INDIAN INSTITUTE OF TECHNOLOGY, DELHI

ASSIGNMENT DESIGN

Problem Statement

Develop an interpreter for MIPS assembly language instructions.

APAR AHUJA | ENTRY NO. 2019CS10465

ARNAV TULI | ENTRY NO. 2019CS10424

Course - COL216 | Prof. Preeti Ranjan Panda

Compiled on March 12, 2021

COL216 - Assignment 3

Arnav Tuli | Apar Ahuja

March 12, 2021

1 Approach and Design

1.1 Input - Output

- **INPUT:** assignment_3.cpp takes input from a .txt file. Enter input file name as an optional command line argument or manually set *file_name* variable in the assignment_3.cpp file.
- **OUTPUT:** It prints the register file in hexadecimal after executing each instruction on the console. After execution, it prints the total number of clock cycles and the number of times each instruction was executed.

1.2 Design

- Global Variables:
. - **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 (699052 by default)
. - **memory[1048576]:** memory array 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 699052).
- 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.
- 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.

1.3 MIPS Memory

According to problem specifications, the total memory available to MIPS program is 2^{20} bytes. For our design, we have partitioned this memory into **two sections:- Instruction memory and Data memory.** Instruction memory spans from address 0 to 699051, whereas, Data memory spans from address 699052 to 1048575 (basically, all the way to the end). **The division is nearly 2:1**, which is justifiable, as for every memory location a minimum of 2 instructions will be required (lw and sw). Hence, the division cannot be more optimal, while keeping the implementation general.
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 699051, a **maximum of 174763 instructions** can be stored. 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). Hence, a **maximum of 87381 words** can be stored in the data memory. Also, to account for partitioning the memory, a constant DATA_START (= 699052) 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 register file contents after each clock cycle. *simulator()* also prints executions statistics of the
. MIPS program 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 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:....*

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:
. - **Single:** Only one instruction is input. Test cases each of varying size - Small: 1 and Large: 10
. - **Relational:** Only bne/beq/slt/j are used for testing. Size of Test Case used: Small-10, Large-15
. - **Arithmetic:** Only addi/add/sub/mul are used for testing. Size of Test Case used: Small-5, Large-10
. - **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**

3 Result

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

Console Interface (relational instructions) –

```
Register file contents after 1 clock cycles:

zero: 00000000 at: 00000000 v0: 00000000 v1: 00000000 a0: 00000000
a1: 00000000 a2: 00000000 a3: 00000000 t0: 00000000 t1: 00000000
t2: 00000000 t3: 00000000 t4: 00000000 t5: 00000000 t6: 00000000
t7: 00000000 s0: 00000000 s1: 00000000 s2: 00000000 s3: 00000000
s4: 00000000 s5: 00000000 s6: 00000000 s7: 00000000 t8: 00000000
t9: 00000000 k0: 00000000 k1: 00000000 gp: 00000000 sp: 00000000
fp: 00000000 ra: 00000000


Register file contents after 2 clock cycles:

zero: 00000000 at: 00000000 v0: 00000000 v1: 00000000 a0: 00000000
a1: 00000000 a2: 00000000 a3: 00000000 t0: 00000000 t1: 00000000
t2: 00000001 t3: 00000000 t4: 00000000 t5: 00000000 t6: 00000000
t7: 00000000 s0: 00000000 s1: 00000000 s2: 00000000 s3: 00000000
s4: 00000000 s5: 00000000 s6: 00000000 s7: 00000000 t8: 00000000
t9: 00000000 k0: 00000000 k1: 00000000 gp: 00000000 sp: 00000000
fp: 00000000 ra: 00000000


Register file contents after 3 clock cycles:

zero: 00000000 at: 00000000 v0: 00000000 v1: 00000000 a0: 00000000
a1: 00000000 a2: 00000000 a3: 00000000 t0: 00000000 t1: 00000000
t2: 00000001 t3: 00000000 t4: 00000000 t5: 00000000 t6: 00000000
t7: 00000000 s0: 00000000 s1: 00000000 s2: 00000000 s3: 00000000
s4: 00000000 s5: 00000000 s6: 00000000 s7: 00000000 t8: 00000000
t9: 00000000 k0: 00000000 k1: 00000000 gp: 00000000 sp: 00000000
fp: 00000000 ra: 00000000


Register file contents after 4 clock cycles:

zero: 00000000 at: 00000000 v0: 00000000 v1: 00000000 a0: 00000000
a1: 00000000 a2: 00000000 a3: 00000000 t0: 00000000 t1: 00000001
t2: 00000001 t3: 00000000 t4: 00000000 t5: 00000000 t6: 00000000
t7: 00000000 s0: 00000000 s1: 00000000 s2: 00000000 s3: 00000000
s4: 00000000 s5: 00000000 s6: 00000000 s7: 00000000 t8: 00000000
t9: 00000000 k0: 00000000 k1: 00000000 gp: 00000000 sp: 00000000
fp: 00000000 ra: 00000000
```

```
Register file contents after 3 clock cycles:

zero: 00000000 at: 00000000 v0: 00000000 v1: 00000000 a0: 00000000
a1: 00000000 a2: 00000000 a3: 00000000 t0: 00000000 t1: 00000000
t2: 00000001 t3: 00000000 t4: 00000000 t5: 00000000 t6: 00000000
t7: 00000000 s0: 00000000 s1: 00000000 s2: 00000000 s3: 00000000
s4: 00000000 s5: 00000000 s6: 00000000 s7: 00000000 t8: 00000000
t9: 00000000 k0: 00000000 k1: 00000000 gp: 00000000 sp: 00000000
fp: 00000000 ra: 00000000


Register file contents after 4 clock cycles:

zero: 00000000 at: 00000000 v0: 00000000 v1: 00000000 a0: 00000000
a1: 00000000 a2: 00000000 a3: 00000000 t0: 00000000 t1: 00000001
t2: 00000001 t3: 00000000 t4: 00000000 t5: 00000000 t6: 00000000
t7: 00000000 s0: 00000000 s1: 00000000 s2: 00000000 s3: 00000000
s4: 00000000 s5: 00000000 s6: 00000000 s7: 00000000 t8: 00000000
t9: 00000000 k0: 00000000 k1: 00000000 gp: 00000000 sp: 00000000
fp: 00000000 ra: 00000000


Register file contents after 5 clock cycles:

zero: 00000000 at: 00000000 v0: 00000000 v1: 00000000 a0: 00000000
a1: 00000000 a2: 00000000 a3: 00000000 t0: 00000000 t1: 00000001
t2: 00000001 t3: 00000000 t4: 00000000 t5: 00000000 t6: 00000000
t7: 00000000 s0: 00000000 s1: 00000000 s2: 00000000 s3: 00000000
s4: 00000000 s5: 00000000 s6: 00000000 s7: 00000000 t8: 00000000
t9: 00000000 k0: 00000000 k1: 00000000 gp: 00000000 sp: 00000000
fp: 00000000 ra: 00000000


Total clock cycles: 5

Number of instructions executed for each type are given below:-
add: 0, addi: 1, beq: 1, bne: 0, j: 2
lw: 0, mul: 0, slt: 1, sub: 0, sw: 0
Program executed successfully!
```