5DV118 Computer Organization and Architecture Fall 2012
Obligatory Exercise 2
Due date: January 07, 2013 at 1640 (4:40pm)

1 Overall Task and Goal

The task of this exercise is to write a simulator for a small subset of the MIPS instruction set. The goal is to simulate the internal control structure of a single-cycle implementation as faithfully as practical within an imperative programming language, and thereby to obtain a better understanding of the internal operation of a modern microprocessor.

2 Design and Implementation Requirements

2.1 Input and Supported Processor Features

The simulator should operate directly on mnemonic representations of instructions; *e.g.*, `mul $1, $0, $at` and not on the numeric representation; *e.g.*, `0x71014802`. Thus, the simulator need not and should not deal with the numerical representations of the instructions.

g1. The simulator should accept as input a file containing a sequence of MIPS instructions in mnemonic format, one per line, such as:

```
add $t0, $t1, $t2
sub $s0, $t0, $v0
beq $3, $8, -2
exit
```

g2. The following instructions must be supported: `add`, `sub`, `and`, `or`, `nor`, `slt`, `lw`, `sw`, `beq`, and `nop`.

g3. The directive `exit` must also be supported, indicating that the simulation should terminate. The simulation must begin with the first instruction in the list, and must be terminated when an *exit* directive is reached.

g4. There are separate data and instruction memories, and the first instruction should be found beginning at location `0` of the instruction memory.

g5. 1000 bytes of data memory, numbered 0-999, must be supported.

g6. A minimum of 1000 bytes of instruction memory must be supported.

g7. No labels for instruction or data memory locations need be supported.

g8. Symbolic names (*e.g.*, `$s2`) for registers must be supported.

g9. Each register and memory location has the initial value of `0`.

2.2 Simulation Requirements

It is essential that the simulator implement the control structure of a single-cycle implementation of the processor architecture, as illustrated in Figures 4.17-4.21 of the textbook. The details of what is required are sketched in that which follows.

s1. Each major block in the figures of the book, including in particular Instruction Memory, Data Memory, Registers, ALU, Control, ALU Control, PC, and the Add unit for the PC must be implemented as separate, logical units which take as inputs and outputs the lines which are shown in the figures.

s2. Each control line, including RegDist, Branch, MemRead, MemtoReg, ALUOp, MemWrite, ALUSrc, and RegWrite must be implemented as a distinct Boolean variable.

s3. Each step of the simulation must proceed by determining and then setting the appropriate values of these control lines, and then triggering the events which must occur for those control values. The `op` field of the instruction (bits 31-26) are used to determining the values of the control lines. The table of Figure 4.22 of the textbook may be helpful in this regard.

s4. For arithmetic instructions with `op=0`, there must be a similar decoding of the `funct` field, the rightmost six bits of the instruction. This decoding provides details of the required operation to the ALU.

s5. For this to work, the simulation must begin by representing each instruction by the numerical values for its constituent fields. For example, the instruction `add $t0, $t1, $t2` is represented as `0 9 10 8 0 32`. This should be done only once for each instruction in the input file, and not once for each time the instruction is executed. Note that the representation of an instruction depends upon its type. For example, the representation of `lw $t1, 8($t2)` is `35 10 9 8`. The simulator must of course be aware of how many bits are associated with each field of the representation.

2.3 Interface requirements:

The simulator must be capable of displaying its operation on a per-instruction basis. The interface may be either graphical or textual, but must meet the following requirements.

i1. The interface must display the following:

- i1.1. Each instruction of the program, with a pointer or highlighting indicating the one which is currently being executed.
- i1.2. The numerical constituent fields of each instruction.
- i1.3. The current value of each register which has been changed during the execution of the program.
- i1.4. The current value of the program counter.
- i1.5. The current value of each memory location which has been changed during the execution of the program.

i2. There must be a choice of whether values are displayed in decimal or hexadecimal. Minimally, this may be implemented as an option at startup time, but it would be preferable to allow the form of display to be changed during the execution of the simulator.

i3. The interface must support at least the following operations:

`Step` Execute the next instruction and then wait.

`Run` Run the program until it ends.

`Reset` Reset to the initial state when the program file was loaded. This should be possible even when the program is in (a possibly unending) loop.

2.4 Development requirements

d1. The software may be written in C or Java. If C is used, then basic libraries for building GUIs, such as gtk, may also be used. The final product must compile, load, and runs on the departmental Linux machine `salt`. Submissions which require other systems for any of these steps will not be accepted. Documentation on how to build the executable from source must be included as part of the user manual. See Section 3 below.

2.5 Grading and Extra Credit

The basic assignment is worth up to 50 points. Up to 25 extra points may be obtained by supporting extra instructions.

xc1. 5 additional points may be obtained by supporting the immediate instructions `addi`, `slti`, `ori`, and `lui`.

xc2. 5 additional points may be obtained by supporting the shift operations `sll`, `sllv`, `sra`, `srav`, `srl`, and `srlv`.

xc3. 5 additional points may be obtained by supporting the jump instructions `j`, `jal`. `jalr`, and `jr`.

xc4. 10 additional points may be obtained by supporting the multiplication instructions `mult`, `multu`, and `mul`.

An effective way to check your simulator is to run the input file with SPIM simulator and see whether it produces the same results.

3 User Manual

The final submission must include a user manual, written in English, and prepared using a professional typesetting tool such as LaTeX, or else a document processor such as LibreOffice Writer. The electronic version of the final submission must be in PDF. Word-processor formats such as `.doc`, `.docx`, `.odt`, `.rtf`, and the like are not acceptable for the final version, nor is straight ASCII text.

It may be assumed that the reader of the manual has some familiarity with the MIPS instruction set, but it must otherwise be self contained. In particular, it should not be necessary to study the source code or to experiment in order to determine how to use the software. It should thus contain step-by-step instructions, preferably with screenshots or other illustrations, as well as a reference section which indicates which instructions are supported.

The user manual must also contain a description of how to build the executable from source. This information should be in a section of the manual which is separate from that which provides information on how to use the program.

Half of the total quality points will be based upon an evaluation of the user manual.

Clarity is the most important aspect of this manual. It must be easy for a user to see how to use the software. While perfection in English is not a grading criterion, submissions which are written in a sloppy fashion, or which have not been run through a spelling checker will lose points.

4 Submission Rules

A printed copy of the user manual must be placed in the appropriate course mailbox on the fourth floor of MIT-huset. The user-id of each group member for the submission must be indicated clearly on a cover page of the printed submission. In addition, an electronic copy of the manual (PDF), together with a file containing the source code, including any necessary make files and the like (all plain ASCII), must be submitted in the form of a gzipped tarball (`.tar.gz`) to `labs-5dv118@cs.umu.se`, with the user-id of each group member given in the subject line of the message. A submission is not considered to be complete until both paper and electronic versions have been delivered. (Note that a printed version of the source code need not be submitted.)

A suite of test programs may be supplied at a later date. In that case, the results of running the program on the programs of that suite will be a required part of the submission as well.

If all members of the group will be away at the submission deadline, alternate arrangements for the submission of the printed user manual are possible, but in this case, it is essential to discuss those arrangements with the grader well in advance.

5 Further Guidelines

g1. Solutions may be developed and submitted by groups of up to three individuals.

g2. Late solutions will receive $p\%$ of the quality points determined by the grader, where $p = 100 - 10 * (\text{number of working days or partial working days late})$.

g3. Students who have already completed the "laboratory" part of 5DV008 are permitted to submit this exercise for points only. Please inform the grader if you submit something for points only.