COL-216
ASSIGNMENT-5
DATA REQUEST MANAGER FOR MULTICORE PROCESSOR

Made By

Dishant Dhiman *2019CS10347*

R Hari Shankar *2019CS10386*

DESIGN DESCRIPTION

We have created a structure for cpu which will contain the structure for the multiple cores and the Memory Request Manager structure which we will use to perform the instructions based on memory . The cpu is the parent body and will be used to simulate the multicore processor .

Each core will perform the function of reading the file into the instruction vector and checking whether the syntax is correct. The function for the various instruction type like add,lw,beq etc has been created to perform the required instruction. The simulate function of core performs the instruction along with the required increment in instruction set.

The MRM structure (Memory Request Manager) is the component where the dram requests are stored along with access to the DRAM memory. The send request function is used to send a request to the pending request vector which keeps track of memory instructions which need to be executed. The request issued is used to send the pending request to DRAM request and contains the conditions for forwarding, reordering request etc. and helps to resolve hazards. The editCore function is used to execute the memory instructions.

The simulate function of cpu is used to run the whole multicore processor which evaluates all the files and also provides provision for halting when memory request is full.

We have maximized the throughput as we have implemented memory reordering so that instructions calling same row address are implemented together which will significantly reduce the number of clock cycles required . It is also a non-blocking implementation so the instruction which do not depend on changes due to lw,sw are executed without blocking.

Moreover forwarding is also implemented so that changes to same memory/register location can be performed in significantly less clock cycles.

STRENGTHS OF THIS IMPLEMENTATION

1) The memory request ordering allows us to execute sw or lw commands in less clock cycle than the initial DRAM implementation as these commands are executed such that the memory changes are made by performing changes in memory having same

row buffer in order to reduce clock cycles taken for the row access delay thus enhancing the performance of the simulator.
2) The non-blocking and forwarding allows us to improve throughput significantly.
3) The ordering also helps to reduce the number of row buffer updates.
4) The multicore helps us to perform multiple programs simultaneously.
5) These reduction in running time can be seen from the figures given in the testcases.
6) Stalling is avoided as much as possible unless an unsafe(dependence detected) instruction needs to be executed or the dramRequest runs out of space.
7) Taken care of data and control hazards
8) Throughput and clock cycle is minimized.

"dramActive" variable also keeps tabs of the DRAM so that if another DRAM command comes it will wait for the first one to be executed . This minimizes cross channelling of the DRAM memory .

All the changes in memory, register and clock are given if intermediate values need to be checked.

WEAKNESS (SHORTCOMINGS) OF THIS IMPLEMENTATION

In the case when multiple files (inputs) perform changes to the same memory location, it will lead to ambiguity in result as the memory is common for all the cores which might lead to incorrect answer.

The program will stall when no. of memory request exceed size of request manager or when data hazard is detected.

HOW TO RUN THE PROGRAM

1) Set your directory to the directory in which the program is present using your terminal/console.
2) Now type g++ multicore.cpp -o interpreter.out
3) Now type ./interpreter.out
4) Input format as given in problem statement.

TESTING STRATEGY

To check the correctness of our code as an multicore processor interpreter of the mips assembly language we have used exhaustive test cases so that our program runs correctly in all cases (including corner cases).

Test Cases:

- Testcase1 : When multiple files is given using only the add ,sub ,mult ,beq ,bne ,slt ,addi ,j instructions . (multicore processor check)
- Testcase2: To check non blocking
- Testcase3: To check request reordering along with non blocking
- Testcase4: To check request reordering along with non blocking and forwarding
- Testcase5: When multiple files change same memory location (ambiguous case).
- Testcase6: when clock cycle > halt time

Invalid Input Cases:

- When input file is not found.

- When input file is empty.
- Incorrect register name is given.
- Syntax error for various instruction.
- Memory used more than 2^{20} bytes.
- When instruction other than add ,sub ,mult ,beq ,bne ,slt ,lw ,sw ,addi ,j is used.