Chapter 8

Signed Addition

In this chapter we present circuits for adding and subtracting signed numbers that are represented by two's complement representation. Although the designs are obtained by very minor changes of a binary adder designs, the theory behind these changes requires some effort.

8.1 Representation of negative integers

We use binary representation to represent non-negative integers. We now address the issue of representing positive and negative integers. Following programming languages, we refer to non-negative integers as *unsigned numbers* and to negative and positive numbers as *signed numbers*.

There are three common methods for representing signed numbers: sign-magnitude, one's complements, and two's complement.

Definition 8.1 *The number represented in* sign-magnitude *representation by* $A[n-1:0] \in \{0,1\}^n$ *and* $S \in \{0,1\}$ *is*

$$(-1)^S \cdot \langle A[n-1:0] \rangle.$$

Definition 8.2 *The number represented in* one's complement *representation by* $A[n-1:0] \in \{0,1\}^n$ *is*

$$-(2^{n-1}-1) \cdot A[n-1] + \langle A[n-2:0] \rangle.$$

Definition 8.3 *The number represented in* two's complement *representation by* $A[n-1:0] \in \{0,1\}^n$ *is*

$$-2^{n-1} \cdot A[n-1] + \langle A[n-2:0] \rangle.$$

The most common method for representing signed numbers is two's complement representation. The main reason is that adding, subtracting, and multiplying signed numbers represented in two's complement representation is almost as easy as performing these computations on unsigned (binary) numbers.

8.2 Negation in two's complement representation

We denote the number represented in two's complement representation by $A[n-1:0]$ as follows:

$$[A[n-1:0]] \triangleq -2^{n-1} \cdot A[n-1] + \langle A[n-2:0] \rangle.$$

We denote the set of signed numbers that are representable in two's complement representation using n-bit binary strings by T_n.

Claim 8.1

$$T_n \triangleq \left\{-2^{n-1}, -2^{n-1}+1, \ldots, 2^{n-1}-1\right\}.$$

Question 8.1 *Prove Claim 8.1*

The following claim deals with negating a value represented in two's complement representation.

Claim 8.2

$$-[A[n-1:0]] = [\text{INV}(A[n-1:0])] + 1.$$

Proof: Note that $\text{INV}(A[i]) = 1 - A[i]$. Hence,

$$\begin{aligned}
[\text{INV}(A[n-1:0])] &= -2^{n-1} \cdot \text{INV}(A[n-1]) + \langle \text{INV}(A[n-2:0]) \rangle \\
&= -2^{n-1} \cdot (1 - A[n-1]) + \sum_{i=0}^{n-2} (1 - A[i]) \cdot 2^i \\
&= \underbrace{-2^{n-1} + \sum_{i=0}^{n-2} 2^i}_{=-1} + \underbrace{2^{n-1} \cdot A[n-1] - \sum_{i=0}^{n-2} A[i])}_{=-[A[n-1:0]]} \\
&= -1 - [A[n-1:0]].
\end{aligned}$$

□

In Figure 8.1 we depict a design for negating numbers based on Claim 8.2. The circuit is input $\vec{A}$ and is supposed to compute $-\left[\vec{A}\right]$. The bits in the string $\vec{A}$ are first inverted to obtain $\overline{A}[n-1:0]$. An increment circuit outputs $C[n] \cdot B[n-1:0]$ such that

$$\langle C[n] \cdot B[n-1:0] \rangle = \langle \overline{A}[n-1:0] \rangle + 1.$$

Such an increment circuit can be implemented simply by using a binary adder with one addend sting fixed to $0^{n-1} \cdot 1$.

We would like to claim that the circuit depicted in Fig. 8.1 is correct. Unfortunately, we do not have yet the tools to prove the correctness. Let us try and see the point in which we run into trouble.

Claim 8.2 implies that all we need to do to compute $-\left[\vec{A}\right]$ is invert the bits of $\vec{A}$ and increment. The problem is with the meaning of increment. The increment circuit computes:

$$\langle \overline{A}[n-1:0] \rangle + 1.$$

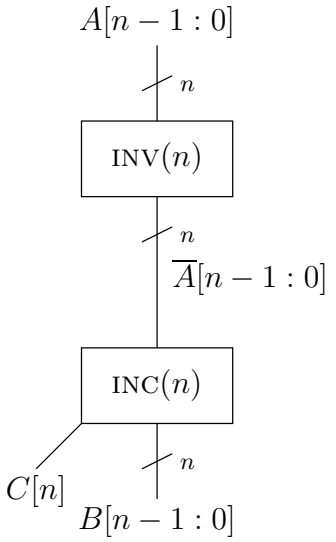

Figure 8.1: A circuit for negating a value represented in two's complement representation.

However, Claim 8.2 requires that we compute

$$\left[\overline{A}[n-1:0]\right]+1.$$

Now, let $C[n]\cdot B[n-1:0]$ denote the output of the incrementor. We know that

$$\langle C[n]\cdot B[n-1:0]\rangle = \langle \overline{A}[n-1:0]\rangle + 1.$$

Assume we are "lucky" and no overflow occurs, namely, $C[n]=0$. In this case,

$$\langle B[n-1:0]\rangle = \langle \overline{A}[n-1:0]\rangle + 1.$$

Why should this imply that

$$[B[n-1:0]] = \left[\overline{A}[n-1:0]\right]+1?$$

At this point we leave this issue unresolved. We prove a more general result in Theorem 8.7. (Question 8.12 deals with the correctness of the circuit for negating two's complement numbers.) Note, however, that the circuit errs with the input $A[n-1:0]=1\cdot 0^{n-1}$. The value represented by $\vec{A}$ equals -2^{n-1}. Inversion yields $\overrightarrow{\overline{A}}=0\cdot 1^{n-1}$. Increment yields $C[n]=0$ and $B[n-1:0]=1\cdot 0^{n-1}=A[n-1:0]$. This, of course, is not a counter-example to Claim 8.2. It is an example in which an increment with respect to $\langle\overline{A}[n-1:0]\rangle$ is not an increment with respect to $\left[\overline{A}[n-1:0]\right]$. This is exactly the point which concerned us. A more careful look at this case shows that every circuit must err with such an input. The reason is that $-\left[\vec{A}\right]\notin T_n$. Hence, the negated value cannot be represented using an n-bit string, and negation had to fail.

Question 8.2 *Propose circuits for negation with respect to sign-magnitude and one's-complement representation.*

8.3 Properties of two's complement representation

Alternative definition of two's complement representation. The following claim follows immediately from the definition of two's complement representation. The importance of this claim is that it provides an explanation for the definition of two's complement representation. In fact, one could define two's complement representation based on the following claim.

Claim 8.3 *For every* $A[n-1:0] \in \{0,1\}^n$

$$\mathrm{mod}(\langle \vec{A} \rangle, 2^n) = \mathrm{mod}(\left[\vec{A} \right], 2^n).$$

Question 8.3 *Prove Claim 8.3.*

Sign bit. The most significant bit $A[n-1]$ of a string $A[n-1:0]$ that represents a two's complement number is often called the *sign-bit* of $\vec{A}$. The following claim justifies this term.

Claim 8.4

$$[A[n-1:0]] < 0 \quad \Longleftrightarrow \quad A[n-1] = 1.$$

Question 8.4 *Prove Claim 8.4.*

Do not be misled by the term sign-bit. Two's complement representation is not sign-magnitude representation. In particular, the prefix $A[n-2:0]$ is not a binary representation of the magnitude of $[A[n-1:0]]$. Computing the absolute value of a negative signed number represented in two's complement representation involves inversion of the bits and an increment.

Sign extension. The following claim is often referred to as "sign-extension". It basically means that duplicating the most significant bit does not affect the value represented in two's complement representation. This is similar to padding zeros from the left in binary representation.

Claim 8.5 *If* $A[n] = A[n-1]$*, then*

$$[A[n:0]] = [A[n-1:0]] .$$

Proof:

$$\begin{aligned}
[A[n:0]] &= -2^n \cdot A[n] + \langle A[n-1:0] \rangle \\
&= -2^n \cdot A[n] + 2^{n-1} \cdot A[n-1] + \langle A[n-2:0] \rangle \\
&= -2^n \cdot A[n-1] + 2^{n-1} \cdot A[n-1] + \langle A[n-2:0] \rangle \\
&= -2^{n-1} \cdot A[n-1] + \langle A[n-2:0] \rangle \\
&= [A[n-1:0]] .
\end{aligned}$$

□

We can now apply arbitrarily long sign-extension, as summarized in the following Corollary.

Corollary 8.6

$$[A[n-1]^* \cdot A[n-1:0]] = [A[n-1:0]]\,.$$

Question 8.5 *Prove Corollary 8.6.*

8.4 Reduction: two's complement addition to binary addition

In Section 8.2 we tried to use a binary incrementor for incrementing a two's complement signed number. In this section we deal with a more general case, namely computing

$$\left[\vec{A}\right] + \left[\vec{B}\right] + C[0].$$

The following theorem deals with the following setting. Let

$$A[n-1:0], B[n-1:0], S[n-1:0] \in \{0,1\}^n$$
$$C[0], C[n] \in \{0,1\}$$

satisfy

$$\langle A[n-1:0]\rangle + \langle B[n-1:0]\rangle + C[0] = \langle C[n] \cdot S[n-1:0]\rangle. \tag{8.1}$$

Namely, $\vec{A}, \vec{B}$, and $C[0]$ are fed to a binary adder ADDER(n). The theorem addresses the following questions:

- When does the output $S[n-1:0]$ satisfy:

$$\left[\vec{S}\right] = [A[n-1:0]] + [B[n-1:0]] + C[0]? \tag{8.2}$$

- How can we know that Equation 8.2 holds?

Theorem 8.7 *Let $C[n-1]$ denote the carry-bit in position $[n-1]$ associated with the binary addition described in Equation 8.1 and let*

$$z \triangleq [A[n-1:0]] + [B[n-1:0]] + C[0].$$

Then,

$$C[n] - C[n-1] = 1 \quad \Longrightarrow \quad z < -2^{n-1} \tag{8.3}$$
$$C[n-1] - C[n] = 1 \quad \Longrightarrow \quad z > 2^{n-1} - 1 \tag{8.4}$$
$$z \in T_n \quad \Longleftrightarrow \quad C[n] = C[n-1] \tag{8.5}$$
$$z \in T_n \quad \Longrightarrow \quad z = [S[n-1:0]]\,. \tag{8.6}$$

Proof: Recall that the definition of the functionality of FA_{n-1} in a Ripple-Carry Adder $\text{RCA}(n)$ implies that

$$A[n-1] + B[n-1] + C[n-1] = 2C[n] + S[n-1].$$

Hence

$$A[n-1] + B[n-1] = 2C[n] - C[n-1] + S[n-1]. \tag{8.7}$$

We now expand z as follows:

$$\begin{aligned} z &= [A[n-1:0]] + [B[n-1:0]] + C[0] \\ &= -2^{n-1} \cdot (A[n-1] + B[n-1]) + \langle A[n-2:0] \rangle + \langle B[n-2:0] \rangle + C[0] \\ &= -2^{n-1} \cdot (2C[n] - C[n-1] + S[n-1]) + \langle C[n-1] \cdot S[n-2:0] \rangle. \end{aligned}$$

The last line is based on Equation 8.7 and on

$$\langle A[n-2:0] \rangle + \langle B[n-2:0] \rangle + C[0] = \langle C[n-1] \cdot S[n-2:0] \rangle.$$

This implies that

$$\begin{aligned} z &= -2^{n-1} \cdot (2C[n] - C[n-1] - C[n-1]) + [S[n-1] \cdot S[n-2:0]] \\ &= -2^{n} \cdot (C[n] - C[n-1]) + [S[n-1:0]]\,. \end{aligned}$$

We distinguish between three cases:

1. If $C[n] - C[n-1] = 1$, then

$$\begin{aligned} z &= -2^n + [S[n-1:0]] \\ &\leq -2^n + 2^{n-1} - 1 = -2^{n-1} - 1. \end{aligned}$$

   Hence Equation 8.3 follows.

2. If $C[n] - C[n-1] = -1$, then

$$\begin{aligned} z &= 2^n + [S[n-1:0]] \\ &\geq 2^n - 2^{n-1} = 2^{n-1}. \end{aligned}$$

   Hence Equation 8.4 follows.

3. If $C[n] = C[n-1]$, then $z = [S[n-1:0]]$, and obviously $z \in T_n$.

Equation 8.5 follows from the fact that if $C[n] \neq C[n-1]$, then either $C[n] - C[n-1] = 1$ or $C[n-1] - C[n] = 1$. In both these cases $z \notin T_n$. Equation 8.6 follows from the third case as well, and the theorem follows. □

8.4.1 Detecting overflow

Overflow occurs when the sum of signed numbers is not in T_n. Using the notation of Theorem 8.7, overflow is defined as follows.

Definition 8.4 *Let* $z \triangleq [A[n-1:0]] + [B[n-1:0]] + C[0]$. *The signal* OVF *is defined as follows:*

$$\text{OVF} \triangleq \begin{cases} 1 & \text{if } z \notin T_n \\ 0 & \text{otherwise.} \end{cases}$$

Note that overflow means that the sum is either too large or too small. Perhaps the term "out-of-range" is more appropriate than "overflow" (which suggests that the sum is too big). We choose to follow the common term rather than introduce a new term.

By Theorem 8.7, overflow occurs iff $C[n-1] \neq C[n]$. Namely,

$$\text{OVF} = \text{XOR}(C[n-1], C[n]).$$

Moreover, if overflow does not occur, then Equation 8.2 holds. Hence, we have a simple way to answer both questions raised before the statement of Theorem 8.7. The signal $C[n-1]$ may not be available if one uses a "black-box" binary-adder (e.g., a library component in which $C[n-1]$ is an internal signal). In this case we detect overflow based on the following claim.

Claim 8.8

$$\text{XOR}(C[n-1], C[n]) = \text{XOR}_4(A[n-1], B[n-1], S[n-1], C[n]).$$

Proof: Recall that

$$C[n-1] = \text{XOR}_3(A[n-1], B[n-1], S[n-1]).$$

□

Question 8.6 *Prove that*

$$\text{OVF} = \text{OR}(\text{AND}_3(A[n-1], B[n-1], \text{INV}(S[n-1])), \text{AND}_3(\text{INV}(A[n-1]), \text{INV}(B[n-1]), S[n-1])).$$

8.4.2 Determining the sign of the sum

How do we determine the sign of the sum z? Obviously, if $z \in T_n$, then Claim 8.4 implies that $S[n-1]$ indicates whether z is negative. However, if overflow occurs, this is not true.

Question 8.7 *Provide an example in which the sign of* z *is not signaled correctly by* $S[n-1]$.

We would like to be able to know whether z is negative regardless of whether overflow occurs. We define the NEG signal.

Definition 8.5 *The signal* NEG *is defined as follows:*

$$\text{NEG} \triangleq \begin{cases} 1 & \text{if } z < 0 \\ 0 & \text{if } z \geq 0. \end{cases}$$

A brute force method based on Theorem 8.7 for computing the NEG signal is as follows:

$$\text{NEG} = \begin{cases} S[n-1] & \text{if no overflow} \\ 1 & \text{if } C[n] - C[n-1] = 1 \\ 0 & \text{if } C[n-1] - C[n] = 1. \end{cases} \tag{8.8}$$

Although this computation obviously signals correctly whether the sum is negative, it requires some further work if we wish to obtain a small circuit for computing NEG that is not given $C[n-1]$ as input.

Instead pursuing this direction, we compute NEG using a more elegant method.

Claim 8.9

$$\text{NEG} = \text{XOR}_3(A[n-1], B[n-1], C[n]).$$

Proof: The proof is based on playing the following "mental game". We extend the computation to $n+1$ bits. We then show that overflow does not occur. This means that the sum bit in position n indicates correctly the sign of the sum z. We then express this sum bit using n-bit addition signals.
Let

$$\begin{aligned} \tilde{A}[n:0] &\triangleq A[n-1] \cdot A[n-1:0] \\ \tilde{B}[n:0] &\triangleq B[n-1] \cdot B[n-1:0] \\ \langle \tilde{C}[n+1] \cdot \tilde{S}[n:0] \rangle &\triangleq \langle \tilde{A}[n:0] \rangle + \langle \tilde{B}[n:0] \rangle + C[0]. \end{aligned}$$

Since sign-extension preserves value (see Claim 8.5), it follows that

$$z = \left[\tilde{A}[n:0]\right] + \left[\tilde{B}[n:0]\right] + C[0].$$

We claim that $z \in T_{n+1}$. This follows from

$$\begin{aligned} z &= [A[n-1:0]] + [B[n-1:0]] + C[0] \\ &\leq 2^{n-1} - 1 + 2^{n-1} - 1 + 1 \\ &\leq 2^n - 1. \end{aligned}$$

Similarly $z \geq 2^{-n}$. Hence $z \in T_{n+1}$, and therefore, by Theorem 8.7

$$\left[\tilde{S}[n:0]\right] = \left[\tilde{A}[n:0]\right] + \left[\tilde{B}[n:0]\right] + C[0].$$

We conclude that $z = \left[\tilde{S}[n:0]\right]$. It follows that $\text{NEG} = \tilde{S}[n]$. However,

$$\begin{aligned}\tilde{S}[n] &= \text{XOR}_3(\tilde{A}[n], \tilde{B}[n], \tilde{C}[n]) \\ &= \text{XOR}_3(A[n-1], B[n-1], C[n]),\end{aligned}$$

and the claim follows. □

Question 8.8 *Prove that* $\text{NEG} = \text{XOR}(\text{OVF}, S[n-1])$.

8.5 A two's-complement adder

In this section we define and implement a two's complement adder.

Definition 8.6 *A* two's-complement adder *with input length n is a combinational circuit specified as follows.*

Input: $A[n-1:0], B[n-1:0] \in \{0,1\}^n$, *and* $C[0] \in \{0,1\}$.

Output: $S[n-1:0] \in \{0,1\}^n$ *and* $\text{NEG}, \text{OVF} \in \{0,1\}$.

Functionality: *Define z as follows:*

$$z \triangleq [A[n-1:0]] + [B[n-1:0]] + C[0].$$

The functionality is defined as follows:

$$\begin{aligned} z \in T_n &\implies [S[n-1:0]] = z \\ z \in T_n &\iff \text{OVF} = 0 \\ z < 0 &\iff \text{NEG} = 1. \end{aligned}$$

Note that no carry-out $C[n]$ is output. We denote a two's-complement adder by $\text{S-ADDER}(n)$. The implementation of an $\text{S-ADDER}(n)$ is depicted in Figure 8.2 and is as follows:

1. The outputs $C[n]$ and $S[n-1:0]$ are computed by a binary adder $\text{ADDER}(n)$ that is fed by $A[n-1:0], B[n-1:0]$, and $C[0]$.
2. The output OVF is simply $\text{XOR}(C[n-1], C[n])$ if $C[n-1]$ is available. Otherwise, we apply Claim 8.8, namely, $\text{OVF} = \text{XOR}_4(A[n-1], B[n-1], S[n-1], C[n])$.
3. The output NEG is compute according to Claim 8.9. Namely, $\text{NEG} = \text{XOR}_3(A[n-1], B[n-1], C[n])$.

Note that, except for the circuitry that computes the flags OVF and NEG, a two's complement adder is identical to a binary adder. Hence, in an arithmetic logic unit (ALU), one may use the same circuit for signed addition and unsigned addition.

Question 8.9 *Prove the correctness of the implementation of* $\text{S-ADDER}(n)$ *depicted in Figure 8.2.*

Question 8.10 *Is the design depicted in Figure 8.3 a correct* $\text{S-ADDER}(2n)$*?*

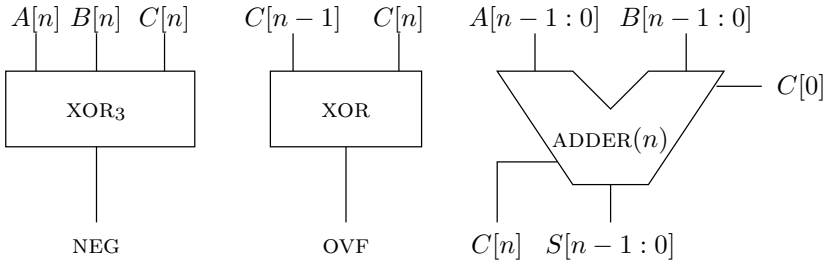

Figure 8.2: A two's complement adder S-ADDER(n)

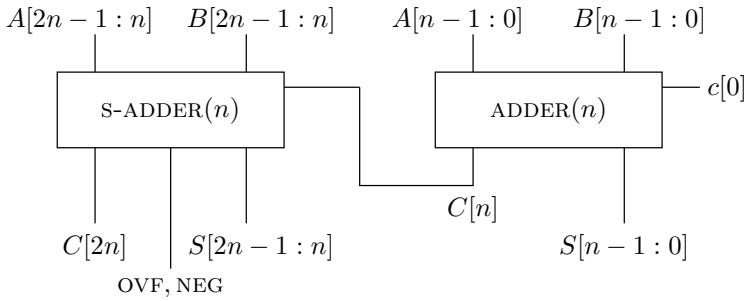

Figure 8.3: Concatenating an S-ADDER(n) with an ADDER(n).

8.6 A two's complement adder/subtracter

In this section we define and implement a two's complement adder/subtracter. A two's complement adder/subtracter is used in ALUs to implement addition and subtraction of signed numbers.

Definition 8.7 *A* two's-complement adder/subtracter *with input length n is a combinational circuit specified as follows.*

Input: $A[n-1:0], B[n-1:0] \in \{0,1\}^n$, *and* $\text{sub} \in \{0,1\}$.

Output: $S[n-1:0] \in \{0,1\}^n$ *and* NEG, OVF $\in \{0,1\}$.

Functionality: *Define z as follows:*

$$z \triangleq [A[n-1:0]] + (-1)^{\text{sub}} \cdot [B[n-1:0]] .$$

The functionality is defined as follows:

$$\begin{aligned} z \in T_n \quad &\Longrightarrow \quad [S[n-1:0]] = z \\ z \in T_n \quad &\Longleftrightarrow \quad \text{OVF} = 0 \\ z < 0 \quad &\Longleftrightarrow \quad \text{NEG} = 1. \end{aligned}$$

We denote a two's-complement adder/subtracter by ADD-SUB(n). Note that the input *sub* indicates if the operation is addition or subtraction. Note also that no carry-in bit $C[0]$ is input and no carry-out $C[n]$ is output.

An implementation of a two's-complement adder/subtracter ADD-SUB(n) is depicted in Figure 8.4. The implementation is based on a two's complement adder S-ADDER(n) and Claim 8.2.

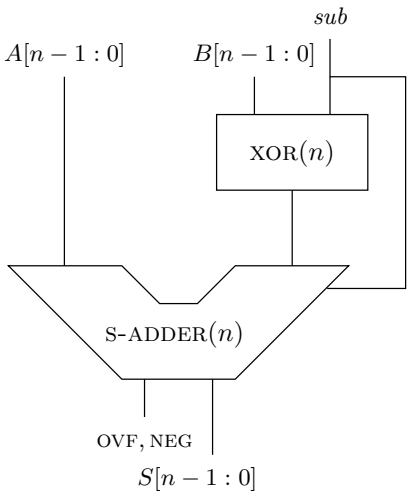

Figure 8.4: A two's-complement adder/subtracter ADD-SUB(n).

Claim 8.10 *The implementation of* ADD-SUB(n) *depicted in Figure 8.4 is correct.*

Question 8.11 *Prove Claim 8.10.*

Question 8.12 (back to the negation circuit) *Consider the negation circuit depicted in Figure 8.1.*

1. *When is the circuit correct?*
2. *Suppose we wish to add a signal that indicates whether the circuit satisfies* $\left[\vec{B}\right] = -\left[\vec{A}\right]$. *How should we compute this signal?*

Question 8.13 (wrong implementation of ADD-SUB(n)**)** *Find a input for which the circuit depicted in Figure 8.5 errs. Can you list all the inputs for which this circuit outputs a wrong output?*

8.7 Additional questions

Question 8.14 (OVF **and** NEG **flags in high level programming)** *High level programming languages such as C and Java do not enable one to see the value of the* OVF *and* NEG *signals (although these signals are computed by adders in all microprocessors).*

1. *Write a short program that deduces the values of these flags. Count how many instructions are needed to recover these lost flags.*

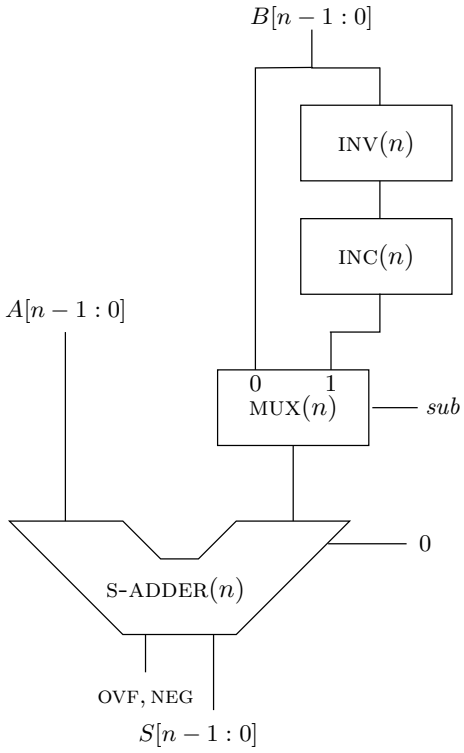

Figure 8.5: A wrong implementation of ADD-SUB(n).

2. *Short segments in a low level language (Assembly) can be integrated in C programs. Do you know how to see the values of the* OVF *and* NEG *flags using a low level language?*

Question 8.15 (bi-directional cyclic shifting) *The goal in this question is to design a bi-directional barrel-shifter.*

Definition 8.8 *A bi-directional barrel-shifter* BI-BARREL-SHIFTER(n) *is a combinational circuit defined as follows:*

Input: $x[n-1:0]$, $dir \in \{0,1\}$, *and* $sa[k-1:0]$ *where* $k = \lceil \log_2 n \rceil$.

Output: $y[n-1:0]$.

Functionality: *If* $dir = 0$ *then* $\vec{y}$ *is a cyclic left shift of* $\vec{x}$ *by* $\langle \vec{sa} \rangle$ *positions. Formally,*

$$\forall j \in [n-1:0]: \quad y[j] = x[\mathrm{mod}(j + \langle \vec{sa} \rangle, n)].$$

If $dir = 1$ *then* $\vec{y}$ *is a cyclic right shift of* $\vec{x}$ *by* $\langle \vec{sa} \rangle$ *positions. Formally,*

$$\forall j \in [n-1:0]: \quad y[j] = x[\mathrm{mod}(j - \langle \vec{sa} \rangle, n)].$$

1. *Suggest a reduction of right cyclic shifting to left cyclic shifting. (Hint: shift by* x *to the right is equivalent to shift by* $2^k - x$ *to the left.)*

2. *If your reduction includes an increment, suggest a method that avoids the logarithmic delay associated with incrementing.*

Question 8.16 (Comparison) *Design a combinational circuit* COMPARE(n) *defined as follows.*

Inputs: $A[n-1:0, B[n-1:0] \in \{0,1\}^n$.

Output: $LT, EQ, GT \in \{0,1\}$.

Functionality:

$$\begin{aligned} \left[\vec{A}\right] > \left[\vec{B}\right] \quad &\Longleftrightarrow \quad GT = 1 \\ \left[\vec{A}\right] = \left[\vec{B}\right] \quad &\Longleftrightarrow \quad EQ = 1 \\ \left[\vec{A}\right] < \left[\vec{B}\right] \quad &\Longleftrightarrow \quad LT = 1. \end{aligned}$$

1. *Design a comparator based on a two's complement subtracter and a zero-tester.*

2. *Design a comparator from scratch based on a* PPC–OP(n) *circuit.*

Question 8.17 (one's complement adder/subtracter) *Design an adder/subtracter with respect to one's complement representation.*

Question 8.18 (sign-magnitude adder/subtracter) *Design an adder/subtracter with respect to sign-magnitude representation.*

8.8 Summary

In this chapter we presented circuits for adding and subtracting two's complement signed numbers. We started by describing three ways for representing negative integers: sign-magnitude, one's-complement, and two's complement. We then focused on two's complement representation.

The first task we consider is negating. We proved that negating in two's complement representation requires inverting the bits and incrementing. The claim that describes negation was insufficient to argue about the correctness of a circuit for negating a two's complement signed number. We also noticed that negating the represented value is harder in two's complement representation than in the other two representations.

In Section 8.3 we discussed a few properties of two's complement representation: (i) We showed that the values represented by the same n-bit string in binary representation and in two's complement representation are congruent module 2^n. (ii) We showed that the most-significant bit indicates whether the represented value is negative. (iii) Finally, we discussed sign-extension. Sign-extension enables us to increase the number of bits used to represent a two's complement number while preserving the represented value.

The main result of this chapter is presented in Section 8.4. We reduce the task of two's complement addition to binary addition. Theorem 8.7 also provides a rule that enables us to tell when this reduction fails. The rest of this section deals with: (i) the detection of overflow - this is the case that the sum is out of range; and (ii) determining the sign of the sum even if an overflow occurs.

In Section 8.5 we present an implementation of a circuit that adds two's complement numbers. Finally, in Section 8.6 we present an implementation of a circuit that can add and subtract two's complement numbers.