Chapter 7
Arithmetic and Logic

The whole of arithmetic now appeared within the grasp of mechanism.

– C. Babbage

Abstract So far we have seen how to represent numbers as binary vectors and how, using transistors, one can implement logic gates that perform the basic Boolean operations NOT, AND, and OR. In order to realise the goal of performing useful computation on integer values represented as n-bit vectors, we need to compose these logic gates into higher-level circuits. Generally speaking, the **Arithmetic and Logic Unit (ALU)** packages circuits for different operations into one component that performs computation for the processor. To aid the discussion in examples we set $n = 8$ throughout, i.e., set the size of a word to equal one byte. However, it is important to see that the methods are presented in a general form: one can easily extend them to cope with larger word sizes.

7.1 Introduction

The ALU is at the heart of any processor; when one requires an addition or comparison to happen, *it* is the component that performs the computation under the guidance of a control unit. This design was first formalised by John von Neumann in 1945 in his pioneering design for the EDVAC computer. Many principles outlined in his design are still retained today. Specifically he noted that any computer would have to perform basic mathematical operations on numbers and that it is "reasonable that [the computer] should contain specialised organs for these operations" [48]. The modern ALU is an example of such an organ.

In a very rough sense one can view the ALU as a primitive calculator: it accepts operands as input, waits to be told what operation to perform, and then produces results as output after some time. As such, one can view the high-level architecture of a basic ALU as being similar to Figure 7.1. In this diagram, there are a number of units within the ALU which all take inputs and generate results, potentially in

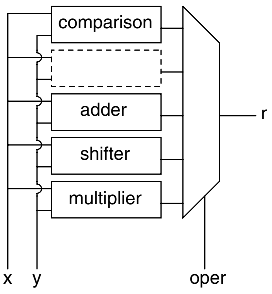


Figure 7.1 A high-level architectural design for a basic ALU.

parallel, for their associated operations. For example, we might have one unit for addition, one unit for multiplication and one unit for shift operations. These results are selected between by a signal which is determined by the control-path; depending on which operator it uses, a different result is produced on the ALU output. In reality it might be advantageous to turn off units which are not being used at any one time so as to reduce power consumption or heat. Additionally it is very likely that replicated logic in different units could be shared; for example, the unit for addition might also be used in multiplication. We will not worry about these sorts of optimisation here. Finally note that at this high-level, one usually views the ALU as a single component. However, at a lower-level, it might be attractive to split it into parts. For example, it is common to have dedicated floating point arithmetic within a **Floating Point Unit (FPU)**, or to split the integer ALU into two parts for arithmetic and comparisons. In this way, one can perform different types of computation at the same time; for example, we might compare two numbers and add another two numbers at the same time.

7.2 Comparisons

Recall from Chapter 2 that we can easily construct functions that describe 1-bit equality and less than comparisons; we use $equ(x,y)$ to denote the truth value related to the test $x = y$, and $lth(x,y)$ to denote the truth value related to $x < y$ for 1-bit values x and y. The basic idea in constructing comparators for larger values, n-bit integers in our case, is to combine many of these 1-bit comparators together in order to produce the final result.

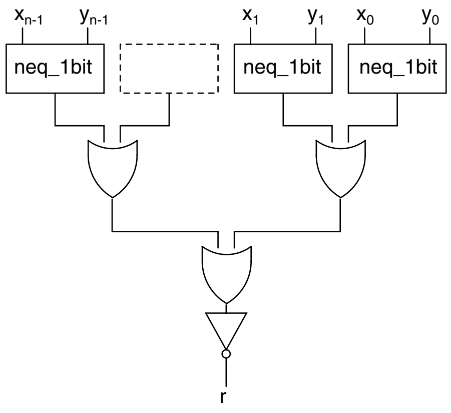


Figure 7.2 Design for an unsigned equality comparator.

```
module equ_nbit( r, x, y );

  parameter n = 8;

  output wire         r;
  input  wire [n-1:0] x;
  input  wire [n-1:0] y;

         wire [n-1:0] w0 = x ^ y;

  assign r = ~( |w0 );

endmodule
```

Listing 7.1 Implementation of an n-bit unsigned equality comparator.

7.2.1 Unsigned Comparisons

7.2.1.1 Equality

Implementing a hardware circuit to test values for equality is a simple matter of using the 1-bit equality test in a component-wise manner on the inputs and then ensuring all the 1-bit comparisons are true. This final step can be implemented by ANDing all the 1-bit comparisons together; the whole values are unequal if even one of the 1-bit comparisons fail, so all must be true for the comparison to be true. For efficiency, one can arrange this operation as a tree to minimise the critical path.

However, we can be slightly more clever than this naive approach and reduce the number of required gates somewhat. Instead of 1-bit equality modules we use 1-bit not-equal modules, which are essentially just XOR gates; this change saves n NOT gates. By ORing these results together, instead of ANDing, and then negating the result we compute the same function: we have just rearranged the expression a little.

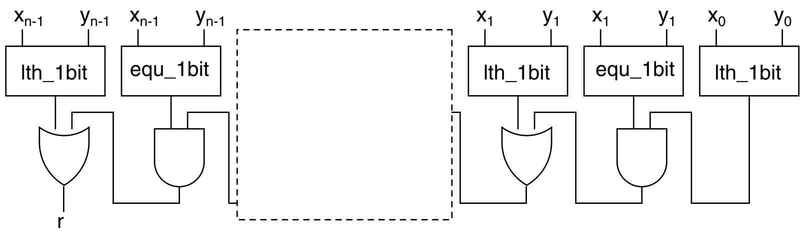


Figure 7.3 Design for an unsigned less than comparator.

```
module lth_nbit( r, x, y );

  parameter n = 8;

  output wire         r;
  input  wire [n-1:0] x;
  input  wire [n-1:0] y;

         wire [n-1:0] w0 = ~(  x ^ y );
         wire [n-1:0] w1 =  ( ~x & y );
         wire [n-1:0] w2;

  assign w2[0] = w1[  0];
  assign r     = w2[n-1];

  genvar i;
  generate
  begin
    for( i = 1; i < n; i = i + 1 )
    begin:gen_lth
      assign w2[i] = w1[i] | ( w0[i] & w2[i-1] );
    end
  end
  endgenerate

endmodule
```

Listing 7.2 Implementation of an n-bit unsigned less than comparator.

The end result is shown in Figure 7.2 and implemented in Listing 7.1, the overall saving is $n-1$ NOT gates. Note that the implementation uses a reduction operator: the assumption is that a Verilog tool-chain will be clever enough to translate this into a tree structure.

7.2.1.2 Less Than

To test if one n-bit value x is less than another one called y, we compare the i-th bits of x and y with each other starting from the most-significant and find that $x < y$ if either of the following is true:

1. $lth(x_i, y_i)$.
2. $equ(x_i, y_i)$ and taking x' and y' to be the $(i-1)$-th down to the 0-th bits of x and y respectively, we recursively find that $x' < y'$.

This is somewhat of an awkward description; more concretely we can write the operation on 8-bit values as the sequence

$$\begin{aligned}
t_0 &= lth(x_0, y_0) \\
&\vdots \\
t_5 &= lth(x_5, y_5) \lor (equ(x_5, y_5) \land t_4) \\
t_6 &= lth(x_6, y_6) \lor (equ(x_6, y_6) \land t_5) \\
t_7 &= lth(x_7, y_7) \lor (equ(x_7, y_7) \land t_6)
\end{aligned}$$

with the final result appearing as t_7. Consider an even smaller example where we set $x = 6_{(10)} = 110_{(2)}$ and $y = 7_{(10)} = 111_{(2)}$ such that $n = 3$. That is, we want to test if $110_{(2)} < 111_{(2)}$. Examining each bit of x and y we can clearly see that

$$\begin{aligned}
lth(x_0, y_0) &= \text{true} \\
lth(x_1, y_1) &= \text{false} \\
lth(x_2, y_2) &= \text{false}
\end{aligned}$$

and also that

$$\begin{aligned}
t_0 &= \text{true} \\
t_1 &= \text{true} \\
t_2 &= \text{true}
\end{aligned}$$

because

$$\begin{aligned}
t_0 &= lth(x_0, y_0) \\
&= \text{true} \\
\\
t_1 &= lth(x_1, y_1) \lor (equ(x_1, y_1) \land t_0) \\
&= \text{false} \lor \text{true} \\
&= \text{true} \\
\\
t_2 &= lth(x_2, y_2) \lor (equ(x_2, y_2) \land t_1) \\
&= \text{false} \lor \text{true} \\
&= \text{true}
\end{aligned}$$

Thus we find that $x < y$, or rather than $6_{(10)} < 7_{(10)}$, because the final output in the sequence is $t_2 = \text{true}$. Hopefully it is clear that we can chain together instances of our 1-bit comparator modules in order to implement the sequence above; this approach is shown in Figure 7.3 and implemented in Listing 7.2. Wires `w0` and `w1` implement the 1-bit equality and less than tests respectively; the generate statement chains these together as described above to produce the result `r`.

7.2.2 Signed Comparisons

A signed equality comparison is the same as an unsigned equality comparison since we are essentially just testing corresponding bits against each other. Signed less than comparison is a little more tricky however. We can formulate a set of rules for how signed less than behaves depending on the signs of the inputs x and y:

$$\begin{array}{l}
x \text{ +ve and } y \text{ -ve } \rightarrow x \not< y \\
x \text{ -ve and } y \text{ +ve } \rightarrow x < y \\
x \text{ +ve and } y \text{ +ve } \rightarrow x < y \text{ if } \textsc{Abs}(x) < \textsc{Abs}(y) \\
x \text{ -ve and } y \text{ -ve } \rightarrow x < y \text{ if } \textsc{Abs}(y) < \textsc{Abs}(x) .
\end{array}$$

Thus we can realise a signed less than comparison just using some simple tests of the MSBs of x and y, which determine their sign, and controlled use of our unsigned less than comparison circuit.

The final thing to note is that once we have built hardware for equality and less than comparison, other operations can be derived using low-cost identities rather than additional dedicated circuits. For example, one can easily verify that

$$\begin{array}{l}
x \neq y \rightarrow \neg(x = y) \\
x \leq y \rightarrow (x < y) \vee (x = y) \\
x \geq y \rightarrow \neg(x < y) \\
x > y \rightarrow \neg(x < y) \wedge \neg(x = y)
\end{array}$$

such that all six useful comparisons can be generated with hardware to test for equality and less than, and four extra logic gates.

7.3 Addition and Subtraction

7.3.1 Addition

Most people learn the method for multi-digit addition in school; we describe the steps in Algorithm 7.1. Say we select example inputs $x = 123_{(10)}$ and $y = 218_{(10)}$. Using our algorithm, we can produce a trace which details the values before and after major assignments as execution of the algorithm progresses:

i	x_i	y_i	c	t	z_i	c
0	3	8	0	11	1	1
1	2	1	1	4	4	0
2	1	2	0	3	3	0

and find that the result is $z = 341_{(10)}$. However, we are often used to writing such operations more diagrammatically. To calculate the sum $z = x + y$ in base b by hand for example, we first write out the digits of x and y with similar powers of b above

```
Input: The $n$-digit integers $x$ and $y$ in a base-$b$ representation.
Output: The $n+1$-digit integer $z = x + y$ in a base-$b$ representation.

ADD$(x, y)$
    $z \leftarrow 0$
    $c \leftarrow 0$
    for $i = 0$ upto $n - 1$ step $1$ do
        $t \leftarrow x_i + y_i + c$
        $z_i \leftarrow t \bmod b$
        $c \leftarrow \lfloor t/b \rfloor$
    $z_n \leftarrow c$
    return $z$
end
```

Algorithm 7.1: An algorithm for multi-digit addition.

each other. In decimal, this should be a familiar process; again taking $x = 123_{(10)}$ and $y = 218_{(10)}$ we produce the sum

$$\begin{array}{ccc|l}
1 & 2 & 3 & \text{input } x \\
2 & 1 & 8 & \text{input } y \\
\hline
0 & 1 & 0 & \text{carry} \\
\hline
3 & 4 & 1 & \text{result } z = x + y \\
\hline
\end{array}$$

Note the zero in the first carry-in denoting that there is implicitly no carry into the first addition: we count this as zero. The same algorithm or diagrammatic method works in any base we care to represent our numbers in. If we consider binary numbers $x = 123_{(10)} = 1111011_{(2)}$ and $y = 281_{(10)} = 11011010_{(2)}$, we proceed as follows:

$$\begin{array}{ccccccccc|l}
 & & 1 & 1 & 1 & 1 & 0 & 1 & 1 & \text{input } x \\
 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & \text{input } y \\
\hline
1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & \text{carry} \\
\hline
1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & \text{result } z = x + y \\
\hline
\end{array}$$

to again compute the result $z = x + y = 341_{(10)} = 101010101_{(2)}$. Notice the carries propagate from right to left: we produce a carry from the i-th step which is propagated into the sum for the $(i+1)$-th step.

Although our algorithm works fine in general, we are presented with a problem when restricted to working with n-bit integers. The problem, termed overflow, is that the result of adding two n-bit integers together can create an $(n+1)$-bit result which is obviously too large to represent in n bits. This occurs when there is a carry-out of step $n-1$ of our algorithm and is demonstrated perfectly by the example above. Specifically, the inputs $x = 123_{(10)} = 1111011_{(2)}$ and $y = 281_{(10)} = 11011010_{(2)}$

are both 8 bits long but the result $z = x + y = 341_{(10)} = 101010101_{(2)}$ is 9 bits long. Given we want to represent z in 8 bits we have two main choices

- **Truncate** the result to 8 bits such that $z = 85_{(10)} = 01010101_{(2)}$ and signal by some method that an overflow has occurred so the error can be detected.
- **Clamp** or saturate the result to the highest value representable in the 8-bit size so that we have that $z = 255_{(10)} = 11111111_{(2)}$.

Clamped addition is useful in some contexts where accuracy is not important, for example when computing with graphical information represented as pixels. However, for other contexts we prefer to simply truncate the result and flag that overflow has occurred.

This detection of overflow is slightly complicated when one considers signed numbers. As an example, consider adding together $x = 127_{(10)}$ and $y = -127_{(10)}$ represented as 8-bit twos-complement binary vectors:

$$\begin{array}{cccccccccl}
 & & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \text{input } x \\
 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & \text{input } y \\
\hline
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & \text{carry} \\
\hline
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \text{result } z = x + y \\
\hline
\end{array}$$

We truncate the result to 8 bits and find $z = 0_{(10)} = 0_{(2)}$ but there is a carry-out of the last step which means $z_9 = 1$. Using the reasoning above, this should mean there is a carry-out yet we know the result should be zero and this can clearly be represented as an 8-bit value so what went wrong ? In short, we need to consider the sign of the two inputs and the sign of the output; detection of overflow can then be formulated not in terms of the carry-out of the addition but by some simple rules. Assuming z is the value of the result $z = x + y$ after truncation has occurred, we have

$$\begin{array}{lcl}
x \text{ +ve and } y \text{ -ve} & \rightarrow & \text{no overflow} \\
x \text{ -ve and } y \text{ +ve} & \rightarrow & \text{no overflow} \\
x \text{ +ve and } y \text{ +ve and } z \text{ +ve} & \rightarrow & \text{no overflow} \\
x \text{ +ve and } y \text{ +ve and } z \text{ -ve} & \rightarrow & \text{overflow} \\
x \text{ -ve and } y \text{ -ve and } z \text{ +ve} & \rightarrow & \text{overflow} \\
x \text{ -ve and } y \text{ -ve and } z \text{ -ve} & \rightarrow & \text{no overflow}
\end{array}$$

Overflow cannot occur if the input numbers have differing signs since the result *must* lie somewhere in the range of numbers representable. The overflow cases offer perhaps the most intuitive way to explain the rest: for example if we are adding two positive numbers together and get a negative result, something has clearly gone wrong. Given that x_{n-1} is the MSB of the value x and that this determines the sign of x in a twos-complement representation, we have that overflow occurs exactly when the expression

$$(x_{n-1} \wedge y_{n-1} \wedge \neg z_{n-1}) \vee (\neg x_{n-1} \wedge \neg y_{n-1} \wedge z_{n-1})$$

is true.

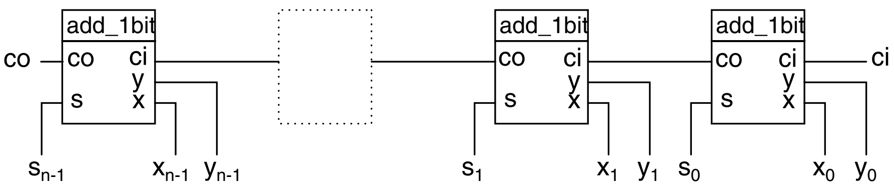


Figure 7.4 Design for an n-bit ripple-carry adder.

```
module add_nbit( co, ci, r, x, y );

  parameter n = 8;

  output wire         co;
  input  wire         ci;

  output wire [n-1:0] r;
  input  wire [n-1:0] x;
  input  wire [n-1:0] y;

         wire [n-2:0] w0;

  add_1bit t0( .co(w0[  0]), .s(r[  0]),
               .ci(ci     ), .x(x[  0]), .y(y[  0]) );
  add_1bit t1( .co(co     ), .s(r[n-1]),
               .ci(w0[n-2]), .x(x[n-1]), .y(y[n-1]) );

  genvar i;
  generate
    for( i = 1; i < n-1; i = i + 1 )
    begin:gen_add
      add_1bit t2( .co(w0[i  ]), .s(r[i]),
                   .ci(w0[i-1]), .x(x[i]), .y(y[i]) );
    end
  endgenerate

endmodule
```

Listing 7.3 Implementation of an n-bit ripple-carry adder.

7.3.1.1 Ripple-Carry Adder

We already know how to construct half and full-adders from basic logic gates that are capable of adding two 1-bit values and a carry-in to give a sum and carry-out as a result. Our task in using this basic component to add larger n-bit values is fairly simple: we just need to chain together the carry-in and carry-out signals of n full-adders. This chaining technique was described previously in Chapter 2 where we built a 4-bit adder from 1-bit full-adder components.

Following this, Figure 7.4 shows the very similar, but general structure for an n-bit adder. This design is called a **ripple-carry adder** since the flow of carries through the adders mimics the flow of carries in our algorithm: the carries ripple or propagate from the i-th adder to the $(i+1)$-th adder. As such, the design can only

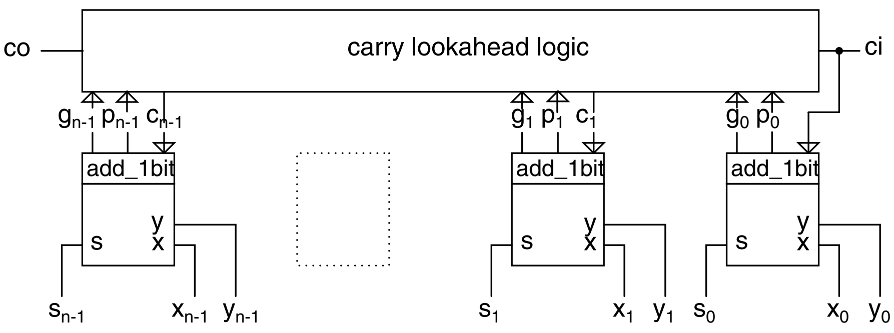


Figure 7.5 Design for an n-bit carry look-ahead adder.

calculate the i-th bit of the result once the adders to the right (i.e., those for the less-significant bits) have completed their results and propagated the carry. Implementing the design is very easy; Listing 7.3 shows a general-purpose n-bit adder module built assuming we have the standard `add_1bit` module available to us. To make the description more compact, we use a generate statement to instantiate the $n-2$ full-adders in the middle of the chain and only explicitly instantiate the two at either end which carry-in and carry-out `ci` and `co` respectively.

7.3.1.2 Carry Look-Ahead Adder

The goal of a **carry look-ahead adder** design is to reduce the propagation delay associated with the carry chain through a ripple-carry adder. The basic idea behind achieving this is to directly compute the carry-in for the i-th step rather than have it fed through from the $(i-1)$-th step.

We classify each step as either generating a carry-out or propagating a carry-in. Working in base-b, a step can generate a carry-out if $x_i + y_i \geq b$ while it can propagate a carry-in if $x_i + y_i = b - 1$. Consider an example with decimal, i.e., base-10, values:

$$\begin{array}{rl} 1\ 4\ 5 & \text{input } x \\ \underline{1\ 5\ 5} & \text{input } y \\ \underline{1\ 1\ 0} & \text{carry} \\ \underline{3\ 0\ 0} & \text{result } z = x + y \end{array}$$

The first step generates a carry-out because $5+5 \geq 10$ so the carry is passed into the next step. The second step propagates this carry; although it does not directly generate a carry-out since $4+5=9$, the carry-in from the previous step increments this such that the carry is again passed onto the next step. The final step neither generates nor propagates a carry since $1+1=2$ which, when incremented to accommodate the carry-in, is still less than the base-10.

In binary things are again a little more simple than in the general case. To determine if the i-th step of the sum $x+y$ generates or propagates a carry, which we denote by g_i and p_i respectively, we can use

$$\begin{aligned} g_i &= x_i \wedge y_i \\ p_i &= x_i \oplus y_i. \end{aligned}$$

That is, the step generates a carry-out if both digits are 1 and propagates a carry-in if either digit is 1 (but not both, otherwise it would generate a carry-out). Using these simple starting points, we can directly compute c_i, the carry-in for the i-th step. Given c_0 is the carry-in for the whole addition, we continue to find

$$\begin{aligned} c_1 &= g_0 \vee (p_0 \wedge c_0) \\ c_2 &= g_1 \vee (p_1 \wedge c_1) = g_1 \vee (p_1 \wedge g_0) \vee (p_1 \wedge p_0 \wedge c_0) \\ c_3 &= g_2 \vee (p_2 \wedge c_2) = g_2 \vee (p_2 \wedge g_1) \vee (p_2 \wedge p_1 \wedge g_0) \vee (p_2 \wedge p_1 \wedge p_0 \wedge c_0) \\ &\vdots \end{aligned}$$

Reading the first line, we have a carry-in for step one if step zero generates a carry-out or if step zero propagates a carry-in *and* there is a carry-in for step zero. Subsequent lines follow a similar reasoning, but are increasingly complex. However, using this approach we have significantly reduced the propagation delay associated with generating a result. Figure 7.5 demonstrates the general design of a carry look-ahead adder circuit.

As an aside, note that it is common to see g_i and p_i written as

$$\begin{aligned} g_i &= x_i \wedge y_i \\ p_i &= x_i \vee y_i. \end{aligned}$$

Of course, when used in the above this change does not alter the meaning: if $x_i = 1$ and $y_i = 1$, then $g_i = 1$ so it does not matter what the corresponding p_i is in this case. As such, use of an OR gate rather than an XOR is preferred because the former requires less transistors. In the ripple-carry adder case the circuit has a depth of $O(n)$ gates to generate a result, with carry look-ahead we reduce this depth to $O(\log n)$. The trade-off is that even though the circuit is less deep and hence generates a result faster, it generally uses many more gates: roughly $O(n^2)$ for a carry look-ahead adder versus $O(n)$ for a ripple-carry adder. As a compromise between these advantages and disadvantages, it can be attractive to chain together small carry look-ahead adders, say four 8-bit adders, in a ripple-carry configuration to form a larger, say a 32-bit, adder.

7.3.2 Subtraction

Subtraction is performed in a similar way to addition except we substitute carries from the i-th step to the $(i+1)$-th step into borrows from the $(i+1)$-th step by the

Input: The n-digit integers x and y in a base-b representation.
Output: The $n+1$-digit integer $z = x - y$ in a base-b representation.

```
SUB($x,y$)
    $z \leftarrow 0$
    $c \leftarrow 0$
    for $i = 0$ upto $n-1$ step $1$ do
        $t \leftarrow x_i - y_i + c$
        $z_i \leftarrow t \bmod b$
        $c \leftarrow \lfloor t/b \rfloor$
    return $z$
end
```

Algorithm 7.2: An algorithm for multi-digit subtraction.

i-th step. In decimal we show this using the values $x = 218_{(10)}$ and $y = 123_{(10)}$ to make our life slightly easier, and execute Algorithm 7.2 to produce the trace

i	x_i	y_i	c	t	z_i	c
0	8	3	0	5	5	0
1	1	2	0	−1	9	−1
2	2	1	−1	0	0	0

and calculate $z = x - y = 95_{(10)}$. Written in the more conventional diagram form we get

$$\begin{array}{rrrl} 2 & 1 & 8 & \text{input } x \\ 1 & 2 & 3 & \text{input } y \\ \hline -1 & 0 & 0 & \text{borrow} \\ \hline & 9 & 5 & \text{result } z = x - y \\ \hline \end{array}$$

Translating this into the binary case results in exactly the same process; for our test values $x = 281_{(10)} = 11011010_{(2)}$ and $y = 123_{(10)} = 1111011_{(2)}$ we again compute $z = x + y = 95_{(10)} = 1011111_{(2)}$ as follows:

$$\begin{array}{rrrrrrrrl} 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & \text{input } x \\ & 1 & 1 & 1 & 1 & 0 & 1 & 1 & \text{input } y \\ \hline -1 & -1 & -1 & -1 & -1 & -1 & -1 & 0 & \text{borrow} \\ \hline & 1 & 0 & 1 & 1 & 1 & 1 & 1 & \text{result } z = x - y \\ \hline \end{array}$$

However, implementing separate addition and subtraction is somewhat pointless given we can deal with subtraction by using the identity $z = x - y = x + (-y)$. That is, we first compute the negation of y and then add this to x to get our result. At first glance it seems like we will still need an extra circuit to perform the negation, but in fact this can be achieved using the following convenient method to convert a

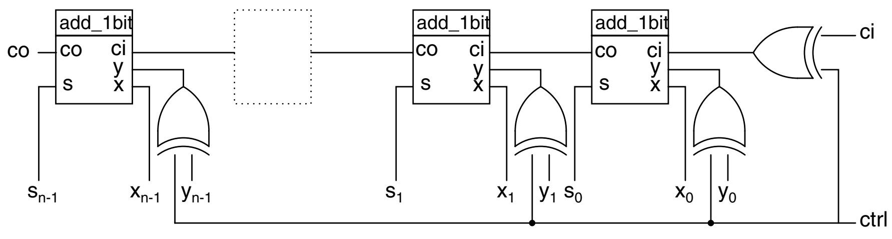


Figure 7.6 Design for an n-bit ripple-carry adder/subtractor.

number y into the twos-complement negation:

$$-y \equiv \neg y + 1.$$

So we simply invert each bit in y and then add one. However, we need to take care with regard to the carry-in since in subtraction this is essentially a borrow-from so we really want to compute $x + y + ci$ during addition and $x - y - ci$ during subtraction.

Inverting each bit of y is simple: we just have a control wire which determines if our adder should do an addition or a subtraction and XOR this control wire with each bit in y. As a result, when the control wire $ctrl = 1$ we have y_i inverted and when $ctrl = 0$ we pass y_i through unchanged. Adding one to the result is also simple: we just set the initial carry-in of our adder chain to 1. However, we need to take care to only do this when required; based on $ctrl$ and ci the four possible cases are

$$\begin{array}{lcl}
ctrl = 0 \text{ and } ci = 0 \rightarrow x + y + 0 & = & x + y \\
ctrl = 0 \text{ and } ci = 1 \rightarrow x + y + 1 & = & x + y + 1 \\
ctrl = 1 \text{ and } ci = 0 \rightarrow x + (\neg y + 1) - 0 & = & x + (\neg y) + 1 \\
ctrl = 1 \text{ and } ci = 1 \rightarrow x + (\neg y + 1) - 1 & = & x + (\neg y)
\end{array}$$

Therefore, we actually set the initial carry-in of our adder chain to equal $ctrl \oplus ci$ so that it is 1 only where either we are doing an addition and there is a carry-in, or we are doing a subtraction and there is no borrow-from.

Figure 7.6 shows a design for a combined circuit that can perform addition and subtraction depending on a control signal. This design is implemented in Listing 7.4 and looks very similar to our initial adder implementation in that we again rely on the basic `add_1bit` module. This time, instead of feeding `y` into each adder as the second input, we use `w1` which is conditionally negated by using an XOR of each bit of `y` with the control value `ctrl`. This is accomplished using a replication operator to form an n-bit vector where each bit is equal to `ctrl`. Then, instead of feeding in `ci` as the carry-in to the first adder we use `ctrl` again so as to get an extra addition of 1 where negation of `y` is required. Actually, this is not quite true: to cope with the fact that we might still need to use the carry-in value in addition, we

```
module sub_nbit( ctrl, co, ci, r, x, y );

  parameter n = 8;

  input  wire         ctrl;

  output wire         co;
  input  wire         ci;

  output wire [n-1:0] r;
  input  wire [n-1:0] x;
  input  wire [n-1:0] y;

         wire [n-2:0] w0;
         wire [n-1:0] w1 = y ^ {n{ctrl}};

  add_1bit t0( .co(w0[  0]  ), .s(r[  0]),
               .ci(ci ^ ctrl), .x(x[  0]), .y(w1[  0]) );
  add_1bit t1( .co(co       ), .s(r[n-1]),
               .ci(w0[n-2]  ), .x(x[n-1]), .y(w1[n-1]) );

  genvar i;
  generate
    for( i = 1; i < n-1; i = i + 1 )
    begin:gen_sub
      add_1bit t2( .co(w0[i  ]), .s(r[i]),
                   .ci(w0[i-1]), .x(x[i]), .y(w1[i]) );
    end
  endgenerate

endmodule
```

Listing 7.4 Implementation of an *n*-bit ripple-carry adder/subtractor.

actually XOR `ctrl` with `ci` to form the first carry-in. This allows us, for example, to set `ctrl` to 0 for an addition yet still set `ci` equal to 1.

7.4 Shift and Rotate

Formally, a left or right **shift** of an integer x represented in base-b by a distance y multiplies x by b^{+y} or b^{-y}. This basically has the effect of moving the digits left or right in the expansion. Consider an example value of $x = 123_{(10)}$. Working in decimal, shifting x left by setting $s = 2$ multiplies it by $10^2 = 100$ giving

$$12300_{(10)}$$

while shifting right by setting $y = -2$ multiplies it by $10^{-2} = 0.01$, i.e., it divides it by 100, to give

$$1.23_{(10)}.$$

This process works exactly the same in binary. So with $x = 1111011_{(2)}$, shifting x left by setting $y = 2$ multiplies x by $2^2 = 4$ to give

$$111101100_{(2)}$$

while shifting right by setting $y = -2$ multiplies x by $2^{-2} = 0.25$, i.e., it divides it by 4, to give

$$11110.11_{(2)}.$$

When representing numbers as n-bit binary vectors, there are some additional questions that need answering. For example, consider the vector $x = 01111011$ representing $1111011_{(2)}$ in 8 bits. Shifting left by setting $y = 2$ gives

$$111011??.$$

We have lost two digits from the left-hand, most-significant end because they cannot be accommodated in 8 bits. More importantly the right-hand, least-significant end of the vector needs two extra digits so it remains 8 bits long; these are marked with ?. The same is true when shifting in the other direction so that shifting right by setting $y = -2$ gives

$$??011110.$$

This time we have lost two digits from the right-hand, least-significant end and need two extra digits inserted into the left-hand, most-significant end.

How we fill the bits marked ? depends on the type of operation we require. Roughly, there are three important operations: **logical shifts**, **arithmetic shifts**, and **rotations**. These are best explained by example, starting with operations in the left direction:

- If we are performing a logical left-shift, we fill the least-significant bits of the result with 0. Thus, a logical left-shift of the vector $x = 01111011$ by $y = 2$ results in 11101100.
- If we are performing a left-rotate, we fill the least-significant bits of the result with the most-significant bits expelled from the most-significant end. Thus, a logical left-shift of the vector $x = 01111011$ by $y = 2$ results in 11101101.

Similar behaviour can be applied to operations in the right direction:

- If we are performing a logical right-shift, we fill the most-significant bits of the result with 0. Thus, a logical right-shift of the vector $x = 01111011$ by $y = -2$ results in 00011110.
- If we are performing an arithmetic right-shift, we fill the most-significant bits of the result with the most-significant or sign bit. Thus, an arithmetic right-shift of the vector $x = 01111011$ by $y = -2$ results in 00011110 while using the value $x = 11111011$ results in 11111110. This preserves the sign so we get the result we would expect if viewing right-shift as a form of division.
- If we are performing a right-rotate, we fill the most-significant bits of the result with the least-significant bits expelled from the least significant end. Thus, a logical right-shift of the vector $x = 01111011$ by $y = 2$ results in 11011110.

The idea of differentiating between logical and arithmetic shifts is that arithmetic shifts preserve the sign of the value involved while logical shifts do not. This can be important depending on the context in which the operation is used: if we really want to view right-shifting as a division by 2^y, we need an arithmetic shift otherwise the sign of the result will be wrong.

Once we know what the various operations should actually do, we can start to think about ways of implementing them in hardware. There are two main categories of implementation which are described in detail below. To make things easier for ourselves, we will only consider shifting values by unsigned distances. The assumption is that the device controlling our shifter will translate, for example, a logical left-shift by a negative distance into a logical right-shift by a positive distance of the same magnitude if required. Thus the assumption is that for an n-bit value x we always have the distance $0 \leq y < \log_2(n)$.

Finally, we will assume that the style of shift performed by our hardware is dictated by some input which can take one of the following constant values:

```
`define SHF_R_LOGIC  3'b000
`define SHF_R_ARITH  3'b001
`define SHF_R_ROTATE 3'b010
`define SHF_L_LOGIC  3'b011
`define SHF_L_ARITH  3'b100
`define SHF_L_ROTATE 3'b101
```

with `` `SHF_L_ARITH `` being an unused placeholder.

7.4.1 Bit-Serial Shifter

The most simple way to realise the required behaviour is to use a sequential or bit-serial approach. The basic idea is to build a hardware device capable of shifting values by a distance of one bit and then iterating the use of this hardware using y steps to shift a distance of y bits. The advantage of this approach is that it uses the least logic; we only require hardware to shift by distances of one bit coupled to some control hardware. The drawback is that it is slow to produce a result; we must make y sequential steps to get the required result.

Listing 7.5 details the implementation of a simple **bit-serial shifter**. The module uses the input `oper` to decide which type of operation is required; one can perform left or right logical or arithmetic shifts and rotations. In order to realise the iterated behaviour, we maintain a counter in register `r1` that tracks how many iterations we have performed so far. This value is incremented by `add_nbit` instance `t0` and compared with the desired number of iterations by `equ_nbit` instance `t1`.

Execution of a shift operation is initialised when a positive edge is detected on the reset input named `rst`. When this event occurs we reset the iteration counter to zero and the temporary value `r0`, which stores the value being shifted, to the initial input `x`. On positive edges of the `clk` signal the shifter state is updated; that is, one iteration is executed. We start by updating our temporary shifted value `r0` by shifting it by a distance of one bit in a style determined by `oper`. We then check if enough iterations have been performed by examining `w1`, the result of comparing the

```
module shf_nbit( clk, rst, oper, r, x, y );

  parameter n = 8;
  parameter m = 3;

  input  wire         clk;
  input  wire         rst;

  input  wire [  2:0] oper;

  output reg  [n-1:0] r;
  input  wire [n-1:0] x;
  input  wire [n-1:0] y;

         reg  [n-1:0] r0;
         reg  [m-1:0] r1;
         reg          r2;

         wire [m-1:0] w0;
         wire         w1;

  add_nbit t0(.ci(1'b0),.r(w0),.x(r1),.y(1       ) );
  equ_nbit t1(          .r(w1),.x(w0),.y(y[m-1:0]) );

  defparam t0.n = m;
  defparam t1.n = m;

  always @ ( posedge rst )
  begin
    r0 = x;
    r1 = 0;
    r2 = 0;
  end

  always @ ( posedge clk )
  begin
    case( oper )
      `SHF_R_LOGIC  : r0 = { 1'b0,    r0[n-1:1] };
      `SHF_R_ARITH  : r0 = { r0[n-1], r0[n-1:1] };
      `SHF_R_ROTATE : r0 = { r0[  0], r0[n-1:1] };
      `SHF_L_LOGIC  : r0 = { r0[n-2:0], 1'b0    };
      `SHF_L_ROTATE : r0 = { r0[n-2:0], r0[n-1] };
    endcase

    if( !r2 )
    begin
      r1 = w0;

      if( w1 )
      begin
        r  = r0;

        r2 =  1;
      end
    end
  end

endmodule
```

Listing 7.5 Implementation of an *n*-bit bit-serial shifter.

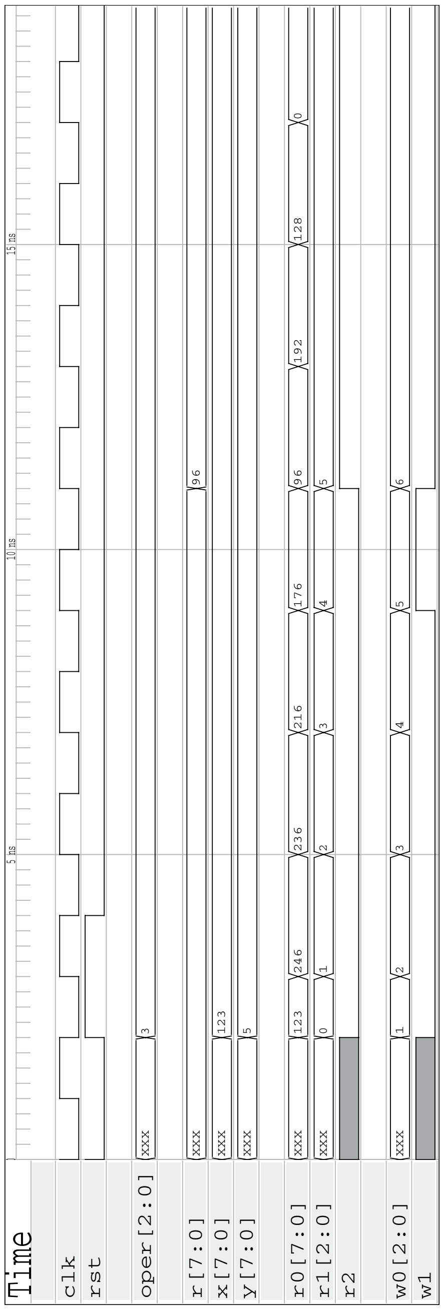


Figure 7.7 Behaviour of an *n*-bit bit-serial shifter.

iteration counter with the desired number of iterations. If we have done enough, we set the shifter output `r` to equal our temporary value `r0` to complete the operation. Otherwise we update the iteration counter by incrementing it and wait for `clk` to trigger another iteration.

Figure 7.7 demonstrates the behaviour of this design for the inputs $x = 123_{(10)} = 1111011_{(2)}$ and $y = 5_{(10)} = 101_{(2)}$ when asked to perform a left logical shift. The clock has a period of 2ns and the shifter is reset at time $t = 2$ns by the positive edge of `rst`. Then, on each positive edge of the clock one can see the loop counter, accumulator and so on being updated. By time $t = 12$ns the result is set as $r = 96_{(10)} = 1100000_{(2)}$ which is what we expect; the left-most five bits of x have been shifted off and zeros inserted as padding on the right-hand side.

7.4.2 Logarithmic Shifter

In reality, it is often desirable to generate results quickly rather than to minimise the amount of logic used. As such, one can consider alternative designs that by using more logic can produce a result in just one step rather than y steps. This approach demands that we make a combinatorial circuit rather than a clocked circuit; the speed one can generate results is then dominated by the critical path of the circuit rather than the clock speed. We consider a conceptually simple approach which is termed a **logarithmic shifter**. Essentially this design is a series of multiplexers which shift the input by differing constant amounts to form the result.

The key idea is that each bit of the shift distance y determines a shift by a constant amount 2^i. So for example, if $y_1 = 1$ then we need to shift by a distance of two at some point. The combination of the different constant shifts results in the overall result so if $y = 3_{(10)} = 11_{(2)}$, we have $y_0 = 1$ and $y_1 = 1$ and so need to shift by a distance of one then a distance of two. Since the different constant shifts are independent, we can do them separately and have one unit shifting by 1 then feed this result to a unit shifting by two. Both units are easy to construct since the shift distances are now constant. The end result is a structure shown in Figure 7.8. The design is combinatorial and produces the end result in one clock cycle; the obvious drawback is the amount of logic used and longer associated critical path.

Implementing a general n-bit logarithmic shifter in Verilog is quite hard due to the constraints on use of generate statements; one could easily write a C program to generate the required code however. To provide a concrete example, we present the implementation of a specific 8-bit shifter. Hopefully it is clear that similar techniques can easily be applied to larger designs. Listing 7.6 details the implementation which consists of three similar stages. Each stage consists of two multiplexers: a `mul8_8bit` instance which shifts the input by a constant distance using a style dictated by `oper`, and a `mul2_8bit` instance which decides between the shifted and unshifted input depending on the associated bit of input `y`. So for example multiplexer instance `t0` shifts the input `w0`, an alias for `x`, by a distance of one bit. Multiplexer instance `t1` then decides whether to feed the original value `w0` or the

```
module shf_8bit( input  wire [2:0] oper,

                 output wire [7:0] r,
                 input  wire [7:0] x,
                 input  wire [7:0] y );

  wire [7:0] w0;
  wire [7:0] w1;
  wire [7:0] w2;
  wire [7:0] w3;
  wire [7:0] w4;
  wire [7:0] w5;
  wire [7:0] w6;

  assign w0 =  x;
  assign r  = w6;

  mux8_8bit t0( .r(w1),

                .i0({ 1'b0,         w0[7:1] }),
                .i1({ {1{w0[7:7]}}, w0[7:1] }),
                .i2({ w0[0:0],      w0[7:1] }),
                .i3({ w0[6:0],      1'b0    }),
                .i5({ w0[6:0],      w0[7:7] }),

                .s0(oper[0]), .s1(oper[1]), .s2(oper[2]) );

  mux2_8bit t1( .r(w2), .i0(w0), .i1(w1), .s0(y[0]) );

  mux8_8bit t2( .r(w3),

                .i0({ 2'b0,         w2[7:2] }),
                .i1({ {2{w2[7:7]}}, w2[7:2] }),
                .i2({ w2[1:0],      w2[7:2] }),
                .i3({ w2[5:0],      2'b0    }),
                .i5({ w2[5:0],      w2[7:6] }),

                .s0(oper[0]), .s1(oper[1]), .s2(oper[2]) );

  mux2_8bit t3( .r(w4), .i0(w2), .i1(w3), .s0(y[1]) );

  mux8_8bit t4( .r(w5),

                .i0({ 4'b0,         w4[7:4] }),
                .i1({ {4{w4[7:7]}}, w4[7:4] }),
                .i2({ w4[3:0],      w4[7:4] }),
                .i3({ w4[3:0],      4'b0    }),
                .i5({ w4[3:0],      w4[7:4] }),

                .s0(oper[0]), .s1(oper[1]), .s2(oper[2]) );

  mux2_8bit t5( .r(w6), .i0(w4), .i1(w5), .s0(y[2]) );

endmodule
```

Listing 7.6 Implementation of an 8-bit logarithmic shifter.

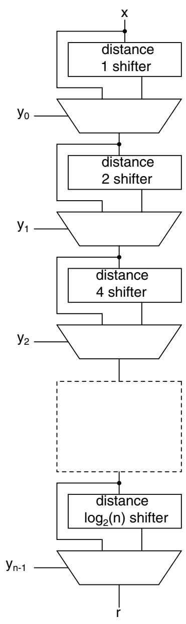


Figure 7.8 General design of a basic logarithmic shifter architecture.

shifter value `w1` through to the result `w2` depending on the value of `y[0]`. The next stage, comprising the multiplexer instances `t2` and `t3`, performs a similar operation on `w2` to produce `w4` but for a shift distance of two bits; `t4` and `t5` then shift `w4` by a distance of four bits to produce `w6`. The end result `w6` is the correctly shifted value and is assigned to the shifter output `r`.

7.5 Multiplication

Like addition, most people learn the method for multi-digit multiplication in school. When calculating $z = x \cdot y$ we term x the **multiplicand** and y the **multiplier**; since multiplication is associative we can switch these roles if we want. To make things easier for ourselves, we will only consider multiplication of unsigned numbers. Using some extra logic one can easily deal with signed input by forcing the inputs to be unsigned and then fixing up the sign of the result.

In a similar way to addition, we can describe multiplication using Algorithm 7.3 and produce a trace of execution for our example inputs $x = 123_{(10)}$ and $y = 218_{(10)}$

```
Input: The $n$-digit integers $x$ and $y$ in a base-$b$ representation.
Output: The $n+1$-digit integer $z = x \cdot y$ in a base-$b$ representation.
MUL$(x, y)$
    $z \leftarrow 0$
    for $i = 0$ upto $n-1$ step $1$ do
        $c \leftarrow 0$
        for $j = 0$ upto $n-1$ step $1$ do
            $t \leftarrow x_i \cdot y_j + z_{i+j} + c$
            $z_{i+j} \leftarrow t \bmod b$
            $c \leftarrow \lfloor t/b \rfloor$
        $z_{i+w} \leftarrow c$
    return $z$
end
```

Algorithm 7.3: An algorithm for multi-digit multiplication.

i	j	x_i	y_j	z_{i+j}	c	t	z_{i+j}	c
0	0	3	8	0	0	24	4	2
0	1	3	1	0	2	5	5	0
0	2	3	2	0	0	6	6	0
1	0	2	8	5	0	21	1	2
1	1	2	1	6	2	10	0	1
1	2	2	2	0	1	5	5	0
2	0	1	8	0	0	8	8	0
2	1	1	1	5	0	6	6	0
2	2	1	2	0	0	2	2	0

Alternatively, we can describe the operation diagrammatically as

$$\begin{array}{rl}
1\ 2\ 3 & \text{input } x \\
2\ 1\ 8 & \text{input } y \\
\hline
9\ 8\ 4 & \text{partial product for } +x \cdot 8 \\
1\ 2\ 3\ & \text{partial product for } +x \cdot 10 \\
2\ 4\ 6\ & \text{partial product for } +x \cdot 200 \\
\hline
2\ 6\ 8\ 1\ 4 & \text{result } z = x \cdot y \\
\hline
\end{array}$$

At the i-th step we produce a partial product equal to $(y_i \cdot x) \cdot b^i$. That is, we take the i-th digit of y, multiply it by x and then shift it along i places into the right column to form the partial product. The partial products are then added together to produce the result. As with all our algorithms we can work in binary just as easily as decimal and thus for $x = 123_{(10)} = 1111011_{(2)}$ and $y = 211_{(10)} = 11011010_{(2)}$ we have

```
Input: The n-bit integers x and y in a base-2 representation.
Output: The 2n-bit integer z = x · y in a base-2 representation.
MUL-BIT-SERIAL(x, y)
    t ← 0
    for i = n − 1 downto 0 step −1 do
        t ← 2 · t
        if y_i = 1 then
            t ← t + x
    return t
end
```

Algorithm 7.4: A bit-serial algorithm for multiplication of binary numbers.

```
              0 1 1 1 1 0 1 1 input x
              1 1 0 1 1 0 1 0 input y
-----------------------------
              0 0 0 0 0 0 0 0 partial product for +x · 0
            0 1 1 1 1 0 1 1   partial product for +x · 10
          0 0 0 0 0 0 0 0     partial product for +x · 000
        0 1 1 1 1 0 1 1       partial product for +x · 1000
      0 1 1 1 1 0 1 1         partial product for +x · 10000
    0 0 0 0 0 0 0 0           partial product for +x · 000000
  0 1 1 1 1 0 1 1             partial product for +x · 1000000
0 1 1 1 1 0 1 1               partial product for +x · 10000000
-----------------------------
1 1 0 1 0 0 0 1 0 1 1 1 1 1 0 result z = x · y
-----------------------------
```

One minor issue that presents itself immediately from this description is that given two n-bit inputs, any multiplier design we select will generate a $2n$-bit result. This differs from addition and logical operations which both generate n-bit results if one ignores the carry-out. This is not a major problem: we just need an extra output from the ALU so we can return the high and low halves of the result to the processor core from the multiplier.

Either way, since multiplication is one of the most costly operations it is worthwhile considering the best way to realise the functionality of Algorithm 7.3 in hardware. There are a large number of different implementation options but most multipliers fall roughly into one of three categories which are described in detail below.

7.5.1 Bit-Serial Multiplier

At face value, the description of multiplication in binary above seems much more complicated than the decimal case. However, it is actually made far easier because at the i-th step, the i-th bit of the multiplier can only be zero or one: we simply either

```
module mul_nbit( clk, rst, r, x, y );

  parameter n = 8;
  parameter m = 3;

  input  wire           clk;
  input  wire           rst;

  output reg  [2*n-1:0] r;
  input  wire [  n-1:0] x;
  input  wire [  n-1:0] y;

         reg  [2*n-1:0] r0;
         reg  [  n-1:0] r1;
         reg  [  m-1:0] r2;
         reg            r3;

         wire [  m-1:0] w0;
         wire           w1;
         wire [2*n-1:0] w2 = { r0[2*n-2:0], 1'b0 };
         wire [2*n-1:0] w3;

  add_nbit t0(.ci(1'b0),.r(w0),.x(r2),.y(               1));
  equ_nbit t1(          .r(w1),.x(r2),.y(             n-1));
  add_nbit t2(.ci(1'b0),.r(w3),.x(w2),.y({ {n{1'b0}}, x }));

  defparam t0.n =   m;
  defparam t1.n =   m;
  defparam t2.n = 2*n;

  always @ ( posedge rst )
  begin
    r0 = 0;
    r1 = y;
    r2 = 0;
    r3 = 0;
  end

  always @ ( posedge clk )
  begin
    if( r1[n-1] )
      r0 = w3;
    else
      r0 = w2;

    r1 = { r1[n-2:0], 1'b0 };

    if( !r3 )
    begin
      r2 = w0;

      if( w1 )
      begin
        r  = r0;
        r3 =  1;
      end
    end
  end

endmodule
```

Listing 7.7 Implementation of an *n*-bit bit-serial multiplier.

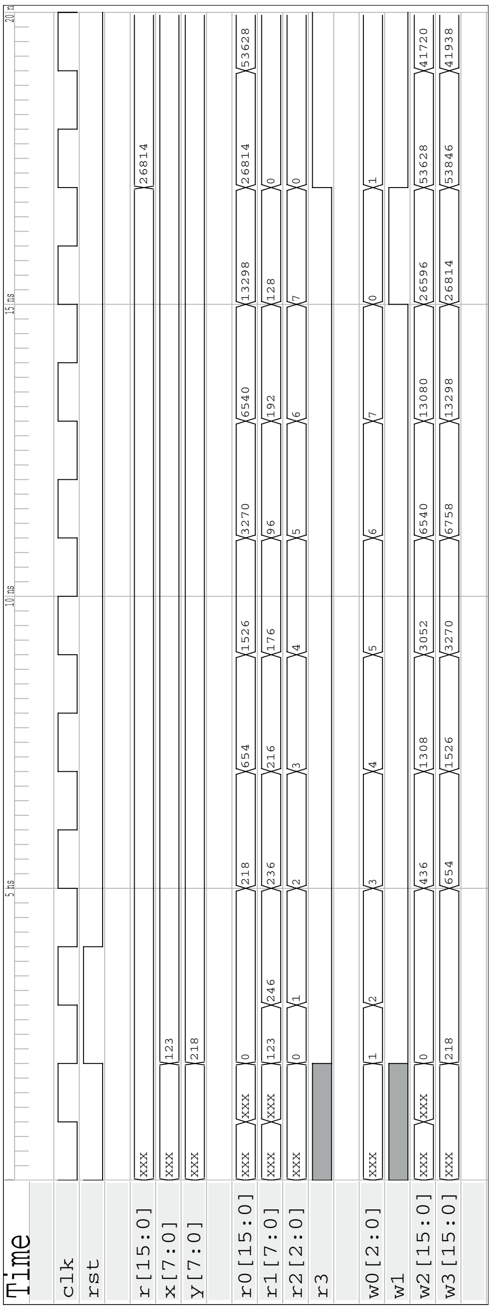


Figure 7.9 Behaviour of an *n*-bit bit-serial multiplier.

add the partial product into the accumulated total or not. Combined with the fact that the partial product at the *i*-th step is simply the multiplicand shifted left *i* digits, we can formulate the bit-serial method for multiplication in Algorithm 7.4. To see why this algorithm works, we need to think about what multiplication "means". One way to look at it is that multiplication is just repeated addition. For example

$$x \cdot y = \underbrace{x + x + \cdots + x + x}_{\text{total of } y \text{ terms}}$$

so if we select $y = 6_{(10)}$, then we obviously have

$$x \cdot 6 = x + x + x + x + x + x.$$

But there is a clear drawback to this in the sense that the bigger y is, the longer that list of terms is and hence the more additions we are going to need to do if we actually want to compute $x \cdot y$. Another way of looking at it is that multiplication adds another "weight" into the summation above. It might look odd, but imagine we wrote y out in binary; then we could write $x \cdot y$ as

$$x \cdot y = x \cdot \sum_{i=0}^{n-1} y_i \cdot 2^i.$$

Again selecting $y = 6_{(10)} = 110_{(2)}$ we find that we still get the result we would expect to:

$$\begin{aligned} x \cdot y &= x \cdot y_2 \cdot 2^2 + x \cdot y_1 \cdot 2^2 + x \cdot y_0 \cdot 2^0 \\ &= x \cdot 1 \cdot 2^2 \quad + x \cdot 1 \cdot 2^2 \quad + x \cdot 0 \cdot 2^0 \\ &= x \cdot 4 \qquad + x \cdot 2 \qquad + x \cdot 0 \\ &= x \cdot 6 \qquad\qquad\qquad\qquad . \end{aligned}$$

The problem is, this still looks unpleasant to compute. For example, we keep having to compute powers of two to "weight" the terms. Fortunately, British mathematician William Horner worked out a scheme to do this more neatly. Basically we just bracket the thing we started with in such a way that instead of having to compute the powers of two independently, we sort of accumulate them; this is best shown by example:

$$\begin{aligned} x \cdot y &= (\,(\,x \cdot y_2\,) \cdot 2 + x \cdot y_1\,) \cdot 2 + x \cdot y_0 \\ &= (\,(\,x \cdot 1\;\,) \cdot 2 + x \cdot 1\;\,) \cdot 2 + x \cdot 0 \\ &= (\quad x \qquad \cdot 2 + x \cdot 1\;\,) \cdot 2 + x \cdot 0 \\ &= \quad\; x \qquad \cdot 4 + x \cdot 2 \qquad\; + x \cdot 0 \\ &= \quad\; x \qquad \cdot 6 \qquad\qquad . \end{aligned}$$

Working inside out (i.e., from the inner-most term in the nest of brackets to the outer-most) the computation no longer looks so unpleasant, plus we can see the relationship to Algorithm 7.4. That is, at the *i*-th step we take the accumulated value and double it then add x if $y_i = 1$.

As one can imagine, this approach uses the least area of our three choices since it requires only an adder to add partial products into the accumulator, a shifter to ensure the partial products are added into the right place, and some control logic to execute the loop. However, it also yields the worst performance in that it takes us n steps to multiply two n-bit values. Listing 7.7 describes an implementation of Algorithm 7.4 but requires some explanation. The multiplier is reset to an initial state on positive edges of the `rst` signal. On positive edges of the `clk` signal the multiplier state is updated; one can think of this as executing one iteration of the loop on each tick of the clock. To control the loop we use register `r2` as the loop counter, incrementing it via `add_nbit` instance `t0` and comparing it with the loop bound via `equ_nbit` instance. The register `r0` is used to store the accumulator from Algorithm 7.4. The wire `w2` represents `r0` shifted left by one bit, i.e., multiplied by two. On each iteration of the loop we update `r0` with either `w3`, the result of adding `x` to `w2` via the `add_nbit` instance `t2`, or `w2` itself. That is, on each iteration of the loop we either double `r0` or double it and add `x` depending on the related bit in `y`. To make life easier, we store `y` in register `r1` and shift the value left one bit each time the loop is iterated; this means we always inspect bit $n-1$ of `r1` rather than bit i of `y`. Finally, when we have done enough iterations the accumulator `r0` is assigned to the multiplier result `r` to complete the operation.

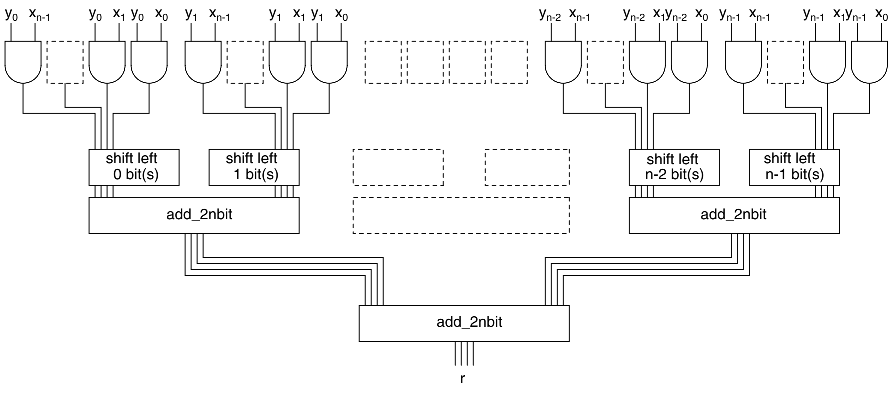
The behaviour of this design when fed our example inputs $x = 123_{(10)}$ and $y = 218_{(10)}$ is shown in Figure 7.9. The clock has a period of 2ns and the multiplier is reset at time $t = 2$ns by the positive edge of `rst`. Then, on each positive edge of the clock one can see the loop counter, accumulator and so on being updated. At time $t = 16$ns the final loop iteration completes and we output the accumulated result on `r`, the multiplier is then ready for any subsequent operations required of it.

7.5.2 Tree Multiplier

A parallel multiplier attempts to achieve high-performance by using the largest possible area of our three options. An example of this type is a **tree multiplier**; the basic idea is to generate all the partial products in parallel and then implement a large combinatorial circuit to add them together. This means that although we use much more logic than a bit-serial multiplier, we generate a result in one step and in a time bounded by propagation delay through the adders.

In implementing such a multiplier, instead of using a linear chain of n adders to accumulate the n partial products we reduce the associated propagation delay by using a tree. The result is the delay before a result is produced is $O(\log n)$ rather than $O(n)$. The general outline for such a design is shown in Figure 7.10. One can see how the n partial products would be accumulated by a tree of depth $\log_2 n$. The i-th partial product itself is constructed by ANDing input x with a vector replicated from the i-th bit of input y; the result is x if we want to add this partial product to the final result and zero otherwise.

Like the logarithmic shifter, describing a general n-bit parallel multiplier in Verilog is quite difficult; we again implement an example 8-bit design in Listing 7.8. The first block of code defines wires `w0` to `w7` and uses them to represent the eight



y_0 x_{n-1} y_0 x_1 y_0 x_0 y_1 x_{n-1} y_1 x_1 y_1 x_0
y_{n-2} x_{n-1} y_{n-2} x_1 y_{n-2} x_0 y_{n-1} x_{n-1} y_{n-1} x_1 y_{n-1} x_0
shift left 0 bit(s)
shift left 1 bit(s)
shift left n-2 bit(s)
shift left n-1 bit(s)
add_2nbit
add_2nbit
add_2nbit
r

Figure 7.10 Design for a basic parallel multiplier architecture.

partial products. The *i*-th partial product is shifted left by *i* bits to place it in the right position for accumulation. The partial products are then accumulated by the second block of code which defines `t0` to `t6`, a tree of `add_nbit` instances which outputs the final result in `r`. Note that in this simple example we have padded all our partial products to 16 bits and parametrised our adder tree accordingly. In reality this is somewhat wasteful since, for example, in the addition of `w0` and `w1` we know some bits of either input are zero: we know that bit eight upwards of `w0` and bit zero of `w1` are all zero so this is a "special" addition in a sense. The bottom line is that by eliminating unnecessary operations that can be identified at design-time, we can reduce the overall cost in terms of logic.

7.5.3 Digit-Serial Multiplier

One way to view the previous designs is that a bit-serial approach deals with the multiplier one bit at a time while the parallel approach deals with all n multiplier bits at once. The natural compromise lies somewhere between these extremes and deals with say d bits in parallel with $1 < d < n$. We term such a design a digit-serial multiplier; d is said to be the digit size and is normally selected to divide n exactly to simplify implementation.

Algorithm 7.5 presents a rough outline of the algorithm where we use **par** to denote parallel execution of a sequence of statements. The basic idea is that we now step through the outer loop in steps of d instead of steps of 1 as in the original bit-serial version; in each iteration of the loop we accumulate d partial products in parallel. Thus for around a d-fold increase in size, we produce a d-fold increase in performance. This is a nice trade-off since we can scale d to match our size budget or our performance goals.

```
module mul_8bit( output wire [15:0] r,
                 input  wire [ 7:0] x,
                 input  wire [ 7:0] y );

  wire [15:0] w0 = {8'b0, {8{y[0]}} & x}      ;
  wire [15:0] w1 = {8'b0, {8{y[1]}} & x} << 1;
  wire [15:0] w2 = {8'b0, {8{y[2]}} & x} << 2;
  wire [15:0] w3 = {8'b0, {8{y[3]}} & x} << 3;
  wire [15:0] w4 = {8'b0, {8{y[4]}} & x} << 4;
  wire [15:0] w5 = {8'b0, {8{y[5]}} & x} << 5;
  wire [15:0] w6 = {8'b0, {8{y[6]}} & x} << 6;
  wire [15:0] w7 = {8'b0, {8{y[7]}} & x} << 7;

  wire [15:0] s0;
  wire [15:0] s1;
  wire [15:0] s2;
  wire [15:0] s3;
  wire [15:0] s4;
  wire [15:0] s5;

  add_nbit t0(.ci(1'b0),.r(s0),.x(w0),.y(w1));
  add_nbit t1(.ci(1'b0),.r(s1),.x(w2),.y(w3));
  add_nbit t2(.ci(1'b0),.r(s2),.x(w4),.y(w5));
  add_nbit t3(.ci(1'b0),.r(s3),.x(w6),.y(w7));

  add_nbit t4(.ci(1'b0),.r(s4),.x(s0),.y(s1));
  add_nbit t5(.ci(1'b0),.r(s5),.x(s2),.y(s3));

  add_nbit t6(.ci(1'b0),.r(r ),.x(s4),.y(s5));

  defparam t0.n = 16;
  defparam t1.n = 16;
  defparam t2.n = 16;
  defparam t3.n = 16;

  defparam t4.n = 16;
  defparam t5.n = 16;

  defparam t6.n = 16;

endmodule
```

Listing 7.8 Implementation of an 8-bit parallel multiplier.

7.5.4 Early Termination

To reduce the number of iterations taken by the basic bit-serial multiplier, one can consider the concept of **early termination** or **early exit**. We demonstrate this concept as a continuation of our left-to-right bit-serial multiplier from above. The basic idea is to check at some iteration i if the remaining bits of y are zero, i.e., $y_{i\ldots0} = 0$. In this case we would simply iterate through the remaining bits always performing the operation $t \leftarrow 2 \cdot t$ but never performing the operation $t \leftarrow t + x$. So instead, we can exit the loop and scale t in one go, returning $t \cdot 2^i$ and thereby replicating the effect of i iterations of the operation $t \leftarrow t + t$. This is inexpensive because the scaling can be achieved simply by shifting t by a known distance. The benefit is that we do not do all those extra iterations, so potentially we can save some time.

Input: The n-bit integers x and y in a base-2 representation, a digit-size d.
Output: The $2n$-bit integer $z = x \cdot y$ in a base-2 representation.

```
MUL-DIGIT-SERIAL($x, y$)
    $t \leftarrow 0$
    for $i = n-1$ downto $0$ step $-d$ do
        $t \leftarrow 2^d \cdot t$
        par $j = 0$ upto $d-1$ step $1$
            if $y_{i+j} = 1$ then
                $t \leftarrow t + x \cdot 2^j$
    return $t$
end
```

Algorithm 7.5: A digit-serial algorithm for multiplication of binary numbers.

Input: The n-bit integers x and y in a base-2 representation, an integer block size $b|n$.
Output: The $2n$-bit integer $z = x \cdot y$ in a base-2 representation.

```
MUL-BIT-SERIAL-ET($x, y$)
    for $i = n/b$ downto $1$ step $-1$ do
        if $y_{b \cdot i-1 \ldots 0} = 0$ then
            return $2^{b \cdot i} \cdot t$
        else
            for $j = 0$ upto $b-1$ step $1$ do
                $t \leftarrow 2 \cdot t$
                if $y_{b \cdot i-1-j} = 1$ then
                    $t \leftarrow t + x$
    return $t$
end
```

Algorithm 7.6: A bit-serial algorithm for multiplication of binary numbers with early termination.

Algorithm 7.6 roughly models what is going on. The idea is to perform the multiplication in n/b steps; in the case of $n = 32$ we might set $b = 8$ for example to perform four steps, dealing with eight bits of y in each step. The first thing we do in each step is check whether the remaining bits of y are all zero: if they are, the scaled t is returned straight away, otherwise we continue more or less as in the case of bit-serial multiplication.

Some examples might illustrate better what is going on. Imagine we use the values $n = 32$ and $b = 8$, and consider the inputs $x = 5_{(10)}$ and $y = 6_{(10)} = 00000006_{(16)}$; by writing y in hexadecimal, it is easy to see that the bottom 8-bit chunk is non-zero.

Therefore, when the algorithm checks the remaining bits of y at each of the four sets, there are still some which are non-zero so it behaves the same as the original bit-serial algorithm:

i	j	$8 \cdot i - 1 - j$	$y_{8 \cdot i - 1 - j}$	t	t
4	0	31	0	0	$0+0 = 0$
4	1	30	0	0	$0+0 = 0$
4	2	29	0	0	$0+0 = 0$
4	3	28	0	0	$0+0 = 0$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
1	4	3	0	0	$0+0 = 0$
1	5	2	1	0	$0+0+5 = 5$
1	6	1	1	5	$5+5+5 = 15$
1	7	0	0	15	$15+15 = 30$

Now consider the inputs $x = 5_{(10)}$ and $y = 1536_{(10)} = 0000600_{(16)}$:

i	j	$8 \cdot i - 1 - j$	$y_{8 \cdot i - 1 - j}$	t	t
4	0	31	0	0	$0+0 = 0$
4	1	30	0	0	$0+0 = 0$
4	2	29	0	0	$0+0 = 0$
4	3	28	0	0	$0+0 = 0$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
2	4	11	0	0	$0+0 = 0$
2	5	10	1	0	$0+0+5 = 5$
2	6	9	1	5	$5+5+5 = 15$
2	7	8	0	15	$15+15 = 30$

At this point, i.e., the final step, the algorithm notices the least-significant 8-bit chunk of y is zero so returns the value $30 \cdot 2^8$ (which is $7680_{(10)}$ as expected) without performing the loop iterations associated with the final step.

Some ARM processors use early termination but combine this with some extra techniques and use a right-to-left approach rather than left-to-right as we have done. The reasoning for this is simple: in a normal workload, most multipliers will be small and hence their more-significant bits are more likely to be zero than their less-significant bits. For small multipliers (e.g., loop counters, array indices and so on) the left-to-right approach will yield a higher chance of early termination, and hence greater reduction in the number of iterations.

7.5.5 Wallace and Dadda Trees

Tree multipliers offer a good trade-off in favour of performance versus size. However, a drawback of such designs is the long critical path which runs through the tree

	Stage 1	Stage 2		Stage 3	
	Output	Action	Output	Action	Output
Weight-1	1	*PT*	1	*PT*	1
Weight-2	2	*HA*	1	*PT*	1
Weight-4	3	*FA*	2	*HA*	1
Weight-8	4	*FA*	3	*FA*	2
Weight-16	3	*FA*	2	*HA*	2
Weight-32	2	*HA*	2	*HA*	2
Weight-64	1	*PT*	2	*HA*	2
Weight-128	0		0		1

(a) Actions in constructing Wallace multiplier for 4-bit inputs.

	Stage 1	Stage 2		Stage 3	
	Output	Action	Output	Action	Output
Weight-1	1	*PT*	1	*PT*	1
Weight-2	2	*PT*	2	*PT*	2
Weight-4	3	*FA*	1	*PT*	1
Weight-8	4	*FA*	3	*FA*	1
Weight-16	3	*FA*	2	*HA*	2
Weight-32	2	*PT*	3	*FA*	2
Weight-64	1	*PT*	1	*PT*	2
Weight-128	0		0		0

(b) Actions in constructing Dadda multiplier for 4-bit inputs.

Table 7.1 A tabular description of the stages in Wallace and Dadda multipliers for 4-bit inputs.

of adders; if we use ripple-carry adders to build an n-bit tree adder, the number of 1-bit full-adders the critical path passes through is essentially $O(n \log_2 n)$.

A **carry-save** adder takes a different approach to ripple-carry and carry look-ahead adders: instead of computing an addition per-se, it compresses three n-bit inputs x, y and z into two n-bit outputs. That is, it computes s and c, which we call the partial sum and shifted carry, where

$$\begin{aligned} s_i &= x_i \oplus y_i \oplus z_i \\ c_i &= (x_i \wedge y_i) \vee (x_i \wedge z_i) \vee (y_i \wedge z_i). \end{aligned}$$

Sometimes this structure is called a 3 : 2 **compressor**. To compute the real $(n+2)$-bit sum of x, y and z one needs to combine s and c in a further addition step by computing $2c + s$ using a standard (ripple-carry for example) adder. The idea is that use of a carry-save adder followed by a ripple-carry adder is faster than two sequential uses of ripple-carry adders.

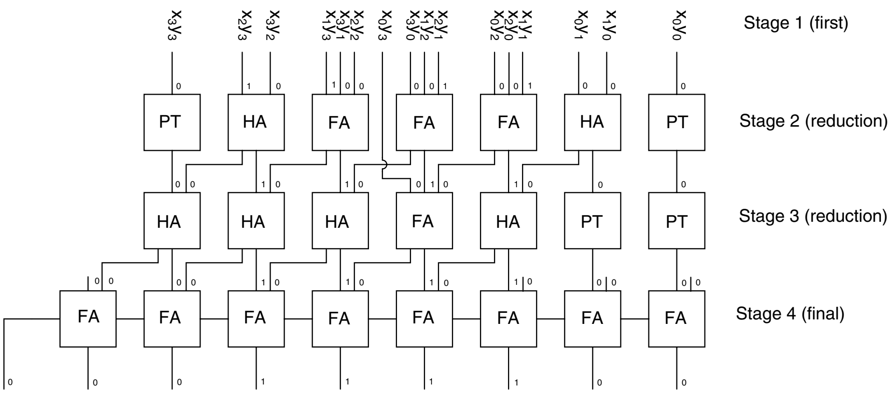


Figure 7.11 An example 4-bit Wallace multiplier.

Wallace multiplier [63] and **Dadda multiplier** [19] designs capitalise on the concept of carry-save adders. Both are tree-like multiplier designs in the sense that they generate and sum partial products in parallel, but improve the critical path through the multiplier. For n-bit inputs, both are constructed from a number of stages: an initial stage to generate the partial products; one or more reduction stages which act to reduce the partial products into two $2n$-bit values; and a final stage which adds the $2n$-bit values to produce the result. With a Wallace multiplier the basic approach to multiplying x and y is as follows:

1. In the first stage, multiply together (i.e., AND together) each x_i with each y_j to produce a total of n^2 intermediate wires. Each wire is said to have a weight, for example $x_0 \cdot y_0$ has weight 1 as $2^0 \cdot 2^0 = 1$, $x_1 \cdot y_2$ has weight 8 as $2^1 \cdot 2^2 = 8$.
2. Reduce the number of intermediate wires using additional stages composed of full-adders and half-adders:

   - Combine any three wires with same weight using a full-adder; result in next stage is one wire of the same weight (i.e., the sum) and one wire a higher weight (i.e., the carry).
   - Combine any two wires with same weight using a half-adder; result in next stage is one wire of the same weight (i.e., the sum) and one wire a higher weight (i.e., the carry).
   - If there is only one wire with a given weight, just pass it through to the next stage.

3. In the final stage, all weights have just one or two wires in them: combine the wires into two $2n$-bit values and add them with a standard adder.

The corresponding rules for a Dadda multiplier are similar in concept to the Wallace multiplier except the reduction stages are more complicated; that is, we replace the second step above with the following:

- Combine any three wires with same weight using a full-adder; the result in next stage is one wire of the same weight (i.e., the sum) and one wire a higher weight (i.e., the carry).
- If there are two wires with the same weight left, let w be that weight and then:
  - If $w \equiv 2 \bmod 3$ then combine the wires using a half-adder; the result in next stage is one wire of the same weight (i.e., the sum) and one wire a higher weight (i.e., the carry).
  - Otherwise, just pass them through to the next stage.
- If there is one wire with the same weight left, just pass it through to the next stage.

Consider a case were we are dealing with 4-bit inputs x and y; Table 7.1 demonstrates the actions applied in constructing the various stages of the Wallace and Dadda multipliers. Since they are similar, we consider only the Wallace case more carefully. The first stage multiplies each x_i with each y_j; we have one weight-0 wire from $x_0 \cdot y_0$ but three weight-4 wires from $x_1 \cdot y_3$, $x_3 \cdot y_1$ and $x_2 \cdot y_2$ for example. Following the first stage, we apply two reduction stages according to the rules. For example, in the first reduction stage there is one weight-1 wire as input so we use a pass-through (PT) operation which results in one weight-1 wire as output; there are two weight-2 wires so we use a half-adder (HA) operation which results in one weight-2 wire and one weight-4 wire as output; there are three weight-4 wires so we use a full-adder (FA) operation which results in one weight-4 wire and one weight-8 wire as output. Finally, we are left with the output of the second reduction stage in which all weights have two or less wires in them; we group the wires into two 8-bit values and add them using, for example, a ripple-carry adder. This might all sound a bit abstract; consider a concrete example where we want to multiply $x = 0110_{(2)} = 6_{(10)}$ and $y = 1010_{(2)} = 10_{(10)}$. Figure 7.11 shows the structure of the entire Wallace multiplier and the intermediate results produced. The final stage computes the result $00111100_{(2)} = 60_{(10)}$ as the sum of $00111100_{(2)}$ and $00000000_{(2)}$ where zeros are inserted where there are not enough wires.

Notice that the Wallace multiplier used six half-adders and four full-adders and the final addition has 6-bit inputs; the Dadda multiplier used one half-adder and five full-adders but the final addition has 5-bit inputs. As such, one can view the two approaches as a trade-off between complexity in the reduction layers against complexity of the (single) final addition. In each structure, there are $O(\log n)$ layers of reduction plus the initial and final layer. However, notice that within each reduction stage each half-adder or full-adder can operate independently from the rest. That is, each reduction stage has an $O(1)$ critical path which is why we see an improvement versus the original tree multiplier: the critical path is essentially through only $O(\log_2 n)$ full-adders rather than $O(n \log_2 n)$.

7.5.6 Booth Recoding

The technique of **recoding** either the multiplier or multiplicand to allow more efficient multiplication can be attractive: essentially we do some extra work before executing the multiplication so that the actual operation takes less steps. The **Booth encoding** technique [5] is a popular example of this; it takes advantage of repeated sequences of 1 or 0 digits in the multiplier. Using the recoding strategy, addition operations are only required at the boundaries of such sequences; no additions are required within the sequences themselves.

For example, consider the 8-bit multiplier value $30_{(10)} = 00011110_{(2)}$ where one can see there is a repeated sequence of four 1 digits in the binary representation. Where such a sequence exists from bits position i to bit position j, we can treat the sequence as a single digit of weight $2^{j+1} - 2^i$. So in this case we re-write the digits from $i = 1$ to $j = 4$ as a single digit of weight $2^{4+1} - 2^1 = 2^5 - 2^1 = 30$. Since we have recoded a long sequence as a single digit, a generic multiplication algorithm can take less steps: it requires less additions of partial products resulting from the combination of multiplier digits and the multiplicand.

Consider the example of multiplying $x = 6_{(10)} = 00000110_{(2)}$ by $y = 30_{(10)} = 00011110_{(2)}$. Using the method introduced previously, we compute the result as

```
              0 0 0 0 0 1 1 0 input x
              0 0 0 1 1 1 1 0 input y
-----------------------------
              0 0 0 0 0 0 0 0 partial product for +x·0
            0 0 0 0 0 1 1 0   partial product for +x·10
          0 0 0 0 0 1 1 0     partial product for +x·100
        0 0 0 0 0 1 1 0       partial product for +x·1000
      0 0 0 0 0 1 1 0         partial product for +x·10000
    0 0 0 0 0 0 0 0           partial product for +x·000000
  0 0 0 0 0 0 0 0             partial product for +x·0000000
0 0 0 0 0 0 0 0               partial product for +x·00000000
-----------------------------
0 0 0 0 0 0 0 1 0 1 1 0 1 0 0 result z = x·y
-----------------------------
```

If we first recode y using the Booth method, the result is instead calculated as follows:

$$
\begin{array}{rrrrrrrrrrrrrrrl}
 & & & & & & & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & \text{input } x \\
 & & & & & & & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & \text{input } y \\
 & & & & & & & 0 & 0 & +1 & 0 & 0 & 0 & -1 & 0 & \text{recoded } y \\
\hline
 & & & & & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \text{partial product for } +x \cdot 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & & \text{partial product for } -x \cdot 10 \\
 & & & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & \text{partial product for } +x \cdot 000 \\
 & & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & \text{partial product for } +x \cdot 0000 \\
 & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & & \text{partial product for } +x \cdot 00000 \\
 & & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & & & & & & \text{partial product for } +x \cdot 100000 \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & & & & \text{partial product for } +x \cdot 0000000 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & & & & & \text{partial product for } +x \cdot 00000000 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & \text{result } z = x \cdot y \\
\hline
\end{array}
$$

which requires less addition operations, i.e., additions of non-zero partial products, during accumulation of the partial products and hence potentially less time.

The problem with this approach is that in the worse case, it actually causes us to require *more* addition operations during accumulation of the partial products than the standard method. To show this, consider a multiplier $y = 5_{(10)} = 00000101_{(2)}$. Using the standard multiplication method we would require two addition operations, one for each 1 digit in y. Using a Booth recoding of y, we actually perform four additions:

$$
\begin{array}{rrrrrrrrrrrrrrrl}
 & & & & & & & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & \text{input } x \\
 & & & & & & & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & \text{input } y \\
 & & & & & & & 0 & 0 & 0 & 0 & +1 & -1 & +1 & -1 & \text{recoded } y \\
\hline
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & \text{partial product for } -x \cdot 1 \\
 & & & & & & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & & \text{partial product for } +x \cdot 10 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & & & \text{partial product for } -x \cdot 100 \\
 & & & & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & & & & \text{partial product for } +x \cdot 1000 \\
 & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & & \text{partial product for } +x \cdot 00000 \\
 & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & & & \text{partial product for } +x \cdot 000000 \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & & & & \text{partial product for } +x \cdot 0000000 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & & & & & \text{partial product for } +x \cdot 00000000 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & \text{result } z = x \cdot y \\
\hline
\end{array}
$$

which is clearly worse not better ! To combat this worst case and exploit the fact that the recoded multiplier implies less steps in multiplication, we use a further recoding strategy which is often termed the **modified Booth recoding**. The basic idea is to group the recoded digits into pairs (reading them from right to left). Within a pair (y_{i+1}, y_i) grouped from the recoded value y, the digit y_{i+1} has twice the weight of y_i so one can think of the pair as having the weight $2y_{i+1} + y_i$. This allows us to think of the pair $(+1, -1)$ as $2 - 1 = 1$ for example. Seven such pairs are possible, with the other two impossible due to the original Booth recoding technique; the pairs contribute the following weights:

$$\begin{array}{llllll}
y_{i+1} = 0 & \text{and} & y_i = 0 & \rightarrow & 0 \\
y_{i+1} = 0 & \text{and} & y_i = +1 & \rightarrow & +1 \\
y_{i+1} = 0 & \text{and} & y_i = -1 & \rightarrow & -1 \\
y_{i+1} = +1 & \text{and} & y_i = 0 & \rightarrow & +2 \\
y_{i+1} = +1 & \text{and} & y_i = +1 & \rightarrow & \text{not possible} \\
y_{i+1} = +1 & \text{and} & y_i = -1 & \rightarrow & +1 \\
y_{i+1} = -1 & \text{and} & y_i = 0 & \rightarrow & -2 \\
y_{i+1} = -1 & \text{and} & y_i = +1 & \rightarrow & -1 \\
y_{i+1} = -1 & \text{and} & y_i = -1 & \rightarrow & \text{not possible}
\end{array}$$

Thus, using these pairings we can re-write our worst case as

$$\begin{array}{cccccccccccccc|l}
 & & & & & & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & \text{input } x \\
 & & & & & & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & \text{input } y \\
 & & & & & & 0 & 0 & 0 & 0 & +1 & -1 & +1 & -1 & \text{recoded } y \\
 & & & & & & & 0 & & 0 & & +1 & & +1 & \text{grouped } y \\
\hline
 & & & & & & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & \text{partial product for } +x \cdot 1 \\
 & & & & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & & & \text{partial product for } +x \cdot 100 \\
 & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & & \text{partial product for } +x \cdot 00000 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & & & & \text{partial product for } +x \cdot 0000000 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & \text{result } z = x \cdot y \\
\hline
\end{array}$$

which is no worse than using the standard method and makes explicit the fact that there are less steps: we only need to consider half the number of recoded multiplier digits versus the original.

7.6 Putting It All Together

To flesh out the components in the basic Verilog processor model developed at the end of Chapter 5, we need ALU modules to perform comparison and arithmetic on integer inputs. Previously we assumed that such operations could be created for us by the Verilog tool-chain, for example we used statements such as

```
  alu_dst_lo = GPR[dec_rs] + GPR[dec_rt]
```

to model the behaviour of an addition instruction. In reality, we would like to conglomerate all such operations within the ALU so that we can minimise the amount of replicated logic and modularise the overall processor design.

7.6.1 Comparison ALU

The ALU for comparisons is the simplest to construct. We already have modules that perform unsigned equality and less than comparisons; our task is to use and pack-

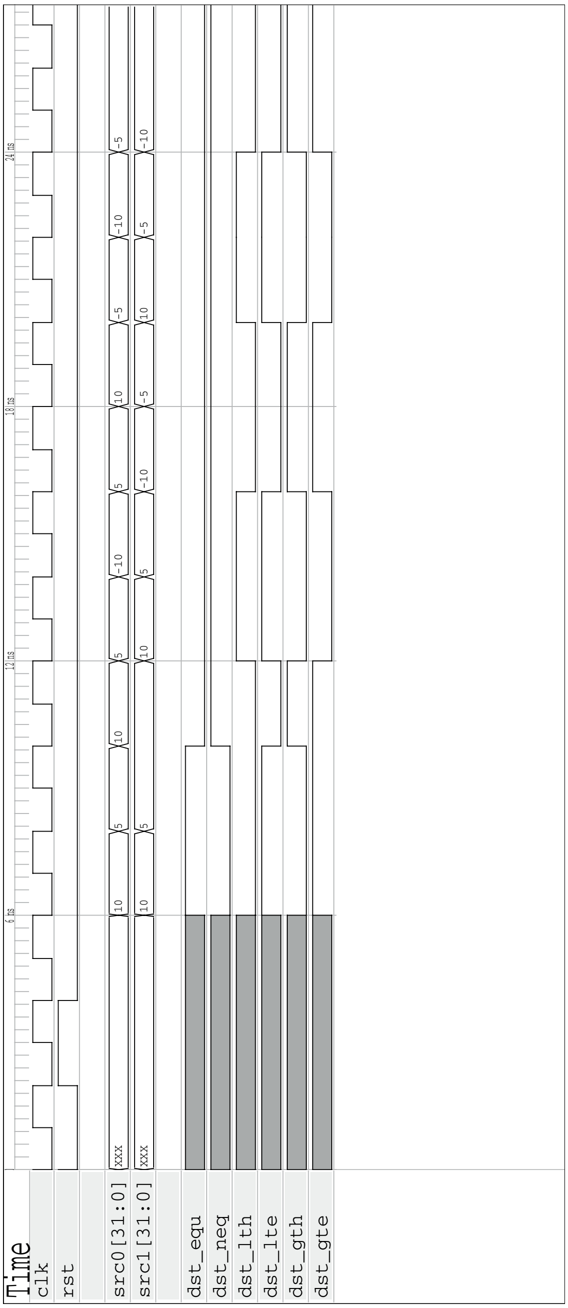


Figure 7.12 Behaviour of the ALU for comparisons.

```
module alu_cmp( input  wire        clk,
                input  wire        rst,

                input  wire [31:0] src0,
                input  wire [31:0] src1,

                output wire        dst_equ,
                output wire        dst_neq,
                output wire        dst_lth,
                output wire        dst_lte,
                output wire        dst_gth,
                output wire        dst_gte );

  wire w0;
  wire w1;
  wire w2;

  equ_nbit t0( .r(w0), .x(src0[31:0]), .y(src1[31:0]) );
  lth_nbit t1( .r(w1), .x(src0[30:0]), .y(src1[30:0]) );

  defparam t0.n = 32;
  defparam t1.n = 31;

  assign w2 = (  src0[31] &  src1[31] & ~w0 & w1 ) |
              (  src0[31] & ~src1[31]            ) |
              ( ~src0[31] & ~src1[31] & ~w0 & w1 ) ;

  assign dst_equ =  w0;
  assign dst_lth =  w2;

  assign dst_neq = ~dst_equ ;
  assign dst_lte =  dst_lth |
                    dst_equ ;
  assign dst_gth = ~dst_lte ;
  assign dst_gte = ~dst_lth ;

endmodule
```

Listing 7.9 A Verilog module that models an ALU for comparisons.

age these modules so they perform all variants of signed comparison. The Verilog module is shown in Listing 7.9. We have the ALU generate all possible comparison outputs continuously from the inputs `src0` and `src1`. That is, there is no need to pass the ALU an opcode: the processor simply sets the inputs and then bases any decision on the right comparison output. For example, if it wants to test if the values are equal then it examines the `dst_equ` output. This design is motivated by the fact that the MIPS32 instructions for conditional branches do not use the `funct` field of the R-type instruction encoding to determine the ALU operation like, for example, an addition would. Branches use the I-type encoding and so the operation performed is derived just from the `opcode` field.

The slight complication then is simply that we need signed comparison from our basic unsigned modules. We achieve this using a slight of hand. Firstly, note that signed equality is the same as unsigned equality: we just test if each bit is the same or not. Signed less than is the problem; to solve it we adopt the simple rules presented previously. That is, because of the way the twos-complement representation works we have that

```
module alu_exe( input  wire        clk,
                input  wire        rst,

                input  wire [ 5:0] oper,

                input  wire [31:0] src0,
                input  wire [31:0] src1,

                output wire [31:0] dst_lo,
                output wire [31:0] dst_hi );

  ...

endmodule
```

Listing 7.10 The interface to a Verilog module that models an ALU for arithmetic.

1. If `src0` is negative and `src1` is negative, they are not equal, and an unsigned less than comparison is true, then `src0` is less than `src1`.
2. If `src0` is negative and `src1` is positive we know `src0` is less than `src1`.
3. If `src0` is positive and `src1` is positive, they are not equal, and an unsigned less than comparison is true, then `src0` is less than `src1`.

Testing the sign of `src0` and `src1` is simply a matter of examining the most-significant bit. Using this fact we can apply our rules, and an unsigned comparison on the remaining 31-bit unsigned value, to generate the correct signed comparison. Then, since we have equality and less than, we construct all other comparisons from these. For example, `src0` is greater than `src1` if `src0` is not less than or equal to `src1`. The behaviour of the result is shown in Figure 7.12.

7.6.2 Arithmetic ALU

The ALU for arithmetic is somewhat more complex, partly because we have more operations and sub-module instances and partly because some of the operations are multi-cycle rather than continuous. The latter difference requires that we manage the clock and reset signals for the multiplier and shifter modules so they operate correctly. The Verilog code is too long to present on one page but the interface is shown in Listing 7.10. The general idea is that the ALU perform an operation, using `src0` and `src1` as inputs, which is determined by the `oper` input. The output is supplied in low and high parts via `dst_lo` and `dst_hi` which allows for both 32-bit outputs, for example addition, and 64-bit outputs, for example multiplication. The design does not include the facility to manage carry or overflow flags.

Listing 7.11 details the instantiation of the main sub-modules within the ALU. Instances `t3`, `t4` and `t5` are the addition/subtraction, multiplier and shifter modules described previously. Adder instances `t0`, `t1` and `t2` are used to manage the multiplier by ensuring it is supplied only with positive inputs. To achieve this, we negate `src0` and `src1` to produce `n0` and `n1` and select whichever one is pos-

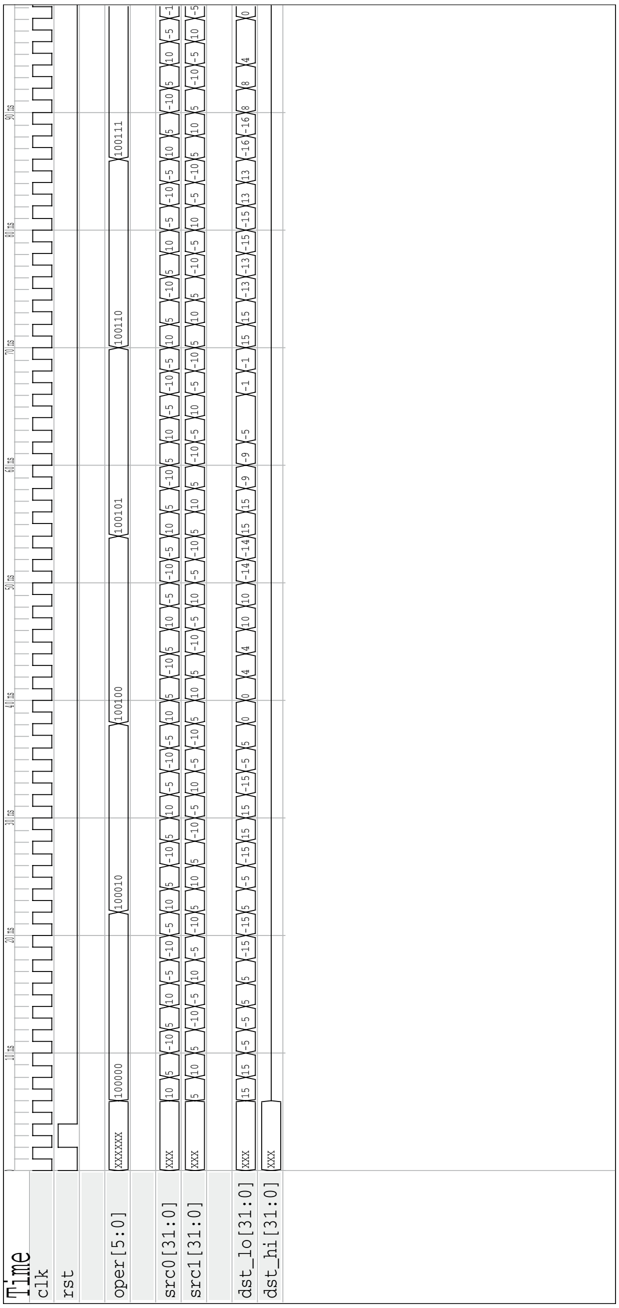


Figure 7.13 Behaviour of the ALU for arithmetic.

```
add_nbit t0(.ci(1'b0),.r(n0),.x(~src0),.y(32'b1));
add_nbit t1(.ci(1'b0),.r(n1),.x(~src1),.y(32'b1));
add_nbit t2(.ci(1'b0),.r(n2),.x(~  w1),.y(64'b1));

defparam t0.n = 32;
defparam t1.n = 32;
defparam t2.n = 64;

assign n3 = src0[31]            ? n0 : src0;
assign n4 = src1[31]            ? n1 : src1;
assign n5 = src0[31] != src1[31] ? n2 :   w1;

assign ci = 0;

sub_nbit t3( .ctrl(     s0),

             .ci   (     ci),
             .co   (     co),
             .x    (   src0),
             .y    (   src1),
             .r    (     w0) );

mul_nbit t4( .clk (    clk),
             .rst (mul_rst),

             .x    (     n3),
             .y    (     n4),
             .r    (     w1) );

shf_nbit t5( .clk (    clk),
             .rst (shf_rst),

             .oper(     s1),
             .x    (   src0),
             .y    (   src1),
             .r    (     w2) );

defparam t3.n = 32;
defparam t4.n = 32;
defparam t4.m =  5;
defparam t5.n = 32;
defparam t5.m =  5;
```

Listing 7.11 Verilog code to instantiate internal sub-modules.

itive to produce n3 and n4; these are fed to the multiplier. The multiplier output is patched up to generate the right sign by comparing the input signs and selecting either the actual multiplier output w1 or the negation n2 to produce the final result n5. Note also that we tie the addition/subtraction carry-in to 0 although in reality this value might come from a carry flag stored in the processor status register.

Listing 7.12 details the generation of the basic logic operations and the two multiplexers used to select the result from all those possible, which is then fed to the ALU output. Notice that the multiplexers are controlled by the s3 signal; this is managed by two processes shown in Listing 7.13. The second process is the most interesting: whenever the oper signal changes, i.e., we want to perform a new operation, a case statement is executed which compares the opcode to the list of possibilities and sets the right control signals in each case. The multiplier and shifter cases are the most

```
genvar i;
generate
  for( i = 0; i < 32; i = i + 1 )
  begin:gen_alu_exe
          and t6( w3[i], src0[i], src1[i] );
           or t7( w4[i], src0[i], src1[i] );
          xor t8( w5[i], src0[i], src1[i] );
          nor t9( w6[i], src0[i], src1[i] );

    mux8_1bit ta( .r(dst_lo[i]), .i0(w0[i]), .i1(n5[i+ 0]),
                                 .i2(w2[i]), .i3(w3[i   ]),
                                 .i4(w4[i]), .i5(w5[i   ]),
                                 .i6(w6[i]), .i7(1'bZ    ),

                                 .s0(s2[0]), .s1(s2[1]), .s2(s2[2]) );

    mux8_1bit tb( .r(dst_hi[i]), .i0(1'bZ ), .i1(n5[i+32]),
                                 .i2(1'bZ ), .i3(1'bZ    ),
                                 .i4(1'bZ ), .i5(1'bZ    ),
                                 .i6(1'bZ ), .i7(1'bZ    ),

                                 .s0(s2[0]), .s1(s2[1]), .s2(s2[2]) );
  end
endgenerate
```

Listing 7.12 Verilog code to generate logic operations and multiplexers.

involved; as well as selecting the right multiplexer control signal, these need to reset the associated device to clear the internal state and initiate a new operation. The first process ensures this reset signal is cleared soon afterwards ready for the next operation to be started. The behaviour of the result is shown in Figure 7.13.

7.7 Further Reading

- M.D. Ercegovac and T. Lang.
  Digital Arithmetic.
  Morgan Kaufmann, 2003. ISBN: 1-558-60798-6.
- M.J. Flynn and S.F. Oberman.
  Advanced Computer Arithmetic Design.
  John Wiley, 2001. ISBN: 0-471-41209-0.
- D. Harris and S. Harris.
  Digital Design and Computer Architecture: From Gates to Processors.
  Morgan-Kaufmann, 2007. ISBN: 0-123-70497-9.
- I. Koren.
  Computer Arithmetic Algorithms.
  A.K. Peters, 2001. ISBN: 1-568-81160-8.
- J.E. Stine Jr.
  Digital Computer Arithmetic Datapath Design Using Verilog HDL.
  Kluwer, 2003. ISBN: 1-402-07710-6.

```
always @ ( negedge clk )
begin
  shf_rst = 1'b0;
  mul_rst = 1'b0;
end

always @ ( oper )
begin
  case( oper )
    `FUNCT_ADD  : begin
                    s0 = 0;
                    s2 = 0;
                  end
    `FUNCT_SUB  : begin
                    s0 = 1;
                    s2 = 0;
                  end
    `FUNCT_MUL  : begin
                    mul_rst = 1;
                    s2      = 1;
                  end
    `FUNCT_MULT : begin
                    mul_rst = 1;
                    s2      = 1;
                  end

    `FUNCT_AND  : s2 = 3;
    `FUNCT_OR   : s2 = 4;
    `FUNCT_XOR  : s2 = 5;
    `FUNCT_NOR  : s2 = 6;

    `FUNCT_SLLV : begin
                    shf_rst = 1;
                    s1      = `SHF_L_LOGIC;
                    s2      = 2;
                  end
    `FUNCT_SRLV : begin
                    shf_rst = 1;
                    s1      = `SHF_R_LOGIC;
                    s2      = 2;
                  end
  endcase
end
```

Listing 7.13 Verilog processes that determine the ALU behaviour.

7.8 Example Questions

29. a. Comparison operations for a given processor take two 16-bit operands and return zero if the comparison is false or non-zero if it is true. By constructing some of the comparisons using combinations of other operations, show that implementing all of $=, \neq, <, \leq, >$ and $\geq$ is wasteful. State the smallest set of comparisons that need dedicated hardware such that all the standard comparisons can be executed.

b. The ALU in the same processor design does not include a multiply instruction. So that programmers can still multiply numbers, write an efficient C function to multiply two 16-bit inputs together and return the 16-bit lower half of the result. You can assume the inputs are always positive.

c. A population count operation takes an n-bit value x and computes the Hamming weight $H(x)$, i.e., the number of bits in x which are set to 1. Some processors have a dedicated instruction to do this but the proposed one does not; write an efficient C function to compute the population count of 16-bit inputs.

30. Imagine we want to compute the result of multiplying two n-bit numbers x and y together, i.e., $r = x \cdot y$, where n is even. One can adopt a divide-and-conquer approach to this computation by splitting x and y into two parts each of size $n/2$ bits

$$\begin{aligned} x &= x_1 \cdot 2^{n/2} + x_0 \\ y &= y_1 \cdot 2^{n/2} + y_0 \end{aligned}$$

and then computing the full result

$$r = r_2 \cdot 2^n + r_1 \cdot 2^{n/2} + r_0$$

via the parts

$$\begin{aligned} r_2 &= x_1 \cdot y_1 \\ r_1 &= x_1 \cdot y_0 + x_0 \cdot y_1 \\ r_0 &= x_0 \cdot y_0. \end{aligned}$$

The naive approach above uses four multiplications of $(n/2)$-bit values. The Karatsuba-Ofman method reduces this to three multiplications (and some extra low-cost operations); show how this is achieved.

31. Assume we are using unsigned integers represented in 4 bits.

a. What is the result of using a normal 4-bit adder circuit to compute the sum $10 + 12$?

b. A saturating or clamped adder is such that if an overflow occurs, that is the result does not fit into 4 bits, the highest possible number is returned as a result. With a clamped 4-bit addition denoted by $\uplus$, we have that $10 \uplus 12 = 15$. In general, for an n-bit clamped adder

$$x \uplus y = \begin{cases} x + y & \text{if } x + y < 2^n \\ 2^n - 1 & \text{otherwise} \end{cases}$$

Design a circuit that implements a 4-bit adder of this type.

32. Give a 1-line C expression to test if a non-zero integer x is an exact power-of-two; i.e., if $x = 2^n$ for some n then the expression should evaluate to a non-zero value, otherwise it evaluates to zero.