B649: Parallel Architectures and Programming
Assignment 2: Instruction-Level Parallelism

Announced: 2011-10-19 Due: **2011-11-11, 11:59 pm**

"What I hear, I forget; What I see, I remember; But what I do, I understand."
–Confucious

1 Motivation

This assignment will give you an opportunity to learn the details of, and compare, two important architectural techniques that we have seen in the course, pipelining and dynamic scheduling using Tomasulo's approach. You will write programs to simulate each of these architectures, and a test program in a simple RISC assembly language to compare the two.

This assignment involves non-trivial amount of programming. You are strongly advised to start as early as possible.

2 Deliverables

You have been given access to a mercurial repository called `A2` that is accessible only to you and the instructor, which you can access as follows:

```
$ hg clone ssh://username@sharks.cs.indiana.edu//u/achauhan/HG/Teaching/B649/2011-Fall/username/A2
```

Replace *`username`* by your own CS username. All your code, examples, and report should be committed within this repository. Make sure that you commit **only** the essential files. In other words, **do not commit** editor backup files, object (`.o`) files, executables, and any other temporary files. You are strongly encouraged to use `make` or similar utility to manage your code, testing, and report generation. You may use any language of your choice, but beware that realistically sized simulations are likely to take significant amount of time, which might cause certain language choices to result in extremely slow simulations.

The deliverables are:

1. A simple parser for the assembly language, called TRITE, described in Section 4.
2. A simulator for a simple pipelined architecture described in Section 5.
3. A simulator for an architecture based on Tomasulo's approach described in Section 6.
4. An example program for matrix-matrix multiplication written in the assembly language described in Section 4.
5. Simulation results as described in Section 7.

3 Interface

In order to make it possible to write automated scripts to run your code, you must strictly adhere to the interface described in this section. Departure from this interface will attract penalty.

1. You may develop on any environment, but your code must run on Linux.
2. The executable for the pipelined simulator must be called `sim_p` and that for Tomasulo's approach must be called `sim_t`.
3. Each executable must read assembly code in TRITE from `stdin` and output the simulation report on `stdout`.
4. Each executable must accept two file name arguments, the first file containing architectural parameters in the format specified in Sections 5 and 6, and the second containing a binary memory dump to initialize the simulated computer's memory.

4 Assembly Language

We will call our assembly language TRITE (TRivial Instruction-set for TEsting). It closely follows the MIPS instruction set used throughout the textbook. TRITE instructions are described below. A name starting with `t` denotes a result (target) register. A name starting with `s` denotes a source register. The suffix `.I` denotes an integer register and `.D` denotes a double-precision floating point register. The actual register names are `R0` through `R15` for the 16 integer registers and `F0` through `F16` for the 16 floating-point registers. `o` represents the immediate offset, which is assumed to be a signed 16-bit value. All instructions are 32-bit long. Integers **and addresses** are 32-bit long and double precision floating point numbers are 64-bit long.

`NOOP`		Do nothing
`ADD`	t.I, s_1.I, s_2.I	Integer add
`SUB`	t.I, s_1.I, s_2.I	Integer subtract
`MUL`	t.I, s_1.I, s_2.I	Integer multiply
`DIV`	t.I, s_1.I, s_2.I	Integer divide
`L`	t.I, o(s.I)	Integer load
`LI`	t.I, o	Integer load immediate
`S`	**s_1.I, o(s_2.I)**	Integer store
`BEQ`	s_1.I, s_2.I, o	Branch if equal
`BNEQ`	s_1.I, s_2.I, o	Branch if not equal
`BGTZ`	s.I, o	Branch if greater than zero
`BLTZ`	s.I, o	Branch if less than zero
`MOVE`	s.I, t.D	Move an integer register into a floating point register
`ADD.D`	**t.D, s_1.D, s_2.D**	Floating point add
`SUB.D`	**t.D, s_1.D, s_2.D**	Floating point subtract
`MUL.D`	**t.D, s_1.D, s_2.D**	Floating point multiply
`DIV.D`	**t.D, s_1.D, s_2.D**	Floating point divide
`L.D`	t.D, o(s.I)	Floating point load
`S.D`	s_1.D, o(s_2.I)	Floating point store
`BEQ.D`	s_1.D, s_2.D, o	Branch if equal
`BNEQ.D`	s_1.D, s_2.D, o	Branch if not equal
`BGTZ.D`	s.D, o	Branch if greater than zero
`BLTZ.D`	s.D, o	Branch if less than zero
`MOVE.D`	s.D, t.I	Move a floating point register into an integer register

We will assume that the register `R0` represents the integer 0. Thus, no other value can be loaded into `R0`. Notice a few simplifications:

- There are no logical operations.
- There are no unsigned versions of arithmetic operations.
- Only a few common types of branches are supported.
- There is no unconditional jump. It can be simulated using `BEQ R0,R0,o`, where `o` is the offset.
- Integer load immediate is a convenience instruction to load a register with a 16-bit signed immediate value.
- Exceptions are ignored.
- Memory subsystem is ignored. In other words, all memory access is assumed to hit in cache.

Assume a reasonable amount of data memory, but at least 1MB. You may increase the size if you want to simulate programs requiring large data memory. You do not need to simulate instruction memory.

Finally, the assembly code may contain optional labels in front of each instruction and an optional comment at the end of each line starting with a semi-colon (`;`). The symbolic labels may be used for the immediate values (`o` field) in the branch instructions. The immediate values in arithmetic instructions are specified as decimal numbers.

Following is an example of a simple loop to add two floating point arrays. The arrays are assumed to contain 100 elements each, and start at memory locations 0 and 800, respectively. The result is written in an array starting at location 1600.

```
      ADD    R1,R0,R0     ; initialize R1 to zero
      LI     R2,8         ; save the increment value into R2
      LI     R3,800       ; save the index at which the loop ends
loop: L.D    F0,0(R1)     ; load the first operand into F0
      L.D    F1,800(R1)   ; load the second operand into F1
      ADD.D  F2,F0,F1     ; F2 = F0 + F1
      S.D    F2,1600(R1)  ; store the result
      ADD    R1,R1,R2     ; R1 = R1 + 8
      BNEQ   R1,R3,loop   ; termination condition
                          ; use label as the offset value
```

Task 1 Your first task is to write a simple interpreter for the above assembly language. Notice that you don't have to "assemble" the input into a byte-code. You can simply interpret the assembly code in text form. You will need to keep track of the labels to be able to jump to them.

Hints:

- You may consider using a hash table to maintain the instruction number corresponding to each label.
- It might be useful to write a simple routine to dump sections of the memory that you simulate to make sure that your simulation is correct.

5 Simple Pipeline

The second step is to enhance your simulator to simulate a simple 5-stage pipelined architecture. We will assume no forwarding and no delay slots. Your simulator, called `sim_p` will be invoked on the command line as follows:

```
$ sim_p arch_params_file init_mem_file < assembly_file > output_file
```

The `arch_params_file` consists of rows, each row containing the instruction name and the **latency of its execution stage**[1] in cycles, separated by one or more spaces. For example the file might look something like this:

```
NOOP  1
ADD   1
ADD.D 5
DIV   20
DIV.D 20
...
```

You may assume that latencies will always be positive integer values.

The `init_mem_file` contains a binary dump of the initial portion of memory. This is to be able to simulate a realistic computation without having to initialize the input data using assembly code. Thus, you might write a separate program to create the dump containing input values and then initialize your simulated computer's memory using it.

Task 2 (35 points) Your second task is to write `sim_p`. Its output consists of the number of cycles your program took to complete, the number of stall cycles it encountered, and the average CPI.

Hint: You will need to finish Task 1 before you can undertake Task 2. However, Task 2 should be a relatively small increment over Task 1.

[1]In a simple pipeline, every other stage has a latency of one cycle.

6 Tomasulo's Dynamic Scheduler

Next, you will undertake the simulation of a more advanced architecture, based on Tomasulo's approach. As in the textbook, we will assume unpipelined functional units for the sake of simplicity. Your simulator, called `sim_t` will be invoked as follows:

```
$ sim_t arch_params_file init_mem_file < assembly_file > output_file
```

This time the parameter file will contain additional information about the number of reservation stations. At the end of latencies for each instruction additional parameters will specify the sizes of load buffers, store buffers, and the reservation stations for each functional unit.

```
NOOP  1
ADD   1
ADD.D 5
DIV   20
DIV.D 20
...
loads   15
stores  15
intadds 10
intmuls  5
intdivs  3
fpadds   8
fpmuls   5
fpdivs   2
```

Notice that the add functional units are used for both adds and subtracts. Assume that integer moves and integer branches go into integer add reservation stations. Similarly, floating point moves and branches go into floating point add reservation stations.

Task 3 (35 points) Your third task is to write `sim_t`. Its output consists of the number of cycles your program took to complete, the number of stall cycles it encountered, and the average CPI.

Hint: This task may require significantly more effort than Task 2. However, you may be able to reuse code from Task 2. For example, you can write a single function / class for reading the parameter file. The pipelined architecture will simply ignore the additional parameters related to Tomasulo's approach.

7 Simulations

You final task is to simulate a double precision matrix-matrix computation by varying different architectural parameters. For example, by varying the number of reservation stations for different types of functional units you can determine the limits on the performance improvement that Tomasulo's approach gives you. Similarly, by varying the matrix sizes you can determine the relative performance difference of short versus longer-running programs. You may design more experiments.

Task 4 (30 points) Compare the performance of architectures based on simple pipelining and Tomasulo's approach by simulating double-precision matrix-matrix multiplication. Present the results in graphical form in a short report. You may also perform loop optimizations to assess their benefits. In your report, clearly indicate any assumptions your simulators make that have not been specified here.