Jacob Schwell
Ivan Sekyonda
EECE 552, Computer Design
December 10, 2017

Final Project Design Report

Approach:

In order to calculate the scoreboard with Tomasulo's algorithm and a reorder buffer, javascript is used to parse the data, setup the system, and then simulate what would happen on each clock cycle. The javascript code adds some additional overhead to include error catching that can help the user debug their assembly code for issues including a problem initially parsing, accessing memory that is not available, or anything else that may while running. In the event of an error, an alert message will pop up at the top of the webpage and a call stack will be printed to the javascript console. Once the system is setup with data from the text boxes in the html page, the program simulates what each hardware component would do.

Program:

When all the code and settings are entered, the user presses the run button. The run button calls startScoreboard() which calls runScoreboard() in a try-catch block to handle any errors that get thrown from any functions throughout the process. Much of the process has been abstracted through functions and classes to help make the code more readable and modular.

runScoreboard() starts in a setup phase, which initializes the floating point and integer registers as well as the data memory to the specified values. It also initializes the Instruction Counter (IC), the clock, and an empty scoreboard. Using various functions to gather the instructions and parameters from the html page, the list of instructions is gathered, a reorder buffer is created, and an array of the hardware available is generated. The instructions are parsed using regular expressions to trim them and turn them into *instruction* objects, using a function, getInstructionType(), that parses each line with additional regular expressions.

Once the setup is complete, a loop is started that runs as long as there are instructions left to process or instructions have not yet committed in the reorder buffer. Each iteration of the loop loops through every instruction element in the reorder buffer and checks its current state to determine what to do next.

If the instruction issued, but has not read its source and target yet, it checks if it can read by looking through all instructions in the reorder buffer that were issued before it and ensures that none will write to its source or target. In the event one does, if it already wrote the result to the reorder buffer, this instruction can read directly from that result. If the instruction is able to read, it takes the appropriate values and puts them as inputs to the reservation station it is currently occupying. When an instruction is put in a reservation station, it has a default boolean value that says it cannot start executing; however, later in the loop, there is a check to see if the instruction can start executing in the same clock cycle it read. This prevents multiple instructions in the same functional unit from getting their data and trying to start execution at the same time.

Although instructions can run simultaneously in pipelined hardware, they cannot start in the same cycle too. If an instruction is still in the issue stage, read its data, and its hardware unit is ready, it's next state is set to *exec* so the scoreboard can reflect the transition on the correct clock cycle since *exec* marks completion of execution.

If the current instruction in the reorder buffer loop finished execution, its results need to be written to the reorder buffer; however, only one instruction can use the common data bus per cycle, so a boolean value is used to track if any instruction has written in the current cycle. If this instruction is able to write, an execute instruction function is called and passed the current instruction. The execute instruction uses the data from the specific hardware unit holding the instruction to determine how the instruction should be executed and the result is saved to the instruction object's result member. If the instruction was unable write its results, nothing happens to it and it will try again on the next clock cycle. Instructions get analyzed in each cycle in a FIFO style, so there can't be starvation of instructions, where one always loses its turn to write. Otherwise, the instruction's next state is set to *writeback*. On the next clock cycle, once the write is complete, the reorder buffer recognizes this instruction is ready to be committed.

After the reorder buffer loop, the main loop checks if there is an instruction at the current IC and if there is room in the reorder buffer to try issuing another instruction. If the first two conditions are met, it will check the reservation stations for the hardware required by the next instruction. If there is room, it will issue the next instruction and advance the IC; otherwise, it will stall the IC and not issue another command.

Following attempting to issue another instruction, a loop iterates over every instruction in the reorder buffer to update the states of those that need to be updated. The instruction objects have state and nextState members to ensure the all get updated synchronously, so by looping through each instruction the program can make their state member equal to their nextstate member, if there is a difference, and note the clock cycle the transition happened on in another member variable. Afterwards, a function is called on the reorder buffer to commit the next instruction. If the instruction at the beginning of the reorder buffer is not ready, it will return null; otherwise, it will set the appropriate value in the register files or memory, or branch if appropriate, then remove the instruction from the buffer, add it to the scoreboard, and return it at the end of the function. If a branch was taken, the reorder buffer removes all following instructions and adds them to the scoreboard. The instruction is then returned so runScoreboard(). If the committed instruction was a taken branch, the clock is incremented to account for the cycle required to flush the buffer.

Once an instruction's result has been written to the reorder buffer, it is no longer needed in the reservation station, so a function is called to clear any instructions that have been written from every reservation station. Once each section of the loop completes, the clock can be incremented and the process repeated. A check was put in place to compare the clock cycle against the instruction count and allow the user to quit the process conditions look like a possible infinite loop.

After the scoreboard is complete, the scoreboard array is sorted based on issue count to ensure any out of order additions, from taken branches, are corrected. Lastly, functions are called to interact with the html document and display the scoreboard, registers, and data memory.

Tests:
This simulator was tested with several small assembly programs to ensure it works for all the requirements. The most common ones used for debugging and ensuring major components were functioning properly are described below.

```
SUBI $1, $1, $1
SUBI $2, $1, $1
ADDI $1, $1, #16
L.D F9, 1($1)
MULT.D F0, F1, F0
ADD.D F4, F0, F2
S.D F4, 0($2)
ADDI $2, $2, #8
BNE $1, $2, #4
DIV.D F11, F9, F9
MULT.D F8, F8, F8
L.D F4, 1($2)
DIV.D F10, F4, F4
```

This program was used to check different arithmetic functions, branches, and load/store operations. It was tested against the mid-semester scoreboarding project to ensure the data memories were the same in the end and also matched expectations.

```
SUB.I $1, $1, $1  ; clear $1
SUB.I $2, $1, $1  ; clear $2
ADD.I $1, $1, #16 ; $1 = 16
L.D F9, 1($1)     ; ld dm[17]->f9
MULT.D F0, F1, F0 ; f0 = f1 * f2 <-- Loop dest
ADD.D F4, F0, F2  ; f4 = f0 + f2
S.D F4, 0($2)     ; store f4->dm[0+$2]
ADD.I $2, $2, #8  ; $2 = $2 + 8
BNE $1, $2, #4    ; if ($1 != $2) go to Loop dest
DIV.D F11, F9, F9 ; f11 = f9/f9      These are just to make sure
MULT.D F8, F8, F8 ; f8 = f8^2        branches clear instructions
L.D F4, 1($2)     ; f4 = dm[1+$2]    appropriately if taken
DIV.D F10, F4, F4 ; f10 = f4/f4
```

This assembly code was used to ensure the branch would use the correct data and the reorder buffer would clear the instructions after the branch when it was taken, as well as more

functionality tests for load and store operations. It also shows that comments can be put after each instruction to help describe what each instruction is trying to accomplish.

```
SUB $1 $1 $1
ADD.I $1 $1 #5
L.D f4 0($1)
```

This short program was used when a problem was found with instructions using the wrong, or old, data when used one after the other, then later specifically with loads due to how the data was read for them. This program allows easy verification that the load took the correct value of $1.

```
L.D F6 34($2)
L.D F2 20($3)
MULT.D F0 F2 F4
SUB.D F8 F6 F2
DIV.D F10 F0 F6
ADD.D F6 F8 F2
```

This was a quick program to see that instructions wait until the data has been written to the reorder buffer before it reads and starts executing; although, it can read and write in the same cycle.

```
L.D F6 10($2)
ADD.D F5 F6 F6
ADD.D F4 F6 F6
```

This program was used to show that although two instructions waiting for the same data can read in the same cycle, only one can start execution in a single cycle for each functional unit.