RISCV RV32IM Programming Model

General Registers

x0 (zero)	Always 0
x1 (ra)	Return Address
x2 (sp)	Stack Pointer
x3 (gp)	Global Pointer
x4 (tp)	Thread Pointer
x5 (t0)	
x6 (t1)	
x7 (t2)	
x8 (s0/fp)	Frame Pointer
x9 (s1)	
x10 (a0)	Argument / Return
x11 (a1)	
x12 (a2)	
x13 (a3)	
x14 (a4)	
x15 (a5)	
x16 (a6)	
x17 (a7)	
x18 (s2)	Saved Values
x19 (s3)	
x20 (s4)	
x21 (s5)	
x22 (s6)	
x23 (s7)	
x24 (s8)	
x25 (s9)	
x26 (s10)	
x27 (s11)	
x28 (t3)	Temporaries
x29 (t4)	
x30 (t5)	
x31 (t6)	

pc	Program Counter

32 General Purpose 32bit Registers: x0 - x31

1 Program Counter: pc

x0 always returns 0 on read, discards writes
x1 - x31 have no hardware enforced usage limitations

There are software conventions for the registers and a set of aliases (ra, sp, a0, a1, etc) which align with these conventions.

When a function is called, the first 8 arguments are passed in a0 - a7 and the return address is passed in ra

Functions use a0 and a1 to return 32bit or 64bit values or structs.

Additional arguments or return values are passed via the stack.

sp is the Stack Pointer, and is decremented to make room for new values on the stack, incremented to discard values.
(The stack grows downward, and should always be 16byte aligned)

Functions may freely use ra, a0 - a7, and t0 - t6 (caller-save)

Functions using s0 - s11 must preserve their values (callee-save)

swetland / 2022

RISCV RV32IM Assembly Language Sample

C Source (input to gcc)

```
void* memchr(void const *buf, int c, size_t len)
{
    unsigned char const *b = buf;
    unsigned char        x = (c & 0xff);

    for (size_t i = 0; i < len; i++) {
        if (b[i] == x) {
            return (void *)(b + i);
        }
    }

    return NULL;
}
```

Compared to ARM or X86 Assembly

- No flags (Z,X,V,C) set as side-effects

- Instead, conditional branches compare two registers

```
beq a, b, target  # if a == b goto target
blt a, b, target  # if a < b  goto target
```

- Or you can set a register to 1 or 0 based on a comparison

```
seq z, a, b       # z = (a == b) ? 1 : 0
sge z, a, b       # z = (a >= b) ? 1 : 0
```

RISCV Assembly Language Quick Reference:
https://github.com/swetland/os-workshop/blob/main/docs/riscv-assembly.md
https://github.com/jameslzhu/riscv-card/blob/master/riscv-card.pdf
https://inst.eecs.berkeley.edu/~cs61c/fa17/img/riscvcard.pdf

RV32 Assembly (generated by gcc)
cleaned up for human readability

```
memchr:                  # a0 (ptr), a1 (chr), a2 (len)
    zext.b a1, a1        # zero extend char (and w/ 0xFF)
    beqz   a2, done      # if len == 0 goto done
    mv     a5, a0        # a5 = ptr
    add    a2, a0, a2    # a2 = end (ptr + len)
    j      begin         # goto begin
loop:
    beq    a5, a2, done  # if ptr == end goto done
begin:
    lbu    a4, 0(a5)     # a4 = load byte from a5[0]
    mv     a0, a5        # a0 (retval) = a5 (ptr)
    addi   a5, a5, 1     # a5 = a5 + 1
    bne    a4, a1, loop  # if a4 != chr goto loop
    ret                  # return (pc = ra)
done:
    li     a0, 0         # a0 (retval) = 0
    ret                  # return (pc = ra)
```

64bit unsigned addition

```
addu64:                  # a1,a0 (x), a3,a2 (y)
    mv     a5, a0        # save x.lo
    add    a0, a0, a2    # z.lo = x.lo + y.lo
    sltu   a5, a0, a5    # if z.lo < x.lo: tmp = 1 (else 0)
    add    a1, a1, a3    # z.hi = x.hi + y.hi
    add    a1, a1, a5    # z.hi = z.hi + tmp
    ret                  # return (pc = ra, result in a1,a0)
```

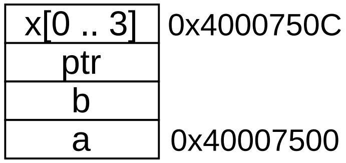
swetland / 2022

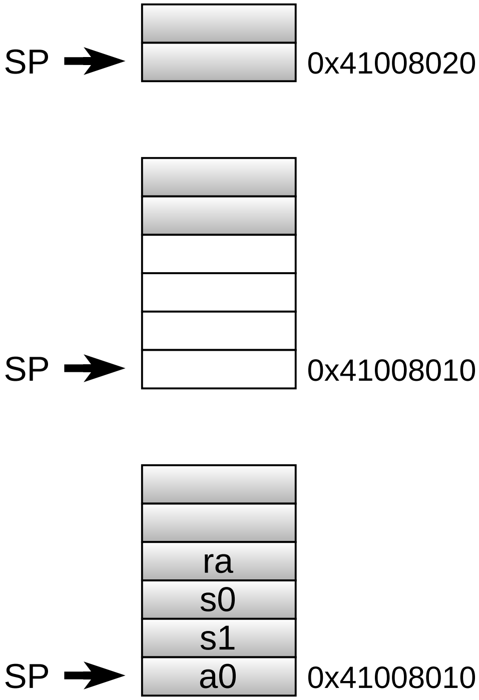
Function Calling Conventions, Stack Conventions, Structures

```
ssize_t write(int fd, const void* buf, size_t count);

# code to call write()

    li a0, 0            # arguments in a0, a1, ...
    la a1, msg
    li a2, 5
    jal write           # ra = next instruction, jump to write
                        # return value in a0
    li t0, 5
    bne a0, t0, error

msg:
    .string "Hello"
```

Structures members are stored in memopry in order (low to high)

```
struct thing {
    uint32_t a;
    uint32_t b;
    void *ptr;
    uint8_t x[4];
};
```





By convention the **sp** register is used as a stack pointer.

The stack provides storage for data that doesn't fit in the registers, or a place to save register contents when they're needed for other purposes.

The stack grows **downward** in memory

Code usually adjusts the stack at the beginning and end of functions, in what are called the function "prologue" and "epilogue".

In the prologue, first, space is made by subtracting some amount from the stack pointer:

```
    addi sp, -16
```

Then registers that need to be saved are written to the stack:

```
    sw ra, 12(sp)   # save return addr
    sw s0, 8(sp)    # save callee-save regs
    sw s1, 4(sp)    # so they may be used
    sw a0, 0(sp)    # save param1 for later
```

It's important that the subtraction happen first -- why?

Some space may be reserved but not used until later on in the function.

In the epilogue, things are reversed: callee-saved registers are restored from the stack, the sp is adjusted back to its original value, and then usually the **ret** instruction is used to return to the caller (address in **ra**)

swetland / 2022

void context_switch(cframe_t* from, cframe_t* to);

The cframe structure records all the callee-save regs:

```
typedef struct {
    uint32_t pc;
    uint32_t sp;
    uint32_t gp;
    uint32_t tp;
    uint32_t s0;
    uint32_t s1;
    uint32_t s2;
    uint32_t s3;
    uint32_t s4;
    uint32_t s5;
    uint32_t s6;
    uint32_t s7;
    uint32_t s8;
    uint32_t s9;
    uint32_t s10;
    uint32_t s11;
} cframe_t;
```

It's part of the thread structure, to hold the register state when the thread is not running:

```
typedef struct {
    uint32_t id;

    // ...

    cframe_t ctxt;
} thread_t;
```

context_switch(from, to) is called to save the current context to the *from* thread.ctxt and restore the new context from the *to* thread.ctxt

It's called with *from*'s stack active, and register contents loaded, but returns with *to*'s stack active and register contents loaded.

We don't need to save the other 16 registers because they are caller-save. The code calling context_switch() will have preserved any that are needed.

```
context_switch:
   // save old context to 'from'
   sw ra, 0x00(a0)
   sw sp, 0x04(a0)
   sw gp, 0x08(a0)
   sw tp, 0x0C(a0)
   sw s0, 0x10(a0)
   sw s1, 0x14(a0)
   sw s2, 0x18(a0)
   sw s3, 0x1C(a0)
   sw s4, 0x20(a0)
   sw s5, 0x24(a0)
   sw s6, 0x28(a0)
   sw s7, 0x2C(a0)
   sw s8, 0x30(a0)
   sw s9, 0x34(a0)
   sw s10, 0x38(a0)
   sw s11, 0x3C(a0)

   // load new context from 'to'
   lw ra, 0x00(a1)
   lw sp, 0x04(a1)
   lw gp, 0x08(a1)
   lw tp, 0x0C(a1)
   lw s0, 0x10(a1)
   lw s1, 0x14(a1)
   lw s2, 0x18(a1)
   lw s3, 0x1C(a1)
   lw s4, 0x20(a1)
   lw s5, 0x24(a1)
   lw s6, 0x28(a1)
   lw s7, 0x2C(a1)
   lw s8, 0x30(a1)
   lw s9, 0x34(a1)
   lw s10, 0x38(a1)
   lw s11, 0x3C(a1)

   // return to new context
   ret
```

More Context Switching (Chickens and Eggs)

So, assuming every thread has a **thread_t** structure

And all the threads eligible to run are in a list or a queue of some sort.

A thread could call a **yield**() or **schedule**() function, which would choose the next thread to run and call **context_switch**() passing the current thread and the newly selected thread as arguments.

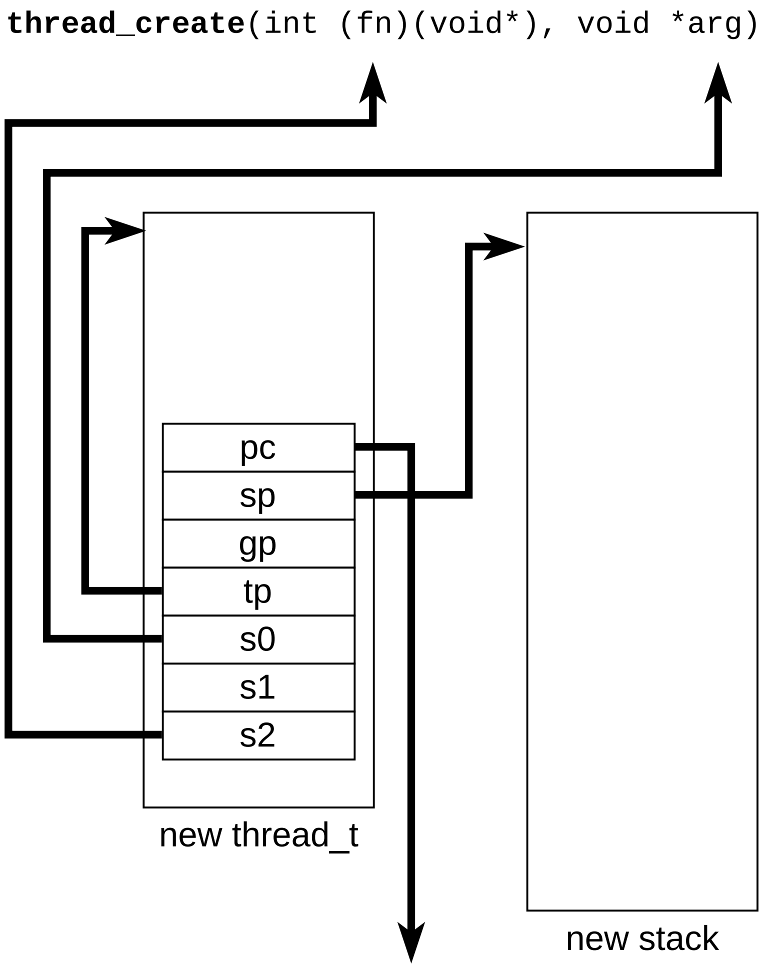
context_switch() will stow the current thread's state (including **pc** and **sp**) in the ctxt in its thread_t, load new state from the new thread, and when it returns the new thread's **pc** and **sp** will be active.

But how do we *start* a thread?

1. We allocate a **thread_t** and some memory for the stack.

2. We manually fill out the ctxt -- with the **sp** field containing the address of the top of the new stack, and the **pc** field containing the address of a helper function, and the **s0**, **s1**, etc fields containing the arguments for and the address of the actual thread function we want to run.

3. Why the helper? Because the context frame doesn't have the argument registers (**a0**, etc) -- so we need a bit of code to shuffle the arguments from **s0** to **a0** before jumping into the real thread function.



thread_create(int (fn)(void*), void *arg)

pc
sp
gp
tp
s0
s1
s2

new thread_t

new stack

```
thread_start_helper:
    mv a0, s0      # load argument

    jalr s2        # call thread fn

    j thread_exit  # exit using the
                   # return value from
                   # thread fn
```

RISCV RV32IM Registers

Litex FPGA SoC Memory

General Registers

x0 (zero)	Always 0
x1 (ra)	Return Address
x2 (sp)	Stack Pointer
x3 (gp)	Global Pointer
x4 (tp)	Thread Pointer
x5 (t0)	
x6 (t1)	
x7 (t2)	
x8 (s0/fp)	Frame Pointer
x9 (s1)	
x10 (a0)	Argument / Return
x11 (a1)	
x12 (a2)	
x13 (a3)	
x14 (a4)	
x15 (a5)	
x16 (a6)	
x17 (a7)	
x18 (s2)	Saved Values
x19 (s3)	
x20 (s4)	
x21 (s5)	
x22 (s6)	
x23 (s7)	
x24 (s8)	
x25 (s9)	
x26 (s10)	
x27 (s11)	
x28 (t3)	Temporaries
x29 (t4)	
x30 (t5)	
x31 (t6)	

pc	Program Counter

Supervisor Registers

SSTATUS	Status / Control
SIE	Interrupt Enable Bitmask
SIP	Interrupt Pending Bitmask
SSCRATCH	Scratch (Software Use)
STVEC	Trap Vector
SEPC	Trap Program Counter
SCAUSE	Trap Cause (bit 31: 1=IRQ 0=Exception)
STVAL	Trap Value
SATP	Address Translation / Protection

Interrupt Cause Codes

1	Supervisor Software IRQ
5	Supervisor Timer IRQ
9	Supervisor External IRQ

Exception Cause Codes

0	Instruction Address Misaligned
1	Instruction Access Fault
2	Illegal Instruction
3	Breakpoint
4	Load Address Misaligned
5	Load Access Fault
6	Store / AMO Address Misaligned
7	Store / AMO Access Fault
8	ECALL from U-Mode
12	Instruction Page Fault
13	Load Page Fault
15	Store / AMO Page Fault

MMIO Peripherals

0xF0005000	SPISDCARD
0xF0004800	SPIFLASH_PHY
0xF0004000	SPIFLASH_CORE
0xF0003800	VTG
0xF0003000	VFB
0xF0002800	UART0
0xF0002000	TIMER0
0xF0001900	SDRAM_CTL
0xF0001000	LEDS
0xF0000800	SOC_IDENT
0xF0000000	SOC_CTRL

System Memory Map

0xF0000000	MMIO
0x80000000	
0x40000000	EXT RAM (32MB)
0x10000000	SRAM (8KB)
0x00000000	ROM / FLASH

swetland / 2022

Trap Entry

An Exception or Interrupt Occurs

Supervisor Interrupt State is saved
SSTATUS.SPIE = SSTATUS.SIE

Supervisor Interrupts are disabled
SSTATUS.SIE = 0

Previous Privilege Level is saved
SSTATUS.SPP = 0 (if User) or 1 (if Supervisor)

Address of current (exception) or
next (interrupt) instruction is saved
SEPC = PC

Exception or Interrupt number is stored
SCAUSE = <reason>

Exception-specific information is stored
STVAL = <value>

Control is transferred to Trap Handler,
at Supervisor Privilege Level
PC = STVEC

Trap Exit

The `sret` instruction is executed

Supervisor Interrupt State is restored
SSTATUS.SIE = SSTATUS.SPIE

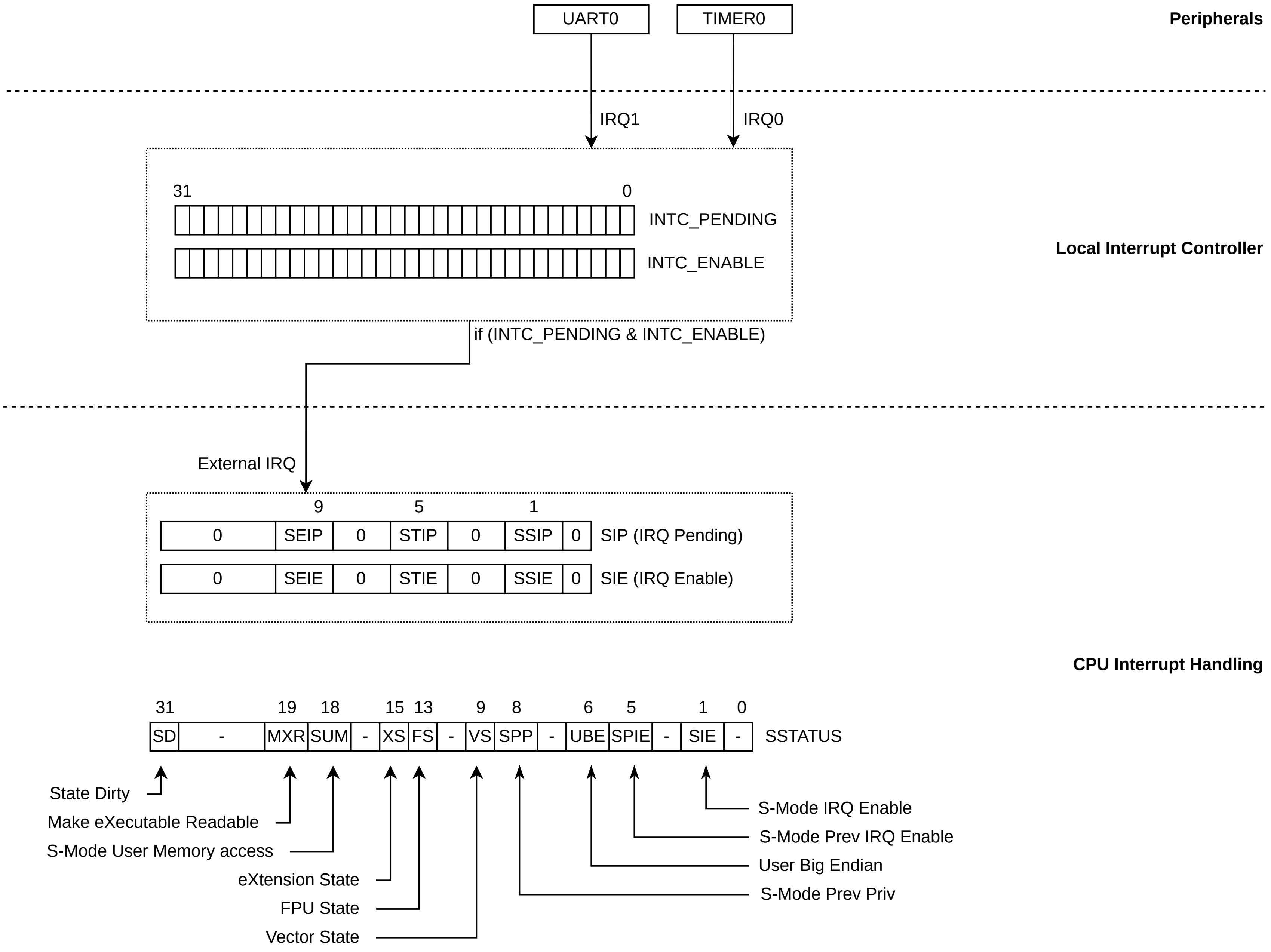
Previous Privilege Level is restored
SSTATUS.SPP == 0 ? User : Supervisor

Control is transferred to addres in SEPC
PC = SEPC

STVAL

Will contain the relevant virtual address for
Breakpoint, Address Misaligned,
Page Fault or Access Fault Exceptions

RISCV RV32IM Interrupt Processing



Peripherals

UART0

TIMER0

IRQ1

IRQ0

Local Interrupt Controller

31 … 0

INTC_PENDING

INTC_ENABLE

if (INTC_PENDING & INTC_ENABLE)

External IRQ

	9		5		1		
0	SEIP	0	STIP	0	SSIP	0	SIP (IRQ Pending)
0	SEIE	0	STIE	0	SSIE	0	SIE (IRQ Enable)

CPU Interrupt Handling

31		19	18		15	13		9	8		6	5		1	0	
SD	-	MXR	SUM	-	XS	FS	-	VS	SPP	-	UBE	SPIE	-	SIE	-	SSTATUS

State Dirty

Make eXecutable Readable

S-Mode User Memory access

eXtension State

FPU State

Vector State

S-Mode IRQ Enable

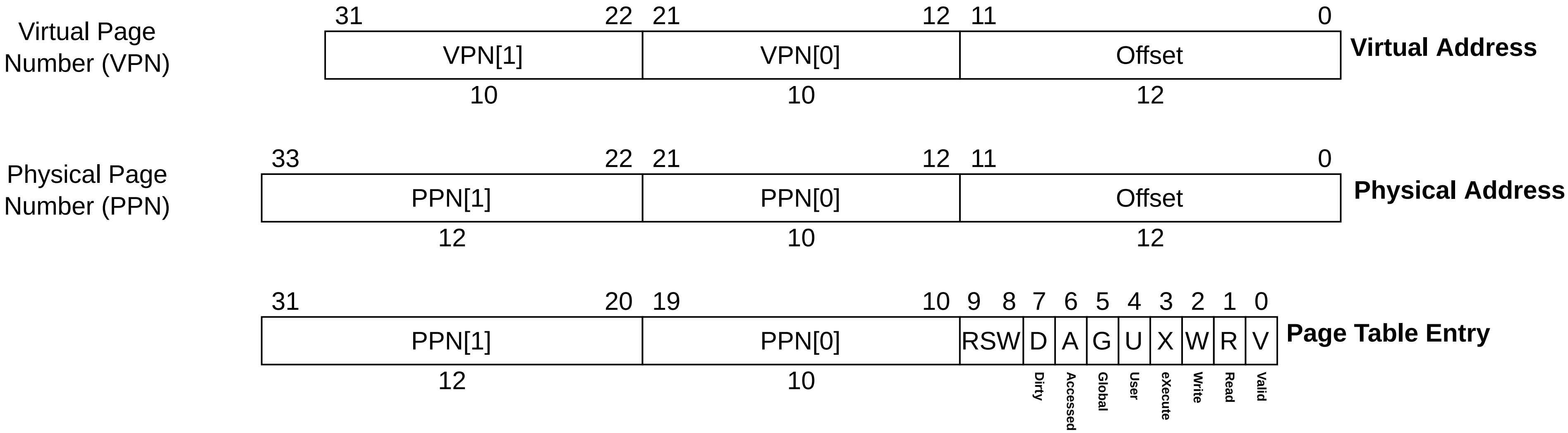
S-Mode Prev IRQ Enable

User Big Endian

S-Mode Prev Priv

swetland / 2022

RISCV RV32IM Sv32 MMU Data Structures



Virtual Page Number (VPN)

Virtual Address

31–22	21–12	11–0
VPN[1]	VPN[0]	Offset
10	10	12

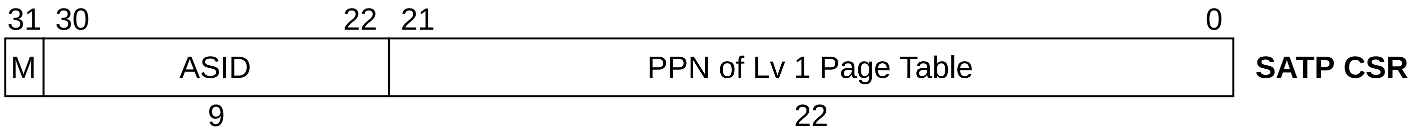
Physical Page Number (PPN)

Physical Address

33–22	21–12	11–0
PPN[1]	PPN[0]	Offset
12	10	12

Page Table Entry

31–20	19–10	9–8	7	6	5	4	3	2	1	0
PPN[1]	PPN[0]	RSW	D	A	G	U	X	W	R	V
12	10		Dirty	Accessed	Global	User	eXecute	Write	Read	Valid

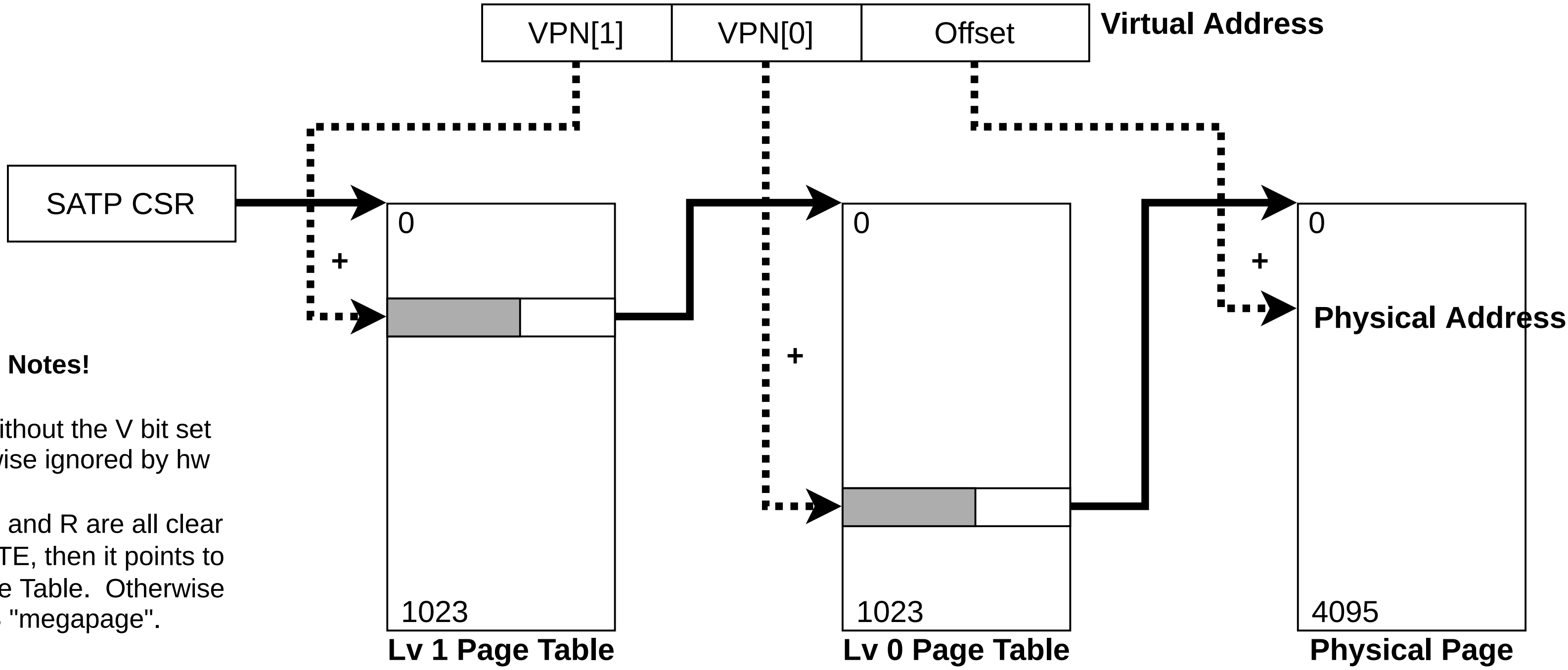


SATP CSR

31	30–22	21–0
M	ASID	PPN of Lv 1 Page Table
	9	22

Important Note!

The flags in a PTE
are only 10 bits wide.
When storing a phys
addr in one, be sure
to shift it appropriately.



Virtual Address

VPN[1] | VPN[0] | Offset

SATP CSR

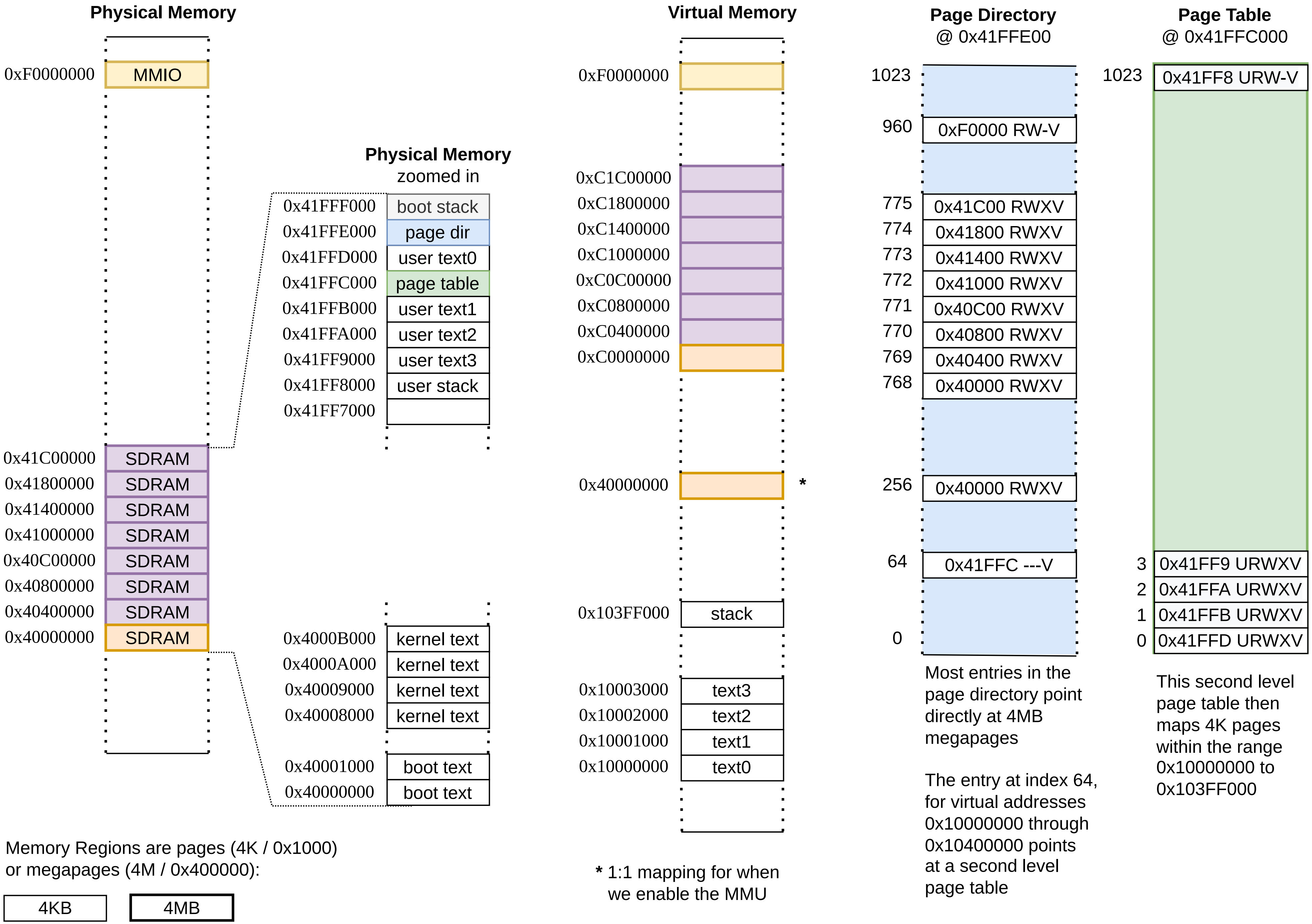
Important Notes!

1. PTEs without the V bit set are otherwise ignored by hw

2 . If X, W, and R are all clear in a Lv1 PTE, then it points to a Lv0 Page Table. Otherwise it is a 4MB "megapage".

swetland / 2022

Physical Memory / Virtual Memory / Page Tables



Physical Memory

0xF0000000	MMIO
0x41C00000	SDRAM
0x41800000	SDRAM
0x41400000	SDRAM
0x41000000	SDRAM
0x40C00000	SDRAM
0x40800000	SDRAM
0x40400000	SDRAM
0x40000000	SDRAM

Physical Memory
zoomed in

0x41FFF000	boot stack
0x41FFE000	page dir
0x41FFD000	user text0
0x41FFC000	page table
0x41FFB000	user text1
0x41FFA000	user text2
0x41FF9000	user text3
0x41FF8000	user stack
0x41FF7000	
0x4000B000	kernel text
0x4000A000	kernel text
0x40009000	kernel text
0x40008000	kernel text
0x40001000	boot text
0x40000000	boot text

Virtual Memory

0xF0000000	
0xC1C00000	
0xC1800000	
0xC1400000	
0xC1000000	
0xC0C00000	
0xC0800000	
0xC0400000	
0xC0000000	
0x40000000	*
0x103FF000	stack
0x10003000	text3
0x10002000	text2
0x10001000	text1
0x10000000	text0

Page Directory
@ 0x41FFE00

1023	
960	0xF0000 RW-V
775	0x41C00 RWXV
774	0x41800 RWXV
773	0x41400 RWXV
772	0x41000 RWXV
771	0x40C00 RWXV
770	0x40800 RWXV
769	0x40400 RWXV
768	0x40000 RWXV
256	0x40000 RWXV
64	0x41FFC ---V
0	

Page Table
@ 0x41FFC000

1023	0x41FF8 URW-V
3	0x41FF9 URWXV
2	0x41FFA URWXV
1	0x41FFB URWXV
0	0x41FFD URWXV

Most entries in the page directory point directly at 4MB megapages

The entry at index 64, for virtual addresses 0x10000000 through 0x10400000 points at a second level page table

This second level page table then maps 4K pages within the range 0x10000000 to 0x103FF000

Memory Regions are pages (4K / 0x1000) or megapages (4M / 0x400000):

4KB 4MB

* 1:1 mapping for when we enable the MMU

swetland / 2022

Resources

RISCV Instruction Set Architecture (ISA) Spec I: Unprivileged
https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf

RISCV Instruction Set Architecture (ISA) Spec II: Privileged
https://github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf

RISCV Calling Convention Spec:
https://riscv.org/wp-content/uploads/2015/01/riscv-calling.pdf

RISCV Application Binary Interface (ABI) Spec:
https://github.com/riscv-non-isa/riscv-elf-psabi-doc/releases/download/v1.0-rc2/riscv-abi.pdf

lk (LittleKernel) - Small OS that supports RV32, and a number of other architectures
https://github.com/littlekernel/lk

xv6 - Re-implementation of UNIX v6 for RV64 (a RV32 port might be fun!)
https://github.com/mit-pdos/xv6-riscv
https://pdos.csail.mit.edu/6.S081/2021/xv6/book-riscv-rev2.pdf
https://pdos.csail.mit.edu/6.S081/2021/overview.html (lecture notes, videos, etc)