Agda, Full Adders, and Flags

Dennis Sprokholt

July 2022

1 Introduction

Machine instruction semantics are often difficult to grasp. Especially the effect of an instruction on *CPU flags* is mysterious, which is often merely described in prose. For me, rigorous formal definitions often dispel this veil of mystery. To achieve that, we define instruction semantics *from the ground up* in a *principled* way. This is a small step-by-step guide on doing so in Agda. In particular, we look at *ripple carry* circuits and the semantics of the *carry* and *overflow* flags.

2 Bits

We first define *bits*.

```
data Bit : Set where
  O : Bit
  I  : Bit
```

Then, we define several common bitwise operators:

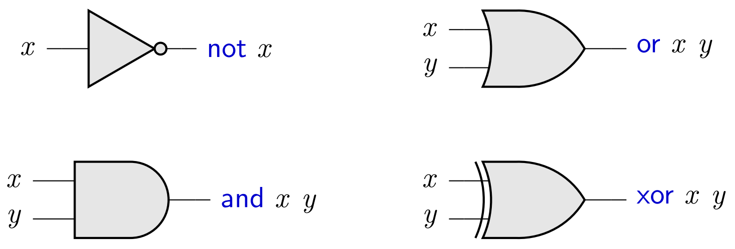
```
not : Bit → Bit
not O = I
not I  = O
```

```
or : Bit → Bit → Bit
or O y = y
or I  y = I
```

```
and : Bit → Bit → Bit
and O y = O
and I  y = y
```

```
xor : Bit → Bit → Bit
xor O y = y
xor I  y = not y
```

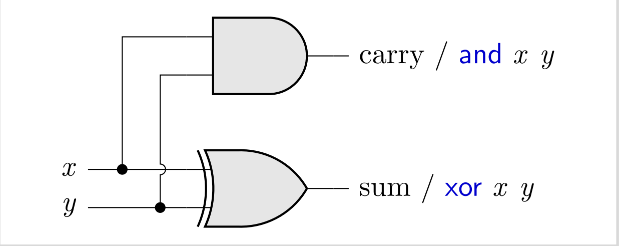
Now, let's look at their corresponding logic gates:



3 Adder Circuits

Now we will gradually build more complex circuits. Consider the *half adder* in Figure 1.

Figure 1: Half Adder



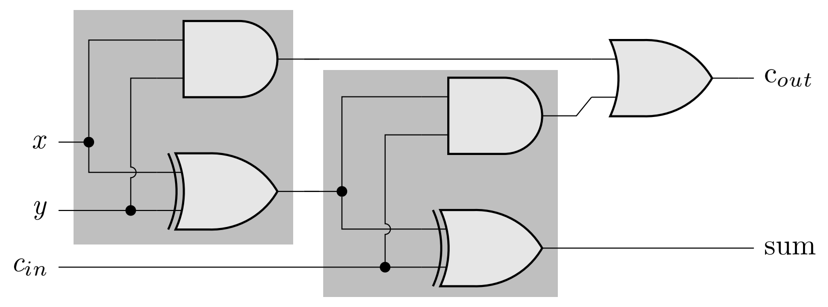
The carry bit represents the "leftover bit" in a higher position. For instance, in the decimal system, $8 + 7 = 5$ carrying 1; It represents 15, but that does not fit in a single digit.

We define this circuit in Agda as:

```
half-adder : Bit → Bit → Bit × Bit
half-adder x y = (and x y , xor x y)
```

Using two half adders, we can construct a *full adder*, as shown in Figure 2.

Figure 2: Full Adder



Which we similarly compose of half-adders in our Agda definition:

```
full-adder : Bit → Bit → Bit → Bit × Bit
full-adder x y cᵢₙ =
  let (c₁ , s₁) = half-adder x y
      (c₂ , s₂) = half-adder s₁ cᵢₙ
  in (or c₁ c₂ , s₂)
```

Table 1 contains the full corresponding truth table; It should help get an intuition for the carry bit (c_{out}).

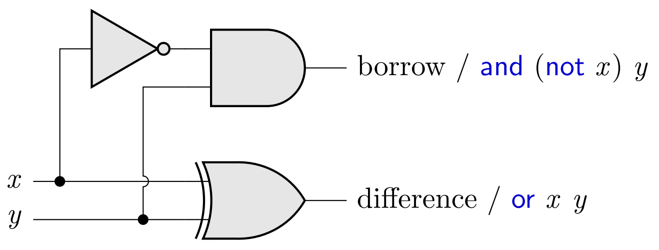
Table 1: Full Adder Truth Table

x	y	c_{in}	c_{out}	sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

4 Subtractor Circuits

Similarly to adders, we can define subtractors. Consider the half-subtractor in Figure 3.

Figure 3: Half Subtractor



A subtractor does *not* have a carry bit. It has a *borrow* bit instead. The half-subtractor's truth table is given in Table 2.

Table 2: Half Subtractor Truth Table

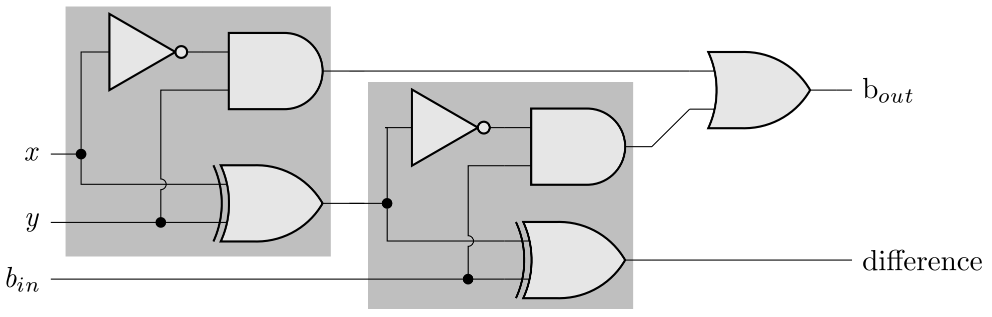
x	y	borrow	difference
0	0	0	0
0	1	1	1
1	0	0	1
1	1	0	0

Clearly, $0-0=0$, $1-0=1$, and $1-1=0$. The notable case is $0-1$, whose difference *doesn't* fit in a single bit. Intuitively, we borrow a bit from the next position – which represents 2 – and subtract from it. Its result is thus $2-1=1$. Of course, we need to remember that we borrowed a bit. We define the *half subtractor* in Agda as:

```
half-subtractor : Bit → Bit → Bit × Bit
half-subtractor x y = (and (not x) y , xor x y)
```

Similarly to the full adder, we compose two half-subtractors to create a full-subtractor, which we give in Figure 4.

Figure 4: Full Subtractor



Which in Agda becomes:

```
full-subtractor : Bit → Bit → Bit → Bit × Bit
full-subtractor x y b_in =
  let (b₁ , d₁) = half-subtractor x y
      (b₂ , d₂) = half-subtractor d₁ b_in
  in (or b₁ b₂ , d₂)
```

Table 3 is its truth table.

Table 3: Full Subtractor Truth Table

x	y	b_{in}	b_{out}	difference
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

The value of b_{in} represents the "remembered" borrow bit by subtractions in the previous (lower) position. Intuitively, the full subtractor computes $x - y - b_{in}$.

5 Higher-Order Circuits

One could observe that the composition of half adders into full adders (Figure 2) is *identical* to the composition of half subtractors into full subtractors (Figure 4). Hence, we can generalize over that structure, resulting in "*higher-order circuits*", if you will.

```
-- | Converts half adders/subtractors to their full counterparts
lift :   ( Bit → Bit → Bit × Bit )
         --------------------------------
       → ( Bit → Bit → Bit → Bit × Bit )
lift f x y c_in =
  let (c₁ , s₁) = f x y
      (c₂ , s₂) = f s₁ c_in
  in (or c₁ c₂ , s₂)

full-adder₂      = lift half-adder

full-subtractor₂ = lift half-subtractor
```

Of course, these new versions are *definitionally equal* to our previous functions:

```
_ : full-adder ≡ full-adder₂
_ = refl

_ : full-subtractor ≡ full-subtractor₂
_ = refl
```

6 Bitvectors

With addition and subtraction on *bits*, we can extend these operations to *bitvectors*. First, we define bitvectors (of length n) as:

```
Bv : ℕ → Set
Bv n = Vec Bit n
```

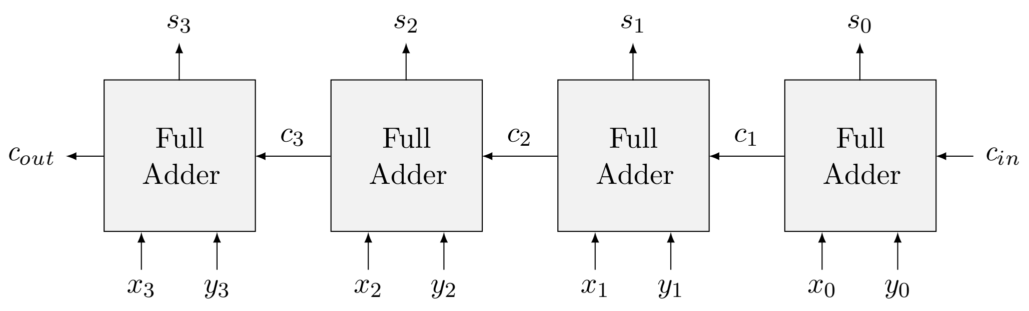
So, now we can – for instance – construct the bitvector (of length 4) representing[A] 5_{10}, which is 0101_2 in binary. In Agda that becomes:

```
five : Bv 4
five = O :: I :: O :: I :: []
```

7 Ripple Carry Circuits

Now we compose the bitvector adders from the full-adders for bits. Intuitively, the carry ”ripples upwards”, to the higher-positioned bits. Figure 5 illustrates this.

Figure 5: Ripple Carry Adder (of length 4)



The “ripple borrow subtractor” has the same structure. Hence, in the Agda implementation we can generalize over that “ripple”-structure.

```
-- | Lifts a full-adder/subtractor to its bitvector variant.
ripple :   ( Bit → Bit → Bit → Bit × Bit )
           -------------------------------
        → ( Bv n → Bv n → Bit → Bit × Bv n )
ripple f [] [] cᵢₙ = (cᵢₙ , [])
ripple f (x :: xs) (y :: ys) cᵢₙ =
  let (c₁ , xs+ys) = ripple f xs ys cᵢₙ
      (c₂ , x+y)   = f x y c₁
  in (c₂ , x+y :: xs+ys)

ripple-adder ripple-subtractor : Bv n → Bv n → Bit → Bit × Bv n
ripple-adder = ripple full-adder₂
ripple-subtractor = ripple full-subtractor₂
```

[A]Notation: x_{10} is in the decimal system whereas x_2 is in binary.

8 Specification Testing

We can now *test* our specification:

```
-- # Adder

-- | 6 + 3 + 0 = 9
_ : ripple-adder (O :: I :: I :: O :: []) (O :: O :: I :: I :: []) O ≡ (O , (I :: O :: O :: I :: []))
_ = refl

-- | 9 + 11 + 1 = 21 = 16 + 5
_ : ripple-adder (I :: O :: O :: I :: []) (I :: O :: I :: I :: []) I ≡ (I , (O :: I :: O :: I :: []))
_ = refl

-- # Subtractor

-- | 6 - 3 - 0 = 3
_ : ripple-subtractor
  (O :: I :: I :: O :: []) (O :: O :: I :: I :: []) O ≡ (O , (O :: O :: I :: I :: []))
_ = refl

-- | 9 - 11 - 1 = -3 = (-16) + 13
_ : ripple-subtractor (I :: O :: O :: I :: []) (I :: O :: I :: I :: []) I ≡ (I , I :: I :: O :: I :: [])
_ = refl
```

The subtraction in the final test case resolves to a negative number in the decimal system (-3_{10}). Yet, that has no representation as bitvector[B]. Note that its decimal representation is only given for illustrative purposes; It is *irrelevant* for the semantics of bitvector subtraction, as the circuits precisely capture it.

Working with a proof assistant does *not* guarantee correctness; After all, our specifications *could* be erroneous. Checking the correctness of a specification is a matter of *validation* w.r.t. intended characteristics. Hence, we provide *tests*.

9 Flags

Now that we have correct specifications for bitvector addition and subtraction, we can include specifications for their effect on CPU flags. We discuss the semantics – in both x86 and Armv8 – for:

- the *carry flag* (`CF`) and
- the *overflow flag* (`OF`).

[B]We assume bitvectors are not inherently signed or unsigned.

9.1 Carry Flag

The carry flag for *addition* is identical between the two architectures.

```
add-CF : Bv n → Bv n → Bit
add-CF x y = proj₁ (ripple-adder x y O)
```

However, for *subtraction* their carry flags are *inverted*. The x86 manual[1] states:

> *Carry flag* – Set if an arithmetic operation generates a carry or a borrow out of the most significant bit of the result; cleared otherwise. ...

Hence, after executing the `SUB` instruction, the carry flag represents the borrow bit:

```
x86-sub-CF : Bv n → Bv n → Bit
x86-sub-CF x y = proj₁ (ripple-subtractor x y O)
```

The Armv8 manual[2, C6.2.318-320] states:

```
operand2 = NOT(operand2);
(result, nzcv) = AddWithCarry(operand1, operand2, '1');

PSTATE.<N,Z,C,V> = nzcv;
```

Which means that it defines `SUBS` with “AddWithCarry”. Hence, it also affects the flags as such. Our specification follows the Arm manual:

```
Armv8-sub-CF : Bv n → Bv n → Bit
Armv8-sub-CF x y = proj₁ (ripple-adder x (map not y) I)
```

We can show that the Armv8 `CF` is the inverse (not) of x86's `CF`:

```
CF-inv : ∀ (x y : Bv n) → x86-sub-CF x y ≡ not (Armv8-sub-CF x y)
-- proof omitted
```

9.2 Overflow Flag

The *overflow flag* is similar to the *carry flag*. Whereas the carry flag signifies that the result of *unsigned* arithmetic does not fit in a register, the overflow flag signifies that the *signed* result does not fit. Integer operations in machines are not inherently signed or unsigned. x86 and Armv8 set flags for both cases; It is up to the programmer to read the appropriate flag for their use case.

The x86 manual[1] states:

> *Overflow flag* — Set if the integer result is too large a positive number or too small a negative number (excluding the sign-bit) to fit in the destination operand; cleared otherwise. This flag indicates an overflow condition for signed-integer (two's complement) arithmetic.

The Arm manual[2] similarly states:

> Overflow Condition flag. Set to:
>
> - 1 if the instruction results in an overflow condition, for example a signed overflow that is the result of an addition.
> - 0 otherwise.

For addition, both architectures behave similarly. We compute the overflow flag by xoring the incoming and outgoing *carry* bit of the most-significant adder[C]:

```
add-OF : Bv (suc n) → Bv (suc n) → Bit
add-OF (x :: xs) (y :: ys) =
  let c_in  = proj₁ (ripple-adder xs ys O)
      c_out = proj₁ (full-adder x y c_in)
  in xor c_in c_out
```

This satisfies both prose specifications, which we can *test* as follows:

```
-- | (-3) + (-6) = -9 = (-8) + (-1)  =>  overflow
_ : add-OF (I :: I :: O :: I :: []) (I :: O :: I :: O :: []) ≡ I
_ = refl

-- | (-5) + 1 = -4  =>  no overflow
_ : add-OF (I :: O :: I :: I :: []) (O :: O :: O :: I :: []) ≡ O
_ = refl
```

However, for subtraction, their *specifications* are different. In x86, subtraction's `OF` follows from the definitions of subtraction, which we modelled with the ripple subtractor:

```
x86-sub-OF : Bv (suc n) → Bv (suc n) → Bit
x86-sub-OF (x :: xs) (y :: ys) =
  let b_in  = proj₁ (ripple-subtractor xs ys O)
      b_out = proj₁ (full-subtractor x y b_in)
  in xor b_in b_out
```

[C] We use Bv (suc n), because 0-bit bitvectors have no MSB, and thus no c_{in} and c_{out}.

Two test cases that demonstrate it makes sense:

```
-- | (-3) - 6  =  -9  =  (-8) - 1  =>  overflow
_ : x86-sub-OF (I :: I :: O :: I :: []) (O :: I :: I :: O :: []) ≡ I
_ = refl

-- | (-1) - 7 = -8  =>  no overflow
_ : x86-sub-OF (I :: I :: I :: I :: []) (O :: I :: I :: I :: []) ≡ O
_ = refl
```

In Arm – like before – subtraction is defined in terms of addition. The definition of `OF`[D] for subtraction also builds on addition, which we modelled with the ripple *adder*:

```
Armv8-sub-OF : Bv (suc n) → Bv (suc n) → Bit
Armv8-sub-OF (x :: xs) (y :: ys) =
  let c_in = proj₁ (ripple-adder xs (map not ys) I)
      c_out = proj₁ (full-adder x (not y) c_in)
  in xor c_in c_out
```

Interestingly, though both are computed differently, the value of `OF` is *identical* between x86 and Armv8 (for the same operands):

```
OF-eq : ∀ (x y : Bv (suc n)) → x86-sub-OF x y ≡ Armv8-sub-OF x y
-- proof omitted
```

10 Conclusion

We formalized the semantics of two bitvector operations in Agda (being addition and subtraction). We did this in a *principled* way, by analyzing the circuits and modeling our semantics accordingly. We also looked at – and modelled – the *carry* and *overflow* flags for the x86 and Armv8 architectures. Both architectures set the flags similarly, but differ on the *carry* flag for *subtraction*. Additionally, we provided *some* test cases to *validate* our models.

More generally, this document aims to serve as a brief guide on accurately specifying ISA semantics. As a final note, proof assistants – particularly Agda – are an invaluable tool for *exploring* semantic models, and often help to uncover subtle properties.

[D]`OF` is actually called `V` in Arm

References

[1] Intel Corporation, *Intel 64 and IA-32 Architectures Software Developer's Manual - Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4*, April 2022.

[2] Arm Limited, *Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*, 2021.