1 Example of Translation

This document serves two purposes: to show a side by side example of ACL2 code and its Sail translation; and to give a brief introduction to the ACL2 model itself. The functions demonstrate a simple call trace of the instruction `add $0x1, %eax`, which increments the register `eax`. A diagrammatic summary can be found in Figure 2.1 of the dissertation.

Note that extraneous keyword parameters (e.g. from `define`) and comments are elided to save space. The indentation in the Sail column is mine and some brackets have been removed manually—hopefully this has a similar effect to pretty printing.

1.1 Navigating the x86 ISA model

This is a quick example of one way to navigate the ACL2 model without grepping through code files (which is often unhelpful in Lisp given the presence of code generation). First, load the X86ISA package as follows (from an ACL2 interpreter):

```
!> (include-book "projects/x86isa/tools/execution/top" :ttags :all :dir :system)

!> (in-package "X86ISA")
```

Let's say we want to find out more about the function `rme08-opt` (encountered below). One way would be to run something like `rg rme08-opt` from a terminal and find this is a macro defined in `machine/top-level-memory.lisp`. Once we open that file, look at its contents and decide to explore more about one of the functions it calls, `rme08`, we become stuck: `rg rme08` only shows calls to it, but no definition!

To solve this we can run `:pe rme08` from the ACL2 shell. This immediately gives us the file in which it is defined (we can narrow down where the code generation occurs) and its actual definition.

Running `:pe rme08-opt` does not seem as helpful at first: it gives us a huge mess of nested conses of quoted code—it is a macro. If, however, we have a use of the macro, we can translate it with actual arguments. For instance, using an example from below: `:trans1 (rme08-opt proc-mode temp-rip #.*cs* :x x86)` gives us some very reasonable code:

```
(MBE :LOGIC (RME08 PROC-MODE TEMP-RIP 1 :X X86)
  :EXEC (IF (EQL PROC-MODE 0)
            (B* (((MV FLG (THE (SIGNED-BYTE 48) LIN-ADDR))
                  (MV NIL TEMP-RIP))
                 ((WHEN FLG) (MV FLG 0 X86)))
                (RML08 LIN-ADDR :X X86))
            (RME08 PROC-MODE TEMP-RIP 1 :X X86)))
```

Running `:trans` (as opposed to `:trans1`) is often less helpful as it translates macros recursively instead of just the first level.

1.2 Step Function

Lisp

```
(define x86-fetch-decode-execute (x86)

  (b* ((ctx 'x86-fetch-decode-execute)
       ((when (or (ms x86) (fault x86))) x86)

       (proc-mode (x86-operation-mode x86))
       (64-bit-modep (equal proc-mode #.*64-bit-mode*))

       (start-rip (the (signed-byte #.*max-linear-address-size*)
                       (read-*ip proc-mode x86)))
```

Sail

```
val x86_fetch_decode_execute : (int) -> int effect {wreg, undef, rmem,
    ↪ escape, rreg, wmv, eamem}
function x86_fetch_decode_execute (x86) =
  let ctx = "x86-fetch-decode-execute" : string in
    if ms(0)) | (fault(0)
    then 0
    else
      let proc_mode = (x86_operation_mode(0)) : int in
      let n64_bit_modep_var = (proc_mode == 0) : bool in
      let start_rip =(the_range(-140737488355328, 140737488355327, read_iptr
          ↪ (proc_mode, 0))) : int in
```

This is the beginning of the top-level step function which simulates one processor cycle.

Translation Notes

- The Lisp `b*` binder simulates control flow, binding many variables one after the other and occasionally escaping when there is an error (`when`). This is translated as a series of nested `let` expressions with an occasional `if` when escaping.
- In `let <var> = <expr> in <body>` the `<expr>` always has a type annotation. This is to avoid some type checking errors—see `FutureWork.md` for more information.
- We see an example of `the_range` in Sail: a dynamic type check. The large numbers are the result of macro-expanding `#.*max-linear-address-size*`. See `FutureWork.md` for more information.

Operation

- Overall, this snippet first binds the current `ctx`, occasionally used for debugging, before checking if there is a model state (`ms`) error or `fault`. If this is OK, it finds the processor mode, and if 64-bit mode is active before retrieving the current instruction pointer.

Lisp

```
      ((mv flg (the (unsigned-byte #.*prefixes-width*) prefixes)
           (the (unsigned-byte 8) rex-byte)
           x86)
       (get-prefixes proc-mode start-rip 0 0 15 x86))
      ((when flg)
        (!!ms-fresh :error-in-reading-prefixes flg))
```

Sail

```
    let (flg, prefixes, rex_byte, x86) = (get_prefixes(proc_mode,
        ↪ start_rip, 0, 0, 15, 0)) : (option(string), int, int, int) in
      if is_some(flg)
      then throw(Emsg("Model␣state␣error:␣:ERROR-IN-READING-PREFIXES"))
          ↪ else
    let opcode_vex_evex_byte = (prefixes_get_nxt(prefixes)) : int in
    let prefix_length = (prefixes_get_num(prefixes)) : int in
```

```
  ((the (unsigned-byte 8) opcode/vex/evex-byte)
   (prefixes->nxt prefixes))

  ((the (unsigned-byte 4) prefix-length)
   (prefixes->num prefixes))

  ((mv flg temp-rip) (add-to-*ip proc-mode start-rip (1+ prefix-length)
      ↪ x86))
  ((when flg) (!!ms-fresh :increment-error flg))

  ;; VEX/EVEX decoding and dispatch elided

  (opcode-byte opcode/evex-byte)

  (modr/m? (one-byte-opcode-ModR/M-p proc-mode opcode-byte))
  ((mv flg (the (unsigned-byte 8) modr/m) x86)
   (if modr/m?
       (if (or vex-byte0? evex-byte0?)
           (mv nil les/lds-distinguishing-byte x86)
         (rme08-opt proc-mode temp-rip #.*cs* :x x86))
     (mv nil 0 x86)))
  ((when flg)
   (!!ms-fresh :modr/m-byte-read-error flg))

  ((mv flg temp-rip)
   (if modr/m?
       (add-to-*ip proc-mode temp-rip 1 x86)
     (mv nil temp-rip)))
  ((when flg) (!!ms-fresh :increment-error flg))

  ;; Finding of SIB byte elided

  ((mv flg temp-rip)
   (if sib?
       (add-to-*ip proc-mode temp-rip 1 x86)
     (mv nil temp-rip)))
  ((when flg) (!!ms-fresh :increment-error flg)))

(one-byte-opcode-execute
 proc-mode start-rip temp-rip prefixes rex-byte opcode-byte
 modr/m sib x86))
```

```
let (flg, temp_rip) = (add_to_iptr(proc_mode, start_rip, 1 +
    ↪ prefix_length, 0)) : (option(string), int) in
  if is_some(flg)
  then throw(Emsg("Model␣state␣error:␣:INCREMENT-ERROR")) else

// VEX/EVEX decoding and dispatch elided

let opcode_byte = (opcode_evex_byte) : int in
let modr_m? = (one_byte_opcode_modr_m_p(proc_mode, opcode_byte)) :
    ↪ bool in
let (flg, modr_m, x86) =
  (if modr_m?
  then
    if vex_byte0? | evex_byte0?
    then (None() : option(string), les_lds_distinguishing_byte : int, 0
        ↪  : int)
    else rme08(proc_mode, temp_rip, 1, ":X", 0)
  else (None() : option(string), 0 : int, 0 : int)) : (option(string),
      ↪ int, int) in
  if is_some(flg)
  then throw(Emsg("Model␣state␣error:␣:MODR/M-BYTE-READ-ERROR")) else
let (flg, temp_rip) =
  (if modr_m?
  then add_to_iptr(proc_mode, temp_rip, 1, 0)
  else (None() : option(string), temp_rip)) : (option(string), int) in
  if is_some(flg)
  then throw(Emsg("Model␣state␣error:␣:INCREMENT-ERROR")) else

// Finding of SIB byte elided

let (flg, temp_rip) =
  (if sib?
  then add_to_iptr(proc_mode, temp_rip, 1, 0)
  else (None() : option(string), temp_rip)) : (option(string), int) in
  if is_some(flg)
  then throw(Emsg("Model␣state␣error:␣:INCREMENT-ERROR"))
else one_byte_opcode_execute(proc_mode, start_rip, temp_rip, prefixes,
    ↪  rex_byte, opcode_byte, modr_m, sib, 0)
```

This is the rest of the top level step function.

Translation Notes

- This is an example of the result when compiling with the `translate_the` flag in `config_files.py` set to `False`. We no longer have the `the` dynamic type check. This improves readability but risks errors in handwritten support functions going undetected for longer.
- The control flow pattern of escaping when there is an error is seen multiple times (`(when flg)` in ACL2 and `if is_some(flg)` in Sail). In ACL2 `flg` is either `nil` or a descriptive string—we translate as an `option` type in Sail.

Operation

- x86 instructions may have a number of prefixes, each one indicated by a certain bitpattern. These prefixes are gathered first in `get_prefix` (a complex function in itself) before we extract the number of prefixes and the next byte following them: the opcode. Information about prefixes is encoded in an integer in ACL2 and translated as such in Sail—it would be better to store this information in a struct.
- VEX/EVEX (SIMD ISA extensions) decoding and dispatch is performed here (but not shown). Assuming the instruction is not VEX/EVEX then the `modR/M` byte is decoded if present. This byte changes how certain instructions regard certain registers, specifically modifying their size. `rme08-opt` is used to read single bytes from effective memory (hence the 'e' in `rme`) and represents a stack of complex functions as address translation from effective to linear address is required.
- A `SIB` byte (which specifies how to calculate operand addresses) is also found if it exists for this instruction. Not shown here.
- Finally, control is passed to `one_byte_opcode_execute`.

1.3 Opcode Dispatch

Lisp

```
(define
 one-byte-opcode-execute
 ((proc-mode :type (integer 0 4))
  (start-rip :type (signed-byte 48))
  (temp-rip :type (signed-byte 48))
  (prefixes :type (unsigned-byte 52))
  (rex-byte :type (unsigned-byte 8))
  (opcode :type (unsigned-byte 8))
  (modr/m :type (unsigned-byte 8))
  (sib :type (unsigned-byte 8))
  x86)

(case opcode
  ((0
  (LET
```

Sail

```
val one_byte_opcode_execute : (int, int, int, int, int, int, int, int, int)
    ↪ -> int effect {rreg, undef, wmv, rmem, eamem, escape, wreg}
function one_byte_opcode_execute (proc_mode, start_rip, temp_rip, prefixes,
    ↪ rex_byte, opcode, modr_m, sib, x86) =
(match opcode {
  0 =>
    let fault_var =
      (if (modr_m_get_mod(modr_m) == 3) & (240 == prefixes_get_lck(prefixes)
          ↪ )
      then Some(":UD")
      else None() : option(string)) : option(string) in
      if is_some(fault_var)
      then
        (match fault_var {
          Some(":UD") => throw(Emsg("A␣fault␣occurred.␣␣Original␣ACL2␣AST:␣['
```

```
   ((FAULT-VAR (IF (OR (UD-LOCK-USED-DEST-NOT-MEMORY-OP))
                   ':UD
                   NIL)))
   (IF FAULT-VAR
       (CASE FAULT-VAR
            (:UD (X86-ILLEGAL-INSTRUCTION "#UD Encountered!"
                                          START-RIP TEMP-RIP X86))
            (T (X86-STEP-UNIMPLEMENTED "Unimplemented exception in x86isa!"
                                       X86)))
       (X86-ADD/ADC/SUB/SBB/OR/AND/XOR/CMP/TEST-E-G
           0 PROC-MODE START-RIP TEMP-RIP PREFIXES
           REX-BYTE OPCODE MODR/M SIB X86))))
(1 ... )

;; Opcodes 1-14 elided

(15
 (TWO-BYTE-OPCODE-DECODE-AND-EXECUTE PROC-MODE START-RIP
                                     TEMP-RIP PREFIXES REX-BYTE OPCODE X86))

 ;; Remaining opcodes elided

 )
))
```

```
                ↪ X86-ILLEGAL-INSTRUCTION', STRING: UD Encountered!, 'START-
                ↪ RIP', 'TEMP-RIP', 'X86']")),
              _ => throw(Emsg("Model state error: Unimplemented exception in
                ↪ x86isa!"))
            }) : int
          else x86_add_adc_sub_sbb_or_and_xor_cmp_test_e_g(0, proc_mode,
              ↪ start_rip, temp_rip, prefixes, rex_byte, opcode, modr_m, sib,
              ↪ 0),
    1 =>

    // Opcodes 1-14 elided

    15 => two_byte_opcode_decode_and_execute(proc_mode, start_rip, temp_rip,
        ↪ prefixes, rex_byte, opcode, 0),

    // Remaining opcodes elided

    }
  )
```

Translation Notes
Here we see the use of more macro expansions which decrease readability. For instance, whereas in ACL2 we have `UD-LOCK-USED-DEST-NOT-MEMORY-OP`, which at least gives us a hint about its meaning, this expands and translates in Sail to `(modr_m_get_mod(modr_m) == 3) & (240 == prefixes_get_lck(prefixes))`.
Operation
The Lisp on the left hand side is actually the result of a large macro expansion. Here, two cases are shown which dispatch opcodes `0` and `15`. The `0` branch is the one we follow, dispatching to the operation specification function `X86-ADD/ADC/SUB/SBB/OR/AND/XOR/CMP/TEST-E-G`. The `15` branch shows how control escapes to the two-byte opcode dispatch table (which can then escape a second time to three-byte opcodes if needed).

1.4 Operation Specification Function

Lisp	Sail
`def-inst x86-add/adc/sub/sbb/or/and/xor/cmp/test-E-G` `  (b* ((G (rgfi-size operand-size`	`val x86_add_adc_sub_sbb_or_and_xor_cmp_test_e_g : (int, int, int, int, int,` `  ↪ int, int, int, int, int) -> int effect {rreg, undef, wmv, wreg, rmem,`

```
      (the (unsigned-byte 4)
        (reg-index reg rex-byte #.*r*))
      rex-byte x86))
    ((mv flg0
   E
   (the (unsigned-byte 3) increment-RIP-by)
   (the (signed-byte 64) E-addr)
   x86)
(x86-operand-from-modr/m-and-sib-bytes proc-mode #.*gpr-access*
        operand-size
        inst-ac?
        nil ;; Not a memory pointer operand
        seg-reg
        p4?
        temp-rip
        rex-byte
        r/m
        mod
        sib
        0 ;; No immediate operand
        x86))
    ((when flg0)
(!!ms-fresh :x86-operand-from-modr/m-and-sib-bytes flg0))

;; The following binders are elided: ctx, r/m, mod, reg, p2, p4?,
;; byte_operand?, operand_size, seg_reg, inst_ac?, temp_rip,
;; badlength?
```

```
    ↪  eamem, escape}
function x86_add_adc_sub_sbb_or_and_xor_cmp_test_e_g (operation, proc_mode,
    ↪ start_rip, temp_rip, prefixes, rex_byte, opcode, modr_m, sib, x86) =
  let g = (rgfi_size(operand_size, reg_index(reg, rex_byte, 2), rex_byte, 0)
      ↪ ) : int in
  let (flg0, e, increment_rip_by, e_addr, x86) = (
      ↪ x86_operand_from_modr_m_and_sib_bytes(proc_mode, 0, operand_size,
      ↪ inst_ac?, false, seg_reg, p4?, temp_rip, rex_byte, r_m, mod_var,
      ↪ sib, 0, 0)) : (option(string), int, int, int, int) in
    if is_some(flg0)
    then throw(Emsg("Model␣state␣error:␣:X86-OPERAND-FROM-MODR/M-AND-SIB-
        ↪ BYTES")) else

  // The following binders are elided: ctx, r/m, mod, reg, p2, p4?,
  // byte_operand?, operand_size, seg_reg, inst_ac?, temp_rip,
  // badlength?
```

```
    ((the (unsigned-byte 32) input-rflags) (rflags x86))
    ((mv result
   (the (unsigned-byte 32) output-rflags)
   (the (unsigned-byte 32) undefined-flags))
(gpr-arith/logic-spec operand-size operation E G input-rflags))

    ;; Updating the x86 state with the result and eflags.
    ((mv flg1 x86)
(if (or (eql operation #.*OP-CMP*)
  (eql operation #.*OP-TEST*))
   ;; CMP and TEST modify just the flags.
   (mv nil x86)
```

```
  let input_rflags = (r_rflags(0)) : int in
  let (result, output_rflags, undefined_flags) =
    ((match operand_size {
      1 => gpr_arith_logic_spec_1(operation, e, g, input_rflags),
      2 => gpr_arith_logic_spec_2(operation, e, g, input_rflags),
      4 => gpr_arith_logic_spec_4(operation, e, g, input_rflags),
      _ => gpr_arith_logic_spec_8(operation, e, g, input_rflags)
    }) : (int, int, int)) : (int, int, int) in
  let (flg1, x86) =
    (if operation == 8 | operation == 7
    then (None() : option(string), 0 : int)
    else x86_operand_to_reg_mem(proc_mode, operand_size, inst_ac?, false,
```

```
    (x86-operand-to-reg/mem proc-mode operand-size
          inst-ac?
          nil ;; Not a memory pointer operand
          result
          seg-reg
          (the (signed-byte 64) E-addr)
          rex-byte
          r/m
          mod
          x86)))
      ((when flg1)
  (!!ms-fresh :x86-operand-to-reg/mem flg1))

      (x86 (write-user-rflags output-rflags undefined-flags x86))
      (x86 (write-*ip proc-mode temp-rip x86)))

    x86)
```

```
        ↪ result, seg_reg, e_addr, rex_byte, r_m, mod_var, 0)) : (option(
        ↪ string), int) in
    if is_some(flg1)
    then throw(Emsg("Model␣state␣error:␣:X86-OPERAND-TO-REG/MEM")) else
  let x86 = (write_user_rflags(output_rflags, undefined_flags, 0)) : int in
  let x86 = (write_iptr(proc_mode, temp_rip, 0)) : int in
  0
```

Translation Notes and Operation

- The upper section performs more decoding—most of the binders are not shown because they follow the patterns shown above of nested `let` expressions. Two important binders, `G` and `E` are shown: these are the operands to the instruction. They are called as such because `G` and `E` represent different addressing modes. There are other variants of this function with different addressing modes (e.g. `E-G`, or `E-I`), but their content is very similar.
- The lower part of the function performs the computation (seen in the next section). Note the macro `gpr-arith/logic-spec` has expanded to a Sail `match` expression which dispatches to a function of the appropriate data-width. Once the results have been computed, it saves them back to the model state.
- Note also the differing return values. In ACL2 we return the updated model state `x86`. In Sail state is global so this is replaced with a dummy return integer, `0`.

1.5 (Dispatch to) Instruction Specification Function

Lisp

```
(define gpr-arith/logic-spec-4
  ((operation :type (member 0 2 4 6 8 1 3 5 7))
   (dst :type (unsigned-byte 32))
   (src :type (unsigned-byte 32))
   (input-rflags :type (unsigned-byte 32)))

  (case operation
```

Sail

```
val gpr_arith_logic_spec_4 : (int, int, int, int) -> (int, int, int) effect
    ↪ {escape}
function gpr_arith_logic_spec_4 (operation, dst, src, input_rflags) =
  (match operation {
    0 => gpr_add_spec_4(dst, src, input_rflags),
    1 => gpr_or_spec_4(dst, src, input_rflags),
    2 => gpr_adc_spec_4(dst, src, input_rflags),
```

```
      (0 (gpr-add-spec-4 dst src input-rflags))
      (1 (gpr-or-spec-4 dst src input-rflags))
      (2 (gpr-adc-spec-4 dst src input-rflags))
      (3 (gpr-and-spec-4 dst src input-rflags))
      (4 (gpr-sub-spec-4 dst src input-rflags))
      (5 (gpr-xor-spec-4 dst src input-rflags))
      (6 (gpr-sbb-spec-4 dst src input-rflags))
      (7 (gpr-and-spec-4 dst src input-rflags))
      (8 (gpr-sub-spec-4 dst src input-rflags))
      (otherwise (mv 0 0 0))))
```

```
    3 => gpr_and_spec_4(dst, src, input_rflags),
    4 => gpr_sub_spec_4(dst, src, input_rflags),
    5 => gpr_xor_spec_4(dst, src, input_rflags),
    6 => gpr_sbb_spec_4(dst, src, input_rflags),
    7 => gpr_and_spec_4(dst, src, input_rflags),
    8 => gpr_sub_spec_4(dst, src, input_rflags),
    _ => (0 : int, 0 : int, 0 : int)
  }) : (int, int, int)
```

```
(define
  gpr-add-spec-4
  ((dst :type (unsigned-byte 32))
   (src :type (unsigned-byte 32))
   (input-rflags :type (unsigned-byte 32)))

  (b*
   ((dst (mbe :logic (n-size 32 dst) :exec dst))
    (src (mbe :logic (n-size 32 src) :exec src))
    (input-rflags (mbe :logic (n32 input-rflags)
                       :exec input-rflags))
    (raw-result (the (unsigned-byte 33)
                  (+ (the (unsigned-byte 32) dst)
                     (the (unsigned-byte 32) src))))
    (signed-raw-result
         (the (signed-byte 33)
              (+ (the (signed-byte 32) (n32-to-i32 dst))
                 (the (signed-byte 32)
                      (n32-to-i32 src)))))
    (result (the (unsigned-byte 32)
                 (n-size 32 raw-result)))
    (cf (the (unsigned-byte 1)
             (cf-spec32 raw-result)))
    (pf (the (unsigned-byte 1)
             (pf-spec32 result)))
    (af (the (unsigned-byte 1)
             (add-af-spec32 dst src)))
    (zf (the (unsigned-byte 1)
             (zf-spec result)))
```

```
val gpr_add_spec_4 : (int, int, int) -> (int, int, int) effect {escape}
function gpr_add_spec_4 (dst, src, input_rflags) =
  let dst = (n_size(32, dst)) : int in
  let src = (n_size(32, src)) : int in
  let input_rflags = (n32(input_rflags)) : int in
  let raw_result = ((dst) + (src)) : int in
  let signed_raw_result = ((n32_to_i32(dst)) + (n32_to_i32(src))) : int in
  let result = (n_size(32, raw_result)) : int in
  let cf = (cf_spec32(raw_result)) : int in
  let pf = (pf_spec32(result)) : int in
  let af = (add_af_spec32(dst, src)) : int in
  let zf = (zf_spec(result)) : int in
  let sf = (sf_spec32(result)) : int in
  let of = (of_spec32(signed_raw_result)) : int in
  let output_rflags = (change_rflagsbits(input_rflags, Some(cf), None(),
      ↪ Some(pf), None(), Some(af), None(), Some(zf), Some(sf), None(),
      ↪ None(), None(), Some(of), None(), None(), None(), None(), None(),
      ↪ None(), None(), None(), None(), None())) : int in
  let output_rflags = (n32(output_rflags)) : int in
  let undefined_flags = (0) : int in
  (result, output_rflags, undefined_flags)
```

```
(sf (the (unsigned-byte 1)
         (sf-spec32 result)))
(of (the (unsigned-byte 1)
         (of-spec32 signed-raw-result)))
(output-rflags
 (mbe
  :logic (change-rflagsbits input-rflags
                            :cf cf
                            :pf pf
                            :af af
                            :zf zf
                            :sf sf
                            :of of)
  :exec
  (the
   (unsigned-byte 32)
   (!rflagsbits->cf
    cf
    (!rflagsbits->pf
     pf
     (!rflagsbits->af
        af
        (!rflagsbits->zf
            zf
            (!rflagsbits->sf
                sf
                (!rflagsbits->of of input-rflags)))))))))
(output-rflags (mbe :logic (n32 output-rflags)
                    :exec output-rflags))
(undefined-flags 0))
(mv result output-rflags undefined-flags))
```

Translation Notes

- The ACL2 and Sail for the top two functions look similar - perhaps unsurprisingly as they simply dispatch to the correct instruction specification function based on the opcode. Note the type annotation in the `match` expression in Sail: this is to avoid a type error (see `FutureWork.md` for more details.)
- The entirety of the ACL2 instruction specification function is shown for comparison with the size of the Sail translation. It is so much larger mostly because of the use of `mbe`s, which allow different implementations for logical reasoning vs. execution. The two branches are proved equivalent statically in ACL2 so we can choose which to translate into Sail: the `:logic` branch, translated here, produces clean code.
- One downside of the Sail code is the `change_rflagsbits()` function. In ACL2 keywords are used to pass values to this function. This could be mimicked in Sail by passing a struct instead.

- The `-4` at the end of both function names indicates these functions handle 4-byte (32 bit) operands. Code generation is used to create functions for different data widths (`-1`, `-2` and `-8`). It would be good to combine these into a single function which uses Sail's dependent type system (see `FutureWork.md` for details).

Operation

- The instruction specification function produces the result from the operands and calculates the updated flags.