Interrupts

ECE 391

Topics and Concepts

- Control and Status Registers
- RISC-V execution modes (M, S, and U)
- Interrupt controller (PLIC on RISC-V)
- Interrupt priorities and masking
- CPU interrupt delivery and trap handling

Know how to …

- Read, write, and modify CSRs
- Configure the PLIC to deliver interrupts from source
- Configure CPU to receive external interrupts
- Write a RISC-V interrupt handler

Know these terms …

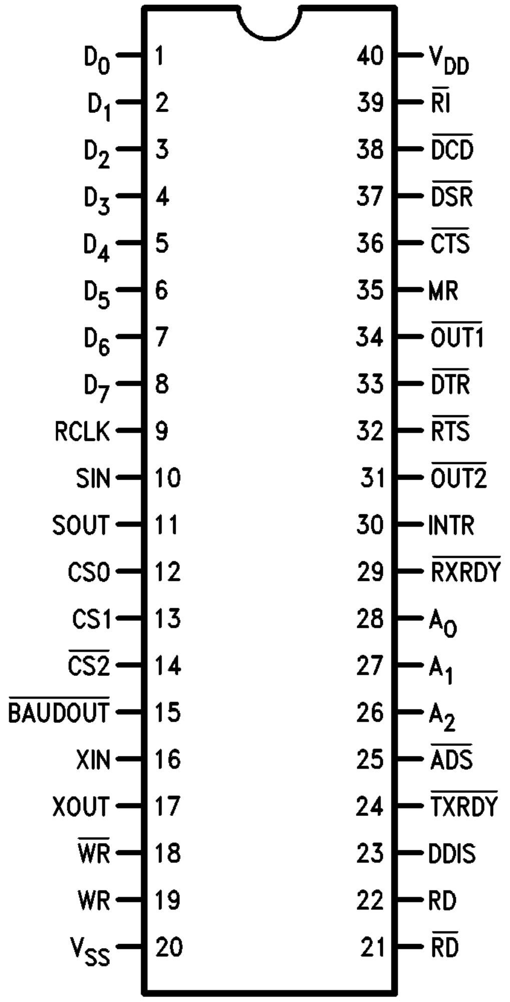
- CSRs
- Exceptions
- Interrupts
- Traps
- PLIC
- Hart
- Context (PLIC interupt sink)
- Atomic

- CSRs
  - mstatus
  - mip and mie
  - mtvec
  - mcause
  - mepc

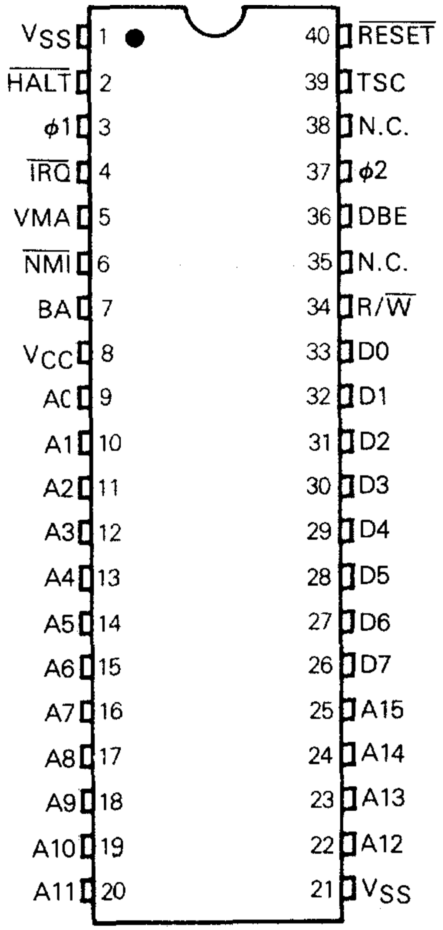
Interrupts

- The CPU has one interrupt request pin: *what if we have multiple devices?*

Simplest Interrupt Wiring



NS16550A



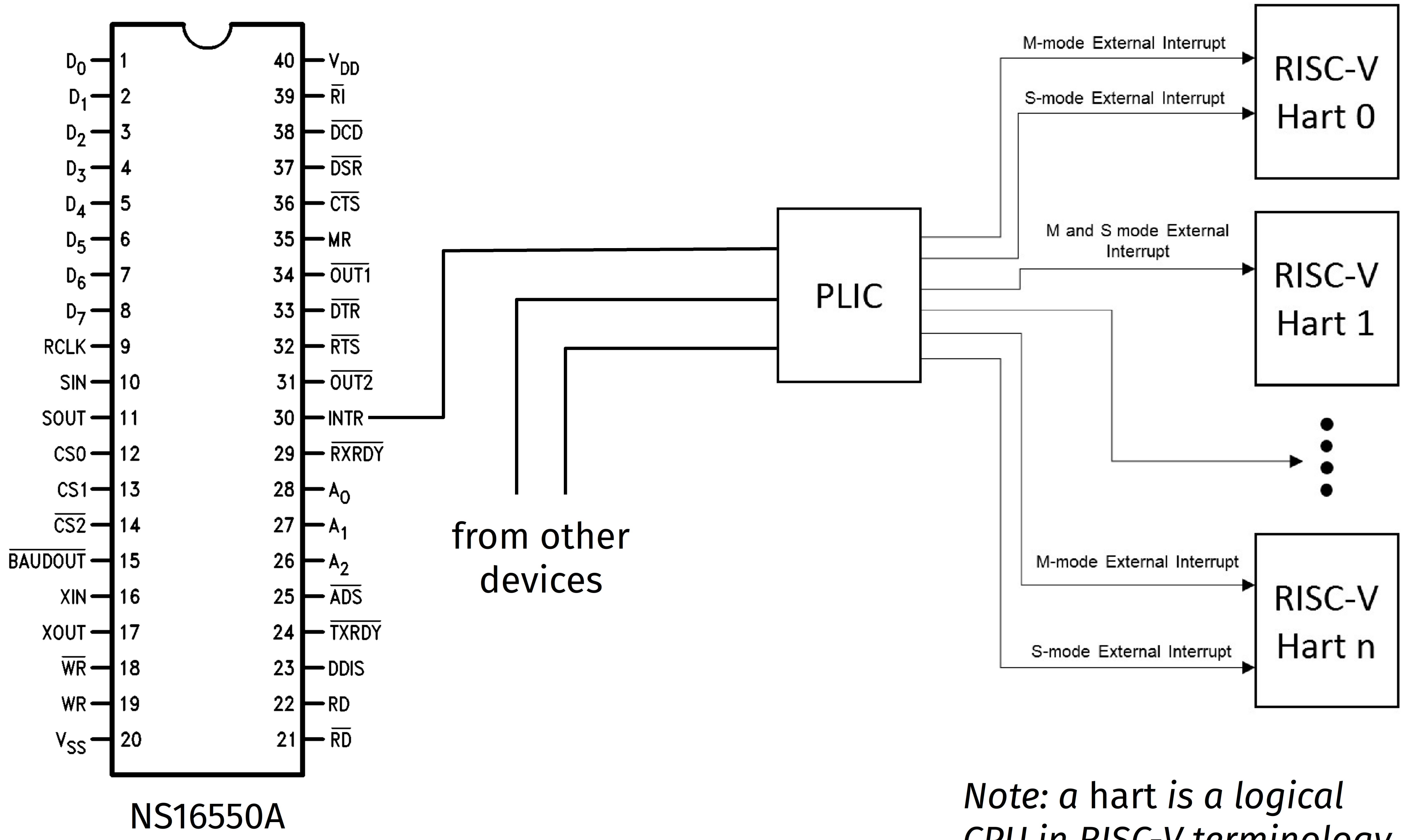
MC6800

K. Levchenko
Images copyright National Semiconductor Corp., 1990; Motorolla, Inc., 1984.

Interrupt Controller

- The one-gate-to-or-them-all interrupt wiring scheme works for simple systems
  - Check every device to know who raised the interrupt
  - No way to disable some interrupt sources
- Modern systems use an **interrupt controller** to give the CPU more control over external interrupts
- RISC-V systems use an interrupt controller called a *Platform Level Interrupt Controller* (*PLIC*)

PLIC Wiring



Note: a hart is a logical CPU in RISC-V terminology

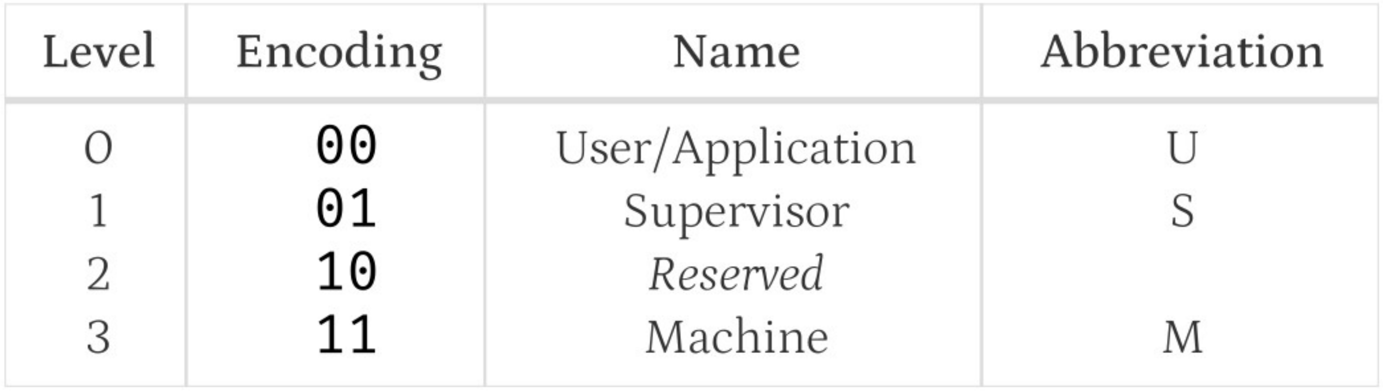
K. Levchenko
Images copyright National Semiconductor Corp., 1990; RISC-V international, 2017-2023, CC BY 4.0.

Hart to Heart

- RISC-V documentation uses the term ***hart*** to refer to a computing resource that behaves like a discrete CPU from the programmer’s point of view
  - I will refer to this as a *logical CPU* and often just *CPU* when the context is software (rather than architecture)
- Logical CPUs may share units (such as ALUs) but appear as separate processing units
- From the programmer’s point of view, these are just CPUs

RISC-V Privilege Modes

- At any given time, a RISC-V CPU is operating in one of three modes:
  - **M**achine mode: most privileged, CPU starts in this mode
  - **S**upervisor mode: privileged mode intended for the OS
  - **U**ser mode: least privileged mode for user programs


RISC-V privilege levels.

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

K. Levchenko

RISC-V Privilege Modes

- At any given time, a RISC-V CPU is operating in one of three modes:
  - **M**achine mode: most privileged, CPU starts in this mode
  - **S**upervisor mode: privileged mode intended for the OS
  - **U**ser mode: least privileged mode for user programs
- An OS can run in M-mode or S-mode
  - In the course we will use M mode for the OS and U mode for user-space programs (later in the course)
  - So far we have been running all code in M mode

RISC-V Control and Status Registers

- In addition to general purpose registers and PC, a RISC-V CPU has *Control and Status Registers* (*CSRs*)
- Special registers that indicate CPU status and control its behavior
- Accessed using six instructions
  - CSR**RW**, CSR**RS**, CSR**RC** (register source operand)
  - CSR**RWI**, CSR**RSI**, CSR**RCI** (immediate source operand)
- *Access may have side-effects*

CSR Atomic Read and Write

rd ← *csr* ← *rs*	*rd* ← *csr* ← *imm*
`csrrw` *rd*, *csr*, *rs*	`csrrwi` *rd*, *csr*, *imm*

- If *rd* is not `zero`: Read named CSR, write the value read into general purpose register *rd*
  - If *rd* is `zero`, CSR is not read
- Write contents of *rs* (csrrw) or sign-extended immediate argument (csrrwi) to named CSR
- Read and write operations happen *atomically*
  - In a single indivisible action without an observable intermediate state

CSR Atomic Read and Set Bits

rd ← *csr* ← *csr* | *rs*

`csrrs` *rd*, *csr*, *rs*

rd ← *csr* ← *csr* | *imm*

`csrrsi` *rd*, *csr*, *imm*

- Read named CSR, write the value read into general purpose register *rd*
  - If *rd* is `zero`, CSR is still read (but result is ignored)
- If *rs* is not `zero`: Bitwise-OR the CSR with the contents of *rs* (csrrs)
- If *imm* ≠ 0: Bitwise-OR the CSR with the zero-extended immediate value (csrrsi)
- If *rs* is `zero` or *imm* = 0: no CSR write takes place

CSR Atomic Read and Clear Bits

rd ← *csr* ← *csr* & ~*rs*

`csrrc` *rd*, *csr*, *rs*

rd ← *csr* ← *csr* & ~*imm*

`csrrci` *rd*, *csr*, *imm*

- Read named CSR, write the value read into general purpose register *rd*
  - If *rd* is `zero`, CSR is still read (but result is ignored)
- If *rs* is not `zero`: Bitwise-AND the CSR with the bitwise-negated contents of *rs* (csrrs)
- If *imm* ≠ 0: Bitwise-AND the CSR with the bitwise-negated zero-extended immediate value (csrrsi)
- If *rs* is `zero` or *imm* = 0: no CSR write takes place

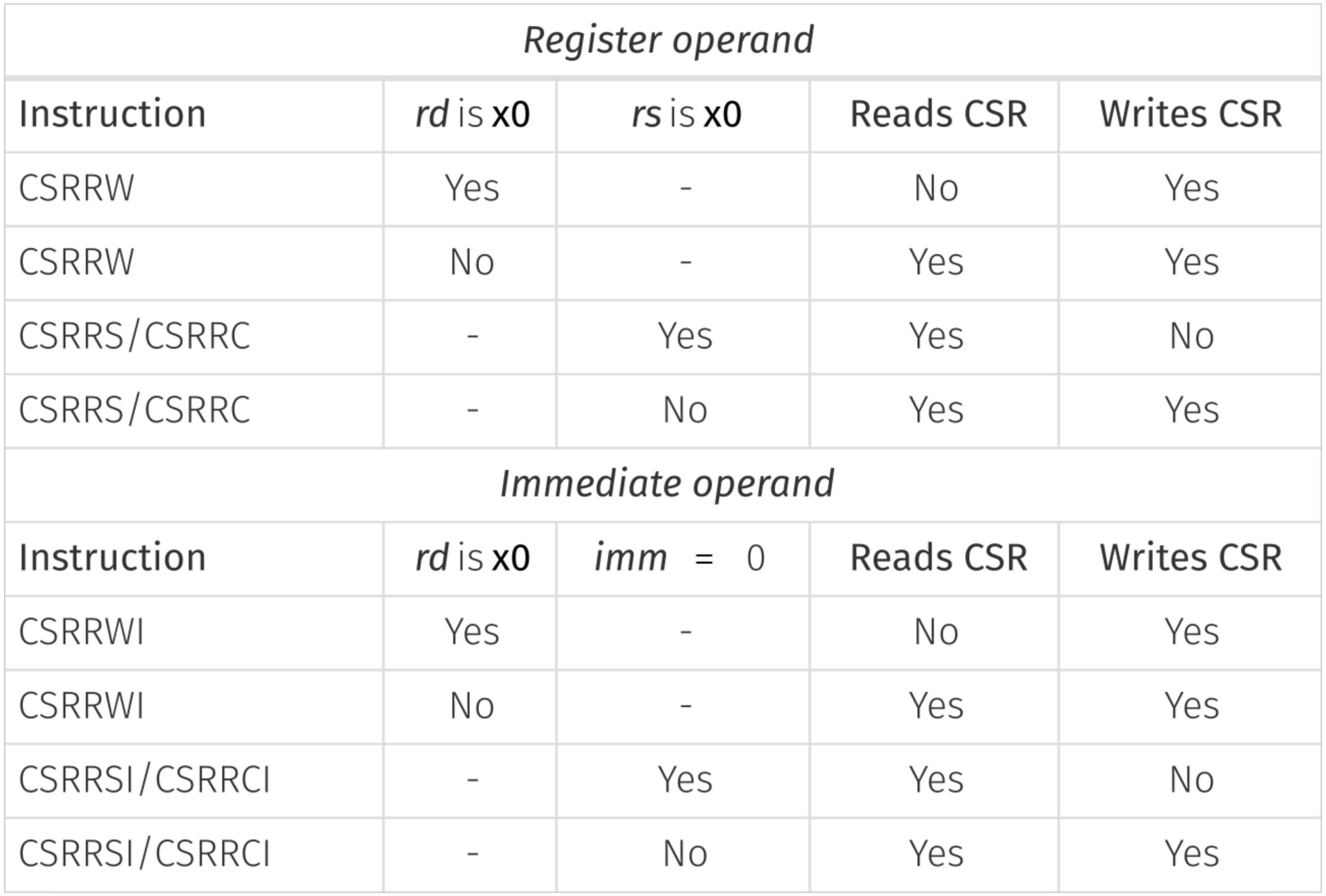
CSR Immediate Instructions

- The immediate value in the CSRRxI instructions is only 5 bits (zero-extended to 32 or 64 bits)
  - Can only modify bits 0 to 4 of the CSR

CSR Pseudo-Instructions

- `csrw` *csr*, *rs* ≡ `csrrw zero`, *csr*, *rs* *(CSR write)*
- `csrwi` *csr*, *imm* ≡ `csrrwi zero`, *csr*, *imm* *(CSR write)*
- `csrr` *csr*, *rs* ≡ `csrrs` *rd*, *csr*, `zero` *(CSR read)*
- `csrs` *csr*, *rs* ≡ `csrrs zero`, *csr*, *rs* *(CSR set bit)*
- `csrsi` *csr*, *imm* ≡ `csrrs zero`, *csr*, *imm* *(CSR set bit)*
- `csrc` *csr*, *rs* ≡ `csrrc zero`, *csr*, *rs* *(CSR clear bit)*
- `csrci` *csr*, *imm* ≡ `csrrc zero`, *csr*, *imm* *(CSR clear bit)*

CSR Instructions Summary



Register operand				
Instruction	*rd* is **x0**	*rs* is **x0**	Reads CSR	Writes CSR
CSRRW	Yes	-	No	Yes
CSRRW	No	-	Yes	Yes
CSRRS/CSRRC	-	Yes	Yes	No
CSRRS/CSRRC	-	No	Yes	Yes

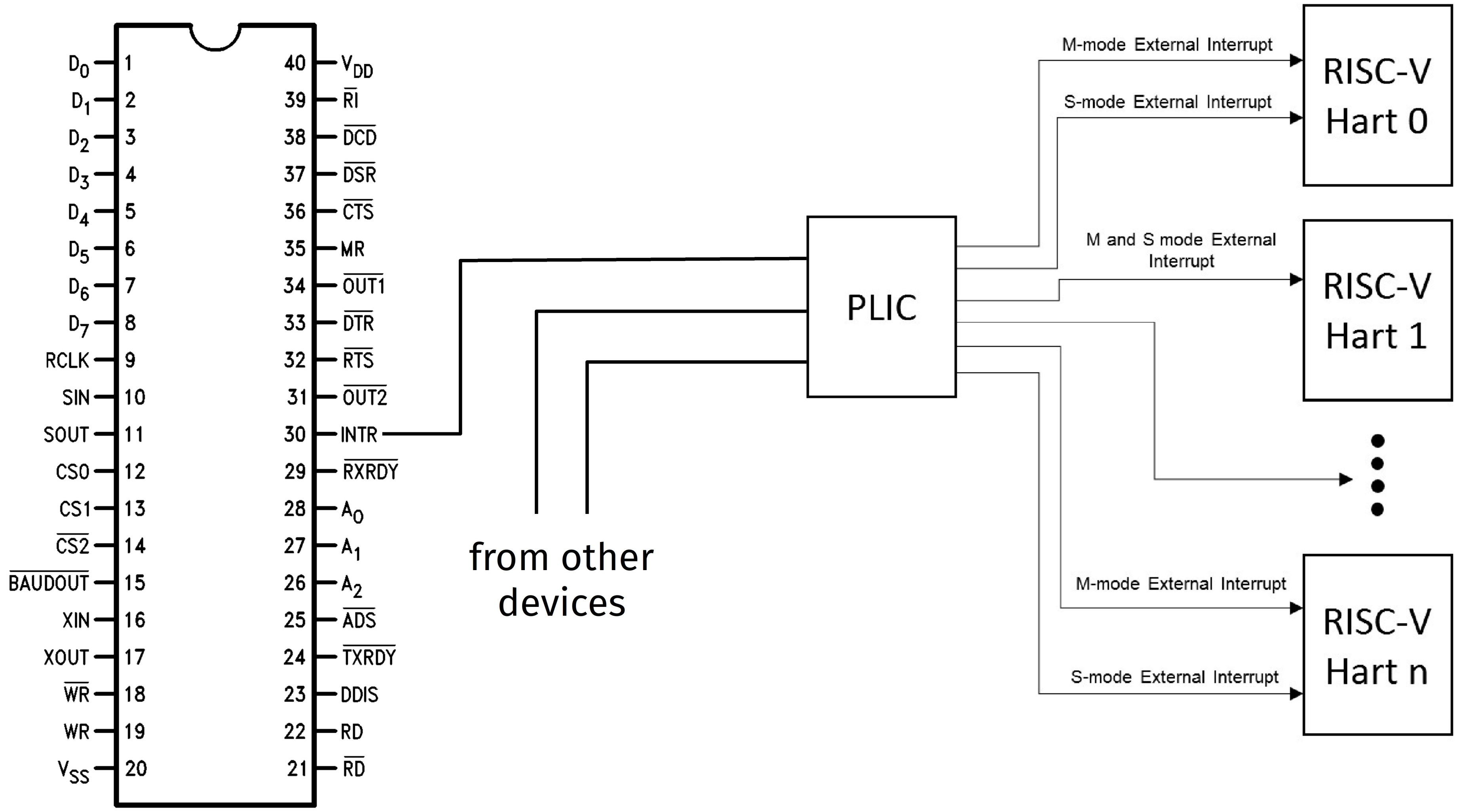
Immediate operand				
Instruction	*rd* is **x0**	$imm = 0$	Reads CSR	Writes CSR
CSRRWI	Yes	-	No	Yes
CSRRWI	No	-	Yes	Yes
CSRRSI/CSRRCI	-	Yes	Yes	No
CSRRSI/CSRRCI	-	No	Yes	Yes

K. Levchenko

RISC-V Terminology

- An **exception** is “an unusual condition occurring at run time associated with an instruction in the current RISC-V hart.”
- An **interrupt** is “an external asynchronous event that may cause a RISC-V hart to experience an unexpected transfer of control.”
- A **trap** is “the transfer of control to a trap handler caused by either an exception or an interrupt.”

Back to Interrupts



NS16550A

K. Levchenko

What Can PLIC Do for You?

- Assign a priority to each interrupt source
  - Priority 0 means interrupt is disabled (*masked*)
- Route interrupts to different CPUs
  - Can have dedicated CPU for each interrupt or spread interrupt delivery between multiple CPUs
  - In this course, we are working with a single-CPU system
- Set a priority threshold for each CPU
  - Only interrupts with priority above threshold delivered
- Indicates which interrupt source raised the interrupt

Interrupt Sources on `virt`

Device	PLIC source no.
VirtIO device 0	1
VirtIO device 1	2
VirtIO device *i*	1+*i*
Virtio device 7	8
RTC	9
UART 0	10
UART 1	11
UART *j*	10+*j*
UART 7	17

CPU Modes and Interrupts

- PLIC can deliver interrupts to M mode or S mode
- By default, all interrupts handled in M mode
- M mode can delegate interrupts to S mode
- PLICs route interrupts to *PLIC contexts*
  - Context is overloaded: we will say “PLIC context” when we mean context as defined by PLIC spec
- PLIC contexts map to M or S mode on CPUs in a system-dependent manner (not specified by spec)

PLIC Context to CPU and Mode

- On the virt system we are using:
  - **PLIC context 0 maps to M mode on hart 0**
  - PLIC context 1 maps to S mode on hart 0
  - PLIC context 2 maps to M mode on hart 1
  - PLIC context 3 maps to S mode on hart 1
  - PLIC context 2*k* maps to M mode on hart *k*
  - PLIC context 2*k*+1 maps to S mode on hart *k*
- In this course: *we will emulate one CPU and will not use S mode: only PLIC context 0 matters*

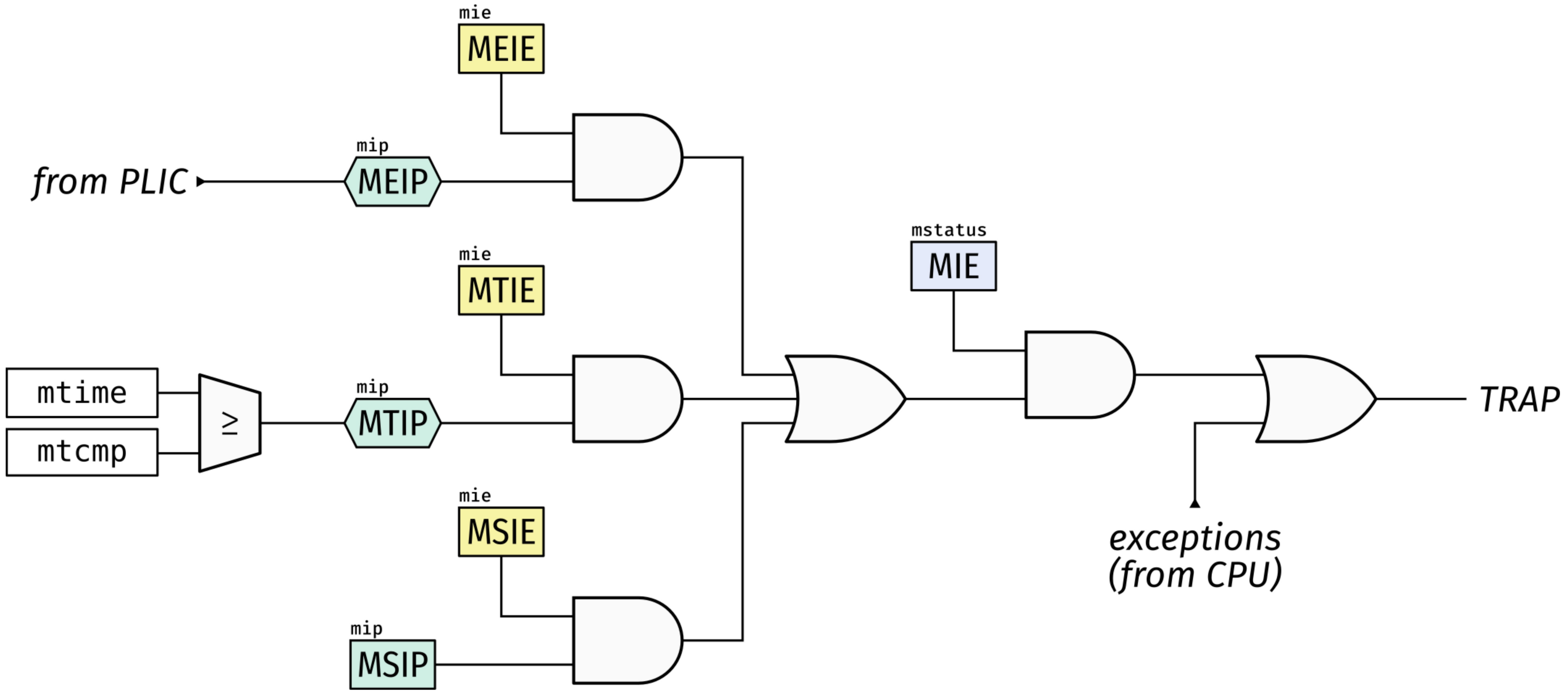
Our PLIC API

- You will write some functions for controlling the PLIC
  - `void plic_set_src_prio (`
    `unsigned int srcid, unsigned int prio);`
  - `void plic_enable_ctx_src (`
    `unsigned int ctxid, unsigned int srcid);`
  - `void plic_disable_ctx_src (`
    `unsigned int ctxid, unsigned int srcid);`
  - `void plic_set_ctx_threshold (`
    `unsigned int ctxid, unsigned int prio);`

The Picture So Far

- External devices are interrupt *sources* for the PLIC
  - One device could originate multiple interrupts
- When an interrupt is asserted, the PLIC decides whether to send it to a PLIC context based on:
  - Whether the source is enabled (priority non-zero)
  - The set of sources enabled for each PLIC context
  - Whether (source priority) > (PLIC context threshold)
- *We will only use context 0 in the course*

Interrupt Signals Inside the CPU



`mie` and `mip`

`mip` - **Pending interrupts** (status register)

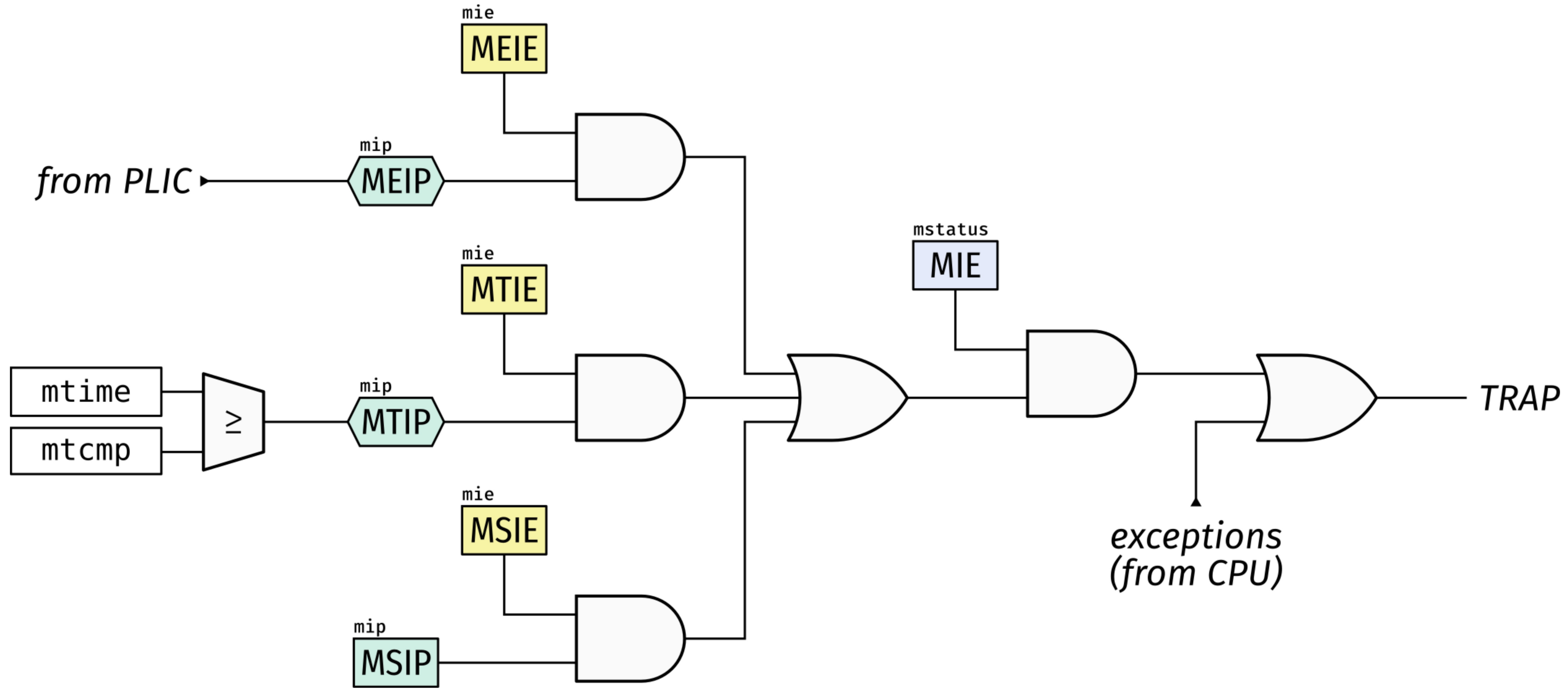
15 14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	LCOFIP	0	MEIP	0	SEIP	0	MTIP	0	STIP	0	MSIP	0	SSIP	0
2	1	1	1	1	1	1	1	1	1	1	1	1	1	1

`mie` - **Enable interrupts** (control register)

15 14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	LCOFIE	0	MEIE	0	SEIE	0	MTIE	0	STIE	0	MSIE	0	SSIE	0
2	1	1	1	1	1	1	1	1	1	1	1	1	1	1

- Bits `mip`.MEIP and `mie`.MEIE are the interrupt-pending and interrupt-enable bits for machine-level external interrupts. MEIP is read-only in `mip`, and is set and cleared by a platform-specific interrupt controller.
- Bits `mip`.MTIP and `mie`.MTIE are the interrupt-pending and interrupt-enable bits for machine timer interrupts. MTIP is read-only in `mip`, and is cleared by writing to the memory-mapped machine-mode timer compare register.

Interrupt Signals Inside the CPU



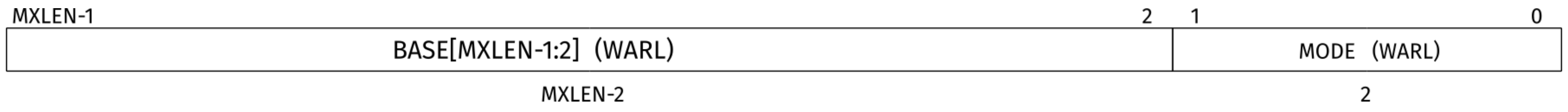
mstatus

`mstatus` - Machine status register

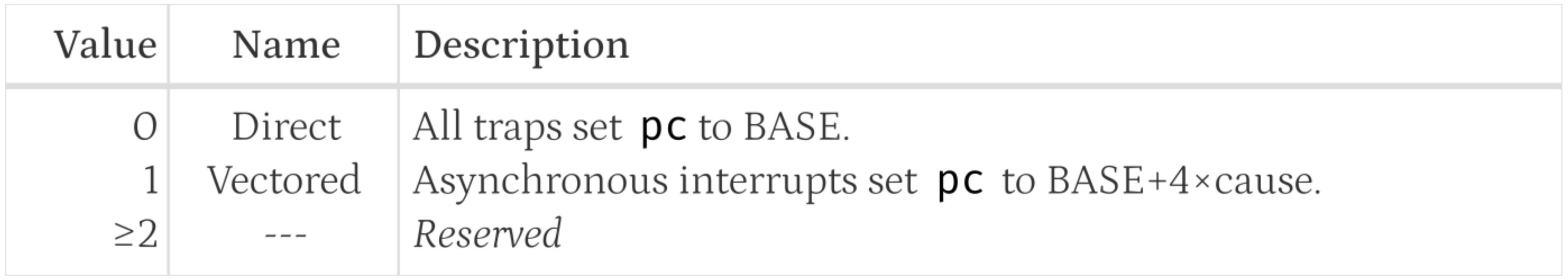
22	21	20	19	18	17	16 15	14 13	12 11	10 9	8	7	6	5	4	3	2	1	0
TSR	TW	TVM	MXR	SUM	MPRV	XS[1:0]	FS[1:0]	MPP[1:0]	VS[1:0]	SPP	MPIE	UBE	SPIE	WPRI	MIE	WPRI	SIE	WPRI
1	1	1	1	1	1	2	2	2	2	1	1	1	1	1	1	1	1	1

- MIE: global interrupt enable for M mode
- MIE can be set and cleared using CSRRSI and CSRRCI instructions

mtvec

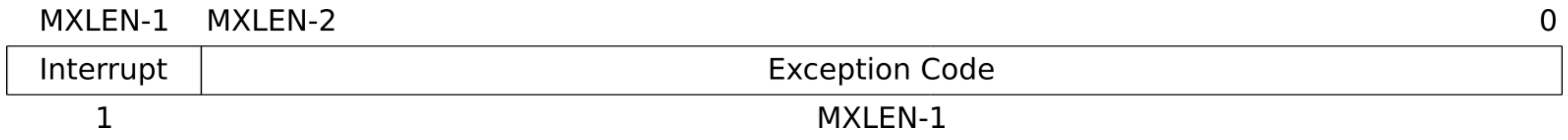


- MODE:



Value	Name	Description
0	Direct	All traps set `pc` to BASE.
1	Vectored	Asynchronous interrupts set `pc` to BASE+4×cause.
≥2	---	*Reserved*

mcause



Machine cause register (mcause) values after trap.

Interrupt	Exception Code	Description
1	0	*Reserved*
1	1	Supervisor software interrupt
1	2	*Reserved*
1	3	Machine software interrupt
1	4	*Reserved*
1	5	Supervisor timer interrupt
1	6	*Reserved*
1	7	Machine timer interrupt
1	8	*Reserved*
1	9	Supervisor external interrupt
1	10	*Reserved*
1	11	Machine external interrupt
1	12	*Reserved*
1	13	Counter-overflow interrupt
1	14-15	*Reserved*
1	≥16	*Designated for platform use*

Enabling Interupts

1. Set trap handler address (`mtvec`)
2. Enable interrupts on device (e.g. set DRIE=1 on NS16550A)
3. Set source priority on PLIC to non-zero value
4. Enable source for context 0 (`M` mode on hart 0)
5. Set context 0 interrupt priority threshold
   - Only sources whose priority is greater than context's priority threshold will be sent to the context
6. Enable M mode external interrupts on CPU (`mie`.MEIE = 1)
7. Enable M mode interrupts (`mstatus`.MIE = 1)

Accessing CSRs from C

- Create functions to access CSRs using C inline assembly:

```
int64_t csrr_mcause(void) {
    int64_t val;

    asm inline ("csrr %0, mcause" : "=r" (val));
    return val;

}

void csrw_mtvec(void (*handler)(void)) {   // MODE 0

    asm inline ("csrw mtvec, %0" :: "r" (handler));

}
```

https://gcc.gnu.org/onlinedocs/gcc/extensions-to-the-c-language-family/how-to-use-inline-assembly-language-in-c-code.html

Enabling Interupts

1. Set trap handler address (`mtvec`)
   - `csrw_mtvec(trap_handler);`
2. Enable interrupts on device (e.g. set DRIE=1 on NS16550A)
   - *E.g.* `UART0.ier = DRIE;`
3. Set source priority on PLIC to non-zero value
   - `plic_set_src_prio(srcid, prio);`
4. Enable source for context 0 (M mode on hart 0)
   - `plic_enable_ctx_src(ctxid, srcid);`

Enabling Interupts

5. Set context 0 interrupt priority threshold
   - `plic_set_ctx_threshold(ctxid, level);`
6. Enable M mode external interrupts on CPU (`mie`.MEIE = 1)
   - `csrs_mie(MEIE);`
7. Enable M mode interrupts (`mstatus`.MIE = 1)
   - `csrs_mstatus(MIE);`

Interrupt Handler

- CPU jumps to trap handler when PLIC interrupts
  - Can happen at any time when `mie`.**MEIE** = 1 and `mstatus`.**MIE** = 1
- All general purpose registers contain whatever values they had when CPU was interrupted
- After servicing the interrupt, CPU should resume execution at the point where the interrupted
- Need to restore register values to their original state
  - Interrupted program assumes register values will not mysteriously change between instructions!
- This is tricky!

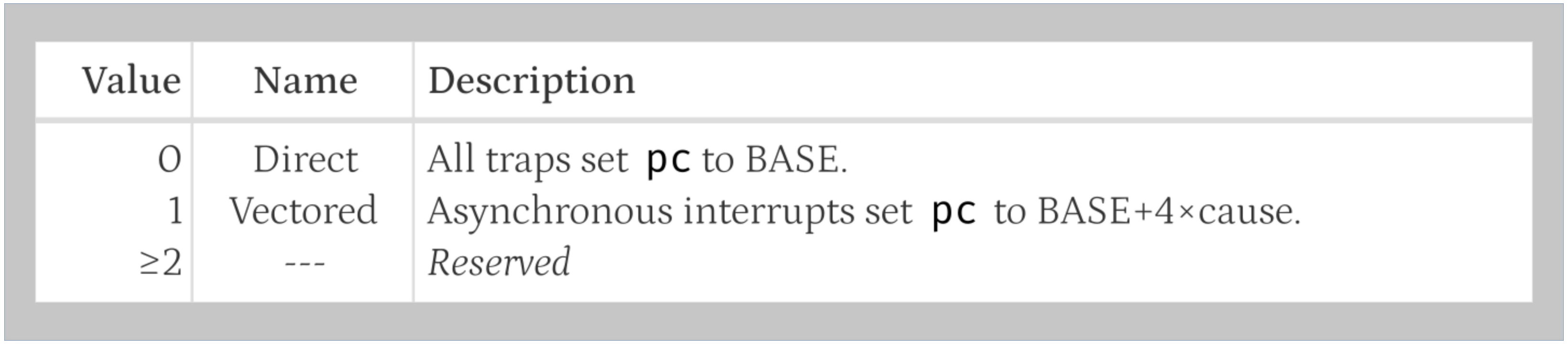
Execution State

- We need to save the current program execution state on entry so we can resume execution later:
  - PC and general purpose registers
- **Need stack for handler** (unless handler very simple):
  - Use separate stack for trap handlers
  - Use current kernel stack
- Where to save PC and registers?
  - To special area
  - **To handler stack** (which may be the main kernel stack)

Trap Entry

- Interrupts and exceptions both handled as *traps*
- Traps are handled in M mode
  - M mode can delegate to S mode, but we won't use this
- On trap entry, the CPU does the following:
  - Places current PC into `mepc` CSR
  - Sets `mstatus`.MPP to previous privilege mode
  - Sets `mstatus`.MPIE to `mstatus`.MIE
  - Sets `mstatus`.MIE to 0 (interrupts disabled on entry)
  - Sets PC to mtvec.BASE (if mode 0) or mtvec.BASE + 4×*cause*

Trap Entry



Value	Name	Description
0	Direct	All traps set `pc` to BASE.
1	Vectored	Asynchronous interrupts set `pc` to BASE+4×cause.
≥2	---	*Reserved*

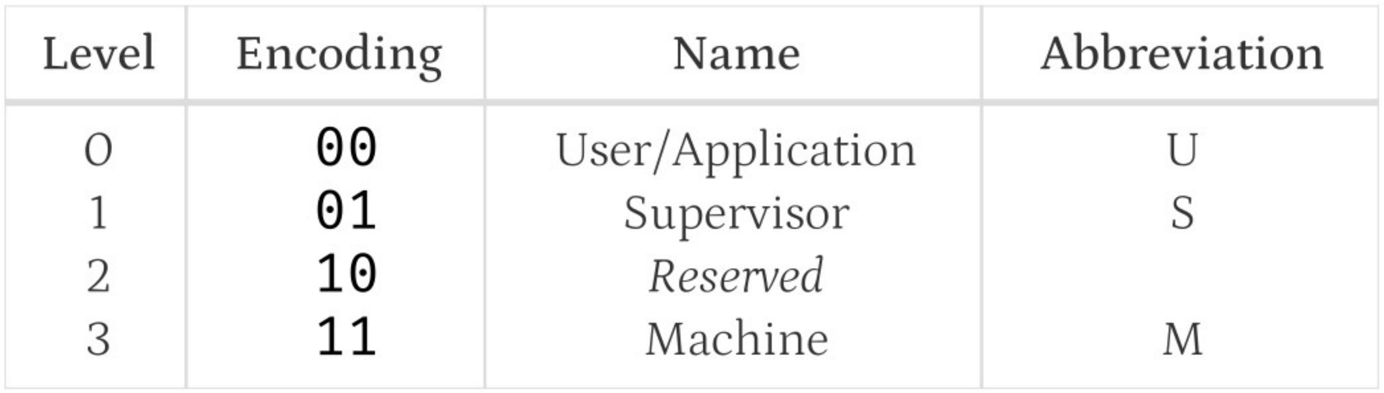
- On trap entry, the CPU does the following:
  - Places current PC into `mepc` CSR
  - Sets `mstatus`.MPP to previous privilege mode
  - Sets `mstatus`.MPIE to `mstatus`.MIE
  - Sets `mstatus`.MIE to 0 (interrupts disabled on entry)
  - Sets PC to mtvec.BASE (if mode 0) or mtvec.BASE + 4×*cause*

K. Levchenko

Trap Exit

- The mret instruction is used to exit a trap:
  - Sets mstatus.mie to `mstatus`.**MPP**; sets **MPIE** to 1
  - Changes privilege mode to `mstatus`.**MPP**; sets **MPP** to 0
  - Set PC to `mepc`

RISC-V privilege levels.



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

Trap Handler Overview

1. Save all registers to the stack
   - We will re-use the kernel stack
2. Save mepc and mstatus
   - Optional, will be useful later
3. Call exception handler or interrupt handler
   - These can be normal C functions
4. Restore mepc and mstatus
5. Restore registers from stack (incl. stack pointer)
6. Execute `mret` instruction (this re-enables interrupts)

trap.s

```
# trap.s
#

        .text
        .global _trap_entry
        .type  _trap_entry, @function

        .align 4 # Trap entry must be 4-byte aligned for mtvec CSR

_trap_entry:

        # Save current state to the stack.
        #
        #   struct trap_frame {
        #       uint64_t x[32]; // x[0] unused
        #       uint64_t mstatus;
        #       uint64_t mepc;
        #   };
        #
```

trap.s

```
_trap_entry:

        # Save current state to the stack.
        #
        #   struct trap_frame {
        #       uint64_t x[32]; // x[0] unused
        #       uint64_t mstatus;
        #       uint64_t mepc;
        #   };
        #



        nop                 # x0 is zero
        sd      x1, 1*8(sp)   # x1 is ra
        nop                 # x2 is sp
        sd      x3, 3*8(sp)   # x3 is gp
        sd      x4, 4*8(sp)   # x4 is tp
        sd      x5, 5*8(sp)   # x4 is t0

        #  and so on
```

trap.s

```
        sd      x30, 30*8(sp) # x30 is t5
        sd      x31, 31*8(sp) # x31 is t6

        # Save mstatus and mepc

        csrr    t0, mstatus
        sd      t0, 32*8(sp)
        csrr    t0, mepc
        sd      t0, 33*8(sp)



        # ALL HANDLERS RETURN HERE

       # ... restore registers

1:      csrr    a0, mcause           # put mcause in a0
        mv      a1, sp               # put pointer to trap_frame in a1
        bgez    a0, fault_handler
        j       intr_handler
```