C language Compiler for SAYAC[1]

Developed By
Arpit Kesharwani
Anand Amar

Under the supervision of
Dr. Srinivas Pinisetty
Dr. Srinivas Boppu

May, 2023

INDIAN INSTITUTE OF TECHNOLOGY BHUBANESWAR

---

1. *SAYAC Microprocessor developed at University of Tehran by -*
   Professor Zain Navabi, Katayoon Basharkhah, Hanieh Tootoonchi Asl

Contents

Chapter 1

Introduction to LLVM

1.1 LLVM Background

LLVM is a popular open-source compiler infrastructure project that is used to build, optimize, and analyze code. It was originally developed at the University of Illinois at Urbana-Champaign, but is now maintained by a global community of developers.

LLVM is an extremely modular implementation of the three-phase compilation process to solve the reuse problem. The idea is that LLVM's core, i.e. the IR and optimizer, are fixed, but the frontend and backend can be changed to retarget the compiler for a different programming language or instruction set. For example, we can compile C/C++ code using Clang (a frontend for LLVM) and the x86 backend to emit code for that instruction set. We could just as easily replace LLVM's backend with one for ARM to compile C/C++ for that instruction set.

One of the key benefits of LLVM is its ability to perform sophisticated code optimizations. The LLVM optimizer uses a variety of techniques to analyze and transform code in order to improve its performance and reduce its memory footprint. This can result in significant speedups for programs, especially when targeting high-performance computing applications or resource-constrained environments.

LLVM uses clang to generate LLVM IR instructions. clang is a frontend framework and it needs to know about for which backend to generate IR files, if no target is specified the system in which code is run is chosen as default target.

1.2 Code Generation at Backend

A LLVM backend compiles IR down to object or assembly code. Each backend targets a single architecture, but possibly multiple instruction sets. For example, LLVM has only one backend for the ARM architecture that can emit code for instruction sets like ARMv6 and ARMv7. Every backend is built on top of LLVM's Target-Independent Code Generator. The code generator is a framework that implements key algorithms like

register allocation. The task of a backend is to configure and adapt that framework to the particular needs of its target instruction set.

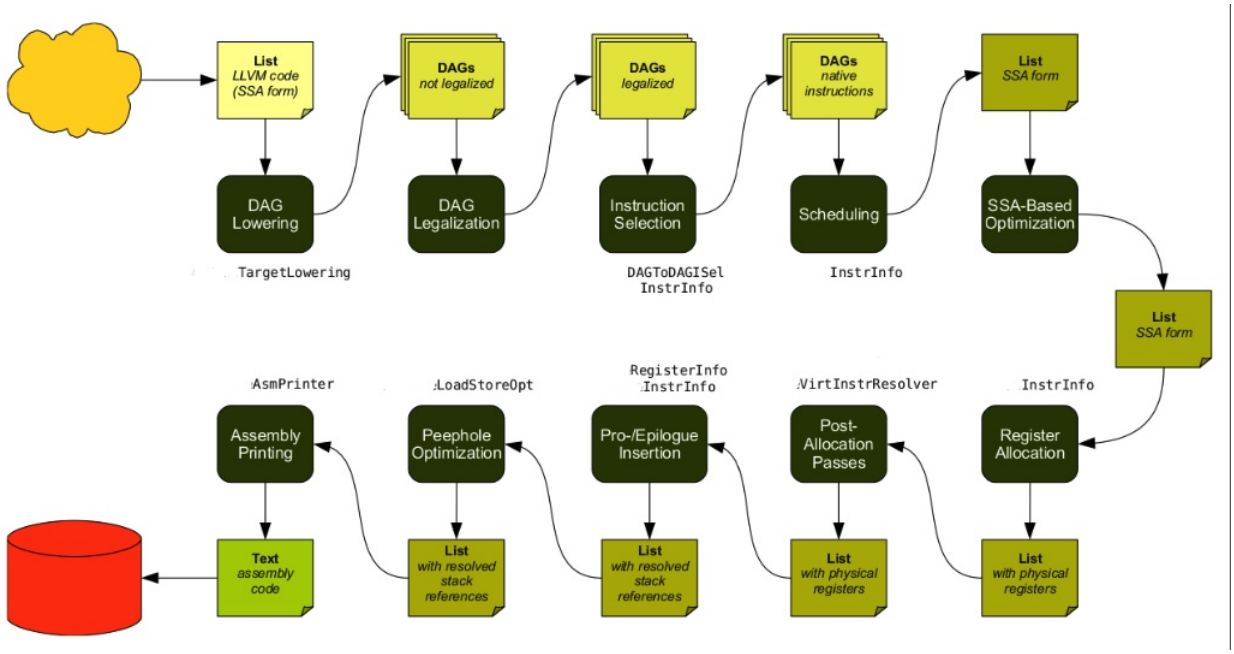

Figure 1.1: Code Generation Phases in Backend

The code generation has the following stages:

1. **Instruction Lowering:** The initial LLVM IR is lowered to instructions that would be supported by the backend. This includes lowering unsupported data types to supported data types in the backend, lowering function calls, lowering global addresses, etc.

2. **Instruction Selection:** The input LLVM IR is mapped to instructions in the target instruction set. At this stage, the program is using an infinite set of virtual registers and abstract references to the function call stack.

3. **Scheduling and Formation:** Determines an ordering of the instructions. There was already an ordering to the instructions at the instruction selection stage, but here we can choose to reorder some of those instructions depending on the register allocation strategy or the instruction latencies.

4. **Register Allocation(RA):** Maps the virtual register to physical registers.

5. **Prolog/Epilog insertion:** Inserts the machine instructions at the beginning (or prolog) and end (or epilog) of every function. These would typically be instructions that extend the stack when entering a function or return to the caller. Abstract stack references are also resolved since the stack size is known at this stage. It also includes saving some of the register's state to the stack and restoring it before returning from the function.

6. **Late Machine Code Optimizations:** The stage performs target specific optimizations.

7. **Code Emission:** Emits the object or assembly code.

All these steps are implemented as machine function passes.

It is recommended to study LLVM CodeGeneration steps from: `https://llvm.org/docs/WritingAnLLVMBackend.html` to get proper idea of each of the steps.

Chapter 2

Architecture Description

LLVM infrastructure has created TableGen language that allows backend writers to specify target description in `.td` files, that is compiled to generate files with `.inc` extension in build directory. These generated files are used by `C++` source code.

2.1 TableGen

TableGen is a tool used in LLVM (Low-Level Virtual Machine) compiler infrastructure that is responsible for generating code for instruction selection, register allocation, and other compiler-related tasks. TableGen is a domain-specific language (DSL) used to define target machine instruction sets, registers, and other platform-specific details, as well as to generate code that can be used by the LLVM framework.

The primary purpose of TableGen is to generate efficient and correct machine code for the target architecture by providing a structured way of specifying machine instructions and their behavior. It uses a declarative language that describes the architecture and allows TableGen to generate code for various stages of the compiler pipeline. This code includes instruction selection patterns, register classes, register allocation information, and other information required for generating efficient code.

TableGen is also used to define the target-specific information for other LLVM tools like LLVM Assembler, LLVM Disassembler, and LLVM MC (Machine Code) tools. It can also generate code for the target-specific components of the LLVM runtime library, such as the JIT (Just-In-Time) compiler and the code generator.

Overall, TableGen plays a crucial role in the LLVM compiler infrastructure and is an essential tool for generating efficient code for a wide range of architectures.

There are various TableGen files required to be created for describing target.

2.1.1 SAYAC.td

The `SAYAC.td` file wires together all other .td files and provides information about which part of the backend description is present in which file, as well as any other target-specific

information(like number of processors, etc.). It is used in LLVM backend to generate target-specific code.

2.1.2 SAYACRegisterInfo.td

The `SAYACRegisterInfo.td` file is used in the LLVM backend to describe the register usage and allocation strategy for the target architecture. It defines the target-specific register classes and their properties, such as the number of registers in the class, the preferred register allocation order, etc..

The register allocation strategy is also defined in this file, which determines how virtual registers are mapped to physical registers. This includes defining the rules for register spilling and restoring, as well as the heuristics used to prioritize register allocation decisions.

The mapping of register names to their corresponding ABI names are also defined in this file, thus it provides a meaning of all registers in the architecture (which one is stack pointer, which is used for return, etc.).

2.1.3 SAYACInstrFormats.td

`SAYACInstrFormats.td` file is used in the LLVM backend to describe the instruction formats for the target architecture. It defines the layout of the machine instructions and their operands, as well as any restrictions on their usage. This information is used by the code generator to produce efficient machine code for the target architecture.

Examples of instruction formats include fixed-width instructions, variable-length instructions, and instructions with complex operand encoding.

2.1.4 SAYACInstrInfo.td

The `SAYACInstrInfo.td` file is a part of the LLVM backend and is responsible for defining the instruction semantics of the target architecture. It contains the instruction set architecture (ISA) description of the target, which includes the specification of each instruction, its encoding, and its behavior. This file contains the instruction definitions in TableGen format, which are then used to generate the corresponding `C++` code for the instructions.

This file is used in the LLVM backend to describe the instruction selection process. It defines the instruction selection patterns for each instruction, which are used by the LLVM code generator to select the appropriate machine instruction during code generation. It also defines the register classes used by the target architecture. It also defines various operands that are used in the Instructions like 4 bit Immediate which should be zero extended, etc.

Apart from defining Instruction set architecture (ISA), the file defines few pseudo instructions which are used for easy conversion of LLVM IR codes and they are later converted to their complex equivalent through `C++` codes written in `SAYACInstrInfo.cpp` file.

The file even covers identifying complex patterns like frameIndex in LLVM IR instructions and mapping them to `C++` functions defined in `SAYACISelDAGToDAG.cpp` which extracts the numeric offset of instruction from the start of frame and then the instruction selection occurs further.

2.1.5 SAYACCallingConv.td

`SAYACCallingConv.td` file in LLVM is used to define the calling convention for the target architecture. The calling convention specifies how function arguments are passed between the caller and the callee, how the return value is passed back to the caller, which registers state should be saved while invoking functions, and how the function's stack frame is managed.

The `SAYACCallingConv.td` file defines the layout of the function stack frame, including the size and alignment of the stack, the layout of function arguments and local variables, and the calling convention used for function calls. It also defines the register usage conventions, including which registers are used for argument passing, return values, and callee-saved registers.

2.1.6 SAYACSchedule.td

`SAYACSchedule.td` file in LLVM is used to define the scheduling model for the target. The scheduling model is used by code generation to order the instructions in an optimal way. Defining a scheduling model improves the performance of the generated code, but it is not necessary for code generation. Therefore, only a placeholder for the model is defined.

These files are important part of the target description for LLVM and is used by the code generator to produce efficient code for the target architecture. By defining these files LLVM can generate optimized code for a wide range of target architectures without requiring manual optimization by the developer.

TableGen is a simple language and hence cannot describe many intricacies of the backend, therefore we require `C++` files to specify them. Work has been ongoing in LLVM community to allow `.td` files to completely specify target descriptions without requiring `C++` files, but that goal is far from being achieved.

Chapter 3

Register Description

3.1 Describing Register Information

The register definition for SAYAC is present in `SAYACRegisterInfo.td` file. At first, the following `SAYACReg` class is defined. It inherits from internalized `Register` class defined in *llvm/include/llvm/Target/Target.td*.

```
let Namespace = "SAYAC" in {
    class SAYACReg<bits<4> Enc, string n, list<string> alt =[]> : Register<n> {
      let HWEncoding{15-4} = 0;
      let HWEncoding{3-0} = Enc;
      let AltNames = alt;
    }

    def ABIRegAltName: RegAltNameIndex ;
}
```

The arguments for this class are:

- The encoding for the register `Enc` which is a 4-bit integer of type `bits<4>`. This uniquely identifies any one of 16 registers in Register File of SAYAC.
- A string `n` indicating the human-readable name of the register like `r0`, `r1`, etc.
- An optional list of alternative human-readable names for this register. For example, `r2` in SAYAC is used as the stack pointer (`sp`).

Next, we find the following register definitions in the file. Each line of code is a TableGen record that defines a single register. The records inherit from `SAYACReg` class .

```
def R0  : SAYACReg<0,  "r0",   ["zero"]>;
def R1  : SAYACReg<1,  "r1",   ["ra"]>;
```

```
def R2  : SAYACReg<2,  "r2",   ["sp"]>;
def R3  : SAYACReg<3,  "r3",   ["fp", "s0"]>;
.....
```

Next, General Purpose Register (`GPR`) Register Class is defined.

```
def GPR : RegisterClass<"SAYAC", [i16], 16,
                            (add
                            (sequence "R%u", 7, 11),
                            (sequence "R%u", 12, 14),
                            (sequence "R%u", 3, 6),
                            (sequence "R%u", 0, 2),
                            R15
                            )>;
```

It inherits from the `RegisterClass` class which takes four argument.

- The namespace: `SAYAC`.
- A list with the data types supported by the registers in this class. This is a list because registers in some architectures can operate with multiple types.
- The register alignment when these are stored or loaded from memory.
- A DAG indicating the previously defined registers in this class. The `GPR` DAG has an `ADD` mode with 6 child `sequence` nodes. A `sequence` is an operation that takes a string format argument along with start and end values for the sequence. Each value in the sequence is replaced in the format to generate an element. The DAG lists the registers in the order that the register allocator must prioritize their use.

3.2 ABI names

The ABI name of a register refers to the name or identifier assigned to a register by the Application Binary Interface (ABI) of a target architecture. Each target architecture typically has its own ABI that specifies register usage, parameter passing conventions, stack layout, calling conventions, and other aspects of binary interfaces. The ABI name of a register is an abstract name assigned to a physical or virtual register that represents its role and purpose within the ABI.

The ABI names and their description for SAYAC Registers are as follows

Register	ABI Name	Description
R0	zero	hardwired zero
R1	ra	return address
R2	sp	stack pointer
R3	s0/fp	saved register 0 / frame pointer
R4	s1	saved register 1
R5	s2	saved register 2
R6	s3	saved register 3
R7	a0	function argument 0 / return value 0
R8	a1	function argument 1 / return value 1
R9	a2	function argument 2
R10	a3	function argument 3
R11	a4	function argument 4
R12	t0	temporary register 0
R13	t1	temporary register 1
R14	t2	temporary register 2
R15	flag	flag register

Figure 3.1: ABI names for SAYAC Register

Chapter 4

Instruction Lowering

In the LLVM backend, a `lowering` pass is a transformation pass that converts high-level language constructs or operations into lower-level instructions that can be directly executed by the target hardware. The instruction Lowering informations are described in `SAYACISelLowering.cpp`. The instructions are lowered so that each operation in the flow graph represents a single instruction in the target machine. We have lowered two types of instruction as follows:

```
// It expands the instruction brcc to brcond so that instruction
// selection phase can match the pattern.
setOperationAction(ISD::BR_CC, MVT::i16, Expand);

// It will lower the GlobalAddress instruction by calling
// LowerGlobalAddress function which we have defined.
setOperationAction(ISD::GlobalAddress, MVT::i16, Custom);
```

Apart from instructions, the function calls, arguments and return values needs to be lowered which are implemented through following functions:

4.1 LowerCall

A `lower call` pass specifically targets function calls and converts them into machine instructions. When a function is called in high-level code, the compiler generates a `call` instruction, which typically includes the name of the function and any arguments passed to it. The `lower call` pass replaces this `call` instruction with one or more machine-level instructions that actually perform the function call.

4.2 Lower Formal Arguments

The process of lowering formal parameters involves several steps:

1. **Mapping the formal parameters to machine-level registers:** The LLVM backend uses a register allocation pass to assign virtual registers to the formal parameters of the function. These virtual registers are then mapped to physical registers during code generation.

2. **Inserting code to save and restore registers that are modified by the function:** If a function modifies any registers that are used outside of the function, the lowering pass inserts code to save the register value before the function executes and restore it after the function completes.

3. **Variable-length argument lists (varargs):** *Not implemented*

4. **Handling stack-based parameter passing:** When the number of arguments is too large to fit in the available registers, stack is required to pass parameters. *Not Implemented*

4.3 LowerReturn

`LowerReturn` is responsible for generating code to lower the return instruction in the target machine's assembly language.

The `LowerReturn` function uses this information to generate the appropriate machine-level instructions that implement the return instruction. This can include instructions to store the return value in a register or memory location, as well as instructions to restore the previous stack frame and return control to the calling function.

4.4 LowerGlobalAddress

GlobalAddress are a type of `SelectionDAG` node pointing to addresses within the code. This function lowers them by finding the target address location which the given instruction is pointing to and then replacing it with `LOAD_SYM` node, which will convert to `msym` instruction in further passes.

Chapter 5

Instruction Selection

The `SAYACInstrInfo.td` file is used to specify various properties and characteristics of instructions, which can be used by LLVM during code generation, optimizations, and analysis. It typically includes the following information:

1. **Instruction Description:** The file defines a `def` directive for each instruction, which specifies the mnemonic and other descriptive information for the instruction. This includes the opcode, instruction name, and any additional details that are relevant to the instruction.

2. **Instruction Properties:** Various properties of the instruction can be specified, such as its size in bytes, whether it is a terminator instruction, whether it modifies control flow, and other characteristics that are specific to the instruction.

3. **Operand Information:** The operands of the instruction, including their types, register classes, and any constraints or restrictions on the operands. This information is used during instruction selection, register allocation, and other code generation stages. The definition for different immediate operands is also provided in this file.

4. **Custom SDNode:** The SelectionDAG contains SelectionDAG Nodes, called as `SDNode`. In this file we can define custom `SDNode` as per our requirement and replace nodes in initial selection dag with our nodes.

This file uses various `classes` and `defs` defined in *llvm/include/llvm/Target/Target.td* and *llvm/include/llvm/Target/TargetSelectionDAG.td* files. Going through those files is recommended.

5.1 Arithmetic and Logic Instructions

The following code is used to describe the addition operation in SAYAC:

```
def ADDrr  : F_LAR<0b1001,
                   (outs GPR:$rd), (ins GPR:$rs1, GPR:$rs2),
                   "adr $rd $rs1 $rs2",
                   [(set GPR:$rd, (add GPR:$rs1, GPR:$rs2))]>;
```

This instruction is of type `F_LAR` class (defined in `SAYACInstrFormats.td`) The various arguments passed here are:

- `Opcode:` The initials `0b` means binary format.
- `OutOperandList:` It is defined using the outs keyword, followed by the name and type of the output operand. Here, `GPR` (defined in `RegisterInfo.td`) represents the output operand, and `$rd` is the name of the variable that will hold the output value.
- `InOperandList:` It is defined using the ins keyword, followed by the name and type of each input operand. The multiple input operands can be defined by separating them with commas.
- `AsmString:` It is a string representing corresponding SAYAC assembly code. The `$` symbol is followed by the operand name to refer to the operands defined in the `OutOperandList` and `InOperandList` fields. Here, `$rd`, `$sr1`, and `$rs2` are placeholders representing the corresponding output and input operands defined in the `OutOperandList` and `InOperandList` fields.
- `Pattern:` It accepts a list of dag patterns with which the instruction will be matched. If LLVM finds the given pattern in the `SelectionDAG` it will replace it with the defined instruction. In the given pattern add is an LLVM `SelectionDAG` node type and set is a special Tablegen `DAG` node type that will set the destination register $rd to summation of register `$rs1` and `$rs2` and will replace `add` with `adr` instruction in the `SelectionDAG` .

5.2 Load/Store Instruction

The following code is used to describe the load store operation in SAYAC .

```
let mayStore = 1 in
def STR : F_Store<0b00, (outs), (ins GPR: $rd, GPR: $rs1),
                   "str $rd $rs1",
                   []
                   >;

let mayLoad = 1 in
```

```
def LDR: F_Load<0b00, (outs GPR:$rd), (ins GPR:$rs1),
                  "ldr $rd $rs1",
                  []
```

`F_Load` and `F_Store` are instruction formats defined in `SAYACInstrFormats.td`.

The selection pattern for load and store instructions are defined separately, since it can match several types of instructions from `SelectionDAG`. Selection Pattern for store is defined as follows:

```
multiclass StPat<PatFrag StoreOp, InstSAYAC Inst> {
  def : Pat<(StoreOp GPR:$rs1, GPR:$rs2), (Inst GPR:$rs2, GPR:$rs1)>;
  def : Pat<(StoreOp GPR:$rs1, addr:$rs2), (Inst GPR:$rs2, addr:$rs1)>;
}

defm : StPat<truncstorei16, STR>;
defm : StPat<truncstorei8, STR>;
defm : StPat<store, STR>;
```

`multiclass` is like a function that takes whatever is specified within <> as arguments and substitutes them in all its definitions. Here the following substitution takes place:

```
defm : StPat<truncstorei16, STR>;

// This will be converted to :

def : Pat<(truncstorei16 GPR:$rs1, GPR:$rs2),  (STR GPR:$rs2, GPR:$rs1)>;
def : Pat<(truncstorei16 GPR:$rs1, addr:$rs2), (STR GPR:$rs2, addr:$rs1)>;
```

`Pat` class is defined in `TargetSelectionDAG.td`. It takes a dag pattern as first argument and matches it to the result dag pattern given in the second argument.

`truncstorei16`, `truncstorei8`, store are variants of store instruction in `SelectionDAG` and each of these instructions will be matched to `STR` instruction in our case.

`addr` is an Operand defined in the same file, it is used as an address operand (pointer):

```
def addr : ComplexPattern<iPTR, 1, "SelectAddr", [frameindex], []>;
```

The`ComplexPattern` class is defined in `TargetSelectionDAG.td`.

5.3 Compare Instructions

The following code is used to describe the compare instructions in SAYAC .

```
let isCompare = 1 in {
  def CMR : F_CMR<0b0, (outs), (ins GPR:$rs,  GPR:$rd),
                    "cmr $rs $rd", []>;
  def CMI : F_CMI<0b1, (outs), (ins GPR:$rd, imm16zx5: $imm),
                    "cmi $imm $rd", []> ;
}
```

`F_CMR` and `F_CMI` are instruction formats defined in `SAYACInstrFormats.td`

`imm16zx5` is defined as an immediate operand in the same file, it represents that the immediate part of `CMI` instructions can only store 5 bits. It is defined as follows:

```
defm imm16zx5 : Immediate<i16, [{
  return (N->getZExtValue() & ~0x000000000000001fULL) == 0;
}], NOOP_SDNodeXForm, "U5Imm">;
```

The compare instructions of SelectionDAG are mapped to Pseudo instructions, which are then converted to actual `SAYAC::CMR` instructions in `expandPostRAPseudo` function. For example to match the `seteq` instruction (compare instruction in which two registers are compared for equality and result stored in destination), the following pseudo instruction is created

```
// Define a pseudo instruction SEQ.
let isPseudo = 1 in
def SEQ : InstSAYAC<(outs GPR:$rd), (ins GPR: $rs1, GPR: $rs2),
                      "SEQ $rd $rs1 $rs2", [] >;

// Match seteq in SelectionDAG to pseudo instruction SEQ.
def : Pat<(seteq GPR:$rs1, GPR:$rs2), (SEQ $rs1, $rs2)>;
```

This mapping is required because one compare instruction requires multiple SAYAC instructions to be emitted which cannot be handled through Tablegen and requires writing custom C++ code.

5.4 Branch Instruction

The following code is used to describe the Branch instructions in SAYAC.

```
let hasSideEffects = 0, mayLoad = 0, mayStore = 0,
isBranch = 1, isTerminator = 1 in {
  def BRCeq : F_BRANCH<0b0, 0b000, (outs), (ins GPROpnd:$rd),
                    "brc 0x0 $rd", []>;
```

```
  def BRCne : F_BRANCH<0b0, 0b101, (outs), (ins GPROpnd:$rd),
                       "brc 0x5 $rd", []>;
  def BRCgt : F_BRANCH<0b0, 0b010, (outs), (ins GPROpnd:$rd),
                       "brc 0x2 $rd", []>;
  def BRCge : F_BRANCH<0b0, 0b011, (outs), (ins GPROpnd:$rd),
                       "brc 0x3 $rd", []>;
  def BRClt : F_BRANCH<0b0, 0b001, (outs), (ins GPROpnd:$rd),
                       "brc 0x1 $rd", []>;
  def BRCle : F_BRANCH<0b0, 0b100, (outs), (ins GPROpnd:$rd),
                       "brc 0x4 $rd", []>;
}
```

`F_Branch` are instruction formats defined in `SAYACInstrFormats.td`.

The second operand in `F_Branch` is flag interpretation bit (`fib`) defined in the instructions themselves.

Since branch instructions also require multiple SAYAC instructions to be generated, it is even handled through Pseudo instructions, the pattern matching code for pseudo instruction is defined as follows:

```
// Defines a pseudo instruction to branch to a target if compare equal is true.
let isPseudo = 1 in
def CBeq : InstSAYAC<(outs GPR:$rd), (ins GPR: $rs1, GPR: $rs2, brtarget8:$imm8),
                          "cmpbeq $rs1 $rs2 $imm8", [] >;


class BccPat<PatFrag CondOp, InstSAYAC Inst>
    : Pat<(brcond (XLenVT (CondOp GPR:$rs1, GPR:$rs2)), bb:$imm8),
          (Inst GPR:$rs1, GPR:$rs2, brtarget8:$imm8)>;

def : BccPat<seteq, CBeq>;

// The above two lines of code generate the following pattern matching code:

def : Pat<(brcond (XLenVT (seteq GPR:$rs1, GPR:$rs2)), bb:$imm8),
          (CBeq GPR:$rs1, GPR:$rs2, brtarget8:$imm8)>;
```

Here in the pattern matching code, `XLenVT` is a data type of the result of `seteq` (compare) instruction, which is defined as `i16` in `SAYACRegisterInfo.td` . `brcond` is branch conditional Selection DAG node, which goes to given location (referred through `bb:$imm8`) if the `seteq` (compare) operator returns true. The operand `brtarget8` defined in the code refers to target label that could be stored in 8-bit.

The definition of `brtarget8` is as follows:

```
class PCRelOperand<ValueType vt, AsmOperandClass asmop> : Operand<vt> {
  let PrintMethod = "printPCRelOperand";
  let ParserMatchClass = asmop;
  let OperandType = "OPERAND_PCREL";
}

def PCRel10 : PCRelAsmOperand<"10">;

def brtarget8 : PCRelOperand<OtherVT, PCRel10> {
  let EncoderMethod = "getPC10Encoding";
  let DecoderMethod = "decodePC10BranchOperand";
}
```

The `brcond` node is present as `brcc` in LLVM SelectionDAG, the expansion of `brcc` instruction is defined in `SAYACISelLowering.cpp` as follows:

```
// Expands brcc to brcond...which can match the pattern defined.
setOperationAction(ISD::BR_CC, MVT::i16, Expand);
```

5.5 Jump Instructions

The following code is used to describe the Jump instructions in SAYAC.

```
// Jump and doesn't save PC
let isTerminator = 1 in {
def JMR : F_CH<0b0010100, (outs), (ins GPR: $rd), "jmr $rd", []>;
}

// Jump and save PC
let isTerminator = 1, isCall = 1, Defs = [R1] in {
  def JMRS : F_CH<0b0010101, (outs GPR: $rd), (ins GPR: $rs), "jmrs $rd $rs", []>;
}

let isPseudo = 1 in {
  def PJMRS : InstSAYAC<(outs), (ins GPR: $addr), "pjmrs $addr", []>;
}

// Function call is mapped to pseudo jump which resolves
// itself (in expandPostRAPseudo) to JMRS by saving PC to R1 (return address)
def : Pat<(call i16: $addr), (PJMRS i16: $addr)>;
```

NOTE: Defining at least one unconditional branch (or jump) instruction is required

for any target, otherwise it doesn't compile. Branch/Jump instructions should have `isTerminator` attribute as 1 for toolchain to identify it.

5.6 Custom SDNode

In the file `SAYACInstrInfo.td` various custom `SDNode`'s have been defined as follows:

```
// Selection DAG types.

// These are target-independent nodes, but have target-specific formats.
def SDT_Call         : SDTypeProfile<0, -1, [SDTCisPtrTy<0>]>;

def call             : SDNode<"SAYACISD::CALL", SDT_Call,
                              [SDNPHasChain, SDNPOptInGlue, SDNPOutGlue,
                              SDNPVariadic]>;

// We will use this node to load a symbol to register (msym instruction)
def load_sym : SDNode<"SAYACISD::LOAD_SYM", SDTIntUnaryOp>;
```

Chapter 6

Important Functionalities

This chapter describes the important functions distributed across various files and what work do they accomplish. It is important to understand their contribution to code generation.

6.1 SAYACIselDAGToDAG.cpp

This file is responsible for instruction selection phase and it converts the input dag with LLVM IR insrtuctions lowered to supported target to an output dag that contains only target specific instructions.

6.1.1 Select

This function is responsible for selection of target instructions. This takes an `SDNode` as argument and uses pattern matching to select the desired target instruction. The function uses `SelectCode` function generated from `SAYACInstrInfo.td` to perform pattern matching but we can describe custom selection of instruction in the `C++` file. In the given code we have modified the `FrameIndex` node to corresponding `SAYAC::FI` (SAYAC frameIndex) node so that it could be resolved later. Selection phase is important because if any instruction cannot be selected at this stage then further processing cannot take place.

6.2 SAYACFrameLowering.cpp

6.2.1 determineCalleeSaves

When a new function is called few registers need to be saved in the stack so that they can be restored even if the called function modifies them. The registers which needs to be saved are specified here. In our case since we support frame pointer we require to save it and the value of return address also needs to be saved, which can be seen in the code.

6.2.2 spillCalleeSavedRegisters

This function is responsible for saving the registers into stack, it saves those registers which are specified by the `determineCalleeSaves` function.

6.2.3 restoreCalleeSavedRegisters

This function is responsible for restoring the callee saved registers from stack into the corresponding registers.

Note that the order of storing and restoring is important. If saved registers are stored in a particular order to the stack, then while restoring the stack values, the register in which those values would be put must be reverse.

6.2.4 determineFrameLayout

This function takes a machine function pointer as argument and calculates the amount of space this function will need in the stack. The stack space is allocated in alignment of 8 bits.

6.2.5 emitPrologue

This function generates prologue code for each function in the code. It gets the required stack size for the function from `determineFrameLayout` function, and allocates that much space in the stack by adjusting the stack pointer. It is also responsible to generate instructions to adjust frame pointer to point to top of the current machine function frame. The instruction is generated after `spillCaleeSavedRegisters` instructions so that `framepointer` is not modified before it is saved.

6.2.6 emitEpilogue

This function generates epilogue code for each function in the code. It is responsible for deallocating the stack space and then adjusting the stack pointer to its old position. It clears up the temporary variables used in the functions.

6.3 SAYACRegisterInfo.cpp

The file `SAYACRegisterInfo.cpp` defines `eliminateFrameIndex` function which will be responsible for removing the `frameindex` from `selectionDAG` and converting them into corresponding offset. Another important aspect of the file is the definition of call-preserved and reserved registers. Call preserved registers are registers that must be preserved across function calls, whereas reserved registers are those registers that must not be used by llvm.

These registers are defined using the `getCallPreservedMask` and `getReservedRegs` functions. `SAYACRegisterInfo` class inherits from `SAYACGenRegisterInfo`, which is generated from `SAYACRegiterInfo.td` file using TableGen. This takes the return address that target would use as argument.

6.3.1 eliminateFrameIndex

This function is present in `SAYACRegisterInfo.cpp`. As the function name states the main aim of this function is to eliminate frame indexes to actual frame offset in the code. For example, `frameindex<1>` points to the first space allocated in the frame, now if there are 2 saved registers then the offset of `frameindex<1>` would be 4 bytes (2 byte per integer - offset 0 stores first integer, offset 2 stores second integer). This function will calculate the offset and replace loading from `frameindex<i>` or storing into `frameindex<i>` to loading or storing from actual address. If offset is `i` then address would be `[fp+i]` , where `fp` is frame pointer.

6.4 SAYACInstrInfo.cpp

This file defines the stack layout, alignment and is responsible for providing utility function for code generation. We have `copyPhysReg` function to generate code for copying one register value to another, `movImm` is responsible for moving a 16 bit immediate value to the given destination register. `expandPostRAPseudo` is responsible for expanding Pseudo instructions and is described in more detail below.

6.4.1 expandPostRAPseudo

During the code generation process, LLVM transforms the high-level intermediate representation (IR) into a lower-level representation that is closer to the target machine architecture. This transformation involves several stages, including register allocation, instruction selection, and scheduling.

After the register allocation phase, some pseudo instructions may still remain in the generated code. These pseudo instructions often represent high-level language constructs or target-independent abstractions that need to be expanded into sequences of target-specific instructions.

The `expandPostRAPseudo` is called to perform this expansion. It analyzes the pseudo instructions and replaces them with the appropriate target-specific instructions that achieve the desired behavior. This expansion process takes into account the target-specific characteristics, such as available instructions, addressing modes, calling conventions, and other architectural features.

Consider the `EQUAL` comparison instruction of SAYAC. The result of comparison is stored in flag bits of R15 register.

For the instruction,

```
cmr rs1 rd
```

If `(rs1 > rd)`, then the `G flag, R15[5]`, will be set to 1 and if `(rs1 == rd) E flag, R15[4]` will be set to 1. Otherwise, it is considered less than.

But we need to store a `boolean` result of this instruction in some register to handle C instructions such as

```
int x = ( a == b );
```

So, we need to extract the `R15[4]` and store it in destination register. For this first we move constant 16 into `destReg` and then perform `bitwise AND` of `R15` with `destReg` with result being stored in `destReg`. Next we can perform `logical right shift` on `destReg` by `4 bits` to get the boolean value. The following code snippet from `expandPostRAPseudo` function performs this expansion of Pseudo Instruction `SEQ`. The definition for `SEQ` is present in `SAYACInstrInfo.td` file

```
case SAYAC::SEQ:
  {
    DebugLoc DL = MI.getDebugLoc();
    MachineBasicBlock &MBB = *MI.getParent();

    const unsigned destReg = MI.getOperand(0).getReg();
    const unsigned src1 = MI.getOperand(1).getReg();
    const unsigned src2 = MI.getOperand(2).getReg();

    BuildMI(MBB, MI, DL, get(SAYAC::CMR)).addReg(src1).addReg(src2);

    BuildMI(MBB, MI, DL, get(SAYAC::MSI), destReg).addImm(16);
    BuildMI(MBB, MI, DL, get(SAYAC::ANDrr), destReg)
        .addReg(SAYAC::R15)
        .addReg(destReg);
    BuildMI(MBB, MI, DL, get(SAYAC::SHIl), destReg).addReg(destReg).addImm(4);

    MBB.erase(MI);

    return true;
  }
```

Here, `MI` refers `MachineInstruction` which is passed as parameter to `expandPostRAPseudo` function. We can get operands associated with pseudo instruction using `getOperand` function of `MI`. It expects the index of operand as the parameter. The indexing starts with 0

from `outOperandList` and then followed by `inOperandList` .

Next, `BuildMI()` is used to create new machine instructions that correspond to the desired behavior of the pseudo instruction being expanded. The following parameters are passed to this function

1. **MBB** `(MachineBasicBlock&)`: This parameter represents the machine basic block where the new instruction will be inserted. `MachineBasicBlock` is a container for machine instructions within a function's machine code representation. It holds a sequence of instructions that are executed sequentially.

2. **MI** `(MachineBasicBlock::iterator)`: This parameter is an iterator pointing to the insertion point within the `MBB`. The new instruction created using `BuildMI()` will be inserted before the instruction pointed to by `MI`. This allows precise control over the insertion location within the basic block.

3. **DL** `(DebugLoc)`: This parameter represents the debug location associated with the new instruction. Debug information can be useful for debugging and error reporting purposes.

4. **opcode** `(unsigned)`: This parameter specifies the opcode of the new machine instruction.

Operands specific to the target instruction are added to `MIB` using the appropriate `addReg()`, `addImm()`, or other operand-specific functions. Finally, the original pseudo instruction is erased using `MBB.erase()` to remove it from the basic block.

Chapter 7

Results

This chapter describes few simple code examples and their explanations which can be compiled to SAYAC target assembly code. See Appendix-B for more code and their conversion.

7.1 Detailed example: Adding two numbers

C Code:

```
int main() {
        int a = 32, b = 54 ;
        int sum = a + b;
        return 0;
}
```

The given code when compiled with clang produces an LLVM intermediate representation code in `.ll`, which looks as follows:

```
target datalayout = "e-m:e-p:16:16-i32:16-a:0:16-n16-S16"
target triple = "sayac"

; Function Attrs: noinline nounwind optnone
define dso_local i16 @main() #0 {
entry:
  %retval = alloca i16, align 2
  %a = alloca i16, align 2
  %b = alloca i16, align 2
  %sum = alloca i16, align 2
  store i16 0, i16* %retval, align 2
  store i16 32, i16* %a, align 2
  store i16 54, i16* %b, align 2
  %0 = load i16, i16* %a, align 2
```

```
  %1 = load i16, i16* %b, align 2
  %add = add nsw i16 %0, %1
  store i16 %add, i16* %sum, align 2
  ret i16 0
}
```

The given code is passed through various phases and finally shared object file (`.s`) is generated.

The dag representation after various phases are as follows:

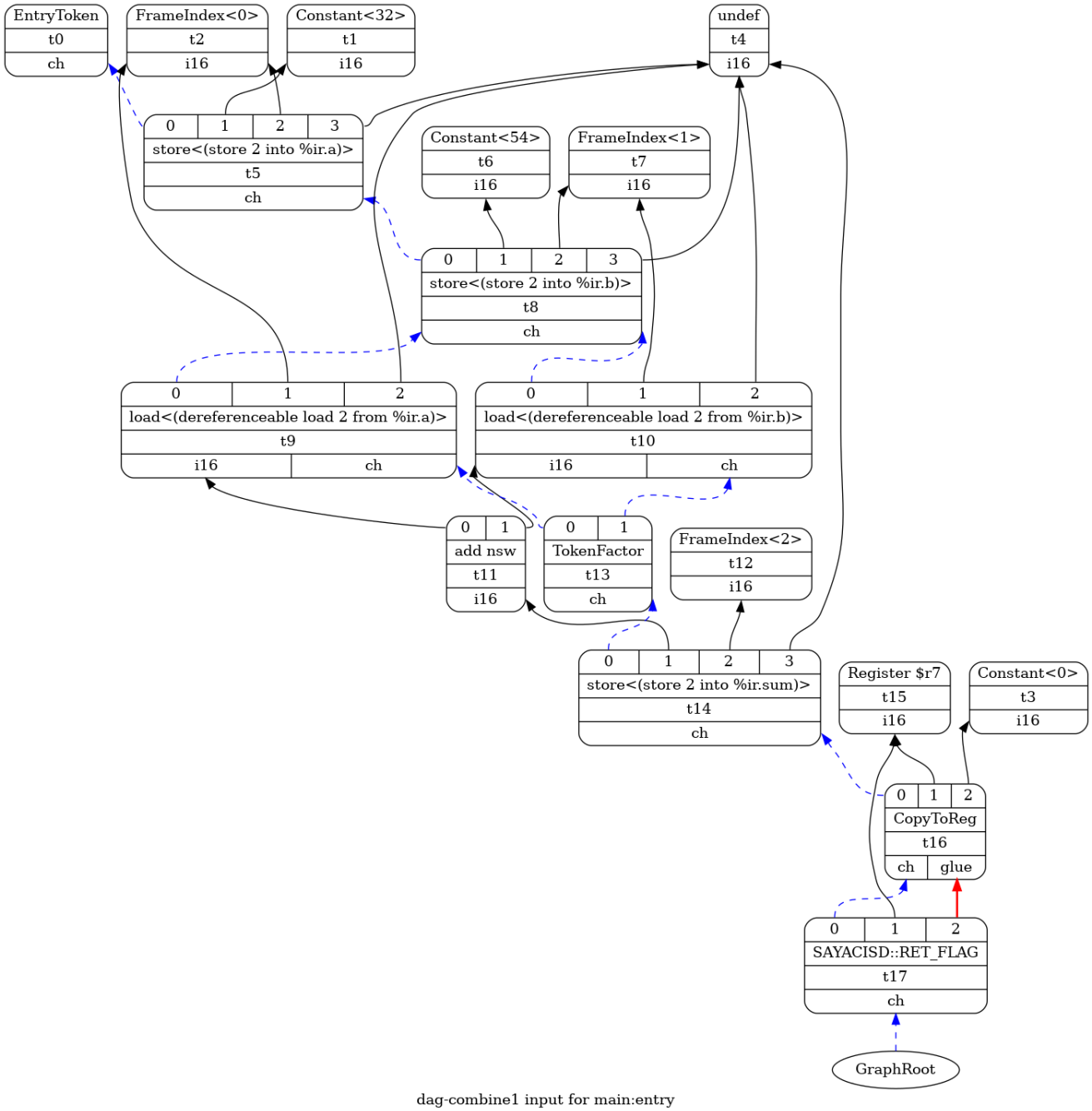

Figure 7.1: DAG After instruction lowering

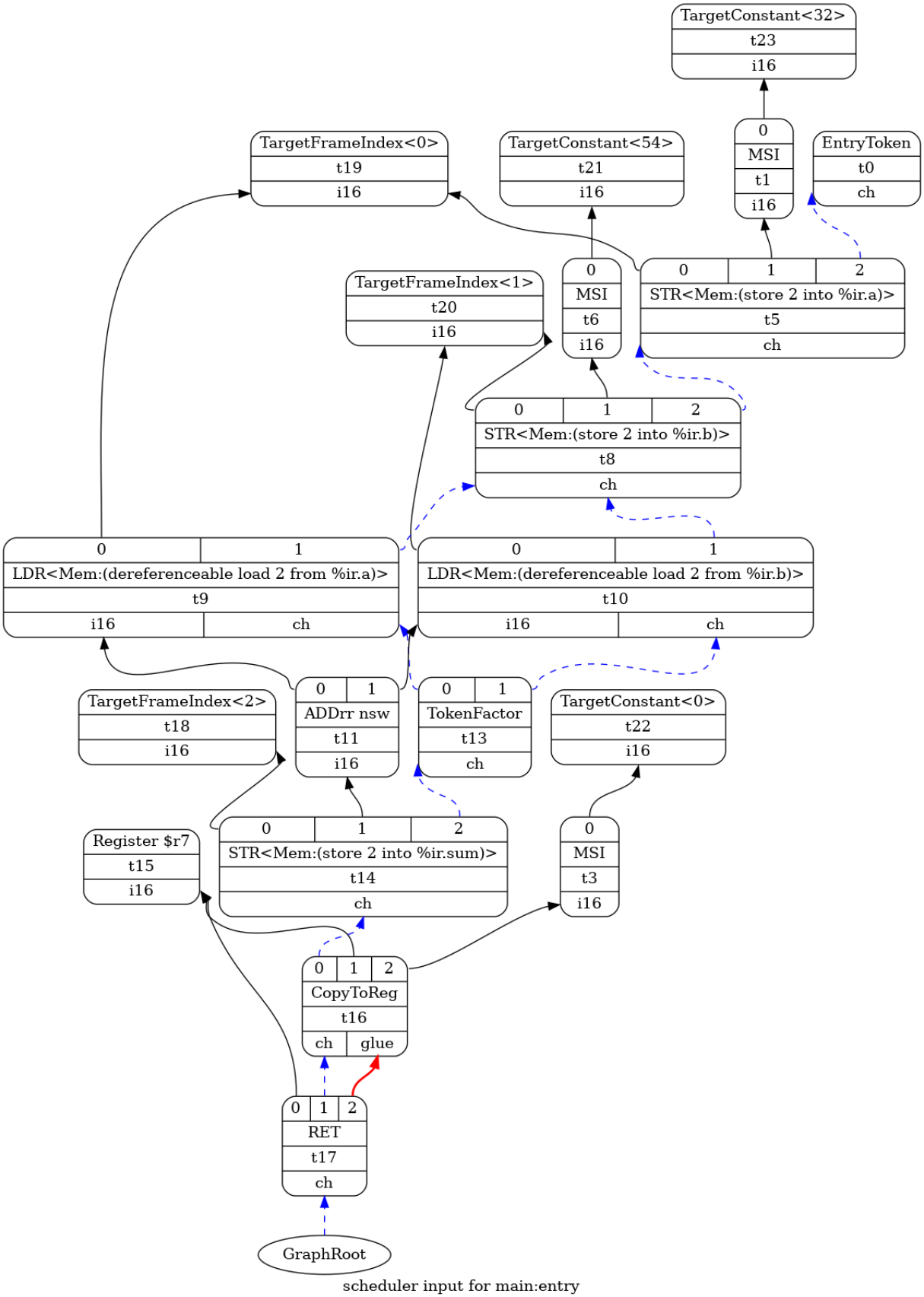

Figure 7.2: DAG After instruction Selection

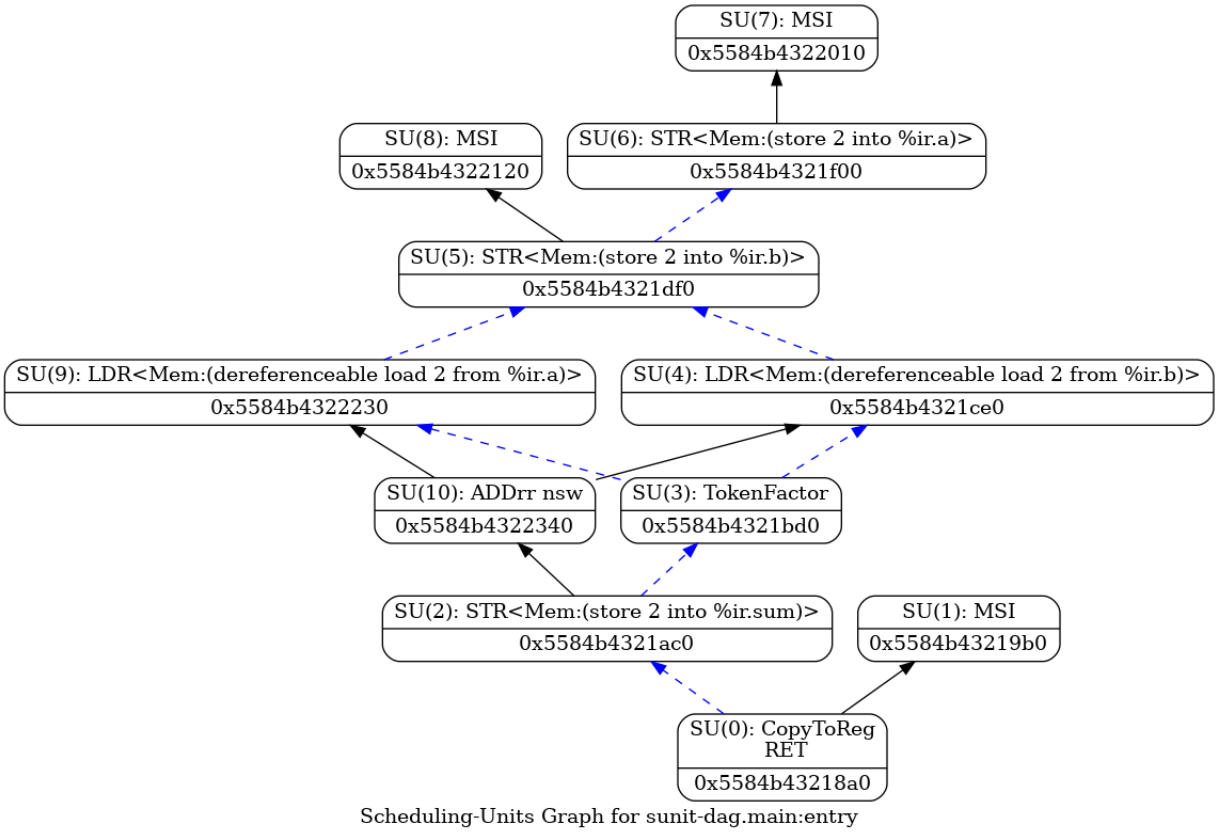

Figure 7.3: Instruction Scheduling DAG

After instruction scheduling Prologue/Epilogue Code is inserted and the finally generated assembly code is as follows (Function arguments are passed in a0 and a1 register and return value is stored in a0 as defined in Register ABI Section):

```
.text
        .file        "sum.c"
        .globl        main                                # -- Begin function main
        .align        3
        .type        main,@function
main:                                   # @main
# %bb.0:                                # %entry
        adi sp -16              # allocate 16 byte to function stack

        msi a0 14
        adr a0 sp a0
        str a0 ra               # Store return address in stack

        msi a0 12
        adr a0 sp a0
        str a0 fp               # Store frame pointer in stack

        msi a0 16
        adr fp sp a0            # Adjust frame pointer
```

```
                        # Prologue code ends here
        msi a0 0
        msi a1 6
        adr a1 fp a1
        str a1 a0               # Stores retval in stack

        msi a0 32
        msi a1 8
        adr a1 fp a1
        str a1 a0               # Store a (=32) in stack

        msi a0 54
        msi a1 10
        adr a1 fp a1
        str a1 a0               # Store b (=54) in stack

        msi a0 8
        adr a0 fp a0
        ldr a0 a0               # load a to register a0

        msi a1 10
        adr a1 fp a1
        ldr a1 a1               # load b to register a1

        adr a0 a0 a1            # Add a, b and store result in a1

        msi a1 12
        adr a1 fp a1
        str a1 a0               # Store sum to stack

        msi a0 0                # Store 0 in a0 (return value)
                        # Epilogue code begins
        msi a1 12
        adr a1 sp a1
        ldr fp a1           # Restore frame pointer

        msi a1 14
        adr a1 sp a1
        ldr ra a1           # Restore return address

        adi sp 16           # Deallocate stack space
        cmi 0 zero          # Sets FIB for eq to zero
        brc 0x0 ra          # Branch to return address (ra)
.Lfunc_end0:
        .size        main, .Lfunc_end0-main
                                        # -- End function
        .ident       "clang version 12.0.1 (https://github.com/llvm/llvm-project/
8724eb480dea541e9bddc86757e240b70852fb65)"
```

```
        .section        ".note.GNU-stack","",@progbits
```

7.2 Example: Factorial

C Code :

```
    int main() {
        int n = 6, factorial_n = 1;
        for(int i = 2; i <=n ; ++i) {
            factorial_n *= i;
        }
    }
```

SAYAC Assembly code output:

```
        .text
        .file        "factorial.c"
        .globl        main                            # -- Begin function main
        .align        3
        .type        main,@function
main:                                   # @main
# %bb.0:                                # %entry
        adi sp -16              # allocate 16 byte to function stack

        msi a0 14
        adr a0 sp a0
        str a0 ra               # Store return address in stack

        msi a0 12
        adr a0 sp a0
        str a0 fp               # Store frame pointer in stack

        msi a0 16
        adr fp sp a0          # Adjust frame pointer
                        # Prologue Code ends here
        msi a0 0
        msi a1 -6
        adr a1 fp a1
        str a1 a0               # Store retval (0) in stack

        msi a0 6
        msi a1 -8
        adr a1 fp a1
```

```
        str a1 a0               # Store n = 6 in stack

        msi a0 1
        msi a1 -10
        adr a1 fp a1
        str a1 a0               # Store factorial_n = 1 in stack

        msi a0 2
        msi a1 -12
        adr a1 fp a1
        str a1 a0               # Store i = 2 in stack

        msym a0 .LBB0_1
        jmr a0                  # Goto %for.cond

.LBB0_1:                                # %for.cond
                                        # =>This Inner Loop Header: Depth=1
        msi a0 -12
        adr a0 fp a0
        ldr a0 a0                # Load i in a0

        msi a1 -8
        adr a1 fp a1
        ldr a1 a1                # Load n in a1

        cmr a0 a1                # Compare i and n
        msym a0 .LBB0_4
        brc 0x2 a0          # Goto %for.end if i>n (0x2 FIB represents greater than)
        msym a0 .LBB0_2
        jmr a0              # Goto %for.body

.LBB0_2:                                # %for.body
                                        #   in Loop: Header=BB0_1 Depth=1
        msi a0 -12
        adr a0 fp a0
        ldr a0 a0                # Load i in a0

        msi a1 -10
        adr a1 fp a1
        ldr a1 a1                # Load factorial_n in a1

        mul a0 a1 a0            # Multiple i and factorial_n

        msi a1 -10
        adr a1 fp a1
        str a1 a0                # Store factorial_n at its position

        msym a0 .LBB0_3
```

```
        jmr a0                  # Goto %for.inc

.LBB0_3:                                # %for.inc
                                        #   in Loop: Header=BB0_1 Depth=1
        msi a0 -12
        adr a0 fp a0
        ldr a0 a0               # Load i in a0

        adi a0 1               # Add 1 to a0

        msi a1 -12
        adr a1 fp a1
        str a1 a0               # Store i at its position

        msym a0 .LBB0_1
        jmr a0                 # Goto %for.cond

.LBB0_4:                                # %for.end
        msi a0 -6
        adr a0 fp a0
        ldr a0 a0              # Load retval to a0 (this will be returned from function)
                        # Epilogue code begins here
        msi a1 12
        adr a1 sp a1
        ldr fp a1              # Restore frame pointer

        msi a1 14
        adr a1 sp a1
        ldr ra a1              # Restore return address

        adi sp 16              # Deallocate stack space
        cmi 0 zero             # Sets FIB for eq to zero
        brc 0x0 ra             # Branch to return address (ra)
.Lfunc_end0:
        .size        main, .Lfunc_end0-main
                                        # -- End function
        .ident        "clang version 12.0.1 (https://github.com/ak821/SAYAC-
 Compiler.git 76547d3e98c59a447f55dea8242812e7e96fef9e)"
        .section        ".note.GNU-stack","",@progbits
```

The next part describes handled and unhandled C constructs and improvements that can be done.

Discussions and Conclusions

Developing a compiler requires to handle various different constructs and standard libraries. Currently including standard libraries are not supported and given below are listed few handled and unhandled C constructs

Handled C Constructs

- Basic Datatype: integers, character.
- Derived Datatype: Array, Structure.
- Pointers.
- Conditional statements : if - else, ternary operator.
- Looping statements: for loop, while loop.
- Function calling (can pass upto 5 parameters).
- Arithmetic Operators: addition(+), subtraction(-), multiplication(*), division(/), increment(++), decrement(–).
- Comparison Operators: greater than (>), greater than or equal (>=), less than (<), less than or equal (<=), equal to (==), not equal to (!=)
- Logical Operators: logical and (&&), logical or (||), logical not (!)
- Bitwise operators: bitwise and (&), bitwise or (‖), bitwise exclusive-or (∧), left shift (<<), right shift (>>),

Unhandled C constructs

- Float data type.
- Function calling with more than 5 arguments
- Dynamic memory allocation (int n = 5, int arr[n])

- The assembly code has a data section, which behaves incorrectly currently. E.g: Arr[] = 1, 2, 3, 4, 5 stores the array value in data section and loads them incorrectly.
- Modulo operator: In target assembly code modulo operator requires two consecutive register, which needs to be handled by register tuple.
- Handle multiplication overflow.
- Using standard library.

Improvements and Future Scope

The first and most important part to improve is to support standard libraries and their codes. This will require few more ISA instructions to be handled like load/store from IO devices.

There are various unhandled constructs that needs to be added as described in above section, along with that there are minor issues in code (commented at places - like Xor requires addition which may cause overflow). If code size exceeds 256 bytes Msym operator will fail and we will require to add higher bits of symbol address as well.

The Project needs to implement other scheduling and code optimizations passes to generate better quality code.

Another important part of code generation is to continue development process without breaking anything, this would require to add test cases and test modules. LLVM has support for adding test cases and automated testing, which needs to be inculcated into the development process.

Bibliography

[1] Writing LLVM BackEnd: https://llvm.org/docs/WritingAnLLVMBackend.html

[2] Nacke, K. (2021). Learn LLVM 12

[3] Backend for LEG Architecture: https://github.com/frasercrmck/llvm-leg

[4] LLVM Developer Meeting: https://www.youtube.com/watch?v=AFaIP-dF-RA

[5] LLVM Backend Writing Tutorial and code patches:
https://sourcecodeartisan.com/2020/09/13/llvm-backend-0.html

[6] RISCV Specification: https://github.com/riscv-non-isa

[7] RISCV Backend Code:
https://github.com/llvm/llvm-project/tree/release/12.x/llvm/lib/Target/RISCV