SiFive

An introduction of vector support in RISC-V Linux

20200824

Greentime Hu <greentime.hu@sifive.com>
Vincent Chen <vincent.chen@sifive.com>

Outline

- **What is riscv vector and its current status**
- **User space**
  - What ifunc is and how ifunc work
  - What should glibc/libgcc port for vector
- **Kernel space**
  - What should kernel port for vector
  - sigcontext/ptrace/context switch
- **Conclusions**
- **References**

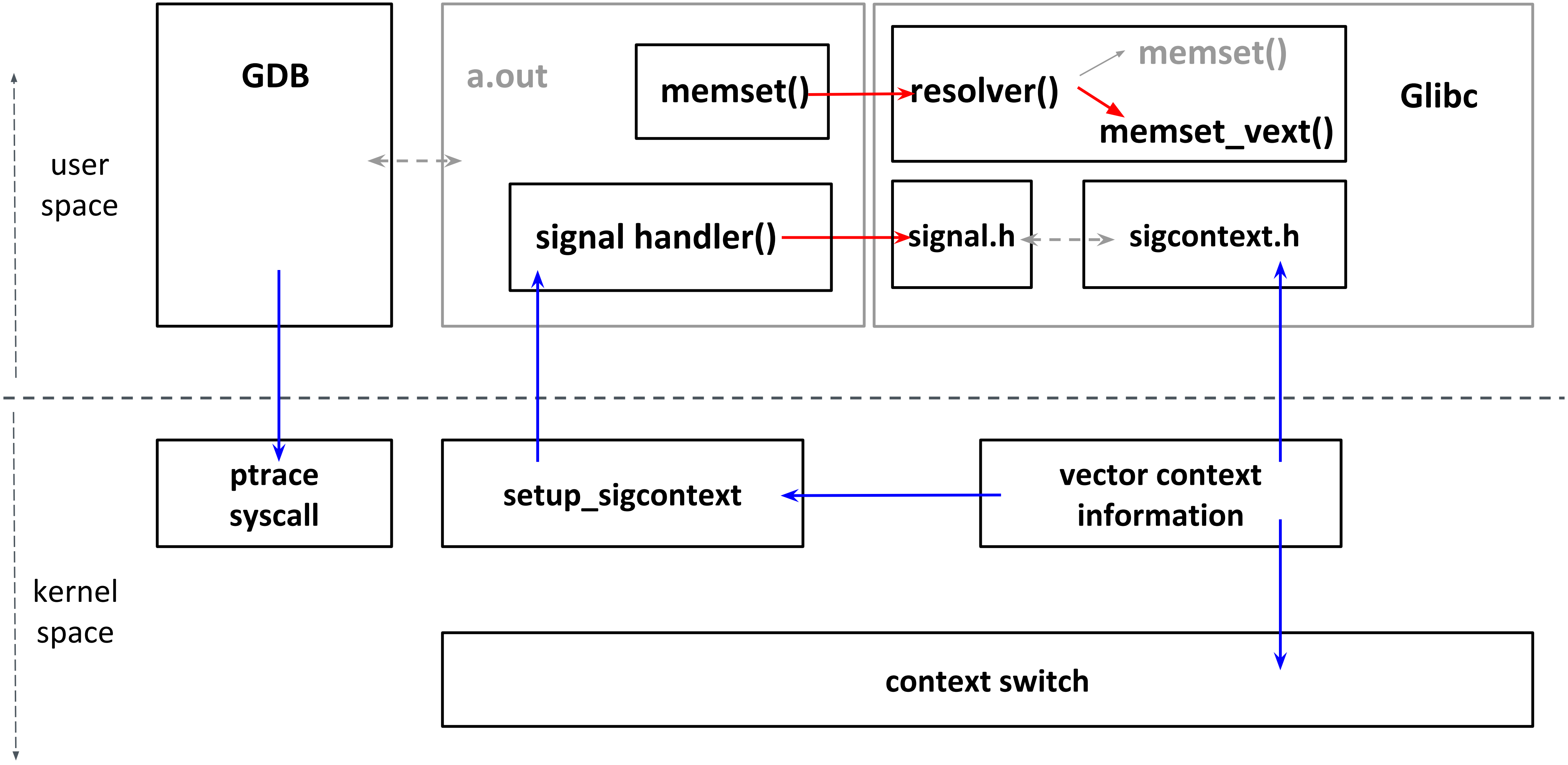
What is riscv vector

- **Quote from The RISC-V Reader**
  - Data-level parallelism
  - Application can compute plenty of data concurrently
  - A more elegant SIMD(single instruction multiple data)
  - **The size of the vector registers is determined by the implementation**, rather than backed into the opcode, as with SIMD
    - The programmer can take advantage of longer vectors **without rewriting the code**
    - Vector architectures have **fewer instructions than SIMD** architectures
- **What is the CSR vlenb**
  - The XLEN-bit-wide read-only CSR vlenb holds the value VLEN/8, i.e., the **vector register length in bytes**.
  - The value in **vlenb is a design-time constant** in any implementation.
  - ex: 512bits vlen => 512/8 = 64 bytes => 64 * 32registers = **2KB**

What is current riscv vector status

- **Spec**
  - v0.9 is released
- **glibc**
  - RFC Patch is sent
    - ifunc
    - Align signcontext header with Linux kernel
  - WIP
    - memcpy/memset/memcmp/memmove/strcmp/strlen...
    - setcontext/getcontext/swapcontext
- **Linux kernel**
  - V6 RFC patchset based on 0.9 spec is sent
  - WIP
    - kernel mode vector
    - lazy vector
- **Tested in spike and QEMU by**
  - user space vector testcases
  - kernel mode XOR optimized by vector instructions
  - stress-ng

Vector porting overview

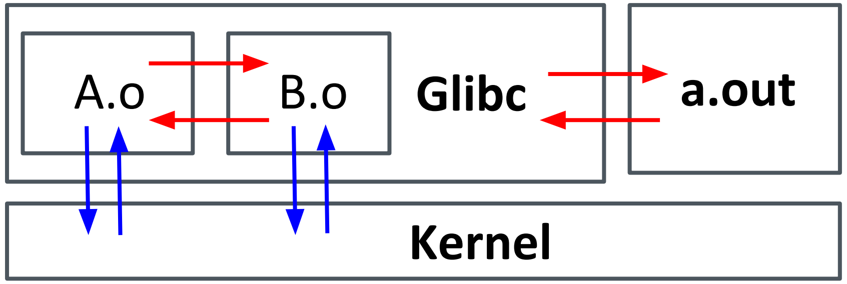


What should glibc/libgcc port for vector

- **Our goals**
  - Integrate the functions optimized by V-extension to the Glibc
  - Make the existing glibc feature work
- **Integrate the functions optimized by V-extension**
  - memcpy, memset, strcmp, … could be optimized by V-extension
  - How to use the optimized functions without sacrificing the Glibc portability?
    - Using GNU indirect function support(a.k.a IFUNC)
      - make Glibc follow user's rule to select an appropriate function based on the hardware capability **in runtime**.
    - Add GNU indirective function (a.k.a IFUNC) support to RISCV
      - Thanks Nelson Chu for adding IFUNC support to GCC and Binutils

Make the existing Glibc feature work

- **Make the existing Glibc feature work**
  - V-extension means new instructions + new registers
    - Most works is to address the ABI issues such as calling convention
      - Between two binaries in User space
      - Between glibc and kernel



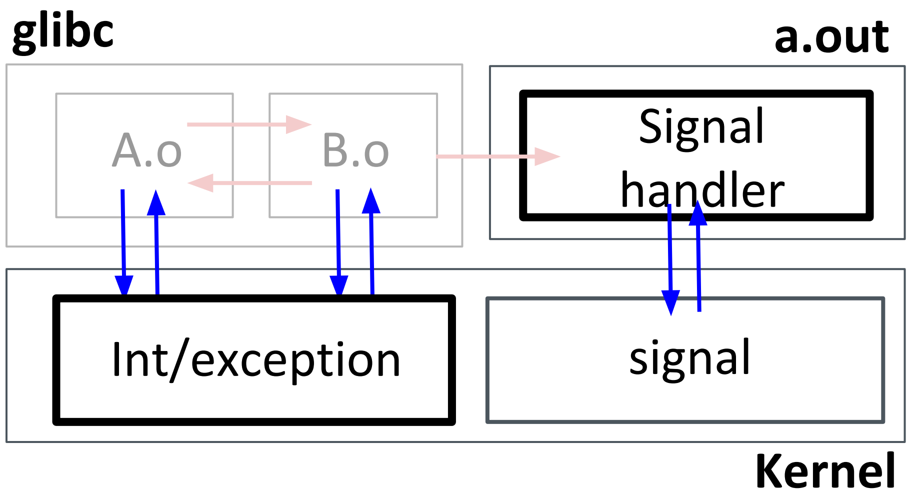
A. ABI issues in two user space binaries

- Check according to the calling convention in v0.9 spec
- Needed to add save/restore mechanism for VCSR in setcontext, getcontext and swapcontext.

Make the existing Glibc feature work

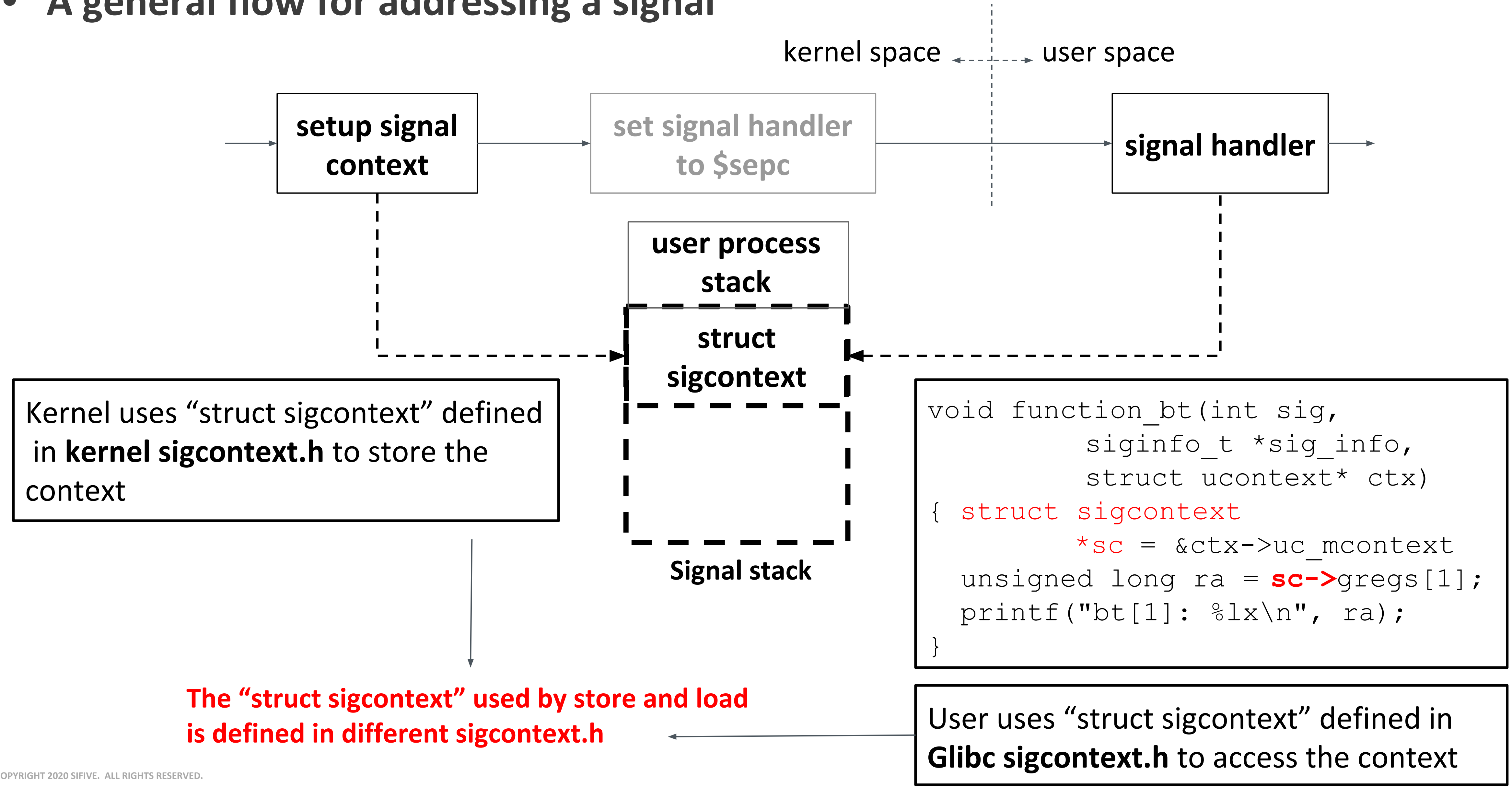
B. ABI issues in kernel and glibc

- Exception, Interrupt
  - Kernel needs to ensure the correctness of vector context
- Signal handler
  - The caller of signal handler caller is kernel
  - Glibc provides user with some information to handle the signal
    - struct sigcontext
    - signal stack



Make the existing Glibc feature work

- A general flow for addressing a signal



Kernel uses “struct sigcontext” defined in **kernel sigcontext.h** to store the context

```
void function_bt(int sig,
          siginfo_t *sig_info,
          struct ucontext* ctx)
{ struct sigcontext
        *sc = &ctx->uc_mcontext
  unsigned long ra = sc->gregs[1];
  printf("bt[1]: %lx\n", ra);
}
```

The “struct sigcontext” used by store and load is defined in different sigcontext.h

User uses “struct sigcontext” defined in **Glibc sigcontext.h** to access the context

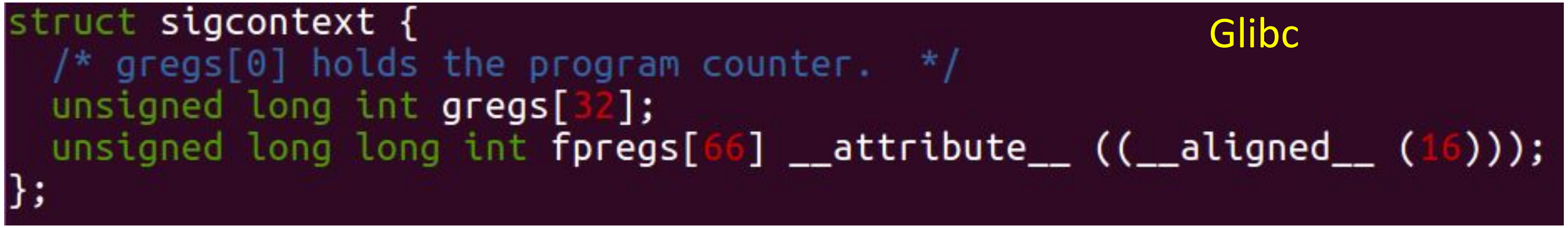
Make the existing Glibc feature work

- **We need to add the vector registers to “struct sigcontext” in Linux and Glibc**
  - Every time we support a new extension, we need to modify the "struct sigcontext" in Linux and Glibc.

- **Could we make glibc include kernel sigcontext.h?**
  - Yes, the Glibc’s generic sigcontext.h already did.
  - However, it may cause ABI incompatibility in RISC-V user program

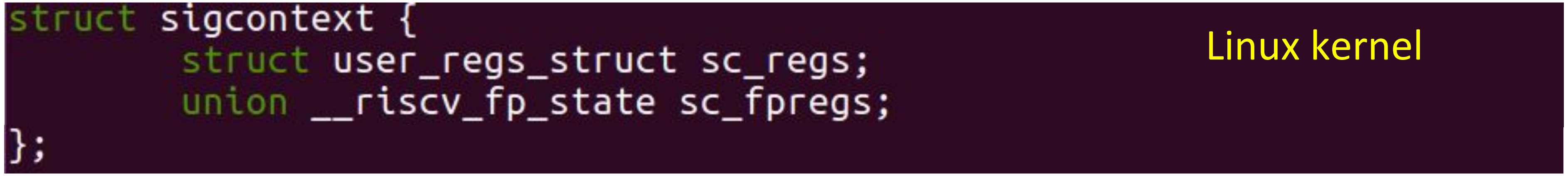
Make the existing Glibc feature work

- **the content of "struct sigcontext" in Glibc is different from the "struct sigcontext" in kernel**
  - Same memory layout, but different element name



```
struct sigcontext {
  /* gregs[0] holds the program counter.  */
  unsigned long int gregs[32];
  unsigned long long int fpregs[66] __attribute__ ((__aligned__ (16)));
};
```

Glibc



```
struct sigcontext {
        struct user_regs_struct sc_regs;
        union __riscv_fp_state sc_fpregs;
};
```

Linux kernel

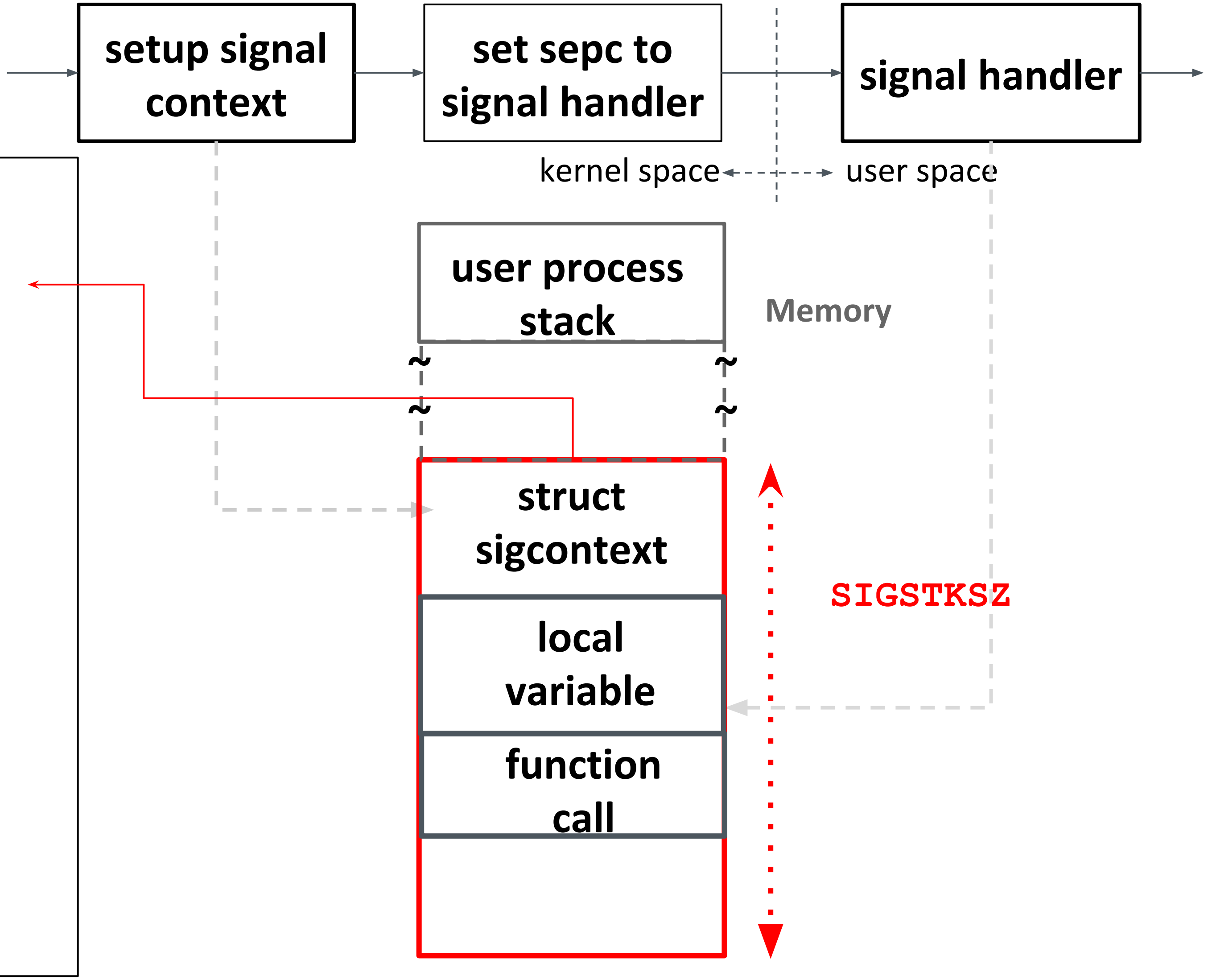
Make the existing Glibc feature work

- **ABI incompatibility in RISC-V user program**
  - The libgcc unwinding scheme cannot work
    - It uses glibc sigcontext to get the return address from signal stack
    - LLVM does’t have this issue because it directly access the sigcontext via memory offset
  - Build multiple images via **OpenEmbedded** to evaluate the impacts in ECO system (Thanks to Jim Wilson’s suggestions)
    - No build error is encountered.
    - **Now may be a good moment to do this**
      - Align the Glibc “struct sigcontext” with kernel
        - » **[RFC PATCH] riscv: remove riscv-specific sigcontext.h**

Make the existing Glibc feature work

- **A general flow for addressing a signal**

```
int main() {

char *stack =
malloc(SIGSTKSZ);

stack_t ss = {
.ss_size = SIGSTKSZ,
.ss_sp = stack, };

struct sigaction sa = {
.sa_handler = handler,
.sa_flags = SA_ONSTACK };

sigaltstack(&ss, 0);

}
```



Make the existing Glibc feature work

- **The signal stack size >= sizeof (struct sigcontext)**
  + sizeof (local variables and function call)
  - POSIX defines two **constant value** for user to allocate a signal stack
    - **MINSIGSTKSZ** : The minimum stack size for a signal handler
      >= sizeof (struct sigcontext)
    - **SIGSTKSZ**: It is a system default specifying the stack size that
      would be used to cover the **usual case**
      (= 3~4 *MINSIGSTKSZ for most architectures)

- **However, the theoretical maximum VLEN is 2^(XLEN -1)**
  - Size of (struct sigcontext) is an astronomical number as well

Make the existing Glibc feature work

- **How to determine MINSIGSTKSZ and SIGSTKSZ?**
  - Assumed maximum VLEN
    - To meet the current mechanism
    - Max VLEN = 4096 bit
  - Kernel pass precise MINSIGSTKSZ to User space via auxiliary entry
    - Reference to ARM64 implementation
      - Create a new auxiliary entry to pass precise MINSIGSTKSZ to User
    - Avoid wasting memory if the real VLEN < max VLEN
    - Break the limitation of “Max VLEN = 4096 bit”
    - [RFC patch] riscv: signal: Report signal frame size to userspace via auxv

What should glibc/libgcc port for vector

- **Conclusion**
  - Add IFUNC support
    - **[RFC PATCH] riscv: Add support for STT_GNU_IFUNC symbols**
  - Add save/restore mechanism for **VCSR** in setcontext/getcontext/swapcontext
  - Align the Glibc “struct sigcontext” with kernel
    - **[RFC PATCH] riscv: remove riscv-specific sigcontext.h**
  - A mechanism to make user get appropriate MINSIGSTKSZ
    - **[RFC PATCH] riscv: signal: Report signal frame size to userspace via auxv**

What should kernel port for vector

- The most important parts
  - riscv: Add vector struct and assembler definitions
  - riscv: Add sigcontext save/restore for vector
  - riscv: Add ptrace vector support
  - riscv: Add task switch support for vector
- And others
  - riscv: signal: Report signal frame size to userspace via auxv
  - riscv: Reset vector register
  - riscv: Add has_vector/riscv_vsize to save vector features.
  - riscv: Add vector feature to compile
  - riscv: Add new csr defines related to vector extension
  - riscv: Extending cpufeature.c to detect V-extension
  - riscv: Rename __switch_to_aux -> fpu
  - riscv: Separate patch for cflags and aflags
  - ptrace: Use regset_size() for dynamic regset

ucontext, sigcontext and vector structure

```
struct ucontext {
        unsigned long    uc_flags;
        struct ucontext  *uc_link;
        stack_t          uc_stack;
        sigset_t         uc_sigmask;
        __u8             __unused[1024 / 8 - sizeof(sigset_t)];
        struct sigcontext uc_mcontext;
};
```

```
struct sigcontext {
        struct user_regs_struct sc_regs;
        union __riscv_fp_state sc_fpregs;
        struct __riscv_v_state sc_vregs;
};
```

```
struct user_regs_struct
{
        unsigned long pc;
        unsigned long ra;
        unsigned long sp;
        unsigned long gp;
        unsigned long tp;
        unsigned long t0;
        unsigned long t1;
        unsigned long t2;
        unsigned long s0;
        …
        unsigned long t3;
        unsigned long t4;
        unsigned long t5;
        unsigned long t6;
};
```

```
struct __riscv_f_ext_state {
        __u32 f[32];
        __u32 fcsr;
};

struct __riscv_d_ext_state {
        __u64 f[32];
        __u32 fcsr;
};

struct __riscv_q_ext_state {
        __u64 f[64];
        __u32 fcsr;
        __u32 reserved[3];
};

union __riscv_fp_state {
        struct __riscv_f_ext_state f;
        struct __riscv_d_ext_state d;
        struct __riscv_q_ext_state q;
};
```

ucontext, sigcontext and vector structure

```
struct ucontext {
        unsigned long    uc_flags;
        struct ucontext  *uc_link;
        stack_t          uc_stack;
        sigset_t         uc_sigmask;
        __u8             __unused[1024 / 8 - sizeof(sigset_t)];
        struct sigcontext uc_mcontext;
};
```

```
struct sigcontext {
        struct user_regs_struct sc_regs;
        union __riscv_fp_state sc_fpregs;
        struct __riscv_v_state sc_vregs;
};
```

```
struct user_regs_struct
{
        unsigned long pc;
        unsigned long ra;
        unsigned long sp;
        unsigned long gp;
        unsigned long tp;
        unsigned long t0;
        unsigned long t1;
        unsigned long t2;
        unsigned long s0;
        …
        unsigned long t3;
        unsigned long t4;
        unsigned long t5;
        unsigned long t6;
};
```

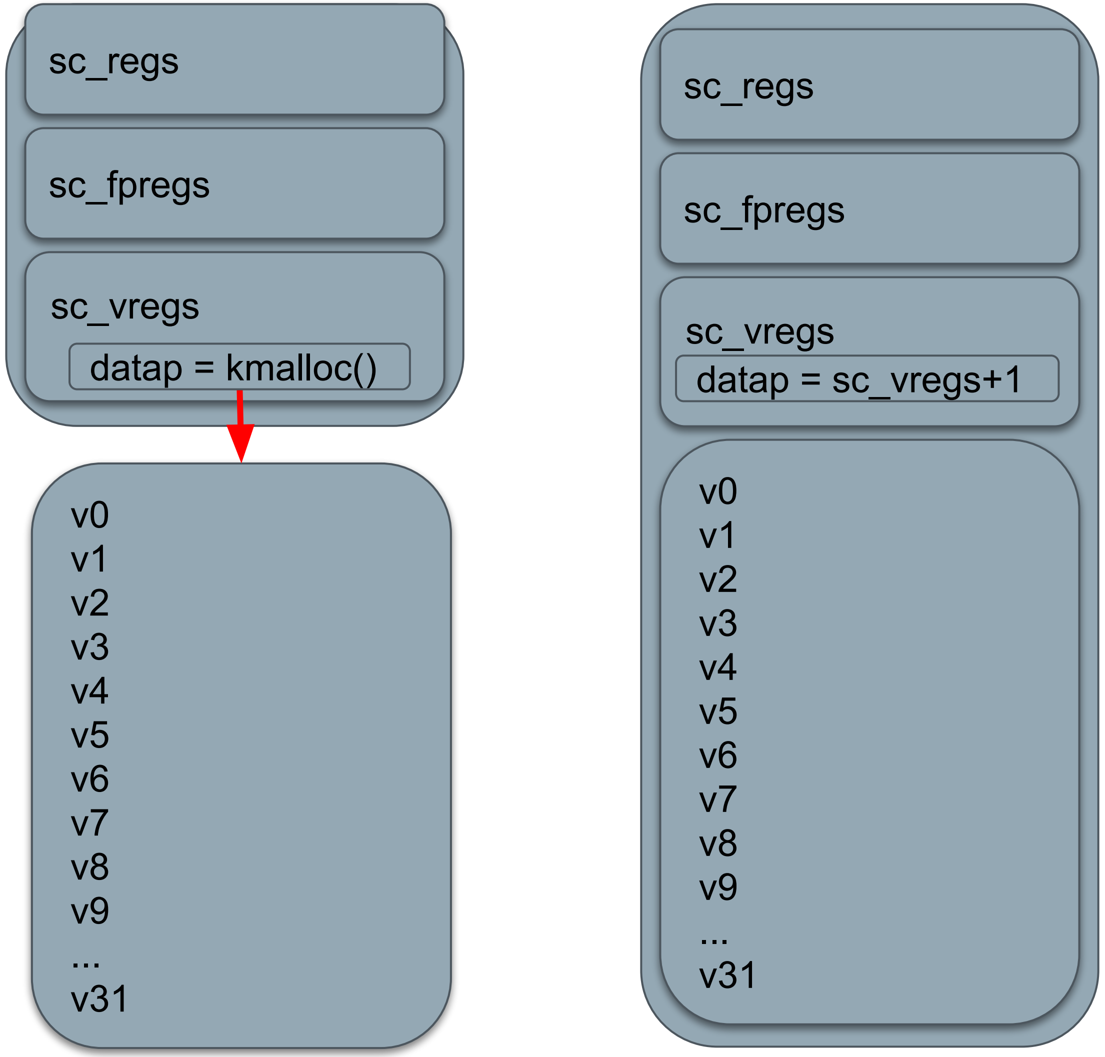
```
struct __riscv_f_ext_state {
        __u32 f[32];
        __u32 fcsr;
};

struct __riscv_d_ext_state {
        __u64 f[32];
        __u32 fcsr;
};

struct __riscv_q_ext_state {
        __u64 f[64];
        __u32 fcsr;
        __u32 reserved[3];
};

union __riscv_fp_state {
        struct __riscv_f_ext_state f;
        struct __riscv_d_ext_state d;
        struct __riscv_q_ext_state q;
};
```

vector structure

```
struct __riscv_v_state {
        __u32 magic;
        __u32 size;    /* size of all vector registers */
        unsigned long vstart;
        unsigned long vl;
        unsigned long vtype;
        unsigned long vcsr;
        void *datap;
#if __riscv_xlen == 32
        __u32 __padding;
#endif
} ;
```

sigcontext layout in kernel

sigcontext layout In user space



sc_regs

sc_fpregs

sc_vregs

datap = kmalloc()

v0
v1
v2
v3
v4
v5
v6
v7
v8
v9
...
v31

sc_regs

sc_fpregs

sc_vregs

datap = sc_vregs+1

v0
v1
v2
v3
v4
v5
v6
v7
v8
v9
...
v31

What should kernel do for signal handler

- **sigcontext.h**
  - void sighandler(int sig, siginfo_t sif, **ucontext_t ctx**)
    - Kernel has to save all registers(including vector registers) to ucontext_t in setup_sigcontext()
      - regs->a0 = ksig->sig; /* a0: signal number */
      - regs->a1 = (unsigned long)(&frame->info); /* a1: siginfo pointer */
      - regs->a2 = (unsigned long)(&**frame->uc**); /* a2: ucontext pointer */
    - Then go to user space sighandler()
  - Sigreturn syscall
    - Kernel has to restore all the registers of ucontext_t to CPU
    - Then resume to user program

What should Linux kernel do for ptrace syscall porting

- **Use ptrace syscall to get vector registers**
  - User have to allocate a space in v_iovec.iov_base for vector csr and registers
    - ptrace(PTRACE_GETREGSET, child, **NT_RISCV_VECTOR**, &v_iovec);
    - ptrace(PTRACE_SETREGSET, child, **NT_RISCV_VECTOR**, &v_iovec);
- **Kernel space to handler ptrace syscall**
  - It copied struct __riscv_v_state and its datap to/from **v_iovec.iov_base**
    - riscv_vr_get() => copy to user
    - riscv_vr_set() => copy from user

Context switch for vector

- **From task_struct to __riscv_v_vstate**

```
struct task_struct task {
        ...
        struct thread_struct thread {
                . . .
                struct __riscv_d_ext_state fstate;
                struct __riscv_v_state vstate; // Save all vector context here
        }
}
```

- **Only partial lazy mechanism for saving or restoring vector registers**
  - When to save
    - If this process is going to be scheduled out and its SR_VS is DIRTY
      - save vector registers to memory, set SR_VS to CLEAN
  - When to restore
    - If this process is going to be scheduled in and its SR_VS is not OFF
      - restore vector registers from memory, set SR_VS to CLEAN

Conclusions

- **Major features are implemented with Spec v0.9 which is very close to v1.0**
- **WIP features**
  - kernel mode vector
  - kernel mode XOR optimization
  - user mode memcpy/memset optimization
  - lazy vector registers save and restore
- **Future work**
  - Native gdb to support vector

References

- v-spec.adoc
- Patterson, David. & Waterman, Andrew. (2017). The RISC-V Reader.