5DV008 **Computer Architecture** **Fall 2010**

Obligatory Exercise 2
Due date: December 22, 2010 at 0800 (8am)

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 the 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.

Version: December 06, 2010

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 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 any language, as long as the final product compiles, loads, and runs on the departmental Linux systems. Submissions will be evaluated on the local Linux systems, and submissions which require other systems for any of these steps will not be accepted. It is highly recommended that the software be written in *C*, *C++*, or *Java*, because it is unlikely that the course staff will be able to help you with problems in other languages. For *C* and *C++*, the *gcc* compiler must be used.

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 Submission Rules

The submission must also include a user manual which describes how to use the program and how to interpret the results. This manual will be worth 15 out of the 50 points for the base task.

A printed copy of the solution 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 must be submitted to `labs-5dv008@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 copies have been delivered.

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.

4 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 $p = 10 -$ number of working days or partial working days late.

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