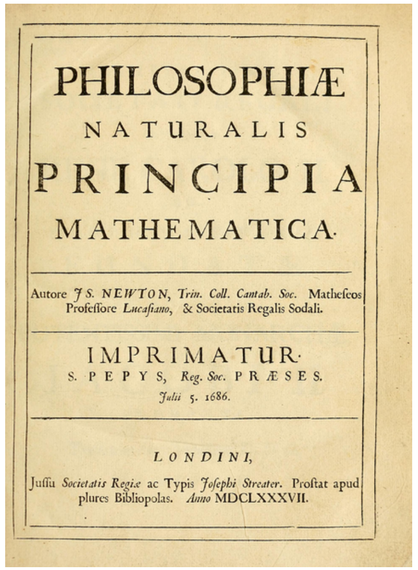
Principia Mathematica

The foundations of arithmetic in C++

Lisa Lippincott

CppCon, September 2022



PHILOSOPHIÆ
NATURALIS
PRINCIPIA
MATHEMATICA.

Autore *J. S. NEWTON, Trin. Coll. Cantab. Soc.* Matheseos Professore *Lucasiano*, & Societatis Regalis Sodali.

IMPRIMATUR.
S. PEPYS, *Reg. Soc.* PRÆSES.
Julii 5. 1686.

LONDINI,
Jussu *Societatis Regiæ* ac Typis *Josephi Streater*. Prostat apud plures Bibliopolas. *Anno* MDCLXXXVII.

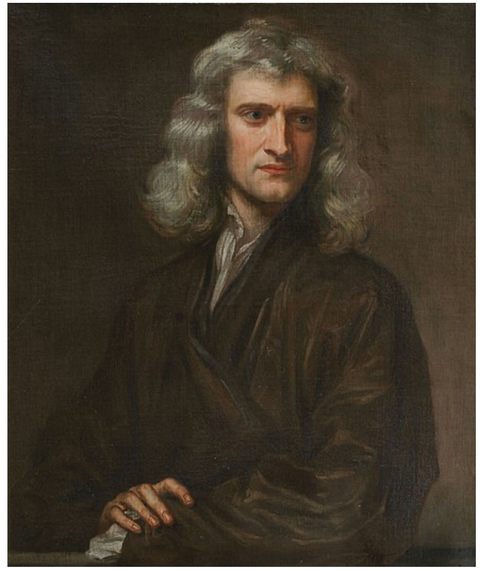
Philosophiæ Naturalis Principia Mathematica

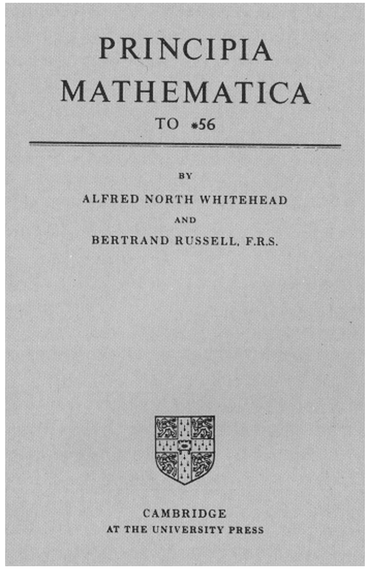
(Mathematical Principles of Natural Philosophy)

1687

Isaac Newton

1643-1727 (Gregorian)





Principia Mathematica

(Principles of Mathematics)

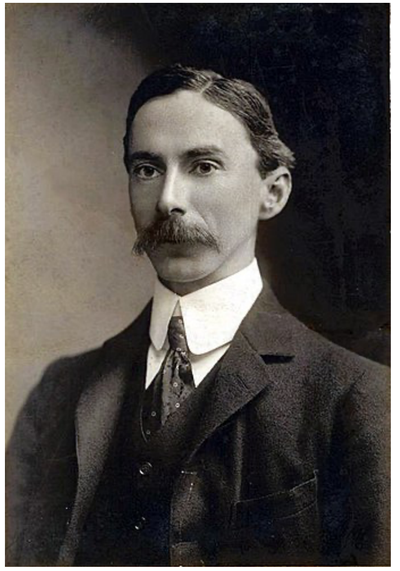
1910, 1912, 1913

Alfred North Whitehead

1861-1947

Bertrand Russell

1872-1970



***110·632.** $\vdash : \mu \in \mathrm{NC} . \supset . \mu +_c 1 = \hat{\xi} \{(\exists y) . y \in \xi . \xi - \iota`y \in \mathrm{sm}``\mu\}$

Dem.

$\vdash . *110·631 . *51·211·22 . \supset$

$\vdash : \mathrm{Hp} . \supset . \mu +_c 1 = \hat{\xi} \{(\exists \gamma, y) . \gamma \in \mathrm{sm}``\mu . y \in \xi . \gamma = \xi - \iota`y\}$

[*13·195] $= \hat{\xi} \{(\exists y) . y \in \xi . \xi - \iota`y \in \mathrm{sm}``\mu\} : \supset \vdash . \mathrm{Prop}$

***110·64.** $\vdash . 0 +_c 0 = 0$ [*110·62]

***110·641.** $\vdash . 1 +_c 0 = 0 +_c 1 = 1$ [*110·51·61 . *101·2]

***110·642.** $\vdash . 2 +_c 0 = 0 +_c 2 = 2$ [*110·51·61 . *101·31]

***110·643.** $\vdash . 1 +_c 1 = 2$

Dem.

$\vdash . *110·632 . *101·21·28 . \supset$

$\vdash . 1 +_c 1 = \hat{\xi} \{(\exists y) . y \in \xi . \xi - \iota`y \in 1\}$

[*54·3] $= 2 . \supset \vdash . \mathrm{Prop}$

The above proposition is occasionally useful. It is used at least three times, in *113·66 and *120·123·472.

*110·7·71 are required for proving *110·72, and *110·72 is used in *117·3, which is a fundamental proposition in the theory of greater and less.

***110·7.** $\vdash : \beta \subset \alpha . \supset . (\exists \mu) . \mu \in \mathrm{NC} . \mathrm{Nc}`\alpha = \mathrm{Nc}`\beta +_c \mu$

Dem.

$\vdash . *24·411·21 . \supset \vdash : \mathrm{Hp} . \supset . \alpha = \beta \cup (\alpha - \beta) . \beta \cap (\alpha - \beta) = \Lambda .$

[*110·32] $\supset . \mathrm{Nc}`\alpha = \mathrm{Nc}`\beta +_c \mathrm{Nc}`(\alpha - \beta) : \supset \vdash . \mathrm{Prop}$

***110·71.** $\vdash : (\exists \mu) . \mathrm{Nc}`\alpha = \mathrm{Nc}`\beta +_c \mu . \supset . (\exists \delta) . \delta \, \mathrm{sm} \, \beta . \delta \subset \alpha$

Dem.

$\vdash . *100·3 . *110·4 . \supset$

$\vdash : \mathrm{Nc}`\alpha = \mathrm{Nc}`\beta +_c \mu . \supset . \mu \in \mathrm{NC} - \iota`\Lambda$ (1)

$\vdash . *110·3 . \supset \vdash : \mathrm{Nc}`\alpha = \mathrm{Nc}`\beta +_c \mathrm{Nc}`\gamma . \equiv . \mathrm{Nc}`\alpha = \mathrm{Nc}`(\beta + \gamma) .$

[*100·3·31] $\supset . \alpha \, \mathrm{sm} \, (\beta + \gamma) .$

[*73·1] $\supset . (\exists R) . R \in 1 \to 1 . \mathrm{D}`R = \alpha . \mathrm{C}`R = \downarrow \Lambda_\gamma ``\iota ``\beta \cup \Lambda_\beta \downarrow ``\iota ``\gamma .$

[*37·15] $\supset . (\exists R) . R \in 1 \to 1 . \downarrow \Lambda_\gamma ``\iota ``\beta \subset \mathrm{C}`R . R`` \downarrow \Lambda_\gamma ``\iota ``\beta \subset \alpha .$

[*110·12.*73·22] $\supset . (\exists \delta) . \delta \subset \alpha . \delta \, \mathrm{sm} \, \beta$ (2)

$\vdash . (1) . (2) . \supset \vdash . \mathrm{Prop}$

Volume II, page 86
(1st edition)

$*110{\cdot}641.\ \vdash .\ 1 +_c 0 = 0 +_c 1 = 1 \quad [*110{\cdot}51{\cdot}61 . *101{\cdot}2]$

$*110{\cdot}642.\ \vdash .\ 2 +_c 0 = 0 +_c 2 = 2 \quad [*110{\cdot}51{\cdot}61 . *101{\cdot}31]$

Volume II, page 86
(1st edition)

$*110{\cdot}643.\ \vdash .\ 1 +_c 1 = 2$

Dem.

$$\vdash .\ *110{\cdot}632 . *101{\cdot}21{\cdot}28 . \supset$$

$$\vdash .\ 1 +_c 1 = \hat{\xi}\{(\exists y) . y \in \xi . \xi - \iota`y \in 1\}$$

$$[*54{\cdot}3] \quad = 2 . \supset \vdash . \text{Prop}$$

The above proposition is occasionally useful. It is used at least three times, in *113·66 and *120·123·472.

*110·7·71 are required for proving *110·72, and *110·72 is used in *117·3, which is a fundamental proposition in the theory of greater and less.

$*110{\cdot}7.\quad \vdash :\beta \subset \alpha . \supset . (\exists \mu) . \mu \in \text{NC} . \text{Nc}`\alpha = \text{Nc}`\beta +_c \mu$

The foundations of arithmetic in C++

The foundations of (arithmetic in C++)

(The foundations of arithmetic) in C++

result_type function_name (*parameter_list*)
interface
  {

   // preconditions…

   implementation;

   // postconditions…

  }

The calling function is responsible for the top part of the interface.

The called function is responsible for the bottom part of the interface.

```
  +b    a  +  b    a  &  b    a == b
  -b    a  -  b    a  |  b    a  <  b
        a  *  b    a  ^  b    a  >  b
  ~b    a  /  b               a <= b
        a  %  b    a << b    a >= b
                   a >> b    a  != b
                             a <=> b


  ++b   a += b    a &= b    a  =  b
  --b   a -= b    a |= b
        a *= b    a ^= b
a++     a /= b
a--     a %= b    a <<= b
                  a >>= b
```

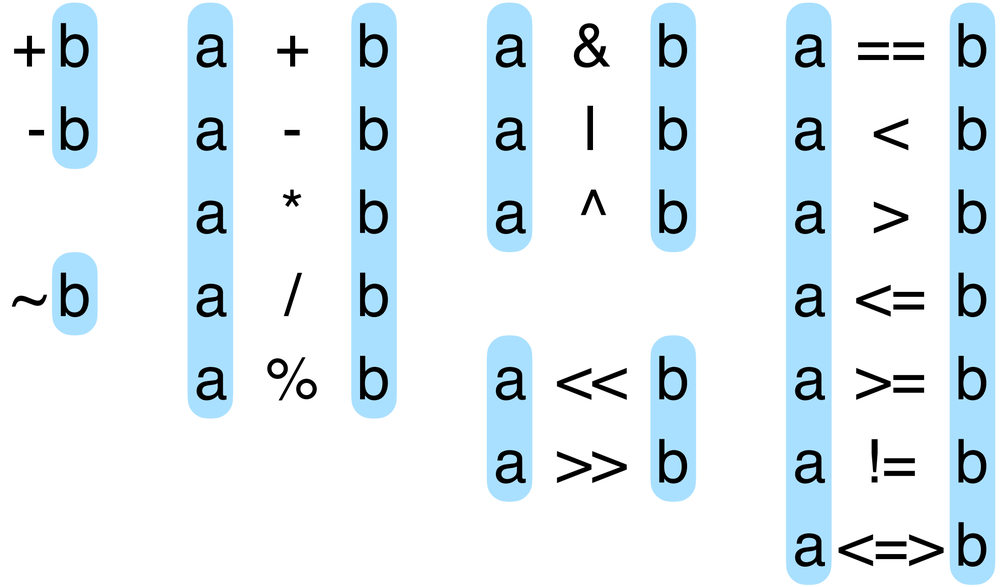
Stability	Over certain periods, an object's state, and therefore its value, remains stable.
Substitutability	At certain times, two different objects of the same type have interchangeable values.
Repeatability	An operation may be repeated by a sufficiently similar operation, producing similar results.

```
+b    a + b    a & b    a == b
-b    a - b    a | b    a <  b
      a * b    a ^ b    a >  b
~b    a / b             a <= b
      a % b    a << b   a >= b
               a >> b   a != b
                        a <=> b
```

```
++b    a += b    a &= b    a = b
--b    a -= b    a |= b
       a *= b    a ^= b
a++    a /= b
a--    a %= b    a <<= b
                 a >>= b
```

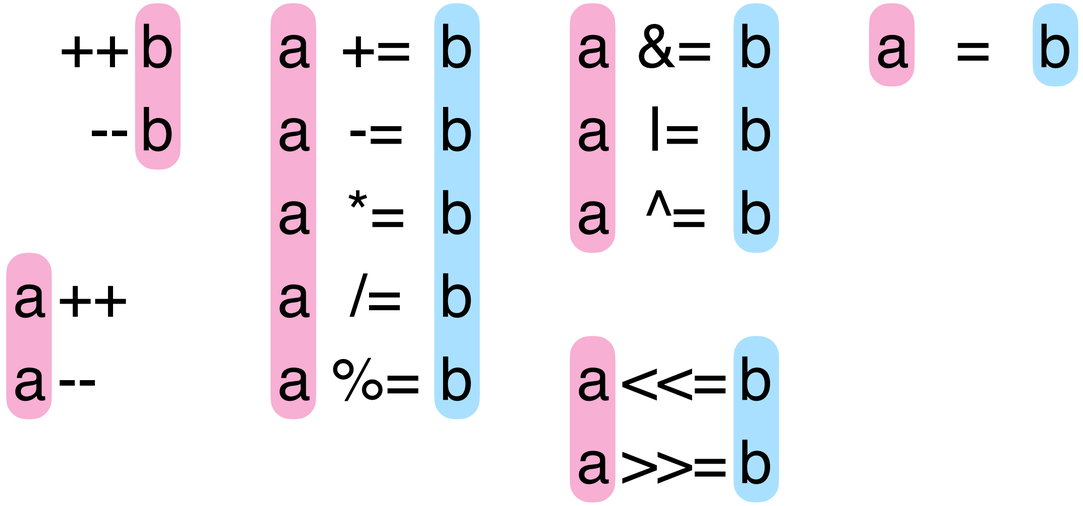
Right of stability

A right of stability is transferred from the caller to the implementation on entry, and from the implementation to the caller on exit.



Immunity from instability

The caller extends immunity from instability to the implementation for the duration of the operation.



Right of stability

A right of stability is transferred from the caller to the implementation on entry, and from the implementation to the caller on exit.

```
+b
-b

~b
```

```
a + b
a - b
a * b
a / b
a % b
```

```
a & b
a | b
a ^ b

a << b
a >> b
```

```
a == b
a < b
a > b
a <= b
a >= b
a != b
a <=> b
```

```
++b
--b

a++
a--
```

```
a += b
a -= b
a *= b
a /= b
a %= b
```

```
a &= b
a |= b
a ^= b

a <<= b
a >>= b
```

```
a = b
```

Right of stability

A right of stability of the function result is transferred from the implementation to the caller.

```
+b    a + b    a & b    a == b
-b    a - b    a | b    a < b
      a * b    a ^ b    a > b
~b    a / b             a <= b
      a % b    a << b   a >= b
               a >> b   a != b
                        a <=> b
```

```
++b    a += b    a &= b    a = b
--b    a -= b    a |= b
       a *= b    a ^= b
a++    a /= b
a--    a %= b    a <<= b
                 a >>= b
```

Substitutability

Most operations here that modify an argument return a reference aliased to that argument.

```
claim substitutable( &result, &a );

claim substitutable( &result, &b );
```

```
+b    a + b    a & b    a == b
-b    a - b    a | b    a < b
       a * b    a ^ b    a > b
~b    a / b             a <= b
       a % b    a << b   a >= b
                a >> b   a != b
                         a <=> b
```

```
if ( result )
  claim substitutable( a, b );
```

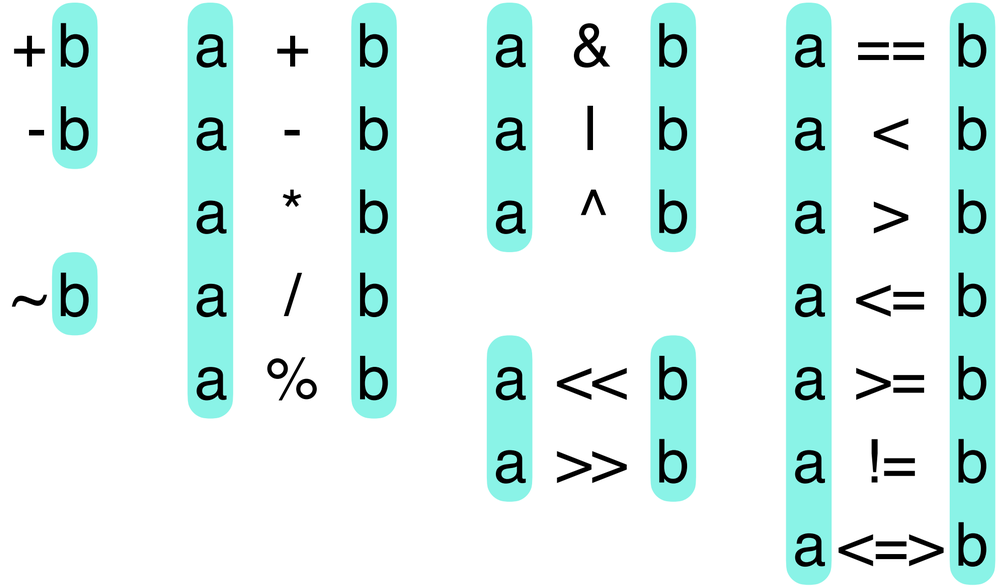
```
if ( !result )
  claim substitutable( a, b );
```

```
++b    a += b    a &= b    a = b
--b    a -= b    a |= b
       a *= b    a ^= b
a++    a /= b
a--    a %= b    a <<= b
                 a >>= b
```

```
claim substitutable( a, b );
```

Substitutability

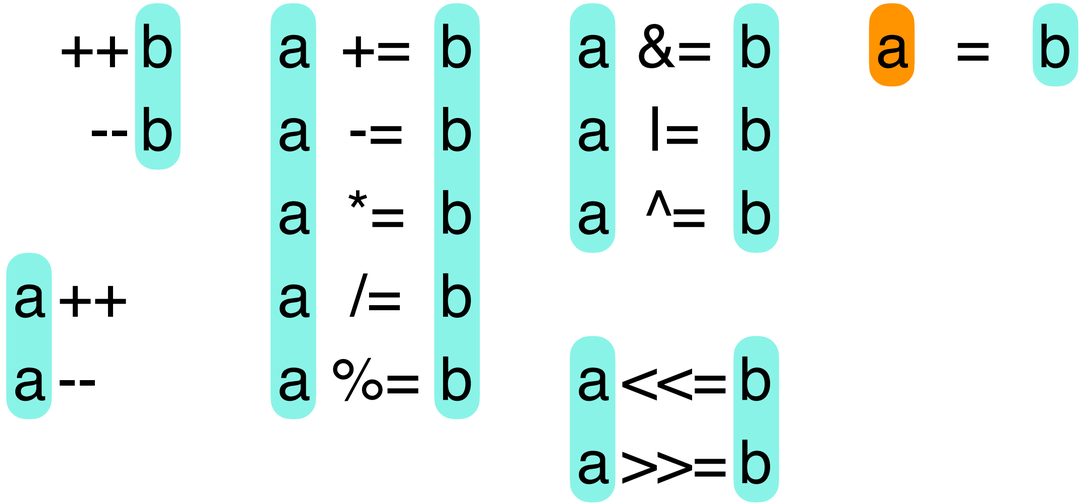
These operations may also have substitutability as a postcondition.



Discernible input

Most parameters are part of the discernible input to the operations.

An operation is repeated when these parameters are repeated.



The left parameter of **operator=** is not part of the discernible input.

```
  +b    a  +  b    a  &  b    a == b
  -b    a  -  b    a  |  b    a  <  b
        a  *  b    a  ^  b    a  >  b
  ~b    a  /  b               a <= b
        a  %  b    a << b    a >= b
                   a >> b    a  != b
                             a <=> b


  ++b   a += b    a &= b    a  =  b
  --b   a -= b    a |= b
        a *= b    a ^= b
a ++    a /= b
a --    a %= b    a <<= b
                  a >>= b
```

Discernible output

The function result is part of the discernible output for every operation listed here.

Left-hand arguments of **operator++** and **operator--** are also part of the discernible output.

If an operation is repeated, the discernible output will be repeated.

```
int& operator=( int& a, const int b )
interface
  {
   claim_right a;                                  // right of stability
   claim_immunity b;                               // immunity from instability
   discern b;                                      // discernible input

    implementation;

   claim substitutable( &a, &result );             // substitutability
   claim_right result;                             // right of stability
   discern result;                                 // discernible output

   claim substitutable( result, b );               // substitutability
  }
```

bool

unsigned char

char

signed char

char8_t

unsigned short int

char16_t

short int

char32_t

unsigned int

wchar_t

int

unsigned long int

long int

unsigned long long int

long long int

bool

char

unsigned char

char8_t

signed char

unsigned short int

char16_t

short int

unsigned int

char32_t

int

unsigned long int

wchar_t

long int

unsigned long long int

long long int

unsigned longer long int

longer long int

unsigned extremely long int

extremely long int

unsigned overwhelmingly long int

overwhelmingly long int

unsigned mind numbingly vast int

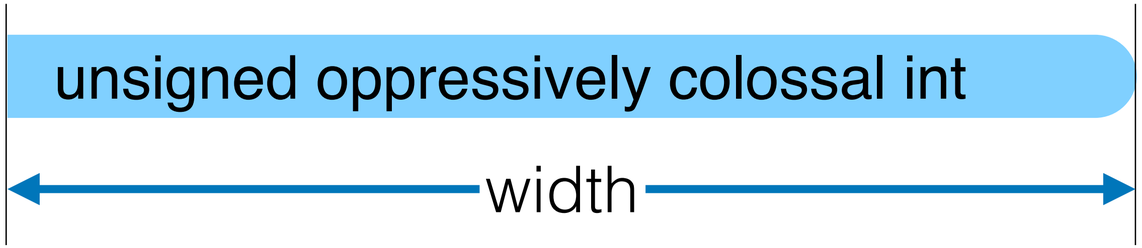
mind numbingly vast int

unsigned oppressively colossal int

oppressively colossal int

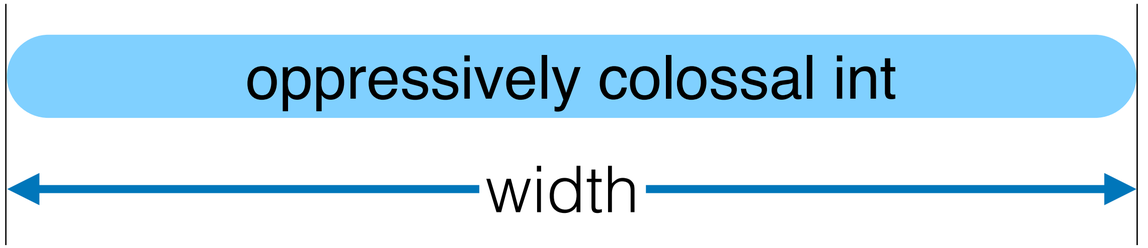
Unsigned

$0 \leq \text{value} < 2^{\text{width}}$



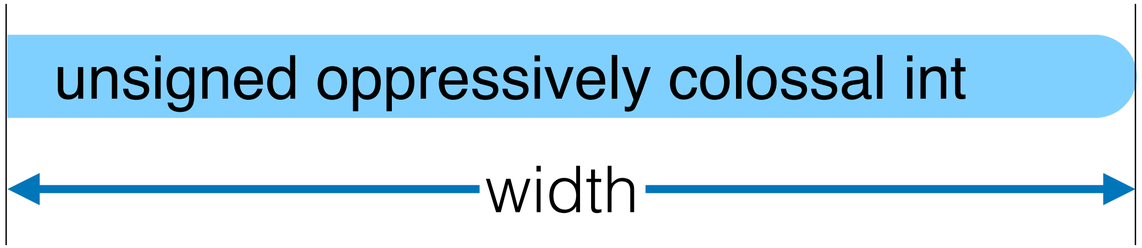
Signed

$-2^{\text{width-1}} \leq \text{value} < 2^{\text{width-1}}$



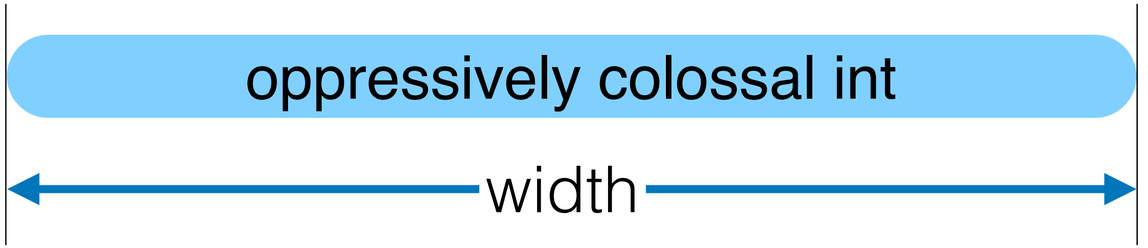
Unsigned

$0 \leq value < 2^{width}$



Signed

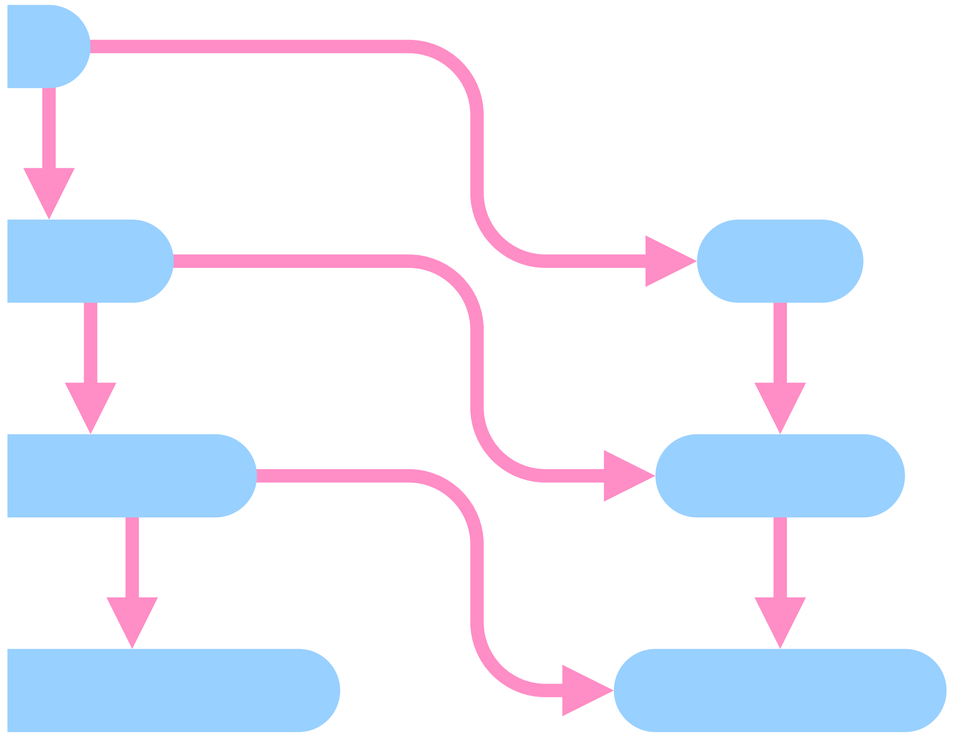
$-2^{width-1} \leq value < 2^{width-1}$

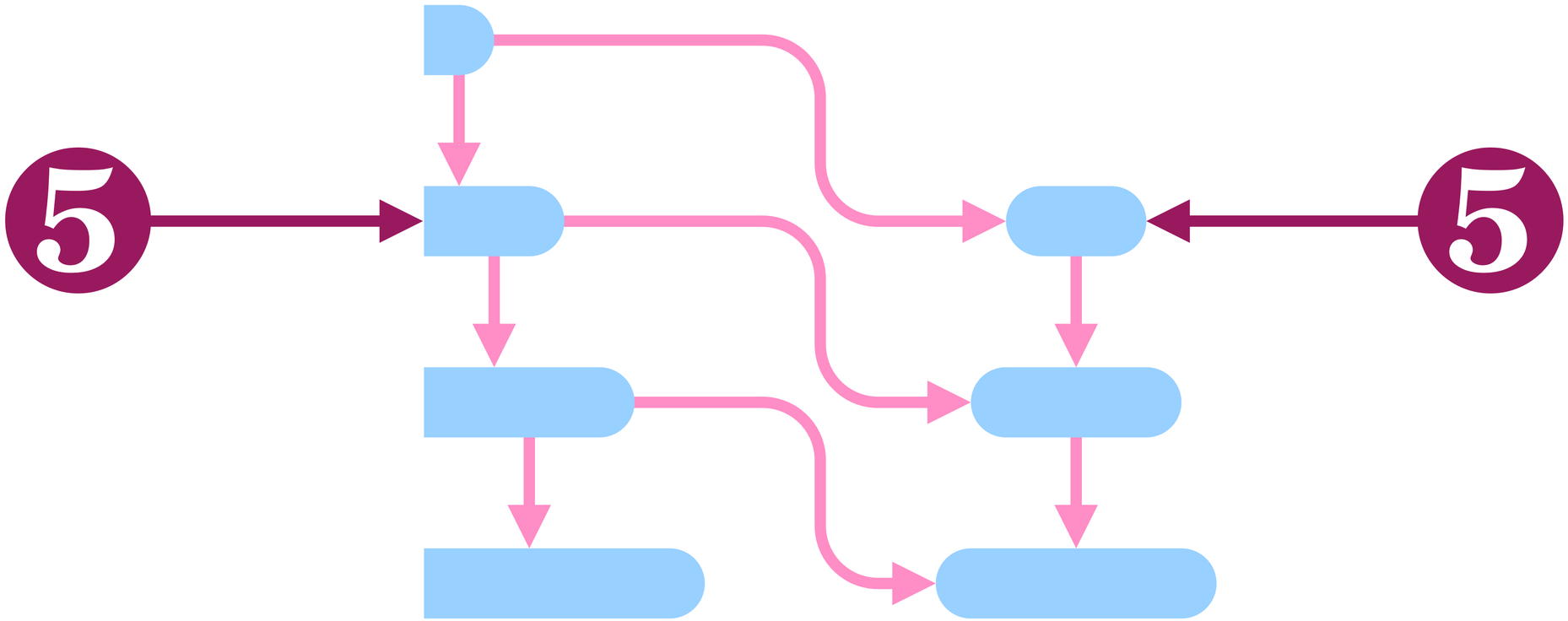


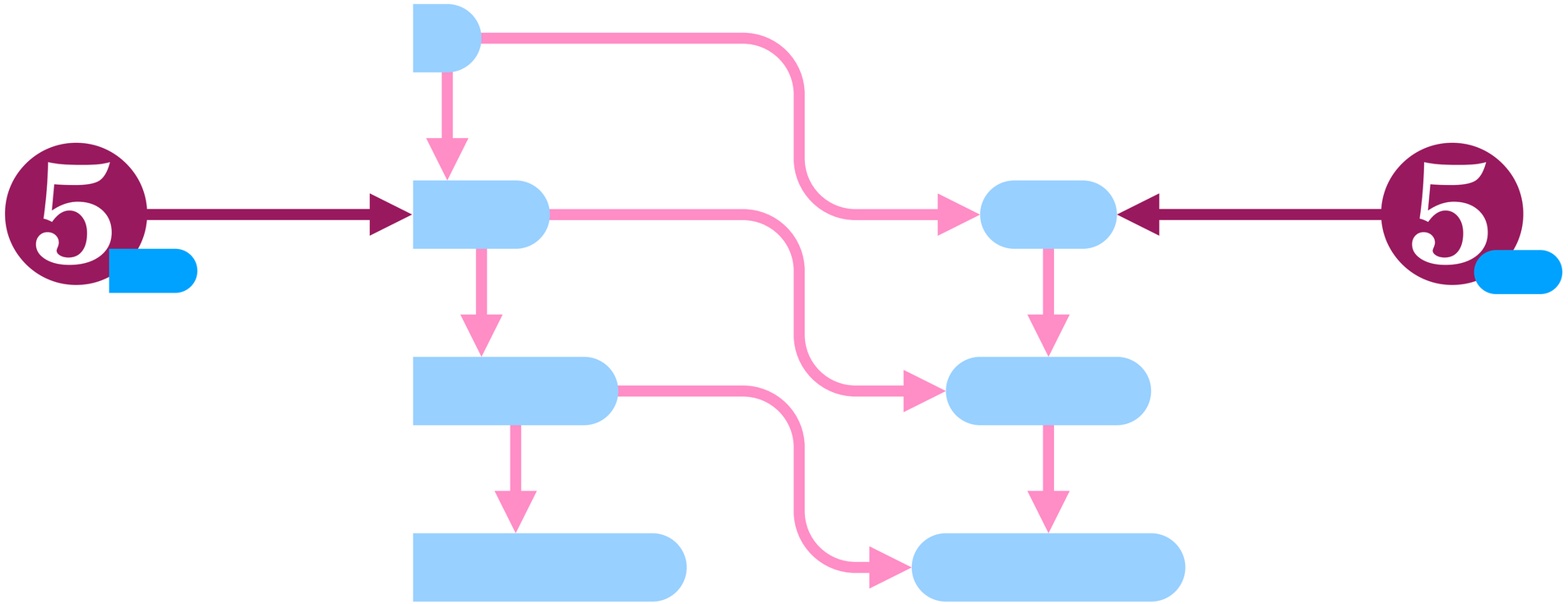

```
class integer_kind
  {
   // …
   constexpr bool          is_signed()  const;
   constexpr bit_size_t    width()      const;
  };
```

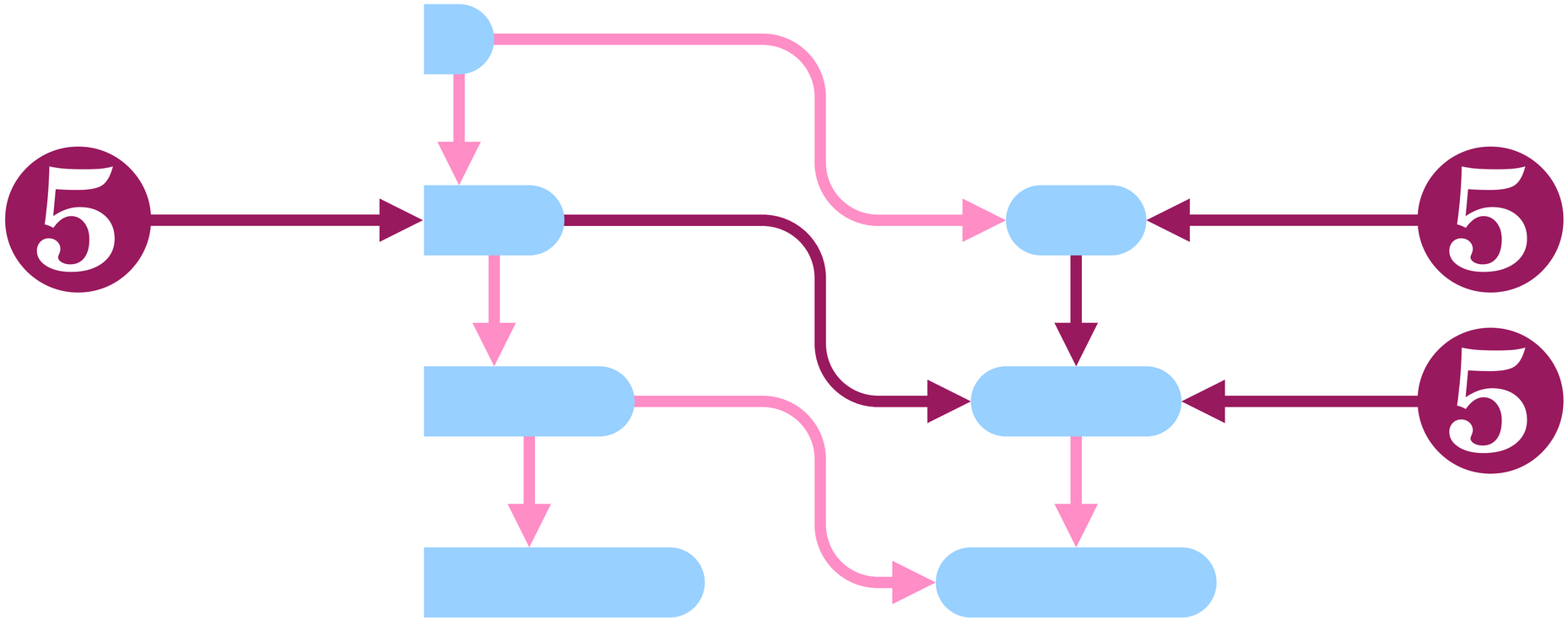
```
class integer_kind
  {
   // …
   constexpr bool          is_signed()  const;
   constexpr bit_size_t    width()      const;
  };



inline constexpr
bool operator<( integer_kind a, integer_kind b )
  {
   return  a.is_signed()  <=  b.is_signed()
       &&  a.width()      <   b.width();
  }
```

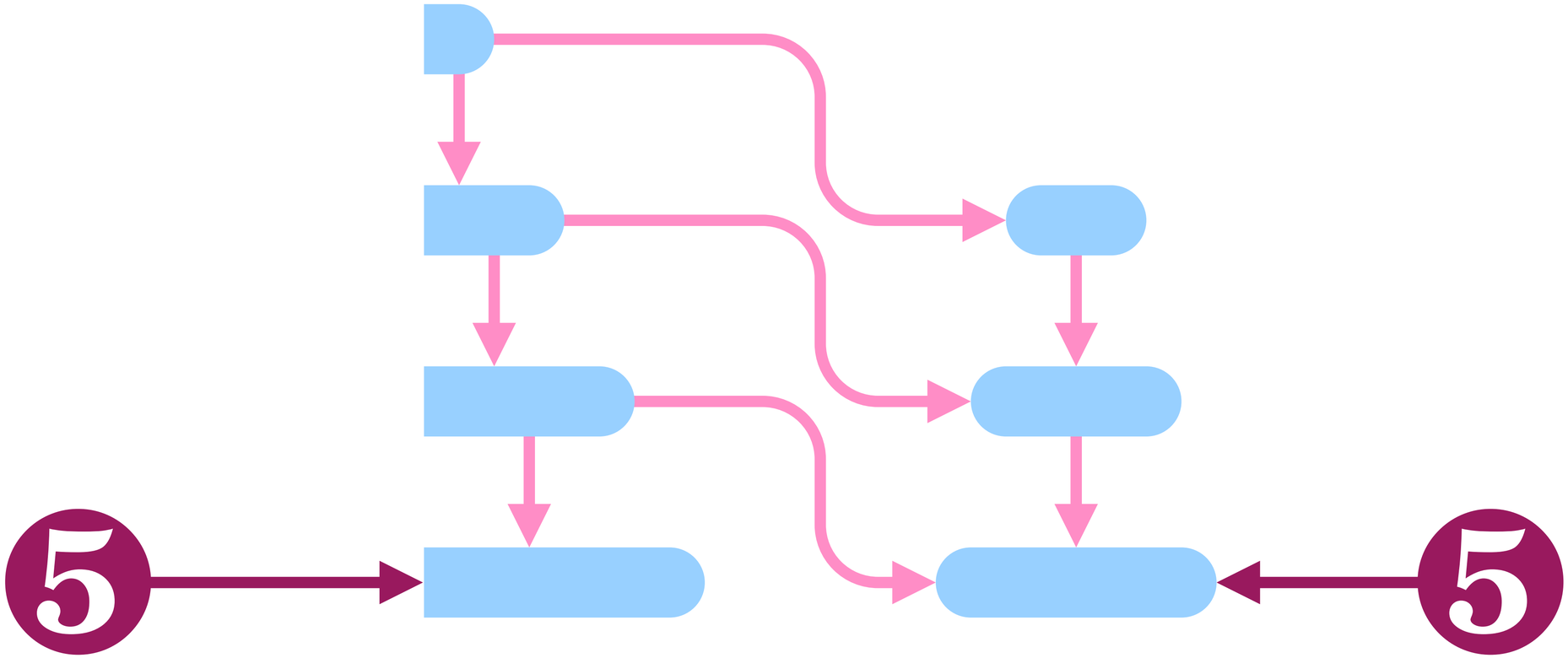
```
class integer_kind
  {
   // …
   constexpr bool          is_signed()  const;
   constexpr bit_size_t    width()      const;
  };


inline constexpr
bool operator<( integer_kind a, integer_kind b )
  {
   return  a.is_signed()  <=  b.is_signed()
       &&  a.width()      <   b.width();
  }


template < class To, class From >
  requires (  integer_kind_of<To>  >=  integer_kind_of<From>  )
To convert( const From& );
```

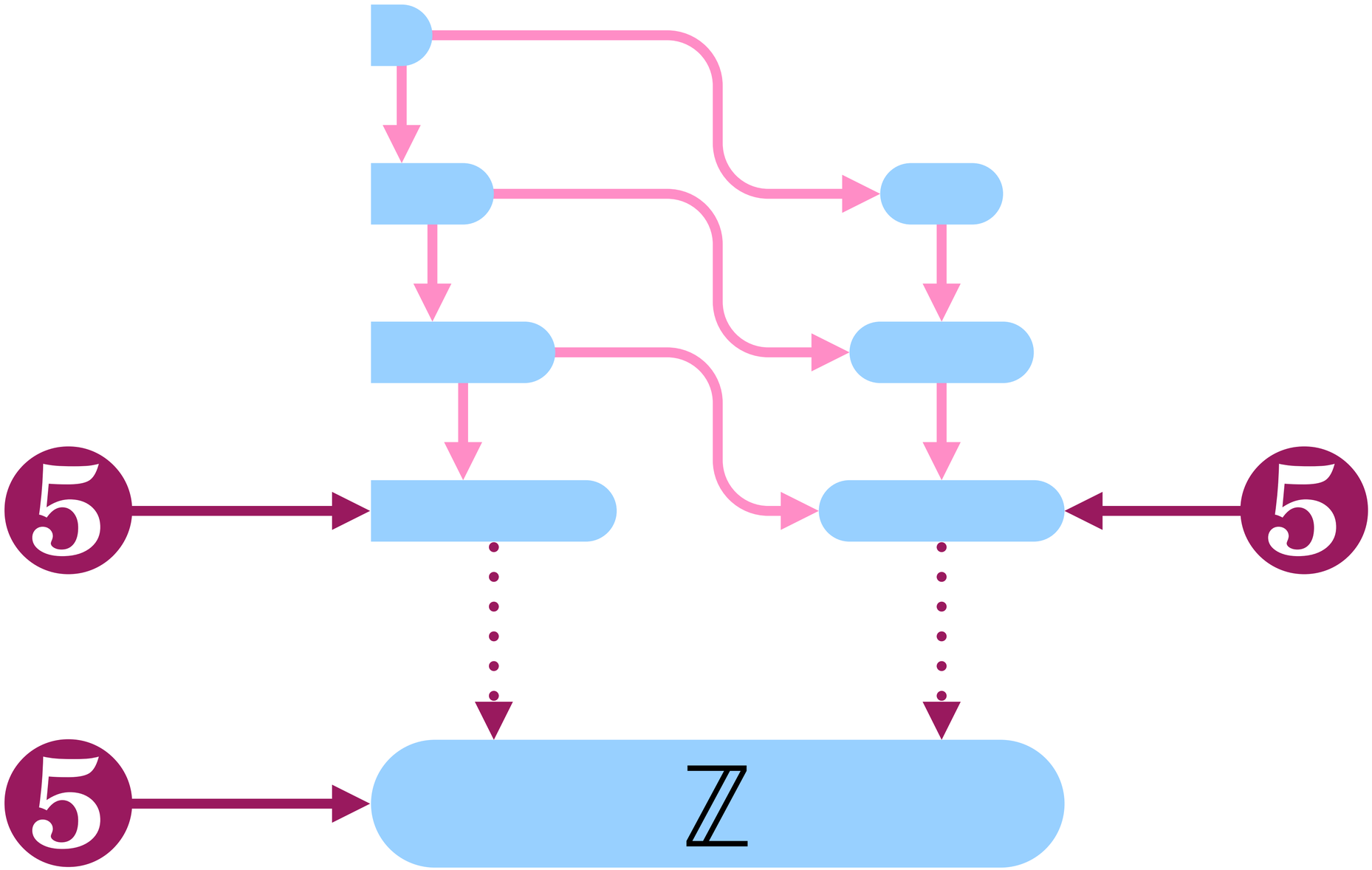



5
5


5
5


5
5
5


5
5


5
5
5
ℤ

To convert(const From&); ——— Strong type requirement:
All values must be convertible

To convert_narrowing(const From&); ——— Strong precondition:
Argument must be convertible

To convert_modular(const From&); ——— Weak postcondition:
Only the low bits are converted

```
int& operator=( int& a, const int b )
interface
  {
   claim_right a;                                  // right of stability
   claim_immunity b;                               // immunity from instability
   discern b;                                      // discernible input

    implementation;

   claim substitutable( &a, &result );             // substitutability
   claim_right result;                             // right of stability
   discern result;                                 // discernible output

   claim substitutable( result, b );               // substitutability
  }
```

```
template < std::integral A, std::integral B >
A& operator=( A& a, const B b )
interface
  {
   claim_right a;                                   // right of stability
   claim_immunity b;                                // immunity from instability
   discern b;                                       // discernible input

    implementation;

   claim substitutable( &a, &result );              // substitutability
   claim_right result;                              // right of stability
   discern result;                                  // discernible output

   claim substitutable( result, convert_modular<A>( b ) );
  }
```

+b	a + b	a & b	a == b	convert
-b	a - b	a | b	a < b	convert_modular
	a * b	a ^ b	a > b	convert_narrowing
~b	a / b		a <= b	
	a % b	a << b	a >= b	
		a >> b	a != b	
			a <=> b	

++b	a += b	a &= b	a = b
--b	a -= b	a |= b	
	a *= b	a ^= b	
a++	a /= b		
a--	a %= b	a <<= b	
		a >>= b	

+b	a + b	a & b	a == b	convert
-b	a - b	a | b	a < b	convert_modular
	a * b	a ^ b	a > b	convert_narrowing
~b	a / b		a <= b	
	a % b	a << b	a >= b	
		a >> b	a != b	
			a <=> b	

++b	a += b	a &= b
--b	a -= b	a |= b
	a *= b	a ^= b
a++	a /= b	
a--	a %= b	a <<= b
		a >>= b

The behavior of an expression of the form **E1 *op*= E2** is equivalent to **E1 = E1 *op* E2** except that **E1** is evaluated only once.

7.6.19 [expr.ass]

```
template < std::integral A, std::integral B >
A& operator+=( A& a, const B b )
interface
  {
   claim_right a;                                   // right of stability
   claim_immunity b;                                // immunity from instability
   discern a;                                       // discernible input
   discern b;                                       // discernible input

   const auto expected_result = convert_modular< A >( a+b );

    implementation;

   claim substitutable( &a, &result );              // substitutability
   claim_right result;                              // right of stability
   discern result;                                  // discernible output

   claim substitutable( result, expected_result );  // substitutability
  }
```

```
template < std::integral A, std::integral B >
A& operator+=( A& a, const B b )
interface
  {
   claim_right a;                                   // right of stability
   claim_immunity b;                                // immunity from instability
   discern a;                                       // discernible input
   discern b;                                       // discernible input

   const auto expected_result = convert_modular< A >( a+b );

    implementation;

   claim substitutable( &a, &result );              // substitutability
   claim_right result;                              // right of stability
   discern result;                                  // discernible output

   claim result == expected_result;                 // substitutability
  }
```

+b	a + b	a & b	a == b	convert
-b	a - b	a | b	a < b	convert_modular
	a * b	a ^ b	a > b	convert_narrowing
~b	a / b		a <= b	
	a % b	a << b	a >= b	
		a >> b	a != b	
			a <=> b	

++b	a += b	a &= b
--b	a -= b	a |= b
	a *= b	a ^= b
a++	a /= b	
a--	a %= b	a <<= b
		a >>= b

The behavior of an expression of the form **E1 *op*= E2** is equivalent to **E1 = E1 *op* E2** except that **E1** is evaluated only once.

7.6.19 [expr.ass]

+b	a + b	a & b	a == b	convert
-b	a - b	a | b	a < b	convert_modular
	a * b	a ^ b	a > b	convert_narrowing
~b	a / b		a <= b	
	a % b	a << b	a >= b	
next		a >> b	a != b	
			a <=> b	

++b ——— const auto expected_result = next(b);

--b

a ++

a --

+b	a + b	a & b	a == b	convert
-b	a - b	a | b	a < b	convert_modular
	a * b	a ^ b	a > b	convert_narrowing
~b	a / b		a <= b	
	a % b	a << b	a >= b	
next		a >> b	a != b	
prev			a <=> b	

--b ——— const auto expected_result = prev(b);

a ++

a --

+b	a + b	a & b	a == b
-b	a - b	a | b	a < b
	a * b	a ^ b	a > b
~b	a / b		a <= b
	a % b	a << b	a >= b
next		a >> b	a != b
prev			a <=> b

convert
convert_modular
convert_narrowing

a ++
a --

```
const auto expected_result = a;
const auto expected_a      = next( a );

// …implementation…

claim result == expected_result;
claim a      == expected_a;
```

+b	a + b	a & b	a == b
-b	a - b	a | b	a < b
	a * b	a ^ b	a > b
~b	a / b		a <= b
	a % b	a << b	a >= b
next		a >> b	a != b
prev			a <=> b

convert
convert_modular
convert_narrowing

a --

```
const auto expected_result = a;
const auto expected_a      = prev( a );

// …implementation…

claim result == expected_result;
claim a      == expected_a;
```

bool

char

unsigned char

char8_t

signed char

unsigned short int

char16_t

short int

unsigned int

char32_t

int

unsigned long int

wchar_t

long int

unsigned long long int

long long int

unsigned longer long int

longer long int

unsigned extremely long int

extremely long int

unsigned overwhelmingly long int

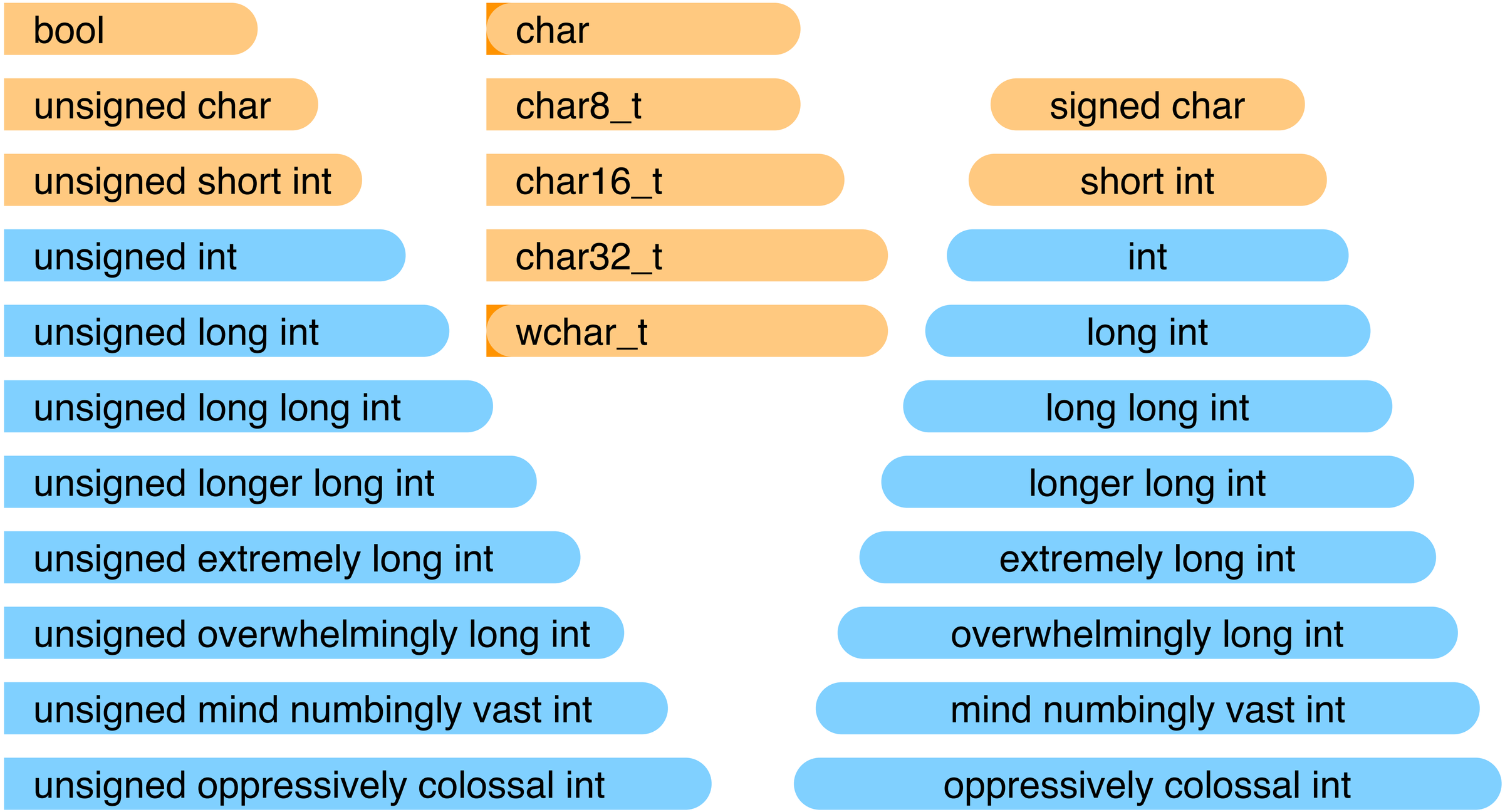
overwhelmingly long int

unsigned mind numbingly vast int

mind numbingly vast int

unsigned oppressively colossal int

oppressively colossal int


bool
char
unsigned char
char8_t
signed char
unsigned short int
char16_t
short int
unsigned int
char32_t
int
unsigned long int
wchar_t
long int
unsigned long long int
long long int
unsigned longer long int
longer long int
unsigned extremely long int
extremely long int
unsigned overwhelmingly long int
overwhelmingly long int
unsigned mind numbingly vast int
mind numbingly vast int
unsigned oppressively colossal int
oppressively colossal int

If T is on this list:	T promotes to the first type in this list to which it has a nice conversion:	
wchar_t char16_t char32_t		int unsigned int long unsigned long long long unsigned long long the underlying type of T
If T is any other integral type	T promotes to the first type in this list to which it has a nice conversion:	int unsigned int T

```
template < std::integral T >
using promoted_type =
   std::conditional_t< !appears_in_list<   T,       wchar_t, char16_t, char32_t       >,
                       first_nicely_convertable<    T,
                                                    int,            unsigned int,
                                                    T    >,
                       first_nicely_convertable<    T,
                                                    int,            unsigned int,
                                                    long,           unsigned long,
                                                    long long,   unsigned long long,
                                                    std::underlying_type_t<T>         > >;
```

⚠️ This gives the wrong answer for bit fields.

⚠️ **underlying_type** doesn't work for character types.

```
template < std::integral T >
using promoted_type = decltype( +std::declval<T>() );


template < class T >
concept promoted_integral     =         std::integral< T >
                                   &&   std::same_as<  T, promoted_type< T >  >;

template < class T >
concept unpromoted_integral =         std::integral< T >
                                   && ! std::same_as<  T, promoted_type< T >  >;
```

⚠️ This gives the wrong answer for bit fields.

```
template < std::integral B >
promoted_type<B> operator+( const B& b )
interface
  {
   claim_immunity b;                                  // immunity from instability
   discern b;                                         // discernible input

   const auto expected_result = convert< promoted_type<B> >( b );

    implementation;

   claim_right result;                                // right of stability
   discern result;                                    // discernible output

   claim result == expected_result;                   // substitutability
  }
```

+b	a + b	a & b	a == b	convert
-b	a - b	a | b	a < b	convert_modular
	a * b	a ^ b	a > b	convert_narrowing
~b	a / b		a <= b	
	a % b	a << b	a >= b	
next		a >> b	a != b	
prev			a <=> b	

```
template < unpromoted_integral B >
promoted_type<B> operator-( const B& b )
interface
  {
   claim_immunity b;                                  // immunity from instability
   discern b;                                         // discernible input

   const auto expected_result = -( +b );

    implementation;

   claim_right result;                                // right of stability
   discern result;                                    // discernible output

   claim result == expected_result;                   // substitutability
  }
```

Promoted parameters

	a + b	a & b	a == b
-b	a - b	a | b	a < b
	a * b	a ^ b	a > b
~b	a / b		a <= b
	a % b	a << b	a >= b
next		a >> b	a != b
prev			a <=> b

Integral parameters

convert

convert_modular

convert_narrowing

```
template < promoted_integral T >
inline auto usual_arithmetic_conversions( const T& a, const T& b )
  {
   return std::pair( a, b );
  }
```

```
template < promoted_integral A, promoted_integral B >
inline auto usual_arithmetic_conversions( const A& a, const B& b )
  {
   // …
  }
```

```
template < promoted_integral A, promoted_integral B >
inline auto usual_arithmetic_conversions( const A& a, const B& b )
  {
   constexpr ak = integer_kind_of<A>;
   constexpr bk = integer_kind_of<B>;

   if       constexpr ( ak  >  bk )     return std::pair(                a,  convert<A>(b) );
   else if constexpr ( ak  <  bk )     return std::pair( convert<B>(a),                b  );
   else if constexpr ( ak == bk )
     {
      // Can convert either way…
     }
   else // if the integer kinds are unordered
     {
      // Cannot convert either way…
     }
  }
```

```
{
 constexpr ak = integer_kind_of<A>;
 constexpr bk = integer_kind_of<B>;

 if      constexpr ( ak  >  bk )    return std::pair(                a,  convert<A>(b) );
 else if constexpr ( ak  <  bk )    return std::pair( convert<B>(a),                b  );
 else if constexpr ( ak == bk )
   {
    // Can convert either way. Break the tie with integer conversion rank.

    if constexpr ( integer_conversion_rank_is_less< A, B > )
       return std::pair( convert<B>(a), b );
    else
       return std::pair( a, convert<A>(b) );
   }
 else // if the integer kinds are unordered
   {
    // Cannot convert either way…
   }
}
```

```
else if constexpr ( ak < bk ) return std::pair( convert<B>(a), b );
else if constexpr ( ak == bk )
  {
   // Can convert either way. Break the tie with integer conversion rank.

   if constexpr ( integer_conversion_rank_is_less< A, B > )
      return std::pair( convert<B>(a), b );
   else
      return std::pair( a, convert<A>(b) );
  }
else // if the integer kinds are unordered
  {
   // Cannot convert either way. Try again with unsigned arguments.

   const auto unsigned_a = convert_modular< std::make_unsigned_t<A> >( a );
   const auto unsigned_b = convert_modular< std::make_unsigned_t<B> >( b );

   return usual_arithmetic_conversions( unsigned_a, unsigned_b );
  }
}
```

```
template < std::integral A, std::integral B >
using UAC_type = decltype( std::declval<A>() + std::declval<B>() );


template < std::integral A, std::integral B >
inline auto usual_arithmetic_conversions( const A& a, const B& b )
  {
   using U = UAC_type< A, B >;

   return std::pair(  convert_modular<U>( a ),  convert_modular<U>( b )  );
  }
```

```
template < std::promoted_integral A, std::promoted_integral B >
  requires ( !std::same_as<A,B> )
UAC_type<A,B> operator+( const A& a, const B& b )
interface
  {
   claim_immunity a;                                    // immunity from instability
   claim_immunity b;                                    // immunity from instability
   discern a;                                           // discernible input
   discern b;                                           // discernible input

   const auto [ a1, b1 ] = usual_arithmetic_conversions( a, b );
   const auto expected_result = a1 + b1;

    implementation;

   claim_right result;                                  // right of stability
   discern result;                                      // discernible output

   claim result == expected_result;                     // substitutability
  }
```

Promoted parameters

-b	a << b	next
~b	a >> b	prev

Integral parameters

convert
convert_modular
convert_narrowing

Matching, promoted parameters

a + b	a & b	a == b
a - b	a | b	a < b
a * b	a ^ b	a > b
a / b		a <= b
a % b		a >= b
		a != b
		a <=> b

Regarding `a/b` and `a%b`:

> If the second operand of `/` or `%` is zero the behavior is undefined.
>
> …if the quotient **`a/b`** is representable in the type of the result, **`(a/b)*b + a%b`** is equal to **`a`**; otherwise, the behavior of both **`a/b`** and **`a%b`** is undefined.
>
> 7.6.5 [expr.mul]

Regarding `a << b` and `a >> b`:

> The behavior is undefined if the right operand is negative, or greater than or equal to the width of the promoted left operand.
>
> 7.6.7 [expr.shift]

a / b

```
claim b != 0;

// …implementation…
```

a % b

```
claim b != 0;
const auto quotient = a / b;

// …implementation…

claim quotient * b + result == a;
```

a << b
a >> b

```
const auto b1 = convert_narrowing< bit_size_t >( b );
claim   b1   <   integer_kind_of<A>.width();

// …implementation…
```

If during the evaluation of an expression, the result is not mathematically defined or not in the range of representable values for its type, the behavior is undefined.

7.1 [expr.pre]

Arithmetic for the unsigned type is performed modulo 2^N. [*Note*: Unsigned arithmetic does not overflow. Overflow for signed arithmetic yields undefined behavior (7.1). —*end note*]

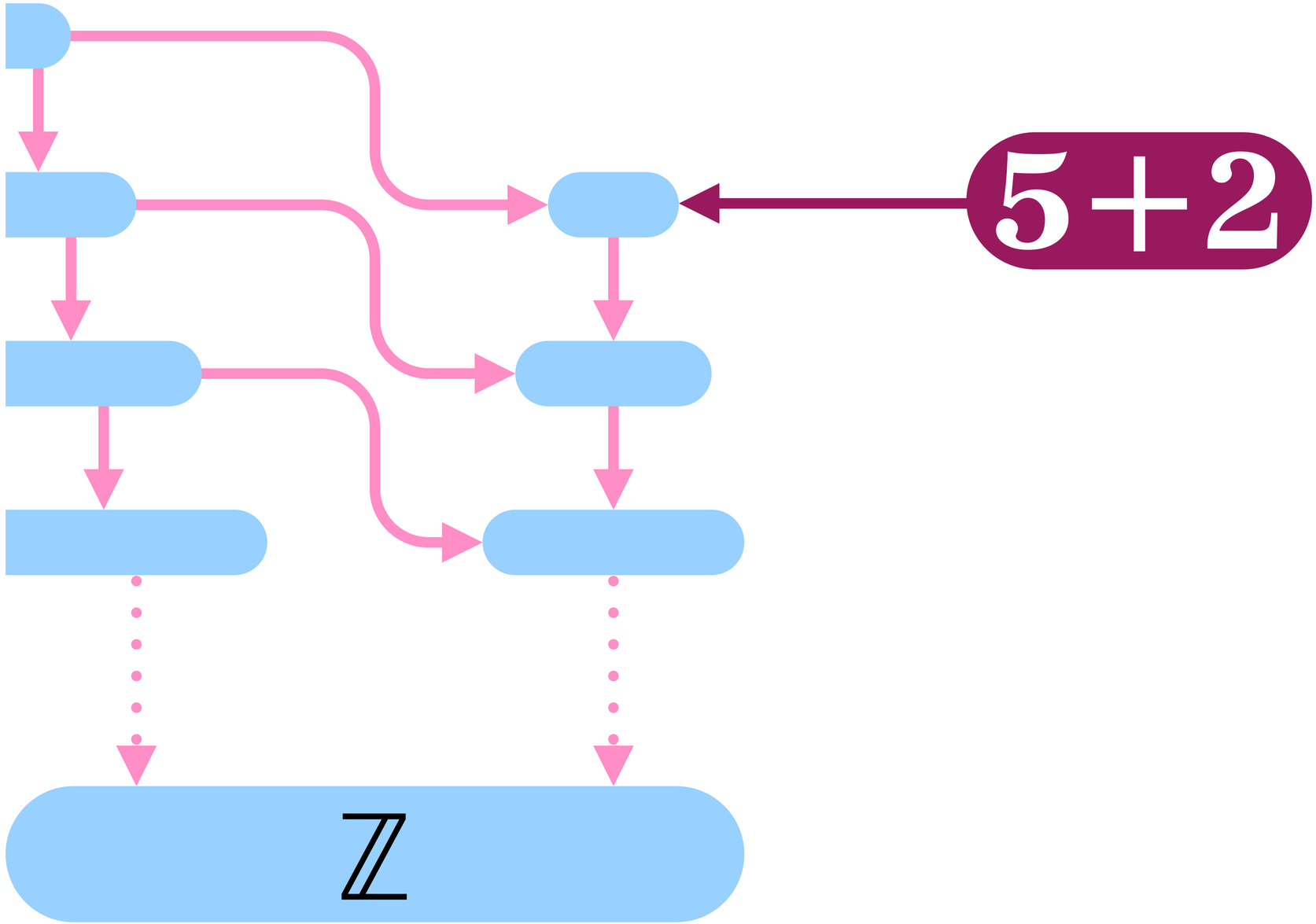
6.7.1 [basic.fundamental]

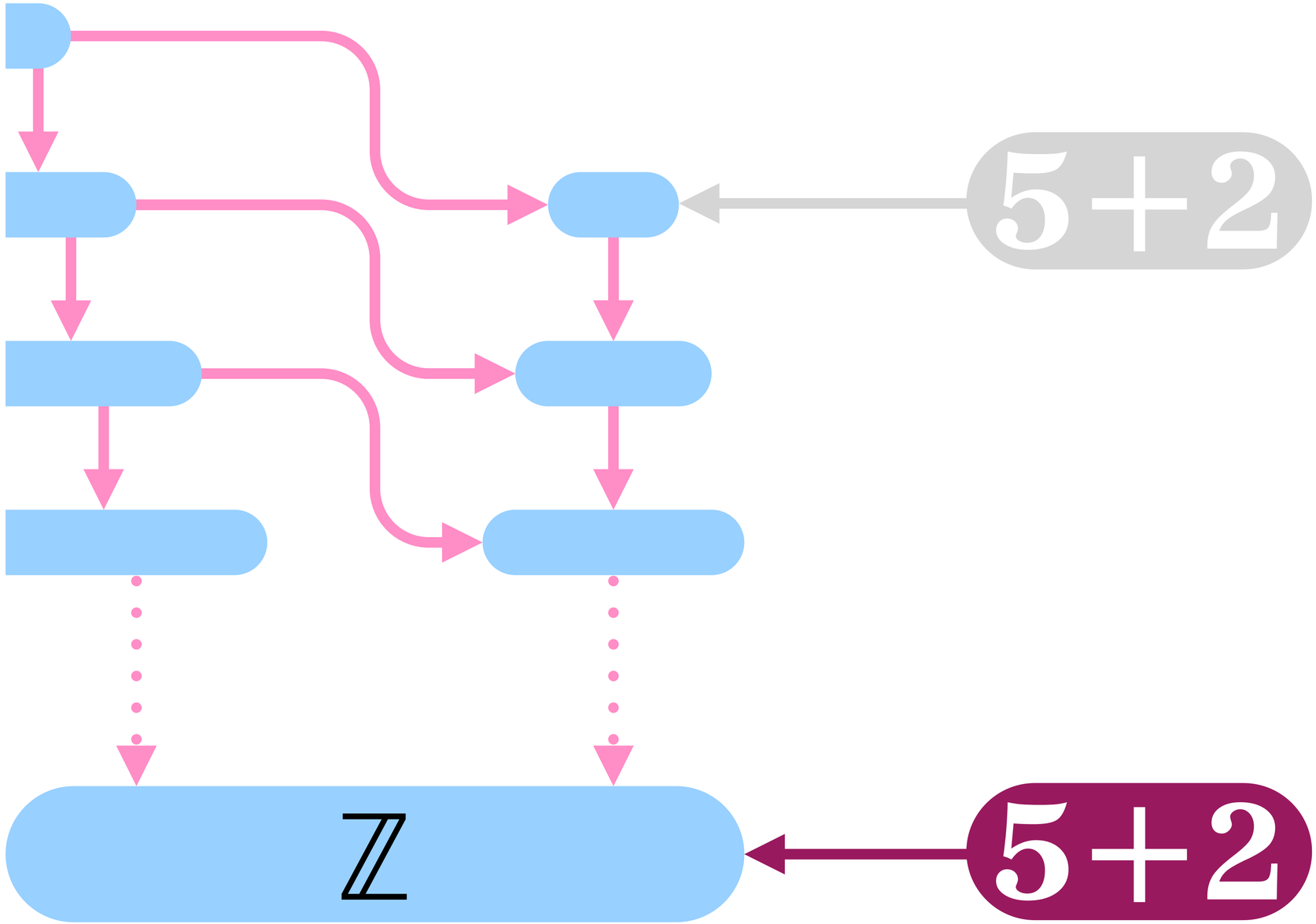
If during the evaluation of an expression, the result is not mathematically defined or not in the range of representable values for its type, the behavior is undefined.

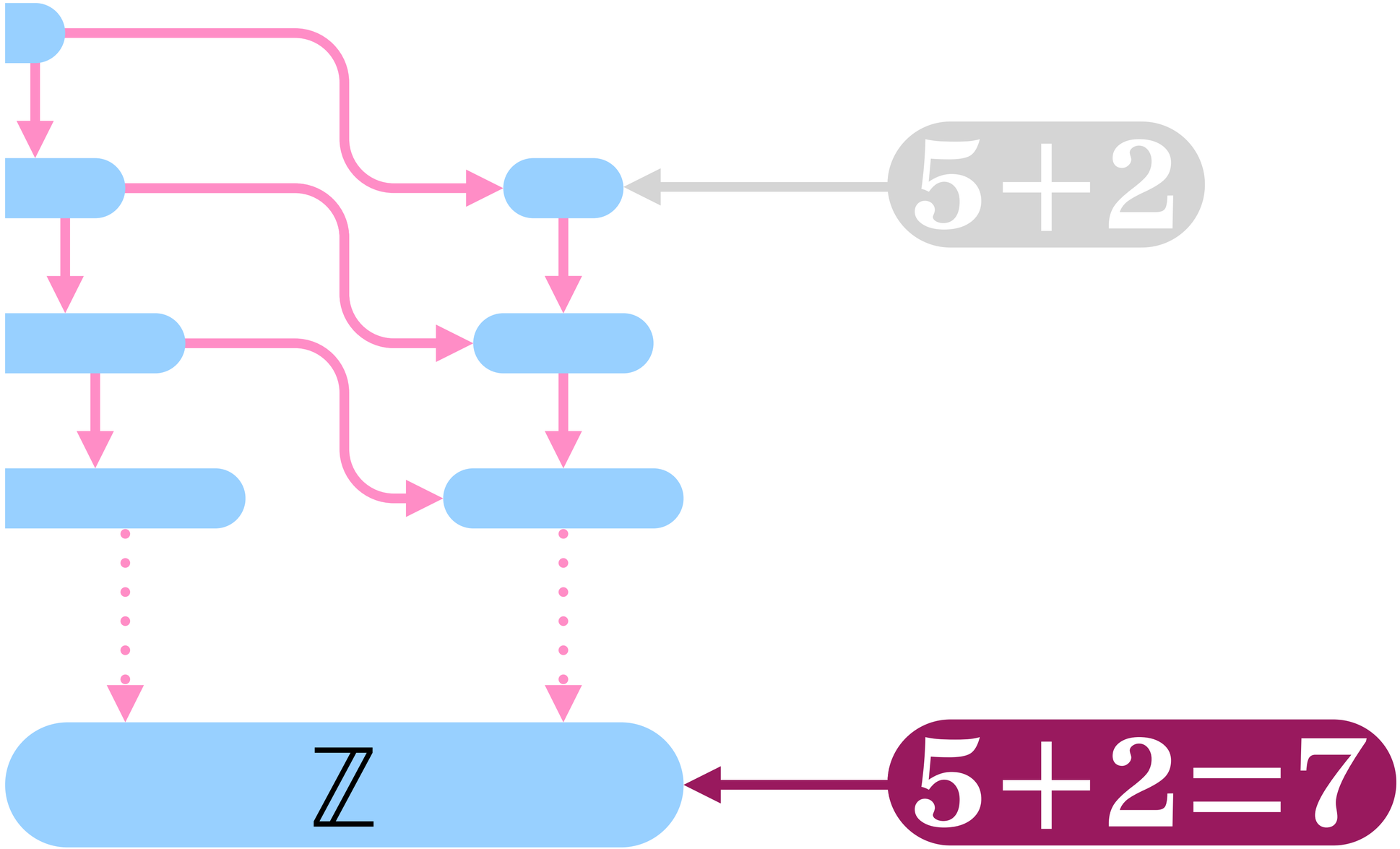
7.1 [expr.pre]

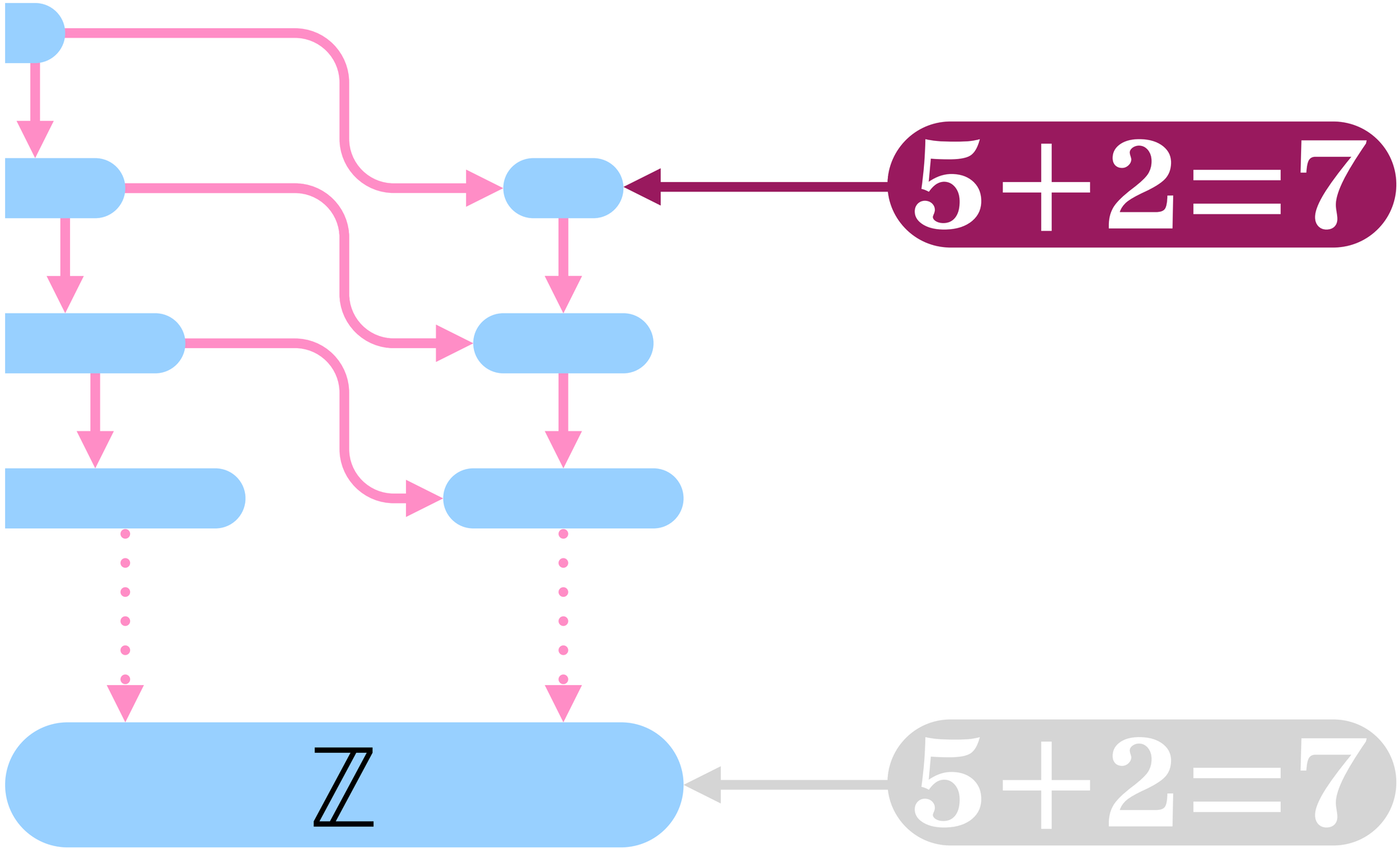
The value of **`E1 << E2`** is the unique value congruent to **`E1`** $\times 2^{\mathtt{E2}}$ modulo 2^N , where N is the width of the type of the result.

7.6.7 [expr.shift]

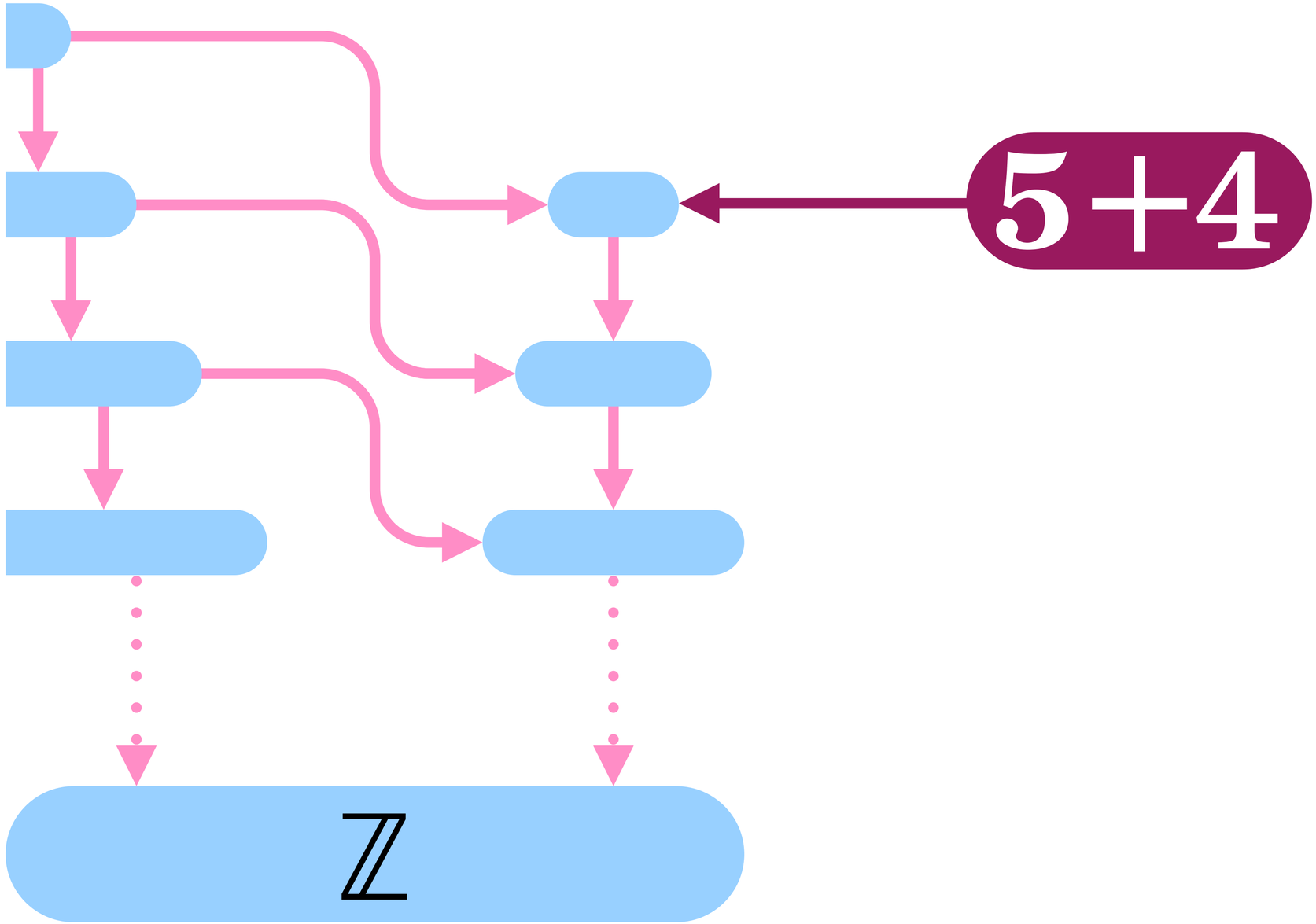

5+2
ℤ

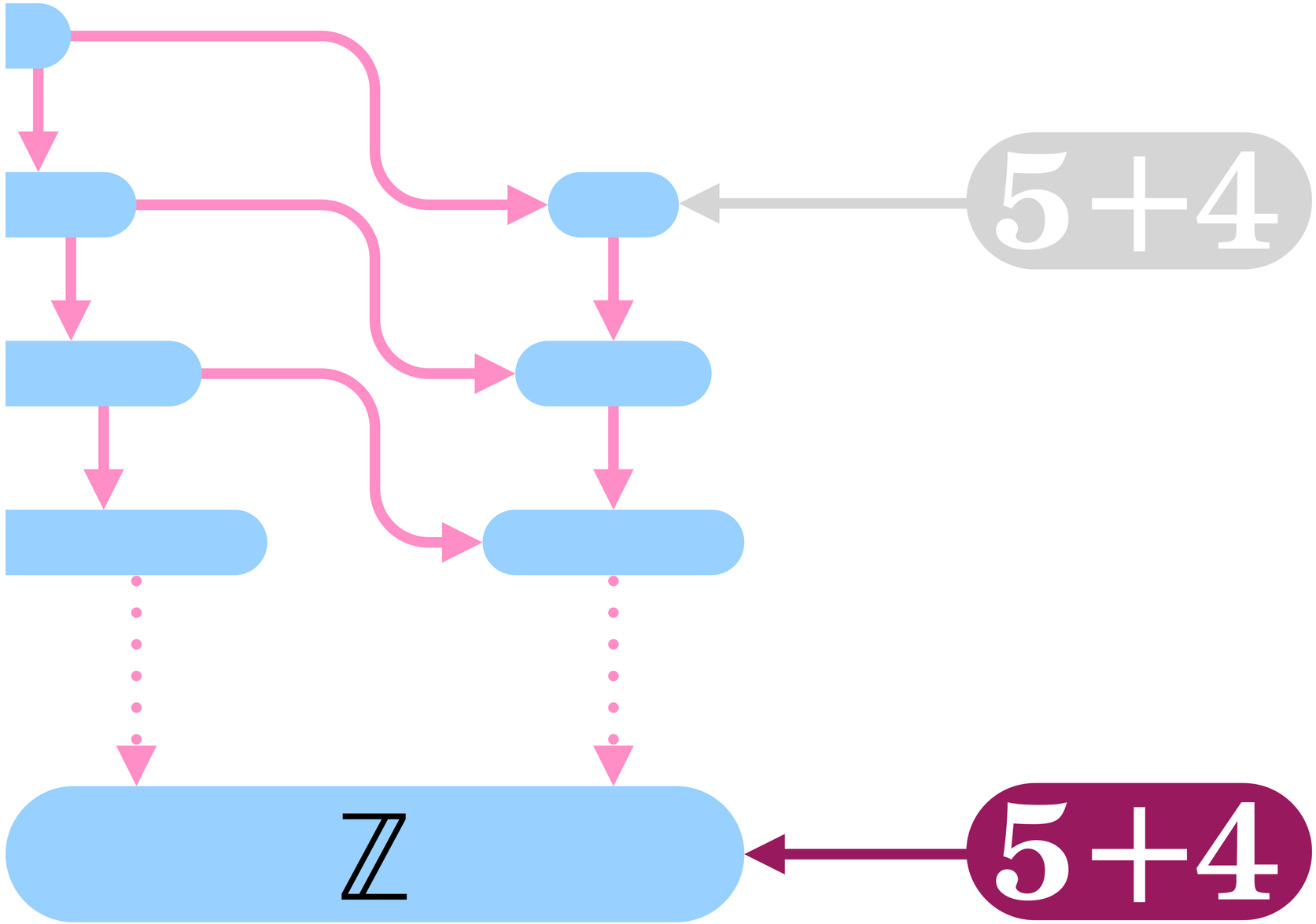

5+2
ℤ
5+2

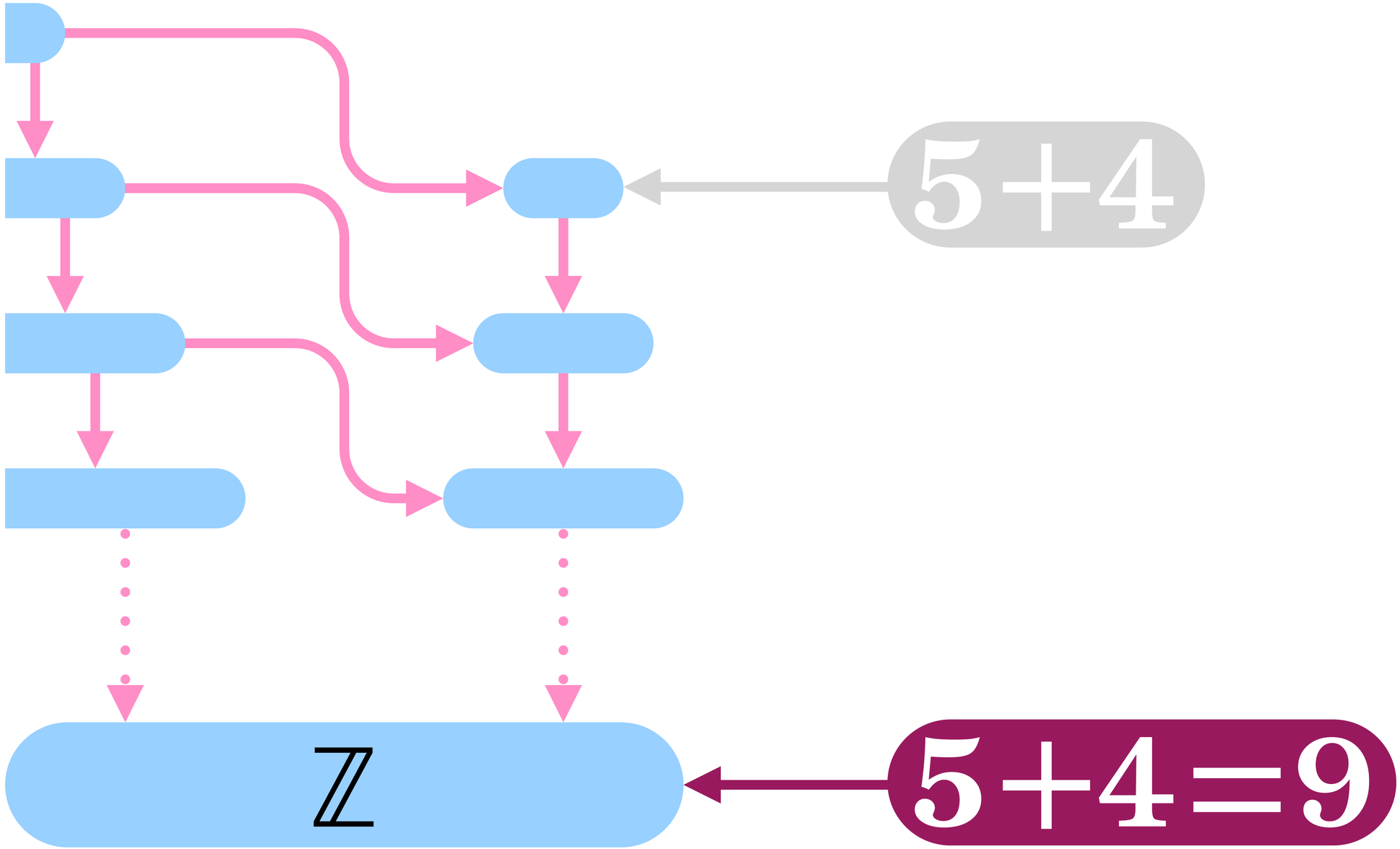

5+2
ℤ
5+2=7

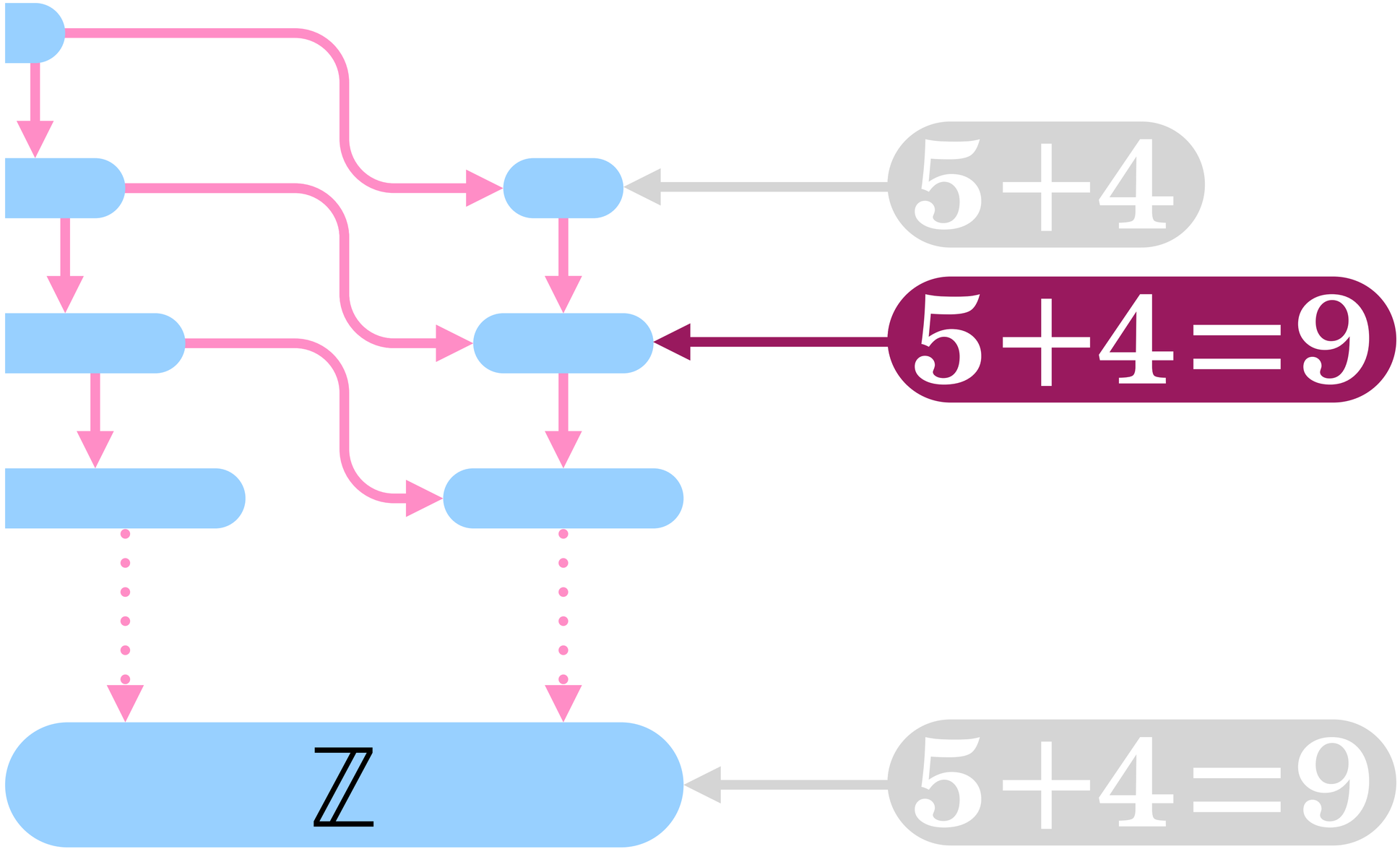

5+2=7
ℤ
5+2=7

```
const auto result_in_ℤ        = convert<ℤ>( a ) + convert<ℤ>( b );

const auto expected_result = integer_kind_of<T>.is_signed()
                                 ? convert_narrowing< T >( result_in_ℤ )
                                 : convert_modular   < T >( result_in_ℤ );

// …implementation…

claim result == expected_result;
```


5+4
ℤ


5+4
ℤ
5+4


5+4
ℤ
5+4=9


5+4
5+4=9
ℤ
5+4=9

```
template < integer_kind k > class widening;
```

The types **widening<k>** do not have:

- 🚫 implicit narrowing
- 🚫 implicit modular arithmetic
- 🚫 modular conversion on assignment
- 🚫 integral promotion
- 🚫 usual arithmetic conversions
- 🚫 compound assignment operators
- 🚫 increment or decrement operators
- 🚫 bit fields
- 🚫 character types

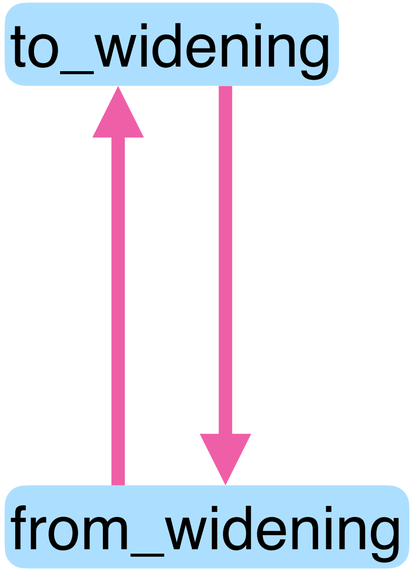
The types **widening<k>** do have:

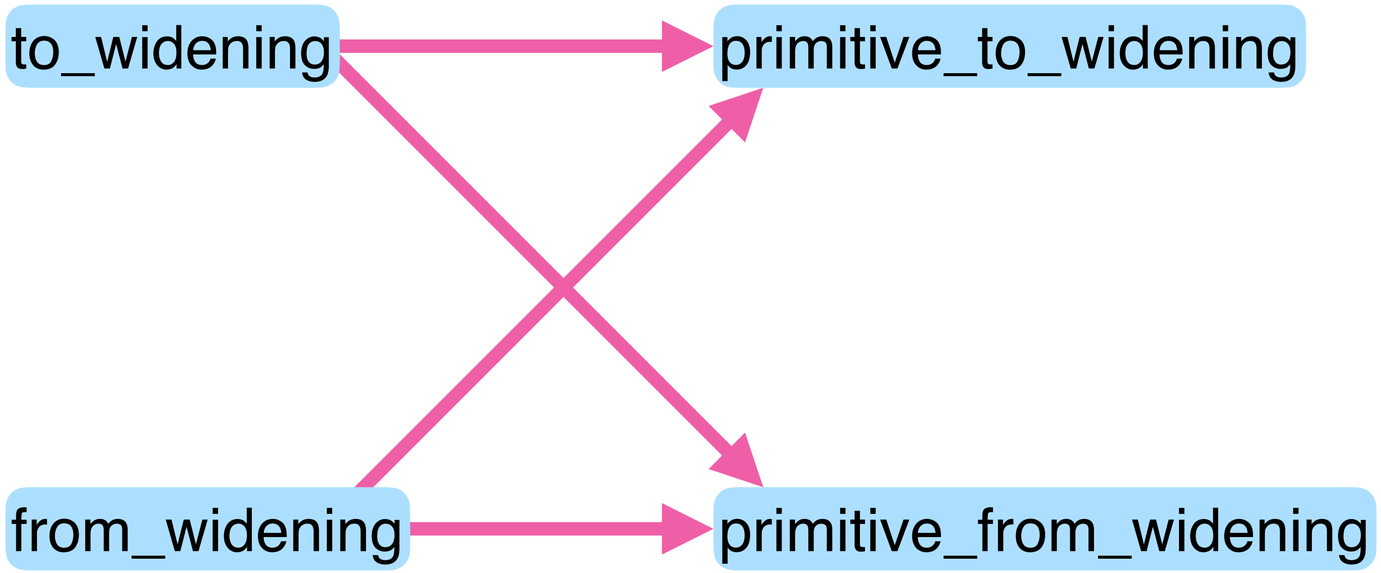
- ✅ integer kind **k**
- ✅ operations with results wider than their parameters
- ✅ operations that agree with $\mathbb{Z}$

```
inline constexpr operator+( integer_kind ak, integer_kind bk )
  {
   return join( ak, bk ) + one_bit;
  }
```

```
template < integer_kind ak, integer_kind bk >
widening<ak+bk> operator+( widening<ak>& a, widening<bk>& b );
```

```
template < std::integral T >
widening< integral_kind_of<T> >   to_widening( const T& a )
interface
   {
    // …implementation…
    claim from_widening< T >( result ) == a;
   }


template < std::integral T >
T from_widening( const widening< integral_kind_of<T> >& a )
interface
   {
    // …implementation…
    claim to_widening( result ) == a;
   }
```


to_widening
from_widening


to_widening
primitive_to_widening
from_widening
primitive_from_widening

```
template < std::integral T >
widening< integral_kind_of<T> >   to_widening( const T& a )
interface
  {
   // …implementation…
   claim result == primitive_to_widening( a );
   claim primitive_from_widening< T >( result ) == a;
  }

template < std::integral T >
T from_widening( const widening< integral_kind_of<T> >& a )
interface
  {
   // …implementation…
   claim result == primitive_from_widening<T>( a );
   claim primitive_to_widening( result ) == a;
  }
```

```
const auto result_in_ℤ        = convert<ℤ>( a ) + convert<ℤ>( b );

const auto expected_result = integer_kind_of<T>.is_signed()
                                 ? convert_narrowing< T >( result_in_ℤ )
                                 : convert_modular   < T >( result_in_ℤ );

// …implementation…

claim result == expected_result;
```

```
const auto wide_result        = to_widening( a ) + to_widening( b );

using lifted_type             = widening< integer_kind_of<T> >;

const auto lifted_result      = integer_kind_of< T >.is_signed()
                                    ? convert_narrowing< lifted_type >( wide_result )
                                    : convert_modular   < lifted_type >( wide_result );

const auto expected_result = from_widening< T >( lifted_result );

// …implementation…

claim result == expected_result;
```

-b	a + b	a & b	a == b
~b	a - b	a | b	a < b
	a * b	a ^ b	a > b
next	a / b		a <= b
prev	a % b		a >= b
			a != b
convert			a <=> b
convert_modular			
convert_narrowing			

Widening parameters

a << b

a >> b

Widening × bit_size_t

The result of the binary + operator is the sum of the operands.

7.6.6 [expr.add]

```
template < integer_kind ak, integer_kind_bk >
widening< ak + bk >   operator+(   const widening<ak>& a,
                                   const widening<bk>& b   )
interface
  {
   claim_immunity a;                                  // immunity from instability
   claim_immunity b;                                  // immunity from instability
   discern a;                                         // discernible input
   discern b;                                         // discernible input

   const auto expected_result = add_with_carry( a, b );

    implementation;

   claim_right result;                                // right of stability
   discern result;                                    // discernible output

   claim result == expected_result;                   // substitutability
  }
```

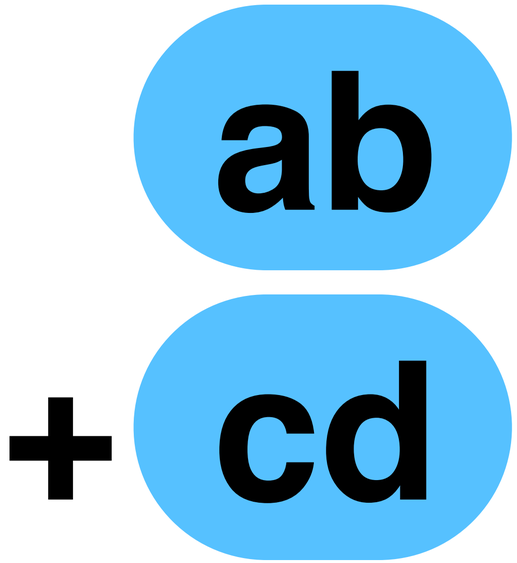
```
template < integer_kind ak, integer_kind_bk >
widening< ak + bk >   add_with_carry(   const widening<ak>& a,
                                        const widening<bk>& b,
                                        const positive_bit& carry = {}   )
interface
  {
    claim_immunity a;                                   // immunity from instability
    claim_immunity b;                                   // immunity from instability
    claim_immunity carry;                               // immunity from instability
    discern a;                                          // discernible input
    discern b;                                          // discernible input
    discern carry;                                      // discernible input

     implementation;

    claim_right result;                                 // right of stability
    discern result;                                     // discernible output



  }
```

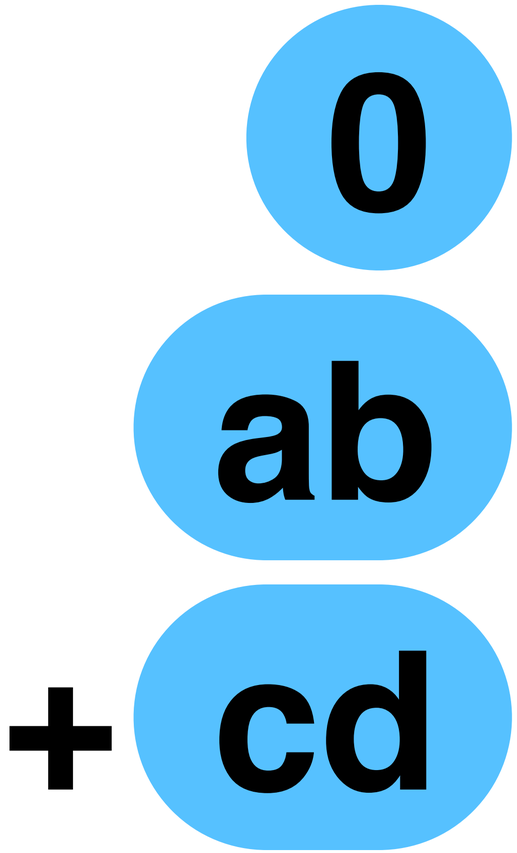
```
template < integer_kind ak, integer_kind_bk >
widening< ak + bk >   add_with_carry(   const widening<ak>& a,
                                        const widening<bk>& b,
                                        const positive_bit& carry = {}   )
interface
  {
    claim_immunity a;                                   // immunity from instability
    claim_immunity b;                                   // immunity from instability
    claim_immunity carry;                               // immunity from instability
    discern a;                                          // discernible input
    discern b;                                          // discernible input
    discern carry;                                      // discernible input

    implementation;

    claim_right result;                                 // right of stability
    discern result;                                     // discernible output

    // For a postcondition about the value of the result,
    // call reference_add_axiom( a, b, carry );
  }
```

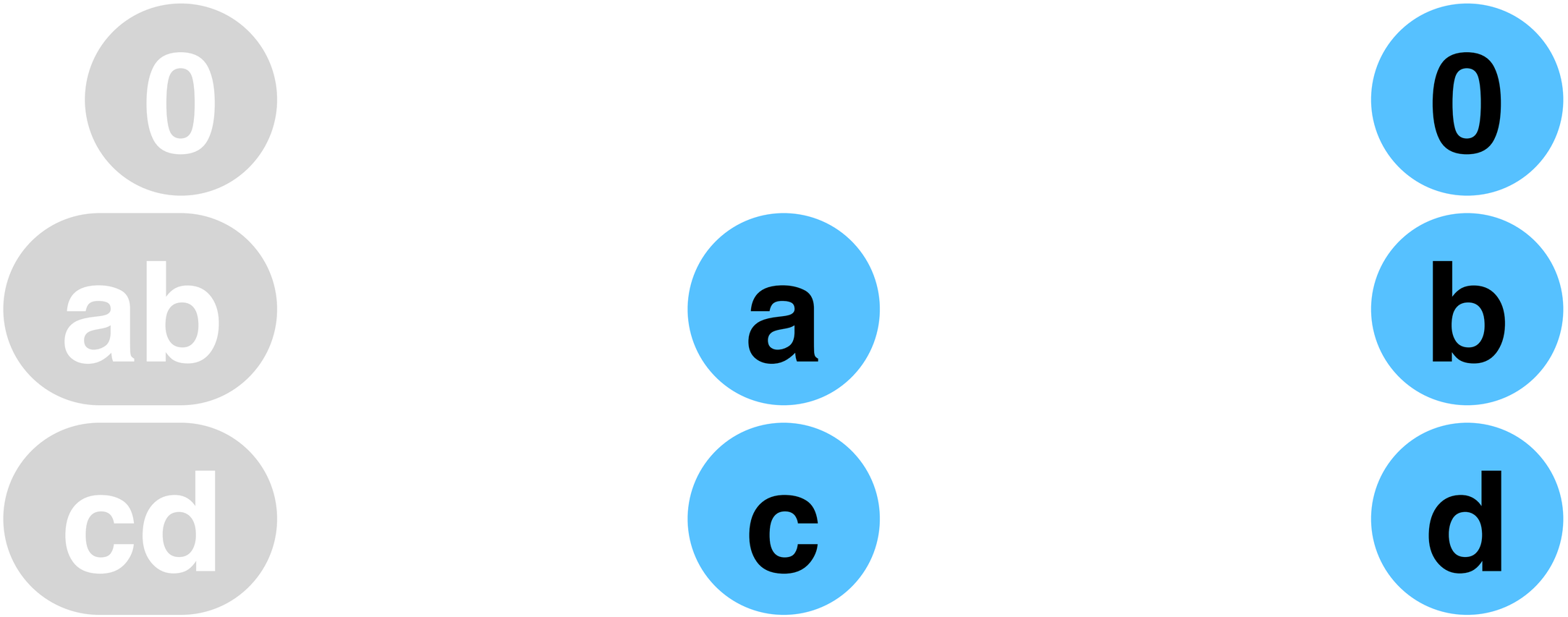
```
template < integer_kind ak, integer_kind_bk >
void reference_add_axiom( const widening<ak>& a,
                          const widening<bk>& b,
                          const positive_bit& carry = {} )
interface
  {
   const auto reference_sum = reference_add_with_carry( a, b, carry );

   posit implementation;

   const auto sum = add_with_carry( a, b, carry );

   claim substitutable( sum, reference_sum );
  }
```

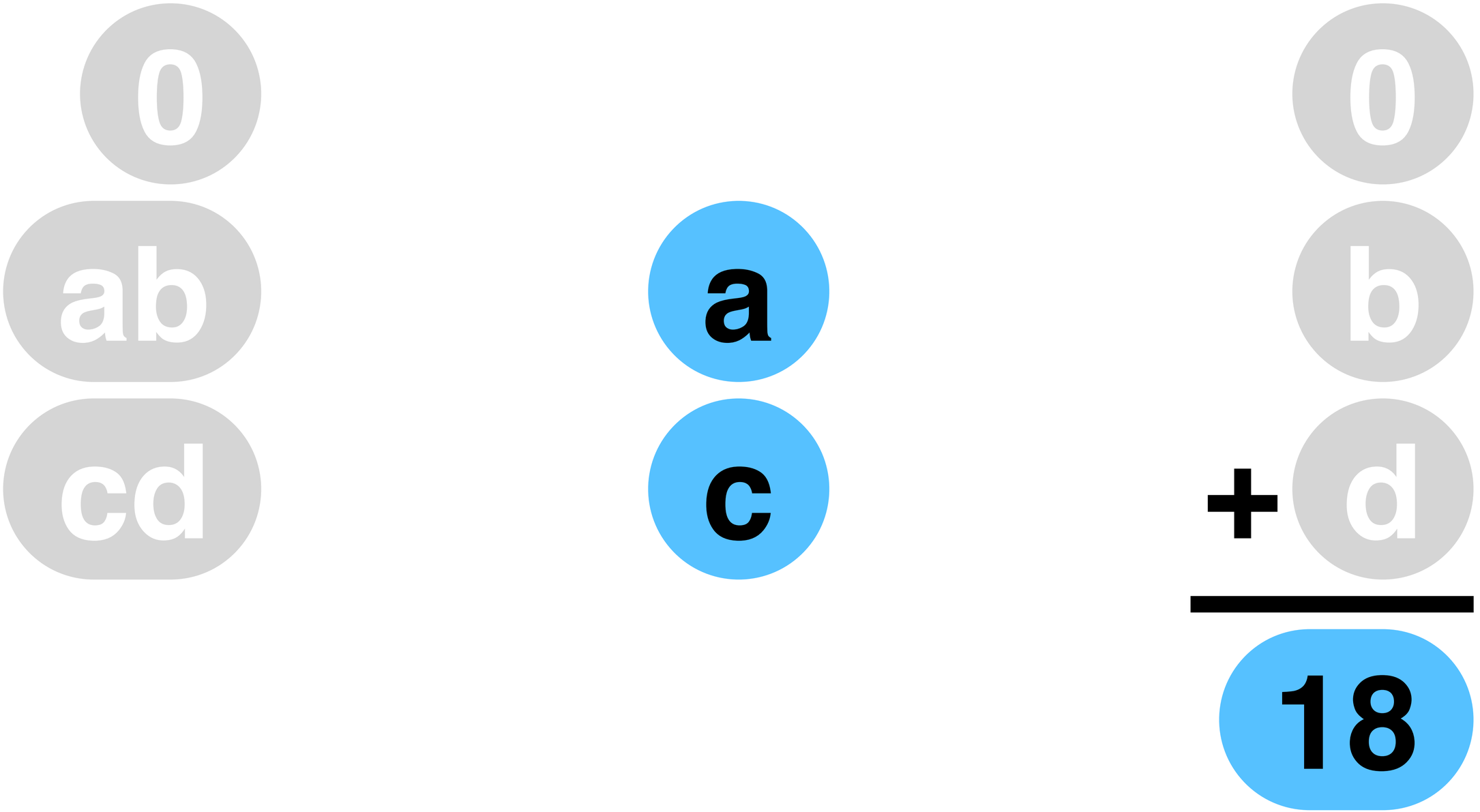
The calling function is responsible for the top part of the interface.

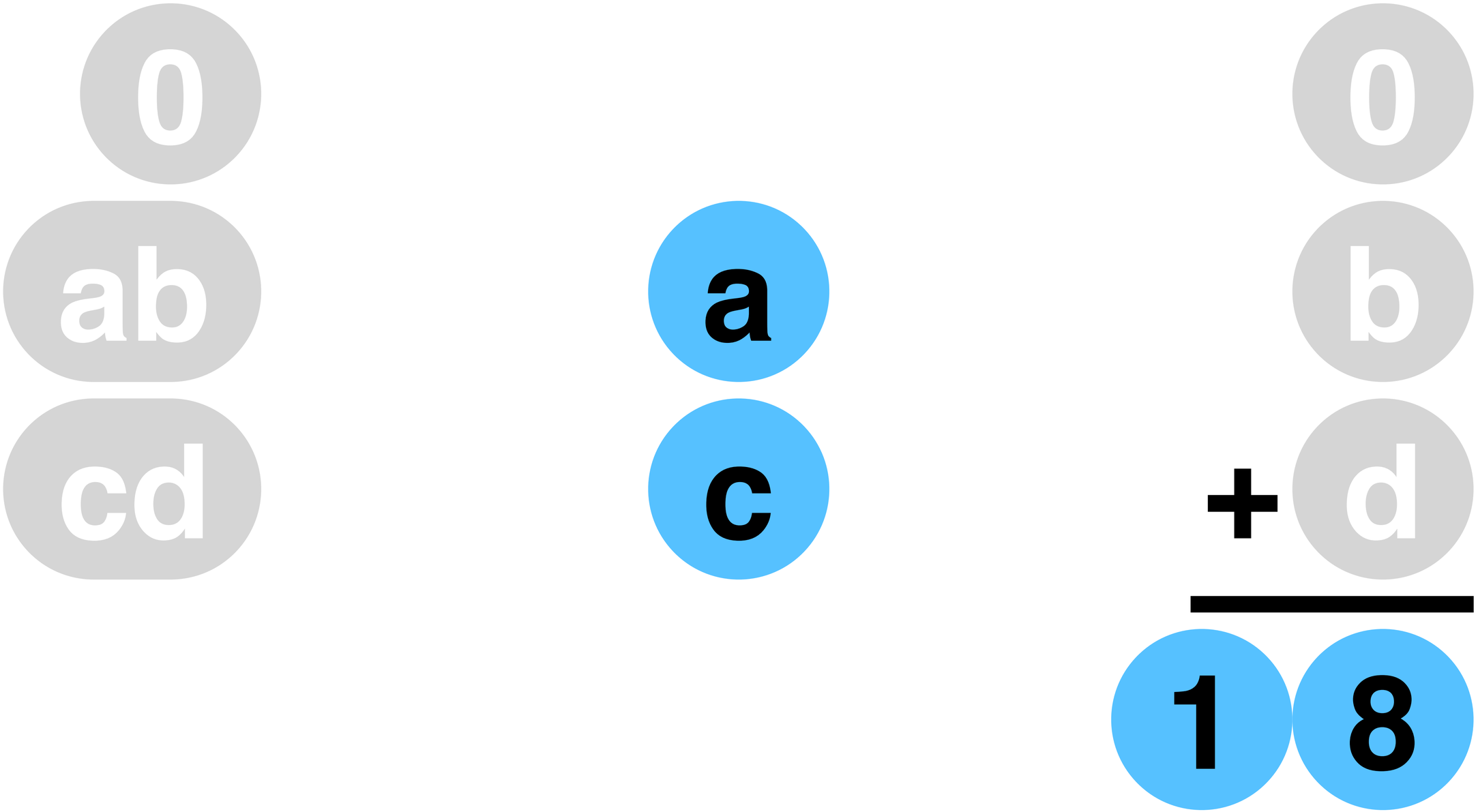
There may be no function responsible for the bottom part of the interface.

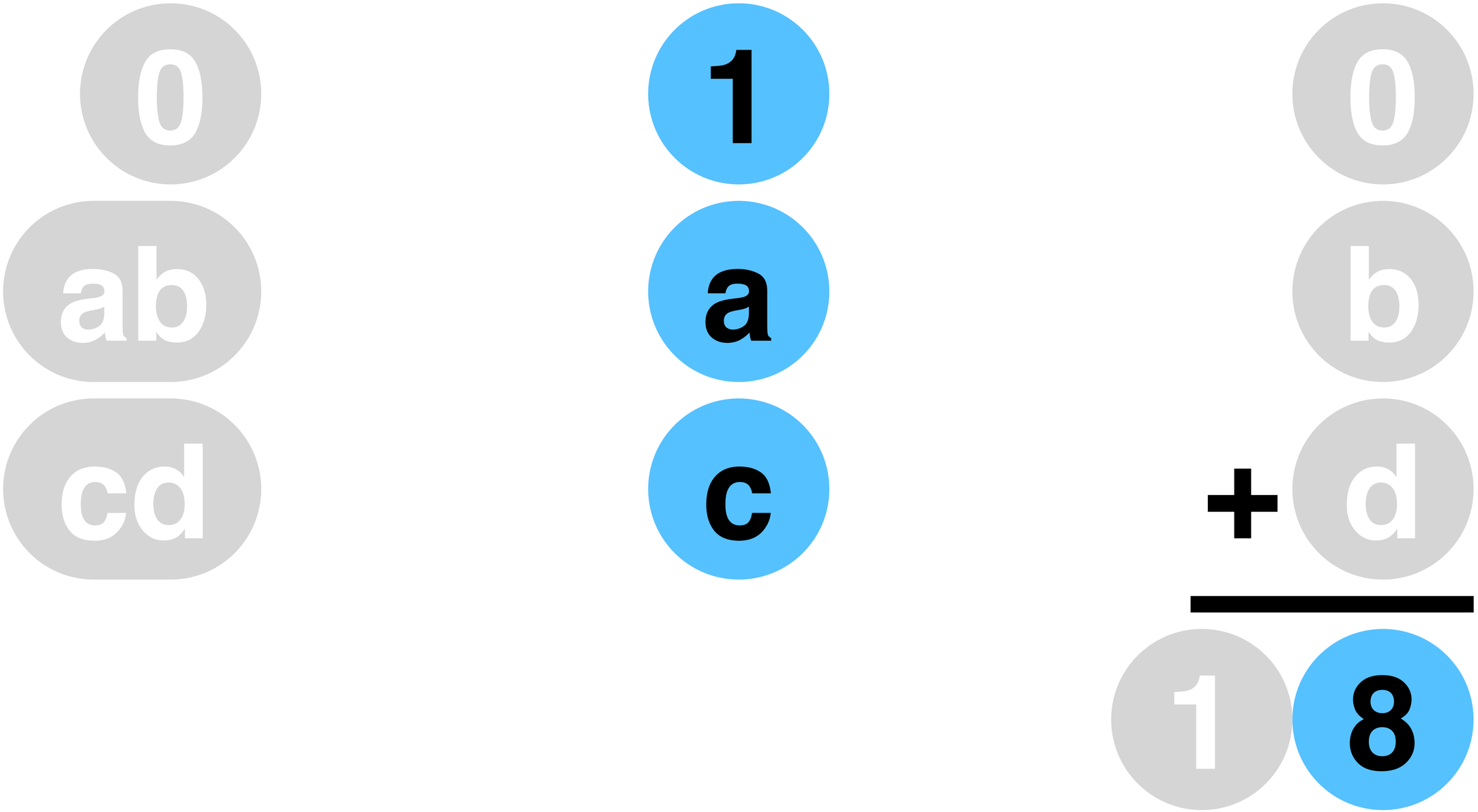

ab
+
cd

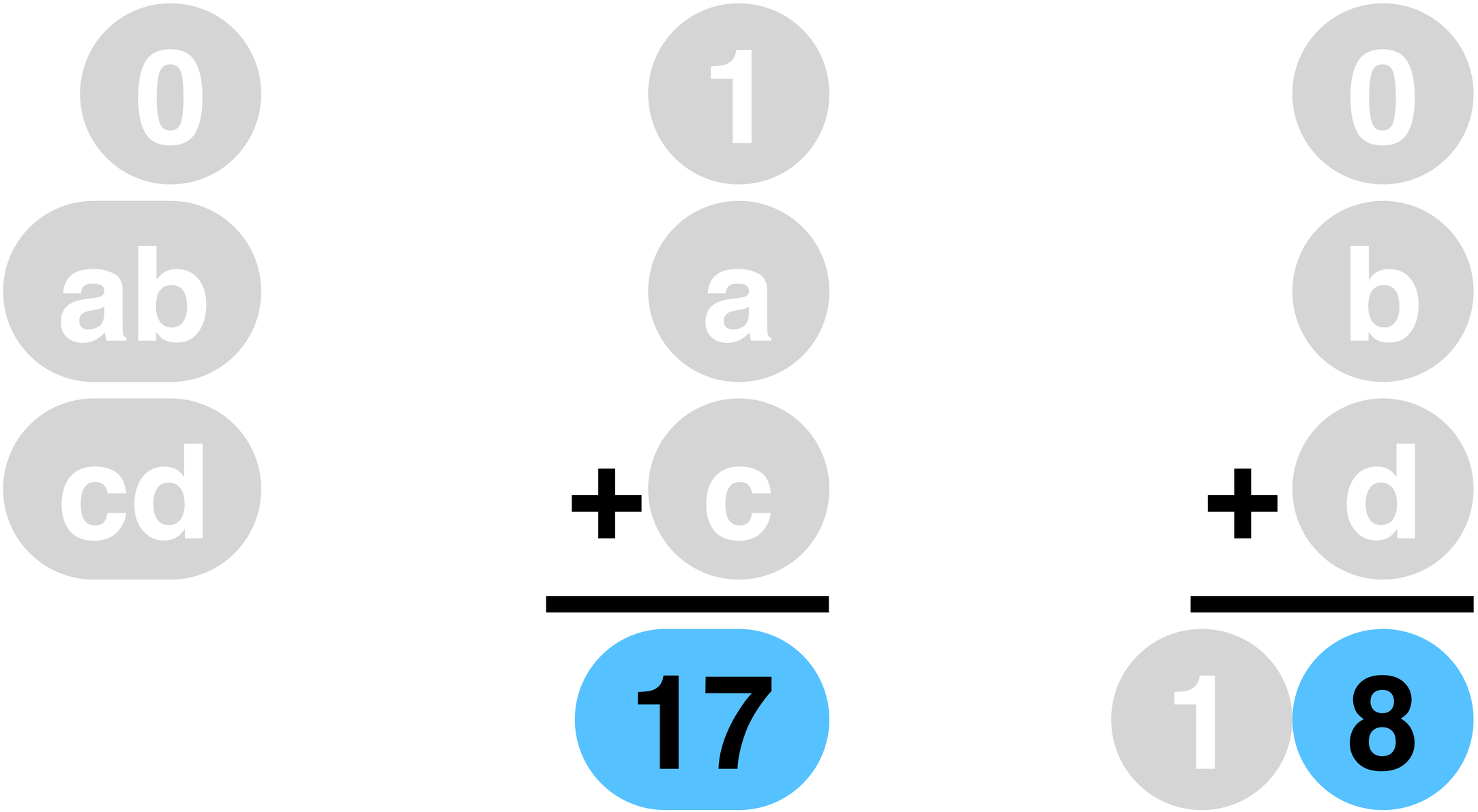

0
ab
+ cd

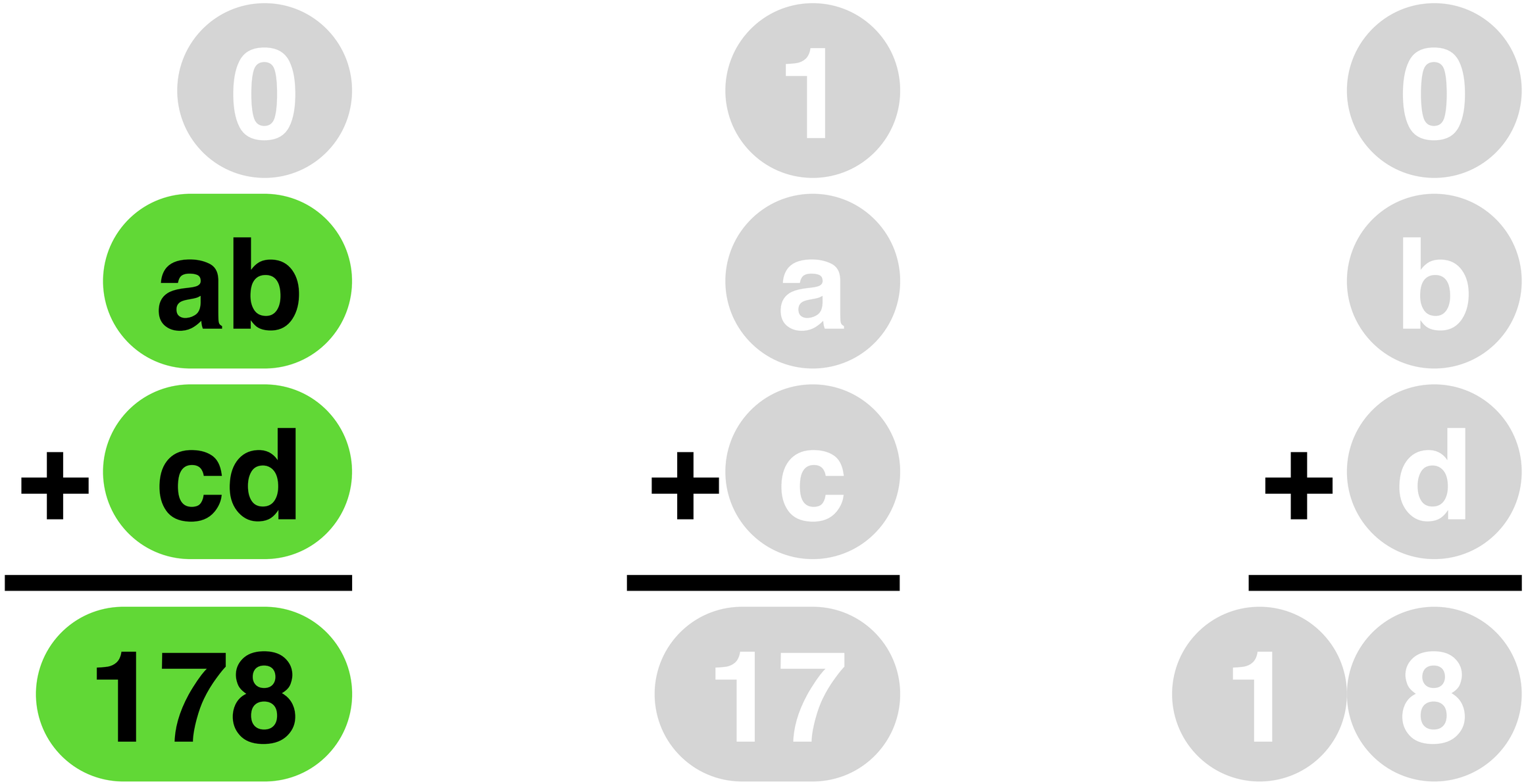

0
ab
cd
a
c
0
b
d


0
ab
cd
a
c
0
b
+
d
18


0
ab
cd
a
c
0
b
+
d
1
8

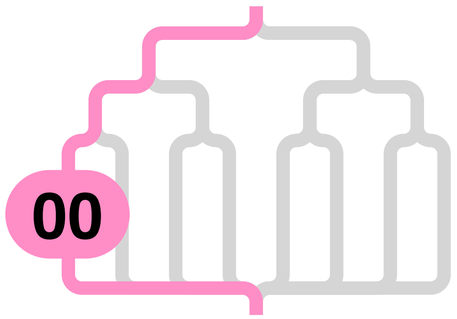

0
ab
cd
1
a
c
0
b
+ d
1 8

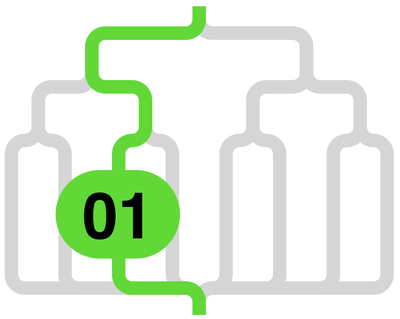

0
ab
cd
1
a
+ c
17
0
b
+ d
1 8


0
ab
+ cd
178
1
a
+ c
17
0
b
+ d
18

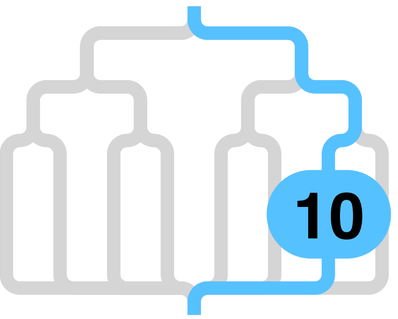
```
template < integer_kind abk, integer_kind cdk >
inline auto reference_add_with_carry( const widening<abk>& ab,
                                      const widening<cdk>& cd,
                                      const positive_bit& x0 = {} )
  {
   if constexpr ( abk.width() > 1 && cdk.width() > 1 && lo_width( abk ) == lo_width( cdk ) )
     {
      const auto [    a,      b    ] = split_bits( ab );
      const auto [    c,      d    ] = split_bits( cd );

      const auto             s0     = add_with_carry( b, d, x0 );
      const auto [  x1,      r0    ] = split_bits( s0 );

      const auto     r1              = add_with_carry( a, c, x1 );

      return join_bits( r1, r0 );
     }
   else
     // ...
```

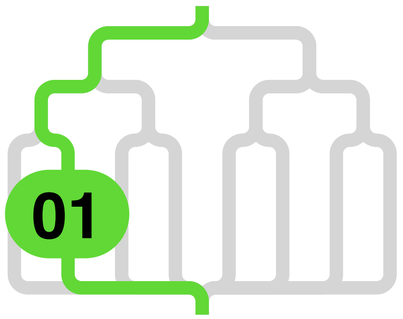
```
// ...
else if constexpr (   abk == cdk   &&   abk.width() == one_bit   &&   !abk.is_signed()  )
  {
   const auto bit_0 = bool_to_bit( false );
   const auto bit_1 = bool_to_bit( true );

   if ( bit_to_bool( ab ) )
     if ( bit_to_bool( cd ) )
       if ( bit_to_bool( x0 ) )
         return join_bits( bit_1, bit_1 );
       else
         return join_bits( bit_1, bit_0 );
     else
       if ( bit_to_bool( x0 ) )
         return join_bits( bit_1, bit_0 );
       else
         return join_bits( bit_0, bit_1 );
   else
     if ( bit_to_bool( cd ) )
       if ( bit_to_bool( x0 ) )
         return join_bits( bit_1, bit_0 );
       else
         return join_bits( bit_0, bit_1 );
     else
       if ( bit_to_bool( x0 ) )
         return join_bits( bit_0, bit_1 );
       else
         return join_bits( bit_0, bit_0 );
  }
else
  //...
```

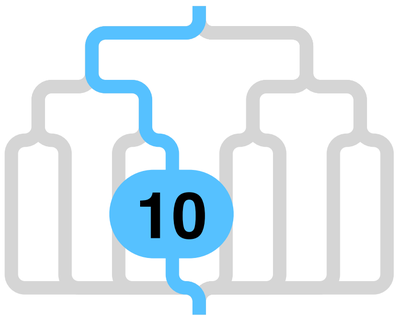

00

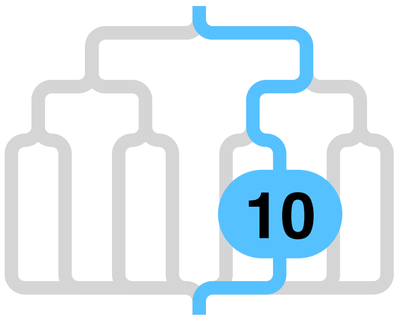

01


01


10


01


10


10


11

This is all that's left

split_bits
join_bits

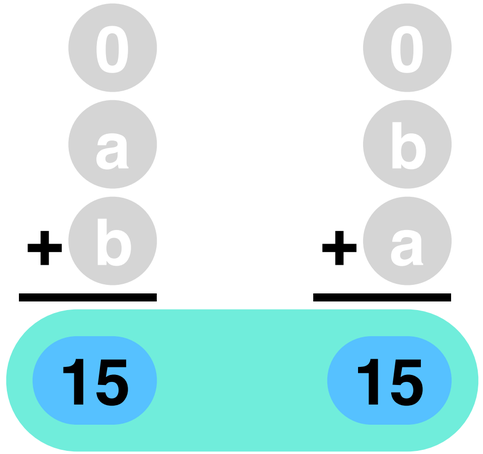
bit_to_bool
bool_to_bit

sign_bit_to_bool
bool_to_sign_bit

```
template < integer_kind ak, integer_kind bk >
void addition_is_commutative( const widening<ak>& a,
                              const widening<bk>& b,
                              const positive_bit& x = {} )
interface
  {
   const auto a_plus_b = add_with_carry( a, b, x );
   const auto b_plus_a = add_with_carry( b, a, x );

   claim implementation;

   claim substitutable( a_plus_b, b_plus_a );
  }
```

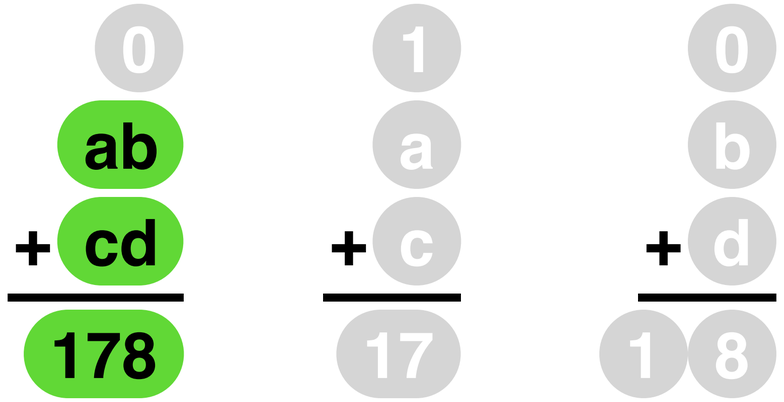
The calling function is responsible for the top part of the interface.

The called function is responsible for the bottom part of the interface.

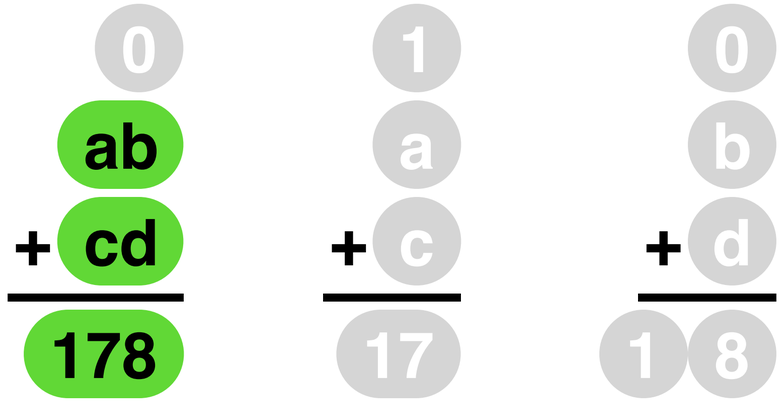


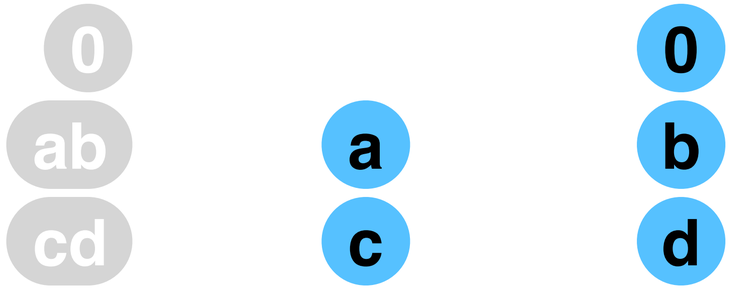

```
template < integer_kind ak, integer_kind bk >
void addition_is_commutative( const widening<ak>& a,
                              const widening<bk>& b,
                              const positive_bit& x = {} )
interface
  {
   const auto a_plus_b = add_with_carry( a, b, x );
   const auto b_plus_a = add_with_carry( b, a, x );



   claim implementation;



   claim substitutable( a_plus_b, b_plus_a );
  }
```

reference_add_axiom(**ab**, **cd**, **0**);

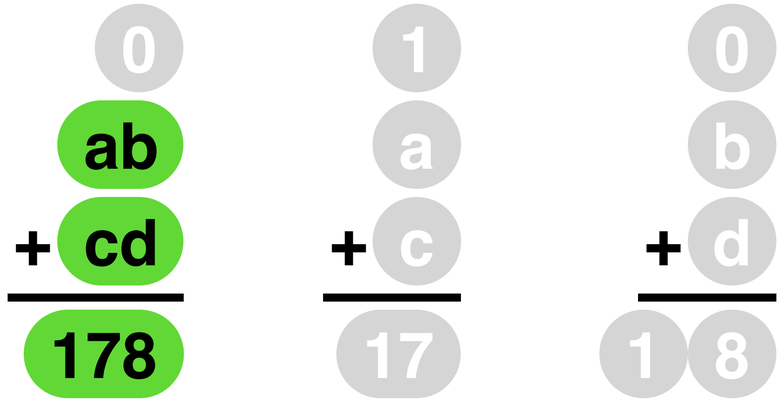


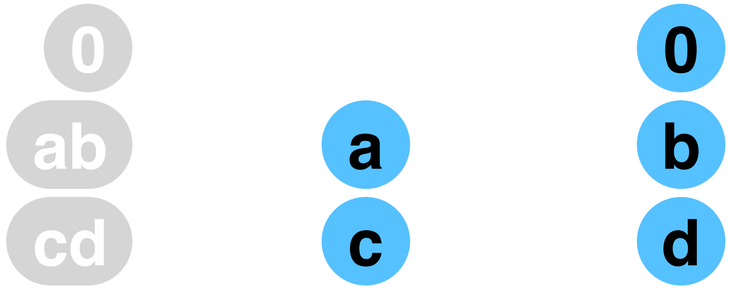
reference_add_axiom(**ab**, **cd**, **0**);



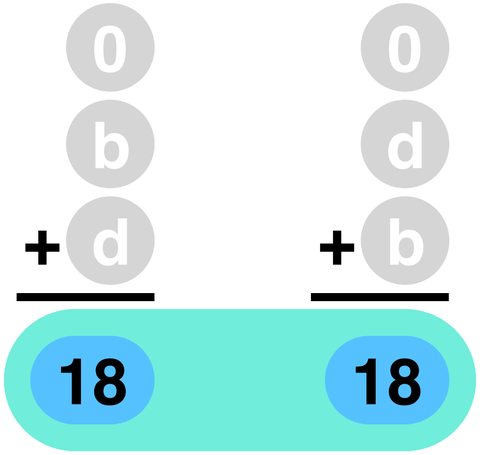


reference_add_axiom(**ab**, **cd**, **0**);

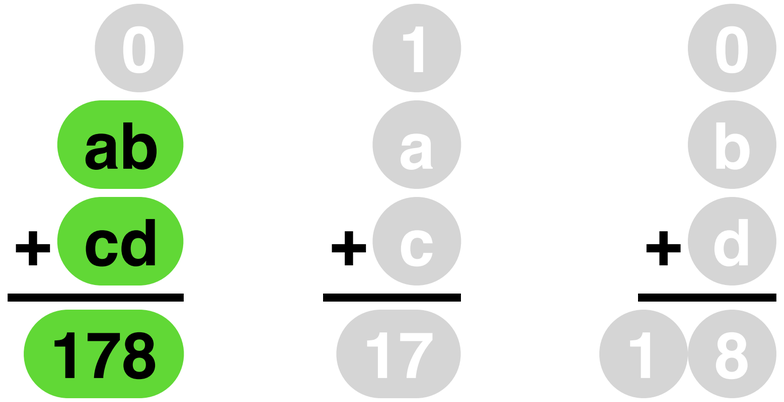


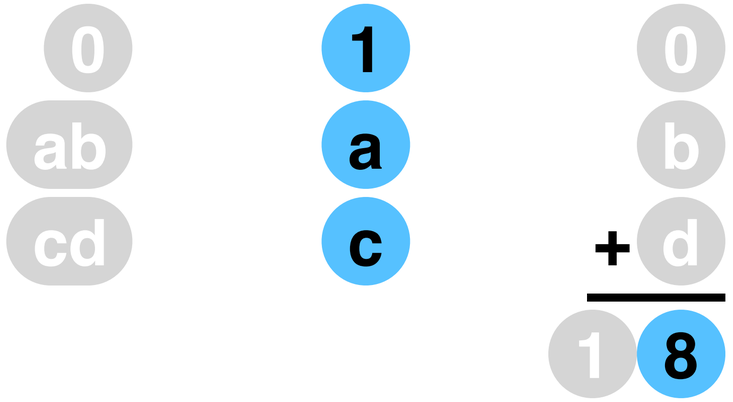


reference_add_axiom(**ab**, **cd**, **0**);

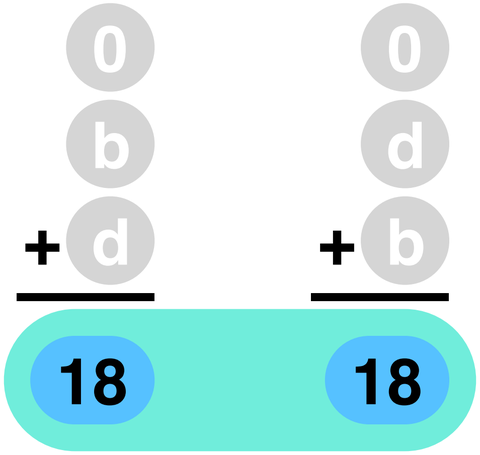


addition_is_commutative(**b**, **d**, **0**);

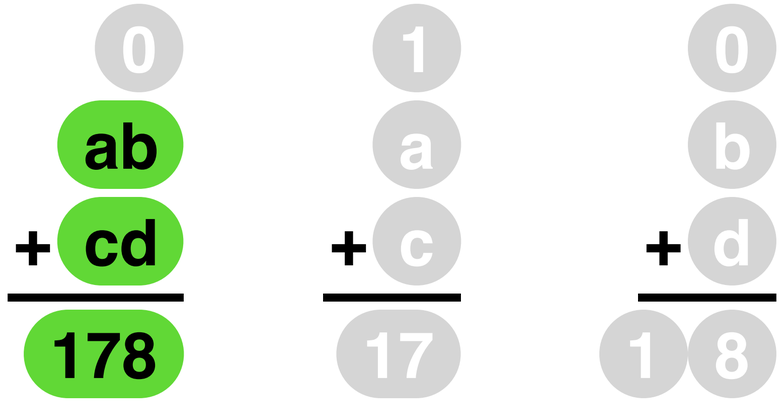




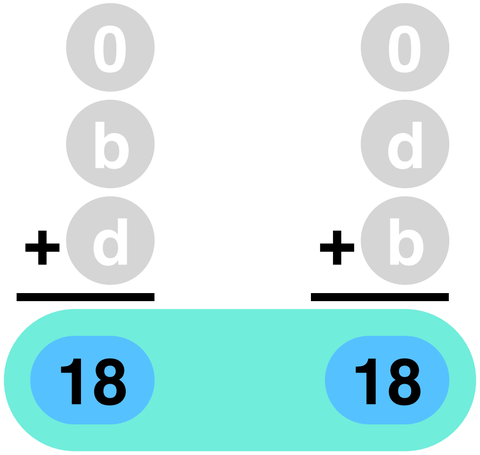
reference_add_axiom(**ab**, **cd**, **0**);



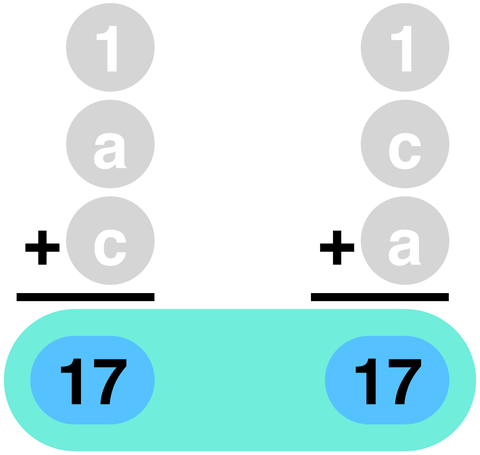
addition_is_commutative(**b**, **d**, **0**);



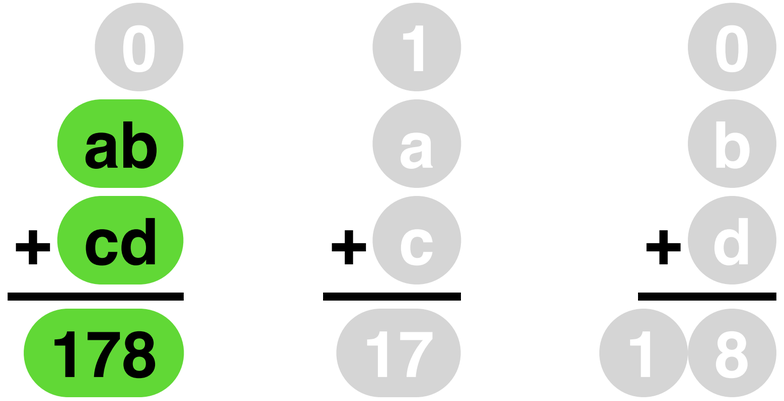
reference_add_axiom(**ab**, **cd**, **0**);



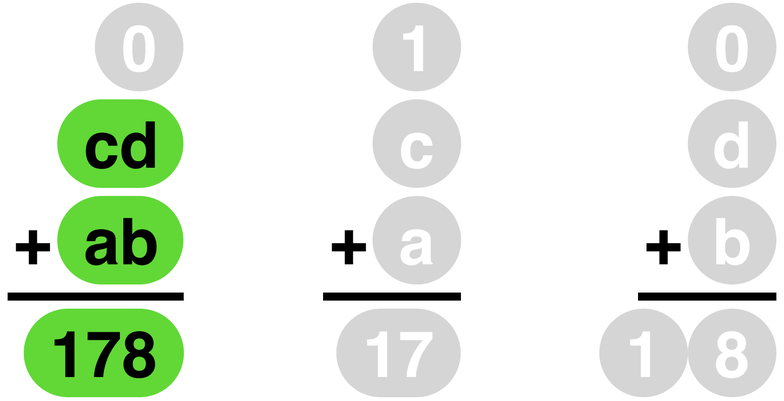
addition_is_commutative(**b**, **d**, **0**);



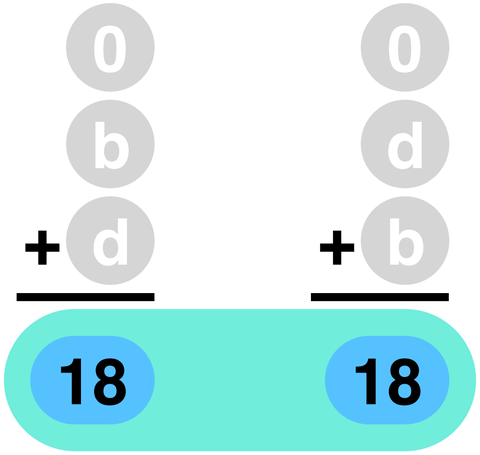
addition_is_commutative(**a**, **c**, **1**);



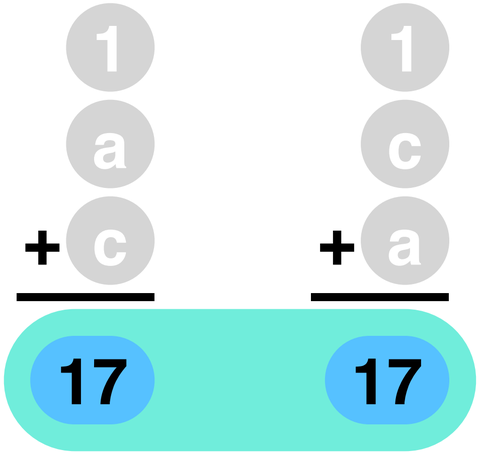
reference_add_axiom(**ab**, **cd**, **0**);



reference_add_axiom(**cd**, **ab**, **0**);

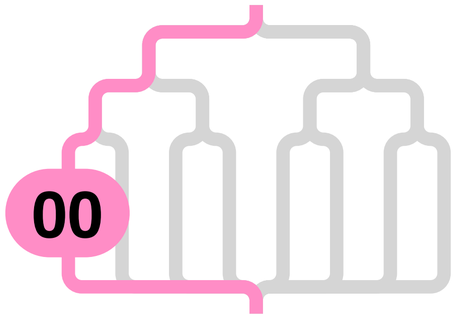


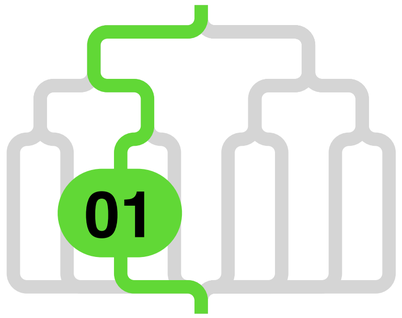
addition_is_commutative(**b**, **d**, **0**);



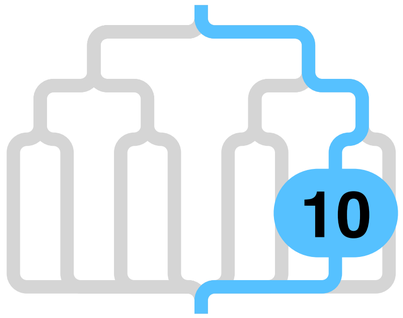
addition_is_commutative(**a**, **c**, **1**);

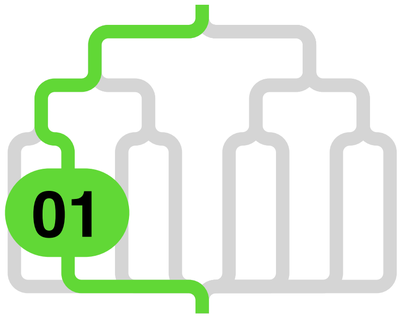
```
template < integer_kind abk, integer_kind cdk >
void addition_is_commutative( const widening<abk>& ab, const widening<cdk>& cd,
                                        const positive_bit& x0 = {} )
implementation
  {
   if constexpr ( abk.width() > 1 && cdk.width() > 1 &&  lo_width( abk ) == lo_width( cdk ) )
     {
      reference_add_axiom( ab, cd, x0 );

      const auto [     a,     b     ]  = split_bits( ab );
      const auto [     c,     d     ]  = split_bits( cd );
      addition_is_commutative( b, d, x0 );

      const auto             s0       = add_with_carry( b, d, x0 );
      const auto [   x1,    r0     ]  = split_bits( s0 );
      addition_is_commutative( a, c, x1 );

      reference_add_axiom( cd, ab, x0 );
     }
  else // ...
```

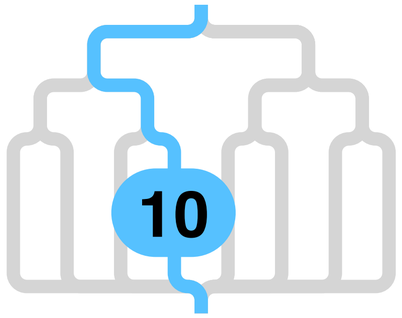

00

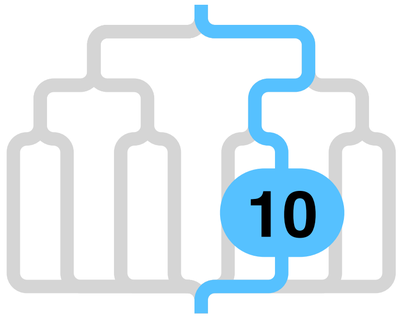

01


01

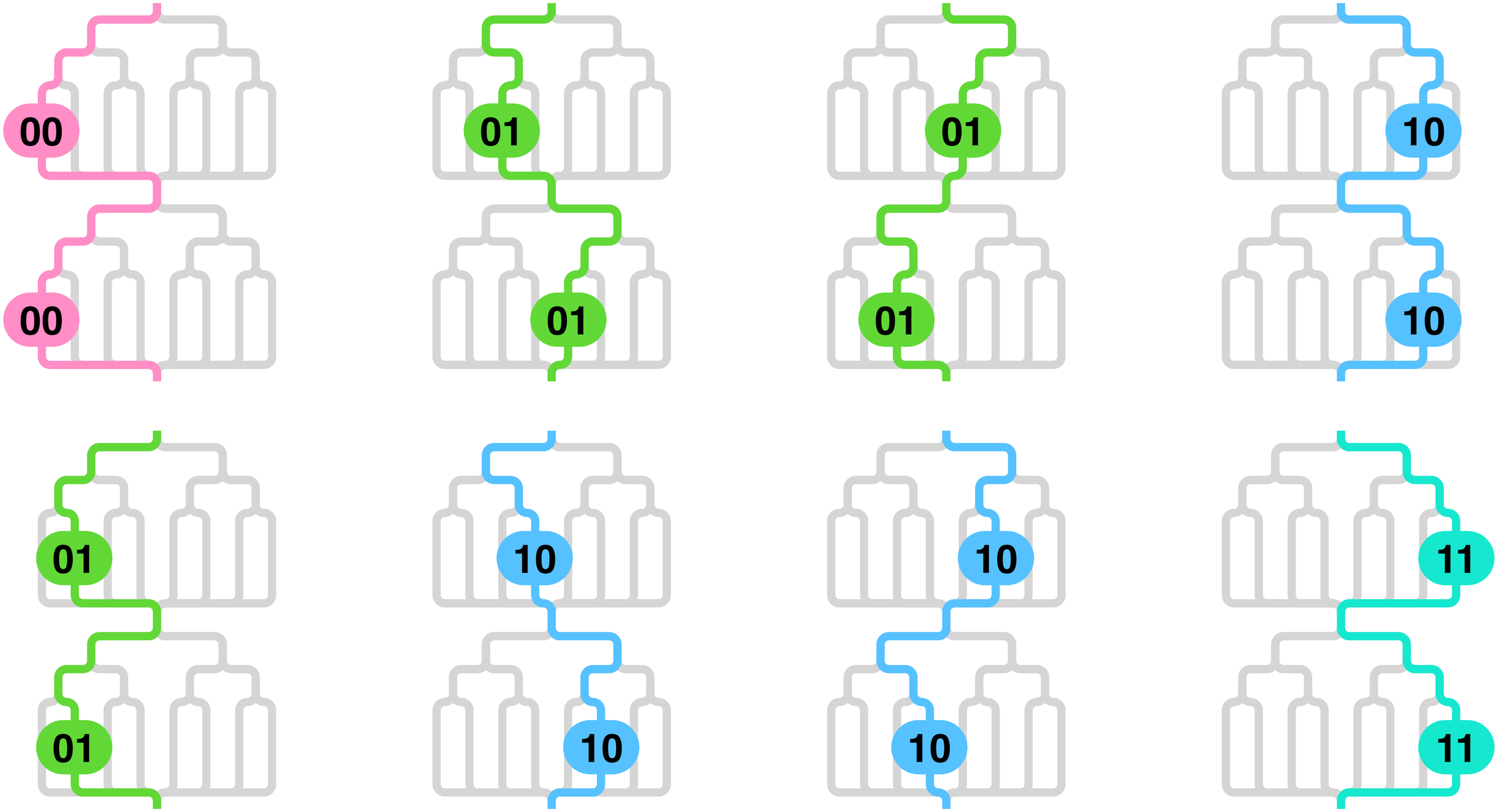

10


01

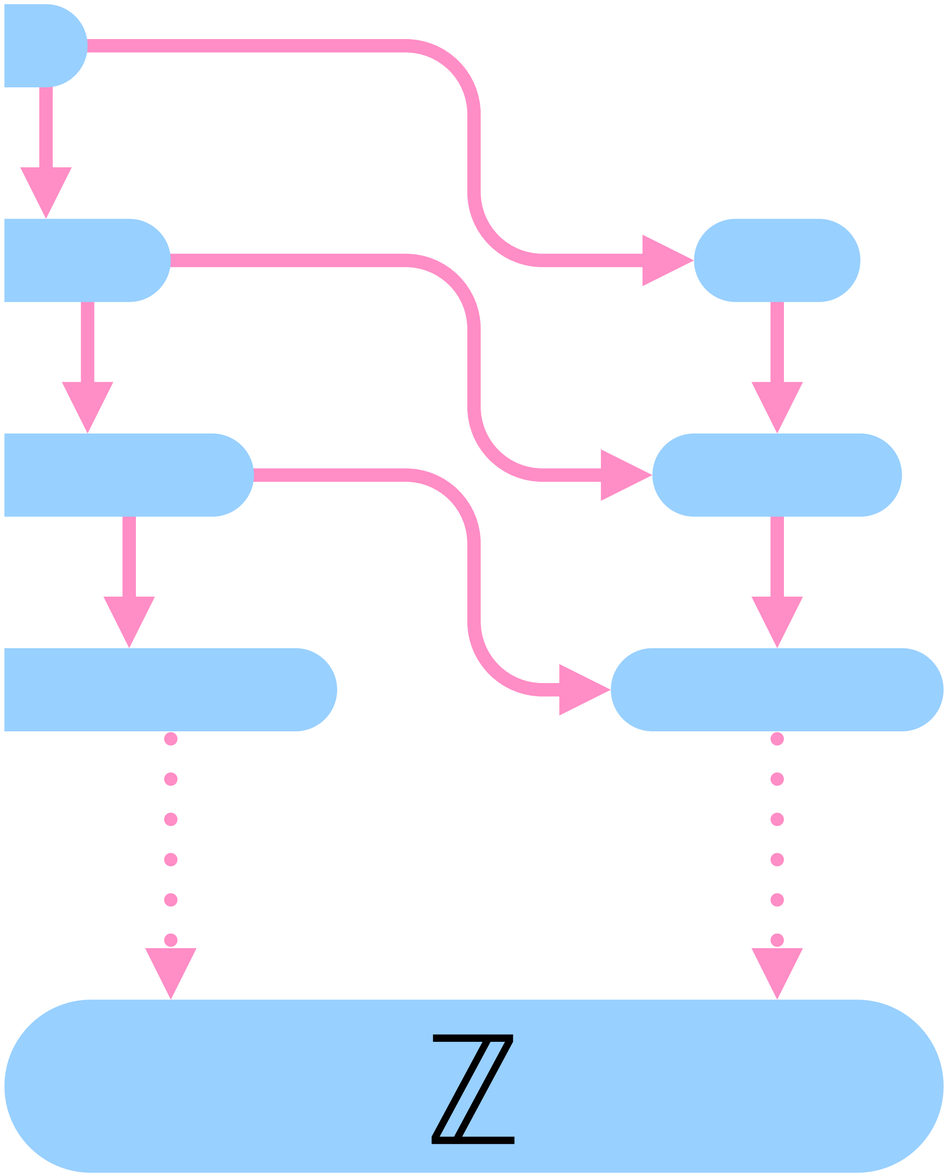

10

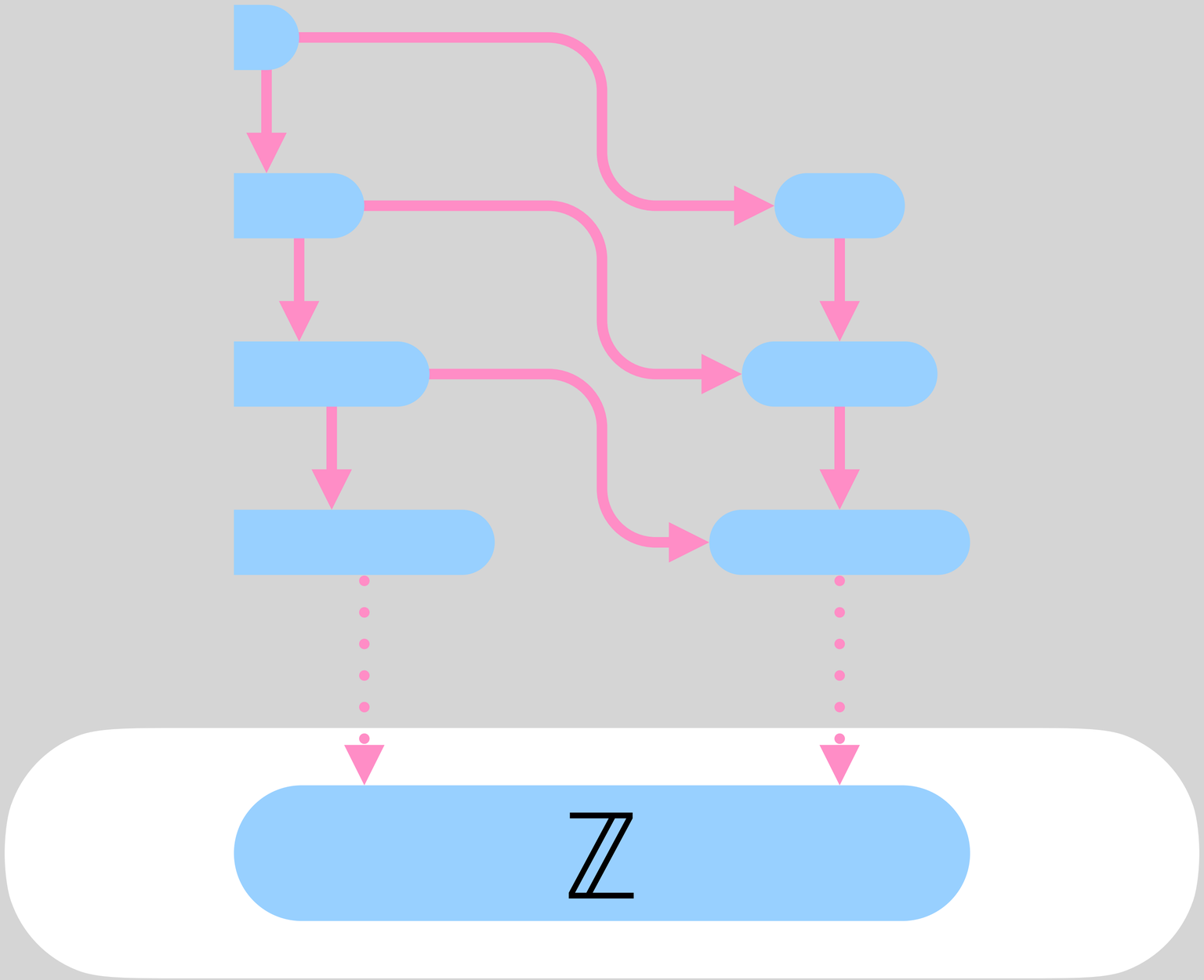

10

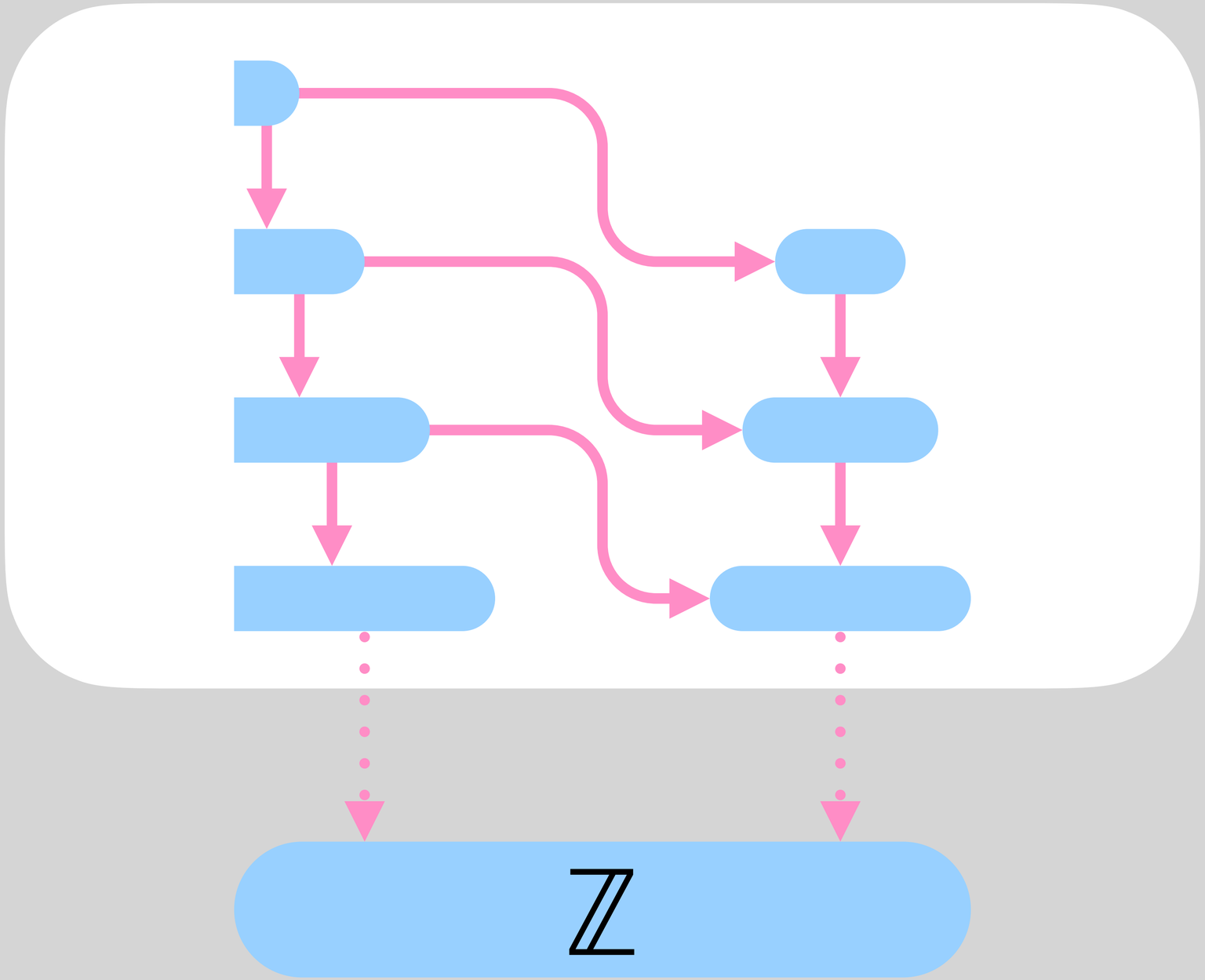

11


00
00
01
01
01
01
10
10
01
01
10
10
11
11

```
// …
else if constexpr (  abk == cdk   &&   abk.width() == one_bit   &&   !abk.is_signed()  )
  {
   reference_add_axiom( ab, cd, x0 );

   reference_add_axiom( cd, ab, x0 );
  }
else // …
```


ℤ


ℤ


ℤ

Thank you for listening.

Questions?