RA FS 21 Series 3

Alp Sari Eren, Sepehr Sameni, Abdelhak Lemkhenter

The third series has to be solved and uploaded to ILIAS until Tuesday, 12. April 2021 at 3 p.m. If questions arise, you can use the Piazza forum at any time. Any problems should be communicated as soon as possible, we will be happy to help.
Have fun!

Theoretical Questions

Total score: 11 points

1 Performance calculations (3 points)

Suppose a CPU is clocked at 500Mhz. Suppose further that the CPU performs the following operations (with the specified amount of time):

`ALU` 4nsec,

`LOAD` 8nsec,

`STORE` 12nsec,

`Branch` 6nsec.

(a) First, we assume that all operations are carried out equally often.

- How much faster/slower is a machine that needs 6 clock cycles for the `LOAD` instruction?
- How much faster is a CPU when its STORE works twice as fast?

(b) Now suppose not all operations are carried out equally often: ALU 50%, LOAD 15%, STORE 25%, BRANCH 10%. Calculate the ideal CPI of the processor (without taking stalls through memory access into account).

2 Using a stack in subroutines (2 points)

Give two possible reasons why one needs the stack for assembler subroutines.

3 ALU & `SLT` (2 points)

- What happens during the `slt` command in the ALU?
- How does the ALU support the `slt` command?

4 `loadi` (2 points)

Specify how the loading of a (32-bit) constant into a MIPS registers can be implemented with the MIPS instruction set.

5 ALU: OPCodes (2 points)

Describe how the control bits of the ALU must be set for the following commands:

`and or add subtract slt nor`

Explain in addition the relationship between these bits/commands and the individual elements of the ALU.

Programming part

Your task is to complete the given program structure as follows:

(a) Download the files for series 3 from ILIAS and study them carefully. Try to understand what the existing parts mean.[1]

(b) Put your name (and the name of a possible exercise partner) in the space provided in the files `compile.c`, `mips.c` und `memory.c`.

(c) Complete the function `main` in `compile.c` such that on the command line the expression to compile and the filename of the file, in which the compiled program is stored, can be passed as parameters. At incorrect usage there should be an output like

```
usage: <commandname> expression filename
```

(where `<commandname>` is the actual program name, even if the program is renamed). If entered correctly, so for example

```
./compile '(3*(45+6))+12' test.mips
```

should

```
Input:   (3*(45+6))+12
Postfix:  3 45 6 + * 12 +

MIPS binary saved to test.mips
```

be displayed.

Hints:

- The main work performs the already implemented function `void compiler(char* exp, char *filename)` (in `compiler.c`). You just have to make sure that `main` is called with the correct number of arguments and that they are correctly passed to `compiler`.
- `http://publications.gbdirect.co.uk/c_book/chapter10/arguments_to_main.html`

(d) Complete the function `loadFile(char*)` in `memory.c`, so that a generated MIPS binary file can be loaded into the memory and executed.

You can assume that the MIPS binary file is equivalent to the expected image in the memory, i.e. the individual words are stored in big endian byte order in the file without a break.

Hints:

- Have a look at `void store(word w)` in `compiler.c`
- Create some files (using e.g. `./compile 1+1 test.mips`) and view them with a hex editor, in order to better understand the file format.
- `http://publications.gbdirect.co.uk/c_book/chapter9/input_and_output.html` and the following sections.

(e) Complete the function `error` in `mips.c`, so that the error messages can be output correctly formatted. Also have a look at the macro `ERROR` in `mips.h`. The error message should always specify in which function, on which line and in which file the error has arose and print out a detailed error message. A call

```
ERROR("Unknown opcode: %x", instruction->i.opcode);
```

in `mips.c`, function `undefinedOperation` on line 166 for example should result in

```
undefinedOperation in mips.c, line 166: Unknown opcode 2b
```

(assuming `instruction->i.opcode == 0x2b`).

After the output of the error, the program should be terminated by `exit(EXIT_FAILURE)`.

Hints:

- `http://linux.die.net/man/3/vfprintf`

[1] `compiler.c` and `compiler.h` do not need to be studied in detail, important points are mentioned below. For the remaining parts there is a bit of theory in the appendix.

- `http://linux.die.net/man/3/va_start`

(f) Make sure that your program compiles without any errors or warnings, ensure this with `make`

This is an essential requirement to pass the programming exercise!

(g) Make sure that your program completes all tests (the given ones and your own) without any errors or warnings, ensure this with `make test`

This is an essential requirement to pass the programming exercise!

(h) Create a Zip file named `<lastname>.zip` (where `<lastname>` is replaced with your last name) from your solution.

(i) Hand in your solution electronically by uploading the file to ILIAS.

A Theory

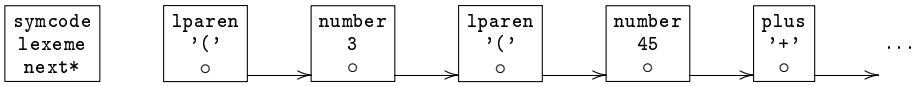
In `compiler.c` and `compiler.h` you can find an implementation of a simple compiler for arithmetic expressions. Our "programming language" is given in the following Extended Backus–Naur Form (EBNF):

```
digit      = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
number     = digit, {digit};
factor     = number | "(", expression, ")";
term       = factor, { ( "*" | "/" ), factor };
expression = term, { ( "+" | "-" ), term };
```

A possible "program" would be for example:

```
(3*(45+6))+12
```

This input will first be separated by a lexical scanner (sometimes called lexer) into its individual parts (in this case symbols and numbers), so called tokens. Some additional informations are added and stored as a list; the above expression for example will become:



In `compiler.c` there is a simple compiler, which transforms the given expression into postfix notation. The above input will become:

```
3 45 6 + * 12 +
```

Expression in postfix notation can be easily processed using a stack machine. Every argument will be passed on a stack and operations are applied to the elements at the top: the elements are loaded into the register of the processor, the operation is performed and the result is pushed back onto the stack.

Expression in postfix notation can be easily transformed into assembly code, the above code fragment for example will become (in pseudo assembly code)

```
push  3         // Push  3 onto the stack
push 45         // Push 45 onto the stack
push  6         // Push  6 onto the stack
pop   A         // Pop element at the top of the stack into register A
pop   B         // Pop element at the top of the stack into register B
add   A, A, B   // Add register A and B, store result in A
push  A         // Push element in A onto the stack
pop   A         // Pop element at the top of the stack into register A
pop   B         // Pop element at the top of the stack into register B
mult  A, A, B   // Multiply register A and B, store result in A
push  A         // Push element in A onto the stack
push 12         // Push 12 onto the stack
pop   A         // Pop element at the top of the stack into register A
pop   B         // Pop element at the top of the stack into register B
add   A, A, B   // Add register A and B, store result in A
push  A         // Push element in A onto the stack
stop            // Exit the program
```

The final result of the calculation will be stored in the stack.