CSCE 4114
Interrupts

David Andrews
dandrews@uark.edu

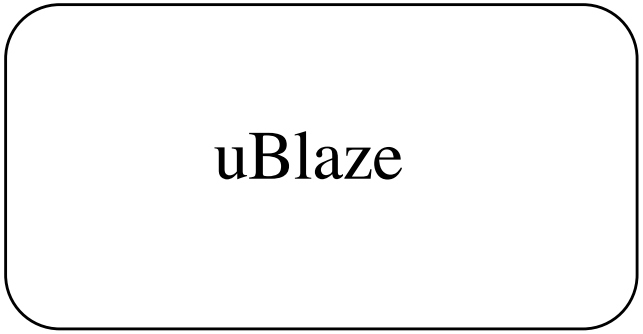
Interrupts

"An Asynchronous signal indicating the need for attention or a synchronous event in software indicating the need for a change in execution."

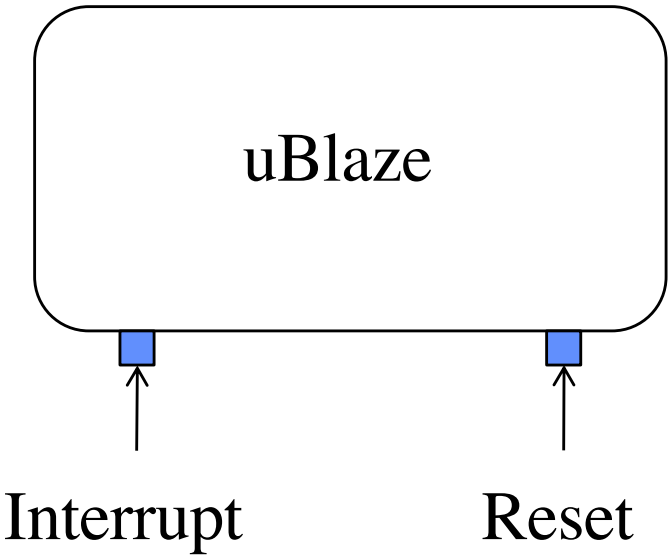
Hardware interrupts introduced to avoid wasting the processor's valuable time in polling loops, waiting for external events.

-Wikipedia

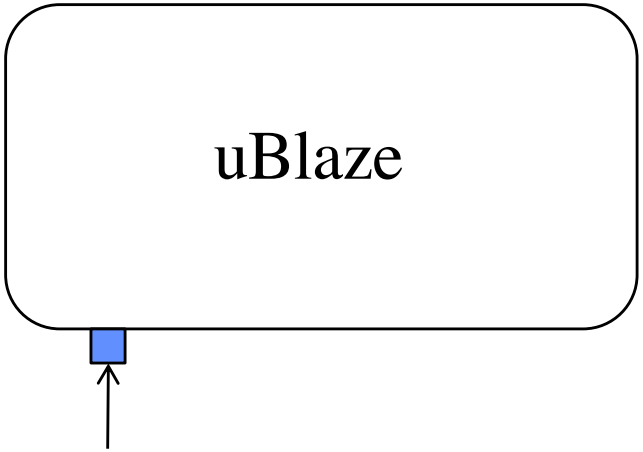
Interfacing



Interfacing

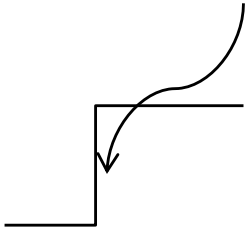


Interfacing

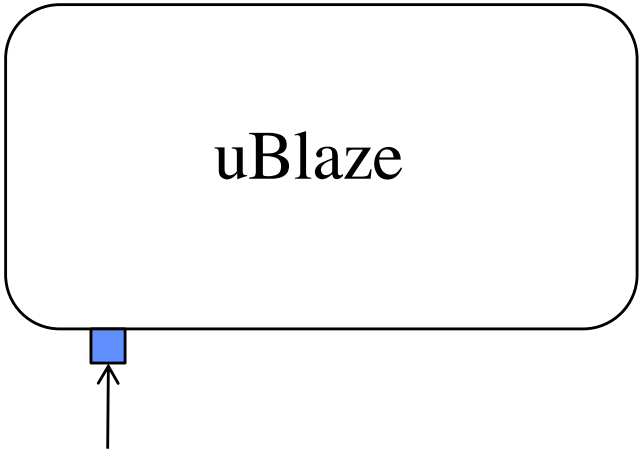


Interrupt

Signal can be edge triggered

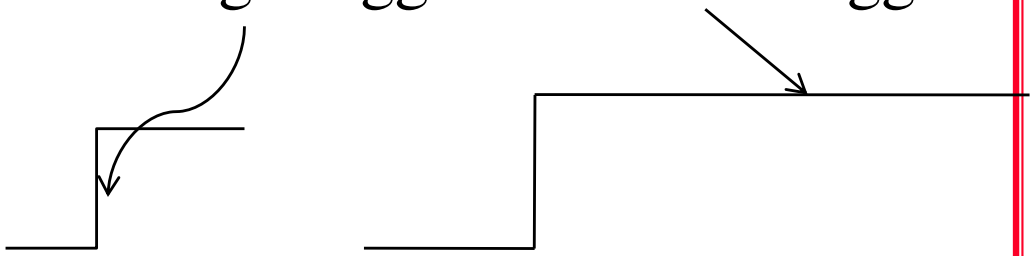


Interfacing

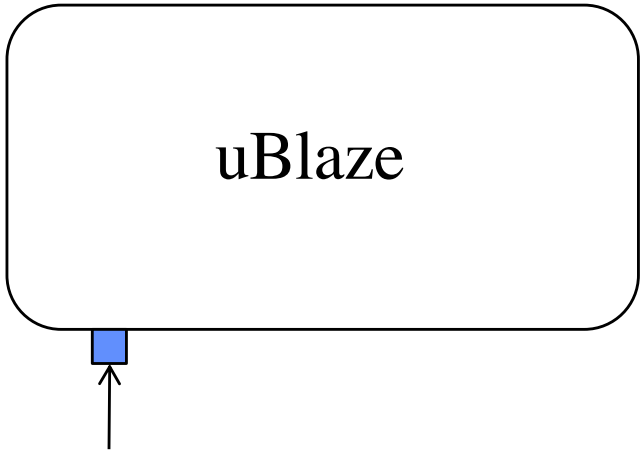


Interrupt

Signal can be edge triggered or level triggered



Internal Processing

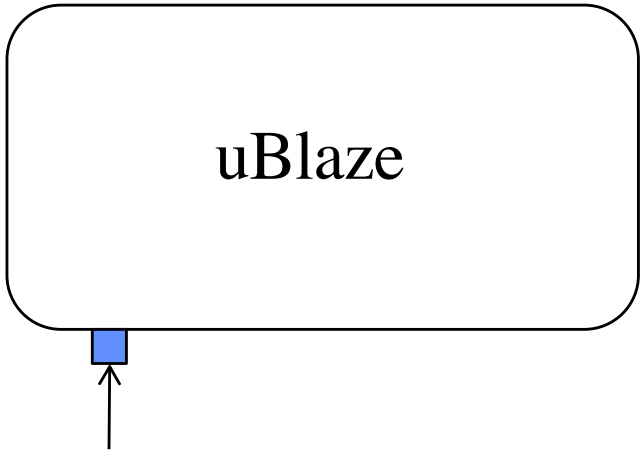


Save Program Counter

Clear Interrupt Enable (IE) Bit in MSR

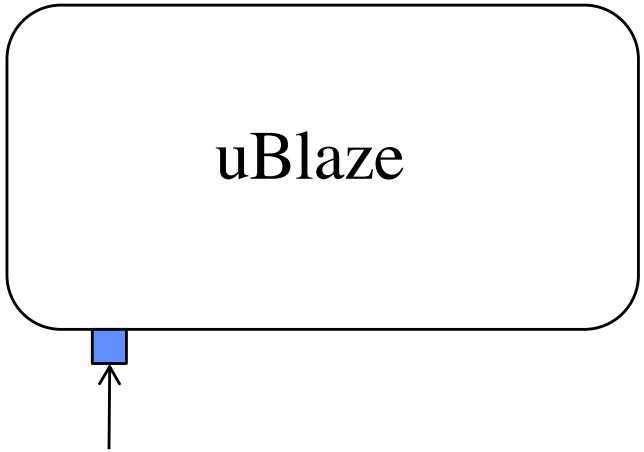
---Jump to ISR routine and execute---

Internal Processing



r14 ←PC
PC ← 0x00000010
MSR[IE] ←0
MSR[UMS] ←MSR[UM], MSR[UM] ← 0,
MSR[VMS] ← MSR[VM], MSR[VM] ← 0
Reservation ←0

Internal Processing

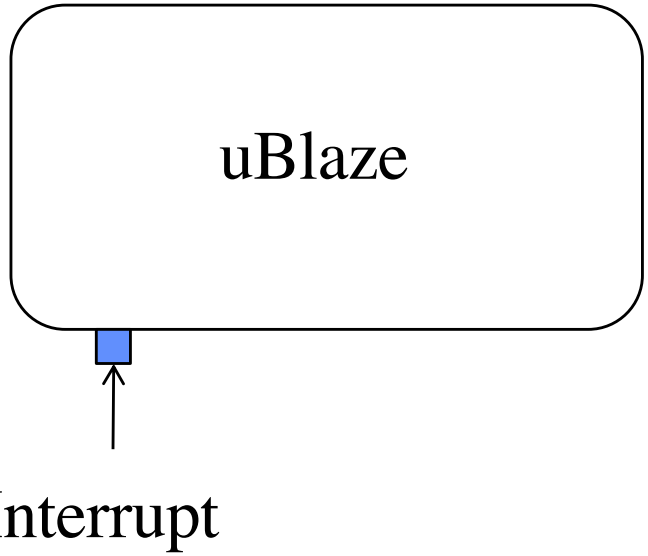


r14 ←PC
PC ←0x00000010
MSR[IE] ←0
MSR[UMS] ←MSR[UM], MSR[UM] ← 0;
MSR[VMS] ← MSR[VM], MSR[VM] ← 0
Reservation ←0

Only with MMU

Internal Processing

r14 ←PC
PC ←0x00000010
MSR[IE] ←0



New Concept: "Vectoring"

Event	Vector Address	Register File Return Address
Reset	0x00000000 - 0x00000004	-
User Vector (Exception)	0x00000008 - 0x0000000C	Rx
Interrupt	0x00000010 - 0x00000014	R14
Break: Non-maskable hardware	0x00000018 - 0x0000001C	R16
Break: Hardware		
Break: Software		
Hardware Exception	0x00000020 - 0x00000024	R17 or BTR
Reserved by Xilinx for future use	0x00000028 - 0x0000004F	-

New Concept: "Vectoring"

```
0x00:    bri     _start1
0x04:    nop
0x08:    imm     high bits of address (user exception handler)
0x0c:    bri     _exception_handler
0x10:    imm     high bits of address (interrupt handler)
0x14:    bri     _interrupt_handler
0x20:    imm     high bits of address (HW exception handler
0x24:    bri     _hw_exception_handler
```

New Concept: "Vectoring"

```
0x00:   bri     _start1
0x04:   nop
0x08:   imm     high bits of address (user exception handler)
0x0c:   bri     _exception_handler
0x10:   imm     high bits of address (interrupt handler)
0x14:   bri     _interrupt_handler
0x20:   imm     high bits of address (HW exception handler
0x24:   bri     _hw_exception_handler
```

Execution Flow

o
o
o
addi r1,r1,4
sub r1,r2,r3 ← *Bamb: Interrupt !*
add r1,r2,r3
bri loop
o
o
o

Execution Flow

o
o
o
addi r1,r1,4
sub r1,r2,r3 ← *Bamb: Interrupt !*
add r1,r2,r3
bri loop
o
o
o

1) R14 ← PC

Execution Flow

o
o
o
addi r1,r1,4
sub r1,r2,r3 ← *Bamb: Interrupt !*
add r1,r2,r3
bri loop
o
o
o

1) R14 ← PC

2) MSR[IE] ← 0

Execution Flow

o
o
o
addi r1,r1,4
sub r1,r2,r3 ← *Bamb: Interrupt !*
add r1,r2,r3
bri loop
o
o
o
o

1) R14 ← PC

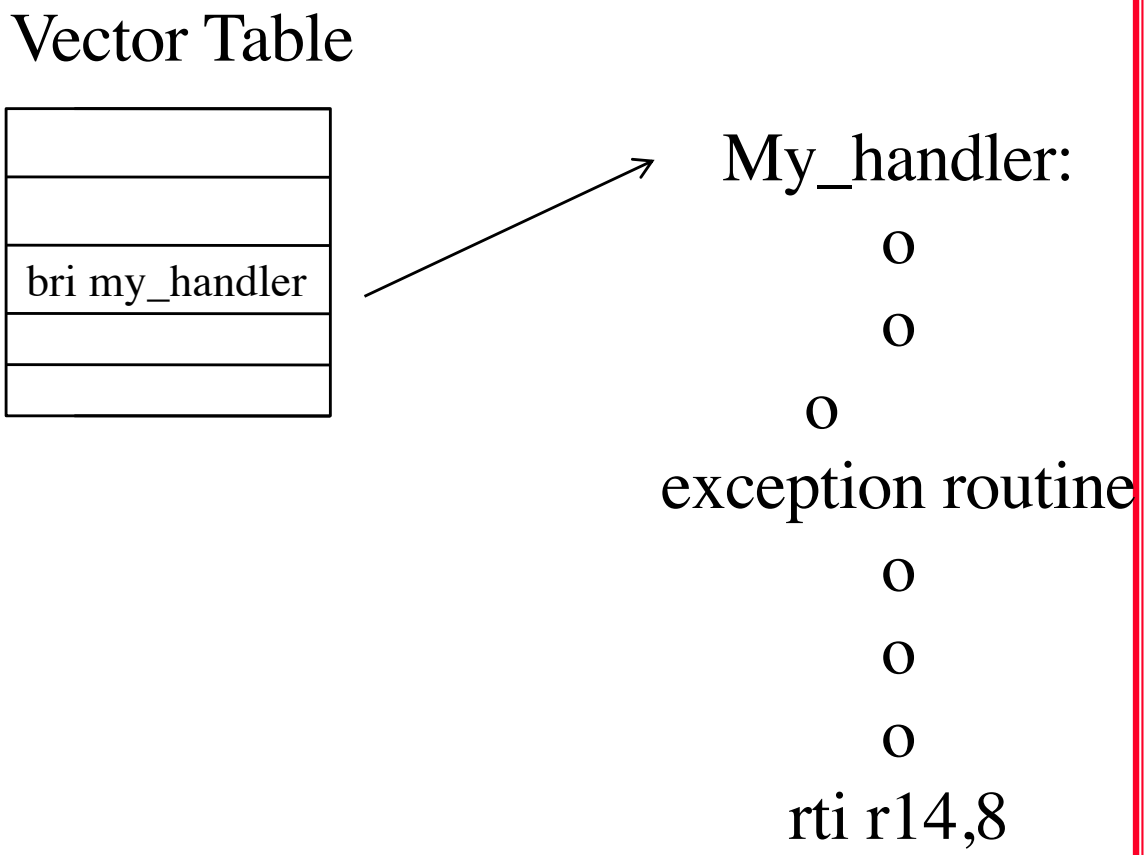
2) MSR[IE] ← 0

3) PC ← 0x0000010

Execution Flow

o
o
o
addi r1,r1,4
sub r1,r2,r3
add r1,r2,r3
bri loop
o
o
o

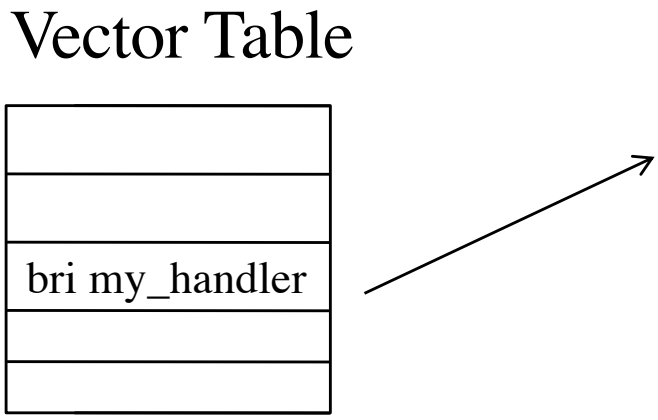
4)



Execution Flow

o
o
o
addi r1,r1,4
sub r1,r2,r3
add r1,r2,r3
bri loop
o
o
o

4)



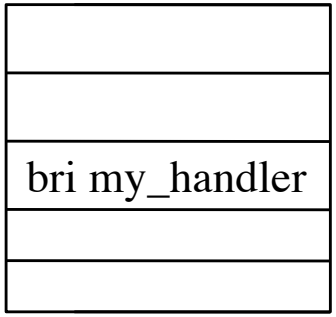
My_handler:
o
o
o
exception routine
o
o
o
5) rti r14,8

Execution Flow

o
o
o
addi r1,r1,4
sub r1,r2,r3
add r1,r2,r3
bri loop
o
o
o

4)

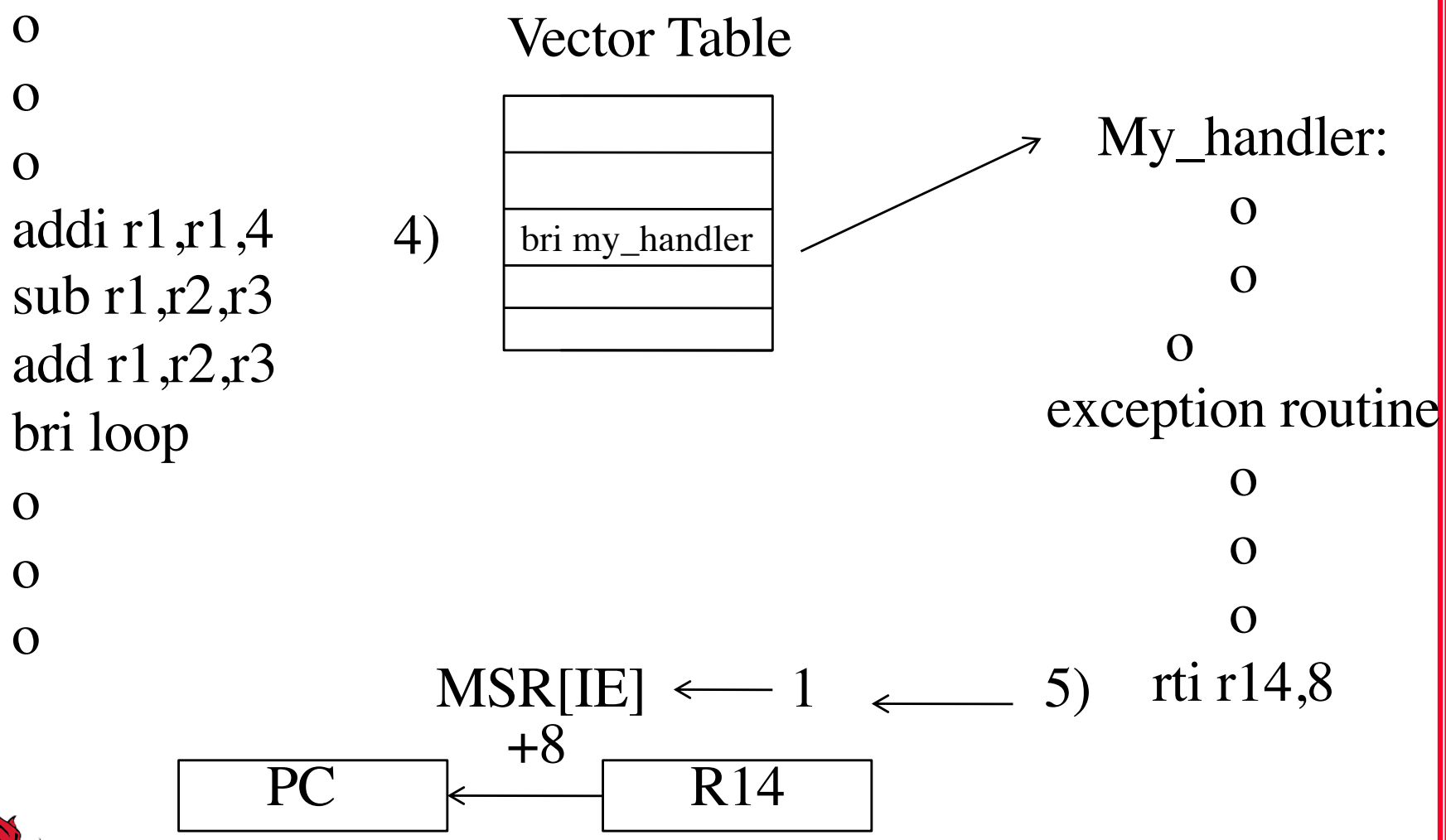
Vector Table



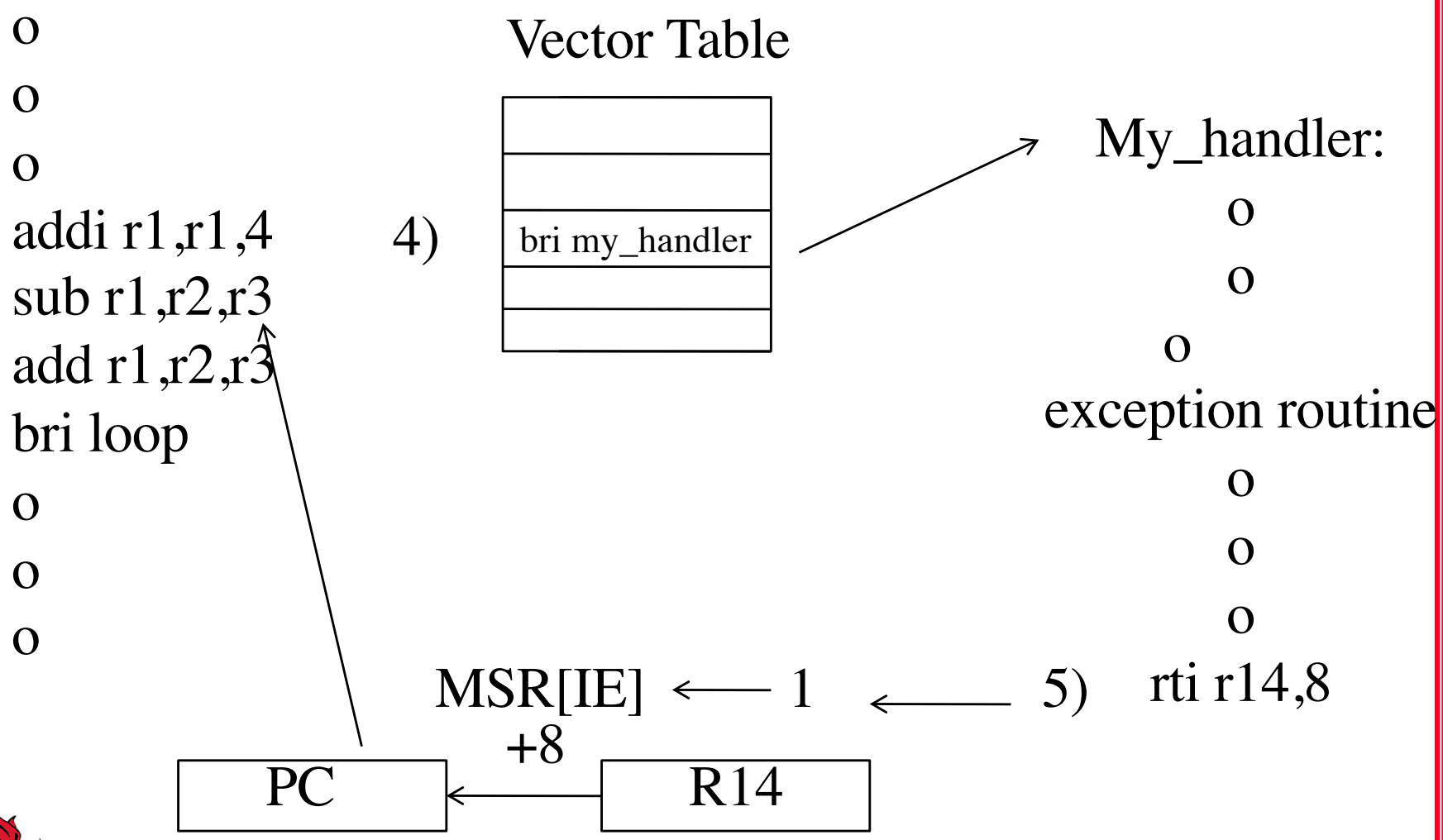
My_handler:
o
o
o
exception routine
o
o
o
o

MSR[IE] ← 1 ← 5) rti r14,8

Execution Flow



Execution Flow



MSR: Machine Status Register

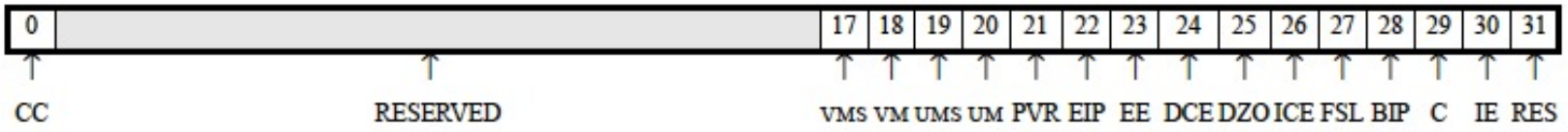


Figure 2-4: **MSR**

MSR: Machine Status Register

Figure 2-4: **MSR**

IE:= Interrupt Enable
1= Enabled
0=Disabled

MSR: Machine Status Register

Figure 2-4: **MSR**

MSR is a "Privileged" Register:

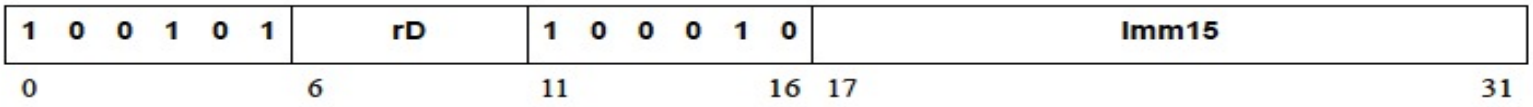
MSR: Machine Status Register

Figure 2-4: **MSR**

MSR is a "Privileged" Register:

msrclr **Read MSR and clear bits in MSR**

msrclr rD, Imm



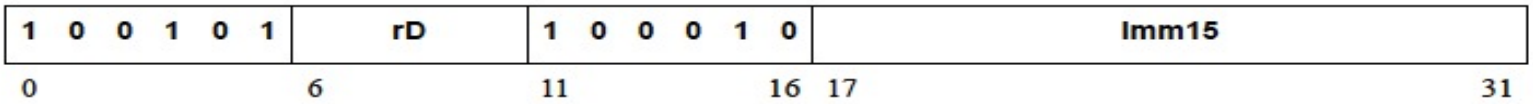
MSR: Machine Status Register

Figure 2-4: **MSR**

MSR is a "Privileged" Register:

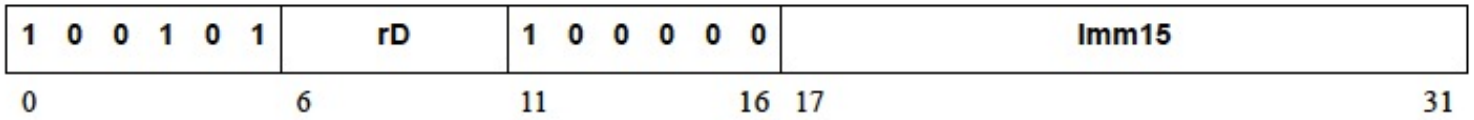
msrclr **Read MSR and clear bits in MSR**

msrclr rD, Imm



msrset **Read MSR and set bits in MSR**

msrset rD, Imm



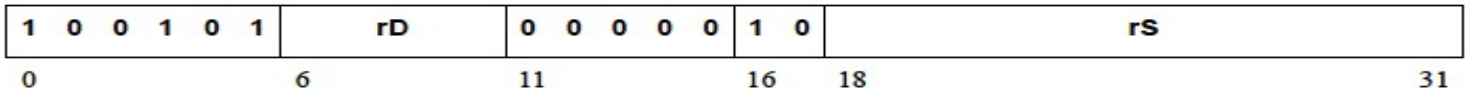
MSR: Machine Status Register

Figure 2-4: **MSR**

MSR is a "Privileged" Register:

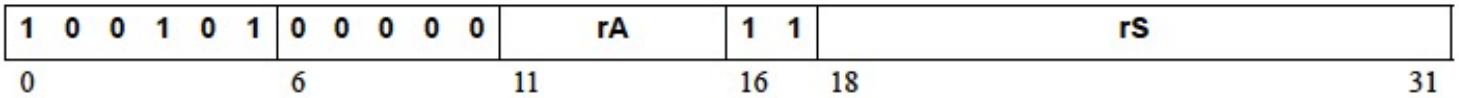
mfs **Move From Special Purpose Register**

mfs rD, rS

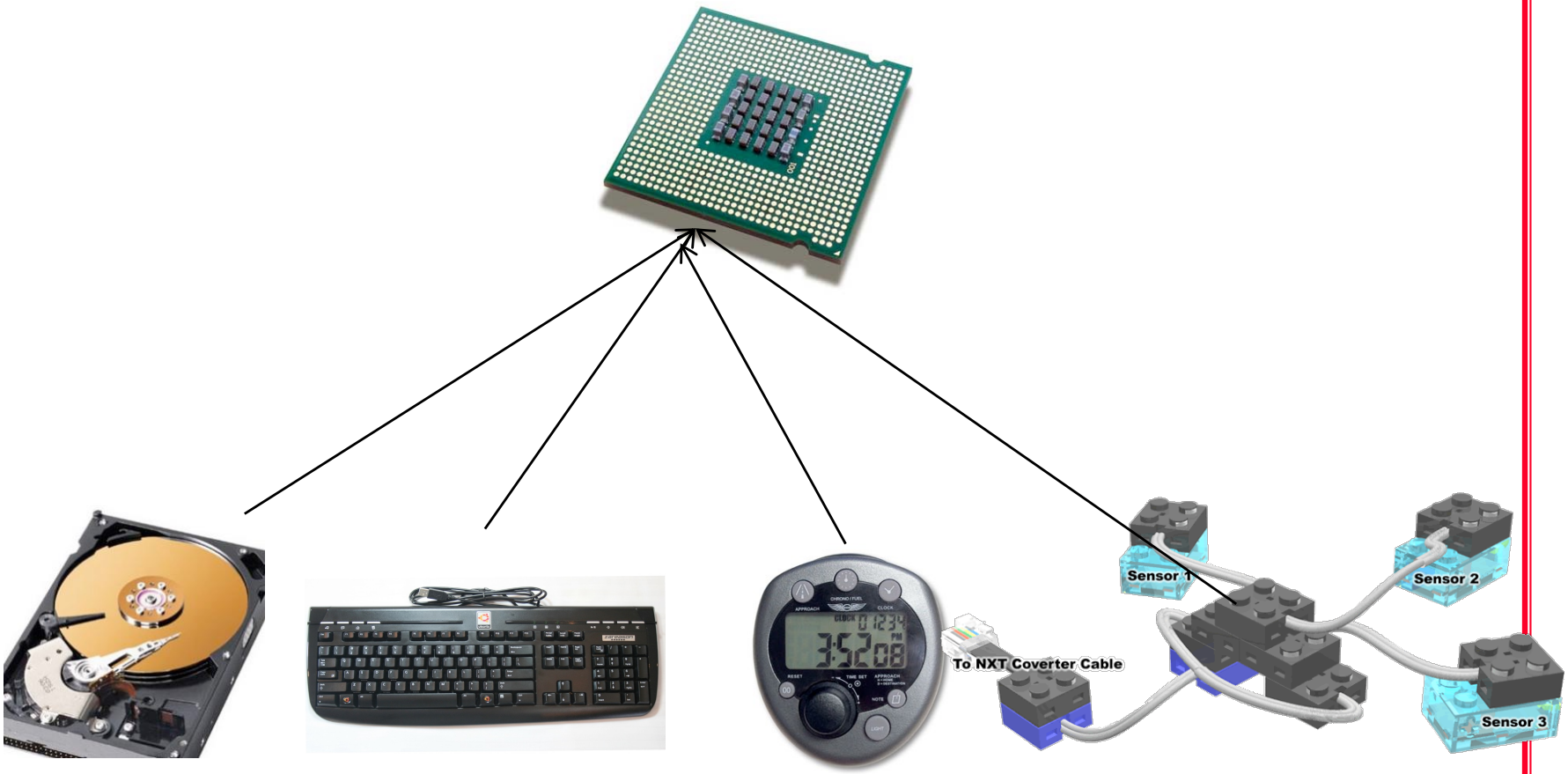


mts **Move To Special Purpose Register**

mts rS, rA

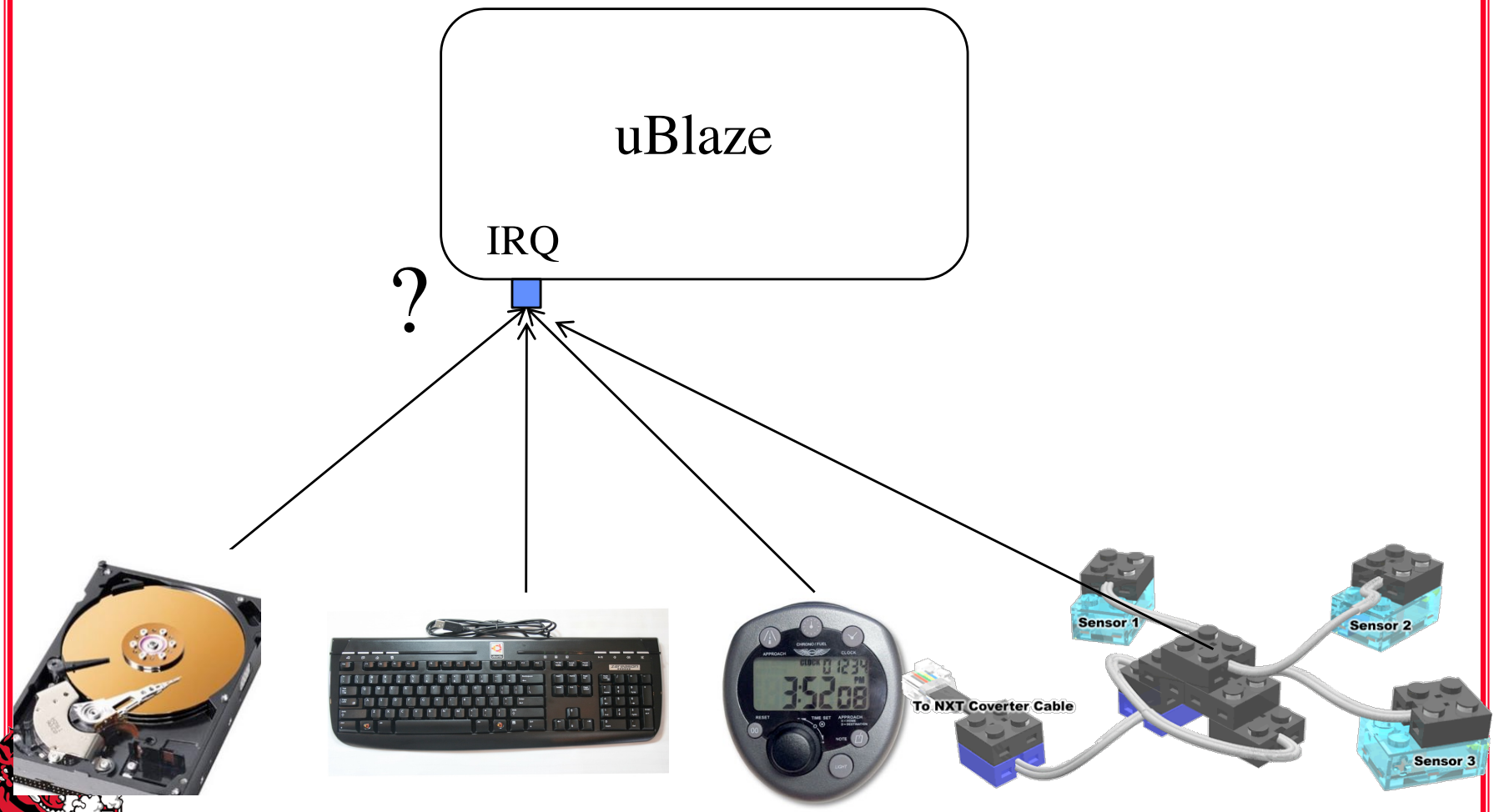


Where do Interrupts Come from ?

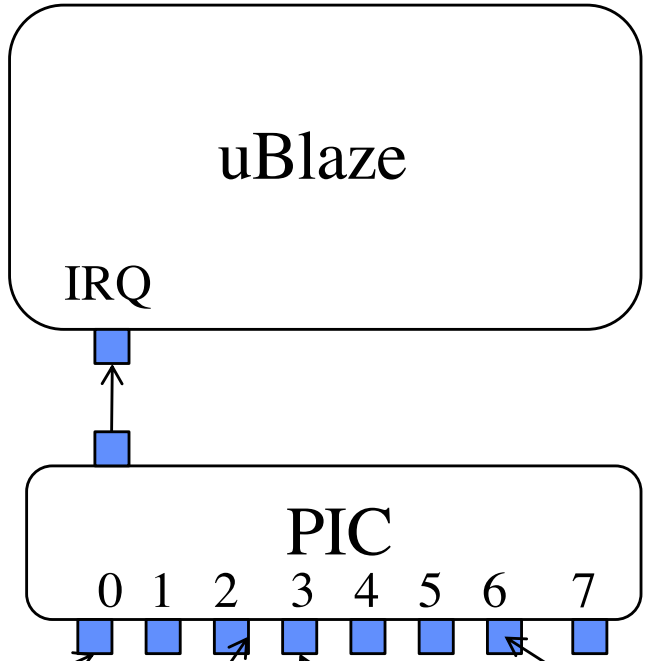


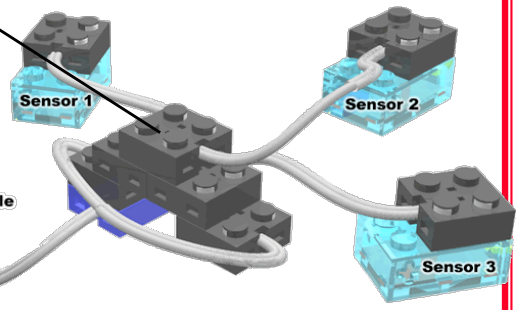
Devices External to CPU

Our CPU Has 1 External Input

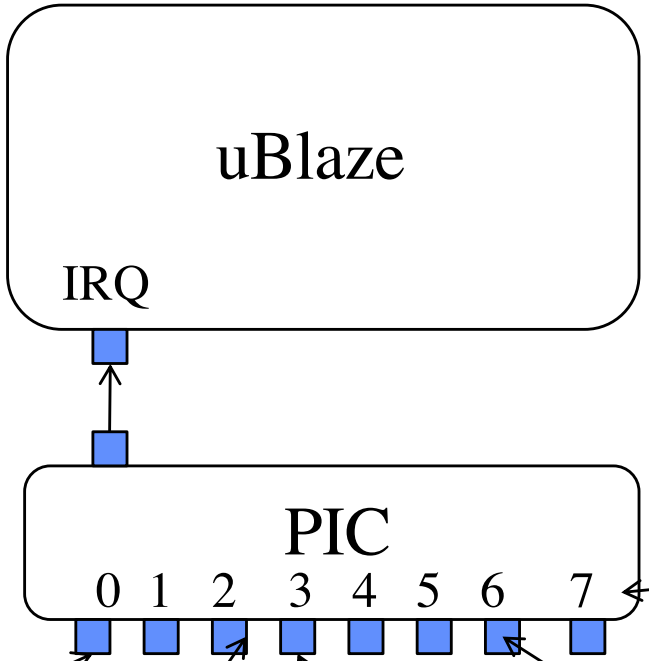


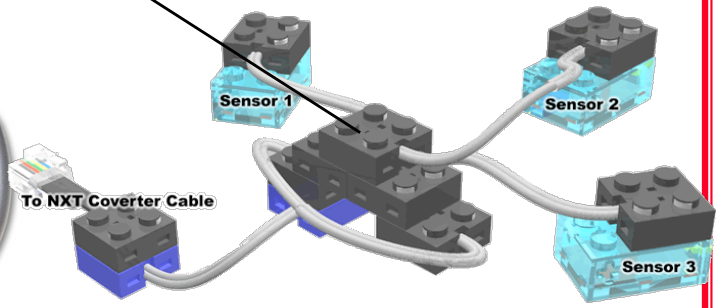
Priority Interrupt Controller (PIC)



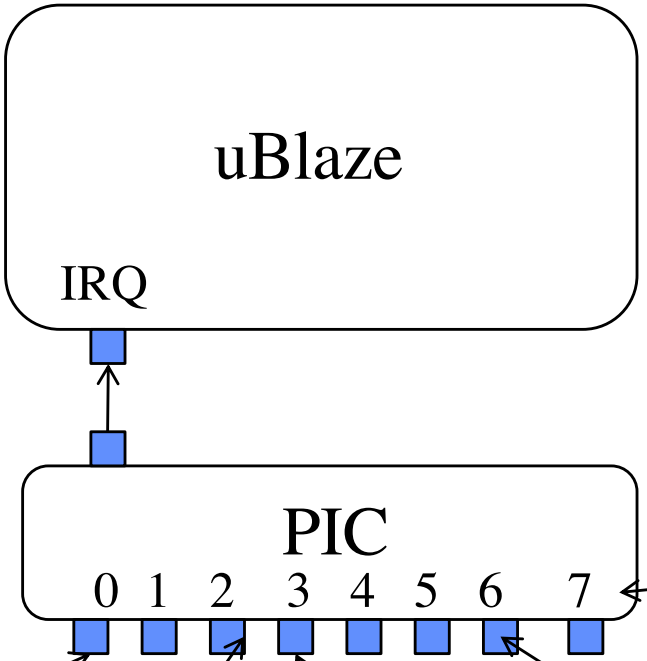


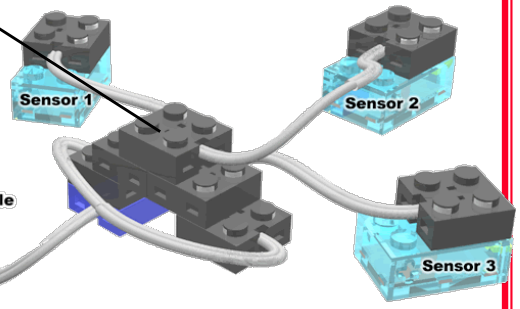
Priority Interrupt Controller (PIC)





Priority Interrupt Controller (PIC)





PIC REGISTER SET

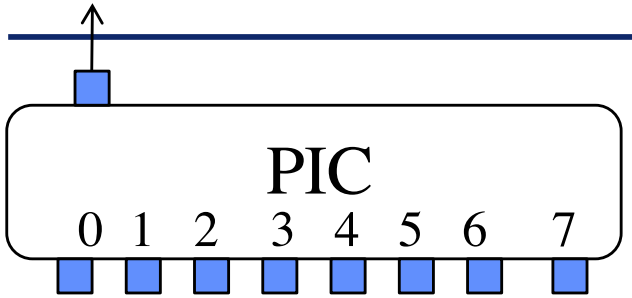


Table 2-4: **Register Address Mapping**

Address Offset	Register Name	Description
00h	ISR	Interrupt Status Register (ISR)
04h	IPR	Interrupt Pending Register (IPR)
08h	IER	Interrupt Enable Register (IER)
0Ch	IAR	Interrupt Acknowledge Register (IAR)
10h	SIE	Set Interrupt Enables (SIE)
14h	CIE	Clear Interrupt Enables (CIE)
18h	IVR	Interrupt Vector Register (IVR)
1Ch	MER	Master Enable Register (MER)
20h	IMR	Interrupt Mode Register (IMR)
24h	ILR	Interrupt Level Register (ILR)
100h to 17Ch	IVAR	Interrupt Vector Address Register (IVAR)
200h to 2FCh	IVEAR	Interrupt Vector Extended Address Register (IVEAR)

PIC REGISTER SET

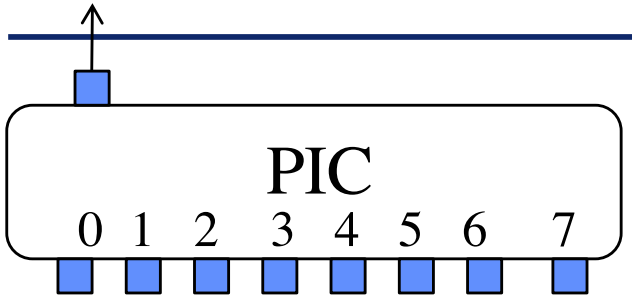
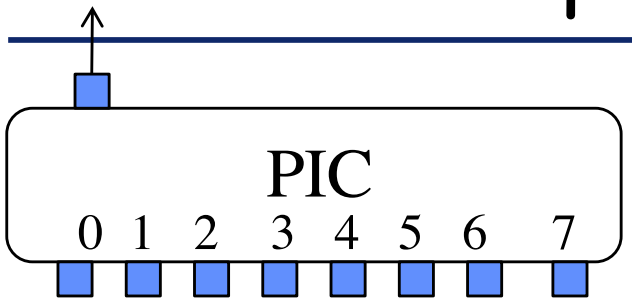


Table 2-4: **Register Address Mapping**

Address Offset	Register Name	Description
00h	ISR	Interrupt Status Register (ISR)
08h	IER	Interrupt Enable Register (IER)
0Ch	IAR	Interrupt Acknowledge Register (IAR)
1Ch	MER	Master Enable Register (MER)
100h to 17Ch	IVAR	Interrupt Vector Address Register (IVAR)

IsR: Interrupt Status Register



a)Which Interrupts requesting service
b) Debugging: Writing a
"1" will actually trigger interrupt

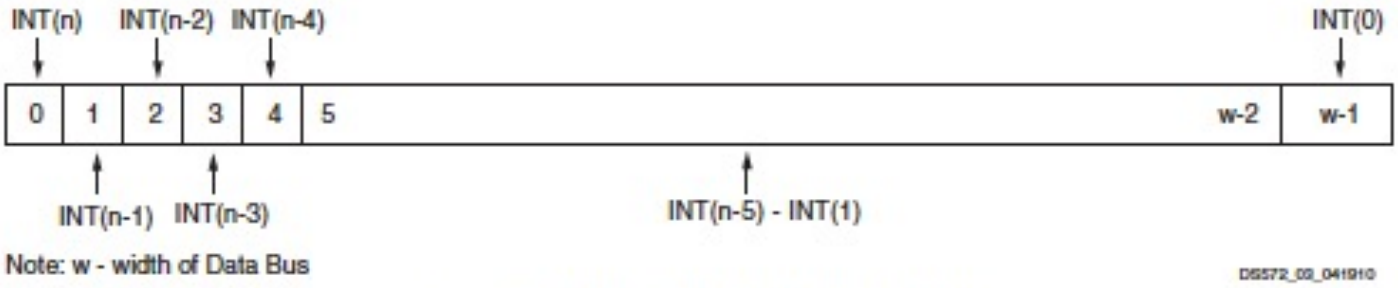


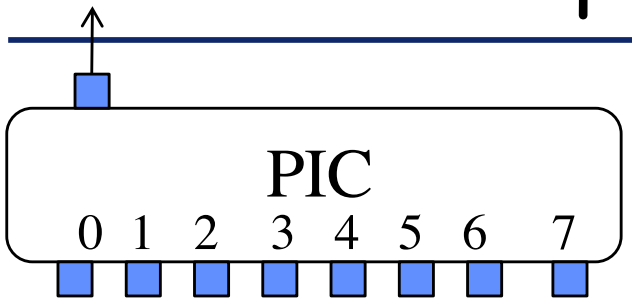
Figure 3: **Interrupt Status Register (ISR)**

Table 5: **Interrupt Status Register**

Bits	Name	Core Access	Reset Value	Description
0 : (w[1] –1)	INT(n) – INT(0) ($n \leq w-1$)	Read / Write	All Zeros	**Interrupt Input (n) – Interrupt Input (0)** '0' = Not active '1' = Active

1. w - Width of Data Bus

IER: Interrupt Enable Register



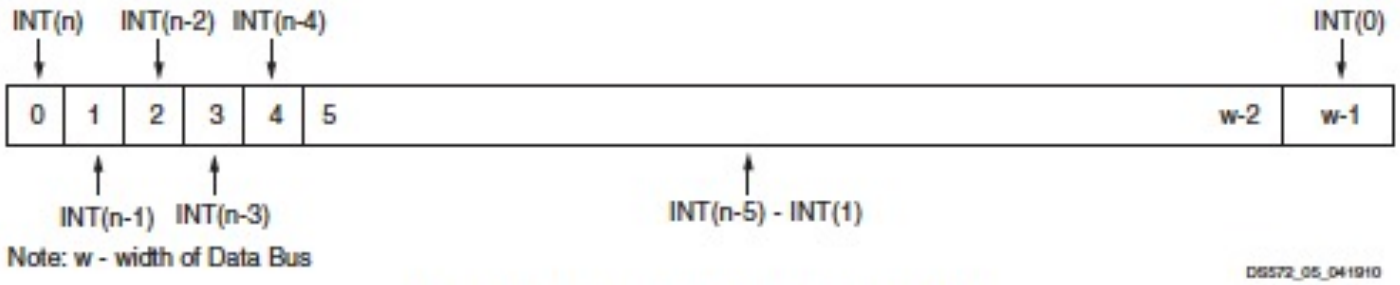


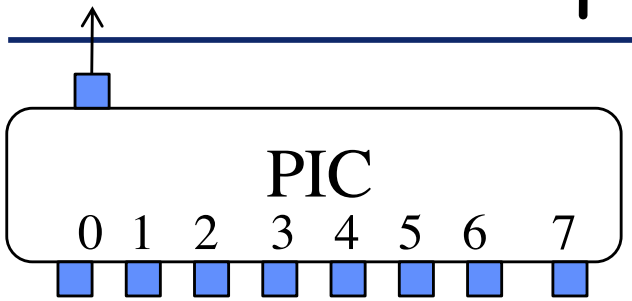
Figure 5: **Interrupt Enable Register (IER)**

Table 7: **Interrupt Enable Register**

Bits	Name	Core Access	Reset Value	Description
0 : ($w^{(1)}$ – 1)	INT(n) – INT(0) ($n \leq w - 1$)	Read / Write	All Zeros	**Interrupt Input (n) – Interrupt Input (0)** '1' = Interrupt enabled '0' = Interrupt disabled

1. w - Width of Data Bus

IER: Interrupt Enable Register



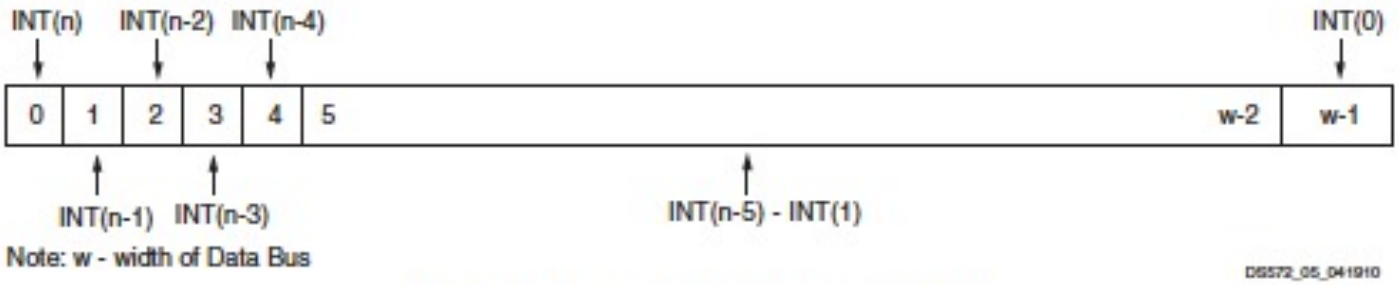


Figure 5: **Interrupt Enable Register (IER)**

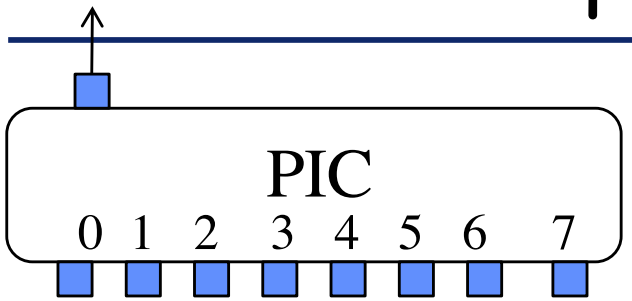
Table 7: **Interrupt Enable Register**

Bits	Name	Core Access	Reset Value	Description
0 : ($w^{(1)}$ – 1)	INT(n) – INT(0) ($n \leq w - 1$)	Read / Write	All Zeros	**Interrupt Input (n) – Interrupt Input (0)** '1' = Interrupt enabled '0' = Interrupt disabled

1. w - Width of Data Bus

Only Enables/Disables each interrupt: does not cause int

IAR: Interrupt Acknowledge Register



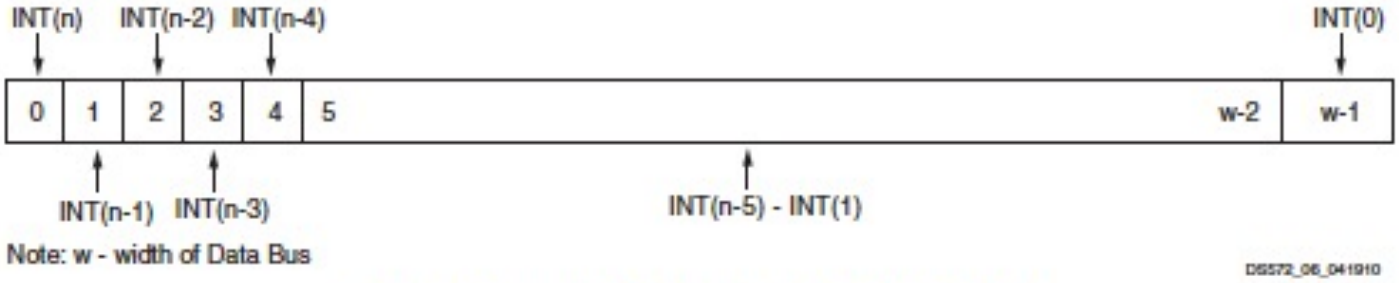


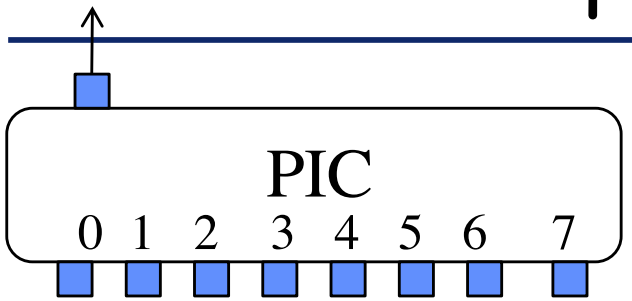
Figure 6: **Interrupt Acknowledge Register (IAR)**

Table 8: **Interrupt Acknowledge Register**

Bits	Name	Core Access	Reset Value	Description
0 : (w[1] – 1)	INT(n) – INT(0) ($n \leq w - 1$)	Write	All Zeros	**Interrupt Input (n) – Interrupt Input (0)** '1' = Clear Interrupt '0' = No action

1. w - Width of Data Bus

IAR: Interrupt Acknowledge Register



Clears a pending interrupt:
What would happen if you did not clear the pending interrupt ?

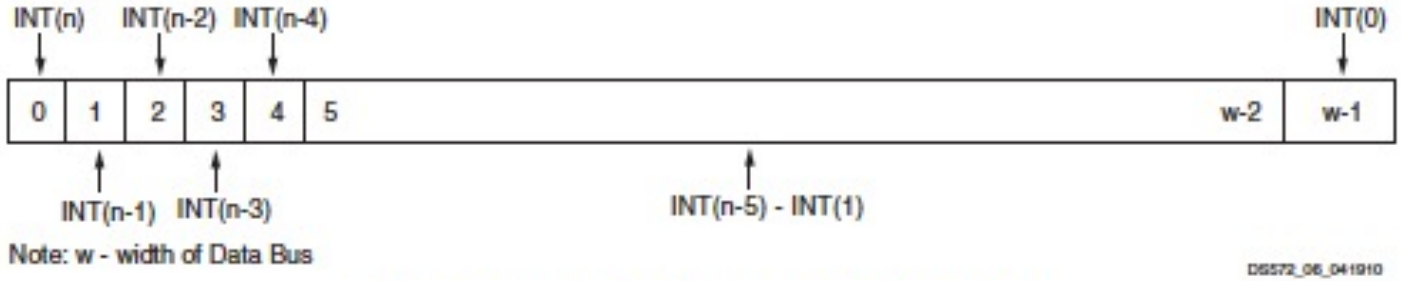


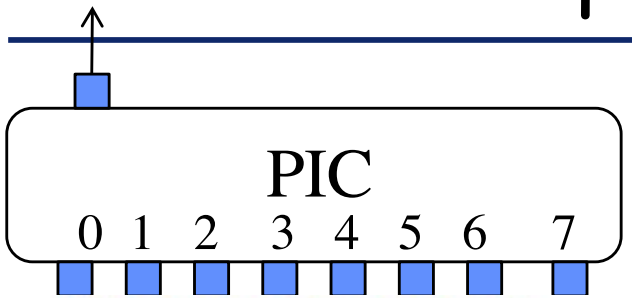
Figure 6: **Interrupt Acknowledge Register (IAR)**

Table 8: **Interrupt Acknowledge Register**

Bits	Name	Core Access	Reset Value	Description
0 : (w[1] – 1)	INT(n) – INT(0) ($n \leq w - 1$)	Write	All Zeros	**Interrupt Input (n) – Interrupt Input (0)** '1' = Clear Interrupt '0' = No action

1. w - Width of Data Bus

IVR: Interrupt Vector Register



Interrupt Vector Register (IVR)

The IVR is a read-only register and contains the ordinal value of the highest priority, enabled, active interrupt input. INT0 (always the LSB) is the highest priority interrupt input and each successive input to the left has a correspondingly lower interrupt priority.

If no interrupt inputs are active then the IVR will contain all ones. The IVR is optional in the XPS INTC and can be parameterized out by setting C_HAS_IVR = 0. The Interrupt Vector Register (IVR) is shown in Figure 9 and described in Table 11.

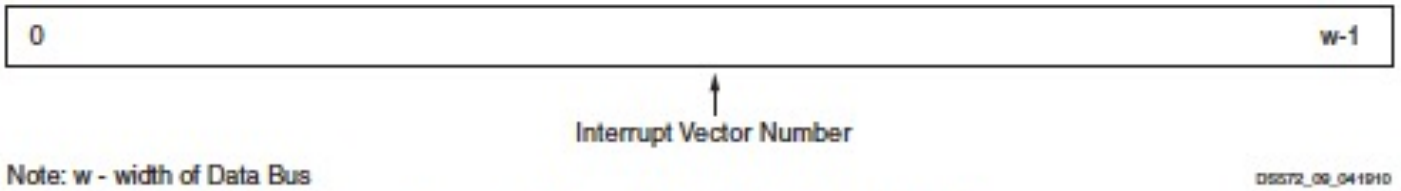


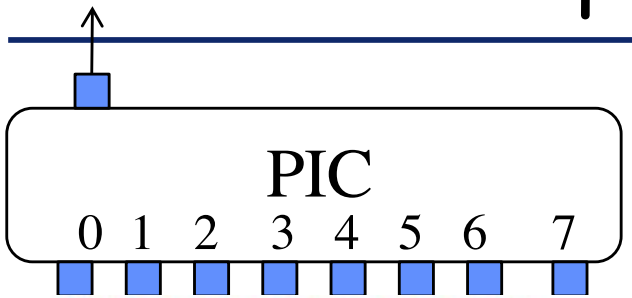
Figure 9: **Interrupt Vector Register (IVR)**

Table 11: **Interrupt Vector Register**

Bits	Name	Core Access	Reset Value	Description
0 : (w[1] – 1)	Interrupt Vector Number	Read	All Ones	**Ordinal of highest priority, enabled, active interrupt input**

1. w - Width of Data Bus

IVR: Interrupt Vector Register



Tells you highest priority pending Interrupt

Interrupt Vector Register (IVR)

The IVR is a read-only register and contains the ordinal value of the highest priority, enabled, active interrupt input. INT0 (always the LSB) is the highest priority interrupt input and each successive input to the left has a correspondingly lower interrupt priority.

If no interrupt inputs are active then the IVR will contain all ones. The IVR is optional in the XPS INTC and can be parameterized out by setting C_HAS_IVR = 0. The Interrupt Vector Register (IVR) is shown in Figure 9 and described in Table 11.

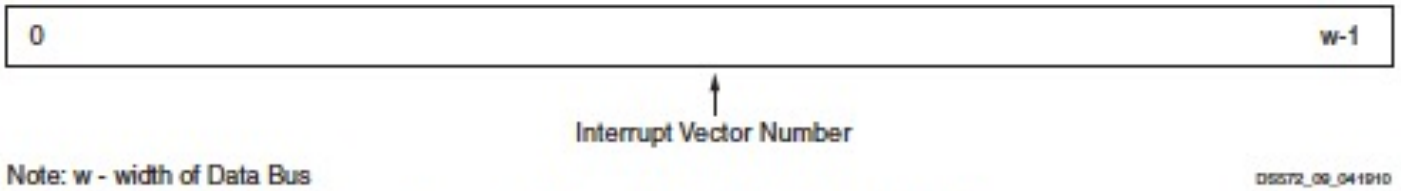


Figure 9: **Interrupt Vector Register (IVR)**

Table 11: **Interrupt Vector Register**

Bits	Name	Core Access	Reset Value	Description
0 : (w(1) – 1)	Interrupt Vector Number	Read	All Ones	**Ordinal of highest priority, enabled, active interrupt input**

1. w - Width of Data Bus

MER: Master Enable Register

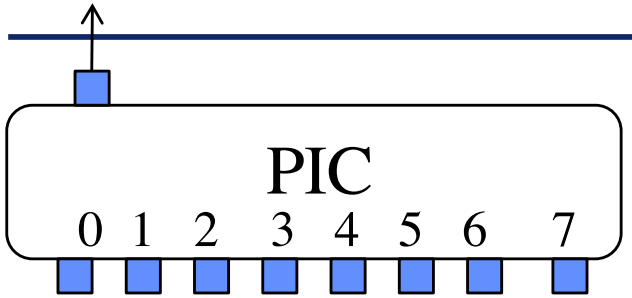


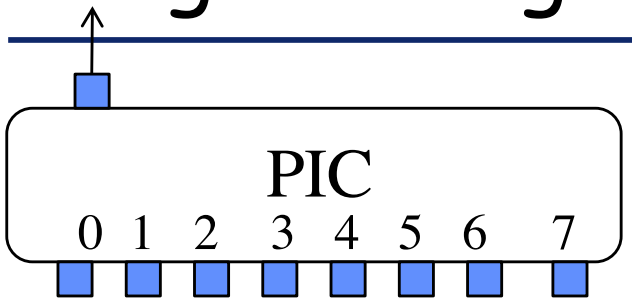
Figure 10: **Master Enable Register (MER)**

Table 12: **Master Enable Register**

Bits	Name	Core Access	Reset Value	Description
0 : (w[1] – 3)	Reserved	N/A	All Zeros	**Reserved**
(w[1] – 2)	HIE	Read / Write	'0'	Hardware Interrupt Enable '0' = Read – SW interrupts enabled Write – No effect '1' = Read – HW interrupts enabled Write – Enable HW interrupts
(w[1] – 1)	ME	Read / Write	'0'	Master IRQ Enable '0' = IRQ disabled – All interrupts disabled '1' = IRQ enabled – All interrupts enabled

1. w - Width of Data Bus

Programming Basics



To Set up:

1)Write to IER to set/clear particular interrupts

2)Write to MER to turn on and wait……

To Respond to Interrupt:

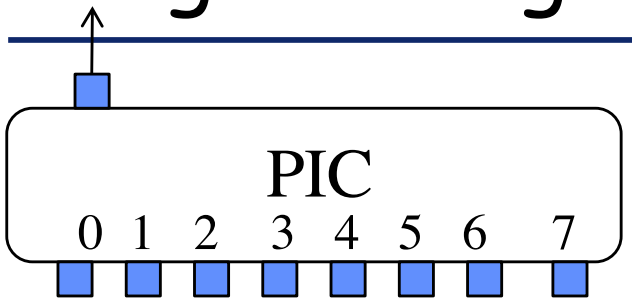
1)Read IVR to find out highest priority interrupt

2)Use (IVR) to branch to appropriate handler routine

-handler routine:

1st handshake device to clear request—why ?

Programming Basics



To Set up:

1)Write to IER to set/clear particular interrupts

2)Write to MER to turn on and wait……

To Respond to Interrupt:

1) Read IVR to find out highest priority interrupt
2) Use (IVR) to branch to appropriate handler routine

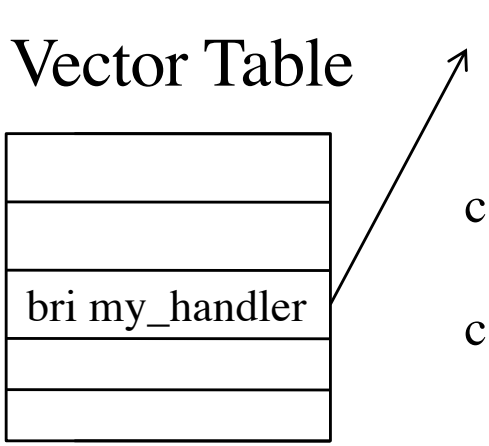
-handler routine:

1st handshake device to clear request

write to IAR register to clear

Execution Flow

o
o
o
addi r1,r1,4
sub r1,r2,r3
add r1,r2,r3
bri loop
o
o
o



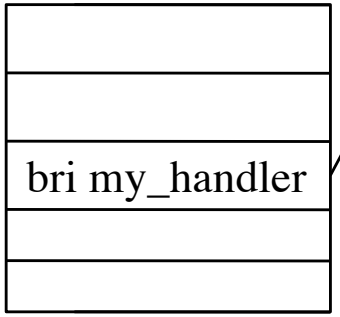

```
My_handler:
   Read IVR
   switch(IVR)
case o: bri isr_routine_0
        break;
case 1: bri isr_routine_1
        break;
        o
        o
        o
   end case:
```

5)

Execution Flow

o
o
o
addi r1,r1,4
sub r1,r2,r3
add r1,r2,r3
bri loop
o
o
o

Vector Table



My_handler:
Read IVR
switch(IVR)
case o: bri isr_routine_0
break;
case 1: bri isr_routine_1
break;
o
o
o
end case:

Isr_routine_x
Handshake device
(to clear rqst to PIC)
Write to IAR
(to clear PIC rqst)
Execute routine
o
o
o
rti r14,8

5)