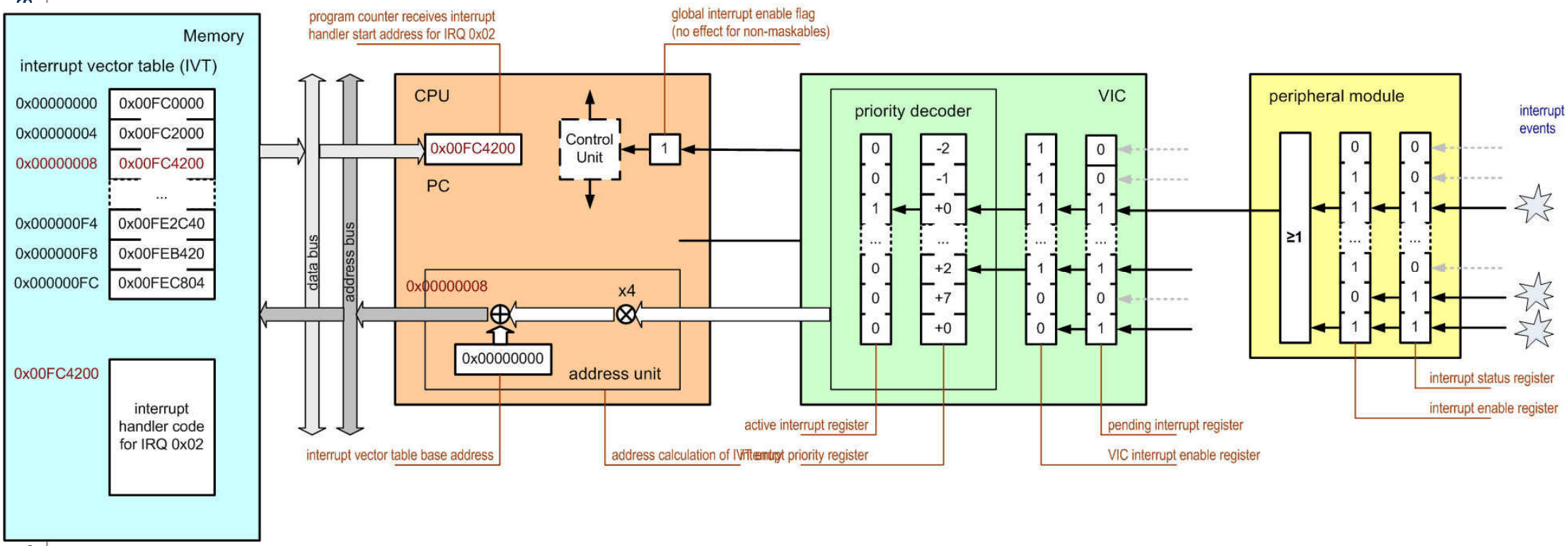
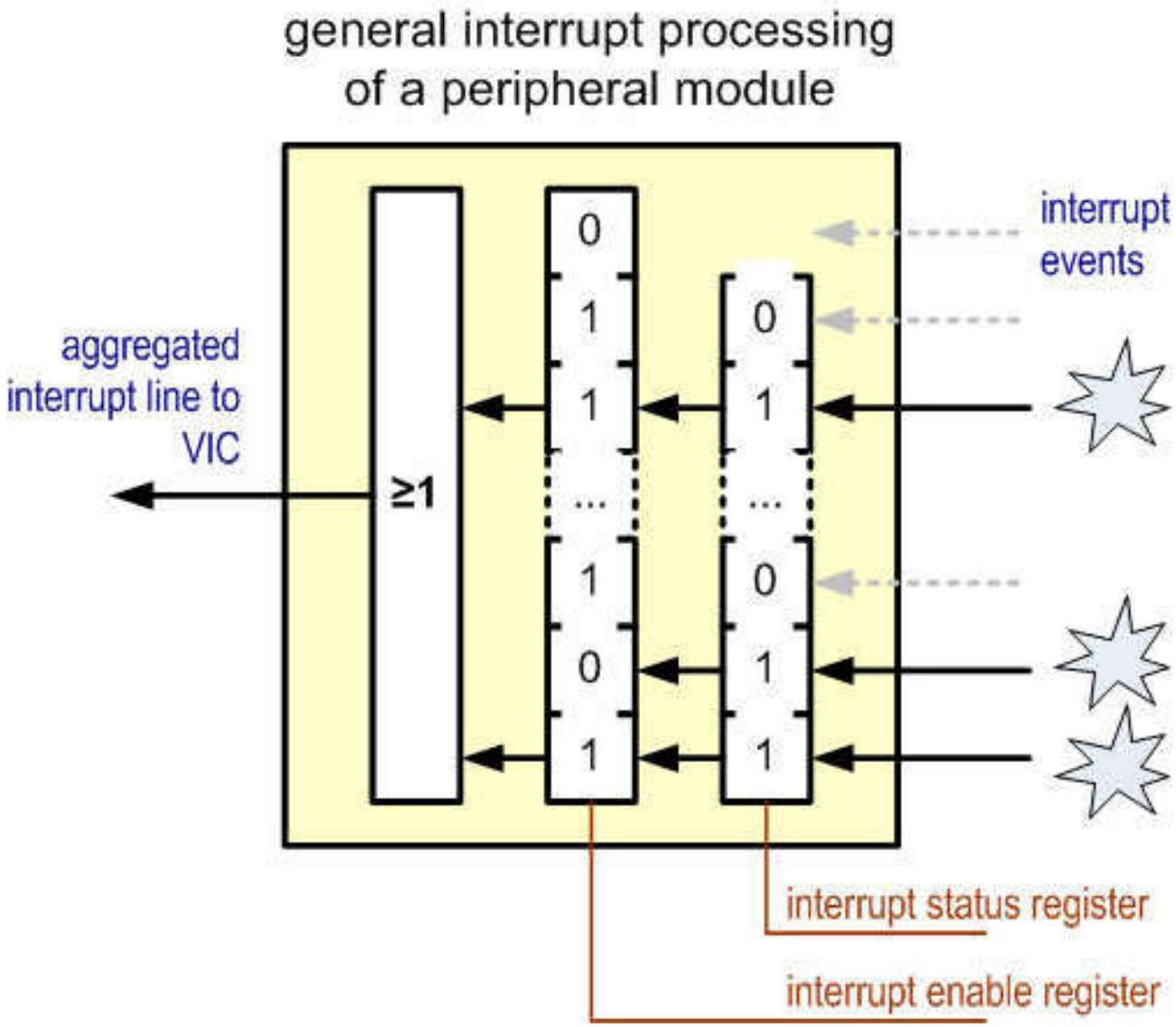
# Overall General Interrupt Architecture



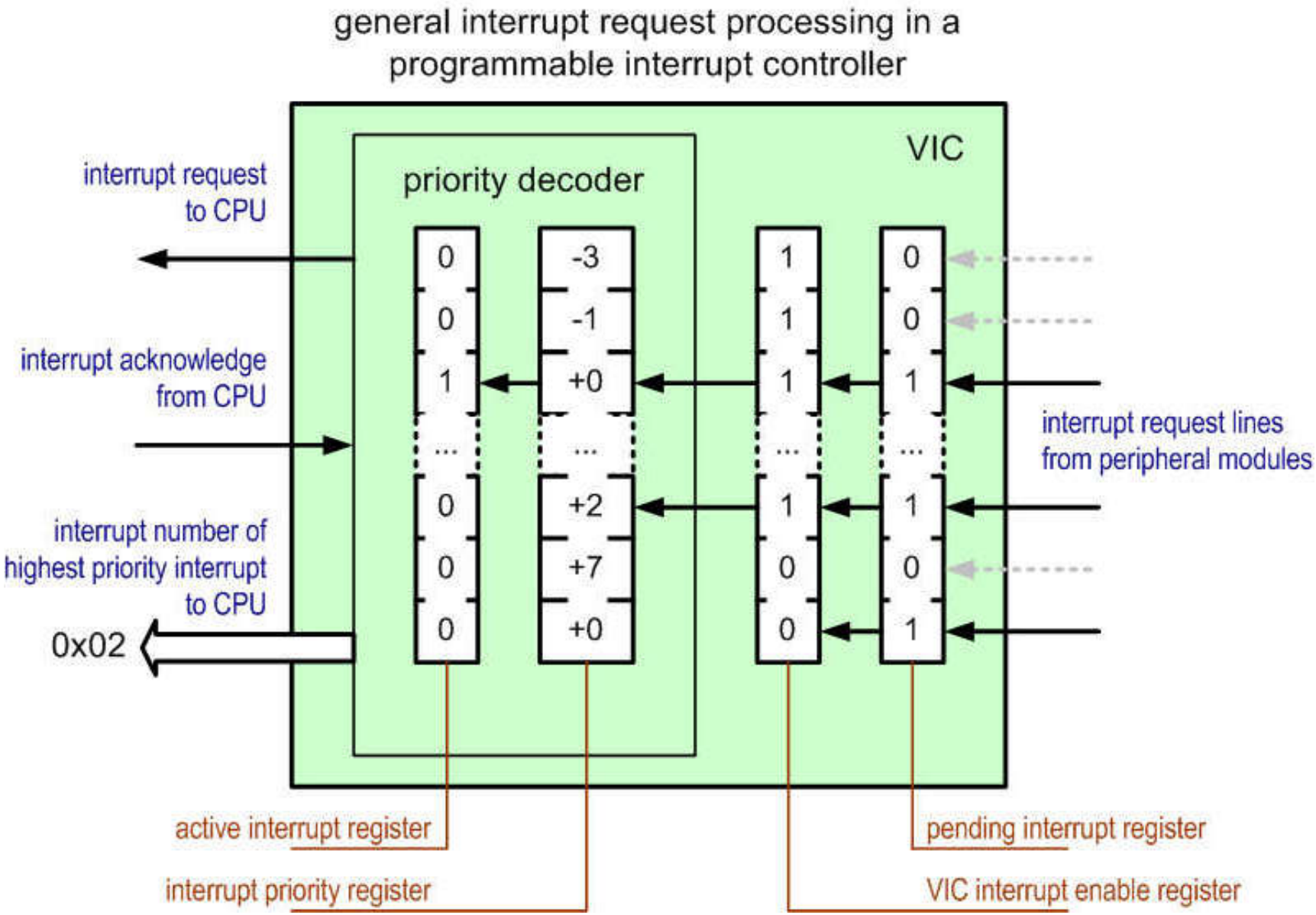
# Interrupt processing in peripheral modules



# Interrupt processing in peripheral modules (cont'd)

- **every peripheral module typically has several interrupt events**
  examples: every pin of a GPIO port can generate interrupts, a timer has several interrupt conditions (compare matches, overflow)
- **every interrupt source needs to be enabled** (interrupt enable register *or* interrupt mask register)
- **occurrence of an interrupt event is signaled by a flag** in the interrupt status register
- **flag** in the interrupt status register **must be cleared by SW** after processing the interrupt handler otherwise reentry to interrupt handler even if no interrupt event occurred in the meantime
- **all interrupts of a peripheral module are aggregated** to a single interrupt request line to the CPU
- to **identify the exact interrupt event** the interrupt handler must check the interrupt status register of the peripheral module

# Programmable Interrupt Controller*

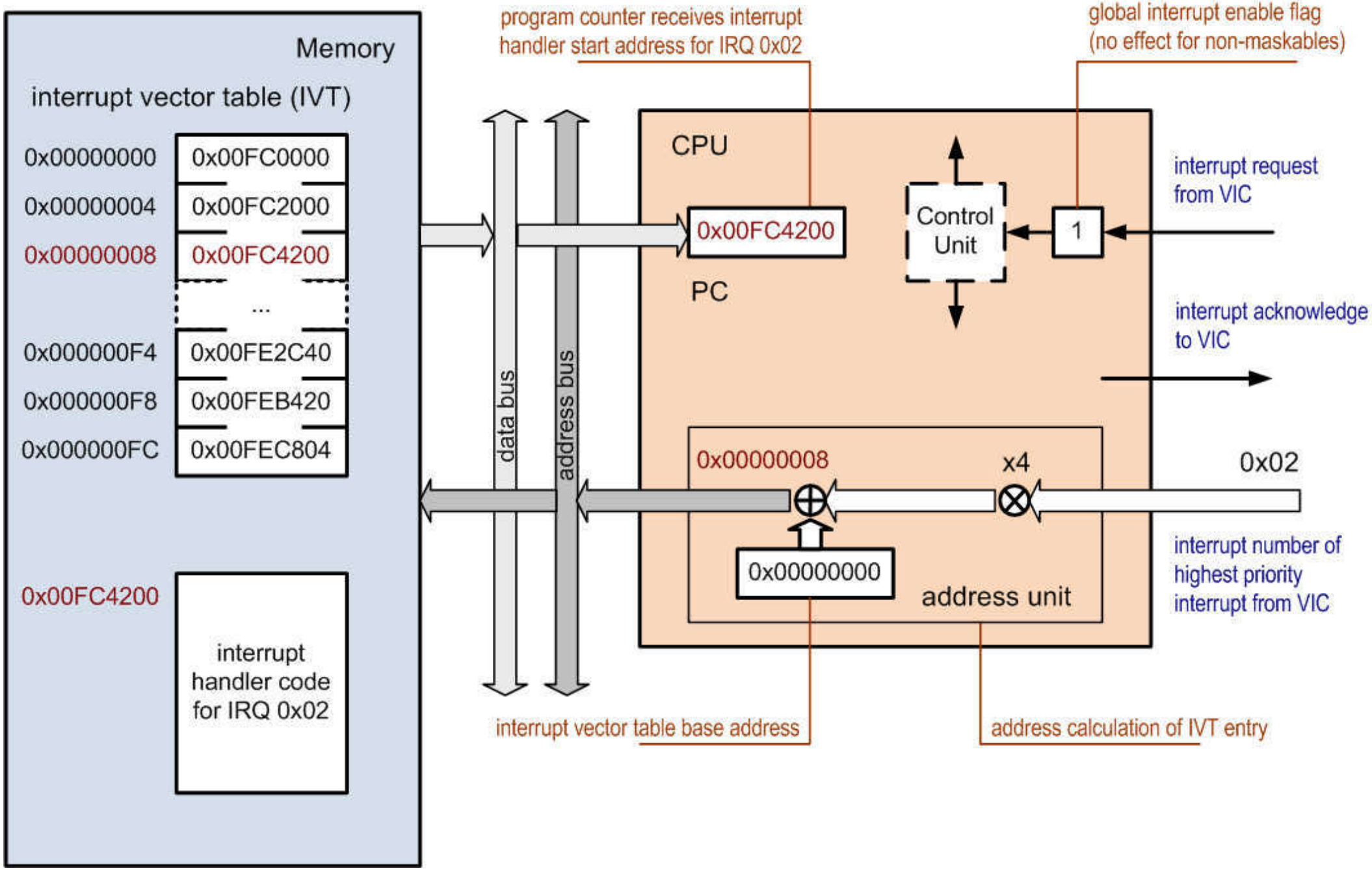



*also named “Vectored Interrupt Controller” VIC (interrupt lines from peripheral modules are bundled to a “vector”)

# Programmable Interrupt Controller(cont'd)

- **interrupt request lines** of every peripheral module are **routed to the VIC**
- the ***pending interrupts register*** shows which interrupt request from peripheral modules wait for being serviced
- if enabled in the ***VIC interrupt enable register***, the request is forwarded to the priority decoder
- every interrupt is assigned a **programmable priority** by the *interrupt priority register*, the lower the value the higher the priority (negative priority values often indicate non-maskable interrupts like reset or on hardware faults)
- the interrupt request with the **highest priority is forwarded to the CPU**, the CPU is informed about the interrupt source by a **system wide unique *interrupt number***
- when the CPU acknowledges an interrupt request, the VIC changes the status of the highest prioritized interrupt **from *pending* to *active***

# CPU Interrupt Handling



# CPU Interrupt Handling (cont'd)

- if the CPU accepts interrupts (*global interrupt enable bit* set) it will **finish (or sometimes abort) the current instruction and grant the interrupt request**
- the **Interrupt Vector Table (IVT)** in memory holds the start addresses of the interrupt handlers
- the CPU address unit calculates the **address of the corresponding IVT entry**:

  IVTEntryAddr = InterruptNum x 4 + IVTBaseAddress
- in parallel, typically the **relevant CPU registers and the current program counter (PC) value are pushed to the stack**
- the **start address of the interrupt handler** is read from IVTEntryAddr and loaded to the PC
- CPU loads and executes the first instruction of the interrupt handler
- **on completion of the interrupt handler**, the CPU registers and the former PC value are restored