Range Analysis of Microcontroller Code Using Bit-Level Congruences

Jörg Brauer[1], Andy King[2], and Stefan Kowalewski[1]

[1] Embedded Software Laboratory, RWTH Aachen University, Germany
[2] Portcullis Computer Security, Pinner, HA5 2EX, UK

Abstract. Bitwise instructions, loops and indirect data access pose difficult challenges to the verification of microcontroller programs. In particular, it is necessary to show that an indirect write does not mutate registers, which are indirectly addressable. To prove this property, among others, this paper presents a relational binary-code semantics and details how this can be used to compute program invariants in terms of bit-level congruences. Moreover, it demonstrates how congruences can be combined with intervals to derive accurate ranges, as well as information about strided indirect memory accesses.

1 Introduction

Microcontroller assembly code[1] presents different challenges to verification than those posed by programs written in high-level languages. Microcontroller code typically consists of a loop in which input ports are read. Data is then stored and processed – often using bitwise operations – before values are written to output ports. Bitwise operations and control logic formulated in terms of status flags necessitate reasoning at the granularity of bits. This presents one problem.

On hardware such as the ATMEL ATmega16 [1], any verification argument must also pay special attention to the targets of indirect writes[2]. An indirect write is a store operation in which the contents of one register are stored at a target address that is held in another register. On the ATmega family of microcontrollers, registers are reserved locations in the same address space as the SRAM. Thus, it is possible to mutate a register, such as the stack pointer, if the target coincides with the address of the register. One approach to microcontroller verification is to assume that indirect writes never mutate registers [20]. Though appealing in its simplicity, this assumption is dubious for handcrafted assembly code, and it is not unknown for compilation itself to introduce errors [12]. The problem of reasoning about targets is compounded by the fact that indirect writes often arise in loops that are, for example, responsible for data initialisation. Then the same store operation may write to a number of different targets. Another problem is therefore showing that all targets are within range [5].

[1] We often refer to assembly code, although our implementation operates on a disassembled binary, and thus, does not rely on correctness of assemblers and linkers.

[2] We illustrate our method for the ATmega16 platform, but the techniques are easily transferable to other platforms as well as high-level languages.

S. Kowalewski and M. Roveri (Eds.): FMICS 2010, LNCS 6371, pp. 82–98, 2010.
© Springer-Verlag Berlin Heidelberg 2010

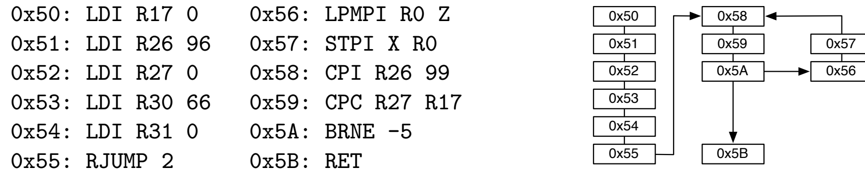

Fig. 1. An initialisation loop for the ATMEL ATmega16

1.1 Illustrative Example

This paper addresses the problem of statically analysing the targets of indirect writes, whilst simultaneously modelling data at the bit-level. Since the set of possible targets cannot be exactly determined statically, we employ abstract interpretation techniques [8] to compute a range of addresses that includes all possible targets. If the enclosing range is suitably tight, it is possible to verify that the registers are not overwritten. Figure 1 illustrates some ATmega16 assembly code. The instructions at locations `0x50` - `0x54` assign 8-bit registers to (decimal) constants. The relative jump passes control to location `0x58`. The `LPMPI R0 Z` instruction first loads `R0` with the contents of the byte at the address in program memory stored in the 16-bit `Z` register, then `Z` is incremented. `Z` is obtained by concatenating the 8-bit registers `R30` and `R31`. Likewise, the 8-bit registers `R26` and `R27` constitute the 16-bit `X` register. `STPI X R0` stores the contents of `R0` into the byte at address `X` and then increments `X`.

The ATmega has a Harvard architecture, and hence, program memory is separate from SRAM. Location `98`, for instance, in program memory is different from location `98` in SRAM. Thus, program memory is accessed with special instructions such as `LPMPI`. Therefore, detecting self-modifying code, which we do not consider, is trivial. The instructions `CPI R26 99` and `CPC R27 R17` compare `X` against `99`, setting the zero flag if `X` equals `99`. Control loops back to location `0x56` iff the zero flag is cleared, that is, if `X` is not equal to `99`. The net effect of the code is to copy the contents of three locations in program memory starting at 66 into the SRAM locations $96 - 98$. This initialises three global variables to constant values.

A non-relational interval analysis as described in [5] can derive that $\texttt{X} \in [96, 99]$ in program location `0x5A`. The interval analyser derives the bound on `X` based on the combination of `CPI`/`CPC` instructions followed by `BRNE`. However, it fails to discover that $\texttt{Z} \in [66, 69]$ and has to assume that the loop body could be entered with values $\texttt{X} \in [96, 98] \wedge \texttt{Z} \in [66, 69]$, $\texttt{X} \in [96, 98] \wedge \texttt{Z} \in [66, 70]$, and so forth, which eventually yields $\texttt{Z} \in [0, 65535]$. If the `CPI`/`CPC` instructions were to restrict `Z` instead of `X`, then the value of `X` were unbounded. This is in fact a well-known drawback of non-relational interval analysis. To resolve this type of imprecision, we combine the results of a relational analysis for equalities with a computationally cheap interval analysis, with the goal of deriving that `X` is incremented only in combination with `Z`, and consequently that $\texttt{X} \in [96, 98] \wedge \texttt{Z} \in [66, 68]$ when the indirect loads/stores are executed.

1.2 Approach

In microcontroller code for the ATmega16 platform, a memory region typically is statically reserved rather than dynamically allocated. Thus, the address of the start of a region that is used as an array is fully determined. Hence, when verifying such code, it is not necessary to use a symbolic name to refer to a memory region: an address will suffice. The force of this is that there is no need to adopt a memory model in which regions with different symbolic names are assumed to be non-interfering. Symbolic memory models are often employed when the position of a region is unknown, as with dynamically allocated memory in C, but this nevertheless compromises soundness [3]. Furthermore, when analysing statically reserved regions, it is even possible to infer a relationship between each address of a region, and the contents of that address.

To represent such relations, we turn to linear congruences [2, 18]. In this classical abstract domain [13], the relationships between variables are described as systems of linear equations of the form $\sum_{i=0}^{n-1} c_i x_i \bmod m = d$, denoted by $\sum_{i=0}^{n-1} c_i x_i \equiv_m d$, where $c_i \in \mathbb{Z}$ are integer coefficients, x_i are variables, $m \in \mathbb{N}$ is a modulus, and $d \in \mathbb{Z}$ is an integer constant. Such a system may have none, one or many solutions, where a solution is an assignment to the values of the n variables $x_0, \ldots, x_{n-1}$ that satisfies each of the equations. For example, the system $u+2v \equiv_{256} 3$ and $v+w \equiv_{256} 1$ has solutions $\{\langle 1+256k_1+2k_3, 1+256k_2-k_3, k_3\rangle \in [0, 255]^3\}$ where $k_1, k_2, k_3 \in \mathbb{Z}$. Such relationships arise between program variables, or memory locations in the case of microcontroller code, because of the modular nature of computer arithmetic. It is therefore natural to consider moduli corresponding to the size of a machine word [18]. Such systems can only represent linear relationships, but not ranges, and therefore, we adopt a more expressive class of congruences based on decomposing variables into their consistent bits [15].

For instance, suppose u is represented by an unsigned byte whose bits are $\langle u_0, \ldots, u_7\rangle$ where $u_i \in \{0, 1\}$ and the value of u is $\sum_{i=0}^{7} 2^i u_i$. Suppose too that v and w are likewise represented by $\langle v_0, \ldots, v_7\rangle$ and $\langle w_0, \ldots, w_7\rangle$. Then the above system can be expressed as $\sum_{i=0}^{7} 2^i(u_i + 2v_i) \equiv_{256} 3$, $\sum_{i=0}^{7} 2^i(v_i + w_i) \equiv_{256} 1$ without any loss of information. It has been shown how such systems can be applied to verify bit-twiddling algorithms [15, 16].

1.3 Contributions

In this paper, we make the following contributions. (1) We deploy congruence systems to derive program invariants for assembly code at the level of bits. (2) Further, we combine intervals [5] and congruence relations to derive accurate ranges. To do so, we present a new algorithm for refining the precision of abstract descriptions in both domains. (3) We show how a contiguous range, such as $[0, 6]$, can be refined to a set of non-contiguous values, such as $\{0, 2, 4, 6\}$, by applying congruences to ranges. (4) To summarise, this paper shows by that it is possible to infer accurate ranges using congruences and intervals, and thereby verify the correctness of microcontroller assembly code.

2 Abstract Domains

This section briefly reviews results on the abstract domains our work builds on, namely intervals and congruences. In the following, let $m = 2^w$ where $w = 8$ is the word-length of the microcontroller, $\mathbb{Z}_m = \{i \in \mathbb{N} \mid 0 \leq i \leq m-1\}$, and let $\mathcal{V} = \{v_0, \ldots, v_{n-1}\}$ be a set of variables for some $n \in \mathbb{N}$. Further, let $\mathcal{P}$ denote the set of program locations (or instructions, equivalently).

2.1 Intervals

The interval abstract domain, probably the most widely used numerical domain, is used to over-approximate the value-sets of memory cells. In case of the 8-bit ATmega16, a memory location can hold a contiguous subset of values in $\mathbb{Z}_m$ defined through its bounds. Denote the domain Int. A partial order on intervals is induced by the subset relation over the concrete value-sets. Then, $(\mathsf{Int}, \subseteq)$ forms a complete lattice with $\bot = \emptyset$ and $\top = \mathbb{Z}_m$. Define auxiliary functions $\mathsf{fst} : \mathsf{Int} \to \mathbb{Z}_m$ and $\mathsf{snd} : \mathsf{Int} \to \mathbb{Z}_m$ that map intervals to their bounds. Abstraction $\alpha_{\mathsf{Int}} : 2^{\mathbb{Z}_m} \to \mathsf{Int}$ and concretisation $\gamma_{\mathsf{Int}} : \mathsf{Int} \to 2^{\mathbb{Z}_m}$ are defined as

$$\alpha_{\mathsf{Int}}(v) = \begin{cases} \emptyset & : \bot \\ [\mathsf{min}(v), \mathsf{max}(v)] & : \text{otherw.} \end{cases} \qquad \gamma_{\mathsf{Int}}(i) = \{z \in \mathbb{Z}_m \mid \mathsf{fst}(i) \leq z \leq \mathsf{snd}(i)\}$$

for $i \in \mathsf{Int}$ and $v \subseteq \mathbb{Z}_m$. An abstract interpretation framework for deriving non-relational interval abstractions of microcontroller code has been described in [5], however, space constraints prevent us from repeating these results here. We assume that for each program location $p \in \mathcal{P}$ and each memory location $v \in \mathcal{V}$, an interval abstraction has been computed, given through a map $I : \mathcal{V} \times \mathcal{P} \to \mathsf{Int}$.

2.2 Congruences

Additionally, our analysis is based on representing Boolean functions as congruence systems. To explain this idea, let $\mathsf{sol}(f)$ denote the set of solutions of a Boolean function f over n propositional variables. Our method relies on the computation of the so-called *congruent closure*, which yields a congruence system c over n bitwise variables such that $\mathsf{sol}(f) \subseteq \mathsf{sol}(c) \cap \mathbb{B}^n$ with $\mathbb{B} = \{0, 1\}$ holds. For example, given a function $f = x_1 \wedge (x_2 \vee x_3)$, we have $\mathsf{sol}(f) = \{\langle 1, 0, 1\rangle, \langle 1, 1, 0\rangle, \langle 1, 1, 1\rangle\}$. Congruent closure, with a fixed modulo of 4, then computes $c = (x_1 \equiv_4 1)$. The solutions of this congruence equation are $\mathsf{sol}(c) \cap \mathbb{B}^3 = \{\langle 1, x_2, x_3\rangle \mid x_2, x_3 \in \mathbb{B}\}$. Note that $\mathsf{sol}(c) \setminus \mathsf{sol}(f) = \{\langle 1, 0, 0\rangle\}$.

Definition 1. The operator $\mathsf{cong} : 2^{\mathbb{B}^{nw}} \to 2^{\mathbb{B}^{nw}}$ is defined:

$$\mathsf{cong}(S) = \left\{ \boldsymbol{x} \in \mathbb{B}^{nw} \;\middle|\; \begin{array}{l} \{\boldsymbol{y}_0, \ldots, \boldsymbol{y}_{k-1}\} \subseteq S \wedge \{\lambda_0, \ldots, \lambda_{k-1}\} \subseteq \mathbb{Z} \wedge \\ \sum_{j=0}^{j<k} \lambda_j \equiv_{2^w} 1 \qquad \wedge\, \boldsymbol{x} \equiv_{2^w} \sum_{j=0}^{j<k} \lambda_j \boldsymbol{y}_j \end{array} \right\}$$

An algorithm for deriving optimal congruent abstractions of Boolean formulae was described by King and Søndergaard [16]. Given a formula φ, the key idea of their method is to derive a congruent abstraction $\alpha_{\mathsf{Cong}}(\varphi)$ through successive calls to a SAT solver. Therefore, their algorithm is similar in spirit to the symbolic implementation of a best transformer as described by Reps et al. [21]. In the following, let Cong denote the domain of bit-level congruences over $\mathcal{V}$.

3 Worked Examples

We illustrate the power of bit-level reasoning using the congruence domain for some illustrative sequences of ATmega16 assembly. The key idea of our approach is to derive a template transfer function for each instruction using SAT solving up-front, and then instantiate the transfer functions to infer program invariants. The invariants are then strengthened with intervals, yielding more precise representations of congruences as well as intervals.

3.1 Reasoning about Bit-Wise Operations

Consider the instruction `EOR R0 R1`, which computes the exclusive-or of registers `R0` and `R1` and stores the result in `R0`. First, a template abstraction of this instruction that does not depend on the concrete registers `R0` and `R1` is synthesised from a Boolean encoding. To express the semantics of `EOR r s`, introduce bit-vectors $\boldsymbol{r}[i]$ and $\boldsymbol{s}[i]$ for the inputs as well as $\boldsymbol{r}'[i]$ and $\boldsymbol{s}'[i]$ for the outputs (with $0 \leq i \leq 7$). Then, `EOR r s` is encoded symbolically as

$$[\![\texttt{EOR r s}]\!] = \textstyle\bigwedge_{i=0}^{7} (\boldsymbol{r}'[i] \leftrightarrow \boldsymbol{r}[i] \oplus \boldsymbol{s}[i] \wedge \boldsymbol{s}'[i] \leftrightarrow \boldsymbol{s}[i])$$

where $\oplus$ denotes the Boolean exclusive-or. By computing the congruent closure of $[\![\texttt{EOR r s}]\!]$ with a modulus of 256, denoted α_{Cong}, we obtain:

$$\alpha_{\mathsf{Cong}}([\![\texttt{EOR r s}]\!]) = \begin{cases} \bigwedge_{i=0}^{7} (128 \cdot \boldsymbol{r}'[i] \equiv_{256} 128 \cdot \boldsymbol{r}[i] + 128 \cdot \boldsymbol{s}[i]) \wedge \\ \bigwedge_{i=0}^{7} \boldsymbol{s}'[i] \equiv_{256} \boldsymbol{s}[i] \end{cases}$$

Note that $\mathsf{sol}(\alpha_{\mathsf{Cong}}([\![\texttt{EOR r s}]\!])) = \mathsf{sol}([\![\texttt{EOR r s}]\!])$, and thus, this congruent transfer function is just as accurate as its Boolean counterpart.

3.2 Relational Composition without Ranges

In the previous example, we have seen how a template abstraction of a single instruction is derived. Here, we consider the program fragment `EOR R0 R1; EOR R1 R0; EOR R0 R1` and the instantiation of templates. In [15], it was shown that best transformers for blocks (sequences of instructions) can be obtained by encoding the sequence propositionally as a whole. Since our goal is to derive range information for different program locations that may be located in the middle of a block, we deviate from following this approach, and combine the obtained transfer functions using relational composition $\circ : \mathsf{Cong} \times \mathsf{Cong} \rightarrow \mathsf{Cong}$.

A template transfer function c, derived analogously to the first example, is instantiated with the corresponding variables $\boldsymbol{r0}$, $\boldsymbol{r1}$, $\boldsymbol{r0'}$, and $\boldsymbol{r1'}$, which amounts to renaming variables in the template. This gives $c_1 = c(\boldsymbol{r0}, \boldsymbol{r1}, \boldsymbol{r0'}, \boldsymbol{r1'})$, $c_2 = c(\boldsymbol{r1}, \boldsymbol{r0}, \boldsymbol{r1'}, \boldsymbol{r0'})$, and $c_3 = c(\boldsymbol{r0}, \boldsymbol{r1}, \boldsymbol{r0'}, \boldsymbol{r1'})$, for instance:

$$c_1 = \bigwedge_{i=0}^{7} (128 \cdot \boldsymbol{r0'}[i] \equiv_{256} 128 \cdot \boldsymbol{r0}[i] + 128 \cdot \boldsymbol{r1}[i]) \wedge \bigwedge_{i=0}^{7} (\boldsymbol{r1'}[i] \equiv_{256} \boldsymbol{r1}[i])$$

To combine the effects of c_1 and c_2, introduce additional disjoint bit-vectors $\boldsymbol{r0''}$ and $\boldsymbol{r1''}$, and put $c_1' = c_1 \wedge \left(\wedge_{i=0}^{7} \boldsymbol{r0''}[i] \equiv_{256} \boldsymbol{r0}[i]'\right) \wedge \left(\wedge_{i=0}^{7} \boldsymbol{r1''}[i] \equiv_{256} \boldsymbol{r1}[i]'\right)$ and $c_2' = c_2 \wedge \left(\wedge_{i=0}^{7} \boldsymbol{r0''}[i] \equiv_{256} \boldsymbol{r0}[i]\right) \wedge \left(\wedge_{i=0}^{7} \boldsymbol{r1''}[i] \equiv_{256} \boldsymbol{r1}[i]\right)$. The net effect of this construction is to relate the outputs of c_1 to the inputs of c_2. Then, define $c_1 \circ c_2 = \exists_{\boldsymbol{r0''}, \boldsymbol{r1''}} (\exists_{\boldsymbol{r0'}, \boldsymbol{r1'}} (c_1') \wedge \exists_{\boldsymbol{r0}, \boldsymbol{r1}} (c_2'))$ where the operation $\exists_{\boldsymbol{X}}(f)$ eliminates the variables $\boldsymbol{X}$ from f using projection. Observe that projection can be implemented by computing upper triangular form after reordering the variables in the system [18, 15]. As a result, we obtain:

$$c_1 \circ c_2 = \bigwedge_{i=0}^{7} (\boldsymbol{r1'}[i] \equiv_{256} \boldsymbol{r0}[i]) \wedge \bigwedge_{i=0}^{7} (128 \cdot \boldsymbol{r0'}[i] \equiv_{256} 128 \cdot (\boldsymbol{r0}[i] + \boldsymbol{r1}[i]))$$

That is, after the second instruction, register `R1` holds the original value of `R0`. Further, by computing $c_1 \circ c_2 \circ c_3$ analogously, we derive:

$$c_1 \circ c_2 \circ c_3 = \bigwedge_{i=0}^{7} (\boldsymbol{r0'}[i] \equiv_{256} \boldsymbol{r1}[i]) \wedge \bigwedge_{i=0}^{7} (\boldsymbol{r1'}[i] \equiv_{256} \boldsymbol{r0}[i])$$

This congruent representation reveals that the sequence of instructions performs an in-place swapping of `R0` and `R1` using consecutive exclusive-or operations.

3.3 Reasoning about Ranges Using Invariants

Recall again the example program from Fig. 1, which copies three values from program memory into SRAM. The interval analysis infers a map $I : \mathcal{V} \times \mathcal{P} \to \mathsf{Int}$, which states that before instruction `0x5A` is executed, the registers `X` and `Z` hold the values $I(\texttt{X}, \texttt{0x5A}) = [96, 99]$ and $I(\texttt{Z}, \texttt{0x5A}) = [0, 65535]$.

To derive program invariants, we express the behaviour of the program fragment in terms of a flowchart program $\langle \mathcal{P}, \mathcal{V}, p_0, T \rangle$, where $\mathcal{P}$ is the set of program locations, $\mathcal{V}$ is the set of program variables, $p_0 \in \mathcal{P}$ is the initial program location and $T \subseteq \mathcal{P} \times \mathcal{P}$ defines the possible transitions between the instructions as given by the control flow graph. Consequently, we have $\mathcal{P} = \{\texttt{0x50}, \ldots, \texttt{0x5A}\}$, $\mathcal{V} = \{\texttt{R17}, \texttt{R26}, \texttt{R27}, \texttt{R30}, \texttt{R31}\}$ and $p_0 = \texttt{0x50}$. The semantics of the program can be stated as the least fixed point of a system of equations, given through:

- $\mathsf{inv}(p_0) = \bigwedge_{v \in \mathcal{V}} \left(\bigwedge_{i=0}^{7} \boldsymbol{v'}[i] \equiv_{256} \boldsymbol{v}[i] \right)$ for the initial program location p_0.
- $\mathsf{inv}(p_j) = \bigsqcup_{(p_i, p_j) \in T} (\mathsf{inv}(p_i) \circ c_{i,j})$, where $c_{i,j}$ denotes the instantiated congruent transfer function connecting $p_i \in \mathcal{P}$ and $p_j \in \mathcal{P}$.

Here, $\bigsqcup$ denotes the least upper bound operator over congruences as defined in [15]. Applying the first equation $\mathsf{inv}(p_{\texttt{0x51}}) = \mathsf{inv}(p_0) \circ c_{\texttt{0x50}}$ then gives

$$\mathsf{inv}(p_{\texttt{0x51}}) = \begin{cases} \bigwedge_{i=0}^{7} (\boldsymbol{r17'}[i] \equiv_{256} 0) \quad \wedge \\ \bigwedge_{i=0}^{7} (\boldsymbol{r26'}[i] \equiv_{256} \boldsymbol{r26}[i]) \wedge \bigwedge_{i=0}^{7} (\boldsymbol{r27'}[i] \equiv_{256} \boldsymbol{r27}[i]) \wedge \\ \bigwedge_{i=0}^{7} (\boldsymbol{r30'}[i] \equiv_{256} \boldsymbol{r30}[i]) \wedge \bigwedge_{i=0}^{7} (\boldsymbol{r31'}[i] \equiv_{256} \boldsymbol{r31}[i]) \end{cases}$$

and thereafter, the invariant is stable. To express the program invariant $\mathsf{inv}(p_{\texttt{0x5A}})$, let $\langle\!\langle \boldsymbol{x} \rangle\!\rangle = \sum_{i=0}^{7} 2^i \boldsymbol{x}[i]$. Proceeding with the computations eventually yields:

$$\mathsf{inv}(p_{\texttt{0x5A}}) = \begin{cases} (\langle\!\langle \boldsymbol{r26'} \rangle\!\rangle - \langle\!\langle \boldsymbol{r30'} \rangle\!\rangle \equiv_{256} 30) \wedge \\ \bigwedge_{i=0}^{7} (\boldsymbol{r17'}[i] \equiv_{256} 0 \wedge \boldsymbol{r27'}[i] \equiv_{256} 0 \wedge \boldsymbol{r31'}[i] \equiv_{256} 0) \end{cases}$$

From $\mathsf{inv}(p_{\texttt{0x5A}})$ and $I(\texttt{X}, \texttt{0x5A}) = [96, 99]$, we can now derive $I(\texttt{Z}, \texttt{0x5A}) = [66, 69]$. In the following, we will first see how program invariants of this kind are derived for arbitrary assembly programs, and then describe a systematic way of refining congruences and intervals in parallel. This operation amounts to triangularisation and checking satisfiability in order to strengthen the descriptions in both domains. Formally speaking, we will derive an operator $\mathsf{reduce} : \mathsf{Int} \times \mathsf{Cong} \to \mathsf{Int} \times \mathsf{Cong}$ such that $\mathsf{reduce}(i, c) \sqsubseteq (i, c)$ for $(i, c) \in \mathsf{Int} \times \mathsf{Cong}$ (cf. Sect. 6).

4 Relational Semantics for Assembly Code

In his seminal paper on congruence analysis, Granger [13] lamented the difficulty of handcrafting transformers for the congruence domain. However, since each of the 131 instructions on the ATmega16 has a well-defined semantics on the level of bits, we synthesise templates of transfer functions, based on a propositional encoding of the instructions and the computation of congruent closure to remedy this difficulty. When modelling the effects of instructions, no abstraction is applied, such that the formulae define the concrete semantics of the instructions.

Instructions for the ATmega platform have either zero, one, or two operands. Here, we present a relational encoding $[\![\cdot]\!]$ for a representative subset of the instruction-set. The semantics for other instructions can be derived analogously from the instruction set manual [1]. Given a set of memory locations accessed by an instruction, its encoding is given over disjoint bit-vectors for representing each accessed memory location, where the outputs are primed. Formally speaking, given a set of program variables $\mathcal{V}$, the Boolean formulae $[\![\cdot]\!]$ are defined over $\mathbb{B}_{\mathbf{V} \cup \mathbf{V}'}$, where $\mathbf{V} = \{\boldsymbol{v}[i] \mid v \in \mathcal{V}, 0 \leq i \leq 7\}$, $\mathbf{V}' = \{\boldsymbol{v}'[i] \mid v \in \mathcal{V}, 0 \leq i \leq 7\}$, and $\mathbb{B}_Y$ defines the class of Boolean formulae over propositional variables Y. Additionally, we require $\mathbf{V} \cap \mathbf{V}' = \emptyset$.

4.1 Copy and Load Instructions

The instruction `MOV r s` copies a register `s` into `r`. Similarly, given $\texttt{c} \in \mathbb{Z}_m$, the instruction `LDI r c` loads the constant value `c` into `r`. To express, introduce a bit-vector $\boldsymbol{c} \in \mathbb{B}^8$ with $\langle\!\langle \boldsymbol{c} \rangle\!\rangle = \texttt{c}$. The semantics of these instructions can be encoded relationally over bit-vectors $\boldsymbol{r}$, $\boldsymbol{s}$, $\boldsymbol{r}'$ and $\boldsymbol{s}'$ as:

$$[\![\texttt{MOV r s}]\!] = \bigwedge_{i=0}^{7} (\boldsymbol{r}'[i] \leftrightarrow \boldsymbol{s}[i]) \wedge \bigwedge_{i=0}^{7} (\boldsymbol{s}'[i] \leftrightarrow \boldsymbol{s}[i])$$
$$[\![\texttt{LDI r c}]\!] = \bigwedge_{i=0}^{7} (\boldsymbol{r}'[i] \leftrightarrow \boldsymbol{c}[i])$$

Computing the congruent closure of $[\![\texttt{MOV r s}]\!]$, e.g., yields:

$$\alpha_{\mathsf{Cong}}([\![\texttt{MOV r s}]\!]) = \bigwedge_{i=0}^{7} (\boldsymbol{r}'[i] \equiv_{256} \boldsymbol{s}[i]) \wedge \bigwedge_{i=0}^{7} (\boldsymbol{s}'[i] \equiv_{256} \boldsymbol{s}[i])$$

Observe that for these instructions, a modulus of 2 would suffice, but this is not always so. However, choosing the modulus to match the register-length is safe. Moreover, note that the status register (called `SREG` in case of the ATmega16) is not affected by these instructions, which is different for logical or arithmetic instructions. Overall, the status register contains 8 different flags that can be affected by instructions: carry flag $\boldsymbol{C}$, zero flag $\boldsymbol{Z}$, negative flag $\boldsymbol{N}$, overflow flag $\boldsymbol{O}$, sign flag $\boldsymbol{S}$, half-carry flag $\boldsymbol{H}$, transfer flag $\boldsymbol{T}$, and interrupt flag $\boldsymbol{I}$. The exact way these bits are set or cleared, however, depends on the concrete instruction.

4.2 Bitwise Instructions

As bitwise operations, the ATmega16 supports bitwise-and (`AND`), bitwise-and with a constant value (`ANDI`), bitwise negation (`COM`), exclusive-or (`EOR`), bitwise-or (`OR`), and bitwise-or with a constant (`ORI`). The effects of these operations on the destination register, denoted $\theta(\texttt{op})$, are bit-blasted as follows:

$$\begin{aligned}
\theta(\texttt{AND r s}) &= \textstyle\bigwedge_{i=0}^{7} (\boldsymbol{r}'[i] \leftrightarrow \boldsymbol{r}[i] \wedge \boldsymbol{s}[i]) \wedge \bigwedge_{i=0}^{7} (\boldsymbol{s}'[i] \leftrightarrow \boldsymbol{s}[i]) \\
\theta(\texttt{COM r}) &= \textstyle\bigwedge_{i=0}^{7} (\boldsymbol{r}'[i] \leftrightarrow \neg \boldsymbol{r}[i]) \\
\theta(\texttt{EOR r s}) &= \textstyle\bigwedge_{i=0}^{7} (\boldsymbol{r}'[i] \leftrightarrow \boldsymbol{r}[i] \oplus \boldsymbol{s}[i]) \wedge \bigwedge_{i=0}^{7} (\boldsymbol{s}'[i] \leftrightarrow \boldsymbol{s}[i]) \\
\theta(\texttt{OR r s}) &= \textstyle\bigwedge_{i=0}^{7} (\boldsymbol{r}'[i] \leftrightarrow \boldsymbol{r}[i] \vee \boldsymbol{s}[i]) \wedge \bigwedge_{i=0}^{7} (\boldsymbol{s}'[i] \leftrightarrow \boldsymbol{s}[i])
\end{aligned}$$

The encodings for `ANDI r c` and `ORI r c` are derived by replacing $\boldsymbol{s}[i]$ in the respective formulae with $\boldsymbol{c}[i]$ defined as above. As an example, consider the abstraction of `COM r`, which flips all bits in `r`:

$$\alpha_{\mathsf{Cong}}(\theta(\texttt{COM r})) = \textstyle\bigwedge_{i=0}^{7} (128 \cdot \boldsymbol{r}'[i] \equiv_{256} 128 \cdot \boldsymbol{r}[i] + 128)$$

Bitwise instructions also alter status flags. These effects are encoded in formulae $\psi(\texttt{op})$, leading to an encoding $[\![\texttt{op}]\!] = \theta(\texttt{op}) \wedge \psi(\texttt{op})$. `AND r s`, for instance, behaves as follows with respect to the status flags: It clears the overflow flag, sets the negative flag iff $\boldsymbol{r}'[7]$ is set, sets the sign flag to $\boldsymbol{N}' \oplus \boldsymbol{O}'$, and sets the zero flag iff all bits in $\boldsymbol{r}'$ are cleared. The other flags remained unchanged. To express, let $\mathsf{id}(\boldsymbol{x}) = \boldsymbol{x}' \leftrightarrow \boldsymbol{x}$. Then:

$$\psi(\texttt{AND r s}) \quad = \quad \begin{cases} \neg \boldsymbol{O}' \ \wedge & \boldsymbol{Z}' \leftrightarrow (\bigwedge_{i=0}^{7} \neg \boldsymbol{r}'[i]) \wedge \mathsf{id}(\boldsymbol{T}) \ \wedge \boldsymbol{N}' \leftrightarrow \boldsymbol{r}'[7] \ \wedge \\ \mathsf{id}(\boldsymbol{C}) \wedge & \boldsymbol{S}' \leftrightarrow \boldsymbol{N}' \oplus \boldsymbol{O}' \qquad \wedge \mathsf{id}(\boldsymbol{H}) \wedge \mathsf{id}(\boldsymbol{I}) \end{cases}$$

Encodings $\psi(\texttt{op})$ for `ANDI`, `EOR`, `OR`, and `ORI` are equal to this case. `COM` differs in that it always sets the carry flag. Observe that the congruence domain is too weak to express the relationship on $\boldsymbol{Z}'$, but it can represent the other ones.

4.3 Shifts

In terms of shifts, the ATmega16 supports arithmetic shift right (`ASR`), logical shift left (`LSL`), logical shift right (`LSR`), rotate left through carry (`ROL`), and rotate right through carry (`ROR`). All these operations shift the value of the source

register by a single position, shifts by a higher or variable number of positions are not supported. `ASR r` shifts all bits in `r` to the right, the most significant (MSB) bit is held constant, and the least significant bit (LSB) is shifted into the carry. Thus, the instruction divides a signed `r` by two without changing its sign. `LSR r` behaves analogously for an unsigned value. `LSL r` multiplies `r` by two, shifting the MSB into the carry and clearing the LSB. `ROL r` and `ROR r` are used to multiply and divide multi-byte signed and unsigned values by two, by shifting the carry flag into the LSB/MSB of `r` and shifting the value of the MSB/LSB bit into the carry. Expressed in propositional logic, this gives:

$$\begin{aligned}
\theta(\texttt{ASR r}) &= \textstyle\bigwedge_{i=0}^{6} (\boldsymbol{r}'[i] \leftrightarrow \boldsymbol{r}[i+1]) \wedge \boldsymbol{r}'[7] \leftrightarrow \boldsymbol{r}[7] \wedge \boldsymbol{C}' \leftrightarrow \boldsymbol{r}[0] \\
\theta(\texttt{LSL r}) &= \textstyle\bigwedge_{i=0}^{6} (\boldsymbol{r}'[i+1] \leftrightarrow \boldsymbol{r}[i]) \wedge \neg \boldsymbol{r}'[0] \wedge \boldsymbol{C}' \leftrightarrow \boldsymbol{r}[7] \\
\theta(\texttt{ROR r}) &= \textstyle\bigwedge_{i=0}^{6} (\boldsymbol{r}'[i] \leftrightarrow \boldsymbol{r}[i+1]) \wedge \boldsymbol{r}'[7] \leftrightarrow \boldsymbol{C} \wedge \boldsymbol{C}' \leftrightarrow \boldsymbol{r}[0]
\end{aligned}$$

Encodings for `LSR` and `ROL` are specified similarly. The updates of the status flags are then expressed analogously to before with $[\![\texttt{op}]\!] = \theta(\texttt{op}) \wedge \psi(\texttt{op})$ and $\psi(\texttt{op}) = \varphi(\texttt{op}) \wedge \xi(\texttt{op})$, where

$$\varphi(\texttt{op}) = \begin{cases} \boldsymbol{N}' \leftrightarrow \boldsymbol{r}'[7] \quad \wedge \boldsymbol{Z}' \leftrightarrow \bigwedge_{i=0}^{7} \neg \boldsymbol{r}'[i] \wedge \mathsf{id}(\boldsymbol{T}) \ \wedge \mathsf{id}(\boldsymbol{I}) \ \wedge \\ \boldsymbol{O}' \leftrightarrow \boldsymbol{N}' \oplus \boldsymbol{C}' \wedge \boldsymbol{S}' \leftrightarrow \boldsymbol{N}' \oplus \boldsymbol{O}' \quad \wedge \mathsf{id}(\boldsymbol{H}) \end{cases}$$

is the same among all shift instructions, whereas $\xi(\texttt{op}) = \boldsymbol{C}' \leftrightarrow \boldsymbol{r}[0]$ for $\texttt{op} \in \{\texttt{ASR}, \texttt{LSR}, \texttt{ROR}\}$ and $\xi(\texttt{op}) = \boldsymbol{C}' \leftrightarrow \boldsymbol{r}[7]$ otherwise.

4.4 Arithmetic Instructions

Let us consider encodings for two arithmetic instructions, in this case for summing up two registers (`ADD`) and incrementing a register by 1 (`INC`). Here, `ADD r s` is expressed using a cascade of full-adders using additional carry bits $\boldsymbol{c}$:

$$\begin{aligned}
\theta(\texttt{ADD r s}) &= \left(\textstyle\bigwedge_{i=0}^{7} \boldsymbol{r}'[i] \leftrightarrow \boldsymbol{r}[i] \oplus \boldsymbol{s}[i] \oplus \boldsymbol{c}[i]\right) \wedge \neg \boldsymbol{c}[0] \wedge \\
&\quad \left(\textstyle\bigwedge_{i=0}^{6} \boldsymbol{c}[i+1] \leftrightarrow (\boldsymbol{r}[i] \wedge \boldsymbol{s}[i]) \vee (\boldsymbol{r}[i] \wedge \boldsymbol{c}[i]) \vee (\boldsymbol{s}[i] \wedge \boldsymbol{c}[i])\right) \\
\theta(\texttt{INC r}) &= \textstyle\bigwedge_{i=0}^{7} (\boldsymbol{r}'[i] \leftrightarrow \boldsymbol{r}[i] \oplus \bigwedge_{j=0}^{i-1} \boldsymbol{r}[j])
\end{aligned}$$

Bit-wise encodings for other arithmetic instructions such as computation of the two's complement (`NEG`) or subtraction (`SUB`) are derived accordingly. The effects $\psi(\texttt{op})$ on the status register can be derived analogously to the previous examples to obtain $[\![\texttt{op}]\!] = \psi(\texttt{op}) \wedge \theta(\texttt{op})$. Abstracting the increment using congruences then gives:

$$\alpha_{\mathsf{Cong}}(\theta(\texttt{INC r})) = (\langle\!\langle \boldsymbol{r} \rangle\!\rangle' \equiv_{256} \langle\!\langle \boldsymbol{r} \rangle\!\rangle + 1)$$

Using the same approach, Boolean encodings for the complete instruction set of the ATmega16 and the corresponding congruent abstractions are computed. For instance, branching instructions such as `BRNE` do not alter the status of the addressable memory, but only the program counter, which is implicitly encoded in the control flow graph. Compare instructions such as `CP`, `CPC`, or `CPI` subtract two values, but they only alter the status flags accordingly and do not store the result at a memory location.

5 A Discussion of Soundness

As stated in Sect. 3.3 already, defining a program analysis over congruences amounts to the application of four operations: instantiating template functions, relational composition $\circ$, join $\sqcup$, and checking entailment $\models$. Since congruences satisfy the finite ascending chain condition, no widening is needed [18]. We make no contributions in this regard. However, two open issues warrant discussion: the effect of indirect stores on the validity of invariants and relationships between addresses of a region and the contents of that address.

In Sect. 3.3, we have not modelled the effects of indirect stores on memory locations `96`–`98` in SRAM. Thus, no relational constraints are put onto these memory locations. However, suppose that a value is copied from `s` into a target register `r` using a direct access, which generates an equality constraint $\bigwedge_{i=0}^{7} \boldsymbol{r}[i] \equiv_{256} \boldsymbol{s}[i]$, and later `r` is overwritten using an indirect store. Following the approach described so far, the equality constraint remains in the program invariant, which is unsound. The strength of using a concrete memory model, where each cell is represented by an integer address, is that the intervals provide an upper-approximation of the targets of indirect stores. Hence, we can simply modify the $\circ$ operator such that all constraints on targets of indirect writes are eliminated when $\circ$ is applied. This is achieved by removing all equalities that involve the target register from the invariant. This strategy recovers soundness. As a matter of fact, this method typically yields the same results as if the constraints on the targets were joined (since indirect stores are modelled as weak updates).

Even though it is not possible to derive relationships on the targets of indirect stores using weak updates, it is possible to derive a relationship between indirectly written locations and their contents. To illustrate, suppose we have an indirect store operation `ST X R0`, and a program invariant is generated. Then, if the invariant exhibits a relationship between `X` and `R0`, it follows that if a target memory location is written (which cannot be guaranteed), the target address is congruently related to the source register `R0` as described by the invariant.

6 Reducing Abstract Descriptions

Thus far we have derived bit-level invariants, which are systems of linear congruences. In this section, we show how congruences and intervals are combined to derive more precise abstractions in both domains. Finally, strides – that is, sets of values that are separated by a constant $k \in \mathbb{N}$ – are extracted from the refined ranges.

6.1 A Reduce Operator

Given $S_1, S_2 \subseteq \mathbb{B}^{nw}$, where S_1 represents the models of the interval abstraction and S_2 represents the models of the congruent invariant, we construct $S_1 \cap S_2$ formally. To represent the models of intervals, let $\boldsymbol{\ell}_i, \boldsymbol{u}_i \in \mathbb{B}^w$ denote bitwise encodings of the extremal values of $v_i \in \mathcal{V}$ for a fixed $p \in \mathcal{P}$ as defined through the map $I : \mathcal{V} \times \mathcal{P} \to \mathsf{Int}$. Then:

Definition 2. The operator $\mathsf{cube} : 2^{\mathbb{B}^{nw}} \to 2^{\mathbb{B}^{nw}}$ is defined:

$$\mathsf{cube}(S) = \left\{ \boldsymbol{x} \in \mathbb{B}^{nw} \;\middle|\; \begin{array}{ll} \forall i \in [0, n-1] : \boldsymbol{\ell}_i, \boldsymbol{u}_i \in S & \wedge \\ \ell'_i = \langle\!\langle \langle \boldsymbol{\ell}_i[0], \ldots, \boldsymbol{\ell}_i[w-1] \rangle \rangle\!\rangle & \wedge \\ u'_i = \langle\!\langle \langle \boldsymbol{u}_i[0], \ldots, \boldsymbol{u}_i[w-1] \rangle \rangle\!\rangle & \wedge \\ \ell'_i \le \langle\!\langle \langle \boldsymbol{x}[iw], \ldots, \boldsymbol{x}[iw+w-1] \rangle \rangle\!\rangle \le u'_i & \end{array} \right\}$$

It is straightforward to show that $\mathsf{cube} : 2^{\mathbb{B}^{nw}} \to 2^{\mathbb{B}^{nw}}$ and $\mathsf{cong} : 2^{\mathbb{B}^{nw}} \to 2^{\mathbb{B}^{nw}}$ are closure operators, that is, extensive, increasing and idempotent. Further, suppose $S_1, S_2, \ldots \subseteq \mathbb{B}^{nw}$. If $\mathsf{cube}(S_i) = S_i$ for all $i \in \mathbb{N}$ then $\mathsf{cube}(\cap_{i\in\mathbb{N}} S_i) = \cap_{i\in\mathbb{N}} S_i$m, and if $\mathsf{cong}(S_i) = S_i$ for all $i \in \mathbb{N}$ then $\mathsf{cong}(\cap_{i\in\mathbb{N}} S_i) = \cap_{i\in\mathbb{N}} S_i$. To derive Galois connections, and accordingly safety of our computations, we define abstraction and concretisation as follows:

Definition 3. The abstraction and concretisation maps are defined as:

$$\alpha_{\mathsf{cube}}(S) = \cap\{S' \subseteq \mathbb{B}^{nw} \mid S \subseteq S' \wedge S' = \mathsf{cube}(S')\} \qquad \gamma_{\mathsf{cube}}(S) = S$$
$$\alpha_{\mathsf{cong}}(S) = \cap\{S' \subseteq \mathbb{B}^{nw} \mid S \subseteq S' \wedge S' = \mathsf{cong}(S')\} \qquad \gamma_{\mathsf{cong}}(S) = S$$

Then, any subset of $\mathbb{B}^{nw}$ (or equivalently $\mathbb{Z}_m$) closed under affine combination can be represented congruently. A similar observation holds for the cube of S. Further, we have $\mathsf{cube}(S) = S$ iff there exists $\ell'_0, \ldots, \ell'_{n-1} \in [-2^{w-1}, 2^{w-1} - 1]$ and $u'_0, \ldots, u'_{n-1} \in [-2^{w-1}, 2^{w-1} - 1]$ such that:

$$S = \left\{ \boldsymbol{x} \in \mathbb{B}^{nw} \;\middle|\; \forall i \in [0, n-1] : \ell'_i \le \langle\!\langle \langle \boldsymbol{x}[iw], \ldots, \boldsymbol{x}[iw+w-1] \rangle \rangle\!\rangle \le u'_i \right\}$$

For congruences, it is $\mathsf{cong}(S) = S$ iff there exists a matrix $[A \mid \boldsymbol{b}] \in \mathbb{Z}^{k,nw+1}$ such that $S = \{\boldsymbol{x} \in \mathbb{B}^{nw} \mid A\boldsymbol{x} \equiv_{2^w} \boldsymbol{b}\}$.

Finally, we present a constructive approach to computing the affine intersection between S_1 and S_2. This construction is based on strengthening S_2 using constraints from S_1 (or I, respectively). The key idea in this construction is introduce fresh equalities to express the non-negativity of $\langle\!\langle \boldsymbol{v}_i \rangle\!\rangle - \langle\!\langle \boldsymbol{\ell}_i \rangle\!\rangle$ and $\langle\!\langle \boldsymbol{u}_i \rangle\!\rangle - \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle$ in order to enforce $\langle\!\langle \boldsymbol{\ell}_i \rangle\!\rangle \le \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle \le \langle\!\langle \boldsymbol{u}_i \rangle\!\rangle$. This is achieved by imposing a zero-constraint on the MSB of the difference, which corresponds to the sign bit. This construction is followed by putting the resulting system into upper triangular form.

Proposition 1. Suppose $\ell'_0, \ldots, \ell'_{n-1}, u'_0, \ldots, u'_{n-1} \in [0, 2^w - 1]$ and let $[A \mid \boldsymbol{b}] \in \mathbb{Z}^{k,nw+1}$. Define

$$S_1 = \left\{ \boldsymbol{x} \in \mathbb{B}^{nw} \;\middle|\; \forall i \in [0, n-1] : \ell'_i \le \langle\!\langle \langle \boldsymbol{x}[iw], \ldots, \boldsymbol{x}[iw+w-1] \rangle \rangle\!\rangle \le u'_i \right\}$$
$$S_2 = \{\boldsymbol{x} \in \mathbb{B}^{nw} \mid A\boldsymbol{x} \equiv_{2^w} \boldsymbol{b}\}$$

Let $\boldsymbol{e}$, $\boldsymbol{f} \in \mathbb{B}^w$ such that $\boldsymbol{e} = \langle 0, 0, \cdots, 0, 1 \rangle$ and $\boldsymbol{f} = \langle 1, 2, \cdots, 2^{w-2}, 2^{w-1} \rangle$. Moreover, let $A' \in \mathbb{Z}^{k+4n,3nw}$, $E \in \mathbb{Z}^{n,nw}$ and $F \in \mathbb{Z}^{n,nw}$ defined by:

$$A' = \left[\begin{array}{c|c|c} E & 0 & 0 \\ \hline 0 & E & 0 \\ \hline 0 & -F & F \\ \hline F & 0 & -F \\ \hline 0 & 0 & A \end{array}\right] \qquad E = \left[\begin{array}{c|c|c|c} \boldsymbol{e} & \boldsymbol{0} & \cdots & \boldsymbol{0} \\ \hline \boldsymbol{0} & \boldsymbol{e} & \cdots & \boldsymbol{0} \\ \hline \vdots & \vdots & \vdots & \vdots \\ \hline \boldsymbol{0} & \boldsymbol{0} & \cdots & \boldsymbol{e} \end{array}\right] \qquad F = \left[\begin{array}{c|c|c|c} \boldsymbol{f} & \boldsymbol{0} & \cdots & \boldsymbol{0} \\ \hline \boldsymbol{0} & \boldsymbol{f} & \cdots & \boldsymbol{0} \\ \hline \vdots & \vdots & \vdots & \vdots \\ \hline \boldsymbol{0} & \boldsymbol{0} & \cdots & \boldsymbol{f} \end{array}\right]$$

Additionally, let $\boldsymbol{l} \in \mathbb{Z}^n$, $\boldsymbol{u} \in \mathbb{Z}^n$, $\boldsymbol{b}' \in \mathbb{Z}^{k+4n}$ where

$$\boldsymbol{l} = \begin{bmatrix} \ell'_0 \\ \hline \ell'_1 \\ \hline \vdots \\ \hline \ell'_{n-1} \end{bmatrix} \qquad \boldsymbol{u} = \begin{bmatrix} u'_0 \\ \hline u'_1 \\ \hline \vdots \\ \hline u'_{n-1} \end{bmatrix} \qquad \boldsymbol{b}' = \begin{bmatrix} \boldsymbol{0} \\ \hline \boldsymbol{0} \\ \hline \boldsymbol{l} \\ \hline \boldsymbol{u} \\ \hline \boldsymbol{b} \end{bmatrix} \qquad \boldsymbol{x}' = \begin{bmatrix} \boldsymbol{z} \\ \hline \boldsymbol{y} \\ \hline \boldsymbol{x} \end{bmatrix}$$

Then $S_1 \cap S_2 = \{\boldsymbol{x} \in \mathbb{B}^{nw} \mid A'\boldsymbol{x}' \equiv_{2^w} \boldsymbol{b}'\}$.

Refining intervals follows a method for maximising values in Boolean formulae described by Codish et al. [6] using successive calls to a decision procedure. The key idea is to maximise single bits – starting from the MSB – and checking satisfiability of a system of linear 0/1 constraints using SAT [15]. We use SAT solving because triangularisation only provides an incomplete decision procedure for 0/1 variables. In the following definition, the symbol : denotes the concatenation of bit-vectors.

Definition 4. Define $\mathsf{max}(A, b, i) = \mathsf{extr}(A, b, i, w, 0, 1)$ where $\mathsf{extr}(A, b, i, j, v_1, v_2)$:

- ϵ if $j = 0$.
- $\langle v_1 \rangle : \mathsf{extr}(\begin{bmatrix} e_i \\ \hline A \end{bmatrix}, \begin{bmatrix} v_1 \\ \hline \boldsymbol{b} \end{bmatrix}, i-1, j-1, v_2, v_2)$ if $\begin{bmatrix} e_i \\ \hline A \end{bmatrix} \boldsymbol{x} \equiv_{2^w} \begin{bmatrix} v_1 \\ \hline \boldsymbol{b} \end{bmatrix}$ is satisfiable.
- $\langle \neg v_1 \rangle : \mathsf{extr}(\begin{bmatrix} e_i \\ \hline A \end{bmatrix}, \begin{bmatrix} \neg v_1 \\ \hline \boldsymbol{b} \end{bmatrix}, i-1, j-1, v_2, v_2)$ otherwise.

Conversely define $\mathsf{min}(A, b, i) = \mathsf{extr}(A, b, i, w, 1, 0)$. Finally, reduce follows from the combination of min, max, and $\cap$:

Corollary 1. *Let* $S_1 \cap S_2 = A\boldsymbol{x} \equiv_{2^w} \boldsymbol{b}$. *Then* $\mathsf{reduce}(S_1, S_2) = (I', A\boldsymbol{x} \equiv_{2^w} \boldsymbol{b})$ *where* $I' = \langle [\mathsf{min}(A, b, 0), \mathsf{max}(A, b, 0)], \ldots, [\mathsf{min}(A, b, n-1), \mathsf{max}(A, b, n-1)] \rangle$.

Example 1. Suppose $w = 4$, $n = 2$ and $S_2 = \{\boldsymbol{x} \in \mathbb{B}^8 \mid A\boldsymbol{x} \equiv_{2^4} \boldsymbol{b}\}$ where

$$A = \left[\begin{array}{cccc|cccc} 1&0&0&0&-1&0&0&0 \\ 0&1&0&0&0&-1&0&0 \\ 0&0&1&0&0&0&-1&0 \\ 0&0&0&1&0&0&0&-1 \\ 0&0&0&0&1&0&0&0 \end{array}\right] \qquad \boldsymbol{b} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

To interpret $[A \mid \boldsymbol{b}]$, let $\boldsymbol{u} = \langle \boldsymbol{x}[0], \boldsymbol{x}[1], \boldsymbol{x}[2], \boldsymbol{x}[3] \rangle$ and $\boldsymbol{v} = \langle \boldsymbol{x}[4], \boldsymbol{x}[5], \boldsymbol{x}[6], \boldsymbol{x}[7] \rangle$. Then the system $A\boldsymbol{x} \equiv_{2^4} \boldsymbol{b}$ implies that $\langle\!\langle \boldsymbol{u} \rangle\!\rangle \equiv_{2^w} \langle\!\langle \boldsymbol{v} \rangle\!\rangle$ and $\langle\!\langle \boldsymbol{v} \rangle\!\rangle \equiv_2 0$. Now let

$$S_1 = \{\boldsymbol{x} \in \mathbb{B}^8 \mid 4 \leq \langle\!\langle \boldsymbol{u} \rangle\!\rangle \leq 15 \wedge 0 \leq \langle\!\langle \boldsymbol{v} \rangle\!\rangle \leq 7\}$$

and consider $S_1 \cap S_2$ as characterised by $[A' \mid \boldsymbol{b}']$ which is:

$$\left[\begin{array}{cccc|cccc||cccc|cccc||cccc|cccc||c}
0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
\hline
0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0\\
\hline
0&0&0&0&0&0&0&0&-1&-2&-4&-8&0&0&0&0&1&2&4&8&0&0&0&0&4\\
0&0&0&0&0&0&0&0&0&0&0&0&-1&-2&-4&-8&0&0&0&0&1&2&4&8&0\\
\hline
1&2&4&8&0&0&0&0&0&0&0&0&0&0&0&0&-1&-2&-4&-8&0&0&0&0&15\\
0&0&0&0&1&2&4&8&0&0&0&0&0&0&0&0&0&0&0&0&-1&-2&-4&-8&7\\
\hline
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&-1&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&-1&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&-1&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&-1&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0
\end{array}\right]$$

Putting this into a triangular form, we achieve:

$$\left[\begin{array}{cccc|cccc||cccc|cccc||cccc|cccc||c}
1&2&4&8&0&0&0&0&0&0&0&0&0&0&0&0&-1&-2&-4&-8&0&0&0&0&15\\
0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&1&2&4&8&0&0&0&0&0&0&0&0&0&0&0&0&-1&-2&-4&-8&7\\
0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&-1&-2&-4&-8&0&0&0&0&1&2&4&8&0&0&0&0&4\\
0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&-1&-2&-4&-8&0&0&0&0&1&2&4&8&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&-1&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&-1&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&-1&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&-1&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0
\end{array}\right]$$

Here, rows 3 and 4 impose the constraint $\langle\!\langle \boldsymbol{v} \rangle\!\rangle \leq 7$ by requiring $7 - \langle\!\langle \boldsymbol{v} \rangle\!\rangle \geq 0$. The constraints can be projected from $S_1 \cap S_2$, which yields $\boldsymbol{u}[3] \equiv_{2^w} 0$, and thus, $\langle\!\langle \boldsymbol{u} \rangle\!\rangle \leq 7$. With $\boldsymbol{u}[0] \equiv_{2^w} 0$, applying SAT yields $4 \leq \langle\!\langle \boldsymbol{u} \rangle\!\rangle \leq 6$. Note, however, that more precise congruences could be extracted by encoding the equation system in propositional logic and recomputing congruent closure.

6.2 Refinement for Strides

For $p \in \mathcal{P}$, the respective invariant $\mathsf{inv}(p)$, and a variable $v \in \mathcal{V}$, let $i \in \mathbb{N}$ be the maximum index of bit-vector $\boldsymbol{v}$ such that $\mathsf{inv}(p)$ contains relations $\boldsymbol{v}[j] \equiv_{256} k_j$ for all $0 \leq j \leq i$ and $k_j \in \{0,1\}$. The size of the stride is then defined by 2^{i+1}, and the set of possible values constrained by the invariant is given through $Z = \{(\sum_{j=0}^{i} 2^j \boldsymbol{v}[j]) + k \cdot 2^{i+1} \mid k \in \mathbb{N}\}$. Thus, the resulting value-set is $I(v,p) \cap Z$.

Table 1. Optimality of synthesised transfer functions

Class	Instructions	$\mathsf{sol}(\alpha(\llbracket c \rrbracket)) = \mathsf{sol}(\llbracket c \rrbracket)$?
load & copy	`LDI`, `MOV`	yes
shift	`ASR`, `LSL`, `LSR`, `ROL`, `ROR`	yes
logical	`COM`, `EOR`, `SWAP`	yes
logical	`AND`, `ANDI`, `OR`, `ORI`	no
arithmetic	`ADC`, `ADD`, `DEC`, `INC`, `NEG`, `SBC`, `SUB`, `SUBI`	yes
arithmetic	`MUL`, `MULS`, `MULSU`	no
compare	`CP`, `CPC`, `CPI`	no
branching	`BRBC`, `BRBS`, ...	no

7 Experiments

We have integrated the ideas and algorithms described in this paper into the [MC]SQUARE verification platform for microcontroller binary code. In this section, we discuss our experiences with respect to optimality and the runtime requirements. All experiments were performed on a MacBook Pro, equipped with a 2.4 Ghz dual-core processor and 4 GB of RAM.

7.1 Optimality

Let $\alpha_{\mathsf{Cong}}(\llbracket f \rrbracket)$ denote a transfer function synthesised from a Boolean encoding $\llbracket f \rrbracket$. The congruence domain is optimal for abstracting f iff $\mathsf{sol}(\alpha_{\mathsf{Cong}}(\llbracket f \rrbracket)) = \mathsf{sol}(\llbracket f \rrbracket)$. Considering the classes of instructions that were described Sect. 4, optimality results given in Tab. 1 are obtained (ignoring the effects of arithmetic and logical instructions on the status register). Observe that compare and branching instructions, which are required to handle conditional branches and loop conditions, sometimes cannot be modelled precisely (recall the congruent abstraction of $\boldsymbol{Z'} \leftrightarrow \bigwedge_{i=0}^{7}(\neg \boldsymbol{r'}[i])$). This drawback, however, is remedied through the interval analysis, which constrains the ranges through branching conditions.

7.2 Runtime

Synthesing transfer functions up-front requires less than 1s for each instruction. Abstracting `INC r`, e.g., requires 17 SAT instances over 32 propositional variables to be solved with an overall runtime of 0.18s using SAT4J. Composing congruences is implemented using triangularisation, as is $\sqcup$. For the initialisation loop in Sect. 3.3, the loop invariant stabilised after 2 iterations, which led to 18 applications of $\circ$ and 2 applications of $\sqcup$, which required $0.3s$ overall. The runtime for operations on matrices is very susceptible to the number of variables in the system, and hence, $\boldsymbol{r17}$, $\boldsymbol{r26}$, $\boldsymbol{r27}$, $\boldsymbol{r30}$, and $\boldsymbol{r31}$ were eliminated prior to range-refinement as they are unrelated to the invariant. Since the runtime grows polynomially with the number of bits, computing invariants for complete programs is not tractable. Instead, an invariant generator should detect program fragments where the interval analyser loses precision.

Computing reduce to derive refined ranges requires 16 SAT instances to be solved which amounts to 0.25s. That is, two instances for each bit are required, whereas deriving strides is linear in the number of congruence relations.

8 Related Work

Defining and computing transformers for relational domains has been an active topic in abstract interpretation for decades, and numerous techniques for expressing relational constraints have been described [11, 17]. Most existing approaches, however, operate on unbounded integers, with the additional duty to verify that no overflow can occur [10]. The technique from [22] suggests to revise the truncation map to reflect overflows for polyhedral analysis.

In assembly code for 8-bit architectures, overflows can be observed commonly due to the limited bit-width. Therefore, it is natural to deploy congruence relations [18,13] where the modulus is 256. Instead of expressing ranges in a domain that handles wraps, our approach combines relational invariants with computationally inexpensive intervals [5]. The idea of reducing two abstract descriptions in parallel was already formalised by Cousot and Cousot [9]. Later, Codish et al. [7] have applied a similar technique to pair and set-sharing analysis.

The difficulty of designing optimal transfer functions was already discussed in [11]. However, it took several decades until it was observed that optimal transformers can be derived for any abstract domain that satisfies the ascending chain condition [21]. Our work builds on this to remedy both the difficulty and the workload of handcrafting transfer functions for the complete instruction set of the microcontroller as in [4]. Contemporaneously to [21], Regehr et al. [20] observed that optimal transfer functions for interval analysis of ATmega16 assembly can be derived using BDDs. However, the time needed for computing best transformers is considerably longer due to the use of BDD-based encodings without abstraction.

9 Conclusion and Future Work

We have shown that bit-level congruences provide a suitable means for deriving invariants for assembly code. We have detailed techniques for verifying, inferring, and refining ranges in presence of indirect reads and writes. The work calls for further research into the handling of indirect stores in order to derive strong updates instead of weak updates. Existing work on lifting abstract interpreters to quantified domains [14] could serve as a basis for this. Another interesting application is model checking, where congruences could be used to reduce the over-approximation introduced through abstractions [19] similar to the refinement described in Sect. 6, leading to smaller state spaces and fewer false alarms.

Acknowledgements

This work was supported, in part, by a Royal Society industrial secondment and the UMIC Research Centre at the RWTH Aachen University.

References

1. Atmel Corporation. 8-bit AVR Instruction Set (July 2008)
2. Bagnara, R., Dobson, K., Hill, P., Mundell, M., Zaffanella, E.: Grids: A domain for analyzing the distribution of numerical values. In: Puebla, G. (ed.) LOPSTR 2006. LNCS, vol. 4407, pp. 219–235. Springer, Heidelberg (2007)
3. Balakrishnan, G., Reps, T.W.: WYSINWYX: What You See Is Not What You eXecute. ACM Trans. Program. Lang. Syst. (to appear, 2010)
4. Brauer, J., King, A.: Automatic abstraction for intervals using boolean formulae. In: SAS 2010. LNCS. Springer, Heidelberg (2010)
5. Brauer, J., Noll, T., Schlich, B.: Interval analysis of microcontroller code using abstract interpretation of hardware and software. In: SCOPES. ACM, New York (to appear, 2010)
6. Codish, M., Lagoon, V., Stuckey, P.J.: Logic programming with Satisfiability. Theory and Practice of Logic Programming 8(1), 121–128 (2008)
7. Codish, M., Mulkers, A., Bruynooghe, M., García de la Banda, M.J., Hermenegildo, M.V.: Improving abstract interpretations by combining domains. ACM Trans. Program. Lang. Syst. 17(1), 28–44 (1995)
8. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, pp. 238–252. ACM, New York (1977)
9. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL, pp. 269–282 (1979)
10. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Mine, A., Monniaux, D., Rival, X.: The Astrée analyser. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 21–30. Springer, Heidelberg (2005)
11. Cousot, P., Halbwachs, N.: Automatic Discovery of Linear Restraints Among Variables of a Program. In: POPL, pp. 84–97. ACM Press, New York (1978)
12. Eide, E., Regehr, J.: Volatiles are miscompiled, and what to do about it. In: EMSOFT, pp. 255–264. ACM, New York (2008)
13. Granger, P.: Static analysis of linear congruence equalities among variables of a program. In: Abramsky, S. (ed.) CAAP 1991 and TAPSOFT 1991. LNCS, vol. 493, pp. 169–192. Springer, Heidelberg (1991)
14. Gulwani, S., McCloskey, B., Tiwari, A.: Lifting abstract interpreters to quantified logical domains. In: POPL, pp. 235–246. ACM, New York (2008)
15. King, A., Søndergaard, H.: Inferring congruence equations using SAT. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 281–293. Springer, Heidelberg (2008)
16. King, A., Søndergaard, H.: Automatic abstraction for congruences. In: Barthe, G., Hermenegildo, M. (eds.) VMCAI 2010. LNCS, vol. 5944, pp. 281–293. Springer, Heidelberg (2010)
17. Miné, A.: The Octagon Abstract Domain. Higher-Order and Symbolic Computation 19(1), 31–100 (2006)

18. Müller-Olm, M., Seidl, H.: Analysis of Modular Arithmetic. ACM Trans. Program. Lang. Syst. 29(5) (August 2007)
19. Noll, T., Schlich, B.: Delayed nondeterminism in model checking embedded systems assembly code. In: Yorav, K. (ed.) HVC 2007. LNCS, vol. 4899, pp. 185–201. Springer, Heidelberg (2008)
20. Regehr, J., Reid, A.: HOIST: A system for automatically deriving static analyzers for embedded systems. Operating Systems Review 38(5), 133–143 (2004)
21. Reps, T., Sagiv, M., Yorsh, G.: Symbolic Implementation of the Best Transformer. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 252–266. Springer, Heidelberg (2004)
22. Simon, A., King, A.: Taming the wrapping of integer arithmetic. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 121–136. Springer, Heidelberg (2007)