

# ECE 353 Lab C<sup>1</sup>

**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 in Lab A.
- (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 we recommend that you start doing this right away. Due to the complexity of this lab, it has twice the weight of Lab A.

## **Statement of Work:**

Simulate a simple pipelined machine. The design will include writing a detailed simulation program (written in C) which simulates the architecture, cycle by cycle. 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 we covered in ECE 232. It implements just the following instructions: `add`, `sub`, `addi`, `mul`, `lw`, `sw`, `beq`, as well as “quasi-instructions” which are described below. Only integer operations are available; no floating point units or instructions exist. The instruction format is the same as that of the MIPS architecture: see your ECE 232 text (or do an online search). (Note that the multiplication instruction being implemented is `mul`, not `mult`. 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`.

To simplify your task of parsing the input program, you can drop the commas between fields and directly use register numbers. For example, `add 3 4 5` says to add the contents of registers \$4 and \$5, and place the result in register number \$3.

There are 32 registers; register \$0 is hardwired to 0. In addition, there is a Program Counter (PC) and any other hardware that you need to provide.

**Memory:** The memory is just one word wide, and so is the memory bus. The memory

---

<sup>1</sup>This should be read in conjunction with the lecture slides, which provide further details and hints.

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*. As mentioned before, each memory access takes  $c$  cycles, where  $c$  is an input parameter to the simulator.

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 pipelined.

- Stage 1: Fetch. This is a semi-autonomous unit, responsible for fetching instructions, and is described in greater detail later.
- Stage 2: Instruction decode and register file read. It takes one cycle for all instructions.
- Stage 3: Execution. Arithmetic operations are executed in this stage. Also, comparisons associated with conditional branches are carried out here.
- Stage 4: Memory access and conditional branch execution.
- Stage 5: Write back.

Fetch Unit: The fetch unit is loosely coupled to the rest of the system. It has its own adder (separate from that in the ALU) for suitably incrementing the Program Counter. Its job is to bring in instructions and place them in a *fetch buffer* that can be accessed by the decode stage. The fetch buffer has a total of  $q$  entries, each of which is capable of holding one instruction (plus any flags that may be necessary). That is, each entry has  $32 + f$  bits, where  $f$  is the number of flags.

The job of the fetch unit is to keep fetching instructions sequentially until it encounters an conditional branch. When it does so, it must stop fetching until that branch is resolved: no speculative fetching or branch guessing takes place in this machine.

Part of the task of designing the fetch unit is managing the way in which the fetch and decode units communicate. Note that the fetch unit writes into the fetch buffer, while the decode unit reads from it. Obviously, the fetch unit must not overwrite an instruction in the fetch buffer before the decode unit has read it. Similarly, the decode unit must not wrap around the fetch buffer and read the same instruction twice.

EX Stage: The EX stage takes  $m$  cycles to multiply and  $n$  cycles for all other arithmetic and logic operations, where  $m$  and  $n$  are inputs to the simulator.

*Simplifying Assumptions:* Do not worry about interrupts. This machine does not support out-of-order execution (that would require a reorder buffer and approaches to deal with data hazards). 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.

Also, the machine does not do speculative execution following conditional branches: it waits for such a branch to be resolved before executing the instructions that follow.

Assume that the program being executed is read by the simulator from a file. Your simulator should include code which parses the program. To make this code simpler, you can redefine the assembly code to do away with commas and assume that all registers are specified by register number, not by register name. Thus, you can assume that instead of saying `add $s0, $s1, $s2`, the program will say `add 16 17 18` (e.g., since `$s0` is register 16). Also, assume that the `lw` and `sw` instructions have zero offsets and have the format `lw i j` or `sw i j` where `j` is the number of the register which contains the memory address to be accessed.

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:

- Average number of fetch buffer entries used.
- Utilization of each pipeline stage.
- Total execution time (for batch mode operation).

### Lab Report

Your lab report should include the following items:

- A detailed description of your design. This should include a block diagram of the various components, showing the signals between them and all pipeline registers.

- The source code of your simulator, fully documented. You should identify each module.
- Performance and utilization results for a program involving the multiplication of two  $10 \times 10$  matrices, A and B, with the result placed in C. These matrices will be stored in data memory: their values are  $A[i][j] = B[i][j] = 2i + j$ . Assume that the address of the first items of matrices A, B, and C are already stored in registers \$1, \$2, and \$3 when execution starts. The six statistics of interest will be the utilization of the Decode, EX, MEM, and WB stages, the time taken to execute the entire program, and the average fraction of fetch buffer entries that are occupied. (By utilization we mean when a stage is doing something useful, not when it is idle or just holding an instruction that has been held up there due to a hazard).
  - Vary the buffer size,  $q$  from 1 to 40 in steps of 1 and plot the fetch buffer utilization (average number of buffer slots occupied divided by the buffer size) for the following parameter values:
    - \*  $m = 1, n = 1$ .
    - \*  $m = 5, n = 2$ .
    - \*  $m = 10, n = 6$ .
    - \*  $m = 15, n = 10$ .

Set  $c = 1$  in this case. You will have six plots, one for each of the statistics of interest mentioned above; the x axis will be  $q$ . Each plot will have four curves, corresponding to each of the  $m, n$  combinations listed above. Discuss your results.

- Now, set  $c = 10$  and repeat the above simulation and plot your results as before. Discuss your results.
- Now, extend your machine to execute a pseudo-program consisting solely of “quasi-instructions.” (Note: “Quasi-instructions” is NOT a standard term; it is used here just to allow you to carry out more tradeoff studies with the simulator without having to write more assembly language programs). Each quasi-instruction has OPcode `QUASI` and no arguments. It does not affect the contents of any registers or memory; all it does is to consume varying amounts of time in the pipeline. It takes the  $c$  cycles to fetch a quasi-instruction from the instruction memory and one cycle to decode it. It takes  $e$  cycles in the execution stage  $e$  is a random number, uniformly distributed among the integers in the interval  $[a, b]$ , where  $a$  and  $b$  are parameters that you can vary. These quasi-instructions do not access data memory. To find  $e$ , simply generate a random number,  $r$ , uniformly over the range  $[0, 1)$  (i.e., discard the  $r = 1$  case) and set  $e = a + \text{floor}[(b - a + 1)r]$ .

Random numbers can be approximated in a number of ways on a computer. I suggest using the `rand()` function that is available in the C standard library, `stdlib.h` (it is

not a very good generator, but it will do for our purposes). `rand()` returns pseudo-random integers in the range 0 to `RAND_MAX`, which is a constant defined within the standard library. To obtain a random number, `r`, in the range (0, 1), you should set

$$r = \frac{\text{rand}()}{\text{RAND\_MAX}}$$

(Remember to cast these quantities into `double` before dividing since they are integers).

We will now consider the impact of the fetch buffer in dealing with execution time variability. Set `c = 10` for all cases. Vary the buffer size, `q`, from 1 to 30 in steps of 1, with the following parameter values:

- `a=1, b=19`.
- `a=5, b=15`.
- `a=9, b=11`.

Note that the average execution time of each instruction remains 10; however, its variability is very different for the three cases.

Run a program consisting of 10,000 quasi-instructions.

The statistics of interest here are the utilization of the fetch buffer, the total execution time, and the utilizations of the decode and execute stages. You should generate one plot for each of these statistics; there will be three curves on each plot, corresponding to the three `[a,b]` ranges listed above.

Provide an intuitive explanation for your results.

For further pointers, see the lecture slides for this lab.

Also, visit the Spark website regularly for clarifications, updates, or other announcements.

The demo signup sheet can be accessed starting November 12 and is posted at

[www1.mysignup.com/cgi-bin/view.cgi?datafile=3531abc](http://www1.mysignup.com/cgi-bin/view.cgi?datafile=3531abc).