VxWorks on RISC-V:

What we've learned about LLVM RISC-V toolchain

VxWorks 7 RISC-V Architecture Support Status

Wind River is going to officially support RISC-V architecture in the upcoming VxWorks 7 SR0650 release on July 17th.

1. Support RV32I / RV64I plus optional extensions (MAFDC) and arbitrary combinations, SV32/SV39/SV48 + ASID MMU
2. Support using Workbench for graphical debugging kernel and user applications (single stepping, breakpoint, etc)
3. LLVM 10.0.0 + GNU binutils 2.34 toolchain (GCC was supported in last year's EAR, and will be no longer supported in the GA release)
4. Support using QEMU 5.0.0 virt / sifive_u machines to do fast simulation / emulation
5. Support OpenSBI v0.7 + U-Boot v2020.07

Issue 1

Last November, an EAR (Early Access Release) was released to support RISC-V, and the compiler used was GCC 8.3.0. Starting from March this year, we started to switch the compiler to LLVM.

1. Initially LLVM 9.0.0, the latest LLVM release at that time, was used. This version is the first version that officially supports RISC-V.

2. There are some issues with this LLVM release, for example when compiling the following codes:

```
void foo (char* cond)
{
    *cond = 0;
}
```

```
$ clang --target=riscv64 -march=rv64imafdc -mabi=lp64d -funwind-tables -c -o a.o a.c
$ ldriscv -m elf64lriscv --oformat=elf64-littleriscv a.o
ldriscv: error in a.o(.eh_frame); no .eh_frame_hdr table will be created
```

Besides, the -mcmodel that LLVM supports is incompatible with GCC. GCC is using medlow / medany, while LLVM is using small / medium.

3. Later we found that issues like above have already been fixed in the LLVM 10.0.0 release.
After LLVM 10.0.0 was officially released on Mar 24, in late April, we started to shift our work to the latest 10.0.0.

4. Another thing that is worth mentioning is that LLVM behaves differently when handling global variables with initial value of 0.
Unlike GCC which puts these variables into .data section, LLVM is so “smart” to put them all in the .bss section.

Issue 2

Clang crashes when compiling codes that have a call to __builtin_thread_pointer()

1. We have codes like below to get the value of TP register, with the help of __builtin_thread_pointer()

```
void * _start()
{
    char * tp = __builtin_thread_pointer();
    return tp;
}

$ clang --target=riscv64 -march=rv64imafdc -mabi=lp64d -c -o a.o a.c
```

```
fatal error: error in backend: Cannot select: intrinsic %llvm.thread.pointer
...
clang: error: clang frontend command failed with exit code 70 (use -v to
see invocation)
clang: note: diagnostic msg:
********************
PLEASE ATTACH THE FOLLOWING FILES TO THE BUG REPORT:
Preprocessed source(s) and associated run script(s) are located at:
clang: note: diagnostic msg: /tmp/tp-86ca29.c
clang: note: diagnostic msg: /tmp/tp-86ca29.sh
clang: note: diagnostic msg:
********************
```

2. Reported a bug to the LLVM community, see:
https://bugs.llvm.org/show_bug.cgi?id=45303
and submitted the bug fix (https://reviews.llvm.org/D76828) (merged)

3. @lenary from LLVM community discussed this issue with @kito-cheng from GCC community, and they both agreed to introduce __builtin_thread_pointer() for the RISC-V target in GCC and LLVM.

Issue 3

Under certain conditions, clang with “-O2” optimization wrongly generates a “lwu” instruction

1

```
int _start(int * pAddr)
{
    int val;
    int count;

    val = *pAddr;
    count = val & 0x7FFFFFFF;
    __sync_bool_compare_and_swap(pAddr, val, 1);
    return count;
}
```

```
$ clang --target=riscv64 -march=rv64imafdc -mabi=lp64d -O2 -c a.c -o a.o
$ ldriscv -m elf64lriscv --oformat=elf64-littleriscv -m elf64lriscv a.o -o a.out
```

```
00000000000100b0 <_start>:
   100b0:   00056583        lwu     a1,0(a0)
   100b4:   80000637        lui     a2,0x80000
   100b8:   4685            li      a3,1
   100ba:   1605272f        lr.w.aqrl a4,(a0)
   100be:   00b71563        bne     a4,a1,100c8 <_start+0x18>
   100c2:   1ed527af        sc.w.aqrl a5,a3,(a0)
   100c6:   fbf5            bnez    a5,100ba <_start+0xa>
   100c8:   fff6051b        addiw   a0,a2,-1
   100cc:   8d6d            and     a0,a0,a1
   100ce:   8082            ret
```

2

Actually back to February, there was a bug fix submitted in the community:
[LegalizeTypes][RISCV] Correctly sign-extend comparison for ATOMIC_CMP_XCHG (https://reviews.llvm.org/D74453)

Backporting this fix resolved the problem.

Issue 4

An optimization introduced in LLVM 10.0.0 leads to an UB issue exposed in our dynamic linker
https://github.com/NetBSD/src/blob/trunk/libexec/ld.elf_so/headers.c#L404

1

```
404         } else if (use_pltrela) {
405                 obj->pltrela = (const Elf_Rela *)(obj->relocbase + pltrel);
406                 obj->pltrellim = 0;
407                 obj->pltrelalim = (const Elf_Rela *)(obj->relocbase + pltrel + pltrelsz);
408                 /* On PPC and SPARC, at least, REL(A)SZ may include JMPREL.
409                    Trim rel(a)lim to save time later. */
410                 if (obj->relalim && obj->pltrela &&
411                     obj->relalim > obj->pltrela &&
412                     obj->relalim <= obj->pltrelalim)
413                         obj->relalim = obj->pltrela;
414         }
```

```
40079f6:    beqz    a2,4007a04 <_rtld_digest_dynamic+0x314>
40079f8:    bltu    a4,a0,4007a04 <_rtld_digest_dynamic+0x314>
40079fc:    bgeu    a1,a0,4007a04 <_rtld_digest_dynamic+0x314>
4007a00:    beqz    a3,4007a04 <_rtld_digest_dynamic+0x314>
4007a02:    sd      a1,136(s1)
4007a04:    beqz    s10,4007a0e <_rtld_digest_dynamic+0x31e>
```

2

By analyzing the codes, we figured out that a2 is obj->rela, and a3 is obj->relocbase, corresponding to obj->relalim and obj->pltrela in the if statement.

obj->relalim = (const Elf_Rela *)((caddr_t)obj->rela + relasz);
obj->pltrela = (const Elf_Rela *)(obj->relocbase + pltrel);

caddr_t is a pointer to char, so it's a pointer adding an integer. It looks there is no issue, but why when the code executes at the if statement, the integer offset is dropped and it only tests obj->rela and obj->relocbase? It turns out we hit an UB of adding a non-zero offset to a null pointer. So LLVM is very smart to utilize this UB to do an optimization: if (ptr + offset) {…} => if (ptr) {…}. In other words, "ptr + offset" is not NULL if and only if "ptr" is not NULL.

3

This issue cannot be reproduced in LLVM 9.0.0. There was an optimization change in LLVM 10.0.0 that led to the problem found.
[InstCombine] icmp eq/ne (gep inbounds P, Idx..), null -> icmp eq/ne P, null (https://reviews.llvm.org/D66608)
GCC 8.3.0 did not expose this issue, so there is still room for GCC to improve optimization (?) ☺

Issue 5

When disabling M extension, LLVM intrinsics __udivdi3 and __udivmoddi4 call each other to form a cyclic loop

1

```
int _start(unsigned int a, unsigned int b)
{
    if (b != 0)
        return a/b;

    return 0;
}
```

```
$ clang --target=riscv64 -march=rv64iafdc -mabi=lp64d -c a.c -o a.o
$ ldriscv -m elf64lriscv --oformat=elf64-littleriscv -lllvm a.o -o a.out
```

```
00000000000100fc <__udivdi3>:
  ...
  10116:       00e000ef          jal  ra,10124 <__udivmoddi4>
  ...

0000000000010124 <__udivmoddi4>:
  ...
  101a0:       f5dff0ef          jal  ra,100fc <__udivdi3>
  ...
```

2

In fact there was once a bug report in the community back to September 2019:
https://bugs.llvm.org/show_bug.cgi?id=43388
Unfortunately this bug was kept open and remains unfixed.

We submitted the bug fix (https://reviews.llvm.org/D80465) (merged)

Issue 6

When disabling F and D extension, LLVM intrinsics __fixunsdfdi calls itself to form a cyclic loop

1

```
unsigned long long cvt(double x)
{
    return (unsigned long long) x;
}

unsigned long long _start()
{
    return cvt(2.5);
}

$ clang --target=riscv64 -march=rv64imac -mabi=lp64 -c a.c -o a.o
$ ldriscv -m elf64lriscv --oformat=elf64-littleriscv -lllvm a.o -o a.out
```

```
000000000001012c <__fixunsdfdi>:
   ...
   1016e:       15d2            slli    a1,a1,0x34
   10170:       fcb43023        sd      a1,-64(s0)
   10174:       454010ef        jal     ra,115c8 <__divdf3>
   10178:       fb5ff0ef        jal     ra,1012c <__fixunsdfdi>
   1017c:       fca42e23        sw      a0,-36(s0)
   10180:       fe043503        ld      a0,-32(s0)
   10184:       fdc46583        lwu     a1,-36(s0)
   ...
```

2

We submitted the bug fix to the LLVM community:
[RISCV64] emit correct lib call for fp(double) to ui/si：(https://reviews.llvm.org/D80526) (merged)

Alex (@asb) accepted the proposed fix and said:
“Thanks, this looks good to me. I wasn't aware of MakeLibCallOptions and IsSoften - I think I've wanted something like that before.”

Issue 7

1. In the latest LLVM 11.0.0 development branch, there is still no libunwind support for 32-bit RISC-V.

   We have to do it ourslves ^_^
   [RISCV] Support libunwind for riscv32：(https://reviews.llvm.org/D80690)

   This review is still ongoing.

2. -ftrapping-math option crashes clang RISC-V backend
   During testing -ffast-math option this problem was found. -ffast-math enables -fno-trapping-math implicitly, meaning the codes generated can assume no floating exception to be triggered and handled by user (divided by zero, or overflow).

   We submitted the bug fix to the LLVM community:
   [RISCV] Do not crash when using -ftrapping-math：(https://reviews.llvm.org/D81391)

   The review is also in progress.

A bug unrelated to RISC-V: __SOFT_FP__ ?

compiler-rt/lib/builtins/fixunsdfdi.c

1

```
#ifndef __SOFT_FP__
// Support for systems that have hardware floating-point; can set the invalid
// flag as a side-effect of computation.
...

#else
// Support for systems that don't have hardware floating-point; there are no
// flags to set, and we don't want to code-gen to an unknown soft-float
// implementation.
...

#endif
```

2

We sent an email to the LLVM development mailing list, hoping someone could clarify it:
__SOFT_FP__ not defined when building compiler-rt for Soft Float
(http://lists.llvm.org/pipermail/llvm-dev/2020-June/142129.html)

Saleem Abdulrasool replied that he believed this should be a typo, and the correct macro should be __SOFTFP__
We filed a bug report (https://bugs.llvm.org/show_bug.cgi?id=46294) and submitted the bug fix (https://reviews.llvm.org/D82014).

Summary

After we switched from GCC to LLVM …

1. On the performance part, our kernel performance benchmarking testing results show that there is no significant performance number changes.

   On the static code size (text + data + bss) part, the footprint increased slightly when the compiler is changed from GNU to LLVM,
   by comparing kernel / applications built by the two compilers.

2. Why do we want to switch to LLVM?

   - It costs us a lot to maintain two compilers.

   - The same clang executable can be used to cross-compile objects for different target architectures, significantly reducing the binary size of our products.

Q&A

Wind River Press Release for RISC-V

My GitHub Homepage

Bin Meng, VxWorks Engineering Team, Wind River