Certifying Machine Code Safe from Hardware Aliasing

RISC is not necessarily risky

Peter T. Breuer[1] and Jonathan P. Bowen[2]

[1] Department of Computer Science, University of Birmingham, UK
ptb@cs.bham.ac.uk
[2] Department of Informatics, London South Bank University, UK
jonathan.bowen@lsbu.ac.uk

Abstract. Sometimes machine code turns out to be a better target for verification than source code. RISC machine code is especially advantaged with respect to source code in this regard because it has only two instructions that access memory. That architecture forms the basis here for an inference system that can prove machine code safe against 'hardware aliasing', an effect that occurs in embedded systems. There are programming memes that ensure code is safe from hardware aliasing, but we want to certify that a given machine code is provably safe.

1 Introduction

In a computer system, 'software' aliasing occurs when different logical addresses simultaneously or sporadically reference the same physical location in memory. We are all familiar with it and think nothing of it, because the same physical memory is nowadays reused millisecond by millisecond for different user-space processes with different addressing maps, and we expect the operating system kernel to weave the necessary illusion of separation. The kernel programmer has to be aware that different logical addresses from different or even the same user-space process may alias the same physical location, but the application programmer may proceed unawares.

We are interested in a converse situation, called 'hardware' aliasing, where different physical locations in memory are sporadically bound to the same logical address. If software aliasing is likened to one slave at the beck of two masters, hardware aliasing is like identical twins slaved to one master who cannot tell which is which. In this paper we will investigate the safety of machine code in the light of hardware aliasing issues.

Aliasing has been studied before [10] and is the subject of some patents [7,11]. There appears to be no theoretical treatment published, although the subject is broadly treated in most texts on computer architecture (see, for example, Chapter 6 of [1]) and is common lore in operating systems kernel programming. The 'hardware' kind of aliasing arises particularly in embedded systems where the arithmetic components of the processor are insufficient to fill all the address lines. Suppose, for example, that the memory has 64-bit addressing but the processor only has 40-bit arithmetic. The extra lines might be grounded, or sent high, and this varies from platform to platform. They may be connected to 64-bit address registers in the processor, so their values change from moment to moment as the register is filled. In that case, it is up to the software to set the 'extra' bits reliably to zero, or one, or some consistent value, in order that computing an address may yield a consistent result.

We first encountered the phenomenon in the context of the KPU [6], a general purpose 'crypto-processor', i.e., a processor that performs its computations in encrypted form in order to provide security against observation and protection from malware. Because real encryptions are one-to-many, the result of the encrypted calculation of the address $1 + 1$ will always mean '2' when decrypted, but may be different from another encryption of 2. If the two different *physical aliases* are used as addresses, then two different memory cell contents are accessed and the result is chaotic. The same effect occurs in the embedded system that has processor arithmetic with fewer bits than there are address lines; add $1 + 1$ in the processor and instead of 2, $0\mathrm{xff}01000000000002$ may be returned. If those two aliases of the arithmetic '2' are used as addresses, they access different memory cells. The upshot is that what is meant both times to be '2' accesses different locations according to criteria beyond the programmer's control.

There are programming memes that are successful in an aliasing environment: if a pointer is needed again in a routine, it must be copied exactly and saved for the next use; when an array or string element is accessed, the address must always be calculated in exactly the same way. But whatever the programmer says, the compiler may implement as it prefers and ultimately it is the machine code that has to be checked in order to be sure that aliasing is not a risk at run-time. Indeed, in an embedded environment it is usual to find the programmer writing in assembler precisely in order to control the machine code emitted. The Linux kernel consists of about 5% hand-written assembly code, for example (but rarely in segments of more than 10-15 lines each). One of our long term objectives is to be able to boot a Linux kernel on an embedded platform with aliasing, the KPU in particular. That requires both modifying a compiler and checking the hand-written machine-level code in the open source archive.

An inference system will be set out here that can guarantee a (RISC [2,9]) machine code program safe against hardware aliasing as described. The idea is to map a stack machine onto the machine code. We will reason about what assembly language instructions for the stack machine do computationally. Choosing an inference rule to apply to a machine code instruction is equivalent to choosing a stack machine assembly language [5] instruction to which it disassembles [3,4]. The choice must be such that a resulting proof tree is well-formed, and that acts as a guide. The stack machine is aliasing-proof when operated within its intended parameters so verifying alias-safety means verifying that the stack machine assembly language code obtained by disassembly of the RISC machine code does not cause the stack machine to overstep certain bounds at run-time.

The RISC machine code we can check in this way is *ipso facto* restricted to that which we can disassemble. At the moment, that means code that uses string or string-like data structures and arrays which do not contain further pointers, and which uses machine code 'jump and link' and 'jump register' instructions only for subroutine call and return respectively, and in which subroutines make their own local frame and do not access the caller's frame (arguments are passed to subroutines in registers). These restrictions are not fundamental, but in any case there are no functional limitations implied by them; one call convention is functionally as good as another and data structures may always be laid out flat, as they are in a relational DB.

Mistakes in disassembly are possible: if a 'jump register' instruction, for example, were in fact used to implement a computed goto and not a subroutine return, it could still be treated as a subroutine return by the verification, which would end prematurely,

possibly missing an error further along and returning a false negative. A mistaken return as just described would always fail verification in our system, but other such situations are conceivable in principle. So a human needs to check and certify that the proposed disassembly is not wrongheaded. The practice is not difficult because, as noted above, hand-written machine code at a professional standard consists of short, concise, commented segments. The difficulty is that there is often a great deal of it to be checked and humans tire easily. But our system reduces the burden to checking the disassembly proposed by the system against the comments in the code.

This paper is structured as follows: after an illustration of programming against aliasing in Section 2 and a discussion of disassembly in Section 3, code annotation is introduced in sections 4, 5 and 6, with a worked example in Section 7. Section 8 argues that code annotation gives rise to the formal assurance that aliasing cannot occur.

2 Programming memes

We model aliasing as being introduced when memory addresses are calculated in different ways. That model says that a memory address may be *copied* exactly and used again without hazard, but if even 0 is added to it, then a different alias of the address may result, and reads from the new alias do not return data deposited at the old alias of the address. Arithmetically the aliases are equivalent in the processor; they will test as equal but they are not identical, and using them as addresses shows that up.

☠	✓
`foo:`	`foo:`
	`gp = sp`
`sp -= 32`	`sp -= 32`
...code...	...code...
`sp += 32`	`sp = gp`
`return`	`return`

Table 1: Aliasing in function *foo*.

That is particularly a problem for the way in which a compiler – or an assembly language programmer – renders machine code for the stack pointer movement around a function call. Classically, a subroutine starts by decrementing the stack pointer to make room on the stack for its local frame. Just before return, it increments the stack pointer back to its original value. The pseudo-code is shown on the left in Table 1. In an aliasing context, the attempt at arithmetically restoring the pointer puts an alias of the intended address in the **sp** register, and the caller may receive back a stack pointer that no longer points to the data. The code on the right in Table 1 works correctly; it takes an extra register (**gp**) and instruction, but the register content may be moved to the stack and restored before return, avoiding the loss of the slot.

string	☠	☠	✓
array	✓	☠	☠
		`s+= 2`	`s++; s++`
	`x = s[2]`	`x = *s`	`x = *s`

Table 2: Aliasing while accessing a string or array.

Strings and arrays are also problematic in an aliasing environment because different calculations for the address of the same element cause aliasing. To avoid it, the strategy we will follow is that elements of 'string-like' structures will be accessed by incrementing the base address in constant steps (see the pseudo-code at right in Table 2) and array elements will be accessed via a unique offset from the array base address (see the

A RISC machine code processor consists of 32 (32-bit) integer registers R, a vector of 2^{32} (32-bit) integer memory locations M, and the program counter p. The latter gives the address of the current instruction. The **ra** register is used to hold a subroutine call return address. Only two instructions, **sw** and **lw**, access memory.

instruction	mnemonic	semantics
$\mathbf{sw}\ r_1\ k(r_2)$	store word	$M' = M \oplus \{R\,r_2 \mathbin{+} k \mapsto R\,r_1\};\ R' = R;\ p' = p{+}4$
$\mathbf{lw}\ r_1\ k(r_2)$	load word	$M' = M;\ R' = R \oplus \{r_1 \mapsto M(R\,r_2 \mathbin{+} k)\};\ p' = p{+}4$
$\mathbf{move}\ r_1\ r_2$	move/copy	$M' = M;\ R' = R \oplus \{r_1 \mapsto R\,r_2\};\ p' = p{+}4$
$\mathbf{li}\ r_1\ k$	load immediate	$M' = M;\ R' = R \oplus \{r_1 \mapsto k\};\ p' = p{+}4$
$\mathbf{addiu}\ r_1\ r_2\ k$	add immediate	$M' = M;\ R' = R \oplus \{r_1 \mapsto R\,r_2 \mathbin{+} k\};\ p' = p{+}4$
$\mathbf{addu}\ r_1\ r_2\ r_3$	add variable	$M' = M;\ R' = R \oplus \{r_1 \mapsto R\,r_2 \mathbin{+} R\,r_3\};\ p' = p{+}4$
$\mathbf{nand}\ r_1\ r_2\ r_3$	bitwise not-and	$M' = M;\ R' = R \oplus \{r_1 \mapsto R\,r_2 \mathbin{\overline{\&}} R\,r_3\};\ p' = p{+}4$
$\mathbf{beq}\ r_1\ r_2\ k$	branch-if-equal	$M' = M;\ R' = R;\ \mathbf{if}\ (R\,r_1 \mathbin{=} R\,r_2)\ p' = k\ \mathbf{else}\ p' = p{+}4$
$\mathbf{jal}\ k$	jump-and-link	$M' = M;\ R' = R \oplus \{\mathbf{ra} \mapsto p{+}4\};\ p' = k$
$\mathbf{jr}\ r$	jump-register	$M' = M;\ R' = R;\ p' = R\,r$

Notation. $M \oplus \{a \mapsto v\}$ means the vector M overwritten at index a with the value v; the processor arithmetic (bold font '+') is distinguished from the instruction addressing arithmetic (light font '$+$'); r_1, r_2 are register names or indices; k is a signed 16-bit integer; x and x' are respectively initial and final value after the instruction has acted.

Box 1: RISC machine code instructions and their underlying semantics.

pseudo-code at left in Table 2). This technique ensures that there is only one calculation possible for the address of each string element (it is $((s{+}1){+}1){+}0$ in Table 2) or array element ($s{+}2$ in Table 2), so aliasing cannot occur. The middle code in Table 2 gives address $(s{+}2){+}0$ which matches exactly neither string nor array calculations. The decision over whether to treat a memory area like a string or an array depends on the mode of access to be used.

3 Disassembly

Say that the stack pointer s is in the stack pointer register **sp** in the machine code processor. A corresponding abstract stack machine state is a 4-tuple $(\mathcal{R}, \mathcal{K}, \mathcal{H}, p)$, where $\mathcal{R}$ consists of the 31 registers excluding the stack pointer register, the stack $\mathcal{K}$ consists of the top part of memory above the stack pointer value s, the heap $\mathcal{H}$ consists of the bottom part of memory below the stack pointer, and the address p is that of the current instruction.

$$\begin{array}{ll} \mathcal{K}\,k = M(s \mathbin{+} k) & s = R\,\mathbf{sp},\ k \geq 0 \\ \mathcal{R}\,r = R\,r & r \neq \mathbf{sp},\ r \in \{0, \ldots 31\} \\ \mathcal{H}\,a = M\,a & a < s \end{array}$$

The (hidden) stack pointer value s is needed to recreate the machine code processor state (R, M, p) from the stack machine state $(\mathcal{R}, \mathcal{K}, \mathcal{H}, p)$, so the latter is more abstract.

Box 2: Relation of processor to stack machine.

Nothing in the machine code indicates which register holds a subroutine return address, and that affects which machine code instructions may be interpreted as a return from a subroutine call. To deal with this and similar issues in an organised manner, we describe rules of reasoning about programs both in terms of the machine code instruction to which they apply and an assembly language instruction for a more abstract *stack machine* that the machine code instruction may be disassembled to and which we imagine the programmer is targeting.

The core RISC machine code instructions are listed in Box 1, where their semantics are given as state-to-state transformations on the three components of a RISC processor: 32 32-bit registers R, memory M

Table 3: Stack machine instructions: the n are small integers, the r are register names or indices, and the a are relative or absolute addresses.

s ::=	**cspt** r | **cspf** r | **rspf** r | **push** n	// stack pointer movement
|	**get** r n | **put** r n | ...	// stack access
|	**newx** r a n | **stepx** r n | **getx** r n(r) | **putx** r n(r) | ...	// string operations
|	**newh** r a n | **lwfh** r n(r) | **swfh** r n(r) | ...	// array operations
|	**gosub** a | **return** | **goto** a | **ifnz** r a | ...	// control operations
|	**mov** r r | **addaiu** r r n | ...	// arithmetic operations

Table 4: Machine code may be disassembled to one of several alternate assembly language instructions for a stack machine.

machine code	assembly language
move r_1 r_2	**cspt** r_1 **cspf** r_2 **rspf** r_2 **mov** r_1 r_2
addiu r r n	**push** -n **stepx** r n **addaiu** r r n
lw r_1 $n(r_2)$	**get** r_1 n **lwfh** r_1 $n(r_2)$ **getx** r_1 $n(r_2)$
sw r_1 $n(r_2)$	**put** r_1 n **swfh** r_1 $n(r_2)$ **putx** r_1 $n(r_2)$

machine code	assembly language
lb r_1 $n(r_2)$	**getb** r_1 n **lbfh** r_1 $n(r_2)$ **getbx** r_1 $n(r_2)$
sb r_1 $n(r_2)$	**putb** r_1 n **sbth** r_1 $n(r_2)$ **putbx** r_1 $n(r_2)$
jal a	**gosub** a
jr r	**return**
j a	**goto** a
li r a	**newx** r a n **newh** r a n
bnez r a	**ifnz** r a

and a 32-bit program counter p. The corresponding abstract stack machine is described in Box 2. The stack pointer address s in the machine code processor notionally divides memory M into two components: stack $\mathcal{K}$ above and heap $\mathcal{H}$ below. The stack machine manipulates the stack directly via instructions that operate at the level of stack operations, and they are implemented in the machine code processor via instructions that act explicitly on the stack pointer. No stack pointer is available in the abstract machine. Its registers $\mathcal{R}$ consist of the set R in the machine code processor *minus* the register that contains the stack pointer, usually the **sp** register. The program counter p is the same in the abstract stack machine as in the machine code processor, because instructions correspond one-to-one between programs for each machine. However, there is usually a choice of more than one abstract stack machine instruction that each machine code instruction could have been disassembled to, even though only one is chosen.

For example, several different stack machine instructions may all be thought of as manipulating the hidden stack pointer, register **sp** in the machine code processor, and they all are implemented as a **move** ('copy') machine code instruction. Thus the **move** instruction disassembles to one of several stack machine instructions as follows:

1. The **cspt** r_1 ('copy stack pointer to') instruction saves a copy of the stack pointer in register r_1. It corresponds to the **move** r_1 **sp** machine code processor instruction.
2. The **cspf** r_1 ('copy stack pointer from') instruction *refreshes* the stack pointer from a copy in r_1 that has the same value and was saved earlier (we will not explore here

the reasons why a compiler might issue such a 'refresh' instruction). It corresponds to the **move sp** r_1 machine code instruction.

3. The **rspf** r_1 ('restore stack pointer from') instruction returns the stack pointer to a value that it held previously by copying an old saved value from r_1. It also corresponds to **move sp** r_1.

A fourth disassembly of the machine code **move** instruction, to the stack machine **mov** instruction, encompasses the case when the stack pointer is not involved at all; it does a straight copy of a word from one register to another at the stack machine level. The full set of stack machine instructions is listed in Table 3, and their correspondence with RISC machine code instructions is shown in Table 4.

We will not work through all the instructions and disassembly options in detail here, but note the important **push** n instruction in the stack machine, which can be thought of as decrementing the hidden stack pointer by n, extending the stack downwards. It corresponds to the **addiu sp sp** m machine code instruction, with $m = -n$. Also, the stack machine instructions **put** r_1 n and **get** r_1 n access the stack for a word at offset n bytes, and they correspond to the machine code **sw** r_1 $n(\mathbf{sp})$ and **lw** r_1 $n(\mathbf{sp})$ instructions, respectively.

The very same machine code instructions may also be interpreted as stack machine instructions that manipulate not the stack but either a 'string-like' object or an array. Strings/arrays are read with **getx**/**lwfh** and written with **putx**/**swth**. Table 4 shows that these are implemented by **lw**/**sw** in the machine code processor, applied to a base register $r_2 \neq \mathbf{sp}$. Stepping through a string is done with the **stepx** instruction in the stack machine, which is implemented by **addiu** in the machine code procesor. Introducing the address of a string/array in the stack machine needs **newx**/**newh** and those are both implemented by the **li** ('load immediate') instruction in the machine code processor.

There are also 'b' ('byte-sized') versions of the **get**, **lwfh**, **getx** stack machine instructions named **getb**, **lbfh**, **getbx** respectively. These are implemented by **lb** in the machine code processor. For **put**, **swth**, **putx** we have byte versions **putb**, **sbth**, **putbx**.

4 Introducing annotations and annotated types

```
foo:
  move gp sp
  addiu sp sp -32
  ...code...
  move sp gp
  jr ra
```

Table 5: Non-aliasing subroutine machine code.

Consider the 'good' pseudo-code of Table 1 implemented as machine code and shown in Table 5. How do we show it is aliasing-safe? Our technique is to *annotate* the code in a style akin to verification using Hoare logic, but the annotation logic is based on the stack machine abstraction of what the machine code does. We begin with an annotation that says the **sp** register is bound to a particular *annotation type* on entry:

$$\{\, \mathbf{sp} = \mathbf{c}!0!4!8 \,\}$$

The '**c**' as base signifies a variable pointer value is in register **sp**. It is the stack pointer value. The '!0!4!8' means that that particular value has been used as the base address for writes to memory at offsets 0, 4 and 8 bytes from it, respectively.

The first instruction in subroutine *foo* copies the stack pointer to register **gp** and we infer that register **gp** also gets the '**c**' annotation, using a Hoare-triple-like notation:

$$\{\, \mathbf{sp}^* = \mathbf{c}!0!4!8 \,\}\ \texttt{move gp sp}\ \{\, \mathbf{sp}^*, \mathbf{gp} = \mathbf{c}!0!4!8 \,\}$$

The stack pointer location (in the **sp** register) should always be indicated by an asterisk.

The arithmetic done by the next instruction destroys the offset information. It cannot yet be said that anything has been written at some offset from the new address, which is 32 distant from the old only up to an arithmetic equivalence in the processor:

$$\{\, \mathbf{sp}^*, \mathbf{gp} = \mathbf{c}!0!4!8 \,\}\ \texttt{addiu sp sp -32}\ \{\, \mathbf{gp} = \mathbf{c}!0!4!8;\ \mathbf{sp}^* = \mathbf{c} \,\}$$

Suppose the annotation on the **gp** register is still valid at the end of subroutine *foo*, so the stack pointer register is finally refreshed by the **move** instruction with the same annotation as at the start:

$$\{\, \mathbf{sp}^* = \mathbf{c};\ \mathbf{gp} = \mathbf{c}!0!4!8; \,\}\ \texttt{move sp gp}\ \{\, \mathbf{sp}^*, \mathbf{gp} = \mathbf{c}!0!4!8 \,\}$$

The return (**jr ra**) instruction does not change these annotations. So the calling code has returned as stack pointer a value that is annotated as having had values saved at offsets 0, 4, 8 from it, and the caller can rely on accessing data stored at those offsets. That does not guarantee that the *same* value of the stack pointer is returned to the caller, however. It will be shown below how this system of annotations may be coaxed into providing stronger guarantees.

5 Types for stack, string and array pointers

The annotation discussed above is not complete. The *size* in bytes of the local stack frame needs to be recorded by following the '**c**' with the frame size as a superscript. Suppose that on entry there is a local stack frame of size 12 words, or 48 bytes. Then here is the same annotation with superscripts on, written as a derivation in which the appropriate disassembly of each machine code instruction is written to the right of the machine code as the 'justification' for the derivation:

$$\dfrac{\dfrac{\dfrac{\{\mathbf{sp}^* = \mathbf{c}^{48}!0!4!8\}}{\{\mathbf{sp}^*, \mathbf{gp} = \mathbf{c}^{48}!0!4!8\}}\ \textbf{move}\ \text{gp sp} \qquad / \textbf{cspt}\ \text{gp}}{\{\mathbf{sp}^* = \mathbf{c}^{32^{48}};\ \mathbf{gp} = \mathbf{c}^{48}!0!4!8\}}\ \textbf{addiu}\ \text{sp sp -32} \quad / \textbf{push}\ 32}{\vdots}$$

$$\dfrac{\{\mathbf{sp}^* = \mathbf{c}^{32^{48}};\ \mathbf{gp} = \mathbf{c}^{48}!0!4!8\}}{\{\mathbf{sp}^*, \mathbf{gp} = \mathbf{c}^{48}!0!4!8\}}\ \textbf{move}\ \text{sp gp} \qquad / \textbf{rspf}\ \text{gp}$$

The **push** 32 abstract stack machine instruction makes a *new* local stack frame of 8 words or 32 bytes. It does not increase the size of the current frame. Accordingly, the 32 'pushes up' the 48 in the annotation so that 32^{48} is shown. This makes the size of the previous stack frame available to the annotation logic.

A different disassembly of **addaiu** $r\ r\ n$ is required when r contains a string pointer, not the stack pointer, which means that register r lacks the asterisk in the annotation.

The disassembly as a step along a string is written **stepx** r n, and requires n to be positive. In this case, the string pointer in r will be annotated with the type

$$\mathbf{c}^{\ddot{1}}$$

meaning that it is a 'calculatable' value that may be altered by adding 1 to it repeatedly. The form $\mathbf{c}^{\ddot{1}}$ hints that a string is regarded as a stack $\mathbf{c}^{1^{\cdot^{\cdot}}}$ that starts 'pre-charged' with an indefinite number of frames of 1 byte each, which one may step up through by 'popping the stack' one frame, and one byte, at a time. So annotation types may be either like $\mathbf{c}^{32^{48}}$ or $\mathbf{c}^{\ddot{1}}$ and these may be followed by offsets $!0!4!8!\ldots$. There is just one more base form, described below, completing the list in Box 3.

The RISC instruction **lw** r_1 $n(r_2)$ is also disassembled differently according to the annotated type in r_2. As **get** r_1 n it retrieves a value previously stored at offset n in the stack, when $n \geq 0$ and r_2 is the stack pointer register. As **lwfh** r_1 $n(r_2)$ it retrieves an element in an array from the *heap* area. In that case, r_2 will be annotated

$$\mathbf{u}^{m}$$

meaning an 'unmodifiable' pointer to an array of size m bytes, and $m - 4 \geq n \geq 0$. A third possibility is dissassembly as retrieval from a string-like object in the heap, when, as **getx** r_1 $n(r_2)$, register r_2 will have a 'string-like' annotation of the form $\mathbf{c}^{\ddot{m}}$, meaning that it must be stepped through in increments of m bytes.

Similarly the RISC **sw** r_1 $n(r_2)$ instruction can be dissassembled as **put** r_1 n of a value at offset n to the stack, or **swth** r_1 $n(r_2)$ to an array or **putx** r_1 $n(r_2)$ to a string, depending on the type bound to register r_2. These register types drive the disassembly.

Annotations a assert a binding of registers r or stack slots (n) to an *annotated type* t. One of the register names may be starred to indicate the stack pointer position. A type is either 'uncalculated', **u**, or 'calculated', **c**. Either may be decorated with '$!n$' annotations indicating historical writes at that offset from the typed value when used as an address. A **c** base type may also be superscripted by a 'tower' of natural numbers n denoting 'frame sizes' (see text), while a **u** base type may have a single superscript (also denoting size). We also use $\ddot{1}$ for a tower $1^{1^{\cdot^{\cdot}}}$ of undetermined extent and a single repeated size. Also, formal type variables ***x***, ***y***, etc are valid stand-ins for annotated types, and formal 'set of offsets variables' ***X***, ***Y***, etc are valid stand-ins for sets of offsets.

$$a ::= r^{[*]}\mathbf{,}\ldots\mathbf{,(}n\mathbf{),}\ldots\mathbf{=}t\mathbf{;}\ldots$$
$$t ::= \mathbf{c}^{[n^{\cdot^{\cdot}}]}\mathbf{!}n\mathbf{!}\ldots \mid \mathbf{u}^{[n]}\mathbf{!}n\mathbf{!}\ldots$$

Box 3: Syntax of annotations and types.

6 Formal logic

We can now write down formal rules for the logic of annotations introduced informally in the 'derivation' laid out in the previous section. Readers who would prefer to see a worked example first should jump directly to Section 7.

We start with a list of so-called 'small-step' program annotations justified by individual stack machine instructions, each the disassembly of a machine code instruction. The small-step rules relate the annotation before each machine code instruction to the annotation after. Table 6 helps to reduce *a priori* the number of possible disassemblies

Table 6: Possible disassemblies of machine code instructions as constrained by the stack pointer register location changes (SP←SP) or absence (×), and changes to the stack content ('delta').

move r_1 r_2	r_1	r_2	stack delta
rspf r_2	SP ↻	×	yes
cspf r_2	SP ↻	×	no
cspt r_1	×	SP ↻	no
mspt r_1	SP←—SP		no
mov r_1 r_2	×	×	no

addiu r_1 r_2 m	r_1	r_2	stack delta
step r m	×		no
stepto r_1 r_2 m	×	×	no
push $-m$	SP ↻		yes
pushto r_1 $-m$	SP←—SP		yes
addaiu r_1 r_2 m	×	×	no

lw r_1 $m(r_2)$	r_1	r_2	stack delta
get r_1 m	×	SP ↻	no
lwfh r_1 $m(r_2)$	×	×	no
getx r_1 $m(r_2)$	×	×	no

sw r_1 $m(r_2)$	r_1	r_2	stack delta
put r_1 m	×	SP ↻	no
swth r_1 $m(r_2)$	×	×	no
putx r_1 $m(r_2)$	×	×	no

for each machine code instruction, but in principle disassembly to stack machine code does not have to be done first, but can be left till the last possible moment during the annotation process, as each dissassembly choice corresponds to the application of a different rule of inference about which annotation comes next. If the corresponding inference rule may not be applied, then that disassembly choice is impossible.

Here is how to read Table 7. Firstly, 'offsets variables' ***X***, ***Y***, etc, stand in for sets of offset annotations '$!k$'. For example, the **put** $\mathbf{gp}$ 4 instruction is expected to start with a prior annotation pattern $\mathbf{sp}^* = \mathbf{c}^f!\boldsymbol{X}$ for the stack pointer register. Secondly, the stack pointer register is indicated by an asterisk. Thirdly, f in the table stands for some particular stack frame tower of integers; it is not a variable, being always some constant in any patrticular instance. In the case of the **put** $\mathbf{gp}$ 4 instruction, f must start with some particular number at least 8 in size, in order to accommodate the 4-byte word written at offset 4 bytes within the local stack frame. Just '8' on its own would do for f here. Lastly, 'type variables' ***x***, ***y***, etc, where they appear, stand in for full types.

The table relates annotations before and after each instruction. So, in the case of the **put** $\mathbf{gp}$ 4 instruction, if the prior annotation for the stack pointer register is $\mathbf{sp}^* = \mathbf{c}^f!\boldsymbol{X}$, then the post annotation is $\mathbf{sp}^* = \mathbf{c}^f!4!\boldsymbol{X}$, meaning that 4 is one of the offsets at which a write has been made. It may be that 4 is also a member of the set denoted by ***X*** (which may contain other offsets too), or it may be not in ***X***. That is not decided by the formula, which merely says that whatever other offsets there are in the annotation, '4' is put there by this instruction. At any rate, the annotation pattern for the **put** $\mathbf{gp}$ 4 instruction is:

$$\{\ldots;\mathbf{sp}^* = \mathbf{c}^f!\boldsymbol{X};\ldots\}\ \mathbf{put}\ \mathbf{gp}\ 4\ \{\ldots;\mathbf{sp}^* = \mathbf{c}^f!4!\boldsymbol{X};\ldots\}$$

and considering the effect on the **gp** register (which may be supposed to have the type denoted by the formal type variable ***x*** initially) and the stack slot denoted by '(4)' gives

$$\{\mathbf{gp}{=}\boldsymbol{x};\mathbf{sp}^*{=}\mathbf{c}^f!\boldsymbol{X}\}\ \mathbf{put}\ \mathbf{gp}\ 4\ \{\mathbf{sp}^*{=}\mathbf{c}^f!4!\boldsymbol{X};\mathbf{gp},(4){=}\boldsymbol{x}\}$$

because whatever the description ***x*** of the data in register **gp** before the instruction runs, since the data is transferred to stack slot '(4)', the latter gains the same description.

Table 7: ‘Small-step’ annotations on assembly instructions.

$\{\ \}$	**newx** $r\ n$	$\{r\ {=}\mathbf{c}^{\ddot{n}}!\boldsymbol{X}\}$	// Set reg. r content
$\{r_1{=}\mathbf{c}^{f_1}!\boldsymbol{Y}; r_2{=}\mathbf{u}^{f_2}!\boldsymbol{X}\}$	**putx** $r_1\ n(r_2)$	$\{r_1{=}\mathbf{c}^{f_1}!\boldsymbol{Y}; r_2{=}\mathbf{u}^{f_2}!n!\boldsymbol{X}\}$	// Store word to string
$\{r_2{=}\mathbf{u}^{f}!n!\boldsymbol{X}\}$	**getx** $r_1\ n(r_2)$	$\{r_1{=}\mathbf{c}^{0}; r_2{=}\mathbf{u}^{f}!n!\boldsymbol{X}\}$	// Load word from string
$\{r{=}\mathbf{c}^{n^f}!\boldsymbol{X}\}$	**stepx** $r\ n$	$\{r{=}\mathbf{c}^{f}!\boldsymbol{Y}\}$	// Step along string
$\{\ \}$	**newh** $r\ n$	$\{r\ {=}\mathbf{u}^{n}!\boldsymbol{X}\}$	// Set reg. r content
$\{r_1{=}\mathbf{c}^{f_1}!\boldsymbol{Y}; r_2{=}\mathbf{u}^{f_2}!\boldsymbol{X}\}$	**swth** $r_1\ n(r_2)$	$\{r_1{=}\mathbf{c}^{f_1}!\boldsymbol{Y}; r_2{=}\mathbf{u}^{f_2}!n!\boldsymbol{X}\}$	// Store word to array
$\{r_2{=}\mathbf{u}^{f}!n!\boldsymbol{X}\}$	**lwfh** $r_1\ n(r_2)$	$\{r_1{=}\mathbf{c}^{0}; r_2{=}\mathbf{u}^{f}!n!\boldsymbol{X}\}$	// Load word from array
$\{r_1{=}\boldsymbol{x}; r_2^*{=}\mathbf{c}^{f}!\boldsymbol{X}\}$	**put** $r_1\ n$	$\{r_1{,}(n){=}\boldsymbol{x}; r_2^*{=}\mathbf{c}^{f}!n!\boldsymbol{X}\}$	// Store word to stack
$\{r_2^*{=}\mathbf{c}^{f}!n!\boldsymbol{X}; (n){=}\boldsymbol{x}\}$	**get** $r_1\ n$	$\{r_1{,}(n){=}\boldsymbol{x}; r_2^*{=}\mathbf{c}^{f}!n!\boldsymbol{X}\}$	// Load word from stack
$\{r^*{=}\mathbf{c}^{f}!\boldsymbol{X}\}$	**push** n	$\{r^*{=}\mathbf{c}^{n^f}\}$	// New frame
$\{r_2^*{=}\mathbf{c}^{f}!\boldsymbol{X}\}$	**cspt** r_1	$\{r_1{,}r_2^*{=}\mathbf{c}^{f}!\boldsymbol{X}\}$	// Copy SP to reg. r_1
$\{r_1^*{=}\mathbf{c}^{f}!\boldsymbol{Y}; r_2{=}\mathbf{c}^{f}!\boldsymbol{X}\}$	**cspf** r_2	$\{r_1^*{,}r_2{=}\mathbf{c}^{f}!\boldsymbol{X}\}$	// Copy SP from reg. r_2
$\{r_1^*{=}\mathbf{c}^{n^f}!\boldsymbol{Y}; r_2{=}\mathbf{c}^{f}!\boldsymbol{X}\}$	**rspf** r_2	$\{r_1^*{,}r_2{=}\mathbf{c}^{f}!\boldsymbol{X}\}$	// Restore SP from reg. r_2
$\{\ \}$	**nop**	$\{\ \}$	// No-op, do nothing
$\{r_2{=}\boldsymbol{x}\}$	**mov** $r_1\ r_2$	$\{r_1{,}r_2{=}\boldsymbol{x}\}$	// Copy from reg. r_2
$\{r_2{=}\mathbf{c}^{f}!\boldsymbol{X}\}$	**addaiu** $r_1\ r_2\ n$	$\{r_1{=}\mathbf{c}^{0}; r_2{=}\mathbf{c}^{f}!\boldsymbol{X}\}$	// Arithmetic add

Notation. The $\boldsymbol{X}$, $\boldsymbol{Y}$, etc stand for a set of offsets $!n_1!n_2!\ldots$, for literal natural numbers n. The stack frame size (or ‘tower of stack frame sizes’) f is a literal natural number (or finite sequence of natural numbers). The $\boldsymbol{x}$, $\boldsymbol{y}$, etc stand for any type (something that can appear on the right of an equals sign).

Generalising the stack offset ‘4’ back to n, and generalising registers **gp** and **sp** to r_1 and r_2 respectively, one obtains exactly the small-step signature listed for instruction **put** $r_1\ n$. Registers whose annotations are not mentioned in this signature have bindings that are unaffected by the instruction.

Small-step annotations $\{\Theta\}\ \kappa\ \{\Psi\}$ for an instruction ι at address a with a disassembly κ generate a so-called ‘big step’ rule

$$\frac{T\ \triangleright\ \{\Psi\}\ a+4\ \{\Phi\}}{T\ \triangleright\ \{\Theta\}\ a\ \{\Phi\}}[a \mid \iota\ /\ \kappa]$$

in which Φ is the final annotation at program end and T denotes a list of big-step annotations $\{\Psi\}\ a\ \{\Phi\}$, one for each instruction address a in the program (note that, in consequence, branches within the program must get the same annotation at convergence as there is only one annotation there). Thus the big-step rule is an inference about what *theory* T contains. The rule above says that if $\{\Psi\}\ a+4\ \{\Phi\}$ is in theory T, then so is $\{\Theta\}\ a\ \{\Phi\}$. The label justifies the inference by the fact that instruction ι is at address a, and disassembly κ has been chosen for it.

The big-step rules aim to generate a ‘covering’ theory T for each program. That is, an annotation before every (reachable) instruction, and thus an annotation *between* every instruction. The rule above tells one how to extend by one further instruction a theory that is growing from the back of the program towards the front.

Where does theory construction start? It is with the big-step rule for the final **jr ra** instruction that classically ends a subroutine. The action of this instruction is to jump back to the ‘return address’ stored in the **ra** register (or another designated register).

The annotation for it says that there was a program address (an 'uncalculatable value', $\mathbf{u}^0$) in the **ra** register before it ran (and it is still there after), and requires no hypotheses:

$$\frac{}{T \ \triangleright \ \{r{=}\mathbf{u}^0\}\ a\ \{r{=}\mathbf{u}^0\}}[a \mid \ \textbf{jr}\ r \ /\ \textbf{return}]$$

The '0' superscript indicates that the address may not be used as a base for offset memory accesses; that would access program instructions if it were allowed. Calling code conventionally places the return address in the **ra** register prior to each subroutine call.

There are just three more big-step rules, corresponding to each of the instructions that cause changes in the flow of control in a program. Jumps (unconditional branches) are handled by a rule that refers back to the target of the jump:

$$\frac{T \ \triangleright \ \{\Theta\}\ b\ \{\Phi\}}{T \ \triangleright \ \{\Theta\}\ a\ \{\Phi\}}[a \mid \textbf{j}\ b \ /\ \textbf{goto}\ b]$$

This rule propagates the annotation at the target b of the jump back to the source a. At worst a guess at the fixpoint is needed.

The logic of branch instructions (conditional jumps) at a says that the outcome of going down a branch to b or continuing at $a + 4$ must be the same. But the instruction **bnez** $r\ b$ ('branch to address b if register r is nonzero, else continue') and variants first require the value in the register r to be tested, so it is pre-marked with **c** ('calculatable'):

$$\frac{T \ \triangleright \ \{r{=}\mathbf{c}^f!\boldsymbol{X};\Theta\}\ b\ \{\Phi\} \quad T \ \triangleright \ \{r{=}\mathbf{c}^f!\boldsymbol{X};\Theta\}\ a+4\ \{\Phi\}}{T \ \triangleright \ \{r{=}\mathbf{c}^f!\boldsymbol{X};\Theta\}\ a\ \{\Phi\}}[a \mid \textbf{bnez}\ r\ b \ /\ \textbf{ifnz}\ r\ b]$$

The case $b < a$ (backward branch) requires a guess at a fixpoint as it does for jump. The annotated incremental history f, likely none, of the value in the tested register is irrelevant here, but it is maintained through the rule. The set of offsets $\boldsymbol{X}$ already written to is also irrelevant here, but it is maintained through the rule.

The RISC **jal** b machine code instruction implements standard imperative programming language subroutine calls. It puts the address of the next instruction in the **ra** register (the 'return address') and jumps to the subroutine at address b. The calling code will have saved the current return address on the stack before the call. The callee code will return to the caller by jumping to the address in the **ra** register with **jr ra**, and the calling code will then restore its own return address from the stack.

Because of **jal**'s action in filling register **ra** with a program address, **ra** on entry to the subroutine at b must already have a $\mathbf{u}^0$ annotation, indicating an unmodifiable value that cannot even be used for memory access. And because the same subroutine can be called from many different contexts, we need to distinguish the annotations per call site and so we use a throwaway lettering T' to denote those annotations that derive from the call of b from site a. The general rule is:

$$\frac{T' \ \triangleright \ \{\mathbf{ra}{=}\mathbf{u}^0;\Psi\}\ b\ \{\Theta\} \qquad T \ \triangleright \ \{\Theta\}\ a+4\ \{\Phi\}}{T \ \triangleright \ \{\Psi\}\ a\ \{\Phi\}}[a \mid \textbf{jal}\ b \ /\ \textbf{gosub}\ b]$$

The '0' superscript means that memory accesses via the return address as base address for **lw**/**sw** are not allowed; that would access the program instructions. The stack pointer register has not been named, but it must be distinct from the **ra** register.

We have found it useful to apply extra constraints at subroutine calls. We require (i) that each subroutine return the stack to the same state it acquired it in (this is not a universal convention), and (ii) that a subroutine make and unmake all of its own local stack frame (again, not a universal convention). That helps a Prolog implementation of the verification logic start from a definitely known state at the end of each subroutine independent of the call context – namely, that the local stack frame at subroutine end (and beginning) is size zero. These constraints may be built into the **jal** rule as follows:

$$\frac{T' \triangleright \{\mathbf{ra}{=}\mathbf{u}^0; r^*{=}\mathbf{c}^0!\boldsymbol{X}, \Psi\}\ b\ \{r^*{=}\mathbf{c}^0!\boldsymbol{Y}; \Theta\} \quad T \triangleright \{r^*{=}\mathbf{c}^f!\boldsymbol{Y}; \Theta\}\ a{+}4\ \{\Phi\}}{T \triangleright \{r^*{=}\mathbf{c}^f!\boldsymbol{X}; \Psi\}\ a\ \{\Phi\}}$$

The requirement (i) is implemented by returning the stack pointer in the same register (r^* with the same r on entry and return) and with no stack cells visible in the local stack frame handed to the subroutine and handed back by the subroutine (the two 0s). The requirement (ii) is implemented by setting the local stack frame on entry to contain no stack, just the general purpose registers, which forces the subroutine to make its own stack frame to work in. Other calling conventions require other rule refinements.

As noted, the small-step and big-step rules can be read as a Prolog program with variables the bold-faced offsets variables ***X***, ***Y***, etc, and type variables ***x***, ***y***, etc.

7 Example annotation

Below is the annotation of the simple main routine of a Hello World program that calls 'printstr' with the Hello World string address as argument, then calls 'halt'. The code was emitted by a standard compiler (*gcc*) and modified by hand to be safe against aliasing, so some compiler 'quirks' are still visible. The compiler likes to preserve the **fp** register content across subroutine calls, for example, even though it is not used here.

The functionality is not at issue here, but, certainly, knowing what each instruction does allows the annotation to be inferred by an annotator without reference to rules and axioms. The **li a0** instruction sets the **a0** ('0th argument') register, for example, so the only change in the annotation after the instruction is to the **a0** column. The annotator introduces the string type, $\mathbf{c}^{\ddot{1}}$, into the annotation there, since the instruction sets **a0** to the address of the Hello World string. The annotator assumes that the stack pointer starts in the **sp** register and that 'main' is called (likely from a set-up routine) with a return address in the **ra** register. Changes are marked in grey:

		sp*	**ra**	**a0**	**fp**	**gp**	**v0**	**v1**	(16)	(24)	(28)
main:		$\mathbf{c}^0$	$\mathbf{u}^0$		$\boldsymbol{x}$		$\mathbf{c}^{\ddot{1}}!0$	$\mathbf{c}^0$			
`move gp sp`	cspt gp	$\mathbf{c}^0$	$\mathbf{u}^0$		$\boldsymbol{x}$	$\mathbf{c}^0$	$\mathbf{c}^{\ddot{1}}!0$	$\mathbf{c}^0$			
`addiu sp sp -32`	push 32	$\mathbf{c}^{32^0}$	$\mathbf{u}^0$		$\boldsymbol{x}$	$\mathbf{c}^0$	$\mathbf{c}^{\ddot{1}}!0$	$\mathbf{c}^0$			
`sw ra 28(sp)`	put ra 28	$\mathbf{c}^{32^0}!28$	$\mathbf{u}^0$		$\boldsymbol{x}$	$\mathbf{c}^0$	$\mathbf{c}^{\ddot{1}}!0$	$\mathbf{c}^0$			$\mathbf{u}^0$
`sw fp 24(sp)`	put fp 24	$\mathbf{c}^{32^0}!24!28$	$\mathbf{u}^0$		$\boldsymbol{x}$	$\mathbf{c}^0$	$\mathbf{c}^{\ddot{1}}!0$	$\mathbf{c}^0$		$\boldsymbol{x}$	$\mathbf{u}^0$
`move fp sp`	cspt fp	$\mathbf{c}^{32^0}!24!28$	$\mathbf{u}^0$		$\mathbf{c}^{32^0}!24!28$	$\mathbf{c}^0$	$\mathbf{c}^{\ddot{1}}!0$	$\mathbf{c}^0$		$\boldsymbol{x}$	$\mathbf{u}^0$
`sw gp 16(sp)`	put gp 16	$\mathbf{c}^{32^0}!16!24!28$	$\mathbf{u}^0$		$\mathbf{c}^{32^0}!24!28$	$\mathbf{c}^0$	$\mathbf{c}^{\ddot{1}}!0$	$\mathbf{c}^0$	$\mathbf{c}^0$	$\boldsymbol{x}$	$\mathbf{u}^0$
`li a0 <helloworld>`	newx a0 ... 1	$\mathbf{c}^{32^0}!16!24!28$	$\mathbf{u}^0$	$\mathbf{c}^{\ddot{1}}$	$\mathbf{c}^{32^0}!24!28$	$\mathbf{c}^0$	$\mathbf{c}^{\ddot{1}}!0$	$\mathbf{c}^0$	$\mathbf{c}^0$	$\boldsymbol{x}$	$\mathbf{u}^0$
`jal <printstr>`	gosub ...	$\mathbf{c}^{32^0}!16!24!28$	$\mathbf{u}^0$	$\mathbf{c}^0$	$\mathbf{c}^{32^0}!24!28$	$\mathbf{c}^0$	$\mathbf{c}^0$	$\mathbf{u}^1!0$	$\mathbf{c}^0$	$\boldsymbol{x}$	$\mathbf{u}^0$

`lw gp 16(sp)`	get gp 16	$\mathbf{c}^{32^0}!16!24!28$	$\mathbf{u}^0$	$\mathbf{c}^0$	$\mathbf{c}^{32^0}!24!28$	$\mathbf{c}^0$	$\mathbf{c}^0$	$\mathbf{u}^1!0$	$\mathbf{c}^0$	x	$\mathbf{u}^0$
`jal <halt>`	gosub …	$\mathbf{c}^{32^0}!16!24!28$	$\mathbf{u}^0$	$\mathbf{c}^0$	$\mathbf{c}^{32^0}!24!28$	$\mathbf{c}^0$	$\mathbf{c}^0$	$\mathbf{u}^1!0$	$\mathbf{c}^0$	x	$\mathbf{u}^0$
`nop`											
`lw gp 16(sp)`	get gp 16	$\mathbf{c}^{32^0}!16!24!28$	$\mathbf{u}^0$	$\mathbf{c}^0$	$\mathbf{c}^{32^0}!24!28$	$\mathbf{c}^0$	$\mathbf{c}^0$	$\mathbf{u}^1!0$	$\mathbf{c}^0$	x	$\mathbf{u}^0$
`nop`											
`lw ra 28(sp)`	get ra 28	$\mathbf{c}^{32^0}!16!24!28$	$\mathbf{u}^0$	$\mathbf{c}^0$	$\mathbf{c}^{32^0}!24!28$	$\mathbf{c}^0$	$\mathbf{c}^0$	$\mathbf{u}^1!0$	$\mathbf{c}^0$	x	$\mathbf{u}^0$
`lw fp 24(sp)`	get fp 24	$\mathbf{c}^{32^0}!16!24!28$	$\mathbf{u}^0$	$\mathbf{c}^0$	x	$\mathbf{c}^0$	$\mathbf{c}^0$	$\mathbf{u}^1!0$	$\mathbf{c}^0$	x	$\mathbf{u}^0$
`move sp gp`	rspf gp	$\mathbf{c}^0$	$\mathbf{u}^0$	$\mathbf{c}^0$	x	$\mathbf{c}^0$	$\mathbf{c}^0$	$\mathbf{u}^1!0$	$\mathbf{c}^0$	x	$\mathbf{u}^0$
`jr ra`	return	$\mathbf{c}^0$	$\mathbf{u}^0$	$\mathbf{c}^0$	x	$\mathbf{c}^0$	$\mathbf{c}^0$	$\mathbf{u}^1!0$	$\mathbf{c}^0$	x	$\mathbf{u}^0$
helloworld:	⟨string data⟩										

That the '!' annotations are always less than the bottom element of the tower on the stack pointer annotation means that no aliasing occurs. Reads are at an offset already marked with a '!', hence within the same range that writes are constrained to.

The 'halt' subroutine does not use the stack pointer; its function is to write a single byte to the hard-coded I/O-mapped address of a system peripheral. The annotation for register **v1** on output is the taint left by that write.

```
halt:                                  # zero = c^0; ra = u^0
li v1 0xb0000x10    newh v1 ... 1      # v1 = u^1; zero = c^0; ra = u^0
sb zero 0(v1)       sbth v1 0(v1)      # v1 = u^1!0; zero = c^0; ra = u^0
jr ra               return             # v1 = u^1!0; zero = c^0; ra = u^0
```

The **zero** register is conventionally kept filled with the zero word in RISC architectures.

The *printstr* routine takes a string pointer as argument in register **a0**. A requirement that registers **v0**, **v1** have certain types on entry is an artifact of annotation. Since '$B' comes after writes to **v0**, **v1**, those two registers are bound to types at that point. The forward jump (**j**) to '$B' forces the same annotations at the jump instruction as at the target. But, at the jump, no write to **v0**, **v1** has yet taken place, so we are obliged to provide the types of **v0**, **v1** at entry. The table below is constructed using the same display convention as the table for *main*.

			sp*	**fp**	**ra**	**a0**	**gp**	**v0**	**v1**	(12)	(20)	(24)	(28)
printstr:		#	$\mathbf{c}^0$	x	$\mathbf{u}^0$	$\mathbf{c}^{\ddot{1}}!0$		$\mathbf{c}^{\ddot{1}}!0$	$\mathbf{u}^1!0$				
`move gp sp`	cspt gp	#	$\mathbf{c}^0$	x	$\mathbf{u}^0$	$\mathbf{c}^{\ddot{1}}!0$	$\mathbf{c}^0$	$\mathbf{c}^{\ddot{1}}!0$	$\mathbf{u}^1!0$				
`addiu sp sp -32`	push 32	#	$\mathbf{c}^{32^0}$	x	$\mathbf{u}^0$	$\mathbf{c}^{\ddot{1}}!0$	$\mathbf{c}^0$	$\mathbf{c}^{\ddot{1}}!0$	$\mathbf{u}^1!0$				
`sw ra 24(sp)`	put ra 24	#	$\mathbf{c}^{32^0}!24$	x	$\mathbf{u}^0$	$\mathbf{c}^{\ddot{1}}!0$	$\mathbf{c}^0$	$\mathbf{c}^{\ddot{1}}!0$	$\mathbf{u}^1!0$			$\mathbf{u}^0$	
`sw fp 20(sp)`	put fp 20	#	$\mathbf{c}^{32^0}!20!24$	x	$\mathbf{u}^0$	$\mathbf{c}^{\ddot{1}}!0$	$\mathbf{c}^0$	$\mathbf{c}^{\ddot{1}}!0$	$\mathbf{u}^1!0$		x	$\mathbf{u}^0$	
`move fp sp`	cspt fp	#	$\mathbf{c}^{32^0}!20!24$	$\mathbf{c}^{32^0}!20!24$	$\mathbf{u}^0$	$\mathbf{c}^{\ddot{1}}!0$	$\mathbf{c}^0$	$\mathbf{c}^{\ddot{1}}!0$	$\mathbf{u}^1!0$		x	$\mathbf{u}^0$	
`sw gp 12(sp)`	put gp 12	#	$\mathbf{c}^{32^0}!12!20!24$	$\mathbf{c}^{32^0}!20!24$	$\mathbf{u}^0$	$\mathbf{c}^{\ddot{1}}!0$	$\mathbf{c}^0$	$\mathbf{c}^{\ddot{1}}!0$	$\mathbf{u}^1!0$	$\mathbf{c}^0$	x	$\mathbf{u}^0$	
`sw a0 28(sp)`	put a0 28	#	$\mathbf{c}^{32^0}!12!20!24!28$	$\mathbf{c}^{32^0}!20!24$	$\mathbf{u}^0$	$\mathbf{c}^{\ddot{1}}!0$	$\mathbf{c}^0$	$\mathbf{c}^{\ddot{1}}!0$	$\mathbf{u}^1!0$	$\mathbf{c}^0$	x	$\mathbf{u}^0$	$\mathbf{c}^{\ddot{1}}!0$
`move a0 zero`	mov a0 zero	#	$\mathbf{c}^{32^0}!12!20!24!28$	$\mathbf{c}^{32^0}!20!24$	$\mathbf{u}^0$	$\mathbf{c}^0$	$\mathbf{c}^0$	$\mathbf{c}^{\ddot{1}}!0$	$\mathbf{u}^1!0$	$\mathbf{c}^0$	x	$\mathbf{u}^0$	$\mathbf{c}^{\ddot{1}}!0$
`j ⟨$B⟩`	j ⟨$B⟩	#											
$A:		#	$\mathbf{c}^{32^0}!12!20!24!28$	$\mathbf{c}^{32^0}!20!24$	$\mathbf{u}^0$	$\mathbf{c}^0$	$\mathbf{c}^0$	$\mathbf{c}^0$	$\mathbf{u}^1!0$	$\mathbf{c}^0$	x	$\mathbf{u}^0$	$\mathbf{c}^{\ddot{1}}!0$
`lw v0 28(sp)`	get v0 28	#	$\mathbf{c}^{32^0}!12!20!24!28$	$\mathbf{c}^{32^0}!20!24$	$\mathbf{u}^0$	$\mathbf{c}^0$	$\mathbf{c}^0$	$\mathbf{c}^{\ddot{1}}!0$	$\mathbf{u}^1!0$	$\mathbf{c}^0$	x	$\mathbf{u}^0$	$\mathbf{c}^{\ddot{1}}!0$
`nop`	nop	#											
`lb v0 0(v0)`	getbx v0 0(v0)	#	$\mathbf{c}^{32^0}!12!20!24!28$	$\mathbf{c}^{32^0}!20!24$	$\mathbf{u}^0$	$\mathbf{c}^0$	$\mathbf{c}^0$	$\mathbf{c}^0$	$\mathbf{u}^1!0$	$\mathbf{c}^0$	x	$\mathbf{u}^0$	$\mathbf{c}^{\ddot{1}}!0$
`move v1 v0`	mov v1 v0	#	$\mathbf{c}^{32^0}!12!20!24!28$	$\mathbf{c}^{32^0}!20!24$	$\mathbf{u}^0$	$\mathbf{c}^0$	$\mathbf{c}^0$	$\mathbf{c}^0$	$\mathbf{c}^0$	$\mathbf{c}^0$	x	$\mathbf{u}^0$	$\mathbf{c}^{\ddot{1}}!0$
`lw v0 28(sp)`	get v0 28	#	$\mathbf{c}^{32^0}!12!20!24!28$	$\mathbf{c}^{32^0}!20!24$	$\mathbf{u}^0$	$\mathbf{c}^0$	$\mathbf{c}^0$	$\mathbf{c}^{\ddot{1}}!0$	$\mathbf{c}^0$	$\mathbf{c}^0$	x	$\mathbf{u}^0$	$\mathbf{c}^{\ddot{1}}!0$
`addiu v0 v0 1`	step v0 1	#	$\mathbf{c}^{32^0}!12!20!24!28$	$\mathbf{c}^{32^0}!20!24$	$\mathbf{u}^0$	$\mathbf{c}^0$	$\mathbf{c}^0$	$\mathbf{c}^{\ddot{1}}!0$	$\mathbf{c}^0$	$\mathbf{c}^0$	x	$\mathbf{u}^0$	$\mathbf{c}^{\ddot{1}}!0$

`sw v0 28(sp)`	put v0 28	# $\mathbf{c}^{32^0}!12!20!24!28$	$\mathbf{c}^{32^0}!20!24$	$\mathbf{u}^0$	$\mathbf{c}^0$	$\mathbf{c}^0$	$\mathbf{c}^{\ddot{1}}!0$	$\mathbf{c}^0$	$\mathbf{c}^0$	x	$\mathbf{u}^0$	$\mathbf{c}^{\ddot{1}}!0$
`move a0 v1`	mov a0 v1	# $\mathbf{c}^{32^0}!12!20!24!28$	$\mathbf{c}^{32^0}!20!24$	$\mathbf{u}^0$	$\mathbf{c}^0$	$\mathbf{c}^0$	$\mathbf{c}^{\ddot{1}}!0$	$\mathbf{c}^0$	$\mathbf{c}^0$	x	$\mathbf{u}^0$	$\mathbf{c}^{\ddot{1}}!0$
`jal` ⟨printchar⟩	gosub printchar	# $\mathbf{c}^{32^0}!12!20!24!28$	$\mathbf{c}^{32^0}!20!24$	$\mathbf{u}^0$	$\mathbf{c}^0$	$\mathbf{c}^0$	$\mathbf{c}^{\ddot{1}}!0$	$\mathbf{u}^1!0$	$\mathbf{c}^0$	x	$\mathbf{u}^0$	$\mathbf{c}^{\ddot{1}}!0$
`lw gp 12(sp)`	get gp 12	# $\mathbf{c}^{32^0}!12!20!24!28$	$\mathbf{c}^{32^0}!20!24$	$\mathbf{u}^0$	$\mathbf{c}^0$	$\mathbf{c}^0$	$\mathbf{c}^{\ddot{1}}!0$	$\mathbf{u}^1!0$	$\mathbf{c}^0$	x	$\mathbf{u}^0$	$\mathbf{c}^{\ddot{1}}!0$
$B:		#										
`lw v0 28(sp)`	get v0 28	# $\mathbf{c}^{32^0}!12!20!24!28$	$\mathbf{c}^{32^0}!20!24$	$\mathbf{u}^0$	$\mathbf{c}^0$	$\mathbf{c}^0$	$\mathbf{c}^{\ddot{1}}!0$	$\mathbf{u}^1!0$	$\mathbf{c}^0$	x	$\mathbf{u}^0$	$\mathbf{c}^{\ddot{1}}!0$
`lb v0 0(v0)`	getbx v0 0(v0)	# $\mathbf{c}^{32^0}!12!20!24!28$	$\mathbf{c}^{32^0}!20!24$	$\mathbf{u}^0$	$\mathbf{c}^0$	$\mathbf{c}^0$	$\mathbf{c}^0$	$\mathbf{u}^1!0$	$\mathbf{c}^0$	x	$\mathbf{u}^0$	$\mathbf{c}^{\ddot{1}}!0$
`bnez v0` ⟨$A⟩	bnez v0 ⟨$A⟩	# $\mathbf{c}^{32^0}!12!20!24!28$	$\mathbf{c}^{32^0}!20!24$	$\mathbf{u}^0$	$\mathbf{c}^0$	$\mathbf{c}^0$	$\mathbf{c}^0$	$\mathbf{u}^1!0$	$\mathbf{c}^0$	x	$\mathbf{u}^0$	$\mathbf{c}^{\ddot{1}}!0$
`move sp fp`	cspf fp	# $\mathbf{c}^{32^0}!20!24$	$\mathbf{c}^{32^0}!20!24$	$\mathbf{u}^0$	$\mathbf{c}^0$	$\mathbf{c}^0$	$\mathbf{c}^0$	$\mathbf{u}^1!0$	$\mathbf{c}^0$	x	$\mathbf{u}^0$	$\mathbf{c}^{\ddot{1}}!0$
`lw ra 24(sp)`	get ra 24	# $\mathbf{c}^{32^0}!20!24$	$\mathbf{c}^{32^0}!20!24$	$\mathbf{u}^0$	$\mathbf{c}^0$	$\mathbf{c}^0$	$\mathbf{c}^0$	$\mathbf{u}^1!0$	$\mathbf{c}^0$	x	$\mathbf{u}^0$	$\mathbf{c}^{\ddot{1}}!0$
`lw fp 20(sp)`	get fp 20	# $\mathbf{c}^{32^0}!20!24$	x	$\mathbf{u}^0$	$\mathbf{c}^0$	$\mathbf{c}^0$	$\mathbf{c}^0$	$\mathbf{u}^1!0$	$\mathbf{c}^0$	x	$\mathbf{u}^0$	$\mathbf{c}^{\ddot{1}}!0$
`move sp gp`	rspf gp	# $\mathbf{c}^0$	x	$\mathbf{u}^0$	$\mathbf{c}^0$	$\mathbf{c}^0$	$\mathbf{c}^0$	$\mathbf{u}^1!0$	$\mathbf{c}^0$	x	$\mathbf{u}^0$	$\mathbf{c}^{\ddot{1}}!0$
`jr ra`	return	# $\mathbf{c}^0$	x	$\mathbf{u}^0$	$\mathbf{c}^0$	$\mathbf{c}^0$	$\mathbf{c}^0$	$\mathbf{u}^1!0$	$\mathbf{c}^0$	x	$\mathbf{u}^0$	$\mathbf{c}^{\ddot{1}}!0$

The 'printchar' subroutine writes a character received in register **a0** to the hard-coded address of a printer device:

```
printchar:                              # a0 = c^0; ra = u^0
li v1 0xb0000000    newh v1 ... 1       # v1 = u^1; a0 = c^0; ra = u^0
sb a0 0(v1)         sbth a0 0(v1)       # v1 = u^1!0; a0 = c^0; ra = u^0
jr ra               return              # v1 = u^1!0; a0 = c^0; ra = u^0
```

Like *halt*, it does not use the stack pointer.

8 How does annotation ensure aliasing does not happen?

How to ensure memory aliasing does not happen is intuitively simple: make sure that each address used can have been calculated in only one way. There are in principle two constraints that can be enforced directly via annotation and which will have this effect:

(i) Both stack reads and writes with **get** and **put** may be restricted to offsets n that lie in the range permitted by the local stack frame size (look for a stack pointer tower $m^{\cdot^{\cdot}}$ on the annotation before the instruction, with $0 \leq n \leq m-4$);
(ii) stack reads with **get** may be restricted to offsets n at which writes with **put** have already taken place (look for a $!n$ mark on the annotation before the instruction).

Similarly for strings and arrays. It is (i) that makes memory aliasing impossible, but (ii) is also useful because it (a) reduces (i) to be required on writes alone, and (b) prevents 'read before write' faults. Without (i), code could validly try to access an element of the caller's frame, and that would fail because of aliasing via two distinct calculations for the same address, from caller's and callee's frames respectively.

If these constraints are satisfied, we argue as follows that memory-aliasing cannot occur. The base address used for access via the RISC **lw** or **sw** instructions is either:

1. The stack pointer (disassembly of the access instruction is to **put**, **get**, **putb**, **getb**);
2. the base address of a string, incremented several times by the string increment (the disassembly is to **putx**, **getx**, **putbx**, **getbx**);
3. the base address of an array (the disassembly is to **swth**, **sbth**, **lwfh**, **lbfh**).

and the offset in the instruction is in the first case less than the stack frame size, in the second case less than the string increment, and in the third case less than the array size.

Why are these and no other case possible? Firstly, if the program is annotated, then every use of a base address for the underlying machine code **lw** and **sw** instructions matches exactly one of these cases, because the annotation rules have no other option.

Next we claim that the annotations on a program are *sound*. This is a technical claim that we cannot formally substantiate here that says that in an annotated program the annotations around each instruction reflect what the instruction does computationally. The full statement requires a model of each instruction's semantics as a state-to-state transformation (given in Appendix A) and a proof that the big-step rules of Section 6 express those semantics. Given that, the three cases above for the base address used in a **lw** and **sw** instruction may be characterized thus:

1. It is the stack pointer, which is marked with an asterisk in the annotatiion and typed with $\mathbf{c}^f$ where the tower f consists of the sizes of current and calling stack frames;
2. it is a string pointer, which is typed with $\mathbf{c}^{\ddot{m}}$ in the annotation and is equal to the base address of the string plus a finite number of increments m;
3. it is an array pointer, which is typed with $\mathbf{u}^m$ in the annotation and is equal to the base address of the array, which is of size m.

In each of those three cases, the offset used in the **lw** or **sw** instruction is only permitted by the annotation to lie in the range 0 to $m - 4$, where m is respectively the current frame size, the string step size, and the array size. The first of these cases implements condition (i), and the second and third implement the equivalent condition for strings and arrays respectively. I.e., there is only one calculation possible for each address used.

Similar arguments hold for byte-wise access via **lb** and **sb**. In addition, however, one must require that menory areas accessed via these instructions are not also accessed via **lw** and **sw**, in order to avoid different calculations for the addresses of the individual bytes in a word. The simplest way to ensure that is to forbid use of **lb** and **sb** entirely, relying instead on **lw** and **sw** plus arithmetic operations to extract the byte. The next simplest alternative is to allow **lb** and **sb** only on strings with step size less than 4 and arrays of size less than 4, which word-wise instructions are forbidden from accessing by the annotation rules.

9 Conclusion and Future Work

We have set out a method of annotation that can ensure that a RISC machine-code program is safe against 'hardware' aliasing. We model aliasing as introduced by the use of different arithmetic calculations for the same memory address, and successful annotation guarantees that a unique calculation will be used at run-time for the address of each execution stack, string or array element accessed by the program. Annotation also means disassembling the machine code to a slightly higher level assembly language, for a stack machine, and a human being is required to certify that the disassembly matches the programmer's intentions.

Note that one may add disassembly rules to the system that are (deliberately) semantically wrong, with the aim of correcting the code. For example, one may choose to

(incorrectly) disassemble the RISC **addiu sp sp** 32 instruction to a stack machine **pop** instruction. The RISC instruction is not a correct implementation of the higher level instruction in an aliasing context, although it was likely intended to be. But one may then replace the original RISC code with a correct implementation.

Also note that the equational annotations here may be generalised to quite arbitrary first-order predicates. It also appears that our system of types may be generalised to arrays of arrays and strings of strings, etc, which offers the prospect of a static analysis technology that can follow pointers.

References

1. Michael Barr. *Programming Embedded Systems in C and C++*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1st edition, 1998.
2. J. P. Bowen. Formal specification of the ProCoS/Safemos instruction set. *Micoprocessors and Microsystems*, 14(10):637–643, December 1990.
3. J. P. Bowen and P. T. Breuer. Decompilation. In H. van Zuylen, editor, *The REDO Compendium: Reverse Engineering for Software Maintenance*, chapter 10, pages 131–138. John Wiley & Sons, 1993.
4. P. T. Breuer and J. P. Bowen. Decompilation: The enumeration of types and grammars. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1613–1647, September 1994.
5. P. T. Breuer and J. P. Bowen. Typed assembler for a RISC crypto-processor. In *Proc. ESSOS'12: Intl. Symp. on Engineering Secure Software and Systems*, number 7159 in LNCS, pages 22–29. Springer, February 2012.
6. P. T. Breuer and J. P. Bowen. A fully homomorphic crypto-processor design: Correctness of a secret computer. In *Proc. ESSOS'13: Intl. Symp. on Engineering Secure Software and Systems*, number 7781 in LNCS. Springer, February 2013.
7. F. H. Fischer, V. Sindalovsky, and S. A. Segan. Memory aliasing method and apparatus, August 20 2002. US Patent 6,438,672.
8. Bruce Jacob, 2004. http://www.eng.umd.edu/~blj/RiSC/RiSC-isa.pdf.
9. D. A. Patterson. Reduced instruction set computers. *Communications of the ACM*, 28(1):8–21, January 1985.
10. T. Sato. Speculative resolution of ambiguous memory aliasing. In *Innovative Architecture for Future Generation High-Performance Processors and Systems*, pages 17–26. IEEE, 1997.
11. Malcolm J. Wing and Edmund J. Kelly. Method and apparatus for aliasing memory data in an advanced microprocessor, July 20 1999. US Patent 5,926,832.

APPENDIX – *NOT FOR PUBLICATION*

A Motivating semantics

We will restrict the commentary here to the ten instructions from the 32-bit RISC instruction set architecture shown in Table 1. These are also the elements of a tiny RISC-16 machine code/assembly language [8]. Because of their role in RISC-16, we know that they form a complete set that can perform arbitrary computations.

We suppose in this paper that programs are such that the stack pointer always remains in the **sp** register. Copies may be made of it elsewhere using the **move** (copy) instruction, and it may be altered in situ using the **addiu** instruction. Adding a negative amount increases the stack size, and stack conventionally grows top-down in the address space. We also suppose that the *return address* pointer is always in the source register r at the point where a ('jump register') **jr** r instruction is executed, so that the latter may be interpreted as a stack machine **return** instruction.

A program induces a set of *dataflow traces* through registers. A dataflow trace is a unique path through registers and stack memory cells that traces movement of data. The segments of the trace may be labelled with *events* as detailed below, signifying data transformation, or they may be unlabelled, signifying transfer without transformation. Each trace starts with the introduction of a value into a register, either from the instruction itself in the case of the **li** and the source is shown as a blank triangle, or by hypothesis at the start of a subroutine and the source is shown as a vertical bar.

li $r\ a$	hypothesis
newh $r\ a\ n$	$[r = \tau]$

The left hand diagram above shows the introduction of the address a of an array of size n into register r, the **li** machine code instruction having been disassembled to **newh**. The indices of those elements already written to the array are recorded in the set $\boldsymbol{X}$. Usually that is the full set of indices up to n and the address is that of an array written earlier. The label on the arrow is a *annotated type* (Table 3), indicating an introduction event. The annotated type brought in with the array pointer introduction is

$$\mathbf{u}^n!\boldsymbol{X}$$

standing for an address that may not subsequently be altered ('**u**', or 'uncalculatable') of n bytes of memory, that has been written to at each of the offsets in the set $\boldsymbol{X}$.

If the **li** instruction is instead interpreted as introducing the address a of a 'string-like' object, then the annotated type brought in is

$$\mathbf{c}^{\ddot{n}}!\boldsymbol{X}$$

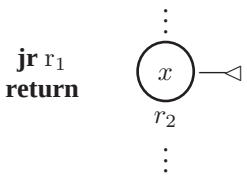
standing for an address that may be altered ('**c**', or 'calculatable') and stepped in increments of n bytes. The $\boldsymbol{X}$ again stands for a set of offsets from the base (up to n bytes) at

which the structure has been written. The same pattern $\boldsymbol{X}$ applies at every increment n along the string. The form '$\ddot{n}$' is meant to be understood as '$n^{n^{\cdot^{\cdot^{\cdot}}}}$', with the n repeated an indefinite number of times. This may be viewed as a variant of the annotated type

$$\mathbf{c}^{n_1^{\cdot^{\cdot^{n_k}}}}\,!\boldsymbol{X}$$

that the stack pointer is associated with (for some finite sequence n_1,...,n_k as superscripts) and which records a historical sequence of local stack frames created one within the scope of the other culminating in a current stack frame of size n_1 bytes.

Each trace that we consider ends with the **return** from a subroutine call. Only traces that have reached some register r_2 at that moment are 'properly terminated'. Any other trace (i.e., one that has reached a stack cell) is not considered further. In the call protocol that we allow here, the subroutine's local frame is created at entry and destroyed at return and the data in it is not shared with the caller:



We aim to constrain the possible sequences of events along traces. The events are:

1. $!k$ for a write at stack offset k with **put** $r\ k$ (or **putx**, **swth** for strings, arrays);
2. $?k$ for a read at stack offset k with **get** $r\ k$ (or **getx**, **lwfh** for strings, arrays);
3. $\mathbf{u}^n$ for the introduction of an array data address a via **newh** $r\ a\ n$;
4. $\mathbf{c}^n$ for the introduction of a 'string' data address a via **newx** $r\ a\ n$;
5. τ for the introduction of data of any kind τ 'by hypothesis';
6. $\mathbf{c}^0$ for the production of new data via the **addaiu** or other arithmetic instruction;
7. $n{\uparrow}$ for the creation of a new stack frame of size n bytes via **push** n;
8. $n{\downarrow}$ for restoring the previous stack frame, terminating a frame of size n bytes via **rspf** n. (or **stepx** when moving along a string);
9. nothing, for maintaining the data as-is or copying it.

An event does not always occur on the link one might expect: for example, reading data to r_1 with **lw** $r_1\ 4k(\mathbf{sp})$ evokes an event on a '**sp** to **sp**' link in Fig. 2, not on the '(k) to r_1' ('stack slot k to register r_1') link that the data flows along. We wish to enforce the following restrictions. First, on the stack pointer:

(a) every $!k$ and $?k$ event is preceded by a last $n{\uparrow}$ event that has $n - w \geq k \geq 0$ (where w is the number of bytes written), so stack reads and writes do not step outside the local frame of the subroutine;
(b) every $?k$ event is preceded by a $!k$ event that takes place after the last preceding $n{\uparrow}$ event, so every read is of something that has been written;
(c) every $n{\downarrow}$ event is preceded by a last $m{\uparrow}$ event with $m = n$, and so on recursively so stack pushes and pops match up like parentheses;

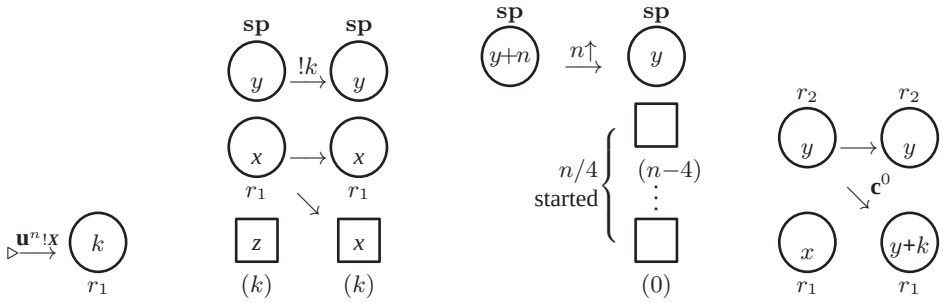


Fig. 1: Dataflow semantics of machine code/assembly language instructions.

(d) no trace containing a **c** or **u** event other than an originating $\mathbf{c}^0$ may eventually pass through the stack pointer register, so the only operations allowed on the stack pointer are shifts up and down;
(e) every $n\uparrow$ event is with $n > 0$.

Secondly, on the traces through registers containing a string pointer:

(a) every $!k$ and $?k$ event is within the bound n established by the introduction $\mathbf{c}^{\ddot{n}}$ on the trace, in that $n - w \geq k \geq 0$, where w is the width of the transferred data;
(b) there is no (b) constraint;
(c) every $n\downarrow$ event is with n equal to the string increment established by the introduction $\mathbf{c}^{\ddot{n}}$ on the trace;
(d) no trace containing any other event than the $\mathbf{c}^{\ddot{n}}$ introduction and subsequent $n\downarrow$ shifts may later pass through the string pointer register, so the only modifications allowed to the string pointer are shifts down;
(e) there is no (e) constraint.

The constraints applied to traces through array pointers are stricter:

(a) every $!k$ and $?k$ event is within the bound n established by the preceding introduction $\mathbf{u}^n$ on the trace, in that $n - w \geq k \geq 0$.
(b) there is no (b) constraint;
(c) there are no $n\downarrow$ or $n\uparrow$ events allowed;
(d) no trace containing any other event than the $\mathbf{u}^n$ introduction may later pass through the array pointer register, so no modifications to the array pointer are allowed;
(e) there is no (e) constraint.

We express these constraints formally below. Starting with the event that introduces an annotated type τ we accumulate a running 'total' annotated type along each trace. The

first two equations and their guards express the constraints on an array pointer. Shifts of the base address are not allowed and reads and writes are restricted to the array bound:

$$\mathbf{u}^n!\boldsymbol{X} \cdot !k = \mathbf{u}^n!(\boldsymbol{X} \cup \{k\}) \qquad n - w \geq k \geq 0 \tag{1}$$
$$\mathbf{u}^n!\boldsymbol{X} \cdot ?k = \mathbf{u}^n!\boldsymbol{X} \qquad \boldsymbol{X} \ni k \geq 0 \tag{2}$$

The next three equations express the constraints on a string pointer. Additionally, over the array pointer equations, shifts-down on (increasing) the pointer are allowed:

$$\mathbf{c}^{\ddot{n}}!\boldsymbol{X} \cdot n{\downarrow} = \mathbf{c}^{\ddot{n}} \qquad n > 0 \tag{3}$$
$$\mathbf{c}^{\ddot{n}}!\boldsymbol{X} \cdot !k = \mathbf{c}^{\ddot{n}}!(\boldsymbol{X} \cup \{k\}) \qquad n - w \geq k \geq 0 \tag{4}$$
$$\mathbf{c}^{\ddot{n}}!\boldsymbol{X} \cdot ?k = \mathbf{c}^{\ddot{n}}!\boldsymbol{X} \qquad \boldsymbol{X} \ni k \geq 0 \tag{5}$$

The next four equations express the constraints on the stack pointer. Additionally, over the string pointer equations, shifts-up on (decreasing) the pointer are allowed. The first two equations make shifts nest like parentheses:

$$\mathbf{c}^f!\boldsymbol{X} \cdot n{\uparrow} = \mathbf{c}^{n^f} \qquad n > 0 \tag{6}$$
$$\mathbf{c}^{n^f}!\boldsymbol{X} \cdot n{\downarrow} = \mathbf{c}^f \qquad n > 0 \tag{7}$$
$$\mathbf{c}^{n^f}!\boldsymbol{X} \cdot !k = \mathbf{c}^{n^f}!(\boldsymbol{X} \cup \{k\}) \qquad n - w \geq k \geq 0 \tag{8}$$
$$\mathbf{c}^{n^f}!\boldsymbol{X} \cdot ?k = \mathbf{c}^{n^f}!\boldsymbol{X} \qquad \boldsymbol{X} \ni k \geq 0 \tag{9}$$

These calculations bind an annotated type to each register and stack cell at each point in the program.

Does the same register get the same type in every trace calculation? Traces converge only after a **nand** (when the type computed is $\mathbf{c}^0$, so 'yes it does' in this case) and after a jump or branch. In these latter two cases we specify:

> The calculated type at the same registers or stack slots must be the same across different traces starting from the same entry point for the programs considered. (*)

The programs in which (*) is true are the only programs we consider. They are programs that re-establish the same pattern of annotated types at each point at every pass through a loop and no matter which path through to a given point is taken.

The annotated types that get bound to registers and stack slots are the values in the states of an *abstract stack machine* whose instruction semantics is described by Figs. 1 and 2. That may be shown to be an abstract interpretation of the instruction trace semantics in a stack machine. That in turn abstracts a machine code processor via disassembly.

Call an attempt in the stack machine to read or write beyond the current local frame *out-of-bounds*. That the abstract stack machine that calculates with annotated types is an abstract interpretation of the stack machine that calculates with integer words means that an out-of-bounds access in the stack machine must evoke a $!k$ or $?k$ event on a trace through the abstract stack machine where k is not bounded by the size n of the last $n{\uparrow}$

event on the trace. But that is forbidden by (1-9) in the abstract stack machine. So if we can verify that (1-9) hold of a program in the abstract stack machine, out-of-bounds accesses cannot happen in the stack machine.

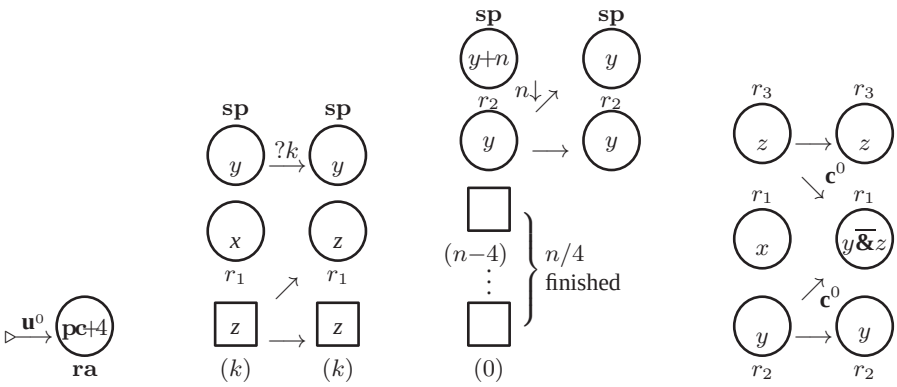
If out-of-bounds accesses in the stack machine cannot happen, then we argue that aliasing cannot happen in the machine code processor. The argument goes as follows: the base address used for access via the RISC **lw** or **sw** instructions must be either

1. the stack pointer (disassembly is to **put**, **get**, **putb**, **getb** and the base address register gets the annotated type $\mathbf{c}^{f}!\boldsymbol{X}$ for some finite tower of frame sizes f);
2. the base address of a string, incremented several times by the string increment (disassembly is to **putx**, **getx**, **putbx**, **getbx** and the base address register gets the annotated type $\mathbf{c}^{\ddot{n}}!\boldsymbol{X}$ for some string step n);
3. the base address of an array (disassembly is to **swth**, **sbth**, **lwfh**, **lbfh** and the base address register gets the annotated type $\mathbf{u}^{n}!\boldsymbol{X}$ for some array size n).

Those are the only annotated types allowed by (1-9) on the abstract stack machine to be bound to the pointer's register at the moment the event $!k$ or $?k$ happens.

In the first case, the offset in the accessing instruction is less than the stack frame size, in the second case less than the string increment, and in the third case less than the array size. Those calculations are the only ones that can be made for the address of the accessed element, and they are each unique. For example, in case 1, the address used is $s + k$, where s is the stack pointer and $0 \leq k \leq n - w$, where n is the local frame size and w is the size of the data accessed. If two such accesses from the same frame are at arithmetically equal address aliases $s + k_1 \equiv s + k_2$ but $s + k_1 \neq s + k_2$ identically. So $k_1 \equiv k_2$ arithmetically but $k_1 \neq k_2$ identically. But k_1 and k_2 are small numbers in the range 0 to n, where n is the frame size. If they cannot be distinguished by the processor arithmetic, then something is deeply wrong with the processor design. Accessing an element of a parent frame with $s_1 + k_1 \equiv s_2 + k_2$ where $s_1 = s_2 - n$ is simply out of the question because k_1 is restricted to the range 0 to n.

We conclude that accessing different aliases of the same address is impossible if the abstract interpretation of the program as set out by Figs. 1 and 2 can be verified to satisfy (1-9).



jal b
gosub b

lw $r_1\ k(\mathbf{sp})$
get $r_1\ k$

move sp r_2
rspf r_2

nand $r_1\ r_2\ r_3$
nand $r_1\ r_2\ r_3$

Fig. 2: Dataflow semantics of four more machine code instructions.