Computer Architecture Project

Yuan Xue@PPI

Objectives of the projects

- To help you develop the ability to apply knowledge in computer architecture learned in the course.
- To help you develop the ability to use the techniques, skills and modern engineering tools needed for scientific engineering practice.
- To help you obtain the knowledge of related topics in computer science discipline.

The objective of the course projects is to give you an opportunity to learn how to design and implement real computer architecture. Through these projects, we hope you will get an idea of what simulation is and what it could do. And moreover, we hope to consolidate and refine your understanding of pipeline design of the processor, cache design issues, parallel programming, and etc.
There will be 3 projects in this course:

- In the first project, you will learn how to simulate instruction execution and detailed pipeline of the processor.
- In the second project, you will learn parallel programming skills to accelerate simulator performance, which is one of the hottest researches in computer architecture simulation.
- In the third project, you will implement a trace-driven multi-core cache simulator and use it to do some interesting case studies.

It will "simulate" a real engineering project in open source community. You should cooperate to download documents and source code from the web-site, read those documents and hack the source code of the software, make clear the goal of the project, write down your design document clearly, and implement your design efficiently.
Now let's go to project 1, it has two parts. In **Part 1**, you will learn how to simulate instruction execution. And in **Part 2**, you will simulate how a 5-stage pipeline works.

Handout Instructions

checkout the lab from svn server, named as ArchitectureLab, give the command:

svn --username=StuID checkout svn://10.176.34.18/xueyuan-repo/StuID

Start by copying *architecturelab-handout.tar* to a (protected) directory on a Linux machine in which you plan to do your work. Then give the command

```
unix> tar xvf  architecturelab-handout.tar
```

This will cause a number of files to be unpacked in the directory. The handin files are listed in the following part separately.

Project 1, Part 1

1. Your Mission in Part

Before starting this part, you should first get familiar with the SimpleScalar toolset, which is one of the most popular simulators used in academic and research. You should be familiar to use sslittle-

nasstrix-gcc, the GCC cross compiler provided by SimpleScalar, to build MIPS-like binary code for running on the SimpleScalar architecture, and etc. For more detail, please refer to the document "***SimpleScalarSetup.pdf***".

Still remember the exciting moments in your first lab of ICS course? We have tried hard to reduce the number of operations for some specific arithmetic computations. We are glad to announce that only ONE operation is needed now since some of them have been implemented at hardware level. ;-) Now it's time for you to add two of such instructions into our SimpleScalar simulator.

The two instructions to be implemented are as following:

addOk

"addOk" checks whether the sum of two operands is overflow, returns 0 when overflows. The format of the addOk instruction just follows the way of "add" instruction.

For example，"addOK $s1, $s2, $s3" means setting "$s1" with value "0" if expression "$s2+$s3" overflows, otherwise with value "1".

bitCount

"bitCount" counts the number of "0"s or "1"s in a 32-bit integer. The format of bitCount instruction is the same as "xori". You may refer to the manual for the format or just simply copy the format. For example，bitCount(0x5,1) counts the number of "1" in integer 0x5, and returns 2. And bitCount(0x5,0) counts the number of "0" in integer 0x5, and returns 30 (not 1!).

We have provided you with two test cases. Each of them contains a call for one of newly added instructions. Since our simulator is 'out-of-date', it will complain and lead to a panic when it meets unrecognized instructions. sim: ** starting *fast* functional simulation **

panic: attempted to execute a bogus opcode [sim main:sim-fast.c, line 444]

Your mission is to study on the implementation of the simulator in order to make it work again.

Please follow the steps described below to setup your working environment and begin your work:

1. Build and install SimpleScalar tool chains (gcc, binutil), see the simpleScalarSetup.
2. Run the test cases offered with this document by sim-fast.
3. Find the reason why the test cases fail, we suggest that you use some debug tools, such as gdb and the object dump tool supplied by SimpleScalar.
4. Indentify the OP-Code number of two instructions.
5. Try to implement the new instructions in SimpleScalar.

2. Suggestions and Hints

1. The core of the sim-fast simulator resides in two files: *sim-fast.c* and *machine.def*. There is a "while (TRUE)" loop in *sim-fast.c* which you should pay much attention. There is a detailed specification in *machine.def*. I would suggest you to read it carefully before you begin your work.
2. Every time when you modified the SimpleScalar source code and want to rebuild a new version, please follow the steps below:

make clean make

config-pisa

make sim-fast

Remember, you should "make clean" to clear your directory. Otherwise, some old code will break your program.

3. Your mission is to get familiar with SimpleScalar in order to prepare for further projects. To pass the test cases provided is the only requirement. You do NOT need to worry about other flags such as overflow or alignment. Also, there's no restriction as in your ICS lab. For example, feel free to count the number of 1s with a loop statement for bitCount.
4. **No cheating, otherwise, both will fail the project. This rule will always work in this course.**

3. Handins

1. **machine.def** Where you implement your instructions.
2. **sim-fast.c** and other related source code.
3. P**art1-design.pdf.** A detailed document that describes your debug process and your solution.

Project 1, Part 2

1. Your Mission in Part

The mission in Part 2 for you is to extend the shabby sim-fast simulator to a 5-stage pipelined sim-pipe simulator that can do operand forwarding or bypassing. Unlike Part 1, in which you only need to read a couple of C files and write no more than 10 lines of code, in Part 2, you need to hack the internals of the sim-fast source code and try to understand the implementation details of the simfast simulator. Meanwhile, you should master the principles of pipeline theory in computer architecture before you can begin your design. The work in this project needs more creative ideas than the last one. In this document, you will be given some suggestions on how to start your work and how to check what you have done. But they are only suggestions. You are encouraged to try some different and creative ideas to do the work.

2. Hints

2.1. Design hints for classical 5-stage pipeline

Before you start to write code in file: sim-fast.c, you'd better have a good understanding of the pipeline theory. If you can go through the diagrams from Figure 6.10 to Figure 6.36 in Patterson & Hennessy's book “A Computer Organization & Design" like you are reading a comic book, you know you are ready to think about the implementation now. In the release package, a good lecture note (***pipeline lecture.ppt***) on the pipeline basics is also included to help you make better preparation. Still, don't touch the C source code in a hurry. You'd better convert your understanding of the pipeline implementation into some kind of software design representation first. The block design diagram is always the best way to describe and clarify your ideas.

We have five pipeline stages:
IF: fetch instruction from memory based on PC+4 for non-branch instruction, or based on the PC provided by branch resolution for branch instruction.
ID: decode instruct, detect data /control hazard, read register for operands.
EX: execute computation operations (ALU/others), calculate effective memory address for load/store instructions, resolve branches.
MEM: load/store data from/to memory address calculated in EX stage.
WB: write data back to register file and finish the instruction.

The run time information of an instruction that is executed in the pipeline is actually stored in one of the four pipeline buffers: IF/ID, ID/EX, EX/MEM, MEM/WB.

Please write down clearly in your design block diagram that what kind of data needs to be passed between those pipeline stages, what kind of information needs to be stored in those pipeline buffers, what data structure is used to represent those data and what information is needed to generate new data for the next stage.

2.2. Coding hints

When people start to write code for some project, they always try to use the existing code as much as possible to save time. You are also encouraged to do so in this project. In fact, you don't need to be concerned with manipulating the instructions and data of the program being executed. The original sim-fast source has already provided a lot of utility functions or macros to help you. Such as fetching instruction (MD_FETCH_INSTI), instruction decoding (MD_SET_OPCODE), accessing the register file (GPR, SET_GPR), doing arithmetic, and accessing the memory (READ_WORD). However, once you located the main loop in sim-fast.c, you will find that these operations are done in one shot. I didn't say "in one cycle" because there isn't any concept of cycle in the original sim-fast code at all. So you have to add a cycle counter and increase by each iteration of the main loop. Now the question is what should be done in the main loop to make the 5-stage pipeline work. If you finished the design of the data structures for those pipeline buffers, the only thing you need to do is to update each pipeline buffer according to the values in the pipeline buffer preceding it. For example, values in EX/MEM should be updated according to the values in the ID/EX one cycle later. Intuitively, you need five functions. Four of them update IF/ID, ID/EX, EX/MEM, MEM/WB pipeline buffers and the last one does the write back operation with the value stored in MEM/WB. Moreover, we have provided you with a base framework in sim-pipe.c which has divided the main loop into several stages. You are encouraged to implement the project creatively, so feel free to ignore sim-pipe.c and even build all your work from scratch.

Whenever the opcode of the instruction is used, you must use the machine.def (pisa.def is a complete version) file in the way used in the original code. There is plenty of information about all instructions described in that file. The only issue is that, the implementation of the instruction, OP_IMPL definition for example, has to be modified, since we are using pipeline buffers not registers. It would be a huge amount of work to modify all the instruction of PISA architecture, but for your project, you only need to modify those used in the test program.

Remember, your program is a sequential program, so the order you run the five functions that update the pipeline buffers is very important. Please think about it carefully and give your answer in your design.

2.3. Test framework

We all have the experience to use C compiler to drive assembler and linker to generate binaries for SimpleScalar simulator. But that's not suitable in this project because the linker will link many library codes into our program which we don't want. So we need to write our own assembly code and use the assembler and linker directly by ourselves. Well, I have already prepared a piece of code which has a loop in it. Following the instructions below you can run this piece of code.

1. Update the loader.c in the target-pisa directory with the one we provided. You can look at the difference between them and think about the reason of the changes if you are interested. Then rebuild your simulator.
2. Download the test program loop.s.
3. You may implement your work based on sim-pipe.c. If so, download sim-pipe.[c/h] and Makefile as well. Then rebuild your simulator. 4. Assemble, link and run the test code:
   $SIMPLESCALAR/bin/sslittle-na-sstrix-as -o loop.o loop.s

$SIMPLESCALAR/bin/sslittle-na-sstrix-ld -o loop loop.o
$SIMPLESCALAR/simplesim-3.0/sim-pipe loop
($SIMPLESCALAR is the path of your actual SimpleScalar directory)

You may need to read the loop.s file to understand what the program is doing. After the program finishes, there will be a magic number stored in general purpose register 6. This magic number will be used to check the correctness of the simulator. You can imagine, even if your pipeline works fine, you probably still won't get the right result. This is because the data and control dependence between those instructions, and your simulator may still not be able to handle them at this step. You have to think about where to insert "nop" instructions manually to help the simulator to solve these dependence problems.

2.4. Pipeline trace output

Your program, I mean the 5-stage pipelined sim-fast simulator you built, should be able to print out a pipeline trace for the program. What need to be shown in this trace file are the instructions that are executed in different pipeline stages in each cycle, as shown below:

IF	ID	EXE	MEM	WB
li $sp,0x7fff	Empty	Empty	Empty	Empty
move $fp,$sp	li $sp,0x7fff	Empty	Empty	Empty
sw $0,16($fp)	move $fp,$sp	li $sp,0x7fff	Empty	Empty
li $6,0	sw $0,16($fp)	move $fp,$sp	li $sp,0x7fff	Empty
lw $2,16($fp)	li $6,0	sw $0,16($fp)	move $fp,$sp	li $sp,0x7fff
slt $3,$2,10	lw $2,16($fp)	li $6,0	sw $0,16($fp)	move $fp,$sp
...	...	...	...	...

You can easily find the function that can disassemble instructions for you. So, use that function to print out the instructions. Also there may be "nop" instructions in the trace that are used to help the simulator to run the program correctly before it can perform operand forwarding.

2.5. Operand forwarding or bypassing

When you have successfully implemented the 5-stage pipeline, let's take a snapshot of the pipeline that executes the code below:

IF	ID	EXE	MEM	WB
move $4,$2	Empty	Empty	Empty	Empty
sll $5,$4,2	move $4,$2	Empty	Empty	Empty
subu $4,$5,13	sll $5,$4,2	move $4,$2	Empty	Empty
add $6,$6,$4	subu $4,$5,13	sll $5,$4,2	move $4,$2	Empty
lw $3,16($fp)	add $6,$6,$4	subu $4,$5,13	sll $5,$4,2	move $4,$2
addu $2,$3,1	lw $3,16($fp)	add $6,$6,$4	subu $4,$5,13	sll $5,$4,2
move $3,$2	addu $2,$3,1	lw $3,16($fp)	add $6,$6,$4	subu $4,$5,13
sw $3,16($fp)	move $3,$2	addu $2,$3,1	lw $3,16($fp)	add $6,$6,$4

It is clear that the result "$4" produced by instruction "move $4, $2" feed into instruction "sll $5, $4, 2". Therefore, these two instructions have RAW (Read After Write) dependence. However, the value produced by "move" is only available after it is written back to the register file at WB stage. So, "sll" cannot enter EXE stage before "move" finished WB stage. This will generate bubbles in pipeline, which brings performance penalty. And this is the exact reason why I mentioned several times in the

previous section that you may have to insert some "nop" instructions in loop.s to make it run correctly on the simulator. Operand forwarding or bypassing is one of the solutions that can fix this problem. See, after "move" finishes its execution in EXE stage, it can forward its result to the instruction "sll" that is still on ID stage. Therefore, we don't need to insert "nop", i.e. bubbles, in the pipeline. You must extend your design of the pipeline buffers to take care of operand forwarding. Actually, this is quite easy after you finished the 5-stage pipeline design.

This project also requires you to handle data hazard which needs detection and then insertion of a stall after the instruction and branch hazards. All these are not demonstrated above. Please assume the strategy of "branch not taken". As to at which stage to select the branch address, it depends on your choice. But basically, you should at least select the branch address at the end of the EX stage. But if you implement the selection at ID stage, it will be a plus as it is somewhat complicated! Another problem you should pay attention to is the implementation of unconditional jump which is not covered in the textbook. So you should think over it and have your own solution. And remember to show your decision about it in your project design document.

The final issue is about the "syscall" instruction which terminates the program. (You can find the how the loop program is terminated by debugging using sim-fast.) You should carefully implement this instruction as the program can exit only if this syscall instruction finishes its 5 pipeline stages. Also show your decision in the design document.

3. Handins

1. **sim-pipe.h** The header file including all the related data structure or function prototypes.
2. **sim-pipe-withstall.c** A simple pipeline simulator implementation, in which data hazards and branches are handled by stalling the entire pipe until the dependence disappears. Please dump your pipeline trace into trace-withstall.txt.
3. **sim-pipe.c** A relatively advanced pipeline simulator implementation, in which data forwarding and branch prediction (always not taken) should be implemented. Please dump your pipeline trace into trace.txt.
4. **Part2-design.pdf** Your design document, including a complete 5-stage pipeline design block diagram on functional level. Write down clearly in your design block diagram the data that need to be passed between pipeline stages, the information that need to be stored in these pipeline buffers, the data structures that are used to represent those data and the information that is needed to generate new data for the next pipeline stage. You should also briefly presents your ideas on detecting data hazards, forwarding (or bypassing) and resolving branch.
5. **trace-withstall.txt and trace.txt** Pipeline trace files respectively generated by your two simulators, which are consisting of the snapshot of each cycle, including the cycle number, the instructions corresponding to each stage, the register values (*r2,r3,r4,r5,r6*) related to our testcase as well as the content stored in a specified address(*16(r30)*).Here, you dump file is REQUIRED to be formatted as follow:

   [Cycle *CycleNumber*]----------
   [IF] *instruction in IF stage*
   [ID] *instruction in ID stage*
   [EX] *instruction in EX stage*
   [MEM] *instruction in MEM stage*
   [WB] *instruction in WB stage*
   [REGS] r[2]=*r2* r[3]=*r3* r[4]=*r4* r[5]=*r5* r[6]=*r6* mem=*memory value of 16(r30)*
   ----

6. **inst_list.pdf** The 5-stage pipelined implementation of the instructions in loop.s(I mean the machine instructions can be recognized and executed by the simulator).

4. Suggestions

- When doing the project, a good way is to start from the simple items. After getting enough experience, you can try to fix those difficult ones. In Part 2, you can first try to develop a 5-stage pipeline for only one instruction, e.g. "add" or "or". Then develop the pipeline for other instructions in loop.s.
- There are some pseudo instructions in loop.s. Pay attention to this: What you need to do is to develop pipeline for real machine instructions, i.e. instructions that can be recognized by sim-pipe simulator, not for those pseudo instructions.
- Try your best to do as much as possible. Don't be scared by the source code.