# CONTENTS

# BASICS OF COMPUTER ARITHMETIC

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

– Babbage

*In Chapter **??**, 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 **??**, 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

In general terms, an **Arithmetic and Logic Unit (ALU)** is a component (or collection thereof) tasked with computation. The concept stems from the design of EDVAC by John von Neumann [9]: he foresaw that a 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: as part of a micro-processor, an ALU supports execution of instructions by computing results associated with arithmetic expressions such as `x + y` in a given C program.

One can view a concrete ALU at two levels, namely 1) at a low(er) level, in terms of how the constituent components themselves are designed, or 2) at a high(er) level, in terms of how said components are organised. In this Chapter we focus primarily on the former, which implies a focus on computer arithmetic. The challenge is roughly as follows: given one or more $n$-bit sequences that represent numbers, say $\hat{x}$ and $\hat{y}$, how can we design a component, i.e., a Boolean function $f$ we can then implement as a circuit, whose output represents an arithmetic operation? For example, if we want to compute

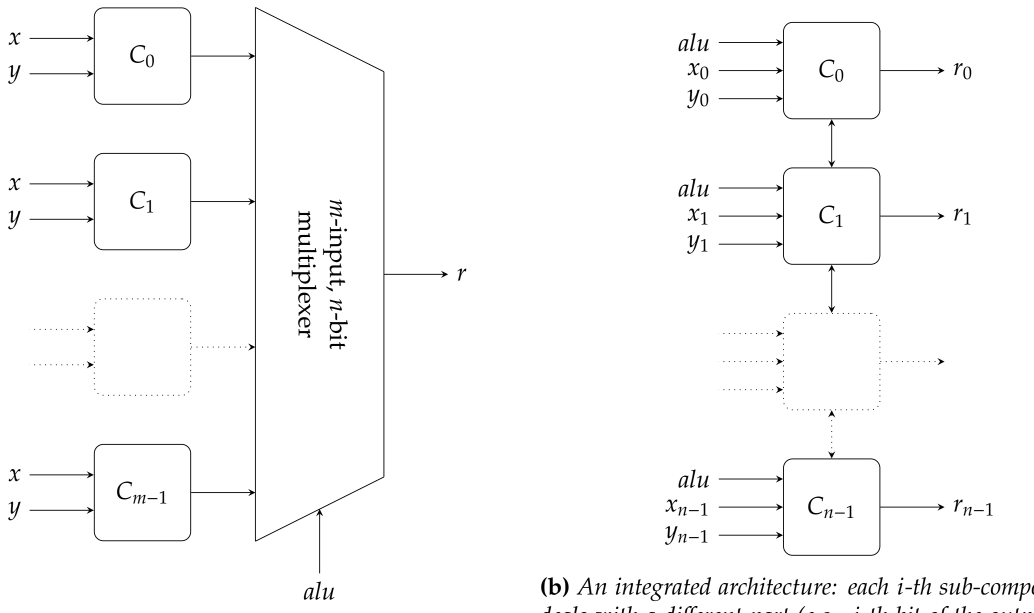
$$\hat{r} = f(\hat{x}, \hat{y}) \quad \mapsto \quad x + y,$$

i.e., an $\hat{r}$ that represents the sum of $\hat{x}$ and $\hat{y}$, how can we design a suitable function

$$f : \mathbb{B}^n \times \mathbb{B}^n \to \mathbb{B}^n$$

that realises the operation correctly while also satisfying additional design metrics once implemented as a circuit?

Often you will have *already* encountered long-hand, "school-book" techniques for arithmetic operations such as addition and multiplication. These allow you to perform the associated computation manually, which



**(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.*

can can be leveraged to address the challenge of designing such an $f$. That is, we can use an approach whereby we a) recap on your intuition about what the arithmetic operation means and works at a fundamental level, b) formalise this as an algorithm, then, finally, c) design a circuit to implement the algorithm (often by starting with a set of 1-bit building blocks, then later extending them to cope with $n$-bit inputs and outputs). Although effective, the challenge of applying this approach is magnified by what is typically a large **design space** of options and trade-offs. For example, we might implement $f$ using combinatorial components alone, or widen this remit by considering sequential components to support state and so on: with any approach involving a trade-off, the sanity of opting for one option over another requires careful analysis of the context.

After first surveying higher-level, architectural options for an abstract ALU, this Chapter deals more concretely with a set of low-level components: each Section basically applies the approach above to a different arithmetic operation. From here on, keep in mind that the scope is constrained by several simplifications:

1. The large design space of options for any given operation dictates we take a somewhat selective approach. A major example of this is our focus on integer arithmetic *only*: arithmetic with fixed- and floating-point numbers is an important topic, but we ignore it entirely and instead refer to [10, Part V] for a comprehensive treatment.

2. We use sequences of $n = 8$ bits to represent integers, assuming use of two's-complement representation where said integers are signed; any algorithms (eventually) operate in base-2 (i.e., set $b = 2$) as a result. Even so, most algorithms are developed and presented in a manner amenable to generalisation. For example, they often support larger $n$ or different $b$ with minimal alterations.

3. Having fixed the representation of integers, writing $\hat{x}$ is somewhat redundant: we relax this notation and simply write $x$ as a result. However, we introduce extra notation to clarify whether a given operation is signed or unsigned: for an operation $\odot$, we use $\odot_s$ and $\odot_u$ to denote signed and unsigned versions respectively. 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; use of a dedicated (so separate from the ALU) **Floating-Point Unit (FPU)** for floating-point computation is a common example.

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 [11, 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: $x + y = y + x$ but $x - y \neq y - x$.

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

**Example 0.1.** Consider the following unsigned, base-10 addition of $x = 107_{(10)}$ to $y = 14_{(10)}$:

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

If we write them naturally, it is clear that $|107_{(10)}| = 3$ and $|14_{(10)}| = 2$. However, the resulting mismatch will become inconvenient: in this example and from here on, we force $x$ and $y$ to have the same length by padding

### An aside: sign extension.

Although not an arithmetic operation per se, the issue of type conversion is a an important related concept none the less. Where such a conversion is performed explicitly (e.g., by the programmer) it is formally termed a **cast**, and where performed implicitly (or automatically, e.g., by the compiler) it is termed a **coercion**; either conversion, depending on the types involved, may or may not retain the same value due to the range of representable values involved.

As an example, imagine you write a C program that includes a cast of an $n$-bit integer `x` into an $n'$-bit integer `r`. Four cases can occur:

1. If `x` and `r` are unsigned and $n \leq n'$, `r` is formed by padding `x` with $n' - n$ bits equal to 0, at the most-significant end.
2. If `x` and `r` are signed and $n \leq n'$, `r` is formed by padding `x` with $n' - n$ bits equal to the sign bit (i.e., the MSB or $(n-1)$-th bit of `x`) at the most-significant end.
3. If `x` and `r` are unsigned and $n > n'$, `r` is formed by truncating `x`, i.e., removing (and discarding) $n - n'$ bits from the most-significant end.
4. If `x` and `r` are signed and $n > n'$, `r` is formed by truncating `x`, i.e., removing (and discarding) $n - n'$ bits from the most-significant end.

The second case above is often termed **sign extension**, and is required (vs. the first case) because simply padding `x` with 0 may turn it from a negative to positive value. For example, imagine $n = 16$ (i.e., the `short` type) and $n' = 32$ (i.e., the `int` type): if $\texttt{x} = -1_{(10)}$, the two options yield

$$\begin{array}{lclcr} \texttt{x} & = & 1111111111111111_{(2)} & = & -1_{(10)} \\ & & & & \\ \mathrm{ext}_0^{32}(\texttt{x}) & = & 00000000000000001111111111111111_{(2)} & = & 65535_{(10)} \\ \mathrm{ext}_{\pm}^{32}(\texttt{x}) & = & 11111111111111111111111111111111_{(2)} & = & -1_{(10)} \end{array}$$

where the latter retains the value of `x`, whereas the former does not.

them with more-significant zero digits. Although this may *look* odd, keep in mind this padding can be ignored without altering the associated value (i.e., we are confident $14_{(10)} = 014_{(10)}$, however odd the latter looks when written down).

Most people will have at least *seen* something similar to this, but, to ensure the underlying concept clear, $r$ is being computed by working 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 $y_{n-1}$) of the operands $x$ and $y$. In English, in each $i$-th step (or column, as read from right to left) we sum the $i$-th digits $x_i$ and $y_i$ and a **carry-in** $c_i$ (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 $r_i$ and a **carry-out** $c_{i+1}$ (for use by the next, $(i+1)$-th step). We call $c$ a (or the) **carry chain**, and say carries propagate from one step to the next.

This description can be written more formally: Algorithm 1 offers one way to do so. Notice the loop in lines #2 to #5 iterates through values of $i$ from 0 to $n-1$, with the body in lines #3 and #4 computing $r_i$ and $c_i$ respectively. You can read the latter as "if the sum of $x_i$, $y_i$ and $c_i$ is smaller than a single base-$b$ digit there is a carry into the next step, otherwise there is no carry". Notice that the algorithm sets $c_0 = ci$ to allow a carry-into the overall operation (in the example we assumed $ci = 0$), and $co = c_n$ allowing a carry-out; the sum of two $n$-digit integers is an $(n+1)$-digit result, but the algorithm produces an $n$-digit result $r$ and separate 1-digit carry-out $co$ (which you could, of course, think of as two parts of a single, larger result).

A reasonable question is why a *larger* carry (i.e., a $c_{i+1} > 1$) is not possible? To answer this, we should first note that although line #4 is written as a conditional statement, it could be rewritten st.

$$\begin{array}{lcl} r_i & \leftarrow & (x_i + y_i + c_i) \bmod b \\ c_{i+1} & \leftarrow & (x_i + y_i + c_i) \ \mathrm{div} \ b \end{array}$$

where mod and div are integer modulo and division: this makes more sense in a way, because the latter assignment can be read as "the number of whole multiples of $b$ carried into the next, $(i+1)$-th column". By considering bounds on (i.e., the maximum values of) each of the inputs, we can show $c_i \leq 1$ for $0 \leq i \leq n$. That is,

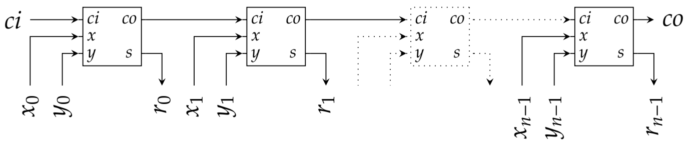


**Figure 2:** *An n-bit, ripple-carry adder described using a circuit diagram.*

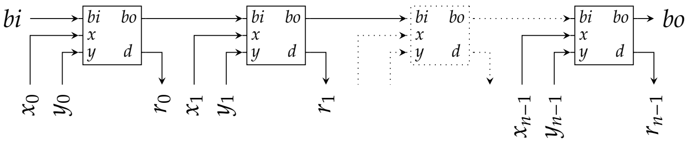


**Figure 3:** *An n-bit, ripple-carry subtractor described using a circuit diagram.*

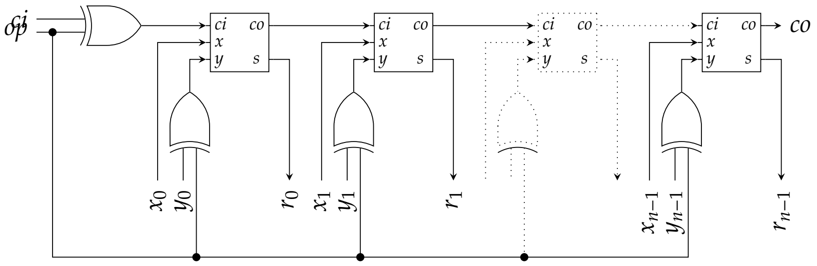


**Figure 4:** *An n-bit, ripple-carry adder/subtractor described using a circuit diagram.*

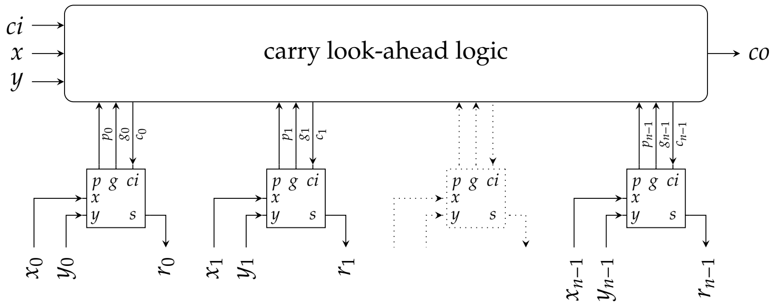


**Figure 5:** *An n-bit, carry look-ahead adder described using a circuit diagram.*

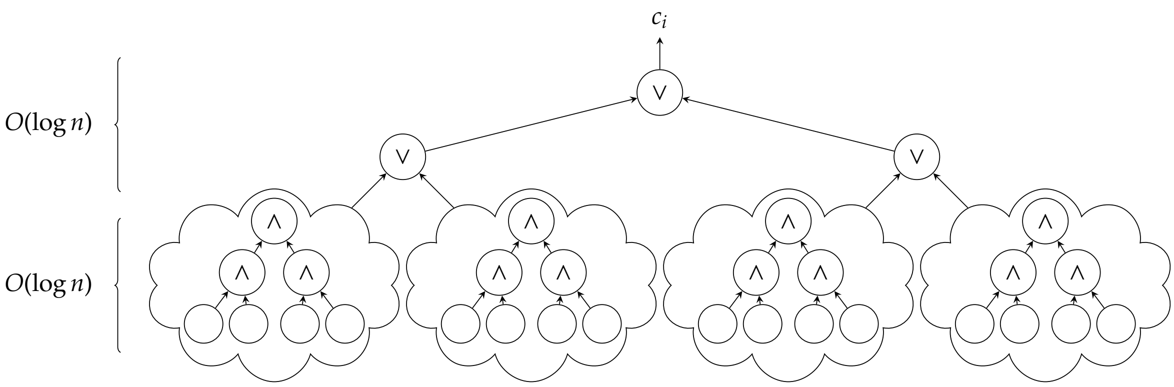


**Figure 6:** *An illustration depicting the structure of carry look-ahead logic, which is formed by an upper- and lower-tree of OR and AND gates respectively (with leaf nodes representing $g_i$ and $p_i$ terms for example).*

```
    Input: Two unsigned, $n$-digit, base-$b$ integers $x$ and $y$, and a 1-digit carry-in $ci \in \{0,1\}$
    Output: An unsigned, $n$-digit, base-$b$ integer $r = x + y$, and a 1-digit carry-out $co \in \{0,1\}$
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 $bi \in \{0,1\}$
    Output: An unsigned, $n$-digit, base-$b$ integer $r = x - y$, and a 1-digit borrow-out $bo \in \{0,1\}$
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.

1. in the 0-th step we compute $x_0 + y_0 + c_0$, which can be at most $(b-1)+(b-1)+1 = 2 \cdot b - 1$ (given we know $x_0, y_0 \in \{0, 1, \ldots, b-1\}$, and set $c_0 = ci \in \{0,1\}$ in line #1); as a result, $c_1$ can be at most $(2b-1) \operatorname{div} b = 1$,

2. in the $i$-th step we compute $x_i + y_i + c_i$, which can be at most $(b-1)+(b-1)+1 = 2 \cdot b - 1$ (given we know $x_i, y_i \in \{0, 1, \ldots, b-1\}$, and set $c_i \in \{0,1\}$ per the above); as a result, $c_{i+1}$ can be at most $(2b-1) \operatorname{div} b = 1$,

so we know (inductively) that the carry out of the $i$-th step into the next, $(i+1)$-th step is at most 1 (and so either 0 or 1); this is true no matter what value of $b$ is selected.

**Example 0.2.** Consider the following trace of Algorithm 1, for $x = 107_{(10)}$ and $y = 14_{(10)}$:

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

Throughout this Chapter, a similar style is used to describe step-by-step behaviour of an algorithm for specific inputs (particularly those which include one or more loops). 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 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.

An important feature in the presentation of Algorithm 1 is use of a *general* $b$: when invoking it, we can select any concrete value of $b$ we want. When discussing *representation* of integers, $b = 2$ was a natural selection because it aligned with concepts in Boolean algebra; the same is true here, within a discussion of *computation* involving such integers.

**Example 0.3.** Consider the following unsigned, base-2 addition of $x = 107_{(10)} = 01101011_{(2)}$ to $y = 14_{(10)} = 00001110_{(2)}$

$$
\begin{array}{rclcccccccccccc}
x & = & 107_{(10)} & \mapsto & & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & \\
y & = & 14_{(10)} & \mapsto & & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & + \\
\cline{6-13}
c & = & & & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & \\
r & = & 121_{(10)} & \mapsto & & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & \\
\cline{6-13}
\end{array}
$$

and the corresponding trace of Algorithm 1

| $i$ | $x_i$ | $y_i$ | $c_i$ | $r$ | $x_i + y_i + c_i$ | $c_{i+1}$ | $r_i$ | $r'$ |
|---|---|---|---|---|---|---|---|---|
| | | | | $\langle 0,0,0,0,0,0,0,0 \rangle$ | | | | $\langle 0,0,0,0,0,0,0,0 \rangle$ |
| 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 | $\langle 1,0,0,1,1,1,1,0 \rangle$ | | | | |

which produces $r = 01111001_{(2)} = 121_{(10)}$ as expected.

Better still, if we assume use of two's-complement (which we reasoned in Chapter **??** was sane), then the algorithm can compute the sum of *signed* $x$ and $y$ *without* change:

**Example 0.4.** Consider the following signed, base-2 addition of $x = 107_{(10)} \mapsto 01101011_{(2)}$ to $y = -14_{(10)} \mapsto 00001110_{(2)}$ (both represented using two's-complement)

$$
\begin{array}{rcrcccccccccc}
x & = & 107_{(10)} & \mapsto & & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 \\
y & = & -14_{(10)} & \mapsto & & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & + \\
\cline{5-13}
c & = & & & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
r & = & 93_{(10)} & \mapsto & & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\
\cline{5-13}
\end{array}
$$

and the corresponding trace of Algorithm 1

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

which produces $r = 01011101_{(2)} \mapsto 93_{(10)}$ as expected.

Intuitively, the *reason* no change is required is because both the unsigned and signed, two's-complement representations perfectly fit the definition of a positional number system: *they* express the value as a summation of weighted terms, whereas sign-magnitude, for example, needs a special case (namely the factor $-1^{x_{n-1}}$) to capture the sign. As a result of this feature the carry chain still functions in the same way, for example.

### 3.1.1 Ripple-carry adders

Having developed and reasoned about an algorithm for addition, the next challenge is to translate it into a concrete design we can implement as a circuit. At first glance this may seem difficult, not least because the algorithm contains a loop. Crucially however, we can *unroll* this loop once $n$ is fixed: this means we copy and paste the loop body (i.e., lines #3 and #4) $n$ times, replacing $i$ with the correct value in each $i$-th copy.

**Example 0.5.** Given $n = 4$, the loop in Algorithm 1 can be unrolled into the straight-line alternative

```
$c_0 \leftarrow ci$
$r_0 \leftarrow (x_0 + y_0 + c_0) \bmod b$
if $(x_0 + y_0 + c_0) < b$ then $c_1 \leftarrow 0$ else $c_1 \leftarrow 1$
$r_1 \leftarrow (x_1 + y_1 + c_1) \bmod b$
if $(x_1 + y_1 + c_1) < b$ then $c_2 \leftarrow 0$ else $c_2 \leftarrow 1$
$r_2 \leftarrow (x_2 + y_2 + c_2) \bmod b$
if $(x_2 + y_2 + c_2) < b$ then $c_3 \leftarrow 0$ else $c_3 \leftarrow 1$
$r_3 \leftarrow (x_3 + y_3 + c_3) \bmod b$
if $(x_3 + y_3 + c_3) < b$ then $c_4 \leftarrow 0$ else $c_4 \leftarrow 1$
$co \leftarrow c_4$
```

Notice that if we select $b = 2$, the body of the loop and therefore each replicated step in the unrolled alternative computes the 1-bit addition

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

Put another way, it matches the full-adder cell we produced a design for in Chapter **??**. Substituting one for the other, we simply have $n$ full-adder instances connected via respective carry-in and carry-out: each $i$-th instance computes the sum of $x_i$ and $y_i$ and a carry-in $c_i$, and produces the sum $r_i$ and a carry-out $c_{i+1}$. The design, which is termed a **ripple-carry adder** since the carry "ripples" or propagates through the chain chain, is shown in Figure 2.

The algorithm and associated design satisfy the required functionality: they can compute the sum of $n$-bit addends $x$ and $y$. As such, one might question whether exploring *other* designs is necessary. Any metric applied to the design *may* provide some motivation, but the concept of critical path is particularly important here. Recall from Chapter **??** that the critical path of a circuit is defined as the longest sequential sequence of gates; here, the critical path runs through the entire circuit from the 0-th to the $(n-1)$-th full-adder instance. Put another way, the carry chain represents an order on the computation of digits in $r$: $r_i$ cannot be computed until $c_i$ is known, so the $i$-th step cannot be computed until *every* $j$-th step for $j < i$ is computed, due to the carry chain. This implies the critical path can be approximated by $O(n)$ gate delays; our motivation for exploring other designs is therefore the possibility of improving on this, and thus computing $r$ with lower latency (i.e., less delay).

### 3.1.2 Carry look-ahead adders

One approach to removing the constraint imposed by a carry chain might be to *separate* computation of the carry and sum. At first glance this probably seems impossible, or at least difficult: we argued above that the latter depends on the former! However, notice that we can say at least *something* about how each $i$-th step of the ripple-carry adder works independently of the others. We know for instance that

1. if $x_i + y_i > b - 1$ it *generates* a carry, i.e., sets $c_{i+1} = 1$ irrespective of $c_i$,
2. if $x_i + y_i = b - 1$ it *propagates* a carry, i.e., sets $c_{i+1} = 1$ iff. $c_i = 1$. and
3. if $x_i + y_i < b - 1$ it *absorbs* a carry, i.e., sets $c_{i+1} = 0$ irrespective of $c_i$.

**Example 0.6.** Consider the following unsigned, base-10 addition of $x = 456_{(10)}$ to $y = 444_{(10)}$

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

where the three rules above 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 limit 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 using base-2 makes application of the rules simple. In particular, imagine we use $g_i$ and $p_i$ to indicate whether the $i$-th step will generate or propagate a carry respectively. We can express these as

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

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 both expressions have the same meaning: if $x_i = 1$ and $y_i = 1$, then $g_i = 1$ so it does not *matter* what the corresponding $p_i$ is (given the OR will yield 1, since the left-hand term $g_i = 1$, irrespective of the right-hand term). 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 shown in Figure 6: both the bottom- and top-half are balanced binary trees (st. leaf nodes are $g_i$ and $p_i$ terms, and internal nodes are AND and OR gates respectively) that implement the SoP expression for a a given $c_i$. We are able to use this organisation as a result of having decoupled computation of $c_i$ from the corresponding $r_i$, which is, essentially, what yields an advantage: the critical path (i.e., the depth of the structure, or longest path from the root to some leaf) is *shorter* than for a ripple-carry adder. Stated in a formal way, the former is described by $O(\log n)$ gate delays due to the tree structure, and the latter by $O(n)$ as a result of the linear structure.

The resulting design is shown in Figure 5. In contrast with the ripple-carry adder design in Figure 2, all the full-adder instances are *independent*: the carry chain previously linking them has now been eliminated. Instead, the $i$-th such instance produces $g_i$ and $p_i$; these inputs are used by the carry look-ahead logic to produce $c_i$. The design hides an important trade-off, namely the associated gate count. Although we have reduced the critical path, the gate count is now much higher: a rough estimate would be $O(n)$ and $O(n^2)$ gates for a ripple-carry and carry look-ahead adders. It can therefore be attractive to combine several, small(er) carry look-ahead adders (e.g., 8-bit adders) in a large(r) ripple-carry configuration (e.g., to form a larger, 32-bit, adder).

### 3.1.3 Carry-save adders

A second approach to eliminating the carry chain is to bend the rules a little, and look at a slightly different problem. The ripple-carry and the carry look-ahead adder compute the sum of two addends $x$ and $y$; what happens if we consider *three* addends $x$, $y$, and $z$, and thus compute $x + y + z$ rather than $x + y$?

A **carry-save adder** offers a solution for this alternative problem. It is often termed a $3 : 2$ **compressor** because it *compresses* three $n$-bit inputs $x$, $y$ and $z$ into two $n$-bit outputs $r'$ and $c'$ (termed the **partial sum** and **shifted carry**). Put another way, a carry-save "adder" computers the *actual* sum $r = x + y + z$ in two steps: 1) a compression step produces a partial sum and shifted carry, then 2) an addition step combines them into the actual sum.

The first step amounts to replacing $c$ with $z$ in the ripple-carry adder design, meaning that for the $i$-th full-adder instance we have

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

Unlike the ripple-carry adder, where the instances are connected via a carry chain, the expressions for $r'_i$ and $c'_i$ only use the $i$-th digits of $x$, $y$, and $z$: computation of each $i$-th digit of $r'$ and $c'$ is independent. Crucially, this means each $r'_i$ and $c'_i$ can be computed at the same time; the critical path runs through just *one* full-adder instance, rather than *all* $n$ instances as in a ripple-carry adder.

**Example 0.7.** Consider computation of the partial sum and shifted carry from $x = 96_{(10)} = 01100000_{(2)}$, $y = 14_{(10)} = 00001110_{(2)}$ and $z = 11_{(10)} = 00001011_{(2)}$:

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

After computing $r'$ and $c'$, we combine them via the second step by computing $r = r' + 2 \cdot c'$ using a standard (e.g., ripple-carry) adder. You could think of this step as propagating the carries, now represented separately (from the sum) by $c'$.

**Example 0.8.** Consider computation of the actual sum from $r' = 01100101$ and $c' = 00001010$:

$$\begin{array}{rcrcrrrrrrrrrrl}
r' & = & & & & & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & \\
2 \cdot c' & = & & & & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & + \\
\cline{5-14}
c & = & & & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & \\
r & = & 121_{(10)} & \mapsto & & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & \\
\cline{5-14}
\end{array}$$

which produces $r = 001111001_{(2)} = 121_{(10)}$ as expected.

Given we need this step to produce $r$, it is reasonable to question to ask why we bother with this approach at all: it *seems* as if we are limited in the same way as if we used a ripple-carry adder in the first place. With $m = 1$ compression step, the answer is that we have a critical path of $O(1) + O(n)$ gate delays vs. $O(n) + O(n)$ if we used two ripple-carry adders (one to compute $t = x + y$, then another to compute $r = t + z$). The more general idea, however, is we compute *many* compression steps (i.e., $m > 1$) and then a *single*, addition step: if we do this, the cost associated with the addition step becomes less significant (i.e., is amortised) as $m$ grows larger. Later, in Section 5 when we look at designs for multiplication, the utility of this approach should become clear.

## 3.2 Subtraction

### 3.2.1 Re*designing* a ripple-carry adder

Subtraction is conceptually, and so computationally similar to addition. In essence, the same steps are evident: 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 *difference* of the $i$-th digits $x_i$ and $y_i$ and a **borrow-in** produced by the previous, $(i-1)$-th step; this difference is potentially smaller than zero, so we produce the $i$-th digit of the result and a **borrow-out** into the next, $(i+1)$-th step. This description is formalised in a similar way by Algorithm 2. 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 use.

**Example 0.9.** Consider the following unsigned, base-10 subtraction of $y = 14_{(10)}$ from $x = 107_{(10)}$

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

and the corresponding trace of Algorithm 2

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

which produces $r = 93_{(10)}$ as expected.

**Example 0.10.** Consider the following unsigned, base-2 subtraction of $y = 14_{(10)} = 00001110_{(2)}$ from $x = 107_{(10)} = 01101011_{(2)}$

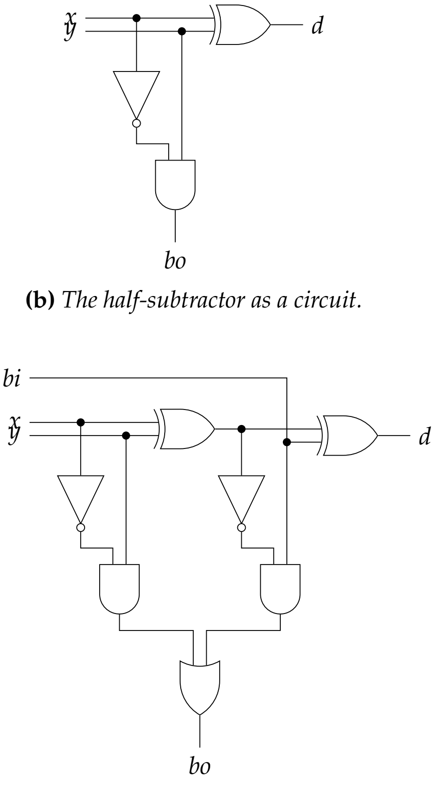
$$\begin{array}{rcrcrrrrrrrrrl}
x & = & 107_{(10)} & \mapsto & & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & \\
y & = & 14_{(10)} & \mapsto & & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & - \\
\cline{5-13}
c & = & & & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & \\
r & = & 93_{(10)} & \mapsto & & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & \\
\cline{5-13}
\end{array}$$

| $x$ | $y$ | $bo$ | $d$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 0 |

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

| $bi$ | $x$ | $y$ | $bo$ | $d$ |
|---|---|---|---|---|
| 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.*



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

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

**Figure 7:** *An overview of half- and full-subtractor cells.*

and the corresponding trace of Algorithm 2

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

which produces $r = 01011101_{(2)} = 93_{(10)}$ as expected.

Since the algorithm is more or less the same, it follow that a circuit to implement it would *also* be the same: Figure 2 illustrates this. The only difference, of course, is in the loop body, where we need the subtraction equivalent to half- and full-adder cells. More specifically, we need 1) a half-*subtractor* that 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 $bo$, and 2) a full-*subtractor* that extends a half-subtractor by including a borrow-in $bi$ as an additional input. Unsurprisingly, Figure 7 demonstrate the components themselves are simple to design and write as the Boolean expressions

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

and

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

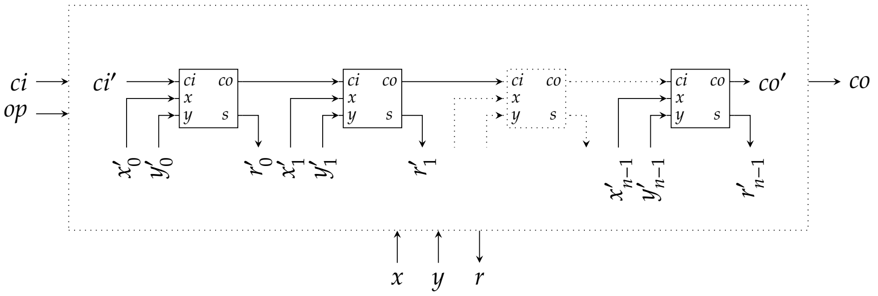
respectively. Keep in mind that $bi$ and $bo$ perform the same role as $ci$ and $co$ previously: the subtraction analogue of the ripple-carry adder, an $n$-bit ripple-*borrow* subtractor perhaps, is identical except for the *borrow* chain through all $n$ full-*subtractor* instances.

### 3.2.2 Re*using* a ripple-carry adder

As we have seen, subtraction is similar to addition. This is even more obvious still if we write $x - y \equiv x + (-y)$: the subtraction required (on the LHS) could be computed by adding $x$ to the negation of $y$ (on the RHS). Given we compute 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 might have

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

for example. Notice that as well as controlling computation of the sum or difference of $x$ and $y$, $op$ will control use of $ci$ as a carry- or borrow-in depending whether an addition of subtraction is computed. The advantage is that, at a high-level, the design



includes *one* internal adder. Versus *two* separate, similar components (i.e., an adder *and* a subtractor), this is already a useful optimisation outright; in designs for multiplication this will be amplified further.

The question is, how should we control the internal inputs to the adder (namely $x'$, $y'$ and $ci'$) st. given all the external inputs (namely $op$, $x$, $y$ and $ci$) the correct output $r$ is produced? By using two's-complement representation, we saw in Chapter **??** that

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

for any given $y$. The idea is to use this identity, translating 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 just captures what we said above: if $op = 0$ (in the top two rows) we want to compute $x + y + ci$, but if $op = 1$ (in the bottom two rows) we want to compute $x - y - ci$. Moving from left-to-right, we substitute in values of $ci$ then apply the identity for $-y$ in the bottom rows; the right-most table simply folds the constants together. In the right-most table, *all* the cases (for addition *and* subtraction, so where $op = 0$ or $op = 1$) are of the *same* form, which we can cope with using the internal adder: we have $op$, $x$, $y$ and $ci$, so 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 $y'_i = y_i \oplus op$. That is $x$ is unchanged whereas $y_i$ and $ci$ are XOR'ed with $op$ to conditionally invert them (in the bottom two rows, where we need $\neg y_i$ rather than $y_i$). Figure 4 illustrates the result, where it is important to see that the overhead, versus in this case a ripple-carry adder, is simply extra $n + 1$ XOR gates.

## 3.3 Carry and overflow detection

Consider the addition of some $n$-bit inputs $x$ and $y$: the magnitude of $r$ will be too large to represent in $n$ bits, st. it is incorrect, if either

1. $x$ and $y$ are (and hence the addition is) unsigned and there is a carry-out, *or*

2. $x$ and $y$ are (and hence the addition is) signed but the sign of $r$ makes no sense

which are termed **carry** and **overflow** errors respectively. The two cases can be illustrated using some (specific) examples:

**Example 0.11.** Consider the following unsigned, base-2 addition of $x = 15_{(10)} \mapsto 1111_{(2)}$ to $y = 1_{(10)} \mapsto 0001_{(2)}$

$$\begin{array}{rcrcccccccl}
x & = & 15_{(10)} & \mapsto & & 1 & 1 & 1 & 1 & \\
y & = & 1_{(10)} & \mapsto & & 0 & 0 & 0 & 1 & + \\ \cline{5-9}
c & = & & & 1 & 1 & 1 & 1 & 0 & \\
r & = & 0_{(10)} & \mapsto & & 0 & 0 & 0 & 0 & \\ \cline{5-9}
\end{array}$$

which produces an incorrect result $r = 0000_{(2)} \mapsto 0_{(10)}$ due to a carry error.

**Example 0.12.** Consider the following signed, base-2 addition of $x = -1_{(10)} \mapsto 1111_{(2)}$ to $y = 1_{(10)} \mapsto 0001_{(2)}$ (both represented using two's-complement)

$$\begin{array}{rcrcccccccl}
x & = & -1_{(10)} & \mapsto & & 1 & 1 & 1 & 1 & \\
y & = & 1_{(10)} & \mapsto & & 0 & 0 & 0 & 1 & + \\ \cline{5-9}
c & = & & & 1 & 1 & 1 & 1 & 0 & \\
r & = & 0_{(10)} & \mapsto & & 0 & 0 & 0 & 0 & \\ \cline{5-9}
\end{array}$$

which produces a correct result $r = 0000_{(2)} \mapsto 0_{(10)}$.

**Example 0.13.** Consider the following signed, base-2 addition of $x = 7_{(10)} \mapsto 0111_{(2)}$ to $y = 1_{(10)} \mapsto 0001_{(2)}$ (both represented using two's-complement)

$$\begin{array}{rcrcccccccl}
x & = & 7_{(10)} & \mapsto & & 0 & 1 & 1 & 1 & \\
y & = & 1_{(10)} & \mapsto & & 0 & 0 & 0 & 1 & + \\ \cline{5-9}
c & = & & & 0 & 1 & 1 & 1 & 0 & \\
r & = & -8_{(10)} & \mapsto & & 1 & 0 & 0 & 0 & \\ \cline{5-9}
\end{array}$$

which produces an incorrect result $r = 1000_{(2)} \mapsto -8_{(10)}$. due to an overflow error.

To deal with such errors in a sensible manner, we really need two steps: 1) detect that the error has occurred, then 2) apply some mechanism, e.g., to communicate or correct the error.

Detecting the carry error is simple: as suggested by the first example above, we need to inspect the carry-out. In this example, that assumes $n = 4$, the *correct* result $r = 16$ has a magnitude which cannot be accommodated in the number of bits available; an *incorrect* result $r = 0$ is therefore produced, with the carry-out (i.e., the fact that if the result is computed by Algorithm 1, it produces $co = 1$) signalling an error. However, notice that if we have *signed* $x$ and $y$, as in the second example, any carry-out is irrelevant: in this case, the result $r = 0$ is correct and the carry-out should be discarded.

This suggests detecting the overflow error requires more thought, with the third example suggesting a starting point. In this case, $x$ is the largest positive integer we can represent using $n = 4$ bits; adding $y = 1$ means the value wraps-around (as discussed in Chapter **??**) to form a negative result $r = -8$. Clearly this is impossible, in the sense that for 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):

$$\begin{array}{llllr}
x \text{ +ve} & y \text{ -ve} & & \Rightarrow & \text{no overflow} \\
x \text{ -ve} & y \text{ +ve} & & \Rightarrow & \text{no overflow} \\
x \text{ +ve} & y \text{ +ve} & r \text{ +ve} & \Rightarrow & \text{no overflow} \\
x \text{ +ve} & y \text{ +ve} & r \text{ -ve} & \Rightarrow & \text{overflow} \\
x \text{ -ve} & y \text{ -ve} & r \text{ +ve} & \Rightarrow & \text{overflow} \\
x \text{ -ve} & y \text{ -ve} & r \text{ -ve} & \Rightarrow & \text{no overflow}
\end{array}$$

Note that testing the sign of $x$ or $y$ is trivial, because it will be determined by their MSBs as a result of how two's-complement is defined: $x$ is positive, for example, iff. $x_{n-1} = 0$ and negative otherwise. Based on this, detection of an overflow error is computed as

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

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

**An aside: shift operators in C and Java.**

The fact there are two different classes of shift operation demands some care when writing programs; put simply, in a given programming language you need to make sure you select the correct operator. In C, both left- *and* right-shifts use the 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 right-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,

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 the 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 concepts and theory

### 4.1.1 Abstract shift operations, described as arithmetic

Although one would not normally think of doing a long-hand **shift**, as with addition or subtraction, it *is* possible to consider such an operation in arithmetic 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,

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

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 the sense it assumes a more-significant position. If $y$ is negative, on the other hand, it decreases the weight associated with $x_i$ and the digit "shifts" to the right; in this case, the operation acts as a division instead, because clearly

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

This argument applies for any $b$, and, as you might expect, we will ultimately be interested in $b = 2$ since this aligns with our approach for representing integers.

**Example 0.14.** Consider a base-10 shift of $x = 123_{(10)}$ by $y = 2$

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

which produces $r = x \cdot b^y = 123_{(10)} \cdot 10^2 = 12300_{(10)}$ as expected.

**Example 0.15.** Consider a base-10 shift of $x = 123_{(10)}$ by $y = -2$

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

which produces $r = x \cdot b^y = 123_{(10)} \cdot 10^{-2} = 1.23_{(10)}$ as expected.

**Example 0.16.** Consider a base-2 shift of $x = 51_{(10)} = 110011_{(2)}$ by $y = 2$

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

which produces $r = x \cdot b^y = 51_{(10)} \cdot 10^2 = 204_{(10)}$ as expected.

**Example 0.17.** Consider a base-2 shift of $x = 51_{(10)} = 110011_{(2)}$ by $y = -2$

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

which produces $r = x \cdot b^y = 51_{(10)} \cdot 10^{-2} = 12.75_{(10)}$ as expected.

### 4.1.2 Concrete shift (and rotate) operations of $n$-bit sequences

Recall from Chapter **??** that we represent signed or unsigned integers using an $n$-bit sequence *or* an equivalent literal. For example, wrt. 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: using the literal notation in what follows is more natural, but keep in mind that the equivalence above allows us to translate the same reasoning to *any* of the alternatives.

Based on our description the in previous Section, we need to consider what a shift operation *means* when applied to an integer represented by an $n$-bit sequence.

**Definition 0.1.** *Two types of shift can be applied to an $n$-bit sequence $x$:*

1. *a* **left-shift**, *where $y > 0$, can be defined as*

$$r \;=\; x \ll y \;=\; \underbrace{x_{n-1-\mathrm{abs}(y)} \parallel \cdots \parallel x_1 \parallel x_0 \parallel ??\ldots?}_{n\ bits}$$

*and*

2. *a* **right-shift**, *where $y < 0$, can be defined as*

$$r \;=\; x \gg y \;=\; \underbrace{??\ldots? \parallel x_{n-1} \parallel \cdots \parallel x_{\mathrm{abs}(y)+1} \parallel x_{\mathrm{abs}(y)}}_{n\ bits}$$

*where $y$ is termed the* **distance**, *and each ? represents a "gap" bit that must be filled to ensure $r$ has $n$ bits.*

**Definition 0.2.** *When computing a shift, any gap is filled in according to some rules:*

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 by the other end of $x$ rather than zero: that is,*

1. *a* **left-rotate** *(for which we use the operator $\lll$, vs. $\ll$ for the corresponding shift) yields a gap in the LSBs which is filled by the MSBs that would be discarded by a left-shift, and*

2. *a* **right-rotate** *(for which we use the operator $\ggg$, vs. $\gg$ for the corresponding shift) yields a gap in the MSBs which is filled by the LSBs that would be discarded by a right-shift.*

**Example 0.18.** Consider the base-2 shift and rotation of an $n = 8$ bit $x = 218_{(10)} = 11011010_{(2)} \equiv 11011010$ by a distance of $y = 2$:

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

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

These examples hopefully illustrate the somewhat convoluted definitions more clearly: in reality, the underlying concepts *are* reasonably simple. Consider the the logical left-shift: looking step-by-step at

$$\begin{array}{rcl} x \ll_u y &=& 11011010 \ll_u 2 \\ &=& 011010?? \\ &=& 01101000 \end{array}$$

the idea is that

1. we discard 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 to fill the resulting gap: this is a logical shift, so they are replaced with 0.

On the other hand, some more subtle points are important. First, note the importance of knowing $n$ when performing these operations. If we did *not* know $n$, or did not fix it say, a left-shift would just elongate the literal: instead of discarding MSBs, the literal grows to form an $n + y$ bit result. Likewise, if we do not know $n$ then rotate cannot be defined in a sane way; a left-rotate cannot fill the gap in the LSBs with discarded MSBs, because they are not discarded! Second, use of terminology including "left" and "right" explains why it is easier to reason about these operations by using literals. In short, doing so means the operations both have the intuitive effect by moving elements left or right: using a sequence, under our notation at least, the effect is counter-intuitive (i.e., the wrong way around). Third, and finally, 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, we typically assume a) the sign of $y$ is always positive (which is captured in the above via use of $\text{abs}(y)$), which is sane because we have specific left- and right-shift (or rotate) operations vs. one generic operation, and b) the magnitude of $y$ is restricted to $0 \leq y < n$ 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 let $r$ be undefined for any $y > n$ or $y < 0$ or consider $y$ modulo $n$.

Although logical and arithmetic left-shift are equivalent (i.e., a gap is zero-filled in both cases), this is not so for right-shift; as such, it is fair to question *why* arithmetic right-shift is included as a special case. Recall the original description above, where a shift of $x$ by $y$ was equated to a multiplication of $x$ by $b^y$. If $x$ uses a signed representation, a multiplication, and therefore also a shift, ideally preserves the sign: if $x$ is positive (resp. negative) then we expect $x \cdot b^y$ to be positive (resp. negative). This is essentially the purpose of an arithmetic right-shift, in the sense it preserves the sign of $x$ and hence has the correct arithmetic meaning. Both the underlying issue and impact of this special case is clarified by an example:

**Example 0.19.** Assuming $n = 8$, consider that

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

when represented using two's-complement. We know a shift using $y = -1$ *should* mean

$$x \cdot b^y = x \cdot 2^{-1} = x \cdot \frac{1}{2} = \frac{x}{2}.$$

However, using logical right-shift, we get

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

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

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

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 in $r$. This highlights the reason there is no need for a special case for arithmetic left-shift. With right-shift we fill MSBs of, so dictate the sign bit of, the result; in contrast, a left-shift means filling LSBs in the result, so the sign bit remains as is (i.e., is preserved by default).

## 4.2 Iterative designs

Imagine we want to shift some $x$ (wlog. left or right) by a distance of $y$, then shift that result *again* by a distance $y'$; a subtle but important fact is that we can combine the two shifts into one, i.e., we know that

$$(x \ll y) \ll y' \equiv x \ll (y + y').$$

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

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

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

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

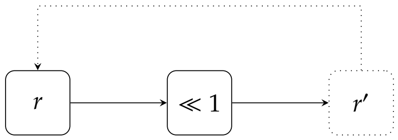


**Figure 8:** *An iterative design for n-bit (left-)shift described using a circuit diagram.*

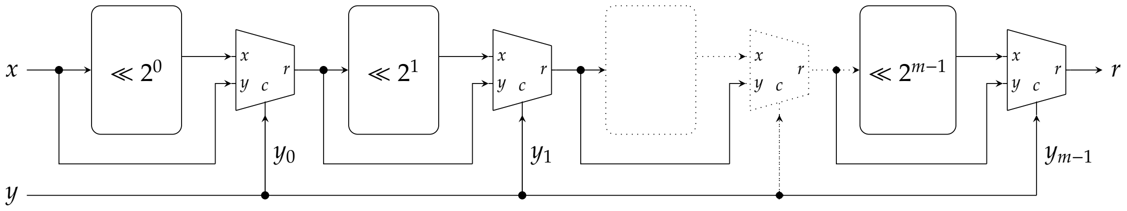


**Figure 9:** *A combinatorial design for n-bit (left-)shift described using a circuit diagram.*

**Example 0.20.** Consider the base-2, logical left-shift of an $n = 8$ bit $x = 218_{(10)} = 11011010_{(2)} \equiv 11011010$, first by a distance of $y = 2$ then by a distance of $y' = 4$:

$$\begin{array}{ccccccc}
(\,x \ll y\,) & \ll & y' & = & (\,11011010 \ll 2\,) & \ll & 4 \\
 & & & = & (\,01101000 \qquad\;\;) & \ll & 4 \\
 & & & = & 10000000 & & \\
 & & & & & & \\
x & \ll & (y + y') & = & 11011010 & \ll & (2+4) \\
 & & & = & 11011010 & \ll & 6 \\
 & & & = & 10000000 & &
\end{array}$$

Using the same reasoning, it should be obvious that if $y = 6$ then

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

Put another way, we can decompose one large shift on the LHS into several smaller shifts on the RHS: six repeated shifts each by a distance of 1 bit produce the same result as one shift by a distance of 6 bits. This approach is formalised by Algorithm 3.

**Example 0.21.** Consider the following trace of Algorithm 3, for $y = 6_{(10)}$:

| $i$ | $r$ | $r'$ | |
|---|---|---|---|
| | $x$ | | |
| 0 | $x$ | $x \ll 1$ | $r' \leftarrow r \ll 1$ |
| 1 | $x \ll 1$ | $x \ll 2$ | $r' \leftarrow r \ll 1$ |
| 2 | $x \ll 2$ | $x \ll 3$ | $r' \leftarrow r \ll 1$ |
| 3 | $x \ll 3$ | $x \ll 4$ | $r' \leftarrow r \ll 1$ |
| 4 | $x \ll 4$ | $x \ll 5$ | $r' \leftarrow r \ll 1$ |
| 5 | $x \ll 5$ | $x \ll 6$ | $r' \leftarrow r \ll 1$ |
| | $x \ll 6$ | | |

which produces $r = x \ll 6$ as expected.

Figure 8 captures the components required to implement this algorithm; the design highlights a trade-off between area and latency in which smaller area is favoured. Specifically, we only need a) a register to store $r$ (left-hand side), and b) a component to perform a 1-bit left-shift (center), which realises line #3) of Algorithm 3 and so needs no *actual* logic (since the shift distance is constant). However, this data-path demands an associated control-path that realises the loop. We can do so using an FSM of course: in each $i$-th step, the FSM latches $r'$ (representing the combinatorial result $r \ll 1$) into $r$ 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). Even so, the trade-off is essentially that we have a simple computational step but, as a result, need to iterate through $y$ such steps to compute the (eventual) result.

So far so good, but what about right-shift? Or, rotate?! Crucially, we can support the *entire* suite of shift-like operations via a fairly simple alteration to line #3 of Algorithm 3: we simply need to the component at the center of our design. For example, if we replace $r \leftarrow r \ll 1$ with $r \leftarrow r \gg 1$ we get a design that performs a right- vs. left-shift. Even better, if we replace $r \leftarrow r \ll 1$ with something more involved, namely

$$r \leftarrow \left\{ \begin{array}{lll}
r \ll 1 & \text{if } op = 0 \\
r \gg 1 & \text{if } op = 1 \\
r \lll 1 & \text{if } op = 2 \\
r \ggg 1 & \text{if } op = 3
\end{array} \right.$$

then provided we supply $op$ as an extra input, the resulting design can perform left- and right-shift, and left- and right-rotate: a multiplexer, controlled by $op$, decides which result of (produced by each different, individual operation) to update $r$ 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 $y$. One can view this as an application of the unintegrated ALU architecture in Figure 1a, but at a lower (or internal, component) level vs. higher, ALU level.

### 4.3 Combinatorial designs

Again following the same reasoning as above, it should be clear that if $y = 6$ then

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

Although the example is the same, the underlying strategy is to express $y$ st. each smaller shift is by a power-of-two distance (i.e., by $2^i$ for some $i$). As such, if we write $y$ in base-2 then each bit $y_i$ tells us whether or not to shift by a distance derived from $i$: we can 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.

**Example 0.22.** Consider the following trace of Algorithm 3, for $y = 6_{(10)}$ st. $m = \lceil \log_2(n) \rceil = 3$:

| $i$ | $r$ | $y_i$ | $r'$ | |
|---|---|---|---|---|
| | $x$ | | | |
| 0 | $x$ | 0 | $x$ | $r' \leftarrow r$ |
| 1 | $x$ | 1 | $x \ll 2$ | $r' \leftarrow r \ll 2^1$ |
| 2 | $x \ll 2$ | 1 | $x \ll 6$ | $r' \leftarrow r \ll 2^2$ |
| | $x \ll 6$ | | | |

which produces $r = x \ll 6$ as expected.

Translating the algorithm into a corresponding design harnesses the same idea as the ripple-carry adder: once $n$ is known, we unroll the loop by copy and pasting the loop body (i.e., lines #3 to #5) $n$ times, replacing $i$ with the correct value in each $i$-th copy. Doing so given $y = 6$, for example, produces the straight-line alternative

```
$r \leftarrow x$
if $y_0 = 1$ then $r \leftarrow r \ll 2^0$
if $y_1 = 1$ then $r \leftarrow r \ll 2^1$
if $y_2 = 1$ then $r \leftarrow r \ll 2^2$
```

which makes it (more) clear that we are essentially performing a series of choices: if the $(i-1)$-th stage produces $t$ as output, the $i$-th uses $y_i$ to choose between producing $t$ or $t \ll 2^i$ for use by the $(i+1)$-th stage. All the shifts themselves are by fixed constants (which we already argued are trivial), so these stages are really just a cascade of multiplexers.

Figure 9 translates this idea into a concrete circuit. The trade-off between latency and area is swapped vs. that for the previous, iterative design. On one hand, the component is combinatorial: it takes 1 step to perform each operation (vs. $n$), whose latency is dictated by the critical path, and can do so *without* the need for an FSM. On the other hand, however, it is likely to use significantly more area (relating to the logic gates required for each multiplexer).

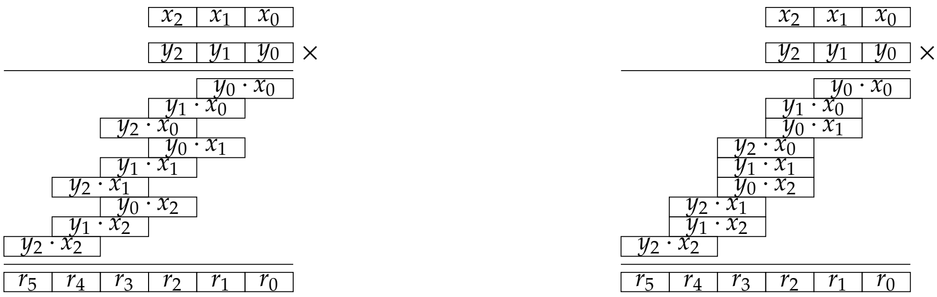
## 5 Components for multiplication

Formally, a multiplication operation[1] computes the **product** $r = y \cdot x$ based on the **multiplier** $y$ and **multiplicand** $x$. 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, $x$ could be 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, we covered several possible strategies with some associated trade-offs. This is exacerbated with multiplication, where a much larger design space exists. Even so, the same approach[2] is adopted: we again start by investigating the computation above from an algorithmic perspective, then somehow translate this into a design for a circuit we can construct from logic gates.

[1] Why write $y \cdot x$ rather than $x \cdot y$, which would match addition for example?! Since multiplication is commutative, we could 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 $[y]x$ sometimes used to denote general scalar multiplication.

[2] 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 $r = x^2$. See, for example, [10, Chapter 12.5].



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

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

**Figure 10:** *Two examples demonstrating different strategies for accumulation of base-b partial products resulting from two 3-digit operands.*

## 5.1 Introductory concepts and theory

### 5.1.1 Options for long-hand multiplication

**Example 0.23.** Consider the following unsigned, base-10 addition of $x = 623_{(10)}$ to $y = 567_{(10)}$:

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

The idea of long-hand multiplication is that to compute $r = y \cdot x$ (at the bottom) from $x$ and $y$ (at the top), we generate and then sum a set of **partial products** (in the middle): each $p_i$ is generated by 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 0.3.** *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 multiplied: if $x_j$ has weight $j$ and $y_i$ has weight $i$, the result will have weight $j + i$.*

Informally at least, this explains why each partial product is offset by some distance from the right-hand edge. In the example above, note that

- $y_0$ and $x_0$ have weight 0, so $p_0 = y_0 \cdot x_0 = 21_{(10)}$ has weight $0 + 0 = 0$,
- $y_1$ and $x_1$ have weight 1, so $p_4 = y_1 \cdot x_1 = 12_{(10)}$ has weight $1 + 1 = 2$, and
- $y_2$ and $x_1$ have weight 2 and 1 respectively, so $p_7 = y_2 \cdot x_1 = 10_{(10)}$ has weight $2 + 1 = 3$

st. $p_7$ is offset (or left-shifted) by 3 digits and so weighted by $10^3$: during summation of the partial products, it is representing $y_2 \cdot x_1 \cdot 10^3 = 10000_{(10)}$.

The question then is how to generate and sum the partial products. It turns out there are (at least) two strategies for doing so. These are described in Figure 10, which hightlights a difference wrt. how the digit-multiplications are managed. More specifically:

- The left-hand strategy is termed **operand scanning**, and is formalised by Algorithm 5. 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.
- The right-hand strategy is termed **product scanning**, and is formalised by Algorithm 6. 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 $y$.

```
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.

**Example 0.24.** Consider the following trace of Algorithm 5, which computes a base-10 operand scanning multiplication for $x = 623_{(10)}$ and $y = 567_{(10)}$

| $j$ | $i$ | $r$ | $c$ | $y_i$ | $x_j$ | $t = y_i \cdot x_i + r_{i+j} + c$ | $r'$ | $c'$ |
|---|---|---|---|---|---|---|---|---|
| | | $\langle 0,0,0,0,0,0\rangle$ | | | | | | |
| 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 |
| | | $\langle 1,4,2,3,5,3\rangle$ | | | | | | |

producing $r = 353241_{(10)}$ as expected.

**Example 0.25.** Consider the following trace of Algorithm 6, which computes a base-10 product scanning multiplication for $x = 623_{(10)}$ and $y = 567_{(10)}$

| $k$ | $j$ | $i$ | $r$ | $c_2$ | $c_1$ | $c_0$ | $y_i$ | $x_j$ | $t = y_i \cdot x_i$ | $r'$ | $c'_2$ | $c'_1$ | $c'_0$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | $\langle 0,0,0,0,0,0\rangle$ | 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 |
| | | | $\langle 1,4,2,3,5,3\rangle$ | | | | | | | | | | |

producing $r = 353241_{(10)}$ as expected.

Notice that given $n$-digit $x$ and $y$, we produce a larger $2n$-digit product $r = y \cdot x$; this can be rewritten as

$$y \cdot x = r_1 \cdot b^n + r_0$$

to show the $2n$-digit $r$ can be considered as two $n$-digit halves of the same size as $x$ and $y$. The reason for doing so is to stress the fact that although we typically want to compute $r$, *sometimes* it is enough to compute $r_0$: this so-called **truncated multiplication** basically just discards $r_1$, the $n$ most-significant digits of $r$ (or does not compute them in the first place), and retains $r_0$.

### 5.1.2 Multiplication as repeated addition

In Section 3.1, the study of long-hand addition led naturally to a implementable design: the ripple-carry adder in Figure 2 is a very direct translation of Algorithm 1. It is harder to make the same claim here, in the sense there is no (or at least a lot less of an) obvious route from one to the other. This suggests taking a step back to rethink what multiplication actually *means*.

At a more fundamental level than the long-hand approaches described, one can view multiplication as just repeated addition. Put another way,

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

```
Input: Two unsigned, $n$-digit, base-$b$ integers $x$ and $y$
Output: An unsigned, $2n$-digit, base-$b$ integer $r = y \cdot x$
1 $r \leftarrow 0$
2 for $i = 0$ upto $y - 1$ step $+1$ do
3     $r \leftarrow r + x$
4 end
5 return $r$
```

**Algorithm 7:** An algorithm for multiplication, using repeated addition (with $y$ treated as any integer).

```
Input: Two unsigned, $n$-digit, base-$b$ integers $x$ and $y$
Output: An unsigned, $2n$-digit, base-$b$ integer $r = y \cdot x$
1 $r \leftarrow 0$
2 for $i = 0$ upto $y - 1$ step $+1$ do
3     $r \leftarrow r + y_i \cdot x \cdot b^i$
4 end
5 return $r$
```

**Algorithm 8:** An algorithm for multiplication, using repeated addition (with $y$ is written in base-$b$).

st. 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: Algorithm 7 states the obvious, in the sense it captures this idea by simply adding $x$ to $r$ (which is initialised to 0) in a loop that iterates $y$ times. *Directly* using repeated addition is unattractive, however, since the number of operations performed relates to the magnitude of $y$. That is, we need $y - 1$ operations[3] in total, so for some $n$-bit $y$ we perform $O(2^n)$ operations: this grows quickly, even for modest values of $n$ (say $n = 32$).

Fortunately, improvements are easy to identify. Another way to look at the multiplication of $x$ by $y$ is as inclusion of an extra weight to the digits that describe $y$. That is, writing $y$ in base-$b$ yields

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

**Example 0.26.** Consider a base-2 multiplication of $x$ by $y = 14_{(10)} \mapsto 1110_{(2)}$, which we can expand into a sum of $n = 4$ terms as follows:

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

Intuitively, this should already *seem* more attractive: there are only $n$ terms (relating to the $n$ digits in $y$) in the summation so we only need $n - 1$, or $O(n)$, operations to compute their sum. Using a similar format to Algorithm 7, this is formalised by Algorithm 8. However, a problem lurks: line #3 of the algorithm reads

$$r \leftarrow r + y_i \cdot x \cdot b^i$$

or, put another way, our goal is to compute a multiplication but each step in doing so needs a further two multiplications itself! This is a chicken-and-egg style problem, but can be resolved by our selecting $b = 2$:

1. Multiplying $x$ by $y_i$ can be achieved without a multiplication: given we know $y_i \in \{0, 1, \ldots b - 1\} = \{0, 1\}$, if $y_i = 0$ then $y_i \cdot x = 0$, 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* $y_i$.

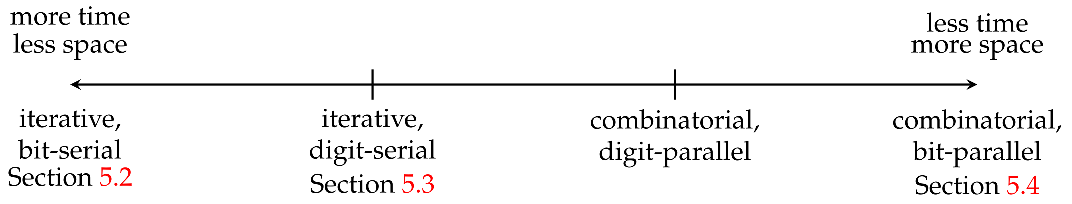
[3] Why $y - 1$ and not $y$: Algorithm 7 certainly performs $y$ iterations of the loop! If you think about it, although doing so would make it more complicated, the algorithm could be improved given we *know* the first addition (i.e., when $i = 0$) will add $x$ to 0. Put another way, we could avoid this initial addition and simply initialise $r \leftarrow n$ and perform one less iteration (i.e., $y - 1$ vs. $y$). Although the difference is minor, and so a distraction from the main argument here, you can see this more easily by counting the number of $+$ operators in the expansion above: for $y = 14$ we have 13 such additions.

2. Multiplying $r$ by 2 can be achieved without a multiplication: clearly $2 \cdot r = r + r = r \ll 1$, so we use a shift (or, if you prefer, an addition) instead.

So, in short, these facts mean the two multiplications in line #3 are pseudo-multiplications (or "fake" multiplications) because we can replace them with a non-multiplication equivalent whenever $b = 2$.

### 5.1.3 A high-level overview of the design space

Although the reformulations above might not seem useful *yet*, they represent an important starting point from which we can later construct various concrete algorithms and associated designs and implementations. Within the large design space of all possible options, we will focus on a selection summarised as follows:



You can think of options within this design space in a similar way to Section 4, where, for example, we overviewed options for the shift operation. Iterative options for multiplication typically deal with one (or at least few) partial products in each step; many (i.e., more than 1) steps and hence more time will be required to compute the result, but less space is required to do so (essentially because less computation is performed per step). Combinatorial options make the opposite trade-off, requiring just a single step to compute the result. However, the a) critical path, and so time said step takes due to the associated delay, and b) the space required, are typically both large(r). *Unlike* shift operations, a clear separation between iterative and combinatorial options is harder to make; trade-offs that blur the boundaries between some options are attractive, and explored where relevant.

### 5.1.4 Decomposition (or divide-and-conquer) techniques

Consider two designs which compute $r = y \cdot x$. Irrespective of *how* they compute $r$, they differ wrt. the limits placed on $x$ and $y$: the first design can deal with $n$-bit $y$ and $x$, whereas the second can only deal with smaller, $m$-bit values (st. $m < n$).

Within this context, consider the specific case of $m = \frac{n}{2}$ (in which we assume $n$ is even). As such, we can split $x$ and $y$ into two parts, i.e., write

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

where each $x_i$ and $y_i$ are $\frac{n}{2}$-bit integers. Likewise, we can write

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

where

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

and st. working through the multiplication as follows

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

demonstrates the result is correct. The more general, underlying idea is we decompose the single, larger $n$-bit multiplication into several, smaller $\frac{n}{2}$-bit multiplications: in this case, we compute the larger $n$-bit product $r$ using four $\frac{n}{2}$-bit multiplications (plus several auxiliary additions). In a sense, this is an instance of divide-and-conquer, a strategy often used in the design of algorithms: sorting algorithms such as merge-sort and quick-sort, for example, will decompose the problem of sorting a larger sequence into that of sorting several smaller sequences. The **Karatsuba-Ofman** [8] (re)formulation[4] offers further improvement, 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

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

Doing so requires three $\frac{n}{2}$-bit multiplications (although now there are more auxiliary additions and/or subtractions). This suggests a general trade-off: we could consider performing fewer, larger $n$-bit multiplications or more, smaller $\frac{n}{2}$-bit multiplications. If we accept the premise that designs for $\frac{n}{2}$-bit multiplication will be inherently less complex than an $n$-bit equivalent, this leads us to adopt one of (at least) two approaches:

1. instantiate and operate several smaller multipliers 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 multipliers (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, making a decision 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 both iterative *and* combinatorial designs; we focus on examples of the former, but keep in mind 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.

In simple terms, recoding $y$ means using a different representation: we represent the *same* value, but in a way that allows some sort of advantage during multiplication.

**Example 0.27.** Consider $y = 30_{(10)}$, here written in base-10. Among many options, some alternative representations for this value are

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

The first case is $y$ represented in base-2, but, after this, which is somewhat obvious given what we already know; after this, the cases less obviously represent the same value. However, it is important to see *why* they are equivalent. The second case requires a larger digit set. Each $y_i \in \{0, 1, 2, 3\}$ as a result of it using a base-4 representation, but, even so, we still have

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

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 offer any advantage. Intuitively, however, note two things:

[4] Other extensions and generalisations also exist. For example, we could apply the strategy recursively (i.e., decomposing the $\frac{n}{2}$-bit multiplications in a similar way), or attempt other forms of split (st. $x$ and $y$ are split into say 3 parts, rather than 2 as above).

```
Input: Two unsigned, $n$-bit, base-2 integers $x$ and $y$
Output: An unsigned, $2n$-bit, base-2 integer $r = y \cdot x$
r ← 0
for $i = n - 1$ downto 0 step −1 do
    $r \leftarrow 2 \cdot r$
    if $y_i = 1$ then
        $r \leftarrow r + x$
    end
end
return $r$
```

**Algorithm 9:** 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 $r = y \cdot x$
r ← 0
for $i = 0$ upto $n - 1$ step +1 do
    if $y_i = 1$ then
        $r \leftarrow r + x$
    end
    $x \leftarrow 2 \cdot x$
end
return $r$
```

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

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

2. the first case requires four non-zero base-2 digits to represent this $y$, but the forth case 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 select, however, it is *crucial* that 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 requires careful analysis, for example to judge the relative merits of a specific recoding strategy given a specific $n$.

## 5.2 Iterative, bit-serial designs

**Definition 0.4.** *As originally written,* **Horner's Rule** *[7] is a method for evaluating polynomials: it states that a polynomial $a(x)$ can be written as*

$$a_0 + a_1 \cdot x + \cdots + a^{n-1} \cdot x^{n-1} \equiv a_0 + x \cdot (a_1 + x \cdot (\cdots + x \cdot (a^n)))$$

*where the RHS factors out powers of the indeterminate $x$ from the LHS.*

This fact provides a starting point for an iterative multiplier design. Consider the similarity between a polynomial

$$a(x) = \sum_{i=0}^{i<n} a_i \cdot x^i$$

and an integer $y$ represented using a positional number system

$$y = \sum_{i=0}^{i<n} y_i \cdot b^i.$$

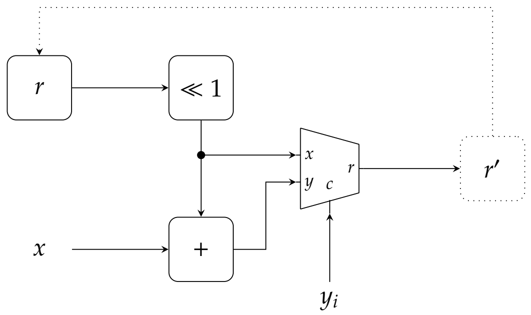


**Figure 11:** *An iterative, bit-serial design for $(n \times n)$-bit multiplication described using a circuit diagram.*

Put simply, there is no difference wrt. the form: only the *names* of variables are changed, plus $b$ represents an implicit parameter in the latter whereas $x$ is an explicit indeterminate in the former. As a result, we can consider a similar way of evaluating

$$y \cdot x = \sum_{i=0}^{i<n} y_i \cdot x \cdot b^i$$

which again has the same form.

**Example 0.28.** Consider a base-2 multiplication of $x$ by $y = 14_{(10)} \mapsto 1110_{(2)}$. As previously stated we would write

$$\begin{array}{rcl} y \cdot x & = & \sum_{i=0}^{i<n} y_i \cdot x \cdot b^i \\ & = & y_0 \cdot x \cdot 2^0 + y_1 \cdot x \cdot 2^1 + y_2 \cdot x \cdot 2^2 + y_3 \cdot x \cdot 2^3 \end{array}$$

but now this can be rewritten using Horner's Rule as

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

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

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, note 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. Putting everything together, to compute the result we maintain an accumulator $r$ that hold the current (or partial) result during evaluation; using $r$, the computation could be described as

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

- to realise each step of evaluation, apply a simple 2-part rule: first double the accumulated result (i.e., set $r$ to $2 \cdot r$), then add $y_i \cdot x$ to the accumulated result (i.e., set $r$ to $r + y_i \cdot x$).

Slightly more formally this implies iterative application of the rule

$$r \leftarrow y_i \cdot x + 2 \cdot r$$

which is further formalised by Algorithm 9: notice that line #1 realises the first point above while lines #3 to #6 realise the second point, with a loop spanning lines #2 to #7 iterating over them to realise each step. Although we will continue to focus on this approach, it is interesting to note, as an aside, that Algorithm 10 will yield the same result.

**Example 0.29.** Consider the following trace of Algorithm 9, for $y = 14_{(10)} \mapsto 1110_{(2)}$:

| $i$ | $r$ | $y_i$ | $r'$ | |
|---|---|---|---|---|
| | $0$ | | | |
| $3$ | $0$ | $1$ | $x$ | $r' \leftarrow 2 \cdot r + x$ |
| $2$ | $x$ | $1$ | $3 \cdot x$ | $r' \leftarrow 2 \cdot r + x$ |
| $1$ | $3 \cdot x$ | $1$ | $7 \cdot x$ | $r' \leftarrow 2 \cdot r + x$ |
| $0$ | $7 \cdot x$ | $0$ | $14 \cdot x$ | $r' \leftarrow 2 \cdot r$ |
| | $14 \cdot x$ | | | |

Algorithm 9 is termed the left-to-right variant, since it processes $y$ from the most- down to the least-significant bit (i.e., starting with $y_{n-1}$, on the left-hand end of $y$ when written as a literal).

**Example 0.30.** Consider the following trace of Algorithm 10, for $y = 14_{(10)} \mapsto 1110_{(2)}$:

| $i$ | $r$ | $x$ | $y_i$ | $r'$ | $x'$ | |
|---|---|---|---|---|---|---|
| | $0$ | $x$ | | | | |
| $0$ | $0$ | $x$ | $0$ | $0$ | $2 \cdot x$ | $x' \leftarrow 2 \cdot x$ |
| $1$ | $0$ | $2 \cdot x$ | $1$ | $2 \cdot x$ | $4 \cdot x$ | $r' \leftarrow r + x,\ x' \leftarrow 2 \cdot x$ |
| $2$ | $2 \cdot x$ | $4 \cdot x$ | $1$ | $6 \cdot x$ | $8 \cdot x$ | $r' \leftarrow r + x,\ x' \leftarrow 2 \cdot x$ |
| $3$ | $6 \cdot x$ | $8 \cdot x$ | $1$ | $14 \cdot x$ | $16 \cdot x$ | $r' \leftarrow r + x,\ x' \leftarrow 2 \cdot x$ |
| | $14 \cdot x$ | | | | | |

Algorithm 10 is termed the right-to-left variant, since it processes $y$ from the least- up to the most-significant bit (i.e., starting with $y_0$, on the right-hand end of $y$ when written as a literal).

Whereas the left-to-right variant only updates $r$, the right-to-left alternative updates $r$ *and* $x$; this may be deemed an advantage for the former, since we only need one register (at least one that is updated in any way, vs. simply a fixed input) 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? In both algorithms, the number of operations performed is dictated by the number of loop iterations: using Algorithm 9 as an example, in *each* iteration we a) always perform a shift to compute $r \leftarrow 2 \cdot r$, then b) conditionally perform an addition to compute $r \leftarrow r + x$ (which will be required 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* of $y$ (say $n = 8$ or $n = 32$) rather than the the *magnitude* of $y$ (say $2^n = 2^8 = 256$ or $2^n = 2^{32} = 4294967296$) as it was before.

Whether we use Algorithm 9 or Algorithm 10, the general strategy is termed **bit-serial** multiplication because use the 1-bit value $y_i$ in each iteration; the remaining challenge is to translate this strategy into a concrete design we can implement. We did something similar by translating Algorithm 3 into an iterative design for left-shift in Figure 8, so can adopt the same idea here: Figure 11 outlines a (partial) design that implements the loop body (in lines #3 to #6) of Algorithm 9. Notice that, as before,

- the left-hand side shows a register to store $r$ (i.e., the *current* value of $r$ at the start of the loop body),
- the right-hand side shows a register to store $r'$ (i.e., the *next* value of $r$ at the end of the loop body), and
- the middle shows some combinatorial logic that computes $r'$ from $r$: this is more complex than the left-shift case, but the idea is that a) the 1-bit left-shift component computes $r \ll 1 = 2 \cdot r$, then b) the multiplexer component selects between $2 \cdot r$ and $2 \cdot r + x$ (the latter of which is computed by an adder) depending on $y_i$.

To control this data-path, we again need an FSM: in each $i$-th step it will take $r'$ (representing $y_i \cdot x + 2 \cdot r$, per the above) and latches it back into $t$ ready for the $(i + 1)$-th step. A similar trade-off is again evident, in the sense that although we only need an adder and multiplexer (plus a register for $r$ and the FSM), the result will be computed after $n$ steps.

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

**Algorithm 11:** 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 $r = y \cdot x$
r ← 0
for $i = 0$ upto $n - 1$ step $+d$ do
    if $y_{i+d-1 \dots i} \neq 0$ then
        $r \leftarrow r + y_{i+d-1 \dots i} \cdot x$
    end
    $x \leftarrow 2^d \cdot x$
end
return $r$
```

**Algorithm 12:** An algorithm for multiplication of base-2 integers using a iterative, right-to-left, digit-serial strategy.

## 5.3 Iterative, digit-serial designs

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

By definition, a bit-serial multiplier processes a 1-bit digit of $y$ in each of $n$ steps. However, this actually represents a special case of more general **digit-serial** multiplication: a **digit size** $d$ is selected, and used to recode $y$ by splitting it into $d$-bit digits then processed in each of $\frac{n}{d}$ steps (noting that $d = 1$ is the special case referred to above). Provided $d$ divides $n$, extracting each $d$-bit digit from $y$ is easy: by writing $y$ in binary, recoding it to form $y'$ means splitting the sequence of bits into $d$-element sub-sequences.

**Example 0.31.** Consider $d = 2$, and some $y$ st. $n = 4$: this implies we process $\frac{n}{d} = \frac{4}{2} = 2$ digits in $y$, each of 2 bits. Based on what we covered originally, we already know, for example, that in base-2

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

The only change is to combine $y_0$ and $y_1$ into a *single* digit whose value is $y_0 + 2 \cdot y_1$; this is basically just treating the two 1-bit digits in $y$ as one 2-bit digit. By doing so, we can rewrite the expression as follows:

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

The term $y_{1 \dots 0}$ should be read as "the bits of $y$ from 1 down to 0 inclusive", so clearly $y_{1 \dots 0} \in \{0, 1, 2, 3\}$. As such, consider a base-2 multiplication of $x$ by $y = 14_{(10)} \mapsto 1110_{(2)}$:

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

To implement this new strategy, however, Algorithm 9 needs to be generalised for *any* $d$. Recall that for the special case of $d = 1$, we already saw and used a rule

$$r \leftarrow y_i \cdot x + 2 \cdot r.$$

Looking at the example above, a similar form

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

can be identified, which differs slightly in both left- and right-hand terms of the addition.

- The right-hand term is simple to accommodate. Rather than multiply $r$ by 2 as before, we now multiply it by $2^d$; we already know this can be realised by left-shifting $r$ by a distance of $d$, i.e., computing $2^d \cdot r \;\equiv\; r \ll d$.
- The left-hand term is more tricky. For $d = 1$, we needed to compute $y_i \cdot x$ but argued doing so was essentially a choice: because $y_i \in \{0, 1\}$, the result is either 0 or $x$. 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 rather than $2^1 = 2$, so either a) the choice is more involved, i.e., includes more cases, or b) we abandon the idea of it being a choice at all, instead using a combinatorial $(d \times n)$-bit multiplier to compute $y_{i \ldots i-d+1} \cdot x$ directly (related designs are covered in Section 5.4); you can view this component as replacing the multiplexer shown in Figure 11, which, by analogy, realised the $(1 \times n)$-bit multiplication $y_i \cdot x$.

Making these changes yields Algorithm 11. Note that line #4 could be implemented via either option above, and that, as outlined above, extracting the digit $y_{i \ldots i-d+1}$ from $y$ is simple enough that we view is as happening during multiplication, ignoring the need to formally recode $y$ into $y'$ beforehand. Either way, a clear advantage is already evident: we now require $\frac{n}{d}$ steps to compute the result.

**Example 0.32.** Consider the following trace of Algorithm 12, for $y = 14_{(10)} \mapsto 1110_{(2)}$:

| $i$ | $r$ | $y_{i \ldots i-d+1}$ | $r'$ | |
|---|---|---|---|---|
| | $0$ | | | |
| $3$ | $0$ | $11_{(2)} = 3_{(10)}$ | $3 \cdot x$ | $r' \leftarrow 2^2 \cdot r + 3 \cdot x$ |
| $1$ | $3 \cdot x$ | $10_{(2)} = 2_{(10)}$ | $14 \cdot x$ | $r' \leftarrow 2^2 \cdot r + 2 \cdot x$ |
| | $14 \cdot x$ | | | |

Assuming use of a combinatorial $(d \times n)$-bit multiplier, one way to think about a digit-serial multiplier is as a hybrid combination of iterative and combinatorial designs: it *is* iterative, in that it performs $\frac{n}{d}$ steps, but now each $i$-th such step utilises a $(d \times n)$-bit combinatorial multiplier component. Given we can select $d$, the hybrid can be configured to make a trade-off between time and space: larger $d$ implies fewer steps of computation but also a larger combinatorial multiplier, 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 $y = 15$? We already know that left-shifting $x$ by a fixed distance requires no computation, so a reasonable first answer might be to compute

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

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

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

Intuitively, this latter strategy should seems preferable given we only sum two terms rather than four. **Booth recoding** [2] is a standard recoding-based strategy for multiplication which generalises this example. Although various versions of the approach are considered in what follows, the advantages they *all* offer stem from use of a *signed* representation of $y$ and hence use of addition *and* subtraction operations.

### Original, base-2 Booth recoding

**Definition 0.5.** *Given a binary sequence $y$, a* **run** *of* 1 *(resp.* 0*) bits between $i$ and $j$ means $y_k = 1$ (resp. $y_k = 0$) for $i \leq k \leq j$; in simply terms, this means there is a sub-sequence of consecutive bits in $y$ whose value is* 1 *(resp.* 0*).*

**Example 0.33.** Consider $y = 30_{(10)} \mapsto 00011110_{(2)}$: we can clearly identify

- a run of one 0 bit between $i = 0$ and $j = 0$, i.e., $y_k = 0$ for $0 \leq k \leq 0$,
- a run of four 1 bits between $i = 1$ and $j = 4$, i.e., each $y_k = 1$ for $1 \leq k \leq 4$, and
- a run of three 0 bits between $i = 5$ and $j = 7$ i.e., each $y_k = 0$ for $5 \leq k \leq 7$.

As a starting point we consider base-2 Booth recoding; the idea is to identify a run of 1 bits in $y$ between $i$ and $j$, and then replace it with a single digit whose weight is $2^{j+1} - 2^{i}$.

**Example 0.34.** Consider $y = 30_{(10)} \mapsto 00011110_{(2)}$: since there is a run of four 1 bits between $i = 1$ and $j = 4$, and the fact that

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

we can recode

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

into

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

which clearly still represents the same value (albeit now via a signed digit set, st. $y'_i \in \{0, \pm 1\}$ vs. $y_i \in \{0, 1\}$). Using the same intuition as previously, the recoded $y'$ is preferable to $y$ because it has a lower weight (i.e., number of non-zero digits). We can see the impact this feature has by illustrating how such a $y'$ might be used during multiplication. Given $x = 6_{(10)} \mapsto 00000110_{(2)}$, for example, we would normally compute $r = y \cdot x$ as

$$\begin{array}{llrl*{16}{c}c}
x & = & 6_{(10)} & \mapsto &&&&&&&& & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & \\
y & = & 30_{(10)} & \mapsto &&&&&&&& & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & \times \\
 &&&&&&&&&&&&&&&&&&&& \\
\cline{5-20}
p_0 & = \;\; 0 \cdot x \cdot 2^{0} = & 0_{(10)} & \mapsto &&&&&&&& & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \\
p_1 & = +1 \cdot x \cdot 2^{1} = & +12_{(10)} & \mapsto &&&&&&& & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & & \\
p_2 & = +1 \cdot x \cdot 2^{2} = & +24_{(10)} & \mapsto &&&&&& & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & & & \\
p_3 & = +1 \cdot x \cdot 2^{3} = & +48_{(10)} & \mapsto &&&&& & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & & & & \\
p_4 & = +1 \cdot x \cdot 2^{4} = & +96_{(10)} & \mapsto &&&& & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & & & & & \\
p_5 & = \;\; 0 \cdot x \cdot 2^{5} = & 0_{(10)} & \mapsto &&& & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & & & \\
p_6 & = \;\; 0 \cdot x \cdot 2^{6} = & 0_{(10)} & \mapsto && & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & & & & \\
p_7 & = \;\; 0 \cdot x \cdot 2^{7} = & 0_{(10)} & \mapsto & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & & & & & \\
\cline{5-20}
r & = & 180_{(10)} & \mapsto & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & \\
\cline{5-20}
\end{array}$$

and thus accumulate four non-zero partial products. However, by first recoding $y$ into $y'$ we find

$$\begin{array}{llrl*{16}{c}c}
x & = & 6_{(10)} & \mapsto &&&&&&&& & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & \\
y & = & 30_{(10)} & \mapsto &&&&&&&& & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & \times \\
y'_{(2)} & = & 30_{(10)} & \mapsto &&&&&&&& & 0 & 0 & +1 & 0 & 0 & 0 & -1 & 0 & \\
\cline{5-20}
p_0 & = \;\; 0 \cdot x \cdot 2^{0} = & 0_{(10)} & \mapsto &&&&&&&& & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \\
p_1 & = -1 \cdot x \cdot 2^{1} = & -12_{(10)} & \mapsto & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & & \\
p_2 & = \;\; 0 \cdot x \cdot 2^{2} = & 0_{(10)} & \mapsto &&&&&& & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & \\
p_3 & = \;\; 0 \cdot x \cdot 2^{3} = & 0_{(10)} & \mapsto &&&&& & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & \\
p_4 & = \;\; 0 \cdot x \cdot 2^{4} = & 0_{(10)} & \mapsto &&&& & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & & \\
p_5 & = +1 \cdot x \cdot 2^{5} = & +192_{(10)} & \mapsto &&& & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & & & & & & \\
p_6 & = \;\; 0 \cdot x \cdot 2^{6} = & 0_{(10)} & \mapsto && & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & & & & \\
p_7 & = \;\; 0 \cdot x \cdot 2^{7} = & 0_{(10)} & \mapsto & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & & & & & \\
\cline{5-20}
r & = & 180_{(10)} & \mapsto & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & \\
\cline{5-20}
\end{array}$$

which requires accumulation of two non-zero partial products.

**Modified, base-4 Booth recoding** A base-2 Booth recoding already *seems* to produce what we want. However, there is a subtle problem: using $y'$ does not always yield an improvement over $y$ itself. This can be demonstrated by example:

**Example 0.35.** Consider $x = 6_{(10)} \mapsto 00000110_{(2)}$ and $y = 5_{(10)} \mapsto 00000101_{(2)}$, which, based on recoding $y$, we would compute $r = y \cdot x$ as

$$
\begin{array}{llrl*{16}{c}c}
x & = & 6_{(10)} & \mapsto & & & & & & & & & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & \\
y & = & 5_{(10)} & \mapsto & & & & & & & & & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & \times \\
y'_{(2)} & = & 5_{(10)} & \mapsto & & & & & & & & & 0 & 0 & 0 & 0 & +1 & -1 & +1 & -1 & \\
\cline{5-20}
p_0 & = -1 \cdot x \cdot 2^0 = & -6_{(10)} & \mapsto & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & \\
p_1 & = +1 \cdot x \cdot 2^1 = & +12_{(10)} & \mapsto & & & & & & & & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & & \\
p_2 & = -1 \cdot x \cdot 2^2 = & -24_{(10)} & \mapsto & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & & & \\
p_3 & = +1 \cdot x \cdot 2^3 = & +48_{(10)} & \mapsto & & & & & & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & & & & \\
p_4 & = \quad 0 \cdot x \cdot 2^4 = & 0_{(10)} & \mapsto & & & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & & \\
p_5 & = \quad 0 \cdot x \cdot 2^5 = & 0_{(10)} & \mapsto & & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & & & \\
p_6 & = \quad 0 \cdot x \cdot 2^6 = & 0_{(10)} & \mapsto & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & & & & \\
p_7 & = \quad 0 \cdot x \cdot 2^7 = & 0_{(10)} & \mapsto & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & & & & & \\
\cline{5-20}
r & = & 30_{(10)} & \mapsto & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & \\
\cline{5-20}
\end{array}
$$

This requires accumulation of four non-zero partial products, as would be the case if using $y$ as is.

Based on the original Booth recoding as a first step, to resolve this problem we employ a second recoding step based on the $y'_{(2)}$ we already have:

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

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

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

Given we originally had a signed base-2 recoding of $y$, we now have a signed base-4 recoding of the same $y$ (termed the modified Booth recoding): each pair represents a digit in $\{0, \pm 1, \pm 2\}$. Note that the two invalid (or impossible) pairs exists because of the original Booth recoding: we cannot encounter them, because the first recoding step will have already eliminated the associated run.

**Example 0.36.** Consider $x = 6_{(10)} \mapsto 00000110_{(2)}$ and $y = 5_{(10)} \mapsto 00000101_{(2)}$; based on the modified recoding, we would compute $r = y \cdot x$ as

$$
\begin{array}{llrl*{16}{c}c}
x & = & 6_{(10)} & \mapsto & & & & & & & & & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & \\
y & = & 5_{(10)} & \mapsto & & & & & & & & & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & \times \\
y'_{(2)} & = & 5_{(10)} & \mapsto & & & & & & & & & 0 & 0 & 0 & 0 & +1 & -1 & +1 & -1 & \\
y'_{(4)} & = & 5_{(10)} & \mapsto & & & & & & & & & & & & & & +1 & & +1 & \\
\cline{5-20}
p_0 & = +1 \cdot x \cdot 2^0 = & +6_{(10)} & \mapsto & & & & & & & & & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & \\
p_2 & = +1 \cdot x \cdot 2^2 = & +24_{(10)} & \mapsto & & & & & & & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & & & \\
p_4 & = \quad 0 \cdot x \cdot 2^4 = & 0_{(10)} & \mapsto & & & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & & \\
p_6 & = \quad 0 \cdot x \cdot 2^6 = & 0_{(10)} & \mapsto & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & & & & \\
\cline{5-20}
r & = & 30_{(10)} & \mapsto & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & \\
\cline{5-20}
\end{array}
$$

and this accumulate two non-zero partial products rather than four.

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

**Algorithm 13:** An algorithm for base-2 Booth recoding.

**An algorithm for Booth-based recoding** Both the first and second recoding steps above are still presented in a somewhat informal manner, because the goal was to demonstrate the idea; to make *use* of them in practice, we obviously need an algorithm. Fortunately, such an algorithm is simple to construct: notice that in a base-2 Booth recoding

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

and since these digits are paired to form the base-4 Booth recoding, each digit in that depends on $y_{i-1}$, $y_i$, *and* $y_{i+1}$. Thanks to this observation, the recoding process is easier than it may appear: assuming suitable padding of $y$ (i.e., $y_j = 0$ for $j < 0$ and $j \geq n$), we can produce digits of $y'$ from a 2- or 3-bit sub-sequence (or window) of bits in $y$ via

| Unsigned base-2 | | | Signed base-2 | | Signed base-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 |

Algorithm 13 and Algorithm 14 capture these rules in algorithms that produce a base-2 and base-4 recodings of a given $y$ respectively. Crucially, one can unroll the loop to produce a combinatorial circuit. For Algorithm 14 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 recoding as output. This implies that the recoding could be performed during rather than before the subsequent multiplication; the only significant overhead relates to increased area.

**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 should be more efficient than the bit-serial starting point. Algorithm 15 captures the result, which one can think of as a form of digit-serial multiplier: each iteration of the loop processes a digit of a recoding formed from multiple bits in $y$.

1. In Algorithm 9, $|y| = n$ dictates the number of loop iterations; Algorithm 11 improves this to $\frac{n}{d}$ for appropriate choices of $d$. In comparison, Algorithm 15 requires fewer, i.e.,

$$|y'| \simeq \frac{|y|}{2} \simeq \frac{n}{2},$$

**Input:** An unsigned, $n$-bit, base-2 integer $y$
**Output:** A base-4 Booth recoding $y'$ 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 14:** An algorithm for base-4 Booth recoding.

**Input:** An unsigned, $n$-bit, base-2 integer $x$, and a base-4 Booth recoding $y'$ of some integer $y$
**Output:** An unsigned, $2n$-bit, base-2 integer $r = y \cdot x$

```
1 $r \leftarrow 0$
2 for $i = |y'| - 1$ downto $0$ step $-1$ do
3     $r \leftarrow 2^2 \cdot r$
4     $r \leftarrow \begin{cases} r - 2 \cdot x & \text{if } y'_i = -2 \\ r - 1 \cdot x & \text{if } y'_i = -1 \\ r + 1 \cdot x & \text{if } y'_i = +1 \\ r + 2 \cdot x & \text{if } y'_i = +2 \end{cases}$
5 end
6 return $r$
```

**Algorithm 15:** An algorithm for multiplication of base-2 integers using an iterative, left-to-right, digit-serial strategy with base-4 Booth recoding.

iterations. As with the digit-serial strategy, to allow this to work, we need to compute $2^2 \cdot r$ in line #3 (rather than $2 \cdot r$), but this can be realised by left-shifting $r$ by a distance of $d$, i.e., computing $2^2 \cdot r \equiv r \ll 2$.

2. In Algorithm 9 we had $y_i \in \{0,1\}$ and in Algorithm 9 we had $y_{i \ldots i-d+1} \in \{0, 1, \ldots 2^d - 1\}$. In Algorithm 15, however, we have $y'_i \in \{0, \pm1, \pm2\}$. This basically means we have to test each non-zero $y'_i$ against more cases than before: line #4 captures them in one rather than use a more lengthy set of conditions. In short, dealing with $y'_i = -1$ vs. $y'_i = +1$ is easy: we simply subtract $x$ from $r$ rather than adding $x$ to $r$. In the same way, dealing with $y'_i = -2$ and $y'_i = +2$ mean subtracting (resp. adding) $2 \cdot x$ from (resp. to) $r$; since $2 \cdot x$ can be computed via a shift of $x$ (vs. an extra addition), there is no real overhead vs. subtracting (resp. adding) $x$ itself.

**Example 0.37.** Consider $y = 14_{(10)} \mapsto 1110_{(2)}$: we first use Algorithm 14 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 $y'$, then use Algorithm 15 as follows

| $i$ | $y'_i$ | $r$ | $r'$ | |
|---|---|---|---|---|
| | | 0 | | |
| 2 | +1 | 0 | $x$ | $r' \leftarrow 2^2 \cdot r + 1 \cdot x$ |
| 1 | 0 | $x$ | $4 \cdot x$ | $r' \leftarrow 2^2 \cdot r$ |
| 0 | −2 | $4 \cdot x$ | $14 \cdot x$ | $r' \leftarrow 2^2 \cdot r - 2 \cdot x$ |
| | | $14 \cdot x$ | | |

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

**Algorithm 16:** An algorithm for multiplication of base-2 integers using an iterative, left-to-right, digit-serial strategy with early termination.

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

### 5.3.3 Improvements via early termination: avoiding unnecessary iterations

**Example 0.38.** Consider digit-serial multiplication using $d = 2$ where $y = 30_{(10)} \mapsto 00011110_{(2)}$: as a first step we recode $y$ into $y' = \langle 10_{(2)}, 11_{(2)}, 01_{(2)}, 00_{(2)} \rangle$ by splitting the former sequence of bits into 2-bit sub-sequences. We then process $y'$ either left-to-right or right-to-left, as reflected by traces of Algorithm 11

| $i$ | $r$ | $y_{i...i-d+1}$ | $r'$ | |
|---|---|---|---|---|
| | $0$ | | | |
| $7$ | $0$ | $00_{(2)} = 0_{(10)}$ | $0$ | $r' \leftarrow 2^2 \cdot r$ |
| $5$ | $2 \cdot x$ | $01_{(2)} = 1_{(10)}$ | $1 \cdot x$ | $r' \leftarrow 2^2 \cdot r + 1 \cdot x$ |
| $3$ | $1 \cdot x$ | $11_{(2)} = 3_{(10)}$ | $7 \cdot x$ | $r' \leftarrow 2^2 \cdot r + 3 \cdot x$ |
| $1$ | $7 \cdot x$ | $10_{(2)} = 2_{(10)}$ | $30 \cdot x$ | $r' \leftarrow 2^2 \cdot r + 2 \cdot x$ |
| | $30 \cdot x$ | | | |

Algorithm 12

| $i$ | $r$ | $x$ | $y_{i+d-1...i}$ | $r'$ | $x'$ | |
|---|---|---|---|---|---|---|
| | $0$ | $x$ | | | | |
| $0$ | $0$ | $x$ | $10_{(2)} = 2_{(10)}$ | $2 \cdot x$ | $2^2 \cdot x$ | $r' \leftarrow r + 2 \cdot x,\ x' \leftarrow 2^2 \cdot x$ |
| $2$ | $2 \cdot x$ | $2^2 \cdot x$ | $11_{(2)} = 3_{(10)}$ | $14 \cdot x$ | $2^4 \cdot x$ | $r' \leftarrow r + 3 \cdot x,\ x' \leftarrow 2^2 \cdot x$ |
| $4$ | $14 \cdot x$ | $2^4 \cdot x$ | $01_{(2)} = 1_{(10)}$ | $30 \cdot x$ | $2^6 \cdot x$ | $r' \leftarrow r + 1 \cdot x,\ x' \leftarrow 2^2 \cdot x$ |
| $6$ | $30 \cdot x$ | $2^6 \cdot x$ | $00_{(2)} = 0_{(10)}$ | $30 \cdot x$ | $2^8 \cdot x$ | $r' \leftarrow r + 0 \cdot x,\ x' \leftarrow 2^2 \cdot x$ |
| | $30 \cdot x$ | | | | | |

respectively; both produce $r = 30 \cdot x$ as expected.

Notice that the 2 MSBs of $y$ are both 0, i.e., $y_7 = y_6 = 0$ st. $y_{7...6} = 0$ and hence $y'_3 = 00_{(2)}$. This fact can be harnessed to optimise both algorithms. Algorithm 11 processes $y'$ left-to-right so $y'_3$ is the *first* digit: the iteration where $i = 7$ extracts the digit l

$$y_{i...i-d+1} = y_{7...6} = y'_3 = 0.$$

As such, we *know* that

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

leaves $r$ unchanged: the condition $y_{i...i-d+1} \neq 0$ allows us to skip said update for $i = 7$, and thus be more efficient. Algorithm 12 processes $y'$ right-to-left so $y'_3$ is the *last* digit: the iteration where $i = 6$ extracts the digit l

$$y_{i+d-1...i} = y_{7...6} = y'_3 = 0.$$

The same argument applies here, in the sense we can skip the associated update of $r$. In fact, we can be more aggressive by skipping multiple such updates. If in some $i$-th iteration the digits processed by all $j$-th iterations for $j > i$ are zero, then we may as well stop: none of them will update $r$, meaning the algorithm can return it early as is (rather than perform extra iterations). This strategy is normally termed **early termination**; using Algorithm 12 as a starting point, it is realised by Algorithm 16.

**Example 0.39.** Consider the following trace of Algorithm 16 for $y = 30_{(10)} \mapsto 00011110_{(2)}$:

| $i$ | $r$ | $x$ | $y$ | $y_{d-1\ldots 0}$ | $r'$ | $x'$ | $y'$ | |
|---|---|---|---|---|---|---|---|---|
| | $0$ | $x$ | $00011110_{(2)}$ | | | | | |
| $0$ | $0$ | $x$ | $00011110_{(2)}$ | $10_{(2)} = 2_{(10)}$ | $2 \cdot x$ | $2^2 \cdot x$ | $00000111_{(2)}$ | $r' \leftarrow r + 2 \cdot x$, $x' \leftarrow x \cdot 2^2$, $y' \leftarrow y/2^2$ |
| $2$ | $2 \cdot x$ | $2^2 \cdot x$ | $00000111_{(2)}$ | $11_{(2)} = 3_{(10)}$ | $14 \cdot x$ | $2^4 \cdot x$ | $00000001_{(2)}$ | $r' \leftarrow r + 3 \cdot x$, $x' \leftarrow x \cdot 2^2$, $y' \leftarrow y/2^2$ |
| $4$ | $14 \cdot x$ | $2^4 \cdot x$ | $00000001_{(2)}$ | $01_{(2)} = 1_{(10)}$ | $30 \cdot x$ | $2^6 \cdot x$ | $00000000_{(2)}$ | $r' \leftarrow r + 1 \cdot x$, $x' \leftarrow x \cdot 2^2$, $y' \leftarrow y/2^2$ |
| | $30 \cdot x$ | | | | | | | |

Once $r$, $x$, and $y$ have been updated within the iteration for $i = 4$, we find $y' = 0$: this triggers the conditional statement, meaning $r$ is returned early after three (via line #9) vs. four (via line #12) iterations: the correct result $r = 30 \cdot x$ is produced as expected.

Although this should *seem* attractive, some trade-offs and caveats apply. First, the loop body, spanning lines #3 to #10 of Algorithm 16, is obviously more complex than the equivalent in Algorithm 12. Specifically, $r$, $x$, and $y$ all need to be updated, and the FSM controlling iteration needs to test $y$ and conditionally return $r$: this makes *it* more complex as well. Second, this added complexity, which typically means an increased area, only potentially (rather than *definitively*) reduces the latency of multiplication. Put simply, the number of iterations now depends on the value of $y$ (i.e., whether $y'$ contains more-significant digits that are 0 st. the algorithm can skip them), which we cannot know a priori: if this property does *not* hold, the algorithm will be no better than standard digit-serial multiplication.

## 5.4 Combinatorial designs

### 5.4.1 A vanilla tree multiplier

In Section 5.2, we made use of Horner's Rule as our starting point; the iterative nature by which the associated expression

$$y \cdot x = y_0 \cdot x + 2 \cdot (y_1 \cdot x + 2 \cdot (\cdots y_{n-1} \cdot x + 2 \cdot (0)))$$

was evaluated translated naturally into an iterative algorithm. For a combinatorial alternative, however, we adopt a different starting point and (re)consider

$$y \cdot x = \sum_{i=0}^{i<n} y_i \cdot x \cdot b^i .$$

Developing a design directly from this expression is surprisingly easy: we just need to generate each term, which represents a partial product, then add them up. Figure 13 is a (combinatorial) **tree multiplier** whose design stems from this idea. It can be viewed, from top-to-bottom, as three layers:

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 $(1 \times 1)$-bit multiplications of some $x_j$ and $y_i$, or a less general form of multiplexer that selects 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.

In Section 5.2, both iterative multiplier designs we produced made a trade-off: they required $O(n)$ time and $O(1)$ space, thus representing high(er) latency but low(er) area. Here we have more or less the exact opposite trade-off. The design is combinatorial, so takes $O(1)$ time where the constant involved basically represents the critical path. However, it *clearly* takes a lot more space; is is difficult to state formally how much, but the fact the design includes a tree of several adders vs. one adder hints this could be significant.

Beyond this comparison, it is important to consider various subtleties that emerge if the block diagram is implemented as a concrete circuit. First, notice that the critical path *looks* like $O(\log_2(n))$ gate delays, because this describes the depth of the (balanced) tree as used to form the bottom layer. However, because each node in said tree is *itself* an adder, the *actual* critical path is more like $O(n \log_2(n))$. Even this turns out to be optimistic: notice, second, that those adders lower-down in the tree (i.e., closer to the root) must be larger (and hence more complex) than those higher-up. This is simply because the intermediate results get larger; the first level adds two $n$-bit partial products to produce a $(n+1)$-bit intermediate result, whereas the last level adds two $(2n-1)$-bit intermediate values to produce the $2n$-bit result.

### 5.4.2 Wallace and Dadda tree multipliers

An obvious next question is whether and how we can improve the vanilla tree multiplier design. There are various possibilities, but one would be to focus on reducing the critical path and latency. One idea is to reimplement the tree using carry-save rather than ripply-carry adders; this is a natural replacement given the former was *specifically* introduced for use in contexts where we accumulate multiple inputs (or partial products). Another idea is to examine Figure 13 in detail, and identify features that can be optimised at a low(er)-level. Candidates might include where we can use half- vs. full-adder cells, which are, of course, less complex. The **Wallace multiplier** [13], and **Dadda multiplier** [5] designs employ a combination of *both* approaches, with the goal of reducing the critical path: they still represent combinatorial designs, but aim to have a smaller critical path and hence lower latency.

As with the vanilla tree multiplier, Wallace and Dadda multipliers are comprised of a number of layers. More specifically, you should think both as comprising

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

where the difference is, basically, how those layers are designed. The initial layer generates the partial products, then the second and third layers accumulate them; this is somewhat similar to the tree multiplier. However, rather than perform the latter using a tree of general-purpose adders, however, a carefully designed, special-purpose tree is employed. Producing a design for Wallace and Dadda multipliers follows a different process than we have used before. Rather than develop an algorithm then translate it into design, the multipliers are generated directly *by* an algorithm. Given a value of $n$ as input, Algorithm 17 and Algorithm 18 generate Wallace and Dadda multipliers respectively; both are described in three steps that mirror the layers above.

**Example 0.40.** Consider $n = 4$, where we want to produce a Wallace multiplier design that computes the product $r = y \cdot x$ for 4-bit $x$ and $y$; to do so, we use Algorithm 17.

An initial layer multiplies $x_j$ with $y_i$ for $0 \leq i, j < 4$, st. we produce

- 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 $x_3 \cdot y_2$,
- one weight-6 wire, i.e., $x_3 \cdot y_3$, and, finally,
- zero weight-7 wires.

Figure 12a details the subsequent two reduction layers. 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

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.

**Algorithm 17:** An algorithm to generate a Wallace tree multiplier design.

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).
   - 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.

**Algorithm 18:** An algorithm to generate a Dadda tree multiplier design.

| | 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 example* $(4 \times 4)$*-bit Wallace and Dadda tree multiplier designs.*

- 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.

The resulting design, including the final layer, is illustrated by Figure 14.

Notice that $n$ is the only input to the algorithm(s), so although the example is specific to $n = 4$ the general structure will remain similar. In fact, the example highlights some important general points:

- we have 1 initial layer, $\log_2(n) = \log_2(4) = 2$ reduction layers, and 1 final layer,
- the reduction layers yield at most two wires with a given weight; we form then sum two $2n$-bit values (e.g., using a ripple-carry adder) to produce the result, and, crucially,
- there are no intra-layer carries in the reduction layer(s): the only carry chains that appear are inter-layer, during reduction, or in the final layer.

Phrased as such, it should be clear *why* the concept of carry-save addition is relevant: the reduction and final layers employ essentially the same concept, by *compressing* many inputs into few(er) outputs until the point they can be summed to produce the result. If you look again at Algorithm 17 and Algorithm 18, the difference between the two is within the second step: in the Dadda design, the number of wires of a given weight remains, by-design, close to a multiple of three, which facilitate use of $3 : 2$ compressors as a means of reduction. As hinted at, a crucial feature of both designs is that each adder cell within the reduction layer(s) operates in parallel so has an $O(1)$ critical path; this suggests the overall critical path will be $O(1 + \log_2(n) + n) = O(n)$ gate delays in both cases. Comparing Figure 12a with Figure 12b, we see the main difference is wrt. space not time. More specifically, for $n = 4$ the Wallace multiplier uses 6 half-adders and 4 full-adders, and the Dadda multiplier would use 1 half-adder and 5 full adders; this is a trend that holds for larger $n$.

## 5.5 Some multiplier case-studies

One of the challenges outlined at the start of this Section related to the large design space wrt. multiplication; although we have only covered a sub-set of that design space, hopefully the challenge is already clear! As a result of the possible trade-offs, it is hard to identify a single "correct" design. This means different micro-processors, for example, *legitimately* opt for different designs so as to match their design constraints. In this Section, we

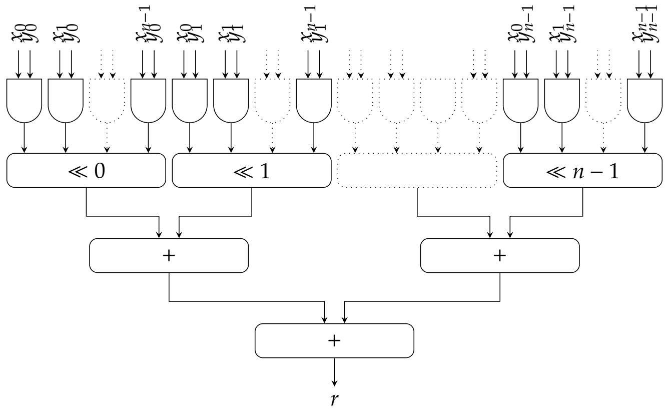


**Figure 13:** *An $(n \times n)$-bit tree multiplier design, described using a circuit diagram.*

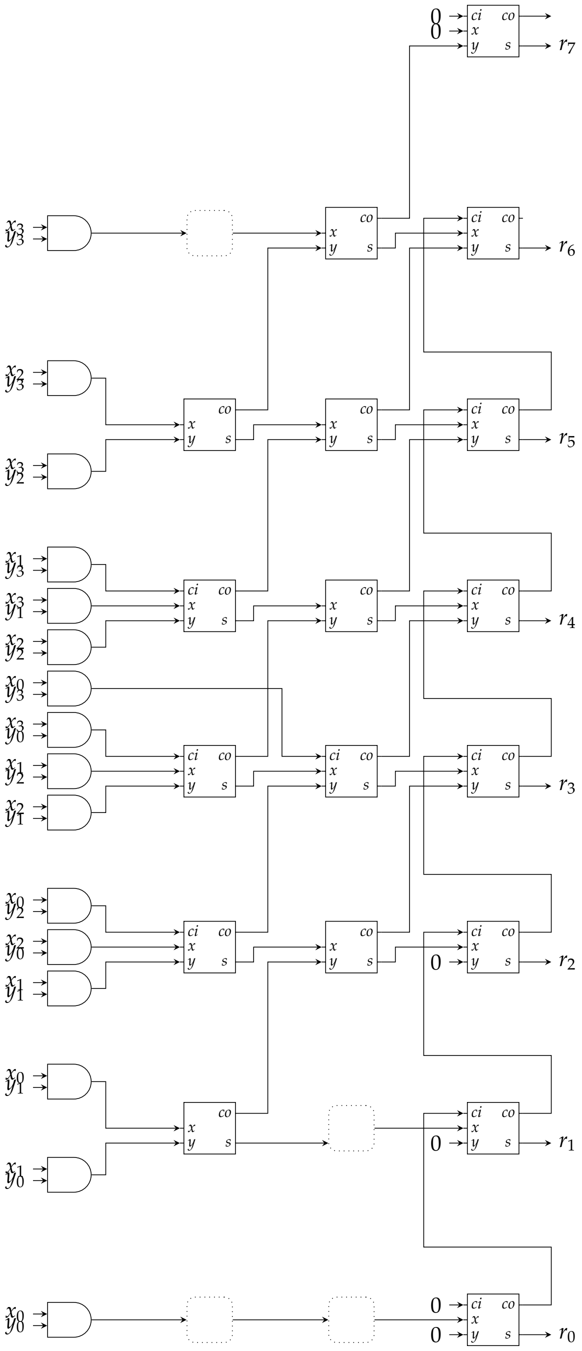


**Figure 14:** *A example, $(4 \times 4)$-bit Wallace-based tree multiplier design, described using a circuit diagram.*

attempt to survey such choices in a set of real micro-processors: the survey is by no means exhaustive, but will, none the less, offer better understanding of the associated constraints and designs used to address them.

**Example 0.41.** The ARM Cortex-M0 [4] processor supports multiplication via the `muls` instruction: it yields a truncated result, st. $r = x \cdot y$ is the least-significant 32 bits of the actual product given 32-bit $x$ and $y$. The instruction can be supported in two ways by a given implementation: the design can be a combinatorial, requiring 1 cycle, *or* a iterative, requiring 32 cycles. The Cortex-M0 is typically deployed in an embedded context, where area and power consumption are paramount. The latter, iterative multiplier design may therefore be attractive choice: assuming increased latency (or time) can be tolerated, it satisfies the goal of minimising area (or space) associated with this component of the associated ALU.

**Example 0.42.** The ARM7TDMI [1] processor houses a $(8 \cdot 32)$-bit combinatorial multiplier; it supports digit-serial multiplication (for the fixed case where $d = 8$) *with* early termination, as invoked by a range of instructions including `umull`. 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: quite a significant proportion will relate to address arithmetic, where $y$ is (relatively) small (e.g., as used to compute offsets from a fixed base address).

**Example 0.43.** Thomas and Balatoni [12] describe a $(12 \times 12)$-bit multiplier design, intended for use in the PDP-8 computer: the design is based on an iterative strategy that makes use of Booth recoding.

**Example 0.44.** The MIPS R4000 [6] processor takes a somewhat similar, somewhat *dis*similar approach to that described here: it houses a Booth-based multiplier using exactly the recoding strategy described, *but* within a $(64 \cdot 64)$-bit combinatorial rather than an iterative design.

Mirapuri et al. [6, Page 13] detail the design, which splits the multiplication into Booth recoding, multiplicand selection, partial product generation and product accumulation steps. A series of carry-save adders accumulate the partial products, which produces a result $r$ that is stored into two 64-bit registers called `hi` and `lo` (meaning more- and less-significant 64-bit halves).

**Example 0.45.** An iterative, bit-serial multiplier requires $n$ steps to compute the product; with no further optimisation, this constraint is inherent in the design. Although the data-path required is minimal, the need for iterative use of that data-path demands a control-path (i.e., an FSM) of some sort. When placed in a micro-processor, a resulting question is why we bother having a dedicated multiplier at all: why not just have an instruction that performs one step of multiplication, and let the program make iterative use of it?

The MIPS-X processor [3] provides an concrete example of this approach: using a slightly rephrased notation to match what has been described here, [3, Section 4.4.4] basically defines

$$\texttt{mstep}\ \mathsf{GPR}[x], \mathsf{GPR}[y], \mathsf{GPR}[r] \;\mapsto\; \left\{ \begin{array}{l} \textbf{if } \mathsf{GPR}[y]_{31} = 1 \textbf{ then} \\ \quad \mathsf{GPR}[r] \leftarrow \mathsf{GPR}[r] + \mathsf{GPR}[x] \\ \quad \mathsf{GPR}[y] \leftarrow \mathsf{GPR}[y] \lll 1 \\ \textbf{else} \\ \quad \mathsf{GPR}[r] \leftarrow \mathsf{GPR}[r] \\ \quad \mathsf{GPR}[y] \leftarrow \mathsf{GPR}[y] \lll 1 \\ \textbf{end} \end{array} \right.$$

i.e., a multiply-step instruction essentially matching lines #3 to #6 in Algorithm 9. As such, the idea is implement a loop that iterates over `mstep` as described in [3, Appendix IV]. The reason $y$ is left-shifted, is so that one can test $\mathsf{GPR}[y]_{31}$ rather than $\mathsf{GPR}[y]_i$; the former is updated by the shift, st. in each $i$-th iteration it does contain $\mathsf{GPR}[y]_i$ as required (give iteration is left-to-right, so starts with $i = 31$ and ends with $i = 0$).

There are advantages and disadvantages of either approach, i.e., use of a dedicated multiplier vs. an `mstep` instruction, with some examples including:

- The `mstep` instruction removes the need for an FSM to control the dedicated multiplier, essentially harnessing the existing processor as a control-path. As such, the overhead to support multiplication within the processor is further reduced.
- On one hand, the 1-step nature of `mstep` suggests single-cycle execution; in contrast, the $n$-step nature of the dedicated multiplier suggests multi-cycle execution. However, this is phrased in terms of processor cycles: it could be reasonable for a dedicated multiplier and processor to make use of *different* clock frequencies. Iff. the former is higher than the latter, $n$ multiplier cycles can be less than $n$ processor cycles and so execution of $n$ `mstep` instructions.

**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).

```
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
```

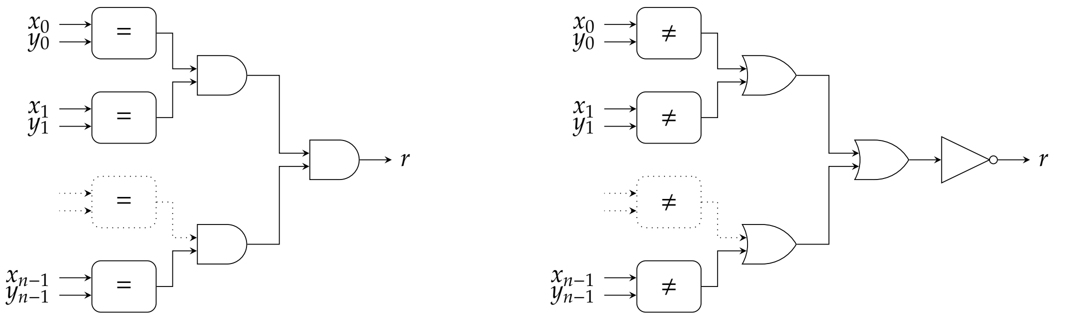
**Algorithm 19:** An algorithm for equality comparison between base-$b$ integers.

- By including the `mstep` instruction, the MIPS-X ISA exposes details of the implementation and so fixes how a given program should compute $r = y \cdot x$. If, in contrast, it had a `mul` instruction with obvious semantics, then any given implementation of the processor could opt for an iterative or combinatorial multiplier while maintaining compatibility.

- At least for simple processors, one instruction is executed at a time: for $n$-bit $x$ and $y$, this means the processor will be kept busy for $n$ cycles while executing $n$ `mstep` instruction. With a dedicated multiplier, however, one could at least *imagine* the processor doing something *else* in the $n$ cycles while the multiplier is kept busy.

# 6 Components for comparison

As with the 1-bit building blocks for addition (namely the half- and full-adder), we *already* covered designs for 1-bit equality and less than comparators in Chapter **??**; these require a handful of logic gates to implement. Again as with addition, the challenge is essentially how we extend these somehow. The idea of this Section is to tackle this step-by-step: first we consider designs for comparison of 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 a suite of comparison operations beyond just equality and less than.



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

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

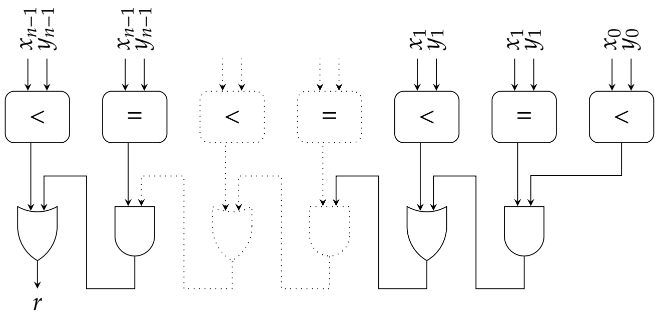


**Figure 16:** *An n-bit, unsigned less than comparison described using a circuit diagram.*

## 6.1 Unsigned comparison

### 6.1.1 Unsigned equality

**Example 0.46.** Consider two cases of comparison between unsigned $x$ and $y$ expressed in base-10

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

where, obviously, $x = y$ in the left-hand case, and $x \neq y$ in the right-hand case.

More formally, $x$ and $y$ are equal iff. each digit of $x$ is equal to the corresponding digit of $y$, so $x_i = y_i$ for $0 \leq i < n$. As such, in the left-hand case $x = y$ because $x_i = y_i$ for $0 \leq i < 3$; in the right-hand case $x \neq y$ because $x_i \neq y_i$ for $i = 0$. This fact is true in any base, and in base-2 we have a component that can perform the 1-bit comparison $x_i = y_i$: to cope with larger $x$ and $y$, we just combine instances of it together.

Read out loud, "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 a 1-bit equality comparator will compare $x_i$ and $y_i$, then we AND together the results. However, we need to take care wrt. then 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 (inequality) is simply an XOR gate, whereas the latter (equality) 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 $y$, then invert the result.

```
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 20:** An algorithm for less than comparison between base-$b$ integers.

Both designs are illustrated in Figure 15: 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 Unsigned less than

**Example 0.47.** Consider three cases of comparison between unsigned $x$ and $y$ expressed in base-10

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

where, obviously, $x < y$ in the left-hand case, $x > y$ in the middle case, and $x = y$ in the right-hand case.

Although the examples offer intuitively obvious results, determining *why*, in a formal sense, $x$ is less than $y$ (or not) is more involved than the case of equality. A somewhat algorithmic strategy is as follows: 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 a set of rules that say

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 $y_{i-1}$.

This can be used to explain the example:

- 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 $x = y$.

Figure 20 captures this more formally: we described, a loop iterates from the most- to least-significant digits of $x$ and $y$, and at each $i$-th step applies the rules above. That is, if $x_i < y_i$ then $x < y$ and if $x_i > y_i$ then $x \not< y$; if $x_i = y_i$ then the loop continues iterating, dealing with the next $(i-1)$-th step until it has processed all the digits. Notice that if the loop actually concludes, then we know that $x_i = y_i$ for all $i$ and so $x \not< y$.

Of course when $x$ and $y$ are written in base-2, our task is easier still because each $x_i, y_i \in \{0, 1\}$; this means we can use our existing 1-bit comparators. As such, translating the algorithm into a concrete design means reformulating more directly it wrt. said comparators. The idea is to recursively compute

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

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 $y$". Each step simply requires one of each comparator plus an extra AND and an extra OR gate; if we have $n$-bit $x$ and $y$, we have $n$ such steps as illustrated in Figure 16.

**Example 0.48.** Consider less than comparison for $n = 4$ bit $x$ and $y$, st. the unwound recursion

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

yields a result $t_3$. For $x = 5_{(10)} \mapsto 0101_{(2)}$ and $y = 7_{(10)} \mapsto 0111_{(2)}$ we can see that

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

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 $t_3 = \textbf{true}$, conclude that $x <_u y$ as expected.

## 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 tests whether each $x_i$ is the same as $y_i$ for $0 \leq i < n$. For signed $x$ and $y$ we do exactly the same thing: if $x_i$ differs from $y_i$, then the value $x$ represents will differ from the value $y$ represents *irrespective* of whether the representation is signed or unsigned.

However, signed less than comparison is not as simple. To produce the behaviour required, we use unsigned less than as a sub-component within a design for signed less than: for $x <_s y$ the rules

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

produce the result we want. The first two cases are obvious: if $x$ is positive and $y$ is negative it cannot ever be true that $x < y$, while if $x$ is negative and $y$ is positive it is always true that $x < y$. The other two cases need more explanation, but basically the idea is to consider the magnitudes of $x$ and $y$ only by computing then comparing $\mathrm{abs}(x)$ and $\mathrm{abs}(y)$, the absolute values of $x$ and $y$. Note 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 be less than a smaller negative $y$ (and vice versa); when considering their absolute values, the comparison is therefore reversed.

**Example 0.49.** 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.

**Example 0.50.** 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 $\mathrm{abs}(x) = 4 <_u 6 = \mathrm{abs}(y)$,

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

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

where $\text{chop}(x) = x_{n-2...0}$, meaning $\text{chop}(x)$ is $x$ with the MSB (which determines the sign of $x$) removed; this is valid because a small negative integer becomes a large positive integer (and vice versa) when the MSB is removed. Doing so is *much* simpler than computing $\text{abs}(x)$, because we just truncate or ignore the MSBs.

**Example 0.51.**

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 $\text{chop}(x) = 4 <_u 6 = \text{chop}(y)$,
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 $\text{chop}(x) = 6 \not<_u 4 = \text{chop}(y)$,
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 $\text{chop}(x) = 4 \not<_u 2 = \text{chop}(y)$, 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 $\text{chop}(x) = 2 <_u 4 = \text{chop}(y)$.

The question is, finally, how do we implement these rules as a design? As in the case of overflow detection, we use the fact that testing the sign of $x$ or $y$ is trivial. As a result, we can write

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

which can be realised by a multiplexer: producing the LHS just amounts to selecting an option from the RHS using $x_{n-1}$ and $y_{n-1}$, i.e., the sign of $x$ and $y$, as 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 can be derived using a set of 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.

# References

[1] *ARM7TDMI Technical Reference Manual*. Tech. rep. DDI-0210C. ARM Ltd., 2004. URL: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0210c/index.html (see p. 44).

[2] A.D. Booth. "A Signed Binary Multiplication Technique". In: *Quarterly Journal of Mechanics and Applied Mathematics* 4.2 (1951), pp. 236–240 (see p. 33).

[3] P. Chow. *MIPS-X Instruction Set And Programmer's Manual*. Tech. rep. CSL-86-289. Computer Systems Laboratory, Stanford University, 1998 (see p. 44).

[4] *Cortex-M0 Technical Reference Manual*. Tech. rep. DDI-0432C. ARM Ltd., 2009. URL: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0432c/index.html (see p. 44).

[5] L. Dadda. “Some Schemes for Parallel Multipliers”. In: *Alta Frequenza* 34 (1965), pp. 349–356 (see p. 40).

[6] J. Heinrich. *MIPS R4000 Microprocessor User's Manual*. 2nd. 1994 (see p. 44).

[7] W.G. Horner. “A new method of solving numerical equations of all orders, by continuous approximation”. In: *Philosophical Transactions* (1819), pp. 308–335 (see p. 29).

[8] A. Karatsuba and Y. Ofman. “Multiplication of Many-Digital Numbers by Automatic Computers”. In: *Physics-Doklady* 7 (1963), pp. 595–596 (see p. 28).

[9] J. von Neumann. *First Draft of a Report on the EDVAC*. Tech. rep. 1945 (see p. 3).

[10] B. Parhami. *Computer Arithmetic: Algorithms and Hardware Designs*. 1st ed. Oxford University Press, 2000 (see pp. 4, 22).

[11] A.S. Tanenbaum and T. Austin. *Structured Computer Organisation*. 6th ed. Prentice Hall, 2012 (see p. 5).

[12] P.A.V. Thomas and N. Balatoni. “A hardware multiplier/divider for the PDP 8S computer”. In: *Behavior Research Methods & Instrumentation* 3.2 (1971), pp. 89–91 (see p. 44).

[13] C.S. Wallace. “A Suggestion for Fast Multipliers”. In: *IEEE Transactions on Computers* 13.1 (1964), pp. 14–17 (see p. 40).