Adding Fundamental Type for N-bit integers

Committee: ISO/IEC JTC1 SC22 WG14

Document Number: N2472

Authors: Melanie Blower, Tommy Hoffner, Erich Keane

Reply to:

Melanie.Blower@intel.com

Tommy.Hoffner@intel.com

Erich.Keane@intel.com

Contents

Abstract

We propose adding a set of special integer types spelled as _ExtInt(N), where N is an integral constant expression representing the number of bits to be used to represent the type. The goal is to provide a language spelling for all the supported extended integer types.

Motivation

In most hardware programmed with C compilers, the usual 8-, 16-, 32-, 64-bit width provides satisfactory expressiveness. However, in the case of FPGA hardware, using normal integer types where the full bit-width isn't used is extremely wasteful and creates severe performance/space concerns.

These types can be useful beyond FPGAs, for example using the type in a loop bound would provide information to the optimizer, potentially resulting in better code generation.

Existing solutions

Because of this, Intel has introduced this functionality in the High Level Synthesis (HLS) compiler (https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html) under the name "Arbitrary Precision Integer" (ap_int for short). This 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 HLS compiler has many users that program against the ap_int interface on a near daily basis.

A second version of this feature, implemented from scratch, is also available in Intel's oneAPI product under the -hls switch, currently in beta test: https://software.intel.com/en-us/oneapi

Intel plans to contribute an implementation of this proposal to clang/llvm after this paper is submitted to WG14.

Proposed solution

A set of special extended 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 is a type specifier, thus it can be used in any place a type can, including as the type of a bitfield.

An _ExtInt can be declared 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 and can represent negative values.

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, so these architectures emulate these types using larger integers. In these cases, they are expected to follow the 'as-if' rule and do math 'as-if' they were done at the specified number of bits.

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. However, they have lower precision than any of the built-in or other integer types (such as __int128). Usual arithmetic conversions also work the same, where the smaller ranked integer is converted to the larger.

There are two exceptions to the C rules for integers for these types is Integer Promotion. Unary, -, and ~ operators typically will promote operands to 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 in these cases. Likewise, if a Binary expression involves operands which are both _ExtInt, rather than promoting both operands to int the narrower operand will be promoted to match the size of the wider operand, and the result of the binary operation is the wider type.

_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.

Implementation Options

The LLVM compiler provides support for iN types in the intermediate representation, so it is straightforward to implement in this compiler. The maximum bit width supported is implementation defined: other compilers can provide a simple implementation by creating an upper limit on the bit width already supported and bumping any specific bit width to the nearest convenient size.

Impact on the standards

Lexical convention

A new keyword is added, _ExtInt. The use of underscore and capital letter conforms to C11 conventions.

Declarations

The type specifier _ExtInt(N) is proposed. For signed types, N >= 2. For unsigned types, N >= 1.

Expressions

All integer operations are supported. This includes:

- Arithmetic operators: + - * /
- Bitwise operators: % | & ^ >> << ~
  - As in ordinary integers, shifting by a negative quantity, or by a value larger than the width, is undefined.
  - Shift operations are performed in the width of the widest operand, so for example if shifting _ExtInt(9) by an integer literal, the left hand side will be widened to 32 bits.
- Casting operators: (bool) (char) (short) (int) (long)
- Compound assignment operators: += -= *= /= %= |= &= ^= >>= <<=
- Increment and decrement operators: x++ x-- ++x --x
- Miscellaneous operators: = +x -x !x sizeof() &x *x
- Relational operators: == != > < >= <=

Overflow
Overflow occurs when a value exceeds the allowable range of a given data type. For instance, (_ExtInt(3)) 7 + (_ExtInt(3)) 2 overflows, and the result is undefined. For unsigned operations, overflow behavior is well defined.

Conversions and Promotions
- If all operands are _ExtInt type, then all operands are interpreted as _ExtInt type and consequently the result is an _ExtInt type.
- For operations with two operands of different types, the larger type takes precedence. Note that an unsigned type is considered larger than a signed type of the same width.

C library
The C library does not support _ExtInt.

Compatibility
Adding the _ExtInt type does not create backward compatibility problems

References

1. https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html) refer to "Arbitrary Precision Integer"
2. https://reviews.llvm.org/D59105 An earlier version of this feature was proposed for acceptance into clang/llvm, the code review is here.
3. An earlier version of this feature is available in Intel's oneAPI product under the -hls switch, currently in beta test: https://software.intel.com/en-us/oneapi