LLVM sBPF sign-extend optimzation

By @Richard Patel Jul 4, 2024

Problem

In sBPF there are no dedicated comparison operators and no condition registers.
Instead, sBPF features conditional jumps like `jsgt r2, r3, OFF` (jump to `OFF` if `r2 > r3`).

There are conditional jumps for 64-bit unsigned and 64-bit signed registers.
Expressing 32-bit signed conditions is quite difficult.

Example

```
void ex1(int r1, int r2) {
  if(r1 < r2) abort();
}
```

Suppose the 32-bit value of w1 is `0` and the value of w2 is `1`.

- `w1 = 00000000`
- `w2 = 00000001`

Now remember that `w1` and `w2` are sub-registers of `r1` and `r2`. Since we have no `jslt` instruction that operates on 32-bit registers, we will have to use `r1` and `r2` instead.

From the compiler's point of view, since we have been dealing with sub-registers, the upper half of the "parent" register is undefined:

- `r1 = ????????00000000`
- `r2 = ????????00000001`

Note that every sBPF instruction clearly defines what the upper 32-bit of an output register are. But crucially, the output **differs** between different 32-bit ALU instructions. In x86, using `eax` as an output implicitly zeros the upper half of `rax`. However, sBPF 32-bit instructions do not always write zeros.

This means that we could encounter a bit pattern like so:

- `r1 = ffffffff00000000`
- `r2 = 0000000000000001`

Both `jslt` and `jlt` (signed and unsigned "jump if less") would return the incorrect result.

Current Solution

The compiler sign extends `w1` and `w2` (as of anza-xyz/llvm-project@8bc30e821e)

```
mov32 w8, w1
lsh64 r8, 32
arsh64 r8, 32 # sign extend

mov32 w9, w2
lsh64 r9, 32
arsh64 r9, 32 # sign extend

jslt r8, r9
```

This is unfortunate. We require two extra registers and 6 extra instructions in total.

Slightly Better Solution

The `add32 REG, IMM` instruction implicitly sign extends. Thus, we could instead do

```
mov32 w8, w1
add32 w8, 0 # sign extend

mov32 w9, w2
add32 w9, 0 # sign extend

jslt r8, r9
```

This saves two instructions.

The compiler is still dumb

Let's consider the following code (see llvm/test/CodeGen/SBF/loop-exit-cond.ll)

```
typedef unsigned long u64;
void foo(char *data, int idx, u64 *);
int test(int len, char *data) {
  if (len < 100) {
    for (int i = 1; i < len; i++) {
      u64 d[1];
      d[0] = data[0] ?: '0';
      foo("%c", i, d);
    }
  }
  return 0;
}
```

Let's look at the generated code. To prepare, run the following commands in your LLVM checkout:

```
# Build LLVM
cmake --build build -j

# Run the test
build/bin/llvm-lit ./llvm/test/CodeGen/SBF/loop-exit-cond.ll -v

# Inspect the generated assembly
build/bin/llc -march=sbf -mcpu=v3 /data/ripatel/llvm-project/build/test/CodeGen/SBF/Output/loop-exit-cond.ll.tmp1
```

With the current compiler code, the generated loop exit condition is this:

```
; w7 stores "int i"
; r8 stores "long len" (sign extended)

        add32 w7, 1
        mov64 r2, r7
        lsh64 r2, 32
        arsh64 r2, 32
        jslt r2, r8, LBB0_8
```

With the "slightly better" patch, the generated code is this:

```
; w8 stores "int i"
```

```
; r3 stores "long len" (sign extended)

    add32 w8, 1
    mov32 w2, w8
    add32 w2, 0
    jslt r2, r3, LBB0_8
```

The above snippet is still problematic. `r8` is already sign extended, so the `mov32 w2, w8` and `add32 w2, 0` instructions are unnecessary.

This is fixed by a peephole optimization that loops over SSA blocks. Whenever we find an explicit sign extension construct with an input produced by a sign-extending opcode (add32, sub32, mul32), we eliminate the sign extension.

Finally, we get:

```
; w8 stores "int i"
; r2 stores "long len" (sign extended)

    add32 w8, 1
    jslt r8, r2, LBB0_8
```