Mixed Low- and High Level Programming Language Semantics and Automated Verification of a Small Hypervisor.

Dissertation

zur Erlangung des Grades
des Doktors der Ingenieurswissenschaften (Dr.-Ing.)
der Naturwissenschaftlich-Technischen Fakultäten
der Universität des Saarlandes

Andrey Shadrin

ashadrin@gmail.com

Saarbrücken, September 2012

Tag des Kolloquiums:	27. August 2012
Dekan:	Prof. Dr. Mark Groves
Vorsitzender des Prüfungsausschusses:	Prof. Dr. Reinhard Wilhelm
1. Berichterstatter:	Prof. Dr. Wolfgang J. Paul
2. Berichterstatter:	Prof. Dr. Andreas Podelski
Akademischer Mitarbeiter:	Dr. Mark Kaminski

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet. Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form in einem Verfahren zur Erlangung eines akademischen Grades vorgelegt.

Saarbrücken, September 2012

Danksagung

An dieser Stelle möchte ich mich bei allen bedanken, die zum Gelingen der vorliegenden Arbeit beigetragen haben. Zunächst gilt mein Dank meiner Familie, die mich stets ermutigt hat meinen Weg zu gehen und mich während meines gesamten Studiums unterstützt hat.

Herrn Prof. Paul danke ich fur die Möglichkeit in einem der weltbesten Forscherteam im Systemverifikationsbereich mitgearbeitet zu haben.

Diese Arbeit wurde teilweise im Rahmen des Verisoft XT Vorhabens vom Bundesministerium fur Bildung und Forschung (BMBF) unter dem Förderkennzeichen 01 IS 07 008 und der Saarbrücker Graduiertenschule für Informatik gefördert.

Abstract

Hypervisors are system software programs that virtualize the architecture they run on and are usually implemented in a mix of (macro) assembly and a high-level language like C. To verify such a software, assembly parts as well as C parts should be verified, where reasoning about those parts is done in different semantics. At the end, both semantics should be brought together in an overall correctness theorem of such a software program. The formal integration of correctness results accomplished in distinct semantics is challenging but inevitable for systems verification.

This thesis is split into two parts. In the first one, we will present the mixed semantics of C and macro assembly. This semantics can handle mixed-language implementations where the execution context is changed by an external function call from assembly to C and vice versa. Also, we state a step-by-step simulation theorem between mixed programs and the compiled and assembled code.

In the second part, we present the correctness of a small hypervisor, called *Baby Hypervisor* (BHV), described by the mixed semantics. BHV virtualizes a 32-bit RISC architecture. The BHV functional verification was shown using Microsoft's VCC, an automatic verifier for C with contracts. BHV C portions were verified with VCC in [AHPP10]. For making macro assembly feasible with VCC the original macro assembly is translated to C code simulating processor. This is called the *simulation approach* [MMS08].

Kurzzusammenfassung

Hypervisor sind Software-Systemprogramme, die eine Hardwarearchitektur virtualisieren, eine Umgebung für die virtuelle Maschinen schaffen und gemischte Implementierungen in C- und Makroassemblerprogrammiersprachen haben. Um solche Programme zu verifizieren, müssen Assembler- sowie C-Portionen des Quellcodes verifiziert werden, wobei die Verifikation beider Teile in den verschiedenen Semantiken erfolgt. Schließlich sollten beide Semantiken zusammen in einen umfassenden Korrektheitssatz gebracht werden. Die Aufgabe der formalen Integration der Korrektheitsergebnisse aus verschiedenen semantischen Ebenen ist anspruchsvoll aber unvermeidlich für die Systemverifikation.

Diese Arbeit ist in zwei Teile aufgeteilt. Im ersten Teil stellen wir die gemischte Semantik von C- und Makroassemblerprogrammiersprache vor. Diese Semantik kann die gemischte Implementierung beschreiben, in der der Ausführungskontext von einem externen Funktionsaufruf von Assembler nach C und umgekehrt wechselt. Außerdem formulieren wir einen Schritt-für-Schritt-Simulationssatz zwischen den gemischten Programmen und dem übersetzten Quellcode.

Im zweiten Teil präsentieren wir die Korrekheit eines kleinen Hypervisors, genannt *Baby Hypervisor* (BHV), der in der gemischten Semantik beschrieben wird. BHV virtualisiert eine 32-Bit RISC-Architektur. Die funktionale Verifikation des BHVs wurde mit Hilfe des Microsoft-VCCs, eines automatischen C-Beweisers, durchgeführt. Die C-Portionen des BHVs wurden schon mit Hilfe des VCCs in [AHPP10] verifiziert.

Um eine Verifikation des Makroassembler-Quellcodes mit Hilfe des VCC zu ermöglichen, wird der Quellcode in C-Quellcode, der den Prozessor simuliert, übersetzt. Dies wird *simulation approach* genannt [MMS08].

Contents

CHAPTER 1

Introduction

Computer systems become more and more complex every day. At the same time, the growing complexity of those systems leads to an increasing number of possible failures in them. Since those systems are used in more and more safety critical places, for example, in automotive engineering, in security technology and in the sector of medical technology, problems of reliability of those systems have to be solved. Hence the absence of errors becomes crucial.

First, failures in software and hardware could lead to situations hazardous to one's life. Second, they could lead to a loss of a significant amount of money for companies. There are many sad accidents caused by faulty hard- and software:

- A software bug in the operating system of the radiation therapy device Terac-25 caused at least five casualties in the years 1985 to 1987.
- A hardware bug was found in the floating-point division operation in the first release of the Intel Pentium processor from 1993. Intel became aware of this bug after it has already sold more then 3 million chips. That bug caused a total loss of $475 million for Intel.
- The flight Ariane-5 flight crashes 40 seconds after launch on June 4, 1996; it crashed due to an error in the conversion of a 64-bit floating point number into a 16-bit integer value. The damage has been estimated with $500 million.
- Toyota issued a recall of several vehicle models, because of a problem with the accelerator pedal. The recent (Prius) hybrid model has also been recalled due to a software problem in the anti-lock braking system. Up to 19 U.S. crash deaths over the past decade may be linked to accelerator-related problems at Toyota, congressional officials have said.

How can one ensure that computer systems do what they are intended to do? One approach to solve the problem is simply to run a system repeatedly with various inputs, i.e. to simulate the system. For example, to reduce the probability of a failure to 10^{-9}

for an hour-long NASA mission, one should test for more than 114,000 years, which is unfeasible. Another approach is the use of formal methods exploiting the expressivity and unambiguousness of the mathematical language to specify our systems, that is, proving correctness in a mathematical sense, by first formulating the accurate models of these systems and then verifying the formal assertions over these models. This approach is called *formal verification*. In industry formal methods are becoming more and more familiar tools for verifying individual modules of safety-critical computer systems.

The correctness of operating systems and hypervisors is critical for the security and reliability of computer systems. Hypervisors[1] are software programs that virtualize the underlying host architecture and which allow multiple *guests*(also known as operating systems together with their user processes) to run concurrently on a single host. Formal verification can provide solid evidence in arguing about implementation correctness of a system consisting of a hypervisor and operating systems. The code of many operating systems and hypervisors consist of (macro-)assembly and C functions calling each other. Formal verification of such programs must consider the correct interaction of code written in these languages which can only be guaranteed by using the same compiler calling conventions (or application binary interfaces (ABIs)).

The goals of this thesis are: (i) to present the integrated semantics of programming languages like C and macro-assembly, (ii) to formally verify a small hypervisor written in C and macro-assembly using an automatic C verifier (called *VCC* [Cor08]).

Related Work

The related work for this thesis is summarized in two categories: (i) macro-assembly semantics, and (ii) kernel verification.

Macro-Assembly Semantics There are few publications that one can somehow relate to the macro-assembly semantics. In [Fru08] Fruja specified the static and dynamic semantics model of the Common Intermediate Language (CIL) and showed the CIL's type safety for a large subset of CIL. In this publication the author defined an abstract stack-based machine for running CIL bytecode programs. In this machine the stack is explicitly modeled as a list of call frames and a new frame is put on the stack every time a method is called. This work might be helpful for those who are trying to do the verification of programs written in one of .NET compliant language (cf. [NET11] for an up-to-date list).

In [MCGW98] a stack-based typed low assembly language is proposed as a target language for code verification. The authors can encode any compiler calling conventions in their type system since everything about the stack including the stack frame header layout is exposed. This language is too low in a sense a lot of details about the underlying architecture and assembler are visible. These details makes the integration of this language with a high-level programming language like C complicated.

Integrated Semantics Based on W.J.Paul's lecture notes [Pau07], in [Tsy09] Tsyban presented an approach how to reason formally about the code implemented in the C dialect C0 with inline-assembly. This approach deals with inline-assembly statements as follows: whenever an assembly portion is encountered, the compiler simulation

[1] also called *virtual machine managers*(VMMs)

relation [Lei07] is applied to reach an equivalent assembly configuration and the execution continues in the assembly semantics until all assembly instructions are executed from this assembly portion. Then, the execution switches back to the consistent C0 configuration. This approach does not treat external function calls from C to assembly.

In the dissertation [Alk09] Alkassar showed how to verify a device driver against a realistic device and system model. For that, Alkassar used the original C0 semantics extended with so-called atomic *XCalls*[2](extended calls). Also, he extended the whole language stack used in Verisoft project so that each language from the stack could deal with device drivers. He abstracts the effect of low-level computations implemented in assembly in these atomic XCalls of the extended C0 semantics. Alkassar said: "An XCall is a procedure call that makes a transition on external state and communicates with C0 via parameter passing and return values." In the framework developed by Alkassar one can verify any code implemented in a mixture of C0 and inline assembly programming languages. In comparison to work done by Tsyban in [Tsy09], Alkassar has the aforementioned computation model that performs either C0 steps or assembly steps which are abstracted in XCalls. We believe that is possible to implement external assembly procedure calls from C0 in the C0 semantics with XCalls without additional effort. But to model external C0 function calls from assembly requires additional extensions of his model and the C0 linker [IdR09].

Kernel Verification In [Kle09] Klein summarizes approaches used presently and in the past for the verification of operating systems. He writes that first attempts to apply formal methods for verifying operating systems date back to the mid 1970s in PSOS [FN79, NBF+80, NF03] and UCLA [WKP79] projects.

In the end of the 1980s, a substantial progress in kernel verification was achieved with the KIT kernel [Bev87,Bev89] of the CIL stack [BHMY89a,BHMY89b,Moo03]. KIT (Kernel for Isolated Tasks) is a small operating-system kernel implemented in assembly that provides services for process scheduling, error handling, single-word message passing, IO-devices. This project is considered as groundbreaking in the area of *pervasive system verification*. This project is the first example of a completely verified kernel.

The FLINT group focuses on studying various problems in the system verification of operating systems. They developed an assembly code verification framework XCAP [NS06]and reported in [NYS07] and in [FSDG08] on their successful experience of applying it. So far, no integration of results into high-level programming languages has been reported yet.

The L4.verified kernel [KEH+09] is an example of a recent operating system verification effort that has achieved impressive code verification results. A detailed memory model for low-level pointer programs in C was applied in combination with separation logic [TKN07]. The assembler portions have not been verified in conjunction with the C code yet to our knowledge. Judging from their choice of C semantics, however, we are certain that all gaps present can be closed with reasonable additional effort when the right models and theories are applied.

Verisoft(XT) The Verisoft project [Vera] aimed at the pervasive formal verification of entire computer systems. In the Verisoft a micro-kernel framework was formally verified [APST10, Tsy09]. This framework was implemented in the C dialect C0 together with inline-assembly. This work perfectly demonstrates how to formally

[2]The concept of XCalls was introduced in [AHL+09].

reason about mixed implementations. In this work a non-optimizing verified compiler for C like language is considered [LP08].

Within the VerisoftXT project [Verb], Microsoft in collaboration with EMIC and Saarland University developed an automatic verifier for low-level and concurrent C code, called VCC [Cor08] to verify components of the Microsoft Hypervisor, the virtualization kernel of Hyper-V([LS09]).

The Baby Hypervisor project which is a subproject of the Microsoft Hyper-V project played an important role of driving the development of VCC as well as applying it to system verification. As a result of this subproject the C portions of a small hypervisor, called *baby hypervisor*, were verified in VCC [AHPP10].

Also, within the VerisoftXT project a simulation approach for verifying assembly code in VCC was invented [MMS08, Mau11].

Foundations and Contributions

This work is based on a number of results achieved within the Verisoft and VerisoftXT projects. We use the VAMP assembly semantics formally specified by Tsyban in [Tsy09] as a target implementation language in this thesis. We use the results achieved by Alkassar et al. [AHPP10] and then complete the formal verification of a small hypervisor. Also, we use the idea of Maus [Mau11] how to verify assembly with a C verifier.

This thesis presents five main contributions: (i) an approach to model macro-assembly languages with explicit dynamic stack, (ii) an approach to integrate C- and high-level assembly languages together in a single language developed in collaboration with S.Schmaltz, (iii) the complete paper-and-pencil proof and the complete formal proof of a small hypervisor, (iv) the soundness of a simulation approach used to verify code written in an assembly language with a general-purpose C verifier.

Outline

The remainder of the thesis is organized in six chapters.

- Chapter 2 introduces the computational model of VAMP assembly.
- In Chapter 3, we define the high level VAMP assembly language (VAMP macro assembly) as well as the VAMP macro assembler itself. Also, we will state the simulation theorem justifying correctness of execution VAMP macro assembly programs running on VAMP assembly machines.
- Chapter 4 presents the abstract C intermediate language. Also, we present the compiler correctness theory for an optimizing C compiler.
- In Chapter 5, we define the mixed language and assuming the C compiler correctness we state the simulation theorem which justifies the correctness of mixed programs running on VAMP assembly machines.
- Chapter 6 presents the specification and correctness of a small hypervisor. Here, we close the gaps in the theory developed in [AHPP10].
- In Chapter 7, we present the soundness of a translation based approach for verifying high level assembly portions with a C verifier.

- In Chapter 9 we outline possible future work and conclude.

Notation

Basics types, terms and operators We define the following basis types:

$$\begin{array}{rcl} \mathbb{N} & \stackrel{\text{def}}{=} & \text{set of all natural numbers including zero,} \\ \mathbb{Z} & \stackrel{\text{def}}{=} & \text{set of all integer numbers,} \\ \mathit{bool} & \stackrel{\text{def}}{=} & \text{set of boolean values } \{\mathsf{F}, \mathsf{T}\}, \\ \mathbb{B} & \stackrel{\text{def}}{=} & \text{set of binary digits } \{0, 1\}, \\ \mathit{String} & \stackrel{\text{def}}{=} & \text{set of strings.} \end{array}$$

The restricted sets of naturals and integers are denoted by

$$\begin{array}{rcl} \mathbb{N}_n & \stackrel{\text{def}}{=} & \{x \mid x \leq 2^n - 1 \ \wedge \ x \in \mathbb{N}\}, \\ \mathbb{Z}_n & \stackrel{\text{def}}{=} & \{x \mid -2^{n-1} \leq x \leq 2^{n-1} - 1 \ \wedge \ x \in \mathbb{Z}\}. \end{array}$$

The function $i2n :: \mathbb{Z} \mapsto \mathbb{N}$ converts an integer number to natural one. The function $n2i :: \mathbb{N} \mapsto \mathbb{Z}$ does the conversion in the other direction. The function $bool2nat :: bool \mapsto \mathbb{N}$ returns a natural number corresponding to a given boolean value, i.e. $bool2nat(\mathsf{T}) = 1$ and $bool2nat(\mathsf{F}) = 0$. The power set of $\mathbb{T}$ is denoted by $2^{\mathbb{T}}$. We use the term A to denote an arbitrary value. We write $t_\perp$ to extend a given type t with some error value $\perp$ (i.e. $t_\perp = t \cup \{\perp\}$). Such a type $t_\perp$ we call an *option* type. If a value $x \in t_\perp$ is a non-error value, then we write $\lfloor x \rfloor$ to denote that x is a non-error value.

Lists An n-tuple $(t_0, t_1, \ldots, t_{n-1})$, where $t_0 \in \mathbb{T}, t_1 \in \mathbb{T}, \ldots, t_{n-1} \in \mathbb{T}$, is an element of a sequence type of the length n over a type $\mathbb{T}$. We call such a sequence type a list type of n elements over type $\mathbb{T}$, and denote it by $\mathbb{T}^n$. We write $l \in \mathbb{T}^n$ to denote a list l of type $\mathbb{T}^n$. To denote the empty list of any list types $\mathbb{T}^n$ we introduce the polymorphic constant []. We denote a list type of arbitrary length over type $\mathbb{T}$ by $\mathbb{T}^*$. We can formally define this by

$$\mathbb{T}^* \quad \stackrel{\text{def}}{=} \quad \bigcup_{n \in \mathbb{N}} \mathbb{T}^n$$

We construct a list from elements $t_{n-1}, t_{n-2}, \ldots, t_0$ of the same type by $[t_{n-1}, t_{n-2}, \ldots, t_0]$. Let l be a list $[t_{n-1}, t_{n-2}, \ldots, t_0]$. The number of elements in a list l is returned by means of the function $|l|$. The functions $\mathbf{hd}(l)$ and $\mathbf{tl}(l)$ return the head and the tail of a given list l, respectively, where the head of a list is the leftmost element of this list, i.e. $\mathbf{hd}(l) = t_{n-1}$ and $\mathbf{tl}(l) = [t_{n-2}, \ldots, t_0]$. An arbitrary element i in a list l is accessed by $l[i]$. We write $l_1 \circ l_2$ to denote the concatenation of two lists l_1 and l_2. After concatenating two lists l_1 and l_2 the following hold:

$$\forall i \in \mathbb{N}. \ i < |l_1| \quad \longrightarrow \quad l_1[i] = (l_1 \circ l_2)[i + |l_2|].$$

The function $\mathbf{rev}(l)$ reverses a given list l. Formally,

$$\mathbf{rev}([t_{n-1}, t_{n-2}, \ldots, t_1, t_0]) \quad \stackrel{\text{def}}{=} \quad [t_0, t_1, \ldots, t_{n-2}, t_{n-1}].$$

We flatten a list l, whose elements are lists too, by means of the function $\mathbf{flat}(l)$. Update of the i-th element of a list l to a value v is denoted by $l[i := v]$. The function $\mathbf{sublist}(l_1, l_2)$ takes a list l_1 and a list l_2 of indices as input and returns a sublist of the list l_1 such that it contains only those elements from the list l_1 whose indices are

present in the list l_2. The function **take**(l, n) takes a list l and a natural number n as input and returns the list $[t_{|l|-1}, \ldots, t_{|l|-1-n}]$ of n elements. The function **drop**(l, n) takes a list l and a natural number n as input and returns such a list $[t_{|l|-1-n}, \ldots, t_0]$. The function

$$\textbf{top} \stackrel{\text{def}}{=} |l| - 1$$

returns the index of the head element from the list l. We write $a \in l$ to denote that an element a is present in a list l.

Bit vectors A bit vector is represented as a list of binary digits, i.e. an element in $\mathbb{B}^*$, in which the leftmost bit is the most significant bit and the rightmost bit is the least significant bit. For a bit vector bv of length n we denote by $\langle bv \rangle_n$ and $[bv]_n$ the conversion to the natural number with the n-bit binary representation bv and the integer number with the n-bit two's complement representation bv, respectively.

For natural numbers x and n we denote by $bin_{\mathrm{n}}(x)$ the conversion to the n-bit binary representation of x. For an integer number x and a natural number n we denote by $two_{\mathrm{n}}(x)$ the conversion to the n-bit two's complement representation of x. For a bit vector bv and two natural numbers l, h such $l < h$ we denote by $bv[h : l]$ the sub-bit-vector of the bit vector from the bit l up to the bit h.

Records Record types are defined in the following way

$$\mathbb{T} \stackrel{\text{def}}{=} \{f_1 :: \mathbb{T}_1, f_2 :: \mathbb{T}_2, \ldots, f_n :: \mathbb{T}_n\},$$

where $f_1, f_2, \ldots, f_n$ are component(field) names of string type and $\mathbb{T}_1, \mathbb{T}_2, \ldots, \mathbb{T}_n$ are types of these components. The record type $\mathbb{T}$ yields the set of all elements $(f_1 = v_1, f_2 = v_2, \ldots, f_n = v_n)$ with $v_1 \in \mathbb{T}_1, v_2 \in \mathbb{T}_2, \ldots, v_n \in \mathbb{T}_n$. Let r be an instance of the record type $\mathbb{T}$. We write $r.f_i$ to access the component f_i of the record r. Updating a component f_i of the record r by a value v is abbreviated by $r[f_i := v]$.

CHAPTER 2

VAMP Assembly

In this section we describe the assembly language of the verified architecture microprocessor (VAMP). The VAMP is a formally verified DLX-like processor [BJK+03, BJK+05, DHP05, Dal06]. The VAMP implements the full DLX instruction set from [HP96], delayed branch mechanism with one delay slot, two privilege levels (user and system mode), single address translation in user mode and interrupt handling.

The VAMP assembly model provides a handy programming model by abstracting some features of the VAMP machine model [Tve09]. While describing the VAMP assembly semantics we follow the thesis [Tsy09].

2.1 Configurations

Configurations c_{ASM} of the VAMP assembly semantics are represented by record type C_{ASM}

◄ DEFINITION 2.1
VAMP Assembly Configuration

$$C_{\text{ASM}} \stackrel{\text{def}}{=} \{pc :: \mathbb{N}, dpc :: \mathbb{N}, gpr :: \mathbb{Z}^*, spr :: \mathbb{Z}^*, m :: \mathbb{N} \mapsto \mathbb{B}^8\}$$

that consists of five components:

- the program counter *pc* and the delayed program counter *dpc*. They implement the delayed branch mechanism with one delay slot (cf. [MP00]). The delayed program counter $c_{ASM}.dpc$ stores the address where the current instruction is fetched from and the program counter $c_{ASM}.pc$ stores the address of the next instruction.

- the general purpose register file *gpr* and the special purpose register file *spr* which are modeled as list of integers. $c_{ASM}.gpr[0]$ is always zero as it is required by the underlying computational model not discussed in the framework of this thesis (cf. [MP00]). Some SPRs are not used, the used SPRs are listed in Table 2.1.

Index	Alias	Usage
0	*sr*	Status register
1	*esr*	Exceptional status register
2	*eca*	Exceptional cause
3	*epc*	Exceptional program counter
4	*edpc*	Exceptional delayed program counter
5	*edata*	Exceptional data
9	*pto*	Page table origin
10	*ptl*	Page table length
11	*emode*	Exceptional mode
16	*mode*	Mode

Table 2.1: Special Purpose Registers

- the memory m is a mapping from memory addresses (naturals) to bytes. This memory is to be interpreted as byte-addressable memory of bytes. In [Tsy09] Tsyban defined the memory of VAMP assembly configurations as word-addressable. Sure, the corresponding changes must be done in the VAMP assembly semantics due to different representations of memory in Tsyban's dissertation and thesis, but these changes are simple and straightforward so that we omit them. Thus, we assume that Tsyban's VAMP assembly semantics works for the byte-addressable memory.

We define read and write functions over the assembly memory.

DEFINITION 2.2 ► Assembly Memory Read

$$m_{\text{word}}(a) \stackrel{\text{def}}{=} [m(a+3) \circ m(a+2) \circ m(a+1) \circ m(a)]_{32}$$

DEFINITION 2.3 ► Assembly Memory Write

$$\mathit{mem\text{-}update}_{\text{ASM}}(m, a, \mathit{data}) \stackrel{\text{def}}{=} m',$$

such that

$$m'(i) = \begin{cases} \mathit{two}_{32}(\mathit{data})[8 \cdot (i+1) \,:\, i] & \text{if } i \in [a, .., a+3] \\ m(i) & \text{otherwise} \end{cases}$$

Validity of a c_{ASM} assembly configuration is stated by means of the following predicate.

DEFINITION 2.4 ► Valid Assembly Configuration

$$\frac{\begin{array}{c} c_{ASM}.pc \in \mathbb{N}_{32} \qquad c_{ASM}.dpc \in \mathbb{N}_{32} \\ |c_{ASM}.gpr| = 32 \qquad |c_{ASM}.spr| = 32 \\ \forall i < 32.\ c_{ASM}.gpr[i] \in \mathbb{Z}_{32} \wedge c_{ASM}.spr[i] \in \mathbb{Z}_{32} \end{array}}{\mathit{valid}_{asm}(c_{ASM})}$$

2.2 Instructions

We do not model all instructions supported by VAMP assembly. We avoid the support for instructions which perform: (i) read and write of byte, half-word from and into

Load Word / Store Word	Arithmetics	Logical	Shift
`lw` *rd rs imm*	`addi` *rd rs imm*	`andi` *rd rs imm*	`slli` *rd rs sa*
`sw` *rd rs imm*	`add` *rd* rs_1 rs_2	`ori` *rd rs imm*	`srli` *rd rs sa*
	`subi` *rd rs imm*	`xori` *rd rs imm*	`srai` *rd rs sa*
	`sub` *rd* rs_1 rs_2	`lhgi` *rd imm*	`sll` *rd* rs_1 rs_2
		`and` *rd* rs_1 rs_2	`srl` *rd* rs_1 rs_2
		`or` *rd* rs_1 rs_2	`sra` *rd* rs_1 rs_2
		`xor` *rd* rs_1 rs_2	
		`lhg` *rd rs*	
Test		**Control**	**Special**
`clri` *rd*	`clr` *rd*	`beqz` *rs imm*	`movs2i` *rd sa*
`sgri` *rd rs imm*	`sgr` *rd* rs_1 rs_2	`bnez` *rs imm*	`movi2s` *rd sa*
`seqi` *rd rs imm*	`seq` *rd* rs_1 rs_2	`jr` *rs*	
`sgei` *rd rs imm*	`sge` *rd* rs_1 rs_2	`jalr` *rs*	
`slsi` *rd rs imm*	`sls` *rd* rs_1 rs_2	`j` *imm*	
`snei` *rd rs imm*	`sne` *rd* rs_1 rs_2	`jal` *imm*	
`slei` *rd rs imm*	`sle` *rd* rs_1 rs_2	`trap` *imm*	
`seti` *rd*	`set` *rd*	`rfe`	

Table 2.2: VAMP Assembly Instructions

memory, (ii) arithmetics operations causing an overflow interrupt. The instructions we want to be supported by VAMP assembly are presented in Table 2.2[1].

Instructions of the VAMP assembly machine are modeled by the inductive data type *Instr*. Each instruction is modeled by a separate constructor with parameters for register names and immediate constants. Parameters to the constructors of an instruction *instr* like source and destination registers or an immediate constant are obtained like $rs_1(instr)$, $rd(instr)$, $imm(instr)$, etc. We omit their formal definitions as they are straightforward and rather voluminous, the reader can consult the definitions in Section 3.2.5. in [Tsy09].

Since memory cells are modeled as integers we need to handle that by introducing two functions *int-to-instr* $:: \mathbb{Z} \mapsto$ *Instr*, converting integers to assembly instructions, and *instr-to-int* $::$ *Instr* $\mapsto \mathbb{Z}$, performing the conversion in other direction. As it is not possible to convert every integer to an instruction the predicate *decodable* $:: \mathbb{Z} \mapsto$ *bool* is introduced which tests whether the given integer could be converted to an instruction. We omit their definitions due to their simplicity, the reader can consult the definitions in Section 3.2.5 in [Tsy09].

The current instruction of the assembly configuration c_{ASM} is defined by the following function:

◀ DEFINITION 2.5
Current Instruction

$$instr(c_{ASM}) \stackrel{\text{def}}{=} \textit{int-to-instr}(c_{ASM}.m_{\text{word}}(c_{ASM}.dpc))$$

We introduce the predicates over instructions to determine what instruction group or group of instructions it corresponds to. It is done for each instruction constructor, e.g., *instr-trap*?, or for a group, like *instr-ls*? for all memory access instructions, or

[1] The complete list of VAMP assembly instructions is presented in Table 3.2 of [Tsy09]

instr-store? only for memory write access.

2.3 Transition Function

In order to be able to define the VAMP assembly transition function we need to elaborate on

- address translation that takes place during VAMP assembly instruction execution,
- interrupts that suspends the execution of a VAMP assembly machine,
- interrupt mechanism that starts every time when some interrupt raises(happens).

2.3.1 Address Translation

The semantics distinguishes two execution modes: *system*, which is defined as

DEFINITION 2.6 ► Is System Mode?

$$\frac{c_{ASM}.spr[mode] = 0}{sys\text{-}mode_{\text{ASM}}?(c_{ASM})}$$

and *user*, defined as

DEFINITION 2.7 ► Is User Mode?

$$\frac{c_{ASM}.spr[mode] = 1}{user\text{-}mode_{\text{ASM}}?(c_{ASM})}$$

A memory access is subject to address translation in user mode. Hence, the semantics of load/store instructions uses translated effective addresses and all instructions are fetched from the translated delayed program counter in user mode.

In user mode all addresses for instruction fetch as well as for load/store operations are treated as virtual ones and they are translated by the processor to physical ones with the help of special purpose registers *pto* and *ptl* (cf. Table 2.1). Virtual address space is divided into pages of size PAGE_SIZE = 2^{12} bytes = 2^{10} words. The binary representation of a virtual address *va* is split into two parts: (i) the most significant bits $bin_{32}(va)[31:12]$ give the *virtual page index*, and (ii) the less significant bits $bin_{32}(va)[11:0]$ give the address of the byte within the page (*page offset*). We introduce functions extracting natural number representations of virtual page index and page offset from a virtual address $va \in \mathbb{N}_{32}$, $px_{va} :: \mathbb{N}_{32} \mapsto \mathbb{N}_{20}$ and $bx_{va} :: \mathbb{N}_{32} \mapsto \mathbb{N}_{12}$, respectively.

DEFINITION 2.8 ► Virtual Address: Page Index and Byte Offset

$$\begin{aligned} px_{va}(va) &\stackrel{\text{def}}{=} va \;/\; \texttt{PAGE\_SIZE} \\ bx_{va}(va) &\stackrel{\text{def}}{=} va \;\textbf{mod}\; \texttt{PAGE\_SIZE} \end{aligned}$$

Based on the definitions of functions px_{va} and bx_{va} the following holds:

$$va = px_{va}(va) \cdot \texttt{PAGE\_SIZE} + bx_{va}(va)$$

The main data structure for address translation is a *page table* which resides in the processor memory. It is used to map, to translate, virtual page indices to physical page indices, where each translation is represented by a *page table entry* (pte). That is, the page table contains page table entries to support the address translation. The

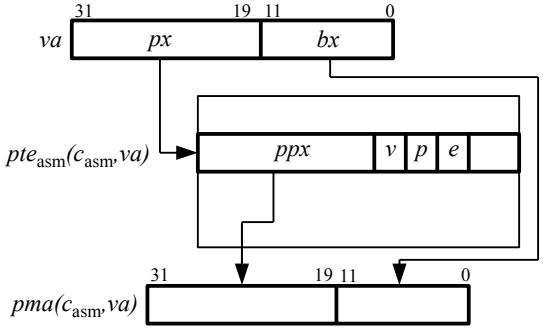


Figure 2.1: Address Translation

Component	Notation	Computation
ppx	$ppx_{PTE}(pte)$	$pte/\texttt{PAGE\_SIZE}$
valid bit	$v_{PTE}(pte)$	$pte/2^{11}$ **mod** 2
protected bit	$p_{PTE}(pte)$	$pte/2^{10}$ **mod** 2
executable bit	$e_{PTE}(pte)$	$pte/2^{9}$ **mod** 2

Table 2.3: Extracting components from a page table entry

page table origin register $c_{ASM}.spr[pto]$[2] and the virtual page index $px_{va}(va)$ specify one page table entry. Formally, the page table entry in an assembly configuration c_{ASM} for a virtual address va is:

$$pte_{\text{ASM}}(c_{ASM}, va) \stackrel{\text{def}}{=} c_{ASM}.m_{\text{word}}(c_{ASM}.spr[pto] \cdot \texttt{PAGE\_SIZE} + bx_{va}(va) \cdot 4)$$

◀ DEFINITION 2.9
Page Table Entry

Each page table entry is represented by a single word and it encodes: (i) physical page index, (ii) valid bit denoting whether the page resides in memory, (iii) protected bit indicating whether the page is allowed to be written, and (iv) executable bit denoting whether the page contains executable code. Table 2.3, taken from [Sta10], defines functions computing physical page index, valid bit, protected bit, and executable bit from the page table entry.

A physical page index combined with a byte index yields the complete *physical memory address* (Fig. 2.1 shows how a virtual address is translated to a physical one):

$$pma_{\text{ASM}}(c_{ASM}, va) \stackrel{\text{def}}{=} ppx_{PTE}(pte_{\text{ASM}}(c_{ASM}, va)) + bx_{va}(va)$$

◀ DEFINITION 2.10
Physical Memory Address

Exceptions

The page table length register $c_{ASM}.spr[ptl]$ is used to specify the amount of allocated virtual memory. In case the virtual address does not belong to the user memory address translation results in a *page table length exception* which is one of the supported

[2]The statement $c_{ASM}.spr[pto] < 2^{20}$ is always true and it follows from the VAMP assembly semantics (cf. [Tsy09])

i	Name	Meaning	Mask.	Ext.	Type
0	*reset*?	Reset	No	Yes	Abort
1	*ill*?	Illegal instruction	No	No	Abort
2	*imal*?∨*dmal*?	Instr. / data misalignment	No	No	Abort
3	*pff*?	Page fault on fetch	No	No	Repeat
4	*pfls*?	Page fault on load/store	No	No	Repeat
5	*trap*?	Trap / System call	No	No	Cont.
6	*ovf*?	Overflow	Yes	No	Cont.
≥ 12	eev_i?	Device interrupts	Yes	Yes	Cont.

Table 2.4: Interrupts of VAMP Assembly

interrupts by VAMP assembly which we examine in the next section. Formally, in configuration c_{ASM} the page table length exception for a virtual address *va* is:

DEFINITION 2.11 ► Page-Table Length Exception

$$\frac{c_{ASM}.spr[ptl] < px_{va}(va)}{ptlexcp_{\text{ASM}}?(c_{ASM})}$$

There are some situations when the translated address should not be used, either because invalid data was used for the translation, or the processor must forbid the attempted operation at this address. This happens if the memory which stores an instruction is not tagged as executable, or protected memory is accessed for writing, or even the page containing this address is not present in the physical memory. The next predicate tests whether a problem, namely interrupt, occurs during the address translation of a virtual address *va* in the assembly configuration c_{ASM}. Note that the flag *is-mw* indicates memory write access and the flag *is-fetch* indicates the instruction fetch:

DEFINITION 2.12 ► Virtual Address Translation Exception

$$\frac{\begin{array}{c} ptlexcp_{\text{ASM}}?(c_{ASM}, va) \ \vee \ v_{PTE}(pte_{\text{ASM}}(c_{ASM}, va)) = 0 \\ \vee \ is\text{-}fetch \ \wedge \ e_{PTE}(pte_{\text{ASM}}(c_{ASM}, va)) = 0 \\ \vee \ is\text{-}mw \ \wedge \ p_{PTE}(pte_{\text{ASM}}(c_{ASM}, va)) = 1 \end{array}}{translexcp_{\text{ASM}}?(c_{ASM}, va, is\text{-}mw, is\text{-}fetch)}$$

2.3.2 Interrupts

VAMP Assembly computations could be broken by interrupt signals numbered with indices from zero to thirty one. Table 2.4 depicts interrupts supported by the VAMP assembly model and they are classified according to the following criteria: (i) maskable or not maskable, (ii) internal or external, and (iii) of repeat, continue, or abort type. Maskable interrupts can be ignored under software control. If an interrupt signal arrives during the execution of some instruction *i* and it is of repeat type, then the instruction *i* is repeated when the program execution is resumed. If the interrupt is a continue interrupt then the instruction that follows *i* in the program is executed after the interrupt handling. In the remaining case the program execution is aborted.

The *instruction misalignment* interrupt occurs if the delayed program counter is not word aligned.

DEFINITION 2.13 ► Instruction misalignment

$$\frac{c_{ASM}.dpc \ \mathbf{mod} \ 4 \neq 0}{imal?_{\text{ASM}}(c_{ASM})}$$

The *page fault on fetch* happens in the user mode whenever the address translation of the current instruction to be fetched cannot be done:

$$\frac{\neg imal?_{\mathrm{ASM}}(c_{ASM}) \qquad \neg sys\text{-}mode_{\mathrm{ASM}}?(c_{ASM}) \qquad translexcp_{\mathrm{ASM}}?(c_{ASM}, c_{ASM}.dpc, \mathsf{F}, \mathsf{T})}{pff?_{\mathrm{ASM}}(c_{ASM})}$$

◀ DEFINITION 2.14
Page Fault On Fetch

The *illegal instruction* interrupt is triggered if the current instruction cannot be decoded or it is an unprivileged one. An instruction is unprivileged in case the instruction is one of the following instructions: `movi2s`, `movs2i`, or `rfe`, and it is executed in the user mode.

$$\frac{\begin{array}{c}\neg imal?_{\mathrm{ASM}}(c_{ASM}) \qquad \neg translexcp_{\mathrm{ASM}}?(c_{ASM}, c_{ASM}.dpc, \mathsf{F}, \mathsf{T}) \\ (\neg decodable(c_{ASM}.m_{\mathrm{word}}(c_{ASM}.dpc)) \vee \neg sys\text{-}mode_{\mathrm{ASM}}?(c_{ASM}) \longrightarrow \\ (\neg instr\text{-}rfe?(c_{ASM}) \vee \neg instr\text{-}movi2s?(c_{ASM}) \vee \neg instr\text{-}movsi2?(c_{ASM})))\end{array}}{ill?_{\mathrm{ASM}}(c_{ASM})}$$

◀ DEFINITION 2.15
Illegal Instruction

The *data misalignment* exceptions is raised if the effective address of the current instruction accessing memory is not word aligned. The function *ea* computing the effective address will be defined in Section 2.3.4.

$$\frac{\begin{array}{cc}\neg imal?_{\mathrm{ASM}}(c_{ASM}) & \neg translexcp_{\mathrm{ASM}}?(c_{ASM}, ea(c_{ASM}), instr\text{-}sw?(c_{ASM}), \mathsf{F}) \\ \neg ill?_{\mathrm{ASM}}(c_{ASM}) & \neg(ls\text{-}w?(instr(c_{ASM})) \longrightarrow ea(c_{ASM}) \bmod 4 = 0)\end{array}}{dmal?_{\mathrm{ASM}}(c_{ASM})}$$

◀ DEFINITION 2.16
Data misalignment

The *page fault on load/store* is similar to the page fault on fetch, but here the effective address of the load/store instruction is examined:

$$\frac{\begin{array}{ccc}\neg dmal?_{\mathrm{ASM}}(c_{ASM}) & \neg sys\text{-}mode_{\mathrm{ASM}}?(c_{ASM}) & instr\text{-}mem?(c_{ASM}) \\ \multicolumn{3}{c}{translexcp_{\mathrm{ASM}}?(c_{ASM}, ea(c_{ASM}), instr\text{-}sw?(c_{ASM}), \mathsf{F})}\end{array}}{pfls?_{\mathrm{ASM}}(c_{ASM})}$$

◀ DEFINITION 2.17
Page Fault On Load/Store

The *trap* interrupt occurs whenever the instruction `trap` is executed.

$$\frac{\begin{array}{cc}\neg imal?_{\mathrm{ASM}}(c_{ASM}) & \neg translexcp_{\mathrm{ASM}}?(c_{ASM}, c_{ASM}.dpc, \mathsf{F}, \mathsf{T}) \\ \multicolumn{2}{c}{\neg ill?_{\mathrm{ASM}}(c_{ASM}) \qquad instr\text{-}trap?(c_{ASM})}\end{array}}{trap?_{\mathrm{ASM}}(c_{ASM})}$$

◀ DEFINITION 2.18
Trap Interrupt

Remark: Since we do not use instructions which could cause an overflow we just avoid overflow interrupts.

2.3.3 Interrupt Mechanism

So far we defined predicates indicating particular interrupts that may occur during the execution in an assembly configuration c_{ASM}. At this point we want to stress our assembly model is used in the environment without any external devices. That means, we consider a single external interrupt like *reset*. Here we define the *JISR* (Jump to Interrupt Service Routine) signal indicating that there is an interrupt to be handled by the interrupt mechanism.

Interrupt signals raised in the configuration c_{ASM} are collected in the *cause* function $ca(c_{ASM})$. [3]

DEFINITION 2.19 ► Cause Vector

$$\begin{aligned} ca(c_{ASM}) \stackrel{\text{def}}{=} & \quad 2^1 \cdot bool2nat(ill?_{\text{ASM}}(c_{ASM})) \\ + & \quad 2^2 \cdot bool2nat(imal?_{\text{ASM}}(c_{ASM}) \vee dmal?_{\text{ASM}}(c_{ASM})) \\ + & \quad 2^3 \cdot bool2nat(pff?_{\text{ASM}}(c_{ASM})) \\ + & \quad 2^4 \cdot bool2nat(pfls?_{\text{ASM}}(c_{ASM})) \\ + & \quad 2^5 \cdot bool2nat(trap?_{\text{ASM}}(c_{ASM})). \end{aligned}$$

At this place we introduce the predicate $repeat?(c_{ASM})$ indicating that an interrupt is of repeat type.

DEFINITION 2.20 ► Is Interrupt of Repeat Type?

$$\frac{\begin{array}{c} pff?_{\text{ASM}}(c_{ASM}) \vee pfls?_{\text{ASM}}(c_{ASM}) \\ \neg(ill?_{\text{ASM}}(c_{ASM}) \vee imal?_{\text{ASM}}(c_{ASM}) \vee dmal?_{\text{ASM}}(c_{ASM})) \end{array}}{repeat?(c_{ASM})}$$

Since we do not consider maskable interrupts the definition of a function which returns the *masked cause vector* is equal to the definition of the function computing *cause vector*.

DEFINITION 2.21 ► Masked Cause Vector

$$mca(c_{ASM}) \stackrel{\text{def}}{=} ca(c_{ASM})$$

If $mca(c_{ASM})$ is greater than zero, then the JISR signal $jisr(c_{ASM})$ is activated.

DEFINITION 2.22 ► JISR Signal

$$\frac{0 < mca(c_{ASM})}{jisr(c_{ASM})}$$

Jump To Interrupt Service Routine Now we define the function

$$exec_{\text{jisr}} :: C_{\text{ASM}} \mapsto C_{\text{ASM}}$$

which prepares the assembly machine c_{ASM} for the execution of an *interrupt service routine* (ISR) that resides at address 0 in memory.

DEFINITION 2.23 ► JISR Semantics

$$exec_{\text{jisr}}(c_{ASM}) \stackrel{\text{def}}{=} c'_{ASM}$$

The function $exec_{\text{jisr}}(c_{ASM})$ yields an updated VAMP assembly configuration c'_{ASM}. The program counters of c'_{ASM} point to the ISR start address.

$$\begin{aligned} c'_{ASM}.dpc &= 0 \\ c'_{ASM}.pc &= 4 \end{aligned}$$

The program counters as well as the status and mode registers are saved in the corresponding exceptional registers:

$$\begin{aligned} c'_{ASM}.spr.[edpc] &= c_{ASM}.dpc \\ c'_{ASM}.spr[epc] &= c_{ASM}.pc \\ c'_{ASM}.spr[esr] &= c_{ASM}.spr[sr] \\ c'_{ASM}.spr[emode] &= c_{ASM}.spr[mode] \end{aligned}$$

[3] Since we do not consider maskable interrupts, like overflow and external interrupts from devices, we omitted the second parameter to the function *ca* representing external interrupts vector. For the complete definition refer to [Tsy09]

The exceptional cause register *eca* is set to the masked interrupt cause:

$$c'_{ASM}.spr[eca] \;=\; mca(c_{ASM}, eev)$$

The exceptional data register *edata* has the following value as it is defined below:

- the delayed program counter in case of a page fault interrupt on fetch,
- the load/store effective address in case of an interrupt on load/store,
- the immediate constant in case of a trap instruction, and
- zero in all other cases.

Formally,

$$c'_{ASM}.spr[edata] \stackrel{\text{def}}{=} edata(c_{ASM}) \stackrel{\text{def}}{=}$$

$$\begin{cases} c_{ASM}.dpc & \text{if } imal?_{\text{ASM}}(c_{ASM}) \vee pff?_{\text{ASM}}(c_{ASM}) \\ ea(c_{ASM}) & \text{if } instr\text{-}mem?(c_{ASM}) \\ i2n(imm(instr(c_{ASM}))) & \text{if } instr\text{-}trap?(c_{ASM}) \\ 0 & \text{otherwise} \end{cases}$$

The mode and status registers are set to zero, i.e. the assembly machine c'_{ASM} operates in system mode and all interrupts are masked. Formally,

$$\begin{aligned} c'_{ASM}.spr[mode] &= 0 \\ c'_{ASM}.spr[sr] &= 0. \end{aligned}$$

Return From Exception The last instruction to be executed in the ISR must be `rfe` (*return from exception*). This instruction restores the content of saved registers at JISR from exceptional registers. Formally,

◀ DEFINITION 2.24
RFE Semantics

$$\begin{aligned} c'_{ASM}.dpc &= c_{ASM}.spr[edpc], \\ c'_{ASM}.pc &= c_{ASM}.spr[epc], \\ c'_{ASM}.spr[sr] &= c_{ASM}.spr[esr], \\ c'_{ASM}.spr[mode] &= c_{ASM}.spr[emode]. \end{aligned}$$

2.3.4 Transitions Without Interrupts

The execution semantics of VAMP assembly without interrupts is given by the transition function $\delta^{woi}_{\text{ASM}} :: C_{\text{ASM}} \mapsto C_{\text{ASM}}$ which yields for a configuration c_{ASM} the next state $c'_{ASM} = \delta^{woi}_{\text{ASM}}(c_{ASM}) = exec_{instr}(c_{ASM}, instr(c_{ASM}))$, where the function

$$exec_{instr} :: C_{\text{ASM}} \times Instr \mapsto C_{\text{ASM}}$$

◀ DEFINITION 2.25
Instruction Execution Without Interrupt

defines the effect of a single instruction execution on the assembly configuration (cf. Sec 3.2.6 in [Tsy09]).

The definition of $exec_{instr}$ splits cases depending on the instruction to be executed. We will specify the case for executing the load word instruction `lw` in system mode as well as in user mode. The semantics for the remaining instructions is presented in [Tsy09].

The effective address $ea(c_{ASM})$ of load/store instructions is computed as the sum of the content of the source register one ($c_{ASM}.gpr[rs_1(c_{ASM})]$) and the immediate constant ($imm(c_{ASM})$).

DEFINITION 2.26 ► Effective Address

$$ea(c_{ASM}) \stackrel{\text{def}}{=} (c_{ASM}.gpr[rs_1(c_{ASM})] + imm(c_{ASM})) \textbf{ mod } 2^{32}.$$

The effect of the execution of the load word instruction `lw` is that the destination register $c_{ASM}.gpr[rd(c_{ASM})]$ is updated with the content of memory cell at the translated (or non-translated) effective address $ea_{mode}(c_{ASM})$. Also, program counters are updated correspondingly.

DEFINITION 2.27 ► Load-Word Instruction Semantics

$$exec_{instr}(c_{ASM}, \texttt{lw } rd\ rs\ imm) = c'_{ASM}$$

where the updated VAMP assembly configuration c'_{ASM} differs from the original one c_{ASM} in the following components:

$$\begin{aligned} c'_{ASM}.gpr[rd(c_{ASM})] &= c_{ASM}.m_{\text{word}}(ea_{mode}(c_{ASM})) \\ c'_{ASM}.dpc &= c_{ASM}.pc \\ c'_{ASM}.pc &= (c_{ASM}.pc + 4) \textbf{ mod } 2^{32} \end{aligned}$$

where the memory address ea_{mode} depends on the mode the assembly machine c_{ASM} is running in:

DEFINITION 2.28 ► Translated Effective Address

$$ea_{mode}(c_{ASM}) = \begin{cases} ea(c_{ASM}) & \text{if } sys\text{-}mode_{\text{ASM}}?(c_{ASM}) \\ pma_{\text{ASM}}(c_{ASM}, ea(c_{ASM})) & \text{otherwise} \end{cases}$$

2.3.5 Transitions

The transition system of VAMP assembly is given by the next-step function $\delta_{\text{ASM}} :: C_{\text{ASM}} \mapsto C_{\text{ASM}}$. The definition of the transition function δ_{ASM} splits cases depending on interrupts. In case there is no interrupt we apply the semantics of the assembly next step function without interrupts $\delta^{woi}_{\text{ASM}}$. If there is an interrupt, then we make cases distinction on type of the interrupt.

$$\delta_{\text{ASM}}(c_{ASM}) = \begin{cases} exec_{\text{jisr}}(c_{ASM}) & \text{if } jisr(c_{ASM}) \wedge repeat?(c_{ASM}) \\ exec_{\text{jisr}}(\delta^{woi}_{\text{ASM}}(c_{ASM})) & \text{if } jisr(c_{ASM}) \wedge \neg repeat?(c_{ASM}) \\ \delta^{woi}_{\text{ASM}}(c_{ASM}) & \text{otherwise} \end{cases}$$

Several assembly steps are done by the function δ^n_{ASM}, which is defined by induction on n:

$$\begin{aligned} \delta^0_{\text{ASM}}(c_{\text{ASM}}) &= c_{\text{ASM}}, \\ \delta^n_{\text{ASM}}(c_{\text{ASM}}) &= \delta^{n-1}_{\text{ASM}}(\delta_{\text{ASM}}(c_{\text{ASM}})). \end{aligned}$$

CHAPTER 3

VAMP Macro Assembly

In this section we present the high level VAMP assembly model, called VAMP macro assembly or μ_{ASM}. This μ_{ASM} model is an abstracted VAMP assembly. The reason of the abstraction is to have nice high-level control flow such that instead of using relative and absolute offsets for branch and jump instructions in VAMP assembly, we use labels and procedure names in μ_{ASM}. Modeling procedure calls requires the introduction of a stack in μ_{ASM}, where the stack is modeled explicitly. Having such a high level control flow simplifies the task of interfacing assembly with the model of a high level programming language like C and we indeed want to have such a nice interface which we present in Chapter 5. Since we have a new control flow in μ_{ASM} we need to introduce new instructions which do: jump to labels, call procedures, etc. The VAMP macro assembly language does not support the complete set of macros that is typically supported by a macro assembly language as we do not need them for this thesis. The VAMP macro assembly language can be easily extended with missing macros which implement loops, if-then-else statements, etc.

The way we want to proceed in this section is: (i) we present the semantics of the μ_{ASM} model, (ii) we examine the macro assembling of a μ_{ASM} program to VAMP assembly, and (iii) we state the correctness of the μ_{ASM} assembler.

3.1 Semantics

We start the description of the μ_{ASM} computation model with the introduction of instructions modeled in μ_{ASM}.

3.1.1 Instructions

The μ_{ASM} instructions are modeled by inductive type $\mathbb{S}_{\mu_{\text{ASM}}}$. It is the union of constructors of the new μ_{ASM} instructions introduced later on and the VAMP assembly instruction set modeled by inductive type *Instr* (introduced in Section 3.1.1 on page 17). Since we have a high level control flow in μ_{ASM} in comparison to the VAMP as-

sembly we exclude all control instructions from *Instr*. The set of control instructions is presented in Table 2.2. So that we obtain a reduced VAMP assembly instructions set $Instr_{cut4\mu_{\text{ASM}}}$ defined as:

DEFINITION 3.1 ► Supported VAMP assembly instructions in μ_{ASM}

$$Instr_{cut4\mu_{\text{ASM}}} \stackrel{\text{def}}{=} Instr \setminus \{\text{control instructions}\}$$

As equivalent to VAMP assembly jump and branch instructions we introduced two new instructions changing the control flow within a procedure: local absolute jump `goto` and local absolute conditional jump `ifnez` *r* `goto`. There is only one possibility to have a non-local jump in μ_{ASM}: a procedure call which is implemented by a new introduced instruction `call`. To read and to update parameters of a procedure new instructions `lparam` and `sparam` are introduced, respectively. To return from the procedure call we use a new instruction `ret`. Besides the instructions mentioned above there are another two new instructions: `push` and `pop` putting and taking away a word on and from stack, respectively. Below we list all instructions we model in μ_{ASM}:

- $r, i \in \mathbb{N} \longrightarrow$ `lparam` $r\ i \in \mathbb{S}_{\mu_{\text{ASM}}} \wedge$ `sparam` $r\ i \in \mathbb{S}_{\mu_{\text{ASM}}}$
- $r \in \mathbb{N} \longrightarrow$ `push` $r \in \mathbb{S}_{\mu_{\text{ASM}}} \wedge$ `pop` $r \in \mathbb{S}_{\mu_{\text{ASM}}}$
- $P \in P_{name} \longrightarrow$ `call` $P \in \mathbb{S}_{\mu_{\text{ASM}}}$, where $P_{name} \in String$
- $l \in \mathbb{N} \longrightarrow$ `goto` $l \in \mathbb{S}_{\mu_{\text{ASM}}}$
- $r, l \in \mathbb{N} \longrightarrow$ `ifnez` r `goto` $l \in \mathbb{S}_{\mu_{\text{ASM}}}$
- $Instr_{cut4\mu_{\text{ASM}}} \in \mathbb{S}_{\mu_{\text{ASM}}}$

We introduced the set of instructions supported by μ_{ASM}. At this point we want to stress that some of the introduced μ_{ASM} instructions like `lparam` and `sparam` are present in order to have a clear separation between the implementation and semantics. These two instructions can be implemented by supported VAMP assembly instructions like `sw` and `lw` in μ_{ASM}. If we had implemented them using `sw` and `lw` in μ_{ASM}, then we would expose the stack layout. That is, exposing the implementation details that could differ depending on the underlying architecture and the implementation of a macro assembler itself. Definitely we do not want to expose the stack layout in the μ_{ASM} semantics as it is only assembler(compiler) relevant.

Before we present the semantics of μ_{ASM} instructions we first examine how a μ_{ASM} program is defined and the definition of a μ_{ASM} configuration .

3.1.2 Program

A μ_{ASM} program $\pi_{\mu_{\text{ASM}}}$ is represented by a partial mapping of procedure names to procedure declarations. Also, such a mapping is called *procedure table*.

DEFINITION 3.2 ► μ_{ASM} Program

$$Prog_{\mu_{\text{ASM}}} \stackrel{\text{def}}{=} P_{name} \rightharpoonup ProcT$$

μ_{ASM} procedure declarations are represented by record type *ProcT*. This record represents a declared procedure in a μ_{ASM} program and it comprises three components: (i) the number of input arguments to the procedure (*npar*), (ii) the procedure body

(*body*), if the procedure is defined outside the μ_{ASM} program, then it is declared as `extern`, and (iii) the list of register indices (*uses*), this list contains those GPR indices whose values should be saved on the procedure entry and restored at return from this procedure [1].

◀ DEFINITION 3.3
μ_{ASM} Procedure Declaration

$$ProcT \stackrel{\text{def}}{=} (npar :: \mathbb{N}, body :: \mathbb{S}^*_{\mu_{\text{ASM}}} \cup \{\texttt{extern}\},\ uses :: \mathbb{N}^*)$$

As you might have noticed that some procedures can be declared as external ones. In the thesis we consider a single type of external procedures, namely external C procedure calls.

We introduce the predicate $main?_{\mu_{\text{ASM}}} :: P_{name} \mapsto bool$ indicating whether a given μ_{ASM} procedure name is main one in the μ_{ASM} program. In any μ_{ASM} program there is a single main procedure.

◀ DEFINITION 3.4
Is main μ_{ASM} procedure?

$$\frac{P = \text{"\_start"}}{main?_{\mu_{\text{ASM}}}(P)}$$

3.1.3 Configurations

A μ_{ASM} machine state c_μ is modeled by record type $C_{\mu_{\text{ASM}}}$ which comprises a byte addressable memory ($\mathcal{M}$), general and special register files (*gpr* and *spr*), and an abstract stack (*stack*) that can be in one of two states depending on whether the stack is created or not.

◀ DEFINITION 3.5
μ_{ASM} Configuration

$$C_{\mu_{\text{ASM}}} = (\mathcal{M} :: \mathbb{N} \mapsto \mathbb{B}^8,\ gpr :: \mathbb{Z}^*,\ spr :: \mathbb{Z}^*,\ stack :: frame^*_\mu \cup frame^{no}_\mu)$$

The components $\mathcal{M}$, *gpr* and *spr* have the same purpose as in the VAMP assembly assembly configuration, the only new component is *stack* which is represented either by a list of abstract μ_{ASM} stack frames or by no-stack. If the stack exists, i.e., represented as a list of stack frames, then each stack frame from the stack corresponds to a call to a procedure which has not yet returned, i.e. not terminated with a return instruction. A stack frame contains information which describes a procedure execution context in terms of the procedure name (p) and the location counter (*loc*) indicating the instruction that should be executed as next one in this procedure. So stack frames are added and removed only at procedure calls' entries and returns. The μ_{ASM} stack frame is modeled by record type $frame_\mu$:

◀ DEFINITION 3.6
μ_{ASM} Stack Frame

$$frame_\mu \stackrel{\text{def}}{=} (p :: P_{name}, loc :: \mathbb{N}, pars :: \mathbb{Z}^*, saved :: \mathbb{Z}^*, lifo :: \mathbb{Z}^*)$$

which comprises the following components:

- p is the name of a procedure the stack frame corresponds to,
- *loc* is the location of the current instruction in the body of a procedure,
- *pars* is a list of parameters passed to a procedure,
- *saved* is a list of values. It serves as a container for data to be preserved during procedure execution. It is used to store values of those GPRs whose indices are present in the *uses* component of the declaration of a procedure.

[1] Originally the name of *uses* comes from Microsoft Macro Assembler (MASM) and it is used as described (cf. [Cor])

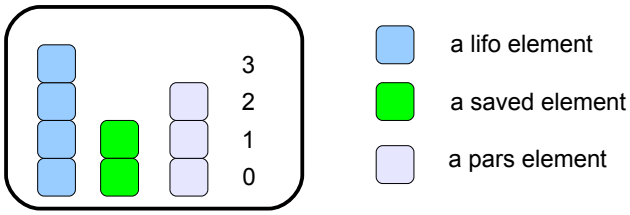


Figure 3.1: μ_{ASM} Stack Frame (Without Control Components)

- $lifo$ is a list of integers representing a *last in, first out* (LIFO) data structure. "By definition, in a LIFO structured linear list, elements can be added or taken off from only one end, called the "top"" ([Lip86]). The last element to be placed on top is also the first to be taken off the top. The $lifo$ component is used to store temporaries and for passing input parameters to a call to a procedure.

No-stack is modeled by record $frame_{\mu}^{no}$ that comprises two components p and loc modeling the control flow.

DEFINITION 3.7 ► μ_{ASM} No-Stack

$$frame_{\mu}^{no} \stackrel{\text{def}}{=} (p :: P_{name}, loc :: \mathbb{N})$$

We introduce the function $top :: C_{\mu_{\text{ASM}}} \mapsto \mathbb{N}$ returns the index of the top-most stack frame in the configuration c_{μ}:

DEFINITION 3.8 ► Top Most Stack Frame Index

$$top(c_{\mu}) \stackrel{\text{def}}{=} |c_{\mu}.stack| - 1$$

The predicate is_stack indicates whether the stack of a given μ_{ASM} configuration exists or not:

DEFINITION 3.9 ► Is Stack Set Up?

$$\frac{c_{\mu}.stack \notin frame_{\mu}^{no}}{is\_stack(c_{\mu})}$$

The predicate $last?_{\mu} :: \mathbb{N} \times P_{name} \times Prog_{\mu_{\text{ASM}}} \mapsto \mathbb{B}$ indicates whether the μ_{ASM} statement indicated by a given location loc is the last statement in the body of the μ_{ASM} procedure P.

DEFINITION 3.10 ► Is Last Statement?

$$\frac{|\pi_{\mu_{\text{ASM}}}(p).body| - 1 = loc}{last?_{\mu}(loc, p, \pi_{\mu_{\text{ASM}}})}$$

We introduce a few shorthands for the top stack frame and no-stack as well as for components of the top-most stack frame and no-stack of the μ_{ASM} configuration c_{μ}.

- $c_{\mu}.stack_{top} = \begin{cases} c_{\mu}.stack & \text{if } \neg is\_stack(c_{\mu}) \\ c_{\mu}.stack[top(c_{\mu})] & \text{otherwise} \end{cases}$
- $c_{\mu}.p_{top} = c_{\mu}.stack_{top}.p$
- $c_{\mu}.loc_{top} = c_{\mu}.stack_{top}.loc$
- $c_{\mu}.pars_{top} = c_{\mu}.stack_{top}.pars$

- $c_\mu.saved_{top} = c_\mu.stack_{top}.saved$
- $c_\mu.lifo_{top} = c_\mu.stack_{top}.lifo$

Since we cannot return from the first stack frame in the μ_{ASM} semantics (at least it is not modeled) it is required that parameters and saved components of the first stack frame of the stack of a given μ_{ASM} machine c_μ are empty. This and the other well-formedness requirements over μ_{ASM} configurations are stated by means of the following predicate:

◀ DEFINITION 3.11
Valid μ_{ASM} Configuration

$$\frac{\begin{array}{c}|c_\mu.gpr| = 32 \qquad |c_\mu.spr| = 32 \\ \forall i < 32.\ c_\mu.gpr[i] \in \mathbb{Z}_{32} \wedge c_\mu.spr[i] \in \mathbb{Z}_{32} \\ is\_stack(c_\mu) \longrightarrow c_\mu.stack[0].pars = [] \wedge c_\mu.stack[0].saved = []\end{array}}{valid_{\mu_{\text{ASM}}}(c_\mu)}$$

We introduce the predicate $are\_sregs\_set\_up$ which indicates that the stack registers (sp and bp from Table 3.1) of a given VAMP assembly machine c_{ASM} are initialized. It is the case if and only if the content of stack registers equals the constant `STACK_BASE_ADR` which denotes the stack base address.

◀ DEFINITION 3.12
Are Stack Registers Set Up?

$$\frac{c_{ASM}.gpr[sp] = c_{ASM}.gpr[bp] = \texttt{STACK\_BASE\_ADR}}{are\_sregs\_set\_up(c_{ASM})}$$

We indicate the emptiness of the stack of a given μ_{ASM} machine c_μ if its stack has a single stack frame whose lifo is empty.

◀ DEFINITION 3.13
Is Empty Stack?

$$\frac{top(c_\mu) = 0 \qquad c_\mu.stack[0].lifo = [] \qquad is\_stack(c_\mu)}{is\_empty\_stack(c_\mu)}$$

The current instruction to be executed by a given μ_{ASM} machine running a given *C-IL* program is returned by the function defined below:

◀ DEFINITION 3.14
Current Instruction

$$instr_{curr} :: C_{\mu_{\text{ASM}}} \times Prog_{\mu_{\text{ASM}}} \mapsto \mathbb{S}_{\mu_{\text{ASM}}}$$

$$instr_{curr}(c_\mu, \pi_{\mu_{\text{ASM}}}) \stackrel{\text{def}}{=} \pi_{\mu_{\text{ASM}}}(c_\mu.p_{top}).body[c_\mu.loc_{top}]$$

Further, we define a few functions updating the state of a given μ_{ASM} machine. The function $inc_{loc} :: C_{\mu_{\text{ASM}}} \mapsto C_{\mu_{\text{ASM}}}$ increments the location counter of the top stack frame or no-stack of a given μ_{ASM} configuration c_μ:

$$inc_{loc}(c_\mu) \stackrel{\text{def}}{=}$$

$$\begin{cases} c_\mu[stack := c_\mu.stack[loc := c_\mu.loc_{top} + 1]] & \text{if } \neg is\_stack(c_\mu) \\ c_\mu[stack := \mathbf{hd}(c_\mu.stack)[loc := c_\mu.loc_{top} + 1]] \circ \mathbf{tl}(c_\mu.stack)] & \text{otherwise} \end{cases}$$

The function $set_{loc} :: C_{\mu_{\text{ASM}}} \times \mathbb{N} \mapsto C_{\mu_{\text{ASM}}}$ sets the location of the top stack frame of a given configuration c_μ to a given location l:

$$set_{loc}(c_\mu, l) \stackrel{\text{def}}{=}$$

$$\begin{cases} c_\mu[stack := c_\mu.stack[loc := l]] & \text{if } \neg is\_stack(c_\mu) \\ c_\mu[stack := \mathbf{hd}(c_\mu.stack)[loc := l] \circ \mathbf{tl}(c_\mu.stack)] & \text{otherwise} \end{cases}$$

The function $drop_{frame} :: C_{\mu_{\text{ASM}}} \mapsto C_{\mu_{\text{ASM}}}$ removes the top stack frame in a given μ_{ASM} configuration c_μ:

$$drop_{frame}(c_\mu) \stackrel{\text{def}}{=} c_\mu[stack := \mathbf{tl}(c_\mu.stack)].$$

The function $set_{pars} :: C_{\mu_{\text{ASM}}} \times \mathbb{N} \times \mathbb{N} \mapsto C_{\mu_{\text{ASM}}}$ updates the i-th parameter of the top stack frame of a given μ_{ASM} machine c_μ by a given value v:

$$set_{pars}(c_\mu, i, v) \stackrel{\text{def}}{=}$$

$$c_\mu[stack := \mathbf{hd}(c_\mu.stack)[pars := pars_{top}[i := v]] \circ \mathbf{tl}(c_\mu.stack)].$$

The function $put_{lifo} :: C_{\mu_{\text{ASM}}} \times \mathbb{N}^* \mapsto C_{\mu_{\text{ASM}}}$ puts on top of the lifo of the top stack frame of a given configuration c_μ a given list of integer values l:

$$put_{lifo}(c_\mu, l) \stackrel{\text{def}}{=}$$

$$c_\mu[stack := \mathbf{hd}(c_\mu.stack)[lifo := l \circ c_\mu.lifo_{top}] \circ \mathbf{tl}(c_\mu.stack)].$$

The function $drop_{lifo} :: C_{\mu_{\text{ASM}}} \mapsto C_{\mu_{\text{ASM}}}$ removes the n top elements from the lifo of the top stack frame of a given configuration c_μ:

$$drop_{lifo}(c_\mu, n) \stackrel{\text{def}}{=}$$

$$c_\mu[stack := \mathbf{hd}(c_\mu.stack)[lifo := \mathbf{drop}(c_\mu.lifo_{top}, n)] \circ \mathbf{tl}(c_\mu.stack)].$$

The function $setup\_sregs :: C_{\mu_{\text{ASM}}} \mapsto C_{\mu_{\text{ASM}}}$ sets up the stack registers to the stack base address:

DEFINITION 3.15 ►
Set Up Stack Registers

$$setup\_sregs(c_\mu) \stackrel{\text{def}}{=} c_\mu[gpr := c_\mu.gpr\begin{bmatrix} bp & := \texttt{STACK\_BASE\_ADR}, \\ sp & := \texttt{STACK\_BASE\_ADR} \end{bmatrix}]$$

In order to examine the semantics of μ_{ASM} instructions there is still a convention to be discussed, called *compiler calling convention*.

3.1.4 Compiler Calling Convention

The *compiler calling convention* describes the interface between the caller and the callee, namely: (i) how parameters are passed from the caller to the callee (placed in registers, pushed on the stack, or a mix of both), (ii) how a result is returned from the callee to the caller, and (iii) who is responsible for cleaning up the stack from parameters after a procedure call. The μ_{ASM} compiler calling convention consists of the following rules:

1. The first four input parameters are passed through GPRs $i_0, \ldots i_3$ (Table 3.1)

Alias	Index	Usage
0	*zero*	always zero
1 ... 13	$t_0 \ldots t_{12}$	temporary
14	*rv*	return value
15 ... 18	$i_0 \ldots i_3$	input arguments
19 ... 28	$t_{13} \ldots t_{22}$	temporary
29	*sp*	stack pointer
30	*bp*	stack frame base pointer
31	t_{23}	temporary

Table 3.1: Special Usage of GPRs

before the call instruction is executed,

- Caller does not expect to have the same values in those registers after callee returns.

2. The rest of input parameters, if existent at all, is passed on the lifo of the caller's stack frame in right-to-left order before the call instruction is executed, i.e. a parameter with the lowest index is on the top of the lifo and a parameter with the highest index is put first on the lifo,

   - Moreover, we reserve a space on the lifo for parameters passed in GPRs. It is done for two main reasons: (i) if callee wants to call a procedure, then it should save values of these parameters to the reserved space on the lifo, and (ii) if callee ran out of available GPRs.

3. Callee passes a return value to caller in GPR *rv* (Table 3.1),

4. Callee is responsible for cleaning up the stack from parameters.

In the thesis we refer to this compiler calling convention as *CCL* and to rules of *CCL* as *CCL*.1, *CCL*.2, *CCL*.3 and *CCL*.4.

3.1.5 Transition Function

The transition function

$$\delta_{\mu_{\text{ASM}}} :: C_{\mu_{\text{ASM}}} \times Prog_{\mu_{\text{ASM}}} \mapsto C_{\mu_{\text{ASM}}}\bot$$

of the μ_{ASM} semantics maps, with respect to a program $\pi_{\mu_{\text{ASM}}}$, a configuration c_μ to its successor c'_μ, such that $\lfloor c'_\mu \rfloor = \delta_{\mu_{\text{ASM}}}(c_\mu, \pi_{\mu_{\text{ASM}}})$ or, in case of an error to $\bot$ (i.e. the execution got stuck). At some places we write

$$\pi_{\mu_{\text{ASM}}} \vdash c_\mu \rightarrow_{\mu_{\text{ASM}}} \lfloor c'_\mu \rfloor$$

to denote $\delta_{\mu_{\text{ASM}}}(c_\mu, \pi_{\mu_{\text{ASM}}}) = \lfloor c'_\mu \rfloor$.

The transition function is defined by induction on the μ_{ASM} instruction constructor type of the current statement to be executed. We split the μ_{ASM} instructions by their functionality into four groups: (i) VAMP assembly, (ii) goto, (iii) load and store parameters, and (iv) stack - instructions. For instructions from the group (i) we apply the semantics of the assembly next step function without interrupt $exec_{instr}$ which was introduced in Section 2.3.4.

Vamp Assembly Instructions

If the current instruction to be executed by the μ_{ASM} machine is of VAMP assembly type and it does not modify the content of the stack pointer and base pointer registers (i.e. the return destination register of this instruction is neither sp nor bp (cf. Table 3.1)), and it does not read stack registers when the stack of the μ_{ASM} machine is set up, then we simply apply the VAMP assembly semantics over a VAMP assembly machine constructed from the μ_{ASM} machine and map changes done at VAMP assembly level back to μ_{ASM} level. Program counters of the constructed VAMP assembly machine have arbitrary values as VAMP assembly instructions we model in μ_{ASM} do not change the control flow. Also, after applying the VAMP assembly semantics the location counter of the top stack frame is increased so that it points to the next instruction to be executed at the next μ_{ASM} step.

DEFINITION 3.16 ► μ_{ASM} Semantics For VAMP Assembly Instructions

$$\frac{\begin{array}{c} I = instr_{curr}(c_\mu, \pi_{\mu_{\text{ASM}}}) \qquad I \in Instr_{cut4\mu_{\text{ASM}}} \qquad \neg(rd(I) \in \{sp, bp\}) \\ rs_1(I) \in \{sp, bp\} \vee rs_2(I) \in \{sp, bp\} \longrightarrow \neg is\_stack(c_\mu) \\ c_{ASM} = exec_{instr}((\mathsf{A}, \mathsf{A}, c_\mu.gpr, c_\mu.spr, c_\mu.\mathcal{M}), I) \end{array}}{\pi_{\mu_{\text{ASM}}} \vdash c_\mu \rightarrow_{\mu_{\text{ASM}}} \lfloor inc_{loc}(c_\mu) \begin{bmatrix} \mathcal{M} & := c_{ASM}.\mathcal{M}, \\ gpr & := c_{ASM}.gpr, \\ spr & := c_{ASM}.spr \end{bmatrix} \rfloor}$$

However, there is a software condition stating that this VAMP assembly instruction must not write in a memory region where the stack resides. In the μ_{ASM} semantics we do not know where the stack resides in memory, but we know it when a μ_{ASM} program gets assembled. Thus, we state this software condition as a precondition to a μ_{ASM} program in Section 3.3.3 in which we will talk about the correctness of the μ_{ASM} assembler.

In case the execution of the VAMP assembly instruction modeled in μ_{ASM} accesses (either reads or writes) the content of one of the stack registers we distinguish on whether the stack of the μ_{ASM} machine

- was not set up before executing this instruction. Here we consider the write case. If the content of stack registers indicates that the stack is set up after executing this instruction, then the no-stack of a μ_{ASM} configuration is substituted by the corresponding empty stack. This stack frame is constructed based on the no-stack in the following way: (i) the procedure name is the same as in the no-stack, (ii) the location counter contains the location counter of the no-stack incremented by 1, (iii) the rest of components of the stack frame are left empty.

DEFINITION 3.17 ► μ_{ASM} Semantics For VAMP Stack Set-Up

$$\frac{\begin{array}{c} I = instr_{curr}(c_\mu, \pi_{\mu_{\text{ASM}}}) \qquad I \in Instr_{cut4\mu_{\text{ASM}}} \\ rd(I) \in \{sp, bp\} \qquad \neg is\_stack(c_\mu) \\ c_{ASM} = exec_{instr}((\mathsf{A}, \mathsf{A}, c_\mu.gpr, c_\mu.spr, c_\mu.\mathcal{M}), I) \\ are\_sregs\_set\_up(c_{ASM}) \end{array}}{\pi_{\mu_{\text{ASM}}} \vdash c_\mu \rightarrow_{\mu_{\text{ASM}}} \lfloor c_\mu \begin{bmatrix} \mathcal{M} & := c_{ASM}.\mathcal{M}, \\ gpr & := c_{ASM}.gpr, \\ spr & := c_{ASM}.spr, \\ stack & := [(c_\mu.p_{top}, c_\mu.loc_{top} + 1, [], [], [])] \end{bmatrix} \rfloor}$$

Here we want to stress that after performing the rule presented above the stack registers are still present in the μ_{ASM} configuration with the set up stack. In the μ_{ASM}

semantics any try to access them when the stack is set up and not empty yields the execution of an instruction performing this access to an error state. In this way we keep the stack registers "invisible" artificially in the μ_{ASM} semantics when the stack of a μ_{ASM} machine c_μ being executed is set up and not empty. But it is possible to access the stack registers when the stack is set up and empty. In this situation we can simply construct the corresponding μ_{ASM} machine with no-stack and with set up stack registers. If two μ_{ASM} machines c_μ^1 and c_μ^2 have different type of stacks, i.e. one has a set up stack which is empty and another has no-stack, then we call them semantically equivalent machines if and only if the following conditions hold:

$$\neg is\_stack(c_\mu^1) \wedge is\_stack(c_\mu^2) \wedge is\_empty\_stack(c_\mu^2) \wedge$$
$$c_\mu^1.gpr = setup\_sregs(c_\mu^2).gpr \wedge c_\mu^1.spr = c_\mu^2.spr \wedge c_\mu^1.\mathcal{M} = c_\mu^2.\mathcal{M}$$

- was set up and it is empty before executing this instruction. Before applying the VAMP assembly semantics to a given μ_{ASM} machine c_μ we set its stack registers to stack base address. Here we consider two cases whether the content of stack registers still indicates that the stack is set up after executing this instruction or not:

  a) If the content of these registers indicates that the stack is set up, i.e. their content is the same as it was before executing the instruction, then the μ_{ASM} machine still has the set up stack. The reader might wonder why me map changes done at the assembly level back to the μ_{ASM} level: it could be that the executed instruction read one of stack registers and saved it in either GPRs, SPRs or memory.

◀ DEFINITION 3.18
μ_{ASM} Semantics For VAMP Stack Register Update

From Empty Stack To Empty Stack

$$\frac{\begin{array}{c} I = instr_{curr}(c_\mu, \pi_{\mu_{\text{ASM}}}) \qquad I \in Instr_{cut4\mu_{\text{ASM}}} \\ rd(I) \in \{sp, bp\} \vee rs_1(I) \in \{sp, bp\} \vee rs_2(I) \in \{sp, bp\} \\ is\_stack(c_\mu) \qquad is\_empty\_stack(c_\mu) \\ c'_\mu = setup\_sregs(c_\mu) \\ c_{ASM} = exec_{instr}((\mathsf{A}, \mathsf{A}, c'_\mu.gpr, c'_\mu.spr, c'_\mu.\mathcal{M}), I) \\ are\_sregs\_set\_up(c_{ASM}) \end{array}}{\pi_{\mu_{\text{ASM}}} \vdash c_\mu \rightarrow_{\mu_{\text{ASM}}} \lfloor c_\mu \left[\begin{array}{ll} \mathcal{M} & := c_{ASM}.\mathcal{M}, \\ gpr & := c_{ASM}.gpr, \\ spr & := c_{ASM}.spr, \\ stack & := [(c_\mu.p_{top}, c_\mu.loc_{top} + 1, [], [], [])] \end{array} \right] \rfloor}$$

  b) If the content of these registers does not indicate that the stack is set up, then the empty stack of μ_{ASM} machine is substituted by the corresponding no-stack with the incremented location counter.

DEFINITION 3.19 ► μ_{ASM} Semantics For VAMP Stack Register Update

From Empty Stack To No-Stack

$$\frac{\begin{array}{c} I = instr_{curr}(c_\mu, \pi_{\mu_{\text{ASM}}}) \qquad I \in Instr_{cut4\mu_{\text{ASM}}} \\ rd(I) \in \{sp, bp\} \vee rs_1(I) \in \{sp, bp\} \vee rs_2(I) \in \{sp, bp\} \\ is\_stack(c_\mu) \qquad is\_empty\_stack(c_\mu) \\ setup\_sregs(c_\mu, c'_\mu) \\ c_{ASM} = exec_{instr}((\mathsf{A}, \mathsf{A}, c'_\mu.gpr, c'_\mu.spr, c'_\mu.\mathcal{M}), I) \\ \neg are\_sregs\_set\_up(c_{ASM}) \end{array}}{\pi_{\mu_{\text{ASM}}} \vdash c_\mu \rightarrow_{\mu_{\text{ASM}}} \lfloor c'_\mu \left[\begin{array}{ll} \mathcal{M} & := c_{ASM}.\mathcal{M}, \\ gpr & := c_{ASM}.gpr, \\ spr & := c_{ASM}.spr, \\ stack & := (c_\mu.p_{top}, c_\mu.loc_{top} + 1) \end{array} \right] \rfloor}$$

In case the execution of the VAMP assembly instruction tries either to modify the content of one of the stack registers or to read it when the stack of the μ_{ASM} machine is set up and not empty, the execution of this instruction in the μ_{ASM} semantics yields to an error.

DEFINITION 3.20 ► μ_{ASM} Semantics For VAMP Assembly Instructions

Run-Time Error

$$\frac{\begin{array}{c} I = instr_{curr}(c_\mu, \pi_{\mu_{\text{ASM}}}) \in Instr_{cut4\mu_{\text{ASM}}} \\ rd(I) \in \{sp, bp\} \vee rs_1(I) \in \{sp, bp\} \vee rs_2(I) \in \{sp, bp\} \longrightarrow \\ is\_stack(c_\mu) \wedge \neg is\_empty\_stack(c_\mu) \end{array}}{\pi_{\mu_{\text{ASM}}} \vdash c_\mu \rightarrow_{\mu_{\text{ASM}}} \bot}$$

Goto Instructions

There are two μ_{ASM} goto instructions: `goto` l and `ifnez` r `goto` l performing an unconditional(direct) jump and a conditional jump to the target label l, respectively. Thus, these instructions affect only the location counter of the current stack frame (or no-stack). That is, these instructions can be also executed in the μ_{ASM} semantics even though the stack is not set up.

If the target label l points outside the current procedure body, then the instruction executions yields to an error state. This is reflected in the semantics of both instructions.

The instruction `goto` l This instruction jumps directly to a location indicated by the target label l if the target location is within the current procedure body.

DEFINITION 3.21 ► `goto` l Semantics

$$\frac{instr_{curr}(c_\mu, \pi_{\mu_{\text{ASM}}}) = \texttt{goto}\ l \qquad l < |\pi_{\mu_{\text{ASM}}}(c_\mu.p_{top}).body|}{\pi_{\mu_{\text{ASM}}} \vdash c_\mu \rightarrow_{\mu_{\text{ASM}}} \lfloor set_{loc}(c_\mu, l) \rfloor}$$

Else, the execution ends up with an error state.

DEFINITION 3.22 ► `goto` l Semantics

Run-Time Error

$$\frac{instr_{curr}(c_\mu, \pi_{\mu_{\text{ASM}}}) = \texttt{goto}\ l \qquad |\pi_{\mu_{\text{ASM}}}(c_\mu.p_{top}).body| \leq l}{\pi_{\mu_{\text{ASM}}} \vdash c_\mu \rightarrow_{\mu_{\text{ASM}}} \bot}$$

The instruction `ifnez` r `goto` l This instruction jumps to the location indicated by the target label l in case GPR r does not equal 0.

$$\frac{instr_{curr}(c_\mu, \pi_{\mu_{\text{ASM}}}) = \texttt{ifnez}\ r\ \texttt{goto}\ l \qquad c_\mu.gpr[r] \neq 0 \qquad l < |\pi_{\mu_{\text{ASM}}}(c_\mu.p_{top}).body| \qquad r \notin \{sp, bp\}}{\pi_{\mu_{\text{ASM}}} \vdash c_\mu \rightarrow_{\mu_{\text{ASM}}} \lfloor set_{loc}(c_\mu, l) \rfloor}$$

◀ DEFINITION 3.23
`ifnez` r `goto` l Semantics
Condition Succeeds

Else, the location counter is increased by 1. So, it has the same effects as a no-operation instruction.

$$\frac{instr_{curr}(c_\mu, \pi_{\mu_{\text{ASM}}}) = \texttt{ifnez}\ r\ \texttt{goto}\ l \qquad c_\mu.gpr[r] = 0 \qquad l < |\pi_{\mu_{\text{ASM}}}(c_\mu.p_{top}).body| \qquad r \notin \{sp, bp\}}{\pi_{\mu_{\text{ASM}}} \vdash c_\mu \rightarrow_{\mu_{\text{ASM}}} \lfloor inc_{loc}(c_\mu, l) \rfloor}$$

◀ DEFINITION 3.24
`ifnez` r `goto` l Semantics
Condition Fails

In case the target label l points to outside of the current procedure body, the execution yields to an error state.

$$\frac{instr_{curr}(c_\mu, \pi_{\mu_{\text{ASM}}}) = \texttt{ifnez}\ r\ \texttt{goto}\ l \qquad |\pi_{\mu_{\text{ASM}}}(c_\mu.p_{top}).body| \leq l \vee r \in \{sp, bp\}}{\pi_{\mu_{\text{ASM}}} \vdash c_\mu \rightarrow_{\mu_{\text{ASM}}} \bot}$$

◀ DEFINITION 3.25
`ifnez` r `goto` l Semantics
Run-Time Error

Instructions Loading and Storing Procedure Parameters

The instruction `lparam` r i reads the i-th parameter from the top-most stack frame and loads it to GPR r.

$$\frac{instr_{curr}(c_\mu, \pi_{\mu_{\text{ASM}}}) = \texttt{lparam}\ r\ i \qquad i < |c_\mu.pars_{top}| \qquad is\_stack(c_\mu) \qquad r \notin \{sp, bp\}}{\pi_{\mu_{\text{ASM}}} \vdash c_\mu \rightarrow_{\mu_{\text{ASM}}} \lfloor inc_{loc}(c'_\mu) \rfloor}$$

◀ DEFINITION 3.26
`lparam` Semantics

where

$$c'_\mu = c_\mu[gpr[r := c_\mu.pars_{top}[i]]]$$

If the parameter index i is greater than the number of parameters that are passed to the current procedure, then the execution yields to an error state.

$$\frac{instr_{curr}(c_\mu, \pi_{\mu_{\text{ASM}}}) = \texttt{lparam}\ r\ i \qquad |c_\mu.pars_{top}| \leq i \vee \neg is\_stack(c_\mu) \vee r \in \{sp, bp\}}{\pi_{\mu_{\text{ASM}}} \vdash c_\mu \rightarrow_{\mu_{\text{ASM}}} \bot}$$

◀ DEFINITION 3.27
`lparam` Semantics
Run-Time Error

The instruction `sparam` r i updates the i-th parameter of the current stack frame with the value of GPR r.

$$\frac{instr_{curr}(c_\mu, \pi_{\mu_{\text{ASM}}}) = \texttt{sparam}\ r\ i \qquad i < |c_\mu.pars_{top}| \qquad is\_stack(c_\mu) \qquad r \notin \{sp, bp\}}{\pi_{\mu_{\text{ASM}}} \vdash c_\mu \rightarrow_{\mu_{\text{ASM}}} \lfloor set_{pars}(inc_{loc}(c'_\mu), i, c_\mu.gpr[r]) \rfloor}$$

◀ DEFINITION 3.28
`sparam` Semantics

In case the parameter index i is greater than the number of passed parameters, then the execution yields to an error state.

DEFINITION 3.29 ► `sparam` Semantics Run-Time Error

$$\frac{instr_{curr}(c_\mu, \pi_{\mu_{\text{ASM}}}) = \texttt{sparam}\ r\ i \qquad |c_\mu.pars_{top}| \leq i \vee \neg is\_stack(c_\mu) \vee r \in \{sp, bp\}}{\pi_{\mu_{\text{ASM}}} \vdash c_\mu \rightarrow_{\mu_{\text{ASM}}} \bot}$$

Stack Instructions

The instruction `push` r It puts the value of GPR r on the top of the top-most stack frame lifo of a given μ_{ASM} machine c_μ if the stack of this machine is set up.

DEFINITION 3.30 ► `push` Semantics

$$\frac{instr_{curr}(c_\mu, \pi_{\mu_{\text{ASM}}}) = \texttt{push}\ r \qquad is\_stack(c_\mu) \qquad r \notin \{sp, bp\}}{\pi_{\mu_{\text{ASM}}} \vdash c_\mu \rightarrow_{\mu_{\text{ASM}}} \lfloor put_{lifo}(inc_{loc}(c_\mu), [c_\mu.gpr[r]]) \rfloor}$$

Any try to execute a push instruction by a given μ_{ASM} machine c_μ without stack and with non set up stack registers yields to an error state.

DEFINITION 3.31 ► `push` Semantics Run-Time Error

$$\frac{instr_{curr}(c_\mu, \pi_{\mu_{\text{ASM}}}) = \texttt{push}\ r \qquad \neg is\_stack(c_\mu) \vee r \in \{sp, bp\}}{\pi_{\mu_{\text{ASM}}} \vdash c_\mu \rightarrow_{\mu_{\text{ASM}}} \bot}$$

The instruction `pop` r It removes the top item from the top stack frame lifo of the μ_{ASM} machine c_μ and saves it in GPR r.

DEFINITION 3.32 ► `pop` Semantics

$$\frac{instr_{curr}(c_\mu, \pi_{\mu_{\text{ASM}}}) = \texttt{pop}\ r \qquad 0 < |c_\mu.lifo_{top}| \qquad is\_stack(c_\mu) \qquad r \notin \{sp, bp\}}{\pi_{\mu_{\text{ASM}}} \vdash c_\mu \rightarrow_{\mu_{\text{ASM}}} \lfloor drop_{lifo}(inc_{loc}(c'_\mu), 1) \rfloor}$$

where

$$c'_\mu = c_\mu[gpr[r := \mathbf{hd}(c_\mu.lifo_{top})]]$$

If the top-most stack frame lifo is empty, i.e. nothing to be ”popped” out, then the μ_{ASM} execution gets stuck.

DEFINITION 3.33 ► `pop` Semantics Run-Time Error

$$\frac{instr_{curr}(c_\mu, \pi_{\mu_{\text{ASM}}}) = \texttt{pop}\ r \qquad c_\mu.lifo_{top} = [] \vee \neg is\_stack(c_\mu) \vee r \in \{sp, bp\}}{\pi_{\mu_{\text{ASM}}} \vdash c_\mu \rightarrow_{\mu_{\text{ASM}}} \bot}$$

The instruction `call` P The call to a procedure P is successful if and only if the procedure to be called is declared and defined in the procedure table $\pi_{\mu_{\text{ASM}}}$ and the lifo of the top stack frame of the machine c_μ contains at least so many elements as the number of parameters passed on the stack. As a result of the successful call a new stack frame $frame_{new}$ is created and is put on top of the stack of the machine c_μ. Also, the location counter of a stack frame corresponding to the calling procedure is increased by 1. So that, the execution resumes at the next instruction after the call instruction whenever

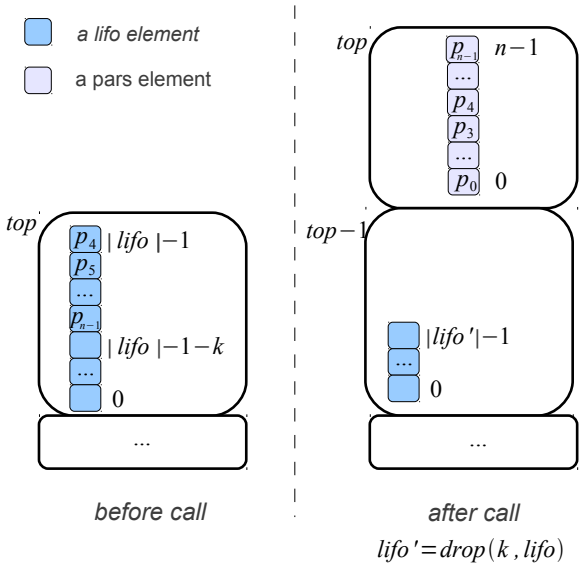


Figure 3.2: Moving Parameters From Callee Frame To Caller at CALL

the called procedure P returns. Constraints on the new created stack frame and the updated lifo are captured in the predicate $\mathit{Call\text{-}Constraints}(c_\mu, \pi_{\mu_{\mathrm{ASM}}}, P, \mathit{frame}_{new}, lifo')$ that we present a bit later.

DEFINITION 3.34 ► `call` Semantics

$$\frac{\begin{array}{c}\mathit{instr}_{curr}(c_\mu, \pi_{\mu_{\mathrm{ASM}}}) = \texttt{call}\ P \qquad \mathit{is\_stack}(c_\mu) \qquad \pi_{\mu_{\mathrm{ASM}}}(P).npar - 4 \leq |c_\mu.lifo_{top}| \\ \pi_{\mu_{\mathrm{ASM}}}(P) \neq \mathit{None} \qquad \pi_{\mu_{\mathrm{ASM}}}(P).body \neq \texttt{extern} \\ \mathit{Call\text{-}Constraints}(c_\mu, \pi_{\mu_{\mathrm{ASM}}}, P, \mathit{frame}_{new}, lifo')\end{array}}{\pi_{\mu_{\mathrm{ASM}}} \vdash c_\mu \rightarrow_{\mu_{\mathrm{ASM}}} \lfloor\, c_\mu[\mathit{stack} := \mathit{frame}_{new} \circ \mathit{inc}_{loc}(\mathit{put}_{lifo}(c_\mu, lifo')).stack]\,\rfloor}$$

The predicate $\mathit{Call\text{-}Constraints}(c_\mu, \pi_{\mu_{\mathrm{ASM}}}, P, \mathit{frame}_{new}, lifo')$ enforces constraints over a new stack frame and an updated lifo. The procedure name, location counter and lifo of a new stack frame are initialized as:

$$\mathit{frame}_{new}.p = P \quad \mathit{frame}_{new}.loc = 0 \quad \mathit{frame}_{new}.lifo = []$$

The list of saved registers of the new stack frame contains those GPRs whose indices are in the uses list of the declaration of the procedure P to be called.

$$\mathit{frame}_{new}.saved = \textbf{sublist}(c_\mu.gpr, \pi_{\mu_{\mathrm{ASM}}}(P).uses)$$

The list of parameters of a new stack frame is set up in a way following compiler calling conventions $CCL.1$ and $CCL.2$. Note that before the call instruction the programmer passes input parameters on lifo in right-to-left order ($CCL.2$). Moving parameters passed on lifo to the parameters component of the new created stack frame at call requires the index preservation. That is, the parameter found on top of the lifo of the top stack frame before call corresponds to the parameter with the highest index in the parameters component of the new created stack frame after call. (cg. Figure 3.2

depicts how it is done). Formally,

$$frame_{new}.pars = \begin{cases} \mathbf{rev}(\mathbf{take}(k_{p,\pi_{\mu_{\text{ASM}}}}, c_\mu.lifo_{top})) \circ \overbrace{[\mathbf{A},\mathbf{A},\mathbf{A},\mathbf{A}]}^{\pi_{\mu_{\text{ASM}}}(p).npar} & \text{if } 4 < \pi_{\mu_{\text{ASM}}}(p).npar \\ \overbrace{[\mathbf{A},\ldots,\mathbf{A}]} & \text{otherwise} \end{cases}.$$

where $k_{p,\pi_{\mu_{\text{ASM}}}}$ is the number of parameters passed on the stack to a procedure p to be called. This number is defined as:

$$k_{p,\pi_{\mu_{\text{ASM}}}} \stackrel{\text{def}}{=} \begin{cases} \pi_{\mu_{\text{ASM}}}(p).npar - 4 & \text{if } 4 < \pi_{\mu_{\text{ASM}}}(p).npar \\ 0 & \text{otherwise} \end{cases}$$

If there are parameters passed on the stack, then this number of elements is dropped from the lifo component of the calling stack frame as these elements are already in the parameter component of the new stack frame.

$$0 < k_{p,\pi_{\mu_{\text{ASM}}}} \longrightarrow lifo' = \mathbf{drop}(k_{p,\pi_{\mu_{\text{ASM}}}}, c_\mu.lifo_{top})$$

In case we want to call a procedure P which is undeclared in the procedure table $\pi_{\mu_{\text{ASM}}}$, then the μ_{ASM} execution ends up with an error state.

DEFINITION 3.35 ► `call` Semantics Run-Time Error

$$\frac{instr_{curr}(c_\mu, \pi_{\mu_{\text{ASM}}}) = \texttt{call } P \qquad (\pi_{\mu_{\text{ASM}}}(P) = None \vee \neg is\_stack(c_\mu))}{\pi_{\mu_{\text{ASM}}} \vdash c_\mu \rightarrow_{\mu_{\text{ASM}}} \bot}$$

The instruction `ret` If there is a frame to return to in a given μ_{ASM} configuration c_μ, then the execution returns to a procedure corresponding to the stack frame next to the top-most one. In the μ_{ASM} semantics this return is done by removing the top stack frame from the stack. Following the rule $CCL.4$ the parameters passed to the procedure corresponding to the top stack frame are removed at return. Note that these parameters have been moved from the lifo to the parameter component at a call (see the call instruction semantics defined before). So that, by removing the top stack frame we also follow the rule $CCL.4$. Also, we restore the values of those GPRs whose indices are in the *uses* component of the declaration of the procedure which corresponds to the top-most stack frame. The constraints on the new values of GPRs are captured in the predicate $Return\text{-}Constraints(c_\mu, \pi_{\mu_{\text{ASM}}}, l, gpr')$.

DEFINITION 3.36 ► `ret` Semantics

$$\frac{instr_{curr}(c_\mu, \pi_{\mu_{\text{ASM}}}) = \texttt{ret} \qquad 2 \leq |c_\mu.stack| \qquad is\_stack(c_\mu) \qquad Return\text{-}Constraints(c_\mu, \pi_{\mu_{\text{ASM}}}, gpr')}{\pi_{\mu_{\text{ASM}}} \vdash c_\mu \rightarrow_{\mu_{\text{ASM}}} drop_{frame}(c_\mu[gpr := gpr'])}$$

The $Return\text{-}Constraints(c_\mu, \pi_{\mu_{\text{ASM}}}, gpr')$ enforces the following constraints:

$$gpr'[i] = \begin{cases} c_\mu.saved_{top}[j] & \text{if } \exists j < |\pi_{\mu_{\text{ASM}}}(c_\mu.p_{top}).uses|. \\ & \quad \pi_{\mu_{\text{ASM}}}(c_\mu.p_{top}).uses[j] = i \\ c_\mu.gpr[i] & \text{otherwise} \end{cases}$$

Any try to return from the first stack frame yields to an error state.

DEFINITION 3.37 ► `ret` Semantics Run-Time Error

$$\frac{instr_{curr}(c_\mu, \pi_{\mu_{\text{ASM}}}) = \texttt{ret} \qquad (|c_\mu.stack| = 1 \vee \neg is\_stack(c_\mu))}{\pi_{\mu_{\text{ASM}}} \vdash c_\mu \rightarrow_{\mu_{\text{ASM}}} \bot}$$

Offset	Full Name	Description
0	previous base pointer	start address of the previous frame base pointer
4	return address	where to jump after completion of the corresponding procedure

Table 3.2: Stack Frame Header Layout

Multiple Step Transition Function

The n-step transition function $\delta_{\mu_{\text{ASM}}} :: \mathbb{N} \times C_{\mu_{\text{ASM}}} \times Prog_{\mu_{\text{ASM}}} \mapsto C_{\mu_{\text{ASM}}}\perp$ is defined by induction on n:

$$\delta^0_{\mu_{\text{ASM}}}(c_\mu, \pi_{\mu_{\text{ASM}}}) \stackrel{\text{def}}{=} \lfloor c_\mu \rfloor$$

$$\delta^n_{\mu_{\text{ASM}}}(c_\mu, \pi_{\mu_{\text{ASM}}}) \stackrel{\text{def}}{=} \begin{cases} \delta_{\mu_{\text{ASM}}}(c'_\mu, \pi_{\mu_{\text{ASM}}}) & \text{if } \delta^{n-1}_{\mu_{\text{ASM}}}(c_\mu, \pi_{\mu_{\text{ASM}}}) = \lfloor c'_\mu \rfloor \\ \perp & \text{if } \delta^{n-1}_{\mu_{\text{ASM}}}(c_\mu, \pi_{\mu_{\text{ASM}}}) = \perp \end{cases}$$

3.2 Assembling μ_{ASM} Programs

In this section we present one of possible ways to implement the μ_{ASM} assembler. The crucial points in such a code generation is the stack layout of the compiled μ_{ASM} program and the way we manipulate on the stack. First, we will expose the stack layout as well as we mention resources we need to implement operations on stack. Finally, we will examine the μ_{ASM} code generation itself.

3.2.1 Stack Layout

The stack layout used by the μ_{ASM} assembler is presented in Figure 3.3. Our stack grows downwards from its origin, i.e. the stack grows in the opposite direction than memory addresses grow. The first stack frame is found at a stack base address `STACK_BASE_ADR` which is a parameter to the μ_{ASM} assembler and thus treated as a constant here. The base address of an *active frame*, or most of time we refer to it as top stack frame, is indicated by GPR bp (Table 3.1). The address of a most recently referenced location on the stack is indicated by GPR sp (Table 3.1). So we use two GPRs: bp and sp, to implement the operations on stack[2].

Each stack frame has a so-called *stack frame header* which occupies two words and stores the *previous base pointer* and *return address*. The offsets to the base address of a stack frame and the meaning of stack frame header fields are explained in Table 3.2.

3.2.2 Extending Compiler Calling Convention

There exists another rule in *CCL* which is not visible in the μ_{ASM} semantics, but it becomes visible in the implementation of the μ_{ASM} assembler. This rule splits up the task of setting up for and restoring the stack after a procedure call between the caller and the callee, respectively.

- What the caller should do for setting up the stack for a procedure call:

[2]Intel X86 uses two registers *esp* and *ebp* for the same purpose to implement operations on stack.

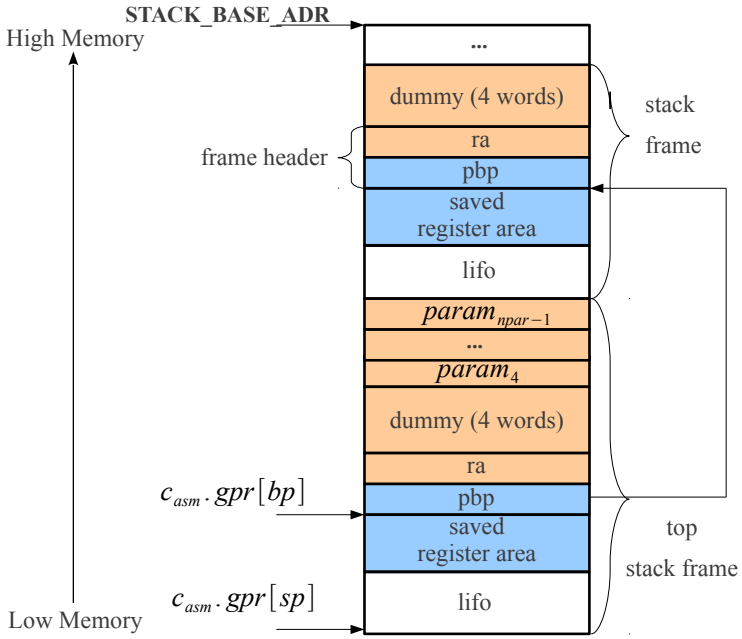


Figure 3.3: μ_{ASM} Stack Layout

  - reserve space on stack for parameters passed in registers ($CCL.2$),
  - set up the return address field of the frame header.

  They are colored as brown in Figure 3.3.

- What the callee should do for setting up the stack for a procedure call, i.e. for itself:

  - set up the previous base pointer field of the frame header,
  - set up the base pointer register (GPR bp)
  - save registers on stack whose values should preserved during the callee execution.

  They are colored as blue in Figure 3.3.

- What the callee should do for restoring the stack before return from a call:

  - restore register values from the stack and clean up the stack from these values,
  - set up GPR bp to its old value it had at the procedure entry, and clean up the stack from it,
  - clean up the stack from the return destination in the frame header,
  - remove the passed parameters from the stack ($CCL.4$).

It is easy to notice that unwinding the stack at a return is completely complementary to setting up the stack at a procedure call.

Code Generation

Here we introduce the function $code_\mu :: Prog_{\mu_{\text{ASM}}} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \mapsto infoT$ that takes a program $\pi_{\mu_{\text{ASM}}}$, a program base address PROG_BASE_ADR, a stack base address STACK_BASE_ADR and a stack length STACK_MAX_SIZE in words and produces static information $info_\mu$ describing the macro-assembled program[3]. This static information comprises the following components:

$$code_\mu(\pi_{\mu_{\text{ASM}}}, \texttt{PROG\_BASE\_ADR}, \texttt{STACK\_BASE\_ADR}, \texttt{STACK\_MAX\_SIZE}) = info_\mu$$

- $info_\mu.code :: Instr^*$ is a list of VAMP assembly instructions which represents the compiled μ_{ASM} program,
- $info_\mu.code_{ia} :: P_{name} \times \mathbb{N} \mapsto \mathbb{N}$ maps pairs of procedure names and locations, indicating instructions, to base addresses of those assembled instructions,
- $info_\mu.P_{adr} :: P_{name} \mapsto \mathbb{N} \cup \bot$ maps a procedure (if exists in the procedure table) to their compiled start addresses,
- $info_\mu.code_{ba} :: \mathbb{N}$ is the base address at which the compiled program starts,
- $info_\mu.stack_{ba} :: \mathbb{N}$ is the stack base address,
- $info_\mu.stack_{len} :: \mathbb{N}$ is the maximum stack size in words.

To present how the μ_{ASM} assembler static information is generated we first introduce auxiliary functions needed for that. We start with the definition of the function $code_{\mathbb{S}\mu}(instr, p, \pi_{\mu_{\text{ASM}}})$ that generates the list of VAMP assembly instructions which implements a given μ_{ASM} statement *instr*, where this μ_{ASM} statement is found in the body of the procedure p implemented in a given μ_{ASM} program $\pi_{\mu_{\text{ASM}}}$.

◀ DEFINITION 3.38
μ_{ASM} Statement Code Generation

$$code_{\mathbb{S}\mu} :: \mathbb{S}_{\mu_{\text{ASM}}} \times P_{name} \times Prog_{\mu_{\text{ASM}}} \mapsto Instr^*$$

We define this function by case distinctions on a given μ_{ASM} statements *instr*. Note that for some statements we use already pre-computed data, i.e. the μ_{ASM} assembler static information is generated in a multi-pass fashion. In our case it is a two-pass assembler.

VAMP Assembly instructions The code generation of VAMP assembly instructions modeled in μ_{ASM} is trivial as nothing extra should be generated besides the instruction itself. Formally, for $instr \in Instr_{cut4\mu_{\text{ASM}}}$:

$$code_{\mathbb{S}\mu}(instr, p, \pi_{\mu_{\text{ASM}}}) \stackrel{\text{def}}{=} [instr]$$

[3] in this thesis we refer to the static information of a macro-assembled (or compiled) program as *static info(-rmation)*

Goto instructions To generate the corresponding code for goto instructions we need to know the absolute address where we need to jump to. That is, we need to translate the target location of a μ_{ASM} goto statement to the corresponding absolute address. We denote this address as a_l and it is computed as

$$a_l = adr4loc(info_\mu.P_{adr}(p), p, l, \pi_{\mu_{\text{ASM}}})$$

where the function *adr4loc* computes the desirable address. The definition of this function will be presented in this section after we treated all instructions. This function takes a procedure base address $info_\mu.P_{adr}(p)$ as input which is pre-computed data from the first μ_{ASM} assembler pass. In the end of this section we will talk about the order of computations done during the μ_{ASM} assembling. After computing the address we can generate the VAMP assembly code corresponding to `goto` l (cf. Listing 3.1). But there is still something to discuss, namely, how to load a 32-bit constant representing this address in a register. There is no special instruction that can load a 32-bit constant in a register (cf. VAMP assembly semantics examined in Chapter 2). So we split up the computed address a_l into two parts each 16-bits wide and load it with the help of two instructions[4] (lines 1-2 of Listing 3.1). In line 3 of Listing 3.1 the jump is performed to the translated target location.

$$code_{\mathbb{S}\mu}(\texttt{goto}\ l, p, \pi_{\mu_{\text{ASM}}}) \stackrel{\text{def}}{=}$$

Listing 3.1: *goto l*

```
1  lhgi   t20  a_bv[15] ⊕ a_bv[31 : 16]   // load 32−bit constant(1)
2  xori   t20  t20  a_bv[15 : 0]          // load 32−bit constant(2)
3  jr     t20                             // jump to l
4  add    zero zero zero                  // delay slot filled with nop
```

where the bit vector a_{bv} is the binary representation of the address a_l, i.e. $a_{bv} = bin_{32}(a_l)$

The code generation for the instruction `ifnez` r `goto` l is similar to the previous instruction except for the zero test on the value of GPR r. In case GPR r does not equal zero we jump to the last instruction of the generated code (Listing 3.2). Otherwise, the same steps are performed as for the instruction `goto` l.

$$code_{\mathbb{S}\mu}(\texttt{ifnez}\ r\ \texttt{goto}\ l, p, \pi_{\mu_{\text{ASM}}}) \stackrel{\text{def}}{=}$$

Listing 3.2: `ifnez` r `goto` l

```
1  beqz   r  20                    // conditinonal test on zero of r
2  add    zero zero zero           // delay slot filled with nop
3  lhgi   t20  a_bv[15] ⊕ a_bv[31 : 16]   // load 32−bit constant(1)
4  xori   t20  t20  a_bv[15 : 0]          // load 32−bit constant(2)
5  jr     t20                             // jump to l
6  add    zero zero zero           // delay slot filled with nop
```

where the bit vector a_{bv} is $bin_{32}(a_l)$.

[4] Note that immediate constant is always sign extended in the VAMP assembly semantics.

Load and Store Parameters The instructions from this group read and store parameters which belong to the active stack frame. The code generation is pretty simply. We read and store parameters with the corresponding relative offsets to the frame base pointer (GPR *bp*).

$$code_{\mathbb{S}\mu}(\texttt{lparam}\ r\ j, p, \pi_{\mu_{\text{ASM}}}) \stackrel{\text{def}}{=}$$

Listing 3.3: `lparam` *r j*

```
1 lw   r   bp  4 · (2 + j)
```

$$code_{\mathbb{S}\mu}(\texttt{sparam}\ r\ j, p, \pi_{\mu_{\text{ASM}}}) \stackrel{\text{def}}{=}$$

Listing 3.4: `sparam` *r j*

```
1 sw   r   bp   4 · (2 + j)
```

Push and Pop instructions As the stack pointer GPR *sp* points to the lowest word in the stack. The generated code for the μ_{ASM} statement `push` *r* first decrements the stack pointer by the size of a single word in bytes as the stack grows downwards. Then the value is stored on the stack. The corresponding the code generation is presented below.

Listing 3.5: `push` *r*

```
1 subi  sp  zero  4     // decrement stack pointer
2 sw    r   sp    0     // put r on top of stack
```

The generated code for the μ_{ASM} statement `pop` *r* is complementary to `push` *r*.

Listing 3.6: `pop` *r*

```
1 lw    r   sp    0     // read value from top of stack and save it into r
2 addi  sp  zero  4     // increase stack pointer
```

The `call` instruction The μ_{ASM} instruction `call` gets translated to code which must follow the extended *CCL*. The space is reserved in the stack for parameters passed in registers (*CCL*.2). The number of parameters passed in registers is (cf. the compiler calling convention *CCL*.1):

$$par_{regs}(p, \pi_{\mu_{\text{ASM}}}) \stackrel{\text{def}}{=} \begin{cases} 4 & \text{if } 4 < \pi_{\mu_{\text{ASM}}}(p).npar \\ \pi_{\mu_{\text{ASM}}}(p).npar & \text{otherwise} \end{cases}$$

Also, a single word is reserved on the stack for the return address component of the frame header of a procedure *P*. This component is updated by the instruction at line 5. Next, instructions load the start address of the procedure *P* (lines 2-5). Then the jump to the procedure *P* is performed (line 6). The delay slot of this jump instruction is filled with the instruction saving the return address in the frame header on the stack.

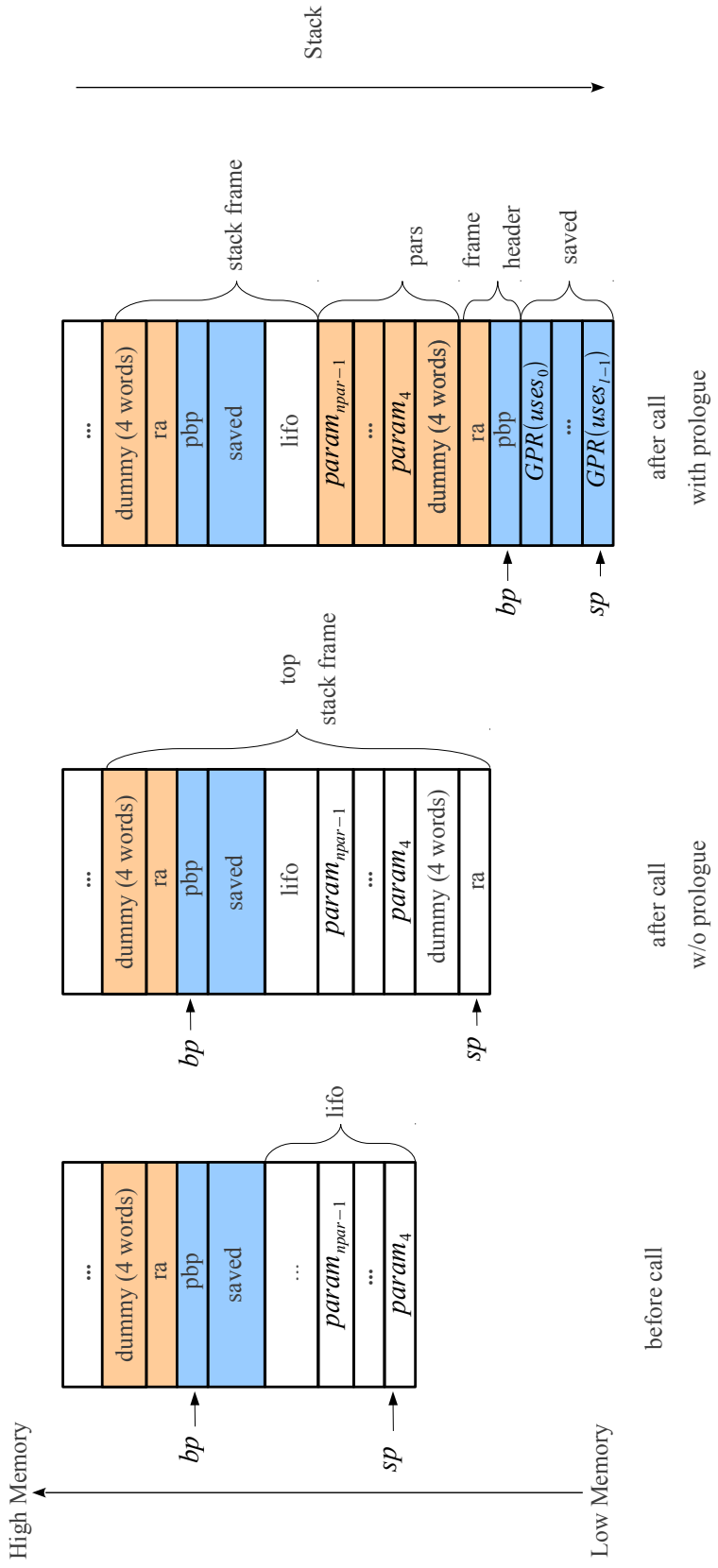


Figure 3.4: Stack states before and after a call

Figure 3.4 depicts the stack state before and after executing the code generated for a μ_{ASM} call instruction.

Listing 3.7: `call` P

```
1 subi  sp  sp  4 · (par_regs(P, π_μASM) + 1)   // reserve dummy space on stack
2 lhgi  t20  a_bv[15] ⊕ a_bv[31 : 16]           // load 32−bit constant(1)
3 xori  t20  t20  a_bv[15 : 0]                  // load 32−bit constant(2)
4 jal   t20                                     // jumping to P
5 sw    31  sp  0   // delay slot: save return address in the frame header
```

where the bit vector a_{bv} is $bin_{32}(info_{\mu}.P_{adr}(P))$.

The code generated for a μ_{ASM} call instruction does not do the whole work what must be done. Like, setting up the base pointer to a new value, saving registers on the stack, etc. This missing code, called *prologue*, is generated and put at the beginning of each μ_{ASM} procedure from a μ_{ASM} program being assembled. That is, we can say that the code generated for a μ_{ASM} call instruction is split up into two parts where the first part is present on the caller side and the second part is present on the callee side. Next, we examine how exactly the μ_{ASM} procedure prologue gets generated.

Prologue The prologue is generated for every procedure of a μ_{ASM} program except for main one. The prologue code we generate has two parts, where the first is always generated, and the second is generated in case the *uses* component of the procedure declaration is not empty. The first part of the generated prologue is a part of the `call` instruction but on the callee side. Here we put the caller GPR bp on the stack. So we have stored GPR bp in the callee frame header. Then we set up a new frame base pointer GPR bp for the callee. It is equal to the stack pointer GPR sp decremented by 4 as the previous instruction which stored GPR bp on stack did not decrement the stack pointer GPR sp.

The second part of the generated prologue saves those values of GPRs whose indices are in the list $uses = \pi_{\mu_{\text{ASM}}}(p).uses$ of the procedure declaration, where $l = |uses|$ is the length of this list. This code consists of chunk of store word instructions which save GPR $uses_i$ in order of appearance in the list *uses* starting from the very first element. Figure 3.4 depicts the stack state after executing the prologue code.

Listing 3.8: Prologue

```
1 sw  bp  sp  −4              // save bp in previous base pointer component
2 subi  bp  sp  4             // switch base pointers (with adjuststed stack poiner)
3 sw  uses_0  sp  −8
4 sw  uses_1  sp  −12
5 ...
6 sw  uses_{l−1}  sp  −(l · 4 + 4)
7 subi  sp  sp  (l · 4 + 4)
```

The `ret` instruction A μ_{ASM} return instruction gets translated to code that consists of two parts. The first part is generated in case the *uses* component of the procedure declaration of a procedure p we return from is not empty (In Listing. 3.9 lines 1-7 represent this first part). This first part restores values of those GPRs whose indices are in the *uses* component. The second part is always generated and it performs the

following: (i) adjusts the stack pointer and cleans up the stack from parameters (line 7), (ii) restores the old value of GPR bp (line 5 and line 8), (iii) reads the return address (line 6) and returns the control to the caller (line 9).

Listing 3.9: `ret`

```
1  lw uses_{l-1} sp 0
2  ...
3  lw uses_0 sp (l − 1) · 4
4  addi sp sp l · 4
5  lw tr_0 bp 0 // load pbp
6  lw tr_1 bp 4 // load ra
7  addi sp sp 8 + 4 · π_{μASM}(p).npar // adjust the stack pointer
8  add bp tr_0 zero                    // restore old bp
9  jr tr_1                             // return control to caller
10 add zero zero zero                  // nop in delay slot
```

The function $num_{\mathbb{S}\mu}(instr, p, \pi_{\mu_{\text{ASM}}})$ returns the number of VAMP assembly instructions needed to implement a given μ_{ASM} instruction $instr$. After we introduce the code generation for each μ_{ASM} instruction it would be easy to check it.

DEFINITION 3.39 ► Size of Assembled μ_{ASM} Statement in Instructions

$$num_{\mathbb{S}\mu} :: \mathbb{S}_\mu \times P_{name} \times Prog_{\mu_{\text{ASM}}} \mapsto \mathbb{N}$$

$$num_{\mathbb{S}\mu}(instr, p, \pi_{\mu_{\text{ASM}}}) \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } instr \in Instr \\ 1 & \text{if } instr \in \{\texttt{lparam}\ r\ i, \texttt{sparam}\ r\ i\} \\ 2 & \text{if } instr \in \{\texttt{push}\ r, \texttt{pop}\ r\} \\ 6 & \text{if } instr = \texttt{goto}\ l \\ 8 & \text{if } instr = \texttt{ifnez}\ r\ \texttt{goto}\ l \\ 7 & \text{if } instr = \texttt{call}\ P \\ 6 & \text{if } instr = \texttt{ret} \wedge l = 0 \\ 7 + l & \text{if } instr = \texttt{ret} \wedge 0 < l \end{cases}$$

where $l = |\pi_{\mu_{\text{ASM}}}(p).uses|$.

The function $size_{\mathbb{S}\mu}(instr, p, \pi_{\mu_{\text{ASM}}})$ returns the number of bytes needed to represent the μ_{ASM} instruction $instr$ after assembling it to VAMP assembly instructions. Recall that from Chapter 2 each VAMP assembly instruction is represented by 4 bytes in memory.

DEFINITION 3.40 ► Size of Assembled μ_{ASM} Statement in Bytes

$$size_{\mathbb{S}\mu}(instr, p, \pi_{\mu_{\text{ASM}}}) \stackrel{\text{def}}{=} 4 \cdot num_{\mathbb{S}\mu}(instr, p, \pi_{\mu_{\text{ASM}}})$$

Also, we define the function $size_{prol}(p, \pi_{\mu_{\text{ASM}}})$ returning the size of code in bytes needed to store the generated prologue for a given procedure p implemented in a given μ_{ASM} program $\pi_{\mu_{\text{ASM}}}$. This size depends on whether a given procedure p is the main one and on the length of the component $uses$ in the declaration of the procedure p.

DEFINITION 3.41 ► Prologue Size in Bytes

$$size_{prol}(p, \pi_{\mu_{\text{ASM}}}) \stackrel{\text{def}}{=}$$

$$\begin{cases} 0 & \text{if } main?_{\mu_{\text{ASM}}}(p) \\ 8 & \neg main?_{\mu_{\text{ASM}}}(p) \wedge \pi_{\mu_{\text{ASM}}}(p).uses = [] \\ 12 + 4 \cdot |\pi_{\mu_{\text{ASM}}}(p).uses| & \text{otherwise} \end{cases}$$

The function *adr4loc* computes the start address at which the assembled version of a μ_{ASM} instruction resides in memory. This μ_{ASM} instruction is indicated by the location counter *loc* and by the procedure P, where the assembled code of the procedure P starts at address $proc_{ba}$.

◄ DEFINITION 3.42
Address For μ_{ASM} Statement Location

$$adr4loc :: \mathbb{N} \times P_{name} \times \mathbb{N} \times Prog_{\mu_{\text{ASM}}} \mapsto \mathbb{N}$$

$$adr4loc(proc_{ba}, p, loc, \pi_{\mu_{\text{ASM}}}) \stackrel{\text{def}}{=}$$

$$proc_{ba} + size_{prol}(p, \pi_{\mu_{\text{ASM}}}) + \sum_{i=0}^{loc} size_{\mathbb{S}\mu}(\pi_{\mu_{\text{ASM}}}(p).body[i], p, \pi_{\mu_{\text{ASM}}})$$

The function $last(p, \pi_{\mu_{\text{ASM}}}) :: P_{name} \times Prog_{\mu_{\text{ASM}}} \mapsto \mathbb{N}$ returns the location of the last statement of a given procedure P.

$$last(p, \pi_{\mu_{\text{ASM}}}) \stackrel{\text{def}}{=} |\pi_{\mu_{\text{ASM}}}[p].body| - 1$$

Using definitions defined before in this section we define the function $code_{\mu}$ as follows

$$\begin{array}{lcl} info_{\mu}.code & = & \bigcirc\!\!\!\!\vee_{\ p \in P_{name}} \Big(code_{prol}(p, \pi_{\mu_{\text{ASM}}}) \circ \\ & & \bigcirc\!\!\!\!\vee_{\ i < last(p, \pi_{\mu_{\text{ASM}}})} code_{\mathbb{S}\mu}(\pi_{\mu_{\text{ASM}}}[p].body[i], p, \pi_{\mu_{\text{ASM}}}) \Big) \\ info_{\mu}.P_{adr}(p) & = & \texttt{PROG\_BASE\_ADR} + \sum_{p' < p} \Big(size_{prol}(p', \pi_{\mu_{\text{ASM}}}) + \\ & & \sum_{i < |\pi_{\mu_{\text{ASM}}}(p').body|} size_{\mathbb{S}\mu}(\pi_{\mu_{\text{ASM}}}(p').body[i], p', \pi_{\mu_{\text{ASM}}}) \Big) \\ info_{\mu}.code_{ia}(p, i) & = & info_{\mu}.P_{adr}(p) + size_{prol}(p, \pi_{\mu_{\text{ASM}}}) + \\ & & \sum_{j \le i} size_{\mathbb{S}\mu}(\pi_{\mu_{\text{ASM}}}(p).body[i], p, \pi_{\mu_{\text{ASM}}}) \\ info_{\mu}.code_{ba} & = & \texttt{PROG\_BASE\_ADR} \\ info_{\mu}.stack_{ba} & = & \texttt{STACK\_BASE\_ADR} \\ info_{\mu}.stack_{len} & = & \texttt{STACK\_MAX\_SIZE} \end{array}$$

Here the order of computations is: First the component $info_{\mu}.P_{adr}$ is computed during the first pass of the μ_{ASM} assembler, and during the second pass the rest of the μ_{ASM} assembler static information is computed.

It is not presented in the definition presented above, but we assume that the first μ_{ASM} procedure generated is the main one. Formally,

$$info_{\mu}.P_{adr}(\text{"\_start"}) = \texttt{PROG\_BASE\_ADR}$$

3.3 Simulation Theorem

In this section we present the simulation theorem which states the correctness of the μ_{ASM} assembler defined in the previous section. An important aspect of the simulation theorem[5] is the simulation relation between states of the μ_{ASM} semantics and states

[5] the motivation of the simulation relation is described in Section 8.2.1 in [Lei07]

of VAMP assembly semantics. To guarantee the semantically equivalent execution of μ_{ASM} and VAMP assembly machines running a μ_{ASM} program and its assembled version, respectively, we define the simulation relation $consis_\mu$ indicating that both machines are consistent. We also call this simulation relation *assembler consistency*. Then we can prove the correctness of this theorem by a stepwise simulation between both semantics.

The simulation relation $consis_\mu(c_\mu, info_\mu, c_{ASM})$ states that the VAMP assembly configuration c_{ASM} encodes the μ_{ASM} configuration c_μ via the μ_{ASM} static information $info_\mu$. It comprises the *control consistency* $consis_\mu^{control}(c_\mu, info_\mu, c_{ASM})$ and the *data consistency* $consis_\mu^{data}(c_\mu, info_\mu, c_{ASM})$.

DEFINITION 3.43 ► Consistency μ_{ASM}

$$\frac{consis_\mu^{control}(c_\mu, info_\mu, c_{ASM}) \qquad consis_\mu^{data}(c_\mu, info_\mu, c_{ASM})}{consis_\mu(c_\mu, info_\mu, c_{ASM})}$$

Before we start with the formal definition of the simulation relation we introduce some auxiliary functions. We first introduce the function

$$dist_{\mu_{\text{ASM}}}^{bp} :: \mathbb{N} \times frame_\mu^* \mapsto \mathbb{N}$$

returning the distance (number of bytes) between frame base addresses of the i-th and $(i+1)$-th stack frames of the list of stack frames s. The number of passed parameters to the $(i+1)$-th stack frame should be taken into account in the computation of the distance between i-th and $(i+1)$-th stack frames as well as the size of a stack frame header (8 bytes) (cf. Figure 3.3). If the i-th stack frame is the top one, then we do not take into account the number of passed parameters to this stack frame as it is already taken into account for the computation of distance between i-th and $(i-1)$-th stack frames. Formally,

DEFINITION 3.44 ► Distance Between Stack Frame Base Addresses

$$dist_{\mu_{\text{ASM}}}^{bp}(i, s) \stackrel{\text{def}}{=}$$

$$\begin{cases} 4 \cdot (|s[i].lifo| + |s[i].saved|) & \text{if } i = |s| - 1 \\ 8 + 4 \cdot (|s[i].lifo| + |s[i].saved| + |s[i+1].pars|) & \text{otherwise} \end{cases}$$

Next we define a function

$$frame^{ba} :: frame_\mu^* \times \mathbb{N} \mapsto \mathbb{N}$$

computing an absolute frame base address for the i-th stack frame from the stack frame list s, where the base address of the first stack frame is identified by the constant `STACK_BASE_ADR` introduced in the previous section. This function is based on the function $dist_{\mu_{\text{ASM}}}^{bp}$ defined above which computes relative offsets between base addresses of adjacent stack frames.

DEFINITION 3.45 ► Stack Frame Base Address

$$frame^{ba}(s, i) \stackrel{\text{def}}{=} \texttt{STACK\_BASE\_ADR} - \sum_{j=0}^{j<i} (dist_{\mu_{\text{ASM}}}^{bp}(j, s))$$

Having a function computing a stack frame base address and knowing the stack layout itself (including the frame header) we can define functions reading frame headers of stack frames in memory. First, we define a function

$$stack^{word} :: frame_\mu^* \times \mathbb{N} \times \mathbb{N} \times (\mathbb{N} \mapsto \mathbb{B}^8) \mapsto \mathbb{Z}$$

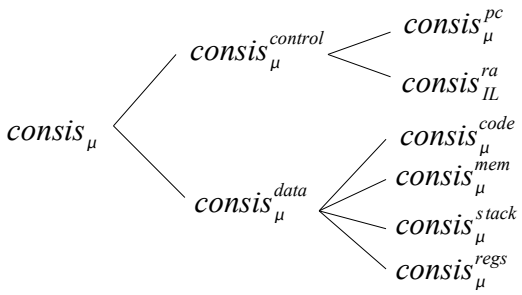


Figure 3.5: μ_{ASM} Consistency

reading the j-th word belonging to i-th stack frame from VAMP assembly machine memory m (cf. Figure 3.3).

◄ DEFINITION 3.46
Reading Word From Stack

$$stack^{word}(s, i, j, m) \stackrel{\text{def}}{=} m_{word}(frame^{ba}(s, i) - 4 \cdot j)$$

Having such a function like $stack^{word}(s, i, j, m)$ we can easily define functions reading the previous base pointer and return address components from the frame header of the i-th stack frame.

◄ DEFINITION 3.47
Reading Frame Headers

$$stack^{pbp}(s, i, m) \stackrel{\text{def}}{=} stack^{word}(s, i, 0, m)$$
$$stack^{ra}(s, i, m) \stackrel{\text{def}}{=} stack^{word}(s, i, -1, m)$$

3.3.1 Control Consistency

The control consistency comprises the program counter consistency $consis_\mu^{pc}(c_\mu, info_\mu, c_{ASM})$ and the return address consistency $consis_\mu^{ra}(c_\mu, info_\mu, c_{ASM})$.

◄ DEFINITION 3.48
Control Consistency

$$\frac{consis_\mu^{pc}(c_\mu, info_\mu, c_{ASM}) \qquad consis_\mu^{ra}(c_\mu, info_\mu, c_{ASM})}{consis_\mu^{control}(c_\mu, info_\mu, c_{ASM})}$$

The program counter consistency states that the delayed program counter dpc of the assembly machine c_{ASM} points to the current instruction in the μ_{ASM} machine c_μ. Also, it states that the normal program counter of the assembly machine c_{ASM} points to the next instruction.

◄ DEFINITION 3.49
Program Counter Consistency

$$consis_\mu^{pc} :: C_{\mu_{\text{ASM}}} \times infoT \times C_{\text{ASM}} \mapsto \mathbb{B}$$

$$\frac{\begin{array}{c} c_{ASM}.dpc = info_\mu.code_{ia}(c_\mu.p_{top}, c_\mu.loc_{top}) \\ c_{ASM}.pc = c_{ASM}.dpc + 4 \end{array}}{consis_\mu^{pc}(c_\mu, info_\mu, c_{ASM})}$$

The return address consistency states that return address components of frame headers of all stack frames (except for 0-th one) of the μ_{ASM} machine c_μ point directly behind the code of call instructions which generated these stack frames. The return address of i-th stack frame is saved in the memory of the assembly machine c_{ASM} at address identified by $stack^{ra}(c_\mu.stack, i)$. The next instruction of a procedure which called a procedure corresponding to the i-th stack frame is identified by the procedure name and the current location counter of $(i-1)$-th stack frame of the machine c_μ.

DEFINITION 3.50 ► Return Address Consistency

$$\frac{\begin{array}{c}\neg is\_stack(c_\mu) \\ \forall 0 < i \leq top(c_\mu).\ c_{ASM}.m_{word}(stack^{ra}(c_\mu.stack, i)) = \\ info_\mu.code_{ia}(c_\mu.stack[i-1].p, c_\mu.stack[i-1].loc)\end{array}}{consis^{ra}_\mu(c_\mu, info_\mu, c_{ASM})}$$

3.3.2 Data Consistency

The data consistency comprises the register consistency $consis^{regs}_\mu$, code consistency $consis^{code}_\mu$, stack consistency $consis^{stack}_\mu$ and memory consistency $consis^{mem}_\mu$.

DEFINITION 3.51 ► Data Consistency

$$\frac{\begin{array}{cc}consis^{regs}_\mu(c_\mu, c_{ASM}) & consis^{code}_\mu(c_\mu, info_\mu, c_{ASM}) \\ consis^{stack}_\mu(c_\mu, info_\mu, c_{ASM}) & consis^{mem}_\mu(c_\mu, info_\mu, c_{ASM})\end{array}}{consis^{data}_\mu(c_\mu, info_\mu, c_{ASM})}$$

The register consistency argues about the content GPRs and SPRs in the μ_{ASM} machine c_μ and in the assembly machine c_{ASM}. Here we do not argue about program counters as we did it in the program counter consistency. There are two GPRs (sp and bp) which are used for stack operations and whose content should have values according to their intended meaning from Table 3.1. GPR bp should point to the base address of the top stack frame of the μ_{ASM} machine c_μ. The difference between the values stored in GPRs sp and bp of the assembly machine c_{ASM} is the distance returned by the function $dist^{bp}_{\mu_{\text{ASM}}}$ for the top stack frame (cf. Figure 3.3). Next, we do not require the equality of GPRs sp and bp as they are invisible[6] in the μ_{ASM} semantics as long as there is a stack, but we do require the equality for the other registers.

DEFINITION 3.52 ► Register Consistency

$$\frac{\begin{array}{rc}is\_stack(c_\mu) \longrightarrow & c_{ASM}.gpr[bp] = frame^{ba}(c_\mu.stack, top(c_\mu)) \\ & \wedge\ c_{ASM}.gpr[sp] = c_{ASM}.gpr[bp] - dist^{bp}_{\mu_{\text{ASM}}}(top(c_\mu), c_\mu.stack) \\ \neg is\_stack(c_\mu) \longrightarrow & c_\mu.gpr[sp] = c_{ASM}.gpr[sp] \wedge c_\mu.gpr[bp] = c_{ASM}.gpr[bp] \\ & \forall i < 32.\ i \notin \{sp, bp\} \implies c_\mu.gpr[i] = c_{ASM}.gpr[i] \\ & \forall i < 32.\ c_\mu.spr[i] = c_{ASM}.spr[i]\end{array}}{consis^{regs}_\mu(c_\mu, c_{ASM})}$$

The code consistency requires that the compiled code of a μ_{ASM} program is stored in the memory of the assembly configuration c_{ASM} at address `PROG_BASE_ADR`. In other words, the code modification is not allowed.

DEFINITION 3.53 ► Code Consistency

[6] In the μ_{ASM} semantics the stack registers are artificially invisible. Their invisibility is guaranteed by the following: any try access them when the stack is created and non-empty yields to an error state.

$$\frac{\forall i < |info_\mu.code|. \quad info_\mu.code[i] = \textit{int-to-instr}(c_{ASM}.m_{word}(info_\mu.code_{ba} + 4 \cdot i))}{consis^{code}_\mu(c_\mu, info_\mu, c_{ASM})}$$

The stack consistency comprises the previous base pointer consistency $consis^{pbp}_\mu$ and the item consistency $consis^{item}_\mu$.

◀ DEFINITION 3.54
Stack Consistency

$$\frac{consis^{pbp}_\mu(c_\mu, c_{ASM}) \qquad consis^{item}_\mu(c_\mu, info_\mu, c_{ASM})}{consis^{stack}_\mu(c_\mu, info_\mu, c_{ASM})}$$

The previous base pointer consistency states that previous base pointer components of frame headers of stack frames (except for 0-th one) of μ_{ASM} machine c_μ to be consistent with their intended meaning from Table 3.2, i.e they point to base addresses of prior stack frames.

◀ DEFINITION 3.55
Previous Base Pointer Consistency

$$\frac{\begin{array}{c}\neg is\_stack(c_\mu) \\ \forall 0 < i \leq top(c_\mu).\ stack_{pbp}(c_\mu.stack, i, c_{ASM}.m) = frame^{ba}(c_\mu.stack, i-1)\end{array}}{consis^{stack}_\mu(c_\mu, info_\mu, c_{ASM})}$$

After presenting the item consistency we will completely expose the stack layout, namely how it is represented in memory (cf. Figure 3.3). Note that we do not state the consistency for components of stack frame headers as it is already covered by the return address and frame header consistencies. Here we state the relation between values stored in memory corresponding to a μ_{ASM} stack frame and their abstracted equivalents from that stack frame: $saved$, $lifo$ and $pars$.

We introduce a few shorthands

$$\begin{aligned} saved_i &= c_\mu.stack[i].saved \\ lifo_i &= c_\mu.stack[i].lifo \\ pars_i &= c_\mu.stack[i].pars \end{aligned}$$

for the sake of easy understanding of the consistency defined below.

◀ DEFINITION 3.56
Item Consistency

$$\frac{\begin{array}{l} is\_stack(c_\mu) \longrightarrow \\ \forall\ i \leq top(c_\mu). \quad \forall\ j < dist^{bp}_{\mu_{\text{ASM}}}(i, c_\mu.stack). \\ \quad stack^{word}(c_\mu.stack, i, j+1, c_{ASM}.m) = \\ \quad \begin{cases} saved_i[j] & \text{if } 0 \leq j < |saved_i| \\ lifo_i[j - |saved_i|] & \text{if } |saved_i| \leq j < |saved_i| + |lifo_i| \\ pars_{i+1}[|pars_{i+1}| - (j - |saved_i| - |lifo_i|)] & \text{otherwise} \end{cases} \end{array}}{consis^{item}_\mu(c_\mu, info_\mu, c_{ASM})}$$

Please note that for the top stack frame the third case is never considered (by definition of the function $dist^{bp}_{\mu_{\text{ASM}}}$).

The memory consistency states that the memory content of both machines is the same except for memory regions where the stack and the assembled code reside.

DEFINITION 3.57 ►
Memory Consistency

$$\frac{\begin{array}{rl} & a \notin [info_\mu.stack_{ba} : info_\mu.stack_{ba} - info_\mu.stack_{max-size}] \\ \forall a \in \mathbb{N}_{32}. & \wedge\ a \notin [info_\mu.code_{ba} : info_\mu.code_{ba} + 4 \cdot |info_\mu.code|] \\ & \quad \Longrightarrow\ c_{ASM}.m(a) = c_\mu.\mathcal{M}(a) \end{array}}{consis^{mem}_\mu(c_\mu, info_\mu, c_{ASM})}$$

3.3.3 Simulation Theorem

In order to succeed with μ_{ASM} executions that could be simulated by VAMP assembly machine steps, preconditions have to be fulfilled by a μ_{ASM} program. These preconditions are stated as predicates over: (i) a μ_{ASM} program , and (ii) a μ_{ASM} machine executing this program. These preconditions are divided into static and dynamic properties.

Static Properties They can be shown by a static check without any execution of the μ_{ASM} program. Hence, they are stated for a given μ_{ASM} program $\pi_{\mu_{\text{ASM}}}$. Namely, they are stated over declared and defined procedures in this program and these properties are: (i) declared indices in the *uses* component of the procedure declaration should be unique, (ii) the last statement should be a return instruction in each defined procedure (except for the main one) (iii) there exists a single return statement for each μ_{ASM} procedure, except for the main one, and (iv) all μ_{ASM} goto instructions do not jump out of the body of the μ_{ASM} procedure it belongs to, this property indicates whether the procedure can be translated to VAMP assembly or not. Formally,

DEFINITION 3.58 ►
Static Properties On μ_{ASM} Program

$$\frac{\begin{array}{rl} & \forall\ p \in P_{name}.\ \pi_{\mu_{\text{ASM}}}(p) \neq \bot \wedge \pi_{\mu_{\text{ASM}}}(p).body \neq \texttt{extern} \longrightarrow \\ & dstnct?(\pi_{\mu_{\text{ASM}}}(p).uses) \\ \wedge & (\neg main?_{\mu_{\text{ASM}}}(p) \longrightarrow \exists!i.\ \pi_{\mu_{\text{ASM}}}(p).body[i] = \texttt{ret}\ \wedge\ i = \pi_{\mu_{\text{ASM}}}(p).body - 1) \\ \wedge & \forall\ loc < |\pi_{\mu_{\text{ASM}}}(p).body|.\ \pi_{\mu_{\text{ASM}}}(p).body[loc] \in \{\texttt{goto}\ l, \texttt{ifnez}\ r\ \texttt{goto}\ l\} \longrightarrow \\ & \qquad 0 \leq l < |\pi_{\mu_{\text{ASM}}}(p).body| \end{array}}{stat\text{-}prop_{\mu_{\text{ASM}}}(\pi_{\mu_{\text{ASM}}})}$$

Dynamic Properties The μ_{ASM} semantics does not react anyhow on a stack overflow. Thus we cannot guarantee equivalent execution between μ_{ASM} program and its assembled version in this case. We fill this gap in the same way as Leinenbach did in [Lei07] by introducing a predicate $avail\text{-}stack_{\mu_{\text{ASM}}}$ which checks whether a given μ_{ASM} machine c_μ runs out of the stack memory or not.

DEFINITION 3.59 ►
Enough stack available

$$avail\text{-}stack_{\mu_{\text{ASM}}} :: C_{\mu_{\text{ASM}}} \times \mathbb{N} \times \mathbb{N} \mapsto \mathbb{B}$$

$$\frac{\sum_{i=0}^{i<|c_\mu.stack|} dist^{bp}_{\mu_{\text{ASM}}}(i, c_\mu.stack) < \texttt{STACK\_MAX\_SIZE}}{avail\text{-}stack_{\mu_{\text{ASM}}}(c_\mu)}$$

Also, we require that the μ_{ASM} program does not modify itself as well as the stack. We use the following shortcuts $I = instr_{curr}(c_\mu, \pi_{\mu_{\text{ASM}}})$ and $eadr = c_\mu.gpr[rs(I)] + imm(I)$.

DEFINITION 3.60 ►
No Self Modification

$$\frac{I \in Instr_{cut4\mu_{\text{ASM}}}\ \wedge\ instr\text{-}store?(I) \longrightarrow \neg(adr \leq\ eadr \wedge\ eadr < adr + 4 \cdot len)}{no\text{-}self\text{-}mod_{\mu_{\text{ASM}}}(c_\mu, \pi_{\mu_{\text{ASM}}}, adr, len)}$$

where *adr* is the program base address and *len* is the program length.

◀ DEFINITION 3.61
No Stack Update

$$\frac{I \in Instr_{cut4\mu_{\text{ASM}}} \ \wedge \ instr\text{-}store?(I) \longrightarrow \neg(\texttt{STACK\_BASE\_ADR} \leq \ eadr \wedge \ eadr < \texttt{STACK\_BASE\_ADR} + \texttt{STACK\_MAX\_SIZE})}{no\text{-}stack\text{-}mod_{\mu_{\text{ASM}}}(c_\mu, \pi_{\mu_{\text{ASM}}})}$$

A μ_{ASM} program could produce an interrupt in case the effective address of memory instructions is not word aligned.

◀ DEFINITION 3.62
No Data Misalignment

$$\frac{I \in Instr_{cut4\mu_{\text{ASM}}} \qquad instr\text{-}ls?(I) \longrightarrow eadr \ \mathbf{mod}\ 4 = 0}{no\text{-}dmal_{\mu_{\text{ASM}}}(c_\mu, \pi_{\mu_{\text{ASM}}})}$$

Before we state the dynamic properties, we join all step properties together in the predicate $step\text{-}prop_{\mu_{\text{ASM}}}$.

◀ DEFINITION 3.63
Step Properties

$$step\text{-}prop_{\mu_{\text{ASM}}} :: C_{\mu_{\text{ASM}}} \times Prog_{\mu_{\text{ASM}}} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \mapsto \mathbb{B}$$

$$\frac{\begin{array}{cc} no\text{-}self\text{-}mod_{\mu_{\text{ASM}}}(c_\mu, adr, len) & no\text{-}stack\text{-}mod_{\mu_{\text{ASM}}}(c_\mu, \pi_{\mu_{\text{ASM}}}) \\ no\text{-}dmal_{\mu_{\text{ASM}}}(c_\mu, \pi_{\mu_{\text{ASM}}}) & c_\mu.spr[mode] = 0 \end{array}}{step\text{-}prop_{\mu_{\text{ASM}}}(c_\mu, \pi_{\mu_{\text{ASM}}}, adr, len)}$$

The μ_{ASM} dynamic properties are stated by the predicate $dyn\text{-}props?_{\mu_{\text{ASM}}}$ defined below:

◀ DEFINITION 3.64
Dynamic Properties

$$dyn\text{-}props?_{\mu_{\text{ASM}}} :: C_{\mu_{\text{ASM}}} \times Prog_{\mu_{\text{ASM}}} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \mapsto \mathbb{B}$$

$$\frac{\forall\, i < n. \quad \delta^i_{\mu_{\text{ASM}}}(c_\mu, \pi_{\mu_{\text{ASM}}}) = \lfloor c^i_\mu \rfloor \longrightarrow avail\text{-}stack_{\mu_{\text{ASM}}}(c^i_\mu) \ \wedge \ step\text{-}prop_{\mu_{\text{ASM}}}(c_\mu, \pi_{\mu_{\text{ASM}}}, adr, len)}{dyn\text{-}props?_{\mu_{\text{ASM}}}(c_\mu, \pi_{\mu_{\text{ASM}}}, adr, len, n)}$$

◀ THEOREM 3.65
μ_{ASM} simulates VAMP Assembly

Let $\pi_{\mu_{\text{ASM}}}$ be a μ_{ASM} program, and c_{ASM} be the initial state of the VAMP assembly machine. Then for any number of steps n of the μ_{ASM} machine executing program $\pi_{\mu_{\text{ASM}}}$ there exists such a number of steps of the VAMP assembly machine such that the μ_{ASM} machine after n steps is still consistent with the VAMP assembly machine after T steps. However, new machine states are consistent if the following requirements about the μ_{ASM} and VAMP assembly machines are fulfilled:

- The μ_{ASM} program satisfies static properties,
- The μ_{ASM} configuration c_μ and VAMP assembly configuration c_{ASM} are valid and they are consistent,
- After performing n steps the μ_{ASM} machine c_μ did not get stuck,
- For all steps of the μ_{ASM} machine up to step n the dynamic properties hold.

Formally, this theorem is stated as follows:

$$
\begin{array}{lll}
 & & \mathit{stat\text{-}prop}_{\mu_{\mathrm{ASM}}}(\pi_{\mu_{\mathrm{ASM}}}) \\
\wedge & & \mathit{valid}_{asm}(c_{AS\,M}) \\
\wedge & & \mathit{valid}_{\mu_{\mathrm{ASM}}}(c_\mu) \\
\wedge & & \mathit{consis}_\mu(c_\mu, \mathit{info}_\mu, c_{AS\,M}) \\
\wedge & & \delta^n_{\mu_{\mathrm{ASM}}}(c_\mu, \pi_{\mu_{\mathrm{ASM}}}) = \lfloor c'_\mu \rfloor \\
\wedge & & \mathit{dyn\text{-}props?}_{\mu_{\mathrm{ASM}}}(c_\mu, \pi_{\mu_{\mathrm{ASM}}}, adr, len, n) \\
\Longrightarrow & & \\
 & & \exists\, T, c'_{AS\,M}. \\
 & & \delta^T_{\mathrm{ASM}}(c_{AS\,M}) = c'_{AS\,M} \\
 & \wedge & \mathit{valid}_{asm}(c'_{AS\,M}) \\
 & \wedge & \mathit{valid}_{\mu_{\mathrm{ASM}}}(c'_\mu) \\
 & \wedge & \mathit{consis}_\mu(c'_\mu, \mathit{info}_\mu, c'_{AS\,M})
\end{array}
$$

Proof: We prove this theorem by induction on the step number n of the μ_{ASM} machine.

We start the induction with n equals zero. As nothing was changed, preconditions conclude the validity of both machines and the assembler consistency.

For the induction step we have to conclude from step n to $n+1$. We prove this case by a case distinction on the instruction to be executed at step $i+1$ by the μ_{ASM} machine. For all μ_{ASM} statements the proof follows from the μ_{ASM} assembler implementation presented in the previous section.

■

CHAPTER 4

Abstract Intermediate Language C

An *intermediate representation* (IR) is the ”heart” of every compiler. It acts as central structure around which compilers are built. The compiler front-end translates a source program into its IR, some set of optimizations is performed over the IR if needed, then the back-end of the compiler generates from that IR a target code that can be run on the target processor [ASU86]. Figure 4.1 illustrates a usual compilation process.

What kind of advantages we get from using an IR in compilers:

- Compiler front- and back-ends are separated that allows to support re-targeting (in other words, cross-platform compilation),
- Compiler performs most optimizations on IR as IR is machine-independent.

In this chapter, as the next formal contribution in terms of C Semantics to the model stack, the semantics of a very low-level unstructured intermediate representation of a programming language C is presented, called *Abstract C-Intermediate-Language*, short, *C-IL* (pronounced ”c i l”), not to be confused with Common Intermediate Language (CIL)[1]. The *C-IL* semantics was invented by S.Schmaltz and was presented in our joint paper [SS12]. So, this semantics is not considered as a contribution to this thesis.

C-IL is an intermediate language that has a rich type system while at the same time being as short as possible regarding control flow and number of instructions. The main goal S.Schmaltz wants to achieve is not have a nice language for programming and optimizations but instead to be an intermediate representation language used for proofs, e.g. of low-level operating systems and hypervisors, optimizing C compilers, etc., and for combining with high-level assembly languages, like μ_{ASM}.

The *C-IL* language is a goto-language defined in form of a labeled transition system, like μ_{ASM} presented in the previous chapter. *C-IL* is defined using an abstract syntax that can be extracted from a concrete syntax tree in a straight forward way. *C-IL* operates on a global byte-addressable memory and on an abstract stack. The

[1] formerly called Microsoft Intermediate Language (MSIL)

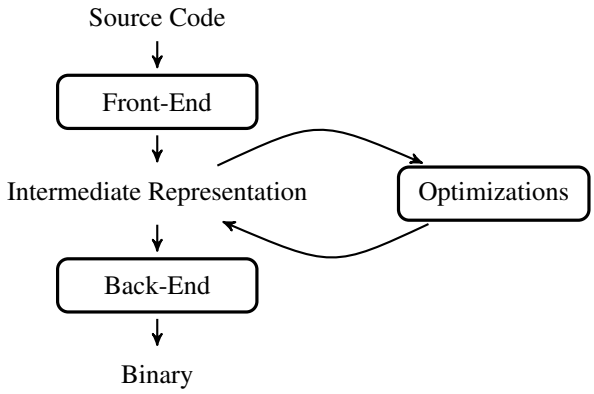


Figure 4.1: Typical Compilation Process

heap is not considered as a separate memory since the notion of a heap only exists when there is some form of memory allocation system available (e.g. provided by the standard library or the operating system). Pointer arithmetics is allowed on pointers to the global memory, but for local variables it is restricted to calculating offsets inside local memories. In *C-IL* every memory access corresponds to dereferencing a left value, where the left value is a pointer to the global memory or a reference to some offset in a local variable. There is an issue regarding *C-IL*, namely it is dependent on the underlying architecture and compiler. So that, *C-IL* is parameterizable to make it applicable to at least the most common cases.

The way we proceed in this chapter is we first introduce Schmalzt's *C-IL* semantics. At last we present the compiler correctness theory for *C-IL* compiled down to the target language VAMP assembly. We assume that the compilation is performed by an optimizing C compiler, which we refer to as the C compiler in question. To be able to present such a theory we will introduce an information about the C compiler in question which can be extracted from its specification. We want to stress that the compiler correctness theory is joint work with S.Schmaltz.

4.1 Semantics

4.1.1 Types

The *C-IL* type system supports:

Primitive Types

Depending on the compiler and underlying architecture, different primitive types may be provided. For a given instance of *C-IL*, the *set of primitive types* $\mathbb{T}_P$ must be defined as a finite subset of the *set of possible primitive types* $\mathbb{T}_{PP}$.

The set of possible primitive types $\mathbb{T}_{PP}$ is defined as follows:

- $n \in \mathbb{N} \wedge n \in \{8, 16, 32, 64\} \Rightarrow \mathbf{i}n \in \mathbb{T}_{PP}$ - signed integer types of size n-bit
- $n \in \mathbb{N} \wedge n \in \{8, 16, 32, 64\} \Rightarrow \mathbf{u}n \in \mathbb{T}_{PP}$ - unsigned integer types of size n-bit
- **void** $\in \mathbb{T}_{PP}$

Note that a primitive type size in bits must be a multiple of 8.

In *C-IL* there is no direct support of boolean type, like in any C. This type is implicitly modeled as one of the primitive types of *C-IL*.

Complex Types

We inductively define the set of complex types $\mathbb{T}$ as follows:

- Primitive types: $t \in \mathbb{T}_P \Rightarrow t \in \mathbb{T}$
- Pointers: $t \in \mathbb{T} \Rightarrow \mathbf{ptr}(t) \in \mathbb{T}$
- Array types: $t \in \mathbb{T} \ \wedge \ n \in \mathbb{N} \Rightarrow \mathbf{array}(t, n) \in \mathbb{T}$
- Function pointers: $t, t_1, \ldots, t_n \in \mathbb{T} \Rightarrow \mathbf{funptr}(t, [t_1, \ldots, t_n]) \in \mathbb{T}$
- Struct types: $t_C \in \mathbb{T}_C \Rightarrow \mathbf{struct}\ t_C \in \mathbb{T}$

Here, $\mathbb{T}_C$ is a set of *struct type names*. This set is defined individually for each *C-IL*-program we consider. A struct type itself is a type that consists of so-called fields modeled by the *set of field names* $\mathbb{F}$. This set contains all possible field names. A struct is always identified by a unique name, the *composite type name*. These are from the set of composite type names $\mathbb{T}_C$. The struct type declarations are part of the *C-IL* program declaration which we examine later on.

Type Predicates We define predicates *is-ptr*?, *is-funptr*?, and *is-array*? which indicate whether a given type represents either a pointer, an array or function pointer, respectively.

$$\textit{is-ptr}? :: \mathbb{T} \to \textit{bool} \qquad \textit{is-array}? :: \mathbb{T} \to \textit{bool} \qquad \textit{is-funptr}? :: \mathbb{T} \to \textit{bool}$$

$$\frac{\exists t'. \quad t = \mathbf{ptr}(t')}{\textit{is-ptr}?(t)} \qquad \frac{\exists t', n'. \quad t = \mathbf{array}(t', n'))}{\textit{is-array}?(t)} \qquad \frac{\begin{array}{c}\exists t, t_1, \ldots, t_n, n'. \\ t = \mathbf{funptr}(t, [t_1, \ldots, t_n])\end{array}}{\textit{is-funptr}?(t)}$$

4.1.2 Values

In *C-IL* values consist of an actual value and a type information. The value is represented as a byte-string The type tells us in what way the value needs to be interpreted. We define the set of values *val* as the union of the following sets:

$$\textit{val} \quad \stackrel{\text{def}}{=} \quad \{\textit{val}_{\mathbf{u}k} \cup \textit{val}_{\mathbf{i}k} \mid k \in \mathbb{N}\} \cup \textit{val}_{\mathbf{struct}} \cup \textit{val}_{\mathbf{ptr}} \cup \textit{val}_{\mathbf{funptr}} \cup \textit{val}_{\mathbf{fun}} \cup \textit{val}_{\mathbf{lref}}$$

where definitions of the aforementioned sets are:

- Set of primitive values:
  - $n \in \mathbb{N} \wedge b \in \mathbb{B}^n \Rightarrow \mathbf{val}(b, \mathbf{u}n) \in \textit{val}_{\mathbf{u}n}$
  - $n \in \mathbb{N} \wedge b \in \mathbb{B}^n \Rightarrow \mathbf{val}(b, \mathbf{i}n) \in \textit{val}_{\mathbf{i}n}$

  where $n \in \{8, 16, 32, 64\}$ is the number of bits.

  We introduce functions $\textit{val2Int} :: \textit{val} \mapsto \mathbb{Z}$ and $\textit{val2Nat} :: \textit{val} \mapsto \mathbb{N}$ which convert *C-IL* values of signed and unsigned integer types to integer and natural numbers, respectively. We omit their definitions due to their simplicity.

- Set of structs:
  - $t_C \in \mathbb{T}_C \wedge B \in (\mathbb{B}^8)^* \Rightarrow \mathbf{val}(B, \mathbf{struct}\ t_C) \in val_{\mathbf{struct}}$
- Set of pointers[2]:
  - $a \in \mathbb{B}^{size_{ptr}} \wedge t \in \mathbb{T} \wedge (is\text{-}ptr?(t) \vee is\text{-}array?(t)) \Rightarrow \mathbf{val}(a, t) \in val_{\mathbf{ptr(t)}} \subseteq val_{\mathbf{ptr}}$
  - $a \in \mathbb{B}^{8size_{ptr}} \Rightarrow \mathbf{val}(a, \mathbf{fptr}) \in val_{\mathbf{funptr}}$

  Here, for pointers the type information is the one from the pointer itself and not the type of the value the pointer points to. Note that array values are also pointers, thus, we allow them to occur as pointer values. (They will be used synonymous to a pointer to the first element of the array in expression evaluation). Note that, in the case of function pointers, to define operational semantics, we do not attach the type of the function pointed to to the value of the function pointer since we can get this information easily from the function table which we will define in this chapter later on. Instead, we just use **fptr** to denote that this value represents some function pointer (which can then be looked up in the function table).
- Set of local references:
  - $v \in \mathbb{V} \wedge o \in \mathbb{N} \wedge i \in \mathbb{N} \wedge t \in \mathbb{T} \wedge (is\text{-}ptr?(t) \vee is\text{-}array?(t)) \Rightarrow \mathbf{lref}((v, o), i, t) \in val_{\mathbf{lref}}$
    where $\mathbb{V}$ is the set of variable names.

  Note that we differentiate between pointers to the global memory and pointers to local memories (on the stack) here. The latter ones we call *local references* and they can only occur as intermediate values during expression evaluation or as return destinations. A local reference contains information about the local variable name v, byte-offset in that variable o, and the stack frame number i it belongs to. Worth to note that function pointers are never local references.
- Set of symbolic function values:
  - $f \in \mathbb{F}_{name} \Rightarrow \mathbf{fun}(f) \in val_{\mathbf{fun}}$

  In a *C-IL*-program, inline functions may occur or the function pointer address may not be defined in the context for a given function for other reasons. Obviously, then a function pointer value cannot be computed during expression evaluation. Thus, for these cases we introduce the symbolic value standing for the function. This is used in the operational semantics to call such functions.

4.1.3 Expressions

We inductively define the set of expressions $\mathbb{E}$ as follows:

- Constants: $c \in val \Rightarrow c \in \mathbb{E}$
- Variable names: $v \in \mathbb{V} \Rightarrow v \in \mathbb{E}$
- Function names: $f \in \mathbb{F}_{name} \Rightarrow f \in \mathbb{E}$

[2] $size_{ptr} \in \mathbb{N}$ - is a constant describing the size of a pointer in bytes and that depends on the compiler.

- Binary operation on expressions: $e_0, e_1 \in \mathbb{E} \wedge \oplus \in \mathbb{O}_2 \Rightarrow (e_0 \oplus e_1) \in \mathbb{E}$.

  Here we assume that the C compiler translates C code to *C-IL* representation in such a fashion that all values passed to operators are correctly typed (where addition of pointer with any unsigned integer or integer type value counts as correctly typed). The set of binary operators $\mathbb{O}_2$ is

  $$\mathbb{O}_2 \stackrel{\text{def}}{=} \{+, -, *, /, \%, <<, >>, <, >, <=, >=, ==, !=, \ \&\ , |, \hat{}, \ \&\&\ , ||\}$$

- Unary operation on expression: $e \in \mathbb{E} \wedge \ominus \in \mathbb{O}_1 \Rightarrow \ominus e \in \mathbb{E}$. The set of unary operators $\mathbb{O}_1$ is

  $$\mathbb{O}_1 \stackrel{\text{def}}{=} \{-, \sim, !\}$$

- Ternary operator: $e, e_0, e_1 \in \mathbb{E} \Rightarrow (e\ ?\ e_0 : e_1) \in \mathbb{E}$

- Type cast: $t \in \mathbb{T}, e \in \mathbb{E} \Rightarrow (t)e \in \mathbb{E}$

- Dereferencing pointer: $e \in \mathbb{E} \Rightarrow *e \in \mathbb{E}$

- Address of expression: $e \in \mathbb{E} \Rightarrow \&e \in \mathbb{E}$

- Field access: $e \in \mathbb{E} \wedge f \in \mathbb{F} \Rightarrow (e).f \in \mathbb{E}$

- Size of type: $t \in \mathbb{T} \Rightarrow$ **sizeof**$(t) \in val$

- Size of expression: $e \in \mathbb{E} \Rightarrow$ **sizeof**$(e) \in val$

- Offset in type: $t_c \in \mathbb{T}_C \wedge f \in \mathbb{F} \Rightarrow$ **offsetof**$(t_c, f) \in val$

Note, we do not support array access and pointer field access here directly, instead, they simply are translated:

- $a[k]$ to $*(a + k)$

- $a \rightarrow b$ to $(*a).b$

4.1.4 Statements

We define the set of *C-IL*-statements $\mathbb{S}$ as follows:

- Assignment: $e_0, e_1 \in \mathbb{E} \Rightarrow e_0 = e_1 \in \mathbb{S}$,

- Goto: $l \in \mathbb{N} \Rightarrow$ **goto** $l \in \mathbb{S}$,

- If-Not-Goto: $e \in \mathbb{E} \ \wedge \ l \in \mathbb{N} \Rightarrow$ **ifnot** e **goto** $l \in \mathbb{S}$,

- Function Call: $e_0, e \in \mathbb{E}, E \in \mathbb{E}^* \Rightarrow e_0 =$ **call** $e(E) \in \mathbb{S}$,

- Procedure Call: $e \in \mathbb{E}, E \in \mathbb{E}^* \Rightarrow$ **call** $e(E) \in \mathbb{S}$,

- Return: $e \in \mathbb{E} \Rightarrow$ **return** $e \in \mathbb{S}$ and **return** $\in \mathbb{S}$.

4.1.5 Program

A *C-IL*-program π is represented by record type $Prog_{IL}$ defined as follows:

◄ DEFINITION 4.1
C-IL Program

$$Prog_{IL} \stackrel{\text{def}}{=} \{\mathcal{V}_G :: (\mathbb{V} \times \mathbb{T})^*, T_F :: \mathbb{T}_C \to (\mathbb{F} \times \mathbb{T})^*, \mathcal{F} :: \mathbb{F}_{name} \to FunT\}.$$

where the purpose of components of this record is:

- $\mathcal{V}_G$ - a list of global variable declarations,
- T_F - a type table for struct types. Given a composite type name t_C, the function returns the list of field declarations for that struct type. A field declaration is a pair of field name and type. In case there is no type declared for composite type name t_C, the function returns the empty list [].
- $\mathcal{F}$ - a function table that is represented as a partial function from the set of function names $\mathbb{F}_{name}$ to function table entries. A function table entry is represented by record type $FunT$ that will be defined now.

Function table entry A function table entry is defined as follows:

DEFINITION 4.2 ►
C-IL Function Declaration

$$FunT \stackrel{\text{def}}{=} \{rettype :: \mathbb{T}, npar :: \mathbb{N}, \mathcal{V} : (\mathbb{V} \times \mathbb{T})^*, \mathcal{P} : (\mathbb{S}^* \cup \{\textbf{extern}\})\}.$$

where the purpose of components of that record type is:

- *rettype* - a return value type,
- *npar* - a number of input parameters,
- $\mathcal{V}$ - a parameter and local variable declaration list, in which the first *npar* elements are parameters,
- $\mathcal{P}$ - a function body represented as the list of *C-IL* statements. If function is not defined within a *C-IL* program, then it is marked by a special keyword `extern`. It means that the function is implemented outside the *C-IL* program, e.g. in an assembly program.

4.1.6 System Parameters

C-IL is such a low-level language that its semantics cannot be fully specified without knowledge about the system parameters it runs under. These system parameters depend on the compiler itself as well on the underlying architecture. In this thesis we are interested in the following system parameters grouped in record $System_{IL}$:

- *endianness* :: {**little**, **big**} - indicates the endianness of the underlying architecture, i.e. the order in which bytes are stored in memory,
- $G_{adr} :: \mathbb{V} \to val_{\textbf{ptr}}$ - returning the address for a given global variable name.
- $\mathcal{F}_{adr} :: \mathbb{F}_{name} \rightharpoonup val_{\textbf{ptr}}$ - is a partial mapping from a function name to a function pointer address. Note that for any given program, this mapping is injective and only yields addresses from the code range. For functions declared `extern` and for inline functions this function need not be defined.

- $size_{ptr} :: \mathbb{N}$ - size of the pointer type in bytes,
- $size_{struct} :: \mathbb{T}_C \to \mathbb{N}$ - size of struct types in bytes,
- $offset :: \mathbb{T}_C \times \mathbb{F} \rightharpoonup \mathbb{N}$ - returning the byte-offset of a field in a struct type,
- $cast :: val \times \mathbb{T} \to val$ - which performs type cast for values,
- $size\_t :: \mathbb{T}_P$ - the type of the value returned by the `sizeof`-operator,

Size of Types

For each type, there is an associated number that describes the number of bytes needed to store a value of that type in memory, the *size* of the type. We define the function $size :: \mathbb{T} \times System_{IL} \to \mathbb{N}$ that returns the size of a given *C-IL* type t that depends on given system parameters θ.

$$size_\theta(t) = \begin{cases} \frac{k}{8} & t = \mathbf{i}k \vee t = \mathbf{u}k \\ \theta.size_{ptr} & is\text{-}ptr?(t) \vee is\text{-}funptr?(t) \\ n \cdot size(t') & t = \mathbf{array}(t', n) \\ \theta.size_{struct}(t_C) & t = \mathbf{struct}\ t_C \end{cases}$$

4.1.7 Configurations

C-IL configurations are represented by record type C_{IL} that consists of a byte-addressable memory ($\mathcal{M}$) and an abstract stack (*stack*), where the latter one is represented as a list of *C-IL* stack frames. Obviously, we do not need to model a no-stack in *C-IL* as it was done in the μ_{ASM} semantics examined in Chapter 3.

$$C_{IL} \stackrel{\text{def}}{=} \{\mathcal{M} :: \mathbb{B}^{size_{ptr}} \mapsto \mathbb{B}^8, stack :: frame^*_{C\text{-}IL}\}$$

◀ DEFINITION 4.3
C-IL Configuration

A stack frame in *C-IL* corresponds to a function (procedure) which has not yet been terminated with return, like in μ_{ASM}. It is represented by record type $frame_{C\text{-}IL}$ that is defined as follows:

$$frame_{C\text{-}IL} \stackrel{\text{def}}{=} \{\mathcal{M}_\mathcal{E} :: \mathbb{V} \mapsto (\mathbb{B}^8)^*, rds :: val_{\mathbf{ptr}} \cup val_{\mathbf{lref}} \cup \{\bot\}, f :: \mathbb{F}_{name}, loc :: \mathbb{N}\}$$

◀ DEFINITION 4.4
C-IL Stack Frame

where the components of this record are:

- $\mathcal{M}_\mathcal{E}$ is a local variable environment mapping variable names to local byte-offset-addressable memories (represented as list of bytes).
- rds is a return value destination describing where the result value returned to the current frame has to be stored. It is either a pointer to where the return value of the function is going to be stored, a local reference to a variable of the stack frame "below" at which offset the return value has to be written to, or $\bot$ denoting the absence of a return destination.
- f and loc is a function name and a statement location counter, respectively. Like it was done in μ_{ASM}, these two components describe the control flow of a *C-IL* program.

Memory Semantics

On the one hand we have byte-addressable memories, on the other we have typed values. The function $read^{\theta}(c_{IL}, a)$ derefences a given pointer value v in a given configuration c_{IL} and returns it.

$$read^{\theta} :: C_{IL} \times (val_{\mathbf{ptr}} \cup val_{\mathbf{lref}}) \mapsto val$$

The function $write^{\theta}(c_{IL}, a, v)$ writes a given value v to a given address a in memory of a given configuration c_{IL}.

$$write^{\theta} :: C_{IL} \times val \times (val_{\mathbf{ptr}} \cup val_{\mathbf{lref}}) \mapsto C_{IL}$$

To specify the effect, similar functions $read^{\theta}_{\mathcal{E}}$ and $write^{\theta}_{\mathcal{E}}$ are provided to read and write a local variable/parameter (identified by variable name) from a stack frame. They have the following signatures:

$$read^{\theta}_{\mathcal{E}} :: frame_{C\text{-}IL} \times \mathbb{V} \to val$$

$$write^{\theta}_{\mathcal{E}} :: frame_{C\text{-}IL} \times \mathbb{V} \times val \to frame_{C\text{-}IL}$$

Note that, since addresses of local variables are not explicitly modeled (this would either expose stack layout) the *C-IL* semantics carries the limitation that pointers to local variables cannot be stored in memory.

We have omitted their definitions because it is too detailed for this thesis.

4.1.8 Operational Semantics

Expression Evaluation

Expressions are evaluated by a function that returns either a *C-IL*-value or the special value $\perp$ that denotes that the expression cannot be evaluated:

$$[e]^{\pi,\theta}_{c} \in val \cup \{\perp\}$$

Depending on the expression, we may need the complete state, i.e. configuration, program and system parameters, to evaluate it. We omitted its definition as it is too detailed for this thesis.

Statement Execution

In defining the semantics of *C-IL* we will use the following shorthand notation:

$$c_{IL}.f_i \stackrel{\text{def}}{=} c_{IL}.stack[i].f$$

$$c_{IL}.loc_i \stackrel{\text{def}}{=} c_{IL}.stack[i].loc$$

As shorthands for the members of the top stack frame in a *C-IL*-configuration:

- $c_{IL}.\mathcal{M}_{\mathcal{E}_{top}} \stackrel{\text{def}}{=} \mathbf{hd}(c_{IL}.stack).\mathcal{M}_{\mathcal{E}}$
- $c_{IL}.rds_{top} \stackrel{\text{def}}{=} \mathbf{hd}(c_{IL}.stack).rds$
- $c_{IL}.f_{top} \stackrel{\text{def}}{=} \mathbf{hd}(c_{IL}.stack).f$
- $c_{IL}.loc_{top} \stackrel{\text{def}}{=} \mathbf{hd}(c_{IL}.stack).loc$

The next statement to be executed by a *C-IL* machine running some *C-IL* program is identified by the control components of the top stack frame of the stack of this machine. We define the function

$$stmt_{curr} :: frame^*_{C\text{-}IL} \times Prog_{IL} \mapsto \mathbb{S}$$

that takes a *C-IL* stack s (instead of a *C-IL* configuration) and a *C-IL* program π as input and returns the next statement to be executed by a *C-IL* machine whom the stack s belongs to. This function is defined as

$$stmt_{curr}(s, \pi) \stackrel{\text{def}}{=} \pi.\mathcal{F}(\mathbf{hd}(s).f).\mathcal{P}[\mathbf{hd}(s).loc]$$

◀ DEFINITION 4.5
Current Statement

The function $top :: C_{IL} \mapsto \mathbb{N}$ returns the index of the top stack frame in a given configuration c_{IL}:

$$top(c_{IL}) \stackrel{\text{def}}{=} |c_{IL}.stack| - 1$$

◀ DEFINITION 4.6
Top Stack Frame Index

Further, we define the functions

$$inc_{loc} :: C_{IL} \mapsto C_{IL}$$

$$inc_{loc}(c_{IL}) \stackrel{\text{def}}{=} c_{IL}[stack := \mathbf{hd}(c_{IL}.stack)[loc := c_{IL}.loc_{top} + 1] \circ \mathbf{tl}(c_{IL}.stack)]$$

which increments the location of the top stack frame of a given *C-IL* configuration c_{IL}, and

$$set_{loc} :: C_{IL} \times \mathbb{N} \mapsto C_{IL}$$

$$set_{loc}(c_{IL}, l) \stackrel{\text{def}}{=} c_{IL}[stack := \mathbf{hd}(c_{IL}.stack)[loc := l] \circ \mathbf{tl}(c_{IL}.stack)]$$

which sets the location of the top stack frame to location l, and

$$drop_{frame} :: C_{IL} \mapsto C_{IL}$$

$$drop_{frame}(c_{IL}) \stackrel{\text{def}}{=} c_{IL}[stack := \mathbf{tl}(c_{IL}.stack)]$$

which removes the top stack frame from a given *C-IL* configuration c_{IL}, and

$$set_{rds} :: C_{IL} \times val \mapsto C_{IL}$$

$$set_{rds}(c_{IL}, v) \stackrel{\text{def}}{=} c_{IL}[stack := \mathbf{hd}(c_{IL}.stack)[rds := v] \circ \mathbf{tl}(c_{IL}.stack)]$$

which sets the return destination of the top stack frame to a given value v.

The predicate **zero** :: $val \mapsto bool$ indicates whether a given value represents a zero value.

The predicate *is-function* :: $val_{\mathbf{ptr}} \times \mathbb{F}_{name} \times System_{IL} \mapsto bool$ indicates whether a given function pointer address corresponds to a given function name

$$\frac{\theta.\mathcal{F}_{adr}(v) = f}{\textit{is-function}(v, f, \theta)}$$

4.1.9 Transition Function

We denote a *C-IL* configuration c_{IL} making a step to configuration c'_{IL} under the program π in the context θ by

DEFINITION 4.7 ►
C-IL Transition Function

$$\pi, \theta \vdash c_{IL} \rightarrow_{IL} c'_{IL}.$$

It is defined by treating each *C-IL* statement.

Assignment As an effect of such a statement execution the evaluated right hand expression of this statement is stored in the memory of the *C-IL* configuration c_{IL} at address that is the evaluated left hand side expression of this statement. Also, the location counter of the top stack frame of a *C-IL* configuration gets increased by one. Formally,

$$\frac{stmt_{curr}(c_{IL}.stack, \pi) = e_0 = e_1}{\pi, \theta \vdash c_{IL} \rightarrow_{IL} inc_{loc}(write^{\theta}(c_{IL}, [\&e_1]^{\theta,\pi}_{c_{IL}}, [e_0]^{\theta,\pi}_{c_{IL}}))}$$

Goto This statement performs a so-called unconditional jump to the target location and it is defined as follows:

$$\frac{stmt_{curr}(c_{IL}.stack, \pi) = \textbf{goto } l}{\pi, \theta \vdash c_{IL} \rightarrow_{IL} set_{loc}(c_{IL}, l)}$$

If-Not-Goto In comparison to the goto statement this statement performs a conditional jump. That is, there are two cases: (i) if the evaluated logical test expression e of the statement is equal zero, then the same happens as it just were a goto statement with the same target location,

$$\frac{stmt_{curr}(c_{IL}.stack, \pi) = \textbf{ifnot } e \textbf{ goto } l \qquad \textbf{zero}([e]^{\theta,\pi}_{c_{IL}})}{\pi, \theta \vdash c_{IL} \rightarrow_{IL} set_{loc}(c_{IL}, l)}$$

(ii) in case the evaluated test expression e fails, i.e. it is not equal zero, then the execution of such a statement does not change anything except for the location counter of the top stack frame of the configuration c_{IL} which gets increased by one. That is, it operates like a *no-operation* statement. Formally,

$$\frac{stmt_{curr}(c_{IL}.stack, \pi) = \textbf{ifnot } e \textbf{ goto } l \qquad \neg\textbf{zero}([e]^{\theta,\pi}_{c_{IL}})}{\pi, \theta \vdash c_{IL} \rightarrow_{IL} inc_{loc}(c_{IL})}$$

Call A call statement is treated here if it is not external and it either is a function or procedure call. The function name string identifer is obtained from the expression e of the call statement (the expression e represents the function f). As a result of the call statement execution a new stack frame is created and is put on the top of the stack. Also, the location counter of the original top stack frame gets increased by one. So that, the execution will resume at the next statement after the call statement whenever the called function(procedure) returns.

$$\frac{\begin{array}{c} stmt_{curr}(c_{IL}.stack, \pi) \in \{\textbf{call } e(E), e_0 = \textbf{call } e(E)\} \\ is\text{-}function([e]^{\pi,\theta}_{c_{IL}}, f, \theta) \qquad \pi.\mathcal{F}(f).\mathcal{P} \neq \textbf{extern} \\ call_{frame}(c_{IL}, f, E, frame_{new}) \end{array}}{\pi, \theta \vdash c_{IL} \rightarrow_{IL} set_{rds}(c_{IL}, [\&e_0]^{\theta,\pi}_{c})[stack := frame_{new} \circ inc_{loc}(c_{IL}).stack]}$$

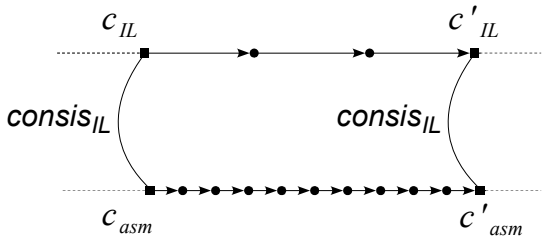


Figure 4.2: C_{IL} Compiler Consistency at IO-points

The new stack frame is chosen nondeterministically according to the following constraints captured in the predicate $call_{frame}(c_{IL}, f, E, frame_{new})$:

$$\forall 0 \leq i < npar: \quad read^{\theta}_{\mathcal{E}}(frame_{new}, v_i) = [E[i]]^{\theta,\pi}_{c}$$

$$\forall npar \leq i < \mathbf{len}(\mathcal{V}): \quad \mathbf{len}(frame_{new}.\mathcal{M}_{\mathcal{E}}(v_i)) = size(t_i)$$

$$frame_{new}.loc = 0, \quad frame_{new}.f = f, \quad frame_{new}.rds = \bot$$

Here, $npar = \pi.\mathcal{F}(f).npar$ is the number of input parameters of the function f, $\mathcal{V} = \pi.\mathcal{F}(f).\mathcal{V}$ is the declaration list of parameters and local variables of the function f, and $(v_i, t_i) = \mathcal{V}[i]$ is the i-th declaration in that declaration list. Note that we only place a strict constraint on the parameter values. The initial content of local variables is chosen nondeterministically with appropriate size for the declared type.

Return There are two return statements: (i) return from a function call, i.e. return with result, and (ii) return from a procedure call, i.e. return without result. In the first case the evaluated result expression $[e]^{\theta,\pi}_{c_{IL}}$ of the return statement is stored in the memory at the address identified by the return destination component $drop_{frame}(c_{IL}).rds_{top}$. Also, for both statements the top stack frame of a *C-IL* configuration is removed.

◄ DEFINITION 4.8
Return With Result

$$\frac{stmt_{curr}(c_{IL}.stack, \pi) = \mathbf{return}\ e \qquad drop_{frame}(c_{IL}).rds_{top} \neq \bot}{\pi, \theta \vdash c_{IL} \rightarrow_{IL} write^{\theta}(drop_{frame}(c_{IL}), [e]^{\theta,\pi}_{c_{IL}}, drop_{frame}(c_{IL}).rds_{top})}$$

◄ DEFINITION 4.9
Return Without Result

$$\frac{stmt_{curr}(c_{IL}.stack, \pi) = \mathbf{return}}{\pi, \theta \vdash C\text{-}IL \rightarrow_{IL} drop_{frame}(c_{IL})}$$

4.2 Compiler Correctness Theory

In this section we introduce C_{IL} simulation relation, called *compiler consistency*, for an optimizing C compiler. The simulation relation $consis_{IL}(c_{IL}, info_{IL}, c_{ASM})$ states that the VAMP assembly configuration c_{ASM} encodes the *C-IL* configuration c_{IL} via the *C-IL* static information $info_{IL}$ at IO points. An IO point[3] is a point indicating

[3] the notion of IO points was introduced in [DPS09]

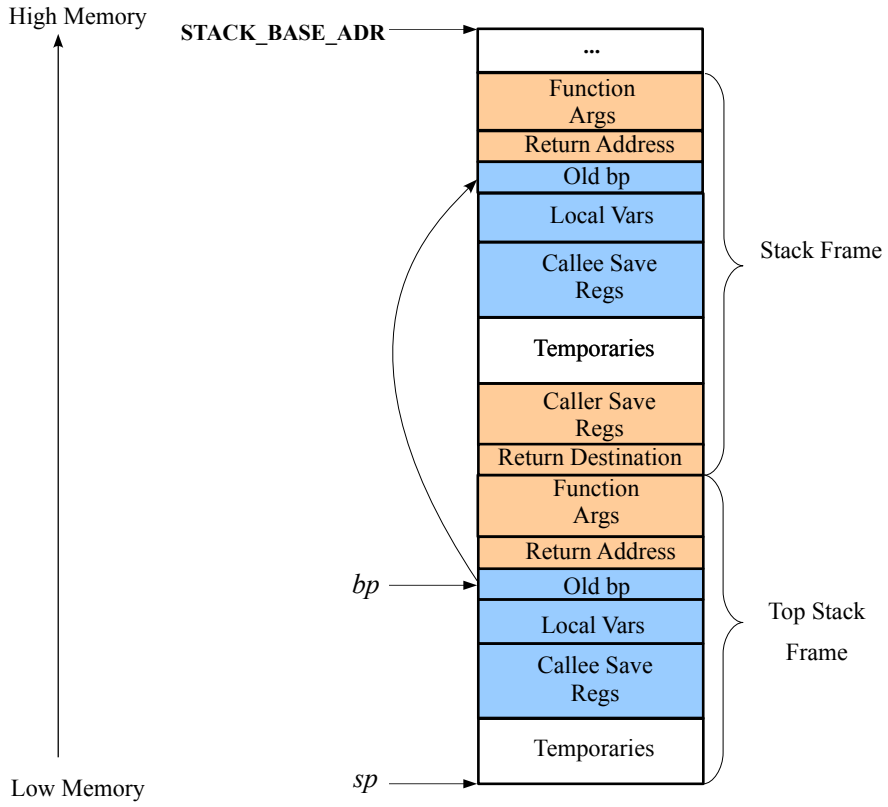


Figure 4.3: *C-IL* Stack Layout

a statement in the execution sequence of statements of the source code at which the execution effect of preceding instructions in the compiled code with optimizations is the same as it had been compiled without optimizations. IO points we are interested in are function and external function calls. At this place we assume that the optimizing C compiler in question does not perform interprocedural optimizations. We need such an assumption to guarantee that an IO point like a function call in a source code will not be removed, e.g. inlined, after compiling it to target code. Also, we assume the evaluation of parameters to a call is done within the code generated for this call. So that, any pre-computations for evaluating parameters of a call cannot be done before the code generated for this call. In order to allow for an optimizing C compiler in question to generate these pre-computations we would need to split the call and return statements of the original *C-IL* semantics into the sequence of new statements. By that we obtain a new semantics, called *C-IL*-Minor. That will be a future work for extending the compiler correctness theory.

In order to be able to state the simulation theorem we need at least to agree how the stack layout looks like of the generated code by an optimizing C compiler in question.

4.2.1 Stack Layout

The stack layout we use for *C-IL* is depicted in Figure 4.3. This stack layout differs from the stack layout depicted in Figure 3.3 at some places, but they have the same frame headers and the region containing passed parameters is found at the same place in both stack layouts. This is an important aspect to a technically sound model coupling the μ_{ASM} and *C-IL* languages.

Having fixed the stack layout it is not difficult to figure out what the code generated for a call by the compiler is doing. However, let us go through blocks of a stack frame depicted in 4.3 which are saved at a call. Before a call temporaries are on top of the stack. Temporaries in a *C-IL* stack frame corresponds to lifo in a μ_{ASM} stack frame. The code generated for a call puts first caller-save registers on stack, then it puts the return destination address in case a result value must be returned from the call. Next, input arguments are put on the stack as well as the return address at which the execution will resume when the called function returns. That is what is done by the code generated for a call at the caller side. The code generated for a call at the callee side, called *prologue*, puts first the frame base pointer on the stack, then reserves the space for local variables, and at last stores callee-save registers.

Static Information Describing Compiled *C-IL* Program

Here we introduce the function $code_{IL} :: Prog_{IL} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \mapsto infoT_{IL}$ that takes a program π, a program base address `PROG_BASE_ADR`, a stack base address `STACK_BASE_ADR` and the stack length `STACK_MAX_SIZE` and produces static information $info_{IL}$ describing the compiled *C-IL* program. This static information comprises the following components:

- $info_{IL}.code$: $Instr^*$ is the list of VAMP assembly instructions that represents the compiled *C-IL* program,
- $info_{IL}.code_{ba} : \mathbb{N}$ is the base address of the compiled code,
- $info_{IL}.stack_{ba} : \mathbb{N}$ is the stack base address,
- $info_{IL}.stack_{len} : \mathbb{N}$ is the stack size in words,
- $info_{IL}.code_{ia} : (\mathbb{F}_{name} \times \mathbb{N}) \rightarrow \mathbb{N}$ maps pairs of function names and locations, indicating statements, to base addresses of those compiled statements,
- $info_{IL}.fun_{ba} : \mathbb{F}_{name} \rightarrow \mathbb{N} \cup \{\perp\}$ maps functions (if declared and defined in the *C-IL* program) to their compiled start addresses,
- $info_{IL}.iop : \mathbb{F}_{name} \times \mathbb{N} \rightarrow bool$ indicates whether the statement identified by a given location and function name is an IO point.
- $info_{IL}.lvar_{reg} : \mathbb{V} \times \mathbb{F}_{name} \times \mathbb{N} \rightarrow \mathbb{N} \cup \{\perp\}$ maps a given local variable name that belongs to the function identified by a given function name and a given IO point index to the index of a assembly machine register that contains the value of this local variable.
- $info_{IL}.lvar_{off} : \mathbb{V} \times \mathbb{F}_{name} \rightarrow \mathbb{N}$ maps local variables and function names to offsets where local variable (input parameter) values are stored in stack frames which corresponds to the function names. This offset is relative to the base address of a stack frame.

- $info_{IL}.size_{caller-save} : \mathbb{F}_{name} \times \mathbb{N} \to \mathbb{N}$ returns the number of bytes that will be used on the stack after executing the statement identified by a given pair of the location counter and the function name (cf. Figure 4.4). Here we want to stress that the number of bytes that is used to store caller's data on the stack can differ for the same function call called from different places within the *C-IL* program.

  Here we introduce two lists *caller* and *callee* containing caller- and callee-save GPR indices, respectively, which are compiler dependent. Typically, the concatenation of both lists should contain all GPR indices. But we do not rely anyhow on that.

- $info_{IL}.size_{params} : \mathbb{F}_{name} \to \mathbb{N}$ returns the number of bytes that is occupied by input parameters passed on the stack to a call to the function identified by a given function name (cf. Figure 4.4).

- $info_{IL}.size_{lvars} : \mathbb{F}_{name} \to \mathbb{N}$ returns the number of bytes that is occupied by local variables stored on the stack for a given function.

- $info_{IL}.dist_{bp} : \mathbb{F}_{name} \times \mathbb{N} \to \mathbb{N}$ is a mapping that returns the number of bytes occupied in a stack frame by a given function at the statement indicated by a given location, where this statement must be an IO point (cf. Figure 4.4).

Note that we do not present the definition of the function $code_{IL}$ as it completely depends on the specification of an optimizing C compiler which is another topic that is not a part of this thesis.

Before we present the *C-IL* compiler consistency we first introduce a few auxiliary functions. We first define the function

$$dist_{IL}^{bp} :: \mathbb{N} \times frame_{C\text{-}IL}^{*} \times infoT_{IL} \mapsto \mathbb{N}$$

computing the distance (number of bytes) between stack frame base addresses of the i-th and $(i+1)$-th stack frames from a given list of stack frames s with respect to the information $info_{IL}$ about the compiled code. In the definition of this function we make case distinctions on whether the i-th stack frame from the list of stack frames s is the top one or not.

- If the i-th stack frame is the top one, then the distance returned by this function is identified by the static information component $info_{IL}.dist_{bp}$ (cf Figure 4.4).

- Else, i.e. i-th stack frame is not the top one, the distance is computed as the sum of (cf. Figure 4.4):

  - the stack frame size before a call that created the $(i+1)$-th stack frame. Recall that after executing a call statement in the *C-IL* semantics the location counter of the stack frame corresponding to caller is incremented. So that, when the called function terminates the execution resumes at the next statement. Because of this incrementing in "advance", in the definition of the function $dist_{IL}^{bp}$, the location counter of the i-th non-top stack frame is adjusted by decrementing it so that it points to a call statement(whose execution created the $(i+1)$-th stack frame) that is an IO point.
  - the caller-save block size in the i-th stack frame,
  - 4 bytes containing a return destination. These four bytes are added to the distance in case the return destination is defined for the i-th stack frame,

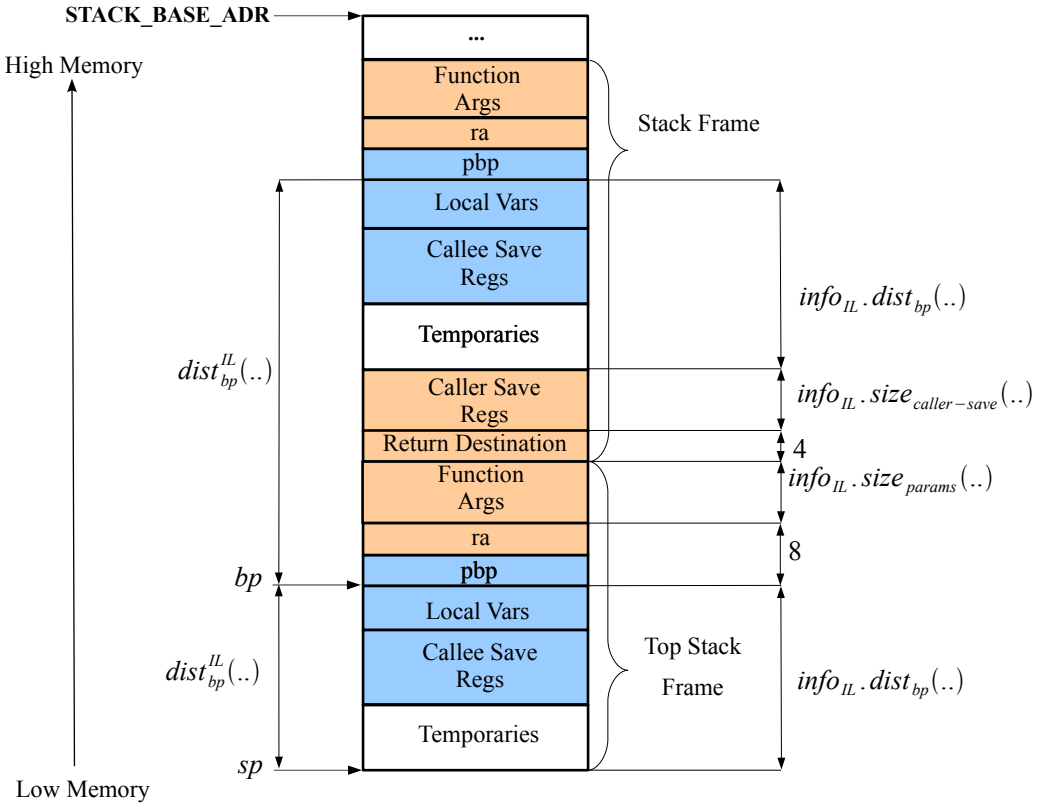


Figure 4.4: *C-IL* Stack Layout

– the size of a block containing the parameters passed to a function call corresponding to the $(i+1)$-th stack frame,

– 4 bytes representing the return address of the frame header corresponding to the $(i+1)$-th stack frame, and

– 8 bytes representing the frame header corresponding to the $(i+1)$-th stack frame

So the function $dist^{bp}_{IL}$ is defined as

$$dist^{bp}_{IL}(i, s, info_{IL}) \stackrel{\text{def}}{=}$$

◄ DEFINITION 4.10
Distance Between Frame Base Addresses

$$\begin{cases} info_{IL}.dist_{bp}(s[i].f, s[i].loc) & \text{if } i = top(s) \\ \\ info_{IL}.dist_{bp}(s[i].f, s[i].loc - 1) & \\ + info_{IL}.size_{caller-save}(s[i].f, s[i].loc - 1) & \\ + \begin{cases} 4 & \text{if } s[i].rds \neq \bot \\ 0 & \text{otherwise} \end{cases} & \text{otherwise} \\ + info_{IL}.size_{params}(s[i+1].f) & \\ + 8 & \end{cases}$$

where f is a function name obtained from the *C-IL call*-statement identified by the function name $s[i].f$ and the location counter $s[i].loc - 1$ in a *C-IL* program.

Next we define the function

$$frame_{IL}^{ba} :: \mathbb{N} \times C_{IL} \times infoT_{IL} \mapsto \mathbb{N}$$

that computes the absolute memory address of the i-th stack frame of a given *C-IL* configuration c_{IL}, where the stack base address is identified by the component $stack_{ba}$ of a given static information $info_{IL}$. Recall that the stack grows downwards.

DEFINITION 4.11 ►
Frame Base Address

$$frame_{IL}^{ba}(i, c_{IL}, info_{IL}) \stackrel{\text{def}}{=} info_{IL}.stack_{ba} - \sum_{j<i} dist_{IL}^{bp}(j, c_{IL}.stack[j], info_{IL})$$

We define two functions with the same signature which read the return addresses and previous stack pointers from stack frame headers. Let i be a stack frame number, m the memory of an assembly configuration, $info_{IL}$ a static information. Then, the following functions read frame header information from the i-th stack frame of a given *C-IL* configuration c_{IL}.

DEFINITION 4.12 ►
Reading frame headers

$$stack_{IL}^{pbp}(i, c_{IL}, m, info_{IL}) \stackrel{\text{def}}{=} m_{word}(frame_{IL}^{ba}(i, c_{IL}, info_{IL}) + 0)$$

$$stack_{IL}^{ra}(i, c_{IL}, m, info_{IL}) \stackrel{\text{def}}{=} m_{word}(frame_{IL}^{ba}(i, c_{IL}, info_{IL}) + 4)$$

Next we define the function $stack_{IL}^{rds}$ reading the return destination put on the stack at a function(procedure) call that created the $(i+1)$-th stack frame (cf. Figure 4.4).

DEFINITION 4.13 ►
Reading return destination

$$\begin{aligned} stack_{IL}^{rds}(i, c_{IL}, m, info_{IL}) \stackrel{\text{def}}{=}\ & m_{word}(frame_{IL}^{ba}(i+1, c_{IL}, info_{IL}) + 8 \\ +\ & info_{IL}.size_{params}(c_{IL}.stack[i+1].f)) \end{aligned}$$

4.2.2 Simulation Relation

We define a simulation relation $consis_{IL}$ between states of *C-IL* and VAMP assembly machines.

The simulation relation $consis_{IL}(c_{IL}, info_{IL}, c_{ASM})$ states that the assembly configuration c_{ASM} encodes the C-IL configuration c_{IL} via the static information $info_{IL}$. It comprises (i) control consistency $consis_{IL}^{control}(c_{IL}, info_{IL}, c_{ASM})$, and (ii) data consistency $consis_{IL}^{data}(c_{IL}, info_{IL}, c_{ASM})$.

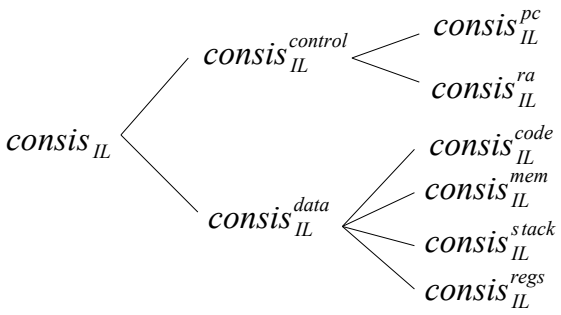


Figure 4.5: C-IL Compiler Consistency

◄ DEFINITION 4.14
C-IL Compiler Consistency

$$\frac{consis_{IL}^{control}(c_{IL}, info_{IL}, c_{ASM}) \qquad consis_{IL}^{data}(c_{IL}, info_{IL}, c_{ASM})}{consis_{IL}(c_{IL}, info_{IL}, c_{ASM})}$$

Control Consistency

Control consistency comprises program counter consistency $consis_{IL}^{pc}(c_{IL}, info_{IL}, c_{ASM})$ and return address consistency $consis_{IL}^{ra}(c_{IL}, info_{IL}, c_{ASM})$.

◄ DEFINITION 4.15
Control Consistency

$$\frac{consis_{IL}^{pc}(c_{IL}, info_{IL}, c_{ASM}) \qquad consis_{IL}^{ra}(c_{IL}, info_{IL}, c_{ASM})}{consis_{IL}^{control}(c_{IL}, info_{IL}, c_{ASM})}$$

The program counter consistency states that the assembly configuration's program counters point to the current *C-IL* instruction.

◄ DEFINITION 4.16
Program Counter Consistency

$$consis_{IL}^{pc} :: C_{IL} \times infoT \times C_{\text{ASM}} \mapsto \mathbb{B}$$

$$\frac{\begin{array}{c} c_{ASM}.pc = c_{ASM}.dpc + 4 \\ c_{ASM}.dpc = info_{IL}.code_{ia}(c_{IL}.f_{top}, c_{IL}.loc_{top}) \end{array}}{consis_{IL}^{pc}(c_{IL}, info_{IL}, c_{ASM})}$$

The return address consistency states that return addresses of all stack frames point directly behind the code of call instructions which generated these stack frames.

◄ DEFINITION 4.17
Return Address Consistency

$$\frac{\begin{array}{l} \forall 0 < i < |c_{IL}.stack|.\ c_{ASM}.m_{word}(stack_{IL}^{ra}(i, c_{IL}, c_{ASM}.m, info_{IL}) = \\ \qquad\qquad info_{IL}.code_{ia}(c_{IL}.f_{i-1}, c_{IL}.loc_{i-1}) \end{array}}{consis_{IL}^{ra}(c_{IL}, info_{IL}, c_{ASM})}$$

Data Consistency

The data consistency comprises code consistency $consis_{IL}^{code}$, memory consistency $consis_{IL}^{mem}$, register consistency $consis_{IL}^{regs}$, and stack consistency $consis_{IL}^{stack}$.

DEFINITION 4.18 ► Data Consistency

$$\frac{\begin{array}{cc} consis_{IL}^{code}(c_{IL}, c_{ASM}) & consis_{IL}^{mem}(c_{IL}, info_{IL}, c_{ASM}) \\ consis_{IL}^{stack}(c_{IL}, info_{IL}, c_{ASM}) & consis_{IL}^{regs}(c_{IL}, info_{IL}, c_{ASM}) \end{array}}{consis_{IL}^{data}(c_{IL}, info_{IL}, c_{ASM})}$$

The code consistency requires that the compiled code is stored at address *progbase* in the assembly configuration.

DEFINITION 4.19 ► Code Consistency

$$\frac{\forall i < |info_{IL}.code|.\ info_{IL}.code[i] = int\text{-}to\text{-}instr(c_{ASM}.m_{word}(info_{IL}.code_{ba} + 4 \cdot i))}{consis_{IL}^{code}(c_{IL}, info_{IL}, c_{ASM})}$$

The memory consistency states that the memory content of both machines is the same. Since the stack is abstracted in C-IL we do not state memory equality for the region where the stack resides. As well we do not state the memory equality for the region where the compiled code is allocated.

DEFINITION 4.20 ► Memory Consistency

$$\frac{\forall a \in \mathbb{N}_{32}.\ \begin{array}{c} a \notin [info_{IL}.stack_{ba} : info_{IL}.stack_{ba} - info_{IL}.stack_{max-size}) \\ \wedge\ a \notin [info_{IL}.code_{ba} : info_{IL}.code_{ba} + 4 \cdot |info_{IL}.code|) \\ \Longrightarrow \\ c_{IL}.\mathcal{M}(a) = c_{ASM}.m(a) \end{array}}{consis_{IL}^{mem}(c_{IL}, info_{IL}, c_{ASM})}$$

The register consistency argues about the content of the stack pointer and base pointer registers of the assembly configuration. We do not argue about the content of the other registers like GPRs and SPRs. Do not forget that we have already argued about the content of program counter in the program counter consistency before. The stack- and base pointer registers should have values according to their intended meaning from Table 3.1. Namely, GPR bp of the assembly machine c_{ASM} should point to the base address of the top-most stack frame of the C-IL machine c_{IL}. The difference between values stored in GPRs sp and bp of the assembly machine is the distance between frame base addresses for the top stack frame of the machine c_{IL} (cf. Figure 4.4).

DEFINITION 4.21 ► Register Consistency

$$\frac{\begin{array}{c} c_{ASM}.gpr[bp] = frame_{IL}^{ba}(top(c_{IL}), c_{IL}, info_{IL}) \\ c_{ASM}.gpr[sp] = c_{ASM}.gpr[bp] - dist_{IL}^{bp}(top(c_{IL}), c_{IL}.stack, info_{IL}) \end{array}}{consis_{IL}^{regs}(c_{IL}, info_{IL}, c_{ASM})}$$

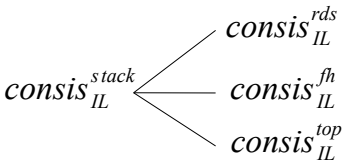


Figure 4.6: *C-IL* Compiler Stack Consistency

The stack consistency comprises return destination consistency $consis_{IL}^{rds}$, frame header consistency $consis_{IL}^{fh}$, and top stack frame consistency $consis_{IL}^{top}$ (cf. Figure 4.6).

◀ DEFINITION 4.22
Stack Consistency

$$\frac{consis_{IL}^{rds}(c_{IL}, info_{IL}, c_{ASM}) \quad consis_{IL}^{fh}(c_{IL}, info_{IL}, c_{ASM}) \quad consis_{IL}^{top}(c_{IL}, info_{IL}, c_{ASM})}{consis_{IL}^{stack}(c_{IL}, info_{IL}, c_{ASM})}$$

The return destination consistency states that for all abstract stack frames (except for the top stack frame) with defined return destinations the memory of an assembly machine contains these return destinations in the corresponding representation.

◀ DEFINITION 4.23
Return Destination Consistency

$$\frac{\begin{array}{c}\forall\ 0 \leq i < |c_{IL}.stack| - 1.\\ c_{IL}.stack[i].rds \neq \bot \longrightarrow\\ stack_{IL}^{rds}(i, c_{IL}, c_{ASM}.m, info_{IL}) = val2Int(read^{\theta}(c_{IL}, c_{IL}.stack[i].rds))\end{array}}{consis_{IL}^{rds}(c_{IL}, info_{IL}, c_{ASM})}$$

The frame header consistency requires the values in the frame headers of the stack frames to be consistent with their intended meaning (cf. Table 3.2). We do not argue about the return addresses in this consistency as it is already covered by the return address consistency.

◀ DEFINITION 4.24
Frame Header Consistency

$$\frac{\begin{array}{c}\forall\ 0 < i < |c_{IL}.stack|.\\ stack_{IL}^{pbp}(i, c_{IL}, c_{ASM}.m, info_{IL}) = frame_{IL}^{ba}(i-1, c_{IL}, info_{IL})\end{array}}{consis_{IL}^{fh}(c_{IL}, info_{IL}, c_{ASM})}$$

The top stack frame consistency states where input parameters and local variables of the function call are saved. The value of an input parameter (local variable) is either saved in the register or put on the stack.

◀ DEFINITION 4.25
Top Stack Frame Consistency

$$\frac{\begin{array}{c}\forall\ i < |c_{IL}.f_{top}.\mathcal{V}|.\\ info_{IL}.lvar_{reg}(v_i, c_{IL}.f_{top}, c_{IL}.l_{top}) = \bot \longrightarrow\\ c_{IL}.\mathcal{M}_{\mathcal{E}top}(v_i) = m_{\text{word}}(c_{ASM}.gpr[bp] + info_{IL}.lvar_{off}(v_i, f_{top}) \wedge\\ info_{IL}.lvar_{reg}(v_i, c_{IL}.f_{top}, c_{IL}.loc_{top}) \neq \bot \longrightarrow\\ c_{IL}.\mathcal{M}_{\mathcal{E}top}(v_i) = c_{ASM}.gpr[info_{IL}.lvar_{reg}(v_i, c_{IL}.f_{top}, c_{IL}.loc_{top}))]\end{array}}{consis_{IL}^{top}(c_{IL}, info_{IL}, c_{ASM})}$$

where $(v_i, t_i) = c_{IL}.f_{top}.\mathcal{V}(i)$ is the i-the declaration in the list of parameters and local variables of the function corresponding to the top most stack frame.

On the one hand we are done with the definition of the *C-IL* compiler consistency, but on the other hand one we are not. The reason why we are still not done with the definition is that some relations are not defined, but it is not possible to define them in *C-IL* as it is. These relations must talk about the content preservation of callee-save registers. The content of registers is not visible in *C-IL* (but it is only true as long as interrupts are not modeled in *C-IL*). In order to be able to talk about the content of registers we add to the record $frame_{C\text{-}IL}$ modeling *C-IL* stack frames a specification(ghost) component, called $gregs$. This newly added component represents a copy of GPRs before a function(procedure) call.

DEFINITION 4.26 ► *C-IL* Stack Frame (with $gregs$)

$$frame_{C\text{-}IL} \stackrel{\text{def}}{=} \{.., gregs :: \mathbb{Z}^*\}$$

So we need to extend the *call*-rule in the *C-IL* semantics with the initialization of the component $gregs$ of the newly created stack frame $frame_{new}$. Since we do not see values of registers in *C-IL* we initialized this component with non-deterministically chosen values so that we can instantiate the corresponding values in the correctness proof of a C compiler later on.

In the following we state the theorem talking about the preservation of callee-save registers in code generated by an optimizing C compiler in question.

THEOREM 4.27 ► $gregs$ Content

Let π be a *C-IL* program, c^0_{ASM} be the initial state of a target machine. Then for any sequence of a target machine: c^0_{ASM}, c^1_{ASM}, ..., there exists a *C-IL* machine sequence c^0_{IL}, c^1_{IL}, ... that does not get stuck and does not produce a stack overflow. Moreover, for any number steps i of a *C-IL* machine there exists the corresponding host machine state $c^{s(i)}_{ASM}$, such that if the current statement to be executed by the *C-IL* machine at step i is:

- a call statement, then the callee-save ghost registers $gregs$ of the new created stack frame of the machine c^{i+1}_{IL} (as an effect of the execution of this call statement) must be the same as the callee-save GPRs of the host machine $c^{s(i)}_{ASM}$ (i.e. before a call),
- a return statement, then the callee-save ghost registers $gregs$ of the top stack frame of the machine c^i_{IL} must be the same as the callee-save GPRs of the machine $c^{s(i+1)}_{ASM}$ (the state corresponding to the state c^{i+1}_{IL} with the executed return statement).

Formally, this theorem is stated as follows:

$$\begin{array}{l}
\exists\, info_{IL}.\ \forall\, (c^i_{ASM})_i, \exists (c^i_{IL})_i. \\
\quad (\forall\, i. \quad c^{i+1}_{ASM} = \delta_{\text{ASM}}(c^i_{ASM}) \\
\qquad \wedge \quad \pi, \theta \vdash c^i_{IL} \rightarrow_{IL} c^{i+1}_{IL} \\
\qquad \wedge \quad \text{the execution of } c_{IL} \text{ does not get stuck} \\
\qquad \wedge \quad \text{there is no stack overflow}) \\
\wedge \quad \exists\, s :: \mathbb{N} \mapsto \mathbb{N}. \\
\quad (\forall\, i.\ info_{IL}.iop(c^i_{IL}.f_{top}, c^i_{IL}.loc_{top})). \\
\quad \forall\, j.\ j \in callee. \\
\quad \wedge\ stmt_{curr}(c^i_{IL}.stack, \pi) \in \{\textbf{call}\ e(E), e_0 = \textbf{call}\ e(E)\} \\
\qquad \longrightarrow\ c^{i+1}_{IL}.stack_{top}.gregs[j] = c^{s(i)}_{ASM}.gpr[j] \\
\quad \wedge\ stmt_{curr}(c^i_{IL}.stack, \pi) \in \{\textbf{return}, \textbf{return}\ e\} \\
\qquad \longrightarrow\ c^i_{IL}.stack_{top}.gregs[j] = c^{s(i+1)}_{ASM}.gpr[j])
\end{array}$$

We did not prove this theorem as without knowing the implementation of an optimizing C compiler being used we are not able to do that. So we assume that this theorem holds for an optimizing C compiler in question. But one important argument to prove this theorem is that the callee-save registers must not be changed by code generated for a call and the callee-save registers must be restored before a return statement if changed.

◀ THEOREM 4.28
Compiler Correctness

Let π be a *C-IL* program, c^0_{ASM} be the initial state of a target machine. Then for any sequence of a target machine: c^0_{ASM}, c^1_{ASM}, ..., there exists a *C-IL* machine sequence c^0_{IL}, c^1_{IL}, ... that does not get stuck and does not produce a stack overflow. Moreover, for any number steps i of a *C-IL* machine there exists the corresponding target machine state $c^{s(i)}_{ASM}$, such that the compiler consistency $consis_{IL}(c^i_{IL}, info_{IL}, c^{s(i)}_{ASM})$ holds if the statement to be executed at step i by the *C-IL* machine is an IO point. However, these states are consistent if the following requirements about the *C-IL* and VAMP assembly machines are fulfilled:

- The *C-IL* program π is translatable,
- The *C-IL* configuration c_{IL} and VAMP assembly configuration c_{ASM} are valid and they are consistent,
- After performing i steps the machine c_{IL} did not get stuck,

Formally, this theorem is stated as follows:

$$
\begin{aligned}
&\exists\, info_{IL}.\ \forall\, (c^i_{ASM})_i, \exists (c^i_{IL})_i.\\
&\quad (\forall\, i.\quad c^{i+1}_{ASM} = \delta_{\text{ASM}}(c^i_{ASM})\\
&\qquad \wedge\quad \pi, \theta \vdash c^i_{IL} \rightarrow_{IL} c^{i+1}_{IL}\\
&\qquad \wedge\quad \text{the execution of } c_{IL} \text{ does not get stuck}\\
&\qquad \wedge\quad \text{there is no stack overflow})\\
&\wedge\quad \exists\, s :: \mathbb{N} \mapsto \mathbb{N}.\\
&\qquad (\forall\, i.\, info_{IL}.iop(c^i_{IL}.f_{top}, c^i_{IL}.loc_{top})\\
&\qquad\qquad \longrightarrow consis_{IL}(c^i_{IL}, info_{IL}, c^{s(i)}_{ASM}))
\end{aligned}
$$

Proof: We sketch a proof for this theorem. So the proof is done by induction on the step number i of the *C-IL* machine.

We start the induction with i equals zero. In this case we need to show that the target machine $c^{s(0)}_{ASM}$ after performing $s(0)$ steps will be valid and consistent with the *C-IL* machine c^0_{IL}. The proof for similar case is shown in Leinenbach's dissertion [Lei07] for non-optimizing C compiler.

For the induction step we have to conclude from step i to $i + 1$. We prove this case by a case distinction on the *C-IL* statement to be executed at step $i + 1$ by the *C-IL* machine. For all *C-IL* statements the proof must follow from the implementation of an optimizing C compiler in question and the theorem talking about the content preservation of callee-save registers across calls.

■

CHAPTER 5

Integrated Semantics μ_{ASM} and *C-IL*

In this section we present a model, called *MX* (from mixed), obtained by integrating ("mixing") the μ_{ASM} and *C-IL* semantics previously examined in this thesis. This model was invented in collaboration with S.Schmaltz which resulted in our joint publication [SS12] presenting this model. To achieve such an integration, we use the *compiler calling convention* introduced in Section 3.1.4 and apply it to interface the two languages. The goal of this integration is to 'slice' the model stack horizontally, providing a self-contained model to argue about a system layer that involves both *C-IL* and μ_{ASM} code executions. As a result, in the mixed semantics we can implement a non-concurrent software written in two different programming languages: *C-IL* and μ_{ASM}. Considering verification engineering as a social process, providing integrated semantics offers certain benefits: The person who does soundness proofs for code-verification tools no longer needs to consider the inner workings of compilers - instead, they can use the integrated semantics as foundation of their proofs.

We do not introduce a new set of statements in *MX* as it is done in the μ_{ASM} and *C-IL* models as we do not consider a so-called inter-language statement execution in the frame of this thesis. That is, programs we want to describe by the *MX* semantics do not contain *inline assembly* portions. However, we model so-called inter-language-call and -return statements. In the combined semantics *MX* μ_{ASM} and *C-IL* execution sequences are interleaved at external calls and -returns. That is why the two languages must follow the same compiler calling convention in the mixed semantics.

5.1 Semantics

We start the description of the *MX* semantics with the definition of an *MX* program.

5.1.1 Program

An *MX*-program π is simply a record consisting of a *C-IL*-program $\pi_{C\text{-}IL}$ and μ_{ASM}-program $\pi_{\mu_{\text{ASM}}}$.

DEFINITION 5.1 ► *MX*-program

$$Prog_{MX} \stackrel{\text{def}}{=} \{\pi_{\text{C-IL}} :: Prog_{IL}, \pi_{\mu_{\text{ASM}}} :: Prog_{\mu_{\text{ASM}}}\}$$

5.1.2 System Parameters

Since we need the information about the compiler to execute *C-IL* we keep the system parameters θ from *C-IL*.

5.1.3 Configurations

The most basic way to define an *MX* configuration is to consider a list of alternating *C-IL*- and μ_{ASM}-configurations that represents the stack between two languages. The top-most stack configuration we consider *active* while the rest of them *inactive*. Observations that can be made, however, is that (i) in both semantics we use the same byte-addressable memory, which can be shared, and (ii) we know that SPRs are modeled as a part of μ_{ASM} configuration and they might also be relevant for *C-IL* executions[1], which can be shared as well.

In order to eliminate redundancy, we introduce the notion of *execution context* for *C-IL* and μ_{ASM} in the *MX* semantics. We distinguish between active and inactive execution contexts for each language in the *MX* semantics. After defining these contexts we will present how *MX* configurations are modeled.

Active Execution Context An active execution context is a configuration of the corresponding language without memory. So the *C-IL* active execution context is represented by a list of abstract *C-IL* frames.

DEFINITION 5.2 ► Active *C-IL* Execution Context

$$context_{\text{C-IL}} \stackrel{\text{def}}{=} frame^{*}_{\text{C-IL}}$$

The μ_{ASM} active execution context has the same components as the μ_{ASM} configuration besides the memory and SPRs. As we have already told that SPRs are shared between two languages in the *MX* semantics. Hence, there is no need to model them twice. So the μ_{ASM} active execution context is defined as

DEFINITION 5.3 ► Active μ_{ASM} Execution Context

$$context_{\mu_{\text{ASM}}} \stackrel{\text{def}}{=} \{gpr :: \mathbb{Z}^{*}, s :: frame^{*}_{\mu} \cup frame^{no}_{\mu}\}$$

Inactive Execution Context An inactive execution context must contain information needed to continue the execution as expected after becoming active again.

We first define the inactive μ_{ASM} execution context. Here we make an observation that it is not meaningful to store values of all GPRs in the inactive μ_{ASM} execution context, with one exception: we can keep the values of callee-save registers, since, assuming *C-IL* compiler respects the calling conventions, they will be restored when control is returned back to this inactive context. So the inactive μ_{ASM} execution context is defined as

DEFINITION 5.4 ► Inactive μ_{ASM} Execution Context

[1] The reader might wonder how it can be relevant for *C-IL* executions: In case interrupts are modeled at the *C-IL* level, i.e interrupts are visible in the *C-IL* semantics. From the VAMP assembly semantics presented in Chapter 2 we know that some SPRs gets updated when an interrupt happens. Right now, interrupts are invisible in the *C-IL* semantics.

$$context^{inactive}_{\mu_{\text{ASM}}} \stackrel{\text{def}}{=} \{gpr_{\textbf{callee}} :: \mathbb{Z}^*,\ s :: frame^*_{\mu}\}$$

consisting of a callee-save register file $gpr_{\textbf{callee}}$ that holds values of callee-save registers of the execution context, and a μ_{ASM} stack s.

The inactive *C-IL* execution context definitely must contain a *C-IL* stack. Another observation we made is that in order to integrate the two semantics we need to have the following component to be a part of inactive *C-IL* execution contexts:

- a callee-save register file that contains values of callee-save registers. The *C-IL* compiler relies on the callee-save convention being respected by the programmer: in case callee-save registers have modified values when the *C-IL* execution context becomes active again, there is no guarantee whatsoever that *C-IL* executions will continue as expected.

So we define the inactive *C-IL*-execution context as record

◀ DEFINITION 5.5
Inactive *C-IL* Execution Context

$$context^{inactive}_{C\text{-}IL} \stackrel{\text{def}}{=} \{gpr_{\textbf{callee}} :: \mathbb{Z}^*,\ s :: frame^*_{C\text{-}IL}\}$$

which consists of a callee-save general purpose register file $gpr_{\textbf{callee}}$ that contains values of those registers expected when the control is returned to the execution context, and a *C-IL* stack s.

Configuration *MX* configurations are represented by record type C_{MX} defined as

◀ DEFINITION 5.6
MX-Configuration

$$C_{MX} \stackrel{\text{def}}{=} \begin{array}{rcl} \{\ \mathcal{M} & :: & \mathbb{N} \mapsto \mathbb{B}^8, \\ spr & :: & \mathbb{Z}^*, \\ ac & :: & context_{C\text{-}IL} \cup context_{\mu_{\text{ASM}}}, \\ sc & :: & (context^{inactive}_{C\text{-}IL} \cup context^{inactive}_{\mu_{\text{ASM}}})^*\} \end{array}$$

that consists of the same byte-addressable memory $\mathcal{M}$ we have seen in the μ_{ASM} and *C-IL* semantics, shared special purpose register file spr within two languages (in this thesis it is used only for μ_{ASM} language), active execution context ac, and an abstract stack of inactive(suspended) execution contexts sc. Note that the i-th and $(i+1)$-th elements of an abstract stack of inactive execution contexts must be of different types. This and all other properties about *MX* configurations are captured in the predicate $valid_{MX}$ stating the well-formedness of a given *MX* configuration c_{MX}. This predicate is defined with $sc = c_{MX}.sc$ and $l = |sc|$ as

$$\frac{\begin{array}{c} c_{MX}.ac \in context_{C\text{-}IL} \longrightarrow l = 0 \vee sc[l-1] \in context^{inactive}_{\mu_{\text{ASM}}} \\ c_{MX}.ac \in context_{\mu_{\text{ASM}}} \longrightarrow l = 0 \vee sc[l-1] \in context^{inactive}_{C\text{-}IL} \\ \forall\ i+1 < l.\ sc[i] \in context^{inactive}_{C\text{-}IL} \longleftrightarrow sc[i+1] \in context^{inactive}_{\mu_{\text{ASM}}} \\ \forall\ i+1 < l.\ sc[i] \in context^{inactive}_{\mu_{\text{ASM}}} \longleftrightarrow sc[i+1] \in context^{inactive}_{C\text{-}IL} \\ |c_{MX}.spr| = 32 \\ c_{MX}.ac \in context_{\mu_{\text{ASM}}} \longrightarrow |c_{MX}.ac.gpr| = 32 \\ \forall i < l.sc[i] \in context^{inactive}_{C\text{-}IL} \longrightarrow |sc[i].gpr_{callee}| = |callee| \\ \forall i < l.sc[i] \in context^{inactive}_{\mu_{\text{ASM}}} \longrightarrow |sc[i].gpr_{callee}| \leq |callee| \\ \neg is\_stack(c_{MX}) \longrightarrow sc = [] \end{array}}{valid_{MX}(c_{MX})}$$

where the predicate $is\_stack :: C_{MX} \mapsto bool$ indicates whether the stack is created or not in a given *MX* configuration c_{MX} and it is defined as

$$\frac{c_{MX}.ac \in context_{\mu_{\text{ASM}}} \longrightarrow c_{MX}.ac.s \notin frame_{\mu}^{no}}{is\_stack(c_{MX})}$$

5.1.4 Auxiliary Functions and Shorthands

Here we introduce the definition of functions which help to describe the *MX* semantics and compiler consistency in a brief way.

We define the function

$$stack^{flat} :: C_{MX} \mapsto (frame_{\mu} \cup frame_{C\text{-}IL})^{*}$$

flattening stack frames of the active and inactive contexts for a given *MX* configuration. Note that stack frames of the active execution context must be on top of the list of flattened stack frames reconstructed from inactive contexts by means of the function *flat-inact-ctx*. In case the stack is not created in a given *MX* configuration c_{MX} this function returns an empty list.

$$stack^{flat}(c_{MX}) \stackrel{\text{def}}{=} \begin{cases} c_{MX}.ac.s \circ \textit{flat-inact-ctx}(c_{MX}.sc) & \text{if } is\_stack(c_{MX}) \\ [] & \text{otherwise} \end{cases}$$

where the function *flat-inact-ctx* is defined as

$$\textit{flat-inact-ctx}(sc) \stackrel{\text{def}}{=} \begin{cases} \mathbf{hd}(sc).s \circ \textit{flat-inact-ctx}(\mathbf{tl}(sc)) & \text{if } \mathbf{tl}(sc) \neq [] \\ [] & \text{otherwise} \end{cases}$$

The function $top :: C_{MX} \mapsto \mathbb{N}$ returns the index of the top stack frame in the flattened list of stack frames of a given *MX* configuration.

$$top(c_{MX}) \stackrel{\text{def}}{=} |c_{MX}.stack^{flat}| - 1$$

We denote by $c_{MX}.stack_{top}$ the top stack frame or no-stack of an *MX* configuration c_{MX} depending on whether the stack is created in this *MX* configuration:

$$c_{MX}.stack_{top} \stackrel{\text{def}}{=} \begin{cases} c_{MX}.ac.s & \text{if } \neg is\_stack(c_{MX}) \\ \mathbf{hd}(c_{MX}.ac.s) & \text{otherwise} \end{cases}$$

Based on the above introduced shorthand we denote by $c_{MX}.f_{top}$ and $c_{MX}.loc_{top}$ the function name and location counter components of the top stack frame or no-stack of a *MX* configuration, respectively.

We introduce another two shorthands $c_{MX}.f_i$ and $c_{MX}.loc_i$ to denote the function (procedure) name and the location counter, respectively, of i-th stack frame in the flattened list of stack frames of a given configuration c_{MX}.

$$c_{MX}.f_i \stackrel{\text{def}}{=} \begin{cases} c_{MX}.stack^{flat}[i].f & \text{if } c_{MX}.stack^{flat}[i] \in frame_{C\text{-}IL} \\ c_{MX}.stack^{flat}[i].p & \text{otherwise} \end{cases}$$

$$c_{MX}.loc_i \stackrel{\text{def}}{=} c_{MX}.stack^{flat}[i].loc$$

The function

$$inc_{loc} :: frame^{no}_{\mu} \cup frame^{*}_{\mu} \cup frame^{*}_{C\text{-}IL} \mapsto frame^{no}_{\mu} \cup frame^{*}_{\mu} \cup frame^{*}_{C\text{-}IL}$$

increments the location counter of either the top stack frame of a given list of μ_{ASM} or *C-IL* stack frames or no-stack.

$$inc_{loc}(s) \stackrel{\text{def}}{=}$$

$$\begin{cases} s[loc := s.loc + 1]] & \text{if } s \in frame^{no}_{\mu} \\ s[\mathbf{top}(s) := \mathbf{hd}(s)[loc := \mathbf{hd}(s).loc + 1]] & \text{if } s \in frame^{*}_{\mu} \cup frame^{*}_{C\text{-}IL} \end{cases}$$

The function $drop_{lifo} :: frame^{*}_{\mu} \mapsto frame^{*}_{\mu}$ removes the n top elements from the top stack frame of a given list of μ_{ASM} stack frames.

$$drop_{lifo}(s, n) \stackrel{\text{def}}{=}$$

$$s[\mathbf{top}(s) := \mathbf{hd}(s)[lifo := \mathbf{drop}(\mathbf{hd}(s).lifo, n)]]$$

The function $set_{rds} :: frame^{*}_{C\text{-}IL} \times val \mapsto frame^{*}_{C\text{-}IL}$ sets the return destination of the top stack frame of a given list of *C-IL* stack frames to a given value v.

$$set_{rds}(s, v) \stackrel{\text{def}}{=} s[\mathbf{top}(s) := \mathbf{hd}(s)[rds := v]]$$

5.1.5 Expression Evaluation

In *MX* the expression evaluation is performed if and only if the active context of a configuration c_{MX} is of *C-IL* type and its definition is based on the *C-IL* expression evaluation function:

$$[e]^{\theta,\pi}_{c_{MX}} \stackrel{\text{def}}{=} [e]^{\theta,\pi,\pi_{C\text{-}IL}}_{(c_{MX}.\mathcal{M}, c_{MX}.ac)}$$

◀ DEFINITION 5.7
C-IL Expression Evaluation in *MX*

5.1.6 Transition Function

We denote an *MX* configuration c_{MX} making a step to a configuration c'_{MX} under the program π and system parameters θ by:

$$\pi, \theta \vdash c_{MX} \rightarrow_{MX} c'_{MX}.$$

In the transition function there are six cases we need to consider: two cases for pure language steps, two cases for inter-language call, and another two cases for inter-language return.

Considering a given *MX* configuration, it is easy to decide which of these has to happen next. Calling a function which is declared as external in the function table of one language and as defined(non-external) in the function of another language is considered as an inter-language call causing the exchange of the execution context. Executing a return statement when the stack of the active stack frame is empty should activate the last context which was put on a stack of inactive contexts. Everything else is considered as a pure language step for which we have already defined the semantics. Let $ext(c_{MX}, \pi)$ denotes a predicate that checks in the described way whether the next step is an inter-language step. There are four cases that make this predicate true. We do not define this predicate, one can look it up these conditions in cases 3-6. First, we treat non-external cases (1-2), called *pure steps*, in the transition function, and then we treat external ones, called *inter-language steps*.

1. Pure *C-IL* Step It is the case when the active context is of *C-IL* type and the current statement of the *MX* configuration to be executed does not cause the exchange of the execution context. So that, we consider only steps which are handled by the *C-IL* semantics. For this case the *MX* semantics is defined as follows:

$$\frac{c_{MX}.ac \in context_{C\text{-}IL} \qquad c_{MX}.ac = s \qquad \neg ext(s,\pi) \qquad \pi.\pi_{C\text{-}IL}, \theta \vdash (c_{MX}.\mathcal{M}, s) \rightarrow_{IL} c'_{IL}}{\pi, \theta \vdash c_{MX} \rightarrow_{MX} c_{MX}\left[\begin{array}{lcl} \mathcal{M} & := & c'_{IL}.\mathcal{M}, \\ ac & := & c'_{IL}.s \end{array}\right]}$$

2. Pure μ_{ASM} Step This case is similar to the previous one in a sense that a pure step is performed here, too. The case presented before covers a *pure C-IL* step and here a *pure* μ_{ASM} step is considered.

$$\frac{c_{MX}.ac \in context_{\mu_{\text{ASM}}} \qquad c_{MX}.ac = (gpr, s) \qquad \neg ext(s,\pi) \qquad \pi.\pi_{\mu_{\text{ASM}}} \vdash (c_{MX}.\mathcal{M}, gpr, c_{MX}.spr, s) \rightarrow_{\mu_{\text{ASM}}} c'_{\mu}}{\pi, \theta \vdash c_{MX} \rightarrow_{MX} c_{MX}\left[\begin{array}{lcl} \mathcal{M} & := & c'_{\mu}.\mathcal{M}, \\ ac & := & (c'_{\mu}.gpr, c'_{\mu}.spr, c'_{\mu}.s) \end{array}\right]}$$

3. Call from *C-IL* To μ_{ASM} This case is identified by the following preconditions: (i) the active context is of *C-IL* type, (ii) the current statement of the c_{MX} machine is either a function or procedure call, (iii) the function (or procedure) to be called is represented either by a function pointer or a symbolic name, and (iv) the function to be called is declared as external in the *C-IL* function table and as non-external in the μ_{ASM} procedure table.

If all aforementioned preconditions are fulfilled, then a new μ_{ASM} context is created, initialized according to the calling convention. The currently active *C-IL* context is retired to stack of inactive contexts of the *MX* configuration. Constraints on the nondeterministically chosen new context and callee-save registers to be remembered by the active *C-IL* context to be retired are captured in the predicate $CIL2MASM$.

$$\frac{\begin{array}{c} c_{MX}.ac \in context_{C\text{-}IL} \qquad c_{MX}.ac = s \\ stmt_{curr}(s, \pi.\pi_{C\text{-}IL}) \in \{\textbf{call } e(E), e_0 = \textbf{call } e(E)\} \\ is\text{-}function([e]^{\theta,\pi}_{c_{MX}}, f, \theta) \\ \pi.\pi_{C\text{-}IL}.\mathcal{F}(f).\mathcal{P} = \textbf{extern} \qquad \pi.\pi_{\mu_{\text{ASM}}}(f).body \neq \textbf{extern} \\ CIL2MASM(c_{MX}, f, E, gpr_{\textbf{callee}}, rds, context_{new}) \end{array}}{\pi, \theta \vdash c_{MX} \rightarrow_{MX} c_{MX}\left[\begin{array}{lcl} ac & := & context_{new}, \\ sc & := & (gpr_{\textbf{callee}}, set_{rds}(inc_{loc}(s), rds)) \circ c_{MX}.sc \end{array}\right]}$$

The predicate $CIL2MASM(c_{MX}, f, E, gpr_{\textbf{callee}}, rds, context_{new})$ enforces the following constraints with $n = \pi.\pi_{\mu_{\text{ASM}}}(f).npar$:

- The content of GPRs of the new μ_{ASM} active context is chosen non-deterministically except for input registers. According to the calling convention $CCL.1$ the first four parameters are passed in registers, the remaining parameters are passed on the stack.

$$context_{new}.gpr[i_{j+1}] = [E[j]]^{\theta,\pi,\pi_{C\text{-}IL}}_{c_{MX}} \quad \text{if } 0 \leq j < 4 \ \wedge \ j < n$$

- The stack of the new μ_{ASM} active context has a single stack frame

$$context_{new}.stack = [frame_{new}]$$

where the content of this stack frame is defined as follows

$$\begin{array}{lcl} frame_{new}.p & = & f \\ frame_{new}.loc & = & 0 \\ \forall\, i < n.\, frame_{new}.pars[i] & = & \begin{cases} [E[i]]^{\theta,\pi}_{c_{MX}} & \text{if } 4 \le i \wedge i < n \\ A & \text{otherwise} \end{cases} \\ frame_{new}.saved & = & \textbf{sublist}(gpr, \pi.\pi_{\mu_{\text{ASM}}}(f).uses) \\ frame_{new}.lifo & = & [] \end{array}$$

- The callee-save registers expected by the retired *C-IL*-execution context are the same that reside in the new μ_{ASM}-execution-context:

$$gpr_{\textbf{callee}} = \textbf{sublist}(context_{new}.gpr, callee \setminus \pi.\pi_{C\text{-}IL}(f).uses)$$

Note that those GPRs whose indices are declared in $\pi.\pi_{\mu_{\text{ASM}}}(f).uses$ will be restored at return. It is guaranteed by the μ_{ASM} semantics. Thus, there is no need to remember them in $gpr_{\textbf{callee}}$ of the *C-IL* execution context to be retired.

- The return destination component of the retired *C-IL*-execution-context is:

$$rds = \begin{cases} [\&e_0]^{\theta,\pi,\pi_{C\text{-}IL}}_{c_{MX}} & \text{function call} \\ \bot & \text{procedure call} \end{cases}$$

4. Return From μ_{ASM} To *C-IL* This case is complementary to the previous one. The preconditions for this case are: (i) the active context is of μ_{ASM} type, (ii) the statement to be executed is a return instruction, (iii) there is only one stack frame in the stack of the active context, (iv) in order to return to the previously active context it must exists in the stack of inactive contexts, and (v) callee-save registers must have the values expected by the *C-IL* context we return to.

If the preconditions are fulfilled, then the return statement performs an inter-language return such that the *C-IL* execution context that has created the current active μ_{ASM} context becomes active again. This is done by substituting the current active μ_{ASM} context with the *C-IL* execution context found on the top of inactive execution contexts. Also, the content of the return value register rv[2] of the current active μ_{ASM} context is written to the return destination rds given in the *C-IL* context we return to.

$$\frac{\begin{array}{c} c_{MX}.ac \in context_{\mu_{\text{ASM}}} \qquad c_{MX}.ac = (gpr, s) \qquad s \notin frame^{no}_{\mu} \\ instr_{curr}(s, \pi.\pi_{\mu_{\text{ASM}}}) = \textbf{ret} \\ 0 < |c_{MX}.sc| \qquad \textbf{hd}(c.sc) = (gpr_{\textbf{callee}}, s') \\ \textbf{sublist}(gpr, callee \setminus \pi.\pi_{\mu_{\text{ASM}}}(c_{MX}.f_{top}).uses) = \\ \qquad \textbf{sublist}(gpr_{\textbf{callee}}, callee \setminus \pi.\pi_{\mu_{\text{ASM}}}(c_{MX}.f_{top}).uses) \\ c'_{MX} = write^{\theta}((c_{MX}.\mathcal{M}, s'), \textbf{hd}(s').rds, gpr[rv]) \end{array}}{\pi, \theta \vdash c_{MX} \rightarrow_{MX} c_{MX} \left[\begin{array}{lcl} \mathcal{M} & := & c'_{MX}.\mathcal{M}, \\ ac & := & c'_{MX}.s, \\ sc & := & \textbf{tl}(c_{MX}.sc) \end{array} \right]}$$

[2] rv is the index of the return value register (cf. Table 3.1). Note that in the μ_{ASM} semantics the return value register contains the result value (cf. compiler calling convention in Section 3.1.4).

5. Call From μ_{ASM} To *C-IL* This case is indicated by the following preconditions: (i) the active context is of μ_{ASM} type, (ii) the next instruction to be executed is a procedure call, (iii) the procedure to be called is declared as external in the μ_{ASM} procedure table and as non-external in the *C-IL* procedure table, and (iv) the lifo of the top stack frame of the active μ_{ASM} context is big enough to contain parameters passed on stack.

If all of the aforementioned preconditions are fulfilled, then a new *C-IL* context is created and it becomes active in the *MX* machine. The current active μ_{ASM} context is put on the stack of inactive contexts. Also, the location counter of the top stack of the current active μ_{ASM} context is incremented. Constraints on the non-deterministically chosen new *C-IL* context are captured in the predicate $MAS\,M2CIL$. Here it is assumed that the indices listed in the uses component of the procedure being called must be callee-save registers, i.e.

$$\forall\ r \in \pi.\pi_{\mu_{\text{ASM}}}(P).uses.\ r \in callee$$

After all this case is treated with $npar = \pi.\pi_{\mu_{\text{ASM}}}(P).npar$, $\mathcal{V} = \pi.\pi_{\mu_{\text{ASM}}}(P).\mathcal{V}$ and $lifo = \mathbf{hd}(s.stack).lifo$ as

$$\frac{\begin{array}{c} c_{MX}.ac \in context_{\mu_{\text{ASM}}} \qquad c_{MX}.ac = (gpr, s) \qquad s \notin frame_{\mu}^{no} \\ instr_{curr}(s, \pi.\pi_{\mu_{\text{ASM}}}) = \mathbf{call}\ P \qquad k = \mathbf{max}\{npar - 4, 0\} \qquad k \leq |lifo| \\ \pi.\pi_{\mu_{\text{ASM}}}(P).body = \mathbf{extern} \qquad \pi.\pi_{C\text{-}IL}.\mathcal{F}(P).\mathcal{P} \neq \mathbf{extern} \\ gpr_{\mathbf{callee}} = \mathbf{sublist}(gpr, callee) \\ MAS\,M2CIL(P, npar, gpr, lifo, context_{new}) \end{array}}{\pi, \theta \vdash c_{MX} \rightarrow_{MX} c_{MX}\left[\begin{array}{lcl} ac & := & context_{new}, \\ sc & := & (gpr_{\mathbf{callee}}, drop_{lifo}(inc_{loc}(s), k)) \circ c_{MX}.sc \end{array}\right]}$$

The predicate $MAS\,M2CIL(P, npar, \mathcal{V}, gpr, lifo, context_{new})$ enforces the following constraints:

$$context_{new} = [frame_{new}]$$

where a single stack frame $frame_{new}$ of the new context is constrained with $(v_j, t_j) = \mathcal{V}[j]$ by the following:

- The first four parameters are taken from the corresponding GPRs of the μ_{ASM} context to be retired. The values stored in those GPRs are converted to *C-IL*-values of the type expected by the function and store them at the corresponding places in the local memory environment of the stack frame $frame_{new}$.

$$\forall 0 \leq j < 4 : j < npar \longrightarrow read_{\mathcal{E}}^{\theta}(frame_{new}, v_j) = Int2val(gpr[i_{j+1}], t_j)$$

- The remaining input parameters (if existent) are taken from the lifo of the top stack frame of the current active μ_{ASM} context to be retired. Recall the rule *CCL* . 2 introduced in Section 3.1.4 that first parameters are passed on stack ($lifo$) in right-to-left order.

$$\forall 4 \leq j < npar : read_{\mathcal{E}}^{\theta}(frame_{new}, v_j) = Int2val(lifo[|lifo| - 1 - j], t_j)$$

- The space is reserved for local variables in the newly created frame:

$$\forall npar \leq j < |\mathcal{V}| : \quad |frame_{new}.\mathcal{M}_{\mathcal{E}}(v_j)| = size(t_j)$$

- The components modeling the execution flow are constrained by the following:

$$\begin{array}{lcl} frame_{new}.f & = & P \\ frame_{new}.loc & = & 0 \end{array}$$

- The return destination component is always left undefined as the active μ_{ASM} context, namely the μ_{ASM} semantics, does not have any notion of such a component.

$$frame_{new}.rds = \bot$$

6. Return From *C-IL* To μ_{ASM} This case is complementary to the previous one. The preconditions indicating this case are: (i) the active context is of *C-IL* type, (ii) the statement to be executed is return, (iii) there is only one stack frame in the active context, (iv) there exists at least one context in the stack of inactive contexts, and (v) callee-save registers must have the values expected by the μ_{ASM} context we return to.

If the preconditions are fulfilled, then the previously active context becomes active again. In the new context the caller-save GPRs are assigned non-deterministically except for the return value register *rv*. The result value is returned from the current active *C-IL* context to the μ_{ASM} context to be activated and is stored in the return value register *rv* of this μ_{ASM} context.

$$\frac{\begin{array}{c} c_{MX}.ac \in context_{C\text{-}IL} \qquad c_{MX}.ac = s \\ stmt_{curr}(s, \pi.\pi_{C\text{-}IL}) = \textbf{return}\ e \\ 0 < |c_{MX}.sc| \qquad \textbf{hd}(c_{MX}.sc) = (gpr_{\textbf{callee}}, s') \\ \textbf{sublist}(gpr', callee) = gpr_{\textbf{callee}} \\ gpr'[rv] = val2Int([e]^{\pi,\theta}_{c_{MX}}) \end{array}}{\pi, \theta \vdash c_{MX} \rightarrow_{MX} c_{MX}\left[\begin{array}{lcl} ac & := & (gpr', s'), \\ sc & := & \textbf{tl}(c.sc) \end{array}\right]}$$

5.2 Simulation Theorem

Here we present the simulation relation between *MX* and VAMP assembly configurations. We defined similar relations for μ_{ASM} and *C-IL* in Chapters 3 and 4, respectively. In order to be able to state the simulation relation between the *MX* and VAMP assembly execution sequences, where the latter runs the compiled *MX* execution sequence. We need to have an additional information about the compiler code, like the *C-IL* static information introduced in Section 4.2. Since we have *C-IL* executions sequences as well as μ_{ASM} together in the *MX* semantics and the *C-IL* static information can describe the μ_{ASM} ones[3], we keep the static information from *C-IL* in *MX*.

The *MX* compiler meta information $infoT_{MX}$ is described by record type

[3] it is assumed that $P_{name} \subset \mathbb{F}_{name}$

DEFINITION 5.8 ►
MX Compiler Meta Info

$$infoT_{MX} \stackrel{\text{def}}{=} infoT_{IL}$$

We define the function $dist^{bp}_{MX}(i, s, info_{MX})$ returning the distance between frame base addresses of i-th and $(i+1)$-th stack frames from a list s of *C-IL*- and μ_{ASM} stack frames. To define it we use the functions $dist^{bp}_{\mu_{\text{ASM}}}$ and $dist^{bp}_{IL}$ defined in Sections 3.3 and 4.2, respectively. Here we need to consider six cases: if the i-th stack frame is

- the top stack frame and it is of *C-IL* type, then we apply the function $dist^{bp}_{IL}$ to a list containing a single element which is the top stack frame from the list s (cf. the top stack frame in Figure 3.3).
- the top stack frame of μ_{ASM} type, then we apply the function $dist^{bp}_{\mu_{\text{ASM}}}$ to a list containing a single element which is the top stack frame from the list s (cf. the top stack frame in Figure 4.4).
- a non-top stack frame and it is of *C-IL* type and the $(i+1)$-th stack frame is of *C-IL* type, then we apply the function $dist^{bp}_{IL}$ (cf. the non-top stack frame in Figure 4.4), then we apply the function $dist^{bp}_{IL}$ to a list containing two elements $([s[i+1]] \circ s[i]])$.
- a non-top stack frame and it is of *C-IL* type and the $(i+1)$-th stack frame is of μ_{ASM} type, then we compute the distance by summing up the following (cf. the non-top *C-IL* frame in Figure 5.1(a)):
  - the size of the local variables block, callee-save registers block and temporaries of the i-th stack frame. This size is computed by the function $dist^{bp}_{IL}$ on a list containing a single element which is the i-th stack frame from the list s.
  - 4 bytes representing a return destination (in case of a function call),
  - caller-save registers block,
  - 8 bytes representing a stack frame header, and
  - the size of all parameters passed to the $(i+1)$-th μ_{ASM} stack frame.
- a non-top stack frame and it is of μ_{ASM} type and the $(i+1)$-th stack frame is of μ_{ASM} type, then we apply the function $dist^{bp}_{\mu_{\text{ASM}}}$ (cf. the non-top stack frame in Figure 3.3).
- a non-top stack frame and it is of μ_{ASM} type and the $(i+1)$-th stack frame is of *C-IL* type, then we compute the distance by summing up the following (cf. the non-top μ_{ASM} frame in Figure 5.1(b)):
  - the size of the save register block and lifo of the i-th stack frame. This size is computed by the function $dist^{bp}_{\mu_{\text{ASM}}}$ on a list containing a single element which is the i-th stack frame from the list s.
  - 8 bytes representing a stack frame header, and
  - the size of all parameters passed to the $(i+1)$-th *C-IL* stack frame.

◀ DEFINITION 5.9
Distance Between Frame Base Addresses

$$dist^{bp}_{MX}(i, s, info_{MX}) \stackrel{\text{def}}{=}$$

$$\begin{cases} dist^{bp}_{IL}(0, [s[i]], info_{MX}) & \text{if } top(s) = i \wedge s[i] \in frame_{C\text{-}IL} \\ dist^{bp}_{\mu_{\text{ASM}}}(0, [s[i]]) & \text{if } top(s) = i \wedge s[i] \in frame_{\mu} \\ dist^{bp}_{IL}(0, [s[i+1]] \circ [s[i]], info_{MX}) & \text{if } s[i] \in frame_{C\text{-}IL} \wedge s[i+1] \in frame_{C\text{-}IL} \\ \begin{array}{l} dist^{bp}_{IL}(0, [s[i]], info_{MX}) \\ \quad + \begin{cases} 4 & \text{if } s[i].rds \neq \perp \\ 0 & \text{otherwise} \end{cases} \\ \quad + info_{IL}.size_{caller-save}(s[i].f, s[i].loc - 1) \\ \quad + 8 + 4 \cdot n \end{array} & \text{if } s[i] \in frame_{C\text{-}IL} \wedge s[i+1] \in frame_{\mu} \\ dist^{bp}_{\mu_{\text{ASM}}}(0, s[i+1] \circ s[i]) & \text{if } s[i] \in frame_{\mu} \wedge s[i+1] \in frame_{\mu} \\ dist^{bp}_{\mu_{\text{ASM}}}(0, [s[i]]) + 8 + 4 \cdot n & \text{otherwise} \end{cases}$$

where n is the number of input parameters passed on stack to a call to the function $s[i+1].f$:

$$n = \mathbf{max}\{\pi.\pi_{C\text{-}IL}.\mathcal{F}(s[i+1].f).npar - 4, 0\}.$$

Next we define a function

$$frame^{ba}_{MX} :: frame^{*}_{\mu} \times \mathbb{N} \mapsto \mathbb{N}$$

computing an absolute frame base address for the i-th stack frame from the stack frame list s, where the base address of the first stack frame is identified by the component $info_{MX}.stack_{ba}$. This function is based on the function $dist^{bp}_{MX}$ defined above which computes relative offsets between frame base addresses of adjacent stack frames.

◀ DEFINITION 5.10
Frame Base Address

$$frame^{ba}_{MX}(i, s, info_{MX}) \stackrel{\text{def}}{=} info_{MX}.stack_{ba} - \sum_{j=0}^{j<i} (dist^{bp}_{MX}(i, s, info_{MX}))$$

We define functions reading frame headers with the same signature as we did for μ_{ASM} and *C-IL* in previous chapters. Note that the previous stack pointer and return address components of the frame header of μ_{ASM} and *C-IL* stack frames are located in memory at the same relative offsets to the base address of a stack frame (cf. Table 3.2 and Figure 5.1).

◀ DEFINITION 5.11
Reading frame headers

$$stack^{pbp}_{MX}(i, s, m, info_{MX}) \stackrel{\text{def}}{=} m_{word}(frame^{ba}_{MX}(i, s, info_{MX}) + 0)$$

$$stack^{ra}_{MX}(i, s, m, info_{MX}) \stackrel{\text{def}}{=} m_{word}(frame^{ba}_{MX}(i, s, info_{MX}) + 4)$$

Next we define the function $stack^{rds}_{MX}$ reading the return destination put on the stack at a function(procedure) call that created the $(i+1)$-th stack frame.

◀ DEFINITION 5.12
Reading return destination

$$\begin{array}{rcl} stack^{rds}_{MX}(i, s, m, info_{MX}) & \stackrel{\text{def}}{=} & m_{word}(frame^{ba}_{IL}(i+1, s, info_{MX}) + 8 \\ & + & info_{MX}.size_{params}(s[i+1].f)) \end{array}$$

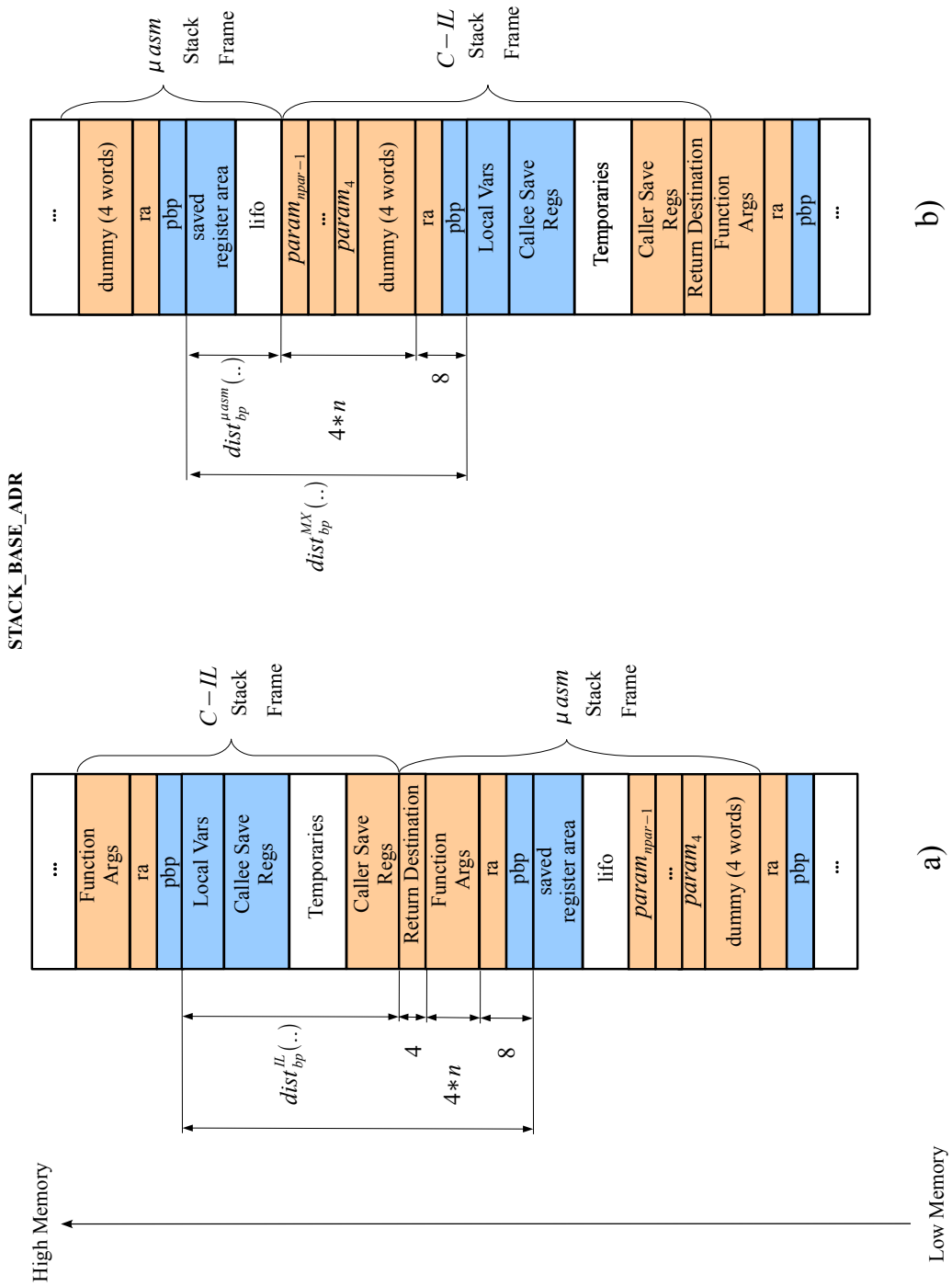


Figure 5.1: *MX* Stack Layout

The function

$$stack^{item}_{MX} :: frame^{*}_{\mu} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times (\mathbb{N} \mapsto \mathbb{Z}) \mapsto \mathbb{Z}$$

returns the value of j-th word stored on the i-th stack frame (cf. Figure 5.1).

◀ DEFINITION 5.13
Reading Items From Stack Frames

$$stack^{item}_{MX}(s, i, j, m, info_{MX}) \stackrel{\text{def}}{=} m_{word}(frame^{ba}_{MX}(s, i, info_{MX}) - 4 \cdot (j + 1))$$

5.2.1 Simulation Relation

We define a simulation relation $consis_{MX}$ between states of *MX* and VAMP assembly machines.

The simulation relation $consis_{MX}(c_{MX}, info_{MX}, c_{ASM})$ states that the assembly configuration c_{ASM} encodes the *MX* configuration c_{MX} via the compiler information $info_{MX}$. It has the similar structure as the simulation relation introduced for *C-IL*.

◀ DEFINITION 5.14
Compiler Consistency

$$consis_{MX} :: C_{MX} \times infoT_{MX} \times C_{\text{ASM}} \mapsto bool$$

$$\frac{consis^{control}_{MX}(c_{MX}, info_{MX}, c_{ASM}) \qquad consis^{data}_{MX}(c_{MX}, info_{MX}, c_{ASM})}{consis_{MX}(c_{MX}, info_{MX}, c_{ASM})}$$

Control Consistency

Control consistency comprises program counter consistency $consis^{pc}_{MX}(c_{MX}, info_{MX}, c_{ASM})$ and return address consistency $consis^{ra}_{MX}(c_{MX}, info_{MX}, c_{ASM})$.

◀ DEFINITION 5.15
Control Consistency

$$\frac{consis^{pc}_{MX}(c_{MX}, info_{MX}, c_{ASM}) \qquad consis^{ra}_{MX}(c_{MX}, info_{MX}, c_{ASM})}{consis^{control}_{MX}(c_{MX}, info_{MX}, c_{ASM})}$$

The program counter consistency states that the assembly configuration's program counters point to the current *MX* statement(instruction) to be executed in a given *MX* machine.

◀ DEFINITION 5.16
Program Counter Consistency

$$\frac{\begin{array}{c} c_{ASM}.pc = c_{ASM}.dpc + 4 \\ c_{ASM}.dpc = info_{MX}.code_{ia}(c_{MX}.f_{top}, c_{MX}.loc_{top}) \end{array}}{consis^{pc}_{MX}(c_{MX}, info_{MX}, c_{ASM})}$$

The return address consistency states that return addresses of all stack frames point directly behind the code of call instructions which generated these stack frames.

◀ DEFINITION 5.17
Return Address Consistency

$$\frac{\begin{array}{c} \forall 0 < i < |c_{MX}.stack^{flat}|.\ c_{ASM}.m_{word}(stack^{ra}_{MX}(i, c_{MX}.stack^{flat}, c_{ASM}.m, info_{MX}) = \\ info_{MX}.code_{ia}(c_{MX}.f_{i-1}, c_{MX}.loc_{i-1}) \end{array}}{consis^{ra}_{MX}(c_{MX}, info_{MX}, c_{ASM})}$$

Data Consistency

The data consistency comprises code consistency $consis^{code}_{MX}$, memory consistency $consis^{mem}_{MX}$, stack consistency $consis^{stack}_{MX}$, and register consistency $consis^{regs}_{MX}$.

DEFINITION 5.18 ► Data Consistency

$$\frac{\begin{array}{ll} consis^{code}_{MX}(c_{MX}, info_{MX}, c_{ASM}) & consis^{mem}_{MX}(c_{MX}, info_{MX}, c_{ASM}) \\ consis^{stack}_{MX}(c_{MX}, info_{MX}, c_{ASM}) & consis^{regs}_{MX}(c_{MX}, info_{MX}, c_{ASM}) \end{array}}{consis^{data}_{MX}(c_{MX}, info_{MX}, c_{ASM})}$$

The code consistency requires that the compiled code is stored at the program base address in memory of a VAMP assembly configuration. That is, the compiled *MX* program must not be changed by statements(instructions) being executed by a *MX* machine.

DEFINITION 5.19 ► Code Consistency

$$\frac{\forall i < |info_{MX}.code|.\ info_{MX}.code[i] = \textit{int-to-instr}(c_{ASM}.m_{word}(info_{MX}.code_{ba} + 4 \cdot i))}{consis^{code}_{MX}(c_{MX}, info_{MX}, c_{ASM})}$$

The memory consistency states that the memory content of both machines is the same. Since the stack is abstracted we do not state memory equality for the region where the stack resides. As well we do not state the memory equality for the region where the compiled code is allocated.

DEFINITION 5.20 ► Memory Consistency

$$\frac{\forall a \in \mathbb{N}_{32}.\ \begin{array}{c} a \notin [info_{MX}.stack_{ba} : info_{MX}.stack_{ba} - info_{MX}.stack_{max-size}) \\ \wedge\ a \notin [info_{MX}.code_{ba} : info_{MX}.code_{ba} + 4 \cdot |info_{MX}.code|) \\ \Longrightarrow \\ c_{MX}.\mathcal{M}(a) = c_{ASM}.m_{\text{word}}(a) \end{array}}{consis^{mem}_{MX}(c_{MX}, info_{MX}, c_{ASM})}$$

The register consistency argues about the content of the stack pointer and base pointer registers of the assembly configuration c_{ASM}. Do not forget that we have already argued about the content of program counter in the program counter consistency before. The stack- and base pointer registers should have values according to their intended meaning from Table 2.2. Namely, GPR bp should point to the base address of the top stack frame in the *MX* machine c_{MX}. The difference between values stored in GPRs sp and bp of the assembly machine is the distance between frame base addresses for the top stack frame of the *MX* machine (cf. Figure 5.1). This consistency requires the equality of all GPRs (except for sp and bp) of the assembly machine and the μ_{ASM} active context of the *MX* machine. Also, it requires the equality of all SPRs of the assembly and *MX* machines. In case the stack of the *MX* exists the aforementioned statements hold. The aforementioned conditions hold in case the stack of the *MX* machine is set up and they are grouped in the register sub-consistency $consis^{regs(1)}_{MX}$.

DEFINITION 5.21 ► Register Consistency Stack Is Set Up

$$\frac{\begin{array}{c} is\_stack(c_{MX}) \\ c_{ASM}.gpr[bp] = frame^{ba}_{MX}(top(c_{MX}), c_{MX}.stack^{flat}, info_{MX}) \\ c_{ASM}.gpr[sp] = c_{ASM}.gpr[bp] - dist^{bp}_{MX}(top(c_{MX}), c_{MX}.stack^{flat}, info_{MX}) \\ c_{MX}.stack_{top} \in frame_{\mu} \Longrightarrow \forall i < 32.\ i \notin \{sp, bp\} \Longrightarrow c_{MX}.ac.gpr[i] = c_{ASM}.gpr[i] \\ \forall i < 32.\ c_{MX}.spr[i] = c_{ASM}.spr[i] \end{array}}{consis^{regs(1)}_{MX}(c_{MX}, info_{MX}, c_{ASM})}$$

If there is no stack in the *MX* configuration, then all corresponding registers of the *MX* and VAMP assembly machines must have the same values. This is expressed in the register sub-consistency $consis_{MX}^{regs(2)}$.

◀ DEFINITION 5.22
Register Consistency
(No Stack)

$$\frac{c_{MX}.ac \in context_{\mu_{\text{ASM}}} \qquad \neg is\_stack(c_{MX}) \qquad \forall i < 32.\ c_{MX}.ac.gpr[i] = c_{ASM}.gpr[i] \wedge c_{MX}.spr[i] = c_{ASM}.spr[i]}{consis_{MX}^{regs(2)}(c_{MX}, info_{MX}, c_{ASM})}$$

The above defined register consistencies form the *MX* register consistency in the following way:

◀ DEFINITION 5.23
Register Consistency

$$\frac{consis_{MX}^{regs(1)}(c_{MX}, info_{MX}, c_{ASM}) \vee consis_{MX}^{regs(2)}(c_{MX}, info_{MX}, c_{ASM})}{consis_{MX}^{regs}(c_{MX}, info_{MX}, c_{ASM})}$$

The stack consistency comprises frame header consistency $consis_{MX}^{fh}$, item consistency $consis_{MX}^{item}$, and top stack frame consistency $consis_{MX}^{top}$. The top stack frame consistency is considered only in case the active context of a given *MX* machine c_{MX} is of *C-IL* type.

◀ DEFINITION 5.24
Stack Consistency

$$\frac{consis_{MX}^{fh}(c_{MX}, c_{ASM}) \qquad consis_{MX}^{item}(c_{MX}, info_{MX}, c_{ASM}) \qquad c_{MX}.stack_{top} \in frame_{C\text{-}IL} \Longrightarrow consis_{MX}^{top}(c_{MX}, info_{MX}, c_{ASM})}{consis_{MX}^{stack}(c_{MX}, info_{MX}, c_{ASM})}$$

The frame header consistency requires the values in the frame headers of the stack frames to be consistent with their intended meaning (cf. Table 3.2). We do not argue about the return addresses in this consistency as it is already covered by the return address consistency. As there is no return destination component in frame headers of μ_{ASM} stack frames we do not state the corresponding consistency in this case. Note that for the first stack frame there is nothing to be stated as there is nothing behind the first stack frame.

◀ DEFINITION 5.25
Frame Header Consistency

$$\frac{\begin{array}{c}\forall\ 0 \le i \le top(c_{MX}).\\ i \neq 0 \longrightarrow stack_{MX}^{pbp}(i, c_{MX}.stack^{flat}, c_{ASM}.m, info_{MX}) = frame_{MX}^{ba}(i-1, c_{MX}.stack^{flat}, info_{MX})\\ \wedge\quad i \neq top(c_{MX}) \wedge c_{MX}.stack^{flat}[i] \in frame_{C\text{-}IL} \wedge c_{MX}.stack^{flat}[i].rds \neq \perp \longrightarrow\\ stack_{MX}^{rds}(i, c_{MX}.stack^{flat}, c_{ASM}.m, info_{MX}) = val2Int(read^{\theta}(c_{MX}, c_{MX}.stack^{flat}[i].rds))\end{array}}{consis_{MX}^{fh}(c_{MX}, info_{MX}, c_{ASM})}$$

The top stack frame consistency states where input parameters and local variables of the function call are saved. The value of an input parameter (local variable) is either saved in the register or put on the stack.

◀ DEFINITION 5.26
Top Stack Frame Consistency

$$\frac{\begin{array}{c}\forall\ i < |c_{MX}.\mathcal{P}_{top}.\mathcal{V}|.\\ info_{MX}.lvar_{reg}(v_i, c_{MX}.f_{top}, c_{MX}.loc_{top}, info_{MX}) = \perp \longrightarrow\\ \quad c_{MX}.\mathcal{M}_{\mathcal{E}top}(v_i) = c_{ASM}.m_{\text{word}}(c_{ASM}.gpr[bp] + info_{MX}.lvar_{off}(v_i, c_{MX}.f_{top}))\\ \wedge\\ info_{MX}.lvar_{reg}(v_i, c_{MX}.f_{top}, c_{MX}.loc_{top}, info_{MX}) \neq \perp \longrightarrow\\ \quad c_{MX}.\mathcal{M}_{\mathcal{E}top}(v_i) = c_{ASM}.gpr[info_{MX}.lvar_{reg}(v_i, c_{MX}.f_{top}, c_{MX}.loc_{top})]\end{array}}{consis_{MX}^{top}(c_{MX}, info_{MX}, c_{ASM})}$$

where $(v_i, t_i) = c_{MX}.\mathcal{P}_{top}.\mathcal{V}(i)$ is the i-th declaration in the list of parameters and local variables of the function corresponding to the top most stack frame.

The item consistency exposes the complete stack layout of μ_{ASM} stack frames of the *MX* machine in memory of the assembly machine Figure (cf. 5.1). Note that we do not state the consistency for components of stack frame headers as it is already covered by the return address and frame header consistencies. Here we state the relation between each value stored in the memory region corresponding to a μ_{ASM} stack frame and its abstracted version. This consistency is similar to the μ_{ASM} item consistency, here we need to consider additional case to state the consistency for passed parameters on stack. That case is, if a stack frame on top [4] of a μ_{ASM} stack frame is of *C-IL* type, then the values of parameters are read from the local memory environment of that *C-IL* stack frame.

We introduce a few shorthands

$$\begin{aligned} s &= c_{MX}.stack^{flat} \\ s_i &= c_{MX}.stack^{flat}[i] \\ saved_i &= s_i.saved \\ lifo_i &= s_i.lifo \\ pars_i &= s_i.pars \\ v_{ij} &= c_{MX}.\mathcal{P}_i.\mathcal{V}(j) \end{aligned}$$

that are used in the definition of the item consistency defined as follows:

DEFINITION 5.27 ►
Item Consistency

$$\frac{\begin{array}{c} \forall i \leq top(c_{MX}).\ \ s_i \in frame_\mu \longrightarrow \\ \forall\ j < dist^{bp}_{MX}(i, s, info_{MX}). \\ stack^{item}_{MX}(s, i, j, c_{ASM}.m, info_{MX}.stack_{ba}) = \\ \begin{cases} saved_i[j] & \text{if } 0 \leq j < |saved_i| \\ lifo_i[j - |saved_i|] & \text{if } |saved_i| \leq j < |saved_i| + |lifo_i| \\ pars_{i+1}[|pars_{i+1}| - 1 - (j - |saved_i| - |lifo_i|)] & \text{if } s_{i+1} \in frame_\mu \\ val2Int(read^\theta_{\mathcal{E}}(s_{i+1}, v_{ij})) & \text{otherwise, i.e. } s_{i+1} \in frame_{C\text{-}IL} \end{cases} \end{array}}{consis^{item}_{MX}(c_{MX}, info_{MX}, c_{ASM})}$$

Please note that for the top stack frame the third case is never considered (by definition of the function $dist^{bp}_{MX}$).

In the following we state the mixed compiler correctness and sketch its proof.

THEOREM 5.28 ►
Mixed Compiler Correctness

Let π be a *MX* program, c^0_{ASM} be the initial state of a target machine. Then for any sequence of a target machine: c^0_{ASM}, c^1_{ASM}, ..., there exists a c_{MX} machine sequence c^0_{MX}, c^1_{MX}, ... that does not get stuck and does not produce a stack overflow. Moreover, for any number steps i of a *MX* machine there exists the corresponding host machine state $c^{s(i)}_{ASM}$, such that the compiler consistency $consis_{IL}(c^i_{MX}, info_{MX}, c^{s(i)}_{ASM})$ holds if the statement to be executed at step i by the c_{MX} machine is an IO point. However, these states are consistent if the following requirements about the *MX* and VAMP assembly machines are fulfilled:

- The *MX* program π is translatable,
- The μ_{ASM} configuration c_μ and VAMP assembly configuration c_{ASM} are valid and they are consistent,

[4] here top does not mean the top stack frame, it means the stack frame above.

- After performing i steps the machine c_{MX} did not get stuck,

Formally, this theorem is stated as follows:

$$
\begin{array}{l}
\exists\, info_{MX}.\ \forall\ (c^i_{ASM})_i, \exists (c^i_{MX})_i. \\
\quad (\forall\ i. \quad c^{i+1}_{ASM} = \delta_{\mathrm{ASM}}(c^i_{ASM}) \\
\qquad \wedge \quad \pi, \theta \vdash c^i_{MX} \rightarrow_{MX} c^{i+1}_{MX} \\
\qquad \wedge \quad \text{the execution of } c_{MX} \text{ does not get stuck} \\
\qquad \wedge \quad \text{there is no stack overflow}) \\
\wedge \quad \exists\, s :: \mathbb{N} \mapsto \mathbb{N}. \\
\qquad (\forall\ i.\ info_{MX}.iop(c^i_{MX}.f_{top}, c^i_{MX}.loc_{top}) \\
\qquad\qquad \longrightarrow consis_{MX}(c^i_{MX}, info_{MX}, c^{s(i)}_{ASM}))
\end{array}
$$

Proof: We prove this theorem by induction on the number of steps i performed by the *MX* machine.

For the induction start we have to show that both machines are consistent, i.e. $consis_{MX}(c^0_{MX}, info_{MX}, c^{s(0)}_{ASM})$. We need to show that the target machine c_{ASM} executes instructions that perform the corresponding initializations and achieves the corresponding consistent state. If the *MX* machine starts its execution in the *C-IL* context, then the first statement to be executed by the *MX* machine c^0_{MX} must be an IO point. At this place we assume that the C compiler does not perform any kind of optimizations before the first statement in the main C function.

For the induction step we have to conclude from step i to i+1. Here we distinguish three cases:

- *C-IL* or μ_{ASM} pure steps: In this case we apply the correctness of the corresponding compiler(macro-assembler).
- inter-language statement: We list main arguments which must show this case:
  - the compiled inter-language statements (instructions) and non-external-calls and -returns obey the same compiler calling convention,
  - the content preservation of callee-save registers are preserved across *C-IL* function calls (cf. Theorem 4.27).
  - the content preservation of callee-save registers across *C-IL* to μ_{ASM} calls. These registers can be changed during the execution μ_{ASM} context. From the *MX* semantics we know that if μ_{ASM} context changes one of callee-save registers then the *MX* execution gets stuck at return to the *C-IL* execute context whose callee-registers were overwritten (the rule: Return from μ_{ASM} to *C-IL*). If the *MX* semantics had not give guarantees on the callee-save registers in this case, then the work-around would be to have the indices of these callee-save registers in the *uses* declaration list of the procedure corresponding to this μ_{ASM} execution context. In this case their content preservation is guaranteed by the μ_{ASM} semantics.

■

CHAPTER 6

Small Hypervisor Correctness

In this chapter we present the paper-and-pencil proof of the correctness of a simple hypervisor, called *baby hypervisor* (BHV) in [AHPP10]. The baby hypervisor is a small software system implemented in two programming languages: C and VAMP macro assembly, and it virtualizes VAMP ISA ([Tve09], [Tsy09]) and allows to run a number of guest machines, called *partitions*, on a single so-called *host processor*. Note that VAMP ISA was abstracted to VAMP assembly in Chapter 2 and VAMP assembly differs from VAMP ISA only in data representations, but they are semantically equivalent (proven by Tsyban in [Tsy09]). Thus, it suffices to show that the baby hypervisor virtualizes VAMP assembly.

The baby hypervisor always lets the guests run in user mode on the host. Hence, the guests run translated and unprivileged, i.e. every memory access performed by guests is the subject of the address translation of the host and it is forbidden for them to execute privileged instructions on the host. So that, the baby hypervisor has the full control over the guests. The guests of the baby hypervisor have their own notion of execution mode[1], i.e. there are *guest user mode* and *guest system mode*. That is, each guest can think that it runs in system mode on the host, indeed it runs in guest system mode in its own "virtual world" and in user mode on the host. The presence of two guest execution modes means that there are two address translations (one for each mode) that are performed for the running guests on the host. Two address translations for guests means that there are two virtual memories for them. These guest virtual memories are supported by so-called *host-* and *shadow page tables* [ACH+10] that are set-up for guests running in guest system mode and in guest user mode, respectively. In [AHPP10] on 9 page the authors write: "The host page tables will map injectively into host memory with different regions allocated for each guest. The shadow page table of a guest is set up as the concatenation of the guest's own page table and its host page table. To make sure that the guest cannot break this invariant, we map all host or shadow page table entries to the guest page table read-only. Thus, attempts of a guest

[1] this differs hypervisors from kernels as the guests(processes) of kernels run only in user mode on the host while the guests of hypervisors can run in user mode and in system mode on the host

to edit its page table or perform a privileged operation (e.g., change the page-table origin) will cause an exception on the host and be intercepted by the hypervisor. The hypervisor will then emulate the exception operation in software."

The way we proceed in this chapter is we first specify the abstract baby hypervisor. Next we examine the baby hypervisor implementation data structures used in [AHPP10] and invariants stated over them defined in [AHPP10]. Then we present intended invariants defined in [AHPP10] for the simulation of guests on the host. It is mainly the generalization of the $\mathcal{B}$-relation from [GHLP05]. In [Tsy09] Tsyban presented the detailed definition of such a relation and its application for showing the correctness of the CVM microkernel. In [AHPP10] the authors showed two main proof's obligations: (i) the functional correctness of the baby hypervisor's main function implemented in C, and (ii) the simulation of guests while they run without exceptions on the host. In the end we will state the baby hypervisor correctness theorem where we treat the μ_{ASM} code of the BHV program and the compiler calling convention. Note that the μ_{ASM} code and the compiler calling convention were neither treated nor presented in [AHPP10].

In the rest of the thesis we refer to the baby hypervisor as *BHV*.

6.1 Abstract BHV

6.1.1 Configurations

The abstract(spec) BHV configurations c_{HV} are represented by the record type C_{HV}

DEFINITION 6.1 ► BHV Configuration

$$C_{\text{HV}} \stackrel{\text{def}}{=} \{gs :: \mathbb{N} \mapsto C_{\text{ASM}},\ cg :: \mathbb{N}\}$$

comprising two components:

- gs is the mapping of guest ids to VAMP assembly machines representing guests,
- cg is the id of a guest that is currently making steps (called *active guest*). The other guests do not make any progress at all, they wait until they will be scheduled (become active). So we call them *sleeping guests*.

Valid Configurations The requirements over BHV configurations are: (i) the number of modeled guests is restricted by a given number ng, and (ii) each modeled guest g_{id} has at most $gptl(g_{id})$ PTEs. These requirements are captured by the predicate $valid_{BHV}$ defined below.

DEFINITION 6.2 ► Valid BHV Configuration

$$valid_{BHV} :: C_{\text{HV}} \times \mathbb{N} \times (\mathbb{N} \mapsto \mathbb{Z}) \mapsto bool$$

$$\frac{c_{\text{HV}}.cg < ng \qquad \forall\ g_{id} < ng.\ c_{\text{HV}}.gs(g_{id}).spr[ptl] \leq gptl(g_{id})}{valid_{BHV}(c_{\text{HV}}, ng, gptl)}$$

Init Configuration We denote by c_{HV}^{init} the initial BHV configuration that is defined as in [AHPP10]. It is not explicitly defined in [AHPP10], there is a link to BHV sources and there one can look it up.

6.1.2 Transition Function

The BHV transition function $\delta_{\mathrm{HV}} :: C_{\mathrm{HV}} \times \mathbb{N} \mapsto C_{\mathrm{HV}}$ takes a BHV configuration c_{HV} and the number ng of modeled guests as input and yields an updated BHV configuration c'_{HV}. This transition function lets the active guest to perform a single step. The next guest is scheduled to be active if the currently active guest is running in system mode and it tries to execute the trap instruction. Such a step is considered as a hypercall. BHV supports a single hypercall like this to schedule the next guest to run. We introduce the predicate *is-hypercall* to indicate this case:

◀ DEFINITION 6.3
Is Hypercall

$$\frac{\mathit{trap}(c_{ASM}) \wedge \mathit{system?}(c_{ASM}.\mathit{spr}[\mathit{mode}]))}{\mathit{is\text{-}hypercall}(c_{ASM})}$$

We denote by $\tilde{g}$ the active guest configuration of c_{HV}, i.e., $\tilde{g} = c_{\mathrm{HV}}.gs(c_{\mathrm{HV}}.cg)$. The BHV transition function is specified as follows:

◀ DEFINITION 6.4
BHV Steps Without Hypercalls

$$\frac{\neg \mathit{is\text{-}hypercall}(\tilde{g})}{\delta_{\mathrm{HV}}(c_{\mathrm{HV}}, ng) = c_{\mathrm{HV}}[gs := c_{\mathrm{HV}}.gs[cg := \delta_{\mathrm{ASM}}(\tilde{g})]]}$$

◀ DEFINITION 6.5
BHV Steps With Hypercalls

$$\frac{\mathit{is\text{-}hypercall}(\tilde{g})}{\delta_{\mathrm{HV}}(c_{\mathrm{HV}}, ng) = c_{\mathrm{HV}}\left[\begin{array}{lcl} gs & := & c_{\mathrm{HV}}.gs[cg := \delta_{\mathrm{ASM}}(\tilde{g})], \\ cg & := & c_{\mathrm{HV}}.cg + 1 \ \mathbf{mod}\ ng \end{array}\right]}$$

6.2 BHV Implementation

Here we examine BHV implementation details presented in [AHPP10]. We first examine data structures used in the BHV implementation.

6.2.1 Data Structures

The C data structure `proc_t` represents the state of the sleeping guests. Such a state is called *program control block* (PCB). This structure comprises all components of an abstract guest (besides the memory component) represented by a VAMP assembly configuration.

Listing 6.1: Data Structure `proc_t`

```
1 struct proc_t {
2     unsigned __int32 gpr[32]; // general-purpose register file
3     unsigned __int32 spr[32]; // special-purpose register file
4     unsigned __int32 dpc;     // delayed PC
5     unsigned __int32 pcp;     // primed PC
6 };
```

The C data structure `guest_t` comprises the guest PCB C data structure and additional components describing the guest: (i) the pointer to a guest memory (represented as array of unsigned integers) and the base page index of this memory (called the guest memory origin), (ii) the pointer to an array of host PTEs representing host page table, the base page index of this table (called the host page table origin) and the maximum

number of PTEs in this array, (iii) the pointer to an array of shadow PTEs representing shadow page table, the base page index of this table (called the shadow page table origin) and the maximum number of PTEs in this array.

Listing 6.2: Data Structure guest_t

```
1  struct guest_t {
2      struct proc_t pcb;
3      unsigned __int32 *gm;       // guest memory (array)
4      unsigned __int32 gmo;       // guest memory origin
5      unsigned __int32 *hpt,      // array of host PTEs
6      unsigned __int32 hpto;      // host page−table origin
7      unsigned __int32 *spt;      // array of shadow PTEs
8      unsigned __int32 spto;      // shadow page−table origin
9      unsigned __int32 max_ppx; // maximum number of physical PTEs
10     unsigned __int32 max_vpx; // maximum number of shadow PTEs
11 };
```

The C data structure hv_t is considered as main one and it comprises: (i) the number of modeled guests in BHV, (ii) the stack base address of the BHV program, (iii) the pointer to an array of guest C data structures, and (iv) the pointer to the active guest C data structure.

In [AHPP10] the structure hv_t is defined without the stack base address field.

Listing 6.3: Data Structure hv_t

```
1  struct hv_t {
2      unsigned __int32 ng;   // number of guests
3      unsigned __int32 sba; // stack base address
4      struct guest_t *cg;    // pointer to current guest
5      struct guest_t *g;     // array of guests
6  };
```

6.2.2 Parameters

BHV is configured at boot time by a few parameters encoded in four bytes at the beginning of its data segment base address.

- DS_START = 8192 : is the base address in the memory at which the BHV C data structure *hv_t* resides. In [AHPP10] this constant is 4096. The reason why we changed it is that we need to have a scratch memory for boot code and macro-assembly portions.
- DS_SIZE the data segment size. The data segment contains allocated BHV C data structures and the stack used by BHV.
- NG = 8 : is the number of guests in BHV,
- MAX_PPX = 2^{20} : is the maximum number of PTEs per host page table,
- MAX_VPX = 2^{20} : is the maximum number of PTEs per guest page table,

Note that the size of a memory page is PAGE_SIZE = 1024. We use the constant LOG_PAGE_SIZE to denote the binary logarithm of PAGE_SIZE, i.e.

$$\texttt{LOG\_PAGE\_SIZE} = log_2(\texttt{PAGE\_SIZE}) = 10$$

The stack base address is not passed a parameter to the BHV program, but it is computed as the sum of DS_START and DS_SIZE.

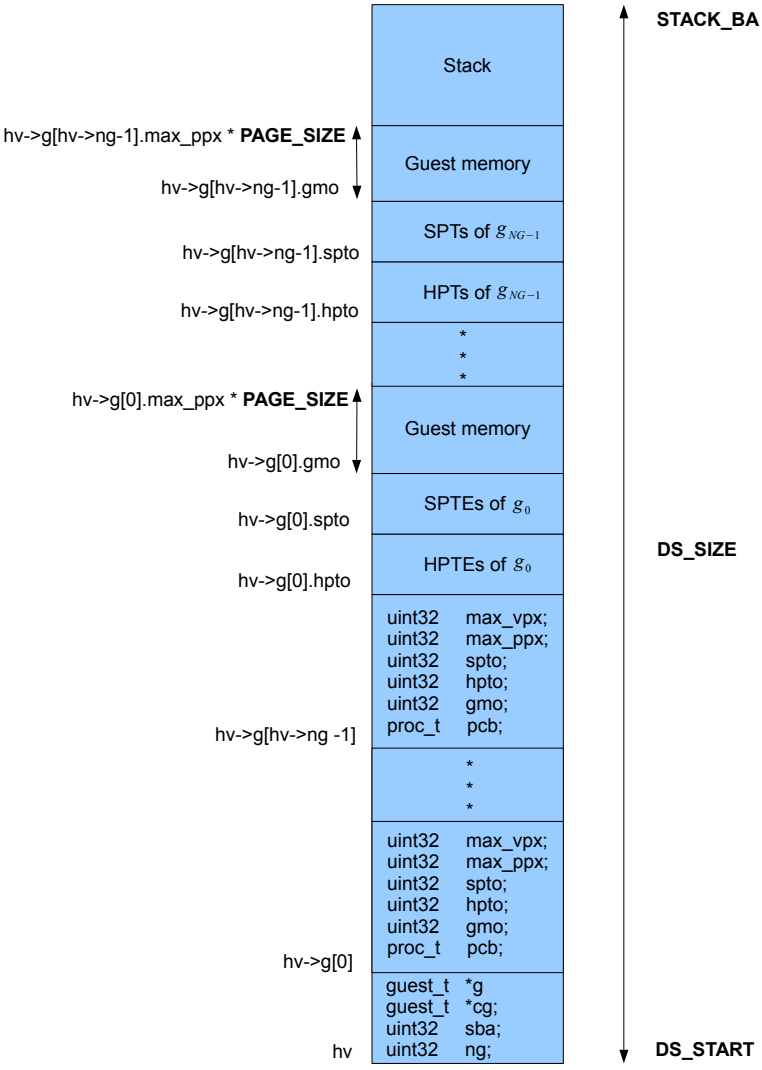


Figure 6.1: Memory Layout of BHV Data Structure

6.2.3 Implementation

The BHV program allocates its data structures in the data segment every time right after the reset signal happens. Fig. 6.1 depicts the memory layout of the data segment

after BHV allocated its data structures in that segment. In the figure the constant *hv* is used and it is defined as

```
#typedef  hv ((struct hv_t *)DS_START)
```

The data segment should be big enough to contain BHV data structures and the stack being used by the BHV program. The stack base address is the highest address that belongs to the data segment in memory. We are not going to present the complete BHV C implementation, the reader can look it up in [AHPP10].

Here we present the implementation of a mixed code from the BHV program. Note that it is not a part of the BHV program presented in [AHPP10]. This code is always executed if any interrupt happens during the execution of the guest on the host. This code is the interrupt service routine of the BHV program and it is split into 4 parts:

1. *Save Guest.* In lines 1-9 GPRs of the last running guest on the host are saved to the PCB corresponding to the last interrupted guest. Namely, the instruction at line 1 saves temporary the value of GPR register 2 in scratch memory since a free register is needed to contain the PCB base address. In line 2 the base address of the PCB is loaded to the GPR register 2. In lines 3-4 the GPR registers 0 and 1 of the host are saved in the corresponding GPRs of the PCB in memory. In lines 5-6 this value is restored in GPR 1 of the host and then saved in GPR register 2 of the PCB in memory. Starting from line 7 until line 9 the GPR registers 3-31 of the host are saved in the corresponding GPRs of the PCB in memory. In lines 10-13 the exception DPC and PC registers of the host machine are saved to the corresponding normal (i.e. non-exception) registers of the same PCB as before.

2. *Set Up Stack Pointers, Parameters Passing and Handler Invocation.* In lines 15-23 the stack pointer and base pointer are initialized. In case of the reset interrupt the stack base address is read from the scratch memory. Otherwise, it is read from the BHV data structure. In lines 26-27 exception cause and exception data registers are passed in first two input registers (i.e. we follow the *CCL* defined in Sec.3.1.4) to the BHV C function *hv_dispatch* that handles the caused interrupt. This function gets invoked with a *call* macro in line 29.

3. *Guest Restore.* After handling the interrupt the execution mode of the scheduled guest is checked in lines 31-34. If the execution mode of the scheduled guest is user mode, then the shadow page table is loaded on the host (lines 35-38) Otherwise, in lines 42-45 the host page table is loaded on the host.

   In lines 48-51 DPC and PC registers of the PCB corresponding to the scheduled guest are written to the corresponding exception registers of the host. In lines 53-58 GPRs of the scheduled guest PCB are saved to GPRs of the host.

4. *Return From Exception*. In line 56 the instruction `rfe` starts the execution of the loaded (restored) guest on the host.

Later in this chapter we refer to code consisting of lines 1-23 and lines 31-58 from Listing 6.4 as interrupt service routines 1 (*ISR1*) and 2 (*ISR2*), respectively.

Listing 6.4: BHV Context Switch Implementation

```
sw r2 r0 4096         // store r2 in scratch memory
lw r2 r0 8200         // get PCB address, 8200 = &(HV->cg)
sw r0 r2 0
sw r1 r2 4
lw r1 r0 4096         // restore r2 and save it to r1
sw r1 r2 8            // save it to PCB
sw r3 r2 12
...
sw r31 r2 124
movs2i EPC r7        // save EPC to PCB
sw r7 r2 140         // OFFSET(proc_t, spr[EPC] = 140)
movs2i EDPC r7       // save EDPC to PCB
sw r7 r2 144         // OFFSET(proc_t, spr[EDPC] = 144)

movs2i ECA r1
ifnez r1 goto LSBA_BHV
lw   r1 r0 4100          // read sba from scratch memory
goto LSBA_BHV_SKIP

LSBA_BHV:
lw r1 r0 8196   // read sba from BHV DS, 8196 = &(HV->sba)
add  r30 r0 r1           // initialize bp
add  r29 r0 r1           // initialize sp (bp = sp)

LSBA_BHV_SKIP:
movs2i ECA   r6      // fst input arg
movs2i EDATA r7      // snd input arg

CALL hv_dispatch     // invoke BHV handler

lw   r2 r0 8200         // load adr of current guest (r2 := &(HV->cg)))
addi r3 r2 0            // r3 := r2
lw   r4 r2 192  // r4 := pcb->spr[MODE], OFFSET(proc_t, spr[MODE] = 192
ifnez r4 goto LHOST          // if (r4 != 0) jump to LHOST
lw   r5 r2 164   // r5 := pcb->spr[PTL], OFFSET(proc_t, spr[PTL] = 164
movi2s PTL r5
lw   r5 r3  168  // r5 := cg->spto, OFFSET(proc_t, spr[PT0]) = 168
movi2s PTO r5
goto LHOST_SKIP

LHOST:
lw r5 r3 288      // r5 := cg->max_ppx, OFFSET(guest_t, max_ppx) =  288
movi2s PTL r5
lw r5 r3 276      // r5 := cg->hpto ,  OFFSET(guest_t, hpto) = 276
movi2s PTO r5

LHOST_SKIP:
lw   r5 r2  256      // r5 := pcb->dpc, OFFSET(proc_t, dpc) = 256
movi2s EDPC r5
lw   r5 r2  260      // r5 := pcb->pcp, OFFSET(proc_t, pcp) = 260
movi2s EPC r5

lw r0 r2 0
lw r1 r2 4
lw r3 r2 12
//...
lw r31 r2 124
lw r2 r2 8
rfe
```

6.3 Correctness

In this section we examine correctness criteria and a theorem of the baby hypervisor. We define criteria as invariants between the BHV C implementation, simulated guests, the host and the abstract BHV. In order to be able to talk about the BHV implementation we assume the followings: (i) the BHV program is properly encoded in *MX* program π, and (ii) the information about the C compiler being used to compile this C program is encoded in system parameters θ.

We first define an invariant called $\mathcal{B}$-relation that reconstructs virtual guests for every guest supported by the BHV implementation and check whether they match the corresponding guests of the abstract BHV. Then we examine the implementation invariants of the BHV implementation. Before we introduce auxiliary functions for reading guest components of the BHV data structures stored in memory of the *MX* configuration c_{MX}. Note that in [Tsy09] Tsyban defined similar functions reading C variables in memory of the host machine.

6.3.1 Reading Guest Components

We define the function $PCB :: \mathbb{N} \times C_{MX} \mapsto C_{\text{ASM}}$ that returns for a given g_{id} an assembly machine constructed from registers of the corresponding PCB in the memory of a given *MX* machine c_{MX}. Right now we are interested in the PCB of a guest. Thus, the memory of the assembly configuration returned by that function is left untouched.

$$PCB(g_{id}, c_{MX}) \stackrel{\text{def}}{=} g$$

where the assembly machine g is constructed as follows[2]:

$$\begin{array}{lcl} g.dpc & = & val2Nat([\texttt{hv} \rightarrow \texttt{g}[g_{id}] \rightarrow \texttt{pcb.dpc}]_{c_{MX}}^{\pi,\theta}) \\ g.pc & = & val2Nat([\texttt{hv} \rightarrow \texttt{g}[g_{id}] \rightarrow \texttt{pcb.pcp}]_{c_{MX}}^{\pi,\theta}) \\ g.gpr[i] & = & val2Int([\texttt{hv} \rightarrow \texttt{g}[g_{id}] \rightarrow \texttt{pcb.gpr}[i]]_{c_{MX}}^{\pi,\theta}) \\ g.spr[i] & = & val2Int([\texttt{hv} \rightarrow \texttt{g}[g_{id}] \rightarrow \texttt{pcb.spr}[i]]_{c_{MX}}^{\pi,\theta}) \end{array}$$

Below we list the definition of functions which return for a given guest g_{id} the components of the corresponding C guest data structure stored in memory of a given *MX* machine c_{MX}:

- host page table entry (*HPTE*) x of the guest g_{id}:

  $$HPTE(x, g_{id}, c_{MX}) \stackrel{\text{def}}{=} val2Nat([\texttt{hv} \rightarrow \texttt{g}[g_{id}] \rightarrow \texttt{hpt[x]}]_{c_{MX}}^{\pi,\theta})$$

  where $x \in \mathbb{N}$ is a HPTE index.

- shadow page table (*SPTE*) x of the guest g_{id}:

  $$SPTE(x, g_{id}, c_{MX}) \stackrel{\text{def}}{=} val2Nat([\texttt{hv} \rightarrow \texttt{g}[g_{id}] \rightarrow \texttt{spt[x]}]_{c_{MX}}^{\pi,\theta})$$

  where $x \in \mathbb{N}$ is a SPTE index.

- maximum number of PTEs for the host page table of the guest g_{id}:

  $$MPPX(g_{id}, c_{MX}) \stackrel{\text{def}}{=} val2Nat([\texttt{hv} \rightarrow \texttt{g}[g_{id}] \rightarrow \texttt{max\_ppx}]_{c_{MX}}^{\pi,\theta})$$

[2]Here and in the rest of this chapter we assume the implicit conversions from natural and integer numbers that appear within *C-IL* expressions to the corresponding *C-IL* expressions.

- maximum number of PTEs for the shadow page table of the guest g_{id}:

$$MVPX(g_{id}, c_{MX}) \stackrel{\text{def}}{=} val2Nat([\texttt{hv} \rightarrow \texttt{g}[g_{id}] \rightarrow \texttt{max\_vpx}]^{\pi,\theta}_{c_{MX}})$$

- HPT origin of the guest g_{id}:

$$HPTO(g_{id}, c_{MX}) \stackrel{\text{def}}{=} val2Nat([\texttt{hv} \rightarrow \texttt{g}[g_{id}] \rightarrow \texttt{hpto}]^{\pi,\theta}_{c_{MX}})$$

- SPT origin of the guest g_{id}:

$$SPTO(g_{id}, c_{MX}) \stackrel{\text{def}}{=} val2Nat([\texttt{hv} \rightarrow \texttt{g}[g_{id}] \rightarrow \texttt{spto}]^{\pi,\theta}_{c_{MX}})$$

- guest memory origin of the guest g_{id}:

$$GMO(g_{id}, c_{MX}) \stackrel{\text{def}}{=} val2Nat([\texttt{hv} \rightarrow \texttt{g}[g_{id}] \rightarrow \texttt{gmo}]^{\pi,\theta}_{c_{MX}})$$

6.3.2 $\mathcal{B}$-Relation

The $\mathcal{B}$-relation binds the guest processes configurations $c_{\text{HV}}.gs$ and virtual guest machines reconstructed from the *MX* configuration c_{MX} and the host configuration c_{ASM}. This relation is considered as a correctness criterion of guest processes. In order to define this relation we need to introduce a few functions performing:

- virtual guest machine reconstruction,
- VAMP assembly machines comparison.

Virtual Machines

We introduce the function $\mathcal{G}(g_{id}, c_{MX}, c_{ASM})$

$$\mathcal{G} :: \mathbb{N} \times C_{MX} \times C_{\text{ASM}} \mapsto C_{\text{ASM}}$$

to construct a virtual assembly machine corresponding to a given guest g_{id} which might be either running on the host c_{ASM} or be sleeping (suspended). That is, in the definition of the function $\mathcal{G}$ we distinguish two cases whether the guest g_{id} is active or not. The guest g_{id} is active if and only if the host machine c_{ASM} operates in user mode and the pointer $[\texttt{hv} \rightarrow \texttt{cg}]^{\pi,\theta}_{c_{MX}}$ points to the guest data structure corresponding to the guest g_{id}. Note that the *MX* machine c_{MX} runs the BHV program π and this machine is bound with the host machine c_{ASM} via the compiler consistency $consis_{MX}$ defined in Section 5.2.

Let us first consider the case when the guest g_{id} is active, i.e. the guest g_{id} is currently running on the host c_{ASM}. In this case the function $\mathcal{G}(g_{id}, c_{MX}, c_{ASM})$ returns the virtual guest representation corresponding to the guest g_{id} which is reconstructed from registers (except for SPRs)[3] of the host machine c_{ASM} and SPRs from the corresponding PCB stored in the memory of the *MX* configuration c_{MX} by means of the function $\mathcal{G}_{active}$. In another case, i.e. when the guest g_{id} is sleeping, its virtual representation is reconstructed from the corresponding PCB stored in the memory of the *MX* configuration c_{MX} with the help of the function $\mathcal{G}_{sleeping}$. In both cases the memory of a virtual assembly machine is reconstructed from the c_{MX} memory by the function *mem*. The definition of the functions $\mathcal{G}_{active}$, $\mathcal{G}_{sleeping}$ and *mem* will follow.

[3] In the baby hypervisor SPRs are virtualized. That is, SPRs of the host machine are not a part of the guest being executed on this host. Moreover, the guest cannot access SPRs while it is being executed on the host as the host operates in user mode (by the VAMP assembly semantics presented in Chapter 2)

DEFINITION 6.6 ►
Virtual Guest Machine

$$\mathcal{G}(g_{id}, c_{MX}, c_{ASM}) \stackrel{\text{def}}{=}$$

$$\begin{cases} \mathcal{G}_{active}(g_{id}, c_{MX}, c_{ASM}) & \text{if } user?(c_{ASM}.spr[mode]) \quad \wedge \\ & \quad [\texttt{hv} \rightarrow \texttt{cg}]^{\pi,\theta}_{c_{MX}} = [\&(\texttt{hv} \rightarrow \texttt{g}[g_{id}])]^{\pi,\theta}_{c_{MX}} \\ \mathcal{G}_{sleeping}(g_{id}, c_{MX}) & \text{otherwise} \end{cases}$$

The definition of the function $\mathcal{G}$ was constructed from the abstraction functions *abs* and *abs_gh* defined in [AHPP10].

Reconstructing VM: Active Guest Since the machine *MX* does not make any steps while the guest is running on the host the memory of the virtual guest is reconstructed from the host memory. For that, the memory of the *MX* machine is substituted with the memory of the host machine.

$$\mathcal{G}_{active}(g_{id}, c_{MX}, c_{ASM}) \stackrel{\text{def}}{=} g$$

where the assembly machine g is constructed as:

$$\begin{array}{lcl} g.dpc & = & c_{ASM}.dpc \\ g.pc & = & c_{ASM}.pc \\ g.gpr[i] & = & c_{ASM}.gpr[i] \\ g.spr[i] & = & PCB(g_{id}, c_{MX}).spr[i] \\ g.m & = & mem(g_{id}, c_{MX}[\mathcal{M} := c_{ASM}.m]) \end{array}$$

The definition of this function corresponds to the abstraction function *abs_gh* defined in [AHPP10].

Reconstructing VM: Sleeping Guest

$$\mathcal{G}_{sleeping}(g_{id}, c_{MX}) \stackrel{\text{def}}{=} g$$

where the assembly machine g is constructed in the following way:

$$\begin{array}{lcl} g.dpc & = & PCB(g_{id}, c_{MX}).dpc \\ g.pc & = & PCB(g_{id}, c_{MX}).pc \\ g.gpr[i] & = & PCB(g_{id}, c_{MX}).gpr[i] \\ g.spr[i] & = & PCB(g_{id}, c_{MX}).spr[i] \\ g.m & = & mem(g_{id}, c_{MX}) \end{array}$$

The definition of this function corresponds to the abstraction function *abs* defined in [AHPP10].

Reconstructing Virtual Memory The virtual memory of a guest g_{id} is reconstructed by the function *mem* directly from the memory of the configuration c_{MX} running the BHV program π. It does not matter whether a given guest g_{id} is active or not. The definition of this function relies on the following facts:

- the BHV program maintains the linear address translation for guests. This fact is expressed as a part of the host page table invariant presented later.
- the size of a guest memory does not change (after the memory was allocated), i.e. the memory cannot be dynamically allocated for guests (by the design of the BHV program).

Thus, the function *mem* is defined as

$$mem :: \mathbb{N} \times C_{MX} \mapsto (\mathbb{N} \mapsto \mathbb{B}^8)$$

$$mem(g_{id}, c_{MX}) \stackrel{\text{def}}{=} m$$

where the content of the virtual memory m at word address

$$a \in [0, \ldots, val2Nat([\texttt{hv} \rightarrow \texttt{g[}g_{id}\texttt{]} \rightarrow \texttt{max\_ppx}]^{\pi,\theta}_{c_{MX}}) \cdot \texttt{PAGE\_SIZE})$$

is

$$m_{word}(a) = val2Int([\texttt{hv} \rightarrow \texttt{g[}g_{id}\texttt{]} \rightarrow \texttt{gm[}a\texttt{]}]^{\pi,\theta}_{c_{MX}})$$

The definition of this function corresponds to the abstraction function *abs_m* defined in [AHPP10].

Identical VAMP Assembly Configurations

Two given VAMP assembly machines are identical(equal) if and only if the following is true:

- the content of memory at some restricted address space of both machines is the same. For that we introduce the equality function for byte addressable memories at addresses $[base, .., base + size)$

$$\frac{\forall a.\ base \leq a < base + size \longrightarrow m^1(a) = m^2(a)}{m^1 =_{base,size} m^2}$$

◀ DEFINITION 6.7
Assembly Memory Equality

- the content of PCs, DPCs, all GPRs and four SPRs (status register, page table origin, page table length and mode) of both machines are the same.

So that, the VAMP assembly machines equality function takes two assembly machines (c^1_{ASM} and c^2_{ASM}) and the number *size* of bytes to be compared in memories of both machines starting from address 0.

$$\frac{\begin{array}{c} c^1_{ASM}.dpc = c^2_{ASM}.dpc \qquad c^1_{ASM}.pc = c^2_{ASM}.pc \\ c^1_{ASM}.gpr = c^2_{ASM}.gpr \qquad c^1_{ASM}.spr[sr, pto, ptl, mode] = c^2_{ASM}.spr[sr, pto, ptl, mode] \\ c^1_{ASM}.m =_{0,size} c^2_{ASM}.m \end{array}}{c^1_{ASM} =_{size} c^2_{ASM}}$$

◀ DEFINITION 6.8
Assembly Configurations Equality

This function corresponds to the function *asm-equal* defined in [Tsy09].

$\mathcal{B}$ Relation The $\mathcal{B}$-relation is stated as the parameterized equality with the right amount of virtual memory between abstract guest machines gs and virtual guests reconstructed from a given configuration c_{MX} and from a given host machine c_{ASM}. The right amount of virtual memory is computed as $\texttt{PAGE\_SIZE} \cdot (\texttt{MAX\_PPX} + 1)$ as we know that each guest has such an amount of memory (from the implementation details of the BHV program [AHPP10])

$$\mathcal{B} :: (\mathbb{N} \mapsto C_{\text{ASM}}) \times C_{MX} \times C_{\text{ASM}} \times \mathbb{N} \mapsto bool)$$

$$\frac{\forall\ g_{id} < ng.\ \mathcal{G}(g_{id}, c_{MX}, c_{ASM}) =_{\texttt{PAGE\_SIZE}\cdot(\texttt{MAX\_PPX}+1)} gs(g_{id})}{\mathcal{B}(gs, c_{MX}, c_{ASM}, ng)}$$

◀ DEFINITION 6.9
$\mathcal{B}$ Relation

6.3.3 Implementation Invariants

We require a number of invariants to be hold over the BHV data structures during the execution of BHV code and guests. That is why we call them implementation invariants and we formally define them in the following.

Almost all implementation invariants examined here have been stated in [AHPP10]. Here we present them in our framework.

Page Table Entries Invariants

We first define auxiliary functions before presenting page table invariants. The functions

$$PX(a) \stackrel{\text{def}}{=} a \;/\; \texttt{PAGE\_SIZE}$$

and

$$PA(x) \stackrel{\text{def}}{=} x \cdot \texttt{PAGE\_SIZE}$$

defined in [AHPP10] return the page index of a given address a and the start address of the page indicated by a given page index x, respectively.

The predicate $v\_bus\_error(a, ptl)$ described in [AHPP10] indicates whether a given address a is outside of the page table with a given length ptl.

DEFINITION 6.10 ► Bus Error

$$v\_bus\_error :: \mathbb{N} \times \mathbb{N} \mapsto bool$$

$$\frac{ptl < PX(a)}{v\_bus\_error(a, ptl)}$$

The predicate $in\text{-}pt(x, pto, ptl)$ defined in [AHPP10] indicates whether a given page index x falls into the page table identified by a given page table origin pto and a given page table length ptl.[4]

DEFINITION 6.11 ► In Page Table

$$in\text{-}pt :: \mathbb{N} \times \mathbb{N} \times \mathbb{N} \mapsto bool$$

$$\frac{x \in \{pto, \ldots, pto + ptl/1024\}}{in\text{-}pt(x, pto, ptl)}$$

The function $ptea\_vpx(pto, x)$ defined in [AHPP10] computes the address of PTE corresponding to a given page index x in the page table with a given page table origin pto.

DEFINITION 6.12 ► PTE Physical Address

$$ptea\_vpx :: \mathbb{N} \times \mathbb{N} \mapsto \mathbb{N}$$

$$ptea\_vpx(pto, x) \stackrel{\text{def}}{=} PA(pto) + 4 \cdot x$$

Host PTE Invariant The invariant is indicated by the predicate $inv\_hpt\_entry(x, g_{id}, c_{MX})$ defined in [AHPP10]. It states a few conditions on the host PTE with index x of the guest g_{id} of the BHV implementation configuration c_{MX}. We introduce a few shorthands:

- $hpte = HPTE(x, g_{id}, c_{MX})$ is a host PTE,

[4] The size of a single memory page is 1024 words (4096 bytes) and the size of a single page table entry (PTE) is 1 word (4 bytes). So that, there are 1024 PTEs per a single page.

- $spr = PCB(g_{id}, c_{MX}).spr$ is SPRs from the PCB corresponding to the guest g_{id}.

to make nice and brief the host PTE invariant definition. The invariant states that the host PTE *hpte* must

- c_1: be valid ($v_{PTE}(hpte)$),
- c_2: point to the guest page x as stored on the host

  $$ppx_{PTE}(hpte) = GMO(g_{id}, c_{MX}) + x.$$

  This condition indicates that the BHV program maintains the linear address translation for guests.
- c_3: be protected if covered by the guest's page table

  $$p_{PTE}(hpte) \longleftrightarrow in\text{-}pt(x, spr[pto], spr[ptl])$$

The listed above conditions build the host PTE invariant which are gathered together in the predicate defined below.

◀ DEFINITION 6.13
Host PTE Invariant

$$\frac{c_1 \qquad c_2 \qquad c_3}{inv\_hpt\_entry(x, g_{id}, c_{MX})}$$

Shadow PTE Invariant The invariant is covered by the predicate $inv\_spt\_entry(x, g_{id}, c_{MX})$ that was defined in [AHPP10]. We introduce a few shorthands

- $spr = PCB(g_{id}, c_{MX}).spr$ is SPRs from the PCB corresponding the guest g_{id}
- *spte* is a shadow PTE corresponding to the page index x of the guest g_{id} and is computed as $SPTE(x, g_{id}, c_{MX})$,
- $gpte\_vpx = ptea\_vpx(spr[pto], x)$ is the address of the guest PTE x, and
- $gpte = [\texttt{hv} \rightarrow \texttt{g}[g_{id}] \rightarrow \texttt{gm}[gpte\_vpx]]^{\pi,\theta}_{c_{MX}}$ is the guest PTE.

to define this invariant in a nice and brief way. So the invariant states that the shadow PTE *spte* must:

- c_1 : The shadow SPT *spte* is valid if and only if the translation yields no guest bus error, the page index of the guest PTE does not exceed the maximum number of page indices allowed for the guest g_{id}, and the guest PTE is valid.

  $$\begin{aligned} v_{PTE}(spte) \quad \longleftrightarrow \quad & \neg v\_bus\_error(gpte\_vpx, MVPX(g_{id}, c_{MX})) \\ & \wedge \; ppx_{PTE}(gpte) \leq MVPX(g_{id}, c_{MX}) \\ & \wedge \; v_{PTE}(gpte) \end{aligned}$$

- if the shadow PTE is valid, then
  - c_2: it points to the host page designated by guest

    $$ppx_{PTE}(spte) = ppx_{PTE}(HPTE(ppx_{PTE}(gpte), g_{id}, c_{MX}))$$

  - c_3: it is protected if the guest page is protected or part of the guest page table.

    $$p_{PTE}(gpte) \; \vee \; in\text{-}pt(ppx_{PTE}(gpte), spr[pto], spr[ptl]) \longrightarrow p_{PTE}(spte)$$

Based on the conditions c_1, c_2 and c_3 introduced above the shadow PTE invariant defined as:

DEFINITION 6.14 ► Shadow PTE Invariant

$$\frac{c_1 \qquad v_{PTE}(spte) \longrightarrow c_2 \wedge c_3}{inv\_spt\_entry(x, g_{id}, c_{MX})}$$

The invariants presented above are stated over a single PTE. Thus, we define the predicate $pt\text{-}inv(c_{MX}, c_{ASM}, ng)$, called page table (or address manager) invariant, that requires host PTE and shadow PTE invariants to hold over all host- and shadow PTEs of all guests supported by BHV.

DEFINITION 6.15 ► Address Manager Invariant

$$\frac{\begin{array}{c}\forall\ g_{id} < ng.\ \forall\ x < MPPX(g_{id}, c_{MX}).\ inv\_hpt\_entry(x, g_{id}, c_{MX}, \mathcal{G}(g_{id}, c_{MX}, c_{ASM})) \\ \forall\ g_{id} < ng.\ \forall\ x < PCB(g_{id}, c_{MX}).spr[ptl].\ inv\_spt\_entry(x, g_{id}, c_{MX}, \mathcal{G}(g_{id}, c_{MX}, c_{ASM}))\end{array}}{pt\text{-}inv(c_{MX}, c_{ASM}, ng)}$$

Code Invariant

The code invariant states that a given compiled program code $info_{MX}.code$ resides at address $info_{MX}.code_{ba}$ in the memory of the host machine c_{ASM} and a given compiled boot code $code_{boot}$ resides at the first memory page in memory of the host machine. Both codes are represented as a list of assembly instructions.

DEFINITION 6.16 ► Code Invariant

$$\frac{\begin{array}{c}\forall\ i < |code_{boot}|.\ int\text{-}to\text{-}instr(c_{ASM}.m_{word}(i \cdot 4)) = code_{boot}[i] \\ \forall\ i < |info_{MX}.code|.\ int\text{-}to\text{-}instr(c_{ASM}.m_{word}(info_{MX}.code_{ba} + i \cdot 4)) = info_{MX}.code[i]\end{array}}{code\text{-}inv?(info_{MX}, code_{boot}, c_{ASM})}$$

Guest Steps

Here we present the invariant when the guest g_{id} is being executed on the host machine c_{ASM}. This invariant is captured by the predicate

$$guest\text{-}inv(g_{id}, c_{MX}, c_{ASM}) :: \mathbb{N} \times C_{MX} \times C_{\text{ASM}} \mapsto bool$$

and this predicate $guest\text{-}inv(g_{id}, c_{MX}, c_{ASM})$ comprises the following conditions:

- c_1: A given guest id g_{id} is indeed the current one. It is checked by looking up the pointer to the current guest PCB and comparing it with the address of the PCB corresponding to the guest g_{id}.

$$[\texttt{hv} \rightarrow \texttt{cg}]^{\pi,\theta}_{c_{MX}} = [\texttt{hv} \rightarrow \texttt{g}[g_{id}]]^{\pi,\theta}_{c_{MX}}$$

- c_2: The stack base address stored in the BHV program equals to the sum of DS_START and DS_SIZE, i.e.

$$val2Nat([\texttt{hv} \rightarrow \texttt{sba}]^{\pi,\theta}_{c_{MX}}) = \texttt{DS\_START} + \texttt{DS\_SIZE}$$

- An active guest runs either in guest system mode or in guest user mode. The content of SPRs *pto* and *ptl* of the host machine c_{ASM} depends on the execution mode of the guest g_{id}. The execution mode of the guest is stored in the SPR *mode* in the PCB corresponding to the guest g_{id}.

  If the guest g_{id} runs in guest system mode, then

– c_3: SPRs *pto* and *ptl* of the host c_{ASM} have values of the host PTO and the maximum number of PTEs per HPT, respectively, which are stored in the PCB corresponding to the guest g_{id}

$$\begin{aligned} c_{ASM}.spr[pto] &= HPTO(g_{id}, c_{MX}) \\ c_{ASM}.spr[ptl] &= MPPX(g_{id}, c_{MX}) \end{aligned}$$

In case the guest g_{id} runs in guest user mode the following holds:

– c_4: SPRs *pto* and *ptl* of the host have values of the shadow PTO and the SPR *ptl*, respectively. The latter is stored in the PCB corresponding to the guest g_{id}.

$$\begin{aligned} c_{ASM}.spr[pto] &= SPTO(g_{id}, c_{MX}) \\ c_{ASM}.spr[ptl] &= PCB(g_{id}, c_{MX}).spr[ptl]) \end{aligned}$$

- c_5: memory of the c_{MX} and c_{ASM} machines are the same in:

  – the address space not intersecting with the memory address space of a given guest g_{id}:

$$\begin{aligned} a \notin [&GMO(g_{id}, c_{MX}), .., GMO(g_{id}, c_{MX}) + PA(\texttt{MAX\_PPX} + 1)) \\ &\longrightarrow c_{MX}.\mathcal{M}(a) = c_{ASM}.m(a) \end{aligned}$$

  – the address space where the guest page table of a given guest g_{id} resides:

$$\begin{aligned} a \in [&PCB(g_{id}, c_{MX}).spr[pto], .., PCB(g_{id}, c_{MX}).spr[pto] + PA(PCB(g_{id}, c_{MX}).spr[ptl])] \\ &\longrightarrow c_{MX}.\mathcal{M}(a) = c_{ASM}.m(a) \end{aligned}$$

The conditions c_1, c_2, c_3, c_4 and c_5 form the guest implementation invariants defined as follows:

◄ DEFINITION 6.17
Guest Execution
Implementation Invariants

$$\frac{\begin{array}{c} \overset{c_1}{} \quad \overset{c_2}{} \quad \overset{c_5}{} \\ system?(PCB(g_{id}, c_{MX}).spr[mode]) \longrightarrow c_3 \\ user?(PCB(g_{id}, c_{MX}).spr[mode]) \longrightarrow c_4 \end{array}}{guest\text{-}inv(g_{id}, c_{MX}, c_{ASM})}$$

6.3.4 Putting Implementation Invariants Together

We gather all implementation invariants that must hold during guest steps in the predicate $impl\text{-}inv_{\text{user}}$. So during the guest steps the following must hold: code invariant, page table(address manager) invariants, and guest execution invariants.

◄ DEFINITION 6.18
Guest Execution Invariants

$$\frac{\begin{array}{c} code\text{-}inv?(info_{MX}, code_{boot}, c_{ASM}) \\ pt\text{-}inv(c_{MX}, c_{ASM}, ng) \\ guest\text{-}inv(c_{\text{HV}}.cg, c_{MX}, c_{ASM}) \end{array}}{impl\text{-}inv_{\text{user}}(c_{\text{HV}}, c_{MX}, info_{MX}, code_{boot}, c_{ASM}, ng)}$$

The predicate $impl\text{-}inv_{\text{hv}}$ contains all implementation invariants that must hold during the BHV execution: code invariant, page table(address manager) invariants, and the *MX* compiler consistency. Where the *MX* compiler consistency must hold if the active context of a given *MX* machine is of μ_{ASM} type or if the active context of a given

MX machine is of *C-IL* type and the current statement to be executed by the *MX* machine is an IO point.

DEFINITION 6.19 ► BHV Execution Invariants

$$\frac{\begin{array}{c} code\text{-}inv?(info_{MX}.code, info_{MX}.code_{ba}, code_{boot}, c_{ASM}) \\ pt\text{-}inv(c_{MX}, c_{ASM}, ng) \\ c_{MX}.ac \in context_{\mu_{\text{ASM}}} \longrightarrow consis_{MX}(c_{MX}, info_{MX}, c_{ASM}) \\ c_{MX}.ac \in context_{C\text{-}IL} \wedge info_{MX}.iop(c_{MX}.f_{top}, c_{MX}.loc_{top}) \longrightarrow consis_{MX}(c_{MX}, info_{MX}, c_{ASM}) \end{array}}{impl\text{-}inv_{\text{hv}}(c_{\text{HV}}, c_{MX}, info_{MX}, code_{boot}, c_{ASM}, ng)}$$

We collect all implementation invariants that must hold during guest and hypervisor steps in the predicate *impl-inv*? defined as:

DEFINITION 6.20 ► Implementation Invariants

$$\frac{\begin{array}{c} user?(c_{ASM}.spr[mode]) \longrightarrow impl\text{-}inv_{\text{user}}(c_{\text{HV}}, c_{MX}, c_{ASM}, c_{\text{HV}}.cg) \\ system?(c_{ASM}.spr[mode]) \longrightarrow impl\text{-}inv_{\text{hv}}(c_{\text{HV}}, c_{MX}, c_{ASM}) \end{array}}{impl\text{-}inv?(c_{\text{HV}}, c_{MX}, info_{MX}, code_{boot}, c_{ASM}, ng)}$$

6.3.5 Correctness Theorem

THEOREM 6.21 ► BHV Correcntess

Let π be a *MX* program representing a given BHV program, c^0_{ASM} be the initial state of a host and $c^0_{\text{HV}} = c^{init}_{\text{HV}}$ be the initial state of a BHV machine. Then for any sequence of the BHV machine $c^0_{\text{HV}}, c^1_{\text{HV}}, \ldots$, there exists a host machine sequence: $c^0_{ASM}, c^1_{ASM}, \ldots$ and a mixed machine sequence $c^0_{MX}, c^1_{MX}, \ldots$ that does not get stuck and does not produce a stack overflow. Moreover, for any number steps i of the BHV machine there exists a corresponding mixed machine state $c^{s(i)}_{MX}$ and a host machine state $c^{t(i)}_{ASM}$, such that the $\mathcal{B}$ relation and the implementation invariants hold. However, the $\mathcal{B}$ relation and the implementation invariants holds if the following preconditions on the BHV program π, the mixed machine c_{MX}, and the host machine c_{ASM} hold:

- the μ_{ASM} procedure "*_main*" is declared in the BHV program π and it resides at the program base address $info_{MX}.code_{ba}$ in memory. Formally,
$$info_{MX}.code_{ba} = info_{MX}.fun_{ba}(\text{"}\_main\text{"})$$
- the BHV handler *C-IL* function "*hv_disptach*" is declared as external in the μ_{ASM} procedure table and as non-external in the *C-IL* function table of the program π. Formally,
$$\pi.\pi_{\mu_{\text{ASM}}}(\text{"}hv\_disptach\text{"}).body = \textbf{extern}$$
and
$$\pi.\pi_{C\text{-}IL}.\mathcal{F}(\text{"}hv\_disptach\text{"}).\mathcal{P} \neq \textbf{extern}$$
- the number of input parameters to the BHV handler is 2 and they are of integer types. Since parameters passed at external calls in the *MX* semantics can only be of integer types we do not state the second condition.
$$\pi.\pi_{C\text{-}IL}.npar = 2$$

- The following constraints are put on the BHV parameters:

$$
\begin{aligned}
& \texttt{NG} \leq 8 \\
\wedge\ & \texttt{DS\_START} + \texttt{DS\_SIZE} < 2^{32} - 1 \\
2\wedge\ & \texttt{DS\_SIZE} \geq \mathit{size}_{\theta}(\texttt{hv\_t}) + 4096 + 4096 + \\
& \qquad \texttt{NG} \cdot (\mathit{size}_{\theta}(\texttt{guest\_t}) + \\
& \qquad\quad ((\texttt{MAX\_PPX} + 1) \cdot 4 + 4096 + (\texttt{MAX\_VPX} + 1) \cdot 4 + \\
& \qquad\quad 4096 + (\texttt{MAX\_PPX} + 1) \cdot \texttt{PAGE\_SIZE} \cdot 4 + 4096))
\end{aligned}
$$

  Note the data segment must be big enough to contain the BHV data structures as well as the stack for the BHV program (cf. Fig. 6.1).

- a boot code $\mathit{code}_{boot} \in \mathit{Instr}^*$ must be residing in the first memory page of the host machine c_{ASM}.

$$\forall\ i < |\mathit{code}_{boot}|.\ \mathit{int\text{-}to\text{-}instr}(c_{ASM}.m_{word}(i \cdot 4)) = \mathit{code}_{boot}[i]$$

  Here we do not present how the boot code is exactly implemented but we describe how it must be implemented for the proof.

Formally, this theorem is stated as follows:

$$
\begin{aligned}
& \exists\ \mathit{info}_{MX}.\ \forall\ (c^i_{ASM})_i, \exists (c^i_{MX})_i, \exists (c^i_{\mathrm{HV}})_i. \\
& \quad (\forall\ i.\quad c^{i+1}_{ASM} = \delta_{\mathrm{ASM}}(c^i_{ASM}) \\
& \qquad \wedge\quad \pi, \theta \vdash c^i_{MX} \rightarrow_{MX} c^{i+1}_{MX} \\
& \qquad \wedge\quad c^{i+1}_{\mathrm{HV}} = \delta_{\mathrm{HV}}(c^i_{\mathrm{HV}}, \texttt{NG}) \\
& \qquad \wedge\quad \text{the execution of } c_{MX} \text{ does not get stuck} \\
& \qquad \wedge\quad \text{there is no stack overflow}) \\
\wedge\quad & \exists\ s :: \mathbb{N} \mapsto \mathbb{N}.\ \exists\ t :: \mathbb{N} \mapsto \mathbb{N}. \\
& \quad (\forall\ i.\ (\mathcal{B}(c^i_{\mathrm{HV}}, c^{s(i)}_{MX}, c^{t(i)}_{ASM}, \texttt{NG})) \\
& \qquad \wedge\ \mathit{impl\text{-}inv}?(c^i_{\mathrm{HV}}, c^{s(i)}_{MX}, \mathit{info}_{MX}, \mathit{code}_{boot}, c^{t(i)}_{ASM}, \texttt{NG}))
\end{aligned}
$$

Proof: We split the proof according to various types of steps which can take place in the BHV model: guest and hypervisor steps which are performed in host user- and -system modes, respectively (cf. Fig. 7.5).

We prove the theorem by induction on the number BHV steps i:

- Base case ($i = 0$). The BHV configuration c^0_{HV} is in the init state (from preconditions to the theorem). The corresponding to the *MX* machine state is at the point when the first guest loaded on the host starts to run on the host.[5] For that, we need to argue about all steps done by the host machine and the mixed machine before this point. The proof of this case will be presented later on.

- Induction step ($i \longrightarrow i + 1$). There are two possible steps that can take place at $(i+1)$-th step of the BHV machine: guest step or hypercall. Hypercalls are invoked through trap interrupts during guest steps. Right now, in the BHV model a single hypercall is supported that schedules the next guest to run. To prove the induction step we need to show that the $(i+1)$-th step made by the

[5] the execution of the whole system starts from the beginning after the reset signal happens

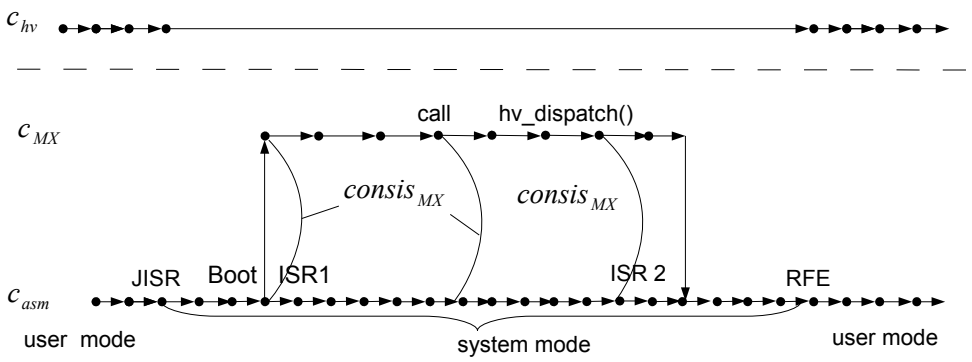


Figure 6.2: Guest and BHV Steps

BHV machine is consistent with the *MX* machine and the host after $s(i+1)$ and $t(i+1)$ steps, respectively.

6.3.6 Reset Case

After the reset signal has occurred the host machine c^0_{ASM} starts the execution from address 0. We know that the boot code resides in memory starting from address 0. So the host machine starts the execution with the boot code. After executing boot code for the reset case the host jumps into the middle of the first part of the BHV interrupt service routine that sets up the stack pointers. Next the BHV handler is invoked to handle reset interrupt. After handling the reset interrupt the registers of the scheduled guest gets loaded on the host. After all the return from exception instruction is executed by the host machine and the guest loaded before starts its execution. Figures 6.3 and 6.6 depict the aforementioned execution scenario[6]:

- c^0_{ASM} is the host machine state after reset signal, but before execution of any instructions from the boot code,
- c^1_{ASM} is the host machine state after executing the boot code,
- c^2_{ASM} is the host machine state after setting up stack pointers and c^2_{MX} is a *MX* machine state that is consistent with the host machine state c^2_{ASM}.
- c^3_{MX} is the *MX* machine state after passing input parameters to the BHV handler, but before the call.
- c^4_{MX} is the *MX* machine state after the reset interrupt was handled by the BHV handler, and

[6]indexed machine states do not correspond to indexed machine states from the theorem

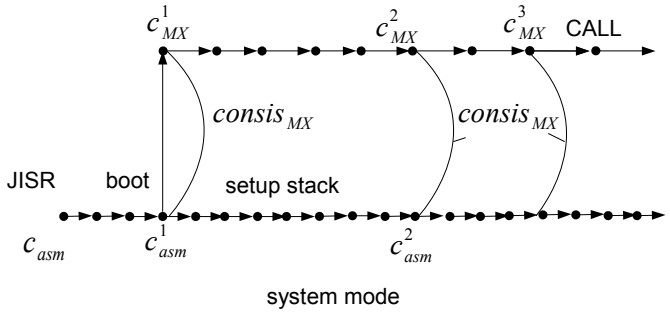


Figure 6.3: Boot + ISR1 (Reset)

- c^5_{ASM} is the host machine state after loading the scheduled guest on the host and executing the instruction `rfe`.

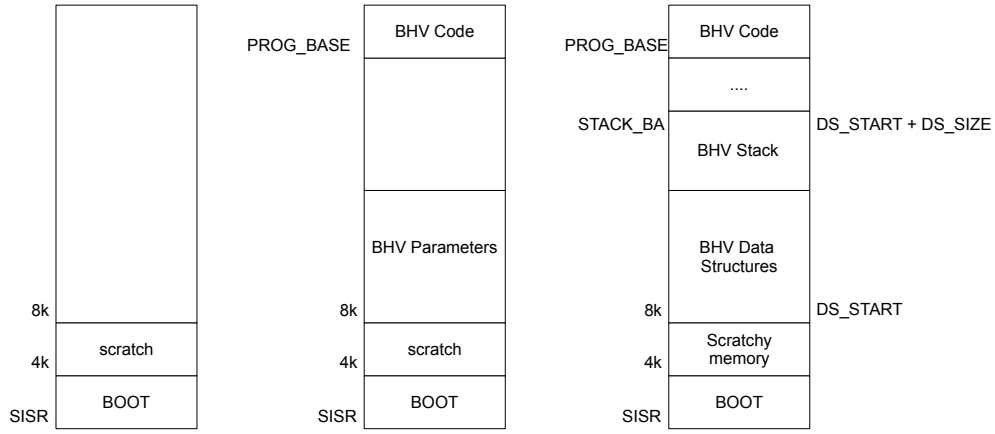
From preconditions we know that initially the memory of the host machine look likes as it is depicted in Fig. 6.4a.

Boot

First instructions of the boot code perform the test on reset interrupt. The test is done by reading the value of the special cause register. In case of reset the boot code stores the compiled BHV code at address PROG_BASE_ADR. Also it saves BHV parameters which BHV will use to initialize its data structures. As a result of the boot code execution for the reset case we have: (i) the compiled code of the BHV program π resides at the base address PROG_BASE_ADR, (ii) the BHV parameters are encoded in the first word of the data segment in memory (cf. Figure 6.5), (iii) the boot code computes the stack base address and saves it in the second word of the scratch memory (second physical memory page), (iii) the memory region which is reserved for BHV data structures including the stack region do not intersect with the memory region where the compiled BHV code resides. After executing the boot code the host machine gets updated and we denote the state of the update host machine as c^1_{ASM} and the memory of the update host machine looks like as it is depicted in Fig. 6.4b. Formally, Lemma 6.22 states what was updated in the host machine.

◄ LEMMA 6.22
After Booting
Reset Case

$$
\begin{aligned}
& \mathit{system?}(c^1_{ASM}.\mathit{spr}[\mathit{mode}]) \\
& \mathit{code\text{-}inv?}(\mathit{info}_{MX}, \mathit{code}_{boot}, c^1_{ASM}) \\
\wedge\ & \langle c^1_{ASM}.m(\texttt{DS\_START})\rangle_8 = \texttt{NG} \\
\wedge\ & \langle c^1_{ASM}.m(\texttt{DS\_START}+1)\rangle_8 = \texttt{MAX\_PPX}/2^{12} \\
\wedge\ & \langle c^1_{ASM}.m(\texttt{DS\_START}+2)\rangle_8 = \texttt{MAX\_VPX}/2^{12} \\
\wedge\ & \langle c^1_{ASM}.m(\texttt{DS\_START}+3)\rangle_8 = \texttt{DS\_SIZE}/2^{24} \\
\wedge\ & \langle c^1_{ASM}.m_{\text{word}}(4100)\rangle_{32} = \texttt{DS\_SIZE} + \texttt{DS\_START} \\
\wedge\ & [\texttt{DS\_START}, .., \texttt{DS\_START} + \texttt{DS\_SIZE}] \cap \\
& [\mathit{info}_{MX}.\mathit{code}_{ba}, .., \mathit{info}_{MX}.\mathit{code}_{ba} + 4 \cdot |\mathit{info}_{MX}.\mathit{code}|) = \varnothing
\end{aligned}
$$



(a) Before Boot (b) After Boot (c) Initialized BHV

Figure 6.4: Memory Snapshots

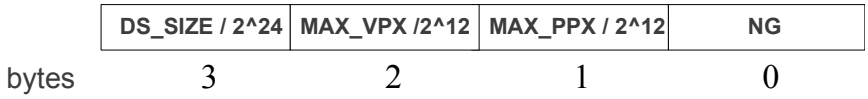


Figure 6.5: BHV Parameters Encoded in a Single Word

ISR1: Set Up Stack Pointers

After the boot code performed the initialization it jumps to the compiled BHV program by skipping a few instructions (lines 1-13 in Listing 6.4) which are not relevant for handling reset interrupt. Moreover their execution can destroy the code invariant as these instructions writes to non-existing PCB in memory and we cannot predict what will be updated without providing additional instructions (i.e. extending the implementation with new instructions). Note that PCBs of guests are allocated and initialized after the handling the reset interrupt by the BHV handler that was not called yet.

After jumping to the interrupt service routine we can easily construct the *MX* machine c^1_{MX} (without stack) corresponding to (consistent with) the host machine c^1_{ASM}.

$$\begin{aligned} c^1_{MX}.\mathcal{M} &= c^1_{ASM}.m \\ c^1_{MX}.ac.gpr &= c^1_{ASM}.gpr \\ c^1_{MX}.spr &= c^1_{ASM}.spr \\ c^1_{MX}.stack &= (\text{"}\_main\text{"}, 39), \\ c^1_{MX}.sc &= [] \end{aligned}$$

The procedure name and location counter of no-stack of the newly constructed *MX* machine is the μ_{ASM} procedure "*_main*" of the BHV program π and 39, respectively. The implementation of the μ_{ASM} procedure "*_main*" is listed in Listing 6.4. The

location 39 in this procedure corresponds to the instruction at line 15 in Listing 6.4.[7]

Before we can proceed further we need to show that the *MX* configuration c^1_{MX} and the host machine c^1_{ASM} are consistent, i.e. $consis_{MX}(c^1_{MX}, info_{MX}, c^1_{ASM})$ holds. We go through all sub-consistencies of the *MX* consistency and show them:

- The program counter consistency follows from the assumption over the static info $info_{MX}$, the construction of the *MX* configuration c^1_{MX} and the content of the program counters of the host machine c^1_{ASM}.
- The return address consistency holds by the construction of the *MX* configuration c^2_{MX} (as the stack is not set up in the configuration c^1_{MX}).
- The register consistency: By the construction of the *MX* configuration c^1_{MX} all GPRs and SPRs of the *MX* configuration c^1_{MX} and of the host machine c^1_{ASM} are the same. Thus this consistency holds, too.
- The code consistency follows from the code invariant (lemma 6.22).
- The stack consistency holds as the stack is not set up in the configuration c^1_{MX}.
- The memory consistency holds as memories of c^1_{MX} and c^1_{ASM} are identical (by the construction of c^1_{MX}).

After jumping from the boot code to the interrupt service routine and constructing the *MX* machine the following lemma must hold:

◀ LEMMA 6.23
Constructed *MX* Machine

$$\begin{aligned} & consis_{MX}(c^1_{MX}, info_{MX}, c^1_{ASM}) \\ \wedge\ & \neg is\_stack(c^1_{MX}) \\ \wedge\ & \textit{lemma 6.22 for } c^1_{ASM} \end{aligned}$$

So after executing the code (lines 16-23 in Listing 6.4) that sets up stack pointer registers of the *MX* machine and frame header in memory of this machine the following lemma must hold

◀ LEMMA 6.24
After Setting Up Stack Pointers

$$\begin{aligned} & c^2_{ASM}.gpr[bp] = \texttt{DS\_START} + \texttt{DS\_SIZE} \\ \wedge\ & c^2_{ASM}.gpr[sp] = \texttt{DS\_START} + \texttt{DS\_SIZE} \\ \wedge\ & consis_{MX}(c^2_{MX}, info_{MX}, c^2_{ASM}) \\ \wedge\ & is\_stack(c^2_{MX}) \\ \wedge\ & \textit{lemma 6.22 for } c^2_{ASM} \end{aligned}$$

ISR1: Pass Parameters

Further we continue the execution of instructions in lines 26-27 in Listing 6.4. These instructions save the cause and exception data registers in the input registers i_0 and i_1 , respectively, of the *MX* machine and we get an update machine c^3_{MX}. Here we followed the rule *CCL*.1 before calling a function that will be handling the reset interrupt. After passing parameters the lemma 6.25 must hold.

◀ LEMMA 6.25
Input Parameters Are Passed

$$\begin{aligned} & c^3_{MX}.ac.gpr[i_1] = c^2_{MX}.spr[eca] \\ \wedge\ & c^3_{MX}.ac.gpr[i_0] = c^2_{MX}.spr[edata] \\ \wedge\ & \textit{lemma 6.24 for } c^3_{MX} \textit{ and } c^3_{ASM} \end{aligned}$$

[7] There are 27 instructions which are represented as dots at line 8 in Listing 6.4.

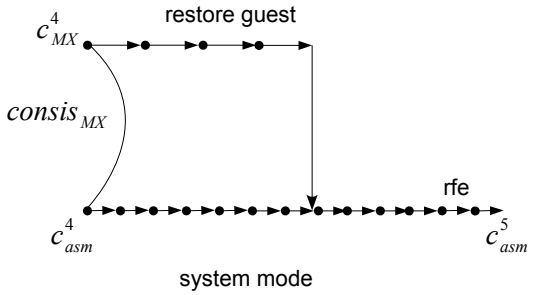


Figure 6.6: ISR2

ISR1: Call BHV Handler

The next instruction to be executed (line 29 in Listing 6.4) by c^3_{MX} is a call instruction. The external *C-IL* function $hv\_dispatch$ gets called from the μ_{ASM} context of the *MX* machine c^3_{MX}. In order to perform such a call to the function it is enough to have this function declared as external in the μ_{ASM} procedure table and non-external in the *C-IL* function table of the BHV program π. At this place we can see what we gain from having the *MX* semantics: no additional argumentions and proves are required to perform this external call. During the execution of the called function $hv\_dispatch$ the host is still operating in system mode, the code invariant holds, and the *MX* consistency holds at IO points.

ISR2: After Handling the Reset Interrupt

After handling the reset interrupt by the BHV handler the following is done: (i) the BHV data structures are allocated, (ii) guest data structures including PCBs, host PTEs and shadow PTEs are initialized, (iii) the first guest is scheduled to run on the host. Lemma 6.26 states what must hold after returning from the BHV handler.

LEMMA 6.26 ►
After Handling Reset by BHV C Function

$$
\begin{aligned}
& system?(c^4_{ASM}.spr[mode]) \\
\wedge\ & code\text{-}inv?(info_{MX}, code_{boot}, c^4_{ASM}) \\
\wedge\ & consis_{MX}(c^4_{MX}, info_{MX}, c^4_{ASM}) \\
\wedge\ & \mathcal{B}(c^{init}_{\text{HV}}.gs, c^4_{MX}, c^4_{ASM}, \texttt{NG}) \\
\wedge\ & pt\text{-}inv(c^4_{MX}, c^4_{ASM}, \texttt{NG}) \\
\wedge\ & \text{conditions } c_1, c_2 \text{ and } c_5 \text{ of } guest\text{-}inv(c^{init}_{\text{HV}}.cg, c^4_{MX}, c^4_{ASM}) \\
\wedge\ & is\_stack(c^4_{MX}) \\
\wedge\ & |c^4_{MX}.ac.s| = 1 \\
\wedge\ & c^4_{MX}.ac.s[0].lifo = []
\end{aligned}
$$

ISR2: Load Guest and RFE (Common to Reset and Non-Reset)

Next, a guest, scheduled to be run on the host, gets loaded on the host, i.e. the guest context from the corresponding PCB gets restored on the host machine. It is performed by instructions in lines 31-59 in Listing 6.4. After loading the guest context from the corresponding PCB residing in memory of the *MX* machine c^4_{MX} on the host the stack pointers of the host machine gets overwritten, i.e. the stack of the *MX* machine gets destroyed. But it does not prevent to continue the execution in the μ_{ASM} semantics since the stack of the *MX* machine was empty before the stack pointers were overwritten (by Lemma 6.26 and the μ_{ASM} semantics).

The next instruction to be executed by the host is a return from exception instruction that switches the host execution mode to user and restores exception registers to normal ones. From [AHPP10] we know that $c^{init}_{\text{HV}}.cg = 0$. After restoring the guest on the host and performing the return from the exception (cf. Figure 6.6) Lemma 6.27 must hold.

◄ LEMMA 6.27
After RFE

$$
\begin{aligned}
& user?(c^5_{ASM}.spr[mode]) \\
\wedge\ & code\text{-}inv?(info_{MX}, code_{boot}, c^5_{ASM}) \\
\wedge\ & \mathcal{B}(c^{init}_{\text{HV}}.gs, c^5_{MX}, c^5_{ASM}, \texttt{NG}) \\
\wedge\ & pt\text{-}inv(c^5_{MX}, c^5_{ASM}, \texttt{NG}) \\
\wedge\ & guest\text{-}inv(c^{init}_{\text{HV}}.cg, c^5_{MX}, c^5_{ASM}) \\
\wedge\ & \neg is\_stack(c^5_{MX})
\end{aligned}
$$

The reset case was completely considered from the reset signal until the scheduled guest starts to run on the host. At this point the proof of the BHV program for the reset case is completed and with that we have proven the base case of the theorem.

6.3.7 Induction Step

The lemma stated below is the statement that we need to prove for the induction step of the main theorem.

◄ LEMMA 6.28
Statement for Induction Step (Without Preconditions)

$$
\begin{aligned}
& \mathcal{B}(c^{i+1}_{\text{HV}}), c^{s(i+1)}_{MX}, c^{t(i+1)}_{ASM}, \texttt{NG}) \\
\wedge\ & impl\text{-}inv?(c^{i+1}_{\text{HV}}, c^{s(i+1)}_{MX}, info_{MX}, code_{boot}, c^{t(i+1)}_{ASM}, \texttt{NG}))
\end{aligned}
$$

We conclude this statement from the main theorem holding for i and all preconditions to the main theorem. Here we make case distinctions on what will be executed by the active guest $c^i_{\text{HV}}.cg$ of the BHV machine state c^i_{HV}:

- it invokes a hypercall. This case is indicated by the followings:
  - the active guest runs in system mode:
    $$system?(c_{\text{HV}}.gs(c_{\text{HV}}.cg).spr[mode])$$
  - the instruction to be executed by the guest $c^i_{\text{HV}}.gs(c^i_{\text{HV}}.cg)$ is a trap instruction.

  In this case we need to show that the BHV implementation saves the context of the currently running guest $c^i_{\text{HV}}.cg$ and schedules the next guest to run ($c^{i+1}_{\text{HV}} = c^i_{\text{HV}}.cg + 1$ **mod** NG). The detailed proof for this case will be presented later on.

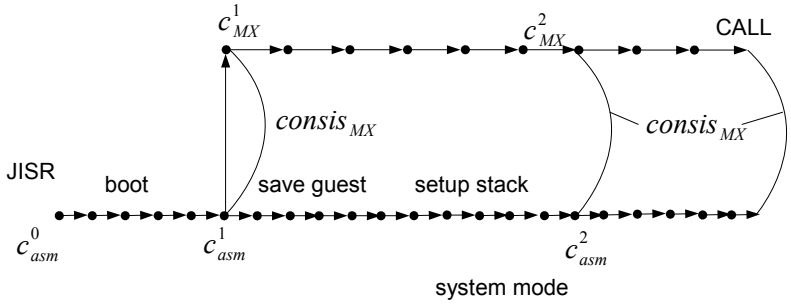


Figure 6.7: Boot + ISR1 (Non Reset)

- it executes an instruction (that is not a hypercall). Since the guest step emulation is not visible in the BHV model, but it is visible in the BHV implementation. We consider the effect of the instruction execution on the host machine $c^{t(i)}_{ASM}$. The host machine $c^{t(i)}_{ASM}$ will be executing the same instruction as the active guest of the BHV machine c^i_{HV} (by the induction hypothesis). If the execution of this instruction by $c^{t(i)}_{ASM}$ does not cause any interrupt, then we can easily conclude that Lemma 6.28 holds with $s(i+1) = s(i)$ and $t(i+1) = t(i)+1$ from the following:

  - Any try to change SPRs of the host will result into an interrupt as the host machine $c^{t(i)}_{ASM}$ operates in user mode (by the BHV implementation and the VAMP assembly semantics).
  - The data structures of the BHV implementation, the compiled code of the BHV program π with the boot code residing in memory cannot be corrupted as the guest running on the host works in its own memory space. That is, it is shown by the address manager invariants.
  - The HPTEs and SPTEs are protected with the corresponding bit for each valid PTE. Thus, any try to change them causes a page-fault.

  If the execution of the instruction by the host machine $c^{t(i)}_{ASM}$ causes an interrupt, then we need to show that the BHV implementation correctly emulates (or injects) the caused interrupt so that Lemma 6.28 holds.

In the following we present the proof for the case when the execution of the instruction causes an interrupt. Here we also treat the case with a hypercall as the only way to invoke a hypercall is possible with the execution of a trap instruction which causes a trap interrupt. Recall the VAMP assembly semantics from Chapter 2 that JISR is performed when an interrupt happens.

JISR

After an interrupt occurred the host machine performs JISR and we denote the new host state by c^0_{ASM} (cf. Figure 6.7). The host starts to operate in system mode, before

it was operating in user mode. Since we have proven the correctness of the BHV program for the reset case we assume here that the happened interrupt is not a reset interrupt. After JISR we have almost the same invariants as in guest steps, except for the $\mathcal{B}$-relation and the host execution mode. In case of a non-continue interrupt the exception program counters of host machine point to the instruction caused this interrupt. In this case we still have $\mathcal{B}$-relation holding with undone effect of JISR on the host machine.

That is,

$$\mathcal{B}(c^0_{\mathrm{HV}}.gs[c^0_{\mathrm{HV}}.cg := jisr^{-1}(c^0_{\mathrm{HV}}.gs(c^0_{\mathrm{HV}}.cg))], c^0_{MX}, jisr^{-1}(c^0_{ASM}), \mathtt{NG})$$

where the function $jisr^{-1}$ is undoing the effect of JISR over a given VAMP assembly configuration c:

$$jisr^{-1}(c) \stackrel{\text{def}}{=} c\left[\begin{array}{lcl} pc & := & c.spr[epc], \\ dpc & := & c.spr[edpc], \\ spr & := & \left[\begin{array}{lcl} mode & := & c.spr[emode], \\ sr & := & c.spr[esr] \end{array}\right] \end{array}\right]$$

In case of a continue interrupt after performing JISR the exception program counters of the host machine point to an instruction following the instruction caused this interrupt. Formally,

$$\mathcal{B}(c^0_{\mathrm{HV}}.gs[c^0_{\mathrm{HV}}.cg := g'], c^0_{MX}, jisr^{-1}(c^0_{ASM}), \mathtt{NG})$$

where g' is the active guest with the increased program counters after the effect JISR was undone. We define g' with $g = jisr^{-1}(c^0_{\mathrm{HV}}.gs(c^0_{\mathrm{HV}}.cg))$ as

$$g' = g\left[\begin{array}{lcl} dpc & := & g.pc, \\ pc & := & g.pc + 4 \end{array}\right]$$

After JISR was performed by the host Lemma 6.29 must hold:

◀ LEMMA 6.29
After JISR

$$\begin{array}{ll} & system?(c^0_{ASM}.spr[mode]) \\ \wedge & code\text{-}inv?(info_{MX}, code_{boot}, c^0_{ASM}) \\ \wedge & non\text{-}continue\ interrupt \longrightarrow \\ & \qquad \mathcal{B}(c^0_{\mathrm{HV}}.gs[c^0_{\mathrm{HV}}.cg := jisr^{-1}(c^0_{\mathrm{HV}}.gs(c^0_{\mathrm{HV}}.cg))], c^0_{MX}, jisr^{-1}(c^0_{ASM}), \mathtt{NG}) \\ \wedge & continue\ interrupt \longrightarrow \\ & \qquad \mathcal{B}(c^0_{\mathrm{HV}}.gs[c^0_{\mathrm{HV}}.cg := g'], c^0_{MX}, jisr^{-1}(c^0_{ASM}), \mathtt{NG}) \\ \wedge & pt\text{-}inv(c_{MX}, c^0_{ASM}) \\ \wedge & guest\text{-}inv(c_{\mathrm{HV}}.cg, c^0_{MX}, c^0_{ASM}) \end{array}$$

Boot

Next, the boot code performs the check on reset interrupt as it was done in the reset case. Since it is assumed that a non-reset interrupt happened the boot code jumps to the beginning of the interrupt service routine of the BHV program. The host machine c^1_{ASM} corresponds to point at which the boot code was executed for the non-reset case (cf. Figure 6.3). Lemma 6.30 states what must hold after executing the boot code for a non-reset interrupt.

◀ LEMMA 6.30
After Booting
Non-Reset Interrupt

lemma 6.29 for c^1_{ASM}

ISR1: Save Guest and Set Up Stack Pointers

After jumping to the interrupt service routine of the BHV program we construct the *MX* machine corresponding to the host machine state c^1_{MX}.

The execution of the *MX* machine proceeds with instructions from Listing 6.4 (lines 1-14). These instructions save the guest context from the host machine into the PCB corresponding to the active guest so that we do not change any other guests' PCB as well as the other components of the BHV components. Thus, the address manager invariants are not destroyed because HPTEs and SPTEs of all guests are not changed.

The next instructions to be executed set up the stack pointers. So after saving the guest context in the corresponding PCB in memory and setting up the stack pointers (cf. Figure 6.3) the following lemma must hold

LEMMA 6.31 ►
Save Guests and Set Up Stack Pointers

$$
\begin{aligned}
& \mathit{system?}(c^2_{ASM}.\mathit{spr}[\mathit{mode}]) \\
\wedge\ & \mathit{code\text{-}inv?}(\mathit{info}_{MX}, \mathit{code}_{\mathit{boot}}, c^2_{ASM}) \\
\wedge\ & \mathit{consis}_{MX}(c^2_{MX}, \mathit{info}_{MX}, c^2_{ASM}) \\
\wedge\ & \mathit{non\text{-}continute\ interrupt} \longrightarrow \\
& \qquad \mathcal{B}(c^0_{\mathrm{HV}}.gs[c^0_{\mathrm{HV}}.cg := \mathit{jisr}^{-1}(c^0_{\mathrm{HV}}.gs(c^0_{\mathrm{HV}}.cg))], c^2_{MX}, c^2_{ASM}, \mathtt{NG}) \\
\wedge\ & \mathit{continue\ interrupt} \longrightarrow \mathcal{B}(c^0_{\mathrm{HV}}.gs[c^0_{\mathrm{HV}}.cg := g'], c^2_{MX}, c^2_{ASM}, \mathtt{NG}) \\
\wedge\ & \mathit{pt\text{-}inv}(c^2_{MX}, c^2_{ASM}, \mathtt{NG}) \\
\wedge\ & \mathit{guest\text{-}inv}(c_{\mathrm{HV}}.cg, c^2_{MX}, c^2_{ASM})
\end{aligned}
$$

ISR1: Pass Parameters and Call BHV Handler

The next step to be performed is to pass exception data and cause registers in the corresponding input registers and only then the BHV handler is called. We have already seen these steps when we have treated the reset case.

ISR2: After Handling Non-Reset Interrupt

After handling a non-reset interrupt by the BHV handler the following lemma must hold

LEMMA 6.32 ►
After Handling Interrupt by BHV C Function

$$
\begin{aligned}
& \mathit{system?}(c^3_{ASM}.\mathit{spr}[\mathit{mode}]) \\
\wedge\ & \mathit{code\text{-}inv?}(\mathit{info}_{MX}, \mathit{code}_{\mathit{boot}}, c^3_{ASM}) \\
\wedge\ & \mathit{consis}_{MX}(c^3_{MX}, \mathit{info}_{MX}, c^3_{ASM}) \\
\wedge\ & \mathcal{B}(c^{\mathit{init}}_{\mathrm{HV}}.gs, c^3_{MX}, c^3_{ASM}, \mathtt{NG}) \\
\wedge\ & \mathit{pt\text{-}inv}(c^3_{MX}, c^3_{ASM}, \mathtt{NG})
\end{aligned}
$$

ISR2: Restore Guest and RFE

We have already treated instructions that restores the guest context on the host in the proof of the base case of the theorem.

■

CHAPTER 7

Assembly Verification Approach

S.Schmaltz and me initially started to work on the soundness of an assembly verification approach like presented in [MMS08] two years ago. By this time there were no semantics for languages like μ_{ASM} and *MX* and there was not so much progress done on this topic except for ideas that motivated S.Schmaltz and me to invent the aforementioned semantics. The rest of this chapter is considered as a contribution to the thesis.

Here we present our translation-based approach for verifying programs, written in a low-level C programming language (like *C-IL*) and high-level assembly (like μ_{ASM}), with a general purpose C verifier (like VCC[1]). Using this approach we can verify μ_{ASM} code portions together with C code portions using VCC, where C functions call μ_{ASM} procedures and vice versa. This approach is not about how to verify C code, but it is about how to create an environment in which the correctness of the μ_{ASM} assembly portions can be shown and then can be used to show the correctness of a complete mixed program without any additional effort.

During the work on this topic two meaningful approaches were pointed out to verify μ_{ASM} portions of mixed programs. The first approach based on a hardware simulator implemented in C. This simulator should completely correspond to the hardware we want to operate on. Note that the hardware simulator uses the same C memory as the C portions of mixed programs. The assembly verification itself concludes in the execution of the hardware simulator (golden model) implemented in C running over the assembly code located in C memory. That is, no translation of assembly instructions to C statements is required. The advantage of this approach is the verification of self-modifying code. But the disadvantage is that the assembly verification becomes less transparent to a verification engineer. Definitely, such a disadvantage destroys what a verification engineer really wants to have, namely the verification transparency for searching bugs and stating top level correctness statements.

The second approach, called *simulation approach*, is based on the translation of

[1]VCC is an automatic verifier for low-level and concurrent C with contracts developed at Microsoft [Cor08]

(macro) assembly instructions into C statements which must simulate these instructions. The simulating statements operate on registers modeled in a so-called *hybrid memory*[2]. Note that the simulating statements operate on the hybrid memory as well as on the C memory. This approach was pioneered by Maus in [Mau11], but he didn't mention the hybrid memory which is one of important aspects to show the soundness of this approach. The advantage of this approach is easy understanding and verification of translated assembly code. A complicated soundness argument on this verification approach can be considered as disadvantage in general. Maus didn't provide any formal specification of this approach and therefore no correctness proof was done.

We used the simulation approach to verify assembly and macro-assembly portions of the simple hypervisor examined in Chapter 6. To justify the correctness of this approach we first specify how we translate a *MX* program to a *C-IL* program. Then the step-by-step simulation theorem is stated between *MX* programs and translated programs with a sketch of this theorem's proof.

7.1 Translating *MX* Programs To *C-IL*

Here we specify the translation of *MX* programs to *C-IL* programs which we will refer to as translated programs in the rest of this chapter. As a result of this translation any C verifier can be applied to verify these translated programs.

To have semantically equivalent *MX* and translated programs we need to model all registers and stack frame components that can be accessed during the execution of *MX* programs on *MX* machines. For that purpose we add a few global variables to translated *C-IL* programs. These global variables are allocated in a memory region that cannot be accessed by *MX* programs. Namely, this memory region must not be accessed by any instructions(statements) of *MX* programs. To guarantee this we assume that the working memory address space of *MX* programs is $[0, .., 2^{32})$ and the newly added global variables to translated programs are allocated in memory within the high memory address space (for instance it can be the region $[2^{32}, .., 2^{33})$). We call these global variables *hybrid global variables* and the memory where these hybrid global variables are allocated *hybrid memory*.

7.1.1 Modeling Registers and Stack Frame Components

As it was already mentioned we model registers and components of abstract μ_{ASM} stack frames of a *MX* configuration with global hybrid variables in translated programs. Obviously, there is no need to model components of *C-IL* stack frames in the translated programs.

We model GPRs and SPRs of a *MX* machine in a straightforward way by two hybrid global arrays gpr_{hyb} and spr_{hyb} of size 32, respectively, whose elements are of *C-IL* type $\mathbf{i}32$.

We model components of μ_{ASM} stack frames in the following way: there are three components $lifo$, $pars$ and $saved$ of abstract μ_{ASM} stack frames that we need to model in the translated programs. The other components like the location counter and the procedure name are modeled by the *C-IL* semantics itself. Each of three μ_{ASM} stack

[2]The notion of a *hybrid* memory was introduced in the Verisoft XT project. This memory is used to model some components like processor registers and stack which are not visible in a pure C code. It is a counterpart to XCalls introduced in the Verisoft project.

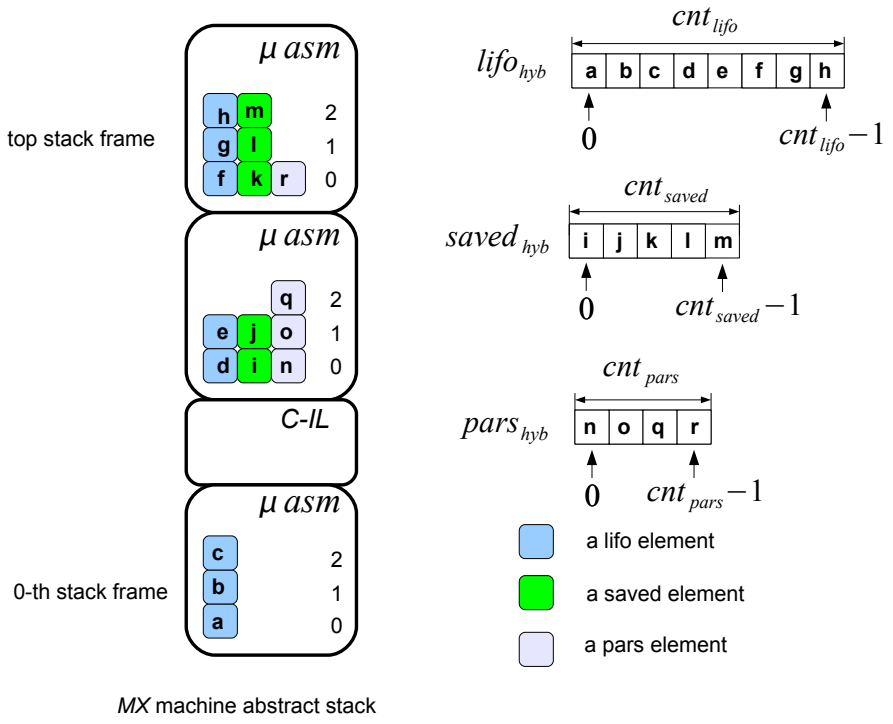


Figure 7.1: Representation μ_{ASM} Stack Components by Hybrid Variables

frame components that is needed to be modeled in the translated programs is represented by a pair of hybrid global variables representing a fixed size array and a counter containing the number of "meaningful" elements in this array.

- The component $lifo$ is modeled by hybrid global variables $lifo_{hyb}$ and cnt_{lifo}.
- The component $pars$ is modeled by hybrid global variables $pars_{hyb}$ and cnt_{pars}.
- The component $saved$ is modeled by hybrid global variables $saved_{hyb}$ and cnt_{saved}.

Figure 7.1 depicts an example of the abstract call stack of an *MX* machine and the content of these hybrid global variables. Note that the abstract call stack depicted in this figure grows from bottom to top as well as the components of μ_{ASM} stack frames.

7.1.2 Translation Function

Here we define the function

$$\tau_{MX2IL} :: Prog_{MX} \mapsto Prog_{IL}$$

that (i) first translates the implementation of all defined μ_{ASM} procedures in the μ_{ASM} procedure table $\pi.\pi_{\mu_{\text{ASM}}}$ of a given mixed program π into the corresponding (and hopefully) equivalent *C-IL* implementations, (ii) updates the bodies of the corresponding *C-IL* functions from the *C-IL* function $\pi.\pi_{C\text{-}IL}$ table with these new translated implementations, (iii) adds hybrid global variables discussed in the previous section to the

declaration list of global variables in the *C-IL* program $\pi.\pi_{C\text{-}IL}$, and (iv) returns the updated *C-IL* program.

We first compute a new *C-IL* function table $\mathcal{F}'$ which contains the translated μ_{ASM} procedures and the original *C-IL* functions from a given *MX* program π.

$$\mathcal{F}'(f) = \begin{cases} F & \text{if } \pi.\pi_{\mu_{\text{ASM}}}(f).body \neq \textbf{extern} \\ \pi.\pi_{C\text{-}IL}.\mathcal{F}(f) & \text{otherwise} \end{cases}$$

where F is a new *C-IL* function declaration obtained by the translation of a μ_{ASM} procedure f entity in the following way:

$$F = \left[\begin{array}{rcl} rettype & := & \textbf{void}, \\ npar & := & n, \\ & & \\ \mathcal{V} & := & \overbrace{(p_{n-1}, \textbf{i32}) \circ \ldots \circ (p_0, \textbf{i32})}^{n}, \\ \mathcal{P} & := & \tau_{\mu asm2c}(\pi.\pi_{\mu_{\text{ASM}}}(f).body, f, \pi.\pi_{\mu_{\text{ASM}}})) \end{array} \right]$$

where $n = \pi_{\mu_{\text{ASM}}}(f).npar$ is the number of input parameters to a μ_{ASM} procedure f and $\tau_{\mu asm2c}(il, f, \pi)$ is a function that translates a given list of μ_{ASM} instructions il that represents the body of a given μ_{ASM} procedure f from a given μ_{ASM} procedure table π to its *C-IL* equivalent representation. We will define this function in the next section.

The translation function τ_{MX2IL} is defined as follows:

$$\tau_{MX2IL}(\pi) \stackrel{\text{def}}{=} \pi.\pi_{C\text{-}IL}\left[\begin{array}{rcl} \mathcal{V}_G & := & vars_{hybrid} \circ \pi.\pi_{C\text{-}IL}.\mathcal{V}_G \\ \mathcal{F} & := & \mathcal{F}' \end{array} \right]$$

where a list $vars_{hybrid}$ of *C-IL* variable declarations contains the declaration of all hybrid global variables modeling μ_{ASM} stack components in *C-IL*:

$$\begin{array}{rcl} vars_{hybrid} & = & (gpr_{hyb}, \textbf{array}(\textbf{i}32, 32)) \circ (spr_{hyb}, \textbf{array}(\textbf{i}32, 32)) \circ \\ & & (cnt_{lifo}, \textbf{u}32) \circ (cnt_{pars}, \textbf{u}32) \circ (cnt_{saved}, \textbf{u}32) \circ \\ & & (lifo_{hyb}, \textbf{array}(\textbf{i}32, \texttt{LIFO\_SIZE})) \circ \\ & & (pars_{hyb}, \textbf{array}(\textbf{i}32, \texttt{PARS\_SIZE})) \circ \\ & & (saved_{hyb}, \textbf{array}(\textbf{i}32, \texttt{SAVED\_SIZE})) \end{array}$$

Note that the constants `LIFO_SIZE`, `PARS_SIZE`, `SAVED_SIZE` are parameters in our approach and they represent the maximum size of the corresponding hybrid array. These constants must be big enough to avoid a so-called buffer overflow that can happen at an array access. We will return to this issue in Section 7.2 where we will state the translation correctness.

The counters cnt_{lifo}, cnt_{pars} and cnt_{saved} must be initialized with zeros before a *C-IL* machine will start to run a translated program.

7.1.3 Translating μ_{ASM} Procedure Bodies To *C-IL*

Here we define the recursive function

$$\tau_{\mu asm2c} :: \mathbb{S}^*_{\mu_{\text{ASM}}} \times P_{name} \times Prog_{\mu_{\text{ASM}}} \to \mathbb{S}^*$$

which was already introduced in Section 7.1.2. This function produces a list of *C-IL* statements that must simulate a given list of μ_{ASM} instructions. Also, this function takes a procedure name and a μ_{ASM} procedure table as input.

This recursive function starts the translation from the last instruction[3] from a given list of μ_{ASM} instructions il and then makes recursion on the tail of the list il. A μ_{ASM}

[3] Recall that $\textbf{hd}(il) = il[|il| - 1]$

instruction is translated into a list of *C-IL* statements by means of the function $\tau_{\mu asmi2c}$. When all instructions from the list *il* were translated this function adds the prologue returned by the function $prologue(f, \pi)$. We define the functions $\tau_{\mu asmi2c}$ in the next section. So that, the function $\tau_{\mu asm2c}$ is defined in the following way:

$$\tau_{\mu asm2c}(il, f, \pi) \stackrel{\text{def}}{=} \begin{cases} \tau_{\mu asmi2c}(\mathbf{hd}(il), f, \pi) \circ \tau_{\mu asm2c}(\mathbf{tl}(il), f, \pi) & \text{if } il \neq [] \\ prologue(f, \pi) & \text{otherwise} \end{cases}$$

Translating a μ_{ASM} Instruction To *C-IL* Statements

A single μ_{ASM} instruction is translated into either a single *C-IL* statement or many *C-IL* statements which we call *simulating statements*. The simulating statements perform accesses to: (1) registers modeled by hybrid global arrays gpr_{hyb} and spr_{hyb}, (2) memory, and (3) stack components modeled by hybrid global arrays $lifo_{hyb}$, $pars_{hyb}$, and $saved_{hyb}$ and counters cnt_{lifo}, cnt_{pars} and cnt_{saved}.

The translation of a single μ_{ASM} instruction is performed by the following function

$$\tau_{\mu asmi2c} :: \mathbb{S}_{\mu_{\text{ASM}}} \times P_{name} \times Prog_{\mu_{\text{ASM}}} \to \mathbb{S}^*.$$

This function takes a μ_{ASM} instruction, a μ_{ASM} procedure name and a μ_{ASM} procedure table as input and produces a list of *C-IL* statements whose execution must simulate the execution of this μ_{ASM} instruction. We will talk about the equivalent execution of a μ_{ASM} instruction and simulating *C-IL* statements in Section 7.2.

In the following we define the function $\tau_{\mu asmi2c}$ by case split on μ_{ASM} instructions: (i) VAMP assembly instructions modeled in μ_{ASM}, (ii) goto instructions, (iii) push and pop instructions, (iv) load and store parameter instructions, and (v) a call to a procedure instruction, and (vi) a return from a call instruction. Note that the translation of each instruction must obey the semantics defined in Chapters 3 and 5.

VAMP Assembly Instructions We define the function $\tau_{\mu asmi2c}$ for two VAMP assembly instructions like `add` $rd\ rs_1\ rs_2$ and `sw` $rd\ rs_1\ imm$. The rest of VAMP assembly instructions modeled in μ_{ASM} can also be easily translated into the corresponding *C-IL* statements. Therefore, we omit their translations.

In Chapter 3 we defined the semantics when VAMP assembly instructions try to change the content of one of GPRs *bp* and *sp* (cf. Table 3.1). In this case the stack of a μ_{ASM} machine is either created or destroyed. We model this effect by setting the content of all hybrid counters to zeros and translating in translated programs.

The VAMP assembly instruction `add` $rd\ rs_1\ rs_2$ gets translated into a single *C-IL* statement that reads values stored at positions rs_1 and rs_2 in the hybrid array gpr_{hyb}, sums up these read values, and updates the hybrid array gpr_{hyb} at position *rd* with this computed sum. Here it is assumed that the destination register is neither GPR *bp* nor GPR *sp*.

$$\tau_{\mu asmi2c}(\texttt{add}\ rd\ rs_1\ rs_2, p, \pi) = \left[\ gpr_{hyb}[rd] = gpr_{hyb}[rs_1] + gpr_{hyb}[rs_2]]\ \right]$$

If an instruction is going to change the content of one of GPRs *bp* and *sp*, then the sequence of the simulating *C-IL* statements obtained by the translation of this VAMP assembly instruction is extended with three *C-IL* statements setting hybrid counters to zeros. For example, If the destination register of a VAMP assembly addition instruction is either GPR *bp* or GPR *sp*, then this instruction will be translated to the following sequence of *C-IL* statements:

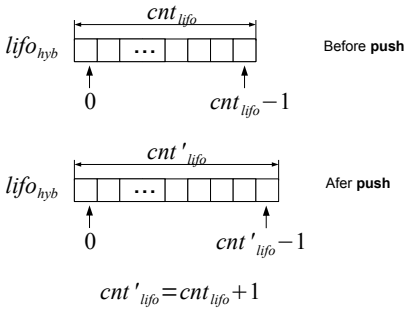


Figure 7.2: Simulating Push in *C-IL*

Listing 7.1: *C-IL* Statements simulating a VAMP assembly instruction updating a stack register

```
1 gpr_hyb[rd] = gpr_hyb[rs_1] + gpr_hyb[rs_2]
2 cnt_lifo = 0
3 cnt_pars = 0
4 cnt_saved = 0
```

The VAMP assembly instruction `sw` $rd\ rs_1\ imm$ gets translated into a single *C-IL* statement that updates the memory at address $gpr_{hyb}[rs_1] + imm$ with the value stored in $gpr_{hyb}[rd]$. Here it is assumed that $rd \notin \{bp, sp\}$.

$$\tau_{\mu asmi2c}(\texttt{sw}\ rd\ rs_1\ imm, p, \pi) = \left[\ *((\mathbf{ptr}(\mathbf{i}32))(gpr_{hyb}[rs_1] + imm)) = gpr_{hyb}[rd]\ \right]$$

Goto Instructions The μ_{ASM} goto instructions are translated to their equivalent *C-IL* statements with translated target locations. The translation of a target location l is done by means of the function $Location_{MX2IL}(l, p, \pi)$ which we will define later.

$$\tau_{\mu asmi2c}(\texttt{goto}\ l, p, \pi) = \left[\ \mathbf{goto}\ Location_{MX2IL}(l, p, \pi)\ \right]$$

$$\tau_{\mu asmi2c}(\texttt{ifnez}\ r\ \texttt{goto}\ l, p, \pi) = \left[\ \mathbf{ifnot}\ (gpr_{hyb}[r] == 0)\ \mathbf{goto}\ Location_{MX2IL}(l, p, \pi)\ \right]$$

Push and Pop Instructions The instruction `push` r is translated into two *C-IL* statements. The first *C-IL* statement updates the hybrid array $lifo_{hyb}$ at the position cnt_{lifo} and the second *C-IL* statement increments the hybrid lifo counter cnt_{lifo}. Figure 7.3 depicts the corresponding changes done by these *C-IL* statements and we list these *C-IL* statements below:

$$\tau_{\mu asmi2c}(\texttt{push}\ r, p, \pi) =$$

Listing 7.2: *C-IL* Statements Simulating the `push` r instruction

```
1 lifo_hyb[cnt_lifo] = gpr_hyb[r]
```

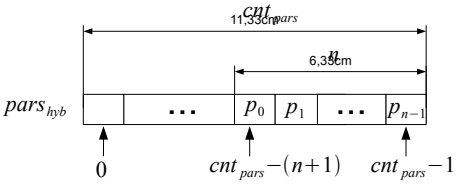


Figure 7.3: μ_{ASM} Parameters in *C-IL*

```
2  cnt_lifo = cnt_lifo + 1
```

The instruction `pop` r is translated into two *C-IL* statements that remove the last saved element in the hybrid array $lifo_{hyb}$, save it in the hybrid array of GPRs gpr_{hyb} at the position r, and decrement the hybrid lifo counter cnt_{lifo}.

$$\tau_{\mu asmi2c}(\texttt{pop}\ r, p, \pi) =$$

Listing 7.3: *C-IL* Statements Simulating the `pop` r instruction

```
1  cnt_lifo = cnt_lifo − 1
2  gpr_hyb[r] = lifo_hyb[cnt_lifo]
```

Load and Store Parameter Instructions The load parameter instruction `lparam` $r\ i$ is translated into a single *C-IL* statement that updates the hybrid array of GPRs gpr_{hyb} at the position r with the value stored in the hybrid array of parameters $pars_{hyb}$ at the position corresponding to the i-th parameter[4] of a given procedure p that is declared in a given μ_{ASM} procedure table π. This position of an element to be read in the array is computed as

$$cnt_{pars} - (n + 1) + i,$$

where $n = \pi_{\mu_{\text{ASM}}}(p).npar$ is the number of input parameters to a μ_{ASM} procedure p. Note that the hybrid array $pars_{hyb}$ at the position $cnt_{pars} - 1$ contains the value of the parameter with the highest index. Thus, the hybrid array $pars_{hyb}$ at the position $cnt_{pars} - (n + 1)$ is the parameter with index 0 (cf. Figure 7.3).

$$\tau_{\mu asmi2c}(\texttt{lparam}\ r\ i, p, \pi) = \left[\ gpr_{hyb}[r] = pars_{hyb}[cnt_{pars} - (n + 1) + i]\ \right]$$

The store parameter instruction `sparam` $r\ i$ is translated into a single *C-IL* statement that updates the hybrid array of parameters $pars_{hyb}$ at the position corresponding to the i-th parameter with the value stored in $gpr_{hyb}[r]$.

$$\tau_{\mu asmi2c}(\texttt{sparam}\ r\ i, p, \pi) = \left[\ pars_{hyb}[cnt_{pars} - (n + 1) + i] = gpr_{hyb}[r]\ \right]$$

Call Instruction We split the translation of the `CALL` f instruction into portions. The first portion is present on the caller side, and the second portion, called *prologue*, is present on the callee side. The prologue is added to the beginning of the *C-IL* function

[4]Recall that we index parameters from zero, i.e. $i \in [0, .., \pi(f).npar - 1]$

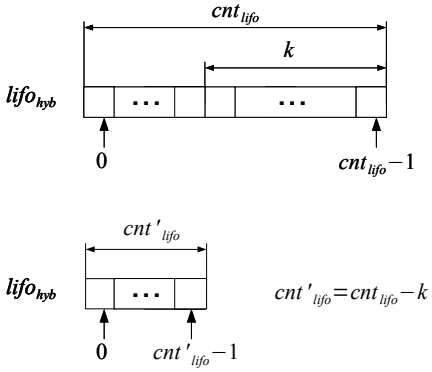


Figure 7.4: Decreasing the lifo counter

implementation obtained by the translation from a μ_{ASM} procedure f. You can see it in the definition of the function $\tau_{\mu asm2c}$ examined in Section 7.1.3.

On the caller side the `CALL` f instruction is translated into two *C-IL* statements. It does not matter whether this is either a call to a μ_{ASM} procedure or an inter-language call to a *C-IL* function in a *MX* program. Thus for both cases we have the same *C-IL* statements. The first statement decreases the lifo counter cnt_{lifo} by

$$k \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } 4 < \pi_{\mu_{\text{ASM}}}(f).npar \\ \pi_{\mu_{\text{ASM}}}(f).npar - 4 & \text{otherwise} \end{cases},$$

where k is the number of values passed through the hybrid lifo to a procedure f. The second *C-IL* statement performs the invocation of a procedure f. The place where input values to this call are stored is defined by the compiler calling convention *CCL* introduced in Chapter 3. That is, the first four input values are taken from the input registers and the rest is taken from the hybrid array $lifo_{hyb}$. Note that before decrementing the lifo counter the parameter with the highest index was stored in the hybrid array $lifo_{hyb}$ at the position $cnt_{lifo} - k + 1$ and the 4-th parameter was stored in the same hybrid array at the position $cnt_{lifo} - 1$ (cf. Figure 7.4)

$$\tau_{\mu asmi2c}(\texttt{CALL}\ f, p, \pi) =$$

Listing 7.4: *C-IL* Statements Simulating the `CALL` f instruction

```
1 cnt_lifo = cnt_lifo − k
2 call f(gpr_hyb[i_0], ..., gpr_hyb[i_3], lifo_hyb[cnt_lifo + k − 1], .., lifo_hyb[cnt_lifo])
```

On the callee side the instruction `CALL` f is translated to *C-IL* statements that represent a prologue of a procedure f. The execution effect of these *C-IL* statements is:

- values of hybrid GPRs whose indices are listed in the list $uses = \pi_{\mu_{\text{ASM}}}(f).uses$ are saved in the hybrid array $saved_{hyb}$ starting from the position identified by the saved counter cnt_{saved},

- the first four input parameters (if existent) of a given procedure f are saved in the hybrid GPRs in the input registers,
- the rest of input parameters (if existent) are saved in right-to-left order in the hybrid array of parameters $pars_{hyb}$ starting from the position $cnt_{pars} + 4$. Note that the positions $cnt_{pars}, .., cnt_{pars} + 3$ in the hybrid array of parameters are reserved for input values passed in input registers $gpr_{hyb}[i_0], \ldots, gpr_{hyb}[i_3]$.
- the hybrid counters cnt_{saved} and cnt_{pars} are increased by the number of elements in the list $uses$ and of input parameters to a procedure f, respectively.

$$prologue(f, \pi) =$$

Listing 7.5: *C-IL* Statements Simulating the effect of the `CALL` f instruction on the callee side

```
1  saved_hyb[cnt_saved] = gpr_hyb[uses_0]
2  ...
3  saved_hyb[cnt_saved + |uses| - 1] = gpr_hyb[uses_|uses|-1]
4  gpr_hyb[i_0] = p_0
5  ...
6  gpr_hyb[i_3] = p_3
7  pars_hyb[cnt_pars + 4] = p_4
8  ...
9  pars_hyb[cnt_pars + n - 1] = p_n-1
10 cnt_saved = cnt_saved + |uses|
11 cnt_pars = cnt_pars + n
```

Return Instruction The `ret` instruction is translated to *C-IL* statements that perform the following steps:

- it restores values of registers whose indices are listed in the list $uses = \pi_{\mu_{\text{ASM}}}(f).uses$ from the hybrid array $saved_{hyb}$,
- it decreases the hybrid parameter- and saved counters by the number of input parameters taken by a given procedure p and the number of elements in the list $uses$, respectively.
- it returns from a call.

$$\tau_{\mu asmi2c}(\texttt{ret}, p, \pi) =$$

Listing 7.6: *C-IL* Statements Simulating the "inter-language" `ret` instruction

```
1  gpr_hyb[uses_0] = saved_hyb[cnt_saved - |uses| + 1]
2  ...
3  gpr_hyb[uses_|uses|-1] = saved_hyb[cnt_saved - 1]
4  cnt_pars = cnt_pars - n
5  cnt_saved = cnt_saved - |uses|
6  return
```

Recomputing Locations

After translating *MX* programs to *C-IL* programs we would like to have a function that maps locations of instructions of μ_{ASM} procedures from *MX* programs to the corresponding locations of simulating statements of translated procedures from translated programs. This mapping is done by means of the function

$$Location_{MX2IL} :: \mathbb{N} \times P_{name} \times Prog_{\mu_{\text{ASM}}} \mapsto \mathbb{N}.$$

The definition of this function is based on the definition of functions

$$size_{MX2IL} :: \mathbb{S}_{\mu_{\text{ASM}}} \times P_{name} \times ProcT \to \mathbb{N},$$

and

$$size^{prologue}_{MX2IL} :: P_{name} \times Prog_{\mu_{\text{ASM}}} \mapsto \mathbb{N}$$

that return the number of *C-IL* statements needed to implement a given μ_{ASM} instruction and a prologue for a given procedure p from a given μ_{ASM} procedure table π, respectively. We first define the function $size_{MX2IL}$ by case split on μ_{ASM} instructions:

- $I \in Instr_{cut4\mu_{\text{ASM}}} \cup \{\texttt{goto}\ l, \texttt{ifnez}\ r\ \texttt{goto}\ l\}$

$$size_{MX2IL}(I, p, \pi) \stackrel{\text{def}}{=} 1$$

- $I \in \{\texttt{push}\ r, \texttt{pop}\ r, \texttt{sparam}\ r\ i, \texttt{lparam}\ r\ i, \texttt{CALL}\ f\}$

$$size_{MX2IL}(I, p, \pi) \stackrel{\text{def}}{=} 2$$

- I = `ret`

$$size_{MX2IL}(\texttt{ret}, p, \pi) \stackrel{\text{def}}{=} 3 + |uses|$$

The function $size^{prologue}_{MX2IL}$ is defined as follows

$$size^{prologue}_{MX2IL}(p, \pi) \stackrel{\text{def}}{=} 2 + |uses| + npar.$$

In the following we can easily define the function $Location_{MX2IL}$ as

$$Location_{MX2IL}(loc, p, \pi) \stackrel{\text{def}}{=} size^{prologue}_{MX2IL}(p, \pi) + \sum_{i<loc} size_{MX2IL}(\pi(f).body[i], p, \pi)$$

7.2 Simulation Theorem

In this section we present the simulation theorem which states the correctness of the translation function τ_{MX2IL} defined in the previous section. An important aspect of the simulation theorem is the simulation relation between states of the *MX* semantics and states of *C-IL* semantics. To guarantee the semantically equivalent execution of *MX* and *C-IL* machines running a *MX* program and its translated version, respectively, we define the simulation relation rel_{MX-IL} indicating that both machines are consistent. We also call this simulation relation *translater consistency*. Then we can prove the correctness of this theorem by a stepwise simulation between both semantics.

The translater consistency rel_{MX-IL} comprises the register consistency rel^{regs}_{MX-IL}, memory consistency rel^{mem}_{MX-IL}, stack consistency rel^{stack}_{MX-IL} and control consistency $rel^{control}_{MX-IL}$. The first three consistencies form a so-called data consistency. Note that the memory consistency does not cover the content of hybrid variables in a *C-IL* machine memory. But the register and stack consistencies cover their content.

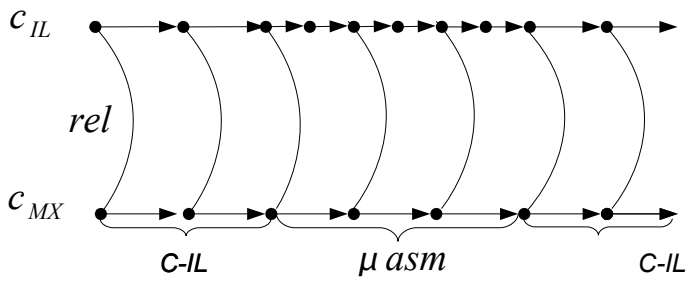


Figure 7.5: Simulation Relation *MX* To *C-IL*

$$\frac{\begin{array}{c} c_{MX}.stack_{top} \in frame_{\mu} \longrightarrow rel^{regs}_{MX-IL}(c_{MX}, c_{IL}) \\ rel^{stack}_{MX-IL}(c_{MX}, c_{IL}) \qquad rel^{mem}_{MX-IL}(c_{MX}, c_{IL}) \qquad rel^{control}_{MX-IL}(c_{MX}, \pi, c_{IL}) \end{array}}{rel_{MX-IL}(c_{MX}, c_{IL})}$$

If the top stack frame of a *MX* machine c_{MX} is of μ_{ASM} type, then the register consistency rel^{regs}_{MX-IL} states that the content of registers of the two given *MX*- and *C-IL* machines are the same. Recall that we model registers in the hybrid memory of a *C-IL* machine with arrays. We do not require equality between stack registers of the *MX* and *C-IL* machines in case the *MX* machine has a stack.

$$\frac{\begin{array}{c} c_{MX}.stack_{top} \in frame_{\mu} \\ \forall\, i < 32. \quad i \notin \{sp, bp\} \longrightarrow c_{MX}.ac.gpr[i] = [gpr_{hyb}[i]]^{\theta,\pi}_{c_{IL}} \\ \forall\, i < 32. \quad c_{MX}.spr[i] = [spr_{hyb}[i]]^{\theta,\pi}_{c_{IL}} \\ \neg is\_stack(c_{MX}) \longrightarrow c_{MX}.ac.gpr[sp] = [gpr_{hyb}[sp]]^{\theta,\pi}_{c_{IL}} \wedge c_{MX}.ac.gpr[bp] = [gpr_{hyb}[bp]]^{\theta,\pi}_{c_{IL}} \end{array}}{rel^{regs}_{MX-IL}(c_{MX}, c_{IL})}$$

The memory consistency rel^{mem}_{MX-IL} states that the content of non-hybrid memories of both machines is the same.

$$\frac{\forall\, a < 2^{32} - 1. \quad c_{MX}.\mathcal{M}(a) = c_{IL}.\mathcal{M}(a)}{rel^{mem}_{MX-IL}(c_{MX}, c_{IL})}$$

The stack consistency rel^{stack}_{MX-IL} states that the content of components of all stack frames of two given *MX* and *C-IL* machines is the same. If the i-th stack frame of the *MX* machine c_{MX} is of *C-IL* type, then it must be completely the same as the i-th stack frame of the *C-IL* machine c_{IL}. Otherwise, values stored in components $lifo$, $pars$ and $saved$ of the i-th stack frame of the *MX* machine c_{MX} are the same as values stored in the hybrid global variables $lifo_{hyb}$, $pars_{hyb}$ and $saved_{hyb}$ of the *C-IL* machine c_{IL}, respectively, at the corresponding positions. Figure 7.6 depicts the relation between the lifo of μ_{ASM} stack frames of some particular stack of a *MX* machine and the corresponding parts of the hybrid lifo of a *C-IL* machine.

In the following we define the relation rel^{stack}_{MX-IL} with $s = c_{MX}.stack^{flat}$, $s_i = s[i]$, $s' = \textbf{filter}(\lambda\, a.\, a \in frame_{\mu}, s)$ and $s'_i = s'[i]$ as

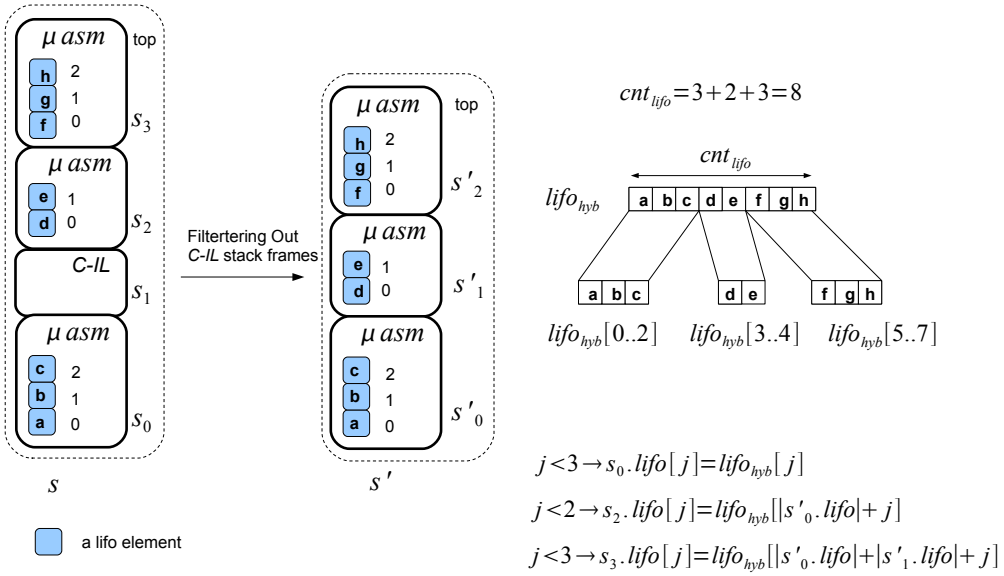


Figure 7.6: Relation between μ_{ASM} Lifo and Hybrid Lifo

$$\frac{\begin{array}{ll}(\forall\, i < |s|. & (s_i \in frame_{C\text{-}IL} \longrightarrow s_i = c_{IL}.stack[i])) \\ \wedge & \\ (\forall\, i < |s'|. & \quad (\forall\, j < |s'_i.lifo|.\ s'_i.lifo[j] = [\ lifo_{hyb}[j + \textstyle\sum_{t<i} |s'_t.lifo|]\ ]^{\theta,\pi}_{c_{IL}}) \\ & \wedge \quad (\forall\, j < |s'_i.pars|.\ s'_i.pars[j] = [\ pars_{hyb}[j + \textstyle\sum_{t<i} |s'_t.pars|]\ ]^{\theta,\pi}_{c_{IL}}) \\ & \wedge \quad (\forall\, j < |s'_i.saved|.\ s'_i.saved[j] = [\ saved_{hyb}[j + \textstyle\sum_{t<i} |s'_t.saved|]\ ]^{\theta,\pi}_{c_{IL}}))\end{array}}{rel^{stack}_{MX-IL}(c_{MX}, c_{IL})}$$

The control consistency $rel^{control}_{MX-IL}$ states that the location counter and the function name of the i-the stack frame of the both machines are the same. In case the i-th stack frame of the MX machine is of μ_{ASM} type the location counter of this stack frame must be recomputed with the help of the function $Location_{MX2IL}$.

$$\frac{\begin{array}{ll} & \forall\, i < |c_{MX}.stack^{flat}|. \\ & c_{MX}.f_i = c_{IL}.stack[i].f \\ \wedge & (c_{MX}.stack^{flat}[i] \in frame_{C\text{-}IL} \longrightarrow c_{MX}.loc_i = c_{IL}.stack[i].loc) \\ \wedge & (\ c_{MX}.stack^{flat}[i] \in frame_{\mu} \longrightarrow \\ & \qquad Location_{MX2IL}(c_{MX}.f_i, c_{MX}.loc_i, \pi) = c_{IL}.stack[i].loc)\end{array}}{rel^{control}_{MX-IL}(c_{MX}, \pi, c_{IL})}$$

Simulation Theorem

After defining the simulation relation we can finally state the simulation theorem:

THEOREM 7.1 ► Simulation Theorem

Let π be a MX program, θ be system properties. Then for all steps i of the MX machine c_{MX} executing program π there exists a step number $s(i)$ of the $C\text{-}IL$ machine c_{IL} executing the translated version of the program π, such that the MX machine after

i steps is consistent with the *C-IL* machine after $s(i)$ steps. However, it is true if the following requirements over π, c_{MX} and c_{IL} are fulfilled:

1. The *MX* program π must be translatable,
2. c_{IL} and c_{MX} machines start their executions in same functions at the corresponding locations (for the c_{IL} machine it can be a translated function),
3. The registers of the initial configuration of *MX* machine c_{IL}^{0} and the hybrid registers modeled in the hybrid memory of the *C-IL* machine $c_{IL}^{s(0)}$ have the same content as well as the content of non-hybrid memories of both machines is the same.
4. All μ_{ASM} procedures called from *C-IL* functions in the *MX* program do not change the callee-save registers or they restore them before the return if changed.
5. The hybrid counters are initialized with zeros in the machine *C-IL*,
6. For all steps of the *MX* machine up to step i this machine does not get stuck,
7. Regions allocated for hybrid variables do not intersect with each other, and
8. For all steps of the *C-IL* machine up to step $s(i)$ there is no buffer overflow when accessing hybrid arrays.

$$
\begin{array}{lll}
\forall\, i. & \exists\, s :: \mathbb{N} \mapsto \mathbb{N}.\ \exists\, c'_{MX}.\exists\, \pi'. & \\
 & \pi, \theta \vdash c_{MX} \rightarrow^{i}_{MX} \lfloor c'_{MX} \rfloor \quad \text{(precondition 6)} & \\
\wedge & \pi' = \tau_{MX2IL}(\pi) & \\
\wedge & \text{preconditions 1,2,3,4,5,7,8,} & \\
\longrightarrow & \exists\, c'_{IL}.\quad \pi', \theta \vdash c_{IL} \rightarrow^{s(i)}_{IL} c'_{IL} & \\
 & \qquad \wedge\ rel_{MX-IL}(c'_{MX}, \pi.\pi_{\mu_{\text{ASM}}}, c'_{IL}) &
\end{array}
$$

Proof: We prove this theorem by induction on the number of steps i performed by the *MX* machine.

For the induction start we have to show that both machines are consistent. From the precondition 2, we know that both machines will start their execution in the same function (procedure). Here we distinguish on type of a function (procedure) to be executed by the c_{MX} machine.

- If the c_{MX} machine starts the execution in a *C-IL* function from the program π, then the *C-IL* machine starts with the execution with the same statement in the same *C-IL* function of the translated program π', too. By the definition of the τ_{MX2IL} function it preserves the implementation of original *C-IL* functions from the program π in the translated program π'. Therefore, both machines are consistent at the start.
- In case the c_{MX} machine starts with the execution of a μ_{ASM} procedure, then the *C-IL* machine starts with the execution of this translated procedure. To show that both machines are consistent we need to show a single consistency, namely the control consistency. The other consistencies hold as the *MX* machine starts its execution with a completely empty stack, the registers in both machines have the same values (precondition 3), the non-hybrid memories are the same in both machines. The control consistency must hold by the definition of the function $size_{MX2IL}$.

For the induction step we have to conclude from step i to $i+1$. Here we distinguish on the type of a *C-IL* statement (a μ_{ASM} instruction) to be executed at step $i+1$ by the c_{MX} machine.

- a *C-IL* statement. Here we distinguish three cases whether this statement makes
  - a pure step. Since after translating the *MX* program π to the *C-IL* program π' the original implementation of *C-IL* functions from the program π stays the same in the translated programs. Therefore, after executing the *C-IL* statement both machines are still consistent after performing this step.
  - an inter-language return (from *C-IL* to μ_{ASM}). This case is proven by the definition of the translation function $\tau_{\mu asm2c}$.
  - an inter-language call (from *C-IL* to μ_{ASM}). This case is shown by the definition of the translation function $\tau_{\mu asm2c}$.
- a μ_{ASM} instruction. We distinguish on the type of a μ_{ASM} instruction to be executed. All cases can be shown by the definition of the translation function $\tau_{\mu asm2c}$ except for the case when the μ_{ASM} instruction to be executed at step $i+1$ by the *MX* machine is an inter-language return (from μ_{ASM} to *C-IL*) where we need to apply the precondition 4. Recall that this precondition states that the μ_{ASM} procedures called from *C-IL* do not change the callee-save registers or they restore them before the return if changed.

  Let us consider the case when a μ_{ASM} instruction to be executed is of VAMP assembly type and its execution will update the content of one of stack registers bp and sp. We know from preconditions to the main theorem that the execution of the *MX* machine can not get stuck. Thus, the execution of this instruction by the *MX* machine does not yield to an error state. That means, after updating one of these stack registers there are three possible cases :
  - Stack is created: It is the case when two stack registers are properly set put in the c_{MX} machine. From the definition of the function $\tau_{\mu asm2c}$ we know that the corresponding *C-IL* statements simulating this instruction will set up hybrid counters to zeros (cf. Listing 7.1). After the stack is created as the side effect of the instruction execution in the *MX* machine all components of the first stack frame of this machine are empty. And after executing the *C-IL* statements simulating this instruction all hybrid counters contain zeros. With that we can simply show that the translater consistency holds after executing this VAMP assembly instruction and the simulating *C-IL* statements by the *MX* and *C-IL* machines, respectively.
  - One of stack registers is set up: In this case the hybrid counters contain zeros after executing the corresponding *C-IL* statements (cf. Listing 7.1).
  - Stack is destroyed: The stack destruction in the *MX* machine also means that all hybrid counters must be zeros after executing the corresponding *C-IL* statements (cf. Listing 7.1).

■

CHAPTER 8

Automated Verification of a Small Hypervisor

In [AHPP10] the authors have formally proven Lemmata 6.26 and 6.32 (cf. Table 8.1) and assumed Lemmata 6.22, 6.30 (cf. Table 8.1) in VCC [CDH+09]. From [Cor08]: "VCC is a mechanical verifier for concurrent C programs. VCC takes a C program, annotated with function specifications, data invariants, loop invariants, and ghost code, and tries to prove these annotations correct. If it succeeds, VCC promises that your program actually meets its specifications.". That is, they shown the correctness of C code of the BHV program and guest steps in VCC. For that, they have created a simulation framework in which verification engineers can reason on overall system correctness in an efficient and pervasive manner. In [AHPP10] they left a few gaps for future work: (i) the implementation of an interrupt service routine (like presented in Listing 6.4), (ii) the verification of this interrupt service routine which they have just specified but partially (we will come to it a bit later). In [AHPP10] the authors did not talk about the boot code at all, but they have assumed that the interrupt service routine is located in memory starting from address zero. In Chapter 6 we assumed that the boot code is located in memory starting from address zero and the context switch (which they called an interrupt service routine) is a part of the baby hypervisor code.

As practical work for this thesis we have formally proven Lemmata 6.25, 6.24, 6.27 and 6.31 in VCC (cf. Table 8.1). So we have shown the functional correctness of BHV μ_{ASM} code presented in Listing 6.4. To verify this μ_{ASM} code we first translated it to C and then applied the automatic C verifier VCC to show the correctness of the translated code. Previously, we have shown the correctness of the translation we used here. Here we want to stress that each instruction from the macro-assembly code is translated to the corresponding call of a C function wrapping its simulating C statements. Doing the translation this way we save a number of annotations in VCC and reduce the run-time of the verification.

After we were done with the formal verification of the context switch of the BHV we have successfully integrated formal proofs done within this thesis into the sim-

Paper/Pencil Statements	Status in VCC	Function		File
Lemma 6.22	assumed	*sim*		sim.c
Lemma 6.30	assumed	*sim*		sim.c
Lemma 6.26	verified	*hv_dispatch*	contracts	baby.c
Lemma 6.32	verified	*hv_dispatch*	contracts	baby.c
Lemma 6.25	verified	*save_guest*	block 4	sim_masm.c
Lemma 6.24	verified	*save_guest*	block 5	sim_masm.c
Lemma 6.27	verified	*restore_guest*	contracts	sim_masm.c
Lemma 6.31	verified	*save_guest*	contracts	sim_masm.c
BHV Theorem	verified	*sim*	loop-inv	sim.c

Table 8.1: BHV Paper/Pencil Vs Formal Proofs

ulation framework in which the authors of [AHPP10] have successfully verified C portions of the BHV. In order to accomplish this integration we extended the specification of the BHV context switch which authors presented in [AHPP10] and we extended all corresponding loop invariants and function contracts. For example, the authors left contracts of the function *page_alloc* for the allocated number of bytes imprecise (it should be a multiple of a page size (4Kbytes)), but the authors of [AHPP10] made a note about that.

As a result of this integration there is a simulation framework[1] in which proofs of authors of [AHPP10] run together with our proofs in VCC. In this simulation framework we made notes at all places where we added conditions needed for this integration.

[1] Verified source is available at `http://dl.dropbox.com/u/5648134/BHV.rar`

CHAPTER 9

Summary and Future Work

This thesis presents to the best of our knowledge

- the first macro-assembly (high-level assembly) languages semantics with explicit dynamic stack. The ability to model stack explicitly in an assembly language is critical for supporting procedure calls.
- the first mixed semantics to integrate C- and high-level assembly languages together in a single language. So we presented a so-called *mixed* semantics which describes this language and which offers certain benefits: the person who does soundness proofs for code-verification tools no longer needs to consider the inner workings of compilers- instead, they can use integrated semantics as foundation of their proofs.
- the first complete paper-and-pencil proof of a hypervisor programmed in C and macro-assembly. The mixed semantics shows how it really becomes simple to reason about the functional correctness of mixed implementations.
- the first detailed justification of a simulation approach used to verify code written in an assembly language with a general-purpose C verifier. We conclude the following that the simulation approach as it sketched in [MMS08, Mau11] is sound if and only if the assembly code to be verified does not update stack registers when the stack is set up.
- the first completely formally proven functional correctness of a hypervisor. The C portions of the small hypervisor were formally shown in [AHPP10]. The implementation of this hypervisor is sequential and non-preemptive and the architecture it virtualizes has a single core.

The work created by the authors of [AHPP10] consists of four component: VAMP specification and simulator, hypervisor implementation, and system program. This work comprises ca. 2.5k C code tokens and ca. 7.7k annotation tokens, which comprise function contracts, assertions, data and loop invariants, and spec(ghost) code.

After extending this work with macro-assembly verification environment (implemented in C) it comprises ca. 2.7k C code tokens and ca. 8.2k annotation tokens. Overall proof time is about 1.5-2 hours on one core of a 2.4 GHz Intel Core Duo machine (cf. Appendix A) in the second version of VCC[1].

Finally, we point out possible directions of future work.

- Generalize the high-level assembly and the integrated semantics for any kind of compiler calling conventions.
- Consider interprocedural optimizations in the optimizing C compiler correctness theory.
- Extend the high-level assembly and the integrated semantics so that it would be possible to describe implementations doing a *stack switch* (a part of the *thread switch*).
- To extend the framework presented in [AHPP10] to more complex hardware and software designs (as it is proposed in [AHPP10]): modeling of multi-core systems, translation-look aside buffers (TLBs), and devices (i.e. external interrupts), and verification of multi-threaded or pre-emptive kernels.
- To adapt the baby hypervisor annotations (pre- and post-conditions, data structure invariants, assertions, etc.) written for the second version of VCC to the third version of VCC.
- To verify boot code and integrate the achieved correctness results into the overall formal correctness proof. But it requires to model the VAMP architecture with devices in VCC.
- To justify the simulation framework created by the authors of [AHPP10] which is presented in Sections 5 and 6 of [AHPP10]. This framework is used to verify a system software, like the BHV, with a C verifier, like VCC. Namely, this simulation framework is used to express top-level system correctness of the baby hypervisor as a *system program* whose execution models the system behavior by an infinite loop of host machine steps. The top-level system correctness is then expressed as an invariant of this loop.

[1] there is a third version of VCC which is available since 2011

Appendix A: Verification Run-Time of a Small Hypervisor in Seconds

```
Verification of proc_t#adm succeeded. [3,89]
Verification of guest_t#adm succeeded. [0,48]
Verification of hv_t#adm succeeded. [0,30]
Verification of compute_ea succeeded. [0,03]
Verification of systemmode succeeded. [0,02]
Verification of asm_lw succeeded. [0,22]
Verification of asm_lw_guest_t_ptr succeeded. [0,16]
Verification of asm_sw succeeded. [0,05]
Verification of asm_addi succeeded. [0,09]
Verification of asm_andi succeeded. [0,08]
Verification of asm_movs2i succeeded. [0,09]
Verification of asm_movi2s succeeded. [0,25]
Verification of asm_rfe succeeded. [0,14]
Verification of save_guest succeeded. [61,69]
Verification of restore_guest succeeded. [24,63]
Verification of sim_inner succeeded. [39,95]
Verification of sim succeeded. [57,08]
Verification of asm_sw#bv_lemma#0 succeeded. [0,45]
Verification of restore_guest#bv_lemma#0 succeeded. [0,38]
Verification of restore_guest#bv_lemma#1 succeeded. [0,44]
Verification of restore_guest#block#0 succeeded. [5,66]
Verification of restore_guest#block#1 succeeded. [2,88]
Verification of restore_guest#block#2 succeeded. [3,36]
Verification of restore_guest#block#3 succeeded. [4,95]
Verification of save_guest#block#0 succeeded. [1,70]
Verification of save_guest#block#1 succeeded. [4,20]
Verification of save_guest#block#2 succeeded. [4,11]
Verification of save_guest#block#3 succeeded. [3,98]
Verification of sim#block#0 succeeded. [2,00]
Verification of sim_inner#block#0 succeeded. [0,33]
Verification of sim_inner#block#1 succeeded. [465,00]
Verification of sim_inner#block#1#0 succeeded. [0,88]
Verification of proc_t#adm succeeded. [3,89]
Verification of guest_t#adm succeeded. [0,31]
Verification of hv_t#adm succeeded. [0,19]
Verification of page_alloc_t#adm succeeded. [0,05]
```

```
Verification of compute_ea succeeded. [0,02]
Verification of systemmode succeeded. [0,02]
Verification of min succeeded. [0,00]
Verification of ptea_vpx succeeded. [0,02]
Verification of hv_dispatch succeeded. [1498,03]
Verification of update_spt succeeded. [8,25]
Verification of handle_movi2pto succeeded. [52,66]
Verification of handle_movi2ptl succeeded. [57,72]
Verification of handle_pf succeeded. [103,20]
Verification of page_alloc_init succeeded. [0,38]
Verification of page_alloc succeeded. [1,70]
Verification of reset_guest succeeded. [431,72]
Verification of wrap_hv_test succeeded. [0,81]
Verification of handle_reset succeeded. [1636,11]
Verification of handle_illegal succeeded. [75,56]
Verification of handle_movi2ptl#bv_lemma#0 succeeded. [0,45]
Verification of handle_movi2ptl#bv_lemma#1 succeeded. [0,42]
Verification of handle_movi2ptl#bv_lemma#2 succeeded. [0,38]
Verification of handle_movi2ptl#bv_lemma#3 succeeded. [0,42]
Verification of handle_movi2ptl#bv_lemma#4 succeeded. [0,48]
Verification of handle_movi2ptl#bv_lemma#5 succeeded. [0,38]
Verification of handle_movi2ptl#bv_lemma#6 succeeded. [0,39]
Verification of handle_movi2ptl#bv_lemma#7 succeeded. [0,41]
Verification of handle_movi2pto#bv_lemma#0 succeeded. [0,41]
Verification of handle_movi2pto#bv_lemma#1 succeeded. [0,44]
Verification of handle_movi2pto#bv_lemma#2 succeeded. [0,38]
Verification of handle_movi2pto#bv_lemma#3 succeeded. [0,41]
Verification of handle_movi2pto#bv_lemma#4 succeeded. [0,38]
Verification of handle_movi2pto#bv_lemma#5 succeeded. [0,41]
Verification of handle_movi2pto#bv_lemma#6 succeeded. [0,42]
Verification of handle_pf#bv_lemma#0 succeeded. [0,42]
Verification of hv_dispatch#bv_lemma#0 succeeded. [0,47]
Verification of page_alloc#bv_lemma#0 succeeded. [0,50]
Verification of reset_guest#bv_lemma#0 succeeded. [0,44]
Verification of reset_guest#bv_lemma#1 succeeded. [0,44]
Verification of reset_guest#bv_lemma#2 succeeded. [0,39]
Verification of reset_guest#bv_lemma#3 succeeded. [0,42]
Verification of update_spt#bv_lemma#0 succeeded. [0,39]
Verification of update_spt#bv_lemma#1 succeeded. [0,42]
Verification of update_spt#bv_lemma#2 succeeded. [0,38]
Verification of update_spt#bv_lemma#3 succeeded. [0,44]
Verification of update_spt#bv_lemma#4 succeeded. [0,39]
Time: 4581,92s total, 10,25s compiler, 0,00s boogie, 4571,67s verification
```

Bibliography

[ACH+10] E. Alkassar, E. Cohen, M. Hillebrand, M. Kovalev, and W. Paul. Verifying shadow page table algorithms. In *Formal Methods in Computer Aided Design (FMCAD) 2010*, pages 267–270, Lugano, Switzerland, 2010. IEEE.

[AHL+09] E. Alkassar, M. A. Hillebrand, D. C. Leinenbach, N. W. Schirmer, A. Starostin, and A. Tsyban. Balancing the load: Leveraging semantics stack for systems verification. In *Journal of Automated Reasoning: Special Issue on Operating Systems Verification*. Springer, 2009.

[AHPP10] E. Alkassar, M. Hillebrand, W. Paul, and E. Petrova. Automated verification of a small hypervisor. In *Third International Conference on Verified Software: Theories, Tools, and Experiments (VSTTE'10)*, volume 6217 of *LNCS*, pages 40–54, Edinburgh, UK, 2010. Springer.

[Alk09] Eyad Alkassar. *OS Verification Extended. On the Formal Verification of Device Drivers and the Correctness of Client/Server Software.* PhD thesis, Saarland University, Computer Science Department, 2009.

[APST10] Eyad Alkassar, Wolfgang J. Paul, Artem Starostin, and Alexandra Tsyban. Pervasive verification of an os microkernel: inline assembly, memory consumption, concurrent devices. In *Proceedings of the Third international conference on Verified software: theories, tools, experiments*, VSTTE'10, pages 71–85, Berlin, Heidelberg, 2010. Springer-Verlag.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[Bev87] W. R. Bevier. *A Verified Operating System Kernel*. PhD thesis, University of Texas at Austin, 1987.

[Bev89] William R. Bevier. Kit and the short stack. *J. Autom. Reasoning*, 5(4):519–530, 1989.

[BHMY89a] William R. Bevier, Warren A. Hunt, Jr., J S. Moore, and William D. Young. An approach to systems verification. 5(4):411–428, December 1989.

[BHMY89b] W.R. Bevier, W.A. Hunt, J. Strother Moore, and W.D. Young. Special issue on system verification. *Journal of Automated Reasoning*, 5(4):409–530, 1989.

[BJK+03] S. Beyer, C. Jacobi, D. Kröning, D. Leinenbach, and W.J. Paul. Instantiating uninterpreted functional units and memory system: functional verification of the vamp. In *CHARME 2003*, volume 2860 of *LNCS*, pages 51–65. Springer, 2003.

[BJK+05] S. Beyer, C. Jacobi, D. Kröning, D. Leinenbach, and W.J. Paul. Putting it all together - formal verification of the vamp. 2005.

[CDH+09] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michael Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. Vcc: A practical system for verifying concurrent c. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference. International Conference on Theorem Proving in Higher Order Logics (TPHOLs-09), August 17-20, Munich, Germany*, volume 5674 of *Lecture Notes in Computer Science, LNCS*, pages 23–42. Springer, 8 2009.

[Cor] Microsft Corp. *Programmer's Guide Microsft MASM Version 6.1*.

[Cor08] Microsfot Corp. Vcc: A c verifier, 2008.

[Dal06] Iakov Dalinger. *Formal Verification of a Processor with Memory Management Units*. PhD thesis, Saarland University, Computer Science Department, July 2006.

[DHP05] Iakov Dalinger, Mark Hillebrand, and Wolfgang Paul. On the verification of memory management mechanisms. In D. Borrione and W. Paul, editors, *Proceedings of the 13th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME 2005)*, volume 3725, pages 301–316. Springer, 2005.

[DPS09] Ulan Degenbaev, Wolfgang J. Paul, and Norbert Schirmer. Pervasive theory of memory. In Susanne Albers, Helmut Alt, and Stefan Näher, editors, *Efficient Algorithms – Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday*, volume 5760 of *Lecture Notes in Computer Science*, pages 74–98. Springer, 2009.

[FN79] R. Feiertag and P. Neumann. The foundations of a provably secure operating system (PSOS). In *Proceedings of the National Computer Conference 48*, pages 329–334, 1979.

[Fru08] Nicu G. Fruja. Towards proving type safety of .net cil. *Sci. Comput. Program.*, 72:176–219, August 2008.

[FSDG08] Xinyu Feng, Zhong Shao, Yuan Dong, and Yu Guo. Certifying low-level programs with hardware interrupts and preemptive threads. In *2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*, New York, NY, USA, June 2008. ACM.

[GHLP05] Mauro Gargano, Mark Hillebrand, Dirk Leinenbach, and Wolfgang Paul. On the correctness of operating system kernels. In J. Hurd and T. F. Melham, editors, *18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005)*, volume 3603, pages 1–16. Springer, 2005.

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, second edition, 1996.

[IdR09] Thomas In der Rieden. *Verified Linking for Modular Kernel Veri fication*. PhD thesis, Saarland University, Saarbrcken, 2009.

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 207–220, Big Sky, MT, USA, October 2009. ACM.

[Kle09] Gerwin Klein. Operating system verification — an overview. *Sādhanā*, 34(1):27–69, February 2009.

[Lei07] Dirk Leinenbach. *Compiler Verification in the Context of Pervasive System Verification*. PhD thesis, Saarland University, Computer Science Department, 2007.

[Lip86] Seymour Lipschutz. *Schaum's Outline of Theory and Problems of Data Structures*. McGraw-Hill, New York, 1986.

[LP08] D. Leinenbach and E. Petrova. Pervasive compiler verification – from verified programs to verified systems. In *3rd intl Workshop on Systems Software Verification (SSV08)*. Elsevier Science B. V., 2008.

[LS09] D. Leinenbach and T. Santen. Verifying the microsoft hyper-v hypervisor with vcc. In *16th International Symposium on Formal Methods (FM 2009)*, volume 5850 of *Lecture Notes in Computer Science*, pages 806–809, Eindhoven, the Netherlands, 2009. Springer.

[Mau11] Stefan Maus. *Verification of Hypervisor Subroutines written in Assembler*. PhD thesis, Freiburg University, Computer Science Department, 2011.

[MCGW98] J. Gregory Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. In *Types in Compilation*, pages 28–52, 1998.

[MMS08] Stefan Maus, Michal Moskal, and Wolfram Schulte. Vx86: x86 assembler simulated in c powered by automated theorem proving. In *AMAST*, pages 284–298, 2008.

[Moo03] J Strother Moore. A grand challenge proposal for formal methods: A verified stack. In Bernhard K. Aichernig and T. S. E. Maibaum, editors, *10th Anniversary Colloquium of UNU/IIST*, volume 2757 of *LNCS*, pages 161–172. Springer, 2003.

[MP00] Silvia M. Mueller and Wolfgang J. Paul. *Computer Architecture: Complexity and Correctness*. Springer, 2000.

[NBF+80] P.G. Neumann, R.S Boyer, R.J. Feiertag, K.N. Levitt, and L. Robinson. *A provably secure operating system: The system, its applications, and proofs. Technical Report CSL-116*. Computer Science Laboratory, SRI International, Menlo Park, California, May 1980.

[NET11] .net languages, web pages at, 2011.

[NF03] P. Neumann and R. Feiertag. PSOS revisited. In *19th Annual Computer Security Applications Conference*, 2003.

[NS06] Zhaozhong Ni and Zhong Shao. Certified assembly programming with embedded code pointers. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 320–333, New York, NY, USA, 2006. ACM.

[NYS07] Zhaozhong Ni, Dachuan Yu, and Zhong Shao. Using xcap to certify realistic systems code: Machine context management. In *TPHOLs*, pages 189–206, 2007.

[Pau07] Wolfgang Paul. System architecture. lecture notes. `http://busserver.cs.uni-sb.de/lehre/vorlesung/info2/ss08/material/mitschrift07.pdf`, 2007.

[SS12] Sabine Schmaltz and Andrey Shadrin. Integrated semantics of intermediate-language-c+macro-assembler for pervasive formal verification of operating systems and hypervisors from verisoftxt. In *The Fourth International Conference on Verified Software: Verified Software: Theories, Tools, and Experiments (VSTTE 2012) (to appear)*, January 2012.

[Sta10] Artem Starostin. *Formal Verification of Demand Paging*. PhD thesis, Saarland University, Saarbrcken, 2010.

[TKN07] Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 97–108, New York, NY, USA, 2007. ACM.

[Tsy09] Alexandra Tsyban. *Formal Verification of a Framework for Microkernel Programmes*. PhD thesis, Saarland University, Computer Science Department, 2009.

[Tve09] Sergey Tverdyshev. *Formal Verification of Gate-Level Computer Systems*. PhD thesis, Saarland University, Computer Science Department, 2009.

[Vera] Verisoft Consortium. The Verisoft Project. `http://www.verisoft.de/`.

[Verb] VerisoftXT Consortium. The VerisoftXT Project. `http://www.verisoftxt.de/`.

[WKP79] Bruce J. Walker, Richard A. Kemmerer, and Gerald J. Popek. Specification and verification of the UCLA Unix security kernel. In *SOSP '79: Proceedings of the seventh ACM symposium on Operating systems principles*, pages 64–65, New York, NY, USA, 1979. ACM.