Two Binary Algorithms for Calculating the Jacobi Symbol and a Fast Systolic Implementation in Hardware

George Purdy, Carla Purdy, and Kiran Vedantam
ECECS Department, University of Cincinnati, Cincinnati, OH, 45221-0030, USA
email: george.purdy@uc.edu,carla.purdy@uc.edu, vedantkk@email.uc.edu

***Abstract*--Efficiently computing the Jacobi symbol $J(a,b)$ for integers a and b is an important step in a number of cryptographic processes. We present two algorithms for computing $J(a,b)$ which can easily be implemented in hardware and which are efficient with respect to time and space. The first algorithm we describe is slower but also easier to implement in hardware than the second. The algorithms are systolic and thus each can be implemented as an array of identical cells. We have developed VHDL descriptions of these algorithms, and we provide here example code for the process statements which are central to the implementation of each algorithm. Each algorithm has been tested on an Altera Cyclone EP1C6Q240 device and simulated on an Altera Stratix-II EP2S15F484C3 device.**

I. Introduction

A. Motivation

The Jacobi symbol $J(a,b)$ defined for integer a and odd integer b has many applications in cryptography. Three that we can mention are the Solovay-Strassen primality test, the Goldwasser-Micali probabilistic public-key cryptosystem [1], and the standard algorithm for determining the number of points on an elliptic curve [2]. All of these would benefit from a fast hardware implementation of $J(a,b)$.

B. History

Algorithms for finding the Jacobi symbol $J(a,b)$ are closely related to GCD (greatest common divisor) algorithms. For example, the standard algorithm for calculating $J(a,b)$ [1] is essentially Euclid's algorithm with some rules added for sign changes. The algorithms for $J(a,b)$ discussed in this paper are also modified GCD algorithms. We call them *binary* algorithms because they are particularly well suited to binary operations--e.g., they never divide by anything but 2. The history of binary GCD algorithms begins with an algorithm by Josef Stein published in 1961 (see [3] for a thorough discussion). We published one [4]. One of the first algorithms to be implemented systolically was the binary GCD algorithm by Brent and Kung [5], who coined this use of the word *systolic*. There are some more binary algorithms for the Jacobi symbol that are more suitable for software implementations than hardware. See [6] and the references given there.

II. Definition of $J(a,b)$

If p is an odd prime, then we usually refer to $J(a,b)$ as the *Legendre symbol*, and it is defined as follows. We say that a is a quadratic residue or QR modulo p if there is a c not divisible by p such that $a \equiv c^2 \bmod p$, and we say that a is a quadratic non-residue or NR if there is no such c. If $p \mid a$ then a is neither a QR nor an NR. We define

$$J(a,p) = \begin{cases} +1 \text{ if } a \text{ is a QR} \\ 0 \text{ if } p \mid a \\ -1 \text{ if } a \text{ is NR} \end{cases}$$

Note that $J(a+p,p) = J(a,p)$, and $J(aa',p) = J(a,p)J(a',p)$. Let $b = p_1 p_{2.....} p_k$, where the p's are odd primes, not necessarily distinct. Then we define the Jacobi symbol $J(a,b)$ to be

$$J(a,b) = J(a,p_1)J(a,p_2).....J(a,p_k).$$

Note that if $GCD(a,b) > 1$ then $p_j \mid a$ for some j, and so $J(a,p_j) = 0$, and so $J(a,b) = 0$. Note also that

$$J(a+b,b) = J(a,b)$$
$$J(aa',b) = J(a,b), J(a',b) \text{ and}$$
$$J(a,bb') = J(a,b)J(a,b')$$

If $GCD(a,b) = 1$ then $J(a,b)$ is 1 or -1, and $J(a,b)$ is only defined when b is odd. All these facts about $J(a,b)$ can be found in [1].

A. Algorithm 1(a) for J(a,b) and Its Analysis

Consider Algorithm 1(a) to compute $J(a,b)$, given in Figure 1.We claim that (a,b) always converges to (1,1) which then remains fixed inside the *while (count < Limit)* loop and J stops changing. Thus the algorithm will work as long as Limit is large enough. That (1,1) is a fixed point is obvious.

For example,

$$a = (a + 3*b)/4 = 1 \text{ if } (a,b) = (1,1).$$

Also, if *(a,b) = (1, 1)*, the sign of $J(a,b)$ will not change. The convergence comes from the fact that the conditions $a \le 2^{\alpha}$ and $b \le 2^{\beta}$ are preserved each time through the loops. This is obvious except for the lines

```
if( (a+b)%4==0 )   { a=(a+b)/4;--alpha; }
else {a = (a+3*b)/4;}
```

Here, if $(a+b)\%4 = 0$, then

$a = (a+b)/4 \le 1/4(2^{\alpha} + 2^{\beta}) \le 1/4(2^{\alpha} + 2^{\alpha}) \le 2^{\alpha-1}$, whereas

if $(a+3b)\%4 = 0$, then

$a = 1/4(a+3b) \le 1/4(2^{\alpha} + 3*2^{\beta}) \le 2^{\alpha}$,

so that the condition is preserved in both cases. Now the variable count counts the arithmetic operations, and, assuming randomness, there is a 50% probability that $\alpha + \beta$ is decreased by one each time that count is incremented. Since $\alpha + \beta$ is initially $2n$, count will on the average reach $4n$ before the stable value $(a,b) = (1,1)$ is reached. A limit of $5n$ will therefore not be exceeded very often. (By Chebyshev's inequality the probability that $5n$ is not enough is less than $5/18n = 0.278/n$. If $n = 15$, this is about 0.019. In cryptographic applications, when n is around 1000, the probability of $5n$ not being enough would be less than 0.000278.) In practice however, things are much better. In an exhaustive search that we carried out of all a and all odd b less than 2^{15} the value of count = $4n$ was always enough. (The biggest value of count was 59.) With several sample runs using random 25-digit a's and b's, the results were even better. The stable $(1,1)$ was always reached with count very close to $2n$.

B. Identities for J(a,b)

To justify Algorithm 1(a), we need some identities involving $J(a,b)$. In what follows, GCD ($a.,b$) = 1, b is odd, and $a > 0, b > 1$. [1]

(1.1) $J(2,b) = (-1)^{(b^2-1)/8}$. This is needed for the step *a = a/2*.

It is not hard to derive from this the more useful formula:

(1.1a) $J(2,b) = -1$ if $b \bmod 8$ is 3 or 5 and +1 otherwise.

(1.2) $J(a,b) = J(b,a)(-1)^{(a-1)(b-1)/4}$. This is needed for the *swap(a,b)* command. Again, it is not hard to derive the more useful formula:

(1.2a) $J(a,b)J(b,a) = -1$ if a mod 4 and b mod 4 are both 3 and +1 otherwise.

(1.3) $J(4,b) = J(2,b)J(2,b) = +1$, so the steps of the form $a = a/4$ don't change J.

```
// b is odd, GCD(a,b) = 1, a ≤ 2^n and b ≤ 2^n.
int Limit = 5*n; int count = 0; int J = 1;
int alpha = n; int beta = n;
while(count < Limit)
{
  while(a even)                          //SHIFT
    {a=a/2; --alpha;
     if (b%8 ==3||b%8==5) J= -J;
     ++count;}

  if(alpha <=beta)                       //SWAP
    {swap(a,b); swap(alpha,beta);
    if (a%4==3 && b%4==3) J= -J;}

  if ((a+b)%4==0){a=(a+b)/4;--alpha;}    //NEWA
  else a = (a+3*b)/4; ++count;
}
//J now has the value J(a,b)
```

Figure 1. Algorithm 1(a) for computing *J(a,b)*.

C. Implementation Issues

In the actual implementation of Algorithm 1(a) it is simpler to maintain a variable $\delta = \alpha - \beta$, and eliminate α and β. Also, since α and β are approximately the base two logarithms of a and b, δ is relatively small and can be represented in unary notation. The resulting algorithm is given in Figure 2.

III. The Algorithm for Signed *a* and *b*

We have also implemented a second algorithm, Algorithm 2, based on the Brent-Kung GCD algorithm, which allows a and b to become negative. It has a rigorous time bound of $2n$. Algorithm 2 is given below in Figure 3.

A. Analysis of Algorithm 2

The instruction *if (a<0&&b<0) J=-J;* is difficult to implement systolically because the signs of a and b require complete carry resolution. Fortunately the operation $J = -J$ commutes with all other instances of $J = -J$, and so these can be performed at the end, when the sign bits of a and b become available. The conditions $|a| \le 2^{\alpha}$ and $|b| \le 2^{\beta}$ are maintained throughout the algorithm, and $\alpha + \beta$ is reduced by

one every time an arithmetic operation is performed, and so at most $2n$ operations are required. The computation ends when a becomes 0 so that the divide-by-2 loop keeps dividing 0 by 2 until α becomes 0. Assuming that the GCD of the original a and b is one, a can only become zero when (a,b) becomes (1,1), (1,-1), (-1,1), or (-1,-1). And by (1.6) below $J(a,b)=1$ for these values. Hence the correct value of $J(a,b)$ for the original (a,b) is computed when GCD (a,b) = 1.

```
// b is odd, GCD(a,b) = 1, a ≤ 2^n and b ≤ 2^n.
int Limit = 5*n;
int count = 0;
int J = 1;
int delta =0;
while(count<Limit)
{
  while(a even)                          //SHIFT

    {a=a/2; --delta;
     if(b%8 ==3||b%8==5) J=-J;
     ++count;}

  if(delta <=0)                          //SWAP
    {swap(a,b); delta = -delta;
     if(a%4==3 && b%4==3)J= -J;}

  if((a+b)%4==0){a=(a+b)/4;--delta;}     //NEWA
  else a = (a+3*b)/4;

  ++count;
}
// J now has the value J(a,b).
```

Figure 2. Algorithm 1(b) for computing *J(a,b)*.

B. Identities for Signed a and b

Algorithm 2 requires in place of (1.2) the signed version (1.4) and it also needs (1.5) to evaluate $J(-1,b)$. These can be found in [7].

(1.4) $J(a,b)J(b,a)=(-1)^{(a-1)(b-1)/4+(sign(a)-1)(sign(b)-1)/4}$, where sign(a) = 1 if $a>0$ and -1 if $a<0$.

Again, it is easy to derive from this a more useful form:

(1.4a) $J=1$;
if a mod 4 and b mod 4 are both 3 then $J=-J$;
if $a<0$ and $b<0$ then $J=-J$;
$J(a,b)J(b,a)=J$;

(1.5) $J(-1,b)=(-1)^{(b-1)/2+(sign(b)-1)/2}$. This is needed for the end of the algorithm, and one can derive the more useful form:

(1.5a) $J=1$;
if b mod 4 = 3 then $J=-J$;
if $b<0$ then $J=-J$;
$J(-1,b)=J$.

(1.6) It follows from (1.5a) that
$J(1,1)=J(1,-1)=J(-1,1)=J(-1,-1)=1$.

This is needed when the computation ends.

Strictly speaking, $J(a,b)$ is not normally defined when $a\in\{1,-1\}$, but we can extend the definition using (1.7) below.

For any b, $J(1,b)=J(1,b)J(1,b)=1$, and (1.5a) implies that $J(-1,1)=J(-1,-1)=1$. We can extend the definition of $J(a,b)$ as: if $b=1$, then $J(a,b)=J(a,p_1)J(a,p_2)....J(a,p_k)=1$, the empty product. Then (1.4a) and (1.5a) are consistent with each other, if $J(a,-1)=1\ \forall a$. Thus we define

(1.7) $J(a,1)=J(a,-1)=1\ \ \forall\ a$.

Notice that this allows us to keep the property that $J(a+b,b)=J(a,b)$.

```
// b is odd, GCD(a,b) = 1, a ≤ 2^n and b ≤ 2^n.
int J(long a, long b)
{
int alpha=n;
int beta=n;
int J=1;

while(true)
  {
    while(a even)                                  //SHIFT

      {a=a/2;--alpha;
       if(b mod 8=3||b mod 8=5) J= -J;
       if (alpha = 0) return J;
       }

    if(alpha<=beta)                                //SWAP

    {swap(a,b); swap(alpha,beta);
     if(a mod 4=3 & b mod 4=3) J= -J;
     if(a<0 && b<0) J= -J;
    }

    if((a+b) mod 4=0) a=(a+b)/4;else a=(a-b)/4;  //NEWA
    --alpha;
  }

// J now has the value J(a,b).
```

Figure 3. Algorithm 2 for computing *J(a,b)*.

IV. HARDWARE IMPLEMENTATIONS

We have implemented both Algorithm 1(b) and Algorithm 2 in VHDL, as arrays of identical cells. We have compiled and simulated the algorithms in the Altera Quartus II system for the target device Cyclone EP1C6Q240C8 and downloaded the resulting arrays onto the Altera UP3 prototyping board. Cyclone FPGAs are a low-cost device family, with logic elements (LEs) each consisting of a 4-input look-up table (LUT), one D flip-flop, and additional control and routing signals, arranged in larger logic array blocks (LABs), and with additional RAM blocks. The EP1C6Q240C8 has a total of 5980 LEs. We have also compiled and simulated the algorithms for a larger Altera device, the Stratix-II EP2S15F484C3, which contains a total of 12,480 ALUTs (adaptive look-up tables). Figure 4 shows a single cell for Algorithm 2. In this systolic implementation, a and b are pumped through the array, least significant bit first. After receiving the initial 3 least significant bits of each input (*a, b, start*, etc.), a cell determines which algorithm action it needs to perform, and thus which state the cell will be in while it processes the remaining data bits. A 1 in the least significant bit of the *start* data stream (s) indicates the position of the least significant bit for each data item. For the signed Algorithm 2, some cells will also need to know when the most significant bit arrives.

The main component of the VHDL code for each cell is a process which is sensitive to a clock event. Each cell functions as a finite state machine (FSM). When the start signal is received, the cell decides what its job is (e.g., SHIFT which divides a by 2, or SWAP a and b, or calculate a new a value, NEWA) and whether or not to change the sign of $J(a,b)$. This defines the control within a given cell. Thereafter it performs the correct computation on the remaining bits of a and b. The VHDL behavioral code for determining the cell's function is given in Figure 5.

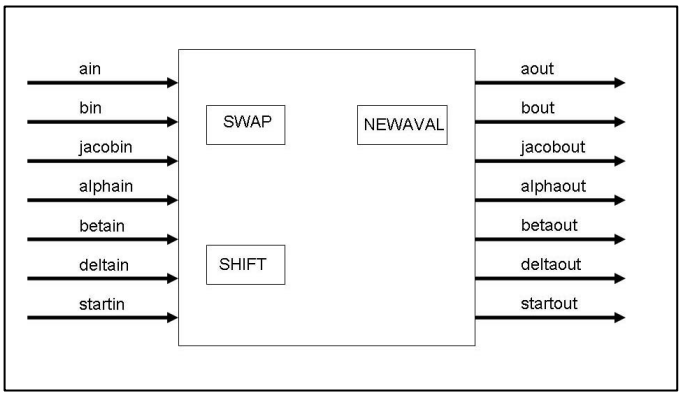

Figure 4. One cell in the systolic implementation of Algorithm 2. The values inside the boxes indicate state bits which are set once the first 3 bits of the numbers a and b are received. The remaining bits of a and b go through the same transformation in the cell.

The results for our implementations of Algorithm 1(b) and Algorithm 2 in Altera Cyclone EP1C6Q240 devices, the devices provided on the Altera UP-3 education boards, are given in Table I. Recall that for Algorithm 1(b) to process n-bit numbers, an array of 5n systolic cells is required. For Algorithm 2 to process n-bit numbers, an array of 2n systolic cells is required.

For cryptographic applications, we will often need to deal with large integers. Thus we also simulated Algorithm 2 on the largest available Altera device, the Stratix-II EP2S15F484C3 device. The results for this simulation are given in Table II.

For each algorithm, we give the number of LEs (or ALUTs) needed to implement one cell and also the number needed for the largest array of cells that will fit in one device. Note that the clock frequency decreases significantly as we move from one systolic cell to an array of cells. For our implementations and simulations, we relied on the Altera Quartus II automated design tools. It may be that custom design techniques would allow us to achieve higher clock rates.

V. CONCLUSIONS AND FUTURE WORK

We have provided unsigned and signed systolic algorithms for computing the Jacobi symbol $J(a,b)$ of integers a,b. Each algorithm is implemented in an array of identical cells, and each cell's state is set by 1-3 lsb's of the inputs. The unsigned version, Algorithm 1(b), requires $5n$ cells, and thus $J(a,b)$ is completed in time $k*5n$, where k is the processing time for one cell. The signed version, Algorithm 2, requires only $2n$ cells, but there is also a delay of at most n clock cycles for the final value of $J(a,b)$ to be set, due to the need to determine the signs of a and b. We have implemented our algorithms in Altera FPGAs. We are currently working on custom layouts for the cells, to obtain higher clock rates and thus greater processing efficiency.

REFERENCES

1. D.R. Stinson, *Cryptography Theory and Practice*, third edition, CRC Press, 2006.
2. N. Koblitz, *A Course In Number Theory and Cryptography*, Springer-Verlag, 1987.
3. D.E. Knuth, *Art of Computer Programming, Vol 2,* third edition, 1997.
4. G. Purdy, A carry-free algorithm for finding the greatest common divisor of two integers, *Computers and Math. with Applications*, 9 (2), pp. 311-316, 1983.
5. R.P. Brent and H.T. Kung, Systolic VLSI arrays for linear-time GCD computation, *Proc. VLSI '83*, pp. 145-154, 1983.
6. J. Shallit and J. Sorenson, A binary algorithm for the Jacobi symbol, *ACM SIGSAM Bulletin* 27 (1), 1993.
7. H. Hasse, *Number Theory*, Springer Verlag, 1970, p.86.

```
process (clk,reset)
begin
if ( reset = '1' ) then
      state <= INIT;
elsif (clk'event and clk = '1' ) then
      case state is
            when INIT=>
                  if ( stopin = '1' ) then
                        state <= STOP;
                  elsif ((start(2) and alpha(2)) = '1' )
                        state <= STOP;
                  elsif ( start(2) = '1' ) then
                        if (a(2) = '0' ) then
                              state <= SHIFT;
                        elsif (sdelta(2) = '1') then
                              state <= SWAP;
                        else
                              state <= NEWA;
                        end if;
                  else
                        state <= INIT;
                  end if;
            when SHIFT=>
                  if (start(2) = '1' ) then
                        state <= INIT;
                  else
                        state <= SHIFT;
                  end if;
            when SWAP=>
                   if (start(2) = '1' ) then
                        state <= INIT;
                  else
                        state <= SWAP;
                  end if;
            when NEWA=>
                  if (start(2) = '1' ) then
                        state <= INIT;
                  else
                        state <= NEWA;
                  end if;
            when STOP=>
                  state <= STOP;
            when others=>
                  state <= INIT;
            end case;
      end if;
end process;
```

Figure 5. VHDL code for the process to put each cell in the correct computational state for Algorithm 2.

TABLE I. IMPLEMENTING ALGORITHMS 1(b) & 2 IN AN ALTERA CYCLONE EP1C6Q240 DEVICE (5980 LES).

Alg.	LEs	% Dev. Used	n (Bits per Dev.)	Devices per 1000 Bits	Clk Freq. (MHz)
1(b): 1 Cell	35	<1%	--	--	312.08
1(b): Array	5862	98% (210 Cells)	42	~24	90.00
2: 1 Cell	46	<1%	--	--	275.03
2: Array	5861	98% (140 Cells)	70	~14	88.3

TABLE II. IMPLEMENTING ALGORITHM 2 IN AN ALTERA STRATIX-II EP2S15F484C3 DEVICE (12,480 ALUTS).

	ALUTs	% Dev. Used	n (Bits per Dev.)	Devices Per 1000 Bits	Clk Freq. (MHz)
1 Cell	44	<1%	--	--	487.33
Array	12,121	97% (340 Cells)	170	~6	127.81