RISC-V Bitmanip Optimizations

Maxim Blinov

RISC-V Bit Manipulation Instructions

- Current draft spec version 0.92
- Developed by Claire Wolf and Ken Dockser
- ~50 new instructions spread across 9 sub-extensions
- Intended for fast and convenient bitwise arithmetic
- Supported in LLVM 11.0.0
- Experimental branch available for GCC at:

```
github.com/embecosm/riscv-gcc/tree/riscv-bitmanip
github.com/embecosm/riscv-binutils-gdb/tree/riscv-bitmanip
```

EMBECOSM®

RISC-V Bit Manipulation Instructions

- Embench size results suggest code size improvement:

target	edn	crc32	cubic	nettle-aes	aha-mont64	sglib-combined	picojpeg	st	wikisort	qrduino	minver	huffbench	matmult-int	ud	slre	nsichneu	statemate	nettle-sha256	nbody	total	-delta	-delta (rel)
size-test-gcc-bitmanip-zbs	1452	230	2330	2848	1042	2316	7990	848	4186	6052	1054	1626	420	702	2400	15036	3686	5552	700	60470	12	0.02%
size-test-gcc-bitmanip-zbb	1434	234	2332	2834	1042	2316	7916	848	4186	6040	1054	1622	420	702	2400	15036	3686	4042	700	58844	1638	2.71%
size-test-gcc-bitmanip-zbt	1452	230	2332	2848	1042	2316	8000	848	4186	6052	1054	1626	420	702	2400	15036	3686	5502	700	60432	50	0.08%
size-test-gcc-bitmanip-zbr	1452	230	2332	2848	1042	2316	8000	848	4186	6052	1054	1626	420	702	2400	15036	3686	5552	700	60482	0	0.00%
size-test-gcc-bitmanip-zbf	1452	230	2332	2848	1042	2316	8000	848	4186	6052	1054	1626	420	702	2400	15036	3686	5552	700	60482	0	0.00%
size-test-gcc-bitmanip-zbm	1452	230	2332	2848	1042	2316	8000	848	4186	6052	1054	1626	420	702	2400	15036	3686	5552	700	60482	0	0.00%
size-test-gcc-bitmanip-zbp	1434	234	2332	2834	1042	2316	7920	848	4190	6052	1054	1626	420	702	2400	15036	3686	4026	700	58852	1630	2.70%
size-test-gcc-bitmanip-zbe	1452	230	2332	2848	1042	2316	8000	848	4186	6052	1054	1626	420	702	2400	15036	3686	5552	700	60482	0	0.00%
size-test-gcc-bitmanip-base	1434	230	2330	2834	1042	2316	7906	848	4182	6036	1054	1622	420	702	2400	15036	3686	4026	700	58804	1678	2.77%
size-test-gcc-bitmanip-zbc	1452	230	2332	2848	1042	2316	8000	848	4186	6052	1054	1626	420	702	2400	15036	3686	5552	700	60482	0	0.00%
size-test-gcc-bitmanip-none	1452	230	2332	2848	1042	2316	8000	848	4186	6052	1054	1626	420	702	2400	15036	3686	5552	700	60482	0	0.00%
Best individual benchmark improvement:																						
min	1434	230	2330	2834	1042	2316	7906	848	4182	6036	1054	1622	420	702	2400	15036	3686	4026	700	58804		
-delta	18	0	2	14	0	0	94	0	4	16	0	4	0	0	0	0	0	1526	0	1678		
-delta (% decrease)	1.24%	0.00%	0.09%	0.49%	0.00%	0.00%	1.18%	0.00%	0.10%	0.26%	0.00%	0.25%	0.00%	0.00%	0.00%	0.00%	0.00%	27.49%	0.00%	2.77%		

- Significant improvement for SHA256 benchmark
  – lots of bitwise rotates and shifts

EMBECOSM®

RISC-V Bit Manipulation Instructions

	benchmark runtime (clock cycles)	benchmark runtime (clock cycles)	percent improvement
aha-mont64	2.34E+07	2.34E+07	0.00%
crc32	2.29E+07	2.29E+07	0.07%
cubic	2.56E+07	2.56E+07	0.00%
edn	3.54E+07	3.53E+07	0.45%
huffbench	1.48E+07	1.48E+07	0.00%
matmult-int	3.26E+07	3.26E+07	0.00%
minver	2.03E+07	2.03E+07	0.00%
nbody	2.25E+07	2.25E+07	0.00%
nettle-aes	2.10E+07	2.08E+07	0.73%
nettle-sha256	2.21E+07	1.33E+07	39.88%
nsichneu	1.57E+07	1.57E+07	0.00%
picojpeg	2.20E+07	2.10E+07	4.34%
qrduino	2.20E+07	2.22E+07	-1.06%
sglib-combine	1.63E+07	1.63E+07	0.00%
slre	1.67E+07	1.67E+07	0.02%
st	2.45E+07	2.45E+07	0.00%
statemate	1.64E+07	1.64E+07	0.00%
ud	1.92E+07	1.92E+07	0.00%
wikisort	1.75E+07	1.75E+07	0.00%
Total	4.11E+08	4.01E+08	2.40%

- Embench speed result also favorable, closely tracks code size improvement
- Run on verilated picorv32 core with bitmanip support
- Primary focus: Teach GCC codegen about new insns

EMBECOSM®

RISC-V Bit Manipulation Instructions

Example: sbset (“Single Bit Set”)

- Zbs class of instructions
- Greatly simplifies single-bit flag set/unset and bit extract operations
- Intrinsics are OK, but what about code generation?

RISC-V Bit Manipulation Instructions

```
#include <stdlib.h>
#include <stdint.h>

void f(uint32_t *data, size_t n, uint32_t
*flags) {
  uint32_t max = 0;

  for(size_t i = 0; i < n; ++i) {
    if (data[i] > max) {
      max = data[i];
    }
  }

  if (max < 200) {
    *flags |= 1 << 24;
  }
  else if (max > 500) {
    *flags |= 1 << 25;
  }
  else {
    *flags |= 1 << 26;
  }
}
```

```
00000000 <f>:
   0:   00062803            lw      a6,0(a2)
   4:   c99d                beqz    a1,3a <.L2>
   6:   058a                slli    a1,a1,0x2
   8:   00b50733            add     a4,a0,a1
   c:   4681                li      a3,0

0000000e <.L4>:
   e:   411c                lw      a5,0(a0)
  10:   0511                addi    a0,a0,4
  12:   00f6f363            bgeu    a3,a5,18 <.L3>
  16:   86be                mv      a3,a5

00000018 <.L3>:
  18:   fee51be3            bne     a0,a4,e <.L4>
  1c:   0c700793            li      a5,199
  20:   00d7fd63            bgeu    a5,a3,3a <.L2>
  24:   1f400793            li      a5,500
  28:   02d7f063            bgeu    a5,a3,48 <.L7>
  2c:   020007b7            lui     a5,0x2000
  30:   00f86833            or      a6,a6,a5
  34:   01062023            sw      a6,0(a2)
  38:   8082                ret

0000003a <.L2>:
  3a:   010007b7            lui     a5,0x1000
  3e:   00f86833            or      a6,a6,a5
  42:   01062023            sw      a6,0(a2)
  46:   8082                ret

00000048 <.L7>:
  48:   040007b7            lui     a5,0x4000
  4c:   00f86833            or      a6,a6,a5
  50:   01062023            sw      a6,0(a2)
  54:   8082                ret
```

EMBECOSM®

RISC-V Bit Manipulation Instructions

```
#include <stdlib.h>
#include <stdint.h>

void f(uint32_t *data, size_t n, uint32_t
*flags) {
  uint32_t max = 0;

  for(size_t i = 0; i < n; ++i) {
    if (data[i] > max) {
      max = data[i];
    }
  }

  if (max < 200) {
    *flags |= 1 << 24;
  }
  else if (max > 500) {
    *flags |= 1 << 25;
  }
  else {
    *flags |= 1 << 26;
  }
}
```

```
00000000 <f>:
   0:   00062803            lw      a6,0(a2)
   4:   c985                beqz    a1,34 <.L2>
   6:   058a                slli    a1,a1,0x2
   8:   00b506b3            add     a3,a0,a1
   c:   4781                li      a5,0

0000000e <.L3>:
   e:   4118                lw      a4,0(a0)
  10:   0511                addi    a0,a0,4
  12:   0ae7f7b3            maxu    a5,a5,a4
  16:   fed51ce3            bne     a0,a3,e <.L3>
  1a:   0c700713            li      a4,199
  1e:   00f77b63            bgeu    a4,a5,34 <.L2>
  22:   1f400713            li      a4,500
  26:   00f77c63            bgeu    a4,a5,3e <.L6>
  2a:   29981813            sbseti  a6,a6,0x19
  2e:   01062023            sw      a6,0(a2)
  32:   8082                ret

00000034 <.L2>:
  34:   29881813            sbseti  a6,a6,0x18
  38:   01062023            sw      a6,0(a2)
  3c:   8082                ret

0000003e <.L6>:
  3e:   29a81813            sbseti  a6,a6,0x1a
  42:   01062023            sw      a6,0(a2)
  46:   8082                ret
```

Integer construction gets folded into single sbseti insn

EMBECOSM®

RISC-V Bit Manipulation Instructions

- GCC quite happy to convert shifts and masks into single bit set/unset
- Saves on code size, both in terms of integer construction and constant pools

RISC-V Bit Manipulation Instructions

Example: cmov instruction (“Conditional move”)

- One of the few ternary (Zbt) instructions
- Real-world example: Existing core designed around two-input operations
  - Implementing ternary support just for one instruction not profitable (requires significant core changes)
  - Instead teach the compiler about 2-input cmov and “see what happens” - Can we get away with 2 input?
  - Really keen to avoid expensive branch instructions
  - Compare with 3 input version

RISC-V Bit Manipulation Instructions

Example from Embench:

"Vanilla" 3-input cmov

```
.L150:
        sh      a5,0(a4)
        addi    s10,s10,2
        bne     s10,s8,.L124
        cmov    a5,s9,s3,s2

        addi    a3,s7,%lo(.LANCHOR19)
        addi    a2,a5,128
```

Custom 2-input cmov

```
.L150:
        sh      a5,0(a4)
        addi    s9,s9,2
        bne     s9,s8,.L124
        mv      a5,s2
        cmov2   a5,s3,a5
        addi    a3,s7,%lo(.LANCHOR19)
        addi    a2,a5,128
```

Here the usual load was omitted:

```
.L343:
        lbu     a4,0(s1)
        cmov    s5,a4,a4,s5
```

```
.L343:
        lbu     a4,0(s1)
        cmov2   s5,a4,s5
```

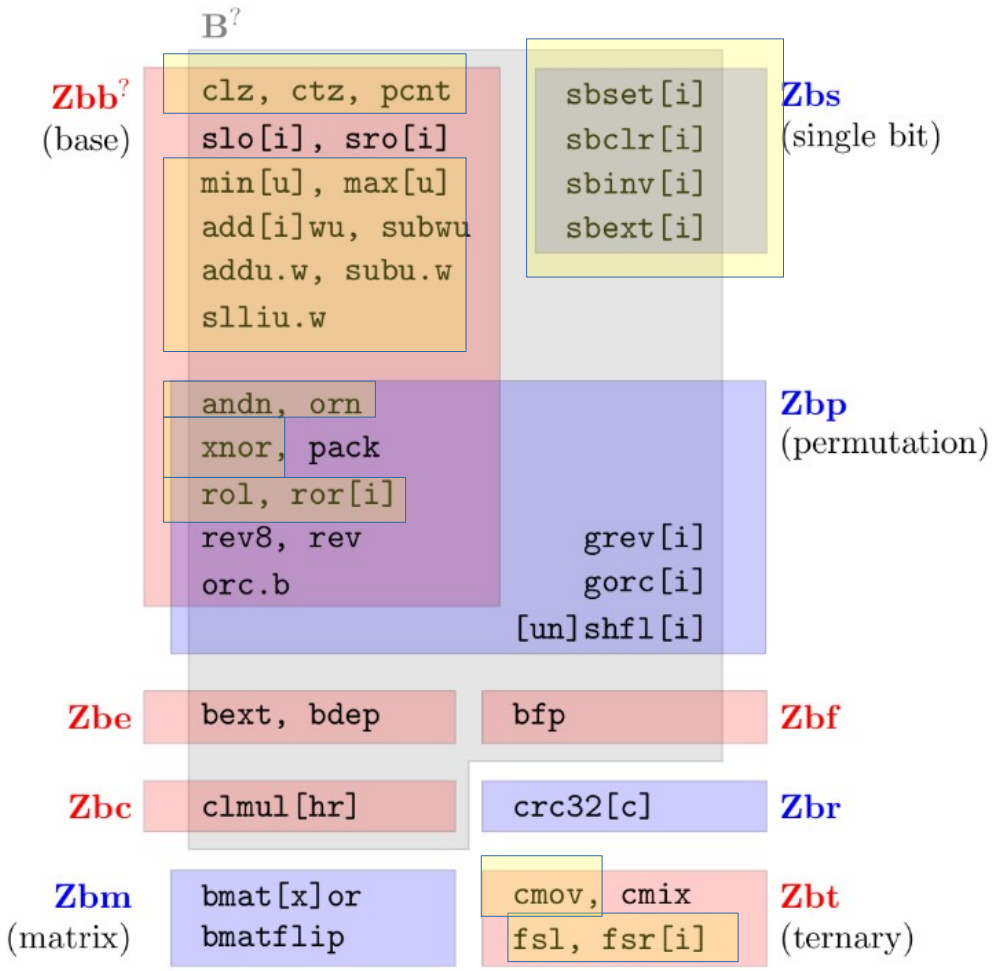
EMBECOSM®

RISC-V Bit Manipulation Instructions

Results:

- 2-input cmov is still useful in avoiding generating expensive branches
- Solves the issue of accommodating potentially tricky third register fetch
- Embench results demonstrate that the compiler can use a 2-input everywhere it used a 3-input cmov

RISC-V Bit Manipulation Instructions



- Still a lot of opportunity for adding code gen to GCC
- Red/blue highlight show insn subset groupings
- Yellow highlight show insns which GCC is capable of generating code-gen patterns for

EMBECOSM®

Questions?

www.embecosm.com