A verifying ARM compiler

Thomas Tuerk

ARG Lunch, November 7th, 2006

Overview

The CryptVer Project

Main Goal

create verified ARM implementations of elliptic curve cryptographic algorithms

Important Parts

- a formalisation of elliptic curve operations (Joe Hurd)
- a compiler from first order tail-recursive functions to ARM assembler (Konrad Slind)
- a very high fidelity model of the ARM instruction set (Anthony Fox)
  - a Hoare Logic for this ARM model (Magnus Myreen)

Placement of this work

Gap

- the compiler uses an own formalisation of ARM assembler
- this formalisation is simplified
  - no processor modes
  - simplified instruction set
  - separation between program and data memory
  - simplified PC
  - ...

Goal

Use the compiler to create programs in the Fox's ARM model that are specified by Myreen's Hoare Logic!

ARM Compiler

- developed by Konrad Slind, Guodong Li and Scott Owens at the University of Utah
- input language is formed by first order tail-recursive equations
- argument and result have to be vectors of 32 bit words
- output is a ARM assembler program and a proof that this program implements the original equation

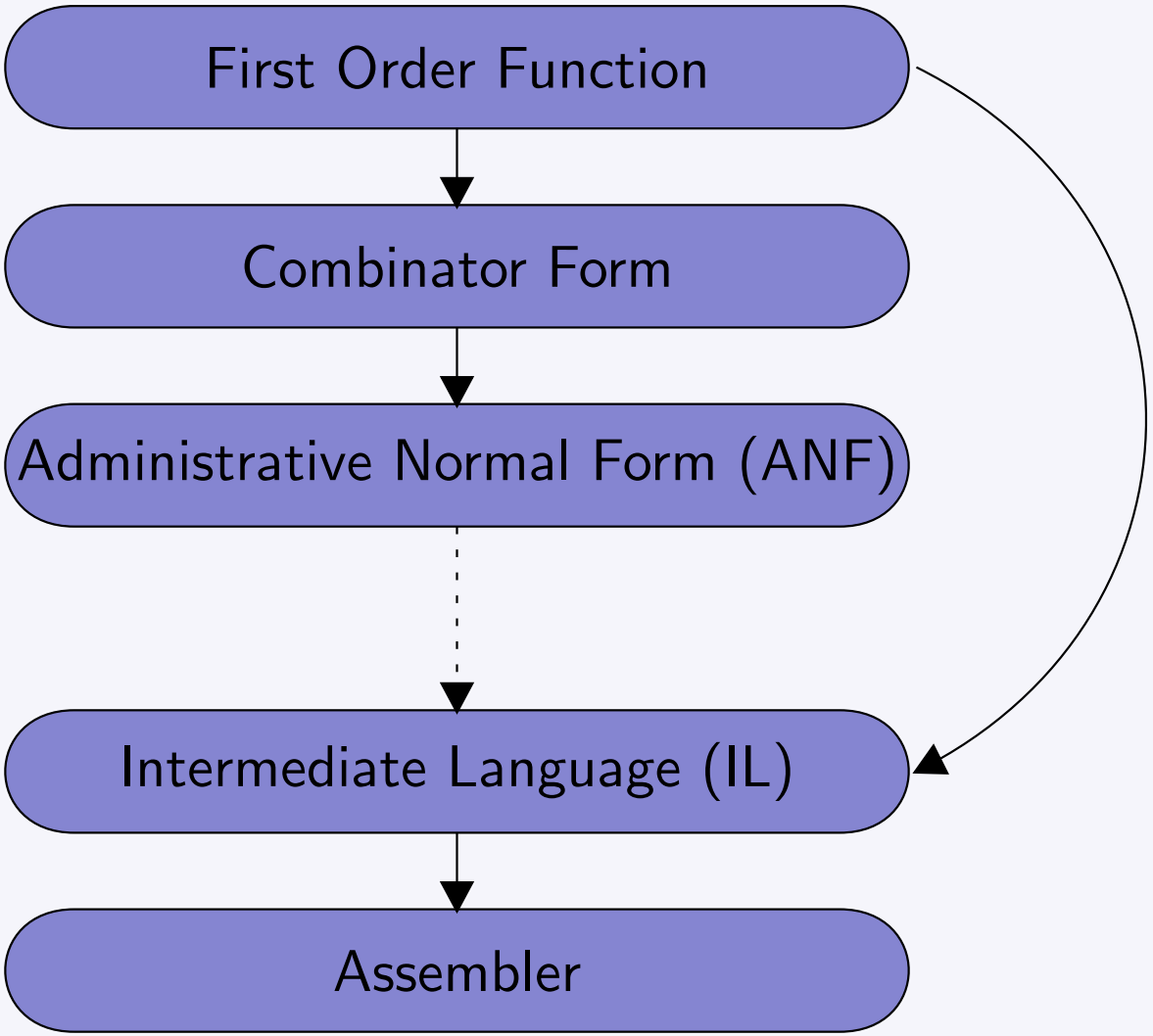
ARM Compiler - Example

Example

```
val ShiftXor_def = Define
  `ShiftXor (x:word32,s,k0,k1) =
      ((x << 4) + k0) ?? (x + s) ?? ((x >> 5) + k1)`;

- val spec = #5 (pp_compile ShiftXor_def);
|- !st. get_st (run_arm
  [((LSL,NONE,F),SOME (REG 4),[REG 0; WCONST 4w],NONE);
   ((ADD,NONE,F),SOME (REG 2),[REG 4; REG 2],NONE);
   ((ADD,NONE,F),SOME (REG 1),[REG 0; REG 1],NONE);
   ((EOR,NONE,F),SOME (REG 1),[REG 2; REG 1],NONE);
   ((ASR,NONE,F),SOME (REG 0),[REG 0; WCONST 5w],NONE);
   ((ADD,NONE,F),SOME (REG 0),[REG 0; REG 3],NONE);
   ((EOR,NONE,F),SOME (REG 0),[REG 1; REG 0],NONE)]
  ((0,0w,st),))<MR R0> =
  ShiftXor (st<MR R0>,st<MR R1>,st<MR R2>,st<MR R3>)
```

ARM Compiler - Steps



ARM Compiler - Steps II

First Order Function

```
ShiftXor (x:word32,s,k0,k1) =
  ((x << 4) + k0) ?? (x + s) ?? ((x >> 5) + k1)
```

Combinator Form

```
|- ShiftXor =
   Seq (Par (Seq (Par (Seq (Par (\(x,s,k0,k1). x)
   (\(x,s,k0,k1). 4)) (UNCURRY $<<)) (\(x,s,k0,k1). k0))
   (UNCURRY $+)) (Seq (Par (Seq (Par (\(x,s,k0,k1). x)
   (\(x,s,k0,k1). s)) (UNCURRY $+))
   (Seq (Par (Seq (Par (\(x,s,k0,k1). x) (\(x,s,k0,k1). 5))
   (UNCURRY $>>)) (\(x,s,k0,k1). k1)) (UNCURRY $+)))
   (UNCURRY $??))) (UNCURRY $??)))
```

ARM Compiler - Steps III

ANF

```
!v1 v2 v3 v4.
   ShiftXor (v1,v2,v3,v4) =
     (let v5  = UNCURRY $<< (v1,4) in
      let v6  = UNCURRY $+ (v5,v3) in
      let v7  = UNCURRY $+ (v1,v2) in
      let v8  = UNCURRY $>> (v1,5) in
      let v9  = UNCURRY $+ (v8,v4) in
      let v10 = UNCURRY $?? (v7,v9) in
      let v11 = UNCURRY $?? (v6,v10) in
      v11)
```

ARM Compiler - Steps IV

IL

```
!st. run_ir (BLK
    [MLSL R4 R0 4w; MADD R2 R4 (MR R2); MADD R1 R0 (MR R1);
     MASR R0 R0 5w; MADD R0 R0 (MR R3); MEOR R0 R1 (MR R0);
     MEOR R0 R2 (MR R0)]) st<RR R0> =
ShiftXor (st<RR R0>,st<RR R1>,st<RR R2>,st<RR R3>)
```

ARM Compiler - Steps V

Assembler

```
|- !st. get_st (run_arm
  [((LSL,NONE,F),SOME (REG 4),[REG 0; WCONST 4w],NONE);
   ((ADD,NONE,F),SOME (REG 2),[REG 4; REG 2],NONE);
   ((ADD,NONE,F),SOME (REG 1),[REG 0; REG 1],NONE);
   ((EOR,NONE,F),SOME (REG 1),[REG 2; REG 1],NONE);
   ((ASR,NONE,F),SOME (REG 0),[REG 0; WCONST 5w],NONE);
   ((ADD,NONE,F),SOME (REG 0),[REG 0; REG 3],NONE);
   ((EOR,NONE,F),SOME (REG 0),[REG 1; REG 0],NONE)]
  ((0,0w,st),))<MR R0> =
  ShiftXor (st<MR R0>,st<MR R1>,st<MR R2>,st<MR R3>)
```

First Approach

Naive Approach

- the compiler output and Fox's ARM assembler look similar
- translate the compiler output to Fox's ARM assembler
- do not modify the compiler

Problems

- technical problems
- compiler's assembler is to general
- semantics of compiler's assembler is different to ARM assembler

$\Longrightarrow$ it was useful to modify the compiler

Addressing of the technical problems

- porting the compiler to the new word library
- translation of IL to ARM assembler
  - IL is simpler than the compiler's assembler
  - IL contains no hard coded jump addresses
  - thus one instruction can be easily split into several

Changing IL to be closer to ARM assembler

- allow memory access only with load and store operations
- a data operation may at most work on one constant
  - e. g. `ADD r0, #1, #2` is not allowed
- change constants from arbitrary 32 bit words to 8 bit words and 4 bit shifts
  - e.g. the 32 bit word $0x00010000$ is represented by $(0x01, 0x8)$
  - e.g. the constant $0x00010001$ can not be represented directly

Changing IL to be closer to ARM assembler II

- fix a bug in addressing memory
  - memory is addressed using a base register and an offset
  - the upper 30 bit of the base register are used for addressing
  - thus, to go to the next 32 bit address 4 has to be added to the base register
  - implementation added just 1 and used the whole 32 bit for addressing
- address memory with 30 bit words instead with natural numbers
- fix bug with comparison operators and flags

Other changes

- a preprocessing step was added to get rid of ugly constants
  - e.g. `example_const x = (x + (0xFF0012Fw - 0x1003w))` is rewritten to `example_const x = x + (0x12Cw !! 0xFF000w !! 0xFE00000w)`
- the translation to ANF has been adapted to the new IL
- the simulation step of IL programs had to be adapted

Translation to ARM assembler

- there is now a close correspondence between ARM assembler and IL
- the translation of IL to ARM assembler is straight forward
- the correctness proof of this translation for programs not involving memory is laborious, but easy
- however, the compiler had to be modified to export internal informations
  - termination proofs of loops
  - theorem about unchanged registers
  - information about input and output variables
- Myreen's Hoare Logic can be used easily

Example

Example

```
val ShiftXor_def = Define
  `ShiftXor (x:word32,s,k0,k1) =
      ((x << 4) + k0) ?? (x + s) ?? ((x >> 5) + k1)`;

- val spec = #6 (pp_compile ShiftXor_def);

val spec =
  |- !st. (run_ir (BLK [MLSL R4 R0 4w;
                        MADD R2 R4 (MR R2);
                        MADD R1 R0 (MR R1);
                        MASR R0 R0 5w;
                        MADD R0 R0 (MR R3);
                        MEOR R0 R1 (MR R0);
                        MEOR R0 R2 (MR R0)]) st)<RR R0> =
     ShiftXor (st<RR R0>,st<RR R1>,st<RR R2>,st<RR R3>)
```

Example II

```
|- !state_old.
(!e. e < 7 ==>
 (state_old.memory
   (ADDR30 (FETCH_PC state_old.registers + n2w (4 * e))) =
    enc (EL e
      [MOV AL F 4w (Dp_shift_immediate (LSL 0w) 4w);
       ...
       EOR AL F 0w 2w (Dp_shift_immediate (LSL 0w) 0w)]))) /\
~state_old.undefined ==>
?step_num.
  (REG_READ (STATE_ARMe step_num state_old).registers
    (DECODE_MODE ((4 >< 0) (CPSR_READ (STATE_ARMe step_num state_old).psrs))) (MREG2REG R0) =
ShiftXor
 (REG_READ state_old.registers (DECODE_MODE ((4 >< 0) (CPSR_READ state_old.psrs))) (MREG2REG R0),
  REG_READ state_old.registers (DECODE_MODE ((4 >< 0) (CPSR_READ state_old.psrs))) (MREG2REG R1),
  REG_READ state_old.registers (DECODE_MODE ((4 >< 0) (CPSR_READ state_old.psrs))) (MREG2REG R2),
  REG_READ state_old.registers (DECODE_MODE ((4 >< 0) (CPSR_READ state_old.psrs))) (MREG2REG R3)) /\
  ((STATE_ARMe step_num state_old).memory = state_old.memory) /\
  ~(STATE_ARMe step_num state_old).undefined /\
   (?N Z C V.(STATE_ARMe step_num state_old).psrs = CPSR_WRITE state_old.psrs
              (SET_NZCV (N,Z,C,V) (CPSR_READ state_old.psrs))) /\
  (owrt_visible_regs (STATE_ARMe step_num state_old) =  owrt_visible_regs state_old) /\
  (FETCH_PC (STATE_ARMe step_num state_old).registers = FETCH_PC state_old.registers + 28w)
```

Example III

```
|- !rv0 rv1 rv2 rv3.
ARM_PROG
  (R 0w rv0 * R 1w rv1 * R 2w rv2 * R 3w rv3 *
   ~R 4w * ~R 11w * ~R 12w * ~R 13w * ~S)
  (MAP enc
     [MOV AL F 4w (Dp_shift_immediate (LSL 0w) 4w);
      ADD AL F 2w 4w (Dp_shift_immediate (LSL 2w) 0w);
      ADD AL F 1w 0w (Dp_shift_immediate (LSL 1w) 0w);
      MOV AL F 0w (Dp_shift_immediate (ASR 0w) 5w);
      ADD AL F 0w 0w (Dp_shift_immediate (LSL 3w) 0w);
      EOR AL F 0w 1w (Dp_shift_immediate (LSL 0w) 0w);
      EOR AL F 0w 2w (Dp_shift_immediate (LSL 0w) 0w)])
  (R 0w (ShiftXor (rv0,rv1,rv2,rv3)) * R 3w rv3 *
   ~R 1w * ~R 2w * ~R 4w * ~R 11w * ~R 12w * ~R 13w * ~S)
```

Unsolved Problems

- the current version of the compiler uses memory just for function calls
- however, function calls are buggy
- therefore, the translation to ARM assembler does not consider memory yet
- the compiler often fails

Future Work

- Guodong Li is currently working on a new version of the compiler
- this version will improve the translation of ANF to IL
- function calls will be fixed
- the translation to ARM assembler should then be extended to handle memory
- compile some functions used in elliptic curve cryptography