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, 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 basically all smartphone processors. In addition, with Apple adapting 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. It can assemble ARMv7 instructions into their binary representations and even emulate them, as seen in Figure B.

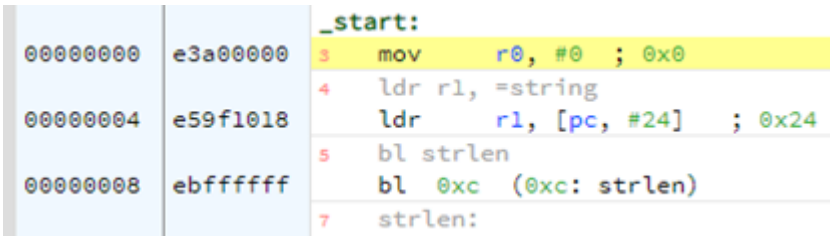

Figure B. Assembly of ARMv7 instructions.

Using the cpulator tool was very tedious, however. It required us to manually copy and convert the hexadecimal bytes to decimal and then to paste it into the file. Hence, 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 because the Picanae system did not support it. However, this does not change any of the operations we can perform on the representation. We can still use critical methods such as xbits and shift.

Once we had 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. 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 characteris-

tics of the encodings. A huge 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 if so, then 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 were 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 pretty straightforward when the instruction did not have any side effects such as setting/getting flags. In which case, it 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 foremost 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 to 0. There are 3 other flags that need to be set based on certain conditions as well, but we'll just look at the Z flag. We can model this in Picanae 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
- Converted report to LaTeX form

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 no way of verifying whether an instruction was lifted appropriately, rather, we are just "eyeballing" it. A good method of verifying a lifted instruction is running it within the Picanae System and observing it's 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 support. For example, we ignore instruction variants that depend on conditional flags to execute. Another lack of implementation is the CMP instruction. It has an immense number of side-effects and we are unsure of how to implement it properly.

We also do not support every instruction in the ARMv7 reference manual. There are about a hundred, each with many variants. We only support approximately 30.

Related Work

There are many ARMv7 decoders out there that we looked at for reference on how to design ours. We also very heavily relied on a previous implementation of a decoder and lifter for the Picanae system for the RISC-V architecture.