Homework 1 Report

Lee, Ching-Jui, 0616329

***Abstract*—This electronic document is the homework 1 report for the course—Microprocessor Systems: Principles and Implementation at NCTU in 2020 fall.**

I. Introduction

The goal of this assignment is to use full-time system cycle-accurate simulation or ILA to observe the behavior of Aquila SOC and I choose ILA to analyze the signal to do this assignment. Through tracing the instruction execution cycle by cycle, it gives me a deeper understanding of the workflow of a processor execution.

II. ILA

ILA, which stands for Integrated Logic Analyzer, is customizable IP core is a logic analyzer that can be used to monitor the internal signals of a design. Through embedding an ILA core in RTL design, it can capture the waveform of signals. We can set signals as triggers to designate the condition when the ILA starting to capture the waveform.

III. PREPROCESSING

One of the problems I met is that when I check netlist to select the signal I wanted to debug, I couldn't find the signal name that declared in the Verilog code. I found that there were many unfamiliar net name in the netlist and many signal name didn't show in the netlist. I searched for the reasons, and understood that it is because of optimization by the synthesis tool. By default, Synthesis tool will do optimization of combinatorial logic. It will merge combinatorial functionality together, pull them apart, replicated them or do any variety of Boolean optimization on them. As a result, any net that is part of a combinatorial network may not exist. I followed the pdf tutorial, and one of the steps is to set code_addr_o as a trigger. However, I just couldn't find the signal in the netlist, even I searched the signal I couldn't find it. Knowing that it was eliminated through the process of optimization, I added the flag (*mark_debug = "true"*) in front of the place where code_addr_o was declared in the code and solved the problem. The signal, code_addr_o, was compulsorily preserved. Also, I found another solution to the problem, that is, set "flatten hierarchy" (in the synthesis option) to none. In this way, the synthesis tool will not flatten the hierarchy, and the output synthesis will have the same hierarchy as the original RTL, while the default setting, rebuilt, allows synthesis tool to flatten the hierarchy, synthesize, and rebuild the hierarchy based on the original RTL. It will do some optimization and the final hierarchy will be similar to the original one but not the same, so some signal will be eliminated. All the following of my works are done under the setting of none flatten hierarchy.

IV. STRING FUNCTION MODIFICATION

A. *strcpy*

In C, the reason for string operation takes more time than others and have Dhrystone performance decrease is that a string is an array of characters with null as terminal. As a result, compiler doesn't know how long a string is at compilation time. Some functions, such as "strcpy" and "strcmp", which need to know the end of a string to operate has to check each character if it is a null character to identify the end of a string, which is a time-consuming step. I found there are several methods to deal with or ameliorate the problem on Internet[1], the first method is to write the string functions in assembly language code. ANSI C also explicitly allows the strings functions to be implemented as macros. Another method is to set the source of the assignment a string constant but this is not a general purpose. I found another approach, I rewrite the function and have the assembly code change slightly. I rewrite the "strcpy" function as follows:

```
char *strcpy(char *dst, char *src)
{
   if(dst == NULL)
      return NULL;
   char *ptr = dst;
   while(*src != '\0'){
      *dst = *src;
      dst++;
      src++;
   }
   *dst = '\0';
   return ptr;
}
```

Although the logic are the same as the original code, I find that in the assembly code, they are different in order. The instructions of the loop body is shown below:

instruction address	**machine code**	**instruction**
0x1e94	0x00170713	addi a4, a4, 1
0x1e98	0x00158593	addi a1, a1, 1
0x1e9c	0xfef70fa3	sb a5, -1 (a4)
0x1ea0	0x0005c783	lbu a5, 0 (a1)
0x1ea4	0xfe0798e3	bnez a5, 1e94

I think that the key is the place of the instruction "**lbu a5,0(a1)**" and "**bnez a5,1e94**". In the original code, the first instruction is followed by the second instruction and the destination of the first instruction, a5, is the source of the second instruction. As a result, processor has to stall for a longer time to wait for the result of the first instruction being figured out, and then do the second instruction. However, in my code, there are no problem like this, so the processor stalling time is much shorter. The instructions of the loop body of my code is shown below:

instruction address	**machine code**	**instruction**
0x1e98	0x00f70023	sb a5, 0 (a4)
0x1e9c	0x00158593	addi a1, a1, 1
0x1ea0	0x0005c783	lbu a5, 0 (a1)
0x1ea4	0x00170713	addi a4, a4, 1
0x1ea8	0xfe0798e3	bnez a5, 1e98

From the signal point of view, the original code stall at the instruction "**bnez a5, 1e94**" for 4 clock cycles, and a round of loop takes 8 clock cycles to be done. However, my version stall at the instruction "**lbu a5, 0(a1)**" and "**bnez a5, 1e98**" and each for just 2 cycles. It takes only 7 clock cycles to run a round of loop. I think this is the reason that the Dhrystone increase by 0.02.

B. *strcmp*

For "strcmp" function, I do the following revision, which makes the code shorter.:

```
int  strcmp(char *s1, char *s2)
{
      int t1, t2;
   do {
      t1 = *s1++;
      t2 = *s2++;
      if( t1 == 0 )
         break;
   } while(t1 == t2);
   return (t1 < t2) ? -1 : (t1 > t2);
}
```

The original code and my code have the similar loop bodies with some instructions be placed in different order like the changes I did in "strcpy" function. In the original code, the processor stalls for 3 CPU cycles when execute the branch instruction "**beq a5,a4,1fac**", which is the instruction decide whether to loop again. The instruction takes 4 clock cycles to execute and it takes 9 clock cycles to run a round of the loop. The instructions of the loop body are shown below:

instruction address	**machine code**	**instruction**	
0x1fac	0x02078663	beqz a5, 1fd8	
0x1fb0	0x00150513	addi a0, a0, 1	
0x1fb4	0x00158593	addi a1, a1, 1	
0x1fb8	0x00054783	lbu a5, 0(a0)	
0x1fbc	0x0005c703	lbu a4, 0(a1)	
0x1fc0	0xfee786e3	beq a5, a4, 1fac	

However, it takes only eight clock cycles to have a round of loop done in my version. The processor stall at instruction "**lbu a4, -1(a1)**". I think the reason is that its register source is the register destination of the last instruction, so it stalls for one cycle to wait for the result of the last instruction to be figured out. The branch instruction "**beqz a5, 1fb8**" cause processor to stall for one cycle as well. Besides, the assembly code of my "strcmp" function is much shorter than the original one. Also the assembly code of my version (10 lines) is also shorter than the original one (16 lines) and it takes only 170 cycles to complete the function for my code while it takes 194 cycles in the original code. As a result, the DMIPS/Mhz improve by 0.02.

instruction address	**machine code**	**instruction**	
0x1fa0	0x00150513	addi a0, a0, 1	
0x1fa4	0xfff54783	lbu a5, -1(a0)	
0x1fa8	0x00158593	addi a1, a1, 1	
0x1fac	0xfff5c703	lbu a4, -1(a1)	
0x1fb0	0x00078463	beqz a5, 1fb8	
0x1fb4	0xfee786e3	beq a5, a4, 1fa0	

If both "strcpy" and "strcmp" functions are modified, the DMMIPS/MHz can be increased to 0.76.

V. *OTHER FINDINGS*

I observe the whole process of a result from figured out by ALU to be written into a register, since in the beginning, I wanted to use the register value to check each instruction's behavior, but soon I found that the register value didn't correspond to the instruction in the same clock time. To be more specific, a result from ALU to being stored in a register undergoes a series of steps and it takes a few cycles, so observing the signal, the register didn't match instructions right away in the same clock time. For example, looking at the register rf[11], which is register a1. The instruction "**addi a1, a1, 1**", the value of register rf[11] will not increase one right away. In fact, the result will be figured out in execute module and passed to forwarding module, writeback module and finally arrive the register, which is 3 clock cycles later from the instruction fetching. The figure 1 in the bottom of this report is the signal I observe.

VI. REVIEW

Among the two ways of analyzing the signal, I choose to use ILA. It is because I think that this tool is much more powerful and easier to use. Although I have taken Digital Circuit Lab, I use ILA tool first time. Its setting is much easier. Just pick out the signal I want to observe, and then the tool will automatically connect the signal to the ILA debug core and by setting a trigger, it is handy to set the timing when the core start to capture the signals. But it still have some disadvantage, every time I setup a new signal to ILA, I have to resynthesize the circuit again and generate bitstream. The whole process takes about 15 minutes every time. However, the method of simulation takes just a few seconds. There are other function of vivado that I haven't learned from DLab, for example, vivado provide schematic view of the whole system, so it gives me a much clearer view of the system. I can easily find out how each wire and module connect. I think that this is very cool since in DLab I haven't learned this function and I need to draw the diagram by myself, with the approach, it is easier to find out which wire or module are connected rightly or wrongly.

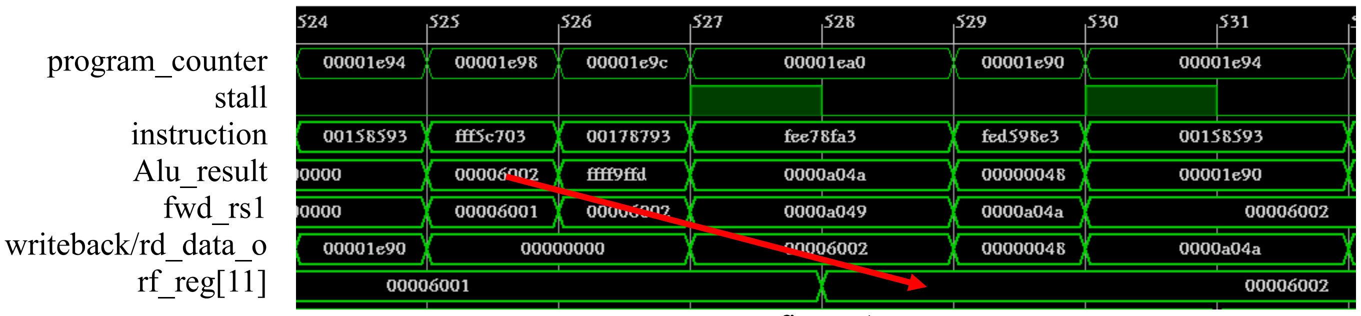

figure 1

VII. REFERENCE

[1] https://www.netlib.org/benchmark/dhry-c