NIOS Processor Interrupts

Engineering 304 Lab

What are Exceptions or Interrupts?

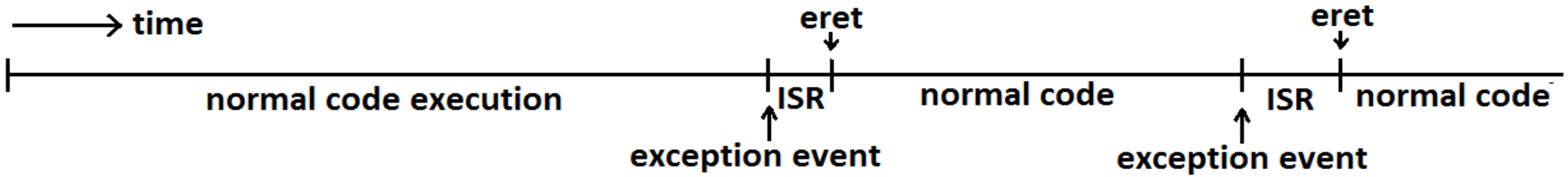
- Exceptions and interrupts are unplanned events that the system must respond to.
- Examples
  - Packet of data received from another system
  - Potential buffer overflow problem
  - Invalid opcode in an instruction
  - Illegal math operation
  - “Sleep” button pressed
  - Power supply losing power
  - Keyboard entry has occurred

Handling External Inputs

- Polling Method
  - Main routines in software continually check I/O devices to see if anything has arrived (e.g. lab 5 and the TO flag)
- Interrupt Method
  - CPU runs normal code
  - Interrupt Service Routine (ISR) is special code, written to handle interrupts
  - CPU hardware detects an interrupt signal and causes normal code to be suspended and the ISR to be executed
  - After ISR completes, normal code is resumed
  - A mechanism is sometimes needed to communicate between ISR and normal code

Interrupts

- ISRs have some similarities to a function called at random times by hardware-based events
- Must protect all registers except **et** (=r24) and **ea** (=r29) registers while running the ISR code
  - Normal code should not sense anything happened except via the communication mechanism
  - **et** and **ea** are exclusively used by interrupt service routines
- Multiple levels of interrupt enabling is common

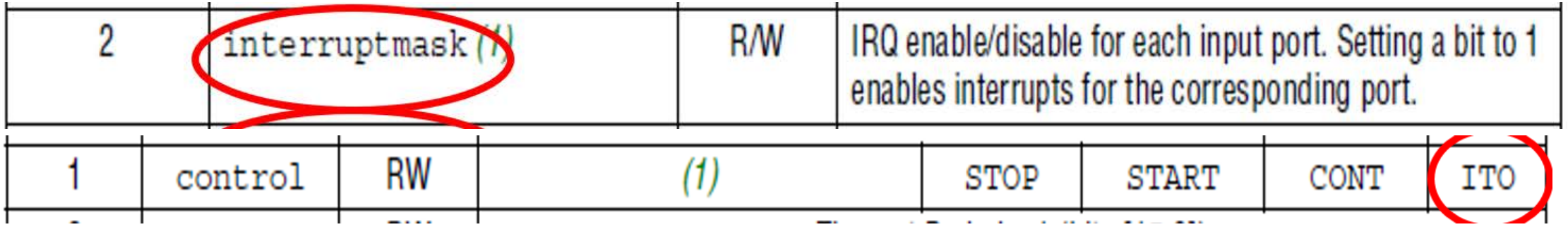


Sources of NIOS Interrupts/Exceptions

- Three common types of exceptions
  - Invalid instruction
    - E.g. `mul r2, r16, r2` when there isn't hardware for doing multiply
    - E.g. invalid opcode found in the op field of the machine code fetched
  - Math errors
    - E.g. divide-by-zero errors and others that might be implemented
  - External Interrupts
    - I/O ports capable of asserting the "int" interrupt signal
      - NIOS allows up to 32 different I/O ports to each cause an interrupt
      - We are only using two: timer timeout and key press edge detection
    - E.g. timer times out and causes an interrupt to the CPU

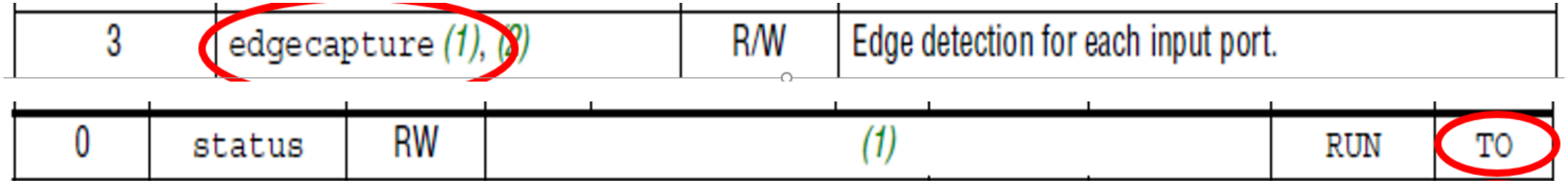
Turning on NIOS Interrupts

- Begin by enabling interrupts in each port
  - Usually an enable bit in a port register must be set (e.g. MASKing)



2	interruptmask (1)		R/W	IRQ enable/disable for each input port. Setting a bit to 1 enables interrupts for the corresponding port.			
1	control	RW	(1)	STOP	START	CONT	ITO

- Next, be sure any current interrupt conditions are cleared
  - Clear any interrupt-generating flag bits in the port registers



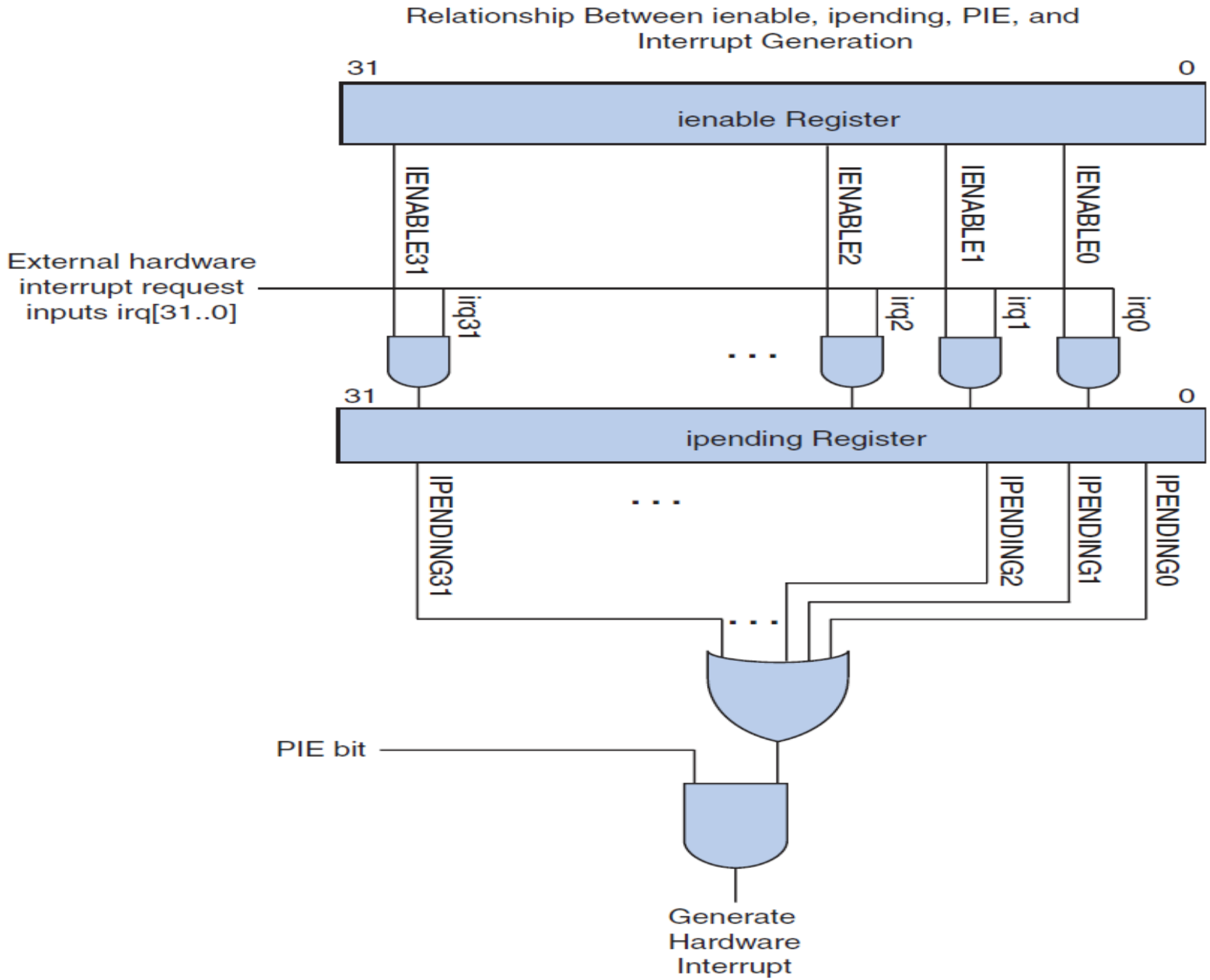
3	edgecapture (1), (2)		R/W	Edge detection for each input port.			
0	status	RW	(1)			RUN	TO

Turning on NIOS Interrupts – cont.

- Next, enable the desired port interrupts in the CPU
  - By setting the bits high in the IENABLE register (CTL3, not CT13)
- Finally, set the PIE bit in the CPU STATUS register (CTL0) to enable interrupts globally on the CPU
- Note, “rdctl” and “wrctl” instructions allow you to access the CTLx registers to set or clear specific bits

Table 3–2. Control Register & Bits

Register	Name	31…2	1	0
ctl0	status	Reserved	U	PIE
ctl1	estatus	Reserved	EU	EPIE
ctl2	bstatus	Reserved	BU	BPIE
ctl3	ienable	Interrupt-enable bits		
ctl4	ipending	Pending-interrupt bits		
ctl5	cpuid	Unique processor identifier		


Relationship Between ienable, ipending, PIE, and Interrupt Generation
31
0
ienable Register
IENABLE31
IENABLE2
IENABLE1
IENABLE0
External hardware interrupt request inputs irq[31..0]
irq31
irq2
irq1
irq0
31
0
ipending Register
IPENDING31
IPENDING2
IPENDING1
IPENDING0
PIE bit
Generate Hardware Interrupt

NIOS Interrupt Execution Sequence

- An interrupt condition in an I/O port causes “INT” signal to be asserted
- CPU hardware copies the PC to the EA register (like RA) and copies the STATUS register (CTL0) to the ESTATUS (CTL1) register (backup)
- CPU hardware disables further interrupts (clearing PIE bit)
- CPU hardware sets the PC to the address of the ISR
- ISR software is executed, clearing interrupt conditions, handling the interrupt, and communicating with the main program as needed
- The last ISR instruction is “eret” which restores CTL0 and restores the PC (based on the EA register) (like “ret”)

Table 3–2. Control Register & Bits

Register	Name	31…2	1	0
ctl0	status	Reserved	U	PIE
ctl1	estatus	Reserved	EU	EPIE
ctl2	bstatus	Reserved	BU	BPIE
ctl3	ienable	Interrupt-enable bits		
ctl4	ipending	Pending-interrupt bits		
ctl5	cpuid	Unique processor identifier		

Creating an ISR

- Begin by pushing any register used besides ET and EA
- Next, the code must correct the EA register if the exception is from an external interrupt
- Each of the up to 32 possible interrupts are checked one at a time
  - IPENDING register (CTL4) holds flags for each interrupt port
- For each pending interrupt, ISR needs to
  - Communicate with normal software routines (e.g. global variable) or take the appropriate action if it is simple
  - Clear the port register flag that is causing INT to be asserted (e.g. TO bit)
- Pop any registers pushed on the stack
- End the ISR with the “eret” instruction

PIO Core (Peripheral I/O device/port)

Table 9–2. Register Map for the PIO Core

Offset	Register Name		R/W	(n-1)	...	2	1	0
0	data	read access	R	Data value currently on PIO inputs				
		write access	W	New value to drive on PIO outputs				
1	direction *(1)*		R/W	Individual direction control for each I/O port. A value of 0 sets the direction to input; 1 sets the direction to output.				
2	interruptmask *(1)*		R/W	IRQ enable/disable for each input port. Setting a bit to 1 enables interrupts for the corresponding port.				
3	edgecapture *(1)*, *(2)*		R/W	Edge detection for each input port.				

Interval Timer Core

Table 24–3. Register Map—32-bit Timer

Offset	Name	R/W	Description of Bits						
			15	...	4	3	2	1	0
0	status	RW	*(1)*					RUN	TO
1	control	RW	*(1)*			STOP	START	CONT	ITO
2	periodl	RW	Timeout Period – 1 (bits [15:0])						
3	periodh	RW	Timeout Period – 1 (bits [31:16])						
4	snapl	RW	Counter Snapshot (bits [15:0])						
5	snaph	RW	Counter Snapshot (bits [31:16])						