Diploma Thesis

Automatic Verification of Conditions for Absence of Interrupts

Pavel Emeliyanenko
asm@wjpserver.cs.uni-sb.de

Advisors:
Tom In der Rieden
Anne Proetzsch

Prof. Dr. W. J. Paul
Prof. Dr. B. Finkbeiner

Saarland University, Computer Science Department
Institute for Computer Architecture and Parallel Computing
Reactive Systems Group
September 2005

Abstract

In order to the have possibility of verifying assembler programs written for the VAMP (Verified Architecture Microprocessor[1]) using the abstract *software machine* we need to relate the VAMP formal specification with the abstract software machine specification.

The software machine does not support interrupt handling and therefore programs to be executed on this machine should not produce any interrupts. In the previous work [11] the conditions for absence of interrupts were identified and their validity was proved using the PVS verification system. Besides that, a theorem was established which states that under certain conditions (including the conditions for absence of interrupts) the execution of a program on the software machine is equivalent to the execution of this program on the VAMP.

Consequently, if some property of a program has been proved using the software machine and the conditions stated in the theorem hold, then the same property holds during the execution of this program on the VAMP. The goal of this work is to develop software with help of which the required conditions for absence of interrupts can be proved automatically for most assembler programs.

[1] http://www-wjp.cs.uni-sb.de/forschung/projekte/VAMP

Contents

Introduction

Nowadays a great part of jobs is done by computers. Computers have found a good use not only in the field of science. Most of the modern technological processes are fully automated, i.e. they are operated by microprocessors. It brings us a lot of advantages, since a huge amount of errors is happened through human's fault, and application of the computer systems in industry makes possible to avoid most of them. Undoubtedly, it is extremely important to verify that a computer system works correctly in all possible cases, especially when talking about real-time systems where each discovered bug may result in a huge material or human loss.

Previously exploited techniques of verification such as simulation or testing are not really trustworthy. Quite effective in the very early stages of debugging they become absolutely inapplicable for the entire designs since the state space of modern systems is huge and tests never attain full coverage.

Another advantageous alternative to simulation and testing is the approach of formal verification. Using the formal verification technique all possible behaviours of system can be thoroughly explored and the correctness of the whole system can be proved.

At Saarland University the correctness of the complete processor called the VAMP has been formally proved (see [2]). It features a Tomasulo-scheduled five-stage pipeline, precise interrupts, delayed branch, cache memory, and a fully IEEE-compliant dual-precision floating point unit that handles denormals and exceptions entirely in hardware.

An important task now is to develop verified software for the VAMP processor. The verification of the assembler programs is the task of *Assembler Verification Project*. Solely for this purposes a *formal assembler machine* specification has been developed which implements a subset of the VAMP instruction set. Several design decisions have been made to keep the assembler verification with this machine feasible and simple.

However, to argue about the correctness of programs using the abstract software machine we need to state that under certain conditions the executions of an assembler program on the abstract software machine and on the VAMP produce the same results. These conditions include the equivalence of the memory regions of both machines and the absence of interrupts. The last property states that a program should not produce any interrupts, when running on the VAMP. The validity of these conditions has been formally proved in [11] using the PVS theorem proving system.

The goal of this thesis is to develop software which will help us to prove auto-

matically that a certain assembler program satisfies the property for the absence of interrupts. The idea behind this automated method is to translate a given assembler program into *control flow graph* representation and to formulate the required conditions in the form of verification diagrams.

In the control flow representation a program is subdivided into *basic blocks*. Basic blocks are sequences of instructions which are always executed sequentially by the processor. This means, that branches or jumps are only allowed at (or near) the end of basic blocks. Basic blocks are connected with each other by edges representing the flow of a control in a program.

Afterwards, the nodes (basic blocks) of a control flow graph are identified with the nodes of a verification diagram which for most cases can be constructed automatically based on the program source code. Verification diagrams [8] are directed graphs whose nodes are labeled by propositional formulas (assertions), representing sets of system states, and whose edges represent possible system transitions. Verification diagrams correspond to a completed, direct proof, and offer a compact representation of the necessary verification conditions. Later on the refinement tool[1] extracts the verification conditions based on a given control flow graph and a verification diagram and proves the validity of the required properties.

To show the effectiveness of this approach we present a proof for the absence of interrupts for a simple assembler program.

[1] https://react.cs.uni-sb.de/software/tv-tool

1 Overview of the VAMP

In the VAMP [1] (Verified Architecture Microprocessor) project, the correctness of a complete microprocessor called VAMP has been formally proved. The features of this processor are:

- the full DLX instruction set;
- delayed branches;
- Tomasulo scheduler;
- maskable nested precise interrupts;
- pipelined fully IEEE 754 compatible dual precision floating point unit with variable latency;
- separate instruction and data caches;

The processor has been designed, functionally verified, and synthesized. Specification and verification has been performed using the interactive theorem proving system PVS [10]. All formal specifications and proofs can be found on the project web site[1].

The hardware description of the processor has been automatically extracted from PVS and translated into Verilog HDL by a tool called pvs2hdl. The tool unrolls recursive definitions and then performs fairly straightforward translation.

The Verilog representation of the processor (including caches and floating point unit) has been synthesized, implemented, and tested on a Xilinx FPGA hosted on a PCI board. Some additional unverified hardware for controlling the VAMP processor and for accessing its memory from the host PC is also present on this FPGA. The VAMP processor occupies about 18000 slices of a Xilinx Virtex FPGA. This accounts for a gate count of 1.5 million gates as reported by the Xilinx tools. The design contains 9100 bits of registers (not counting memory and caches) and runs at 10 MHz.

In this chapter we make a short overview of the VAMP processor, we describe the DLX instruction set architecture, and give a short description of the VAMP interrupts.

1.1 The VAMP Processor

The VAMP processor is a variant of the DLX processor described in [6]. It supports full DLX fixed point instruction set as well as floating point extension given in [9].

[1] http://www-wjp.cs.uni-sb.de/forschung/projekte/VAMP

The VAMP processor is a RISC architecture. The VAMP instruction set architecture uses general-purpose registers with a load/store architecture, supports register, immediate and displacement addressing modes and uses fixed instruction encoding. It uses distinct register files, namely a general purpose register file GPR, a special purpose register file SPR and a floating point register file FPR. The first two register files are described below in detail.

Instructions are stored in the memory (each instruction is 32-bit wide) and fetched using a 32-bit register called *program counter* and denoted PC which represents an address of the fetched instruction in the memory.

The VAMP processor uses the so called *delayed PC* architecture with one delay slot instruction described in [9]. In such a delayed PC architecture, instruction updates to the PC do not affect the next instruction, only the instruction after the next one. Hence, the instruction after a jump instruction is always executed before the actual jump takes place and we have two PCs in the programmer's model, PC' and DPC. In an execution step, the instruction pointed to by DPC is executed, but the PC update of the instruction only affects PC'. Simultaneously, the old value of PC' is written into DPC which creates the delay slot. Note that, according to the delayed PC semantics, branch or jump instructions cannot occupy the delay slots.

The VAMP also has two separate caches (for data and instructions), connected to a single main memory. The VAMP uses a write back policy for the data cache, i.e., on a write access of the CPU, the data cache is updated and the corresponding data is marked as dirty. Thus, a slow access to the main memory is not necessary. If dirty data has to be thrown out from the cache, it is written back to the main memory in order to ensure data consistency.

1.2 Instruction set architecture

The VAMP ISA uses the general purpose register file GPR and can access the special purpose register file SPR. The fixed point instruction set includes loads and stores for double words, words, half words, and bytes, various shift operations, and two jump-and-link operations. Loads of bytes and half words can be unsigned or signed.

The general purpose register file GPR contains 32 registers, named $R0$, $R1$, ..., $R31$. Each register is 32 bit wide. They can be used by fixed point instructions but there are also instructions which can transfer data between GPR and other register files. The value of $R0$ is always 0. This register can be used to synthesize different useful operations from the simple instruction set.

Number	Register	Usage
0	SR	Status register. Contains interrupt mask bits.
1	ESR	Exception status register. Saves SR in case of an interrupt.
2	ECA	Exception cause register. Saves exception cause in case of an interrupt.
3	EPC	Exception PC. Saves PC' in case of an interrupt.
4	$EDPC$	Exception DPC. Saves DPC in case of an interrupt.
5	$EData$	Exception data. Saves additional exception data in case of an interrupt.
6	RM	Rounding mode. Encodes currently used rounding mode for all floating point operations.
7	$IEEEf$	IEEE flags register. Required by the IEEE standard to accumulate floating point interrupts.
8	FCC	Floating point condition code. Used to store result of floating point comparisons.
9	PTO	Page table origin register
10	PTL	Page table length register
16	$MODE$	Mode register. Holds the current mode of the VAMP (0 - system mode, 1 - user mode)
11	$EMODE$	Exception mode register. Stores the old $MODE$ value during the execution of the ISR

Table 1.1: Special purpose register of the VAMP fixed point core

Special purpose registers (SPRs) are used to serve specific tasks, they are listed in table 1.1. The remaining registers from the SPR file are unused and always return zero.

As mentioned above all instructions in the VAMP are encoded with 32-bit values. The VAMP processor has five different instruction formats (three of them are used to describe fixed point instructions the rest is used to encode floating-point instructions), they are depicted in figure 1.1.

Here we describe only fixed point instruction formats:

- I-type format specifies two registers and a 16-bit constant. This is the standard layout for instructions with an immediate operand (an immediate operand is always sign-extended).

- J-type format is used for control-transfer instructions. Here a larger 26-bit immediate operand is used.

- R-type format defines two source and one destination register operand. Also it provides a 5-bit constant which is used to specify a shift amount (for a shift instruction) or to address a special purpose register.

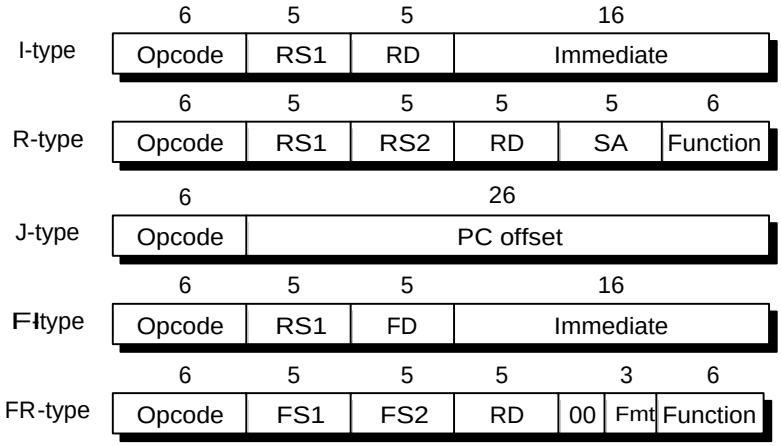

Figure 1.1: The VAMP instruction formats

Except for shifts, immediate constants are always sign-extended.

In the tables below the following notation is used:

- ✓ $sext(imm)$ is a sign-extended version of the immediate constant imm.
- ✓ GPR and SPR denote general purpose and special purpose register files.
- ✓ M denotes the main memory.
- ✓ $(c?a : b)$ denotes the selection operator (i.e. if c then a else b).
- ✓ $R[a : b]$ - the bits with indices from a to b of the register R.
- ✓ RD is the destination register (abbreviation for $GPR[RD]$), in the same way $RS1$ and $RS2$ stand for the first and the second source operand respectively.
- ✓ SA denotes a shift amount for immediate shift operations (5-bit constant) or index of a special purpose register.

Unless it is stated explicitly, each instruction increments the value of PC by four. I-type (Immediate), R-type (Register) and J-type (Jump) instructions are listed in tables 1.2, 1.3 and 1.4 respectively.

Mnemonic	d	Syntax	Operation
Load/store operations $ea = RS1 + sxt(imm)$			
lb	1	lb RD, RS1(imm)	RD $\leftarrow sext(M[ea + d - 1 : ea])$
lh	2	lh RD, RS1(imm)	RD $\leftarrow sext(M[ea + d - 1 : ea])$
lw	4	lw RD, RS1(imm)	RD $\leftarrow M[ea + d - 1 : ea]$
lbu	1	lbu RD, RS1(imm)	RD $\leftarrow 0^{24}M[ea + d - 1 : ea]$
lhu	2	lhu RD, RS1(imm)	RD $\leftarrow 0^{16}M[ea + d - 1 : ea]$
sb	1	sb RS1(imm), RD	$M[ea + d - 1 : ea] \leftarrow$ RD[7:0]
sh	2	sh RS1(imm), RD	$M[ea + d - 1 : ea] \leftarrow$ RD[15:0]
sw	4	sw RS1(imm), RD	$M[ea + d - 1 : ea] \leftarrow$ RD
Arithmetic/logical operations			
addi		addi RD,RS1,imm	RD $\leftarrow$ RS1 + sext(imm) (no overflow)
addio		addio RD,RS1,imm	RD $\leftarrow$ RS1 + sext(imm)
subi		subi RD,RS1,imm	RD $\leftarrow$ RS1 $-$ sext(imm) (no overflow)
subio		subio RD,RS1,imm	RD $\leftarrow$ RS1 $-$ sext(imm)
andi		andi RD,RS1,imm	RD $\leftarrow$ RS1 $\wedge$ sext(imm)
ori		subi RD,RS1,imm	RD $\leftarrow$ RS1 $\vee$ sext(imm)
xori		subi RD,RS1,imm	RD $\leftarrow$ RS1 $\oplus$ sext(imm)
lhgi		lhgi RD,imm	RD $\leftarrow$ imm $\ll$ 16
Test & set operations			
clri		clri RD	RD $\leftarrow$ (false ? 1 : 0)
sgri		sgri RD,RS1,imm	RD $\leftarrow$ (RS1 $>$ sext(imm) ? 1 : 0)
seqi		seqi RD,RS1,imm	RD $\leftarrow$ (RS1 $=$ sext(imm) ? 1 : 0)
sgei		sgei RD,RS1,imm	RD $\leftarrow$ (RS1 $\geq$ sext(imm) ? 1 : 0)
slsi		slsi RD,RS1,imm	RD $\leftarrow$ (RS1 $<$ sext(imm) ? 1 : 0)
snei		snei RD,RS1,imm	RD $\leftarrow$ (RS1 $\neq$ sext(imm) ? 1 : 0)
slei		slei RD,RS1,imm	RD $\leftarrow$ (RS1 $\leq$ sext(imm) ? 1 : 0)
seti		seti RD	RD $\leftarrow$ (true ? 1 : 0)
Control operations			
beqz		beqz RS1,imm	PC $\leftarrow$ PC + 4 + (RS1 $=$ 0 ? imm : 0)
bnez		bnez RS1,imm	PC $\leftarrow$ PC + 4 + (RS1 $\neq$ 0 ? imm : 0)
jr		jr RS1	PC $\leftarrow$ RS1
jalr		jalr RS1	R31 $\leftarrow$ PC + 8; PC $\leftarrow$ RS1

Table 1.2: I-type instruction layout

Mnemonic	Syntax	Operation
Shift operations		
slli	slli RD,RS1,SA	RD $\leftarrow$ RS1 $\ll$ SA
slai	slai RD,RS1,SA	RD $\leftarrow$ RS1 $\ll$ SA (arithmetic)
srli	srli RD,RS1,SA	RD $\leftarrow$ RS1 $\gg$ SA
srai	srai RD,RS1,SA	RD $\leftarrow$ RS1 $\gg$ SA (arithmetic)
sll	sll RD,RS1,RS2	RD $\leftarrow$ RS1 $\ll$ RS2[4:0]
sla	sla RD,RS1,RS2	RD $\leftarrow$ RS1 $\ll$ RS2[4:0] (arithmetic)
srl	srl RD,RS1,RS2	RD $\leftarrow$ RS1 $\gg$ RS2[4:0]
sra	sra RD,RS1,RS2	RD $\leftarrow$ RS1 $\gg$ RS2[4:0] (arithmetic)
Data transfer operations		
movs2i	movs2i RD,SPR[SA]	RD $\leftarrow$ SPR[SA]
movi2s	movi2s SPR[SA],RS1	SPR[SA] $\leftarrow$ RS1
Arithmetic/Logical Operation operations		
add	add RD,RS1,RS2	RD $\leftarrow$ RS1 + RS2 (no overflow)
addo	addo RD,RS1,RS2	RD $\leftarrow$ RS1 + RS2
sub	sub RD,RS1,RS2	RD $\leftarrow$ RS1 $-$ RS2 (no overflow)
subo	subo RD,RS1,RS2	RD $\leftarrow$ RS1 $-$ RS2
and	and RD,RS1,RS2	RD $\leftarrow$ RS1 $\wedge$ RS2
or	sub RD,RS1,RS2	RD $\leftarrow$ RS1 $\vee$ RS2
xor	sub RD,RS1,RS2	RD $\leftarrow$ RS1 $\oplus$ RS2
lhg	lhg RD,RS1	RD $\leftarrow$ RS1[15:0]0^{16}
Test & Set Operations		
clr	clr RD	RD $\leftarrow$ (false ? 1 : 0)
sgr	sgr RD,RS1,RS2	RD $\leftarrow$ (RS1 $>$ RS2 ? 1 : 0)
seq	seq RD,RS1,RS2	RD $\leftarrow$ (RS1 $=$ RS2 ? 1 : 0)
sge	sge RD,RS1,RS2	RD $\leftarrow$ (RS1 $\geq$ RS2 ? 1 : 0)
sls	sls RD,RS1,RS2	RD $\leftarrow$ (RS1 $<$ RS2 ? 1 : 0)
sne	sne RD,RS1,RS2	RD $\leftarrow$ (RS1 $\neq$ RS2 ? 1 : 0)
sle	sle RD,RS1,RS2	RD $\leftarrow$ (RS1 $\leq$ RS2 ? 1 : 0)
set	set RD	RD $\leftarrow$ (true ? 1 : 0)

Table 1.3: R-type instruction layout

Mnemonic	Syntax	Operation
Control operations		
j	j imm	PC $\leftarrow$ PC + 4 + imm
jal	jal imm	R31 $\leftarrow$ PC + 8; PC $\leftarrow$ PC + 4 + imm
trap	trap imm	trap $\leftarrow$ 1; EDATA $\leftarrow$ imm
rfe	rfe	SR $\leftarrow$ ESR; PC' $\leftarrow$ EPC; DPC $\leftarrow$ EDPC

Table 1.4: J-type instruction layout

1.3 Interrupts

The VAMP supports nested interrupts. Interrupts are maskable and precise. Floating-point interrupts are accumulated in the lower five bits of the special purpose register IEEEf (IEEE flag) as specified by the IEEE standard.

Table 1.5 shows the supported interrupts as described in [2]. The *internal* interrupts are generated by the CPU or the memory system, the *external* interrupts are generated by external I/O devices. An interrupt is called *maskable* if it can be ignored under software control, otherwise *non maskable*. The interrupted program execution can be resumed in three different ways: repeat an interrupted instruction (in this case the corresponding interrupt is of type *repeat*), continue with the instruction which would follow an interrupted one in the uninterrupted program execution (*continue* interrupt), abort the program execution (*abort* interrupt).

The special purpose registers for the interrupt mechanism are the status register SR, the exception PC register EPC, the exception DPC register $EDPC$, the exception cause register ECA, the exception status register ESR, and the exception data register $EData$.

The interrupts are triggered by the activation of event signals, denoted by $ev[j]$, $j = 0, 1, \ldots$. In the implementation, there is a nonvisible register CA called cause register, which catches these event signals. The interrupt masks are stored in the status register SR. For a maskable interrupt j, bit $SR[j]$ stores the mask of interrupt j. Masking means that interrupt j is disabled (masked) if $SR[j] = 0$, and it is unmasked otherwise. Masked interrupt signals are stored into the masked cause register MCA (nonvisible).

If at least one bit of MCA is set, the signal $JISR$ (jump to interrupt service routine) is caused. Activation of the signal $JISR$ causes the jump to the interrupt service routine. In this case, the values of the registers PC', DPC, CA and SR are stored in the corresponding registers. The register $EData$ then stores the immediate constant given in the instruction in case of a `trap` instruction, and an address of a memory access if a pagefault occurred. The return from an interrupt is performed by `rfe` instruction (return from exception), all saved parameters are restored (see table 1.4).

Since proving the absence of interrupts for assembler programs is the goal of this thesis we consider them in detail in the next chapters.

Index	Name	Type	Description	Maskable	External
0	$reset$	abort	reset	no	yes
1	ill	abort	illegal instruction	no	no
2	mal	abort	misaligned memory access	no	no
3	pff	repeat	page fault during fetch	no	no
4	$pfls$	repeat	page fault during load/store	no	no
5	$trap$	continue	trap instruction	no	no
6	ovf	continue	fixed point overflow	yes	no
7	OVF	continue	floating point (FP) overflow	yes	no
8	UNF	continue	floating point (FP) underflow	yes	no
9	INX	continue	floating point inexact result	yes	no
10	$DIVZ$	continue	floating point division by zero	yes	no
11	INV	continue	floating point invalid operation	yes	no
12	$UNIMP$	conitnue	floating point unimplemented	no	no
j $>$ 12	$eev[j]$	continue	external I/O	yes	yes

Table 1.5: The VAMP interrupts

	VAMP	Assembler Machine
Interrupts	yes	no
Address Translation	yes	no
Data Type	bitvectors	integers
Instruction memory	unified	separated
Instruction set	full	reduced

Table 2.6: Differences between the VAMP and the assembler machine

2 VAMP and Assembler Machine

In this chapter we consider the differences between the VAMP and the abstract software machine, and give the formal specifications for both machines. At the end, we give a detailed overview of the VAMP interrupts and formulate the conditions under which the execution of a program on the VAMP is equivalent to execution of this program on the abstract software machine.

2.1 Differences between the VAMP and the Assembler Machine

The interrupts supported by the VAMP processor were discussed in the previous chapter. The full list of these interrupts is given in table 1.5. The assembler machine specification does not support interrupts. Thus, the use of the assembler machine for the verification of assembler programs is only possible if we prove the absence of interrupts during the execution of a given program.

The VAMP processor contains a memory management unit to support virtual memory. Instructions and data are stored in one unified memory and can be allocated in the main memory as well as in the swap memory (e.g. on the hard disk). The memory management unit can cause exceptions. In particular, if a page accessed by a user program is not in the main memory, it produces a page fault exception. Then, the page fault handler which is part of the operating system software loads this page into the main memory and the execution of the interrupted instruction is repeated. In the assembler machine we have no address translation, i.e., all instructions and all data are allocated in the uniform virtual memory and there is no paging.

Moreover, in the assembler machine specification we distinguish two separate memories: instruction memory and data memory. The part of the main memory which contains the program instructions is supposed to be read-only and corresponds to the instruction memory of the assembler machine. The remaining memory space which is used for program data corresponds to the data memory of the assembler machine.

For the VAMP specification all registers and memory are specified by *bitvectors*, i.e. the content of register $GPR[i]$ is a bit string $GPR[i] = GPR[i]_{31} \dots GPR[i]_0$. But this is not so convenient for software verification which is mostly concerned with the numerical values stored in a register file or a memory cell. Therefore, the assembler machine operates on integers.

To keep the correspondence between the assembler machine and the VAMP, integer values in the assembler machine are strictly bounded. So, a 32-bit string in the VAMP specification can take 2^{32} different unsigned values with numerical representation in $\{0, 1, \dots, 2^{32} - 1\}$.

The main differences between the VAMP and the assembler machine are summarised in table 2.6.

2.2 The Assembler Machine Specification

2.2.1 Instruction set

The full instruction set of the VAMP processor is given in tables 1.2, 1.3 and 1.4. In the assembler machine only a subset of these instructions is realized. The current instruction set of the assembler machine is shown in table 2.7. In the assembler machine specification all computations are done modulo 2^{32}. The effective address is computed as $ea = \lfloor (RS1 + imm \bmod 2^{32})/4 \rfloor$ and the addressed memory cell is $mem = DM[ea]$, where DM denotes the data memory. Note that, the assembler specification supports the delayed PC mechanism described in [9]. Thus, we have two program counters: PC' and DPC. Instructions are fetched from the instruction memory address pointed to by DPC. Unless this is stated explicitly, each instruction increments the value of DPC by four.

2.2.2 Configuration

A mathematical machine is a triple $M = (C, c^0, \delta)$ which consists of the following components:

- C is the set of all possible configurations of M. An element of $c \in C$ is called configuration or state of the machine;
- $c^0 \in C$ is the initial configuration of M;
- $\delta : C \to C$ is the transition (step) function of M. It maps a configuration c^T to its successor c^{T+1}.

Mnemonic	Syntax	Operation
$mem = DM[\lfloor (RS1 + imm \mod 2^{32})/4 \rfloor]$		
lw	lw RD,RS1,imm	RD $\leftarrow$ mem
sw	sw RD,RS1,imm	mem $\leftarrow$ RD
movs2i	movs2i RD,SPR[SA]	RD $\leftarrow$ SPR[SA]
movi2s	movi2s SPR[SA],RS1	SPR[SA] $\leftarrow$ RS1
add	add RD,RS1,RS2	RD $\leftarrow$ RS1 + RS2
sub	sub RD,RS1,RS2	RD $\leftarrow$ RS1 $-$ RS2
slli	slli RD,RS1,SA	RD $\leftarrow$ RS1 $\ll$ SA
srli	srli RD,RS1,SA	RD $\leftarrow$ RS1 $\gg$ SA
addi	addi RD,RS1,imm	RD $\leftarrow$ RS1 + imm
lhgi	lhgi RD,imm	RD $\leftarrow$ imm $\ll$ 16
andi	andi RD,RS1,imm	RD $\leftarrow$ RS1 $\wedge$ imm
ori	ori RD,RS1,imm	RD $\leftarrow$ RS1 $\vee$ imm
xori	xori RD,RS1,imm	RD $\leftarrow$ RS1 $\oplus$ imm
sne	sne RD,RS1,RS2	RD $\leftarrow$ (RS1 $\neq$ RS2 ? 1 : 0)
sgt	sgt RD,RS1,RS2	RD $\leftarrow$ (RS1 $>$ RS2 ? 1 : 0)
sge	sge RD,RS1,RS2	RD $\leftarrow$ (RS1 $=$ RS2 ? 1 : 0)
beqz	beqz RS1,imm	PC $\leftarrow$ PC + 4 + (RS1 $=$ 0 ? imm : 0)
bnez	bnez RS1,imm	PC $\leftarrow$ PC + 4 + (RS1 $\neq$ 0 ? imm : 0)
jal	jal imm	R31 $\leftarrow$ PC + 8; PC $\leftarrow$ PC + 4 + imm
jr	jr RS1	PC $\leftarrow$ RS1
nop	nop	

Table 2.7: Instruction set of the assembler machine

A sequence $(c^0, c^1, \ldots, c^n)$ of configurations is called a computation of M iff $c^{T+1} = \delta(c^T)$ holds for all $0 \leq T < n$.

A configuration of the assembler machine is a pair including the processor configuration together with the data memory: $(proc, dm)$. We denote all possible configurations of the software machine by $Conf$.

The processor configuration is defined as a tuple: $proc = (gpr, spr, dpc, pc')$ where dpc denotes the value of DPC register and pc' denotes the value of PC' register.

We define the general purpose register file GPR and the special purpose register file SPR as a function mapping the set of register numbers $\{0, 1, \ldots, 31\}$ into the set $\{0, 1, \ldots, 2^{32} - 1\}$ of feasible register contents:

$$gpr : \{0, 1, \ldots, 31\} \rightarrow \{0, 1, \ldots, 2^{32} - 1\}$$
$$spr : \{0, 1, \ldots, 31\} \rightarrow \{0, 1, \ldots, 2^{32} - 1\}$$

We consider two separate memories, the data memory and the instruction memory. The data memory maps the data memory address space into the values representable by machine words and the instruction memory maps the instruction memory address

space into an assembler instruction (of type Asm) from table 2.7. The address space of both memories is $\{0, 1, \ldots, 2^{30} - 1\}$, i.e., the data and the instruction memory may store up to 2^{30} words (that is 2^{32} bytes) or instructions. Thus, the formal definition of these memories has the following form:

$$dm : \{0, 1, \ldots, 2^{30} - 1\} \to \{0, 1, \ldots, 2^{32} - 1\}$$
$$im : \{0, 1, \ldots, 2^{30} - 1\} \to Asm$$

The instruction memory im is considered fixed and therefore is not included in the configuration C of the assembler machine. We denote the set of all possible configurations by $Conf$.

The transition function of the assembler machine takes an instruction memory and a machine configuration $C \in Conf$ and returns the configuration of the machine after execution of the current instruction $C' \in Conf$ from a given instruction memory:

$$\mathsf{step} : im \times Conf \to Conf$$

To execute a sequence of n instructions the function $\mathsf{comp\_rec} : im \times Conf \times \mathbb{N} \to Conf$ is used which applies the step function n times:

$$\mathsf{comp\_rec}(im, C, 0) = C$$
$$\mathsf{comp\_rec}(im, C, n) = \mathsf{step}(im, \mathsf{comp\_rec}(im, C, n - 1))$$

2.3 The VAMP specification

The configuration of the VAMP is a 5-tuple $(GPR, SPR, DPC, PC', mem)$, which contains the following components (we do not consider the floating point registers here):

- $GPR : \{0, 1\}^5 \to \{0, 1\}^{32}$ represents the contents of the general purpose register file;
- $SPR : \{0, 1\}^5 \to \{0, 1\}^{32}$ represents the contents of the special purpose register file (the registers are indexed as described in table 1.1);
- $DPC \in \{0, 1\}^{32}$ is the value of the delayed PC;
- $PC' \in \{0, 1\}^{32}$ is the value of the PC';

- $mem : \{0,1\}^{29} \to \{0,1\}^{64}$ represents the contents of the main memory. The VAMP has a 64-bit wide memory with 29-bit wide addressing.

The initial configuration is the configuration with $DPC = 0$, $PC' = 4$, and $ECA = 1$. The values of the other special purpose registers are equal to zero, and the values of the general purpose registers and of the main memory are undefined. We abbreviate all possible configurations of the VAMP by dlx_conft.

The transition function of the VAMP takes the current configuration of the machine and returns the configuration of the machine after executing the current instruction (i.e. instruction which is stored in the main memory at the address pointed to by DPC):

$$\mathsf{dlx\_step} : dlx\_conft \to dlx\_conft$$

To compute the configuration of the machine after n steps the function $\mathsf{dlx\_conf} : dlx\_conft \times \mathbb{N} \to dlx\_conft$ is used:

$$\mathsf{dlx\_conf}(C, 0) = C$$
$$\mathsf{dlx\_conf}(C, n) = \mathsf{dlx\_step}(\mathsf{dlx\_conf}(C, n-1))$$

2.4 Conditions for absence of interrupts

In this section we formulate the conditions for absence of each type of interrupt [11] separately. Using them, we can show the absence of the interrupt for one step of the execution, separately for each type.

In this section the following notation will be used:

- $R(Conf)$ denotes the component R of the configuration $Conf$;
- $c\_vamp \in dlx\_conft$ denotes a configuration of the hardware machine;
- $c\_asm \in Conf$ denotes a configuration of the software machine;
- $asm \in Asm$ denotes an arbitrary assembler instruction;
- for a bitvector $a = a_{n-1} \dots a_0 \in \{0,1\}^n$ we denote by

$$\langle a \rangle = \sum_{i=0}^{n-1} a_i \cdot 2^i$$

the natural number with binary representation a.

All conditions for the absence of interrupts are formulated in the following way: we assume that in the current step the configurations of both machines are equivalent and an interrupt e does not occur, then we can conclude that interrupt e does not occur during the execution of the current step. Besides that, we assume, that the hardware machine is always in the system mode, i.e. $MODE(c\_vamp) = 0$. In the system mode the address translation is not performed (i.e. the machine operates always with the physical addresses) and the programs can access the special purpose register file, which is not allowed in the user mode.

We need to specify the equivalence of configurations of two machines ($c\_vamp \equiv c\_asm$). Configurations of the hardware and the software machine are equivalent, iff:

- The register contents of the both machines are equivalent, namely: $R_A = \langle R_V \rangle$ for all $R \in \{GPR(0), \ldots, GPR(31), SPR(0), \ldots, SPR(31), PC', DPC\}$, here R_A denotes a register of the software machine and $\langle R_V \rangle$ denotes a register of the VAMP.

- The contents of the instruction memories for both machines are equivalent, this can be expressed by the following predicate, it takes a configuration of the VAMP and an instruction memory of the assembler machine and returns true if the code regions of both machines are equivalent:

$$\begin{aligned}\mathsf{code\_region\_equiv}(c\_vamp, im) = \forall r \in \{0,1\}^{29} :& \\ \mathsf{code\_reg\_start} \leq r \leq \mathsf{code\_reg\_end} \Rightarrow& \\ \mathsf{decode}((mem(c\_vamp)(r))[31:0]) = im(2 \cdot \langle r \rangle) \wedge& \\ \mathsf{decode}((mem(c\_vamp)(r))[63:32]) = im(2 \cdot \langle r \rangle + 1)&\end{aligned}$$

  Function *decode* maps a bitvector into an assembler instruction, code_reg_start and code_reg_end specify the boundaries for the code region of the assembler machine.

- The contents of the data memories for both machines are equivalent:

$$\begin{aligned}\forall r \in \{0,1\}^{29} : \mathsf{data\_reg\_start} \leq r \leq \mathsf{data\_reg\_end} \Rightarrow& \\ (\langle mem(c\_vamp)(r))[31:0] \rangle) = dm(c\_asm)(2 \cdot \langle r \rangle) \wedge& \\ (\langle mem(c\_vamp)(r))[63:32] \rangle) = dm(c\_asm)(2 \cdot \langle r \rangle + 1)&\end{aligned}$$

  Here data_reg_start and data_reg_end denote the boundaries of the data memory of the software machine.

2.4.1 *reset* interrupt

For the absence of this interrupt we do not need any additional conditions, since it is tied to zero in the VAMP formal specification (we assume, that no hardware reset occurs during the program computation):

$$CA(c\_vamp)(0) = 0$$

2.4.2 *ill* interrupt

Since we assume that the hardware machine is always in the system mode, thus the *ill* interrupt occurs only if the VAMP fetches an illegal instruction.

$$\mathsf{decode}(iw) = asm \Rightarrow \neg \mathsf{I\_illegal}(iw)$$

where the function decode is defined above and I_illegal is the predicate over bitvectors, which is true iff the instruction word does not represent a valid VAMP instruction, and iw is current instruction word.

2.4.3 *imal* interrupt

This interrupt occurs during the instruction fetch if the content of the DPC register is not aligned on word boundary, i.e. we can define the aligned predicate for $x \in \mathbb{Z}$:

$$aligned(x) \Leftrightarrow x \bmod 4 = 0$$

For the absence of this interrupt in the next step of computation it is enough that contents of PC' register is aligned in the current configuration:

$$c\_asm \equiv c\_vamp \wedge aligned(PC'(c\_asm)) \Rightarrow \neg imal(\mathsf{dlx\_step}(c\_vamp))$$

The content of the PC' register in the current step is computed depending on the instruction executed in the previous step:

$$PC'_k = \begin{cases} imm_{k-1} & \text{if } I_{k-1} \text{ is j or jal} \\ GPR_{k-1}[RS1_{k-1}] & \text{if } I_{k-1} \text{ is jr} \\ PC'_{k-1} + imm_{k-1} & \text{if } I_{k-1} \text{ is a taken branch} \\ PC'_{k-1} + 4 & \text{otherwise} \end{cases}$$

The conditions below state that if the PC' is aligned in the current step then it will be aligned in the next step. Here $asm = im(DPC(c\_asm))$ denotes an assembler instruction pointed to by DPC in the current configuration c_asm. The predicate

$instr?(asm)$ for $asm \in Asm$ is used to recognise the instruction type. For the execution of a sequential instruction:

$$\neg(beqz?(asm) \vee bnez?(asm) \vee jal?(asm) \vee jr(asm)) \wedge$$
$$aligned(PC'(c\_asm)) \Rightarrow aligned(PC'(\mathsf{step}(im, c\_asm)))$$

For `beqz`, `bnez` or `jal` instructions:

$$(beqz?(asm) \vee bnez?(asm) \vee jal?(asm)) \wedge$$
$$aligned(PC'(c\_asm) + imm(asm)) \Rightarrow aligned(PC'(\mathsf{step}(im, c\_asm)))$$

For `jr` instruction:

$$jr?(asm) \wedge aligned(GPR[RS1(asm)]) \Rightarrow aligned(PC'(step(im, c\_asm)))$$

2.4.4 *dmal* interrupt

This interrupt occurs on load or store operations, if the effective address of a memory access is not aligned. The effective address of the read/write access is computed as: $ea = GPR[RS1] + imm$. Now we can formulate the absence of this interrupt for one step of the computation.

For `lw` or `sw` instructions:

$$c\_asm \equiv c\_vamp \wedge (lw?(asm) \vee sw?(asm)) \wedge$$
$$aligned(GPR[RS1(asm)] + imm(asm)) \Rightarrow \neg dmal(c\_vamp)$$

For all other instructions (they never cause *dmal* interrupt):

$$c\_asm \equiv c\_vamp \wedge \neg(lw?(asm) \vee sw?(asm)) \Rightarrow \neg dmal(c\_vamp)$$

2.4.5 *pff* and *pfls* interrupts

The event signals for these interrupts are named ipf and dpf respectively. These interrupts can occur only in user mode, but the machine always is in the system mode, so the absence of them is trivial:

$$ipf(c\_vamp) = 0$$
$$dpf(c\_vamp) = 0$$

2.4.6 *trap* interrupt

This interrupt occurs during the execution of a `trap` instruction. Since this instruction is not present in the instruction set of the software machine, this interrupt cannot occur:

$$trap(c\_vamp) = 0$$

2.4.7 ovf interrupt

This interrupt is generated by integer additions and subtractions which signal overflows (see tables 1.2 and 1.3). Since these are not present in the instruction set of the software machine, this interrupt cannot occur:

$$ovf(c\_vamp) = 0$$

2.4.8 Floating point and external interrupts

Since the instruction set of the software machine does not include floating point instructions, and there are no external devices for the VAMP at the moment hence the absence of these interrupts follows immediately:

$$\forall 7 \leq i \leq 31 : CA(c\_vamp)(i) = 0$$

2.5 Simulation theorem

Now we can introduce the *simulation theorem*. Let $c\_v \in dlx\_conft$ and $c\_a \in Conf$ be initial configurations for the VAMP and the assembler machine respectively, $c\_vamp(i) = \mathsf{dlx\_conf}(c\_v, i)$, $c\_asm(i) = \mathsf{comp\_rec}(im, c\_a, i)$, and init be an initialization condition over the configurations of the assembler machine, which contains initial values for the PC and DPC and any other conditions, which the program needs to be run correctly. Thus we have:

$$\begin{aligned}\forall k \in \mathbb{N} :&(\forall j < k : \neg JISR(c\_vamp(j)))\wedge \\ &c\_v \equiv c\_a \wedge \mathsf{code\_region\_equiv}(c\_v, im) \wedge \mathsf{init}(code)(c\_a) \Rightarrow \\ &\forall i \leq k : c\_vamp(i) \equiv c\_asm(i) \wedge \mathsf{code\_region\_equiv}(c\_vamp(i), im)\end{aligned}$$

The theorem has the following meaning: for any number of steps k, if the computation of a program starts with equivalent configurations and code regions, and if there were no interrupts in all of the steps $\{0, \ldots, k-1\}$, then the configurations and code regions of both machines in all of the steps $\{0, \ldots, k\}$ will also be equivalent.

The absence of interrupts in step k can be shown separately for each interrupt. Each of the following lemmas for some interrupt e assumes the absence of all interrupts in the steps before k (exactly the assumption of the simulation theorem) and conditions for the absence of this interrupt and states that this interrupt e does not occur in step k under these assumptions. Let $asm = im(DPC(c\_asm(k))/4)$ be the current assembler instruction of the configuration of the software machine in step k.

reset interrupt:

$$\forall k \in \mathbb{N} : (\forall j < k : \neg JISR(c\_vamp(j))) \Rightarrow \neg CA(c\_vamp(k))(0)$$

ill interrupt:

$$\forall k \in \mathbb{N} : (\forall j < k : \neg JISR(c\_vamp(j))) \Rightarrow \neg illegal(c\_vamp(k))$$

pff and $pfls$ interrupts:

$$\forall k \in \mathbb{N} : (\forall j < k : \neg JISR(c\_vamp(j))) \Rightarrow \neg ipf(c\_vamp(k))$$
$$\forall k \in \mathbb{N} : (\forall j < k : \neg JISR(c\_vamp(j))) \Rightarrow \neg dpf(c\_vamp(k))$$

ovf interrupt:

$$\forall k \in \mathbb{N} : (\forall j < k : \neg JISR(c\_vamp(j))) \Rightarrow \neg ovf(c\_vamp(k))$$

Floating point and external interrupts:

$$\forall k \in \mathbb{N} : (\forall j < k : \neg JISR(c\_vamp(j))) \Rightarrow \forall 7 \leq e \leq 31 : \neg CA(c\_vamp(k))(e)$$

imal interrupt:

$$\begin{aligned}\forall k \in \mathbb{N} :&(\forall j < k : \neg JISR(c\_vamp(j)))\wedge \\ &aligned(PC'(c\_a)) \wedge aligned(DPC(c\_a))\wedge \\ &((beqz?(asm) \vee bnez?(asm) \vee jal?(asm)) \Rightarrow aligned(imm(asm)))\wedge \\ &(jr?(asm) \Rightarrow aligned(GPR(c\_asm(k)[RS1(asm)]))) \Rightarrow \neg imal(c\_vamp(k))\end{aligned}$$

dmal interrupt:

$$\begin{aligned}\forall k \in \mathbb{N} :&(\forall j < k : \neg JISR(c\_vamp(j))) \wedge [(lw?(asm) \vee sw?(asm)) \Rightarrow \\ &aligned(GPR(c\_asm(k)[RS1(asm)]) + imm(asm))] \Rightarrow \neg dmal(c\_vamp(k))\end{aligned}$$

Besides that, we need to specify two additional conditions, namely: there is no branch or jump instruction with destination outside the code region and there is no memory access with address outside the data region. We denote (code_reg_start;code_reg_end) and (data_reg_start;data_reg_end) as the boundaries of the code and data regions of the assembler machine.

No jump or branch instruction with destination outside the code region:

$$\begin{aligned}\forall k \in \mathbb{N} :&\{[(beqz?(asm) \vee bnez?(asm) \vee jal?(asm)) \Rightarrow \\ &\text{code\_reg\_start} \leq imm(asm) \leq \text{code\_reg\_end}]\wedge \\ &[jr?(asm) \Rightarrow \text{code\_reg\_start} \leq GPR(c\_asm(k)[RS1(asm)]) \leq \text{code\_reg\_end}]\}\end{aligned}$$

No memory access with address outside the data region:

$$\begin{aligned}\forall k \in \mathbb{N} :&[(lw?(asm) \vee sw?(asm)) \Rightarrow \\ &\text{data\_reg\_start} \leq (GPR(c\_asm(k)[RS1(asm)]) + imm(asm)) \leq \text{data\_reg\_end}]\end{aligned}$$

3 Program verification

[1]In this chapter we shortly describe the concept of the verification diagrams. Programs verification is used to prove that a design or product under consideration possesses certain properties. The properties to be verified can be quiet elementary, e.g. a system will never reach a situation in which no further progress can be made. These properties are obtained from the system's specification. The specification defines what the system has to do and what not. The system is called "correct" if it satisfies all the required properties from its specification.

There are two different formal methods concerning program verification. Their aim is to establish system correctness with mathematical rigour. The first is a *model-based* technique. Model-based techniques are based on models describing the possible system behaviour in a mathematical precise and unambiguous manner. The system models are accompanied by algorithms that systematically explore all states of the system model. This provides the basis for a whole range of verification techniques ranging from an exhaustive exploration (model checking) to experiments with a restrictive set of scenarios in the model (simulation), or in reality (testing).

With deductive methods, the correctness of systems is determined by precisely formulated mathematical properties, i.e. the correctness of a system is established with respect to a certain formal specification or property, using formal methods. The properties are proved using tools such as theorem provers and proof checkers. In this work the deductive verification technique is used in order to prove the properties for absence of interrupts.

3.1 Transition systems

Let V be a set of typed variables of a program (data variables and labels). A *state* is an interpretation over V. Let Σ be the set of all states. Every program can be described as a set of states with the transitions between them, Transitions express relations between the variables in different states. Such a representation is called a *transition system* [3].

A transition system is a tuple $\Phi = \langle V, \Theta, \mathcal{T} \rangle$, where:

- $\mathcal{V}$ is an finite set of variables (all variables that are used in program including data variables and labels).
- Θ is the initial assertion (an assertion that states the initial conditions of the system). It identifies initial states of the system.

[1]for convenience see [8]

- $\mathcal{T}$ is a (finite) set of transitions. A transition $\tau \in \mathcal{T}$ is a function $\tau : \Sigma \longmapsto 2^{\Sigma}$, i.e. each transition is a function from states to set of states.

Let $s \in \Sigma$ be a some state. Each state in τ(s) is called a τ-successor of s. A transition τ is called *enabled* on s if $\tau(s) \neq \{\}$, a transition is called *disabled* on s if $\tau(s) = \{\}$. Each transition τ is represented by a transition relation $\rho_\tau(s, s')$ - an assertion that expresses the relation between the values of V in s and the values of V' in any τ-successor s'. Here, x' denotes the value of a variable $x \in V$ in the next state (i.e. after taking a transition that is enabled in the current state). Predicate $pres(P)$, where $P \subseteq V$, denotes that the values of all variables in P are preserved by a transition. We implicitly assume that there is an idling transition τ_I which is always enabled (an idling transition does nothing and it is used in case if there is no other transition enabled in the current state to prevent a program from being stuck).

Example:

local x: integer where $x \geq 0$
l_0: while $x < 5$
l_1: $x := x + 1$
l_2:

$\Phi = \langle V, \Theta, \mathcal{T} \rangle$, where:
$V = \{l_0, l_1, l_2\} \cup \{x\}$, $\Theta = l_0 \wedge x \geq 0$, $\mathcal{T} = \{\tau_0, \tau_1, \tau_I\}$,
$\rho_{\tau_0} = l_0 \wedge \neg l_0' \wedge ((x < 5 \wedge l_1' \wedge l_2' = l_2) \vee (x \geq 5 \wedge l_2' \wedge l_1' = l_1)) \wedge pres(\{x\})$
$\rho_{\tau_1} = l_1 \wedge \neg l_1' \wedge l_0' \wedge x' = x + 1 \wedge pres(V - \{l_0, l_1\})$
$\rho_{\tau_I} = pres(V)$ (idling transition)

A computation of a transition system Φ is an infinite sequence of states s.t. the first state satisfies Θ and any two consecutive states satisfy a ρ_τ for some $\tau \in \mathcal{T}$. We will use linear-time temporal logic (LTL) as specification language (for details see [7]). The syntax of temporal formulas is described below.

An assertion (atomic formula) over the set of variables V is a predicate which represents the relations between program variables (i.e. $x+1 > y$). A temporal formula is constructed as follows:

$$\phi ::= p \text{ (atomic formula)} \mid \neg\phi \mid \phi \vee \phi \mid X\phi \mid \phi U \phi$$

A temporal formula $X\phi$ (next operator) is valid in the current state if ϕ holds in the next state. $\phi U \psi$ (until operator) is valid in the current state of a computation P if ψ holds in a some state s further in this computation and ϕ holds in all states between the current state and s.

A temporal formula ϕ is valid for a computation P if it holds in the first state of P. A transition system Φ satisfies a temporal formula ϕ if all of its computations satisfy ϕ. We call such a formula ϕ *P-valid* for a program P.

A temporal formula ϕ is said to be *P-valid* for a program P if it holds in the first position of every computation of P.

To formulate verification conditions we use the following notation:
$\{\phi\}\tau\{\psi\}$ stands for $\rho_\tau \wedge \phi \rightarrow \psi'$
A property q is called an invariant for system P ($P \models \Box q$) iff

$$P \models \Theta \rightarrow q$$
$$\{q\}\tau\{q\} \text{ for all } \tau \in \mathcal{T}$$

3.2 Verification diagrams

Verification diagrams [8] are a succinct and intuitive way of representing proofs that programs satisfy a given temporal property. The method of proof by verification diagram, called diagram verification, is based on the representation of programs by transition systems and on the representation of the specification by a temporal formula.

For a given program P and a temporal formula ϕ, diagram verification is accomplished by constructing a verification diagram Ψ and showing that Ψ faithfully represents all computations of the corresponding transition system Φ, means that it satisfies ϕ.

There are different types of diagrams depending on the types of properties we want to prove. Since the conditions for absence of interrupts are safety properties, we describe here only *invariance* diagrams which are used to establish the correctness of safety properties.

A verification diagram is a directed labeled graph constructed as follows:

- nodes in the graph are labeled by assertions (ϕ_j)
- edges in the graph represent transitions between assertions. Each edge departs from one assertion, connects to another and is labeled by the name of a transition

An invariance verification diagram does not have terminal nodes. The assertions labelling nodes in a diagram are intended to represent the intermediate assertions (ϕ_j) appearing in a proof of a given safety property. A τ-labeled edge connecting node ϕ_i to node ϕ_j implies that it is possible for a ϕ_i-state to have a τ-successor satisfying ϕ_j.

For a node ϕ_j and transition τ connecting ϕ_i to ϕ_j we say that ϕ_j is a τ-successor of ϕ_i. If a node labeled ϕ has τ-successors $\phi_1, ...\phi_k$ than we associate with ϕ and τ the

verification condition:

$$\{\phi\}\tau\{\phi \vee \phi_1 \vee ... \vee \phi_k\}$$

The case that a transition τ does not label any edges departing from ϕ is covered by $k = 0$. That is, we associate $\{\phi\}\tau\{\phi\}$ with such a transition.

An invariance diagram is said to be valid over a program P (P-valid) if all the verification conditions associated with nodes of the diagram are P-state valid (i.e. hold in all states which are accessible by a program).

A P-valid invariance diagram establishes that

$$\bigvee_{j=1}^{m} \phi_j \rightarrow \Box(\bigvee_{j=1}^{m} \phi_j)$$

is P-valid. If in addition

$$\bigvee_{j=1}^{m} \phi_j \rightarrow q \text{ and } \Theta \rightarrow \bigvee_{j=1}^{m} \phi_j$$

are P-valid then assertion q is an invariance for a program P (i.e. $P \models \Box q$).

The properties formulated in chapter 2 are supposed to be invariants for an assembler program, thus to prove them we build a corresponding invariance diagram, but at first we need to construct an appropriate *Control Flow Graph* for a program (a transition system), which is discussed in the next chapter.

4 Control flow graphs

4.1 Overview

A control flow graph (CFG) is a universal data structure to represent sequential program code where the code is subdivided into basic blocks (nodes of the graph) connected with edges representing the flow of a control in program. *Basic blocks* are maximal-length sequences of straight-line code of a program and hence they can only have one control-transfer instruction at the end.

A CFG is a very useful abstraction since it abstracts away from sequential code, it enables graph-based transformations and allows them to be expressed without reference to a linear code layout. Besides that, it facilitates checking the validity of the given properties, since the basic blocks of a CFG can be associated with the nodes of the verification diagrams, as explained in section 3. This approach is used in this work in order to prove the absence of interrupts for a given assembler program.

First we introduce the CFG construction algorithm as well as parsing some special instructions, afterwards we describe the syntax of the CFG files produced by the DLX assembler parser and at the end we present a control flow graph for a simple assembler program.

4.2 CFG construction

Aforementioned, a CFG can be viewed as an abstract representation of a program's code written in any given programming language. The major difference between the constructing a CFG for a program written in a high-level programming language and in assembler, is that in most cases, the CFG for a high-level program can be built by analyzing solely the static structure of the program (we assume that program doesn't contain in-line assembly code). So, only in rare cases, the resulting graph depends on a certain program execution.

However, assembler code is hardware dependent and we have to take into account the specifics of the hardware, that is reflected in assembler instructions' semantics. Therefore, we come to main two features that complicate CFG construction:

- *branch-to-register* instructions which hold a target address in a register (as opposed to an explicit or immediate constant) introduce a level of uncertainty that can add spurious edges to the CFG. For most cases, one is able to predict the value being loaded to the register when the branch instruction is encountered,

i.e. frequently branch-to-register instructions realize jumps relative to the jump-table or they serve as return-from-procedure instructions. Nevertheless, in some cases, it is unable to determine the branch targets without executing the program (when the destination register is computed according to a nontrivial algorithm). If this happens, the compiler must add an edge from the block containing the branch to every reachable block. Naively, this set contains all blocks.

- *branch delay slots* complicate finding the first and the last operation in each block. Moreover, it is not prohibited to have instructions pointing to delay slot of another branch instruction, which necessitates duplication of delay slot instructions in basic blocks, and makes the resulting CFG more complicated.

Below, we introduce an algorithm that solves both problems. The algorithm consists of the following steps:

1. Build the CFG taking into account the presence of delay slots after branch-or-jump instructions. In the first step, only branch-or-jump instructions with destinations given by immediate operands are handled. For each procedure call, store the set of addresses from which it can be called, i.e. for each `jal` instruction store the destination and the return address in a special list.

2. For each branch-to-register instruction, determine whether it performs a jump relative to the jump-table.

3. Check whether a branch-to-register instruction represents a return-from-procedure instruction using the list obtained in the first step.

4. The remaining indirect branches– the ones that did not obtain the proper destinations in the previous steps– are treated during the program execution.

4.2.1 The first step of the algorithm

The basic CFG construction algorithm (the first step) is sketched in figure 4.2. Here, **split** denotes splitting the node of CFG depending on the type of the control-transfer instruction.

Splitting nodes of a CFG is depicted in figure 4.3. Here, `delay` denotes a delay slot instruction of a branch-or-jump instruction, while `tail` stands for the rest part of the CFG node. There are for different cases of splitting. Below we describe all of them in detail.

```
Q: queue of CfgNode;
// the resulting CFG is stored as a list of its nodes
CFG: list of CfgNode;
// for each procedure call stores the starting node of procedure
// and the node after the procedure call we need to return to
CallList: list of CfgNode * CfgNode;
// collects all CFG nodes with indirect branch instructions at the end
Unresolved: list of CfgNode;
cfgnode: CfgNode;
let L be a list of instructions obtained from the parser;
cfgnode.list = L; // initially all instructions are held in a single block
CFG.add(cfgnode); // add the first node to CFG
Q.push_back(cfgnode);
while not Q.empty() do
  cfgnode = Q.pop_front(); // get the topmost node from the queue
  let i be the first control-transfer instruction occurring
    in the instruction list associated with cfgnode
  split cfgnode at instruction i+1
  let cfgnode' be the remainder of cfgnode
// if cfgnode has not already been split at instruction i+1 then we add its remainder to the queue
  if successfully_split then
    Q.push_back(cfgnode');
    CFG.push_back(cfgnode');
  endif
  // add the node containing jump-to-register instruction to the special list
  if i.type() == jump_to_register then
    Unresolved.add(cfgnode); continue;
  endif
  let target be a node instruction i refers to
// split the target node according to destination address of the instruction i
  split target at i.target_instruction
  let target' be the remainder of target
  // add the starting node of the procedure and the return address to CallList
  if i.type() == call then CallList.add(target',cfgnode');
// add a node to the queue only in case it has not been split before
  if successfully_split then
    Q.push_back(target');
    CFG.push_back(target');
  endif
  add an edge from cfgnode to target'
// in case of a branch instruction
  if i.conditional_jump() then // add an edge corresponding to the untaken branch
    add an edge from cfgnode to cfgnode'
  endif
enddo
```

Figure 4.2: First step of the CFG construction algorithm

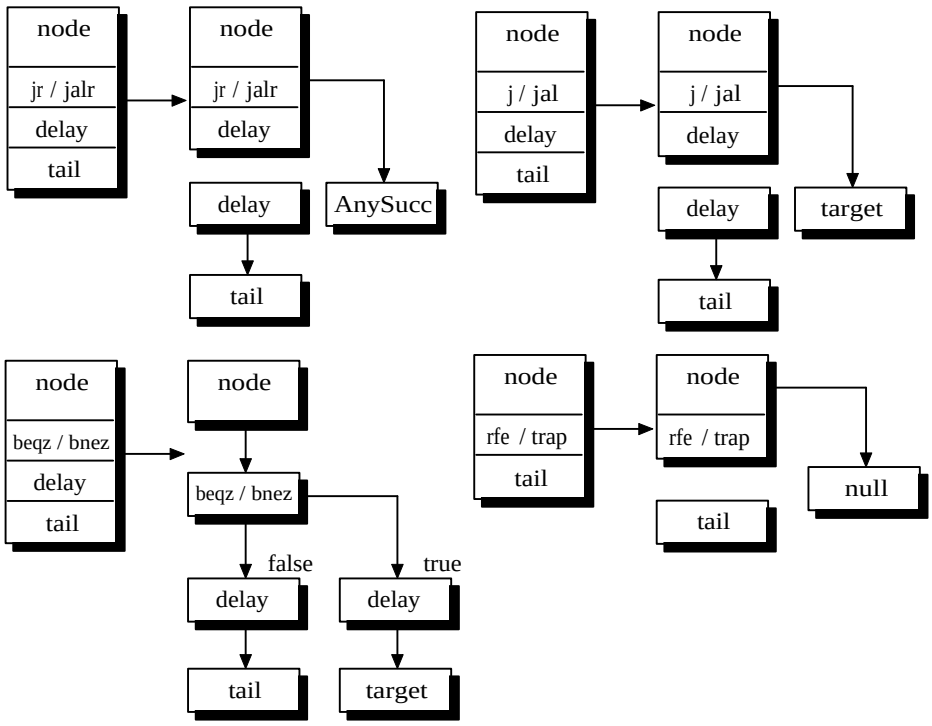

Figure 4.3: Splitting CFG nodes depending on the type of the control-transfer instruction

1. `j` and `jal`. In this case we split a node containing a jump instruction after the delay slot and add an edge to the location pointed to by the immediate operand.

2. `beqz` and `bnez`. Here, the original node is split twice: just before and after the jump instruction. This is necessary, since according to the syntax of CFG files (see section 4.3) these instructions should be placed into isolated nodes.

3. `jr` and `jalr`. For these instructions we don't know the proper successors in advance. Therefore, during the parsing phase, they are set to be pointing to the temporary node *AnySucc*.

4. concerning `rfe` and `trap` instructions, the block containing one of them is supposed to have no successors (`null` in figure 4.3). The behaviour of these instructions strongly depends on the concrete realization of the interrupt mechanism, and it is not feasible to determine the proper destination address knowing the program text only.

Note that, in figure 4.3, a delay slot for branch instructions is duplicated for both branches. This is required, because branch instructions that branch to delay slots of another branch or jump instructions are not prohibited. If that's the case, the corresponding delay slot instruction (the one we jump to) should be placed to the

untaken branch, such that the preceding branch instruction – whose delay slot we jump to – cannot influence the program execution. But for the taken branch the delay slot instruction should be executed *as well*. This is the reason for duplicating the delay slot instruction for both branches.

For the remaining control-transfer instructions (`j`, `jal`, `jr` and `jalr`) the delay slot should also be duplicated due to the same reason. However, in this case the CFG syntax does not require to put the jump instructions into isolated nodes, that is why a jump instruction with its corresponding delay slot instruction reside in the same block. In figure 4.3, one can see that after splitting a block with a jump instruction, an *isolated node* containing the delay slot instruction is inserted, which points to the remaining part of the CFG (**tail** in figure). If some control-transfer instruction points to the delay slot of a jump instruction, we add an edge to this isolated node. In this case, the preceding jump instruction cannot influence the program execution.

4.2.2 The second step of the algorithm

After performing the first step of the algorithm we have a set of nodes, where each node represents the maximal length sequence of a straight-line code.

The next step concerns with the recognition of the jump-tables. Each jump-table includes a starting address and a set of entries to which the jump-table refers. Jump-tables are recognised during the parsing phase; typically a jump-table is a label in the data region which points to a set of program locations. The actual jump relative to a jump-table is performed as follows: first, an entry from a jump-table is loaded into the destination register and then jump to a location addressed by this register is performed.

To determine, whether a branch-to-register instruction is a jump-table jump we do the following: once an instruction that loads some entry from a jump-table is encountered, the corresponding register is marked to be pointing to an entry of the jump-table. Afterwards, if there is a `jr` or `jalr` instruction with this register as a destination and the value of this register has not been changed between these instructions, we add edges from the node containing `jr` or `jalr` instruction to all locations comprised in this jump-table. Note that, we add edges exactly to *all* locations contained in the jump-table, since at that moment we only know that a branch-to-register instruction *refers* to the jump-table. The exact destination will be known on the execution phase only. That is why, it is possible to jump to any location stored in the jump-table.

Normally, load-from-memory instruction and branch-to-register instruction are

located closely to each other in the program. That is why, we look for the corresponding load-from-memory instruction only in the node containing branch-to-register instruction. If this is not the case, the destination of such a branch-to-register instruction is determined during the program execution, i.e. in the fourth step of the algorithm.

4.2.3 The third step of the algorithm

In the third step of the algorithm, we determine whether a jump-to-register instruction serves as a return-from-procedure instruction. In DLX assembler the procedure calls are realized with `jal` or `jalr` instructions, and the return address is saved into register `r31`. Recall that, in the first step of the algorithm we stored the destination and the return address for each `jal` instruction in the special list.

The algorithm is realized as follows: for each procedure found in a program - for each entry of the list mentioned above - we traverse the corresponding part of CFG keeping track of the register `r31`. Finally, if a jump-to-register instruction has been encountered and its destination register holds the value that has been loaded into register `r31` at the start of the procedure, then we add edges from the block containing that jump-to-register instruction to all locations this procedure is called from. Note that, one procedure may have more than one return instruction. Therefore, we need to explore all branches from the start of the procedure to a jump-to-register instruction or until we reach a node with no successors. In the latter case the corresponding procedure does not have a return instruction.

4.2.4 The fourth step of the algorithm

The fourth step of the algorithm concerns with the program execution. This means, that we traverse the resulting CFG with all the registers and memory cells being initialized according to the configuration file (a configuration file is passed as a command line parameter to the Parser program). The traversal starts in the initial node of the CFG, and proceeds until a node with no successors is encountered. During this execution phase of the algorithm, all blocks containing yet unresolved branch-to-register instructions obtain the proper successors. These successors are determined according to the current values of the target registers. After performing the fourth step of the algorithm, we ensure that the CFG does not contain any basic blocks with unknown successors, i.e. pointing to the *AnySucc* temporary node.

4.2.5 Construction of an example CFG

The following example shows the transformation applied to a CFG in each step of the algorithm. In the first step, only immediate jump and branch instructions are processed (Figure 4.4). At first, all instructions are contained in a single CFG node, then we subdivide the CFG after instructions in marked lines.

First, we subdivide the CFG node after the delay slot instruction `add r3,r0,r31` and add an edge to the target instruction `addi r1,r0,0`. The delay slot instruction is duplicated and set to be pointing to the remaining part of the CFG – to the instruction `jr r3`. The created isolated node, having no predecessors, is redundant for CFG structure. But, as mentioned before, if some instructions points to the delay slot of a jump instruction, we add an edge leading to this node, such that the preceding instruction `j loc2` does not take affect on the program execution.

Then, the CFG node is subdivided after instruction `add r6,r1,r1`, but, in this case the target instruction is unknown. The delay slot instruction is also duplicated, and the created CFG node, containing this instruction, is set to be pointing to the sequentially next instruction `addi r1,r0,0`. All further subdivisions proceed analogously, only the branch instruction `beqz r0,.` branches to itself. The list of unresolved jump-to-register instructions will contain all CFG nodes with jump-to-register instructions. Namely, *Unresolved* list contains three CFG nodes with `jr r3`, `jalr r2` and `jr r31` instructions.

In the second step, for scan each CFG node in *Unresolved* list, looking for references to jump-tables. In our case, the register `r2` in line 8 refers to the jump table. That is why, we insert a selection statement, which leads from the delay slot instruction `addi r3,r3,4` either to location `loc1`, `loc2`, or `loc3` depending on the value loaded to the register `r2` (see Figure 4.5). Notice that, the labels `loc1` and `loc3` points to the *delay slots*.

Instruction `jalr r2` performs jump to a subroutine. Thus, we add information about the procedure calls to the *CallList*, and remove this instruction from the *Unresolved* list. In particular, this list contains tuples consisting of the starting and the return address for each procedure: $\mathit{CallList} = \{(3,11),(6,11),(12,11)\}$. Here, the list contains line numbers of the program except real addresses.

In the third step of the algorithm we traverse the corresponding parts of the CFG starting from lines 3, 6 and 11 respectively. Instructions in lines 4 and 14 are the return statements, we add edges from lines 5 and 15 to line number 11 (the return address). Instructions `jr r3` and `jr r31` are removed from the *Unresolved* list and the algorithm terminates.

```
1   .text 0
2             j loc2
3   loc1:     add r3,r0,r31
4             jr r3
5             add r6,r1,r1
6   loc2:     addi r1,r0,0
7             slli r1,r1,1
8             lw r2,jtbl(r1)
9             jalr r2
10            addi r3,r3,4
11            beqz r0,.
12  loc3:     set r4
13            sub r4,r0,r4
14            jr r31
15            clr r3
16  .data .
17  jtbl:
18            .word loc1
19            .word loc2
20            .word loc3
```

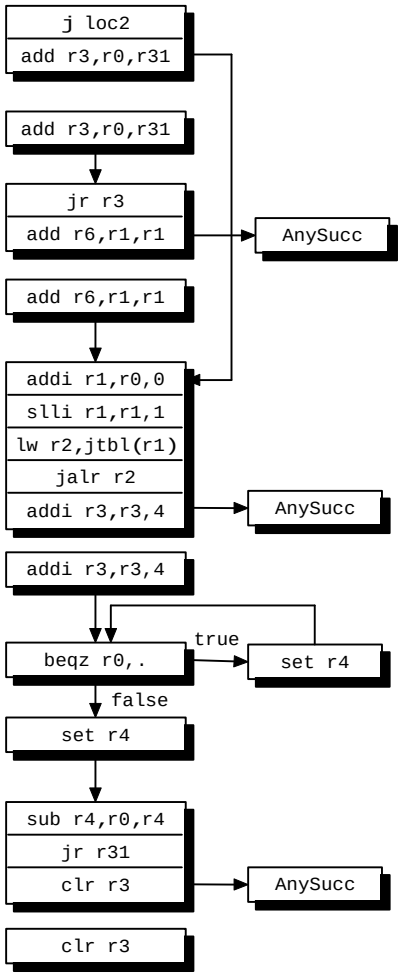

Figure 4.4: The first step of the CFG construction algorithm

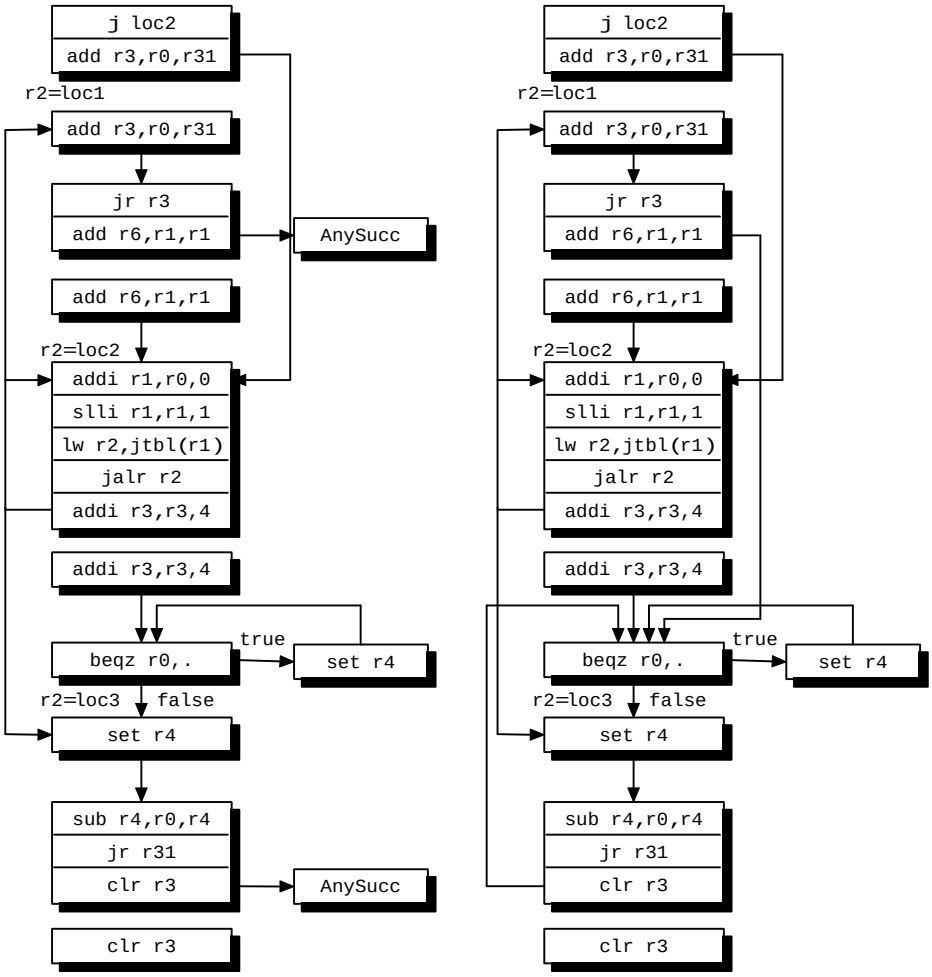

Figure 4.5: The second and the third steps of the CFG construction algorithm

4.3 The syntax of CFG files

In this section we describe the general syntax of CFG files generated by the Parser program and accepted by the refinement tool[1] performing the formal proof of the required properties. The CFG files are used both for describing control flow graphs and verification diagrams. Some syntactic constructs are used only for CFGs or verification diagrams respectively. Unless it is stated explicitly, the syntactic construct is common both for CFGs and verification diagrams.

The CFG files have several sections each of which have the same general form (all syntactic components given in square brackets are optional):

```
((<section_name>) (<property_1>) (<property_2>) ...  (<property_N>))
```

The sections, in the order they should be declared, are as follows:

[1]https://react.cs.uni-sb.de/software/tv-tool

1. Declaration of global variables:

   This section contains the declarations of all global variables occurring in the CFG for a given program. The form of this section is as follows:

   ```
   ((<decl_1>) (<decl_2>) ...  (<decl_N>))
   <decl_i>:  <type> <variable_name> [<array_size>]
   ```

   Each declaration gives the type and name of a global variable. The declaration of array types, using C syntax is also allowed. The array size can be specified either as an immediate constant, as a variable or as an expression. In the latter case, the expression should be enclosed in parentheses.

   Example: `((int r0) (int r1) (int M[(data_reg_end-data_reg_start+4)]))`

2. Initial conditions for global variables (only for CFGs):

   This section describes initial conditions applied to global variables, the syntax is: `(init(<pred>))`, where `<pred>` is a predicate expression, its syntax is described below.

3. Declaration of functions:

   The global variables section is followed by one or more function declarations. Each declaration has the following general form:

   ```
   (<local_vars> [(<initialnodes_decl>)] (<BB_1>) (<BB_2>) ...  (<BB_N>))
   ```

Below we describe the purpose of each section as well as it specific semantics and syntax:

`<local_vars>` : declares the function name and its return type as well as the types and the names of the variables that are declared within this function's scope (used only to describe CFGs). The syntax of this component is:

```
(<func_name> <ret_type>) [<decl_1> ...  <decl_N>]
```

`<decl_i>` declares a local variable. The syntax is the same both for local variables declarations as well as for global ones.

`<initialnodes_decl>` : declares the set of initial nodes of the verification diagram. This parameter is only used to describe the verification diagrams, and its syntax is: `initialnodes <nodes>`, where `<nodes>` is a space-separated list of initial nodes.

`<BB_i>` : these sections define the basic blocks in a function. The syntax of a basic block definition is: `<block_id> [init] (<successors> [<invariant>] <statements>)` The syntax and semantics of the components in a `BB` section are as follows:

`<block_id>` : an integer that allows to enumerate the basic blocks. The `<block_id>` component should be *unique* for each basic block.

`init` : this optional declaration denotes that this basic block is the initial node of the function (only for CFGs).

`<successors>` : this declaration defines the successors of this basic block in the control flow graph. The syntax for this component is: `succ <succ_1> <succ_2> ... <succ_N>`

Here, `<succ_i>`'s are the id's of the block's successors.

`<invariant>` : this declaration defines a predicate expression in the node of a verification diagram. The syntax is: `inv(<pred>)`, where `pred` is a predicate expression.

`<statements>` : the declaration of the statements has the following syntactic form (CFGs only):

`(<statement_1>) (<statement_2>) ... (<statement_N>)`. Each statement is either a predicate expression for a branching statement or an assignment over the set of local or global variables.

Note that, a CFG node is allowed to have 0, 1 or 2 successors. For the verification diagram the number of successor nodes is not limited. If a node has 0 or 1 successors then it must contain an ordinary statement, whereas a node with two successors represents a *branching* statement. The body of such a node contains a predicate expression. The first branch — the first successor of a node — is taken if the corresponding predicate evaluates to *true*, otherwise the second branch is taken.

Predicates and expressions are declared with the following syntax:

```
<expr> ::= <expr> + <expr> | <expr> - <expr> | <expr> * <expr> | <expr> / <expr>
| <expr> % <expr> | <variable> | numeric_constant
<pred> ::= <expr> > <expr> | <expr> < <expr> | <expr> >= <expr> | <expr> <= <expr> | <expr>
== <expr> | <expr> != <expr> | <pred> && <pred> | <pred> || <pred> | <pred> & <pred> | <pred>
| <pred> | <pred> => <pred> | !<pred>
```

The semantics of these operations is the same as in C-language, except one, namely "`=>`" denotes the logical implication.

4.4 Example control flow graph

Now we present the control flow graph for the program above (see figure 4.4) generated by the Parser program. First, we need to describe a common structure for the control flow graphs generated by the Parser program:

- All the registers and the memory region are declared as global variables, namely:
  - `r0, r1, ..., r31` represent general purpose registers;
  - `sr, esr, ...` represent special purpose registers (see page 8);
  - the array `M[]` denotes the data memory region;
  - `code_reg_start, code_reg_end` and `data_reg_start, data_reg_end` denote the boundaries of the code and data regions respectively (for details see chapter 2);
- In the `init` section we assign the initial values to all registers used in a program. The whole memory region is filled with zeros unless explicitly otherwise specified for certain memory cells.
- The node number 1 denotes an *error state*, its purpose is described in chapter 5.

The control flow graph is depicted in Figure 4.6.

```
( (int r0) (int r1)
(int r2) (int r3)
(int r4) (int r5)
(int r6) (int r31)
(int M[260])
(int code_reg_start)
(int code_reg_end)
(int data_reg_start)
(int data_reg_end) )

(init
((r0==0)&(r1==42)&
(r2==0)&(r3==0)&(r4==0)&
(r5==0)&(r6==0)&
(r31==0)&(M[72]==4)&
(M[76]==16)&(M[80]==48)&
(FORALL(i) (((64<=i)&
(i<260)&(i!=72)&(i!=76)&
(i!=80))=>(M[i]==0)))&
(code_reg_start==0)&
(code_reg_end==64)&
(data_reg_start==64)&
(data_reg_end==260)&
(pc0==0)) )

( (main int)
(0 init (succ 4)
(r3 = (r0 + r31)) )

(2 (succ 3)
(r3 = (r0 + r31)) )

(3 (succ 16)
(r6 = (r1 + r1)) )

(16 (succ 7 16)
(r3 == 44) )

(5 (succ 4)
(r6 = (r1 + r1)) )

(4 (succ 17)
(r1 = (r0 + 0))
(r1 = (r1 * 1))
(r2 = M[(r1 + 72)]) )

(12 (succ 2 13)
(r2 == 4) )

(13 (succ 4 14)
(r2 == 16) )

(14 (succ 8 14)
(r2 == 48) )

(17 (succ 12)
(r5 = (r0 + r31))
(r4 = (r4 % 2))
(r31 = 44)
(r3 = (r3 + 4)) )

(6 (succ 7)
(r3 = (r3 + 4)) )

(7 (succ 10 8)
(r0 == 0) )

(10 (succ 7)
(r4 = 1) )

(8 (succ 9)
(r4 = 1) )

(15 (succ 7 15)
(r31 == 44) )

(9 (succ 15)
(r4 = (r0 - r4))
(r3 = 0) )

(11 (succ )
(r3 = 0) )

(1 (succ 1) )

(18 (succ ) ) )
```

Figure 4.6: A sample control flow graph generated by the Parser program

5 Proving the absence of interrupts

Using the concepts described in the previous chapters, we know present the automatic method used in proving the absence of interrupts for DLX assembler programs.

For convenience, the developed software is presented in the form of two independent programs. The first program, the Parser, takes a DLX assembler file as input and outputs the corresponding CFG in the formerly described syntax of the CFG files. The second program, the VD generator, generates the required verification diagrams based on a given control flow graph. Later on, the refinement tool reads in the CFG together with the verification diagrams and establishes the verification conditions. The validity of these conditions is proved by the automatic theorem-prover *Simplify*[1].

At first, we present additional syntax to declare the assembler directives for DLX assembler files. Then, we discuss how the verification diagram nodes are associated with the nodes of CFGs and which conditions have to be proved to establish the validity of the verification diagram. Afterwards we formulate the conditions for the absence of interrupts in the form they are represented in the verification diagrams. Finally, we explain the approach to the automatic generation of verification diagrams and describe how the assembler instructions unsupported by the refinement tool are handled. In appendix we give the proof for the absence of interrupts for a simple assembler program.

5.1 DLX assembler directives

DLX assembler provides the following set of directives (a directive definition starts with a dot):

- `.word expr` - defines a word in the memory with the value given by evaluating an expression `expr`. Expressions are given in the C-syntax and may use both program labels (treated as variables) and constants.
- `.ascii str` - defines a string `str` in the memory.
- `.space expr` - allocates an empty space in the memory. In case of a code region this space is filled with `nop` instructions.
- `.set ID,expr` - defines a constant with a name `ID` and assigns to it the value of an expression `expr`.

[1] http://research.compaq.com/SRC/esc/Simplify.html

- `.data expr` - starts a data region from the address given by `expr` parameter. There are additional requirements for the data regions. Namely, the starting address must be word aligned and the regions must not overlap with each other or with the code region. All data regions must lie within the boundaries specified by the external parameters `data_reg_start` and `data_reg_end`.

- `.text expr` - indicates that the code region starts from the address given by `expr` parameter; only one consistent code region is allowed. The code region must start from the word boundary, it must not overlap with data regions. It must also lie within the boundaries specified by the external parameters `code_reg_start` and `code_reg_end`.

- `.align expr` - aligns the current PC address to the boundary given by parameter `expr`, `expr` specifies the value in *words*.

Note that all external parameters such as `code_reg_start`, `code_reg_end`, `data_reg_start`, `data_reg_end` and the initial values for all registers are specified in the `.ini` file which is passed as the command line parameter to the Parser program.

5.2 Association of CFG nodes with nodes of the verification diagram

To prove that a some property holds in a CFG node we have to associate this node with a verification diagram node. The required property is defined in a VD node in the form of a predicate expression.

Recall that, a CFG node with *one* successor represents a basic block of an assembler program (for definition of the basic blocks see section 4) containing no branch instructions – branch instructions are represented by decision nodes. In particular, a CFG node with one successor contains a set of statements, where each statement is an assignment of an expression over local or global variables. Formally, this set represents a *transition relation* $\rho_\tau(s, s')$ (for details see section 3), where s is the currently considered node and s' is its successor, i.e. in the CFG file we have: `(s (succ s') ...)`.

Now suppose that the node s is identified with a VD node containing a property ϕ, and consequently the node s' is associated with a VD node containing a property ψ. In this case, the established *verification condition* is: $\phi \wedge \rho_\tau(s, s') \rightarrow \psi$. The verification condition is *only* generated in case that the VD node ψ is a *direct successor* of the VD node ϕ in the verification diagram. As mentioned in section 4.3, the verification diagrams are defined with the same syntax as the CFGs.

For instance, consider the following part of the CFG:

```
(3 (succ 4)
(r2 = (r1 + 2))
(r2 = (r2 * 2))
(r3 = M[r2]) )
(4 (succ ...)  ...)
```

Suppose that the node 3 is associated with a property: $\phi = (r_1 \bmod 4 = 0)$ and the node 4 is associated with a property $\psi = (r_2 \bmod 4 = 0)$. Then, the established verification condition has the following form:

$$(r_1 \bmod 4 = 0) \wedge (r_2' = r_1 + 2) \wedge (r_2'' = r_2' \cdot 2) \wedge (r_3' = M[r_2'']) \Rightarrow (r_2'' \bmod 4 = 0)$$

Note that it is not necessary to associate *each* node of the CFG with a node of the verification diagram. In this case the generation of the verification conditions proceeds as follows: assume that a node s_0 is associated with a property ϕ, there is a simple path from s_0 to s_n. A simple path in the CFG is a sequence of nodes $s_0, s_1, \ldots, s_n$ where s_{i+1} is a successor of s_i for all $0 \leq i < n$, and all the nodes are different. If the node s_n is associated with a property ψ, the corresponding verification condition is formulated as follows:

$$\phi \wedge (\bigwedge_{0 \leq i < n} \rho_\tau(s_i, s_{i+1})) \Rightarrow \psi$$

It means that all statements on the path from s_0 to s_n are combined by logical conjunction, this is called a *compressed transition relation*.

Note that for all *decision nodes* – the nodes with two successors – the corresponding predicate expression is inserted to the left-hand side of the implication. Since a decision node has two successors, this implies that for the first successor node (the *true* branch) the predicate expression is inserted without any changes, whereas for the second successor node the predicate expression is *negated*.

This is illustrated with the following equations. Assume that s is associated with a property ϕ, and s_1, s_2 are associated with the properties ψ_1 and ψ_2 respectively:

```
(s (succ s1 s2)
(pred) )
```

The verification conditions in this case are generated as follows:

$$\phi \wedge pred \Rightarrow \psi_1 \text{ and } \phi \wedge \neg pred \Rightarrow \psi_2$$

In addition, there are no restrictions in associating the CFG nodes with the VD nodes. By giving a set of the CFG nodes associated with the VD nodes, the refinement tool

generates the verification conditions for all possible simple shortest paths found between the CFG nodes within this set.

The associations of the CFG nodes with the verification diagram nodes are described in the `.abs` files. An `.abs` file is passed as a parameter to the refinement tool. Each entry in this file has the following form:

```
(<CFG_ID>) => (<VD_ID_1> <VD_ID_2> ...<VD_ID_N>)
```

it denotes that a CFG node `<CFG_ID>` is associated with a set of VD nodes `<VD_ID_1> <VD_ID_2> ...<VD_ID_N>`. If a CFG node is associated with a *set* of VD nodes, the verification conditions are generated for each VD node from this set separately.

5.3 The conditions for the absence of interrupts

Recall that, in chapter 2 we describe the conditions required to state that an assembler program doesn't generate any interrupts during the execution on the VAMP. Now we need to represent these conditions in the suitable form for the diagram verification.

In this work only the absence of the following interrupts is formally proved: $ill, imal, dmal$. Bear in mind, that all the remaining conditions are formulated in the form: $CA(c\_vamp)(i) = 0$ (see chapter 2), i.e. they do not depend on the assembler source code and hence their validity is not required to be proved with the verification diagrams.

Now we consider the proof of the absence for these three types of interrupts in detail.

5.3.1 *ill* interrupt

An illegal interrupt is triggered if there is an unimplemented instruction encountered in the assembler source code. The crucial point is that the presence of an illegal instruction is revealed during the *parsing phase*. The absence of this type of interrupt is proved as follows: if during the program parsing an illegal instruction is encountered then we add an edge from a basic block containing this instruction to a *special* basic block with ID 1. This block formally represents an *error state*. The property to be satisfied is that we never reach the *error state* during the program execution. In the next section we explain how the corresponding verification diagram is generated.

5.3.2 *imal* interrupt

According to the properties given in section 2.4, the *imal* interrupt does not occur if the initial address of PC' is word aligned and for each branch or jump instruction

the destination address is also word aligned. Additionally, the destination address of each branch or jump instruction must lie within the code region. The presence of this type of interrupt is determined during the control flow graph construction. The main problem is that we *cannot* build a correct control flow graph if a branch or jump instruction has a misaligned destination address, because in this case this is not possible to determine a proper successor of a basic block containing such a branch or jump instruction. Consider the following piece of code; the numbers to the left are the current PC addresses:

```
0:   add r1,r2,r3
4:   beqz r1, .+ 4 + 3
8:   nop
12:  sub r1,r1,r2
```

Here, we cannot determine whether to jump to `nop` or to `sub` instruction, since the destination address of `beqz` instruction is misaligned. So, this piece of code cannot be subdivided into basic blocks correctly. That is why, if a branch or jump instruction with a misaligned operand is detected or this instruction points to outside the code region, then we add an edge from the basic block containing this instruction to the basic block representing the *error state* (the one with ID 1). As before, the verification diagram states that we never reach the error state during the program execution.

All control-transfer instructions with immediate operands are examined on the first step of the CFG construction algorithm (see page 30). For the remaining jump-to-register instructions we perform the required computations on the fourth step of the CFG algorithm.

5.3.3 *dmal* interrupt

For the absence of the *dmal* interrupt we have to prove that the effective address is aligned for each load or store instruction. To prove this we associate each node of the CFG containing a load or store instruction with a VD node having the following property:

$$(ea \bmod 4 = 0) \wedge (data\_reg\_start \leq ea \leq data\_reg\_end)$$

where ea denotes the effective address of the memory access, $ea = GPR[RS1] + imm$. In the next section we show how the corresponding diagram is generated.

5.4 Verification diagrams generation

In the previous section we formulated the conditions that should be placed into a verification diagram. Strictly speaking, there are two different types of diagrams to

be generated. As mentioned before, the verification diagrams are generated by the Diagram generator, which takes a control flow graph constructed by the Parser program and outputs two verification diagrams. One of them is for the absence of *ill* and *imal* interrupts; the other one for the absence of the *dmal* interrupt.

In particular, each verification diagram is described by two files. The `abs` file stores information about which CFG node corresponds to which VD node. The `vd` file defines the verification diagram itself. For the syntax of the `abs` files see section 5.2, for the `vd` files – see section 4.3.

At first, we consider construction of the VDs for *ill* and *imal* interrupts. This is a trivial case, since the presence of these interrupts is reflected in the control flow graph structure. Formally, we need to show that the program execution will never reach the *error state*. For these purposes, it is assumed that each CFG contains an implicitly defined variable `pc0`.

During the verification conditions' generation this variable holds the ID of the currently processed CFG node. This is illustrated with the following example:

```
(4 (succ 5) )
(5 (succ ...)  ...)
```

Let both of these CFG nodes be associated with the following property: $pc_0 \geq 4$. As a result, the refinement tool generates the following conditions:

$$(pc_0 = 4) \wedge (pc_0 \geq 4) \Rightarrow (pc'_0 = 5) \wedge (pc'_0 \geq 4)$$

Now it is clear, that we have to prove that $pc_0 \neq 1$ is an invariant for our transition system (CFG) in order to be able to show the absence of these interrupts. The property is called invariant for a system if it holds in each of its states. The corresponding verification diagram expressed in the syntax of the CFG files is:

```
(0 (succ 0)
inv(pc != 1) )
```

This verification diagram has a self-loop, which states that if a given property holds in the current state then it should also hold in the next state, i.e. it is an invariant for the system. During the traversal of the CFG we associate all the nodes found on the path with the diagram above. So, the required property holds, if there is no node on the path with a successor node 1.

Note that the CFG traversal means that we execute the program represented by the CFG with the initial values of all registers and memory cells defined in the `init` block of the CFG. The remaining memory region, i.e. except the memory cells defined explicitly, is supposed to be filled with *zero* values. Traversal is done until a node with

no successors is reached or an infinite loop is encountered. A loop is treated as infinite if the number of iterations spent in it exceeds the given limit (this limit is specified in the configuration file which is passed as a parameter to the Parser). This precaution has been specially made in order to prevent the program suspension. By treating loops as infinite after certain number of iterations exceeds, we sacrifice completeness of our proof. But, it is only feasible solution, since the termination is a response property, and it can only be proved with the help of *chain* verification diagrams [3], which is not part of this work.

With regard to the *dmal* interrupt, we need to construct the verification diagram corresponding to the CFG, since its presence is not reflected in the CFG structure. Recall that, the required property must be established for each memory access instruction. But this is not enough to identify each CFG node containing a memory access instruction with the corresponding VD node. In that case, it is possible to some instruction, which resides in the same CFG node after the memory access instruction to *overwrite* the register required for the effective address computation. Thus, the generated verification conditions may be violated. In order to prevent this, we subdivide each basic block containing a memory access instruction into 2 blocks. The memory access instruction with its preceding instructions will be comprised in the first block, while the subsequent instructions will belong to the latter block. After that, each CFG node will have at most one memory access instruction.

The verification diagram generation then proceeds as follows: we traverse the CFG and identify a successor for each CFG node containing a memory access instruction with a VD node which specifies the following property, here *ea* stands for the effective address of a memory access:

$$(ea \bmod 4 = 0) \wedge (data\_reg\_start \leq ea \leq data\_reg\_end)$$

It is impossible to prove the given property without knowing the information about all variables (registers) occurring in it. That is why, we extend our property with assertions regarding all registers and memory cells known in the current step. This is illustrated in the following example:

```
(4 (succ 5)
(r1 = (r1 + 3))
(r2 = (r1 * 4))
(r3 = M[(r2 + 8)]) )
(5 (succ ...)  ...)
```

Let the values of the registers and memory cells before the execution of the 4th CFG node be: $r_1 = 4, r_2 = 1, r_3 = 0, M[36] = 15$, then the property to be identified with the 5th node is:

```
inv((r1 == 7)&(r2 == 28)&(r3 == 15)&(M[36] == 15)&((r2+8)% 4 == 0)&
    (data_reg_start <= (r2+8))&((r2+8) <= data_reg_end))
```

Here, the values of registers r_1, r_2 and r_3 are changed, whereas the value of the memory cell $M[36]$ stays the same. Although, the values of r_1, r_3 and $M[36]$ are not required to prove the absence of the *dmal* interrupt in the current step, they are still presented in the VD node. Since, they possibly be required at a later time.

The VD generator associates only the nodes containing memory access instructions with the VD nodes. The remaining work of finding the shortest paths between associated CFG nodes and generating the verification conditions is left to the refinement tool.

The main problem with the generation of such conditions is concerned with *loops*. If a program has a loop, the corresponding VD must contain the invariance property which holds at each iteration of this loop. Let's say, we want to prove, that if the property ϕ holds at the beginning of the next loop:

while cond **do** body **end**

then the property ψ must hold after execution of this loop. To do that we have to establish an invariant I, such that:

$$\phi \rightarrow I$$
$$I \wedge cond \wedge (\bigwedge_{\tau \in body} \rho_\tau) \rightarrow I$$
$$I \wedge \neg cond \rightarrow \psi$$

where $\bigwedge_{\tau \in body} \rho_\tau$ represents a joint transition relation for the loop's body. The automatic generation of such invariants (especially for nested loops) is quite a tough task to accomplish [5].

So, in this work the loops are *straightforwardly unrolled* and the required conditions are generated immediately for each iteration of a loop. Remark that, for loops containing memory access instructions we specify the required condition for every memory access instruction found in the loop on each iteration. Whereas, for loops without memory access instructions, an arbitrary node within the loop's body is selected as a pivot point, and is identified with a VD node on each loop iteration. This is necessary to correctly propagate the values of all registers and memory cells after execution of this loop. Let's consider an example in Figure 5.7.

Let the values of the registers before execution be: $r_1 = 100$, $r_4 = 0$. The first loop in example (nodes 1 and 2) does not contain memory access instructions. Then,

CFG	Verification Diagram
(1 (succ 2) (r4 = (r4 + 7)) (r1 = (r1 - 1))) (2 (succ 1 3) (r1 != 0)) (3 (succ 4) (r1 = (r1 + 4)) (r4 = (r4 - 1)) (M[r1] = r4)) (4 (succ 3 5) (r4 != 0)) (5 (succ ...) ...)	(1 (succ 2) inv((r1 == 100)&(r4 == 0))) (2 (succ 3) inv((r1 == 99)&(r4 == 7))) ... (100 (succ 101) inv((r1 == 0)&(r4 == 700)))

Figure 5.7: Loop unrolling

the 1st CFG node is associated with the VD nodes on each iteration (the diagram is depicted in Figure 5.7 to the right).

The next loop (nodes 3 and 4) contains a memory access instruction. That is why, the 4th CFG node – the successor of a node with the memory access instruction – is associated with a VD node on each loop iteration:

```
(101 (succ 102)
inv((r1 == 4)&(r4 == 699)&((r1 % 4) == 0)&(data_reg_start <= r1)&
    (r1 <= data_reg_end)) )

(102 (succ 103)
inv((r1 == 8)&(r4 == 698)&((r1 % 4) == 0)&(data_reg_start <= r1)&
    (r1 <= data_reg_end)))
...
```

The drawbacks of this approach are that the generated diagrams may become very large depending on the number of loop iterations. If this is the case, the work dealing with the construction of the loop invariants and with adjustment of the verification diagrams is left to the end-user. Nevertheless, the advantage of this method is that it is fully automated and requires no further manual work. Moreover, it is feasible to unroll loops having several tens of thousands of iterations and generate the verification conditions in satisfiable amount of time. In most cases it fulfills the needs of assembler program verification.

The method presented here is applicable for most assembler programs. It is only restricted in the way, that the resulting verification diagram may be very large if a loop has a big number of iterations and contains a lot of memory access instructions, or if a program contains too deep nested loops. In the former case too many CFG nodes require to be associated with the VD nodes, in the latter case the diagram size grows *exponentially* with the nesting level. This can make a verification diagram huge and complicated for understanding.

5.5 Handling unsupported instructions

In this section we consider dealing with the assembler instructions unsupported by the refinement tool. Unfortunately, there are several DLX assembler instructions whose semantics cannot be correctly expressed in the CFGs. In this case, the refinement tool fails to generate the verification conditions. Strictly speaking, these conditions can be represented in CFG but the Simplify [4] reports a counterexample while trying to prove them. The difficulty relates to the fact that the refinement tool operates on integers, whereas the VAMP ISA uses the *bitvectors*. This forces to emulate bitwise operations with integer arithmetic, but this strategy works not in all cases.

Below we present the list of unsupported DLX instructions:

1. All shift instructions with a shift amount specified in a register (`sll`, `srl`, `sra`). These operations require multiplication or division of two variables (registers). The Simplify tool supports only multiplication when one of the operands is given by an immediate constant. Division is supported as inverse to multiplication operation, and, hence, is restricted in the same way.

2. All bitwise logical operations (`and(i)`, `or(i)`, `xor(i)`), except for the trivial cases, where the second operand is one of the immediate constants 0 or 1. In this case, at first, the operands should be converted to the bitvectors, which again requires multiplication and division of the variables.

Hopefully, the future development of the Parser program and the refinement tool will cover these operations. By now, the refinement tool supports a specific *assignment* operation, which allows to partially verify the programs containing such operations.

The uninterpreted instructions are processed as follows: once such an instruction has been encountered in a program, the Parser inserts the special '?' assignment `(r = ?)` to the appropriate place in the CFG, where `r` is a destination register of an unsupported operation. This assignment denotes that the value of the corresponding register is *unknown* in the current moment. Later on, the VD generator keeps track of all expressions where the unknown registers occur. Evidently, a register, to which the result of evaluating such an expression is assigned, also becomes unknown. If during the execution an unknown register is encountered in a predicate expression of a decision node, the execution goes to the *error state*. The execution stops, since the value of the predicate expression is unknown, and, hence, we cannot decide which successor to take.

Afterwards, during the generation of the verification diagrams, all assertions regarding the values of currently unknown registers are *omitted*. This is illustrated with the following example:

```
(1 (succ 2)
(r1 = ?)
(r2 = (r1 * 2))
(r3 = (r0 + 5))
(M[(r2 + 8)] = r3) )
(2 (succ ...)  ...)
```

Let $r_3 = 10$ before the execution of the 1st node, then the property associated with the 2nd CFG node is:

```
inv((r3 == 5)&((r2+8)% 4 == 0)&(data_reg_start <= (r2+8))&
((r2+8) <= data_reg_end))
```

This means that no assumptions are made about the values of registers `r1` and `r2`, whereas the value of the register `r3` is known. The refinement tool omits all assumptions about currently unknown registers on the left-hand side of the implication while generating the verification conditions, and, thus, any assertions about them will be invalid.

In this chapter we considered the automatic generation of verification diagrams. Only the absence of *ill*, *imal* and *dmal* interrupts requires the construction of verification diagrams. Proof for absence of *ill* and *imal* interrupts turned out to be trivial, while in proving the absence of the *dmal* interrupt we encountered difficulties regarding to necessity of loop invariants generation. The problem is solved by loop unrolling. We do not pretend to find elegant and universal solution, however it works in most cases and does not require any user interaction.

6 Conclusion

This work presents a tool for automatic proofs of the conditions for absence of interrupts. The validity of these conditions was proved before using the PVS verification system in [11].

The tool is realized in the form of two independent programs: the Parser which constructs a control flow graph from the DLX assembler source code, and the VD generator which automatically generates the verification diagrams corresponding to the required conditions. The refinement tool generates the verification conditions based on a control flow graph and a corresponding verification diagram. The validity of the verification conditions is checked using the Simplify theorem prover.

The results of this work can be used in the VAMP software verification. It allows to verify assembler programs independently on the hardware. Once we have proved that an assembler program satisfies the conditions for the absence of interrupts, this program can be then correctly simulated on the assembler machine.

The future work in this field might be oriented in the direction of the automatic generation of loop invariants which would allow us to build more compact and readable verification diagrams, and extending the set of supported assembler instructions to cover all instructions from the DLX ISA. Moreover, the range of properties automatically proved can be extended such that other types of properties can be handled. These can, among others, include, for example, program termination, which requires an introduction of another type of verification diagrams. This would hopefully simplify and speedup the verification of assembler programs.

Bibliography

[1] S. Beyer, C. Jacobi, D. Kröning, D. Leinenbach, and W. J. Paul. *Instantiating uninterpreted functional units and memory system: functional verification of the VAMP processor*. Accepted to 12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME), 2003.

[2] Sven Beyer. *Putting It All Together: Formal Verification of the VAMP*. PhD thesis, Saarland University, Computer Science Department, 2004.

[3] I. A. Browne, Z. Manna, and H. D. Sipma. Generalized temporal verification diagrams. Technical report, Computer Science Department, Stanford University, Stanford, 1995.

[4] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. *Journal of the ACM*, 52, 2005.

[5] M. Ernst. *Dynamically Detecting Likely Program Invariants*. PhD thesis, University of Washington, Department of Computer Science and Engineering, August 2000.

[6] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, second edition, 1996.

[7] Joost-Pieter Katoen. Principles of model checking. Formal Methods and Tools Groups. University of Twente. *Lecture notes in Computer Science*, 2002.

[8] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems. Safety*. Springer-Verlag, 1995.

[9] S. M. Müller and W. J. Paul. *Computer Architecture: Complexity and Correctness*. Springer-Verlag, 2000.

[10] S. Owre, Shankar N., and Rushby J. M. *A prototype verification system*. Springer-Verlag, 1992.

[11] O. Parshin. Formal simulation of machine instructions with interrupts by assembler instructions. Master's thesis, Saarland University, Computer Science Department, 2004.

A Appendix A

A.1 Example Proof

As an example we consider the program which calculates the square root of an integer. Since the DLX ISA does not support multiplication, we need to emulate it using the school multiplication method. The program operates according to the following simple algorithm:

$$N = 0;\ X \geq 0;\ \textbf{while}\ N^2 \leq X\ \textbf{do}\ N = N + 1;\ \textbf{end};\ N = N - 1;$$

The result of this program satisfies: $N^2 \leq X < (N+1)^2$ where $X \geq 0$, i.e. $N = \lfloor \sqrt{X} \rfloor$ is an integer square root of X. In order to show the abilities of the developed software we supply the program with the additional features, namely:

- the multiplication algorithm is realized as a single subroutine;
- the multiplication procedure is called from the main program using the jump-table, the result is returned in a memory cell;
- the intermediate results of the program (namely N^2) are stored in the designated memory region;
- the resulting integer square root is stored in a memory cell.

This program uses two nested loops, which would make loop invariants generation quite complicated, moreover the invariant for the outer loop is supposed to be non-linear, since the termination condition $N^2 \leq X$ is non-linear. The proof for the absence of interrupts for this program proceeds fully automatically and requires no additional work.

At first we present the original source code of the example program. The initial values for all registers and external parser parameters (such as the boundaries of the memory regions) are given in the configuration file (Figure A.8) which is passed as a command line parameter along with the assembler source code file (Table A.8) to the Parser program.

```
(code_reg_start==500) ; the code region start-end addresses
(code_reg_end==676)
(data_reg_start==676) ; the data region start-end addresses
(data_reg_end==4788)
; the initial values for all registers
(r0==0) (r1==0) (r2==0) (r3==0)
(r4==0) (r5==0) (r6==0) (r7==0)
(r8==0) (r9==0) (r10==0) (r11==0)
(r12==0) (r13==0) (r14==0) (r15==0) (r16==0)
(r17==0) (r18==0) (r19==0) (r20==0) (r21==0)
(r22==0) (r23==0) (r24==0) (r25==0) (r26==0)
(r27==0) (r28==0) (r29==0) (r30==0) (r31==0)
(sr==0) (esr==0) (eca==0) (epc==0) (edpc==0)
(edata==0) (rm==0) (ieeef==0) (fcc==0) (pto==0)
(ptl==0) (emode==0) (mode==0)
```

Figure A.8: The configuration file

```
.text 500 ; start of the code region
main:
    clr r4 ; initialize the loop counter (N)
square:
; pass the parameters to the multiplication subroutine
    add r24,r4,r0
    lw r3,jtable(r0)
    jalr r3 ; calculates res = r24 * r25
    add r25,r4,r0
    lw r26,res(r0) ; obtain the result of imul
; the number of which we want compute the square root
    lw r3,X(r0)
    sle r1,r26,r3 ; terminate if N^2 > X
; store the intermediate result into the memory region
    slli r2,r4,2
    sw memfill(r2),r26
    beqz r1,end
    nop
    j square
    addi r4,r4,1
imul:  ; integer multiplication procedure
; calculates r26 = r25 * r24
    beqz r25, imul_exit ; exit if one operand = 0
    clr r26
    sls r1, r25, r0 ; test whether the first operand is negative
    beqz r1, test2
    clr r2
    set r2
    sub r25, r0, r25
test2:  ; r2 will hold the sign of the result
    sls r1, r24, r0 ; test whether the second operand is negative
    beqz r1, mul
    nop
    sub r2, r1, r2
```

Table A.8:The program source code

```
    sub r24, r0, r24
mul:  ; the multiplication loop
    beqz r24, sign
; check the current bit of the factor
    andi r1, r24, 1
    beqz r1, skip
    srai r24, r24, 1
    add r26, r26, r25
skip:
    j mul
    slli r25, r25, 1
sign:  ; adjust the sign of the result
    beqz r2, imul_exit
    nop
    sub r26, r0, r26
imul_exit:
    lhgi r24,(res&0x8000?~(res>>16):res>>16)
    xori r24,r24,res&0xFFFF
; return from the procedure, save the result into memory
    jr r31
    sw 0(r24),r26
end:  ; store the resulting square root in a memory cell
    lhgi r25,(sqr&0x8000?~(sqr>>16):sqr>>16)
    xori r25,r25,sqr&0xFFFF
    subi r4,r4,1
    sw 0(r25),r4
.data .  ; the beginning of the data region
memfill:  ; an array to store intermediate results
    .space 4096
X: ; a number of which we want to compute the square root
    .word 453
res:  ; a memory cell to store the result of the imul subroutine
    .space 4
sqr:  ; a memory cell to store the resulting square root
    .space 4
jtable:  ; jump-table for the imul subroutine
    .word imul
```

Table A.8:The program source code

This program is passed as a command line parameter to the Parser program, the generated CFG is presented in Figures A.9 and A.10.

Then, the resulting CFG is passed to the VD Generator program. The VD Generator builds two verification diagrams: one for the absence of *ill* and *imal* interrupts and one for the absence of the *dmal* interrupt. As mentioned above (see Section 5.3), the absence of *ill* and *imal* interrupts are determined during the parsing phase, thus the corresponding verification diagram is trivial. The `abs` file and the verification diagram are depicted in figure A.11. Here all the nodes visited during the CFG traversal

(program execution) are identified with a node of the verification diagram.

For *dmal* interrupt the corresponding verification diagram is bit large since it requires unrolling all program loops containing the memory access instructions. Thus we present here only the beginning and the end of the diagram. The `abs` file is also described partially. Note that the `FORALL` statement on the diagram (figures A.12 and A.13) states that all memory cells, except explicitly defined ones, are initialized with zero values. From the verification diagram one can see that all the intermediate results (namely N^2) are stored into the memory region staring from the address 680, i.e. $M[680] = 1$, $M[684] = 4$, $M[688] = 9$, ... The number whose square root we want to compute is stored into the memory cell $M[4772] = 453$, the resulting integer square root is stored to the address 4780, i.e. $M[4780] = 21$, since $\lfloor\sqrt{453}\rfloor = 21$.

The CFG nodes 42, 40, 41, 47, 4 are identified with the VD nodes, since their predecessors contain the memory access instructions (namely 11 is a successor of the node 42, 14 is a successor of the node 40, 3 - successor of 41, 41 - successor of 47, and 48 is a successor of 4). For instance, the CFG node 11 contains the instruction `r3 = M[(r0 + 4784)]`, the CFG node 42 is identified with the VD node 1, this VD node contains the following condition:

```
((r0 + 4784) % 4 == 0)&(676 <= (r0 + 4784))&((r0 + 4784) < 4788)
```

i.e. it states that the address of a memory access should be word-aligned and it should lie within the boundaries given by `data_reg_start` and `data_reg_end` parameters.

The CFG nodes 42, 40, 41, 47, 4 lie in the loop. Since the loops containing memory access instructions are unrolled, these nodes are identified with different VD nodes (see the `abs` file definition in figures A.12 and A.13), i.e. we prove the absence of *dmal* interrupt on each loop iteration separately.

```
( (int r0) (int r1) (int r2)
(int r3) (int r4) (int r24)
(int r25) (int r26) (int r31)
(int M[4788]) (int code_reg_start)
(int code_reg_end)
(int data_reg_start)
(int data_reg_end) )

(init
((r0 == 0)&(r1 == 42)&(r2 == 0)&
(r3 == 0)&(r4 == 0)&(r24 == 0)&
(r25 == 0)&(r26 == 0)&(r31 == 0)&
(M[4772] == 453)&(M[4784] == 556)&
(FORALL (i) (((676 <= i)&
(i < 4788)&(i != 4772)&
(i != 4784)) => (M[i] == 0)))&
(code_reg_start == 500)&
(code_reg_end == 676)&
(data_reg_start == 676)&
(data_reg_end == 4788)&(pc0 == 0))
)

( (main int)
(0 init (succ 11)
(r4 = 0) )

(11 (succ 42)
(r24 = (r4 + r0))
(r3 = M[(r0 + 4784)]) )

(42 (succ 39)
(r31 = 520)
(r25 = (r4 + r0)) )

(39 (succ 10 39)
(r3 == 556) )

(2 (succ 3)
(r25 = (r4 + r0)) )

(3 (succ 41)
(r26 = M[(r0 + 4776)]) )

(41 (succ 47)
(r3 = M[(r0 + 4772)]) )

(47 (succ 50)
(r1 = 0) )

(50 (succ 49 48)
(r26 <= r3) )

(49 (succ 48)
(r1 = 1) )

(48 (succ 4)
(r2 = (r4 * 4))
(M[(r2 + 676)] = r26) )

(4 (succ 8 5)
(r1 == 0) )

(5 (succ 6) )

(6 (succ 11)
(r4 = (r4 + 1)) )

(9 (succ 10)
(r4 = (r4 + 1)) )

(10 (succ 15 12)
(r25 == 0) )

(12 (succ 13)
(r26 = 0) )

(13 (succ 44)
(r1 = 0) )

(44 (succ 43 16)
(r25 < r0) )

(43 (succ 16)
(r1 = 1) )

(16 (succ 20 17)
(r1 == 0) )

(17 (succ 18)
(r2 = 0) )
```

Figure A.9: The CFG for the integer square root program (1st part)

```
(18 (succ 19)
(r2 = 1)
(r25 = (r0 - r25)) )

(20 (succ 19)
(r2 = 0) )

(19 (succ 46)
(r1 = 0) )

(46 (succ 45 22)
(r24 < r0) )

(45 (succ 22)
(r1 = 1) )

(22 (succ 26 23)
(r1 == 0) )

(23 (succ 24) )

(24 (succ 25)
(r2 = (r1 - r2))
(r24 = (r0 - r24)) )

(26 (succ 25) )

(25 (succ 30 27)
(r24 == 0) )

(27 (succ 28)
(r1 = (r24 % 2)) )

(28 (succ 34 31)
(r1 == 0) )

(31 (succ 32)
(r24 = (r24 / 2)) )

(32 (succ 33)
(r26 = (r26 + r25)) )

(34 (succ 33)
(r24 = (r24 / 2)) )
```

```
(33 (succ 25)
(r25 = (r25 * 2)) )

(38 (succ 29)
(r25 = (r25 * 2)) )

(30 (succ 29)
(r1 = (r24 % 2)) )

(29 (succ 37 35)
(r2 == 0) )

(35 (succ 36) )

(36 (succ 14)
(r26 = (r0 - r26)) )

(37 (succ 14) )

(15 (succ 14)
(r26 = 0) )

(14 (succ 40)
(r24 = (r0 + 4776))
(M[(r24 + 0)] = r26) )

(40 (succ 3 40)
(r31 == 520) )

(21 (succ 7)
(M[(r24 + 0)] = r26) )

(8 (succ 7) )

(7 (succ 51)
(r25 = (r0 + 4780))
(r4 = (r4 - 1))
(M[(r25 + 0)] = r4) )

(51 (succ ) )

(1 (succ 1) ) )
```

Figure A.10: The CFG for the integer square root program (2nd part)

Verification diagram	ABS file
(check (initialnodes 0) (0 (succ 0) inv(pc0!=1)))	(0) => (0) (11) => (0) (42) => (0) (39) => (0) (10) => (0) (15) => (0) (14) => (0) (40) => (0) (3) => (0) (41) => (0) (47) => (0) (50) => (0) (49) => (0) (48) => (0) (4) => (0) (5) => (0) (6) => (0) (11) => (0) (13) => (0) (44) => (0) (16) => (0) (20) => (0) (19) => (0) (46) => (0) (22) => (0) (26) => (0) (25) => (0) (27) => (0) (28) => (0) (31) => (0) (32) => (0) (33) => (0) (25) => (0) (29) => (0) (37) => (0) (14) => (0) (33) => (0) (7) => (0) (51) => (0) (1) => (0)

Figure A.11: The verification diagram for the absence of *ill* and *imal* interrupts

Verification diagram	ABS file
(mem (initialnodes 0) (0 (succ 1) inv((r0 == 0)&(M[4772] == 453)&(M[4784] == 556)& (FORALL (i) (((676 <= i)&(i < 4788)&(i != 4772)& (i != 4784)) => (M[i] == 0))))) (1 (succ 2) inv((r0 == 0)&(r24 == 0)&(r3 == 556)&(r4 == 0)& (M[4772] == 453)&(M[4784] == 556)& (FORALL (i) (((676 <= i)&(i < 4788)&(i != 4772)& (i != 4784)) => (M[i] == 0)))&((r0 + 4784) % 4 == 0)& (676 <= (r0 + 4784))&((r0 + 4784) < 4788))) (2 (succ 3) inv((r0 == 0)&(r24 == 4776)&(r25 == 0)&(r26 == 0)& (r3 == 556)&(r31 == 520)&(r4 == 0)&(M[4772] == 453)& (M[4784]==556)&(FORALL (i) (((676 <= i)&(i < 4788)& (i != 4772)&(i != 4784)) => (M[i] == 0)))& ((r24 + 0) % 4==0)&(676 <= (r24 + 0))& ((r24 + 0) < 4788))) (3 (succ 4) inv((r0 == 0)&(r24 == 4776)&(r25 == 0)&(r26 == 0)& (r3 == 556)&(r31 == 520)&(r4 == 0)&(M[4772] == 453)& (M[4784] == 556)&(FORALL (i) (((676 <= i)&(i < 4788)& (i != 4772)&(i != 4784)) => (M[i] == 0)))& ((r0 + 4776) % 4 == 0)&(676 <= (r0 + 4776))& ((r0 + 4776) < 4788))) ..... (114 (succ 115) inv((r0 == 0)&(r1 == 0)&(r2 == 0)&(r24 == 4776)& (r25 == 704)&(r26 == 484)&(r3 == 453)&(r31 == 520)& (r4 == 22)&(M[680] == 1)&(M[684] == 4)&(M[688] == 9)& ... &(M[4776]==484)&(M[4784]==556)& (FORALL (i) (((676 <= i)&(i < 4788)&(i!=680)& ... & (i != 4784)) => (M[i] == 0)))&((r0 + 4772) % 4 == 0)& (676 <= (r0 + 4772))&((r0 + 4772) < 4788)))	(0) => (0) (42) => (1) (40) => (2) (41) => (3) (47) => (4) (4) => (5) (42) => (6) (40) => (7) (41) => (8) (47) => (9) (4) => (10) (42)=>(11) (40)=>(12) (41)=>(13) (47)=>(14) (4) =>(15) .....

Figure A.12: The verification diagram for the absence of $dmal$ interrupt (1st part)

Verification diagram	ABS file
(115 (succ 116) inv((r0 == 0)&(r1 == 0)&(r2 == 88)&(r24 == 4776)& (r25 == 704)&(r26 == 484)&(r3 == 453)&(r31 == 520)& (r4 == 22)&(M[680] == 1)&(M[684] == 4)&(M[688] == 9)& ... &(M[4776]==484)&(M[4784]==556)& (FORALL (i) (((676 <= i)&(i < 4788)&(i != 680)& ... & (i != 4784)) => (M[i] == 0)))&((r2 + 676) % 4 == 0)& (676 <= (r2 + 676))&((r2 + 676) < 4788))) (116 (succ 117) inv((r0 == 0)&(r1 == 0)&(r2 == 88)&(r24 == 4776)& (r25 == 4780)&(r26 == 484)&(r3 == 453)&(r31 == 520)& (r4 == 21)&(M[680] == 1)&(M[684] == 4)&(M[688] == 9)& ... &(M[4776] == 484)&(M[4780] == 21)&(M[4784] == 556)& (FORALL (i) (((676 <= i)&(i < 4788)&(i != 680)& ... & (i != 4780)&(i != 4784)) => (M[i] == 0)))& ((r25 + 0) % 4 == 0)&(676 < =(r25 + 0))& ((r25 + 0) < 4788))) (117 (succ) inv(TRUE)))	..... (4)=>(105) (42)=>(106) (40)=>(107) (41)=>(108) (47)=>(109) (4)=>(110) (42)=>(111) (40)=>(112) (41)=>(113) (47)=>(114) (4)=>(115) (51)=>(116)

Figure A.13: The verification diagram for the absence of $dmal$ interrupt (2nd part)