SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Deriving Machine Code Generators from LLVM-IR

Tobias Kamm

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Deriving Machine Code Generators from LLVM-IR

Herleitung von Maschinencodegeneratoren aus der LLVM-IR

Author:	Tobias Kamm
Supervisor:	Prof. Dr. Thomas Neumann
Advisor:	Dr. Alexis Engelke
Submission Date:	23.08.2024

Acknowledgments

First, I'd like to thank my advisor, Dr. Alexis Engelke, for his continued support and guidance throughout this thesis and my studies. Furthermore, I'd especially like to thank Tobias Schwarz for providing advice and support concerning this thesis and TPDE, even while he was on his exchange semester. You have effectively been a second advisor to me.

This thesis benefited greatly from the picky proofreading of Andreas Dachsberger, Leonard Endriß, and Konrad Gößmann.

Apart from that, my studies were greatly influenced by people I met along the way, which is why I would like to mention them here as well: Thank you to Jerg Kappeler for being the best teammate in (Advanced) Binary Exploitation I could have asked for, and being a good bouldering partner[1]. Philipp Erhardt deserves a special mention for his continued support of my favourite programming project, MemeAssembly.

Moreover, a big thank you to everyone else that I got to share my time at TUM with until now, which includes, but is not limited to: Christian Zimmerer, Philipp Erhardt, Jerg Kappeler, Jim Teichgräber, Konrad Gößmann, Janez Rotman, Alexandra Graß, Viktor Boskovski, Benjamin Rickels, Dominic Prinz, Daniel Mayer, and Miriam Fehn.

Thank you to my family for supporting me throughout my studies.

Finally, thanks to ChatGPT for entertaining me all throughout the development process with hilariously wrong answers regarding LLVM[2].

Here's also a shout-out to Pau, my roommate. But as he actually does not deserve it, here it is in a tiny font size. You're welcome, now you're famous. Better post about that on LinkedIn :P

[1]"Einfach aufstehen."
[2]https://chat.openai.com/share/5437ecf9-8017-4b39-b989-0342d992b12d

Abstract

While LLVM has been an impactful compiler framework in the domain of AOT compilation, its JIT compilation support is currently lacking due to a slow `O0`-backend. TPDE, a novel code generation backend taking LLVM-IR as input, was created to make using LLVM-IR more suitable for JIT compilation environments, such as database systems.

However, TPDE's development process is aggravated through its use of hand-written machine code generators, which exist for each possible LLVM-IR instruction. Aside from being difficult to create and verify, such generator functions have to be rewritten for every new target architecture, increasing the development and maintenance effort when supporting AArch64 or RISC-V.

To address this issue, we present `tpde-asmgen`, a tool that runs during the build process of TPDE and automatically generates register-agnostic machine code generators. The behavioural specifications are expressed in LLVM-IR or a higher-level language that lowers to it. `tpde-asmgen` then uses the standard LLVM backend but stops before register allocation. From this Machine IR state, `tpde-asmgen` extracts the target instructions and parameter information to derive machine code generators emitted as C++ code.

Limited integration of `tpde-asmgen`'s machine code generators into TPDE affected the compile and run time of the intspeed SPEC benchmarks. An average code generation time improvement of 0.6% and an average runtime performance increase of 0.18% was measured. The run results of the utilised SPEC benchmarks were verified to confirm the correctness of `tpde-asmgen`'s output.

Contents

1 Introduction

In environments where the source code remains unknown until its execution, such as when running database queries, the usage of just-in time (JIT) compilation can lead to substantial performance improvements over interpretation [1]. A JIT compiler executes the compiled source code directly after it was compiled. Hence, compile-time is a crucial area of improvement, as it is part of the program's overall runtime.

A widely used framework for ahead-of-time (AOT) compilation is LLVM, which not only features code generation backends for a wide range of platforms but also provides many optimization passes and a code generation back-end that is able to produce high-quality code. While it supports JIT compilation [2], low compile times have not been the development focus, leading to relatively slow compile times even for unoptimised (`O0`[1]) code generation [3].

TPDE [4] attempts to balance the aforementioned up- and downsides of using LLVM for JIT compilation. Fundamentally, it implements its own code generation backend while taking LLVM-IR as input, making it possible to utilise LLVM's data structures while still providing a considerable compile-time improvement. As of March 2024, it achieves a performance improvement by a factor of 10–20x over LLVM's `O0` backend, with the generated code having a comparable runtime performance.

However, building a code generation backend from scratch is not trivial. For code generation, every possible operation needs to be mapped to a series of calls to the encoder, generating the required machine code instructions, and register allocator. These machine code generators also include smaller optimisations to speed up the execution time in trivial cases, such as performing a shift instead of dividing or multiplying by a power of two. However, this approach currently has some problems:

1. Writing such machine code generators by hand is time-consuming.
2. Finding mistakes, may it be during code review or debugging, is complex.
3. These machine code generators are not portable. When supporting a new architecture, such as ARM or RISC-V, all code generators must be rewritten, exacerbating the aforementioned problems.

In this thesis, we present `tpde-asmgen`: a tool written in C++ that generates register-agnostic machine code generators to be used in TPDE. As input, `tpde-asmgen` receives code in a higher-level language compiled to LLVM-IR, describing the behaviour the resulting machine code generator should encode. For instance, if the resulting machine code generator should encode an integer addition, the input function to `tpde-asmgen` is a function that takes two integers as parameters and returns the sum of them, as seen later in Listing 2.5. `tpde-asmgen` then uses LLVM's code generation backend to select appropriate machine instructions but avoids register allocation.

By traversing the selected machine instructions, `tpde-asmgen` generates C++ code that integrates with TPDE to encode the instructions and allocate the required registers when the machine

[1]pronounced as "o zero"

2 Background

This chapter will cover the basics required to understand our chosen approach. This includes an introduction to LLVM and its Machine IR.

2.1 Static Single Assignment (SSA)

Static Single Assignment (SSA) is a property often used in compiler-internal intermediate representations, requiring all variables to be assigned only once [5].

```c
int y = x + 1;
x = x + 1;
int z = x + 1;
```

Listing 2.1: *A simple C-code snippet with implicit dataflow.*

```c
const int y = x + 1;
const int x1 = x + 1;
const int z = x1 + 1;
```

Listing 2.2: *The code of Listing 2.1 in SSA form.*

From a compiler's perspective, consider the code snippet in Listing 2.1. A typical code transformation involves merging redundant computations, known as Common Subexpression Elimination [6]. While this is not the case here, as `x` holds a different value in both computations due to a modification, it still requires some context to conclude this, as `y` and `z` are both computed using the same expression, being `x + 1`.

This example shows a typical problem with traditional source code representations: data flow (meaning information about which values are used where) is implicit. To mitigate this, the code of Listing 2.1 is transformed so that all variables are only assigned once. Consequently, modifications of variables must be assigned to a new variable with a different name. While compilers typically perform this transformation in a compiler-internal Intermediate Representation (IR), we resort to modifying the source code for now. This modification can be seen in Listing 2.2. Following this modification, it is evident that Common Subexpression Elimination cannot be applied here since `y` and `z` are now computed using different expressions.

Listing 2.2 is now in Static Single Assignment (SSA) form. This property allows one to establish a chain from a variable's definition to its uses, forming a directed acyclic graph, known as the Data Flow Graph [7]. This sets the basis for a myriad of transformations[1] [5, 7].

However, in its current form, SSA's support for control flow structures is limited. This is because the value of a variable might depend on the origin, for instance, when it is updated in one branch while not in another. This is especially true for loops, which almost always require at least one variable to be assigned more than once, often being a loop counter. To highlight this problem, we take a `for`-loop as an example, as seen in Listing 2.3, this time using pseudocode.

[1]Compiler developers tend to use the term transformations instead of optimisations since one cannot know for sure that they actually improve the code in the specific case [6].

```
entry:
   i = 0
   goto header
header:
   if i < 10:
      goto loop
   else:
      goto end
   endif
loop:
   //...
   i = i + 1
   goto header
end:
   [...]
```

Listing 2.3: *A simple for-loop in pseudocode.*

```
entry:
   goto header
header:
   i = ϕ(entry: 0; loop: new_i)
   if i < 10:
      goto loop
   else:
      goto end
   endif
loop:
   //...
   new_i = i + 1
   goto header
end:
   [...]
```

Listing 2.4: *The code of Listing 2.3 with a ϕ-node.*

One can see that such a loop is currently not easily transferrable into SSA. This is because the loop counter `i` is assigned twice. First, it is zero-initialised, after which it is updated with an increased value.

One approach to solve this problem involves using so-called ϕ-nodes (PHI-nodes), which set the value of a variable depending on the branch origin. All ϕ-nodes are executed concurrently while entering a code block. In the case of Listing 2.3, `i` would be zero if we entered the loop for the first time (meaning that we entered `header` from `entry`), and the incremented `i` if we came from the loop body. Listing 2.4 shows said modifications. As we assign `i` only once, this Listing is now in SSA form.

As it forms the foundation for numerous analyses and transformations, SSA is a vital part of compilers and thus is used in many. Examples include Clang, which uses LLVM-IR [8], and GCC [9].

2.2 LLVM

The LLVM project, one of the most widely known compiler frameworks, describes itself as a "collection of modular and reusable compiler and toolchain technologies" [10]. Among others, the project includes:

- **The LLVM Core libraries** – providing a language- and target-independent optimiser and code generators for many CPUs
- **Clang** – a compiler for C, C++ and Objective-C that uses LLVM as its backend
- **LLDB** – a debugger
- **libc++** – LLVM's own implementation of the C++ Standard Library

However, only the LLVM Core libraries are of interest in the scope of this thesis. Thus, the term LLVM will refer to the LLVM Core libraries throughout this thesis. All details regarding LLVM in this thesis refer to the state at version 18.1.6, specifically, commit 1118c2e[2].

[2]https://github.com/llvm/llvm-project/commit/1118c2e05e67a36ed8ca250524525cdb66a55256

Due to its language-independent nature, LLVM is not only used by Clang. Since designing an IR is a non-trivial task, and LLVM already features a mature code generation backend with lots of transformations and code generation targets implemented, some compiler developers prefer to lower the source code to LLVM-IR and use the LLVM compiler framework instead of their own. Programming languages using LLVM include Julia [11], Rust [12], Haskell [13], and Swift [14]. Third-party compiler implementations using LLVM also exist for languages such as Python [15] and Go [16].

To explain LLVM's backend further, we will use the C-function in Listing 2.5 as an example: the function `add` takes two integers, adds them together, and returns the result.

```
int add(int a, int b) {
    return a + b;
}
```

Listing 2.5: *A simple C function that adds two integers.*

```
define i32 @add(i32 %0, i32 %1) {
  %3 = add i32 %1, %0
  ret i32 %3
}
```

Listing 2.6: *Listing 2.5 in LLVM-IR, attributes were omitted.*

2.2.1 LLVM-IR

When running LLVM's C-compiler (Clang) on the code in Listing 2.5, the input function is lowered to a language-independent IR. This representation is known as LLVM-IR, LLVM's implementation of an SSA-based IR. It captures ideas from higher-level languages, as well as key Instruction Set Architecture (ISA) operations: Apart from basic instructions (such as `add` or `div`), LLVM-IR also supports, among others: vectors, structs, a C-like `switch` statement and advanced pointer arithmetic using its `getelementptr`-Instruction [17].

In LLVM-IR, as in most intermediate representations, a function is divided into multiple basic blocks being straight-line code sequences without branches that must end with a Terminator Instruction. Terminator Instructions include a jump to another basic block and a return statement [17].

Listing 2.6 shows the code of Listing 2.5 lowered to LLVM-IR. Note that it only has one basic block (the entry block) since we do not have any control flow. In LLVM-IR, local identifier names always start with a percent-symbol and may be followed by a number (unnamed value) or name (named value) [17].

2.2.2 LLVM's Backend

In the context of compiler design, a backend is responsible for target-specific code generation. While a compiler backend typically consists of three steps (instruction selection, instruction scheduling, and register allocation) [18], LLVM's backend cannot be separated into these three phases. Instead, the backend consists of multiple *passes*, which are routines that take an IR as input and return a modified version that will be used in the next pass. The number and order of run passes depend on the optimisation level and target architecture. Notable examples of passes include the following:

- `x86-isel` – performs initial instruction selection.
- `dead-mi-elimination` – removes IR instructions that are unused.
- `livevars` – analyses the variable usage and annotates the IR values accordingly.

- `phi-node-elimination` – destroys ϕ-nodes, resulting in an IR that is no longer in SSA form.
- `twoaddressinstruction` – operations in the form of `A = B op C` are rewritten to `A = B; A op= C`, to make use of target instructions overwriting their source operand.
- `greedy` – performs initial register allocation.
- `AsmPrinter` – converts the finalised IR to textual assembly or machine code.

This architecture allows for improved modularity, as different pipelines only need to specify which passes are run. This also makes it possible to stop the backend before or after a certain pass to extract the IR at that point.

2.2.3 Machine IR

While having a target-independent IR for general code transformations is beneficial, the code generation backend of a compiler is very architecture-specific, requiring the annotation of specific hardware instructions and registers. For this, LLVM's Machine IR (MIR) was created. Listing 2.7 shows the code from Listing 2.5 in Machine IR after some instruction selection passes but before register allocation. Specifically, we stopped before the elimination of ϕ-nodes.

```
# Machine code for function add: IsSSA, TracksLiveness
Function Live Ins: $edi in %0, $esi in %1

bb.0 (%ir-block.2):
  liveins: $edi, $esi
  %1:gr32 = COPY killed $esi
  %0:gr32 = COPY killed $edi
  %2:gr32 = ADD32rr killed %1:gr32(tied-def 0), killed %0:gr32, implicit-def dead $eflags
  $eax = COPY killed %2:gr32
  RET 0, killed $eax
```

Listing 2.7: *The generated Machine IR of our add-function, obtained by running LLVM with `--stop-before=phi-node-elimination`, some attributes were omitted.*

To gain a better understanding of LLVM's Machine IR, we will discuss Listing 2.7 in more detail by only focusing on a limited number of lines at once:

```
# Machine code for function add: IsSSA, TracksLiveness
```

This lists all properties of the following MIR function. `IsSSA` means that the IR still (partly) operates on SSA. As we will see later, the last use of variables (meaning when they are not *live* anymore) is currently tracked, hence the `TracksLiveness` property.

```
bb.0 (%ir-block.2):
  liveins: $edi, $esi
```

Here, we define our only basic block of this function. `liveins` are values that are already defined when entering this basic block and have future uses. In this case, the machine registers `edi` and `esi` are considered live, as they contain our two parameters [19]. Consequently, values still live when leaving our basic block are known as `liveouts`.

```
%1:gr32 = COPY killed $esi
%0:gr32 = COPY killed $edi
```

As our IR still operates on SSA, the values that live in hardware Registers are first copied into SSA-values. This is done using the `COPY`-instruction. In Machine IR, IR-values are treated as virtual registers. After the variable name, the register class is specified, here being `gr32`: a general-purpose register with a size of 32 bits. `killed` refers to this being the last use of the variable within this control flow branch. Variables used as parameters to instructions are referred to as *operands*.

Generally speaking, all possible machine instructions in MIR are in one of three categories:

- **meta-instructions** – These instructions are not related to any hardware instruction but exist to provide additional context and metadata. An example would be `DBG_VALUE`, representing debug information. In the context of `tpde-asmgen`, they can be ignored.
- **pseudo-instructions** – These instructions might be translated to actual hardware instructions later but do not directly represent one. Examples would be `COPY`, also used in this function, and `PHI`, representing a ϕ-node.
- All other instructions directly map to a hardware instruction of the selected target architecture.

```
%2:gr32 = nsw ADD32rr killed %1:gr32(tied-def 0), killed %0:gr32, implicit-def dead
    $eflags
```

Here, we have an example of a machine instruction that directly maps to an x86-64 instruction: `ADD32rr`, which is an `add`-Instruction that adds two 32-bit registers, hence the `32rr`-suffix.

In general, the operands of machine instructions can be categorised into four types [20]:

- **explicit-def** – These operands are defined by this instruction, which is done explicitly. They are always left of the equals sign. In this case, the IR-value `%2` is defined by the `MachineInstruction` and holds the result of the addition.
- **implicit-def** – This instruction also defines these operands. However, they are not explicitly stated in the machine code and cannot be changed. In this example, our addition results in modified flags, such as the Overflow-Flag [21]. Thus, `eflags` (the flags-register) is marked as `implicit-def`.
- **explicit** – These operands are inputs to our `MachineInstruction`, which are not fixed and must be provided. As `ADD32rr` takes two arbitrary registers, they must be provided as parameters, which makes `%0` and `%1` explicit. Explicit operands can be recognised by having no implicit/explicit-related attribute.
- **implicit** – Unlike explicit parameters, the instruction defines these as fixed parameters that cannot be changed. This is often due to ISA-constraints. An example would be x86-64's variable shift instruction, which uses the `cl`-register to determine the number of shifts needed [21]. Consequently, such a `MachineInstr` has `cl` as an implicit parameter.

This instruction includes a few more attributes, which are discussed below [20]:

- **killed** – This attribute signals that this is the last use of this operand.

- **dead** – Signals that this operand is defined but never used.
- **tied-def** – `ADD32rr` is a so-called instruction with destructive source. x86-64's `add`-instruction takes two registers yet overwrites the first register with the result of the addition [21]. The `tied-def` (tied definition) attribute exists to reflect this relationship and signals that this instruction will overwrite this operand. The number following the attribute, being `0` in this case, refers to the operand index that is the related register. In this example, it is the first one, which is `%2`.

```
  $eax = COPY killed %2:gr32
  RET 0, killed $eax
```

After performing the addition, the result needs to be returned. Following the System V Calling Convention [19], that value is returned in the register `eax`. Hence, the value is first copied into that register. After that, the basic block is terminated by a `RET`-instruction, which returns `eax`.

It is important to note that we will not operate on the string representation of LLVM's Machine IR, but on its data structure, `llvm::MachineFunction`. Its design and functions will not be discussed further as they are irrelevant to our approach. For more details, refer to LLVM's documentation [22]. More information regarding Machine IR can be found in the respective reference manual [20].

If we let the code generation backend perform the register allocation, our Machine IR is no longer in SSA form and uses hardware registers throughout the IR. This can be seen in Listing 2.8. The aforementioned Listing also highlights the fact that there are multiple passes concerning instruction selection, as this machine function now uses `lea` instead of `add`.

```
# Machine code for function add: NoPHIs, TracksLiveness, NoVRegs, TiedOpsRewritten
Function Live Ins: $edi, $esi

bb.0 (%ir-block.2):
  liveins: $edi, $esi
  renamable $esi = KILL $esi, implicit-def $rsi
  renamable $edi = KILL $edi, implicit-def $rdi
  renamable $eax = LEA64_32r killed renamable $rdi, 1, killed renamable $rsi, 0, $noreg
  RET64 $eax
```

Listing 2.8: *Machine IR of our add-function after register allocation, obtained by running LLVM with `--stop-after=unpack-mi-bundles`.*

As register allocation is irrelevant to our approach, the term "Machine IR" will refer to Machine IR after instruction selection and before register allocation, as seen in Listing 2.7, except where otherwise noted.

2.3 TPDE

TPDE [4] is a compiler backend generating machine code from LLVM-IR. As of March 2024, it achieves a performance improvement by a factor of 10–20x over LLVM's `O0` backend, with the generated code having a comparable runtime performance.

As this thesis focuses on our chosen approach and not the implementation details, code details will be kept to a minimum. However, as `tpde-asmgen`'s resulting code will often interact with TPDE's register allocator, we will briefly overview it.

2.3.1 TPDE's Register Allocator

TPDE is a single pass backend, implying that register allocation is done while the instructions are emitted with no changes done at a later point. TPDE's register allocator features two types of registers:

- `Reg` – a wrapper for an LLVM-IR value, meaning it can only be used for computation results. It is either a constant value, stored in a register, or stored in memory. It employs reference counting to track at which point a value is dead and how often it is currently used. If, at any point, one party wants to modify the value while the reference count is not 1, it must use another register. If required, registers held by a `Reg` can be *spilled* to the stack to temporarily free up a register.
- `ScratchRegister` – a wrapper for a temporary register, meaning that it will only be used until the end of a computation. It is important to note that, in contrast to `Reg`, registers held by a `ScratchRegister` cannot be spilled to the stack. This makes them useful for blocking a register from being used in the future by allocating the specific register. However, it implies that allocating too many `ScratchRegisters` might block all registers, leading to an error.

2.3.2 Machine Code Generator Example

In Listing 2.9, we provide an example of a machine code generator that calculates the absolute value of an integer. As input, it receives the LLVM-IR instruction (`inst`) that it has to encode. It then gets the register currently holding the input value, creates a destination register, and encodes the necessary instructions.

```
//Get the Reg that holds the input parameter
Reg reg = argumentReg(valIdx(inst->getOperand(0)), 0);
//Allocate result register
Reg result = resultRegEager(valIdx(inst), 0);
Asm::Reg resReg = result.getMachineReg();

ScratchRegister scratch(*this);
//Force the input parameter to be in a register.
//If it is stored in memory, it will be loaded into a register
Asm::Reg valReg = getAsRegister(reg, scratch);

//Encode the instructions. Syntax is ASMx(instName, parameters...)
ASM2(mov, resReg, valReg);
ASM1(neg, resReg);
ASM2(cmovs, resReg, valReg);

//Move the result value into the register where the result should be stored
setValue(result, resReg);
```

Listing 2.9: *A machine code generator for the abs intrinsic function. Comments were added for explanation, assertions were removed for brevity, auto-types were expaned for clarity.*

Listing 2.10 shows our machine code generator after integrating `tpde-asmgen`. While we still query the current `Reg` holding the input parameter, instruction encoding and register handling are now done by `tpde-asmgen`, making the code easier to read and verify.

```
Reg reg = argumentReg(valIdx(inst->getOperand(0)), 0);
//Create a Reg for the result, but don't set a register yet (*lazy*)
Reg result = resultRegLazy(valIdx(inst), 0);
ScratchRegister scratch(*this);

//Encode using tpde-asmgen, scratch now holds the register containing the result
encode_abs(this, AsmOperand(std::move(reg)), scratch);

//Set the result register
setValue(result, std::move(scratch));
```

Listing 2.10: *The abs machine code generator after using* `tpde-asmgen` *for encoding the instructions and allocating the registers.*

3 Machine Code Generator Derivation from MIR

This chapter describes the code generation approach we chose in `tpde-asmgen` to support the x86-64 architecture. This chapter will explain the approach based on an example specification, which can be seen in Listing 3.1. The corresponding Machine IR that will be used by `tpde-asmgen` to derive the machine code generator is shown in Listing 3.2.

```c
unsigned rem(unsigned a, unsigned b) {
    return a % b;
}
```

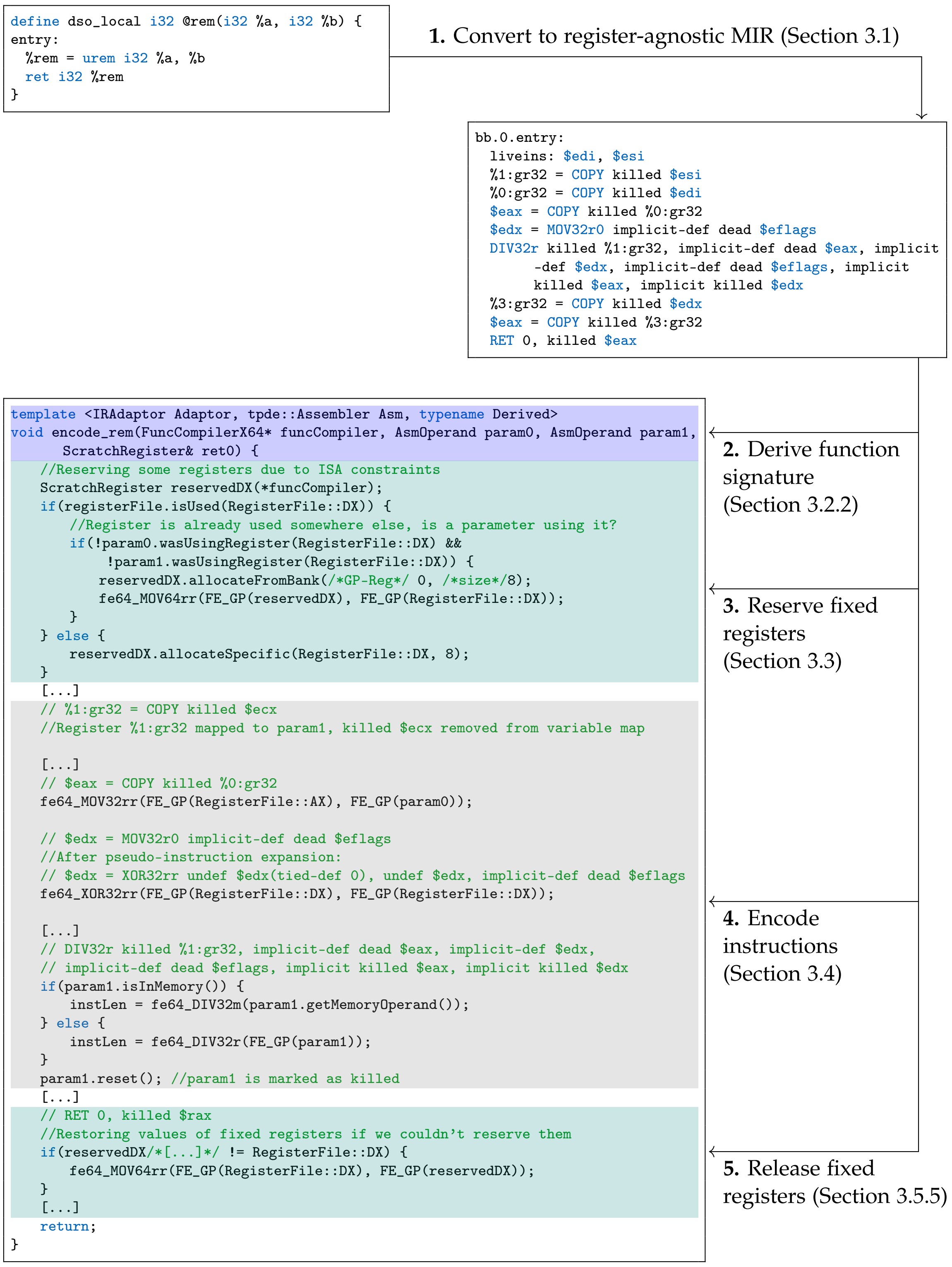
Listing 3.1: *A simple C-function performing a remainder operation.*

```
bb.0.entry:
  liveins: $edi, $esi
  %1:gr32 = COPY killed $esi
  %0:gr32 = COPY killed $edi
  $eax = COPY killed %0:gr32
  $edx = MOV32r0 implicit-def dead $eflags
  DIV32r killed %1:gr32, implicit-def dead $eax, implicit-def $edx, implicit-def dead
      $eflags, implicit killed $eax, implicit killed $edx
  %3:gr32 = COPY killed $edx
  $eax = COPY killed %3:gr32
  RET 0, killed $eax
```

Listing 3.2: *remainder-function in Machine IR.*

We chose to use this function as an example, as it highlights many cases that we cover in this chapter. It features interactions with specific physical registers, employs pseudo-instructions such as `COPY` or `MOV32r0`, and features an instruction for which we change the encoding based on the input type.

An overview of our approach is shown in Figure 3.1. The resulting code is for demonstration purposes only, as many types were simplified, and some implementation details of the encoder and register allocator were omitted. Nonetheless, it gives an overview of what we generate using our approach: A function encoding the necessary instructions to perform the requested calculation using provided input values.


1. Convert to register-agnostic MIR (Section 3.1)
2. Derive function signature (Section 3.2.2)
3. Reserve fixed registers (Section 3.3)
4. Encode instructions (Section 3.4)
5. Release fixed registers (Section 3.5.5)

```
define dso_local i32 @rem(i32 %a, i32 %b) {
entry:
  %rem = urem i32 %a, %b
  ret i32 %rem
}
```

```
bb.0.entry:
  liveins: $edi, $esi
  %1:gr32 = COPY killed $esi
  %0:gr32 = COPY killed $edi
  $eax = COPY killed %0:gr32
  $edx = MOV32r0 implicit-def dead $eflags
  DIV32r killed %1:gr32, implicit-def dead $eax, implicit
        -def $edx, implicit-def dead $eflags, implicit
        killed $eax, implicit killed $edx
  %3:gr32 = COPY killed $edx
  $eax = COPY killed %3:gr32
  RET 0, killed $eax
```

```
template <IRAdaptor Adaptor, tpde::Assembler Asm, typename Derived>
void encode_rem(FuncCompilerX64* funcCompiler, AsmOperand param0, AsmOperand param1,
      ScratchRegister& ret0) {
    //Reserving some registers due to ISA constraints
    ScratchRegister reservedDX(*funcCompiler);
    if(registerFile.isUsed(RegisterFile::DX)) {
        //Register is already used somewhere else, is a parameter using it?
        if(!param0.wasUsingRegister(RegisterFile::DX) &&
             !param1.wasUsingRegister(RegisterFile::DX)) {
            reservedDX.allocateFromBank(/*GP-Reg*/ 0, /*size*/8);
            fe64_MOV64rr(FE_GP(reservedDX), FE_GP(RegisterFile::DX));
        }
    } else {
        reservedDX.allocateSpecific(RegisterFile::DX, 8);
    }
    [...]
    // %1:gr32 = COPY killed $ecx
    //Register %1:gr32 mapped to param1, killed $ecx removed from variable map

    [...]
    // $eax = COPY killed %0:gr32
    fe64_MOV32rr(FE_GP(RegisterFile::AX), FE_GP(param0));

    // $edx = MOV32r0 implicit-def dead $eflags
    //After pseudo-instruction expansion:
    // $edx = XOR32rr undef $edx(tied-def 0), undef $edx, implicit-def dead $eflags
    fe64_XOR32rr(FE_GP(RegisterFile::DX), FE_GP(RegisterFile::DX));

    [...]
    // DIV32r killed %1:gr32, implicit-def dead $eax, implicit-def $edx,
    // implicit-def dead $eflags, implicit killed $eax, implicit killed $edx
    if(param1.isInMemory()) {
        instLen = fe64_DIV32m(param1.getMemoryOperand());
    } else {
        instLen = fe64_DIV32r(FE_GP(param1));
    }
    param1.reset(); //param1 is marked as killed
    [...]
    // RET 0, killed $rax
    //Restoring values of fixed registers if we couldn't reserve them
    if(reservedDX/*[...]*/ != RegisterFile::DX) {
        fe64_MOV64rr(FE_GP(RegisterFile::DX), FE_GP(reservedDX));
    }
    [...]
    return;
}
```

Figure 3.1: *An overview of* `tpde-asmgen`*'s process. The resulting code is significantly shortened and simplified*

3.1 Input Preparation

Before running `tpde-asmgen`, the behavioural specifications are externally compiled to LLVM-IR, which can be done using `Clang`. As input, `tpde-asmgen` then takes an LLVM-IR module encoded as a file, referred to as a bitcode file. As the first step, LLVM is configured using the `--stop-before` flag to stop the code generation pipeline before phi node elimination. We chose this stage in the code generation pipeline for two reasons: Firstly, it is after the liveness analysis of variables (`livevars`), which makes it easier to spot whether values are no longer used or not used at all. Secondly, our approach currently does not support ϕ-nodes. Hence, we can easily detect if the input function is supported by searching for `PHI` instructions and aborting if one is found. The exact reasons why ϕ-nodes are unsupported and how ϕ-handling could be implemented will be discussed in more detail in Section 3.7.1. Furthermore, destroying ϕ-nodes implies that the resulting Machine IR is no longer in SSA form.

As a next step, the specific code generation target has to be defined for LLVM. We define the target architecture as x86-64 and set the optimisation level to `1`. The reason why we do not choose a higher level (`2` or `3`) is to avoid optimising transformations for divisions that add ϕ-nodes, introduced by the `codegenprepare` pass. All other configuration values (such as the code model) are configured to their defaults.

After creating and running a backend pipeline, the MIR functions can be queried individually. For each `MachineFunction`, we employ the approach described in the following sections.

3.2 Initial Analysis

In this Section, we collect all required information to derive the function signature and check if we need to reserve any registers. Listing 3.6 highlights all MIR components considered in this Section.

```
bb.0.entry:
  liveins: $edi, $esi
  %1:gr32 = COPY killed $esi
  %0:gr32 = COPY killed $edi
  $eax = COPY killed %0:gr32
  $edx = MOV32r0 implicit-def dead $eflags
  DIV32r killed %1:gr32, implicit-def dead $eax, implicit-def $edx, implicit-def dead
      $eflags, implicit killed $eax, implicit killed $edx
  %3:gr32 = COPY killed $edx
  $eax = COPY killed %3:gr32
  RET 0, killed $eax
```

Listing 3.6: *An overview of the relevant instructions for the initial analysis.*

3.2.1 Detecting Fixed Registers

Fixed registers are registers that are required for instructions. This might include input registers (such as the variable shift using `cl`) and output registers (such as x86-64's `div` instruction, which saves the result in `rdx` and `rax` [21]).

To find all fixed registers, we iterate over all machine instructions. Specifically, we consider the implicit definitions and uses of each instruction. Some registers need to be ignored, which include `eflags` (the flags register) and `mxcsr` (the floating point configuration register). If a register is found that is not on a blacklist, then it is added to a list of fixed registers, which will be used later.

3.2.2 Deriving the Function Signature

To derive the function signature, we extract the function name, the number of input registers, and the number of output registers.

Function Name

The function name is a modification of the name used by the input function. To avoid target-specific function naming schemes, the function name is taken from the LLVM-IR function [23]. The function name is prefixed with `encode_`. For example, `add` becomes `encode_add`.

Input Registers

The number of input registers is the number of live registers when entering the function's entry block. As we want `tpde-asmgen`'s resulting code to allow for multiple operand types, we created a class that wraps multiple operand types, named `AsmOperand`. The class supports the following types:

- **Register** – a register which can always be read but may not be writable.
- **Memory Operand** – a value stored at a specific address. This means that using the value would result in a load operation.
- **Address** – references the address where something is stored, not the stored value itself. In contrast to memory operands, materialising the value would result in the actual address being computed.
- **Immediate** – a constant integer value.

That way, one function can handle all operand combinations by querying the contained type at runtime instead of creating multiple functions that take different parameter types. This class also provides helper functions to query information about the contained variable type and change it.

Output Registers

We determine the number of output registers by finding the `RET` instruction and counting the number of used operands. The parameter type of the output registers is a `ScratchRegister` passed as a reference, meaning that modifying the reference will also change the referenced value in the caller's scope. This means that the return type of the encoder function is `void`, and the scratch registers are filled when the function returns. Using a return value is not an option, as multiple registers might be returned, for example, when performing 128 bit computations or returning structs.

```
void encode_rem(FuncCompilerX64* funcCompiler, AsmOperand param0, AsmOperand param1,
    ScratchRegister& ret0)
```

Listing 3.7: *The derived function signature for the `rem`-function, types were simplified for readability.*

3.3 Reserving Registers

In the previous Section, we described how we obtain a list of all fixed registers. We want to ensure we do not destroy the current value of a fixed register when using it within our machine code generator. Hence, we will reserve the register at the start of our generated function, preventing it from being used by another value. We do this by allocating a `ScratchRegister` to that register.

If the register is already used, we attempt to permanently relocate its value to a different register or, if that is not possible, temporarily back up the register's value to restore it at the end of our function.

3.4 Encoding Regular Instructions

```
bb.0.entry:
  liveins: $edi, $esi
  %1:gr32 = COPY killed $esi
  %0:gr32 = COPY killed $edi
  $eax = COPY killed %0:gr32
  $edx = MOV32r0 implicit-def dead $eflags
  DIV32r killed %1:gr32, implicit-def dead $eax, implicit-def $edx, implicit-def dead
       $eflags, implicit killed $eax, implicit killed $edx
  %3:gr32 = COPY killed $edx
  $eax = COPY killed %3:gr32
  RET 0, killed $eax
```

Listing 3.8: *An overview of the instructions that support our encoding approach described in the following paragraphs*

This Section explains how we encode regular instructions, meaning instructions that directly map to hardware instructions. Listing 3.8 highlights the instruction we will handle with the approach below.

3.4.1 Definitions

Many instructions explicitly define new values. However, not every definition in MIR results in a new register in our machine code generator. This can be due to an instruction overwriting an input operand instead of using a separate register as its destination.

How we handle a definition depends on whether it is marked as `tied-def`. If it is not, then this is a new register, which we must allocate first.

If the definition is tied to an input operand, we must check whether the tied input register is marked as `killed`. This is necessary, as `tied-def` input operands are not assumed to be overwritten in MIR; a second assignment to a value would break SSA. An example of this can be seen in Listing 3.9, where `%0` is used in two instructions with a `tied-def` definition but only marked as killed in the second usage. Consequently, we can only reuse the tied input register if it is marked as killed. If the value is still used afterwards, we must move the input value to a newly allocated register to avoid clobbering the original value.

If we can reuse the source register according to LLVM, we emit a runtime check to see if the register can be overwritten. If the register has a reference count greater than one, we must not destroy the value and need to copy the value to a newly allocated `ScratchRegister`, which will be used for the calculation.

```
bb.0.entry:
  liveins: $edi
  %0:gr32 = COPY killed $edi
  %1:gr32 = IMUL32rr %0:gr32(tied-def 0), %0:gr32, implicit-def dead $eflags
  %2:gr32 = IMUL32rr killed %1:gr32(tied-def 0), killed %0:gr32, implicit-def dead
      $eflags
  $eax = COPY killed %2:gr32
  RET 0, killed $eax
```

Listing 3.9: *Machine IR for a function that takes one integer and multiplies it with itself twice, showing that tied-def operands can exist without a killed-attribute.*

Another attribute that should be considered is `early clobber`, signalling that an operand is written to before all input operands are read. This differs from `tied-def` in that `tied-def` output operands will be overwritten after all inputs are read. It is necessary to label this explicitly, as it will lead to unexpected behaviour when said output operand is also used as an input operand. In that case, writing to the output operand would also modify the input operand.

`early clobber` is currently unsupported, mainly used when compiling inline Assembly. While it is used in `AArch64` instructions, x86-64 only uses `early clobber` in some AVX512 instructions. These instructions are currently unsupported.

3.4.2 Encoding all Operand Combinations

To improve the code quality of `tpde-asmgen`'s machine code generators, we also account for variants of instructions that use immediate values or a value stored in memory, instead of register operands. To support this, we need to be able to derive the variant information, which is architecture-specific. Section 3.6 explains this for the x86-64 architecture. An example of what we will generate can be seen in Listing 3.10. As `IDIV`, a signed division, can also take a memory operand as input, we explicitly check at runtime whether the provided parameter is stored in memory and change the encoding based on that.

```
if(param1.isInMemory()) {
    //Encode IDIV32m
} else {
    //If immediate value, load value of param1 into register
    //Encode IDIV32r
}
```

Listing 3.10: *Pseudo-code that checks for all possible encodings of IDIV based on the input type.*

When emitting the calls to the encoder for an instruction, the different Machine IR operand types of an instruction are handled as follows:

Virtual Register

Our instruction variant info is considered when encountering a virtual register operand in MIR. We first check the type we received as input (a register, a memory operand, or an immediate). We can use the value as-is if a variant uses the provided type. If not, we emit code that loads the value into a register.

Immediate

If the MIR operand is an immediate value (for example, when adding a constant to a register), then we encode the operand as-is.

Physical register

While it is possible to encode variants here, it would require significantly more effort to do so. Details on why this is the case and how this could be implemented are shared in Section 3.7.6. For now, the register is encoded as-is, without any variants.

Memory Operand

In MIR, a memory operand is represented as multiple operands. While the exact structure of a memory operand depends on the target architecture, it usually consists of the following types:

- **Base register** – takes the value from a register as the address' base.
- **Index register** – this allows to split the address into two registers. The value of this register can sometimes be further manipulated using multiplications or shifts, allowing it to be used for index-based addressing of arrays.
- **Displacement** – a constant value added to the address.

If we encounter a memory operand as a parameter, we first extract the values for the individual components. However, the actual type used as the base or index register might not be a register, which requires us to perform some transformations and allows for optimisations.

```c
void store64(int64_t* addr, int64_t val) { *addr = val; }
```

Listing 3.11: *A simple C function that stores a 64 bit value at an address.*

Take the specification function in Listing 3.11 as an example. The resulting MIR function will use a target-specific store operation using a memory operand with a base register, unused index register, and unused displacement. LLVM assumes that the base register is a register, which might not be true. As `addr` is a parameter to our specification function, it will become an input parameter of type `AsmOperand` to our machine code generator. Thus, it might also be an immediate, an address (with its own base/index/displacement) or a value stored in memory.

This means we must adapt our memory operand at the machine code generator's runtime. The following transformations hold for both the base and index registers:

- **Register** – no transformation is required; use the register as-is.
- **Immediate** – attempt to merge the constant value with the displacement.
- **Value stored in memory** – load the value into a register.
- **Address** – compute the address and load it into a register. If the other register of LLVM's memory operand is unused, we attempt to replace the entire memory operand with this address. For example, suppose the variable used as the base register is an address, and the index register is unused. In that case, we use the variable's base register as the base register and the variable's index register as the index register, merging the variable's displacement with the memory operand's displacement.

In the context of x86-64, a memory operand in Machine IR consists of the following values:

- **Base register** – a register containing an address. Common examples are `rbp` for frame pointer relative addressing and `rip` for position-independent code.
- **Index register** – a register containing a value that will be added to the base register's value.
- **Scale** – a value multiplied by the index register's value. Must be one of $\{1, 2, 4, 8\}$.
- **Displacement** – a constant value added to the result.
- **Segment register** – This is only used for accesses into thread-local storage and is currently not supported.

This means we can employ the strategy explained before, with one limitation: displacements in x86-64 have a maximum size of 32 bits. Thus, we can only merge displacements if the resulting value fits into 32 bits.

3.4.3 Final Checks

After an instruction is successfully encoded with all possible variants, we check if any input operands are marked as `killed`. For all `killed` variables, we destroy the variable that holds its value. For example, if the value was stored in a `ScratchRegister`, then the register is no longer used and can be acquired by others.

If we supported control flow, destroying registers marked as `killed` would not be valid, as `killed` would then only refer to the current branch.

3.5 Handling Special Instructions

As pseudo instructions do not directly map to hardware instructions, they require special handling. Listing 3.12 gives an overview of some instructions we will cover in this Section.

```
bb.0.entry:
  liveins: $edi, $esi
  %1:gr32 = COPY killed $esi
  %0:gr32 = COPY killed $edi
  $eax = COPY killed %0:gr32
  $edx = MOV32r0 implicit-def dead $eflags
  DIV32r killed %1:gr32, implicit-def dead $eax, implicit-def $edx, implicit-def dead
      $eflags, implicit killed $eax, implicit killed $edx
  %3:gr32 = COPY killed $edx
  $eax = COPY killed %3:gr32
  RET 0, killed $eax
```

Listing 3.12: *An overview of the instructions that will be handled in this section.*

3.5.1 COPY

`COPY` instructions copy values between virtual or physical registers. Depending on whether the source and destination register are physical (for example: `rax`) or virtual (for example: `%5`), we must change our strategy:

Virtual to Virtual COPY

If the source is marked as `killed`, make the destination register use the source variable's destination. If the source is live, copy the current value to a newly allocated `ScratchRegister`.

Virtual to Physical COPY

This might occur for two reasons: Firstly, there might be a COPY into a return value. In this case, we do not copy the value into the specified physical register (for example: `eax`) but into the return variable provided to us.

The second option is performing a COPY into a reserved register, for example, into `cl` for a variable shift instruction. In that case, we emit a `mov`-instruction into the specified register.

Physical to Virtual COPY

This kind of COPY might be emitted when moving parameters into virtual registers at the beginning of our function. Here, we set the destination register to reference the physical register.

Physical to Physical COPY

While physical-to-physical `COPY` instructions are not assumed to be present in MIR at that point, they can still be emitted through a chain of copies. An example of this can be seen in Listing 3.13. The first `COPY` is a physical-to-virtual `COPY`, resulting in `%5` referencing `edx`. The next `COPY` becomes a physical-to-physical `COPY`, as the source variable references a physical register. Here, we generate a `mov`-instruction between the two physical registers.

```
%5:gr32 = COPY killed $edx
$ecx = COPY killed %5:gr32
```

Listing 3.13: *An example MIR snippet that would result in a physical-to-physical `COPY`.*

3.5.2 EXTRACT_SUBREG

As the name implies, `EXTRACT_SUBREG` extracts a specific subregister from a source register and stores it in the destination register. We treat it as a Virtual-to-virtual `COPY`: If the source register is `killed`, make the destination reference the source variable's destination. If not, copy the subregister value into a newly allocated `ScratchRegister`.

3.5.3 INSERT_SUBREG

This instruction inserts a subregister into another register. In most cases, the destination register is defined by an `IMPLICIT_DEF` instruction, being an instruction that indicates an unused register. In that case, this is a zero extension on the current register.

If a different instruction defines the destination register, we insert the subregister into the destination register using a fitting `mov`-instruction.

3.5.4 SUBREG_TO_REG

This instruction just hints that the register size changed but does not affect the register itself. Apart from making the destination register use the same variable as the source register, no actions are performed.

3.5.5 RET

When a return instruction is found, it is important to note that no `ret`-instruction should be generated. `RET` signals to us that this is the end of our machine code generator. As we moved all values to the corresponding return variables beforehand, this step can be omitted here.

What still needs to be taken care of is releasing fixed registers. As a reminder, if we required an in-use register to be free and the value could not be relocated to a different one, we temporarily saved its value in a different register.

In case the fixed register was unused or we were able to relocate the current value, no further action is required. When leaving our function, the `ScratchRegister`'s destructor automatically releases the register. If not, we must restore the original value to the register before leaving our encoder function.

3.5.6 Other Pseudo Instructions

As explained before, pseudo instructions might result in hardware instructions being emitted but do not directly represent one. In our `rem`-example (Listing 3.12), an example of such an instruction is, apart from `COPY` or `RET`, `MOV32r0`. It is an instruction that sets a specified register to zero.

We currently cannot handle this instruction, as it does not directly map to a hardware instruction we can encode. For this, we attempt to turn the pseudo instruction into a regular instruction using `expandPostRAPseudo`, a function found in `llvm::TargetInstrInfo` [24]. The result of this expansion in our current example is found in Listing 3.14. If the expansion is successful, we can continue with the standard approach described in Section 3.4.

```
$edx = XOR32rr undef $edx(tied-def 0), undef $edx, implicit-def dead $eflags
```

Listing 3.14: *The resulting machine instruction after expanding `MOV32r0`*

However, this approach only works for some pseudo instructions. Pseudo instructions that `expandPostRAPseudo` does not handle are currently not supported.

3.6 Mapping LLVM Instructions to Fadec Functions

This Section addresses the issue of deriving the necessary information from a machine instruction required to encode an instruction and its variants using Fadec. First, we introduce Fadec, the encoder that we use. Next, we examine the reasons why this task presents a challenge.

3.6.1 The Fadec Encoder

Fadec [25] is a library featuring a decoder for x86-64 and x86-32, as well as an encoder for x86-64. As `tpde-asmgen` only uses Fadec's encoder, we will disregard the decoder and refer to the encoder as Fadec, though its official name is "fadec-enc".

Instructions are encoded in Fadec using function calls. Every x86-64-instruction has its own function, with the operands to that instruction being passed as function arguments. Listing 3.15 shows an example of encoding an addition between `rax` and `rdx`.

```
unsigned instrLen = fe64_ADD64rr(buf, 0, FE_AX, FE_DX);
```

Listing 3.15: *A function call to Fadec that encodes "`add rax, rdx`". `buf` is a pointer to which the resulting bytes will be written.*

Another option could have been to use the encoder built into LLVM instead of an external library. We decided against this and to use Fadec simply because Fadec's encoder was developed with performance in mind. This matters greatly, as the encoding must be done at compile time.

However, choosing an encoder different from the one LLVM offers led to some challenges, which we will discuss next.

3.6.2 Challenges

There are multiple challenges while performing this derivation:

1. **LLVM and Fadec employ different instruction naming schemes.**
   While some instructions are named exactly the same (for example, `ADD32rr`), this is sadly not true for all instructions. While Fadec mostly follows a coherent naming scheme, this is not true for LLVM. For example, `MOVSDto64rr`, being a move of 64 bits from an xmm-register into a general purpose register, is named in Fadec as `SSE_MOVQ_X2XGr`. The x86-64 instruction to which this will be mapped is a `movq`-instruction, meaning that LLVM's naming does not always correlate with the instruction's mnemonic.

2. **Neither LLVM nor Fadec provides explicit information about instruction variants.**
   Each variant is a separate instruction/function, though with similar naming. However, a similar name does not guarantee that a function is a variant.

3. **Performing a manual map between LLVM and Fadec is not feasible.**
   At the time of writing, LLVM has 19627 different machine instructions. While most of them can be ignored, for instance, because they are meta instructions or would never be generated by LLVM (such as VMX instructions), manually performing the mapping would be an error-prone and time-consuming task. However, this is exactly the kind of task we wanted to eradicate from TPDE using `tpde-asmgen`.

In the rest of this Section, we give an overview on how we achieved an automatic mapping between LLVM's instructions and Fadec's functions, supporting a large subset of LLVM's instructions. Limitations to this approach will be analysed later in Section 3.7.5.

3.6.3 Storing Fadec-Specific Information

To map machine instructions to Fadec function calls, we first need to store Fadec-specific information in `tpde-asmgen`. We chose to store the following information, which was generated using a script written in Python.

Simplified Fadec Names

To simplify instruction name matching, Fadec's function names are modified. An example includes removing unused instruction prefixes.

Listing Possible Instruction Variants

Thanks to Fadec's consistent instruction naming scheme, it is easy to find possible variants of an instruction from a function name. A visualisation of this approach using `ADD32rr` can be found in Figure 3.2.

1. We iterate through the string until a lowercase character is found. They are used to denote the parameter types of this instruction.
2. If the parameter is a register, meaning that the character is an `r`, attempt to replace it with `i` and `m` and check if that function exists.
   The same cannot be done for immediate (`i`) and memory (`m`) parameters, as we cannot easily morph other parameter types into those types without altering the behaviour of the resulting code.
3. If the variant exists, add the newly found function to a list and start a recursive call that searches for variants with the replaced parameter character.
4. After all function invocations have finished, the list will contain all possible variants of that instruction.

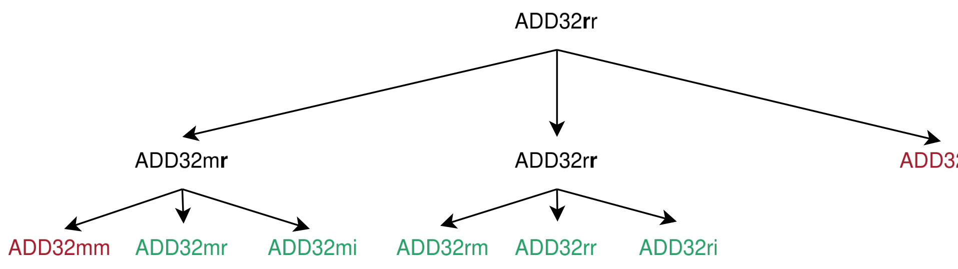


Figure 3.2: *A visualisation of how the possible instruction variants are found for* `ADD32rr`

However, it is important to note that the derived instruction names are not guaranteed valid variants, so we referred to them as *possible* instruction variants. Out of the possible variants seen in Figure 3.2, only `ADD32rr`, `ADD32ri`, and `ADD32rm` are valid variants.

Listing all Parameter Types

For each function that Fadec has, we also save the parameter types it expects. For example, for `ADD32rr`, we store the information that it takes two parameters, which are both general-purpose registers.

3.6.4 Automatically Deriving all Instruction Information

We now describe how we derive all necessary instruction information for encoding instructions using Fadec, including all variants on an instruction.

Finding the Correct Function

First, we check if the instruction name matches our list of Fadec's function names (with some of them being simplified as described before). This is sufficient for simple cases like `ADD32rr`. If there is not function matching the instruction name, we manually build the Fadec name ourselves, considering Fadec's instruction naming scheme. This requires us to collect the following information about the instruction:

1. **Prefix** – If the instruction has the `LOCK_` or `MMX_` prefix, then we retain this prefix, as Fadec also uses it.
2. **Mnemonic** – This is done since Fadec almost always uses the instruction mnemonic, the instruction name used when writing Assembly. We derive it from LLVM using the `llvm::MCInstPrinter` [26] class.
3. **Suffix** – If the instruction requires a suffix for further clarification, we add that suffix. The only suffix we require right now is for `movd` and `movq` instructions, which both require clarification on whether it is from general-purpose to xmm-register (`_G2X`) or vice-versa (`_X2G`). Unsupported suffixes include EVEX suffixes, such as `_sae`.
4. **Definition size** – If the instruction defines a result register, then we query the size of that register.
5. **Parameter characters** – For this, we iterate through the operands and, depending on the parameter type, either add `r` (register), `m` (memory operand), or `i` (immediate). We derive a version with and without implicit operands.

Not all of this information is always required, as it is only used by Fadec if there would otherwise be ambiguous naming. We now attempt to derive the name used by Fadec, taking its instruction naming scheme into account. This is done by creating possible names utilising the information collected about the instruction and comparing it against the list of existing instructions. If we cannot find a match, the provided instruction is unsupported.

Deriving a Parameter Mapping

Once the correct Fadec function is selected, the operands provided by LLVM's instruction must be mapped to Fadec's function parameters. This is done on a "first come, first serve" basis while iterating through the operands of the instruction in question. For register operands, we first consider the regular operands, then the implicit definition, and lastly, the implicit uses. Some implicit registers are always ignored, which includes the flags register (`eflags`) or the floating point control register (`mxcsr`).

If not all explicit parameters can be mapped, we assume the mapping is incorrect and abort. A warning is generated for unmapped implicit operands.

Deriving the Function Variants

Until now, we have successfully mapped an LLVM instruction to its Fadec equivalent. In the next step, we attempt to find all instruction variants we can use. For this, we use the list of variants that was generated before.

The following steps are performed for each possible variant: First, we iterate through the variant's parameters. If all parameters adhere to the rules listed below, we add the variant:

1. If a changed parameter type is a fixed register in LLVM, skip this variant, as we currently do not support this (see also Section 3.7.6).
2. If the operand connected to a changed parameter type is marked as `tied-def` or if it is a definition, it may only be a register. This is obvious for immediate values; we must promote them into registers if they are to be overwritten. Regarding memory operands, we want to promote them into registers as early as possible to avoid repeating load and store operations when the value is used multiple times.

3.7 Limitations

While we believe that `tpde-asmgen` considerably impacts TPDE's development, the employed approach is by no means complete. We want to highlight a few of its limitations. While they are partly irrelevant in the context of TPDE, they might become relevant in the future.

3.7.1 Control Flow

As of now, ϕ-nodes are unsupported. This is because ϕ-node destruction is not a trivial task. Without delving into too much detail, if done incorrectly, problems such as the lost-copy problem can occur [27]. Consider reading the cited paper for more details.

For most types of ϕ-nodes, we believe that they are out of scope for `tpde-asmgen`. ϕ-nodes are mostly used for more complex control flow structures such as loops. However, there is one simple case that we would like to highlight. Consider the C-code in Listing 3.16.

```c
long div(long a, long b) { return a / b; }
```

Listing 3.16: *A C-function that divides two signed 64 bit integers.*

When compiling with `Clang` with `-O2`, the following transformation is performed: If both input numbers fit into 32 bits, a 32 bit division is performed, as it has a significantly lower latency [28]. Specifically, the bits of both input registers are combined using the OR operation and shifted by 32 bits to the right. This means that if the resulting register is zero, the values of both input registers fit into 32 bits.

A ϕ-node is thus generated in the final basic block, as the control flow is split and merged later. In this case, one can avoid ϕ-nodes by not merging the two branches and simply duplicating the final basic block for both possible branches. This technique is known as Tail Duplication and can be done by LLVM using the `llvm::TailDuplicator` [29] class. While this technique has the advantage of supporting ϕ-nodes in simple cases with little effort, it is important to note that this results in a larger code size than necessary due to duplicate code. This might not be problematic for simple cases, but applying this technique to functions with multiple branches after each other results in an exponential code size growth.

Alternatively, we could let LLVM destroy the ϕ-nodes for us and operate on an MIR later in the pipeline. We chose not to move forward with this approach because TPDE, a compiler from an SSA-based IR, already features ϕ-handling. Thus, it would make more sense to integrate with that instead of relying on LLVM. Moreover, our MIR functions would no longer be in SSA form.

Furthermore, supporting control flow constructs would also involve emitting jump instructions. However, we do not know the jump offset at runtime of the machine code generator due to our variant checks emitting instructions of different sizes. This means that we would need to patch the jump instructions to their correct offsets once all offsets in our generated machine code are known.

3.7.2 Stack Allocation

Currently, values created during our machine code generators may only reside in registers; storing values on the stack is unsupported. This also means that any MIR function that creates a stack frame is unsupported.

We do not consider this a significant problem as of now, as we do not assume it is relevant for `tpde-asmgen`'s usage in TPDE. If it does become relevant, it would be rather simple to implement: MIR functions store explicit information about the size of the created stack frame and which values are stored at which offsets. This information could be passed to TPDE's register allocator, which already provides functionality for creating and destroying stack frames.

To support specification functions that use structs, functions having more than six parameters, or functions requiring more than two registers for returning values, we recommend using the *regcall* calling convention. This custom calling convention of Clang assumes that as many parameters reside in registers as possible. This can be used by prefixing the function declaration with `__regcall`.

3.7.3 Constant Pool

When a Machine IR function uses instructions that cannot be encoded into an instruction, it resorts to the constant pool. In this read-only memory region, constant values are stored. This is relevant for TPDE, as some machine code generators regarding floating point operations must use it, for example, when testing if a floating point value is $\pm Inf$.

This could be implemented in two ways:

1. At the start of the machine code generator, write the constant values into the instruction buffer following a jump instruction that jumps over the constants. The constants can then be accessed using `rip`-relative addressing. This is relatively simple to implement since the offset to the constants is known without any further effort. However, it results in read-only data being stored in the `.text`-section. Furthermore, this would emit the constants every time the machine code generator is called, potentially leading to an unnecessarily large binary size increase.

2. Make use of TPDE's data symbol support, which involves registering a symbol with the data that should be stored and then requesting TPDE to patch instructions using this data to use the correct addressing. This would avoid data in the `.text`-section, but is more complex to implement. To avoid generating a new symbol every time our machine code generator is called, we could either add a function into TPDE that checks if a symbol was already added or move the creation of all necessary symbols from their respective machine code generators into one function that TPDE then calls at the start of compilation.

3.7.4 Register Spilling

As we heavily rely on scratch registers for temporary values, our approach is limited to smaller functions without many temporary values being live simultaneously since scratch registers cannot be spilled to the Stack. While we do not assume this to be a problem in the context of TPDE, it is important to note that large functions with many live values will likely lead to the register allocator running out of registers and aborting.

3.7.5 Instruction Support

Using the option `--dump-instInfos`, `tpde-asmgen` iterates through all of LLVM's machine instructions and prints information about the instruction, as well as the Fadec mapping that was derived. If the derivation fails, it also prints further information to determine why that is the case. Currently, 10150 instructions are unsupported. They belong to one of the following ten categories:

- Most AVX-512 and some AVX instructions, especially instructions using an AVX-512 mask register, instructions accessing memory using an xmm-register, and instructions where

Fadec uses suffixes that were not implemented yet. Excluding these instructions alone results in the number of unsupported instructions to drop to 930.
Examples: `VBROADCASTF32X2Z256rr`, `VSHUFPSZ256rmbikz`, `VUNPCKHPDZ256rrkz`

- x86-32 instructions that are unsupported in x86-64.
  Examples: `AAA`, `LDS32rm`
- Operations on other registers that are unimportant in this context.
  Examples: `MOV16sr` (segment register), `INCSSPD` (SSP)
- Instructions from irrelevant extensions.
  Examples: all X87 FPU, SGX, VMX, AES, and MMX instructions
- APX instructions, which are not implemented by hardware yet.
  Examples: `ADD32rr_ND`, `SHR32mCL_NF`
- Instructions where we could not derive the definition size because the defined operand is in memory.
  Example: `LOCK_BTC64m`
- Instructions with incorrectly labelled operand types.
  Examples: `LEA64r` has the operand type `UNKNOWN` instead of Memory operand.
- Pseudo instructions that are not labelled as such.
  Example: `ROR64r1`
- Instructions where Fadec and LLVM use different operand ordering. This only affects `XCHG` instructions with memory.
  Example: `XCHG32rm`
- Control Flow instructions.
  Examples: `JCC_1`, `LOOP`

While many instructions are still unsupported, it is important to note that most will likely never be used in the context of TPDE. Hence, no further effort was made to support more instructions. While basic AVX instructions are supported, implementing more complex machine code generators using AVX will require modifications to our current approach. To support new instructions, one can either update the automatic mapping or define the mapping manually.

3.7.6 Instruction Variant Support

While we support instruction variants, there are some cases that we would like to highlight where our variant support is suboptimal.

Commutable Operands

In this case, the instruction's operands can be swapped without changing the result. For instance, all integer additions are commutable. We can exploit this property to support more types of input operands. For example, if we were to pass an immediate as the first and a register as the second argument to a machine code generator that encodes an addition, it would result in the immediate value being loaded into a newly allocated register before it is added to the second operand. The first operand is also the destination, so it must reside in a register.

Alternatively, we could swap the operands and use the register as the first operand. This would allow us to encode an addition with a constant and avoid allocating another register. This is not implemented.

Variants of Physical Registers

Instructions that use fixed registers sometimes have variants with different operand types. An example is the variable shift instruction, which supports shifting by a constant value apart from shifting by `cl`. However, suppose we pass an immediate value to a machine code generator encoding a variable shift. In that case, the immediate value is loaded into `cl` and a variable shift is encoded instead of a shift by a constant value.

This is because we currently only consider instructions one after another without any context. Once we reach our variable shift instruction in MIR, the immediate value was already moved into `cl` by a `COPY` instruction. This could be avoided by delaying copies into physical registers until they are used and determining if they are required.

Constant Folding

If all input values to a machine code generator are known at compile time, meaning they are all constants, we could perform the calculation at compile time and store the result in the result register. This is known as Constant Folding [6], a well-known transformation that is not implemented by `tpde-asmgen`. This would involve needing much more information about LLVM's instructions, specifically, what calculations they encode.

Size variants

We assume the input register uses the size reported by the MIR function. For instance, an MIR function performing a 64 bit addition will result in a machine code generator operating on 64 bit registers. This is not a problem in this case, as we can still provide smaller registers without leading to incorrect results beyond erroneous flags. However, this is not true for all instructions: instructions such as bitwise rotates would result in incorrect results if an incorrect rotate instruction is used.

Size variants are unsupported. While the used size can be queried from `Reg` classes, scratch registers do not provide such functionality. If we want to change the encoding based on the register size, we must call a different machine code generator in TPDE based on the input register size.

Incorrect variants

We are aware of two cases where the current variant derivation produces incomplete or incorrect results:

- `IMULrri` is a variant of `IMULrr` if one uses the first register twice. This is not found in our current approach; our class does not support such a variable mapping.

- `BTmr` is *not* a variant of `BTrr`, as their behavior differs [21]. This has been fixed by manually mapping these two instructions to Fadec.

3.8 Extensibility

As the primary goal of `tpde-asmgen` also was to simplify the development process for new architectures, we need to analyse if and how easily support for new architectures can be added into `tpde-asmgen`.

3.8.1 Areas of Concern

Most functionality is split into multiple functions. We identified the following areas that are not fully architecture-agnostic:

- Pseudo-instructions that may emit target-specific `mov` instructions. The pseudo-instruction this does not apply to is `SUBREG_TO_REG`; all others call a separate function that emits the fitting encoder call into the output file. Once this function supports the new architecture, all pseudo-instructions are supported.
- While detecting fixed registers is architecture-agnostic, reserving a register may result in a `mov`-instruction being emitted using the same function mentioned before.
- Supporting a new architecture would result in choosing a different encoder (as Fadec only supports x86-64), and thus, a different mapping strategy from LLVM instructions to encoder calls is required, as the current approach will likely not work.
- Instruction generation is non-portable; the subroutine that emits the encoder calls for an instruction, including all variants, must also be rewritten, as it assumes the operand types of x86-64.
- The `AsmOperand` class, our input variable type, might not wrap all input types of other architectures. Furthermore, some helper functions may also emit x86-64 instructions; they would have to be rewritten as well.
- Smaller helper functions, such as determining whether a register is a general-purpose register, would have to be rewritten due to register classes being architecture-specific.

3.8.2 Supporting AArch64

```
bb.0 (%ir-block.2):
  liveins: $w0, $w1
  %1:gpr32 = COPY killed $w1
  %0:gpr32 = COPY killed $w0
  %2:gpr32 = UDIVWr %0:gpr32, %1:gpr32
  %4:gpr32 = MSUBWrrr killed %2:gpr32, killed %1:gpr32, killed %0:gpr32
  %6:gpr64all = SUBREG_TO_REG 0, killed %4:gpr32, %subreg.sub_32
  $x0 = COPY killed %6:gpr64all
  RET_ReallyLR implicit killed $x0
```

Listing 3.17: *MIR of our commonly used `rem`-example, for the AArch64 architecture.*

We will detail `tpde-asmgen`'s compatibility with AArch64, also known as ARM64, as it will be the next architecture supported by TPDE. Listing 3.17 shows our commonly used `rem`-example in AArch64 MIR. It shows that basic concepts, such as `killed` annotations or `COPY` instructions, can be kept without much modification.

In the following paragraphs, we will cover some key aspects of implementing support for AArch64 into `tpde-asmgen`. Firstly, the following operand types are new in AArch64:

- **Shifted register operand** – registers can be shifted or rotated by a constant amount before being passed as operand to an operation. An example is `ADD R0, R1, R2, LSL #3`, which calculates $R0 = R1 + (R2 << 3)$.

- **Extended register operand** – A register value can be extended (zero or sign-extended) from a smaller width (like 8, 16, or 32 bits) to the full 64-bit width before being used in an instruction.

LLVM does not treat them as separate operand types to a different instruction. For example, the aforementioned example of a shifted register operand is represented in MIR using the `ADDXrs` instruction; an extended register operand would use the `ADDXrx` instruction. While this does introduce new parameter types into `tpde-asmgen`, we do not believe that much work is required to support this.

Secondly, AArch64, a load-store architecture, does not allow memory operands in their instructions except for explicit load and store instructions. This differs from x86-64, which allows using memory operands as source or destination in many instructions.

```
LDR R0, [R1, #16] ; Load from address (R1 + 16) into R0
LDR R0, [R1, R2] ; Load from address (R1 + R2) into R0
LDR R0, [R1, #16]! ; Load from address (R1 + 16) into R0, then update R1 to (R1 + 16)
LDR R0, [R1], #16 ; Load from address R1 into R0, then update R1 to (R1 + 16)
LDR R0, [R1, R2, LSL #3] ; Load from address (R1 + (R2 << 3)). Similar to x86-64's scale
```

Listing 3.18: *Some example AArch64 instructions that demonstrate how memory operands work on this architecture*

Furthermore, memory operands are handled differently than on x86-64. Listing 3.18 gives an overview of how memory operands are used. In summary, while x86-64 does not support updating the base register with the displacement, x86-64 can encode more complex addresses using its combination of base register, index register, scale, and displacement. Such memory access on AArch64 would result in multiple instructions to calculate the address. LLVM implements these load operations using multiple instructions, such as `LDRAAindexed` or `LDRABwriteback`.

This design choice affects our current approach in three ways:

1. Instruction variants are more straightforward, as memory operands can only occur in load and store operations. If a memory operand is found, a load instruction can be generated before using the register as an operand.

2. The Address class currently wrapped in AsmOperand is insufficient for AArch64's memory operands, as it does not encode all possibilities. For instance, writebacks are unsupported.

3. Handling addresses passed as input to machine code generators that perform load or store operations is more complex, as it must often be split into multiple instructions, especially when different writebacks are used.

4 Evaluation

4.1 Setup

For evaluating `tpde-asmgen`, we utilised the SPEC CPU 2017 [30] Benchmark Suite. It features 43 Benchmarks across 4 suites that simulate real-world applications. Examples include data compression, public transport route planning and weather forecasting.

To use TPDE with SPEC, we modified the source code of Clang 18.1.2[1] to use the TPDE back-end when instructed.

Clang was then built using GCC 14.2.1, linked statically against LLVM 18.1.2, and linked dynamically against glibc 2.39-17 and libstdc++ 14.2.1-1. `tpde-asmgen`, as well as TPDE, was linked dynamically against LLVM 18.1.6 and built using the same compiler.

All tests were run on a system using an AMD Ryzen 7 5800X3D processor with CPU frequency scaling and frequency boost disabled and all cores set to 3.4 GHz. The system had 16GiB of main memory at 2133 MT/s and 10 GiB of swap space. The system ran Fedora 40, 64 bit, with Linux kernel `6.9.12-200.fc40.x86_64`.

4.2 Correctness

We chose to use the SPEC benchmarks to ensure that `tpde-asmgen`'s machine code generators produce correct code since they also verify the runtime results. This implies that if SPEC's benchmarks compile and run without errors, we can assume that the used machine code generators generate correct code.

The following operations were updated in TPDE to use the machine code generators created by `tpde-asmgen`:

- All binary integer operations (addition, subtraction, multiplication, and, xor, ...).
- 64 bit and 128 bit wide stores into memory.
- `isNaN` and `isNormal`, functions that are both used in the compilation of LLVM's `isfpclass` intrinsic function.

While this does not cover all of TPDE's machine code generators, we can use these to verify the following aspects of our generated code:

- Correct variant matching and encoding for a subset of all possible instructions.
- Correct handling of addresses.
- Correct handling of floating point operations in combination with general-purpose registers.

All benchmarks from the *intspeed* suite were successfully run without any issues. This excludes the 648.exchange2_s benchmark as it is written in Fortran, which Clang does not support.

This test does not cover all instructions supporting the automatic Fadec mapping.

[1] `https://github.com/llvm/llvm-project/tree/llvmorg-18.1.2`

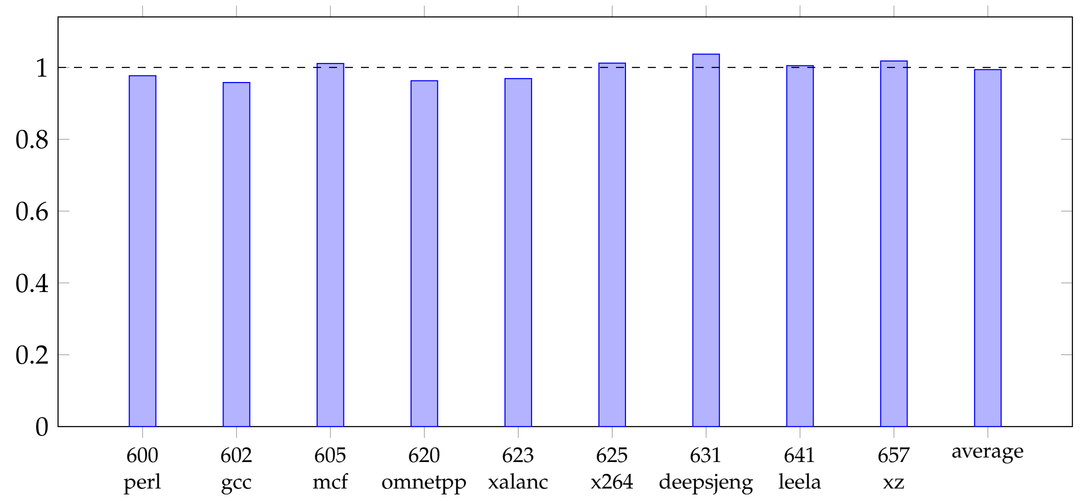


Figure 4.1: *Code generation times of TPDE with* `tpde-asmgen` *partly integrated, relative to TPDE's regular compile times.*

4.3 Code Generation Time

We begin by analysing changes in the code generation time of TPDE. Taking accurate measurements for this required some changes, as measuring the code generation time in SPEC's benchmark build setup is suboptimal. This is because SPEC's benchmarks consist of numerous source files, all compiled separately. This means that TPDE is called multiple times during compilation, with most code generation times within TPDE being in the range of 10^{-4} seconds. Even after adding all separate times together, such short execution times result in a high inaccuracy, even when using multiple runs. To mitigate this, we used the options `-emit-llvm -c` to compile all benchmark files to LLVM-IR bitcode instead of object files. After this, we manually linked them to a single module using `llvm-link`. The result was one bitcode file per benchmark, which we then used as input to TPDE. The exception is the 625.x264 benchmark, which consists of three targets. We compiled them separately and summed up their code generation times. The measurements were taken from the latest TPDE and TPDE with some machine code generators implemented, as described in Section 4.2. We modified both TPDE versions to measure just the code generation using `std::chrono::steady_clock`. This ensured that LLVM bitcode parsing and output file generation were not measured. If the code generation was too fast, code generation was repeated in a loop to ensure each measurement was at least over one second. We repeated the measurements at least five times until the average of all measurements did not change up to its third significant digit.

Results As shown in Figure 4.1, code generation times have improved on average by 0.6%. We believe this is because we encode fewer size variants than before. The old machine code generators use Umbra's x86-64 encoder for encoding instructions instead of Fadec, which exposes only one function for each mnemonic and then queries the register sizes to choose the best-fitting encoding. On the other hand, we only use all size variants when necessary to avoid further operations. For example, when encoding a 64 bit right shift when the register in question is smaller than 64 bits, we must ensure that the upper bits are all set to zero. Otherwise, we would be shifting garbage bits into our result. In that case, the bit width is explicitly checked to avoid generating a zero extension for smaller registers. For all instructions where this does not affect the computation result, we only encode the 32 and 64 bit versions.

To verify whether adding more size variants noticeably affected the code generation times, we introduced checks for 8 and 16 bit register sizes wherever possible. All tests were then re-run using the methodology described before. The results show that encoding all size variants resulted in an average code generation time similar to using TPDE without `tpde-asmgen`, being 0.17% worse than regular TPDE on average.

However, the results in Figure 4.1 also show some benchmarks where the code generation time has increased. For example, the code generation time of 631.deepsjeng has increased by 3.1%. Furthermore, as mentioned already, we have shown that after encoding all size variants, our code generation times are worse than regular TPDE. We attribute this in part to two consequences of our approach. Firstly, since our input parameter of type `AsmOperand` can hold different types, we often query said type to change our encoding. Furthermore, some helper functions behave differently based on the wrapped type, such as when checking whether we may overwrite a register. This introduces more branches into our machine code generators than before.

Secondly, our current use of `tpde-asmgen`'s machine code generators results in unnecessary conversions between `ScratchRegister` and `Reg` when calling our machine code generators using `Reg` parameters. Suppose a `Reg` parameter used as input also holds the computation result at the end. In that case, it is then converted to a `ScratchRegister` by destroying the `Reg` and allocating a `ScratchRegister` to the register holding the value. After we left our machine code generator, we set the result variable, being of the type `Reg`, to the `ScratchRegister`'s currently allocated register using `setValue`, which does the same conversion the other way around. As smaller, temporary computations might also use `tpde-asmgen`'s machine code generators, it makes sense to preserve the option to use a `ScratchRegister` as the return value. This could be solved by implementing a custom return type similar to `AsmOperand`, which can both be a reference to `Reg` or `ScratchRegister`. This would result in fewer type conversions in the aforementioned case and still allow `tpde-asmgen`'s machine code generators to be used in temporary computations.

4.4 Binary Size

In this section, we compare the sizes of binaries compiled by TPDE before and after `tpde-asmgen` was integrated. As `tpde-asmgen` does not always encode the exact instructions as before, observing changes in binary size is interesting, though it is not a primary concern for us. We compared the compiled binary size of all benchmarks before and after our partial integration of `tpde-asmgen`'s code.

Results As shown in Figure 4.2, binary size grew by 1.8% on average after introducing `tpde-asmgen`. All binaries increased in size, except for the binary of the 605.mcf benchmark, whose size did not change.

We attribute this to some omitted register variant encodings. In the case of binary shifts, we cannot generate 8 bit and 16 bit variants, as LLVM's instruction selection resorts to 32 bit shifts in that case, assuming that the integer is zero-extended. In case of a right shift on 8 or 16 bit registers, we not only encode a larger shift size than necessary but also need to generate a zero-extension to at least 32 bits.

Other operations whose encoding grew as well are all constant shifts and rotates. As mentioned in Section 3.7.6, we currently do not encode constants into shifts, but first move them into the `cl` register, resulting in two instructions instead of one.

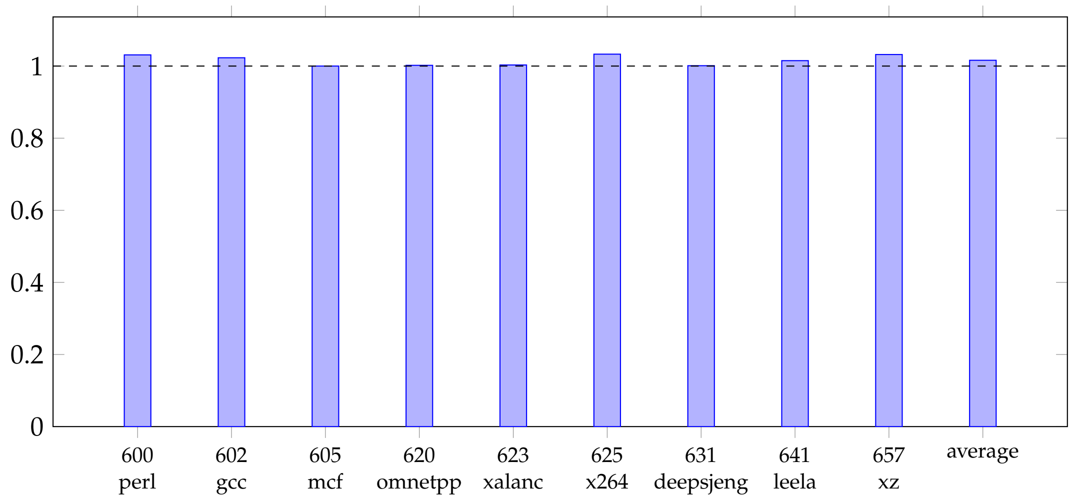


Figure 4.2: *Binary size comparison of SPEC benchmarks generated by TPDE with* `tpde-asmgen` *partly integrated, relative to binary sizes generated by regular TPDE.*

4.5 Runtime Performance

Finally, we want to ensure that the integration of `tpde-asmgen` did not significantly worsen runtime performance. To test this, we ran all benchmarks of SPEC's intspeed suite, again excluding 648.exchange2_s. This was done with at least three iterations per benchmark, ensuring that the relative standard deviation was always below 1%.

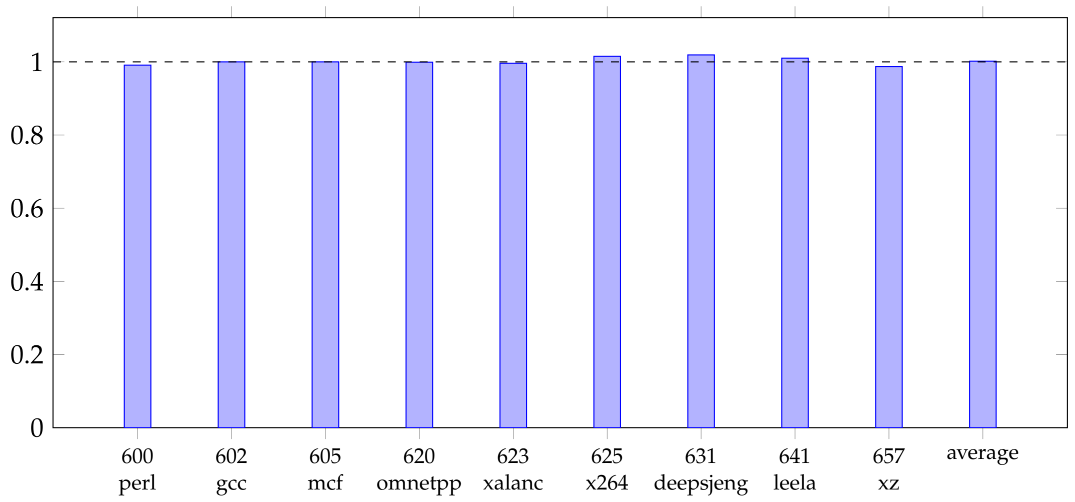


Figure 4.3: *Benchmark runtimes compiled by TPDE with* `tpde-asmgen` *partly integrated, relative to when using TPDE without modifications.*

Results As seen in Figure 4.3, the runtime performance of TPDE's generated code is largely unaffected by the integration of `tpde-asmgen`. On average, runtimes increased by 0.18%. Out of all benchmarks, only the performance differences of 625.x264, 631.deepsjeng, 641.leela, and

657.xz are beyond the margin of error. We attribute this slight runtime increase to our generation of inefficient instruction sequences, which were already covered in Section 4.4.

When only encoding 64 bit size variants where it does not affect the computation result, our average runtimes increase noticeably, being 2.1% worse than regular TPDE. This is because using a superregister after an operation was performed on a subregister results in a delay of a few clock cycles. This is known as a partial register stall [31]. Implementing 8 bit size variants in addition to 32 and 64 bit ones, however, only resulted in a runtime decrease within the margin of error. While there might be test cases that have a runtime impact due to our omission of 8 and 16 bit variants in the aforementioned cases, we want to point out that this would be just a temporary issue. This is because we currently employ both old machine code generators and newer ones using `tpde-asmgen`. This issue will be resolved once all machine code generators use `tpde-asmgen` and thus do not encode smaller register size variants than necessary.

5 Related Work

Deriving code generation patterns from higher-level languages is not a new approach. In this chapter, we will discuss previous approaches and compare their goals and strategy with `tpde-asmgen`.

QEMU dyngen In the original version of QEMU, a dynamic binary and system translator, a tool named dyngen [32] was utilised to translate operations to the target's ISA. It generates a code generator from so-called *micro operations*, functions written in C and compiled to an object file. At compile-time of QEMU, these object files are parsed to extract the encoded instructions as bytes. When a constant parameter is required, it is declared in the C source code as an `extern` variable, resulting in the compiler emitting a *relocation*, signalling that a variable is missing. A simple example where this is required is an addition with a constant. The relocation is detected by dyngen, resulting in the code generator generated by dyngen replacing the relocated variables with a provided constant at runtime of QEMU.

Similar to `tpde-asmgen`, dyngen is only run during compile-time of the compiler/translator, with the resulting code generators used at runtime. An upside of dyngen is that it is mostly architecture-agnostic, as it does not have to know which instructions were generated by the compiler. The exception to this is architecture-specific handling of relocations. However, dyngen's technique could not be employed in TPDE. This is because dyngen takes the generated instructions as-is, including the registers used in the instructions. This is not a problem in the context of QEMU, as it uses virtual registers stored in memory for correct translation. Nevertheless, TPDE operates on physical registers. This would require patching all instructions to use other registers than anticipated. Furthermore, the same operation using a different register or operand type has to be encoded in dyngen using a different micro operation. `tpde-asmgen` does not require this, as it can take multiple types as input and change the encoding accordingly.

Copy-and-patch The copy-and-patch [33] code generation technique proposed by Xu et al. employs so-called *stencils*, being binary implementations of, for example, bytecode instructions or node types from an abstract syntax tree. At runtime, a copy-and-patch code generator concatenates the required stencils and patches missing information, such as constants or stack offsets. The stencil collection is automatically generated using their MetaVar compiler, which is run at compile-time of the copy-and-patch code generator and uses stencil generators. A stencil generator is a templated C++ function that heavily relies on macros and MetaVar-specific function parameters to represent specific behaviour, such as an integer addition. During template instantiation, the stencils are compiled to object code using Clang and information is extracted from the generated object code using LLVM. As in dyngen, relocations are used to mark placeholder values.

Similar to dyngen, MetaVar's generated stencils only support specific parameter types. Thus, different instruction variants must be encoded as multiple stencils. Xu et al. implemented a high-level language compiler backend whose stencil library employed 98831 stencils using 17.5 MB of memory. Furthermore, `tpde-asmgen`'s machine code generators are easier to configure, as the specification functions can be written in any language that can be lowered to LLVM-IR and do not require in-depth knowledge of how `tpde-asmgen` works. In contrast, MetaVar's stencil generators require significantly more implementation effort.

Drescher et al. [34] improved upon the copy-and-patch technique by allowing the stencils, here named *templates*, to be compiled from LLVM MLIR specification functions. Similar to `tpde-asmgen`, these MLIR functions demonstrate the behaviour of the resulting code. They are lowered to LLVM-IR functions that use variables from memory, whose location is referred to as the value storage. Next, similar to the original copy-and-patch approach, they are compiled to binary objects to extract the generated machine code. The extracted templates are then used during compilation to generate code. To avoid excessively loading from and storing into the value storage, so-called cache registers temporarily hold values used in subsequent computations.

While this alleviated many disadvantages of Xu et al.'s approach, both approaches differ from `tpde-asmgen` in that they are independent code generation backends. In contrast, `tpde-asmgen` generates code for an existing backend infrastructure, which includes a register allocator.

BURS BURS [35] is a technique for instruction selection by performing bottom-up pattern matching on a program's abstract syntax tree. The patterns are defined in a table. Each pattern includes a set of rewriting rules, a cost estimation, and an action that is performed. Using dynamic programming, the least-cost instruction sequence is selected for each subtree. The difficulty of creating such tables is that all possible trees have to be anticipated. Proebsting [36] improved on this technique by proposing a tool for automatically generating BURS tables at compile-time of the compiler. As input, Proebsting's tool receives a list of the available operators and a list of grammar rules.

Apart from `tpde-asmgen` using Proebsting's idea of generating aspects of a compiler automatically at compile-time, the approach and goal differ. BURS is only used for instruction selection, while `tpde-asmgen` derives a strategy for instruction selection and register allocation. Furthermore, Proebsting does not employ external compiler frameworks for the derivation and instead relies on the target specification received as input.

BURS has not seen significant adoption in practice. Since then, more advanced techniques have been developed, such as performing instruction selection on the data flow graph [37].

6 Conclusion and Future Work

6.1 Conclusion

In this thesis, we outlined problems with the current use of hand-written machine code generators by TPDE, including non-portability and complexity. To alleviate these issues, we implemented `tpde-asmgen`, a C++ tool for automatically generating register-agnostic machine code generators from branch-free MIR. It is run during the build process of TPDE, taking behavioural specifications compiled to LLVM-IR as input. It supports a large subset of LLVM's MIR instructions for x86-64, including instructions requiring specific registers to function and pseudo-instructions. By introducing the `AsmOperand` class, `tpde-asmgen` can emit machine code generators supporting as many input types as possible without creating multiple generators for each input type variant. Special care was taken to make the generated code easily reviewable.

We evaluated `tpde-asmgen`'s machine code generations through a limited integration into TPDE both in terms of code generation time and runtime performance of the SPEC benchmark suite. Compared to TPDE without `tpde-asmgen`, we measured an average code generation time decrease of 0.6% and an average runtime increase of 0.18%.

In conclusion, we see `tpde-asmgen` and the approach it employs as an impactful change to TPDE's development process. It makes the current code easier to read and verify and simplifies future support for new architectures, as many components of TPDE can now be architecture-agnostic. However, this comes at the cost of being dependent on LLVM's instruction selector to generate desirable code. We have shown that this is not the case regarding operations on partial registers, where LLVM's backend often resorts to integer promotion instead of operating on the desired subregisters.

Regarding the development process, while we consider the automatic mapping from LLVM's instructions to Fadec functions a success, we would like to point out that deriving this strategy took several weeks. In hindsight, manually mapping the required instructions could have been a better choice. This would have left us more time to focus on a more extensive implementation of `tpde-asmgen` into TPDE and adding support for the constant pool.

6.2 Future work

In Section 3.7, we already outlined some of `tpde-asmgen`'s limitations. We will now discuss more ways in which `tpde-asmgen` and its approach could be improved and repeat some previously mentioned limitations that we consider of higher priority.

The next step is implementing support for MIR's constant pool. Strategies on how to implement this were given in Section 3.7.3. This would allow support for all `isfpclass` functions, drastically reducing the number of hand-written lines within TPDE.

Next, all machine code generators in TPDE should be updated to use `tpde-asmgen`. This will exclude smaller snippets that specification functions cannot generate, such as subtracting a specified value from the stack pointer or performing a `lea` operation.

After `tpde-asmgen` has been fully implemented into TPDE, we can work on implementing support for AArch64.

6.2.1 Verification of the Automatic Fadec Mapping

As the mapping between LLVM's machine instructions and the emitted encoder calls to Fadec is done automatically, an essential aspect of correctness is to verify that this mapping is correct. Currently, this verification has only been done for the instructions used within TPDE, as the correctness of the generated code has been verified in these cases.

Manual verification is not feasible due to the number of instructions that must be checked. A possible automated approach to verify the correctness of this mapping is to let LLVM encode each machine instruction and check if the resulting bytes are the same as the ones emitted by Fadec.

However, it is important to note that instruction encoding in x86-64 is ambiguous. One reason is that instruction prefixes can be encoded in different orders [21]. Thus, a byte-by-byte comparison might lead to many false negatives, especially if LLVM and Fadec handle prefix encoding differently.

6.2.2 Compiler plugin

Initially, `tpde-asmgen` was planned to be a compiler plugin for Clang. The advantage of this is that one would not need to externally compile specification functions to LLVM-IR and use them as input to an external tool. Instead, they could be supplied as lambda expressions. An example from a prototype is shown in Listing 6.1. During the compilation of TPDE, the compiler plugin would then generate functions from the provided lambda expression.

```
CodeGenerator cg;
Reg resultReg = cg.subst<+[](int x, int y) -> int {
    return x + y;
}>({arg1, arg2});
```

Listing 6.1: *A concept of what the usage of tpde-asmgen as a compiler plugin could look like*

However, this approach comes with some disadvantages. Firstly, it would force TPDE to always be built with Clang. Secondly, our current approach of extracting the MIR functions would not work in this context since the `--stop-after` command line option would apply to the entire compilation and not just the lambda functions. Thirdly, and most notably, we cannot emit C++ code at this stage but must emit our machine code generators as LLVM-IR functions, adding another layer of complexity.

While this concept offers some benefits over our chosen approach, we believe that `tpde-asmgen` in its current form already brings many advantages and that implementing it as a compiler plugin would not be worth the extra effort, especially considering the aforementioned disadvantages.

Abbreviations

SSA	Static Single Assignment
IR	Intermediate Representation
JIT	just-in time
AOT	ahead-of-time
MIR	Machine IR
ISA	Instruction Set Architecture

List of Figures

Listings

Bibliography

[1] T. Neumann. “Efficiently compiling efficient query plans for modern hardware.” In: *Proc. VLDB Endow.* 4.9 (June 2011). DOI: 10.14778/2002938.2002940.

[2] *1. Building a JIT: Starting out with KaleidoscopeJIT – LLVM 20.0.0git documentation*. URL: https://llvm.org/docs/tutorial/BuildingAJIT1.html (visited on 07/27/2024).

[3] A. Engelke and T. Schwarz. “Compile-Time Analysis of Compiler Frameworks for Query Compilation.” In: *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2024, pp. 233–244. DOI: 10.1109/CGO57630.2024.10444856.

[4] A. Engelke and T. Schwarz. “Building a fast Back-end for LLVM.” In: *Eighth LLVM Performance Workshop at CGO*. Mar. 2, 2024.

[5] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. “Global value numbers and redundant computations.” In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’88. 1988. DOI: 10.1145/73560.73562.

[6] F. E. Allen and J. Cocke. *A Catalogue of Optimizing Transformations*. International Business Machines Corporation (IBM), 1971.

[7] M. Braun, S. Buchwald, S. Hack, R. Leißa, C. Mallon, and A. Zwinkau. “Simple and Efficient Construction of Static Single Assignment Form.” In: *Compiler Construction*. 2013. DOI: 10.1007/978-3-642-37051-9_6.

[8] C. Lattner and V. Adve. “LLVM: a compilation framework for lifelong program analysis & transformation.” In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 2004. DOI: 10.1109/CGO.2004.1281665.

[9] *GIMPLE - GCC Wiki*. URL: https://gcc.gnu.org/wiki/GIMPLE (visited on 05/01/2024).

[10] *The LLVM Compiler Infrastructure*. URL: https://llvm.org/ (visited on 05/01/2024).

[11] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman. “Julia: A Fast Dynamic Language for Technical Computing.” In: *ArXiv* abs/1209.5145 (2012). DOI: 10.48550/arXiv.1209.5145.

[12] *Overview of the compiler - Rust Compiler Development Guide*. URL: https://rustc-dev-guide.rust-lang.org/overview.html#overview-of-the-compiler (visited on 05/01/2024).

[13] D. A. Terei and M. M. Chakravarty. “An LLVM Backend for GHC.” In: *Proceedings of the Third ACM Haskell Symposium on Haskell*. Haskell ’10. Baltimore, Maryland, USA, 2010. DOI: 10.1145/1863523.1863538.

[14] *Swift – Apple Developer*. URL: https://developer.apple.com/swift/ (visited on 05/02/2024).

[15] S. K. Lam, A. Pitrou, and S. Seibert. “Numba: a LLVM-based Python JIT compiler.” In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. LLVM ’15. Austin, Texas, 2015. DOI: 10.1145/2833157.2833162.

[16] *llgo - A Go compiler based on LLVM*. URL: https://github.com/goplus/llgo (visited on 07/27/2024).

[17] *LLVM Language Reference Manual*. URL: https://llvm.org/docs/LangRef.html (visited on 05/02/2024).

[18] G. Blindell. *Instruction selection: Principles, methods, and applications*. Jan. 2016. DOI: 10.1007/978-3-319-34019-7.

[19] *System V application binary interface. AMD64 architecture processor supplement*. URL: https://gitlab.com/x86-psABIs/x86-64-ABI/-/jobs/6235340483/artifacts/file/x86-64-ABI/abi.pdf (visited on 05/20/2024).

[20] *Machine IR (MIR) Format Reference Manual*. URL: https://llvm.org/docs/MIRLangRef.html (visited on 05/02/2024).

[21] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html. Intel Corporation. Apr. 2016.

[22] *llvm::MachineFunction Class Reference*. URL: https://llvm.org/doxygen/classllvm_1_1MachineFunction.html (visited on 05/20/2024).

[23] *llvm::Function Class Reference*. URL: https://llvm.org/doxygen/classllvm_1_1Function.html (visited on 08/09/2024).

[24] *llvm::TargetInstrInfo Class Reference*. URL: https://llvm.org/doxygen/classllvm_1_1TargetInstrInfo.html (visited on 07/27/2024).

[25] *Fadec — Fast Decoder for x86-32 and x86-64 and Encoder for x86-64*. URL: https://github.com/aengelke/fadec/ (visited on 08/05/2024).

[26] *llvm::MCInstPrinter Class Reference*. URL: https://llvm.org/doxygen/classllvm_1_1MCInstPrinter.html (visited on 07/28/2024).

[27] P. Briggs, K. D. Cooper, T. J. Harvey, and L. T. Simpson. "Practical improvements to the construction and destruction of static single assignment form." In: *Softw. Pract. Exper.* 28.8 (1998). DOI: 10.1002/(sici)1097-024x(19980710)28:8<859::aid-spe188>3.0.co;2-8.

[28] A. Abel and J. Reineke. "uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures." In: *ASPLOS*. ASPLOS '19. ACM, 2019. DOI: 10.1145/3297858.3304062.

[29] *llvm::TailDuplicator Class Reference*. URL: https://llvm.org/doxygen/classllvm_1_1TailDuplicator.html (visited on 08/05/2024).

[30] Standard Performance Evaluation Corporation. *SPEC CPU® 2017*. URL: https://www.spec.org/cpu2017/ (visited on 08/08/2024).

[31] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual Volume 1*. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html. Intel Corporation. Apr. 2024.

[32] F. Bellard. "QEMU, a fast and portable dynamic translator." In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC '05. Anaheim, CA: USENIX Association, 2005.

[33] H. Xu and F. Kjolstad. "Copy-and-patch compilation: a fast compilation algorithm for high-level languages and bytecode." In: *Proc. ACM Program. Lang.* (2021). DOI: 10.1145/3485513.

[34] F. Drescher and A. Engelke. "Fast Template-Based Code Generation for MLIR." In: *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction*. 2024. DOI: 10.1145/3640537.3641567.

[35] E. Pelegrí-Llopart and S. L. Graham. "Optimal code generation for expression trees: an application of BURS theory." In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '88. 1988. DOI: 10.1145/73560.73586.

[36] T. A. Proebsting. “BURS automata generation.” In: *ACM Trans. Program. Lang. Syst.* (1995). DOI: 10.1145/203095.203098.

[37] D. R. Koes and S. C. Goldstein. “Near-Optimal Instruction Selection on DAGs.” In: *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO ’08. 2008. DOI: 10.1145/1356058.1356065.