Adding Fundamental Type for N-bit integers

Committee: ISO/IEC JTC1 SC22 WG14

Document Number: N2501

Revises 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 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 has many users that relies on ap_ints to achive efficient programs on a daily basis.

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.

Intel has submitted an implementation of this proposal to clang/llvm under review here, https://reviews.llvm.org/D73967. The rules for integer promotion have been disabled in this patch pending recommendation from WG14.

Since submitting the patch to clang/llvm, we learned that the XiLinux 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 Xilink the maximum width is limited to 1024 bits.

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 rank than any of the built-in or other integer types with the same precision (such as __int128). (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.

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.

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.

A new decimal constant suffix, xi, is added to denote extended integer constants. The type of the constant will be the narrowest _ExtInt that can hold the value. This makes it easy to write expressions involving literals and _ExtInt that will be computed in the most efficient solution, and circumvents automatic promotion of the literal value to int. The availability of decimal constants with more precision than 64 bits will be very useful for programmers who need the _ExtInt feature.

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: % | & ^ >> << ~
- 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.

C library

A new entry EXTINT_MAXBITS provides the highest supported value for N in _ExtInt(N)

A new set of macros PRXI(N) defined in <inttypes.h> provides the formatting macro for _ExtInt(N) intput and output.

In section 7.21.6.1:paragraph 7, a parameterized length modifier xi(N) is added to describe arguments of type _ExtInt(N). Use similar language as is used to describe "ll", "Specifies that a following d, i, o, u, x, or X conversion specifier applies to a _ExtInt(N) argument; or that a following n conversion specifier applies to a pointer to a _ExtInt(N) argument". Also added for scanf and the wide character formatting functions.

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. They number of chunks are 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.

Compatibility

Adding the _ExtInt type does not create backward compatibility problems

Acknowledgements

The authors would like to recognize the following people for their help with this work: Aaron Ballman, Jens Gustedt, Joseph S. Myers, 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 FPGA compiler: https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html
3. The current clang review: https://reviews.llvm.org/D73967
4. https://reviews.llvm.org/D59105 An earlier version of this feature was proposed for acceptance into clang/llvm, the code review is here.
5. An earlier version of this feature is available in Intel's oneAPI product, currently in beta test: https://software.intel.com/en-us/oneapi
6. The XiLinux HLS compiler arbitrary precision data types https://www.xilinx.com/support/documentation/sw_manuals/ug998-vivado-intro-fpga-design-hls.pdf