EEL 5934 Formal Methods for Industrial Applications

Project Report on

Verification of a multiplication program for M1-machine using ACL2

Mohammad Bin Monjil

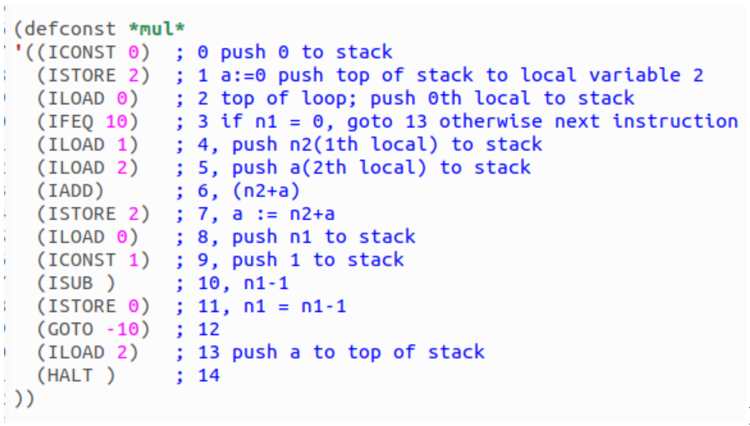
Md Saad Ul Haque

Introduction:

In this project, a multiplication program written for the M1-machine has been written and verified in ACL2 theorem prover. Two additional programs were also verified as stretch goals.

The Program:

The multiplication program was written using the provided instructions in the M1-machine. Designed only for natural numbers it was done by repeated addition. Specifically, a accumulator was initialized to zero and inside a loop multiplier was added to accumulator. The multiplicand was used as the loop counter which was being decreased by 1 in each iteration of the loop.



```
(defconst *mul*
'((ICONST 0)   ; 0 push 0 to stack
  (ISTORE 2)   ; 1 a:=0 push top of stack to local variable 2
  (ILOAD 0)    ; 2 top of loop; push 0th local to stack
  (IFEQ 10)    ; 3 if n1 = 0, goto 13 otherwise next instruction
  (ILOAD 1)    ; 4, push n2(1th local) to stack
  (ILOAD 2)    ; 5, push a(2th local) to stack
  (IADD)       ; 6, (n2+a)
  (ISTORE 2)   ; 7, a := n2+a
  (ILOAD 0)    ; 8, push n1 to stack
  (ICONST 1)   ; 9, push 1 to stack
  (ISUB )      ; 10, n1-1
  (ISTORE 0)   ; 11, n1 = n1-1
  (GOTO -10)   ; 12
  (ILOAD 2)    ; 13 push a to top of stack
  (HALT )      ; 14
))
```

Fig 1: Multiplication program

The program works by making an initial state with pc = 0 and loading the multiplier and multiplicand in the local variable 0 and 1 respectively. After the program runs for a minimum number of steps the multiplication result should on top of the stack and local variable 2. Variable 0 should become 0 while variable 1 being unchanged. So the initial and final states should look like the following:

States	pc	local	stack	program
Initial	0	(n0, n1)	nil	*mul*
Final	14	(0, n1, n0*n1)	(n0*n1)	*mul*

Table 1: Initial and Final State of M1-machine for the multiplication program

Proving Correctness of the program requires proving a theorem that states if the m1-machine starts with the initial state and runs for a minimum number of steps, the state will be equal to the Final state. Proving such a theorem requires to prove a number of lemmas to setup symbolic computation of the m1-machine and loop correcness.

Lemmas for symbolic computation:

The following two lemmas help with symbolic computation of the M1-machine.

```
(defthm m1-run-opener
(and (equal (m1-run s 0) s)
     (implies (natp i)
              (equal (m1-run s (+ 1 i))
                     (m1-run (m1-step s) i)))))

(defthm m1-step-opener
(and (implies (haltedp s) (equal (m1-step s) s))
     (implies (consp (next-inst s) )
              (equal (m1-step s) (do-inst (next-inst s) s)) )))
```

Essentially, they help us run the machine with symbolic state and state updates occuring from program executions.

Lemma for loop correctness:

The loop needs to be proved by induction. First, we need to analyze the loop iteration number. In our program, each iteration takes 11 steps and there are n0 number of loop iteration. We define this loop step number as a recursive function called *lp-clk which is* needed for proving the loop using induction.

```
(defun lp-clk (n)
(if (zp n)
    0
    (+ 11 (lp-clk (- n 1)))
))
```

In each iteration of the loop, n0 decreases by 1, n1 remains unchanged and n1 is added to n2. The loop runs untill n0 is zero. Our induction scheme must follow that. We define the following function that will help us setup the induction scheme.

```
(defun induction-plan (n0 n1 n2)
 (if (zp n0)
      (list n0 n1 n2)
     (induction-plan (- n0 1) n1 (+ n2 n1 )))))
```

The following lemma is needed in the induction step.

```
(defthm m1-run-split
( implies (and (natp p) (natp q))
          (equal (m1-run s (+ p q))  (m1-run (m1-run s p) q))))
```

With these necessary lemmas now we can prove the loop correctness.

```
(defthm loop-working-lemma
(implies
      (and (natp n0) (natp n1) (natp n2))
      (equal ( m1-run (make-state 2 (list n0 n1 n2) nil *mul*) (lp-clk n0) )
             ( make-state 2 (list 0 n1 (+ n2 (* n1 n0))) nil *mul* )))
      :hints (("Goal"
               :induct (induction-plan n0 n1 n2)

               )))
```

Essentially, the theorem states that starting from the top of the loop running the m1-machine for lp-clk(n0) number of steps would provide the desired state.

Proving the program correctness:

The program correctness can now to be proved using the previously proved lemmas. First, we need to analyze how many minimum number of steps would complete the program. This is basicllay (2 + (lp-clk n0) + 3) where 2 steps to get to the loop, lp-clk (n0) steps for the loop and 3 steps after the loop. However, writing this way, ACL2 always simplifies and makes it (5 + lp-clk) and runs for 5 steps and as a result can not use the loop-correctness lemma. To avoid this we defined the following function called clk-add and proved a supporting lemma.

```
(defun clk-add (x y z)
(+ x y z))

(defthm clk-add-run
( implies (and (natp x) (natp y) (natp z))
          (equal (m1-run s ( clk-add x y z))  (m1-run (m1-run (m1-run s x) y) z) )))
```

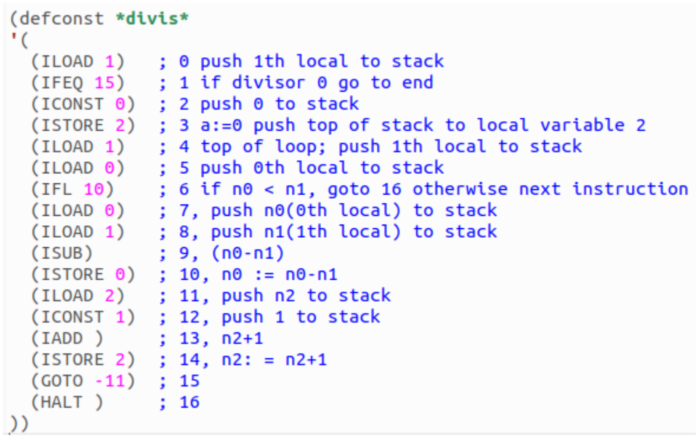
Finally, we proved the following theorem that states the corretness of the total program execution such that final state should match as described in table 1 while starting from the initial state.

```
(defthm program-working
(implies (and (natp n0)(natp n1))
         (equal (m1-run (make-state 0 (list n0 n1) nil *mul*) (clk-add 2 (lp-clk n0) 3 ))
                (make-state 14 (list 0 n1 (* n0 n1)) (list (* n0 n1)) *mul* ) )
         )
         :hints(
                ("Goal"
                  :do-not-induct t
                  :use ((:instance loop-working-lemma (n0 n0) (n1 n1) (n2 0)))
                  :in-theory (disable clk-add)
                )))
```

The clk-add function is disabled so that the function is not expanded which will defeat the purpose. A hint was provided to ACL2 to use an instance of the loop-working lemma with proper instantiation.

Stretch Goal 1:

We completed our strech goal 1 that is to write a division program and verify it in ACL2. One added complexity for this program was that the iteration dependent on two variables instead of one.



```
(defconst *divis*
'(
  (ILOAD 1)    ; 0 push 1th local to stack
  (IFEQ 15)    ; 1 if divisor 0 go to end
  (ICONST 0)   ; 2 push 0 to stack
  (ISTORE 2)   ; 3 a:=0 push top of stack to local variable 2
  (ILOAD 1)    ; 4 top of loop; push 1th local to stack
  (ILOAD 0)    ; 5 push 0th local to stack
  (IFL 10)     ; 6 if n0 < n1, goto 16 otherwise next instruction
  (ILOAD 0)    ; 7, push n0(0th local) to stack
  (ILOAD 1)    ; 8, push n1(1th local) to stack
  (ISUB)       ; 9, (n0-n1)
  (ISTORE 0)   ; 10, n0 := n0-n1
  (ILOAD 2)    ; 11, push n2 to stack
  (ICONST 1)   ; 12, push 1 to stack
  (IADD )      ; 13, n2+1
  (ISTORE 2)   ; 14, n2: = n2+1
  (GOTO -11)   ; 15
  (HALT )      ; 16
))
```

Fig 2. Division program

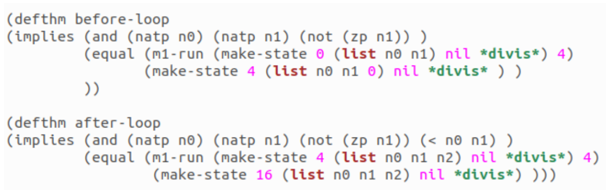
The division is performed through repeated subtraction untill dividend becomes less than the divisor, at this point that becomes the remainder while another variable stores the quotient. In our program if the divisor is zero the program simply halts without changing any variables. Otherwise, variable 0 stores the remainder and variable 3 stores the quotient.

States	pc	local	stack	program
Initial	0	(n0, n1)	nil	*divis*
Final	16	(mod(n0, n1), n1, floor(n0,n1))	nil	* divis*

Table 2. Initial and final state for the division program

As in the main goal, we proved a number of lemmas for formal symbolic computation. In a similar fashion we attempted to prove the loop correctness using induction. However, we faced an issue where ACL2 was not able to simplify arithmetic and prove a trivial equality. The issue was later resolved by including the arithmetic-5 book.

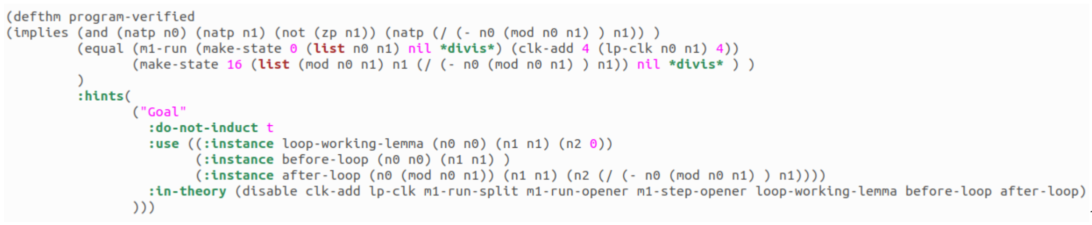
We faced another issue where ACL2 could prove program correctness up until the last iteration of the loop but not the rest of the program. There was a case-splitting explosion. As of writing this report, we have not figured out the cause of this issue. However, as a workaround, we proved two additional lemmas in addition to the loop correctness lemma. These lemmas can be used to verify the program in a compositional manner without relying on symbolic computation.



```
(defthm before-loop
(implies (and (natp n0) (natp n1) (not (zp n1)) )
         (equal (m1-run (make-state 0 (list n0 n1) nil *divis*) 4)
                (make-state 4 (list n0 n1 0) nil *divis* ) )
         ))

(defthm after-loop
(implies (and (natp n0) (natp n1) (not (zp n1)) (< n0 n1) )
         (equal (m1-run (make-state 4 (list n0 n1 n2) nil *divis*) 4)
                 (make-state 16 (list n0 n1 n2) nil *divis*) )))
```

We proved the following theorem that states the total program correctness. We hint ACL2

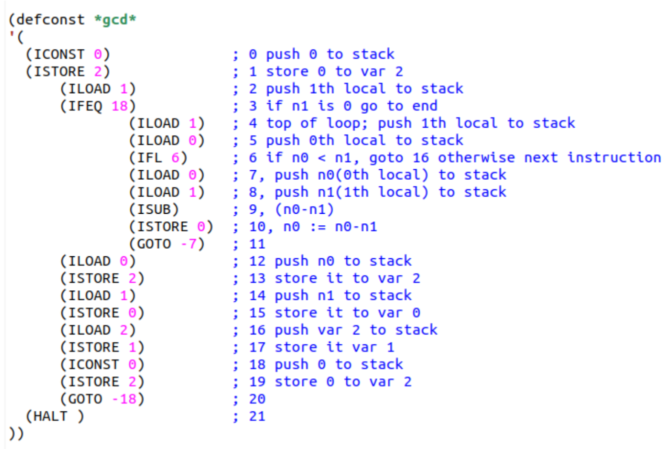


```
(defthm program-verified
(implies (and (natp n0) (natp n1) (not (zp n1)) (natp (/ (- n0 (mod n0 n1) ) n1)) )
        (equal (m1-run (make-state 0 (list n0 n1) nil *divis*) (clk-add 4 (lp-clk n0 n1) 4))
               (make-state 16 (list (mod n0 n1) n1 (/ (- n0 (mod n0 n1) ) n1)) nil *divis* ) )
        )
        :hints(
               ("Goal"
                 :do-not-induct t
                 :use ((:instance loop-working-lemma (n0 n0) (n1 n1) (n2 0))
                       (:instance before-loop (n0 n0) (n1 n1) )
                       (:instance after-loop (n0 (mod n0 n1)) (n1 n1) (n2 (/ (- n0 (mod n0 n1) ) n1))))
                 :in-theory (disable clk-add lp-clk m1-run-split m1-run-opener m1-step-opener loop-working-lemma before-loop after-loop)
               )))
```

to use the lemmas with proper instantiation. We also disable a number of lemmas to prevent symbolic computation.

Stretch Goal 2:

We also completed strech goal 2 which is to write and verify a program for gcd calculation. It was written using a nested loop with the inner loop calculating the modulas.



```
(defconst *gcd*
'(
  (ICONST 0)              ; 0 push 0 to stack
  (ISTORE 2)              ; 1 store 0 to var 2
      (ILOAD 1)           ; 2 push 1th local to stack
      (IFEQ 18)           ; 3 if n1 is 0 go to end
              (ILOAD 1)   ; 4 top of loop; push 1th local to stack
              (ILOAD 0)   ; 5 push 0th local to stack
              (IFL 6)     ; 6 if n0 < n1, goto 16 otherwise next instruction
              (ILOAD 0)   ; 7, push n0(0th local) to stack
              (ILOAD 1)   ; 8, push n1(1th local) to stack
              (ISUB)      ; 9, (n0-n1)
              (ISTORE 0)  ; 10, n0 := n0-n1
              (GOTO -7)   ; 11
      (ILOAD 0)           ; 12 push n0 to stack
      (ISTORE 2)          ; 13 store it to var 2
      (ILOAD 1)           ; 14 push n1 to stack
      (ISTORE 0)          ; 15 store it to var 0
      (ILOAD 2)           ; 16 push var 2 to stack
      (ISTORE 1)          ; 17 store it var 1
      (ICONST 0)          ; 18 push 0 to stack
      (ISTORE 2)          ; 19 store 0 to var 2
      (GOTO -18)          ; 20
  (HALT )                 ; 21
))
```

Fig 3. GCD program for M1 machine

Here variable 2 is used to perform value interchage between variable 0 and 1. We load 0 to variable 2 in the beginning to make the verification easier. We verified the program in compositional manner with first verifying the inner loop, then verifying the outer loop by verifying before the inner loop and after the inner loop portion of the outer loop. For before the inner loop portion we had to consider two cases where the code goes to inner loop and other being when the code exits the outer loop. We had to define two loop clocks one for each loop while the outer loop's clock had to be defined in terms of the inner loop.

```
(defun inner-lp-clk (n0 n1)
( if (and (natp n0)(natp n1)(not (zp n1)))
      (if (< n0 n1)
          3
          (+ 8 (inner-lp-clk (- n0 n1) n1)))
      0 ))
```

```
(defun outer-lp-clk (n0 n1)
( if (and (natp n0)(natp n1))
      (if (zp n1)
          2
          (clk-add 2 (inner-lp-clk n0 n1) 9 (outer-lp-clk n1 (mod n0 n1))  ))
      0 ))
```

```
(defun clk-add (a b c d)
(+ a b c d))

(defthm clk-add-run
( implies (and (natp a) (natp b) (natp c) (natp d) )
          (equal (m1-run s ( clk-add a b c d))  (m1-run (m1-run (m1-run (m1-run s a) b) c) d )))
          )
```

One difficulty we faced is that ACL2 was not triggering clk-add-run lemma on the outer-lp-clk term. For that we proved two additional lemmas that says that the loop clk functions always return natural.

To prove correctness, we defined our gcd function.

```
(defun my-gcd (a b)
  ( if ( and (natp a) (natp b))
  (if (zp b)
      a
      (my-gcd b (mod a b)))
     0 ))
```

States	pc	local	stack	program
Initial	0	(n0, n1)	nil	*gcd*
Final	21	(gcd(n0,n1), 0, 0)	nil	* gcd*

Table 3. Initial and final state for the gcd program

We verify the final theorem using all the lemmas. We assume n1 is not zero.

```
(defthm program-verified
(implies (and (natp n0)(natp n1)(not (zp n1)))
          (equal (m1-run (make-state 0 (list n0 n1) nil *gcd*) (+ 2 (outer-lp-clk n0 n1) ))
                 (make-state 21
                 (list (my-gcd n0 n1) 0 0)
                 nil
                 *gcd*)))
                 :hints (("Goal"
                          :use ( (:instance before-outer-loop)
                                 (:instance outer-loop-working-lemma (n0 n0)(n1 n1)(n2 0)) ))))
```

Conclusion:

In this project, we verified three programs for the M1-machine using ACL2. While working on the first goal, we learned how to verify a loop. The first strech goal added a little more complexity as iteration dependent on two variables. In the second strech goal we learned how to verify a nested loop. We learned how an interactive theorem prover works and discovered many tricks and workarounds to make it do what we want. Some issues like why symbolic computation was not working in the division program are yet to be solved. Overall, it was definitely a valuable experience. The fact that a computing system can be modeled with functions and verified using theorem proving was intriguing, to say the least.