

# Interrupts

Stopping program flow to execute a special piece of code that handles an event

- Definition and classification
- PowerPC interrupt structure
- Precise exceptions
- Handling multiple interrupts
  - Nesting, prioritization

Reference: Chapter 6, Programming Environment

# I/O Data Transfer

Two key questions to determine how data is transferred to/from a non-trivial I/O device:

1. How does the CPU know when data is available?

(a) Polling                      (b) Interrupts

2. How is data transferred into and out of the device?

(a) Programmed I/O              (b) Direct Memory Access (DMA)

# Interrupts

Interrupt (a.k.a. exception or trap): An event that causes the CPU to stop executing the current program and begin executing a special piece of code called an **interrupt handler** or **interrupt service routine** (ISR). Typically, the ISR does some work and then resumes the interrupted program.

Interrupts are really glorified procedure calls, except that they:

- **can occur between any two instructions**
- are transparent to the running program (usually)
- are not explicitly requested by the program (typically)
- call a procedure at an address determined by the type of interrupt, not the program

# Two basic types of interrupts (1/2)

- Those caused by an instruction
  - Examples:
    - TLB miss
    - Illegal/unimplemented instruction
    - div by 0
  - Names:
    - Trap, exception, **synchronous interrupt**

# Two basic types of interrupts

## (2/2)

- Those caused by the external world
  - External device
  - Reset button
  - Timer expires
  - Power failure
  - System error
- Names:
  - interrupt, external interrupt, **asynchronous interrupt**.

# How it works

- Something tells the processor core there is an interrupt
- Core transfers control to code that needs to be executed
- Said code “returns” to old program
- Much harder then it looks.
  - Why?

## ... is in the details

- How do you figure out *where* to branch to?
- How to you insure that you can get back to where you started?
- Don't we have a pipeline? What about partially executed instructions?
- What if we get an interrupt while we are processing our interrupt?
- What if we are in a “critical section?”

# Where

- If you know *what* caused the interrupt then you want to jump to the code that handles that interrupt.
  - If you number the possible interrupt cases, and an interrupt comes in, you can just branch to a location, using that number as an offset (this is a branch table)
  - If you don't have the number, you need to *poll* all possible sources of the interrupt to see who caused it.
    - Then you branch to the right code



# Get back to where you once belonged

- Need to store the return address somewhere.
  - Stack *might* be a scary place.
    - *That* would involve a load/store and might cause an interrupt!
  - So a dedicated register seems like a good choice
    - But that might cause problems later...

# Snazzy architectures

- A modern processor has *many* (often 50+) instructions in-flight at once.
  - What do we do with them?
- Drain the pipeline?
  - What if one of them causes an exception?
- Punt all that work
  - Slows us down
- What if the instruction that caused the exception was executed before some other instruction?
  - What if that other instruction caused an interrupt?

# Nested interrupts

- If we get one interrupt while handling another what to do?
  - Just handle it
    - But what about that dedicated register?
    - What if I'm doing something that can't be stopped?
  - Ignore it
    - But what if it is important?
  - Prioritize
    - Take those interrupts you care about. Ignore the rest
    - Still have dedicated register problems.

# Critical section

- We probably need to ignore some interrupts but take others.
  - Probably should be sure *our* code can't cause an exception.
  - Use same prioritization as before.

# Power PC

- Power PC is a great teaching tool for this because it is so messed-up in how it handles things
  - It supports almost every option you could ever want.
- Names
  - Synchronous interrupts
  - Asynchronous interrupts

# PowerPC Interrupt Structure

## **Machine Status Register (MSR)**

- Defines the state of the processor (Table 6-5): Power management, endian mode, external interrupt enable, machine check enable, privilege level, recoverable exceptions, etc.
- When an exception occurs, MSR bits are altered as determined by the exception.
- Support for interrupts
  - Two special-purpose registers to store program counter and MSR: Save/Restore Registers 0 and 1 (SRR0 and SRR1)
  - One instruction: return from interrupt (rfi)

# PowerPC: Interrupt Process

- Basic interrupt process
  - Stop executing current program (stop fetching new instructions)
  - Save program counter of next instruction in SRR0
  - Save processor mode bits from MSR in SRR1
  - Change some of the processor mode bits in MSR
  - Branch to address determined by type of interrupt
  - The last instruction in an ISR will be an **rfi** which will
  - Restore the processor mode bits from SRR1
  - Branch to the address in SRR0

# MSR

0-12		reserved
13	POW	power management
14		reserved
15	ILE	exception little-endian
<b>16</b>	<b>EE</b>	<b>external interrupt enable</b> (0: delay recognition; 1: enabled)
17	PR	privilege level (0: user and supervisor-level; 1: user only)
18	FP	floating point available
19	ME	machine check enable
20	FE0	floating point exception mode 0
21	SE	single-step trace enable
22	BE	branch trace enable
23	FE1	floating point exception mode 1
24		reserved
<b>25</b>	<b>IP</b>	<b>exception prefix (000 or FFF)</b>
26	IR	instruction address translation
27	DR	data address translation
28-29		reserved
<b>30</b>	<b>RI</b>	<b>recoverable exception</b>
31	LE	little-endian enable (0: big-endian; 1: little-endian)



# Example: External Interrupt

An external interrupt is signaled to the processor through the assertion of the external interrupt signal.

Execution continues at offset 0x00500 from base physical location determined by MSR[IP] bit.

SRR0: Set to the effective address that processor would have attempted to execute next if no interrupt condition were present.

SRR1: 1-4 cleared

10-15 cleared

16-23, 25-27, 30-31 loaded with equivalent bits from MSR

MSR:	POW = 0	FP = 0	BE = 0	DR = 0
	ILE = same	ME = same	FE1 = 0	RI = 0
	EE = 0	FE0 = 0	IP = same	LE = ILE
	PR = 0	SE = 0	IR = 0	

# MPC823 Interrupts (Table 7-1)

Type	Cause
<ul style="list-style-type: none"><li>• System reset</li></ul>	Implementation dependent.
<ul style="list-style-type: none"><li>• Machine check</li></ul>	Bus-parity errors, access invalid physical address.
<ul style="list-style-type: none"><li>• DSI</li></ul>	Cannot perform data memory access due to page faults, protection violations etc.
<ul style="list-style-type: none"><li>• ISI</li></ul>	Cannot fetch next instruction due to page faults, protection violations etc.
<ul style="list-style-type: none"><li>• External interrupt</li></ul>	Assertion of external interrupt signal.
<ul style="list-style-type: none"><li>• Alignment</li></ul>	Cannot perform memory access due to alignment problems, endian problems, etc. Implementation dependent.
<ul style="list-style-type: none"><li>• Program</li></ul>	Floating point operations, illegal instruction, privilege violation etc.
<ul style="list-style-type: none"><li>• Floating point unavailable</li></ul>	
<ul style="list-style-type: none"><li>• Decrementer</li></ul>	MSB of decrementer changes from 0 to 1.
<ul style="list-style-type: none"><li>• System call</li></ul>	
<ul style="list-style-type: none"><li>• Trace</li></ul>	Complete instruction w/o exception or context change.

# Precise Exceptions

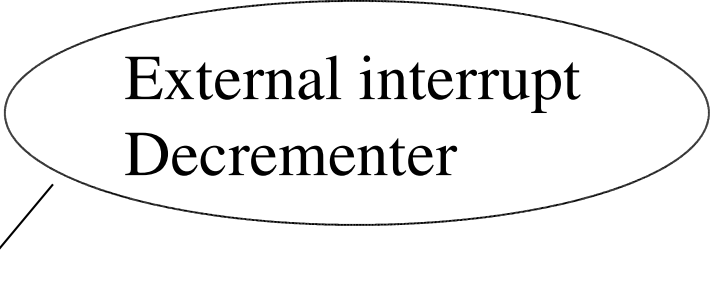
- Concurrent execution: To maximize performance, instructions are processed concurrently, independent of the sequence in program.
- Hardware ensures that program semantics are preserved.
- Difficult requirement to assure when interrupt occurs after instructions following “faulting” instruction have started or completed.
- Precise exception model: Automatically back machine up to the instruction that caused interrupt.
- MPC823 uses a history buffer to support precise exceptions.

# Interrupt Ordering

- Synchronous (i.e. instruction-related) exceptions are detected at any stage during instruction execution.
- The earliest exception found in the processing of an instruction precludes detection of further exceptions and is eventually handled.
- If more than one instruction in the pipeline causes an exception, only the first is taken. Any remaining synchronous exceptions are ignored.
- More than one asynchronous interrupt causes may be present at any time, in which case only the highest priority interrupt is taken.

# PowerPC Exception Priorities

Class	Priority	Exception
Non-maskable, async	1	System reset
	2	Machine check
Synchronous precise	3	Instruction dependent
Synchronous imprecise	4	Floating point
Maskable, async	5	External interrupt
	6	Decrementer



If MSR[EE]=0, delayed until bit is set

# Nesting Interrupts

- Nested interrupt: An interrupt that happens during the execution of an ISR.
  - Multiple interrupting devices with long ISRs
  - Virtual memory support for ISRs
  - Debugging ISR code
- What must ISRs do to support nested interrupts?

# Disabling Interrupts

- Sometimes interrupts must be avoided.
  - Time-critical instruction sequences (real-time applications)
  - Prologue sequence of ISRs
  - Changes in data structures shared with ISRs
- Synchronous interrupts: Not much choice. Can only be avoided by not executing instructions that might cause them such as illegal instructions or loads or stores to non-existent addresses.
- Asynchronous interrupts from external devices: Can be disabled (masked) using the external interrupt enable (EE) mode bit of MSR.
  - If 0, external interrupt signal is ignored.
  - If 1, external interrupt signal causes interrupt when asserted.
  - EE is automatically set to 0 in the beginning of **every** interrupt.
- Some interrupts are not maskable: Reset

# Managing Interrupts from Multiple Devices

- When multiple devices can interrupt the CPU, the system must determine which device to service
- Related issues
  - Identification: Which device caused the current interrupt?
  - Prioritization: If more than one device is simultaneously interrupting, which one is handled first?
- Three standard approaches
  - Non-vectored
  - Vectored
  - Autovectored

<http://www.chipcenter.com/eexpert/dgilbert/dgilbert005.html>



# Non-Vectored Interrupts

- Simplest hardware, least flexible.
- 6802, PowerPC, MIPS,...
- Single interrupt input to CPU.
- CPU always branches to same ISR.
- ISR polls each device to see which may have caused interrupt.
- Prioritization?

# Vectored Interrupts

- Used in 8080, 80x86, Z80
- As with non-vectorized interrupts, single CPU interrupt input.
- Interrupt handler (implemented in hardware) enables a branch to occur to a different address for each specific interrupt.
- CPU performs special *interrupt acknowledge* bus cycle to obtain interrupt vector number directly from device.
- Typically requires that devices be designed to work with specific CPU.
- Prioritization typically via daisy chain (look up in 370 material).

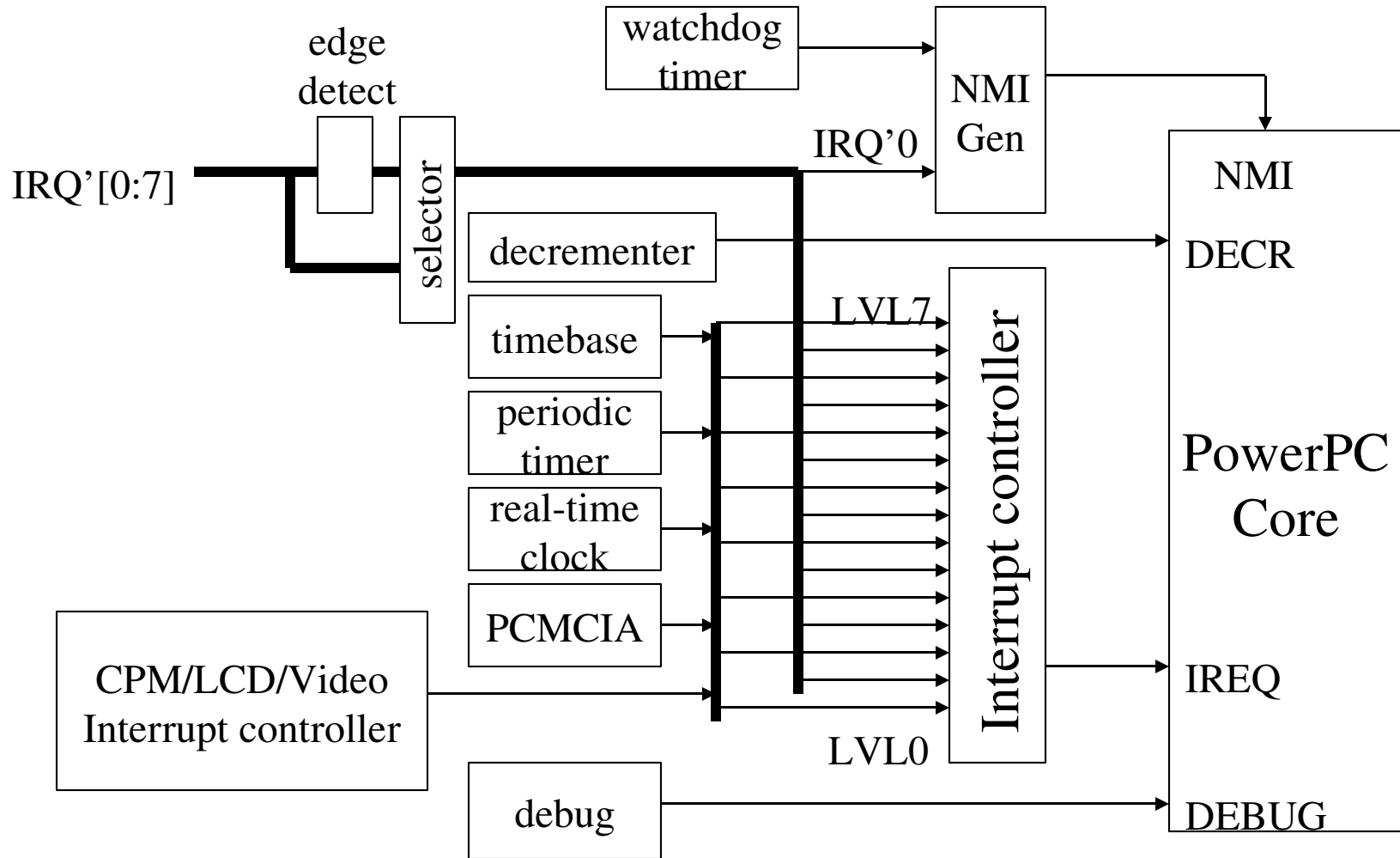
# Auto-Vectored Interrupts

- Used in 68000, SPARC.
- Can be built on top of vectored or non-vectored interrupts.
- Multiple CPU interrupt inputs, one for each priority level.
- Interrupt vector is supplied automatically based on highest-priority asserted input.
- CPU *interrupt priority level* controls which inputs are recognized.
  - For example, if IPL=3, levels 3, 4, 5, ... are disabled.
  - On interrupt, CPU automatically raises IPL to match level of request being serviced.
- Intel 8259A interrupt controller builds autovectoring on top of 80x86.
- MPC823 provides on-chip interrupt controller for pseudo-autovectored interrupts.

# MPC823 Interrupt Controller

- Part of on-chip System Interface Unit (SIU), not part of core. (Section 12.3 of data book)
- Eight external interrupt pins, each with its own dedicated interrupt priority level: IRQ0' (highest priority) through IRQ7' (lowest priority).
- Eight internal interrupt priorities, Level 0 through 7, generated by on-chip devices.
- Sixteen interleaved priorities: IRQ0' (non-maskable), Level 0, IRQ1', Level 1,...
- Assertion of any of these 16 interrupts can potentially assert the PowerPC external interrupt signal.

# MPC823 System Interface Unit



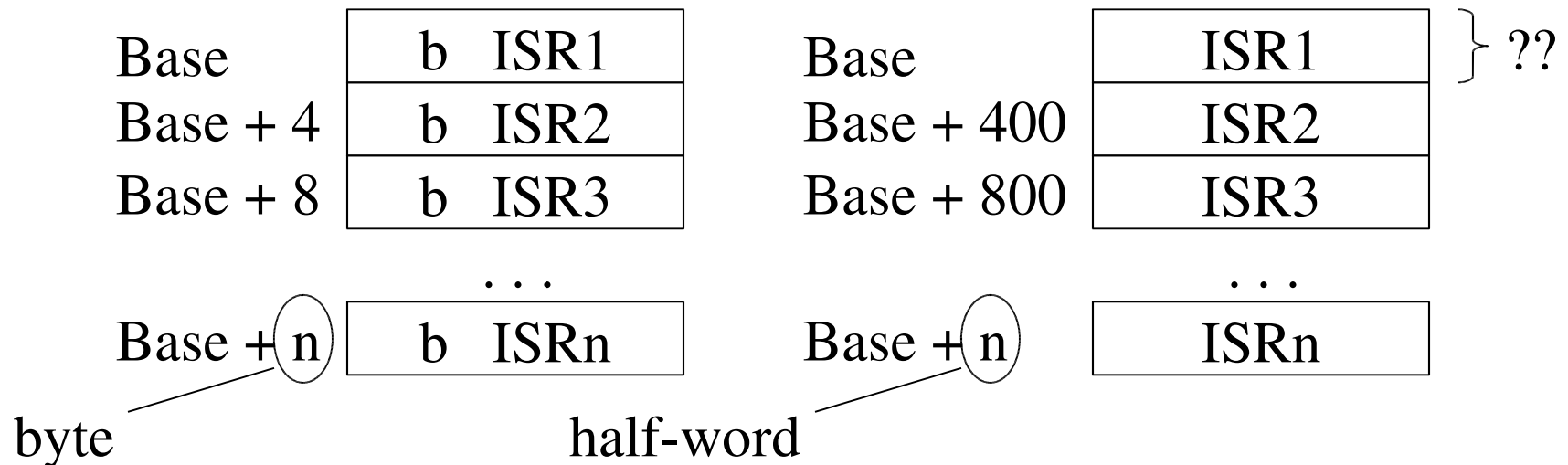
# Priority of SUI Interrupt Sources

number	priority level	source	code
0	highest	IRQ'0	00000000
1		Level 0	00000100
2		IRQ'1	00001000
3		Level 1	00001100
4		IRQ'2	00010000
5		Level 2	00010100
..		.....	.....
..		.....	.....
14	lowest	IRQ'7	00111000
15		Level 7	00111100
16—31		reserved	--

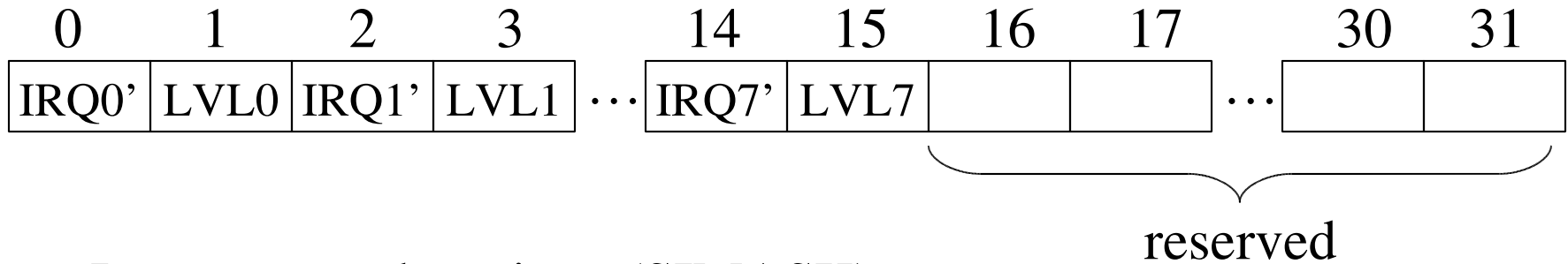
# Programming the Interrupt Controller (1/2)

8-bit code	reserved	reserved	reserved
------------	----------	----------	----------

- Interrupt vector register (SIVEC)
  - 8-bit code gives the unmasked interrupt source of the highest priority level (0-15).
  - Read-only memory-mapped register gives interrupt code (= source number \* 4) that may be read as byte, half-word, or word.
  - Used to index interrupt table.



# Programming the Interrupt Controller (2/2)



- Interrupt mask register (SIMASK)
  - Read/Write memory-mapped register.
  - Bits enable/disable each interrupt.
- Interrupt pending register (SIPEND)
  - Read/Write memory-mapped register.
  - Bits indicate which interrupts are pending.
- Interrupt edge/level register (SIEL)
  - Read/Write memory-mapped register.
  - Bits indicate level-sensitive/edge-triggered interrupt detection and exit from low-power mode.



# PPC: Interrupts in detail

