Timing Analysis - From Predictions to Certificates

Nuno Miguel Pires Gaspar

Submitted to University of Beira Interior in candidature for the degree of Master of Science in Computer Science

Supervised by Simão Melo de Sousa
Co-Supervised by Rogério Reis

Departamento de Informática
University of Beira Interior
Covilhã, Portugal
http://www.di.ubi.pt
I always liked the acknowledgement chapter. In spite that they all follow, more or less, the same structure (thank your advisors, thank your lab colleagues, thank your family), they always tell you something more about the author. I suppose this is natural, in view of the fact that this is most likely the chapter that is less often read carefully, and thus the author feel the freedom to get creative without fear. Also, this is the only place where you can be biased. Very few keep it simple and straight to the point. Inside jokes, clever remarks, Asian sentences, you name it. I have seen it all! Me? Well, I just wrote this paragraph.

First of all, I would like to thank my Advisor, Professor Simão Melo de Sousa, for being the one who first got me into this challenging but fascinating world of Formal Methods. Only time will tell if I will make a career out of it. To my Co-Advisor, Professor Rogério Reis, for always being very friendly, and for all the extensive (and painful. Painful, in a good way!) reviews.

I should also thank all my colleagues from University of Beira Interior. In particular, the ones from the RELEASE - RELiable And SEcure Computation Group, Carlos Carloto, Diogo Fialho, Vasco Nicolau and Joaquim Tojal. Their presence made the hard days more tolerable. Oh, scratch that, I meant enjoyable.

To the PhD-wannabes I met in the context of the RESCUE - RELiable and Safe Code execUtion for Embedded systems project, David Pereira and Vítor Rodrigues. For being very friendly, and always willing to help out.

Last, and definitely not least, to my Mom. For supporting me all these years, providing me the education that I have today. Also, for her love and patience in my, too many often, absent days.
Abstract

In real-time systems timing properties must be satisfied in order to guarantee that deadlines will be met. In this context, the calculation of the worst-case execution time (WCET) is of paramount importance for schedulability analysis. However, this problem can be difficult if the underlying architecture possesses features like caches and pipelines.

This thesis presents all the necessary steps for the safe and precise WCET calculation. We focus ourselves in the use of static analysis-based methods, and in the ARM architecture as target platform. Moreover, in order to ensure the correctness of our calculation to a program consumer, we produce a certificate (or proof) whose validity entails compliance with the calculated WCET. This evidence permits to locally validate the calculated WCET, avoiding the need of a blind confidence on the producer.
Keywords

Timing Analysis; Worst-Case Execution Time; Static Analysis; Fixpoint computation; Abstract Interpretation; Abstraction-Carrying Code
Contents

Acknowledgements iii
Abstract v
Keywords vii
Contents ix
List of Figures xi
List of Tables xiii

1 Introduction 1
1.1 Motivation 1
1.2 Approach 2
1.2.1 Abstract Interpretation 3
1.2.2 Abstraction-Carrying Code 5
1.3 Contributions 6
1.4 Organization of this thesis 6

2 The ARM Architecture 9
2.1 Main Characteristics 9
2.2 ARM Instruction set 11
2.3 Pipeline 14
2.3.1 Instruction Cycle Count Summary 15
2.3.2 Pipeline Stalls 17
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>ACCEPT: Abstraction-Carrying Code Platform for Timing validation</td>
<td>3</td>
</tr>
<tr>
<td>2.1</td>
<td>Acessible Registers in the different operating modes</td>
<td>10</td>
</tr>
<tr>
<td>2.2</td>
<td>Current Program Status Register (CPSR) register user-configurable bits</td>
<td>11</td>
</tr>
<tr>
<td>2.3</td>
<td>The instruction pipeline</td>
<td>14</td>
</tr>
<tr>
<td>3.1</td>
<td>Basic concepts of Timing Analysis</td>
<td>20</td>
</tr>
<tr>
<td>3.2</td>
<td>Timing Anomaly caused by speculation</td>
<td>21</td>
</tr>
<tr>
<td>3.3</td>
<td>Core components of a static timing-analysis tool</td>
<td>23</td>
</tr>
<tr>
<td>4.1</td>
<td>Basic Block Graph</td>
<td>28</td>
</tr>
<tr>
<td>4.2</td>
<td>The number of times that the incoming and outgoing edges are crossed is equal, and so is the execution count of $v$</td>
<td>47</td>
</tr>
<tr>
<td>4.3</td>
<td>Basic block graph</td>
<td>48</td>
</tr>
<tr>
<td>5.1</td>
<td>CATA - Certificates for Timing Analysis</td>
<td>51</td>
</tr>
<tr>
<td>5.2</td>
<td>Certificates for Timing Analysis (CATA)’s architecture</td>
<td>52</td>
</tr>
</tbody>
</table>
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>ARM Instruction Set relevant for the purposes of our work</td>
<td>12</td>
</tr>
<tr>
<td>2.2</td>
<td>Condition Codes</td>
<td>13</td>
</tr>
<tr>
<td>2.3</td>
<td>Instruction cycle counts</td>
<td>16</td>
</tr>
<tr>
<td>4.1</td>
<td>$kill_{rd}$ function</td>
<td>32</td>
</tr>
<tr>
<td>4.2</td>
<td>$gen_{rd}$ function</td>
<td>33</td>
</tr>
<tr>
<td>4.3</td>
<td>$kill_{rd}$ applied to the ARM program in listing 4.2</td>
<td>34</td>
</tr>
<tr>
<td>4.4</td>
<td>$gen_{rd}$ applied to the ARM program in listing 4.2</td>
<td>35</td>
</tr>
<tr>
<td>4.5</td>
<td>data-flow equations</td>
<td>36</td>
</tr>
<tr>
<td>4.6</td>
<td>ARM program’s fixpoint for our Reaching Definition Analysis</td>
<td>37</td>
</tr>
<tr>
<td>4.7</td>
<td>$gen_{ls}$ function</td>
<td>39</td>
</tr>
<tr>
<td>4.8</td>
<td>$gen_{ls}$ applied to the ARM program in listing 4.2</td>
<td>40</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Real-time systems can be seen as sets of tasks, that are expected to perform some functionality under predefined timing constraints. In general, in order to ensure a correct system behaviour (schedulability analysis), an upper bound estimative for the Worst-Case Execution Time (WCET) of each task is required.

To improve its performance, modern processors include mechanisms like memory caches and pipelines which make instruction’s execution time context dependent, increasing the difficulty of static timing analysis. To cope with this, one could be tempted to always assume the local worst case scenario (e.g. cache miss) in order to obtain safe predictions, but two problems could arise of such approach. On one hand, it could lead to an excessive over-approximation of the actual WCET, resulting in a waste of hardware resources, and since most modern real-time systems are mass-produced, the determination of tight predictions can lead to production costs cut [1]. By other hand, due to timing anomalies [2], the obtained prediction could in fact, be unsafe (an under-approximation).

1.1 Motivation

The determination of safe and tight upper bounds for the WCET has been the object of intensive study in the literature [3], yet, to the best of our knowledge, there is no attempt to provide some warranties of the correctness of the predicted WCET.

Previous works related with the derivation of execution time bounds embrace the use

---

1 In the following, only the term cache is used for simplicity.
of formal methods [4, 5], inherently increasing the confidence that one can put on its approach. Furthermore, the use of sound techniques like Abstract Interpretation [6] has already made its way into the industry [7], proving to be well suited for the determination of tight, but safe, execution time bounds.

However, in the context of mobile code safety, which has been progressively gaining notoriety in the sphere of real-time systems [8], both at research and industry levels, since they represent an enabling technology to tackle the limitations of standard client-server based approaches, in spite of the fact that previous methodologies are leveraged by the use of a formal approach, one would still have to put his faith on a potential untrusted third party, without being able to independently validate the correctness of the predicted WCET.

For instance, embedded systems such as coral sensors or control computers for satellites, which cannot be easily reachable, would highly profit from mobile real-time code [9]. In fact, even a software/system update can be seen as mobile code. Thus, being able to independently validate its timing behaviour would be of major interest for such systems. Moreover, this could also be a valuable asset for original equipment manufacturers and sub-contractors applications.

This lack of an independent validation process is the issue that we address in this work.

1.2 Approach

The general framework of our proposal is illustrated by Figure 1.1. The overall project in which this thesis is included in, has as starting point the extension of the C programming language with annotations. This is accomplished by following a Design-by-Contract approach, that define the intended timing properties for each function. The timing specification of the main function is of most importance, however one may also want to define some constraints on the auxiliary functions. Moreover, by placing these annotations directly into the source code, we are also able to express valuable informations for the subsequent timing analysis, such as infeasible paths and loop bounds. This is achieved by the use of a WCET-aware compilation process, targeting the ARM instruction set [10], that preserve the annotations semantics.

Due to the intricacies of modern processors, only at the hardware level one can accu-
1.2. APPROACH

rately perform timing analysis [3]. However, feedback about the compliance of the given timing specification should be done at source-code level. Hence, in order to perform timing validation w.r.t. the functions’ timing specification, we perform the timing analysis at machine-code level, taking into account the effects of the hardware specific features, and alert of any possible non-compliance through to use of back annotations [11]. The idea of this mechanism is to propagate the timing information back to the source-code level, warning the system developers about the violation w.r.t. the timing specification. However, details on this mechanism are out of the scope of this work and will be reported elsewhere.

The concept of Abstraction-Carrying CodE Platform for Timing validation (ACCEPT) is that a potential untrusted program producer supplies the code augmented with a certificate. This certificate will allow a program consumer to validate the received code w.r.t. to its timing behaviour.

1.2.1 Abstract Interpretation

Let the set of execution times of a program \( P \) be denoted by \([P]\). The problem of verifying the compliance of the given timing specification can thus be formulated as follows:

\[ P \text{ respects the timing specification } \mathcal{I} \text{ if } [P] \subseteq \mathcal{I}, \]
where $\mathcal{I}$ stands for the intended timing behaviour, i.e., the set of accepted execution times. The idea is to express the set of concrete states occurring at each program point, i.e. the collecting semantics\(^2\) of $P$ as the fixpoint of a set of recursive equations.

In general, however, the state space to be considered is too large to exhaustively explore all possible executions and some abstraction of the application domain is required in order to make the timing analysis feasible. With this in mind, our approach relies on abstract interpretation\(^6\) as the underlying technique. With its use, not only it provides us an adequate framework to reason about, but also with an elegant way to infer an abstract model of the program that can play the role of a certificate.

In the abstract interpretation framework, a program $P$ is interpreted over a simpler abstract domain $\mathcal{D}_\alpha$. This abstract domain permits to trade efficiency over precision, i.e., although it is an approximation, by computing the fixpoint over this abstract domain, we will be able to produce precise, yet safe, (over-)approximations of the collecting semantics.

The fixpoint calculation over this abstract domain will allow us to safely predict the processor behaviour for the program's execution (e.g. pipeline stall). With that information, and following the approach from the standard WCET architecture\(^{13}\), one can compute the execution times of the program's basic blocks\(^3\) and then obtain the WCET and Best-Case Execution Time (BCET), by determining their paths through the control-flow graph. This is achieved by solving the corresponding integer linear program, maximizing it for execution time in case of WCET computation, and analogously for BCET computation.

Let $[P]_\alpha$ be the set of execution times calculated over the abstract domain $\mathcal{D}_\alpha$. It is clear that $[P]_\alpha$ is lower- and upper-bounded by BCET and WCET respectively. Moreover, since the comparison between actual and intended semantics is easier if done in the same domain, we assume that the intended timing specification is also given in the abstract domain, i.e., $\mathcal{I}_\alpha \in \mathcal{D}_\alpha$.

The problem of verifying the compliance with the given timing specification can now be reformulated as:

$$P \text{ respects the timing specification } \mathcal{I}_\alpha \text{ if } [P]_\alpha \subseteq \mathcal{I}_\alpha.$$  

\(^2\)Also called static semantics in the literature\(^6\).

\(^3\)A basic block is a maximal sequence of straight-line code in the program, i.e., it has one entry point and one exit point, without any branches contained within it.
1.2. APPROACH

At this stage, feedback regarding any possible non-compliance of $P$ w.r.t. the timing specification can be reported through the use of back-annotations [11], permitting the system developers to proceed accordingly.

1.2.2 Abstraction-Carrying Code

Proof-Carrying Code (PCC) [14] is a general mechanism enabling a program consumer to locally check the validity of the code w.r.t. some safety policy. The inherent key benefit is that there is no need to trust any third party. However, there are three essential challenges for PCC to be used in practice:

(i) definition of expressive safety policies,

(ii) automatic generation of the certificate, i.e., proving the program correct, and

(iii) efficient certificate checking in the consumer side.

In the context of mobile code safety, most approaches rely on theorem proving, whereas Abstraction-Carrying Code (ACC) [15] relies on abstract interpretation.

In ACC, and in particular for the purposes of our platform, the above challenges are addressed by (i) getting hold of the effects that the processor specific features (e.g. caches, pipeline) have on the execution time, which has already been addressed in the literature [16, 17]; (ii) using a fixpoint static analyser to automatically infer an abstract model of the program, which can be then used as a certificate; and (iii) by a simple, easy-to-trust fixpoint checker.

As mentioned earlier (see Subsection 1.2.1), this fixpoint computation will yield the execution times of the program’s basic blocks, and the remaining task of calculating the BCET and WCET being achieved by means of integer linear programming techniques. However, we can also let this fixpoint play the role of a certificate. The idea of ACC is that when a consumer receives a program, it is augmented with a certificate. Since the certificate is supposed to be a fixpoint, another iteration over it cannot produce any changes. Thus, if it is a fixpoint, then the consumer can independent and locally compute the BCET and WCET and check if the program complies with the intended timing behaviour.
1.3 Contributions

The work reported here is included in a broader effort to provide a source level feedback on a BCET and WCET computation platform with certificate generation in the context of mobile code. This platform considers an high-level annotation language, preserving its semantics through a WCET-aware compilation process, and a back-annotation mechanism [11]. However, these features are not a subject of this work, and will be discussed elsewhere. Here, we focus on the low level interface. In this sense, this work reports on the certificate generation and validation, and intends to introduce and justify the underlying architecture.

The main contributions of this work are the following:

- An open-source architecture for BCET and WCET computation that follows state of the art algorithms for its implementation.
- The first application of the Abstraction-Carrying Code concepts to the static timing analysis field.

The following publications resulted in the context of the work developed in this thesis:


1.4 Organization of this thesis

The remainder of this thesis is organised as follows. Chapter 2 presents an overview of the ARM 7 architecture. The main ingredients of real-time systems and timing analysis
1.4. **ORGANIZATION OF THIS THESIS**

are discussed in Chapter 3. The theory underlying the tool developed in the context of this thesis is discussed in Chapter 4. Chapter 5 presents this tool. Finally, conclusions and directions for future work are discussed in Chapter 6.
Chapter 2

The ARM Architecture

In this Chapter we present an overview of the ARM 7 architecture that was the target for our timing analysis platform. The instruction set and the major features of the considered processor, ARM7TDMI-S, are shown and discussed.

2.1 Main Characteristics

The ARM7TDMI-S processor is a member of the ARM family of general-purpose 32-bit microprocessors, known for their high performance while keeping a very low-power consumption and gate count. It is characterised by being an highly popular Reduced Instruction Set Computer (RISC) architecture. It is very common in the market, being broadly incorporated in widespread embedded systems such as the Game Boy Advance, Nintendo DS or the Apple iPod.

Although being a 32-bit microprocessor, it is also capable of running a 16-bit instruction set, known as Thumb [10], and is actually, the reason why the ARM7TDMI-S processor has a "T" in its name. The Thumb instruction set helps the achievement of a greater code density, and in fact, can be seen as a compressed form of a subset of the ARM instruction set. It is usual to mix the ARM and Thumb instruction sets in a single application, this flexibility allows to use one instruction set or another on a routine-by-routine basis. In order to determine how the instructions stream is to be interpreted, whether in ARM or Thumb state, the processor examines the 5th bit, known as the T bit, of the CPSR register. If the T bit is set, the instruction stream is interpreted as 16-bit Thumb instructions, otherwise is interpreted as 32-bit ARM instructions.
In ARM state, whenever operating in System or User mode, as illustrated in Figure 2.1, there are 16 general registers and a status register accessible at any time. The general registers range from R0 to R15, with the last six having alias names. Three of this last six have special roles\footnote{The remaining three registers, in descending order, are aliased: Intra-Procedure call scratch register, Frame Pointer and Stack Limit, respectively.} For instance, R13 is also known as the Stack Pointer (SP), R14 as the Link Register (LR), and R15 as the Program Counter (PC). The remaining 13 registers have no special hardware purpose. Their uses are defined solely by software.

Similarly to the state, either ARM or Thumb, the contents of the CPSR register also determines the operating mode, and thus the accessible registers. Moreover, besides holding the bit that determines how the instruction stream is to be interpreted (i.e. either in ARM or Thumb state), and the operating mode, it also contains other flags, that, in fact, are particularly relevant for the purposes for this work. The user-configurable bits of the CPSR register are depicted in Figure 2.2. In order to serve our interests, we will be

### Figure 2.1: Accessible Registers in the different operating modes

<table>
<thead>
<tr>
<th>System &amp; User</th>
<th>FIQ</th>
<th>Supervisor</th>
<th>Abort</th>
<th>IRQ</th>
<th>Undefined</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>R0</td>
<td>R0</td>
<td>R0</td>
<td>R0</td>
<td>R0</td>
</tr>
<tr>
<td>R1</td>
<td>R1</td>
<td>R1</td>
<td>R1</td>
<td>R1</td>
<td>R1</td>
</tr>
<tr>
<td>R2</td>
<td>R2</td>
<td>R2</td>
<td>R2</td>
<td>R2</td>
<td>R2</td>
</tr>
<tr>
<td>R3</td>
<td>R3</td>
<td>R3</td>
<td>R3</td>
<td>R3</td>
<td>R3</td>
</tr>
<tr>
<td>R4</td>
<td>R4</td>
<td>R4</td>
<td>R4</td>
<td>R4</td>
<td>R4</td>
</tr>
<tr>
<td>R5</td>
<td>R5</td>
<td>R5</td>
<td>R5</td>
<td>R5</td>
<td>R5</td>
</tr>
<tr>
<td>R6</td>
<td>R6</td>
<td>R6</td>
<td>R6</td>
<td>R6</td>
<td>R6</td>
</tr>
<tr>
<td>R7</td>
<td>R7</td>
<td>R7</td>
<td>R7</td>
<td>R7</td>
<td>R7</td>
</tr>
<tr>
<td>R8</td>
<td>R8_fiq</td>
<td>R8</td>
<td>R8</td>
<td>R8</td>
<td>R8</td>
</tr>
<tr>
<td>R9</td>
<td>R9_fiq</td>
<td>R9</td>
<td>R9</td>
<td>R9</td>
<td>R9</td>
</tr>
<tr>
<td>R10 (SL)</td>
<td>R10_fiq</td>
<td>R10</td>
<td>R10</td>
<td>R10</td>
<td>R10</td>
</tr>
<tr>
<td>R11 (FP)</td>
<td>R11_fiq</td>
<td>R11</td>
<td>R11</td>
<td>R11</td>
<td>R11</td>
</tr>
<tr>
<td>R12 (IP)</td>
<td>R12_fiq</td>
<td>R12</td>
<td>R12</td>
<td>R12</td>
<td>R12</td>
</tr>
<tr>
<td>R13 (SP)</td>
<td>R13_fiq</td>
<td>R13_svc</td>
<td>R13_abt</td>
<td>R13_irq</td>
<td>R13_und</td>
</tr>
<tr>
<td>R14 (LR)</td>
<td>R14_fiq</td>
<td>R14_svc</td>
<td>R14_abt</td>
<td>R14_irq</td>
<td>R14_und</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CPSR</th>
<th>CPSR</th>
<th>CPSR</th>
<th>CPSR</th>
<th>CPSR</th>
<th>CPSR</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPSR_fiq</td>
<td>SPSR_svc</td>
<td>SPSR_abt</td>
<td>SPSR_irq</td>
<td>SPSR_und</td>
<td></td>
</tr>
</tbody>
</table>
devoting more attention to the *Condition Code Flags*.

The N, Z, V and V (meaning Negative, Zero, Carry and Overflow, respectively) bits are collectively known as the condition code flags, often referred simply as flags. As we shall see in section 2.2, in the ARM Assembly language all instructions can be conditionally executed. Hence, the purpose of these flags is to determine whether an instruction is to be executed or ignored.

### 2.2 ARM Instruction set

The ARM instruction set summary is given in table 2.1.

As mentioned earlier, in the ARM Assembly language the instructions can be conditionally executed. This is achieved by appending a suffix to the instruction’s mnemonic. The possible suffixes are known as the *condition codes*, and are listed in table 2.2.

As shown in table 2.2, there are fifteen condition codes. Indeed, the last one always evaluates to *true*, meaning that one instruction without a condition code, and the same suffixed with the condition code AL, are semantically equivalent. For the sake of clarity, let us consider the following two examples.

1. MOVAL r1 #5
2. BEQ LABEL

In example (1), the condition code AL always evaluates to *true*, thus, this instruction is semantically equivalent to MOV R1 #5. However, in (2) the instruction would only be executed if the CPSR's Z bit would be set.

Furthermore, the *Data Processing* and *Multiply and Multiply-Accumulate* instructions can also be suffixed by S (for instance, ADDS R1 R3). If this is the case, it means that...
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Branch and Exchange</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BX</td>
<td>Branch and Exchange</td>
<td>( R_{15} := R_n, T_{bit} = R_n[0] )</td>
</tr>
<tr>
<td><strong>Branch and Branch with Link</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>Branch</td>
<td>( R_{15} := \text{address} )</td>
</tr>
<tr>
<td>BL</td>
<td>Branch with Link</td>
<td>( R_{14} := R_{15}, R_{15} := \text{address} )</td>
</tr>
<tr>
<td><strong>Data Processing</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOV</td>
<td>Move register or constant</td>
<td>( R_n := O_p^2 )</td>
</tr>
<tr>
<td>MVN</td>
<td>Move negative register</td>
<td>( R_n := 0xFFFFFFFF \ xor O_p^2 )</td>
</tr>
<tr>
<td>CMP</td>
<td>Compare</td>
<td>( CPSR_{flags} := R_n - O_p^2 )</td>
</tr>
<tr>
<td>CMN</td>
<td>Compare Negative</td>
<td>( CPSR_{flags} := R_n + O_p^2 )</td>
</tr>
<tr>
<td>TEQ</td>
<td>Test bitwise equality</td>
<td>( CPSR_{flags} := R_n \ xor O_p^2 )</td>
</tr>
<tr>
<td>TST</td>
<td>Test bits</td>
<td>( CPSR_{flags} := R_n \ and O_p^2 )</td>
</tr>
<tr>
<td>AND</td>
<td>AND</td>
<td>( R_d := R_n \ and O_p^2 )</td>
</tr>
<tr>
<td>EOR</td>
<td>Exclusive or</td>
<td>( R_d := R_n \ xor O_p^2 )</td>
</tr>
<tr>
<td>SUB</td>
<td>Subtract</td>
<td>( R_d := R_n - O_p^2 )</td>
</tr>
<tr>
<td>RSB</td>
<td>Reverse subtract</td>
<td>( R_d := O_p^2 - R_n )</td>
</tr>
<tr>
<td>ADD</td>
<td>Add</td>
<td>( R_d := R_n + O_p^2 )</td>
</tr>
<tr>
<td>ADC</td>
<td>Add with carry</td>
<td>( R_d := R_n + O_p^2 + \text{Carry} )</td>
</tr>
<tr>
<td>SBC</td>
<td>Subtract with carry</td>
<td>( R_d := R_n - O_p^2 - 1 + \text{Carry} )</td>
</tr>
<tr>
<td>RSC</td>
<td>Reverse Subtract with carry</td>
<td>( R_d := O_p^2 - R_n - 1 + \text{Carry} )</td>
</tr>
<tr>
<td>ORR</td>
<td>OR</td>
<td>( R_d := R_n \ or O_p^2 )</td>
</tr>
<tr>
<td>BIC</td>
<td>Bit clear</td>
<td>( R_d := R_n \ and \ not O_p^2 )</td>
</tr>
<tr>
<td><strong>PSR Transfer</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MRS</td>
<td>Move PSR status/flags to register</td>
<td>( R_n := \text{PSR} )</td>
</tr>
<tr>
<td>MSR</td>
<td>Move register to PSR status/flags</td>
<td>( \text{PSR} := R_m )</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MUL</td>
<td>Multiply</td>
<td>( R_d := R_m \ast R_s )</td>
</tr>
<tr>
<td>MLA</td>
<td>Multiply and Accumulate</td>
<td>( R_d := R_m \ast R_s + R_n )</td>
</tr>
<tr>
<td><strong>Single Data Transfer</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDR</td>
<td>Load register from memory</td>
<td>( R_d := \text{address} )</td>
</tr>
<tr>
<td>STR</td>
<td>Store register to memory</td>
<td>( \text{address} := R_d )</td>
</tr>
<tr>
<td><strong>Block Data Transfer</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDM</td>
<td>Load multiple registers</td>
<td>Stack Manipulation (Pop)</td>
</tr>
<tr>
<td>STM</td>
<td>Store Multiple</td>
<td>Stack Manipulation (Push)</td>
</tr>
<tr>
<td><strong>Single Data Swap</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SWP</td>
<td>Swap register with memory</td>
<td>( R_d := [R_n], [R_n] := R_m )</td>
</tr>
<tr>
<td><strong>Software Interrupt</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SWI</td>
<td>Software Interrupt</td>
<td>OS Call</td>
</tr>
</tbody>
</table>

Table 2.1: ARM Instruction Set relevant for the purposes of our work
### Suffix Flags

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Flags</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQ</td>
<td>Z set</td>
<td>equal</td>
</tr>
<tr>
<td>NE</td>
<td>Z clear</td>
<td>not equal</td>
</tr>
<tr>
<td>CS</td>
<td>C set</td>
<td>unsigned higher or same</td>
</tr>
<tr>
<td>CC</td>
<td>C clear</td>
<td>unsigned lower</td>
</tr>
<tr>
<td>MI</td>
<td>N set</td>
<td>negative</td>
</tr>
<tr>
<td>PL</td>
<td>N clear</td>
<td>positive or zero</td>
</tr>
<tr>
<td>VS</td>
<td>V set</td>
<td>overflow</td>
</tr>
<tr>
<td>VC</td>
<td>V clear</td>
<td>no overflow</td>
</tr>
<tr>
<td>HI</td>
<td>C set and Z clear</td>
<td>unsigned higher</td>
</tr>
<tr>
<td>LS</td>
<td>C clear or Z set</td>
<td>unsigned lower or same</td>
</tr>
<tr>
<td>GE</td>
<td>N equals V</td>
<td>greater or equal</td>
</tr>
<tr>
<td>LT</td>
<td>N not equal to V</td>
<td>less than</td>
</tr>
<tr>
<td>GT</td>
<td>Z clear AND (N equals V)</td>
<td>greater than</td>
</tr>
<tr>
<td>LE</td>
<td>Z set OR (N not equal to V)</td>
<td>less than or equal</td>
</tr>
<tr>
<td>AL</td>
<td>(ignored)</td>
<td>always</td>
</tr>
</tbody>
</table>

**Table 2.2:** Condition Codes
The instruction is fetched from memory

The registers used in the instruction are decoded

The registers are read from the register bank
The shift and ALU operations are performed
The registers are written back to the register bank

Figure 2.3: The instruction pipeline

the \texttt{CPSR} register is to be affected accordingly. For the sake of clarity, let us consider the following example:

\[ 2^{31} - 1 + 1 = -2^{32} \]

This operation provokes an overflow and produces a negative result, thus the \texttt{CPSR} register would be affected as follows: \texttt{NZCV} = 1001.

The reader may wonder the significance of the presence of such suffix, \texttt{S}, in the instructions \texttt{CMP}, \texttt{CMN}, \texttt{TEQ} and \texttt{TST}. These instructions already affect the \texttt{CPSR} register, thus the presence or absence of the \texttt{S} suffix is irrelevant, since it is implied anyway.

2.3 Pipeline

The ARM7TDMI-S processor features a 3-stage pipeline, allowing the overlapping of the instruction’s execution, and thus better performance. As shown in Figure 2.3, the three stages of the pipeline are \texttt{Fetch}, \texttt{Decode} and \texttt{Execute}. The hardware of each stage is designed so that each stage is independent, and thus, up to three instructions can be processed simultaneously.

The basic idea of instruction pipelining is to split the processing of an instruction into a series of independent steps, and thus, different steps from different instructions can be
overlapped in the pipeline. For the sake of clarity, let us consider the following example, where time runs horizontally, with each tick of time corresponding to a processor cycle, and the pipeline stages are represented on the vertical axis. An instruction A takes 3 cycles to execute, and an instruction B takes 4 cycles:

<table>
<thead>
<tr>
<th>Fetch</th>
<th>A</th>
<th></th>
<th>Fetch</th>
<th>B</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decode</td>
<td>A</td>
<td></td>
<td>Decode</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>Execute</td>
<td>A</td>
<td></td>
<td>Execute</td>
<td>B</td>
<td></td>
</tr>
</tbody>
</table>

Without instruction pipelining, the execution of AB would take 7 cycles, however, by overlapping A and B we obtain a 5 cycles execution:

<table>
<thead>
<tr>
<th>Fetch</th>
<th>A</th>
<th>B</th>
<th></th>
<th>Fetch</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decode</td>
<td>A</td>
<td>B</td>
<td>B</td>
<td></td>
<td>Execute</td>
<td>A</td>
</tr>
</tbody>
</table>

Early approaches to timing analysis assumed context independence, ignoring the execution history [3]. However, as seen in this trivial example, this assumption is no longer true for modern processors featuring pipelines.

Further, the performance of the pipeline can be measured by the Clock Cycles per Instruction (CPI). In the optimal case, as soon as the pipeline has been completely filled, the execution of an instruction takes one cycle. In such cases, we say that the CPI is equal to 1.

### 2.3.1 Instruction Cycle Count Summary

In the ARM7TDMI-S pipeline while one instruction is being fetched, the previous is being decoded, and the one prior to that is being executed. Table 2.3 presents the number of cycles required by an instruction, when the Execute stage is reached.

In table 2.3,

- n is the number of words transferred

- m is:
  - 1 if bits [32:8] of the multiplier operand are all zero or one.
  - 2 if bits [32:16] of the multiplier operand are all zero or one.
### Table 2.3: Instruction cycle counts

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Qualifier</th>
<th>Cycle count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any unexecuted</td>
<td>Condition codes fail</td>
<td>1</td>
</tr>
<tr>
<td>Data processing</td>
<td>Single-cycle</td>
<td>1</td>
</tr>
<tr>
<td>Data processing</td>
<td>Register-specified shift</td>
<td>2</td>
</tr>
<tr>
<td>Data processing</td>
<td>R15 destination</td>
<td>3</td>
</tr>
<tr>
<td>Data processing</td>
<td>R15, register-specified shift</td>
<td>4</td>
</tr>
<tr>
<td>MUL</td>
<td>m + 1</td>
<td></td>
</tr>
<tr>
<td>MLA</td>
<td>m + 2</td>
<td></td>
</tr>
<tr>
<td>MULL</td>
<td>m + 2</td>
<td></td>
</tr>
<tr>
<td>MLAL</td>
<td>m + 3</td>
<td></td>
</tr>
<tr>
<td>Branches</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>LDR</td>
<td>Non-R15 destination</td>
<td>3</td>
</tr>
<tr>
<td>LDR</td>
<td>R15 destination</td>
<td>5</td>
</tr>
<tr>
<td>STR</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>SWP</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>LDM</td>
<td>Non-R15 destination</td>
<td>(n-1) + 3</td>
</tr>
<tr>
<td>LDM</td>
<td>R15 destination</td>
<td>(n-1) + 5</td>
</tr>
<tr>
<td>STM</td>
<td></td>
<td>(n-1) + 3</td>
</tr>
<tr>
<td>MSR, MRS</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>SWI</td>
<td></td>
<td>3</td>
</tr>
</tbody>
</table>
2.3. PIPELINE

- 3 if bits [31:24] of the multiplier operand are all zero or one.
- 4 otherwise.

2.3.2 Pipeline Stalls

Instruction pipelining is a powerful method to enhance performance. However, some situations give rise to pipeline stalls which provokes the execution to take extra cycles to complete. Consider the two following instructions A and B:

<table>
<thead>
<tr>
<th>Fetch</th>
<th>A</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>Fetch</th>
<th>B</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Decode</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Decode</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execute</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td></td>
<td>Execute</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The execution of A takes 6 cycles, which will make the execution of B in the pipeline stall:

<table>
<thead>
<tr>
<th>Fetch</th>
<th>A</th>
<th>B</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>Fetch</th>
<th>A</th>
<th>B</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Decode</td>
<td>A</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Decode</td>
<td>A</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execute</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td>Execute</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
</tbody>
</table>

Circumstances leading to pipeline stalls are denominated pipeline hazards. There are thee types of such hazards. (i) Structural hazards arise from resource conflicts, (ii) data hazards are caused by data dependences between overlapping instructions, and (iii) control hazards are due to control flow changes, i.e., branches.

In this chapter we presented the essential features of the ARM7TDMI-S processor. The ARM instruction set, the accessible registers in System/User mode and the instruction pipeline were the topics of main focus. For further details regarding the ARM architecture we refer the reader to the information discussed in [10].

In the next chapter we present the basic notions concerning the timing analysis of Real-Time Systems [RTS].
Chapter 3

Real-Time Systems and Timing Analysis

In this chapter we present the main ingredients of Real-Time Systems with emphasis on their timing analysis. The main challenges, methods and tools for timing analysis are presented and discussed.

3.1 Real-Time Systems

Real-Time Systems (RTS) are required to satisfy stringent timing properties. These are often defined as output computations responses to some input stimuli, that need to be completed under some timing interval. In general, in order to guarantee that the imposed deadlines will be met, an upper bound on each task's execution time is necessary. However, determining the execution time on any arbitrary program is *undecidable*, otherwise one could solve the *halting problem* \[18\]. Hence, a restricted form of programming is typically employed, recursion and loops bounds are explicitly bounded, and thus ensuring termination.

The tasks that compose RTS characteristically present variations on their execution times depending on many aspects. Hardware features, like caches and pipelines, and input data are some of the elements that make the execution time context dependent. To the longest time of all possible executions we call it \([\text{WCET}]\) and analogously \([\text{BCET}]\) to the shortest.

*Timing Analysis* is the process of deriving execution time bounds or estimates. In Figure \[3.1\] its basic concepts are presented. The set of execution times is upper- and lower-bounded by \([\text{WCET}]\) and \([\text{BCET}]\) respectively. In general, due to the complexity of modern processors, and the large program's state space, the actuals \([\text{WCET}]\) and \([\text{BCET}]\)
Figure 3.1: Basic concepts of Timing Analysis

cannot be determined. These characteristics of the overall system, affecting the timing analysis, form the notion of *timing predictability*.

The two main criteria to assess a particular method for timing analysis are *safety* and *precision*. The former is directly related with whether the method produces bounds or estimates, i.e., if it produces an upper-bound for the WCET and thus is safe, or if the predicted value can in fact be lower than the actual WCET (and vice-versa for BCET), and thus unsafe. The latter, is concerned about how far is the prediction of the actual WCET/BCET. In general, only static analysis-based methods provide guarantees w.r.t. the safety criteria. Measurement-based methods cannot ensure that the measured executions capture the WCET and BCET. These usually underestimate the WCET and overestimate the BCET and thus are unsafe.

### 3.1.1 Timing Anomalies

Most powerful microprocessors suffer from *timing anomalies* [19]. These are contra-intuitive influences of the (local) execution time of one instruction on the (global) execution of the whole task [3]. For instance, tacitly assuming that a cache-miss penalty yields a longer execution time than a cache hit is wrong for processor with timing anomalies. In Figure 3.2 a *speculation-caused anomaly* is depicted.

While intuition would suggest that a cache hit would lead to a shorter execution time,
in [20] it has been shown that this need not be true. This was observed on the Motorola ColdFire 5307 processor. The cause of this anomaly arises from the speculation on the outcome of conditional branches, i.e., it prefetches instructions in one of the directions of a conditional branch. However, when the condition is evaluated, it might be the case that the processor speculated in the wrong direction. All the effects that the speculation gave rise, have to be undone. Further, the contents of the cache have been damaged, since the wrong instructions have been fetched. Hence, the burden of a branch misprediction surpass the costs of a cache miss.

### 3.2 Methods and Tools

A lot of research has been dedicated to the safe and precise derivation of the WCET [3]. Indeed, the increasing amount of RTS being used, and their inherent challenges make it a very interesting research topic. There are essentially two classes of methods employed in the timing analysis field: static analysis-based methods and measurement-based methods. The former class of methods, does not rely on executing code, but rather receive the code itself as input, with or without some extra informations (annotations, user domain-specific knowledge, etc), perform a control-flow analysis, and derive bounds by combining the control flow with some abstract model of the considered hardware. While the latter executes the code, or parts of it, in a given hardware or simulator for some set of

\footnote{Also denominated preemptive execution by Intel, instruction prefetch is a technique to speed up the execution of programs. It occurs when a processor requests an instruction from the main memory before it is actually needed.}
3.2.1 Measurement-based Methods

The current trend in this field is focused on static analysis-based methods. However, there are several tools, commercial and research prototypes, based on measurements. RapiTime \[21\] tries to derive the WCET by measuring how long sections of code take from predefined inputs. It is also known as the commercial quality version of pWCET \[22\]. SymTA/P \[23\] targets C programs running on microcontrollers. It uses an hybrid approach, doing path analysis on source code level combined with measurements on object code level. From Sweden, another two research prototypes using an hybrid approach are known: the Chalmers prototype \[24\] and SWEET \[25\]. Both use simulation as the underlying methodology. The former works directly on Power-PC binaries, while the latter integrates a research C compiler in order to perform flow analysis in the intermediate code.

The benefit of measurement-based methods is that they are easier to apply to new target processors, since modelling the processor’s behaviour is not necessary. It is also claimed that the produced estimates for the WCET and BCET are more precise than the bounds produced by static analysis-based methods. However, the exacts BCET and WCET cannot be determined in general. Studies regarding precision of estimates or bounds are habitually comparisons with the lowest/greatest observed time from a large, but not exhaustive \[3\], set of tests \[3\]. On the other hand, the main difficulty of measurement-based methods is to measure execution times accurately. Obtaining a fine granularity, without affecting the program’s behaviour may require a processor-specific solution. A general approach, is to place the code fragment to be measured in a loop with a predetermined bound, measure its execution, and then divide by the number of iterations.

3.2.2 Static Analysis-based Methods

Static analysis-based methods are essentially constituted by the components depicted in Figure 3.3\[4\].

Further, they can be classified according to the level they operate. All require the pres-

---

\[3\] Indeed, this can be seen as the classic Testing Vs Verification debate.

\[4\] This Figure is taken from \[3\].
3.2. METHODS AND TOOLS

![Figure 3.3: Core components of a static timing-analysis tool](image)

ence of a binary file, since only at that level one can obtain all the necessary information for timing analysis, however some also require the source code. *Heptane* [26], *Chronos* [27] and *TuBound* [28] require both the source and object codes. From Vienna University of Technology, there are three research prototypes [29]: *Vienna S*, *Vienna M* and *Vienna H*. While the first one relies mostly on static program analysis, the remaining employ genetic algorithms and model checking, respectively. Only *Vienna M* work solely at the object level, the remaining two also require the presence of the source code. Finally, *aiT* [7] is the state-of-the-art static timing analyser, designed

"according to the requirements of Airbus France for validating the timing behavior of critical avionics software, including the flight control software of the A380, the world’s largest passenger aircraft".

It operates at the object level, the presence of the source code is not necessary. However, it may also need some user input in order to obtain a result, or to improve the precision of the predicted bound.

The inherent advantage of static analysis-based methods is that running the program is not necessary, which may be particularly relevant for programs that require complex and expensive equipments. Moreover, rather than estimates, these methods produce bounds and thus of crucial importance for hard RTS. On the other hand, its main challenge lies in the modelling of the processor behaviour.

---

5While the strict definition of an hard RTS is that missing the deadline would constitute failure of the system, it is referred in the literature as RTS whose lack of success in meeting a deadline could endanger human lives.
3.3 Related Work

There are numerous approaches, found in the literature, focused in algorithms and tools for the derivation of the WCET \cite{4,30,7,31}. The goal of this work is not to present yet another approach of that type, but rather to emphasize the production of a certificate that can be then used to validate the predicted WCET and BCET.

Nevertheless, we should refer that the problem of certifying resource consumption, namely execution time, as already been addressed before, like in the work of Crary & Weirich \cite{32}, that use an extended type system capable of specifying and certifying bounds on resource consumption. However, this work makes no effort to determine bounds on execution times, but rather provides a mechanism to certify those bounds (for instance, obtained via a previous program analysis). The result of their approach is an executable object that is certified w.r.t. resource consumption. Furthermore, Bonenfant et al and Jost et al \cite{33,34} present an interesting combination of information retrieved at source code level, with low-level timing information gathered with AbsInt’s aiT tool \cite{7}. Both works provide guaranteed bounds on worst-case execution times for a strict, higher-order programming language.

The Mobility, Ubiquity and Security (MOBIUS) \cite{35}, and the Mobile Resource Guarantees (MRG) \cite{36} research projects also aim at the certification of resource consumption, their approaches mostly rely on theorem proving, whereas ours relies on abstract interpretation.

Moreover, in \cite{37}, Kwon et al, present a safe mobile code representation for Real-Time Java programs. Based on a Static Single Assignment (SSA) representation, they argue that their approach is able to cope with the intrinsic challenges of high-integrity real-time software (such as subsystems upgrades and software portability). Unlike prior approaches to theirs \cite{38}, they focus more on temporal predictability and safety of Java programs.

In this chapter we presented the basic notions concerning the timing analysis of RTS. Its challenging aspects, methods and tools were discussed.

In the next chapter we present the core of the work developed in the context of this thesis.
Chapter 4

From Predictions to Certificates

In this Chapter we present the work developed in the context of this thesis, where the mathematical formalisms will be introduced as needed.

In Section 4.1 the process of reconstructing the control-flow graph from an ARM assembly file is shown. From the resulting control-flow graph, in Section 4.2 we present our efforts towards a pipeline analysis, that is supported by the sound theory of Abstract Interpretation. Section 4.3 demonstrates how to compute bounds on execution times. Sections 4.4 and 4.5 discuss the production of the certificate, and how it can be used for validation purposes, respectively.

4.1 Control-Flow Graph Analysis

A Control-Flow Graph (CFG) is standard representation in program analysis, using a graph notation, it expresses all the paths that a program can have during its execution. Usually, the actual control flow of a program is not fully known due to complex control transfers such as computed branches. Hence, when reconstructing a CFG, it is common to start by a safe (over-)approximation, and then try to refine it by means of static analysis.

The control flow of a program can be described completely by its Interprocedural Control-Flow Graph (ICFG) [39]. It is composed by two elements:

- A Call Graph (CG) describing the relationships between the program’s procedures\(^1\)

\(^1\)Also addressed as procedures or functions in the literature.
The nodes stand for procedures, while the edges entail procedure calls.

- A Basic Block Graph (BBG) describing the intraprocedural control flow for each of the program's procedures.

The more precise the ICFG is, the better the timing analysis can perform. However, more importantly, in the context of real-time systems it is imperative that the ICFG is safe under all circumstances. Removing an edge without certainty that it represents a path that will never be taken by the program, could seriously compromise the timing analysis. For instance, if the removed edge was in fact, part of the path that led to longest time of execution, then our WCET prediction would be an under-approximation, and thus unsafe.

### 4.1.1 Interprocedural Control-Flow Graph Reconstruction

The process of reconstructing the ICFG is separated in two phases. First the CG is built by gathering the called procedures, and then we proceed by constructing the BBG of each of them. In ARM, procedure calls are made by means of the branch with link instruction, BL (see Table 2.1). Hence, gathering the called procedures is a matter of collecting the targets of these branches, and perform analogously in the targeted procedures.

Having the CG, it is time to build the BBG of each of the called procedures. There are two possible approaches for reconstructing the control flow from a sequential list of assembly instructions: a top-down approach, or a bottom-up approach. Our methodology follows a top-down approach, since it is the most natural, and allows to cope more efficiently with control flow branches. For instance, consider the following example:

```
B label+7
```

A branch target can be a static label or it can also be an arithmetic operation, and thus coping with these would only require to evaluate the target expression in order to determine the node's successor. On the other hand, in a bottom-up approach, one wants to determine the node's predecessors, and thus a prior analysis would be required in order to be able to know which are the nodes predecessors (or add an edge to the successor whenever encountering a branch).
At this stage we are only concerned about obtaining a safe (over-)approximation of the CFG. Hence, whenever in doubt regarding a node's successors, a conservative approach will be taken. For instance, let us consider the C program represented in listing 4.1.

Listing 4.1: Toy Example

```c
1 int main() {
2
3 int x;
4
5 x = 5;
6
7 if (x > 6)
8     return 1;
9
10 return 0;
11 }
```

After compiled with GNU compiler for ARM [40], the assembly code presented in listing 4.2 is generated.

Listing 4.2: Compiled ARM Assembly

```assembly
1 mov ip, sp
2 stmfd sp!, {fp, ip, lr, pc}
3 sub fp, ip, #4
4 sub sp, sp, #8
5 mov r3, #5
6 str r3, [fp, #-16]
7 ldr r3, [fp, #-16]
8 cmp r3, #6
9 ble .L2
10 mov r3, #1
11 str r3, [fp, #-20]
12 b .L1
13 .L2: mov r3, #0
14 str r3, [fp, #-20]
```
Proceeding to the BBG reconstruction, the obtained control flow is as illustrated by Figure 4.1. By inspecting the source code of our toy example (listing 4.1), it is clear that the conditional will always evaluate to false, and thus the instruction on the 8th line (return 1) will never be executed. Inspecting the assembly (listing 4.2) however, is not as straightforward. In the 5th line the value 5 is stored in register r3 and later compares it with the value 6 (8th line). This comparison instruction performs a subtraction as indicated in Table 2.1.

\[ R3 - 6 = 5 - 6 = -1, \]

and affects the CPSR flags accordingly:

\[ NZCV = 1010. \]
4.1. CONTROL-FLOW GRAPH ANALYSIS

The careful reader may wonder why the C flag (Carry) is set in this operation. This is due to the fact that in subtraction operations, clearing the Carry flag (C=0) means the presence of a borrow\(^2\), thus setting the Carry flag means its absence \[41\], which is the case in this operation.

In the 9\(^{th}\) line a conditional branch takes place. The condition to be evaluated, *Less or Equal* (LE), will always evaluate to *true*, since, according to Table 2.2, the LE condition code is evaluated as follows:

\[
Z \text{ or } (N \text{ not equal to } V) = \\
\text{false or (true not equal to false)} = \\
\text{false or true} = \\
\text{true},
\]

thus, the program will always branch to the address evaluated by the .L2 label, where the value 0 is stored in register R3, and later stored in register R0 (15\(^{th}\) line).

As mentioned earlier, at this stage the main concern is to produce a safe approximation of the program’s control-flow. Having this, an analysis can be performed in order to try to determine the outcome of such conditional branches, and, if possible, safely remove the edges that are never taken during program’s execution.

### 4.1.2 Data-Flow Analysis for Conditional Branches

Data-Flow Analysis (DFA) is a well established technique in the area of compiler construction and used to obtain information about a program that holds for all executions of the program. Typically, it is achieved by setting up data-flow equations and computing a fixpoint over these equations. For instance, *dead-code elimination* or *code motion* are examples of applications \[42\]. The attractiveness of DFA is that it computes safety properties for each instruction of the program, that behave like invariants, i.e., they are valid for all executions.

The major problem in the reconstruction of the BBG are indirections of the control flow, i.e., branches whose argument is a register value rather than a assembly label \[39\]. In our case, the challenge is to statically determine the outcome of the branches’ conditionals, that directly depends on the contents of the CPSR register. However, it should be

\(^2\)In arithmetic, a borrow is the opposite of a carry.
noted that it might not be always possible to solve these conditionals, since, for instance, there might be a dependence with user interaction.

Static analysis allows to determine dynamic properties of programs, statically. In the following, an algorithm tailored for ARM programs that tries to cope with the conditional branches will be presented.

Let $Prog$ denote the set of all ARM programs. Further, $PC$ is a set ranging over the natural numbers\(^3\) and $p \in Prog$ is the program of interest, the one being analysed. The function

$$flow : Prog \rightarrow \mathcal{P}(PC_\ast \times PC_\ast),$$

maps programs to sets of flows, where $PC_\ast$ is defined as a finite subset of $PC$, holding the program counter values of $p$. It should be clear that $\mathcal{P}(D)$ stands for the powerset of $D$. Formally, it is defined as

$$\mathcal{P}(D) = \{D' \mid D' \subseteq D\},$$

i.e., the set of all subsets of $D$, including the empty set and $D$ itself. Hence, for the ARM program shown in listing 4.2 we obtain the following:

$$flow(p) = \{(1, 2), (2, 3), (3, 4), (4, 5), (5, 6), (6, 7), (7, 8), (8, 9), (9, 10), (10, 11), (11, 12), (9, 13), (13, 14), (12, 15), (14, 15), (15, 16)\}$$

Another important characteristic of a program is the stores that it manipulates. Let us define

$$Store = Register_\ast \cup Mem_\ast,$$

as the set holding all the stores of a program, where $Register_\ast \subseteq Register$ is a subset of the ARM registers, and analogously for $Mem_\ast \subseteq Mem$ w.r.t. to the memory areas. Hence, for our running example (see listing 4.2), we got:

$$Store = \{IP, SP, FP, LR, PC, R3, R0\} \cup \{[FP, # - 16], [FP, # - 20]\}$$

\(^3\)In the beginning of the nineteenth century a definition of the natural numbers containing the element 0 appeared. In this case, we mean the traditional definition, i.e., the positive integers.
4.1.2.1 Reaching Definition Analysis

The *Reaching Definition Analysis* \[^\text{RDA}\] is a common DFA. In [42] it is defined as:

"For each program point, which assignments may have been made and not overwritten, when program execution reaches this point along some path."

While it is clear what an *assignment* means for high-level programming languages, let us clarify this concept for the ARM programming language. In the following, an assignment will be considered whenever a store is written to. For instance, the instruction `CMP R3, #6` is an assignment to the CPSR register.

The *Reaching Definition Analysis* is specified by two functions. The first,

\[
\text{kill}_{rd} : \text{Instruction} \rightarrow \mathcal{P}(\text{Store} \times \text{PC}^\ast),
\]

produces the set of pairs of stores and program counters of assignments that are *destroyed* by the instruction. An assignment is destroyed if the instruction assigns a new value to the store. To cope with uninitialized stores, we use the special program counter "?", hence

\[
\text{PC}^* = \text{PC} \cup \{?\}.
\]

Further, *Instruction* is the set holding the ARM instructions. The definition of the \(\text{kill}_{rd}\) function is as represented in Table 4.1.

The second function specifying the *Reaching Definition Analysis*,

\[
\text{gen}_{rd} : \text{Instruction} \rightarrow \mathcal{P}(\text{Store} \times \text{PC}^\ast),
\]

produces the set of pairs of stores and program counters of assignments generated by the instruction. Its definition can be seen in Table 4.2. Furthermore, as mentioned in section 2.2 in ARM any instruction can be suffixed by S. Whenever this is the case, the CPSR register needs to be updated, i.e., it is also assigned. Thus,

\[
\text{gen}_{rd}(<Op>S) = \text{gen}_{rd}(Op) \cup \{(\text{CPSR}, pc) \mid pc \text{ is the instruction’s program counter}\}
\]

For our running example (see listing 4.2), the results of application of the functions \(\text{kill}_{rd}\) and \(\text{gen}_{rd}\) are as illustrated by Tables 4.3 and 4.4 respectively.
Table 4.1: $kill_{rd}$ function

<table>
<thead>
<tr>
<th>$kill_{rd}$</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$kill_{rd}(BX R_i)$</td>
<td>$((R15, ?)) \cup ((R15, pc) \mid pc \text{ is pointing to an assignment})$</td>
</tr>
<tr>
<td>$kill_{rd}(B\ exp)$</td>
<td>$((R15, ?)) \cup ((R15, pc) \mid pc \text{ is pointing to an assignment})$</td>
</tr>
<tr>
<td>$kill_{rd}(BL R_i)$</td>
<td>$((R14, ?), (R15, ?))$</td>
</tr>
<tr>
<td>$kill_{rd}(MOV R_i, Op2)$</td>
<td>$((R_i, ?)) \cup ((R_i, pc) \mid pc \text{ is pointing to an assignment})$</td>
</tr>
<tr>
<td>$kill_{rd}(MVN R_i, Op2)$</td>
<td>$((R_i, ?)) \cup ((R_i, pc) \mid pc \text{ is pointing to an assignment})$</td>
</tr>
<tr>
<td>$kill_{rd}(CMP R_i, Op2)$</td>
<td>$((CPSR, ?)) \cup ((CPSR, pc) \mid pc \text{ is pointing to an assignment})$</td>
</tr>
<tr>
<td>$kill_{rd}(CMN R_i, Op2)$</td>
<td>$((CPSR, ?)) \cup ((CPSR, pc) \mid pc \text{ is pointing to an assignment})$</td>
</tr>
<tr>
<td>$kill_{rd}(TEQ R_i, Op2)$</td>
<td>$((CPSR, ?)) \cup ((CPSR, pc) \mid pc \text{ is pointing to an assignment})$</td>
</tr>
<tr>
<td>$kill_{rd}(TST R_i, Op2)$</td>
<td>$((CPSR, ?)) \cup ((CPSR, pc) \mid pc \text{ is pointing to an assignment})$</td>
</tr>
<tr>
<td>$kill_{rd}(AND R_i, R_j, Op2)$</td>
<td>$((R_i, ?)) \cup ((R_i, pc) \mid pc \text{ is pointing to an assignment})$</td>
</tr>
<tr>
<td>$kill_{rd}(EOR R_i, R_j, Op2)$</td>
<td>$((R_i, ?)) \cup ((R_i, pc) \mid pc \text{ is pointing to an assignment})$</td>
</tr>
<tr>
<td>$kill_{rd}(SUB R_i, R_j, Op2)$</td>
<td>$((R_i, ?)) \cup ((R_i, pc) \mid pc \text{ is pointing to an assignment})$</td>
</tr>
<tr>
<td>$kill_{rd}(RSB R_i, R_j, Op2)$</td>
<td>$((R_i, ?)) \cup ((R_i, pc) \mid pc \text{ is pointing to an assignment})$</td>
</tr>
<tr>
<td>$kill_{rd}(ADD R_i, R_j, Op2)$</td>
<td>$((R_i, ?)) \cup ((R_i, pc) \mid pc \text{ is pointing to an assignment})$</td>
</tr>
<tr>
<td>$kill_{rd}(ADC R_i, R_j, Op2)$</td>
<td>$((R_i, ?)) \cup ((R_i, pc) \mid pc \text{ is pointing to an assignment})$</td>
</tr>
<tr>
<td>$kill_{rd}(SBC R_i, R_j, Op2)$</td>
<td>$((R_i, ?)) \cup ((R_i, pc) \mid pc \text{ is pointing to an assignment})$</td>
</tr>
<tr>
<td>$kill_{rd}(RSC R_i, R_j, Op2)$</td>
<td>$((R_i, ?)) \cup ((R_i, pc) \mid pc \text{ is pointing to an assignment})$</td>
</tr>
<tr>
<td>$kill_{rd}(ORR R_i, R_j, Op2)$</td>
<td>$((R_i, ?)) \cup ((R_i, pc) \mid pc \text{ is pointing to an assignment})$</td>
</tr>
<tr>
<td>$kill_{rd}(BIC R_i, R_j, Op2)$</td>
<td>$((R_i, ?)) \cup ((R_i, pc) \mid pc \text{ is pointing to an assignment})$</td>
</tr>
<tr>
<td>$kill_{rd}(MRS R_i, CPSR)$</td>
<td>$((R_i, ?)) \cup ((R_i, pc) \mid pc \text{ is pointing to an assignment})$</td>
</tr>
<tr>
<td>$kill_{rd}(MSR R_i, CPSR)$</td>
<td>$((CPSR, ?)) \cup ((CPSR, pc) \mid pc \text{ is pointing to an assignment})$</td>
</tr>
<tr>
<td>$kill_{rd}(MUL R_i, R_j, R_k)$</td>
<td>$((R_i, ?)) \cup ((R_i, pc) \mid pc \text{ is pointing to an assignment})$</td>
</tr>
<tr>
<td>$kill_{rd}(MLA R_i, R_j, R_k, R_l)$</td>
<td>$((R_i, ?)) \cup ((R_i, pc) \mid pc \text{ is pointing to an assignment})$</td>
</tr>
<tr>
<td>$kill_{rd}(LDR R_i, Addr)$</td>
<td>$((R_i, ?)) \cup ((R_i, pc) \mid pc \text{ is pointing to an assignment})$</td>
</tr>
<tr>
<td>$kill_{rd}(STR R_i, Addr)$</td>
<td>$((Addr, ?)) \cup ((Addr, pc) \mid pc \text{ is pointing to an assignment})$</td>
</tr>
<tr>
<td>$kill_{rd}(LDM R_i, R_list)$</td>
<td>$\cup_{r \in R_list} { (r, ?) } \cup { (r, pc) \mid pc \text{ is pointing to an assignment} }$</td>
</tr>
<tr>
<td>$kill_{rd}(SWP R_i, R_j, R_k)$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$kill_{rd}(SWI R_i, exp)$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>
### 4.1. CONTROL-FLOW GRAPH ANALYSIS

<table>
<thead>
<tr>
<th>Function</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{gen}_{rd}(\text{BX} R_i) )</td>
<td>( ({ (R15, \text{pc}) \mid \text{pc is the instruction's program counter } } )</td>
</tr>
<tr>
<td>( \text{gen}_{rd}(\text{B exp}) )</td>
<td>( ({ (R15, \text{pc}) \mid \text{pc is the instruction's program counter } } )</td>
</tr>
<tr>
<td>( \text{gen}_{rd}(\text{BL} R_i) )</td>
<td>( ({ (R14, \text{pc}), (R15, \text{pc}) \mid \text{pc is the instruction's program counter } } )</td>
</tr>
<tr>
<td>( \text{gen}_{rd}(\text{MOV} R_i, \text{Op2}) )</td>
<td>( ({ (R_i, \text{pc}) \mid \text{pc is the instruction's program counter } } )</td>
</tr>
<tr>
<td>( \text{gen}_{rd}(\text{MVN} R_i, \text{Op2}) )</td>
<td>( ({ (R_i, \text{pc}) \mid \text{pc is the instruction's program counter } } )</td>
</tr>
<tr>
<td>( \text{gen}_{rd}(\text{CMP} R_i, \text{Op2}) )</td>
<td>( ({ (\text{CPSR}, \text{pc}) \mid \text{pc is the instruction's program counter } } )</td>
</tr>
<tr>
<td>( \text{gen}_{rd}(\text{CMN} R_i, \text{Op2}) )</td>
<td>( ({ (\text{CPSR}, \text{pc}) \mid \text{pc is the instruction's program counter } } )</td>
</tr>
<tr>
<td>( \text{gen}_{rd}(\text{TEQ} R_i, \text{Op2}) )</td>
<td>( ({ (\text{CPSR}, \text{pc}) \mid \text{pc is the instruction's program counter } } )</td>
</tr>
<tr>
<td>( \text{gen}_{rd}(\text{TST} R_i, \text{Op2}) )</td>
<td>( ({ (\text{CPSR}, \text{pc}) \mid \text{pc is the instruction's program counter } } )</td>
</tr>
<tr>
<td>( \text{gen}_{rd}(\text{AND} R_i, R_j, \text{Op2}) )</td>
<td>( ({ (R_i, \text{pc}) \mid \text{pc is the instruction's program counter } } )</td>
</tr>
<tr>
<td>( \text{gen}_{rd}(\text{EOR} R_i, R_j, \text{Op2}) )</td>
<td>( ({ (R_i, \text{pc}) \mid \text{pc is the instruction's program counter } } )</td>
</tr>
<tr>
<td>( \text{gen}_{rd}(\text{SUB} R_i, R_j, \text{Op2}) )</td>
<td>( ({ (R_i, \text{pc}) \mid \text{pc is the instruction's program counter } } )</td>
</tr>
<tr>
<td>( \text{gen}_{rd}(\text{RSB} R_i, R_j, \text{Op2}) )</td>
<td>( ({ (R_i, \text{pc}) \mid \text{pc is the instruction's program counter } } )</td>
</tr>
<tr>
<td>( \text{gen}_{rd}(\text{ADD} R_i, R_j, \text{Op2}) )</td>
<td>( ({ (R_i, \text{pc}) \mid \text{pc is the instruction's program counter } } )</td>
</tr>
<tr>
<td>( \text{gen}_{rd}(\text{ADC} R_i, R_j, \text{Op2}) )</td>
<td>( ({ (R_i, \text{pc}) \mid \text{pc is the instruction's program counter } } )</td>
</tr>
<tr>
<td>( \text{gen}_{rd}(\text{SBC} R_i, R_j, \text{Op2}) )</td>
<td>( ({ (R_i, \text{pc}) \mid \text{pc is the instruction's program counter } } )</td>
</tr>
<tr>
<td>( \text{gen}_{rd}(\text{RSC} R_i, R_j, \text{Op2}) )</td>
<td>( ({ (R_i, \text{pc}) \mid \text{pc is the instruction's program counter } } )</td>
</tr>
<tr>
<td>( \text{gen}_{rd}(\text{ORR} R_i, R_j, \text{Op2}) )</td>
<td>( ({ (R_i, \text{pc}) \mid \text{pc is the instruction's program counter } } )</td>
</tr>
<tr>
<td>( \text{gen}_{rd}(\text{BIC} R_i, R_j, \text{Op2}) )</td>
<td>( ({ (R_i, \text{pc}) \mid \text{pc is the instruction's program counter } } )</td>
</tr>
<tr>
<td>( \text{gen}_{rd}(\text{MRS} R_i, \text{CPSR}) )</td>
<td>( ({ (R_i, \text{pc}) \mid \text{pc is the instruction's program counter } } )</td>
</tr>
<tr>
<td>( \text{gen}_{rd}(\text{MSR} R_i, \text{CPSR}) )</td>
<td>( ({ (\text{CPSR}, \text{pc}) \mid \text{pc is the instruction's program counter } } )</td>
</tr>
<tr>
<td>( \text{gen}_{rd}(\text{MUL} R_i, R_j, R_k) )</td>
<td>( ({ (R_i, \text{pc}) \mid \text{pc is the instruction's program counter } } )</td>
</tr>
<tr>
<td>( \text{gen}_{rd}(\text{MLA} R_i, R_j, R_k, R_l) )</td>
<td>( ({ (R_i, \text{pc}) \mid \text{pc is the instruction's program counter } } )</td>
</tr>
<tr>
<td>( \text{gen}_{rd}(\text{LDR} R_i, \text{Addr}) )</td>
<td>( ({ (R_i, \text{pc}) \mid \text{pc is the instruction's program counter } } )</td>
</tr>
<tr>
<td>( \text{gen}_{rd}(\text{STR} R_i, \text{Addr}) )</td>
<td>( ({ (\text{Addr}, \text{pc}) \mid \text{pc is the instruction's program counter } } )</td>
</tr>
<tr>
<td>( \text{gen}<em>{rd}(\text{LDM} R_i, R</em>{List}) )</td>
<td>( \bigcup_{r \in R_{List}} ({ r, \text{pc} \mid \text{pc is the instruction's program counter } } )</td>
</tr>
<tr>
<td>( \text{gen}<em>{rd}(\text{STM} R_i, R</em>{List}) )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( \text{gen}_{rd}(\text{SWP} R_i, R_j, R_k) )</td>
<td>( ({ (R_i, \text{pc}), (R_k, \text{pc}) \mid \text{pc is the instruction's program counter } } )</td>
</tr>
</tbody>
</table>

| Table 4.2: \( \text{gen}_{rd} \) function |
CHAPTER 4. FROM PREDICTIONS TO CERTIFICATES

killed assignments of ARM program in listing 4.2

<table>
<thead>
<tr>
<th>Killrd (Operation)</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>killrd (MOV IP, SP)</td>
<td>{(IP,?), (IP,0), (IP,1), (IP,2)}</td>
</tr>
<tr>
<td>killrd (STMFD SP!, {FP, IP, LR, PC})</td>
<td>φ</td>
</tr>
<tr>
<td>killrd (SUB FP, IP, #4)</td>
<td>{([FP,?], [FP,1], [FP,2], [FP,15])}</td>
</tr>
<tr>
<td>killrd (SUB SP, SP, #8)</td>
<td>{([SP,?], [SP,0], [SP,3], [SP,3], [SP,15])}</td>
</tr>
<tr>
<td>killrd (MOV R3, #5)</td>
<td>{([R3,?], [R3,4], [R3,5], [R3,6], [R3,7], [R3,9], [R3,10], [R3,12], [R3,13])}</td>
</tr>
<tr>
<td>killrd (STR R3, [FP, #-16])</td>
<td>{([FP, #-16],?), ([FP, #-16],5), ([FP, #-16],6)}</td>
</tr>
<tr>
<td>killrd (LDR R3, [FP, #-16])</td>
<td>{([R3,?], [R3,4], [R3,5], [R3,6], [R3,7], [R3,9], [R3,10], [R3,12], [R3,13])}</td>
</tr>
<tr>
<td>killrd (CMP R3, #6)</td>
<td>{([CPSR,?], [CPSR,7], [CPSR,8])}</td>
</tr>
<tr>
<td>killrd (BLE .L2)</td>
<td>{([R15,?], [R15,8], [R15,11])}</td>
</tr>
<tr>
<td>killrd (MOV R3, #1)</td>
<td>{([R3,?], [R3,4], [R3,5], [R3,6], [R3,7], [R3,9], [R3,10], [R3,12], [R3,13])}</td>
</tr>
<tr>
<td>killrd (STR R3, [FP, #-20])</td>
<td>{([FP, #-20],?), ([FP, #-20],10), ([FP, #-20],13), ([FP, #-20],14)}</td>
</tr>
<tr>
<td>killrd (B .L1)</td>
<td>{([R15,?], [R15,8], [R15,11])}</td>
</tr>
<tr>
<td>killrd (MOV R3, #0)</td>
<td>{([R3,?], [R3,4], [R3,5], [R3,6], [R3,7], [R3,9], [R3,10], [R3,12], [R3,13])}</td>
</tr>
<tr>
<td>killrd (STR R3, [FP, #-20])</td>
<td>{([FP, #-20],?), ([FP, #-20],10), ([FP, #-20],13), ([FP, #-20],14)}</td>
</tr>
<tr>
<td>killrd (LDR R0, [FP, #-20])</td>
<td>{([R0,?], [R0,14])}</td>
</tr>
<tr>
<td>killrd (LDMEA FP, {FP, IP, PC})</td>
<td>{([PC,?])(PC,1), (PC,15), (SP,?)(SP,0), (SP,3), (SP,15), ([FP,?], [FP,1], [FP,2], [FP,15])}</td>
</tr>
</tbody>
</table>

Table 4.3: killrd applied to the ARM program in listing 4.2
Table 4.4: \( \text{gen}_{rd} \) applied to the ARM program in listing 4.2

| \( \text{gen}_{rd}(\text{MOV} \ IP, \ SP) \) | \( \{(IP, 0)\} \) |
| \( \text{gen}_{rd}(\text{STMFD} \ SP!, \ \{FP, IP, LR, PC\}) \) | \( \emptyset \) |
| \( \text{gen}_{rd}(\text{SUB} \ FP, \ IP, \ #4) \) | \( \{(FP, 2)\} \) |
| \( \text{gen}_{rd}(\text{SUB} \ SP, \ SP, \ #8) \) | \( \{(SP, 3)\} \) |
| \( \text{gen}_{rd}(\text{MOV} \ R3, \ #5) \) | \( \{(R3, 4)\} \) |
| \( \text{gen}_{rd}(\text{STR} \ R3, \ [FP, \ #-16]) \) | \( \{(FP, # - 16), 5\} \) |
| \( \text{gen}_{rd}(\text{LDR} \ R3, \ [FP, \ #-16]) \) | \( \{(R3, 6)\} \) |
| \( \text{gen}_{rd}(\text{CMP} \ R3, \ #6) \) | \( \{(CPSR, 7)\} \) |
| \( \text{gen}_{rd}(\text{BLE} \ .L2) \) | \( \{(R15, 8)\} \) |
| \( \text{gen}_{rd}(\text{MOV} \ R3, \ #1) \) | \( \{(R3, 9)\} \) |
| \( \text{gen}_{rd}(\text{STR} \ R3, \ [FP, \ #-20]) \) | \( \{(FP, # - 20), 10\} \) |
| \( \text{gen}_{rd}(\text{B} \ .L1) \) | \( \{(R15, 11)\} \) |
| \( \text{gen}_{rd}(\text{MOV} \ R3, \ #0) \) | \( \{(R3, 12)\} \) |
| \( \text{gen}_{rd}(\text{STR} \ R3, \ [FP, \ #-20]) \) | \( \{(FP, # - 20), 13\} \) |
| \( \text{gen}_{rd}(\text{LDR} \ R0, \ [FP, \ #-20]) \) | \( \{(R0, 14)\} \) |
| \( \text{gen}_{rd}(\text{LDMEA} \ FP, \ \{FP, IP, PC\}) \) | \( \{(PC, 15), (SP, 15), (FP, 15)\} \) |
Finally, the analysis is completed by setting the data-flow equations composed by the pair of functions

\[ RD_{\text{entry}}, RD_{\text{exit}} : PC_* \rightarrow \mathcal{P}(\text{Store} \times PC^2_*), \]

which are defined as in Table 4.5

\[
RD_{\text{entry}}(pc) = \begin{cases} 
\{(R_i,?) | R_i \in \text{stores}(p)\} & \text{if } pc = 1, \\
\bigcup RD_{\text{exit}}(pc') | (pc', pc) \in \text{flow}(p) & \text{otherwise},
\end{cases}
\]

where \(\text{stores} : \text{Prog} \rightarrow \mathcal{P}(\text{Store})\) returns the stores of the program being analysed.

\[
RD_{\text{exit}}(pc) = (RD_{\text{entry}}(pc) \setminus \text{kill}_{rd}(\text{Inst}^{pc})) \cup \text{gen}_{rd}(\text{Inst}^{pc}),
\]

where \(\text{Inst}^{pc}\) is the instruction pointed by the program counter \(pc\).

**Table 4.5**: data-flow equations

Applying \(RD_{\text{entry}}\) and \(RD_{\text{exit}}\) to our running example (see listing 4.2) gives rise to the following data-flow equations:

\[
\begin{align*}
RD_{\text{entry}}(1) &= \{(FP,?), (PC,?), (R0,?), (R15,?), (CPSR,?), ([FP, # - 16],?) , \\
&\quad (LR,?), (IP,?), (R3,?), ([FP, # - 20],?), (SP,?)\}, \\
RD_{\text{entry}}(2) &= RD_{\text{exit}}(1) \\
&\quad \vdots \\
RD_{\text{entry}}(13) &= RD_{\text{exit}}(9) \\
&\quad \vdots \\
RD_{\text{entry}}(15) &= RD_{\text{exit}}(14) \cup RD_{\text{exit}}(12) \\
&\quad \vdots \\
RD_{\text{exit}}(1) &= (RD_{\text{entry}}(1) \setminus \text{kill}_{rd}(\text{MOVIP, SP})) \cup \text{gen}_{rd}(\text{MOVIP, SP}) \\
&\quad \vdots
\end{align*}
\]

Computing the fixpoint using Kildall’s worklist algorithm [43], the solution shown in Table 4.6 is obtained.

\[4^*\]As argued in [42], this analysis would be more properly called reaching assignments
<table>
<thead>
<tr>
<th>pc</th>
<th>$RD_{entry}(pc)$</th>
<th>$RD_{exit}(pc)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(FP?, PC?, (R0,?), (R15,?), (CPSR,?), (FP# = 16,?), (LR?, IP?), (R3,?), (FP# = 20,?), (SP?,))</td>
<td>(FP?, PC?, (R0,?), (R15,?), (CPSR,?), (FP# = 16,?), (LR?, IP?), (R3,?), (FP# = 20,?), (SP?,))</td>
</tr>
<tr>
<td>2</td>
<td>(FP?, PC?, (R0,?), (R15,?), (CPSR,?), (FP# = 16,?), (LR?, R3,?), (FP# = 20,?), (SP?,), (IP3,))</td>
<td>(PC?, (R0,?), (R15,?), (CPSR,?), (FP# = 16,?), (LR?, R3,?), (FP# = 20,?), (SP?,), (IP1,))</td>
</tr>
<tr>
<td>3</td>
<td>(FP?, PC?, (R0,?), (R15,?), (CPSR,?), (FP# = 16,?), (LR?, R3,?), (FP# = 20,?), (SP?,), (IP1,))</td>
<td>(PC?, (R0,?), (R15,?), (CPSR,?), (FP# = 16,?), (LR?, R3,?), (FP# = 20,?), (SP?,), (IP3,))</td>
</tr>
<tr>
<td>4</td>
<td>(PC?, (R0,?), (R15,?), (CPSR,?), (FP# = 16,?), (LR?, R3,?), (FP# = 20,?), (SP?,), (IP1,))</td>
<td>(PC?, (R0,?), (R15,?), (CPSR,?), (FP# = 16,?), (LR?, R3,?), (FP# = 20,?), (SP?,), (IP3,))</td>
</tr>
<tr>
<td>5</td>
<td>(PC?, (R0,?), (R15,?), (CPSR,?), (FP# = 16,?), (LR?, R3,?), (FP# = 20,?), (IP1,), (FP3,))</td>
<td>(PC?, (R0,?), (R15,?), (CPSR,?), (FP# = 16,?), (LR?, R3,?), (FP# = 20,?), (IP1,), (FP3,))</td>
</tr>
<tr>
<td>6</td>
<td>(PC?, (R0,?), (R15,?), (CPSR,?), (FP# = 16,?), (IP?, (FP1, (FP3, (SP4, (R3,5))), (IP1,))</td>
<td>(PC?, (R0,?), (R15,?), (CPSR,?), (FP# = 16,?), (IP?, (FP1, (FP3, (SP4, (R3,5))), (IP1,)))</td>
</tr>
<tr>
<td>7</td>
<td>(PC?, (R0,?), (R15,?), (CPSR,?), (FP# = 16,?), (IP?, (FP1, (FP3, (SP4, (R3,5))), (IP1,)))</td>
<td>(PC?, (R0,?), (R15,?), (CPSR,?), (FP# = 16,?), (IP?, (FP1, (FP3, (SP4, (R3,5))), (IP1,)))</td>
</tr>
<tr>
<td>8</td>
<td>(PC?, (R0,?), (R15,?), (CPSR,?), (FP# = 16,?), (IP?, (FP1, (FP3, (SP4, (R3,7))), (IP1,)))</td>
<td>(PC?, (R0,?), (R15,?), (CPSR,?), (FP# = 16,?), (IP?, (FP1, (FP3, (SP4, (R3,7))), (IP1,)))</td>
</tr>
<tr>
<td>9</td>
<td>(PC?, (R0,?), (R15,?), (CPSR,?), (FP# = 16,?), (IP?, (FP1, (FP3, (SP4, (R3,7))), (IP1,)))</td>
<td>(PC?, (R0,?), (R15,?), (CPSR,?), (FP# = 16,?), (IP?, (FP1, (FP3, (SP4, (R3,7))), (IP1,)))</td>
</tr>
<tr>
<td>10</td>
<td>(PC?, (R0,?), (CPSR,?), (FP# = 20,?), (IP1, (FP3, (SP4, (R3,7))), (CPSR,8), (CPSR,8)))</td>
<td>(PC?, (R0,?), (CPSR,?), (FP# = 20,?), (IP1, (FP3, (SP4, (R3,7))), (CPSR,8), (CPSR,8)))</td>
</tr>
<tr>
<td>11</td>
<td>(PC?, (R0,?), (CPSR,?), (FP# = 20,?), (IP1, (FP3, (SP4, (R3,7))), (CPSR,8), (CPSR,8)))</td>
<td>(PC?, (R0,?), (CPSR,?), (FP# = 20,?), (IP1, (FP3, (SP4, (R3,7))), (CPSR,8), (CPSR,8)))</td>
</tr>
<tr>
<td>12</td>
<td>(PC?, (R0,?), (CPSR,8), (FP# = 20,?), (IP1, (FP3, (SP4, (R3,10))), (CPSR,8), (CPSR,8)))</td>
<td>(PC?, (R0,?), (CPSR,8), (FP# = 20,?), (IP1, (FP3, (SP4, (R3,10))), (CPSR,8), (CPSR,8)))</td>
</tr>
<tr>
<td>13</td>
<td>(PC?, (R0,?), (CPSR,8), (FP# = 20,?), (IP1, (FP3, (SP4, (R3,10))), (CPSR,8), (CPSR,8)))</td>
<td>(PC?, (R0,?), (CPSR,8), (FP# = 20,?), (IP1, (FP3, (SP4, (R3,10))), (CPSR,8), (CPSR,8)))</td>
</tr>
<tr>
<td>14</td>
<td>(PC?, (R0,?), (CPSR,8), (FP# = 20,?), (IP1, (FP3, (SP4, (R3,13))), (CPSR,8), (CPSR,8)))</td>
<td>(PC?, (R0,?), (CPSR,8), (FP# = 20,?), (IP1, (FP3, (SP4, (R3,13))), (CPSR,8), (CPSR,8)))</td>
</tr>
<tr>
<td>15</td>
<td>(FP# = 20,14), (R3,13), (R15,9), (PC?, (R0,?), (L1, (IP1, (FP3, (SP4, (FP# = 16,6), (CPSR,8), (CPSR,8), (CPSR,8), (CPSR,8), (R3,10), (FP# = 20,11), (R15,12)))</td>
<td>(FP# = 20,14), (R3,13), (R15,9), (PC?, (R0,?), (L1, (IP1, (FP3, (SP4, (FP# = 16,6), (CPSR,8), (CPSR,8), (CPSR,8), (CPSR,8), (R3,10), (FP# = 20,11), (R15,12)))</td>
</tr>
<tr>
<td>16</td>
<td>(FP# = 20,14), (R3,13), (R15,9), (PC?, (L1, (IP1, (FP3, (SP4, (FP# = 16,6), (CPSR,8), (R3,10), (FP# = 20,11), (R15,12), (R0,15), (PC, (SP16), (FP16)))</td>
<td>(FP# = 20,14), (R3,13), (R15,9), (PC?, (L1, (IP1, (FP3, (SP4, (FP# = 16,6), (CPSR,8), (R3,10), (FP# = 20,11), (R15,12), (R0,15), (PC, (SP16), (FP16)))</td>
</tr>
</tbody>
</table>

**Table 4.6:** ARM program’s fixpoint for our Reaching Definition Analysis
4.1.2.2 Live Stores

Further, we want to know what stores are read, but not assigned to, by an instruction. Thus, we define the function

$$gen_{ls}: Instruction \rightarrow \mathcal{P}(Store),$$

that produces this set. It is defined by Table 4.7.

Applying $gen_{ls}$ to the instructions from our running example, we obtain the results shown in Table 4.8.

As stated in section 2.2 all ARM instruction can be conditionally executed. If an instruction is suffixed by a condition code (see Table 2.2), the CPSR register is read in order to check if the condition is met. Thus,

$$gen_{ls}(<Op>CC) = gen_{ls}(Op) \cup \{CPSR\}.$$  

4.1.2.3 Putting it all together

Having the reaching definition analysis’ fixpoint, that give us the assignments that may have been made at each program point, and the stores read by each instruction, we can now define chains that link program points that produce a value to the program points that use them. We shall call links that for each use of a store, associate all assignments that reach that use as Use-Definition chains or ud-chains. We will define such chains, $UD: Store \times PC_s \rightarrow \mathcal{P}(PC_s)$, by:

$$UD(store, pc) = \begin{cases} \{pc' \mid (store, pc') \in RD_{entry}(pc)\} & \text{if } store \in gen_{ls}(Ins^{pc}), \\ \emptyset & \text{otherwise.} \end{cases}$$

Where $Ins^{pc}$ is the instruction being pointed by the program counter $pc$. For the sake of clarity, let the us consider the following application of the $UD$ function on our running example.

$$UD([FP, # - 20], 15) = \{14, 11\}$$
\[ \text{gen}_{ls} : \text{live stores} \]

\[
\begin{align*}
gen_{ls}(BX R_i) & = \{ R_i \} \\
gen_{ls}(Bexp) & = \emptyset \\
gen_{ls}(BL R_i) & = \emptyset \\
gen_{ls}(MOV R_i, Op2) & = \{ R_j \mid R_j \in Op2 \} \\
gen_{ls}(MVN R_i, Op2) & = \{ R_j \mid R_j \in Op2 \} \\
gen_{ls}(CMP R_i, Op2) & = \{ R_i \} \cup \{ R_j \mid R_j \in Op2 \} \\
gen_{ls}(CMN R_i, Op2) & = \{ R_i \} \cup \{ R_j \mid R_j \in Op2 \} \\
gen_{ls}(TEQ R_i, Op2) & = \{ R_i \} \cup \{ R_j \mid R_j \in Op2 \} \\
gen_{ls}(TST R_i, Op2) & = \{ R_i \} \cup \{ R_j \mid R_j \in Op2 \} \\
gen_{ls}(AND R_i, R_j, Op2) & = \{ R_j \} \cup \{ R_k \mid R_k \in Op2 \} \\
gen_{ls}(EOR R_i, R_j, Op2) & = \{ R_j \} \cup \{ R_k \mid R_k \in Op2 \} \\
gen_{ls}(SUB R_i, R_j, Op2) & = \{ R_j \} \cup \{ R_k \mid R_k \in Op2 \} \\
gen_{ls}(RSB R_i, R_j, Op2) & = \{ R_j \} \cup \{ R_k \mid R_k \in Op2 \} \\
gen_{ls}(ADD R_i, R_j, Op2) & = \{ R_j \} \cup \{ R_k \mid R_k \in Op2 \} \\
gen_{ls}(ADC R_i, R_j, Op2) & = \{ R_j \} \cup \{ R_k \mid R_k \in Op2 \} \\
gen_{ls}(SBC R_i, R_j, Op2) & = \{ R_j \} \cup \{ R_k \mid R_k \in Op2 \} \\
gen_{ls}(RSC R_i, R_j, Op2) & = \{ R_j \} \cup \{ R_k \mid R_k \in Op2 \} \\
gen_{ls}(ORR R_i, R_j, Op2) & = \{ R_j \} \cup \{ R_k \mid R_k \in Op2 \} \\
gen_{ls}(BIC R_i, R_j, Op2) & = \{ R_j \} \cup \{ R_k \mid R_k \in Op2 \} \\
gen_{ls}(MRS R_i, CPSR) & = \{ CPSR \} \\
gen_{ls}(MSR R_i, CPSR) & = \{ R_i \} \\
gen_{ls}(MUL R_i, R_j, R_k) & = \{ R_j, R_k \} \\
gen_{ls}(MLA R_i, R_j, R_k, R_l) & = \{ R_j, R_k, R_l \} \\
gen_{ls}(LDR R_i, Addr) & = \{ Addr \} \\
gen_{ls}(STR R_i, Addr) & = \{ R_i \} \\
gen_{ls}(LDM R_i, R_{List}) & = \emptyset \\
gen_{ls}(STM R_i, R_{List}) & = \{ R_i \} \\
gen_{ls}(SWP R_i, R_j, R_k) & = \{ R_j \} \\
gen_{ls}(SWI R_i, exp) & = \emptyset
\end{align*}
\]

\textbf{Table 4.7: gen}_{ls} \text{ function}
stores read of ARM program in listing 4.2

<table>
<thead>
<tr>
<th>gen_{ls}(MOV IP, SP)</th>
<th>=</th>
<th>[SP]</th>
</tr>
</thead>
<tbody>
<tr>
<td>gen_{ls}(STMFD SP!, { FP, IP, LR, PC })</td>
<td>=</td>
<td>{PC, LR, IP, FP}</td>
</tr>
<tr>
<td>gen_{ls}(SUB FP, IP, #4)</td>
<td>=</td>
<td>{IP}</td>
</tr>
<tr>
<td>gen_{ls}(SUB SP, SP, #8)</td>
<td>=</td>
<td>{SP}</td>
</tr>
<tr>
<td>gen_{ls}(MOV R3, #5)</td>
<td>=</td>
<td>∅</td>
</tr>
<tr>
<td>gen_{ls}(STR R3, [FP; #16])</td>
<td>=</td>
<td>{R3}</td>
</tr>
<tr>
<td>gen_{ls}(LDR R3, [FP; #16])</td>
<td>=</td>
<td>{[FP, #16]}</td>
</tr>
<tr>
<td>gen_{ls}(CMP R3, #6)</td>
<td>=</td>
<td>{R3}</td>
</tr>
<tr>
<td>gen_{ls}(BLE, L2)</td>
<td>=</td>
<td>{CPSR}</td>
</tr>
<tr>
<td>gen_{ls}(MOV R3, #1)</td>
<td>=</td>
<td>∅</td>
</tr>
<tr>
<td>gen_{ls}(STR R3, [FP; #20])</td>
<td>=</td>
<td>{R3}</td>
</tr>
<tr>
<td>gen_{ls}(B, L1)</td>
<td>=</td>
<td>∅</td>
</tr>
<tr>
<td>gen_{ls}(MOV R3, #0)</td>
<td>=</td>
<td>∅</td>
</tr>
<tr>
<td>gen_{ls}(STR R3, [FP; #20])</td>
<td>=</td>
<td>{R3}</td>
</tr>
<tr>
<td>gen_{ls}(LDR R0, [FP; #20])</td>
<td>=</td>
<td>{[FP, #20]}</td>
</tr>
<tr>
<td>gen_{ls}(LDMEA FP, { FP, IP, PC })</td>
<td>=</td>
<td>∅</td>
</tr>
</tbody>
</table>

Table 4.8: gen_{ls} applied to the ARM program in listing 4.2
This tell us that the value of the \([FP, # - 20]\) store, at program counter \(pc = 15\), either comes from the instruction being pointed by the program counter \(pc = 14\), or the one at \(pc = 11\) (see Figure 4.1).

Typically, these ud-chains are part of a means for dead code elimination and code motion. In our case we are concerned with the evaluation of conditional branches. Determining the origin(s) of the value held by the \(\text{CPSR}\) register at a conditional branch instruction, will yield the instructions that influence its contents, and thus may allow us to predict the outcome of the condition.

Computing the ud-chain at the conditional branch instruction we get

\[
UD(\text{CPSR}, 9) = \{8\},
\]

meaning that the value of the \(\text{CPSR}\) register depends on the instruction pointed by the program counter \(pc = 8\). Now, for each of the yielded program counters, it is a matter of computing its ud-chain(s). For our running example, the following would be computed:

\[
\begin{align*}
UD(R3, 8) &= \{7\}, \\
UD([FP, # - 16], 7) &= \{6\}, \\
UD(R3, 6) &= \{5\}, \\
UD(*, 5) &= \emptyset,
\end{align*}
\]

i.e., the only store yielding a result different than \(\emptyset\) by the instruction pointed by \(pc = 8\) is \(R3\). From this, we determine that the value of the \(R3\) register depends on the instruction being pointed by the program counter \(pc = 7\). The two subsequent ud-chains are performed analogously. Finally, the instruction pointed by program counter \(pc = 5\) does not yield any value, meaning that the trace of dependences has ended. This is due to the fact that in the instruction pointed by program counter \(pc = 5\), MOV \(R3, #5\), the store \(R3\) is directly affected by a constant.

The computed ud-chains yield one trace, \([5, 6, 7, 8]\), that tells us which are the instructions necessary to consider in order to determine the outcome of the conditional branch. In this case, as we seen above, it will always evaluate to \(true\), thus, we can safely remove the edge branching to the opposite outcome. This will allow to produce a more precise timing analysis, since there is no point in considering a path that can never be taken.
4.2 Pipeline Analysis

Having generated the [ICFG], it is now time to cope with the overlapping effects of the pipeline. As mentioned in section 2.3, an instruction goes through a number of pipeline stages and consumes a number of cycles when it reaches the execute stage. The execution of a sequence of instructions overlaps in the pipeline as far as the data dependences between instructions allow it. Depending on its initial state, the execution of a sequence of instructions may result in different traces, i.e., sequence of execution states, and thus different number of cycles. Naturally, the length of a trace, is the number of cycles that this execution took. Thus, concrete execution can be seen as the execution of a sequence of instructions, starting in a concrete pipeline state, producing a trace of such states, whose length is the number of cycles it spent.

In the following, before moving to the analysis of the pipeline itself, we will first introduce and motivate the general concepts of the abstract interpretation framework.

4.2.1 Abstract Interpretation

A program is characterized by its control-flow graph, with a set of edges \( E \subseteq V \times Ins \times V \) and with the induced set of vertices, where \( V \) represents the program points, \( v_i \in V \) models the program’s entry point and \( Ins \) models the instruction to be executed whenever taking that edge.

A semantic function \([\_] : Ins \rightarrow (S \rightarrow S)\) assigns to each \( ins \in Ins \), a transfer function that models its effect on the program state \( S \), being evaluated. For instance, in the ARM instruction set \([10]\), the instruction \( B \) address is specified as \( R15 := address \), i.e., update of the program counter register to the address given by the evaluation of the expression \( address \), and thus would have to be modelled accordingly to its specification.

The collecting semantics assigns for each program point \( V \), the set of program states \( S \), which may occur in any possible execution, i.e., \( CS : V \rightarrow \mathcal{P}(S) \) (where \( \mathcal{P}(S) \) stands for powerset of \( S \)). The analysis to be performed, can be specified by extracting a number of equations from the program being considered. There are two types of equations. The first one, relates exit with entry information for each program point \( V \). While the second, relates entry information of a program point \( V_i \), with exit information of nodes from which there exists an edge to the program point \( V_i \), i.e., \( \bigcup \{V_j \mid (V_j, ins, V_i) \in E\} \).

The resulting system of equations can be solved by computing the least fixpoint \( lfp(F) = \)
4.2. PIPELINE ANALYSIS

\[ F^n(\lambda v.\varnothing) \text{ of the functional } F : (V \to \mathcal{P}(S)) \to (V \to \mathcal{P}(S)) : \]

\[
F(f)(v') = \begin{cases} 
S_0 & \text{if } v' = v_{\text{in}}, \\
\bigcup_{(v_{\text{ins}},v') \in E} [\text{ins}](f(v)) & \text{otherwise},
\end{cases}
\]

where \( S_0 \subseteq S \) is the set of the program’s initial states.

However, as mentioned earlier computing the collecting semantics of a large and complex program \( P \) can be too expensive to be feasible. Hence, the analysis is performed on a simpler abstract domain \( \mathcal{D}_\alpha = (S, L, \beta, \gamma) \), where \( L = (L, \subseteq, \sqsubseteq, \perp, \top) \) is a complete semi-lattice and \( \beta : S \to L \) is a representation function, mapping concrete to abstract states. The idea is that \( \beta \) maps a state \( S \) to the best property describing it. Finally, \( \gamma : L \to \mathcal{P}(S) \) is a concretization function mapping abstract states to concrete states.

As we seen above, the collecting semantics operates over sets of states, while our abstract domain, operates over sets of properties. Thus, with the purpose of relating these two domains, we define an abstraction function \( \alpha : \mathcal{P}(S) \to L \), by \( \alpha(S') = \sqsubseteq \{ \beta(s) | s \in S' \} \). The concretization function \( \gamma \), and the abstraction function \( \alpha \), will therefore yield the following relation:

\[ \mathcal{P}(S) \xrightarrow{\gamma} L \]

If for the above relation the following condition

\[ \alpha(X) \subseteq l \iff X \subseteq \gamma(l), \]

holds, we shall call the pair \( (\alpha, \gamma) \) a Galois connection. Furthermore, in order to ensure termination we require the Ascending Chain Condition to hold, i.e., every ascending chain of elements eventually terminates. For this, both the abstraction function \( \alpha \), and the concretization function \( \gamma \), must be monotonic w.r.t. the \( \subseteq \) and \( \sqsubseteq \) operators, respectively.

The semantic function defined above, can now be redefined as an abstract semantic function \( \llbracket \cdot \rrbracket_\alpha : \text{Ins} \to (L \to L) \), over the abstract domain. The abstract counterparts of the transfer functions \( \llbracket \text{ins} \rrbracket \), i.e., \( \llbracket \text{ins} \rrbracket_\alpha \), must also be monotonic w.r.t. the \( \subseteq \) operator. Finally, the analysis can now be applied with the abstract collecting semantics \( CS_\alpha : V \to L \), such that \( \forall v \in V : CS(v) \subseteq \gamma(CS_\alpha(v)) \), i.e., the computed results are precise, or an over-approximation of the collecting semantics, and thus are safe.

The resulting system of equations can be solved by computing the fixpoint \( \text{If p}(F_\alpha) = F^n_\alpha(\lambda v.\perp) \) of the functional \( F_\alpha : (V \to L) \to (V \to L) : \)
CHAPTER 4. FROM PREDICTIONS TO CERTIFICATES

\[ F_\alpha(f)(v') = \begin{cases} 
  l_0 & \text{if } v' = v_{in}, \\
  \bigsqcup_{(v', ins, v') \in E} [ins]_\alpha(f(v)) & \text{otherwise}, 
\end{cases} \]

where the abstraction of the concrete initial states is defined as initial abstract state, thus \( \alpha(S_0) \sqsubseteq l_0 \).

It should be clear that, since the abstract transfer functions, \([ins]_\alpha\), are monotonic w.r.t. the \( \sqsubseteq \) operator, by induction we obtain \( F^n_\alpha(\lambda v. \perp) \sqsubseteq F^{n+1}_\alpha(\lambda v. \perp) \) for all \( n \). All the elements of the sequence are in \( L \), and since this is a finite set, not all elements of the sequence can be distinct, thus, there must be some \( n \) such that:

\[ F^{n+1}_\alpha(\lambda v. \perp) = F^n_\alpha(\lambda v. \perp) \]

Furthermore, since \( F_n^{n+1}(\lambda v. \perp) = F_\alpha(F^n_\alpha(\lambda v. \perp)) \), we have reached the least fixpoint of \( F_\alpha \), i.e., \( \text{lfp}(F_\alpha) \), and thus a solution to the equation system.

The analysis to be instantiated depends on the target processor being evaluated. In our current prototype implementation of ACCEPT we focus on the ARM7TDMI-S processor (see Chapter 2). Featuring a three-stage pipeline, the analysis to be performed must cope with the overlapping effects of instruction pipelining.

4.2.2 Analysis

Pipeline analysis tries to find how instructions move through the pipeline. However, rather than considering concrete execution on a concrete pipeline model\(^5\), it regards abstract execution of sequences of instructions on an abstract pipeline model\(^6\). This execution produces abstract traces, i.e., sequences of abstract states, where information contained in the concrete states might be missing. Register contents are unknown, thus, w.r.t. WCET bound prediction, in order to obtain a safe analysis the worst-case scenario must always be assumed to happen. For instance, the MUL instruction takes \( m + 1 \) cycles to execute, the value of \( m \) depending on the contents of the operand, since information about register contents is lacking, for a safe upper-bound \( m \) must be assumed to have value 4 (see Table 2.3).

\(^5\)Which may be described in some formal language such as VHDL.

\(^6\)This abstract model is obtained by eliminating the components that are not relevant for the timing behaviour, and can also be described in a language like VHDL.
In the following, we will present our first steps towards the analysis of the pipeline. We base ourselves on the approach for pipeline analysis of a state-of-the-art processor described in [16].

In [16], the algorithm for pipeline analysis feeds to the abstracted pipeline model the instructions from a basic block, starting in the abstract pipeline state, and producing an abstract trace. Further, since the considered processor possesses a cache, and there might be basic blocks with more than one predecessor, one analysis for each pair of basic block and context per initial state is performed. From the set of yielded traces, the one with greatest length is taken as the upper-bound for this basic block in this context. In our case however, we are also concerned with the BCET but only need to cope with the effects of the pipeline.

Hence, an abstract state is a 3-tuple, where the first component is a set of states of the timing model $s_\alpha \in S_\alpha$ (like in [16]), the second component is a bound on the BCET and the third component a bound on the WCET. For the set $S_\alpha$, let $\mathcal{P}(S_\alpha) \times Z \times Z$ be ordered by

$$(S_{\alpha 1}, b_1, w_1) \sqsubseteq (S_{\alpha 2}, b_2, w_2) \text{ iff } S_{\alpha 1} \subseteq S_{\alpha 2} \land b_1 \geq b_2 \land w_1 \leq w_2$$

Our join operator will then be defined by

$$\bigsqcup\{(S_i, b_i, w_i \mid i = 1, ..., k)\} = \bigsqcup\{S_i \mid i = 1, ..., k\}, \min_i b_i, \max_i w_i\}$$

An abstract timing model is a state machine whose transitions corresponds to clock cycles. In our case we have to consider the three-stages of the pipeline, and keep track on which stage the instructions is on. In [16] a fictitious stage is added in order to determine whenever an instruction has finished execution. Further, no Galois Connections was established. Instead, they rely solely on a concretization function.

Indeed, up to the present, there is no suitable representation of sets of concrete pipeline states to single abstract states, and is very likely hard to find [12].

4.3 Bound Calculation

The previous pipeline analysis determined execution time bounds on the basic blocks. Determining the WCET and BCET is now a matter of finding the paths that lead to the
longest and shortest execution time, respectively. In timing analysis this is usually achieved by means of Integer Linear Programming (ILP) techniques.

Let $t_{wcet}$ denote the longest execution time of a program. Computing $t_{wcet}$ could be performed simply by summing the execution time of the $n$ nodes $v \in V$, belonging to the path leading to the longest execution time:

$$t_{wcet} = \sum_{i=1}^{n} t(v_i),$$

where $t(v_i)$ is the execution time of the node $v_i$. This could be feasible, if this path leading to the longest execution would be known beforehand. Determining it can be too much expensive for real programs. To surpass this obstacle, rather than analysing paths explicitly, Implicit Path Enumeration (IPE) counts the number of executions of basic blocks. Hence, the above sum can be reformulated as:

$$t_{wcet} = \sum_{v \in V} t(v_i).cnt(v_i),$$

where $cnt(v_i)$ stands for the execution count of the node $v_i$. This sum is a linear combination, since the execution times $t(v_i)$ are constants, as they have been computed in the previous pipeline analysis. Moreover, it is clear that the $cnt(v_i)$’s must be integer values. These are constrained according to the program’s behaviour. There are two types of constraints: program structural constraints and program functionality constraints. The former constraints are derived automatically from the program’s BBG, while the latter are either derived via a previous static analysis or (preferably) provided by the user, for instance, via annotations.

The number of times that the execution reaches a node $v$, is equal to the number of times that the execution leaves the node, which is also equal to the node execution count, i.e. $cnt(v)$. This is depicted in Figure 4.2. This principle can be applied as a basis to automatically obtain the structural constraints.

However, the structural constraints do not tell anything about loop bounds. This information is supplied as functionality constraints, and since they cannot be always automatically inferred, they are therefore usually provided as annotations by the user.

Let us demonstrate this by means of an example retrieved from [45]. Figure 4.3 depicts a basic block graph of a program with an if-statement nested within a loop.

Using the principle depicted in Figure 4.2 the structural constraints are inferred from the basic block graph, yielding:
4.3. BOUND CALCULATION

Figure 4.2: The number of times that the incoming and outgoing edges are crossed is equal, and so is the execution count of \( v \)

The first structural constraint stipulates that the edge \( e_1 \) is crossed once. The remaining are inferred by applying the principle shown in Figure 4.2. For instance, looking on the third structural constraint, the execution count of the basic block represented by node \( v_2 \), is equal to the number of times that its incoming edges, \( e_2 \) and \( e_8 \), are crossed, and also equal to the number of times that its outgoing edges, \( e_3 \) and \( e_9 \), are crossed.

However, as stated above, the structural constraints do not express any information regarding loop bounds. The idea is that the user may have domain-specific knowledge (e.g. ranges for input values), and thus can leverage it as functionality constraints. For instance, by knowing that the value of the variable \( k \), is positive before entering the loop, we know that the loop body will be executed between 0 and 10 times. Thus, the following functionality constraint can be defined:

\[
0 \leq \text{cnt}(v_1) \leq \text{cnt}(v_3) \leq 10 \text{cnt}(v_1)
\]

Besides loop bounds, functionality constraints can also be used to cope with other aspects of the program’s behaviour. A closer inspection of the code shows that node \( v_5 \) can only be executed once. In order to deal with this, we define another functionality constraint:

\[
\begin{align*}
\text{cr}(e_1) &= 1 \\
\text{cnt}(v_1) &= \text{cr}(e_1) = \text{cr}(e_2) \\
\text{cnt}(v_2) &= \text{cr}(e_2) + \text{cr}(e_8) = \text{cr}(e_3) + \text{cr}(e_9) \\
\text{cnt}(v_3) &= \text{cr}(e_3) = \text{cr}(e_4) + \text{cr}(e_5) \\
\text{cnt}(v_4) &= \text{cr}(e_4) = \text{cr}(e_6) \\
\text{cnt}(v_5) &= \text{cr}(e_5) = \text{cr}(e_7) \\
\text{cnt}(v_6) &= \text{cr}(e_6) + \text{cr}(e_7) = \text{cr}(e_8) \\
\text{cnt}(v_7) &= \text{cr}(e_9) = \text{cr}(e_{10})
\end{align*}
\]
Determining the WCET of the BBG in Figure 4.3 is now performed using $t_{\text{wcet}}$'s sum in the objective function of the ILP and maximizing it accordingly with the defined structural and functionality constraints. Therefore, the objective function of the generated ILP is as follows.

$$\max: \sum_{v \in V} t(v_i) \cdot \text{cnt}(v_i),$$

It should be clear that determining the BCET is performed analogously. Rather than maximizing the objective function, it is minimized.
4.4 Certificate Production

In section 2.3 a pipeline analysis was performed, by which bounds on the execution times of the basic blocks were produced. This was achieved by means of a fixpoint computation. This fixpoint allowed us to calculate both the BCET and WCET by means of ILP techniques (see section 4.3), and with this verify the compliance with a giving timing specification. However, we can also let this fixpoint play the role of a certificate.

In the context of mobile code safety one cannot trust the origin of the program. Hence, by adding to the code the certificate and sending both to a program consumer, it can be performed a local and independent check of the program's timing behaviour, thereby avoiding the need to trust in the code producer.

4.5 Certificate Validation

The program consumer receives a program with its certificate. In order to check the compliance with the intended timing specification, the first step is to compute the program's ICFG and verify that the certificate is a valid abstraction. Then, since the certificate is supposedly a fixpoint, the checking procedure can be written as:

\[
\text{Check}(\text{Certificate}) = \begin{cases} 
\text{True} & \text{if } F_{\alpha}(\text{Certificate}) = \text{Certificate}, \\
\text{False} & \text{otherwise.}
\end{cases}
\]

Since the certificate is supposed to be a fixpoint, another iteration over it cannot change anything, thus, on the program consumer side, a simple one-pass computation is sufficient to check that the certificate is indeed a fixpoint.

In case that the received certificate do not behave as a fixpoint, the program consumer can simply reject the program. One could argue that we could let the program run, and kill its execution in the case of a timing behaviour non-compliance. However, that would be a waste of resources, and in the context of embedded systems, that tend to have very limited computational resources, is unacceptable. On the other hand, if the certificate is indeed a fixpoint, then the program consumer can locally compute the BCET and WCET by the ILP techniques described in section 4.3, and thus check the compliance with the

---

7Since the certificate is in fact a fixpoint, Abstraction-Carrying Code is sometimes also referred as Fixpoint-Carrying Code [46]
timing specification. Furthermore, it should be noted that in this framework, it is also possible for the program consumer to define new timing policies. For instance, one can be interested in tightening the timing constraints.

This validation process requires that both the producer and consumer share the same abstract transfer functions. Indeed, if the consumer used different abstract transfer functions the certificate checking process would be inefficient, and thus prohibitive for such scarce resource equipments as embedded systems. One could argue that the independence in the timing validation process is compromised by that fact, however, it should be noted that the trusted computing base is limited to this checking operation, i.e., a simple, easy-to-trust fixpoint checker, that only has to perform a one iteration process. Hence, this approach allows to detect if a program has been tampered with, since an adulteration in the program code would be detected when performing the checking operation, i.e., the fixpoint iteration. In the context of mobile code, this is particularly relevant since, rather than simply put a blind confidence on a previous timing analysis, one can validate the program’s timing behaviour by solely relying on a fixpoint checker.

△ △ △

In this chapter we presented the fundamentals that underlies the tool developed in the context of this thesis. In the next chapter we present CATA - Certificates for Timing Analysis.
Chapter 5

CATA - CertificAtes for Timing Analysis

In this chapter we present an overview of the functionalities of CATA, the tool that implements the fundamentals presented in the previous chapter.

5.1 Main Features

CATA is the tool developed in the context of this work. It is a command-line application, written in OCaml [47], that features the options displayed in Figure 5.1.

![Figure 5.1: CATA - CertificAtes for Timing Analysis](image)

It was developed as part of the ACCEPT project, with the goal of computing timing bounds and producing a certificate that could be used to perform an independent validation w.r.t the predicted bounds. However, CATA can also be used as a standalone appli-
CHAPTER 5. CATA - CERTIFICATES FOR TIMING ANALYSIS

The analysis input can be either an ARM assembly file or an AIR file [48]. AIR files are used to represent control-flow graphs for real-time systems analyses. This format is based on Common Representation Language 2 (CRL2), a format developed in cooperation by Saarland University and AbsInt Angewandte Informatik GmbH, and is used as an internal intermediary representation for the tools developed at AbsInt (e.g. aiT).

The goal of AIR is to allow different timing analysis tools to exchange code and analysis results. Its development was made together with the cooperation of world-leading timing analysers tool providers, within the ARTIST2 project [49]. This goal originated several thesis [50, 51, 52], all with the objective of fostering collaboration between different tools. For instance, in [50] the purpose of this MSc thesis was to adapt the SWEET timing analyser to cope with the AIR format so that it could interact with aiT.

In order to promote cooperation with other timing analysers tools, CATA supports the AIR format, both as an input for timing analysis, and as output when receiving an ARM assembly file as input. However, the only available documentation regarding the AIR format is AIR File

ARM Assembly File

CATA

Control-Flow Reconstruction

Pipeline Analysis

Bound Calculation

Certificate (fixpoint)

BCET and WCET bounds

Figure 5.2: CATA’s architecture
5.1. MAIN FEATURES

format [48], dates from 2006, is labelled as ongoing work, and lacks information about its grammar definition. Nevertheless, our implementation of the AIR format is still strongly based on the one found in [48], and is discussed in Appendix 6.

As shown in Figure 5.2, there is an external element involved in the bound computations: lp_solve [53]. lp_solve is an open-source ILP solver that is used by CATA to compute bounds on the BCET and WCET.

The process of checking the generated certificate is performed by CATA_check, a tiny version of CATA. It starts by extracting the ICFG of the received program and confronts it with the received certificate. Then, performing one iteration over it will yield the validity of the program, and allow the computation of the BCET and WCET.
Chapter 6

Conclusions and Future Work

Abstract Interpretation has been widely used in the industry, being static timing analysis one of its most successful applications [12]. In our approach we also plan on using the Abstract Interpretation framework as the underlying technique, enabling us to take into account the effects of instruction overlapping caused by the pipeline. By explicitly following a standard fixpoint computation strategy [43], we are then able to apply standard integer linear programming techniques that yield the BCET and WCET.

Moreover, this fixpoint computation will allow us to infer an abstract model of the program, which can then be used as a certificate, i.e., a program consumer can locally validate the received program w.r.t. to its timing behaviour, by simply checking that this abstract model is indeed a fixpoint (a one-pass process), and then compute the BCET and WCET with the received certificate.

Our current prototype implementation of CATA is still a work in progress. At this stage there are still some open issues that remain to be addressed. Indeed, the all idea behind ACCEPT should be seen as a proof-of-concept with still large room for improvement. Namely:

- Rather than a binary file, CATA considers an ARM assembly file as input. In the context of ACCEPT since it features a WCET-aware compilation process, considering receiving an ARM assembly file as input is perfectly feasible, in view of the fact that the timing analysis would be integrated with the compilation process itself. However, as a standalone application, CATA would need to cope with binary files in order to make it suitable for a real scenario.

While it might seem that a decompilation process from a binary would be straight-
forward, there are in fact several challenges associated with it \cite{54}. For instance, doing a simple direct mapping between bit-patterns and instructions would be error-prone, since some compilers insert switch tables between instructions. Furthermore, there is actually a decompilation framework based on abstract interpretation \cite{55}.

- The data-flow analysis for conditional branches presented in Section \ref{sec:dataflow} is limited to a intra-procedure analysis. Extending it to an inter-procedural analysis would be of high benefit to enhance the power of the analysis. Also, this analysis is only applied to branch instructions, but indeed, since in ARM all instructions can be conditionally executed, the same principle could be applied to the remaining instructions. For instance, detecting that a MOVLT instruction would never be executed, would allow us to determine that it would only consume 1 cycle (see Table \ref{tab:instructions}), and thus leverage a more accurate timing analysis.

- One of the main challenges that we face in order to make ACCEPT useful in practice is the size of the produced certificates. Embedded systems are known for their scarce resources, and thus, cannot afford to waste computational means. In \cite{56}, Albert et al introduce the notion of a reduced certificate, with the objective of producing a certificate that only contains the essential information which the program consumer cannot reproduce by itself, while not yielding an overhead in the certificate checking process. Moreover, in \cite{57} a certified fixpoint compression algorithm is presented that is able to produce reduced certificates from the results of untrusted static analysers.

- Our pipeline analysis is still not completed. For now, we let the fixpoint computed in the data-flow analysis for conditional branches play the role of the certificate. The calculated execution times do not take into account the overlapping effects of the pipeline. Hence, although these computed bounds are safe, they are not as precise as they would be if the pipeling effects were taken into account.

Our actual focus on the certification generation part of the ACCEPT platform is now on the pragmatical evaluation of our proposal. For now we are not concerned with performance, but with correctness and adequacy (in the context of mobile code). However, a timing analysis platform is only useful if it provides tight and safe bounds. In this sense, we use state of the art algorithms for their calculation. Nevertheless, comparing
equitably this kind of platform against reference tools [7], even pragmatically in the form of benchmark, is not a trivial task, the same source-code, compilation process and/or low-level code and target architecture must be considered. However, as future work we also plan to report on case studies and practical results of our framework.

To the best of our knowledge this is the first work applying the concepts of Abstraction-Carrying Code to the static timing analysis field.
References


REFERENCES


Acronyms

**RISC**  Reduced Instruction Set Computer

**CPSR**  Current Program Status Register

**PC**  Program Counter

**LR**  Link Register

**SP**  Stack Pointer

**CFG**  Control-Flow Graph

**ICFG**  Interprocedural Control-Flow Graph

**CG**  Call Graph

**BBG**  Basic Block Graph

**RTS**  Real-Time Systems

**WCET**  Worst-Case Execution Time

**BCET**  Best-Case Execution Time

**CPI**  Clock Cycles per Instruction

**DFA**  Data-Flow Analysis

**PCC**  Proof-Carrying Code

**ACC**  Abstraction-Carrying Code

**ACCEPT**  Abstraction-Carrying CodE Platform for Timing validation
**ACRONYMS**

**IPE**  Implicit Path Enumeration

**ILP**  Integer Linear Programming

**CATA**  CertificAtes for Timing Analysis

**AIR**  ARTIST 2 Interchange Format

**CRL2**  Common Representation Language 2
AIR Specification Format

In this Appendix we describe the grammar of the AIR format.

.1 Language Grammar

In the following listing[1] we present our implementation of the AIR format. Terminal symbols are written in capitals and their meaning should always be clear.

Listing 1: AIR Grammar

```
1  air:
2    | EOF
3    | header body END EOF
4  
5  header:
6    | CRL
7    SPECIFICATION specification_name
8    IMPLEMENTATION implementation_name
9    VERSION
10       generation_code
11       safety_code
12       change_code
13       implementation_version
14       implementation_subversion
15  
16  specification_name:
17    | UUIDidentifier
18  
19  implementation_name:
```
AIR SPECIFICATION FORMAT

<table>
<thead>
<tr>
<th>UUIDidentifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDENT</td>
</tr>
</tbody>
</table>

| dummy:          |
| */empty*/       |
| IDENT          |

| generation_code: |
| INT            |

| safety_code:    |
| INT            |

| change_code:    |
| INT            |

| implementation_version: |
| INT            |

| implementation_subversion: |
| INT            |

| lc_name:        |
| IDENT          |

| body:           |
| body body_element |
| body_element    |

| body_element: |
| GLOBAL global |
| ROUTINE routine |
| DATA data    |
| META meta    |
1. LANGUAGE GRAMMAR

lc_id_def:
   | lc_name

global:
   | simple_item_lc_name_and_global_special

simple_item_lc_name_and_global_special:
   | lc_id_def global_special lc_name_attr_list COLUMN
   | lc_id_def global_special COLUMN

lc_name_attr_list:
   | TWODOTS attr_lc_name_comma_list

attr_lc_name_comma_list:
   | attr_lc_name COMMA attr_lc_name_comma_list
   | attr_lc_name

attr_lc_name:
   | lc_name EQUALS value

global_special:
   | dummy

routine:
   | lc_id_def optional_lc_name_ctxt_attr_list COLUMN
   | lc_id_def optional_lc_name_ctxt_attr_list
   | LBRAC optional_context_def_list optional_block_list RBRAC

optional_context_def_list:
   | /*empty*/
   | context_def_list

context_def_list:
   | context_def context_def_list
   | context_def
optional_block_list:
  | /*empty*/
  | block_list

block_list:
  | block block_list
  | block

optional_lc_name_ctxt_attr_list:
  | /*empty*/
  | lc_name_ctxt_attr_list

lc_name_ctxt_attr_list:
  | TWODOTS attr_lc_name_ctxt_comma_list

attr_lc_name_ctxt_comma_list:
  | lc_name_ctxt COMMA attr_lc_name_ctxt_comma_list
  | lc_name_ctxt

block:
  | lc_id_def block_special lc_name_ctxt_attr_list COLUMN
  | lc_id_def block_special lc_name_ctxt_attr_list
      LBRAC edge_list instruction_list RBRAC

edge_list:
  | edge_list edge
  | edge

instruction_list:
  | instruction instruction_list
  | instruction

block_type:
  | NORMAL | START | END | CALL | RETURN | EXTERNAL | IMPASSE
1. LANGUAGE GRAMMAR

edge:
  | simple_item_lc_name_ctxt_edge_special

simple_item_lc_name_ctxt_edge_special:
  | lc_id_def edge_special lc_name_ctxt_attr_list COLUMN
  | lc_id_def edge_special COLUMN

edge_special:
  | /* empty */
  | LPAREN edge_type RPAREN

edge_type:
  | NORMAL | TRUE | FALSE | ZERO | DELAY | CALL | RETURN | IMPASSE

instruction:
  | item_lc_name_ctxt_op_instr_special

item_lc_name_ctxt_op_instr_special:
  | lc_id_def instr_special optional_lc_name_ctxt_attr_list COLUMN
  | lc_id_def instr_special optional_lc_name_ctxt_attr_list LBRAC op_list RBRAC

op_list:
  | op op_list
  | op

instr_special:
  | addr_and_width

addr_and_width:
  | INT TWODOTS QUESTION_MARK
  | QUESTION_MARK TWODOTS QUESTION_MARK
  | QUESTION_MARK TWODOTS INT
  | INT TWODOTS INT

op:
| simple_item_lc_name_ctxt_op_special |

simple_item_lc_name_ctxt_op_special:
| lc_id_def op_special lc_name_ctxt_attr_list COLUMN |
| lc_id_def op_special COLUMN |

op_special:
| STRING |

data:
| item_lc_name_bytes_data_special |

item_lc_name_bytes_data_special:
| lc_id_def data_special optional_lc_name_attr_list COLUMN |
| lc_id_def data_special optional_lc_name_attr_list LBRAC bytes_list RBRAC |

optional_lc_name_attr_list:
| */empty*/ |
| lc_name_attr_list |

lc_name_attr_list:
| TWODOTS lc_name_comma_list |

lc_name_comma_list:
| lc_name COMMA lc_name_comma_list |
| lc_name |

data_special:
| dummy |

bytes_list:
| bytes bytes_list |
| bytes |

bytes:
| simple_item_lc_name_bytes_special |
bytes_special:
  | /* empty */
  | addr_and_width

simple_item_lc_name_bytes_special:
  | lc_id_def bytes_special COLUMN
  | lc_id_def bytes_special lc_name_attr_list COLUMN

meta:
  | item_lc_name_info_meta_special

item_lc_name_info_meta_special:
  | lc_id_def meta_special COLUMN
  | lc_id_def meta_special lc_name_attr_list COLUMN
  | lc_id_def meta_special LBRAC info_list RBRAC
  | lc_id_def meta_special lc_name_attr_list LBRAC info_list RBRAC

info_list:
  | info info_list
  | info

meta_special:
  | dummy

info:
  | simple_item_lc_name_info_special

simple_item_lc_name_info_special:
  | lc_id_def info_special optional_lc_name_attr_list COLUMN

info_special:
  | dummy
lc_name_ctxt:
  | lc_name
  | lc_name LESSTHAN context_ref GREATERTHAN

context_ref:
  | lc_name
  | STAR

context_def:
  | CONTEXT lc_id_def TWODOTS optional_context_seq COLUMN

optional_context_seq:
  | /*empty*/
  | context_seq

context_seq:
  | context_simple_comma_list

context_simple_comma_list:
  | context_simple COMMA context_simple_comma_list
  | context_simple

context_simple:
  | context_match
  | context_minimum
  | context_repeat
  | LPAREN context_seq RPAREN

context_match:
  | lc_name RIGTH_ARROW lc_name
  | QUESTION_MARK RIGTH_ARROW lc_name
  | lc_name RIGTH_ARROW QUESTION_MARK
  | QUESTION_MARK RIGTH_ARROW QUESTION_MARK

context_minimum:
1. LANGUAGE GRAMMAR

```
context_simple STAR
context_simple STAR MINUS
context_simple PLUS
context_simple LBRAC INT COMMA RBRAC

context_repeat:
| context_simple QUESTION_MARK
| context_simple LBRAC INT COMMA INT RBRAC

value:
| value_numeric
| value_range
| value_date
| value_item
| value_vector
| value_map_symbol_value
| value_map_item_value
| value_functor

value_numeric:
| value_unsigned
| value_float

value_unsigned:
| INT

value_float:
| QUESTION_MARK FLOAT_KEYWORD RPAREN FLOAT LPAREN

value_range:
| DOT DOT
| value_numeric DOT DOT
| DOT DOT value_numeric
| value_numeric DOT DOT value_numeric

value_date:
```
| value_date_posix
| value_date_humal

value_date_posix:
| QUESTION_MARK GMT LPAREN INT RPAREN

value_date_humal:
| QUESTION_MARK GMT LPAREN INT MINUS INT MINUS INT MINUS INT MINUS INT MINUS INT MINUS INT MINUS INT MINUS

value_item:
| lc_name

value_vector:
| LLPAREN RRPAREN
| LLPAREN vector_list RRPAREN

vector_list:
| vector_element_comma_list

vector_element_comma_list:
| vector_element COMMA vector_element_comma_list
| vector_element

vector_element:
| value
| INT EQUALS value

value_map_symbol_value:
| LBRAC RBRAC
| LBRAC symbol_map RBRAC

symbol_map:
| symbol_map_entry_comma_list

symbol_map_entry_comma_list:
| symbol_map_entry COMMA symbol_map_entry_comma_list
.1. LANGUAGE GRAMMAR

```
344  | symbol_map_entry
345
346  symbol_map_entry:
347    | symbol_map_key
348    | symbol_map_key EQUALS value
349
350  symbol_map_key:
351    | lc_name
352    | STRING
353    | INT
354
355  value_map_item_value:
356    | AT LBRAC RBRAC
357    | AT LBRAC item_map RBRAC
358
359  item_map:
360    | item_map_entry_comma_list
361
362  item_map_entry_comma_list:
363    | item_map_entry COMMA item_map_entry_comma_list
364    | item_map_entry
365
366  item_map_entry:
367    | item_map_key
368    | item_map_key EQUALS value
369
370  item_map_key:
371    | lc_name
372
373  value_functor:
374    | functor_c
375    | functor_mathematica
376    | functor_lisp
377    | functor_prefix
378    | functor_infix
379    | functor_circumfix
```
functor_suffix

functor_c:
| identifier LPAREN RPAREN
| identifier LPAREN vector_list RPAREN

functor_mathematica:
| identifier LLPAREN RRPAREN
| identifier LLPAREN vector_list RRPAREN

functor_lisp:
| LPAREN identifier RPAREN
| LPAREN identifier COMMA vector_list RPAREN

functor_prefix:
| LPAREN identifier value_vector RPAREN

functor_infix:
| LPAREN value_vector identifier value_vector RPAREN

functor_circumfix:
| LPAREN identifier value_vector identifier RPAREN

functor_suffix:
| LPAREN value_vector identifier RPAREN

identifier:
| IDENT