COL216 Assignment 5 Report

Navneel Singhal, Akash S

January 2020

Implementation details

We developed a simple processor implementing a subset of the MIPS instruction set (add, sub, sll, srl, sw, lw, beq, bne, blez, bgtz, j, jr, jal).
In the previous assignment, we implemented the subset of this subset without the jump and branch instructions, which was tested extensively. In this assignment, we added the jump and branch instructions to allow for the implementation of non-leaf procedures. We tested this subset both on the board as well as Vivado simulation.

Output specifications during running of synthesized code

When you press the reset button the FPGA starts running the instructions stored in the memory. When the toggle switch is '0' then the seven segment displays the last 16 bits of the target register of the last non-zero instruction, and when it is '1' the seven segment display displays the number of clock cycles taken to execute all the instructions.

Specifications of processor

1. We have used a dual port memory, due to which we were able to do sw, lw in two clock cycles. One port was used for reading the current instruction from the memory, while the other was used for reading the data from the memory, which essentially worked like two different memory modules, one for instructions and the other for the data.
2. We have two constants which store the memory range parameters - `memzero` and `memmax`, which are 1024 and 4095 respectively.
3. The machine instructions are loaded into the memory from index 0 to `memzero` - 1.
4. The data memory is stored from `memzero` to `memmax`.
5. The memory is loaded into the BRAM module using a .coe file.
6. All the registers are initialised to zero once at the start of the program, other than the stack pointer, which is initialised to 4095.
7. Invalid instructions are ignored.
8. Each instruction other than lw and sw is executed in a clock cycle, and these two take 2 clock cycles to run.
9. The offset for any branch instruction is given as per MIPS specifications; more specifically, we compute the offset from the program counter, and not the instruction number of the current instruction being executed (which is 1 less than the program counter).

Testing

Testing is an important part of programming. It verifies the correctness of the code.

Corner case testing

Corner case testing tests cases which could have caused an error in a hypothetical design of a processor using similar design decisions but these errors are either resolved or are handled like the MIPS architecture subset should have been expected to be handled (either unhandled or ignored).

1. Number of registers out of bounds - We have used 32 registers in our register file. The register number is specified by a 5-bit value in the MIPS instructions set (rs, rt, rd). So there cannot be number of registers out of bounds as any value specified by the 5-bit value has to lie between 0 and 31.

2. Shifting left and right can be only done for unsigned numbers correctly. If a negative number is shifted then only binary shifting is done (i.e. negative number can become positive or positive number can become negative)

3. Overriding register values also works (i.e. add $t1, $t1, $t2 works, the value of t1 is overridden)

4. There is a limit for number of instructions (as there can only be `memzero` instructions including the program terminating indication line consisting of all 0s), as instructions are stored in memory from 0 to `memzero` - 1.

5. There is also a limit for number of data memory (which is `memmax - memzero` + 1).

6. Since we had to check whether the value is being stored in the memory or not. We used sw followed by lw, which allowed us to check if value was being stored in the memory or not.

7. We used multiple arithmetic instructions followed by lw, sw instructions, and vice versa, and ensured that no instruction was skipped.

8. The MIPS specification tells us that the jump addresses or the branch offsets can be much more than 10 bits (the addressing should correspond to a valid memory location which has an instruction), and this might be an issue with *some* implementations. However, with our implementation, we ensure that for the jump instructions, there will be no error in the running of a program, because we take the last 10 bits, to ensure that the data memory isn't read as an instruction in the jump statement. However for the branch instructions, we don't do this check because the branch is PC-relative and the offset can be negative (hence taking the last few bits can be an issue), and there can be a lot of much more complicated issues like memory out of bounds or data memory being read as instructions and so on. The bottom line is that the programmer should take care of the instruction memory completely on his/her own while using our processor for running a MIPS program.

9. Stack overflow is a very dangerous thing in the implementation of the processor according to the specifications of the memory module given to us, because each time the stack overflows, the instruction memory gets corrupted as well, as we were told to not make two memory modules separately for the instructions and data. We assume that the programmer knows about this issue and can handle this on his/her own. Since we were not sure about whether to end the program execution or give a signal that stack overflow has happened, we didn't handle this and leave this job to the programmer.

Most of these test cases were handled in the assignment 3 and 4, and we focus on the testing of the jump and branch instructions, for which we have made test cases for recursive procedures as well as procedures that call other procedure.

The first example finds the sum of numbers from 1 to 5 using a recursive procedure similar to the classical factorial example, and the second finds $2a + b$ by calling a procedure from a procedure.