UNIVERSITY OF BRISTOL
DEPARTMENT OF COMPUTER SCIENCE
http://www.cs.bris.ac.uk

A Practical Introduction to Computer Architecture

Dan Page
⟨page@cs.bris.ac.uk⟩

COMPUTER ARITHMETIC

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

– C. Babbage

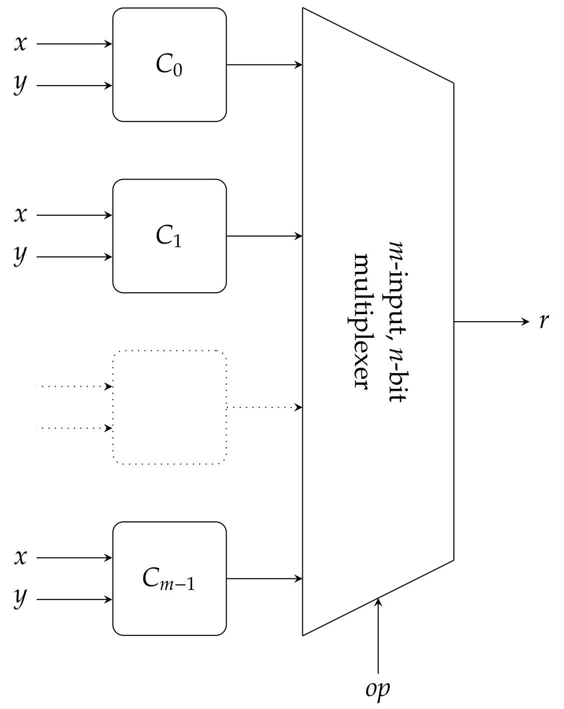
In Chapter 1, we saw how numbers could be represented using bit-sequences. More specifically, we demonstrated various techniques to represent both unsigned and signed integers using n-bit sequences. In Chapter 2 and Chapter 4, we then investigated how logic gates capable of computing Boolean operations (such as NOT, AND, and OR) and higher-level building block components could be designed and manufactured.

One way to view this content is as a set of generic techniques. We have the ability to design and implement components that computes any Boolean function, for example, and *reason about their behaviour in terms of Physics. A natural next step is to be more specific:* what *function would be useful? Among many possible options, the field of* **computer arithmetic** *provides some good choices. In short, arithmetic is something most people would class as computation; something as simple as a desktop calculator could still be classed as a basic computer. As such, the goal of this Chapter is to combine the previous material, producing a range of high-level building blocks that perform computation involving integers: although this useful and interesting in itself, it is important to keep in mind that it also represents a starting point for study of more general computation.*

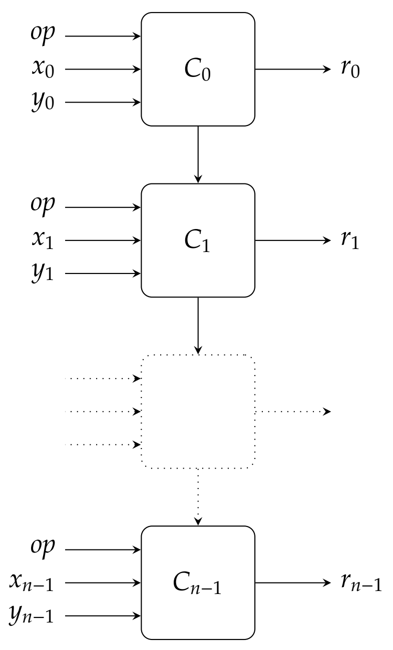
1 Introduction

The term **Arithmetic and Logic Unit (ALU)** is often used to describe a collection of components that perform computation within a micro-processor: when you write a C program with the expression `x + 1` in it, the ALU is what computes a result for you during execution. Historically, use of a dedicated ALU could be viewed as stemming from the design of EDVAC by John von Neumann [?]: he stressed any general-purpose computer would need to perform basic Mathematical operations on numbers, so it is "reasonable that [the computer] should contain specialised organs for these operations". In short, the modern ALU is an example of such an organ.

The design of an ALU and constituent components represents an application of computer arithmetic, a larger and more general field. Very roughly, the challenge is as follows: given one or more n-bit sequences that represent numbers, say x and y, how can we design a Boolean function $f(x, y)$ whose output represents a valid Mathematical operation? For example, imagine we want to compute $r = f(x, y) = x + y$: how can we design a Boolean function f whose output represents the sum of x and y? Often you already known of long-hand or "school-book" techniques for operations such as addition and multiplication. So a general strategy for designing such an f is to first recap on your intuition about how the operation works, then formalise this as an algorithm, and finally design a circuit to implement the algorithm (often using 1-bit building blocks, and extending these to cope with n-bit inputs and outputs). Beyond this, the challenge (and arguably the fun) is multiplied by what is a large **design space** of options and trade-offs. For example, we might implement f using only combinatorial components, or widen this remit by considering sequential components that support state and so on. With any strategy involving a trade-off, the sanity of opting for



(a) *An unintegrated architecture: each i-th sub-component C_i deals with all of a different operation.*



(b) *An integrated architecture: each i-th sub-component C_i deals with a different part (e.g., i-th bit of the output) of all operations.*

Figure 1: *Two high-level ALU architectures: each combines a number of sub-components, but does so using a different strategy.*

one option over another requires careful analysis of the context: in some situations a given option might represent a good choice because it matches the requirements, other times not.

One can view a concrete ALU at two levels: at a low level in terms of how the components themselves are designed, and at a higher level (i.e., the ALU architecture) in terms of how the components are organised. Our focus in this Chapter is primarily the former, but we start with a brief overview of the latter. To aid discussion, two simplifications are made here:

1. We focus on small scale examples throughout, considering integers represented in two's-complement (where signed) using say $n = 8$ bits. It is important to see, however, that the techniques are presented in a general form: one can extend them to cope with larger n.

2. We focus on integer arithmetic only: although a given ALU may be capable of performing floating-point arithmetic, we ignore this feature.

Additionally, we use some extra notation to clarify when operations are signed or unsigned: given some operation $\odot$, we use $\odot_s$ and $\odot_u$ to denote signed and unsigned versions respectively. For example, $x <_s y$ denotes signed less than comparison while $x <_u y$ denotes the unsigned alternative. With no annotation of this type, you can assume the signed'ness of the operator is irrelevant.

2 High-level ALU architecture

As the name suggests, a typical ALU will perform roughly three classes of operation: arithmetic, logical (typically focused on operations involving individual bits, contrasting with arithmetic operations on representations of integers using multiple bits), and comparison. Although a given ALU is often viewed as a single unit, having a separate ALU for each class can have advantages. For example, this allows different classes of operation (e.g., an addition and comparison) to be performed at the same time. To prevent a single unit becoming too complex, it can also be advantageous to have separate ALUs for different classes of input; a key example is use of a dedicated **Floating-Point Unit (FPU)** to operate on floating-point rather than integer inputs.

These possibilities aside, at a high-level an ALU is simply a collection of sub-components; we provide one or more inputs (wlog. say x and y), and control it using op to select the operation required. Of course, some operations will produce a different sized output than others: an $(n \times n)$-bit multiplication produces a $2n$-bit output, but any comparison will only ever produce a 1-bit output for example. One can therefore view the ALU as conceptually producing a single output r, but in reality it might have *multiple* outputs that are used as and when appropriate. To be concrete, imagine we want an ALU which performs say $m = 11$ different operations

$$\odot \in \{+, -, \cdot, \wedge, \vee, \oplus, \overline{\vee}, \ll, \gg, =, <\}$$

meaning it can perform addition, subtraction, multiplication, a range of bit-wise Boolean operations (AND, OR, XOR and NOR), left- and right-shift, and two comparisons (equality and less than): it computes $r = x \odot y$ for an $\odot$ selected by op. Figure 1 shows two strategies for realising the ALU, each using sub-components (the i-th of which is denoted C_i) of a different form and in a different way:

1. Figure 1a illustrates an architecture where each sub-component implements all of a different operation. For example, C_0 and C_1 might compute all n bits of $x + y$ and $x - y$ respectively; the ALU output is selected, from the m sub-component outputs, using op to control a suitable multiplexer.

   Although, as shown, each sub-component is always active, in reality it might be advantageous to power-down a sub-component which is not being used. This could, for example, reduce power consumption or heat dissipation.

2. Figure 1b illustrates an architecture where each sub-component implements all operations, but does so wrt. a single bit only. For example, C_0 and C_1 might compute the 0-th and 1-st bits of $x + y$ and $x - y$ respectively (depending on op).

Tanenbaum and Austin [**?**, Chapter 3, Figures 3-18/3-19] focus on the second strategy, discussing a 1-bit ALU slice before dealing with their combination. Such 1-bit ALUs are often available as standard building blocks, so this focus makes a lot of sense on one hand. On the other hand, an arguable disadvantage is that such a focus complicates the overarching study of computer arithmetic. Put another way, focusing at a low-level on 1-bit ALU slices arguably makes it hard(er) to see how some higher-level arithmetic works. As a result, we focus instead on the first strategy in what follows: we consider designs for each i-th sub-component, realising each operation (a C_i for addition, for example) in isolation.

Essentially this means we ignore high-level organisation and optimisation of the ALU from here on, but of course both strategies have merits. For example, as we will see in the following, overlap exists between different arithmetic circuit designs: intuitively, the computation of addition and subtraction is similar for example. The second strategy is advantageous therefore, since said overlap can more easily be capitalised upon to reduce overall gate count. However, arithmetic circuits that require multiple steps to compute an output (using an FSM for example) are hard(er) to realise using the second strategy than the first. As a result, a mix of *both* strategies as and when appropriate is often a reasonable compromise.

3 Components for addition and subtraction

Perhaps more so than other aspects of computer arithmetic, the meaning and use of addition and subtraction should be familiar. (Very) formally, an addition operation computes the **sum** $r = x + y$ using an x and y which are both termed an **addend** in this context; likewise, subtraction computes the **difference** $r = x - y$ using a **minuend** x and a **subtrahend** y. This terminology hints at the fact that addition is commutative but subtraction is not: changing the order of x and y in the latter case changes the result we get.

The challenge of course is *how* we compute these results. The goal in each case is to first describe the computation algorithmically, then translate this into a design (or set of designs) for a circuit we can construct from logic gates.

3.1 Addition

First some good news: you already know to add together integers. Put simply, if we use base-10 then you have almost certainly seen something like

$$\begin{array}{rclcccccc}
x & = & 107_{(10)} & \mapsto & & 1 & 0 & 7 & \\
y & = & 14_{(10)} & \mapsto & & 0 & 1 & 4 & + \\
\cline{5-8}
c & = & & & 0 & 0 & 1 & 0 & \\
r & = & 121_{(10)} & \mapsto & & 1 & 2 & 1 & \\
\cline{5-8}
\end{array}$$

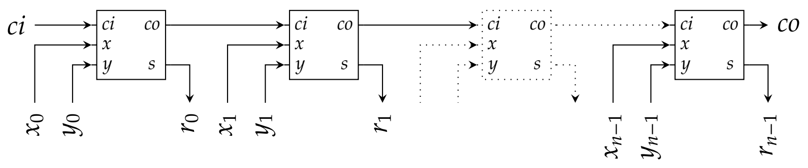


Figure 2: *An n-bit, ripple-carry adder described using a circuit diagram.*

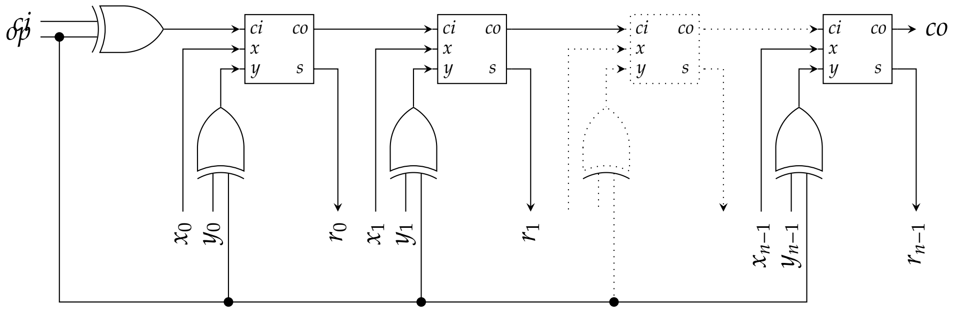


Figure 3: *An n-bit, ripple-carry adder/subtractor described using a circuit diagram.*

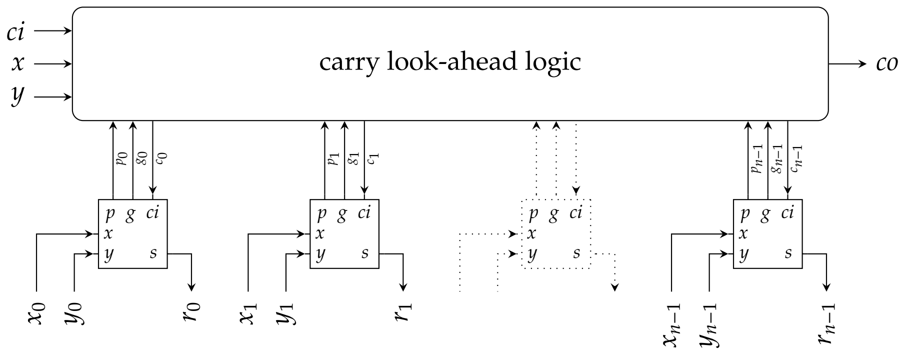


Figure 4: *An n-bit, carry look-ahead adder described using a circuit diagram.*

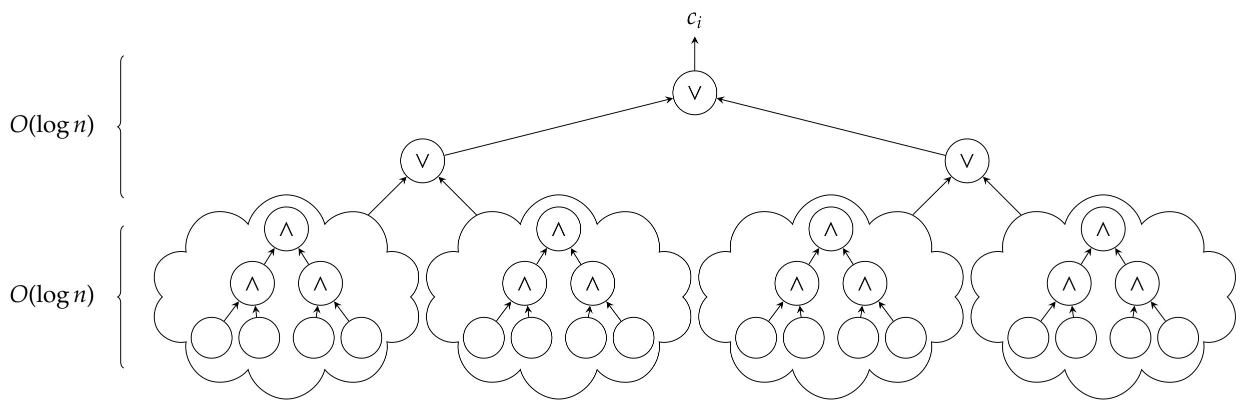


Figure 5: *An illustration depicting the structure of carry look-ahead logic, which is formed by an upper- and lower-tree of OR and AND gates respectively (with leaf nodes representing g_i and p_i terms for example).*

Input: Two unsigned, n-digit, base-b integers x and y, and a 1-digit carry-in ci
Output: An unsigned, n-digit, base-b integer $r = x + y$, and a 1-digit carry-out co

```
1 $r \leftarrow 0, c_0 \leftarrow ci$
2 for $i = 0$ upto $n - 1$ step $+1$ do
3     $r_i \leftarrow (x_i + y_i + c_i) \bmod b$
4     if $(x_i + y_i + c_i) < b$ then $c_{i+1} \leftarrow 0$ else $c_{i+1} \leftarrow 1$
5 end
6 $co \leftarrow c_n$
7 return $r, co$
```

Algorithm 1: An algorithm for addition of base-b integers.

Input: Two unsigned, n-digit, base-b integers x and y, and a 1-digit borrow-in bi
Output: An unsigned, n-digit, base-b integer $r = x - y$, and a 1-digit borrow-out bo

```
1 $r \leftarrow 0, c_0 \leftarrow bi$
2 for $i = 0$ upto $n - 1$ step $+1$ do
3     $r_i \leftarrow (x_i - y_i - c_i) \bmod b$
4     if $(x_i - y_i - c_i) \geq 0$ then $c_{i+1} \leftarrow 0$ else $c_{i+1} \leftarrow 1$
5 end
6 $bo \leftarrow c_n$
7 return $r, bo$
```

Algorithm 2: An algorithm for subtraction of base-b integers.

used to illustrate[1] the strategy (or at least the result *from* it) for n-digit integers x and y. The strategy is to work from the least-significant, right-most digits (i.e., x_0 and y_0) towards the most-significant, left-most digits (i.e., x_{n-1} and y_{n-1}). At each i-th step (or column), we sum the i-th digits x_i and y_i *and* a **carry-in** produced by the previous, $(i-1)$-th step; since this sum is potentially larger than a single base-b digit is allowed to be, we produce the i-th digit of the result *and* a **carry-out** into the next, $(i+1)$-th step.

This can be written more formally as Algorithm 1: the loop in lines #2 to #5 captures the idea of working through digits of x and y, in each step computing r_i and c_{i+1} from x_i, y_i and c_i. In particular, lines #3 and #4 compute r_i and c_i respectively; you can read the latter as "if the sum of x_i, y_i and c_i is smaller than one digit there is a carry into the next step, otherwise there is no carry" for example. Using the same x and y, a trace of algorithm invocation

i	x_i	y_i	c_i	r	$x_i + y_i + c_i$	c_{i+1}	r_i	r'
				$\langle 0,0,0\rangle$				$\langle 0,0,0\rangle$
0	7	4	0	$\langle 0,0,0\rangle$	11	1	1	$\langle 1,0,0\rangle$
1	0	1	1	$\langle 1,0,0\rangle$	2	0	2	$\langle 1,2,0\rangle$
2	1	0	0	$\langle 1,2,0\rangle$	1	0	1	$\langle 1,2,1\rangle$
			0	$\langle 1,2,1\rangle$				

illustrates[2] how it deals with the original example.

We call c a **carry chain** and say that carries propagate from one step to the next: the algorithm, but more so the circuit which results from it, is termed a **ripple-carry adder** since the carry ripples through this chain. Notice that the algorithm sets $c_0 = ci$ to allow a carry-into the overall operation (in the examples we assume this is 0), and $co = c_n$ allowing a carry-out. Also notice that we have written the algorithm in terms of general, base-b integers: this is deliberate of course, and means it will also work in base-2. Using the same values, but now written using binary so $x = 107_{(10)} = 01101011_{(2)}$ and $y = 14_{(10)} = 00001110_{(2)}$, we have

$$
\begin{array}{lclcccccccccccc}
x & = & 107_{(10)} & \mapsto & & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & \\
y & = & 14_{(10)} & \mapsto & & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & + \\
\cline{5-13}
c & = & & & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & \\
r & = & 121_{(10)} & \mapsto & & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & \\
\cline{5-13}
\end{array}
$$

[1]Note that to make x and y the same length in this and following examples, we sometimes pad them with more-significant zero digits; this might look odd however, so keep in mind that you can safely ignore them without altering the associated value.

[2] All similar traces in this Chapter, particularly if the associated algorithm includes a loop, are of roughly the same format: read from left-to-right, there is typically a section of loop counters, such as i and j, a section of variables as they are at the start of each iteration, a section of variables computed during an iteration, and a section of variables as they are at the end of each iteration. If a variable t in the left-hand section is updated during an iteration, we write it as t' (read as "the new value of t") in the right-hand section.

for example, with the corresponding trace as follows:

i	x_i	y_i	c_i	r	$x_i + y_i + c_i$	c_{i+1}	r_i	r'
				$\langle 0,0,0,0,0,0,0,0 \rangle$				$\langle 0,0,0,0,0,0,0,0 \rangle$
0	1	0	0	$\langle 0,0,0,0,0,0,0,0 \rangle$	1	0	1	$\langle 1,0,0,0,0,0,0,0 \rangle$
1	1	1	0	$\langle 1,0,0,0,0,0,0,0 \rangle$	2	1	0	$\langle 1,0,0,0,0,0,0,0 \rangle$
2	0	1	1	$\langle 1,0,0,0,0,0,0,0 \rangle$	2	1	0	$\langle 1,0,0,0,0,0,0,0 \rangle$
3	1	1	1	$\langle 1,0,0,0,0,0,0,0 \rangle$	3	1	1	$\langle 1,0,0,1,0,0,0,0 \rangle$
4	0	0	1	$\langle 1,0,0,1,0,0,0,0 \rangle$	1	0	1	$\langle 1,0,0,1,1,0,0,0 \rangle$
5	1	0	0	$\langle 1,0,0,1,1,0,0,0 \rangle$	1	0	1	$\langle 1,0,0,1,1,1,0,0 \rangle$
6	1	0	0	$\langle 1,0,0,1,1,1,0,0 \rangle$	1	0	1	$\langle 1,0,0,1,1,1,1,0 \rangle$
7	0	0	0	$\langle 1,0,0,1,1,1,1,0 \rangle$	0	0	0	$\langle 1,0,0,1,1,1,1,0 \rangle$
			0	$\langle 1,0,0,1,1,1,1,0 \rangle$				

With any choice of b, the sum of two n-digit integers is an $(n + 1)$-digit result: the algorithm produces an n-digit result r and separate 1-digit carry-out co, but you can of course think of them as two parts of the same, single result.

3.1.1 Ripple-carry adders

Now we have an algorithm, the next challenge is to translate it into a more concrete design we can then implement as a circuit. At first glance this may seem difficult because the algorithm contains a loop. Crucially however, we can unroll this loop once n is fixed: all this means is that we copy and paste the loop body (i.e., lines #3 and #4) n times, replacing i with the right value in each i-th copy. Given $n = 4$, for example, we know the loop can be unrolled into the straight-line alternative

```
1  $c_0 \leftarrow ci$
2  $r_0 \leftarrow (x_0 + y_0 + c_0) \bmod b$
3  **if** $(x_0 + y_0 + c_0) < b$ **then** $c_1 \leftarrow 0$ **else** $c_1 \leftarrow 1$
4  $r_1 \leftarrow (x_1 + y_1 + c_1) \bmod b$
5  **if** $(x_1 + y_1 + c_1) < b$ **then** $c_2 \leftarrow 0$ **else** $c_2 \leftarrow 1$
6  $r_2 \leftarrow (x_2 + y_2 + c_2) \bmod b$
7  **if** $(x_2 + y_2 + c_2) < b$ **then** $c_3 \leftarrow 0$ **else** $c_3 \leftarrow 1$
8  $r_3 \leftarrow (x_3 + y_3 + c_3) \bmod b$
9  **if** $(x_3 + y_3 + c_3) < b$ **then** $c_4 \leftarrow 0$ **else** $c_4 \leftarrow 1$
10 $co \leftarrow c_4$
```

The key thing to notice is that the body of the loop, and hence each replicated step in the unrolled alternative, does the same thing as a full-adder cell: it computes the 1-bit addition

$$\begin{array}{rcl} r_i & = & x_i \oplus y_i \oplus c_i \\ c_{i+1} & = & (x_i \wedge y_i) \vee (x_i \wedge c_i) \vee (y_i \wedge c_i) \end{array}$$

for which we already have a component. As Figure 2 shows, the circuit is then basically just n full-adder instances connected via their carry-in and carry-out input and outputs: each adds i-th digits of x and y to a carry-in c_i, and produces the i-th digit of the result r and a carry-out c_{i+1}. Even better, given the discussion in Chapter 1 this component can deal with signed values of x and y with no change: provided we use two's-complement, the rules for addition are exactly the same.

So given this design already solves the problem of adding x to y, why might we bother with *other* designs? Although any metric can provide general motivation, the concept of critical path is important here. If you think about it, in this design r_{n-1} cannot be computed without knowing the carry-in c_{n-1}. More generally, the carry chain imposes an order on computation of the digits in r: a given r_i cannot be computed until all j-th steps for $j < i$ are computed, meaning the critical path of the design *is* the carry chain. The overall delay can be approximated by $O(n)$ gate delays since the carry chain runs through n full-adder instances: one question then, is whether we can improve on this somehow?

3.1.2 Carry look-ahead adders

One idea to "break" the carry chain, is to separate the computation of carries from the sum. At first glance this may seem impossible, but the key thing to notice is that we can say at least *something* about how the i-th stage of the ripple-carry adder works in isolation. We know for instance that

1. if $x_i + y_i > b - 1$ it *generates* a carry,

2. if $x_i + y_i = b - 1$ it *propagates* a carry, and
3. if $x_i + y_i < b - 1$ it *absorbs* a carry.

Consider a slightly contrived example

$$\begin{array}{rclcrrrrl} x & = & 456_{(10)} & \mapsto & & 4 & 5 & 6 & \\ y & = & 444_{(10)} & \mapsto & & 4 & 4 & 4 & + \\ \cline{5-8} c & = & & & 0 & 1 & 1 & 0 & \\ r & = & 900_{(10)} & \mapsto & & 9 & 0 & 0 & \\ \cline{5-8} \end{array}$$

using base-10 to make things easier, where the three rules apply as follows:

1. In the 0-th column, $x_i + y_i = x_0 + y_0 = 6 + 4 = 10$ which is greater than $b - 1 = 10 - 1 = 9$. Put another way, this is already too large to represent using a single base-b digit and will hence always generates a carry into the next, $(i + 1)$-th step irrespective of whether there is a carry-in or not.
2. In the 1-st column, $x_i + y_i = x_1 + y_1 = 5 + 4 = 9$ which is equal to $b - 1 = 10 - 1 = 9$. Put another way, this is at the cusp of what we can represent using a single base-b digit: iff. there is a carry-in, then there will be a carry-out.
3. In the 2-nd column, $x_i + y_i = x_2 + y_2 = 4 + 4 = 8$ which is less than $b - 1 = 10 - 1 = 9$. Put another way, even if there is a carry into the i-th stage there will never be a carry-out because $8 + 1$ can be accommodated within the single base-b digit r_2 of the sum.

A **Carry Look-Ahead (CLA) adder** takes advantage of the fact that moving to base-2 makes application of the rules fairly simple. In particular, imagine g_i and p_i tell us whether the i-th stage will generate or propagate a carry respectively. We can write these as

$$\begin{array}{rcl} g_i & = & x_i \wedge y_i \\ p_i & = & x_i \oplus y_i \end{array}$$

which can be explained in words:

- we generate a carry-out if *both* $x_i = 1$ *and* $y_i = 1$ since no matter what the carry-in is, their sum cannot be represented in a single base-b digit, and
- we propagate a carry-out if *either* $x_i = 1$ *or* $y_i = 1$ since this plus any carry-in will also produce a sum which cannot be represented in a single base-b digit.

Given g_i and p_i we have that

$$c_{i+1} = g_i \vee (c_i \wedge p_i)$$

where, again, $c_0 = ci$ and we produce a carry-out $c_n = co$. Again this can be explained in words: at the i-th stage "there is a carry-out if either the i-th stage generates a carry itself, or there is a carry-in and the i-th stage will propagate it". As an aside, note that it is common to see g_i and p_i written as

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

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.

Like the ripple-carry adder, once we fix n, we can unwind the recursion to get an expression for the carry into each i-th full-adder cell:

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

This *looks* horrendous, but notice that the general structure is of the form illustrated in Figure 5: both the bottom- and top-half are balanced binary trees (where leaf nodes are g_i and p_i terms, and internal nodes are AND and OR gates respectively) implementing the SoP expression for a a given c_i.

We can arrange things in this way basically because of having decoupled computation of c_i from the corresponding r_i. This is what gives carry look-ahead adders an advantage: the critical path (i.e., the depth of the structure, or longest path from the root to some leaf) is actually *shorter* than for a ripple-carry adder. The former is described by $O(\log n)$ gate delays (as a result of the tree structure) and the latter by $O(n)$ (as a result of the linear sequence).

The resulting circuit is shown in Figure 4. In contrast with the ripple-carry adder, each full-adder instance is now independent (in the sense the carry chain is not there). Rather, each i-th instance produces g_i and p_i and provides this as input to the carry look-ahead logic, which generates c_i for it. Of course said logic hides a clear trade-off involved with this approach, namely the associated gate count is much greater (a rough estimate would be $O(n)$ and $O(n^2)$ gates for a ripple-carry and carry look-ahead adders respectively). As a compromise therefore, 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.

3.1.3 Carry-save adders

A second approach to eliminating the carry chain is to bend the rules a bit, and look at a slightly different problem. The ripple-carry and the carry look-ahead adder compute the sum of two inputs x and y; what happens if we consider *three* inputs x, y and z and hence try to compute $x + y + z$ rather than $x + y$? A **carry-save adder** deals with this problem. This type of adder is often termed a $3:2$ **compressor** since we actually *compress* the three n-bit inputs x, y and z into *two* n-bit outputs r' and c' (called the **partial sum** and **shifted carry**). That is, we compute the actual sum r in two steps: a compression step that produces the partial sum and shifted carry, and an addition step that combines them into the actual sum.

The first step is achieved by replacing c with z in the ripple-carry adder, meaning

$$\begin{array}{rcl} r'_i & = & x_i \oplus y_i \oplus z_i \\ c'_i & = & (x_i \wedge y_i) \vee (x_i \wedge z_i) \vee (y_i \wedge z_i) \end{array}$$

The key thing to notice is that unlike the ripple-carry adder, where the i-th step made reference to the $(i+1)$-th step via c_{i+1}, the two expressions for r'_i and c'_i *only* use x_i, y_i and z_i. That is, each i-th digit of r' and c' can be computed independently from and hence in parallel with the others; this fact is what improves on the ripple-carry adder limitation, imposed by the propagation of carries from one step to the next. For example, if we set $x = 96_{(10)} = 01100000_{(2)}$, $y = 14_{(10)} = 00001110_{(2)}$ and $z = 11_{(10)} = 00001011_{(2)}$, then we find

$$\begin{array}{rclccccccccc} x & = & 96_{(10)} & \mapsto & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ y & = & 14_{(10)} & \mapsto & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ z & = & 11_{(10)} & \mapsto & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ & & & & & & & & & & & \\ r' & = & & & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ c' & = & & & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \end{array}$$

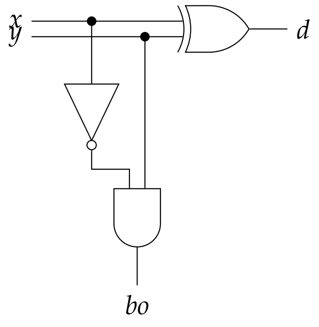
After computing r' and c' we then combine them via a second, addition step. More specifically, we compute $r = r' + 2 \cdot c'$ using a standard adder, i.e., a ripple-carry adder or similar:

$$\begin{array}{rclcccccccccccc} r' & = & & & & & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & \\ 2 \cdot c' & = & & & & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & + \\ \hline c & = & & & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & \\ r & = & 121_{(10)} & \mapsto & & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & \\ \hline \end{array}$$

You can think of this step as propagating all the carries now represented separately by c', which of course was the original problem. So given we need to do this second step it is reasonably to question why we bother with the first step at all?! With $m = 1$ compression step, the value is indeed unclear. The more general idea however, is we compute *many* compression steps (i.e., $m > 1$) and then a *single*, final addition step. If we do this, the cost associated with the addition step becomes less and less significant as m grows larger. Later, in Section 5 when we look at multiplication, examples of exactly this type of scenario become clear.

HALF-SUBTRACTOR			
x	y	bo	d
0	0	0	0
0	1	1	1
1	0	0	1
1	1	0	0

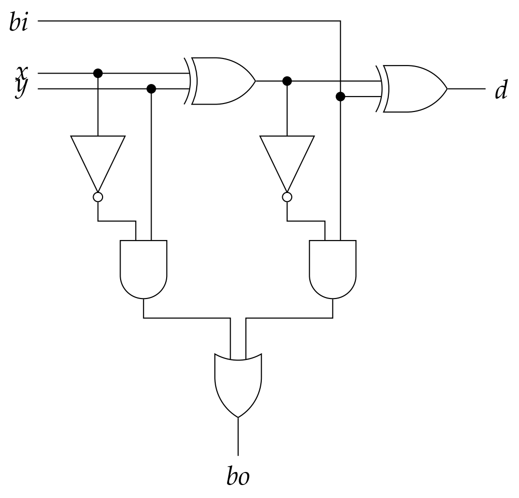
(a) *The half-subtractor as a truth table.*



(b) *The half-subtractor as a circuit.*

FULL-SUBTRACTOR				
bi	x	y	bo	d
0	0	0	0	0
0	0	1	1	1
0	1	0	0	1
0	1	1	0	0
1	0	0	1	1
1	0	1	1	0
1	1	0	0	0
1	1	1	1	1

(c) *The full-subtractor as a truth table.*



(d) *The full-subtractor as a circuit.*

Figure 6: *An overview of half- and full-subtractor cells.*

3.2 Subtraction

3.2.1 Re*designing* a ripple-carry adder

Adopting the same underlying idea as for addition, the first strategy would be to formalise something like

$$\begin{array}{rcrcrrrrl}
x & = & 107_{(10)} & \mapsto & & 1 & 0 & 7 & \\
y & = & 14_{(10)} & \mapsto & & 0 & 1 & 4 & - \\
\cline{5-8}
c & = & & & 0 & 1 & 0 & 0 & \\
r & = & 93_{(10)} & \mapsto & & 0 & 9 & 3 & \\
\cline{5-8}
\end{array}$$

into an Algorithm 2. In essence, the same steps are evident during subtraction as addition: we again work from the least-significant, right-most digits (i.e., x_0 and y_0) towards the most-significant, left-most digits (i.e., x_{n-1} and y_{n-1}). At each i-th step (or column), we now compute the different of the i-th digits x_i and y_i *and* a **borrow-in** produced by the previous, $(i-1)$-th step; since this difference is potentially smaller than zero, we produce the i-th digit of the result *and* a **borrow-out** into the next, $(i+1)$-th step. Using the same x and y, a trace of algorithm invocation

i	x_i	y_i	c_i	r	$x_i - y_i - c_i$	c_{i+1}	r_i	r'
				$\langle 0,0,0\rangle$				$\langle 0,0,0\rangle$
0	7	4	0	$\langle 0,0,0\rangle$	3	0	3	$\langle 3,0,0\rangle$
1	0	1	0	$\langle 0,0,0\rangle$	-1	1	9	$\langle 3,9,0\rangle$
2	1	0	1	$\langle 0,0,0\rangle$	0	0	0	$\langle 3,9,0\rangle$
			0	$\langle 3,9,0\rangle$				

illustrates how it deals with the original example. Note that although the name c is slightly counter-intuitive (it now represents a borrow- rather than carry-chain), we stick to the same notation as an adder to stress the similar strategy.

So if the algorithm is more or less the same, it follow that a circuit to implement it would *also* be the same apart from the loop body: here, we need the subtraction equivalent to half- and full-adder cells. More specifically, a half-*subtractor* takes two 1-bit values, say x and y, and subtracts one from the other to produce a difference and a borrow-out, say d and bo; a full-subtractor extends this by including a borrow-in bi as an additional input. Unsurprisingly, Figure 6 demonstrates that the components themselves are simple to design, and also write as the Boolean expressions

$$\begin{array}{rcrcl} bo & = & \neg x & \wedge & y \\ d & = & x & \oplus & y. \end{array}$$

and

$$\begin{array}{rcl} bo & = & (\neg x \wedge y) \vee (\neg(x \oplus y) \wedge bi) \\ d & = & x \oplus y \oplus bi \end{array}$$

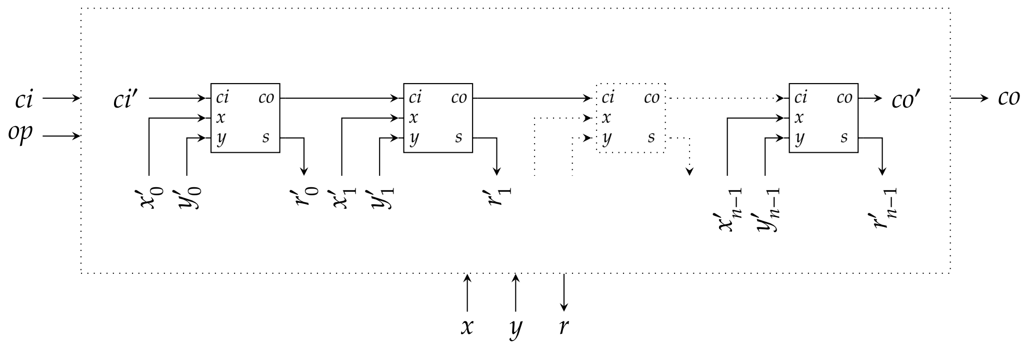
respectively. Keep in mind that bi and bo perform the same role as ci and co previously: the subtraction analogue of a ripple-carry adder, an n-bit ripple-*borrow* subtractor perhaps, is exactly the same except there is now a borrow chain through all n full-subtractor instances.

3.2.2 Re*using* a ripple-carry adder

As we have seen, subtraction is already similar to addition. This is more obvious still if we write $x - y \equiv x + (-y)$, meaning the subtraction we want (on the LHS) can be computed by adding x to the negation of y (on the RHS). Given we do an addition in both cases, we might opt for a second approach by designing a single component that allows selection of either addition *or* subtraction: given a control signal op, we want a result

$$r = \begin{cases} x + y + ci & \text{if } op = 0 \\ x - y - ci & \text{if } op = 1 \end{cases},$$

Notice that as well as controlling computation of the sum or difference of x and y, op controls use of ci as carry- or borrow-in depending if we want to compute an addition of subtraction. The main advantage of this approach is that at a high-level, the design will look as follows



with a single, internal adder; this saving (versus two separate and fairly similar components) is useful outright, but also in discussion of multiplier designs where the ability to do either operation can be capitalised upon.

The question is then how to control the internal inputs to the adder (namely x', y' and ci') so given all the external inputs (namely op, x, y and ci), the correct output r is produced? By using two's-complement representation we can show that

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

for any given y. Put more simply, to negate y we invert each bit y_i via a NOT gate, then add 1 to the result. The idea is to make use of this identity via simple translation from *what* we want to compute into what we already *can* compute:

op	ci	r
0	0	$x + y + ci$
0	1	$x + y + ci$
1	0	$x - y - ci$
1	1	$x - y - ci$

$\equiv$

op	ci	r
0	0	$x + y + 0$
0	1	$x + y + 1$
1	0	$x - y - 0$
1	1	$x - y - 1$

$\equiv$

op	ci	r
0	0	$x + y + 0$
0	1	$x + y + 1$
1	0	$x + (\neg y + 1) - 0$
1	1	$x + (\neg y + 1) - 1$

$\equiv$

op	ci	r
0	0	$x + y + 0$
0	1	$x + y + 1$
1	0	$x + (\neg y) + 1$
1	1	$x + (\neg y) + 0$

The left-most table illustrates the fact (from above) that when $op = 0$ we want to produce the result $x + y + ci$, while setting $op = 1$ changes this to $x - y - ci$. Moving from left-to-right, we then simply substitute in values of ci and apply our identity in the bottom cases were $-y$ is required; finally, the right-most table folds all the constants together. In this final table, all cases (for addition *and* subtraction) are of the same form in the sense they all add together three operands, which of course we can cope with using the internal adder: we have op, x, y and ci as input so we can just translate via

op	ci	x_i	y_i	ci'	x'_i	y'_i
0	0	0	0	0	0	0
0	1	1	1	1	1	1
1	0	0	0	1	0	1
1	1	1	1	0	1	0

i.e., $ci' = ci \oplus op$, $x'_i = x_i$ and $y'_i = y_i \oplus op$. Put another way, x remains unchanged while each y_i and ci are XOR'ed with op in order to conditionally invert them (in the bottom two, subtraction cases). Figure 3 illustrates the result, where it is important to see that the overhead, versus in this case a ripple-carry adder, is simply $n + 1$ XOR gates.

3.3 Carry and overflow detection

Consider the addition of some n-bit inputs x and y: the magnitude of a result is too large to represent in n bits if either

1. the operands are unsigned and there is a carry-out, *or*
2. the operands are signed but the sign of the result makes no sense

which are termed **carry** and **overflow** error conditions respectively. Detecting the carry condition is simple. Imagine we have an example where $n = 4$ and we add together unsigned x and y using a ripple-carry adder:

$$\begin{array}{rclcccccccl}
x & = & 15_{(10)} & \mapsto & & 1 & 1 & 1 & 1 & \\
y & = & 1_{(10)} & \mapsto & & 0 & 0 & 0 & 1 & + \\
\hline
c & = & & & 1 & 1 & 1 & 1 & 0 & \\
r & = & 0_{(10)} & \mapsto & & 0 & 0 & 0 & 0 & \\
\hline
\end{array}$$

The *correct* result $r = 16$ has a magnitude which cannot be accommodated in the number of bits available; an incorrect result $r = 0$ is produced, with the carry-out signalling the associated error. If we use signed x and y however, any carry-out is irrelevant. For example, using a similar example and again using a ripple-carry adder directly

$$\begin{array}{rclcccccccl}
x & = & -1_{(10)} & \mapsto & & 1 & 1 & 1 & 1 & \\
y & = & 1_{(10)} & \mapsto & & 0 & 0 & 0 & 1 & + \\
\hline
c & = & & & 1 & 1 & 1 & 1 & 0 & \\
r & = & 0_{(10)} & \mapsto & & 0 & 0 & 0 & 0 & \\
\hline
\end{array}$$

the carry-out should simply be discarded: the result $r = 0$ is now correct. So if this is true, what *does* signal the occurrence of an overflow? Again consider an example with signed x and y:

$$\begin{array}{rclcccccccl}
x & = & 7_{(10)} & \mapsto & & 0 & 1 & 1 & 1 & \\
y & = & 1_{(10)} & \mapsto & & 0 & 0 & 0 & 1 & + \\
\hline
c & = & & & 0 & 1 & 1 & 1 & 0 & \\
r & = & -8_{(10)} & \mapsto & & 1 & 0 & 0 & 0 & \\
\hline
\end{array}$$

In this case, x is the largest positive integer we can represent using $n = 4$ bits; adding $y = 1$ using the ripple-carry adder means the value wraps-around (as discussed in Chapter 1), forming a negative result $r = -8$. Clearly this is impossible in that if we have a positive x and y, we can *never* end up with a negative sum: this mismatch allows us to conclude than an overflow error occurred. More specifically, in the case of addition, we apply the following set of rules (with a similar set possible for subtraction):

x +ve	y -ve		$\Rightarrow$	no overflow
x -ve	y +ve		$\Rightarrow$	no overflow
x +ve	y +ve	r +ve	$\Rightarrow$	no overflow
x +ve	y +ve	r -ve	$\Rightarrow$	overflow
x -ve	y -ve	r +ve	$\Rightarrow$	overflow
x -ve	y -ve	r -ve	$\Rightarrow$	no overflow

How? Notice that testing the sign of x or y is trivial: their MSBs x_{n-1} and y_{n-1} determines this because of how two's-complement is defined, implying for example that x is positive iff. $x_{n-1} = 0$ and negative iff. $x_{n-1} = 1$. Based on this, the overflow condition is computed as

$$\begin{array}{rcllllll} of & = & (& x_{n-1} & \wedge & y_{n-1} & \wedge & \neg r_{n-1} &) & \vee \\ & & (& \neg x_{n-1} & \wedge & \neg y_{n-1} & \wedge & r_{n-1} &) & \end{array}$$

or in words: "there is an overflow if either x is positive and y is positive and r is negative, or if x is negative and y is negative and r is positive". This can be further simplified to

$$of = c_{n-1} \oplus c_{n-2}$$

where c is the carry chain during addition of x and y: basically this XORs the carry-in and the carry-out of the $(n-1)$-th full-adder. As such, an overflow is signalled, i.e., $of = 1$, in two cases: either

1. $c_{n-1} = 0$ and $c_{n-2} = 1$, which can only occur of $x_{n-1} = 0$ and $y_{n-1} = 0$ (i.e., x and y are both positive but r is negative), or
2. $c_{n-1} = 1$ and $c_{n-2} = 0$, which can only occur of $x_{n-1} = 1$ and $y_{n-1} = 1$ (i.e., x and y are both negative but r is positive).

Once an error condition is detected (during a relevant operation by the ALU, for example), the next question is what to do about it: clearly the error needs to be managed somehow, or the incorrect result will be used as normal. There are numerous options, but two in particular illustrate their general approach:

1. provide the incorrect result as normal, (e.g., truncate the result to n bits by discarding bits we cannot accommodate), but signal the error condition somehow (e.g., via a status register or some form of exception), or
2. fix the incorrect result somehow, according to pre-planned rules (e.g., **saturate** or **clamp** the result to the largest integer we can represent in n bits).

In short, the choice is between delegating responsibility to whatever is using the ALU (in the former) and making the ALU itself responsible (in the latter); both have advantages and disadvantages, and may therefore be appropriate in different situations.

4 Components for shift and rotation

4.1 Introductory theory

4.1.1 Phrasing shift operations as Mathematics

Although one would not normally think of doing a long-hand **shift**, as with addition or subtraction, it is possible to consider an operation of this type in Mathematical terms: a shift of some base-b integer x by a distance of y digits has the same effect as multiplying x by b^y. That is,

$$\begin{array}{rcccl} r & = & x \cdot b^y & = & b^y \cdot \sum_{i=0}^{n-1} x_i \cdot b^i \\ & & & = & \sum_{i=0}^{n-1} x_i \cdot b^{i+y} \end{array}$$

Notice that if y is positive it increases the weight associated with a given digit x_i, hence shifting said digit to the left. In contrast, if y is negative then this decreases the weight associated with x_i and the digit shifts to the right; in this case you can think of the operation as a division instead, because clearly

$$x \cdot b^{-y} = x \cdot \frac{1}{b^y} = \frac{x}{b^y}.$$

Consider an example: using base-10, if we set $x = 123_{(10)}$ then x has $n = 3$ digits. If we select $y = 2$, then

$$\begin{array}{rcccl} r & = & x \cdot b^y & = & \sum_{i=0}^{n-1} x_i \cdot b^{i+y} \\ & & & = & x_0 \cdot b^{0+2} + x_1 \cdot b^{1+2} + x_2 \cdot b^{2+2} \\ & & & = & 3 \cdot 10^2 + 2 \cdot 10^3 + 1 \cdot 10^4 \\ & & & = & 300 + 2000 + 10000 \\ & & & = & 12300_{(10)} \end{array}$$

An aside: shift operators in C and Java.

The fact there are two different classes of shift operation needs care: in a given programming language, you need to make sure you use the right one for the right job! In C, left- and right-shifts use operators << and >> irrespective of whether they are arithmetic or logical; the type of the operand being shifted dictates the class of shift. For example

1. if `x` is of type `int` (i.e., `x` is a signed integer) then the expression `x >> 2` implies an arithmetic right-shift, whereas
2. if `x` is of type `unsigned int` (i.e., `x` is an unsigned integer) then the expression `x >> 2` implies a logical right-shift.

In contrast, Java has no unsigned integer data types so needs to take a different approach: arithmetic and logical shifts are specified by two different operators, meaning

1. the expression `x >> 2` implies an arithmetic right-shift. whereas
2. the expression `x >>> 2` implies a logical right-shift,

meaning we have multiplied x by $b^y = 10^2 = 100$ as expected. If we now change the example so $y = -2$, then

$$\begin{array}{rclcl} r & = & x \cdot b^y & = & \sum_{i=0}^{n-1} x_i \cdot b^{i+y} \\ & & & = & x_0 \cdot b^{0-2} + x_1 \cdot b^{1-2} + x_2 \cdot b^{2-2} \\ & & & = & 3 \cdot 10^{-2} + 2 \cdot 10^{-1} + 1 \cdot 10^0 \\ & & & = & 0.03 + 0.2 + 1 \\ & & & = & 1.23_{(10)} \end{array}$$

and again we have divided x by $b^{-y} = 10^{-(-2)} = 10^2 = 100$ as expected. Of course this works the same way for any b, and as you might expect we are interested in $b = 2$. Imagine we set $x = 110011_{(2)} = 51_{(10)}$. As before, using $y = 2$ multiplies x by $b^y = 2^2 = 4$ to give

$$\begin{array}{rclcl} r & = & x \cdot b^y & = & \sum_{i=0}^{n-1} x_i \cdot b^{i+y} \\ & & & = & x_0 \cdot b^{0+2} + x_1 \cdot b^{1+2} + x_2 \cdot b^{2+2} + x_3 \cdot b^{3+2} + x_4 \cdot b^{4+2} + x_5 \cdot b^{5+2} \\ & & & = & 1 \cdot 2^2 + 1 \cdot 2^3 + 0 \cdot 2^4 + 0 \cdot 2^5 + 1 \cdot 2^6 + 1 \cdot 2^7 \\ & & & = & 4 + 8 + 0 + 0 + 64 + 128 \\ & & & = & 11001100_{(10)} \\ & & & = & 204_{(10)} \end{array}$$

and using $y = -2$ multiplies x by $b^y = 2^{-2} = 0.25$, i.e., it divides x by 4, to give

$$\begin{array}{rclcl} r & = & x \cdot b^y & = & \sum_{i=0}^{n-1} x_i \cdot b^{i+y} \\ & & & = & x_0 \cdot b^{0-2} + x_1 \cdot b^{1-2} + x_2 \cdot b^{2-2} + x_3 \cdot b^{3-2} + x_4 \cdot b^{4-2} + x_5 \cdot b^{5-2} \\ & & & = & 1 \cdot 2^{-2} + 1 \cdot 2^{-1} + 0 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3 \\ & & & = & 0.25 + 0.5 + 0 + 0 + 4 + 8 \\ & & & = & 1100.11_{(2)} \\ & & & = & 12.75_{(10)} \end{array}$$

4.1.2 More concrete shifting and rotation of n-bit sequences

Recall from Chapter 1 that we can represent signed or unsigned integers using an n-bit sequence written as an actual sequence, or as a bit-literal. For example once we select an unsigned representation using $n = 8$ bits, each of the following

$$\begin{array}{rcl} x & = & 218_{(10)} \\ & = & 11011010_{(2)} \\ & \mapsto & \langle 0,1,0,1,1,0,1,1 \rangle \\ & \mapsto & 11011011 \end{array}$$

describes the same value; it turns out that using bit-literals within the following is more natural.

The idea is that based on our description the in previous Section, we need to consider what a shift operation means for integers represented by n-bit sequences. In particular, there is a problem: given our x and a choice of say $y = 2$, the left-shift $y = 2$ gives

$$r = x \ll 2 = 011010??$$

This illustrates two points:

1. we discarded two bits from the left-hand, most-significant end because they cannot be accommodated, plus
2. at the right-hand, less-significant end we need two extra but unknown bits (denoted ?)

since the result should still be $n = 8$ bits. Similar issues occur in the other direction when using right-shift: using $y = -2$ gives

$$r = x \gg 2 = ??110110$$

so this time we discarded two digits from the right-hand, least-significant end and need two extra digits at the left-hand, most-significant end. We cannot do much about the need to discard the MSBs (resp. LSBs) as a result of performing a left-shift (resp. right-shift): this is simply an artefact of using fixed-length n-bit sequences. However, we do need to solve the problem of how the gap formed in the LSBs (resp. MSBs) should be filled. The usual solution is to offer two *different* classes of shift operation:

1. **logical shift**, where left-shift discards MSBs and fills the gap in LSBs with zeros, and right-shift discards LSBs and fills the gap in MSBs with zeros, and
2. **arithmetic shift**, where left-shift discards MSBs and fills the gap in LSBs with zeros, and right-shift discards LSBs and fills the gap in MSBs with a sign bit.

Phrased in this way, a **rotate** operation (of some x by a distance of y) is the same as a logical shift except that any gap is filled with the other end of x rather than zero. Put another way, a left-rotate (resp. right-rotate) fills the LSBs (resp. MSBs) of r using the discarded MSBs (resp. LSBs) of x. Some examples of each operation perhaps make this clearer: for $n = 8$, again writing $x = 11011010_{(2)}$ as a bit-literal 11011010 and using $y = 2$, we see that

1. logical left- and right-shift produce

$$\begin{array}{rclcl} x \ll_u y & = & 11011010 \ll_u 2 & = & 01101000 \\ x \gg_u y & = & 11011010 \gg_u 2 & = & 00110110 \end{array}$$

2. arithmetic left- and right-shift produce

$$\begin{array}{rclcl} x \ll_s y & = & 11011010 \ll_s 2 & = & 01101000 \\ x \gg_s y & = & 11011010 \gg_s 2 & = & 11110110 \end{array}$$

   and

3. logical left- and right-rotate produce

$$\begin{array}{rclcl} x \lll y & = & 11011010 \lll 2 & = & 01101011 \\ x \ggg y & = & 11011010 \ggg 2 & = & 10110110 \end{array}$$

Logical and arithmetic left-shift are the same, so it is fairly reasonable to question the purpose of arithmetic right-shift. Recalling the general description above, the idea is that if x is signed (rather than unsigned, which is the case we started with) multiplication by b^y will ideally preserve this sign; the hint is that $\gg_u$ and $\gg_s$ are sort of unsigned and signed versions of right-shift. An example makes the issue clearer: selecting a signed representation means

$$\begin{array}{lclcl} -38_{(10)} & & & \mapsto & 11011010 \\ -38_{(10)}/2 & = & -19_{(10)} & \mapsto & 11101101 \end{array}$$

and right-shift by $y = 1$ bit *should* mean "divide by two". However, using logical right-shift, we actually get

$$\begin{array}{rclcl} r & = & x \gg_u y & = & 11011010 \gg_u 1 \\ & & & = & 01101101 \\ & & & \mapsto & 109_{(10)} \end{array}$$

```
Input: An $n$-bit sequence $x$, and an unsigned integer distance $0 \le y < n$
Output: The $n$-bit sequence $x \ll y$
$t \leftarrow x$
for $i = 0$ upto $y - 1$ step $+1$ do
    $t \leftarrow t \ll 1$
end
return $t$
```

Algorithm 3: An iterative algorithm for n-bit (left-)shift.

```
Input: An $n$-bit sequence $x$, and an unsigned integer distance $0 \le y < n$
Output: The $n$-bit sequence $x \ll y$
$t \leftarrow x$, $m \leftarrow \lceil \log_2(n) \rceil$
for $i = 0$ upto $m - 1$ step $+1$ do
    if $y_i = 1$ then
        $t \leftarrow t \ll 2^i$
    end
end
return $t$
```

Algorithm 4: An iterative algorithm for n-bit (left-)shift.

whereas if we use *arithmetic* right-shift we get

$$\begin{array}{rcrcl} r & = & x \gg_s y & = & 11011010 \gg_s 1 \\ & & & = & 11101101 \\ & & & \mapsto & -19_{(10)} \end{array}$$

as expected: in the former we fill the MSBs with zero which turns x into a positive r, in the later we fill the MSBs with the sign bit of x to preserves the sign[3] in r.

A crucial implication of this description is that if y is a known, fixed constant then shift and rotate operations require no actual arithmetic: we are simply moving bits in x left or right. As a result, a circuit that left- or right-shifts x by a fixed distance y simply connects wires from each x_i (or zero say, to fill LSBs or MSBs) to r_{i+y}. We can use this fact as a building block in more general circuits that can cater for scenarios where y is *not* fixed. Even then however, since left- and right-shift (or rotate, and of what ever class) are distinct operations we typically assume y is always positive, restricting it to $0 \le y < n$ meaning y has $m = \lceil \log_2(n) \rceil$ bits. We then have a choice of how to deal with a y outside this range. Typically, we either

1. let r be undefined for any $y > n$ or $y < 0$, or

2. consider y modulo n, so for example if $n = 8$ then a left-shift by $y = 9$ bits would be equivalent to a left-shift by $9 \bmod 8 \equiv 1$ bit.

4.2 Iterative designs

It should be fairly obvious that if $y = 6$, for example, then

$$\begin{array}{rcrcl} r & = & x \ll y & = & x \ll 6 \\ & & & = & (((((x \ll 1) \ll 1) \ll 1) \ll 1) \ll 1) \ll 1 \end{array}$$

Put another way, we can decompose computation of the large shift on the LHS into a series of smaller shifts on the RHS. The smaller shifts combine to produce the same result: six repeated shifts, each by a distance of one bit, produce the same result as one shift by a distance of six bits. This approach is formalised by

[3]This highlights the reason there is no need for special treatment of arithmetic left-shift. With right-shift we fill MSBs of, and hence dictate the sign bit of, the result; in contrast, with left-shift we fill LSBs in the result with the sign bit automatically preserved.

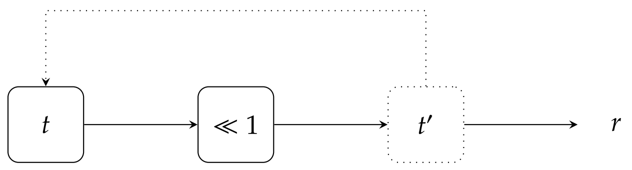


Figure 7: *An iterative design for n-bit (left-)shift described using a circuit diagram.*

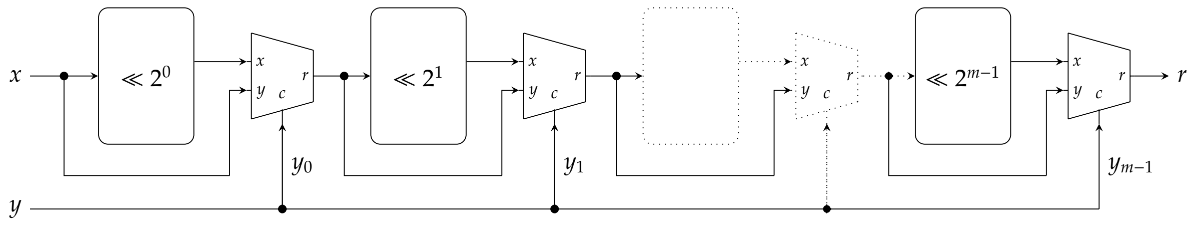


Figure 8: *A combinatorial design for n-bit (left-)shift described using a circuit diagram.*

Algorithm 3: given $y = 6$, it proceeds as follows

i	t	t'	
	x		
0	x	$x \ll 1$	$t' \leftarrow t \ll 1$
1	$x \ll 1$	$x \ll 2$	$t' \leftarrow t \ll 1$
2	$x \ll 2$	$x \ll 3$	$t' \leftarrow t \ll 1$
3	$x \ll 3$	$x \ll 4$	$t' \leftarrow t \ll 1$
4	$x \ll 4$	$x \ll 5$	$t' \leftarrow t \ll 1$
5	$x \ll 5$	$x \ll 6$	$t' \leftarrow t \ll 1$
	$x \ll 6$		

where the right-hand annotation shows each steps performed within the loop, updating t to form t'.

Figure 7 illustrates the main computational component required to implement this algorithm, which one can view as a trade-off between area and latency in which smaller area is favoured. Specifically, we only need a register to store t (on the left-hand side) and component to perform a 1-bit left-shift (in the center, which obviously contains no actual logic gates); you can view this component as realising the loop body (in line #3) of Algorithm 3. However, this implies we need to realise the loop which iterates use of this component. We can do so using an FSM of course: in each i-th step, the FSM takes t', which represents the combinatorial result $t \ll 1$, and latches it back into t ready for the $(i + 1)$-th step; implementation of such an FSM clearly demands a register to hold i and suitable control logic, both of which add somewhat to the area and design complexity. Either way, the trade-off for such simple computational steps is that the FSM will require y such steps to compute the eventual result r.

So far so good, provided we can design the FSM, but what about right-shift? Or, rotate?! The key thing to realise is that if we change line #3 of Algorithm 3, we essentially change the component at the center of our circuit. For example, if we replace $t \leftarrow t \ll 1$ with $t \leftarrow t \gg 1$ then we get a design that performs right- rather than left-shift. If we now replace $t \leftarrow t \ll 1$ with something more involved, namely

$$t \leftarrow \begin{cases} t \ll 1 & \text{if } op = 0 \\ t \gg 1 & \text{if } op = 1 \\ t \lll 1 & \text{if } op = 2 \\ t \ggg 1 & \text{if } op = 3 \end{cases}$$

then provided we also supply op as an extra input, the design can perform left- and right-shift and left- and right-rotate: a multiplexer, controlled by op, decides which result of (produced by the different operations) to update t with. We still iterate through y steps, meaning the end result is now a left- or right-shift, or left- or right-rotate by a distance of y.

4.3 Combinatorial designs

Following a similar argument as with the iterative approach, if $y = 6$ then

$$\begin{array}{rcccl} r & = & x \ll y & = & x \ll 6 \\ & & & = & (x \ll 2) \ll 4 \\ & & & = & (x \ll 2^1) \ll 2^2 \end{array}$$

We again decompose computation of a large shift on the LHS into a series of smaller shifts on the RHS, but this time each smaller shift uses a distance which is a power-of-two. Even so, the smaller shifts again combine to produce the same result: shifting x by a distance of six bits produces the same result as first shifting it by two bits, then shifting that by a further four bits. The choice of distances should hint at the underlying strategy: if we write y in base-2, then each bit y_i tells us whether or not to shift by an associated distance. That is, we compute the result via application of a simple rule "if $y_i = 1$ then shift the accumulated result by a distance of 2^i, otherwise leave it as it is" which is formalised by Algorithm 4.

Imagine we set $n = 8$, so following the above $0 \leq y < 8$ and hence y has $m = \lceil \log_2(n) \rceil = 3$ bits. The algorithm proceeds as follows

i	t	y_i	t'	
	x			
0	x	0	x	$t' \leftarrow t$
1	x	1	$x \ll 2$	$t' \leftarrow t \ll 2^1$
2	$x \ll 2$	1	$x \ll 6$	$t' \leftarrow t \ll 2^2$
	$x \ll 6$			

eventually producing $r = x \ll 6$ as expected. Translating the algorithm into a corresponding circuit design can harness the same approach used when we constructed the ripple-carry adder. More specifically, once we know n we can unroll the loop by copy and pasting the loop body (i.e., lines #3 to #5) a total of n times, replacing i with the correct value in each i-th copy. This approach produces the straight-line alternative

```
1 $t \leftarrow x$
2 **if** $y_0 = 1$ **then** $t \leftarrow t \ll 2^0$
3 **if** $y_1 = 1$ **then** $t \leftarrow t \ll 2^1$
4 **if** $y_2 = 1$ **then** $t \leftarrow t \ll 2^2$
```

which makes it (more) clear that we are essentially performing a series of choices: in each i-th stage, depending on y_i we produce either t or $t \ll 2^i$ for use by the $(i+1)$-th stage. Given the shifts themselves are by fixed constants (which we already argued are trivial), these stages are therefore a cascade of multiplexers.

Figure 8 translates this idea into a concrete circuit design. Crucially, the trade-off between latency and area is now swapped versus the previous one: the component is combinatorial, so takes a single step (whose latency is now dictated by the critical path, rather than clock speed for example) to perform each operation without the need for an FSM, but is likely to use significantly more area (relating to the logic gates required for each multiplexer).

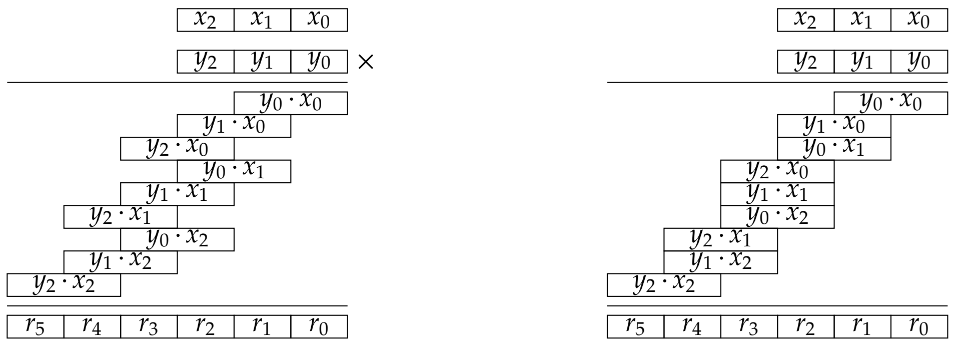
5 Components for multiplication

Formally, a multiplication operation[4] computes the **product** $r = y \cdot x$ using a **multiplicand** x and a **multiplier** y. Despite a focus on integer values of x and y here, the techniques covered sit within a more general case often described as scalar multiplication: abstractly x is *any* object from a suitable structure (wlog. an integer, meaning $x \in \mathbb{Z}$) that is multiplied, while y is an integer scalar that does the multiplying.

In the case of addition for instance, we covered several possible strategies with associated trade-offs. This is further exacerbated with multiplication, where there is a *much* larger design space. Even so, the same approach[5] is adopted: we again start by investigating the computation above from an algorithmic perspective, then translate this into a design (or set thereof) for a circuit we can construct from logic gates.

[4] Why write $y \cdot x$ rather than $x \cdot y$, which would match addition for example?! Since multiplication is commutative, we can legitimately use the operands either way around: it makes no difference to the result. Given the choice, we opt for $y \cdot x$ basically because it matches the notation $[y]\,x$ often used for more general scalar multiplication.

[5] Note that we ignore various optimisations for **squaring** operations, i.e., a multiplication $r = y \cdot x$ where we know $x = y$ so in fact $r = x^2$. For a brief explanation, see for example [**?**, Chapter 12.5].



(a) *Using operand scanning.* **(b)** *Using product scanning.*

Figure 9: *Two examples demonstrating different strategies for accumulation of base-b partial products resulting from two 3-digit operands.*

5.1 Introductory theory

5.1.1 Options for long-hand multiplication

Performing a long-hand multiplication might seem more complicated than addition. Using base-10 to start with however, we can illustrate the process in a similar way:

$$
\begin{array}{lcrcccccccl}
x & = & 623_{(10)} & \mapsto & & & & 6 & 2 & 3 & \\
y & = & 567_{(10)} & \mapsto & & & & 5 & 6 & 7 & \times \\
\hline
p_0 = 7\cdot 3\cdot 10^0 & = & 21_{(10)} & \mapsto & & & & & 2 & 1 & \\
p_1 = 7\cdot 2\cdot 10^1 & = & 140_{(10)} & \mapsto & & & & 1 & 4 & & \\
p_2 = 7\cdot 6\cdot 10^2 & = & 4200_{(10)} & \mapsto & & & 4 & 2 & & & \\
p_3 = 6\cdot 3\cdot 10^1 & = & 180_{(10)} & \mapsto & & & & 1 & 8 & & \\
p_4 = 6\cdot 2\cdot 10^2 & = & 1200_{(10)} & \mapsto & & & 1 & 2 & & & \\
p_5 = 6\cdot 6\cdot 10^3 & = & 36000_{(10)} & \mapsto & & 3 & 6 & & & & \\
p_6 = 5\cdot 3\cdot 10^2 & = & 1500_{(10)} & \mapsto & & & 1 & 5 & & & \\
p_7 = 5\cdot 2\cdot 10^3 & = & 10000_{(10)} & \mapsto & & 1 & 0 & & & & \\
p_8 = 5\cdot 6\cdot 10^4 & = & 300000_{(10)} & \mapsto & 3 & 0 & & & & & \\
\hline
r & = & 353241_{(10)} & \mapsto & 3 & 5 & 3 & 2 & 4 & 1 &
\end{array}
$$

The idea is that to compute the product $r = y \cdot x$ (at the bottom) from x and y (at the top), we form and then sum a set of **partial products** (in the middle). Each partial product p_i is produced as a result of multiplying a digit from y with a digit from x, which we term a digit-multiplication. Within this context, and multiplication in general, we use the following definition:

Definition 0.1 *The result of a digit-multiplication between x_j and y_i is said to be* **reweighted** *by the combined weight of the digits being multiplier. So given x_j has weight j and y_i has weight i, the result will have weight $j + i$.*

Here for example, notice that

- y_0 and x_0 both have weight 0, so we get $p_0 = y_0 \cdot x_0 \cdot 10^{0+0} = 7 \cdot 3 \cdot 10^0 = 21_{(10)}$ meaning p_0 has weight $0 + 0 = 0$,
- y_1 and x_1 both have weight 1, so we get $p_4 = y_1 \cdot x_1 \cdot 10^{1+1} = 6 \cdot 2 \cdot 10^2 = 1200_{(10)}$ meaning p_4 has weight $1 + 1 = 2$, and
- y_2 and x_1 have weight 2 and 1 respectively, so we get $p_7 = y_2 \cdot x_1 \cdot 10^{2+1} = 5 \cdot 2 \cdot 10^3 = 10000_{(10)}$ meaning p_7 has weight $2 + 1 = 3$.

The question then is how to generate and accumulate the partial products. It turns out there are (at least) two strategies for doing so. These are described in Figure 9, which shows that the difference lies in how the digit-multiplications are managed: they both perform the same number, just in a different order. More specifically:

```
Input: Two unsigned, $n$-digit, base-$b$ integers $x$ and $y$
Output: An unsigned, $2n$-digit, base-$b$ integer $r = y \cdot x$
$r \leftarrow 0$
for $j = 0$ upto $n - 1$ step $+1$ do
    $c \leftarrow 0$
    for $i = 0$ upto $n - 1$ step $+1$ do
        $u \cdot b + v = t \leftarrow y_j \cdot x_i + r_{j+i} + c$
        $r_{j+i} \leftarrow v$
        $c \leftarrow u$
    end
    $r_{j+n} \leftarrow c$
end
return $r$
```

Algorithm 5: An algorithm for multiplication of base-b integers using on operand scanning.

```
Input: Two unsigned, $n$-digit, base-$b$ integers $x$ and $y$
Output: An unsigned, $2n$-digit, base-$b$ integer $r = y \cdot x$
$r \leftarrow 0, c_0 \leftarrow 0, c_1 \leftarrow 0, c_2 \leftarrow 0$
for $k = 0$ upto $n + n - 1$ step $+1$ do
    for $j = 0$ upto $n - 1$ step $+1$ do
        for $i = 0$ upto $n - 1$ step $+1$ do
            if $(j + i) = k$ then
                $u \cdot b + v = t \leftarrow y_j \cdot x_i$
                $c \cdot b + c_0 = t \leftarrow c_0 + v$
                $c \cdot b + c_1 = t \leftarrow c_1 + u + c$
                $c_2 \leftarrow c_2 + c$
            end
        end
    end
    $r_k \leftarrow c_0, c_0 \leftarrow c_1, c_1 \leftarrow c_2, c_2 \leftarrow 0$
end
$r_{n+n-1} \leftarrow c_0$
return $r$
```

Algorithm 6: An algorithm for multiplication of base-b integers using on product scanning.

- The left-hand strategy is termed **operand scanning**: the idea is to loop through digits of x and y, accumulating the associated digit-multiplications into whatever the relevant digit of the result r is.

  For instance, the primitive multiplication $y_0 \cdot x_0$ that stems from the 0-th digits of x and y influences both the 0-th and 1-st digits of r; within the first step of the illustrated operand scanning multiplication therefore, this value is accumulated into r_0 and r_1.

- The right-hand strategy is termed **product scanning**: the idea is to loop through digits of the result r, so that when computing the i-th such digit r_i we accumulate all relevant digit-multiplications stemming from x and y.

  For instance, the 1-st digit of the result r is influenced by three primitive multiplications, namely $y_0 \cdot x_0$, $y_1 \cdot x_0$ and $y_0 \cdot x_1$; within the second step of the illustrated product scanning multiplication therefore, r_1 is computed as the sum of these values.

These two strategies are captured algorithmically in Algorithm 5 and Algorithm 6 respectively; as with the case of addition, we can show a trace of either algorithm invocation to demonstrate they will compute the correct result. For example, imagine we set $b = 10$, $n = 3$, $x = 623_{(10)} = \langle 3, 2, 6 \rangle_{(10)}$, and $y = 567_{(10)} = \langle 7, 6, 5 \rangle_{(10)}$ to match the case above. The operating scanning multiplication proceeds as follows

j	i	r	c	y_i	x_j	$t = y_i \cdot x_i + r_{i+j} + c$	r'	c'
		$\langle 0,0,0,0,0,0 \rangle$						
0	0	$\langle 0,0,0,0,0,0 \rangle$	0	7	3	21	$\langle 1,0,0,0,0,0 \rangle$	2
0	1	$\langle 1,0,0,0,0,0 \rangle$	2	7	2	16	$\langle 1,6,0,0,0,0 \rangle$	1
0	2	$\langle 1,6,0,0,0,0 \rangle$	1	7	6	43	$\langle 1,6,3,0,0,0 \rangle$	4
0		$\langle 1,6,3,0,0,0 \rangle$	4				$\langle 1,6,3,4,0,0 \rangle$	
1	0	$\langle 1,6,3,4,0,0 \rangle$	0	6	3	24	$\langle 1,4,3,4,0,0 \rangle$	2
1	1	$\langle 1,4,3,4,0,0 \rangle$	2	6	2	17	$\langle 1,4,7,4,0,0 \rangle$	1
1	2	$\langle 1,4,7,4,0,0 \rangle$	1	6	6	41	$\langle 1,4,7,1,0,0 \rangle$	4
1		$\langle 1,4,7,1,0,0 \rangle$	4				$\langle 1,4,7,1,4,0 \rangle$	
2	0	$\langle 1,4,7,1,4,0 \rangle$	0	5	3	22	$\langle 1,4,2,1,4,0 \rangle$	2
2	1	$\langle 1,4,2,1,4,0 \rangle$	2	5	2	13	$\langle 1,4,2,3,4,0 \rangle$	1
2	2	$\langle 1,4,2,3,5,0 \rangle$	1	5	6	35	$\langle 1,4,2,3,5,0 \rangle$	3
2		$\langle 1,4,2,3,5,0 \rangle$	3				$\langle 1,4,2,3,5,3 \rangle$	3
		$\langle 1,4,2,3,5,3 \rangle$						

yielding the expected result $r = \langle 1, 4, 2, 3, 5, 3 \rangle_{(10)} = 353241_{(10)}$, whereas the product scanning multiplication computes the same result in an obviously different way:

k	j	i	r	c_2	c_1	c_0	y_i	x_j	$t = y_i \cdot x_i$	r'	c'_2	c'_1	c'_0
			$\langle 0,0,0,0,0,0 \rangle$	0	0	0							
0	0	0	$\langle 0,0,0,0,0,0 \rangle$	0	0	0	7	3	21	$\langle 0,0,0,0,0,0 \rangle$	0	2	1
0			$\langle 0,0,0,0,0,0 \rangle$	0	2	1				$\langle 1,0,0,0,0,0 \rangle$	0	0	2
1	0	1	$\langle 1,0,0,0,0,0 \rangle$	0	0	2	7	2	14	$\langle 1,0,0,0,0,0 \rangle$	0	1	6
1	1	0	$\langle 1,0,0,0,0,0 \rangle$	0	1	6	6	3	18	$\langle 1,0,0,0,0,0 \rangle$	0	3	4
1			$\langle 1,0,0,0,0,0 \rangle$	0	3	4				$\langle 1,4,0,0,0,0 \rangle$	0	0	3
2	0	2	$\langle 1,4,0,0,0,0 \rangle$	0	0	3	7	6	42	$\langle 1,4,0,0,0,0 \rangle$	0	4	5
2	1	1	$\langle 1,4,0,0,0,0 \rangle$	0	4	5	6	2	12	$\langle 1,4,0,0,0,0 \rangle$	0	5	7
2	2	0	$\langle 1,4,0,0,0,0 \rangle$	0	4	7	5	3	15	$\langle 1,4,0,0,0,0 \rangle$	0	7	2
2			$\langle 1,4,0,0,0,0 \rangle$	0	7	2				$\langle 1,4,2,0,0,0 \rangle$	0	0	7
3	1	2	$\langle 1,4,2,0,0,0 \rangle$	0	0	7	6	6	36	$\langle 1,4,2,0,0,0 \rangle$	0	4	3
3	2	1	$\langle 1,4,2,0,0,0 \rangle$	0	4	3	5	2	10	$\langle 1,4,2,0,0,0 \rangle$	0	5	3
3			$\langle 1,4,2,0,0,0 \rangle$	0	5	3				$\langle 1,4,2,3,0,0 \rangle$	0	0	5
4	2	2	$\langle 1,4,2,3,0,0 \rangle$	0	0	5	5	6	30	$\langle 1,4,2,3,0,0 \rangle$	0	3	5
4			$\langle 1,4,2,3,0,0 \rangle$	0	3	5				$\langle 1,4,2,3,5,0 \rangle$	0	0	3
			$\langle 1,4,2,3,5,0 \rangle$	0	0	3				$\langle 1,4,2,3,5,3 \rangle$	0	0	3
			$\langle 1,4,2,3,5,3 \rangle$										

As with addition, the fact we have focused on algorithms for base-b means they naturally work with binary, base-2 values of x and y; we omit examples here for brevity.

5.1.2 Multiplication as repeated addition

With addition, our long-hand approach naturally led to a circuit design (namely the ripple-carry adder). However, with multiplication there is a less obvious route from one to the other. In fact it can make sense to rethink what multiplication actually *means*, and use *this* as the starting point instead.

At a more fundamental level than the long-hand approaches above, it is important to see that multiplication is really just repeated addition. Put another way,

$$r = y \cdot x = \underbrace{x + x + \cdots + x + x}_{y \text{ terms}},$$

so if we select $y = 14_{(10)}$, then we obviously have

$$r = 14 \cdot x = x + x + x + x + x + x + x + x + x + x + x + x + x + x.$$

This is important because we already covered how to compute an addition, plus how to design associated circuits. So to compute a multiplication, we essentially just need to reuse our addition circuit in the right way. Directly using repeated addition as above is unattractive however, since the number of operations relates to the magnitude of y. Clearly we need to perform $y - 1$ operations in total (you can see this by counting the + operators above), so as y grows the number of operations also grows: for an n-bit y we perform $O(2^n)$ operations, which grows quickly even for modest values of n. More efficient approaches are therefore vital.

Fortunately, such improvements are easy to identify. Another way to look at the multiplication of x by y is as inclusion of another weight to the digits that describe y. Given we can write y in binary, we can write

$$\begin{array}{rcl} r = y \cdot x & = & x \cdot \sum_{i=0}^{n-1} y_i \cdot 2^i \\ & = & \sum_{i=0}^{n-1} y_i \cdot x \cdot 2^i \end{array}$$

for instance (although doing so might look odd). Based on this, if we set $n = 4$ and use an example such as $y = 14_{(10)} = 1110_{(2)}$, then

$$\begin{array}{rcccccccc} y \cdot x & = & y_0 \cdot x \cdot 2^0 & + & y_1 \cdot x \cdot 2^1 & + & y_2 \cdot x \cdot 2^2 & + & y_3 \cdot x \cdot 2^3 \\ & = & 0 \cdot x \cdot 2^0 & + & 1 \cdot x \cdot 2^1 & + & 1 \cdot x \cdot 2^2 & + & 1 \cdot x \cdot 2^3 \\ & = & 0 \cdot x & + & 2 \cdot x & + & 4 \cdot x & + & 8 \cdot x \\ & = & 14 \cdot x & & & & & & \end{array}$$

Intuitively, this should already seem more attractive sense there are only n terms (relating to the n digits in y): ignoring computation of the terms themselves we only need $n - 1$, or $O(n)$, operations to compute their sum.

Of course this only gives a rough estimate because of the terms themselves: how can we compute these efficiently, so as to minimise their impact? One option is to bracket the expression per **Horner's rule [?]**, named after British mathematician William Horner who used it to efficiently evaluate polynomials. The underlying idea is to allow accumulation rather than independent computation of the terms: if we let $y = 14_{(10)} = 1110_{(2)}$ again, the expression is rewritten as

$$\begin{array}{rcl} y \cdot x & = & y_0 \cdot x + 2 \cdot (y_1 \cdot x + 2 \cdot (y_2 \cdot x + 2 \cdot (y_3 \cdot x + 2 \cdot (0)))) \\ & = & 0 \cdot x + 2 \cdot (1 \cdot x + 2 \cdot (1 \cdot x + 2 \cdot (1 \cdot x + 2 \cdot (0)))) \\ & = & 0 \cdot x + 2 \cdot (1 \cdot x + 2 \cdot (1 \cdot x + 2 \cdot (1 \cdot x + 0))) \\ & = & 0 \cdot x + 2 \cdot (1 \cdot x + 2 \cdot (1 \cdot x + 2 \cdot (1 \cdot x))) \\ & = & 0 \cdot x + 2 \cdot (1 \cdot x + 2 \cdot (1 \cdot x + 2 \cdot x)) \\ & = & 0 \cdot x + 2 \cdot (1 \cdot x + 2 \cdot (3 \cdot x)) \\ & = & 0 \cdot x + 2 \cdot (1 \cdot x + 6 \cdot x) \\ & = & 0 \cdot x + 2 \cdot (7 \cdot x) \\ & = & 0 \cdot x + 14 \cdot x \\ & = & 14 \cdot x \end{array}$$

There are two sane approaches to evaluation of the bracketed expression: either we

1. work inside-out, starting with the inner-most sub-expression and processing y from most- to least-significant bit (i.e., from y_{n-1} to y_0), meaning they are read left-to-right, or
2. work outside-in, starting with the outer-most sub-expression and processing y from least- to most-significant bit (i.e., from y_0 to y_{n-1}), meaning they are read right-to-left.

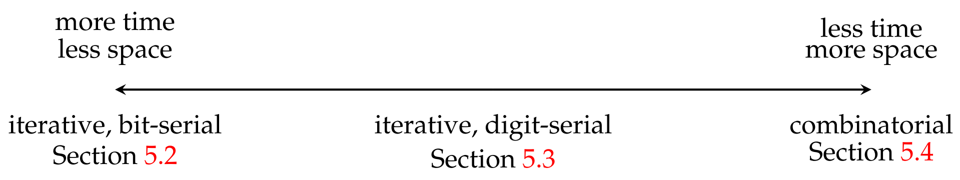
Either way, notice that each successive multiplication by 2 eventually accumulates to produce each 2^i. Using $y_3 \cdot x$ as an example, we see it multiplied by 2 a total of three times: this means we end up with

$$2 \cdot (2 \cdot (2 \cdot (y_3 \cdot x))) = y_3 \cdot x \cdot 2^3,$$

and hence the original term required, rather than needing to compute 2^3 independently from 2^2 for example.

5.1.3 A high-level overview of the design space

These reformulations, namely the summation of separate weighted terms and the accumulation of terms via Horner's rule (in inside-out/left-to-right, or outside-in/right-to-left formats) are important: they form the basis for concrete algorithms and associated iterative and combinatorial circuit designs. Within the large design space of *all* options, we overview a selection that can be summarised as follows:



You can think of these as roughly similar to Section 4, where we overviewed several options for shift operations. For instance iterative multiplication strategies deal with one (or at least few) partial products in each step; as a result it needs several steps and hence more time to compute the result, but less space due to less computation per-step. A combinatorial strategy takes exactly the opposite trade-off. As a result of being combinatorial, it requires a single step to compute the result: the time taken is dictated by the critical path. This, and the associated space required, is typically large(r) due to the amount of computation within that step. *Unlike* the case of shift operations however, a perfect separation between iterative and combinatorial designs is hard; various trade-offs that blur boundaries between such approaches are attractive, and explored where relevant.

5.1.4 Decomposition (or divide-and-conquer) techniques

Consider two circuits, both of which compute the product $r = y \cdot x$. The first circuit can deal with n-bit values of y and x, while the second can only deal with smaller, n'-bit values so we know $n' < n$. It should be reasonable that irrespective of their design, the first circuit will be more complex than the second: typically, it will have a larger latency and/or use more area than the first.

Within this context, consider the specific case of $n' = \frac{n}{2}$ in which we assume n is even; for a power-of-two n this is clearly no problem. We can split some n-bit value of x into two parts, i.e., write

$$x = x_1 \cdot 2^{n/2} + x_0$$

where x_1 and x_0 are $\frac{n}{2}$-bit integers, and likewise for y. The reason for doing so is that given the goal of computing $r = y \cdot x$, we can write

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

where

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

Simply working through the multiplication as follows

$$\begin{array}{rcccl} r & = & y \cdot x & = & (y_1 \cdot 2^{n/2} + y_0) \cdot (x_1 \cdot 2^{n/2} + x_0) \\ & & & = & (y_1 \cdot 2^{n/2} \cdot x_1 \cdot 2^{n/2}) + (y_1 \cdot 2^{n/2} \cdot x_0) + (y_0 \cdot x_1 \cdot 2^{n/2}) + (y_0 \cdot x_0) \\ & & & = & (y_1 \cdot x_1 \cdot 2^n) + (y_1 \cdot x_0 \cdot 2^{n/2}) + (y_0 \cdot x_1 \cdot 2^{n/2}) + (y_0 \cdot x_0) \\ & & & = & (y_1 \cdot x_1) \cdot 2^n + (y_1 \cdot x_0 + y_0 \cdot x_1) \cdot 2^{n/2} + (y_0 \cdot x_0) \\ & & & = & r_2 \cdot 2^n + r_1 \cdot 2^{n/2} + r_0 \end{array}$$

demonstrates why the result is correct. Beyond this, the underlying idea is that we decompose the original, larger n-bit multiplication into several smaller $\frac{n}{2}$-bit multiplications: in this specific case we compute the larger n-bit product r via four $\frac{n}{2}$-bit multiplications (plus some auxiliary additions). This mirrors the more general strategy of divide-and-conquer in algorithm design; algorithms for sorting such as merge-sort and quick-sort, for instance, decompose the task of sorting a larger sequence into that of sorting numerous smaller sequences.

The **Karatsuba-Ofman** [?] formulation improves matters further by first computing

$$\begin{array}{rcl} t_2 & = & y_1 \cdot x_1 \\ t_1 & = & (y_0 + y_1) \cdot (x_0 + x_1) \\ t_0 & = & y_0 \cdot x_0 \end{array}$$

then rewrites the terms of r as

$$\begin{array}{rcl} r_2 & = & t_2 \\ r_1 & = & t_1 - t_0 - t_2 \\ r_0 & = & t_0 \end{array}$$

This requires only three $\frac{n}{2}$-bit multiplications, although this is counterbalanced somewhat by a need for more auxiliary additions (or subtractions). Other extensions and generalisations exist: the strategy can be applied recursively for instance (i.e., decomposing each of the $\frac{n}{2}$-bit multiplications in the same way), and also consider other forms of split (e.g., a 3-way split of y and x rather than 2-way as above).

Either way, a trade-off becomes possible in terms of designing a circuit to compute $r = y \cdot x$. Basically we can either perform fewer, larger n-bit multiplications, or more, smaller $\frac{n}{2}$-bit multiplications. Revisiting the premise that a circuit for $\frac{n}{2}$-bit values will be less complex than an equivalent for n-bit values, we might therefore adopt one of (at least) two approaches:

1. instantiate and operate several smaller multiplier circuits in parallel (e.g., compute $y_1 \cdot x_1$ at the same time as $y_0 \cdot x_0$) in an attempt to reduce the overall latency,
2. instantiate and reuse one smaller multiplier (e.g., first compute $y_1 \cdot x_1$ then $y_0 \cdot x_0$) in an attempt to reduce the overall area.

Although this can be useful in the sense it widens the design space of options for our circuit to compute $r = y \cdot x$, making a decision as to whether the original monolithic approach or the decomposed approach is better wrt. some metric can be quite subtle (and depend delicately on the concrete value of n).

5.1.5 Multiplier recoding techniques

The use of multiplier **recoding** provides a broad set of strategies for improving iterative *and* combinatorial designs; we focus on examples of the former, but it is important to keep in mind that the general principles can be applied to both. The underlying idea is to

1. spend some effort *before* multiplication to **recode** (or transform) y into some equivalent y', then
2. be more efficient *during* multiplication by using y' as the multiplier rather than y.

This is a rough description however, because simple (enough) recoding may be possible during multiplication rather than strictly beforehand.

Basically, recoding just means we use a different representation for y which represents the same value, but does so in a way that allows some sort of advantage during multiplication. Think about $y = 30_{(10)}$ for instance, here written in base-10: what *other* ways could we use to represent the same value? Among many options, here are some examples:

$$\begin{array}{rcl} y & = & 30_{(10)} \\ & \mapsto & \langle 0, 1, 1, 1, 1, 0, 0, 0 \rangle_{(2)} \\ & \mapsto & \langle 2, 3, 1, 0 \rangle_{(4)} \\ & \mapsto & \langle 0, -1, 0, 0, 0, +1, 0, 0 \rangle_{(2)} \\ & \mapsto & \langle -2, 0, 2, 0 \rangle_{(4)} \end{array}$$

The first case should be obvious: this is just y represented in base-2. After this, the examples less obviously represent the same value, so it is important to see *why* they are equivalent. The second case requires a larger digit set, since each $y_i \in \{0, 1, 2, 3\}$ as a result of using base-4. By doing so, we still have

$$\begin{array}{rcl} \langle 2, 3, 1, 0 \rangle_{(4)} & \mapsto & 2 \cdot 4^0 + 3 \cdot 4^1 + 1 \cdot 4^2 + 0 \cdot 4^3 \\ & = & 2 + 12 + 16 \\ & = & 30 \end{array}$$

Input: Two unsigned, n-bit, base-2 integers x and y
Output: An unsigned, $2n$-bit, base-2 integer $r = y \cdot x$

```
t ← 0
for i = n − 1 downto 0 step −1 do
    t ← 2 · t
    if y_i = 1 then
        t ← t + x
    end
end
return t
```

Algorithm 7: An algorithm for multiplication of base-2 integers using a iterative, left-to-right, bit-serial strategy.

Input: Two unsigned, n-bit, base-2 integers x and y
Output: An unsigned, $2n$-bit, base-2 integer $r = y \cdot x$

```
t ← 0
for i = 0 upto n − 1 step +1 do
    if y_i = 1 then
        t ← t + x
    end
    x ← 2 · x
end
return t
```

Algorithm 8: An algorithm for multiplication of base-2 integers using a iterative, right-to-left, bit-serial strategy.

The third and fourth cases use signed digit sets; for example, in the third case each $y_i \in \{-1, 0, +1\}$. Again, we still have

$$\begin{array}{rcl} \langle 0, -1, 0, 0, 0, +1, 0, 0 \rangle_{(2)} & \mapsto & 0 \cdot 2^0 - 1 \cdot 2^1 + 0 \cdot 2^2 + 0 \cdot 2^3 + 0 \cdot 2^4 + 1 \cdot 2^5 + 0 \cdot 2^6 + 0 \cdot 2^7 \\ & = & -2 + 32 \\ & = & 30 \end{array}$$

It might not be immediately clear *why* these representations could give us an advantage. Intuitively however, notice two things:

1. the first example requires eight base-2 digits to represent a given y, but the second example can do the same with only four, and

2. the first example requires four non-zero base-2 digits to represent this y, but the third example can do the same with only two.

In short, features such as these, when generalised, allow more efficient strategies (in time and/or space) for multiplication using y' than y.

Whatever representation we use however, it is crucial any overhead related to producing and using the recoded y' is always less than the associated improvement during multiplication. Put another way, if the improvement is small and the overhead is large, then overall we are worse off: we may as well not using recoding at all! This demands careful analysis, for example to judge the relative merits of a specific recoding strategy given a specific n.

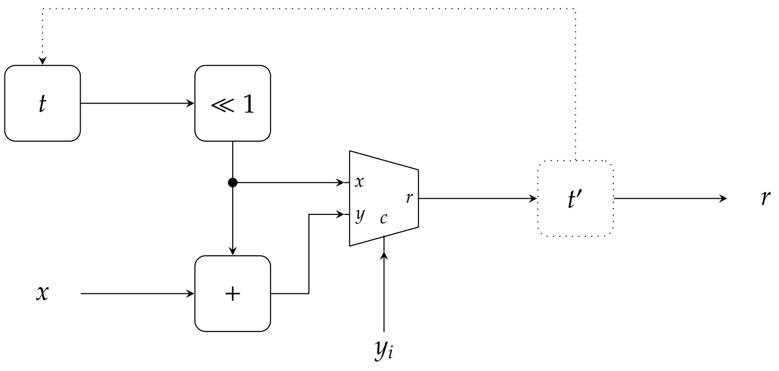


Figure 10: *An iterative, bit-serial design for $(n \times n)$-bit multiplication described using a circuit diagram.*

5.2 Iterative, bit-serial designs

Recall the expansion of $r = y \cdot x$ using Horner's rule, as reproduced below

$$\begin{array}{lcl}
y \cdot x & = & y_0 \cdot x + 2 \cdot (y_1 \cdot x + 2 \cdot (y_2 \cdot x + 2 \cdot (y_3 \cdot x + 2 \cdot (0)))) \\
 & = & 0 \cdot x + 2 \cdot (1 \cdot x + 2 \cdot (1 \cdot x + 2 \cdot (1 \cdot x + 2 \cdot (0)))) \\
 & = & 0 \cdot x + 2 \cdot (1 \cdot x + 2 \cdot (1 \cdot x + 2 \cdot (1 \cdot x + 0))) \\
 & = & 0 \cdot x + 2 \cdot (1 \cdot x + 2 \cdot (1 \cdot x + 2 \cdot (1 \cdot x))) \\
 & = & 0 \cdot x + 2 \cdot (1 \cdot x + 2 \cdot (1 \cdot x + 2 \cdot x)) \\
 & = & 0 \cdot x + 2 \cdot (1 \cdot x + 2 \cdot (3 \cdot x)) \\
 & = & 0 \cdot x + 2 \cdot (1 \cdot x + 6 \cdot x) \\
 & = & 0 \cdot x + 2 \cdot (7 \cdot x) \\
 & = & 0 \cdot x + 14 \cdot x \\
 & = & 14 \cdot x
\end{array}$$

and imagine we adopt the inside-out approach to evaluation, starting with the inner-most sub-expression (in this case the constant 0) and working outward. This process basically means repeatedly performing a general step of the form

$$y_i \cdot x + 2 \cdot t$$

which you can match against specific values of i and sub-expressions t. Doing so immediately hints at a chicken-and-egg style problem however: our goal is to compute a multiplication, but each step of the process needs two multiplications! Two facts save us by simplifying matters considerably:

1. Multiplying x by y_i can be achieved without a multiplication: given $y_i \in \{0, 1\}$, if $y_i = 0$ then we have $y_i \cdot x = 0$ and if $y_i = 1$ then $y_i \cdot x = x$. Put simply, we make a choice between 0 and x *using* y_i rather than multiply x *by* y_i.

2. Multiplying t by 2 can be achieved without a multiplication: clearly $2 \cdot t = t + t = t \ll 1$, so we just use a shift instead.

If we apply these facts then our problem is resolved: all of the seemingly impossible multiplications *within* an evaluation disappear. Putting everything together, imagine we maintain an accumulator t which holds the current (or partial) result during evaluation. Using t, the process of evaluation can be described as follows:

- start with the inner sub-expression, initially setting $t = 0$, then

- to realise each step of evaluation, apply a simple 2-part rule: first "double the accumulated result" (i.e., set t to $2 \cdot t$), then "if $y_i = 1$ then add x to the accumulated result, otherwise leave it as it is" (i.e., iff. $y_i = 1$, set t to $t + x$).

This is formalised by Algorithm 7: notice the assignment in line #1 realises the first point above, and the loop iterates over lines #3 to #6 that collectively realise the second point. If we let $n = 4$ and $y = 14_{(10)} = 1110_{(2)}$, the algorithm computes a result as follows

i	t	y_i	t'	
	0			
3	0	1	x	$t' \leftarrow 2 \cdot t + x$
2	x	1	$3 \cdot x$	$t' \leftarrow 2 \cdot t + x$
1	$3 \cdot x$	1	$7 \cdot x$	$t' \leftarrow 2 \cdot t + x$
0	$7 \cdot x$	0	$14 \cdot x$	$t' \leftarrow 2 \cdot t$
	$14 \cdot x$			

Although we will continue to focus on this left-to-right algorithm, it is interesting to note as an aside that Algorithm 8 yields the same result

i	t	x	y_i	t'	x'	
	0	x				
0	0	x	0	0	$2 \cdot x$	$x' \leftarrow 2 \cdot x$
1	0	$2 \cdot x$	1	$2 \cdot x$	$4 \cdot x$	$t' \leftarrow t + x, x' \leftarrow 2 \cdot x$
2	$2 \cdot x$	$4 \cdot x$	1	$6 \cdot x$	$8 \cdot x$	$t' \leftarrow t + x, x' \leftarrow 2 \cdot x$
3	$6 \cdot x$	$8 \cdot x$	1	$14 \cdot x$	$16 \cdot x$	$t' \leftarrow t + x, x' \leftarrow 2 \cdot x$
	$14 \cdot x$					

but operating right-to-left instead.

Note that whereas the left-to-right algorithm only needs to update t, the right-to-left algorithm updates *t and x*; this might be deeded an advantage for the former, since we only need one register rather than two. Beyond this however, how does either strategy compare to the approach based on repeated addition, which took $O(2^n)$ operations in the worst case? If n is fixed to suit the size of x and y, the number of operations performed will be dictated by the number of loop iterations: in each iteration of Algorithm 7 for instance, we perform one shift to compute $t \leftarrow 2 \cdot t$ and then (conditionally) one addition to compute $t \leftarrow t + x$ (in half the iterations on average, assuming a random y). In other words, we perform $O(n)$ operations which is now dictated by the *size* (say $n = 8$ or $n = 32$) not the *magnitude* of y (say $2^n = 2^8 = 256$ or $2^n = 2^{32} = 4294967296$) as before.

This strategy is termed **bit-serial** multiplication, because we process y guided by the 1-bit value y_i within each iteration. In a similar way to the iterative design for left-shift in Figure 7 and described previously, we can translate the strategy into a circuit design shown in Figure 10. We again only need a register to store t (on the left-hand side) and components to realise the the loop body (in lines #3 to #6) of Algorithm 7. This time the components are more complicated: the idea is that the 1-bit left-shift computes $t \ll 1 = t + t = 2 \cdot t$, then the multiplexer selects between $2 \cdot t$ and $2 \cdot t + x$ (the latter of which is computed by an adder) depending on y_i. We again need an FSM, which performs a similar task: in each i-th step, the FSM takes t', which represents the combinatorial result $2 \cdot t$ or $2 \cdot t + x$ (or more specifically $y_i \cdot x + 2 \cdot t$ as explained above), and latches it back into t ready for the $(i + 1)$-th step. The trade-off is also similar, in the sense that although we only need an adder and multiplexer (plus a register for t and the FSM), computation of the result r will take n steps. That should seem like an attractive deal however, given values of $n = 8$ or $n = 32$ the design simultaneously uses little area, produces a result in a (relatively) small *and* fixed number of steps.

To support instructions such as `muls`, two different multiplier configurations are possible for the ARM Cortex-M0 [**?**]: it can be manufactured either with a combinatorial, single cycle (32×32)-bit multiplier *or* a 32-cycle bit-serial multiplier (presumably of the type outlined above). The Cortex-M0 is used as a micro-controller within embedded applications where area and power consumption are paramount. The bit-serial multiplier therefore represents an attractive choice: assuming the increased latency can be tolerated, it satisfies the goal of minimising area associated with the component.

5.3 Iterative, digit-serial designs

5.3.1 Improvements via standard digit-serial multiplication: using an unsigned digit set

Within a bit-serial multiplier, we process a 1-bit digit of y, namely y_i, in each of n steps. In fact, this represents a special case of more general **digit-serial** multiplication. Based on selection of an appropriate **digit size** d, the idea is to recode y by splitting it into d-bit digit; these are then processed in each of $\frac{n}{d}$ steps.

Imagine $d = 2$ meaning we want to process 2-bit digits of y, and some y such that $n = 4$. Based on what

```
Input: Two unsigned, $n$-bit, base-2 integers $x$ and $y$, an integer digit size $d$
Output: An unsigned, $2n$-bit, base-2 integer $r = y \cdot x$
t ← 0
for $i = n - 1$ downto 0 step $-d$ do
    $t \leftarrow 2^d \cdot t$
    if $y_{i \ldots i-d+1} \neq 0$ then
        $t \leftarrow t + y_{i \ldots i-d+1} \cdot x$
    end
end
return $t$
```

Algorithm 9: An algorithm for multiplication of base-2 integers using a iterative, left-to-right, digit-serial strategy.

```
Input: Two unsigned, $n$-bit, base-2 integers $x$ and $y$, an integer digit size $d$
Output: An unsigned, $2n$-bit, base-2 integer $r = y \cdot x$
t ← 0
for $i = 0$ upto $n - 1$ step $+d$ do
    if $y_{i+d-1 \ldots i} \neq 0$ then
        $t \leftarrow t + y_{i+d-1 \ldots i} \cdot x$
    end
    $x \leftarrow 2^d \cdot x$
end
return $t$
```

Algorithm 10: An algorithm for multiplication of base-2 integers using a iterative, right-to-left, digit-serial strategy.

we covered originally, we know that

$$\begin{array}{rcrcl} r & = & y \cdot x & = & x \cdot \sum_{i=0}^{n-1} y_i \cdot 2^i \\ & & & = & \sum_{i=0}^{n-1} y_i \cdot x \cdot 2^i \\ & & & = & y_0 \cdot x \cdot 2^0 + y_1 \cdot x \cdot 2^1 + y_2 \cdot x \cdot 2^2 + y_3 \cdot x \cdot 2^3 \\ & & & = & y_0 \cdot x + 2 \cdot (y_1 \cdot x + 2 \cdot (y_2 \cdot x + 2 \cdot (y_3 \cdot x + 2 \cdot (0)))) \end{array}$$

The basic idea is to combine y_0 and y_1, for example, into a single digit whose value is $y_0 + 2 \cdot y_1$. This just means rewriting the expression as follows:

$$\begin{array}{rcrcl} r & = & y \cdot x & = & (y_0 + 2 \cdot y_1) \cdot x \cdot 2^0 + (y_2 + 2 \cdot y_3) \cdot x \cdot 2^2 \\ & & & = & y_{1 \ldots 0} \cdot x \cdot 2^0 + y_{2 \ldots 3} \cdot x \cdot 2^2 \\ & & & = & y_{1 \ldots 0} \cdot x + 2^2 \cdot (y_{2 \ldots 3} \cdot x + 2^2 \cdot (0)) \end{array}$$

Reading $y_{1 \ldots 0}$ as “the bits of y from 1 down to 0 inclusive” we clearly have $y_{1 \ldots 0} \in \{0, 1, 2, 3\}$. The expression should otherwise seem roughly similar, and if you expand it certainly produces the same result. Using $y = 14_{(10)} = 1110_{(2)}$ for instance, we get

$$\begin{array}{rcrcl} r & = & y \cdot x & = & y_{1 \ldots 0} \cdot x + 2^2 \cdot (y_{2 \ldots 3} \cdot x + 2^2 \cdot (0)) \\ & & & = & 10_{(2)} \cdot x + 2^2 \cdot (11_{(2)} \cdot x + 2^2 \cdot (0)) \\ & & & = & 2 \cdot x + 2^2 \cdot (3 \cdot x + 2^2 \cdot (0)) \\ & & & = & 2 \cdot x + 12 \cdot x \\ & & & = & 14 \cdot x \end{array}$$

The important thing to realise is that provided d divides n, extracting each d-bit digit from y is easy even if we select a larger d or n. Taking y written in binary, recoding it to form y' means splitting the sequence of bits into d-element sub-sequences; this is simple enough that we view is as happening during multiplication, ignoring the need to formally recode y beforehand.

To implement this new strategy however, Algorithm 7 needs to be generalised for *any* d. Recall that our general step in bit-serial multiplication was

$$y_i \cdot x + 2 \cdot t.$$

Looking at the example above, a similar form

$$y_{i \ldots i-d+1} \cdot x + 2^d \cdot t$$

can be identified, which differs in the left- and right-hand terms of the addition. In the right-hand term, rather than multiply t by 2, we now multiply it by 2^d. This is a fairly simple change however, because we can compute this result by repeatedly doubling t a total of d times or even just left-shifting it by d bits (since $2^d \cdot t = t \ll d$). The left-hand term is more tricky. During bit-serial multiplication, we needed to compute $y_i \cdot x$; this was translated into a conditional statement that was able to select between 0 and x because $y_i \in \{0, 1\}$. Now, each d-bit digit

$$y_{i \ldots i-d+1} \in \{0, 1, \ldots 2^d - 1\}$$

could be any one of 2^d values. Now need a *lot* of conditions, or more probably more likely a single a $(d \times n)$-bit multiplication to compute

$$t \leftarrow t + y_{i \ldots i-d+1} \cdot x.$$

Making these changes produces Algorithm 9 as a result. Looking at a trace of the algorithm for the now familiar case of $y = 14_{(10)} = 1110_{(2)}$, i.e.,

i	t	$y_{i \ldots i-d+1}$	t'	
	0			
3	0	$11_{(2)} = 3_{(10)}$	$3 \cdot x$	$t' \leftarrow 2^2 \cdot t + 3 \cdot x$
1	$3 \cdot x$	$10_{(2)} = 2_{(10)}$	$14 \cdot x$	$t' \leftarrow 2^2 \cdot t + 2 \cdot x$
	$14 \cdot x$			

a clear advantage is evident: we now require $\frac{n}{d}$ steps to compute the result. However, there is also a clear *disadvantage* in that we currently lack a way to compute the $(d \times n)$-bit multiplication in line #5.

Section 5.4, which presents a number of combinatorial multiplier designs, offers one way to resolve this problem. The idea is to think of a digit-serial multiplier design as a hybrid of iterative and combinatorial: the design is iterative in the sense it performs $\frac{n}{d}$ steps, but now the i-th such step utilises a $(d \times n)$-bit combinatorial multiplier component. You can view this component as replacing the multiplexer shown in Figure 7, which realised the previous $(1 \times n)$-bit multiplication of y_i by x via selection between 0 and x. Since we can select d, the nature of this hybrid can also be chosen. This means a choice of trade-off between time and space: a larger d means less steps of computation but a larger combinatorial multiplier for the $(d \times n)$-bit multiplication step, and vice versa.

5.3.2 Improvements via Booth multiplication: using a signed digit set

A question: what is the most efficient way to compute $r = 15 \cdot x$, i.e., $r = y \cdot x$ for the fixed multiplier $y = 15$? Since we already know that shifting x by a fixed distance requires no computation, a reasonable first attempt would be to compute

$$\begin{aligned} r = 15 \cdot x &= 1 \cdot x + 2 \cdot x + 4 \cdot x + 8 \cdot x \\ &= 2^0 \cdot x + 2^1 \cdot x + 2^2 \cdot x + 2^3 \cdot x \\ &= x \ll 0 + x \ll 1 + x \ll 2 + x \ll 3 \end{aligned}$$

A better strategy exists however: remember that computing a subtraction is (more or less) as easy as an addition, so might instead opt for

$$\begin{aligned} r = 15 \cdot x &= 16 \cdot x - 1 \cdot x \\ &= 2^4 \cdot x - 2^0 \cdot x \\ &= x \ll 4 - x \ll 0 \end{aligned}$$

Intuitively, this latter strategy might be preferred given we only sum two terms rather than four.

Booth recoding [?] is a standard recoding-based strategy for multiplication which generalises this example, allowing the same sort of advantage but for *any* y; a central advantage versus standard digit-serial multiplication is the use of a signed representation of y, and hence use of both addition *and* subtraction operations.

Example of the strategy include a (12×12)-bit multiplier design described by Thomas and Balatoni [?] and deployed in the PDP-8 computer. Likewise, the MIPS R4000 [?] processor housed a Booth-based multiplier, using exactly the recoding strategy described above but a $(64 \cdot 64)$-bit combinatorial rather than iterative design. Mirapuri et al. [?, Page 13] explain the design, which, interestingly, still splits the multiplication itself into several steps: Booth recoding, multiplicand selection, partial product generation and product accumulation. A series of carry-save adders are used to accumulate the partial products, which produces a result r that is stored into two 64-bit registers called `hi` and `lo` (meaning the more- and less-significant 64-bit halves).

Basic, radix-2 Booth recoding The basic approach to Booth recoding is reasonably simple to explain, since it just formalises the previous informal strategy above.

The idea is to first identify sub-sequences of y between indices i and j st. $y_k = 1$ for $i \le k \le j$. Put more simply, these **runs** are just consecutive bits of y whose value is 1. An example makes this clearer: if $n = 8$ and $y = 30_{(10)} = 00011110_{(2)}$, then we can clearly identify a suitable sub-sequence for $i = 1$ to $j = 4$: for each value of $k \in \{1, 2, 3, 4\}$, i.e., $i \le k \le j$, we have $y_k = 1$. Then, once we identify a suitable sub-sequence, we replace it with a single digit whose weight is $2^{j+1} - 2^i$. In our example, we therefore recode the sub-sequence as a single digit of weight

$$2^{j+1} - 2^i = 2^{4+1} - 2^1 = 2^5 - 2^1 = 30$$

meaning the original

$$\begin{array}{rcl} y & = & 30_{(10)} \\ & \mapsto & 2^4 + 2^3 + 2^2 + 2^1 \\ & \mapsto & \langle 0, +1, +1, +1, +1, 0, 0, 0 \rangle_{(2)} \end{array}$$

is now recoded as

$$\begin{array}{rcl} y' & = & \langle 0, -1, +0, +0, +0, +1, 0, 0 \rangle_{(2)} \\ & \mapsto & -2^1 + 2^5 \\ & \mapsto & 30_{(10)} \end{array}$$

which clearly still represents the same value. Strictly speaking y' is still written in binary, albeit *signed* binary since each digit is now in the set $\{0, \pm 1\}$ rather than $\{0, 1\}$; as a result, we say y' is a base-2 or more often radix-2 recoding of y, using $y'_{(2)}$ to show this fact.

Using the same intuition as previously, the recoded y' is preferable to y since it has a lower weight (i.e., number of non-zero digits). We can see the impact this feature has by again illustrating the process: consider setting $x = 6_{(10)} = 00000110_{(2)}$ and $y = 30_{(10)} = 00011110_{(2)}$. Normally, we would write

$$\begin{array}{lcrr|cccccccccccccccc|l}
x & = & & 6_{(10)} \mapsto & & & & & & & & & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & \\
y & = & & 30_{(10)} \mapsto & & & & & & & & & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & \times \\
\hline
p_0 & = & 0 \cdot x \cdot 2^0 = & 0_{(10)} \mapsto & & & & & & & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \\
p_1 & = & +1 \cdot x \cdot 2^1 = & +12_{(10)} \mapsto & & & & & & & & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & & \\
p_2 & = & +1 \cdot x \cdot 2^2 = & +24_{(10)} \mapsto & & & & & & & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & & & \\
p_3 & = & +1 \cdot x \cdot 2^3 = & +48_{(10)} \mapsto & & & & & & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & & & & \\
p_4 & = & +1 \cdot x \cdot 2^4 = & +96_{(10)} \mapsto & & & & & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & & & & & \\
p_5 & = & 0 \cdot x \cdot 2^5 = & 0_{(10)} \mapsto & & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & & & \\
p_6 & = & 0 \cdot x \cdot 2^6 = & 0_{(10)} \mapsto & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & & & & \\
p_7 & = & 0 \cdot x \cdot 2^7 = & 0_{(10)} \mapsto & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & & & & & \\
\hline
r & = & & 180_{(10)} \mapsto & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & \\
\hline
\end{array}$$

which requires accumulation of four non-zero partial products. However, by first recoding y into y' we find

$$\begin{array}{lcrr|cccccccccccccccc|l}
x & = & & 6_{(10)} \mapsto & & & & & & & & & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & \\
y & = & & 30_{(10)} \mapsto & & & & & & & & & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & \times \\
y'_{(2)} & = & & 30_{(10)} \mapsto & & & & & & & & & 0 & 0 & +1 & 0 & 0 & 0 & -1 & 0 & \\
\hline
p_0 & = & 0 \cdot x \cdot 2^0 = & 0_{(10)} \mapsto & & & & & & & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \\
p_1 & = & -1 \cdot x \cdot 2^1 = & -12_{(10)} \mapsto & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & & \\
p_2 & = & 0 \cdot x \cdot 2^2 = & 0_{(10)} \mapsto & & & & & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & \\
p_3 & = & 0 \cdot x \cdot 2^3 = & 0_{(10)} \mapsto & & & & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & \\
p_4 & = & 0 \cdot x \cdot 2^4 = & 0_{(10)} \mapsto & & & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & & \\
p_5 & = & +1 \cdot x \cdot 2^5 = & +192_{(10)} \mapsto & & & & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & & & & & & \\
p_6 & = & 0 \cdot x \cdot 2^6 = & 0_{(10)} \mapsto & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & & & & \\
p_7 & = & 0 \cdot x \cdot 2^7 = & 0_{(10)} \mapsto & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & & & & & \\
\hline
r & = & & 180_{(10)} \mapsto & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & \\
\hline
\end{array}$$

which requires accumulation of two rather than four non-zero partial products. Put another way, and as in the initial example, the recoded y' is preferable since it has a lower weight (i.e., fewer non-zero digits).

Modified, radix-4 Booth recoding Although basic Booth recoding *seems* to produce what we want, there is a problem lurking behind the scenes: in the worst-case, y' does not yield an improvement over using y itself.

This can be demonstrated using an example: if we set $x = 6_{(10)} = 00000110_{(2)}$ and $y = 5_{(10)} = 00000101_{(2)}$ then we get

$$\begin{array}{llr|cccccccccccccccc|l}
x & = & 6_{(10)} \mapsto & & & & & & & & & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & \\
y & = & 5_{(10)} \mapsto & & & & & & & & & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & \times \\
y'_{(2)} & = & 5_{(10)} \mapsto & & & & & & & & & 0 & 0 & 0 & 0 & +1 & -1 & +1 & -1 & \\
\cline{4-19}
p_0 & = +1 \cdot x \cdot 2^0 = & -6_{(10)} \mapsto & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & \\
p_1 & = -1 \cdot x \cdot 2^1 = & +12_{(10)} \mapsto & & & & & & & & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & & \\
p_2 & = +1 \cdot x \cdot 2^2 = & -24_{(10)} \mapsto & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & & & \\
p_3 & = -1 \cdot x \cdot 2^3 = & +48_{(10)} \mapsto & & & & & & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & & & & \\
p_4 & = \ 0 \cdot x \cdot 2^4 = & 0_{(10)} \mapsto & & & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & & \\
p_5 & = \ 0 \cdot x \cdot 2^5 = & 0_{(10)} \mapsto & & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & & & \\
p_6 & = \ 0 \cdot x \cdot 2^6 = & 0_{(10)} \mapsto & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & & & & \\
p_7 & = \ 0 \cdot x \cdot 2^7 = & 0_{(10)} \mapsto & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & & & & & \\
\cline{4-19}
r & = & 30_{(10)} \mapsto & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & \\
\cline{4-19}
\end{array}$$

which (still) requires accumulation of four non-zero partial products (as would using y). The solution is to employ a modified Booth recoding in a second recoding step based on the $y'_{(2)}$ we already have:

1. reading right-to-left, group the recoded digits into pairs of the form (y'_{i+1}, y'_i), then
2. treat each pair as a single digit whose value is $y'_i + 2 \cdot y'_{i+1}$ per

$$\begin{array}{lcrlcrcr}
y'_{i+1} & = & 0 & y'_i & = & 0 & \mapsto & 0 \\
y'_{i+1} & = & 0 & y'_i & = & +1 & \mapsto & +1 \\
y'_{i+1} & = & 0 & y'_i & = & -1 & \mapsto & -1 \\
y'_{i+1} & = & +1 & y'_i & = & 0 & \mapsto & +2 \\
y'_{i+1} & = & +1 & y'_i & = & +1 & \mapsto & \text{not possible} \\
y'_{i+1} & = & +1 & y'_i & = & -1 & \mapsto & +1 \\
y'_{i+1} & = & -1 & y'_i & = & 0 & \mapsto & -2 \\
y'_{i+1} & = & -1 & y'_i & = & +1 & \mapsto & -1 \\
y'_{i+1} & = & -1 & y'_i & = & -1 & \mapsto & \text{not possible}
\end{array}$$

   meaning that in a sense y'_{i+1} has twice the weight of y'_i.

Since the original Booth recoding was a signed radix-2 recoding of y, strictly speaking we now have a signed radix-4 recoding of the same y: each digit in what we write as $y'_{(4)}$ to denote the difference, is in the set $\{0, \pm 1, \pm 2\}$. Note that the two invalid (or impossible) pairs result from the original Booth recoding: we cannot ever encounter them, because the first recoding step will have eliminated the associated run.

So again setting $x = 6_{(10)} = 00000110_{(2)}$ and $y = 5_{(10)} = 00000101_{(2)}$ then applying this second recoding step, we now get

$$\begin{array}{llr|cccccccccccccccc|l}
x & = & 6_{(10)} \mapsto & & & & & & & & & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & \\
y & = & 5_{(10)} \mapsto & & & & & & & & & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & \times \\
y'_{(2)} & = & 5_{(10)} \mapsto & & & & & & & & & 0 & 0 & +1 & 0 & +1 & -1 & +1 & -1 & \\
y'_{(4)} & = & 5_{(10)} \mapsto & & & & & & & & & & & & & & +1 & & +1 & \\
\cline{4-19}
p_0 & = +1 \cdot x \cdot 2^0 = & +6_{(10)} \mapsto & & & & & & & & & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & \\
p_2 & = +1 \cdot x \cdot 2^2 = & +24_{(10)} \mapsto & & & & & & & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & & & \\
p_4 & = \ 0 \cdot x \cdot 2^4 = & 0_{(10)} \mapsto & & & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & & \\
p_6 & = \ 0 \cdot x \cdot 2^6 = & 0_{(10)} \mapsto & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & & & & \\
\cline{4-19}
r & = & 30_{(10)} \mapsto & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & \\
\cline{4-19}
\end{array}$$

which has a more reasonable number of non-zero partial products.

An algorithm for Booth-based recoding Both the first and second recoding steps above are presented in a somewhat informal manner; the goal was simply to demonstrate the intuition. To use them in practice, we of course need a more formal algorithm. Fortunately such an algorithm is simple to construct: notice that within the radix-2 Booth recoded $y'_{(2)}$,

```
Input: An unsigned, $n$-bit, base-2 integer $y$
Output: A radix-2 Booth recoding $y'_{(2)}$ of $y$
1 $y' \leftarrow \emptyset$
2 for $i = 0$ upto $n$ step 2 do
3     if $(i-1) < 0$ then $t_0 \leftarrow 0$ else $t_0 \leftarrow y_{i-1}$
4     if $i \geq n$ then $t_1 \leftarrow 0$ else $t_1 \leftarrow y_i$
5     if $(i+1) \geq n$ then $t_2 \leftarrow 0$ else $t_2 \leftarrow y_{i+1}$
6     $y'_{i+1} \leftarrow \begin{cases} 0 & \text{if } t = 000_{(2)} \\ 0 & \text{if } t = 001_{(2)} \\ +1 & \text{if } t = 010_{(2)} \\ +1 & \text{if } t = 011_{(2)} \\ -1 & \text{if } t = 100_{(2)} \\ -1 & \text{if } t = 101_{(2)} \\ 0 & \text{if } t = 110_{(2)} \\ 0 & \text{if } t = 111_{(2)} \end{cases}$  $y'_i \leftarrow \begin{cases} 0 & \text{if } t = 000_{(2)} \\ +1 & \text{if } t = 001_{(2)} \\ -1 & \text{if } t = 010_{(2)} \\ 0 & \text{if } t = 011_{(2)} \\ 0 & \text{if } t = 100_{(2)} \\ +1 & \text{if } t = 101_{(2)} \\ -1 & \text{if } t = 110_{(2)} \\ 0 & \text{if } t = 111_{(2)} \end{cases}$
7 end
8 return $y'$
```

Algorithm 11: An algorithm for radix-2 Booth recoding.

```
Input: An unsigned, $n$-bit, base-2 integer $y$
Output: A radix-4 Booth recoding $y'_{(4)}$ of $y$
1 $y' \leftarrow \emptyset$
2 for $i = 0$ upto $n$ step 2 do
3     if $(i-1) < 0$ then $t_0 \leftarrow 0$ else $t_0 \leftarrow y_{i-1}$
4     if $i \geq n$ then $t_1 \leftarrow 0$ else $t_1 \leftarrow y_i$
5     if $(i+1) \geq n$ then $t_2 \leftarrow 0$ else $t_2 \leftarrow y_{i+1}$
6     $y'_{i/2} \leftarrow \begin{cases} 0 & \text{if } t = 000_{(2)} \\ +1 & \text{if } t = 001_{(2)} \\ +1 & \text{if } t = 010_{(2)} \\ +2 & \text{if } t = 011_{(2)} \\ -2 & \text{if } t = 100_{(2)} \\ -1 & \text{if } t = 101_{(2)} \\ -1 & \text{if } t = 110_{(2)} \\ 0 & \text{if } t = 111_{(2)} \end{cases}$
7 end
8 return $y'$
```

Algorithm 12: An algorithm for radix-4 Booth recoding.

- y'_i depends on y_{i-1} and y_i, while
- y'_{i+1} depends on y_i and y_{i+1}.

These digits are paired to form the radix-4 Booth recoded $y'_{(4)}$, so each digit in $y'_{(4)}$ depends on three bits y_{i-1}, y_i and y_{i+1} from y. Thanks to this observation, the Booth recoding of y can be produced much more easily than first appears: we can produce each digit of $y'_{(2)}$ or $y'_{(4)}$ from a 3-bit sub-sequence (or window) of bits in y. More explicitly, for even values of i, we have

Unsigned radix-2			Signed radix-2		Signed radix-4
y_{i+1}	y_i	y_{i-1}	y'_{i+1}	y'_i	$y'_{i/2}$
0	0	0	0	0	0
0	0	1	0	+1	+1
0	1	0	+1	−1	+1
0	1	1	+1	0	+2
1	0	0	−1	0	−2
1	0	1	−1	+1	−1
1	1	0	0	−1	−1
1	1	1	0	0	0

Input: An unsigned, n-bit, base-2 integer x, and a radix-4 Booth recoding $y'_{(4)}$ of some integer y
Output: An unsigned, $2n$-bit, base-2 integer $r = y \cdot x$

```
t ← 0
for i = |y'| − 1 downto 0 step −1 do
    t ← t · 2^2
    if y'_i = −2 then
        t ← t − 2 · x
    end
    else if y'_i = −1 then
        t ← t − x
    end
    else if y'_i = +1 then
        t ← t + x
    end
    else if y'_i = +2 then
        t ← t + 2 · x
    end
end
return t
```

Algorithm 13: An algorithm for multiplication of base-2 integers using an iterative, left-to-right, digit-serial strategy with radix-4 Booth recoding.

assuming suitable padding of y (i.e., $y_j = 0$ for $j < 0$ and $j \geq n$).

Algorithm 11 and Algorithm 12 capture these rules in algorithms which respectively produce radix-2 and radix-4 recodings of a given y. Crucially, one can unroll the loop to produce an associated combinatorial circuit. For Algorithm 12 say, one would replicates a single recoding cell: each instance of the cell would accepts three bits of y as input (namely y_{i+1}, y_i, and y_{i-1}) and produce a digit of the recoded $y'_{(4)}$ as output. This implies that recoding y into $y'_{(4)}$ can be performed *during* rather than *before* the subsequent multiplication; a side-effect is that the only significant overhead relates to area rather than time.

An algorithm for Booth-based multiplication Finally, we can address the problem of *using* the recoded multiplier to actually perform the multiplication above: ideally this will be more efficient than the bit-serial starting point. Algorithm 13 shows the result, which one can consider as a type of digit-serial multiplier: each iteration of the loop processes a digit of $y'_{(4)}$ formed from multiple bits in y.

1. In Algorithm 7, $|y| = n$ dictates the number of loop iterations, and Algorithm 9 further improves this to $\frac{n}{d}$ for appropriate choices of d. Algorithm 13 on the other hand requires

$$|y'_{(4)}| \simeq \frac{|y|}{2} \simeq \frac{n}{2}$$

   iterations. As with the digit-serial strategy, to make this work, we need to compute $2^2 \cdot t$ in line #3 (rather than just $2 \cdot t$ as with the bit-serial case), but this can again be realised using an addition or shift since

$$2^2 \cdot t = 4 \cdot t = t \ll 2.$$

   Versus the bit-serial strategy, the Booth-based strategy will perform half the number of iterations; this suggests the result is computed with lower latency. We *could* make the same improvement using the digit-serial strategy of course, and the ability to select d might be an advantage that favours doing so: as described at least, modified Booth recoding always takes two digits and groups them into one so the improvement is fixed unless we further generalise the algorithm.

2. In Algorithm 7 we had $y_i \in \{0, 1\}$ and in Algorithm 7 we had $y_{i \ldots i-d+1} \in \{0, 1, \ldots 2^d - 1\}$. In Algorithm 13 however, we now have $y'_i \in \{0, \pm 1, \pm 2\}$.

   Versus the bit-serial strategy, this just means we have to test each non-zero y'_i against more cases than before (i.e., just $y_i = 1$) and take appropriate action: lines #4 to #15 show such cases. For $y'_i = -1$ versus $y'_i = +1$ this is easy: we subtract x from t rather than adding x to t. Likewise, the $y'_i = -2$ and $y'_i = +2$ mean adding (resp. subtracting) $2 \cdot x$ to (resp. from) t. Notice that unlike the digit-serial strategy however, where we needed a combinatorial $(d \cdot n)$-bit multiplier to deal with the larger set

Input: Two unsigned, n-bit, base-2 integers x and y, an integer digit size d
Output: An unsigned, $2n$-bit, base-2 integer $r = y \cdot x$

```
t ← 0
for i = 0 upto n − 1 step +d do
    if y_{d−1...0} ≠ 0 then
        t ← t + y_{d−1...0} · x
    end
    x ← 2^d · x
    y ← y/2^d
    if y = 0 then
        return t
    end
end
return t
```

Algorithm 14: An algorithm for multiplication of base-2 integers using an iterative, left-to-right, digit-serial strategy with early termination.

of cases in each iteration, here we avoid the associated area overhead. Specifically, since $2 \cdot x$ can be produced via a shift of x (rather than an extra addition), it is more or less the same overhead as adding (resp. subtracting) x itself.

Consider an example: setting $n = 4$ and $y = 14_{(10)} = 1110_{(2)}$, we first use Algorithm 12 as follows

i	y_{i+1}	y_i	y_{i-1}	t_2	t_1	t_0	t	y'
								$\emptyset$
0	1	0	$\perp$	1	0	0	$100_{(2)}$	$\langle -2 \rangle$
2	1	1	1	1	1	1	$111_{(2)}$	$\langle -2, 0 \rangle$
4	$\perp$	$\perp$	1	0	0	1	$001_{(2)}$	$\langle -2, 0, +1 \rangle$
								$\langle -2, 0, +1 \rangle$

to recode y into $y'_{(4)} = \langle -2, 0, +1 \rangle$. Then we use Algorithm 13 as follows

i	y'_i	t	t'	
		0		
2	+1	0	x	$t' \leftarrow 2^2 \cdot t + 1 \cdot x$
1	0	x	$4 \cdot x$	$t' \leftarrow 2^2 \cdot t$
0	−2	$4 \cdot x$	$14 \cdot x$	$t' \leftarrow 2^2 \cdot t - 2 \cdot x$
		$14 \cdot x$		

to produce the result expected in three rather than six steps.

5.3.3 Improvements via early termination: avoiding unnecessary iterations

Look again at Algorithm 9, which describes left-to-right digit-serial multiplication, and imagine we parameterise it with $d = 2$ and $n = 8$: it accepts an 8-bit multiplier y as input, processing successive 2-bit digits in a total of $\frac{8}{2} = 4$ iterations. For example, if

$$\begin{array}{rcl} y & = & 30_{(10)} \\ & = & 00011110_{(2)} \\ & \mapsto & \langle 10_{(2)}, 11_{(2)}, 01_{(2)}, 00_{(2)} \rangle \end{array}$$

the last line illustrates each 2-bit digit extracted from y. Notice that the last digit in this sequence is zero. Put another way, in the iteration where $i = 7$ the algorithm extracts the digit

$$y_{i \ldots i-d+1} = y_{7 \ldots 6} = 0.$$

In the left-to-right algorithm this zero-digit is the first one processed, a fact which is evident if we examine a trace of the algorithm:

i	t	$y_{i\ldots i-d+1}$	t'	
	0			
7	0	$00_{(2)} = 0_{(10)}$	0	$t' \leftarrow 2^2 \cdot t$
5	$2 \cdot x$	$01_{(2)} = 1_{(10)}$	$1 \cdot x$	$t' \leftarrow 2^2 \cdot t + 1 \cdot x$
3	$1 \cdot x$	$11_{(2)} = 3_{(10)}$	$7 \cdot x$	$t' \leftarrow 2^2 \cdot t + 3 \cdot x$
1	$7 \cdot x$	$10_{(2)} = 2_{(10)}$	$30 \cdot x$	$t' \leftarrow 2^2 \cdot t + 2 \cdot x$
	$30 \cdot x$			

We could avoid performing the associated multiplication, because of course we know the result will also be zero and hence t will remain unchanged. Depending how this is realised, doing so may reduce the overall latency; this is roughly analogous to avoiding a given $t \leftarrow t + x$ step, using a conditional statement, in Algorithm 7 for bit-serial multiplication. Given the same example, the right-to-left version of the algorithm allows an even more aggressive form of optimisation:

i	t	x	$y_{i+d-1\ldots i}$	t'	x'	
	0	x				
0	0	x	$10_{(2)} = 2_{(10)}$	$2 \cdot x$	$2^2 \cdot x$	$t' \leftarrow t + 2 \cdot x,\ x' \leftarrow 2^2 \cdot x$
2	$2 \cdot x$	$2^2 \cdot x$	$11_{(2)} = 3_{(10)}$	$14 \cdot x$	$2^4 \cdot x$	$t' \leftarrow t + 3 \cdot x,\ x' \leftarrow 2^2 \cdot x$
4	$14 \cdot x$	$2^4 \cdot x$	$01_{(2)} = 1_{(10)}$	$30 \cdot x$	$2^6 \cdot x$	$t' \leftarrow t + 1 \cdot x,\ x' \leftarrow 2^2 \cdot x$
6	$30 \cdot x$	$2^6 \cdot x$	$00_{(2)} = 0_{(10)}$	$30 \cdot x$	$2^8 \cdot x$	$t' \leftarrow t + 0 \cdot x,\ x' \leftarrow 2^2 \cdot x$
	$30 \cdot x$					

In contrast to the left-to-right case, the right-to-left algorithm makes no update to t in the last iteration where $i = 6$: the reason is that the last digit processed is a zero-digit. You could take this point further still. If at some i-th iteration the digits processed by all j-th iterations for $j > i$ are zero-digits, then we may as well stop: *none* of them will update t, so we can just return it early rather than perform those extra iterations.

Using Algorithm 10 as a starting point, this strategy is implemented by Algorithm 14. Invoking the algorithm with the same inputs yields the following:

i	t	x	y	$y_{d-1\ldots 0}$	t'	x'	y'	
	0	x	$00011110_{(2)}$					
0	0	x	$00011110_{(2)}$	$10_{(2)} = 2_{(10)}$	$2 \cdot x$	$2^2 \cdot x$	$00000111_{(2)}$	$t' \leftarrow t + 2 \cdot x,\ x' \leftarrow 2^2 \cdot x,\ y' \leftarrow y/2^2$
2	$2 \cdot x$	$2^2 \cdot x$	$00000111_{(2)}$	$11_{(2)} = 3_{(10)}$	$14 \cdot x$	$2^4 \cdot x$	$00000001_{(2)}$	$t' \leftarrow t + 3 \cdot x,\ x' \leftarrow 2^2 \cdot x,\ y' \leftarrow y/2^2$
4	$14 \cdot x$	$2^4 \cdot x$	$00000001_{(2)}$	$01_{(2)} = 1_{(10)}$	$30 \cdot x$	$2^6 \cdot x$	$00000000_{(2)}$	$t' \leftarrow t + 1 \cdot x,\ x' \leftarrow 2^2 \cdot x,\ y' \leftarrow y/2^2$
	$30 \cdot x$							

Once t, x and y have been updated within the iteration for $i = 4$, we find that $y' = 0$: this triggers the conditional statement, meaning t is returned after just three (via line #9) rather than four (via line #12) iterations. This approach is termed **early termination**, and represents another trade-off:

1. In Algorithm 14, the loop body spanning lines #3 to #10 is more complex than Algorithm 10. More specifically, we now need to update t, x *and* y, plus get the FSM that controls iteration to test y and conditionally return t.

2. This added complexity, which typically means more area as the result of the additional logic, *potentially* yields a reduction in the latency of multiplication. This is a *potential* rather than definitive improvement however, because it depends on the multiplier y containing more-significant zero-digits: if this does not hold, the strategy is no better than standard digit-serial multiplication.

As an example, the ARM7TDMI [?] processor houses a $(8 \cdot 32)$-bit combinatorial multiplier, which is used within early terminating digit-serial multiplication (for the fixed case where $d = 8$). This is invoked by various instructions, such as `umull` that perform a (32×32)-bit multiplication. One must assume ARM selected this design based on careful analysis. For instance, it seems fair to claim that

- using a digit-serial multiplier makes an good trade-off between time and space (due to the hybrid, combinatorial and iterative nature), which is particularly important for embedded processors, plus

- although early termination adds some overhead, it often produces a reduction in latency because of the types of y used by programs executing on such a processor: a significant proportion will be address arithmetic with (relatively) small values of y, meaning a high(er) chance of one (or more) more-significant zero-digits.

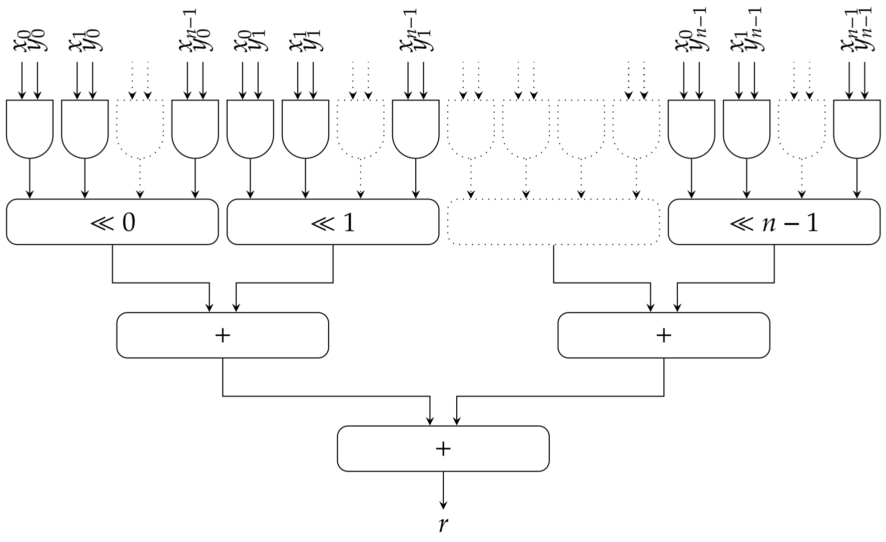


Figure 11: *A combinatorial, $(n \times n)$-bit tree multiplier described using a circuit diagram.*

5.4 Combinatorial designs

5.4.1 A tree multiplier

From iterative to combinatorial: a viable but non-ideal starting point Motivated by the combinatorial shift design we developed, a starting point for combinatorial multiplication might be to convince ourselves that such an approach is at least viable. Recall that said combinatorial shift was produced by basically unrolling the loop body within Algorithm 4 for a fixed n.

We can pull the same trick using Algorithm 7, which uses the Horner-based evaluation strategy: recall that it processes a 1-bit digit of y, left-to-right meaning from most- to least-significant bit, in each of n steps. By unrolling the algorithm given $n = 4$ for instance, we get a straight-line result

```
1 $t \leftarrow 0$
2 $t \leftarrow 2 \cdot t$, **if** $y_3 = 1$ **then** $t \leftarrow t + x$
3 $t \leftarrow 2 \cdot t$, **if** $y_2 = 1$ **then** $t \leftarrow t + x$
4 $t \leftarrow 2 \cdot t$, **if** $y_1 = 1$ **then** $t \leftarrow t + x$
5 $t \leftarrow 2 \cdot t$, **if** $y_0 = 1$ **then** $t \leftarrow t + x$
```

which translates into a combinatorial circuit: the n unrolled steps are now n stages of the circuit, the i-th of which is essentially the same as Figure 10. So on one hand, we get something that works by simply reusing what we already know. However, doing so arguably produces two drawbacks:

1. the overall critical path runs through n adder components, meaning $O(n^2)$ date delays if we use ripple-carry adders (since each adder implies $O(n)$ gate delays per Section 3.1), plus
2. the general-purpose multiplexers, required essentially to select between inclusion of a partial product or not, mean delay and area overhead beyond this estimate.

Optimising partial product generation and accumulation Originally, we opted for the Horner-based strategy because it provided an easy, direct way to develop an iterative algorithm and associated circuit design. Now we want a different outcome though, so what other strategy could we use? Well, recall the expansion of $r = y \cdot x$ using summation of separate terms as reproduced below

$$\begin{array}{lcccccccc} y \cdot x & = & y_0 \cdot x \cdot 2^0 & + & y_1 \cdot x \cdot 2^1 & + & y_2 \cdot x \cdot 2^2 & + & y_3 \cdot x \cdot 2^3 \\ & = & 0 \cdot x \cdot 2^0 & + & 1 \cdot x \cdot 2^1 & + & 1 \cdot x \cdot 2^2 & + & 1 \cdot x \cdot 2^3 \\ & = & 0 \cdot x & + & 2 \cdot x & + & 4 \cdot x & + & 8 \cdot x \\ & = & 14 \cdot x & & & & & & \end{array}$$

Developing a circuit design directly from this expression is surprisingly easy: we simply need to generate each of the terms, representing partial products, then add them up. Figure 11 is the combinatorial, $(n \times n)$-bit **tree multiplier** that results. It can be viewed as three layers which, from top-to-bottom, are as follows:

1. The top layer is comprised of n groups of n AND gates: the i-th group computes $x_j \wedge y_i$ for $0 \leq j < n$, meaning it outputs either 0 if $y_i = 0$ or x if $y_i = 1$.

   You can think of the AND gates as performing all n^2 possible (1×1)-bit multiplications of some x_j and y_i. Or, think of them as a less general form of multiplexer, selecting between 0 and x based on y_i.

2. The middle layer is comprised of n left-shift components. The i-th component shifts by a fixed distance of i bits, meaning the output is either 0 if $y_i = 0$, or $x \cdot 2^i$ if $y_i = 1$. Put another way, the output of the i-th component in the middle layer is

   $$y_i \cdot x \cdot 2^i$$

   i.e., some i-th partial product in the long-hand description of $y \cdot x$.

3. The bottom layer is a balanced, binary tree of adder components: these accumulate the partial products resulting from the middle layer, meaning the output is

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

   as required.

The standard trade-off versus an iterative alternative is still obvious: a tree multiplier uses a *lot* of logic in comparison, but requires just one step to compute the result. Crucially however, it improves our initial, naive translation from iterative to combinatorial. First, to accumulate the partial products, we have moved from using a (linear) sequence of adder components to a tree: the tree has depth $\log_2(n)$, meaning a shorter critical path due to parallelism in the accumulation process (e.g., the adder relating to the terms $y_0 \cdot x \cdot 2^0$ and $y_1 \cdot x \cdot 2^1$ can operate at the same time as the one relating to terms $y_2 \cdot x \cdot 2^2$ and $y_3 \cdot x \cdot 2^3$). Second, generation of those partial products is now more directly realised using n^2 AND gates; this improves on the use of general-purpose multiplexers previously.

Although this is certainly an improvement, and one which already yields a usable design, some subtleties emerge if the block diagram is implemented as a concrete circuit. The first is that although the critical path may *look* like $O(\log_2(n))$ gate delays, we again need to account for the fact it runs through the adder at each level of the tree; again using a ripple-carry adder, the overall delay is then more like $O(n \log_2(n))$. Even this is a bit optimistic however, because another subtlety is that the adders lower-down in the tree need to be *wider* than those higher-up. For example, the first level adds two n-bit partial products to produce a $(n + 1)$-bit intermediate result; the intermediate results get larger and larger until the last level adds two $(2n - 1)$-bit intermediate values to produce the $2n$-bit result.

5.4.2 Wallace and Dadda tree multipliers

So, given one can level some criticisms at a tree multiplier design, can we do better? One high-level approach would be to capitalise on the use of carry-save rather than ripply-carry addition: remember this "breaks" the carry chain acting as a limit, and is particularly effective when we need to accumulate multiple operands (which in this case are represented by the generated partial products). Alternatively, we could simply look in more detail at the block diagram in Figure 11, trying to identify cases that can be optimised at a lower-level (e.g., inside and between the blocks). Roughly speaking, the

1. **Wallace multiplier** [?], and
2. **Dadda multiplier** [?]

designs both employ a combination of *both* approaches with the overall goal of producing a combinatorial multiplier whose critical path is shorter.

Multiplier generation algorithms Somewhat like the original tree multiplier, Wallace and Dadda multipliers are comprised of a number of layers. More specifically, you can think of them as

1. an initial layer,
2. $O(\log n)$ layers of reduction, and
3. a final layer

You can again think of the first layer as generating the partial products, and the second and third layers as accumulating them; rather than perform the latter in one go using a tree of generic adders however, a carefully designed tree is employed. Crucially, each adder cell in a given reduction layer can operate in parallel; *unlike* the original tree multiplier therefore, each of the layers has an $O(1)$ critical path.

Rather than proceed as before, developing an algorithm and then translating it into a circuit design, Wallace and Dadda multipliers are *generated* by an algorithm. Given a value of n, a Wallace multiplier is generated by following these steps:

1. In the initial layer, multiply (i.e., AND) together each x_j with each y_i to produce a total of n^2 intermediate wires. Recall each wire (essentially the result of a 1-bit digit-multiplication) has a weight stemming from the digits in x and y, e.g., $x_0 \cdot y_0$ has weight 0, $x_1 \cdot y_2$ has weight 3 and so on.
2. Reduce the number of intermediate wires using layers composed of full and half adders:
   - Combine any three wires with same weight using a full-adder; the result in the next layer 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; the result in the next layer 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 layer.
3. In the final layer, after enough reduction layers, there will be at most two wires of any given weight: merge the wires to form two $2n$-bit values (padding as required), then add them together with an adder component.

The corresponding steps for a Dadda multiplier are similar, but we replace the second step:

1. In the initial layer, multiply (i.e., AND) together each x_j with each y_i to produce a total of n^2 intermediate wires. Recall each wire (essentially the result of a 1-bit digit-multiplication) has a weight stemming from the digits in x and y, e.g., $x_0 \cdot y_0$ has weight 0, $x_1 \cdot y_2$ has weight 3 and so on.
2. Reduce the number of intermediate wires using layers composed of full and half adders:
   - Combine any three wires with same weight using a full-adder; the result in the next layer is one wire of the same weight (i.e. the sum) one wire a higher weight (i.e. the carry).
   - If there are two wires with the same weight left, let w be that weight then:
     - If $w \equiv 2 \bmod 3$ then combine the wires using a half-adder; the result in the next layer 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 layer.
   - If there is only one wire with a given weight, just pass it through to the next layer.
3. In the final layer, after enough reduction layers, there will be at most two wires of any given weight: merge the wires to form two $2n$-bit values (padding as required), then add them together with an adder component.

to alter how the reduction layers are generated. This alteration means that wrt. the output of a given reduction layer, the number of wires with a given weight remains close to a multiple of three; this is ideal when using $3:2$ compressors as the means of combining them.

A small example Consider a small (or smaller than usually) example where we select $n = 4$ meaning 4-bit values of x and y. Since they are quite similar, we only consider the Wallace case in detail. The initial layer multiplies each x_j with each y_i for $0 \leq i, j < 4$, producing

- one weight-0 wire, i.e., $x_0 \cdot y_0$,
- two weight-1 wires, i.e., $x_0 \cdot y_1$ and $x_1 \cdot y_0$,
- three weight-2 wires, i.e., $x_0 \cdot y_2$, $x_2 \cdot y_0$, and $x_1 \cdot y_1$,
- four weight-3 wires, i.e., $x_0 \cdot y_3$, $x_3 \cdot y_0$, $x_1 \cdot y_2$, and $x_2 \cdot y_1$,
- three weight-4 wires, i.e., $x_1 \cdot y_3$, $x_3 \cdot y_1$, and $x_2 \cdot y_2$,
- two weight-5 wires, i.e., $x_2 \cdot y_3$, and $x_3 \cdot y_2$,

	Layer 1			Layer 2		
Weight	Input Wires	Operation	Output Wires	Input Wires	Operation	Output Wires
0	1	*PT*	1	1	*PT*	1
1	2	*HA*	1	1	*PT*	1
2	3	*FA*	2	2	*HA*	1
3	4	*FA*	3	3	*FA*	2
4	3	*FA*	2	2	*HA*	2
5	2	*HA*	2	2	*HA*	2
6	1	*PT*	2	2	*HA*	2
7	0		0	0		1

(a) *Using a Wallace-based multiplier.*

	Layer 1			Layer 2		
Weight	Input Wires	Operation	Output Wires	Input Wires	Operation	Output Wires
0	1	*PT*	1	1	*PT*	1
1	2	*PT*	2	2	*PT*	2
2	3	*FA*	1	1	*PT*	1
3	4	*FA*	3	3	*FA*	1
4	3	*FA*	2	2	*HA*	2
5	2	*PT*	3	3	*FA*	2
6	1	*PT*	1	1	*PT*	2
7	0		0	0		0

(b) *Using a Dadda-based multiplier.*

Figure 12: *A tabular description of stages in Wallace and Dadda designs for* (4×4)*-bit multiplication.*

- one weight-6 wire, i.e., $x_3 \cdot y_3$, and, finally,
- zero weight-7 wires.

Using these wires as input, we need two reduction layers that are detailed in Figure 12a. For example, in the first reduction layer

- there is one input wire with weight-0, so we use a pass-through operation (denoted *PT*) which results in one weight-0 wire as output,
- there are two input wires with weight-1, so we use a half-adder operation (denoted *HA*) which results in one weight-2 wire and one weight-4 wire as output, and
- there are three input wires with weight-2, so we use a full-adder operation (denoted *FA*) which results in one weight-4 wire and one weight-8 wire as output.

Figure 13 illustrates the structure of the generated Wallace multiplier resulting from this. In combination with the tabular description of the operations performed to generated it, highlights a few important points:

- we have one initial layer, $log_2(n) = \log_2(4) = 2$ reduction layers and one final layer,
- after the reduction layers we are left with at most two wires with any given weight, meaning we can form two values with $2n = 8$ bits which are added together by the final layer (which is basically a ripple-carry adder), and, crucially,
- there are no intra-layer carries within the a given reduction layer: the only carry chains that appear are inter-layer during reduction, or in the final layer.

Comparing Figure 12a with Figure 12a, we see that the Wallace and Dadda tree designs are similar in terms of latency: the main difference is in their area. More specifically, the Wallace tree used 6 half-adders and 4 full-adders, and the Dadda tree would use 1 half-adder and 5 full adders.

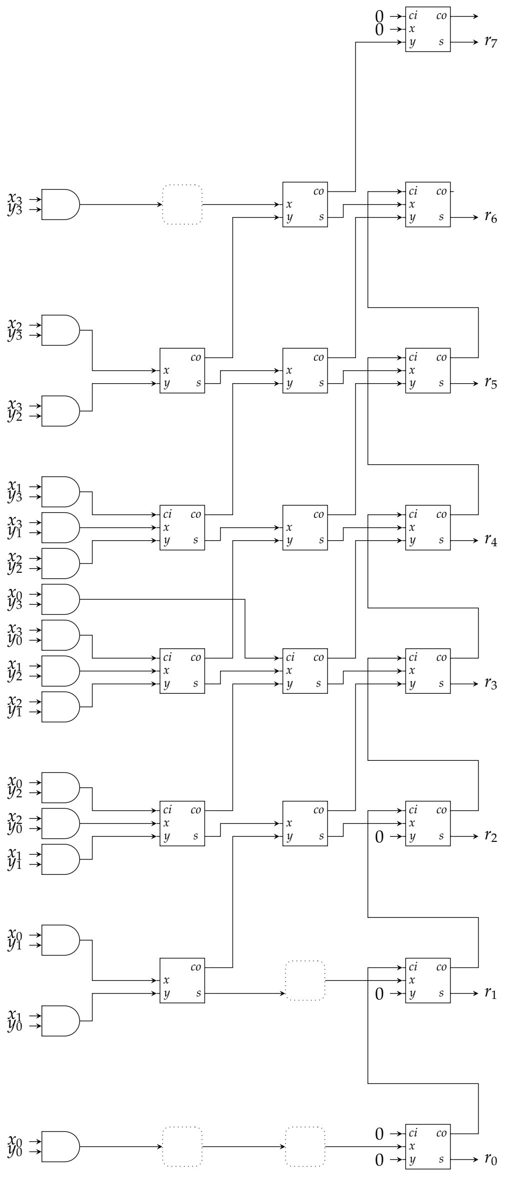
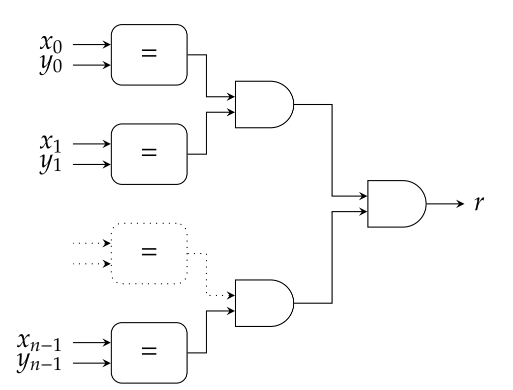
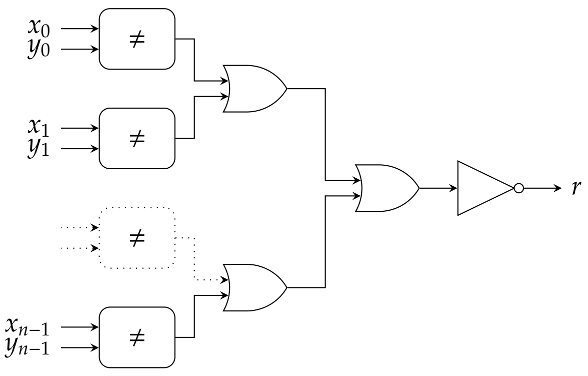


Figure 13: *A combinatorial,* (4×4)*-bit Wallace-based tree multiplier described using a circuit diagram.*



(a) *An AND plus equality comparator based design.*



(b) *An OR plus non-equality comparator based design.*

Figure 14: *An n-bit, unsigned equality comparison described using a circuit diagram.*

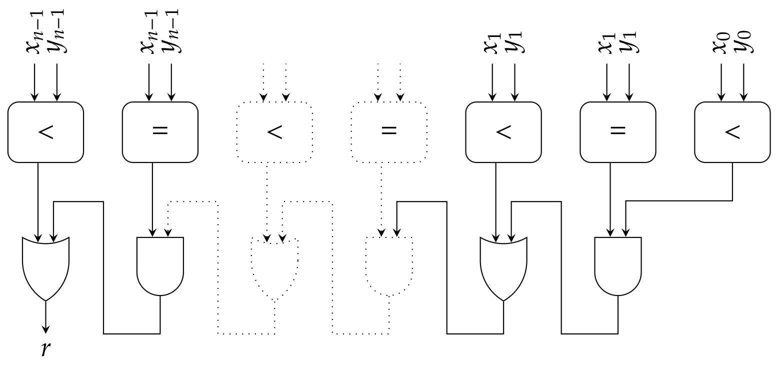


Figure 15: *An n-bit, unsigned less than comparison described using a circuit diagram.*

6 Components for comparison

Recall that like the 1-bit building blocks for addition (namely the half- and full-adder), in Chapter 2 we already covered designs for 1-bit equality and less than comparators; these require only a handful of logic gates to implement. Again as in the case of addition, the challenge is essentially how to extend these somehow. The idea of this Section is to tackle this step-by-step: first we consider designs only unsigned yet larger, n-bit, x and y, then we extend these designs to cope with signed x and y, and finally consider how to support other types of comparison.

6.1 Unsigned comparison

6.1.1 Equality

If some x and y are to be deemed equal, it must be the case that each digit of x is equal to the corresponding digit of y; formally, we must have $x_i = y_i$ for all $0 \le i < n$. Consider two examples using base-10:

$$\begin{array}{ll} x = 123_{(10)} & x = 121_{(10)} \\ y = 123_{(10)} & y = 123_{(10)} \end{array}$$

In the left-hand case we know $x = y$ since $x_i = y_i$ for all $0 \le i < 3$; in the right-hand case we know $x \ne y$ since there is at least one i (here $i = 0$) where $x_i \ne y_i$. The same strategy is true in *any* base, and fortunately in base-2 we already have a component that can perform the 1-bit comparison $x_i = y_i$: to cope with larger x and y, we just need to combine instances of it together.

Read out loud, the statement "if x_0 equals y_0 and x_1 equals y_1 and ... x_{n-1} equals y_{n-1} then x equals y, otherwise x does not equal y" highlights the basic strategy: each i-th of n instances of the 1-bit equality comparator will compare x_i and y_i, then we AND together the results. However, we need to take care wrt.

```
Input: Two unsigned, $n$-digit, base-$b$ integers $x$ and $y$
Output: If $x = y$ then true, otherwise false
for $i = n - 1$ downto $0$ step $-1$ do
    if $x_i \neq y_i$ then
        return false
    end
end
return true
```

Algorithm 15: An algorithm for equality comparison between base-b integers.

```
Input: Two unsigned, $n$-digit, base-$b$ integers $x$ and $y$
Output: If $x < y$ then true, otherwise false
for $i = n - 1$ downto $0$ step $-1$ do
    if $x_i < y_i$ then
        return true
    end
    else if $x_i > y_i$ then
        return false
    end
end
return false
```

Algorithm 16: An algorithm for less than comparison between base-b integers.

gate count. By looking at the truth table

x_i	y_i	$x_i \neq y_i$	$x_i = y_i$
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	1

it should be clear that the former is simply an XOR gate, while the latter needs an XOR and a NOT gate to implement directly. So we could either

1. use a dedicated XNOR gate whose cost is roughly the same as XOR given that

$$x \oplus y \;\equiv\; (x \wedge \neg y) \vee (\neg x \wedge y)$$

   and

$$x \overline{\oplus} y \;\equiv\; (\neg x \wedge \neg y) \vee (x \wedge y),$$

   or

2. compute $x =_u y \equiv \neg(x \neq_u y)$ instead, i.e., test whether x is *not* equal to y and then invert this result.

Both circuits are illustrated in Figure 14: it is important to see that both compute the same result, but use a different internal design motivated loosely by the standard cell library available (i.e., what gate types we can use and their relative efficiency in time and space).

6.1.2 Less Than

Determining whether some unsigned x is less than y is more involved than the case of equality. To help develop an approach, consider three examples using base-10:

$$\begin{array}{lll} x = 121_{(10)} & x = 323_{(10)} & x = 123_{(10)} \\ y = 123_{(10)} & y = 123_{(10)} & y = 123_{(10)} \end{array}$$

Intuitively, you might immediately say that in the left-hand case $x < y$, in the middle case $x > y$ and in the right-hand case $x = y$. Thinking in a little more depth however, *how* did you produce these answers? In your head, probably you did something like the following: work from the most-significant, left-most digits (i.e., x_{n-1} and y_{n-1}) towards the least-significant, right-most digits (i.e., x_0 and y_0) and at each i-th step, apply the following rules:

1. if $x_i < y_i$ then $x < y$,
2. if $x_i > y_i$ then $x > y$, but
3. if $x_i = y_i$ then we need to check the rest of x and y, i.e., move on to look at x_{i-1} and y_{i-1}.

So,

- in the left-hand case we find $x_i = y_i$ for $i = 2$ and $i = 1$ but $x_0 = 1 < 3 = y_0$ and conclude $x < y$,
- in the middle case, when $i = 2$, we find $x_2 = 3 > 1 = y_2$ and conclude $x > y$, while
- in the left-hand case, we find $x_i = y_i$ for all i and conclude $x = y$.

We can write this process as an algorithm, outlined in Figure 16: the loop iterates from the most- to least-significant digits of x and y, at each i-th step applying the rules above. That is, if $x_i < y_i$ then $x < y$ and if $x_i > y_i$ then $x \not< y$ since $x > y$; if $x_i = y_i$ the loop continues iterating, dealing with the next $(i-1)$-th step until it has processed all the digits. If the loop concludes, then we know that $x_i = y_i$ for all i and hence $x \not< y$ since $x = y$.

Of course when x and y are written in base-2, our task is easier still since each $x_i, y_i \in \{0, 1\}$ meaning we can reuse our existing 1-bit comparators. As such, the first step towards producing a design in terms of logic gates is to write the same process a little more formally. The idea is to recursively compute

$$\begin{array}{rcl} t_0 & = & (x_0 < y_0) \\ t_i & = & (x_i < y_i) \quad \vee \quad ((x_i = y_i) \wedge t_{i-1}) \end{array}$$

which matches our less formal rules above: at each i-th step, "x is less than y if $x_i < y_i$ *or* $x_i = y_i$ and comparing *the rest* of x is less than the rest of y". Consider an example with $n = 4$; we find

$$\begin{array}{rcl} t_0 & = & (x_0 < y_0) \\ t_1 & = & (x_1 < y_1) \quad \vee \quad ((x_1 = y_1) \wedge t_0) \\ t_2 & = & (x_2 < y_2) \quad \vee \quad ((x_2 = y_2) \wedge t_1) \\ t_3 & = & (x_3 < y_3) \quad \vee \quad ((x_3 = y_3) \wedge t_2) \end{array}$$

with the final result appearing as t_3. Giving values to x and y, $x = 5_{(10)} = 0101_{(2)}$ and $y = 7_{(10)} = 0111_{(2)}$ say, we can see that

$$\begin{array}{rclcrcl} x_0 < y_0 & = & \textbf{false} & & x_0 = y_0 & = & \textbf{true} \\ x_1 < y_1 & = & \textbf{true} & & x_1 = y_1 & = & \textbf{false} \\ x_2 < y_2 & = & \textbf{false} & & x_2 = y_2 & = & \textbf{true} \\ x_3 < y_3 & = & \textbf{false} & & x_3 = y_3 & = & \textbf{true} \end{array}$$

so

$$\begin{array}{rcl} t_0 & = & (x_0 < y_0) \\ & = & \textbf{false} \\ \\ t_1 & = & (x_1 < y_1) \vee ((x_1 = y_1) \wedge t_0) \\ & = & \textbf{true} \vee \textbf{false} \\ & = & \textbf{true} \\ \\ t_2 & = & (x_2 < y_2) \vee ((x_2 = y_2) \wedge t_1) \\ & = & \textbf{false} \vee \textbf{true} \\ & = & \textbf{true} \\ \\ t_3 & = & (x_3 < y_3) \vee ((x_3 = y_3) \wedge t_2) \\ & = & \textbf{false} \vee \textbf{true} \\ & = & \textbf{true} \end{array}$$

and since $t_3 = \textbf{true}$, we conclude that $x <_u y$. The good news is, this more formal process is written exactly in terms of logic gates: we have 1-bit comparators for $x_i < y_i$ and $x_i = y_i$, and each step simply requires one of each plus an AND and an OR gate. If we have n-bit x and y, we have n such steps as illustrated in Figure 15.

An aside: comparison using arithmetic.

It is tempting to avoid designing dedicated circuits for general-purpose comparison, by instead using arithmetic to make the task easier (or more special-purpose at least). Glossing over the issue of signed'ness, we know for example that

- $x = y$ is the same as $x - y = 0$, and
- $x < y$ is the same as $x - y < 0$

so we could re-purpose a circuit for subtraction to perform both tasks: we just compute $t = x - y$ and then claim

- $x = y$ iff. each $t_i = 0$, and
- $x < y$ iff. $t < 0$, or rather $t_{n-1} = 1$ given we are using two's-complement.

There idea here is that the general-purpose comparison of x and y is translated into a special-purpose comparison of t and 0.

This slight of hand seems attractive, but turns out to have some arguable disadvantages. Primarily, we need to cope with signed x and y, and hence deal with cases where $x - y$ overflows for example. In addition, one could argue a dedicated circuit for comparison can be more efficient than subtraction: even if we reuse one circuit for subtraction for both operations, cases might occur when this is not possible (e.g., in a micro-processor, where often we need to do both at the same time).

6.2 Signed comparison

Signed and unsigned equality comparison are equivalent, meaning we can use the unsigned comparison above in both cases. To see why, note the unsigned comparison we formulated basically tests whether each x_i is the same as y_i. For signed x and y, we of course do the same thing: remember x and y represent values (per Chapter 1), so if an x_i differs from y_i then the value x represents will differ from the one y represents irrespective of whether the representation is signed or unsigned.

Signed less than comparison is not as simple however; in this case we need an alternative design, which is produced using the unsigned comparison as a sub-component. In short, for $x <_s y$ we use a set of rules

$$\begin{array}{llcl} x \text{ +ve} & y \text{ -ve} & \mapsto & x \not<_s y \\ x \text{ -ve} & y \text{ +ve} & \mapsto & x <_s y \\ x \text{ +ve} & y \text{ +ve} & \mapsto & x <_s y \text{ if } \text{abs}(x) <_u \text{abs}(y) \\ x \text{ -ve} & y \text{ -ve} & \mapsto & x <_s y \text{ if } \text{abs}(y) <_u \text{abs}(x) \end{array}$$

The first two cases are obvious: if x is positive and y is negative it cannot ever be true that $x < y$ for example, while if x is negative and y is positive it is always true that $x < y$. For example, imagine we set $n = 4$:

1. if $x = +4_{(10)} \mapsto \langle 0, 0, 1, 0 \rangle_{(2)}$ and $y = -6_{(10)} \mapsto \langle 0, 1, 0, 1 \rangle_{(2)}$, then $x \not<_s y$ since x is +ve and y is -ve,
2. if $x = +6_{(10)} \mapsto \langle 0, 1, 1, 0 \rangle_{(2)}$ and $y = -4_{(10)} \mapsto \langle 0, 0, 1, 1 \rangle_{(2)}$, then $x \not<_s y$ since x is +ve and y is -ve,
3. if $x = -4_{(10)} \mapsto \langle 0, 0, 1, 1 \rangle_{(2)}$ and $y = +6_{(10)} \mapsto \langle 0, 1, 1, 0 \rangle_{(2)}$, then $x <_s y$ since x is -ve and y is +ve, and
4. if $x = -6_{(10)} \mapsto \langle 0, 1, 0, 1 \rangle_{(2)}$ and $y = +4_{(10)} \mapsto \langle 0, 0, 1, 0 \rangle_{(2)}$, then $x \not<_s y$ since x is -ve and y is +ve.

The other two cases need a little more explanation, but basically the idea is to consider the magnitudes only by computing then comparing abs(x) and abs(y), the absolute values of x and y. Notice that in the case where x and y are both negative, the order of comparison is flipped. This is because a larger negative x will actually be less than a smaller negative y (and vice versa); hence, when considering their absolute values the comparison is reversed. This is illustrated in the following:

1. if $x = +4_{(10)} \mapsto \langle 0, 0, 1, 0 \rangle_{(2)}$ and $y = +6_{(10)} \mapsto \langle 0, 1, 1, 0 \rangle_{(2)}$, then $x <_s y$ since x is +ve and y is +ve and $\text{abs}(x) = 4 <_u 6 = \text{abs}(y)$,

2. if $x = +6_{(10)} \mapsto \langle 0,1,1,0 \rangle_{(2)}$ and $y = +4_{(10)} \mapsto \langle 0,0,1,0 \rangle_{(2)}$, then $x \not<_s y$ since x is +ve and y is +ve and $\mathrm{abs}(x) = 6 \not<_u 4 = \mathrm{abs}(y)$,
3. if $x = -4_{(10)} \mapsto \langle 0,0,1,1 \rangle_{(2)}$ and $y = -6_{(10)} \mapsto \langle 0,1,0,1 \rangle_{(2)}$, then $x \not<_s y$ since x is -ve and y is -ve and $\mathrm{abs}(y) = 6 \not<_u 4 = \mathrm{abs}(x)$, and
4. if $x = -6_{(10)} \mapsto \langle 0,1,0,1 \rangle_{(2)}$ and $y = -4_{(10)} \mapsto \langle 0,0,1,1 \rangle_{(2)}$, then $x <_s y$ since x is -ve and y is -ve and $\mathrm{abs}(y) = 4 <_u 6 = \mathrm{abs}(x)$.

Since x and y are representing using two's-complement, we can make a slight improvement by rewrite the rules more simply as

$$\begin{array}{llcl}
x \text{ +ve} & y \text{ -ve} & \mapsto & x \not<_s y \\
x \text{ -ve} & y \text{ +ve} & \mapsto & x <_s y \\
x \text{ +ve} & y \text{ +ve} & \mapsto & x <_s y \text{ if } \mathrm{chop}(x) <_u \mathrm{chop}(y) \\
x \text{ -ve} & y \text{ -ve} & \mapsto & x <_s y \text{ if } \mathrm{chop}(x) <_u \mathrm{chop}(y)
\end{array}$$

where $\mathrm{chop}(x) = x_{n-2\ldots0}$, meaning $\mathrm{chop}(x)$ is x with the MSB (i.e., the bit which determines the sign of x) removed. The alternative rules work because a small negative integer becomes a large positive integer (and vice versa) when the MSB is removed; doing so is clearly easier than computing $\mathrm{abs}(x)$, in that we simply ignore the MSBs.

1. if $x = +4_{(10)} \mapsto \langle 0,0,1,0 \rangle_{(2)}$, $y = +6_{(10)} \mapsto \langle 0,1,1,0 \rangle_{(2)}$, $x <_s y$ since x is +ve and y is +ve and $\mathrm{chop}(x) = 4 <_u 6 = \mathrm{chop}(y)$,
2. if $x = +6_{(10)} \mapsto \langle 0,1,1,0 \rangle_{(2)}$, $y = +4_{(10)} \mapsto \langle 0,0,1,0 \rangle_{(2)}$, $x \not<_s y$ since x is +ve and y is +ve and $\mathrm{chop}(x) = 6 \not<_u 4 = \mathrm{chop}(y)$,
3. if $x = -4_{(10)} \mapsto \langle 0,0,1,1 \rangle_{(2)}$, $y = -6_{(10)} \mapsto \langle 0,1,0,1 \rangle_{(2)}$, $x \not<_s y$ since x is -ve and y is -ve and $\mathrm{chop}(x) = 4 \not<_u 2 = \mathrm{chop}(y)$, and
4. if $x = -6_{(10)} \mapsto \langle 0,1,0,1 \rangle_{(2)}$, $y = -4_{(10)} \mapsto \langle 0,0,1,1 \rangle_{(2)}$, $x <_s y$ since x is -ve and y is -ve and $\mathrm{chop}(x) = 2 <_u 4 = \mathrm{chop}(y)$.

The question is, finally, how do we implement these rules as a circuit? As in the case of overflow detection, we use the fact that testing the sign of x or y is trivial: we simply use the MSBs x_{n-1} and y_{n-1}. As a result, we can write

$$x <_s y = \begin{cases} \textbf{false} & \text{if } \neg x_{n-1} \wedge y_{n-1} \\ \textbf{true} & \text{if } x_{n-1} \wedge \neg y_{n-1} \\ \mathrm{chop}(x) <_u \mathrm{chop}(y) & \text{otherwise} \end{cases}$$

which can be realised using a multiplexer: the LHS is produced by selecting one of the options from the RHS depending on x_{n-1} and y_{n-1} which effectively act as the control signals.

6.3 Beyond equality and less than

Once we have components for equality and less than comparison, whether they are signed *or* unsigned, all other comparisons (in terms of the same signed'ness) can be derived using low-cost identities. For example, one can easily verify that

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

meaning the result of all six comparisons between x and y on the LHS can easily be realised using just

- one component for $x = y$,
- one component for $x < y$, and
- four (two NOT, and OR and an AND) extra logic gates

rather than instantiating additional, dedicated components.

BIBLIOGRAPHY (AND FURTHER READING)