

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), Picanæ, 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 Picanæ systems that target similar architectures like RISC-V, there is no ARM focused decoder that lifts instructions from a binary up to Picanæ 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 B.

00000000	e3a00000	3	mov	r0, #0 ; 0x0
00000004	e59f1018	4	ldr	r1, =string
			ldr	r1, [pc, #24] ; 0x24
00000008	ebffffff	5	bl	strlen
			bl	0xc (0xc: strlen)
		7	strlen:	

Figure B. 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 compatibility 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 Z. 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.

```
(* 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 Z. The top-level decoding definition.

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 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 A.

```
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 A. 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 Picae IL using If statements and expressions as seen in Figure X and Figure Y.

```
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 X. 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 Y. Setting a particular flag.

Team Organization

Corey Wingo

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

Saffat Ahmed

- Decoded and lifted a significant number of ARMv7 instructions

Nolan Kuo

- Decoded and lifted a significant number of ARMv7 instructions

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

Work and answer goes here

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.

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 you can reference the existing code.

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.

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.

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.

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>