ARM Instruction Decoder and Lifter

Corey Wingo, Saffat Ahmed, Nolan Kuo, Jacob Wiedemeier

12/15/2022

Intro / Motivation

The main focus of this project is the creation of a novel ARM instruction decoder and lifter that takes ARMv7 assembly and lifts it to an intermediary language (IL), Picanae in Coq in this case, that can be used for future works in proving the correctness of the ARM system.

ARM is the most widely used family of instruction set architectures, having a presence in the majority of smartphone processors. In addition, with Apple adopting the ARM architecture in its new M1 Chips, the need for security techniques that can guard against exploits is steadily increasing. Naturally, these techniques will need some way to verify correctness, hence the necessity of an ARM decoder and lifter.

To our knowledge, while there are Picanae systems that target similar architectures like RISCV, there is no ARM focused decoder that lifts instructions from a binary up to Picanae for usage in proofs of correctness. As such, our prototype system is novel in its targeted architecture and application.

Technical Approach

We first approached the technical challenge of how to obtain the bytes to decode. Fortunately for us, as we are not supporting thumb mode, all ARMv7 instructions are 4 bytes long. However, as we are decoding ARMv7 binaries, we must be able to assemble ARMv7 instructions into their binary representation. For that, we used CPUlator [1]. It can assemble ARMv7 instructions into their binary representations and even emulate them, as seen in Figure 1.

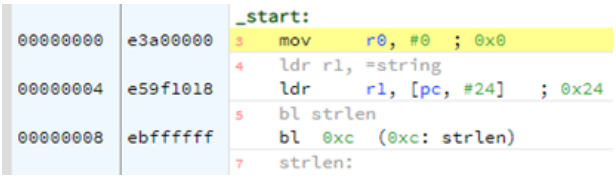

Figure 1. Assembly of ARMv7 instructions.

We found that using the CPUlator tool was very tedious. It required us to manually copy and convert the hexadecimal bytes to decimal and then to paste it into the file. To mitigate this tedium, we tasked one of our members to write a tool that would grab the bytes directly from a binary and compile it into a COQ list that we could simply run.

We had to convert the representation of the assembled instructions from hexadecimal to decimal for compatability with the Picanae system. This did not change any of the operations we could perform on the representation. We can still use critical methods such as xbits and shift.

Once we have the instruction represented as a decimal, we are ready to decode it. In order to decode the instruction, we have to first understand the instruction encoding, as explained in the ARM Architecture Reference Manual [2]. The architecture manual includes a chapter on how the instructions are precisely encoded down to the bit. It follows a tree-like structure of instruction types such as data-processing, load/store, branches, etc., as seen in Figure 2. Once you've followed a branch by matching the specified bits, all instructions are of that type. This might go on for one or more recursive levels. This tree models a decision tree, where the internal nodes make decisions and the leaves are the instructions themselves, which are combinations of matches on the bits of its binary representation.

After reading the reference manual for a few weeks, we had a general understanding of the characteristics of the encodings. A vast majority of the instructions encoding contained a 4-bit conditional value that represented the condition that the

```
(* Corresponds to Section A5.1 *)
Definition armv7_decode n :=
  match (xbits n 28 32) with
  | 15 => A7_InvalidI
  | _  => match (xbits n 26 28) with
          | 0 => armv7_decode_data_processing n
          | 1 => match (xbits n 25 26, xbits n 4 5) with
                 | (0, 0) | (0, 1) | (1, 0) => armv7_decode_load_store n
                 | _ => A7_InvalidI
                 end
          | 2 => armv7_decode_branch n
          | _ => armv7_decode_supervisor_call n
          end
  end.
```

Figure 2. The top-level decoding definition.

instruction should be executed on, such as EQUAL, NOT_EQUAL, LESS_THAN, ALWAYS, etc. These conditions are evaluated based on four flags: N, V, C, and Z. If the right flags are set then the condition is true and so the instruction is executed. However, almost all instructions other than the branches used the ALWAYS condition, so we opted to ignore the rest at the moment to focus our efforts at more critical points.

Another thing we learned was how common objects like registers and immediate values were encoded within the instruction bytes. Registers, which can range from r0, r1, . . . , r14 are represented as the 4-bit value 0, 1, . . . , 14, respectively. Immediate values are directly placed in the instruction bytes as 5, 12, or 16-bit values, usually on the lower-end side. Hence, grabbing the registers and immediate values was as simple as using xbits, as seen in Figure 3.

```
Definition armv7_decode_data_processing_immediate (n: N) :=
  match (xbits n 21 25) with
  | 10 => A7_CMP_imm (xbits n 16 20) (xbits n 0 12)
  | _ => A7_Todo
  end.
```

Figure 3. Example of decoding registers and immediates.

Lifting the decoded instructions was straightforward when the instruction did not have any side effects, such as setting or getting flags. Without side effects, lifting an instruction was a simple mapping from the decoded instruction and its values (registers and immediates) to the lifted one.

Most instruction have side-effects, however. This includes setting certain flags within the ASPR register or setting certain registers to certain values in-addition to its primary effect. The solution to the former was verbose. Take the CMP instruction for example: if the value within the register minus the immediate value is zero, then set the Z flag to 1, otherwise set the Z flag to 0. There are 3 other flags that need to be set based on certain conditions as well, but for this example we will only look at the Z flag. We can model this in Picanae IL using If statements and expressions as seen in Figure 4 and Figure 5.

```
Definition handle_compare (v: exp) (e: exp) : stmt :=
  (Seq
    (If (BinOp OP_EQ (BinOp OP_MINUS v e) (Word 0 32)) (set_conf_flag_on 3) Nop)
    (If (BinOp OP_NEQ (BinOp OP_MINUS v e) (Word 0 32)) (set_conf_flag_off 3) Nop)).
```

Figure 4. Part of the handle_compare definition.

```
(* offset referring to the specific flag.
 * offset = 1 -> set V to 1
 * offset = 2 -> set C to 1
 * offset = 3 -> set Z to 1
 * offset = 4 -> set N to 1
 s*)
Definition set_conf_flag_on (offset: N) : stmt :=
  (Move
    (A7_APSR)
    (BinOp OP_OR
      (Var A7_APSR)
      (BinOp OP_LSHIFT
        (Word 1 32)
        (BinOp OP_PLUS (Word 28 32) (Word offset 32))
      )
    )
  ).
```

Figure 5. Setting a particular flag.

With these side effects in mind, we were able to implement a structure that could generate a Picanae equivalent of the decoded ARMv7 instruction, with the IL statement generated fully describing the actions taken by the instruction on the processor, ie registers affected, operations executed, etc. The example provided in Figure 6 demonstrates how the decoded instructions correspond to their lifted counterpart. As you can see, the IL statement reveals all actions that the instruction is performing at time of execution, which is critical when proving correctness of a program's instructions

```
(* offset referring to the specific flag.
 * offset = 1 -> set V to 1
 * offset = 2 -> set C to 1
 * offset = 3 -> set Z to 1
 * offset = 4 -> set N to 1
 s*)
Definition set_conf_flag_on (offset: N) : stmt :=
  (Move
    (A7_APSR)
    (BinOp OP_OR
      (Var A7_APSR)
      (BinOp OP_LSHIFT
        (Word 1 32)
        (BinOp OP_PLUS (Word 28 32) (Word offset 32))
      )
    )
  ).
```

Figure 6. Lifting decoded instructions to the intermediate language.

Team Organization

Corey Wingo

- Project leader
- Decoded and lifted a significant number of ARMv7 instructions
- Presenter
- Contributed and edited report

Saffat Ahmed

- Decoded and lifted a significant number of ARMv7 instructions
- Contributed and edited report

Nolan Kuo

- Decoded and lifted a significant number of ARMv7 instructions
- Contributed and edited report

Jacob Wiedemeier

- Wrote a python script to generate a lookup table in Coq for individual opcodes in an ARMv7 binary
- Edited report and converted to LaTeX

Evaluation

When we set out to complete this project, our goal was to build an ARMv7 instruction decoder/lifter framework that could successfully lift a stringlen binary. Based on this initial goal, it can be said that we were about 90% successful with this project. We were able to fully decode instructions from the binary we picked out and lift all of the instruction except for a few, where their side effects were not lifted fully, namely the ADD_imm and CMP_imm instructions which are present in our target binary. This is because they contain severe side effects such as updating the 4 flags that ARMv7 has. We were not sure how to utilize the Picaane IL system to lift those side effects properly.

To elaborate on the CMP_imm instruction, which is central to branching in ARMv7, it needed to update all 4 flags based on the subtraction of a register value and an immediate value. If it was equal, set the Z flag to 1, otherwise 0. If the operation carried, then set the C flag to 1, otherwise 0. The problem we had was checking for this carry using the Picanae IL expressions as we couldn't use the COQ standard library or write our own because we had to operate on the Var ASPR that the Picanae system alone could only access.

To compensate for the lack of side effects, we extended the framework to include more instructions that could be decoded and lifted into Picanae, rounding the amount of ARMv7 instructions supported by our decoder to roughly 30 instructions. In addition, a

few instructions were added that could be decoded, but not lifted successfully by us due to their similarity to the mentioned CMP_imm instruction and its side effects.

Future Work

The following list contains potential future work on the ARMv7 Decoder and Lifter. It is arranged in order of priority from top to bottom.

1. Verify lifted instructions
2. Add support for Thumb Mode
3. Support all side effects of every instruction
4. Support the entire instruction set

Currently, we have not worked on verifying whether an instruction was lifted appropriately. A good method of verifying a lifted instruction would be running it within the Picanae System and observing its primary and side effects. This could be accomplished through running exhaustive tests on instructions with other existing decoders such as BinaryAnalysisPlatform and Angr and comparing their lifted instruction set to the Picanae lifted instruction to verify the correctness of the framework.

Supporting thumb mode is as simple as duplicating the existing code, except using encoding TX for the instruction within the ARMv7 Reference Manual. The decoding and lifting would need to be matched using different paths as they have different patterns and instruction parameters, such as different bit-width immediate values and registers. This would be tedious, but fairly easy, as one could reference the existing code. One could also utilize a variable related to the decoder state in order to track when the binary transitions in and out of thumb mode in order to better lift all the effects of the instruction.

We currently do not support every side effect for the instructions we lifted. For example, we ignore instruction variants that depend on conditional flags to execute. Another lack of implementation is the CMP instruction, which has an immense number of side-effects and represents an imposing challenge to decode and lift. Future work should focus on supporting these side effects and expanding the Picanae system to account for carry variables, which entails proving the correctness of those operations.

We do not support every instruction in the ARMv7 reference manual. There are about a hundred main instructions, each with many variants. We currently support approximately thirty main instructions, many of which have a similar side effect issue to the aforementioned CMP instruction.

Related Work

There are many ARMv7 decoders out there that we used as references on how to design such a decoder.

Yaxpeax-arm [3] is one such project that we referenced. Implemented in Rust, the project supports ARMv7, ARMv7/thumb, and aarch instruction sets. It can successfully decode and display the ARMv7 instruction from the actual instruction bytes through the usage of masking bytes and instruction logic. The source file also demonstrates the usage of a Thumb state for the decoder to keep track of the decoder state while processing instructions, which supports our expected future work regarding the support of thumb mode. Similar to our own implementation, their work does not have a formal method for proving the correctness of their instruction decoding. Instead, their verification involves exhaustive comparative tests on ARMv7 instructions with multiple proven decoders to verify the correctness of their decoder implementation - a process that will likely be mirrored in the future work of our own architecture.

We also very heavily relied on a previous implementation of a decoder and lifter for the Picanae system for the RISC-V architecture. The Picanae system provides a number of formally verified methods that allow projects to create decoder frameworks without needing to re-verify the correctness of methods, such as binary operations and bit extraction from a decimal number. The aforementioned RISC-V implementation was able to use these helper methods to extract instruction bytes in order to successfully decode and lift the RISC-V instruction set to the Picanae IL, providing a path to formal verification of RISC-V binaries. Since RISC-V has a simpler instruction set compared to ARMv7, we were able to utilize this implementation as the base template for our own implementation, with informal verification of our lifted instructions being compared to those from RISC-V to determine if the instruction behavior functioned

similarly to our own.

Setup Instructions

The framework for both the ARMv7 deocder and the subsequent lifter to the Picanae IL are compiled using Coq, namely (insert Coq version here), and is currently located within a single (insert file name).v file. Given the IL base, it is recommended to run the Coq file from within the Picanae system folder, provided that the project has been compiled and built into a library for the .v file to be utilized. The file includes the selected stringlen binary ARMv7 instruction code, alongside the corresponding decimal values that the decoder reads in order to successfully decode the instruction. In addition to the values from the test binary, the file also contains several computed statements that use decimal values corresponding to all of the instructions supported by our framework, but not present within the selected test binary. Running the file through each instruction should yield a successful run as one steps through the instruction set.

We have also included a Python3 script that is capable of parsing an ARMv7 binary and dumping the decimal representation of each instruction into a Coq .v file. Currently, due to the nature of how the script is meant to interact with binaries, the script currently only works on Linux environments. Our own environment was Ubuntu (insert version here). In order to parse custom binaries, the following steps must be taken.

1. Ensure that the Linux distribution being utilized has the standard Python library installed
2. In terminal, run "apt-get install binutils-arm-linux-gnueabi" to install the library used to parse ARMv7 binaries
3. With both the python script and the binary in the same location, execute "python Picinae_ARM_to_Function.py (name of ARMv7 binary file) in order to generate a Coq .v file to transfer the decimal representations to our (insert coq file name).v file for decoding and lifting.

Citations

[1] - CPUlator ARMv7 System Simulator. (2019, March 9). Retrieved December 13, 2022, from https://cpulator.01xz.net/?sys=arm

[2] - ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition. arm Developer. (2011, November 23). Retrieved December 13, 2022, from https://developer.arm.com/documentation/ddi0406/c

[3] - Iximeow. (2022, September 29). Arm decoders for the Yaxpeax Project. GitHub. Retrieved December 13, 2022, from https://github.com/iximeow/yaxpeax-arm