Sistemas Operativos

Arquitectura RISC-V

Marcelo Arroyo

Dpto de Computación - FCEFQyN
Universidad Nacional de Río Cuarto

2025

Descripción general

RISC-V

Estándar de una arquitectura *abierta* de un conjunto de instrucciones (ISA) basado en *Reduced Instruction Set Computer (RISC)*.
El proyecto se inició en 2010 en la Universidad de California, Berkeley.
Actualmente soportado por el consorcio *RISC-V International*

Características

- Arquitectura *load-store*
- Conjunto de instrucciones base de 32 bits
- Soporta *extensiones* (floating point, . . .)
- Variantes con espacios de direcciones de 32, 64 o 128 bits

Terminología

Cores

Un *core* es componente que tiene una unidad de *fetch de instrucciones* independiente

Hart

Hardware *thread*. Un core puede soportar múltiples *harts*

Coprocesadores

Un *core* puede incluir *extensiones*. Cada coprocesador puede *extender* el conjunto de instrucciones base

Registros

Name	Asm name	Description	Saved by
x0	zero	Contiene 0	
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	
x4	tp	Thread pointer	
x5-x7	t0-t2	Temporaries	Caller
x8	fp	Frame pointer	Callee
x9	s1	Saved register	Callee
x10-x17	a0-a7	Arguments	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporaries	Caller

- Los registros pueden ser de 32 (RV-32) o 64 bits (RV-64)
- Las funciones reciben sus argumentos en a0–a7 y retornan el valor en a0–a1

Instrucciones (no privilegiadas)

Asm. Instruction	Description
`li rd, value`	Load immediate: $rd = value$
`la rd, address`	Load address: $rd = mem[pc - address]$
`l{b | h | w | d} rd, address`	Load: $rd = mem[pc - address]$
`s{b | h | w | d} rs, addr, rb`	Store: $mem[rb + pc - offset] = rs$
`mv rd, rs`	Copy register: $rd \leftarrow rs$
`add rd, rs1, rs2`	Add: $rd = rs1 + rs2$
`addi rd, rs1, v`	Add immediate: $rd = rs1 + v$
`sub rd, rs1, rs2`	Sub:: $rd = rs1 + rs2$
`blt r1, r2, offset`	Branch: if $rt < rs$, $pc = pc + offset$
`beq r1, r2, offset`	Branch: if $rt = rs$, $pc = pc + offset$
`j offset`	Jump: $pc = pc + offset$
`jal offset`	Jump and link: $ra = pc$; $pc = pc + offset$
`ret`	Return from sumbroutine: $pc = ra$
`call offset`	As jal (with long offset)
`fence`	Memory barrier: no reorder load/stores
`amoswap.w.aq rd, rs, s`	Atomic swap:$rs \leftrightarrow mem[s]; rd = old\, mem[s]$

Modos de ejecución: Privilegios

Level	Description
0	User (**U**) or application
1	Supervisor (**S**) (kernel)
2	Hypervisor (**H**) (for VMs)
3	Machine (**M**)

- Cada core inicia en `M` mode (más privilegiado)
- Cada modo tiene un conjunto de *registros de control y estado (CSRs)*

Nota: Trabajaremos con un board *RISC-V32* con modos `U`,`S`,`M`.

CSRs (en machine y supervisor) modes

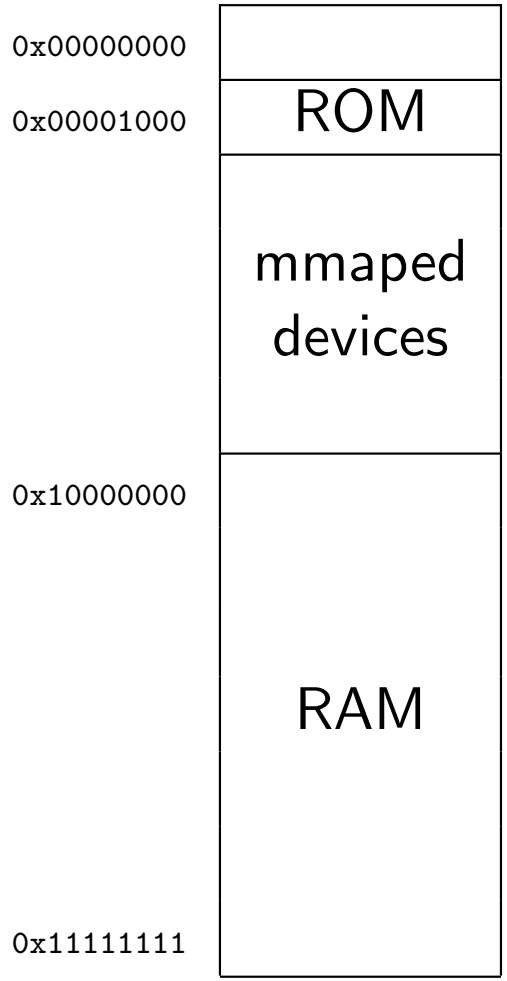
Machine mode	Supervisor mode	Description
mstatus	sstatus	Status register
mie	sie	Interrupts enable/pending
mtvec	stvec	Trap hander/vector base address
mscratch	sscratch	Pointer to data area (for ISR)
mepc	sepc	Saved program counter on trap/irq
mcause	scause	Trap cause (interrupt number)
mtval	stval	Bad address or instruction
mip	sip	Interrupts pending
	satp	Address of root page table
pmpcgf0-63		Phys. memory protection address
medeleg		Exceptions delegation
mideleg		Interrupts delegation
mhartid		Hart (core) number

Cambios de privilegio

- La instrucción `ecall` cambia a un nivel de privilegio mayor.
  Se usa comúnmente para implementar *syscalls*
  Pasos al pasar a modo supervisor desde modo usuario:
  1. La CPU salva el `pc` en `sepc`
  2. Cambia el `pc` a `stvec` (*trap vector address*)
- Instrucción `sret` (ejecutada en el *trap handler*: Retorna al nivel de privilegio anterior (configurado en `sstatus`)
  El `pc` toma el valor del registro `sepc`
- En *machine mode* funciona de manera similar pero se debe usar `mret`. El nivel de privilegio a retornar se configura en `mstatus`.
  El `pc` toma el valor del registro `mepc`

RV-32: Espacios de memoria física

- ROM: A partir de la dirección 0x00001000.
- Luego, espacio para dispositivos mapeados en memoria (registros de los controladores de dispositivos)
- RAM a partir de 0x10000000 (2GB).



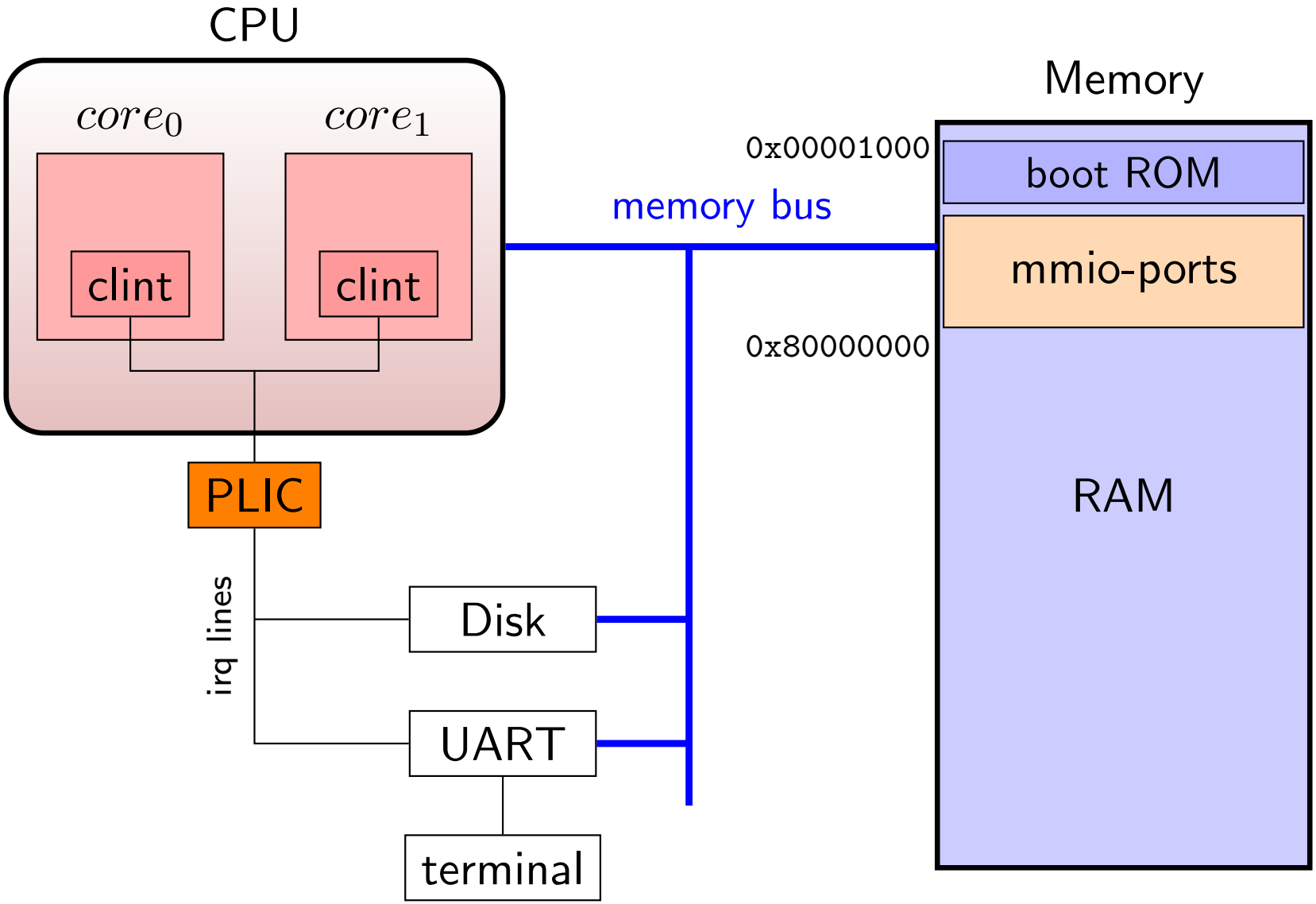
Interrupciones

- Los dispositivos se *mapean en memoria*. No existe un bus de I/O dedicado
- Cada *core* tiene un *core line interrupt controller (clint)*
- Una plataforma (board) comúnmente tiene un *Programmable Interrupt Controller (PLIC)* en cual puede configurarse para *rutear* interrucpiones a diferentes cores
- Las interrupciones se *atrapan* en *machine mode* pero pueden *delegarse* a otros modos (comúnmente supervisor mode)
- La delegación se configura en los CSRs `medeleg` (excepciones) y `mideleg` (interrupciones)
- El csr `stvec` contiene la dirección base del *trap_handler* o la base del vector de interrupciones (vector de punteros a diferentes *handlers*). En el último caso, en una interrupción, en RV-32, $pc = stvec[4 * scause]$

Plataforma Qemu virt (RV-32)

- Arquitectura de 32 bits
- Bus de direcciones físico de 32 bits (4GB)
- Múltiples cores (flag `-smp` n)
- Controlador de interrupciones *PLIC*
- Chip de *Universal Asynchronous Transmitter-Receiver (UART)* serial device
- Interfaces *virtio-mmio*
- Puente genérico de un *host PCI*
- *Real Time Clock (RTC)*
- Memoria *flash* (con *boot firmware*)

Esquema del board QEMU virt (RV32)



Dispositivos

CLINT: Core Level Interrupt controller

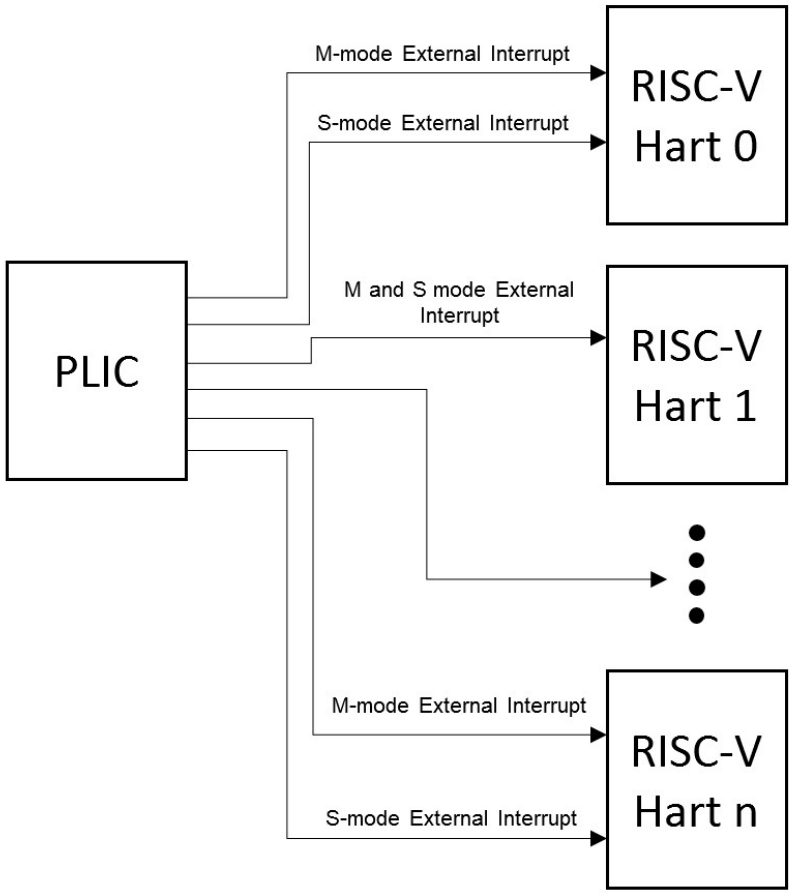
- Dispositivo mapeado a partir de la dirección `0x2000000`
- Registro `MTIMER` de 64 bits en `0x200BFF8`
- Por cada core un *registro comparador* `MTIMECMP` mapeado en `0x2004000 + i` (para cada core i)
- Cada core tiene su correspondiente

Funcionamiento

1. `MTIMER` contiene en número de ciclos desde el inicio (a una frecuencia fija pre-establecida)
2. Un core i pone en `MTIMECMP[i]` un valor
3. Cuando `MTIMER` $\geq$ `MTIMECMP[i]` genera una interrupción pendiente en el core i (set un bit en `mpeip`)

PLIC: Platform Level Interrupt Controller

- Registros:
  1. `M/SENABLE[hart]`,
  2. `M/SPRIORITY[hart]`
  3. `M/SCLAIM[hart]` (irq number)
- Ante una interrupción el CSR `mip/sip` queda con el bit setado con la línea de interrupción.
  También se setea en *m/scause* el *interrupt bit* de y el *exception code*



PLIC: Diagrama de bloques.

PLIC: Operación

1. Un dispositivo genera una interrupción
2. PLIC notifica a un core (*hart*) seteando el bit correspondiente en `m/sip` (*interrupts pending*) y `m/scause` (nro de interrupción)
3. El core dispara la ISR apuntada por `m/stvec`
4. La ISR lee (*claim*) el número o línea de *irq* desde `M/SCLAIM[hart]`[1]
5. El PLIC retorna el número de irq
6. La ISR notifica que procesó la interrupción escribiendo en `M/SCLAIM[hart]` $\leftarrow$ `irq_number`
7. El PLIC puede generar otra interrupción

Para más detalles, ver `plic.c`

[1] Ver función `devintr()` en `trap.c`.

Qemu RISC-V generic virtual platform devices

Universal Asynchronous Receiver-Transmitter (UART)

- Transmisión serial (de a bytes)
- Usado comúnmente para conexión de terminales (ttys)
- Detalles: `http://byterunner.com/16550.html`

Virtio devices

- Estándar propuesto para *dispositivos virtuales*
- Documentación: `https://docs.oasis-open.org/virtio/virtio/v1.1/virtio-v1.1.pdf`
- Sólo usaremos un driver para el disco.

Manuales técnicos

Sitio oficial: riscv.org

- Volume 1: Unprivileged Specification
- Volume 2: Privileged Specification
- Calling convention: `https://riscv.org/wp-content/uploads/2015/01/riscv-calling.pdf`