ISTANBUL TECHNICAL UNIVERSITY

ELECTRICAL-ELECTRONICS FACULTY

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

SENIOR DESIGN PROJECT
INTERIM REPORT

Mehmet Eymen ÜNAY
Bora İNAN
Emrecan YİĞİT

ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT

January 2023

ISTANBUL TECHNICAL UNIVERSITY

ELECTRICAL-ELECTRONICS FACULTY

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

SENIOR DESIGN PROJECT
INTERIM REPORT

Mehmet Eymen ÜNAY
(040190218)

Bora İNAN
(040190205)

Emrecan YİĞİT
(040190203)

ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT

Project Advisor:Dr. Tankut AKGÜL

January 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)

TABLE OF CONTENTS

Page

LIST OF TABLES

<u>Page</u>

LIST OF FIGURES

<u>Page</u>

1. INTRODUCTION

1.1 Project Work Plan and Possible Updates

For the last months our main purpose was matching the S-BOX and ROT patterns. During our work, we found out that the LLVM RISC-V backend includes the ROT instruction as an extension. We enabled its flag and set its opcode. We are able to use the ROT instruction. How to enable the ROT instruction is explained in this form. Implementing S-BOX was challenging for us since it is a much more complicated than the other patterns we matched. Tablegen was not sufficent for our algorithm so we decided to match the pattern using c++ during the instruction selection phase. We used RISCVISelDAGToDAG.cpp file. We started to build the algorithm however we found out matching such a complex pattern is not quite reasonable. To implement s-box instruction, using intrinsic functions is a more preffered way because s-box algorithm is not a common pattern and the user may use it in different ways. It is also more practical way for the user so we decided to use intrinsics in s-box instruction.

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 backend 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 on designing application specific instruction set processors. The purpose of this project is to create a compiler backend 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 code for ASIPs. Therefore a need arose for a compiler backend. 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 with 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 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 [**?**]. These phases can be seen in Figure **??**.

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 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

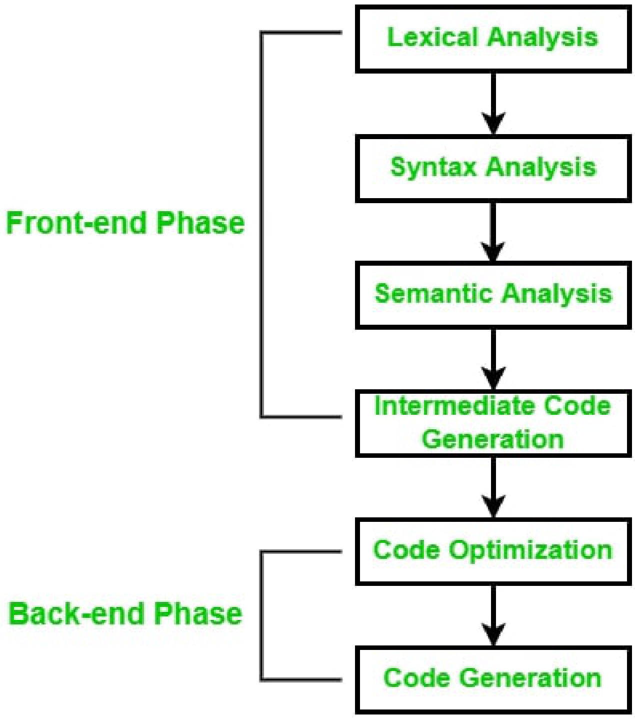


Figure 2.1 : Compiler Stages

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 compiler and 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 frontend also our backend 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 [**?**].

3.1 Parts of the Clang Frontend

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

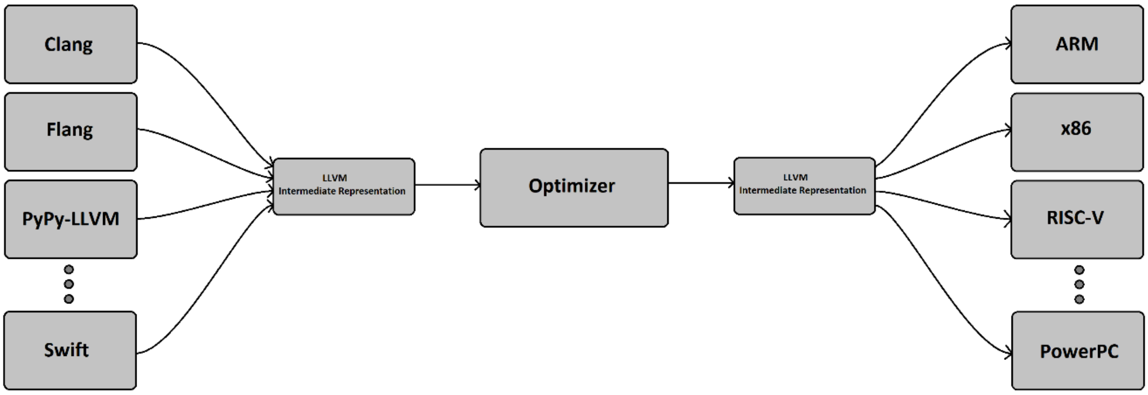


Figure 3.1 : Front-end and Back-end libraries connected by LLVM

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 .ll file which will be used in the backend. 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 Optimizer is a common optimization tool used for every possible source-target combination of a compiler. It takes the output file of CodeGen as input and runs the specified optimizations. The output of the optimizer becomes the input for the RISC-V back-end in our case. LLVM optimizer can be performed with opt [options] [filename] command [**?**].

3.3 Stages of the LLVM RISC-V Back-end

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

3.3.1 Instruction Selection

SelectionDAG is the instruction selector of LLVM backends 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 to the RISC-V Backend.

3.3.2 Scheduling and Formation

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 allocator of RISC-V back-end is responsible for assigning physical registers to virtual registers in the IR. Each target has 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 manipulating the implementation of the selected instructions.

3.3.5 Prolog/Epilog Code Insertion

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

3.3.6 Code Emission

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. 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 backend with TableGen. We explained our addition of MLA instruction thoroughly in Section **??**.

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

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

Listing 4.1 **:** madd.c program

4.1 Clang AST

You can see the Abstract Syntax Tree (AST) produced by Clang in Figure **??**. 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 assignment as an operator. The first argument to this tree is ’a’ and the second argument is the result of multiplication and addition. Figure **??** shows the entire expression tree with the operations multiply and add.

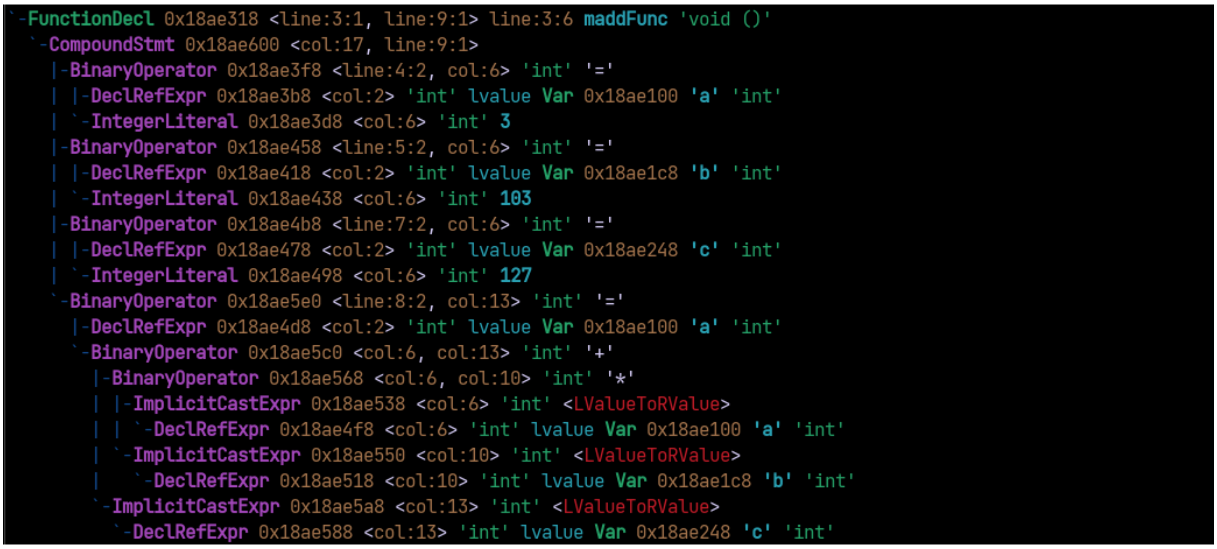
4.2 LLVM IR

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

4.3 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



```
`-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 4.1 : AST generated by Clang

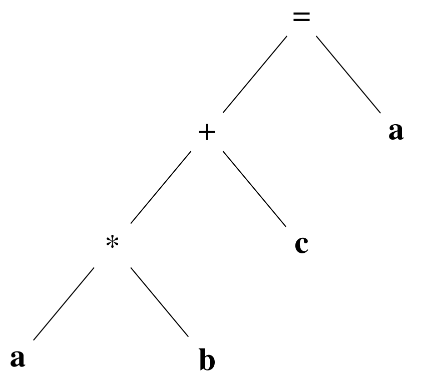
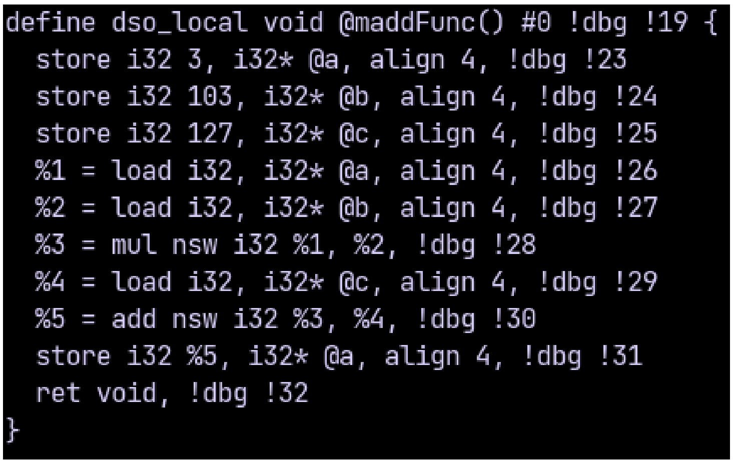


Figure 4.2 : AST of MLA operation



```
define dso_local void @maddFunc() #0 !dbg !19 {
  store i32 3, i32* @a, align 4, !dbg !23
  store i32 103, i32* @b, align 4, !dbg !24
  store i32 127, i32* @c, align 4, !dbg !25
  %1 = load i32, i32* @a, align 4, !dbg !26
  %2 = load i32, i32* @b, align 4, !dbg !27
  %3 = mul nsw i32 %1, %2, !dbg !28
  %4 = load i32, i32* @c, align 4, !dbg !29
  %5 = add nsw i32 %3, %4, !dbg !30
  store i32 %5, i32* @a, align 4, !dbg !31
  ret void, !dbg !32
}
```

Figure 4.3 : LLVM IR file generated at the output of Clang

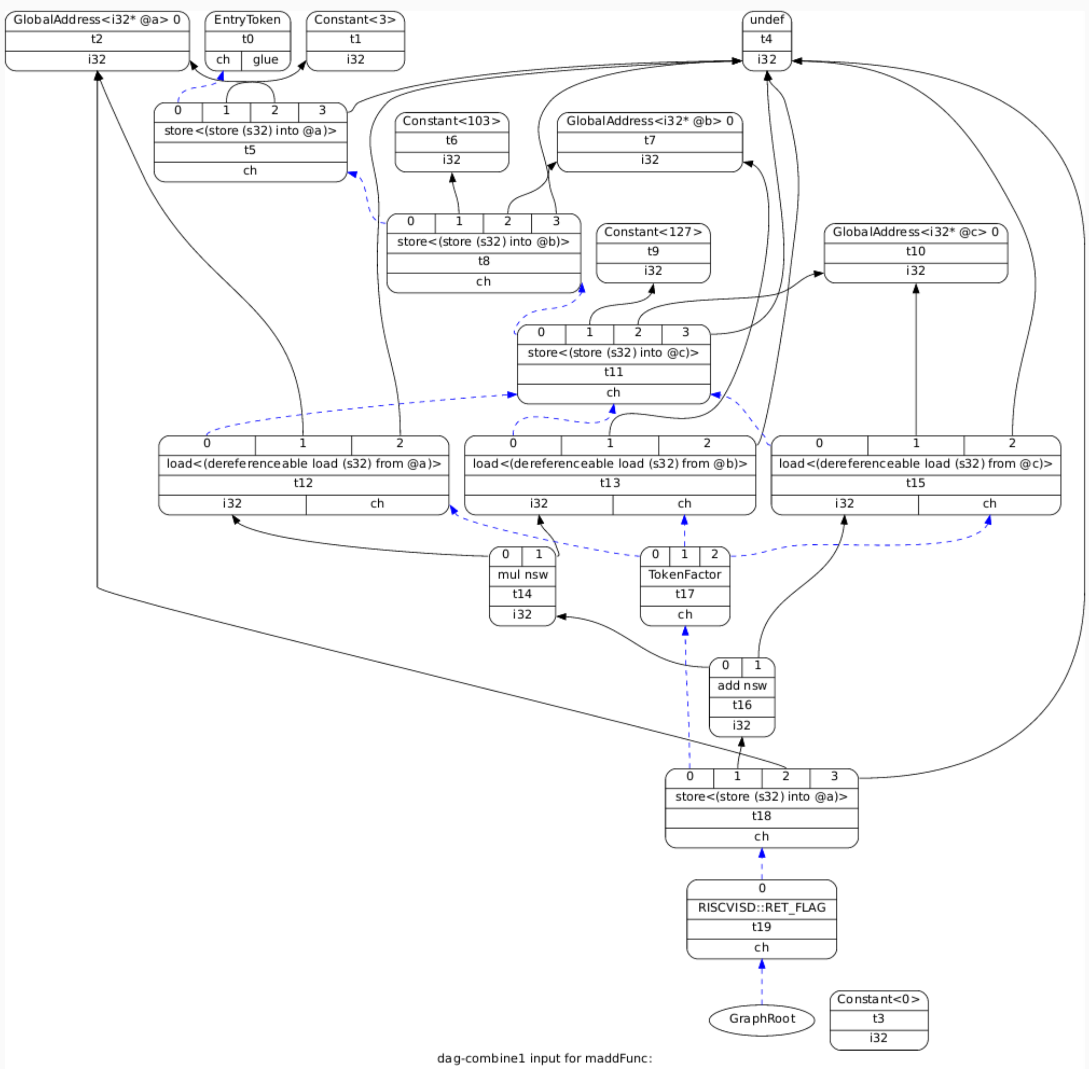


Figure 4.4 : DAG before first optimization pass

7. Register sdsfsdfsdf

4.3.1 First Optimization Pass

Figure **??** 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 **??** 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.

4.3.2 Instruction Legalization

Figure **??** 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 [**?**]. It is the RISCV DAG lowering interface.

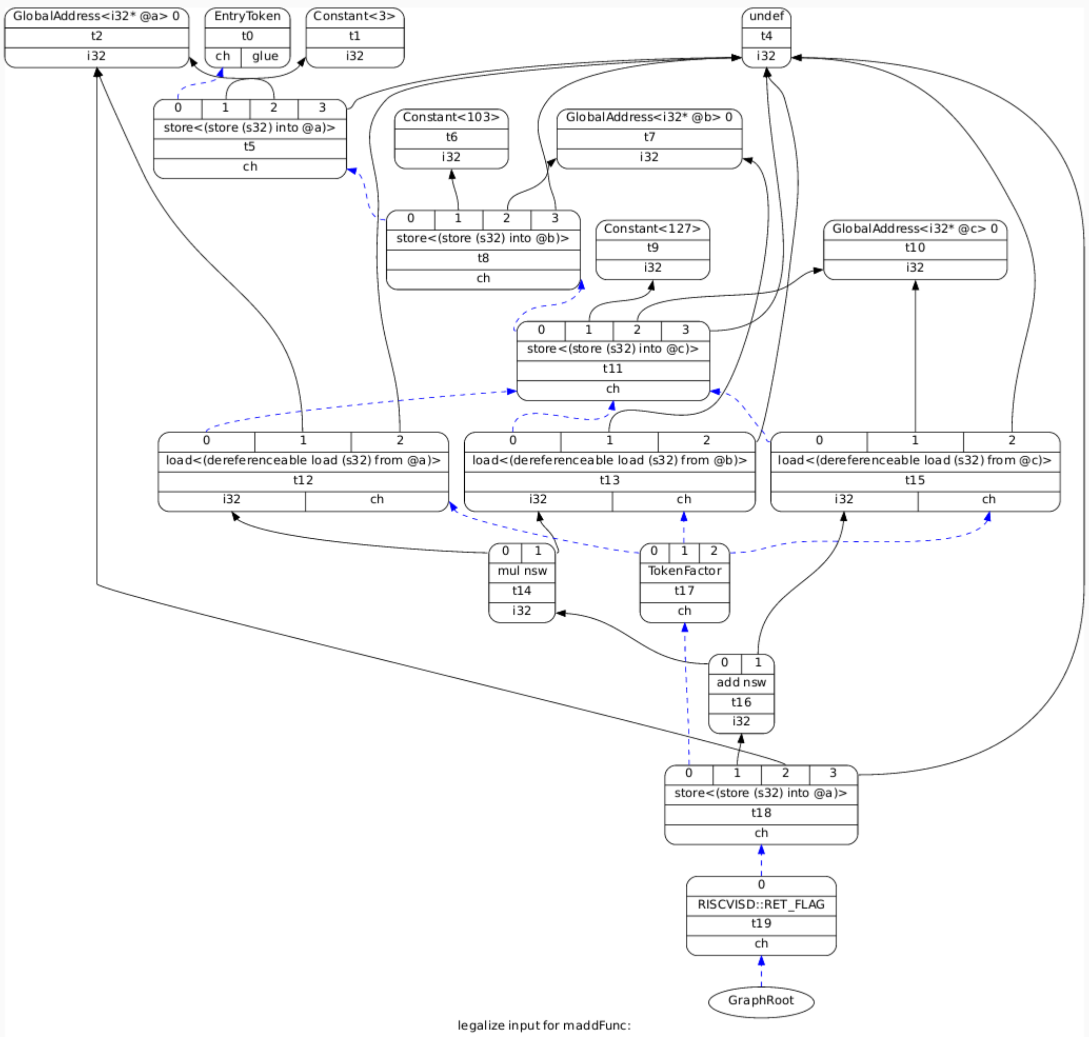


Figure 4.5 : DAG before Legalization

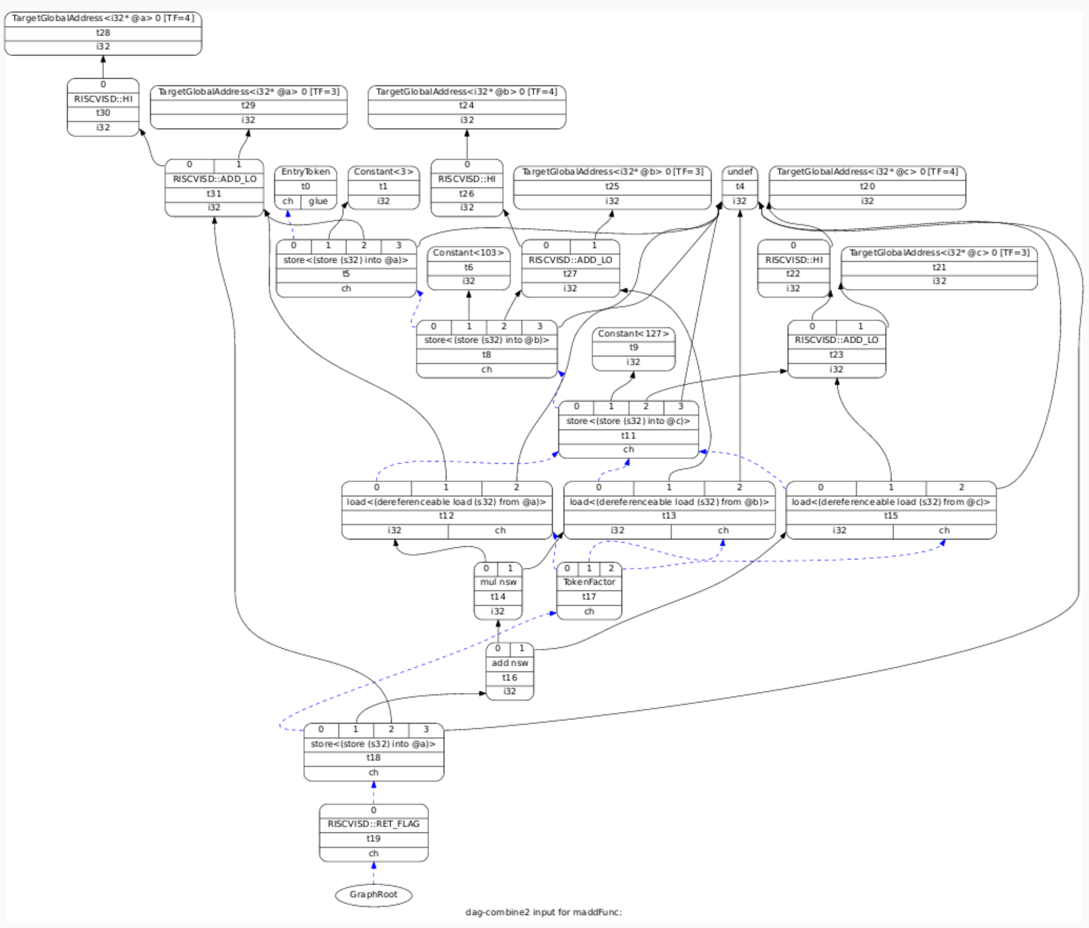

TargetGlobalAddress<i32* @a> 0 [TF=4]
t28
i32
RISCVISD::HI
t30
i32
TargetGlobalAddress<i32* @a> 0 [TF=3]
t29
i32
TargetGlobalAddress<i32* @b> 0 [TF=4]
t24
i32
RISCVISD::ADD_LO
t31
i32
EntryToken
t0
ch
glue
Constant<3>
t1
i32
RISCVISD::HI
t26
i32
TargetGlobalAddress<i32* @b> 0 [TF=3]
t25
i32
undef
t4
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
dag-combine2 input for maddFunc:

Figure 4.6 : DAG before the second optimization

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.

4.3.3 Second Optimization Pass

Figure **??** shows the DAG before the Instruction Selection phase. Comparison of Figure **??** and **??** indicates that the second optimization did not change the DAG. 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.

4.3.4 Instruction Selection

Figure **??** 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.

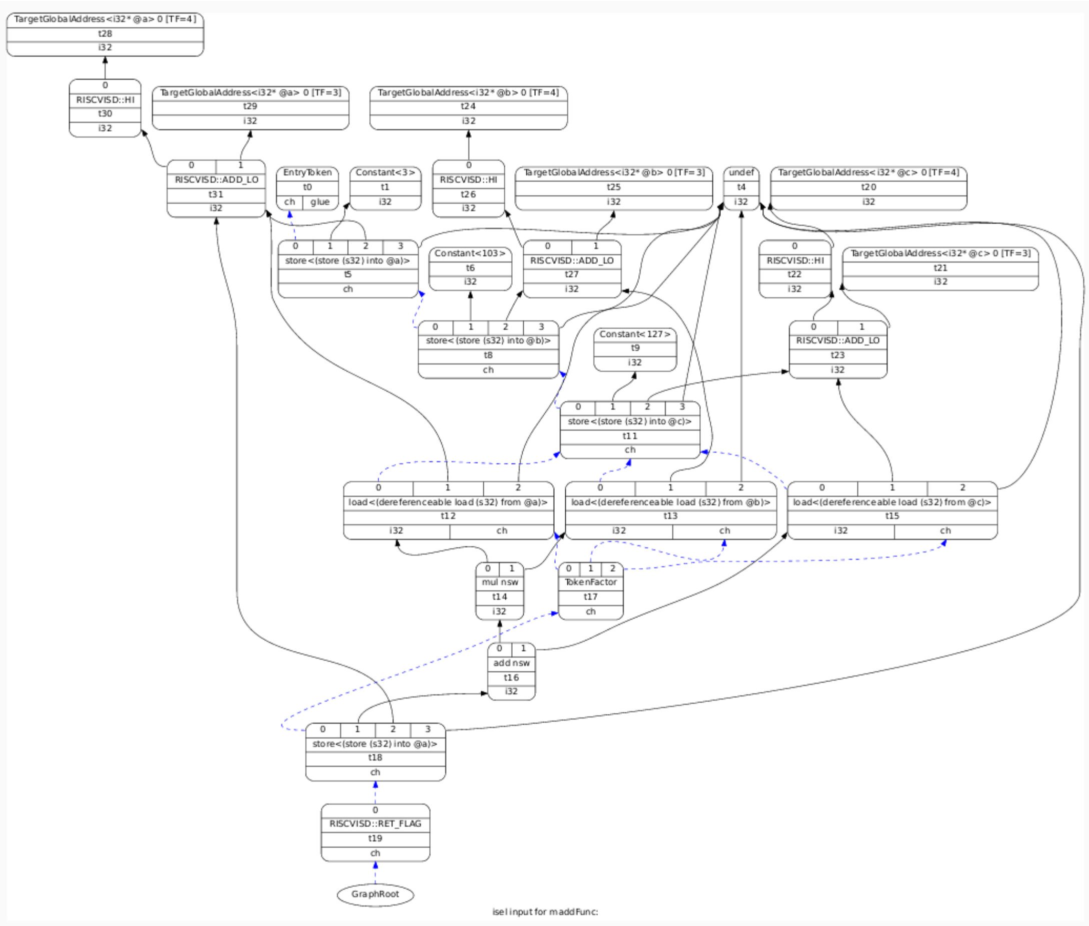

TargetGlobalAddress<i32* @a> 0 [TF=4]
t28
i32
RISCVISD::HI
t30
TargetGlobalAddress<i32* @a> 0 [TF=3]
t29
TargetGlobalAddress<i32* @b> 0 [TF=4]
t24
RISCVISD::ADD_LO
t31
EntryToken
t0
ch
glue
Constant<3>
t1
RISCVISD::HI
t26
TargetGlobalAddress<i32* @b> 0 [TF=3]
t25
undef
t4
TargetGlobalAddress<i32* @c>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

Figure 4.7 : DAG before Instruction Selection

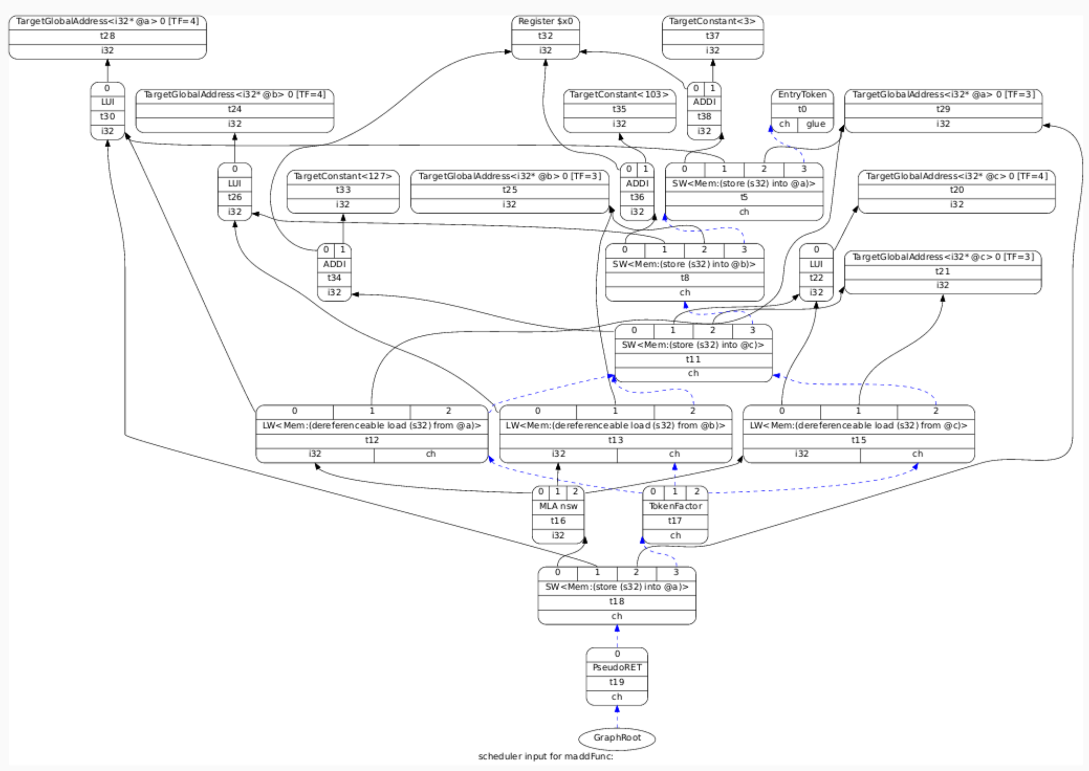

TargetGlobalAddress<i32* @a> 0 [TF=4]
t28
i32
Register $x0
t32
i32
TargetConstant<3>
t37
i32
0
LUI
t30
i32
TargetGlobalAddress<i32* @b> 0 [TF=4]
t24
i32
TargetConstant<103>
t35
i32
0 1
ADDI
t38
i32
EntryToken
t0
ch
glue
TargetGlobalAddress<i32* @a> 0 [TF=3]
t29
i32
0
LUI
t26
i32
TargetConstant<127>
t33
i32
TargetGlobalAddress<i32* @b> 0 [TF=3]
t25
i32
0 1
ADDI
t36
i32
0 1 2 3
SW<Mem:(store (s32) into @a)>
t5
ch
TargetGlobalAddress<i32* @c> 0 [TF=4]
t20
i32
0 1
ADDI
t34
i32
0 1 2 3
SW<Mem:(store (s32) into @b)>
t8
ch
0
LUI
t22
i32
TargetGlobalAddress<i32* @c> 0 [TF=3]
t21
i32
0 1 2 3
SW<Mem:(store (s32) into @c)>
t11
ch
0 1 2
LW<Mem:(dereferenceable load (s32) from @a)>
t12
i32
ch
0 1 2
LW<Mem:(dereferenceable load (s32) from @b)>
t13
i32
ch
0 1 2
LW<Mem:(dereferenceable load (s32) from @c)>
t15
i32
ch
0 1 2
MLA nsw
t16
i32
0 1 2
TokenFactor
t17
ch
0 1 2 3
SW<Mem:(store (s32) into @a)>
t18
ch
0
PseudoRET
t19
ch
GraphRoot
scheduler input for maddFunc:

Figure 4.8 : DAG before Instruction Scheduling

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

4.3.5 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 **??**.

4.3.6 Machine Instruction in SSA Form

The generated Machine Instruction as a result of scheduling is shown in Figure **??**. 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.

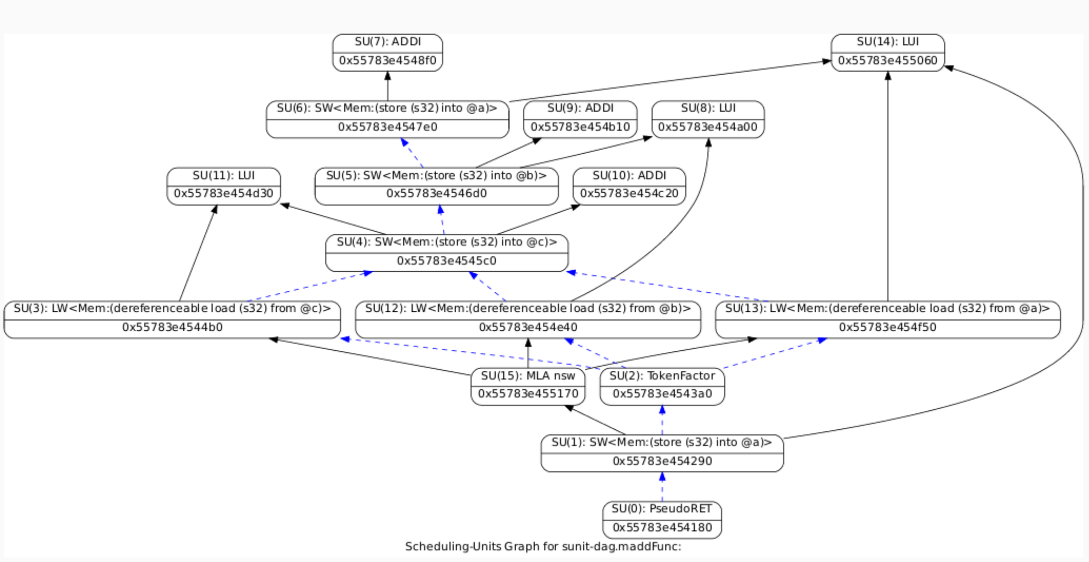
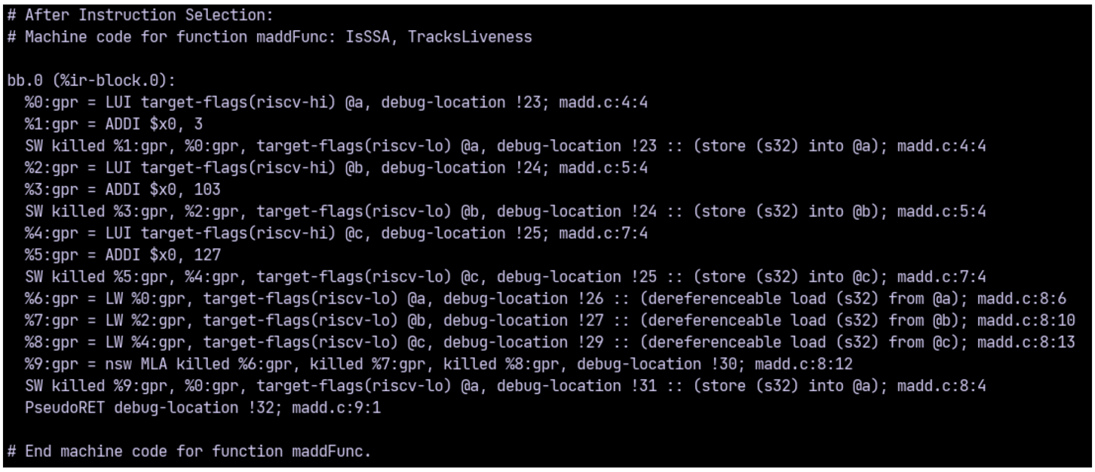

SU(7): ADDI
0x55783e4548f0
SU(14): LUI
0x55783e455060
SU(6): SW<Mem:(store (s32) into @a)>
0x55783e4547e0
SU(9): ADDI
0x55783e454b10
SU(8): LUI
0x55783e454a00
SU(11): LUI
0x55783e454d30
SU(5): SW<Mem:(store (s32) into @b)>
0x55783e4546d0
SU(10): ADDI
0x55783e454c20
SU(4): SW<Mem:(store (s32) into @c)>
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 4.9 : 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 4.10 : Machine Instruction before Register Allocation

4.4 Machine Code Instruction

After register allocation, 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
```

Listing 4.2 **:** madd.s Assembly Output

5. 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, in the near future, 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.

5.1 TableGen Reference

TableGen is a domain-specific language used in 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 code. 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 which 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.

5.2 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. In the end, XOR’s record is defined by putting funct7, funct3 and assembly string manually in a single line with scheduling information added.

5.3 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"
```

Listing 5.1 **:** 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;
}
```

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.

5.4 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 backend. 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 -g 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 backend.
-g option generates the source level debug information.

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.
-mtriple=riscv32 defines the 32 bits RISC-V target architecture.

6. 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. [?] 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. [?] 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.

6.1 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 **??**.

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. [?] 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 **??**. 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.

6.2 RISC-V Base Instructions

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. [?] These formats are given in figure **??**.

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 **??**. By inspecting their formats, we can see which type the instructions belong to. For example, the ADDI instruction is an I-type instruction and XOR is an R-type instruction.

6.3 RISC-V Standard Extensions

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

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 **??**.

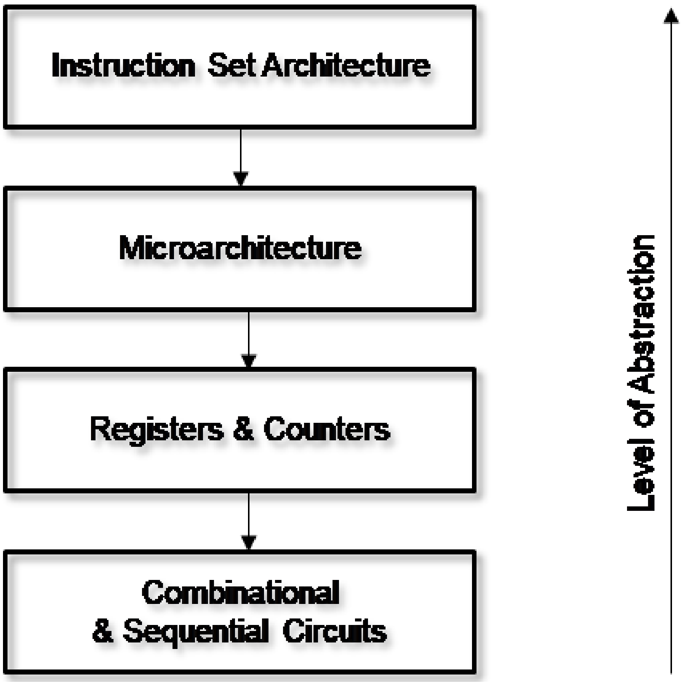
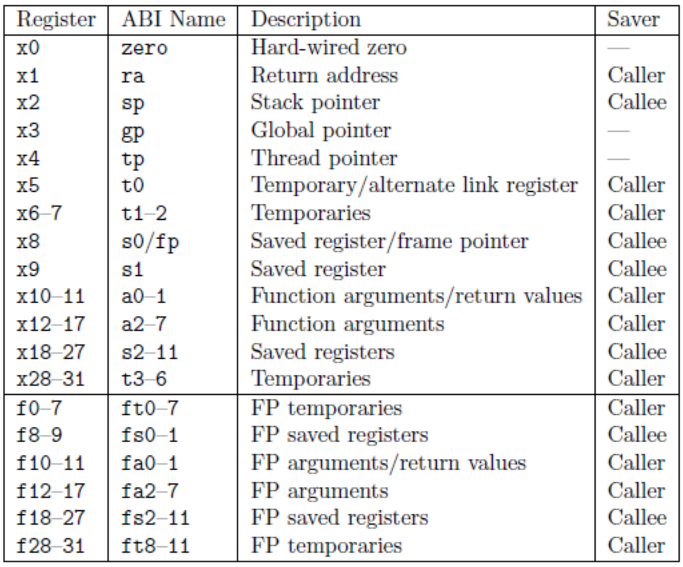


Figure 6.1 : Level of abstraction diagram [?]



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 6.2 : RISC-V registers [?]

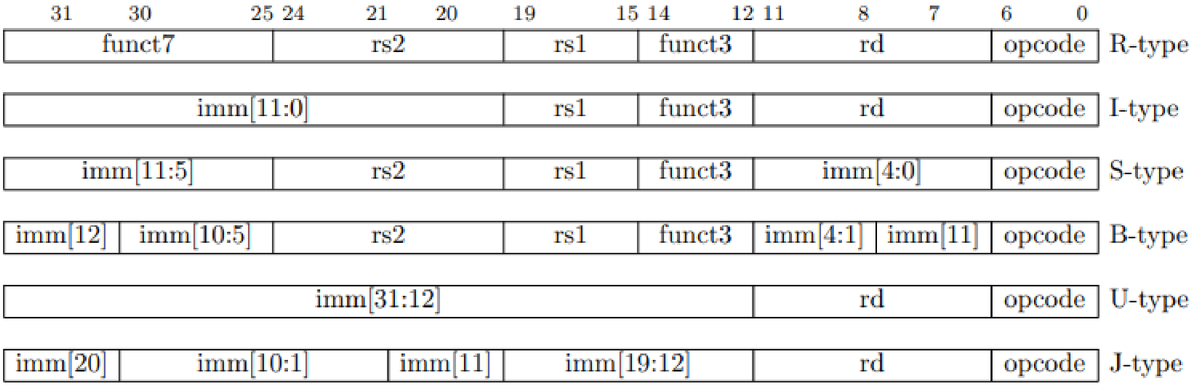
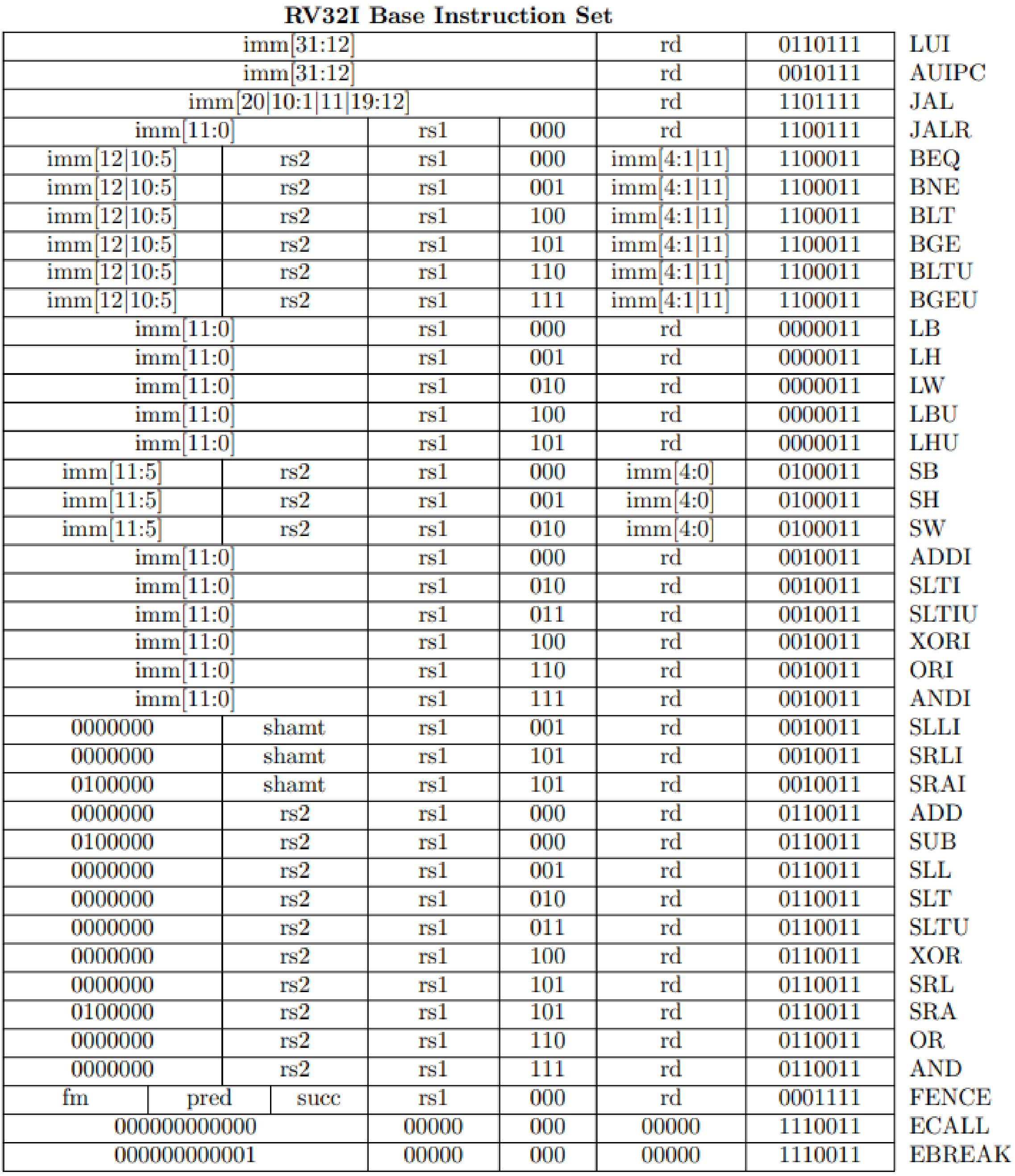


Figure 6.3 : RISC-V base instruction formats [**?**]

Grouping these instructions according to how commonly they are used and 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 **??**.

To give a clearer image of what bit manipulation (B) instructions do, let's explain 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. 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. [**?**] This instruction set extension is also divided into subgroups according to the purpose and similarity of the instructions. The groups are given in figure **??**.

These instructions are not a part of the base instruction set. Therefore, one cannot perform these operations with a single instruction if they use 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.



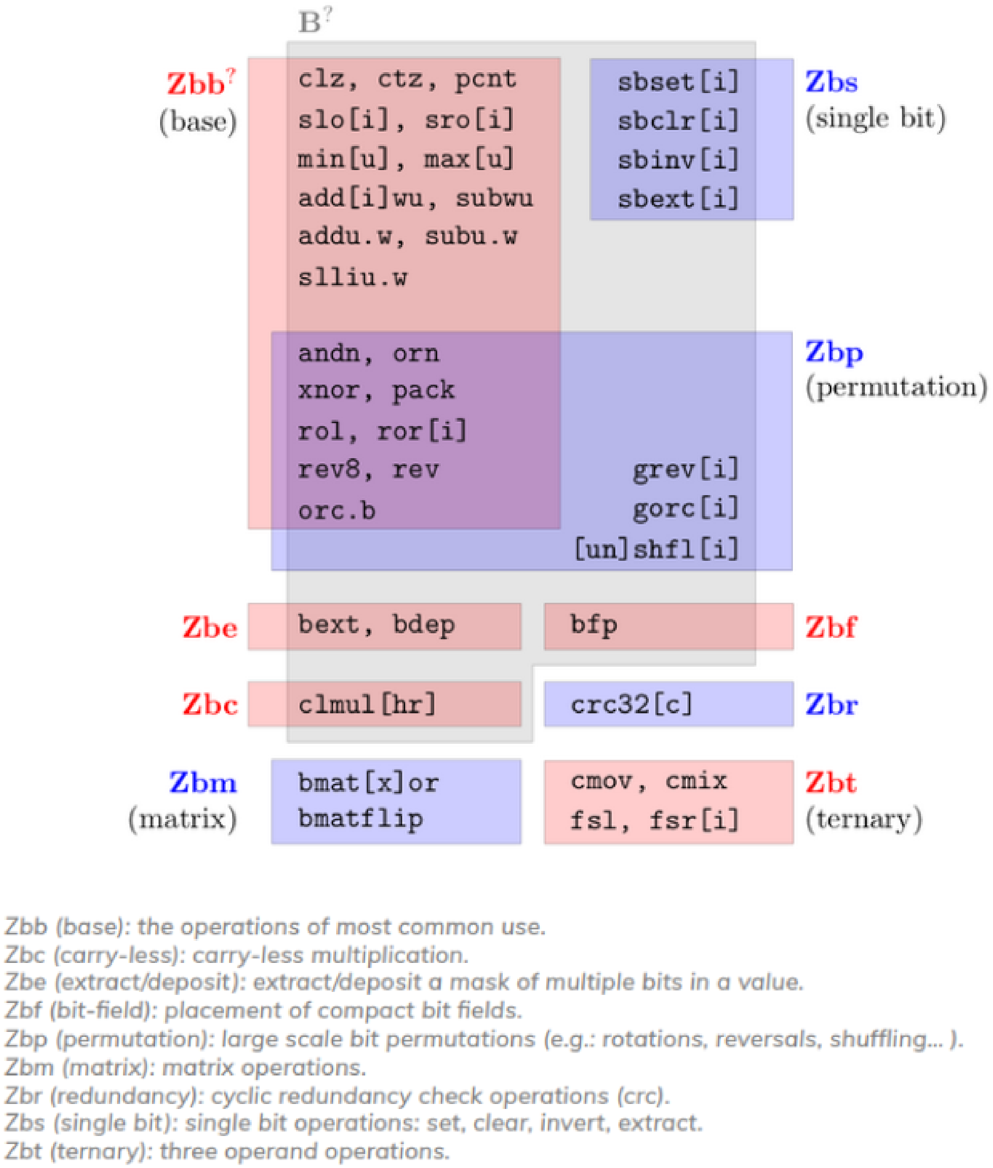
RV32I Base Instruction Set

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 6.4 : RV32I base instruction set [**?**]

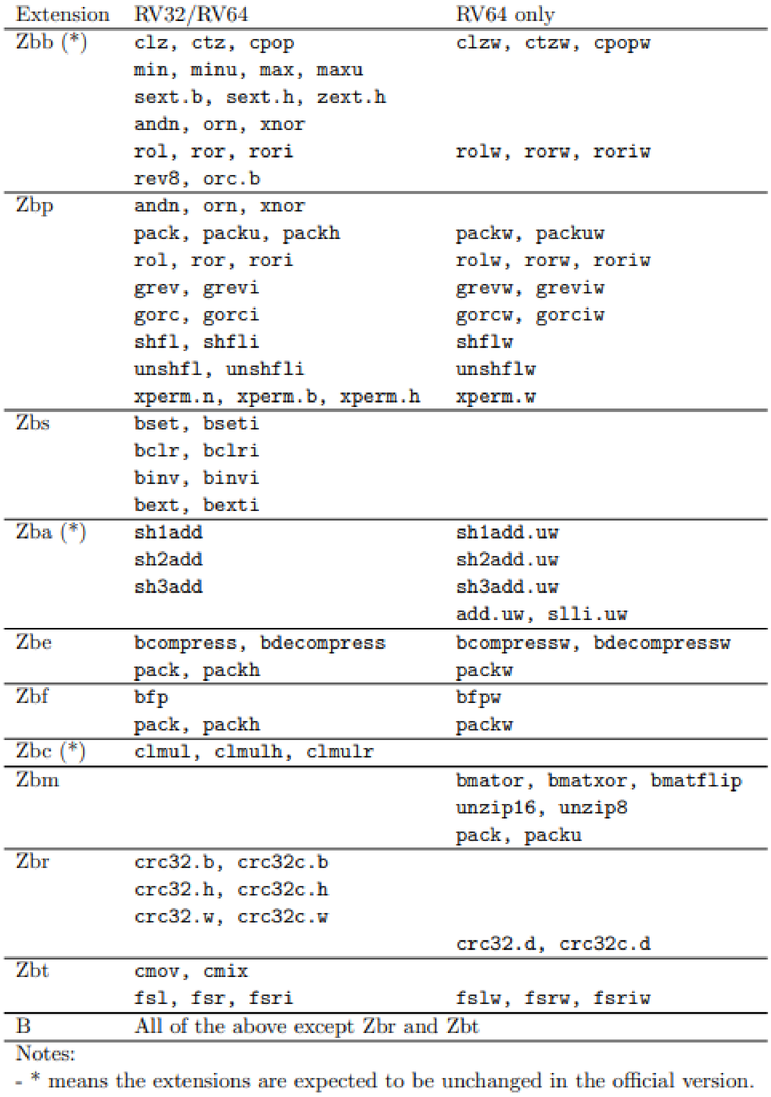
A	Atomic instructions
M	Integer multiplication and division
F	Single precision floating point
D	Double precision floating point
Q	Quad precision floating point
C	Compressed instructions
L	Decimal floating point
B	Bit manipulation
P	Packed SIMD instructions
V	Vector operations
N	User level interrupts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

Figure 6.5 : List of standard extension sets [?]



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 6.6 : Bit manipulation extension groupings [?]



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 6.7 : Bit RV32/RV64 compatibilites and groups [**?**]

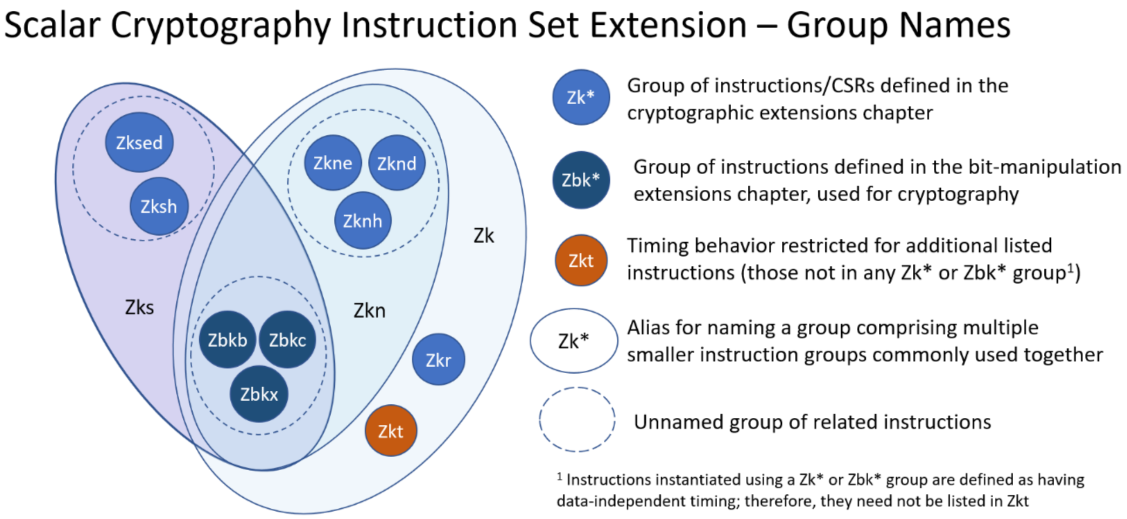

Scalar Cryptography Instruction Set Extension – Group Names
Zksed
Zksh
Zks
Zkne
Zknd
Zknh
Zkn
Zk
Zbkb
Zbkc
Zbkx
Zkr
Zkt
Zk*
Group of instructions/CSRs defined in the cryptographic extensions chapter
Zbk*
Group of instructions defined in the bit-manipulation extensions chapter, used for cryptography
Zkt
Timing behavior restricted for additional listed instructions (those not in any Zk* or Zbk* group1)
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 6.8 : Cryptography extension subgroups [**?**]

7. NEW INSTRUCTIONS

7.1 SHLXOR Instruction

We extended the instruction set by adding the SHLXOR instruction. The purpose of this instruction is to shift the first source operand one bit to the left and then XOR it with the second source operand. Then, the obtained result is stored in the destination register. By adding this instruction, we can perform this operation with a single instruction instead of using shift left and XOR instructions separately, making it more efficient. Let's give an example to make it clearer what this instruction does. RS1 and RS2 are source operands and RD is the output. RS1: 0x0101 RS2: 0xFFFF RD: 0xFDFD 0x0101 is shifted left by one and then XOR'ed with 0xFFFF, giving the result 0xFDFD. As we mentioned, this new instruction requires two source registers and one destination register. Therefore, unlike the MLA instruction, we don't need to create a new class to support it. There is already a class named ALU_rr in the RISCVInstrinfo.td file which has two source and one destination register. Therefore, the new SHLXOR instruction is going to belong to the ALU_rr class. The ALU_rr class is given below.

```
let hasSideEffects = 0, mayLoad = 0, mayStore = 0 in
class ALU_rr<bits<7> funct7, bits<3> funct3, string opcodestr,
bit Commutable = 0>
: RVInstR<funct7, funct3, OPC_OP, (outs GPR:\$rd), (ins GPR:\$rs1, GPR:\$rs2),
opcodestr, "\$rd, \$rs1, \$rs2"> {
let isCommutable = Commutable;
}
```

As we can see, encoding of this type of instructions consists of funct7, funct3, opcode, source registers, and the destination register. The encoding format and the other properties are described in the class. The source registers are described as inputs and the destination register is described as the output. In the RISCVInstrInfoCrypt.td file, we add the definition of the SHLXOR instruction by using the ALU_rr class. In this part, we define funct7, funct3, and the mnemonic of the new instruction as well as the schedulings.

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

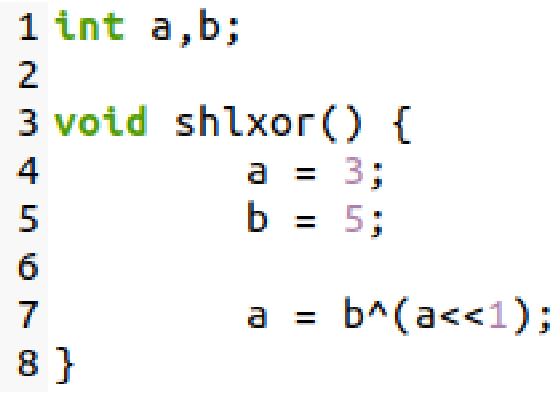
Also in the same file, we define the instruction's pattern. When we examine this definition, we can clearly see what the instruction performs and its pattern. In the inner parentheses, we can see the shifting of the first source operand by one bit. Then, the result of this shifting operation is used as an input for the XOR operation alongside the second source operand.

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

After doing these, we can try it with a simple C code given in **??**. Let's get an assembly output from the C code by running the following commands:

```
clang -S -target riscv32-linux-gnu -emit-llvm -g shlxor.c
$~/CustomInstrLLVM/build/bin/llc -mtriple=riscv32 shlxor.ll$
```

Here, CustomInstrLLVM is the folder that we built llvm in.



```
int a,b;

void shlxor() {
        a = 3;
        b = 5;

        a = b^(a<<1);
}
```

Figure 7.1 : C code for SHLXOR

This C code basically shifts the variable a by one bit and XOR's it with b variable. Then the result is stored in a. We can see that, the register that stores the value a is both the first source register and the destination register. The assembly output is given in figure **??**.

In the assembly output, we can see the SHLXOR instruction. a1 and a2 are the sources and a1 is also the destination as we can see.

7.2 RORI Instruction

One of the Instructions that we worked on is RORI instruction. The purpose of this instruction is to take the operand and rotate it to the right by the amount of the immediate value. This is different from shifting right using an immediate. When shifting a number to the right, the rightmost (least significant) bits are deleted and the leftmost (most significant) bits are filled in with zeros. On the other hand, when a number is rotated right, the least significant bits that are pushed out are not deleted but written into the most significant bits. We want to add add an instruction that performs this operation.

First of all, we tried pattern matching and added the definition and pattern of our new instruction to the InstrInfoCrypt.td file. However, that didn't work and we couldn't see the RORI instruction when we checked our assembly output created by using a simple C code that implements the rotation operation. This is because the pattern can have different combinations and may not match with what we expect. Therefore, the instruction cannot be recognized and we can't see it in the assembly output.

Realizing that, we looked for other options and tried to make use of intrinsics and builtin functions. We added the definitions into the InstrInfoCrypt.td file.

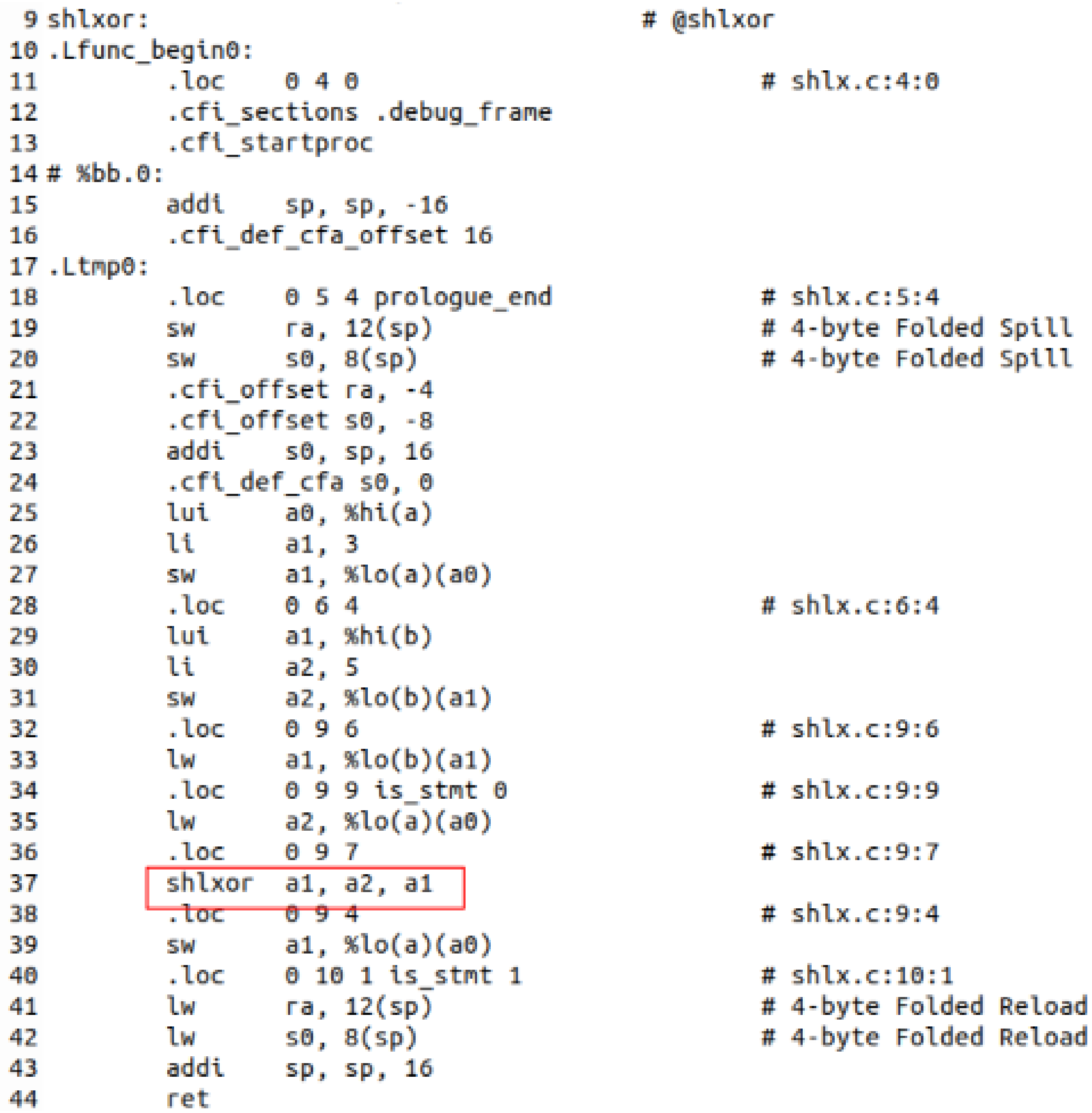
```
def ROTI : ALU_ri<0b101, "roti">;
```

It is ALU_ri type because one of the operands is an immediate value and we only need one register for this instruction.

```
def : Pat<(rotr GPR:\$rs1, simm12:\$imm12),
(ROTI GPR:\$rs1, simm12:\$imm12)>;
```

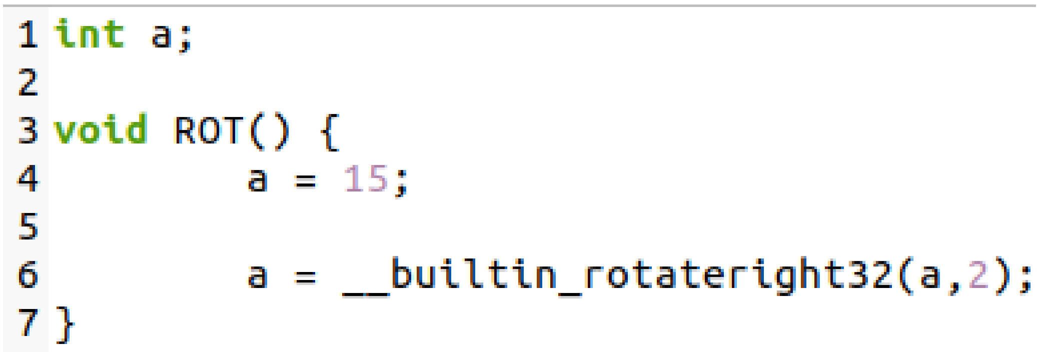
We used rotr here because we wanted to make use of the builtin function __builtin_rotateright32. rotr is defined in the RISVIselLowering.cpp file.

```
if (Subtarget.hasStdExtZbb() || Subtarget.hasStdExtZbkb()) {
if (Subtarget.is64Bit())
setOperationAction({ISD::ROTL, ISD::ROTR}, MVT::i32, Custom);
```



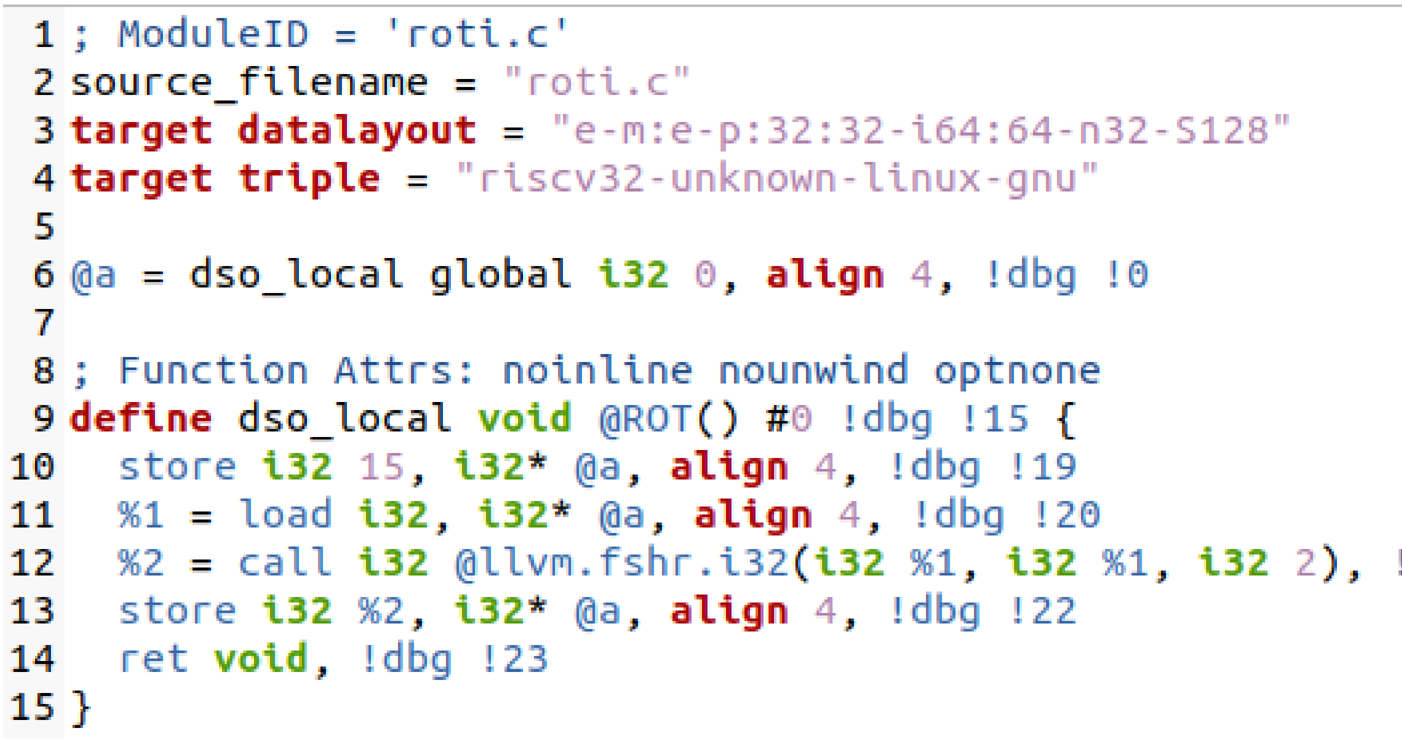
```
shlxor:                                 # @shlxor
.Lfunc_begin0:
        .loc    0 4 0                   # shlx.c:4:0
        .cfi_sections .debug_frame
        .cfi_startproc
# %bb.0:
        addi    sp, sp, -16
        .cfi_def_cfa_offset 16
.Ltmp0:
        .loc    0 5 4 prologue_end      # shlx.c:5:4
        sw      ra, 12(sp)              # 4-byte Folded Spill
        sw      s0, 8(sp)               # 4-byte Folded Spill
        .cfi_offset ra, -4
        .cfi_offset s0, -8
        addi    s0, sp, 16
        .cfi_def_cfa s0, 0
        lui     a0, %hi(a)
        li      a1, 3
        sw      a1, %lo(a)(a0)
        .loc    0 6 4                   # shlx.c:6:4
        lui     a1, %hi(b)
        li      a2, 5
        sw      a2, %lo(b)(a1)
        .loc    0 9 6                   # shlx.c:9:6
        lw      a1, %lo(b)(a1)
        .loc    0 9 9 is_stmt 0         # shlx.c:9:9
        lw      a2, %lo(a)(a0)
        .loc    0 9 7                   # shlx.c:9:7
        shlxor  a1, a2, a1
        .loc    0 9 4                   # shlx.c:9:4
        sw      a1, %lo(a)(a0)
        .loc    0 10 1 is_stmt 1        # shlx.c:10:1
        lw      ra, 12(sp)              # 4-byte Folded Reload
        lw      s0, 8(sp)               # 4-byte Folded Reload
        addi    sp, sp, 16
        ret
```

Figure 7.2 : SHLXOR assembly output



```
int a;

void ROT() {
        a = 15;

        a = __builtin_rotateright32(a,2);
}
```

Figure 7.3 : C code with builtin function for RORI



```
 1 ; ModuleID = 'roti.c'
 2 source_filename = "roti.c"
 3 target datalayout = "e-m:e-p:32:32-i64:64-n32-S128"
 4 target triple = "riscv32-unknown-linux-gnu"
 5
 6 @a = dso_local global i32 0, align 4, !dbg !0
 7
 8 ; Function Attrs: noinline nounwind optnone
 9 define dso_local void @ROT() #0 !dbg !15 {
10   store i32 15, i32* @a, align 4, !dbg !19
11   %1 = load i32, i32* @a, align 4, !dbg !20
12   %2 = call i32 @llvm.fshr.i32(i32 %1, i32 %1, i32 2), !
13   store i32 %2, i32* @a, align 4, !dbg !22
14   ret void, !dbg !23
15 }
```

Figure 7.4 : roti.ll output for roti.c

```
} else {
setOperationAction({ISD::ROTL, ISD::ROTR}, XLenVT, Expand);
}
```

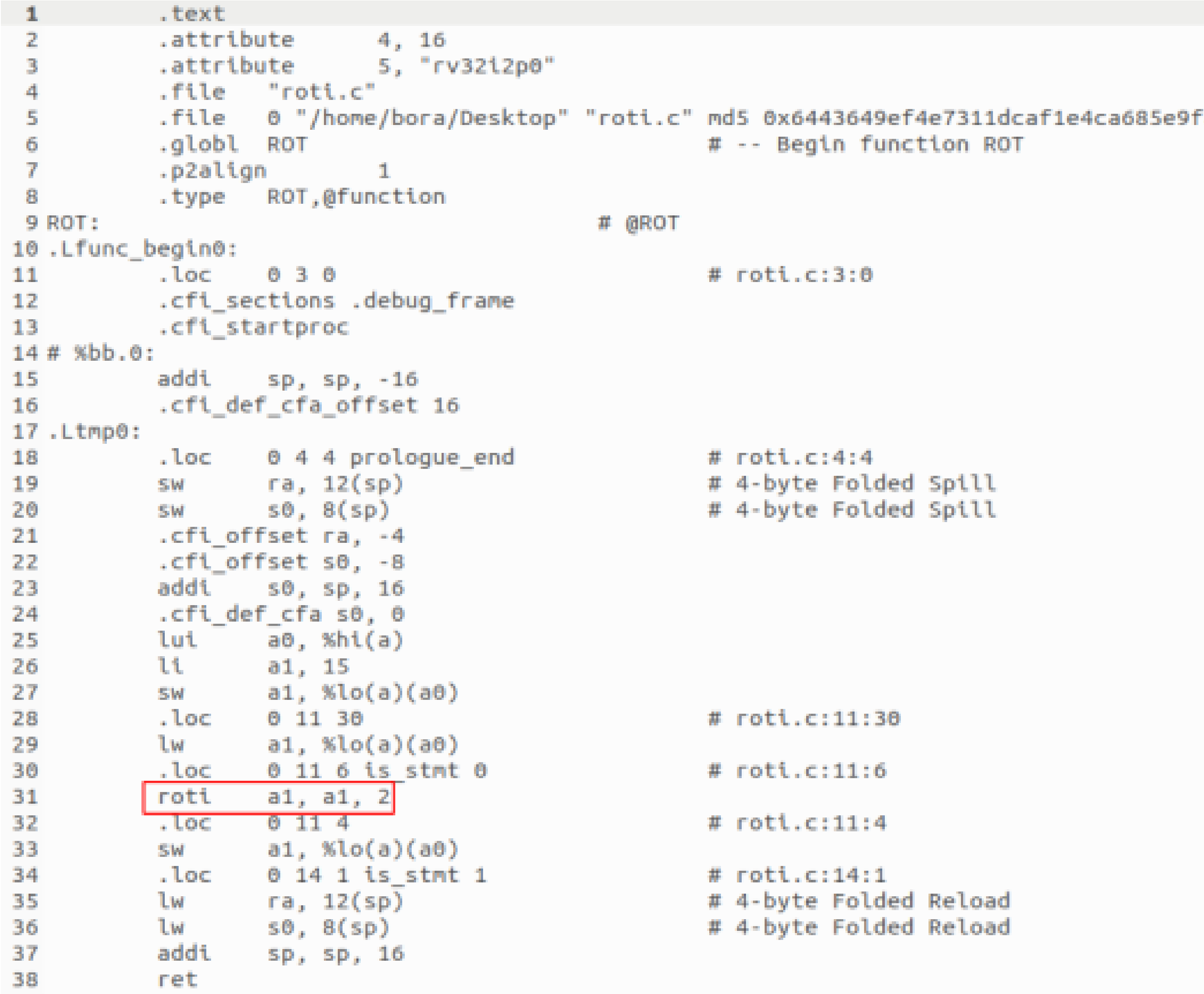
However, we need to change the "Expand" to "Legal" in here otherwise we will not see the ROTI instruction in the assembly output. This way, we are legalizing the action. If we don't do this, we will see the fshr (funnel shift right) intrinsic in the .ll file but ROTI won't make it into the assembly output. After doing these, we can try it with a simple C code. As mentioned before, we used a built in rotate function in the C code given in figure **??**.

This code rotates 15 to the right by 2 bits. We can obtain the .ll file given in figure **??** by running the following command.

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

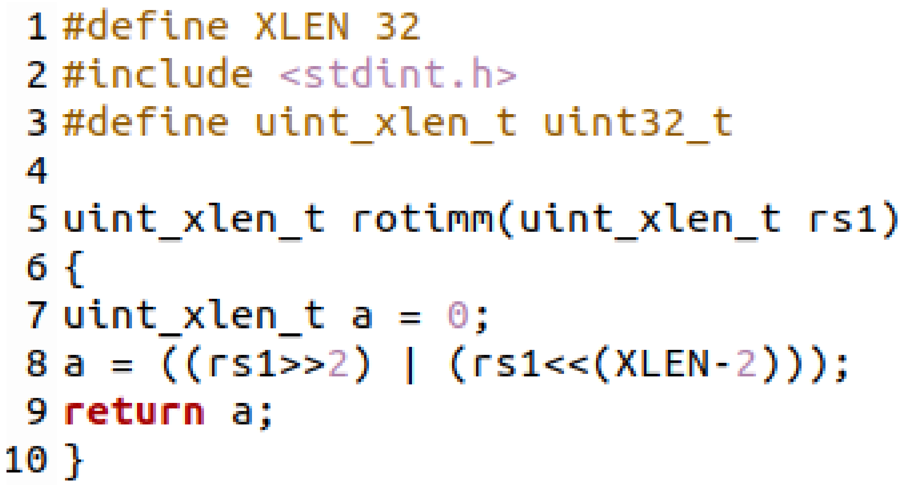
The fshr intrinsic is visible in the twelfth line. After that, the assembly output can be obtained by running the following command. The assembly output is given in figure **??**.

```
$~/CustomInstrLLVM/build/bin/llc -mtriple=riscv32 roti.ll$
```



```
        .text
        .attribute      4, 16
        .attribute      5, "rv32i2p0"
        .file   "roti.c"
        .file   0 "/home/bora/Desktop" "roti.c" md5 0x6443649ef4e7311dcaf1e4ca685e9f
        .globl  ROT                             # -- Begin function ROT
        .p2align        1
        .type   ROT,@function
ROT:                                    # @ROT
.Lfunc_begin0:
        .loc    0 3 0                           # roti.c:3:0
        .cfi_sections .debug_frame
        .cfi_startproc
# %bb.0:
        addi    sp, sp, -16
        .cfi_def_cfa_offset 16
.Ltmp0:
        .loc    0 4 4 prologue_end              # roti.c:4:4
        sw      ra, 12(sp)                      # 4-byte Folded Spill
        sw      s0, 8(sp)                       # 4-byte Folded Spill
        .cfi_offset ra, -4
        .cfi_offset s0, -8
        addi    s0, sp, 16
        .cfi_def_cfa s0, 0
        lui     a0, %hi(a)
        li      a1, 15
        sw      a1, %lo(a)(a0)
        .loc    0 11 30                         # roti.c:11:30
        lw      a1, %lo(a)(a0)
        .loc    0 11 6 is_stmt 0                # roti.c:11:6
        roti    a1, a1, 2
        .loc    0 11 4                          # roti.c:11:4
        sw      a1, %lo(a)(a0)
        .loc    0 14 1 is_stmt 1                # roti.c:14:1
        lw      ra, 12(sp)                      # 4-byte Folded Reload
        lw      s0, 8(sp)                       # 4-byte Folded Reload
        addi    sp, sp, 16
        ret
```

Figure 7.5 : ROTI assembly output



```
1 #define XLEN 32
2 #include <stdint.h>
3 #define uint_xlen_t uint32_t
4
5 uint_xlen_t rotimm(uint_xlen_t rs1)
6 {
7 uint_xlen_t a = 0;
8 a = ((rs1>>2) | (rs1<<(XLEN-2)));
9 return a;
10 }
```

Figure 7.6 : C code for RORI

The ROTI instruction can be seen in the assembly output. As expected, it uses one register as both the destination and the source alongside an immediate value.

This way, we succesfully obtained the ROTI instruction. However, using the builtin function while writing the C code wasn't the prettiest solution. Therefore, we looked further into how we can implement this. After that, we realized that there is a bit manipulation extension for RISCV and what we need was in the RISCVInstrInfoZb.td file. This file contains the instruction extensions for bit manipulations. These instructions operate on the bits of the data and RORI is one of those instructions. However, in order to utilize this extension, we need to add some flags to the command while running clang in order to get an assembly output from the C code we write. The simple C code for RORI is given in figure **??**.

Here, we use 32 as the length because our target is 32 bit and the rotation is implemented in the eighth line. After that, we run the following command with additional flags as mentioned before.

```
$clang --target=riscv32 -O -S rori.c -march=rv32imaczbb$
```

Here, -O defines the level of optimization. -S is used for getting an assembly file as an output. rori.c is the name of our simple C code. -march=rv32imaczbb designates that we want to utilize the zbb subgroup of the bit manipulation extension. The assembly output is given in figure **??**.

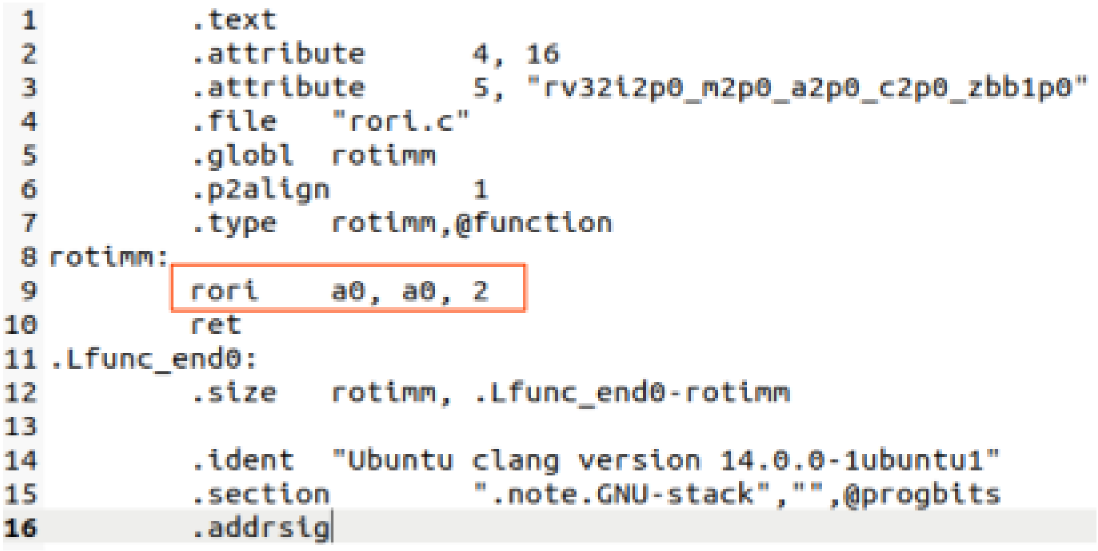
This way, we managed to succesfully obtain RORI instruction in the assembly output.

7.3 NAXOR Instruction

S-box algorithm includes a certain pattern that is used repeatedly. Not and xor pattern is used five times in an s-box cycle. This pattern is lowered into one instruction using TableGen. Definition and specifications of the pattern are added to RISCVInstrInfoCrypt.td file. After matching this pattern 15 rows of the assembly file is reduced to one single instruction. For future works in this field, this pattern can be implemented in a single cycle by adding a new extension to the Ibex core.

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

Since there are three variables in this instruction, custom ALU_rrr class is used which is explained in section 5.3. 11 and 100 numbers are used for funct2 and funct3.



```
        .text
        .attribute      4, 16
        .attribute      5, "rv32i2p0_m2p0_a2p0_c2p0_zbb1p0"
        .file   "rori.c"
        .globl  rotimm
        .p2align        1
        .type   rotimm,@function
rotimm:
        rori    a0, a0, 2
        ret
.Lfunc_end0:
        .size   rotimm, .Lfunc_end0-rotimm

        .ident  "Ubuntu clang version 14.0.0-1ubuntu1"
        .section        ".note.GNU-stack","",@progbits
        .addrsig
```

Figure 7.7 : RORI assembly output

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

This pattern is used 5 times in the rectangular shape of the figure x.x. NAXOR instruction reduces 15 instructions into 5 instructions.

```
    ; 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"

    ; Function Attrs: mustprogress nofree norecurse nosync nounwind willreturn memory(ar
    define dso_local void @sbox(ptr nocapture noundef \%0) local_unnamed_addr #0 {
      \%2 = getelementptr inbounds [5 x i32], ptr \%0, i32 0, i32 4
      \%3 = load i32, ptr \%2, align 4, !tbaa !7
      \%4 = load i32, ptr \%0, align 4, !tbaa !7
      \%5 = xor i32 \%4, \%3
      \%6 = getelementptr inbounds [5 x i32], ptr \%0, i32 0, i32 3
      \%7 = load i32, ptr \%6, align 4, !tbaa !7
      \%8 = xor i32 \%7, \%3
      \%9 = getelementptr inbounds [5 x i32], ptr \%0, i32 0, i32 1
      \%10 = load i32, ptr \%9, align 4, !tbaa !7
      \%11 = getelementptr inbounds [5 x i32], ptr \%0, i32 0, i32 2
      \%12 = load i32, ptr \%11, align 4, !tbaa !7
      \%13 = xor i32 \%12, \%10
      \%14 = xor i32 \%5, -1
      \%15 = xor i32 \%10, -1
      \%16 = xor i32 \%13, -1
      \%17 = xor i32 \%7, -1
      \%18 = xor i32 \%8, -1
      \%19 = and i32 \%10, \%14
      \%20 = and i32 \%12, \%15
      \%21 = and i32 \%7, \%16
      \%22 = and i32 \%3, \%17
      \%23 = and i32 \%5, \%18
      \%24 = xor i32 \%20, \%5
      \%25 = xor i32 \%21, \%10
      \%26 = xor i32 \%13, \%22
      \%27 = xor i32 \%23, \%7
      \%28 = xor i32 \%19, \%8
      store i32 \%28, ptr \%2, align 4, !tbaa !7
      \%29 = xor i32 \%25, \%24
      store i32 \%29, ptr \%9, align 4, !tbaa !7
      \%30 = xor i32 \%24, \%28
      store i32 \%30, ptr \%0, align 4, !tbaa !7
      \%31 = xor i32 \%26, \%27
      store i32 \%31, ptr \%6, align 4, !tbaa !7
      \%32 = xor i32 \%26, -1
      store i32 \%32, ptr \%11, align 4, !tbaa !7
      ret void
    }

    attributes #0 = { mustprogress nofree norecurse nosync nounwind willreturn memory(ar

    !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 (https://github.com/llvm/llvm-project.git e3dd9f7e66fe
    !7 = !{!8, !8, i64 0}
    !8 = !{!"int", !9, i64 0}
    !9 = !{!"omnipotent char", !10, i64 0}
    !10 = !{!"Simple C/C++ TBAA"}
```

Intermediate Representation code input for s-box algorithm.

```
    .text
  .attribute  4, 16
  .attribute  5, "rv32i2p0"
  .file "s-box.c"
  .globl  sbox                            # -- Begin function sbox
  .p2align  1
  .type sbox,@function
sbox:                                   # @sbox
  .cfi_startproc
# \%bb.0:
  lw  a1, 16(a0)
  lw  a2, 0(a0)
  lw  a3, 12(a0)
  lw  a4, 4(a0)
  lw  a5, 8(a0)
  xor a2, a2, a1
  xor a6, a3, a1
  xor a7, a5, a4
  not t0, a2
  not t1, a4
  not t2, a7
  not t3, a3
  not t4, a6
  and t0, a4, t0
  and a5, a5, t1
  and t1, a3, t2
  and a1, a1, t3
  and t2, a2, t4
  xor a2, a2, a5
  xor a4, t1, a4
  xor a1, a7, a1
  xor a3, t2, a3
  xor a5, t0, a6
  sw  a5, 16(a0)
  xor a4, a4, a2
  sw  a4, 4(a0)
  xor a2, a2, a5
  sw  a2, 0(a0)
  xor a3, a3, a1
  sw  a3, 12(a0)
  not a1, a1
  sw  a1, 8(a0)
  ret
.Lfunc_end0:
  .size sbox, .Lfunc_end0-sbox
  .cfi_endproc
                                        # -- End function
  .ident  "clang version 17.0.0 (https://github.com/llvm/llvm-project.git e3dd9f7e66fec2
  .section  ".note.GNU-stack","",@progbits
```

Assembly output without naxor instruction.

```
    .text
  .attribute  4, 16
  .attribute  5, "rv32i2p0"
  .file "s-box.c"
  .globl  sbox                            # -- Begin function sbox
  .p2align  1
  .type sbox,@function
sbox:                                   # @sbox
  .cfi_startproc
# \%bb.0:
  lw  a1, 16(a0)
  lw  a2, 0(a0)
  lw  a3, 12(a0)
  lw  a4, 4(a0)
  lw  a5, 8(a0)
  xor a2, a2, a1
  xor a6, a3, a1
  xor a7, a5, a4
  naxor a5, a4, a5 ,a2
  naxor t0, a7, a3 ,a4
  naxor a1, a3, a1 ,a7
  naxor a3, a6, a2 ,a3
  naxor a2, a2, a4 ,a6
  sw  a2, 16(a0)
  xor a4, t0, a5
  sw  a4, 4(a0)
  xor a2, a2, a5
  sw  a2, 0(a0)
  xor a3, a3, a1
  sw  a3, 12(a0)
  not a1, a1
  sw  a1, 8(a0)
  ret
.Lfunc_end0:
  .size sbox, .Lfunc_end0-sbox
  .cfi_endproc
                                        # -- End function
  .ident  "clang version 17.0.0 (https://github.com/llvm/llvm-project.git e3dd9f7e66fec2
  .section  ".note.GNU-stack","",@progbits
```

Assembly output with naxor instruction. 15 lines of not, and, xor operations are reduced to 5 naxor instructions.

7.4 LXR Instruction

S-box algorithm uses multiple inputs and outputs and they are supposed to be loaded and stored to the registers during implementation. An example pattern is matched using TableGen. Lxr instruction covers an xor operation of two loaded numbers. ALU_rrr class is used to match the pattern since three registers are used in the instruction.

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

ALU_rr class is used and 0011011, 101 numbers are used for opcodes. mayLoad flag is 1 to enable the load instruction in pattern.

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

LXR instruction covers the xor operation of two loaded numbers.

```
    define i32 @foo(ptr \%p1, ptr \%p2) {
        \%a = load i32, ptr \%p1
        \%b = load i32, ptr \%p2
        \%res = xor i32 \%a, \%b
        ret i32 \%res
      }
```

Intermediate Representation code input for lxr algorithm.

```

          .text
          .attribute  4, 16
          .attribute  5, "rv32i2p0"
          .file "lxr.ll"
          .globl  foo                             # -- Begin function foo
          .p2align  2
          .type foo,@function
      foo:                                    # @foo
          .cfi_startproc
      # \%bb.0:
          lw  a0, 0(a0)
          lw  a1, 0(a1)
          xor a0, a0, a1
          ret
      .Lfunc_end0:
          .size foo, .Lfunc_end0-foo
          .cfi_endproc
                                              # -- End function
          .section  ".note.GNU-stack","",@progbits
```

Assembly output without lxr instruction.

```
    .text
  .attribute  4, 16
  .attribute  5, "rv32i2p0"
  .file "lxr.ll"
  .globl  foo                             # -- Begin function foo
  .p2align  2
  .type foo,@function
foo:                                    # @foo
  .cfi_startproc
# \%bb.0:
  lxr a0, a0, a1
  ret
.Lfunc_end0:
  .size foo, .Lfunc_end0-foo
  .cfi_endproc
                                        # -- End function
  .section  ".note.GNU-stack","",@progbits
```

Assembly output with lxr instruction. Two loads and one xor instructions are reduced to lxr instruction.

8. TESTING

LLVM-lit coordinates the testing procedure. The comment lines that start with “RUN” call other programs via LLVM-lit. LLVM-lit also gives the output of a program to another program as an input.

FileCheck, as the name implies, controls the checking process. It basically compares the file and the corresponding lines of the output. It is used with CHECK-* command.

Regression testing is a core part of LLVM because of its size and active development. To make sure newly added features don’t break the already present functionality it is a must to both build functionality and its corresponding tests.

8.1 MC Test

LLVM-MC is an abstracted assembler integrated with the Compiler. [**?**]

Before adding a new instruction, check its byte sequence. LLVM-MC is going to be used to get the output.

Here is a working test file:

```
\# RUN: llvm-mc %s -triple=riscv32 -riscv-no-aliases -show-encoding \
\# RUN:    | FileCheck -check-prefixes=CHECK-ASM,CHECK-ASM-AND-OBJ %s
\# RUN: llvm-mc -filetype=obj -triple=riscv32 < %s \
\# RUN:    | llvm-objdump -M no-aliases -d -r - \
\# RUN:    | FileCheck -check-prefixes=CHECK-ASM-AND-OBJ %s

\# CHECK-ASM-AND-OBJ: shlxor s2, s2, s8
\# CHECK-ASM: encoding: [0x33,0x59,0x89,0x81]
shlxor s2, s2, s8
```

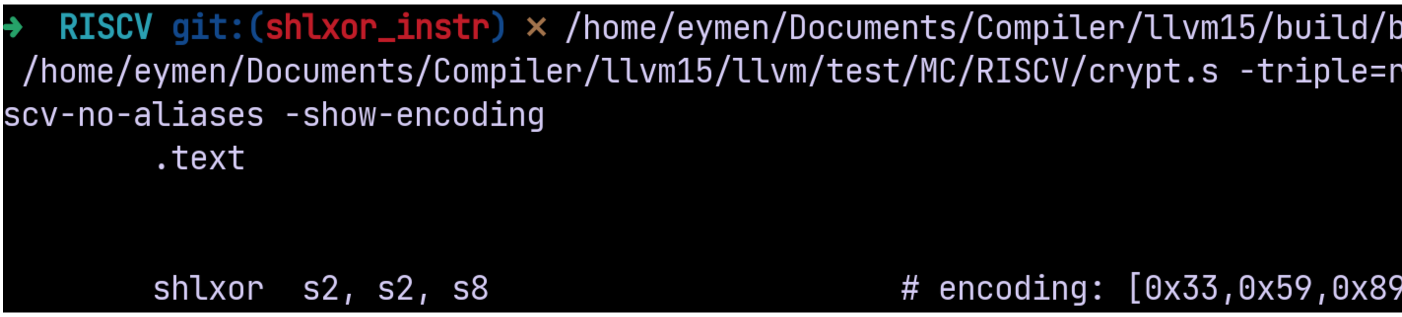
The first RUN command sequence results in the output given in figure **??**.

Here, FileCheck is checking both the Assembly encoding and the string as it was provided with the prefixes: CHECK-ASM, CHECK-ASM-AND-OBJ.

The second RUN sequence with only the llvm-mc part spits out an ELF object which itself isn’t useful. The console output can be seen in figure **??**.

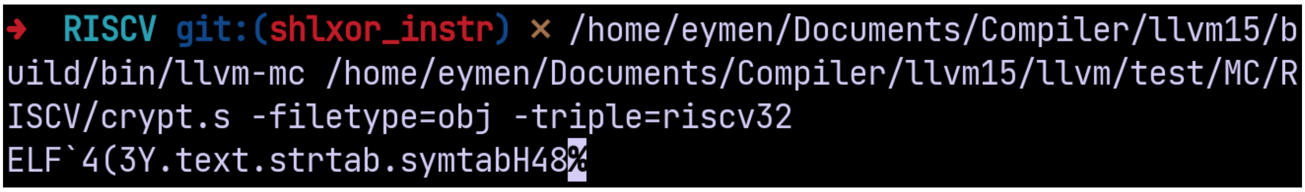
By using the pipe ‘|’ operator like in Shell, we can tell LLVM-lit to feed another program with the output of a program. An example is given in figure **??**.

Since we observed the outputs of the commands, we can make sense of what FileCheck is checking. Here in the second RUN sequence, FileCheck is provided only with



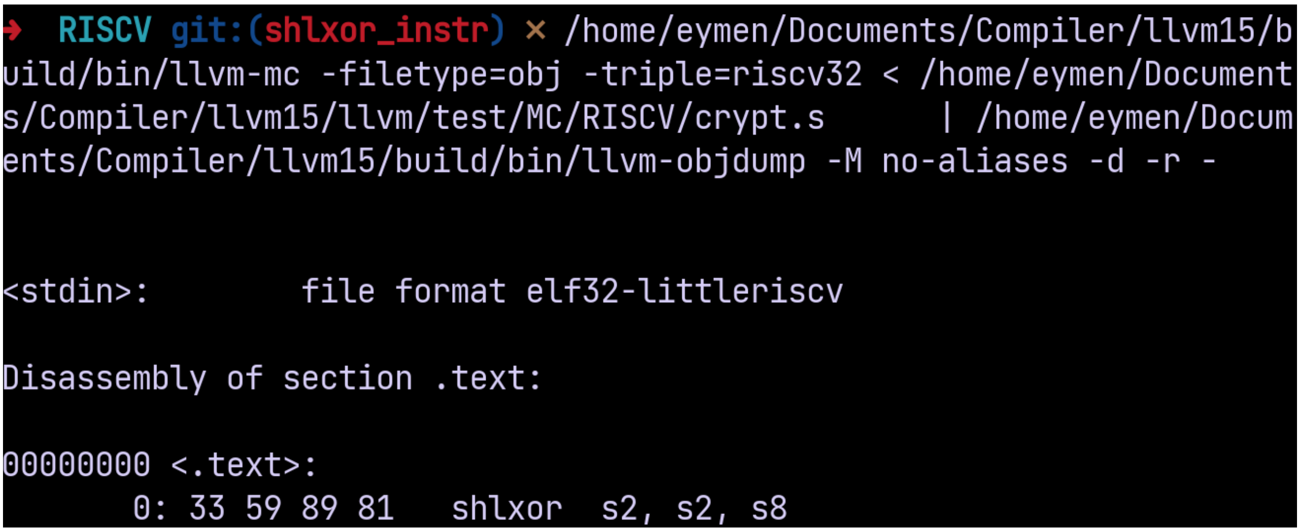
```
➜  RISCV git:(shlxor_instr) ✗ /home/eymen/Documents/Compiler/llvm15/build/b
 /home/eymen/Documents/Compiler/llvm15/llvm/test/MC/RISCV/crypt.s -triple=r
scv-no-aliases -show-encoding
        .text


        shlxor  s2, s2, s8                              # encoding: [0x33,0x59,0x89
```

Figure 8.1 : Result of first "RUN" command



```
➜  RISCV git:(shlxor_instr) ✗ /home/eymen/Documents/Compiler/llvm15/b
uild/bin/llvm-mc /home/eymen/Documents/Compiler/llvm15/llvm/test/MC/R
ISCV/crypt.s -filetype=obj -triple=riscv32
ELF`4(3Y.text.strtab.symtabH48%
```

Figure 8.2 : Result of second "RUN" command



```
➜  RISCV git:(shlxor_instr) ✗ /home/eymen/Documents/Compiler/llvm15/b
uild/bin/llvm-mc -filetype=obj -triple=riscv32 < /home/eymen/Document
s/Compiler/llvm15/llvm/test/MC/RISCV/crypt.s      | /home/eymen/Docum
ents/Compiler/llvm15/build/bin/llvm-objdump -M no-aliases -d -r -


<stdin>:        file format elf32-littleriscv

Disassembly of section .text:

00000000 <.text>:
       0: 33 59 89 81   shlxor  s2, s2, s8
```

Figure 8.3 : Use of pipe operator

CHECK-ASM-AND-OBJ and therefore it does not check the CHECK-ASM line. It seems that the object dump resulted the correct string.

In figure **??** we can observe that the test is passed:

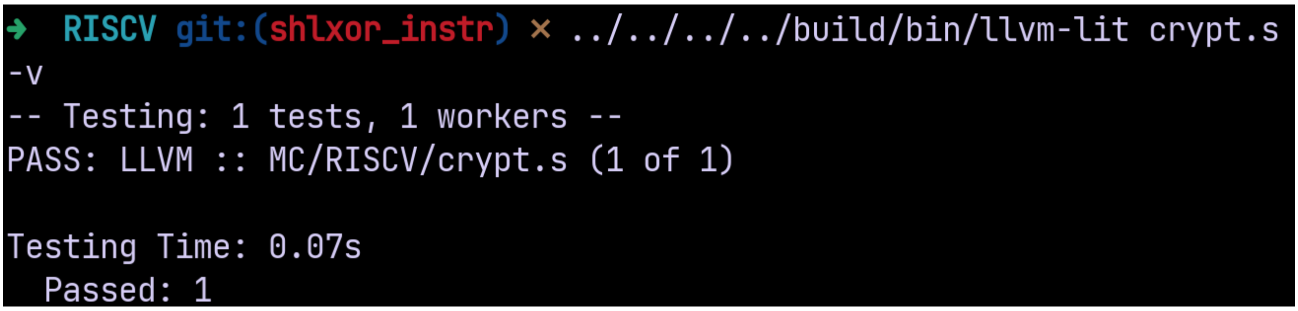
Another complexity LLVM-lit handles is the path of our compiled binaries. The test case is free of the paths and %s placeholders are populated by LLVM-lit.

8.2 Writing the MC Test

LLVM-MC can be run before writing the test to see the encoding or hand-coding is also possible. It would be great if it was automatic like the llc test cases.

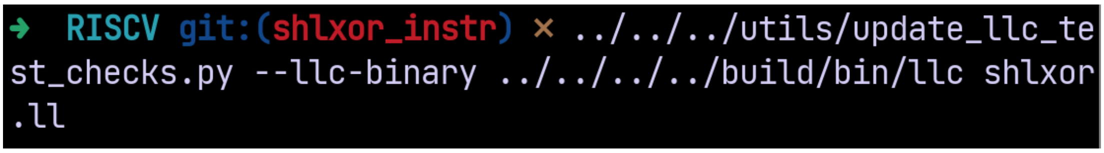
The header of the MC test can be used, if prefixes other than the present one are needed, add them to the –check-prefixes= side and make sure to use them in the file.

8.3 LLC/CodeGen Test



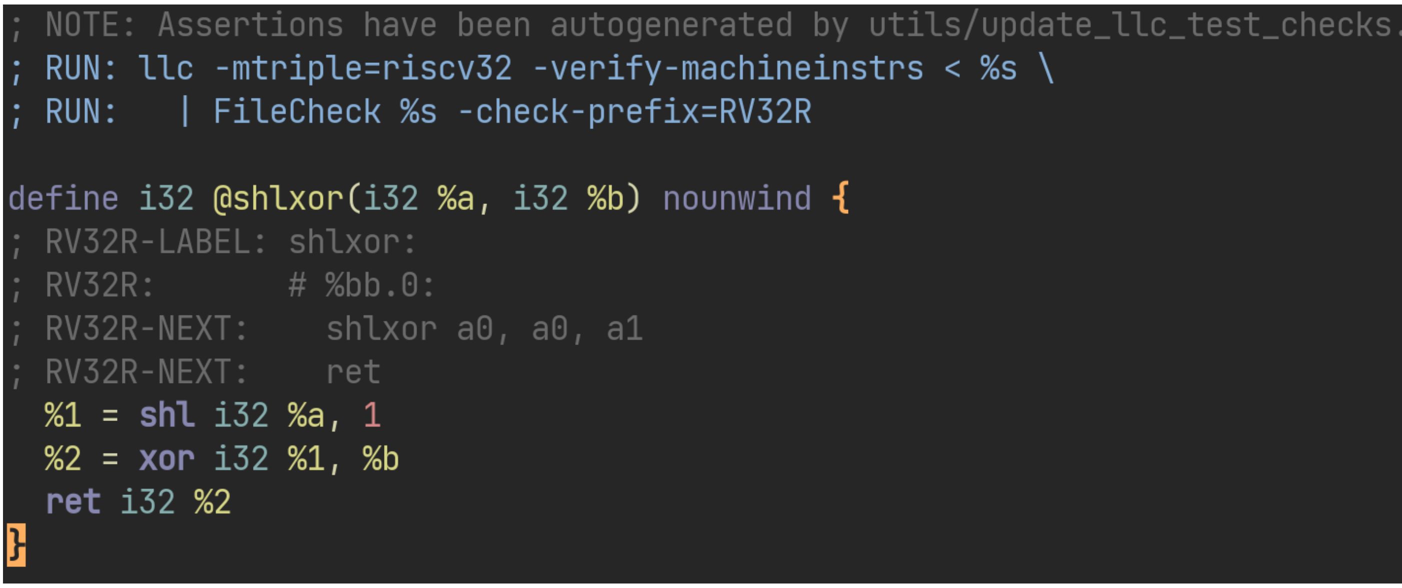
```
➜  RISCV git:(shlxor_instr) ✗ ../../../../build/bin/llvm-lit crypt.s
-v
-- Testing: 1 tests, 1 workers --
PASS: LLVM :: MC/RISCV/crypt.s (1 of 1)

Testing Time: 0.07s
  Passed: 1
```

Figure 8.4 : Output for successfully passed test



```
➜  RISCV git:(shlxor_instr) ✗ ../../../utils/update_llc_test_checks.py --llc-binary ../../../../build/bin/llc shlxor.ll
```

Figure 8.5 : Command for using utility script



```
; NOTE: Assertions have been autogenerated by utils/update_llc_test_checks.
; RUN: llc -mtriple=riscv32 -verify-machineinstrs < %s \
; RUN:   | FileCheck %s -check-prefix=RV32R

define i32 @shlxor(i32 %a, i32 %b) nounwind {
; RV32R-LABEL: shlxor:
; RV32R:       # %bb.0:
; RV32R-NEXT:    shlxor a0, a0, a1
; RV32R-NEXT:    ret
  %1 = shl i32 %a, 1
  %2 = xor i32 %1, %b
  ret i32 %2
}
```

Figure 8.6 : LLC test file

llc tests are more familiar since they check how an .ll produces Assembly-like (MCInst) strings. Instead of the manual checking, this tool can be used and batch-testing can be done.

Here is an example of the LLVM-IR code implemented for the SHLXOR instruction that we want. As it can be seen from its signature, it is a function taking two 32 bit integers and returning one. It takes one input, shifts it left by one and assigns it to a variable %1. %1 is then XOR'ed with the second input and the result is returned.

```
; RUN: llc -mtriple=riscv32 -verify-machineinstrs < %s \
; RUN: | FileCheck %s -check-prefix=RV32R


define i32 @shlxor(i32 %a, i32 %b) nounwind {
%1 = shl i32 %a, 1
%2 = xor i32 %1, %b
ret i32 %2
}
```

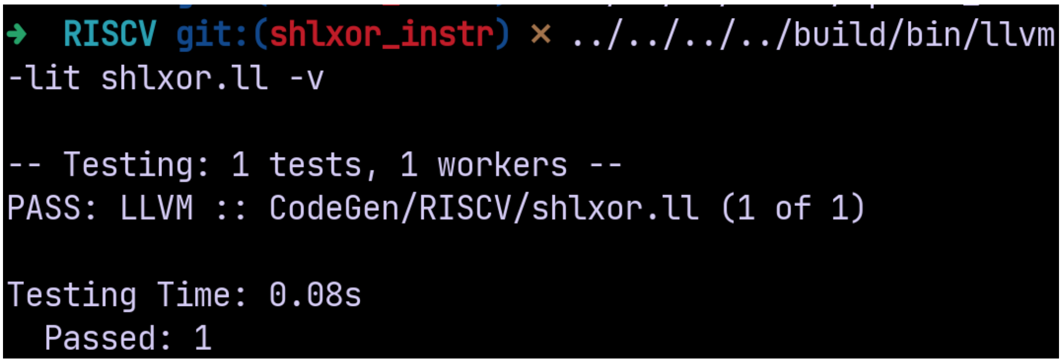
In contrast to MC tests, we can use a utility script as shown in figure **??** to generate the expected result for us.

The script is located at llvm/utils/

The llc test file is populated with FileCheck lines as seen in figure **??**.

We can observe that our newly added SHLXOR instruction is recognized and placed in the check lines. When LLVM-lit is run with the test file, the test is passed as the new instruction is implemented. Output for a successful test is given in figure **??**.

8.4 Running Tests



```
➜  RISCV git:(shlxor_instr) ✗ ../../../../build/bin/llvm
-lit shlxor.ll -v

-- Testing: 1 tests, 1 workers --
PASS: LLVM :: CodeGen/RISCV/shlxor.ll (1 of 1)

Testing Time: 0.08s
  Passed: 1
```

Figure 8.7 : Output for passed .ll test

For running the tests, we should run the following command.

```
build/bin/llvm-lit llvm/test/PATH-OF-FILE -v
```

APPENDICES

APPENDIX A.1 : Installation of Software

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
```

Listing 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
```

Listing 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
```

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

```
ninja llc
```

Listing 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>
```