Solver Aided Reverse Engineering of Architectural Features

Bill Zorn, Dan Grossman, Luis Ceze
{billzorn, djg, luisceze}@cs.washington.edu
Paul G. Allen School for Computer Science & Engineering
University of Washington

ABSTRACT

To program a processor, you need to have some model of how it behaves. But providing accurate functional models of processors is challenging. Traditionally, the behavior of a processor is specified by documentation that describes its Instruction Set Architecture, or ISA. This documentation is usually long, making it laborious to produce, and it is often riddled with errors, typos, and inconsistencies. Additionally, it can be insufficiently formal for applications that need an accurate formal model of a processor, such as superoptimizers, program synthesis tools, or proof frameworks. We observe that formal methods can be used not just to consume these models, but to check them and to help us produce them. To demonstrate, we reverse engineer two interesting architectural features of a TI MSP430 microcontroller. First, we use the SMT solver Z3 [1] to synthesize the timings of many instructions. Second, we use Synapse [2], a program synthesis framework built on the Rosette solver-aided language [3], to synthesize the semantics of arithmetic instructions as bitvector programs. In both cases, we compare our results with the TI user's guide and find that we can provide a more accurate formal model.

1. INTRODUCTION

Formal methods are already well established for use in hardware verification. Techniques like symbolic and bounded model checking use tools such as BDDs and SAT solvers to check that circuits are equivalent or prove they satisfy a specification. We propose something different: using tools from formal methods to reverse engineer models of the observable behavior of processors.

Understanding processors solely by observing them can be challenging. In an interdisciplinary study that spans computer science, electrical engineering, and neuroscience, Eric Jonas and Konrad Kording examine whether data analysis techniques from neuroscience can explain the behavior of the MOS 6502 [4]. The authors conclude that these data-driven techniques do not produce a satisfactory understanding, even for this very simple processor. Without some underlying intuition about how the processor works, no amount of data is sufficient to explain its behavior.

With formal methods, we can do better. Tools from formal methods are interesting because they allow us to supply precisely the kind of intuition that was lacking in the neuroscience study. Rather than attempting to explain the data from scratch, they let us outline the form of the explanation, or provide a partial sketch of a model, and fill in the rest of the model so that it is guaranteed to be consistent with observation. Because the models they produce are both highly accurate and understandable by a human, these tools are well suited for developing models of processor behavior that are useful to programmers.

To illustrate, we consider two behaviors of the TI MSP430 fr5969 microcontroller. In Section 3 we examine the timing behavior of the processor. We empirically test the cycle counts provided in the TI user's guide, and attempt to synthesize a better timing model using Z3. In Section 4 we examine the behavior of arithmetic instructions and synthesize simple bitvector programs that implement them, in the process filling in some behavior that is undefined by the ISA documentation.

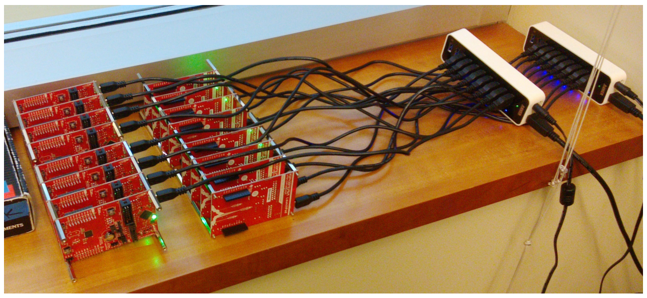
Most processors and ISA documents have bugs and inconsistencies, from the infamous Pentium FDIV bug of the 1990s to much more minor issues found by other automated formal approaches such as [5]. That said, processors and ISA documents tend to be mostly correct. The goal of this paper is to show how we can use formal methods to leverage this mostly correct information, and the specific domain knowledge of computer architects, to provide formal models that are more correct when the need arises.

2. THE MSP430FR5969

We study a simple mixed-signal microcontroller, the TI MSP430 FR5969. We choose this chip because it is simple but capable enough to be interesting and widely used. The MSP430 is an in-order processor with no caches and minimal pipeline state. The ISA only has 27 instructions, although it supports sophisticated CISC-like addressing modes. All our experiments are performed on an array of 16 MSP-EXP430 FR5969 LaunchPad Development Kit evaluation boards, pictured in Figure 1, which we control using `mspdebug` [6].

MSP430 processors are used in a wide variety of applications, from electronic locks [7] to pacemakers [8] to energy harvesting devices such as the WISP [9]. Many of these applications have critical safety or performance properties that

Figure 1: Experimental hardware setup



16x MSP430fr5969@16Mhz, a very low power cluster

could benefit from accurate, architecture-specific functional models. For example, one might want to prove that a program executes in some fixed number of cycles known to be safely within the energy budget available on an energy harvesting device. The proof will not be very convincing unless the cycle counts it assumes are accurate.

3. TIMING BEHAVIOR

TI provides cycle counts for the MSP430 ISA, in Tables 4-8, 4-9, and 4-10 on pages 141-142 of revision N of the user's guide [10]. But how accurate are these numbers?

To find out, we compare this model with the behavior of real processors. We divide this task into two separate parts. First, we measure the behavior of the real processors in our array of evaluation boards. Then we use the SMT solver Z3 to reverse engineer a model from these measurements, which we can compare directly to the model provided in the user's guide.

3.1 Measuring Cycle Counts

We would like our model of timing behavior to provide an accurate cycle count for any sequence of instructions. Unfortunately, there are too many possible sequences, or even individual instructions (an MSP430 instruction might be up to 48 bits long, including extension words) for us to measure all of them. We need to apply domain knowledge to come up with a manageable set of measurements that will demonstrate all of the interesting behaviors.

The user's guide makes the assumption that there are only a few classes of instructions, and each class has a fixed cycle count, regardless of surrounding instructions. To guide our measurements, we make similar assumptions for our own model. Specifically, we distinguish 4,256 classes of instructions with potentially different timing behavior, based on instruction operation, bitwidth, addressing modes, and registers used. We assume that general-purpose registers are indistinguishable from each other, but other registers like the program counter and stack pointer are not.

Examining all of the binary encodings reveals some inconsistencies in the manual. Some combinations of addressing modes and registers do not have a defined behavior; we could choose to test them anyway and infer the behavior ourselves, but for simplicity we simply exclude them from our model. Other combinations produce predefined "constant generator" values. In these cases, addressing modes that would otherwise be redundant or nonsensical are instead used to specify common constants like 1 and 2 in a more compact way. It seems reasonable that these encodings could have interesting timing behavior, but the manual does not describe it explicitly.

Once we've defined the classes, we can enumerate programs and measure their timing behavior. We measure all executable sequences of a single instruction (some are invalid, as they crash the debugger or cause control flow that disrupts measurement) and almost 12 million of the 18 million sequences of two instructions. Ideally, we would follow an approach similar to the one emloyed by Campbell and Stark in [11], and use an SMT solver to either find inputs for all possible instruction sequences that would allow us to run them or prove that no such inputs exist. Instead we use a simpler heuristic approach, which fails to find initializations for about 1.3 million sequences that might have them. Obviously this could be improved, but as we'll see finding a model for the subset that we can measure is already difficult.

3.2 Synthesizing a Model

With the data from our measurements, we can check the accuracy of the timing model in the user's guide. If we only look at measurements of single instructions, then the TI model is mostly correct, except for a few ambiguities and one discrepancy. We present an annotated version of this model in Table 1.

However, if we examine sequences of more than one instruction, we find inconsistencies. For example, based on the documentation we would expect the pair

```
SXT     @R4+
SXT     @R4+
```

to take 6 cycles, but instead it takes 7.

What's going on here? Can we produce a better model that will let us predict it?

The root cause seems to be a pipeline stall or something similar when executing sequences of single-operand instructions with register indirect or autoincrement mode. The first instruction in the sequence takes 3 cycles, as expected, but the next takes 4. This repeats; the next takes 3 cycles again, then 4, then 3, and so on. While this behavior is easy to explain, it is a little tricky to put into a formal model.

The problem is that this behavior depends on state. We can't implement it with a stateless function like a lookup table, even if we provide information about other instructions in the sequence. Fortunately, tools from formal methods are clever enough to reason about state.

To find a better model, we use Z3, a constraint solver for satisfiability modulo theories, or SMT. We can express our measurements to Z3 as a set of constraints on the behavior of a function which tells us the cycle count of an instruction in a sequence. Repeated application of this function to all of the instructions in a sequence allows us to implement our model.

Given these constraints, Z3 will search for an implementation of our timing function that is consistent with all of our data—essentially performing program synthesis for a timing model. If we want to reproduce the model from the user's guide, we specify the function to take just an instruction

Table 1: Annotated timing model from TI user's manual

Double operand instruction cycle counts

source	destination	instruction	cycles
Rn, any CG*	Rn, not PC	all	1
	PC	`BIT`, `CMP`†	1
		all other	3
	X(Rn), ADDR, &ADDR	`MOV`, `BIT`, `CMP`	3
		all other	4
#N	Rn, not PC	all	2
	PC	`BIT`, `CMP`†	2
		all other	3
	X(Rn), ADDR, &ADDR	`MOV`, `BIT`, `CMP`	4
		all other	5
@Rn, @Rn+	Rn, not PC	all	2
	PC	`BIT`, `CMP`†	2
		all other	4
	X(Rn), ADDR, &ADDR	`MOV`, `BIT`, `CMP`	4
		all other	5
X(Rn), ADDR, &ADDR	Rn, not PC	all	3
	PC	`BIT`, `CMP`†	3
		all other	5
	X(Rn), ADDR, &ADDR	`MOV`, `BIT`, `CMP`	5
		all other	6

Single operand instruction cycle counts

mode	`RRA`, `RRC`, `SWPB`, `SXT`,	`PUSH`	`CALL`
Rn, CG*	1	3	4
@Rn	3	3	4
@Rn+	3	3	4
#N	N/A	3	4
X(Rn)	4	4	5
ADDR	4	4	5
&ADDR	4	4	5‡

Other cycle counts

`RETI`	5
any jump `JZ`, `JNZ`, `JC`, `JNC`, `JN`, `JGE`, `JL`, `JMP` whether taken or untaken	2

These tables are taken directly from [10], table 4-8, 4-9 and 4-10 on page 141-142 as of revision N, with three clarifications:

* We find that all constant generator sources have the timing of register sources.

† We find that `BIT` and `CMP` do not incur the typical 2-cycle penalty for using the PC as a destination, as they do not actually write the destination. The user's guide does not state this explicitly.

‡ The user's guide gives this latency as 6 cycles; we measure 5.

class as input and return a cycle count and provide only the constraints from our measurements of single instructions. An example encoding of the constraints is given in Equation (2) of Figure 2. In just a few seconds, Z3 finds an implementation for this function that exactly matches the model in Table 1.

But no function specified in this way can satisfy the constraints from our measurements of two-instruction sequences. Since there is no solution to our constraints, Z3 gives us something else: a small subset of constraints that cannot all simultaneously be satisfied, or unsat core. From this unsat core we can automatically extract the instruction sequences that generated the constraints and present a counterexample to the programmer. For example, our tool reports the following for the state-dependent behavior of single-operand instructions:

```
MOV     &ADDR (R2, 0x350), Rn (R14)
SXT     @Rn+ (R4)
SXT     @Rn+ (R4)
SXT     @Rn+ (R4)
13 total cycles

MOV     &ADDR (R2, 0x350), Rn (R14)
SXT     @Rn+ (R4)
SXT     @Rn+ (R4)
10 total cycles

MOV     &ADDR (R2, 0x350), Rn (R14)
MOV     &ADDR (R2, 0x1d00), Rn (R4)
MOV     &ADDR (R2, 0x1d00), Rn (R4)
MOV     &ADDR (R2, 0x1d00), Rn (R4)
12 total cycles
```

Figure 2: Example SMT encodings

```
MOV &0x350, R14                                  (1)
SXT @R4+
SXT @R4+
 10 total cycles
```

$$p \implies f_{timing}(\texttt{MOV.W &ADDR Rn}) + f_{timing}(\texttt{SXT @Rn+}) + f_{timing}(\texttt{SXT @Rn+}) = 10 \tag{2}$$

$$\begin{aligned} p \implies\ & s_0 = f_{initstate}() \\ & \wedge s_1 = f_{state}(\texttt{MOV.W &ADDR Rn}, s_0) \\ & \wedge s_2 = f_{state}(\texttt{SXT @Rn+}, s_1) \\ & \wedge (f_{timing}(\texttt{MOV.W &ADDR Rn}, s_0) \\ & + f_{timing}(\texttt{SXT @Rn+}, s_1) \\ & + f_{timing}(\texttt{SXT @Rn+}, s_2) = 10) \end{aligned} \tag{3}$$

Two ways (2) and (3) of encoding the constraint that the instruction sequence (1) takes 10 cycles to execute. The guard predicate p is unique to the trace and is used by the solver to identify this particular trace in an unsat core. f_{timing} is the timing function we want to synthesize; (3) also introduces an update function to track persistent state as the instructions are executed.

The last trace serves to fix the cycle count of the measurement `MOV` instruction that must occur at the beginning of all of our traces at 3 cycles; the other two traces exhibit the behavior. This set of traces is interesting because it isn't just a counterexample to one timing model. It is a counterexample to all possible models that could exist under our specification, a set of traces that is in some way hard to model.

A simple script could easily check our measurements against the model from the TI user's guide and flag the first two traces as incorrect, but that would give us a very incomplete picture: both of those traces seem to have consistent timing if we assume the MOV instruction takes 4 cycles. Unsat cores allow us to use Z3 (or any other solver that supports unsat core generation) to reason globally about a very large set of constraints, in our case tens of millions, and condense the outcome down enough that it is readable by a human.

Seeing this counterexample, and perhaps a few more with different numbers of SXT instructions, a programmer can easily figure out what is happening. But what to do about it? As a model synthesizer, Z3 is very flexible: we can specify any signature we want for our timing function. To model state, we introduce a new type with some finite number of values—here 2 is sufficient. We change our specification of the timing function to take a value of this type as input, and simultaneously synthesize another function that updates the state between each instruction in a sequence. Instead of a simple lookup table, Z3 is now searching for a state machine that satisfies our constraints. Equation (3) of Figure 2 gives an example of constraints from this encoding.

This is sufficient to model the discussed state-dependent behavior of single-operand instructions, but it doesn't quite scale to cover all state-dependent timing behavior of the processor. There seem to be a lot of dark corners involving the PUSH instruction:

```
MOV     &ADDR (R2, 0x350), Rn (R14)
PUSH    @Rn+ (R2)
SXT     Rn (R4)
7 total cycles

MOV     &ADDR (R2, 0x350), Rn (R14)
PUSH    @Rn+ (R2)
JC      4, taken=True
8 total cycles

MOV     &ADDR (R2, 0x350), Rn (R14)
PUSH    @Rn+ (R4)
JC      4, taken=True
9 total cycles

MOV     &ADDR (R2, 0x350), Rn (R14)
PUSH    @Rn+ (R4)
SXT     Rn (R4)
7 total cycles
```

We can actually handle this counterexample, and many others, by increasing the number of states to 4. The resulting model takes an entire day and 60GB of memory to synthesize (unfortunately Z3 is difficult to parallelize), but it does accurately predict the timings of all two-instruction sequences in our dataset. Unfortunately, it doesn't generalize beyond that. By extracting traces from real programs, we can find sequences of 3 instructions that the model is not correct for.

The most troubling problem with the accurate 4-state timing model is not the cost of producing it (which nonetheless gives us an idea of the scalability limitations of Z3) nor its inaccuracy on longer sequences. It is the difficulty of explaining the model once it has been synthesized.

The simple timing model in Table 1 fits on half a page. In contrast, the 4-state model is implemented as two lookup tables, one to keep track of state and the other to provide cycle counts, each with multiple thousands of entries. Even though the 4-state model is more accurate, it is probably less useful to a programmer, as it conveys no insights about the MSP430 processor, just a black box oracle that usually predicts the cycle count of a sequences of instructions. For all its accuracy, it lacks explicable structure.

Perhaps a better specification would allow us to synthesize an explicable timing function, and scale it to longer sequences. The limitations we face are not in our ability to collect data or solve constraints, but in our ability to state the right constraints to solve. As we add more state to deal with counterexamples that we don't fully understand, the implementation quickly grows beyond our ability to explain it.

A potential workaround is to limit the scope of the formal model. For example, most of the complex timing interactions can be avoided if we require that code place at least one NOP in between other instructions. This would have a negative effect on both the size and runtime of programs that are legal under our model, but it might be an acceptable trade-off if knowing the exact timing behavior is important.

4. REGISTER OPERATIONS

TI describes the behavior of the MSP430 ISA with a mix of text and pseudocode found on pages 158-208 of revision N of the user's guide. Given the inconsistencies we found with the timing model, we might want to ask: is this documentation sufficient to implement a formal semantics?

Similar to our approach for checking timing behavior, we can answer this question by measuring the behavior of real processors and using the data to construct a formal model which we compare to the documentation. We examine a convenient subset, specifically the behavior of two-operand arithmetic instructions on registers.

Operations on two 16-bit words have billions of possible inputs, too many for us to test exhaustively. However, the MSP430 also supports byte-width operations. With only 20 bits of input (8 for each operand, and another 4 for the flag bits of the status register), we can exhaustively test the behavior of these operations with only a million measurements, which we can perform in a matter of hours.

4.1 Synthesizing Bitvector Programs

We formalize the semantics of arithmetic operations as bitvector programs. This way we can synthesize our formal model using a tool for program synthesis called Synapse [2].

Synapse allows programs to be specified as metasketches, sets of program specifications, or sketches, that it can search in an orderly way. The arithmetic operations we want to model are quite simple, so we get good results with a simple metasketch: the set of all bitvector programs in SSA form, of increasing length. Synapse defines a formal language and interpreter for these programs, bv, which we adopt for our semantics as well.

For most arithmetic operations, the synthesis problem is rather trivial. The operations themselves (ADD, SUB, bitwise AND) tend to be implemented as primitives in the bv language. Most of the actual bv operations in the synthesized programs are devoted to masking the result to the correct number of bits. The hardest part is providing a manageable set of constraints, as the constraint formulas for these program synthesis problems tend to be relatively larger compared to the size of the data than the ones we use to synthesize timing models.

We find that a counterexample-guided search strategy works well. First we provide constraints from only a few hundred inputs to synthesize a candidate program. This candidate might be correct for all inputs; since there are only a million, we can run it on all of them to find out. If it gets an input wrong, then we add that one to our set of constraints and synthesize a new candidate, until we find a final program that is correct in all cases.

There is also a bit of trouble determining the correct values for the flag bits in the status register. Trying to synthesize the value of the status register all at once is extremely difficult—the expression is usually a bitwise OR of several unrelated subexpressions, which becomes quite long when expressed in bv. Fortunately, domain knowledge indicates that we can break this expression down and synthesize the flags individually. Even then, the flag expressions are often harder to synthesize than the operations themselves, as they require computing the operation first and then doing something further to the result. Things are more manageable if both the inputs and outputs of arithmetic are provided as inputs to the flag computations.

Inspecting the synthesized bitvector programs, we find that they agree with the ISA documentation. However, one instruction presents a particular challenge, as its behavior is undefined on a large fraction of its inputs.

4.2 DADD

The MSP430 ISA includes an arithmetic instruction for binary-coded decimal addition, DADD. The ISA documentation on page 173 presents the operation of this instruction as “src + dst + C $\rightarrow$ dst (decimally).” The text description clarifies that “the result is not defined for non-BCD numbers,” i.e. any number that contains a 4-bit nibble not in the range 0-9.

Defined or not, we can still measure the behavior of the operation on non-BCD inputs. We find that the processor doesn’t crash, but as expected the results can be surprising, though fortunately still deterministic. For example, assuming that the carry flag is 0, 0xff + 0xff produces a result of 0x54 and sets the overflow and carry flags. Can we synthesize a model that explains this?

DADD is harder to synthesize than the other MSP430 arithmetic operations because it doesn’t have a direct counterpart in bv. Even if it did, it seems unlikely that the two would agree on all non-BCD inputs. Synthesizing a full 8-bit DADD implementation as a single bv program proves problematic because it’s far too long—even with a timeout of a day, we can only synthesize programs of about 7 or 8 bv operations.

To work around this limitation, we apply domain knowledge to simplify the task. DADD operates on groups of 4 bits, each representing a single decimal digit. We can think of it as the combination of an iteration strategy and a computation: first it looks at the first 4 bits of each input and performs the computation to produce the first 4 bits of the output, then it looks at the next 4 bits and performs the same computation, and so on. The computation is what we want the synthesize.

Figure 3: DADD iterator

```
(define (<< x i)
  (arithmetic-shift x i))
(define (>> x i)
  (arithmetic-shift x (- i)))

(define (iter-4n/cv n compute/c compute/v)
  (define (op sr a b)
    (let ([carry-in (bitwise-and sr 1)])
      (for/fold ([c carry-in]
                 [v 0])
                ([i (in-range 0 (* n 4) 4)])
        (let*
         ([a_n (bitwise-and (>> a i) #xf)]
          [b_n (bitwise-and (>> b i) #xf)]
          [c_n (compute/c c a_n b_n)]
          [v_n (compute/v c a_n b_n)])
          (values
           c_n
           (bitwise-ior v (<< v_n i)))))))
    op)
```

Iterative strategy for computing DADD in 4-bit increments. Definitions for << and >> are provided for clarity. The outer function iter-4n/cv takes a fixed number of steps and implementations for the computations and produces another function that iterates to produce the final result.

The iteration strategy is expressed as a Racket function in Figure 3. Note that we have to be careful with the carry; we treat it as a separate computation to synthesize, but instead of bitwise OR-ing the output together at different offsets like we do with the output value bits, we pass the carry along to the next step of iteration.

We can check if this strategy is valid by looking at the observed behavior for inputs in the range of one step of the iteration: 0 or 1 for the carry, and 0-15 for both inputs. Masking off the the fifth bit of the output value gives us the output carry. A lookup table of these measurements is sufficient to implement the full behavior of DADD, given the iteration strategy. However the 512-entry table is cumbersome and does not provide much intuition about why DADD behaves the way it does.

The reduced, one-nibble table is almost simple enough for us to synthesize bitvector implementations of the value and carry computations to use with our iteration strategy. However, synthesis still times out. Even though there are only 512 pairs of inputs and outputs, expressing the computation still takes too many bv operations. To amend this, we apply one final piece of domain knowledge. DADD differs from ordinary bitvector addition in the way it handles the carry. Specifically, if addition produces a digit value greater than 9, we need to add 6 more, so that we carry to the next digit. If we implement this as a primitive in bv, we can then successfully synthesize both the value and carry computations

Figure 4: Synthesized 4-bit DADD

```
;; DADD 4-bit value computation
(program 3
 #| 0: c |#
 #| 1: a |#
 #| 2: b |#
 (list
  #| 3|# (bvadd 0 1)
  #| 4|# (bvadd 2 3)
  #| 5|# (bcd_carry 4)
  #| 6|# (bv 15)
  #| 7|# (bvand 5 6)))

;; DADD 4-bit carry
(program 3
 #| 0: c |#
 #| 1: a |#
 #| 2: b |#
 (list
  #| 3|# (bvadd 0 2)
  #| 4|# (bvadd 1 3)
  #| 5|# (bcd_carry 4)
  #| 6|# (bvor 4 5)
  #| 7|# (shr4 6)
  #| 8|# (bv 1)
  #| 9|# (bvand 7 8)))
```

Synthesized bitvector programs used to compute DADD 4 bits at a time with the iteration strategy from Figure 3, as output by Synapse. Programs are in SSA form and refer to operands by the index of their binding location, provided in comments. The first three locations hold the three arguments. The `bv` instruction does not take a binding location; it instead returns a bitvector value equal to its argument. Each program returns the value of its last binding.

in under a minute.

Figures 4 and 5 present the resulting 4-bit DADD computations, along with our manually provided implementation of the BCD carry operation. The solver is allowed to use the `bvand`, `bvadd`, `bvor`, `shr4`, and `bcd_carry` instructions, as well as the `bv` constructor with values of 1 and 15. It searches the space of programs in SSA form in order of increasing length until an implementation is found.

This might seem like a lot of code, but the model is expressed much more compactly as a bitvector program than as tables of inputs and outputs, and the iteration strategy gives us a natural way to extend it to operations wider than 8 bits. It also lets us give a coherent explanation why 0xff + 0xff is 0x54:

0xf is 15. 15 + 15 is 30. 30 is greater than 9, so add 6 more to get the proper BCD carry behavior. That's 36; taking the low 4 bits gives us the first digit, 4. When we move on to the next digit, we have a carry of 1, because the previous digit was larger than 15 after we added 6 for the BCD carry. Interestingly, the `bv` implementation has to OR together the values before and after the BCD carry so that a carry to the fifth bit isn't lost due to another carry into the sixth bit. Otherwise, the calculation of the second digit is like the first; 15 + 15 + 1 + 6 is 37, take the low 4 bits to get 5 and a carry out of 1.

Figure 5: Final DADD implementation

```
(define (bcd-carry x)
  (if (or (> x 9) (< x 0)) (+ x 6) x))

(define (dadd-compute/v c a b)
  (bitwise-and (bcd-carry (+ c a b)) 15))

(define (dadd-compute/c c a b)
  (bitwise-and (>> (bitwise-ior
                    (+ c a b)
                    (bcd-carry (+ c a b)))
                4) 1))

(define dadd.b (iter-4n/cv 2
                           dadd-compute/c
                           dadd-compute/v))
```

Synthesized bitvector programs expressed in native racket. The `bcd-carry` function is implemented by hand and provided to the synthesizer; `dadd-compute/v` and `dadd-compute/c` are translated from the Figure 4. The final implementation of DADD is provided by applying the iterator from Figure 3 in `dadd.b`. Passing `n=2` yields an 8-bit DADD. We could also produce implementations of the 16 and 20-bit versions of the instruction by passing `n=4` and `n=5`.

This explanation, and the bitvector programs that let us provide it, are particularly interesting because they are so high-level. A circuit diagram of the relevant hardware logic in the MSP430 processor might also let us accurately predict the behavior, but even if we could obtain such a diagram, it would be less useful to a programmer who is used to thinking about Racket code or bitvector programs. Program synthesis tools let us provide our models as mathematical expressions, which are both easy to reason about and convenient for reuse in other formal analyses.

5. CONCLUSION

Providing accurate models of processors is an important task to ensure the correctness and reliability of software that runs on them. As our experience with the MSP430 shows, it can be surprisingly difficult even for simple processors. And as processors become both more complex and more diverse, the problem isn't getting any easier.

Fortunately, and in contrast to the situation in neuroscience, we have formal tools that seem well suited to reasoning about processors. By combining the insight and domain knowledge of a human architect with the relentless thoroughness of a formal analysis, SMT solvers and program synthesis frameworks can help us to build models that are simple, generalizable, and demonstrably correct, as we saw synthesizing DADD, and even guide us in cases where our initial insight is insufficient, as we saw synthesizing timing models. Based on our explatory work, we draw three conclusions.

First, off-the-shelf SMT solvers and synthesis tools are strong enough to handle interesting datasets measured from real processors. This is perhaps surprising, given the size of our constraint formulas, or perhaps not, given the recent success of solvers on other industrial SAT instances. Either way, the ready availability of these tools is helpful.

Second, off-the-shelf tools are good at telling us when a formal model is wrong. In a sense, the same can be said about simple scripts; the point is that a script can call out to an existing solver and provide a useful explanation of anything that goes wrong, for example by returning unsat cores, at a relatively small cost in terms of complexity.

Third, formal tools can help us to create better formal models, but only if we help them in return. Applying domain knowledge like the iterative nature of DADD is necessary to get good results. In some cases tools can help us refine our knowledge, or at least check that it is accurate, but they can't discover it on their own. And as we saw synthesizing timing models, sometimes the right domain knowledge is hard to come by.

Formal models of processors in languages such as L3 [12] and machine-readable specifications such as recent work at ARM [13] are becoming more prevalent. We believe that formal methods can help accelerate this trend, not just by checking models but by creating them.

6. REFERENCES

[1] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, Springer, 2008.

[2] J. Bornholt, E. Torlak, D. Grossman, and L. Ceze, "Optimizing synthesis with metasketches," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, (New York, NY, USA), pp. 775–788, ACM, 2016.

[3] E. Torlak and R. Bodik, "Growing solver-aided languages with rosette," in *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, (New York, NY, USA), pp. 135–152, ACM, 2013.

[4] E. Jonas and K. Kording, "Could a neuroscientist understand a microprocessor?," *bioRxiv*, 2016.

[5] P. Godefroid and A. Taly, "Automated synthesis of symbolic instruction encodings from i/o samples," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, (New York, NY, USA), pp. 441–452, ACM, 2012.

[6] D. Beer, "Mspdebug." `http://dlbeer.co.nz/mspdebug/`.

[7] C. Chong-chong, "The ultra low power consumptive lock based on msp430f122 [j]," *Techniques of Automation and Applications*, vol. 5, p. 041, 2009.

[8] C. Rotariu, V. Manta, and H. Costin, "Wireless remote monitoring system for patients with cardiac pacemakers," in *Electrical and Power Engineering (EPE), 2012 International Conference and Exposition on*, pp. 845–848, IEEE, 2012.

[9] A. P. Sample, D. J. Yeager, P. S. Powledge, A. V. Mamishev, and J. R. Smith, "Design of an rfid-based battery-free programmable sensing platform," *IEEE Transactions on Instrumentation and Measurement*, vol. 57, no. 11, pp. 2608–2615, 2008.

[10] TI, *MSP430FR58xx, MSP430FR59xx, MSP430FR68xx, and MSP430FR69xx Family User's Guide*.

[11] B. Campbell and I. Stark, "Randomised testing of a microprocessor model using smt-solver state generation," *Science of Computer Programming*, vol. 118, pp. 60–76, 2016.

[12] A. Fox, "Directions in isa specification," in *International Conference on Interactive Theorem Proving*, pp. 338–344, Springer, 2012.

[13] A. Reid, "Trustworthy specifications of armr v8-a and v8-m system level architecture," in *Proceedings of Formal Methods in Computer-Aided Design (FMCAD)*, 2016.