4190.308

Computer Architecture

Spring 2019

Processor Lab Report

04/17 ~ 05/13

Name:	Jinje Han
Student ID:	2018-18786

Introduction

Processor lab is a project that uses Y86-64 instruction set architecture. With Y86-64, students have to write assembly program, add new instructions in assembler, and modify simulator to execute new instructions. Doing this project, students can understand how the compiled codes are assembled and executed. Generally, it's hard to see through how instructions are executed on hardware level. On the contrary, the simulator codes are written in C code that implements the behaviour of hardware logic. Therefore, students can understand the stream of data and what's done on each stage of processor operation more effectively.

Important Concepts

The overall understandings about assembly language and processor operation was applied for implementation. Among them, there were some parts that required particularly deep comprehension. The first part was the grammar of Y86-64 ISA was used to writing assembly code. The information about the type and length of arguments that each instructions require was mainly used. Beside it, I had to take account of the fact that assembly codes start with loading Stack data and ends with halt instructions. Second part was the hardware structure of processing unit that process instructions. Understanding about the components of structure including register, hardware units, wires, etc. was required. In detail, how the components are connected with each other to implement certain functionality was mainly used. In addition, how the functionalities are divided into stages (from fetch to Memory) must be known. Beside these two parts, grammar and role of HCL, and pipelining method were also necessary for implementation.

Part A

Writing <sum.ys> was done by simply translating the corresponding C code into assembly language. One register stores the value of ls, and the programs loops until the value of register becomes zero.

Writing <rsum.ys> was trickier, since it has to call another function, therefore not being able to store the value of ls in one register. First of all, I processed the case when the value of argument is zero. In the non-zero case, I decided to save the value of local long 'val' into %r10. Next, I had to call ca function and get return value. However, if the recursive function is called, the value in %r10 will be destroyed. Therefore I pushed the register into stack and popped after the function returns. Finally, I added the value of (popped) %r10 to the returned value and return the value again.

The assembly code of <copy.ys> was the easiest, since the instructions of the C code matches with assembly instructions. I saved the local long 'val' in %r10 and made a loop that flows as same as the C code, using JLE instruction.

Part B

The work I had to do in Part B was implementing IRMOV instructions that had immediate value smaller than 8 bytes, and CALLO function that operates with 4-byte offset instead of 8-byte immediate address value. These instructions are not included in original Y86-64 ISA. Since the codes were already complete for its own and had a complex structure, I tried to write new codes as same with the existing codes as possible.

In <yas-grammar.lex>, there were instructions, registers, and characters that I can use on Y86-64 assembly program. The words that have to be used in writing a program are defined in this file. That is, as the filename says, it was the grammar of ISA. I found the list of instructions was at the top of the file, so I added IRMOV series and CALLO instruction. Since these instructions share grammar with others, there were nothing more to change.

<isa.h> had typedef declarations of several variable type that are used in overall code. In enum arg_t, I added O_ARG, which stands for offset variable in CALLO function. There was also enum for icode types, so I added the icode of new instructions as others. There were nothing much to change in this file, but later I had to open it several times to get information for the definition of variable types.

<isa.c> file was the main file that execute instructions. At the top of the code was an array of 'instr_t' type, which contains information about which instruction requires which type of operands (immediate value, register, no argument) and their position and length in an instruction line. Information of new instructions must have been added on the array. IRMOV instructions was similar to IRMOVQ, so writing information about them only varied on the integer values. On the other hand, CALLO instruction needed a new type of argument, offset, so I had to find how to write 'offset' in the correct form. I read the definition of 'instr_t' type in <isa.h> and found that I should use 'arg_t' type at the argument position. O_ARG was what I should write.

There was a function that obtains 8-byte word from a memory destination, named get_word_val(). The existing Y86-64 only needs these two function, but some instructions I have to write require obtaining 1, 2, 4-byte value. Therefore, having same format with 8-byte function and just changing the integer value in the function, I wrote extra functions that gets value with different size.

Next part to look at was the main part that actually executes an instruction. There was a section that checks if the instruction requires immediate value, and if yes, get the value with get_word_val(). In this section, I had to use the functions I wrote above, and I needed extra value to check the length of immediate value this instruction requires. I changed need_imm(boolean value which is true when the instruction requires 8-byte immediate value) into need_imm8 and added need_imm1, 2, 4. Using these values, I could get different length of immediate value in different size. I addition, I put the code that sign-extend the immediate value into 8 byte in this section. It's because the length to sign-extend depends on the size of the value.

Last part to change was updating the values, which was written is switch case. IRMOV functions seemed that they can share same code. However, CALLO was different. CALL uses valC intactly as next PC value, but CALLO should add valC to next PC value. Therefore, I made extra case for CALLO (which was actually not that different from CALL case).

<yas.c> had instruction codes that can be used in several functions in many other files (which I realized later). The only part I found I should change was dealing with O_ARG, which I wrote new in <isa.c>. The space for offset was already prepared on the function, so I put byte address the current instruction is at.

Part C

Code files in part C are implementing hardware logic of Y86-64. Having that fact in mind, I matched the code with pipeline as I added and modify the code.

Sim Simulator

Files in <seq> folder implements sequential logic without pipelining. <seq-std-hcl> define control signal, a description of which value should go into each wire (ex: icode, valC, dstM), hardware units (ex: aluA, mem_Addr), and other values. First of all, I added symbolic representation of new instructions. (The comment said not to change or delete it, but didn't say not to add.) No extra computation signals seemed to be needed, so I just changed the control signal definitions. Because I didn't know how the control signals will be used yet, I just added IRMOV functions next to IRMOVQ and CALLO next to call.

<ssim.c> is the main logic about how to use signals in <seq-std-hcl> execute instructions. All I should do was just recombining existing values or modifying length of some values to make new instructions. Therefore, I realized all that I should change was the code with executing one instruction, since other codes were for background logic or other performances.

There was a part that checks if the instruction requires valC and get that value with get_word_val(). Because this function only gets 8-byte value, there needed extra function that checks the length of immediate value this instruction requires. Therefore, I changed need_valC() (function that returns true if this instruction requires 8-byte immediate value) into need_valC8() and added need_valC4, 2, 1() at <seq-std.hcl>. Getting the value with different size could be done by functions I have added in <isa.c>. Similar to the code in <isa.c>, I also wrote sign-extending code here. Code for IRMOV was done for this.

Implementing CALLO here required different problem. To identify which address the calling function is, CALLO had to add the current pc value (which is in valP) and the offset value. However, the ALU was already used for calculating new stack pointer value. Therefore, I couldn't place the address-calculating code on execute part. So I distinguished need_Off4() from need_valC4() and put the code for address calculating there.

Pipe Simulator

Before modifying the files in <pipe>, I compared the code with files is <seq>. Codes in <pipe> seemed to be much longer, but I noticed that except functions and values for pipelining, the core part was almost same. For example, in both <ssim.c> and <psim.c>, get_word_val() that gets 8-byte immediate value had arguments that has same label. Therefore, what I wrote was almost same with that in <seq>. I wrote boolean values that checks if this instruction requires 8/4/2/1-byte immediate value or 4-byte offset value, and address calculating code for CALLO.

After modifying the code, I tried to find the reason that what I've done to files in both directories were almost same, even though one of them were pipelined. Matching the code with real hardware structure, I found that the changes in the code were only in the fetch stage. IRMOV series only differs in immediate value's size, which is processed in fetch stage. The address calculating is also calculated on fetch stage. That is, whether the process is pipelined or not didn't matter

Conclusion

Doing this lab, I could see how assembly codes are assembled into binary code and executed. Especially, I could learn how the simulator fetch and decode each instruction byte by byte. I didn’t have to look at all pipelining code very carefully, but I could know how much hardware designers endeavours to make processing faster. Ironically, the part that spend most time was not understanding the code itself. I had hard time finding the relationship between simulator C file and HCL file. I couldn’t find the main body of gen_func() that are used in simulator file, so I couldn’t know where to start. After typing <make> command on the commandline, I realized that HCL files are translated into C code, putting the prefix ‘gen_’ in front of each variables in HCL files.