Adding a Fundamental Type for N-bit integers

Committee: ISO/IEC JTC1 SC22 WG14

Document Number: N2590
Date: 2020-10-30
Revises document number: N2534
Authors: Aaron Ballman, Melanie Blower, Tommy Hoffner, Erich Keane
Reply to: Aaron.Ballman@intel.com, Melanie.Blower@intel.com,
Tommy.Hoffner@intel.com, Erich.Keane@intel.com

Contents

Summary of Changes

N2590

- Significantly revised the proposed wording.
- Replaced previous inttypes.h macros with the %qN length modifier for I/O support.
- Removed the _STR_TO_EXTINT macro and supported oversized constants directly.
- Made the _Bitwidthof operator apply in constant expressions and apply to bit-fields.
- Added a helper macro for querying lock-free atomic support and a type mapping to an atomic type name.
- Added a section about impacts on generic programming.
- Added more rationale and prior art information.
- Exempted _ExtInt from impacting the definition of [u]intmax_t and the exact-width integer types.

N2534

- Proposed wording

N2501

- High level introduction to the topic without proposed wording, with corrections to the original document.

N2472

- Original report.

Introduction and Rationale

We propose adding a family of integer types spelled as _ExtInt(N), where N is an integral constant expression representing the exact number of bits to be used to represent the type. In most application code, the usual 8-, 16-, 32-, 64-bit width integer types provide satisfactory expressiveness for common integer uses. However, there are use cases for integer types with a specific bit-width in application domains, such as using 256-bit integer values in various cryptographic symmetric ciphers like AES, when calculating SHA-256 hashes, representing a 24-bit color space, or when describing the layout of network or serial protocols.

Further, in the case of FPGA hardware, using normal integer types for small value ranges where the full bit-width isn't used is extremely wasteful and creates severe performance/space concerns. At the other extreme, FPGA's can support really wide integers, essentially providing arbitrary precision, and existing FPGA applications make use of really large integers, for example up to 2031 bits. The clang implementation of _ExtInt provides support for bit widths up to 10^{24} .

An integer type with an explicit bit-width allows programmers to portably express their intent in code more clearly. Today, the programmer must pick an integer datatype of the next larger size and manually perform mask and shifting operations. However, this is error prone because integer widths vary from platform to platform and exact-width types from stdint.h are optional, it is not always obvious from code inspection when a mask or a shift operation is incorrect, implicit conversion can lead to surprising behavior, and the resulting code is harder for static analyzers and optimizers to reason about. By allowing the programmer to express the specific bit-width they need for their problem domain directly from the type system, the programmer can write less code that is more likely to be correct and easier for tooling and humans to reason about.

Proposed solution

A set of bit-precise integer types using the syntax _ExtInt(N) where N is an integer that specifies the number of bits that are used to represent the type, including the sign bit. The keyword _ExtInt followed by a parenthesized integer constant is a type specifier, thus it can be used in any place a type can; whether it can be the type of a bit-field is implementation defined. When polled on whether WG14 would like something along these lines at the Oct 2020 meeting, the results were 16/1/4 (consensus).

We are not proposing this feature as an annex because we believe there is sufficient motivation for a mandatory, portable, exact bit-width integer datatype. Significant prior art exists for such a datatype and demonstrates that the industry sees a need in this space. Specifying a portable, first-class, bit-

precise integer data type is best accomplished by specifying the data type directly in the type system rather than as additional functionality specified via an annex.

An _ExtInt can be explicitly declared as either signed or unsigned by using the signed/unsigned keywords. If no sign specifier is used or if the signed keyword is used, the _ExtInt type is a signed integer.

The N expression is an integer constant expression, which specifies the number of bits used to represent the type, following normal integer representations for both signed and unsigned types. Both a signed and unsigned _ExtInt of the same N value will have the same number of bits in its representation. Many architectures don't have a way of representing non power-of-2 integers, but these architectures can emulate these types using larger integers.

In order to be consistent with the C language and make the _ExtInt types useful for their intended purpose, _ExtInt types follow the usual C standard integer conversion ranks. An _ExtInt type has a greater rank than any integer type with less precision, or any implementation defined integer type with the same precision. However, they have lower rank than any of the standard integer types with the same precision, which means that a binary expression with a 32-bit `int` and an `_ExtInt(32)` will use `int` as a the common type. (cf 6.3.1.1 “The rank of any standard integer type shall be greater than the rank of any extended integer type with the same width.”) Usual arithmetic conversions also work the same, where the smaller ranked integer is converted to the larger. When polled on whether the committee agrees with the chosen integer ranks at the Oct 2020 meeting, the results were 7/2/10 (weak consensus).

There is one crucial exception to the C rules for integer promotion: _ExtInt types are excepted from the integer promotions. Operators typically will promote operands smaller than the width of an int to an int. Doing these promotions would inflate the size of required hardware on some platforms, so _ExtInt types aren't subject to the integer promotion rules. For example, in a binary expression involving an _ExtInt(12) and an unsigned _ExtInt(3), the usual arithmetic conversions would not promote either operand to an int before determining the common type. Because one type is signed and one is unsigned and because the signed type has greater rank than the unsigned type (due to the bit-widths of the types), the unsigned _ExtInt(3) will be converted to _ExtInt(12) as the common type. We do not propose promoting to an _ExtInt wide enough to perform the operation without loss of precision because empirical evidence suggests that mildly complex arithmetic expressions can quickly cause promotion to surprisingly wide _ExtInt types with poor performance characteristics. Further, integer promotions do not always yield intuitive results. While a case like promoting an expression SomeExtInt12 * SomeExtInt8 to use ints in order to avoid overflow may seem attractive, a case like SomeExtInt28 * SomeExtInt8 where integer promotions would promote to a (potentially) 32-bit type that’s still insufficient to represent the resulting value shows that this implicit conversion doesn't always help. By exempting _ExtInt from the integer promotion rules, users are given a more expressive model that allows them to express the bit-precise semantics of expressions in a way that is easier for tooling like static analyzers to reason about. When polled on whether the committee is in favor of excepting _ExtInt from the integer promotions, the results were 14/0/6 (consensus).

We also propose adding the _Bitwidthof unary operator to provide the programmer a convenient and reliable way to access the width of a _ExtInt typed expression. The value could be used to declare other objects, or it could be used as the argument to the formatted i/o macros. It would also be useful in

compiler self-tests, for example static_assert(_Bitwidthof(7xi) == 4). Given that the bit-width of any complete type used in an expression must be known statically at compile time, this operator can be applied to any integer type and to a bit-field.

To support formatted input and output, we propose a new length modifier, q, which must be followed by an integer constant N to describe that the corresponding argument is an _ExtInt of width N. The signedness of the argument's type is inferred from the conversion specifier. A new format specifier for the _ExtInt type is required because _ExtInt does not get converted during the default argument promotions, and so the exact type and width are required to call va_arg to interpret the _ExtInt(N) value.

To support forming _ExtInt literals, we propose adding a new integer literal suffix spelled `xi` which designates a constant of type _ExtInt(N) where N is calculated based on the given literal, plus an extra bit for the sign bit unless the u suffix is also used. For instance: `5xi` would create a constant of type `_ExtInt(4)` (three value bits and one sign bit) and `5uxi` would create a constant of type `unsigned _ExtInt(3)`. Within the preprocessor, if the constant value is too large to fit within the range of values supported by [u]intmax_t, the constant cannot appear within the controlling expression of a #if directive (6.10.1p7) but it will still form a valid integer constant suitable for use within an expression or initialization (6.4.8p4, 6.4.4p3). This can lead to a subtle surprise with code like:

```
#define FOO 0xFFFF…FFFFuxi // … is replaced by a lot of hex digits

_ExtInt(…) i = FOO; // OK (… is replaced by the # of expected bits)

#if FOO // invalid constant expression; value too large for uintmax_t
#endif
```

When polled about whether the committee is in favor of supporting integer literals of _ExtInt type using the xi suffix at the Oct 2020 meeting, the results were 13/4/3 (consensus).

Prior Art

Intel has introduced this functionality in our FPGA targeting compilers; the High Level Synthesis (HLS) compiler and FPGA OpenCL compiler under the name "Arbitrary Precision Integer" (ap_int for short). See References section below for details about these compilers. This feature has been extremely useful and effective for our users, permitting them to optimize their storage and operation space on an architecture where both can be extremely expensive. The Intel Compilers for the Intel FPGA have many users that rely on the ap_int type to achieve efficient programs on a daily basis.

The ac_int type is another demonstration of prior art as a de facto standard for bit-precise integer types in C++ and was originally developed by Mentor Graphics: https://github.com/hlslibs/ac_types/blob/master/pdfdocs/ac_datatypes_ref.pdf.

Another implementation of this feature, implemented from scratch, will also be available in Intel's oneAPI product, currently in beta test: https://software.intel.com/en-us/oneapi.

_ExtInt has been implemented directly in the Clang 11 release as a language extension: https://clang.llvm.org/docs/LanguageExtensions.html#extended-integer-types.

The XiLinx FPGA compiler for HLS also provides users a similar solution: a C++ “arbitrary precision” integer type so that solutions can be optimized. Note that the naming scheme for C types builds the integer width into the type name, a la int9, and for XiLinx the maximum width is limited to 1024 bits.

Prior art exists for the proposed _Bitwidthof operator in the form of the bitWidth type property for integer types from the Swift programming language: https://developer.apple.com/documentation/swift/int/2885648-bitwidth which can be queried to determine the number of bits (not bytes) used as the underlying binary representation for integer types.

Prior art for the proposed %qN length modifier exists in part from the %I64 and %I32 length modifiers supported by Microsoft Visual Studio (https://docs.microsoft.com/en-us/cpp/c-runtime-library/format-specification-syntax-printf-and-wprintf-functions?view=vs-2019) and the PRI and SCN macros provided for specific bit-width types in inttypes.h.

ABI Considerations

_ExtInt(N) types align with existing calling conventions. They have the same size and alignment as the smallest basic type that can contain them. Types that are larger than __int64_t are conceptually treated as struct of register size chunks. The number of chunks is the smallest number that can contain the type.

On Intel64 platforms, _ExtInt types are bit-aligned to the next greatest power-of-2 up to 64 bits: the bit alignment A is min(64, next power-of-2(>=N)). The size of these types is the smallest multiple of the alignment greater than or equal to N. Formally, let M be the smallest integer such that A*M >= N. The size of these types for the purposes of layout and sizeof is the number of bits aligned to this calculated alignment, A*M. This permits the use of these types in allocated arrays using the common sizeof(Array)/sizeof(ElementType) pattern.

Safety

Overflow occurs when a value exceeds the allowable range of a given data type. For example, `(_ExtInt(3))7 + (_ExtInt(3))2` overflows, and the result is implementation-defined as with other signed integer types. To avoid the overflow, the operation type can be widened to 4 bits by casting one of the operands to _ExtInt(4). As with other unsigned integer types, overflow of an unsigned _ExtInt is well-defined and the value wraps around with typical twos complement semantics.

Interaction with _Generic

_ExtInt(N) is a distinct type from _ExtInt(M) when N != M, which means that use within a generic selection expression requires the developer to name the exact bit-width they would like to match. This could feel like a “hole” in the generic programming capabilities of C because it does not provide a direct way to match the entire family of _ExtInt types. However, supporting the ability to have a single association that matches any size _ExtInt is unlikely to meet the user's expectations because C does not support a way to write type generic statements or function declarations to match the actual type.

We believe that any improvements to generic programming should be explored as an orthogonal topic to the introduction of _ExtInt. The proposed behavior gives a defensible model within generic selection expressions that should not be unfamiliar to C programmers who write generic function interfaces in

that _ExtInt(32) being distinct from _ExtInt(31) should be no more difficult to understand than int and long int being unique types.

Proposed Wording

The wording proposed is a diff from WG14 N2573. Green text is new text, while ~~red~~ text is deleted text.

Modify 5.2.4.2.1p1 (add to the end of the list):

> — width for an object of type `_ExtInt` or `unsigned _ExtInt` `EXTINT_MAXWIDTH /* see below */`
>
> The macro `EXTINT_MAXWIDTH` represents the maximum width *N* supported in the declaration of a bit-precise integer type (6.2.5) in the type specifier `_ExtInt(`*N*`)`. The value EXTINT_MAXWIDTH shall expand to a value that is greater than or equal to the value of ULLONG_WIDTH.

Modify 6.2.5p4: *Drafting note: this moves the existing text into a new paragraph.*

> There are five *standard signed integer types*, designated as **signed char**, **short int**, **int**, **long int**, and **long long int**. (These and other types may be designated in several additional ways, as described in 6.7.2.) ~~There may also be implementation-defined *extended signed integer types*.~~[~~N)~~] ~~The standard and extended signed integer types are collectively called *signed integer types*.~~[~~M)~~]

Insert a new paragraph immediately after 6.2.5p4: *Drafting note: the footnotes in the new paragraph are the same as the ones from the preceding paragraph.*

> A *bit-precise signed integer type* is designated as **_ExtInt**(N) where N is an integer constant expression that specifies the number of bits that are used to represent the type, including the sign bit. Each value of N designates a distinct type. [Footnote: Thus, _ExtInt(3) is not the same type as _ExtInt(4).] There may also be implementation-defined *extended signed integer types*.[41)] The standard, bit-precise signed integer types, and extended signed integer types are collectively called *signed integer types*.[42)]

Modify footnotes 41 and 42, respectively:

> 41) ~~Therefore, a~~Any statement in this document about signed integer types also applies to the bit-precise signed integer types and the extended signed integer types.
>
> 42) ~~Therefore, a~~Any statement in this document about unsigned integer types also applies to the bit-precise signed integer types and the extended unsigned integer types.

Modify the existing 6.2.5p6:

> For each of the signed integer types, there is a corresponding (but different) unsigned integer type (designated with the keyword unsigned) that uses the same amount of storage (including sign information) and has the same alignment requirements. The type _Bool and the unsigned integer types that correspond to the standard signed integer types are the standard unsigned integer types. The unsigned integer types that correspond to the bit-precise signed integer types are the *bit-precise unsigned integer types*. The unsigned integer types that correspond to the extended signed integer types are the *extended unsigned integer types*. The standard, bit-

precise unsigned, and extended unsigned integer types are collectively called *unsigned integer types*.

Modify the existing 6.2.5p7:

> The standard signed integer types and standard unsigned integer types are collectively called the *standard integer types*; the bit-precise signed integer types and bit-precise unsigned integer types are collectively called the *bit-precise integer types*; the extended signed integer types and extended unsigned integer types are collectively called the *extended integer types*.

Modify 6.3.1.1p1:

> …
> — The rank of `long long int` shall be greater than the rank of `long int`, which shall be greater than the rank of `int`, which shall be greater than the rank of `short int`, which shall be greater than the rank of `signed char`.
> — The rank of a bit-precise signed integer type shall be greater than the rank of any standard integer type with less width.
> — The rank of any unsigned integer type shall equal the rank of the corresponding signed integer type, if any.
> — The rank of any standard integer type shall be greater than the rank of any extended integer type with the same width or bit-precise integer type with the same width.
> …

Modify 6.3.1.1p2:

> The following may be used in an expression wherever an int or unsigned int may be used:
> — An object or expression with an integer type (other than int or unsigned int) whose integer conversion rank is less than or equal to the rank of int and unsigned int.
> — A bit-field of type _Bool, int, signed int, or unsigned int.
>
> If the original type is not a bit-precise integer type (6.2.5) and if~~If~~ an int can represent all values of the original type (as restricted by the width, for a bit-field), the value is converted to an int; otherwise, it is converted to an unsigned int. These are called the integer promotions. 61) All other types are unchanged by the integer promotions.

Insert a new example after 6.3.1.8p1:

> EXAMPLE 1 One consequence of _ExtInt being exempt from the integer promotion rules (6.3.1.1) is that an `_ExtInt` operand of a binary operator is not promoted to an `int` or `unsigned int` as part of the usual arithmetic conversions. Instead, a lower-ranked operand is converted to the higher-rank operand type and the result of the operation is the higher-ranked type.

```
_ExtInt(2) a2 = 1;
_ExtInt(3) a3 = 2;
_ExtInt(33) a33 = 1;
char c = 3;

a2 * a3 /* As part of the multiplication, a2 is converted to
           _ExtInt(3) and the result type is _ExtInt(3). */
```

```
a2 * c /* As part of the multiplication, c is promoted to int,
          a2 is converted to int and the result type is int. */
a33 * c /* As part of the multiplication, c is promoted to int,
           then converted to _ExtInt(33) and the result type
           is _ExtInt(33). */


void func(_ExtInt(8) a1, _ExtInt(24) a2) {
  /* Cast one of the operands to 32-bits to guarantee the
     result of the multiplication can contain all possible
     values. */
  _ExtInt(32) a3 = a1 * (_ExtInt(32))a2;
}
```

Modify 6.4.1p1 to add new keywords:

_ExtInt
_Bitwidthof

Modify 6.4.4.1p1:

integer-suffix:
unsigned-suffix long-suffix$_{opt}$
unsigned-suffix long-long-suffix
long-suffix unsigned-suffix$_{opt}$
long-long-suffix unsigned-suffix$_{opt}$
unsigned-suffix$_{opt}$ *bit-precise-int-suffix*
bit-precise-int-suffix unsigned-suffix$_{opt}$

bit-precise-int-suffix: one of **xi XI**

Modify 6.4.4.1p5 to add 2 rows to the bottom of the table:

xi or XI	**_ExtInt(N) Where the width N is the smallest N greater than 1 which can accommodate the value and the sign bit.**	**_ExtInt(N) Where the width N is the smallest N greater than 1 which can accommodate the value and the sign bit.**
Both u or U and xi or XI	**unsigned _ExtInt(N) Where the width N is the smallest N greater than 0 which can accommodate the value.**	**unsigned _ExtInt(N) Where the width N is the smallest N greater than 0 which can accommodate the value.**

Add a new paragraph below 6.4.4.1p6:

EXAMPLE 1 The `xi` suffix results in an `_ExtInt` that includes space for the sign bit even if the value of the constant is positive or was specified in hexadecimal or octal notation.

```
-3xi  /* Yields an _ExtInt(3) that is then negated; two value
         bits, one sign bit */
-0x3xi /* Yields an _ExtInt(3) that is then negated; two value
```

```
          bits, one sign bit */
3xi   /* Yields an _ExtInt(3);  two value bits, one sign bit */
3uxi  /* Yields an unsigned _ExtInt(2) */
-3uxi /* Yields an unsigned _ExtInt(2) that is then negated,
         resulting in wrap-around */
```

Modify 6.5.3p1 to add a new alternative to the syntax of *unary-expression*:

> **_Bitwidthof** (*assignment-expression*)

Modify 6.5.3.4**,** changing the section name:

> **The sizeof, _Alignof, and ~~_Alignof~~ _Bitwidthof operators**

Modify 6.5.3.4p1:

> The sizeof operator shall not be applied to an expression that has function type or an incomplete type, to the parenthesized name of such a type, or to an expression that designates a bit-field member. The _Alignof operator shall not be applied to a function type or an incomplete type. The `_Bitwidthof` operator shall be applied to an expression of integer type.

Insert a new paragraph after 6.5.3.4p3:

> The `_Bitwidthof` operator yields the width (6.2.6.2) of its unpromoted operand type, expressed in bits. If the operand designates a bit-field the result is the width of the bit-field. The operand is not evaluated and the result is an integer constant.

Modify the existing 6.5.3.4p5:

> The value of the result of ~~both~~ the `sizeof` and `_Alignof` operators is implementation-defined.~~, and its~~ The ~~type (an~~ unsigned integer type~~)~~ of the result of all three operators is `size_t`, defined in `<stddef.h>` (and other headers).

Modify 6.6p6:

> An *integer constant expression*[125] shall have integer type and shall only have operands that are integer constants, enumeration constants, character constants, `sizeof` expressions whose results are integer constants, `_Alignof` expressions, `_Bitwidthof` expressions, and floating constants that are the immediate operands of casts. Cast operators in an integer constant expression shall only convert arithmetic types to integer types, except as part of an operand to the `sizeof`, ~~or~~ `_Alignof`, or `_Bitwidthof` operator.

Modify 6.6p8:

> An arithmetic constant expression shall have arithmetic type and shall only have operands that are integer constants, floating constants, enumeration constants, character constants, `sizeof` expressions whose results are integer constants, ~~and~~ `_Alignof` expressions, and `_Bitwidthof` expressions. Cast operators in an arithmetic constant expression shall only convert arithmetic types to arithmetic types, except as part of an operand to the `sizeof`, ~~or~~ `_Alignof`, or `_Bitwidthof` operator.

Modify 6.7.2p1 to add a new entry to the *type-specifier* list:

> **`_ExtInt (`** *constant-expression* **`)`**

Modify 6.7.2p2 to add two new items to the list immediately below **unsigned long long**:

> — `_ExtInt(`*constant-expression*`)`, or `signed _ExtInt(`*constant-expression*`)`
> — `unsigned _ExtInt(`*constant-expression*`)`

Insert a new paragraph after 6.7.2p3 to the constraints section:

> The parenthesized constant expression that follows the **_ExtInt** keyword shall be an integer constant expression **N** that specifies the width (6.2.6.2) of the type. The value of **N** for `unsigned _ExtInt` shall be greater than or equal to 1. The value of **N** for `_ExtInt` shall be greater than or equal to 2. The value of N shall be less than or equal to the value of EXTINT_MAXWIDTH (see 5.2.4.2.1).

Modify 6.9p3:

> … (other than as part of the operand of a `sizeof`, ~~or~~ `_Alignof`, or `_Bitwidthof` operator…

Modify 6.9p5:

> … (other than as part of the operand of a `sizeof`, ~~or~~ `_Alignof`, or `_Bitwidthof` operator…

Modify 7.17.1p1 to add a new entry to the list of macros:

```
ATOMIC_EXTINT_LOCK_FREE(N)/* function-like macro corresponding
                             to _ExtInt(N) */
```

Modify 7.17.6p1 to add a new entry to the list of mappings:

> `atomic_extint(`*constant-expression*`) _Atomic _ExtInt(`*constant-expression*`)`

Modify 7.20.1.1p3:

> These types are optional. However, if an implementation provides standard integer types with widths of 8, 16, 32, or 64 bits, and no padding bits, it shall define the corresponding typedef names.

Modify 7.20.1.5p1:

> The following type designates a signed integer type capable of representing any value of any signed integer type other than a bit-precise signed integer type:
>
> `intmax_t`
>
> The following type designates an unsigned integer type capable of representing any value of any unsigned integer type other than a bit-precise unsigned integer type:
>
> `uintmax_t`
>
> These types are required.

Modify 7.21.6.1 p7: *Drafting note: We put the N in italics to make it clear that qN doesn't mean the letter q followed by the letter N. We did the same consistently in the other formatted i/o changes.*

q*N* *N* shall be a nonnegative decimal integer with no leading insignificant 0 digits. Specifies that a following d, i, o, u, x, or X conversion specifier applies to a signed or unsigned _ExtInt(*N*) bit-precise integer type argument of exactly *N* bits.

Modify 7.22.6.2 p11:

q*N* *N* shall be a nonnegative decimal integer with no leading insignificant 0 digits. Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to a signed or unsigned _ExtInt(*N*) of exactly *N* bits.

Modify 7.29.2.1 p7:

q*N* *N* shall be a nonnegative decimal integer with no leading insignificant 0 digits. Specifies that a following d, i, o, u, x, or X conversion specifier applies to a signed or unsigned _ExtInt(*N*) bit-precise integer type argument of exactly *N* bits.

Modify 7.29.2.2 p11:

q*N* *N* shall be a nonnegative decimal integer with no leading insignificant 0 digits. Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to a signed or unsigned _ExtInt(*N*) of exactly *N* bits.

Modify Annex E p2, Adding a new line below `ULLONG_WIDTH`:

```
#define  EXTINT_MAXWIDTH      64    // ULLONG_WIDTH
```

Acknowledgements

The authors would like to recognize the following people for their help with this work: Jens Gustedt, JeanHeyd Meneide, Joseph S. Myers, and Richard Smith.

References

1. The HLS compiler: https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html) refer to "Arbitrary Precision Integer"
2. The OpenCL FPGA compiler: https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html
3. The ac_int datatype: https://github.com/hlslibs/ac_types/blob/master/pdfdocs/ac_datatypes_ref.pdf
4. The current clang review: https://reviews.llvm.org/D73967
5. https://reviews.llvm.org/D59105 An earlier version of this feature was proposed for acceptance into clang/llvm, the code review is here.
6. The XiLinx HLS compiler arbitrary precision data types https://www.xilinx.com/support/documentation/sw_manuals/ug998-vivado-intro-fpga-design-hls.pdf

7. The bitWidth type property in swift:
   https://developer.apple.com/documentation/swift/int/2885648-bitwidth