Informatics

Typsysteme auf Bit-Ebene für Assembler-Sprachen

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Florian Mihola, BSc
Matrikelnummer 00304850

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Franz Puntigam

	Florian Mihola	Franz Puntigam

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Informatics

Bit-Level Type Systems for Assembly Languages

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Florian Mihola, BSc
Registration Number 00304850

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Franz Puntigam

	Florian Mihola	Franz Puntigam

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Florian Mihola, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Florian Mihola

Kurzfassung

Typsysteme helfen uns auf mehreren Wegen dabei, Software zu schreiben: Sie sind eine Art Sicherheitsnetz und machen es schwieriger, aus Versehen unsichere oder fehlerhafte Programme zu schreiben. Typen sind auch eine Form der Dokumentation und ein Vertrag zwischen mehreren Parteien. Manche Typsysteme sind so weit fortgeschritten, dass sie sogar Vorschläge machen können, wie eine Lücke in einem unfertigen Programm gefüllt werden könnte.

Fortschrittlichere Typsysteme findet man üblicherweise in Hochsprachen, während hardwarenahe Sprachen oft untypisiert sind. Um Typsysteme und ihre Vorteile auch in hardwarenahen Sprachen nutzen zu können, ist es also wünschenswert, nicht nur die Befehle einer Hochsprache zu übersetzen, sondern auch ihre Typen. Ein Typsystem für eine Assembly-Sprache muss also Möglichkeiten bieten, die gewünschten Eigenschaften der Typen darzustellen, und zwar vorzugsweise auf allgemeine Weise, und nicht speziell auf eine bestimmte Quellsprache zugeschnitten.

Diese Arbeit handelt weniger von der Übersetzung von Typen. Stattdessen soll folgende Forschungsfrage beantwortet werden: „Wie kann ein Typsystem einzelne Bits, aber auch Gruppen von Bits verarbeiten und sicherstellen, dass diese korrekt verarbeitet werden?“

Zunächst definieren wir Fragen, die sich mit grundlegenden Eigenschaften einer Sprache und der Erstellung von Software befassen. Danach wird ein Prototyp eines Assemblers für eine Teilmenge des Arm A64 Befehlssatzes beschrieben. Dieser soll ein einfaches Typsystem aufweisen, in dem die kleinste Einheit das Bit ist. Es sollen Werkzeuge zur Verfügung stehen, um Typen bis zur kleinsten Einheit zu definieren und maximale Kontrolle zu erhalten. Im dritten Schritt werden die zuvor gestellten Fragen anhand des Prototypen beantwortet. Zu guter Letzt wird die Forschungsfrage anhand der erhaltenen Ergebnisse beantwortet.

Es lässt sich schließen, dass das vorgeschlagene Typsystem implementiert werden kann und es dem Benutzer erlaubt, verschiedene Varianten geläufiger Typsystemfeatures zu implementieren. Einige eher unübliche Features und Sicherheitschecks für spezielle Anwendungen, welche aber üblicherweise in typisierten Programmiersprachen nicht zu finden sind, können ebenfalls implementiert werden. Die Definition eines Programms und seiner Typen kann jedoch einen größeren Aufwand erfordern als gewöhnlich. Weiters kann die Zeit, die für die Typüberprüfung benötigt wird, ausarten, wenn man nicht genügend Sorgfalt walten lässt.

Abstract

Type systems help us to write software in various ways: they act as safety nets and make it harder to accidentally write unsafe or incorrect programs. Types are also a form of documentation and a contract between different parties. Some type systems have advanced to a stage where they may even suggest source code to fill holes in an unfinished program.

But the more advanced features of type systems are usually only found in high-level languages, while low-level languages like assembly are often untyped. In order to introduce type systems and all their benefits to lower level languages, we may thus wish to translate not only the instructions of a high-level language, but also its types. A type system for an assembly language, then, needs to be able to express the desired properties about types, preferably in a general fashion, rather than in a way that is specifically tailored to one specific source language.

In this thesis, we are not concerned so much with the transformation of types, but with the following main research question: "How can a type system handle individual as well as grouped bits and verify that they are processed correctly?"

First, we define a set of questions about basic features of a language, and how these features can be used to write software. Second, a prototype assembler for a subset of the Arm A64 instruction set is described. This assembler shall feature a simple type system in which the smallest unit is the bit. The goal is to give the user the tools to define types which are defined down to the smallest unit, giving maximum control. In the third step, this prototype is used to find answers to the questions, which were defined earlier. Finally, the main research question is answered by considering the results that were gained.

The final conclusion is that a type system like the one proposed may indeed be implemented and allows the user to express different variations of common type system features. Some rather uncommon features and special purpose safety checks, which are usually not found in typed programming languages, can be implemented as well. The definition of programs with their types may require more effort than in other languages, and if appropriate care is not taken, the time spent on type checking may get out of hand.

Contents

CHAPTER 1

Introduction

Type systems help us to write software that is **safe and correct**. They act as safety nets, when they highlight incompatible types. Types can also be a kind of documentation. Advanced type systems, such as that of Idris—see for example Brady [Bra17]—are able to suggest code through their knowledge of which values have a type that "fits" into an unfinished part of a program. It is—in contrast—in **low-level** languages that we often do not have this power at our disposal. Furthermore, even if we can rely on the correctness of types to help us with the construction of programs in high-level languages, how can we be certain that—once the program is translated to machine language—the resulting program is the exact equivalent of the source program; moreover how do we know the types are still intact, when in fact there are no types in machine language? If we are not content with simply trusting the compilation process in the absence of types, we may look to methods for transforming program and types at the same time. For that, a target type system to accompany the target machine code is required. With a specific high-level language in mind, the design of the assembly-level type system might follow naturally from the type system of the source language.

In this thesis, however, I am not concerned with the transformation of types from one level to another, but rather with a type system for a **realistic assembly language** which stands on its own, and which—in a first trial—is meant to be written by hand. I wish to explore the possibilities of a type system that does not conform to any language but the assembly language specifically, via the construction of machine-level programs directly in an assembly language, only aided by a simple type system for the simple operations of the machine instructions.

1.1 What Types are Made Of

But how do these types compare to the types of high-level languages; and what can be considered a successful implementation of a (high-level) type in the low-level type system?

Much of the problem lies in deciding what the requirements for a type are; once this has been decided it is rather straightforward to decide whether it is possible to implement them or not.

```c
enum Source { Tape  = 2,
              Radio = 1 };
enum Mute   { On    = 1,
              Off   = 0 };
```

(a) Enumerations in C, using custom mappings

```haskell
data Source = Tape | Radio
data Mute   = On   | Off

instance Enum Source where
  fromEnum Tape  = 2
  fromEnum Radio = 1
  toEnum   2     = Tape
  toEnum   1     = Radio
```

(b) A very different approach in Haskell. . .

```c
void play(enum Source src,
  enum Mute mute) {
  if (mute == On) {
    printf("Muted\n");
    return;
  }
  if (src == Tape)
    play_tape();
  else if (src == Radio)
    play_radio();
  else panic();
}

play(Radio, Off); // ok
play(Off, Tape);  // panics
playClever(Off, Tape);
  // Muted, no error
```

(c) Enumerations in C are not as safe as we would like

```haskell
play :: Source -> Mute -> IO ()
play src mute =
  case (mute, src) of
    (On, _)     -> muted
    (_, Tape)   -> playTape
    (_, Radio)  -> playRadio

play Radio Off  -- Plays Radio
play Off   Tape -- compiler errors
```

(d) . . . leads to more safety

Figure 1.1: Enumeration types, which hold just single bits of information

Consider, for example, a simple enumeration. The C programming language—described by Kernighan and Ritchie [KR88]—includes syntax to specify an enumeration type along with its set of values, which is then usually translated to some integer type. Two C enums are defined in Figure 1.1a, Figure 1.1b shows what they could look like in Haskell—described by Jones [Jon03]. The two `play` functions in Figures 1.1c and 1.1d implement the same logic: Play music from either the radio or a cassette tape, but do not output any audio if `Mute` is `On`. Not only does C let us mix enum values, it will even allow us to assign values to a variable that lie outside of the perceived range of its type. Checking for `Tape` and Radio should cover all cases, but if we use the `Mute` value

`Off`, C compilers do not even give a warning. `play(Radio, Off)` does what we would expect, `play(Off, Radio)` panics because an invalid value snuck in—but is it really invalid? Ultimately, all values in this example are integers. If we “cleverly optimize” `if (mute == On)` to just `if (mute)`—`On = 1` is `true`, `Off = 0` is `false`—we end up with `playClever(Off, Tape)` which does not produce any errors—even though the arguments are reversed—but does not do what we expect either. The Haskell version in Figure 1.1d does not allow us to implicitly convert between types, and with the appropriate compiler options it will produce a warning, unless all possible cases are handled. Then again, it is not even possible to assign any value other than `Tape` or `Radio` to `src`. Rust—see Klabnik and Nichols [KN19]—and Zig—see the Language Reference for version 0.7.1 [Zig21]—feature two more—and again, different—notions of enumerations. At first glance, both seem very similar to each other. They feature a dedicated `enum` keyword, and it is possible to manually assign integers to enum values. They both prevent us from accidentally casting from one enum type to another. Rust’s enumerations may hold additional data; they are more akin to Haskell’s `data` types. Zig’s enums, on the other hand, let the user choose the integer type that is used to represent the type at runtime, but it will only allow types which are actually able to hold the required values. In a similar vein, when explicitly transforming one enum to another, Zig does catch some subtle errors like casting from an integer that has no match in the destination enum. It is thus possible to convert an `On` to a `Radio`, but it is an error to cast an `Off` value in the same way, see Figure 1.2: the last statement produces an error—`Off` = 0, but 0 does not match `Tape` or `Radio`—while the one previous to it does not. Zig further differentiates between exhaustive and non-exhaustive enumerations, the latter ones behave more as those in C.

```
const Source = enum(u2) {
    Tape = 2, Radio = 1
};
const Mute = enum {
    On = 1, Off = 0
};

const muteOn:  Mute   = .On;  // = 1
const muteOff: Mute   = .Off; // = 0
const srcOn:   Source = @intToEnum(Source, @enumToInt(muteOn));
const srcOff:  Source = @intToEnum(Source, @enumToInt(muteOff));
```

Figure 1.2: Familiar types, this time in Zig parlance

So if we decide to measure our type system’s ability to express enumerations, which features and restrictions do we need to demonstrate?

In order to evaluate a low-level type system, a precise conception of the **most basic features** seems to be necessary. We should then also consider the ease of use for these features. My approach to evaluation is then to split up any idea of a type into whatever can be said to characterize them. An enumeration could mean named integer constants and little more, in one context, and strict differentiation of incompatible values, in another one. In assembly language, and a type system on that level, C's enum concept is practically non-existent. Maybe we can bring some of the basic features of other type systems into an assembly language. What those are is a matter of personal taste, at least to some extent. We might say, for example, that a `Mute` can only be 0 or 1, but never 2. Or, we might require the first argument of function `play` to be marked a `Source`, and the set of acceptable values would then follow from that.

```haskell
tapeValue, radioValue :: Nat2
tapeValue  = Nat2 2
radioValue = Nat2 1

tape, radio :: BitAddr -> NamedExpr
tape  = singletonTag (TNString "Tape")  tapeValue
radio = singletonTag (TNString "Radio") radioValue

source :: BitAddr -> NamedExpr
source = enum (TNString "Source") [ tape, radio ]
```

Figure 1.3: One possible definition of the assembly level `Source` type, defined as Haskell code

The proposed type system allows us to mimic some of these features. Type information and **abstract machine states** are built from rather **simple formulas**, which express things such as a bit being 0, or 1 or unknown, and, a tag being attached to bits, or not. Figure 1.3 shows how the `Source` type could be implemented. First, the integer values 2 and 1, for `Tape` and `Source` are defined. Here, `Nat 2` is a wrapper around an unsigned integer value. It signifies a width in bits of 2, which is used whenever a `Nat 2` is injected into a structure describing the state of an abstract machine. If `Nat 2` is written to a register, only two bits are written; the same value 2 as a `Nat 64` would write all of the 64 bits of a register. A type in this system is a Haskell function, and most of them may be built from a small set of predefined functions. In this example, `singletonTag` takes a type name in the form of a string and a value. The resulting type is an abstract machine state. In the case of `tape`, it includes both the bit pattern to represent the value 2, as well as tags on these two bits which mark them as "Tape bits." `source` combines `tape` and `radio`: Just as a `Source` value is either `Tape` or `Radio`, `source` describes a set of two possible machine states, one where the specified bits are set to value 2 and marked as "Tape" and one where they are set to value 1 and marked as "Radio." Note how in many high-level languages we would say that `Source` is a type and `Tape` and `Radio` are its values or constructors. In contrast, in this example, `tape` and `radio` are considered to be types just like `source`. This means that we may also use the former two on their own, as we will see shortly.

```
let makeTape r =
      do postf (\pc instr state ->
                 map (setTypeTag (TNString "Tape") (RegBit r 0)))
         movImm r 2
    makeRadio r =
      do postf (\pc instr state ->
                 map (setTypeTag (TNString "Radio") (RegBit r 0)))
         movImm r 1
    makeMuteOff r =
      do postf (\pc instr state ->
                 map (setTypeTag (TNString "MuteOff") (RegBit r 0)))
         movImm r 0
    cmpSource x y =
      do lf (\pc -> atPC pc $ source $ RegBit x 0)
         lf (\pc -> atPC pc $ source $ RegBit y 0)
         cmp x y
in makeProgram $
   do "main"      # do makeTape    (X 0)
                       makeMuteOff (X 1)
                       bl "play"
                       makeRadio   (X 0)
                       makeMuteOff (X 1)
                       bl "play"
      "halt"      # b "halt" -- placeholder
      "play"      # do lf (\pc -> atPC pc $ source $ RegBit (X 0) 0)
                       lf (\pc -> atPC pc $ mute   $ RegBit (X 1) 0)
                       cmp XZR (X 1)
                       bCond Equal "testTape"
                       b "muted"
      "testTape"  # do makeTape (X 9)
                       cmpSource (X 0) (X 9)
                       bCond NotEqual "testRadio"
                       b "playTape"
      "testRadio" # do makeRadio (X 9)
                       cmpSource (X 0) (X 9)
                       bCond NotEqual "panic"
                       b "playRadio"
      "panic"     # do lf impossible
                       b "panic"
      "playTape"  # do lf (\pc -> atPC pc $ tape   $ RegBit (X 0) 0)
                       ret -- placeholder
      "playRadio" # do lf (\pc -> atPC pc $ radio  $ RegBit (X 0) 0)
                       ret -- placeholder
      "muted"     # do lf (\pc -> atPC pc $ muteOn $ RegBit (X 1) 0)
                       ret -- placeholder
```

Figure 1.4: The main function calls play twice; all nested do blocks are not strictly necessary, but they make for clearer source code, because they allow all assembly instructions to be indented deeper than their labels

A `play` function is called twice in Figure 1.4: Assembly programs are given in a **domain specific language**, or DSL, and `let` is used to define functions which are used the way macros are. The DSL is used to build a sequence of assembly instructions; these instructions may be modified by other commands which are listed before them. The details are explained as we go along. In this example, each of the macros contains only a single assembly instruction, the important parts are the applications of `postf` and `lf`. For example, `makeTape` is a wrapper around an instruction to move the immediate value 2 into its register argument. It uses `postf` to add a function which may **alter the abstract machine state** after the effects of the `mov` instruction. The given function here ignores mosts of its inputs, it simply adds "Tape" tags to the two least significant bits of the register argument—which will at this point hold the value 2; this function will only affect the operation of the abstract machine during type checking; an assembled binary will only contain the `mov` instruction.

Another macro is `cmpSource` which features `lf`—for "limit function"—and is a type safe wrapper around `cmp`, a compare instruction. `cmpSource` may only be used with two registers holding `Source` values. It guarantees this by attaching two limit functions: `atPC` is given the current program counter and a part of the desired state at this point in the program, which are then combined into a formula expressing the logical consequence of "if execution arrives at this program counter, the following must hold." In this example, the two limit functions say "before the `cmp` instruction is executed, there has to be a `Source` in the two least significant bits of this register."

If any of these **constraints** are not met, type checking halts with an **error**. Errors take the form of trees, see Section 2.6.3. The error in Figure 1.5 can be read as follows: N is a stand-in for some numerical address in memory. The root on the left side means that `X0.0` does not hold a `Source` right before the `cmp` instruction is executed. Why not? Because neither of its children holds: It is not true that the PC is different from N. The lower child does not hold either. Why? Because both of its children would have to hold. $PC = N$ holds; it is not the problem here. It is the `Source` that is missing at `X0.0`.

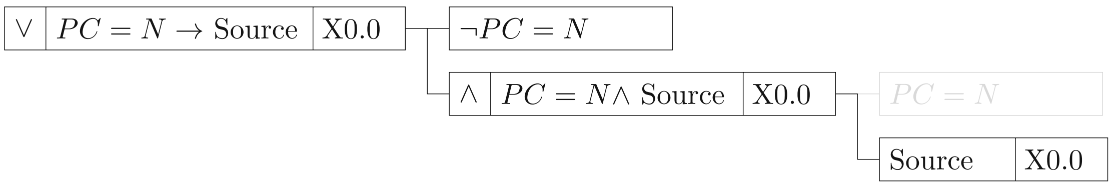


Figure 1.5: An error tree produced if `X0.0` does not hold a `Source` when the `cmp` instruction is executed ($PC = N$)

`lf` is used in other places as well, for example, at the label "playTape" where `X0.0` must hold a `Tape` value. As mentioned previously, this may not be what we usually expect from enumerations; it is often not possible to specify that not only does an argument have to be of the type of the enumeration but that only a specific value of that enumeration will be acceptable. This awards our program some additional safety. On the other hand, nothing prevents us from using plain `cmp` where `cmpSource` should be used; we would then forego the safety of comparing only `Source` to `Source`. Note how the check of the `Mute` value uses the regular `cmp` instruction. In contrast, we could also go one step further than `cmpSource` as shown, and tag the zero flag status bit as well as introduce a `Source`-specific `bCond` conditional jump instruction which only accepts the bit if it is appropriately tagged.

The instruction blocks at labels "playTape", "playRadio", "muted" and "panic" do not do anything useful in this example. They only serve as placeholders. If implemented correctly, the code should never jump to "panic," and indeed, when the program is considered as a whole, "panic" is never encountered. To make sure, we can use the statement `lf impossible`, where `impossible` is a constraint that can simply never be met. If it were encountered, type checking would halt with an error. In a similar manner, type checking tells us that the conditions given at the other labels are met.

Ideally, we would achieve the features and safety of the high-level version of a type, with some additional control. In the case of enumerations, this seems to be the case; while it is often possible to manually specify the exact mapping of, say, integer values to enumeration symbols, the width of the underlying integer type is usually up to the compiler's discretion; "odd" sizes, like an uneven number of bits, are particularly uncommon.

1.2 Method

The main research question considered is "How can a type system handle individual as well as grouped bits and verify that they are processed correctly?" This section gives an overview of the methodological approach to answer this question.

Having the goal of a type system, which does not stipulate a single and fixed notion of what features such as "enumerations" or "structs" entail, has to be factored into the methods used to **evaluate** such a system. One part of the evaluation is rather subjective, as in, what properties we expect a type system to have, when we think of a feature. The high-level languages we have seen previously precisely define their features, and we can clearly make out differences between those, such as when in Zig an (exhaustive) enumeration—the details are explained under "Non-exhaustive enum" in Zig's documentation [Zig21]—will only accept explicitly defined values while C will accept any and all integer values—in fact, in "The C Programming Language" [KR88], enumerations are first mentioned in a chapter primarily about constants. So while it is unclear how to test whether the language and type system at hand "have enumerations," a question like "is it possible to differentiate between two values which share the same bit pattern?" has a clear "yes" or "no" answer.

This leads to an approach which is made up of the following steps:

1. Define three sets of **questions**

   a) Regarding type system features—is it possible to implement a feature?
   b) Regarding ease of use—how much effort is necessary?
   c) Questions which are more open—answering them will be more subjective

2. Implement a **prototype** in two parts: an abstract model, which is applicable to a wide range of instruction sets, and one concrete implementation for a subset of the A64 instruction set
3. **Evaluate** the questions of the first step using the prototype of the second
   a) Regarding type system features—evaluate using the prototype
   b) Regarding ease of use—assess characteristics such as ease of use
   c) Questions which are more open—evaluate based on gained experience
4. Conclude and answer the **main research question**

The questions of the first step are defined next; step two, the implementation of the prototype, is the topic of Chapter 2; all questions are answered in Chapter 3 which represents steps three and four.

This prototype is evaluated through the following *polar* or *yes-no* questions:

- Is it possible to assign a name to a bit pattern, e.g. to use the name as a descriptive stand-in?
- Can the type system differentiate between values which use the exact same bit pattern?
- Is it possible to define a type as describing elements from either one or another type; more generally, is it possible to build a *sum type* from multiple other types?
- On a related note, is it possible to define *product types* like structs?
- Is it possible to specify a repeating pattern for types?
- Can values be "wrapped," i.e. converted to another type without changing the value itself?
- And can they be "unwrapped" again?
- Is it possible to restrict the types of function parameters and return values?

This first set of questions is about what is at all possible within the language. But how much work is it to actually write programs this way? To find out, consider these additional questions:

- Is it enough to specify a combination of name type and value just once?
- Is the creation of types by combining other types as straightforward as in high-level languages?
- Are "wrapping" and "unwrapping" as simple as in high-level languages?
- And are they as safe?

Answering these yes-no questions should also help in finding answers to the next set of questions, which are not as clear cut and leave more room for interpretation:

- How can the type of integer operations be described on the bit level?
- Can the type system safely support dense packing of bits into data words?
- Can complex bit level types be translated into a more human readable format; and how can we encode the real-world meaning of bits and their associated legal values in a type system?
- How can the type system succinctly describe the combined effects of individual instructions in a code block?

1.3 Structure of this Thesis

The rest of this thesis is structured as follows:

Chapter 2 describes the experimental prototype at the heart of this thesis. Section 2.1 gives an overview about the central ideas behind the type system. The logic formulas which "do the heavy lifting" are introduced and expanded in Sections 2.2 and 2.3. I talk about the implementation in Section 2.4 and the details pertaining to the specific target hardware that was chosen in Section 2.5. Section 2.6 explains how users can define and use types, how assembly instructions are structured and handled, and how to attach the behavior we want to them. Programs are written in a very simple domain specific language, which is briefly shown in Section 2.7.

After describing the prototype, it needs to be evaluated as described in Section 1.2. This evaluation, some other concluding thoughts and an outlook on possible future work are found in Chapter 3. It features methodical evaluation in Section 3.1 and then a more informal conclusion in Section 3.2. It is there, that the main research question is finally addressed.

Chapter 4 combines a discussion of related work—Section 4.1—and some ideas for future extensions of the presented approach—Section 4.2.

Lastly, Chapter 5 concludes.

CHAPTER 2

Underlying Framework & Experimental Prototype

While describing the central ideas on which the proposed type system is built, I shall also clarify in what way the proposed method should be considered a "type system." Through a short discussion of different ways to approach type systems we arrive at questions which are answered in subsequent sections.

2.1 Overview

In high-level programming languages, a type may be seen as something to be defined by its structure, its behavior, or both. In contemporary object-oriented programming languages, like Java—see Gosling et al. [Gos+20]—or C++—see for example, an introduction by Bjarne Stroustrup [Str13]—, we differentiate between built-in "primitive" types—chars, booleans, integers of various sizes—and user-defined types—tightly coupled to the concept of **classes**. These two categories represent two distinct approaches to both **structure** and **behavior**.

While the structure of a primitive type will be rather simple and have a direct match in assembly language—a register or a word in memory for an integer—a user-defined type could be a product of (primitive) types, a union, or a combination thereof. A primitive type's behavior is fixed and rarely holds surprises—integers support arithmetic operations; strings support concatenation—while the behavior of a user-defined type is limited only in the methods defined on it—or the messages it will accept. Due to the flexibility afforded by user-defined types, but also because of their relative complexity, we often focus our attention on them. Will this language allow us to construct the types we want? What operations shall we define for a newly designed type?

It is here where a first stark peculiarity emerges: When designing a type system for an existing assembly language, no user-defined operations await their definition—neither by the type system designer, nor by an eventual language user. All available machine instructions—incarnations of what I referred previously as "operations"—are not only fixed in the system architecture, each instruction already has a (general) type. [1]

It is our first task then, to capture the nature of **machine instructions and their types**. Harking back to my discussion of primitive versus user-defined types, I hold the view that on the machine instruction level this distinction dissolves. There, user-defined types do not have to be built from primitive types; it is only in the interpretation of individual and grouped bits that there is a sense of one type—or value—containing another. We may also think of the bit as the only primitive type, all other types being built from multiple bits—even a single byte without any additional meaning attached is not a prime unit. In a similar vein, there is no sense of a machine instruction being incompatible with its input; it is always clear what output bits result from what input. Every machine instruction can be said to transform one machine state into another; and in that sense it cannot fail. With machine instructions being in a sense unchangeable, there is still some leeway in their representation in a type system. I choose to "split up", so to speak, machine instructions along two axes:

First, I treat machine instructions as being made up of a few common operations, not as the indivisible instructions that they are on the level of the assembly language. For example, an addition instruction which features an optional bit shift of one operand, may be divided into separate shift and addition instructions. This way, all addition instructions all use the same **basic addition logic**, all instruction in which inputs are shifted use the same basic shift logic; in total, for the subset of the **Arm A64** instruction set[2] considered in this thesis, 38 machine instructions are replaced by about 11 to 18 **basic operations**—the latter number depending on the way one counts.

Second, just as registers and memory words consist of arrays of bits, operations on them can be considered a **succession of operations** which operate on smaller units—down to operations on individual bits. Bitwise operators demonstrate the simplest cases: in a bitwise and, the i^{th} result bit depends only on the two i^{th} input bits. The basic bit-level operation of arithmetic operations are the equivalent of full-adder digital circuits[3] and the like.

With such a model in mind, I pose these questions:

[1]One may not find type declarations in the manuals describing a machine instruction set or CPU architecture. I assert that such types are omitted because they are deemed self-evident—all inputs and outputs are bits, bytes or words—and without attaching any interpretation as to the meaning of these bits, there is nothing left to be said.

[2]A description of the Arm A64 instruction set can be found in the Arm Instruction Set Reference Guide [Arm18]; for additional information, for example, on the exact workings of individual instructions or on how to convert instructions to and from binary opcodes, see the Arm Architecture Reference Manual [Arm20].

[3]*Ripple-carry addition*, along with half and full-adders is explained in Hennessy and Patterson [HP12].

What shall be considered the **fundamental operations** on the bits in a computer's registers and memory? How can high-level conceptions of what it means for data to have a type, be encoded in entities which are to be understood in terms of these fundamental bit operations?

That is to say, I ask, for example, what can it mean for a bit to be shifted to the left; and, by extension, what does it then mean for an array of bits; and, in anticipation of an implementation of types, how can we guarantee that only those operations we **deem meaningful** are allowed to be performed, if we can only attach some sort of **type mechanism** to machine instructions which cannot be altered?

An approach different from that of OOP is found in functional languages, like Haskell or languages from the ML[4] family. In Haskell, a type is free from any behavior when it is first defined. Using dedicated function definitions, it is then possible to newly create operations specifically for this new type. Via Haskell's type class mechanism—sometimes referred to simply as "classes," see Jones [Jon03]— it is also possible to, in a way, "promote" a type into a previously defined generic class; in other words, we may declare that—and how—a type shall henceforth be subjected to some set of operations.

Alas, on the bit-level, no values different from 0 and 1 can be created and all bits have the same type. I nonetheless aim to provide ways to differentiate between "different types of" bits and a similar level of strictness. At any given point during the execution of a program, for each bit in registers and in memory, we know that it is 0, that it is 1, or that it is one or the other. What can we derive from this limited type of information alone; and in what other ways can we differentiate between these bits? This method of approach is also central to the proposed type system design.

Not only data takes the form of mere arrays of bits in assembly languages; the instructions which make up the program are accessible in the same way. Not only does this blur the distinction between data and instructions, it allows software to be **self-modifying**, which is a possibility many type systems do not have to take into account. In combination with the mechanism of the **program counter**—which is not commonly exposed in high-level languages—I deem it necessary to closely model the hardwares's program counter, instruction opcodes and memory locations. As a consequence, the proposed system must be able to manage abstract presentations of one or multiple machine states—encompassing registers, memory, and thus all of a program's machine instructions in **binary form**—and to perform transformations from one such state to multiple successor states. In this way, we explore the space of all possible states which may be encountered as a consequence of running a given program. More directly, we want to derive from only a few possible starting states—e.g. a state describing the exact moment a program has been loaded and the program counter is pointing at a well defined first instruction—those abstract states describing all states encountered on actual hardware.

[4]Notably Standard ML—see Milner et al. [Mil+97]—and OCaml—see Doligez et al. [Dol+20]—or the more recent multi-paradigm language F#—documented by the F# Software Foundation's website [F#21].

Modeling these low-level processes brings with it some inconveniences: When we wish to allow only arguments of a specific type to be passed to a function, we usually state this in a function's type declaration. In our abstract state, there is neither a function definition nor a type declaration. Assembly code does not "call a function" but merely sets the program counter to point at the desired memory address—consider, for example, the `BranchTo` function in the *Arm Architecture Reference Manual* [Arm20]. The instruction is present only as a bit pattern corresponding to the instruction's opcode. No typing information can be extracted from the opcode alone, so there needs to be some other way to supply typing information for any specific location in a program.

My solution is to represent abstract machine states as a propositional formula—the details are found in Sections 2.2 and 2.3. A formula corresponding to some known starting state is repeatedly expanded until no new information is added. At the same time, another set of formulas representing an "upper limit" is checked against contradictions with any new formulas, i.e. all discovered states must conform to the **constraints** encoded therein. The corresponding event to finding two incompatible types for a value, in my system is discovering a state that is excluded via such an "upper limit" formula.

For example, in order to express that an argument to a function must have some type, we need a formula to the effect that, if the program counter points to the first instruction of the function body, then the location holding the argument in question must have all desired properties that describe the desired type. Section 2.6.3 describes the workings of these constraints and how errors are handled.

The type checking process is then the **interpretation of a program** in an abstract machine—close enough to the semantics of the actual hardware to remain relevant, but diverging enough from it so as not to become a full simulation—with the addition of typing information and its transformation in order to check some desired properties. How this is implemented is the topic of Section 2.4.

2.2 Logic

Traditional typing environments are not part of the proposed method. Instead, all type information is encoded in logic formulas. Let an expression such as $1_{X0.7}$ stand for "the 7th bit of register `X0` has value 1"; and $0_{W3.0}$ for "the 0th bit of register `W3` has value 0." The absence of statements about a bit signify the third possibility, that the bit may be 0 *or* 1. We may freely combine these propositions with $\wedge$, $\vee$ and $\neg$; **disjunctive normal forms** facilitate a natural reading as a set of possible states like $(1_{X0.0} \wedge 1_{X30.0}) \vee (0_{X0.0} \wedge 0_{X30.0})$—either the 0th bits of registers `X0` and `X30` are both 1 or they are both 0. A clause such as $(1_{X0.0} \wedge 1_{X30.0})$ does not say anything about the majority of bits—even when only registers are considered—so it does represent a whole **set of possible machine states** on its own; I still feel it closely mirrors a natural way of thinking about the state of a computation—possibly even more so than an exhaustive enumeration. Another quirk is the presence of two propositions for a bit—having value 0 versus value 1—because they are contradicting each other and can never both be part of a valid state, so we may see 0_L, for any location L, as $\neg 1_L$ in disguise.

Entailment puts abstract machine states into a relation signifying that one state is more general than another. We may read $S \models S'$ as "S is more specific than S'" and "S' is less specific than S", or "S describes fewer machine states than S' and "S' describes more states than S." If S represents the calculated set of all states which may be reached from some starting state, and S' encodes all constraints put on the code by the user, then if and only if $S \models S'$, all constraints are satisfied.

In my implementation—even though one may use any structure to encode the proposed type of statement, such as a tree made of $\wedge$, $\vee$, $\neg$ nodes and leafs—I use a flat two-level structure for disjunctive normal forms; a *disjunction* being a set of *conjunctions*; conjunctions being sets of *literals*; apart from the connection to what I consider a natural way of envisioning machine states, this lends itself well to an implementation that is conceptually simple and reasonably efficient at the same time.

The expressive power of the approach described so far is limited. The method is able to show some useful properties—and some examples are given shortly—but a lot more can be done with an additional group of propositions, described in the next section.

Without complete information about the value of all input bits, one may still derive unambiguous information about some output bits. For example, an addition instruction on 8-bit operands is sometimes known not to produce an overflow, no matter the actual inputs, as long as some key bit values are known. Consider two operands which are 8 bits wide, but can be said to "only use" the three least-significant bits—shown here as the three rightmost. All other bits are known to be zero. Then, a carry might occur going out of the third bit position, into the fourth. There the result is unknown, but because it is an addition of $0 + 0$ with an unknown carry, the fourth carry bit must be 0.

$$\begin{array}{r} 00000??? \\ +\ 00000??? \\ \hline 0000???? \end{array}$$

In general, we may postulate a rule like "The addition of two n bit wide operands produces an $n + 1$ bit wide result without overflow." Similarly, multiplying two three bit wide operands gives a six bit wide result, in a manner of speaking.

$$\begin{array}{r} 00000??? \\ \times\ 00000??? \\ \hline 00?????? \end{array}$$

The simple underlying principle can be seen as a sort of "compressed" form of a multitude of obvious rules. I will return to some not so straightforward examples in Section 2.6.1.

2.3 Type Tags

Let me elaborate on the shortcomings of the approach seen in the previous section.

Any bit pattern can be expressed as a propositional formula, and so it is of course possible to limit the valid states—at the entry point of a code block, for example—to only those where a register holds one of a set of valid values—the low-level representation of some enumeration. All other—invalid—values can be excluded, and each value may be handled individually, but what cannot be accomplished so far, is to differentiate between values of different enumerations that just happen to be mapped to the same bit pattern. We are missing a facility to attach additional typing information to data.

For this reason, I introduce a user-defined set of "type tags"; it suffices that these tags are distinguishable from one another; for my initial implementation I use strings and a few special tags—a family of `Nat` tags indexed by a positive integer, or a `Pointer` wrapper around any other type tag—proved sufficient. A type tag is associated with a width in bits, and a bit can be tagged by zero or more tuples containing a type tag together with an index that gives the position the bit takes as part of a multi-bit value. For example, the five bits of a `Nat 5` tagged value have indices 0 through 4, respectively.

Tagging individual bits circumnavigates problems such as the question of how to handle partial values—which may result from a shift operation—or how to handle endianness when storing to and loading from memory. With all bits being able to be processed and "moved around" individually, handling these issues becomes part of program design; encodings are explicit and transparent. The obvious downside is that the assembler user has to be involved in this matter as well.

It is up to the programmer how type tags are handled. The decision to "split up" all machine instructions into a set of common bit-level operations—which is deliberately kept small—is rooted here. The assembler user defines one or more type tags, along with a few pure functions handling type tag propagation, or rather, conservation, across machine instructions. For the fixed set of basic operations—such as addition of two operands and a carry bit, inverting a single bit, logical operations on two input bits, bit moving operations like logical and arithmetic shifts, or loading and storing bits from and to memory—the user may define functions and what type tags they are concerned with; the function then receives appropriate operation-specific inputs—source and target *bit addresses* for arithmetic operations, bit addresses in registers and memory for load/store operations—along with a state input argument—common over all of these functions—and may return a set of type tags to be attached to the output bit address.

I present one example here and follow up with a more detailed description of the actual instructions available for the Arm A64 implementation in Section 2.6. A bitwise `and` instruction on two input registers and one output registers shall trigger, for each index in the range of the register widths, a function which receives the following arguments, which are custom to the `and` base operation: the address of the i^{th} target register bit, the addresses of the two i^{th} source register bits; it will also receive a full state in the form of a conjunction of literals—i.e. an object from which it may extract information such as whether a bit is 0, 1, or has unknown value, and what type tags are attached to a bit. One may, for example, define for the equivalent of some specific high-level structure type a "Struct" type tag, a "StructMember" tag, and a "Mask" tag for bitmasks used in extracting specific bits that make up the "StructMember" part from the whole structure. An appropriate function might then extract type tag information for both source bits and produce a "StructMember" tag at index j for each j such that one source bit has a "Struct" tag with index j while the other has a "Mask" tag with index j. This way, the only way to extract the struct's desired member value is by a bitwise `or` instruction on registers holding properly tagged "Struct" and "Mask" values. The requirement for matching indices implements safeguarding against incorrectly aligned bit patterns. As shown in Section 2.6 it is up to the user to decide how to handle details like this; one must strike a suitable balance between flexibility, safety, and ease of implementation, among other things.

2.4 Implementation

An implementation for a specific instruction set separates into a part that is common among all instruction sets and a customized part. I first focus on what is instruction set-independent.

My first prototype implementation with a state structure that is a straightforward translation from logic expressions to a simple, flexible data structure proved entirely impracticable. Checking for the presence of a single proposition in an unordered set or list—coupled with the fact that this is such a central part of the algorithm—makes it almost unbearable to run even the smallest of examples.

Concessions to the nature of the hardware must be made. Grouping bits the way they are grouped in hardware leads to the following approach: Individual bits still "live" in orderly groups inside of registers and memory. I cannot follow this structure precisely because a single bit is not enough to store the information I assign to one bit address. I add to my three values of 0, 1 and "unknown", a fourth value that stands for a bit that is both 0 and 1. This fourth state is the product of two circumstances; on the one hand, there is the "accident" that we need two bits to store three states—the fourth state "comes for free;" on the other hand, a way to encode invalid states is necessary. I want to be able to detect—and remove—states which cannot be part of an actual machine's behavior because of contradictions. It adds a little comfort to know where a state "went wrong."

A register of width w in an abstract state is then represented by two w-width words. One word encodes which bits are known to be 0 and the other encodes those which are known to be 1. The i^{th} bit is 1 if the i^{th} bit of this register is known to be 0 or 1, respectively. If both bits are 0 the bit value is unknown; if both are 1 this signifies a contradiction. In the latter case any conjunction that includes this register shall be considered a contradiction; it does not have any models and we may refer to it as a "bottom state."

If we further assign a fixed order to all registers, we can then easily combine all registers into an array of words—of the same size as registers in the target machine—for constant time access to any bit's information.

It remains to be decided how to encode type tags. Mapping each bit address to a set of tags seems adequate; in my examples, type tags are by far not as numerous as other bit information and tend to be more short lived and independent of each other.

Memory is largely handled in the same manner. The logical layout of memory can be directly applied to its abstract representation of bits. While each conjunction includes information about the whole set of registers—even when it is mostly filled with "unknown" bits and little more—memory is broken up into more manageably sized chunks; only those chunks which contain "non-unknown" bits are allocated.

So far, I referred to "propositional" logic and formulas, but one may feel that this is not accurate. Is my tagging mechanism not built on a predicate? It would certainly be suitable to define a predicate *tagged* as a function mapping tuples of bit addresses and type tags to truth values; I take up the position that any predicate on finite input sets is not fundamentally different from a "mere set of propositions"; the number of bit addresses pointing to registers and memory is surely finite, and with the practical limitation that exists on the size of tags—even if they are represented by "arbitrary" character strings—I assert that, under this interpretation, the set of type tags is finite as well. Because no quantifiers are used, we are not concerned with the added complexity of first-order or predicate logic.

A conjunction of information about registers and bytes in memory builds an abstract state; in the most common case, in which values of some bits are left unspecified, a single conjunction represents multiple states which we can consider to be "one possible case" among all the states of a machine. The bulk of the type algorithm is carried out by a transition function from one such conjunction to a disjunction, i.e. a set of conjunctions. This follows my intuition of handling "each case" ony by one.

The **main interpretation function**—which is independent of machine and instruction set—is a simple loop: Its inputs are a transition function—which constitutes the instruction set-dependent part—as well as an abstract starting state and another abstract state which shall serve as an "upper limit", i.e. all constraints on the full abstract state of a program shall be encoded therein. In other words, if we say that the entirety of all reachable machine states is its "type", then the "upper limit" state is a **supertype** of the actual program type that is being calculated. As long as the given transition function produces new states which are not already described by the starting state—which is the case if any newly derived successor states are not entailed by the starting state—these new states are incorporated into the starting state. During each iteration, newly derived states are checked against the "upper limit" for incompatibility, which would signal an error in the program's type as specified by the user; this is again implemented by an entailment function. In theory this process must halt eventually, as the number of possible states is of course limited, given that the size of the simulated machine is finite.

In practice, one may need to specify a type that is general enough to avoid arduous exploration of, for example, each and every possible iteration of a loop.

The next section describes the chosen target instruction set; the ones after that go into more detail about some aspects of the instruction set-dependent part of my implementation.

2.5 Target Machine

As a realistic target, I choose a subset of the A64 instruction set, found in a range of 64-bit Arm cores. This CPU type features 30 general purpose registers which are 64 bits wide. Instructions are available in 32 and 64 bit variants, where either full 64 bit wide registers, 32-bit half registers or a combination are used. In addition, there are some special purpose registers such as a stack pointer, `XZR` and `WZR` registers which always read 0, and registers for status flags. A description of which registers are available in the different execution modes, and how they can be used in instructions, can be found in the *Arm Instruction Set Reference Guide* [Arm18].

The abstract machine implementation is very similar to a **simulator** of the actual hardware with only a few derivations: While in actual hardware each bit will have some fixed—even if unknown—value of either 0 or 1, my abstract version replaces regular bits with a four-state generalization, as detailed in Section 2.4. As for the `XZR` and `WZR` registers—where we can consider `WZR` to refer to the 32 least significant bits of `XZR`—a straightforward transfer is not desirable. These registers are used whenever an instruction calls for a register but an all-zero pseudo-register is needed instead. The `mul` instruction—the analog of $d = n \times m$—is realized as an alias to the more powerful `madd`—$d = n \times m + a$—instruction with a replaced by `XZR` or `WZR`, respectively. Their second common usage is as a target register that discards the result value. This second usage type must be taken into consideration when implementing some arithmetic operations as the result of multiple, more basic, operations: the result value (which may be destined to be discarded) in a previous such basic operation may still be needed as input of another operation, in order to, say, determine which status flags need to be set or cleared as the result of a machine instruction such as `adds` or `cmp`. To achieve the desired behavior, it was necessary to reserve space to store values written into `XZR` and `WZR`, even though to a program on Arm hardware, they are never directly available. Care must be taken to make the proper choice from a set of two access functions for these registers: one that behaves the way the actual hardware does and always reads all zeros, and one that reads the current value, which, I must stress, is not part of an actual machine state—only to be used under certain circumstances and while implementing machine instructions as concatenations of basic instructions.

2.6 User-Defined Types

How does the user of a programming language interact with its type system?

In a functional sense, one may say there are two fundamental types of interaction: The *definition*, or *creation* of types, and the (optional) *assignment* of types to values. Along with the definition of values for constants, variables and functions—or, in the latter case, their behavior—a typical program in a purely functional language consist of declarations of the types which are assigned to the aforementioned values, as well as declarations describing the types themselves.

From the perspective of a procedural language, one might be more concerned with the *state* of a process as it changes over time. As the contents of registers and memory change, so do the types of the values in it, or—if one considers a model where each value has a single fixed type—so do the values held in the system's memory, and with that, the type of value that is stored there. In the present case of an assembly language, the user must then have the ability to specify the structure of types, to assign a type to a register, a piece of memory—to part of a machine state—and to specify how the types which are part of the abstract machine state change—which is to say, to specify what changes a machine instruction shall cause in the realm of types. While the effect of machine instructions on bit values is fixed and must be replicated from its real-world counterpart, typing information is only present in the simulation; we might say it is "purely artificial" and may thus be bent to our will.

In order to model types, the proposed method provides the following mechanisms which I will describe, one after the other:

- The definition of types by creating and combining *type tag*s
- A way to attach type information to parts of an abstract machine state
- "hooks" in order to integrate type tag transformation and to influence some parts of the abstract machine interpretation
- primitives and combinators to describe type constraints

2.6.1 Type Definitions

There are two fundamentally different categories of types, which roughly correspond to structural and nominal typing:

First, there are four base bit types: the bit known to be 0, the bit known to be 1, the unknown bit, and the bit which is said to be both 0 and 1, the "contradiction bit." These four—and their combinations—can be considered structural types; they stand for sets of bit patterns found in the state of both abstract machine states and in the registers and memory of actual hardware.

Second, there are type tags; they do not exist in actual hardware, they only appear in the simulated machine and are manipulated in parallel to the mentioned bit types and values. While, in the proposed method, type tags are crucial for the implementation of nominal types, the distinction between structural and nominal typing can become somewhat fuzzy. Attaching type tags does not replace type information that is expressed using the four bit types. Type tags open up possibilities for additional behavior to be implemented. In order to illustrate this point by means of an example, let me introduce a common type along with its representation.

A type tag is defined by a unique name and its width in bits. The *type name* can be a simple and universal identifier like a string, or a specialized name, for example a family of natural numbers indexed by bit width; I adopt the abbreviation `Nat` as used in Idris—see Brady [Bra17]—or Coq—an interactive theorem prover, see for example its online reference manual [Coq21]. The structure of a type name is not relevant, the type algorithm needs only to be able to differentiate between them. A `Nat 64` is then the analog of the GNU C compiler's `uint64_t`, a `Nat 8` that of an unsigned byte, but we may also use `Nat`s of other, non-standard, sizes. To express that "there is an unsigned 64-bit integer in register `X0`" one attaches 64 `Nat 64` tags to the 64 bits in register `X0`. Such a type tag always consists of a type name—in this example, `Nat 64`—as well as an "index into the type", so to speak. In this way, each of the 64 bits has a different tag associated with it, specifying which type it is part of, along with its position as part of that type.

Returning to the discussion of nominal and structural typing, let us explore options for wrapping an existing type in a named wrapper, similar to Haskell's `newtype` declaration; the role of `newtype` is "only to change the type of a value," as Simon Peyton Jones [Jon03] put it.

A straightforward way to implement an `ID` type is to simply use an integer type of appropriate size. One may simply choose a "naked" standard integer such as `Nat 32`, but as a safeguard against errors, such as accidentally adding two `ID`s, it shall be implemented as a "wrapper type", so as to remove the ability to perform arithmetic operations on values of type `ID`. Because there is no limit on the number of tags attached to a bit—and thus no limit on the number of types assigned to a set of bits—`ID` tags may either be added to the existing `Nat 32` tags, or replace them. The effect of leaving `Nat 32` tags attached is that—assuming appropriate definitions, which will be explored shortly—the value in question may still be treated as a `Nat 32`; it is in fact still considered to be of type `Nat 32`; the addition of `ID` tags merely made the value into a member of the `ID` type in addition to any types attached previously. The other option—replacing `Nat 32` tags—results in a value that is only of type `ID`, allowing only whatever behavior being a member of `ID` entails.

But what does it mean to "allow" behavior when in an assembly language, any instruction may be performed at any time, and when any inputs and outputs of said instruction are not checked for compatible types—as there are, in fact, no types—but are "in the eyes of the machine" always just bits? Because there is no way to prevent the actual hardware from processing data in cases where we consider the types to be incompatible, a weaker type of "penalty" is adopted: to process compatible types means to preserve—or in some cases, newly produce—the “right” type tags; to process incompatible types means to lose them.

For `ID`s which are `Nat 32`s at the same time, this means that performing arithmetic operations on values of this type shall preserve the `Nat 32` type tags, while at the same time all `ID` tags will be dropped, resulting in value that is still a `Nat 32` but not an `ID` anymore. It is these missing `ID` tags which can the be used to trigger errors and abort the type checking process. Without proper constraints in place, the loss of type tags would go undetected. How this is accomplished is discussed in Section 2.6.3.

The implementation of types takes the form of functions which take an argument bit address like `X0.16`—describing the position of typed bits by giving its least significant, or right-most bit—and return a conjunction or disjunction of states. A type tag-based function for a `Nat` type would simply attach an array of `Nat` tags to some bits. For example, to produce the appropriate state for a `Nat 3`—representing an integer in the range of 0 to 7—at bit address `X0.16`–the 17th bit from the right of register `X0`—`Nat 3` tags with indices 0, 1 and 2, would be attached to the 17th, 18th and 19th bits of register `X0`, respectively.

These functions also produce the other category of type, those which assign 0 or 1 to individual bits, as well as larger bit patterns. A single bit pattern can represent a singleton type and its only value. The corresponding function sets and clears the appropriate bits relative to the argument bit address.

I define a few combinators to build more complex types from these two kinds of simple type functions:

Shift functions transform a type function by first transforming the given bit address.

`struct` combines multiple type functions by producing states which contain all of their type tags and bit patterns—this corresponds to a *logical and* operation. Together with shifts, `struct` allows the construction of types which resemble `struct` types in C, hence the name. A type to represent a two-dimensional vector is shown in Figure 2.2.

Enumerations can be built from singleton types and `struct`s counterpart `enum`—a *logical or* operation. A `shiftType` may be one of `lsl`, `lsr`, `asr` or `ror`. Its definition uses `enum` as shown in Figure 2.1. `struct` and `enum` may be combined to build *tagged unions*, for example `tree` and `tree'`, which can be considered identical.

The “body” of a `treeNode` type tree element in Figure 2.1 is made from two `Nat 31`s; they are placed right next to each other in memory, so the second `Nat 31` is shifted 31 bits to the left, just past the end of the first one. The bit width of a `Nat 31` is 31 but in general the bit width of a type tag is very much arbitrary. It may also contain “holes” such as the `rgb'` color type in Figure 2.3.

This figure also shows a larger bit pattern, `invert`, which specifies a value that is explicitly not included in the possible values for type `rgb`. This somewhat peculiar type—which may include either a color specified by its 16-bit red, green and blue components, or a special effect color value which shall invert the RGB values of the colors already present on some drawing canvas—is meant to demonstrate the unusual encodings which may be formed using the presented low-level type mechanism: An enumeration type to discern the two fundamental types of `color` is not needed, as the rest of the subtypes are non-overlapping; another interpretation could be that the discriminating bits are merely "spread out" and put in unusual places, and that one could just as well have put all three `Nat 16`s right next to each other with an additional `colorType` enumeration on the side.

```
enum :: [BitAddr -> State] -> BitAddr -> State
enum fs addr = foldl1 (||) $ map (($) addr) fs

lsl, lsr, asr, ror :: BitAddr -> State

shiftType :: BitAddr -> State
shiftType = enum [ lsl, lsr, asr, ror ]

treeLeaf, treeNode, treeType :: BitAddr -> State
treeLeaf addr = {- assign 0 to addr -}
treeNode addr = {- assign 1 to addr -}
treeType = enum [ treeLeaf, treeNode ]

ptr = nat 31 -- pointer or index into a buffer of tree nodes and leafs
val = nat 63 -- 63 bit integer values stored inside the tree

tree :: BitAddr -> State
tree = enum [ struct [ treeLeaf
                     , shiftLeft 1 val
                     ]
            , struct [ treeNode
                     , shiftLeft  1 ptr
                     , shiftLeft 32 ptr
            ]

tree' :: BitAddr -> State
tree' = struct [ treeType
               , shiftLeft 1 $ enum [ val
                                    , struct [ ptr
                                             , shiftLeft 31 ptr
                                             ]
                                    ]
               ]
```

Figure 2.1: *tree* and *tree'* combine *struct* and *enum* differently, but describe the same type

```
nat :: Int -> BitAddr -> State
nat n addr state = {- ... -}

shiftLeft :: Int -> (BitAddr -> State) -> (BitAddr -> State)
shiftLeft n t = {- transform t to describe the same type
                   but shifted n bits to the left -}

struct :: [BitAddr -> State] -> BitAddr -> State
struct fs addr = foldl1 (&&) $ map (($) addr) fs

vec2d :: BitAddr -> State
vec2d = struct [ nat 16
               , shiftLeft 16 $ nat 16
               ]
```

Figure 2.2: Two `Nat 16`s are combined to form a 32-bit wide `vec2d` type

```
-- contains holes holding arbitrary bits
rgb' :: BitAddr -> State
rgb' = struct [ nat 16                  -- red
              , shiftLeft 17 $ nat 16 -- green
              , shiftLeft 34 $ nat 16 -- blue
              ]

-- contains holes which must always be 0
rgb :: BitAddr -> State
rgb = struct [ nat 16                  -- red
             , shiftLeft 16 zero
             , shiftLeft 17 $ nat 16 -- green
             , shiftLeft 33 zero
             , shiftLeft 34 $ nat 16 -- blue
             , shiftLeft 50 zero
             ]

invert :: BitAddr -> State
invert addr = {- assign all zeros to 51 bits at addr, with the
                 exception of ones at indices 16, 33, and 50 -}

color :: BitAddr -> State
color = enum [ rgb, invert ]
```

Figure 2.3: A densely packed type that fits into a single register

But there is one advantage to the encoding in Figure 2.3: Let us consider the outcome of an arithmetic addition of two `Nat 16`s which are stored in 64-bit registers. In an addition, the i^{th} result bit depends on the two i^{th} input bits, as well as the carry bit going into the i^{th} position—which is the carry bit going out of the $(i-1)^{\text{th}}$ bit. By extension, the i^{th} output bit depends on bits with index j where $0 < j \leq i$, at least indirectly. Whenever an addition takes place "inside of" a register that is larger than the operands we are interested in, it is safe to disregard any bits to the left of—that is, on the more significant side of—operands. For a result that is also not compromised by information from bits to the right "leaking in", we need to take into account the carry bit going into the rightmost bit of operands. Examples with a layout similar to that of type `rgb` shall demonstrate this point; 3 bit wide components should be adequate. Let R stand for a bit that is part of the red component of an operand or result, G for green, and B for blue bits. Superscript indices stand for a bit being part of the first or second operand; subscript indices stand for the position inside a color component; G^0_2 is then the 3rd bit in the green component of the first operand. Further, if ? is a bit of unknown value—subscripts on ? serve to tell them apart—and the superscript r stands for bits of the desired result—the same result as if three individual additions had their results combined into the same format as that of the operands—then line D depicts the desired format, but the actual result—or, what can definitely be said about it—is found in line A.

$$\begin{array}{rr} & ?^0_{11}B^0_2B^0_1B^0_0\,?^0_7G^0_2G^0_1G^0_0\,?^0_3R^0_2R^0_1R^0_0 \\ + & ?^1_{11}B^1_2B^1_1B^1_0\,?^1_7G^1_2G^1_1G^1_0\,?^1_3R^1_2R^1_1R^1_0 \\ \hline D = & ?^r_{11}B^r_2B^r_1B^r_0\,?^r_7G^r_2G^r_1G^r_0\,?^r_3R^r_2R^r_1R^r_0 \\ A = & ?^r_{11}?^r_{10}\,?^r_9\,?^r_8?^r_7\,?^r_6\,?^r_5\,?^r_4?^r_3R^r_2R^r_1R^r_0 \end{array}$$

Because neither $?^0_3$ nor $?^1_3$ are known, we cannot know what value is carried from position 3 to position 4. Adding G^0_0, G^1_0 and an unknown carry bit results in an unknown bit $?^r_4$ and an unknown carry into position 5, and so forth.

The solution is to "fill the gaps" with zeros, as in the `rgb` type.

$$\begin{array}{rr} & 0B^0_2B^0_1B^0_0\;\;0G^0_2G^0_1G^0_0\;\;0R^0_2R^0_1R^0_0 \\ + & 0B^1_2B^1_1B^1_0\;\;0G^1_2G^1_1G^1_0\;\;0R^1_2R^1_1R^1_0 \\ \hline D = & ?^r_{11}B^r_2B^r_1B^r_0\,?^r_7G^r_2G^r_1G^r_0\,?^r_3R^r_2R^r_1R^r_0 \\ A = & ?^r_{11}B^r_2B^r_1B^r_0\,?^r_7G^r_2G^r_1G^r_0\,?^r_3R^r_2R^r_1R^r_0 \end{array}$$

The bit value of $?^r_3$ is still unknown; it is the result of adding 0 to 0 with an unknown carry—we do not know whether the carry from position 2 to 3 is 0 or 1, we only know that it is the carry produced from adding two 3-bit unsigned integers—but we do know the value of the carry from position 3 to 4: it must always be 0. Now the addition of the green color components is undisturbed by any unknown carries; the same logic that applied to $?^r_3$ and the green component applies to $?^r_7$ and the blue component, respectively.

The gaps between the three `Nat 3`s prevent the bits of neighboring components from "getting entangled with each other" or corrupting the result of additions; in a scenario where the full precision of 64-bit registers is not needed, it is possible to pack multiple values into a single register, in order to, for example, save space and open up the possibility of performing some calculations using fewer machine instructions; specialized types can help to catch subtle errors, such as forgetting to clear the "gap bits" in preparation for an addition.

2.6.2 Type Behavior

What is here called behavior is the way in which the state and types of a process, through instructions, change over time. One part is fixed by the instruction set: even though actual hardware does not deal in the four-valued bits described in Section 2.4, it is a straightforward extension of bits which may only be 0 or 1; if a bit is unknown, we have to consider both cases; if all possible inputs from a (partially) unknown state are taken into consideration, then a bit is either 0 or 1 in all cases, or it remains a bit of unknown value. These bits, which might have an unknown value, are represented as the `MBit` type. The language user has no leeway in this regard; there are no choices to be made which would not be an erroneous deviation.

The remaining part—that of type tags—has an entirely different setting. The default behavior of a type tag during execution is to "disappear", that is to say, without specifying otherwise, no instruction will preserve or produce any given tag, so any bits in registers or in memory, which are altered by an instruction, will end up losing all of their attached tags.

Contemporary instruction sets—such as the Arm A64—have hundreds of different instructions, making it impractical to have the user define the type tag behavior of each one individually. Even the limited number of operations performed on a typical type—such as basic arithmetic operations on natural numbers; shifting, and a few bitwise operations to extract fields from structures—one would still have to manage each variation of an operation. There are variations utilizing shifted registers, extended registers, and immediate values, or a slew of different addressing modes, for many common instructions. I opt for an approach that can be described as breaking each machine instruction into smaller, reusable instructions—a load instruction which takes a target register, a base address and an optionally shifted offset, is replaced by three instructions: a shift, an addition, and a load from a then fixed address—an approach which might be characterized as an "extreme case of a reduced instruction set computer." This helps to cut down the number of instructions, on the one hand, by introducing widely applicable base instructions—such as a single instruction for loading from memory to register—and on the other hand, by frequently reducing the introduction of a new machine instruction to picking and gluing together of already available ones. In some cases, it is necessary to use temporary registers which are not accessible—or non-existent, the details do not concern us here—on actual hardware, but this just introduces some minor overhead.

Four groups of instructions are relevant to this discussion:

1. Instructions on individual bits
2. Instructions on individual `MBits`
3. Instructions on the type tags of individual bits
4. Instructions combining formerly mentioned ones to form instructions on whole registers—and sometimes memory cells

Instructions on Bits

First, there are instructions which take boolean inputs and outputs. Most of these correspond to well known concepts like an arithmetic “full-adder” circuit, logical operations such as `and` and `or`, or the `not` operator to invert bit values. They are not part of the implementation, but the other three groups all are.

Second, there is the most direct equivalent of the first group. They process `MBits` instead of “regular” booleans or bits. The functions they implement are extensions of functions in the first group. For example, in addition to giving `True` ∧ `False` = `False`, this group’s `and` instruction will give `True`∧`Unknown` = `Unknown` and `False`∧`Unknown` = `False`. This group implements the calculation of `MBit` values on the bit level.

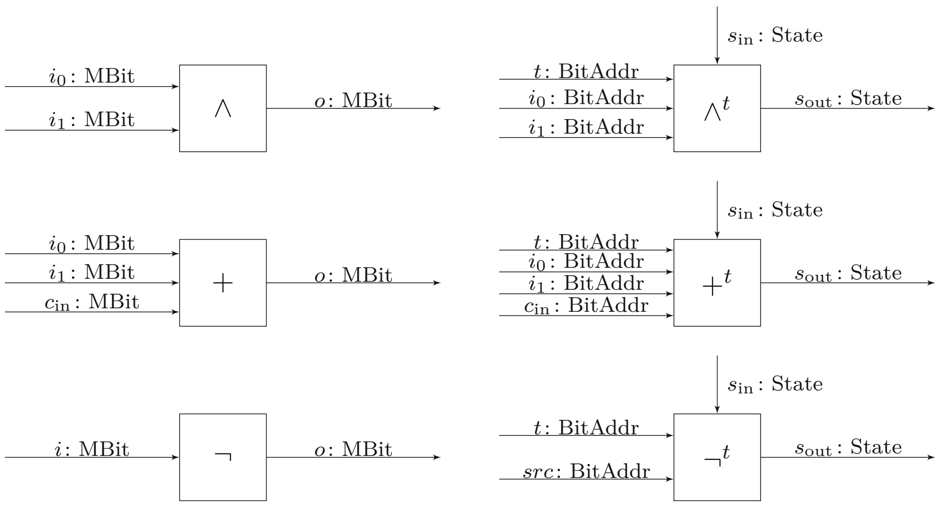


Figure 2.4: Base instructions on `MBits` on the left and their associated type tag instructions on the right, denoted by a t superscript

The third group makes up the remaining part of instructions, the one handling type tags. Alongside the inputs one would expect and sees in the previous group—for example, two inputs `MBits` as well as a carry `MBit` for an arithmetic instruction—instructions in this group all take an additional input `State` argument and produce a single output—again of type `State`. As these instructions are where a user specifies type tag behaviour, passing a complete `State` value to them allows any relevant information to be extracted. In fact, the inputs are not directly specified by their `MBit` values; instead, their bit addresses are given; it is then up to the user to extract the `MBit` and/or type tag information that is needed. Figure 2.4 shows the difference between associated instructions from group two and three. The three instructions on the left have fixed behavior: $\wedge$ produces output $o = i_0 \wedge i_1$; $+$ calculates the sum of three inputs, similar to a full-adder circuit (with the exception that $+$ does not give any carry output; I will come back to this shortly); lastly, $\neg$ inverts regular boolean values, and $\neg$`Unknown` = `Unknown`. Instructions on the right are to be specified by the user, they are free in which, if any, type tags they attach to target bit address t. What differentiates them is only their intended purpose and what information is passed into them. $\wedge^t$ is expected to decide which tags to attach to t according to the tags found on i_0 and i_1; in the case of $+^t$ one will also be interested in the tags on the carry bit c_{in}; and $\neg^t$ may decide which type tags to use from the destination t, the source *src* or both.

Other instructions which have the same structure are available; apart from addition $(+)$ and subtraction $(-)$, which both produce only a single output, but omit a carry output, there are also $+_c$ and $-_b$, instructions which give the carry and borrow outputs of addition and subtraction, respectively. Combined, $+$ and $+_c$ behave as a traditional full-adder would; compare this to the standard "two-in-one" full-adder of Hennessy and Patterson [HP12]. Figures 2.5 and 2.6 show two ways to implement 4-bit addition. The later arrangement might seem factitious and it does not in fact serve a concrete purpose for the calculation of booleans or `MBits`. The plain reason behind it is that this way, the users interface takes the form of four ($+$ and $+_c$ for addition, $-$ and $-_b$ for subtraction) functions of the same form, rather than two which have to handle both sum and carry outputs at the same— when one may often choose to ignore, say, the carry output completely. This allows all type tag functions to be handled in a uniform way as they all produce a single list of `Tags` to be attached to a single bit address.

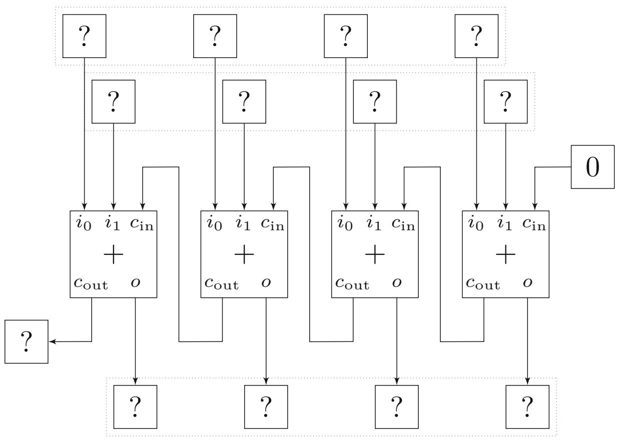


Figure 2.5: Traditional 4-bit addition circuit

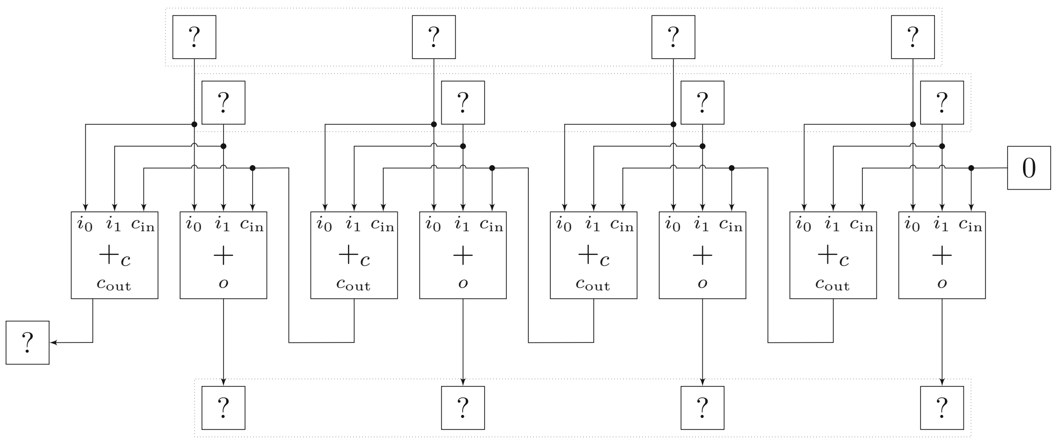


Figure 2.6: 4-bit addition circuit split up into individual sum and carry instructions

Speaking of which, the common part of injecting a set of `Tags` into the `State` input can be extracted, as in Figure 2.7, but as it needlessly complicates diagrams, it will be drawn as one instruction node from here on. The current implementation implements this arrangement, but one could just as easily allow arbitrary modifications of the `State` input.

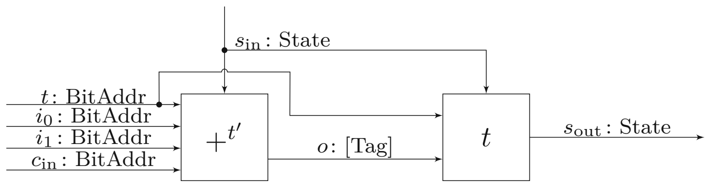


Figure 2.7: Factoring out the common parts; t is a generic "tag attaching instruction"

In a similar vein to addition and subtraction, other boolean operations like inclusive and exclusive `or` follow the exact same pattern as the `and` instruction $\wedge$ in Figure 2.4.

Other instructions only exist in group 3, i.e. producing type tags for bit addresses, but their corresponding boolean instruction would implement the identity function: When bits are shifted or copied between registers and memory, the crucial parts are where the bits "come from" and where they "go to", not the instructions themselves but the "wiring", so to speak. In all of these cases, bits are merely copied from one place to another. We usually think of "shifting a register" rather than a special combination of copy operations on individual bits. Nevertheless, in order to arrive at a more homogenous set of abstract base instructions, the interpretation of copy operations with complex "wiring" is used. In a similar manner to specifying the fate of a single result bit of an arithmetic addition—and have the implementation deal with the combination of these into a "larger" addition on registers—the user specifies how to process type tags for a single bit that is copied from one place to another. Even though logical shifts and rotations, as well as loading from and storing to memory are all built from the same primitive bit-replacement instruction we differentiate between them on the type tag level, that is to say, there are individual instructions for a single bit being shifted, rotated, loaded into a register from memory, or stored from a register to memory.

Figure 2.8 shows the two forms these instructions can take: $\ll^t$—*logical shift left*—shares its structure with instructions *logical shift right*, *arithmetic shift right* and *rotate right*. Shift and rotation instructions can use the input n to incorporate how far a bit has been moved, for example, to interpret a shift operation as multiplication, or division by a power of two. Here, the difference between shifting left versus right comes into play. For other bits, one might choose to interpret shifting as merely changing the position of bits as part of a register. The *store* instruction str^t—to memory—has a complementing counterpart in *load*—from memory. The additional input here is the location or address a, holding the memory address, from which to load a bit—or into which to store it. As the whole address affects the resulting bit, there is no way to assign address bits to the bits which are read or written, meaning `BitAddr` is not a suitable type. In the Arm A64 instruction set and my abstract RISC design, such an address can only be stored in a register; a *register id*—type `RegId`—represents just this piece of information. Some options here include transfering type tags from source bits in memory to the read value, declaring certain addresses in memory to be able to only hold certain types, or only preserving the desired tags for a combination of the correct value type stored at an address in memory which is designated to hold this exact type.

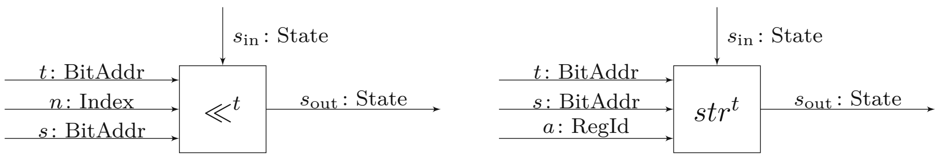


Figure 2.8: Two instructions which are only meaningful on the type tag level

One last remaining instruction constitutes a peculiar case. What bits “go into the making of” a single result bit of a multiplication? On the level of real-world instruction sets, many instructions seem to be of comparable complexity: bitwise or arithmetic, shifting and rotating, they all work on two, three, or sometimes four registers. On the level of individual bits however, more significant differences emerge. Bitwise operations constitute the simplest case, where we can simply iterate over all indices i and it always holds that the i^{th} result bit depends on the two i^{th} input bits. Addition and subtraction are more complex. There, in addition to the i^{th} input bits, output depends on the carry bit produced by the $(i-1)^{\text{th}}$ iteration. $i = 0$ becomes a special case, but apart from that all i cases follow a uniform pattern of a fixed number of inputs and the relations between them. Multiplication could be said to be “one step” more complex still. Not even the number of inputs remains constant now. We can interpret multiplication as the summation of products—David Goldberg’s description of multiplication [HP12] differs sightly, but the principle is the same—, as in this example for 8-bit values:

$$\begin{array}{lllllllll}
 & & a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0 \\
\times & & b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0 \\
\hline
= & b_0 \times & a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0 \\
+ & b_1 \times & a_6 a_5 a_4 a_3 a_2 a_1 a_0 \ 0 \\
+ & b_2 \times & a_5 a_4 a_3 a_2 a_1 a_0 \ 0 \ 0 \\
+ & b_3 \times & a_4 a_3 a_2 a_1 a_0 \ 0 \ 0 \ 0 \\
+ & b_4 \times & a_3 a_2 a_1 a_0 \ 0 \ 0 \ 0 \ 0 \\
+ & b_5 \times & a_2 a_1 a_0 \ 0 \ 0 \ 0 \ 0 \ 0 \\
+ & b_6 \times & a_1 a_0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \\
+ & b_7 \times & a_0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \\
\hline
= & & r_7 r_6 r_5 r_4 r_3 r_2 r_1 r_0
\end{array}$$

This way, we only need to consider multiplication where one operand is either 0 or 1, trivial calculations. Let us now examine the bits of the resulting 8-bit value—r_0 through r_7—individually. The resulting bits may be expressed using boolean operators $\wedge$, $\vee$ and $\oplus$, but it is sufficient to reduce those to simpler operators which only tell us what inputs the result depends on. For example, for arbitrary booleans x and y, $x \wedge y$ can be said to produce a value that is dependent on the values of x and y. $0 \wedge y$, on the other hand is known to be always 0, so it is in fact not dependent on the value of y. Let $\wedge_d$, $\vee_d$ and $\oplus_d$ be operators which produce sets of inputs on which the result of their corresponding boolean operators depend, defined by the following equations. Here x and y stand for individual bits while $\mathbb{X}$ and $\mathbb{Y}$ stand for sets of dependencies. $\wedge_d$, $\vee_d$ and $\oplus_d$ are overloaded to work on both.

$$\begin{aligned}
x \wedge_d y &= \begin{cases} \varnothing & \text{if } x = 0 \vee y = 0 \\ \{x, y\} & \text{otherwise} \end{cases} \\
\mathbb{X} \wedge_d \mathbb{Y} &= \mathbb{X} \cup \mathbb{Y} \\
x \vee_d y &= \begin{cases} \varnothing & \text{if } x = 1 \vee y = 1 \\ \{x, y\} & \text{otherwise} \end{cases} \\
\mathbb{X} \vee_d \mathbb{Y} &= \mathbb{X} \cup \mathbb{Y} \\
x \oplus_d y &= \{x, y\} \\
\mathbb{X} \oplus_d \mathbb{Y} &= \mathbb{X} \cup \mathbb{Y}
\end{aligned}$$

Let $r \mathrel{\widehat{=}} \mathbb{R}$ denote that the value of bit r depends on the set $\mathbb{R}$. Multiplication of individual bits reduces to the boolean operator $\wedge$, so it follows that $r_0 = a_0 \wedge b_0$, and thus $r_0 \mathrel{\widehat{=}} a_0 \wedge_d b_0 = \{a_0, b_0\}$—the value of r_0 depends on a_0 and b_0. All other intermediate products which are summed up in r_0 are independent of any input bits, because following the definitions just given, they are all 0, regardless of input.

For other i the expressions for r_i can become rather unwieldy; building functions equivalent to binary addition circuits should simplify matters. Half-adders are one way to build full-adders; half-adders are split into functions h_d for the sum, and h_d^c for the carry bit; full-adders are split into f_d and f_d^c.

$$\begin{aligned}
h_d(x, y) &= x \oplus_d y &&= \{x, y\} \\
h_d^c(x, y) &= x \wedge_d y &&= \{x, y\} \\
f_d(x, y, c) &= h_d(h_d(x, y), c) &&= \{x, y, c\} \\
f_d^c(x, y, c) &= h_d^c(x, y) \vee_d h_d^c(h_d(x, y), c) &&= \{x, y, c\}
\end{aligned}$$

By defining these new functions—h_d, h_d^c, f_d and f_d^c—using $\wedge_d$, $\vee_d$ and $\oplus_d$, they are, again, applicable to individual bits—as well as sets. For sets they may be further reduced to:

$$\begin{aligned}
h_d(\mathbb{X}, \mathbb{Y}) & &&= \mathbb{X} \cup \mathbb{Y} \\
h_d^c(\mathbb{X}, \mathbb{Y}) & &&= \mathbb{X} \cup \mathbb{Y} \\
f_d(\mathbb{X}, \mathbb{Y}, \mathbb{C}) &= (\mathbb{X} \cup \mathbb{Y}) \cup \mathbb{C} &&= \mathbb{X} \cup \mathbb{Y} \cup \mathbb{C} \\
f_d^c(\mathbb{X}, \mathbb{Y}, \mathbb{C}) &= (\mathbb{X} \cup \mathbb{Y}) \cup ((\mathbb{X} \cup \mathbb{Y})) \cup \mathbb{C}) &&= \mathbb{X} \cup \mathbb{Y} \cup \mathbb{C}
\end{aligned}$$

Let r_i^j be the intermediate sum of only the first $j + 1$ summands of r_i. This way $r_i^i = r_i$ and j runs from 0 to i; similarly, let c_i^j be the carry of the intermediate sum of only the first $j + 1$ summands of r_i. Note that for all i, $c_i^0 \mathrel{\widehat{=}} \varnothing$, because there is no addition operation involved in creating the sum of a single value, so no carry bit is produced. The following equations describe how to arrive at the dependencies of any result bit in a recursive manner.

$$r_i^j \mathrel{\widehat{=}} \begin{cases} f_d(a_{i-j} \wedge_d b_j, r_i^{j-1}, c_{i-1}^j) & \text{if } i \geq j > 0 \\ a_i \wedge_d b_0 & \text{if } i \geq j = 0 \\ r_i^i & \text{otherwise} \end{cases}$$

$$c_i^j \mathrel{\widehat{=}} \begin{cases} f_d^c(a_{i-j} \wedge_d b_j, r_i^{j-1}, c_{i-1}^j) & \text{if } i \geq j > 0 \\ \varnothing & \text{otherwise} \end{cases}$$

The three cases of r_i^j describe—from top to bottom—the general case, the special case of the sum of a single bit, and the remaining cases where one bit of the product is known to be 0, respectively. In particular, we can easily confirm the base cases, so to speak, r_0^0 and c_0^0:

$$\begin{aligned}
r_0^0 = r_0 &\mathrel{\widehat{=}} a_0 \wedge_d b_0 = \{a_0, b_0\} \\
c_0^0 &\mathrel{\widehat{=}} \varnothing
\end{aligned}$$

Slightly abusing syntax, one may derive an example as follows:

$$
\begin{aligned}
r_2 = r_2^2 \mathrel{\widehat{=}} \{a_0, b_2\} \cup r_2^1 \cup c_1^2 \\
&= \{a_0, b_2\} \cup (\{a_1, b_1\} \cup r_2^0 \cup c_1^1) \cup \varnothing \\
&= \{a_0, b_2\} \cup (\{a_1, b_1\} \cup \{a_2, b_0\} \cup (\{a_0, b_1\} \cup r_1^0 \cup c_0^1)) \\
&= \{a_0, b_2\} \cup (\{a_1, b_1\} \cup \{a_2, b_0\} \cup (\{a_0, b_1\} \cup \{a_1, b_0\} \cup \varnothing)) \\
&= \{a_0, a_1, a_2, b_0, b_1, b_2\}
\end{aligned}
$$

As expected, the i^{th} output bit depends on the input bits indexed 0 through i. $\times^t$ can thus be implemented as shown in Figure 2.9, but its type signature does not guarantee that the lists of input bits are in any specific order; a different representation—such as a type for secutive ranges of bits—might be preferable.

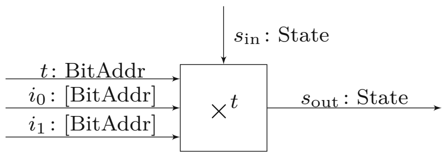


Figure 2.9: Multiplication instruction on type tags

Instructions on Registers

In the final step, instructions on bits may now be combined to form instructions on whole registers, the fourth group of instructions. One could further argue that they represent two distinct categories: instructions on registers made by combining multiple instructions on individual bits, and composite instructions which are themselves just concatenations of instructions on registers. Some instructions of a real-world instruction set fall into the former category, many fall into the latter.

Similar to instructions on type tags, which can have a few different types of interface—see Figure 2.7—it is practical to slightly alter the interface of `MBit` instructions: by incorporating the logic to extract and inject information about bits from and into a `State` value into the instructions themselves, the different types of instructions become more uniform and the next series of diagrams becomes easier to read.

Sometimes it may be necessary to add helper instructions ; these helpers may also only be present on the level of bits or `MBits`, i.e. have no influence on type tags whatsoever; an example is shown in Figure 2.10: This instruction assigns the sum of registers A and B to register T; it also reads from and writes to registers C^i and C^o which hold the bits which are carried into, and out of, their corresponding index positions, respectively. The $=_0$ instruction is only used on the first *carry in* bit and simply sets it to 0; $=$ copies a carry bit going out of the i^{th} position to carry bit going into the $i + 1^{\text{th}}$. C^i and C^o are an example of additional temporary registers which were alluded to in a Section 2.6.2.

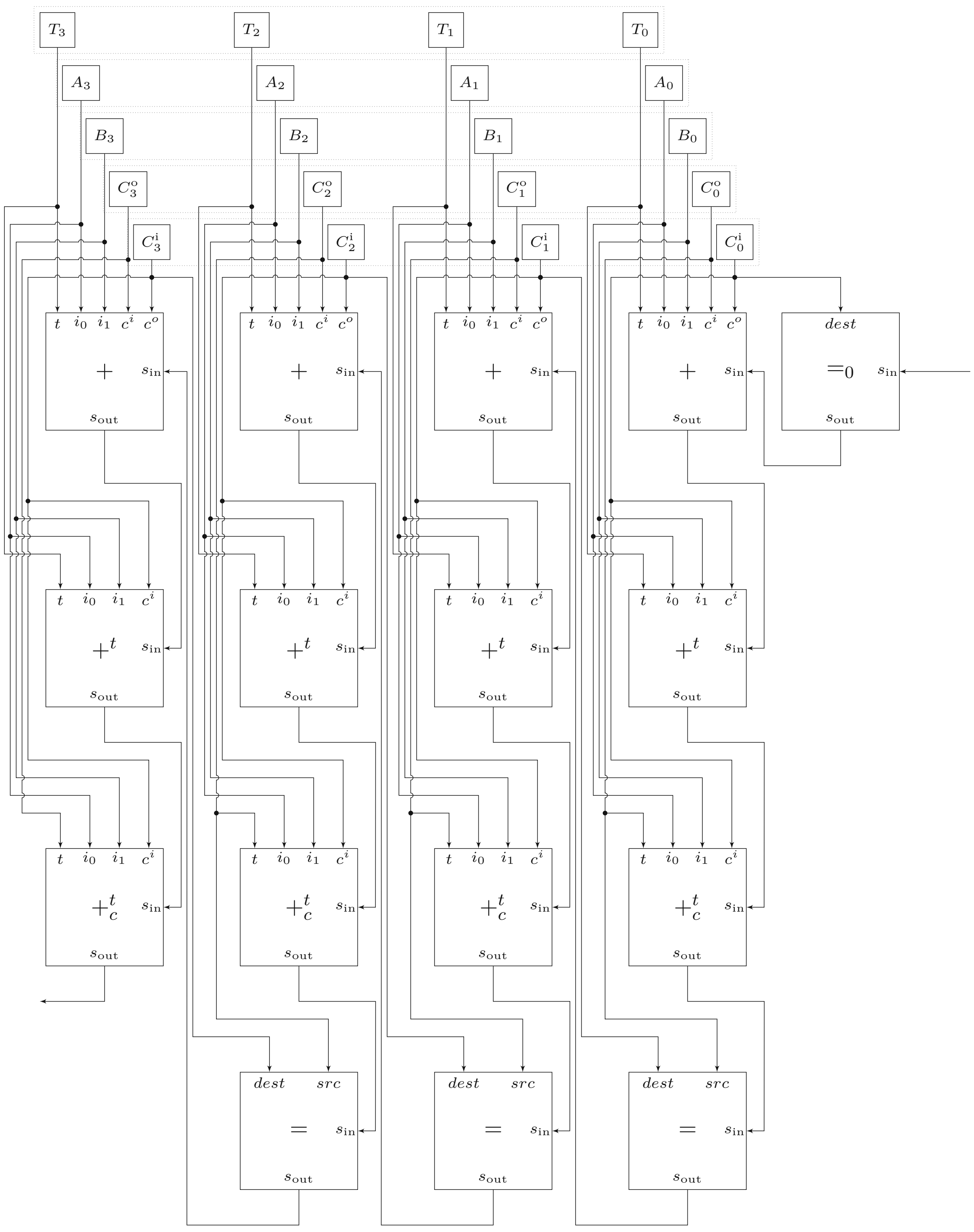


Figure 2.10: Addition instruction on registers; register T is assigned the sum of registers A and B; C^o and C^i are temporary registers

The addition is split up into three instructions per bit: $+$ handles `MBits`, which is fixed; $+^t$ and $+^t_c$ are the user-definable instructions producing tags for result bits and carry bits, respectively. $+$ uses all inputs and handles the `MBit` portion of result and carry bits, while the address of the carry bit is not part of the input of $+^t$ and the result bit address is missing from the inputs of $+^t_c$; all other inputs are the same; a `State` value is first passed into $=_0$ on the far right, and then passed down each column from top to bottom, moving through the remaining columns—with each column handling one indexed position of registers—from right to left.

The more common and simpler case is having one instruction handling `MBits` and another handling type tags for the same bit. In the case of bitwise operators, such as $\wedge$ in Figure 2.11, the wiring is also considerably less complex.

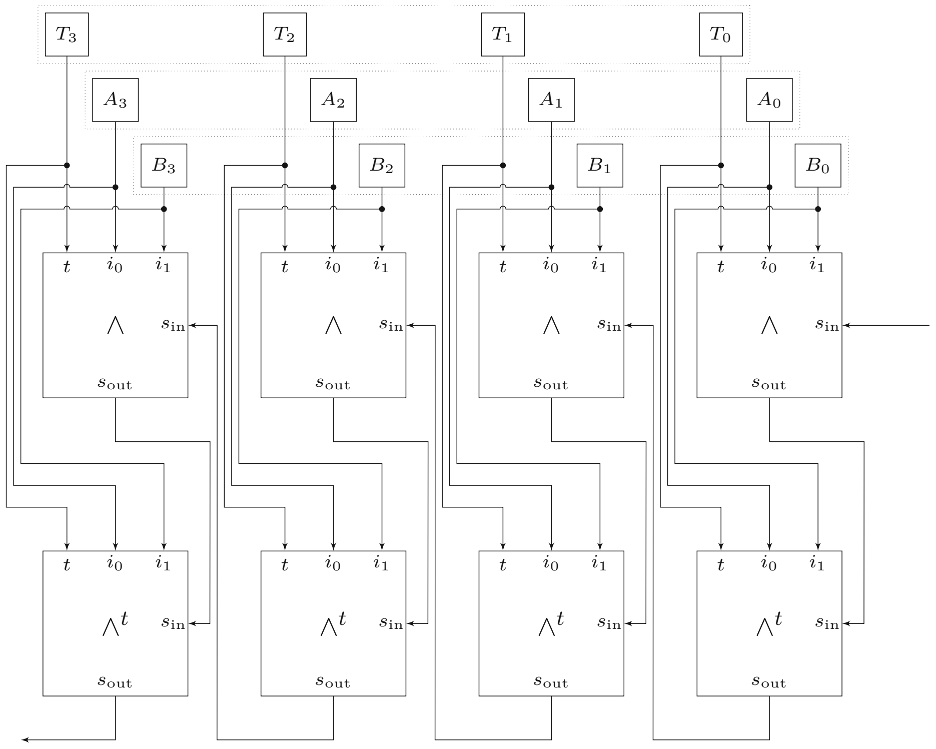


Figure 2.11: Bitwise and instruction on registers; register `T` $=$ `A` $\wedge$ `B`

Lastly, some operations do not separate as cleanly into instructions on individual bits. In logical shift and rotate, as well as load and store instructions, individual bits are merely copied to a new location. They can all be represented by the same assignment instruction—labeled $=$ in the presented diagrams. What matters is the logic behind calculation of the right source and destination bit addresses. This may not ordinarily be specified for individual bits—for example, in a shift register built from a collection of flip flops—but I nevertheless choose to disregard the efficiency of such a solution and to follow the pattern laid out so far, namely, to keep the instructions for individual bits separated where possible. For logical shifts this can be accomplished via specialized multiplexer circuits. For example, in Figure 2.12, the correct source for each of the four bit destinations is chosen from the four input addresses and an additional source for a constant zero bit. Each variant of the *mux* instruction takes into account the number of bits to be shifted and adds a constant that is appropriate for the index in question: for a shift value of n, mux_j outputs the address at input i_{j+n} if $j + n$ lies inside the range of regular input indices 0 through 3, and the zero bit's address i_z otherwise. The second row of $=$ instructions in Figure 2.12 copies `MBit` information from the address determined in the top row to a fixed destination address; the bottom row of $\gg^t$ instructions takes the same inputs, as well as the amount by which the bits are shifted, and produces type tag information.

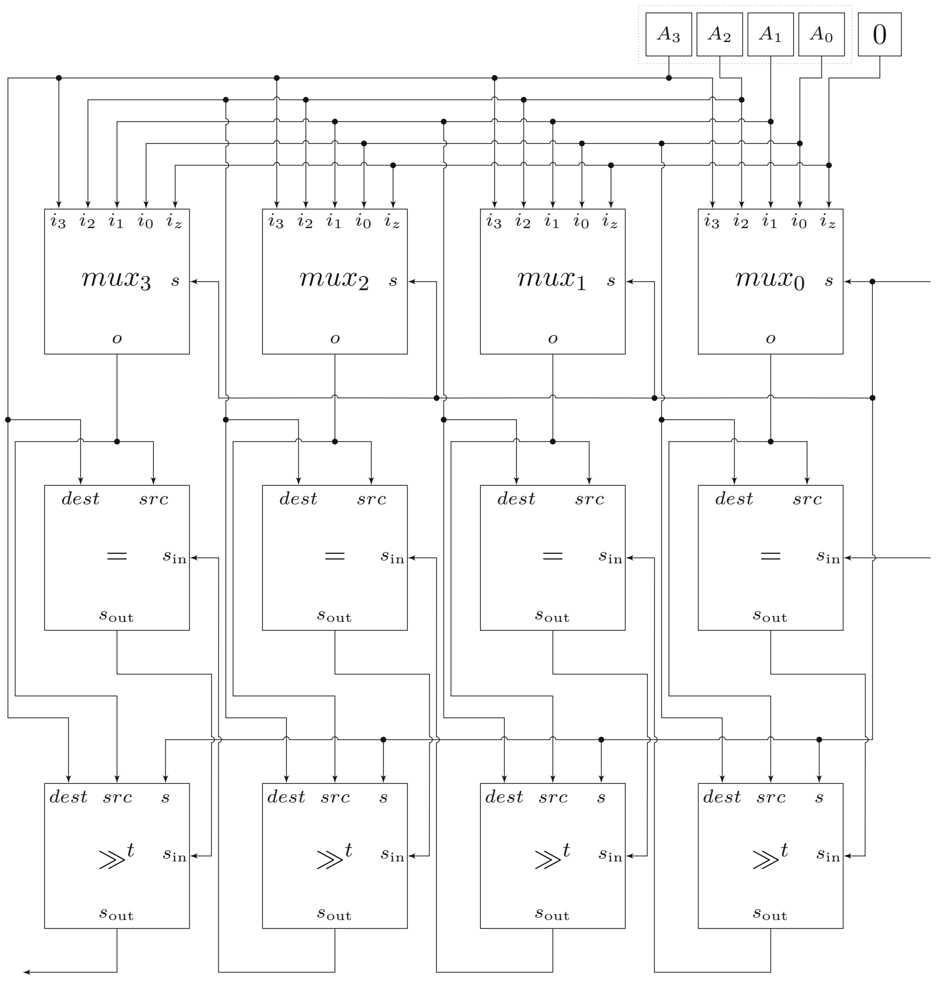


Figure 2.12: 4-bit logical shift right

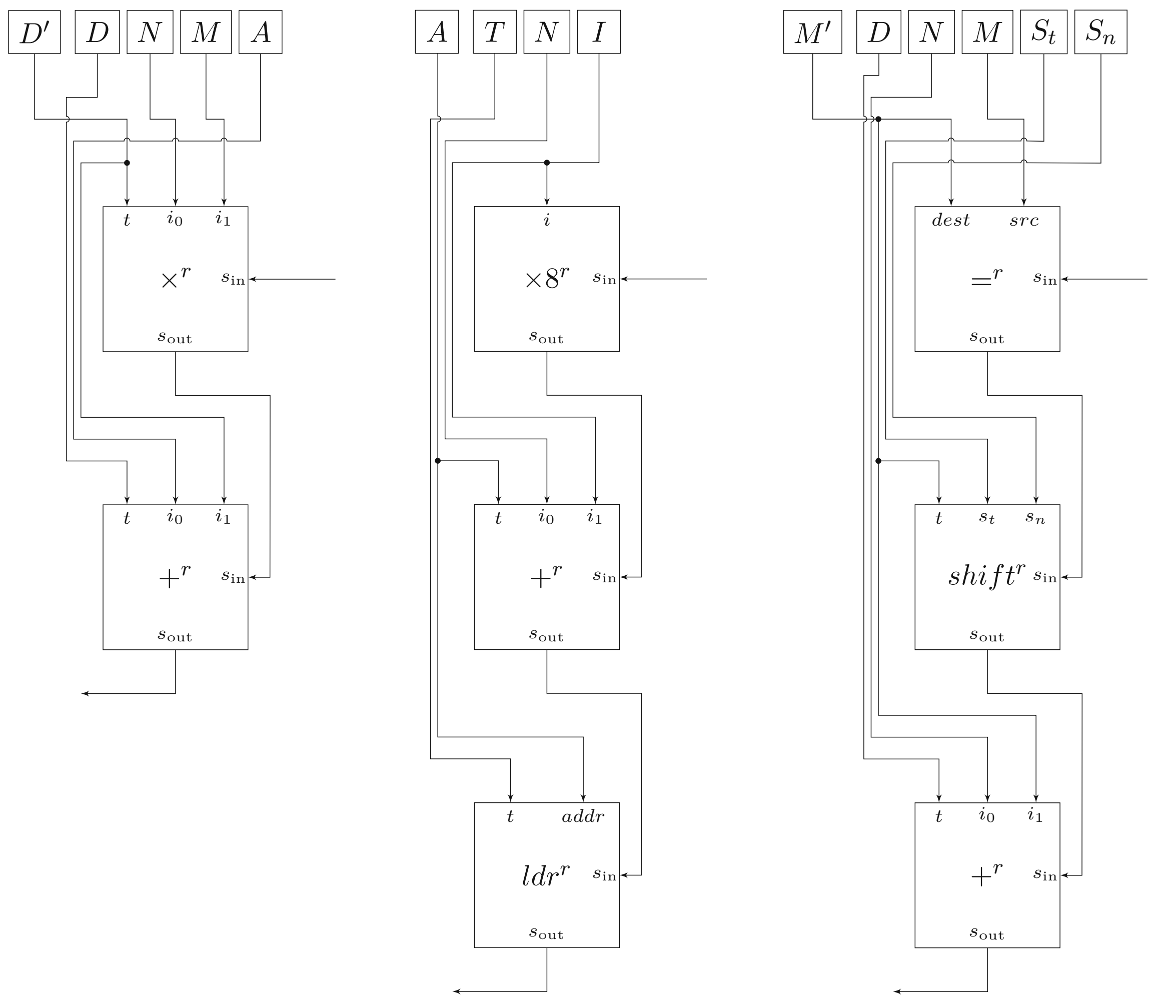
Other shift operations differ only in the wiring necessary to produce the correct source and destination pairs, and the type tag instruction; the $=$ instruction is always the same. Instead of an array of different type tag instructions one could also use a single $=^t$ instruction for copying, with some additional input which would specify what kind of operation is currently being performed. This would trade off the number of different instructions to be implemented with the combined complexity of a single, more general, instruction.

Diagrams for store and load instructions would feature wiring which is even more convoluted than the ones already shown, but the underlying principles would be similar to that of logical shifts, so for now they are omitted.

One last step is required to arrive at instructions which parallel the specific instructions of the Arm A64 instruction set: multiple instructions on registers may now be combined to form more complex instructions, similar to the ones found in real-world instruction sets.

Figure 2.13 gives three examples; an instruction on registers—not on individual bits—is identified by its r superscript. The A64 multiplication instruction `mul` is an alias to the "multiply-add" instruction `madd`—which calculates $d = n \times m + a$—with the a register being fixed to the zero register `XZR`—or `WZR`, depending on whether it is the 64-bit or 32-bit version of the instruction.[5] In Figure 2.13a the inputs `D`, `N`, `M` and `A` are complemented by a temporary register `D'`. First n and m are multiplied and the result d' stored in register `D'`. Then, the result of $d' + a$ is stored in `D`. Not shown here is an instruction to forget about the temporary register `D'`. In the implementation this is mostly done to avoid cluttering the resulting `State` value with values which—while mostly harmless—are ultimately superfluous. Figure 2.13b represents one variant of the `ldr` (immediate) instruction. When loading 64 bits from an address calculated from a base address in register `N` and an offset which is encoded as part of the instructions opcode—here represented by input register `I`—first the offset is multiplied by 8 using special instruction $\times 8^r$, then address $a = n + i \times 8$ is calculated and stored in temporary register `A`, and finally the simple load instruction introduced earlier loads 8 bytes at the address held in register `A` and stores it in the target register `T`. Here the target register is needed because the final address is not retained; all registers except for `T` are left untouched. Lastly, Figure 2.13c shows one of the many variations of addition which are available in Arm A64: `add` allows us to first shift input register `M` according to a shift type S_t—logical shift left, logical shift right, or arithmetic shift right—and the amount by which the register's bits are shifted, S_n, to then store the result of input `N` plus the shifted value of `M` in target register `D`. In order not to lose the original contents of register `M`, its contents are first copied to a temporary register `M'`—using $=^r$—where the value is then shifted—$shift^r$, which is meant to represent all the logic necessary to pick the correct shifting instruction and execute it. $+^r$ calculates $d = n + m'$ which it stores in register `D`, `M'` may be discarded and `N` and `M` remain unaltered.

[5]One might wish to implement a "pure" multiplication instruction, and thus deviate slightly from the actual machine architecture, but this is infeasible due to the way instructions are stored as part of abstract machine states.



(a) Combined multiplication and addition (b) Load from base address plus offset (c) Addition with shifted input register

Figure 2.13: Full instructions

As originally intended, it was possible to use the same fundamental addition instruction $+^r$ in three examples; it is also used in addition instructions, which feature extending a smaller value to a larger register size, addition involving immediate values stored as part of opcodes, load operations which employ different address calculations—often containing an addition—and others. Likewise, shifting is used in various special forms of other instructions like subtraction, bitwise operations and loading of constant values into parts of registers.

One type of instruction needs special handling and is yet to be addressed. What does it mean to load from an address—or write to one—which is not fully known? Some bits of the address may be left unspecified.

No strategy seems to be universally applicable[6], which is why I opt to let the user specify how to handle addresses. Figure 2.14 defines the options which are available; store and load instructions will produce a user defined set of new states, one each per `StoreReq` or `LoadReq` given. `StoreReq` may either specify a single address to use using `StoreAddr`—multiple addresses may given as separate `StoreReq`s—, request to delete all information about a range of memory using `ForgetRange` or to simply signal via `IgnoreStore` that no changes to the state are to be made. `LoadReq` is structured similarly, with the exception that the largest unit that may be necessary to forget is a single register. If the user decides to return an empty set of requests a default set is used; in case the address to load from is unique, it is used as is; if it is partially unknown, the default is to forget all information about the register that is being stored to—using `ForgetReg`; for stores, a fully known address is used, or otherwise the store instruction is ignored and takes no effect. There is an argument to be made that the last default should rather be `ForgetRange` covering the whole of system memory, but this would also remove the programs instructions from memory, leading to the next iteration having to halt the algorithm, because it would be entirely unknown how to proceed. This scenario would be almost equivalent to a hard reset of the machine, hardly an interesting state.

```haskell
type ByteAddr = Word64

data StoreReq = StoreAddr    ByteAddr
              | ForgetRange  ByteAddr ByteAddr
              | IgnoreStore

data LoadReq = LoadAddr    ByteAddr
             | ForgetReg
             | IgnoreLoad

storeMemAddr :: Width -> RegId -> RegId -> State -> [StoreReq]
storeMemAddr width srcReg destAddr state = {- ... -}

loadMemAddr :: Width -> RegId -> RegId -> State -> [LoadReq]
loadMemAddr width destReg srcAddr state = {- ... -}
```

Figure 2.14: User-defined handling of (partially unknown) addresses

2.6.3 Constraints and Errors

So far we have seen how to build instructions for an abstract machine; the user may define some part of their behavior and then combine these instructions to form a *program*, or rather, a description of an abstract machine state, that is, roughly, an image of a

[6]The following approach would always work, but it easily leads to insurmountable numbers of possible states: For each unknown bit we may simply split one state up into two, one where it is 0, and one where it is 1.

computer's registers and memory. Loading and running this program will produce a number of reachable states, but how are we to say which of these states are meant to be reached—and which states are not?

Because, as I claim, one cannot assert that a machine instruction is "illegal", and thus cannot be used at a specific place in a program, we need another way to tell the assembler that certain program configurations are undesirable, and that programs containing them are thus to be rejected. The mechanism offered in order to put constraints on a program—the "upper limit" alluded to in the overview—is an annotated version of abstract machine states—or formulas describing states; the means to report an error in the program is a data structure, describing which parts of a constraint where met and which were not, which reuses the aforementioned annotations.

Constraints and errors both take the form of **trees**.

Any formula describing one or more abstract machine states may be combined with a human-readable annotation to form a constraint **leaf**. The annotation needs to be able to convey some meaning to the user. It could be a simple string, but I choose to add some additional structure: It is useful to have each annotation contain a bit address so one may more easily differentiate between, for example, "`Color` at `X0.0`" and "`Color` at `X20.31`", instead of just being alarmed that a "`Color`" is not present, without telling us where the color is supposed to be. Special cases like that of a consequence may be built from a pair of annotations: "`Leaf-Tag` at `X0.0`" ⇒ "`Payload` at `X0.1`" or "`Node-Tag` at `X0.0`" ⇒ "`Tree-Pointer` at `X0.1` and `Tree-Pointer` at `X0.31`" could be used to describe a tree type implemented as a tagged union. In fact, I also use special annotation variations for $\wedge$, $\vee$ and $\neg$, i.e. conjunction, disjunction and negation of formulas, as well as a special `PC=N` ⇒ `A` variant which is meant to convey "If the program counter has value N, then A" i.e. the ubiquitous case of a condition which is expected to hold only at certain locations of a program.

Leaves may then be combined to form $\wedge$ and $\vee$ nodes, each contains a set of child nodes and leaves, as well as an annotation, which is meant to describe the combined effect of its children.

The usefulness of compartmentalizing and annotating abstract state formulas like this, becomes apparent when looking at how they are used and transformed into errors: Each newly derived abstract machine state is checked for entailment against each user-specified constraint tree. Each node and leaf can be transformed into an un-annotated state. If the root of a tree is not entailed by a newly derived state, that is an error. To pinpoint the precise location of an error—or the reason behind it—$\wedge$ and $\vee$ nodes are further inspected: If $a \wedge b \wedge \ldots$ constitutes an error, is it in a; is it in b; or is it ...? For $x \vee y \vee \ldots$, what exactly is the problem in x; what is it in y; and so on.

By iterating over the tree structure of a constraint, an *error tree* may be derived: For leaves, simply drop the abstract state, it is no longer needed; for $\wedge$ nodes, retain and transform only those children whose formula is not entailed by the new formula which is currently being tested; for ∨ nodes, transform all child nodes; they are all known not to be entailed by the new formula because one would suffice to entail the whole node.

A node that is entailed by a new abstract machine state, can be dropped. If it is the root node of a constraint tree, no error has occurred.

Let me go back to examples introduced in Figure 2.3.

Figure 2.15 represents an error tree that can be read as follows: In contrast to our expectations, a series of bits beginning at `X0.0` does not specify a `Color` type value, because neither the `RGB` nor the `Invert` case where applicable. A value is considered of type `RGB` if it contains three `Nat 16` tagged fields and three zero padding bits in the correct places—just as it was specified in Figure 2.3. The value in question is no `RGB` because the first padding bit at `X0.16` is not zero. All other parts are as expected for an `RGB` value—and thus grayed out in the diagram; they are only shown here to tell us what other errors could occur, and would not be present in the prototype's output. At the same time, the value does not equal the specific bit pattern which represents the singleton `Invert` type. Error trees have the most general description of the error at the root, and traversing its edges leads to error descriptions which become more precise as we go along.

One downside of this approach is that we do not always know which child node holds the information we are seeking: In the `Color` example, the error alone does not tell us whether the intended result was to be an `RGB` value—and we simply forgot to set bit `X0.16` to zero—or whether something went wrong while trying to set `X0.0` through `X0.50` to the `Invert` bit pattern.

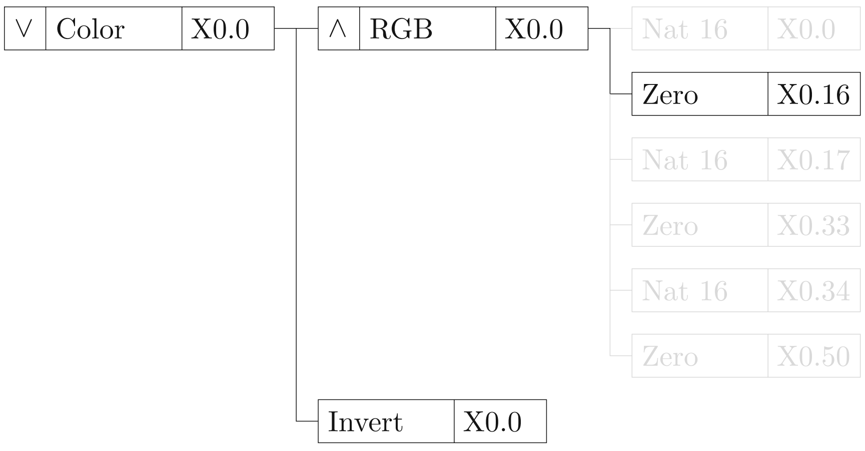


Figure 2.15: Error tree for `Color` type

The helpfulness of an error tree and the ability to even detect an error depends on how precisely the features of a type are specified: Figure 2.16 shows the result of checking the same state—which we previously considered to be erroneous—against a slightly less strict definition of `Color`, namely one where the three padding bits may have arbitrary values. In this situation **no error** would in fact be encountered. Regardless of whether an `RGB′` or an `Invert` value is expected, the fact that the present state conforms to the `RGB′` type is enough to satisfy the whole `Color` constraint. If our intention was to create an `Invert` bit pattern, but we somehow managed to produce the type tags needed for the more lenient `RGB′` constraint, no error is produced, even though checking for an `Invert` type would have produced one—in Figure 2.16, this error, which is ignored, is drawn with a dashed outline.

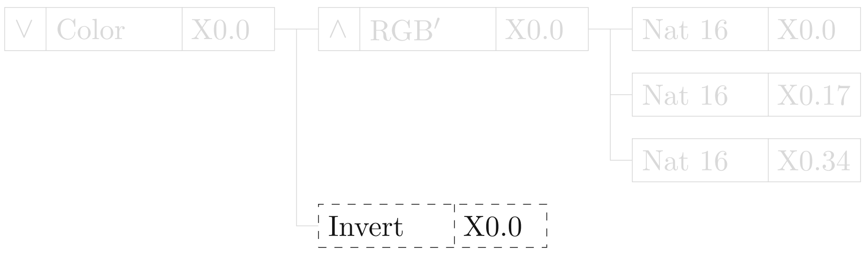
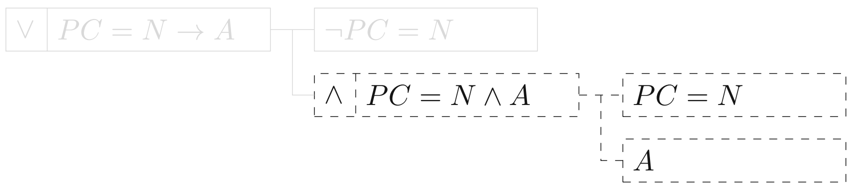
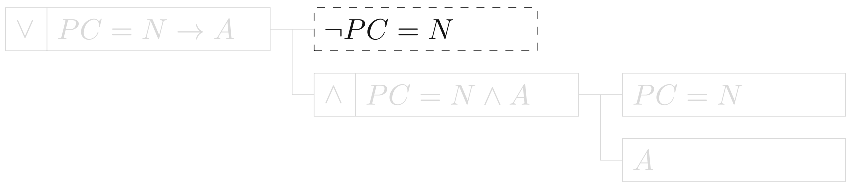


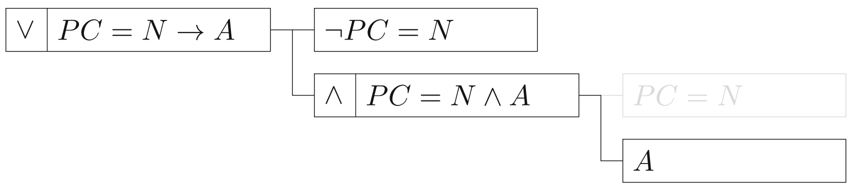
Figure 2.16: Non-Error tree for modified `Color` type



(a) No error; A is not meant to apply to this program location



(b) Constraint met; A holds



(c) Constraint not met; A should hold, but it does not

Figure 2.17: Three error trees for the three common cases of $PC = N$

The `PC=N ⇒ A` constraint may be by far the most widespread, warranting a closer look.

While some constraints are meant for the whole duration of a process, many common high-level language constructs translate to constraints in the form of a consequence. For example, whenever control flow arrives at the head of a function or method, certain registers and memory locations must hold values of specific types. A compile time assertion has a specific location in the program text; it is meant to describe what is required at this exact point. The correct types of return values need to be known at the return instruction. They can all take the form of `PC=N` ⇒ `A` constraints.

Figure 2.17 presents the three most relevant cases: If the value of the `PC` differs from the one specified, then this constraint is not meant to say anything about this position in the program; there is no error, see Figure 2.17a. If `PC=N` holds and `A` holds too, "things went according to plan"; understandably ¬ `PC=N` can not hold, but Figure 2.17b correctly shows the overall result as not an error. Finally, Figure 2.17c demonstrates the safeguarding function of the constraint: `PC=N` holds but `A` does not hold. That is an error.

2.6.4 "Escape Hatch"

Some abstract state transformations do not fit into the supplied functions on individual types, or do not have a clear parallel on actual hardware. We might want to perform a type cast, i.e. attach type tags to bit addresses. For example, when loading a constant into a register, the bits of the instruction's opcode—and the constant in encoded form usually takes up only a small fraction of those—often do not give us "enough space" to hold the desired type information. To alleviate this problem there exists the option to insert state transformation functions between machine instructions. Loading a typed constant may then consist of an instruction writing the desired bit pattern, combined with a function running right after that instruction's execution, which attaches the appropriate type tags. Another application could be setting up some preconditions right before the first instruction of a code block, or deliberately forgetting about some parts of a state. For example, temporary values which are no longer needed may still be lingering inside a register after returning from a function; by removing this information, multiple abstract states may collapse into one, leading to better performance during type checking and less irrelevant data "polluting" the output.

2.7 Domain Specific Language

In order to simplify the implementation I forego the definition of syntax and implementation of a parser. Programs are written in a domain specific language in Haskell, which is implemented with the help of a State monad. Together with Haskell's `do`-syntax this allows for programs to be written in a style close to conventional assembly listings.

Internally a program is simply a list of instructions which include labels, and functions to manipulate the abstract state before and after an instruction is run, as well as functions in order to create constraints from the information pertaining to an instruction—such as its address in memory; everything except for the instruction itself is optional. Using a few helper functions, it is then possible to write programs in the style shown in Figure 2.18: The listing contains, in order, an instruction with a label, one without, a third instruction without label but with functions manipulating the state before and after the instruction is run, as well as a function to build a constraint, and lastly, a jump instruction back to the start of the listing.

```
do -- first instruction, at label "main"
   "main" # mov (X 1) (X 0)
   -- second instruction, without label
   mov (X 2) (X 0)
   -- third instruction
   -- alter state before instruction
   pref (\pc instr -> {- ... -})
   -- alter state after instruction
   postf (\pc instr state states -> {- ... -})
   -- construct constraint using current address
   lf (\pc -> {- ... -})
   lsl (X 3) (X 1) (X 2)
   -- last instruction: jump back to "main"
   -- the label is looked up and replaced by an offset
   b "main"
```

Figure 2.18: An example listing

CHAPTER 3

Evaluation

The goal of this thesis is to answer the question "How can a type system handle individual as well as grouped bits and verify that they are processed correctly?"

A large part of this endeavor is to implement and evaluate an experimental prototype for a typed assembly language, which is not burdened with specific patterns to which programs must adhere.

This chapter splits into two parts: Section 3.1 is concerned with evaluating the prototype in the context of the first two sets of yes-no questions posed in the introduction's Section 1.2. The remaining, more open, questions are addressed in Section 3.2.

3.1 Evaluation of Polar Questions

The following yes-no questions which are repeated from Section 1.2 are used to evaluate the prototype, thereby concluding the experimental part of the thesis.

If questions of the first set can be answered with a "yes," then it is possible to implement a desired "high-level feature", as long as that feature can be said to me "made up of" the basic properties which the questions examined. A feature like enumerations can be implemented if the right questions all receive a positive answer. This may mean enumerations like the ones in the C programming language—then a "yes" on the very first question may suffice—or it may mean a more elaborate version—then other questions would have to be taken into consideration; that could be the second question, which is about differentiating between not just values but also their types.

> Is it possible to assign a name to a bit pattern, e.g. to use the name as a descriptive stand-in?

Yes. With a few limitations, we may implement something similar to a C macro or a constant in other languages. Figure 3.1 shows three ways to give a name to a value. The first line of "main" uses `movImm` which is an alias for the `movz` instruction. It writes the given value to the 16 least significant bits of a register; the remaining register is set to 0. This will only work for constants which have a width of 16 or less. The second option uses `ldrLit` to write a doubleword—64 bits—to register `X1`. The value itself is found at the very end of the listing; the `ldrLit` instruction uses the label "constant1" which is translated to an immediate offset by the assembler. This approach is somewhat cumbersome as it is up to the user to find a suitable place to put the 64 bits of data. The limits of the hardware pertaining to immediate offsets have to be taken into account. For example, if we wish to have multiple instructions use the same constant in memory, they all have to be within a certain range around the data's address in memory. The third and last option automates the placement of the constant in memory, at the expense of code size and runtime: `loadConstant` expands to three statements; the second statement places the 64 bits of data in memory, which are then loaded into a register by a `ldrLit` instruction, which comes last. The first statement is an instruction to jump over the data, which shall not be interpreted as machine instructions. We can then use the `singleton` function in "halt" to check that the constants are set correctly.

```
let constant0 = 2
    constant1 = 3
    constant2 = 5
    loadConstant reg value =
      do b ("load_" ++ show value)
         ("dw_" ++ show value)    # doubleWord value
         ("load_" ++ show value) # ldrLit reg ("dw_" ++ show value)
in makeProgram $
   do "main"      # do movImm (X 0) constant0
                        ldrLit (X 1) "constant1"
                        loadConstant (X 2) constant2
      "halt"      # do lf (\pc -> atPC pc $ singleton
                                              (TNString "constant0")
                                              constant0
                                              (RegBit (X 0) 0))
                        {- similar checks for other constants -}
                        b "halt"
      "constant1" # doubleWord constant1
```

Figure 3.1: Three untyped constants

Can the type system differentiate between values which use the exact same bit pattern?

Yes. In Figure 3.2 `T` and `makeTypedConstant` show how we may simplify the definition of constants which are decorated with type tags. `make` creates a program listing which writes a 16-bit integer constant to a given register, and attaches tags to specify the type of the constant. `is` creates a `NamedExpr` which is can be thought of as a tuple of a formula specifying the bit pattern and tags, as well as a human readable description of that formula. It is used when checking constraints and reporting errors. If we change `is radio (X 0)` at label "halt" to `is radio (X 1)`, an error is reported telling us that (X 1) does not hold a `Radio` type constant; the bit pattern happens to be correct, but the type tags tell us it is in fact a `MuteOn` constant.

```
data T = T { make :: RegId -> DSLListing
           , is   :: RegId -> NamedExpr
           }

makeTypedConstant :: String -> Word16 -> T
makeTypedConstant name value =
  T { make =
        \reg -> do postf (\pc instr state ->
                            map (setTypeTag (TNString name)
                                            (RegBit reg 0)))
                   movImm reg value
    , is = \reg -> singletonTag muteSourceTi (TNString name) value
                                   (RegBit reg 0)
    }

let radio   = makeTypedConstant "Radio"  1
    muteOn  = makeTypedConstant "MuteOn" 1
in makeProgram $
   do "main" # do make radio  (X 0)
                  make muteOn (X 1)
      "halt" # do lf (\pc -> atPC pc $ is radio  (X 0))
                  lf (\pc -> atPC pc $ is muteOn (X 1))
                  b "halt" -- placeholder
```

Figure 3.2: A tool to build `newtype`-like wrappers

> Is it possible to define a type as describing elements from either one **or** another type; more generally, is it possible to build a *sum type* from multiple other types?

Yes. Figure 2.1 in Section 2.6.1 showed how an enumeration type expressed as a "naked" formula can be built from a number of "singleton" types. For constraints a different function does essentially the same: `enum` in Figure 3.3 builds a `NamedExpr` from a `TypeName`—which could be a string—and a list of types—in the form of functions from a `BitAddr` to a `NamedExpr`—and produces yet another such function—again from `BitAddr` to `NamedExpr`. `makeEnum` could be an option to go along with `makeTypedConstant`, which we just saw in Figure 3.2: A number of constants is here combined to a new type which may hold any of the constant, and is additionally tagged using the given name. `source` can then be specified as simply `makeEnum "Source" [ tape, radio ]`. While `makeEnum` is a special case which fixes some decisions—all type names are `String`s; types are placed at bit 0 and use a full register—the more unwieldy `enum` can be used to combine arbitrary types and not just constants; it works in the same way as a logical *or* on propositional formulas.

```
enum :: TypeName -> [BitAddr -> NamedExpr] -> BitAddr -> NamedExpr
enum tn options addr = NamedOr (TSName tn addr) $ map (flip ($) addr) options

makeEnum :: String -> [T] -> T
makeEnum name options =
  T { make =
        \reg -> postf (\pc instr state ->
                           map (setTypeTag (TNString name)
                                           (RegBit reg 0)))
    , is = \reg -> NamedOr (TSName (TNString name) (RegBit reg 0))
                            $ map (\o -> is o reg) options
    }

let muteOn = makeTypedConstant "MuteOn" 1
    radio  = makeTypedConstant "Radio"  1
    tape   = makeTypedConstant "Tape"   2       -- unused
    source = makeEnum "Source" [ tape, radio ] -- unused
in makeProgram $
   do "main" # do make radio   (X 0)
                  make muteOn (X 1)
      "halt" # do lf (\pc -> atPC pc $ is radio   (X 0))
                  lf (\pc -> atPC pc $ is muteOn (X 1))
                  b "halt" -- placeholder
```

Figure 3.3: `enum` and `makeEnum` combine multiple types in a logical *or* fashion—see also Figure 2.1

On a related note, is it possible to define *product types* like structs?

Yes. Product types are but one application of the `struct` function from Figure 2.2 in Section 2.6.1. The corresponding function for constraints and errors is found in Figure 3.4: The only significant difference to `enum` is that `struct` uses `NamedAnd` in place of `NamedOr`. Similar to `makeTypedConstant`, `makeTuple` in Figure 3.5 is a helper function to build tuples; it creates a `T` as before and some assembly code to combine values into a tuple that fits into a single register: `tupleListing` contains code at label "makePair" and is added to the end of the overall listing. It is then called with arguments in registers `X1` and `X2` and outputs a tuple including appropriate type tags in register `X0`. The first constraint at "halt" makes sure there is a tuple in `X10`—which was copied from X0. The other two constraints lead to errors: There is no Pair in X11 because there is no Nat 16 at X11.8, and there is no Pair in X12 because there is no Nat 8 at X12.0.

```
struct :: TypeName -> [BitAddr -> NamedExpr] -> BitAddr -> NamedExpr
struct tn xs addr = NamedAnd (TSName tn addr) $ map (flip ($) addr) xs

bitWidth :: TypeName -> Int
bitWidth t = {- number of bits used to represent a t-typed value -}

makeTuple :: String -> [TypeName] -> (T, DSLListing)
makeTuple name members =
  let indices = scanl (+) 0 $ map bitWidth members
      setters reg =
        zipWith (\n i -> setTypeTag n (RegBit reg i)) members indices
      taggers reg =
        zipWith (\n i -> tagged n (RegBit reg i)) members indices
      t = T { make = \reg -> postf (\pc instr state ->
                                   map (foldl1 (.) (setters reg)))
            , is = \reg -> NamedAnd (TSName (TNString name)
                                          (RegBit reg 0))
                                 $ taggers reg
            }
  in ( t
     , do ("make" ++ name) # movImm (X 0) 0
          mapM_ (\(i, o, n) -> andSr (X 0) (X 0) (X i) LSL (Nat6 o))
                $ zip3 indices [1..] members
          make t (X 0)
          ret
     )
```

Figure 3.4: makeTuple creates “boilerplate code” for simple tuples; struct allows the creation of some other types too

Apart from tuples and structs, we may also combine types which are **overlapping**. Consider a type for three simple geometric shapes, which are centered around the origin: a square of length S, a circle with radius R, or an equilateral triangle with height H. For a single byte size, let us use two bits of an enumeration to specify the shape and the remaining six bits for an unsigned integer. This gives the following three patterns:

0	$0\, S_5\, S_4\, S_3\, S_2\, S_1\, S_0$	Square
0	$1 R_5 R_4 R_3 R_2 R_1 R_0$	Circle
1	$0 H_5 H_4 H_3 H_2 H_1 H_0$	Triangle

```
let (tuple, tupleListing) = makeTuple "Pair" [TNNat 8, TNNat 16]
in makeProgram $
   do "main" # do ldrLit (X 1) "seven"
                   ldrLit (X 2) "nine"
                   bl "makePair"
                   -- completed Pair in X10
                   mov (X 10) (X 0)
                   -- first half of Pair in X11
                   postf (\pc instr state ->
                             map (setTypeTag (TNNat 8)
                                             (RegBit (X 11) 0)))
                   andSr (X 11) (X 11) (X 1) LSL (Nat6 0)
                   -- second half of Pair in X12
                   postf (\pc instr state ->
                             map (setTypeTag (TNNat 16)
                                             (RegBit (X 12) 8)))
                   andSr (X 12) (X 12) (X 2) LSL (Nat6 8)
      "halt" # do lf (\pc -> atPC pc $ is tuple (X 10))
                   lf (\pc -> atPC pc $ is tuple (X 11)) -- no Nat16
                   lf (\pc -> atPC pc $ is tuple (X 12)) -- no Nat8
                   b "halt" -- placeholder
      "seven" # doubleWord 7
      "nine"  # doubleWord 9
      tupleListing
```

Figure 3.5: How to use `makeTuple`

Our simple drawing library shall let us create shapes and apply some 2D transformations to them. Rotation does not have any noticeable effect on circles, so we decide to skip over rotation code when we encounter a circle. As it so happens that the 7$^{\text{th}}$bit from the right is 1 for circles and 0 for other shapes, we may interpret this bit to mean “ignore rotation” if it is set. The whole type can then be described by the following two patterns:

$$\begin{array}{l} E_1 E_0 N_5 N_4 N_3 N_2 N_1 N_0 \\ \quad\ r \end{array}$$

E_1 and E_0 contain an enumeration telling us whether we have a square, circle or triangle; N_0 through N_5 always represent an unsigned integer—whether that be side length, radius or triangle height; and lastly, r, which coincides with E_0, means “do rotation” if it is 0 and “ignore rotation” if it is 1.

Definitions for this type are shown in Figures 3.6 and 3.7. This contains code to build the three types of shape from an integer, tag the bits with `Nat 6`, `Shape`, `Ignore Rotation`, and more. The eight bits of a full `Triangle` value then have tags as follows:

Bit	Tags
0	(Shape 0), (Nat 6 0)
1	(Shape 1), (Nat 6 1)
2	(Shape 2), (Nat 6 2)
3	(Shape 3), (Nat 6 3)
4	(Shape 4), (Nat 6 4)
5	(Shape 5), (Nat 6 5)
6	(Shape 6), (ShapeType 0), (Triangle 0), (Ignore Rotation 0)
7	(Shape 7), (ShapeType 1), (Triangle 1)

We could add tags to specify that the `Nat 6` in a triangle is simultaneously of type `Height`, perhaps in order to protect against mixing up height and radius elsewhere.

```
triangle :: T
triangle =
  T { make =
        \reg -> do andImm reg reg True (shiftBy 57) (ones 63)
                   postf (\pc instr state ->
                             map (setTypeTag (TNString "Triangle")
                                          (RegBit reg 6)))
                   orrImm reg reg True (shiftBy 57) (ones 1)
    , is = \reg -> singletonTag muteSourceTi (TNString "Triangle")
                                (Nat2 2) (RegBit reg 6)
    }

{- square and circle are defined similarly -}

shapeType :: T
shapeType =
  T { make =
        \reg -> postf (\pc instr state ->
                        map (setTypeTag (TNString "ShapeType")
                                     (RegBit reg 6)))
    , is = \reg -> NamedOr (TSName (TNString "ShapeType")
                               (RegBit reg 6))
                          $ map (\o -> is o reg)
                                [square, circle, triangle]
    }
```

Figure 3.6: Two bits mark a byte as a triangle, square or circle

Bit 6 represents the case where what could otherwise be considered two distinct fields of a struct, coincides in one bit; this bit is part of the `ShapeType` but can also be read as `Ignore Rotation`. The code block at label "ignoreRotation" extracts this one bit and makes sure that the result is correctly tagged. This tag has to come from a complete `Shape` value. The first line contains the constraint that `X1` must hold a `Shape` (`t` in

Figure 3.7) and the second to last line makes sure that the correct bit is extracted to `X0`. The former constraint catches errors like calling “ignoreRotation” on the wrong values, while the latter makes sure that this function implements the desired behavior.

```
shapeT :: (T, DSLListing)
shapeT =
  let t = T { make = \reg ->
              do make shapeType reg
                 postf (\pc instr state ->
                           map (setTypeTag (TNNat 6)
                                           (RegBit reg 0)))
                 postf (\pc instr state ->
                           map (setTypeTag
                                  (TNString "Ignore Rotation")
                                  (RegBit reg 6)))
                 postf (\pc instr state ->
                           map (setTypeTag (TNString "Shape")
                                           (RegBit reg 0)))
            , is = \reg ->
              NamedAnd (TSName (TNString "Shape")
                               (RegBit reg 0))
                 [ is shapeType reg
                 , tagged' (TNNat 6) (RegBit reg 0)
                 , tagged' (TNString "Ignore Rotation")
                           (RegBit reg 6)
                 , tagged' (TNString "Shape") (RegBit reg 0)
                 ]
            }
  in ( t
     , do "makeSquare"    # do mov (X 0) (X 1)
                               make square (X 0)
                               make t      (X 0)
                               ret
          {- makeCircle and makeTriangle are defined similarly -}
          "ignoreRotation" # do lf (\pc -> atPC pc $ is t (X 1))
                                movImm (X 0) 6
                                lsr (X 0) (X 1) (X 0)
                                andImm (X 0) (X 0) True
                                       (shiftBy 0) (ones 1)
                                lf (\pc -> atPC pc $ tagged'
                                     (TNString "Ignore Rotation")
                                               (RegBit (X 0) 0))
                                ret
     )
```

Figure 3.7: Type for three simple shapes

```
pair :: TypeName -> BitAddr -> NamedExpr
pair t addr =
  NamedAnd (TSName (TNPair t) addr)
           [ tagged t addr
           , shiftLeft (bitWidth t) (tagged t) addr
           ]

fixedSizeArray :: TypeName -> Int -> BitAddr -> NamedExpr
fixedSizeArray t n addr =
  NamedAnd (TSName (TNArr t n) addr)
           $ map (\i -> shiftLeft (i * (bitWidth t)) (tagged t) addr)
                 [0..n-1]

dominoes :: TypeName -> Int -> BitAddr -> NamedExpr
dominoes t n addr =
  let piece addr =
        NamedAnd (TSName (TNString "Piece") addr)
                 [ tagged t addr
                 , shiftLeft (bitWidth t) (tagged t) addr
                 ]
  in NamedAnd (TSName (TNString "Dominoes") addr)
              $ map (\i -> shiftLeft (i * (bitWidth t)) piece addr)
                    [0..n-1]
```

Figure 3.8: Templates where one type is used in multiple places

Is it possible to specify a repeating pattern for types?

Yes. Types are here made of functions which build the required propositional formulas from type names and booleans. Both may be passed around and copied as usual, and together with helper functions like `bitWidth` and `shiftLeft` it is possible to define type templates for pairs, arrays and so on. Figure 3.8 gives three examples: A `pair` is just two of some type, side by side. The more general case of n times the same type in a row can be written as `fixedSizeArray`. An example how arrays of unknown size can be handled is given in the next section—there a type is not "copied" as it is here; we use special types for pointers or array base addresses. `dominoes` is more unusual: This could be used to represent a set of n domino pieces where the requirement that adjoining pieces have the same number of dots on touching ends. For four pieces we would thus only need five places to store the number of dots. A `dominoes (TNNat 3) 4` type would then be four `Nat 3`-pairs, and at the same time, five `Nat 3`s. This structure is shown in Figure 3.9: three of the `Nat 3`s show up twice.

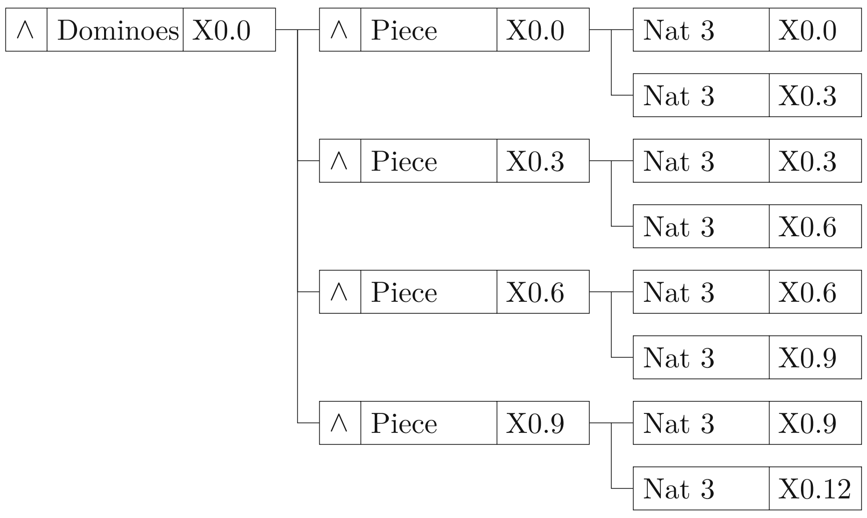


Figure 3.9: Four domino pieces, but only five "ends"

> Can values be "wrapped," i.e. converted to another type without changing the value itself?

Yes. As long as we are content with manipulating type tags, there is no effect during runtime. We can emulate something similar to Haskell's `newtype` or a type cast using combinations of constraints and manipulating the state, see Figure 3.10: `wrap` adds tags but will result in an error if the inner type tags we want to decorate are not present. `unwrap` removes the outer type tags again, but may only be used if both sets of tags are present. One major point is that a constraint asking for the inner type will be met, regardless of whether additional type tags are present. We can add constraints asking for some type tags not to be present, to make sure that only "naked" as opposed to "wrapped" values are accepted. `cast` checks for the presence of one tag, which it then replaces by another one. There could also be a variant of `cast` which removes all tags and then adds only the specified ones. This would be more in line with casts in C, Rust or Zig where only the resulting type needs to be specified. Forgetting "all tags" is slightly complicated by the fact that we do not know the bit width of an unknown type. With `cast` we know which type tags to remove; the bit width of the given set of type tags on individual bits can be looked up. One approach could be to make explicit the size of the units we are working on. For example, `castReg` removes all tags from a single 64-bit register and then transform the register's contents into the one type that is specified. Note that `castReg` does not contain any constraints—differing from the other functions in Figure 3.10; it never fails and is rather unsafe to use.

> And can they be "unwrapped" again?

Yes. We can remove tags at any time. As seen in Figure 3.10, we can pair adding tags—“wrapping”— with removing them again—“unwrapping.”

```
wrap :: TypeName -> TypeName -> BitAddr -> DSLListing
wrap inner outer addr =
  do lf (\pc -> atPC pc $ tagged inner addr)
     postf (\pc instr state -> map (setTypeTag outer addr))

unwrap :: TypeName -> TypeName -> BitAddr -> DSLListing
unwrap inner outer addr =
  do lf (\pc -> atPC pc $ tagged outer addr)
     lf (\pc -> atPC pc $ tagged inner addr)
     postf (\pc instr state -> map (forgetTypeTag outer addr))

cast :: TypeName -> TypeName -> BitAddr -> DSLListing
cast from to addr =
  do lf (\pc -> atPC pc $ tagged from addr)
     postf (\pc instr state -> map (forgetTypeTag from addr))
     postf (\pc instr state -> map (setTypeTag to addr))

castReg :: TypeName -> RegId -> DSLListing
castReg to reg =
  do postf (\pc instr state -> map (forgetAllTagsReg reg))
     postf (\pc instr state -> map (setTypeTag to (RegBit reg 0)))
```

Figure 3.10: Wrapping, unwrapping, and casting types

Is it possible to restrict the types of function parameters and return values?

Yes. Constraints can be attached to any address in memory, which includes every address from which instructions may be executed. The constraints themselves actually all apply to the program as a whole. That is to say, the address in question is included in the constraint. The details are explained in Section 2.6.3. Function parameter types translate to constraints about the contents of registers and memory cells at the first instruction of a function, and return values can be specified as constraints which have to be met at the address of the return instructions of a function.

This concludes the questions regarding what is possible to implement, and the result seems to be a success.

The next set of questions are about ease of use.

Is it enough to specify a combination of name type and value just once?

No. It is at least not quite as simple as specifying a macro in C (which may have deficits in safety) or a typed constant in other languages. While some use cases such as enumerations may be automated with `T`, `makeTypedConstant` and `makeEnum` from Figures 3.2 and 3.3, some flexibility had to be sacrificed. In general, more work is often required to emulate high-level concepts.

> Is the creation of types by combining other types as straightforward as in high-level languages?

No. But this “no” is slightly misleading. In the prototype, the definition of types in general is verbose; in comparison, combining types is relatively straightforward. At least `enum` and `struct` are simple to use, but the verbose nature of the type definitions takes its toll.

> Are “wrapping” and “unwrapping” as simple as in high-level languages?

Yes. Once `wrap`, `unwrap` or `castReg` are defined, there is not much of a difference between code like `@as(type, value);` in Zig and `castReg type (X 0)`—apart from syntax.

> And are they as safe?

No. Unfortunately, we do not get the simpleness and the safety that some languages are able to combine. For example, a `newtype` in Haskell can not be ignored, but has to be handled explicitly. Type tags, on the other hand, can always be ignored, and the tags of a state can always be manipulated.

Summing up, the latter part of the evaluation is not a success. Just as writing assembly code is verbose and more prone to error, the proposed type system, as it stands, requires a lot of effort to produce an executable if we want to use the potential of types.

3.2 Remaining Questions & Discussion

I will now look back at the remaining questions from Section 1.2 and answer them according to the results and insights that could be gained from the experimental implementation.

> How can the type of integer operations be described on the bit level?

In the context of this thesis, there are two answers to this question: On the level of booleans—”regular” bit values 0 and 1—this has long been answered; an addition of two bits a and b gives a result $a \oplus b$ and a carry $a \land b$ and there are similarly simple rules for subtraction, etc; combining these formulas for multi-bit operations leads to larger and more complex expressions but the process itself is simple. Deriving versions for the four-valued `MBit` type is embarrassingly straightforward as well, and those, in turn, combine to operations on multiple `MBits` in just the same way as booleans. This should answer the `MBit` side.

```
addTargetNat16Simple :: BitAddr -> BitAddr -> BitAddr -> BitAddr
  -> State -> State
addTargetNat16Simple target a b _ state =
  if     a has Tag (Nat 16, i) in state
     && b has Tag (Nat 16, j) in state
     && i == j
  then state with added Tag (Nat 16, i) for target
  else state -- no change
```

Figure 3.11: Here, only addition of two `Nat 16`s results in another `Nat 16`

```
addTargetNat16 :: BitAddr -> BitAddr -> BitAddr -> BitAddr
  -> State -> State
addTargetNat16 target a b carryIn state =
  if     a has Tag (Nat 16, i) in state
     && b has Tag (Nat 16, j) in state
     && (carryIn has Tag (NatCarry 16, l) in state
         || carryIn == False)
     && i == j == l
  then state with added Tag (Nat 16, i) for target
  else state -- no change

addCarryNat16 :: BitAddr -> BitAddr -> BitAddr -> BitAddr
  -> State -> State
addCarryNat16 carryOut a b carryIn state =
  if     a has Tag (Nat 16, i) in state
     && b has Tag (Nat 16, j) in state
     && (carryIn has Tag (NatCarry 16, l) in state
         || carryIn == False)
     && i == j == l
  then state with added Tag (NatCarry 16, i + 1) for carryOut
  else state -- no change
```

Figure 3.12: A "stricter" definition of `Nat 16`

For *type tags* however, the answer is much more open; it depends on what the user decides shall be the meaning of an operation, and the level of accuracy that is desired; it is also much more tightly coupled to the example implementation. Figure 3.11 shows examples in Haskell-like pseudocode. As shown here, only `Nats` of a single size, 16 bits wide, are addressed, but a single function could address all sizes of `Nat` at once.

`addTargetNat16Simple` looks for matching `Nat 16` tags; if the indices match, the result is considered a `Nat 16`. This definition only allows addition of two values which are properly aligned.

`addTargetNat16` and `addCarryNat16`—Figure 3.12—together form a stricter version; they demand that the `carryIn` address belongs to a correctly tagged carry bit or is known to be 0. Only then, the correct `Nat 16` and `NatCarry 16` tags are emitted.

Figure 3.13 follows a contrary approach and features a "contagious" `Nat 16` tag where `addTargetNat16Unsafe` is satisfied with either input alone having a `Nat 16` tag. Using more sophisticated functions, one could also implement `Nat n` types which are compatible with other `Nat m` types where $m < n$; this could constitute a kind of automatic promotion to larger types where applicable.

```
addTargetNat16Unsafe :: BitAddr -> BitAddr -> BitAddr -> BitAddr
  -> State -> State
addTargetNat16Unsafe target a b _ state =
  let stateA = if a has Tag (Nat 16, i) in state
               then state with added Tag (Nat 16, i) for target
               else state -- no change
      stateB = if b has Tag (Nat 16, j) in state
               then state with added Tag (Nat 16, j) for target
               else state -- no change
  in stateA `merge` stateB -- combine states
```

Figure 3.13: If one operand is a `Nat 16` the result will be as well

It may seem tedious to have to define the behavior of simple things like addition—and it admittedly is a burden on the user—but then again, we can use this level of control to implement a combination of types like the following: If `t` is a type and n a positive integer, then let `Array t n` be the type of the address of an array holding n values of type t. The default behavior of load and store instructions is not to produce any type tags, so without any implementation by the user, reading from an `Array t n` type address will not produce a `t` value. Assume that, in order to be able to read from or write to such an array, we want the user to first add an appropriate `Offset t n` value, where both type `t` and size n have to be equivalent to those of the `Array t n` base address. Only then do we want to produce a `Pointer t` type, see `addTargetPointer` in Figure 3.14. `Pointer t` misses the size of the two other types, because it is only meant to be used in reading and writing—no pointer arithmetic is implemented. Via `loadPointer` it is finally possible to load a `t` value from memory. There is a third function in Figure 3.14: `addTargetArray` handles the special case that not only the type of the `Offset t n` which is added is known at assembly time, but so is its value. In this case, we may attach additional tags to the resulting pointer. In general, adding an unknown value to an `Array t n`, the only guarantee `Offset t n` gives is that the resulting address points to somewhere in the range of the array. If, on the other hand, the value is known, it is easy to see that the resulting address can be interpreted as the base address of a smaller array: if $m < n$, the m elements after the beginning of an `Array t n` is exactly the first element of an `Array t (n - m)`.

```
addTargetPointer :: BitAddr -> BitAddr -> BitAddr -> BitAddr
  -> State -> State
addTargetPointer target a b _ state =
  if     a has Tag (Array  t  n,  i) in state
     && b has Tag (Offset t' n', j) in state
     && i == j
     && t == t'
     && n == n'
  then state with added Tag (Pointer t, i) for target
  else state -- no change

loadPointer :: BitAddr -> BitAddr -> RegId -> State -> State
loadPointer target src srcAddr state =
  let i = extract index from target
  in if all (\j -> (RegBit srcAddr j) has Tag (Pointer t, j)
                 in state) [0..63]
     then state with added Tag (t, i) for target
     else state -- no change

addTargetArray :: BitAddr -> BitAddr -> BitAddr -> BitAddr
  -> State -> State
addTargetArray target a b _ state =
  if     a has Tag (Array  t  n,  i) in state
     && b has Tag (Offset t' n', j) in state
     && i == j
     && t == t'
     && n == n'
     && register containing b has known value bv -- bv < n
  then state with added Tag (Array t (n - bv), i) for target
  else state -- no change
```

Figure 3.14: A `Pointer` results from an `Array` and an `Offset`; only `Pointers` may be used to load values from memory

Can the type system safely support dense packing of bits into data words?

In this question the key words are "safe" and "dense". When a high-language handles the data layout of structures, the user often does not even have the ability to access fields of product types and structs incorrectly. At the same time, many languages limit the available types to ones of sizes measured in full bytes; automatic addition of padding bytes is also common. As one exception, the Zig programming language gives users more control: Zig features types `u` for unsigned and `i` for signed integers of all sizes up to

65535 bits. Structs in Zig may be *packed*—as explained in Zig's documentation [Zig21]—, meaning no padding bytes or bits are added, booleans are represented by individual bits, and the order of fields remains as specified by the user. My proposed assembler allows the same level of control, albeit at a higher cost. Composite types may specify arbitrary locations, padding or no padding, and fields may even overlap. A stricter approach, more in line with what Zig is offering, might be preferable—even though Zig only seems to offer structs with either an automatically derived layout, or a user-specified gapless sequence of fields, adding, for example, user-specified padding bits is surely possible with just a few dummy fields which serve no other purpose than to take up the right amount of space at the right place. My approach of erring on the side of allowing any and all unorthodox designs does not make for the most readable of type specifications.

```
lsl, lsr, asr, ror :: BitAddr -> State
lsl addr = {- assign 0b00 to addr -}
lsr addr = {- assign 0b01 to addr -}
asr addr = {- assign 0b10 to addr -}
ror addr = {- assign 0b11 to addr -}

shift :: BitAddr -> State
shift = enum [ lsl, lsr, asr ]

shiftOrRotation :: BitAddr -> State
shiftOrRotation = enum [shift, ror]
```

Figure 3.15: Three shift types and rotate right, forming two enumeration types

> Can complex bit level types be translated into a more human readable format; and how can we encode the real-world meaning of bits and their associated legal values in a type system?

Section 2.6.3 introduced annotated states which serve as constraints which have to be met by a program. We may build common kinds of types, like enumerations from singleton types which only allow a single value, or bit pattern. The assignment of bit patterns to members of an enumeration is left to the user; in a way this is the same as assigning names to constants, but if done without overlap—this is again, a disadvantage arising from giving the user perhaps a little too much leeway when designing types—they guarantee that any illegal type raises an error. Customarily, names of an enumeration's members must be distinct; in some cases, they are made distinct via the use of namespaces. No such restriction applies to the proposed types, so we may straightforwardly implement two related enumerations which play a role in Arm A64 assembly as shown in Figure 3.15: There are three types of shift and one type of rotation. They are encoded in two bits. Some instructions only allow for shifting, others allow all four values. For the former, specifying ror—rotate right, 11 in binary—constitutes an illegal opcode. `lsl`, `lsr` and `asr` are legal values of both `shift` and `shiftOrRotation`, `ror` in `shiftOrRotation` is no different from other values; there is no difference between using `shift` in defining `shiftOrRotation` versus listing all options explicitly; there can not be a conflict between, say `lsr` in `shift` and `shiftOrRotation`, because there is only one `lsr`.

Another approach is to attach new meaning to an existing type. Haskell features a straightforward approach using the `newtype` keyword which wraps an existing type using some newly introduced name. We can achieve something similar by introducing a new type tag. We may then either replace existing tags with new ones—which would have a similar effect to Haskell's `newtype`; this would make it necessary to explicitly convert between types where required—or we may just add some tags—no casts necessary, no protection against accidentally "unwrapping" values.

Sometimes it might be enough to have a clear depiction of the structure of a type, like a *tuple* or a simple struct. "(`ID`, `Name`)" may be sufficient, or even preferable to something like `NameLookupEntry`. Just as `struct` in Figure 2.2 simplifies the definition of types, a similar function may automatically produce constraints which are annotated as, for example "ID, Name"—with children "ID" and "Name".

Types and constraints in my proposed type system are verbose, but for some common patterns, the amount of work necessary can be reduced; Section 4.2 will return to this problem.

> How can the type system succinctly describe the combined effects of individual instructions in a code block?

Answering this question is complicated by the very nature of assembly programs. High-level languages put restrictions on, among other things, the ways code blocks may be entered and exited. From strict single-entry single-exit loop bodies, through compromises like loops with breaks, to `goto`— assembly language permits it all. In the world of a machine where advancing to the next instruction means incrementing or setting a global counter, there are no pure functions either. And then there is self-modifying code, which means every statement about part of a program—or even, about any part of any program currently running in the same address space—could need to be augmented with "barring unforeseen circumstances."

Given the right restrictions in a programming language, it may be possible to predict a process' future from the value of the program counter alone. This is not the case with arbitrary assembly programs. "Calling a function"—i.e. jumping to the start of a code block—does not mean that the process will ever return to the calling site. Returning to calling site A in the past does not mean it will return to calling site B, if called from there. We do not even know whether it will return to A again, because we do not know whether or not any instructions have been modified. Without any limitations on what programs may be written, every part of the machine state could potentially change the behavior of a series of instructions. That includes the value of the program counter, as well as the contents of those parts of memory which hold the instructions which are about to be executed.

Splitting a program into smaller units which can then be examined in isolation is a way to manage complexity. A pure function has input, outputs, and nothing else; in a sense, that is often all we have to know about it. If we exclude unorthodox patterns, like using a `goto` to jump from one branch of an `if-then-else` statement to another, we can often make definitive statements about relatively large sequences of instructions, such as "if we enter this branch, all of its instructions will be executed, but no part of the other branch will be." If self-modifying code is prohibited, once a statement about the body of a loop, a branch of a *switch* or `if` statement has been derived correctly, it does not have to be checked again.

With my laissez-faire approach, these individual units may be harder to recognize and they lack the uniform character they might have in other languages. Maybe this function—I use this term here very loosely—can be examined in isolation; maybe there is a group of functions which together can be said to have a type; or maybe the program at hand rewrites itself, such that it is not helpful to speak of functions, and we have to resort to the general and extreme idea that the program we are writing is just a starting state and that its type is a description of all states that may eventually be reached during execution. This is another aspect that the language user would have to take into account and ultimately decide where a line can be drawn between discrete components of a program. This is also a major disadvantage of the proposed method, and for its practical application especially.

In summary, the presented type system, at least in its current iteration, seems esoteric and impracticable compared to others. It does offer the possibility of implementing some unusual—and beneficial?—types, but this may degenerate into implementing custom checks which are lacking when it comes to reusability and flexibility.

On the other hand, the prototype allows the implementation of the following unusual types, which are not possible in some other languages.

- A pair of functions where one may only be called after the other has been called at least once—perhaps the latter performs setup which is necessary for the correct execution of the former. The setup function would attach tags to signal that it may be executed from now on. The other function would include a constraint expressing that its instructions in memory must be tagged correctly when executed.

- A type for instructions whose opcodes may be modified, e.g. to change the registers they are working on. This could be coupled with only certain areas of memory which may hold these instructions. By only allowing modified instructions in special areas, and by only allowing some parts of these instructions to be modified, one may regain some safety which is generally lost if we allow memory locations to be both written and executed.

- More generally, typed memory locations which may only hold values of certain types may be useful, and may be able to replace checks for legal array indices in some situations.
- Bits may be tagged with a unique identifier simply to guarantee that they have not been tampered with.
- A composite type may be defined not to be a single consecutive block of data, but to be distributed to multiple locations.
- Bit patterns permitting, multiple types may be overlapping. Taking a bit to specify part of two values at the same time, would make some combinations impossible. If these impossible combinations are never used, we may just as well use a single bit to represent two bits at once.

Finally, the main research questions can be answered.

> How can a type system handle individual as well as grouped bits and verify that they are processed correctly?

The work of this thesis presents one way to handle types on the bit level. There are certainly others. What characterizes this approach is the combination of the four-valued `MBit` type with type tags which do not have any inherent meaning. This gives the language user a large amount of freedom when giving meaning to type tags, which is counterbalanced by the work required. Another important factor is the way in which the types for individual bits are combined. Ideally, a high-level language offers a few safe and easy to use ways to form new types. The user often does not have a lot of control over how types are combined—for example, in the layout of a struct—but they do not usually need to. It is also likely that only a few "large" types are combined—such as a vector made from three 64-bit integers. When we are handling bits individually however, the situation is very different; a large number of bits are combined, and the bit is the smallest unit available. Additionally, we want to give the user of a typed assembly language the maximum amount of control. The logical $\wedge$ and $\vee$ relationships we use are a simple solution to this problem. If on the other hand, we were to take a few steps towards the approach of high-level languages and make more restrictions on the structure of programs, the result may be easier to use.

CHAPTER 4

Related & Future Work

This chapter goes over related work in Section 4.1, as well as some ideas for what next to explore, sketched in Section 4.2.

4.1 Related Work

This thesis is by far not the first time types were added to assembly languages. My approach does not follow directly from any previous works that I examined. The common goals are of course type systems for "lowest level" languages; the differences lie in things like idealized versus realistic assembly languages or a focus on hand-written code versus translation from another language. Perhaps the most striking divide is that, in my view, the bulk of work is about assembly languages which allow a safe subset of all possible programs to be written—programs which may never "go wrong"—while I—somewhat naively—aim to let the user write whatever assembly code they desire, with (optional) types making for safer programs—with, allowedly, some very impractical results. Nevertheless, the following works can not go unmentioned.

For one, there is a substantial body of work on TIL—typed intermediate languages—and TAL—a typed assembly language—and variations on TAL.

Harper and Morrisett [HM94] use *type directed compilation* in order to avoid some inefficiencies caused by *boxed* data. They use type information to defer the selection of appropriate code—which may be necessary due to non-uniform representations of data—in order to take advantage of more efficient data formats without *boxing* and *tagging*.

In his 1995 thesis, Morrisett [Mor95] describes a *type directed compilation* which is meant to "take advantage of types at all stages," in order to both prove correctness of the compiler and deliver efficient programs. In this compiler from Standard ML to DEC Alpha assembly language, "all but the lowest levels" use *typed intermediate* *languages*, most prominently among them λ_i^{ML}, which features *dynamic type dispatch*. This supports—among other things—the implementation of polymorphism and garbage collection. After alpha-conversion, dead code elimination, closure conversion and other optimizations—via a series of typed intermediate languages—programs are translated into an untyped language, with types being "replaced" by representation information. After another intermediate language—a RISC-style assembly language for an idealized architecture—the result is traditional, untyped DEC Alpha assembly.

The mentioned optimizations are the focus of Tarditi's thesis [Tar96]. In it, optimization techniques which were devised in the context of imperative languages, are adapted to typed intermediate languages which are closer to lambda calculus. One prominent argument is that higher-order functions should be eliminated when possible, instead of—as Tarditi puts it—merely "compiling functions well."

These two works, as well as other collaborations involving Morrisett and Tarditi [Tar+96; Mor+96], are not concerned with typed assembly languages per se, but the typed intermediate languages they describe can be said to be "next of kin" to "real" typed assembly languages, due to their low-level character and the relative simpleness of the remaining transformations to (untyped) assembly language and, eventually, machine language.

Another typed intermediate language is FLINT. Shao [Sha00] argues that intermediate languages are often untyped or do not support some of the higher-order or type features of languages such as ML, Haskell, Scheme, or Java. FLINT's aim is to support these features in a strongly typed environment. It shall further allow optimizations from FLINT to FLINT and to let different languages interact, as well as provide the possibility of shared system-wide garbage collectors and foreign function interfaces. It has some similarities with typed assembly languages, such as integer types in a variety of bit-widths. A comparison is made to Java bytecode and C as a de-facto intermediate language: While Java bytecode is tailored to Java specifically, and C's typesystem is not sufficiently advanced to encode all desired types, the goal for FLINT is to become part of a more generic back-end with support for the different features of multiple modern programming languages.

Morrisett et al. [Mor+99] later describe a basic TAL, along with type-preserving translation from System F. This work extends the approach of typed intermediate languages to all phases of compilation, that is to say, even down to a typed target language. Similar to type systems for high-level languages, TAL provides types for integers, pointers, and tuples. Because operations are permitted only for their appropriate types, TAL programs may get "stuck" if they are not well-typed. This represents a clear distinction to my proposal in which programs cannot get "stuck" during type checking—but may certainly "go wrong" in other ways. In addition, this first TAL is a minimal abstract assembly language and includes a `malloc` instruction for allocating memory. This instruction is not found in conventional assembly languages, and in a practical implementation, is meant to translate into a short sequence of instructions.

Morrisett et al. [Mor+98] extend TAL to STAL, a stack-based typed assembly language. STAL adds `salloc` and `sfree` to allocate and free space on the stack, as well as specialized load and store instructions `sld` and `sst`. Along with the introduction of *stack types*, they make for programs which may again become “stuck”, for example, if a program attempts to free more words than are currently available on the stack. Even though the stack has a unique place in the language, STAL does not stipulate any particular calling convention or other use of the stack.

Building on the ideas behind linear type systems for higher-level languages, Smith, Walker, and Morrisett [SWM00] describe Alias Types. In low-level languages, where for example, the number of registers is limited, it is not possible to maintain the invariant that a linear value is only used once. Registers and memory cells on the stack need to be reused in real-world applications. Aliases are tracked using singleton types and unique names for memory locations. This system is flexible enough to enable some aliasing and destructive updates and to encode stack types which are similar to those of STAL [Mor+98]. Alias Types were added to an implementation of the original TAL and this work was later extended to include recursive data structures by Walker and Morrisett [WM00].

Petersen et al. [Pet+02] take linear logic one step further, and describe an “orderly lambda calculus.” In an ordered context, variables must not only be used exactly once, but also in their fixed order. Their lambda calculus handles low-level allocation and data layouts. It is not a typed assembly language, but their work too is about bridging the gap between low-level memory allocation and type systems.

Speaking of linear type systems, Aspinall and Compagnoni [AC03] manage to provide static guarantees about the heap space usage of programs written in HBAL, the Heap Bounded Assembly Language via linear types: Pointers may not be copied. After loading a pointer from memory, it can not be read a second time. Storing a pointer from a register to memory results in the register value becoming unreadable. HBAL—without extensions—does not include `malloc` and `free` instructions, but in order to facilitate linearity, it includes some other pseudo-instructions which are not present in realistic assembly languages.

With LTAL, Cheney and Morrisett [CM03] explore the expressiveness of a simpler approach to linearly typed assembly languages—but they also point out “just how high the price of linearity can be.” LTAL does away with all pseudo-instructions and does not use a garbage collector, so it is not necessary to gain trust in their implementations. They conclude that—while LTAL is simple, yet expressive—it suffers from inefficiencies due to a lack of sharing data.

Also in an effort to reduce the size of the trusted computing base—one large component would be a trusted garbage collector—Crary, Walker, and Morrisett [CWM99; CWM00] define a typed intermediate language called the Capability Calculus which supports a typed and safe variant of region-based memory management. This is accompanied by the Capability-Based TAL which adds instructions `newrgn` and `freergn`. Similar to `malloc`, they do not translate one-to-one to realistic assembly instructions. Nevertheless, their implementations are claimed to be much simpler than the garbage collectors they are meant to replace, resulting in a smaller trusted computing base.

An extension of a different sort is described by Glew and Morrisett [GM99]. They extend TAL to support the compilation and linking of modules into separate units. Their linker is the typed equivalent of the untyped linkers which are used in contemporary operating systems.

Hornof and Jim [HJ99] describe an unusual feature of a "type safe dialect of C", called Cyclone: Runtime code generation. Cyclone supports function templates with "holes" which may be filled in at runtime to produce specialized versions of the template code. This is translated to yet another extended form of TAL, TAL/T, which adds a few more macro instructions similar to `malloc` in TAL. There are instructions to allocate a fresh memory region, which may then be filled with copies of template code blocks; other instructions fill in holes, finalize a completed specialization or abort the process. For all of these instructions, the type system in TAL/T ensures that code generation is safe, only compatible code fragments are concatenated, and holes are filled with appropriate values. An enhanced approach by Smith et al. [Smi+00] adds an intermediate representation, which allows for data flow analysis and subsequent optimization.

After the original TAL and its extensions, Crary et al. [Cra+99] set out to create a realistic TAL to run on actual hardware. TALx86 is a statically typed version of the Intel IA-32 instruction set. It features extensions described in preceding works, such as stack-allocation and separating programs into modules, as well as types for records, arrays and recursive types. Along with TALx86 they describe a C-based language called Popcorn—the previously mentioned language Cyclone may be considered Popcorn's successor. This language relies on a small runtime including a garbage collector. Even though TALx86 is not tailored to Popcorn specifically, but is more general, it does not support explicit deallocation, and includes a pseudo-instruction `malloc`, which is used in conjunction with the garbage collector. The trusted computing base is still small, as most other instructions are taken directly from the IA-32 instruction set.

In a closely related paper, Grossman and Morrisett [GM00] go into more detail on the implementation of compilation from Popcorn to TALx86. Because Popcorn's calling conventions and exception handling are encoded using TALx86's more primitive types—and not determined by the type system itself—parts of the type annotations in TALx86 are repeated throughout a typical program. The size of these annotations can grow rapidly, and some ways to control and reduce their size are presented.

TALx86 may be the closest relative to my approach; they both suffer from large type annotations, which TALx86 manages to get under control; they both hold back on providing fixed types and conventions; both are in one-to-one relations to instruction sets which are implemented in hardware—with `malloc` being TALx86's single exception. There are important differences as well; the most primitive types in TALx86 are not bits but rather types for commonly sized integers, or arbitrary values of certain sizes—for example, that of 32-bit registers; TALx86 is part of larger project and seems somewhat mature, my assembler stands alone and is only a first experiment; but perhaps most importantly, with my assembler I did not find a way to “close the holes” in the type system—alas, safety can always be sidestepped.

DTAL, described by Xi and Harper [XH01], is an extension of TAL with a limited form of dependent types. DTAL features singleton integer types and array types, which include not only element types but also array size, to enable the elimination of runtime array bound checks, which were mandatory for early TAL versions. Certain checks may also be moved out of loops to further optimize performance without any loss of safety. DTAL and its type annotations are again designed more to be generated by a compiler, than written by hand.

Another dependently typed assembly language is Singleton, presented by Winwood and Chakravarty [WC11]. One might say Singleton goes a step further than DTAL, with complex types closer to those of high-level dependently typed languages like Idris, proofs, and multiple pseudo-instructions which handle these types and values.

Necula and Lee [NL96] describe the virtues of Proof-Carrying Code, and later, they [NL98] advocate a rivaling approach to the aforementioned TIL and TAL by Tarditi, Morrisett and their colleagues. Their Certifying Compiler produces type specifications along with annotated assembly language code. Their input language is a safe subset of C, with a few exceptions: Arrays are bundled with their length and unsafe features like casts and pointer arithmetic are excluded as well. The certifier part of their project then takes both outputs produced by the compiler and checks the type safety and memory safety of the resulting program. Their approach delivers improved safety over C, with competitive performance through some optimizations—the most prominent among them being array bounds-checking elimination.

Seeking to remove some restrictions of this approach, Appel and Felty [AF00] devise a more general approach with support for more types. By means of a small imaginary machine instruction set, they demonstrate this approach, which allows the representation of a *safety policy* as a set of axioms. Appel [App01] goes on to suggest *Foundational Proof-Carrying Code* or FPCC, which—similar to a foundational proof “from just the foundations of mathematical logic”—is PCC where the trusted base is made as small as possible.

Crary and Vanderwaart [CV02] claim that PCC has two advantages over type-theoretic approaches like TAL: greater expressive power and scalability “from without the system,” that is to say, when extending TAL one needs to extend the type system itself; in PCC system, on the other hand, the logic used to express proofs does not change, rather, extensions are written for the same “old” proof checker. By incorporating logical propositions and proofs into the type system, they aim to transfer these two advantages to their novel type system. Although they do not use an assembly language, and opt instead for a high-level “core language,” this approach should in principle be applicable to other targets—such as realistic assembly languages.

Similarly, Shao et al. [Sha+02] develop a typed intermediate language which is able to handle propositions and proofs. Special attention is given to programs with side effects. Their system includes dependent types but they mitigate some problems which could arise, by introducing a separation between the *type language* and the *computation language*; the former may not depend on the latter.

Because foundational proofs can be hard to construct, Hamid et al. [Ham+03] try a syntactic, rather than semantic, approach to FPCC. They define FTAL—Featherweight TAL—which is then translated to FPCC. FTAL is idealized, as is TAL, but it does not contain pseudo or "macro" instructions like `malloc`. Crary [Cra03] applies this model to the IA-32 instruction set. The motivation behind this is to avoid some of the shortcomings of the previously mentioned realistic TAL implementation, TALx86. Using FPCC, typing rules are moved out of the trusted base and a realistic typed assembly language may rely on rigorous proofs, rather than analogies to a different, idealized TAL.

Chen et al. [Che+03] describe yet another FPCC-based method. An almost complete core of ML is translated to LTAL, running on Sparc processors. LTAL is thus not an idealized, but rather realistic language and its soundness proof is machine checkable.

Typed assembly languages may also be used in the context of security and confidentiality. Bonelli, Compagnoni, and Medel [BCM04] describe SIFTAL—the Secure Information Flow Typed Assembly Language—which guarantees non-interference in a low-level language. In the case of software which handles information of multiple security levels—from low to high—it is undesirable for information to leak from a high security level to a lower one. *Non-interference* guarantees that the computation of low level data can not be influenced by high level data. The well-behaved control flow constructs of high-level languages—`if-then-else` rather than `goto`—help in preserving this property. In assembly languages these "safer" constructs are nowhere to be found. A further problem is the discrepancy between a more or less unlimited number of variables and the very limited number of registers found on hardware. If one has to reuse registers for different variables during the life of a process, it is very likely that a single register will be used for variables of different security levels. SIFTAL features a stack of linear continuations, along with `cpush` and `jmpcc`—pop and jump to—instructions; its type system does not allow high level security information to leak, even in the advent of register reuse. Medel, Compagnoni, and Bonelli [MCB05] soon improved on SIFTAL; in SIF, push and pop operations on the continuation stack are now removed during assembling, and transformed to regular jump instructions, respectively.

There is a similar approach and low-level language by Barthe, Basu, and Rezk [BBR05]. Their language includes operations on a stack, conditional and unconditional jumps and procedure calls. Each conditional jump defines a *control dependence region* which is, roughly, the sum of instructions which are executed under its control condition. This notion may be more intuitive in a high-level language; for an `if-then-else` construct this refers to the instructions inside both branches; for a `while` loop it is the body of the loop. Instructions inside a conditional jump's region are then not allowed to write to low security locations if the conditional included high security locations. Their work includes a high-level language which is then compiled to their low-level language, including appropriate typing information to translate security constraints from one language/type system to the other; this language uses the two mentioned high-level constructs: `if-then-else` and `while`.

Yu and Islam [YI06] describe yet another approach, which is similar but also compatible with TAL as described by Morrisett et al. [Mor+99]. They "restore" some of the high-level structures, which are easier to reason about, using type annotations in an assembly language. A program may be envisioned to run in different security contexts; switching between them is implemented using *raise* and *lower* instructions. High security locations can only be used after raising the security context accordingly. After lowering—which may for example, coincide with leaving the branches of an `if-then-else`—it is again possible to write to low security locations.

Maeda [Mae05]—in his 2005 thesis and other works [MY05; MY09] in collaboration with Yonezawa—focuses on one specific but important application of typed assembly languages, an operating system kernel. The language designed for this task, called TALK, features variable-length arrays to implement memory management primitives. To this end, alias types and dependent types—and more specifically singleton types—are included; they are similar to previously mentioned works [XH01; SWM00]. TALK is, again, an idealized RISC-style assembly language, but there is also a realistic implementation for the IA-32 instruction set. The resulting OS kernel has a `malloc` and `free` style of memory management and supports multi-threading on a single CPU. An improvement by Maeda and Yonezawa [MY10] adds support for SMP multi-core and CPU hardware interrupts. Atomic instructions like the IA-32 `xchg` instruction—exchange the values of two registers or a register and a memory location—are implemented using `block` and `unblock` instructions in an extended TAL. Using `xchg` it is possible to implement spin locks, but other atomic operations can be implemented using `block` and `unblock`, and other methods of process synchronization may then be implemented using those.

A typed intermediate language for object oriented programming languages is described by Chen and Tarditi [CT05]. Classes and objects of high-level OOP languages are translated to the simpler, lightweight type system of the "Low-level Intermediate Language with Classes", or LIL$_C$. The language includes support for *records*, *vtables*, *type tags* and instructions which allow testing of these tags and branching accordingly. Building in part on LIL$_C$, Tate, Chen, and Hawblitzel [TCH10] argue that, rather than having types at every level of a compilation process—along with transformations between them—one could also use a TAL featuring a type inference algorithm, such as iTalX. This way, optimizations can be applied as usual, even without typing information; as long as the resulting assembly program can be typed using iTalX, one does not have to trust the compilation phases applied previously. LIL$_C$ was later used again as part of a "large-scale optimizing compiler" by Chen et al. [Che+08]. This work managed to produce marginally slower programs, compared to untyped assembly. The Bartok compiler used in this project was also used in the Singularity project, which also included some experiments on typed assembly languages—see for example Hunt and Larus [HL07].

Hawblitzel and Petrank [HP09] address a vulnerability of PCC or TAL approaches which rely on garbage collection; if the garbage collector is external to the certified code, then bugs in its code could compromise the safety of the whole system. It is thus desirable to prove the safety and correctness of the garbage collector. The two garbage collectors they present work in conjunction with the optimizing compiler Bartok and support a realistic object model, vtables, arrays, stacks, and others. Performance of their verified garbage collectors—when compared to Bartok's standard options—seems to suffer slightly for some combinations of benchmark and heap size, but is "in the same ballpark" in most cases, sometimes even outperforming its non-verified competition.

Via a combination of approaches, Yang and Hawblitzel [YH10] build another operating system that is verified to be safe. This OS, Verve, is made up of two layers: the lower level Nucleus is written in untyped assembly, that is annotated with assertions representing preconditions, postconditions and loop invariants. It handles allocation, garbage collection—taken from previous work [HP09]—, stacks and interrupts; the upper layer of the kernel is written in C# and then translated to TAL. This code builds higher-level abstractions on top of the Nucleus, such as preemptive threads. Applications for Verve are produced the same way. Annotations for the Nucleus are written by hand, but verification of the TAL portion is automatic. There is no support for exceptions and Verve is limited to running on one processor core, but it does run on real x86 hardware. One goal of this work was to demonstrate that only a small part of the operating system can not be implemented using TAL; only those parts, which require the more laborious annotations, were moved into the *Nucleus*.

It may be considered tangential to the main topic of this thesis, but work by Michael and Appel [MA00] encodes machine instructions and syntax in higher-order logic. What they call the "natural factoring" of a CPU architecture contains similar ideas to my way of representing an instruction set. In my case, this came about almost incidentally; their work uses higher-order logic and is presented in much more formal rigor.

Finally, in his 2019 dissertation, Bowman [Bow19] takes a look at the theory behind the full process of typed compilation—in this case, the compilation of dependently typed languages. He argues that it is theoretically possible—but has never been implemented—to transform a dependently typed program all the way down to dependently typed assembly—without ever dropping types along the way. This should eliminate miscompilation and linking errors, which are not fully handled in contemporary implementations. His source language is close to Coq, but does not include its features in full; further, the target of his compilation is not a dependently typed assembly language. Nevertheless, if put into practice, his approach should be able to deliver the first ever fully verified program, in assembly, to execute on real hardware—because so far, as he claims, at least some part of the "verified-ness" of a program goes missing somewhere along the way.

4.2 Future Work

The type system as described is not a real competitor to established programming languages. How could we derive a viable programming language from the presented method? Let me take a look at its shortcomings, as they provide pointers to some options for future work.

One problem is the amount of work required to define useful types. One way forward could be to develop a larger library of pre-made types, or rather, a toolset to make construction of concrete types easier. The construction of types and their corresponding constraints would have to be automated; these constraints would have to be placed at the relevant locations of a program without much involvement by the user. For example, constraints for integer types should be generously applied to whole blocks of code, such that errors could be caught right where they originate. This seems to call for strategies like being able to fix a register's type over a sequence of instructions. Finding a way to strike the right balance between still allowing any program to be written, while at the same time making it easier to write "conventional" programs, seems to be crucial.

By introducing some restrictions on what programs can do, the system would become easier to use and allow for more optimizations. If the modification of instructions in memory was ruled out, instructions could be removed from abstract states; the registers affected by instructions would only have to be determined once; a large part of the control flow graph would be known from the outset. If access to the standard return address register was denied, the behavior of functions could not depend on the caller's address; this would help bring a clearer structure to programs.

Another limitation are the formulas used in the current implementation: Some things are hard to express using propositional formulas alone. Performance is already questionable with the current simpler system, but it could be worthwhile to explore a similar approach, using first-order logic.

During the course of writing this thesis, I came to wonder whether, rather than writing a program in assembly language, one could construct a program, similar to how one would construct any other data structure, by using a powerful set of tools or libraries. No doubt, something similar is being done in some compilers. For example, in a very broad sense, optimizing nested if statements and balancing a tree, are both instances of "optimizing the structure around integral parts without changing the meaning of the whole." Algorithms on programs are usually applied without user intervention and part of this idea of "constructing programs" would be to put these tools into the hands of users. This approach would be for situations where extremely precise control is needed. In general, a modern compiler will make good decisions in how to, for example, precisely lay out a large number of nested branches, but there may be some situations in which one would like to give preferential treatment to some branches, even if that may not generate the best solution overall. Alternatively this could be implemented using priority annotations which are added to *then* and *else* branches, but using a general purpose programming language for the implementation of arbitrary "program construction strategies" may be less work in the long run—if compared to a large set of specialized language features, which then only apply to rare use cases, like the one sketched previously.

If my current approach can be said to be "bottom up"—starting with types for true and false bits, having the user build larger types from these very simple primitives—maybe a "top down" approach would give better results—design a type system similar to one for a high-level language to which special low-level features are then added: Zig-like structs to which special features like custom padding bits may be added; the ability to overload instructions, allowing them to be used for composite types; sensible default behavior with the option to opt out, among other things. Instead of "building up" to a useful set of types, design and expand a constrained language "down" as much as is necessary for the level of control that is needed. The question remains how much low-level bit manipulation is actually useful and necessary.

The types for bits of unknown or known value, combined with a convenient syntax for large structures—as well as the possibility to "flatten" structs of structs—may be enough to transfer the `MBit` type to a traditional type system. Most certainly, there are also more straightforward ways to support partially overlapping enumeration types, such that one does not end up with multiple type constructors for the same value.

It should be reasonably straightforward to implement the simple iterative process as a parallel program; it does not matter in which order successor states are calculated; the central task would be to ensure correct merging of sets of new states and a simple work scheduling algorithm. Predecessor and successor states could also be cached to disk to speed up an iterative development process.

Another way to improve or simplify the implementation could be to have the abstract machine be an implementation, not of the actual machine one is interested in, but rather one that executes the smaller set of instructions which are used to build a realistic instruction set. In the end, this amounts to little more than interpreting a program to be written not in, say, A64 assembly, but another language, which just so happens to have an array of macros defined which neatly coincide with A64 instructions.

CHAPTER 5
Conclusion

This thesis looked at a simple type system for assembly languages, with a goal of not making too many assumptions about the programs that are to be written. Via the implementation of a prototype, it was shown that a very simple system can indeed be used to express the properties, which make up common type system features, like enumerations, structures or functions which have fixed input and output types. A few uncommon type features were also achieved. In relation to other typed assembly languages, this is possibly the most prominent difference: While many of the type systems of related works which are mentioned in Section 4.1 present a familiar set of features and tools, my approach tries not to make any such decisions in the type system itself. As mentioned previously, this can be advantageous if we want to implement special purpose types. But there are significant disadvantages as well: Types are very verbose, and features which we usually take for granted have to be implemented by the user. It is also more work to actually benefit from the safety which types are meant to deliver; there are many ways in which the ad hoc implementation of a feature may subtly go wrong; a certain amount of discipline is required, similar to programming in an untyped language.

Thinking back to the evaluation in Chapter 3, it is my conclusion that the presented method should count as a success in regards to what can be achieved with it. When, on the other hand, we look at how practical it is to write a non-trivial program in this typed assembly language, the result is quite discouraging. Much time is spent handling the type system itself, translating ideas into large statements, or extracting common patterns in order to reduce boilerplate code. And even though I did not accurately measure the performance of the implementation, I think that, one would quickly grow impatient, when waiting for the assembler to finish type checking.

For future work in this area, it is probably more promising to pick a TAL from one of the related works as a starting point, and to then add more flexibility where needed, rather than trying to tame the unwieldy beast that is my prototype.

List of Figures

Bibliography

[AC03] David Aspinall and Adriana Compagnoni. “Heap-bounded assembly language”. In: *Journal of automated reasoning* 31.3-4 (2003), pp. 261–302.

[AF00] Andrew W Appel and Amy P Felty. “A semantic model of types and machine instructions for proof-carrying code”. In: *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 2000, pp. 243–253.

[App01] Andrew W Appel. “Foundational proof-carrying code”. In: *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*. IEEE. 2001, pp. 247–256.

[Arm18] *Arm Instruction Set Reference Guide*. Version 1.0. Arm Limited. 2018.

[Arm20] *Arm Architecture Reference Manual. Armv8, for Armv8-A architecture profile*. Arm Limited. 2020.

[BBR05] Gilles Barthe, Amitabh Basu, and Tamara Rezk. *Security types preserving compilation*. 2005.

[BCM04] Eduardo Bonelli, Adriana Compagnoni, and Ricardo Medel. “SIFTAL: A typed assembly language for secure information flow analysis”. In: *Informal Proceedings of FCS* 5 (2004).

[Bow19] William J Bowman. “Compiling with Dependent Types”. PhD thesis. Northeastern University, 2019.

[Bra17] Edwin Brady. *Type-driven development with Idris*. Manning Publications Company New York, 2017.

[Che+03] Juan Chen et al. “A provably sound TAL for back-end optimization”. In: *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*. 2003, pp. 208–219.

[Che+08] Juan Chen et al. “Type-preserving compilation for large-scale optimizing object-oriented compilers”. In: *ACM SIGPLAN Notices* 43.6 (2008), pp. 183–192.

[CM03] James Cheney and Greg Morrisett. *A linearly typed assembly language*. Tech. rep. Cornell University, 2003.

[Coq21] *Coq 8.13.0 documentation*. URL: `https://coq.inria.fr/distrib/current/refman/`. [accessed 18th February, 2021].

[Cra+99] K Crary et al. "TALx86: A realistic typed assembly language". In: *1999 ACM SIGPLAN Workshop on Compiler Support for System Software Atlanta, GA, USA*. 1999, pp. 25–35.

[Cra03] Karl Crary. "Toward a foundational typed assembly language". In: *ACM SIGPLAN Notices* 38.1 (2003), pp. 198–212.

[CT05] Juan Chen and David Tarditi. "A simple typed intermediate language for object-oriented languages". In: *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 2005, pp. 38–49.

[CV02] Karl Crary and Joseph C Vanderwaart. "An expressive, scalable type theory for certified code". In: *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*. 2002, pp. 191–205.

[CWM00] Karl Crary, David Walker, and Greg Morrisett. *Typed Memory Management in a Calculus of Capabilities*. Tech. rep. Cornell University and Carnegie Mellon University, 2000.

[CWM99] Karl Crary, David Walker, and Greg Morrisett. "Typed Memory Management in a Calculus of Capabilities". In: *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1999, pp. 262–275.

[Dol+20] Damien Doligez et al. *The OCaml system, release 4.11*. 2020. URL: `http://caml.inria.fr/pub/docs/manual-ocaml/`. [accessed 18th February, 2021].

[F#21] *F# Software Foundation*. URL: `https://fsharp.org/`. [accessed 18th February, 2021].

[GM00] Dan Grossman and Greg Morrisett. *Scalable Certification of Native Code: Experience from Compiling to TALx86*. Tech. rep. Cornell University, 2000.

[GM99] Neal Glew and Greg Morrisett. "Type-safe linking and modular assembly language". In: *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1999, pp. 250–261.

[Gos+20] James Gosling et al. *The Java Language Specification*. 2020. URL: `https://docs.oracle.com/javase/specs/jls/se15/html/index.html`. [accessed 17th February, 2021].

[Ham+03] Nadeem A Hamid et al. "A syntactic approach to foundational proof-carrying code". In: *Journal of Automated Reasoning* 31.3-4 (2003), pp. 191–229.

[HJ99] Luke Hornof and Trevor Jim. "Certifying compilation and run-time code generation". In: *Higher-Order and Symbolic Computation* 12.4 (1999), pp. 337–375.

[HL07] Galen C Hunt and James R Larus. “Singularity: rethinking the software stack”. In: *ACM SIGOPS Operating Systems Review* 41.2 (2007), pp. 37–49.

[HM94] Robert Harper and Greg Morrisett. “Compiling polymorphism using intensional type analysis”. In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1994, pp. 130–141.

[HP09] Chris Hawblitzel and Erez Petrank. “Automated verification of practical garbage collectors”. In: *ACM SIGPLAN Notices* 44.1 (2009), pp. 441–453.

[HP12] John L Hennessy and David A Patterson. *Computer Architecture: A Quantitative Approach*. Fifth. Elsevier, 2012.

[Jon03] Simon Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[KN19] Steve Klabnik and Carol Nichols. *The Rust Programming Language (Covers Rust 2018)*. No Starch Press, 2019.

[KR88] Brian W Kernighan and Dennis M Ritchie. *The C programming language*. Second. Prentice Hall, 1988.

[MA00] Neophytos G Michael and Andrew W Appel. “Machine instruction syntax and semantics in higher order logic”. In: *International Conference on Automated Deduction*. Springer. 2000, pp. 7–24.

[Mae05] Toshiyuki Maeda. “Writing an Operating System with a Strictly Typed Assembly Language”. PhD thesis. School of Information Science and Technology, The University of Tokyo, 2005.

[MCB05] Ricardo Medel, Adriana Compagnoni, and Eduardo Bonelli. “Non-interference for a typed assembly language”. In: *Foundations of Computer Security*. Citeseer. 2005, p. 67.

[Mil+97] Robin Milner et al. *The Definition of Standard ML: Revised*. MIT Press, 1997.

[Mor+96] Greg Morrisett et al. “The TIL/ML compiler: Performance and safety through types”. In: *1996 Workshop on Compiler Support for Systems Software*. Vol. 7. 1996.

[Mor+98] Greg Morrisett et al. “Stack-based typed assembly language”. In: *International Workshop on Types in Compilation*. Springer. 1998, pp. 28–52.

[Mor+99] Greg Morrisett et al. “From System F to typed assembly language”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 21.3 (1999), pp. 527–568.

[Mor95] Greg Morrisett. “Compiling with Types”. PhD thesis. School of Computer Science, Carnegie Mellon University, 1995.

[MY05] Toshiyuki Maeda and Akinori Yonezawa. *Writing practical memory management code with a strictly typed assembly language (Extended Version)*. 2005.

[MY09] Toshiyuki Maeda and Akinori Yonezawa. “Writing an OS kernel in a strictly and statically typed language”. In: *Formal to Practical Security*. Springer, 2009, pp. 181–197.

[MY10] Toshiyuki Maeda and Akinori Yonezawa. “Typed Assembly Language for Implementing OS Kernels in SMP/Multi-Core Environments with Interrupts.” In: *SSV*. 2010.

[NL96] George C Necula and Peter Lee. *Proof-carrying code*. 1996.

[NL98] George C Necula and Peter Lee. “The design and implementation of a certifying compiler”. In: *ACM SIGPLAN Notices* 33.5 (1998), pp. 333–344.

[Pet+02] Leaf Petersen et al. *A Type Theory for Memory Allocation and Data Layout (Extended Version)*. 2002.

[Sha+02] Zhong Shao et al. “A type system for certified binaries”. In: *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 2002, pp. 217–232.

[Sha00] Zhong Shao. “Typed common intermediate format”. In: *ACM SIGSOFT Software Engineering Notes* 25.1 (2000), p. 82.

[Smi+00] Frederick Smith et al. *Compiling for runtime code generation (extended version)*. Tech. rep. Cornell University, 2000.

[Str13] Bjarne Stroustrup. *The C++ programming language*. Fourth. Pearson Education, 2013.

[SWM00] Frederick Smith, David Walker, and Greg Morrisett. “Alias types”. In: *European Symposium on Programming*. Springer. 2000, pp. 366–381.

[Tar+96] David Tarditi et al. “TIL: A type-directed optimizing compiler for ML”. In: *ACM Sigplan Notices* 31.5 (1996), pp. 181–192.

[Tar96] David Tarditi. “Design and Implementation of Code Optimizations for a Type-Directed Compiler for Standard ML”. PhD thesis. School of Computer Science, Carnegie Mellon University, 1996.

[TCH10] Ross Tate, Juan Chen, and Chris Hawblitzel. “Inferable object-oriented typed assembly language”. In: *ACM Sigplan Notices* 45.6 (2010), pp. 424–435.

[WC11] Simon Winwood and Manuel Chakravarty. “Singleton: A general-purpose dependently-typed assembly language”. In: *Proceedings of the 7th ACM SIGPLAN workshop on Types in language design and implementation*. 2011, pp. 3–14.

[WM00] David Walker and Greg Morrisett. “Alias types for recursive data structures”. In: *International Workshop on Types in Compilation*. Springer. 2000, pp. 177–206.

[XH01] Hongwei Xi and Robert Harper. “A dependently typed assembly language”. In: *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*. 2001, pp. 169–180.

[YH10] Jean Yang and Chris Hawblitzel. “Safe to the last instruction: automated verification of a type-safe operating system”. In: *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation.* 2010, pp. 99–110.

[YI06] Dachuan Yu and Nayeem Islam. “A typed assembly language for confidentiality”. In: *European Symposium on Programming.* Springer. 2006, pp. 162–179.

[Zig21] *Documentation—The Zig Programming Language.* URL: `https://ziglang.org/documentation/0.7.1/`. [accessed 17th February, 2021].