Description of the BINCOA model
Deliverable 1.1, part 2

BINCOA Project

August 2009

Contents

1 Introduction **2**

2 Overview of the model **2**

3 Formal model **3**
- 3.1 Expressions 3
- 3.2 Instructions 4
- 3.3 Memory zone properties 4
- 3.4 Annotations 4
- 3.5 Asynchronous product of programs 5

4 Operational semantics **5**
- 4.1 Environments 5
- 4.2 Expressions 6
- 4.3 Operational semantics 6

5 Methodology **7**
- 5.1 Using the model for disassembled binary code 7
- 5.2 Modelling open programs 8
- 5.3 Exceptions and interruptions 9
- 5.4 Limitations of the model 9

A Example **10**
- A.1 C source code 10
- A.2 Assembly code 10
- A.3 Explanation of the disassembled code 11
- A.4 A model for this function 11
- A.5 The node function 13

B Bitvector semantics **14**
- B.1 Bitvectors 14
- B.2 BV operators 14
- B.3 Arithmetic properties 17
- B.4 Encoding of signed operators 18

1 Introduction

This document provides a description of the formal model designed for the Binary Code Analysis (BINCOA) project.

The main ideas of the model are:

- A very small set of instructions
- Self-contained models which do not require a separate description of the memory model (like the fact of being big or little endian)
- Models look like transitions systems extended with bitvector variables and memory arrays of bytes
- The ability to define tool-specific annotations

The aim of this model is to be able to represent formally binary programs; many facilities are provided in the model for architectures which address their memory in 8-bit quantities (bytes), and store negative integers in two's complement notation. Simulating architectures outside these will probably be cumbersome, but we are not aware of many.

Main limitations. All common features of low-level programming languages are taken into account by the model, for example dynamic jumps, modification of the call stack, dynamic memory and so on. The two main limitations of the model are the following: first, the model cannot capture self-modifying code; second, the model is untimed. Moreover, modelling asynchronous interruptions is feasible, but it may lead to very large models. Finally, we do not consider floating-point instructions: verification of floating-point programs is a very challenging problem on its own, and we prefer to focus on the specific issues raised by binary code analysis.

2 Overview of the model

The formal model is a graph in which each arc contains one basic instruction. These basic instructions are assignment (**Assign**), no-operation (**Skip**), jump to a non-statically known address (**Jump**), guards to check necessary conditions for progress (**Guard**), and an instruction to handle absent code such as API calls (**External**).

Statically-known jumps are simply encoded as arcs of the graph; this means that such a jump in the original program will be translated into a **Skip** instruction. The case of conditional jumps is handled thanks to guards on the arcs: they are expressions which need to be satisfied in order for the control flow to follow this arc. Jumps which are not determined statically are encoded by a **Jump** arc which is *dangling*, i.e. which has no target vertex.

The memory model is a finite set of variables ranging over bitvectors and some finite memory, accessed as disjoint arrays of bytes (8-bit quantities).

Moreover, each control point in the model is made apparent so that assertions and annotations are easily associated with a given model.

3 Formal model

We denote by *Instr* the set of all possible instructions, and by $\mathit{Expr}(X, M)$ (resp. $\mathit{Form}(X, M)$) the set of expressions (resp. formulae) using memory arrays from the set M and variables from the set X. Expressions are detailed in the following subsection.

A program in our framework is a tuple $\langle X, M, Z, V, E, \mathit{node}\rangle$, where X is a finite set of variables, M is a finite set of memory arrays, Z is a finite set of memory zone properties, V is a finite set of vertices, $E \subseteq V \times \mathit{Instr} \times (V \cup \{\bot\})$ is a finite set of arcs; an arc has target $\bot$ if and only if its instruction is **Jump**. The partial function $\mathit{node} : \mathbb{N} \times E \to V$ gives the entry vertex where a jump to the address given as a parameter should end up. The second parameter may be used by the function in the case of programs generated by a product of programs, to detect which subprogram actually made a jump.

Each variable $x \in X$ has a size expressed in bits, denoted $|x|$. Each memory array $m \in M$ has a size expressed in bytes, also denoted $|m|$. Memory zone properties are described in subsection 3.3, page 4.

Often in this document, we will consider sets X and M respectively of variables and memory arrays. We will build the two sets X' and M' such that those four sets are mutually disjoint, and such that there is a bijection between X and X' and between M and M'. The image by these bijections of a variable x will be x' and of a memory array m will be m'.

3.1 Expressions

Expressions are used in several places: assignments, guards, assertions, and will probably be used in many annotations.

The two sets of expressions $\mathit{Expr}(X, M)$ and $\mathit{Form}(X, M)$ are defined as follows (abbreviated respectively as E and F).

$$
\begin{array}{rcl}
E & ::= & bv \mid x \mid \\
 & & -_b E \mid E +_b E \mid E -_b E \mid E \times_b E \mid \\
 & & E /_u E \mid E \%_u E \mid E /_s E \mid E \%_s E \mid \\
 & & \tilde{}E \mid E | E \mid E \& E \mid E\hat{}E \mid \\
 & & E :: E \mid \\
 & & \mathit{SignExt}(E, k) \mid \mathit{UnsignExt}(E, k) \mid \\
 & & E << E \mid E >>_u E \mid E >>_s E \mid \\
 & & m[E; \overrightarrow{k}] \mid m[E; \overleftarrow{k}] \mid \\
 & & E\{k; k\} \\
F & ::= & \neg F \mid F \vee F \mid F \wedge F \mid E = E \mid E \leq_u \mid E \leq_s E
\end{array}
$$

In this grammar, k denotes an integer constant; x is a variable in the set X; m is a memory array belonging to M.

The arithmetic operators have their usual meaning on bitvectors which is detailed in appendix. The syntax mostly follows that of the C language: oper-

ators |, &, and ^ are the bitwise or, and, and exclusive or, respectively. ~ is the bitwise not. A concatenation operator on bitvectors is provided ($::$).

3.2 Instructions

As mentioned earlier in this document, there are five types of instructions:

- **Skip** does nothing
- **Assign** *l-value* := *Expr* assigns the value of a given expression to an *l-value*. In our model, l-values are a sequence of bytes or a sequence of bits referring to some place in memory or inside a register. The value of the expression must be a bitvector with a size corresponding to the size of the l-value.
- **External** $\mathit{Form}(X \cup X', M \cup M')$ is a special kind of node used to model calls to an API, a system call, or a piece of program that was previously analysed and summarized. The semantics of such a node is given by an assertion.
- **Jump** *Expr* jumps to the address given by the value of the expression. The value of this expression should be a natural integer.
- **Guard** $\mathit{Form}(X, M)$ stops the flow of execution if the expression evaluates to false, and allows the flow of execution to continue otherwise. It does not modify any variable or memory array.

3.3 Memory zone properties

Statically-defined memory zone properties are part of the formal model because for example, the fact that a memory zone is read-only affects the semantics of the model. A memory zone property is a tuple $(m, \mathit{address}, \mathit{size}, \mathit{properties})$ where m is a memory array, *address* and *size* are natural numbers denoting the extent of the memory region within m, in bytes. The last component of a memory region is a set of properties. The following properties are available:

- *write-is-ignored* to declare read-only memory areas.
- *write-aborts* to declare areas where writing is not allowed.
- *read-aborts* to declare areas where reading is not allowed.

Obviously, *write-is-ignored* and *write-aborts* are mutually exclusive.

3.4 Annotations

Annotations are defined in our context as additional information not affecting the semantics of the model, contrary to memory zone properties or assertions. Annotations are mostly intended to help automatic analysis under user guidance and to provide a simple communication mechanism between analysis tools, for example by declaring invariants of the model. We recommend the use of the definition of expressions provided in subsection 3.1 page 3 when expressions are needed in annotations. A set of common annotations will be defined later. Moreover, each tool will be free to define its own annotations, and to support only a subset of all existing annotations.

3.5 Asynchronous product of programs

Given two programs $\langle X_1, M_1, Z_1, V_1, E_1, node_1\rangle$ and $\langle X_2, M_2, Z_2, V_2, E_2, node_2\rangle$, we define the asynchronous program of these two programs by $\langle X, M, Z, V, E, node\rangle$, with:

$$
\begin{array}{lcl}
X & = & X_1 \cup X_2 \\
M & = & M_1 \cup M_2 \\
Z & = & Z_1 \cup Z_2 \\
V & = & V_1 \times V_2 \\
E & = & \{((v_1, v_2), ins, (v_1', v_2)) | (v_1, ins, v_1') \in E_1\} \cup \\
 & & \{((v_1, v_2), ins, (v_1, v_2')) | (v_2, ins, v_2') \in E_2\} \\
node(n, ((v_1, v_2), ins, (v_1', v_2))) & = & (node_1(n, (v_1, ins, v_1')), v_2) \\
node(n, ((v_1, v_2), ins, (v_1, v_2'))) & = & (v_1, node_2(n, (v_2, ins, v_2')))
\end{array}
$$

4 Operational semantics

The operational semantics is given as a graph decorated by vertices of the original model and environments, i.e. the mapping of variables and memory bytes to values, so that following a path in this graph is like following exactly one run of the model.

Exceptional conditions for which no further progress is possible (division by zero, illegal access to memory) are "legal" in the formal model but lead to a special vertex of the semantics which is a trap from which no arc exists. This special vertex should help in practice to detect exceptional conditions, because reaching it means that an error occurred in the initial program.

The operational semantics basically specifies how a program impacts memory and variables after each basic instruction. The first notion needed is thus that of the state memory and variables are in. This is called an environment.

4.1 Environments

An environment $\rho_{X,M}$ is an application such that:

$$
\begin{cases}
\rho_{X,M}(x) \in \mathrm{BV}(|x|), & \text{if } x \in X \\
\rho_{X,M}(m)(n) \in \mathrm{BV}(8), & \text{if } m \in M \text{ and } 0 \leq n < |m|
\end{cases}
$$

and is undefined otherwise.

Environments are also used to specify an initial environment for a program. The set of environments on X and M is written $Env_{X,M}$. Most often, X and M will be known from the context and will be omitted from both ρ and Env.

Substitution in an environment We define a substitution operation which, given a environment ρ, builds a new environment by modifying only one variable or one byte of one memory array.

- $\rho_{X,M}[bv/x]$ is the environment $\rho'_{X,M}$ such that

  $\forall y \in X \setminus \{x\}, \rho'_{X,M}(y) = \rho_{X,M}(y)$,
  $\forall m \in M, \rho'_{X,M}(m) = \rho_{X,M}(m)$,
  $\rho'_{X,M}(x) = bv$

- $\rho_{X,M}[bv/m[i]]$ is the environment $\rho'_{X,M}$ such that

  $\forall x \in X, \rho'_{X,M}(x) = \rho_{X,M}(x)$,
  $\forall a \in M \setminus \{m\}, \rho'_{X,M}(a) = \rho_{X,M}(a)$,
  $\forall 0 \leq n < |m| \wedge n \neq i, \rho'_{X,M}(m)(n) = \rho_{X,M}(m)(n)$,
  $\rho'_{X,M}(m)(i) = bv$

4.2 Expressions

Bitvectors may be specified in two ways: either by extracting bits from the value of an expression with curly braces (e.g. $x\{0;16\}$ represents the sixteen least significant bits of variable x), or by extracting bytes from memory with square braces. In the latter case, the address and size are given in bytes, and the result is a bitvector of length eight times the length in bytes. Furthermore, the arrow above the size specifies if the bytes should be accessed in big endian ($\rightarrow$) or little endian ($\leftarrow$). For example, the following assignment is legal : **Assign** $x\{0;32\} := \text{ram}[y+8;\overrightarrow{4}]$

All arithmetic operators operate on bitvectors of the same size and produce bitvectors of that size.

The semantics of bitvector operators is given in appendix B, page 14.

The value of an expression e using variables in X and memory arrays in M in a given environment $\rho_{X,M}$ is denoted $[\![e]\!]_{\rho_{X,M}}$. If a division by zero or a modulo zero is encountered during evaluation of e, then the value of e is $\perp$. Otherwise, if e is an arithmetic expression, its value is a bitvector; if it is a formula, its value is either true or false.

4.3 Operational semantics

The operational semantics of a program $\langle X, M, Z, V, E, node\rangle$ is a graph whose vertices are called configurations, built inductively starting from a set of initial configurations. The configurations of the graph are in the set $(V \times Env_{X,M}) \cup \{\perp\}$.

For each (v,ρ) already found, and for each arc $(v, ins, v') \in E$, depending on the instruction *ins*, the graph is augmented by the subgraph given below for each instruction. However, one rule preempts all the other rules: if any of the expressions occurring in the instruction uses bytes declared *read-aborts* in Z or if any of the expressions evaluates to $\perp$, the subgraph is then only $(v,\rho) \rightarrow \perp$.

- **Skip**: $(v,\rho) \rightarrow (v',\rho)$
- **Assign** $x\{i;k\} := e$:

  $(v,\rho) \rightarrow (v',\rho')$ where $\rho' = \rho[bv/x]$ with bv being exactly $\rho(x)$ except for bits $i+k-1,\ldots,i$ which are replaced by $[\![e]\!]$.

- **Assign** $m[e_1;\overleftarrow{k}] := e_2$: (big-endian write)

  We write $i = [\![e_1]\!]$.

  $(v,\rho) \rightarrow \perp$ if some memory byte in $m[i],\ldots,m[i+k-1]$ is declared *write-aborts* in Z

  $(v,\rho) \rightarrow (v',\rho')$ otherwise, where ρ' is ρ after performing the substitutions $[[\![e_2]\!]\{8.(j-i);8\}/m[j]]$ for all $i \leq j < i+k$ such that $m[j]$ is not declared *write-is-ignored* in Z.

- **Assign** $m[e_1; \overrightarrow{k}] := e_2$: (little-endian write)

  We write $i = [\![e_1]\!]$.

$(v, \rho) \to \bot$	if some memory byte in $m[i], \ldots, m[i+k-1]$ is declared *write-aborts* in Z
$(v, \rho) \to (v', \rho')$	otherwise, where ρ' is ρ after performing the substitutions $[[\![e_2]\!]\{8.(i+k-j-1); 8\}/m[j]]$ for all $i \leq j < i+k$ such that $m[j]$ is not declared *write-is-ignored* in Z.

- **Jump** e:

$(v, \rho) \to (v'', \rho)$	where $v'' = node([\![e]\!]_u, (v, ins, v'))$ if it is defined
$(v, \rho) \to \bot$	otherwise

- **Guard** e

$(v, \rho) \to (v', \rho)$	if e evaluates to true
nothing	otherwise

- **External** e

E	such that all pairs $((v, \rho), (v', \rho')) \in E$ satisfy e (make e true)

5 Methodology

The formal model described in this document is designed to model mainly binary programs. We explain in this section how it can be done in practice, and take into account several cases of interest.

5.1 Using the model for disassembled binary code

It is expected that models resulting from the disassembly of binary code will follow the following principles:

- Each register in the processor will be modelled by a variable in the model, and additional internal registers will be used to encode assembly instructions which need intermediate results.
- Each disassembled instruction will be translated in at least one vertex and one arc. Complex instructions may create other vertices and arcs to perform some intermediate computations or to simulate loops. The *node* function of the model should ensure that when given the address of the instruction, it yields the entry vertex of the set of vertices and arcs needed to represent it.
- L-values used in assignments will be "simple", i.e. will use only one occurrence of the [] operator, or of the {} operator, meaning that they will be either a range of bytes in memory or a range of bits in a variable.

- Flags which are often summarized in one register can be split into one 1-bit variable per flag, or encoded into a single variable. The flags have to be updated explicitly by each assembly instruction in the model [1] .
- The program counter need not be modelled because its value is known at disassembly time. The link between the model and the PC value is quite shallow and is used only when resolving dynamic jumps. The *node* function embodies this link.
- Most of the time, only one memory array will be used because most programs execute in a single address space. The cases where a user would want to use more memory arrays include for example: distinguishing an I/O bus from a memory bus, distinguishing supervisor virtual memory from user virtual memory.

5.2 Modelling open programs

Most programs are not self-contained and will make use of other software or hardware to perform their task. We call such programs "open".

Specifying behaviour with logic The instruction provided to deal with open programs is **External**. It allows to express an absent piece of behaviour by a logical specification. For example, if a function provided by the implementation of an Application Programming Interface computes the square root of a number, it is possible to abstract this computation by a single **External** instruction specifying that the square of the return register is "close enough" to the input register.

Simulating behaviour with another model Given the product operation on programs (subsection 3.5 p. 5), it is possible to simulate the execution of a complex piece of code or the behaviour of a piece of hardware thanks to another model which can be synchronized with the main program. Please note that using this method is costly in two ways:

- in order to model access to a memory-mapped peripheral device, each access to memory should be modelled as a test to check if the registers of the device are hit by the memory access, and trigger the other model if and only if it is the case. It will thus lead to larger models.
- computing the asynchronous product of several programs makes the resulting model (very) large. The asynchronous product is a well known cause of state-space explosion.

The advantage of this method is that it is very generic and can probably encompass most modelling needs.

[1] Another possibility would have been to allow operations with multiple return values as in OSMOSE. It leads to smaller models, however, since flags are now part of the main instruction, adding a new flag needs to modify the semantic of the model as well as the underlying analysers. Moreover, our new solution allows to easily remove useless flag computations, while in a multiple return value, all values must be useless to remove the operation.

5.3 Exceptions and interruptions

Although interruptions were declared out of scope for the BINCOA project, we provide here hints of methodology to model them because the model is versatile enough to tackle these concepts.

The control flow of a processor may diverge in case of the occurrence of some exceptional condition. Such a condition may be either synchronous (division by zero, forbidden access to memory, . . .), or asynchronous (disk operation finished, timer reached preset value, . . .). We follow the Motorola convention and call a synchronous interruption an *exception*.

The case of exceptions is handled in a simplistic way in the model in the sense that they make the model explicitly abort, so that their occurrence can be checked as a property of the system reaching or not this special configuration. The exception handling they would normally trigger in a processor is too complex and too processor-specific to be put in the definition of the formal model itself. However, it is possible to model exceptions by conditional jumps after every instruction which may trigger an exception whose handling we wish to monitor.

We suggest two ways to manage asynchronous interruptions: either check separately that the interruption handler behaves as it should and that it does not trash the registers or the stack beyond what is expected. Or model the interruption handler in such a way that the product of its model by the model of the main program behaves as desired.

5.4 Limitations of the model

Here is a list of the main limitations of the model.

Our model does not allow modelling of self-modifying code, however, it can detect such cases. There is no notion of time in the model, so there is no way to formally model timing constraints such as clock interruption handlers executing completely before the next interruption occurs.

There is no explicit interruption handling in the model, so it has to be modelled separately or by using asynchronous product of programs. The latter option is very expensive.

We do not plan to handle floating-point numbers at first. These are easy to add to the formal model, but they pose serious problems to standard verification techniques.

Although not a limitation of the model itself, the recommended way of not using an explicit variable for the program counter means that position independant code cannot be verified as such, but must be set to a specific address first.

A Example

We describe here how to model a small function in our formalism. The function simply copies a given amount of bytes in memory and has the same prototype as the standard C library `memcpy()` fucntion. It is written in C but we model the binary code output by the C compiler. The target architecture is the intel x86, and the compiler used is GCC 4.1.3 under NetBSD 5.0 using optimization level 2.

A.1 C source code

```
#include <stddef.h>

void *
memory_copy(void *dest, const void *src, size_t len) {
    char *d = dest;
    const char *s = src;

    while (len-- > 0)
        *d++ = *s++;

    return dest;
}
```

A.2 Assembly code

This code is the result of compiling the above function with GCC 4.1.3 for i386 with option `-O2` and disassembling the object file with `objdump -d`.

```
00000000 <memory_copy>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   56                      push   %esi
   4:   53                      push   %ebx
   5:   8b 75 08                mov    0x8(%ebp),%esi
   8:   8b 5d 0c                mov    0xc(%ebp),%ebx
   b:   8b 4d 10                mov    0x10(%ebp),%ecx
   e:   85 c9                   test   %ecx,%ecx
  10:   74 0d                   je     1f <memory_copy+0x1f>
  12:   31 d2                   xor    %edx,%edx
  14:   8a 04 1a                mov    (%edx,%ebx,1),%al
  17:   88 04 32                mov    %al,(%edx,%esi,1)
  1a:   42                      inc    %edx
  1b:   39 ca                   cmp    %ecx,%edx
  1d:   75 f5                   jne    14 <memory_copy+0x14>
  1f:   89 f0                   mov    %esi,%eax
  21:   5b                      pop    %ebx
  22:   5e                      pop    %esi
  23:   c9                      leave
  24:   c3                      ret
```

A.3 Explanation of the disassembled code

We provide here a simple explanation of the assembly code above so that people who are not familiar with the x86 can read it.

The first seven instructions prepare the stack frame to respect the x86 Application Binary Interface (to make it possible for a debugger to print a backtrace of the stacked function invocations), save a couple of (callee-saved) registers, and fetch the three arguments from the stack.

There is a test for the special case of a zero length copy, which exits through the postamble at address 0x1f.

The register edx is used as an index into both arrays to avoid incrementing the two pointers (ebx and esi). The copying loop is between 0x14 and 0x1d, and uses register al as a temporary register for each copied byte. The condition to exit the loop is when edx reaches the number of bytes to be copied.

The rest of the code puts the stack pointer back in its original place, restore some registers, and returns to the caller. You can note that esi is copied to eax because the ABI specifies that eax shall contain the return value of the function. In this case, the function returns the (untouched) pointer to the destination buffer.

A.4 A model for this function

The model is the following tuple: $\langle X, M, Z, V, E, node\rangle$. The following subsections give the definitions of the various elements of the tuple.

A.4.1 Variables

Examining the disassembly of the function, we observe that the following registers are used : eax, ebx, ecx, edx, esi, ebp, and esp. Each of these registers will be a 32-bit variable in our model. Note that although al appears, it is just the lowest byte of eax and is thus not modelled separately.

A systematic compilation of the binary code into a model would probably update the flags of the processor after every assembly instruction, but in this example, we do not need to model the flags explicitly because only one is used twice (the "equal" flag, used by instructions `je` and `jne`) and immediately, such that there is no point in setting the flag only to check it right away.

In order to model the `ret` instruction, an internal register that we will call $i0$ is used because the return address has to be popped from the stack *somewhere* before jumping to it.

$$X = \{eax, ebx, ecx, edx, esi, ebp, esp, i0\}$$
$$|eax| = |ebx| = |ecx| = |edx| = |esi| = |ebp| = |esp| = |i0| = 32$$

A.4.2 Memory arrays and memory zone properties

We use here only on memory array that we call ram. Its size is not important as long as it is "large enough", but the model needs to impose a size. We will use very small numbers here compared with a real execution environment, but still keep the various areas aligned on a 4 kibibytes boundary to honor the usual

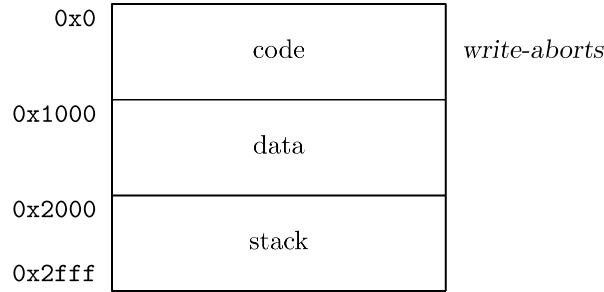
page size on x86 processors. We will thus have a memory of 12 KiB (12288 bytes).

$$M = \{ram\}$$
$$|ram| = 12288$$

With memory zone properties, we will just encode the fact that the page containing the code cannot be written to.

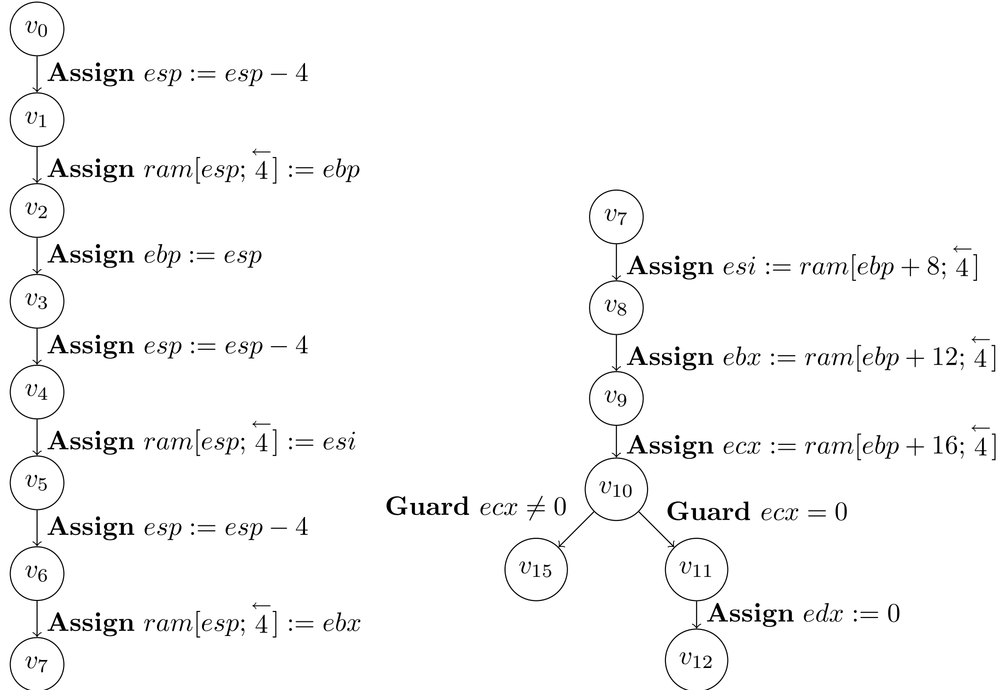
$$Z = \{(ram, 0, 1024, \{\textit{write-aborts}\})\}$$

Although the addresses of the data area and of the stack would appear only in the initial conditions and not in the model itself, we give a small description of our simplified memory organization.



We model the `memory_copy()` function as if it started at address zero, although it would very probably not be the case in practice.

A.4.3 Graph of the model



A.5 The node function

The *node* function of the program is used to map addresses in the code into vertices of the graph. It takes two arguments, but the second argument is useful only when the *node* function belongs to a program which is the result of an asynchronous product; we just ignore it here. Formally, this means that the function returns the same value for any edge passed as the second argument. The following table gives the value of the function for every address where it is defined.

x $(x)_{16}$	0 0	1 1	3 3	4 4	5 5	8 8	11 b	14 e	16 10	18 12
$node(x)$	v_0	v_2	v_3	v_5	v_7	v_8	v_9	v_{10}	v_{10}	v_{11}
x $(x)_{16}$	20 14	23 17	26 1a	27 1b	29 1d	31 1f	33 21	34 22	35 23	36 24
$node(x)$	v_{12}	v_{13}	v_{14}	v_{15}	v_{15}	v_{16}	v_{17}	v_{19}	v_{21}	v_{24}

B Bitvector semantics

B.1 Bitvectors

A bitvector a of size $n \geq 1$ is a sequence $(a_0, \ldots, a_{n-1})$ of $\{0,1\}^n$. We write $a = a_{n-1} \ldots a_0$, and we denote by $\mathrm{BV}(n)$ the set of bitvectors of size n. The size n of a will be denoted by $|a|$. We call a_i the ith bit of a, a_0 its least-significant bit and a_{n-1} its most-significant bit (see below). For $0 \leq k < n$, let $a[k..0] = a_k \ldots a_0$.

Integer concretisation. Bitvectors are classically used in programming languages to encode non-negative integers (power-two encoding) and integers (two's complement encoding). We recall in the following these two encodings.

We define $[\![\cdot]\!]_u$ as a function mapping bitvectors of any size to unsigned integers:

$$[\![a]\!]_u = \sum_{i=0}^{|a|-1} a_i \times 2^i$$

One can check that $[\![\cdot]\!]_u$ is a bijection from $\mathrm{BV}(n)$ to $[0..2^n - 1]$. Given a size $n > 0$ and an unsigned integer $k \in [0..2^n - 1]$, the unique bitvector a of size n satisfying $[\![a]\!]_u = k$ is referred to as the (n-bits) binary encoding of k.

We also define $[\![\cdot]\!]_s$ as a function mapping bitvectors of any size to signed integers:

$$[\![a]\!]_s = -2^{|a|-1} \times a_{|a|-1} + \sum_{i=0}^{|a|-2} a_i \times 2^i$$

Again, one can check that $[\![\cdot]\!]_s$ is a bijection from $\mathrm{BV}(n)$ to $[-2^{n-1}..2^{n-1} - 1]$. Given a size $n > 0$ and an integer $k \in [-2^{n-1}..2^{n-1} - 1]$, the unique bitvector a of size n satisfying $[\![a]\!]_s = k$ is referred to as the (n-bits) two-scomplement encoding of k. Note that the most-significant bit of a is also called the sign bit, since $a_{n-1} = 0$ iff $[\![a]\!]_s \geq 0$.

B.2 BV operators

In the following, we define $\&_1$, $|_1$, $\hat{}^1$: $\mathrm{BV}(1) \times \mathrm{BV}(1) \rightarrow \mathrm{BV}(1)$ as the usual *and*, *or*, *xor* operators on single bits.

Relational operators.

Let $\leq_u \subseteq \mathrm{BV}(n) \times \mathrm{BV}(n)$ denotes the bitvector signed leq relational operator defined by:

$$a \leq_u b = \begin{cases} true & \text{if } a_{n-1} < b_{n-1} \\ false & \text{if } a_{n-1} > b_{n-1} \\ true & \text{if } a_{n-1} = b_{n-1} \wedge |a| = 1 \\ a[n-2..0] \leq_u b[n-2..0] & \text{if } a_{n-1} = b_{n-1} \wedge |a| > 1 \end{cases}$$

Let $\leq_s \subseteq \text{BV}(n) \times \text{BV}(n)$ denotes the signed leq comparaison operator defined by:

$$a \leq_s b = \begin{cases} a \leq_u b & \text{if } a_{n-1} = b_{n-1} \\ true & \text{if } a_{n-1} = 1 \wedge b_{n-1} = 0 \\ false & \text{if } a_{n-1} = 0 \wedge b_{n-1} = 1 \end{cases}$$

Bitwise operations.

All boolean operations are naturally extended to bitwise operations on bitvectors of size n. For example, the bitwise "and" operation $\ \&\ : \text{BV}(n) \times \text{BV}(n) \rightarrow \text{BV}(n)$ is defined by:

$$r = a \ \&\ b$$
$$\text{iff for all } 0 \leq i < n,\ r_i = a_i \ \&_1\ b_i$$

Operators $\ |\ $, $\hat{}$ and $\sim$ are defined similarly.

Other bit-manipulation operations.

Let $\ll$: $\text{BV}(n) \times \mathbb{N} \rightarrow \text{BV}(n)$ denotes the unsigned left shift operation defined by:

$$r = a \ll k$$

$$\text{iff for all } 0 \leq i < n, r_i = \begin{cases} a_{i-k} & \text{if } k \leq i < |a| \\ 0 & \text{if } 0 \leq i < k \end{cases}$$

Let $\gg$: $\text{BV}(n) \times \mathbb{N} \rightarrow \text{BV}(n)$ denotes the unsigned right shift operation defined by:

$$r = a \gg k$$
$$\text{iff for all } 0 \leq i < n, r_i = \begin{cases} a_{i+k} & \text{if } 0 \leq i < |a| - k \\ 0 & \text{if } |a| - k \leq i < |a| \end{cases}$$

Let $\gg_s$: $\text{BV}(n) \times \mathbb{N} \rightarrow \text{BV}(n)$ denotes the signed right shift operation defined by:

$$r = a \gg_s k$$
$$\text{iff for all } 0 \leq i < n, r_i = \begin{cases} a_{i+k} & \text{if } 0 \leq i < |a| - k \\ a_{|a|-1} & \text{if } |a| - k \leq i < |a| \end{cases}$$

Let $ext_u : \text{BV}(n) \times \mathbb{N} \rightarrow \text{BV}(k)$ denotes the bitvector unsigned extension operator defined by:

$$r = ext_u(a, k), k \geq n, |r| = k$$
$$\text{iff for all } 0 \leq i < k, r_i = \begin{cases} a_i & \text{if } i < |a| \\ 0 & \text{if } |a| < i < |r| \end{cases}$$

Let $ext_s : \mathrm{BV}(n) \times \mathbb{N} \rightarrow \mathrm{BV}(k)$ denotes the bitvector signed extension operator defined by:

$$r = ext_s(a, k), k \geq n, |r| = k$$

$$\text{iff for all } 0 \leq i < k, r_i = \begin{cases} a_i & \text{if } 0 \leq i < |a| - 1 \\ 0 & \text{if } |a| - 1 \leq i < |r| - 1 \\ a_{|a|-1} & \text{if } i = |r| - 1 \end{cases}$$

Let $extract : \mathrm{BV}(n) \times \mathbb{N} \times \mathbb{N} \rightarrow \mathrm{BV}(r)$ denotes the bitvector extraction operator defined by:

$$r = a[k : j], j \leq k, |r| = k - j + 1$$

$$\text{iff for all } 0 \leq i < |r| \ : \ r_i = a_{i+j}$$

Let $:: \ : \ \mathrm{BV}(n) \times \mathrm{BV}(m) \rightarrow \mathrm{BV}(n + m)$ denotes the bitvector concatenation operator defined by:

$$r = a :: b, |r| = |a| + |b|$$

$$\text{iff for all } 0 \leq i < |r|, r_i = \begin{cases} b_i & \text{if } 0 \leq i < |b| \\ a_{i-|b|} & \text{if } |b| \leq i < |a| + |b| \end{cases}$$

Linear machine arithmetic.

Let $+_{bv} \ : \ \mathrm{BV}(n) \times \mathrm{BV}(n) \rightarrow \mathrm{BV}(n)$ denotes the bitvector addition operator defined by:

$$r = a +_{bv} b$$

$$\text{iff for all } 0 \leq i < n, \ r_i = (a_i \,\hat{}^1 b_i) \,\hat{}^1 c_i$$

where c_i is recursively defined by

$$c_i = \begin{cases} 0 & \text{if } i = 0 \\ carry(a_{i-1}, b_{i-1}, c_{i-1}) & \text{if } i > 0 \end{cases}$$

with $carry(a, b, c) = (a \ \&_1 \ b) \ |_1 \ ((a \,\hat{}^1 b) \ \&_1 \ c)$.

Let $-_{bv} \ : \ \mathrm{BV}(n) \times \mathrm{BV}(n) \rightarrow \mathrm{BV}(n)$ denotes the bitvector subtraction operator defined by:

$$r = a -_{bv} b$$

$$\text{iff for all } 0 \leq i < n, \ r_i = (a_i \,\hat{}^1 b_i) \,\hat{}^1 bw_i$$

where the borrow bw_i is recursively defined by

$$bw_i = \begin{cases} 0 & \text{if } i = 0 \\ borrow(a_{i-1}, b_{i-1}, c_{i-1}) & \text{if } i > 0 \end{cases}$$

with $borrow(a, b, c) = (\ not_1 \ c \ \&_1 \ (\ not_1 \ a \ \&_1 \ b)) \ |_1 \ (c \ \&_1 \ (\ not_1 \ a \ |_1 \ (a \ \&_1 \ b))))$.

Non-linear machine arithmetic.

Non-linear machine arithmetic operators ($\times_{bv}, /_u, /_s,\ \%_u\ ,\ \%_s$) are too difficult to describe in terms of bit manipulations. We provide instead a translation of $\times_{bv}$ into bitvector operators previously defined. For the four other operators, we rely on their specifiction in terms of integer encoding.

Let $\times_{bv} : \mathrm{BV}(n)\times\mathrm{BV}(n) \rightarrow \mathrm{BV}(n)$ denotes the bitvector multiplication operator defined in terms of basic bv operations by :

$$r = a \times_{bv} b$$

$$r = \sum_{i=0}^{|b|-1} (a \ll i)\ \&\ ext_s(b_i, |a|)$$

B.3 Arithmetic properties

We recall here some well known properties about integer concretisations of bitvectors. In the following theorems, we let $n = |a| = |b|$ when a and b have the same size.

$$a \leq_u b \text{ iff } [\![a]\!]_u \leq [\![b]\!]_u$$

$$a \leq_s b \text{ iff } [\![a]\!]_s \leq [\![b]\!]_s$$

$$[\![a +_{bv} b]\!]_u = ([\![a]\!]_u + [\![b]\!]_u)\ mod\ 2^n$$

$$[\![a +_{bv} b]\!]_s = ([\![a]\!]_s + [\![b]\!]_s)\ mod\ 2^n$$

$$[\![a -_{bv} b]\!]_u = ([\![a]\!]_u - [\![b]\!]_u)\ mod\ 2^n$$

$$[\![a -_{bv} b]\!]_s = ([\![a]\!]_s - [\![b]\!]_s)\ mod\ 2^n$$

$$[\![-_{bv} a]\!]_u = (2^{|a|} - [\![a]\!]_u)\ mod\ 2^{|a|}$$

$$[\![a \times_{bv} b]\!]_u = ([\![a]\!]_u \times [\![b]\!]_u)\ mod\ 2^n$$

$$[\![a \times_{bv} b]\!]_s = ([\![a]\!]_s \times [\![b]\!]_s)\ mod\ 2^n$$

$$[\![a /_u b]\!]_u = [\![a]\!]_u \, // \, [\![b]\!]_u$$

$$[\![a /_s b]\!]_s = ([\![a]\!]_s \, // \, [\![b]\!]_s) \; mod \; 2^n$$

$$[\![a \ \%_u \ b]\!]_u = [\![a]\!]_u \ \ mod \ \ [\![b]\!]_u$$

$$[\![a \ \%_s \ b]\!]_s = ([\![a]\!]_s \ \ mod \ \ [\![b]\!]_s) \; mod \; 2^n$$

$$[\![a \ll k]\!]_u = ([\![a]\!]_u \times 2^k) \; mod \; 2^{|a|}$$

$$[\![a \gg k]\!]_u = [\![a]\!]_u \ \ // \ 2^k$$

$$[\![a \gg_s k]\!]_s = [\![a]\!]_s \ \ // \ 2^k$$

$$[\![ext_u(a, i)]\!]_u = [\![a]\!]_u$$

$$[\![ext_s(a, i)]\!]_s = [\![a]\!]_s$$

$$[\![a :: b]\!]_u = [\![a]\!]_u \times 2^{|b|} + [\![b]\!]_u$$

$$[\![\sim r]\!]_u = 2^{|a|} - 1 - [\![a]\!]_u$$

B.4 Encoding of signed operators

All signed operators can be encoded with unsigned operators, case-splits and basic signed comparisons to 0. For example $/_s : \mathrm{BV}(n) \times \mathrm{BV}(n) \to \mathrm{BV}(n)$ can be encoded by

$$r = a /_s b$$

$$[\![a]\!]_s \neq 0$$

$$\begin{cases} a /_u b & \text{if } \ a \geq_s 0 \wedge b \geq_s 0 \\ -_{bv}(a /_u (-_{bv} b)) & \text{if } \ a \geq_s 0 \wedge b <_s 0 \\ -_{bv}((-_{bv} a) /_u b) & \text{if } \ a <_s 0 \wedge b \geq_s 0 \\ (-_{bv} a) /_u (-_{bv} b) & \text{if } \ a <_s 0 \wedge b <_s 0 \end{cases}$$

The same relationship holds between $\%_s$ and $\%_u$. Moreover, $\leq_s$, ext_s and $\gg_s$ can also be encoded into a similar way.