ECE 353 Lab 2

Motivation: The purpose of this lab is threefold:

- (A) To gain further practice in writing C programs, this time of a more advanced nature than seen before.
- (B) To reinforce what you learned about pipelined machines in ECE 232 and give you some new insights into how these work. Also to understand the impact of some key parameters on machine performance.
- (C) To provide some inkling of how you can simulate (using a sequential machine) machines where multiple events can occur in parallel.

This work is substantially more complex than the previous C program you wrote, and I recommend that you start doing this right away. This is not something you can expect to complete over a weekend (unless you are a very good programmer). Due to the complexity of this lab, it has much more weight than the previous labs in this course.

Statement of Work: Simulate a simple pipelined machine. The design will include writing a detailed simulation program (written in C) which simulates the MIPS architecture, cycle by cycle. The simulator must be cycle-accurate with respect to register contents, i.e., at the end of each simulated clock cycle, the simulated registers should have the same contents as the actual machine would. The simulator user interface tells the simulator to output the contents of all the registers. You will then use this simulator to provide performance estimates and study the impact of various parameter values.

The machine you will simulate is a subset of the MIPS architecture that you learned about in ECE 232. It implements just the following instructions: `add, sub, addi, mul, lw, sw, beq`. Only integer operations are available; no floating point units or instructions exist. The MIPS instruction format is used: see your ECE 232 text (or do an online search). (Note that the multiplication instruction being implemented is `mul`, not `mult`. That is, Recall that the syntax for `mul` is `mul $a, $b, $c,` meaning that we multiply the contents of registers \$b and \$c and place the least significant 32 bits in register \$a. In MIPS, this is a pseudoinstruction; here, we are assuming it is implemented directly in hardware). The opcode and function code that we will use for `mul` is the same that MIPS uses for `mult`.

There are 32 registers; register \$0 is hardwired to 0. In addition, there is a Program Counter (PC) and the rest of the items in MIPS (see Hennessy and Patterson).

Memory: The memory is just one word wide, and so is the memory bus. The memory accepts the address and R/W information as inputs and completes an access request in `c` CPU cycles. There is no cache in this machine. There are physically separate memories for the instruction and for the data; so data loads/stores will not interfere with the instruction fetching. Each of these memories is 2Kbytes in size; *data and instruction address spaces are distinct.* Each memory access takes `c` cycles, where `c` is an input parameter to the simulator. Data memory is initialized to 0 at the start of each simulation run.

Assume that the program is preloaded into memory, starting at location `0x00000000`: the PC should start execution by fetching the instruction stored in this location.

CPU: The CPU is the pipelined MIPS you studied in ECE 232. Recall that it has five stages: IF, ID, EX, MEM and WB. There are pipeline latches between the stages. Again, this is material that was covered in detail in ECE 232.

The processor does not do branch prediction; when the ID stage detects a branch, it tells the IF stage to stop fetching and flushes the IF_ID latch (i.e., puts a NOP in it). When the branch is resolved (in the EX stage), IF is allowed to resume fetching instructions based on the branch outcome.

The output of the simulator will be the following statistics:

- Utilization of each stage. Utilization is the fraction of cycles for which the stage is doing useful work. Just waiting for a structural, control, or data hazard to clear in front of it does not constitute useful work.
- Total time (in CPU cycles) taken to executed the MIPS program on the simulated machine. (This is NOT the time taken to execute the simulation; it is the time taken by the machine being simulated.)

Do not worry about interrupts. Also, this machine does not support out-of-order execution (that would require a reorder buffer and more sophisticated approaches to deal with data hazards: if you curious, look up "scoreboarding" and "Tomasulo's algorithm" in any computer architecture book or online). Only one instruction can occupy a given pipeline stage at any time. This may cause other instructions to be delayed (e.g., if a multiply instruction is being executed and an independent `sw` instruction is behind it, the `sw` will have to wait until the multiply is complete before it can pass through the execute stage). Also, our machine does not support data forwarding. You should assume that register writes are completed in the first half of a clock cycle and that register reads are carried out in the second half.

Assume that the program being executed is read by the simulator from a file. Your simulator should include code which parses the program. In addition to the MIPS instructions being simulated, there is another simulator directive that should be placed at the end of the assembly program: `haltSimulation`. When the simulator sees this directive, it knows the last assembly instruction has been fetched.

You should assume that all assembly language text will be lower-case. As mentioned previously, there is an additional `haltSimulation` directive: this is not part of MIPS assembly but is placed at the end of a program to let the simulator know the end of the program has been reached.

No labels will be used in this simulator; branch instructions will be limited to using offsets. Recall from ECE 232 that these offsets are calculated with respect to the instruction *following* the beq instruction (e.g., an offset of 1 has as its branch target an instruction two locations away from beq, not one).

Reading from a file will be through the `fgets()` function. This returns a character pointer to the line which has been fetched. You can assume that an instruction is limited to no more than 100 characters.

Your simulator should operate in one of two modes:

- Single-cycle Mode: In this mode, the program executes cycle by cycle. After each cycle, it displays the contents of all 31 registers (there is obviously no point in printing the $0 register) and the PC; it moves on to the next cycle after some key is hit on the keyboard.
- Batch Mode: The entire program is executed, followed by an output of statistics and the register contents.

The statistics that the simulator collects are as follows:

- Utilization of each pipeline stage.
- Total execution time (for batch mode operation).

Your code should include the following functions:

- `char *progScanner(...)`: This reads as input a pointer to a string holding the next line from the assembly language program, using the `fgets()` library function to do so. `progScanner()` removes all duplicate spaces, parentheses, and commas from it from it and a pointer to the resulting character string will be returned. Items will be separated in this character string solely by a single space. For example `add     $s0, $s1, $s2` will be transformed to `add $s0 $s1 $s2`. The instruction `lw $s0, 8($t0)` will be converted to `lw $s0 8 $t0`. This is done most easily by using a Finite State Machine abstraction; we will cover this in the lecture. If an error is detected (e.g., 8($t0(instead of 8($t0) or a missing `)`,), this should be reported and the simulation should then stop.

- `char *regNumberConverter(...)`: This function accepts as input the output of the `progScanner()` function and returns a pointer to a character string in which all register names are converted to numbers.

  MIPS assembly allows you to specify either the name or the number of a register. For example, both $zero and $0 are the zero register; $t0 and $8 both refer to register 8,and so on. Your parser should be able to handle either representation. (Use the table of registers in Hennessy and Patterson or look up the register numbers online.)

  The code scans down the string looking for the $ delimiter. Whatever is to the right of this (and to the left of a space or the end of string) is the label or number of the register. If the register is specified as a number (e.g., $5), then the $ is stripped and 5 is left behind. If it is specified as a register name (e.g., $s0), the name is replaced by the equivalent register number). If an illegal register name is detected (e.g., $y5) or the register number is out of bounds (e.g., $987), an error is reported and the simulator halts.

- `struct inst parser(...)`: This function uses the output of `regNumberConverter()`. The instruction is returned as an `inst` struct with fields for each of the fields of MIPS assembly instructions, namely opcode, rs, rt, rd, Imm. Of course, not all the fields will be present in all instructions; for example, `beq` will have just two register and one Imm fields.

  Each of the fields of the `inst` struct will be an integer. You should use the enumeration type to conveniently describe the opcodes, e.g., `enum inst {ADD,ADDI,SUB,MULT,BEQ,LW,SW}`. You can assume that the assembly language instructions are written in lower-case so there is no clash with these enum quantities.

  If an illegal opcode is detected (e.g., a typo such as `sbu` instead of `sub`) or an error is detected in any other field, the error is reported and the simulation is stopped. The error type should be reported; these error types are:

  - Illegal opcode (see the example above).
  - Immediate field that contains a number too large to store in the 16 bits that are available in the machine code format.
  - Missing $ in front of a register name (e.g., a program which says `s0` instead of `$s0`).
  - Any other error categories you can think of.

  Also, the simulator should flag an error and stop further activity whenever a misaligned memory access occurs. Recall from ECE 232 that a misaligned integer access is one where the memory address of the lw or sw source or target is not a multiple of 4.

The function `parser()` will place the parsed instruction (in the form of a struct) in the linear array that is used to represent the Instruction Memory.

- `void IF(...), void ID(...), void EX(..), void MEM(...), void(WB)`: These functions simulate activity in each of the five pipeline stages. All data, structural, and control hazards must be taken into account. Keep in mind that several operations are multicycle and that these stages are themselves not pipelined. For example, if an add takes 4 cycles, the next instruction cannot enter EX until these cycles have elapsed. This, in turn, can cause IF to be blocked by ID. Branches will be resolved in the EX stage. We will cover in the lectures some some issues related to realizing these functions.

  Keep in mind that the only requirement is that the simulator be cycle-by-cycle register-accurate. You don't have to simulate the control signals. So, you can simply pass the instruction from one pipeline latch to the next.

Submission Instructions: Your program will be tested in the quark gcc environment, so you should ensure that it runs successfully *in that environment.* Recall that C programs are sometimes not portable: just because a program runs successfully in one environment is no guarantee that it will do so in any other environment.

Your code will be tested by an automated system. To ensure consistency, do the following:

- The code file should be called **sim-mips.c**
- The code available via the code1.c link should be used at the start of your program. Your code (including variable names) should be consistent with this.
- Code available via the code2.c link should be added to the simulation of every cycle.
- Code available via the code3.c link will output the register value to screen at every cycle and wait for the ENTER key to be pressed before proceeding to the next cycle.
- Code available via the code4.c link should be added to the end of your simulation program to output statistics in batch mode and to close files.

To submit the program, use the same procedure as for Labs 0 and 1.

Also, upload on moodle the summary sheet listing the primary author and tester of each function. *The workload should be distributed fairly evenly among all group members.*

Commonly Asked Questions

1. The simulation will execute as a single thread. How, then, can it simulate a pipeline which has multiple things happening in it at the same time?

You have to order things within the simulation so that even though everything happens sequentially in the simulator, the result is the same as if the operations took place in parallel, just as in the real machine. Note that in MIPS, register writes happen in the first half of the clock cycle while register reads happen in the second half. You also have to ensure that the sequential simulation does not spuriously overwrite needed by some stage. The simplest way of doing this is to write functions to represent the activity in each stage and call the functions backwards in the main routine. That is, your `main()` function should call the functions implementing activity in the WB, MEM, EX, ID, and IF stages in that order. Do a small hand simulation to convince yourself that this approach works.

2. *How can information be moved from one stage to the next?*

   Recall from ECE 232 that there are pipeline latches between stages. Each stage has some memory in which to place items it is forwarding to the next stage. You should simulate the pipeline latches, being sure to ensure that all the information needed by a stage is provided and that information needed for an earlier instruction is not prematurely overwritten by a later instruction.

3. *How can we account for multicycle operations within stages?*

   The memory and arithmetic/logical instructions can each take multiple cycles. You have therefore to maintain a counter in each of the IF, EX and MEM stages, to count the total number of cycles consumed so far by an operation. Note that this counter cannot be an ordinary ("automatic") variable within the function: such a variable would be reset every time the function is called. Instead, one of two approaches can be taken. Either the scope of the counter variable can be made global or it can be declared to be `static`. A `static int` variable, for example, is an integer which maintains its value across function calls.

4. *How do we calculate execution time?*

   This is a clock-driven simulation. (Recall from the lecture that there are two types: clock-driven and event-driven). Your `main()` routine should have a loop that increments a clock counter to represent the passage of time. Execution time does NOT mean the time taken by your laptop/desktop to run the simulation.