System I/O
& Exceptions

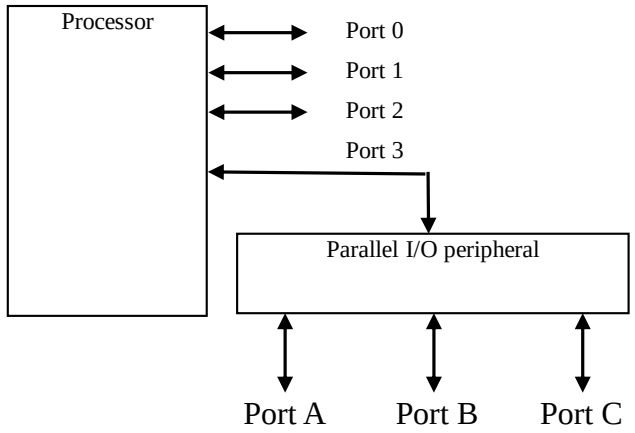
Arquitectura de Computadoras 2021

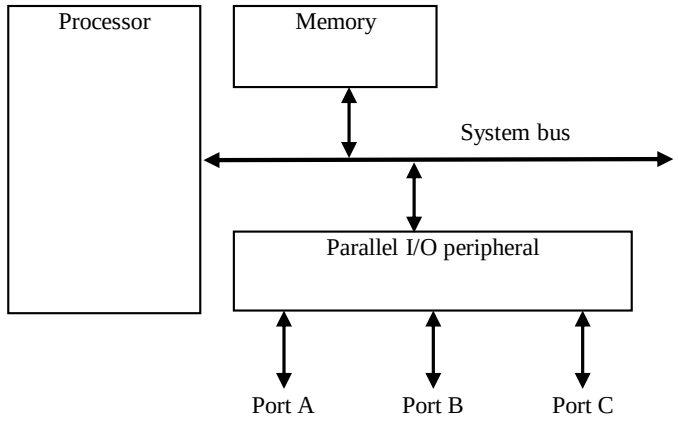
Dr. Ing. Agustin M. Laprovitta (alaprovitta@ucc.edu.ar)

Microprocessor interfacing: I/O addressing

A microprocessor communicates with other devices using some of its pins

- Port-based I/O (*Old Architectures!*)
  - Processor has one or more N-bit ports
  - Processor's software reads and writes a port just like a register
- **Bus-based I/O** (***Nowadays!***)
  - Processor has address, data and control ports that form a system bus
  - Communication protocol is built into the processor
  - A single instruction carries out the read or write protocol on the bus

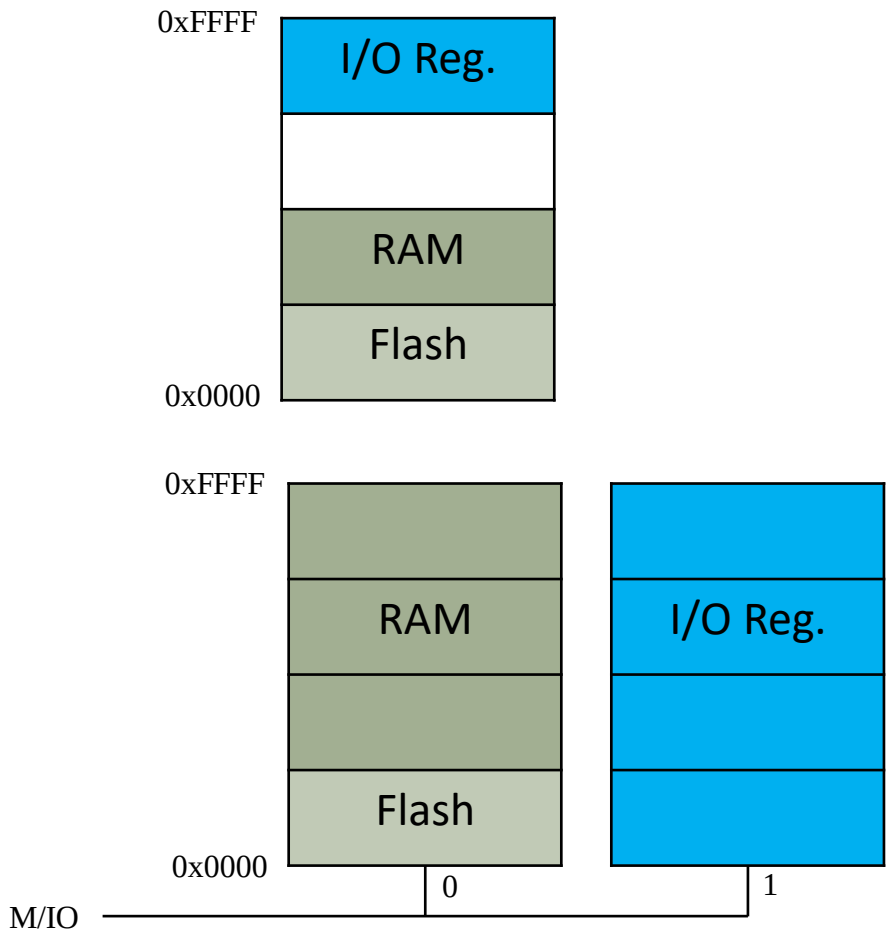




Types of bus-based I/O: **memory-mapped I/O** and **standard I/O**

Processor talks to both memory and peripherals using same bus – two ways to talk to peripherals

- Memory-mapped I/O
  - Peripheral registers occupy addresses in same address space as memory
  - e.g., Bus has 16-bit address
    - lower 32K addresses may correspond to memory
    - upper 32k addresses may correspond to peripherals
- Standard I/O (I/O-mapped I/O)
  - Additional pin (*M/IO*) on bus indicates whether a memory or peripheral access
  - e.g., Bus has 16-bit address
    - all 64K addresses correspond to memory when *M/IO* set to 0
    - all 64K addresses correspond to peripherals when *M/IO* set to 1



Memory-mapped I/O vs. Standard I/O

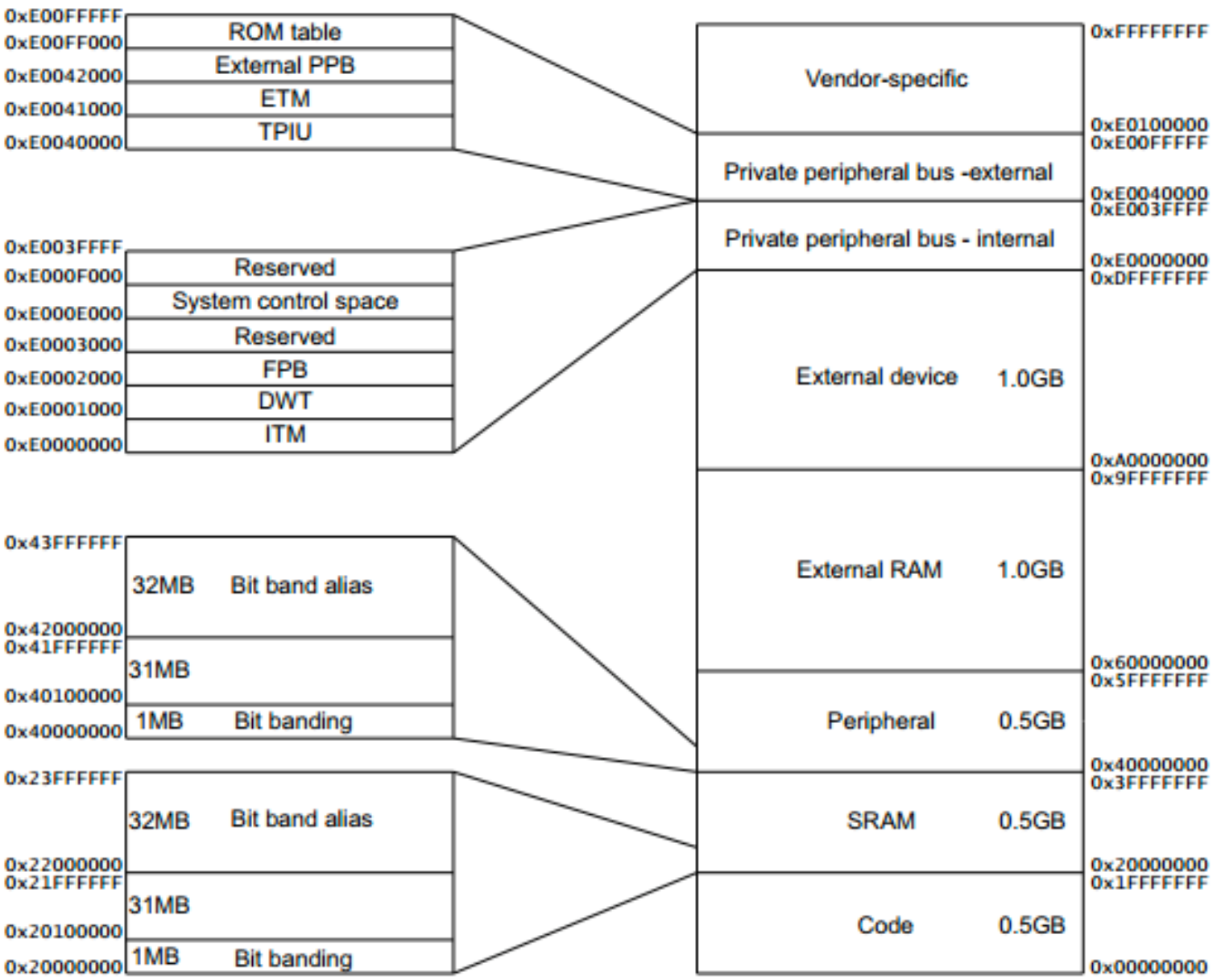
Memory-mapped I/O

- Requires no special instructions
  - Assembly instructions involving memory like MOV and ADD work with peripherals as well
  - Standard I/O requires special instructions (e.g., IN, OUT) to move data between peripheral registers and memory

Standard I/O

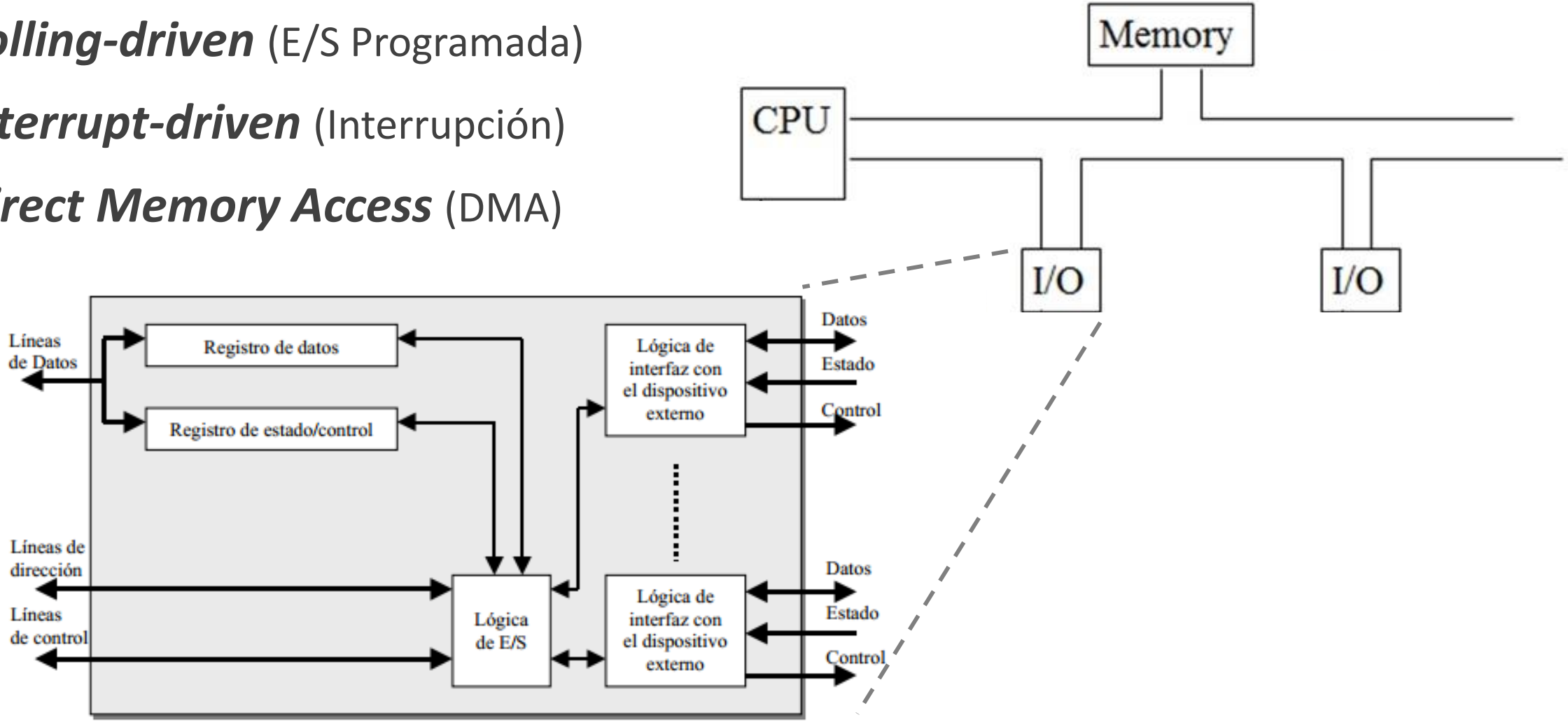
- No loss of memory addresses to peripherals
- Simpler address decoding logic in peripherals possible
  - When number of peripherals much smaller than address space then high-order address bits can be ignored
    - smaller and/or faster comparators

Example: ARM Cortex A9 Memory map



I/O Operation Methods

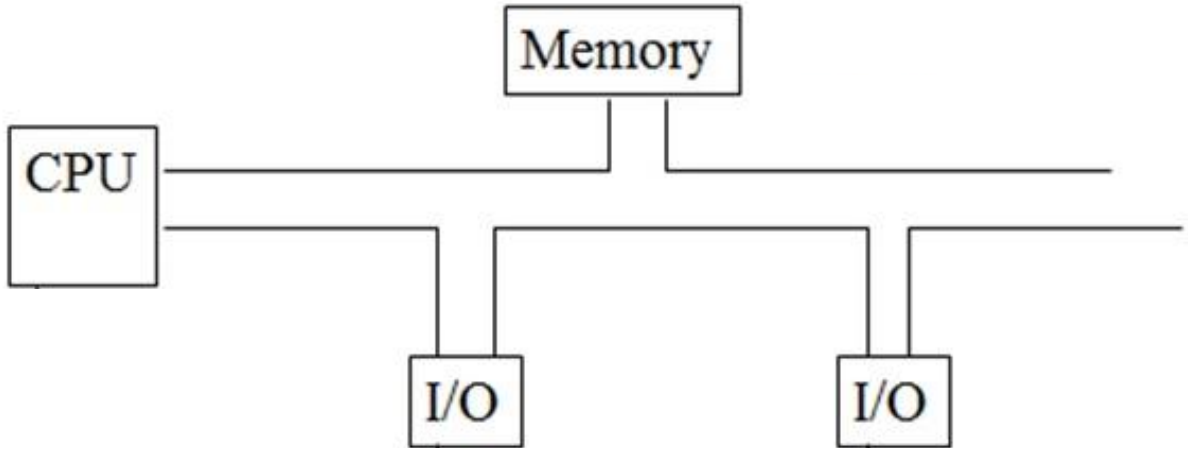
- ***Polling-driven*** (E/S Programada)
- ***Interrupt-driven*** (Interrupción)
- ***Direct Memory Access*** (DMA)

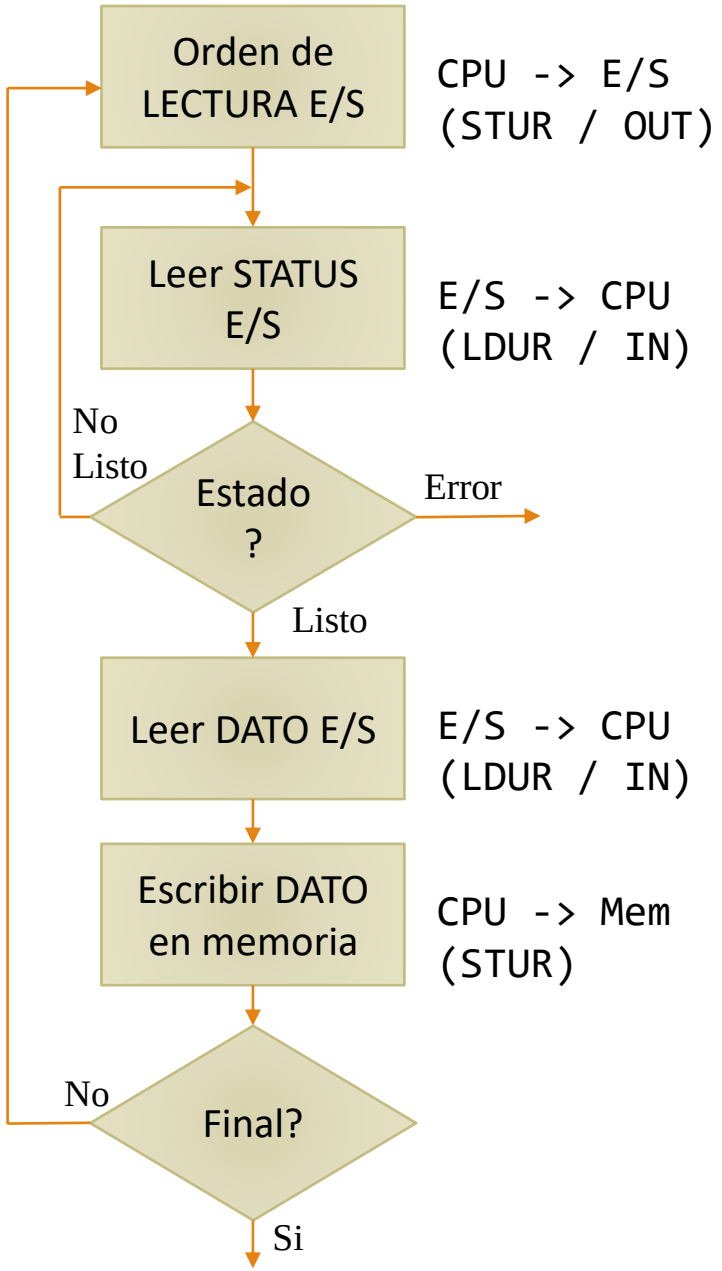


Fuente: W. Stallings, *Organización y Arquitectura de Computadores*, 5° Ed., Prentice Hall, Madrid, 2000

E/S Programada

- El manejo se realiza mediante el uso de instrucciones de E/S por código de programa
- **Principal desventaja**: se desperdician muchos ciclos de instrucción en revisar el estado del módulo E/S (*polling*)

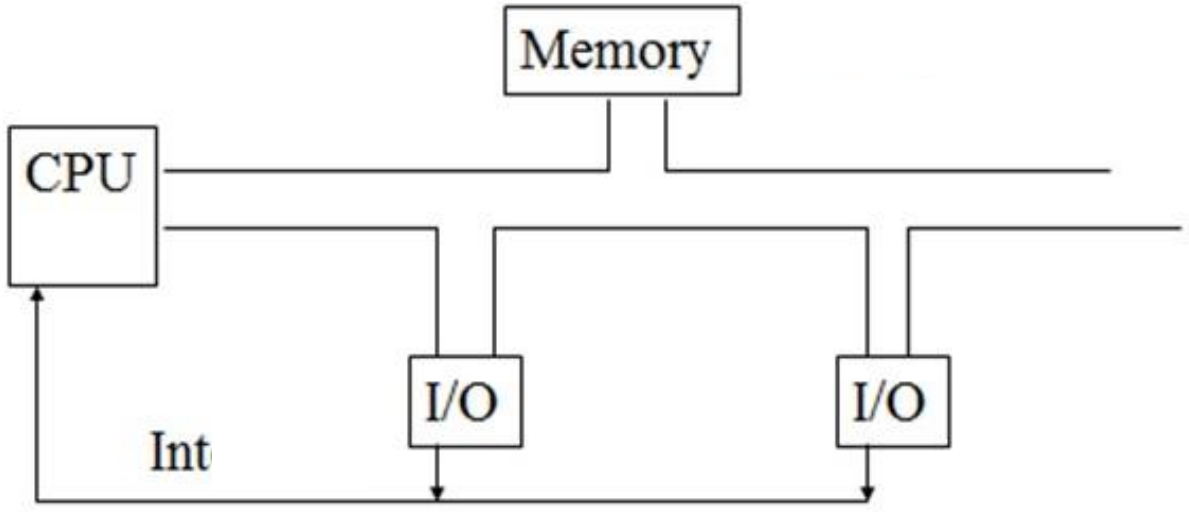


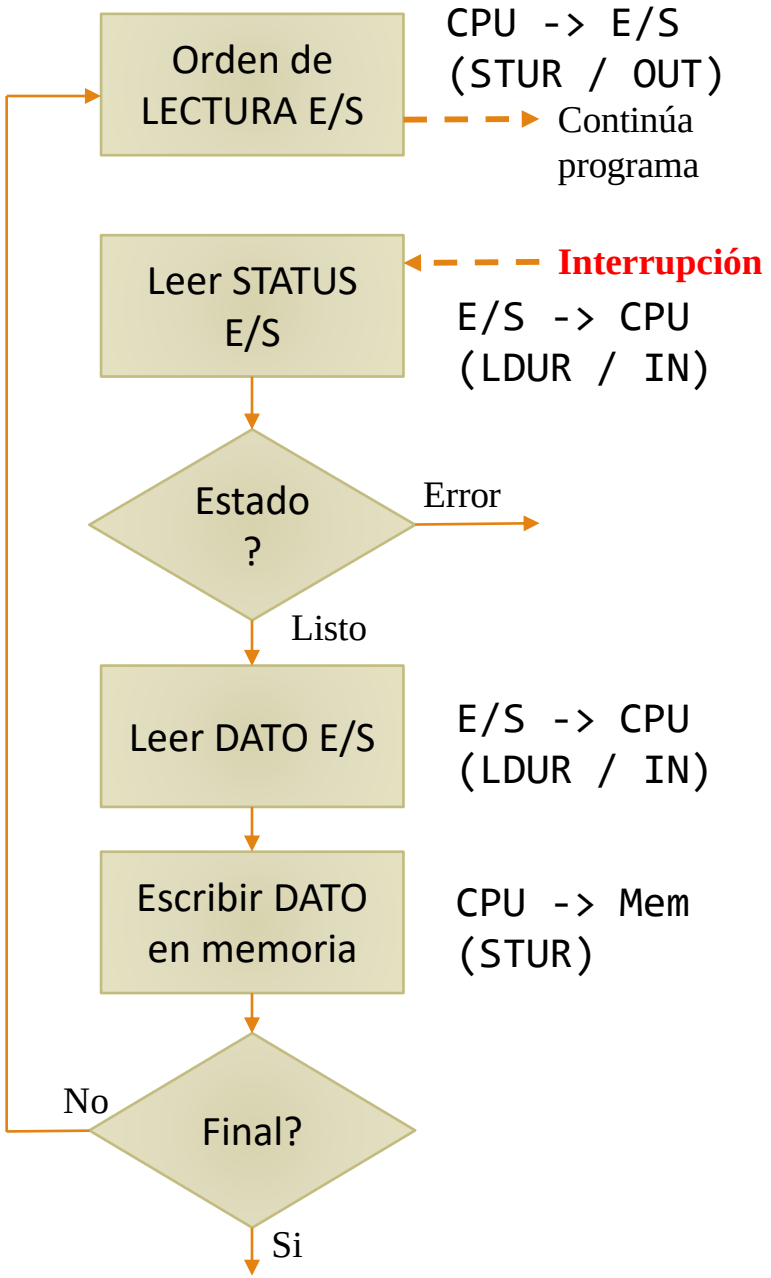


Fuente: W. Stallings, *Organización y Arquitectura de Computadores*, 5° Ed., Prentice Hall, Madrid, 2000

Interrupción

- Es un recurso de HW propio de la CPU: señal de ***Int*** externa para periféricos
- Es literalmente una “interrupción” o quiebre de la ejecución normal del código de programa





Fuente: W. Stallings, *Organización y Arquitectura de Computadores*, 5° Ed., Prentice Hall, Madrid, 2000

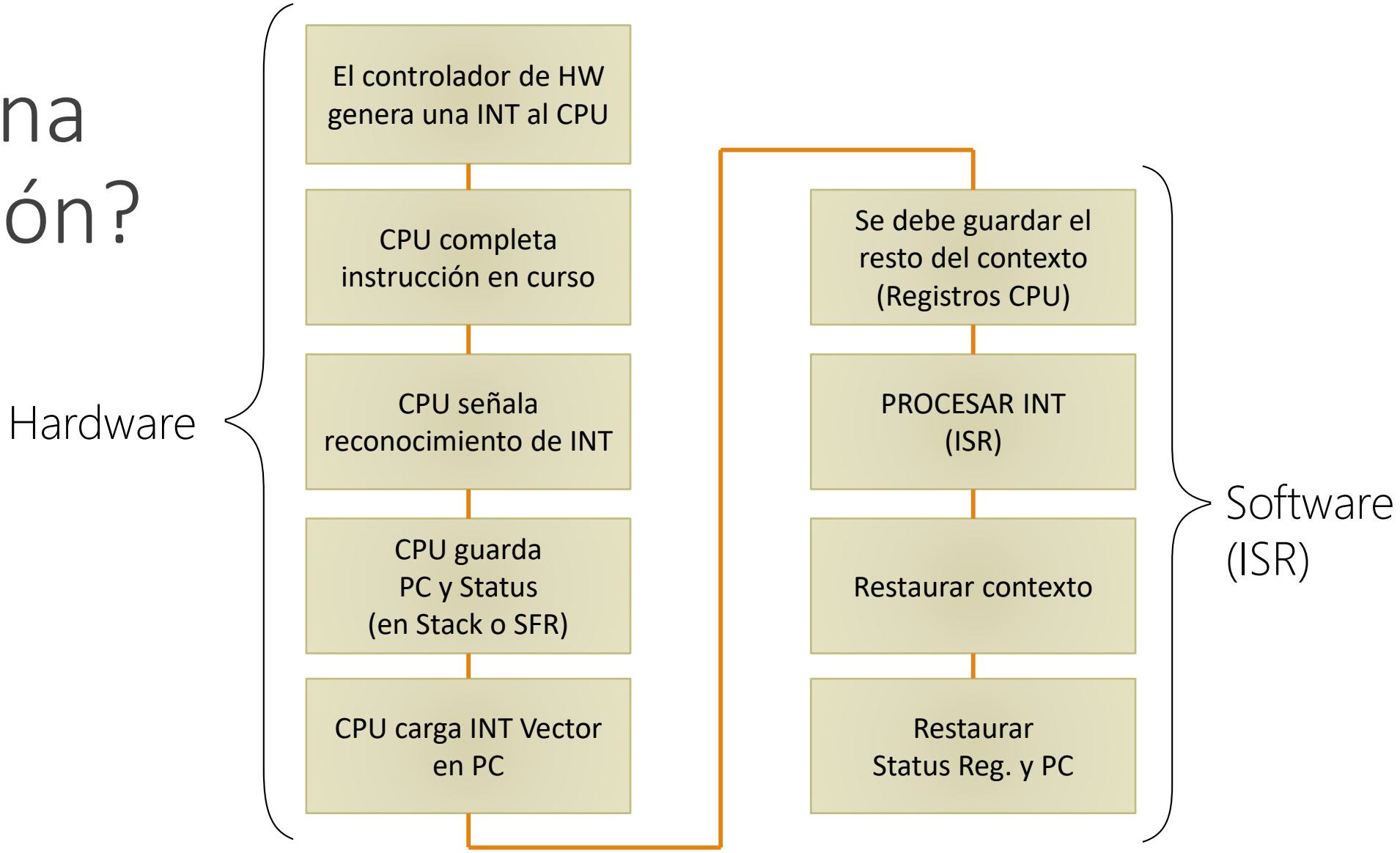
Interrupción (cont.)

- Al finalizar cada ciclo de instrucción el CPU verifica automáticamente si hay ***Int*** pendientes.
- De esta forma el "polling" lo realiza la CPU por HW: **no consume ciclos de instrucción!!**
- Si hay ***Int***, el CPU salta automáticamente a una posición de memoria especifica llamada **vector de interrupciones**.
- El vector de interrupciones contiene el código (o su referencia) con los procedimientos necesarios para dar servicio a dicha interrupción. Este código se denomina ISR (*Interrupt Service Routine*)

Donde se aloja la ISR? (*interrupt address vector*)

- Dirección fija (*Fixed interrupt*)
  - La dirección esta establecida en la lógica de la CPU, no puede ser modificada
  - La CPU puede contener la dirección real, o contener una instrucción de salto a la dirección real de la ISR si no hay suficiente espacio reservado.
- Dirección vectorizada (*Vectored interrupt*)
  - El periférico provee la dirección al CPU por medio del bus de datos
  - Para esto, se agrega una señal mas: INT ACK
  - Muy utilizado en sistemas con múltiples periféricos conectados por un bus
- Solución de compromiso:
  - Tabla de direcciones (*interrupt address table*)

Como se procesa una Interrupción?



Fuente: W. Stallings, *Organización y Arquitectura de Computadores*, 5° Ed., Prentice Hall, Madrid, 2000

Tabla de dirección de interrupciones

Solución de compromiso entre la interrupción fija y vectorizada

- Un solo par de señales *IntReq* / *IntAck* son necesarias
- Se dispone en memoria de una tabla conteniendo las direcciones de las ISR (Ejemplo: 256 vectores)
- Los periféricos ya no proveen la dirección de la ISR, sino un **índice** de la tabla.
- **Ventajas:**
  - Menos bits son enviados desde el periférico (usualmente data bus < address bus)
  - Se puede mover la posición de las IRS sin cambiar datos de configuración en el periférico.

Consideraciones adicionales

Interrupciones Enmascarables vs. no-enmascarables

- Enmascarables: El programador puede modificar un bit que causa que el procesador ignore una solicitud de interrupción (GEI)
  - Muy importante para usar cuando se tienen porciones de códigos temporalmente criticas.
- No-enmascarable: una señal separada que no puede ser ignorada (internas a la CPU)
  - Típicamente reservada para situaciones drásticas, como la ocurrencia de un fallo de alimentación, un acceso indebido a memoria flash o la detección de un código de operación invalido.
  - Estas ISR se las suele conocer con el nombre de “Traps”

Salto a una ISR

- Algunos procesadores tratan el salto de una INT como una llamada a cualquier otra rutina
  - Salvan el entorno completo (PC, status, registers) – toma muchos ciclos
- Otros salvan parcialmente su entorno (solo PC y status)
  - El programador debe asegurarse que el ISR no altere los registros o debe salvarlos previamente
  - Es necesario instrucciones de retornos diferentes en su set de instrucciones (RET – RETI)

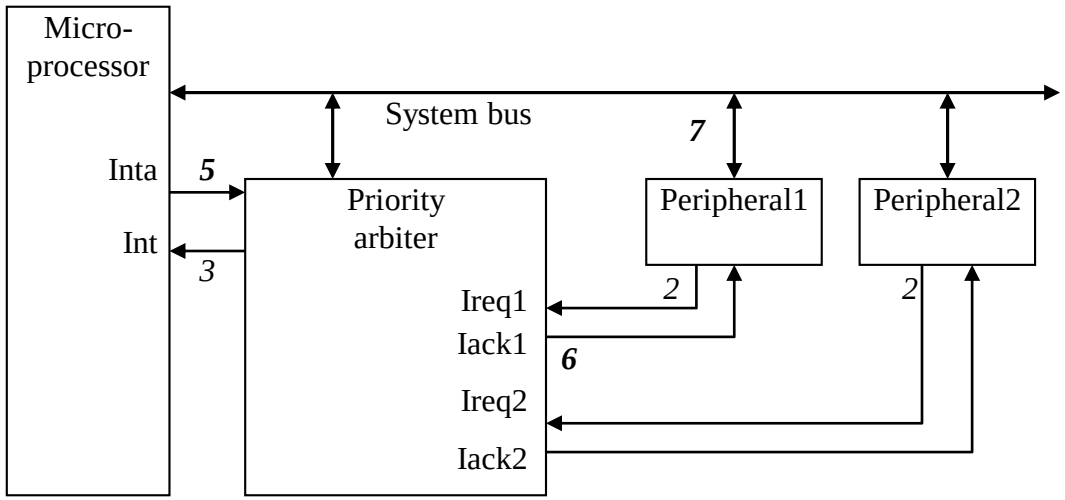
Múltiples periféricos: **Arbitraje**

Considere la situación donde muchos periféricos solicitan el servicio de una CPU simultáneamente (microcontrolador) – **Cual será atendida primero y en que orden?**

- Software polling
  - Muy simple implementación por HW (una sola línea de INT)
  - El programador debe determinar en la ISR el origen de la INT buscando banderas INT FLG
  - La prioridad es establecida en la ISR según el orden de la búsqueda
- Arbitro de prioridades (*Priority arbiter*)
- Conexión en cadena (*Daisy chain*)
- Arbitraje de bus (*Network-oriented*)
  - Utilizado en arquitecturas de múltiples procesadores
  - El periférico debe primero obtener la sesión del bus para luego requerir una interrupción

Arbitraje: **Arbitro de prioridades**

- También conocido como **controlador de interrupciones**
- Muy usado en arquitecturas de una CPU de propósito general
- El periféricos hace la petición INT REQ al controlador, y este a la CPU. En orden inverso para los INT ACK
- Esquema de direccionamiento vectorizado
- Múltiples esquemas de prioridad:
  - Fija
  - Round-robin
  - FIFO
- Conexión al bus del sistema solo con fines de configuración



1. CPU ejecuta su programa principal
2. P1 solicita servicio seteando *Ireq1*. P2 también solicita servicio mediante *Ireq2*.
3. Como el controlador detecta al menos una *Ireq* , entonces setea *Int*.
4. La CPU para la ejecución del programa y guarda el contexto.
5. La CPU setea *Inta*.
6. El controlador setea *Iack1* para responder a P1.
7. P1 pone la dirección del vector en el bus de datos
8. La CPU salta a la dirección del ISR leída del bus, la ISR se ejecuta y retorna.
9. La CPU resume la ejecución del programa principal

Fuente: F. Vahid, T. Givargis, *Embedded System Desig: A Unified Hardware/Software Introduction*, John Wiley & Son., New York, 2002

Arbitraje: **Conexión en cadena**

La lógica de control de arbitraje esta embebida en cada periférico

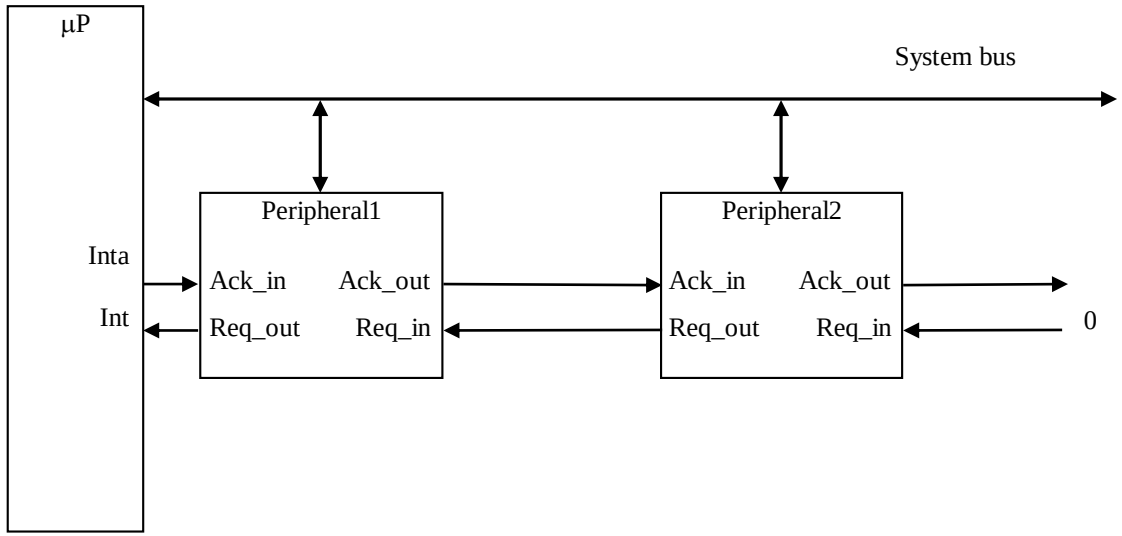
- Se agrega una señal de entrada *IntReq* y de salida *IntAck*

Los periféricos están dispuestos en el sistema en forma de cadena

- Un solo periférico se conecta al CPU (el de mayor prioridad), los otros “aguas arriba”
- Los periféricos difunden la señal ***IntReq*** hacia el CPU y la señal ***IntAck*** hacia el origen de la interrupción

Pros vs. contras:

- Ideal si es necesario agregar o sacar periféricos
- Bajo rendimiento con muchos periféricos
- Es necesario agregar lógica por seguridad (periférico fuera de servicio)
- Esquema de prioridad único

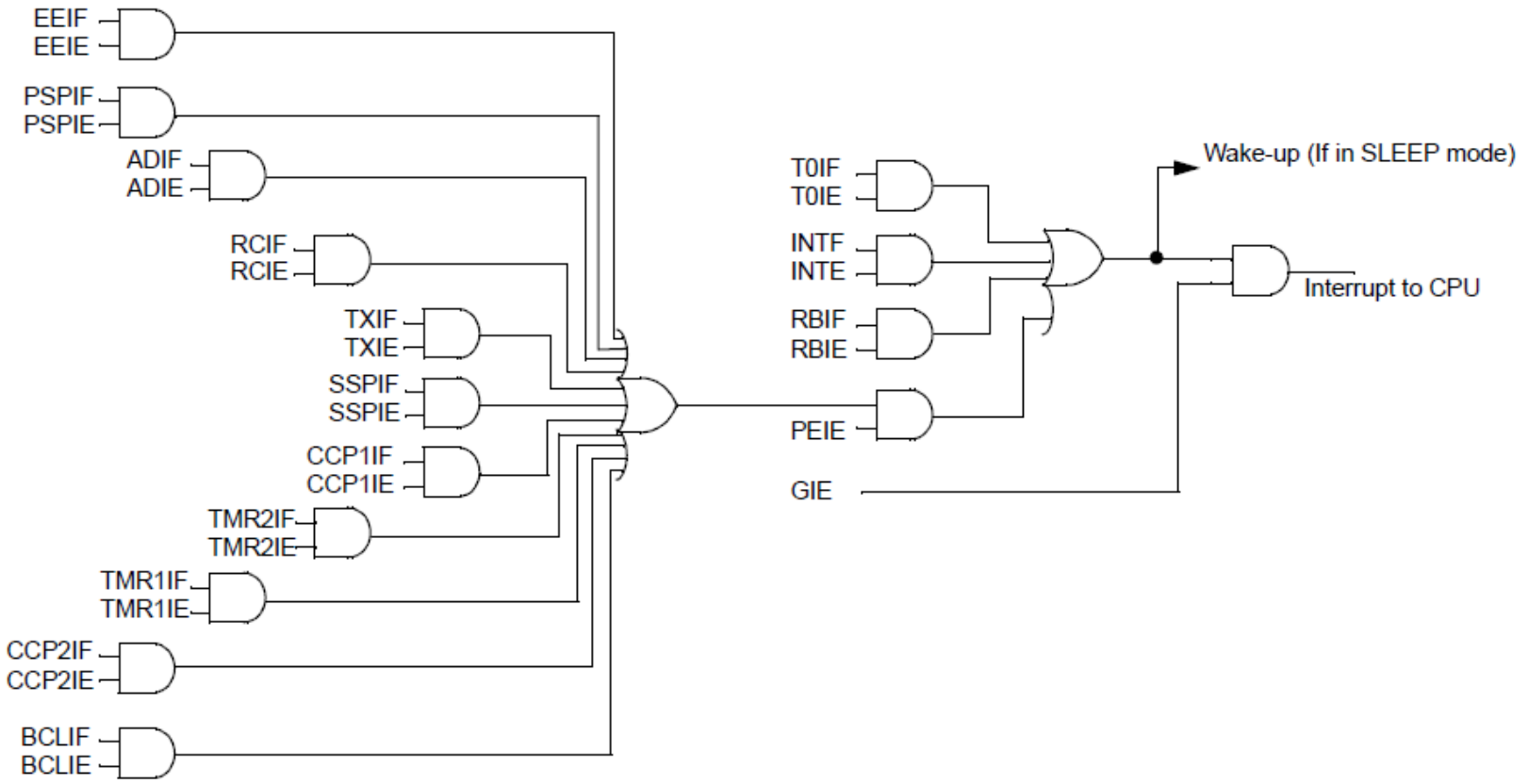


Fuente: F. Vahid, T. Givargis, *Embedded System Desig: A Unified Hardware/Software Introduction*, John Wiley & Son., New York, 2002

Caso de estudio #1: Familia Microchip PIC16x

Esquema de vector único

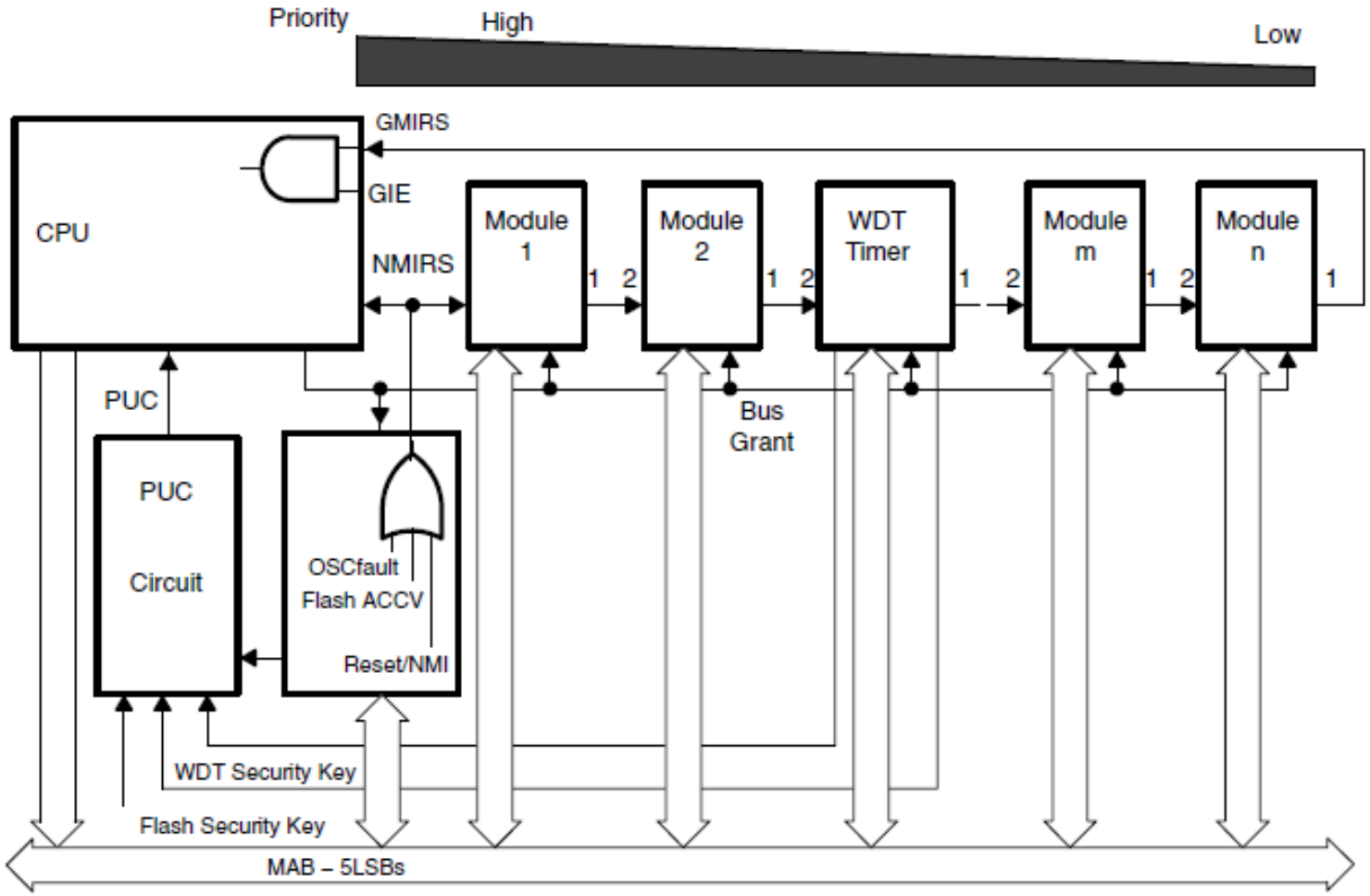
- Cada periférico posee un habilitador (IE) y una flag (IF)
- Esquema software polling
- La prioridad la define el programador en la ISR
- Habilitador general de int (GIE)
- Esquema simple de rendimiento moderado



Caso de estudio #2: Familia MSP430x2xx

Esquema en cadena vectorizado

- Cada periférico posee un habilitador (IE) y una flag (IF)
- Vectores con servicios compartidos
- Esquema de prioridad fija
- Interrupciones enmascarables (periféricos) y no-enmascarables (sistema)
- Habilitador general de int (GIE)
- El guardado del entorno y el retorno de una interrupción (RETI) toman 5 ciclos de reloj cada uno.



Caso #3: Familia ARM Cortex M

Controlador vectorizado de interrupciones anidadas (NVIC, *Nested vectored interrupt controller*)

• Hasta 81 entradas de INT de periféricos (IRQs)

ANIDADAS!!! Una IRQ puede interrumpir a otra de menor prioridad.

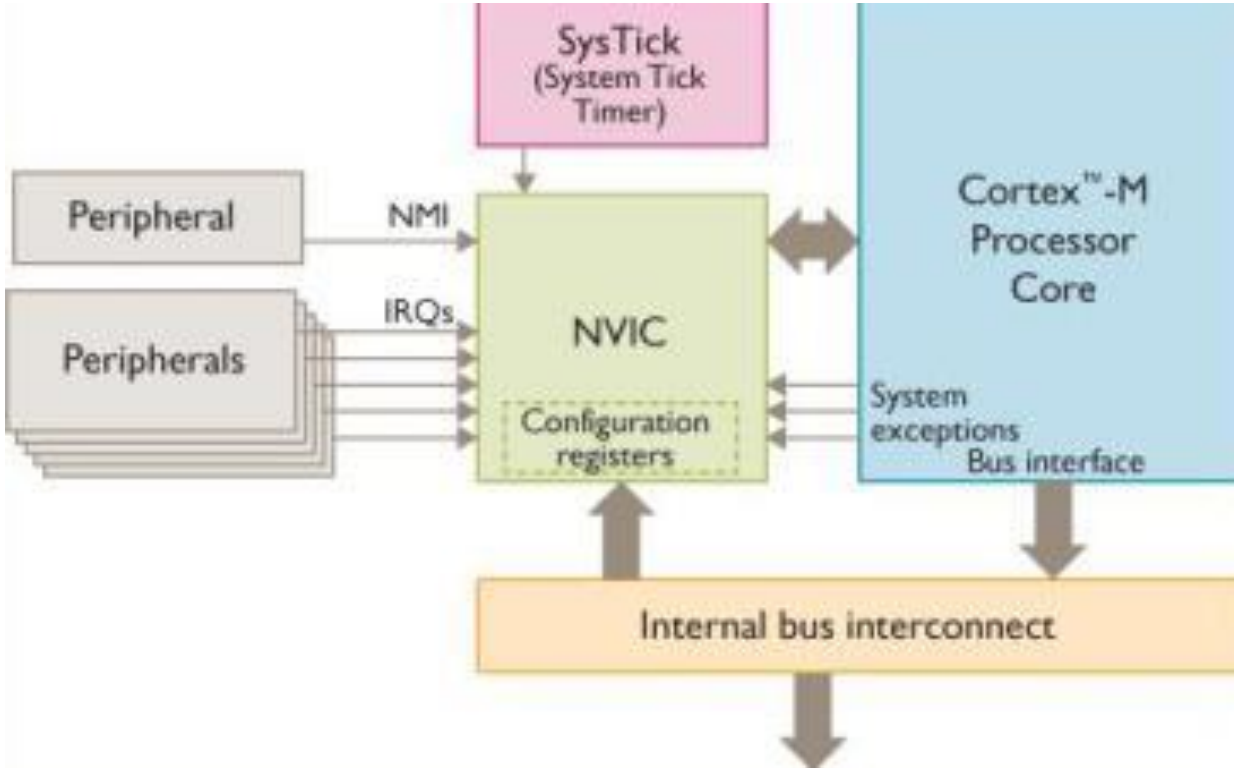
• Nivel de prioridad programable 0-15 para cada IRQ. Los niveles altos corresponden a prioridades bajas.

• Detección por nivel o flanco de las señales de INT

• Reprogramación de las prioridades en forma dinámica

• A cada periférico se le asigna un nivel de prioridad y sub-prioridad

• Interrupción externa No-enmascarable (NMI)



Exceptions and Interrupts

"Unexpected" events requiring change in flow of control

- Different ISAs use the terms differently

Exception

- Arises within the CPU
  - e.g., undefined opcode, FPU overflow, syscall, …

Interrupt

- From an external I/O controller

Dealing with them without sacrificing performance is hard!

Handling Exceptions

- Save PC of offending (or interrupted) instruction
  - In LEGv8: Exception Link Register (ELR)

- Save indication of the problem
  - In LEGv8: Exception Syndrome Rregister (ESR)
  - We'll assume 1-bit
    - 0 for undefined opcode, 1 for overflow

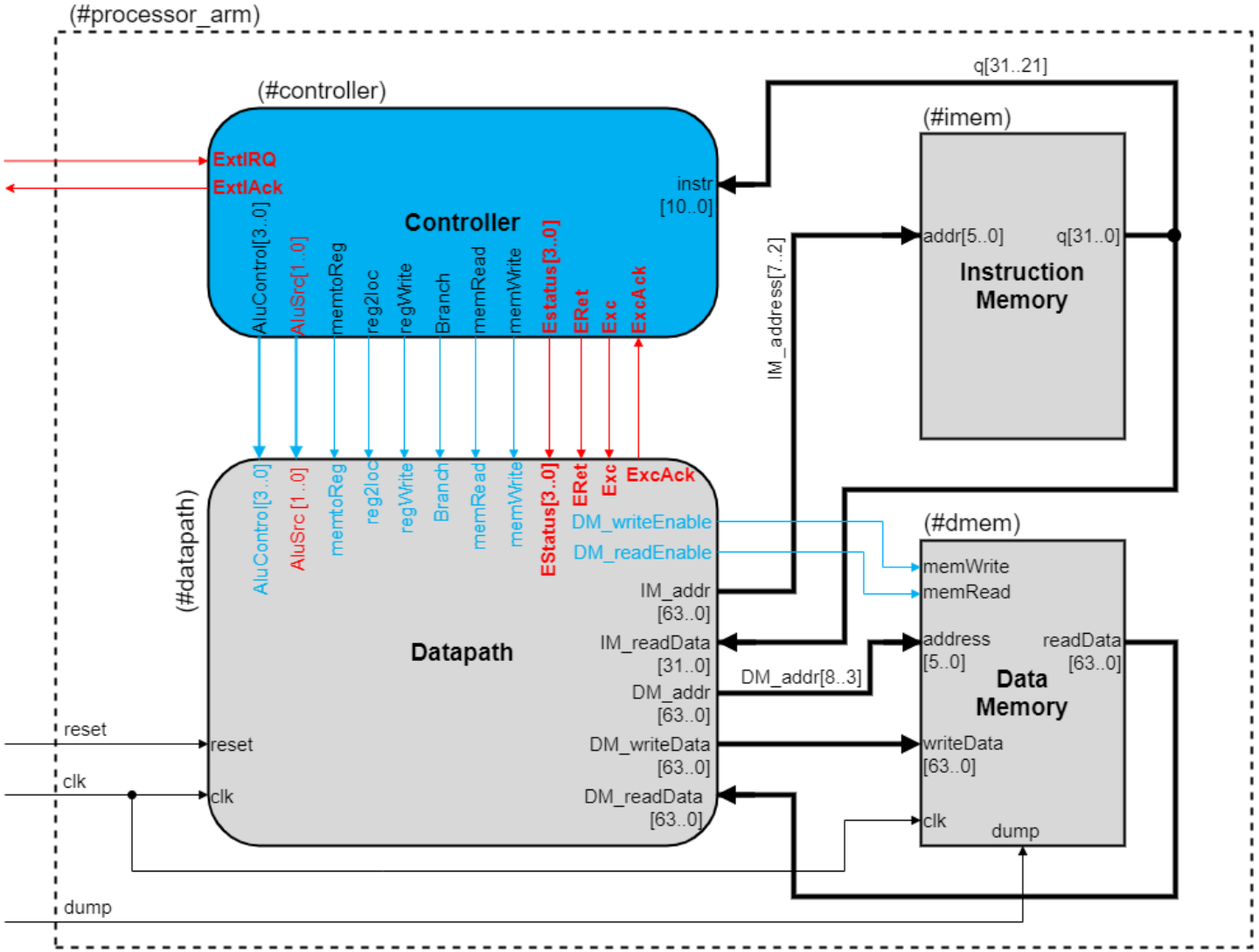
An Alternate Mechanism

- Vectored Interrupts
  - Handler address determined by the cause
- Exception vector address to be added to a vector table base register:
  - Unknown Reason: $00\ 0000_{two}$
  - Overflow: $10\ 1100_{two}$
  - …: $11\ 1111_{two}$
- Instructions either
  - Deal with the interrupt, or
  - Jump to real handler

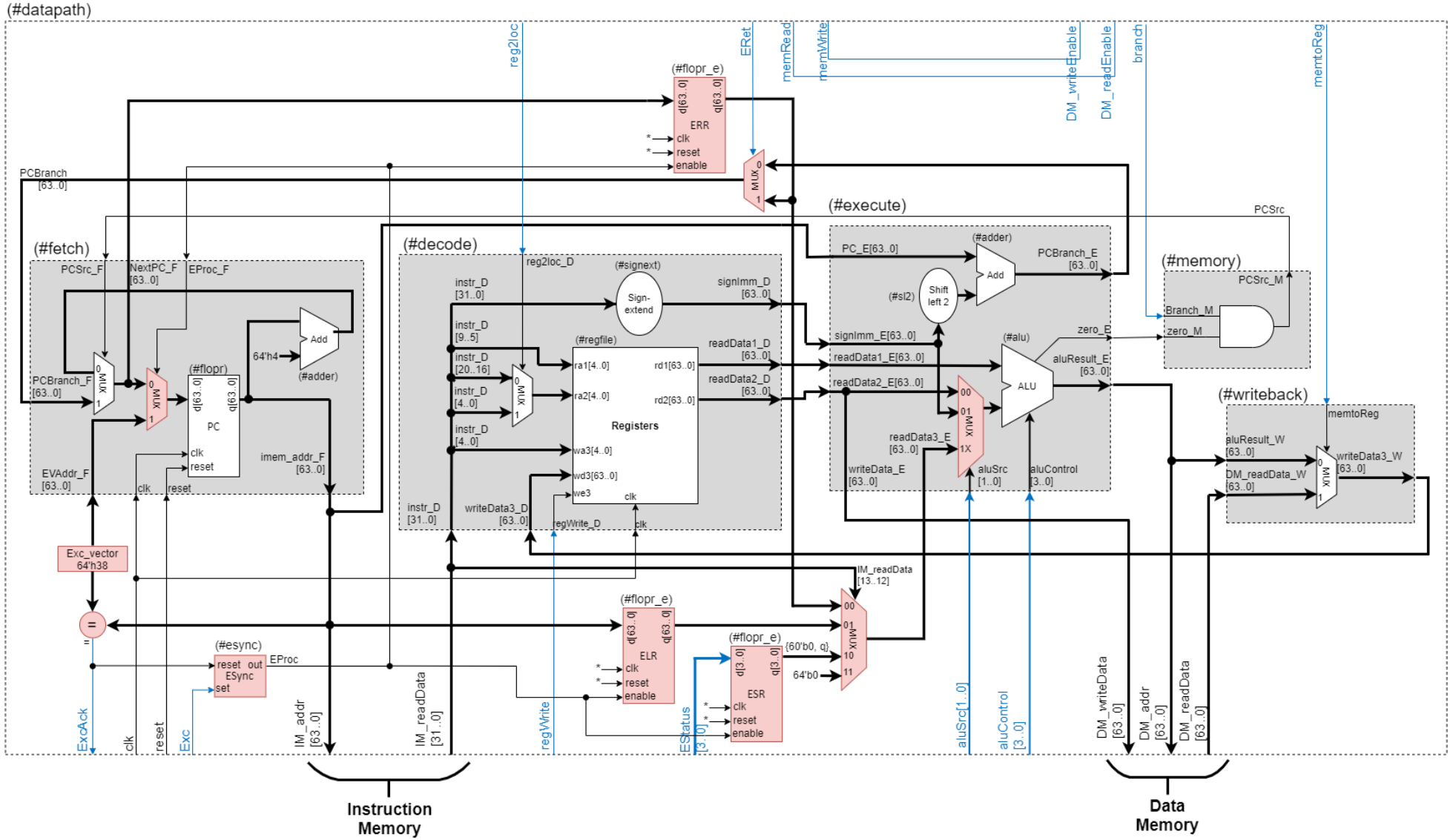
Handler Actions

- Read cause, and transfer to relevant handler
- Determine action required
- If restartable
  - Take corrective action
  - use EPC to return to program
- Otherwise
  - Terminate program
  - Report error using EPC, cause, …

Processor with Exceptions (a LEGv8 approach)



Datapath with Exceptions (just an approach!)



Exceptions – The ISA Approach
New Instructions!

- **ERET** (Exception Return)
  - Type: R - OpCode: 1101011(0100)
  - Syntaxis: ERET

31 25	24 21	20 16	15 10	9 5	4 0
	opc	Rm	shamt	Rn	Rd (Rt)
1 1 0 1 0 1 1	0 1 0 0	1 1 1 1 1	0 0 0 0 0 0	1 1 1 1 1	0 0 0 0 0

- **MRS** (Move (from)SystemReg to GeneralPurposeReg)
  - Syntaxis: `MRS <Rt>, <systemReg>`
  - Type: S (*new!*) - OpCode: 1101010100(1)
  - <systemReg> = "S<2+op0>_<op1>_<CRn>_<CRm>_<op2>"
    - S2_0_C0_C0_0 → ERR
    - S2_0_C1_C0_0 → ELR
    - S2_0_C2_C0_0 → ESR
    - S2_0_C3_C0_0 → Reserved

31 21	20 19	18 16	15 12	11 8	7 5	4 0
	op0	op1	CRn	CRm	op2	Rt
1 1 0 1 0 1 0 1 0 0 1	1 X	X X X	X X X X	X X X X	X X X	X X X X X

Exceptions – The ISA Approach
More useful Instructions…

- **BR** (Branch with Register)

  BR Xt

- **SVC** (Generate exception with 16-bit payload)

  SVC #uimm16

- **MSR** (Move (from)GeneralPurposeReg to SystemReg)

  MSR <system_register>, Xt

Preguntas?
Dudas? Consultas?