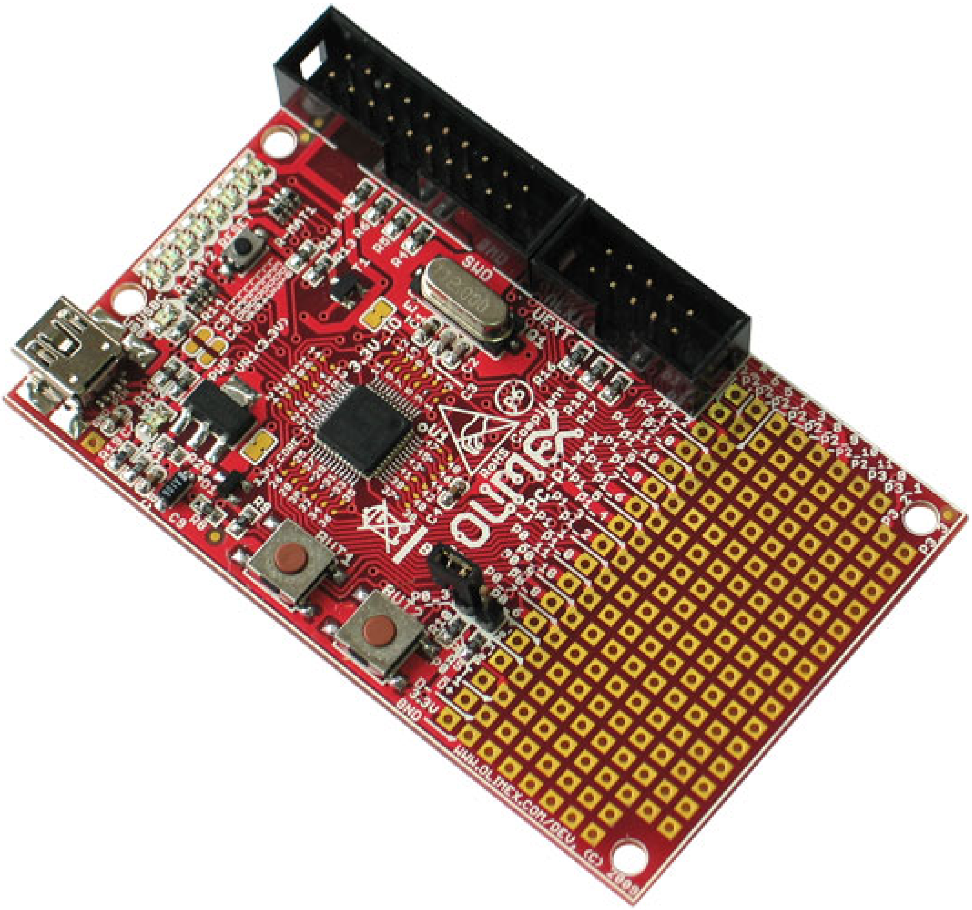
Code EE3D11

Lab Manual

Computer Architecture and Organization

Delft University of Technology



ing. X. van Rijnsoever

Delft University of Technology

Challenge the future

The MIPS Assembly Assignment is based on work by B.H.H. Juurlink
The VHDL MIPS Assignment is based on work by J.C. Verwer

Lab Manual Computer Architecture and Organization V1.3.1 (Tuesday 8th December, 2020)

This manual is typeset with LaTeX in Nimbus Roman 10 pt. The diagrams are created with Dia, the photos and screenshots are edited with GIMP and ImageMagick.

Preface

Welcome!

First of all, welcome to the Computer Architecture and Organization Lab! The aim of the Computer Architecture and Organization course is to provide an insight into the basic structure of computer systems. Furthermore, the course includes a short introduction into the C++ programming language. The lectures mainly focus on one particular architecture: the MIPS. Using MIPS assembly, the hardware/software interface is explained. In the lab you will work with the MIPS architecture as well as with the ARM microcontroller architecture. The ARM controller is programmed in the C programming language rather than assembly, but it will show you real-world implementations of a number of topics covered in the lectures and the book.

The lab provides you with some hands-on experience with all the different aspects of the course, C++, I/O hardware/software interface, interrupts, instruction set architectures, etc. In all assignments, the focus is on the course material and topics of the book.

Delft, January 2021

Ton Slats

The Lab

Time and Location

Location:	Mekelweg 4 (Building 36)
	Tellegen Hall Lab Room 1, 2 and 3 (online week 3.2 –3.5)
Time:	Thursday 13:45 – 17:30, (on campus weeks 3.6 – 3.8)

Prerequisites

The lab expects some prior knowledge:

- Basic programming skills in C;
- Basic programming skills in VHDL.

In case you have a deficiency in one of these areas, you have to acquire the required level yourself before starting the lab.

Homework Assignments

The lab contains a number of homework exercises. You need the answers to the homework assignments in the programming assignments. If you don't do the homework at home, you won't be able to finish the assignment during the lab session. Furthermore, you risk being expelled from the lab!

Lab Organization and Grading

The lab is done in couples, but graded individually. The lab is graded as pass/fail. Note that you have to pass the lab in order to make the grade of the exam valid.

The C++ part of the exam is graded pass/fail based on your solutions of the C++ lab assignments. For this reason, the C++ assignments are graded separately by PhDs.

The Manual

You will find a number of different frames throughout the manual:

Warning!
This framework warns against common errors and mistakes.

Hint!
This framework provides a hint that can be useful in addressing a problem in the assignments.

Background information!
This framework provides background information on such as practical applications, or additional functionality that is not necessary for this exercise, but perhaps could be applicable to other labs.

Software

During the lab you have to use several different software packages. The lab PCs run Windows and Linux, both systems can be used for this lab. Sometimes there are differences in the software on these operating systems, this is indicated with the following frames:

Windows
This frame contains information only of importance when using Windows.

Linux
This frame contains information only of importance when using Linux.

Due the online part of the Lab the sequence of the assignments is changed. The last assignment in the manual, VHDL MIPS Assignments is moved to week 5 see below.

Week 2	C++ Assignment 1
Week 3	C++ Assignment 2
Week 4	MIPS Assembly Assignment
Week 5	VHDL MIPS Assignment
Week 6	ARM Assignment 1
Week 7	ARM Assignment 2
Week 8	ARM Assignment 3

All the required software is already installed on the lab computers. You can of course use your own laptop to run the lab software, but make sure to have the software running *before* the actual lab session, otherwise you won't have enough time to finish the assignment. See the installation manuals on Brightspace.

Questions, remarks

Questions or remarks can be addressed to:

ing. A.M.J. Slats
Address: Mekelweg 4, room LB 01.260
Phone: +31 (0)15-27 88787
Email: A.M.J.Slats@TUDelft.nl

If you can't find me in my office, please try the Lab Support Room in the Tellegen Hall.

Contents

Part I

C++ Assignments

Chapter 1

Documentation C++ Assignments

1.1 Introduction

The first two assignments of the Computer Architecture and Organization lab are on C++. Of course, two lectures and two lab sessions are insufficient to fully discuss all the features of the C++ language. The lab focusses on the most important basics of object oriented programming (OOP).

1.2 Assignments

The subjects of the assignments are closely related to the topics treated in the book (see [2]). In the first assignment you will work on analyzing the performance of different programs running on different computers. The second assignment involves an Instruction Set Simulator (ISS).

> **Your code has to be readable!**
> Always make sure your source code is readable (properly indented, sufficiently commented) before asking an assistant for help!

1.3 Software: Code::Blocks and G++

The manual assumes you are using the Code::Blocks Integrated Development Environment (IDE) running GCC (g++). You are probably already familiar with this environment, since it is also used in the course Programming in C (part of Digital System A, EE1D11). You are of course free to use your own editor and compiler, but assistants can only help you with problems with the IDE when you are using Code::Blocks. You can find a short tutorial on creating a C++ project with Code::Blocks in Appendix A.

Chapter 2

C++ Assignment 1: Performance Analyzer

Preparations

- Read sections *Compound data types* and *Classes I* of [3]
- Read the slides of the first lecture on C++
- Read the slides of the lecture: Computer Abstractions and Technology
- Read Chapter 1 and especially §1.6: *Performance* of [2]

Objectives

- Learn to use classes and objects
- Learn to use function overloading
- Learn about analyzing performance of different computer architectures

2.1 Introduction

Chapter 1 and especially §1.6 of the Computer Architecture and Organization book ([2]) discusses performance of different computer architectures. In this C++ assignment, you will build a simple performance analyzer to analyze the performance of different implementations of an instructruction set architecture running different programs. For analyzing the performance, we need two components:

- the computer;
- the program.

Both parts will be implemented as a C++ class. For simplicity, we assume the instruction set can be divided into four classes:

- Arith, for arithmetic and logical operations;

Table 2.1: Information that needs to be stored in the class `Computer`

Type	Name	Description
`double`	`clockRateGHz`	Clock rate in GHz
`double`	`cpiArith`	CPI of instruction class Arith
`double`	`cpiStore`	CPI of instruction class Store
`double`	`cpiLoad`	CPI of instruction class Load
`double`	`cpiBranch`	CPI of instruction class Branch

- Store, for writing data to the memory;
- Load, for reading data from the memory;
- Branch, for conditional and unconditional jumps.

Different implementations of the instruction set architecture can have a different clock rate and a different CPI for each of these four instruction classes.

2.2 C++ Assignment 1.1: Class `Computer`

Objects of the class `Computer` provide an abstract representation of the computer. Table 2.1 lists the information that should be stored.

2.2.1 Header File

Create a class `Computer` in the header file `computer.h` that stores this information. Also provide the prototype of the constructor:

```
Computer (double, double, double, double, double);
```

Create a header file with header guards in Code::Blocks
You can easily create a header file in Code::Blocks with File→New→File.... Select C/C++ header in the New from template dialog and follow the steps in the wizard. You can automatically include header guards and automatically add the file to the project.

2.2.2 C++ File

Create a C++ file `computer.cpp` that contains the implementations of the member functions of the class `Computer`. Start with implementing the constructor, the arguments are given in the same order as Table 2.1.

2.2.3 Adding Functionality

In order to make the class `Computer` more useful, you have to add some additional member functions.

Table 2.2: Information that needs to be stored in the class `Program`

Instruction Class	C++ Code	Description
Arith	`int numArith`	Arithmetic and logic calculations
Store	`int numStore`	Store data into memory
Load	`int numLoad`	Load data from memory
Branch	`int numBranch`	Conditional and unconditional jumps
Total	`int numTotal`	Total number of instructions

`printStats`

Add a member function `void printStats ();` that prints the configuration of the computer. You are free to choose your output format.

> **Add the prototype of the functions to the class**
> Do not forget to add the prototype of new member functions to the class specification in the header file!

`calculateGlobalCPI`

Add a member function `double calculateGlobalCPI ();` that returns the value of the global CPI of the computer implementation.

2.2.4 Test the Class `Computer`

Modify the file `main.cpp` in the project to create two objects of the class `Computer` and use the member functions to test the class. Show the result to a TA.

> You can use the assignments in §1.13 of [2] as inspiration.

Signature TA

2.3 C++ Assignment 1.2: Class `Program`

Objects of the class `Program` provide an abstract representation of a program on the architecture of a specific computer. The instructions that make up the program are again divided into instruction classes. The class `Program` stores the number of instructions in each instruction class and additionally stores the total number of instructions. The number of instructions per class is stored in an `int` type variable. Table 2.2 summerizes the data that has to be stored in the `Program` class.

2.3.1 Header File and C++ File

Create a class `Program` in the header file `program.h` that stores this information. Also provide the prototype of the constructor:

```
Program (int, int, int, int);
```

Create a C++ file `program.cpp` that implements the constructor. The arguments are given in the same order as Table 2.2. Note that there are only four parameters, `numTotal` has to be calculated in the constructor.

In order to make the class `Program` a bit more useful, you have to add a member function `void printStats ();` that prints the specifications of the program. You are free to choose your output format.

2.3.2 Overloaded Constructor

In some of the assignments in §1.13 of [2] programs are specified by the number of instructions for each of the instruction classes. Some assignments however specify the total number of instructions and percentages for each of the instruction classes. Of course it is possible to manually convert one specification into the other, but in C++ it is possible to directly support this type of specification. This can be done using an *overloaded constructor*. Add the following (overloaded) constructor prototype in the declaration of the class `Program`:

```
Program (int, double, double, double);
```

The first argument is the total number of instructions, the next arguments are the *fractions* of Arith, Store, and Load instructions, you can calculate the number of Branch instructions with this information. Implement this overloaded constructor in the C++ file. You can assume that the fractions given as function arguments sum up to ≤ 1.

Fraction vs. Percentage
Note that the `double` type arguments of the overloaded constructor represent *fractions*, not percentages!

2.3.3 Test the Class `Program`

Modify the file `main.cpp` in the project to create two objects of the class `Program` and use the member functions to test the class. Make sure to use the two different constructors. Show the result to an assistant.

Signature TA

2.4 C++ Assignment 1.3: Performance of Programs on Computers

With the classes `Computer` and `Program` it is possible to examine the effects of changes in the computer architecture implementation on the performance for different programs.

Table 2.3: List of test computer implementations

Computer	1	2	3
Clock Rate	1 GHz	1.2 GHz	2 GHz
CPI Arith	2	2	2
CPI Store	2	3	2
CPI Load	3	4	4
CPI Branch	4	3	6

Table 2.4: List of test programs

Program	A	B	C
Arith	2000	10 %	500
Store	100	40 %	100
Load	100	25 %	2000
Branch	50		200
Total		2000	

The correct measurement of performance is execution time, since it is the only measurement that takes into account clock rate, instruction count, and CPI. Implement the following member function of class `Computer` that calculates the execution time of a `Program` in seconds:

```
double calculateExecutionTime (Program);
```

Additional functions in `program.h/program.cpp`
You might need to implement additional functions in `program.h` and `program.cpp` that provide access to private members of `program` objects.

Another common, but incorrect performance measurement is MIPS (million instructions per second). Implement the following member function of class `Computer` to calculate the MIPS rating:

```
double calculateMIPS (void);
```

One problem with the MIPS rating is that it can vary between different programs due to differences in CPI and instruction counts. However, so does execution time. In order to examine the effect of the program on the MIPS measurement, implement a member function:

```
double calculateMIPS (Program);
```

For testing, generate three computer implementations according to Table 2.3, `computer1`, `computer2`, and `computer3`. Also create three test programs according to Table 2.4, `programA`, `programB`, and `programC`. Modify your main function to calculate the performance of the computers by running all programs on each computer.

You have to output the following information:

- Computer specifications of each computer;
- MIPS rating (global value) of each computer;

- The execution time of each program for each computer;
- The MIPS rating of each computer for each program.

The code to run these simulations should be easily extensible (running more programs on more computers). This means it is not sufficient to write a seperate line of code for each simulation! You are free to choose your output format, but make sure it is easily readable.

Use the results of your simulations to answer the following questions:

- Compare the global MIPS rating to the program-dependent MIPS ratings. Comment on your findings.

  Answer: ______________________________

  ____

  ____

- The book (see §1.10 of [2]) clearifies that the MIPS rating is not a good performance indicator. Does your simulation confirm this? Explain!

  Answer: ______________________________

  ____

  ____

Show your simulation results to a TA and discuss the answers to the questions.

Signature TA

Chapter 3

C++ Assignment 2: Instruction Set Simulator

Preparations

- Read sections *Friendship and inheritance* and *Polymorphism* of [3]
- Read the slides of the second lecture on C++
- Read the slides of the lecture: Instructions: Language of the Computer
- Read Chapter 2: *Instructions: Language of the Computer* of [2]

Objectives

- Learn how to use inheritance
- Learn how to use polymorphism and abstract base classes
- Learn about the MIPS instruction set

3.1 Introduction

In this C++ assignment, you will work on a so-called Instruction Set Simulator (ISS). Such a simulator simulates the instruction set (or subset of it) of a specific processor. The simulator of this assignment will simulate a very limited subset (four instructions) of the MIPS instruction set.

3.2 Overview of the Simulator

The simulator is very simple, it has no memory, so instructions can only operate on registers. The simulator consists of two parts:

- Set of registers
- Program with instructions

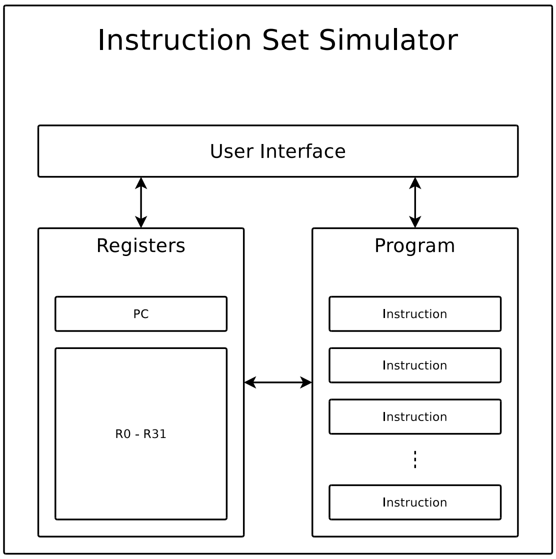


Figure 3.1: Structure of the instruction set simulator

This is shown schematically in Figure 3.1. The simulator is controlled with a very simple text-based user interface.

The registers consist of the Program Counter (PC) and a set of 32 general purpose registers.

The program is implemented as a list[1] of *instructions*.

A simple text-based user interface is used to control the simulator. With the interface a user can disassemble a program, read the contents of the registers and step through the program. Each command consists of a single character, followed by Enter The following commands are possible:

Command	Action
d	Disassemble program
e	Execute program
h	Show help screen
q	Quit
r	Print registers
s	Single step program

The disassemble (d) command will print the instructions of the program in a human readable format.

Part of the code of the simulator is already given. You can download the file `iss.zip` from Brightspace. This archive contains the following files:

[1]This is implemented with the C++ container type `<vector>`

File name	Contents
`main.cpp`	The main program. Instantiates and initializes the simulator
`instruction.h`	The supported instructions of the simulator.
`program.{h,cpp}`	The program, a list of instructions
`registers.{h,cpp}`	The registers
`simulator.{h,cpp}`	The simulator with user interface

3.3 C++ Assignment 2.1: Registers

You have to implement the class `Registers`. This class has to implement the 32 general purpose MIPS registers and the program counter (PC).

All registers can hold an `int` data type. As you know, the MIPS register $0 is special: you can write any value to this register but you will always read back 0. In order to support this behaviour, the registers should only be accessible with the following member functions:

```
void setRegister (int regNum, int value);
int getRegister (int regNum);
void setPC (int value);
int getPC ();
```

When using the simulator, it is useful to read the contents of the registers. To this end, you have to implement the member function:

```
void print ();
```

that will print the contents of the registers and the program counter on the screen. You are free to choose your output format.

Decide on the access specifiers and a suitable data structure for the general purpose registers and write the class specifications of `Registers` in the file `registers.h`. Implement the member functions in the file `registers.cpp`. Use a constructor to initialize the registers.

Write a test program to test the functionality of the class and show the result to a TA.

> **`register` is a reserved word**
> Do not use the word `register` as a variable or class name, it is a reserved word (used as a storage specifier) in C++!

Signature TA

3.4 C++ Assignment 2.2: Instructions

The simulator uses a symbolic representation of the instructions. Each instruction is implemented with a separate class (e.g. `AddInstruction`, `OriInstruction`) and each of these classes implements two operations:

- Disassembly of the instruction (member function `disassemble ();`)
- Execution of the instruction (member function `execute (Registers *);`)

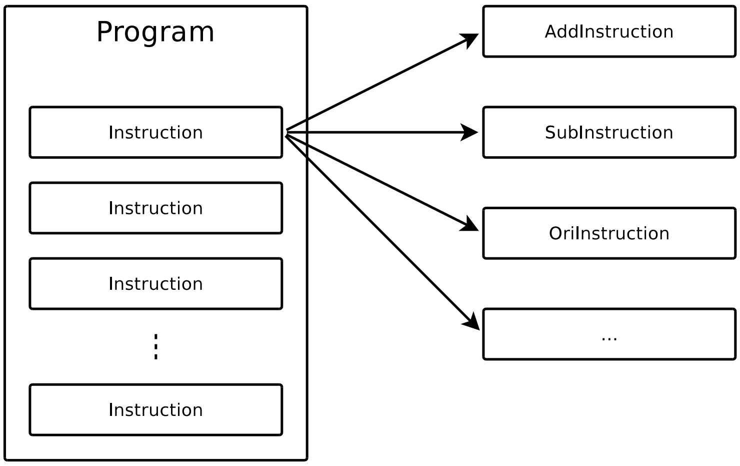


Figure 3.2: Hierarchical structure of the `Program` class

The different instruction classes are a specialization of the abstract concept of the generic instruction. A program consists of a list of instructions, each of which is an object of a specific instruction class (`AddInstruction`, `OriInstruction`, etc). However, the class `Program` only sees the generic instructions, implemented in the class `Instruction`. This relationship is shown in Figure 3.2.

C++ supports this kind of relationship with the concepts of *inheritance* and *polymorphism*. Inheritance is used to describe the hierarchical relationship: each specialized instruction *is* an `Instruction`. Polymorphism is used to extend this abstraction further: the `Program` class can disassemble an instruction via `Instruction->disassemble ()`, which will execute the `disassemble ()` member function of the specialized instruction class.

3.4.1 Class `Instruction`

Since instantiating an object of the general instruction has no meaning (what would be the meaning of executing or disassembling such an instruction?), the class `Instruction` has to be implemented as an abstract base class. Implement the class using the following information:

- Each MIPS instruction supported by the simulator has three parameters of the type `int`. Decide on the access specifiers and the data structure used to store these parameters and implement the constructor;
- Implement the destructor as a *virtual* function;
- Implement the member function **`void`** `disassemble ()` as a *pure virtual* function;
- Implement the member function **`int`** `execute (Registers *)` as a *pure virtual* function.

The `Registers *` parameter specifies the set of registers this instruction reads and writes.

Table 3.1: Supported instructions of the simulator

Class name	Disassembly	Execution
`AddInstruction`	`add $1, $2, $3`	\$1 = \$2 + \$3, PC = PC + 1
`SubInstruction`	`sub $1, $2, $3`	\$1 = \$2 - \$3, PC = PC + 1
`OriInstruction`	`ori $1, $2, 10`	\$1 = \$2 | 10, PC = PC + 1
`BrneInstruction`	`brne $1, $2, -4`	PC = PC + 1 + -4 if \$1 != \$2 else PC = PC + 1

Defining Member Functions in the Class
The code to implement each of the instructions is very small, you can choose to implement this code directly in the class declaration. Also see Section *Classes (I)* of [3].

3.4.2 Derived Classes `AddInstruction`, `SubInstruction`, and `OriInstruction`

Use the abstract base class `Instruction` to create the derived classes `AddInstruction`, `SubInstruction`, and `OriInstruction` according to the Table 3.1.

The `execute` function should return the new value of the program counter (PC). In case of these instructions, this new value is simply the current value + 1.

Note that for the actual MIPS instruction `ori` the immediate value is a zero-extended 16-bit number. For this assignment, you can assume the number is an `int`.

3.4.3 Derived class `BrneInstruction`

The MIPS `brne` instruction is a branch instruction. The `brne` instruction encodes the destination address as a number relative to the current value of the program counter + 4 (PC-relative addressing). In this simple simulator the program counter counts instructions and not actual addresses. The destination address is thus encoded as a number relative to the current value of the program counter + 1. The following example will clearify this:

```
42 loop:    ...
43          ...
44          sub    $1, $1, $2
45          brne   $1, $0, loop
```

Here the branch to the label `loop` will be encoded with the number -4:

- The label `loop` is located at instruction number 42;
- The branch instruction `brne` is located at instruction number 45;
- 42 - (45 + 1) = -4

When the branch in line 45 is taken, the `execute` member function of the `BrneInstruction` class will return 42, otherwise it will return 46 (current PC + 1). Implement the class `BrneInstruction` according to these specifications.

Write a test program to test the functionality of the class and show the result to a TA.

Signature TA

3.5 C++ Assignment 2.3: Implement and test a MIPS program

You can add instructions to a `Program` using the `appendInstruction (*Instruction)` member of the `Program` class. The simulator already loads a small program that tests all the instructions (see file `main.cpp`). For your convenience, this program is printed here in conventional MIPS notation:

```
        ori   $1, $0, 12
        ori   $2, $0, 4
        ori   $3, $0, 1
loop:   add   $4, $4, $1
        sub   $2, $2, $3
        brne  $2, $0, loop
```

- Does the symbolic representation in the C++ program match this MIPS assembly code? Explain!

  Answer: ____

- What does this program do? Verify your answer by compiling your C++ program and running the MIPS program in the simulator. Make sure your program compiles without any warnings and show the working simulator to a TA.

  Answer: ____

Signature TA

Part II

MIPS Assembly Assignments

Chapter 4

Documentation MIPS Assembly Assignments

4.1 Introduction

The next assignment of the Computer Architecture and Organization lab is closely related to the book ([2]) and is about the MIPS instruction set. The lectures of Computer Architecture and Organization have discussed the MIPS instruction set architecture. In the MIPS assembly assignments you will convert a simple C++ program into MIPS assembly and subsequently test it. This is done using the MIPS instruction set simulator `MARS`[1], a MIPS R2000/R3000 simulator. The simulator provides an editor, an assembler and a simulator with a simple set of operating system services.

4.2 Assignments

4.2.1 Register Use Convention

One of the goals of these assignments is learning to correctly use the MIPS register use convention (see Figure A.6.1 of [1]). If you do not properly use the MIPS register use convention, your assignment will not be approved!

4.2.2 Dynamic Memory Allocation

The assignments make use of dynamic memory allocation via the `new` operator. The `MARS` simulator supports this functionality with the system call `sbrk` (see Figure A.9.1 of [1] or Appendix A of the book ([2])). This system call takes as parameter the number of bytes to allocate, this means you have to convert the number of *data elements* into a number of *bytes*. The allocated memory is released with the `delete` or `delete []` operators, there is no equivalent system call in the `MARS` simulator.

Another way to implement dynamic memory allocation is creating a *stack frame*. In this case you have to release the memory by removing the stack frame.

[1] `http://courses.missouristate.edu/KenVollmar/MARS/`

4.2.3 Pseudo Instructions

The lectures mostly discuss the "real" MIPS instructions (instructions implemented in the MIPS processor). The simulator also accepts so-called *pseudo instructions*, that are translated into one or more real MIPS instructions. Examples include `bgt` (branch if greater than), `move`, etc. Using these pseudo instructions can improve the readability of your program, so use them if necessary.

4.2.4 Example Programs

The first example is a program that calculates powers:

```
# Programmer:  Mark Fienup
# Calculate Powers Example
# Algorithm:
# Main:
# maxNum = 3
# maxPower = 5
#
# CalculatePowers(maxNum, maxPower)
#
# end main
#
#
# CalculatePowers(integer numLimit, integer powerLimit)
# begin
#        integer i, j
#
#        for i := 1 to numLimit do
#              for j := 1 to powerLimit do
#                    result = Power(i, j)
#                    print i " raised to " j " power is " result
#              end for j
#        end for i
#
#
# integer Power(integer x, integer y)
# begin
#        integer result, counter
#
#        result = 1
#        for counter := 1 to y do
#              result := result * x
#        end for
#        return result
######################################################################

                .text
                j       main                    # Jump to main-routine

                .data
maxNum:         .word 3
maxPower:       .word 5

                .text
                .globl main
main:
                lw      $a0, maxNum
                lw      $a1, maxPower
                jal     CalculatePowers         # call CalculatePowers

                li      $v0, 10                 # exit system call
                syscall
endMain:

                .data
str1:           .asciiz " raised to "
str2:           .asciiz " power is "
endLine:        .asciiz "\n"

```

```
                .text
CalculatePowers:
#       Register Usage
#       $s0 contains i
#       $s1 contains j
#       $s2 contains numLimit
#       $s3 contains powerLimit
#       $t0 contains result

                addi    $sp, $sp, -20          # make room on stack for 5 registers
                sw      $s0, 0($sp)            # save $s0 on stack
                sw      $s1, 4($sp)            # save $s1 on stack
                sw      $s2, 8($sp)            # save $s2 on stack
                sw      $s3, 12($sp)           # save $s3 on stack
                sw      $ra, 16($sp)           # save $ra on stack

                move    $s2, $a0               # copy param. $a0 into $s2 (numLimit)
                move    $s3, $a1               # copy param. $a1 into $s3 (powerLimit)

for1:
                li      $s0, 1
forCompare1:
                ble     $s0, $s2, forBody1
                j       endFor1
forBody1:

for2:
                li      $s1, 1
forCompare2:
                ble     $s1, $s3, forBody2
                j       endFor2
forBody2:

                move    $a0, $s0               # call Power(i, j)
                move    $a1, $s1
                jal     Power                  # returns result value in $v0
                move    $t0, $v0

                li      $v0, 1                 # system call code for print_int
                move    $a0, $s0               # integer to print
                syscall

                li      $v0, 4                 # system call code for print_str
                la      $a0, str1              # addr. of string to print
                syscall

                li      $v0, 1                 # system call code for print_int
                move    $a0, $s1               # integer to print
                syscall

                li      $v0, 4                 # system call code for print_str
                la      $a0, str2              # addr. of string to print
                syscall

                li      $v0, 1                 # system call code for print_int
                move    $a0, $t0               # integer to print
                syscall

                li      $v0, 4                 # system call code for print_str
                la      $a0, endLine           # addr. of string to print
                syscall

                addi    $s1, $s1, 1
                j       forCompare2
endFor2:

                addi    $s0, $s0, 1
                j       forCompare1
endFor1:

                lw      $s0, 0($sp)            # restore $s0 from stack
```

```
                lw      $s1, 4($sp)           # restore $s1 from stack
                lw      $s2, 8($sp)           # restore $s2 from stack
                lw      $s3, 12($sp)          # restore $s3 from stack
                lw      $ra, 16($sp)          # restore $ra from stack
                addi    $sp, $sp, 20          # remove call frame from stack
                jr      $ra                   # return to calling routine
endCalculatePowers:


Power:
#       Register Usage
#       $a0 contains x
#       $a1 contains y
#       $t0 contains counter
#       $v0 contains result

# Since no #s registers are used and no subprograms are called, we do not need
# to save any registers!!!

                li      $v0, 1
for3:
                li      $t0, 1
forCompare3:
                ble     $t0, $a1, forBody3
                j       endFor3
forBody3:
                mul     $v0, $v0, $a0

                addi    $t0, $t0, 1
                j       forCompare3
endFor3:

                jr      $ra
endPower:
```

The second example is the `main` function of the assignment:

```
                .text
                j       main                  # Jump to main-routine

                .data
str1:           .asciiz "Insert the array size \n"
str2:           .asciiz "Insert the array elements,one per line  \n"
str3:           .asciiz "The sorted array is : \n"
str5:           .asciiz "\n"

                .text
                .globl  main
main:
                la      $a0, str1             # Print of str1
                li      $v0, 4                #
                syscall                       #

                li      $v0, 5                # Get the array size(n) and
                syscall                       # and put it in $v0
                move    $s2, $v0              # $s2=n
                sll     $s0, $v0, 2           # $s0=n*4
                sub     $sp, $sp, $s0         # This instruction creates a stack
                                              # frame large enough to contain
                                              # the array
                la      $a0, str2             #
                li      $v0, 4                # Print of str2
                syscall                       #

                move    $s1, $zero            # i=0
for_get:        bge     $s1, $s2, exit_get    # if i>=n go to exit_for_get
                sll     $t0, $s1, 2           # $t0=i*4
                add     $t1, $t0, $sp         # $t1=$sp+i*4
                li      $v0, 5                # Get one element of the array
                syscall                       #
                sw      $v0, 0($t1)           # The element is stored
                                              # at the address $t1
                la      $a0, str5
                li      $v0, 4
                syscall
```

```
                addi    $s1, $s1, 1             # i=i+1
                j       for_get
exit_get:       move    $a0, $sp                # $a0=base address af the array
                move    $a1, $s2                # $a1=size of the array
                jal     isort                   # isort(a,n)
                                                # In this moment the array has
                                                    been
                                                # sorted and is in the stack frame
                la      $a0, str3               # Print of str3
                li      $v0, 4
                syscall

                move    $s1, $zero              # i=0
for_print:      bge     $s1, $s2, exit_print    # if i>=n go to exit_print
                sll     $t0, $s1, 2             # $t0=i*4
                add     $t1, $sp, $t0           # $t1=address of a[i]
                lw      $a0, 0($t1)             #
                li      $v0, 1                  # print of the element a[i]
                syscall                         #

                la      $a0, str5
                li      $v0, 4
                syscall
                addi    $s1, $s1, 1             # i=i+1
                j       for_print
exit_print:     add     $sp, $sp, $s0           # elimination of the stack frame

                li      $v0, 10                 # EXIT
                syscall                         #
```

4.3 Software: MARS simulator

The MIPS instruction set appendix of the course book (see [1] or Appendix A of [2]) discusses the program `spim`, with the Windows version `PcSpim` and the Unix version `XSpim`. All of these versions are deprecated in favor of the program `QtSpim`[2] which is available for all supported operating systems. The program `QtSpim` is still under development and currently not very stable. For these reasons the `MARS` (Mips Assembler and Runtime Simulator) program is used in the lab.

When you run Windows on the lab PCs you can find `MARS` in the start-menu.

When you run Linux on the lab PCs you can start `MARS` from a terminal or the "run" dialog (Alt F2) with the command `mars`.

Figure 4.1 shows a screenshot of the `MARS` simulator.
The large frame (1) on the left contains two tabs:

- Edit, this is the built-in editor;
- Execute, this is the simulator.

The frame at the bottom (2) also contains two tabs:

- Mars Messages, this shows messages from the simulator

[2] `http://spimsimulator.sourceforge.net/`

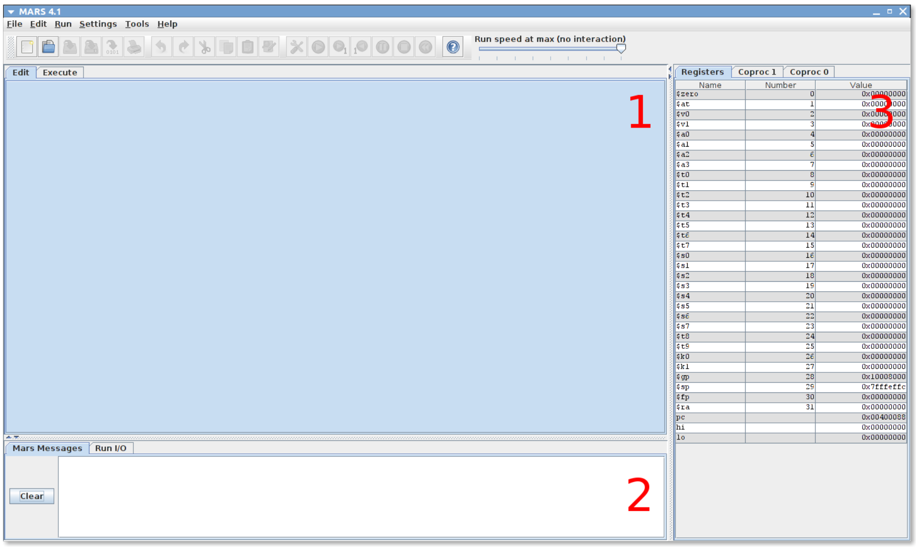


Figure 4.1: Screenshot of `MARS`

- Run I/O, this is the input/output terminal that is used to provide input and output of the program

The frame on the right (3) shows the registers of the MIPS processor, coprocessor 0 (traps and memory) and coprocessor 1 (FPU). Of these three, only the MIPS registers are used in the lab.

4.3.1 Editor

The program `MARS` contains a built-in editor featuring syntax highlighting and instruction auto-completion. Figure 4.2 shows a screenshot of the editor with an open file. The Edit menu contains several standard editor functions that can also be found on the toolbar.

4.3.2 Simulator

The simulator enables you to single step a MIPS program. Before you can use the simulator, you have to *assemble* your MIPS program. You can find the assemble option under Run→Assemble (F3). Any errors encountered during assembly are shown in the Mars Messages part of the program (see frame (2) of Figure 4.1). If the program was assembled without errors, the tab Execute will be opened. Figure 4.3 shows a screenshot of `MARS` with an open Execute tab.

The tab Execute consists of two frames:

- Text Segment, this shows the code of the program;
- Data Segment, this shows the contents of the memory.

The frame Text Segment shows the memory addresses with the numerical value in hexadecimal representation, the disassembled MIPS assembly instruction, and the corresponding line in the original source code. The disassembled MIPS instructions are

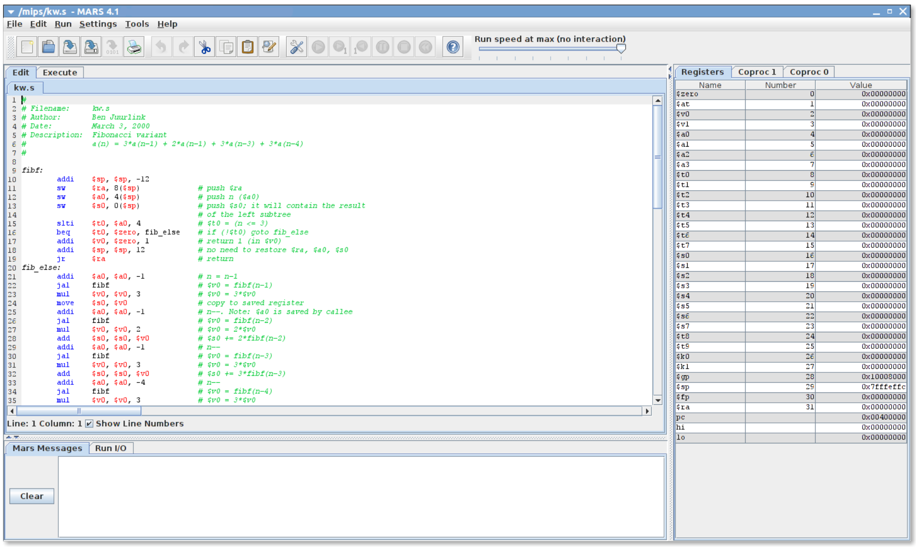


Figure 4.2: Screenshot of `MARS` with the open file `kw.s`

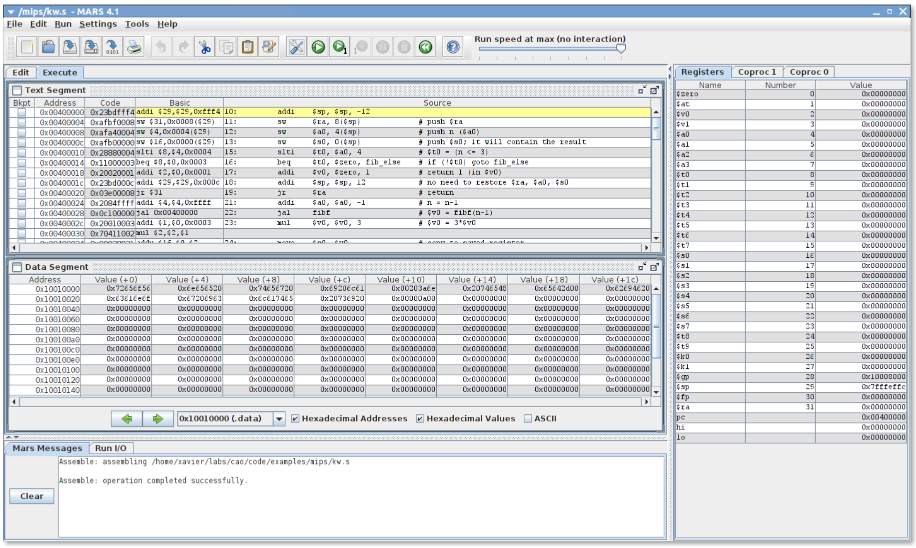


Figure 4.3: Screenshot of `MARS` with an open Execute tab

real MIPS hardware instructions, pseudo instructions are converted into one or more hardware instructions. The current instruction (pointed to by `PC`) is highlighted.

> **Automatically initialize `PC` on the address of `main`**
> By default the assembler initializes `PC` (program counter) to the address 0x00400000. This address contains the first instruction of the assembled program. If the instruction at this address is not the first instruction of the main function or a jump to the main function, your program will most likely not work as intended. You can change `PC` manually or let the assembler automatically initialize `PC` to the correct address with the option Settings→Initialize Program Counter to global 'main' if defined.

The menu Run contains different options to step through the program. You can also use the buttons in the third group on the toolbar. Breakpoints can be set by simply ticking the box Bkpt just in front of the address.

The frame Data Segment shows the contents of the memory. You can use the drop-down menu at the bottom of the frame to easily access different parts of the memory.

4.3.3 Testing the `MARS` simulator

Before you start working on the assignments, you can try the program `MARS` by opening the file `kw.s` (available on Brightspace). This is a MIPS assembly program that generates Fibonacci numbers. Test the following possibilities:

- execution of the program;
- adding and testing a breakpoint;
- single-stepping the program;

Chapter 5

MIPS Assembly Assignments

Preparations

- Read [1] or Appendix A of [2]
- Carefully study the example programs of §4.2.4
- Read all variant of the lab assignments

Objectives

- Learn to convert a simple C++ program into MIPS assembly language
- Learn to properly apply the MIPS register use convention

5.1 Introduction

You have to implement *one* of the variants of this assignment. In Brightspace Grades you see which variant you have to implement.

5.2 MIPS Assignment, Variant 1: Selection Sort

There are many sorting algorithms that can sort an array of numbers, for example the bubble sort algorthm (see §2.13 of [2]). Another algorithm is the Selection Sort algorithm. In this algorithm, the smallest element in the array is swapped with the first element of the array (located at `v[0]`). Then the algorithm finds the smallest element in the array `v[1], v[2], ..., v[n-1]` (with `n` the length of the array) and swaps this element with `v[1]`, etc. The following code is an implementation of this algorithm in C++.

Translate the C++ code into MIPS assembly code and test your program with the `MARS` simulator. Make sure to follow the MIPS register use convention (see Figure A.6.1 of [1]). Decide which registers need to be saved by the *caller* and which need to be saved by the *callee*. Do not forget to add comments to improve readability of your program.
Show your program to a TA.

Don't try to implement the entire program at once, but start with implementing and testing a single function, e.g. the `swap` function. The code of the `main` function is already given as example code and can be downloaded from Brightspace.

```
#include <iostream>

// Action: swap elements v[i] and v[j]
void swap (int v[], int i, int j)
{
	int tmp;

	tmp	= v[i];
	v[i]	= v[j];
	v[j]	= tmp;
}


// Result: index of the smallest element in the array
// v[first], v[first+1], ... , v[last]
int indexMinimum (int v[], int first, int last)
{
	int i, min, mini;

	mini	= first;
	min	= v[first];

	for (i = first + 1; i <= last; i++)
	{
		if (v[i] < min)
		{
			mini	= i;
			min	= v[i];
		}
	}

	return mini;
}


// Action: sort table a[]
void selectionSort (int a[], int length)
{
	int i, mini;

	for (i = 0; i < length - 1; i++)
	{
		mini = indexMinimum (a, i, length - 1);
		swap (a, i, mini);
	}
}


int main (void)
{
	int i, length;
	int *a;

	std::cout << "Insert the array size" << std::endl;
	std::cin >> length;

	a = new int[length];

	std::cout << "Insert the array elements, one per line" << std::endl;
	for (i = 0; i < length; i++)
	{
		std::cin >> a[i];
	}

	selectionSort (a, length);

	std::cout << "The sorted array is:" << std::endl;
```

```
        for (i = 0; i < length; i++)
        {
                std::cout << a[i] << std::endl;
        }

        return 0;
}
```

Signature TA

5.3 MIPS Assignment, Variant 2: Sort by Counting

Another simple sorting algorithm is based on the following pseudo code:

```
For every element
   Let h be the number of elements smaller than this element
   Put this element at position h in the sorted array
```

This algorithm is called sort-by-counting.

This algorithm requires an additional array, otherwise the element at position `h` will be lost. We also need to make sure not to lose elements that occur more than once. To prevent losing such elements, we define `a[i]` to be smaller than `a[j]` if and only if `a[i] < a[j]` or `a[i] = a[j]` *and* `i < j`. The following code is an implementation of this algorithm in C++.

Translate the C++ code into MIPS assembly code and test your program with the `MARS` simulator. Make sure to follow the MIPS register use convention (see Figure A.6.1 of [1]). Decide which registers need to be saved by the *caller* and which need to be saved by the *callee*. Do not forget to add comments to improve readability of your program.

Show your program to a TA.

> Don't try to implement the entire program at once, but start with implementing and testing a single function, e.g. the `countLessThan` function. The code of the `main` function is already given as example code and can be downloaded from Brightspace.

```
#include <iostream>

// Result: the number of elements smaller than a[i]
int countLessThan (int a[], int length, int i)
{
        int j, count = 0;

        for (j=0; j<length; j++)
        {
                if (a[j] < a[i] || (a[j] == a[i] && i < j))
                {
                        count++;
                }
        }
        return count;
}


// Action: sort table a[]
void sortByCounting (int a[], int length)
{
        int i, lessThan;
        int *b = new int[length];

        for (i=0; i<length; i++)
        {
                lessThan = countLessThan (a, length, i);
                b[lessThan] = a[i];
        }

        for (i=0; i<length; i++)
        {
                a[i] = b[i];
        }

        delete [] b;
}
```

```
int main (void)
{
        int i, length;
        int *a;

        std::cout << "Insert the array size" << std::endl;
        std::cin >> length;

        a = new int[length];

        std::cout << "Insert the array elements, one per line" << std::endl;
        for (i = 0; i < length; i++)
        {
                std::cin >> a[i];
        }

        sortByCounting (a, length);

        std::cout << "The sorted array is:" << std::endl;
        for (i = 0; i < length; i++)
        {
                std::cout << a[i] << std::endl;
        }

        return 0;
}
```

Signature TA

5.4 MIPS Assignment, Variant 3: Insertion Sort

Another sorting algorithm can be described as follows: we have an additional array that contains the elements already sorted so far. The array is empty at the start. For every element in the original array, we calculate the position in the sorted array and insert the element. This algorithm is called insertion sort.

Translate the C++ code into MIPS assembly code and test your program with the `MARS` simulator. Make sure to follow the MIPS register use convention (see Figure A.6.1 of [1]). Decide which registers need to be saved by the *caller* and which need to be saved by the *callee*. Do not forget to add comments to improve readability of your program.
Show your program to a TA.

> Don’t try to implement the entire program at once, but start with implementing and testing a single function, e.g. the `insert` function. The code of the `main` function is already given as example code and can be downloaded from Brightspace.

```
#include <iostream>

// Action: insert element elem in array a at positions i
void insert (int a[], int length, int elem, int i)
{
        int j;
        for (j = length - 1; j >= i; j--)
        {
                a[j+1] = a[j];
        }
        a[i] = elem;
}


// Result: smallest i for which holds: a[i] >= elem
int binarySearch (int a[], int length, int elem)
{
        int low=-1, high = length, mid;

        while (low < high - 1)
        {
                mid = (low + high) / 2;

                if (a[mid] >= elem)
                        high = mid;
                else if (a[mid] < elem)
                        low = mid;
        }

        return high;
}


// Action: sort table using the Insertion Sort Algorithm
void insertionSort (int a[], int length)
{
        int i;
        int *b = new int [length];

        for (i = 0; i < length; i++)
        {
                int position = binarySearch (b, i, a[i]);
                insert (b, i, a[i], position);
        }

        for (i = 0; i < length; i++)
        {
                a[i] = b[i];
```

```
        }

        delete [] b;
}


int main (void)
{
        int i, length;
        int *a;

        std::cout << "Insert the array size" << std::endl;
        std::cin >> length;

        a = new int[length];

        std::cout << "Insert the array elements, one per line" << std::endl;
        for (i = 0; i < length; i++)
        {
                std::cin >> a[i];
        }

        insertionSort (a, length);

        std::cout << "The sorted array is:" << std::endl;
        for (i = 0; i < length; i++)
        {
                std::cout << a[i] << std::endl;
        }

        return 0;
}
```

Signature TA

Part III

ARM Assignments

Chapter 6

Documentation ARM Assignments

6.1 Introduction

In the next three assignments of the Computer Architecture and Organization lab you will program an ARM microcontroller. This microcontroller provides a real-world example of a computer system.

The microcontroller used in the lab is the NXP LPC1343. This is an ARM Cortex-M3 microcontroller with the following specifications:

- Clock frequency of 72 MHz;
- 32 kB Flash memory;
- 8 kB RAM

The microcontroller is placed on an Olimex LPC-P1343 development board[1]. The board features leds, buttons, and access to all pins of the microcontroller.

6.2 Assignments

The microcontroller will be programmed in the C language using the Code::Blocks IDE. A special project wizard and additional tools are available for programming the microcontroller. Programming a microcontroller without an Operating System requires some in-depth knowledge of some more specialized topics of the C language. You can find this information in Chapter 7. In Chapter 8 you can find information on CMSIS, the hardware abstraction layer developed by ARM. Chapter 9 contains information on the use of interrupts. Chapter 10 contains information on the I^2C protocol. The assignments can be found in Chapters 11 – 13.

6.3 Software

The required software is already installed on the lab computers. See the installation manuals on Brightspace if you want to install the software on your laptop.

[1] `https://www.olimex.com/Products/ARM/NXP/LPC-P1343/`

6.4 Additional Documentation

During the lab you may need to consult the following additional documentation:

- User Manual of the microcontroller ([5]). You can find this User Manual on Brightspace.
- Schematic of the board. This can be found in Appendix D.
- Datasheet of the I^2C sensor and the schematic of the sensor PCB. These can be found on Brightspace.

Chapter 7

ARM Documentation: Microcontrollers and C

7.1 Introduction

With your knowledge of C from the course Programming in C (part of Digital System A, EE1D11) you can write complicated programs for a PC. Using C to program a microcontroller does require a more in-depth knowledge of some more specialized topics in C. This includes the following:

- Bitwise operations
- Pointers and structs
- Keywords `volatile` and `const`
- Handling of interrupts

Interrupts are discussed separately in Chapter 9, the other topics are discussed in this chapter.

7.2 C Standards and Data Types

There are different standards of the C language. The standard used in the lab is C-99. This standard defines, among other things, the `bool` and `uint32_t` data types. In Table 7.1 you can find an overview of the most commonly used data types in this lab.

The actual size of the default data types in C depends on the compiler. The new types `uint8_t` and `uint32_t` provide fixed-size data types, which can be very useful when programming microcontrollers.

7.3 Bits, Bytes, Words

When programming microcontrollers, the terms bit, byte and word are commonly used. The term *bit* is an abbrivation of binary digit and thus can have the value 0 or 1. The term *byte* usually refers to an 8-bit number. A byte can be used to represent $2^8 = 256$ different numbers. Another common term when working with computers is *word*. A

Table 7.1: Overview of data types

Name	Contents	Example(s)	Remarks
void	empty		Can be used as function parameter, return type, or a generic pointer type
bool	Boolean value: `true` or `false`	`true`, `false`	In header file `stdbool.h`
uint8_t	Numerical data: 0 – 255	0, 12	8-bit unsigned integer. In header file `stdint.h`
char	Single character Numerical data: -128 – 127	'a', 'A', '\0' -10, 24	
int	Numerical data: -2147483648 – 2147483647	-10, 123456	Actual range depends on hardware
uint32_t	Numerical data: 0 – 4294967295	0, 123456	32-bit unsigned integer. In header file `stdint.h`

word is the most natural datatype of the computer in use. On a simple microcontroller, the wordsize might be 8 and in that case a word thus has the same size as a byte. The microcontroller used in the lab has a wordsize of 32, the most natural datatype can thus represent $2^{32} = 4294967296$ different numbers.

In C the most natural datatype is represented with an `int`, with the exception that an `int` is at least 16 bits wide. This means that for an 8-bit platform, an `int` is a 16-bit datatype, while on the LPC1343 an `int` is a 32-bit datatype.

7.3.1 Bit Numbering in a Word: LSB and MSB

The bits in a word (or any other multi-bit entity) are numbered: the right-most bit is bit 0, the bit left of that one is bit 1 etc. The right-most bit is refered to as Least Significant Bit (LSB), the left-most bit is refered to as Most Significant Bit (MSB). These terms refer to the impact a change in such a bit has on the numerical value of the word. A change in the value of the LSB (bit 0) has the least effect on the numerical value of the word. On the other hand, a change in the left-most bit (MSB) leads to the largest change in numerical value of the word. In Figure 7.1 the bit numbering of the bits in a byte and the location of the MSB and the LSB are shown.

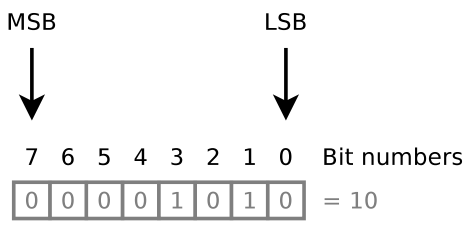


Figure 7.1: Bit numbering in a byte

7.3.2 Byte Numbering in a Word: Endianness

The word size of a computer can be much larger than a byte. The ARM Cortex-M3 microcontroller has a word that contains four bytes (32 bits). Memory is usually addressed with byte addresses, this means there are two schemes to store a multi-byte unit, *little endian* and *big endian*:

- Store the byte with the highest byte-number on the lowest address (big endian)

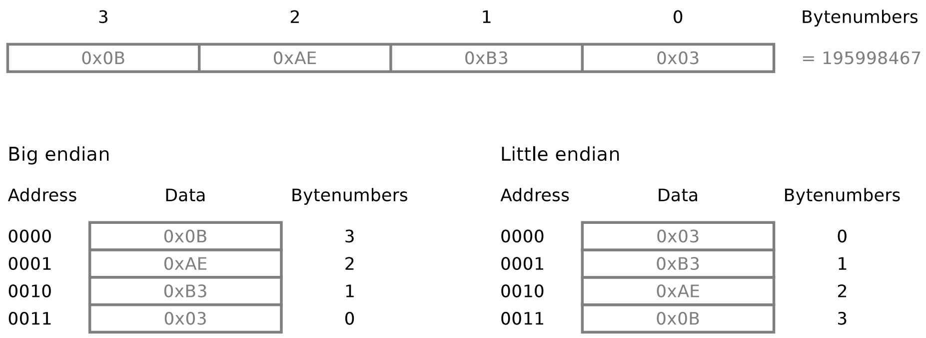


Figure 7.2: Big endian and little endian storage schemes

Table 7.3: Logical operators supported in C

Operation	C Symbol	Use to
NOT	~	Invert a word (unary operator)
OR	|	Selectively set a bit
AND	&	Selectively clear a bit
XOR	^	Selectively invert a bit

- Store the byte with the lowest byte-number on the lowest address (little endian)

This is graphically shown in Figure 7.2. When communicating with other computer-systems in multi-byte units, it is important to understand the concept of endianness. The ARM Cortex-M3 can be implemented with both little endian or big endian storage scheme, the LPC1343 uses the little endian storage scheme only.

7.4 Bitwise Operators

Bitwise operators are used for setting specific bitpatterns. Bitwise operators work, contrary to what their name might suggest, on multibit datatypes like bytes and words. However, the operations they perform, only act on individual bits in matching bit positions. The result of a bitwise operator acting on two words can thus be found by performing the operator on the corresponding individual bits of the words.

Boolean operators
Besides bitwise operators, C also has boolean operators. Boolean operators work on boolean data that have the value `true` (defined as any value unequal to 0) or `false` (defined as 0). Although the naming and the symbols are similar, they have a very different meaning!

7.4.1 NOT, OR, AND, XOR Operators

Table 7.3 shows the symbols used in C for NOT, OR, AND, and XOR operators.

7.4.2 Shift Operators

Shift operators move bit-patterns. There are two types of shift operators:

- left shift operator (<<)
- right shift operator (>>)

The shift operator is a binary operator and shifts the first operand with the number of bits specified in the second operand.

The left shift operator has the mathematical effect of multiplying the first operand with $2^{\text{second operand}}$ (in case no overflow occurs). This is shown graphically in Figure 7.3. Bits that are shifted out are simply lost. The left shift operator shifts in bits on the right with the value 0.

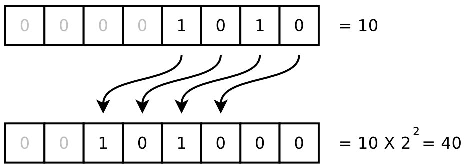


Figure 7.3: The effect of the left shift operator (left shift by 2)

The right shift operator has the mathematical effect of dividing the first operand with $2^{\text{second operand}}$. In this case the behaviour depends on whether the number that is shifted is an unsigned or a signed number. In case of an unsigned number, the right shift operator shifts in bits with the value 0. In case of signed numbers, bits with the value of the original sign bit are shifted in. This is shown in Figure 7.4.

> **Shift right with signed numbers**
> The C standard only specifies the behaviour of the shift right operator when the first operand is an unsigned or a non-negative signed number. For many platforms (including the LPC1343), however, the right shift operator works on negative signed numbers as described here.

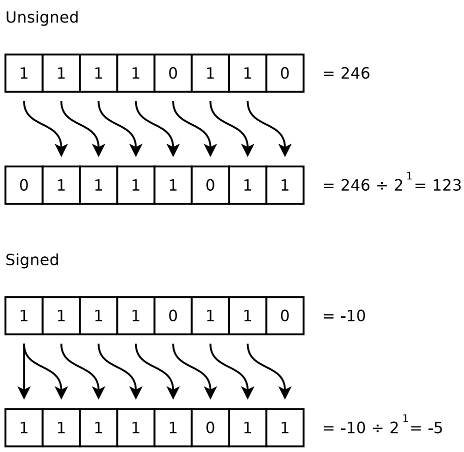


Figure 7.4: The effect of the right shift operator for unsigned and signed numbers

Although in some situations you really only need to shift bit-patterns, in many cases the shift operator is more useful in combination with the bitwise operators, as shown in the following example:

Example: set bits 2 and 5 in a variable named d:

```
d = d | (1<<5) | (1<<2);      // = d | 0x20 | 0x04
```

7.5 Keywords `volatile`, `const` and `static`

7.5.1 `volatile`

The keyword `volatile` is used for variables. It is used to prevent the compiler from optimizing away actions on variables. An example:

```
int i = 100000;
while (--i) { }
```

This code tries to implement a simple busy waiting loop. Unfortunately, the compiler reduces the loop to ... nothing! The compiler sees a variable `i` that will have a value of 0 after the loop and a loop that doesn't perform any work (other than decrementing the variable). Since the loop isn't doing any work and the variable `i` has a known value afterwards, the compiler concludes the loop can be removed. Although in many cases this is exacly the behaviour we would like, it might pose a problem in some situations. The busy waiting loop is a bit of a contrived example, but a similar situation can occur when a piece of code has to wait for a variable to be set by an external piece of hardware. Many of the peripherals of a microcontroller set some kind of flag to notify it has finished. For example, a program can use an ADC (analog digital converter). It can contain a loop that has to check a flag to see if the converter has a valid result, then has to fetch the result, clear the flag and starts the loop again. In code it could look like this:

```
1 while (1)
2 {
3         ADC_done = 0;                // Clear flag to start a conversion
4         while (ADC_done == 0)        // Wait until a conversion is finished
5         {
6                 ...          // Fetch the result, do something useful
7         }
8 }
```

In this case the compiler sees a `while`-loop that checks a flag (line 4) that was set to 0 (see line 3). This means the compiler will replace this code with a simple endless loop. In this case, however, the flag is set by an external piece of hardware (the ADC) that the compiler has no knowledge about.

These examples show that we need a way to tell the compiler that it shouldn't optimize away certain variables. This is done with the keyword `volatile`. The keyword `volatile` tells the compiler the variable can be changed outside of the scope of the variable.

The following, slightly changed, code will implement a busy waiting loop:

```
volatile int i = 100000;
while (--i) { }
```

Flags and variables that are changed externally by built-in peripherals are already declared `volatile` in CMSIS (see Chapter 8).

7.5.2 `const`

The keyword `const` is the counterpart of `volatile`: it defines that a variable cannot be changed. The compiler will generate an error when code tries to change the value

Variable	Address	Data
i	0x24	0x9A
a	0x25	0x30
*p	0x26	0x24
	0x27	...
	0x28	...
	0x29	...

Figure 7.5: Variable names, addresses and data for normal and pointer variables. All memory-locations and addresses are assumed to be bytes.

of a `const` variable. When the target platform is a microcontroller, the use of `const` can influence the memory location a variable is assigned to. Most microcontrollers are equiped with read-only program memory (e.g. flash memory) and read/write RAM. The values of initialized global (or static) variables are stored in the image that is programmed onto the microcontroller. On startup, these values are copied to the RAM, since variables can be changed. However, when a variable is declared with the `const` keyword, the variable is not copied on startup, since it is known that its value can not change. This saves RAM and can be very useful for things like lookup tables.

7.5.3 `static`

The keyword `static` can be used for functions or variables. When the `static` keyword is used for global functions or variables, it limits access to those identifiers to the file in which they are specified. Thus a `static` declared function cannot be called from another C-file.

Local variables that are declared `static` keep their value in between function calls. These variables behave like global variables, but are only accessible from within their own scope. Using `static` variables can be very helpful when writing interrupt routines.

7.6 Pointers and Stucts

Pointers and structs are two important concepts in the C language. Pointers point to variables and allow for indirect access to such variables, structs provide a way of grouping related information. This paragraph will briefly discuss both concepts and also discuss the specific situation when you have a pointer to a struct.

7.6.1 Pointers

Declaring normal variables

Normally in C, you define a variable to have a name, a type and a value. The type is usually something like an `int` or an `uint32_t`. The value of the variable is user-defined, but limited by the datatype. The compiler usually assigns a location in the computer's memory to the variable, the *address*. The value of the variable is stored at that memory location. This is shown in Figure 7.5. In the figure the variable `i` is assigned to location 0x24 and has the value 0x9A, variable `a` is assigned to location 0x25 and has the value 0x30. The following code could lead to the result shown in Figure 7.5:

```
uint8_t i = 0x9A;
uint8_t a = 0x30;
```

Declaring pointer variables

The variable `p` is a special case, it is a *pointer*. The variable `p` by itself is just a variable, in this case assigned to address 0x26 and with a value of 0x24. However, since the variable is a pointer (as denoted by the `*`), this value 0x24 has a special meaning: it is an address, in this case the address of variable `i`. The following code could lead to this situation:

```
uint8_t *p = &i;
```

This statement assigns the address of variable `i` (as denoted by `&i`) to the variable `p`. This variable is a pointer to a variable that contains an `uint8_t`. The so-called *address operator* `&` returns the address of a variable.

> **`uint8_t* p` or `uint8_t *p`?**
> Since the variable `p` is said to be "a pointer to an `uint8_t`", one might be temped to specify the data type as `uint8_t*`. Although it is syntactically correct (one could even leave out the space!), it can easily lead to problems when declaring more than one variable on a single line:
>
> `uint8_t* p, p2;`
>
> This code will declare the variable `p` to be of the type `uint8_t*`, while variable `p2` is assigned the type `uint8_t`! The following code will correctly declare both variables:
>
> `uint8_t *p, *p2;`

Using pointer variables

A pointer variable can be used to read and write the value of the variable it points to. This means the value of the pointer variable (0x24 in case of pointer `*p` of Figure 7.5) is used as an address. Then this address is used to access the variable pointed to. In C this is done using the *dereference operator* `*`. The following code can be used to change the value of variable `i` using the pointer `*p`:

```
*p = 0x10;
```

Now the value of the variable `i` is 0x10.

> **Why use pointers?**
> Instead of using the pointer `*p`, you could also directly change the value of the variable `i`. So why would pointers be usefull? Pointers are, for example, very usefull when you need to change the value of a parameter of a function. Function parameters are copies of the original variables, changing a parameter will not change the original variable. By passing the parameter in the form of a pointer, the function can change the original variable. Pointers are also used in datastructures like lists and trees.

7.6.2 Structs

Defining a struct

A struct defines a new datatype rather than a variable. A struct is used to group information. For example, when you want to store information on a person, you might want to store that persons name, surname and birthdate etc. This can be done in a struct:

```
struct person
{
        char *name;
        char *surname;
        int year_of_birth;
        int month_of_birth;
        int day_of_birth;
};
```

The variables declared inside the struct are called *members*.

Declaring and initializing a struct-type variable

With the previous declaration of the `person` struct, you can create variables of this new type as follows:

```
struct person person1;
```

When you want to initialize a struct, you can provide initialization-values for all the members by placing them between curly-braces and separating them with a comma:

```
struct person person2 = {"John", "Doe", 2000, 1, 1};
```

Using members of a stuct

Accessing members of a struct is done using the dot-operator:

```
// person2 is of type 'struct person'
person2.year_of_birth = 1999;
```

Using members of a struct via a pointer

It is also possible to define a pointer that points to a variable of the new struct type:

```
struct person person3;
struct person *p = &person3;
```

Accessing a member of the struct pointed to by `*p` requires some special care: the dot-operator has a higher precedence level than the dereference-operator. This means that the following code won't work:

```
*p.year_of_birth = 1999;
```

This will assume `p` is struct containing a member `year_of_birth`, which is a pointer to some numerical datatype. In order to enforce the preferred precedence, you have to add braces:

```
(*p).year_of_birth = 1999;
```

This will make sure `*p` is seen as a pointer to a struct type that has a member `year_of_birth`. Since it is very common in C to access members of a struct via a pointer, a special operator is provided for this purpose: `->`. With this operator, the code changes into this:

```
p->year_of_birth = 1999;
```

7.7 Exercises

The following exercises allow you to test your knowledge of the use of bitwise operators. You're not required to do these exercises. However, the level of the exercises is comparable with the required level in the lab assignments. What is the result of the following calculations:

- (~0x04) | (1<<3)

  Answer: ______________________________

- (−10>>2) & (0x03^2)

  Answer: ______________________________

What combination of bitwise operators would you use to perform the following tasks:

- Clear bits 1 and 2, toggle bits 3 and 4.

  Answer: ______________________________

- Copy the value of bit 1 into bit 0.

  Answer: ______________________________

- Multiply a byte by 4.

  Answer: ______________________________

Given the following piece of C code:

```
int x = NUMBER1;
int y = NUMBER2;
x = x ^ y;
y = y ^ x;
x = x ^ y;
```

Can you say anything about the values of `x` and `y` afterwards?

Answer: ______________________________

Given the data definitions:

```
struct t
{
        int     a;
        int     b;
};

struct t a = {1, 2};
struct t *p = &a;
```

What is the result of running the following piece of C code:

```
a.a = 2;
int c = p->a;
(*p).b = 3;
int d = a.b;
```

Answer: ____

____

____

____

Chapter 8

ARM Documentation: CMSIS

8.1 Introduction

ARM doesn't actually build microcontrollers or microprocessors, instead the company licences other companies the design of the ARM architecture and instruction set. The specifications of the ARM design strictly describe the instruction set and the register names, but allow, for example the memory addresses of these registers to be chosen by the manufacturer. This allows different manufacturers to optimize their microcontroller in the way they want to, but makes it difficult for a programmer to switch from an ARM microcontroller build by company A to an ARM microcontroller build by company B.

In order to overcome these problems in case of the Cortex-M line of microcontrollers, ARM has developed CMSIS, Cortex Microcontroller Software Interface Standard[1].

8.2 Versions

There are several versions of CMSIS, the current version is 5.0. The latest version of CMSIS available from NXP is 3.2, which is used in the lab.

8.3 Overview

CMSIS v2 consists of three parts:

- Core Peripheral Access Layer
- Device Peripheral Access Layer
- DSP library

The Core Peripheral Access Layer is independent of the silicon supplier. It provides easy access to registers that are available in every Cortex-M3 implementation and provides a high-level interface to usefull assembly instructions as inline functions.

[1] http://www.arm.com/products/processors/cortex-m/cortex-microcontroller-software-interface-standard.php

The Device Peripheral Access Layer is supplied by the silicon manufacturer. It provides an abstraction layer to the different peripherals in the specific implementation of the Cortex-M3.

The DSP library provides access to a number of mathematical routines and transformations. It is designed for the Coretex-M4 family, which has hardware floating point operations and SIMD[2] instructions, but provides a software emulation for the Cortex-M3. This part of CMSIS will not be used in the lab.

8.4 Files

CMSIS (without the DSP library) consists of 6 C and header files, for the LPC1343 these are:

- `core_cm3.h` contains the core registers
- `core_cmFunc.h` contains functions for accessing some otherwise unavailable core registers
- `core_cmInstr.h` contains functions for using some assembly instructions
- `system_LPCxx.h / system_LPC13xx.c` contains the systemclock initialization
- `LPC13xx.h` contains the device specific registers and interrupts

When using the Code::Blocks LPC1343-wizard (see Appendix B), these files are copied and added to a new project automatically.

8.5 Register Naming

For many microcontroller platforms, the peripheral registers[3] are mapped onto the names used in the reference material with a huge list of `#define` statements. CMSIS uses a somewhat different approach: registers are grouped together in structs, based on functionality. The individual registers are mapped onto members of these structs. For example, all registers having to do with the GPIO functionality of port 1 are grouped in a struct called `LPC_GPIO1`. To access the `DATA` register of this port, one can use the following code:

```
LPC_GPIO1->DATA = 0x01;
value = LPC_GPIO1->DATA;
```

The CMSIS names are not named explicitily in the User Manual ([5]). They can, however, easily be derived from the names used in the User Manual. For example the name for the data-register of port 1 as used in the User Manual is `GPIO1DATA`.

The lab manual will usually explicitily state the CMSIS names of the registers that should be used in a specific assignment.

8.6 Functions

For most of the lab assignments, there is no need to use the functions provided by the different CMSIS files. The `SystemInit` function used to setup the system clock PLL

[2]see `http://en.wikipedia.org/wiki/SIMD`

[3]These registers are used to store configuration or state data of the build-in peripherals

Table 8.1: Interrupt numbers, names and flags

Interrupt number	CMSIS name	Description
0 – 39	`WAKEUP?_IRQn`	Start logic wake-up interrupts
40	`I2C_IRQn`	I^2C interrupt (I2C0)
41	`TIMER_16_0_IRQn`	Timer interrupt for 16-bit timer 0 (TMR16B0)
42	`TIMER_16_1_IRQn`	Timer interrupt for 16-bit timer 1 (TMR16B1)
43	`TIMER_32_0_IRQn`	Timer interrupt for 32-bit timer 0 (TMR32B0)
44	`TIMER_32_1_IRQn`	Timer interrupt for 32-bit timer 1 (TMR32B1)
45	`SSP0`	Synchronous Serial Port 0 interrupt (SSP0)
46	`UART_IRQn`	Universal Asynchronous Receiver and Transmitter interrupt (UART)
47	`USB_IRQn`	Universal Serial Bus low priority interrupt (USB)
48	`USB_FIQn`	Universal Serial Bus high priority interrupt (USB)
49	`ADC_IRQn`	Analog-to-Digital Converter interrupt (ADC)
50	`WDT_IRQn`	Watchdog Timer interrupt (WDT)
51	`BOD_IRQn`	Brown Out Detect interrupt
52	`RESERVED_IRQn`	Reserved
53	`EINT3_IRQn`	PIO_3 interrupt (GPIO3)
54	`EINT2_IRQn`	PIO_2 interrupt (GPIO2)
55	`EINT1_IRQn`	PIO_1 interrupt (GPIO1)
56	`EINT0_IRQn`	PIO_0 interrupt (GPIO0)
57	`SSP1_IRQn`	Synchronous Serial Port 1 interrupt (SSP1)
58	`RESERVED_IRQn`	Reserved

is already used in the startup code. When the use of one of these functions is required in the lab, it will be stated in the assignment.

8.7 Support for Interrupts

CMSIS comes with a number of functions and constants that ease working with interrupts.

8.7.1 Interrupt Constants

CMSIS specifies names for the different interrupt numbers. The interrupt number, names and flags are listed in Table 8.1 and in Appendix C.

8.7.2 Interrupt Support Functions

CMSIS provides a number of functions to enable or disable interrupts, set or clear pending bits and read or write the priority bits.

Enable or Disable Interrupts

- **`void`** `NVIC_EnableIRQ (`*`name`*`)`
- **`void`** `NVIC_DisableIRQ (`*`name`*`)`

Get or Set Pending State

- **`uint32_t`** `NVIC_GetPendingIRQ (`*`name`*`)`
- **`void`** `NVIC_SetPendingIRQ (`*`name`*`)`
- **`void`** `NVIC_ClearPendingIRQ (`*`name`*`)`

Get or Set Priority

- **`uint32_t`** `NVIC_GetPriorityIRQ (`*`name`*`)`
- **`void`** `NVIC_SetPriorityIRQ (`*`name`*`,` **`uint23_t`** `priority)`

Chapter 9

ARM Documentation: Interrupts

9.1 What Are Interrupts

An interrupt is an interruption of the normal program flow, mostly in reaction on an external event. Normally, a program will execute sequentially: it will start at the first instruction and then go to the next, and the next, etc. If the program contains loops or functions, the next instruction may not be on the next address, but the program still executes all instructions in a predefined order. An interrupt, however, will temporarily stop the normal program flow, execute a special function called *interrupt service routine* (ISR) and then resume execution of the original program. Interrupts can occur at any place in a program, so they have to execute transparantly.

> Interrupts can interrupt programs, but not instructions.

9.2 Interrupts on the Cortex-M3

On the Cortex-M3, interrupts are handled by the Nested Vector Interrupt Controller (NVIC). The NVIC will interrupt the normal execution for events like timer matches, serial transfer failures etc. By default, no interrupts will be generated, interrupts will have to be enabled in specific registers. When an interrupt occurs and is enabled, the NVIC executes the ISR. The Cortex-M3 reserves a part of the memory for the so-called vector table. Each interrupt is mapped onto a specific entry in this table. The table contains the addresses of the ISRs for the different interrupts.

9.3 Advantages of Using Interrupts

Up until now, all programs that had to wait for something to happen were implemented using polling: they endlessly checked for a specific condition. Although this works fine, this method has some drawbacks:

- Checking for multiple events always takes place in a fixed order
- You can not easily prioritize events

If, for example, your program performs a time-consuming calculation, but it should report it's progress every second, you could use a timer and then continuously check the timer flag. This would of course slow down the calculation speed, or may not even be possible in a simple way at all. Or maybe your program has to handle two events, one low priority event that occurs often, and one high priority event that happens only every once and then. In a main-loop you can easily check for both events, but if you're currently handling the low priority event and then the high priority event occurs, you will have to finish the low priority event first. In both cases using interrupts can help: when the event of interest occurs, the program will automatically exectute the code to process that event.

9.4 Software Support for Using Interrupts

The use of interrupts is partially supported in CMSIS, in Chapter 8 the functions and constants for use with interrupts are discussed. The names used for the interrupt handlers are defined in the startup code included by the Code::Blocks wizard. These names along with their interrupt number, CMSIS name, and a short description can be found in Appendix C.

Interrupt Service Routine Naming
If the names of the interrupt service routines do not exactly match the names mentioned in Appendix C, the compiler won't give a warning (you've just created another function), but the routine will not used!

9.5 Example

The following code is an example of using the ADC with interrupts. In this example, the result of the AD conversion is dumped to the 8 LEDs on the board. Note the use of the CMSIS function `NVIC_EnableIRQ ()` and the code used to define the interrupt handler.

```
#include <stdint.h>
#include "LPC13xx.h"

#include "leddriver.h"
#include "delay.h"

void init_adc (void)
{
        // Select AD function for PIO0_11
        LPC_IOCON->R_PIO0_11 &= ~(1<<7);
        LPC_IOCON->R_PIO0_11 |=  (2<<0);

        // Disable power down
        LPC_SYSCON->PDRUNCFG &= ~(1<<4);
        // Enable ADC clock
        LPC_SYSCON->SYSAHBCLKCTRL |= (1<<13);

        // SEL        = 1
        // CLK_DIV    = 15     72 MHz/(15+1) = 4.5 MHz
        // BURST      = 1
        // START      = 0
```

```
        // EDGE           = 0
        LPC_ADC->CR = (1<<0) | (0x15<<8) | (1<<16);

        // Enable interrupts only for conversion on channel 0
        LPC_ADC->INTEN = (1<<0);

        // Enable interrupts for ADC using CMSIS function
        NVIC_EnableIRQ (ADC_IRQn);
}


/*
 * This is the Interrupt handler. Make sure the name of the function
 * exactly matches the name mentioned in the table of Appendix C!
 * When the ADC interrupt is enabled, this function is called auto-
 * matically when the ADC has completed a conversion.
 */
void ADC_Handler (void)
{
        // Put the 8 most significant bits of the converted result
        // on the LEDs
        leds_on (LPC_ADC->DR0>>8);
}


int main (void)
{
        init_leds ();
        init_adc ();

        // Wait for an interrupt...
        while (1) { }
}
```

Chapter 10

ARM Documentation: I^2C

10.1 Introduction

Many modern sensors are provided with a build-in controller that performs some signal processing, for example to minimize noise or to linearize a non-linear sensor output. Many of these sensors implement the I^2C (Inter-Integrated Circuit, also commonly known as two-wire interface) protocol. In the lab you will work with a sensor with such an I^2C interface.

10.2 I^2C hardware

The I^2C protocol provides a serial master-slave interface that supports multiple masters and slaves on a single bus. The bus consists of two open-drain[1] lines: SDA (Serial DAta) and SLC (Serial CLock). The lines are pulled up with resistors to a V_{dd} of 5 V or 3.3 V. This maximum voltage can differ for different types of sensors! The value of the resistors is usually in the range 1.8 kΩ – 47 kΩ. A diagram of the connections is shown in Figure 10.1.

Due to the open-drain design, both SDA and SCL start with a high value. In a normal data transfer, SDA isn't allowed to change during the time SCL is high. So in

[1] An open-drain configuration has an output stage with a MOSFET of which the source is connected to ground while the drain is left floating

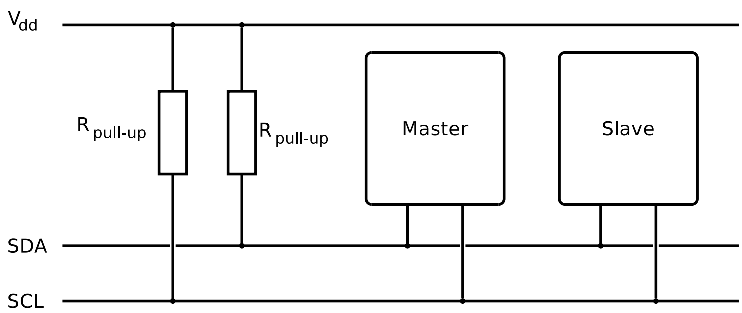


Figure 10.1: Diagram of an I^2C bus with a single master and a single slave

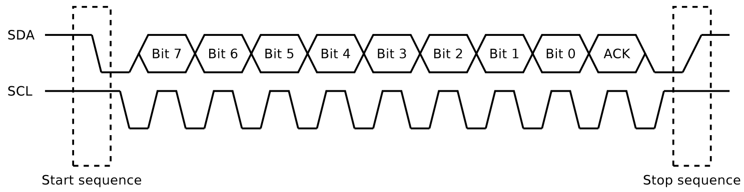


Figure 10.2: Waveforms of a single I^{2}C transfer

Table 10.1: Relationship between the slave address and the read/write addresses

	Binary	Hexadecimal
7-bit slave address	01001000	0x48
8-bit write address	10010000	0x90
8-bit read address	10010001	0x91

order to send a bit, SCL is pulled low, then SDA is set to the value to be transmitted. When SCL goes high again, the bit is clocked in by the receiver. There are two special sequences in which this condition is violated, the start sequence (in which SDA goes from high to low while SCL is high) and the stop sequence (in which SDA goes high when SCL is high). I^{2}C transfers take place in packages of 8-bits and are followed by an ACK (acknowledge) bit from the receiving side. The MSB is send first. The waveforms of an I^{2}C transmission are depicted in Figure 10.2.

The standard speed for I^{2}C is 100 kHz. Newer devices support frequencies up to several MHz. In the lab we will use the standard speed of 100 kHz, since it is supported by virtually every I^{2}C device.

10.3 I^{2}C software

The I^{2}C bus supports multiple slaves on the same two lines. Slaves have a 7-bit address[2] that are used to select a specific device. The bus also supports multiple masters (via an arbitration scheme), but this is not used in the lab. When a slave address is send over the bus, it is send in the first 7 bits, bit 8 indicates whether there is a write request (0) or a read request (1). This means that the 7-bit slave address can be interpreted as two 8-bit addresses: one (even) address for writing and one (odd) address for reading. Table 10.1 gives an example.

Each node on the bus can support one or more of four different transfer schemes possible:

- master transmit: the master transmits data to the slave
- master receive: the master receives data from the slave
- slave transmit: the slave transmits data to the master
- slave receive: the slave receives data from the master

In the lab the LPC143 is only used as a master, so only the master transmit and master receive modes are used.

[2] I^{2}C also supports 11-bit addresses, but this is not used in the lab

Transfers are always initiated by a master. A master initiates a transfer with a start sequence.

10.3.1 Master Transmit Mode

In master transmit mode, the master sends data to a slave. The master starts by transmitting a start sequence, followed by the address of the slave (the write address, with the last bit 0). Then the databytes can be send. The actual meaning of the databytes depends on the slave device, but in most cases the first byte send is a register number of the slave device you want to write to. When all bytes have been send, the master can write a stop sequnce. The slave answers each transfer with an ACK of 1 when it can receive more bytes, or an ACK of 0 when it can not receive more bytes. So writing takes the following steps:

- Send a start sequence
- Send the write address of the slave device
- Send the register number
- Send the databyte(s)
- If you don't want to do other actions, send a stop sequence

10.3.2 Master Receive Mode

In master receive mode, the master recieves data from the slave. Since a transfer can only be initiated by a master, master receive mode actually starts identical to master transmit mode and thus begins with a start sequence. In most cases, you need to specify which register of the slave device you want to read, this is done by issuing a write to the slave device, so after the start sequence the slave write address is send, followed by the register number. In order to switch to reading that register, the master has to send a start sequence again, this is known as a repeated start. Now the read address of the slave device can be send, after which the slave will send the data bytes. The master has to answer each byte with an ACK of 1 to signal it can receive more bytes, or an ACK of 0 to signal that was the last byte and finally finish the transmission by sending a stop sequence. So reading takes the following steps:

- Send a start sequence
- Send the write address of the slave device
- Send the register number
- Send a start sequence (repeated start)
- Send the read address of the slave device
- Read the databyte(s)
- Send an ACK bit after every byte if you want to read the next byte of a NACK if you don't want to read the next byte
- When all bytes are received and you don't want to do other actions, send a stop sequence

10.4 I^2C on the LPC1343

The LPC1343 has extensive support for the I^2C bus, the details can be found in Chapter 13 of [5]. It has support for a master device and up to four slave device addresses. The intended way of operating the I^2C bus is with a state machine in an interrupt routine. The I^2C module has a status register `I2C0STAT` that contains the status of the I^2C bus. It is, however, also possible to use polling on the interrupt flag SI.

Chapter 11

ARM Assignment 1: Simple Input and Output

Preparations

- Read Chapter 7 of this manual
- Do the homework assignments of §11.3.1 and §11.5.1

Objectives

In this session you will

- Familiarize yourself with the programming environment
- Learn to use the LEDs and the buttons
- Work with bitwise operators

11.1 Introduction

In this first assignment you will write some simple programs that use the LEDs and the buttons for input and output.

11.2 ARM Assignment 1.1: The Programming Environment

Before you can do any real work, you have to familiarize yourself with the basic functionality of the programming environment. In Appendix B you will find a tutorial of Code::Blocks and how to program the microcontroller board. Create a new LPC1343 project according to the steps described in this tutorial. Make sure to also flash the microcontroller. Although the program doesn't do any real work, this way you can test whether or not the microcontroller board functions correctly.

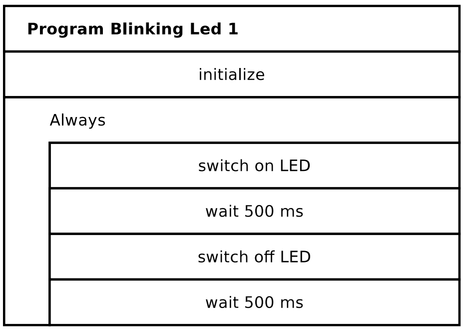


Figure 11.1: PSD of the blinking LED program

11.3 ARM Assignment 1.2: Simple Output

As a simple start, the first real program is a blinking LED program that will blink a single LED with a frequency of 1 Hz. The PSD of this program is given in Figure 11.1.

As can be seen from the PSD, the program consists of four different steps, some of which are repeated in an endless loop:

- initialize
- turn LED on
- wait 500 ms
- turn LED off

Each of these steps translate to one or more lines of C-code. This program directly manipulates hardware (the LED) connected to the microcontroller. You need to know to which pin this LED is connected and if and how to switch the LED on or off. This information can be found in the schematic of the development board (see Appendix D) and the user manual of the microcontroller (see [5]).

11.3.1 Homework Assignments

Answer the following questions:

- Use the schematic of the board (see Appendix D) to find out which pin of the microcontroller is connected to `LED0`.

  Answer: ______________________________

- Is the LED lit by writing a 0 or a 1 to that pin? Explain your answer.

  Answer: ______________________________

  ____

- For writing to the pin, two registers are important, the direction register and the data register. Like all the configuration registers of the Cortex-M3, these registers are mapped onto memory addresses. Use Chapter 9, and more specifically Table 149 and Table 150, of the user manual of the microcontroller ([5]) to find these addresses (all IO ports have these registers, make sure to pick the right one!).

Answer: ____________________________________

____

- Is the pin connected to `LED0` in input or output mode after a reset (hint: see description of the data direction register)? Can you explain why?

Answer: ____________________________________

____

11.3.2 Helper Functions

In order to keep the program readable, it is a good idea to encapsulate the functionality of the different steps in the PSD in separate functions. Create three functions with the following prototypes:

- `static void init (void);`
- `static void led_off (void);`
- `static void led_on (void);`

Use the hint below to write the configuration values to the memory addresses of the different configuration registers.

> **Using a number as a memory location**
> In C you can write to a memory location using a pointer. The addresses of the configuration registers that you've found in the homework assignments are numbers, you need to tell the C-compiler that these numbers actually are the *value* of a pointer to a 32-bit memory-location. When you want to write to such a memory location, you can use the following construction:
> `*(volatile uint32_t *)0x12345678 = 1;`
> This statement means the following: the number 0x12345678 must be interpreted as an address to a 32-bit unsigned memory location (as described with the type-cast `(volatile uint32_t *)`) and you want to write to that memory location (as indicated by the left-most `*`) the value 1 (`= 1`).

Directly writing a number to a configuration register can have unwanted results. If two LEDs, say `LED0` and `LED1`, are used in one program, writing a value to the data register to turn on `LED0` can also influence `LED1`. In order to prevent these problems, you should only change the specific bit you want to change, not all the bits in the configuration register. This can be done with the help of *bitwise* operators, as discussed in Chapter 7.

init_led

The `init` function is used to set up the direction-register. Use the address of the direction-register as found in the homework exercises and set or clear the correct bit in the register.

led_off

The `led_off` function has to write a value to the data register such that the LED is switched off. Use the address of the data register as found in the homework exercises and set or clear the correct bit in the register.

led_on

The `led_on` function has to write a value to the data register such that the LED is switched on. Use the address of the data register as found in the homework exercises and set or clear the correct bit in the register.

11.3.3 Main Function

Now that you've created the different functions, you can easily write the main function of the program according to the PSD. On Brightspace, you can find two c-files, `delay.c` and `delay.h` that implement a function `void delay_ms (uint32_t)`. Add these files to the Code::Blocks project and include the file `delay.h` in the file `main.c` in order make use of the `delay_ms` function.

Build the program and flash it to the board. Remove the jumper from the board and press the `RESET` button. Does the program work? Discuss the result with a TA.

Signature TA

11.4 ARM Assignment 1.3: CMSIS

In the previous assignment you've directly addressed the configuration registers of the LPC1343 by using the addresses of these registers. This is a difficult and error-prone way of working with these registers. A better solution would be to use symbolic names (such as the ones used in the user manual ([5])). For the Cortex-series of microcontrollers, ARM has introduced a solution in the form of *CMSIS*. CMSIS is an acronym that stands for: Cortex Microcontroller Software Interface Standard. With CMSIS it is possible to manipulate the configuration registers of a Cortex-M in a standardized way, even across Coretex-M implementations of different manufacturers (see Chapter 8). In this assignment you will change the code of the blinking LED program to a version using CMSIS.

11.4.1 Helper Functions

Use the previous version of the blinking LED program as a starting point for this assignment. The three helper functions of the blinking LED program have to be changed to use the CMSIS naming now. As before, make sure you only change the single bit you need to change.

> **CMSIS: GPIO pin names**
> The CMSIS basename for the GPIO pins is `LPC_GPIO?` with the `?` being 0 – 3, representing port 0 – 3.

11.4.2 Main Function

The main function shouldn't be changed. Test the program and discuss the results with a TA.

Signature TA

11.5 ARM Assignment 1.3: Simple Input With the Buttons

The LEDs provide you with a simple means of outputting information. In this assignment we will use a button as a simple input device. Pressing button `BUT1` will be used to invert the LED. The PSD of the program is shown in Figure 11.2.

11.5.1 Homework Assignments

Answer the following questions:

- Use the schematic of the board (see Appendix D) to find out which pins are connected to the buttons `BUT1` and `BUT2`.

  Answer: ______________________________________

  ____

- When the button is pressed, what will be the value read on the pin? Explain the answer.

  Answer: ______________________________________

- When the button is released, what will be the value read on the pin? Explain the answer. (hint: what is the default configuration of the pin? (see Chapter 7 of [5]))

  Answer: ______________________________________

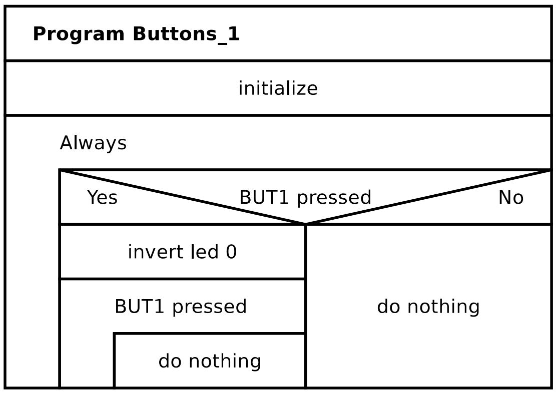


Figure 11.2: PSD of the program `buttons_1`

11.5.2 Helper Functions

In the PSD of Figure 11.2 there is a block `initialize` and there are checks on the state of the button `BUT1`. This functionality should be encapsulated in two functions:

- **static void** `init_button (`**void**`)`
- **static bool** `but1_pressed (`**void**`)`

> **The type `bool`**
> The type `bool` is defined in the header file `stdbool.h`. A `bool` can have the values `true` and `false`.

> **Naming a boolean function**
> When you create a function that returns a boolean, always make sure the return value has a clear meaning! For example, it is clear that the result `true` for the function `but1_pressed` means the button is pressed. However, what is the meaning of a return value `false` for a function called `get_status`? A much better name would be `status_ok`.

`init_button`

The function `init_button` has to initialize the port pins for use of the button `BUT1`.

`but1_pressed`

The `but1_pressed` functions return a 1 when the button `BUT1` is pressed down.

Inverting the LED

Think about how you want to implement the functionality to invert the LED. You can create new helper functions if you want to.

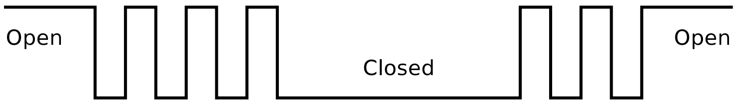


Figure 11.3: Effect of bouncing on an input pin

11.5.3 Main Function

The main function has to be implemented according to the PSD of Figure 11.2. Remember to initialize the port pins of both the LED and the button. Build and flash the program. Does the program work correctly? Discuss the results with a TA.

Signature TA

11.6 ARM Assignment 1.4: Debouncing

As you've probably noticed, the program of the previous assignment doesn't work entirely as intended: sometimes pressing the button does not seem to invert the LED, or the LED inverts when you release the button. This is caused by a phenomenon called *bouncing*. The switches used are mechanical components. When such a switch is opened or closed, it isn't opened or closed instantaniously. Instead, it will alternate between opened and closed state during some time before settling to the required position. On the corresponding pin, this may result in a signal like the one shown in Figure 11.3.

The duration of this bouncing depends on the type of switch used, the switches on the development board are typically settled after 25 ms. Since the microcontroller runs at a frequency of 72 MHz, a change in the state of the button during this 25 ms bouncing period will be seen a new button press. In order to make the program insensitive to this bouncing, the buttons have to be *debounced* by waiting for 25 ms at the correct moments. This will make sure the buttons have settled before the program proceeds any further.

Add 25 ms pauses at the correct spots in the program to debounce the buttons. Build and flash the program, does it work as expected? Show the results to a TA.

> You can use the `delay_ms` function to implement the 25 ms pause.

Signature TA

Chapter 12

ARM Assignment 2: Timers and Interrupts

Preparations

- Read Chapter 16 of the User Manual ([5])
- Read Chapter 9 of this manual
- Do the homework assignments of §12.3.1 and §12.4.1

Objectives

In this session you will

- Use a timer with polling
- Use a timer with interrupts
- Experiment with a low power mode

12.1 Introduction

The power of a microcontroller lies in the large number of builtin peripherals. One of those peripherals is the *timer*. The timer can be used to wait for a specific amount of time and then give a signal. The microcontroller can repeatedly check in a loop if a timer event has occurred. Another method for reacting on timer events is using *interrupts*. Using interrupts has several advantages above polling, but in some cases it can complicate programs.

12.2 PWM Signal and Timers

In this assignment you will write code to use the timer with both polling and interrupts. You will have to generate a simple Pulse Width Modulated (PWM) signal, that can be used for controlling a servo motor, dimming a lamp, etc.

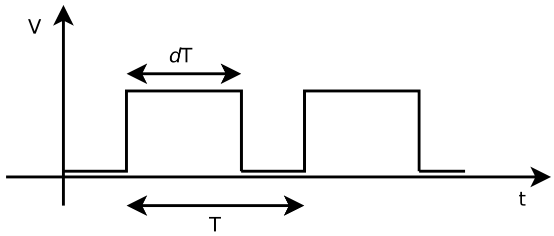


Figure 12.1: PWM signal showing the period T and the duty cycle *d*

12.2.1 PWM Signal

A PWM signal is a periodic signal in which the active part of signal is modulated. Such a signal is shown in Figure 12.1. A PWM signal is specified with two characteristics:

- The period T
- The duty cycle d, the fraction of T that the signal is in the active state

The PWM signal must be outputted on pin PIO1_6. In this assignment, the output has to be generated using a timer.

> **Hardware Support PWM signals**
> It is possible to use the timers to directly generate the PWM signals, but in this assignment we won't use this functionality.

12.2.2 Timers

The LPC1343 has four timers: two 16-bit timers and two 32-bit timers. The timer will increment the Timer Counter when the Prescale Counter overflows. The timer has a four so-called Match Registers (MR0–MR3) that contain a number. When the values of the numbers in the Timer Counter and a Match Register are equal, the timer can initiate a number of actions, like resetting or stopping the timer and generating an interrupt. In this assignment the following setup will be used:

- 32-bit timer 0 (TMR32B0) will be used as timer
- Match Register MR3 will be used for timing the period T
- Match Register MR0 will be used for timing dT

> **Timer interrupt flag without using interrupts**
> One of the possible actions on a match of MR0–3 is generating an interrupt. In fact this does not actually generate an interrupt, but instead sets a flag (in the TMR32B0IR register). When interrupts for TMR32B0 are enabled, the interrupt controller will generate an interrupt when this flag is set. When interrupts for TMR32B0 are disabled, this flag can be used for polling.

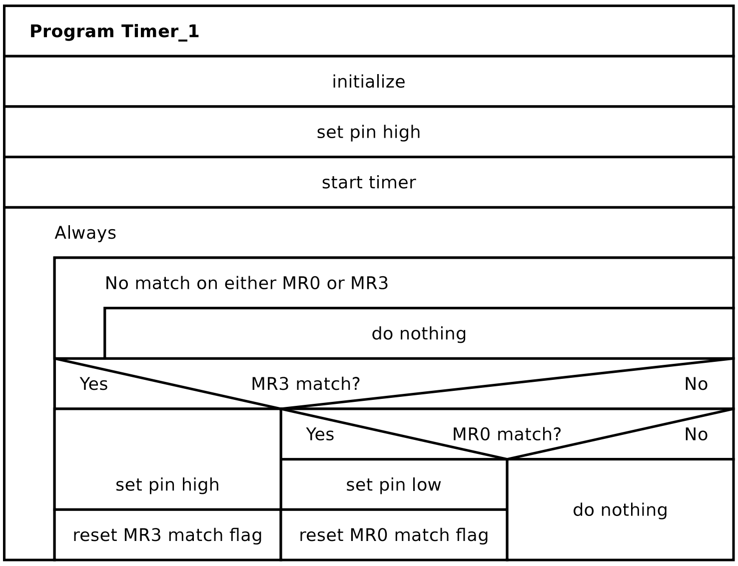


Figure 12.2: PSD of the main part of the `timer_1` program

12.3 ARM Assignment 2.1: Polling

In this assignment, you will create a program that will generate a PWM signal using polling of the timer. The PWM signal has the following specifications:

- Frequency of 100 Hz
- Duty cycle of 80%

A PSD of the main part of the program is shown in Figure 12.2.

12.3.1 Homework Assignments

Read Chapter 16 of the User Manual ([5]) and answer the following questions:

- What should be the initialization value for the following registers: `TCR`, `TC`, `PR`, `PC`, `MR0` and `MR3`?

  Answer: ____

  ____

- Given that MR3 is used for timing the period T and MR0 for timing dT, what should be the initialization value for `MCR`? Explain your answer.

  Answer: ____

  ____

- In the PSD there is a loop that waits until there is a match with one (or both) of the match registers. Using polling on the IR register, what line of C-code (using bitwise operators) can you use to check if there is a match with a Match Register?

  Answer: ____

> **CMSIS: 32-bit timer 0 naming**
> The basename for the 32-bit timer 0 is `LPC_TMR32B0`.

12.3.2 Helper Functions

The `initialization` block in the PSD of Figure 12.2 has to initialize the timer and the port pin. Write the helper functions for initializing the timer and the port pin. It might be useful to have the following funcions as well:

- `void pin_low (void)`
- `void pin_high (void)`
- `void timer_start (void)`
- `void timer_stop (void)`

> **Clocking of the timer**
> For power-saving reasons, the timers are unclocked after reset. Carefully read §16.3 and §16.7 of [5] for the initialization steps you need to perform to enable the clock to the timers.

> **CMSIS: System configuration registers**
> The basename for the system configuration registers is `LPC_SYSCON`. The member names of the registers are the full names as they appear in [5].

12.3.3 Main Function

With the helper functions and the PSD, write the main function. Use the oscilloscope to test if the program works correctly.

> **Clearing the flag**
> Carefully read the documentation on the IR register on how to clear the flag.

Show the working program to a TA.

Signature TA

12.4 ARM Assignment 2.2: Interrupts

In the assignment in §12.3 you've created a program that could generate a periodic signal with a fixed duty cycle of 80%. That program used polling, in this assignment you will change the program to use interrupts.

12.4.1 Homework Assignments

Answer the following questions (see Chapter 9):

- What name should the interrupt service routine (ISR) of timer TMR32B0 have?

  Answer: ______________________________

- What are the parameters and the return type of the ISR?

  Answer: ______________________________

- How do you enable the timer interrupt?

  Answer: ______________________________

12.4.2 Interrupt Service Routine

The interrupt service routine should now handle the checks on the MR0 and MR3 flags. Use the PSD from Figure 12.2 as a guideline and write the interrupt service routine. Note that when the program executes the ISR, at least one of the flags has been set.

12.4.3 Main function

Since you no longer need to check the flags, the main function is very simple now, it just contains an endless loop! Write the main function and make sure to add the neccessary additional initialization steps to enable interrupts. Test the program with the oscilloscope. Does it work?
Show a working program to a TA.

Signature TA

12.5 ARM Assignment 2.3: Making Interrupts Useful

If the program works, you've just recreated the program of §12.3. Although the interrupt version is not very difficult, it does not seem to add any useful functionality. Of course it is possible to do useful things inside the now empty loop in the main function, but for this simple program there is nothing to do. The only thing that loop does, is waiting for interrupts to occur.

But that opens up a maybe unexpected possibility: power saving. The LPC1343 supports four power modes: active, sleep, deep-sleep and deep power-down. These modes are described in §3.9 of[5]. The deep-sleep and deep power-down mode require special care to enter or leave, but sleep mode can be accessed very easily: execution of the ARM instruction `wfi`, which stands for Wait For Interrupt. When executing this instruction (with the default configuration after a reset), the processor will enter the sleep mode. When an interrupt occurs, the processor automatically returns to the active state. The `wfi` instruction can be accessed in C with the CMSIS function `__WFI ()` (two underscores).

In the lab USB adapters are available that can be used to measure the current drawn by the microcontroller board. Examine the effect of sleep mode by measuring the current with and without the use of the `wfi` instruction in the loop.
Show your results to a TA.

Signature TA

Chapter 13

ARM Assignment 3: I^2C

Preparations

- Do the homework assignments of §13.3.2
- Read Chapter 10 of this manual

Objectives

In this session you will

- Work with an I^2C sensor

13.1 Introduction

In the book the text on I/O devices mostly discusses high speed interconnects, such as PCI, PICe, SATA, USB, etc. However, even in modern PCs there is low speed communication with devices such as temperature sensors, intrusion detection systems, etc. Those communications are done via SMBus (System Management Bus), which is derived from I^2C. In this assignment you will write a program that communicates with a sensor that uses the I^2C protocol.

13.2 ARM Assignment 3.1 : LPC1343 I^2C Support

13.2.1 Homework Assignments: Hardware

The I^2C protocol uses two wires for communication: SDA and SCL.

- Which pin is connected to SDA?

  Answer: ______________________________

- Which pin is connected to SCL?

  Answer: ______________________________

13.2.2 Homework Assignments: Software

The User Manual contains extensive documentation on the I^{2}C support in the LPC1343. In the assignment you only need to use a few of the possibilities of the I^{2}C-bus controller.

> **CMSIS: I^{2}C names**
> The CMSIS basename for the I^{2}C interface registers is `LPC_I2C`. Since there is only one I2C interface on the LPC1343, the 0 in the register names in [5] is not used.

Read the §13.8: Register Description of the User Manual ([5]) and answer the following questions:

- What line of code would you use to set the Start flag (`STA`)?

  Answer: ______________________________

- What line of code would you use to clear the I^{2}C interrupt flag?

  Answer: ______________________________

- Which register is used for reading or writing data?

  Answer: ______________________________

- The `SI` flag can be polled to check if a I^{2}C event has occurred. What line of code would you use to wait for a I^{2}C event?

  Answer: ______________________________

Signature TA

13.3 ARM Assignment 3.2: The TMP102

13.3.1 TMP102 I^{2}C Thermometer

The I^{2}C device used in this assignment is a digital temperature sensor manufactured by Texas Instruments: TMP102. The datasheet ([6]) can be found on Brightspace. The device is mounted on a small PCB that contains the TMP102 device, some resistors and a capacitor. The schematic of the PCB can be found in [7], which is also available on Brightspace. The TMP102 can be configured for different functionality, but in the lab we will only use the default settings.

13.3.2 Homework Assignments: TMP102

Use the datasheets of the TMP102 to answer the following questions:

- The address of the TMP102 is configurable via the `ADD0` pin. On the PCB, this pin is connected to GND. Determine the real address and the derived read and write addresses (in hexadecimal).

  Answer: ______________________________

  ____

- By default, the TMP102 uses a 12-bit format to represent the temperature. Using this 12-bit format, what will be the representation of -12°C and 30°C?

  Answer: ______________________________

  ____

- In the default 12-bit format, which temperatures are represented by the numbers 0010 1010 1001 and 1101 0100 1000?

  Answer: ______________________________

  ____

- How many bits of the 12-bit result are sufficient to represent the temperature truncated to whole numbers?

  Answer: ______________________________

Signature TA

13.4 ARM Assignment 3.3: I^{2}C With Polling

To test the I^{2}C functionality and the TMP102, the first assignment is based on polling.

13.4.1 Initialization of the I^{2}C interface

Write an initialization routine that initializes the IO pins and sets up the I^{2}C module in master mode. The clock to the I^{2}C module is disabled on reset, you have to enable the clock (in the SYSAHBCLKCTRL register) and de-assert the reset to the block (in the PRESETCTRL register) before you can configure the I^{2}C module itself. Since the I^{2}C module is only used in master mode, the configuration only consists of setting the high and low duty cycle registers. Configure those registers for a 100 kHz clock with a 50% dutycycle.

The last step in the configuration is enabling the I^{2}C interface (register I2C0CONSET). You can also find these steps in §13.2: *Basic configuration* of the User Manual.

CMSIS: System Clock Frequency
CMSIS defines the global variable `uint32_t SystemCoreClock` that contains the system clock frequency in Hz. This can be useful to set up the I^2C clock.

13.4.2 Using Master Receive Mode

In order to read the sensor values you have to follow the steps described in §10.3.2. These steps are depicted in the state diagram of Figure 13.1 in the parts Send and Receive. Annotate the diagram by filling in which flags (SI, STA, STO, AA, etc) you need to set and clear in each step, *do not fill in the status line yet*.

The start flag STA
Note that the start flag STA isn't automatically cleared! If you send a start condition by setting the STA flag, you have to clear it when the LPC1343 has finished sending the start condition. If you don't clear the STA flag, the LPC1343 will send a start condition before every I^2C transmission.

Write C code to read the sensor values with polling, using the annotated diagram and the answers to the homework exercises. You can repeatedly read the value of the sensor by repeating the steps of the Receive part of the communication protocol, as is shown in the diagram.

On Brightspace you can find the files `leddriver.{c,h}` that you can use to write a byte to the eight LEDs. You have to call the function `init_leds ()` before using any of the other functions. Use these functions to output the value read from the sensor. What temperature do you measure? Show the working program to a TA.

Signature TA

13.5 Assignment 3.4: I^2C with Interrupts

The I^2C-bus module can also work with interrupts. Each I^2C transmission will set the interrupt flag an store a status code in the register I2C0STAT. In the interrupt routine, you can a take different action based on the value in this status register.

13.5.1 Status Register Values

Before you can implement the interrupt routine, you have to know the values in the status registers for each step in the diagram. Modify your program in the following way:

- Make sure you only read the value of the sensor once, thus remove the jump back to the Send Repeated Start step in the diagram;
- Create an array large enough to store all the status register values;
- After each step in the I^2C protocol, store the status register values in the array;

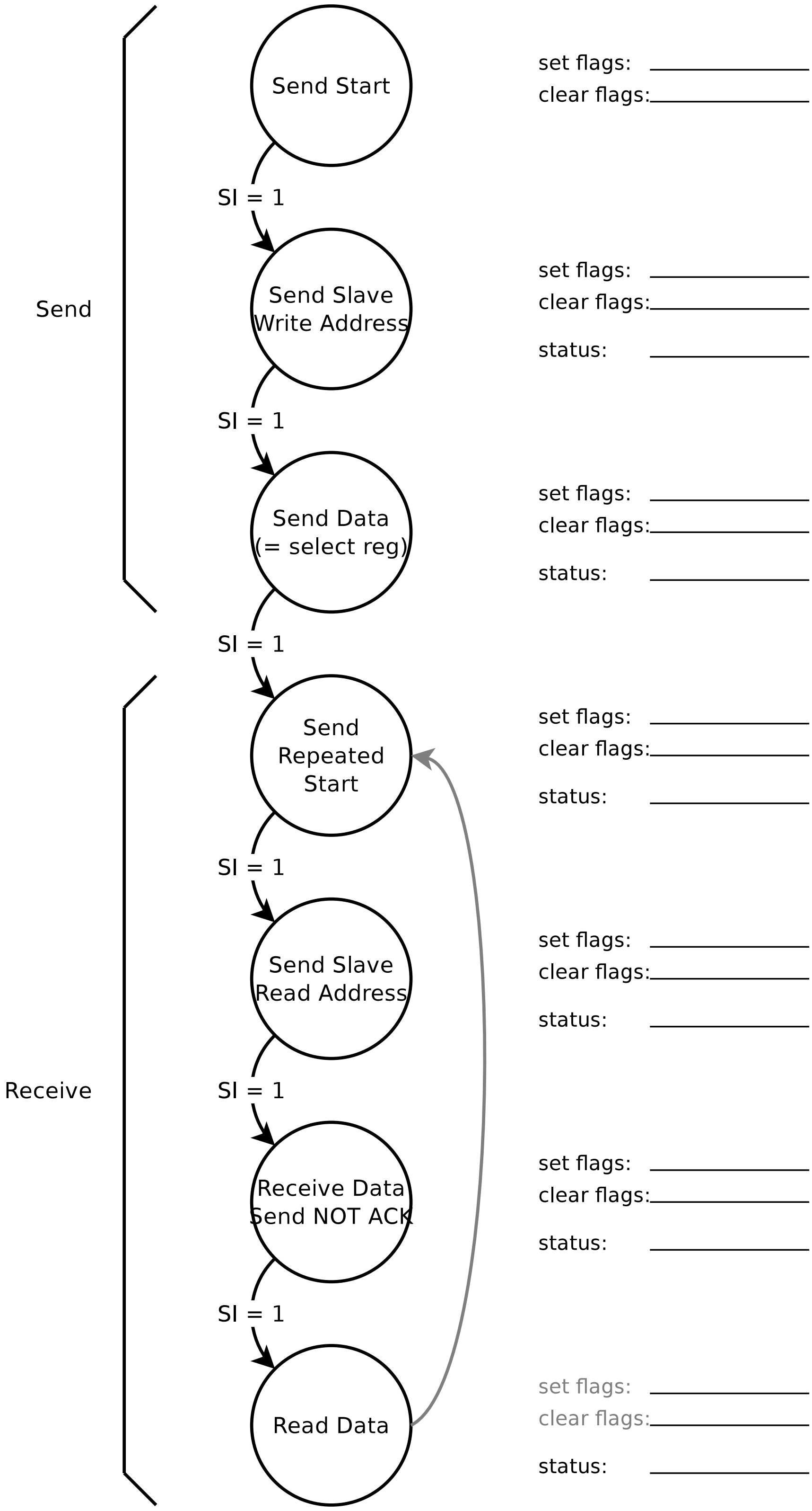


Figure 13.1: State diagram of the steps required to read the sensor value

- Write a loop to show the values stored in the array on the LEDs and pause long enough (use the `delay_ms ()` function) in between to write down each value. Run this code after you've completed the I^2C protocol to read the value of the sensor.

Fill in the status register values at the status lines in the diagram of Figure 13.1. Compare the codes with the values in the user manual ([5] Table 233 (page 228) and Table 234 (page 231)).

> **Timeout function of the TMP102**
> The TMP102 has a timeout function that powers down the device after 30 ms of inactivity. This means you cannot pause your program halfway the I^2C protocol to show the current value of the status register! You have to store the values in an array and show them on the LEDs afterwards.

13.5.2 Implement the Interrupt Service Routine

Now that the status codes are known, you have enough information to implement the interrupt routine. Take the following steps:

- Enable the interrupt in the initialization function;
- Look up the correct name of the interrupt function;
- In the interrupt function, read the value of the status register and take the corresponding action (see your annotated diagram);
- Modify your main-function.

> **C switch-statement**
> You can use the C switch-statement to easily implement the interrupt routine. Make sure to add `break` statements after each `case` block! See [4] for more information.

Show your working program to a TA.

Signature TA

Part IV

VHDL MIPS Assignments

Chapter 14

Documentation VHDL MIPS Assignments

14.1 Introduction

In the last assignment of the Computer Architecture and Organization lab you have to extend the VHDL description of a single cycle MIPS processor with two new instructions. You have to test the new instructions using an assembly program.

14.2 Description of the VHDL Code

The MIPS processor implemented in the VHDL code has support for the following instructions:

add/addu[1]	srl	blez
addi/addiu[1]	sub/subu[1]	bltz
and	xor	bne
andi	xori	j
nor	lui	jr
or	beq	lw
ori	bgez	sw
sll	bgtz	

The test program `test.s` tests all these instructions. The comments after each line of code are divided into two columns: the left column shows the correct result, the right column shows results that can only happen if the processor isn't working correctly.

The VHDL code is spread over several files, each of which is described briefly. Make sure you understand the stucture of the code before you proceed with the assignments!

14.2.1 `memory.vhdl`

The file `memory.vhdl` contains the module that implements the memory of the processor. In an actual processor, this memory will be external, but in this assignment the memory is internal. The memory has a size of 256 bytes, which is sufficient for the test

[1] Overflow is not implemented, thus the signed and unsigned versions of the instructions are identical

program `test.s` (which counts 228 bytes). The program is loaded on a processor reset.

Signals:

- *clk* Clock signal;
- *rst* Reset signal;
- *memread* When this signal is high, data is read from the memory. Note that instructions are always read from the memory, regardless of the value of this signal;
- *memwrite* When this signal is high, data is written to the memory;
- *address1* Address of the instruction that has to be read;
- *address2* Address of the data that has to be read or written;
- *writedata* Data to be written;
- *instruction* Instruction read from the memory;
- *readdata* Data read from the memory.

The entity `memory` is generated from a MIPS assembly file. You can use the program `cao_mips_tools` (use Tools→MIPS to VHDL or press F8) to convert the MIPS assembly file into VHDL code. You can download `cao_mips_tools` from Brightspace. More information on `cao_mips_tools` can be found in §14.3.

> The assembler is case sensitive: all instructions must be entered in lower case!

14.2.2 `registers.vhdl`

The file `register.vhdl` contains the module that contains the registers of the processor. All registers are set to 0 after reset.

Signals:

- *clk* Clock signal;
- *rst* Reset signal;
- *regwrite* When this signal is high, data is written to a register;
- *readreg1* Number of the first source register;
- *readreg2* Number of the second source register;
- *writereg* Number of the destination register;
- *writedata* Data to be written;
- *readdata1* Data read from the first source register;
- *readdata2* Data read from the second source register;

14.2.3 `pc.vhdl`

The file `pc.vhdl` contains the module that implements the program counter. In case of a jump or branch, the module will load the new address, otherwise the program counter will be advanced by four. The program counter is set to 0 after reset.

Signals:

- *clk* Clock signal;
- *rst* Reset signal;
- *jump* When this signal is high, the module will set the address to `pc_in`, otherwise the address will be advanced by four.
- *pc_in* New program counter address in case of a jump or branch;
- *pc_add* Current address + 4;
- *pc* Current address.

14.2.4 `jump.vhdl`

The file`jump.vhdl` contains the module that calculates the new address in case of a jump or branch. The module needs several pieces of data:

- The result of the ALU;
- The type of jump instruction;
- The address required to calculate the new address.

Signals:

- *branchalu* The result of a calculation;
- *branchcontrol* The type of instruction:
  - 00: no jump
  - 01: j
  - 10: jr
  - 11: branch
- *extend* Result of the sign extender;
- *jump* Immediate value of a `j` instruction;
- *registers* The value in the register used in case of a `jr` instruction;
- *current* Value of the current program counter + 4 (this is the signal *pc_add* of file `pc.vhd`);
- *branch* This signal indicates whether or not a jump occurred (the signal *jump* of file `pc.vhd`);
- *address* The calculated address (the signal *pc_in* of file `pc.vhd`).

14.2.5 `control.vhdl`

The file `control.vhdl` contains the module that implements the control block. It is functionally almost identical to the control block described in the book (see [2]).

Signals:

- *instruction* Six most significant bits of the instruction;
- *funct* Six least significant bits of the instruction (required for decoding the jr instruction);
- *branch* Type of jump instruction (the signal *branchcontrol* of the file `jump.vhdl`);
- *regdst* This signal controls the multiplexer that assigns either bits 20-16 (on a low value) or bits 15-11 (on a high value) to the signal *writereg* (file `registers.vhdl`);
- *memread* Signal *memread* of the file `memory.vhdl`;
- *memtoreg* This signal controls the multiplexer that assigns either signal *read-data2* (file `memory.vhdl`) or the signal *result* (file `alu.vhdl`) to the signal *write-data*;
- *aluop* Signal that indicates to alucontrol the type instruction:
  - 000: Other or unknow
  - 001: add
  - 010: and
  - 011: or
  - 100: xor
  - 101: lui
  - 110: bgez, bltz
  - 111: beq, bne, blez, bgtz
- *memwrite* Signal *memwrite* of file `memory.vhdl`;
- *alusrc* This signal controls the multiplexer that assigns either signal *readdata2* (file `registers.vhdl`) or signal *value* (file `extend.vhdl`) to signal *data2* (file `alu.vhdl`);
- *regwrite* Signal *regwrite* of file `registers.vhd`.

14.2.6 `alucontrol.vhdl`

The file `alucontrol.vhdl` contains the module that implements the alu control block. It is functionally almost identical to the alu control block described in the book (see [2]).

Signals:

- *aluop* Signal *aluop* of file `control.vhdl`;
- *instruction* Six least significant bits of the instruction;

- *branch1* Bit sixteen of the instruction, required to distinguish between `bgez` and `bltz`;
- *branch2* Bits twentysix and twentyseven of the instruction, required to distinguish between `beq`, `bne`, `blez`, and `bgtz`;
- *aluinstr* Signal that indicates to the alu the type of instruction:
  - 00000: Other or unknown
  - 00001: lui
  - 00010: add
  - 00011: sub
  - 00100: and
  - 00101: or
  - 00110: nor
  - 00111: xor
  - 01000: sll
  - 01001: srl
  - 01010: bltz
  - 01011: bgez
  - 01100: beq
  - 01101: bne
  - 01110: blez
  - 01111: bgtz

14.2.7 `alu.vhdl`

The file `alu.vhdl` contains the module that implements the arithmetic and logic unit (alu). This specific alu supports a number of different branch instructions and additionally contains a barrel shifter.

Signals:

- *data1* First operand;
- *data2* Second operand;
- *shamt* Shift amount;
- *aluinstr* Signal *aluinstr* of file `alucontrol.vhdl`;
- *result* Result of the calculation;
- *branch* Signal *jump* of file `pc.vhdl`. This signal is high when signal *aluinst* has the value 0, or when a branch has to be taken.

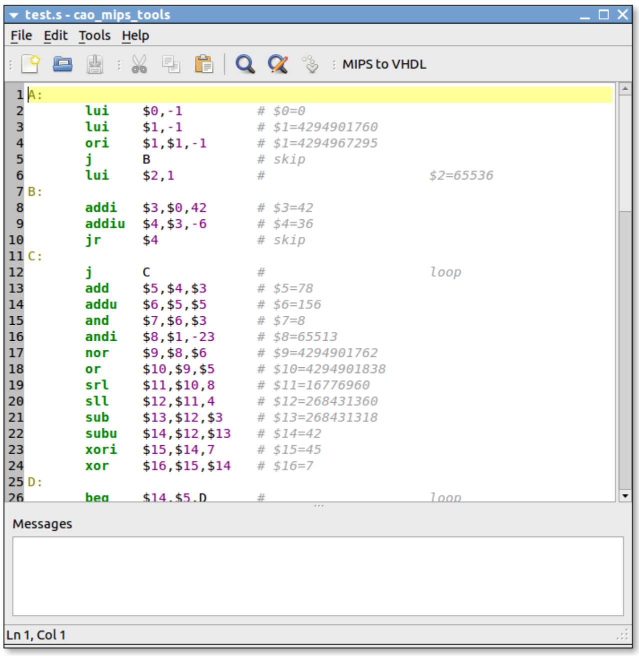


Figure 14.1: Screenshot of `cao_mips_tools` with open file `test.s`

14.2.8 `extend.vhdl`

The file `extend.vhdl` contains the module that implements the sign extender.

Signals:

- *aluop* The signal *aluop* of file `control.vhd`;
- *instruction* The least significant bits of the instruction (this is the immediate value);
- *value* The result.

14.2.9 `mips.vhdl`

The file `mips.vhdl` is the top level entity in which all parts are combined to implment the processor. The structure is mostly identical to processor described in the book. See Figure 4.17 at page 265 in the book ([2]) or the figure in Appendix E of this manual for an overview of the single cycle MIPS processor that is almost identical to the VHDL processor.

14.3 Software: `cao_mips_tools`

The entity `memory` is generated from a MIPS assembly file with the program `cao_mips_tools`. You can download a version for your operating system as a zip-file from Brightspace. A screenshot of the program can be found in Figure 14.1.

14.3.1 Editor

`cao_mips_tools` contains a simple syntax highlighting editor for the supported subset of the MIPS assembly language.

14.3.2 Assembling

You can assemble the current MIPS assembly program with the button MIPS to VHDL or Tools→MIPS to VHDL (F8). The frame Messages will show you the result, any errors will be printed in red.

The assembler is case sensitive: all instructions must be entered in lower case!

Simulate
The Linux version of `cao_mips_tools` also features a Simulate option. This will run the GHDL simulator on the VHDL files in the current directory and show the resulting wave form in the GtkWave viewer. Note that this requires all VHDL file to be located in the same directory as the assembly `.s` file. The path name should *not* contain any spaces.

Chapter 15

VHDL MIPS Assignments

Preparations

- Read §14.2 of this manual
- Read the variant of the assignment that is assigned to you
- Map the VHDL code onto Figure 4.17 of the book [2] (or see Appendix E of this manual)

Objectives

- Understand how the instruction set and the internal construction of the processor are related
- Understand the modifications required to the data path and the control path in order to add support for a new instruction

15.1 Introduction

You have to do VHDL MIPS Assignment 1 and then implement *one* of the variants of VHDL MIPS Assignment 2. In Brightspace Grades you see which variant you have to implement.

15.2 VHDL MIPS Assignment 1: Testing the Processor

Before you start with the actual assignment, you have to test the VHDL code in the ModelSim simulator. Take the following steps to test the processor:

- Create a work directory and extract the file `vhdl_mips.zip` (see Brightspace) into that directory;
- Start ModelSim (Start→Programs→Engineering) and select the work directory created earlier (File→Change Directory);
- Create a new library (File→New→Library…, accept default settings and press OK);

- Compile all VHDL files in the work directory: select Compile→Compile, select the newly created library, than select all VHDL files in the work directory and press the Compile button;
- Start the simulation of the testbench `mips_tb` with Simulate→Start Simulation.
- The tab sim in the leftmost frame now shows a hierarchical overview of the processor. By selecting a design unit, the window Objects will show the available signals of that design unit. Select the signals you want to see in your simulation and add those to the wave output by right-clicking and selecting the option Add Wave;
- Start the actual simulation by clicking the Run button (F9).

You might want to change the radix of the signals *pc* and *registers* to `unsigned` or `hex` in order to improve readability. This can be changed in the window Wave by right-clicking on the signals and selecting Radix.

> Under Linux you can open the `test.s` file in `cao_mips_tools` and use the Simulate menu option.

Show the results to a TA and explain that you can see that the processor is working properly.

Signature TA

15.3 VHDL MIPS Assignment 2, Variant 1: `sra` and `jal`

In this variant of the assignment, you have to implement the following instructions:

- `sra`
- `jal`

As a first assignment, you have to write a test program to test the instructions. Using this program you can later on prove that your modifications to the processor are correct.

15.3.1 Write a Test Program

Write a test program that you can use to test the `sra` and `jal` instructions. Make sure to test different use cases. You can use a similar approach as used in the original test program.

Use `cao_mips_tools` to convert the assembly program to VHDL. Show the finished program to a TA and explain why your program is sufficient to test the instructions.

Signature TA

15.3.2 Implement the `sra` instruction

Implement the `sra` instruction. Use your test program to show that you've implemented the instruction correctly.

Signature TA

15.3.3 Implement the `jal` instruction

Implement the `jal` instruction and use your test program to show that both the `sra` instruction and the `jal` instruction are implemented correcly.

Implementing `jal` instruction requires multiple structural changes to the processsor. Use the figure in Appendix E of this manual to devise the necessary adjustments.

Signature TA

15.3.4 The Complete Processor

Combine the original test program with your test program and show a TA that the processor still supports the initial set of instructions and also supports the new instructions `sra` and `jal`.

Signature TA

15.4 VHDL MIPS Assignment 2, Variant 2: `sllv` and `jalr`

In this variant of the assignment, you have to implement the following instructions:

- `sllv`
- `jalr`

As a first assignment, you have to write a test program to test the instructions. Using this program you can later on prove that your modifications to the processor are correct.

> **Error in the book: instruction `jalr`**
> The description of the `jalr` instruction in [1] is not entirely correct. The instruction is printed as:
> `jalr rs, rd`
> This should have been:
> `jalr rd, rs`
> `rd` stores the return address and the jump address is loaded from `rs`. The assembler uses the correct version of the instruction!

15.4.1 Write a Test Program

Write a test program that you can use to test the `sllv` and `jalr` instructions. Make sure to test different use cases. You can use a similar approach as used in the original test program.

Use `cao_mips_tools` to convert the assembly program to VHDL. Show the finished program to a TA and explain why your program is sufficient to test the instructions.

Signature TA

15.4.2 Implement the `sllv` instruction

Implement the `sllv` instruction. Use your test program to show that you've implemented the instruction correctly.

Signature TA

15.4.3 Implement the `jalr` instruction

Implement the `jalr` instruction and use your test program to show that both the `sllv` instruction and the `jalr` instruction are implemented correcly.

Implementing `jalr` instruction requires multiple structural changes to the processsor. Use the figure in Appendix E of this manual to devise the necessary adjustments.

Signature TA

15.4.4 The Complete Processor

Combine the original test program with your test program and show a TA that the processor still supports the initial set of instructions and also supports the new instructions `sllv` and `jalr`.

Signature TA

15.5 VHDL MIPS Assignment 2, Variant 3: `slt` and `bgezal`

In this variant of the assignment, you have to implement the following instructions:

- `slt`
- `bgezal`

As a first assignment, you have to write a test program to test the instructions. Using this program you can later on prove that your modifications to the processor are correct.

15.5.1 Write a Test Program

Write a test program that you can use to test the `slt` and `bgezal` instructions. Make sure to test different use cases. You can use a similar approach as used in the original test program.

Use `cao_mips_tools` to convert the assembly program to VHDL. Show the finished program to a TA and explain why your program is sufficient to test the instructions.

Signature TA

15.5.2 Implement the `slt` instruction

Implement the `slt` instruction. Use your test program to show that you've implemented the instruction correctly.

Signature TA

15.5.3 Implement the `bgezal` instruction

Implement the `bgezal` instruction and use your test program to show that both the `slt` instruction and the `bgezal` instruction are implemented correcly.

Implementing `bgezal` instruction requires multiple structural changes to the processsor. Use the figure in Appendix E of this manual to devise the necessary adjustments.

Signature TA

15.5.4 The Complete Processor

Combine the original test program with your test program and show a TA that the processor still supports the initial set of instructions and also supports the new instructions `slt` and `bgezal`.

Signature TA

Bibliography

[1] J. R. Larus, *Appendix A: Assemblers, Linkers, and the SPIM Simulator*, June 2004 (`http://pages.cs.wisc.edu/~larus/HP_AppA.pdf`)

[2] D. A. Patterson, J. L. Hennessy, *Computer Organization and Design: the hardware/software interface*, 5th edition, 2014

[3] J. Soulié, 2007, *C++ Language Tutorial* [online] Available at: `http://www.cplusplus.com/doc/tutorial/` [Accessed December 2014]

[4] Kernighan, B.W. and Ritchie, D.M., 1988. *The C Programming Language*. 2nd ed. Englewood Cliffs, NJ: Prentice Hall.

[5] NXP Semiconductors, 2011. *LPC1311/1313/1342/1343 User Manual*, Rev. 4.

[6] Texas Instruments Inc., 2007. *Low Power Digital Temperature Sensor With SMBus™/Two-Wire Serial Interface in SOT563*. [online] Available at: `http://www.ti.com/product/tmp102` [Accessed October 2012]

[7] SparkFun Electronics, 2009. *TMP102 Breakout*. [online] Available at: `http://www.sparkfun.com/datasheets/Sensors/Temperature/TMP102_Breakout-v11.pdf`

Part V

Appendices

Appendix A

Code::Blocks: Creating, building and executing a C++ project

A.1 Introduction

This appendix describes the steps required to use Code::Blocks to build and execute a C++ project. The Code::Blocks IDE works based on *projects* (as do most IDEs). A project contains information on source files, compiler settings, etc. Usually the IDE outputs a single executable based on this information.

> When you run Windows on the lab PCs you can find `Code::Blocks` in the start-menu.

> When you run Linux on the lab PCs you can start `Code::Blocks` from a terminal or the "run" dialog (Alt F2) with the command `codeblocks-17.12`.

A.2 Creating a New Project

To create a new project, first start Code::Blocks and take the following stepts:

- Select file → New → Project... or click Create a new project on the startup screen;

Now the New from template window opens, see Figure A.1. Select the Console appplication template and press Go. This will open the Console application wizard. Take the following steps:

- Press Next > to close the introduction page;

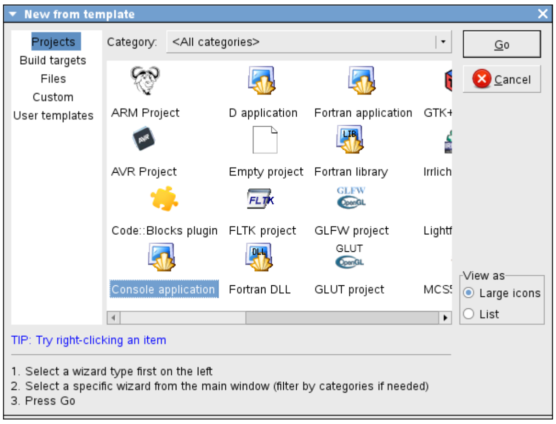


Figure A.1: Screenshot of the Code::Blocks window New from template

- In the language selection page, choose C++ and press Next >;
- On the file paths page enter a project title and a valid project directory[1];
- On the compiler settings page select GNU GCC Compiler;
- Press Finish to create the project with the specified settings.

If you successfully created the project, the Code::Blocks shows the project in the left frame (Management). Under Sources you will find a basic `main.cpp` file. A screenshot of Code::Blocks with a newly created project and the file `main.cpp` opened in the editor is shown in Figure A.2.

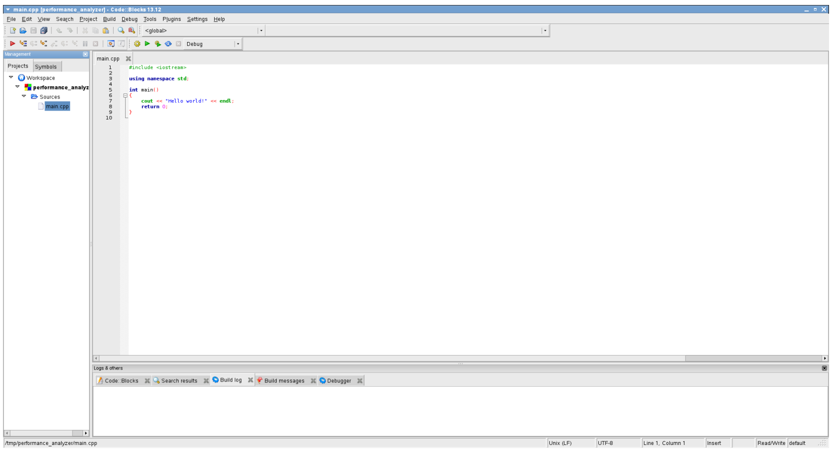


Figure A.2: Screenshot of Code::Blocks with a newly created project

[1]Make sure to select a directory in which you have write permissions

A.3 Compiler Settings

You need to build your project with certain settings in order to have it signed off. You can change these settings with the following steps:

- Select Settings → Compiler. . . ;
- On the tab Compiler flags (this is a sub-tab of Compiler settings) select:
  - In C mode, support all ISO 90 programs. In C++ mode, remove GNU extensions that conflict with ISO C++ [ansi];
  - Enable all common compiler warnings (overrides many other settings) [-Wall];
  - Enable warnings demanded by strict ISO C and ISO C++ [-pedantic].

A.4 Building a Project

Building the project is easily done from the Build-menu. The option Build (also activated with Ctrl F9) will compile all the C-files and link them together to an executable. By default, the Debug build target is build. This can be changed via Build→Select target or the Build target option on the toolbar (see Figure A.3).

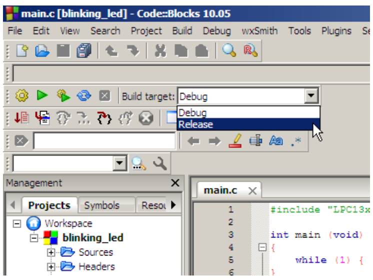


Figure A.3: Screenshot of the Build target option on the toolbar of Code::Blocks

Using Build can hide compilation warnings!
The Build option will only recompile files that have changed. This will save you time since less files will have to be compiled but it will hide compilation warnings in files that were already compiled. With Build→Rebuild (Ctrl F11) you can force recompilation of all files in the project. You can use Build→Compile current file (Ctrl Shift ⇑ F9) to force recompilation of the file you're currently editing.

A.5 Running the Program

You can execute the resulting executable with build → Run. Since you've build a console program, the program will run in a terminal.

A.6 Automatic Source Code Formatting

You can automatically have your code formatted by Code::Blocks:

- Select Plugins → Source code formatter (AStyle)

Your code has to be readable!
Always make sure your source code is readable before asking an assistant for help!

Appendix B

Code::Blocks: Creating, building and uploading an NXP LPC1343 project

B.1 Introduction

This appendix describes the steps required to use Code::Blocks with the NXP LPC1343 project wizard to create a new LPC1343 project.

B.2 Creating a New Project

Start Code::Blocks, if not done already. To create a new NXP LPC1343 project, select File→New→Project. . . . This will open the New from template dialog (see Figure B.1). Select the NXP LPC-1343 project template and press Go.

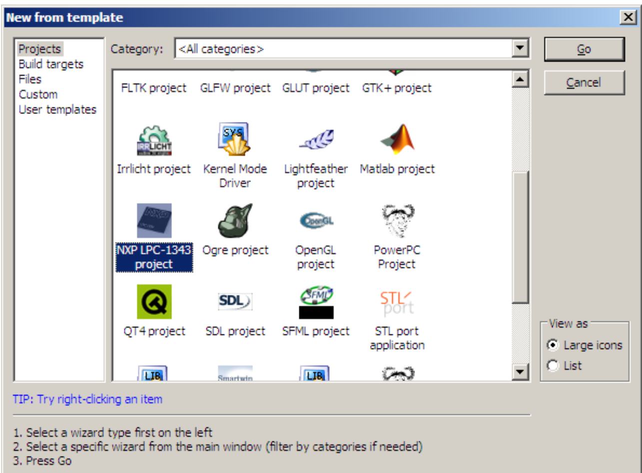


Figure B.1: Screenshot of the New from template dialog

Now the wizard starts with an intro page (see Figure B.2). Press Next to continue.

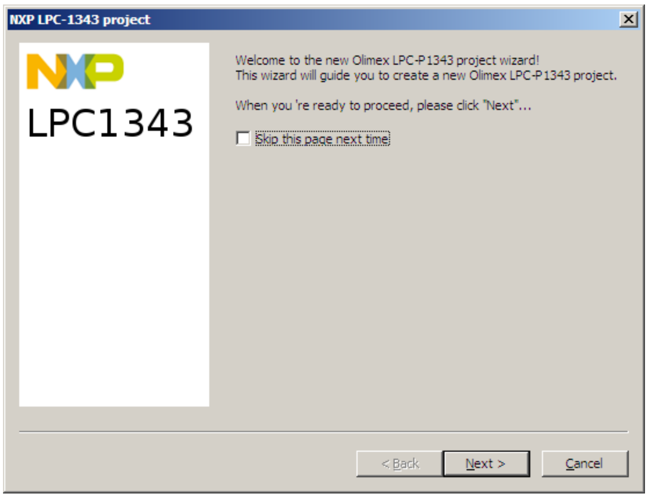


Figure B.2: Screenshot of the intro page of the wizard

The next page allows you to specify the name and location of the new project. Figure B.3 shows this page filled in for the first assignment of the lab. The next page

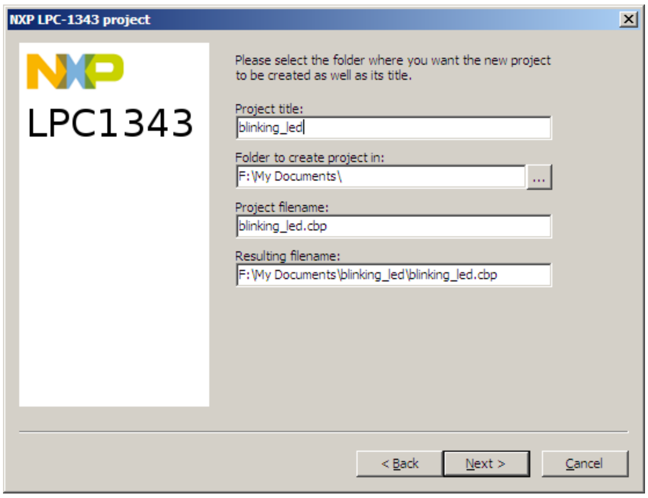


Figure B.3: Screenshot of the filename page of the wizard

(see Figure B.4) shows the compiler. You can not configure anything here.

The last page (see Figure B.5) is used to enable some options for the project. For use in the lab, the default settings are correct, unless specified otherwise. After pressing Finish, the new project will be created. The project contains a number of C and header files. The main function resides in the file `main.c`, which is the only file you should normally edit (you can of course add other C-files as well). A screenshot of Code::Blocks with a new NXP LPC1343 project is shown in Figure B.6.

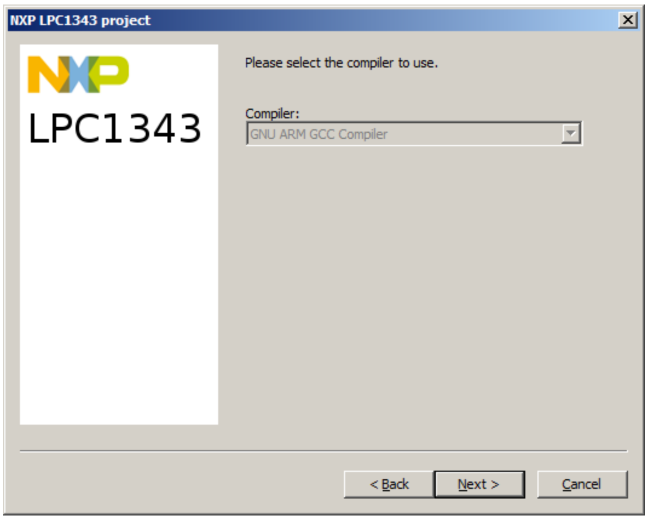


Figure B.4: Screenshot of the configurations page of the wizard

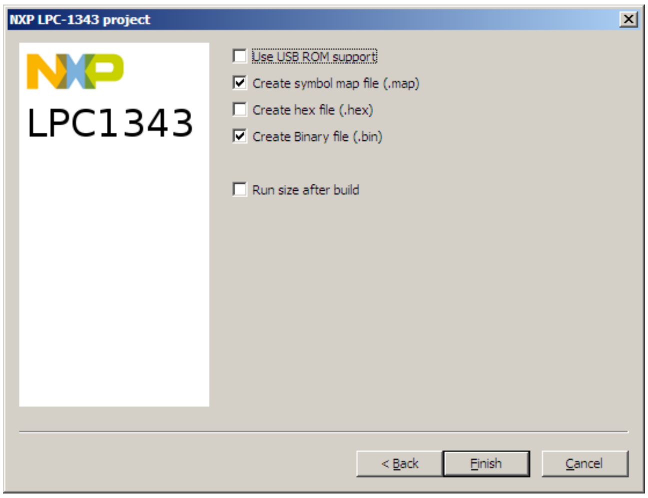


Figure B.5: Screenshot of the options page of the wizard

B.3 Building a Project

Building the project is easily done from the Build-menu. The option Build (also activated with Ctrl F9) will compile all the C-files, link them together to an executable and perform any other steps required to create a file that can be uploaded to the board.

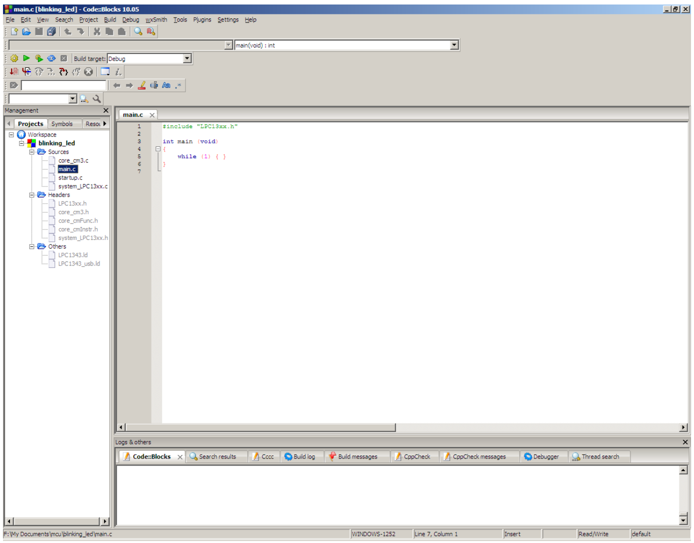


Figure B.6: Screenshot of Code::Blocks with a new NXP LPC1343 project open

Using Build can hide compilation warnings!
The Build option will only recompile files that have changed. This will save you time since less files will have to be compiled but it will hide comilation warnings in files that were already compiled. With Build→Rebuild (Ctrl F11) you can force recompilation of all files in the project. You can use Build→Compile current file (Ctrl Shift ⇧ F9) to force recompilation of the file you're currently editing.

B.4 Uploading a Project to the Board

Uploading the project to the board is simple, but differs a bit depending on whether you work with the Windows or Linux operating system. The first steps are identical:

- Build the required configuration
- Place a jumpercap on jumper `BLD_E` (see Figure B.7)
- Connect the board with an USB cable to the computer

The yellow/orange USB led `USBC` should now be lit.
For the next steps, follow the description for the operating system you use:

Steps for Windows

- The board should show up as USB flash drive, if it doesn't, press the RESET button;
- Select Project→Upload to LPC1343 to flash the board.

Virtual Drive software prevents LPC1343-flash to work correctly
LPC1343-flash cannot correctly determine the drive letter assigned to the LPC1343 board when virtual drive software such as Deamon Tools has a virtual drive connected!

Steps for Linux

- Select Project→Upload to LPC1343 to flash the board;
- If this fails with the message:

```
Opening '/dev/disk/by-id/usb-NXP_LPC134X_IFLASH_ISP000000000-0:0':
Permission denied
```

  press the `RESET`-button and try again.

The program is now flashed onto the microcontroller.

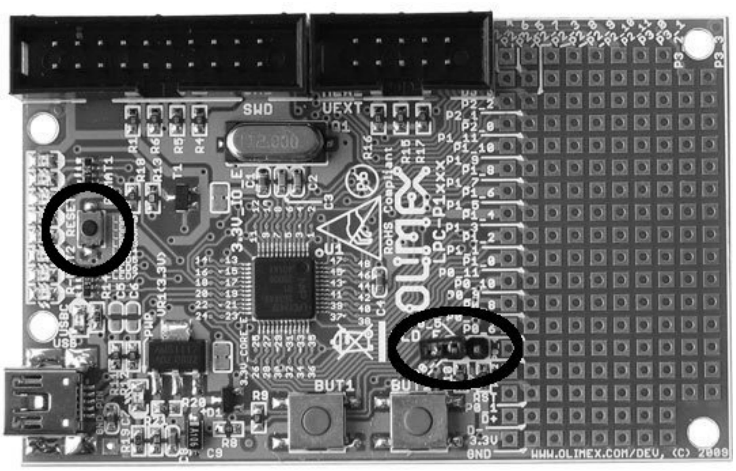


Figure B.7: Photo of the microcontrollerboard. The `RESET` button and jumper `BLD_E` are highlighted.

B.5 Running the Program

To run the program, just remove the jumpercap from `BLD_E` and press `RESET`.

Appendix C

Interrupt Service Routine Naming

The following table shows the interrupt numbers, their CMSIS name and the ISR names as defined by the startup code from the Code::Blocks NXP LPC1343 wizard.

Interrupt number	CMSIS name	ISR name	Description
0 – 39	`WAKEUP?_IRQn`	`WAKEUP_Handler`	Start logic wake-up interrupts
40	`I2C_IRQn`	`I2C_Handler`	I^2C interrupt (I2C0)
41	`TIMER_16_0_IRQn`	`TIMER_16_0_Handler`	Timer interrupt for 16-bit timer 0 (TMR16B0)
42	`TIMER_16_1_IRQn`	`TIMER_16_1_Handler`	Timer interrupt for 16-bit timer 1 (TMR16B1)
43	`TIMER_32_0_IRQn`	`TIMER_32_0_Handler`	Timer interrupt for 32-bit timer 0 (TMR32B0)
44	`TIMER_32_1_IRQn`	`TIMER_32_1_Handler`	Timer interrupt for 32-bit timer 1 (TMR32B1)
45	`SSP0`	`SSP0_Handler`	Synchronous Serial Port 0 interrupt (SSP0)
46	`UART_IRQn`	`UART_Handler`	Universal Asynchronous Receiver and Transmitter interrupt (UART)
47	`USB_IRQn`	`USB_IRQ_Handler`	Universal Serial Bus low priority interrupt (USB)
48	`USB_FIQn`	`USB_FIQ_Handler`	Universal Serial Bus high priority interrupt (USB)
49	`ADC_IRQn`	`ADC_Handler`	Analog-to-Digital Converter interrupt (ADC)
50	`WDT_IRQn`	`WDT_Handler`	Watchdog Timer interrupt (WDT)
51	`BOD_IRQn`	`BOD_Handler`	Brown Out Detect interrupt
52	`RESERVED_IRQn`	`RESERVED_Handler`	Reserved
53	`EINT3_IRQn`	`EINT3_Handler`	PIO_3 interrupt (GPIO3)
54	`EINT2_IRQn`	`EINT2_Handler`	PIO_2 interrupt (GPIO2)
55	`EINT1_IRQn`	`EINT1_Handler`	PIO_1 interrupt (GPIO1)
56	`EINT0_IRQn`	`EINT0_Handler`	PIO_0 interrupt (GPIO0)
57	`SSP1_IRQn`	`SSP1_Handler`	Synchronous Serial Port 1 interrupt (SSP1)
58	`RESERVED_IRQn`	`RESERVED_Handler`	Reserved

Appendix D

Schematic and photo of the microcontroller board

In the schematic of the board, you can see to which pins the leds are connected, if pressing the buttons will result in a 0 or a 1 on a pin etc.

In Figure D.1 you can find the layout of the SWD connector and the UEXT connector.

The enlarged photograph will make it easier to find the locations of the pins. Note that the silkscreen on some of the PCBs is somewhat different from the one on the photograph. The photograph shows the correct pin positions!

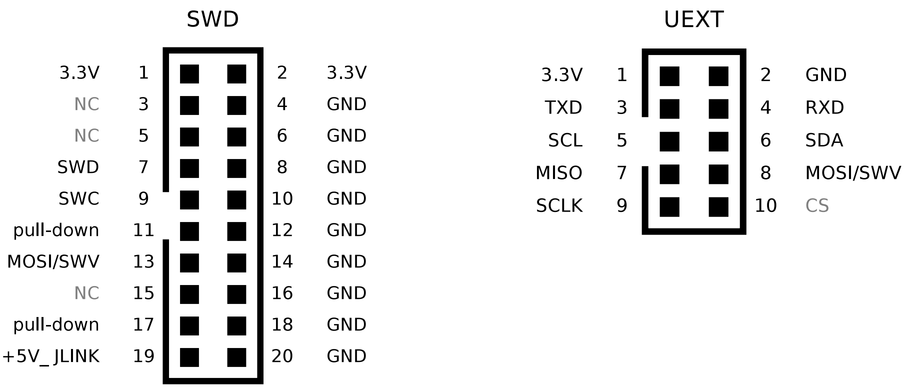


Figure D.1: Layout of the SWD and the UEXT connectors

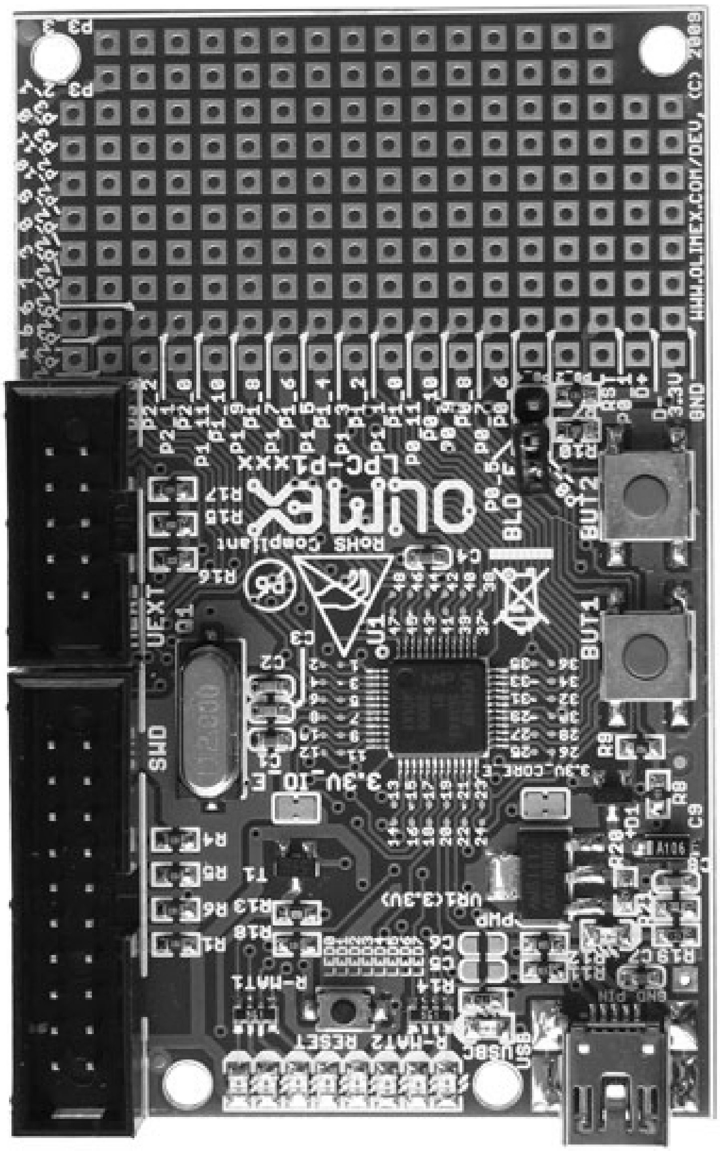
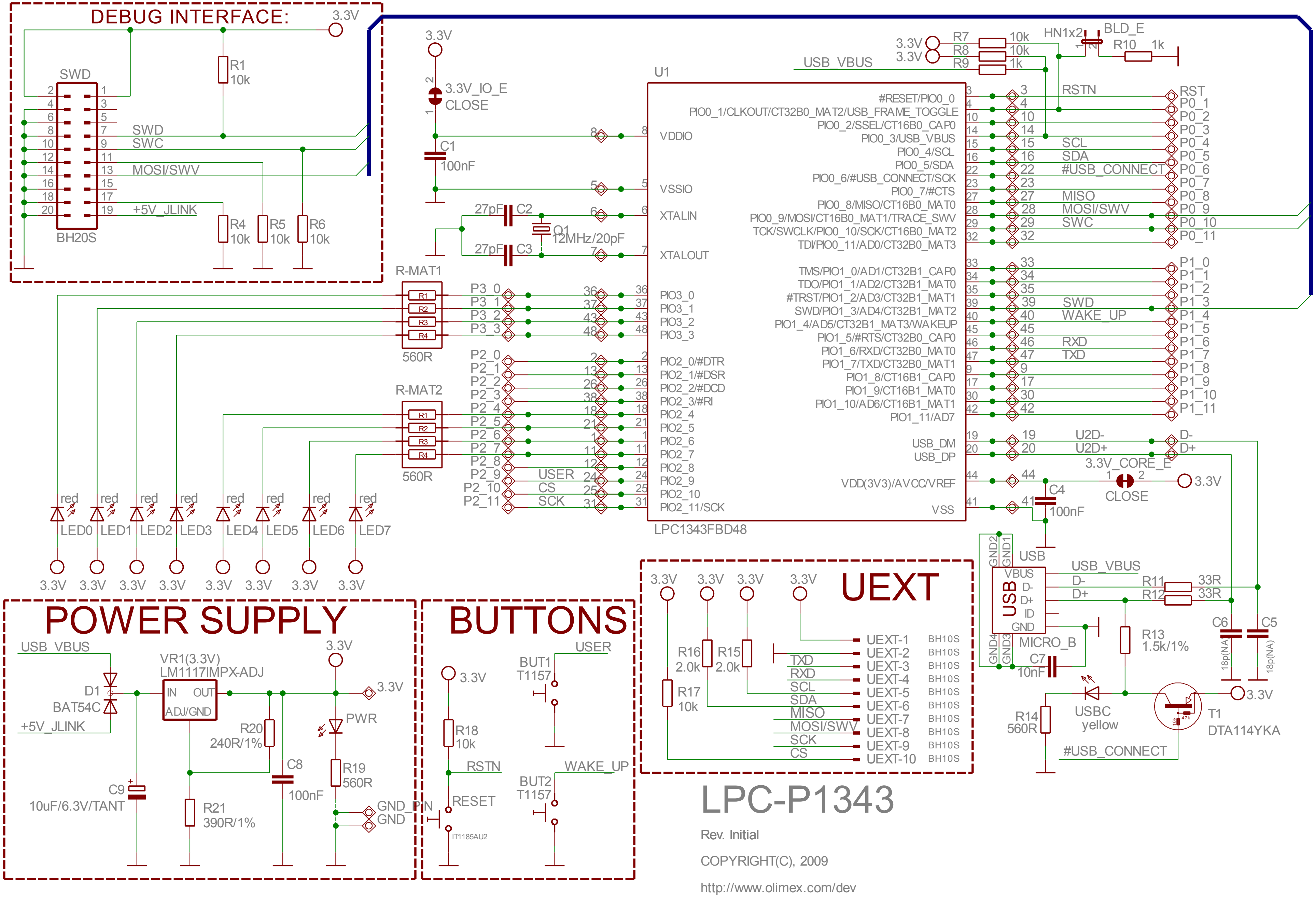


Figure D.2: Photo of the microcontrollerboard.


DEBUG INTERFACE:
SWD
BH20S
SWD
SWC
MOSI/SWV
+5V_JLINK
R1 10k
R4 10k
R5 10k
R6 10k
3.3V
3.3V_IO_E CLOSE
C1 100nF
27pF C2
27pF C3
Q1 12MHz/20pF
U1
LPC1343FBD48
VDDIO
VSSIO
XTALIN
XTALOUT
PIO3_0
PIO3_1
PIO3_2
PIO3_3
PIO2_0/#DTR
PIO2_1/#DSR
PIO2_2/#DCD
PIO2_3/#RI
PIO2_4
PIO2_5
PIO2_6
PIO2_7
PIO2_8
PIO2_9
PIO2_10
PIO2_11/SCK
#RESET/PIO0_0
PIO0_1/CLKOUT/CT32B0_MAT2/USB_FRAME_TOGGLE
PIO0_2/SSEL/CT16B0_CAP0
PIO0_3/USB_VBUS
PIO0_4/SCL
PIO0_5/SDA
PIO0_6/#USB_CONNECT/SCK
PIO0_7/#CTS
PIO0_8/MISO/CT16B0_MAT0
PIO0_9/MOSI/CT16B0_MAT1/TRACE_SWV
TCK/SWCLK/PIO0_10/SCK/CT16B0_MAT2
TDI/PIO0_11/AD0/CT32B0_MAT3
TMS/PIO1_0/AD1/CT32B1_CAP0
TDO/PIO1_1/AD2/CT32B1_MAT0
#TRST/PIO1_2/AD3/CT32B1_MAT1
SWD/PIO1_3/AD4/CT32B1_MAT2
PIO1_4/AD5/CT32B1_MAT3/WAKEUP
PIO1_5/#RTS/CT32B0_CAP0
PIO1_6/RXD/CT32B0_MAT0
PIO1_7/TXD/CT32B0_MAT1
PIO1_8/CT16B1_CAP0
PIO1_9/CT16B1_MAT0
PIO1_10/AD6/CT16B1_MAT1
PIO1_11/AD7
USB_DM
USB_DP
VDD(3V3)/AVCC/VREF
VSS
R7 10k
R8 10k
R9 1k
HN1x2
BLD_E
R10 1k
USB_VBUS
RSTN
SCL
SDA
#USB_CONNECT
MISO
MOSI/SWV
SWC
SWD
WAKE_UP
RXD
TXD
U2D-
U2D+
3.3V_CORE_E CLOSE
C4 100nF
R-MAT1 560R
R-MAT2 560R
USER
CS
SCK
LED0 LED1 LED2 LED3 LED4 LED5 LED6 LED7
red
USB
MICRO_B
R11 33R
R12 33R
R13 1.5k/1%
C5 18p(NA)
C6 18p(NA)
C7 10nF
R14 560R
USBC yellow
T1 DTA114YKA
POWER SUPPLY
USB_VBUS
D1 BAT54C
+5V_JLINK
VR1(3.3V) LM1117IMPX-ADJ
R20 240R/1%
R21 390R/1%
C8 100nF
C9 10uF/6.3V/TANT
PWR
R19 560R
GND_PIN
GND
BUTTONS
BUT1 T1157
USER
R18 10k
RSTN
RESET
BUT2 T1157
WAKE_UP
UEXT
R16 2.0k
R15 2.0k
R17 10k
TXD RXD SCL SDA MISO MOSI/SWV SCK CS
UEXT-1 UEXT-2 UEXT-3 UEXT-4 UEXT-5 UEXT-6 UEXT-7 UEXT-8 UEXT-9 UEXT-10
BH10S
LPC-P1343
Rev. Initial
COPYRIGHT(C), 2009
http://www.olimex.com/dev

Appendix E

MIPS processor overview

The following figure is a copy of Figure 4.17 of [2].