Exceptions & Interrupts

Exceptions are a generalization of interrupts:

- *Interrupts* are generated by hardware devices
- *Software exceptions* are generated by software instructions

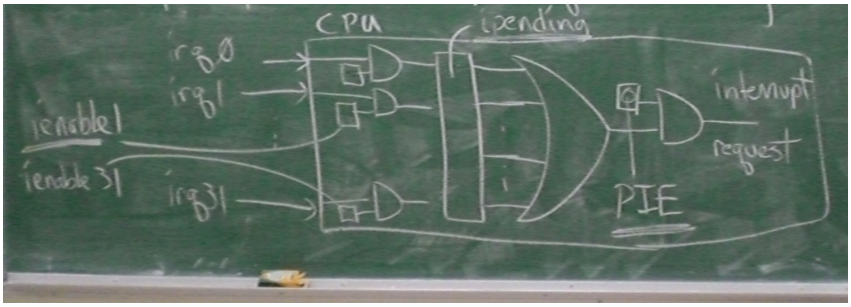
Exceptions refer to both *hardware interrupts* and *software-generated exceptions*.

- Exceptions can occur at any time
- When it occurs, an exception handler is run
- Exception handler will save registers on stack, execute routine, restore registers.

Exception types

• Divide-by-zero (not in Nios) • Illegal memory address (not in Nios) • Unaligned memory address (not in Nios) • Overflow (not in Nios)	*data-dependent exceptions*
• Illegal instruction (not in Nios) • Unimplemented instruction • Trap instruction • Break instruction	*instruction-dependent exceptions*
• Interrupt – a hardware exception triggered when "interrupt request" pin set to 1	

Nios II CPU Interrupt Logic



Registers

Normal ones: r0 to r31

- r31 == ra return address
- r29 == ea exception return address
- r24 == et exception temporary

Special control registers: *ctl0* to *ctl31*

- *ctl0* == *status* bit0 of *status* == *PIE* (processor interrupt enable)
- *ctl1* == *estatus*
- *ctl3* == *ienable*
- *ctl4* == *ipending*

In C, **enableInterrupts();** // sets the PIE bit
disableInterrupts(); // clears the PIE bit

Can only access *ctl0* to *ctl31* using special instructions *rdctl* and *wrctl*, e.g.:

- rdctl r24, *ctl0* = rdctl et, *status* /* read PIE status */
- rdctl r8, *ctl4* = rdctl r8, *ipending* /* read pending interrupts */
- wrctl *ctl0*, r0 = wrctl *status*, *zero* /* disableInterrupts();*/
- wrctl *ctl3*, r13 = wrctl *ienable*, r13 /* enable individual irq */

Nios II Exception Process

When an exception occurs, the CPU *does not execute the current instruction*. Instead it:

1. copy *status* to *estatus*
2. clear PIE bit (to 0) to disable further interruptions
3. modify r29 == ea to hold return address to instruction **after** the one being interrupted (at PC+4)
4. start running *exception handler* at address 0x0000 0020

When done, the *exception handler* must use the **eret** instruction to:

5. Copy *estatus* back to status
6. Jump to address in ea to resume the main program

After finishing an exception, Nios II normally skips past the instruction that caused the exception. This is how it makes forward progress to get past "trap" or unimplemented "multiply" instructions. If it didn't do this, the same instruction would be repeated, triggering another exception and creating an infinite loop.

What happens on return from a hardware interrupt? To which instruction does it resume?

Your Program

Your program must include three major components:

A) exception handler
B) interrupt service routines (ISR)
C) main program setup routines

(note: there may not be enough time to cover the points below in one lecture)

A) Your *exception handler* must:

1. Save registers on the stack
2. Determine the cause of the exception according to the priority order
3. For hardware interrupts, adjust the return address in *ea* by subtracting 4
4. Call the appropriate *interrupt service routine* or *exception service routine*
    - Loop to call the ISR associated for each hardware IRQ in *ipending*
5. Restore registers from the stack
6. Return to the main program using the instruction *eret*

B) Your *interrupt service routine* must:

1. Clear the cause of the interrupt so it will not occur again (eg, tell the device to stop sending the interrupt)
2. Do the appropriate action for the interrupt (eg, read character from serial port)
3. Change the state of the system (ie, modify memory to alter behaviour of system)
4. Return to the exception handler using *ret*

C) Your *main program* must:

1. Place the *exception handler* in memory at address 0x00000020.
2. Enable the use of the stack
3. Specifically enable device to send interrupts (eg: ps2, timer)
4. Specifically enable CPU to receive interrupts from the device (*ienable*)
5. Enable CPU interrupts by setting PIE bit to 1 (i.e. set bit 0 in *status* to a 1)