Subexpression Elimination - 2nd Run of EarlyCSEPass

Similar to the previous EarlyCSE run in Section 3.2.3.3, load instructions to registers are reused in the subsequent instructions.

```
%8 = load i32, ptr %arrayidx9, align 4, !tbaa !7
%xor46 = xor i32 %8, %and34
store i32 %xor46, ptr %arrayidx9, align 4, !tbaa !7
%9 = load i32, ptr %arrayidx11, align 4, !tbaa !7
%xor49 = xor i32 %9, %and37
store i32 %xor49, ptr %arrayidx11, align 4, !tbaa !7
%10 = load i32, ptr %arrayidx4, align 4, !tbaa !7
%xor52 = xor i32 %10, %and40
store i32 %xor52, ptr %arrayidx4, align 4, !tbaa !7
```

```
%11 = load i32, ptr %arrayidx, align 4, !tbaa !7
%xor55 = xor i32 %11, %and
store i32 %xor55, ptr %arrayidx, align 4, !tbaa !7
%12 = load i32, ptr %state, align 4, !tbaa !7
%13 = load i32, ptr %arrayidx9, align 4, !tbaa !7
%xor60 = xor i32 %13, %12
```

Code 3.16 : Redundant Load Instructions in LLVM IR Before 2nd EarlyCSEPass

```
%xor46 = xor i32 %3, %and34
store i32 %xor46, ptr %arrayidx9, align 4, !tbaa !7
%xor49 = xor i32 %xor12, %and37
store i32 %xor49, ptr %arrayidx11, align 4, !tbaa !7
%xor52 = xor i32 %2, %and40
store i32 %xor52, ptr %arrayidx4, align 4, !tbaa !7
%xor55 = xor i32 %xor7, %and
store i32 %xor55, ptr %arrayidx, align 4, !tbaa !7
%xor60 = xor i32 %xor46, %xor43
```

Code 3.17 : Removed Load Instructions in LLVM IR After 2nd EarlyCSEPass

3.2.3.7 Combine Redundant Instructions - 2nd Run of InstCombinePass

```
%arrayidx = getelementptr inbounds [5 x i32], ptr %state, i32 0,↩
  i32 4
%0 = load i32, ptr %arrayidx, align 4, !tbaa !7
%arrayidx4 = getelementptr inbounds [5 x i32], ptr %state, i32 ↩
 0, i32 3
%2 = load i32, ptr %arrayidx4, align 4, !tbaa !7
%xor7 = xor i32 %0, %2
%not25 = xor i32 %2, -1
%and37 = and i32 %xor7, %not25
```

Code 3.18 : XOR Instruction in LLVM IR Before InstCombine

```
%arrayidx = getelementptr inbounds [5 x i32], ptr %state, i32 0,↩
  i32 4
%0 = load i32, ptr %arrayidx, align 4, !tbaa !7
%arrayidx4 = getelementptr inbounds [5 x i32], ptr %state, i32 ↩
 0, i32 3
%2 = load i32, ptr %arrayidx4, align 4, !tbaa !7
%not25 = xor i32 %2, -1
%and37 = and i32 %0, %not25
```

Code 3.19 : XOR Instruction in LLVM IR After InstCombine

The first algebraic optimization in the process can be observed in this example. In Code 3.18, to obtain "%and37" the boolean operations of the following must be performed:

$$(\%0 \oplus \%2) \wedge (\%2 \oplus -1)$$

It can be shown that a simpler boolean form can be obtained by transitioning equivalent boolean equations:

$$(\%0 \oplus \%2) \land \neg \%2$$

$$((\%0 \land \neg \%2) \lor (\neg \%0 \land \%2)) \land \neg \%2$$

$$(\%0 \land \neg \%2 \land \neg \%2) \lor (\neg \%0 \land \%2 \land \neg \%2)$$

$$(\%0 \land \neg \%2) \lor (0)$$

$$\%0 \land (\%2 \oplus -1)$$

In Code 3.19, the redundant operation $\%0 \oplus \%2$ is removed and the result is:

$$\%0 \land (\%2 \oplus -1)$$

According to the statistics obtained from the "opt" tool of LLVM:

60 assume-queries - Number of Queries into an assume assume bundles

5 instcombine - Number of insts combined

1 instcombine - Number of expansions

2 instcombine - Number of instruction combining iterations performed

12 instsimplify - Number of reassociations

3.2.3.8 Reassociate Expressions - ReassociatePass

Reassociates associative expressions, to promote better constant propagation and simplify expression graph to reduce instruction count. It implements an algorithm where the constants have the least rank and the rank increases with the expression reverse post-order traversal.

```
  %and34 = and i32 %2, %not24
  %and37 = and i32 %0, %not25
  %and40 = and i32 %xor, %not26
  %xor43 = xor i32 %xor, %and31
```

Code 3.20 : Instructions in LLVM IR Before ReassociatePass

```
  %and34 = and i32 %not24, %2
  %and37 = and i32 %not25, %0
  %and40 = and i32 %not26, %xor
  %xor43 = xor i32 %and31, %xor
```

Code 3.21 : Instructions with Reassociated Arguments in LLVM IR After ReassociatePass

A basic glance at the debug output of the pass gives more idea about how the reassociation works.

```
Calculated Rank[state] = 3
Combine negations for:   %xor = xor i32 %0, %1
LINEARIZE:   %xor = xor i32 %0, %1
OPERAND:   %0 = load i32, ptr %arrayidx, align 4, !tbaa !7 (1)
ADD USES LEAF:   %0 = load i32, ptr %arrayidx, align 4, !tbaa !7 (1)
OPERAND:   %1 = load i32, ptr %state, align 4, !tbaa !7 (1)
ADD LEAF:   %1 = load i32, ptr %state, align 4, !tbaa !7 (1)
RAIn:   xor i32 [ %0, #262145] [ %1, #262146]
RAOut:  xor i32 [ %1, #262146] [ %0, #262145]
RA:   %xor = xor i32 %0, %1
TO:   %xor = xor i32 %1, %0
Combine negations for:   %xor7 = xor i32 %0, %2
LINEARIZE:   %xor7 = xor i32 %0, %2
OPERAND:   %0 = load i32, ptr %arrayidx, align 4, !tbaa !7 (1)
ADD USES LEAF:   %0 = load i32, ptr %arrayidx, align 4, !tbaa !7 (1)
OPERAND:   %2 = load i32, ptr %arrayidx4, align 4, !tbaa !7 (1)
ADD USES LEAF:   %2 = load i32, ptr %arrayidx4, align 4, !tbaa !7 (1)
RAIn:   xor i32 [ %0, #262145] [ %2, #262148]
RAOut:  xor i32 [ %2, #262148] [ %0, #262145]
RA:   %xor7 = xor i32 %0, %2
TO:   %xor7 = xor i32 %2, %0
```

The debug output deals with the beginning of the function which is given below.

```
  %0 = load i32, ptr %arrayidx, align 4, !tbaa !7
  %1 = load i32, ptr %state, align 4, !tbaa !7
  %xor = xor i32 %1, %0
  store i32 %xor, ptr %state, align 4, !tbaa !7
  %arrayidx4 = getelementptr inbounds [5 x i32], ptr %state, i32 0, i32 3
  %2 = load i32, ptr %arrayidx4, align 4, !tbaa !7
  %xor7 = xor i32 %0, %2
```

Code 3.22 : Instructions in LLVM IR Before ReassociatePass

The pass computed that the reassociation would result in the instruction "%xor = xor i32 %1, %0" which is already how the IR is so it is not changed. However, as it can be seen in Code 3.21, "%xor7 = xor i32 %2, %0" replaced its alternative representation as the rank of "%0" is less than "%2".

According to the statistics obtained from the "opt" tool of LLVM:

16 reassociate - Number of insts reassociated

3.2.3.9 Combine Redundant Instructions - 2nd Run of InstCombinePass

In this last run of InstCombinePass instruction count is not changed. Some reassociations by the previous pass are reversed.

```
%and34 = and i32 %not24, %2
%and37 = and i32 %not25, %0
%and40 = and i32 %not26, %xor
%xor43 = xor i32 %and31, %xor
```

Code 3.23 : Instructions in LLVM IR Before the 3rd InstCombinePass

```
%and34 = and i32 %2, %not24
%and37 = and i32 %0, %not25
%and40 = and i32 %xor, %not26
%xor43 = xor i32 %and31, %xor
```

Code 3.24 : Instructions with Reassociated Arguments in LLVM IR After InstCombinePass

Though it may seem wasteful, it is a common theme in LLVM that some transformations may be done and be completely reversed by another pass.

According to the statistics obtained from the "opt" tool of LLVM:

60 assume-queries - Number of Queries into an assume assume bundles

7 instcombine - Number of insts combined

2 instcombine - Number of instruction combining iterations performed

12 instsimplify - Number of reassociations

3.2.3.10 Dead Store Elimination - DSEPass

Dead code in the Dead Code Elimination pass refers to the variables in any point of the program which are not used in the future. DCE does not eliminate control flow and store instructions, for this reason Dead Store Elimination pass is used to simplify the store instructions of the given program.

Trivial dead stores are eliminated. As the load operations are optimized, most of the store instructions become dead meaning that they do not affect the flow in any way.

Similar to Early CSE pass in Section 3.2.3.3, DSE pass relies on Memory SSA analysis.

```
store i32 %xor43, ptr %state, align 4, !tbaa !7
%xor46 = xor i32 %and34, %3
store i32 %xor46, ptr %arrayidx9, align 4, !tbaa !7
%xor49 = xor i32 %xor12, %and37
store i32 %xor49, ptr %arrayidx11, align 4, !tbaa !7
%xor52 = xor i32 %and40, %2
```

```
  store i32 %xor52, ptr %arrayidx4, align 4, !tbaa !7
```

Code 3.25 : Redundant Store Instructions in LLVM IR Before DSEPass

```
  %xor46 = xor i32 %and34, %3
  %xor49 = xor i32 %xor12, %and37
  %xor52 = xor i32 %and40, %2
```

Code 3.26 : Removed Store Instructions in LLVM IR After DSEPass

To see how the DSE works we can observe two dead Store's and a killer Store, unnecessary code is stripped away.

```
define dso_local void @sbox(ptr noundef %state) local_unnamed_addr #0 {
  store i32 %xor, ptr %state, align 4, !tbaa !7
  store i32 %xor43, ptr %state, align 4, !tbaa !7
  store i32 %xor65, ptr %state, align 4, !tbaa !7
```

Code 3.27 : Store Instructions to the Same Address in LLVM IR Before DSEPass

Here is the debug output of the DSE pass.

```
Trying to eliminate MemoryDefs killed by 4 = MemoryDef(3) (
store i32 %xor43, ptr %state)
  trying to get dominating access
   visiting 3 = MemoryDef(2)->liveOnEntry (store i32 %xor12, ptr %arrayidx11)
   visiting 2 = MemoryDef(1)->liveOnEntry (store i32 %xor7, ptr %arrayidx)
   visiting 1 = MemoryDef(liveOnEntry) (store i32 %xor, ptr %state)
  Checking for reads of 1 = MemoryDef(liveOnEntry) (store i32 %xor, ptr %state)
   4 = MemoryDef(3)->1 (  store i32 %xor43, ptr %state)
    ... skipping killing def/dom access
   2 = MemoryDef(1)->liveOnEntry (store i32 %xor7, ptr %arrayidx)
   3 = MemoryDef(2)->liveOnEntry (store i32 %xor12, ptr %arrayidx11)
 Checking if we can kill 1 = MemoryDef(liveOnEntry) (store i32 %xor, ptr %state)
DSE: Remove Dead Store:
  DEAD:   store i32 %xor, ptr %state
  KILLER:   store i32 %xor43, ptr %state
  trying to get dominating access
   visiting 0 = MemoryDef(liveOnEntry)
   ...  found LiveOnEntryDef
  finished walk
  .
  .
  .
  Trying to eliminate MemoryDefs killed by 10 = MemoryDef(9) (
  store i32 %xor65, ptr %state)
  trying to get dominating access
   visiting 9 = MemoryDef(8) (store i32 %xor60, ptr %arrayidx9)
   visiting 8 = MemoryDef(7) (store i32 %xor55, ptr %arrayidx)
   visiting 7 = MemoryDef(6)->liveOnEntry (store i32 %xor52, ptr %arrayidx4)
   visiting 6 = MemoryDef(4) (store i32 %xor49, ptr %arrayidx11)
   visiting 4 = MemoryDef(liveOnEntry) (store i32 %xor43, ptr %state)
  Checking for reads of 4 = MemoryDef(liveOnEntry) (store i32 %xor43, ptr %state)
   10 = MemoryDef(9)->4 (store i32 %xor65, ptr %state)
    ... skipping killing def/dom access
   6 = MemoryDef(4) (store i32 %xor49, ptr %arrayidx11)
   9 = MemoryDef(8) (store i32 %xor60, ptr %arrayidx9)
   7 = MemoryDef(6)->liveOnEntry (store i32 %xor52, ptr %arrayidx4)
   8 = MemoryDef(7) (store i32 %xor55, ptr %arrayidx)
 Checking if we can kill 4 = MemoryDef(liveOnEntry) (
 store i32 %xor43, ptr %state)
DSE: Remove Dead Store:
  DEAD:   store i32 %xor43, ptr %state
  KILLER:   store i32 %xor65, ptr %state
  trying to get dominating access
   visiting 0 = MemoryDef(liveOnEntry)
   ...  found LiveOnEntryDef
  finished walk
```

It can be observed that when the DSE encounters a Store instruction to the same address as a previous instruction, kills the previous instruction. The last Store instruction survives DSE.

According to the statistics obtained from the "opt" tool of LLVM:

14 aa - Number of MustAlias results

62 aa - Number of NoAlias results

30 basicaa - Number of times a GEP is decomposed

29 dse - Number iterations check for reads in getDomMemoryDef

0 dse - Number of other instrs removed

7 dse - Number of stores deleted

7 dse - Number of times a valid candidate is returned from getDomMemoryDef

5 dse - Number of stores remaining after DSE

1 ir - Number of renumberings across all blocks

71 memory-builtins - Number of arguments with unsolved size and offset

3.2.3.11 Post-Order Function Attributes Pass - PostOrderFunctionAttrsPass

This pass is similar to the InferFunctionAttrsPass in Section 3.2.3.1. It does not change the IR, adds metadata to it for the other passes.

```
; Function Attrs: nounwind uwtable
```

```
define dso_local void @sbox(ptr noundef %state) local_unnamed_addr #0 {
```

Code 3.28 : Function Attributes After PostOrderFunctionAttrsPass

```
; Function Attrs: mustprogress nofree norecurse nosync nounwind willreturn memory(argmem: readwrite) uwtable
define dso_local void @sbox(ptr nocapture noundef %state) local_unnamed_addr #0 {
```

Code 3.29 : Function Attributes Before PostOrderFunctionAttrsPass

The added attributes signal that the function does not deallocate memory, does not recurse by calling itself, never raises an exception, will continue execution at the end according to the call stack, and may read or write any memory.

In the end of optimization passes the IR at Code 3.1 will be generated.

3.2.4 Clang Optimization Levels

Clang can be invoked with optimization levels deciding which optimization passes are going to be run. The optimizations can target speed or code size. Speed optimizing options range from "-O1" to "-O3". "-O2" enables most of the optimizations. "-O3" enables optimizations that can increase the compile time and generate larger code. The main code optimizing options are "-Os" and "-Oz". "-Os" is similar to "-O2" but runs extra optimizations to reduce code size. "-Oz" runs more code-reducing optimizations compared to "-Os" and is similar to "-O2" again [20]. Caution must be taken as when no arguments are given to Clang, at the time of writing, Clang uses the "-O0" optimization level. Implementing pattern matching on unoptimized LLVM IR is not feasible for several reasons. Firstly, the IR is more sensitive to changes in the front-end. Changing the code style in front-end can cause CodeGen to produce a slightly different IR which makes it less predictable. Secondly, the code size can be too large with redundant code which makes pattern matching large instructions cumbersome. We recommend using "-O2" or "-Os" optimization levels while developing instruction selection patterns.

LLVM optimizer can be performed with opt [options] [filename] command [21].

3.3 Stages of the LLVM RISC-V Back-end

LLVM RISC-V back-end is responsible for compiling optimized IR down to RISC-V assembly or object code. LLVM back-end consists of libraries for the code generation steps [22].

3.3.1 Instruction Selection

SelectionDAG is the instruction selector of LLVM back-ends which is responsible for selecting the appropriate RISC-V instructions for a given intermediate representation instruction. It takes the target-independent LLVM code as input and generates the target-dependent DAG of instructions. SelectionDAG is the most important part of the compiler for us since we will be dealing with adding new instructions for the RISC-V back-end.

3.3.1.1 SelectionDAG construction

After IR generating is done, SelectionDAG gets the optimized IR and converts it into target-independent SelectionDAG representation. SelectionDAG consists of SelectionDAG nodes which are created by SelectionDAGBuilder class. SelectionDAGIsel visits all the IR instructions and uses the SelectionDAGBuilder class. The relevant instruction method requests an SDNode to the DAG and assigns its opcode. Every SDNode(SelectionDAG node) has an opcode for the operation it represents. SDNodes have multiple values to return as the result. SDValues(SelectionDAG value) hold the information to determine which number to return. SelectionDAGBuilder class reshapes the linear IR input to a SelectionDAG tree form. At the and of the construction SelectionDAG is a target-independent and illegal DAG.

3.3.1.2 SelectionDAG legalization

SelectionDAG is a target-dependent representation after the construction stage of the instruction selection. Before creating a target-specific code, SelectionDAG checks if the DAG is legal because the constructed DAG may include incompatible instructions and data types to the target architecture [23]. Selectiondag legalization refers to the process of transforming the SelectionDAG to the constraints and requirements of the target architecture. Legalization may involve adding, removing, splitting or merging the nodes which provide to match the register file and instruction set of the target architecture [24]. SelectionDAG legalization also ensures that the data type of the target architecture is compatible with the target architecture by truncating or promoting the data types. For example, if SelectionDAG includes i32 data type nodes targeted to an i64 architecture, SDlegalizer promotes the i32 nodes into i64 data type. For every target architecture type, IselLowering.cpp files are responsible for legalizing the SelectionDAG. SDlegalizer legalizes the illegal DAG into a supported form and ensures that the generated code is efficient and compatible with the target architecture. An example of legalizing the SelectionDAG by truncating the i64 data type DAG into i32 data type target architecture is shown in code 3.30. IR code includes i64 data type variables however, target architecture supports only i32 data type. Before the legalization(figure 3.2) DAG is not converted to the target data type yet and it needed to become i64 compatible. It can be seen that after legalization(figure 3.3), i32 nodes are truncated and there are no i64 nodes in the DAG.

```
define i64 @test(i64 %a, i64 %b, i64 %c) {
  %mul = mul nsw i64 %a, %b
  %add = add i64 %mul, %c
  ret i64 %add
  }

```

Code 3.30 **:** Intermediate Representation code input for legalization example

3.3.1.3 SelectionDAG optimization

SelectionDAG is in an optimizable form after legalization because legalization phase may create unnecessary DAG nodes and the reducable nodes are not combined yet. SelectionDAG optimizer minimizes the DAG nodes before creating the target-specific instructions.

3.3.1.4 SelectionDAG target-dependent instruction selection

At the last phase of the instruction selection, SelectionDAG selects the suitable instructions for the target architecture. SelectionDAG uses the relevant TableGen target description (.td) files to match the patterns and creates the target-specific instructions.

3.3.2 Scheduling and Formation

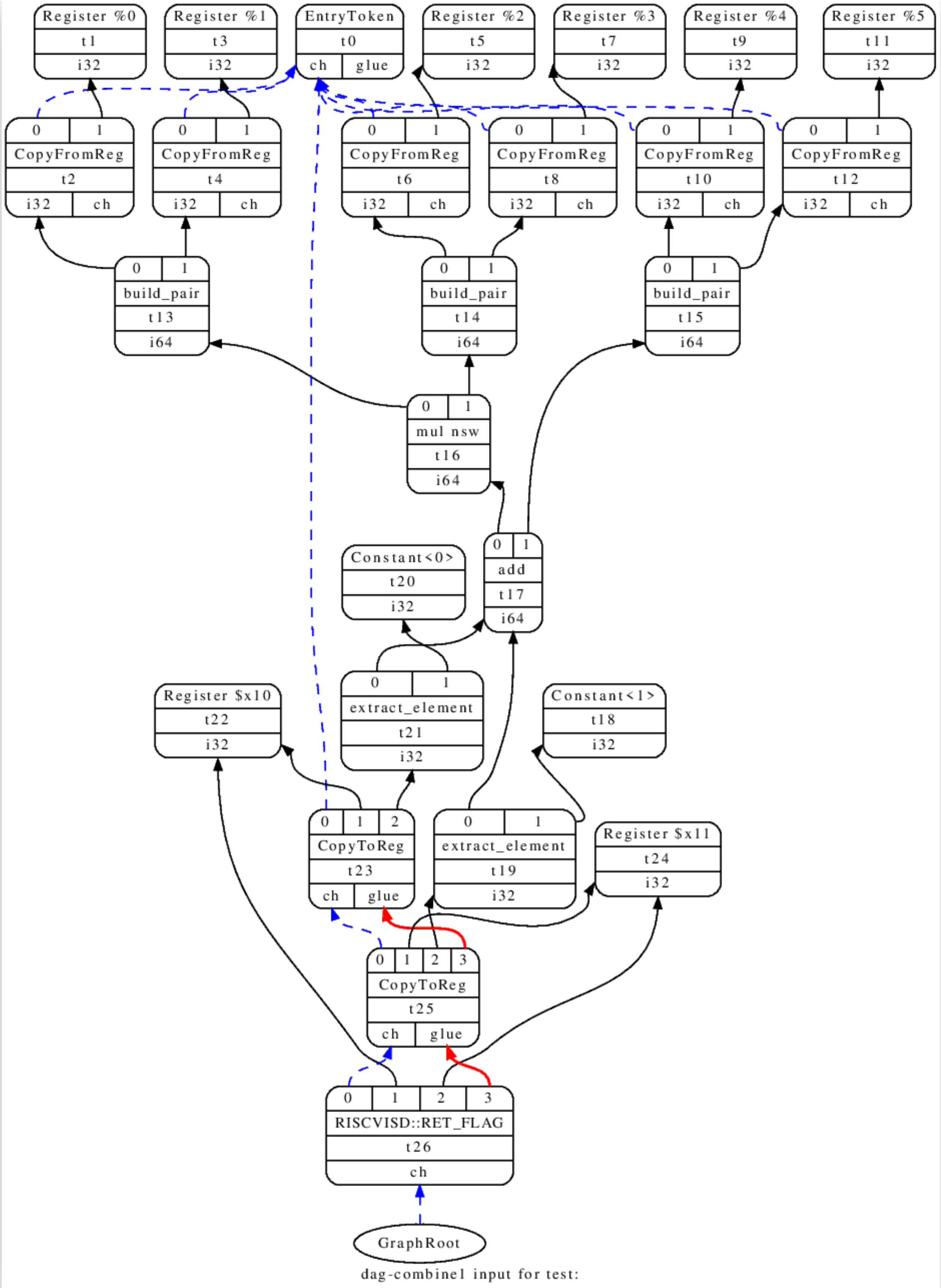


Figure 3.2 : DAG diagram before the legalization stage

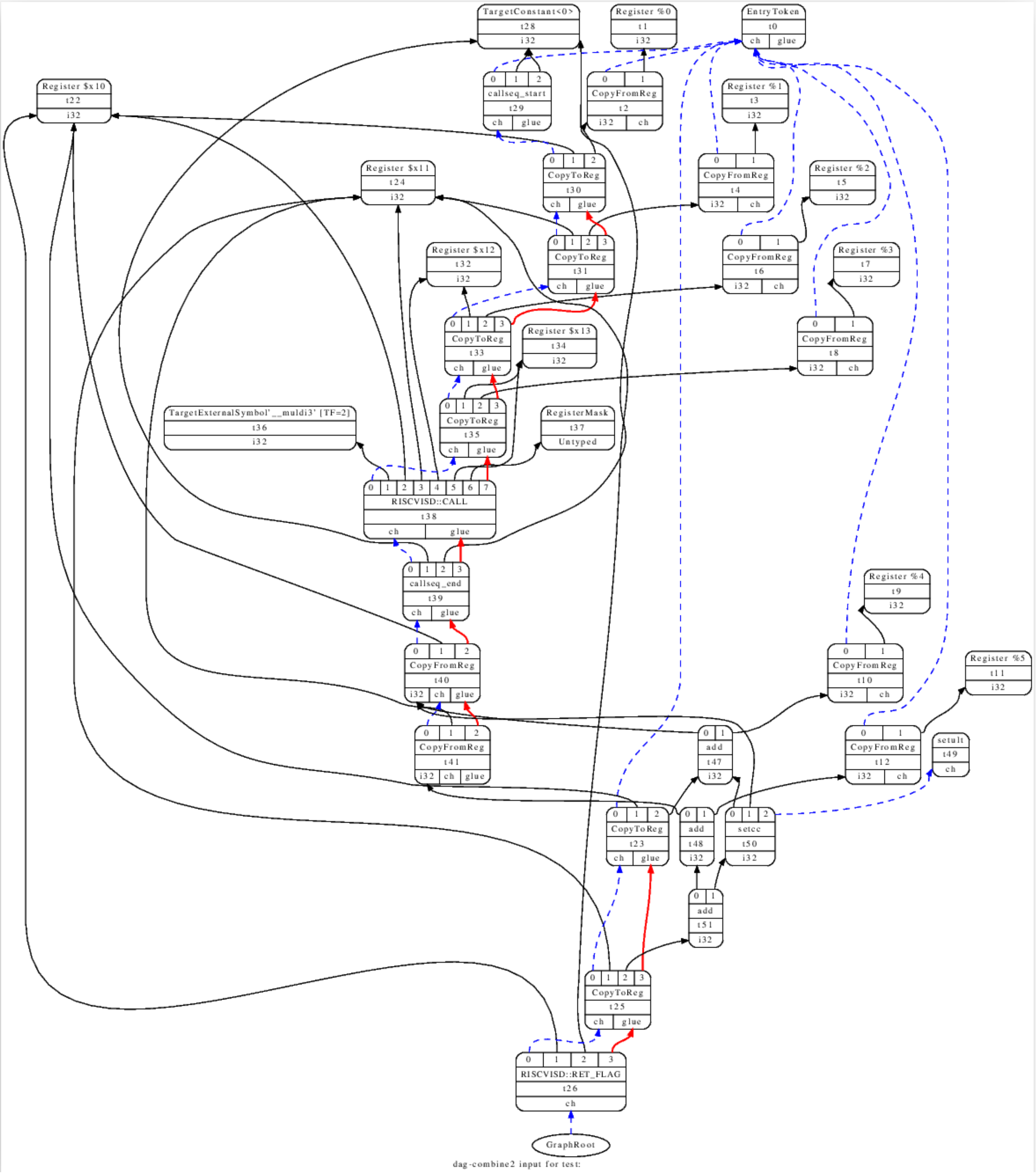


Figure 3.3 : DAG diagram after the legalization stage

Scheduling is the phase of assigning an order to the DAG form of RISC-V instructions. The formation phase is responsible for converting the DAG into a list of machine instructions.

3.3.3 SSA-based Machine Code Optimizations

LLVM uses static single assignment (SSA) based optimizations before register allocation. SSA optimizations ensure that each variable is assigned and defined only once before it is used.

3.3.4 Register Allocation

The register the RISC-V back-end is responsible for assigning physical registers to virtual registers in the IR. Each target has a specific register count and order. Register allocator maps the registers by taking the RISC-V architecture registers into account. It uses the relevant TargetRegisterInfo, and MachineOperand classes. Register allocation will play a key role in our project while we manipulate the implementation of the selected instructions.

3.3.5 Prolog/Epilog Code Insertion

Prolog and epilog code insertion is another optimization phase that is responsible for frame-pointer elimination and stack packing.

3.3.6 Code Emission

The code emission stage is responsible for lowering the code generator abstractions down to the MC layer abstractions. It takes the assembly as input and creates the final RISC-V machine codes.

3.3.7 Linking

LLD is the LLVM linker library that is responsible for combining multiple object files into a single executable file. LLD is invoked after the code emission and generates a file by resolving symbol references, adjusting addresses, and performing other tasks as necessary.

4. RISC-V

In this project, our target is a 32-bit RISC-V core. RISC stands for reduced instruction set computer and RISC-V is an open standard Instruction Set Architecture. [25] It is structured as a small base instruction set architecture and it has different additional extensions. The base instruction set architecture (ISA) is straightforward, rendering RISC-V appropriate for academic and learning purposes, yet extensive enough to function as a cost-effective and energy-efficient ISA for embedded systems. [26] Being open-source and royalty-free is another significant advantage and is an important reason why RISC-V is being commonly used. RISC-V was developed by Prof. Krste Asanović and his students Andrew Waterman and Yunsup Lee. They started working on this project in 2010 as a part of as part of the Parallel Computing Laboratory which was in UC Berkeley. Par Lab was sponsored by several companies and worked on advancing parallel computing.

4.1 RISC-V ISA

The Instruction Set Architecture (ISA) constitutes a part of a computer's abstract design that defines how the CPU is managed by the software. It serves as a bridge between the software and hardware, defining the processor's abilities and the methods by which it performs tasks. Its level in the system can be seen in figure 4.1.

There are different base integer variants of RISC-V such as RV32I, RV64I, and RV128I. These have address spaces of 32, 64, and 128 bits respectively. [27] In our project, we are interested in 32-bits. RISC-V has 32 general-purpose registers. Their application binary interface names and purposes can be seen in figure 4.2. Also in the figure, we can see a different set of registers. These registers are used for floating point operations. Their ABI and purposes are also given.

4.2 RISC-V Base Instructions

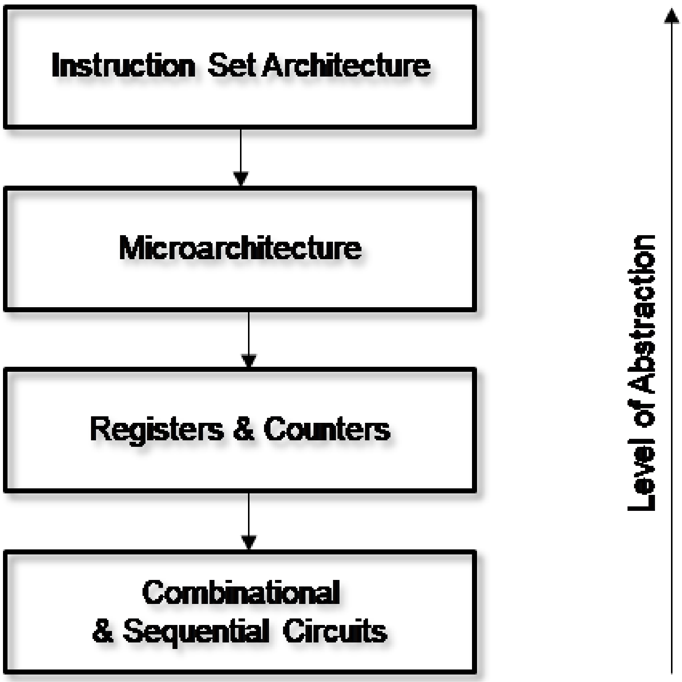
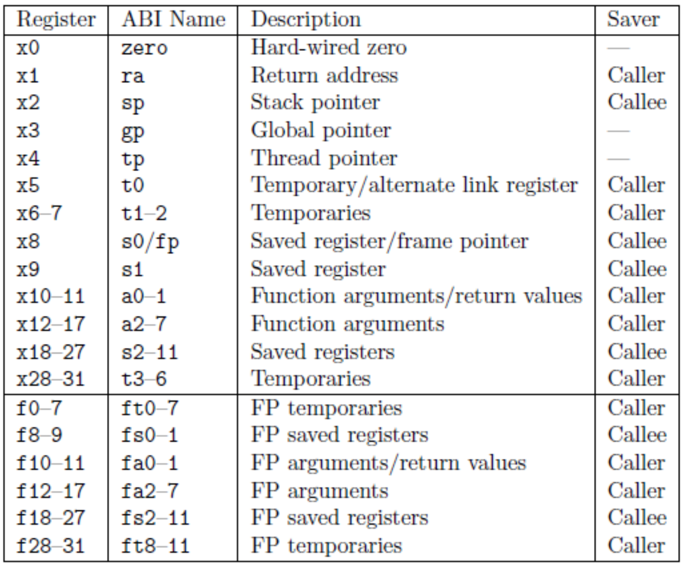


Figure 4.1 : Level of abstraction diagram [1]



Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

Figure 4.2 : RISC-V registers [2]

There are four basic instruction formats in the base RV32I ISA. These are named R, I, S, U and all of these are 32-bits in length. There are two more additional variants named B and J as well. [3] These formats are given in figure 4.3.

Rs1 and Rs2 are the source registers and Rd is the destination register. An immediate value can also be used in some of the formats. The base instructions of the RV32I are given in figure 4.4. By inspecting their formats, we can see which type the

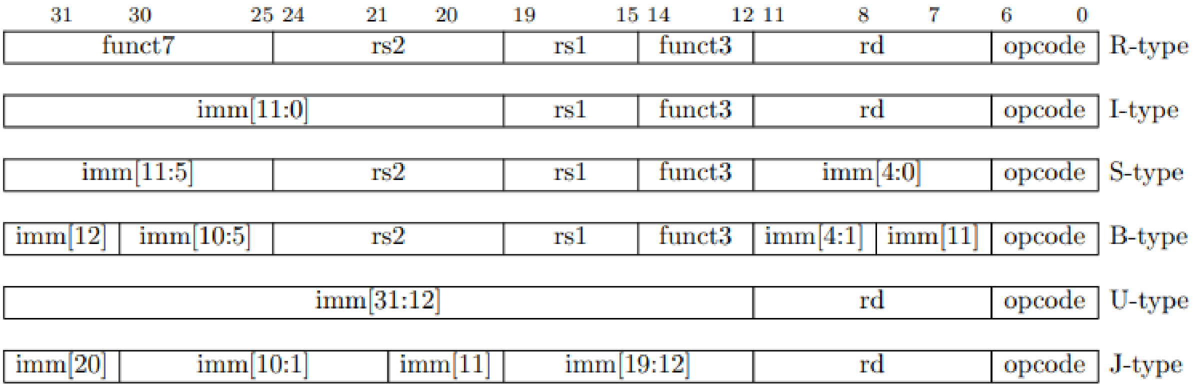


Figure 4.3 : RISC-V base instruction formats [3]

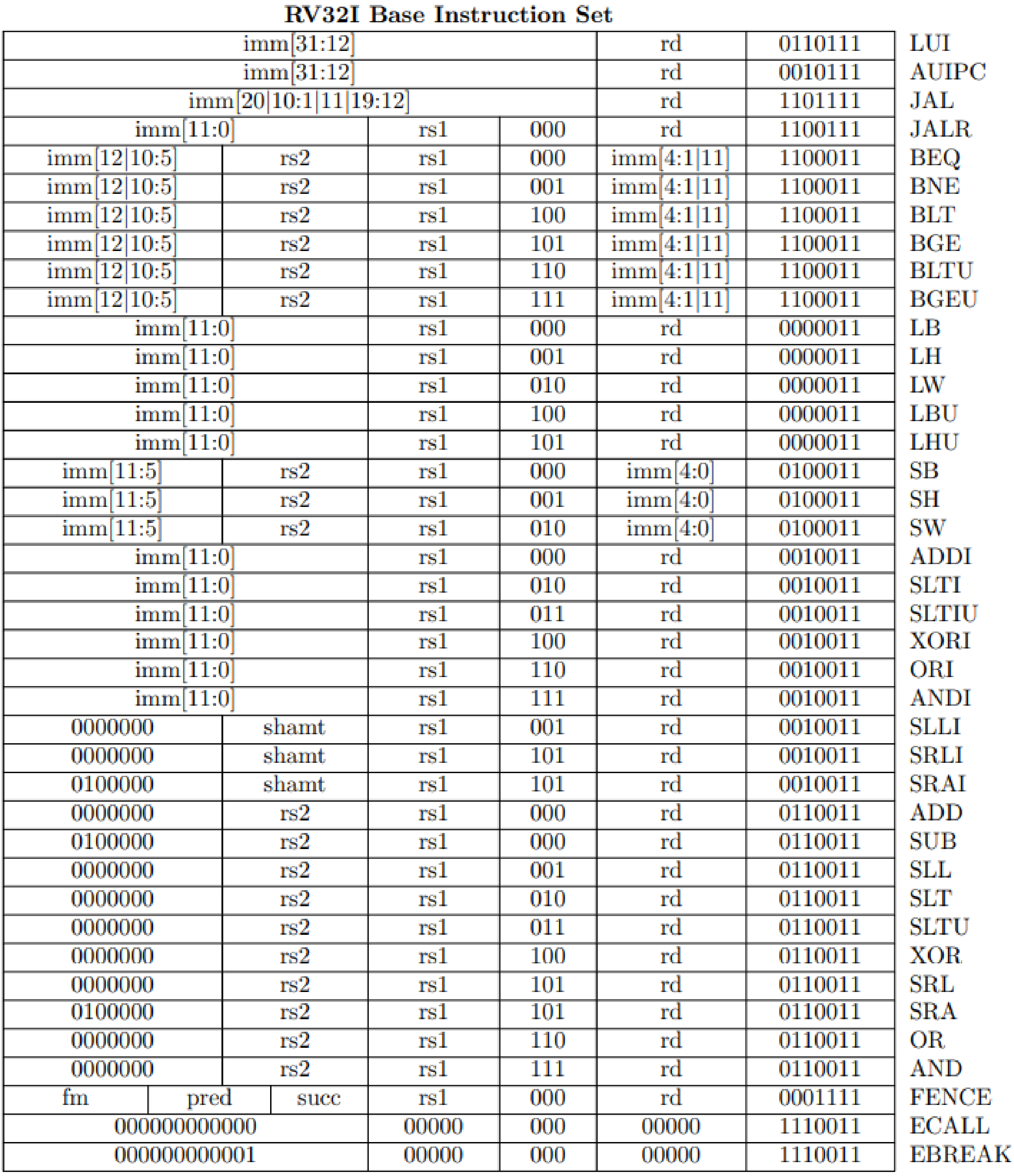
instructions belong to. For example, the ADDI instruction is an I-type instruction and XOR is an R-type instruction.

4.3 RISC-V Extensions

We had mentioned the extensions previously. Abbreviations for these extensions and what they are for are given in figure 4.5.

Thanks to these instruction extensions, more specific tasks can be implemented since we are not limited by the base instructions. Among these, the bit manipulation (B) standard extension is quite important for our project. This extension contains many instructions that operate on the bits. These extensions are also divided into several groups according to common properties. These subgroups and their purposes can be seen in figure 4.6.

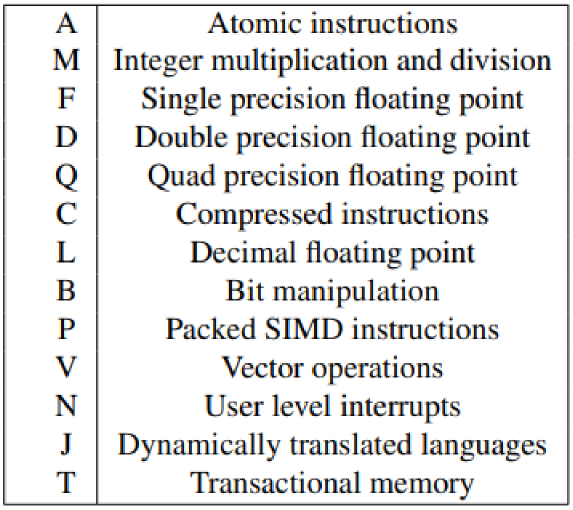
Grouping these instructions according to how commonly they are used and the similarity of the operations that they perform makes them more organized and easier to work on. Some of these extensions are compatible with RV64 only. The compatibilities and the groups the instructions belong to are given in figure 4.7.



RV32I Base Instruction Set

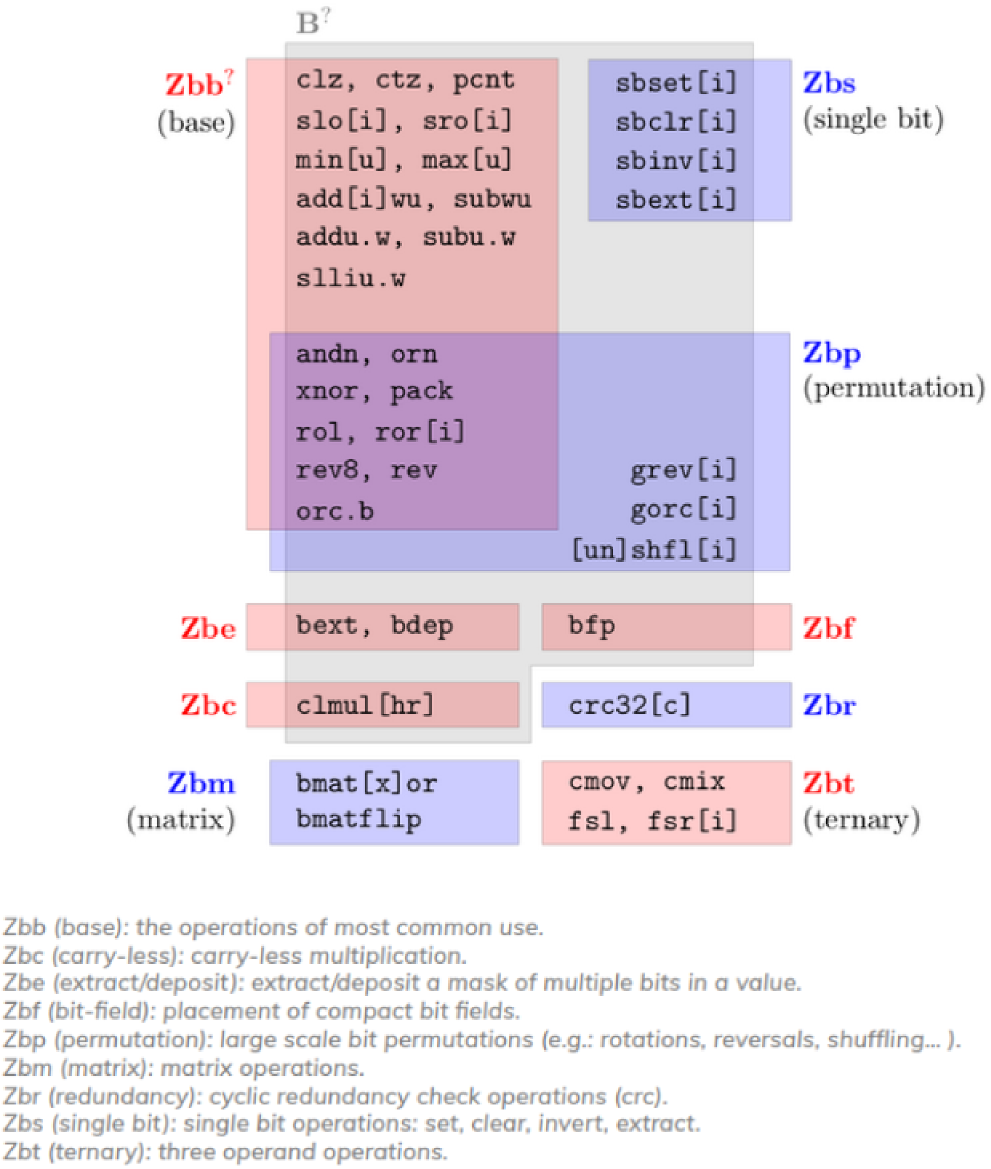
imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
fm pred	succ	rs1	000	rd	0001111	FENCE
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK

Figure 4.4 : RV32I base instruction set [3]



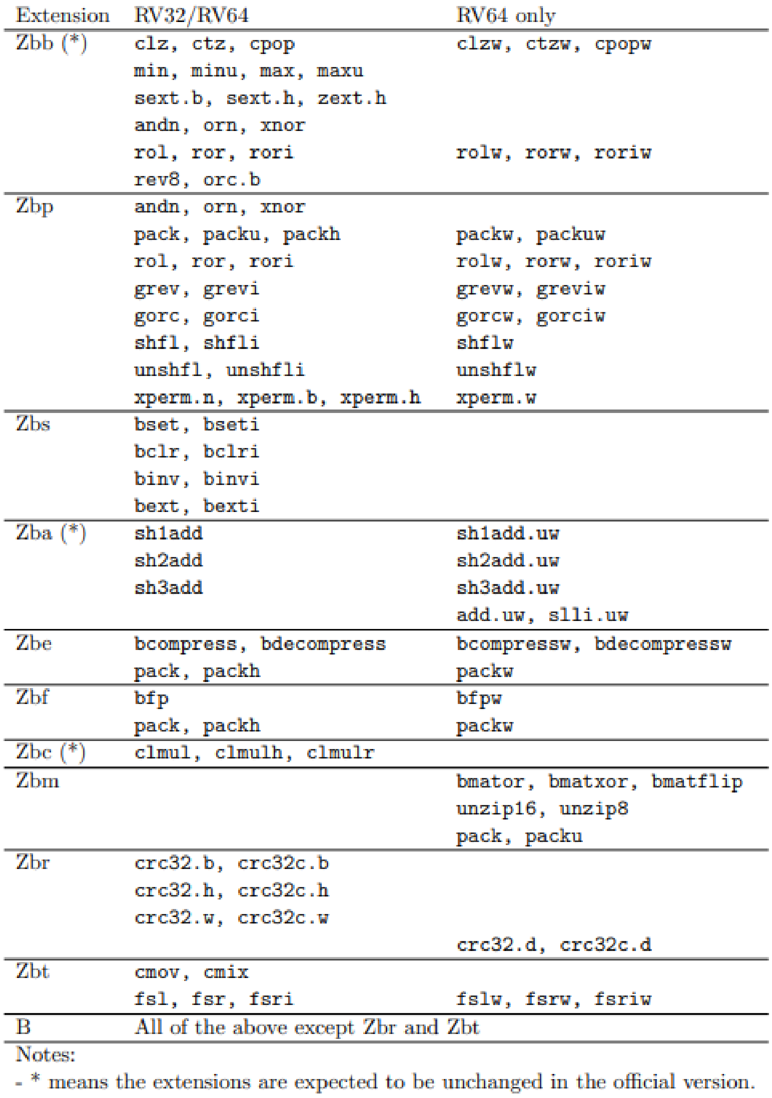
A	Atomic instructions
M	Integer multiplication and division
F	Single precision floating point
D	Double precision floating point
Q	Quad precision floating point
C	Compressed instructions
L	Decimal floating point
B	Bit manipulation
P	Packed SIMD instructions
V	Vector operations
N	User level interrupts
J	Dynamically translated languages
T	Transactional memory

Figure 4.5 : List of standard extension sets [4]



Zbb (base): the operations of most common use.
Zbc (carry-less): carry-less multiplication.
Zbe (extract/deposit): extract/deposit a mask of multiple bits in a value.
Zbf (bit-field): placement of compact bit fields.
Zbp (permutation): large scale bit permutations (e.g.: rotations, reversals, shuffling...).
Zbm (matrix): matrix operations.
Zbr (redundancy): cyclic redundancy check operations (crc).
Zbs (single bit): single bit operations: set, clear, invert, extract.
Zbt (ternary): three operand operations.

Figure 4.6 : Bit manipulation extension groupings [5]



Extension	RV32/RV64	RV64 only
Zbb (*)	clz, ctz, cpop	clzw, ctzw, cpopw
	min, minu, max, maxu	
	sext.b, sext.h, zext.h	
	andn, orn, xnor	
	rol, ror, rori	rolw, rorw, roriw
	rev8, orc.b	
Zbp	andn, orn, xnor	
	pack, packu, packh	packw, packuw
	rol, ror, rori	rolw, rorw, roriw
	grev, grevi	grevw, greviw
	gorc, gorci	gorcw, gorciw
	shfl, shfli	shflw
	unshfl, unshfli	unshflw
	xperm.n, xperm.b, xperm.h	xperm.w
Zbs	bset, bseti	
	bclr, bclri	
	binv, binvi	
	bext, bexti	
Zba (*)	sh1add	sh1add.uw
	sh2add	sh2add.uw
	sh3add	sh3add.uw
		add.uw, slli.uw
Zbe	bcompress, bdecompress	bcompressw, bdecompressw
	pack, packh	packw
Zbf	bfp	bfpw
	pack, packh	packw
Zbc (*)	clmul, clmulh, clmulr	
Zbm		bmator, bmatxor, bmatflip
		unzip16, unzip8
		pack, packu
Zbr	crc32.b, crc32c.b	
	crc32.h, crc32c.h	
	crc32.w, crc32c.w	
		crc32.d, crc32c.d
Zbt	cmov, cmix	
	fsl, fsr, fsri	fslw, fsrw, fsriw
B	All of the above except Zbr and Zbt	

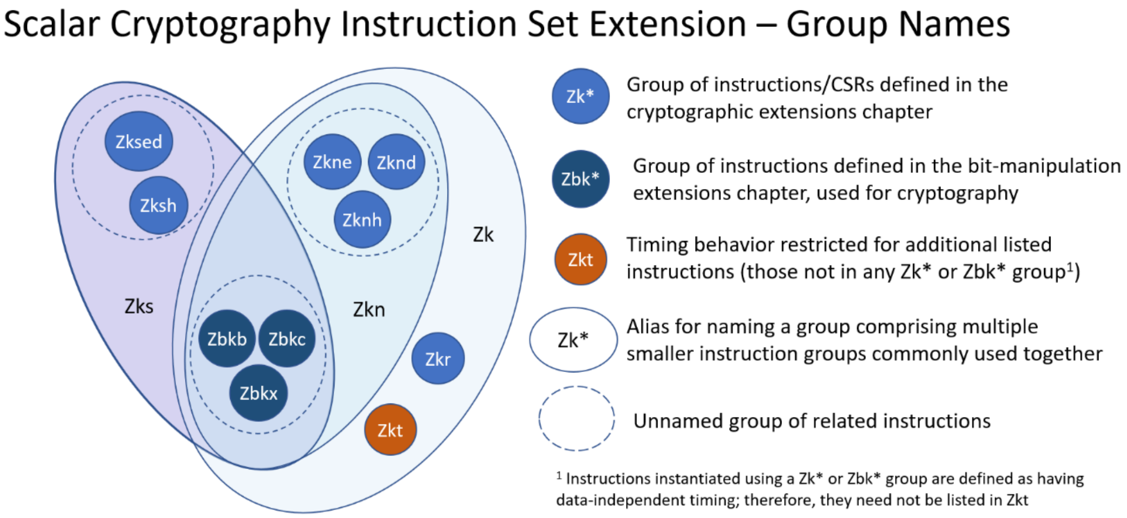
Notes:
- * means the extensions are expected to be unchanged in the official version.

Figure 4.7 : Bit RV32/RV64 compatibilities and groups [6]

To give a clearer image of what bit manipulation (B) instructions do, let's explain a few of them. For example, "CLZ" is an instruction for counting the leading zeros. Its purpose is to find out how many zeros there are before encountering a 1, starting from the most significant bit. Another example is "ORN" instruction. It negates the second operand and performs bitwise or with the first one.
Zba is also a subgroup of the bit manipulation extensions. Shift and add instructions are included in this group and they perform a left shift by 1, 2, or 3 bits since they are commonly used in codes and also because they require only a minimal amount of extra hardware beyond that of a basic adder. This way, lengthening the critical path in implementations can be avoided. For example, SH1ADD is a part of this group and it shifts the operand by 1 and adds 1. [28]
There is also a scalar cryptography instruction set extension for RISC-V. The RISC-V Scalar Cryptography extensions allow cryptographic tasks to be completed more quickly. Furthermore, these extensions significantly reduce the difficulty of implementing fast and secure cryptography in embedded devices and IoT. [29] This instruction set extension is also divided into subgroups according to the purpose and similarity of the instructions. The groups are given in figure 4.8. These groups and their purposes can be explained briefly.

Zbkb contains bit manipulation instructions for cryptography. These are a selection of the bit manipulation extension Zbb that have specific applications in cryptography.
Zbkc contains carry-less multiply instructions.
Zbkx instructions can be useful for implementing s-boxes in constant time.
Zknd contains instructions that help speed up the decryption and key schedule functions of the AES block cipher and Zkne does the same for encryption.
Zknh has some instructions that can help speed up the SHA2 family of cryptographic hash functions.
Zksed contains instructions that speed up the SM4 block cipher.
Zksh instructions help accelerate the SM3 hash function.
Zkr can be useful to seed cryptographic random bit generators. [30]

These extensions are supported by the compiler but pattern matching support for Zbkb and Zbkx is incomplete. Also for Zknd, Zkne, Zknh, Zksed and Zksh, no



Scalar Cryptography Instruction Set Extension – Group Names
Zksed
Zksh
Zks
Zkne
Zknd
Zknh
Zkn
Zk
Zbkb
Zbkc
Zbkx
Zkr
Zkt
Zk*
Group of instructions/CSRs defined in the cryptographic extensions chapter
Zbk*
Group of instructions defined in the bit-manipulation extensions chapter, used for cryptography
Zkt
Timing behavior restricted for additional listed instructions (those not in any Zk* or Zbk* group[1])
Zk*
Alias for naming a group comprising multiple smaller instruction groups commonly used together
Unnamed group of related instructions
[1] Instructions instantiated using a Zk* or Zbk* group are defined as having data-independent timing; therefore, they need not be listed in Zkt

Figure 4.8 : Cryptography extension subgroups [7]

pattern matching exists. Therefore, these instructions can be only used via intrinsic calls or from the assembler. [31]

These instructions are not a part of the base instruction set. Therefore, one cannot perform these operations with a single instruction if one uses the base instructions only. Instruction extensions prove quite useful in these situations. One important thing we need to be careful about is that we should check these standard instruction extensions before we try to implement non-standard extensions by ourselves. These are comprehensive extensions and may already contain the extensions that we want to implement. After making sure that the extension we want is not present, we may try to implement non-standard extensions.

5. ASCON CRYPTHOGRAPHIC ALGORITHM

ASCON (Authenticated Encryption with Associated Data) is a lightweight encryption algorithm that is a family of lightweight authenticated ciphers. ASCON is designed to have both authenticity and confidentiality for transmitted data and it is efficient in terms of both speed and code size. It has a clean and simple design, making it suitable for resource-constrained environments. ASCON was designed by Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer in 2014. It was proposed as a candidate for the lightweight authenticated encryption competition (CAESAR) in 2014 and was selected as one of the finalists.

5.1 ASCON Structure

Ascon is based on Sponge structure that is shown in (figure 5.1). ASCON gets an initial input to start and encyrpt the algorithm. The length of the initial input is 320 bits that consists of five 64-bits words. Initial input includes secret key, initial vector and nonce. Secret key is used for the encrypt and decrypt the transmitted information. Information can be read if the key is known thus it must be kept secret. Initial Vector is a random value to start a iterated process. Nonce increases the protection of the cipher against cryptanalysis techniques.

Concatanated input consists of two parts. The first r bits of the input are the rate bits. The last c = 320 – r bits of the input are called capacity bits. In the initialization stage,

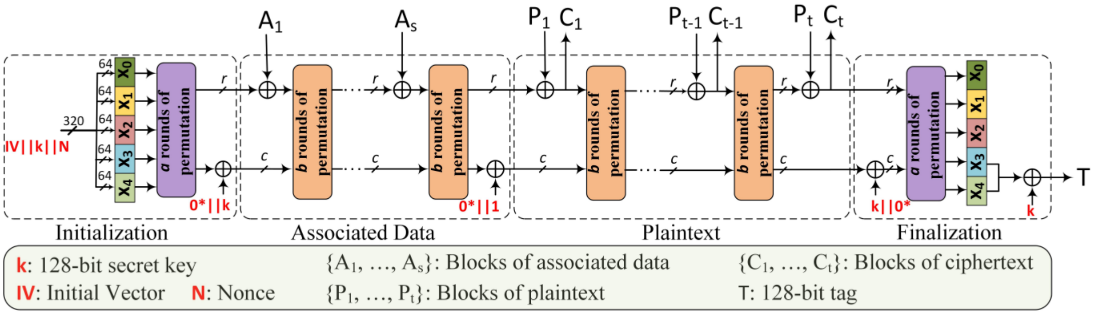


Figure 5.1 : Associated data and plaintext are absorved into the sponge based structure

“a” rounds of permutaton functions are implemented to concatanated input. After permutation, the last 128 bits of the capacity bits are XORed with 128-bit secret key.

At the beggining of the associated data stage, rate bits are XORed with the first block of the associated data then “b” rounds of permutation is implemented to the output. This step is repeated with the previous output and the next block of the associated data until all the blocks are covered. Associated data is absorved into the sponge structure. At the end of the associated stage, capacity bits are XORed with 1’s.

In the Plaintext stage, the plaintext blocks are absorved into the sponge like the associated stage and the ciphertext blocks are obtained. At the beginning of the finalization stage, secret key is XORed with the first 128 bits of the capacity bits. “a” rounds of permutations are implemented and the last 128 bits of the capacity bits are XORed with the secret key. The output of the finalization stage is called the 128-bit tag.

5.2 Permutation Function of the ASCON Algorithm

ASCON’s permutation function consists of a nonlinear substition layer and a linear diffusion layer. The substition layer performs a 5-bit S-box. S-boxes take input as the bits of each five 64-bit words of the concatanated input. 64 S-boxes are performed for every bits of the words in a single permutation function. S-box operations are shown in (figure 5.2). Linear diffusion layer performs the rotations and XORs shown in (figure 5.3) linearly.

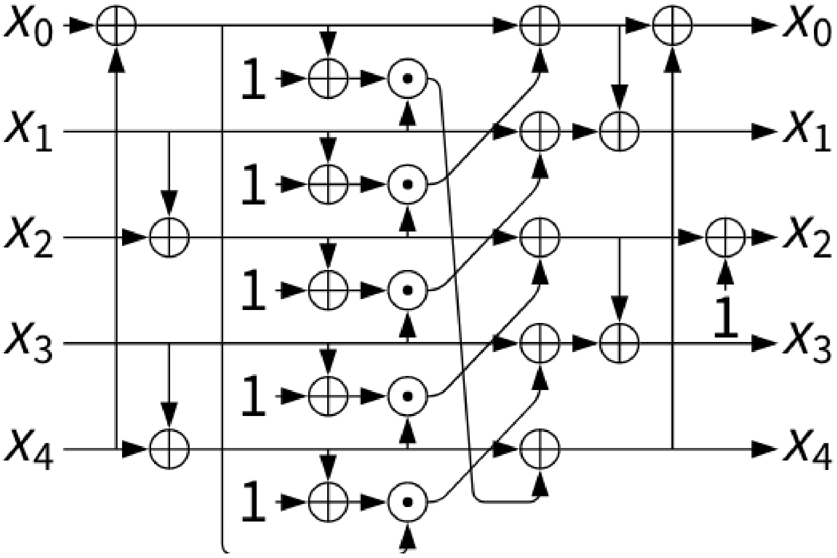


Figure 5.2 : S-box operations

$$
\begin{aligned}
x_0 &\leftarrow \Sigma_0(x_0) = x_0 \oplus (x_0 \ggg 19) \oplus (x_0 \ggg 28)\\
x_1 &\leftarrow \Sigma_1(x_1) = x_1 \oplus (x_1 \ggg 61) \oplus (x_1 \ggg 39)\\
x_2 &\leftarrow \Sigma_2(x_2) = x_2 \oplus (x_2 \ggg \ \ 1) \oplus (x_2 \ggg \ \ 6)\\
x_3 &\leftarrow \Sigma_3(x_3) = x_3 \oplus (x_3 \ggg 10) \oplus (x_3 \ggg 17)\\
x_4 &\leftarrow \Sigma_4(x_4) = x_4 \oplus (x_4 \ggg \ \ 7) \oplus (x_4 \ggg 41)
\end{aligned}
$$

Figure 5.3 : ASCON linear operations

6. PATH OF AN INSTRUCTION

In this chapter, the path of an instruction will be demonstrated and the corresponding DAG input of the most critical phases of SelectionDAG will be shown. We selected the input program as a function that performs multiplication and addition. This was our litmus test code used while adding MLA (Multiply and Add) instruction to the LLVM back-end with TableGen. We explained our addition of MLA instruction thoroughly in Section 7.3.

```c
int a,b,c;
void maddFunc() {
  a = 3;
  b = 103;

  c = 127;
  a = a * b + c;
}
```

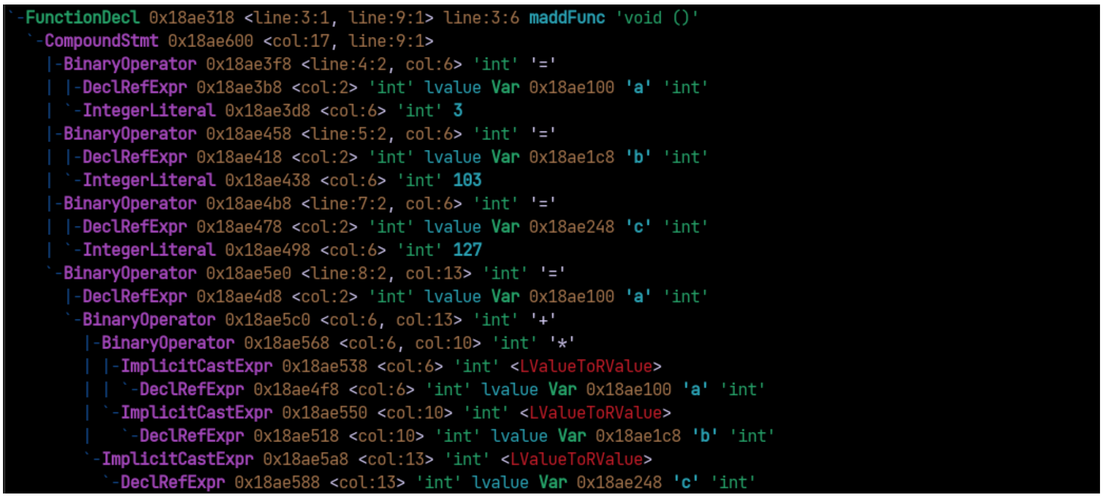
Code 6.1 : madd.c program

6.1 Clang AST

You can see the Abstract Syntax Tree (AST) produced by Clang in Figure 6.1. The AST consists of an expression tree with three levels. At the highest level, there is an expression tree of multiplication between variables ’a’ and ’b’. This expression tree’s result becomes an argument for another expression tree with the addition operator. The second argument at this addition subtree is the variable ’c’. The expression tree at the root has an assignment as an operator. The first argument to this tree is ’a’ and the second argument is the result of multiplication and addition. Figure 6.2 shows the entire expression tree with the operations multiply and add.

6.2 LLVM IR

Clang CodeGen produces LLVM IR with the AST as the input. Figure 6.2 shows the produced LLVM IR. The optimized LLVM IR is the input to SelectionDAG to generate target-specific instructions.



```
`-FunctionDecl 0x18ae318 <line:3:1, line:9:1> line:3:6 maddFunc 'void ()'
  `-CompoundStmt 0x18ae600 <col:17, line:9:1>
    |-BinaryOperator 0x18ae3f8 <line:4:2, col:6> 'int' '='
    | |-DeclRefExpr 0x18ae3b8 <col:2> 'int' lvalue Var 0x18ae100 'a' 'int'
    | `-IntegerLiteral 0x18ae3d8 <col:6> 'int' 3
    |-BinaryOperator 0x18ae458 <line:5:2, col:6> 'int' '='
    | |-DeclRefExpr 0x18ae418 <col:2> 'int' lvalue Var 0x18ae1c8 'b' 'int'
    | `-IntegerLiteral 0x18ae438 <col:6> 'int' 103
    |-BinaryOperator 0x18ae4b8 <line:7:2, col:6> 'int' '='
    | |-DeclRefExpr 0x18ae478 <col:2> 'int' lvalue Var 0x18ae248 'c' 'int'
    | `-IntegerLiteral 0x18ae498 <col:6> 'int' 127
    `-BinaryOperator 0x18ae5e0 <line:8:2, col:13> 'int' '='
      |-DeclRefExpr 0x18ae4d8 <col:2> 'int' lvalue Var 0x18ae100 'a' 'int'
      `-BinaryOperator 0x18ae5c0 <col:6, col:13> 'int' '+'
        |-BinaryOperator 0x18ae568 <col:6, col:10> 'int' '*'
        | |-ImplicitCastExpr 0x18ae538 <col:6> 'int' <LValueToRValue>
        | | `-DeclRefExpr 0x18ae4f8 <col:6> 'int' lvalue Var 0x18ae100 'a' 'int'
        | `-ImplicitCastExpr 0x18ae550 <col:10> 'int' <LValueToRValue>
        |   `-DeclRefExpr 0x18ae518 <col:10> 'int' lvalue Var 0x18ae1c8 'b' 'int'
        `-ImplicitCastExpr 0x18ae5a8 <col:13> 'int' <LValueToRValue>
          `-DeclRefExpr 0x18ae588 <col:13> 'int' lvalue Var 0x18ae248 'c' 'int'
```

Figure 6.1 : AST generated by Clang

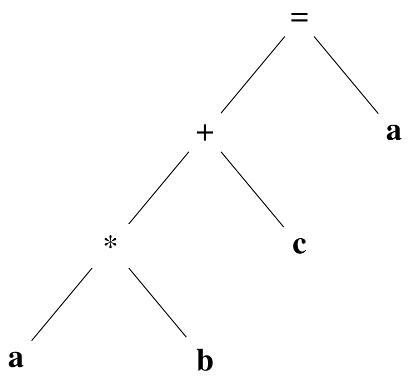


Figure 6.2 : AST of MLA operation

```
define void @maddFunc() {
  store i32 3, i32* @a
  store i32 103, i32* @b
  store i32 127, i32* @c
  %1 = load i32, i32* @a
  %2 = load i32, i32* @b
  %5 = mul nsw i32 %1, %2
  %4 = load i32, i32* @c
  %5 = add nsw i32 %3, %4
  store i32 %5, i32* @a
  ret void
}
```

Code 6.2 **:** LLVM IR file generated at the output of Clang

6.3 SelectionDAG

Input DAGs to SelectionDAG's passes will be demonstrated so on. The following phases will be demonstrated:

1. First Optimization
2. Legalization
3. Second Optimization
4. Instruction Selection
5. Instruction Scheduling
6. Register Allocation

6.3.1 First Optimization Pass

Figure 6.3 shows the DAG before the first optimization pass. It is the direct translation of LLVM IR to DAG form. After optimization, redundant nodes will be removed such as "Constant<0>" node.

Figure 6.4 shows the DAG before legalization. The first optimization took place by removing nodes that do not contribute to the DAG. However, the instructions are not, in LLVM terms, "legal" as these general Machine Instructions do not map directly to every target's instructions.

6.3.2 Instruction Legalization

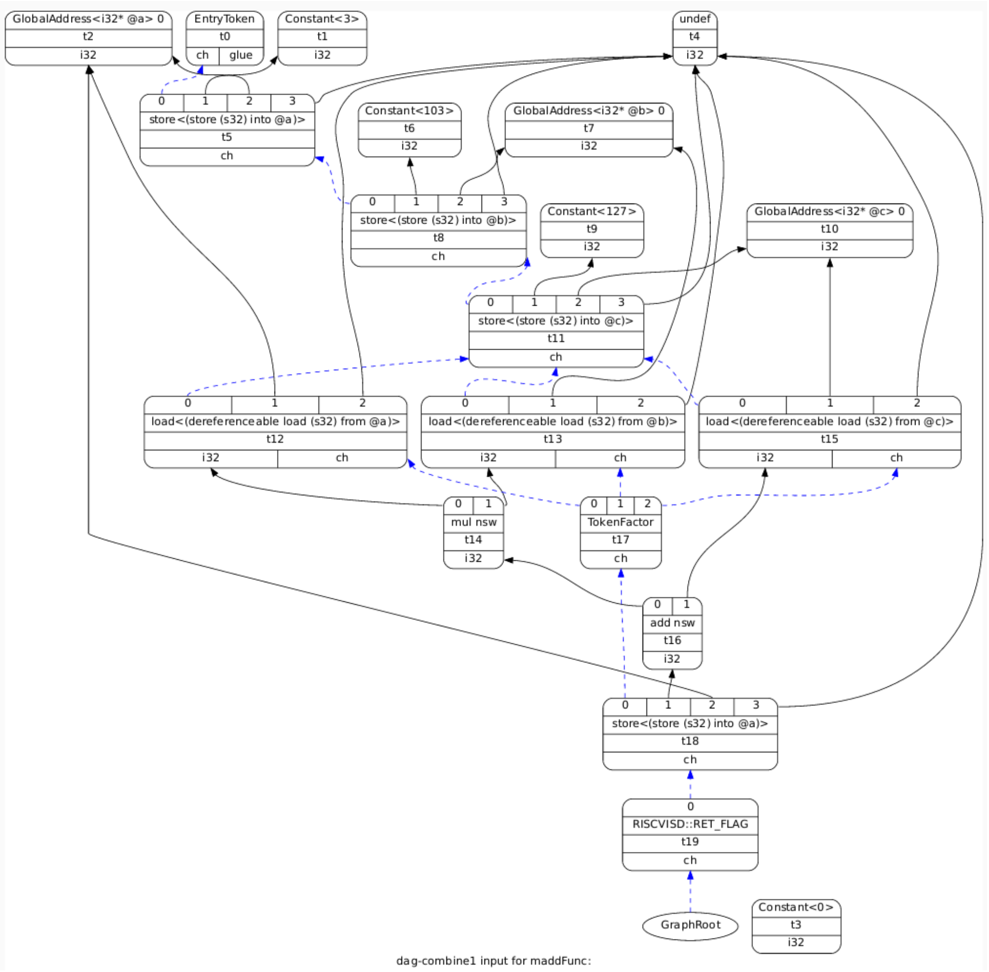


Figure 6.3 : DAG before first optimization pass

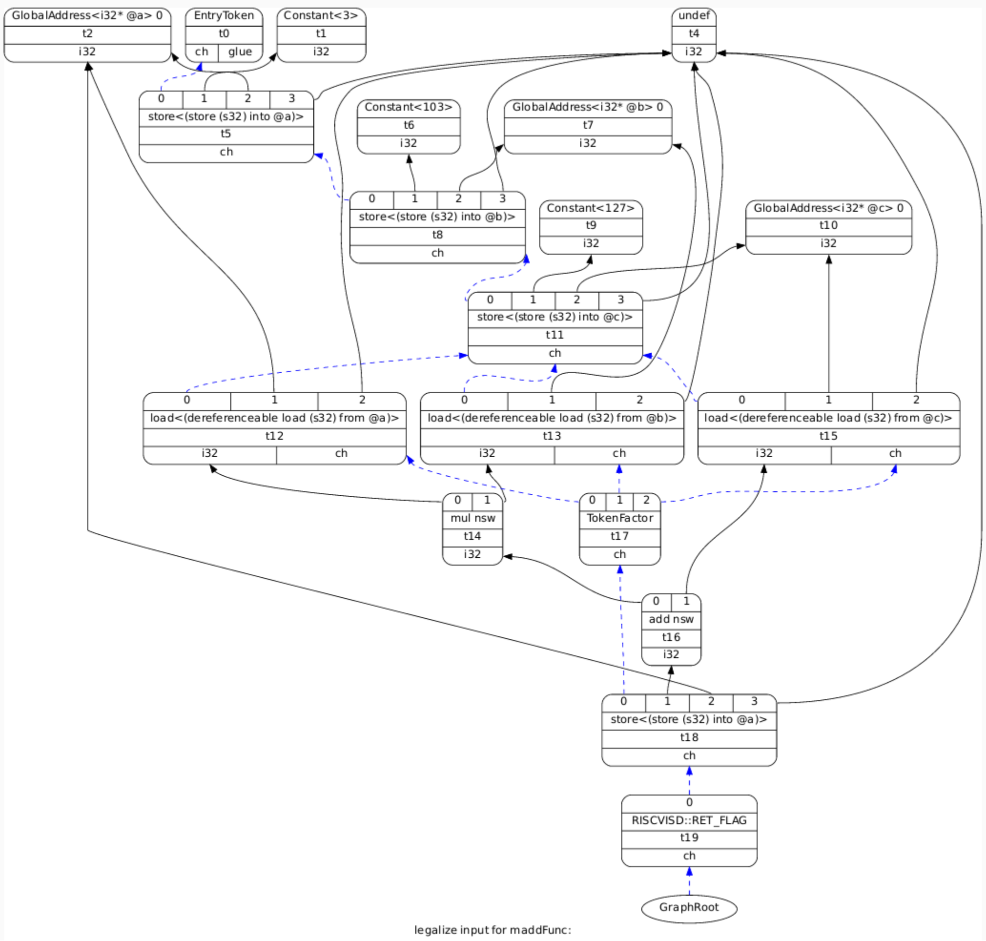


Figure 6.4 : DAG before Legalization

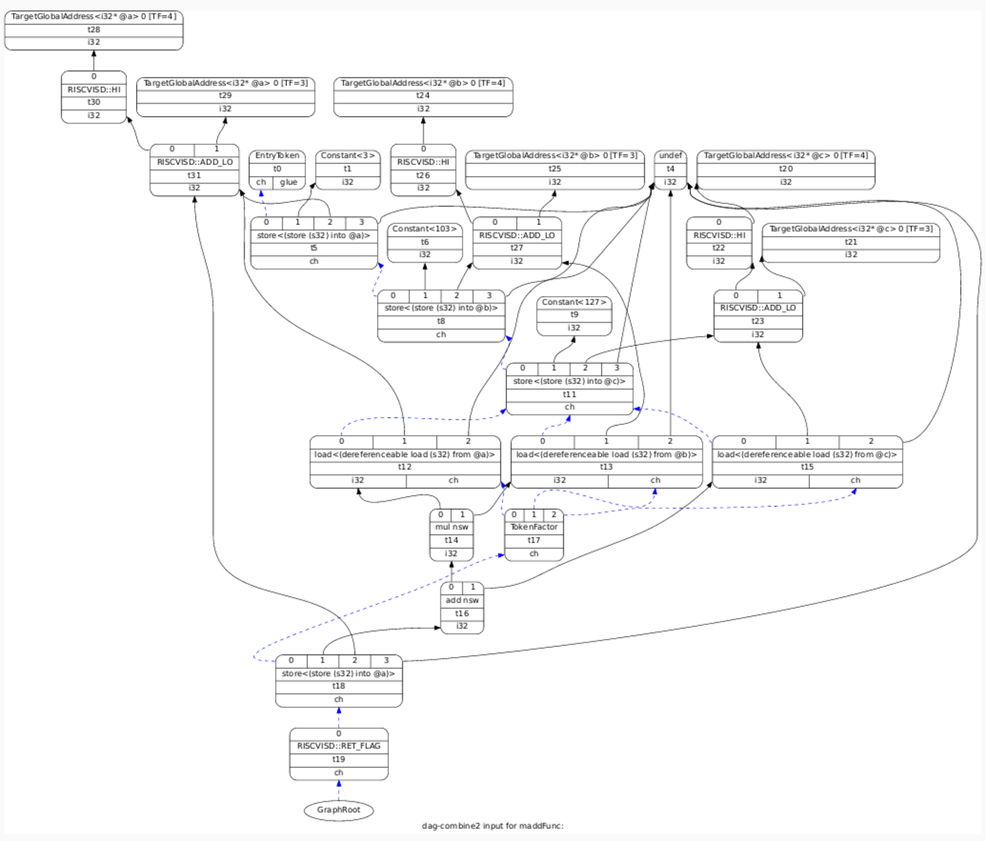

TargetGlobalAddress<i32* @a> 0 [TF=4]
t28
i32
0
RISCVISD::HI
t30
i32
TargetGlobalAddress<i32* @a> 0 [TF=3]
t29
i32
TargetGlobalAddress<i32* @b> 0 [TF=4]
t24
i32
0 1
RISCVISD::ADD_LO
t31
i32
EntryToken
t0
ch
glue
Constant<3>
t1
i32
0
RISCVISD::HI
t26
i32
TargetGlobalAddress<i32* @b> 0 [TF=3]
t25
i32
undef
t4
i32
TargetGlobalAddress<i32* @c> 0 [TF=4]
t20
i32
0 1 2 3
store<(store (s32) into @a)>
t5
ch
Constant<103>
t6
i32
0 1
RISCVISD::ADD_LO
t27
i32
0
RISCVISD::HI
t22
i32
TargetGlobalAddress<i32* @c> 0 [TF=3]
t21
i32
0 1 2 3
store<(store (s32) into @b)>
t8
ch
Constant<127>
t9
i32
0 1
RISCVISD::ADD_LO
t23
i32
0 1 2 3
store<(store (s32) into @c)>
t11
ch
0 1 2
load<(dereferenceable load (s32) from @a)>
t12
i32
ch
0 1 2
load<(dereferenceable load (s32) from @b)>
t13
i32
ch
0 1 2
load<(dereferenceable load (s32) from @c)>
t15
i32
ch
0 1
mul nsw
t14
i32
0 1 2
TokenFactor
t17
ch
0 1
add nsw
t16
i32
0 1 2 3
store<(store (s32) into @a)>
t18
ch
0
RISCVISD::RET_FLAG
t19
ch
GraphRoot
dag-combine2 input for maddFunc:

Figure 6.5 : DAG before the second optimization

Figure 6.5 shows the DAG before the second optimization pass. The DAG is legalized by introducing RISCVISD::ADD_LO and RISCVISD::HI nodes. These SelectionDAG nodes act as flags to give target-specific information to target-independent algorithms. These definitions are introduced at lib/Target/RISCV/RISCVISelLowering.h file [32]. It is the RISCV DAG lowering interface.

According to the interface file, RISCVISD::ADD_LO is meant to add Lo 12 bits from an address and to be replaced by ADDI (Add Immediate) at Instruction Selection. Similarly, RISCVISD::HI is meant to get Hi 20 bits from an address and to be replaced by LUI (Load Upper Immediate). With a legalized DAG the second optimization pass begins.

6.3.3 Second Optimization Pass

Figure 6.6 shows the DAG before the Instruction Selection phase. A comparison of Figure 6.5 and 6.6 indicates that the second optimization did not change the DAG.

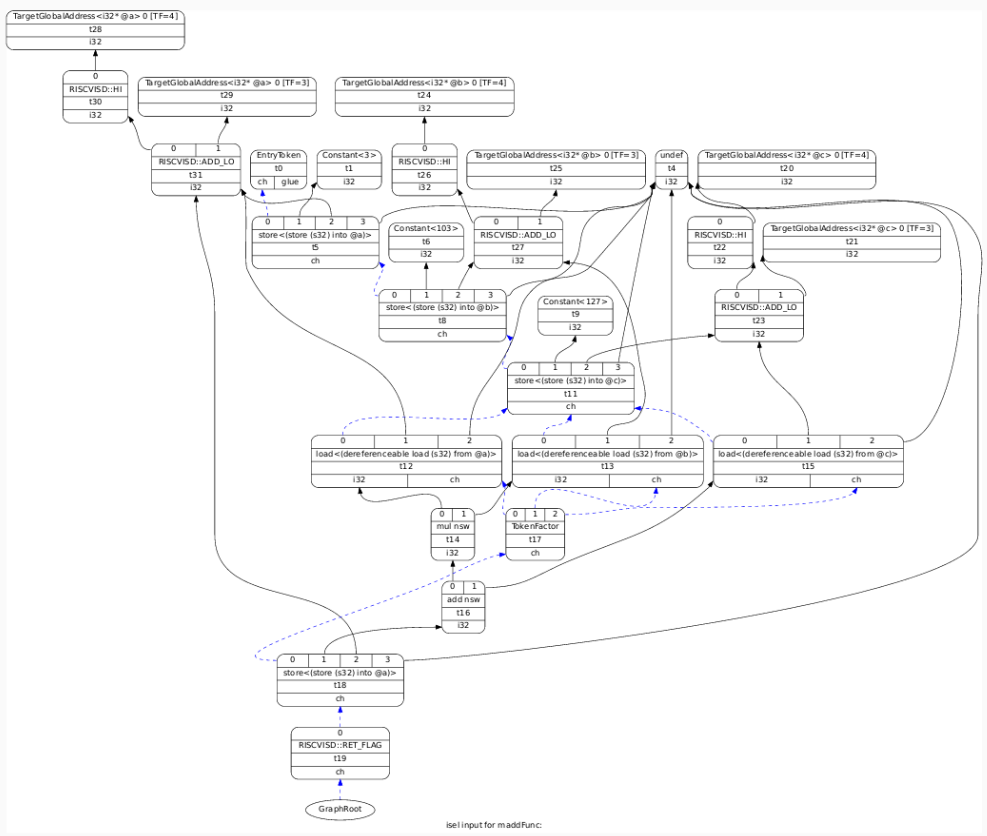

TargetGlobalAddress<i32* @a> 0 [TF=4]
t28
i32
RISCVISD::HI
t30
i32
TargetGlobalAddress<i32* @a> 0 [TF=3]
t29
i32
TargetGlobalAddress<i32* @b> 0 [TF=4]
t24
i32
RISCVISD::ADD_LO
t31
i32
EntryToken
t0
ch
glue
Constant<3>
t1
i32
RISCVISD::HI
t26
i32
TargetGlobalAddress<i32* @b> 0 [TF=3]
t25
i32
undef
t4
i32
TargetGlobalAddress<i32* @c> 0 [TF=4]
t20
i32
store<(store (s32) into @a)>
t5
ch
Constant<103>
t6
i32
RISCVISD::ADD_LO
t27
i32
RISCVISD::HI
t22
i32
TargetGlobalAddress<i32* @c> 0 [TF=3]
t21
i32
store<(store (s32) into @b)>
t8
ch
Constant<127>
t9
i32
RISCVISD::ADD_LO
t23
i32
store<(store (s32) into @c)>
t11
ch
load<(dereferenceable load (s32) from @a)>
t12
i32
ch
load<(dereferenceable load (s32) from @b)>
t13
i32
ch
load<(dereferenceable load (s32) from @c)>
t15
i32
ch
mul nsw
t14
i32
TokenFactor
t17
ch
add nsw
t16
i32
store<(store (s32) into @a)>
t18
ch
RISCVISD::RET_FLAG
t19
ch
GraphRoot
isel input for maddFunc:

Figure 6.6 : DAG before Instruction Selection

This may be due to the reason that the subgraphs including the legalized nodes are not complex enough as the input C code is minimal.

The DAG nodes up until Instruction Selection are instances of SDNode class which are target-independent nodes.

6.3.4 Instruction Selection

Figure 6.7 shows the DAG before the Instruction Scheduling phase. You can see that the instructions are selected according to the RISC-V target. SDNode class nodes are replaced by MachineSDNode class nodes which are target-specific.

RISCVISD nodes are replaced by their counterparts. The general Load and Store instructions are replaced by their type-aware corresponding LW (Load Word) and SW (Store Word) instructions. Most importantly the MLA instruction is selected replacing the subgraph of ’mul’ and ’add’ LLVM instructions.

We defined MLA instruction’s DAG pattern as in the subgraph. The instruction selection phase took it as a reference, detected the pattern inside the global DAG, and

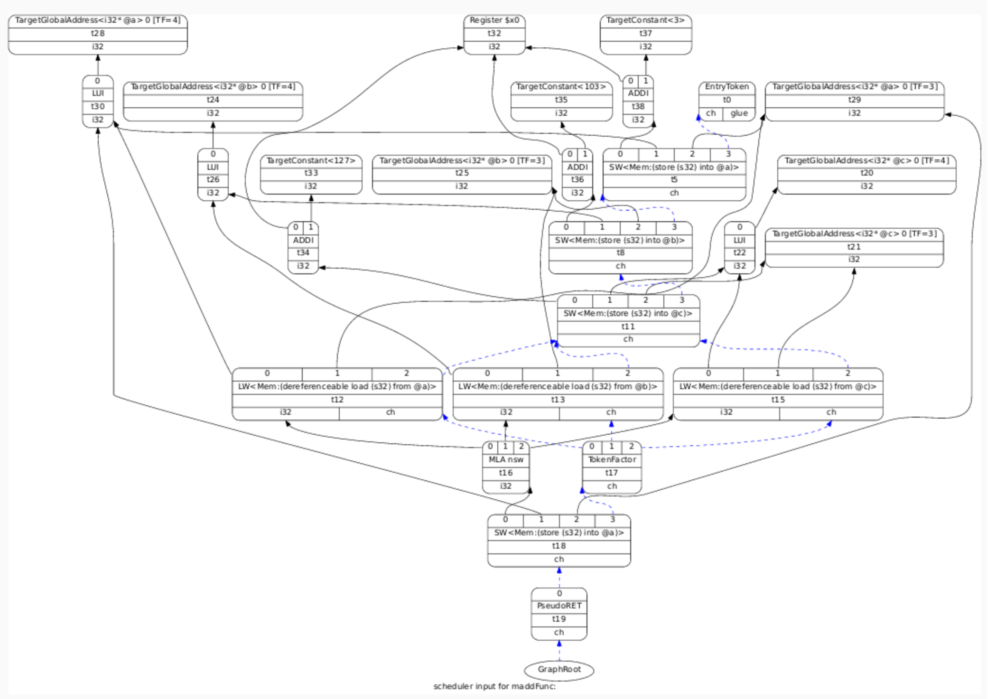

TargetGlobalAddress<i32* @a> 0 [TF=4]
t28
i32
Register $x0
t32
i32
TargetConstant<3>
t37
i32
LUI
t30
i32
TargetGlobalAddress<i32* @b> 0 [TF=4]
t24
i32
TargetConstant<103>
t35
i32
ADDI
t38
i32
EntryToken
t0
ch
glue
TargetGlobalAddress<i32* @a> 0 [TF=3]
t29
i32
LUI
t26
i32
TargetConstant<127>
t33
i32
TargetGlobalAddress<i32* @b> 0 [TF=3]
t25
i32
ADDI
t36
i32
SW<Mem:(store (s32) into @a)>
t5
ch
TargetGlobalAddress<i32* @c> 0 [TF=4]
t20
i32
ADDI
t34
i32
SW<Mem:(store (s32) into @b)>
t8
ch
LUI
t22
i32
TargetGlobalAddress<i32* @c> 0 [TF=3]
t21
i32
SW<Mem:(store (s32) into @c)>
t11
ch
LW<Mem:(dereferenceable load (s32) from @a)>
t12
i32
ch
LW<Mem:(dereferenceable load (s32) from @b)>
t13
i32
ch
LW<Mem:(dereferenceable load (s32) from @c)>
t15
i32
ch
MLA nsw
t16
i32
TokenFactor
t17
ch
SW<Mem:(store (s32) into @a)>
t18
ch
PseudoRET
t19
ch
GraphRoot
scheduler input for maddFunc:

Figure 6.7 : DAG before Instruction Scheduling

used it to place the MLA node. The addition process is explained more thoroughly in Section 7.3.

6.3.5 Instruction Scheduling

The DAG is transformed into a target-specific DAG with the result of legalization and selection phases. However, to generate a linear byte sequence, the DAG must be flattened. The instruction scheduling phase gets the DAG and linearises it according to the dependency graph of nodes. The scheduling dependency can be seen in Figure 6.8.

6.3.6 Machine Instruction in SSA Form

The generated Machine Instruction as a result of scheduling is shown in Figure 6.9. Because register allocation is not yet performed, the instructions are in SSA (Static Single Assignment) form. In SSA form, virtual registers are considered to be infinite unless some specific registers have to be used. In this case, '$x0' is mentioned with ADDI instructions as they are hardwired zero in RISC-V.

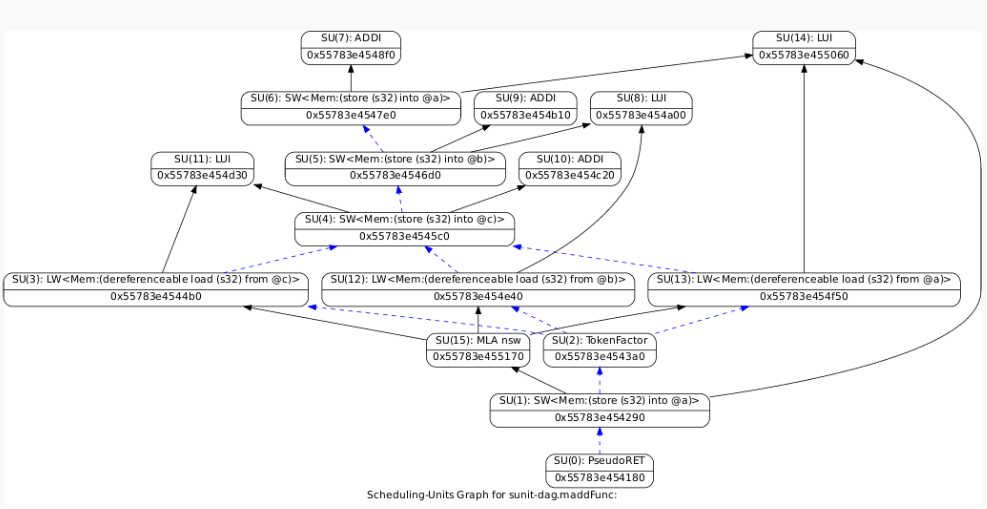
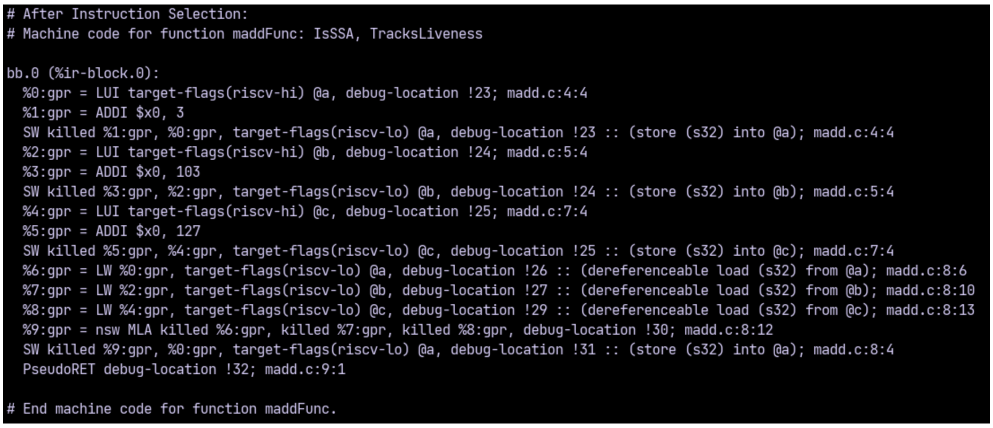

SU(7): ADDI
0x55783e4548f0
SU(14): LUI
0x55783e455060
SU(6): SW<Mem:(store (s32) into @a)>
0x55783e4547e0
SU(9): ADDI
0x55783e454b10
SU(8): LUI
0x55783e454a00
SU(11): LUI
0x55783e454d30
SU(5): SW<Mem:(store (s32) into @b)>
0x55783e4546d0
SU(10): ADDI
0x55783e454c20
SU(4): SW<Mem:(store (s32) into @c)>
0x55783e4545c0
SU(3): LW<Mem:(dereferenceable load (s32) from @c)>
0x55783e4544b0
SU(12): LW<Mem:(dereferenceable load (s32) from @b)>
0x55783e454e40
SU(13): LW<Mem:(dereferenceable load (s32) from @a)>
0x55783e454f50
SU(15): MLA nsw
0x55783e455170
SU(2): TokenFactor
0x55783e4543a0
SU(1): SW<Mem:(store (s32) into @a)>
0x55783e454290
SU(0): PseudoRET
0x55783e454180
Scheduling-Units Graph for sunit-dag.maddFunc:

Figure 6.8 : Scheduling Dependency Graph



```
# After Instruction Selection:
# Machine code for function maddFunc: IsSSA, TracksLiveness

bb.0 (%ir-block.0):
  %0:gpr = LUI target-flags(riscv-hi) @a, debug-location !23; madd.c:4:4
  %1:gpr = ADDI $x0, 3
  SW killed %1:gpr, %0:gpr, target-flags(riscv-lo) @a, debug-location !23 :: (store (s32) into @a); madd.c:4:4
  %2:gpr = LUI target-flags(riscv-hi) @b, debug-location !24; madd.c:5:4
  %3:gpr = ADDI $x0, 103
  SW killed %3:gpr, %2:gpr, target-flags(riscv-lo) @b, debug-location !24 :: (store (s32) into @b); madd.c:5:4
  %4:gpr = LUI target-flags(riscv-hi) @c, debug-location !25; madd.c:7:4
  %5:gpr = ADDI $x0, 127
  SW killed %5:gpr, %4:gpr, target-flags(riscv-lo) @c, debug-location !25 :: (store (s32) into @c); madd.c:7:4
  %6:gpr = LW %0:gpr, target-flags(riscv-lo) @a, debug-location !26 :: (dereferenceable load (s32) from @a); madd.c:8:6
  %7:gpr = LW %2:gpr, target-flags(riscv-lo) @b, debug-location !27 :: (dereferenceable load (s32) from @b); madd.c:8:10
  %8:gpr = LW %4:gpr, target-flags(riscv-lo) @c, debug-location !29 :: (dereferenceable load (s32) from @c); madd.c:8:13
  %9:gpr = nsw MLA killed %6:gpr, killed %7:gpr, killed %8:gpr, debug-location !30; madd.c:8:12
  SW killed %9:gpr, %0:gpr, target-flags(riscv-lo) @a, debug-location !31 :: (store (s32) into @a); madd.c:8:4
  PseudoRET debug-location !32; madd.c:9:1

# End machine code for function maddFunc.
```

Figure 6.9 : Machine Instruction before Register Allocation

6.4 Machine Code Instruction

After register allocation, a Machine Code Instruction (MCInst) representation of the code is created. MCInst can be thought of as an intermediate representation of the lower-level code. It can be used to produce both an object file and an Assembly file. The generated Assembly is presented below:

```
    maddFunc:                               # @maddFunc
# %bb.0:
  addi  sp, sp, -16
.Ltmp0:
  sw  ra, 12(sp)                      # 4-byte Folded Spill
  sw  s0, 8(sp)                       # 4-byte Folded Spill
  addi  s0, sp, 16
  lui a0, %hi(a)
  li  a1, 3
  sw  a1, %lo(a)(a0)
  lui a1, %hi(b)
  li  a2, 103
  sw  a2, %lo(b)(a1)
  lui a2, %hi(c)
  li  a3, 127
  sw  a3, %lo(c)(a2)
  lw  a3, %lo(a)(a0)
  lw  a1, %lo(b)(a1)
  lw  a2, %lo(c)(a2)
  mla a1, a3, a1 ,a2
  sw  a1, %lo(a)(a0)
  lw  ra, 12(sp)                      # 4-byte Folded Reload
  lw  s0, 8(sp)                       # 4-byte Folded Reload
  addi  sp, sp, 16
  ret
```

Code 6.3 **:** madd.s Assembly Output

7. ADDING CUSTOM INSTRUCTIONS

The instruction selection system we focused on at the back-end of the LLVM compiler is SelectionDAG among FastISel and GlobalIsel. SelectionDAG is the most mature Instruction Selection framework with more target support. However, shortly it is worth considering GlobalISel as it is developed recently as an alternative to SelectionDAG. The reasons to replace it are to make it faster, smaller, and more open to low-level optimizations.

7.1 TableGen Reference

TableGen is a domain-specific language used in the LLVM backend side to generate CPP header files. The purpose it serves is that it reduces redundancy of instruction declaration code which can be common to numerous architectures with minor differences. To maintain and scale the framework the minor differences are implemented level by level at a series of inheritance operations between TableGen classes.

LLVM Static Compiler, LLC, is responsible for converting LLVM IR to Assembly codes. To add new instructions LLC is recompiled and its internals change. While the compilation operation of the LLC program, TableGen records are created which declare every instruction's encoding and describe its features. Referring to the records, directed acyclic graphs (DAG) are used in the process of instruction selection. DAG is a graph structure that has no cycles and has directions on the edges. Operations or functions are represented as nodes in the DAG. They are critical parts of declaring the logic or pattern of the new instruction.

The operations represented on the DAG can be LLVM intrinsics as well as instructions. LLVM instructions resemble conventional assembly instructions, in contrast, LLVM intrinsics have higher level abstraction depending on their functionality. Their instruction generation may vary depending on the target hardware. It is possible to define a new complicated instruction either by combining simple LLVM instructions and higher level intrinsics in DAG level or by creating a new LLVM intrinsic which gets created at the Intermediate Level of the compilation process.

7.2 RISC-V TableGen Classes

The most general instruction class used for every target architecture is the "InstructionEncoding" TableGen class defined in llvm/include/llvm/Target/Target.td. This class holds the decoder method and size of instruction in addition to minor variables. It gets inherited to the generic "Instruction" class which is defined in the same class. This class holds input and output DAGs and information which is useful to the compiler and is generalizable to all architectures.

The general class gets inherited to every target-specific class. In RISC-V's case, the next stop of the instruction is the "RVInst" class which inherits from the general "Instruction" class and it resides in llvm/lib/Target/RISCV/RISCVInstrFormats.td TableGen file. It defines the general bit patterns of RISC-V instructions. For example, the opcode being the first 7 bits. It defines additional information like the assembly string pattern. This general class is inherited by every type of instruction of R, I, S, B, U, and J types. As a simple example, XOR instruction can be traced. As XOR is an R type, a register-register instruction, it continues its inheritance journey from "RVInstR". It is common to R type instructions to have funct7, rs2, rs1, funct3, and rd format ordered from MSB to LSB. These variables are assigned corresponding bit fields in the class.

The RISC-V formats mentioned are included in the llvm/lib/Target/RISCV/RISCVInstrInfo.td file which is in the same directory as the RISCVInstrFormats.td file. After inclusion, the "RVInstR" class gets inherited by the "ALU_rr" class. The "ALU_rr" class adds the commutability feature which means swapping source 1 and source 2 does not create a different result like in addition but not in subtraction. At the end, XOR's record is defined by putting funct7, funct3 and assembly string manually in a single line with scheduling information added.

7.3 Adding a New Instruction Using TableGen

We created a tutorial to use as a reference while adding more complex instructions. Create a new TableGen file for custom additions and include it at the end of the RISCVInstrInfo.td file. We named it RISCVInstrInfoCrypt.td as it is going to be cryptography related.

```
include "RISCVInstrInfoCrypt.td"
```

Code 7.1 : Include file

Add the specifications of the instruction to the RISCVInstrInfoCrypt.td file. Here we created a new class of instruction is defined named ALU_rrr. For the MLA instruction, the specifications are:

```
let hasSideEffects = 0, mayLoad = 0, mayStore = 0 in
class ALU_rrr<bits<2> funct2, bits<3> funct3, string opcodestr,
            bit Commutable = 0>
   : RVInstR4<funct2, funct3, OPC_OP,
   (outs GPR:$rd), (ins GPR:$rs1, GPR:$rs2, GPR:$rs3),
             opcodestr, "$rd, $rs1, $rs2 ,$rs3"> {
 let isCommutable = Commutable;
}
```

Code 7.2 : ALU_rrr

Explanation of ALU_rrr:

```
let hasSideEffects = 0, mayLoad = 0, mayStore = 0 in
```

If the instruction has no side effect, hasSideEffects will be 0 If there is no need or possibility to load data from memory, mayLoad will be 0 If there is no need or possibility to store data from memory, mayStore will be 0

```
class ALU_rrr<bits<2> funct2, bits<3> funct3, string opcodestr,
            bit Commutable = 0>
```

Class is defined with ALU_rrr name. Variables are defined. funct2 is two bits of binary number as RVInstR4 is used which reserves 5 bits of funct7 for another register. funct3 is three bits of binary number. opcodestr is the string that will be shown in the assembly file. Commutable is one bit of zero which determines the importance of the order of the inputs.

```
: RVInstR4<funct2, funct3, OPC_OP, (outs GPR:$rd), (ins GPR:$rs1, GPR:$rs2, GPR:$rs3),
```

RVInstR4 instruction type is called from RISCVInstrFormats.td file. funct2, funct3, opcode, output, and inputs are entered in function orderly.

```
opcodestr, "$rd, $rs1, $rs2 ,$rs3"> {
 let isCommutable = Commutable;
}
```

opcode string and activating commutability option.

Create another file in which we are going to add the schedules. In our case, the name of this file is “RISCVInstrInfoCrypt.td”. The definition of the instruction should be added to this file by using ALU_rrr class defined above and inputting the correct scheduling variables. For the MLA instruction, it is:

```
def MLA : ALU_rrr<0b10, 0b100, "mla">,
Sched<[WriteIMul, ReadIMul, ReadIMul]>;
```

MLA instruction is defined and ALU_rrr instruction type is used. funct2,funct3, and opcode strings are entered. Schedules are determined.

Add the instruction’s pattern defining its logic to the RISCVInstrInfoCrypt.td file. For the MLA instruction, it is:

```
def : Pat< (add (mul GPR:$src1, GPR:$src2), GPR:$src3),
(MLA GPR:$src1, GPR:$src2, GPR:$src3)>;
```

MLA instruction is called from RISCVInstrInfoCrypt.td file and inputs and outputs are determined.

7.4 Adding Pattern Matching Support for New Instruction Using C++ in SelectionDAG

TableGen aims to provide a declarative way to introduce new patterns for new instruction developers. However, not all instructions can be described in this scheme. Although it is called "dag" as a keyword in TableGen, it expects a tree of instructions. Custom C++ can be the only way to match until the TableGen based system improves. It is also possible to use C++ in complex patterns in TableGen, which can make the most of the pattern declarative and only the necessary part in imperative style.

A domain that TableGen fails is matching a graph of instructions with dependant operands. As an example we can think of an instruction having two operands of Load Instructions. If the Load instructions are from an array, they must be related to each other by an offset and it might need to be detected for certain patterns.

```
define void @sbox(ptr  %0)  {
  %2 = getelementptr inbounds [5 x i32], ptr %0, i32 0, i32 4
  %3 = load i32, ptr %2
  %4 = load i32, ptr %0
  %5 = xor i32 %4, %3
```

Code 7.3 : Minimal Subtree of Optimized S-box LLVM IR

In Code 7.3, it can be seen that the XOR instruction is between the first element of the input struct and the fifth element of it. As the locations of elements matters, they must be matched by not only looking at Load instructions but also their operands. What we are looking for is to have the base of Load instruction to be equal and the offset operand of it to evaluate to 4, designating the fifth element of struct. TableGen is not suitable for this operation and the source code of SelectionDAG should be analyzed to place the logic to pattern match this set of instructions.

The process for adding an instruction via C++ is as follows:

1. Create a record declaration in TableGen to provide the Assembler support, ignoring the Pattern declaration.

2. Observe the DAG in the debug output or dot file and locate the root of it.

3. Add a function in RISCVISelDAGToDAG.cpp file in the root instruction case.

4. Implement pattern matching and replacement with SelectionDAG node.

SelectionDAG consists of numerous files but the most relevant ones to the developer can be few if the complexity of the pattern is small. The order of files will be from IR to Assembly. RISCVCodeGenPrepare.cpp file provides mechanisms for matching in IR form. This file exists mainly due to the limitation of SelectionDAG which is running per basic-block. RISCVISelLowering.cpp contains the lowering of IR to SelectionDAG nodes. It can decide on whether a type or expression should be legalized or expanded. RISCVISelDAGToDAG.cpp is the instruction selector in C++. Its implementation traverses the DAG from the root and runs the selection functions depending on the SelectionDAG node type which is parallel to instructions.

```
t0: ch,glue = EntryToken
t2: i32,ch = CopyFromReg t0, Register:i32 %0
t8: i32,ch = load<(load (s32) from %ir.0, !tbaa !7)> t0, t2, ↩
  undef:i32
t4: i32 = add nuw t2, Constant:i32<16>
t7: i32,ch = load<(load (s32) from %ir.2, !tbaa !7)> t0, t4, ↩
  undef:i32
t9: i32 = xor t8, t7
```

Code 7.4 **:** The corresponding Optimized and Legalized DAG of Code 7.3

In an attempt to match the three instructions in Code 7.3, the DAG in Code 7.4 is analyzed. First remark is that "getelementptr"is converted to an add instruction which calculates the offset. Another remark is that the load instructions have the same base address in first operand as expected pointing to the same node. If the pattern is large, we can introduce a new function which will contain the logic. The function's prototype should be added to the corresponding header file.

```
void RISCVDAGToDAGISel::Select(SDNode *Node) {
    .
}
    .
switch (Opcode) {
    .
    .
    case ISD::XOR:{
        if (tryShrinkShlLogicImm(Node))
          return;
        if (selectSbox(Node))
          return;
        break;
    }
}
```

Code 7.5 : Introduction of New Function for Pattern Matching in C++

The C++ logic for matching this pattern is provided below.

```
bool RISCVDAGToDAGISel::selectSBox(SDNode *Node) {
    SDValue LOAD0 = Node->getOperand(0);
    SDValue LOAD1 = Node->getOperand(1);
    //Check if there is a load pair
    if(LOAD0.getOpcode() != ISD::LOAD || LOAD1.getOpcode() != ISD::LOAD)
      return false;
    SDValue LOAD0_op0 = LOAD0.getOperand(0);
    SDValue LOAD0_op1_offset = LOAD0.getOperand(1); // t0

    SDValue LOAD1_op0 = LOAD1.getOperand(0);
    SDValue LOAD1_op1_offset = LOAD1.getOperand(1);

    if(LOAD1_op1_offset.getOpcode() != ISD::ADD)
        return false;

    if(LOAD1_op0 != LOAD0_op0)
        return false;

   //Check if the addendum0 is the same as the second
    SDValue LOAD1_op1_offset_Addendum0 = LOAD1_op1_offset.getOperand(0);
    if(LOAD1_op1_offset_Addendum0 != LOAD0_op1_offset)
        return false;

    SDValue LOAD1_op1_offset_Addendum1 = LOAD1_op1_offset.getOperand(1);

    auto *LOAD1_op1_offset_Addendum1C =
        dyn_cast<ConstantSDNode>(LOAD1_op1_offset_Addendum1);
    if(!LOAD1_op1_offset_Addendum1C)
        return false;
    //Check if addendum1 is 16 more than first load offset
    if(LOAD1_op1_offset_Addendum1C->getZExtValue() != 16)
        return false;

    //Pattern is matched replace here

    return true;
}
```

Code 7.6 **:** C++ logic for Pattern Matching the DAG in Code 7.4

The logic starts from the root of the DAG which in this case is XOR instruction or the ISD::XOR SelectionDAG node. Then we iterate through its leaves and check the distinctive features of the pattern. In this pattern we are interested in the distance of offset addresses of Load instructions so we progressively approach them by assuming the pattern holds and quitting if not. Progressive checking is a common theme in LLVM and as Instruction Selection is one of the stages affecting the compilation times significantly, patterns should not be checked in a single if statement by logical combinations.

The checks if statements are doing can be summarised in steps. As the function only runs in XOR case the root can be assumed to be XOR safely.

1. Check if the operands are both Load instructions.
2. Check if the second Load instruction has an Add instruction in its second operand
3. Check if the base offset of the first load and the first addendum of the second load are the same, as they should point to the beginning of the struct.
4. Check if the second addendum is a constant.
5. Check if the unsigned value of constant second addendum is equal to 16.

At this point, it can be safely assumed that the only XOR that conforms to the pattern can be in this line of program. SelectionDAG provides more API to simply replace the nodes in that pattern with the custom instruction.

To interact with the DAG, SelectionDAG's API is used. We encourage the developers to read the source code and learn to use the public functions exposed by SelectionDAG in order to interact with the DAG most effectively. RISCVISelDAGToDAG.cpp file already has many instruction selection mechanisms in place which can be read through.

7.5 Discussion of Pattern Matching in Other Stages of the Compiler

Pattern Matching can be assumed to mainly be an Instruction Selection problem where the pattern will be simply identified and replaced. However, when we take a look at the baseline problem any Compiler technology solves, it is to convert more familiar patterns in some language to a more unfamiliar pattern in machine language. Also this conversion occurs in a large number of steps through optimisation in IR form as discussed in Section 3.2, to DAG formation in SelectionDAG to MCInstr form down the pipeline. Their data structures can differ in representing the instructions which can make some pattern matching schemes to be more fragile than others. Also, as the lowering gets performed high level information about the program is lost but the formation gets closer to the final output of Assembly.

For simple cases where for example a combination of R-type instructions will be matched and replaced, Instruction Selection might be the most convenient stage to extend. Howeveer if the pattern requires the instruction selection to be performed already, pattern match can be done in MCInst level. In contrary, if higher level information of the pattern is required, a pattern can be matched to an intrinsic function in IR level.

Another reason to consider different stages is that there can be multiple patterns mapped to the same instruction. Further optimization opportunities can rise in further stages.

7.5.1 Case Study: SH1ADD in SelectionDAG and MCInst Level

It was discussed that dealing with the lowered DAG to MIR can provide more optimization opportunities. In this section the case of "SH1ADD" instruction which is ratified in RISC-V Zba extension will be analyzed. The instruction shifts rs1 left by one, adds rs2 and writes to rd. Its encoding and pattern in the standard implementation of LLVM in TableGen is as follows:

```
let Predicates = [HasStdExtZba] in {
def SH1ADD : ALU_rr<0b0010000, 0b010, "sh1add">,
             Sched<[WriteSHXADD, ReadSHXADD, ReadSHXADD]>;
def SH2ADD : ALU_rr<0b0010000, 0b100, "sh2add">,
             Sched<[WriteSHXADD, ReadSHXADD, ReadSHXADD]>;
def SH3ADD : ALU_rr<0b0010000, 0b110, "sh3add">,
             Sched<[WriteSHXADD, ReadSHXADD, ReadSHXADD]>;
} // Predicates = [HasStdExtZba]
```

Code 7.7 : Instruction Encoding of the Instructions

As the SH2ADD and SH3ADD have similar implementations to SH1ADD their patterns will be stripped.

```
let Predicates = [HasStdExtZba] in {
def : Pat<(add (shl GPR:$rs1, (XLenVT 1)), non_imm12:$rs2),
          (SH1ADD GPR:$rs1, GPR:$rs2

// More complex cases use a ComplexPattern.
def : Pat<(add sh1add_op:$rs1, non_imm12:$rs2),
          (SH1ADD sh1add_op:$rs1, GPR:$rs2)>;)>;

def : Pat<(add (mul_oneuse GPR:$rs1, (XLenVT 6)), GPR:$rs2),
          (SH1ADD (SH1ADD GPR:$rs1, GPR:$rs1), GPR:$rs2)>;
def : Pat<(add (mul_oneuse GPR:$rs1, (XLenVT 10)), GPR:$rs2),
          (SH1ADD (SH2ADD GPR:$rs1, GPR:$rs1), GPR:$rs2)>;
def : Pat<(add (mul_oneuse GPR:$rs1, (XLenVT 18)), GPR:$rs2),
          (SH1ADD (SH3ADD GPR:$rs1, GPR:$rs1), GPR:$rs2)>;
```

Code 7.8 : Instruction Pattern of the Instructions

We can observe that ComplexPattern's are used to enable using C++ together with TableGen. The Complex Pattern's TableGen declarations are provided below:

```

def sh1add_op : ComplexPattern<XLenVT, 1, "selectSHXADDOp<1>", [], [], 6>;

class binop_oneuse<SDPatternOperator operator>
    : PatFrag<(ops node:$A, node:$B),
              (operator node:$A, node:$B), [{
  return N->hasOneUse();
}]>;

def mul_oneuse : binop_oneuse<mul>;
```

Code 7.9 : TableGen Declaration of ComplexPatterns

"selectSHXADDOp" is a template function which provides the shift amount argument.

```

bool selectSHXADDOp(SDValue N, unsigned ShAmt, SDValue &Val);
template <unsigned ShAmt> bool selectSHXADDOp(SDValue N, SDValue &Val) {
return selectSHXADDOp(N, ShAmt, Val);
}
```

Code 7.10 : Template Function of the ComplexPattern for "sh1add_op"

The C++ logic can be found in RISCVISelDAGToDAG.cpp file, the implementation will be reduced to the patterns described in comments:

```
/// Look for various patterns that can be done with a SHL that can be folded
/// into a SHXADD. \p ShAmt contains 1, 2, or 3 and is set based on which
/// SHXADD we are trying to match.
bool RISCVDAGToDAGISel::selectSHXADDOp(SDValue N, unsigned ShAmt,
                                       SDValue &Val) {
  if (N.getOpcode() == ISD::AND && isa<ConstantSDNode>(N.getOperand(1))) {
    SDValue N0 = N.getOperand(0);

    bool LeftShift = N0.getOpcode() == ISD::SHL;
    if ((LeftShift || N0.getOpcode() == ISD::SRL) &&
        isa<ConstantSDNode>(N0.getOperand(1))) {
      uint64_t Mask = N.getConstantOperandVal(1);
      unsigned C2 = N0.getConstantOperandVal(1);

      unsigned XLen = Subtarget->getXLen();
      if (LeftShift)
        Mask &= maskTrailingZeros<uint64_t>(C2);
      else
        Mask &= maskTrailingOnes<uint64_t>(XLen - C2);

      // Look for (and (shl y, c2), c1) where c1 is a shifted mask with no
      // leading zeros and c3 trailing zeros. We can use an SRLI by c2+c3
      // followed by a SHXADD with c3 for the X amount.
       ...
        // Look for (and (shr y, c2), c1) where c1 is a shifted mask with c2
        // leading zeros and c3 trailing zeros. We can use an SRLI by C3
        // followed by a SHXADD using c3 for the X amount.
        ...
      }
    }
  }

  bool LeftShift = N.getOpcode() == ISD::SHL;
  if ((LeftShift || N.getOpcode() == ISD::SRL) &&
      isa<ConstantSDNode>(N.getOperand(1))) {
    SDValue N0 = N.getOperand(0);
    if (N0.getOpcode() == ISD::AND && N0.hasOneUse() &&
        isa<ConstantSDNode>(N0.getOperand(1))) {
      uint64_t Mask = N0.getConstantOperandVal(1);
      if (isShiftedMask_64(Mask)) {
        unsigned C1 = N.getConstantOperandVal(1);
        unsigned XLen = Subtarget->getXLen();
        unsigned Leading = XLen - llvm::bit_width(Mask);
        unsigned Trailing = llvm::countr_zero(Mask);
        // Look for (shl (and X, Mask), C1) where Mask has 32 leading zeros and
        // C3 trailing zeros. If C1+C3==ShAmt we can use SRLIW+SHXADD.
        ...
        // Look for (srl (and X, Mask), C1) where Mask has 32 leading zeros and
        // C3 trailing zeros. If C3-C1==ShAmt we can use SRLIW+SHXADD.
        ...
      }
    }
  }
  return false;
}
```

Code 7.11 : Implementation of the ComplexPattern for "sh1add_op"

Despite using TableGen and SelectionDAG, there is an optimization opportunity for "SH1ADD" in MIR form. It is done in immediate materialisation where a constant node in the DAG representation is not converted to instructions until needed. In the MCTargetDesc/RISCVMatInt.cpp file, an optimisation for representing immediates is provided as follows:

```
namespace llvm::RISCVMatInt {
InstSeq generateInstSeq(int64_t Val, const FeatureBitset &ActiveFeatures) {
  RISCVMatInt::InstSeq Res;
  generateInstSeqImpl(Val, ActiveFeatures, Res);
  ...

// Perform optimization with SH*ADD in the Zba extension.
  if (Res.size() > 2 && ActiveFeatures[RISCV::FeatureStdExtZba]) {
    int64_t Div = 0;
    unsigned Opc = 0;
    RISCVMatInt::InstSeq TmpSeq;
    // Select the opcode and divisor.
    if ((Val % 3) == 0 && isInt<32>(Val / 3)) {
      Div = 3;
      Opc = RISCV::SH1ADD;
    } else if ((Val % 5) == 0 && isInt<32>(Val / 5)) {
      Div = 5;
      Opc = RISCV::SH2ADD;
    } else if ((Val % 9) == 0 && isInt<32>(Val / 9)) {
      Div = 9;
      Opc = RISCV::SH3ADD;
    }
    // Build the new instruction sequence.
    if (Div > 0) {
      generateInstSeqImpl(Val / Div, ActiveFeatures, TmpSeq);
      TmpSeq.emplace_back(Opc, 0);
      if (TmpSeq.size() < Res.size())
        Res = TmpSeq;
    } else {
      // Try to use LUI+SH*ADD+ADDI.
      int64_t Hi52 = ((uint64_t)Val + 0x800ull) & ~0xfffull;
      int64_t Lo12 = SignExtend64<12>(Val);
      Div = 0;
      if (isInt<32>(Hi52 / 3) && (Hi52 % 3) == 0) {
        Div = 3;
        Opc = RISCV::SH1ADD;
      } else if (isInt<32>(Hi52 / 5) && (Hi52 % 5) == 0) {
        Div = 5;
        Opc = RISCV::SH2ADD;
      } else if (isInt<32>(Hi52 / 9) && (Hi52 % 9) == 0) {
        Div = 9;
        Opc = RISCV::SH3ADD;
      }
      // Build the new instruction sequence.
      if (Div > 0) {
        // For Val that has zero Lo12 (implies Val equals to Hi52) should has
        // already been processed to LUI+SH*ADD by previous optimization.
        generateInstSeqImpl(Hi52 / Div, ActiveFeatures, TmpSeq);
        TmpSeq.emplace_back(Opc, 0);
        TmpSeq.emplace_back(RISCV::ADDI, Lo12);
        if (TmpSeq.size() < Res.size())
          Res = TmpSeq;
      }
    }
  }
```

Code 7.12 : Immediate Materialisation for "SH1ADD"

By using "SH1ADD" in immediate materialization, a single instruction can be used instead of two. It can be checked that in the standard testing suite there is the following function which returns a 64 bit integer:

```
define i64 @PR54812() {
; RV64I-LABEL: PR54812:
; RV64I:       # %bb.0:
; RV64I-NEXT:    lui a0, 1048447
; RV64I-NEXT:    addiw a0, a0, 1407
; RV64I-NEXT:    slli a0, a0, 12
; RV64I-NEXT:    ret
;
; RV64IZBA-LABEL: PR54812:
; RV64IZBA:       # %bb.0:
; RV64IZBA-NEXT:    lui a0, 872917
; RV64IZBA-NEXT:    sh1add a0, a0, a0
; RV64IZBA-NEXT:    ret
;
  ret i64 -2158497792;
}
```

Code 7.13 : Function for Immediate Materialisation

In the FileCheck lines which is explained in Section **??**, we can see the Assembly lines that should be emmitted. It can be observed that the desired immediate can be obtained with "sh1add" in less instructions.

7.5.2 Case Study: ROR in SelectionDAG and IR Level

In LLVM, rotation instruction which is shifting and feeding the carry back to the shift point, is captured in IR level. To match in IR level, a new intrinsic function is defined in TableGen. TableGen is not only used in instruction selection, it is used wherever a declarative form is better suited such as in IR or MLIR levels.

```
//===--------------------- Bit Manipulation Intrinsics ----------------------===//
//

// None of these intrinsics accesses memory at all.
let IntrProperties = [IntrNoMem, IntrSpeculatable, IntrWillReturn] in {
  def int_bswap: DefaultAttrsIntrinsic<[llvm_anyint_ty], [LLVMMatchType<0>]>;
  def int_ctpop: DefaultAttrsIntrinsic<[llvm_anyint_ty], [LLVMMatchType<0>]>;
  def int_bitreverse: DefaultAttrsIntrinsic<[llvm_anyint_ty], [LLVMMatchType<0>]>
  def int_fshl : DefaultAttrsIntrinsic<[llvm_anyint_ty],
      [LLVMMatchType<0>, LLVMMatchType<0>, LLVMMatchType<0>]>;
  def int_fshr : DefaultAttrsIntrinsic<[llvm_anyint_ty],
      [LLVMMatchType<0>, LLVMMatchType<0>, LLVMMatchType<0>]>;
}
```

Code 7.14 : Funnel Shift Intrinsic Definition

’fshr’ is defined as funnel shift right intrinsic function [33]. It is matched in IR optimizations by InstCombine pass.

```
/// Match UB-safe variants of the funnel shift intrinsic.
static Instruction *matchFunnelShift(Instruction &Or, InstCombinerImpl &IC){
  unsigned Width = Or.getType()->getScalarSizeInBits();

  // First, find an or'd pair of opposite shifts:
  // or (lshr ShVal0, ShAmt0), (shl ShVal1, ShAmt1)
  BinaryOperator *Or0, *Or1;
  if (!match(Or.getOperand(0), m_BinOp(Or0)) ||
      !match(Or.getOperand(1), m_BinOp(Or1)))
    return nullptr;

  Value *ShVal0, *ShVal1, *ShAmt0, *ShAmt1;
  if (!match(Or0, m_OneUse(m_LogicalShift(m_Value(ShVal0),
       m_Value(ShAmt0)))) ||
      !match(Or1, m_OneUse(m_LogicalShift(m_Value(ShVal1),
       m_Value(ShAmt1)))) ||
      Or0->getOpcode() == Or1->getOpcode())
    return nullptr;

  // Canonicalize to or(shl(ShVal0, ShAmt0), lshr(ShVal1, ShAmt1)).
  if (Or0->getOpcode() == BinaryOperator::LShr) {
    std::swap(Or0, Or1);
    std::swap(ShVal0, ShVal1);
    std::swap(ShAmt0, ShAmt1);
  }

  // Match the shift amount operands for a funnel shift pattern. This always
  // matches a subtraction on the R operand.
  auto matchShiftAmount = [&](Value *L, Value *R, unsigned Width) -> Value * {
    // Check for constant shift amounts that sum to the bitwidth.
    ...

    return nullptr;
  };

  Value *ShAmt = matchShiftAmount(ShAmt0, ShAmt1, Width);
  bool IsFshl = true; // Sub on LSHR.
  if (!ShAmt) {
    ShAmt = matchShiftAmount(ShAmt1, ShAmt0, Width);
    IsFshl = false; // Sub on SHL.
  }
  if (!ShAmt)
    return nullptr;

  Intrinsic::ID IID = IsFshl ? Intrinsic::fshl : Intrinsic::fshr;
  Function *F = Intrinsic::getDeclaration(Or.getModule(), IID, Or.getType());
  return CallInst::Create(F, {ShVal0, ShVal1, ShAmt});
}
```

Code 7.15 : Funnel Shift Right Pattern Matching

The pattern matching API is provided by IR/PatternMatch.h file. After the pattern for rotation and the shift amount are matched the corresponding intrinsic function is called.

The function is called in OR visiting function, so the root of the pattern is OR:

```
Instruction *InstCombinerImpl::visitOr(BinaryOperator &I) {
  ...
  if (Instruction *Funnel = matchFunnelShift(I, *this))
    return Funnel;
  ...
  return nullptr;
}
```

Code 7.16 : Funnel Shift Right Pattern Function Called

The intrinsic function is converted to ROTR SelectionDAG node in the general SelectionDAGBuilder.cpp file.

```
/// Lower the call to the specified intrinsic function.
void SelectionDAGBuilder::visitIntrinsicCall(const CallInst &I,
                                             unsigned Intrinsic) {
  const TargetLowering &TLI = DAG.getTargetLoweringInfo();
  SDLoc sdl = getCurSDLoc();
  DebugLoc dl = getCurDebugLoc();
  SDValue Res;

  SDNodeFlags Flags;
  if (auto *FPOp = dyn_cast<FPMathOperator>(&I))
    Flags.copyFMF(*FPOp);

  switch (Intrinsic) {
  default:
    // By default, turn this into a target intrinsic node.
    visitTargetIntrinsic(I, Intrinsic);
    return;
  ...

  case Intrinsic::fshl:
  case Intrinsic::fshr: {
    bool IsFSHL = Intrinsic == Intrinsic::fshl;
    SDValue X = getValue(I.getArgOperand(0));
    SDValue Y = getValue(I.getArgOperand(1));
    SDValue Z = getValue(I.getArgOperand(2));
    EVT VT = X.getValueType();

    if (X == Y) {
      auto RotateOpcode = IsFSHL ? ISD::ROTL : ISD::ROTR;
      setValue(&I, DAG.getNode(RotateOpcode, sdl, VT, X, Z));
    } else {
      auto FunnelOpcode = IsFSHL ? ISD::FSHL : ISD::FSHR;
      setValue(&I, DAG.getNode(FunnelOpcode, sdl, VT, X, Y, Z));
    }
    return;
  }
  ...
}
```

Code 7.17 : Funnel Shift Intrinsic converted to ROTL

The intrinsic functions can be defined as target-specific or target independent. "fshl" is a general intrinsic function and targets can either expand it by replacing it with its equivalent instructions or lower it directly to an instruction by legalizing it. RISC-V Zbb extension supports bitwise rotation so LLVM has the extension’s implementation in the source. In RISCVISelLowering.cpp file the legalization of "ROTR" is managed regarding whether the extension is enabled or disabled.

```
RISCVTargetLowering::RISCVTargetLowering(const TargetMachine &TM,
                                         const RISCVSubtarget &STI)
    : TargetLowering(TM), Subtarget(STI) {
    ...
  if (Subtarget.hasStdExtZbb() || Subtarget.hasStdExtZbkb() ||
      Subtarget.hasVendorXTHeadBb()) {
    if (Subtarget.is64Bit())
      setOperationAction({ISD::ROTL, ISD::ROTR}, MVT::i32, Custom);
  } else {
    setOperationAction({ISD::ROTL, ISD::ROTR}, XLenVT, Expand);
  }
  ...
}
```

Code 7.18 : ROTR Legalization Conditional

The "Custom" action is defined in TableGen in RISCVInstrInfoZb.td file as well as instruction encodings.

```
def riscv_rolw   : SDNode<"RISCVISD::ROLW",    SDT_RISCVIntBinOpW>;
def riscv_rorw   : SDNode<"RISCVISD::RORW",    SDT_RISCVIntBinOpW>;
...
let Predicates = [HasStdExtZbbOrZbkb] in {
def ROL    : ALU_rr<0b0110000, 0b001, "rol">,
             Sched<[WriteRotateReg, ReadRotateReg, ReadRotateReg]>;
def ROR    : ALU_rr<0b0110000, 0b101, "ror">,
             Sched<[WriteRotateReg, ReadRotateReg, ReadRotateReg]>;

def RORI   : RVBShift_ri<0b01100, 0b101, OPC_OP_IMM, "rori">,
             Sched<[WriteRotateImm, ReadRotateImm]>;
} // Predicates = [HasStdExtZbbOrZbkb]
...
let Predicates = [HasStdExtZbbOrZbkb] in {
def : PatGprGpr<shiftop<rotl>, ROL>;
def : PatGprGpr<shiftop<rotr>, ROR>;

def : PatGprImm<rotr, RORI, uimmlog2xlen>;
// There’s no encoding for roli in the the ’B’ extension as it can be
// implemented with rori by negating the immediate.
def : Pat<(rotl GPR:$rs1, uimmlog2xlen:$shamt),
          (RORI GPR:$rs1, (ImmSubFromXLen uimmlog2xlen:$shamt))>;
} // Predicates = [HasStdExtZbbOrZbkb]
```

Code 7.19 : ROR Encodings and Pattern

As you can see when the pattern match logic is lifted up to the IR level the modifications in Instruction Selection are straightforward to implement.

8. REALISTIC CONSTRAINTS AND CONCLUSIONS

CONCLUSION BURAYA GELEBILIR!!!!

8.1 Practical Application of This Project

This project can be useful for increasing the efficiency of applications that require the frequent use of specific instructions. Cryptography applications with RISC-V may be one of these.

8.2 Realistic Constraints

LLVM is a huge infrastructure and while working with it, sometimes it may be hard to find what you are looking for. Also, there aren't many sources or documentation to find solutions to the specific problems that we encounter which sometimes slows down the progress.

8.2.1 Social, Environmental, and Economic Impact

The end product is going to help the custom processor to be programmed by a high level programming language. It will make the programming of the custom processor a more efficient process and encourage the use of the custom processor. Because of this efficiency, the energy and time costs would be reduced during the programming of the processor. Also, using a custom processor for handling a problem is faster and requires less power. Therefore, encouraging the use of one would be another benefit of the end product.

8.2.2 Cost

Open-source tools and programs were used on our computers during the project. Therefore, it wasn't costly for us.(bu şekilde mi olsun form 3'teki gibi mi olsun?)

8.2.3 Standards

LLVM project is very selective on the technologies they use. Latest versions of C++ and build tools with software engineering principles are followed. Instructions abide by the RISC-V instruction set standard.

8.2.4 Health and Safety Concerns

Since we are working on software area, there is no possible risk of harm to users.

8.3 Future Work and Recommendations

REFERENCES

[1] (2021), Learn the basics of instruction set architecture - EDN Asia, `https://www.ednasia.com/learn-the-basics-of-instruction-set-architecture`, [Online; accessed 15. Apr. 2023].

[2] (2020), RISC-V Assembly Language, `https://web.eecs.utk.edu/~smarz1/courses/ece356/notes/assembly`, [Online; accessed 15. Apr. 2023].

[3] **Waterman, A. and Asanovic, K.**, (2019), The RISC-V Instruction Set Manual Volume I: Unprivileged ISA.

[4] **Gholizadehazari, E.**, (2021), An FPGA Implementation of a RISC-V Based SOC System with Custom Instruction Set for Image Processing Applications.

[5] (2023), LLVM support for the draft Bit Manipulation Extension for RISC-V – Embecosm, `https://www.embecosm.com/2019/10/22/llvm-risc-v-bit-manipulation-extension`, [Online; accessed 15. Apr. 2023].

[6] **Wolf, C.**, (2021), RISC-V Bitmanip Extension.

[7] (2023), Scalar Cryptography Instruction Set Extension Group Names Diagram - Home - RISC-V International, `https://wiki.riscv.org/display/HOME/Scalar+Cryptography+Instruction+Set+Extension+Group+Names+Diagram`, [Online; accessed 15. Apr. 2023].

[8] **Aho, A.** (2007). *Compilers: Principles, Techniques, and Tools*, Addison-Wesley.

[9] (2022), "Clang" CFE Internals Manual — Clang 16.0.0git documentation, `https://clang.llvm.org/docs/InternalsManual.html`, [Online; accessed 5. Jan. 2023].

[10] **Finkel, H.**, (2016), Intrinsics, Metadata, and Attributes: The story continues! 2016 LLVM Developers' Meeting.

[11] (2023), 9. Intrinsic Functions - SKKU Compiler7987, `https://sites.google.com/site/compiler7987/intermediate-represintation/9-intrinsic-functions`, [Online; accessed 16. May 2023].

[12] LLVM's Analysis and Transform Passes, https://llvm.org/docs/Passes.html.

[13] LLVM Language Reference Manual - Function Attributes, https://llvm.org/docs/LangRef.html#function-attributes.

[14] LLVM Language Reference Manual - Object Lifetime, https://llvm.org/docs/LangRef.html#object-lifetime.

[15] LLVM Source Code - SROA.cpp, https://github.com/llvm/llvm-project/blob/main/llvm/lib/Transforms/Scalar/SROA.cpp#L9.

[16] **Prosser, R.T.** (1959). Applications of Boolean Matrices to the Analysis of Flow Diagrams, *Papers Presented at the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference*, IRE-AIEE-ACM '59 (Eastern), Association for Computing Machinery, New York, NY, USA, p.133–138, https://doi.org/10.1145/1460299.1460314.

[17] MemorySSA — LLVM 17.0.0git documentation, https://llvm.org/docs/MemorySSA.html.

[18] **Novillo, D.** *et al.* (2007). Memory SSA-a unified approach for sparsely representing memory operations, *Proceedings of the GCC Developers' Summit*, Citeseer, pp.97–110.

[19] LLVM's Analysis and Transform Passes, https://releases.llvm.org/9.0.0/docs/Passes.html#instcombine-combine-redundant-instructions.

[20] "Clang" Clang CLI Documentation — Clang 17.0.0git documentation, https://clang.llvm.org/docs/CommandGuide/clang.html#code-generation-options, [Online; accessed 18. Apr. 2023].

[21] (2022), opt - LLVM optimizer — LLVM 16.0.0git documentation, https://llvm.org/docs/CommandGuide/opt.html, [Online; accessed 5. Jan. 2023].

[22] (2022), The LLVM Target-Independent Code Generator — LLVM 16.0.0git documentation, https://llvm.org/docs/CodeGenerator.html, [Online; accessed 5. Jan. 2023].

[23] Legalizer, https://llvm.org/docs/GlobalISel/Legalizer.html.

[24] **Mayur Pandey, S.S.** (2015). *LLVM Cookbook*, Packt Publishing Ltd.

[25] (2023), About RISC-V – RISC-V International, https://riscv.org/about, [Online; accessed 15. Apr. 2023].

[26] **Waterman, A.**, (2016), Design of the RISC-V instruction set architecture.

[27] **Altınay, O.**, (2021), Instruction Extension of RV32I and GCC Back End for ASCON Lightweight Cryptography Algorithm.

[28] (2021), RISC-V Bit-Manipulation ISA-Extensions, version 1.0.0-38-g865e7a7.

[29] (2023), RISC-V Cryptography Extensions Task Group Announces Public Review of the Scalar Cryptography Extensions – RISC-V International, `https://riscv.org/blog/2021/09/risc-v-cryptography-extensions-task-group-announces-public-revie` [Online; accessed 15. Apr. 2023].

[30] (2022), RISC-V Cryptography Extensions Volume I Scalar & Entropy Source Instructions.

[31] (2023), User Guide for RISC-V Target — LLVM 17.0.0git documentation, `https://llvm.org/docs/RISCVUsage.html`, [Online; accessed 20. Apr. 2023].

[32] (2022), LLVM: lib/Target/RISCV/RISCVISelLowering.h Source File, `https://llvm.org/doxygen/RISCVISelLowering_8h_source.html`, [Online; accessed 5. Jan. 2023].

[33] LLVM Language Reference Manual - ‘llvm.fshl.*’ Intrinsic, `https://llvm.org/docs/LangRef.html#llvm-fshl-intrinsic`.

APPENDICES

APPENDIX A.1

1.1 Installation of Software

As the LLVM codebase is large and has many options while building from source, finding the right options that our computers can handle easily was both essential to get started and critical as it decides the time it takes to see a change in code to get compiled. For this purpose, we accumulated the commands and created a tutorial that we can use in the future.

```
git clone https://github.com/llvm/llvm-project
cd llvm-project
mkdir build
cd build
sudo apt install cmake, ninja-build, clang, lld
```

Code A.1 : Clone Repository and Install Necessary Packages

```
cmake -S ../llvm . -G Ninja -DCMAKE_BUILD_TYPE="Debug"  \
-DBUILD_SHARED_LIBS=True -DLLVM_USE_SPLIT_DWARF=True  \
-DLLVM_BUILD_TESTS=True   -DCMAKE_C_COMPILER=clang \
-DCMAKE_CXX_COMPILER=clang++ -DLLVM_TARGETS_TO_BUILD="all" \
-DLLVM_EXPERIMENTAL_TARGETS_TO_BUILD="RISCV" -DLLVM_ENABLE_LLD=ON
```

Code A.2 : CMake Configuration We Used

With this command, we are choosing the type as debug. Shared_libs=TRUE causes all libraries to be built shared instead of static libraries. ..SPLIT_DWARF is set to True to minimize memory usage at link time. We want to use clang as the C compiler. Therefore, it is specified in the command as DCMAKE_C_COMPILER=clang. In addition to that, we want to use lld as the linker instead of gold, so we specify that as well. This configuration is the most efficient in terms of memory and disk usage among our previous attempts at building LLVM from source.

```
Ninja
```

Code A.3 : To build from scratch or to rebuild files with change, automatically

```
ninja llc
```

Code A.4 : To build llc only which is the binary we modify

While running ninja, CPU and ram usage significantly increases. All available cores are used capacity. This may prevent doing other tasks while running ninja. In order to prevent this one may opt to use the following command instead. It allows you to choose how many cores are going to be utilized.

```
  ninja llc -j<number of cores to use>
```

APPENDIX A.2

1.2 Unoptimized S-box IR Code

The output of the unoptimized S-box function is provided below. The C code used to produce this LLVM IR is in Code 3.2.3

```
; ModuleID = 's-box.c'
source_filename = "s-box.c"
target datalayout = "e-m:e-p:32:32-i64:64-n32-S128"
target triple = "riscv32-unknown-linux-gnu"

%struct.ascon_state_t = type { [5 x i32] }

; Function Attrs: nounwind uwtable
define dso_local void @sbox(ptr noundef %state) #0 {
entry:
  %state.indirect_addr = alloca ptr, align 4
  %t0 = alloca i32, align 4
  %t1 = alloca i32, align 4
  %t2 = alloca i32, align 4
  %t3 = alloca i32, align 4
  %t4 = alloca i32, align 4
  store ptr %state, ptr %state.indirect_addr, align 4, !tbaa !7
  call void @llvm.lifetime.start.p0(i64 4, ptr %t0) #2
  call void @llvm.lifetime.start.p0(i64 4, ptr %t1) #2
  call void @llvm.lifetime.start.p0(i64 4, ptr %t2) #2
  call void @llvm.lifetime.start.p0(i64 4, ptr %t3) #2
  call void @llvm.lifetime.start.p0(i64 4, ptr %t4) #2
  %x = getelementptr inbounds %struct.ascon_state_t, ptr %state, ↩
    i32 0, i32 0
  %arrayidx = getelementptr inbounds [5 x i32], ptr %x, i32 0, i32↩
    4
  %0 = load i32, ptr %arrayidx, align 4, !tbaa !11
  %x1 = getelementptr inbounds %struct.ascon_state_t, ptr %state, ↩
    i32 0, i32 0
  %arrayidx2 = getelementptr inbounds [5 x i32], ptr %x1, i32 0, ↩
    i32 0
  %1 = load i32, ptr %arrayidx2, align 4, !tbaa !11
  %xor = xor i32 %1, %0
  store i32 %xor, ptr %arrayidx2, align 4, !tbaa !11
  %x3 = getelementptr inbounds %struct.ascon_state_t, ptr %state, ↩
    i32 0, i32 0
  %arrayidx4 = getelementptr inbounds [5 x i32], ptr %x3, i32 0, ↩
    i32 3
  %2 = load i32, ptr %arrayidx4, align 4, !tbaa !11
  %x5 = getelementptr inbounds %struct.ascon_state_t, ptr %state, ↩
    i32 0, i32 0
  %arrayidx6 = getelementptr inbounds [5 x i32], ptr %x5, i32 0, ↩
    i32 4
  %3 = load i32, ptr %arrayidx6, align 4, !tbaa !11
  %xor7 = xor i32 %3, %2
  store i32 %xor7, ptr %arrayidx6, align 4, !tbaa !11
  %x8 = getelementptr inbounds %struct.ascon_state_t, ptr %state, ↩
    i32 0, i32 0
  %arrayidx9 = getelementptr inbounds [5 x i32], ptr %x8, i32 0, ↩
    i32 1
  %4 = load i32, ptr %arrayidx9, align 4, !tbaa !11
  %x10 = getelementptr inbounds %struct.ascon_state_t, ptr %state,↩
    i32 0, i32 0
  %arrayidx11 = getelementptr inbounds [5 x i32], ptr %x10, i32 0,↩
    i32 2
  %5 = load i32, ptr %arrayidx11, align 4, !tbaa !11
  %xor12 = xor i32 %5, %4
  store i32 %xor12, ptr %arrayidx11, align 4, !tbaa !11
  %x13 = getelementptr inbounds %struct.ascon_state_t, ptr %state,↩
    i32 0, i32 0
  %arrayidx14 = getelementptr inbounds [5 x i32], ptr %x13, i32 0,↩
    i32 0
  %6 = load i32, ptr %arrayidx14, align 4, !tbaa !11
  store i32 %6, ptr %t0, align 4, !tbaa !11
  %x15 = getelementptr inbounds %struct.ascon_state_t, ptr %state,↩
    i32 0, i32 0
  %arrayidx16 = getelementptr inbounds [5 x i32], ptr %x15, i32 0,↩
    i32 1
  %7 = load i32, ptr %arrayidx16, align 4, !tbaa !11
  store i32 %7, ptr %t1, align 4, !tbaa !11
  %x17 = getelementptr inbounds %struct.ascon_state_t, ptr %state,↩
    i32 0, i32 0
  %arrayidx18 = getelementptr inbounds [5 x i32], ptr %x17, i32 0,↩
    i32 2
  %8 = load i32, ptr %arrayidx18, align 4, !tbaa !11
  store i32 %8, ptr %t2, align 4, !tbaa !11
  %x19 = getelementptr inbounds %struct.ascon_state_t, ptr %state,↩
    i32 0, i32 0
  %arrayidx20 = getelementptr inbounds [5 x i32], ptr %x19, i32 0,↩
    i32 3
  %9 = load i32, ptr %arrayidx20, align 4, !tbaa !11
  store i32 %9, ptr %t3, align 4, !tbaa !11
  %x21 = getelementptr inbounds %struct.ascon_state_t, ptr %state,↩
    i32 0, i32 0
  %arrayidx22 = getelementptr inbounds [5 x i32], ptr %x21, i32 0,↩
    i32 4
  %10 = load i32, ptr %arrayidx22, align 4, !tbaa !11
  store i32 %10, ptr %t4, align 4, !tbaa !11
  %11 = load i32, ptr %t0, align 4, !tbaa !11
  %not = xor i32 %11, -1
  store i32 %not, ptr %t0, align 4, !tbaa !11
  %12 = load i32, ptr %t1, align 4, !tbaa !11
  %not23 = xor i32 %12, -1
  store i32 %not23, ptr %t1, align 4, !tbaa !11
  %13 = load i32, ptr %t2, align 4, !tbaa !11
  %not24 = xor i32 %13, -1
  store i32 %not24, ptr %t2, align 4, !tbaa !11
  %14 = load i32, ptr %t3, align 4, !tbaa !11
  %not25 = xor i32 %14, -1
  store i32 %not25, ptr %t3, align 4, !tbaa !11
  %15 = load i32, ptr %t4, align 4, !tbaa !11
  %not26 = xor i32 %15, -1
  store i32 %not26, ptr %t4, align 4, !tbaa !11
  %x27 = getelementptr inbounds %struct.ascon_state_t, ptr %state,↩
    i32 0, i32 0
  %arrayidx28 = getelementptr inbounds [5 x i32], ptr %x27, i32 0,↩
    i32 1
  %16 = load i32, ptr %arrayidx28, align 4, !tbaa !11
  %17 = load i32, ptr %t0, align 4, !tbaa !11
  %and = and i32 %17, %16
  store i32 %and, ptr %t0, align 4, !tbaa !11
  %x29 = getelementptr inbounds %struct.ascon_state_t, ptr %state,↩
     i32 0, i32 0
  %arrayidx30 = getelementptr inbounds [5 x i32], ptr %x29, i32 0,↩
     i32 2
  %18 = load i32, ptr %arrayidx30, align 4, !tbaa !11
  %19 = load i32, ptr %t1, align 4, !tbaa !11
  %and31 = and i32 %19, %18
  store i32 %and31, ptr %t1, align 4, !tbaa !11
  %x32 = getelementptr inbounds %struct.ascon_state_t, ptr %state,↩
     i32 0, i32 0
  %arrayidx33 = getelementptr inbounds [5 x i32], ptr %x32, i32 0,↩
     i32 3
  %20 = load i32, ptr %arrayidx33, align 4, !tbaa !11
  %21 = load i32, ptr %t2, align 4, !tbaa !11
  %and34 = and i32 %21, %20
  store i32 %and34, ptr %t2, align 4, !tbaa !11
  %x35 = getelementptr inbounds %struct.ascon_state_t, ptr %state,↩
     i32 0, i32 0
  %arrayidx36 = getelementptr inbounds [5 x i32], ptr %x35, i32 0,↩
     i32 4
  %22 = load i32, ptr %arrayidx36, align 4, !tbaa !11
  %23 = load i32, ptr %t3, align 4, !tbaa !11
  %and37 = and i32 %23, %22
  store i32 %and37, ptr %t3, align 4, !tbaa !11
  %x38 = getelementptr inbounds %struct.ascon_state_t, ptr %state,↩
     i32 0, i32 0
  %arrayidx39 = getelementptr inbounds [5 x i32], ptr %x38, i32 0,↩
     i32 0
  %24 = load i32, ptr %arrayidx39, align 4, !tbaa !11
  %25 = load i32, ptr %t4, align 4, !tbaa !11
  %and40 = and i32 %25, %24
  store i32 %and40, ptr %t4, align 4, !tbaa !11
  %26 = load i32, ptr %t1, align 4, !tbaa !11
  %x41 = getelementptr inbounds %struct.ascon_state_t, ptr %state,↩
     i32 0, i32 0
  %arrayidx42 = getelementptr inbounds [5 x i32], ptr %x41, i32 0,↩
     i32 0
  %27 = load i32, ptr %arrayidx42, align 4, !tbaa !11
  %xor43 = xor i32 %27, %26
  store i32 %xor43, ptr %arrayidx42, align 4, !tbaa !11
  %28 = load i32, ptr %t2, align 4, !tbaa !11
  %x44 = getelementptr inbounds %struct.ascon_state_t, ptr %state,↩
     i32 0, i32 0
  %arrayidx45 = getelementptr inbounds [5 x i32], ptr %x44, i32 0,↩
     i32 1
  %29 = load i32, ptr %arrayidx45, align 4, !tbaa !11
  %xor46 = xor i32 %29, %28
  store i32 %xor46, ptr %arrayidx45, align 4, !tbaa !11
  %30 = load i32, ptr %t3, align 4, !tbaa !11
  %x47 = getelementptr inbounds %struct.ascon_state_t, ptr %state,↩
     i32 0, i32 0
  %arrayidx48 = getelementptr inbounds [5 x i32], ptr %x47, i32 0,↩
     i32 2
  %31 = load i32, ptr %arrayidx48, align 4, !tbaa !11
  %xor49 = xor i32 %31, %30
  store i32 %xor49, ptr %arrayidx48, align 4, !tbaa !11
  %32 = load i32, ptr %t4, align 4, !tbaa !11
  %x50 = getelementptr inbounds %struct.ascon_state_t, ptr %state, i32 0, i32 0
  %arrayidx51 = getelementptr inbounds [5 x i32], ptr %x50, i32 0, i32 3
  %33 = load i32, ptr %arrayidx51, align 4, !tbaa !11
  %xor52 = xor i32 %33, %32
  store i32 %xor52, ptr %arrayidx51, align 4, !tbaa !11
  %34 = load i32, ptr %t0, align 4, !tbaa !11
  %x53 = getelementptr inbounds %struct.ascon_state_t, ptr %state, i32 0, i32 0
  %arrayidx54 = getelementptr inbounds [5 x i32], ptr %x53, i32 0, i32 4
  %35 = load i32, ptr %arrayidx54, align 4, !tbaa !11
  %xor55 = xor i32 %35, %34
  store i32 %xor55, ptr %arrayidx54, align 4, !tbaa !11
  %x56 = getelementptr inbounds %struct.ascon_state_t, ptr %state, i32 0, i32 0
  %arrayidx57 = getelementptr inbounds [5 x i32], ptr %x56, i32 0, i32 0
  %36 = load i32, ptr %arrayidx57, align 4, !tbaa !11
  %x58 = getelementptr inbounds %struct.ascon_state_t, ptr %state, i32 0, i32 0
  %arrayidx59 = getelementptr inbounds [5 x i32], ptr %x58, i32 0, i32 1
  %37 = load i32, ptr %arrayidx59, align 4, !tbaa !11
  %xor60 = xor i32 %37, %36
  store i32 %xor60, ptr %arrayidx59, align 4, !tbaa !11
  %x61 = getelementptr inbounds %struct.ascon_state_t, ptr %state, i32 0, i32 0
  %arrayidx62 = getelementptr inbounds [5 x i32], ptr %x61, i32 0, i32 4
  %38 = load i32, ptr %arrayidx62, align 4, !tbaa !11
  %x63 = getelementptr inbounds %struct.ascon_state_t, ptr %state, i32 0, i32 0
  %arrayidx64 = getelementptr inbounds [5 x i32], ptr %x63, i32 0, i32 0
  %39 = load i32, ptr %arrayidx64, align 4, !tbaa !11
  %xor65 = xor i32 %39, %38
  store i32 %xor65, ptr %arrayidx64, align 4, !tbaa !11
  %x66 = getelementptr inbounds %struct.ascon_state_t, ptr %state, i32 0, i32 0
  %arrayidx67 = getelementptr inbounds [5 x i32], ptr %x66, i32 0, i32 2
  %40 = load i32, ptr %arrayidx67, align 4, !tbaa !11
  %x68 = getelementptr inbounds %struct.ascon_state_t, ptr %state, i32 0, i32 0
  %arrayidx69 = getelementptr inbounds [5 x i32], ptr %x68, i32 0, i32 3
  %41 = load i32, ptr %arrayidx69, align 4, !tbaa !11
  %xor70 = xor i32 %41, %40
  store i32 %xor70, ptr %arrayidx69, align 4, !tbaa !11
  %x71 = getelementptr inbounds %struct.ascon_state_t, ptr %state, i32 0, i32 0
  %arrayidx72 = getelementptr inbounds [5 x i32], ptr %x71, i32 0, i32 2
  %42 = load i32, ptr %arrayidx72, align 4, !tbaa !11
  %not73 = xor i32 %42, -1
  %x74 = getelementptr inbounds %struct.ascon_state_t, ptr %state, i32 0, i32 0
  %arrayidx75 = getelementptr inbounds [5 x i32], ptr %x74, i32 0, i32 2
  store i32 %not73, ptr %arrayidx75, align 4, !tbaa !11
  call void @llvm.lifetime.end.p0(i64 4, ptr %t4) #2
  call void @llvm.lifetime.end.p0(i64 4, ptr %t3) #2
  call void @llvm.lifetime.end.p0(i64 4, ptr %t2) #2
  call void @llvm.lifetime.end.p0(i64 4, ptr %t1) #2
  call void @llvm.lifetime.end.p0(i64 4, ptr %t0) #2
  ret void
}

; Function Attrs: nocallback nofree nosync nounwind willreturn memory(argmem: readwrite)
declare void @llvm.lifetime.start.p0(i64 immarg, ptr nocapture) #1

; Function Attrs: nocallback nofree nosync nounwind willreturn memory(argmem: readwrite)
declare void @llvm.lifetime.end.p0(i64 immarg, ptr nocapture) #1

attributes #0 = { nounwind uwtable "no-trapping-math"="true" "stack-protector-buffer-size"="8" "target-cpu"="generic-rv32" "target-features"="+32bit,+a,+c,+d,+f,+m,+relax,-e,-experimental-zca,-experimental-zcb,-experimental-zcd,-experimental-zcf,-experimental-zfa,-experimental-zihintntl,-experimental-ztso,-experimental-zvfh,-h,-save-restore,-svinval,-svnapot,-svpbmt,-v,-xtheadba,-xtheadbb,-xtheadbs,-xtheadcmo,-xtheadcondmov,-xtheadfmemidx,-xtheadmac,-xtheadmemidx,-xtheadmempair,-xtheadsync,-xtheadvdot,-xventanacondops,-zawrs,-zba,-zbb,-zbc,-zbkb,-zbkc,-zbkx,-zbs,-zdinx,-zfh,-zfhmin,-zfinx,-zhinx,-zhinxmin,-zicbom,-zicbop,-zicboz,-zicsr,-zifencei,-zihintpause,-zk,-zkn,-zknd,-zkne,-zknh,-zkr,-zks,-zksed,-zksh,-zkt,-zmmul,-zve32f,-zve32x,-zve64d,-zve64f,-zve64x,-zvl1024b,-zvl128b,-zvl16384b,-zvl2048b,-zvl256b,-zvl32768b,-zvl32b,-zvl4096b,-zvl512b,-zvl64b,-zvl65536b,-zvl8192b" }
attributes #1 = { nocallback nofree nosync nounwind willreturn memory(argmem: readwrite) }
attributes #2 = { nounwind }

!llvm.module.flags = !{!0, !1, !2, !3, !4, !5}
!llvm.ident = !{!6}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 1, !"target-abi", !"ilp32d"}
!2 = !{i32 8, !"PIC Level", i32 2}
!3 = !{i32 7, !"PIE Level", i32 2}
!4 = !{i32 7, !"uwtable", i32 2}
!5 = !{i32 8, !"SmallDataLimit", i32 8}
!6 = !{!"clang version 17.0.0 (git@github.com:Eymay/llvm-project.git cf23cb0fcdfc70aa489332bb12056e53d1385ea4)"}
!7 = !{!8, !8, i64 0}
!8 = !{!"any pointer", !9, i64 0}
!9 = !{!"omnipotent char", !10, i64 0}
!10 = !{!"Simple C/C++ TBAA"}
!11 = !{!12, !12, i64 0}
!12 = !{!"int", !9, i64 0}
```

APPENDIX A.3

1.3 Creating Assembly File From C File

LLVM consists of many flexible libraries which allows user to use different libraries with preffered options. To create RISC-V assembly from c code, Clang and LLC are used with the following commands.
Clang is the C compiler front-end which is mainly used with LLVM back-end. Clang is used in this project to produce LLVM intermediate representation code. Following command produces a .ll file in the current directory.

```
-clang -S -target riscv32-linux-gnu -emit-llvm foo.c
```

-S option provides to run only preprocess and compilation steps.
-target option specifies the 32-bit RISC-V target architecture.
-emit-llvm is for targeting the LLVM back-end.
LLC is the LLVM compiler back-end which converts LLVM intermediate representation (IR) into native machine code for a specific target architecture. Following command produces a .s file for RISC-V architecture in the current directory.

```
llvm-project/build/bin/llc -debug-only=isel -view-isel-dags -mtriple=riscv32 lxr.ll
```

-debug-only=isel option gives the debug informations during the DAG lowering process.
-view-isel-dags option prints the directed acylic graph (DAG) image of the IR code.
-view-sched-dags option can be used instead of -view-isel-dags, if the non scheduled DAG wants to be shown.
-mtriple=riscv32 defines the 32-bits RISC-V target architecture.

APPENDIX A.4

1.4 Adding the Crypt extension to the LLVM

To add our extension to LLVM, a new file named RISCVInstrInfoCrypt.td should be created in ../llvm-project/llvm/lib/Target path and code A.5 should be pasted in this file.

```
// Instruction class templates


let hasSideEffects = 0, mayLoad = 0, mayStore = 0 in
class ALU_rrr<bits<2> funct2, bits<3> funct3, string opcodestr,
       bit Commutable = 0>
  : RVInstR4<funct2, funct3, OPC_OP, (outs GPR:$rd), (ins GPR:$rs1, GPR:$rs2, G
        opcodestr, "$rd, $rs1, $rs2 ,$rs3"> {
  let isCommutable = Commutable;
}

// Instructions


def MLA     : ALU_rrr<0b10, 0b100, "mac">,
        Sched<[WriteIMul, ReadIMul, ReadIMul]>;


def NAXOR     : ALU_rrr<0b11, 0b100, "naxor">,
        Sched<[WriteIMul, ReadIMul, ReadIMul]>;


def SHLXOR  : ALU_rr<0b0011000, 0b111, "shlxor">,
       Sched<[WriteIALU, ReadIALU, ReadIALU]>;


let mayLoad = 1 in{
def LXR  : ALU_rr<0b0011011, 0b101, "lxr">,
       Sched<[WriteIALU, ReadIALU, ReadIALU]>;
}

// Instruction infos


def : Pat<  (add (mul GPR:$src1, GPR:$src2), GPR:$src3),
      (MLA GPR:$src1, GPR:$src2, GPR:$src3)>;

def : Pat<  (xor (and (not GPR:$src1), GPR:$src2), GPR:$src3),
      (NAXOR GPR:$src1, GPR:$src2, GPR:$src3)>;

def : Pat<  (xor (shl GPR:$src1, (i32 1)), GPR:$src2),
      (SHLXOR GPR:$src1, GPR:$src2)>;

def : Pat<  (xor (load GPR:$rs1),(load GPR:$rs2)),
      (LXR GPR:$rs1,GPR:$rs2)>;

```

Code A.5 : RISCVInstrInfoCrypt.td file

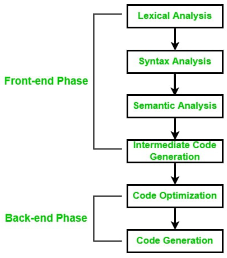
After the file is created properly, code A.6 should be added at the end of the ../llvm-project/llvm/lib/Target/RISCV/RISCVInstrInfo.td file.

```
include "RISCVInstrInfoCrypt.td"
```

Code A.6 **:** Include line

New extension will be ready to use after building the LLVM.

CURRICULUM VITAE



Name Surname : Mehmet Ceylan
Place and Date of Birth :
E-Mail :

EDUCATION :

- **B.Sc.** : Graduation year, Istanbul Technical University, Faculty of Electrical and Electronics, Department of Electrical Engineering
- **M.Sc. (If exists)** : Graduation year, Istanbul Technical University, Faculty of Electrical and Electronics, Department of Electrical Engineering

PROFESSIONAL EXPERIENCE AND REWARDS:

- 1950-1956 Istanbul Technical University at the Central Laboratory of Theoretical Physics.
- 1953 Nobel Prize for Physics
- 1956 Completed Doctorate at Istanbul Technical University

PUBLICATIONS, PRESENTATIONS AND PATENTS ON THE THESIS:

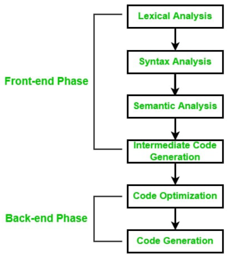
- **Ganapuram S., Hamidov A., Demirel, M. C., Bozkurt E., Kındap U., Newton A.** 2007. Erasmus Mundus Scholar's Perspective On Water And Coastal Management Education In Europe. *International Congress - River Basin Management*, March 22–24, 2007 Antalya, Turkey. (Presentation Instance)

- **Satoğlu, Ş.I., Durmuşoğlu, M. B., Ertay, T. A.** 2010. A Mathematical Model And A Heuristic Approach For Design Of The Hybrid Manufacturing Systems To Facilitate One-Piece Flow, *International Journal of Production Research,* 48(17), 5195–5220. (Article Instance)

- **Chen, Z.** 2013. Intelligent Digital Teaching And Learning All-In-One Machine, Has Projection Mechanism Whose Front End Is Connected With Supporting Arm, And Base Shell Provided With Panoramic Camera That Is Connected With Projector. Patent numarası: CN203102627-U. (Patent Instance)

OTHER PUBLICATIONS, PRESENTATIONS AND PATENTS:

CURRICULUM VITAE



Name Surname : Mehmet Ceylan
Place and Date of Birth :
E-Mail :

EDUCATION :

- **B.Sc.** : Graduation year, Istanbul Technical University, Faculty of Electrical and Electronics, Department of Electrical Engineering
- **M.Sc. (If exists)** : Graduation year, Istanbul Technical University, Faculty of Electrical and Electronics, Department of Electrical Engineering

PROFESSIONAL EXPERIENCE AND REWARDS:

- 1950-1956 Istanbul Technical University at the Central Laboratory of Theoretical Physics.
- 1953 Nobel Prize for Physics
- 1956 Completed Doctorate at Istanbul Technical University

PUBLICATIONS, PRESENTATIONS AND PATENTS ON THE THESIS:

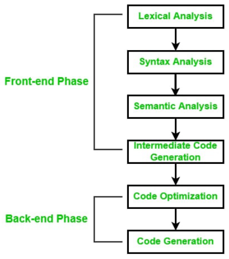
- **Ganapuram S., Hamidov A., Demirel, M. C., Bozkurt E., Kındap U., Newton A.** 2007. Erasmus Mundus Scholar's Perspective On Water And Coastal Management Education In Europe. *International Congress - River Basin Management*, March 22–24, 2007 Antalya, Turkey. (Presentation Instance)

- **Satoğlu, Ş.I., Durmuşoğlu, M. B., Ertay, T. A.** 2010. A Mathematical Model And A Heuristic Approach For Design Of The Hybrid Manufacturing Systems To Facilitate One-Piece Flow, *International Journal of Production Research,* 48(17), 5195–5220. (Article Instance)

- **Chen, Z.** 2013. Intelligent Digital Teaching And Learning All-In-One Machine, Has Projection Mechanism Whose Front End Is Connected With Supporting Arm, And Base Shell Provided With Panoramic Camera That Is Connected With Projector. Patent numarası: CN203102627-U. (Patent Instance)

OTHER PUBLICATIONS, PRESENTATIONS AND PATENTS:

CURRICULUM VITAE



Name Surname	:	Mehmet Ceylan
Place and Date of Birth	:	
E-Mail	:	

EDUCATION	:	
● **B.Sc.**	:	Graduation year, Istanbul Technical University, Faculty of Electrical and Electronics, Department of Electrical Engineering
● **M.Sc. (If exists)**	:	Graduation year, Istanbul Technical University, Faculty of Electrical and Electronics, Department of Electrical Engineering

PROFESSIONAL EXPERIENCE AND REWARDS:

- 1950-1956 Istanbul Technical University at the Central Laboratory of Theoretical Physics.
- 1953 Nobel Prize for Physics
- 1956 Completed Doctorate at Istanbul Technical University

PUBLICATIONS, PRESENTATIONS AND PATENTS ON THE THESIS:

- **Ganapuram S., Hamidov A., Demirel, M. C., Bozkurt E., Kındap U., Newton A.** 2007. Erasmus Mundus Scholar's Perspective On Water And Coastal Management Education In Europe. *International Congress - River Basin Management*, March 22–24, 2007 Antalya, Turkey. (Presentation Instance)

- **Satoğlu, Ş.I., Durmuşoğlu, M. B., Ertay, T. A.** 2010. A Mathematical Model And A Heuristic Approach For Design Of The Hybrid Manufacturing Systems To Facilitate One-Piece Flow, *International Journal of Production Research,* 48(17), 5195–5220. (Article Instance)

- **Chen, Z.** 2013. Intelligent Digital Teaching And Learning All-In-One Machine, Has Projection Mechanism Whose Front End Is Connected With Supporting Arm, And Base Shell Provided With Panoramic Camera That Is Connected With Projector. Patent numarası: CN203102627-U. (Patent Instance)

OTHER PUBLICATIONS, PRESENTATIONS AND PATENTS:

F. M. SURNAME

SUPPORTING CUSTOM INSTRUCTIONS WITH THE LLVM COMPILER FOR RISC-V PROCESSOR

2020