Bridge of Life Education

SOC Design
Peripheral - Interrupt

Jiin Lai

Topics

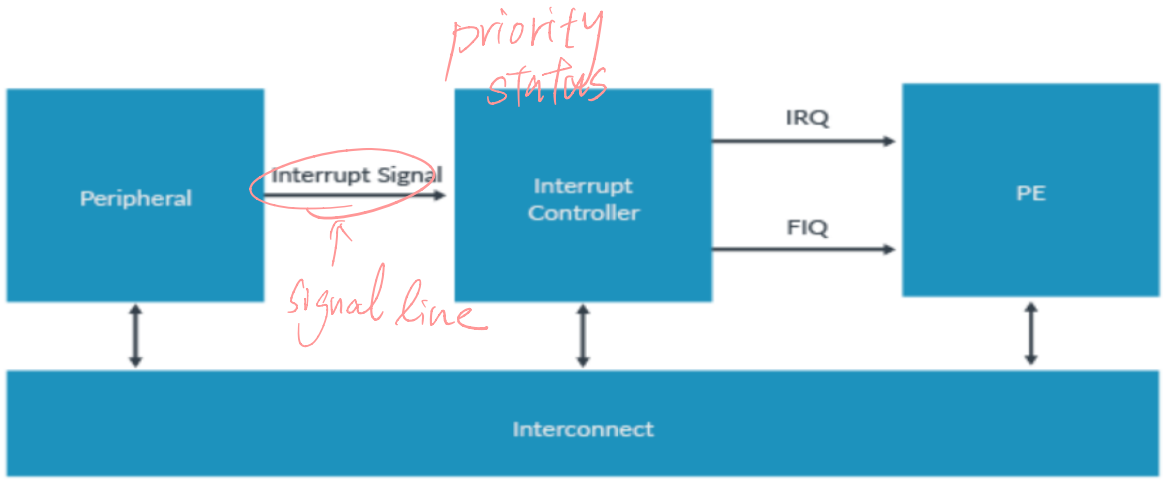
- Interrupt Basics
- RISC-V Interrupt Handling
- General Issues with Interrupt Handling
- ARM GIC (Supplement)
- Interrupt Controller 8259 (Supplement)

Interrupt Basics

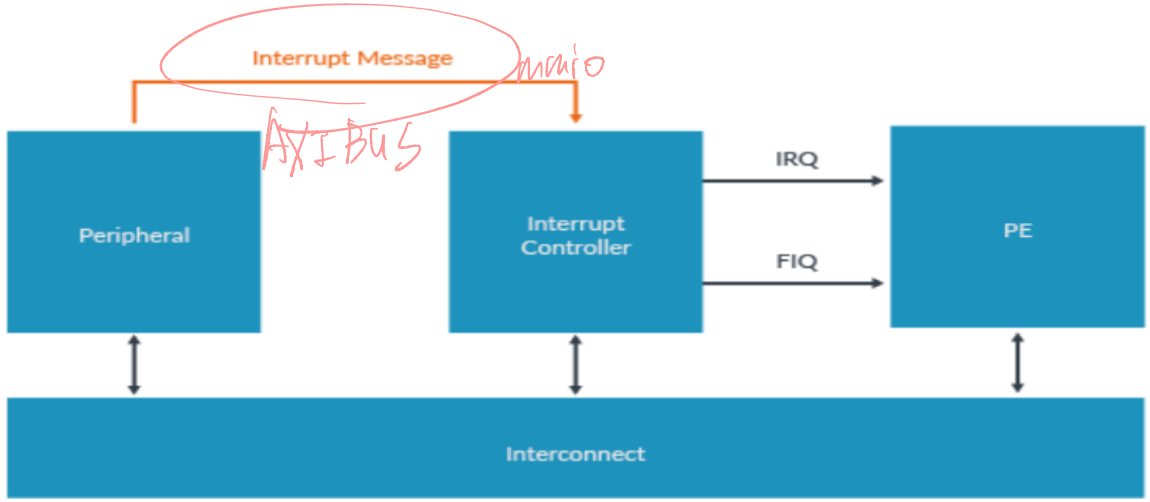
- Types of interrupts:
  - **Hardware interrupts**
    - ***Asynchronous***: not related to what code the processor is currently executing
    - Examples: interrupt is asserted, character is received on serial port, or ADC converter finishes conversion
  - **Exceptions, faults, software interrupts**
    - ***Synchronous***: are the result of specific instructions executing
    - Examples: undefined instructions, overflow occurs for a given instruction
  - We can enable and disable (*mask*) most interrupts as needed (***maskable***), others are ***non-maskable***
- Interrupt service routine (ISR)
  - Subroutine which processor is *forced to execute* to respond to a *specific event*
  - After ISR completes, MCU goes back to previously executing code

How Interrupt Signal to Processor

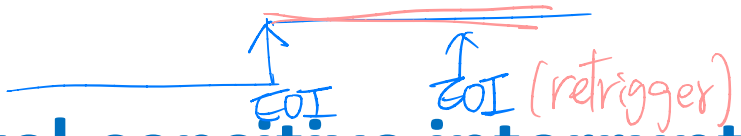
interrupts are signaled from a peripheral to the interrupt controller using a dedicated hardware signal,



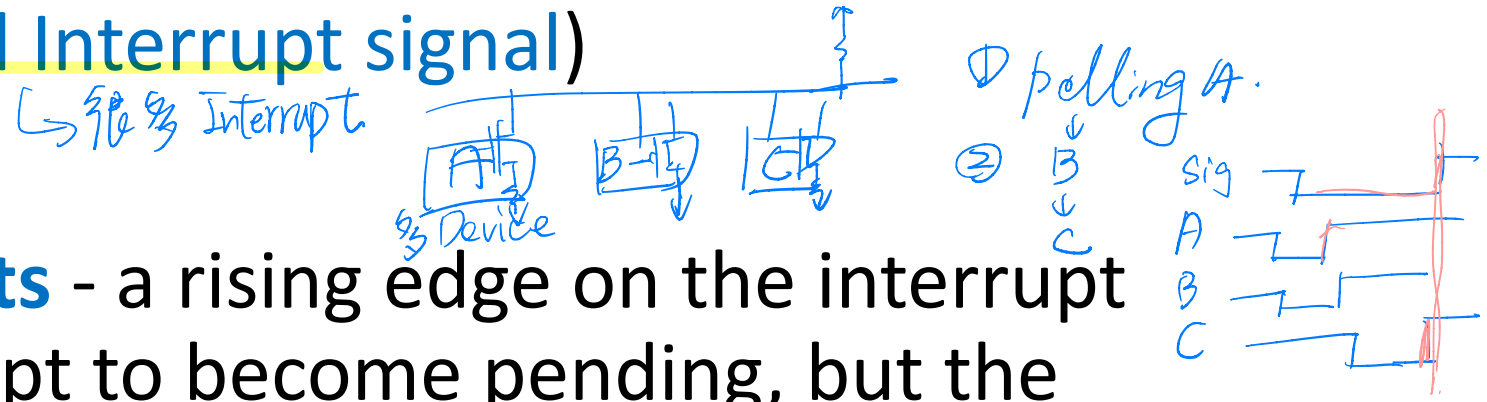
message-signaled interrupts (MSI). transmitted by a write to a register in the interrupt controller.



Life Cycle of Level-Trigger / Edge-Trigger Interrupt



- **Level-sensitive interrupts** - a rising edge on the interrupt input causes the interrupt to become pending, and the interrupt is held asserted until the peripheral de-asserts the interrupt signal. (Shared Interrupt signal)

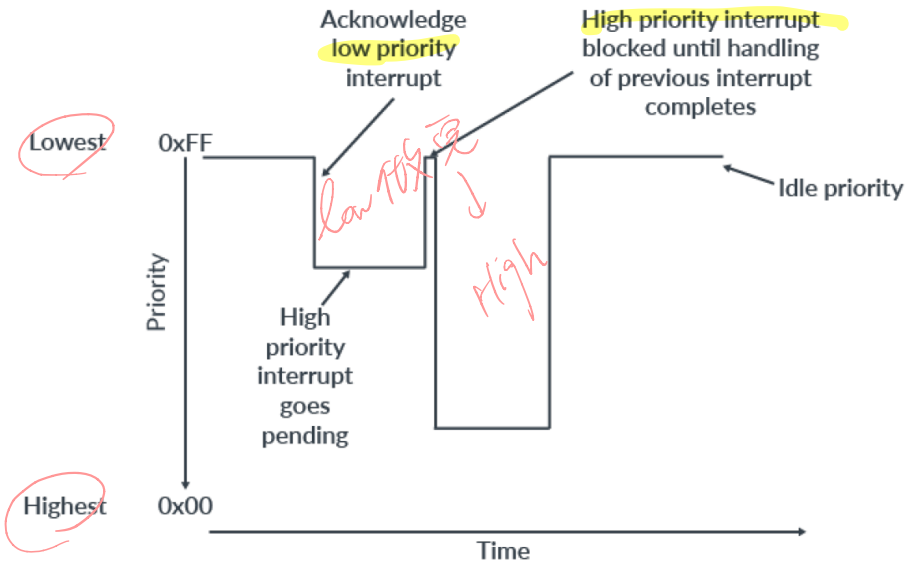


- **Edge-sensitive interrupts** - a rising edge on the interrupt input causes the interrupt to become pending, but the interrupt is not held asserted.

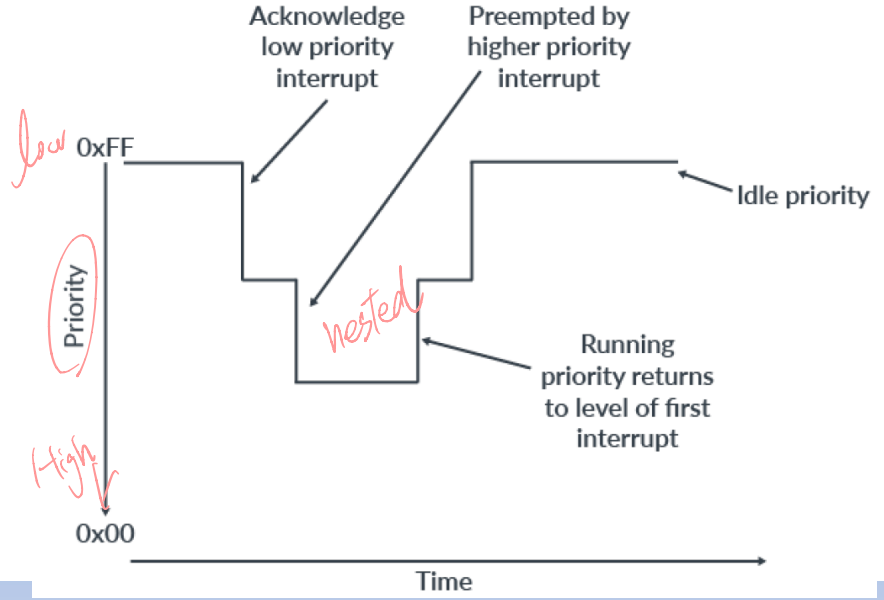
Interrupt Priority with Preemption

Preemption occurs when a high priority interrupt is signaled to a PE that is already handling a lower priority interrupt. Preemption introduces some additional complexity for software, but it can prevent a low priority interrupt from blocking the handling of a higher priority interrupt

Without Preemption



With Preemption

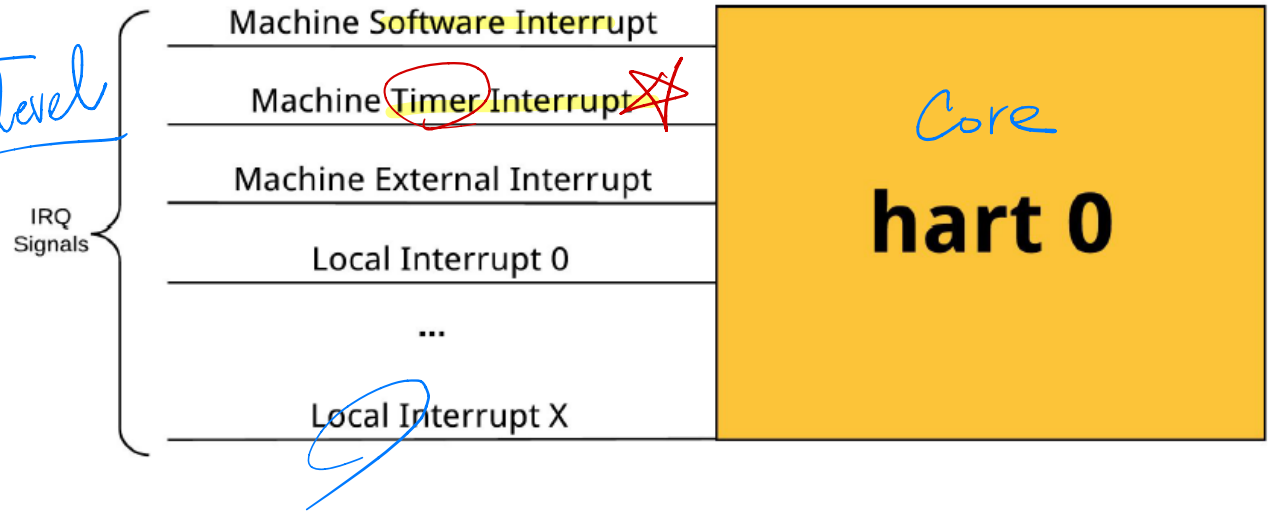


RISC-V Interrupt Handling

RISC-V Interrupts

- **RISC-V defines the following interrupts per Hart**
  - Software – architecturally defined software interrupt
  - Timer – architecturally defined timer interrupt
  - External – Peripheral Interrupts
  - Local - Hart specific Peripheral Interrupts
- **Optionally per privilege level**
  - Can have Supervisor Software/Timer/Machine Interrupts
  - Can have User Software/Timer/Machine
- **Local interrupts are optional and implementation specific**
  - Can be used for hart-specific peripheral interrupts
  - Useful for latency-sensitive embedded systems or small embedded systems with a small number of interrupts

Hart : RISC-V Hardware execution context



What are Control and Status Registers (CSRs)

processor state = thread switch
- PC
- CSR
- RF
- IM
- DM
- MMU

- **CSRs are Registers which contain the working state of a RISC-V machine**

- **CSRs are specific to a Mode**
  - Machine Mode has ~17 CSRs (not including performance monitor CSRs)
  - Supervisor Mode has a similar number, though most are subsets of their equivalent Machine Mode CSRs
    - Machine Mode can also access Supervisor CSRs

- **CSRs are defined in the RISC-V privileged specification**
  - We will cover a few key CSRs here

Interrupt Related CSR

mstatus : keeps track of and controls current operating state.

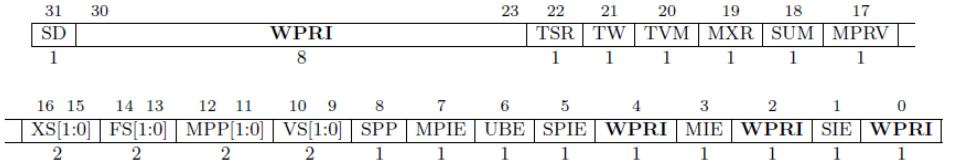
mtvec: Machine Trap-Vector Base-Address Register (BASE) and a vector mode (MODE).

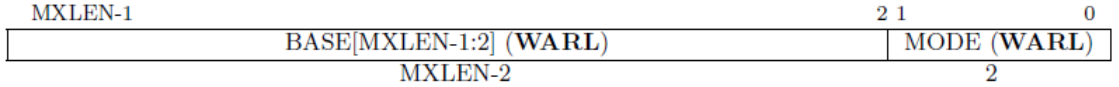
mip :pending interrupts,
mie : interrupt enable bits

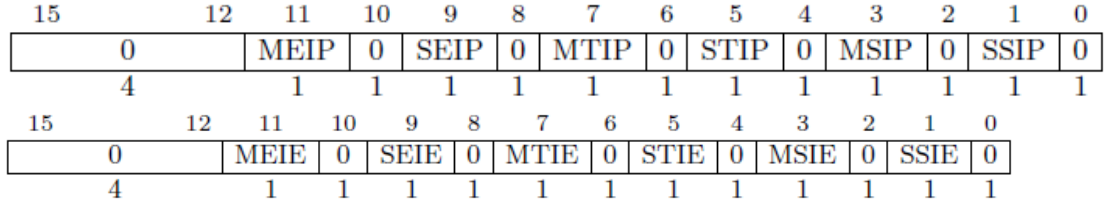
mepc: Machine Exception Program Counter

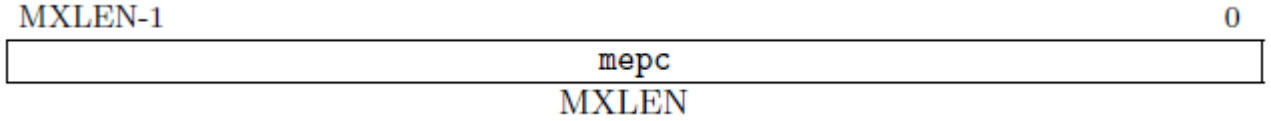
mcause: code indicating the event that caused the trap.

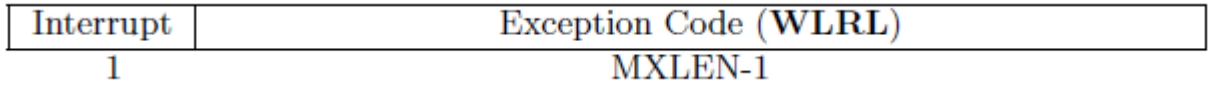
mtval: Machine Trap Value Register (mtval) exception-specific information to assist software in handling the trap

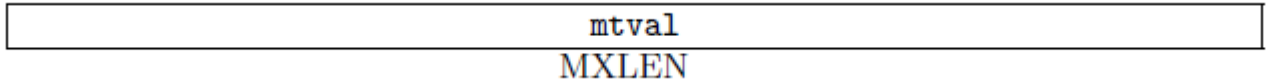












Machine Status (*mstatus*) – As it relates to Interrupts

Bits	Field Name	Description
0	UIE	User Interrupt Enable
1	SIE	Supervisor Interrupt Enable
2	Reserved	
3	MIE	Machine Interrupt Enable
4	UPIE	User Previous Interrupt Enable
5	SPIE	Supervisor Previous Interrupt Enable
6	Reserved	
7	MPIE	Machine Previous Interrupt Enabler
8	SPP	Supervisor Previous Privilege
[10:9]	Reserved	
[12:11]	MPP	Machine Previous Privilege

Bits	Field Name	Description
[14:13]	FS	Floating Point State
[16:15]	XS	User Mode Extension State
17	MPRIV	Modify Privilege (access memory as MPP)
18	SUM	Permit Supervisor User Memory Access
19	MXR	Make Executable Readable
20	TVM	Trap Virtual memory
21	TW	Timeout Wait (traps S-Mode wfi)
22	TSR	Trap SRET
[23:30]	Reserved	
[31]	SD	State Dirty (FS and XS summary bit)

Machine/supervisor/user level

RV32 *mstatus* CSR

- **M/S/U IE – Global Interrupt Enables for Modes which supports interrupts**
- **M/S/U PIE – Encodes the state of interrupt enables prior to an interrupt.**
  - These bits can also be written to in order to enable interrupts when returning to lower privilege modes
- **M/S PP – Encodes the privilege level prior to the previous interrupt**
  - These bits can also be written to in order to enter a lower privilege mode when executing MRET or SRET instructions

https://cdn2.hubspot.net/hubfs/3020607/An%20Introduction%20to%20the%20RISC-V%20Architecture.pdf

Machine Interrupt Cause CSR (*mcause*)

- Interrupts are identified by reading the *mcause* CSR
- The *interrupt* field determines if a trap was caused by an interrupt or an exception

Bits	Field Name	Description
XLEN-1	Interrupt	Identifies if an interrupt was synchronous or asynchronous
[XLEN-2:0]	Exception Code	Identifies the exception

mcause CSR

Interrupt = 1 (interrupt)	
Exception Code	Description
0	User Software Interrupt
1	Supervisor Software Interrupt
2	Reserved
3	Machine Software Interrupt
4	User Timer Interrupt
5	Supervisor Timer Interrupt
6	*Reserved*
7	Machine Timer Interrupt
8	User External Interrupt
9	Supervisor External Interrupt
10	*Reserved*
11	Machine External Interrupt
12 - 15	*Reserved*
≥16	Local Interrupt X

Interrupt = 0 (exception)	
Exception Code	Description
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	Environment Call from U-mode
9	Environment Call from S-mode
10	Reserved
11	Environment Call from M-mode
12	Instruction Page Fault
13	Load Page Fault
14	Reserved
15	Store/AMO Page Fault
≥16	Reserved

Machine Interrupt-Enable and Pending CSRs (*mie*, *mip*)

- ***mie*** **used to enable/disable a given interrupt**
- ***mip*** **indicates which interrupts are currently pending**
  - Can be used for polling
- **Lesser-privilege bits in *mip* are writeable**
  - i.e. Machine-mode software can be used to generate a supervisor interrupt by setting the STIP bit
- ***mip*** **has the same mapping as *mie***

Bits	Field Name	Description
0	USIE	User Software Interrupt Enable
1	SSIE	Supervisor Software Interrupt Enable
2	*Reserved*	
3	MSIE	Machine Software Interrupt Enable
4	UTIE	User Timer Interrupt Enable
5	STIE	Supervisor Timer Interrupt Enable
6	*Reserved*	
7	MTIE	Machine Timer Interrupt Enable
8	UEIE	User External Interrupt Enable
9	SEIE	Supervisor External Interrupt Enable
10	*Reserved*	
11	MEIE	Machine External Interrupt Enable
12-15	*Reserved*	
≥16	LIE	Local Interrupt Enable

mie CSR

Machine Trap Vector CSR (*mtvec*)

2种

***mtvec* sets the Base interrupt vector and the interrupt Mode**

Bits	Field Name	Description
[XLEN-1:6]	Base	Machine Trap Vector Base Address. 64-byte Alignment
[1:0]	Mode	MODE Sets the interrupt processing mode.

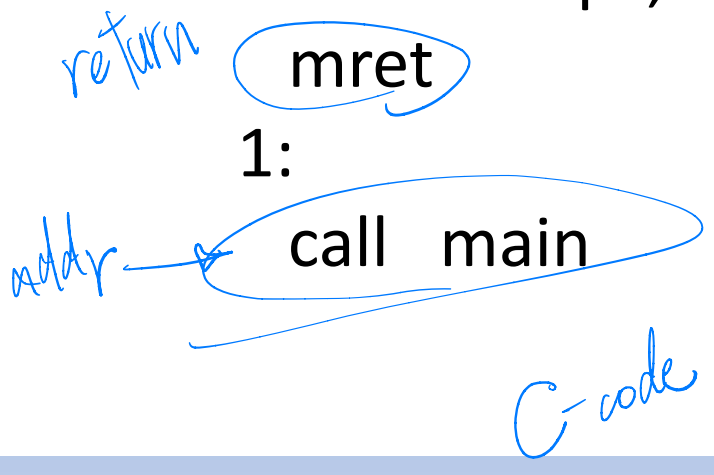
mtvec CSR

mtvec Modes		
Value	Name	Description
0x0	Direct	All Exceptions set PC to mtvec.BASE Requires 4-Byte alignment
0x1	Vectored	Asynchronous interrupts set pc to mtvec.BASE + (4×mcause.EXCCODE) Requires 4-Byte alignment
> 0x01		Reserved

- ① ***mtvec*.Mode = Direct**
  - All Interrupts trap to the address *mtvec*.Base
  - Software must read the *mcause* CSR and react accordingly
- ②***mtvec*.Vectored**
  - Interrupts trap to the address *mtvec*.Base + (4**mcause.ExCode*)
  - Eliminates the need to read mcause for asynchronous exceptions

in the startup script, load trap handle vector to the register mtvec and switch to User mode

```
la      t0, trap_entry
csrw    mtvec, t0
lla     t0, 1f
csrw    mepc, t0
mret
1:
call    main
```



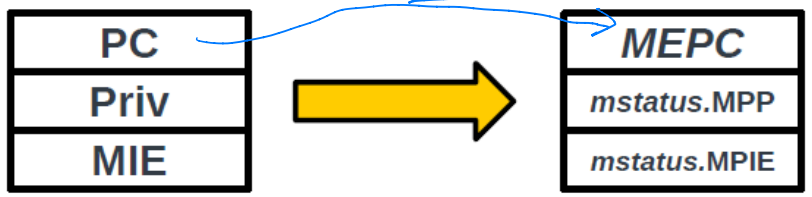
https://mullerlee.cyou/2020/07/09/riscv-exception-interrupt/

Trap Handler – Entry and Exit

mtevc.MODE = Direct

Hardware

- **On entry, the RISC-V hart will**
  - Save the current state



  - Then set PC = *mtvec*, *mstatus*.MIE = 0

- **MRET instruction restores state**

Software

- Typical trap handler software will

```
Push Registers
…
interrupt = mcause.msb
if interrupt
  branch isr_handler[mcause.code]
else
  branch exception_handler[mcause.code]
…
Pop Registers
MRET
```

Interrupt handler pseudo code

Interrupt Entry (done in Hardware)

1. capture the interrupted pc into a CSR — called **mepc**
2. capture the current privilege level into a CSR
3. set the interrupt cause CSR — called **mcause**
4. if the exception was due to a page fault then **mtval** holds the fault address
5. turn off interrupts — **mie**
6. look up the interrupt handler in the vector table specified by a CSR — called **mtvec**
7. and transfer control (set the pc) to the ISR

Interrupt Handler Code

RISC-V Assembly interrupt handler to Push and Pop register file

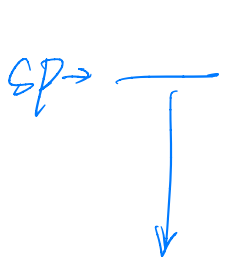
```
  .align 2
  .global trap_entry
trap_entry:
  addi sp, sp, -16*REGBYTES

  //store ABI Caller Registers
  STORE x1,   0*REGBYTES(sp)
  STORE x5,   2*REGBYTES(sp)
      …
  STORE x30, 14*REGBYTES(sp)
  STORE x31, 15*REGBYTES(sp)

  //call C Code Handler
  call handle_trap

  //restore ABI Caller Registers
  LOAD x1,   0*REGBYTES(sp)
  LOAD x5,   2*REGBYTES(sp)
      …
  LOAD x30, 14*REGBYTES(sp)
  LOAD x31, 15*REGBYTES(sp)

  addi sp, sp, 16*REGBYTES
  mret
```

C Code Handler determines interrupt cause and branches to the appropriate function

```
void handle_trap()
{
    unsigned long mcause = read_csr(mcause);
    if (mcause & MCAUSE_INT) {
            //mask interrupt bit and branch to handler
            isr_handler[mcause & MCAUSE_CAUSE] ();
    } else {
            //branch to handler
            exception_handler[mcause]();
    }
}


//write trap_entry address to mtvec
write_csr(mtvec, ((unsigned long)&trap_entry));
```

Upon entry the interrupt handler: manually backup and restore current context with the stack and use MRET to return to User mode

```
.align 2
trap entry:
    addi sp, sp, -17*4

    sw ra, 0*REGBYTES(sp)
    sw a0, 1*REGBYTES(sp)
    sw a1, 2*REGBYTES(sp)
    sw a2, 3*REGBYTES(sp)
    sw a3, 4*REGBYTES(sp)
    sw a4, 5*REGBYTES(sp)
    sw a5, 6*REGBYTES(sp)
    sw a6, 7*REGBYTES(sp)
    sw a7, 8*REGBYTES(sp)
    sw t0, 9*REGBYTES(sp)
    sw t1, 10*REGBYTES(sp)
    sw t2, 11*REGBYTES(sp)
    sw t3, 12*REGBYTES(sp)
    sw t4, 13*REGBYTES(sp)
    sw t5, 14*REGBYTES(sp)
    sw t6, 15*REGBYTES(sp)

    jal handle_trap
```



```
    lw ra, 0*REGBYTES(sp)
    lw a0, 1*REGBYTES(sp)
    lw a1, 2*REGBYTES(sp)
    lw a2, 3*REGBYTES(sp)
    lw a3, 4*REGBYTES(sp)
    lw a4, 5*REGBYTES(sp)
    lw a5, 6*REGBYTES(sp)
    lw a6, 7*REGBYTES(sp)
    lw a7, 8*REGBYTES(sp)
    lw t0, 9*REGBYTES(sp)
    lw t1, 10*REGBYTES(sp)
    lw t2, 11*REGBYTES(sp)
    lw t3, 12*REGBYTES(sp)
    lw t4, 13*REGBYTES(sp)
    lw t5, 14*REGBYTES(sp)
    lw t6, 15*REGBYTES(sp)

    addi sp, sp, 17*4

    mret
```

Compiler Interrupt Attribute

- **Pushing and Popping Registers in Assembly is a pain**
- **The *interrupt* attribute was added to GCC to facilitate interrupt handlers written entirely in C**
  - Interrupt functions only saves/restores necessary registers onto the stack
  - Align function on an 8-byte boundary
  - Calles MRET after popping register file back off the stack

Interrupt handler with *interrupt* attribute.
No assembly Code necessary

```
void handle_trap(void) __attribute((interrupt));
 void handle_trap()
 {
     unsigned long mcause = read_csr(mcause);
     if (mcause & MCAUSE_INT) {
            //mask interrupt bit and branch to handler
            isr_handler[mcause & MCAUSE_CAUSE] ();
     } else {
            //synchronous exception, branch to handler
            exception_handler[mcause & MCAUSE_CAUSE]();
     }
 }


 //write handle_trap address to mtvec
 write_csr(mtvec, ((unsigned long)&handle_trap));
```

ARM Entering exception handler: CPU hardwired

- Finish current instruction
  - Except for lengthy instructions
- Push context (registers) onto current stack (MSP or PSP)
  - xPSR, Return Address, LR(R14), R12, R3, R2, R1, R0
- Switch to handler/privileged mode, use MSP
- Load PC with address of interrupt handler
- Load LR with EXC_RETURN code
- Load IPSR with exception number
- **Start executing code of interrupt handler**

Usually **16 cycles** from exception request to execution of first instruction in handler

General Issues with Interrupt Handling

Interrupt response latency

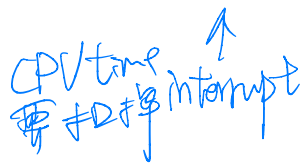
- Finish executing the current instruction or abandon it
- Push various registers on to the stack, fetch vector
  - Overhead for responding to each interrupt
- If we have external memory with wait states, this takes longer

- Why do we care?
  - This is overhead which wastes time, and increases as the interrupt rate rises.
  - This delays our response to external events, which may or may not be acceptable for the application, such as sampling an analog waveform.

Maximum interrupt rate

- We can only handle so many interrupts per second
  - F_{Max_Int}: maximum interrupt frequency
  - F_{CPU}: CPU clock frequency
  - C_{ISR}: Number of cycles ISR takes to execute
  - $C_{Overhead}$: Number of cycles of overhead for saving state, vectoring, restoring state, etc.
  - $\mathbf{F_{Max\_Int} = F_{CPU}/(C_{ISR} + C_{Overhead})}$
  - Note that model applies only when there is one interrupt in the system

- When processor is responding to interrupts, it isn't executing our other code
  - U_{Int}: Utilization (fraction of processor time) consumed by interrupt processing
  - CPU looks like it's running the other code with CPU clock speed of $\mathbf{(1-U_{Int})*F_{CPU}}$



Program design with interrupts

- How much work to do in ISR?
  - Trade-off: faster response for ISR code will delay completion of other code
  - In system with multiple ISRs with short deadlines, **perform critical work in ISR and buffer partial results for later processing**

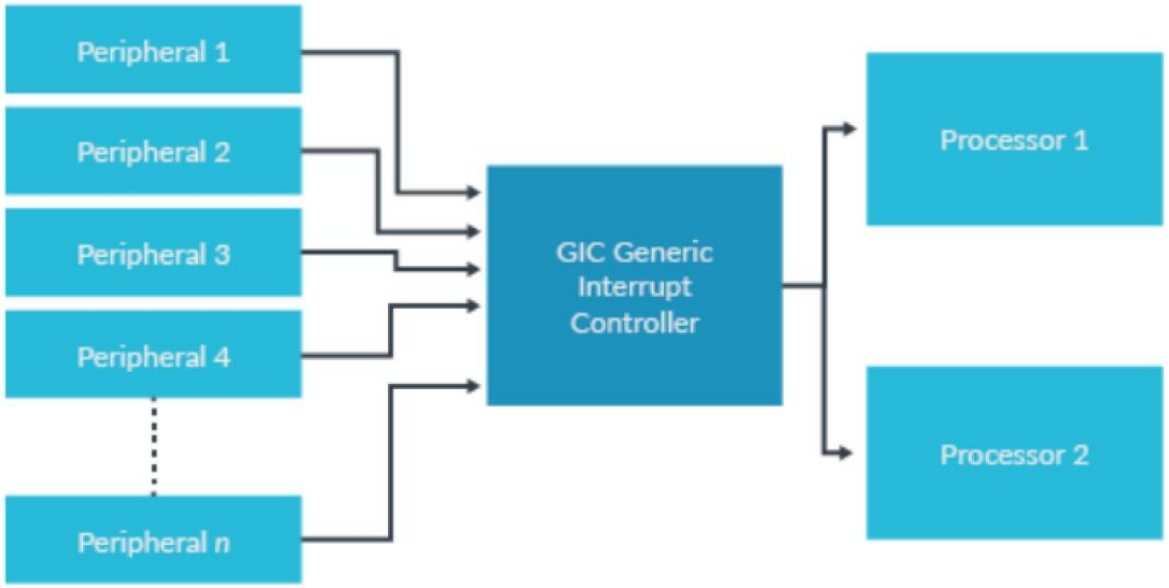
- **Should ISRs re-enable interrupts?**

- How to communicate between ISR and other threads?
  - Data buffering
  - Data integrity and race conditions
    - **Volatile data** – can be updated outside of the program's immediate control
    - **Non-atomic shared data** – can be interrupted partway through read or write, is vulnerable to race conditions

Supplement

ARM Generic Interrupt Controller (GIC)

The Use of GIC

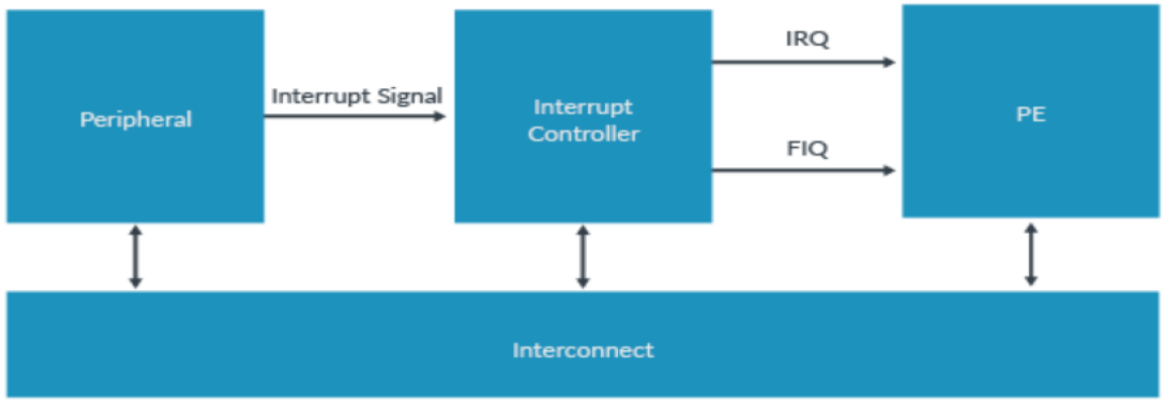


Interrupt Type

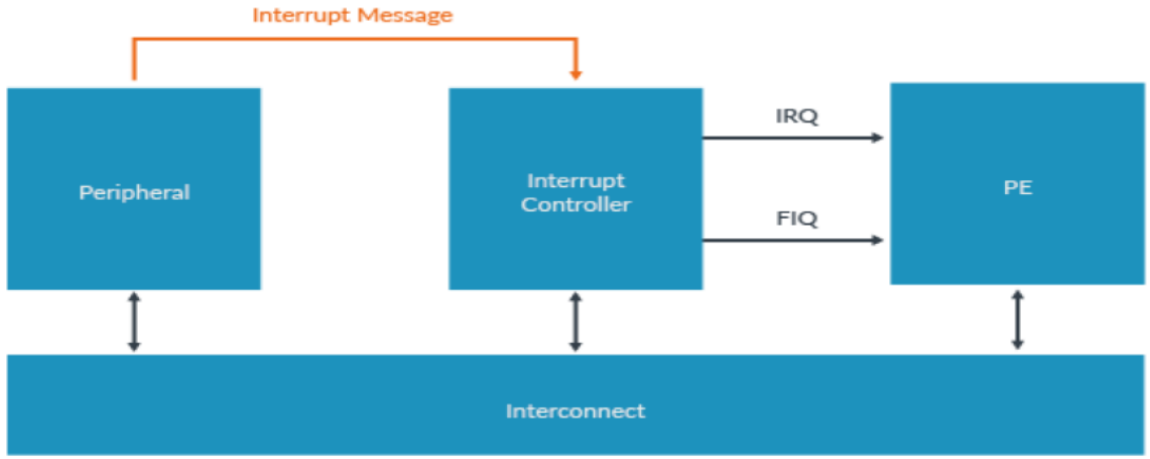
1. **Shared Peripheral Interrupt (SPI).** Peripheral interrupts that can be delivered to any connected core.
2. **Private Peripheral Interrupt (PPI).** Peripheral interrupts that are private to one core. An example of a PPI is an interrupt from the Generic Timer.
3. **Software Generated Interrupt (SGI).** SGIs are typically used for inter-processor communication and are generated by a write to an SGI register in the GIC.
4. **Locality-specific Peripheral Interrupt (LPI).** LPIs were first introduced in GICv3 and have a very different programing model to the other three types of interrupt. The configuration of LPIs is covered in the Arm CoreLink Generic Interrupt Controller v3 and v4: Locality-specific Peripheral Interrupts guide.

How Interrupt Signal to Processor

interrupts are signaled from a peripheral to the interrupt controller using a dedicated hardware signal,

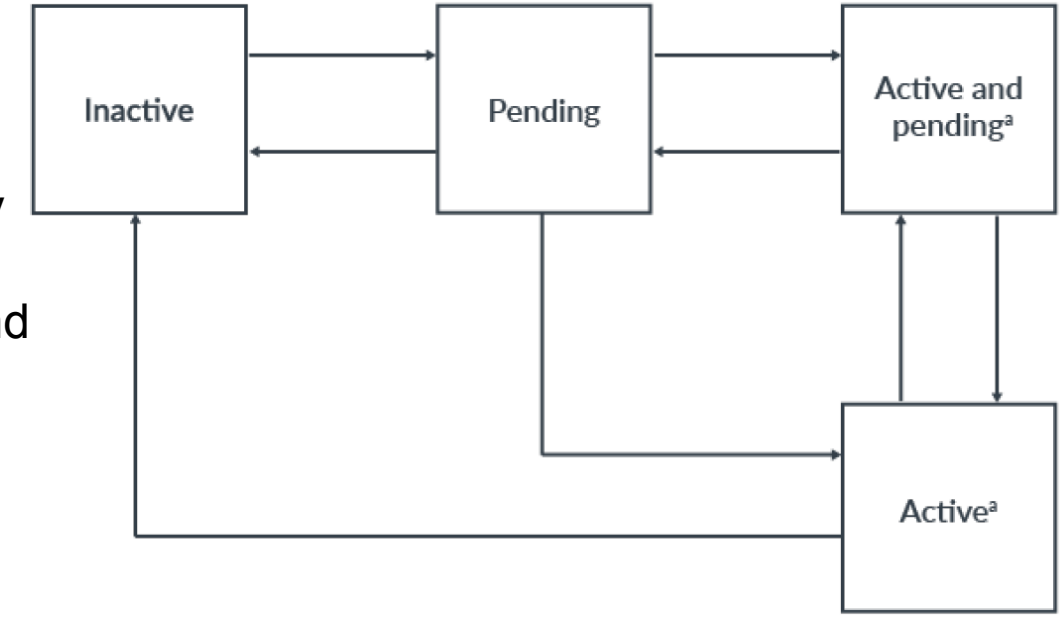


message-signaled interrupts (MSI). transmitted by a write to a register in the interrupt controller.



Interrupt State Machine

- **Inactive**. The interrupt source is not currently asserted.
- **Pending**. The interrupt source has been asserted, but the interrupt has not yet been acknowledged by a PE.
- **Active**. The interrupt source has been asserted, and the interrupt has been acknowledged by a PE.
- **Active and Pending**. An instance of the interrupt has been acknowledged, and another instance is now pending.

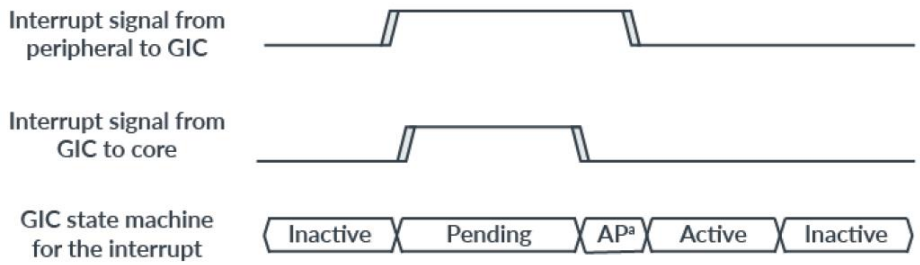


Life Cycle of Level-Trigger / Edge-Trigger Interrupt

- Level-sensitive interrupts - a rising edge on the interrupt input causes the interrupt to become pending, and the interrupt is held asserted until the peripheral de-asserts the interrupt signal.

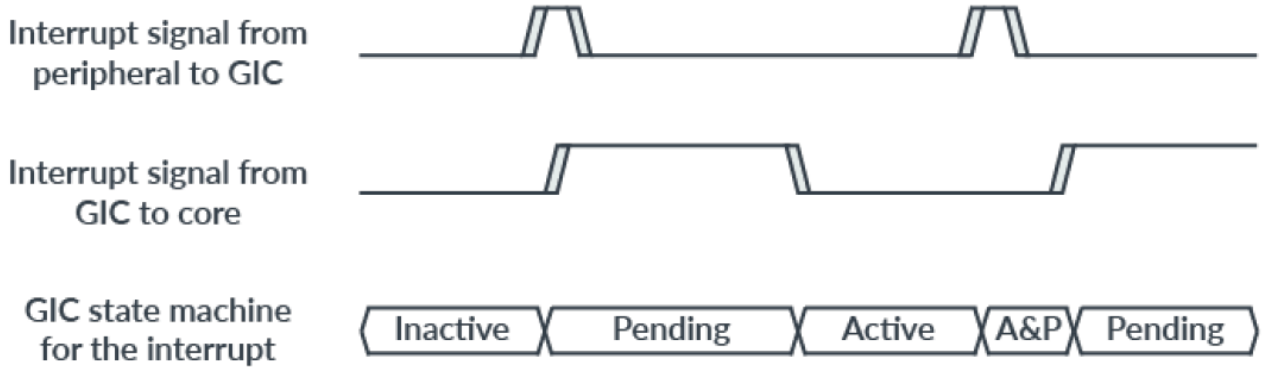
- Edge-sensitive interrupts - a rising edge on the interrupt input causes the interrupt to become pending, but the interrupt is not held asserted.

Level Interrupt



- Inactive to pending. An interrupt transitions from inactive to pending when the interrupt source is asserted. At this point the GIC asserts the interrupt signal to the PE, if the interrupt is enabled and is of sufficient priority.
- Pending to active and pending. An interrupt transitions from pending to active and pending when a Processor Element (PE) acknowledges the interrupt by reading one of the Interrupt Acknowledge Registers (IARs) in the CPU interface. This read is typically part of an interrupt handling routine that executes after an interrupt exception is taken. At this point the GIC deasserts the interrupt signal to the PE.
- Active and pending to active. An interrupt transitions from active and pending to active when the peripheral de-asserts the interrupt signal. This typically happens in response software writing to a status register in the peripheral.
- Active to inactive. An interrupt goes from active to inactive when the PE writes to one of the End of Interrupt Registers (EOIRs) in the CPU interface. This indicates that the PE has finished handling the interrupt.

Edge-Triggered Interrupt

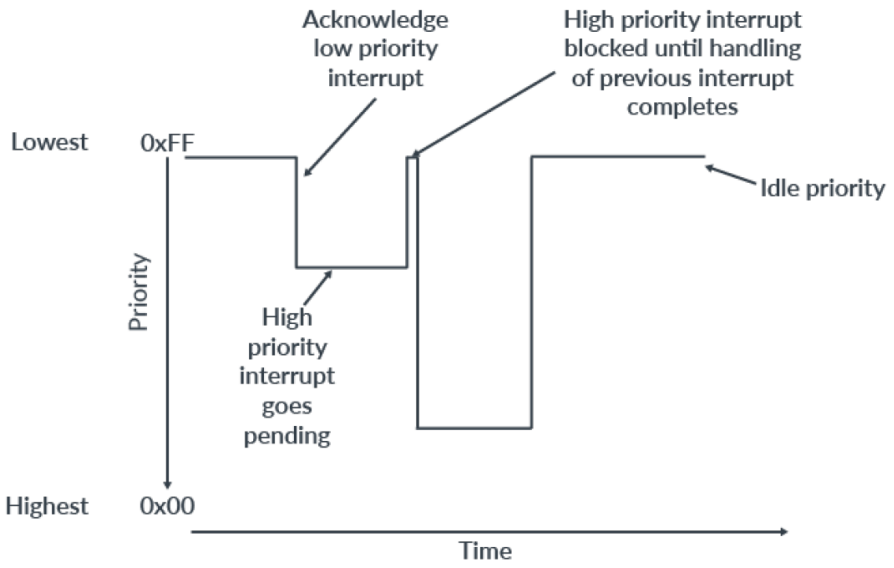


- Inactive to pending. An interrupt transitions from inactive to pending when the interrupt source is asserted. At this point the GIC asserts the interrupt signal to the PE, if the interrupt is enabled and is of sufficient priority.
- Pending to active. An interrupt transitions from pending to active when a PE acknowledges the interrupt by reading one of the IARs in the CPU interface. This read is typically part of an interrupt handling routine that executes after an interrupt exception is taken. However, software can also poll the IARs. At this point the GIC de-asserts the interrupt signal to the PE.
- Active to active and pending. An interrupt goes from active to active and pending if the peripheral re-asserts the interrupt signal.
- Active and pending to pending. An interrupt goes from active and pending to pending when the PE writes to one of the EOIRs in the CPU interface. This indicates that the PE has finished handling the first instance of the interrupt. At this point the GIC re-asserts the interrupt signal to the PE.

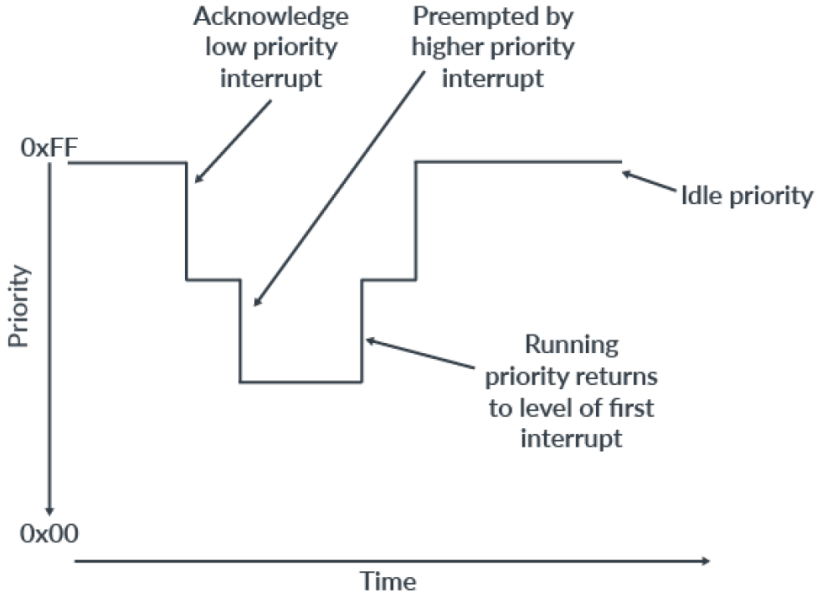
Interrupt Priority with Preemption

Preemption occurs when a high priority interrupt is signaled to a PE that is already handling a lower priority interrupt. Preemption introduces some additional complexity for software, but it can prevent a low priority interrupt from blocking the handling of a higher priority interrupt

Without Preemption



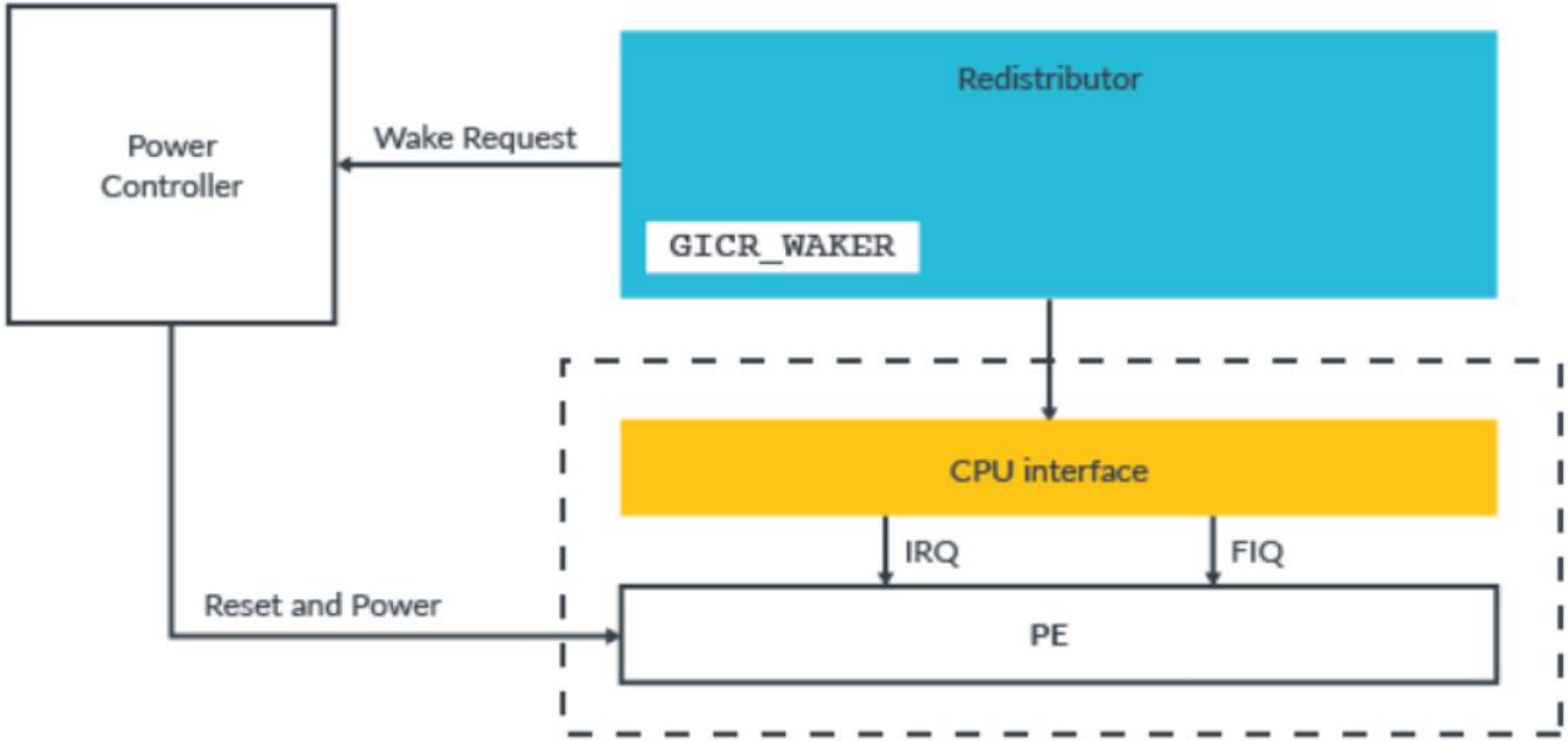
With Preemption



Interrupt Handling

- Routing a pending interrupt to a PE
  - Check interrupt (and its interrupt group) is enable
  - Decide which PEs can receive the interrupt
  - Check interrupt priority and priority mask
  - Check running priority which PEs are available to handle the interrupt
- PE Taking an interrupt – exception handler
  - Decide which interrupt taken – reading Interrupt Acknowledge Register (IAR)
  - Running priority and preemption
  - Service routine
- End of Interrupt
  - Priority drop
  - Deactivation

Interrupt to Wakeup Processor



ARM Interrupt Mechanism

- Refer to the ppt below
- https://github.com/arm-university/Introduction-to-SoC-Design-Education-Kit/blob/main/CortexM0_DesignStart/contents/Module08_InterruptMechanisms/Lecture08_InterruptMechanisms.pptx

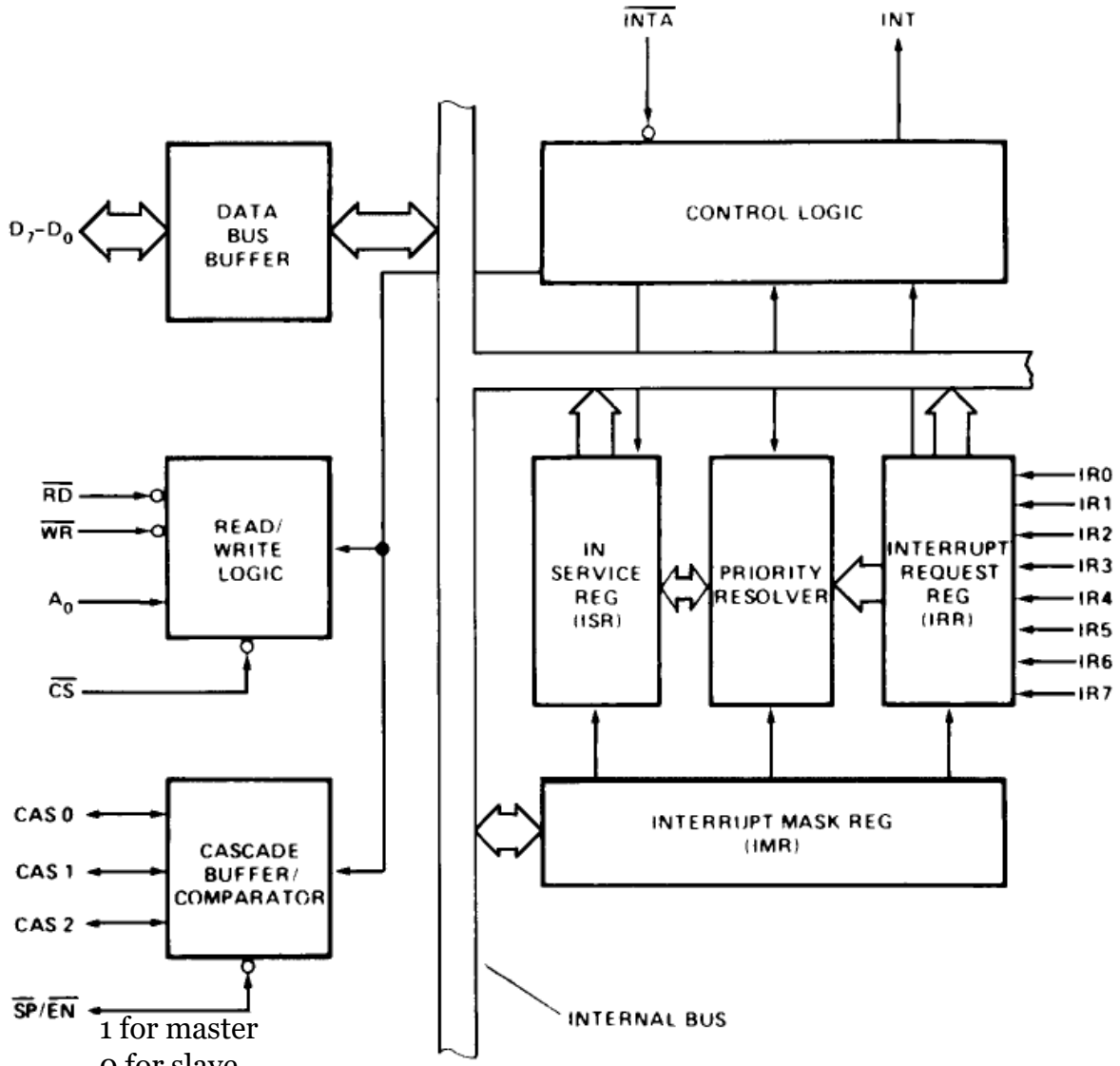
Programmable Interrupt Controller (PIC) - 8259

https://pdos.csail.mit.edu/6.828/2018/readings/hardware/8259A.pdf

Key Features

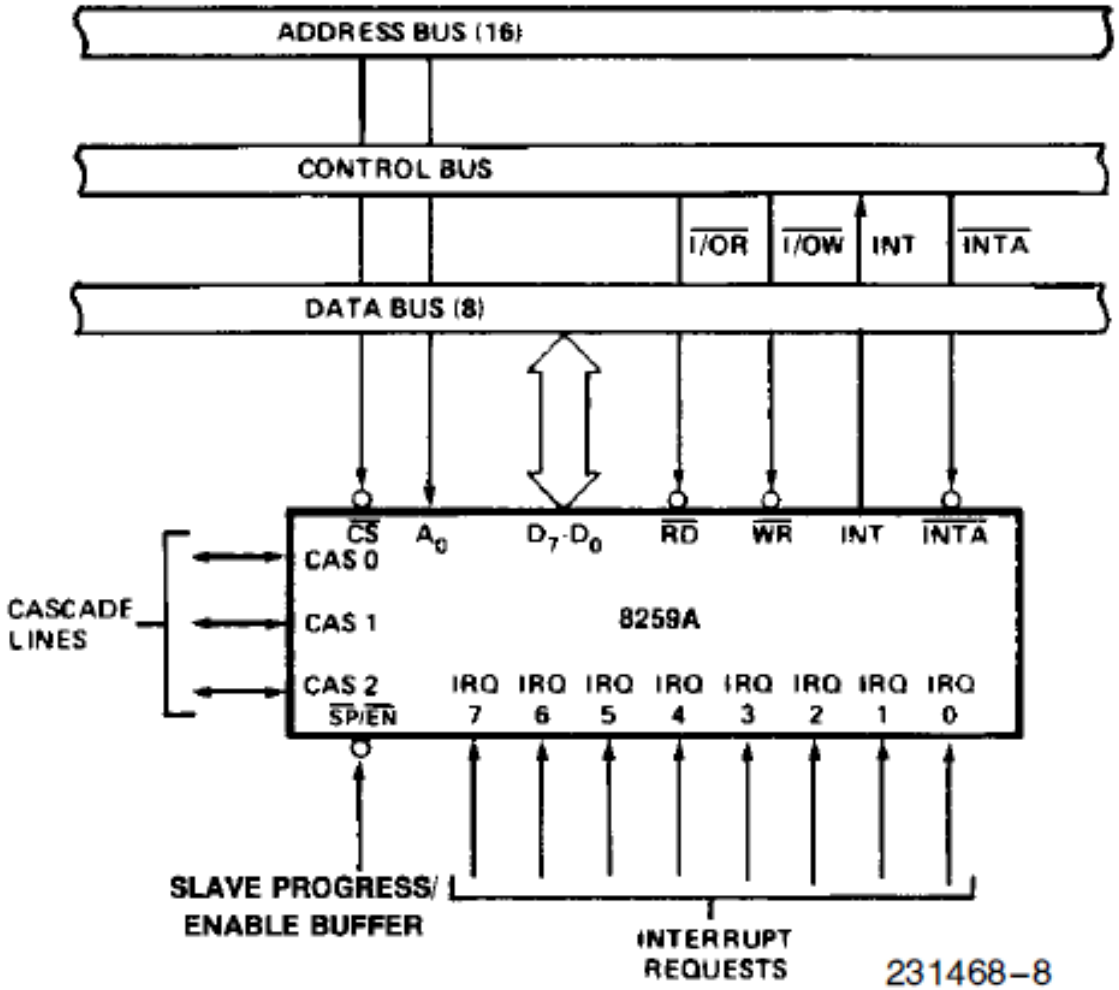
- 8086, 8088 Compatible
- MCS-80, MCS-85 Compatible
- Eight-Level Priority Controller
- Expandable to 64 Levels
- Programmable Interrupt Modes
- Individual Request Mask Capability

Block Diagram



1 for master
0 for slave

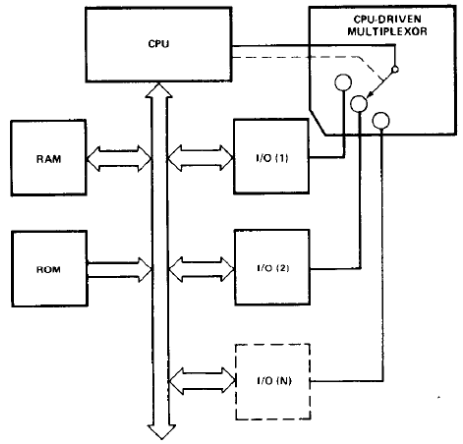
System Interface



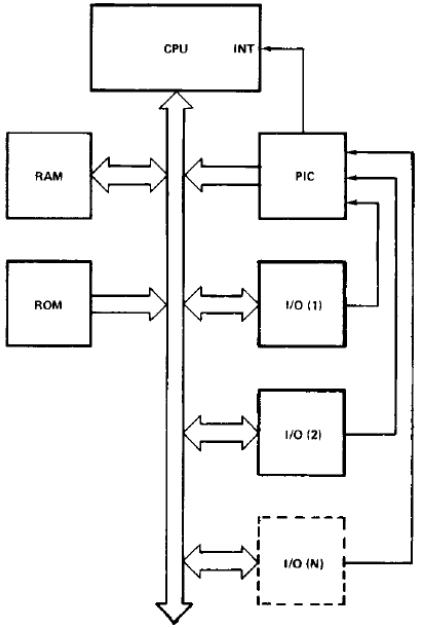
How Processor services Devices?

- Polled
  - Processor “ask” each device if it needs servicing.
  - Detrimental to system throughput
- Interrupt
  - Devices tell processor by an external asynchronous input (INTR signal)
  - Processor complete current instruction and fetch a new routine to service the device.
  - Once the service is complete, processor resume exactly where it left off.

Polled



Interrupt



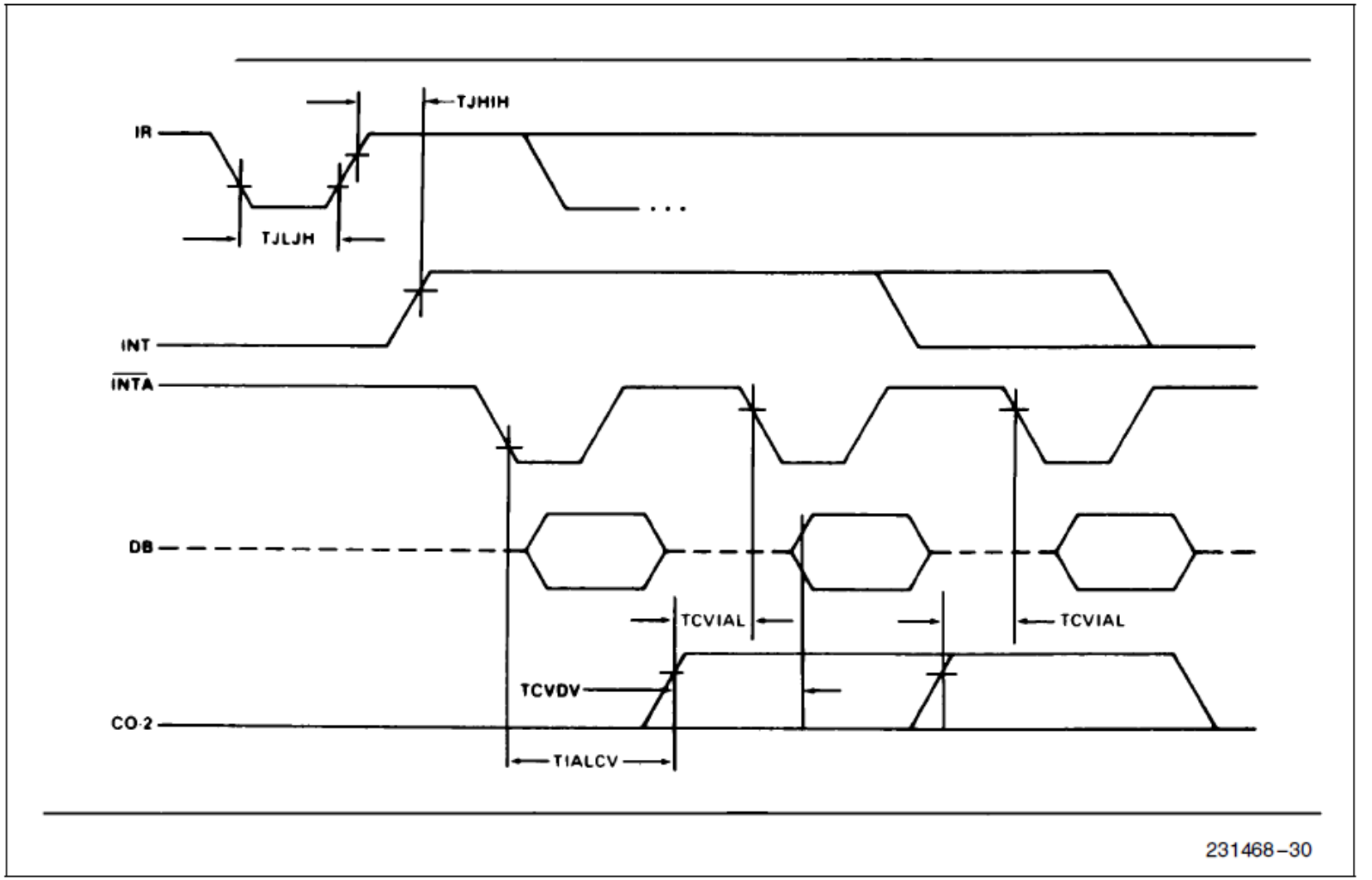
Role of PIC

- Accept requests from the peripherals
- Determine which incoming requests has priority to serve
- Issue interrupt to the processor
- Tell processor which service routine to execute
  - point the program counter to it
  - The “pointer” is an address in a vectoring table. Refer to interrupt vector.

Interrupt Sequence

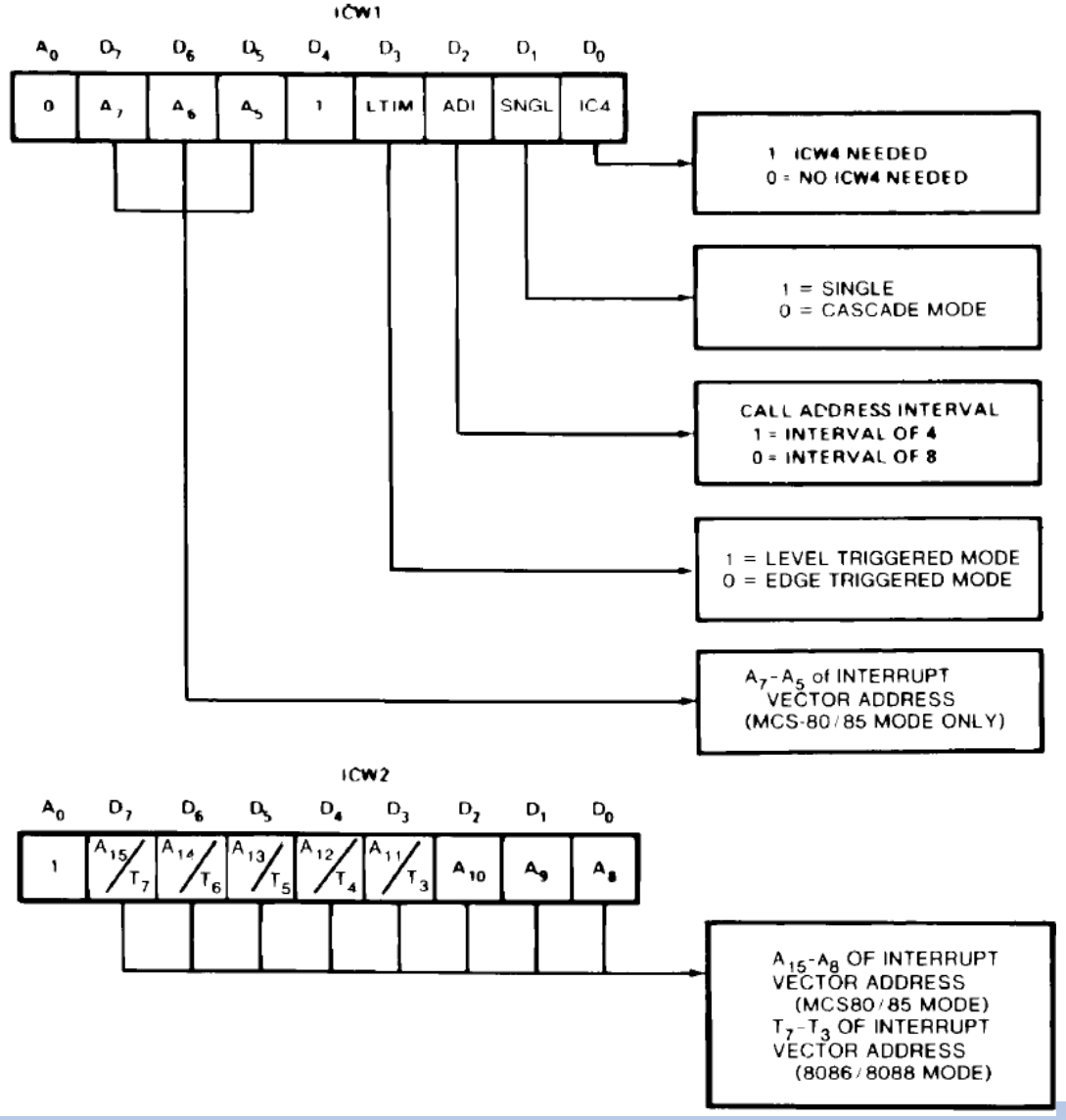
1. One or more of IR7-0One are raised high, setting the corresponding IRR bit(s).
2. The 8259A evaluates these requests, and sends an INT to the CPU.
3. The CPU acknowledges the INT and responds with an INTA pulse.
4. [8080/8085] 3-byte CALL instruction sequence from INTA
    1. 1st INTA: highest priority ISR bit is set, and the corresponding IRR bit is reset. The 8259A releases a CALL instruction code (11001101) onto the 8-bit Data Bus through its D7±0 pins.
    2. 2nd INTA: 8259A releases preprogrammed subroutine address onto Data Bus (lower 8-bit address)
    3. 3rd INTA: 8259A releases higher 8-bit address onto Data Bus.
5. [8086] 8259A release interrupt vector
    1. 1st INTA: highest priority ISR bit is set, and the corresponding IRR bit is reset
    2. 2nd INTA: 8259A releases an 8-bit pointer onto the Data Bus where it is read by the CPU.
6. In the AEOI mode the ISR bit is reset at the end of the third INTA pulse. Otherwise, the ISR bit remains set until an appropriate EOI command is issued at the end of the interrupt sequence.

$\overline{\text{INTA}}$ SEQUENCE



NOTES:
Interrupt output must remain HIGH at least until leading edge of first $\overline{\text{INTA}}$.
1. Cycle 1 in 8086, 8088 systems, the Data Bus is not active.

PROGRAMMING THE 8259A - ICW (Initialization Command Word)



Content of Second Interrupt Vector Byte

IR	Interval = 4							
	D7	D6	D5	D4	D3	D2	D1	D0
7	A7	A6	A5	1	1	1	0	0
6	A7	A6	A5	1	1	0	0	0
5	A7	A6	A5	1	0	1	0	0
4	A7	A6	A5	1	0	0	0	0
3	A7	A6	A5	0	1	1	0	0
2	A7	A6	A5	0	1	0	0	0
1	A7	A6	A5	0	0	1	0	0
0	A7	A6	A5	0	0	0	0	0

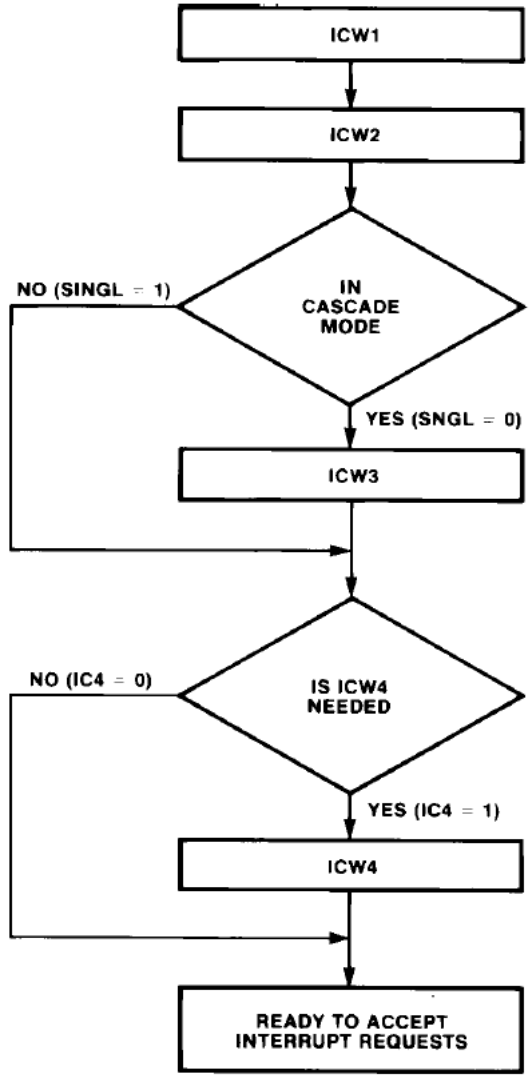
IR	Interval = 8							
	D7	D6	D5	D4	D3	D2	D1	D0
7	A7	A6	1	1	1	0	0	0
6	A7	A6	1	1	0	0	0	0
5	A7	A6	1	0	1	0	0	0
4	A7	A6	1	0	0	0	0	0
3	A7	A6	0	1	1	0	0	0
2	A7	A6	0	1	0	0	0	0
1	A7	A6	0	0	1	0	0	0
0	A7	A6	0	0	0	0	0	0

Content of Third Interrupt Vector Byte

D7	D6	D5	D4	D3	D2	D1	D0
A15	A14	A13	A12	A11	A10	A9	A8

Content of Interrupt Vector Byte for 8086 System Mode

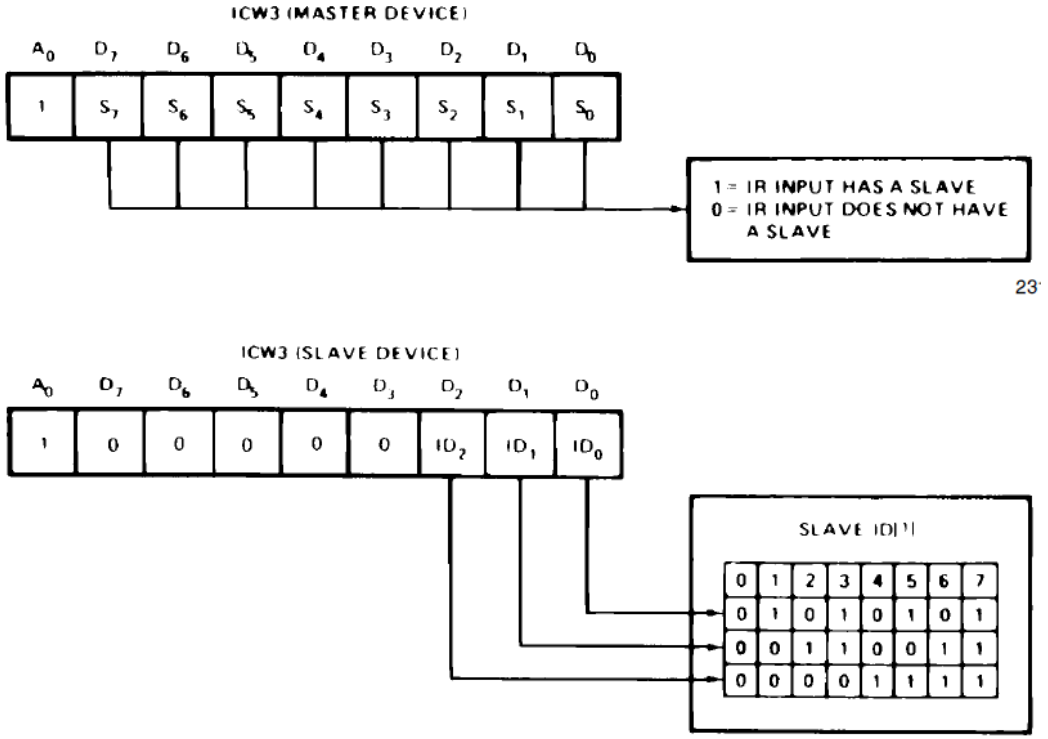
	D7	D6	D5	D4	D3	D2	D1	D0
IR7	T7	T6	T5	T4	T3	1	1	1
IR6	T7	T6	T5	T4	T3	1	1	0
IR5	T7	T6	T5	T4	T3	1	0	1
IR4	T7	T6	T5	T4	T3	1	0	0
IR3	T7	T6	T5	T4	T3	0	1	1
IR2	T7	T6	T5	T4	T3	0	1	0
IR1	T7	T6	T5	T4	T3	0	0	1
IR0	T7	T6	T5	T4	T3	0	0	0



ICW3 – Cascading is used

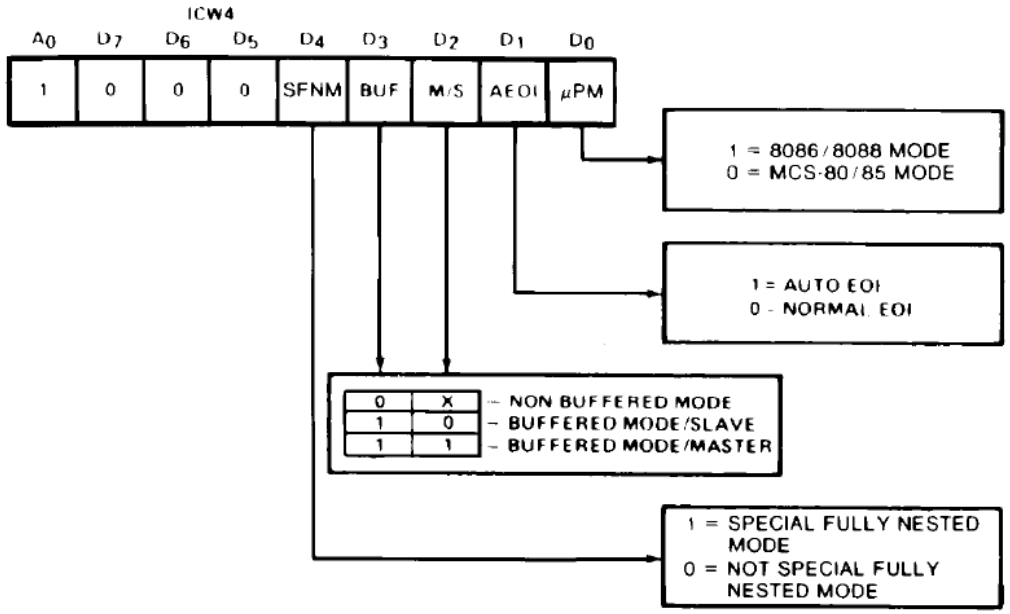
SNGL=0: This word is read only. It will load the 8-bit slave register. The functions of this register are:

a. In the master mode (either when SP=1, or in buffered mode when M/S =1 in ICW4) a ``1" is set for each slave in the system. The master then will release byte 1 of the call sequence (for MCS-80/85 system) and will enable the corresponding slave to release bytes 2 and 3 (for 8086 only byte 2) through the cascade lines.
b. In the slave mode (either when SP=0, or if BUF=1 and M/S=0 in ICW4) bits 2±0 identify the slave. The slave compares its cascade input with these bits and, if they are equal, bytes 2 and 3 of the call sequence (or just byte 2 for 8086) are released by it on the Data Bus.



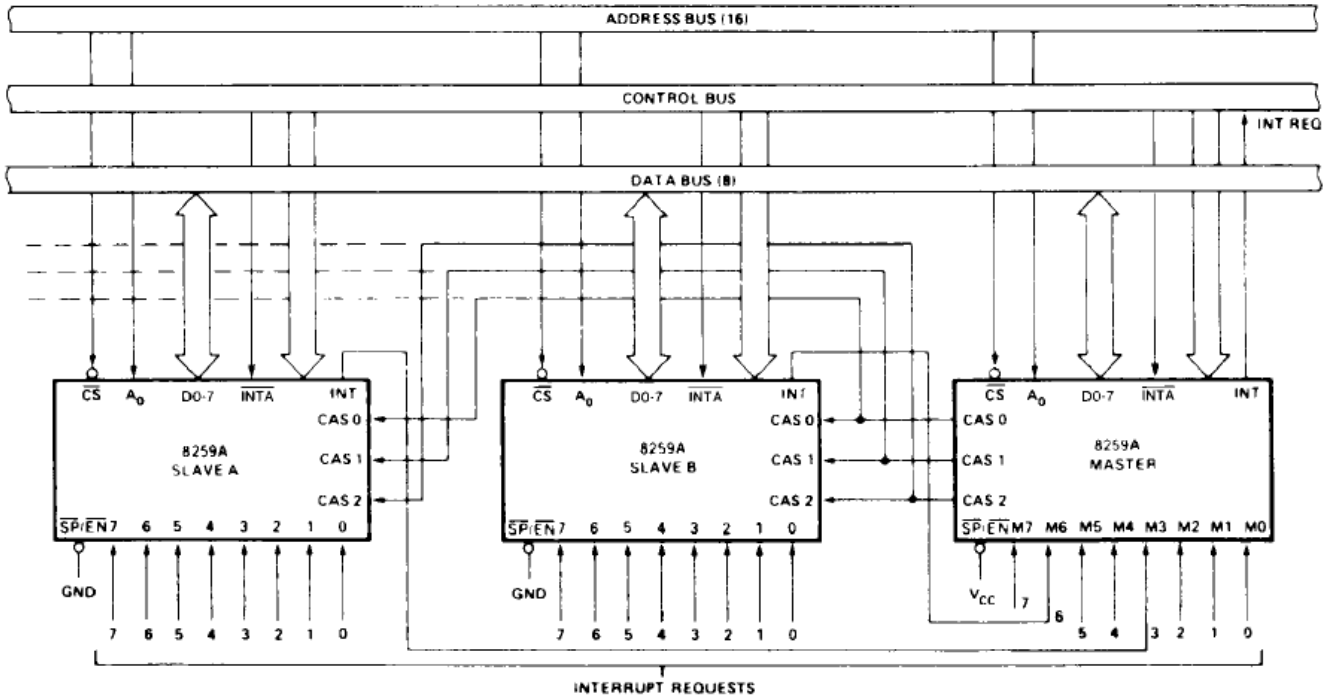
ICW4 –

- **SFNM**: Special fully nested mode
- **BUF**: If BUF = 1 the buffered mode is programmed. In buffered mode, SP/EN becomes an enable output and the master/slave determination is by M/S.
- **M/S**: If buffered mode is selected: M/S = 1 means the 8259A is programmed to be a master, M/S = 0 means the 8259A is programmed to be a slave. If BUF = 0, M/S has no function.
- **AEOI**: automatic end of interrupt mode
- **mPM**: Microprocessor mode:
  - mPM = 0: for MCS-80, 85
  - mPM = 1: for 8086.



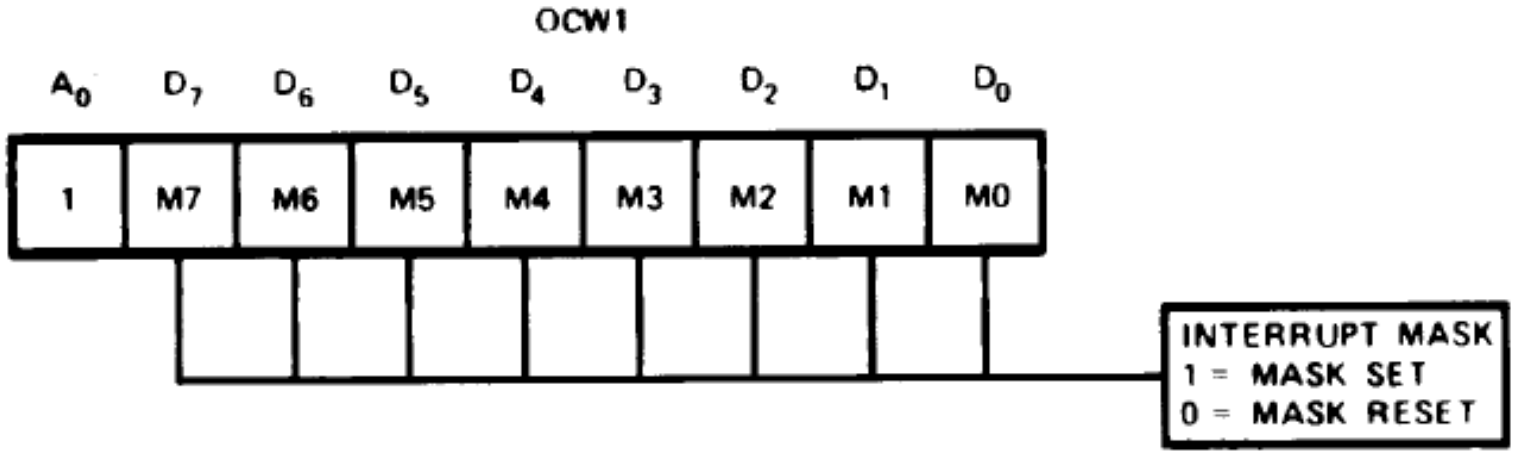
Cascading 8259

- One master with up to eight slaves to handle up to 64 priority levels.
- The 3 line cascade bus like chip selects to the slaves during the INTA sequence.
- The slave interrupt outputs are connected to the master interrupt request inputs
- Master will enable the corresponding slave to release the device routine address during bytes 2 and 3 of INTA.
- EOI command must be issued twice: once for the master and once for the corresponding slave.
- Address decoder is required to activate the Chip Select (CS) input of each 8259A.



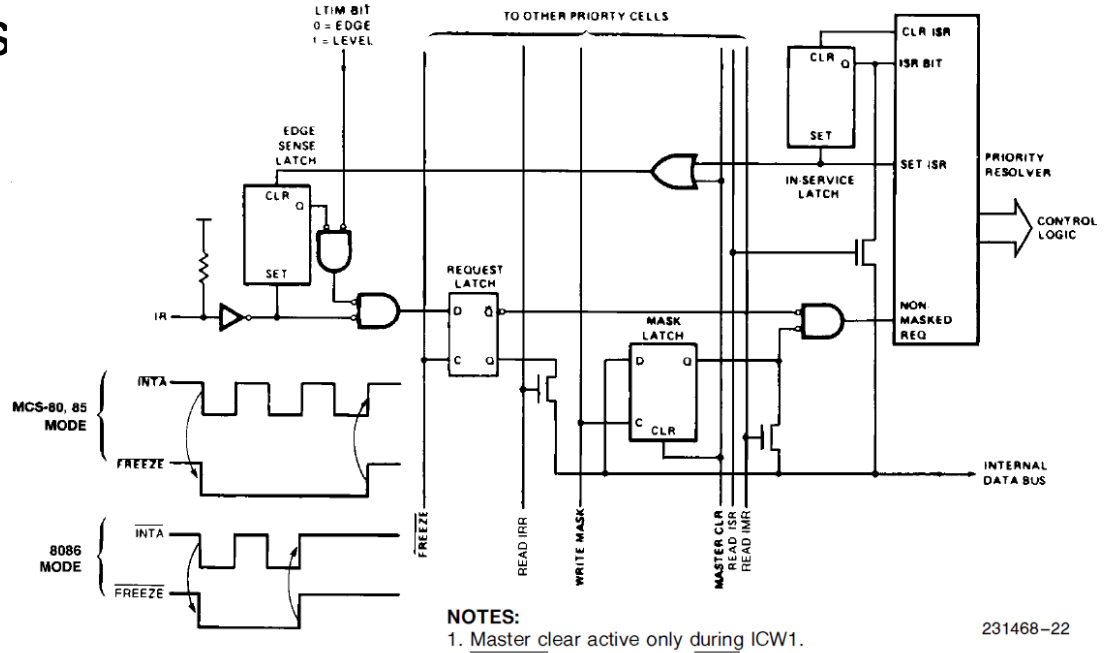
OPERATION COMMAND WORDS – OCW
OCW1 - IMR

Interrupt Mask Register (IMR)
M = 1 indicates the channel is masked (inhibited),
M = 0 indicates the channel is enabled.



Edge and Level Triggered Modes

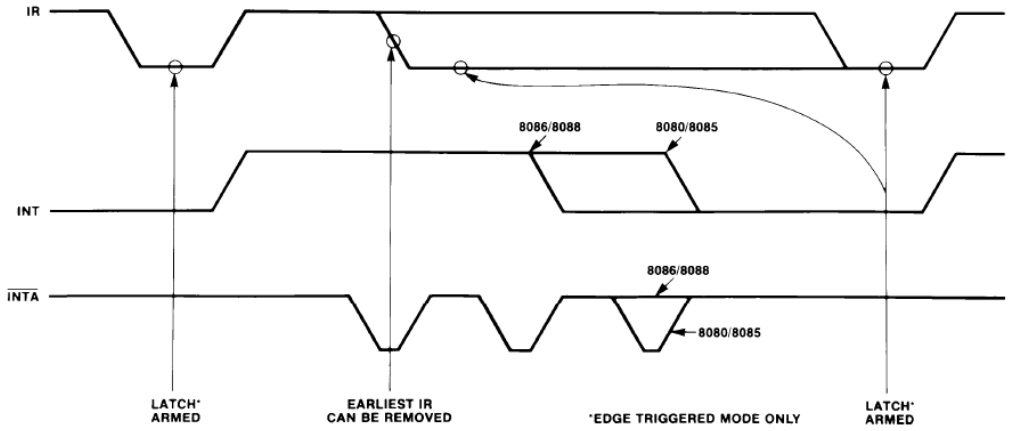
- **If LTIM = `0',** an interrupt request will be recognized by a low to high transition on an IR input. The IR input can remain high without generating another interrupt.
- **If LTIM =`1',** an interrupt request will be recognized by a `high' level on IR Input. The interrupt request must be removed before the EOI command is issued or the CPU interrupts is enabled to prevent a second interrupt from occurring.
- The IR inputs must remain high until after the falling edge of the first INTA. If the IR input goes low before this time a DEFAULT IR7 will occur when the CPU acknowledges.

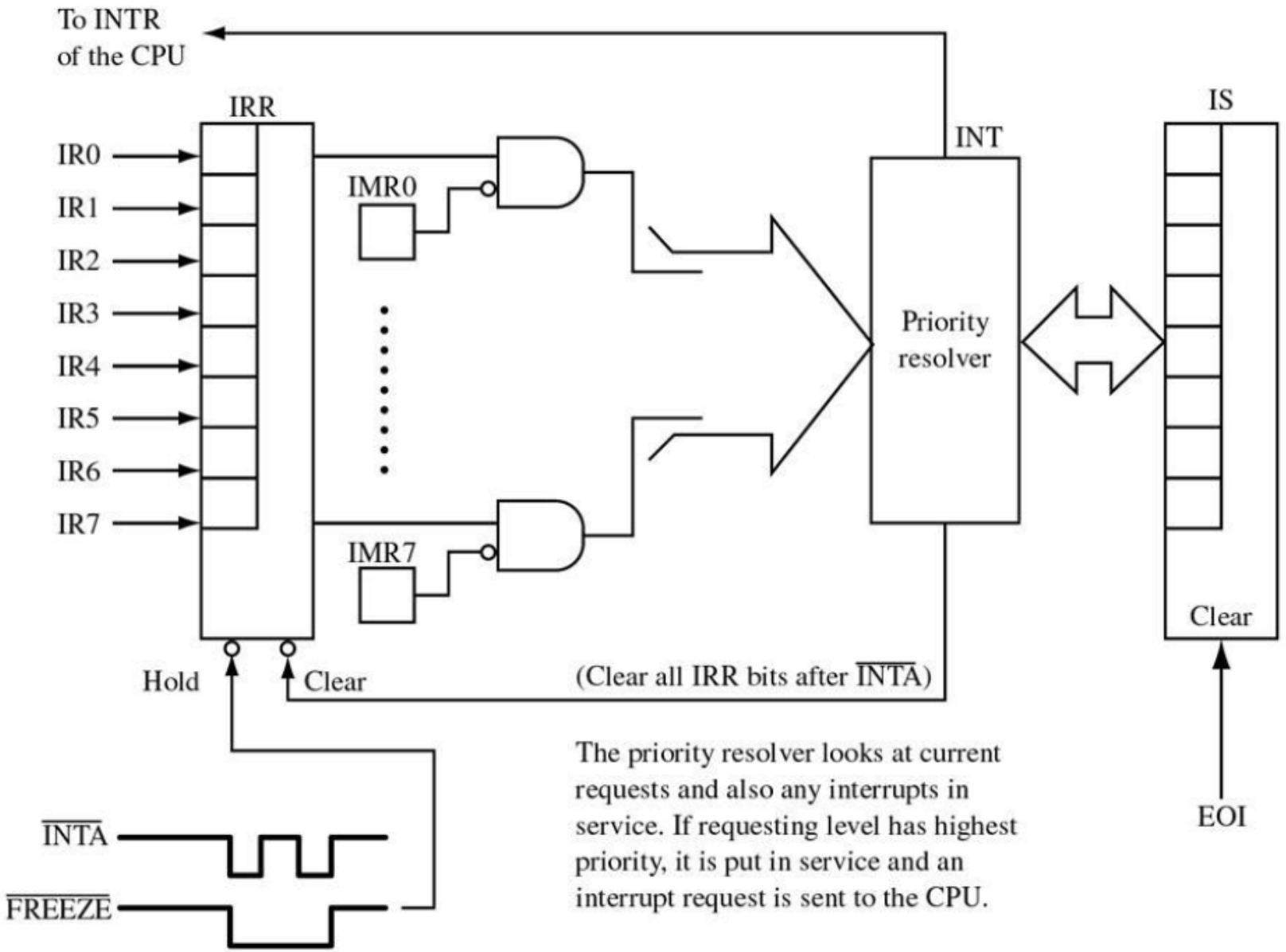


NOTES:
1. Master clear active only during ICW1.
2. FREEZE is active during INTA and poll sequences only.
3. Truth Table for a D-Latch.

C	D	Q	Operation
1	Di	Di	Follow
0	X	Qn-1	Hold

231468–22



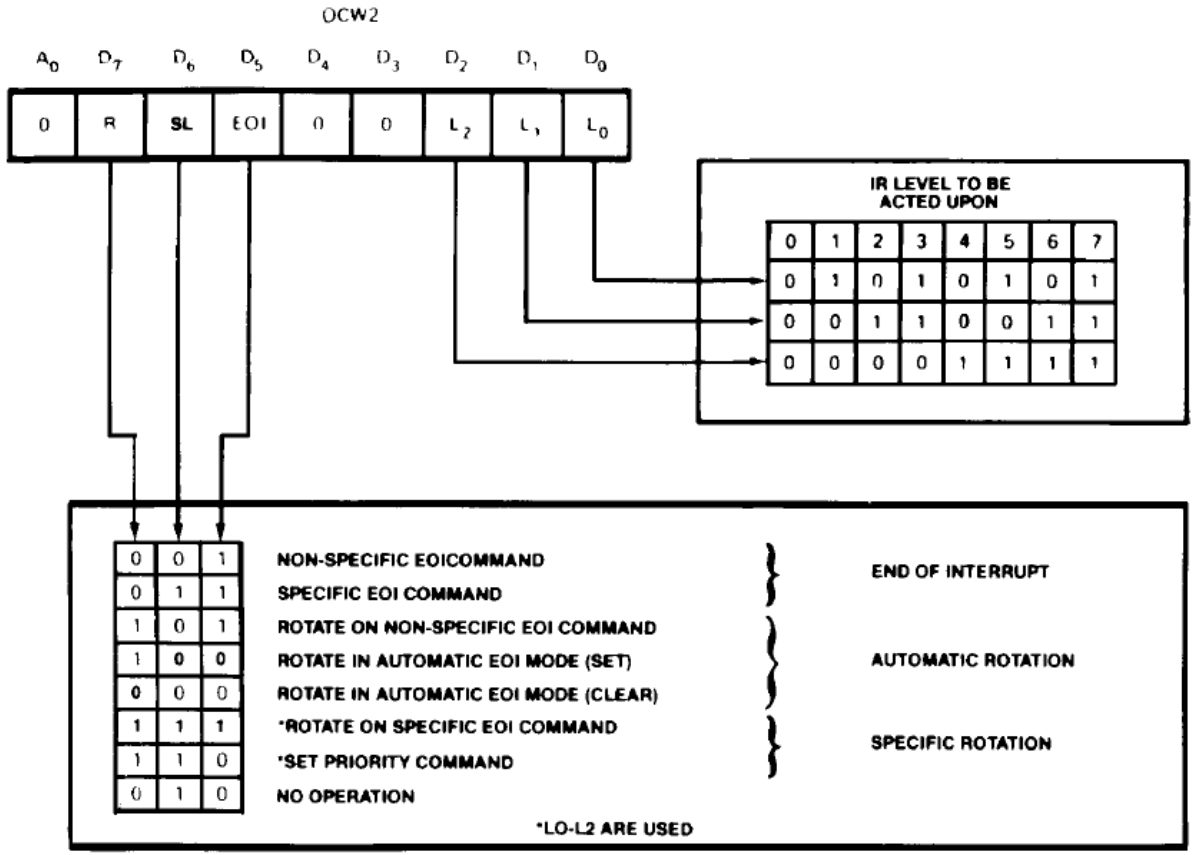


The priority resolver looks at current requests and also any interrupts in service. If requesting level has highest priority, it is put in service and an interrupt request is sent to the CPU.

OCW2

R, SL, EOI: three bits control the Rotate and End of Interrupt modes and combinations of the two.

L_2, L_1, L_0 - These bits determine the interrupt level acted upon when the SL bit is active.

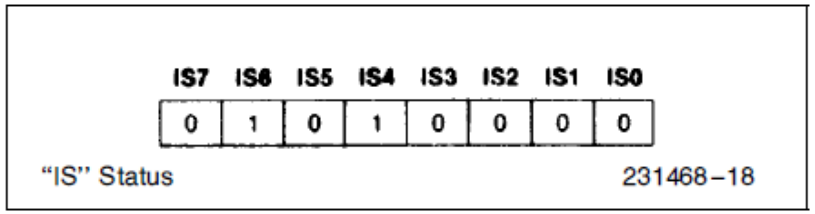


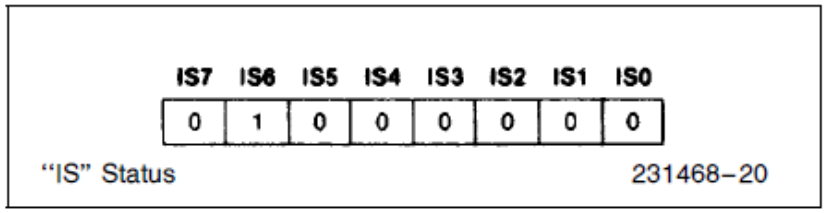
End-of-Interrupt (EOI), Automatic EOI

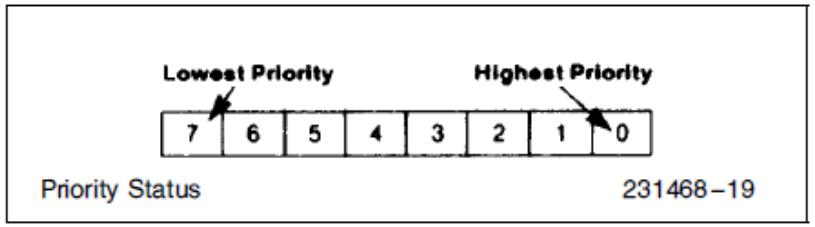
- **EOI** – Processor issues EOI to reset In Service (IS) bit before returning from the service routine.
- **Automatic EOI** – In Service (IS) bit is reset automatically following the trailing edge of the last in sequence INTA pulse
- **Non-specific** – automatically reset the highest IS bit (EOI=1, SL=0, R=0)
- **Specific** – (EOI=1, SL=1, R=0) and L0-L2 is the binary level of the IS bit to be reset.

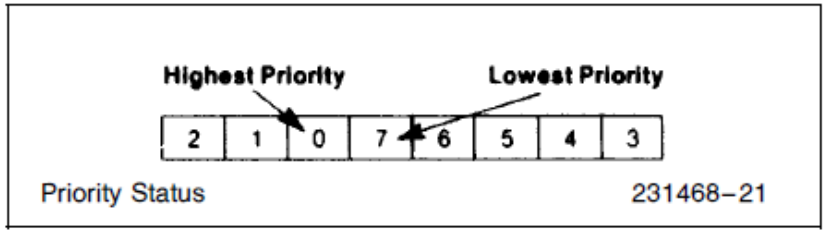
Priority Rotation

- **Automatic Rotation** (Equal Priority Devices)









- **Specific Rotation** - change priorities by programming the bottom priority and thus fixing all other priorities; i.e., if IR5 is programmed as the bottom priority device, then IR6 will have the highest one. (R=1, SL=1, L0-L2)

OCW3

Special Mask Mode - when a mask bit is set in OCW1, it inhibits further interrupts at that level and enables interrupts from all other levels (lower as well as higher) that are not masked.
Thus, any interrupts may be selectively enabled by loading the mask register.
The special Mask Mode is set by OWC3 where:
SSMM = 1, SMM = 1
Poll Command - Processor disable interrupt. Software using a Poll command. The 8259A treats the next RD pulse to the as an interrupt acknowledge, sets the appropriate IS bit if there is a request, and reads the priority level. Interrupt is frozen from WR to RD.