NIOS Processor I/O

Engineering 304 Lab

Outline

- Register Masking
  - Setting specific bits
  - Clearing specific bits
  - Testing specific bits
- Types of I/O systems
- PIO Core Registers
- Internal Timer Registers

Masking of Registers

- A common task when working with I/O registers is to need to either set or clear individual bits in a 32-bit value
- Use bitwise logical operators (AND, OR, XOR, NOT) as needed

  Register <= DATA logical_operator PATTERN

  – DATA would be found in a register
  – PATTERN can be either in a register or provided as an immediate (16 bit only)

Masking Examples for Setting and Clearing Bits

- Example: Force only bit 3 of an 8-bit register high (set):
  - Initial Reg value (R8) = 0b00110001
  - Execute an OR instruction: `ori r8, r8, 0b00001000`
  - Resulting value in R8 = 0b0011**1**001 (notice bit position 3 is now set, the rest are unchanged)
- Example: Force only bit 6 of an 8-bit register low (clear):
  - Initial Reg value (R9) = 0b01111001
  - Execute AND instruction: `andi r9, r9, 0b10111111`
  - Resulting value in R9 = 0b0**0**111001 (notice bit position 6 is now cleared, the rest are unchanged)
- Note: both mask patterns could also have been in registers

Testing States of Bits Using Masks

- To see if bit 3 in Register 8 (r8) is high, mask out the other bits by forcing them to 0 and see if the resulting register is all zero or not
- Example:
  - `andi r9, r8, 0x08  /* same as 0b0000_1000 */`
    - The only bit in r9 that now could possibly be non-zero is bit 3
    - So, check if R9=0 or not
  - `beq r9, r0, mylabel` or `bne r9, r0, mylabel`
- Similar operation would test for a bit being 0 instead of 1

Setting Up Masks in Assembly

- Use the ".equ" compiler directive at the beginning of your assembly file
  - Example:

```
.equ Bit3Mask, 0x08
```

  - Creates a label called "Bit3Mask" that will be replaced with the number 0x08 when the assembler assembles the file
  - Example:

```
andi r9, r8, Bit3Mask
/* is the same as "andi r9, r8, 0b0000_1000" */
```

  - Easier to read and maintain!!

Input and Output – Two Options

- Input and output devices are all accessed via addresses and data – similar to how global/stack memory is accessed
- Option A: Separate memory and I/O address spaces (Intel)
  - Uses special “input” and “output” instructions
- Option B: Unified address space (Motorola, NIOS)
  - I/O units are assigned sections of the memory space and treated as just another kind of memory device
  - Use standard load- and store-like instructions
  - Note: LDWIO and STWIO are simply non-caching LDW and STW and should be used for all I/O accesses

I/O Device Registers

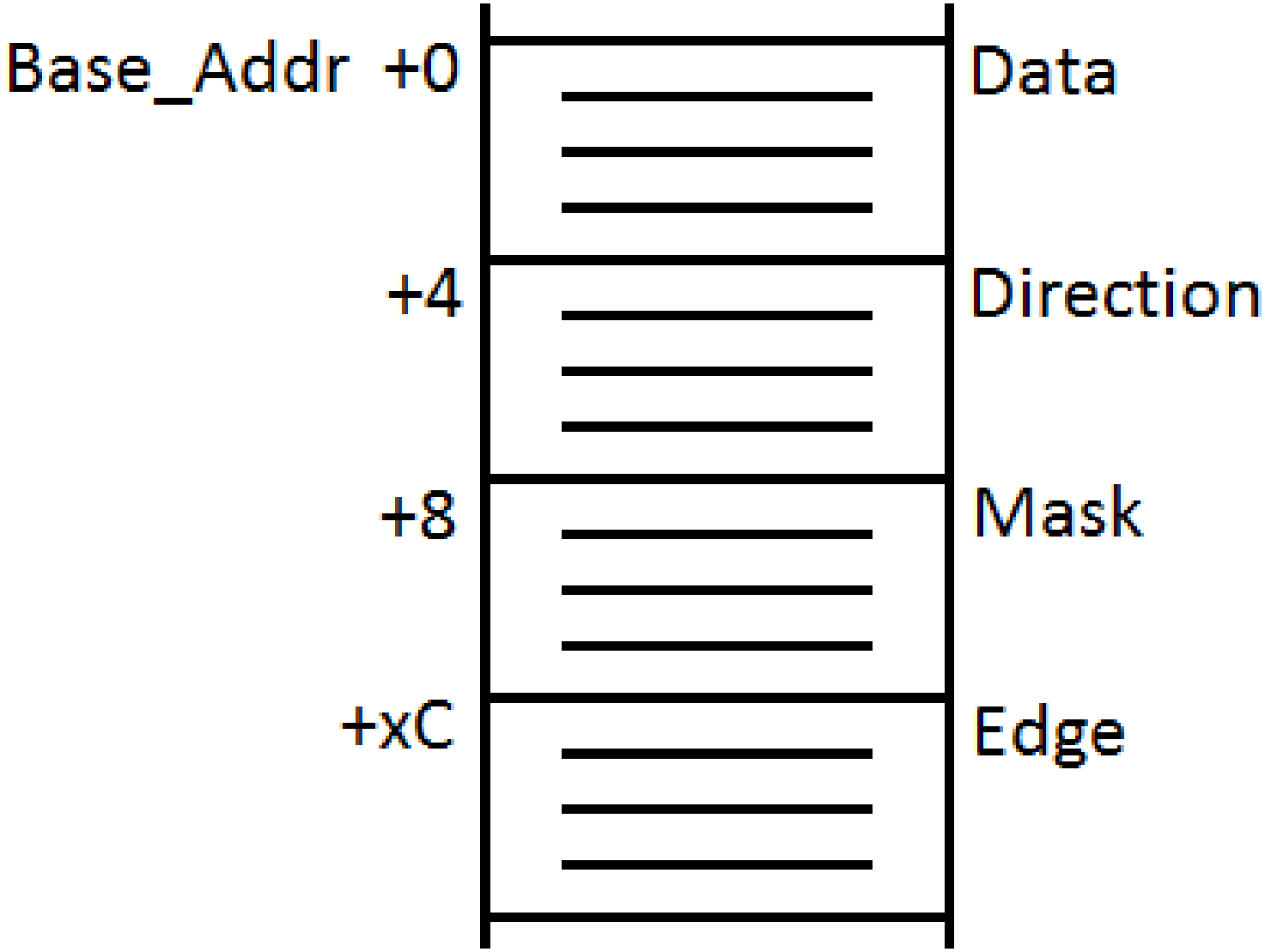
- All communication with I/O devices is done through the special "registers" (memory locations) defined for each type of device
  - Simply write (stwio) or read (ldwio) to/from addresses in the memory space where those "registers" are located
- Look at two example I/O devices
  - General I/O device (PIO core)
  - Timer device (interval timer core)
- Note: Offsets listed are considered word (4-byte) offsets
- Effective address = Port Base_Address + 4*offset

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.				

View of the PIO Port in Memory



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])						

Reading and Writing I/O Registers

- Recommended approach:
  - Pre-define base addresses for each I/O device
    - E.g. timer's base address=0x10800: `.equ TIMER_BASE, 0x10800`
  - Pre-define the word offset amounts (in bytes) for each I/O register
    - E.g. timer's period high register: offset=3*4=12=0xc `.equ PERIODH, 0xC`
  - Load the base address of the I/O device into a register (e.g. gp or r16)

```
movia r16, TIMER_BASE
```

  - Use ldwio and stwio instructions that refer to the base address register and the defined I/O register offset

```
stwio r8, PERIODH(r16)
ldwio r17, DATA(r16)
```

Example Assembly Code

```
/* This code reads the state of the LEDR port,
   sets to one the bit for LED3 (leaving others
   untouched), and then writes that pattern back
   to the port to see the change happen. */
.equ Bit3Mask, 0x08 /* or 0b0000_1000 */
.equ LEDR_BASE_ADDR, 0x08040
.equ DATA, 0x0  /* byte offset for DATA reg in port */
movia gp, LEDR_BASE_ADDR
movi r16, Bit3Mask
ldwio r17, DATA(gp) /* read the port */
ori r17, r17, r16  /* set bit3 */
stwio r17, DATA(gp) /* write the pattern to the port */
```