INDIAN INSTITUTE OF TECHNOLOGY, DELHI

ASSIGNMENT DESIGN

Problem Statement
MIPS simulator with DRAM timing model.

APAR AHUJA | ENTRY NO. 2019CS10465

Course - COL216 | Prof. Preeti Ranjan Panda

Compiled on March 21, 2021

COL216 - Minor

Apar Ahuja

March 21, 2021

1 Approach and Design

1.1 Input - Output

- **Input:** minor.cpp takes line by line input from a *file_name.txt* file.
- **Command Line Input Format:** *./executable file_name.txt row_delay col_delay implement_part1*
. row_delay(default 10) col_delay(default 2) implement_part1(default 0/false) are all optional.
- **Output:** It prints the mips instruction, modified registers, modified memory locations and DRAM
. activity(if any) after executing each instruction on the console. After execution, it prints the total number
. of clock cycles, number of row buffer updates and the number of times each instruction was executed.

1.2 Design

- Global Variables:
. - **row_buffer:** array to store buffer row. It's first index stores row number (-1 by default)
. - **buffer_update_cnt:** stores the number of buffer updates -
. (1. new row copied or 2. buffer value changed in sw)
. - **row_delay:** clock cycles required to transfer/copy rows between memory and buffer.
. - **col_delay:** clock cycles required to transfer/copy element between register and buffer.
. - **part1:** boolean that denotes if only part1 has to be implemented (false by default)
. - **PC:** program counter that represents the memory position of next instruction to be executed.
. - **PARTITION:** represents the end of instructions stored in memory
. - **DATA_START:** represents the starting address of data section in memory (524288 by default)
. - **memory[1024][1024]:** DRAM memory matrix representing the MIPS memory of size 2^{20} bytes.
. - The memory is byte-addresable
. - Program memory has been divided into two sections:-
. - **Instruction** memory (starting from address 0)
. - **Data** memory (starting from address 524288).
- struct REGI: structure constructor to initialize all registers and instruction counts to zero
. - **ins_map:** hash map that maps given instruction(string) to integer. eg. ins_map["add"] = 0
. - **reg_map:** hash map that maps given register(string) to integer. eg. reg_map["$t0"] = 8
. - **labels:** hash map that maps given label(string) to its memory address. eg. labels["loop"] = 12
. - **label:** vector that stores the labels used in jump and branch instructions during compile time
. - **ins_cnt[10]:** array to store the number of times each instruction is executed.
. - **reg[32]:** array to store the register contents during execution of the MIPS program.
. - **print_reg_file:** function to print the register file contents after each cycle.
. - **execution_stats:** function to print the relevant statistics at the end of execution (clock-cycle and
. - instruction count).
. - **stat_update:** function to update the instruction count and clock cycle count.
. - **cycle:** function to count the number of cycles required to perform the given lw/sw instruction.
- function simulator: simulates MIPS program by executing instructions stored in the memory.
- function parseInstruction: parses a given instruction line and returns a vector of arguments present
. and a boolean value representing if the line is valid or not.

- function validId: checks if the given label is valid identifier or not. Returns a boolean value.
- function checkIfLabel: checks if the given line contains a label or not. Returns the label
. (if any) along with the remaining line for future parsing
- function linker: resolves all the labels used in j/beq/bne statements by replacing them with their address
. values (using **labels** hash map).
- function isInteger: checks and returns true if input string represents an integer else returns false.
- function decimalToHexadecimal: converts input decimal(base 10) integer into signed hex(base 16).
- function addToMemory: it stores the given MIPS instruction in memory if it is valid, else raises error.
- function cycles: function to count the number of cycles required to perform the given lw/sw instruction.

1.3 MIPS Memory

According to problem specifications, the total memory available to MIPS program is 2^{20} bytes. For my design, I have partitioned this memory into **two sections:- Instruction memory and Data memory.** Instruction memory spans from address 0 to 524288, whereas, Data memory spans from address 524288 to 1048575. Memory address n is stored at **row = [n/1024] and column = n (mod 1024)**.
Our memory is represented by an **Integer Array**, where each array location represents a byte. The memory is **byte-addressable** and **each instruction occupies 4 array locations (or bytes)**. The storage format used is simple. First location stores the instruction code (for e.g. 0 for "add", 1 for "sub",...,9 for "addi"), while the second, third and fourth locations store the arguments of the instructions (like register number, immediate value, jump address etc.). Since instruction memory is only from 0 to 524288. On the other hand, each integer stored in Data memory also occupies 4 array locations (first location contains the integer, and other 3 act as buffer). Also, to account for partitioning the memory, a constant DATA_START (= 524288) is added to the address provided in lw and sw commands for smooth access.

1.4 Algorithm

- Input file is opened, a new register file *rf* is created, *line_number* is set to 1 and a boolean *flag* is declared.
- Initialize PC to 0
- Read the file line by line and call *addToMemory()*.
- **For each line** -
. - Set *flag* = true if line is successfully added to memory (no errors), else set *flag* = false.
. - Increase *line_number* by 1 after each line and *PC* by 4 after adding an instruction.
. - Add labels (if any) with their addresses to *rf.labels*(for label declaration) and *rf.label*(for label call in
. bne/beq/j)
- Set PARTITION equal to PC i.e. the end of last instruction stored in memory. Reinitialize *PC* to 0.
- Call *linker()* to resolve labels used within j/beq/bne instructions and replace them with their corresponding
. addresses. Set *flag* = true if linking is successful, else false.
- Call *simulator()* to execute instructions stored in memory (starting from address 0) in a sequential manner,
. and print modifications after each clock cycle. *simulator()* also prints executions statistics of the
. MIPS program at the end of execution on the console.
- Set *flag* = true if simulation is successful, else set *flag* = false.
- **Note -** Relevant errors are raised and **program is terminated** if *flag* is false at any moment.

1.5 Simulator Working

1.5.1 Variable Definitions -

- *cycles_to_go*: number of cycles to go through for current dram instruction. It is -1 by default if no DRAM instruction is being run.
- *ins_last*: stores the type(sw/lw) of DRAM instruction being executed.
- *reg_used1 / 2*: registers bring used to go through current dram instruction. It is -10 by default if no DRAM instruction is being run.
- *temp_pc*: PC value of current DRAM instruction stored separately. It is -10 by default if no DRAM instruction is being run.

1.5.2 Algorithm -

while (PC < PARTITION OR cycles_to_go != -1) do
. - if part1 is true and *cycles_to_go* != -1 then go to *skipped_ins* label as DRAM instruction is running
. - switch instruction_code -
. - if dram instruction is going on and a used register is being used. Or PC crosses
. PARTITION then break;
. - otherwise execute instruction(except lw/sw) at PC, increment PC by 4 and then break
. - if sw/lw instruction then set cycles_to_go using the function cycles(). store ins_last, reg_used1 - 2,
. temp_pc, increment PC by 4 and then break
. - *skipped_ins:* if DRAM instruction is going on decrease cycles_to_go by 1. Then, if cycles_to_go
. become 0 then execute the instruction using ins_last, reg_used1 and 2, temp_pc values and
. set cycles_to_go back to -1.
if row_buffer was loaded during execution then perform a write-back, print execution statistics and return.

1.6 Strengths and Weaknesses of Approach

- Strengths:
. - Non-blocking memory model reduces clock cycles required. All instructions executed until it is unsafe.
. - An instruction is safe to execute until it doesn't uses a register that is being used by the DRAM for
. read/write operation.
. - Non-blocking model can be removed by setting the implement_part1 command line argument to 1.
. - **Extra:** All mips instructions of assignment 3 are implemented - j, slt, beq and bne. Hexadecimal
. conversion is available. Print full register file anytime using the function *print_reg_file*. The code prints
. instruction count of all instructions at the end of execution.
- Weaknesses:
. - Consecutive sw/lw instructions take a lot of time and can be improved if not colliding instructions are
. present on the same row.
. - Register dependency can be improved to only one register if dram request carries address value instead
. of register names.
. - Implement a dram queue and dependency list to execute add/addi instructions independent of all
. previous sw/lw instructions can reduce the total number of clock cycles.

1.7 Raising Errors

- if input file name is incorrect we raise *Error: Unable to open file..*
- if number of instructions is greater than **174763** we raise *Error: Instruction memory overflow..*
- if label declaration is not valid we raise *Syntax Error (Illegal Label)*
- if same label is declared twice we raise *Error: Same label used to represent different addresses.*
- if the instruction is syntactically incorrect we raise relevant errors of the form *Syntax Error __*
- if addi tries to load number greater than $2^{15} - 1$ or less than -2^{15} we raise *Immediate Overflow Error*
- if j tries to load address greater than $2^{26} - 1$ we raise *Immediate Overflow Error: At line_number*
- if undeclared label is called then we raise *Linking Error: Unable to resolve label used on line: _line*
- if add/sub/mul leads to a result greater than $2^{31} - 1$ or less than -2^{31} we raise *Error: Arithmetic Overflow.*
- if beq/bne/j try to access non-instruction memory we raise *Warning: Program jumped to a non-instruction.*
- if lw/sw try to access non-data memory we raise *Error: Program is trying to access an unavailable memory.*
- if lw/sw try to access a non-word location we raise *Error: Memory address is not word-aligned.*
- if any instruction tries to access a reserved register (at, k0, k1 or gp) we raise *Access Error:....*
- if row/column delay is negative we raise *Negative Delay Entered.*
- if row/column delay or part1 argument is non-integer we raise *non-integer entered.*

2 Testing Strategy

- Various different types of test cases were generated and used as a part of our extensive testing strategy.
- Types of Test Cases Used:
. - **Individual:** Only one instruction is input. Test cases each of varying size - Small: 5 and Large: 30
. - **DRAM:** Only sw/lw are used for testing. Size of Test Cases: Small-5, Medium-15 and Large-30
. - **Arithmetic:** Only addi/add are used for testing. Size of Test Cases: Small-5, Medium-15 and Large-30
. - **Mixed:** All instructions are used for testing. Size of Test Cases: Small-10 and Large-30
. - **Overlapping:** Registers overlap in consecutive instructions of sw/lw. Size: Small-5 and Large-15
. - **Non-Overlapping:** Registers don't overlap in consecutive instructions of sw/lw: Small-5 and Large-15
. - **Part - 1 Only:** Only part 1 is implemented. Overlapping register instructions are processed separately.
. - **Exponential:** A program to find a^b was tested on our MIPS simulation
. - **Factorial:** A program to find factorial = $a!$ was tested on our MIPS simulation
. - **Array Sum:** A program to find the sum of all elements of an array was tested on our MIPS simulation
. - **Array Max:** A program to find the max element of an array was tested on our MIPS simulation
. - **Corner Cases:** 1. **Incorrect File Name**, 2. **Instruction Overflow**, 3. **Illegal Label**, 4. **Syntax**
. **Error**, 5. **Arithmetic Overflow**, 6. **Immediate Overflow**, 7. **Linking Error**, 8. **Invalid Jumps**,
. 9. **Non-word Memory Address**, 10. **Invalid Data Access**, 11. **Invalid Register Access**,
. 12. **Zero Register functioning**, 13. **Negative row/column delay**, 12. **Non-Integer delay**

3 Result

Source code **passed all test cases**. Relevant error messages(if any) were displayed for invalid inputs.

Console Interface –

```
cycle 1: Instruction - addi $t0 $t0 100: $t0 = 100
cycle 2: DRAM request issued(Store)
cycle 3-14: Instruction - sw $t0 104($zero):
         DRAM Activity: Copy row 512 from memory to buffer.
                        Copy Data to column 104 from $zero.
                        Memory Address: 104 - 107 = 100
cycle 15: DRAM request issued(Load)
cycle 16-17: Instruction - lw $t1 4($t0): $t1 = 100
         DRAM Activity: Copy Data at column 512 to $t1.
                        Memory Address: 104 - 107 loaded
cycle 18: Instruction - addi $t1 $t1 1: $t1 = 101
cycle 19: Instruction - addi $t3 $t3 1: $t3 = 1
cycle 20: DRAM request issued(Store)
cycle 21: Instruction - add $t1 $t1 $t3: $t1 = 102
cycle 22: Instruction - add $t2 $t2 $t3: $t2 = 1
cycle 21-22: Instruction - sw $t0 96($zero):
         DRAM Activity: Copy Data to column 96 from $zero.
                        Memory Address: 96 - 99 = 100
cycle 23: Instruction - add $t3 $t3 $t3: $t3 = 2
cycle 24: Instruction - addi $t0 $t0 2: $t0 = 102
cycle 25: DRAM request issued(Store)
cycle 26: Instruction - add $t1 $t1 $t3: $t1 = 104
cycle 27: Instruction - add $t3 $t3 $t3: $t3 = 4
cycle 26-27: Instruction - sw $t0 96($zero):
         DRAM Activity: Copy Data to column 96 from $zero.
                        Memory Address: 96 - 99 = 102
cycle 28: Instruction - addi $t0 $t0 2: $t0 = 104
cycle 29: Instruction - add $t2 $t2 $t3: $t2 = 5
cycle 30: Instruction - add $t2 $t2 $t3: $t2 = 9

Executing Write-Back. Write-Back clock cycles are excluded below.

Total clock cycles: 30
Total row buffer updates: 4

Number of instructions executed for each type are given below:-
add: 7, addi: 5, beq: 0, bne: 0, j: 0
lw: 1, mul: 0, slt: 0, sub: 0, sw: 3
Program executed successfully!
```