The Intel Nios II Soft Core Processor

Today's Agenda

- The Embedded "Hello, World!", the Blinking LED or how to access memory mapped registers in C...

FPGA-based SoC Design

International Master of Science in Electrical Engineering

Prof. Dr. C. Jakob
University of Applied Sciences Darmstadt
h_da
Faculty of Electrical Engineering and Information Technology
fbeit

The Intel Nios II Soft-Core Processor - "Hello, World!"

International Master of Science in Electrical Engineering

Prof. Dr. C. Jakob

University of Applied Sciences Darmstadt

h_da

Faculty of Electrical Engineering and Information Technology

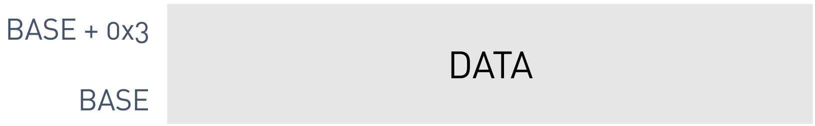
fbeit

"Hello, World!"

- The Embedded **"Hello, World!", ** the Blinking LED or how to access memory mapped registers in C…
- We will see that many ways lead to Rome …

"Hello, World!"

- Flashing an LED connected to the MAX10 FPGA using the Nios II PIO (parallel input/output) peripheral core is fairly straightforward.
- If the PIO core hardware is configured to output-only mode at design time, there is only one relevant register



BASE + 0x3

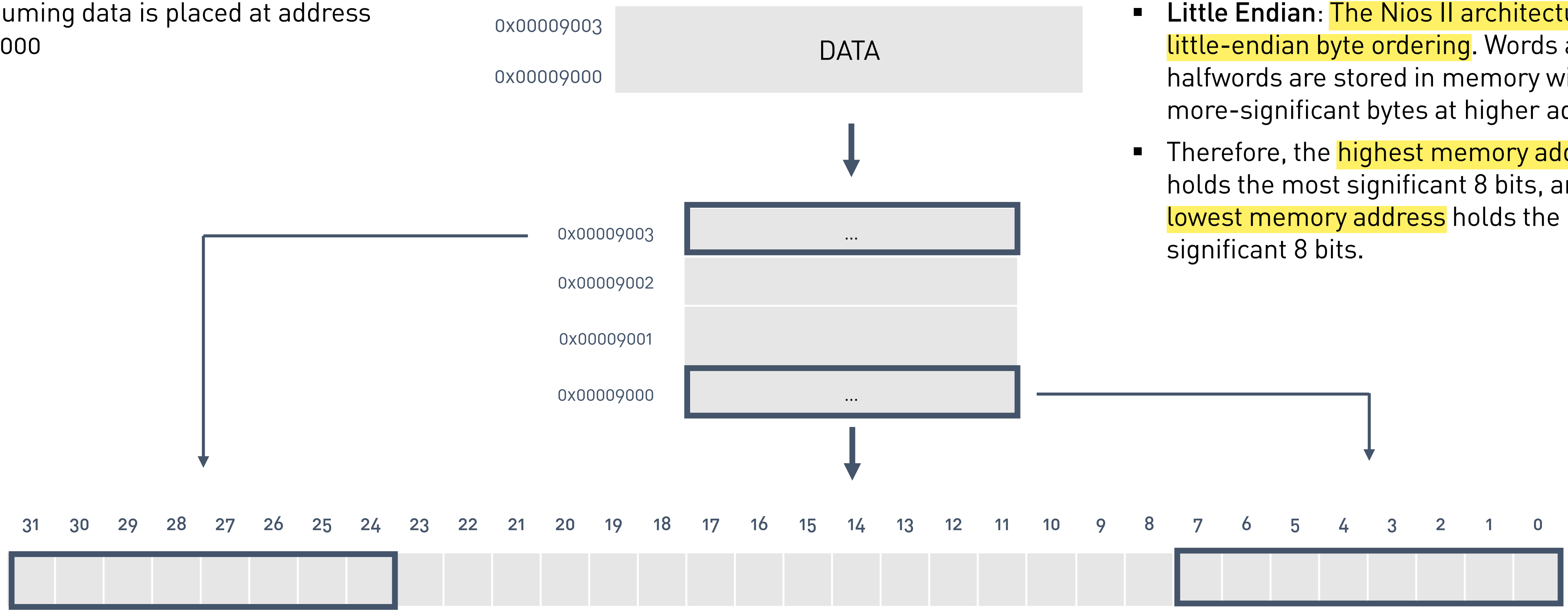
DATA

BASE

- Writing to data stores the value to a register that drives the output ports.
- For further information about the PIO core, please check chapter 27, page 307, of the Intel Embedded Peripherals IP User Guide.

"Hello, World!"

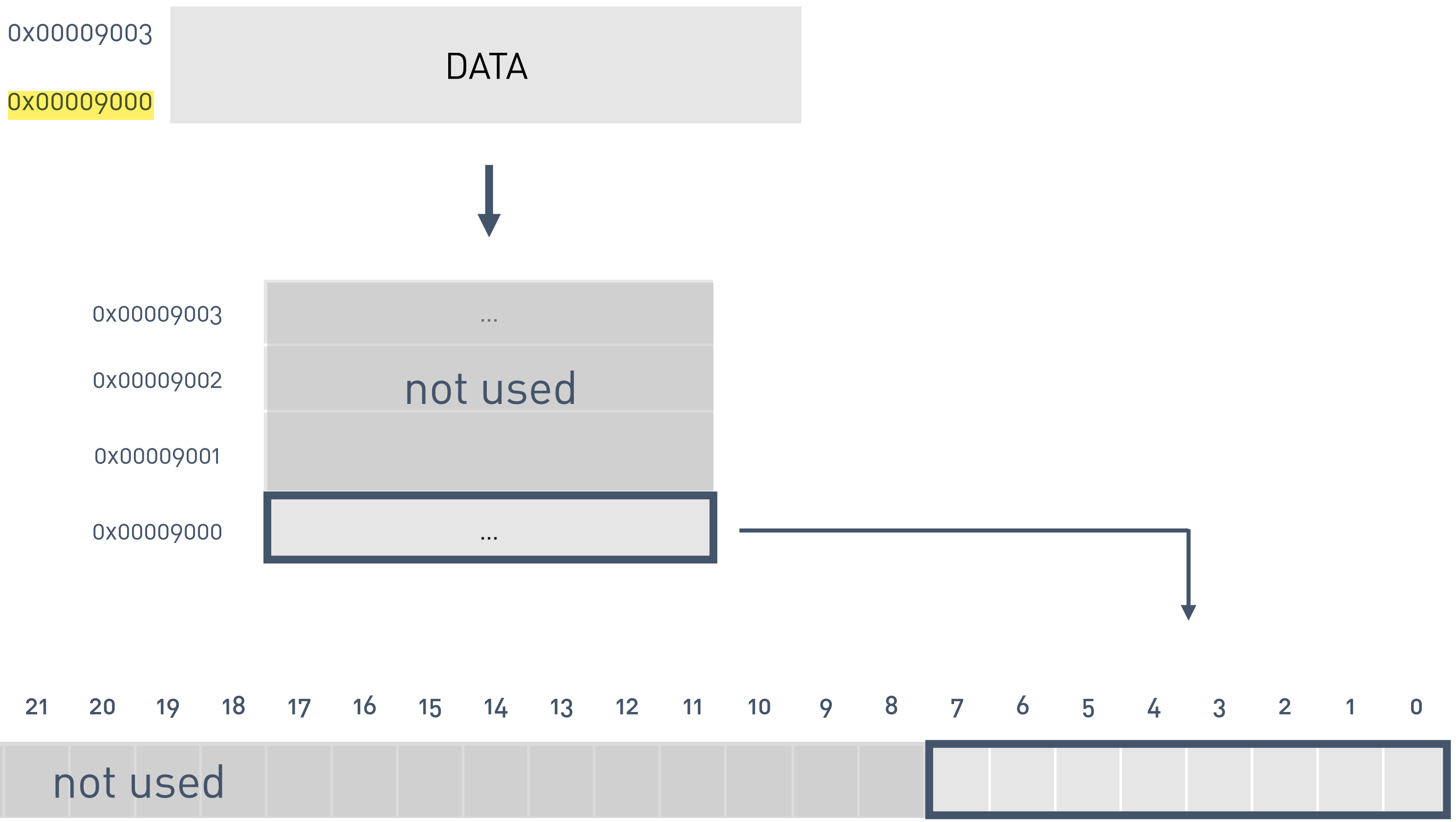
- Assuming data is placed at address 0x9000



- **Little Endian**: The Nios II architecture uses little-endian byte ordering. Words and halfwords are stored in memory with the more-significant bytes at higher addresses.
- Therefore, the highest memory address holds the most significant 8 bits, and the lowest memory address holds the least significant 8 bits.

"Hello, World!"

- We assume that the PIO core is placed at address 0x9000.
- Moreover, we assume an 8-bit PIO output-only configuration with eight outputs driving eight LEDs connected to the MAX10 FPGA.



Memory Mapped I/O

Peripheral Register Access in C

"Hello, World!"

- The Nios II architecture provides **memory-mapped I/O** access. Both data memory and peripherals are mapped into the **address space of the data master port**: Check the Nios II controller design used in the second lab.
- The most common interface between a processor and an I/O peripheral core represents a collection of registers.
  - Data Registers
  - Configuration Registers
  - Status Registers
- The processor treats these registers as memory locations and reads and writes data accordingly.

"Hello, World!"

1/1

```c
volatile unsigned int* const PIO_DATA_REG = (volatile unsigned int *) 0x80009000;

```

[C] Source Code Excerpt: PIO data register access

Notes

- Typically when we access registers in C based on memory-mapped IO we use a pointer notation so that the compiler generates the correct load/store operations at the absolute address needed.
- The **volatile** qualifier tells the compiler that the value of the variable may change at any time, without any action being taken by the code the compiler finds nearby. Generally the compiler will not keep such variables in registers and will re-read them from memory whenever they are used. It will also store new values to memory whenever told to.

IMP
- The pointer itself is constant, so that it cannot be changed to point to some other address. Please be aware that such a pointer must be initialized at declaration time.

- **Review**: A **pointer** is a variable whose value is the address of another variable, i.e., direct address of the memory location.

"Hello, World!"

1/1

```
volatile unsigned int* const PIO_DATA_REG = (volatile unsigned int *) 0x80009000;

...
*PIO_DATA_REG = 0xAA;    → dereferenced pointer!
```

[C] Source Code Excerpt: PIO data register access

Notes

- The memory-mapped register is accessed for reading and writing by dereferencing the pointer `PIO_DATA_REG` whose value is the register's address.
- Any operation applied to the dereferenced pointer will directly affect the value of the register that it points to.

"Hello, World!"

address bit 31 is set ...

1/1

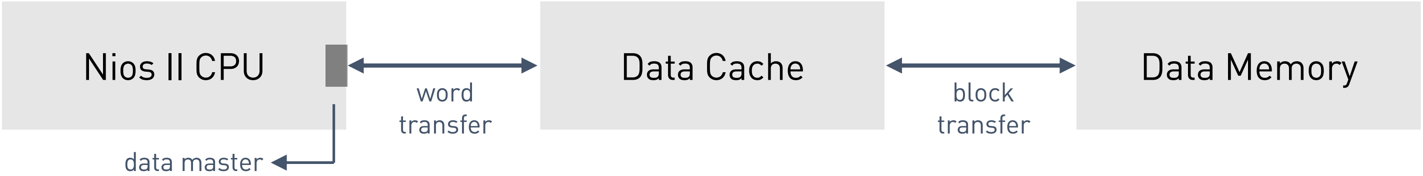
```c
volatile unsigned int* const PIO_DATA_REG = (volatile unsigned int *) 0x80009000;

```

[C] Source Code Excerpt: PIO data register access

Notes

- The Nios II CPU Core designed in the second FSoC lab comes with a data cache that must be bypassed when a accessing peripheral devices.
- **Data cache bypass**: The **bit-31 cache bypass method** on the data master port uses bit 31 of the address as a tag that indicates whether the processor should transfer data to/from cache, or bypass it.
- This is a convenience for software, which might need to cache certain addresses and bypass others. Software can pass addresses as parameters between functions, without having to specify any further information about whether the addressed data is cached or not.
- More methods to bypass the data cache are discussed in future lectures.



"Hello, World!" – Version #1

Peripheral Register Access in C

"Hello, World!“ – Version #1

1/1

```c
volatile unsigned int* const PIO_DATA_REG = (volatile unsigned int *) 0x80009000

volatile unsigned long delay;

int main(void){

    while(1){
        *PIO_DATA_REG ^= 0xFF
        for(delay = 0; delay < 75000; delay++);
    }
    return 0;
}
```

→ poor man's delay

[C] Source Code Excerpt: Nios II "Hello, World!“ – Version #1

IMP

Notes

- In order to perceive the LED flashing, we must reduce the toogle frequency. This is accomplished by delaying the toogle operation itself.
- The delay in this code example is implemented using a **for-loop with an empty loop body**. This is a simple method to actively waste CPU power ...
- In order to prevent the compiler from optimizing this statement, we must use the **volatile** qualifier in the variable declaration.

"Hello, World!" – Version #1

1/1

```c
volatile unsigned int* const PIO_DATA_REG = (volatile unsigned int *) 0x80009000

volatile unsigned long delay;

int main(void){

    while(1){
        *PIO_DATA_REG ^= 0xFF
        for(delay = 0; delay < 75000; delay++);
    }
    return 0;
}
```

→ poor man's delay

[C] Source Code Excerpt: Nios II "Hello, World!" – Version #1

Notes

- Intel Nios II – **Application Binary Interface/Data** C/C++ Types: `unsigned int` – 32bit quantity
- The ABI describes how data is arranged in memory, the behaviour and structure of the stack, as well as the function calling conventions.

"Hello, World!" – Version #2

Peripheral Register Access in C

"Hello, World!" – Version #2

bsp(project) directory - hal - inc - alt_types.h

1/1

```c
#include "alt_types.h"

volatile alt_u32 * const PIO_DATA_REG = (volatile alt_u32 *) 0x80009000

volatile alt_u32 delay;

int main(void){

    while(1){
        *PIO_DATA_REG ^= 0xFF
        for(delay = 0; delay < 75000; delay++);
    }
    return 0;
}
```

[C] Source Code Excerpt: Nios II "Hello, World!" – Version #2

Notes

- In order to highlight that we are accessing 32-bit device registers, we rewrite the former expressions using the `typedefs` provided in the `alt_types` header file.

"Hello, World!“ – Version #3

Peripheral Register Access in C

"Hello, World!“ – Version #3

1/1

```
#include "alt_types.h"

#define PIO_DATA_REG  (*((volatile alt_u32 *)  0x80009000))
#define        LED          (* ( ( volatile unsigned int * )      0x80011030 ) )
volatile alt_u32 delay;

int main(void){

    while(1){
        PIO_DATA_REG ^= 0xFF
        for(delay = 0; delay < 75000; delay++);
    }
    return 0;
}
```

[C] Source Code Excerpt: Nios II "Hello, World!“ – Version #3

Notes

- The **#define statement** defines `PIO_DATA_REG` as a dereferenced pointer. The pointer has been constructed by a casted constant that is identical to the address of the register.

Summary so far ...

Peripheral Register Access in C

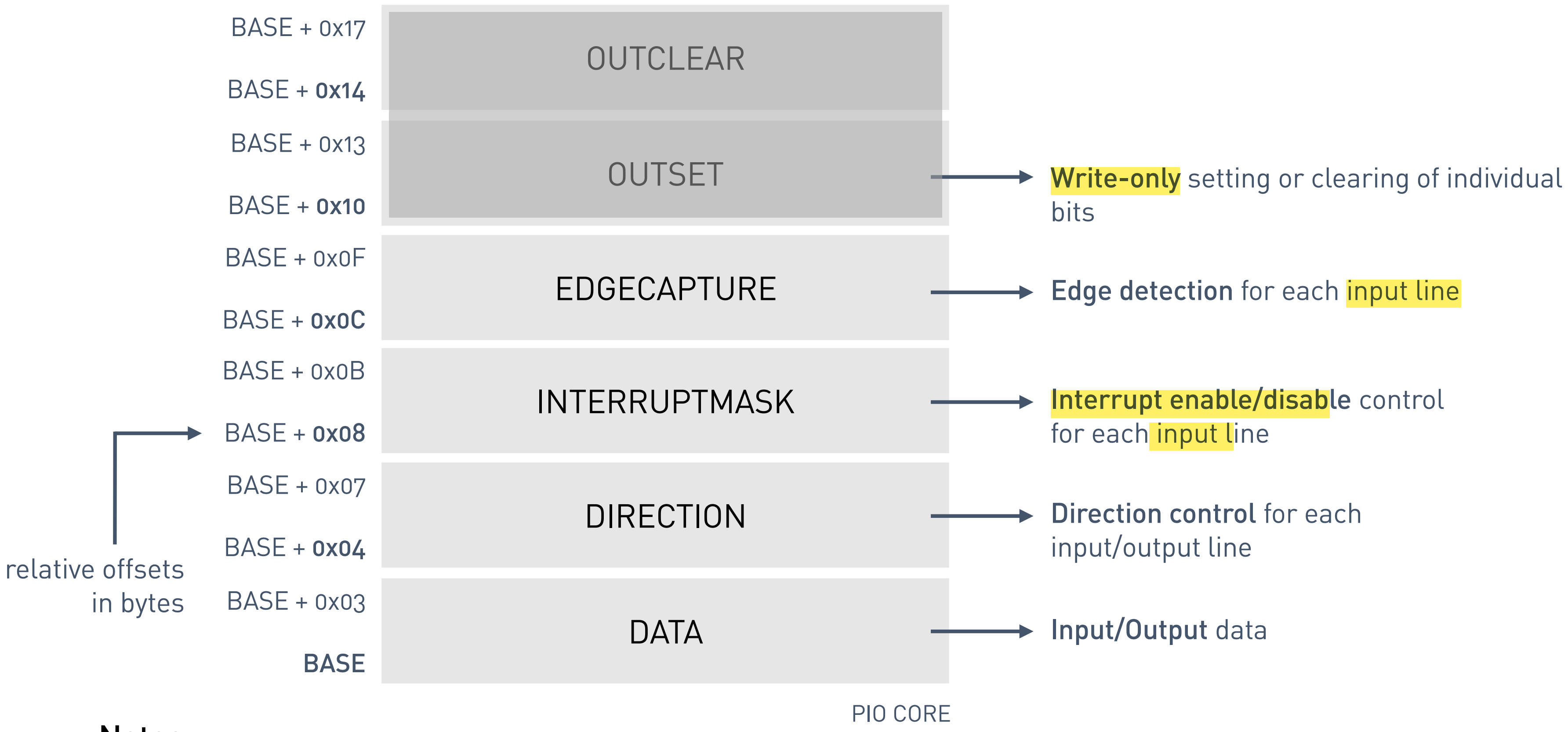
Summary so far ...

- The code so far works just fine, but has a number of shortcomings.
- First, to support multiple IO ports we would have to define a set of pointers for each set of registers ...
- Considering the port actually has 6 different registers we may want to access, this involves a lot of repetition.
- In addition, and more significantly, we can see that the register map of a peripheral has a well defined memory layout

"Hello, World!“ – Version #4

Peripheral Register Access in C

"Hello, World!"



BASE + 0x17
BASE + **0x14**
OUTCLEAR

BASE + 0x13
BASE + **0x10**
OUTSET → **Write-only** setting or clearing of individual bits

BASE + 0x0F
BASE + **0x0C**
EDGECAPTURE → **Edge detection** for each input line

BASE + 0x0B
BASE + **0x08**
INTERRUPTMASK → **Interrupt enable/disable** control for each input line

BASE + 0x07
BASE + **0x04**
DIRECTION → **Direction control** for each input/output line

BASE + 0x03
BASE
DATA → **Input/Output** data

relative offsets in bytes

PIO CORE

Notes

- A more detailed discussion follows within the next lectures.

"Hello, World!" – Version #4

- By using a **struct** to define the relative memory offsets, we can get the compiler to generate all the correct address accesses relative to the base address.

"Hello, World!“ – Version #4

1/1

```c
#define __I    volatile const  // read only permission
#define __IO   volatile        // read/write permission
#define __O    volatile        // write only permission ;-) doesn't work in C...

#define GPIO_BASE_ADDRESS 0x80090000

typedef struct {
    __IO alt_u32 DATA_REG;
    __IO alt_u32 DIRECTION_REG;
    __IO alt_u32 INTERRUPTMASK_REG;
    __IO alt_u32 EDGECAPTURE_REG;
    __O  alt_u32 OUTSET_REG;
    __O  alt_u32 OUTCLEAR_REG;
}
PIO_TYPE;

#define LEDS ((PIO_TYPE *) GPIO_BASE_ADDRESS))
```

[C] Source Code Excerpt: Nios II "Hello, World!“ – Version #4

"Hello, World!"

1/1

```c
LEDS->DATA ^= 0xFF;

```

[C] Source Code Excerpt: Nios II "Hello, World!" – Version #4

Notes

- We use struct-pointer dereferencing to access the individual registers

"Hello, World!“ – Version #5

Peripheral Register Access in C

"Hello, World!“ – Version #5

1/1

```
#define __I    volatile const  // read only permission
#define __IO   volatile        // read/write permission
#define __O    volatile        // write only permission ;-) doesn't work in C...

#define GPIO_BASE_ADDRESS 0x80090000

typedef struct {
    __IO alt_u32 DATA_REG;
    __IO alt_u32 DIRECTION_REG;
    __IO alt_u32 INTERRUPTMASK_REG;
    __IO alt_u32 EDGECAPTURE_REG;
    __O  alt_u32 OUTSET_REG;
    __O  alt_u32 OUTCLEAR_REG;
}
PIO_TYPE;

#define LEDS (*((PIO_TYPE *) GPIO_BASE_ADDRESS))
```

[C] Source Code Excerpt: Nios II "Hello, World!“ – Version #5

"Hello, World!"

1/1

```
LEDS.DATA ^= 0xFF;

```

[C] Source Code Excerpt: Nios II "Hello, World!" – Version #4

Notes

- We can use the dot operator to access the output data register DATA.
- Both statements are equivalent and generate the same set of machine instructions.

Summary so far ...

Peripheral Register Access in C

Summary so far ...

- Both approaches (Version #4 and #5) are equivalent and generate the same assembly code.