Imperial College of Science, Technology and Medicine	Department of Computing
Computing Science (CS) / Software Engineering (SE)	BEng and MEng
Examinations Part I	Integrated Laboratory Course

Laboratory work is a continuously assessed part of the examinations
and is a required part of the degree assessment.
Laboratory work must be handed in for marking by the due date.
Late submissions may not be marked.

Exercise: 1	Working: Groups of 2-4
Title: An Emulator/Assembler	
Issue date: 25/5/11	Due date: 3/6/11 and 9/6/11
System: Linux	Language: C

1 Aims

- To work in groups and undertake independent learning.
- To design and implement two sizeable and inter-related C programs.
- To learn about and use a wide range of C language features, including bit manipulation operators.
- To learn how to read and write binary files.
- To re-inforce your understanding of instruction set architectures and low level aspects of machine operation.

2 Introduction

In this exercise you are going to build an *emulator* and an *assembler* for a simple 'Reduced Instruction Set Computer' (RISC). You already have experience of writing x86 assembler programs; here you will be working with

a much simpler instruction set architecture, loosely based on the MIPS architectural model.[1] We'll call it 'IMPS' for Imperial's MIPS-like Processing System (baulk!). Your job is to write two programs:

- An IMPS emulator, i.e. a program that simulates the execution of an IMPS *binary file* on an IMPS machine.
- An IMPS assembler, i.e. a program that translates an IMPS assembly *source file* into binary file that can subsequently be executed by the emulator.

The two progams can be used together to execute IMPS assembly programs. It is recommended that you write the emulator first, as it will make it easier for you to test the output from your assembler.

You should work in groups of two, three or four. As always you should thoroughly test your code by using the sample assembler and binary files provided and also by writing your own. You should also design the code so that it can be written and tested incrementally.

All members of your group should contribute as equally as possible. You might want to divide the work up among the group members or you might each wish to work independently on a solution and then merge the solutions to form a polished deliverable. The primary objective is to ensure that you can *all* program proficiently in C by the end of the course.

The main assessment for the C component of the course will be a Lexis test at the end of week 6 which you will sit individually. This exercise is an opportunity for you to hone your skills in preparation for this test. By working in groups there is also an opportunity for the stronger programmers to help the weaker ones and you are encouraged to make this happen. As an incentive there will be a special prize, open to groups of size 3 or 4, who are able to demonstrate tangible improvements in the marks for individuals who have, until now, performed less well in programming.

You should provide a description of who did what and include this as a comment in your main programs.

We have provided some assembly language programs, **simple.s**, **factorial.s** and **matmult.s**, some IMPS-binary code programs, **simple.oout**, **factorial.oout** and **matmult.oout** and some result files (giving sample output from the emulator), **simple.res**, **factorial.res** and **matmult.res**. You should use these to check the outputs from your emulator and assembler. You can copy these files using the command **exercise ctests** or you can copy them from CATE.

[1] See `http://en.wikipedia.org/wiki/MIPS_architecture`

Part 1: The Emulator

The emulator main program **emulate.c** should begin by reading in IMPS-binary *object code* (i.e. compiled assembly code) from a *binary file* whose filename is specified as the sole argument on the command line. For example, to choose **factorial.oout** as object code:

```
% ./emulate factorial.oout
```

The object code consists of a number of 32-bit 'words', each of which represents either a machine instruction or data.[2]. (If any data needs to be jumped over, the first instruction will of course be an unconditional jump, over the data, to the program code proper.) These words should be loaded into a C data structure (e.g. an array or a structure of your own invention) representing the memory of the emulated IMPS machine. For simplicity you may assume that the memory has a capacity of 64Kbytes (i.e. $2^{16} = 65536$ bytes) which means that all addresses can be accommodated in 16 bits.

All memory locations and registers should be initialised to 0, reflecting the state of an IMPS machine when it is 'switched on'. Once the binary file has been loaded the emulator then runs the program contained within it by simulating the execution of each instruction in turn, starting with the first instruction (location 0). You should keep track of the current instruction using a C variable that represents the IMPS *program counter* (PC).

The IMPS memory is byte-addressable. However, all instructions are 32-bits (4 bytes) long and aligned on a 4-byte boundary, i.e. all instruction addresses are multiples of 4. In addition to the PC the IMPS also has 32 general-purpose 32-bit registers.

The execution of non-branching instructions will cause the PC to be incremented by 4 (bytes), so that it points at the next instruction in sequence. The execution of a branch instruction may change the PC to a specified target address. Instructions may modify values in both the IMPS memory and general-purpose registers. In addition to the memory array you will also need a data structure (e.g. another array) to store the values of the general-purpose registers. A good idea would be to define a C `struct` to capture the state of an IMPS machine.

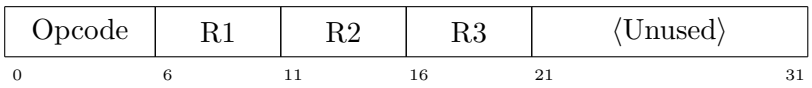
Program execution terminates when a `halt` instruction is reached. At this point the emulator should dump the value of the PC and each of the general-purpose registers to the standard C output stream `stdout`. Its job is then done.

[2]Of course, at the machine level there is no difference between programs and data!

3 Instruction format

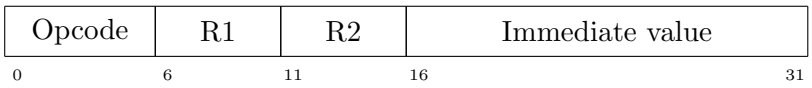
An instruction[3] consists of an operation code (*opcode*) which identifies the operation to be executed and various *operand* fields which may refer to registers, (constant) values or addresses. There are three instruction types:

R-type instructions:



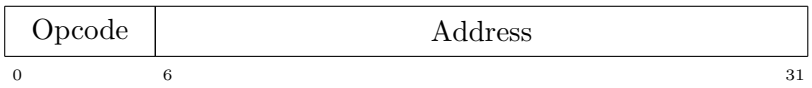
An R-type instruction uses registers as source and destination operands. The first register is the destination operand (where the result of the operation is stored). The second and third operands are the source operands.

I-type instructions:



An I-type instruction has two register operands and an immediate value operand. The first register is the destination operand (where the result of the operation is stored). The second register and immediate value are the source operands. Immediate values are 16-bit *signed* integers, so the sign bit is located in bit 16 of the instruction. Note that in order to map bits 16–31 into a C (signed) integer you will need to perform a *sign extension*.

J-type instructions:



A J-type instruction uses an immediate address as an operand. The purpose of a J-type instruction is to modify the program counter (PC), by replacing it with the address in the operand field. The address is a 26-bit unsigned value and is stored in bits 6–31 of the instruction. However, as the IMPS

[3]Note that throughout this document the most significant bit is numbered *0* and the least significant bit is numbered *31*. Please be aware that many textbooks, manual pages, web sites etc. use the opposite convention.

memory is assumed to have a capacity of only 64KB, the top 10 bits of the address field (bits 6 - 15) will always be 0.

As an example the 32-bit instruction `00001000010001000000000000000010` (`0x08440002` in hexadecimal) corresponds to the assembly language instruction `addi $1 $2 2` (see below). This is an I-type instruction which adds the constant 2 to the value in register `$2` and stores the result in register `$1`. The instruction is decoded as follows:

- Bits 0–5 contain the opcode which is `000001` (`addi` – see the table below)
- Bits 6–10 contain the destination register index (`00001`)
- Bits 11–15 contain the source register index (`00010`)
- Bits 16–31 contain the immediate value (`0000000000000010`)

Note that the instruction type, and hence the layout of the operands, depends on the opcode. Note also that memory can only be accessed through explicit loads and stores (the `lw` and `sw` instructions respectively).

3.1 Object code file format

In the IMPS binary files each word is stored in the byte order "least significant byte first .. most significant byte last". (The discussion of the example object code program shown in part 2 illustrates this byte order.) You should make sure that your memory layout also follows this convention, otherwise loading from and storing to memory (**lw** and **sw** respectively) will not work correctly.

3.2 Opcodes

IMPS currently supports the opcodes specified in Table 3.2.

- $R1$, $R2$ and $R3$ denote the registers referred to by the respective portions of the instruction, if present.
- C refers to the immediate value of an I-type instruction.
- A refers to the address in a J-type instruction.

Mnemonic	Opcode	Type	Interpretation
`halt`	0	N/A	Halts the machine
`add`	1	R	$R1 = R2 + R3$
`addi`	2	I	$R1 = R2 + C$
`sub`	3	R	$R1 = R2 - R3$
`subi`	4	I	$R1 = R2 - C$
`mul`	5	R	$R1 = R2 \times R3$
`muli`	6	I	$R1 = R2 \times C$
`lw`	7	I	$R1 =$ `MEMORY`$[R2 + C]$
`sw`	8	I	`MEMORY`$[R2 + C] = R1$
`beq`	9	I	`if` $R1 \equiv R2$ `then PC` $=$ `PC` $+ (C \times 4)$
`bne`	10	I	`if` $R1 \not\equiv R2$ `then PC` $=$ `PC` $+ (C \times 4)$
`blt`	11	I	`if` $R1 < R2$ `then PC` $=$ `PC` $+ (C \times 4)$
`bgt`	12	I	`if` $R1 > R2$ `then PC` $=$ `PC` $+ (C \times 4)$
`ble`	13	I	`if` $R1 \leq R2$ `then PC` $=$ `PC` $+ (C \times 4)$
`bge`	14	I	`if` $R1 \geq R2$ `then PC` $=$ `PC` $+ (C \times 4)$
`jmp`	15	J	`PC` $= A$
`jr`	16	R	`PC` $= R1$ ($R2$ and $R3$ unused)
`jal`	17	J	`$31` $=$ `PC` $+ 4$; `PC` $= A$

- The `lw` and `sw` instructions load and store whole *words*, so a load from address A loads the byte at A plus the bytes at $A + 1$, $A + 2$ and $A + 3$ to form a 32-bit word; similarly for stores. Note that unlike the program counter, these addresses **need not** be aligned on a 4-byte boundary.

Submit by Friday 3rd June 2011

4 What to do

Your program should be called **emulate.c** and should be broken down into a set of functions that collectively load and execute the object code in a specified binary file. The emulator should check for and report errors that occurred during program execution; examples include an attempt to execute an undefined operation (opcode) or to access a memory location with address greater than 65535.

The emulator involves the following key tasks; if you plan to work on the emulator together as a group then you should consider allocating a subset of these tasks to each member:

- Building a binary file loader.
- Writing the emulator loop, comprising:
  - Simulated execution of R-, I- and J-type instructions, with a special case for handling branch instructions within the I-type group.
  - Sign extension of 16-bit immediate values.
  - Outputting the emulator's state (on termination).

To help you test your emulator we have provided the test files **simple.oout**, **factorial.oout**, **matmult.oout**, **simple.res**, **factorial.res**, and **matmult.res**. When you run the emulator with an object code file <**file.oout**> it should produce on standard output the same content as the corresponding result file <**file.res**>.

Part 2: The Assembler

The assembler main program **assemble.c** should read in source code from an IMPS-source file whose filename is given as the **first** command line argument, and output IMPS-binary code to a file whose filename is given as the **second** command line argument (creating it if it does not already exist).

For example, to assemble the IMPS-source file **factorial.s** to an IMPS-binary file **factorial.oout**:

```
% ./assemble factorial.s factorial.oout
```

4.1 Two-pass Assembly

There are several ways of performing the assembly. Arguably the simplest way is to perform *two* passes over the source code. The first pass creates a *symbol table* which is used for associating *labels* (strings) with memory addresses (integers). In the second pass the assembler reads in the opcode *mnemonic* and operand field(s) for each instruction and generates the corresponding binary encoding of that instruction. As part of this process it replaces label references in operand fields with their corresponding addresses, as defined in the symbol table computed during the first pass.[4]

[4]Note that the symbol table encodes a mapping from strings to integers (c.f. `[(String,Int)]` or `String` → `Int` in Haskell).

4.2 Assembler File Format

Each (non-empty) line of an assembler file contains either an assembly *instruction* or an assembler *directive*, optionally preceded by a *label*.

Labels are strings that begin with an alphabetical character (`a`–`z` or `A`–`Z`) and end with a `:`, as in "`label:`". The value of the label is the address of the machine word corresponding to the position of the label.

Each instruction is mapped to (exactly) one 32-bit word during the assembly process (second pass). An instruction comprises an opcode mnemonic (e.g. `add`, `halt`, ...) and zero, one, or three operand fields, depending on the instruction type. There are currently no opcodes that accept two operands. The mnemonics used are listed in Table 3.2.

Registers are referred to by the symbols `$0`, `$1`, ..., `$31`. The syntax of each instruction depends upon the instruction type. In an R-type instruction the three registers appear after the opcode, separated by whitespace, as in:

```
add $2 $4 $3
```

In an I-type instruction the immediate operand can be either an integer constant or a label reference (label address), as in:

```
label2: muli $0 $1 7
        addi $2 $2 0x1AF
        subi $1 $0 label1
        beq  $5 $7 label2
        bgt  $1 $9 15
```

where `label1` and `label2` are label references. There are several points to note here:

- Intermediate values may be specified in decimal or hexadecimal. In the case of the latter, values will be preceded by the string "`0x`".
- If an immediate value corresponds to an (absolute) address, as in `0x1AF` and `label1` above, it will always be an unsigned value in the range 0–65535, due to our assumption regarding the emulated memory's size.
- The immediate value in a branch instruction is an *offset* in *words* from the address of the branch instruction. Thus, in the last example above, 15 denotes an *offset* of 60 bytes, not an absolute address; similarly `label2` has to be resolved to a word *offset* between the `beq` instruction and the address of `label2` (-3 in the example). The interpretation of a label reference in a branch instruction is thus different to the other immediate instructions.

In a J-type instruction the absolute address follows the opcode, as in the following examples:

```
jmp loop
jal 0x404C
```

where `loop` is a label reference.

4.2.1 Assembler directives

There are currently two assembler directives:

- `.fill` instructs the assembler to write a specified data item at the memory address corresponding to the current line of code. The data item occupies one word (32 bits) of memory.
- `.skip` leaves a 'gap' of a specified number of words at the corresponding position in memory. It is a way of reserving a block of memory without having to specify its contents.

4.2.2 Comments

Any text appearing after the last operand is treated as a comment and can be skipped.

4.3 Example Assembler program (factorial)

An example assembler program for computing the factorial of 5 is given below (it is deliberately not intended to be optimal!):

```
        jmp   start
n:      .fill 5
result: .skip 1
start:  lw    $1 $0 n       - Load n into $1
        addi  $2 $0 1       - Use $2 to store the result (initially 1)
loop:   beq   $1 $0 end     - If n == 0 then we are done
        mul   $2 $2 $1      - Multiply the result by n
        subi  $1 $1 1       - n--
        jmp   loop          - Go around again.
end:    sw    $2 $0 result  - Store the result in memory
        lw    $3 $0 result  - Load the result back into $3
        halt
```

4.4 Example Object Code Program (factorial)

The assembly process should generate the binary shown in Figure 1. Note

```
00111100 00000000 00000000 00001100          jmp   start
00000000 00000000 00000000 00000101  n:      .fill 5
00000000 00000000 00000000 00000000  result: .skip 1
00011100 00100000 00000000 00000100  start:  lw    $1 $0 n
00001000 01000000 00000000 00000001          addi  $2 $0 1
00100100 00100000 00000000 00000100  loop:   beq   $1 $0 end
00010100 01000010 00001000 00000000          mul   $2 $2 $1
00010000 00100001 00000000 00000001          subi  $1 $1 1
00111100 00000000 00000000 00010100          jmp   loop
00100000 01000000 00000000 00001000  end:    sw    $2 $0 result
00011100 01100000 00000000 00001000          lw    $3 $0 result
00000000 00000000 00000000 00000000          halt
```

Figure 1: Textual representation of the 'factorial' binary program

that this is the *textual* representation of the binary file, with the corresponding assembly code entries added for clarity. The spaces within the words are purely to aid readability and do not represent the byte order chosen by the C library functions for reading and writing 32 bit words (**fread()** and **fwrite()** respectively). In particular, if you view the binary file in blocks of 4 bytes (e.g. using `xxd -b -c4`) you will see the physical serialized byte order within each 32 bit word which is in fact opposite to the figure.

```
% xxd -b -c4 factorial.oout
0000000: 00001100 00000000 00000000 00111100  ...<
0000004: 00000101 00000000 00000000 00000000  ....
0000008: 00000000 00000000 00000000 00000000  ....
000000c: 00000100 00000000 00100000 00011100  .. .
0000010: 00000001 00000000 01000000 00001000  ..@.
0000014: 00000100 00000000 00100000 00100100  .. $
0000018: 00000000 00001000 01000010 00010100  ..B.
000001c: 00000001 00000000 00100001 00010000  ..!.
0000020: 00010100 00000000 00000000 00111100  ...<
0000024: 00001000 00000000 01000000 00100000  ..@
0000028: 00001000 00000000 01100000 00011100  ..`.
000002c: 00000000 00000000 00000000 00000000  ....
%
```

Submit by Thursday 9th June 2011

5 What to do

As with the emulator, your code should be broken down hierarchically to match the structure of the problem. You can assume any assembly program being processed is syntactically correct.

The assembler involves the following key tasks, which you should consider dividing up among your group members:

- Constructing a binary file writer.
- Building a symbol table abstract data type (ADT).
- Designing and constructing the assembler, comprising:
  - A tokenizer for breaking a line into its label, opcode and operand field(s)
  - Pass 1
  - Pass 2, including, for example:
    - A function for assembling R-type instructions
    - A function for assembling I-type instructions
    - A function for assembling J-type instructions

Note that the symbol table can be used to map labels into addresses and also to map opcode mnemonics into the opcodes themselves. Hint: You might want to explore the idea of using C's *function pointers*, similar to higher-order functions in Haskell, in order to get from an integer opcode to a C function for assembling the corresponding machine operation. This will avoid the need to build a large switch statement or nested conditional in your simulation loop. Alternatively, you could devise a method of encoding with each opcode its expected operand count.

To help you test your assembler we have provided the test files **simple.s**, **factorial.s**, **matmult.s**, **simple.oout**, **factorial.oout**, and **matmult.oout**. When you run your assembler with an assembly code file <**file.s**> it should assemble to an IMPS-binary file which is **bitwise identical** to the supplied corresponding IMPS-binary file <**file.oout**>. (You can check if two files are bitwise identical by giving them as arguments to the Linux command **diff**. If they are identical **diff** will be silent.)

6 Writing test programs

We have provided some test files but you should develop a suite of programs of your own to test both the assembler and emulator. You are encouraged, as a class, to build and share a repository of sample assembler programs. We will make any interesting programs available to the class if they are emailed to Peter Cutler (psc).

7 Optional Extras

If you want to hone your C programming skills further and/or want to learn more about low-level machinery, you might consider the following extensions.

- Extend the instruction set, for example to include some additional MIPS-like instructions/macros.
- Implement a stack, with associated operations such as `push`, `pop`, `jsr` (c.f. an x86 `call`) etc. You will need to define space for the stack itself, as a default component of the binary file.
- Build a *relocating* loader. The assumption of the current loader is that programs are loaded into memory starting at address 0. Rewrite it so that it can load the code starting at a different location. This requires replacing all absolute memory addresses accordingly.
- Extend your assembler and binary loader to allow programs to be developed in different files with arbitrary cross-references. To do this you will need to provide a mechanism for identifying labels that can be referenced from outside the file (e.g. an *export list*) and, similarly, a way of importing labels from other files. You will then need to extend the binary format to include some form of header data, in order that all external label references can be resolved.

Your programs **emulate.c** and **assemble.c** should have the functionality described in the specification. The programs should expect the arguments and produce the output described in the specification. If you have written code for the optional part which requires any alteration to the arguments or output you should submit it in separate **.h** and **.c** files to the assessed part (and with appropriate comments explaining what it does).

8 Prizes

There are three prizes on offer, for:

- The best solution, as judged by a combination of the quality of the code and the sophistication of any optional add-on features (see above).
- The most interesting IMPS assembler program. Now, there's a challenge!
- The group(s) that have succeeded most in improving the programming competence of their less proficient members. The improvements will be measured based on the C Lexis test results (suitably normalised) relative to the average marks for previous Lexis tests.

Submission

- You will need to make two submissions for this exercise, the first with the main program **emulate.c** and the second with the main program **assemble.c**.
- Please make sure that you make a note of which sections of code were written by which member of your group (if a section was written by more than one person in a collaborative session you can note this) and include this as a comment.
- Put your emulator related **.h** and **.c** code of which **your main program must be "emulate.c"** into a directory (we don't need to know the name of that directory); **cd** into that directory; and type:

  ```
  catejarbuild c1
  ```

  You should submit the emulator by Friday 3rd June 2011

- You should ensure that only the main program and the files necessary for it to compile are present in the directory containing **emulate.c** .
- Submit your file **Emulator.jar** via CATE in the usual way.
- Put your assembler related **.h** and **.c** code of which **your main program must be "assemble.c"** into another directory (we don't need

to know the name of that directory); **cd** into that directory; and type:

```
catejarbuild c2
```

You should submit the assembler by Thursday 9th June 2011

- You should ensure that only the main program and the files necessary for it to compile are present in the directory containing **assemble.c** .
- Submit your file **Assembler.jar** via CATE in the usual way.

9 Assessment

Correctness	(Emulator)	10
Design, style, readability	(Emulator)	10
Correctness	(Assembler)	10
Design, style, readability	(Assembler)	10
Total		40