SWAR Support for LLVM

LLVM SIMD to SWAR Pass - Introductory Example

Type Legalization

- Types are considered legal on an architecture if values of that type are directly supported by a register class on that architecture and if the instruction set provides explicit support for operations on the type.
- Vector types are considered legal on an architecture when the total vector size in bits is equal to the size of SIMD registers on the architecture and the scalar size of vector elements is supported by specific SIMD operations on that architecture.
- In general, vector types such as `<6 x i3>` are considered illegal on practical architectures, because they have neither 18-bit registers, nor SIMD operations that support 3-bit field widths.
- In the LLVM infrastructure, type legalization *(http://llvm.org/docs/CodeGenerator.html#selectiondag-legalizetypes-phase)* is the process of transforming IR code to replace all illegal types and operations with legal ones.
    1. When a vector is too big for architectural registers, *splitting* breaks up the vector into multiple shorter vectors that do fit the architecture.
    2. When a vector element type (e.g. `i3`) is not supported by the natural SIMD field widths of an architecture, the elements are *widened* to a natural field width. For example, a vector of type `<6 x i3>` could be widened to `<6 x i8>`.
    3. If all else fails, vectors are *scalarized* to operate on the elements of the vector individually.

Widening vs. SWAR

Applying a SWAR approach is an alternative to widening. In this case operations on `<6 x i3>` vectors are performed using bitwise logic and masking, while maintaining a total of 18 bits for the size of the elements.

Example

Here is a complete SWAR add program that you can run with `lli` or compile with `llc`.

```
@.str = private constant [34 x i8] c"<6 x i3><%i, %i, %i, %i, %i, %i>\0A\00", align 1
declare i32 @printf(i8*, ...)

define i18 @swar_add_6xi3_as_i18(i18 %a1, i18 %a2)  {
entry:
  %m1 = and i18 %a1, 112347   ; mask with b'011011011011011011'
  %m2 = and i18 %a2, 112347
  %r1 = add i18 %m1, %m2
  %r2 = xor i18 %a1, %a2
  %r3 = and i18 %r2, 149796   ; mask with b'100100100100100100'
  %r4 = xor i18 %r1, %r3
  ret i18 %r4
}

define <6 x i3> @swar_add_6xi3(<6 x i3> %v1, <6 x i3> %v2)  {
entry:
  %x1 = bitcast <6 x i3> %v1 to i18
  %x2 = bitcast <6 x i3> %v2 to i18
  %x3 = call i18 @swar_add_6xi3_as_i18(i18 %x1, i18 %x2)
  %r = bitcast i18 %x3 to <6 x i3>
```

```
  ret <6 x i3> %r
}

define i32 @main() {
  %a = call <6 x i3> @swar_add_6xi3(<6 x i3><i3 3, i3 4, i3 2, i3 1, i3 0, i3 -1>, <6 x i3
  %a0 = extractelement <6 x i3> %a, i32 0
  %a1 = extractelement <6 x i3> %a, i32 1
  %a2 = extractelement <6 x i3> %a, i32 2
  %a3 = extractelement <6 x i3> %a, i32 3
  %a4 = extractelement <6 x i3> %a, i32 4
  %a5 = extractelement <6 x i3> %a, i32 5
  %1 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([34 x i8]* @.str, i32 0, i
  ret i32 0
}
```

Using `lli` we can see proper results in "interpreter mode".

```
$ lli -force-interpreter swar_add_6xi3.ll
<6 x i3><7, 0, 6, 5, 7, 5>
```

But if we skip the interpreter and use JIT'd code, we get erroneous results, because of an LLVM bug.

```
$ lli swar_add_6xi3.ll
<6 x i3><255, 255, 255, 255, 255, 255>
```

Scalarization

The `opt` program has a `-scalarizer` option that provides useful insight into the scalarization process. For example, consider the following LLVM code.

```
define <6 x i3> @add_6xi3(<6 x i3> %a, <6 x i3> %b)  {
entry:
  %0 = add <6 x i3> %a, %b
  ret <6 x i3> %0
}
```

The output from scalarizing this file with `opt -scalarizer -S swaradd1.ll` gives equivalent logic.

```
; ModuleID = 'add_6xi3.ll'

define <6 x i3> @add_6xi3(<6 x i3> %a, <6 x i3> %b) {
entry:
  %a.i0 = extractelement <6 x i3> %a, i32 0
  %b.i0 = extractelement <6 x i3> %b, i32 0
  %.i0 = add i3 %a.i0, %b.i0
  %a.i1 = extractelement <6 x i3> %a, i32 1
  %b.i1 = extractelement <6 x i3> %b, i32 1
  %.i1 = add i3 %a.i1, %b.i1
  %a.i2 = extractelement <6 x i3> %a, i32 2
  %b.i2 = extractelement <6 x i3> %b, i32 2
  %.i2 = add i3 %a.i2, %b.i2
  %a.i3 = extractelement <6 x i3> %a, i32 3
  %b.i3 = extractelement <6 x i3> %b, i32 3
```

```
  %.i3 = add i3 %a.i3, %b.i3
  %a.i4 = extractelement <6 x i3> %a, i32 4
  %b.i4 = extractelement <6 x i3> %b, i32 4
  %.i4 = add i3 %a.i4, %b.i4
  %a.i5 = extractelement <6 x i3> %a, i32 5
  %b.i5 = extractelement <6 x i3> %b, i32 5
  %.i5 = add i3 %a.i5, %b.i5
  %.upto0 = insertelement <6 x i3> undef, i3 %.i0, i32 0
  %.upto1 = insertelement <6 x i3> %.upto0, i3 %.i1, i32 1
  %.upto2 = insertelement <6 x i3> %.upto1, i3 %.i2, i32 2
  %.upto3 = insertelement <6 x i3> %.upto2, i3 %.i3, i32 3
  %.upto4 = insertelement <6 x i3> %.upto3, i3 %.i4, i32 4
  %0 = insertelement <6 x i3> %.upto4, i3 %.i5, i32 5
  ret <6 x i3> %0
}
```


Updated Tue Feb. 16 2016, 13:28 by cameron.