RISC-V Sign Extension Optimizations

Craig Topper

Agenda

- **RISC-V Scalar Integer ISA**
- SelectionDAG Handling of i32
- RISCVSExtWRemoval Pass
- RISCVCodeGenPrepare Pass

RISC-V Scalar Integer ISA Overview

- RISC-V has two main variants, RV32I and RV64I.
  - The 32 or 64 refer to the integer register size.
  - No sub-registers.
- Computational instructions operate on the whole register.
- Can load or store 1, 2, or 4 bytes. RV64I can also load or store 8 bytes.
  - If loaded value is smaller than register size, it can be either sign or zero extended to register size.

RISC-V Narrow Type Example

- Operations on integer types smaller than register size often require additional instructions.

```
bool foo(int16_t x, int16_t y) {
  int16_t z = x + y;
  return z < 2;
}
```

Assembly for RV32I

```
foo(int16_t, int16_t):
  add    a0, a0, a1 // Perform the addition.
  slli   a0, a0, 16 // shift bits [15:0] to [31:16]
  srai   a0, a0, 16 // shift right, filling with sign bits
  slti   a0, a0, 2  // a0 = (a0 < 2) ? 1 : 0
  ret
```

Assembly for RV64I

```
foo(int16_t, int16_t):
  add    a0, a0, a1 // Perform the addition.
  slli   a0, a0, 48 // shift bits [15:0] to [63:48]
  srai   a0, a0, 48 // shift right, filling with sign bits
  slti   a0, a0, 2  // a0 = (a0 < 2) ? 1 : 0
  ret
```

The `slli` and `srai` sign extend the result of the `add` from 16 bits to the whole register.

`slli+srli` are used to zero extend due to limitations of AND with immediate encoding

RV64I and 32-bit integers

- C/C++ code often has many 32-bit integers.
  - `int` type is 32 bits for RISC-V as you'd expect.
  - C integer promotion rules promote operations on `char` or `short` types to `int` or `unsigned int`.
- **Creators of RV64I recognized that the extra `slli` and `srli`/`srai` instructions would be inefficient.**

RV64I Optimization for 32-bit integers

- Most binary arithmetic instructions have a "W"-suffixed variant that:
  - Ignores bits [63:32] of inputs
  - Sign extends the result by copying bit 31 to [63:32]
- AND/OR/XOR don't have "W"-suffixed versions.
  - If inputs are sign extended, result is sign extended.
- Comparison instructions didn't need any changes.
  - Unsigned comparisons work correctly on sign extended inputs.
- `signed int` and `unsigned int` are sign extended to 64 bits for function argument and returns.

- **If "W"-suffixed instructions are used for i32, we often won't need any extra instructions.**
- **At worst we can sign extend by using `addiw` with a 0 immediate (alias `sext.w`)**
- **Zero extend is still 2 shifts.***

*The Zba instruction extension improves this to 1 instruction.

RISC-V i32 Example

- Revisiting our earlier example with `int16_t` replaced by `int32_t`

```
bool foo(int32_t x, int32_t y) {
  int32_t z = x + y;
  return z < 2;
}
```

Assembly for RV32I

```
foo(int32_t, int32_t):
  add   a0, a0, a1  // Addition.
  slti   a0, a0, 2  // a0 = (a0 < 2) ? 1 : 0
  ret
```

Assembly for RV64I

```
foo(int32_t, int32_t):
  addw   a0, a0, a1 // Addition and sign extend.
  slti   a0, a0, 2  // a0 = (a0 < 2) ? 1 : 0
  ret
```

Agenda

- RISC-V Scalar Integer ISA
- **SelectionDAG Handling of i32**
- RISCVSExtWRemoval Pass
- RISCVCodeGenPrepare Pass

SelectionDAG for Narrow Types

- Types narrower than a legal integer type need to be promoted.
- Narrow loads become "any" extending loads. Narrow stores become truncating stores.
- sext_inreg or AND operations are inserted before any operation where the upper bits of the promoted input would affect the lower bits of the promoted result.
  - add/sub/mul/and/or/xor do not need any extra operations before them
    - Bits [63:32] of the input can't affect bits [31:0] of the result.
  - (i32 (udiv X, Y)) -> (i64 (udiv (and X, 0xffffffff), (and Y, 0xffffffff)))
  - (i32 (sdiv X, Y)) -> (i64 (sdiv (sext_inreg X, i32), (sext_inreg Y, i32)))
  - For RISC-V, sext_inreg/and become a `slli+srai/srli` at instruction selection.

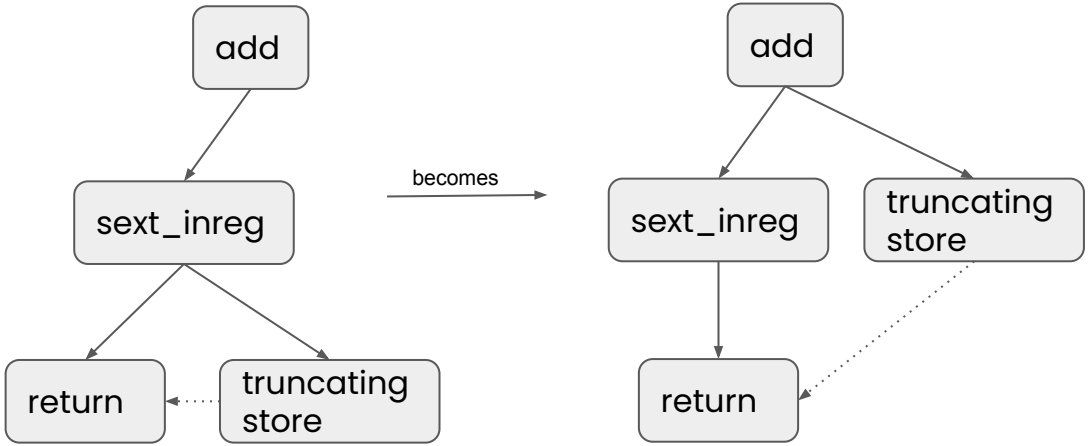
SelectionDAG for "W" instructions

- Many of the promoted sequences cannot be pattern matched to W instructions at instruction selection.
  - Promoted sequence does not guarantee a sign extended result.
  - Promoted sequences can also be created from mixed i32/i64 code.
    - Need to remember that shift amounts >31 were UB.

- Use custom RISC-V "W" operations for udiv/urem/sdiv and shifts by non-constant amounts.
- Loads are promoted to sign extending load.
- Comparisons use sext_inreg for signed *and* unsigned comparisons.
- **Add/sub/mul and shl by constant have a sext_inreg added *after* them.**
  - All uses will see the sign extended result instead of only uses that create a sext_inreg.
    - All uses can pattern match to addw/subw/mulw/slliw
  - DAGCombine can optimize out redundant sext_inreg operations.

add/sub/mul/shl Problems

- Unfortunately, DAG combine will also remove unneeded sext_inreg if the sign extended bits are not used by later operations.
  - Sometimes removed for a subset of uses. Creating the problem we were trying to avoid.



Generated assembly

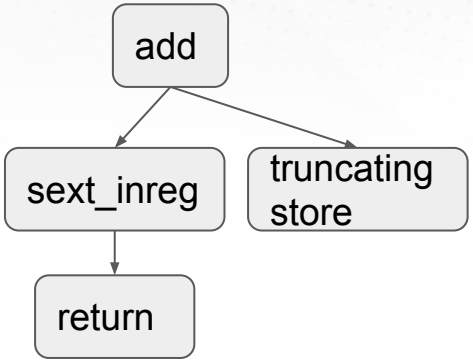
```
add     a3, a0, a1
sext.w  a0, a3
sw      a3, (a2)
ret     a0
```

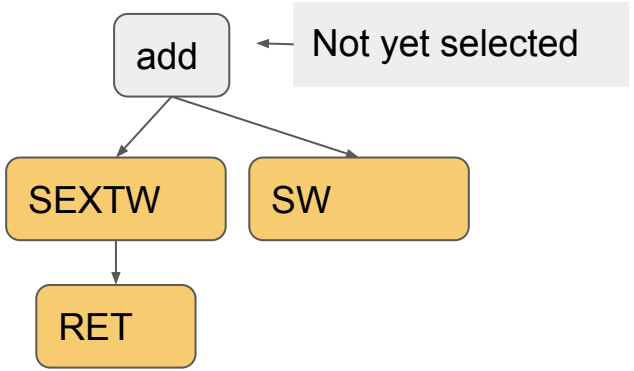
Ideal assembly

```
addw    a0, a0, a1
sw      a0, (a2)
ret     a0
```

add/sub/mul/shl Solution

- Undo the DAGCombine "optimization" at instruction selection
- Instruction Selection occurs bottom up.
  - Select "return" to RET instruction
  - Select sext_inreg to SEXTW
  - Select truncating store to store word instruction (SW).
- When we reach "add"
  - Examine its already selected users.
  - If all users ignore the upper 32 bits
    - Select as ADDW.
    - Otherwise select as ADD.
- After all nodes are selected, post-process remove `SEXTW X` when `X` comes from a W instruction.





add/sub/mul/shl Future

- Current code only examines the direct uses of the "add".
  - AND/OR/XOR are considered to use all bits.
  - We could look instead look at the users of AND/OR/XOR recursively.
  - Need to limit recursion depth to control compile time.

- Maybe we should stop DAGCombine from optimizing "free" sext_inreg operations?

Agenda

- RISC-V Scalar Integer ISA
- SelectionDAG Handling of i32
- **RISCVSExtWRemoval Pass**
- RISCVCodeGenPrepare Pass

RISCVSExtWRemoval

The problem

- SelectionDAG can propagate ComputeNumSignBits information from liveouts of one basic block to the liveins of another block.
  - Requires the producing block to be visited before the consume block. Doesn't work for loops.

RISCVSExtWRemoval

Example

```
void bar(int);


void foo(int x, int y) {
  x += y;
  do {
    bar(x);
    x = x >> 1;
  } while (x != 0);
}
```

Original Generated assembly

```
        addw    s1, a1, a0
        li      s2, 1
.LBB0_1:
        sext.w  s0, s1
        mv      a0, s0
        call    bar@plt
        sraiw   s1, s1, 1
        bltu    s2, s0, .LBB0_1
```

The call to `bar` requires `x` to be sign extended so a `sext.w` is generated by SelectionDAG. This instruction is unneeded. It's input is either the result of the `addw` or the `sraiw`.

RISCVSExtWRemoval

The Solution

- SSA Machine IR pass to remove unneeded **sext.w** instructions.
- Performs a depth first search starting at the sext.w input.
- 3 cases of nodes the search can encounter.
  - A "W" instruction, no need to look any further.
  - An instruction that propagates sign extension such as AND/OR/XOR/COPY/PHI, add the input instructions to the search.
  - Any other instruction, terminate the search. We can't remove the `sext.w`.
- If the search always reaches a "W" instruction, we **can** remove the `sext.w`.
- Visited set used to avoid phi cycles.
- There is currently no recursion limit so we pick up non-loop cases that SelectionDAG misses.

RISCVSExtWRemoval Special Cases

- Copies from sign extended argument registers are treated as sign extended values.
  - Requires extra bookkeeping state created from SelectionDAG argument lowering. MachineIR doesn't preserve this information.
- Some instructions can be freely converted to W instructions, for example ADD->ADDW.
  - If we encounter one of these, search forward through users to see if the upper bits are ignored.
  - If upper bits aren't used, the instruction is convertible
    - Remember it and convert if the backward search determines that it would allow the original sext.w to be removed.

- We need to add support for sign extended values returned from calls.

Agenda

- RISC-V Scalar Integer ISA
- SelectionDAG Handling of i32
- RISCVSExtWRemoval Pass
- **RISCVCodeGenPrepare Pass**

RISCVCodeGenPrepare

The Problem

- Middle-end optimizations replace `sext` instructions with `zext` if the sign bit is known to be zero.
- **i32->i64 `zext` is never cheaper than `sext` for RISC-V and `sext` can be free.**

```
void foo(int *x, int n) {
  for (int i = 0; i < n; ++i)
    x[i] += 1;
}
```

RISC-V Assembly

```
foo:
        blez    a1, .LBB0_2
        slli    a1, a1, 32
        srli    a1, a1, 32
.LBB0_1:
        ; loop body
.LBB0_2:
        ret
```

LLVM IR

```
define void @foo(ptr %x, i32 signext %n) {
entry:
  %cmp3 = icmp sgt i32 %n, 0
  br i1 %cmp3, label %preheader, label %cleanup

preheader:
  ; sign bit of i32 %n is known 0 here.
  %wide.trip.count = zext i32 %n to i64
  br label %for.body

for.body:
  ...

cleanup:
  ret void
```

RISCVCodeGenPrepare

Partial Solution

- Pre-instruction selection IR pass
- For each `zext` instruction in basic blocks with a single predecessor
  - Examine the terminator condition of predecessor
  - If condition implies the sign bit is 0 when branching to the `zext`
    - Replace `zext` with `sext`
- Looking for a single predecessor is a very simple dominance check
  - **It will miss some cases.**

Future Work

- RISCVCodeGenPrepare only handles one case of `zext` that can be treated as `sext`
  - Handling more cases means redoing more middle end analysis.
- Some passes that convert `zext` to `sext`
  - Correlated Value Propagation, InstCombine, SCCP
- Ideally we wouldn't change `sext` to `zext` for RISC-V or have some means of retaining the information
  - Maybe a flag on `zext` instructions similar to `nsw/nuw/exact`

Backup

Comparison with MIPS64 Implementation

- Like RV64I, MIPS64 has 64-bit integer registers with no sub-registers.
- Also has instructions that produce results that are sign extended from bit 31.
- Unlike RV64I, these instructions require the inputs to be sign extended or the result is unpredictable.
- This means the compiler must keep i32 values in sign extended form at all times.
- To do this, i32 is considered a legal type and a 32-bit register class is used.
- Truncate from i64 to i32 becomes a sign extend from bit 31 unless the upper bits are known sign extended.
- A similar approach could work for RISC-V.
  - Might be simpler implementation, but it's big change with unknown impact.
  - Some newer extensions have useful instructions that don't have "W" versions meaning we'd need explicit sign extends if we used them for 32-bit values.