HardenedLinux

Firmware Freedom

coreboot for RISC-V

Xiang Wang & Shawn Chang

Self intro

- Xiang Wang
- Day job at TYA infotech
- Member of HardenedLinux

Self intro

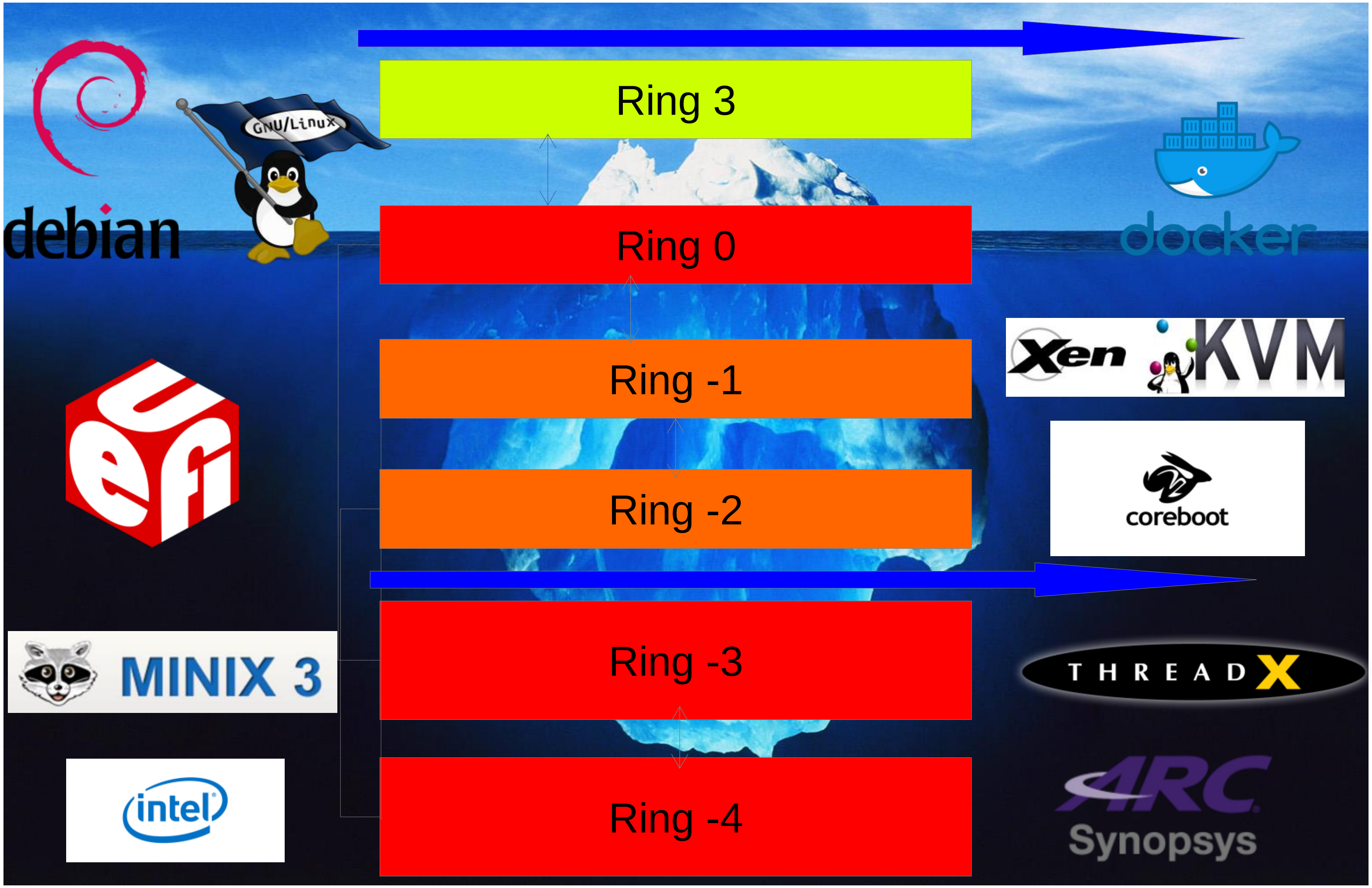
- Shawn C[a.k.a "citypw"]
- Day job at TYA infotech
- Open source security consulting
- GNU/Linux security engineer
- Free/libre SW/FW/HW enthusaists
- Member of EFF/FSF/FSFE/RISC-V
- Patient Zer0 at HardenedLinux community(https://hardenedlinux.github.io/)

Agenda

- What’s wrong with x86 firmware
- Intro to RISC-V & coreboot
- Porting story
- Thank
- Conclusion

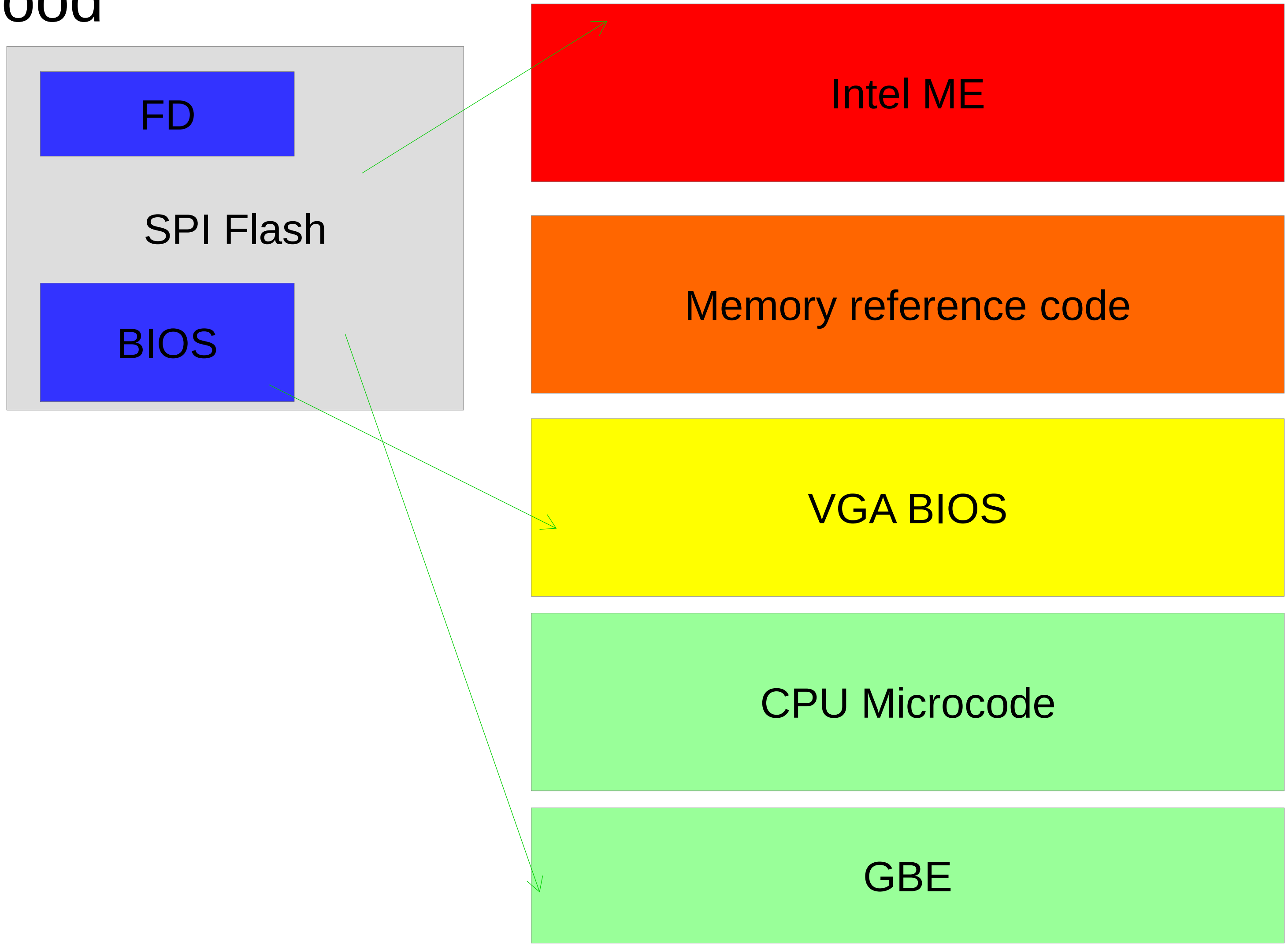
Wait, which CORE exactly?

RING 0 is **not** the CORE anymore



From a libre FW's perspective

Not good



Summary

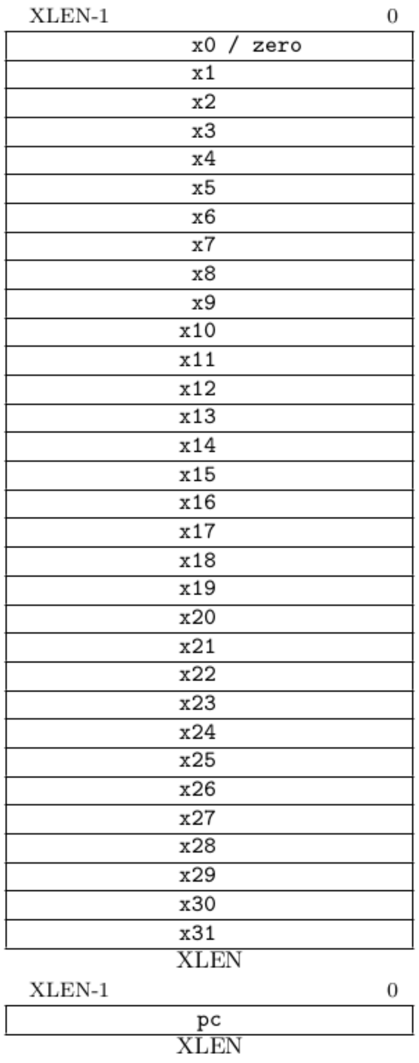
- [Hardening the COREs] solution has its limits in x86
- Too many things can not be audited

Intro to RISC-V

- ➢Pronounced "risk-five".
- ➢Design by University of California, Berkeley.
- ➢Open ISA under BSD license.
- ➢RISC architecture
- ➢load-store type
- ➢conditional jump without status register
- ➢Variable length of instruction encoding
- ➢Instruction set modular

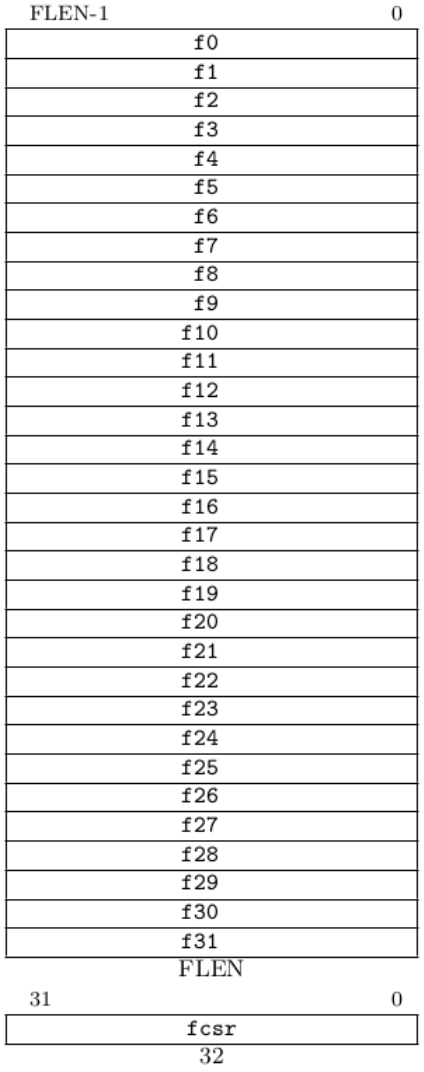
Intro to RISC-V

register



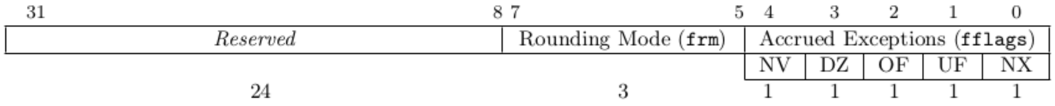
XLEN-1 … 0
x0 / zero
x1
x2
x3
x4
x5
x6
x7
x8
x9
x10
x11
x12
x13
x14
x15
x16
x17
x18
x19
x20
x21
x22
x23
x24
x25
x26
x27
x28
x29
x30
x31
XLEN

XLEN-1 … 0
pc
XLEN

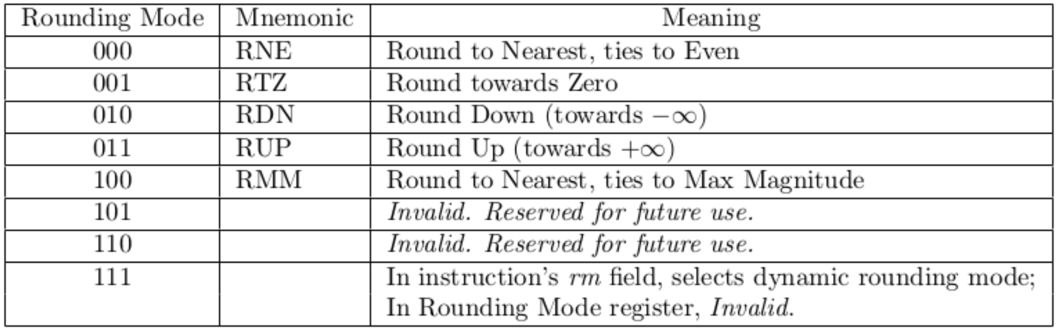


FLEN-1 … 0
f0
f1
f2
f3
f4
f5
f6
f7
f8
f9
f10
f11
f12
f13
f14
f15
f16
f17
f18
f19
f20
f21
f22
f23
f24
f25
f26
f27
f28
f29
f30
f31
FLEN

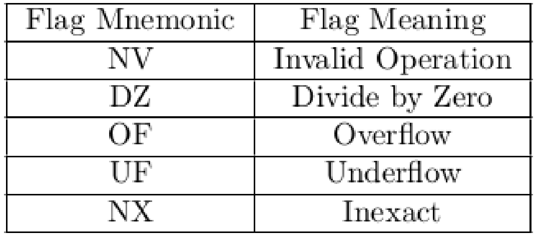
31 … 0
fcsr
32



31 … 8	7 … 5	4	3	2	1	0
Reserved	Rounding Mode (frm)	Accrued Exceptions (fflags)				
		NV	DZ	OF	UF	NX
24	3	1	1	1	1	1



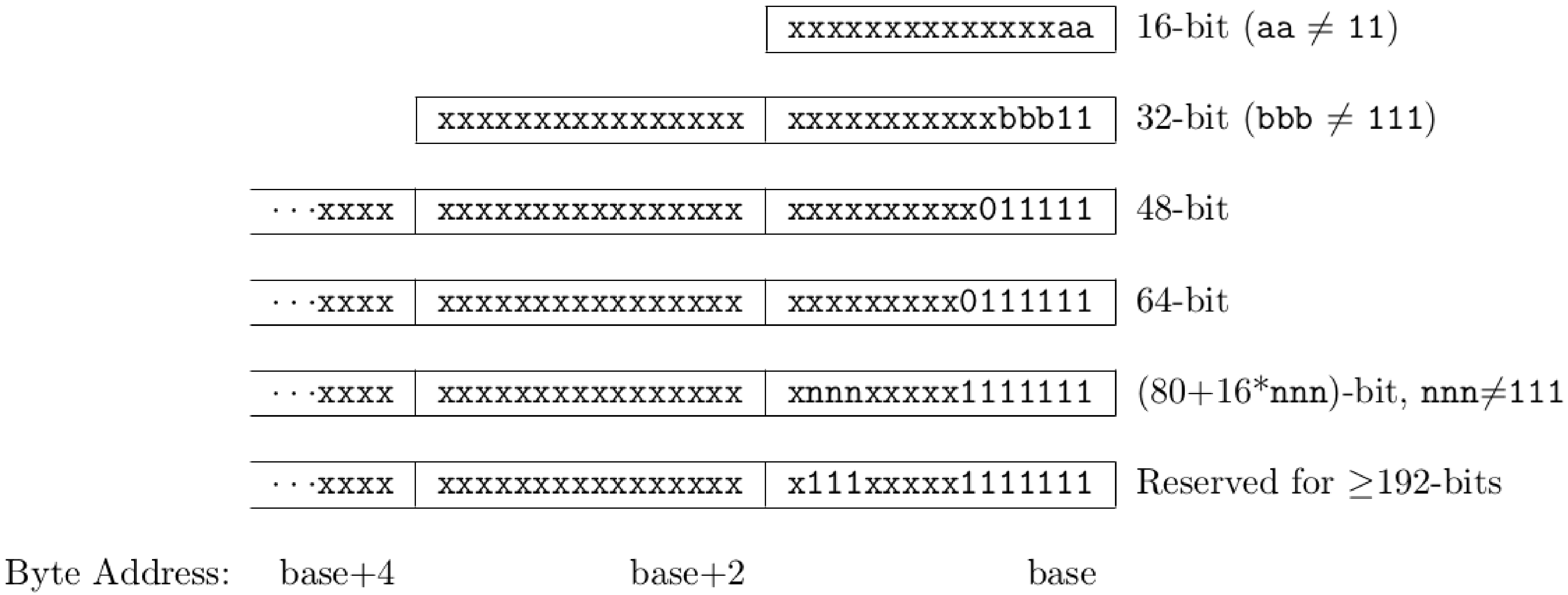
Rounding Mode	Mnemonic	Meaning
000	RNE	Round to Nearest, ties to Even
001	RTZ	Round towards Zero
010	RDN	Round Down (towards $-\infty$)
011	RUP	Round Up (towards $+\infty$)
100	RMM	Round to Nearest, ties to Max Magnitude
101		*Invalid. Reserved for future use.*
110		*Invalid. Reserved for future use.*
111		In instruction's *rm* field, selects dynamic rounding mode; In Rounding Mode register, *Invalid.*



Flag Mnemonic	Flag Meaning
NV	Invalid Operation
DZ	Divide by Zero
OF	Overflow
UF	Underflow
NX	Inexact

Intro to RISC-V

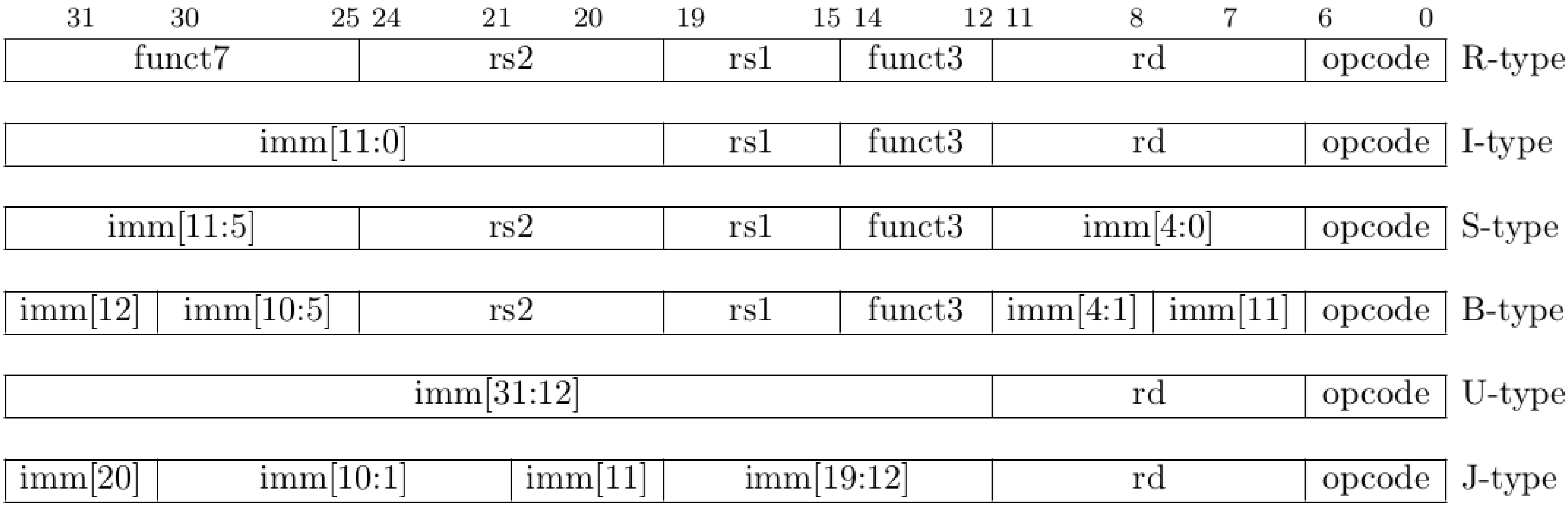
Variable length of instruction encoding



			xxxxxxxxxxxxxxaa	16-bit (aa $\neq$ 11)
		xxxxxxxxxxxxxxxx	xxxxxxxxxxxbbb11	32-bit (bbb $\neq$ 111)
	· · ·xxxx	xxxxxxxxxxxxxxxx	xxxxxxxxxx011111	48-bit
	· · ·xxxx	xxxxxxxxxxxxxxxx	xxxxxxxxx0111111	64-bit
	· · ·xxxx	xxxxxxxxxxxxxxxx	xnnnxxxxx1111111	(80+16*nnn)-bit, nnn$\neq$111
	· · ·xxxx	xxxxxxxxxxxxxxxx	x111xxxxx1111111	Reserved for $\geq$192-bits
Byte Address:	base+4	base+2	base	

Intro to RISC-V

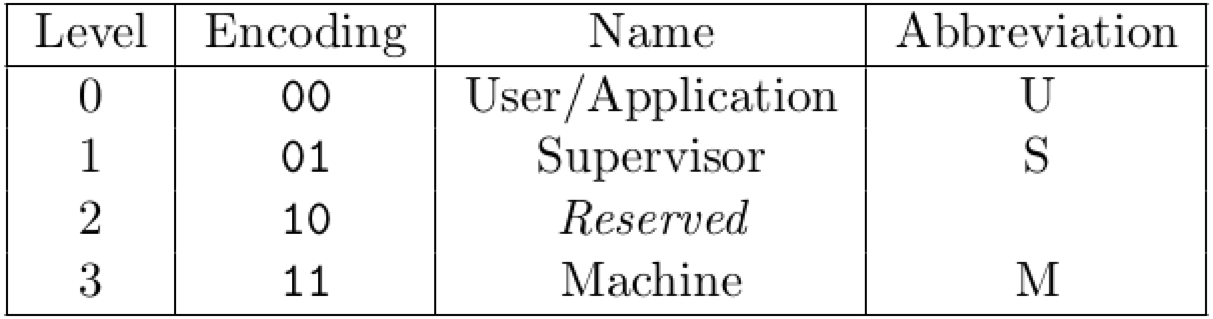
Base Instruct Formats



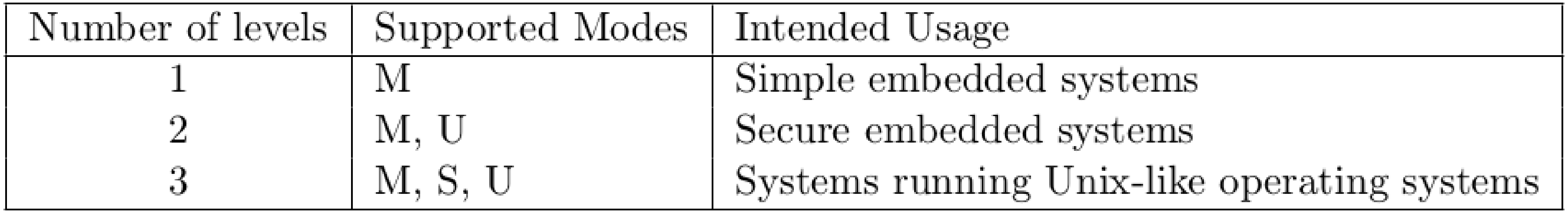
31	30 25	24 21	20	19 15	14 12	11 8	7	6 0	
funct7		rs2		rs1	funct3	rd		opcode	R-type
imm[11:0]				rs1	funct3	rd		opcode	I-type
imm[11:5]		rs2		rs1	funct3	imm[4:0]		opcode	S-type
imm[12]	imm[10:5]	rs2		rs1	funct3	imm[4:1]	imm[11]	opcode	B-type
imm[31:12]						rd		opcode	U-type
imm[20]	imm[10:1]		imm[11]	imm[19:12]		rd		opcode	J-type

Intro to RISC-V

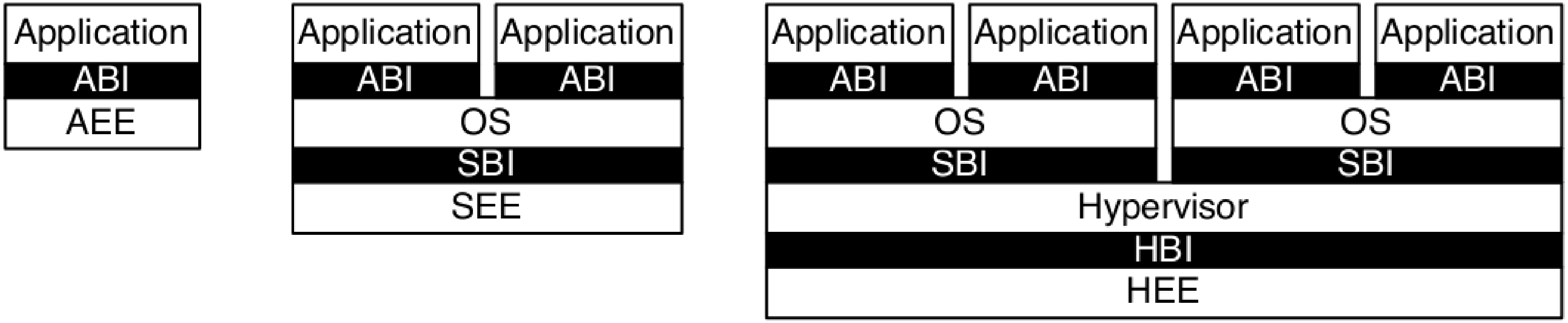
Privilege levels & Privileged Software Stack



Level	Encoding	Name	Abbreviation
0	00	User/Application	U
1	01	Supervisor	S
2	10	*Reserved*	
3	11	Machine	M



Number of levels	Supported Modes	Intended Usage
1	M	Simple embedded systems
2	M, U	Secure embedded systems
3	M, S, U	Systems running Unix-like operating systems



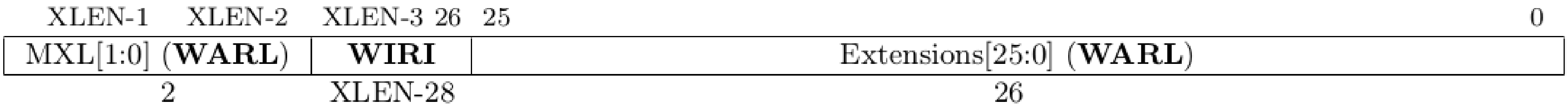
Application
ABI
AEE

Application Application
ABI ABI
OS
SBI
SEE

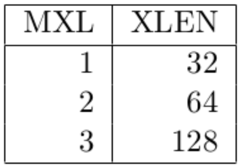
Application Application Application Application
ABI ABI ABI ABI
OS OS
SBI SBI
Hypervisor
HBI
HEE

Intro to RISC-V

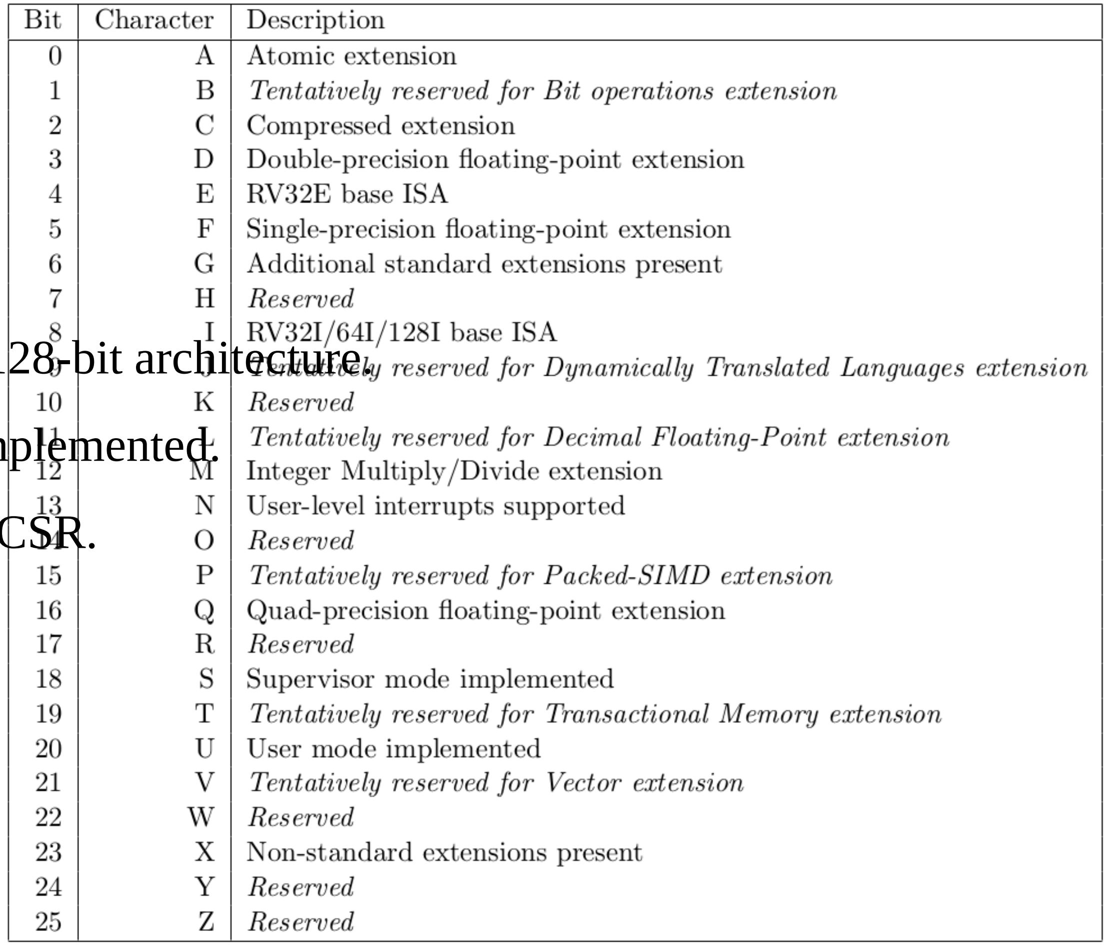
Instruction set modular



XLEN-1 XLEN-2	XLEN-3 26	25 0
MXL[1:0] (**WARL**)	**WIRI**	Extensions[25:0] (**WARL**)
2	XLEN-28	26



MXL	XLEN
1	32
2	64
3	128



Bit	Character	Description
0	A	Atomic extension
1	B	*Tentatively reserved for Bit operations extension*
2	C	Compressed extension
3	D	Double-precision floating-point extension
4	E	RV32E base ISA
5	F	Single-precision floating-point extension
6	G	Additional standard extensions present
7	H	*Reserved*
8	I	RV32I/64I/128I base ISA
9	J	*Tentatively reserved for Dynamically Translated Languages extension*
10	K	*Reserved*
11	L	*Tentatively reserved for Decimal Floating-Point extension*
12	M	Integer Multiply/Divide extension
13	N	User-level interrupts supported
14	O	*Reserved*
15	P	*Tentatively reserved for Packed-SIMD extension*
16	Q	Quad-precision floating-point extension
17	R	*Reserved*
18	S	Supervisor mode implemented
19	T	*Tentatively reserved for Transactional Memory extension*
20	U	User mode implemented
21	V	*Tentatively reserved for Vector extension*
22	W	*Reserved*
23	X	Non-standard extensions present
24	Y	*Reserved*
25	Z	*Reserved*

- RISC-V can be implemented as 32/64/128-bit architecture.
- Only Interger instruction set must be implemented.
- Instruction info can be read from **misa** CSR.

Intro to coreboot

- Began in winter of 1999 in LANL(Los Alamos National Laboratory)
- formerly known as LinuxBIOS
- Armed at replacing proprietary firmware
- Open source under GPLv2
- Platform support : x86,ARM,MIPS,RISC-V and so on
- Written in mostly C, and about 1% in assembly.

Design of coreboot for RISC-V

- The start up divided into multiple stage.
- Bootblock stage : initialize flash memory
- Rom stage : initialize memory and chipset
- Ram stage : set environment for payload
- Payload : It can be Linux kernel image (including Heads) or stand-alone EFI executable, such as iPXE, gPXE, GRUB, SeaBIOS, TianoCore an so on.
- Each stage is separate program
- bootblock is located at the reset entry of SoC.
- Other stage loaded by previous stage (which could be utilized by HardenedBoot).

Design of coreboot for RISC-V

bootblock stage

- First, there are a little assembly code to initialize stack point, exception and interrupt.
- Then jmp to frame code(src/lib/bootblock.c), the code export four interface.
- void bootblock_soc_early_init(void);
- void bootblock_mainboard_early_init(void);
- void bootblock_soc_init(void);
- void bootblock_mainboard_init(void);

Design of coreboot for RISC-V

rom stage

- Initialize clock and memory
- Boot next stage

Design of coreboot for RISC-V

ram stage

- Ram stage have a frame code(src/lib/hardwaremain.c).
- Ram stage divided into 12 steps, which is defined by **boot_state_t**.

```
tpedef enum {
        BS_PRE_DEVICE,  BS_DEV_INIT_CHIPS,  BS_DEV_ENUMERATE,
        BS_DEV_RESOURCES,  BS_DEV_ENABLE,  BS_DEV_INIT,
        BS_POST_DEVICE,  BS_OS_RESUME_CHECK,  BS_OS_RESUME,
        BS_WRITE_TABLES,  BS_PAYLOAD_LOAD,  BS_PAYLOAD_BOOT,
} boot_state_t;
```

- Each step is describe by **boot_state**.
- There are three sets of callback functions in boot_state, through which each step is divided into three phases.
    - Entry callbacks
    - State Actions
    - Exit callbacks
- [**State Actions**] defined by system, which mainly used to process device tree.

Design of coreboot for RISC-V

ram stage

- Callbacks can be describe by **boot_state_init_entry**. So we can insert the desired operation in a specific location.

- **boot_state_schedule_static_entries** function used for associate boot_state and callbacks.

-

- **bs_walk_state_machine** function used to iterate **boot_state** array and perform callbacks step by step.

Design of coreboot for RISC-V

struct of firmware file

- Top layout of firmware file is some continuous blocks.
- One of blocks records the layout of blocks.
- This block holds two structures that describe layout information

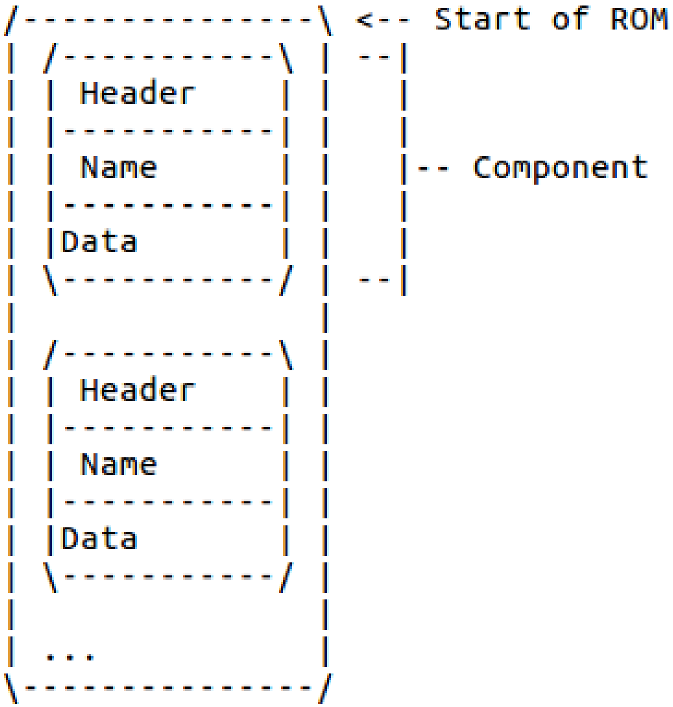
```
struct fmap {
/* "__FMAP__" (0x5F5F464D41505F5F) */
                uint8_t  signature[8];
                uint8_t  ver_major;
                uint8_t  ver_minor;
/* address of the firmware binary */
                uint64_t base;
/* size of firmware binary in bytes */
                uint32_t size;
/* name of this firmware binary */
                uint8_t  name[FMAP_STRLEN];
                uint16_t nareas;
                struct fmap_area areas[];
} __attribute__((packed));
```

```
struct fmap_area {
/* offset relative to base */
                uint32_t offset;
/* size in bytes */
                uint32_t size;
/* descriptive name */
                uint8_t  name[FMAP_STRLEN];
/* flags for this area */
                uint16_t flags;
}  __attribute__((packed));
```

Design of coreboot for RISC-V

struct of firmware file

- A block named **COREBOOT** used to store cbfs.
- **cbfs** consists of components.



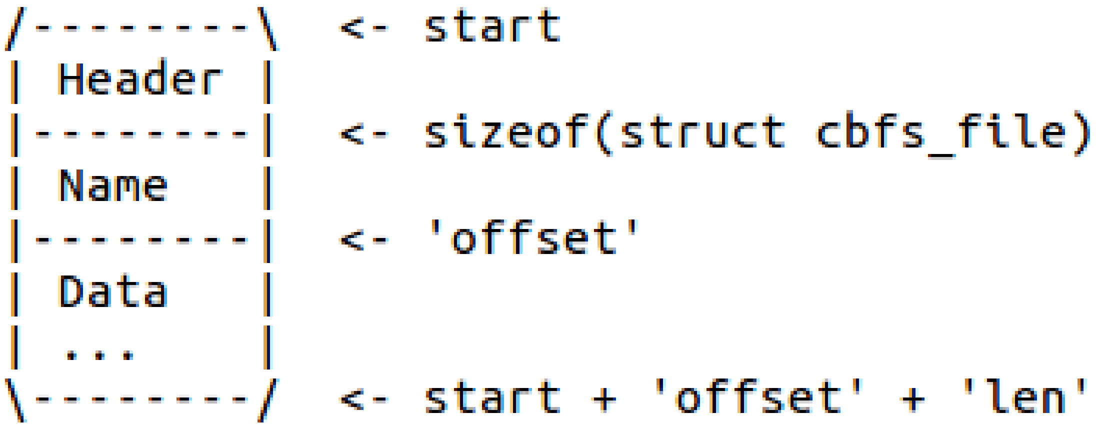
```
/---------------\ <-- Start of ROM
| /-----------\ |  --|
| | Header    | |    |
| |-----------| |    |
| | Name      | |    |-- Component
| |-----------| |    |
| |Data       | |    |
| \-----------/ |  --|
|               |
| /-----------\ |
| | Header    | |
| |-----------| |
| | Name      | |
| |-----------| |
| |Data       | |
| \-----------/ |
|               |
| ...           |
\---------------/
```

Design of coreboot for RISC-V

struct of firmware file

- Component structure as shown.

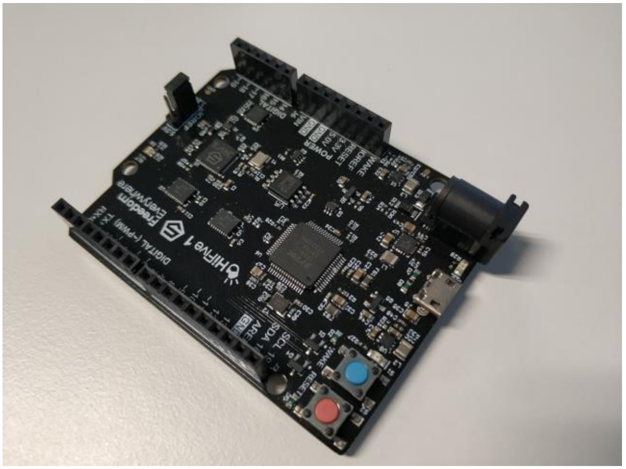
```
struct cbfs_file {
		uint8_t magic[8];
		uint32_t len;
		uint32_t type;
		uint32_t attributes_offset;
		uint32_t offset;
		char filename[];
} __PACKED;
```



```
/---------\   <- start
| Header  |
|---------|   <- sizeof(struct cbfs_file)
| Name    |
|---------|   <- 'offset'
| Data    |
| ...     |
\---------/   <- start + 'offset' + 'len'
```

Porting story board info

- Board: HiFive1
- SoC: SiFive Freedom E310 (FE310)
- Architecture: RV32IMAC
- Speed: 320MHz
- Memory: 16KB instruction cache, 16KB data scratchpad
- Flash Memory: 16Mbit SPI Flash
- Host Interface(microUSB FTD2232HL): P Serial Communication



- SPI controller
- External Interrupt pin: 19
- External Wakeup pin: 1

Porting story
Problem & Scheme

- Problem
- Source of coreboot base on RV64
- Ram is to little, can’t load firmware to memory
- Solution
- Modify build option to support RV32.
- Delete firmware file structure. Layout program directly in flash.

Porting story

build option

- ➢Create directory :
- ➢src/soc/sifive
- ➢src/mainboard/sifive
- ➢Create **Kconfig** in soc directory, add build option in it.
- ➢Modify src/arch/riscv/Makefile.inc pass macro to toolchain.

```
config SOC_SIFIVE_E300
    ......
    bool
    default n
if SOC_SIFIVE_E300
    config RISCV_ARCH
        string
        default "rv32imac"
    config RISCV_ABI
        string
        default "ilp32"
endif
```

```
riscv_flags = ... -march=$(CONFIG_RISCV_ARCH) -mabi=$(CONFIG_RISCV_ABI)
riscv_asm_flags = -march=$(CONFIG_RISCV_ARCH) -mabi=$(CONFIG_RISCV_ABI)
```

Porting story

memory layout

- ➢Linker script (src/lib/program.ld) and memory layout (src/include/memlayout.h) limit code and data segments must be adjacent.

- ➢FE310 program run in flash memory, not closely data segments.

```
#define RAM_START   0x80000000
#define FLASH_START 0x20400000
SECTIONS
{
    BOOTBLOCK(       FLASH_START, 64K)
    ROMSTAGE ( FLASH_START + 64K, 64K)
    RAMSTAGE (FLASH_START + 128K, 64K)

    SRAM_START          (RAM_START)
    STACK               (RAM_START + 4K, 4K)
    PAGETABLES          (RAM_START + 8K, 4K)
    PRERAM_CBMEM_CONSOLE(RAM_START + 12K, 4K)
    SRAM_END            (RAM_START + 16k)
}
```

Porting story
console support

- ➢Reference BSP code and src/console/console.c
- ➢Define macro HAVE_UART_SPECIAL
- ➢Then implement five functions

```
void    uart_init(int idx);
uint8_t uart_rx_byte(int idx);
void    uart_tx_byte(int idx,unsigned char data);
void    uart_tx_flush(int idx);
void    uart_fill_lb(void *data);
```

Porting story

jump to next stage

- ➢Because know where each stage is in memory, so can use jump instructions to jump directly to the next stage.
- ➢Can use function pointer to do this in C language.
- ➢Ex: ((void (*)(void))0x20410000)();

Porting story

link

- Convert each stage to binary by **objcopy**.
- Pack binary to ELF

```
    .global _start
    .section .text
_start:
    .incbin "bootblock.bin"
```

- link

```
ENTRY(_start)
SECTIONS  {
        .text 0x20400000 :  {
                . = 0x00000;
                bootblock-raw.o(.text)
                . = 0x10000;
                romstage-raw.o(.text)
                . = 0x20000;
                ramstage-raw.o(.text)
                . = 0x30000;
                hello-raw.o(.text)
        }
}
```

Porting story

Download

➢Start **openocd**

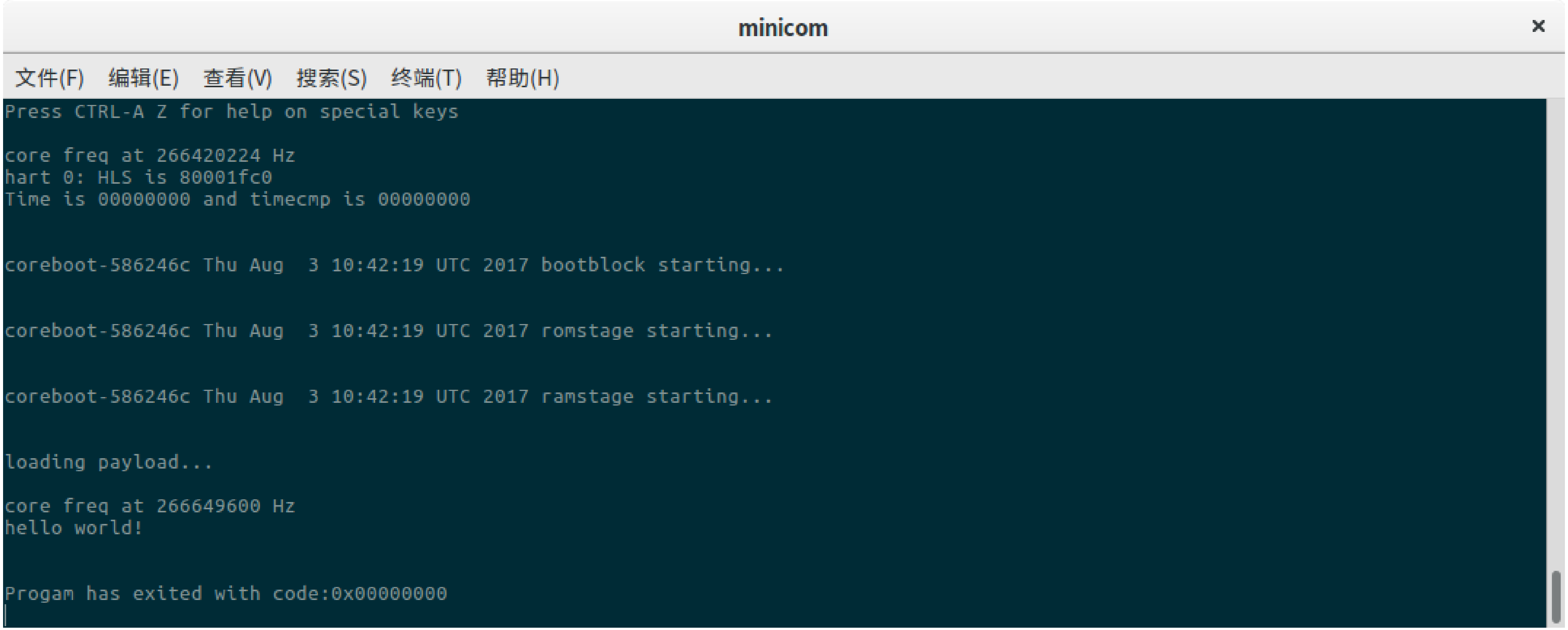
```
openocd -f openocd.cfg >/dev/null 2>&1 &
```

➢Download to board by **gdb**

```
riscv64-unknown-elf-gdb coreboot-raw.elf \
    --batch \
    -ex "set remotetimeout 500" \
    -ex "target extended-remote localhost:3333" \
    -ex "monitor reset halt" \
    -ex "monitor flash protect 0 64 last off" \
    -ex "load" \
    -ex "monitor resume" \
    -ex "monitor shutdown" \
    -ex "quit" && echo Successfully uploaded coreboot-raw.elf
```

Porting story

running screenshorts



```
Press CTRL-A Z for help on special keys

core freq at 266420224 Hz
hart 0: HLS is 80001fc0
Time is 00000000 and timecmp is 00000000


coreboot-586246c Thu Aug  3 10:42:19 UTC 2017 bootblock starting...


coreboot-586246c Thu Aug  3 10:42:19 UTC 2017 romstage starting...


coreboot-586246c Thu Aug  3 10:42:19 UTC 2017 ramstage starting...


loading payload...

core freq at 266649600 Hz
hello world!


Progam has exited with code:0x00000000
```

Porting story

board info

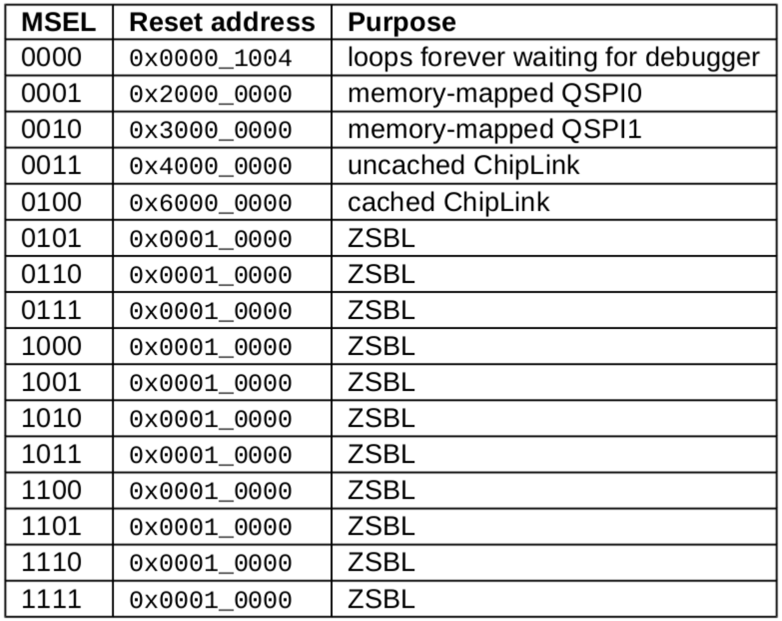
- ➢Board: HiFive-Unleadshed
- ➢SoC: SiFive Freedom U540 (FU540)
- ➢World’s fastest RISC-V Processor
- ➢World’s only GNU/Linux-capable RISC-V SoC
- ➢Architecture: 4xRV64GC + 1xRV64IMAC
- ➢Speed: 1.5GHz
- ➢Cache: Coherent 2M L2 Cache
- ➢Memory: 8G DDR4 with ECC
- ➢Flash: 32MB Quad SPI flash
- ➢MicroSD Card
- ➢Gigabit Ethernet Port

Porting story
original boot procrss

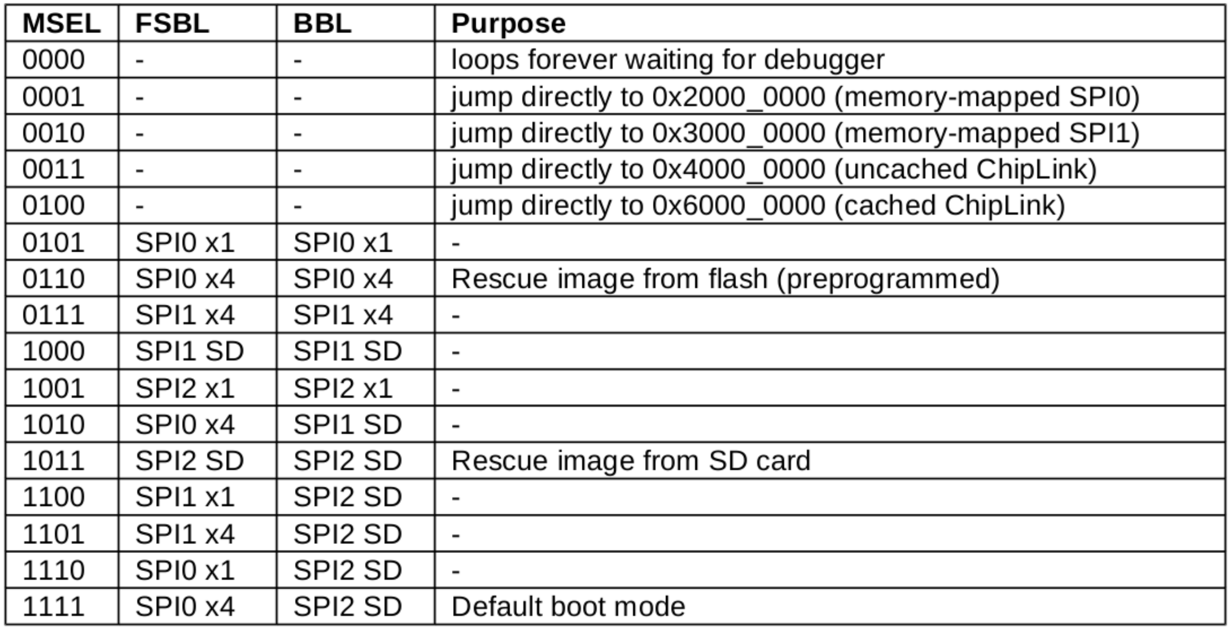
➢SoC has 4 pins called **MSEL** to choose where boot loader is.

➢ZSBL(Zeroth Stage Boot Loader) is stored in the mask ROM of SoC. Download FSBL from a partition with GUID type **5B193300-FC78-40CD-8002-E86C45580B47**.

➢FSBL download BBL from a partition with GUID type **2E54B353-1271-4842-806F-E436D6AF69851**.



MSEL	Reset address	Purpose
0000	0x0000_1004	loops forever waiting for debugger
0001	0x2000_0000	memory-mapped QSPI0
0010	0x3000_0000	memory-mapped QSPI1
0011	0x4000_0000	uncached ChipLink
0100	0x6000_0000	cached ChipLink
0101	0x0001_0000	ZSBL
0110	0x0001_0000	ZSBL
0111	0x0001_0000	ZSBL
1000	0x0001_0000	ZSBL
1001	0x0001_0000	ZSBL
1010	0x0001_0000	ZSBL
1011	0x0001_0000	ZSBL
1100	0x0001_0000	ZSBL
1101	0x0001_0000	ZSBL
1110	0x0001_0000	ZSBL
1111	0x0001_0000	ZSBL

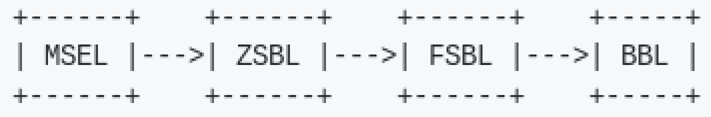


MSEL	FSBL	BBL	Purpose
0000	-	-	loops forever waiting for debugger
0001	-	-	jump directly to 0x2000_0000 (memory-mapped SPI0)
0010	-	-	jump directly to 0x3000_0000 (memory-mapped SPI1)
0011	-	-	jump directly to 0x4000_0000 (uncached ChipLink)
0100	-	-	jump directly to 0x6000_0000 (cached ChipLink)
0101	SPI0 x1	SPI0 x1	-
0110	SPI0 x4	SPI0 x4	Rescue image from flash (preprogrammed)
0111	SPI1 x4	SPI1 x4	-
1000	SPI1 SD	SPI1 SD	-
1001	SPI2 x1	SPI2 x1	-
1010	SPI0 x4	SPI1 SD	-
1011	SPI2 SD	SPI2 SD	Rescue image from SD card
1100	SPI1 x1	SPI2 SD	-
1101	SPI1 x4	SPI2 SD	-
1110	SPI0 x1	SPI2 SD	-
1111	SPI0 x4	SPI2 SD	Default boot mode

Porting story

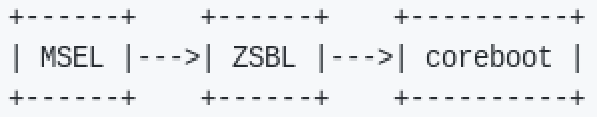
our target

- The original boot process:



```
+-------+    +-------+    +-------+    +------+
| MSEL  |--->| ZSBL  |--->| FSBL  |--->| BBL  |
+-------+    +-------+    +-------+    +------+
```

- Our target should looks like this:



```
+-------+    +-------+    +-----------+
| MSEL  |--->| ZSBL  |--->| coreboot  |
+-------+    +-------+    +-----------+
```

Porting story

our work

- Let the code run on all cores
- How to working with multiple processors
- Hardware related, etc: clock sdram flash sdcard and so on
- DeviceTree: fix mac according to hardware

Porting story

build option

- ➢Management Core RV64IMAC
- ➢Application Cores RV64GC → RV64IMAFDC
- ➢To support the smallest instruct set.

```
riscv_flags = -I$(src)/arch/riscv/ \
                    -mcmodel=$(CONFIG_RISCV_CODEMODEL) \
                    -march=$(CONFIG_RISCV_ARCH) \
                    -mabi=$(CONFIG_RISCV_ABI)
riscv_asm_flags = -march=$(CONFIG_RISCV_ARCH) -mabi=$(CONFIG_RISCV_ABI)
COMPILER_RT_bootblock = $(shell $(GCC_bootblock) $(riscv_flags) -print-libgcc-file-name)
COMPILER_RT_romstage  = $(shell  $(GCC_romstage) $(riscv_flags) -print-libgcc-file-name)
COMPILER_RT_ramstage  = $(shell  $(GCC_ramstage) $(riscv_flags) -print-libgcc-file-name)
```

Porting story

ASMP support

- Solution:
- Minimize initialization after each stage is started, then block other harts for entering a single-threaded state
- Restore multi-threaded state when exiting at each stage

Porting story

ASMP support

- Initialize stack point for each hart.

```
# initialize stack point for each hart
# and the stack must be page-aligned.
# 0xDEADBEEF used to check stack overflow
csrr a0, mhartid
la   t0, _stack
slli t1, a0, RISCV_PGSHIFT
add  t0, t0, t1
li   t1, 0xDEADBEEF
STORE t1, 0(t0)
li   t1, RISCV_PGSIZE - HLS_SIZE
add  sp, t0, t1
```

- Stack is continuous page-aligned memory.
- Hart-local storage(HLS) is a block memory at top of stack.
- HLS can be access by follow code.

```
#define MACHINE_STACK_TOP() ({ \
                register uintptr_t sp asm ("sp"); \
                (void*)((sp + RISCV_PGSIZE) & -RISCV_PGSIZE); })

#define HLS() ((hls_t*)(MACHINE_STACK_TOP() - HLS_SIZE))
#define OTHER_HLS(id) ((hls_t*)((void*)HLS() + RISCV_PGSIZE * ((id) - HLS()->hart_id)))
```

Porting story

ASMP support

- Add a field called entry to HLS and use this memory to communicate between the harts.

```
/* type of entry */
struct blocker {
                void *arg;
                void (*fn)(void *arg);
                atomic_t sync_a;
                atomic_t sync_b;
};
```

Porting story

ASMP support

- Block ASMP

```
void smp_pause(int working_hartid)
{
#define SYNCA (OTHER_HLS(working_hartid)->entry.sync_a)
#define SYNCB (OTHER_HLS(working_hartid)->entry.sync_b)
        int hartid = read_csr(mhartid);
        if (hartid != working_hartid) {
                do {/* waiting for work hart */
                        barrier();
                } while (atomic_read(&SYNCA) != 0x01234567);
                clear_csr(mstatus, MSTATUS_MIE);
                write_csr(mie, MIP_MSIP);
                /* count how many cores enter the halt */
                atomic_add(&SYNCB, 1);
                /* Waiting for interrupt wake up */
                do {
                        barrier();
                        __asm__ volatile ("wfi");
                } while ((read_csr(mip) & MIP_MSIP) == 0);
                set_msip(hartid, 0);
                HLS()->entry.fn(HLS()->entry.arg);
        } else {
                /* Initialize the counter and
                 * mark the work hart into smp_pause */
                atomic_set(&SYNCB, 0);
                atomic_set(&SYNCA, 0x01234567);
                /* waiting for other Hart to enter the halt */
                do {
                        barrier();
                } while (atomic_read(&SYNCB) + 1 < CONFIG_MAX_CPUS);
                /* initialize for the next call */
                atomic_set(&SYNCA, 0);
                atomic_set(&SYNCB, 0);
        }
#undef SYNCA
#undef SYNCB
}
```

Porting story

ASMP support

- Wakeup halt harts

```
void smp_resume(void (*fn)(void *), void *arg)
{
        int hartid = read_csr(mhartid);

        if (fn == NULL)
                die("must pass a non-null function pointer\n");

        for (int i = 0; i < CONFIG_MAX_CPUS; i++) {
                OTHER_HLS(i)->entry.fn = fn;
                OTHER_HLS(i)->entry.arg = arg;
        }

        for (int i = 0; i < CONFIG_MAX_CPUS; i++)
                if (i != hartid)
                        set_msip(i, 1);

        if (HLS()->entry.fn == NULL)
                die("entry fn not set\n");

        HLS()->entry.fn(HLS()->entry.arg);
}
```

Porting story

use coreboot instead of BBL

- Add a GPT head to the bootblock via the python script (util/riscv/sifive-gpt.py)
- This script set the bootblock’s partition type to FSBL(5B193300-FC78-40CD-8002-E86C45580B47)
- Please refer to https://en.wikipedia.org/wiki/GUID_Partition_Table for GPT

Porting story

support Flash or SD Card

- ➢cbfs operates specific devices through region_device

```
void boot_device_init(void);
const struct region_device *boot_device_ro(void);
```

- ➢The specific operation is implemented by region_device->ops (type is struct region_device_ops)

```
/* A region_device operations. */
struct region_device_ops {
        void *(*mmap)(const struct region_device *, size_t, size_t);
        int (*munmap)(const struct region_device *, void *);
        ssize_t (*readat)(const struct region_device *, void *, size_t, size_t);
        ssize_t (*writeat)(const struct region_device *, const void *, size_t,
                size_t);
        ssize_t (*eraseat)(const struct region_device *, size_t, size_t);
};
```

Porting story

support Flash or SD Card

➢coreboot implements code(region_device_ops) for memory mapped devices

```
#define MEM_REGION_DEV_RO_INIT(base_, size_)                                         \
                     MEM_REGION_DEV_INIT(base_, size_, &mem_rdev_ro_ops)

#define MEM_REGION_DEV_RW_INIT(base_, size_)                                         \
                     MEM_REGION_DEV_INIT(base_, size_, &mem_rdev_rw_ops)
```

Porting story

support Flash or SD Card

- ➢Non-memory mapped devices need to implement region_device_ops
- ➢coreboot implements the MMAP_HELPER_REGION_INIT macro to help build region_device

```
#define MMAP_HELPER_REGION_INIT(ops_, offset_, size_)                    \
        {                                                                        \
                .rdev = REGION_DEV_INIT((ops_), (offset_), (size_)),    \
        }
```

Porting story

fix fdt(flattened device tree)

- fdt in maskrom of soc cannot be used to start linux
- I get the correct fdt by modifying bbl to dumping out
- Correct the mac address in fdt by reading otp

Porting story

SBI support

- coreboot uses bbl as payload (runs in M mode)
- bbl provides SBI support, and packages the kernel to output an elf file

What need firmware to do

- Hardware-related initialization
- Initialize SBI(supervisor binary interface)

Interface of SBI

- Trigger SBI via **ecall** instruction
- a0,a1 and a2 registers is used to parameter passing
- a7 register is used to select function
- a0 used to pass back the result

Functions of SBI

- SBI function used by current linux

```
#define SBI_SET_TIMER 0
#define SBI_CONSOLE_PUTCHAR 1
#define SBI_CONSOLE_GETCHAR 2
#define SBI_CLEAR_IPI 3
#define SBI_SEND_IPI 4
#define SBI_REMOTE_FENCE_I 5
#define SBI_REMOTE_SFENCE_VMA 6
#define SBI_REMOTE_SFENCE_VMA_ASID 7
#define SBI_SHUTDOWN 8
```

SBI_SET_TIMER

- ➢It used to set timed events.

- ➢Appear in

```
drivers/clocksource/riscv_timer.c
    static int next_event(unsigned long delta, struct clock_event_device *ce);
```

SBI_CONSOLE_PUTCAHR
SBI_CONSOLE_GETCHAR

- It used to early printk. Appear in

```
arch/riscv/kernel/setup.c
    static void sbi_console_write(struct console *co, const char *buf, unsigned int n);
    struct console riscv_sbi_early_console_dev;
    void __init setup_arch(char **cmdline_p);
kernel/printk/printk.c
    struct console *early_console;
    asmlinkage __visible void early_printk(const char *fmt, ...);
```

- It used for tty driver. Appear in

```
drivers/tty/hvc/hvc_riscv_sbi.c
```

SBI_SEND_IPI

- It used to scheduling hart. Appear in

  arch/riscv/kernel/smp.c

sbi_remote_fence_i sbi_remote_sfence_vma sbi_remote_sfence_vma_asid

- It used to operate cache. Appear in

  arch/riscv/include/asm/cacheflush.h
  arch/riscv/include/asm/tlbflush.h

Conclusion

- ➢From firmware freedom’s perspective, x86 is pretty much dead unless Intel open everything which is unlikely.
- ➢Firmware freedom can be benefits from the nature of RISC-V.
- ➢We should support free/libre firmware projects like coreboot in the 1st place.

Thank

- Jonathan Neuschäfer <j.neuschaefer@gmx.net>
- Philipp Hug <philipp@hug.cx>
- Ronald Minnich <rminnich@gmail.com>

- Special thanks to SiFive .Inc open sourced the firmware(dram init and clock included) for libre firmware community.

References

- ➢https://riscv.org/
- ➢https://coreboot.org/
- ➢https://www.sifive.com/products/hifive1/
- ➢https://www.sifive.com/products/hifive-unleashed/
- ➢https://github.com/sifive/freedom-u540-c000-bootloader
- ➢Notes for HiFive Unleashed to run linux with coreboot
- ➢Try Harder 2 Hardening the COREs
- ➢……

Q&A