Hardware Verification with Cadence SMV: A Multiplier Using Booth's Algorithm

Kristin Hofstee   Peeter Piegaze

March 20, 2000

Abstract

A binary multiplier is an integral part of the arithmetic logic unit (ALU) subsystem found in many processors. Integer multiplication can be inefficient and costly, in time and hardware, depending on the representation of signed numbers. Booth's algorithm suggests a technique for multiplying signed numbers that works well for both negative and positive multipliers. In this project, we have used Cadence-SMV for describing and verifying a hardware design based on Booth's algorithm. Cadence-SMV is a symbolic model verifier well suited to verifying combinational logic and interacting state machines and produces a step by step counter-example if a desired property is false. This allows for much easier debugging. Our analysis confirmed a special case in which Booth's algorithm is known to produce incorrect results.

1 Introduction

Digital hardware design is a lengthy and costly process. Many CAD tools have been built to aid in finding faulty designs before production. In this project, we use a symbolic model checker to verify the design of a 4-bit multiplier based on Booth's algorithm.

When dealing with signed integers most hardware uses a two's complement encoding to represent both positive and negative integers. The two's complement encoding allows n-bits to represent the numbers $-2^{(n-1)}$ to $2^{(n-1)} - 1$. While addition and subtraction of two's complement numbers is straightforward, multiplication is not. Several algorithms have been designed to efficient perform hardware multiplication. Booth's algorithm is one of these. It is a common computer arithmetic technique that, unlike other multipication methods, handles positive and negative multipliers uniformly. It also achieves some performance gain when the multiplier contains large blocks of 1s.

This paper describes the modelling and verification of a 4-bit by 4-bit multiplier using the Cadence-SMV formal verification tool. In the next section (Section 2) we briefly explain the intricacies of two's complement arithmetic, including Booth's algorithm. Section 3 describes the hardware design desired to implement a 4-bit two's complement multiplier. The Cadence-SMV language and the model of the multiplier are explained in Section 4. Verification of the system's behaviour is discussed in Section 5. Finally, in Section 6, our conclusions are presented.

2 Signed Number Binary Arithmetic

In this section the specific arithmetic techniques used in the design are presented. A more thorough discussion of alternative techniques and representation and those presented here can be found in [1]. Familiarity with the basic binary representation of integers is assumed.

2.1 Two's Complement

As stated previously, the two's complement encoding means that the numbers $-2^{(n-1)}$ through $2^{(n-1)} - 1$ can be represented in n-bits. For example, given 3-bits,the numbers 0 through 3 (i.e., $2^{(3-1)} - 1$) are represented as expected in a normal binary representation:

$000 \rightarrow 0$
$001 \rightarrow 1$
$010 \rightarrow 2$
$011 \rightarrow 3$

To form the two's complement the number is first complemented, ie. ones become zeros, zeros become ones, 101 becomes 010. and then 1 is added to the complemented number. To take the two's complement of 011, first, invert the number to get 100, then add one to get 101. If the negative of a number is assigned to it's two's complement then the numbers -4 through 0 can be represented as:

$100 \rightarrow -4$
$101 \rightarrow -3$
$110 \rightarrow -2$
$111 \rightarrow -1$

2.2 Addition and Subtraction

The addition of signed numbers using a two's complement representation is done as for unsigned numbers. Overflow occurs when the sum of two n-bit numbers cannot be represented in n bits.

Subtraction of two numbers, a and b $(a-b)$ is done by forming the two's complement of b and then adding a. Again, the sum will be the correct value if the result falls in the $-2^{(n-1)}$ through $2^{(n-1)}-1$ range.

2.3 Multiplication

Signed-number multiplication is not as straightforward as addition or subtraction. Booth's algorithm is a powerful algorithm that computes a $2n$-bit product and is uniform for both positive and negative numbers. In this algorithm the bits of the multiplier are examined and recoded. The Booth recoding consists of 0s, +1s and -1s and is found by examining the i^{th} and $(i-1)^{th}$ bit of the multiplier. In the case of the least significant bit the $(i-1)^{th}$ bit is taken to be zero.

Bit i	Bit $i-1$	Recoding of bit i
0	0	0
0	1	+1
1	0	-1
1	1	0

For example the Booth recoding of 1011 would be $-1, +1, 0, -1$. Therefore the computation of 1010 (-6) times 1011 (-5) is:

				1	0	1	0
			$\times$	-1	+1	0	-1
0	0	0	0	0	1	1	0
0	0	0	0	0	0	0	
1	1	1	0	1	0		
0	0	1	1	0			
0	0	0	1	1	1	1	0

This final binary number (0001110) is equal to 30.

3 Hardware Design of an n-bit Multiplier

The block diagram of the hardware needed to build a multiplier is shown in Figure 1. The behaviour and design of the adder-subtractor and the shift registers will be described followed by a discussion of the implementation of Booth's algorithm on the described hardware (Figure 1).

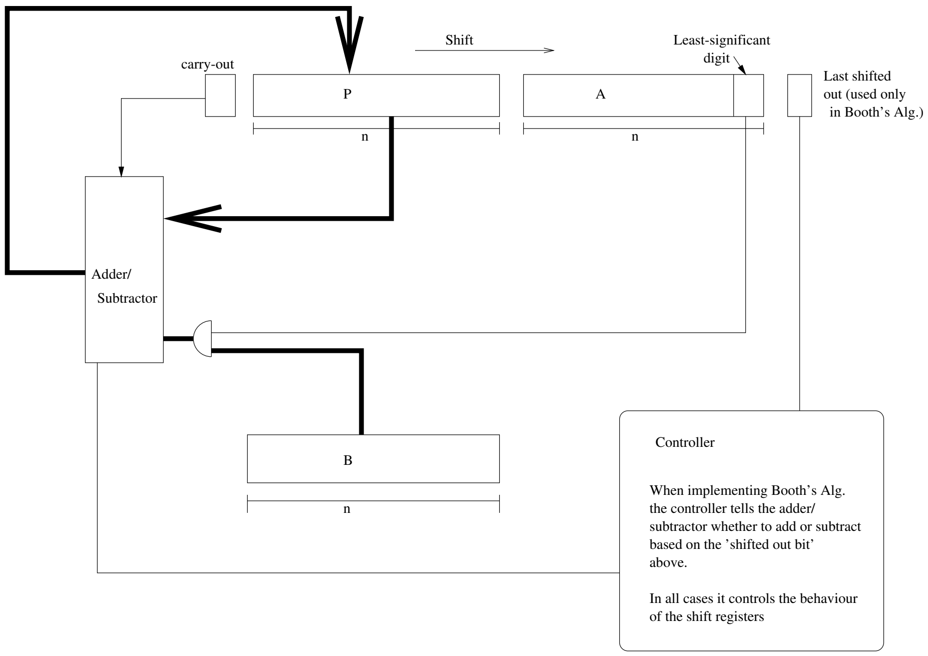

Figure 1: The Multiplier

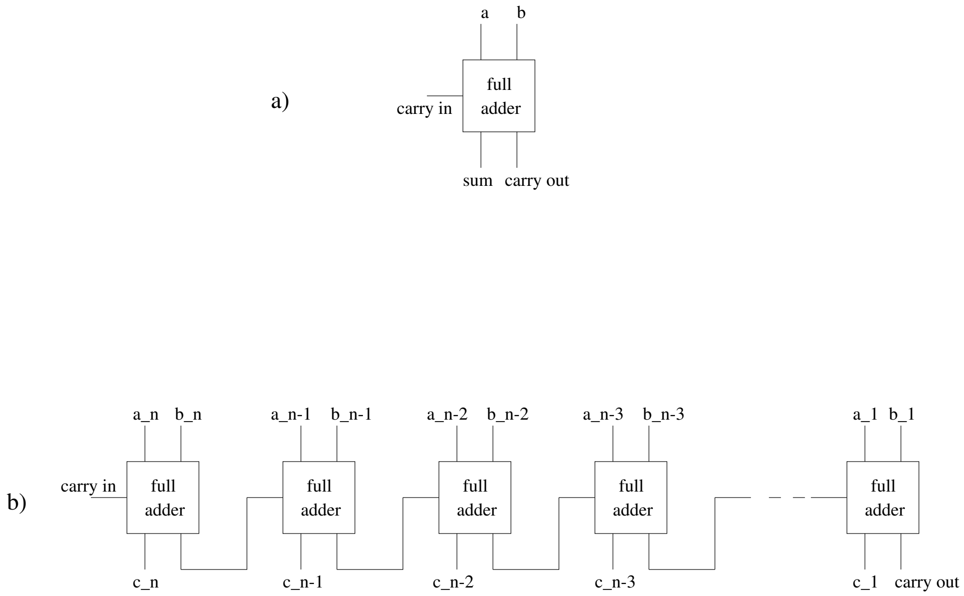

Figure 2: a) Full Adder b) Ripple Carry Adder

3.1 Adder-Subtractor

Multi-bit adders are usually built by combining multiple copies of simple components. The components are usually full adders. These are simple devices which take as input two bits, a and b as well as a carry-in bit c_{in}. The output is the sum bit s and the carry-out bit c_{out}. By stringing a number of these together in such a way that the carry-out of each bit goes into the carry-in bit of the next bit to the left we can build a multi-bit ripple carry adder. The lowest order carry-in bit is, in this case, wired to be permanently 0 (see Figure 2).

Interestingly, such a design can also be easily converted into a subtractor. Recall that in two's complement form, positive numbers are represented as normal binary numbers while negative numbers are represented by taking the corresponding positive number, complementing each of its bits, and adding one. We can use this system to implement subtraction on our adder hardware. In order to implement $a-b$, we simply feed a and $\bar{b}$ (the bit-wise complement of b) into the adder and set the lower-order carry-in bit to 1, instead of the usual 0. This has the effect of adding 1 to the sum of a and $-b$, thus implementing two's complement subtraction.

3.2 Shift Registers

Like the adder, a shift register is built by combining copies of simple components. In this case, each component is a simple flip-flop with some combinational logic. In its most basic form, a flip-flop has an input, receives a clock signal, and has an output. The value of the input when the clock signal is asserted appears at the output and remains at the output until the next time the clock signal is asserted. Therefore, a flip-flop can be used to store one bit.

Flip-flops can be combined to make a register. Each flip-flop in the structure runs off a common clock. That way the input values are loaded into all flip-flops at the same time. The multiplier will require that we shift the data in a register to the right. Data is shifted one bit to the right whenever a `shift` signal is asserted. Suppose we have a 4-bit register containing 1101. We want to shift all the data one bit to the right. The third bit will become the second bit, the second bit will become the first bit, and so on until the zeroth bit is shifted out. We also must specify a bit to be shifted in, say 0. After the shift, the register which contained 1101 will contain 0110; 0 is shifted into the most significant position; all the other bits shift one bit to the right, and a 1 is shifted out.

The shift registers in the multiplier are versatile. They can be shifted, loaded with a new value or cleared (set to all 0s).

3.3 Booth's Algorithm

Booth's algorithm computes the product of two signed numbers (see [1] and [2]). It will be useful to refer to Figure 1 when reading through the algorithm. The numbers $a_n \dots a_0$ and $b_n \dots b_0$ are placed in register A and B, respectively. Register P is initially 0. Each step of the multiplication has two parts:

Part 1: **a)** if $a_0 = 0$ and $a_{lastout} = 0$ then add 0 to P

b) if $a_0 = 0$ and $a_{lastout} = 1$ then add B to P

c) if $a_0 = 1$ and $a_{lastout} = 0$ then subtract 0 from P

d) if $a_0 = 1$ and $a_{lastout} = 1$ then add 0 to P

Place the result back in P. This part implements the booth recoding of the multiplier.

Part 2: P and A are shifted right with the value the most-significant bit being shifted into the leftmost bit of P (called an arithmetic shift), the rightmost bit of P moving to the leftmost bit of A and the rightmost bit of A being shifted out but kept track of for the next iteration of part 1.

After n iterations the product appears in the concatenation of registers P and A with P holding most significant bit and A holding the least significant bits.
Example: Multiply -6=1010 by -5=1011

P	A	B		
0000	1010	1011	Put -6 into A and -5 into B	
0000	1010	1011	Iteration 1:	Part 1.a add 0
0000	0101	1011		Part 2 shift
+0101			Iteration 2:	Part 1.c subtract b (or add -b)
0101	0101	1011		
0010	1010	1011		Part 2 shift
+1011			Iteration 3:	Part 1.b add b
1101	1010	1011		
1110	1101	1011		Part 2 shift
+0101			Iteration 4:	Part 1.c subtract b (or add -b)
0011	1101	1011		
0001	1110	1011		Part 2 shift

Final result is 0001 1110 = 30.

3.4 Control Logic

A finite state machine is used to generate the control signals that tell the shift registers when to shift and when to load, and to tell the adder-subtractor whether it should add or subtract. It also keeps track of the number of iterations and asserts a signal when the multiplication is complete. Finite state machines are implemented with combinational logic and flip-flops which hold the state of the machine and whose output is fed back to the combinational logic (see [1]).

4 The Implementation

4.1 A Quick Introduction to the SMV Language

The SMV language [3] includes type declarations and assignments for signals, constructs for defining modules and structured data types and conditional structures. The latter, especially, makes describing complicated state transitions easier. SMV's expressions are similar to those in hardware description languages and programming languages. A good introduction to Cadence-SMV and the SMV language can be found in [4].

Here we have an example of a module describing a counter bit. A number of such modules could be chained together to form a counter. A module functions much like subroutine. It may have formal parameters. When a module is instantiated, actual signals or expressions are plugged in for the formal parameters. Formal parameters can either be inputs or outputs. The inputs are assigned outside the module, and the outputs are assigned inside the module.

```
MODULE counter_bit(carry_in, clear, bit_out, carry_out)
{
   INPUT carry_in, clear: boolean;
   OUTPUT bit_out, carry_out: boolean;
   next(bit_out) := clear ? 0 : (carry_in ^ bit_out);
   carry_out := carry_in & bit_out;
}
```

The `INPUT` and `OUTPUT` keywords declare the direction of the signals being declared. The `next` operator is a special operator provided for describing recursive systems of equations. This allows sequential systems to be described in SMV. If `x` is a signal, then `next(x)` is intuitively the 'next' value of `x`. In the example above, the 'next' value of `bit_out` will 0 if `clear` is true, otherwise, it will be the exclusive-or of `carry_in` and the current value of `bit_out`. An assignment as in the example above means simply that the left and right sides are equal. `carry_out` is `carry_in & bit_out`. In SMV, the notions of what to compute and when to compute it are separate.

Now we can create four instances of the module `counter_bit` to create a four bit counter.

```
  clear: boolean;
  count: array 3..0 of boolean;

  carry: array 3..0 of boolean;

  bit0: counter_bit(1, clear, count[0], carry[0]);
  bit1: counter_bit(carry[0], clear, count[1], carry[1]);
  bit2: counter_bit(carry[1], clear, count[2], carry[2]);
  bit3: counter_bit(carry[2], clear, count[3], carry[3]);
```

The instances have been named `bit0`, `bit1`, `bit2` and `bit3`. Each instance is copy of the `counter_bit` module. The signals in module `bit0` are `bit0.carry_in, bit0.clear, bit0.bit_out, bit0.carry_out`; thus, the signals in different modules can be differentiated. We have most of the tools we need in this counter example to build the multiplier.

4.2 The Model

The model of a 4-bit by 4-bit two's complement multiplier has basically the same components as the hardware specification, shown in Fig. 1. In this section, the design of each module and the control logic will be explained. The full model can be found in Appendix A.

The control logic consists of a finite state machine and a counter. It produces the various signals to control the proper loading and shifting of the register and the adding and subtracting as demanded by Booth's Algorithm. In SMV, each unit is specified in a particular module.

4.2.1 Shift Register

The shift register was implemented by stringing together four simple `shift_bit` modules.

```
MODULE shift_bit(shift_in, in_bit, shift, clear,
                 load_b, bit_out, shift_out)
```

`shift, clear, load_b` are signals which dictate the behaviour of the module. If `clear` is asserted, then the next values of `bit_out` and `shift_out` are 0 and 0; if `load_b` is asserted the `in_bit` will become the `bit_out`; if `shift` is asserted, then the next value of `bit_out` is the value of `shift_in`, and the next value of `shift_out` is the value of `bit_out`. This is easily described using an `if` structure:

```
 if (clear) {                 -- clear bit_out to zero
   next(bit_out) := 0;         -- clear shift_out to zero
   next(shift_out) := 0;
  }
  else if (load_b) {           -- load a new value
   next(bit_out) := in_bit;    -- clear shift_out to zero
   next(shift_out) := 0;
  }
  else if (shift)
```

```
  {
    next(bit_out) := shift_in;  -- shift in the new value
    next(shift_out) := bit_out; -- shift out the old value
  }
  else
  {
    next(bit_out) := bit_out;     -- nothing changes
    next(shift_out) := shift_out;
  }
```

Clearly, the precedence of signals is `clear, load_b` and then `shift`. The shift register latches four bits on a load. `shift_bit` modules can be strung together to form a larger shift register. The shift register implementation uses four `shift_bit` modules in parallel in the `shift_register` module.

```
MODULE shift_register(shift_in, in_vec, shift, load_s,
                      clear, shift_out, out_vec)
```

The `load` signal causes the register, i.e., the output vector of the register, to be loaded with the vector provided to the module. The `shift` signal causes all bits to be shifted right and the value of `shift_in` to become the most significant bit of the output vector. Upon assertion of the `clear` signal, the output vector is set to 0.

4.2.2 Adder and Subtractor

The adder is implemented as described in Section 3.1. The `add_bit` module describes a full-adder.

```
MODULE add_bit(carry_in, add1, add2, sum, carry_out)
```

Addition is easily implemented in combinational logic. The sum is the exclusive-or of the `carry_in` and the two arguments. The `carry_out` is one if any two of the inputs are 1.

```
   next(sum) := carry_in ^ add1 ^ add2;
   carry_out := (carry_in & add1) | (carry_in & add2) | (add1 & add2);
```

By stringing together 4 `add_bit` instances with the first `carry_in` set to 0, a 4-bit module can be described.

The subtraction module is implemented by stringing together full-adders with the first `carry_in` set to 1 and the second argument complemented.

```
sub0: add_bit(1, a_reg[0], !b_reg[0], diff[0], carry[0]);
sub1: add_bit(carry[0], a_reg[1], !b_reg[1], diff[1], carry[1]);
sub2: add_bit(carry[1], a_reg[2], !b_reg[2], diff[2], carry[2]);
sub3: add_bit(carry[2], a_reg[3], !b_reg[3], diff[3], carry_out);
```

Finally, an `addorsubtract` module was written with two control signals.

```
MODULE addorsubtract(a, b, c, add_sub, zero)
```

If `add_sub` were 0, the result of `a + b` is placed in `c`, otherwise the result of `a - b` is placed in `c`. If the signal `zero` was asserted, regardless of the value of `add_sub`, `c` holds the result of `a + 0`.

4.2.3 Control

The control logic consists of a finite state machine and a counter. It produces the various signals to control the proper loading and shifting of the register and the adding and subtracting as demanded by Booth's algorithm. The particular control signals asserted are dependent on the state of the controller. Since a number of states are visited repeatedly, a counter is used to track when the multiplication is complete.

The finite state machine in the model has nine states. The states are described below and the state transistion diagram can be found in Figure 3.

Wait: The multiplier idles in the *Wait* state until `start` is asserted. The value of a and b when start is asserted become the multiplicand and the multiplier for the multiplication.

Clear P and Load A: The P register will eventually hold the 4 most significant bits of the product and A will hold the 4 least significant bits. To begin the multiplication, P is clear and A is loaded with a, the multiplicand.

Choose: After A and P are loaded, or after the last shift, a choice is made depending on the least significant bit of A and the bit last shifted out. This part essentially implements the Booth recoding.

Add 0 to P: Zero is added to the current value of P.

Add B to P: B, the multiplier, is added to the current value of P.

Subtract B from P: B, the multiplier, is subtracted from the current value of P.

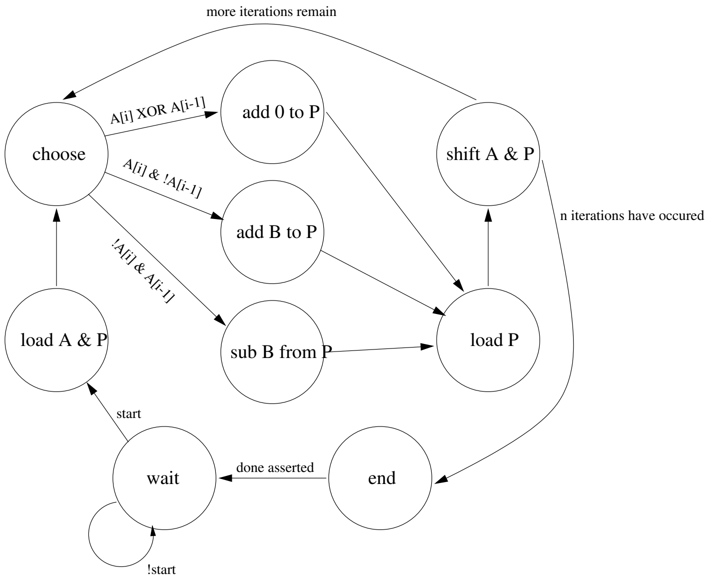

Figure 3: The Control Signal Generator

Load P: The result of the addition or subtraction is loaded back into P.

Shift: P and A are shifted to the right. P uses an arithmetic shift, that is the value of the most significant bit of P renains where it is, *and* is also shifted to the right. The least significant bit of P is shifted out and shifted in to the most significant bit of A. The value shifted out of A is kept track of to choose addition or subtraction in the *Choose* state. If this is the fourth shift, the next state is *End*, otherwise, the next state is *Choose*.

End: The result held in P and A is moved to the output vector, `product`, and the `done` signal is asserted. The multiplier then moves back to the *Wait* state.

The state transistion table is described in SMV by a case statement where the next state is based on the current state and the value of various environment values such as the lower order bits of register A.

```
next(state):=
   case {
     (state = wait): start ? loadAP : wait;
```

```
    (state = loadAP): choose;
    (state = choose & !(A_reg[0] ^ a_out)): add0toP;
    (state = choose & !A_reg[0] & a_out): addBtoP;
    (state = choose & A_reg[0] & !a_out): subBfromP;
    (state = add0toP): loadP;
    (state = addBtoP): loadP;
    (state = subBfromP): loadP;
    (state = loadP): shift_reg;
    (state = shift_reg & count < [1,1,0,0]): choose;
    (state = shift_reg & count >= [1,1,0,0]): end;
    (state = end): wait;
   };
```

The control signals produced in each state are described in a similar table.

4.3 Verification

We expect a 4-bit by 4-bit multiplier to have certain behaviours. First, we could try specifying an exact input and describe the output desired in an assertion such as:

```
mult1: assert G ((a = [1,0,1,1] & b = [1,0,1,0] & start &
       state = wait) -> F (done = 1 & product = [0,0,0,1,1,1,1,0]));
```

Assertions in Cadence-SMV are written in LTL. For example the above assertion claims that it will always be true that if a equals [1,0,1,1], b equals [1,0,1,0], the signal start has been asserted and the state machine is in the *wait* state then in the future *done* will be asserted and the product will be [0,0,0,1,1,1,1,0].

This property confirms that if `start` is asserted or if either a or b changes during multiplication then product is still the result of $a \times b$ as specified when `start` was initially asserted.

We also want to ensure that `done` is not asserted before the product is calculated, i.e., `done` should only be asserted after leaving the *End* state.

```
done_check1: assert G (done = 0 U state = wait);
done_check2: assert G (state = end -> done = 1);
```

Other desired properties in a multiplier:
The product of a positive and a positive should be positive.

```
mult_a: assert G ((a[3] = 0 & b[3] = 0 & start & state = wait) ->
        F (done = 1 & product[7] = 0));
```

The product of a negative and a positive should be negative.

```
mult_b1: assert G ((a[3] = 1 & b[3] = 0 & !(b[2..0] = [0,0,0]) &
         start & state = wait) -> F (done = 1 & product[7] = 1));

mult_b2: assert G ((a[3] = 0 & b[3] = 1 & !(a[2..0] = [0,0,0]) &
         start & state = wait) -> F (done = 1 & product[7] = 1));
```

The product of a negative and a negative should be positive.

```
mult_c: assert G ((a[3] = 1 & b[3] = 1 & !(a[2..0] = [0,0,0]) &
        !(b[2..0] = [0,0,0]) & start & state = wait) ->
        F (done = 1 & product[7] = 0));
```

The product of 0 and a number should be 0.

```
mult_d: assert G (((a = [0,0,0,0] | b = [0,0,0,0]) & start &
        state = wait) -> F (done = 1 & product[7] = 0));
```

The product of an even number and any number should be even.

```
mult_e: assert G (((a[0] = 0 | b[0] = 0) & start & state = wait) ->
        F (done = 1 & product[0] = 0));
```

The product of two odd numbers should be odd.

```
mult_f: assert G ((a[0] = 1 & b[0] = 1 & start & state = wait) ->
        F (done = 1 & product[0] = 1));
```

All these properties were verified to be true except for `mult_b1` and `mult_b2`. Upon further examination it was found that there is a flaw in the hardware implementation of Booth's Algorithm described in [1]. The multiplier is unable to handle the case where b (from $a \times b$) is $-2^{(n-1)}$. In this case the arithmetic shift is not handled correctly. The algorithm would have to extended to detect this case.

Verification was performed on a four processor Sun Ultra Sparc with 1 Gb RAM running SunOS 5.5.1 using Cadence-SMV last modified 23 January 1996.

Verification Statistics	
user time	60.00 s
system time	0.44 s
BDD nodes allocated	702524
Reachable states:	
number of states holding booleans	63
number of states reached	6.87781e+07

5 Conclusion

Cadence-SMV was quite easy to use. The documentation provided is quite extensive although often quite dense. Previous experience using a symbolic model checker facilitated the use of this new system. The main benefit of the graphical interface is the clear display shown for the step by step, variable by variable, counter example provided when a specified property does not hold. This makes the graphical display a very effective debugging tool.

The timing model used by Cadence-SMV makes the system conducive to hardware design verification. Mealy machines are easily written and verified. In Mealy machines, the output is associated with the state transition as opposed to Moore machines where the output is associated with the state. This type of finite state machine are often tricky to specify. It required several iterations of checking to get the timing of the control signals for the multiplier exactly right. Another, roadblock was the exact understanding of the timing interaction when the output of one module is the input to another module. Again, the graphical interface was very useful in understanding timing issues.

One error in particular was tricky to track down. The verifier did not report that a control signal was being assigned as the state of the control FSM. The confusion was between the shift signal and the shift state.

The multiplier can be easily extended to n-bits by increasing the size of the arrays, ensuring that the adder, subtractor and shift register have extra bits added to them and that the final value of the counter is changed. A logical extension of the multiplier would be to design a divider and then model an ALU for verification.

Overall, Cadence-SMV, once learned, is a convenient tool with excellent documentation and interface.

6 References

1. V.C. Hamacher, Z.G. Vranesic, S.G. Zaky, Computer Organization, fourth edition, Toronto, 1996

2. D. Patterson, J. Hennessey, Computer Architecture A Quantitative Approach, 2nd ed., Berkeley, CA, 1990

3. K.L. McMillan, The SMV Language, Berkeley, CA, 1999

4. K.L. McMillan, Getting Started with SMV, Berkeley, CA, 1999

7 Appendix A

```
MODULE main()
{
  -- GOAL multiply a by b

  a, b: array 3..0 of boolean;     -- a is multiplicand, b is multiplier
  start: boolean;    -- signal to start multiplying
  product: array 7..0 of boolean;  -- product of the multiplication
  done: boolean;    -- signal when multiplication is done

  -- controller states
  state : {wait, end, choose, shift_reg, loadAP,
            add0toP, addBtoP, subBfromP, loadP};

  -- control signals
  clear_count, clear_A, clear_P: boolean;
  load_A, load_P, shift, a_out, p_out: boolean;
  add_sub, add_zero: boolean;

  -- intermediate holders of a and b
  b_in, a_in: array 3..0 of boolean;

  -- shift registers, P_reg forms the 4 most sig. bits of the product
  --                  A_reg forms the 4 least sig. bits
  A_reg, P_reg: array 3..0 of boolean;

  -- result holds the sum or difference for the adder/subtractor,
  -- it is then loaded into P_reg
  -- count, holds the count which is used to keep track of the iterations
  result, count: array 3..0 of boolean;

  countup: counter(clear_count, count);

  as: addorsubtract(P_reg, b_in, result, add_sub, add_zero);

  reg_A: shift_register(P_reg[0], a_in, shift, load_A, clear_A, a_out, A_reg);
  reg_P: shift_register(P_reg[3], result, shift, load_P, clear_P, p_out, P_reg);
  next(product[7..4]) := P_reg;
  next(product[3..0]) := A_reg;
```

```
-- Everything below constitutes the state machine for generating the
-- appropriate control signals

-- initial values of control signals
init(state):= wait;
init(load_A) := 0;
init(load_P) := 0;
init(shift) := 0;
init(clear_count) := 1;
init(clear_A) := 0;
init(clear_P) := 1;
init(add_sub) := 0;
init(add_zero) := 0;
init(done) := 0;
init(a_in) := a;
init(b_in) := b;

-- latch a and b so that if they are changed while the multiplication
-- is in progress the correct product is still produced.
next(a_in) := (state = end | state = wait |
               (state = shift_reg & count >= [1,1,0,0])) ? a : a_in;
next(b_in) := (state = end | state = wait |
               (state = shift_reg & count >= [1,1,0,0])) ? b : b_in;


-- state transition table
next(state):=
  case {
    (state = wait): start ? loadAP : wait;
    (state = loadAP): choose;
    (state = choose & !(A_reg[0] ^ a_out)): add0toP;
    (state = choose & !A_reg[0] & a_out): addBtoP;
    (state = choose & A_reg[0] & !a_out): subBfromP;
    (state = add0toP): loadP;
    (state = addBtoP): loadP;
    (state = subBfromP): loadP;
    (state = loadP): shift_reg;
    (state = shift_reg & count < [1,1,0,0]): choose;
```

```
    (state = shift_reg & count >= [1,1,0,0]): end;
    (state = end): wait;
  };


-- control signals generated for each state
(next(load_A), next(load_P), next(shift), next(clear_count),
next(clear_A), next(clear_P), next(add_sub), next(add_zero),
next(done)):=
  case {
    (state = wait & !start) : (0,0,0,1,0,1,0,0,0);
    (state = wait & start) : (1,0,0,1,0,1,0,0,0);
    (state = loadAP) : (0,0,0,0,0,0,0,0,0); --check/choose
    (state = choose & !(A_reg[0] ^ a_out)): (0,0,0,0,0,0,0,1,0); -- add0toP
    (state = choose & !A_reg[0] & a_out)  : (0,0,0,0,0,0,0,0,0); -- addBtoP
    (state = choose & A_reg[0] & !a_out)  : (0,0,0,0,0,0,1,0,0); -- subBfromP
    (state = add0toP) : (0,1,0,0,0,0,0,1,0); -- loadP
    (state = addBtoP) : (0,1,0,0,0,0,0,1,0); -- loadP
    (state = subBfromP)               : (0,1,0,0,0,0,0,1,0); -- loadP
    (state = loadP) : (0,0,1,0,0,0,0,1,0); -- shift
    (state = shift_reg & count < [1,1,0,0]): (0,0,0,0,0,0,0,0,0); -- choose;
    (state = shift_reg & count >= [1,1,0,0]): (0,0,0,0,0,0,0,0,0); -- end;
    (state = end) : (0,0,0,1,0,0,0,0,1);
  };

--  -5 * -6 = 30
mult1: assert G ((a = [1,0,1,1] & b = [1,0,1,0] & start & state = wait) ->
                 F (done = 1 & product = [0,0,0,1,1,1,1,0]));

-- pos * pos
mult_a: assert G ((a[3] = 0 & b[3] = 0 & start & state = wait) ->
                  F (done = 1 & product[7] = 0));
-- neg * pos
mult_b1: assert G ((a[3] = 1 & b[3] = 0 & !(b[2..0] = [0,0,0]) &
                    start & state = wait) ->
                   F (done = 1 & product[7] = 1));
-- pos * neg
mult_b2: assert G ((a[3] = 0 & b[3] = 1 & !(a[2..0] = [0,0,0]) &
                    start & state = wait) ->
                   F (done = 1 & product[7] = 1));
```

```
  -- neg * neg
  mult_c: assert G ((a[3] = 1 & b[3] = 1 & !(a[2..0] = [0,0,0]) &
                     !(b[2..0] = [0,0,0]) & start & state = wait) ->
                    F (done = 1 & product[7] = 0));
  -- 0 * anything
  mult_d: assert G (((a = [0,0,0,0] | b = [0,0,0,0]) & start & state = wait) ->
                    F (done = 1 & product = [0,0,0,0,0,0,0,0]));
  -- even * anything
  mult_e: assert G (((a[0] = 0 | b[0] = 0) & start & state = wait) ->
                    F (done = 1 & product[0] = 0));
  -- odd * odd
  mult_f: assert G ((a[0] = 1 & b[0] = 1 & start & state = wait) ->
                    F (done = 1 & product[0] = 1));
  done_check1: assert G (done = 0 U state = wait);
  done_check2: assert G (state = end -> X done = 1);
}

-- Four bit counter, increment the count by 1 every time step
MODULE counter(clear, count)
{

  INPUT clear: boolean;
  OUTPUT count: array 3..0 of boolean;

  carry: array 3..0 of boolean;

  bit0: counter_bit(1, clear, count[0], carry[0]);
  bit1: counter_bit(carry[0], clear, count[1], carry[1]);
  bit2: counter_bit(carry[1], clear, count[2], carry[2]);
  bit3: counter_bit(carry[2], clear, count[3], carry[3]);

}

-- counter bit, add the carry_in and the current value of the bit
MODULE counter_bit(carry_in, clear, bit_out, carry_out)
{
   INPUT carry_in, clear: boolean;
   OUTPUT bit_out, carry_out: boolean;

   next(bit_out) := clear ? 0 : (carry_in ^ bit_out);
```

```
   carry_out := carry_in & bit_out;

}

-- Adds a and b, subtracts a from b or adds 0 to a depending on the
-- control signals
MODULE addorsubtract(a, b, c, add_sub, zero)
{
  -- add_sub, 1 add, 0 sub
  INPUT add_sub: boolean;
  INPUT a, b: array 3..0 of boolean;
  OUTPUT c: array 3..0 of boolean;

  carry: boolean;
  diff, sum: array 3..0 of boolean;
  latch_as: boolean;
  latch_z: boolean;
  latch_a: array 3..0 of boolean;

  next(latch_as) := add_sub;
  next(latch_z) := zero;
  next(latch_a) := a;

  add: adder(a, b, sum, carry);
  subtract: subtractor(a, b, diff);

  if (latch_z) {
    c := latch_a;
  }
  else if (latch_as) {
    c := diff;
  }
  else {
    c := sum;
  }

-- Assertions used while building up the multiplier,  for verifying
-- different parts of the multiplier
  adding1: assert G ((a = [0,1,1,1] & b = [0,0,0,1]) -> X (sum = [1,0,0,0]));
  adding2: assert G ((a = [1,1,1,1] & b = [0,0,0,1]) -> X (sum = [0,0,0,0]));
```

```
  sub1: assert G ((a = [0,1,1,0] & b = [0,0,1,1]) -> X (diff = [0,0,1,1]));
  sub2: assert G ((a = [1,0,1,1] & b = [0,0,1,1]) -> X (diff = [1,0,0,0]));
  sub3: assert G ((a = [1,0,1,1] & b = [1,1,1,0]) -> X (diff = [1,1,0,1]));
  sub4: assert G ((a = [0,0,1,0] & b = [1,0,1,1]) -> X (diff = [0,1,1,1]));

}

-- Shift register, a 4-bit vector can be loaded into the register, the
-- register can be cleared, ie. all bits set to 0, or the bits can
-- be shifted to the right, ie. in the least significant direction, the
-- most significant bit takes on the value of the signal 'shift_in'
MODULE shift_register(shift_in, in_vec, shift, load_s, clear, shift_out, out_vec)
{
  INPUT shift_in, shift, load_s, clear: boolean;
  INPUT in_vec: array 3..0 of boolean;
  OUTPUT shift_out: boolean;
  OUTPUT out_vec: array 3..0 of boolean;

  out: array 3..1 of boolean;
  -- out is indexed out[3], out[2] and out[1]

  bit3: shift_bit(shift_in, in_vec[3], shift, clear, load_s,
                  out_vec[3], out[3]);
  bit2: shift_bit(out_vec[3], in_vec[2], shift, clear, load_s,
                  out_vec[2], out[2]);
  bit1: shift_bit(out_vec[2], in_vec[1], shift, clear, load_s,
                  out_vec[1], out[1]);
  bit0: shift_bit(out_vec[1], in_vec[0], shift, clear, load_s,
                  out_vec[0], shift_out);

}

-- single bit implementation of a shift register
MODULE shift_bit(shift_in, in_bit, shift, clear, load_b, bit_out, shift_out)
{

  INPUT shift_in, in_bit, shift, clear, load_b: boolean;
  OUTPUT bit_out, shift_out: boolean;

  init(bit_out) := 0;
```

```
  init(shift_out) := 0;

  -- precedence is clear, load and then shift

  if (clear) { -- clear bit_out to zero
    next(bit_out) := 0;         -- clear shift_out to zero
    next(shift_out) := 0;
  }
  else if (load_b) { -- load a new value
    next(bit_out) := in_bit; -- clear shift_out to zero
    next(shift_out) := 0;
  }
  else if (shift)
  {
    next(bit_out) := shift_in;  -- shift in the new value
    next(shift_out) := bit_out; -- shift out the old value
  }
  else
  {
    next(bit_out) := bit_out;   -- nothing changes
    next(shift_out) := shift_out;
  }

}

-- adds carry_in, add1 and add2
MODULE add_bit(carry_in, add1, add2, sum, carry_out)
{

   INPUT carry_in, add1, add2: boolean;
   OUTPUT sum, carry_out: boolean;

   next(sum) := carry_in ^ add1 ^ add2;
   carry_out := (carry_in & add1) | (carry_in & add2) | (add1 & add2);


}

-- A 4-bit adder
MODULE adder(a_reg, b_reg, sum_reg, carry_out)
```

```
{
  INPUT a_reg, b_reg: array 3..0 of boolean;
  OUTPUT sum_reg: array 3..0 of boolean;
  OUTPUT carry_out: boolean;

  carry: array 2..0 of boolean;

  add0: add_bit(0, a_reg[0], b_reg[0], sum_reg[0], carry[0]);
  add1: add_bit(carry[0], a_reg[1], b_reg[1], sum_reg[1], carry[1]);
  add2: add_bit(carry[1], a_reg[2], b_reg[2], sum_reg[2], carry[2]);
  add3: add_bit(carry[2], a_reg[3], b_reg[3], sum_reg[3], carry_out);

-- these assertions  were used to incrementally verify parts of
-- the multiplier
  adding: assert G ((!carry[0] & a_reg[1] & b_reg[1]) -> X !sum_reg[1]);

  -- checked in main
}

-- A 4-bit subtractor, implemented by negating b and using a carry in of 1
MODULE subtractor(a_reg, b_reg, diff)
{

  INPUT a_reg, b_reg: array 3..0 of boolean;
  OUTPUT diff: array 3..0 of boolean;

  carry_out: boolean;
  carry: array 2..0 of boolean;

  sub0: add_bit(1, a_reg[0], !b_reg[0], diff[0], carry[0]);
  sub1: add_bit(carry[0], a_reg[1], !b_reg[1], diff[1], carry[1]);
  sub2: add_bit(carry[1], a_reg[2], !b_reg[2], diff[2], carry[2]);
  sub3: add_bit(carry[2], a_reg[3], !b_reg[3], diff[3], carry_out);


-- these assertions  were used to incrementally verify parts of
-- the multiplier
  difference: assert G (( a_reg = [1,1,1,1] & b_reg = [0,0,0,1]) ->
                        X diff = [1,1,1,0]);
}
```