ARM Architectural Quirks

Phase Locked Loop
Internally the Philips version of the ARM Cortex M0 processor runs at 48 MHz which is a cycle time of just 20.083 nsec. This would imply that a very fast external interface bus would be needed to communicate with the outside world. For many embedded controller systems this high speed interface is unnecessary. The solution to this problem is to add a phase locked loop into the hardware which allows the CPU to run at 48 MHz and to simultaneously communicate with a synchronized 12 MHz system on the outside. Thus, the LPC1114 requires only a 12 MHz crystal and from the outside, the user sees it as a 12 MHz system although inside it is actually executing instructions four times faster.

A system diagram for a phase locked loop (PLL) is shown below. A 12 MHz crystal oscillator runs the external system busses and clocks. Internally a voltage controlled oscillator runs at about 48 MHz. The VCO is adjusted to run in synchronization with the 12 MHz oscillator by comparing the phase of the 12 MHz oscillator with that of the 48 MHz oscillator divided by 4. The 48 MHz oscillator therefore runs in synchronization with the 12 MHz oscillator and is just as precise. Effectively, the PLL acts as a frequency multiplier.

![System diagram for a phase locked loop.](image)

Memory Map
There are no separate I/O instructions on this processor. All I/O is memory mapped. For example, I/O port 0 can be addressed at address 0x50003FFC. Figure 2 shows the memory map for the processor. From the memory map we see that the LPC1114 has 32K of flash programmable memory and 8K of SRAM.
**System Clock**

The ARM Cortex M0 processor can make use of three independent oscillators. These are the internal RC oscillator, the external crystal oscillator, and the internal watchdog timer oscillator. Figure 3 shows a block diagram of the clock generation signals. For the board that we are using in class, a 12 MHz crystal has been added. However, if you remove the 28 pin DIP chip from the board it will run without the crystal and make use of the internal RC oscillator. This internal oscillator is not as accurate as the external crystal oscillator but it is within 1% of 12 MHz and adequate for many applications. So you can develop a program for the board and program it into the chip, remove the chip, and it will run your application without an external crystal.

Note that all of the peripherals have a clock and most have a clock divider that allows them to run at some speed slower than the 12 MHz. For applications that use the peripherals it is necessary to set up the clock divider register. For GPIO there is no clock divider register and on reset the GPIO clock is, by default, enabled and you can ignore it unless you want to turn it off and disable the GPIO.
A Programmer's Model of the LPC1114 ARM Processor

From a C programmer's point of view the LPC1114 consists of a CPU plus several I/O modules. The CPU has 16 general purpose registers which are accessible in assembly language. These are shown in Figure 4. The I/O modules include a 6-channel 10-bit A to D converter, three channels of PWM, one UART, one SPI serial interface, one i²C serial interface, a watchdog timer, four internal timers, an interrupt system that uses vectored interrupts, and 22-bits of general purpose I/O (GPIO). These I/O ports are bit programmable. Each of the I/O modules consists of several special function registers that define their operation. The definitions for all of the special function registers is in LPC1114.h and this file needs to be included in all C code. You can look at this file to see the assigned register names using any text editor. For this course we will be looking at all of the I/O modules except for the SPI module. Figure 5 gives a block diagram for the processor. Figure 6 give the pin assignments for all of the special function and I/O pins.

Figure 3
Clock generation for the ARM Cortex M0 processor.

Figure 4
CPU Registers for the ARM Cortex M0
Digital I/O
Digital I/O is referred to as General Purpose I/O or GPIO in the reference manuals. There are a total of 22 I/O pins with 12 of these in Port 0 and 10 in Port 1. All of the pins are brought out to connectors for the board we are using in class. All of the pins also have multiple functions and the user must select the proper function for each pin used in the software.
Before you use GPIO you should set up the function registers so the GPIO port is configured correctly. Figure 7 shows Table 173 from the User's manual with the registers which can be used for GPIO configuration.

<table>
<thead>
<tr>
<th>Name</th>
<th>Access</th>
<th>Address offset</th>
<th>Description</th>
<th>Reset value</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPIO_DATA</td>
<td>R/W</td>
<td>0x0000 to 0x01F8</td>
<td>Port n data address masking register locations for pins PIO&lt;sub&gt;n&lt;/sub&gt; to PIO&lt;sub&gt;n&lt;/sub&gt;_11 (see Section 12.4.1)</td>
<td>n/a</td>
</tr>
<tr>
<td>GPIO_DATA</td>
<td>R/W</td>
<td>0x02FFC</td>
<td>Port n data register for pins PIO&lt;sub&gt;n&lt;/sub&gt; to PIO&lt;sub&gt;n&lt;/sub&gt;_11</td>
<td>n/a</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>0x4000 to 0x7FFC</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>GPIO_DDR</td>
<td>R/W</td>
<td>0x0000</td>
<td>Data direction register for port n</td>
<td>0x00</td>
</tr>
<tr>
<td>GPIO_DRR</td>
<td>R/W</td>
<td>0x0004</td>
<td>Interrupt source register for port n</td>
<td>0x00</td>
</tr>
<tr>
<td>GPIO_DIR</td>
<td>R/W</td>
<td>0x0008</td>
<td>Interrupt both edges register for port n</td>
<td>0x00</td>
</tr>
<tr>
<td>GPIO_DIR</td>
<td>R/W</td>
<td>0x000C</td>
<td>Interrupt avant register for port n</td>
<td>0x00</td>
</tr>
<tr>
<td>GPIO_DIR</td>
<td>R/W</td>
<td>0x0010</td>
<td>Interrupt mask register for port n</td>
<td>0x00</td>
</tr>
<tr>
<td>GPIO_DIR</td>
<td>R/W</td>
<td>0x0014</td>
<td>Raw interrupt status register for port n</td>
<td>0x00</td>
</tr>
<tr>
<td>GPIO_DIR</td>
<td>R/W</td>
<td>0x0016</td>
<td>Masked interrupt status register for port n</td>
<td>0x00</td>
</tr>
<tr>
<td>GPIO_DIR</td>
<td>W</td>
<td>0x001C</td>
<td>Interrupt clear register for port n</td>
<td>0x00</td>
</tr>
</tbody>
</table>

Note that all GPIO pins come up in the input direction on reset. If you want to do output you have to set the direction to output in the GPIO Direction register. In general, on reset, all of the pins come up as programmable I/O (PIO) pins and you can set the direction as input or output and use them as GPIO. However, if you want to use a pin as say, an input to the A/D converter you must first configure that pin as analog input. You do this in the IO Configuration register (IOCONFIG). There is an IOCONFIG register for every pin although most of the bits in this register are reserved for future use. Chapter 7 of the User's manual has a list of all of the IOCONFIG registers for all of the pins. Figure 8 shows the IOCONFIG register for port 1 pin 10 as an example. For this pin, we see that to make it an analog input pin we need to set pin 7 to 0 in its IOCONFIG register.

Figure 8

IOCONFIG register for port 1 pin 10.
GPIO Examples

Note about define statements and addresses of registers

For the ARM Cortex M0 processor all of the registers are mapped to memory addresses. To access these in the C language we need to create a pointer whose value is the address of the register. To define a pointer we may use a statement like this:

```c
int* pointer1 = 0x50003FFC;  //pointer to GPIO Data register for Port 0
```

To store data at this address we can write:

```c
*pointer1 = 1234; //store 1234 at address 0x50003FFC
```

Likewise, to get the data from this address we can write:

```c
int x;
x = *pointer1;
```

Alternatively, since there is no need for a pointer variable we can create a define statement that looks like this:

```c
#define GPIO0DATA (*((volatile unsigned long *) 0x50003FFC))
```

In our program we can write data to this address like this:

```c
GPIO0DATA = 1234;
```

The compiler will substitute the pointer address and pointer declaration for GPIO0DATA. This allows us to write programs that use pointers without spelling out in detail, all of the pointer syntax. Typically, all of the define statements are placed in a single file with a .h extension and included as part of a program. In the examples below, I have explicitly added in define statements just for fun.

There is another way to write define statements that is often used with the ARM processors. In this second method, each peripheral is given a base address and all of the registers associated with that peripheral are given offsets from the base address. For example, GPIO port 0 will have a base address of 0x5000 and the GPIODATA register will have an offset of 0x3FFC. Other registers associated with port 0 will have different offsets. In the .h file, a structure is defined called, say PORT0. This structure has within it all of the registers that are associated with the port so that, for example, GPIODATA would be written as a variable in the structure and assigned an offset of 0x3FFC. To access the register you need to point to the structure which then needs a way to point to the offset of the register. The syntax looks like this:

```c
PORT0->GPIODATA = 1234;
```

Both of these methods are valid and both generate the same code after being compiled. The advantage of the structure method is that the code becomes more portable since all peripherals maintain the same offsets across the whole ARM family. So, I could write code for the Cortex M0 using these structures and easily run the same code on the Cortex M3 family by changing only the base addresses.

For this course we will use the more traditional define statements.
Example 1
For this example we are going to toggle pin P0_7 by alternately sending a 0 and a 1. I have added empty for
loops between toggles to slow it down. If you remove these by commenting them out, you will find the pulse
train on P0_7 runs at about 1.3 MHz.

// *** GPIO on P0.7
// Outputs pulse at about 1.3 MHz without the for loop delays
#define IOCON_PIO0_7 (*((volatile unsigned long *) 0x40044050)) // Pin control register
#define GPIO0DATA (*((volatile unsigned long *) 0x50003FFC))    // Port 0 data register
#define GPIO0DIR (*((volatile unsigned long *) 0x50008000))     // Port 0 data direction register
#define SYSAHBCLKCTRL (*((volatile unsigned long *) 0x40048080)) // System AHB clock control

int main()
{
    int i;
    SYSAHBCLKCTRL |= (1 << 6); // Enable clock for GPIO (this is not necessary)
    GPIO0DIR |= (1 << 7);      // Pin direction to output
    while(1)
    {
        GPIO0DATA |= (1 << 7); //    for(i=0;i<10000;i++);
        GPIO0DATA &= ~(1 << 7); //    for(i=0;i<10000;i++);
    }
}

Example 2
In this example we input from P0_6 and copy its data to P0.7. You can watch this in the simulator or, you can
program it on the chip and attach a 10K resistor to P0_6 to either +3.3 volts for a 1 or ground for a 0. You will
see the one or zero duplicated on P0.7.

// *** GPIO input and output
// Inputs from P0.6 and copies it to P0.7
#define GPIO0DATA (*((volatile unsigned long *) 0x50003FFC)) // Port 0 data register
#define GPIO0DIR (*((volatile unsigned long *) 0x50008000)) // Port 0 data direction register
#define SYSAHBCLKCTRL (*((volatile unsigned long *) 0x40048080)) // System AHB clock control

int main()
{
    SYSAHBCLKCTRL |= (1 << 6); // Enable clock for GPIO (this is not necessary)
    GPIO0DIR |= (1 << 7);       // Pin P0.7 direction to output
    GPIO0DIR &= ~(0 << 6);      // Pin P0.6 direction to input
    while(1)
    {
        if(GPIO0DATA & (1 << 6)) // If P0.6 is 1
            GPIO0DATA |= (1 << 7); //   make P0.7 = 1
        else                  // else
            GPIO0DATA &= ~(1 << 7); //   make P0.7 = 0
    }
}