A Real-Time Systolic Integer Multiplier

Guy Even *
Technion

November 15, 1993

Abstract

We describe the construction of two practical real-time systolic bit-serial multipliers: a pipelined multiplier and a non-pipelined multiplier. Our non-pipelined multiplier requires roughly half the hardware (number of full-adders) as compared to Atrubin’s multiplier, which is the best real-time systolic multiplier known to-date.

Our starting point is a serial/parallel multiplier which is not systolic, has a large number of primary inputs and does not support pipelining. We modify this design step by step using systematic methodologies until our design is obtained. The methodologies used in the design are: broadcast elimination, replacing parallel inputs with a broadcast mechanism and adding a simplified version of the circuit in order to enable pipelining. Applying these methodologies yields an improved circuit the correctness of which is easily understood.

*e-mail: guy@cs.technion.ac.il
1 Introduction

The ability to produce inexpensive special-purpose complex VLSI chips raises the question of how to design VLSI algorithms which utilize the capabilities this technology. One of the methods suggested for solving this problem is to design systolic arrays which satisfy the following design goals: Fast clocks - the circuits can operate with very fast clock rates. Modularity - the circuits are constructed from a small set of cells. Scalability - the designs are easily modified for longer input lengths. Small number of primary inputs/outputs - facilitates scalability and routing. Simple layouts - facilitates designing the VLSI realization of the circuit.

However, the design of systolic arrays is complicated and the correctness of these circuits is difficult to understand. Retiming is a design methodology which was proposed by Leiserson and Saxe [10] for simplifying and improving the design of systolic arrays. Retiming has been a very successful technique in improving and simplifying previous designs. Leiserson and Saxe [10] described a simple circuit which recognizes palindromes; it is much easier to understand, compared with previous treatments (such as Cole’s [2] and Sieferas’ [12]). Even and Litman [7] provided a proof of Atrubin’s multiplier 29 years after it was invented [1]. This paper exemplifies that using design methodologies, such as retiming, leads to the design of better circuits the correctness of which is easily understood.

We describe the construction of two practical, real-time, bit-serial, systolic integer multipliers. One of the multipliers is pipelined and the other is not. Our non-pipelined multiplier requires roughly half the hardware as compared to Atrubin’s multiplier. Our multipliers receive as input two n-bit multiplicands, and output serially the 2n bits of the product. One of the inputs, a, is input one bit at a time, and the other multiplicand b, is input two bits at a time. The ith bit of the product is output one clock cycle after the ith bit of multiplicand a is input. Our multipliers are constructed from a linear array of identical cells, which do not depend on the length of the multiplicands. The number of cells required to multiply n-bit integers is n/4, and therefore, our circuits require linear area. Wires connect only neighboring cells, and all wires have at least one register. Therefore, the clock period depends only the cells of the linear array, and do not depend on the length of the multiplicands.

The starting point of our design is a non-systolic multiplier known as a serial/parallel multiplier [13, 3]. We improve this circuit step by step using systematic methodologies until our design is obtained. The methodologies used are: broadcast elimination using retiming, replacing parallel inputs with broadcast and adding a simplified version of the circuit in order to enable pipelining. Our method of replacing parallel inputs, which input constant values, with broadcast is new, and can be used for decreasing the number of primary inputs in other circuits as well. Our method of enabling pipelining increases the throughput by a factor of two, yet the increase in the hardware is by a factor less than two. The same method can be applied to Atrubin’s multiplier.

The organization of this paper is as follows: In section 2 we present the serial/parallel multiplier and discuss its disadvantages. In section 3 we show how to eliminate broadcast using retiming. In section 4 we show how to replace parallel inputs with broadcast. In section 5 we present our non-pipelined multiplier. In section 6 we present a pipelined version of the serial/parallel multiplier and discuss how our pipelined multiplier is obtained. In section 7
we summarize the comparison of our multiplier and Atrubin's multiplier.

2 Serial/Parallel Multiplier

In this section we present a multiplication circuit known as a serial/parallel multiplier [13, 3]. This multiplier is a basic design from which we finally obtain a systolic multiplier. The serial/parallel multiplier is a synchronous circuit consisting of a linear array of identical combinational elements and of registers. The serial/parallel multiplier is designed to multiply two $n$-bit positive integers represented by the binary vectors $a = a_{n-1} \ldots a_0$ and $b = b_{n-1} \ldots b_0$. A serial/parallel multiplier of 4-bit integers is depicted in figure 1. The boxes denote combinational units called summand-adders the structure and functionality of which is depicted in figure 2. The short segments along the edges denote clock controlled registers which output at each clock cycle the value input to them in the previous clock cycle.

The circuit has $n$ primary inputs and one primary output. The bits of the multiplicand $a$ are input one bit per clock cycle, least significant bit first, through the primary input $a_{in}$. After the bits of $a$ have been input, zeros are input through $a_{in}$ for $n$ clock cycles. The bits of the multiplicand $b$ are input in parallel; the $i$'th bit, $b_i$, is input through the primary input $b_i$. The primary input $\text{Start}$ is used to mark the beginning of computations: A ‘1’ is input by $\text{Start}$ simultaneously with the least significant bit of $a$. During all other clock cycles a ‘0’ is input by $\text{Start}$. The product is output through the primary output $p_{out}$. The bits of the product are output one bit per clock cycle, least significant bit first, starting from one clock cycle after the computation begins. Since the product is $2n$-bits long, it is necessary to wait $n$ clock cycles after the most significant bit of $a$ has been input before a new computation can be started.

The serial/parallel multiplier is a bit-serial, real-time modular design which can be easily implemented in VLSI. It requires $n$ summand-adders for multiplying $n$-bit positive integers. However it has three disadvantages:

1. The bits of the operand $a$ and the $\text{Start}$ signal are sent along registerless paths to all the summand-adders in the circuit. The disadvantage of this feature, called ‘broadcast’ in [10, 8], is that it sets a lower bound on the clock period which is linear with $n$. 

![Figure 1: Serial/parallel multiplier](image-url)
if $Start = 0$ then
  begin
  $sum_{out} := \text{XOR}(a_j \cdot b_i, sum_{in}, carry_{in})$
  $carry_{out} := \begin{cases} 
  0 & \text{if } a_j \cdot b_i + sum_{in} + carry_{in} < 2 \\
  1 & \text{otherwise}
  \end{cases}$
  end
else
  begin
  $sum_{out} := a_j \cdot b_i$
  $carry_{out} := 0$
  end

Figure 2: Summand-adder: inputs/outputs and functionality

2. The bits of the operand $b$ are input in parallel to all the summand-adders in the circuit. The disadvantage of this feature is that the number of primary inputs required for the multiplication circuit is linear with $n$.

3. The elapsed time between successive computations is $2n$ clock cycles, although the inputs are $n$ bits long. In section 6 we show a pipelined version of the serial/parallel multiplier in which the elapsed time between computations is $n$ clock cycles that requires less than twice the hardware of a single non-pipelined serial/parallel multiplier.

3 Eliminating The Broadcast By Retiming

In this section we show how to eliminate the broadcast from the serial/parallel multiplier. The technique we use for eliminating the broadcast is called retiming and has been used in previous works [10, 7]. We follow the technique of Even and Litman [7, 8] which fully preserves the functionality of the circuit. However, one can also use the technique of [10].

Consider a circuit composed of combinational elements and of registers, in which wires always connect two units, for example: a combinational element with a register (or vice-versa), a register to a register. (We do not consider circuits with buses, etc.). For the purpose of retiming, one is interested in the topology of the circuit rather than in the functionality of the circuit. In [10], a synchronous circuit is abstracted by a directed graph as follows: The vertices of the graph correspond to combinational elements of the circuit and to primary inputs and primary outputs of the circuit, which are referred to as input/output hosts. We connect two vertices of the graph with an edge whenever there is a wire connecting the corresponding elements in the circuit. We label each edge, $e$, with a non-negative integer, $w(e)$, which equals the number of registers on the corresponding wire. The labeled directed graph describing the topology of the circuit is called the communication graph of the circuit.

Retiming is a syntactic transformation which depends only on the communication graph of the circuit. The communication graph of the serial/parallel multiplier is depicted in figure
3. An edge in the communication graph can have a capacity larger than one bit per clock cycle. Therefore, the nets carrying the bits of $a$ and the $Start$ signal are depicted by one edge instead of two. Self-loops are omitted from figure 3 because they are not modified by retiming. Primary inputs and outputs are depicted by ellipse shaped vertices. The summand adders are denoted by the top row of the circle shaped vertices, whereas the vertices in the bottom row are simple junctions, which output through both outputs the value currently input to them.

Following [7] we demonstrate how to eliminate the broadcast mechanism from the communication graph of the serial/parallel on a communication graph called *linear array with broadcast* depicted in figure 4. The communication graph of the serial/parallel multiplier is a subgraph of linear array with broadcast.

![Figure 3: Communication graph of serial/parallel multiplier](image1)

![Figure 4: A linear array with broadcast](image2)

We retime the circuit by determining the lag of each vertex. The lag of a vertex is an integer which describes the number of clock cycles the vertex will lag in the retimed graph relatively to the original graph. Let $\text{lag}(v)$ denote the lag of the vertex $v$. We denote by $w'(e)$ the number of registers on the edge $e$ in the retimed communication graph. Let $e$ be an edge $(v \rightarrow u)$, set $w'(e)$ to be $w'(e) := w(e) + \text{lag}(u) - \text{lag}(v)$. The communication graph with the new labels, $w'(e)$, is the retimed graph. Note, that the original communication graph and the retimed one differ only in the weights of the edges.

The important property of retiming is that, under certain restrictions, the retimed circuit is equivalent to the original one. This equivalence refers to the input/output behavior of the
circuit from the point of view of the hosts. The restrictions on this equivalence do not affect our discussion; more details on retiming can be found in [10, 7, 5].

Figure 5 depicts a retiming of the linear array with broadcast; the lag of each vertex is written in the vertex, and the edges contain the revised number of registers. The retiming does not turn the circuit into a systolic one. A systolic circuit is obtained by joining elements together. If one joins together old elements into new elements, and if one only considers edges between new elements, then this retiming results in a communication graph in which every edge has at least one register. Furthermore, all the new elements in the ‘joined’ circuit are identical. The joined elements are shown in figure 5 by dotted boxes.

![Figure 5: Retimed linear array](image)

Figure 5: Retimed linear array

Figure 6 depicts the serial/parallel multiplier after eliminating the broadcast. The retimed circuit does not have a broadcast mechanism but still has the disadvantage of too many primary inputs. We deal with this in the next section.

![Figure 6: Retimed circuit - broadcast of multiplicand a and of Start signal eliminated](image)

Figure 6: Retimed circuit - broadcast of multiplicand a and of Start signal eliminated

4 Replacing Parallel Inputs With Broadcast

In this section we show how one can replace the parallel inputs of the retimed linear array with a broadcast mechanism. This change involves a modification of the topology of the
circuit and it also involves minor adjustments of the functionality of the circuit. This new transformation can be helpful when one wishes to eliminate parallel inputs, which input constant values, from a linear array. We apply this replacement to the retimed parallel/serial multiplier depicted in figure 6, and obtain a circuit which has only four primary inputs but which has a broadcast mechanism. The broadcast can then be eliminated by using the transformation presented in the previous section.

Consider the retimed linear array depicted in figure 5. Figure 7 depicts the communication graph of the retimed linear array; the joined elements are shown as boxes, and the hosts as ellipses. Suppose that the top row of input hosts, named \( H_0, H_1, \ldots \), input constant values throughout the computation (which lasts \( 2n \) clock cycles). We show how to replace these input hosts with two hosts and two broadcast nets. Let \( b_i \) denote the value input by \( H_i \) during the computation. Notice, that due to the increasing amount of registers along the edges which emanate from the hosts \( H_0, H_1, \ldots \), these hosts input at most two new values to the linear array at every clock cycle. In figure 8, we replace these hosts with two hosts: \( H_{odd} \) and \( H_{even} \), which are connected to two broadcast nets. The values \( b_0, b_2, \ldots \) are input by host \( H_{even} \) starting from the first clock cycle of the computation, and the values \( b_1, b_3, \ldots \) are input by the host \( H_{odd} \) starting from the second clock cycle of the computation. Each host inputs one value at a time, and therefore, these input values reach their destinations in the correct clock cycle.

To insure proper functioning of the array, we slightly modify the circuit. The reason for this is that the functional elements may sample the values input by the parallel inputs.
during the whole computation rather than only during the first time they engage in the computation. We overcome this problem by using an initialization signal and a register for each input (which used to be a parallel input): The initialization signal notifies the functional element when the value input is valid, and at that clock cycle the input is sampled and stored by the register for future clock cycles. It is convenient to assume that the original linear array with broadcast had an initialization signal, which was broadcast simultaneously to all the linear array. This initialization signal is no longer broadcast in the retimed circuit, but it arrives at each element “at the right time”.

We apply this replacement to the serial/parallel multiplier of figure 6, and obtain a circuit which has only four primary inputs. The multiplier circuit has already incorporated an initialization signal, \textit{Start}, so we only have to add to each summand-adder a register which holds the sampled input. The modified summand-adder with the register for holding the sampled value of the input $b_i$ is described in appendix A. We hereafter refer to the modified summand-adder as the summand-adder.

5 A Non-Pipelined Version of Our Multiplier

The transformation described in section 4 reduced the number of primary inputs to a constant number, regardless of the length of the numbers multiplied. However, this transformation replaces parallel inputs with a broadcast mechanism. Our final circuit is obtained by eliminating the broadcast nets using the same transformation described in section 3.

Figure 9 depicts a systolic, real-time, bit-serial multiplier, which is obtained by retiming the multiplier obtained in the previous section. The filled boxes denote summand adders, the dotted boxes which contain two summand adders denote the “joined-elements” of figure 6, and the dotted boxes which contain four summand adders denote the new cells one obtains after the last retiming.

![Figure 9: A non-pipelined version of our multiplier](image-url)
6 A Pipelined Version of Our Multiplier

The non-pipelined version of our multiplier deals only with one computation at a time. This restriction forces one to wait idle for \( n \) clock cycles before starting to feed two new multiplicands. This characteristic may seem to be natural due to the fact that the product is \( 2n \) bits long. However, in single precision multiplications, one is only interested in the \( n \) most significant bits of the product. Even though the output is only \( n \) bits long in such cases, the throughput is one computation every \( 2n \) clock cycles.

In this section we show a pipelined version of the serial/parallel multiplier which can handle two computations simultaneously. This multiplier outputs two bits per clock cycle; one bit belongs to the \( n \) most-significant bits of the product of the previous inputs, and one bit belongs to the \( n \) least-significant bits of the product of the current inputs. The gain obtained by this modification is that the elapsed time between the start of two consecutive multiplication operations is \( n \) clock cycles instead of \( 2n \) clock cycles. The pipelined multiplier has, therefore, a throughput rate which is twice as large as the throughput of the non-pipelined multiplier. However, the amount of hardware needed for implementing the pipelined multiplier is less than the hardware needed for implementing two non-pipelined multipliers. The transformations described in sections 3-9 can also be applied to the pipelined serial/parallel multiplier, yielding a pipelined, systolic, real-time, bit-serial multiplier.

Consider the serial/parallel multiplier. Note, that zeros are input through \( a_{in} \) during the last \( n \) clock cycles of each computation, causing the summand adders to function as half-adders: they add the carry bit, \( carry_{in} \), with the sum bit, \( sum_{in} \). Let us add a new array of half-adders which is capable of simulating the computation performed by the serial/parallel multiplier during the last \( n \) clock cycles of each computation. Both arrays (the summand-adders’ array and the half-adders’ array) are utilized according to the following schedule: Every \( n \) clock cycles, new multiplicands are input in concert with \( Start \) rising to ‘1’. The values stored in the summand-adders’ array are passed to the half-adders’ array which finishes computing the previous product. This frees the summand-adders’ array which starts computing the current product.

Figure 10 depicts the pipelined serial/parallel multiplier for 4-bit integers. Note, that the bits of multiplicand \( a \) and the \( Start \) signal are broadcast to all the summand-adders, whereas only the \( Start \) signal is broadcast to the half-adders. The \( n \) least-significant bits of each product are output by the summand-adders’ array (the top output), and the \( n \) most-significant bits of the product are output by the half-adders’ array (the bottom output).

The input/output ports of a half-adder are named and shown in figure 11 along with the functionality. The value of the \( carry_{val} \) input equals the value of the \( carry_{in} \) input of the summand-adder, and the value of the \( sum_{val} \) input equals the \( sum_{in} \) input of the summand-adder. The summand-adders of the pipelined serial/parallel multiplier are identical to the summand-adders of the serial/parallel multiplier, except for an increased fanout of the \( carry_{in} \) and \( sum_{in} \) signals.

If one joins together the \( i \)’th summand-adder and the \( i \)’th half-adder into a single element, then one obtains a circuit the communication graph of which equals the communication graph of the serial/parallel multiplier. It is, therefore, straightforward to apply the same trans-
formations to the pipelined serial/parallel multiplier in order to obtain a systolic pipelined multiplier.

![Diagram of a 4-bit pipelined serial/parallel multiplier](image)

**Figure 10:** 4-bit pipelined serial/parallel multiplier

```plaintext
if Start = 0 then
  begin
    sum_out := XOR(sum_in, carry_in)
    carry_out := AND(sum_in, carry_in)
  end
else
  begin
    sum_out := XOR(sum_val, carry_val)
    carry_out := AND(sum_val, carry_val)
  end
```

**Figure 11:** Half-adder: inputs/outputs and functionality

7 **Summary and Discussion**

In table 1 we compare the number of hardware elements needed for realizing our multiplier and Atrubin’s multiplier using the realizations suggested in appendix B. We count the number of hardware elements needed for every 4 bits of the multiplicands’ length. This comparison shows that our non-pipelined multiplier requires roughly half the hardware as compared to Atrubin’s multiplier.

The speed of a synchronous circuit (the fastest clock it can operate with) is determined by the longest path of combinational elements a signal has to propagate through during a
Table 1: Hardware Elements Needed For Every 4 Bits

<table>
<thead>
<tr>
<th>Hardware Element</th>
<th>Our Multiplier</th>
<th>Atrubin’s Multiplier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full Adder</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>D-Latch</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>AND gates</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>3-input AND gates</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>Delay Elements</td>
<td>19</td>
<td>28</td>
</tr>
</tbody>
</table>

single clock cycle. In the suggested realizations both multipliers have critical paths of similar length: In our multiplier the longest path goes through 4 AND gates and 4 full-adders. In Atrubin’s multiplier, the longest path goes through 2 AND gates (each with 3 inputs) and through 4 full-adders. Although our multiplier requires less hardware than Atrubin’s, the fact that the multiplicands are input at different rates might complicate integrating it with other circuits.

In section 6 we introduced a technique for transforming a non-pipelined multiplier into a pipelined one. This technique can also be applied to Atrubin’s multiplier, and thus obtaining a pipelined version of Atrubin’s multiplier.

The circuits introduced in this paper are particularly suited for throughput oriented environments which operate with very fast clock rates. Cryptographic uses are examples of such environments in which very long integers are multiplied, therefore, rendering multipliers with quadratic area impractical. For example, the modular multiplier described in [6] includes an integer multiplier, and Atrubin’s multiplier was suggested for that purpose.

Acknowledgements

I would like to thank my father and Ami Litman for the numerous conversations about systolic arrays, retimings and Atrubin’s multiplier. I would also like to thank them for their remarks which helped me improve the presentation.

References


Appendix

A The Modified Summand-Adder

The modified summand-adder with the register for holding the sampled value of the input $b_i$ is illustrated and described in figure 12.

\[
\begin{align*}
\text{if } \text{Start} = 0 \text{ then} & \quad \text{begin} \\
& \quad \text{sum}_{out} := \text{XOR}(a_j \cdot b_{in}, \text{sum}_{in}, \text{carry}_{in}) \\
& \quad \text{carry}_{out} := \begin{cases} 
0 & \text{if } a_j \cdot b_{in} + \text{sum}_{in} + \text{carry}_{in} < 2 \\
1 & \text{otherwise} 
\end{cases} \\
& \quad \text{b}_{out} := \text{b}_{in} \\
& \quad \text{end} \\
\text{else} & \quad \text{begin} \\
& \quad \text{sum}_{out} := a_j \cdot b_i \\
& \quad \text{carry}_{out} := 0 \\
& \quad \text{b}_{out} := b_i \\
& \quad \text{end}
\end{align*}
\]

Figure 12: Modified summand-adder: inputs/outputs and functionality

B Gate Realizations of our Multiplier and Atrubin’s Multiplier

We claim that our multiplier requires roughly half the amount of hardware required by Atrubin’s multiplier. One cannot give an exact comparison, since the amount of hardware required depends on the technology used and on the various decisions which lead to specific designs. In this section we suggest gate level realizations of both multipliers, using similar gates and techniques. We believe that this example supports our claim, and that it shows that our multiplier does not have any ‘hidden’ properties which might complicate its realization. Moreover, it is our belief that the ratios of the gate count in our multiplier and in Atrubin’s multiplier given in this section will carry on to other technologies and designs. This belief stems from the similarity between the functionality of the “basic-cells” of these multipliers.

B.1 Symbols and Notation

In this subsection we describe the hardware components which are the building blocks of the gate-level realizations given in the next subsections. Table 2 gives a description of the gates and memory components we use in these realizations.
B.2 A Gate Level Realization of our Multiplier

In this subsection we suggest a realization of our non-pipelined multiplier. A schematic of the summand adder from figures 12 is shown in figure 13.

![Figure 13: Realization of summand-adder](image)

According to figure 9, one needs 19 delay-elements and 4 summand-adders for every 4 bits of the multiplicands’ length. (We do not assign delay-elements for storing the bits of because we use D-latches for storing them).

B.3 A Gate Level Realization of Atrubin’s Multiplier

A description and an explanation of Atrubin’s multiplier appears in [8]. Therefore, we limit the discussion in this subsection to a suggested hardware realization of Atrubin’s multiplier.

Atrubin’s multiplier can be obtained by retiming a linear array with broadcast. In figure 14 we depict Atrubin’s multiplier after the retiming. The dotted boxes denote the cells of the multiplier after joining together the basic cells, which are denoted by filled boxes. In figure 15 we name the input/output ports of the basic cells. In Atrubin’s multiplier, multiplication of \( n \) bit integers requires \( n \) basic cells. A suggested realization of the basic cell is shown in figure 16.

The dotted boxes in figure 14 are constructed from 2 basic cells and 8 delay-elements (note, that the lines for \( a_i \) and \( b_i \) are two bits wide). The suggested realization of the basic cells in Atrubin’s multiplier shows that each dotted box can be constructed from: \( 2 \times 2 = 4 \) full adders, \( 2 \times 2 = 4 \) D latches, \( 2 \times 3 = 6 \) AND gates and \( 2 \times 2 = 4 \) AND gates with 3-inputs and \( 2 \times 3 + 8 = 14 \) delay elements (3 per basic cell, and 8 for lines in the dotted box). (It is possible to reduce the number of delay-elements in each basic-cell to 2 by deleting the delay-element used to delay \( Init \), and by taking an output of a delay-element which delays the outgoing \( Init \) signal).
Figure 14: Atrubin’s multiplier

2 carry bits, sampled $a_i$ and $b_i$

Figure 15: Ports of a cell in Atrubin’s multiplier

Figure 16: Realization of the basic cell of Atrubin’s multiplier
<table>
<thead>
<tr>
<th>Name</th>
<th>Symbol</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full Adder</td>
<td><img src="image" alt="Diagram of Full Adder" /></td>
<td>$S := \text{XOR of inputs}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$C := \begin{cases} 0 &amp; \text{if sum of inputs} &lt; 2 \ 1 &amp; \text{otherwise} \end{cases}$</td>
</tr>
<tr>
<td>AND Gate</td>
<td><img src="image" alt="Diagram of AND Gate" /></td>
<td></td>
</tr>
<tr>
<td>Delay Element</td>
<td><img src="image" alt="Diagram of Delay Element" /></td>
<td>$out(t + 1) := in(t)$</td>
</tr>
<tr>
<td>D latch</td>
<td><img src="image" alt="Diagram of D latch" /></td>
<td>$Q := \text{last value}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{input in D while the C input was high.}$</td>
</tr>
</tbody>
</table>

Table 2: Gates and memory components