ECE243

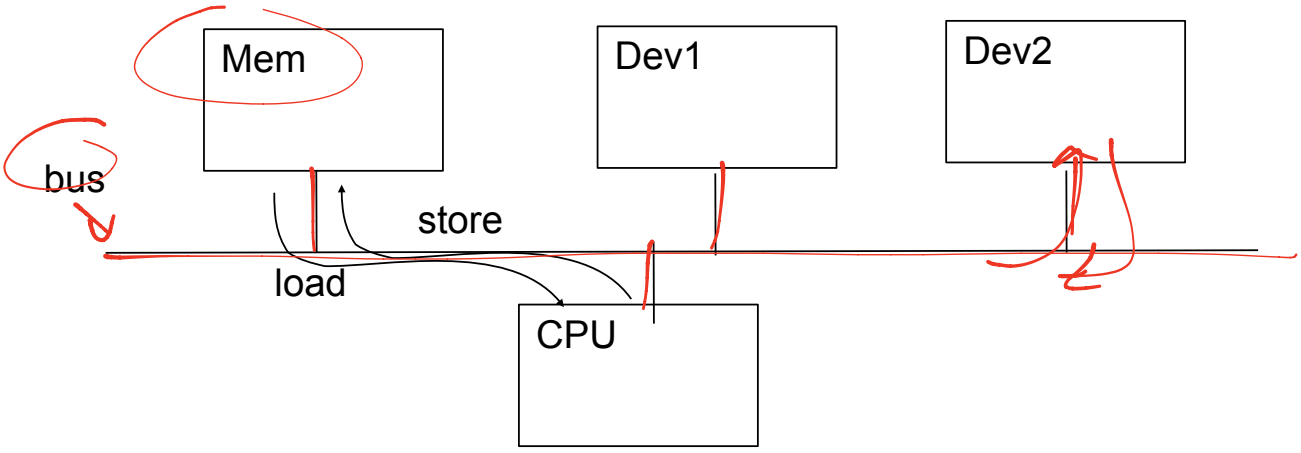
Input/Output (I/O) Software

Prof. Enright Jerger

Memory Mapped Devices

Connecting devices to a CPU

- memory is just a device
- CPU communicates with it
  - through loads and stores (addrs & data)
- memory responds to certain addresses
  - not usually all addresses
- CPU can talk with other devices too
  - using the same method: loads and stores
- devices will also respond to certain addrs
  - addrs reserved for each device



MEMORY MAPPED I/O

- a device:
  - 'sits' on the memory bus
  - watches for certain address(es)
  - responds like memory for those addresses
  - 'real' memory ignores those addresses
- the memory map:
  - map of which devices respond to which addrs

DESL NIOS SYSTEM MEM MAP

0x00000000: 8MB SDRAM (up to 0x007fffff)

0x10001000: JTAG UART

0x10001020: 7 segment display

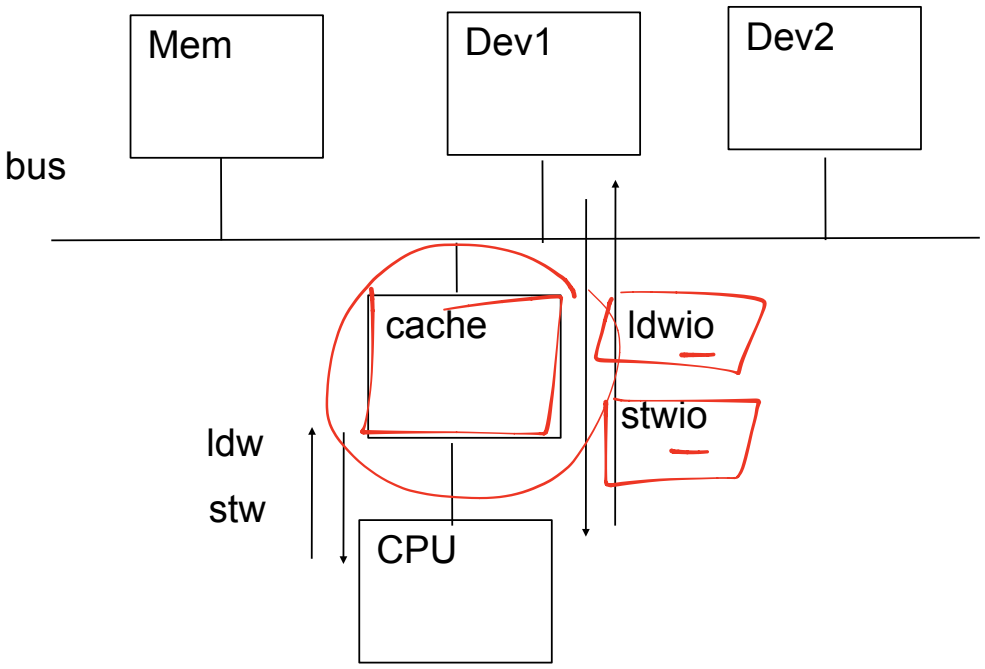
0x10000060: GPIO JP1

0x10000070: GPIO JP2

0x10003050: LCD display

•These are just a few example locations/devices
• see DESL website: NiosII: Reference: Device Address Map for full details

TALKING WITH DEVICES



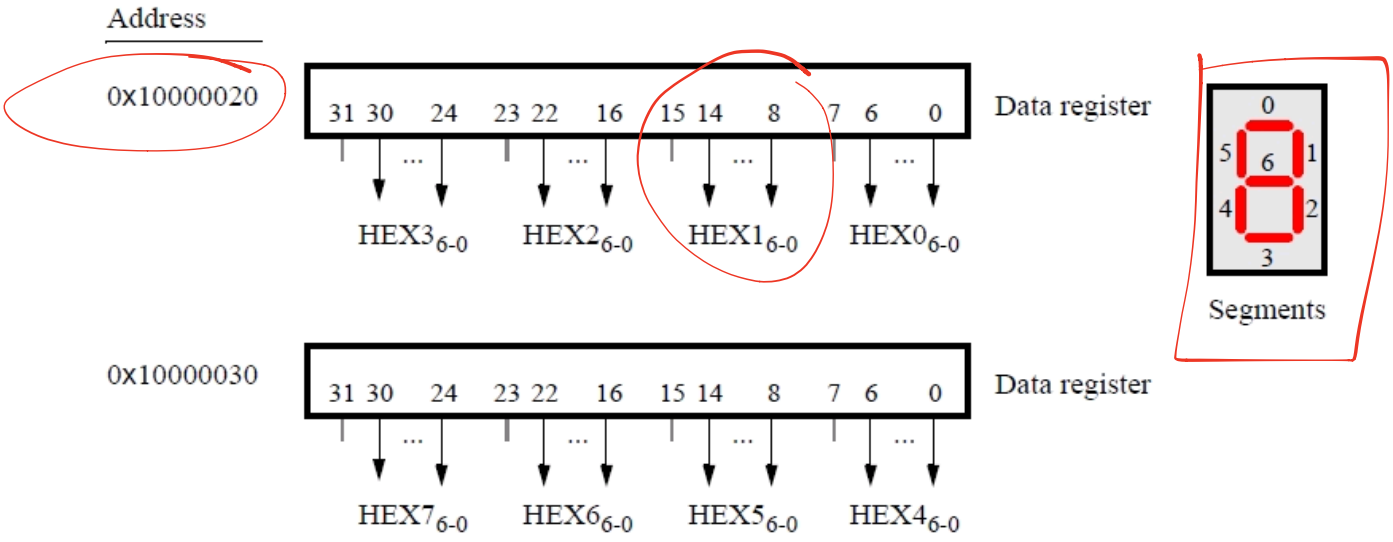
- Note1: use ldwio and stwio
  - to read/write memory mapped device locations
  - io means bypass cache if it exists (more later)
- Note2: use word size even if it is a byte location
  - potentially funny behaviour otherwise

see DESL website: NiosII: Devices for full specs on every device

7-Segment Display

- **base: HEX3-Hex0:** 0x10000020

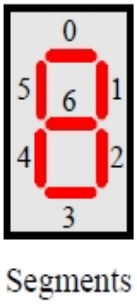
  HEX7-HEX4: 0x10000030
- Controls the individual 'segments' of the hex display
- write only
- handy for debugging,monitoring program status



- **base: HEX3-Hex0:** 0x10000020

  HEX7-HEX4: 0x10000030

Example: write the letter 'F' to 7seg display:



5 | 6
4 |

→ 0b 1110 0001
654 3210

```
.equ 7SEGS_LOWER, 0x1000 0020
movia r8, 7SEGS_LOWER # addr of device
movi r9, 0b1110001 ←
stwio r9, 0(r8)
```

POLLING

- devices often much slower than CPU
  - and run asynchronously with CPU
  - i.e., use a different clock
- how can CPU know when device is ready?
  - must check repeatedly
  - called “polling”
  - asking “are you ready yet? Are you ready yet?…”

TIMER

- like a stopwatch:
  - you set the value to count down from
  - can start/stop, check if done, reset, etc.
- counts at 50MHz

.equ TIMER, 0x10002000

0(TIMER): write zero here to reset the timer;
bit1: 1 if timer is running
bit0: 1 if timer has timed out

4(TIMER): bit3: write 1 to stop timer
bit2: write 1 to start timer
bit1: set to 0 to make timer wait after timeout before continuing

8(TIMER): low 16bits of timeout period

12(TIMER): high 16bits of timeout period

Example: 5-second-wait

- Wait for 5-seconds using timer0
- First: must compute the desired timer period
  - recall: timer runs at 50MHz

50 MHz = 50 million cycles/sec

timer period = 5 sec x 50 MHz
= 5 sec x 50 million cycles/sec
= 250 million cycles
= 0x0EE6 B280
upper lower

```
.equ TIMER, 0x10002000
.equ PERIOD, 0x0EE6B280
movia r8, TIMER

movui r9, %lo(PERIOD) ≠ %lo - macro - assembler
```

```
                                    # extracts bits[15..0]
                                    # movui - move unsigned immed
    stwio r9, 8(r8)   # lower hword of period
    movui r9, %hi(PERIOD)  # %hi . macro assembler
                           # extracts bits[31..16]

    stwio r9, 12(r8)

    stwio r0, 0(r8) # reset timer
    movi r9, 0x6  # 0b110 - bit 2 - start timer
                            bit 1 - timer won't wait
                              after timeout

    stwio r9, 4(r8)

POLL:   ldwio  r9, 0(r8)
        andi   r9, r9, 0x1   # check if timer has timed
        beq    r9, r0, POLL  # loop out & check again
        # 5 sec have elapsed, do action
        stwio r0, 0(r8)  # clear timer
        br POLL # wait another 5 secs
```

INTERFACES

- "serial"
  - means transmit one bit at a time
  - i.e., over a single data wire
- "parallel"
  - means transmit multiple bits at a time
  - over multiple wires

  - more expensive
    - more$$$, wires, pins, hardware
- which is faster?

depends on material, design, protocols etc

GENERAL PURPOSE IO Interfaces

- two parallel interfaces on DE2
  - aka general purpose IO interfaces, GPIO
  - called JP1 and JP2
- each interface has:
  - 32 pins
  - each pin can be configured as input or output
    - individually!
- pins configured as input default to 1
  - called “pull-up”
- pins configured as output default to 0
  - default value of output register

GPIO LOCATIONS

JP1: 0x10000060

JP2: 0x10000070

For each:

0(JPX): DR data in/out (32 bits)

4(JPX): DIR data direction register

each bit configures data pin as in or out (32 bits)

0 means input, 1 means output

Example1

- configure JP1 as all input bits
  - and read a byte

```
.equ JP1, 0x1000 0060
movia r8, JP1
stwio r0, 4(r8) # config as input
ldwio r9, 0(r8) # read word from data pins
```

Example2

- configure JP2 as all output bits
  - and write a character to the lowest 8 bits

```
.equ JP2, 0x1000 0070
movia r8, JP2
movi r9, 0xFFFF # sign extend → r9 = 0xFFFFFFFF
stwio r9, 4(r8) #config as all outputs
movui r9, 'X' # ascii char x
stwio r9, 0(r8) #write to data pins
```

Example3

- configure JP1
  - lower 16bits input, upper 16 bits output, read then write it back

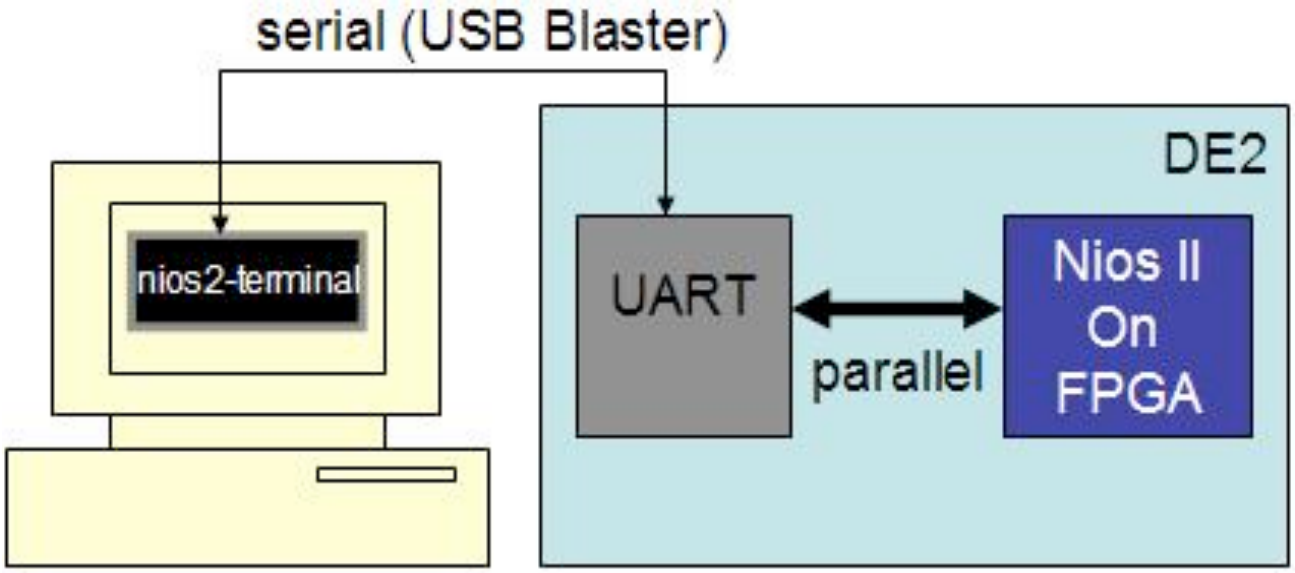
```
.equ JP1, 0x10000060
movia r8, JP1
movia r9, 0xFFFF0000  # config lower 16 - input, upper 16 output

stwio r9, 4(r8)  # write dir reg
ldwio r9, 0(r8)  # read from data pins
andi r9, r9, 0xFFFF  # r9 & 0x0000FFFF - mask lower 16 bits

slli r9, r9, 16    #shift left 16
stwio r9, 0(r8) # device will ignore lower 16 - config as input
```

Serial Interfaces:

- send/recv 1 bit at a time
  - in each direction
- cheap
  - eg., only one data wire, plus a few control wires
- can be very fast

- ex: COM port on a PC, RS-232 is standard
  - Usually a nine pin connector
  - used to be very common in PCs
  - now replaced by USB
  - still very common in embedded systems

JTAG UART



- JTAG: Joint Test Action Group
  - standard interface for test and debug for ICs
  - connects to host PC via USB blaster cable
- UART:
  - Universal Asynchronous Receiver Transmitter
  - serial device
- Asynchronous:
  - data can be sent/rec'd at any time

JTAG UART

.equ JTAG_UART, 0x10001000

0(JTAG_UART): data register: reading gets the next datum

bit15: read valid

bits7-0: data

4(JTAG_UART): control register:

bits31-16: number of character spaces available to write

EXAMPLE: echo

- read a character then send it back

```
.equ JTAG_UART, 0x10001000
movia r8, JTAG_UART

wait_recv: ldwio r9, 0(r8)
andi r10, r9, 0x8000   # check if bit 15 is 1
beq r10, r0, wait_recv # if read invalid try again

andi r9, r9, 0xff      # mask - lowest byte input char

wait_xmit: ldwio r10, 4(r8)
movia r11, 0xFFFF0000
and r10, r10, r11      # mask upper 16-bits
beq r10, r0, wait_xmit # if no space avail try again

stwio r9, 0(r8)
```

NOTE: run "nios2-terminal" in NIOS command window to start a shell

OTHER DE2 MEM-MAPPED DEVICES

- slider switches
- push buttons
- LEDs
- LCD display
- RS232 UART
- audio codec
- VGA adapter
- ps2 connector (mouse)
- Digital protoboard
- see DESL www for full details

ECE243

Interrupts

WHEN IS A DEVICE IS READY

- option1: polling
  - cpu keeps asking until ready
  - can be wasteful
- option2: interrupt
  - cpu interrupted in the middle of what it is doing
  - a.k.a. exception

INTERRUPT BASICS

1. cpu 'configures' for interrupt
   - eg., device, memory, etc.
2. cpu does other work
3. cpu gets interrupted (can happen anytime)
   a) cpu saves its state
   b) cpu jumps to an "interrupt handler"
   c) cpu resumes what it was doing

Example Events handled by interrupts

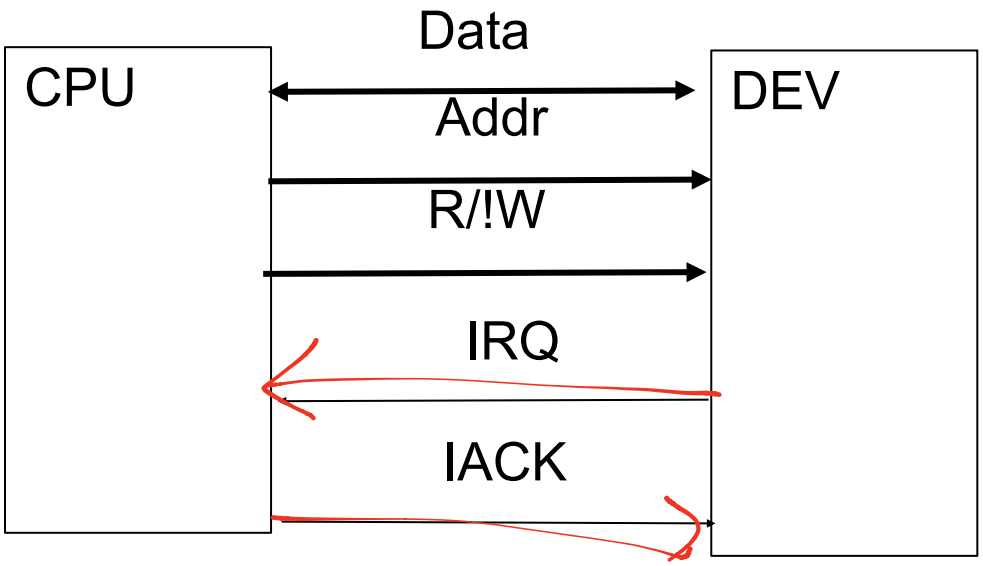
- External Devices (ex UART, USB)
- OS: timers, disk I/O, keyboard I/O
- Debugging: breakpoints
- Program Errors (called "exceptions")

- Divide by zero
- Misaligned memory access
- Memory protection violation (segfault)

POLLING vs INTERRUPTS

	Polling	Interrupt
When?	Explicit polling loop in code	anytime (once setup)
Difficulty	Easy - already seen examples	more complex (next)
Efficiency	good if small wait expected	good for medium to long wait

SIMPLIFIED INTERRUPT HARDWARE



- IRQ: interrupt request line
  - devices asserts to interrupt CPU
- IACK: interrupt acknowledge
  - cpu asserts to acknowledge the device's interrupt

NIOS INTERRUPT SUPPORT

- 32 IRQ lines
  - IRQ0 through IRQ31
- ctlX registers:
  - control registers
  - X=0,1,2,3,4
  - eg., ctl0, ctl3, etc.
- r28 (ea):
  - exception return address
- r24 (et):
  - exception temporary
- 0x1000020:
  - addr (pc) of global interrupt handler code
  - only one for the system
  - eg., all interrupts jump to the same piece of code

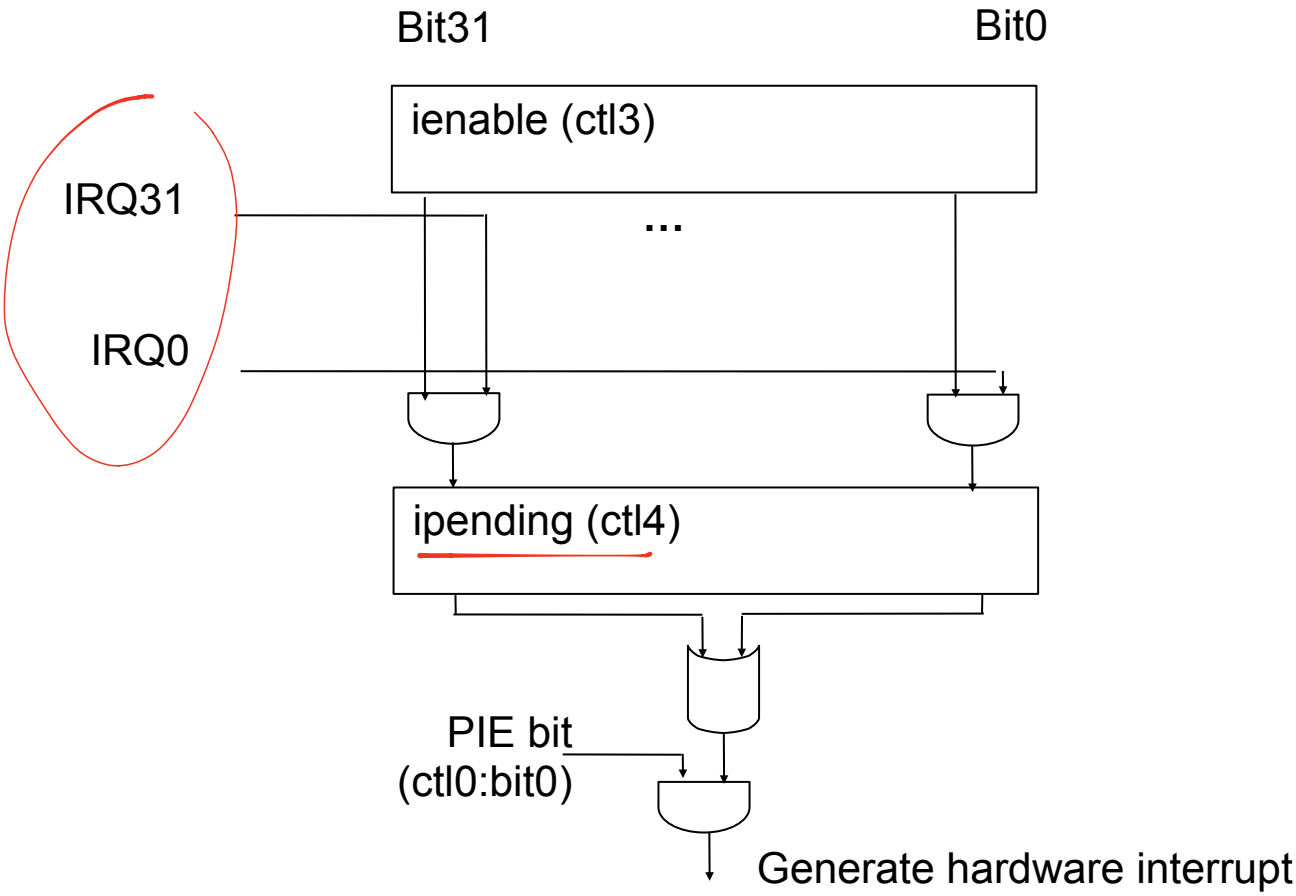
EXAMPLE USE OF IRQ LINES:

- IRQ0: timer
- IRQ1: push buttons
- IRQ6: audio core
- IRQ7: ps/2 controller
- IRQ8: JTAG UART
- IRQ10: RS232 UART
- IRQ11: JP1
- IRQ12: JP2

Relevant CTLX registers:

- rdctl and wrctl
  - special instrs allowed to read/write ctlx registers
- ctl0: status register
  - bit0: PIE: processor interrupt enable bit
  - bit1: U: 1=user mode, 0=supervisor mode
- ctl1: estatus register
  - holds copy of ctl0 when interrupt occurs
- ctl3: ienable register
  - bits 31..0 enable IRQ lines 31..0
- ctl4: ipending register
  - each bit indicates a pending interrupt on IRQ lines

NIOS Interrupt Hardware



CONFIGURING AN INTERRUPT

1. configure device (various setup and enabling)
2. enable IRQ line (appropriate bit of ctl3)
3. enable external interrupts (PIE bit = bit 0 of ctl0)

Example: enable IRQ line 5:

```
movi r8, 0x20 # 0b0010 0000 not 0x5
```

wrctl ctl3, r8 # set ienable to 0x20

3 **Example: enable external interrupts:**

```
movi r8, 1
wrctl ctl0, r8  # set PIE bit to 1
```

ON A HARDWARE INTERRUPT:

1. current instr is aborted
2. addr of next instr is written to ea
3. ctl0 copied to ctl1
4. ctl0:PIE set to 0
   - further interrupts are disabled
5. pc set to 0x1000020
   - addr of global interrupt handler

PLACING AN INTERRUPT HANDLER IN .s FILE

```
.section .exceptions, "ax" # means
                           allocatable, executable
IHANDLER:
... # ihandler code goes here
```

eret # special return from exception instr

copies ctl1 to ctl0

PC = ea

.section .text

.global main

main: ...

MEMORY LAYOUT

Addr	Value
0x1000000	.section .reset
…	…
0x1000020	.section .exceptions
…	…
…	.section .text
…	…
…	.section .data
…	…

advanced

interrupt handler

GENERIC INTERRUPT HANDLER

```
.section .exceptions, "ax"
IHANDLER:
        rdctl et, ctl4  # ipending
        andi et, et, 0x1  # check if interrupt
                          pending for IRQ0
        beq et, r0, EXIT_HANDLER  # if not
                                    exit
        -> # code to handle interrupt from IRQ0
           # code to ack interrupt from IRQ0
EXIT_HANDLER:
        subi ea, ea, 4  # replay instr that
                          got interrupted
        eret
```

TIMER INTERRUPT SUPPORT

- TIMER: 0x10002000
  - 0(TIMER): write zero here to reset timer
    - and/or acknowledge an interrupt
  - 4(TIMER): bit0: write 1 to enable interrupts

- Recall: 5-second delay:

  timer period = 0x0EE6 B280 ←

  upper lower

Ex: TIMER WITH INTERRUPTS

```
.equ TIMER, 0x10002000
.equ PERIOD, 0x0EE6B280

.section .text
.global main

main:
        movia r8, TIMER
        # configure device
        movui r9, %lo(PERIOD)
        stwio r9, 8(r8)
        movui r9, %hi(PERIOD)
        stwio r9, 12(r8)
        stwio r0, 0(r8)
        movi r9, 0b111  # start, cont., enable
                        interrupt
        stwio r9, 4(r8)
           # enable IRQ line
        movi r9, 0b1 # IRQ0 for timer
        wrctl ctl3, r9 # ctl3 - ienable
         # enable external interrupts
        movi r9, 0b1
```

```
        wrctl ctl0, r9   # set PIE = 1
LOOP:   br LOOP          # infinite loop
        ret

        .section .exceptions, "ax"
IHANDLER: rdctl et, ctl4  # ctl4 - ipending
        andi et, et, 0x1  # check if interrupt
                          pending for IRQ0 (ctl4:bit0)

        beq et, r0, EXIT_HANDLER  # if not, exit
        # 5 secs passed, do desired action
        # note must save/restore any reg used
        # other than et
        movia et, TIMER
        stwio r0, 0(et)   # ack interrupt
EXIT_HANDLER: subi ea, ea, 4
        eret
```

OTHER DEVICES THAT SUPPORT INTERRUPTS

- push buttons
- rs232 uart
- JTAG UART
- audio core

- ps/2 (mouse)
- DMA
- GPIO (JP1, JP2)

  see DESL:NIOSII:devices for details

MULTIPLE INTERRUPTS

- **HOW DO WE**:
  - specify relative importance of interrupts
    - eg., priority
  - identify each interrupt
    - eg., which device did it come from?
  - allow nested interrupts
    - eg., allow an interrupt handler to itself be interrupted

Ex: GENERIC MULTIPLE INTERRUPT HANDLER

- Goal:
  - handle IRQ 0 (timer) with higher priority than IRQ11 (JP1)
- allow handler for IRQ11 to be itself interrupted
  - eg., nested interrupt

trick: must save regs: ea, et, ctl1

Ex: Multiple Interrupt Handler

```
.section .exceptions, "ax"
IHANDLER:
        subi sp, sp, 12 # save ea, et, ctl1
        stw et, 0(sp)
        rdctl et, ctl1
        stw et, 4(sp)
        stw ea, 8(sp)
        rdctl et, ctl4
        andi et, et, 0x1 # check if interrupt
                         pending IRQ0 - highest priority
        bne et, r0, HANDLE_TIMER
        rdctl et, ctl4
        andi et, et, 0x800 # check IRQ11
                           ctl4: bit 11
        bne et, r0, HANDLE-JP1
        br EXIT_IHANDLER

HANDLE_TIMER: # do action to handle timer
        # ack timer
        br EXIT_IHANDLER

HANDLE_JP1: movi et, 0x1
        wrctl ctl0, et # re-enable interrupts
        # do actions to handle JP1
```

```
...
# ack JP1

EXIT_1HANDLER:
        ldw et, 4(sp)
        wrctl ctl1, et
        ldw et, 0(sp)
        ldw ea, 8(sp)
        addi sp, sp, 12
        subi ea, ea, 4
        eret
```

FASTER DEVICE IDENTIFICATION

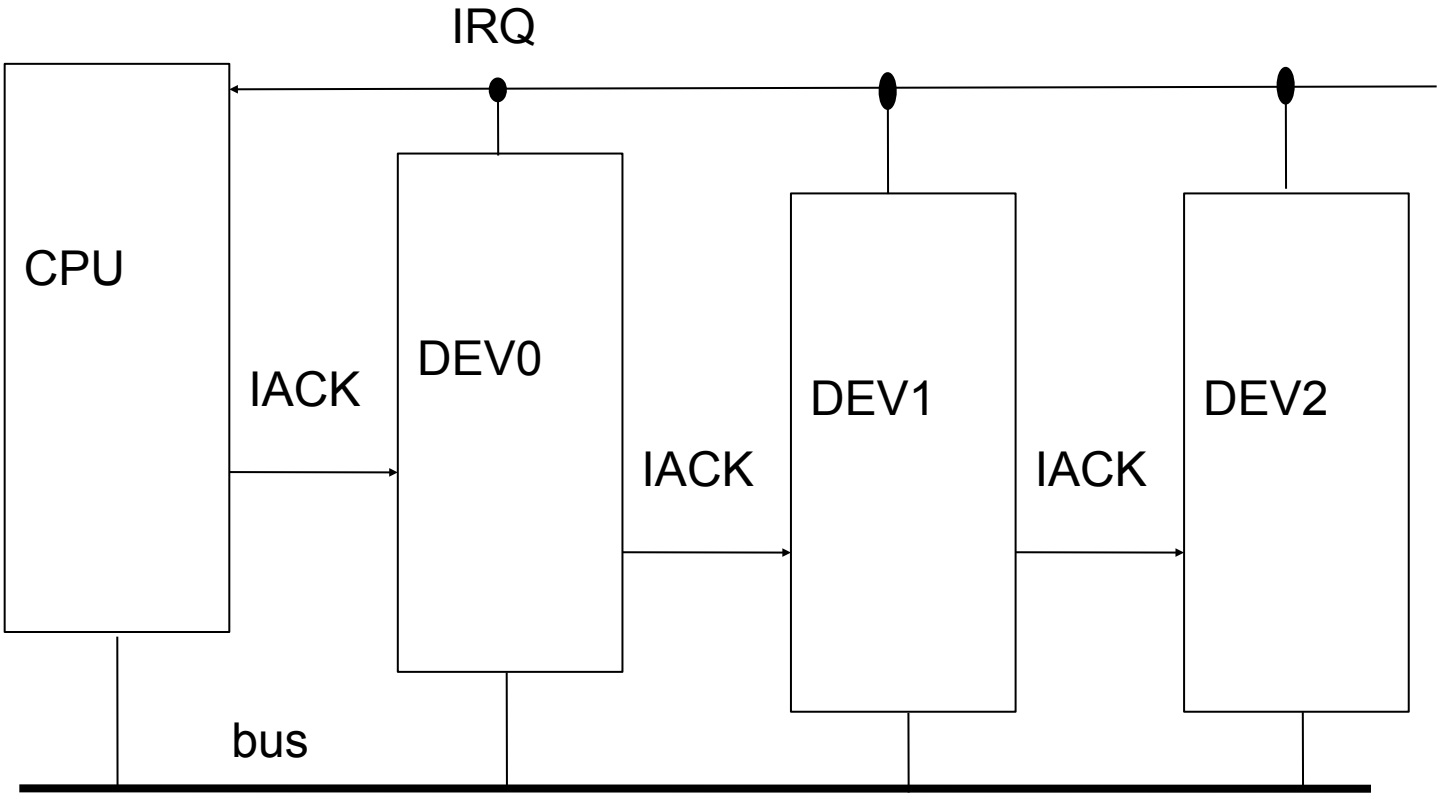
- having a handler for every interrupt can be slow

  - worst case might check all devices
    - before finding interrupting device
  - what if there are 31 devices?
- faster:
  - have hardware/cpu automatically ack device
    - when interrupt occurs
  - allow device to write an identifier on the data bus
    - when interrupt is ack'ed
  - can use the identifier to lookup a specific handler
    - for that device

EXAMPLE: 68000 interrupts

- when interrupt ack'ed device
  - puts a 'vector number' on the databus
- CPU automatically uses vector number
  - to compute an address in memory:
  - addr = vector_number * 4
- at that addr is stored
  - the addr of the start of that device's handler
- range of memory that vector numbers map to
  - is called the interrupt vector table (IVT)
  - a table of device-specific handler addresses

IF YOU RUN OUT OF IRQ LINES?



- SOLUTION: DAISY CHAIN:
  - have multiple devices share an IRQ line
  - IRQ: any dev can assert, don't know which did it
  - IACK: pass along if not used by current device
  - Identification: device puts vector number on data bus when ack'ed
  - Priority: fixed: eg: left to right