Lab-5. EE3401, Jan-Apr2024

In this lab, we are going to access the peripheral devices namely general purpose input/output pins (GPIOs, can be considered digital I/O pins) and timer on Raspberry Pi Pico microcontroller.

Our basic document is https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf, which describes the RP2040 microcontroller hardware organisation. A wider view of the Cortex-M series of processors is provided by this document. RP2040 microcontroller is based on M0+ cores, which use "Thumb" instruction set, which has 16 bit wide instructions, and is similar to ARM ISA. It has only two registers to refer to, therefore an instruction like "add r1, r2" means r1 = r1 + r2. You should be able to understand thumb code without much problem if you know ARM ISA.

The references to the figures and tables below are to the RP2040-datasheet linked above.

The peripherals on microcontrollers (see Fig.2, p.11) are generally accessed by mapping their configuration registers to different addresses in memory. Some of the memory is actual storage (e.g. flash memory, DRAM etc.) but access to other parts meant for peripherals are routed to the peripherals by the memory bus. This map is called a "memory map" (Tab.15, p24). A particular hardware block called "sio" (software IO block) is used on the Pico to interact with the peripherals in a fast manner (a single clock cycle). For our purposes, we can assume that the registers we are interested in are located at some offset from the base pointed by sio, see below for details.

We will use the information contained in this document which gives pointers and this webpage which has some assembly code.

1. Running assembly on Wokwi.com. Use the following code in a new project (select "starter templates" "blink" for Rpi Pico). "Bkpt #42" dumps the register values in the developer console of your browser "To view the register values open the Chrome developer tools console: Option+⌘+J on macOS / Shift+CTRL+J on windows / F12 on Firefox".

```
void setup() {
 asm(R"(
.syntax unified
.cpu cortex-m0plus
.thumb
   movs r0, #0x2     // r0 = 0x2;
   movs r1, #0x20    // r1 = 0x20;
   adds r1, r0       // r1 |= r0;
   bkpt #42         // dump register values
)");
}
void loop() {
```

```
 // put your main code here, to run repeatedly:
 delay(1); // this speeds up the simulation
}
```

Run this code and verify that register r1 has the expected value.

2. Use the following definitions, find out the meaning of “FUNCSEL_SIO” and “GPIO_*” registers mentioned below (they are described in Tab.15, p24).

```
syntax unified
.cpu cortex-m0plus
.thumb
gpio_all_pins_mask: .word 0
gpio_even_pins_mask: .word 0

.equ IO_BANK0_BASE, 0x40014000
.equ IO_BANK0_GPIO_CTRL_BASE, IO_BANK0_BASE + 0x04
.equ FUNCSEL_SIO, 5
.equ GPIO_FUNCSEL, FUNCSEL_SIO
.equ GPIO_INIT_VALUE, (GPIO_FUNCSEL<<0)

@ RP2040 Section: 2.19.6.3. Pad Control - User Bank
.equ PADS_BANK0_BASE, 0x4001c000
.equ PADS_BANK0_GPIO_OFFSET, 0x04
.equ PADS_BANK0_GPIO_BASE, PADS_BANK0_BASE + PADS_BANK0_GPIO_OFFSET
@ Bit 7 - OD Output disable. Has priority over output enable from
peripherals RW 0x0
@ Bit 6 - IE Input enable RW 0x1
@ Bits 5:4 - DRIVE Drive strength.
@           0x0 → 2mA
@           0x1 → 4mA
@           0x2 → 8mA
@           0x3 → 12mA
@           RW 0x1
@ Bit 3 - PUE Pull up enable RW 0x0
@ Bit 2 - PDE Pull down enable RW 0x1
@ Bit 1 - SCHMITT Enable schmitt trigger RW 0x1
@ Bit 0 - SLEWFAST Slew rate control. 1 = Fast, 0 = Slow RW 0x0
@          bit nums:   76543210
.equ PAD_INIT_VALUE, 0b00010110

@ RP2040 Section: 2.3.1. SIO
@ 2.3.1.7. List of Registers
.equ SIO_BASE, 0xd0000000
.equ GPIO_OE,     SIO_BASE + 0x020 @ GPIO output enable
.equ GPIO_OE_SET, SIO_BASE + 0x024 @ GPIO output enable set
.equ GPIO_OE_CLR, SIO_BASE + 0x028 @ GPIO output enable clear
.equ GPIO_OE_XOR, SIO_BASE + 0x02c @ GPIO output enable XOR
.equ GPIO_OUT,     SIO_BASE + 0x010 @ GPIO output value
.equ GPIO_OUT_SET, SIO_BASE + 0x014 @ GPIO output value set
```

```
.equ GPIO_OUT_CLR, SIO_BASE + 0x018 @ GPIO output value clear
.equ GPIO_OUT_XOR, SIO_BASE + 0x01c @ GPIO output value XOR

LEDS: .byte 8, 9, 10, 11, 12, 13, 14, 15
LEDS_LEN = . - LEDS
```

Put these values in the assembly code section given in (1) above and see if it complies.

3. Now we can add functions to turn on the GPIOs.

```
syntax unified
.cpu cortex-m0plus
.thumb
gpio_all_pins_mask: .word 0
gpio_even_pins_mask: .word 0

.equ IO_BANK0_BASE, 0x40014000
.equ IO_BANK0_GPIO_CTRL_BASE, IO_BANK0_BASE + 0x04
.equ FUNCSEL_SIO, 5
.equ GPIO_FUNCSEL, FUNCSEL_SIO
.equ GPIO_INIT_VALUE, (GPIO_FUNCSEL<<0)

@ RP2040 Section: 2.19.6.3. Pad Control - User Bank
.equ PADS_BANK0_BASE, 0x4001c000
.equ PADS_BANK0_GPIO_OFFSET, 0x04
.equ PADS_BANK0_GPIO_BASE, PADS_BANK0_BASE + PADS_BANK0_GPIO_OFFSET
@ Bit 7 - OD Output disable. Has priority over output enable from
peripherals RW 0x0
@ Bit 6 - IE Input enable RW 0x1
@ Bits 5:4 - DRIVE Drive strength.
@           0x0 → 2mA
@           0x1 → 4mA
@           0x2 → 8mA
@           0x3 → 12mA
@           RW 0x1
@ Bit 3 - PUE Pull up enable RW 0x0
@ Bit 2 - PDE Pull down enable RW 0x1
@ Bit 1 - SCHMITT Enable schmitt trigger RW 0x1
@ Bit 0 - SLEWFAST Slew rate control. 1 = Fast, 0 = Slow RW 0x0
@          bit nums:   76543210
.equ PAD_INIT_VALUE, 0b00010110

@ RP2040 Section: 2.3.1. SIO
@ 2.3.1.7. List of Registers
.equ SIO_BASE, 0xd0000000
.equ GPIO_OE,     SIO_BASE + 0x020 @ GPIO output enable
.equ GPIO_OE_SET, SIO_BASE + 0x024 @ GPIO output enable set
.equ GPIO_OE_CLR, SIO_BASE + 0x028 @ GPIO output enable clear
.equ GPIO_OE_XOR, SIO_BASE + 0x02c @ GPIO output enable XOR
.equ GPIO_OUT,     SIO_BASE + 0x010 @ GPIO output value
```

```
.equ GPIO_OUT_SET, SIO_BASE + 0x014 @ GPIO output value set
.equ GPIO_OUT_CLR, SIO_BASE + 0x018 @ GPIO output value clear
.equ GPIO_OUT_XOR, SIO_BASE + 0x01c @ GPIO output value XOR

LEDS: .byte 8, 9, 10, 11, 12, 13, 14, 15
LEDS_LEN = . - LEDS

init_all_gpios

.thumb_func
init_all_gpios:
   push    {r0-r2, LR}
   @ Usage:
   @ r0 - reserved for calling init_single_gpio
   @ r1 - base address of LED numbers array
   @ r2 - initially offset of last value in LED numbers array, then
counted down
   ldr     r1, =LEDS
   movs    r2, #LEDS_LEN-1
_init_all_gpios_loop:
   ldrb    r0, [r1, r2]
   bl      init_single_gpio
   subs    r2, r2, #1
   bpl     _init_all_gpios_loop
   pop     {r0-r2, PC}

.thumb_func
init_single_gpio:
   @ Input:
   @ r0 - LED number
   push    {r1-r3, LR}
   @ Usage:
   @ r1 - base addresses
   @ r2 - computed address offsets
   @ r3 - (computed) values to set in register

   @ init IO_BANK0 register
   ldr     r1, =IO_BANK0_GPIO_CTRL_BASE
   ldr     r3, =GPIO_INIT_VALUE    @ Load early to reduce stall
cycles (Probably makes no difference on Cortex M0+!)
   lsls    r2, r0, #3      @ Multiply LED number by 8 because
IO_BANK0 config is pairs of 32 bit, 4 byte registers
   str     r3, [r1, r2]

   @ init PAD
   ldr     r1, =PADS_BANK0_GPIO_BASE
   ldr     r3, =PAD_INIT_VALUE    @ Load early to reduce stall
cycles (Probably makes no difference on Cortex M0+!)
   lsls    r2, r0, #2      @ Multiply LED number by 4 as 4 bytes
per registers
   str     r3, [r1, r2]
```

```
    @ init SIO (Set low, output enable)
    movs    r3, #1
    lsls    r3, r3, r0
    ldr     r1, =GPIO_OUT_CLR
    str     r3, [r1]
    ldr     r1, =GPIO_OE_SET
    str     r3, [r1]

    pop     {r1-r3, PC}
```

After calling the initialisation function above, you can use a code like this to set the GPIO8 to high. To test if this works, add an LED and a resistor from GPIO8 to the ground (i.e. GPIO8-resistor-LED-gnd loop in the diagram window of wokwi). Run the code and the LED should glow.

```
    ldr     r0, =LEDS
    movs    r1, #0
    movs    r2, #0
    ldrb    r4, [r0, r1]                @ Load LED number
    movs    r5, #1                      @ Compute bit to update
    lsls    r5, r4
    ldr     r6, =GPIO_OUT_SET
    str     r5, [r6]
```

Can you turn on the other GPIOs?

Write a function to turn a particular GPIO and test it with LEDs. Can you make them glow in a sequence i.e. adding delay between them, using the delay function (e.g. delay(1)) of arduino IDE?

4. Now we are going to add a delay using a busy loop. Use the following code to do this:

```
wait_1_second:
    push    {r0-r3, LR}
    @ Usage:
    @ r0 - 1 million
    @ r1 - TIMER_TIMERAWL address
    @ r2 - start time
    @ r3 - current time and delta time
    ldr     r0, =#1000000
    ldr     r1, =TIMER_TIMERAWL
    ldr     r2, [r1]
_wait_1_second_loop:
    ldr     r3, [r1]
    subs    r3, r3, r2  @ r3 = delta time
    cmp     r3, r0      @ compare to 1 million
    blt     _wait_1_second_loop
    pop     {r0-r3, PC}
```

Use this function to glow the LEDs in alternate fashion.

This is not a great use of the microcontroller, since we can use timer interrupts to achieve the same. Can you find any code to do it?
We will do this in the next lab and learn a few things about interrupt handlers.