Processor Assignment

David May, Simon McIntosh-Smith (with thanks to Dan Page)

January 23, 2017

1 Introduction

This assignment will introduce you to how superscalar processors really work. The idea is to investigate such processors by:

1. developing a program to simulate processor behaviour, and
2. performing some experiments which highlight the behaviour of various architectural features when fed suitable example programs.

The ideal simulator should eventually model a processor that includes

- a register file,
- several types of pipelined execution unit (i.e., an integer arithmetic unit, a load-store unit and a branch unit),
- an instruction decode/issue unit employing issue-bound operand fetch,
- a re-order buffer handling consistency and re-naming,
- a means of branch prediction.

Writing this simulator is essentially a "mini" project which will allow you to get a deeper understanding of first half of the unit. Other important things you need to know:

1. There is no prescribed language for implementation; some students prefer to use the language they are most familiar with (e.g., C or Java), other students prefer a language which offers features that are specifically useful (e.g., occam). The one important caveat to this is that assessment will take place using a computer from the CS lab; your code needs to compile cleanly using only standard tools (e.g., GCC) available on such a computer.
2. The marking scheme is biased toward learning about concepts rather than a programming exercise: although the implementation is important, you should treat it as a tool to support the *real* goal of investigating superscalar processors.

3. Many of the stages are "open": there could be many possible aspects of a given problem to investigate, and many valid ways to construct solutions. In these cases, the marking scheme is biased toward *you* being inventive: where there is some debate about the right way to proceed, the assignment demands you make an informed design decision *yourself* and back it up with a reasonable argument.

2 Submission and Marking

The assignment is presented in a number of stages which are roughly arranged in order of difficulty. You shouldn't necessarily expect to complete all stages. Remember that on one hand you may be able to complete some of the later stages without completing the earlier ones, but on the other hand some earlier stages are intentionally included as "stepping stones" to guide you toward a final solution. Some stages may be denoted "extra" and should only be attempted when you have completed all others: they are intended for those of you who are aiming for the top marks bracket or want to attempt more advanced work.

You will submit your coursework in two parts, an interim submission half way through the course, and a final submission at the end of the course. The interim submission will not itself be marked, but we will give you feedback based on this submission that should enable you to achieve a higher mark compared to just submitting everything at the end of the course.

Marking of the final submission will be predominantly via a single 15 minute presentation at the end of the course. This should be notionally divided into 10 minutes of actual presentation and 5 minutes to demonstrate your simulator and answer questions. This has some nice advantages:

1. The assignment is a challenge to complete and hence a challenge for us to understand and mark. By using a more interactive form of marking we can better understand exactly what you have achieved and give you more direct and personalised feedback.

2. Being able to demonstrate your simulator (rather than simply submit it) means you can be sure the quality of your implementation work is taken into account.

Although the assignment content means an exact marking scheme is difficult, we expect to allocate marks from three rough categories:

1/3 **of the marks** for the features in your simulator and their complexity (i.e., which features of the processor you model).

1/3 **of the marks** for the quality and reasoning associated with results obtained from experiments undertaken with your simulator.

1/3 **of the marks** for the presentation quality (e.g., clarity of explanation, answers to questions).

The time and date of presentations will be arranged after the submission deadline; we expect this to be *after* the Christmas break (i.e., in the assessment period in January) so we will have to manage potential conflicts with the exam period. You should submit your work using the online submission system at

https://wwwa.fen.bris.ac.uk/COMSM0109/

For *both* the interim and final submissions you should submit:

1. All source code for your simulator and any auxiliary files which you think are important (e.g., example inputs and outputs). Note that:
   - You should ideally submit unarchived source code; if this is not possible, submit a single ZIP archive called `source.zip`.
   - The source code should compile cleanly and be tested using the default set of software tools (e.g., GCC) available in the CS lab.
   - You should include a single file called `readme.txt` (and a `Makefile` or equivalent) that contains detailed instructions for compilation and execution of the simulator.
2. A set of PDF slides (i.e., no PowerPoint or OpenOffice documents) called `slides.pdf` which describe what you have done. For the final submission these will also form the basis for your presentation. For the final presentation keep in mind that the it should last 10 minutes. Your presentation's organisation should roughly follow:

   1 **slide** A diagram of the processor architecture your simulator models, including major components and relationships between them.

   1 **slide** A list of architectural and micro-architectural features that the processor models (e.g., number of execution units, method for branch prediction, support for register renaming); this should roughly describe how far through the assignment you have progressed.

   3 **slides** To describe the experiments you undertook using the simulator, why you did them and what the outcomes were.

3 Description

For the *interim* submission you should complete up to Stage 3, Part 1, as listed below. For the *final* submission you should then complete as much as you can up to the end of Stage 5.

This means that for your interim submission you should have a working, simple, non-scalar, non-pipelined simulator, with a simple instruction set, and a simple set of benchmarks with which to test your simulator.

Your interim submission should then be extended to form your final submission, making the processor more sophisticated by adding superscalar execution,

branch prediction, out of order execution and so on. Your set of benchmarks should also be more comprehensive, as should the set of experiments you have run through your simulator.

Stage 1

The aim of this stage is to design an instruction set for your processor. On one hand, the less instructions you include the simpler your implementation is; on the other hand you need enough instructions to write interesting programs. It is crucial to be realistic about the operations and addressing modes you use. One way to ensure this is to base your choices on a much reduced subset of a RISC instruction set such as MIPS; in such an instruction set one would expect to find:

1. Instructions that perform arithmetic, comparisons and logic; for example you may opt to include:
   - `add` to perform addition of two register operands,
   - `addi` to perform addition of a register operand and an immediate operand,
   - `cmp` to perform comparison of two register operands with each other (with the result being -1, 0 or $+1$ for less-than, equal-to or greater-than).
2. Instructions that (roughly) perform memory access; for example you may opt to include:
   - `ld` to perform an indexed load from memory using a register operand as the base address and an immediate operand as the offset,
   - `ldc` to load an immediate operand.
3. Instructions that alter the control-flow, i.e., branches and jumps; for example you may opt to include:
   - `b` to perform an absolute branch to an address provided as an immediate operand,
   - `j` to perform a relative branch to an address provided as an immediate operand,
   - `blth` to perform an absolute branch to an address provided as an immediate operand if one register operand is less-than another.

Note that if you have previously taken the COMS30201 unit it will be tempting to use this as a starting point: beware that the COMS30201 instruction set contains a large number of features which are extraneous to this exercise, and that your simulator from said unit will differ enough that starting from scratch (or at least reusing only selecting components) may be a better option.

Stage 2

Your next step should be to write several short benchmark programs using the instruction set defined previously. Rather than a single "golden" benchmark that exercises every aspect of the processor perfectly, a better approach would be to use a suite of kernels that are representative of real-life workloads. A basic example would be a loop to perform vector addition:

```
for( int i = 0; i < 10; i++ ) {
  A[ i ] = B[ i ] + C[ i ];
}
```

As such, one can check that various arithmetic, memory access and control-flow instructions work individually *and* in composition, and also evaluate performance based on how quickly the kernel is executed. Further kernels can be constructed by targeting specific features in the processor, or by using using existing examples such as

- standard kernels (e.g., the so-called "Livermore Loops" [1]),
- sorting algorithms (e.g., bubble sort or quick sort),
- numerical algorithms (e.g., GCD or Hamming weight),
- recursive algorithms (e.g., factorial).

Note that although there are no associated marks, it could be advantageous to write a simple assembler to translate your benchmark programs from a human writable assembly language into a machine readable (i.e., encoded) form. Since instruction encoding and decoding mechanisms are not the main thrust of this assignment however, one might proceed in other ways. For example, it is acceptable to represent the programs as an array of structures where each structure instance represents an instruction containing explicit opcode and operand fields. Also note that benchmark kernels should not be confused with test kernels. The former are used to demonstrate that certain performance-enhancing features of your processor's architecture are working – for example you can demonstrate superscalar instruction issue or improved branching performance with the addition of a branch prediction mechanism. Test kernels are for your own use and will just help you get your simulator working. Of course these are a good idea but they should not be presented as part of your report.

Stage 3

Once you have an instruction set and some programs in a machine readable form, you can start to construct a simple processor simulator to execute them. The best way to proceed is to start with a simple simulator and extend it with more and more features, testing that it executes your benchmark programs correctly after each step:

1. Write a simulator for a non-pipelined, scalar processor which executes instructions in-order; this is essentially a loop which iterates through fetch-decode-execute steps using a program counter, and using a register file and (single) ALU as appropriate. You can assume the processor uses a "Harvard" memory hierarchy whereby data and instructions are accessed via separate, simple interfaces to memory: you do not need to consider cache memories.

2. Pipeline your initial simulator so that the fetch-decode-execute steps can be carried out in parallel for different in-flight instructions. This demands that you model a clock to advance the pipeline, and the pipeline registers that hold partially executed instructions. At this stage you can ignore the problem of dependencies and hazards (i.e., assume the programmer avoids them).

3. Duplicate the execution stage so that your simulation now has several execution units (e.g., two integer ALUs, a load-store unit and and branch unit). You do not need to model the internals of the execution units (e.g., model the actual circuit for addition or multiplication), and can ignore the reservation stations at this stage and use blocking issue. Note that although a valid first step would be to assume each execution unit has some fixed latency, a more reasonable approach would be to use sub-pipelining.

Some students prefer to take an "object oriented" approach to their simulator by defining classes to represent the various components (e.g., register file, reservation stations and so on) in the processor. This is not a requirement and certainly not the only option, but can make the resulting program more modular and easier to understand and develop. For each component in the processor state it is useful to define

- a "current" value (at the start of each simulation cycle),
- a "next" value (by the end of each simulation cycle).

The behaviour of each component can then be modelled by a function which, when called, derives the "next" value from the "current" value. The simulation advances one cycle at a time, by calling all of the functions which model the component behaviours. When they have all been called, it copies the "next" values into the "current" values and is then ready to repeat the process by performing the next simulation cycle.

It may be helpful to provide a method to single-step the simulation in an interactively manner, displaying all of the state on the screen at each step. This is similar to how a debugger works on a real-life processor, and enables you to inspect and debug the simulator based on program and processor behaviour.

You should make sure that your simulator outputs useful metrics, such as:

1. the number of instructions executed,

2. the number of cycles taken for the whole run,
3. the number of instructions per cycle (to a sensible number of decimal places),
4. anything else useful/interesting relevant, such as correct branch prediction rate etc.

Stage 4

At this stage you should have a simulator that can execute "valid" programs (in the sense that they respect any potential hazards) and takes advantage of multiple execution units. However, many features of a real superscalar processor which permit high-performance are still missing. The goal of the next stage is to implement such features. This is a deliberately open question as there are many potential features you could include. As examples you might opt to:

1. add a mechanism for branch prediction or predicated execution,
2. add reservation stations and non-blocking issue,
3. add a re-order buffer,
4. add register re-naming (using the re-order buffer),
5. include an execution unit that supports vector instructions.

Stage 5

The final stage is to conduct some well reasoned experiments with the simulator and benchmark programs you have developed. The aim is to construct a series of "hypothesis, experiment, result" type statements which exercise your simulator in different ways and empirically demonstrate the benefit of specific features. Some of the marks associated with this stage are related to the formulation, number and sophistication of your experiments; a contrived example of what might be expected is:

Hypothesis The branch prediction method X as implemented in my simulator is effective enough to deliver performance for a loop which is within $A\%$ of the unrolled loop.

Experiment I wrote a program B to perform an iterated vector addition, and another program C which unrolled the loop for a value of $n = 100$. I ran both programs on my simulator and counted the number of cycles taken to execute them; this was averaged over m executions to take into account the non-deterministic nature of the branch prediction method.

Result The average number of branch mispredictions for B was E which represents a misprediction ratio of F; this resulted in an average cycle count of G and H with and without branch prediction and I for the unrolled loop C. Within these examples I recorded completion rates of J and K instructions per cycle for B and L instructions per cycle for C which demonstrates that better use of the execution units is achieved. Hence we can conclude that the branch predictor is effective: G is better than H and within $A\%$ of I.

References

[1] F.H. McMahon. The Livermore Fortran Kernels: A Computer Test Of The Numerical Performance Range. Technical Report UCRL-53745, Lawrence Livermore National Laboratory, 1986.