FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

bc. Jan Dubský

Extensible disassembler with support for interactive instruction reordering

Katedra distribuovaných a spolehlivých systémů

Supervisor of the master thesis:	doc. Ing. Lubomír Bulej, Ph.D.
Study programme:	Informatika
Study branch:	Softwarové systémy

Prague 2022

I’d love to say thank you to doc. Ing. Lubomír Bulej, Ph.D. for both proposing the topic of this thesis as well for all his patience and support along the way.

Title: Extensible disassembler with support for interactive instruction reordering

Author: bc. Jan Dubský

Department: Katedra distribuovaných a spolehlivých systémů

Supervisor: doc. Ing. Lubomír Bulej, Ph.D., Katedra distribuovaných a spolehlivých systémů

Abstract: Machine code disassembling is a process of transforming binary machine code into assembly code. The main purpose of this process is to help people to understand the purpose of the program without knowing its source code. Unfortunately, the machine code produced by compilers is quite hard to read due to numerous optimizations applied to it. One substantially problematic optimization is instruction scheduling which mangles instruction order to increase final performance.

The goal of this thesis is to implement a disassembler capable of reordering individual instructions. This would allow the user to restructure the machine code into a more readable form. To provide such functionality, the disassembler has to be able to understand the meaning of machine code instructions. For this reason, we will design a platform-independent internal representation of machine code and we will translate any machine code into it. This representation will be then used to analyze dependencies between instructions which can be further used in instruction reordering algorithm. At the very end, we will discuss the possibility of platform-independent program emulation based on internal disassembler representation.

Keywords: disassembler, machine code, reordering, emulation, machine code analysis

Contents

Introduction

Disassembling of machine code is a process of converting binary machine code instructions into an assembly language. Besides code visualization functionality, many dissemblers provide also some more advanced features for example address naming or basic block analysis. The most advanced disassemblers then have features of decompilers which allow them to lift the machine code into more high-level language, for example C.

The problem with existing disassemblers is that even though they usually work very well for hand-written code, they are much less useful for code generated by a compiler. The reason is that people are quite good at understanding programs written by other people. But people are much worse at understanding complex sequences of instructions which is optimized not to be readable, but to be executed as fast as possible on a given CPU.

One of the common compiler optimizations that significantly decreases readability of the assembly code is instruction scheduling [1]. This optimization is supposed to achieve higher instruction-level parallelism by interleaving independent calculations. In other words, instruction scheduling mixes multiple sequential algorithms into one sequence of instructions. The downside is that when human attempts to read the code, he or she sees many unrelated instructions following one another. This makes it quite hard to understand the overall meaning of the code.

The goal of this thesis is to write a disassembler that would allow the user to manually revert the effects of instruction scheduling. In other words, we would like to change the order of instruction in the code to a format where individual algorithms are written in a sequence. Such a code would be naturally much slower to execute, but it would allow humans to understand the purpose of the code much better. Which is in the end the ultimate goal of human-assisted disassembling.

A natural requirement on any tool which allows us to change instruction order is to guarantee that the meaning of the code will be preserved. To be able to guarantee such a property, the disassembler will have to understand the meaning and side-effects of instructions. Based on this understanding, the disassembler will be able to deduce dependencies between instructions and those dependencies can be used to guarantee equality of the restructured code.

As the disassembler will understand the meaning of instructions, we should also be able to use this knowledge to emulate the code execution. This feature of the disassembler could help the user to further understand the code purpose and all its impact. This property would be a nice extension that would differentiate the disassembler from existing ones.

Goals of the Thesis

The primary goal is to write a disassembler allowing the user to reorder instructions without change of the program meaning. The other goal is to allow to emulate the execution of the program instruction by instruction. The disassembler will demonstrate such functionalities for the RISC-V machine code.

Internally, the disassembler will use a representation of instruction meaning which will not be dependent on any CPU architecture. This will allow us to trivially extend its functionality to other machine codes as well.

This goal is achievable either by implementing our own disassembler or by reusing one of existing disassemblers. As we will see later, existing disassemblers would not give the functionality we would need. So instead of rewriting an existing disassembler and its internals, we will write the disassembler ourselves.

1. Analysis & Design

The goal of understanding instruction meaning, as well as the possibility of emulation, introduces many questions which we need to both ask and answer in this thesis. In this chapter, we will discuss some of the main questions and we will sketch possible answers. Those sketches will then outline the overall structure of the thesis.

1.1 Input of the disassembler

The very first question we have to ask about our disassembler is what will be its input. A natural answer to that question is that disassemblers always accept some sort of binary file. But as there are many different kinds of binary files, this statement requires further specification.

The first differentiation in binary files worth discussing is the difference between statically and dynamically linked binaries [2]. Statically linked binaries contain all machine code they use in a single binary file. The word all in this context does not refer just to the code of the program itself, but also to the code of a language runtime and libraries for example *libc*. We could also say that statically linked binary contains all the code it needs to run.

The idea behind dynamic linking is that the vast majority of machine code used by binaries is in reality shared with many other binaries in a single system. This shared code can then be separated into so-called dynamically-linked libraries (DLLs) and DLLs can be distributed independently on programs using them. This allows the code of a library to be present only once in the whole system rather than with every single binary using it. When a dynamically linked binary starts, dynamic linking and loading take care of adding all shared libraries the program needs to its memory address space. The idea of dynamic linking is designed mostly to reduce the binary size as well as program loading time and its overall memory footprint. The cost for this optimization is all the complexity related to dynamic linking and loading.

As an emulation of dynamic linking and loading is out of the scope of this thesis, our disassembler will not special-case dynamic libraries in any way. On one hand, this means that the disassembler will be able to fully analyze only statically linked binaries. On the other hand, even though dynamic binaries are not explicitly supported, they are also not prohibited in any way. As dynamic binaries still contain some machine code, the disassembler will be able to analyze this code. The only problem user of our disassembler might encounter while analyzing dynamically linked binaries is that some features of the disassembler might not work 100% well. This can happen because some analyses might miss the code located in dynamically linked libraries.

Besides statically and dynamically linked binaries, executable file formats also differ for every operating system. Consequently, another question to ask about the disassembler input is which operating system binaries will we support. As parsing of executables is not the goal of this thesis, we will limit possible inputs to Linux ELF files. Given that all executable formats used nowadays are conceptually very similar, possible extensions to parse different executable formats should not be a complex task in the future.

1.2 Instruction representation

As we already mentioned, we will need to make our disassembler understand the meaning of the instructions. The very first idea one might have is that we will analyze the raw machine code and we just make the disassembler aware of all its effects and possible interactions. In this section, we will discuss why is it not a good idea and what can we use instead.

There are many problems related to raw machine code analysis. None of those problems makes machine code analysis impossible. We just point them out to explain why we want to use a different approach. The first problem is the machine code complexity and irregularity of individual instructions. In other words, many machine code instructions have some special effects or an unusual meaning. This is problematic as we would have to take each of those special effects into account and special-case it in our disassembler. The second reason why machine code is not good for any advanced analysis is that there are simply too many pairs of possible instruction interactions. Any two semantically different instructions can interact together in a unique way and we would have to special-case all those possible interactions. Not mentioning sequences of instructions and their interactions within that sequences. The third problem with the analysis of machine code is that we would make our disassembler specific to a given CPU architecture. We do not say that writing a disassembler specialized for only one CPU architecture is bad in any way. On the other hand, we can make the disassembler CPU agnostic which will make it more useful and much more extensible in the future.

Instead of processing any specific machine code, we design the disassembler to reflect the architecture of existing compilers. Transformation of source code into machine code done by compilers usually comprises several stages of processing. But with a bit of simplification, we can say that compilation always has 3 main parts. The first part is the so-called frontend which converts the source code into an internal representation of the compiler. In compilers, such internal representation is usually called intermediate representation (IR). The intermediate representation is an encoding of instructions that is specific to a given compiler, but which does not depend on the target CPU architecture in any way. The other stage of the compilation then applies several transformations to the intermediate representation. The purpose of those transformations is typically to make the final code faster, smaller, and more effective. Only the last stage of compilation (so-called backend) then transforms intermediate representation into platform-specific machine code. Compilers are usually designed in a way that both frontends and backends are pluggable. In other words, the optimization logic forms the core of the compiler which can be extended for multiple programming languages as well as for multiple target CPU architectures.

The existence of intermediate representation allows the core of the compiler (i.e. the transformation stage) to be fully independent of both the programming language and the target CPU architecture. This is especially useful given the fact that this part of the compiler is typically the most complex one. Moreover, transformations in the compiler do not have to care about irregularities of the target machine code. Instead, they operate on a significantly simpler and much more regular intermediate representation which makes them significantly simpler to implement. Overall, the existence of intermediate representation abstracts away most hardware and programming language-specific problems.

We can use a similar approach in the design of the disassembler. In other words, we can introduce our own internal representation of machine code and use is instead of raw machine code. This design choice will on one hand force us to write a logic transforming machine code instructions into the internal representation of the disassembler first. On the other hand, all analysis of the code meaning will be then much simpler to perform on top of the internal representation rather than on top of the machine code for many reasons mentioned above. Another advantage of this approach is that we can make our disassembler capable of processing different machine codes by implementing multiple machine code translation logics without the need to modify other parts of the disassembler. In the end, even though the introduction of our own internal instruction representation will cost us some additional effort, it will also lower the complexity of both the instruction reordering and emulation logic. Independence of those logics on any specific CPU architecture is then a welcome feature.

To sum our design choice up, we know that we will first design our internal code representation. Once we have it, we will write a parser for a given CPU architecture translating the machine code into the internal representation. All later stages of the disassembler including both instruction reordering and emulation will operate on exclusively top of this internal representation.

1.2.1 Reusing of existing disassemblers

One would hope that there already exists a disassembler that uses such a platform-independent representation of instruction meaning. This assumption makes sense given that some disassemblers support multiple CPU architectures and such representation would make their internal implementation significantly simpler. This is the reason why the original idea behind this thesis was to implement only a plugin for one of the existing disassemblers.

Using an existing disassembler would be beneficial as we would not have to implement the machine code parsing logic as well as we would not have to design our own representation of instruction meaning. The disassembler would do such work for us. This would allow us to focus on instruction reordering and emulation rather than distract ourselves with problems around the custom representation of instruction meaning.

Unfortunately, it turns out that even though some disassemblers support many CPU architectures, their plugin interfaces are by far not as generic as one would expect. Research of some disassembler plugin interfaces including IDA [3] and Ghidra [4] showed that those interfaces typically provide instructions as a plain array of bytes. Sometimes, there are some annotations informing about more detailed properties of the instruction, but those annotations are by far not sufficient to represent the full meaning of an instruction.

For this reason, it was not possible to implement a plugin for an existing disassembler. Instead, we will implement a new disassembler that will be internally fully independent on any platform as it will use its own internal representation of code meaning. This property of our disassembler will make it not just more generic and extensible than existing disassemblers, but it will also allow us to implement different sets of algorithms and transformations than existing disassemblers implement.

1.2.2 Requirements for instruction representation

Before we start to design our internal instruction representation, we should first clarify which features our representation should have. To construct such a list of requirements, we will take a look to existing CPU architectures and their instructions. It is reasonable to assume that even though those architectures are very different, they will use very similar concepts. Based on those observations of common patterns, we should be then able to construct a list of requirements for our representation.

The first concept every CPU architecture has to provide is registers. Every CPU nowadays provides some number of registers that are uniquely identified by numbers. In every architecture, registers are the fastest but very limited kind of storage. Registers are also usually the only possible input and output of instructions performing some operations including calculations and memory accesses. What differs between CPU architectures is register width as well as the total number of registers.

This observation implies that our representation has to provide some way to represent registers. It is also obvious that the representation has to support an arbitrary number of registers and arbitrary register width. And last but not least, we have to be able to use values read from registers and perform calculations on top of them. Results of those calculations again have to be storable into registers.

The second concept shared between all CPU architectures is the existence of memory. From a CPU perspective, memory is a set of bytes that the processor can load and store. In comparison with registers, memory is usually much slower. On the other hand typical memory provides gigabytes of storage compared to low hundreds of bytes in registers. Another substantial difference between memory and registers is that registers store whole values, but memory stores them byte by byte. In other words, a write to a register replaces the whole value of the register. This is not true for memory where each write can write a different number of bytes.

Many CPU architectures use specialized instructions to manipulate memory. Those instructions are usually called *load* and *store* and they load value from memory to a register or they write value from register to memory respectively. The address of memory access is usually provided to those instructions using another register, but the instruction can also define some calculation that produces the final address accessed. It is worth noting that some architectures for example x86 and AMD64 provide single instruction to load value from memory and to perform some calculations on top of the loaded value. Analogously there are instructions to perform calculations and store the result in the memory.

The implication for our intermediate representation is that it has to provide some concept of memory (i.e. byte oriented storage). There also have to be some concepts of memory load and store to an address which is either register or the result of some other calculation. Memory loads and stores will also have to allow to access a different number of bytes. And last but not least, we will have to be able to store the result of any calculation in memory and load any value from memory to be used in calculations.

The third commonly shared concept in CPU architectures is the support for immediate values encoded in instructions. An immediate value is a constant value that is encoded directly in instruction bytes and somehow affects the calculation performed by the instruction. For example, the addition of 2 registers can also encode the value of 5 as its immediate value. Such an addition then adds two registers and adds 5 to the result. Alternatively, those immediate values can be used to alter memory addresses. And last but not least, immediate value can be used to set a register of a memory value to a constant written in the code.

To represent immediate values in our representation, we will have to have some sort of constants. We will have to be able to include those constants in any calculation performed by any instruction. Besides that, we will have to be able to store those constants to both registers and memory.

The last big set of common instructions is control flow instructions. This is quite obvious as there has to be a way to alter the program execution flow in any CPU. In all nowadays CPU architectures, control flow instructions have form of jumps. A jump is an instruction instructing the CPU that the next instruction executed should be loaded from a given address in memory.

One specific feature of jumps is that they can be both conditional and unconditional. Unconditional jumps as their name suggests affect the code execution whenever they are executed. On the other hand conditional jumps affect execution only in certain cases. Different CPU architectures define different ways to decide whether jump will be performed or not. But we can say that every CPU architecture decides on conditional jump execution based on output of some calculation. In some architectures this calculation is just plain load of a single bit from a single register (usually called $FLAGS$), in other platforms there can be more sophisticated calculation performed for example signed integer comparison.

Implications of control flow instructions for our representation are that it also has to be able to describe jumps. Moreover, as jumps can be conditional, we will need a way to represent both the condition under which it will be executed as well as the fact that execution of such instruction depends on some condition.

Besides concepts listed before, there are also other commonly shared concepts. For example, each CPU architecture capable of running an operating system has to provide some sort of syscall. Other common concepts include synchronization instructions which affect memory access order in multiprocessor systems. This again implies that we will need to find some way to represent those special instructions.

1.3 Instruction parsing

Once we have our internal representation designed, we will need to implement a way to transform the input binary file into it. Such an algorithm will accept an array of bytes written in a binary file and its purpose will be to understand which instructions are encoded in it. Once it will know which instructions are encoded in the file, the other purpose of such an algorithm will be to describe the effects of those instructions in the disassembler internal representation.

The first aspect of machine code binary parsing we will have to discuss is which parts of the parsing process are specific to a given CPU architecture and which of them can be generalized for all existing CPU architectures. The shared aspect of any CPU architecture is that code is represented as a sequence of instructions. Another shared aspect of all CPU architectures is where bytes representing instructions will be located in a binary file. On the other hand, the meaning of bytes representing instructions will be definitely specific to a given CPU architecture.

This implies that internally there will have to be some split between the CPU specific and the generic part of the parsing logic. Moreover, the interface at the boundary of those two components will have to be very well defined and generic enough to support parsing of machine code of any existing CPU architecture. The reason is that if we ever decide to make the disassembler capable of parsing another instruction set, we will replace just the CPU-specific part of the parsing logic. In other words, we will provide another implementation of this internal interface.

One of the challenges we will have to consider while designing the interface in between those two parts is that different CPU architectures might have different byte lengths of instructions. Moreover, there are CPU architectures such as x86 and AMD64 where different instructions might have a different number of bytes even within that single architecture. For this reason, we will have to design an interface between those two components to be able to parse instructions encodes in an arbitrary number of bytes.

The purpose of the parser will be to read bytes encoding instructions and to deduce which instruction they encode. The parser will do so by comparing those bytes with all opcodes it knows. An instruction opcode is a set of bits in an instruction encoding which represent the purpose of the instruction. For example, an *add* instruction encoding addition of 2 registers is identified by a specific opcode (i.e. specific values of specific bits).

One might hope that bits representing opcodes are given and fixed for the whole CPU architecture. For example, the first k bits of instruction always encode opcode bits. Unfortunately in reality different instructions have different opcodes and even different sets of bits the opcode is encoded in. Some bit might for example encode the number of a register in the case of one opcode and a bit of opcode in other opcodes. This property of opcodes implies that we will have to design a representation of opcode which will allow to say not just bit values, but also which bits encode opcodes and which does not. Once we have that, we will need to implement an algorithm which is for a given set of bytes able to say if they correspond to a specific opcode.

If we match bytes to a certain opcode known to the parser, we already know enough to parse the whole instruction. The opcode does not encode just the purpose of the instruction (i.e. whether it is addition or multiplication), but it also encodes which bits in the encoding represent registers, which of them are immediate values and so on. Based on this information, we know the set of input values of the instruction as well as the set of output values. We also know the action the instruction performs and how it deduces values written. Consequently, we can use this information to encode the instruction into our internal representation.

1.4 Instruction reordering

An instruction reordering algorithm will be the first CPU-independent logic in the disassembler. This implies that the only input of it will be the array of instructions represented in the disassembler internal representation. The goal of the instruction reordering algorithm will be to give the user ability to change the order of instructions in the program. But as we already mentioned before, we will also require the instruction reordering logic to keep the program meaning unchanged.

To guarantee that change of instruction order will not change the program meaning, the algorithm will have to know which instructions are dependent one on another. If one instruction is dependent on another instruction, then by swapping those two instructions we will obviously change the program meaning. This is given by the fact that the action the latter instruction depends on will happen after its execution. This observation implies that the primary goal of the instruction reordering algorithm should be to find dependencies between instructions.

We know four types of instruction dependencies: true dependencies, anti dependencies, output dependencies, and control dependencies. The first 3 dependencies can be summarized as data dependencies as they exist between instructions that use the same data in either memory or registers. Control dependencies are in a way special as they occur between two instructions when one instruction decides whether and how many times will be the other instruction executed. In other words, control dependencies exist between jump instructions and other instructions in the code including other jumps.

We will start our dependency analysis with control dependencies. To track them, we will split the code into basic blocks. A basic block is a sequence of instructions that is always executed as a whole. We can describe such property as follows: Any instruction in the basic block executes if and only if the first instruction executes and all instructions in the basic block execute in a sequence. This already implies that whenever the first instruction of a basic block is executed, all instructions in the basic block are executed. This property of basic blocks provides us the partition of the original code into many blocks which always execute as a whole.

Properties of basic blocks imply that two instructions from different basic blocks cannot be reordered one for another as such reordering would violate properties of basic blocks. To provide an example, there can be instruction in basic block A which loads value used in both basic blocks B and C. If B and C are two branches of a condition, we cannot reorder instruction from A neither to B nor to C. If we did so, the other basic block would miss the value loaded in basic block A. Using this knowledge, we can deduce that any possible instruction reordering has to take place within a single basic block. This allows us to limit our data flow analysis always to a single basic block. This is also the reason why we started with basic block analysis first.

To find data dependencies, we will iterate basic block instruction by instruction. Given that all 3 data dependencies exist between instructions that either load or store particular register or memory location, we should be able to deduce those dependencies from our internal representation.

Once we know all basic blocks and all dependencies within them, the only remaining part will be to design data structure and operations for instruction reordering. This might seem to be a simple task, but we have to take into account the need to be capable of dependency tracking between instructions even after reordering. For this reason, a plain array of instructions will not be sufficient.

1.5 Emulation

Our internal interpretation of instructions should encode all effects of instructions including both register reads and writes as well as memory accesses and calculations performed on top of those values. Consequently, we should be able to use the representation for program emulation. Moreover, we should be able to interpret the code reordered by the instruction reordering algorithm.

We can emulate the program by emulation instruction one by one. To do so, we will need to represent the program register and memory storage. For each instruction writing a register, we will update the register storage. Whenever an instruction reads a register, we will read that register from the storage. The same logic applies also to memory. When an instruction performs a jump, we simply finish the emulation of the jump instruction and then continue the emulation with the instruction which was targeted by the jump. This way, we should be able to emulate the whole program execution.

This idea of emulation seems trivial, but there are a few complications in that simple premise. The very first problem is that the emulation does not have to always start at first instruction of the program. In a case when we start emulation somewhere in the middle of the program, it naturally lacks all memory and registers values written by the previous instructions. To address this problem, we will have to somehow allow the user to provide the value to the emulation.

Unfortunately, our current formulation of emulation is still very naïve due to the existence of an operating system. Every useful program needs to interact with the operating system to get its inputs and write its outputs. It usually does so using system calls. As the program code relies on the operating system performing its actions whenever it is asked to do so, we cannot simply omit this part from our emulation. But to emulate system calls, we would either have to emulate the whole operating system as well or we would have to understand the meaning of all system calls and emulate their actions appropriately. Both those solutions are out of the scope of the thesis.

Instead of emulating the operating system, our emulation will expose an API allowing to emulate the operating system. The user will be able to use this API to emulate the operating system. This design choice makes good sense given that the ultimate goal is to assist the user in understanding the program meaning rather than to provide perfect emulation. Moreover, some future extensions of the disassembler might implement emulation of operating system actions.

1.6 Analysis summary

To fulfill the goals of the thesis, we will implement the own disassembler from scratch. Internally, the disassembler will use an internal representation of instruction meaning independent on any existing architecture. Even though this implementation will be generic, it will also have to be able to describe concepts of existing CPU instructions.

The disassembling process will have multiple stages:

1. Statically-linked binary is read as bytes.
2. Machine code parsed using CPU specific parsing plugin.
3. Internal representation is produced by the parser.
4. Instruction reordering identifies dependencies and allow the user to reorder instructions.
5. Instruction emulator emulates execution of reordered code based on the internal representation.

The overall design of the disassembler is shown in Figure 1.1.

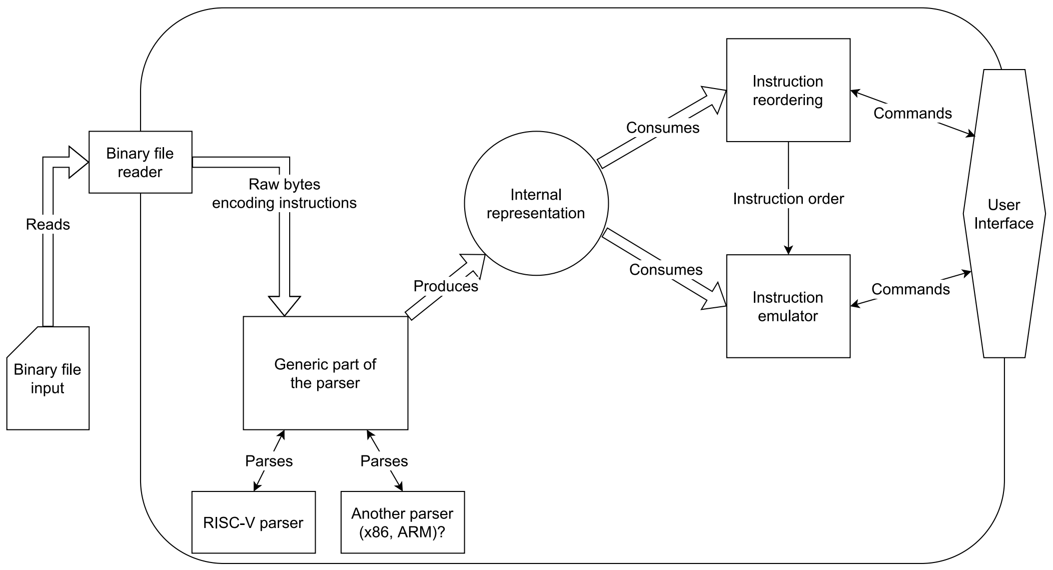

Figure 1.1: Visualization of *mltwist* structure.

This disassembler will be implemented using Go programming language and its name will be *mltwist* [5]. The reasoning behind this name is that it is disassembler allowing instruction reordering. The other way to say instruction reordering is Machine Language Twisting → *mltwist*.

2. Platform independent instruction representation

As we have described in the introduction, we will have a few non-trivial requirements on the internal format representing instructions. One of the most important requirements is that it should be generic enough to support an arbitrary platform. Our other requirement is that it has to describe all instruction effects as this is the property that will provide us dependency tracking as well as the possibility to simulate the program. And last but not least it should be as simple as possible to simplify algorithms and reasoning about them.

It is worth noting that such an internal representation could be also called intermediate code. The intermediate representation is the term used mostly in compilers which is used to describe internal, (typically) platform-independent representation of instructions. The purpose behind intermediate representations existence is exactly the purpose of our disassembler instruction representation - to make compiler optimizations platform-independent and more high-level than real CPU instructions are [6]. To point out this similarity, we will use the term intermediate code[1]to describe our platform-independent instruction representation.

Another aspect of the intermediate code that we should point out is that we do not aim for it to be super-efficient in terms of the total number of instructions. To speed-up CPU execution and also make the program encoding more efficient, most instruction sets are overdetermined. In other words they contain more instructions than strictly necessary. A good example of this are instructions *add*, *sub* and *bit-negate* which are usually present in all architectures even though *sub* can be implemented using appropriate sequence of *add* and *bit-negate*. Another good example are comparison instruction *lt*, *le* and *eq* where obviously one of those 3 could be dropped if we have *or*[2] instruction available.

We want to have our intermediate code as simple as possible, so we will try to avoid having instructions that could be represented by a sequence of other instructions in our intermediate code. Naturally, we have to find a good balance between simplicity and size of the representation. We obviously do not want to represent multiplication as repeated adding as that would blow our representation too much in the disassembler memory. On the other hand we will not care about replacing *sub* with a sequence of *add* and *bit-negate* as we explained above.

[1]We will use ”intermediate code” rather than ”intermediate representation” (IR) to avoid naming conflict with the compiler concept. Even though this concept is similar, its purpose is slightly different, and its technical details also differ. Specifically, our representation will be very close to raw machine code rather than to the abstract algorithm description used by compilers. For example, there will still be clear instruction boundaries which is not the case with intermediate representation in compilers.

[2]We can consider both logical and bitwise *or* as logical *or* can be emulated using bitwise *or*

2.1 Value representation

Before we start to design individual instructions of the intermediate code, we should first design a way to represent values in it. Even though this problem is very simple in typical CPUs, it turns out that this problem is by far not simple in our intermediate code due to its genericity.

2.1.1 Value width

A typical CPU architecture typically defines a set of registers, each containing a certain number of bits to represent a value. There are typically also instructions working on fractions of the full bit range (usually powers of two). But these lower-bit instructions can be implemented using operation on the full register width with appropriate bit masks applied to both input and output operands.

The problem is that we want to support many architectures with different register widths. Consequently, we cannot pick just one register width. One might have an idea that we pick just the widest one and use bit masks to represent operations on registers with fewer bits. This approach would work quite well, but we do not want to depend on the current set of existing architectures only. In other words, there will once come a CPU architecture with wider registers than the maximum we picked and we would not be able to represent the values of this new CPU architecture.

Instead of any finite number of bits, we define our values as an infinite sequence of bits. As we explained above, we will be able to apply bit masking operations for an infinite number of bits to emulate any operation for an arbitrary number of bits. Even though this sounds very theoretical as we simply do not have infinite memory space to represent them, we will not need that in the end. It is just useful to reason about registers as being infinite.

To sum it up, we will use an infinite number of bits to represent a value. We will emulate finite bit operations on top of them using bit masking of both input and output values. An operation width will be one of the parameters of every operation in our intermediate code.

In practical implementation, we will not naturally store infinitely long values. That is by far not necessary. If a value was masked by an operation to K low bits, we are certain that bits $[K..inf)$ (with bit indexing starting at zero) are all set to a certain value - we will discuss which value should it be further in this chapter. So we do not need to store those. Consequently, the in-code representation of a value will be a simple byte array where bytes at indices higher or equal (we are indexing arrays from zero) to array length are implicitly understood as zeros.

Please note that this definition implicitly states that all our values are represented using little-endian byte order. This is implied by the fact that we assume all higher bytes (bytes with the index above K) to be zero. If we used big-endian values, the zero bytes would be the low byte indices, not the high indices. Usage of little endianness is a logical choice as the majority of commonly used CPU architecture uses it. Given that we have to choose some endianness it makes the best sense to choose little endian.

It is worth noting that our representation of values is very close to a type which is usually referred to as *BigInt*. *BitInt* is a type that uses a byte array (or array of some fixed-size, multi-byte types) to represent an arbitrarily big integer. The *BigInt* type then provides typical integer operations on top of the byte array. In *mltwist*, we do not use *BigInt* library type itself as it introduced additional complexity which was unnecessary for value representation. But we use *BigInt* to implement some operations on top of our values.

An interesting problem arises in a situation when the input value is not wide enough. In such a case, we have to extend the value somehow (i.e. to fill the top bits of the value). There are 2 possible ways to do so which are commonly called *signed extension* and *unsigned extension*. As their name suggests, they are used to bit-extend either signed or unsigned integers. An *unsigned extensions* fills all upper bits with zeros. This kind of extension ensures that the unsigned value of an integer is preserved. On the other hand *signed extensions* copies the highest bit of the original number to all new bits. This kind of extension preserves integer value in two's complement signed integer representation. Obviously, there is no silver-bullet solution here. If we chose *unsigned extensions*, signed arithmetics will be more complicated to represent in our intermediate code as *signed extensions* will have to be implemented in the code (read by instruction of the intermediate code). On the other hand, if we decide to use *signed extension*, the unsigned arithmetic will face exactly the same issue.

In the end, we will use *unsigned extension* over *signed extension*. There are several reasons for this decision. First of all, *unsigned extension* is simpler to implement (we can unconditionally just add zeros) compared to *signed extension* where the value of bits filled depends on the top-most bit of the original value. Second, this decision will allow us to deduce more information about the value itself. In some parts of the disassembler, we will perform several optimizations and transformations of the intermediate code. In some of those steps, it might be useful for us to know the most about the value. But as disassembling is a static code analysis, we might not know the value itself – it might be known only at runtime. The advantage of *unsigned extension* in those regards is that we know that if we extend an unsigned value of K bits to L bits, that bits $[K..L)$ are zeros. Which is a stronger statement than in the case of *signed extension* where we would only know that all those bits are either one or zero but their exact value would depend on runtime value. And last but not least, it is in a way simpler and more common for people to think in terms of positive integers than in terms of two's complement integer arithmetic. This argument might sound a bit like hand-waving and it in a way is hand-waving. On the other hand, this promise of simpler reasoning about optimization algorithms later was also one of the reasons why it was decided to use *unsigned extension* over *signed extension*. We will discuss how can to implement a signed extension later in this chapter in section 2.2.3.

The last design choice is whether we need intermediate code operations with different input and output value widths. Given that there does not seem to be any real use-case for different widths of input and output parameters, it was decided to keep the intermediate code as simple as possible. Consequently, every operation has a property called *width* which specified the width of all input parameters as well as the width of all output parameters. This approach is very similar to the way operations in CPU usually work as the majority of real CPU instructions use the same register width for both input and output.

Note that if there ever was a use-case for different *width* of input and output parameters of an operation, such an operation could be still represented in our intermediate code. The only difference is that such an operation would have to be composed of multiple intermediate code operations. To construct such a macro (a set of operations chained to achieve the goal), we would use all operations of width which is the maximum of input and output parameter widths. We would compose the macro of masking input values to an appropriate (input parameter) bit width followed by the operation itself. We would then use masking to remove excess bits from output values. Consequently, the decision of having a single *width* parameter applied to both input and output parameters is not limiting in terms of which operations can we represent in our intermediate code.

2.1.2 Storage representation

Any computer needs some place where it can store values. This storage has to be quite big to allow the computer to perform complex calculations. As our intermediate code is supposed to represent instructions of a real CPU, we need to find some storage representation for our intermediate code which will be able to emulate the behaviour of storage of any existing CPU.

Nowadays, all CPU architectures have two basic kinds of storage - registers and memory. The exact numbers of registers might differ from architecture to architecture. The same applies to memory where both the size of the memory, as well number of independent memory address spaces might differ. Registers are usually fast and are used to store output values of operations as well as to provide input values to operations. Some architectures indeed allow to use memory as both input or output of instruction, but those operations are internally implemented using intermediate registers.

No matter how many registers and how big memory the CPU has, the fundamental difference between those two is how data is stored in them. Registers typically have some width on bits and the whole register is treated as a single value. On the other hand, memory storage is always byte-oriented. Each byte is memory is then assigned a sequential number which is called memory address. A multi-byte memory load and store then affects multiple bytes, which implies that multiple memory addresses are modified by a single operation.

It is important to note that if we were to pick either registers or memory to represent the storage in our intermediate code, memory would be a better choice. The reason is that the byte-oriented property of memory is non-trivial to emulate in registers. Such emulation is possible to implement using bit-shifts, masking, and ORing bits in registers, but it is complex and uncomfortable to use. On the other hand, we can quite trivially represent registers in memory. If our CPU architecture has x registers with byte width w, we can trivially use memory addresses in the form $i * w$ to load or store ith register value. This construction will work even better once we introduce multiple address spaces further in this section.

Even though we will introduce registers in subsection 2.1.3, in the rest of this section we will pay attention only to memory representation. The primary reasoning behind this is that due to the construction described above, it should be possible to extend this work into a state where registers can be replaced by memory. Once such a state will be achieved, there will no longer be any reason to have register storage anymore.

In our intermediate code, we cannot use any finite number of bytes in memory. To conclude so, we can apply the same argument as we did in subsection 2.1.1 that any finite number of bytes in memory will be one day exceeded by a new CPU. So we will again define our memory as infinite sequence of bytes and any parser of CPU-specific code will be able to limit it itself.

So we define our memory storage as an infinite sequence of bytes addressed by non-negative integers. So the first byte of our memory is at address zero. As we stated above, k byte write to address a will set bytes a to $a + k - 1$. Naturally, every memory address is also value in the sense described above in subsection 2.1.1 and as such, it is unlimited. This allows us to emulate infinite memory address space.

One additional aspect we have to take into account is that some CPU architectures have multiple memory address spaces. A good example of such architecture is again x86 and AMD64, which besides standard memory space also defines I/O address space. Given that any real memory address space of a real CPU architecture is finite, we might just say that all those spaces can be represented in a single infinite address space. That statement is true. But to increase readability, it good makes sense to add support for multiple address spaces, where every memory address space is identified by a string *key* and individual address spaces are fully independent. This simple modification allows us to conveniently represent the I/O address space or any other address space in our intermediate representation, but it also simplified the register-in-memory representation described above.

2.1.3 Additional registers

Even though we argued above that an infinite memory is fully sufficient to store all values in an infinite memory, we will also have to add another key-value store for registers. A register is identified by an arbitrary string *key* (this principle is the same as for memory address spaces) and contains a single value. Unlike in memory, the value in a register is not split into individual bytes but it is stored as a whole in that single register.

First of all, it is worth mentioning that registers are much less generic than memory. The reason is that they are not identified by address, but just by a string key. Consequently, we will not be able to implement the operation ”read register with the number written in this register”. We can do this in memory as we simply load the memory with the address and then we use it as an address to memory. But we will not be able to do this with registers. This can be limiting as some CPU architectures for example x86/AMD64 has such kind of operations. A good example of such operations are instructions *RDMSR* and *WRMSR* which read number of MSR from registers *%edx* and *%eax* [7].

The reason to introduce registers is exactly the lack of variability in register operations. The problem with generic memory representation we have is that it is non-trivial to find which memory accesses depend on one another. It should be possible and we will talk about this in section 7.2. But unfortunately, we did not get far enough in the work to be able to perform an analysis of memory accesses. Consequently, we need registers that are less generic but way simpler to analyze to be able to find at least some dependencies between instructions.

Even though registers increase the disassembler complexity to simplify the dependency analysis, one might also argue that it makes sense to have both memory and registers available as that is the model we are used to program in. In a way, this argumentation makes sense given that all current architectures have the register number written directly in the instruction opcodes. This implies that we can determine the register at the time we are decoding the instruction. So, we can understand registers as both simplifying the analysis and simplifying the human mind model of the intermediate code.

2.2 Operations

In this section, we will discuss which instructions we have available in our intermediate code and how can we use them to represent instructions of a real CPU architecture.

2.2.1 Expression composition

First of all, we should discuss how will we compose our intermediate code instructions to form more complex expressions. This is important as our intermediate code instructions will be simple. To build a more complex algorithm, we will need to compose multiple intermediate code instructions together. Given that we will use our intermediate instruction composition frequently, it is desirable to design some intermediate code instruction composition method that will be simple to use.

In CPUs, instructions are naturally interconnected using registers that pass values between them. Such a design works, it is fast in real CPUs and it is as well quite simple for humans to understand. So the very first idea one might have is that we will re-encode every machine code instruction to a sequence of intermediate code instructions. With such a design we could pass values in between them using registers and the whole intermediate code would be very similar to machine code.

The problem with this approach is that we would definitely want to have some macros to represent more complex calculations. For example, later in this chapter, we will discuss that a single bitwise AND will be represented by 4 intermediate code instructions. So we will have a macro representing AND operation. But this implies that the macro will internally use some set of registers to pass values between its four instructions. This would require the user of the macro to guarantee that the registers used by the macro are not for the time being in use. Moreover, this might introduce false dependencies between intermediate instructions based on just register name collisions. To sum it up, the design using registers to pass data between instructions would make it hard for us to compose individual intermediate code macros together due to possible register collisions.

Instead of the registry-based approach of instruction chaining, we will use a functional approach where we will simply use outputs of individual instructions as input of other instructions. The functional approach has several advantages. First, this model describes the semantics of the operation rather implementation details of the calculation. This will be especially important later when we will not have to implement special handling for artificial instruction dependencies. Second, this representation allows us to describe any complex operation as a tree of operations. One huge advantage of a tree compared to a sequential code is that we can perform operations on subtrees independently of the rest of the code. Third, no expression has any side-effect besides producing an output so they can be freely replaced and reordered. This would not be possible in a register-based approach as we would have to be certain that no other instruction consumes the other input value. And last but not least, this will be much simpler to use in the platform code parsing phase as we can trivially combine expressions without having to deal with register collisions, order, intermediate values, and so on.

To mention some disadvantages of the functional approach. We should first point out that this kind of intermediate code representation can increase the total size of the code. In a situation when we need to use some intermediate result multiple times, we cannot simply store the result and reuse it multiple times. A good example of such a situation is RISC-V atomic swap instruction. This instruction takes accepts two input registers – the base of address and the value to write. Based on the base address and its immediate value, this instruction calculates the memory address swapped. It then uses the address to load the current value of memory to the output register as well as to store the input register value into the memory. In such a case, the expression calculating the address is used twice.

Instead of storing the intermediate result and reusing it multiple times, we have to copy the whole tree describing the computation of the value multiple times. This can naturally slow down the execution, but what is even more serious, it can increase memory consumption significantly. As any smart disassembling is already quite memory hungry on itself, this might be an issue. For some big binaries, we might soon run out of memory available to common computers[3].

Luckily, we can remedy the memory consumption problem by making all instructions immutable. Immutability refers to the property of an object which guarantees that the object representing an instruction can never change during its lifetime. If needed to change the object, we would have to allocate a new one. The immutable instruction representation allows sharing subtrees of computation between multiple expression trees as we are certain that the shared instruction subtree can never be modified from other places. So wherever we need to use a result of an operation in many places during the code parsing phase, we can simply reuse the same functional object many times and reduce the additional memory const of functional representation to zero compared to register-based approach.

Indeed, making expressions immutable does not save time spent on expression evaluation. But immutability enables lazy evaluation. In lazy evaluation, we typically assume that we have immutable objects, so we can afford to postpone the evaluation of the expression until it is really necessary to know the value. Once we evaluate the value, we cache it as we are certain that repeated evaluation of the expression will produce the same result. This is guaranteed by immutability. On the other hand, to cache the evaluated result, we would need to have some additional space in the instruction representation object where the result could be

[3]Remember that the disassembler has to keep the whole program code including all its libraries in memory at once which increases the overall memory usage.

stored[4]. Unfortunately, the disadvantage of result caching in expression is that the expression itself suddenly consumes additional memory. We will prioritize lower memory consumption over faster evaluation and for this reason, the result caching field (which would implement lazy evaluation) is not included in the intermediate code representation. Alternatively, we could replace the expression subtree with the result itself. We will discuss something very similar to this in subsection 4.3.1, so let's postpone this discussion there.

To sum up the overall design, every expression (instructions) in our intermediate code will be defined by some number of input expressions (the exact number will depend on the instruction itself) and *width* of the operation. Every expression will then produce a single result whose value *width* will be exactly the *width* of the operation. And naturally, any expression can be used as input to another expression. The schema of expression chaining if visualized in Figure 2.1. This way, we will describe the whole calculation using primitive operations of our intermediate code.

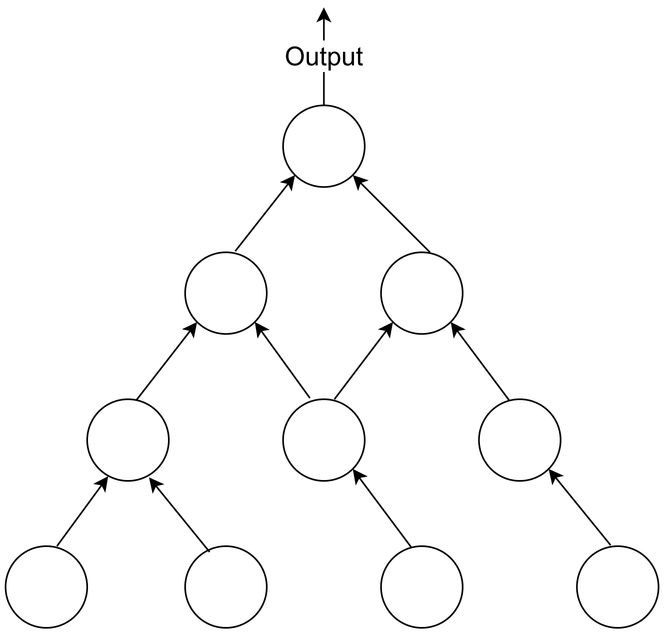

Figure 2.1: Visualization of expression composition.

2.2.2 Effects

The expression composition model explained above works great while describing a single machine code instruction, but it will not help us to represent relations between instructions. We obviously cannot use expression composition to represent data flow between instructions as we do not know the order in which they are executed. So we have to find some other way to describe instruction side-effects which are then consumed by other instructions. In other words, we

[4]It is worth noting that storing the result in an immutable object is technically a modification of an immutable object. But given that the result written is fully determined by the immutable parameter of the object, such a modification is typically not considered to be an object modification.

need a way to modify our memory and registers discussed in subsection 2.1.2 and subsection 2.1.3 respectively.

To represent instruction effects, we will add a new entity to our intermediate code representation which we will call effect. An effect modifies either a single memory place (which can comprise multiple bytes) or a single register value. To obtain the value to write to the memory of set a register, an effect accepts an expression (or multiple expressions) as its input parameter. A visualization of a single effect consuming two expression trees can be found in Figure 2.2.

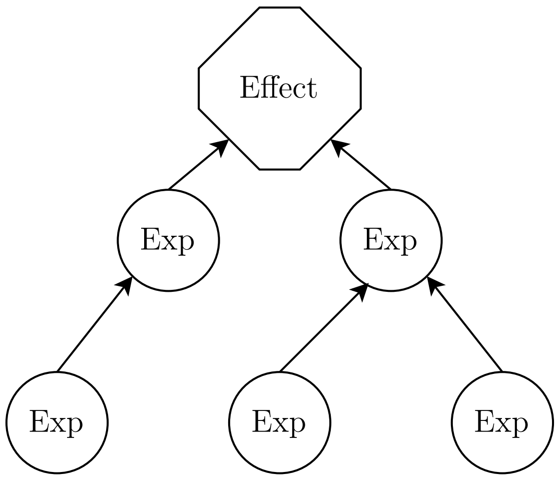

Figure 2.2: Visualization of effect construction.

It is worth mentioning that an effect differs from an expression because it does not produce any output. In other words, it can never be used as an expression to compose more complex expressions. This makes sense because an effect on itself does not perform any calculation, they just move the value from expression to some storage place. If the value written to either memory or register is required in any other expression, an expression passed to effect can be used. This design of effects grants us a nice property: We are certain that an expression always comprises of expressions that have no side effects, so their execution order does not matter.

2.2.3 Intermediate code instructions

At this point, we can introduce a list of instructions in our intermediate code. There will be 5 different kinds of expressions and 2 kinds of effects in our intermediate instruction set. Out of those, one kind of expression will have sub-kinds.

Costant expression

The simplest expression in the intermediate code is a constant expression. A constant expression is an expression with no inputs which produces a constant value. Constants use the same representation as values (i.e. an array of bytes) and as every expression, they have also defined *width*, Constant expressions can be used to emulate constants written in the machine language itself (constant loads).

Loads and Store expressions

As described in subsection 2.1.3, we have registers identified by strings, so we have to have a way to load and store those registers in our intermediate code (do not confuse those loads and stores with memory loads and stores). To do so, we introduce an expression to load a value from the register and an effect to store a value to a register. A register load operation comprises of string *key* of a register and *width* of the value to be read. The store register effect contains both the *key* and *width*, but on top of that, it contains an expression to store to the register.

Besides registers, we have also memory so we need another store memory effect and load memory operation to manipulate the memory. Those are different operations that register loads and stores as they require a different set of parameters. Memory load comprises a memory address space *key*, an expression calculating the address to read from, and *width* of the load. Memory store comprises of the same parameter set, but on top of those, it contains an expression to store to the memory. It is worth mentioning that, unlike the register case, the *width* parameter of both the load and store has one additional meaning here. That additional meaning of *width* is how many bytes are accessed in the memory (i.e. read or written). This is something we cannot see with registers because registers store the whole expression. But as memory is byte-oriented, the *width* of load and store are also important for dependency analysis.

Conditional expression

To implement a Turing-complete computer, we need some sort of condition. As we are using a functional approach to describe expression trees, our condition will also be very functional. The condition instruction will accept two expressions to compare and the other two expressions as *true* and *false* expression. The condition instruction will compare the first two arguments and based on the comparison result it will return either *true* or *false* expression respectively.

An important question here is how many different comparisons we need to describe any condition possible. Even though a typical CPU architecture defines a wide set of conditions, it turns out that we only need one comparison. In our case, this comparison is unsigned less-than (*ltu*) comparison[5].

We can trivially use less-than to implement the greater-than operation by simply swapping operands. We can check the equality of any value to zero by simply comparing if the expression is less-than one as only zero is less-than one in unsigned comparison. We can implement equality comparison by subtracting two expressions and comparing if the result is equal to zero. We can also compose multiple unsigned comparisons to implement signed comparison by checking the signs of operands first and then comparing their absolute values based on those signs. In combination with arithmetics (which will be both described later), we can compose any condition using just unsigned less-than comparison.

It is worth noting that our conditional instruction implements the same operation as the ternary operator in many languages including C and Java. This

[5]Signed less-than comparison (*lts*) would also work, but as we use the unsigned extension of values, it is more natural to use an unsigned comparison rather than a signed version. This way the design is more consistent.

is quite useful while we reason about the operation. The pseudo-code of our conditional instruction looks like this:

```
    (expr1 < expr2) ? exprTrue : exprFalse
```

As our condition instruction is an expression, it also has *width* associated with it. The meaning of *width* might be slightly counterintuitive in case of less-than condition, the *width* applies to both the arguments to be compared as well as on *true* and *false* expressions. The reasoning behind it is that in the case of the condition, the expressions to compare are sort of input parameters, while the *true* and *false* expressions are the output of the expression. One might argue that this is too binding and that the instruction should have two independent widths. In practical usage, there were only a few places where it would be useful to have separate condition *width* and output *width.* It was simpler to special-case those few places and to manually adjust widths before or after conditional expression than to add another parameter to the less-than operator.

Binary expression

The last thing we need to have a general-purpose processor is arithmetics. So we define a binary expression instruction with two input expressions.

Obviously, a single binary operation is not sufficient to efficiently[6]represent all possible arithmetic operations. In other words, we will need multiple binary operations in our intermediate code. To represent multiple binary operations a binary instruction also has an operation *kind* field which specifies which binary operation is applied to input expressions.

One might argue that it would make more sense to introduce multiple instructions each representing a single binary operation *kind.* Others might argue that usage of operation *kinds* as parameters of a single binary operator is inconsistent with the rest of the expression operators as this is the only instruction the functionality of which depends on one of its parameters. Both those arguments are valid. The reason why we have not introduced five separate binary operations is the complexity it brings to further expression processing. We will have some optimization algorithms which will not care much about the nature of a binary instruction but only about the fact that it is binary instruction with two input expressions. By introducing multiple instructions with the same signature, we would force optimization algorithms to special-case each of those instructions independently which would increase the disassembler complexity for no proper reason.

Alternatively, we might of course represent the shared functionality by an interface and call functions on an interface. This would work except for the fact that our interface would have to support situations when one of the input expressions changes. As all our expressions are immutable, we always need to create a new instruction when one of its operands changes. Consequently, our interface would have to expose a function that says ”will produce the same type

[6]By efficient, we refer to the fact that any common CPU instruction can be represented by the constant number of instructions in our intermediate code. So implementing subtraction with 3 intermediate instructions is efficient, but implementing multiplication using repeated addition is not because the number of instructions required is proportional to *width.*

as the original type but with those input expressions". This function seemed to violate the common design patterns so much that we will rather stick to operation *kinds*. To sum it up, we will use a single binary expression parameterized by an operation *kind* rather than multiple types with the same signature.

We define several binary operation *kinds* which are universal enough to compose all other common operations in the CPU. To be specific, we define those 6 binary operation kings:

1. nand
2. add
3. lsh (logical)
4. rsh (logical)
5. mul (unsigned)
6. div (unsigned)

The set of operations available is quite limited but generic enough to be able to represent most operations we can find in modern CPUs in an efficient way.

Any bit-wise operation can be represented using a NAND gate because NAND is a universal logical operation [8]. Namely, we can implement NOT, AND, OR and XOR [9] using only NAND gates. Consequently, this single logical binary operation is sufficient for us to compose any logical operation we might need to represent.

Subtraction can be implemented as the addition of a negated number. An integer negation in two's complement arithmetic can be implemented using bit-wise NOT and addition of one [10]. It is worth mentioning that signed and unsigned addition share the same algorithm, so we do not have to differentiate between signed and unsigned addition and subtraction. Note that in section 2.2.3, we blindly assumed that we will have a subtraction operation to implement equality comparison. But here we prove that we have subtraction operation available so the statement above holds.

To implement many common operations, we need bit shifts. Namely logical bit shifts. There also exists an arithmetical right shift. The difference between logical and arithmetical right shift is that logical shift fills shifted top bits with zero unconditionally. While the arithmetical right shift keeps the sign of an integer unchanged [11], it fills new bits with either zeros or ones depending on the sign-bit value of the shifted value. For obvious reasons, arithmetical left shifts make little sense. Our intermediate code supports only logical shifts, but arithmetical right shift (*rsha*) can be implemented using logical shift followed by a few bit maskings and conditions. The idea behind this algorithm is that we start with a mask full of ones. We can shift this mask right by the same amount of bits as we shift the value. By subtracting the non-shifted (all ones) and shifted mask, we will get a bitmask of bits added by the right shift. We then just check by unsigned less condition if the original vales shifted is negative or not. If it is not, we leave the result of the logical shift unchanged. If the shifted value is negative, we OR it with our mask created by subtraction.

Another important operation we can implement using conditions and logical shifts is the sign extension of an integer value. The idea is very similar to the case of *rsha*. We create a mask of sign-bit by *lsh* of one. We AND sign mask with the value to extract the sign bit. If the number was positive, the result of this operation will be zero and we are done. It was negative (i.e. the result of AND is nonzero), we OR the value with mask created by subtracting one from our sign-bit mask - such a mask masks all non-sign bits. So even though our values by default use an unsigned extension, we can emulate a signed integer extension using this gadget.

Besides addition, bit operations and shifts, many modern CPUs also provide integer multiplication, division, and modulo operation[7]. To support those, we define *mul* and *div* operation to our intermediate instruction set. Both of those instructions are again unsigned. We will discuss the possible implementation of signed multiplication later in this section. The same applies to the modulo (*mod*) operation which we do not define, but which can be implemented using subtraction of value divided and multiplied back by the divisor.

Note that, unlike other operations, the division is not well-defined for all possible inputs. Namely division by zero is undefined. Many CPU architectures address this problem by raising division by zero exception which forces the programmer to manually check for zeros and never allow such an operation.

For obvious reasons our expression model does not allow any exceptions to be raised as we simply have not defined any exception handling. Consequently, we have to find a different way to handle division by zero. Fortunately, there exists a CPU architecture that also decided not to raise division by zero exceptions and that is RISC-V. RISC-V authors argue [12] that it does not make sense to have one and only special arithmetic instruction[8] (division) which would produce an exception. Arguably authors of RISC-V also did not want to design the whole exception model because of a single arithmetic exception. So they instead decided to extend *div* definition in a way that they defined the result of division by zero. In RISC-V, division by zero always returns a value filled with ones (i.e. the highest possible unsigned integer value or signed integer value of minus one). As this approach is more suitable in our model than raising an exception, we use it and define the result of division by zero as a value full of ones.

Analogously, the RISC-V manual defines the result of modulo by zero as the value divided. This definition also suits our definition of modulo as it is true that any value divided by zero is all ones, but all ones multiplied by zero are zero. So we end up subtracting zero from the divided value which always produces the divided value itself. So our modulo is consistent with RISC-V even though that would not be necessary.

This behaviour will not be limited even if we will emulate CPU with division by zero exception. On such a CPU we would have to special-case division by zero using the conditional operator. If the divisor were zero, we would emulate the no-op by setting output registers to their initial value (the value which was stored in them before instruction execution). We could then emulate the exception handler invocation using jump instruction which we will define in subsection 2.2.4. Consequently, such a design choice is not limiting for us in any way.

Using unsigned *mul* and *div*, we can implement signed *mul* and *div*. Let's start with multiplication. First of all, it is important to point out that multiplication on

[7]This does not apply to all CPU architectures. For example in RISC-V, multiplication, division, and modulo are considered to be an optional extension.

[8]Technically modulo as well, but those two are internally the same instructions – division operation produces modulo as a by-produce.

k-bit values which produces k-bit result produces the same result in both signed and unsigned multiplication [13]. The situation changes once we produce a $2k$-bit result as in those upper k-bits the result of signed and unsigned multiplication differs. This observation has an interesting consequence that we are for the first time (and also for the last time as well) in a situation when we need a mismatch between input and output operand widths in order to meaningfully define our operation.

On the other hand, this simple fact about multiplication gives us also an algorithm to implement signed multiplication. It is sufficient to sign-extend (we already described how to do that) both input values to double their *width* and perform an unsigned multiplication with those double *width* expressions. Alternatively, we could implement signed multiplication as unsigned multiplication followed by sign resolution conditions. Given that the first solution is more memory efficient we will use that one even though it might be slightly harder to understand it.

Signed *div* and *mod* are then implemented as unsigned *div* and *mod* followed by a sign resolution. We have already described sign resolution logic in the sign extension algorithms and we described how to negate integers in the subtraction description. So it is obvious how would we implement the sign resolution logic for those operations.

2.2.4 Representation of jumps

Jumps are in a way special instructions because, unlike other instructions, they affect instruction flow. Another irregularity of jumps compared to other instruction types is that jumps might be conditional. Conditional jumps are jumps that apply only under certain conditions. The condition might be either a value of a specific bit in *FLAGS* register (which is the case for example in the x86 instruction set) or it might be a more complex comparison for example in the case of RISC-V instruction set where the condition is integer comparison. Those two properties of jumps make them challenging to represent in our intermediate code.

Before we start designing jump instructions, we should state a few facts about jumps that will affect our final design. First of all, it is important to note that we have to be able to recognize jump instructions in our intermediate code as we will need to track control flow dependencies (i.e. which instruction belongs to which basic block) in our disassembler. Second, when we take a look at real CPU architectures, there are typically multiple ways to specify the address of the jump. There are jumps to a constant address where the jump address is just written in the instruction. There are also jumps to an address specified in a register. And last but not least, there is a mixture of those two. Those are instructions such as "read the value of register x, add immediate constant and jump to the resulting address". All those jump kinds imply that our jump instruction should accept a generic expression as jump address to give us enough freedom to describe all those jump instruction variants. The third fact about jumps is that we can understand jump as a write to instruction pointer register. In many architectures, the instruction pointer register is on purpose not readable or writable by any general-purpose instruction. Instead, special instructions (jumps and some sort of instruction pointer read) are provided to modify and read the instruction pointer register. The motivation behind this is that the CPU has to be able to recognize the jumps and reads of the instruction pointer to optimize the instruction execution. But even though instruction pointer writes are special in execution optimization way, they are in the end just writes to a register. The only difference is that the value of this specific register says to the CPU where to fetch the next instruction from.

With all this knowledge about jumps, we can design a jump instruction as a simple write to instruction pointer register. To do this, our intermediate code has to define a "magic" *key* string of a register which is understood as an instruction pointer by the whole code. Any jump will then write this instruction pointer register value using a register store effect. This construction allows us to use an arbitrarily complex expression as a jump address, which gives us enough freedom to describe register loads, constant addresses, or any sort of calculation of the jump destination address. The "magic" *key* then makes jumps recognizable in the code, which allows us to find basic blocks in the code.

The construction described above obviously works for unconditional jumps, but it might not be obvious how an unconditional write can be used to represent a conditional jump. The important fact here is that if a conditional jump is not taken, the value of the instruction pointer still changes. It changes in the same way as it changes with any non-jump instruction – the length of the instruction is added to the instruction pointer register. As both instruction address and length of the instruction are known at (de-)compile-time, we can express the new value of the instruction pointer in case a conditional jump is not taken as a single constant calculated as the address of the instruction plus the length of the instruction. Consequently, we can use a conditional expression to decide whether the address written to the instruction pointer will be the actual address the conditional jump jumps to or if it will be just the constant address of the following instruction. This way, we reduced the problem of a conditional jump to a single conditional expression and an unconditional register write.

It might be surprising that we use a "magic" register name. One might introduce the third effect representing a jump. The reasoning behind this is again the simplicity of the intermediate code. If we introduced the third effect, every transformation in the disassembler would have to know about its existence and special-case it. The fact that we use register write to a magic register allows any part of the disassembler logic which is not interested in jumps to ignore their existence at all. Besides that, there is no real reason to introduce a special effect. As we explained above, a write to the instruction pointer register is still just a write to a register. The reason to call jump instruction jumps rather than write to a register is first to make the optimization simpler for the CPU and second to build an abstraction for humans. As we do not care about super-efficient handling (out-of-order execution, branch prediction) we can use simple register write and keep the intermediate code simpler.

The instruction pointer register will have one additional property which is not usual for standard registers - reading of it will be prohibited. Our motivation to do so is first that we do not need such read at all and second existence of instruction pointer reads would complicate the future instruction reordering algorithms significantly (more on this in subsection 5.2.2). The reason why we can avoid using instruction pointer loads is that as we parse the machine code from the binary, we always know at which memory address the instruction we parse is. Consequently, we can substitute the value of the instruction pointer by a constant value containing the address of the instruction.

So far, we have ignored one aspect of jumps that exists in some CPU architectures. This aspect of jumps is called branch delay slot. A branch delay slot is a concept that can be found especially in MIPS processors nowadays and which core idea is that the jump instruction effect is postponed for k following instructions. In other words, in MIPS even if the jump is taken, the instruction immediately following the jump instruction is still executed unconditionally.

It is also worth noting that branch delay slots might cause weird corner cases. For example, we would have to define what happens if we perform another jump in a branch delay slot. Or what is the value read from the instruction pointer register in the branch delay slot. The existence of all those weird states is most likely one of the reasons why for example RISC-V decided not to implement branch delay slots.

Moreover, there are not many CPU architectures using branch delay slots nowadays. The x86 and AMD64 architectures are historically rigid and jumps take effect immediately in those architectures. The situation is very similar with both modern CPU architectures - ARM and RISC-V[9]. So the only CPU mainstream architecture with branch delay slots is MIPS.

It is true that our design of jump instructions is not able to describe jump delay slot at all. This is one of the tradeoffs between simplicity and genericity in this thesis. The main reason for this tradeoff was the scope of this thesis. The introduction of branch delay slots complicates not just the representation of jumps but also makes any further analysis way more complex. The overall growth of complexity exceeded the scope of this thesis.

Even though branch delay slots are difficult to represent and we do not need them to disassemble the vast majority of existing CPU architectures, it is still worth further development. Ideally, the jump representation should be reworked to enable the representation of branch delay slots. This change will affect most parts of the disassembler, but the added complexity seems to be necessary to have a truly generic intermediate code representation.

2.3 Instruction representation

Finally, we have enough concepts to represent real CPU instructions. In this section, we will discuss how we will represent individual instructions and all the issues related to that.

To represent a single instruction, we will need to describe all its effects. To do so, we will use a list of effects to represent a single instruction. Each one of those effects will represent a single value change. For example, if our instruction writes a single register and modifies *FLAGS* register, it will be represented by two effects.

[9]It is true that there is a modification of RISC-V architecture which has branch delay slots. Given that the RISC-V architecture described in the official manual has no branch delay slots, it still seems to be appropriate to say that RISC-V has no branch delay slots.

2.3.1 Order of effects

In our model of execution, the effects of a single instruction will not interact one with another. For example, register load of register x_1 in the second effect will not read the value the first effect stored to the register x_1. In the model of execution we will use, all effects and all their underlying expressions are evaluated at once, and only then do all effects take place at once. This implies that all register and memory loads refer to values in those registers and memory which were set by previous instructions.

This semantic of not affecting one another is not limiting as if we need to use some intermediate result multiple times, we can simply reuse the expression calculating it in multiple places. But this isolation of expressions is also important to keep the semantic as simple as possible. If we allowed one effect of the same instruction to affect another effect, we would then have to carefully track those effects and their respective order. In the end, the whole idea of composing functional trees would be in a way violated as those trees would affect one another for no reason.

Even though our expression trees are independent, we still have to reason about the result of multiple effects modifying the same storage. Our first idea could be that we can simply declare this as invalid or undefined usage and not to address this problem at all. Unfortunately, this is true only for registers where we use strings determined during the effect creation. In other words, we know which register will be written by which effect.

The situation is different with memory stores as those accept an expression to identify the address of a memory store. And here comes the problem: We are not able to guarantee that two calculations of memory address will not overlap one another. The value of expressions might be the run-time property of the program which we are not able to analyze in compile-time (or de-compile-time) in any way.

Consequently, even if we try not to, sometimes our effects collide with each other. For this reason, we have to define that effects apply in the order in which they are written in the effects array. There is no scientific reason why to pick exactly this order besides the fact that this order feels the most natural to people. Even though we allow effects to collide and the result of such execution is well-defined, it is still highly recommended to avoid effect collisions as much as possible. Especially because of the ambiguity it brings into the instruction meaning.

2.3.2 Special instructions

We tried hard to find an intermediate code model which would be able to capture any effects of instructions. Unfortunately in a real machine code, there are still instructions with effects that we are not able to describe in our intermediate code. For those instructions, we will have to introduce a different representation.

A good example of such special instructions are atomic instructions. In our model, atomic instructions are just simple memory writes. But in a real CPU, atomic instructions have a special meaning in instruction ordering. Atomic operations typically prohibit both the compiler and the CPU to reorder other memory accesses before or after the atomic instruction. The specific ordering constraints are very much platform-dependent. This fact makes it hard to introduce any generic description of reordering constraints.

There are indeed some memory models [14] of programming languages that are surprisingly compatible with each other[10]. Unfortunately, those memory models are usually much more generic than the real memory model of the CPU. This implies that even if the program we are disassembling is a program written in some programming language with a memory model, most of the memory order information disappeared during the compilation process. The reason is that due to the higher genericity of the programming language memory model multiple memory order rules can result in the same machine code. Consequently, there is no machine-independent way to recover the original memory order described in the program from the machine code.

As we are not able to describe memory order reordering constraints, we have to be conservative and mark the instruction as something special that we do not understand. For this purpose, we introduce a whole new concept of instruction special types, which is a concept absolutely orthogonal to the intermediate code concept with expression and effects. An instruction special type is just a marker saying "this instruction does something special".

We represent instruction special type as a set of bit flags. For this purpose, we add a single field to every instruction and by setting certain bits, we devote which special classes the instruction belongs to[11]. For the time being, we have 3 special instruction type classes:

1. Memory order
2. Syscall
3. CPU State change

As we indicate each class as an individual bit flag, it is possible to say that a single instruction belongs to multiple special classes. This is important as there exists real instruction which fits multiple classes. A good example of such instruction is x86/AMD64 instruction WRMSR which is a write to MSR register, so it is a CPU state change instruction. But the CPU manual [7] also states that the instruction is serializing which in x86/AMD64 means that the instruction also imposes memory order.

The other two classes are syscall and CPU state change. By a syscall, we refer to any sort of OS code invocation. Besides explicit syscall instruction, this might be also invocation of some special interrupt or any other agreed way to invoke kernel action. For example in Linux, there exists a convention that an interrupt $0x80$ on x86 and AMD64 can be used to invoke a syscall.

The reason why we cannot represent syscalls in our intermediate code is that the change of program state is OS specific. In theory, the kernel can change any memory and any register of the running program. It is also worth noting that there are syscalls that do so. For example Linux syscall *read* changes both the memory (where it writes the value read) and a few registers (where it writes information as error code or length of the content read). To sum it up, we would

[10]Memory models of the majority of programming languages comply with the C/C++ memory model.

[11]As our special instruction representation is a set of bit flags, the zero value of it naturally describes an instruction that is not special at all.

need to either have a very accurate emulation of the real kernel or we would have to make syscall dependent on all registers and all memory bytes. Because both those options are very complicated, we just say that it is a syscall and that we cannot describe exactly what will happen to the program state.

The situation is even more complicated with CPU state changes. By a CPU state change, we refer to any change of platform-specific register which might affect code interpretation. Such a change is typically some change in the control registry. A good example of this is the write to a control register *%CR0* in x86/AMD64 architecture. A write to this register can for example enable paging or it can disable cache. Obviously, it is not possible to describe those side effects in our intermediate code, so we need to say that we are not able to describe the side effects of such instructions at all.

2.4 Summary

- Values used by the intermediate code are unsigned integers. Each value has its *width* which describes how many bytes it represents. All higher than *width* bytes are considered to be zeros. This implies that the intermediate code uses an unsigned integer extension.

- Values can be passed between instructions either in registers or in memory. Registers are identified by names while memory is identified by the address (which is an intermediate code value). Registers could be represented using memory as well.

- Each expression has an arbitrary number of inputs, but it always produces a single output. Expressions can be composed to form a tree-like structure where the child expression produces the value consumed by the parent expression. Using expression composition, we can represent more complex operations for example sign extension. The immutable nature of expressions allow to share subtrees.

- There are five basic expressions:
  1. Constant load
  2. Register load
  3. Memory load
  4. Less-than comparison (condition)
  5. Binary operator with 6 kinds: *nand*, *add*, *lsh*, *rsh*, *mul*, and *div*.

- Expressions are consumed by effects. Effects describe the side effects of a machine-code instruction. Each effect changes one external value. We have two effects:
  1. Register store
  2. Memory store

- Instruction effects are represented using an ordered set of effects. The order of those matters when multiple expressions write the same value. In such a case, the last effect wins.

- Properties that cannot be captured by the expression model are marked using a set of bits. We recognize three special instruction classes:
    1. Memory order
    2. Syscall
    3. CPU State change
- The structure representing all properties of instruction then looks like this in *mltwist*:

```
type Instruction struct {
    Effects      []Effect
    SpecialType  uint64
}
```

3. Machine code parsing

In the previous chapter, we have introduced a model of instruction representation which we will use in our disassembler. In this chapter, we will discuss how to convert instructions of a real CPU architecture into our intermediate representation. We will discuss all possible problems related to that and at the end of this chapter, we will use all this knowledge to parse RISC-V machine code.

3.1 Executable file parsing

The input of our disassembler is an executable file with a binary code. So the very first job our disassembler has to do is to read this file and convert instruction into the intermediate code representation. To do so, several steps have to take place. First of all, we need to read the content of an executable file and identify those bytes which represent executable code. Second, the bytes representing an executable code have to be sequenced to individual machine code instructions. And last but not least, those instructions have to be parsed (read converted to intermediate code).

The first part is simple as there are existing libraries to parse both ELF and .exe formats. In *mltwist* written for the purposes of this thesis, we limit ourselves to ELF file parsing. *Mltwist* is on purpose very simple in this part as executable file parsing and all the problems related to that are out of the scope of this thesis. So just for sake of completeness, we will say that both ELF and .exe files contain some headers which describe ranges of executable code. We will read this information to identify byte ranges with an executable code and we will pass them further to the machine code parsing setup.

The machine code parsing setup is the more interesting part from point of view of this thesis. It consists of two parts. The first part is a generic sequencer that applies the same logic to any CPU architecture. The more complex part is the platform-specific instruction parser which does the instruction parsing itself.

This separation of responsibilities into a platform-independent sequencer and platform-dependent parser is important for two reasons. First, we want to make our parser pluggable, so the sequencer in a way plays the role of an adapter in between the instruction parser interface and the rest of the disassembler. And second, the extraction of instruction bytes sequencing logic lowers the complexity of platform-specific parsers.

3.1.1 Machine code parser interface

To make the disassembler extensible for other instruction sets, we have to define a generic interface between the sequencer and the parser. Ideally, we want to define our parser as context-free. In other words, we do not want our parser to take care of iterating instructions and blocks of code memory. Because of that, we will define our interface to parse a single machine code instruction and we will offload the block and instruction iteration logic into the sequencer.

One aspect of platform-dependent instruction parsing we have to deal with are different instruction lengths. Different CPU architectures can use different instruction lengths, or even a single CPU architecture might define multiple instructions of a different length. This is for example the case in RISC-V, where instructions can be either 4 or 2 bytes long. And as always, we have to mention x86 and AMD64 architectures. In both of those architectures a single instruction can span from 1 to 15 bytes [7].

Given this variability in instruction length, we define the interface of our parser to accept some number of bytes and to choose how many bytes it wants to parse as an instruction. This way, we leave the responsibility of picking the right byte length of the instruction parser which is the only component with sufficient (read platform-specific) knowledge to decide this. Once the instruction parsing logic identifies the length of the instruction and parses it, the parser moves to the next instruction. It can do so as the parser will tell it how long the instruction is, so it can adjust its cursor in the byte array appropriately.

We might define the interface so that it says how many bytes the instruction can have and to pass always only that many bytes. This would work, but all of this is unnecessary. We can simply pass the whole array of bytes that remains to be parsed and the parser will simply pick how many bytes it wants to parse. The only property of the byte array we pass into the parser is that the first byte passed to the parser is always a start byte of a new instruction.

Another aspect of the parser interface is what will the parser produce as its output. So far, we have been talking about an intermediate code representation and special instruction flags. Besides those, the parser has to tell the sequencer how many bytes it parsed as the sequencer has to move that many bytes forward to move to the next instruction. With this representation, we are already able to perform the parsing and the follow-up instruction analysis.

Besides the fields mentioned above, we need the parser to return two additional pieces of information which the disassembler will use to visualize instructions to the user. Those two properties are the name of the instruction and its representation in assembly code for the given architecture[1]. We need the parser to provide both those strings as they both follow CPU architecture-specific conventions. This fact is obvious in the case of instruction names. What might be less obvious is why cannot we use the intermediate code to generate the instruction representation in assembly code. The reasoning behind this is that the assembly code of every architecture defines a different way of describing instruction. As we want to make the code visualized to the user as familiar as possible, we need to use the architecture-specific string representation users are used to.

One might say that an assembly representation would be enough because we can deduce the instruction name by splitting the assembly representation at spaces and taking the first part. This is true for most of the architectures and most of the instructions. But there are exceptions. For example in x86 and AMD64, there are instructions as *repmovs* which is a single instruction formed by a *rep* prefix and the instruction itself.

To provide an example of the interface between sequencer and parser, we can present the data structure used by *mltwist*. Note that the interface was slightly modified compared to the real program to abstract unnecessary details (packages

[1]By an instruction name, we mean just the name for example *addi*. On the other hand by assembly code representation, we refer to the instruction with all its arguments. So for example $addi\%x_3, \%x_5, 4\%(x_4)$ which might represent addition of register x_5 to a memory address $x_4 + 4$

etc.):

```
type Parser interface {
    Parse(addr Addr, b []byte) (Instruction, error)
}

type Instruction struct {
    SpecialType    uint64
    ByteLen        Addr
    Effects        []Effect
    Name           string
    Representation string
}
```

3.1.2 The parsing process

When we want to parse some code sections of an ELF file, we have two ways of doing so. The first option is to simply iterate through all bytes in the section and parse them as instructions. This is simple, fast, and it works in some cases, but it has also some downsides we will discuss later in a detail. Alternatively, there is a piece of information about the entrypoint of an executable file. We might start with parsing the entrypoint instruction and then proceed with instructions following it, including jump targets. This way, we might parse all instructions "reachable" from the entrypoint. Both those approaches have their advantages and disadvantages which we will discuss further in this section.

The advantage of the first approach besides being simple is that it can parse all instructions in the section. This applies also to cases when the jump target address is not derivable during (de)compile-time. This might be the case with for example Position Independent Code (PIC) [15], where the address being jumped to might be read from the memory first and passed to the jump instruction in a register. There are also CPU architectures such as RISC-V which simply do not provide any variant of long jump besides jump to an address in a register. For those and some other reasons, the definition of reachability is not trivial and can never be considered 100% reliable.

It is true that by some more advanced analysis of sequences of instructions, we might be able to conclude which address has been loaded to the register we use for jumps. In the case of PIT, the address has been loaded from Global Offset Table (GOT) which is part of the binary. In the case of RISC-V and similar architectures, the address is most likely a constant loaded a few instructions before. So by analyzing blocks of code, we might be able to deduce the majority of jump targets. We will discuss a very similar concept deeper in section 7.3. But to provide some summary here, there is a lot of work to be done before we would be able to perform such an analysis.

The biggest problem of the first approach is that there might not always be just machine code in the code segment. There are several possible reasons for that. First, the compiler tries to align all basic blocks to addresses divisible by some constant (typically 16 bytes). Usually, it uses NOP instructions to do so. But if the code will never be executed by the CPU, the compiler is technically allowed to put any bytes there even if those do not represent valid instructions. As those bytes will be never executed by the CPU, such a program is valid even though it contains invalid instructions in the code segment. Alternatively, it can also on purpose inject invalid opcode instructions in the machine code as a sort of safeguard. To give a practical case of when this happens, the Go compiler for RISC-V injects all-zero instruction to the end of a code segment. As instruction of all zeros is on purpose not valid in RISC-V, those zero bytes guard the instruction execution against leaving the code segment. The second reason why there could be invalid code in the code segment is that sometimes the compiler stores data close to the code in the code segment. For example in some embedded platforms, there exists a convention that constants are not loaded by an instruction (as there are no such instructions), but instead, they are stored close to the actual code and loaded by memory load. And, likely, those constants will not be valid machine code. For those reasons, it is problematic to parse the whole code section as machine code.

The other possible approach parsing only those instructions which are used has opposite properties compared to the approach of parsing the whole section. It always parses only those instructions which are executed[2]. On the other hand, it is non-trivial to say where all the jumps will go, and consequently, there would be blocks of code that would not be parsed by the parser without an advanced analysis following the parsing. This implies that we would have to interleave instruction parsing steps and steps of analysis to parse the code in a reasonable way. Moreover, even if we did our best, we would not be able to guarantee that all jump targets will be found. To remedy this, we would have to expose some way a user could say which bytes are code.

For all the complications caused by jump target analysis and interleaving in the case of the latter approach, we will implement the first approach where the whole code segment is parsed at the very beginning. I have to admit that this decision was not the best as it does not allow us to parse all executables. For example, it is sufficient to parse the code produced by *gcc*, but as mentioned above, it is not able to parse RISC-V machine code produced by the Go compiler. Given that having a perfect instruction parsing algorithm is not the main goal of this thesis, we can consider those limitations acceptable for *mltwist*.

3.2 Instruction opcode matching

Another problem we face when we start to parse real CPU instructions is that we have to be able to identify an instruction opcode. Real CPU instructions are always represented as a sequence of bytes, in which some bits have the meaning of opcodes, other bits have a meaning of register number, other encode an immediate value and some might have a special meaning depending on an instruction opcode. Moreover, in some architectures (x86, AMD64), instructions have different numbers of bytes which makes instruction parsing harder as well. All those

[2]Some machine code might contain branches that will be never taken and in them, there could be invalid machine code. This might be the case for malware as such a technique could make it harder to analyze the code. But analysis of such a code is far behind the scope of this thesis so we will ignore such programs for now.

aspects of instruction encoding have to be taken into account when we design the instruction parser.

The bits describing the opcode can be anywhere in the instruction encoding. It is true that in every CPU architecture, there is some sort of regularity in which bits are used for opcodes, which are used for registers, and which for immediate values. But as there exist many instructions with different requirements on register counts and immediate values, the architecture typically has many irregularities in the instruction encoding. So even though the majority of CPU architectures define some set of bits that always represent opcodes, the other bits of instruction might have a meaning of register number, immediate value or any other different meaning which is defined specifically for that single instruction. Those few bits which are always defined to be part of the opcode then define the overall layout of the instruction encoding bits.

Because of those irregularities, we need a way to describe which bits are part of an opcode and which of them are not. We can represent this information using two arrays. The first array will represent the instruction opcode and the second array will then have bits set to ones in positions where the opcode bits are[3]. Let's call those two arrays *bits* and *mask* respectively. You can see this format visualized in Figure 3.1.

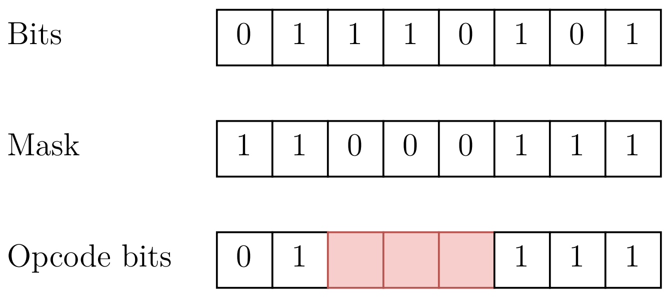

Figure 3.1: Visualization of instruction opcode representation

To enable parsing of instructions with variable length, we can define both *bits* and *mask* arrays as variable length arrays. The only condition those two have to fulfill is that they both have to have the same length. Obviously, the representation would not make sense if one of those arrays was shorter than the other.

For sake of completeness, we say that *bits* has to have zero bits in positions where there are zero bits in *mask*. This requirement simplifies the parsing logic as we will not have to AND *bits* and *mask* in every step of instruction parsing. But given that bits of *bits* at positions where *mask* is zero do not describe the opcode, this requirement is not limiting at all and it could be potentially omitted in the interface.

The opcode parsing process will then apply the following algorithm to every opcode known to the disassembler: First, the input machine code is ANDed with *mask*. Second, the result of AND is compared to the value of *bits*. If they are identical, we found the opcode. If they are not identical, we can start processing

[3]It is obvious that value of the first mask matters only in bit positions where there are ones in the second mask.

another opcode known to the compiler. This algorithm guarantees that if the instruction uses any known opcode to the disassembler, it will be matched.

At the first sight, it might not be obvious why the ANDed result cannot be ambiguous. The reason is that if that would be true, the CPU itself would not be able to recognize those 2 instructions. This is obvious for opcodes with identical bits because then those two opcodes would be identical. The only case we have to consider is that there exist opcodes *o1* and *o2* where all bits of *o1* are in opcode *o2* as well. But then the CPU itself would not be able to recognize opcodes *o1* and *o2* as any occurrence of *o2* would look as *o1* as well. In other words, any instruction opcode cannot be a bitwise subset of another opcode. This implies that any two opcodes always have to differ in at least one bit and that one bit has to be located in the intersection of their opcode bits masks. Consequently, the ANDed result can never be ambiguous.

To speed up the process of parsing, we might group opcodes by the same value of *mask*. This is beneficial for us because many CPUs follow some pattern in assigning opcode bits. By joining them into a single group, we can make the parsing more efficient because we will apply the *mask* to the input machine code just once to match all instructions in a given group.

Once we parse instruction opcode, it is fully up to the instruction-specific parsing logic to define which bits of instruction encoding are the registers, immediate values, and special bits. It is also fully up to the platform-specific logic to define how will those registers and values be used. We cannot unify this part of the code in any way as the meaning of instruction encodings is platform-dependent. The only part we were able to generalize is the opcode matching.

3.3 Expression gadget library

In subsection 2.2.3, we focused on making instructions of our intermediate code as simple as possible. We argued that such a design is not limiting because many other operations can be composed out of our intermediate code instructions. The statement back then was that any instruction parser can use gadgets we described to represent other common operations for example AND, subtraction, and sign extension. On the other hand, given that many of the gadgets described there are quite common instructions, it makes good sense to introduce a library which will provide all the gadgets. So on top of the intermediate code model, we also define a library that defines a set of gadgets implementing common operations.

The interface of such a library will be very simple. For every operation defined in a library, we define a single public function. The function accepts some input expressions (their count depends on the operation) and a desired *width* of the operation. The meaning of *width* is the same as for our intermediate code instructions – it describes *width* of all input expressions as well as *width* of the output[4]. The function then returns a single expression that represents the calculation.

This library allows us to shift more complexity from the machine code-specific parsers into a generic library. This is beneficial as it minimizes code duplication in instruction parsers. Another advantage the existence of this library brings is

[4]This statement does not apply to signed multiplication where the *width* of output is double the input *width*. The reasoning behind this has been described in section 2.2.3.

the definition of the ”standard” shape of gadgets for certain operations. This will allow us to build heuristic intermediate code transformation later in the code which will match those specific gadgets. And last but not least, this library makes human reasoning about those operations simpler. People are not used to thinking in terms of NANDs and hand-written sign extensions. By wrapping this complexity in a library, we make machine code parsers simpler to understand and reason about.

3.4 RISC-V Instruction parsing

Part of this thesis is to implement a proof-of-concept disassembler (*mltwist*) capable of parsing RISC-V code. Consequently, we have to implement a RISC-V instruction parser that transfers RISC-V instructions into an intermediate code representation. Even though we will discuss RISC-V parsing in this section, the majority of concepts we will explain here are trivially generalizable into any machine code parsing.

The reasoning behind the choice of RISC-V architecture for *mltwist* is the following: First of all, RISC-V is a simple RISC architecture which implies that there will not be that many instructions to parse as in the case of CISC-like architectures as x86. Second, the RISC-V specification is divided into independent modules. The manufacturer of any RISC-V CPU is required to include a set of basic instructions which provide arithmetics, comparisons, and branching. But many other instructions are defined as optional extensions which the CPU is not required to implement. This further lowers the number of instructions we need to implement as we can pick just some extensions and ignore other.

In *mltwist*, we parse the basic instruction set for both 32bit and 64bit architecture (RV32I and RV64I). On top of that, we support RV32M and RV64M extensions which define multiplication and division instructions for 32bit and 64bit architecture respectively. This extension was necessary to include as there are surprisingly few useful programs that would not need to multiply numbers. The last extension the disassembler parses is RV32A and RV64A. This extension covers atomic instructions. Parsing of atomic instruction is necessary because runtimes of many programming languages including both *glibc* and *musl* use atomic instructions internally and it is not possible to compile them without atomic instruction support.

3.4.1 Intermediate value parsing

RISC-V manual defines 6 types of immediate value formats that can be used in RISC-V instructions where one of those types is ”no immediate value”. It makes little sense to describe their exact formats in this thesis. Feel free to refer to the RISC-V manual [12] for more information on this topic. The only fact relevant to this thesis is that for every instruction opcode defined in our parser, we specify an immediate value type as an enum value.

When we parse an instruction with a non-empty immediate value, we see all the bits encoding the instruction including the immediate value. Consequently, based on the immediate value type, we can extract the immediate value from it.

As we know the value during the de-compile time, we can use our intermediate code constant to represent the immediate value in an intermediate code.

An immediate value of instruction is typically somehow mixed with other input parameters of the instruction to produce the desired side effect. The exact way how immediate value interacts with other parameters depends on the opcode itself. But no matter what the interaction does, we can use our binary instruction to represent any interaction of input value and the immediate value in our intermediate code.

3.4.2 Register parsing

The RISC-V architecture defines 32 general purpose registers which instructions operate on. Out of those 32 registers, 31 are general purpose registers capable of storing any value. The 32nd register is the register x_0 (register with number zero) which is hardcoded to be zero. This means that any write to x_0 is ignored and any load from x_0 always returns a zero value.

All RISC-V instructions then have at most two input and a single output register. Those registers are always encoded by 5 bits (as we have 32 registers) and their position in instruction bits is given by the specification itself. In other words, register numbers are always in the same position for all instructions. This is a nice difference compared to other architectures where the register position is dependent on an opcode. In RISC-V, if an instruction has as output register, it is always encoded in the same set of bits. And the same applies to both input registers. Moreover, the RISC-V architecture defines that instructions with a single input register always use the first input register.

The fixed position of register encoding in instruction is very convenient for us as the only two pieces of information, we need to know about any opcode is whether it has an output register and how many input registers (0, 1 or 2) it has. Based on those 2 fields, we can identify both input and output register numbers by simple bit-indexing in the instruction encoding.

In our intermediate code, we represent input registers using register load expressions and we represent output register using a register store effect. This rule applies to all registers but one - the x_0 register. As this register ignores any write into it, it does not make any sense to write into it using the register store at all. Consequently, any register store to x_0 is dropped in the parser. The same applies to register loads from x_0 as we are certain that the value read will be always zero. So we replace all loads from register x_0 by zero constants in our intermediate code.

3.4.3 Opcode representation in intermediate code

All side effects of instruction as well as the way both immediate values and registers are treated are given by an instruction opcode itself. Consequently, the most crucial part of the transformation of machine code into an intermediate code is the way we describe the meaning of the opcode. This will also be the least organized part of the parser as opcodes represent many different operations and it is not easy to find any regularities in them.

For every opcode, we specify a function that receives bytes of a machine code instruction and produces an array of effects. Based on instruction bytes and other information about the opcode (number of registers, intermediate value encoding), the function can identify all its input and output values. The code of the function is then supposed to build an expression tree that represents the given machine code instruction.

Given that the function is specific to a given opcode, it knows how many effects the machine code instruction has and how are those effects interconnected with its inputs. Having all this information, this function gathers all input values, composes them using either gadget library functions or raw expressions and produces effects representing a given instruction.

One aspect of machine code parsing which was not yet mentioned is multi-threaded behaviour of instructions. For example, when we try to represent atomic instructions, we can describe exactly what those instructions do, but we cannot say: ”This subtree of expressions is atomic” even though the machine code definition of the opcode states that. This is in reality not an issue as we do not care of parallel execution at all. This approach is identical to the way compilers treat the code as they always optimize single-threaded execution only. The only handling of atomic instructions will be marking them as special which we will discuss in the following section.

3.4.4 Special RISC-V instructions

As any architecture, RISC-V also defines a few special instructions. We can say that RISC-V has all 3 kinds of special instructions we defined. It has atomic instruction in RV32A and RV64A extensions. The basic instruction set defines both a syscall instruction and instruction to read and write CSRs. A CSR is a register that in some way affects the CPU execution, but its meaning is fully platform-dependent.

To represent all those special instructions, we add a special instruction type to every opcode we parse. The parsing logic will just forward the value of this field from the opcode definition to an instruction definition. This way we will guarantee that special instructions will be understood as special by the disassembler.

3.5 Output of parsing process

Until now we have defined only the interface between the sequencer and the parser, but we have not spoken about the sequencer output. In this section, we will discuss the output of the sequencer which will then be an input for the rest of our disassembling process. It is worth noting that the rest of the disassembler depends solely on the information provided to it by the sequencer.

We have already discussed some of the output fields. Obviously, both the list of effects and the set of bit flags indicating special instruction properties will be part of the input. From previous sections, we also know that the rest of the disassembler needs to know the instruction name and string representation, so it is obvious that those will be part of the sequencer output as well.

Besides those properties we already know, the sequencer output will contain two additional pieces of information. Those fields will be induced by the sequences rather than the parse. The first field is a byte array containing instruction bytes and the second field is an address of the instruction. We will not use instruction bytes for any analysis as their meaning is very platform specific and unsuitable for any generic analysis in the later stages of the disassembler. But we still include this array as it could be useful for example to show the hex-dump of the code to the user. Moreover, if we ever extend the disassembler with an interface for plugins, the presence of this field might allow us to write platform-specific plugins which would be able to deduce more information about the instruction. The use-case for instruction addresses is significantly wider and we will use them in many places in the disassembler later in this thesis.

3.5.1 Representation of memory addresses

When we talk about memory address representation, we should clarify how will we represent memory addresses in our disassembler first. A natural option would be to use generic expressions which would allow us to represent an arbitrary address. This would be also well-aligned with the design of our memory load expression where we use a generic expression as an address. Unfortunately, an arbitrary expression is too generic to be used in later parts of the disassembler. We will have to find a different way to represent memory addresses to have a good balance between the generality of the representation and the specificity of the address, which we will need.

The problem with generic expression is that we are not able to use it for any computation. In the later stages of the disassembler, we will need to be able to compare address equality or to even sort based on the address. Those are properties that expressions cannot give us. The problem is that due to both register and memory loads, the value of expression might not be known at (de)compile time of the program.

Another natural option would be to use a constant expression exclusively to represent memory addresses. Alternatively, we might use any other kind of unlimited, unsigned (remember, memory addresses are unsigned) integer representation. As constants are always known to the disassembler, we would be able to define both equality and ordering for them. It also seems to be wise to use an unlimited data type. If we used any limited data type, there might be a future CPU architecture that will use addresses wider than the fixed number of bytes we chose to use.

On the other hand, we might still consider using some native CPU types. An obvious choice, in this case, is the biggest possible unsigned integer type the programming language we use provides. We would indeed lose some generality in the solution, but the question is whether the benefits outweigh the downsides.

The benefit we would gain from the usage of a native type is speed and simplicity. Native types typically have better support in the language and they are significantly simpler to use in many contexts. Speed of execution with native types is also significantly higher as each arithmetic operation typically translates to a single CPU instruction. This is a significant speedup compared to any unlimited integer (including constant expressions) which are typically represented using an array. The array-like representation of unlimited integers implies that every single arithmetic operation has to be implemented using a loop iterating the array. What is even worse, any calculation producing a new value (addition, multiplication, etc.) has to dynamically allocate a new array to store the result which again decreases the overall performance. The last advantage of native types worth mentioning is that they consume significantly less memory than unlimited integers.

The only downside of native type usage could be that the disassembler will be limited in terms of how big addresses it can represent. For example, if we state that nowadays, the biggest unsigned type our programming language provides is *uint64*, then such a disassembler will not be able to represent addresses of 128bit (or even wider) CPU architectures. At first sight, such a choice could seem to be limiting the genericity and possible usage of our disassembler significantly. But there is one important fact to mention – there are no 128bit CPUs nowadays and they most likely will not be introduced in a near future [16]. Not mentioning the fact that 128bit CPUs would also need a 128bit kernel to operate them, which will also take some time to develop. So for the time being, usage of the native unsigned integer with 64 bits is generic enough to represent any program on earth.

One might argue that the fact that 128bit CPUs do not exist nowadays is irrelevant as there is an assumption that 128bit CPUs will exist one day and then *uint64* will not be sufficient anymore. Such an argument is true, but we have to first consider what would it take to adapt the disassembler once 128bit CPUs will be introduced. First, it is reasonable to assume that programming languages will add 128bit integer types much sooner than the first 128bit CPU will be produced. This makes sense because to program a 128bit kernel, we need 128bit integers. So our programming languages have to have 128bit data types then. Second, we do not have to use the *uint64* type explicitly in our code. Almost all programming languages allow to define some sort of typedef. We can then use a typedef to define our own type which will be for now *uint64*. Once our language will add support for 128bit type, we can update our addresses to *uint128* by single line change in our program. As we can see, program simplicity, performance speedup, and less memory consumption induced by usage of *uint64* is in the end compensated by a need for a future single-line change of the code. Such a tradeoff seems to be worth it.

To sum it up, we will use a 64bit unsigned integer to represent memory addresses in our program. To do so, we will typedef *uint64* as our own type called *addr*. Once the language we use adds support for *uint128*, we will update the typedef to *uint128*. If there will even exist 256bit (or higher) CPUs, we can trivially update the disassembler using the same single-line change once again.

3.6 Summary

- Parsing logic has two parts – *sequencer* and *parser*. *Sequencer* is generic and does not depend on any specific CPU architecture. It corresponds to the generic parser part in Figure 1.1. *Parser* is the only component of *mltwist* specific to a given CPU architecture (RISC-V).

- The purpose of *sequencer* is to iterate over code segments in the binary and to iterate over instruction bytes in those segments. It iteratively calls *parser* which parses individual instructions. Sequencer also adds some additional data (address of the instruction, its bytes) to the output of the parser.

- The purpose of *parser* is to match opcode of an instruction in bytes provided by *sequencer*. Then, the parser has to read the instruction encoding and understand the instruction – which registers it reads/writes, immediate values. Using this information, *parser* produces expression trees and effects describing the instruction meaning.
- Extension of *mltwist* for other CPU architectures is possible by providing another implementation of the *parser* interface. All other stages of *mltwist* are fully independent of a specific machine code.
- To simplify implementation of new *parsers*, we define a library implementing standard operations as *sub* or *sign extension* using expressions. Besides simplification, the other purpose of this library is to standardize expression trees of high-level operations.
- Addresses in the *mltwist* are represented using *uint64*. This will be extended to *uint128* once Go introduces *uint128* type.
- We implemented a parser for the RISC-V machine code.

4. Transformations of intermediate code

Before we start to design a deeper analysis of the intermediate code, we will define a few transformations of it. Those transformations will not change the meaning of the code, but they will simplify or alter our expressions. Using those transformations, we will be able to deduce more properties of the intermediate code. We will use those transformations as a building block of more advanced analysis in the later stages of the disassembler.

4.1 Motivation

The very first question one might ask is why would we need to transform or optimize intermediate code at all. We have some simplified representation of instruction side-effects and we can use it to deduce some information about the machine code instruction purpose. But it might not be obvious why we need to further modify the intermediate code. There are many answers to this question which we will discuss one by one.

First of all, a machine code parser does not always have to produce the simplest expression tree possible. The reason is that we want a parser to be simple as we will implement multiple machine code parsers for multiple CPU architectures. By keeping parsers as simple as possible and shifting the complexity into generic transformations applied to the intermediate code, we can lower the total complexity of the disassembler. Another reason to optimize intermediate code is that we on purpose designed the intermediate code to be simple and easy to both understand and optimize which we cannot say about machine code.

A good example of suboptimal code produced by a parser could be the RISC-V parsing of the following instruction $addx1, x0, x0$. This instruction adds register $x0$ to register $x0$ and stores the result to $x1$. In our RISC-V parser, the parser will understand that *add* adds two input values. It will identify both input registers and replace them with a constant zero expression (remember, $x0$ is hardcoded to be zero in RISC-V). But the parser will then produce an *add* expression adding those two constants together even though such addition has a constant result. The reason for this simple handling is that optimizing this special case would increase complexity. This is just a simple example, but there are other situations where identifying constant expressions would be much harder. It is worth mentioning that optimization of constants in a parser would be not just complex but also error-prone as it is quite hard to identify all possible inputs which produce a constant result in machine code. Instead, we can move this logic to an intermediate code transformation which will make it simpler and less error-prone thanks to the intermediate code simplicity.

The second reason to implement intermediate code optimizations is an optimization of the memory footprint of the disassembler. As we already know, each expression tree is formed by multiple sub-expressions which form a tree-like structure. Each of those expressions cost us some memory. So by optimizing the number of nodes (expressions), we can lower the total memory footprint of the disassembler significantly.

The last reason is that along the way, we will deduce more and more information about the intermediate code inputs. For example, if we have an instruction that loads the number 57 to a register and the following instruction reads the register, we can deduce that the value read is 57. By doing those deductions, we might generate more room for expression tree optimization during the disassembling process. This will be most visible in chapter 6 where we will leverage those transformations to fully emulate the program execution.

4.2 Expression modifications

Before we start to describe individual transformation algorithms, we have to describe general concepts of expression modifications. Those concepts will not be specific to any single transformation algorithm. Instead, those concepts will be mostly implied by the properties of our intermediate code.

The first fact worth reminding is that our expressions form a tree-like structure. As we discussed in subsection 2.2.1, each expression has some number (potentially zero) of input expressions and a single output. We can understand each expression as a tree node where all input expressions are child nodes of the node. Naturally, tree nodes with zero input expressions (i.e. constants and register loads) are leaves of the tree and the expression representing the output of all expressions combined is then the root.

As expressions are trees and trees are special-case of a graph, we will use graph algorithms to implement our transformations. Namely, we will use Depth First Search (DFS) [17] to iterate all sub-expressions from an expression root. Technically, we could use Breadth First Search (BFS) [18] as well. But as BFS would not give us any advantage compared to DFS and it is usually harder to implement than DFS, we rather choose DFS and keep our transformations as simple as possible.

So far, we have been talking about intermediate code transformations, but in subsection 2.2.1 we defined our expressions as immutable. This implies that we cannot truly transform an expression. What we can do instead is to create a new expression tree that represents the tree after the transformation is applied. This way of implementing modifications also comes from a functional programming world where this is usually the only way to express object transformation.

It might seem to be suboptimal regarding both memory consumption and time complexity that with every transformation, we will always allocate a whole new tree. Luckily we can introduce an optimization that will not eliminate the additional cost, but that can significantly lower it. The idea behind this optimization is that the immutability of expressions allows us to reuse them. So if a subtree (ultimately the whole expression) was not changed by the transformation, we do not have to allocate any new expression nodes. Instead, we can simply reuse existing nodes from the original tree.

4.3 List of transformations

In this section, we will list which intermediate code transformations we implement in our disassembler. We will discuss the motivation behind them as well as possible use-cases. And last but not least, we will describe how to implement those algorithms.

4.3.1 Constant Folding

The most useful optimization we will use is constant (sub-)expression evaluation which we will call constant folding. The idea behind constant folding is that if all input parameters of an expression are constant, we can substitute the expression with a constant as well. By recursive application of this simple principle, we can simplify our expression tree significantly. Ultimately, it is possible that our constant folding will be able to substitute the whole expression with a constant.

Before we start to discuss implementation details of constant folding, it makes sense to explain why constant folding will be the most important optimization in our de-compiler. The reason is that constants are in a way the only values we know at (de)compile-time. Every expression which is not a constant is a run-time value that is not known to us. So by constant-folding as many expressions as possible, we will be able to state as much as possible about the program. On the other hand, everything that is not a constant is an unknown value. Consequently, we could say that all the analysis described later will be in a way just about deducing more constants in the code and applying constant folding.

To describe the constant folding algorithm, we will describe how will we modify a single expression for every expression type. For constants, the constant fold operation is obviously no-op. The same applies to register load which we are not able to optimize as register load does not contain any expression and we do not know the value of the register during the constant folding. The situation becomes slightly more interesting with memory loads where we also cannot optimize it, but we can constant-fold the addressing expression. The property all those 3 expression types share is that they are never replaced by a constant during constant folding.

The expressions we can optimize are binary expressions and conditions (less-than comparisons). For every binary expression, we consider both input operands and if both are constants, we will calculate the value of a binary expression. To do so, we will emulate unsigned extension rules and arithmetic we defined in section 2.2.3. Using this approach, we can calculate the constant output of a binary expression and we can replace it with a constant. This process is visualized in Figure 4.1.

Constant folding of the less-than operator will be more complex because we have multiple expressions to optimize. First, we will constant-fold both arguments for the less-than comparison. If both expressions become constants, we can say whether the expression returned by the conditional operator will be the *true* or the *false* expression. Consequently, we will replace the conditional operator with a constant fold of either the *true* or the *false* expression respectively. If at least one of the less-than arguments is not a constant, we are not able to tell which branch will be taken. This means that we have to preserve the less-than instruction in

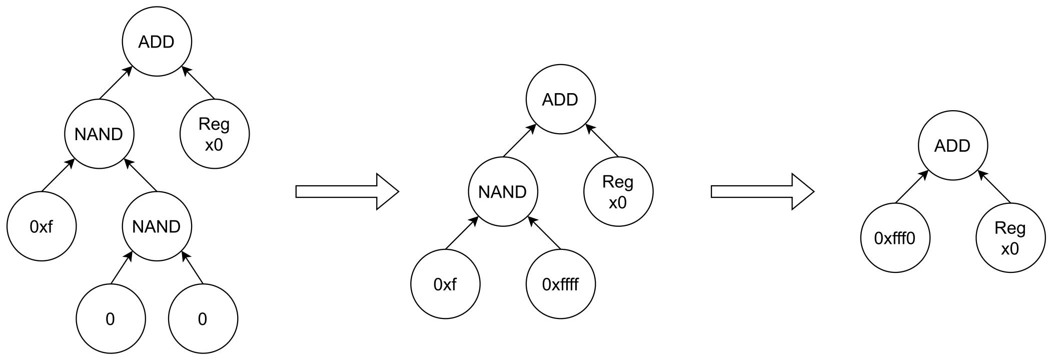

Figure 4.1: Visualization of constant folding of binary expressions.

the code. We can still constant fold both *true* and *false* expressions though.

One tricky part of the less-than expression optimization is the fact that we need to preserve expression *width*. Let's imagine an example when less-than comparison resulted in the *true* expression. So we replace the less-than expression with the *true* expression. But the width of *true* expression might differ from the *width* of the less-than expression. In such a case, we are not allowed to simply replace the less-than expression by the *true* expression directly as the difference in *widths* might change the overall meaning of an expression.

To remedy the problem with *width* when we drop a conditional operator, we introduce a *width gadget*. A *width gadget* is an agreed instruction used to represent an expression that does not perform any calculation but is supposed to just alter the expression *width*. In the case of *mltwist*, we define that such a *width gadget* is a binary operator performing addition of an expression and a constant zero. Obviously, arithmetics-wise, such an instruction is a no-op. This implies that we can replace the less-than operator with a *width gadget* to preserve the *width* effect of the less-than operation without changing the value of an expression. The way we use *width gadget* to replace conditional operator is shown in Figure 4.2.

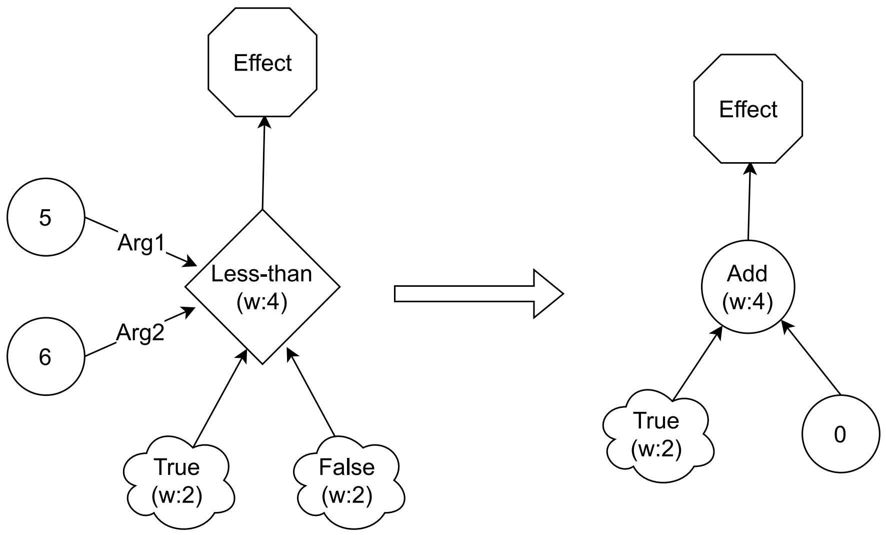

Figure 4.2: Visualization of constant folding of conditional (less-than) expressions.

The *width gadget* trick does not have to be used in all situations when we constant-fold a less-than expression. For example, if the expression we replace less-than with is a constant, we can change the constant *width* directly. We can also change *width* of a register load, but we have to be careful here. The problem with register loads is that we can only lower the *width* without any consequences. The reason is that if we read 4 bytes and then take the bottom two bytes loaded, the result is identical to a load of two bytes. On the other hand, if we load two bytes from a register and then zero-extend the value to 4 bytes, we cannot replace such an operation with a 4-byte register load as the register might contain nonzero bytes 2 and 3 (indexed from zero).

It is worth mentioning that we can never change *width* of a memory load as we might break dependency analysis by such a change. Unlike in the case of register load where we always consider the register to be read and the *width* defines only the *width* of output, in the case of memory load the *width* also defined how many bytes we will consider being read. So by shrinking *width* of a memory load, we might drop some dependencies which exist in the original machine code. We also for obvious reasons cannot directly change *width* of both binary and less-than expressions. Consequently, for those 3 instructions, we have to use the *width gadget* trick instead.

So far, we have described how will a single step for a single expression node look like. But we want to optimize the whole tree of expressions. To do so, we have to perform the steps described above in some step of the DFS algorithm. And the right place to perform those steps is on the return path. So we first walk down the tree and we do the actual work on the way back in the DFS. The reason for that is that there might be multiple levels of an expression that might collapse into a constant. Consequently, we need to constant fold the lower levels of an expression tree first to be able to say on the upper levels whether they are constant or not. Another reason to do work on the way back is the immutable nature of our instructions because we need to know all input values before we create the instruction itself. To sum it up, we will walk down the tree, and on a way back, we will perform all the steps described above, which will grant us to constant-fold as many nodes of an expression tree as possible.

The interface of the constant folding is simple:

```
func ConstantFold(ex Expression) Expression
```

4.3.2 Width gadget pruning

In constant folding, we defined a *width gadget* expression which does not have any arithmetical impact on an expression besides changing the *width*. The slight problem of using *width gadget* is that its usage might increase the expression complexity and memory footprint of the disassembler. As some *width gadgets* (actually the majority of them) are unnecessary, we introduce width gadget pruning transformation which will remove useless *width gadget* expressions from an expression tree.

Before we start to reason about the optimization algorithm itself, we should explain why it is useful to introduce a new optimization algorithm. Given that the only place where we add *width gadgets* is constant folding and we claim that some of them are useless, the question might be why we added them in the first place. And the reason is the contextuality of an algorithm. To perform constant folding, we do not need any context. We just give it an expression node and it produces a constant-folded variant of an expression. On the other hand, to say whether a *width gadget* is useless, we need to know the *width* of an expression consuming the *width gadget* as its input. In other words, we might say that to decide *width gadget* being useless, we need to have a context where we are in the tree and what consumes the output (i.e. context). Naturally, we might merge this logic with the constant folding. But given the different natures of those two transformations and to lower the complexity of the code, it made more sense to introduce *width gadget* pruning as a separate transformation.

A useless *width gadget* is a *width gadget* which changes *width* in the same way the operation consuming its output would do anyway. Let's have an example.

Let's have an expression a of *width* 1 byte which is an input of a *width gadget* b of *width* 2 bytes. If the *width gadget* output is consumed by an expression c of *width* 2 byes as well, the *width gadget* is unnecessary as the output of expression a would be extended to 2 bytes by the operation consuming it anyway. The same applies if the consuming expression c has *width* of 4 bytes as zero extension to 2 bytes followed by zero extension to 4 bytes is equivalent to zero extension to 4 bytes directly. The same logic applies to lowering an expression *width*.

For a better intuition please refer to Figure 4.3.

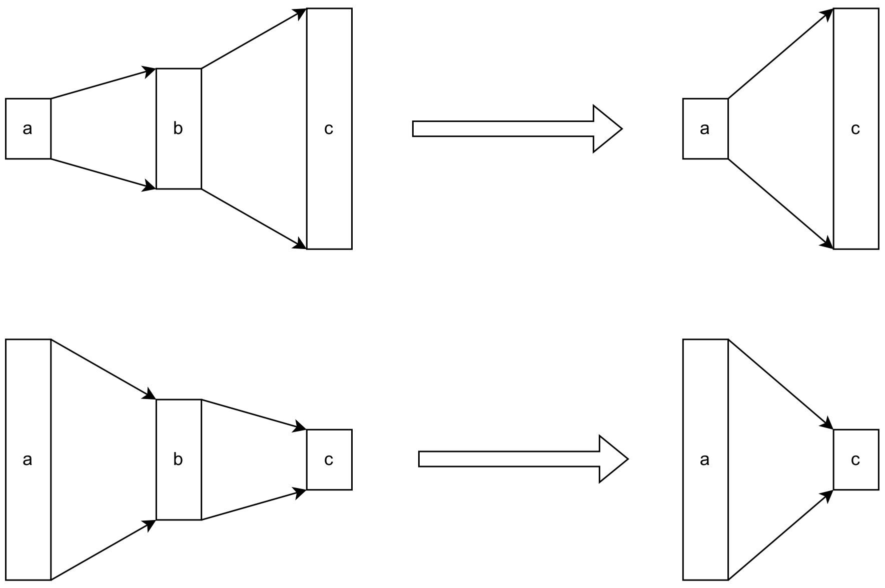

Figure 4.3: Visualization of monotonic *width gadget* pruning
Note that b is always a *width gadget* expression.

The situation gets slightly more complex when the *width gadget* changes *width* to either a smaller or greater value than both the original and the consuming expression. For example if *width* of a is 2 bytes, *width* of *width gadget* b is 4 bytes and width of consuming expression c is again 2 bytes. The other case happens if *width* of b would be just one byte. In the former case, we can drop the *width gadget*, but we cannot do so in the latter case. The reason is that *width gadget* increasing *width* which will be shrunken back (i.e. the former case) will zero extend the value with zeros which will be dropped by the consuming expression c. So by dropping the *width gadget* and not adding those zeros, we do not change the meaning of an expression as they would be dropped anyway. On the other hand in the latter case, the *width gadget* might drop some nonzero bytes which will be then replaced by zero bytes in an unsigned extension in the c expression. So this is the only case when a *width gadget* is useful. Both those cases are visualized in Figure 4.4.

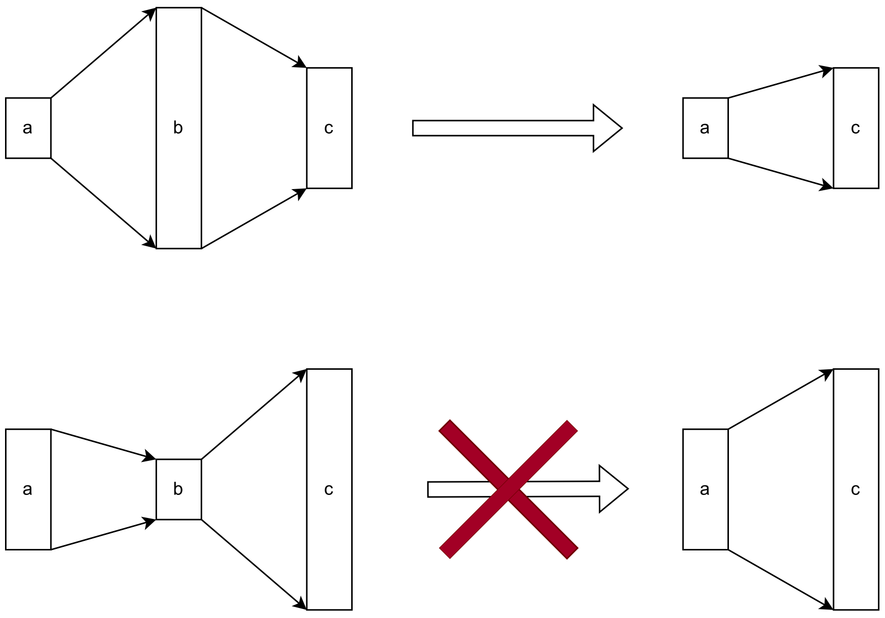

Figure 4.4: Visualization of non-monotonic *width gadget* pruning
Note that b is always a *width gadget* expression.

Similarly, as in the case of constant folding, we apply those steps as we return from the DFS walk. The only difference here is that if we decide to drop a width gadget, we have to re-check if the child expression is not a *width gadget* as well. If it is, we have to again check its usefulness and potentially drop it. The reason for this is that we might have multiple *width gadgets* in a row and by removing the top one, we might allow dropping the bottom one in the tree order. This happens in cases when the first (lower in the tree structure) gadget shrinks *width* and the second one extends it. By dropping the extending gadget, we might make the shrinking gadget unnecessary as well even though we were not allowed to drop it before according to our rules explained above. This fact slightly complicates the algorithms as we need to re-check lower expressions and potentially drop a *width gadget* one level below. On the other hand, checking one level back in the tree does not increase the asymptotic complexity of a walk.

It might not be obvious that it is always sufficient to check for useless width gadgets only one level back in the tree walk when we drop a width gadget. The first intuition might be that such a check can chain and bubble down the tree dropping all useless *width gadgets* along the way. The reason why this does not happen is that if there were 3 *width gadgets* in a row, the lower two checks would always form either ascending or descending sequence (or equal, but we can define the sequence of equal *width gadgets* as descending with no loss of generality). This fact is obvious as in a pair of values, there is always at least one element less or equal to the other. So one of those two lower *width gadgets* would be dropped by the algorithm before we start to process the third (topmost) width gadget.

Consequently, there can never be more than two *width gadgets* in a row and it is always sufficient to check *width gadget* one level lower in a tree.

The interface of width gadget pruning is identical to constant folding interface:

```
func PruneWidthGadgets(ex Expression) Expression
```

4.3.3 Conditional expression unwrapping

Sometimes, we care about all possible results of an expression rather than of the expression itself. This is especially the case in jump target analysis where we want to know a set of all possible values the jump expression might produce. It is possible that some of those possible values will not be constant (i.e. they will depend on a runtime constant). On the other hand, some of those values might be constants and we want to know those. For this reason, we introduce conditional expression unwrapping transformation which unwraps a single expression containing conditional expressions into a set of expressions with no conditional expressions in it.

First, we should discuss why we want to drop only conditional instructions and not other intermediate code instruction types. The reason is that by dropping conditional instructions in an expression, we might produce some expressions which we will be able to fully constant fold (i.e. constant fold to a single constant). This is not the case for any other instruction in our intermediate code. Both memory loads and register loads are always non-constant expressions. The constant expression is on the other hand always constant. A binary instruction is constant if and only if both its inputs are constants. It can also happen that one input of a binary expression is constant and the other one is not. In such a case we cannot say anything about the possible output as the output value always depends on both operands. To give an example $x + 5$ can be an arbitrary number depending on the value of x and we cannot say anything about the expression based on the knowledge that the other operand is 5. The argument with addition is stronger than arguments for other binary operation kinds but in all the cases we can say that the output still depends on both arguments in some way.

The key difference in conditional (less-than) instruction is that we can find some constant expressions even if not all input expressions are constant. This is given by the fact that even if one of the comparison arguments might be non-constant, either the *true* of *false* expression might be constant. This approach also makes sense as we care only about possible values of an expression, not about condition arguments as those do not contribute to the value. Both comparison arguments only contribute to picking which of the two possible expressions will be returned by the less-than expression. By picking both those possible values (*true* and *false*), we can ignore the comparison arguments at all.

A single step of this transformation will be slightly different than other transformations as instead of a single expression it will return a list of expressions[1]. The reason for this is that by replacing the less-than comparisons, we produce two expressions, and to represent them both, we need to return a list. This number is then further multiplied by more conditional expressions and binary expressions along the way. For conditional expression, we always concatenate the list of both *true* and *false* possibilities, which produces even longer list of expressions. But the biggest multiplication of output expressions comes from the binary operator, where each possible value of the first argument has to be paired with every possible value of the other argument. This produces a list containing all elements of a cartesian product of the first and second argument possibilities.

Because the conditional expression unwrapping returns a list of expressions, the the final interface of this function is:

```
func UnwrapConditions(ex Expression) []Expression
```

The other 3 expression types do not multiply the number of possibilities. Both constant and register load have no input expressions, so they always have only one possibility – themselves. Memory load consumes an addressing expression which on itself might have multiple possibilities. For each of those address possibilities, we create a single memory load using it. Consequently memory load does not increase the number of possibilities but it preserves its count.

To provide a better understanding of our conditional expression unwrapping step, let's describe individual transformations using python-style list notation. We use capital letters to represent intermediate code operations and lower-case letters to describe input expressions. An expression $f(x)$ then denotes a list of all possibilities of expression x.

1. Constant C: $C \rightarrow [C]$
2. Register load L: $L \rightarrow [L]$
3. Memory load L: $L(a) \rightarrow [L(x)\ for\ x\ in\ f(a)]$
4. Condition C: $C(a, b, t, f) \rightarrow f(t) + f(f)$
5. Binary expression O: $O(a, b) \rightarrow [O(x, y)\ for\ x\ in\ f(a)\ for\ y\ in\ f(b)]$

Even though we are producing all possible results of an expression, we want to eliminate all impossible results. In other words, if we know the result of a comparison, we do not want to use both *true* and *false* expressions but only the one which is taken by the condition. To achieve that, we will first apply constant folding to an expression to remove constant conditional expression. This approach again makes the algorithm simpler than if we were to take this one special case into account. Moreover, due to the multiplication property of less-than and binary instructions, it is wise to constant-fold the input expression first to limit the multiplication as much as possible. This avoids excessive memory consumption and unnecessary computational time usage in the disassembler.

4.4 Summary

- We use DFS to iterate the expression tree and transform it. Because of the immutable nature of expressions, each transformation has to build a new tree of expressions representing the transformed output. This approach is inspired by functional programming where this is the only way.

[1]Technically we could define the transformation to return a set of expressions. Duplicate values give us no additional information and neither does the number of duplicate expressions. The problem is that so far we have not defined equality of expressions. For this reason, we are technically not able to remove duplicate elements from the list to make it a set. This is not a problem because the worst possible case is that some expressions will be in the returned list multiple times.

- Constant-folding evaluates constant sub-trees of the expression tree and replaces them with constants. Ultimately, the constant-folding of an expression might result in a single constant.
- We define *width gadget* which is an expression with no arithmetic meaning. The only purpose of this expression is to change the *width*.
- Because the majority of *width gadgets* is useless in the code, we implement the width gadget pruning algorithm. It walks expression trees and eliminates useless *width gadgets*. The purpose of this transformation is not to change the expression meaning, but only to lower the expression complexity and memory footprint.
- Sometimes, we want to know all possible values the expression might produce. This is the purpose of conditional expression unwrapping. It removes all conditional (less-than) expressions from the expression tree while producing non-conditioned (i.e. straight) expressions.

5. Reordering of instructions

In this chapter, we will discuss the possibility of using our intermediate code and its transformations to find dependencies between instructions. We will then use instruction dependencies to deduce which instructions can and cannot be reordered. At the very end of this chapter, we will implement the instruction reordering algorithm.

5.1 Instruction dependencies tracking

There are 4 basic instruction dependency types [19]. These are: true (data) dependencies, anti-dependencies, output dependencies, and control dependencies. Two instructions are truly dependent on each other if the latter instruction consumes the output produced by the former instruction. This relation between instructions naturally implies that they cannot be reordered. Anti-dependency is a dependency between instructions that are not necessarily part of a single algorithm, but which use the same register or memory while the latter changes that value. If the first instruction consumes a value in a register (or memory) and the other instruction rewrites it by a new value, then obviously those two instructions cannot be reordered. Output dependency states that the final state of both memory and registers has to be preserved. In other words, the last instruction writing a register or a memory address must always be the last. If we reordered such instruction with another instruction writing the same memory or register, the output would be naturally different.

The last dependency type is control dependencies, which are in a way special. Control dependency is the dependency between all instructions and respective jump instructions which affect whether and how many times will be the instruction executed. The key difference between this dependency compared to other dependencies is the way to handle them. Unlike other dependencies, control dependencies are usually not tracked by a dependency graph but by splitting the code into so-called basic blocks.

5.1.1 Basic block analysis

The basic block is a term from compiler world representing a sequence of consecutive instructions in the program address space which always executes in a sequence. This definition of basic blocks has several implications. First of all, the first instruction of a basic block is either target of some jump in the program or it immediately follows jump instruction in a memory address space. Another property of a basic block is that no instruction but the first are the target of any jump in the program. And last but not least, the last instruction in the basic block is either a jump instruction or an instruction following it is the target of some jump in the program.

Basic block boundaries can be given by one additional cause which was not mentioned above. This cause is very technical and it does not give us any additional abstraction for other work with basic blocks. Yet we have to take this technicality into account during basic block finding. The technicality is that code does not have to be stored in a single section in memory. It can be spread in multiple code sections in the program address space, which do not have to follow each other. So the last cause for basic block start or end can be the start or end of a code section that is not preceded or followed by another code section in the program address space.

The algorithm to identify basic blocks in our disassembler will start with a single basic block spanning the whole program which we will divide into multiple basic blocks along the way. We will start by splitting on code segment boundaries followed by splitting on jump instructions. The last but most complex splitting will use jump target analysis to further split basic blocks on jump targets. We will discuss those 3 steps separately.

During the analysis description, we will assume that input of our basic block analysis is a list of instructions sorted by their address in the program address space. This can be trivially achieved by a simple sort called at the beginning of the analysis. This is possible as we know the addresses of instructions identified by the sequencer.

Steps to identify basic blocks

In our code segment boundary analysis, we do not care about all code segment boundaries. The only segment boundaries we care about are those where code segments do not follow each other. In other words, the boundaries we are looking for are spaces between instructions in our list. To do so, we iterate a list of instructions (which are already sorted by their in-memory address) and we check if the address of the following instruction is the address of the previous instruction plus the length of the previous instruction. In case it is not, we know that there is a space between segments of instructions and we split the basic block into two at the boundary of those two instructions. This way, we iterate through the whole list of instructions and find all basic blocks induced by differently located code segments.

Identification of jump instructions is also quite simple. We iterate over a list of basic blocks produced by the previous stage and find instructions that write to the instruction pointer register. In subsection 2.2.4, we defined a well-known name for such a register. Consequently, we can conveniently identify a jump instruction by iterating through its effects and checking if it writes to the instruction pointer register.

In reality, *mltwist* uses slightly different way of identifying jump instructions. It uses the conditional expression unwrapping transformation (4.3.3) to extract all possible values written to the instruction pointer register followed by constant folding of the possibilities. It then drops those jump targets which were folded to a constant equivalent to the memory address of the following instruction. If the list of possible jump targets remains non-empty after this filtering, the instruction is identified as a jump instruction.

The reason to use this algorithm rather than simply identify writes to the instruction pointer register is that it shows there exist jump instructions jumping exclusively to the following instruction. Such instructions make little sense, but they are for some reason present. Or there is at least one in *grep* compiled by *gcc* for RISC-V CPU. As such a jump is not really a jump and not splitting a basic block on such an instruction does not violate our definition of a basic block, we can ignore such a jump in our analysis. It is also true that we will have to perform a similar form of analysis for the jump target analysis anyway, so using this more complex algorithm here does not in reality cost us any additional computations.

The last and most complex way of splitting basic blocks is splitting based on jump target addresses. We will iterate through all the blocks we have so far and we will again look for jump instructions. But this time, we will not care about them being jumps, but we will care about all possible addresses where the jump jumps to.

To do so, we will use a similar approach to the one we described above for jump identification. We will identify which expressions are written to the instruction pointer register and we will list all possible values of the expression using conditional expression unwrapping (4.3.3). By constant folding each of those possibilities we will identify those jump targets which are constants and which are unknown at (de)compile-time. Only this time, we will drop all non-constant addresses as those depend on some runtime value. The reason for such filtering is that we are not able to deduce those jump addresses so we are not able to use them in the jump target analysis.

From the list of constant jump addresses, we will further filter away constant addresses which point to the address following the jump instruction. The reason for this is that in subsection 2.2.4, we used jump to an address following the instruction to represent the jump not taken in conditional jumps. Moreover, as we already split basic blocks on jumps in the previous step, it makes no sense to use those constants for splitting again (those blocks were already split). The result of this filtering will be jump addresses known to us. For each of those filtered constant jump addresses, we will find the basic block where it points and we will split it into two at this address.

To sum it up the process of basic block finding is the following:

1. Find boundaries between code segments (blocks of machine code).
2. Split blocks on all jump instructions.
3. Iterate all jumps and split blocks on jump targets.

By applying those 3 steps in a sequence, we will identify all the basic blocks we can identify at (de)compile-time.

Limitations of basic block analysis

It is important to note that those might be by far not all the blocks as we might miss some jump targets. When we described splitting based on jump targets, we dropped all non-constant expressions. The reason why we did so was that we were not able to say anything about them as we have not known their values. This implies that our basic block analysis is by its nature incomplete and there might be basic blocks and control dependencies that we will not see in our disassembler.

The root cause of basic block analysis incompleteness is that some expressions written to the instruction pointer might contain memory or register loads. It is true that in some cases we might be able to identify the value read. For example in RISC-V, the only way to perform a far jump (jump longer than 1MB) is encoded as register load instruction followed by a jump to a register value. As those are separate instructions, we are not able to deduce the value of register load in the jump instruction and consequently, we are not able to deduce the address this jump jumps to.

One might argue that such a simple case could be handled by some analysis of following instructions and replacing register loads with the previous register writes. Such a statement is true and we will discuss it in more detail in section 7.3. But to summarize the outcome, we might end up in a state where we might have to backtrack possible solutions, which is the reason why we do not do so.

Even if we were able to perform advanced analysis of basic blocks, we still would not be able to identify all basic blocks in the program. The reason is that even when we see the whole program, we are not able to predict how will it interact with the operating system, the CPU itself, or the rest of the environment. A good example of such a case could be the following C program:

```c
#include <time.h>

int main() {
    const void (*f)(void) = (void(*)(void))time(0);
    (*f)();
    return 0;
}
```

This program reads the current UNIX timestamp, casts it into a memory address, and calls a function with no arguments and no return value at that address. Naturally, this program does not do anything useful and it will most likely seg-fault any time it runs. But the important information here is that there are programs in which control dependencies might be runtime properties. Consequently, even if we do our best, we might not be able to find all control dependencies in such programs as they might be runtime properties of the program.

Useful programs typically do not jump to random addresses very often. We are indeed able to write such programs, but compilers typically do not produce such machine code. In compiler-generated code, almost all calls have a fixed address to which they jump to. This is true as the code generated by the compiler has a fixed address, so it has no reason to use any dynamic values for jumps.

On the other hand, even though the example above is very artificial, there might be also some legitimate reasons to write a program that jumps to an address known only at runtime. A good example of such a program might be the program that dynamically loads libraries at its runtime. The C runtime provides a function as *dlopen* which dynamically loads a library somewhere into an address space of the program. Internally, this library function uses *mmap* Linux syscall to map the shared library into the program memory. The program then uses this value (advanced of some offsets to the code) to jump to the library code and execute it. Another example where we would jump to run-time value might be the case of modern Just-In-Time (JIT) compilers. Those compilers typically allocate a chunk of memory and compile machine code into it. Once the compilation is done, they execute the compiled code by jumping to its address. But as the address is in a dynamically allocated block of memory, which address is given at run-time, the jump target itself is even a runtime property of the program. Both those are examples of (in a way) self-modifiable programs which are way beyond the scope of this thesis. It is just useful to point out that there exist legitimate programs which we are not able to fully analyze.

5.1.2 Analysis of dependencies in basic block

Once we have the code separated into basic blocks, we can start tracking data instruction dependencies. Naturally, we will track dependencies among a single basic block individually. The reason is that instructions in different basic blocks can no longer be reordered due to control dependencies between basic blocks and all instructions in it transitively.

Technically we could move instructions even between basic blocks. But to do so, we would have to perform a global analysis (read the analysis of all basic blocks in the program) to guarantee that such reordering will not change the program meaning. Unfortunately, as we will see later in section 5.2.2 that we are not even able to guarantee that we find all the basic blocks in the program. So for the time being, moving instructions between basic blocks is not possible.

Accepting a list of basic blocks as input of the dependency analysis also slightly modifies our definitions of both output and control dependency. In the single-block definition of output dependency, we will care about the outputs of each individual block rather than of output of the whole program. This makes good sense as each basic block has to return the same outputs in order not to change the program meaning.

It might be surprising that we are still discussing control dependencies among instructions in a single basic block. The reason for that is that the definition of basic blocks allows jump to be the last instruction in a basic block. For obvious reasons, the jump instruction cannot be moved in the block as it would interrupt the flow in the basic block. We can represent this fact by making the jump instruction dependent on all formal instructions in the basic block. As the jump can be only the last instruction of the basic block, there can be no other control dependencies in a single block.

The dependency analysis will iterate block by block and identify dependencies in them. For each basic block, the dependency analysis will iterate all instructions in the block in either forward or backward order (depending on the type of dependency). By applying our analyses, we will find all dependencies among instructions of a single basic block.

Handling of register and memory dependencies

The original reason why we introduced registers in subsection 2.1.3 even though memory would be sufficient for all applications was to simplify the dependency analysis. The issue with memory loads is that they consume other expressions as memory addresses. Because of this, it might happen that we do not know the address as it can be non-constant. This can happen if the address expression contains some register or memory loads. Such a situation might result in a state where we are not able to identify the memory address accessed by memory load or store by the expression. For this reason, there is a significant difference in the way we handle register and memory loads in our data dependency analysis.

Analysis of register loads is simple as registers are identified only by string constants. We can track register dependencies between instructions by simple string comparisons. Given that majority of dependencies between instructions are register dependencies, we could argue that the existence of register operations in our intermediate code (even though it is unnecessary) allows us to quickly and effectively track the majority of dependencies.

One might ask whether memory dependencies are as important to us. The majority of information among instructions in a single basic block should indeed be always passed in registers. On the other hand, there might be algorithms where not all the data fit a limited register set provided by a given CPU architecture. Such algorithms then typically store intermediate values into the program memory (usually to the stack) and pass data via memory. So even though instruction dependencies via memory are significantly less frequent than register dependencies, we still have to track them.

The problem is that when we try to track memory dependencies, it turns out that the vast majority of memory addresses are non-constant. The reason is that majority of expressions use stack pointer relative addressing. And as the register load of the stack pointer is non-constant, the address of memory accesses is also non-constant. There are also other reasons as the usage of dynamically allocated memory, the existence of PIC, and so on. For those non-constant addresses, we are not able to deduce whether they access the same memory or not.

One might argue that for two addresses both using stack pointer, we can deduce whether they overlap or not, which would be true. We will discuss very similar concept to this in section 7.2. But with our current set of tools, we are not able to deduce this kind of information. To sum it up, we are not able to deduce if two memory accesses with non-constant addresses overlap.

Because memory addressing with a constant address is so rare, it does not even make sense to try to find dependencies between memory accesses with constant addresses. Constant addresses are so rare that there is no constant address in the whole *grep* compiled by *gcc* for RISC-V which was used for testing. So even though we could try to match dependencies of memory addresses with constant addresses, it will not give us enough knowledge to make implementing such an analysis worth the effort.

The only smart dependency tracking optimization we can apply to memory accesses is a check whether they access the same memory address space. In subsection 2.1.2, we defined an infinite number of address spaces that are identified by strings. By definition, those memory address spaces are fully disjunct. For this reason, we can consider accesses to different address spaces not to depend one on another in any way. In other words, we will consider each memory access to the same address space to be dependent on all other accesses.

Algorithms to find data dependencies

Even though this fact might be obvious, it makes good sense to remind the difference between reads and writes in dependency analysis. Write modifies (or rewrites) the value, which makes it dependent on both writes and reads of the same value. On the other hand, reads do not consume, nor modify the value read. This implies that reads are not dependent on each other as they do not interact one with another in any way.

Every algorithm to identify data dependencies will iterate through a block of instructions. During this single walk, the algorithm will analyze both memory and register dependencies. To do so, it will always keep a map (key-value store) of registers and memory address spaces respectively. Those maps will always use the string key of register or memory address spaces as the key and the value will be the last instruction that wrote the respective register or memory address.

To find true (data flow) dependencies, we will iterate all instructions in the basic block in forward order. For each instruction, we will find all register loads and memory loads in all its expressions. We can do so by iterating all effects followed by iterating all their expressions. Then we will use the map of previous writes to find all instructions which wrote those values. As all those instructions write values the current instruction reads, we will consider the current instruction dependent on all those instructions. Once we have all the dependencies recorded, we have to update the map of instructions. To do so, we iterate through all effects of the instruction and write the current instruction to all register and memory keys it writes.

Before we describe the algorithm to find anti-dependencies, it might be useful to have a short recap on what anti-dependency is. An anti-dependency is a dependency between two instructions where the first reads a value which a latter instruction rewrites. As the former instructions needs to read the value that the latter instruction "destroys", those 2 instructions cannot be reordered. For example in the Listing 5.1, the instruction (2) (*sub*) consumes the value of $x5$ produced by instruction (1), which is then rewritten by instruction (3). Consequently, instruction (3) is anti-dependent on instruction (2).

```
1       add x5, x4, x3
2       sub x7, x5, 1250
3       add x5, x0, x0
```

Listing 5.1: Example of anti-dependency between instructions.

The description of anti-dependency implies that to find anti-dependencies, we will iterate basic blocks in reverse order. The detailed reasoning behind this is that we need to know which instruction rewrites the value the current instruction reads. Indeed, we might also iterate the list of instructions in forward order. But because unlike writes, reads do not interfere one with another, we might get into a situation where we would have to make multiple reads anti-dependent on the same (later) write. To represent such a fact, we would have to redefine the map of instructions to be a map of arrays of instructions. For this reason, it is simpler to iterate the basic block in the reverse order and remember writes. Thanks to the "destructive" nature of writes, we need to remember only the latest (read latest in the reverse order walk) write in our map.

A special case are instructions that both consume and write the same value (for example the same register). As those instructions require the value which they also destroy (in this order), our first intuition could be that those instructions should be anti-dependent on themselves. Naturally, it does not make sense to make an instruction dependent on itself. The meaning of dependency for us is that the two instructions cannot be reordered. But a statement that a single instruction cannot be reordered with itself makes no sense. It also does not make sense to make such an instruction anti-dependent on the following write as the instruction "destroys" the value itself. Instead, we will not add any anti-dependencies for those instructions and we will only understand them as writes, which implies that we will make it dependent on all previous reads of the value it writes.

We can implement an anti-dependency search by just a slight modification of the true dependency algorithm. The modification is that we will iterate instruction in reversed order and we will first apply effects on our map of writes before we iterate loads. The only additional property of the algorithm is the property of not making instruction dependent on itself. As we apply effects first, the map contains the instruction itself. If we then find a load that consumes the same register, we just skip this dependency. This condition can be implemented using a simple comparison of whether the instruction read from the map is the current instruction or not.

Even though the description of the algorithm to find anti-dependency might seem complex, its implementation in *mltwist* is quite straightforward. The whole algorithm is described in Listing 5.2.

```
for i := len(instrs) - 1; i >= 0; i-=1 {
    ins := instrs[i]
    for r := range ins.outRegs {
        regs[r] = ins
    }

    for r := range ins.inRegs {
        dep, ok := regs[r]
        if !ok || dep == ins {
            continue
        }

        addDep(ins, dep)
    }
}
```

Listing 5.2: Anti-dependency analysis for registers.

The last kind of data dependencies is output dependencies. Output dependency says that the final result of a basic block has to be preserved. To find this dependency we will again iterate instructions in reverse order. But this time, we will care only about register and memory stores.

For each register store, we check if we already have this register name in our map. If not, then the current instruction is the last one that writes this register. For this reason, we add this instruction to the map with a register key. On the other hand, if the key already contains some instruction, then the current instruction is not the last one writing the register. Consequently, we have to add an output dependency between this instruction and the instruction in the map (which is the last write of that register). Obviously, this algorithm never removes register keys from the map, it just appends new keys. This corresponds to the meaning of output dependency where the last write has to be preserved, but we do not care about the order of preceding writes.

The situation gets a bit more complex for memory stores. The problem with memory is that writes might overlap each other to create a unique value in memory. For example, let's imagine that we write four bytes of value *0x80* to bytes [8..11]. Then we can write two bytes of value *0x40* to bytes [10..11]. And then we can write two bytes of value *0xff* to bytes [9..10]. The final value of bytes [8..11] is then *0x80ffff40* and all three memory stores above contributed to it. This example can be generalized to an arbitrary number of memory stores.

As we are not able to analyze which addresses are modified by memory stores, we are unfortunately not able to perform any smart analysis which would say if two memory writes overlay each other. Consequently, we have to see all memory stores to be dependent on each other. In other words, memory stores cannot be reordered as their reordering might change the output value. Because of this fact, we make all memory writes to a single memory address space dependent on each other.

To lower the number of writes dependent on each other from $\binom{n}{2}$ (where n is the number of memory stores in a block) to just a linear number of dependencies, we can leverage our knowledge that relation of dependency is transitive. If we cannot reorder instruction a and b, and we cannot reorder instruction b and c, it is obvious that we also cannot reorder instructions a and c. This observation implies that we can use just a linear number of dependencies where every memory store will be dependent on the later write in the basic block.

To implement the memory output dependency algorithm, we will again iterate instructions in reverse order. We first iterate all effects and index the map for memory address spaces (their keys) written. We then make the current instruction output-dependent on all instructions which we find in the map. Then we update the map and write the current instruction into all keys this instruction writes. This way, the number of dependencies is just linear in the number of memory stores and the number of distinct memory address space keys.

Control dependency

As we discussed before, the only control dependency in a block exists if the last instruction is a jump instruction as in such a case it cannot be reordered with any other instruction in the block. To find this dependency, we simply check if the last instruction in a basic block is a jump instruction. If it is not, we are done as there are no control dependencies in the basic block. If the last instruction is a jump, we simply iterate through all instructions in the block and we make the last instruction dependent on all of them.

Special dependency tracking

In subsection 2.3.2, we specified a set of flags identifying instructions with properties that we are not able to represent in our intermediate code. We have defined three special instruction types: memory order, syscall, and CPU state change. In this section, we will discuss how to track dependencies when those instructions are present in a basic block.

For memory order instructions, we do not know the exact semantics of the instruction, so we must assume that both instructions before and after this instruction might be affected by the memory ordering effects. In other words, the memory ordering instruction has the meaning of memory synchronization in the code. By moving it (or reordering it with other memory accesses), we might affect the multithreaded behaviour of the program. The only thing we know is that memory ordering affects only instructions which access memory. We also know that two special instructions with memory order effects cannot be reordered as we again do not understand the exact semantics and we might accidentally change the program meaning.

To find those dependencies, we will iterate the basic block in both forward and reverse order. In each of those walks, we will perform exactly the same actions. The only reason for having two walks is that one will find memory-order dependencies in forward order and the other will find them in reverse order.

During each of those iterations, we will always remember the latest memory order instruction we have seen. If the current instruction is then memory access (load or store) or a memory ordering instruction, we have found a dependency between this instruction and the last seen memory order instruction. If the current instruction is memory ordering instruction, we then update the last memory ordering instruction seen. Then we will proceed with the next instruction in our direction of iteration.

This algorithm will find all dependencies between memory ordering instructions and other memory accesses. The forward walk will make all later instructions dependent on the previous memory order instruction while the reverse walk will make the later memory order dependent on all previous memory accesses. In both walks, we always store only the latest memory ordering instruction. This is possible as memory ordering instructions are dependent on each other. Transitivity of the dependency relation then guarantees that it will not be possible to reorder memory accesses with non-adjacent memory ordering instruction in the basic block.

Unfortunately, for syscall and CPU state change special instruction types, we know even less than for memory ordering instruction. In the case of syscall, the kernel could change whichever property of the program. This includes both memory and registers of the CPU. Moreover, in theory, the kernel could also change some CPU state which will result in a different interpretation of the machine code. In the case of CPU state change instruction, we by the nature of platform-specific CPU state change do not understand what the change did. And again, it might change the way the machine code is interpreted. To summarize, we do not understand those two special instruction types at all and for this reason, we do not have any better option than to prohibit any reordering of those instructions. In other words, we have to make those special instructions dependent on each other and we have to make all instructions in a basic block dependent on them.

To find those dependencies, we will again iterate instructions of a basic block in both forward and reverse order. We will also remember the last special instruction. The difference compared to memory order dependencies is that this time, we will make any instruction dependent on the special instruction unconditionally. After those two walks, we will have all instructions dependent on adjacent special instructions of ltype syscall or CPU state change and we wil also have adjacent special instructions of those types dependent on each other. The transitivity of the dependency relation again guarantees that no special instructions of those types will be reordered and that no other instructions can be reordered with any special instruction of those types.

5.2 Instruction reordering

So far, we have spoken about the dependency graph of instructions, but we have not spoken about a way to represent it. In this section, we will discuss how to represent basic blocks and instructions so that we can easily track dependencies between them. At the end of this section, we will then discuss how to implement instruction reordering.

5.2.1 Representation of dependency graph

Before we start to design a way to represent our dependency graph, we should summarize our requirements for the representation. The first requirement is that we have to be able to find out whether reordering of two instructions is possible or not in an efficient way. The second requirement is that we have to be able to implement reordering effectively on top of our representation. This means that the maintenance of the dependency graph has to be cheap or even for free. It is for example absolutely unacceptable to perform the whole dependency analysis again after every single reordering. The third requirement is that the representation should be efficient in terms of memory consumption.

Basic blocks

A good starting point in our design is the basic block analysis. We know that there is always some control dependency between individual basic blocks. This implies that instructions in different basic blocks can never be reordered. Another nice property of basic blocks is that every instruction in a program always belongs to exactly one basic block. This gives us a division of a program into blocks of instructions where we know that only instructions inside a single basic block can be reordered.

Due to all properties of basic blocks, we can represent our program as a list of basic blocks. Each block will contain a list of instructions. Instructions will be then stored in a list according to their order of execution. So before we apply any reordering operations, the order of instructions in the basic block will be the order in the binary parsed. Basic blocks themselves will be also stored in an array which will at the beginning follow the order of basic blocks in the program address space. The introduction of basic blocks results in a two-level representation of our program where the program comprises basic blocks and basic blocks comprise instructions.

Dependencies

So far, we have spoken about dependencies very vaguely in subsection 5.1.2. We know that dependency is a transitive relation, but we have not discussed all its properties yet. It is worth noting that we always defined the latter instruction to be dependent on the previous instruction. That is true because our relation of data dependencies forms a Direct Acyclic Graph (DAG) between instructions. This means that edges in our DAG (i.e dependencies) are oriented and they always follow the direction of instruction flow in the basic blocks[1].

The reason why the statement about DAG holds differs for individual dependency types. For true dependencies, it is very obvious that instruction can consume only data produced by previous instructions. In other words, the direction of the DAG edge follows the flow of the data in the program. Anti-dependencies describe that an instruction later in the code is ”destroying” a value which some preceding instruction needs to read. We can rephrase this as the fact that the value-destructive instruction can be executed only once the instruction consuming the value was executed. This again explains why the direction of dependency has to be always in the direction of execution flow for anti-dependencies. Output dependency says that the instruction writing the output value of a register or memory has to execute last. This already gives us the direction of dependency as the instruction writing the final output can be executed only when all previous instructions writing the same register (or memory) were executed. And last but not least, the only control dependency in the block says us that the jump has to be the last instruction executed, which makes the direction of the dependency obvious. To sum it up, all our dependencies (edges) always point in the direction of the program flow.

As DAG is a graph, we will represent dependencies as a graph. In our graph, instructions will be nodes and dependencies will be oriented edges. We will use a standard representation of an oriented graph where each node (instruction) will contain a set of the following nodes.

This representation would be sufficient to describe the whole graph, but it would not on itself fulfill the requirement of fast lookup for possible reordering. If we wanted to move instruction forward in the basic block, we could simply iterate through all dependent instructions and validate if the reordering is valid. But if we wanted to move the instruction backward in the basic block, we would have to iterate all previous instructions in the basic block and check all their dependencies. To make this process more efficient, we add another set comprising of nodes where oriented DAG edges directed to this node originated. To sum it up, each dependency will be represented in two places. In other words, for dependency $a \rightarrow b$, there will be a record in the forward dependency set in a and another record in the backward dependency set in b. This allows us to efficiently list all instructions dependent on instruction as well as all instructions the instruction is dependent on.

Instructions

We have to represent our instructions in a way that will allow us to effectively change the DAG. If we represented a static DAG, we would store instructions in an array and we would represent DAG edges (dependencies) using indices of nodes (instructions) in the array. Alternatively, we could use pointers into that array instead of indices. The problem with this representation is that when we move an instruction, we would have to update all the indices (or pointers) in all dependency sets in all instructions in the basic block. This is too much work to

[1]Technically we might also define the direction of our edges in the opposite order and all the logic would work the same way. It is just more natural to describe dependencies as forward edges rather than backward edges.

be done for a single modification.

We can speed up DAG transformations by ensuring that the instruction will always have the same identifier independently of the order of instructions in the basic block. We achieve that by allocating each instruction individually and storing pointers to instructions in the basic block. As the address of the dynamically allocated instruction never changes during the program lifetime[2], we will not have to update those pointers on instruction reordering. So we will use pointers to instructions in the list of instructions of a single basic block as well as to represent both forward and backward dependency sets in instructions.

By using pointers to dynamically allocated instructions to represent dependencies, we lost one important property compared to the simple array representation. We are no longer trivially able to deduce the index of instruction in a basic block. We might address this problem by adding a pointer to a basic block to every instruction. But it would still require a linear lookup to find the index of the instruction we have a pointer to. To make the instruction index lookup faster, we will store the index of instruction in a basic block in the instruction itself. This will cost us the same amount of memory (or potentially even less) than storing the basic block pointer and it will allow us to read the index in constant time.

As we will want to reorder basic blocks as well, it makes sense to use most of the tricks we described here also to basic blocks. So even our basic blocks will be dynamically allocated and we will store them just as pointers in the array. Moreover, we will also store an index of the basic block in our program.

It is worth noting that both basic blocks and instructions also store their respective memory address in the program virtual address space. For a basic block, this is the address of the first instruction in the basic block. In the case of an instruction, it is the address of the first byte of the instruction.

The overall representation of the program, basic blocks and individual instructions is shown in Figure 5.1. The schema visualizes a single basic block from a program that contains three instructions where the third instruction is dependent on both the first and the second instruction.

5.2.2 Reordering possibilities

In this section, we will discuss the implementation of instruction reordering. We will start with a discussion on which reordering operations make sense and which do not. We will also discuss which parameters we will modify by reordering and which of them we will preserve. And last but not least, we will discuss some limitations of our reordering which we will have to accept.

[2]There are indeed some programming languages where the virtual address of an object changes at the program runtime. This is usually the case for garbage collected languages with compacting garbage collectors. In those environments, the garbage collector tries to avoid heap fragmentation by moving dynamically allocated objects in the program memory. But as compacting garbage collector also updates all existing pointers to that object in the program, we are also certain that the value of the pointer is up-to-date and unique for every dynamically allocated object. For this reason, we can still consider the memory address to be constant. Anyway, the environment we program in behaves as if it was a unique constant.

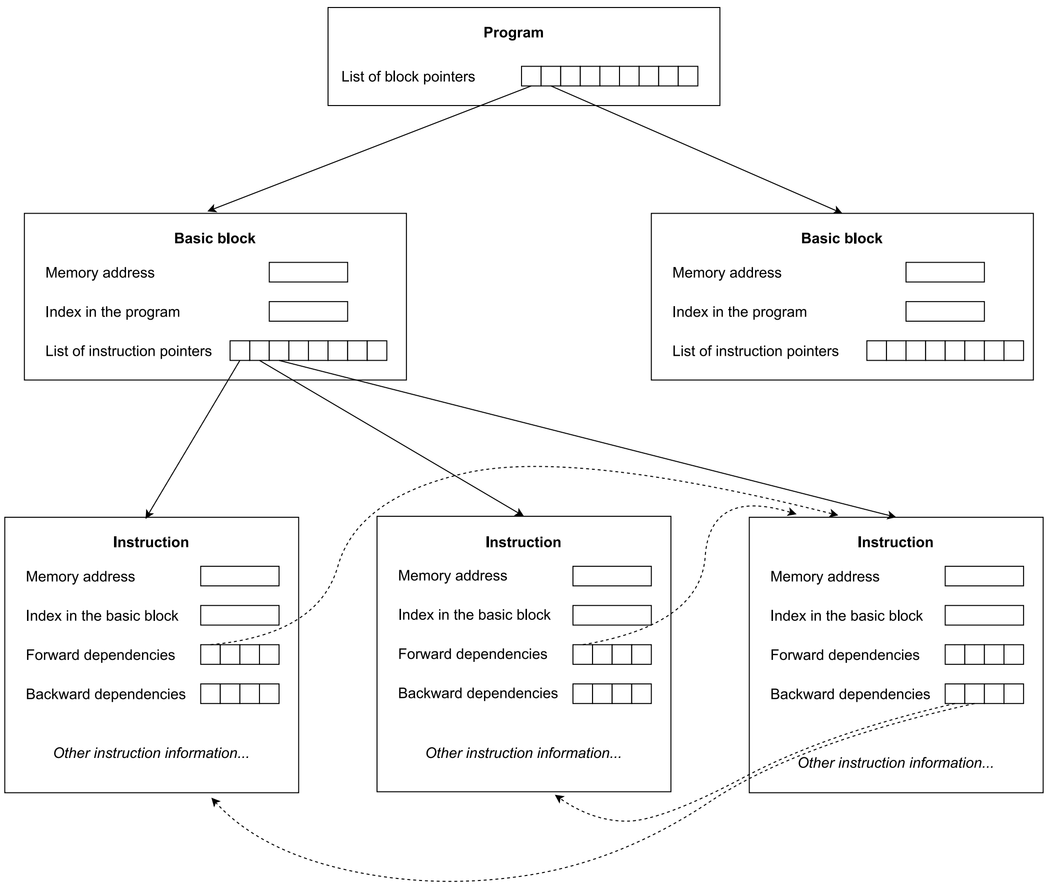

Figure 5.1: Schema of program dependency representation

Instruction reordering

Before we start to discuss instruction reordering, we should first clarify which properties of instructions will the reordering alter. Out of all the properties of instructions, it makes sense to alter only two of them - address and sequential number in the basic block. All other properties of the instruction will remain unchanged by reordering.

It might not be obvious why we will never need to modify the expression tree when we move instruction to a different address. It is true that in the machine code, there are instructions that use instruction pointer relative addressing. Instruction pointer relative addressing uses the current value of the instruction pointer as a base address it refers to. The reason why we are granted that expression will never contain an instruction pointer read is that in subsection 2.2.4 we prohibited the existence of such a read. Back then we argued that we do not need an instruction pointer read at all as we know the address of the instruction at the time of parsing, so we can use it as a constant rather than as register load. This property of the instruction pointer register now allows us to freely relocate instructions as we are certain that no expression depends on the instruction position. If there was such an instruction in the machine code, the parser replaced the address with a constant, and the move of instruction will not change this constant.

Changes of instruction encoding

One aspect of the instruction which we should change but which we are not able to change is instruction encoding (i.e. its bytes). Even though this makes our instruction reordering very incomplete as we cannot save and run the altered program, there are several good reasons for it. First of all, we would need a CPU architecture-dependent module to do such a change as the encoding of instructions differs between CPU architectures. This is on its own problematic and increases complexity significantly. But there are also many possible paradoxes induced by instruction re-encoding which make such re-encoding almost impossible.

The first paradox related to possible instruction re-encoding is that instruction pointer relative instructions might (and typically also do) contain intermediate values which are added to the instruction pointer. When we move instruction with instruction-pointer relative addressing, we have to adjust the immediate value appropriately to compensate for the change in instruction pointer value. The paradox comes from the fact that there is typically a limited number of bits representing an immediate value. Consequently, we might move the instruction in a way that the capacity of intermediate bits is no longer sufficient to capture the new immediate value. In such a case we would not be able to encode the new instruction as such encoding simply does not exist in a given CPU architecture.

One might argue that we might replace such instruction with a sequence of instructions implementing the desired action. This statement is true. Most CPU architectures provide some way to read, write and jump across the entire memory address space. There is usually some threshold (induced by the maximal size of immediate value) on how far (relatively to the current value of instruction pointer) can we get using a single instruction. For further reads, writes, and jumps, the CPU architecture typically guides to calculate the desired memory address using standard operations (constant loads, adds, etc.) and then perform load, store, or jump to an address provided via register. This property of CPU architectures guides us that we might always replace the instruction with a sequence of instructions with an identical meaning.

Unfortunately, there are a few problems when we try to split one instruction into many. First of all, we would need to have a free register to be used by our ”gadget”. The problem is that in a typical, well-optimized program, there are not very often spare registers. It is a much more frequent situation that all the registers contain a valid value. We also cannot store one of those registers into the memory as we do not know which memory is free and which is used for program data. This applies also to the program stack! Even though a typical convention says that no memory above the current value of stack pointer should be valid, it is proven by practice that there are programs using values above the current value of stack pointer. Consequently, we are not able to find a free register and we are even not able to free it using memory. This makes the implementation of such a gadget almost impossible.

But even if we were able to find a register or to find a sequence of instructions that would always guarantee us that no state of the program will be changed by expanding a single instruction to multiple instructions, there would still be other paradoxes. The paradox number two is that when we expand single instruction into multiple, we need more bytes to encode them. So we need to grow the basic block size. In other words, we would need to move all instructions in the basic block following the instruction we expand. But as we move them, we might again need to expand them with the very same gadget as their intermediate value capacity for instruction pointer relative addressing might be insufficient.

Moreover, a basic block is typically surrounded by other basic blocks and to grow a single basic block, we would need to move all the subsequent basic blocks. And this might again cause expansion of many instructions as they all have to be moved together with the basic block. Consequently, re-encoding of a single instruction might cause re-encoding of hundreds or even more instructions in the program.

The third paradox related to basic block shifting is that we would have to adjust all jumps to that basic block in the whole program. To do so, we would first have to be able to identify all such jumps and as we know from 5.1.1, we are not able to do so anyway. But even if we were able to identify all such jumps, we might again hit a limit on the maximal value of immediate operands. Consequently, we might again have to re-encode them to a sequence of instructions forming another gadget. This might again result in more instruction expansions and more shifting across the whole program.

Those three paradoxes are not the only paradoxes related to instruction re-encoding, but they are sufficient to explain why we call them paradoxes. The problem with paradoxes is that they might chain one with another recursively. For example, we might expand a single instruction due to paradox number one. But this might trigger paradox number two which would start shifting basic blocks and re-encoding instructions in them. But due to that, we would trigger paradox number three which would force us to re-encode many jumps in the program. But this can again trigger paradox number two by adding more gadgets. This would result in even more shifting and expansion due to paradox number two, which can again cycle to paradox number three and so on.

To sum it all up, it is a really bad idea to try to re-encode instructions on reordering. We have to consider the disassembler to be just a disassembler and accept the fact that we will not be able to store the reordered code as a new binary. This is simply not possible, because we are not able to deal with all the paradoxes described above in any reasonable way. Moreover, it would not even be a good user experience when a shift of a single instruction would mangle the whole program.

Basic block reordering

Similarly to instructions, we will also allow basic blocks to be reordered in between each other. But in the case of basic blocks, we have to be even more conservative than in the case of instructions. The difference will be that in the case of basic blocks, we will move blocks in their logical order in our disassembler, but we will not modify their addresses and addresses of instruction in them respectively.

The reason for that decision is to avoid one of the problems described in the previous section. If we moved a basic block, we would have to identify all jumps jumping to that basic block in the whole program and modify them appropriately. This would require an extensive analysis of the whole program. Moreover, we most likely would not be able to identify all such instructions as for some of them, the jump target is non-constant and thus unknown to the disassembler.

Another potential problem related to the basic block moving in the memory would be the modification of all constants related to the basic block address. A good example of when the compiler would add such a constant in the code is a call of a function with a callback (i.e. pointer to a function). In such a case the address of the function (basic block) is passed as one of the input values, so it is loaded into a register as a constant or from the program static data (memory). As memory addresses are unrecognizable from plain numeric constants in the code, we are not able to move basic blocks without changing the program meaning.

One might argue that we have already used expressions to identify dependencies and that relocation of basic blocks will not break our dependency graph. Consequently, it might seem that we do not need to preserve expression validity. Such a statement is true for dependency analysis. Unfortunately, we will require expressions to remain valid for our emulation described in chapter 6. For this reason, we cannot move basic blocks in the address space.

For all the reasons mentioned here, we will not change the memory addresses of basic blocks. This choice might feel very limiting, but in reality, it is not a big deal. As we already gave up on storing the reordered program in section 5.2.2, we have no real reason to change the basic block address.

Yet it might be convenient for the user of our disassembler to visually change the order of basic blocks in the program. To provide this functionality to the user, we will allow to modify the basic block index and its position in the array of basic blocks. Using this simple trick, we will be able to provide an illusion of reordering basic blocks and we will avoid all the problems described above.

Possible violation of control dependencies

It is worth reminding that even though we do our best to guarantee that the reordered program will have the same meaning as the original one, it is still possible that the meaning of the reordered program will differ. The reason for this is that our basic block analysis is not perfect. As we discussed in section 5.1.1, there are some instruction pointer stores which are non-constant. Because we are not able to include those jumps into the basic block analysis, it is possible that we missed some control dependencies along the way. Specifically, the unknown jump instruction might jump into the middle of a basic block. In such a case, the basic block should be split into two by the analysis of bas¡ic blocks, but as we were not able to deduce the jump address it was not. Consequently, there might be basic blocks in our disassembler which are not in reality basic blocks (i.e. they comprise multiple real basic blocks).

As we use basic blocks as a boundary for instruction reordering, our disassembler could allow the reordering of instructions in a much greater range than it should if it saw all control dependencies. Naturally, such an instruction reordering might change the program meaning. Unfortunately, there is nothing we can do about this as we are not able to analyze such jumps. We will discuss an idea of how to partially address this problem in section 7.3, but even then, we will not be able to analyze all jumps in the program.

One might argue that we are inconsistent in our requirements because we allow to change the program meaning in case of missed control dependencies, but we do not allow it in many other cases. It is true in many places in this chapter we refused to implement some reordering or some code change algorithm with the reasoning that it could change the program meaning. But there are a few good reasons why we allow this kind of code-meaning change and we disallow all other code changes.

The very first reason is that if we would not accept this potential change of the program meaning, we would not be able to implement any instruction reordering at all. Even if we would be able to implement the idea described in section 7.3, we still would not be able to guarantee that we will not break any control dependency. We have presented an example of a program that cannot be analyzed at all in section 5.1.1. This implies that if we want to be able to reorder machine code, we will always have to accept some risk of breaking control dependencies and there is nothing we can do about it. We might implement better algorithms to find more control dependencies and to lower the risk of missing a control dependency. But we will never be able to shrink the risk to zero due to the existence of the programs we referred to above.

Another reason why we can claim this risk to be acceptable is that compilers typically create quite well-structured code in which we should be able to find the vast majority of basic blocks. We could also rephrase the previous sentence in a way that compilers do not usually use as much freedom in jumps as the CPU provides. The reason for this is that internally, compilers also operate on top of basic blocks. Compilers also optimize to make the execution of the program as fast as possible. So if they can use instruction pointer relative jump with immediate value, they will do so rather than load the value to the register first and jump to a register value. Moreover, compilers have no intention to make the code more complex and harder to analyze. Those facts together imply that the vast majority of basic blocks should be identifiable by our basic block analysis.

An important takeaway from this section is that when the user starts reordering individual instructions, it will be in the end up to him or her to guarantee that the program meaning will be preserved. The disassembler will do its best to assist with this task. But as it will never be able to fully understand all the control dependencies, it will never be able to fully avoid changes in the code meaning.

5.2.3 Reordering implementation

We have so far discussed all the reordering we will do together with all limitations our instruction reordering might have. So the only remaining part is to implement such reordering. In this section, we will discuss how will the interface look like, how will we check for dependencies, and last but not least, how to perform the modification of instruction order.

Instructions

Before we start to discuss the interface, we should review what we mean to by instruction reordering. By instruction reordering, we refer to moving instruction to another place in the basic block. As we represent a basic block as an array of instructions, we can formalize *move* of an instruction at position i to position j where i and j are indices in the array. The *move* operation relocates instruction at index i to index j and shifts all instruction in the basic block either one instruction forward or backward depending whether $i > j$ or $i < j$ respectively. Naturally *move* from i to i is no-op.

An interface of our library will expose *move* operation with two integer arguments i and j. This function will be exposed for every basic block in our program. This implies that the interface will not even allow to specify *move* between basic blocks as each basic block is handled independently. This fact simplifies control dependency checks and the *move* algorithm itself, but it also avoids invalid usage by the user.

To check whether *move* is allowed by our dependency graph (DAG), we will iterate over all instructions in range $[i..j]$ or $[j..i]$ respectively and check whether there is no dependency blocking the *move*. As our move could work in both the forward and the backward direction, we will explain those 2 parts independently to avoid confusions.

For $i < j$, the *move* relocates instruction from i to j and it shifts all instructions in range $(i..j]$ one instruction backward. The shift cannot violate any instruction dependencies because if instructions $(i..j]$ do not violate any dependencies before the *move* call, shifting them one instruction back will not violate any dependencies either. So the only dependencies which can be violated are dependencies between instruction i and instructions in the range $(i..j]$. To check those, we will iterate all forward dependencies of instruction at index i. As our dependencies point to instruction objects and instruction objects know their index, we can implement such a check by simply checking that any dependent instruction does not have the index in the range $(i..j]$. If we find such a forward dependency with index k where $i < k \leq j$, the *move* is not possible as it violates dependency in between i and k. On the other hand, if we do not find such an index, we can do the *move*.

During the forward *move*, we move all pointers in the range $(i..j]$ one instruction back and we then write the pointer which used to be at position i to position j. Then we have to update all indices stored in instructions in the range $[i..j]$ to correspond to their new index in the array. And last but not least, we have to update memory addresses. As instructions, in general, do not have to have the same byte length, we always set the address of instruction to the address of the previous instruction plus the byte length of the previous instruction. And again, we do this update only in the range $[i..j]$ because the length of the whole range could not be changed by *move*. Once we do this we are done and *move* from i to j, where $i < j$ is successfully completed.

For $i > j$, the *move* implementation will be similar, just all terms will be swapped. We will check backward dependencies of instruction at index i and we will look for $j \leq k < i$. If the *move* does not violate any dependency, we will move instructions in the range $[j..i)$ one instruction forward. Both index and address updates will then be identical to the $i < j$ case.

One functionality we might also appreciate is finding the maximal and minimal value of j where we can *move* instruction i to. To perform this lookup, we add two more functions: *lower_bound* and *upper_bound*. Those instructions will be again defined on each basic block object and will operate on that single basic block. Both those functions will accept a single parameter – integer index i.

In function *lower_bound*, we will look for minimal value of j where we can *move* instruction i. To do so, we will iterate all backward dependencies of instruction i

and we will look for the maximal value of an instruction index. Once we iterate all dependencies, we simply return the maximum found plus one. The reason for plus one is that the maximum is the maximal index of instruction dependent on i. So the maximal value of j is $max + 1$ as *move* to max would already violate some dependency.

The implementation of *upper_bound* will be similar. We will iterate over all forward dependencies and we will look for the minimum of all indices. Once we find it, we will subtract one to get the maximal value of j such that $move(i, j)$ will be successful.

It is worth noting that we can implement the dependency check in *move* function using a combination of *lower_bound* and *upper_bound* functions. For $move(j, j)$, we can perform a check on $lower\_bound(i) \leq j \leq upper\_bound(i)$ to check whether such *move* will not violate any dependency. As implementing the *move* check this way lowers the overall complexity of the implementation, it is used in *mltwist*.

Blocks

Moving of blocks is significantly simpler as we do not track any dependencies in between basic blocks. To implement the moving of basic blocks, we will again define function *move* with 2 integer parameters i and j on our program object (i.e. the object containing the list of basic blocks). The difference is that this time, the move cannot fail besides index out-of-range errors.

To implement *move* from i to j, we will again have two variants for $i < j$ and $i > j$. As we do not have any dependencies this time, we do not check for them. We just move blocks in the array, update indices and we are done. The details have been already covered in subsection 5.2.3. One difference compared to instruction *move* is that this time, we do not update the address of the basic block for reasons explained in section 5.2.2.

5.3 Summary

- We start dependency analysis with an analysis of basic blocks. At the beginning, we consider the whole code to be a single basic block. We then further, split basic blocks when we find a reason to do so:
  1. Instructions do not follow one another – code segment boundary
  2. Instruction is a jump instruction – end of a basic block
  3. Instruction is the target of a jump instructions – begin of a basic block
- Data dependencies are found in each basic block separately. To find them, we iterate over a list of basic blocks and in each of them we iterate a list of instructions. We have three data dependency kinds:
  1. True dependency – one instruction consumes the output of the previous instruction
  2. Anti-dependency – instruction rewrites input of previous instruction
  3. Output dependency – output of the basic block must not be changed

- The program is represented as a list of basic blocks. The basic block is then represented by an array of instructions. Because instructions in different basic blocks are control-dependent on basic block boundaries, instruction reordering is possible only within a single basic block.
- The disassembler does not re-encode instructions when they are moved. It is even impossible to do so due to various problems with instruction encoding such functionality might cause.
- The basic block analysis is imperfect and we might miss some basic block boundaries. This is an unavoidable fact because jumps in the program might jump to addresses only known at runtime. The user has to be careful not to change the meaning of the program by reordering instructions.

6. Emulation

In this chapter, we will describe the possibility of program execution emulation. To do so, we will leverage our intermediate code as well as its transformations. On top of that, will design expression storage which will be one of the core components of the emulation. And last but not least, we will use all those building blocks to construct the emulation algorithm itself.

6.1 Design of emulation

The idea behind emulation is that we can deduce which values the program reads and which it writes from an intermediate code representation. Reads are represented by the register and memory load expressions in expression trees. Analogously, writes are encoded as either register or memory store effects. We also know which addresses the program jumps to as those are writes to the instruction pointer registers. Knowing all this information, we can (almost) fully emulate the execution of the program.

The shift from dependency analysis to emulation allows us to change our view on the problem. Until now, we have been operating with generic expressions. But in this chapter, we will be able to limit ourselves to constants. The reason is that as we emulate the program execution, we can gradually build knowledge about the values of program registers as well as contents of the memory. Consequently, we should know the exact values (i.e. constants) in the program. A different explanation might be based on the fact that every value the real program operates on is known to the CPU at run-time. Which makes it clear that when we emulate the program execution, we can also limit ourselves to constants.

Before we continue, we should discuss which data any program operates on. The first possibility is that the program reads the data from its static data section (or loads it as constants in the code). Those values are easy to emulate as we can simply read them either as constants or we can read them from the static data. As the whole binary (including constant and static data) is an input of the disassembler, the disassembler has the static data as well even though we have not discussed their processing yet.

But if all possible program inputs were just static data of the binary, the compiler could simply pre-compute the whole output of the program and optimize the whole program code to just write the result. This implies that any program which performs any useful calculations has to also accept some run-time inputs. Those can be data provided by the user, read from a file on a drive, received as a sequence of UDP packets or they might come from any other run-time source of data. It is not important how those data are obtained.

The consequence is that we need to design a way to provide input (i.e. additional, run-time data) to our emulation. In real programs, the input is exclusively provided by interactions with the operating system. Those might be either synchronous operations for example an execution of a syscall instruction, or asynchronous operations when the operating system execution is invoked for example by an interrupt.

As an emulation of an operating system is out of the scope of this thesis, the only thing we can do is to make the user of our emulation framework responsible for the emulation of operating system actions. This means that our emulation will expose an API to allow the user to emulate an operating system. In the simplest possible case, the API will be used by the user to emulate operating system actions by hand. On the other hand, nothing prevents the user from building an emulation of the operating system using the API to affect program execution.

To gather requirements on our API interface, we have to answer two core questions: when can be operating system action invoked, and which properties of the running program can the operating system change. The answer to the first question is that there are two ways of invoking an operating system – synchronous and asynchronous. The synchronous invocation happens when the program executes some form of syscall instruction. In such a case the program explicitly requests the operating to perform some action. The other cause of operating system action is asynchronous. Those are various interrupts and exceptions. Interrupts, exceptions, and other OS invocations are typically very specific for a given CPU architecture which implies that we are not able to emulate them reasonably. But a shared property of all CPU architectures is that interrupt can happen at any instruction boundary[1]. On the other hand, when an instruction is executed, it is always executed as a whole. In other words, the operating system is never invoked in the middle of instruction execution. The existence of asynchronous OS invocation implies in the most generic case the operating system can execute at any instruction boundary. Consequently, we will have to design our emulation API to allow program state change at every instruction boundary.

One might argue that the asynchronous execution of the operating system should not change the program environment in any way. Indeed, the majority of interrupts do not result in any change in the program state. However, there still exist a few cases when the operating system changes the state of the program during an asynchronous interrupt. This is for example the case with Linux time which is exported to the user-space using shared memory [20]. Specifically, there is a shared memory page between the kernel and the program (the page is read-only for the program) which the kernel periodically updates. As such a change of shared memory can affect program execution, our API has to allow to emulate such an effect of asynchronous OS invocation on program state. Another example of asynchronous interaction of the kernel and the program is Linux *io_uring* API [21], which allows asynchronous execution of program syscalls using shared memory as well. We do not say that we will ever try to emulate any of those kernel functionalities, but the interface we expose should be generic enough to allow the implementation of those special cases if they ever become needed.

The other question was which values can be changed by the operating system action. The answer is that an operating system can change any memory value and any register of the program. Indeed, the operating system can also change other states of the CPU for example MSR in x86 and AMD64 architecture. But

[1]There are exceptions to this rule. For example the x86 and AMD64 architectures both define that any write to *%ss* register is executed atomically with the next instruction. In other words, no interrupt can be delivered between write to *%ss* and the following instruction. It is almost certain that there will be different exceptions in other CPU architectures. As those are rare exceptions to the general rule, we will not address this as a concept of emulation. Instead, we will say that the OS emulation is responsible for handling those.

in our intermediate code, those can be as well represented as register or memory change. So in summary, our interface has to allow to modify both emulation memory and registers of the program emulated.

To support those cases, our emulation framework will expose a function *step* which will emulate the execution of a single instruction currently referenced by the instruction pointer. The *step* call will evaluate all effects of a single instruction and it will modify both registers and memory appropriately. Besides *step*, the emulation interface will also expose full program memory and all registers of the program which the caller of the emulation will be allowed to modify between individual *step* calls.

It is worth noting that the API we described is generic enough to allow to build a full operating system emulation. The emulation would call *step* in a loop to emulate the program execution. If the instruction pointer points to a syscall instruction, the emulation can read the program state, emulate the system call operation and write results back to program registers and memory. Besides that, if it is time to perform any asynchronous action by the kernel, the emulation can also do so using the program register and memory storage. This way, a full kernel emulation could be implemented. But for now, we will leave it all up to the user in *mltwist* and we will focus solely on the emulation of the machine code itself.

Another aspect of the emulation we need to discuss is that we will not always force the emulation to start at the program beginning. Emulation of the program from its very beginning would not be very useful as to emulate the program from the start, the user could simply run the real program with a debugger attached. Instead, we will allow the emulation to start at an arbitrary instruction in the program.

The possibility to start emulation at whichever instruction brings one additional challenge – we might not always have all the values of registers and memory as those might be written by instructions which execution was omitted. In other words, the emulation might start after the register or memory write instruction. In such a case, we need to provide the emulation of all missing values to be able to proceed.

It is true that we might use the same approach that we want the operating system emulation to use. We might say that whoever calls the *step* function has to guarantee that any value read by the instruction executed has to be in program registers and memory. The problem with such an approach is that we would force the caller to implement a significant portion of the emulation algorithm. The reason is that emulation of loads would force us to evaluate (read constant fold) the addressing expression first. Given that addressing expressions can chain, the caller would be forced to evaluate most of the expressions to make the emulation with such an interface work. Obviously, we want to provide the user with a way to provide those values more reasonably.

To provide values unknown to the emulation, we will add an interface with two additional functions which the caller will provide to the emulation. We will call this interface a data-source and its functions will provide either register or memory values respectively. The register function of the data-source will accept a register key string and *width*. The return value will be then a constant of the specified *width*. For the memory function, we will use the same string key for memory address space and *width* of the operation, but this time we also add an address parameter. This function will then also return a constant of specified *width*. Whenever the emulation needs to use an unknown register or memory value, it will ask one of those functions to provide the value. The whole data-source interface will look like this:

```
type DataSource interface {
    Register(key string, w uint8) Const
    Memory(key string, addr Addr, w uint8) Const
}
```

In *mltwist*, such a call will ask the user to provide the input using a console prompt. But the data-source interface is generic enough to allow any smart logic to find the value to return. For example, if the value is supposed to come from the network, the emulator can simply wait for the packet to arrive and then provide it to the application.

The concept of data-source will play its role even when the emulation starts at the program entrypoint. The reason is that each operating system defines some initial state of the program. For example, the operating system preallocates some stack for the program and it initialized the stack pointer register. Another value filled by the operating system is the number of command-line parameters and an array containing their values. Those initial values might indeed be set by the operating system emulation before the first call of the *step* call. On the other hand, as we do not want to emulate the operating system in *mltwist*, the data-source interface is used there to provide those values for the emulation.

To sum up the interface of our emulator, there will be a *step* function that will emulate the execution of a single instruction (the one currently pointed by the instruction pointer). Between steps, the caller of our emulation will be able to modify both registers and memory to emulate both synchronous and asynchronous operation of an operating system. A syscall instruction can be emulated using this mechanism by checking if the current instruction emulated is a syscall and acting appropriately. Moreover, if some value is not known to the emulation during the *step* execution, it will ask the data-source interface to provide it. The data-source interface is then allowed to emulate any other part of the system or prompt the user to provide the value.

The described interface of the emulation looks very similar to this in *mltwist*:

```
type Emulation interface{
    Step() error
    Registers() RegisterStorage
    Memories() MemoryStorage
}

func NewEmulation(s DataSource) Emulation
```

6.2 Storage

The key component of our emulation will be the representation of memory and register storage which will store the state of the program between individual *step* calls. In this section, we will discuss details of such storage. Given that there are two separate storage types in our intermediate code (registers and memory), it is not surprising that we will build 2 storage types. We will start our description with register storage and we will later proceed to memory storage. At the end of this section, we will also present some special storage types.

Even though our whole emulation operates on constants, we will require our storage to operate on generic expressions, rather than exclusively on constants. This requirement might be confusing as we will increase the complexity of the implementation for no obvious reason. The reasoning behind such a decision is that in some future extension of the disassembler we might potentially reuse the same storage implementation in other algorithms where we would need generic expressions to be stored. We will discuss one such extension in section 7.3.

It is worth noting that having generic storage is not limiting to the emulation at all. This is implied by the fact that an expression with only constant inputs can be constant-folded into a constant. So if all the values we will write to the storage will be constants (which is the case for emulation), we are guaranteed that constant folding of the value read from the storage will result in a constant expression as well. This property will allow our emulation to operate exclusively on constants even though the storage stores generic expressions.

6.2.1 Register storage

Register storage is very simple to implement. In our intermediate code, we identify registers by simple string constants. This implies that register equality can be checked using a single string comparison. Moreover, any register write always rewrites the whole register value and each read reads the whole register.

We can represent register storage using a simple key-value map. It does not matter whether we use hashmap or we decide to use any kind of a tree. The only important fact is that we will use a key-value store where our key will be the register key and the value will be an expression representing the value stored in the register. Such a representation fully corresponds to the definition of intermediate code registers.

The only aspect of register operations we have to take into account are different *widths* of register stores and following loads. To emulate the *width* behaviour, we will add *width* argument to both *load* and *store* functions. Both those functions will check if the *width* of expression loaded or stored corresponds to the *width* provided and if it does not, they will adjust it appropriately. Internally the storage will use the same *width* adjusting algorithm as we described in subsection 4.3.1 to adjust *width* of both stored and loaded expressions.

To sum the design up, the interface of the register storage will have two methods: *load* and *store*. Both those methods will accept the register key and a *width* of the register operation. The *store* the function will naturally also accept an expression to be stored. Analogously, the *load* function will return the expression read. Besides the expression read, the *load* function also has to return some form of indication of whether the register key is present in the register storage.

```
type RegisterStorage interface {
    Load(kes string, w Width) (Expr, bool)
    Store(key string, e Expr, w Width)
    }
```

6.2.2 Memory storage

As always, the implementation of memory emulation is significantly more challenging than register storage implementation. The first difference compared to register storage is that memory is not addressed by strings but by expressions. So we will have to find some way to represent addressing using generic expressions. Another complexity added to memory storage is that, unlike register accesses, memory accesses can partially overlap each other. Consequently, a single write can overwrite multiple previous writes or it can just partially overwrite some writes. Analogous problems apply to reads. In this section, we will discuss those possible problems and design choices to address those issues.

The first aspect of our memory storage interface we need to discuss is what will we use for addressing. In our intermediate code, we use an expression as a memory address to make the intermediate code as generic as possible. The problem with expressions as addresses we discussed in section 5.1.2 is that we are not always able to deduce the exact address. Moreover, we are not even able to conclude if two expressions overlap each other when we use them as the address for memory reads and writes. Obviously, those are properties of memory addresses we would need to implement any sort of memory storage. Which is the key reason why we will not use expressions to represent addresses in our memory storage.

Instead of expressions, we will use an unsigned integer of type *addr* for memory addressing. We have defined this type in subsection 3.5.1 and we have also discussed its advantages over the usage of both generic and constant expressions. The *addr* type is sufficient to represent memory addresses in the whole disassembler, and there is no reason why preventing its usa to represent memory addresses in our storage as well.

The key difference between register and memory storage is that register stores always write a single register and loads always read a single register. In memory, the unit written or read is not a single value, but a single byte. Because typical memory load and store operations access multiple bytes at once, we can get into a situation when only half of the previously written value is loaded or rewritten by another store. This fact will make the memory storage design significantly more complex than the design of register storage.

The very first idea of memory representation could be to use an array of bytes. Stores would then write a specific range of bytes and loads would on the other hand read the range of bytes. Such a solution would work perfectly if we were satisfied with storing constant expressions exclusively as then, we could deduce the value of a single byte. But as we want to be able to store generic expressions, we might not be able to deduce the byte value of the expression stored and so we would not be able to update the byte array appropriately.

Another idea might be to store single-byte expressions rather than bytes. It is true that whenever we have an expression with a given *width* (specified by the memory store intermediate code instruction), we can produce expressions representing individual bytes. We can do so using bit shifts and *width* cuts to a *width* of a single byte. So for an n byte write, we could produce n expressions each representing a single byte written. Our storage would not then be an array of bytes, but an array of expressions each representing the value of a single byte in the memory.

The expression array representation of memory has a few downsides. First of all, it is both memory and computation inefficient to store n bytes as n expressions. For example, if we decide to store 8 bytes and then read the same 8 bytes from the memory storage, the expression read from the storage would be composed out of 8 single-byte expressions each evaluating the same subtree. If we did such a store&load sequence k times in a row, the complexity of the expression evaluation would grow exponentially to 8^k times the original complexity. The second problem with the array of expressions representation is that the array itself would have to be quite large. To represent a memory address space of the program, we would need an array of expressions of the same size. Given that (virtual) address space of contemporary programs can reach up to 2^{64} bytes, we would need the same amount of physical memory to represent such an array. Moreover, our expressions (or pointers to them) require more space to be stored than a single byte. Consequently, the disassembler might need more physical memory than any modern computer on the earth has these days.

To remedy the problems of the array of expressions and to satisfy the requirement of storing generic expressions, we can represent the storage as a set of ranges rather than an array of bytes. A single range represents a single expression being written to the storage. The range is encoded as quadruplet: *addr*, *begin*, *end* and *exp* where *exp* represents the expression being written by a single memory store and *addr* is the address of the memory store. Values of *begin* and *end* then say which bytes of *exp* are stored in the range, where *begin* is inclusive and *end* is exclusive. For example at the time the range is written, the value of *begin* is always zero and the value of *end* corresponds to *width* of the memory store effect. Both the value of *begin* and *end* can be modified by later stores which overlap with a given range.

We represent our memory as a set of non-overlapping ranges. The fact that ranges are non-overlapping is important because we have to be able to determine the actual state of the memory. To keep ranges non-overlapping, we will check for conflicts on each *store* of a new range. If there are ranges colliding with the new range being written, the logic will modify either *end* or *begin* and *addr* appropriately. Such a change in the range will have two effects. First, it will change the memory byte sequence occupied by the range. And second, by modifying *begin* or *end* value, it will change the bytes taken from the expression. The modification of existing ranges performed by a new write is visualized in Figure 6.1.

Note that any change of range during its lifetime can only shrink it, but never expand it. In other words both *addr* and *begin* can always only increase and *end* can always only decrease. This observation implies that it is possible to shrink the whole range to zero bytes. Such an event happens once a range is completely rewritten (i.e. the write from the original program was replaced by bytes written by a sequence of later writes). A range is rewritten and though useless once its *width* is zero or negative. Such an event happens when $begin \geq end$. Ranges fulfilling this condition can be dropped from the storage, because it no longer represents any value stored in the memory.

Whenever we read a single range, we have to read only those bytes of *exp*

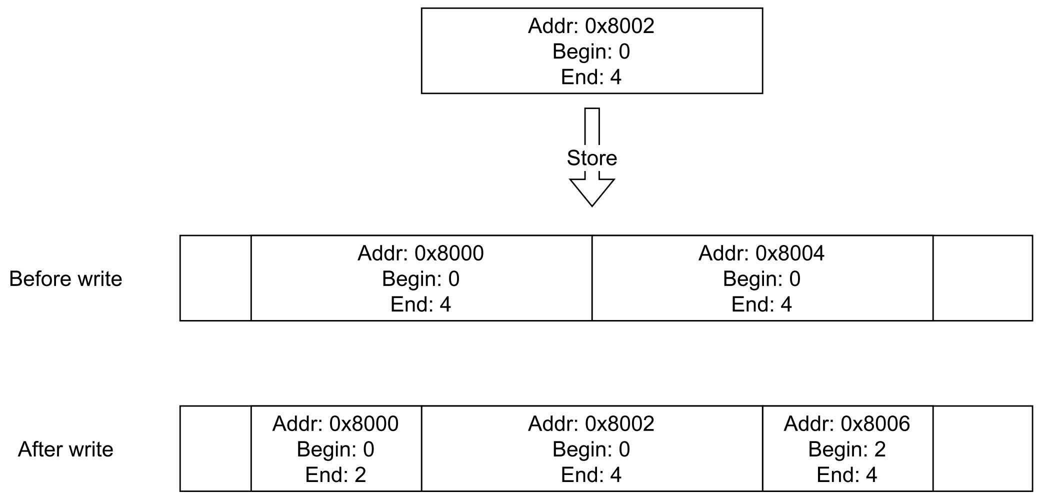

Figure 6.1: Visualization of existing range modification on new range store

which we are supposed to read from the range. Those bytes are given by values of *begin* and *end* where *begin* represents the first byte to read from *exp* and *end* represents the first byte not to read (remember *end* is exclusive). We will start with the *begin* parameter and we will right shift *exp* by *begin* bytes (i.e. $begin * 8$ bits). This is possible as our values are encoded in little-endian which implies that the first bytes in memory represent the lowest bits of the value. Once we have our value shifter appropriately, we can change its *width* to the *width* of the range, which we calculate as $end - begin$. We can do so using the width gadget or we can use a smarter approach for example the one described in subsection 4.3.1. The output of this algorithm is the value of the expression *exp* in byte range $[begin, end)$.

A single *load* operation from the range storage can in general overlap multiple ranges. This case corresponds to a situation when the program reads bytes written by multiple previous writes. To handle such a *load* we have to define the way to combine multiple ranges into a single one. We will start with extracting expressions from each range individually, but we will add a later stage where we will mix them. For each range, we will left-shift produced expressions by the difference between the range address and the load address. For example if a range starts at offset *0x255* and the load starts at address *0x250*, we will left-shift the expression 5 bytes (40 bits) right. Once we have all the expressions shifted to their correct position, we can just OR all the expressions together to produce the final expression.

An important detail of the load algorithm we ignored for a while is that the loaded range does not have to start and stop at the range boundary. In such cases, we can treat the first and last ranges as if they started at the beginning or the end of the loaded range. We can do so by adjusting their *begin* and *end* appropriately.

One might argue that such a form of *load* implementation can produce expressions that are quite complex as they are composed of multiple expressions. Such a statement is true. Unfortunately, it turns out we cannot do anything better than this. If a single *load* reads values written by 3 stores, in general expressions we are not able to do anything better than to use all those 3 expressions to compose the final value.

The way we implement *load* and especially the way we extract expressions from ranges also suggests one possible simplification of *store* function. Unlike the register store, the memory store does not have to change the *width* of an expression written. This is true as the *width* of the *store* is encoded in values of *begin* and *end* and another *width* adjustment is implemented when the value is loaded. This is not an important detail. It would be functionally equivalent to use a width-gadget twice. On the other hand, we can save some memory during the emulation by not doing so.

Another aspect that is more complex than in the case of register emulation is that the value read might not be present in the memory. In register storage, we addressed this problem by returning a boolean flag indicating whether a given register value is present or not. This has been sufficient for register storage as the register value can be either present or not. But with memory, we can get into a situation when only some bytes are missing for successful execution of the *load*. Consequently, a boolean flag indicating successful or failed *load* is no longer sufficient for the emulation to find out which memory ranges are missing.

To allow the caller to find out which bytes are missing for the *load* to succeed, we have to add another function to the memory storage interface. We will call this function *missing* and it will accept exactly the same set of parameters as *load* function. The *missing* function will then iterate the set of ranges in the same way as *load* does, but its purpose is to find out which ranges are missing.

To sum up the interface of the memory storage, it will expose 3 methods: *store*, *load*, and *missing*. The *store* function will accept the key of memory address space, the memory address to write to, the expression to write, and *width* of the write. The *load* function will take 3 parameters: the key of memory address space, the address to load from, and *width* of the load. The return value of *load* function will then be an expression read and a boolean indicator of whether the load was successful. The *missing* method will accept the same set of parameters as *load*, but it will return an array of intervals missing to make the *load* with the same parameters succeed.

```
type Memory interface {
    Load(addr Addr, w Width) (Expr, bool)
    Store(addr Addr, ex Expr, w Width)
    Missing(addr Addr, w Width) IntervalMap[Addr]
}
```

So far, we have not discussed the storage for our ranges. We have quite extensive requirements for our storage. Ideally, we need a data structure which can store ranges and effectively lookup ranges overlapping a given interval (interval of *load*, *store* or *missing* calls). Luckily there exists a data structure with exactly such functionality called Interval Tree [22, 23]. As implementation of an interval tree is out of the scope of this thesis, we can use some existing implementations to store our ranges.

To represent multiple memory address spaces, we will use the same approach we used for registers – key-value storage. Just this time, the value stored for a specific key will not be an expression, but an interval tree full of ranges written to that particular address space. The logic of *load*, *store* and *missing* will first use the memory address space key to find the right tree and then, they will operate only on that single interval tree.

```
type MemoryStorage interface {
    Load(key string, addr Addr, w Width) (Expr, bool)
    Store(key string, addr Addr, ex Expr, w Width)
    Missing(key string, addr Addr, w Width) AddrSet
}
```

6.2.3 Special memory storage types

One problem the emulation algorithm will face is when the program will try to read its static data or access its shared variables. The problem is that even though those data are part of the program binary, they are not executable code. Consequently, if the emulated program tries to access them, the emulation will find out that there is no such value available in the memory storage. In the current setup, the emulation would react to such a situation by asking the data-source interface to provide the value. In this section, we will discuss better way to handle such a situation.

Naturally, we might implement the data-source interface to provide the value from the binary data if such a situation happens. Even though such an approach would technically work, it has several disadvantages. First of all, it takes significantly more memory to store a value as an expression than to store it as plain bytes. This is not an issue for values stored during emulation as there the genericity of the storage is justifiable by properties such as memory sparsity – a fact that the majority of memory addresses will never be written to and therefore it does not make sense to represent memory as an array. But those reasons do not apply to storing a byte array from a binary. The second disadvantage is that to store constants from a binary into the memory storage, we would have to effectively copy all those data into constant expressions first. So we would have the original binary read in the program memory and besides that, we would also have there the same data once again in constants.

Given the disadvantages of using the data-source interface and our memory implementation in general, we might start to consider designing different implementations of memory storage interface. We can make those memory types interchangeable with the original memory implementation by exposing the same interface. This will allow us to define other types which will make it easier for us to give the program its static data and constants.

Byte memory storage

The first memory storage type we will add will allow us to effectively represent bytes of the program binary as memory storage. We will call this kind of storage byte memory storage. A byte memory storage is a memory storage implementation using bytes to store values.

It might be surprising that we now want to build a byte memory storage even though we decided not to use it before. We have designed our memory storage interface to store a generic expression and as we argued with memory storage implementation, a byte array is not sufficient to represent a generic expression.

All those points are valid. But it is important to point out that we will never use byte storage to store expressions. We will use it exclusively for reads of read-only byte arrays. And for this purpose, byte memory storage is the right solution.

One additional aspect of our byte memory storage is that it will not use a single byte array to represent the memory. The reasoning behind this is that we want to be able to effectively represent bytes at any offset in the memory. For example, if we want to represent 1 MB of memory at a 2 GB memory address, we do not want to use 2 GB of memory to get to an offset of 2 GB in memory. Instead, we want to add a block that says that it starts at memory address 2 GB and that it has one megabyte. Such an implementation does not make memory storage much more complex.

To sum up the design of byte memory storage, we will represent the storage as an array of non-overlapping blocks. Each block will start at a specific address and have a byte array representing its content. The *load* function will then find an appropriate block of bytes and create a constant expression from them to emulate read. The *missing* function will analogously check if the bytes requested to read are present in the memory and if some are missing, it will return the missing intervals.

Even though we will use our byte memory storage only as read-only, we still have to implement the whole memory storage interface. In other words, we have to also implement the *store* method. Indeed, we might trivially say that *store* call to a byte memory storage is invalid and throw some form of exception to signal that. On the other hand for the sake of completeness (even though we will never use it), we can first check of the expression written is a constant or not. If the expression is constant, we can store it in our byte array. If the expression is not a constant, there is no way to store such an expression and then the *store* method has to throw an exception or panic.

Overlay memory storage

Another special memory type we will introduce will be memory storage interconnecting multiple storage into a single one. We will call it overlay memory storage. The overlay memory storage is strongly inspired by Linux *overlayfs* [24] and the ways it is used in Linux (for example in docker).

An overlay memory storage comprises two "nested" memory storage interfaces. One storage interface will be called *base* and the other one will be called *overlay*. The idea behind overlay memory storage is that it will create an illusion of *overlay* memory storage being laid over the *base* memory storage. In other words, the *overlay* memory will be used to serve all operations it will be able to serve, but if the operation is not satisfiable using *overlay* memory storage interface, the overlay storage will fallback to the *base* layer.

As the overlay memory storage will again implement the memory storage interface it will define all three functions of the interface: *store*, *load*, and *missing*. So we can define the whole behaviour of the overlay memory storage by defining the functionality of those three methods.

Let's start with *store*, which will be the simplest to define. Each call of *store* function will result in a single *store* call to the *overlay* memory storage. This behaviour is in a way obvious as the *overlay* storage lies on ton of the *base* storage so if we modified *base* we would not be guaranteed that following *load* from the same address will load the value written.

Definition of *load* function will be slightly more complex as *load* operation will eventually touch both *base* and *overlay* memory. The *load* will first try to read as many bytes as possible from the *overlay* memory. But if some bytes loaded are missing in the *overlay* layer, the *load* function will fallback also to the *base* layer. If and only if the there are some bytes loaded missing in both *overlay* and *base*, then the *load* function fails.

The definition of *load* also guides us how to implement the *missing* method. As the *load* operation fails only in case when there are bytes missing in both *overlay* and *base*, the *missing* has to return only set of bytes missing on both layers. So to implement the *missing* call will simply call *missing* methods of both *overlay* and *base* and return an interval intersection of both those results.

6.3 Emulation implementation

Now when we have all the building blocks of our emulation, we just need to interconnect them into the emulation algorithm. So in this section, we will discuss how we initiate the emulation and how will the implementation of the *step* function look like.

6.3.1 Inputs of emulation

The very first question we should answer is where we will take the instructions to emulate. We have two options. We can either take the list of instructions directly from the sequencer or we can use instructions that were reordered by our reordering algorithm. As the latter option is more generic and it allows us to emulate both original (non-reordered) as well as reordered versions of the program, the choice seems to be obvious. So we will access instructions from our reordering algorithms as input.

Another input of our algorithm will be a set of storage (i.e. registers and memory storage). There are no specific requirements on the state of those storages, so those can be pre-filled with arbitrary values. The only requirement on those pre-filled values is that all those values have to be constant expressions. Besides that, the caller is free to fill in any values. For example, if the emulation is supposed to emulate the program from the start of its execution, the register file and memory can be pre-filled with values the operating system loader would fill in for the program.

Specifically, in *mltwist*, we provide a memory constructed by overlay memory storage. We construct the overlay memory storage by using the byte memory storage filled with program binary bytes as the *base* layer and the original (sparse) memory storage as *overlay*. This way, any *store* operation will result in modification of the *overlay* layer, but the program is still able to read its own bytes.

Besides input instructions and storage the caller also has to provide an implementation of the data-source interface. It is fully up to the caller which implementation will be provided and what will be its logic.

```
    func NewEmulation(
        c Code,
        regs RegisterStorage ,
        mems MemoryStorage ,
        ds DataSource ,
    ) Emulation
```

6.3.2 Implementation of single emulation step

The first thing each call of *step* has to do is to identify the instruction it should emulate. To do so, it will use the instruction pointer register value. One detail worth mentioning is that the address of the instruction is the address modified by the instruction reordering as we are emulating the reordered code. As we store both basic blocks and individual instructions sorted by addresses in ascending order, we can find the instruction to emulate using two binary searches – one for blocks and the other one for instructions. If we cannot find the instruction, there is something very wrong and the *step* function will return an error.

Once the *step* function knows which instruction to emulate, we have to gather all its inputs and evaluate all its outputs. In other words, we need to evaluate the inputs (register and memory loads) of all expressions. To do so, we will iterate all effects of the instruction and for each effect, we will walk all its expression trees. During the walk, we will replace each memory and register load with a respective value read from the storage. If the storage does not contain the respective value, the walk will consult the data-source interface which is in such a case mandated to provide the missing value. This way, the algorithm will replace all non-constant inputs of an expression with constants. Once the transformation is done, the emulation uses constant folding to substitute each expression with a single constant.

An important detail of the expression evaluation is that due to the potential presence of memory load instruction in an expression, the constant folding has to happen gradually. Whenever a memory load is found by the transformation, its addressing expression has to be evaluated first to get the address of the memory load. This implies that replacing of register and memory loads has to overlap with constant folding to gradually evaluate parts of the expression tree and deduce addresses to replace memory loads.

Another important detail of the expression evaluation is how the value from the data-source interface is handled. Naturally, the value received from the data-source interface is used to replace the respective register of memory load. But besides that, we also store the value provided by the interface into the register or memory storage. This approach provides several advantages. First of all, we are sure that we will ask the data-source interface only once for every particular value. The other time we need the value, it will be already in the storage. This mitigates any potential problems with inconsistencies caused by data-source calls. For example, if there were two loads of the same register and we would not store the first value provided by the data-source, the data-source could in theory provide a different value for the other load of the same register. This would make the whole emulation inconsistent, so it is good that the data-source is not allowed to do so. The second advantage of this approach is that it simplifies the implementation of the data-source interface itself as it does not have to remember all the values it ever provided to be consistent with itself. This way, the data source is sure that it will never be asked for the same value which allows the data-source implementation to be stateless.

Once all expressions are replaced by constants, the *step* emulation emulates their execution by updating storage. Given that all the expressions written as well as addresses are now constants, the emulation can simply iterate all effects and use *store* calls to update register and memory storage appropriately.

The very last thing the emulation has to handle is the potential update of an instruction pointer register. The instruction pointer register is special in a way that if an instruction writes to it (i.e. if an instruction is a jump instruction), the emulation does not have to update it. But if the instruction emulated does not write to it, the emulation has to update the instruction pointer itself to emulate the step to the next instruction. To implement this logic, we iterate over all effects and if none of them writes to the instruction pointer register, we increment the value in that register by the length of the instruction being emulated.

At this point, the emulation of single program instruction is done. All storages are updated and represent the new state of the program. Analogously, an instruction pointer has been adjusted to the next instruction. The *step* function now returns to give the caller chance to inspect or change the data of the program and to potentially alter the storage before the next *step* calls.

In total, a execution of *step* call performs those steps:

1. Find instructions based on the value of instruction pointer register.
2. Evaluate all its expressions, ask data-source for missing values.
3. Update storage by effects of the instruction.
4. Update instruction pointer (if necessary).

6.4 Summary

- Emulation uses constant expressions to represent the program state. It emulates all reads and writes of instructions using register and memory storage implementations.

- Register storage is implemented as a key-value store of register keys and values. Memory storage is implemented using an interval tree of memory ranges.

- The emulation is performed by *step* function. Each *step* call always emulates the execution of a single instruction. If some value read by an expression is missing in the storage, the emulation asks the data-source interface to provide the value.

- Between *step* calls, the emulation of the operating system is allowed to change the state of the program. For this reason, the emulation exposes both registry and memory storage. Because *mltwist* does not implement emulation of the operating system, the user is required to emulate it.

- Single *step* call performs following actions:
    1. Find an instruction to emulate using on instruction pointer value.

2. Replace all loads by constants and constant-fold the resulted expression to a single constant value.
3. Iterate over instruction effects to update storage.
4. Increment instruction pointer (if necessary).

7. Future work

In this thesis, we have so far discussed many forms of generic analysis of our intermediate code as well as many problems related to that. Unfortunately, we have seen on many occasions that some problems we are facing do not have any generic solution. In this chapter, we will discuss a few ideas on how to improve our analysis to be more accurate.

All algorithms presented in this chapter will be in heuristic algorithms. This implies that they will not solve any of the generic problems we have discussed. On the other hand, we might significantly reduce the impact of the generic problems by using those heuristics. Consequently, we could make analyses implemented in our disassembler more accurate.

7.1 Semantics-aware constant folding of binary expressions

When we described constant folding in subsection 4.3.1, we constant folded binary operations only when we knew both input parameters of the binary operation. We then did so by simple calculating of the output of binary operation based on those 2 input values. But we have absolutely ignored the properties of the binary operations themselves. In this section, we will discuss a semantics-aware constant folding that addresses exactly this problem.

The core idea behind semantics-aware constant folding is that for some binary operations a value of a single input already fully determines the output value. A good example of such an operation might be (bit-wise) NAND where *x NAND 0* is always a value with all bits set independently on the value of x. Another example is multiplication where $x * 0 = 0$ for every value of x. And we could proceed with many other such examples. So for some binary expressions, we do not need to know both parameters to identify the output. Consequently, we could improve our constant folding by adding those conditions to it and checking them for every binary operation.

The main reason why semantics-aware constant folding was not implemented in this thesis is the philosophical problem of whether such a modification of an expression produces an equivalent expression or not. When we do standard constant folding of a binary expression with both constant inputs, we are not just certain that those two constants would always produce the same result, but we are also certain that our replacement will not affect any later stage of the analysis. On the other hand with semantics-aware constant folding, we replace a binary expression with only one constant input and such a replacement might change the expression behaviour in other parts of the analysis. For example let's consider a machine code instruction representing calculation $x_5 * 0$, where x_5 is the register name. Our standard constant folding would not be able to optimize this expression, because of the non-constant register load. This implies that the dependency analysis would see register load from register x_5 in an expression tree. Consequently, it would see all dependencies with surrounding instructions writing to register x_5. But if we performed semantics-aware constant folding, we would constant-fold the expression to a constant of zero which would eliminate the load of register x_5 from the expression tree. In such a case, our dependency analysis would not see any dependencies implied by the load of register x_5. Analogously the emulation part suddenly would not need to know the value of x_5 to emulate such instruction as the expression would not contain the register load of x_5 at all.

It depends on our definition of expression equality whether we consider semantics-aware constant folding to be an equivalent modification or not. On one hand, the value of instruction output value is always preserved by semantics-aware constant folding. On the other hand, it feels counterintuitive not to see a dependency when the instruction clearly consumes the register or memory. Because of this controversy, semantics aware constant folding is not included in *mltwist*. But it might be useful to spend a bit more time in the future on defining whether modifications of semantics aware constant folding should or should not be considered equivalent.

7.2 Memory access dependency tracking

One of the problems we had in our dependency tracking was that we were not able to say whether two memory accesses with non-constant address expressions overlap. An unpleasant consequence of this is that we had to assume a dependency between any two instructions which access memory. In this section, we will discuss an algorithm capable of deducing that certain pairs of memory accesses are independent.

The inability to track memory dependencies is especially limiting in situations when two instructions load or store values in two different places in the program stack. In such a case, it is obvious to the reader of the code that such memory accesses are independent. Unfortunately, our dependency analysis cannot deduce it. The reason is that our dependency analysis would need addresses of both memory accesses to be constants as this is the only case our dependency analysis is currently able to handle.

The problem with the approach described above is that we ask too specifically. We have started our reasoning about dependencies with a statement that if we know the address (and *width*) of the memory access, we can deduce whether two memory accesses are dependent. This is for sure a true statement. On the other hand, this implication does not work the other way around – based on the knowledge of whether two memory accesses overlap or not, we are not able to deduce the address of either of those two memory accesses. This implies that the question we ask about the exact memory address is more specific than the knowledge we need. In other words, we should ask a different, less generic question which will require us to know less of those two instructions.

For simplicity, we will start our reasoning considering single-byte memory accesses exclusively. We consider this simplified model to avoid reasoning about access *width* for a moment. As an example, we will consider a sequence of two instructions: *st %sp+0, %x5* and *st %sp+1, %x6*. Those two instructions obviously store two register values to the program stack. It is also obvious that those two instructions are independent on each other and they can be reordered freely.

We can describe our example case using a single equation: $sp + 0 = sp + 1$. We produced this equation by comparing addresses of instruction accesses. When we then find a non-negative integer that solves the equation, we know that those two memory accesses can overlap at least for some value of stack pointer. On the other hand, if we find out that the equation has no non-negative integer solution, we are certain that for any value of *sp*, those two memory accesses will not overlap. Based on this knowledge, we can conclude that those two instructions are independent on each other.

Naturally, the algorithm described above will not work in all possible cases. For example, if the first load used a register value to add to the stack pointer (i.e. *st %sp+%x14, %x5*), there would always exist a possible value of x_{14} such that the equation would have a solution. This algorithm is not strong enough to recognize all false positive dependencies. On the other hand, because the majority of memory operations are loads from or stored to stack, this algorithm could remove most of the false positive memory dependencies in our program.

Before we start to consider multi-byte memory accesses, we will perform one transformation of our equation. Previously, we were interested in a negative result of the equation. In other words, we knew that two instructions are independent when there was no solution to the equation above. We can also rephrase the problem above into the form where we will look for a positive result of the following inequality: $sp + 0 \neq sp + 1$. We can further rephrase this non-equation into two inequalities: $sp + 0 < sp + 1$ and $sp + 0 > sp + 1$. If one of those two inequalities holds for all possible non-negative integer values of *sp*, we know that the two instructions are not dependent.

Now we can use the inequality formulation to generalize our algorithm for multi-byte memory accesses. In our formulas, we will always use the symbol w to represent *width* of memory access. Multi-byte memory access will then access bytes $[a, a + w - 1]$ (i.e. $[a, a + w)$), where a is an address of the memory access.

As the example we used before will result in overlapping memory accesses for any *width* greater than one, we need to introduce a new example. We will consider the following two instructions: *st %sp+0, %x5* and *st %sp+c, %x6*. In our example, c is an arbitrary (possibly negative) integer constant. It is worth noting that even though we use generic c in our equations, the value of c is always known to the disassembler as it is the *width* of the memory operation.

Our two multi-byte memory accesses can then collide on multiple bytes. Rather than to write an equation for individual collisions, we will write two inequalities which will represent both non-colliding cases:

$$sp + 0 \geq sp + c + w_2$$

$$sp + 0 + w_1 \leq sp + c$$

We constructed those two inequalities by considering one memory access range as fixed and writing down all addresses of the second access which does not collide with it. Given that the case is symmetric, it does not matter which access we start with. So without loss of generality, we can consider the first memory access to address *sp + 0* to be fixed. The first inequality then asks for which values of *sp*, the second memory access touches bytes which are before the bytes touched by the first access. Analogously the latter inequality asks for which values of *sp*, the second memory access touches bytes which are behind bytes touched by the first memory access. If we find out that at least one of those equations holds for all non-negative integer values of *sp*, we can guarantee that those two memory accesses never overlap and that they are independent.

One problem with the formulation above is that it might be quite hard to prove that one of the inequalities holds for any non-negative integer. Especially because for some non-negative integers it might be the former inequality and for some, it can be the latter one. It would be much better to have a system of inequalities where we would ask if there exists at least one non-negative integer solution. We can do so by swapping the meaning of a solution. We can write a system of inequalities that will describe all overlapping memory accesses. If we find a solution to this system, it will imply that the two instructions are dependent because their memory accesses might overlap. On the other hand, if we prove that the system does not have a non-negative integer solution, then we know that the two instructions are independent. We will construct such a system of inequalities by simply swapping the inequality signs:

$$sp + 0 < sp + c + w_2$$

$$sp + 0 + w_1 > sp + c$$

Implementation of this analysis is out of the scope of this thesis. But the idea of memory access dependency tracking is solid and well-defined. Unfortunately is unclear how could we implement such an analysis on top of generic expressions. This is the reason why we leave the implementation of memory access dependency tracking as potential future work.

7.3 Basic blocks joining

When we described dependency analysis, we very frequently argued that instructions consuming a register or memory value are unfortunately opaque to us. This has been problematic especially during the basic block analysis when we were not able to analyze all possible jump targets. There were also other problematic situations for example identifying addresses of memory loads and writes. But the basic block analysis was the biggest problem as we might miss some control dependencies. We also discussed back then that it is not an unusual pattern that the value of the register is stored just a few instructions in advance. Consequently, we might try to deduce the value in the register or memory by looking at instructions preceding the load.

The idea of basic block joining is inspired by our emulation process. Because we know where basic blocks are, we could walk all instructions in that block and store the expressions they write. As later instructions in the basic block typically consume values produced by former instructions, we should be able to deduce some more information about their inputs by doing so. Specifically, in our example with constant being stored to a register followed by a jump to address in that register, we would be able to deduce the jump address.

We could describe basic block joining as an emulation on top of generic expressions instead of constants[1]. The only difference is that this time, we "emulate" each basic block independently. The reason why we cannot pass the basic block boundary is that in emulation, we simulate a single program execution for a given set of inputs. In basic block joining, we care about properties the code has for an arbitrary input. To achieve that, we have to respect the basic block boundary and treat any value coming from outside of the basic block as unknown.

In the simplest possible case, we would implement basic block joining by walking each instruction of a basic block. For each instruction, we would first replace all its register and memory reads in all expressions with expressions read from the register or memory storage. Then, we would update the storage with the effects of the instruction. By doing so for every instruction in a basic block, we would get more information about jump targets and memory addresses later in the block. Using this information, we could make our dependency analysis much more accurate.

It is noting that basic block joining will never be sufficient to analyze all dependencies. The reason is that some values might come to the basic block from other basic blocks. As our algorithm joins only a single basic block, we still would not be able to analyze all expressions in the code.

Unfortunately, basic block joining could result in a paradox. The problem is that as we join one basic block after another, we might find more jump addresses which could split basic blocks we already joined. In other words basic block joining can invalidate its own results by finding out that some basic block it joined comprises in reality of multiple basic blocks and though the joining was invalid.

The existence of this paradox is the reason why basic block joining was not implemented in this thesis. It does not seem to be impossible to remedy or at least lower the impact of this paradox. Unfortunately, remediation of the paradox described above is out of the scope of this thesis. On the other hand, the idea of basic block joining is promising. It just requires some additional work to be done in future development.

[1]This also explains why we designed storage for emulation to store generic expressions in section 6.2. The intention was naturally to reuse the same storage for basic block joining later.

Conclusion

The goal of the thesis was to implement a disassembler capable of understanding instruction meaning. This disassembler should then be able to reorder machine code instructions and guarantee that the meaning of the program will be preserved. Another goal of the thesis was to emulate the instruction execution based on the understanding of its meaning.

We started our effort with a discussion on how can we effectively represent the meaning of machine code instructions. We discovered a few problems along the way but in the end, we were able to design a small but powerful set of intermediate code instructions. At the end of the chapter, we then discussed how can we use our instructions to represent real CPU instructions.

The disassembling effort started with a single binary. We briefly explored possibilities of executable format interpretations, but we quickly shifted to parsing of instructions. We wanted our parsing logic to be extendable for other CPU architectures. To achieve that, we designed a platform-specific parser interface that allows us to extend the disassembler with machine code parsers for different CPU architectures. We then demonstrated the usage of our parser interface by implementing a RISC-V machine code parser.

Before we started the discussion on dependency tracking, we introduced a few transformations which we later used as the building block for many of our algorithms. Then we could finally start to analyze dependencies between instructions. We have discussed different dependency types and algorithms to find them. And last but not least we proposed how to represent those dependencies in a way that allows us to effectively reorder instructions in the machine code.

The last goal of our thesis was to emulate the program execution. In this chapter, we found out that we have almost all building blocks ready and we were able to emulate the program in quite a straightforward way. Of course, there were some challenges, but it turned out that emulation is not that hard with all the the infrastructure we have built so far.

At the very end of this thesis, we have discussed a few ideas on how to improve the disassembler in the future. It is true that none of those algorithms has been properly thought thru and that it would take a bit more effort to implement them in the future. Yet hopefully, this chapter was inspiring for the reader.

All algorithms and concepts discussed in this thesis were used to implement a proof-of-concept disassembler called *mltwist*.

To sum the whole thesis up, we have completed our goals. We implemented a platform-dependent disassembler that understands instruction meaning. It turns out that we can use this knowledge to perform advanced analysis of machine code which even commercial compilers are not able to do.

Bibliography

[1] Instruction scheduling - wikipedia. `https://en.wikipedia.org/wiki/Instruction_scheduling`. (Accessed on 06/13/2022).

[2] Dynamic linking vs. dynamic loading — baeldung on computer science. `https://www.baeldung.com/cs/dynamic-linking-vs-dynamic-loading`. (Accessed on 07/16/2022).

[3] Hex rays - state-of-the-art binary code analysis solutions. `https://www.hex-rays.com/ida-pro/`. (Accessed on 06/13/2022).

[4] Ghidra. `https://ghidra-sre.org/`. (Accessed on 06/13/2022).

[5] jan-dubsky/mltwist. `https://github.com/jan-dubsky/mltwist`.

[6] Introducing llvm intermediate representation — packt. `https://www.packt.com/introducing-llvm-intermediate-representation/`. (Accessed on 06/25/2022).

[7] Intel® 64 and ia-32 architectures software developer manuals. `https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html`. (Accessed on 06/26/2022).

[8] Universal gates: Nand and nor. `http://www.uop.edu.pk/ocontents/Lec-10-universal%20gates.pdf`. (Accessed on 06/20/2022).

[9] Deldsim - implementation of ex-or gate using nand gate. `https://www.deldsim.com/study/material/5/implementation-of-ex-or-gate-using-nand-gate/`. (Accessed on 06/20/2022).

[10] Two's complement. `https://www.cs.cornell.edu/~tomf/notes/cps104/twoscomp.html`. (Accessed on 06/20/2022).

[11] Logical vs. arithmetic shift - open4tech. `https://open4tech.com/logical-vs-arithmetic-shift/`. (Accessed on 06/22/2022).

[12] riscv-spec-v2.2.pdf. `https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf`. (Accessed on 06/22/2022).

[13] assembly - how many least-significant bits are the same for both an unsigned and a signed multiplication? - stack overflow. `https://stackoverflow.com/questions/39147353/how-many-least-significant-bits-are-the-same-for-both-an-unsigned-and-a-signed-m/72309286#72309286`. (Accessed on 06/22/2022).

[14] std::memory_order - cppreference.com. `https://en.cppreference.com/w/cpp/atomic/memory_order`. (Accessed on 06/22/2022).

[15] Position independent code (pic) in shared libraries - eli bendersky's website. `https://eli.thegreenplace.net/2011/11/03/position-independent-code-pic-in-shared-libraries/`. (Accessed on 06/30/2022).

[16] Can we expect a 128-bit processor before 2030? — by apurv jha — medium. https://believeinjha.medium.com/can-we-expect-a-128-bit-processor-before-2030-c04d6336cfd5. (Accessed on 07/08/2022).

[17] Depth first search or dfs for a graph - geeksforgeeks. https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/. (Accessed on 07/02/2022).

[18] Breadth first search or bfs for a graph - geeksforgeeks. https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/. (Accessed on 07/02/2022).

[19] What are data dependencies explain name dependencies with examples between two instructions. https://eduladder.com/viewquestions/819/What-are-data-dependencies-Explain-name-dependencies-with-examples-between-two-instructions-. (Accessed on 07/04/2022).

[20] vclock_gettime.c source code [linux/arch/x86/entry/vdso/vclock_gettime.c] - woboq code browser. https://code.woboq.org/linux/linux/arch/x86/entry/vdso/vclock_gettime.c.html. (Accessed on 07/07/2022).

[21] io_uring(7) — arch manual pages. https://man.archlinux.org/man/io_uring.7.en. (Accessed on 07/07/2022).

[22] Interval trees: One step beyond bst. https://iq.opengenus.org/interval-tree/. (Accessed on 07/07/2022).

[23] Interval tree - wikipedia. https://en.wikipedia.org/wiki/Interval_tree. (Accessed on 07/07/2022).

[24] Overlay filesystem — the linux kernel documentation. https://www.kernel.org/doc/html/latest/filesystems/overlayfs.html. (Accessed on 07/07/2022).

List of Figures