Minor Exam: Computer Architechture (COL216)

Divyanka Chaudhari (2019CS50429)

March 21, 2021

1 Part 1: DRAM Implementation

File name: `MIPS_simulator.cpp`
Input through command line: `ROW_ACCESS_DELAY COLUMN_ACCESS_DELAY inputFile.txt`

This part is a simple extension of previous assignment. In our previous assignment, we used hashmap as a substitute to main memory. Here, we use a big array of size `1024*1024` to simulate it as DRAM. The variable `clockNumber` is used to keep a count of clock cycles. `currentRow` is used to keep track of which row is currently activated or is in the buffer. We run a loop through each of our line and keep executing them sequentially. Instructions, if they're not right in some way, we throw an error. If they are, we continue.

For all instructions except `lw` and `sw` we increment the clock cycle by 1. For `sw` and `lw`, we can have cases as mentioned in the assignment description and we increment `clockNumber` on the basis of `currentRow`. We also keep track of number of row activations using rowBufferUpdates. At each step of our loop, we print necessary information as expected and after the execution we print total number of instructions, clock cycles and row buffer updates.

2 Part 2: Non-blocking memory

File name: `MIPS_simulator2.cpp`
Input through command line: `ROW_ACCESS_DELAY COLUMN_ACCESS_DELAY inputFile.txt`

The code only considers `add, addi, sw` and `sw` instructions (as instructed) unlike Assignment 3, where a wider variety of instructions are considered.

This is an extension of DRAM implementation above. Here, we have additional tracking variable named `saveCycles` which keeps a track of total time an `sw` or `lw` operation may take. Since `lw` operation keeps the register busy and `sw` operation keep the memory busy, we temporarily keep track of them in `busyRegister` and `busyMemory`.

When a DRAM request is made, we keep continuing sequentially to the next step if the step doesn't use `busyRegister` or `busyMemory` by decrementing `saveCycles` simultaneously so that we can keep track if it is being run parallel to the DRAM operation going on. While saveCycles is being decremented and is geater than 0, we do NOT increment `clockNumber` because the process is parallel. If we encounter a busy register or memory space, we convert `saveCycles` to 0 and wait until the DRAM procedure is completed. Once done, we start our operation again sequentially and do the same thing if we encounter `sw` or `lw`.

3 Strengths and Weaknesses

1. **Strengths:** The code perfectly executes what is asked. The optimization here is during non-blocking memory where we effectively utilise the time between `sw` and `lw` to execute other functions. Since we used an array, we have a constant access time. When run parallely, the output shows cycles being exeecuted side-by-side using the same cycle number.
2. **Weaknesses:** We utilise a large space by allocating DRAM to a large memory which may not be fully used. To print memory used, we run a loop over the whole array, which may not be time efficient. We run instructions sequentially and stop when we encounter busy registers or memory. We may run the latter instructions and come back to busy instruction. But, this may require keeping track of more than one busy registers and memory and would be a rather complicated process but it'd be certainly efficient than this.

4 Testing

I tried to include all edge cases, including cases where it throws error.

1. **Syntax errors:**

   (a) add2, 31
   addi `$r2`, `$r2`, −32
   mul `$r3`, `$r2`, `$r1`
   j 15
   sw `$r3`, 83910380
   lw `$r4`, 2
   mul `$r3`, `$r2`, `$r1`
   addi `$r2`, `$r2`, -32

   Output: Invalid Syntax at line:1

   (b) addi `$r1`, `$r2`, 31
   addi `$r2`, `$r2`, -32
   mul i3,r2, `$r1`
   j 15
   sw `$r3`, 83910380
   lw `$r4`, 2
   mul `$r3`, `$r2`, `$r1`
   addi `$r2`, `$r2`, -32

   Since i3 is not a valid register name, the code gives a Syntax error at line 3 after printing the register values for the fisrt 2 instructions.

   Output:
   Cycle description:

   Cycle 1: `$r1` = 0000001f
   Cycle 2: `$r2` = fffffe0
   Invalid Syntax at line:3

   (c) addi `$r2`, `$r2`, -32
   mul `$i3`, `$r2`, `$r1`
   j 15
   sw `$r3`, 83910380
   lw `$r4`, 2
   mul `$r3`, `$r2`, `$r1`
   addi `$r2`, `$r2`, -32

   Since a comma is missing in the first line, the code throws a syntax error at line 1

   Output: Invalid Syntax at line:1

   (d) addi `$r1`, `$r2` 31sdsds
   addi `$r2`, `$r2`, -32
   j 31

   Output: Invalid Syntax at line: 1

(e) a
addi `$r2`, `$r2`, -32

Output: Invalid Syntax at line: 1

2. **Data memory overflow and accessing uninitialized memory**

(a) addi `$r1`, `$r2`, -3221
addi `$r2`, `$r2`, -32
Addition of 2 registers
add `$r3`, `$r2`, `$r1`
lw `$r1`, 0

Output:
Cycle description:

Cycle 1: `$r1` = fffff36b
Cycle 2: `$r2` = fffffe0
Cycle 3: `$r3` = fffff34b
Trying to access uninitialised memory at line:5

(b) addi `$r1`, `$r2`, -3221
addi `$r2`, `$r2`, -32
Addition of 2 registers
add `$r3`, `$r2`, `$r1`
sw `$r1`, 322443

Output:
Cycle description:

Cycle 1: `$r1` = fffff36b
Cycle 2: `$r2` = fffffe0
Cycle 3: `$r3` = fffff34b
Memory location not accesible at line: 5

3. **Instruction memory overflow**

The input file `testcase4.txt` containing 50000 lines of instructions was run.
Output: Instruction Memory Overflow

4. **Working of the code:** Workable test cases and output for each part can be found in the folder testCases.