ECE 448
Lecture 15

Bare Metal System Software Development

Required Reading

P. Chu, FPGA Prototyping by VHDL Examples

9 Bare Metal System Software Development

Revised Source Code (recommended)

Sampler FPro System (including changes proposed by Zach Monte on 03/23/2021)

https://people-ece.vse.gmu.edu/coursewebpages/ECE/ECE448/S23/labs/448_lab5.htm

Zip file:

sampler_example_src.zip

Folders:

sampler_example_src/sw/drv (drivers)

sampler_example_src/sw/app (application)

Original Source Code

Companion Website of

FPGA Prototyping by VHDL Examples 2nd edition

https://academic.csuohio.edu/chu-pong/fpga-vhdl-soc-book/

- **Source codes**

  read_me file: readme_source_code.pdf

  source file: fpga_mcs_vhdl_src.zip
  (last updated 11/10/2017)

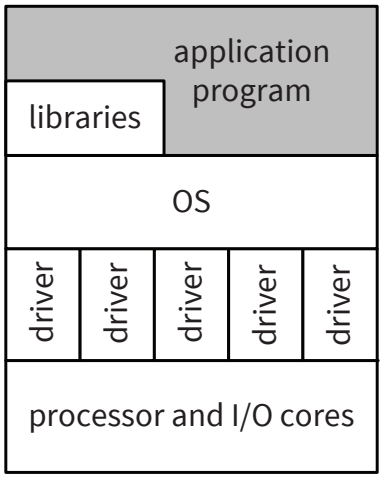
Folders:

fpga_mcs_vhdl_src/cpp/drv (drivers)

fpga_mcs_vhdl_src/cpp/app (applications)

Desktop-like System vs. Bare Metal System

Desktop-like System

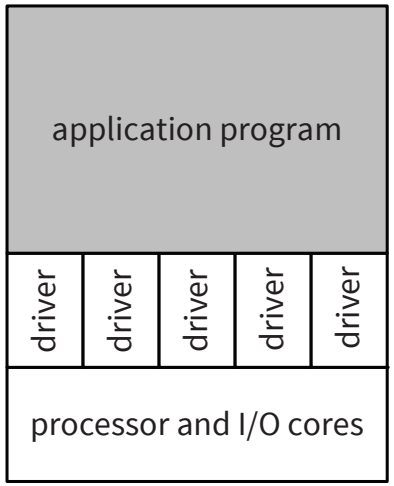


Hosted environment

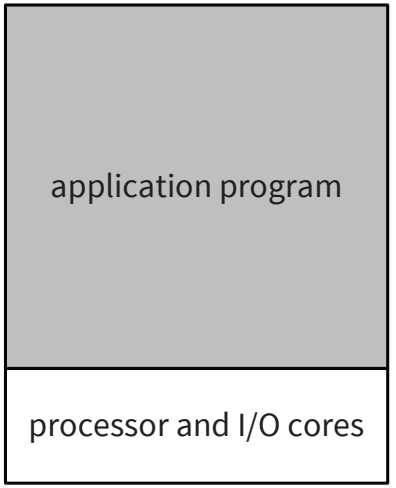
- Before starting main(), the operating system (OS) allocates necessary resources and initializes system services
- After the main() function exits, it returns control to the OS

Bare Metal System

with drivers



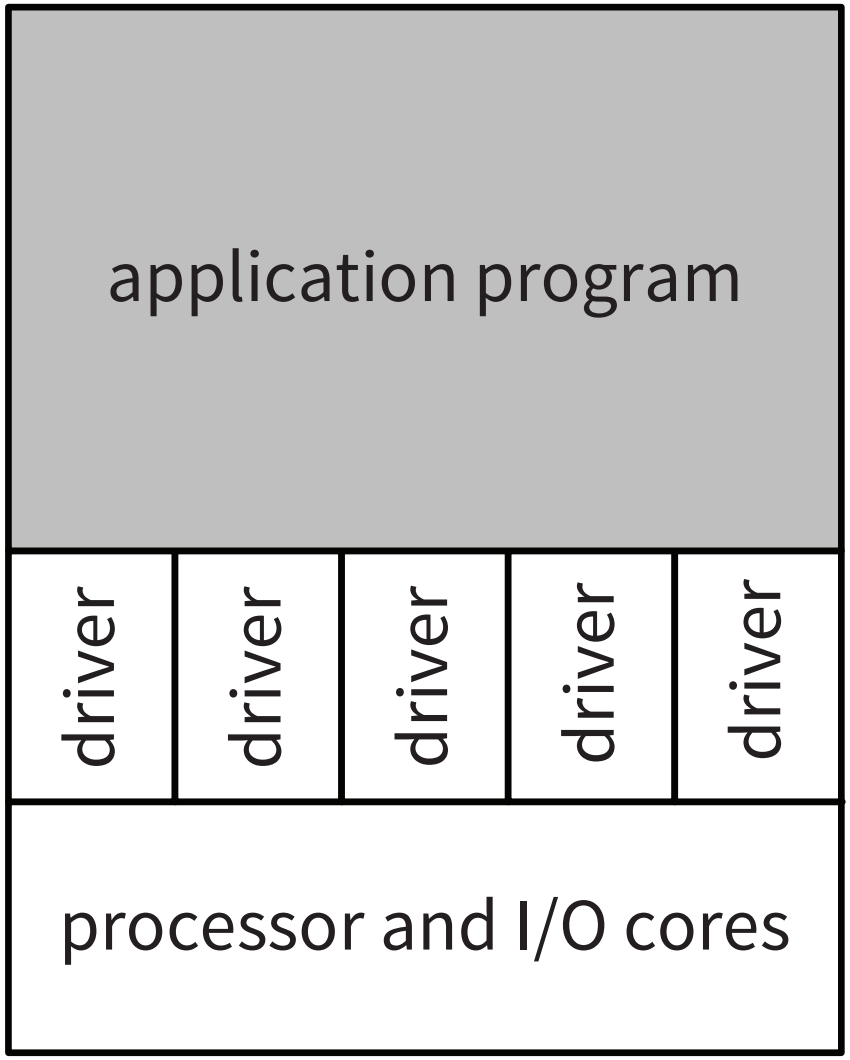
without drivers



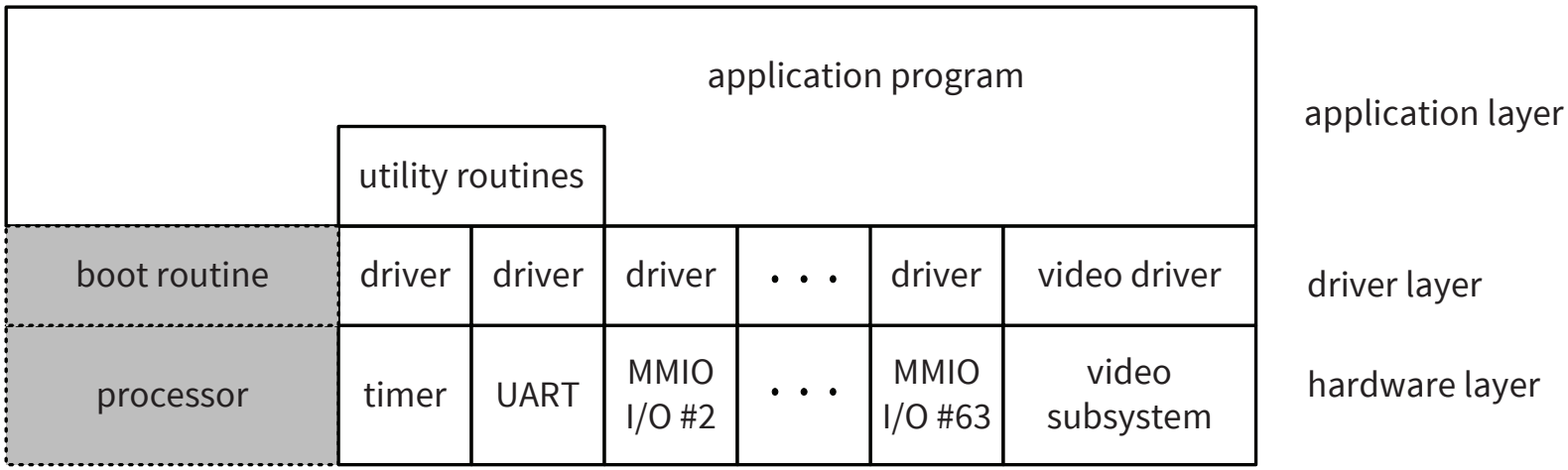
Freestanding environment

- Application itself is responsible for all resource allocations and initializations
- The main() function never exits

Software hierarchy of an FPro SoC system



FPro Software Organization




```
main() {
  sys_init();
  while(1) {
    task_1();
    task_2();
    . . .
    task_n();
  }
```

super loop

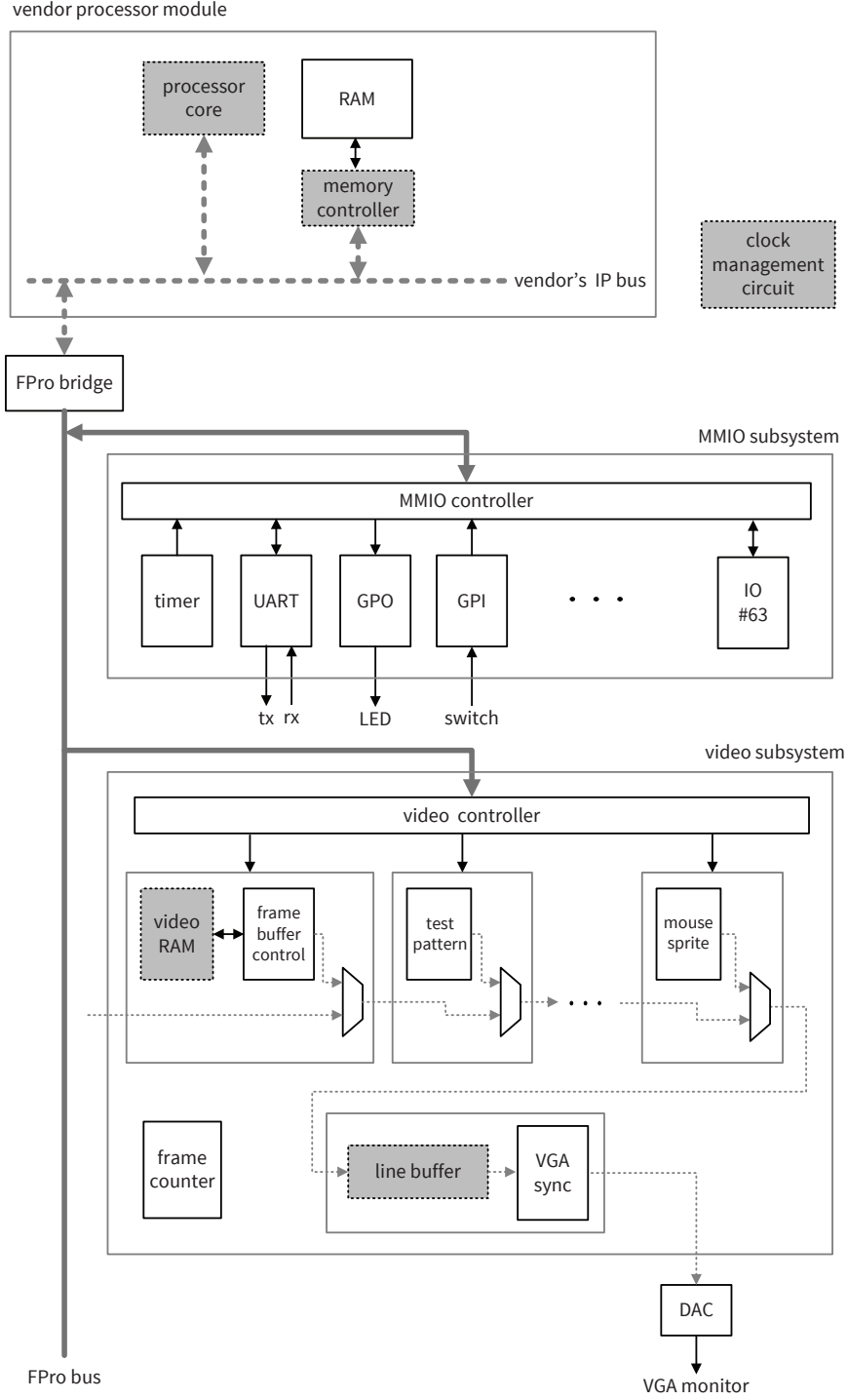
FPro Software Organization

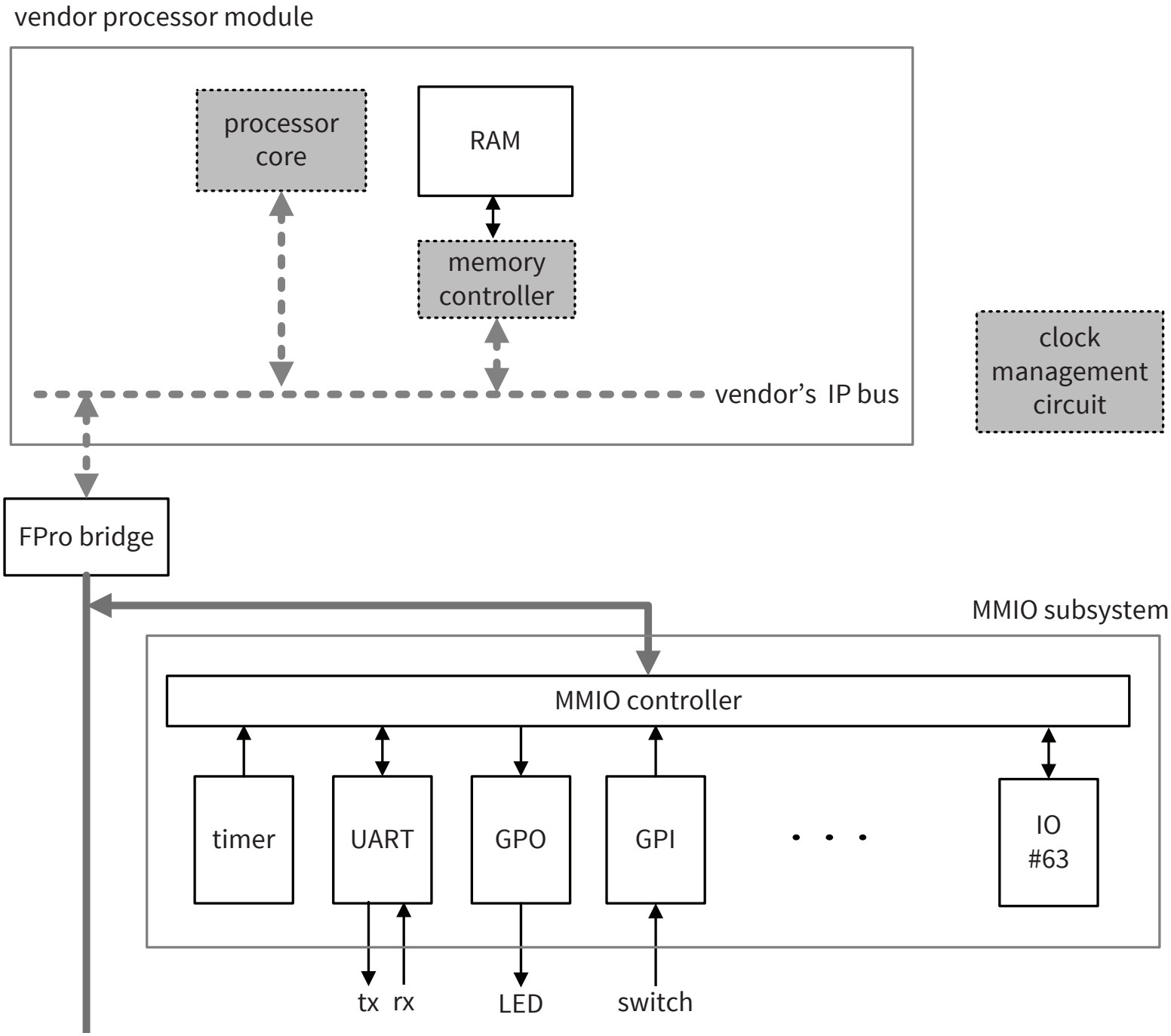
- A simple bare metal software scheme - No operating system
- The processor boots directly into an infinite main loop
- Software hierarchy of an FPro system contains
  - application layer
  - driver layer
  - hardware layer
- A boot routine is associated with the processor.

  It performs the basic initialization process, such as
  - clearing the caches,
  - configuring the stack and heap segments
  - initializing the interrupts,

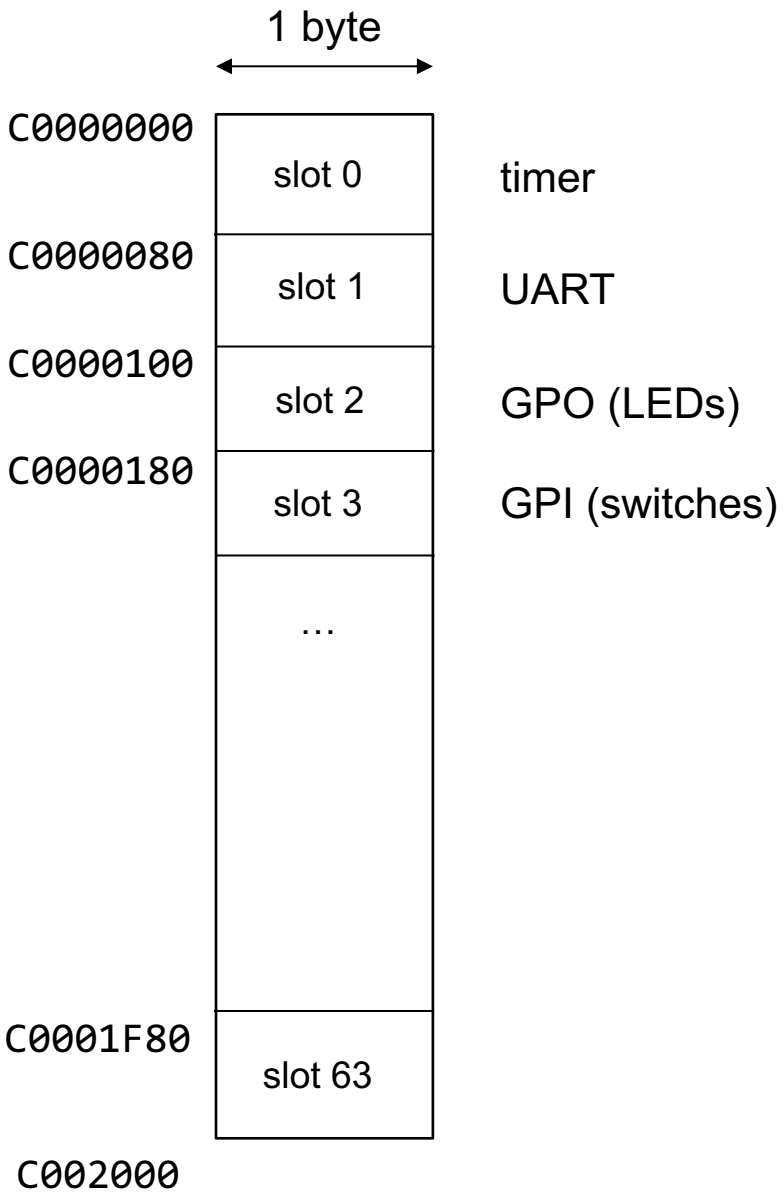
  and then transfers control to the main program.
- The timer core and UART maintain a system time and assist displaying a debug message on the console.

Top-level diagram of an FPro system




vendor processor module
processor core
RAM
memory controller
vendor's IP bus
clock management circuit
FPro bridge
MMIO subsystem
MMIO controller
timer
UART
GPO
GPI
IO #63
tx
rx
LED
switch

Address Map of the MMIO Subsystem



MMIO Subsystem

64 slots x **128 bytes per slot**

$= 2^6 \times 2^7$ words $= 2^{13}$ bytes

128 (decimal) = 0x80 (hex)

2·128 = 256 (decimal) = 0x100 (hex)

3·128 = 384 (decimal) = 0x180 (hex)

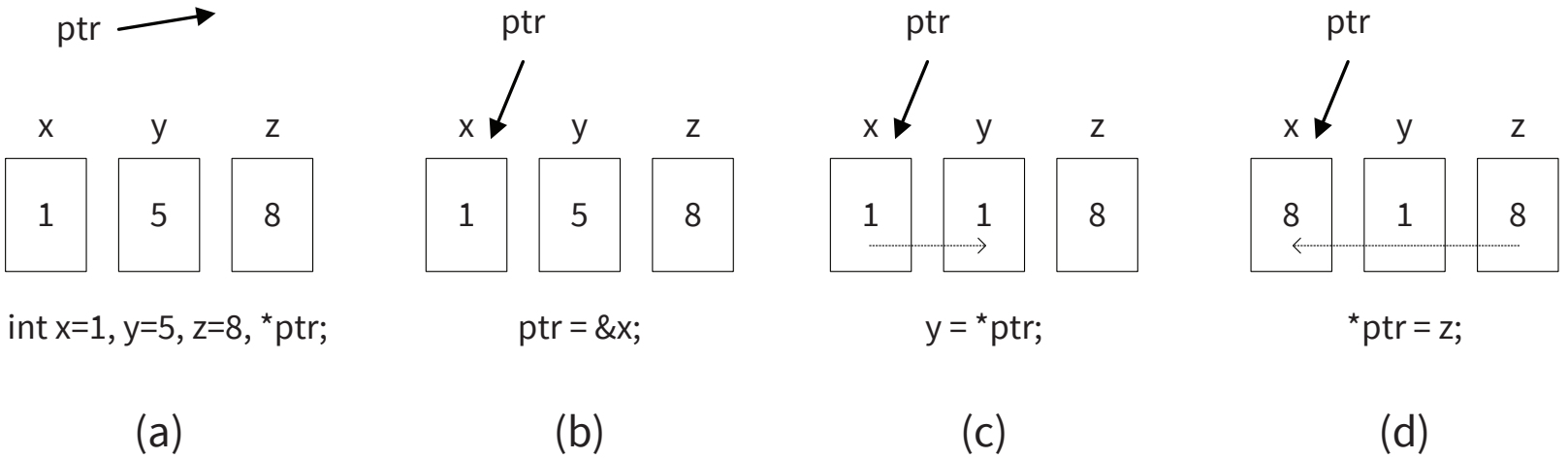
…

63·128 = 8064 (decimal) = 0x1F80 (hex)

2^{13} (decimal) = 0x2000 (hex)

C Pointers

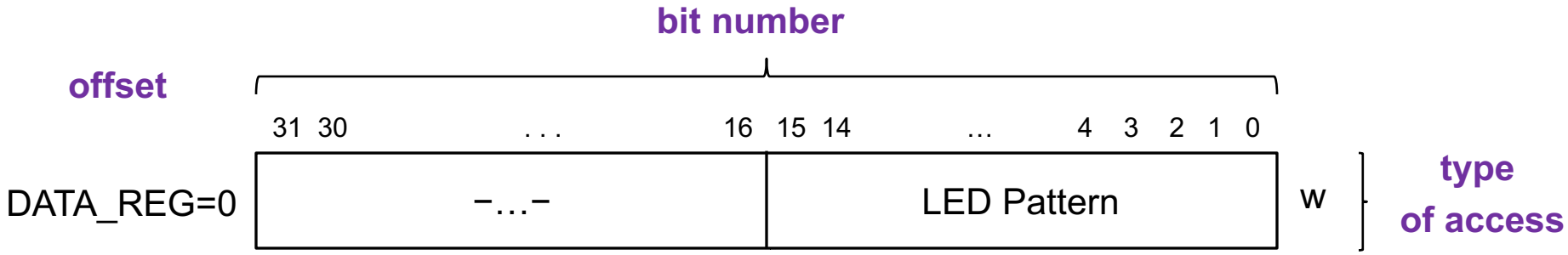
```
int x=1, y=5, z=8, *ptr;

ptr = &x;        // ptr gets symbolic address of x
y = *ptr;        // y is set to the content pointed by ptr
*ptr = z;        // variable pointed a pointer is set to the content of z
```



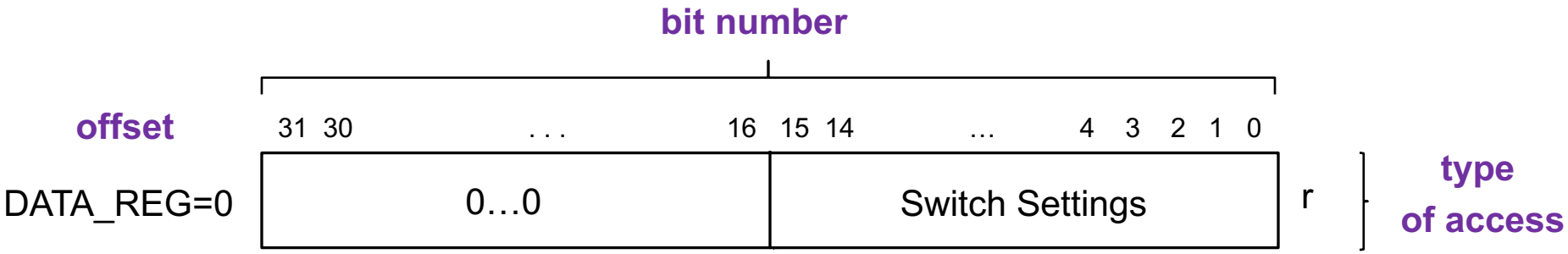
Data Types in inttypes.h

- int8_t: signed 8-bit integer
- uint8_t: unsigned 8-bit integer
- int16_t: signed 16-bit integer
- uint16_t: unsigned 16-bit integer
- int32_t: signed 32-bit integer
- uint32_t: unsigned 32-bit integer
- int64_t: signed 64-bit integer
- uint64_t: unsigned 64-bit integer

IO Register Map of the GPO Core (LEDs)



IO Register Map of the GPI Core (switches)



Writing to LEDs and Reading Positions of Switches w/o Using Drivers (1)

-- Base address of the GPO (LED) core = 0xc0000100
-- Base address of the GPI (SW) core = 0xc0000180

```
uint16_t  pattern  = 0x0055;
uint16_t  sw;

*((uint16_t *) 0xc0000100) = pattern;
sw = *((uint16_t *) 0xc0000180);
```

I/O Macros
in chu_io_rw.h (1)

```
#include <inttypes.h>    // to use unitN_t type


/*****************************************************************************
 * generic low-level read and write access
 *  - offset: 32-bit word offset relative to base
 *  - 4*offset used for byte address
 *  - must bypass data cache for I/O access
 *****************************************************************************/
```

I/O Macros
in chu_io_rw.h (2)

```
/*
 * Write an io register
 * base_addr - base address of an io core
 * offset - register word offset
 * data - 32-bit data
 */

#define io_write(base_addr, offset, data) (*(volatile uint32_t *)((base_addr) + 4*(offset)) = (data))

/* Read an io register.
 * base_addr - base address of an io core
 * offset - register word offset
 * returns 32-bit data of the register
 * macro calculates the byte address of the register and then reads
 */

#define io_read(base_addr, offset) (*(volatile uint32_t *)((base_addr) + 4*(offset)))
```

A volatile keyword in C

A volatile keyword in C is a qualifier used by the programmer when they declare a variable in source code. It is used to inform the compiler that the variable value can be changed any time without any task given by the source code. Volatile is usually applied to a variable when we are declaring it. The main reason behind using volatile keyword is that it is used to prevent optimizations on objects in our source code.

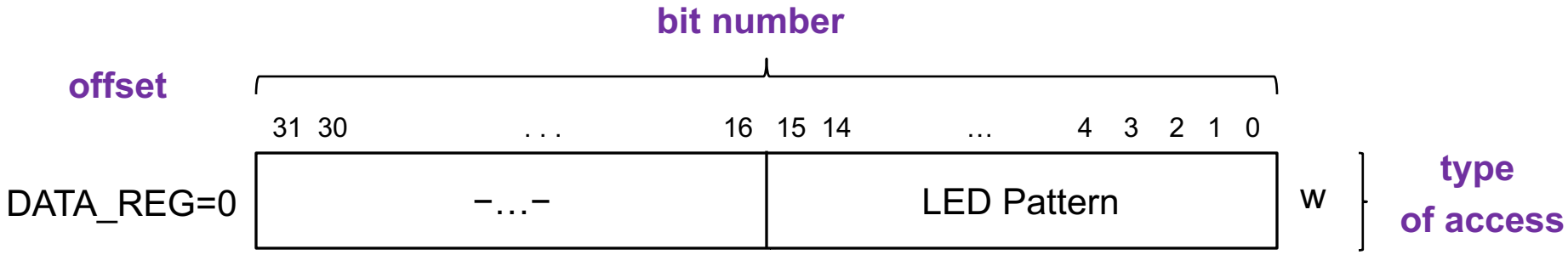
Writing to LEDs and Reading Positions of Switches w/o Using Drivers (2)

-- Base address of the GPO (LED) core = 0xc0000100

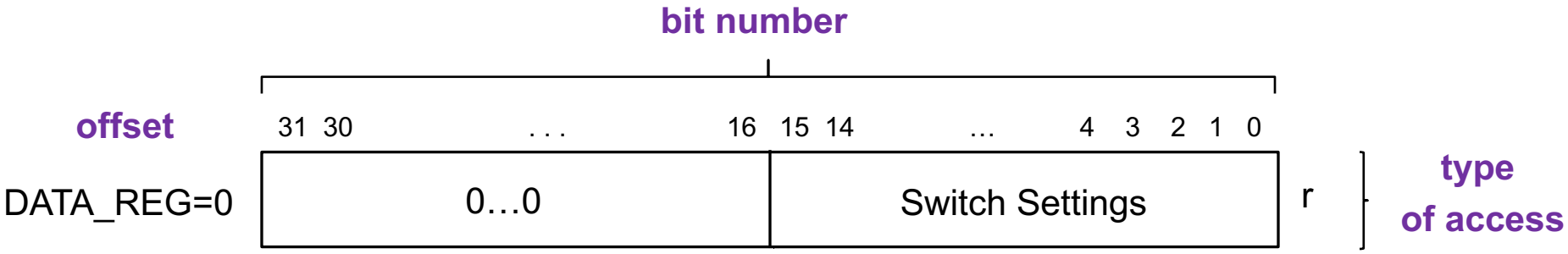
-- Base address of the GPI (SW) core = 0xc0000180

```
uint16_t  pattern  = 0x0055;
uint16_t  sw;

iowrite(0xc0000100, 0, (uint32_t) pattern) = pattern;
sw = (uint16_t) ioread(0xc0000180, 0);
```

IO Register Map of the GPO Core (LEDs)



IO Register Map of the GPI Core (switches)



GpoCore class definition in gpio_cores.h (1)

```
#include "chu_init.h"
class GpoCore {
public:
 /* register map */
   enum {
      DATA_REG = 0 /* output data register */
   };

/* methods */
 GpoCore(uint32_t core_base_addr);
   ~GpoCore();                  // not used

 void write(uint32_t data);
 void write(int bit_value, int bit_pos);

private:
   uint32_t base_addr;
   uint32_t wr_data;      // same as GPO core data reg
};
```

GpiCore class definition in gpio_cores.h (1)

```
#include "chu_init.h"

class GpiCore {
public:
 /* register map */
   enum {
      DATA_REG = 0 /* input data register */
   };

/* methods */
 GpiCore(uint32_t core_base_addr);
   ~GpiCore();                  // not used

 uint32_t read();
 int read(int bit_pos);

private:
   uint32_t base_addr;
};
```

Writing to LEDs and Reading Positions of Switches by Using Drivers

```
#include "chu_init.h"

GpoCore gpo((get_slot_addr(BRIDGE_BASE, S2_LED)));
GpiCore gpi((get_slot_addr(BRIDGE_BASE, S3_SW)));

int main() {

uint16_t  pattern  = 0x0055;
uint16_t  sw;

gpo.write((uint32_t) pattern);
sw = (uint16_t) gpi.read();
}
```

Constants Representing Slot Numbers and Bridge Base Address in C/C++

chu_io_map.h

```
#define S0_SYS_TIMER  0
#define S1_UART1      1
#define S2_LED        2
#define S3_SW         3
#define S4_USER       4
#define S5_XDAC       5
#define S6_PWM        6
#define S7_BTN        7
#define S8_SSEG       8
#define S9_SPI        9
#define S10_I2C      10
#define S11_PS2      11
#define S12_DDFS     12
#define S13_ADSR     13

//io base address for microBlaze MCS
#define BRIDGE_BASE 0xc0000000
```

I/O Macros
in chu_io_rw.h (3)

```
/* Calculate base address of a memory mapped io slot.
 * base base-address of FPro system.
 * slot designated io slot number
 * returns base address of the slot
 */

#define get_slot_addr(base, slot)  ((uint32_t)((base) + (slot)*32*4))
```

chu_init.h

```
#include "chu_io_rw.h"

#include "chu_io_map.h"

#include "timer_core.h"

#include "uart_core.h"
```

No need to include these files on top of chu_init.h

GpoCore class implementation in gpio_cores.cpp (1)

```
#include "gpio_cores.h"

GpoCore::GpoCore(uint32_t core_base_addr) {
   base_addr = core_base_addr;
   wr_data = 0;
}

GpoCore::~GpoCore() {
}

void GpoCore::write(uint32_t data) {
   wr_data = data;
   io_write(base_addr, DATA_REG, wr_data);
}

void GpoCore::write(int bit_value, int bit_pos) {
   bit_write(wr_data, bit_pos, bit_value);
   io_write(base_addr, DATA_REG, wr_data);
}
```

GpiCore class implementation in gpio_cores.cpp (1)

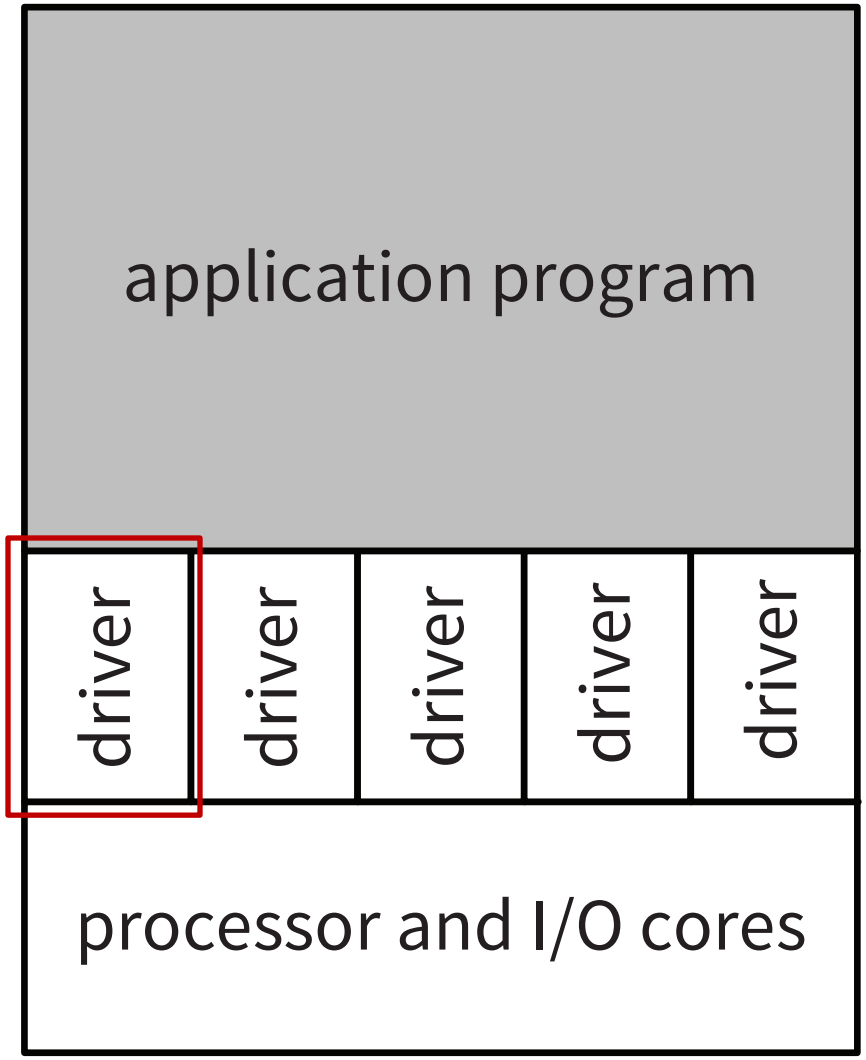
```
GpiCore::GpiCore(uint32_t core_base_addr) {
   base_addr = core_base_addr;
}

GpiCore::~GpiCore() {
}

uint32_t GpiCore::read() {
   return (io_read(base_addr, DATA_REG));
}

int GpiCore::read(int bit_pos) {
   uint32_t rd_data = io_read(base_addr, DATA_REG);
   return ((int) bit_read(rd_data, bit_pos));
}
```

Useful Bit Manipulation Macros

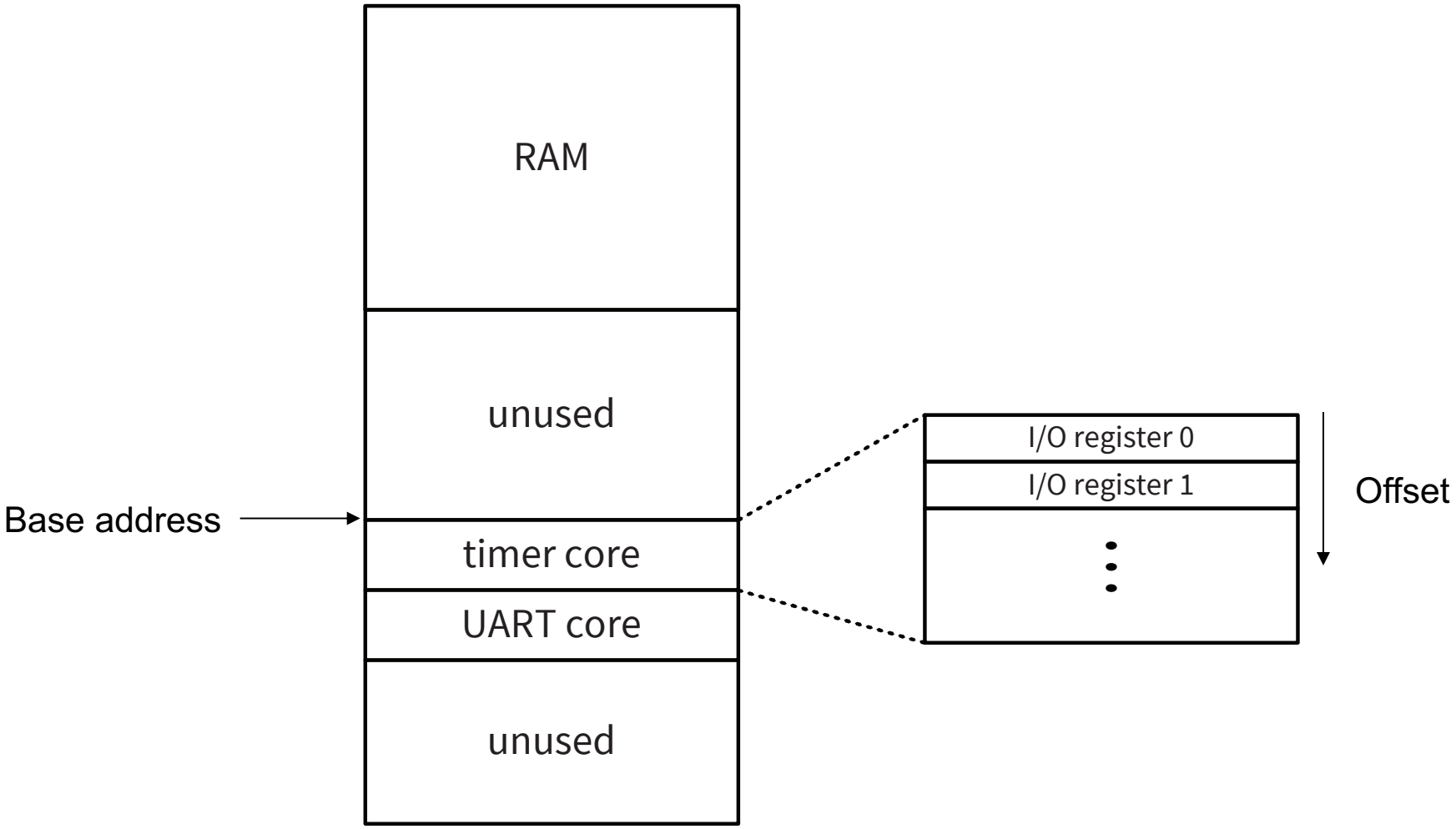
chu_init.h

```
#define bit_set(data, n) ((data) |= (1UL << (n)))

#define bit_clear(data, n) ((data) &= ~(1UL << (n)))

#define bit_toggle(data, n) ((data) ^= (1UL << (n)))

#define bit_read(data, n) (((data) >> (n)) & 0x01)

#define bit_write(data, n, bitvalue)
     (bitvalue ? bit_set((data), n) : bit_clear((data), n))

#define bit(n) (1UL << (n))
```

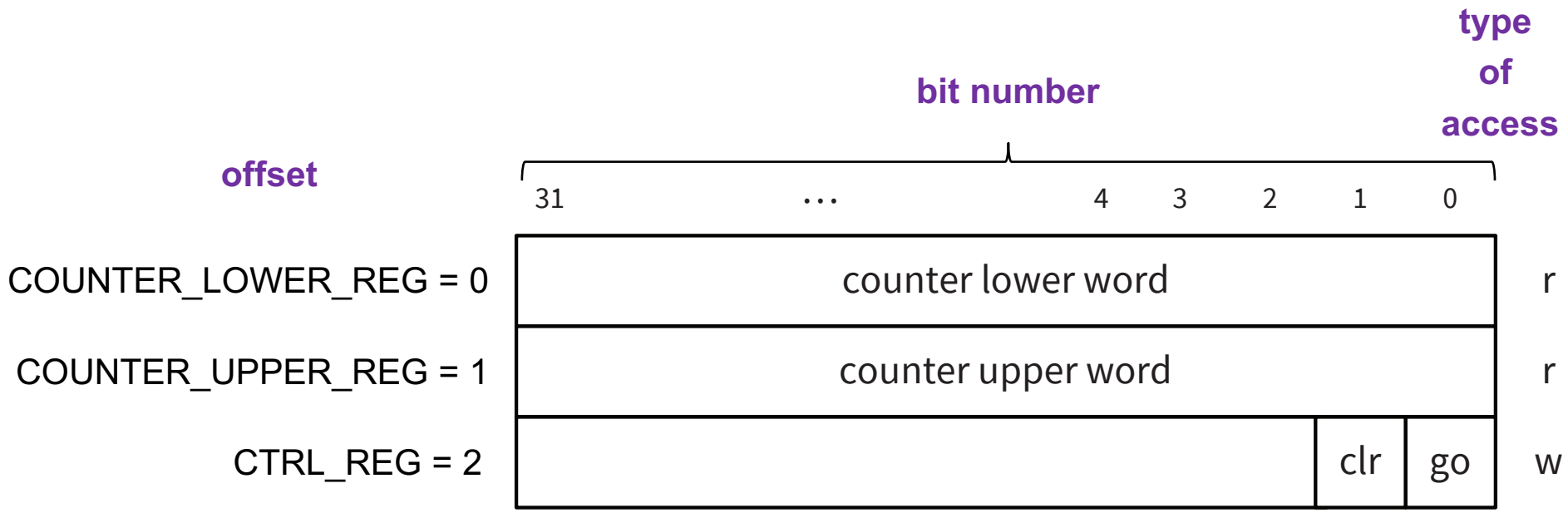
Driver of the Timer Core



Address Map of the FPro System



IO Register Map of the Timer Core



Functions of the Timer Core driver

timer_core.h

```
class TimerCore {
public:
…
void pause();

void go();

void clear();

uint64_t read_tick();

uint64_t read_time();

void sleep(uint64_t us);

private:
   …
};
```

For the detailed descriptions of all functions, see

timer_core.h

located in

sampler_example_src/sw/drv

and

fpga_mcs_vhdl_src/cpp/drv

Calling functions of the Timer Core driver

```
#include "chu_init.h"

TimerCore timer((get_slot_addr(BRIDGE_BASE, S0_SYS_TIMER)));

int main(){

timer.clear();
timer.go();
timer.pause();
uint64_t ticks = timer.read_tick();
timer.sleep(1000000);

}
```

TimerCore class definition in timer_core.h (1)

```
#include "chu_io_rw.h"
#include "chu_io_map.h"     // to obtain system clock rate

class TimerCore {

public:
  /* register map */
  enum {
    COUNTER_LOWER_REG = 0, // lower 32 bits of counter
    COUNTER_UPPER_REG = 1, // upper 16 bits of counter
    CTRL_REG = 2           // control register
  };

  /* masks */
  enum {
    GO_FIELD = 0x00000001, // bit 0 of ctrl_reg; enable bit
    CLR_FIELD = 0x00000002 // bit 1 of ctrl_reg; clear bit
  };
```

TimerCore class definition in timer_core.h (2)

```
/* methods */
   TimerCore(uint32_t core_base_addr);    // constructor
   ~TimerCore();                          // destructor, not used

   void pause();                          // pause counter
   void go();                             // resume counter
   void clear();                          // clear the counter to 0
   uint64_t read_tick();                  // retrieve # clock cycles elapsed
   uint64_t read_time();                  // read time elapsed in microseconds
   void sleep(uint64_t us);               // idle for us microseconds

private:
   uint32_t base_addr;
   uint32_t ctrl;                    // current state of ctrl_reg
};
```

TimerCore class implementation in timer_core.cpp (1)

```
#include "timer_core.h"

TimerCore::TimerCore(uint32_t core_base_addr) {
   base_addr = core_base_addr;
   ctrl = 0x01;
   clear();
   io_write(base_addr, CTRL_REG, ctrl);         // enable the timer
}

TimerCore::~TimerCore() {
}

void TimerCore::clear() {
   uint32_t wdata;
      // write clear_bit to generate a 1-clock pulse
      // clear bit does not affect ctrl
   wdata = ctrl | CLR_FIELD;
   io_write(base_addr, CTRL_REG, wdata);
}
```

TimerCore class implementation in timer_core.cpp (2)

```
void TimerCore::pause() {
   // reset enable bit to 0
   ctrl = ctrl & ~GO_FIELD;
   io_write(base_addr, CTRL_REG, ctrl);
}

void TimerCore::go() {
   // set enable bit to 1
   ctrl = ctrl | GO_FIELD;
   io_write(base_addr, CTRL_REG, ctrl);
}

uint64_t TimerCore::read_tick() {
   uint64_t upper, lower;
   lower = (uint64_t) io_read(base_addr, COUNTER_LOWER_REG);
   upper = (uint64_t) io_read(base_addr, COUNTER_UPPER_REG);
   return ((upper << 32) | lower);
}
```

TimerCore class implementation in timer_core.cpp (3)

```
uint64_t TimerCore::read_time() {
   // elapsed time in microsecond (SYS_FREQ in MHz)
   return (read_tick() / SYS_CLK_FREQ);
}

void TimerCore::sleep(uint64_t us) {
   uint64_t start_time, now;
   start_time = read_time();
   // busy waiting
   do {
      now = read_time();
   } while ((now - start_time) < us);
}
```

Constant

chu_io_map.h

```
// system clock rate in MHz; used for timer and uart

#define SYS_CLK_FREQ 100
```