LAB 9 – The Performance of MIPS

Goals

- Learn how to determine the performance of a processor.
- Improve the processor performance by adding new instructions.

To Do

- Determine the speed of the processor in Lab 8.
- Determine the bottleneck in the calculation.
- Enhance the processor by implementing additional standard MIPS instructions.
- Determine the new performance of the processor.
- Follow the instructions. Paragraphs that have a gray background like the current paragraph denote descriptions that require you to do something.
- To complete the lab, you must demonstrate your work to any of the tasks.
- You will have some questions to answer in the lab report.
- All optional tasks are highly recommended. You can ask assistants for feedback on optional tasks.

Introduction

In the recent labs we have built a small working microprocessor based on the 32-bit MIPS. The ALU was built during Lab 5 and allowed us to implement a subset of the original R-type instructions. In Lab 7, we used these instructions to write a small program that can add a range of consecutive integers. Finally, in Lab 8, we managed to put pieces of the MIPS together to get our first microprocessor. During this lab we will be interested in improving the performance of the processor.

Performance

Before we begin, let us examine the current processor and find out how it can be improved. Refer to Lab 5 if you have forgotten how to read the reports in Vivado. We will be using the program from Lab 7 (which sums up all the numbers between two integers A and B) as a benchmark in this exercise.

The problem of adding consecutive integers is a known one, and we can exploit this to develop a faster algorithm for this task. To find the sum of all numbers from 0 to N one can use the Gauss formula:

$$\sum_{i=1}^{N} i = \frac{N*(N+1)}{2}$$

To calculate the sum of all integers from A to B, we can calculate this first for B and then for A-1. By subtracting these two we will get the desired sum.

However, as our limited processor has neither multiplication nor division instructions, we are unable to make use of this trick and must resort to a brute force method that sums up all numbers between A and B one by one. The problem with this method is that the execution time is proportional to the difference between the numbers A and B.

Additional Instructions

We could improve the performance of the program significantly by using the Gauss formula. The first problem is the *division*: normally division is a complex operation. However, for our purposes we only need

a simple division by two. From the class you should remember that dividing a binary number by powers of two is very simple. We just need to shift the bits of the binary representation of the number to the right.

At the moment we do not have an instruction in our processor that can do this operation, but the original MIPS instruction set has an instruction named **srl** (shift right logical) for this purpose[1]. We can use this instruction to do the final division by two. However, we still need a way to multiply two numbers.

There are two solutions. One of them is to implement the **multu**[2] (multiply unsigned) instruction from the original MIPS instruction set. The other is to implement the **sll** (shift left logical) instruction that would be able to multiply by two. By shifting and conditionally adding the multiplicand, it is possible to implement a bit-serial multiplication with the **sll**, **srl** and **add** commands. You are free to try this approach, but we recommend implementing the **multu** instruction. For the remainder of this lab, we will assume you plan to implement the **multu** instruction.

Multiplication Result

Unfortunately, our problems do not stop here. While it will be very straightforward to implement the core functionality of the **multu** instruction, the problem is that when you multiply two 32-bit numbers, the result requires 64 bits. The MIPS architecture allows only one register write-back, so it does not support writing a 64-bit value back into the register file.

One solution is to use two additional 32-bit registers that are called 'hi' and 'lo' to hold the 64-bit result. These will contain the most significant 32 bits and least significant 32 bits of the multiplication, respectively. Since these registers are not part of the standard register file, they cannot be accessed directly by other instructions. MIPS provides two additional instructions to move data from these registers: **mfhi** (move from hi) and **mflo** (move from lo)[3]. If we decide to use the **mutlu** instruction, we will have to implement these instructions as well.

Determine the performance

Before we start working on such a major change on our processor, we should first make sure that it is actually worth making the effort. For this lab, we assume that the multiplication result can be expressed with less than 32 bits[4]. This means that the multiplication result will be small enough to be stored only in the **Lo** register. You have two options here – choose the one that suits your challenge needs.

Option 1 (challenging): Use the MARS simulator to write a faster version of the code that you have written in Lab 7. Make sure that the code is functional by testing it for smaller values[5]. If you have problems with the MARS simulator or assembly, refer to Lab 7.

Option 2 (easy): Download and extract the Lab9_helpers.zip file from the course website. This file contains a faster version of the assembly code already implemented for you in "helper_mul.asm". You can also try to run it in the MARS simulator to convince yourself that the code is correct.

---

[1] There is also a similar instruction called **sra** (shift right arithmetic), the difference is that in a logical shift zeroes are inserted from the left side, and in the arithmetic shift the sign (MSB) is preserved. Since the numbers in our example are all positive integers, **sra** and **srl** would have given the same result, but **srl** is easier to implement.

[2] Similarly, there is a **mult** instruction where the operands can be signed. Since all numbers will be positive, **mult** and **multu** will deliver the same result, but **multu** is easier to implement.

[3] There are also **mthi** (move to hi) and **mtlo** (move to lo) instructions to store data to these registers, but we don't need this functionality.

[4] To use the full 64 bits, we would need both **Hi** and **Lo** registers. The shift operation would then be slightly more complex as we would have to add the bit that is shifted out of **Hi** register as the MSB of the **Lo** register. In order to make the lab more manageable we will assume that the **Lo** register will contain the complete result; this limits the maximum number that we can use to slightly more than 32'000.

[5] Although it is tempting, do not use values larger than 100 for A and B. The simulation will just take a very long time and will not give you additional information.

Please note that although the simulator will accept any valid instructions, our MIPS only supports a handful of instructions (the ones in Lab 6 and the new **srl**, **mflo** and **multu**). Make sure that you only use the allowed instructions.

Modify the processor

This is the slightly harder part. Note that all three instructions are actually R-type instructions. If you remember, in the last exercise we had left the ALUControl signal to be 6 bits wide in the ControlUnit (although at that time we only needed 4 bits).

Download the Lab9_student.zip file from the course website. It contains a Vivado project with the processor designed in Lab 8 (MIPS.v) and a testbench (MIPS_test.v) to test the processor. If you look at the MIPS.v file, you may notice that we have changed the output of the processor to make debugging easier. The project also contains the ALU.v from Lab 8, which you have to modify.

Your task is to modify the ALU component so that:

1. It accepts a 6-bit aluop signal.
2. It accepts a 5-bit ShAmt (shift amount) from the MIPS.
3. It takes input B and shifts the value by ShAmt bits to the right when aluop is 6'b000010 (**srl**).
4. It multiplies A and B and writes the result to an internal 32-bit register Lo when aluop is 6'b011001 (**multu**). Note that for the register you will need to add clock and reset signals to the interface as well.
5. It takes the present value of the register Lo and copies it to the output when aluop is 6'b010010 (**mflo**).

Make sure that other instructions are not affected.

Since we have changed the interface of the ALU (now the aluop connection is 6 bits instead of four, and we need additional clock and reset connections), we also have to modify the MIPS.v slightly to make sure that the modified ALU is connected correctly.

Make sure that the new ALU is integrated correctly at the top level. This requires small changes to the module instantiation within the MIPS.v file. You will need to extract the ShAmt signal (Shift Amount) from the instruction (Instr signal) to pass it to the ALU. You may need to declare additional wires for this. You can find bit positions for ShAmt from MIPS reference data:

https://safari.ethz.ch/digitaltechnik/spring2020/lib/exe/fetch.php?media=mips_reference_data.pdf

Show and describe your design modifications to a TA.

Performance

We have a new processor, which hopefully reduces the computation time for large numbers considerably. There is a cost associated with this improvement. There is a more complex operation in the ALU component of the processor (multiplication), which should theoretically also increase the length of the critical path.

When you compare the two implementations, you realize that this is not really apparent. First of all, the synthesizer will make use of the built-in multipliers within the FPGA to implement the costly multiplier. These are substantially faster than building a multiplier using individual gates, so most probably you will not see the penalty in the timing.

To keep the code simple, we use a very simple approach for the Instruction Memory. The program is defined as a constant look-up table. Since the program is embedded into the processor, the synthesizer is able to recognize which parts of the processor are not used and is able to optimize those parts away. If you have a multiplier but do not have an instruction that uses it, there is no need to synthesize a multiplier. As a result, the area numbers that you see also depend on the program you have loaded. This makes comparisons tricky.

Verification

We follow the example from Lab 7 and make sure that the final result is written to the register $t2. We then wait until the processor is in a loop (PC equals to the end loop) and check the contents of the register to see if the value is indeed correct. To do this you again have two options:

Option 1 (challenging): Your assembly program needs to be copied into the text file named "insmem_h.txt". To do this, you can use the Memory Dump option within the MARS simulator. By selecting the "memory segment" as ".text" and "Dump Format" as "Hexadecimal Text" you will be able to generate the required file. All you have to do is to use an editor and make sure that the file has exactly 64 lines; all lines after your real code will be filled with zeroes. If something is not clear, refer to Lab 7.

Option 2 (easy): The Lab9_helpers.zip includes a "helper_insmem_h.txt" file that contains the binary program corresponding to the "helper_mul.asm" assembly program. Rename "helper_insmem_h.txt" to "insmem_h.txt" and place it in the same folder as the project files.

Remember that the InstructionMemory.v file reads "insmem_h.txt" to initialize its contents. By changing this file and recompiling your circuit, you effectively reprogram your processor.

Use the file MIPS_test.v file to test your processor as explained previously. It is a simplified version of the one used in Lab 6 in which we no longer have to read in the expected responses. We simply need a clock generator, an initial reset signal and give the processor sufficient time to finish the calculation. Run the new test bench in the Vivado simulator and monitor the value of 'result' and the PC in the wave window.

Show your correctly running MIPS code to a TA.

Note that this is a rather ad-hoc approach to verification and is unsurprisingly unhelpful if the result is incorrect. In this case, you will need to spend more effort in finding out why the circuit is not working as expected. This could involve tracing additional internal signals in the waveform viewer, implementing a more comprehensive testbench, or any other debugging technique you feel may help you identify the issue(s).

Some Ideas

Now that we have almost a full processor, here are some ideas you can try to implement (optional).

- Use the files from the previous labs to display the output of the program on the 7-segment display.
- Write a counter program in assembly and display the counter on the 7-segment display.
- Write an assembly program to take inputs A and B from the switches and display the multiplied output on the 7-segment display.

Final Words

In this lab we have added instructions to 'enhance' the performance of our processor. In fact, rather than improving the original MIPS architecture we have just added some of the 'missing' instructions to the processor. The addition comes at a cost: more instructions have to be decoded and more hardware resources are required. Processor designers often face this trade-off when improving their designs.

Since we are talking about adding instructions, why don't we just add them the way we like? For example, we could add one instruction called **gauss** that takes N, adds one to it to generate (N+1), multiplies these values, divides the result by two and returns the result in one single cycle. This is possible, but once we start adding non-standard instructions to MIPS, tools (for example compilers, debuggers like MARS) will no longer work with this modified architecture. These tools would also need to be modified, which requires additional work.

There is also the issue of how much we would gain by this **gauss** instruction. Adding a multiplication instruction reduced the calculation time tremendously for larger N (saving us millions of cycles). The gauss instruction would be able to reduce 4 instructions into a single one. The problem is that we would only need this instruction twice, so in total we could save only 6 cycles, overall not a very impressive gain for all

our efforts. Clearly, this is not always the case. Instruction set extensions is an active field of research, and in some cases can offer significant gains with little overhead.

This is the last exercise, so these will be the final words of this lab series. First, we hope that you have learned something from them. In a short time (with some help) you have implemented your own 32-bit processor, wrote programs for it and have improved its performance.

We also hope that you enjoyed the labs and have a better understanding about digital circuits and digital design in general. In the end, you can safely say that you have used professional digital design tools and gained practical experience in designing actual circuits, which are not that far from what is needed in industry.