6.888: Certified Systems Software 2015

VerifiedDSP: Verifying Digital Signal Processing Designs in Coq

`https://github.com/JeremyRubin/VerifiedDSP`

Jeremy Rubin

jlrubin@mit.edu

Abstract

Digital Signal Processors and microcontrollers are used widely in a wide range of devices and machines. There are many life critical applications, including medical equipment, transport, and communications. It is therefore of great importance to ensure the proper functionality of such devices. Many of these devices are simple, preferring to rely on a tried and true 8-bit architecture without a full operating system so that they may more easily reason about real-time responses to events. For instance, it could be disastrous to have a garbage collection pause while trying to apply the brakes of a car. However, this simplicity comes at a cost; without higher level constructs a programmer must manually write and check a lot more code due to the high resource constraints. Furthermore, it is hard for a programmer to verify that the code they wrote is correctly translated into the binary loaded onto chip, the compiled version may have different properties than desired.

This paper presents a new framework, VerifiedDSP, for programming 8-bit Intel 8051 series microcontrollers and designing embedded Digital Signal Processing systems. It includes an 8051 simulator, a prototyping framework for mocking out specifications, and some higher level constructs to help programmers formalize the behavior and run time of control loops.

1. Introduction

Modern machines often rely on a combination of mechanical systems, electrical systems, sensors, and software. The advent of embedded systems has allowed for much more accurate and complicated designs to be built because they allow for tighter feedback loops as well as convenient ways to interface between disparate systems.

Often times these embedded systems are built with simple microcontrollers (μCs) such as 8-bit because a more powerful computers are not needed, energy-inefficient, and are harder to reason about for real-time applications. In some sectors, Real-Time Operating Systems (RTOS) have become popular due to programming ease, but there are many applications where a lighter weight solution is needed. Many systems which require light weight microcontrollers also happen to be life-critical. Unfortunately, there have been many instances where the life-critical nature of this role was made all too clear with catastrophic failures.

This paper presents an effort to reduce the likelihood of engineering errors as well as toolchain errors when building a Digital Signal Processing unit (DSP), including complex designs using a microcontroller such as the 8-bit Intel 8051 series microcontrollers[1].

There are two main components of this project: a trusted-but-proof-ready 8051 simulator, and utilities which assist programmers in prototyping formally reasoning about behavior of embedded designs. Behavior is characterized by traces on "pins".

These traces can encode many properties such as return values, execution time, pin output, and non interference.

The internal behavior of the microcontroller is provable, but the higher level specifications of a design, only deal in traces. The internal state of the microcontroller can still be reasoned about for proofs, but it is masked to allow for a better development cycle. Microcontroller functionality can be mocked out, and refinement can be shown between a functional description and a microcontroller.

This system will have many limitations which may affect its real-world usefulness, which will be expounded in the discussion.

2. Motivation

There are several concrete examples of embedded system failures which motivate this work.

2.1 Medical Devices

The Therac-25 was a radiation therapy machine which lethally overdosed 6 people [11]. While a large portion of the of the blame for this could be placed on the lack of hardware failsafe, it was ultimately unverified software which caused the deadly malfunction.

Another interesting medical device is the pacemaker. A 2004 study saw that approximately 1/1,200 pacemaker reprogramming events loaded faulty programs onto the device, which could cause a patient's heart rate to elevate to 185 beats per minutes. The fault was due to multiple timer interrupts being enabled at the same time, when only one was supposed to be enabled at any given time [10].

[1] This chip was chosen because the author is familiar with them and they are very widely deployed.

2.2 Transport

Automobiles are also a large concern with regards to embedded systems as they rely on multiple for the core functionality of the vehicle. Over the past decade, there have been a number of bugs and errors caused – or suspected to be caused – by faulty software. For instance, several major automobile manufacturers – Nissan, Honda, and Subaru – recalled vehicles for faulty embedded systems which could deploy airbags during normal vehicle operation or start the car at random while parked [12].

2.3 Communications

Networked mobile devices use small hardware secure chips called subscriber identification module cards (SIM cards) to reliably identify clients connecting to the network. They contain a μC which manages the processing of cellular data, such as encryption and signatures for requests from a certain number (such as send a text-message). Recently, exploits have been found that allow a remote attacker to take control of the SIM card and issue illegitimate requests [2].

3. Design Overview

The following subsections will contain an overview of the major parts of this project: subsection 3.1 will discuss the component definitions; subsection 3.2 will discuss the wiring framework used to connect the components; subsection 3.3 discusses the execution of a wiring;subsection 3.4 will discuss the microcontroller implementations.

3.1 Component

The fundamental definition in VerifiedDSP is the *IO* module, which has 3 main definitions detailed in Listing 1.

```
Module IO.
  Definition t := nat.
  Definition trace := list (list t).
  Inductive func :=
  | fn_args : nat → (trace → t) → func.
```

```
End IO.
```

Listing 1. IO module definition

IO.func The most important definition is IO.func. An IO.func is a pair of a number of arguments and a function which can process an IO.trace to produce an output IO.t.

IO.t IO.t can be thought of as a *voltage*, and it is the type passed around by other modules.

IO.trace The IO.trace is simply a list of lists of the IO.t voltage. Each trace is implicitly numbered by their position in the top list.

```
Definition integrator : IO.func :=
    IO.fn_args 1 (fun x ⇒suml (hd []
  x)).
Definition incrementor : IO.func :=
    IO.fn_args 1 (fun x ⇒len (hd [] x
  )).
Definition zero_rail : IO.func :=
    IO.fn_args 0 (fun _ ⇒0).
```

Listing 2. An integration unit, an incrementing unit, and an always 0 “rail”

Some example components are shown in Listing 2.

3.2 Wiring

At the core of this project is a way of connecting these components together to perform computations. This is modeled as an infinite breadboard with a single discrete time clock. There is a set of $\mathbb{N}$ pins that a device can connect to. Devices can read any pin, and can write to a single pin. Multiple outputs can be simulated by making duplicate copies of the same hardware which write to different pins. Listing 3 shows the breadboard definition.

```
Inductive wiring :=
| base : wiring
| watch_set : wiring → list nat →
              IO.func → nat →
              wiring
| just_set : wiring →
             IO.func → nat → wiring
| join : wiring → wiring → wiring
| doc : wiring → nat → string →
  wiring.
```

Listing 3. wiring type definition

watch_set observes a list of pins, and writes to a single pin.
just_set only has access to a single clock trace, but sets values without observing other state.
join joins two separate networks together.
doc adds non functional documentation for convenience, which is useful for more complex projects.
We define the following notations for the wiring:

$$w//m \sim> f \sim> n \equiv watch\_set\; w\; m\; f\; n \quad (1)$$

$$w*/m \sim> f \sim> n \equiv just\_set\; w\; f\; n \quad (2)$$

$$w1\; \sim\&\sim\; w2 \equiv join\; w1\; w2 \quad (3)$$

$$w \# p\; c \equiv doc\; w\; p\; c \quad (4)$$

Once a wiring is complete, it should be checked for correctness. The property valid_wiring, as described in Listing 4, checks this. These invariants are important for a correct design.

```
Fixpoint valid_wiring' w ins outs: Prop
  :=
  match w with
    | base ⇒
      set_intersect ins outs = ins
    | w' // from ~> fn_args n f ~> to ⇒
     ~set_in to outs ∧
      n = length from ∧
      valid_wiring' w' (union ins from)
          (set_add to outs)
    | w' */ fn ~> to⇒
     ~set_in to outs
     ∧ valid_wiring' w' ins
              (set_add to outs)
    | w1 ~&~ w2 ⇒
      let w1o := output_pins w1 [] in
      let w2o := output_pins w2 [] in
      valid_wiring' w1 (union w2o ins)
          (union w2o outs) ∧
      valid_wiring' w2 (union w1o ins)
          (union w1o outs)
    | w' # _ _ ⇒
        valid_wiring' w' ins outs
  end.
Definition valid_wiring w :=
    valid_wiring' w nil nil.
```

Listing 4. Check that all pins that are being read from are written to, that there is no contention for

pins (ie, two devices driving the same pin, and that all devices are fully connected to the proper number of pins.)

Wirings are not correct by construction. This is because it is desirable to produce incomplete modules which need to be *plugged in* to one another (ie, using a join) before being finished. The doc constructor is important for this so that developers may easily check to see what the other developer intended to be connected.

If two modules are needed in the same system, but they conflict there is a function *rewire* which provably reconnects a system so as not to conflict on any pin. It works by adding the max pin number + 1 of the other circuit to every pin numbering. If the smallest pin number in a system is greater than the largest in the other, then it is clear that they will not clash. The specific implementation should not be relied as many different rewirings could be performed. Understanding rewirings is another use for the documentation functionality.

```
Definition example :=
    base
    */  zero_rail ~> 0
    */  incrementor ~> 2
    //  [2] ~> integrator ~> 3
    #   3 "Integrated incrementor".
```

Listing 5. A simple wiring

Two modules which obeys valid_wiring can not interfere with one another and can be safely joined. A valid wiring can be joined with an arbitrary (valid or invalid) wiring an either become invalid or stay valid, if it stays valid then certain properties should be preserved as shown in Listing 6.

```
Theorem non_interference1 :
forall w w',
valid_wiring w →
valid_wiring (w ~&~ w') →
forall n t,
let orig := find_trace t (run w n) in
let mod := find_trace t (run (w ~&~ w')
    n) in
match orig, mod with
    | Some a, Some b ⇒ a = b
    | Some a, None ⇒ False
    | None, Some a ⇒ True
    | None, None ⇒ True
end.
```

Listing 6. On adding a new module w´ to a valid wired w, for all traces the outputs are the same if present, or may be added if not present.

3.3 Running

Once a wiring is constructed, it can be *run*. Running a wiring involves walking over the wire structure and applying the latest traces to determine the next states. Functions are computed over the entire trace at each step. This could be optimized, but for proofs it is simpler. The run function can be proven to preserve history (Listing 7) and present consistent views of pins to all functions in the structure. For example, the trace on pin 3 from Listing 5 is [36; 28; 21; 15; 10; 6; 3; 1; 0; 0].

```
Theorem no_modify_history: forall n w,
    run w n = tl (run w (S n)).
```

Listing 7. The next run is identical up to the latest result

3.4 8051

An IO.func can be implemented as any function which can operate over a IO.trace. Thus, it is possible to design significantly more complicated IO.funcs than those in Listing 2. VerifiedDSP includes a general purpose 8051 IO.func that can be integrated into any design. The basic signature of the device is as follows:

```
Definition i8051_Component
  bin adc dac :=
  fn_args (8*4) (fun t ⇒
                        let ps := traces t
    adc in
                        dac (
    run_8051_bin_string bin ps)).
```

Listing 8. The adc and dac provide conversions to and from the networks type, the bin is a list of bytes to be loaded into code memory. The 8051's ports must all be connected. First the program is loaded into memory, then the component can be executed over any set of 32 pin traces. There is

only one output pin, but as noted earlier duplicate devices can be used to simulate multiple output pins.

A specific program on an 8051 can be shown to simulate another function using the relation func_same in Listing 9.

```
Definition func_same (i i':IO.func) :=
    forall (tr:IO.trace),
    let lengths := (map
    (fun x ⇒ length x)
    tr) in
    let f1 := hd 0 lengths
    in
    fold_left (fun acc  x
   ⇒x=f1∧acc) lengths
    True→
    match i, i' with
      |IO.fn_args n f,
     IO.fn_args n' f'⇒
       length tr = n →
       f tr = f' tr ∧ n = n'
    end.

(**
[2;0;0] is 8051 binary for:
.org 0h
main:
ljmp main
 **)
Theorem simulates : func_same (
    i8051_Component [2;0;0] threshold
    dac)
(IO.fn_args (8*4) (fun _ ⇒ 0)).
```

Listing 9. Check that over any well formed IO.trace, two IO.funcs have identical results. In Theorem simulates, an 8051 is shown to simulate a zero function

3.4.1 Implementation and Progress

A complete VerifiedDSP 8051 model is still in progress. Currently, the 8051 can run approximately 5 instructions (eg, JMP, LJMP, SETB, CLR, ANL). The implementations are heavliy derived from RockSalt [8], and is basically a port from their x86 model to 8051. Listing 10 shows the RTL implementation of the SETB instruction.

```
Definition conv_SETB (op1:operand) :
    Conv unit :=
match op1 with
    | Bit_op (bit_addr baddr) ⇒
      if is_valid_bit_addr baddr then
        let bsel := and baddr 3 in
        let addr := and baddr (~3) in
        let ormask := shl 1 bsel in
        ormaskReg <− load_int ormask;
        a <− load_int addr;
        v <− read_byte a;
        v' <− arith or_op v ormaskReg;
        write_byte v' a
      else
          emit error_rtl
    | _ ⇒ emit error_rtl
end.
```

Listing 10. The SETB instructions Register Transfer Logic implementation. The argument type is first checked, and then the argument is checked to be a valid bit address (not all values are bit addressable). The proper bit is or'ed into memory.

The semantics of the processor are basically correct, remaining work would be in implementing the remainder of the instruction set and developing proof automation to make it easier to develop with.

3.5 Extraction

As a last note, the designs can be extracted into running programs in Ocaml using the Coq extraction facilities. This is of dubious value given that mostly the correctness of designs is interesting (and can thus be done from within Coq), but it could be useful for running tests on prototype designs before trying to formally prove them.

4. Related Work

Several projects have related goals:

4.0.1 Bedrock

Bedrock is a project which facilitates the formal verification of low level code. It is implemented as a Coq Library, but it provides significant enough extensions and capabilities that it is more aptly described as a language for separation logic. Bedrock provides significant instruction on the implementation of a separation logic system[3][4].

4.0.2 Reflex

Reflex is a project which makes a verified message passing kernel that can interpret a set of user defined rules about non interference and causality. Reflex is not intended for embedded systems, but the model of non interference and causality is instructive for a style in which one might prove properties about an embedded system [7]. For example, one such property could be that seems natural to try to prove in a Reflex style could be that at least 100 cycles pass in between asking for and reading a value from an Analog Digital Converter.

4.0.3 Correctness Proofs for Device Drivers in Embedded Systems

Duan and Regher[9] discuss a system which can verify low-level device drivers. Their approach allows for separately clocked components interactions to be modeled and verified. They use a preexisting ARMv4 simulator and develop a formal semantics for interacting with UART. Their proofs are all manual, and they hope to build proof automation on top of their proofs for UART.

4.0.4 RockSalt

This effort derives heavily from RockSalt. RockSalt was a project to write a formally proven version of Native Client(NaCl), Google's tool for generating and checking untrusted binaries which can then be safely run with confidence they will never escape the sandbox [8].

A major part of the RockSalt contribution was an excellent model of Register Transfer Language on top of which they modeled MIPS and x86 processors. The ability to end-to-end check properties of the binary is a powerful tool in guaranteeing functional correctness. Their effort was very instructive in the implementation of the 8051 model; it is an adaptation of their code.

4.0.5 Model Checking Software for Microcontrollers

This project formally models an 8-bit Atmega microcontroller. The project reasons about assembly code, but is designed to have proof automation for high level theorems from C, C++, and other programming langauges. There does not seem to be facilities for run time verifition [6].

4.0.6 An Approach for the Formal Verification of DSP Designs using Theorem Proving

This project is very similar in nature to Verified-DSP. They implemented a similar architecture using HOL. However, their implementation does not have a full microcontroller, just a Register Transfer Logic. They prove correct an FFT algorithm in this manner, and then are able to export it into a netlist for hardware synthesis. They seem to lack an elegant method of connecting modules together, unlike the wiring construct in Verified-DSP. There is no source code available for their implementation[1].

5. Discussion

5.1 Future Work

5.1.1 Usability Study

This framework cannot prevent all classes of bugs. In order to test the effectiveness of such a system (once there is more automation), it is imperative to run a user study to see if such a model helps. One such user study could testing and timing users on their ability to complete the following tasks with and without the simulation tools (given a small hardware test-bench):

1. Write a program which makes an LED blink with a certain frequency
2. Write a program which writes a copy of an array into a specific memory location, maps $f(x) = x + 1$ over the array, without modifying any other memory.

3. Write a program which when a push button is hit, turns one LED on at half brightness, and another LED at half brightness otherwise.

These tests would determine if these tools improves a programmer's ability to reason about runtime, non-interference, and memory safety.

5.1.2 Limitations

The current design fails to address several key embedded system properties and has a few weaknesses.

Simulator Correctness The simulator is not guaranteed to be correct. In fact, it is definitely not correct as certain aspects of the model are not fully implemented, such as setting ports IO mode. The model can be proven to be consistent within itself (ie, no confused parse), but there is no way to check that the model matches the real world 8051. A test-jig could be built to verify the 8051 simulator against real hardware.

Interrupts Interrupts are explicitly required to be disabled because they would add a lot of additional complexity to the system. One could potentially, using an 8051 timers, devise a scheme as follows:

```
timer isr:
disable interrupts
go to interrupts_over
set timer to priority high with size
  200 cycles
set all other interrupts priority low
enable interrupts
some interruptible loop
interrupts_over:
```

Listing 11. The timer ISR is used to guaranteed a bound on the amount of time interrupt might be executing.

Such a scheme allows for a provably bounded window on interrupts; this is of dubious value given that polling would also accomplish a similar goal with more deterministic performance.

More Hardware Interfaces Embedded systems fundamentally interface with hardware, which can have difficult to understand behavior. This work only has the semantics for an 8051 and some simple DSP designs. However, one exciting possibility would be the development of semantics for a library peripheral chips. This would allow programmers to verify more complicated and realistic designs.

Furthermore, an additional modulus could be added onto modules to support slower clock cycle than the rest of the network, which may be desirable to simulate communicating with faster/slower hardware.

Nondeterminism This work does not help with verification given unreliable hardware such as sensors or power supply.

Assembler Another challenge would be to implement an assembler so that code in higher level expressions can be reliably compiled. Making an assembler in Coq may have some other benefits in aiding later verifiable macro development [5]. Currently, the standard 8051 as31 compiler is used.

5.2 Reflections on Formal Verification

This was my first experience with formal verification. In sum, formal verification is hard, very hard.

5.2.1 Code Reuse

One of the great promises of formal verification is code reuse. Once algorithms or software are proven correct once, they can become a permanent fixture and a basis for more formalized software. In theory, at least. In practice, I found it very difficult to work with existing code bases for a number of reasons. First and foremost, it seems that the Coq compiler has breaking changes fairly frequently, as I was unable to build fairly recent code from the RockSalt project. The reason for this was, it seems, that Coq makes major tradeoffs in the proof scripts in terms of variable names. Another factor which made it difficult for me was just the overall complexity of the existing code. Hacking on something as complex as RockSalt really frustrated my progress, when I started developing the more novel parts of this project I was able to make a lot more progress because I knew what I wanted and was trying to do. Furthermore, I discovered a bug in the RockSalt code and would be interested to see if it exists in the main code as well and was not a result of my modifications.

```
Definition bad_immediate_16 :=
(field 8) @ ((fun r ⇒ Imm16_op
(@Word.repr 15 r)) :
    _ → result_m operand_t).

Definition immediate_16 :=
(field 16) @ ((fun r ⇒ Imm16_op
(@Word.repr 15 r)) :
    _ → result_m operand_t).
```

Listing 12. The RockSalt bug: the parser helper for immediate constants only reads the MSB of the constant

5.2.2 Abandon Proof!

Stating Theorems is much easier than proving them, and from my experience, a large portion of the benefit of formal verification is simply stating the theorems as it forces a deep reflection on the architecture and functionality of the code. That said, once I had Theorems, proving them seemed to be very difficult, and I abandoned many proofs. However, in the process of trying to prove them (and failing) I discovered some more critical bugs which prevented any progress from proceeding at all, which suggests that trying to prove things is useful even if you give up! I also found that development went more quickly when I posed the theorems then admitted them as I built more architecture.

5.2.3 What Did I Just Prove

Getting the higher level specifications correct in a verification effort is difficult. What is trying to be built? What semantics should it have? When the all the proofs are done, does it actually accomplish the desired goal? This is difficult to know.

5.2.4 Editors

I was a vim user before I started writing Coq code. Luckily evil-mode lessened the transition burden to emacs, but certainly a new editor was a large distraction!

5.2.5 Time

I found it hard to hack lightly on this stuff. Whereas with most projects I can make fine progress with an hours time, I almost never got anything done within the first three contiguous hours of work on this project. This high startup cost made making progress during a busy semester hard. However, scheduling large blocks of time to work on it was ultimately effective.

5.2.6 Interactivity

This is a really great UI feature for programming, and I'm surprised that more languages don't have an interactive compilation mode like Coq's. Perhaps it wouldn't be useful in other languages, but I found being able to quickly go line by line to be a very intuitive way to code compared to my traditional workflow.

5.2.7 Overview

Despite the obstacles faced, I really enjoyed doing this project. Writing Coq code and proofs is insanely addicting; I would routinely find myself at 5 in the morning telling myself to go to sleep. It is challenging, and sometimes I would spend hours trying to prove something trivial. Almost nothing compared to the exhilaration I felt when I got a tough proof Qed'd though! The puzzle solving nature really does something for me.

Writing Coq is certainly not like any other traditional programming language, including the functional varieties such as Haskell or Ocaml. Although it has its negatives, such as poor reusability of code and high barriers to entry, the positives outweighed it by far for me. Simply put, it's just a lot of fun!

5.3 Conclusion

This paper presents a novel framework, Verified-DSP, which can be used to verify and prototype Digital Signal Processing designs in Coq. This framework is very flexible, as it can serve as a backbone for prototyping digital signal processors because functionality can be mocked out and then refined with closer to hardware function descriptions (ie, microcontroller binaries or RTL).

Additionally, this paper details the author's first major experience with formal verification techniques. There were several obstacles and tactics to get around them that might be good advice for someone just starting to explore the field, and can hopefully guide future work in making formal methods more accessible.

Acknowledgments

A huge thank you to professors Frans Kaashoek and Nickolai Zeldovich for teaching the Certified Systems seminar at MIT. It has been one of the most interesting classes I have taken at MIT by far! Without their incessant encouragement and support, I would not have been able to complete this project.

References

[1] B. Akbarpour and S. Tahar. An approach for the formal verification of dsp designs using theorem proving.

[2] S. Anthony. The humble sim card has finally been hacked: Billions of phones at risk of data theft, premium rate scams. http://www.extremetech.com/computing/161870-the-humble-sim-card-has-finally-been-hacked-billions-of-phones-at-risk-of-data-theft-premium-rate-scams, July 2013.

[3] A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *PLDI*. Harvard University, 2011.

[4] A. Chlipala. The bedrock structured programming system, combining generative metaprogramming and hoare logic in an extensible program verifier. In *ICFP*. MIT CSAIL, 2013.

[5] A. K. et al. Coq: The worlds best macro assembler? In *PPDP*. Microsoft Research, 2013.

[6] B. S. et al. Model checking software for microcontrollers. Department of Computer Science of RWTH Aachen University, 2006.

[7] D. R. et al. Automating formal proofs for reactive systems. In *PLDI*. University of California, San Diego, 2014.

[8] G. M. et al. Rocksalt: Better, faster, stronger sfi for the x86. In *PLDI*. Harvard University, 2012.

[9] J. R. Jianjun Duan. Correctness proofs for device drivers in embedded systems.

[10] J. V. Levert and J. C. Hoorntje. Runaway pacemaker due to software-based programming error. In *PACE, Vol. 27*, page 1689. Department of Cardiology, Isala Klinieken, locatie Weezenlanden, Zwolle, the Netherlands, 2004.

[11] N. Leveson. *Safeware: System Safety and Computers*. Addison Wesley, 1995.

[12] J. Traenkenschuh. Secure your embedded systems now! http://www.informit.com/articles/article.aspx?p=2140093, October 2013.