COL216 Assignment 3

Pratyush Saini(2019CS10444)

Anirudha Kulkarni (2019CS50421)

13th March 2021

1 Preface

The assignment consists of a C++ program that takes in a MIPS assembly language program as input and interprets it by maintaining internal data structures representing processor components such as Register File and Memory Unit, and executing the operations indicated by the instructions.

The program handles a small subset of MIPS assembly language instructions: add, sub, mul, beq, bne, slt, j, lw, sw, addi.

2 Input File

The Input to the program will be an InputFile consiting of MIPS assembly program consition of the following instructions:

1. **R-type:** add, sub, mul, slt, addi.
2. **Load and Store** lw, sw.
3. **Branch** beq, bne, j.
4. **Label declarations**.

Any amount of extra white spaces is taken care of while executing the input instructions.
R-type instructions have the general format: inst_Reg1,_Reg2,_Reg3.

Load and Store instructions have the general format: inst_Reg1,_str.
str can be either name of the variable or the address in the format offset($Reg). The address is calculated by adding offset to the value stored in register Reg.

Branch instructions have the general format: inst_Reg1,_Reg2,_label, except for jump instruction (j label).

3 Elements Involved in Data Path

Major components required to execute each class of MIPS instructions are:

3.1 Program Counter

Program Counter is a register that holds the address of the current instruction. After each input of instruction, increment the PC to the address of the next instruction 4 bytes later.
In case the instruction has branching associated with it, Program counter is updated to the address of the label associated with it.

The program counter is a 32-bit register that is written at the end of every clock cycle and thus does not need a write control signal.

3.2 Register File

The processor's 32 general-purpose registers are stored in a class named as Register File. A register file is a collection of registers in which any register can be read or written by specifying the number of the register in the file. The register file contains the register state of the computer.

Variables associated with the RegisterFile:

1. **Register-Array** of size 32 storing the values holded by the corresponding Registers.
2. **Register-Names-Array** of size 32 storing the names of all the registers.

Methods associated with the RegisterFile:

1. **getRegisterData:** Takes in name of Register as the Input and returns the value stored in the corresponding Register location.
2. **setRegisterData:** Takes in name of Register and value to write as Input and writes the value to that Register Location.
3. **printRegisters:** Prints the values stored in all the Registers at the end of the Program.

3.3 Instruction Read and Write Memory Unit

Memory is byte addressable. This means each memory address holds one byte of information. To store a word, four bytes are required which use four memory addresses.
In General Processor Implementations, Memory unit has reserved sections for various units:
The text (or code) section is where the machine language is stored. The data

section is where the initialized data is stored. The uninitialized data section is where declared variables that have not been provided an initial value are stored.The heap is where dynamically allocated data will be stored (if requested). The stack starts in high memory and grows downward.
Our Implementation of Memory has sections reserved only for Instructions and data variable locations. We've partitioned out Memory unit into two halves, where first half is reserved for Instructions storage and second half for data variable storage.

Variables associated with the MemoryUnit:

1. **Memory-Array** of size 2^{20} storing the values holded by the corresponding Registers.
2. **Pointer to current Instruction Id and current Variable Id**: currInstrId stores the pointer to the current Instruction being executed and currVarId stores the pointer to the current Variable address.
3. **Map addofLabels {String : Int}** which maps the Labels with their corresponding address in the MemoryArray.
4. **Map addofVars {String : Int}** which maps the variables with their corresponding address in the MemoryArray.
5. **Map addofVars {Int : Int}** which maps the variables with the value it holds.

Methods associated with the MemoryUnit:

1. **storeInstr** takes in an instruction or label as input in the form of a string and stores it in the memoryArray. In case of label, rather than storing it in the memoryArray, it is stored in the map addofLabels with the value as address of the next instruction in it.
2. **getAddOfLabel** takes in a label as an input in the form of the string and returns the address of the Instruction succeeding it.
3. **getCurrInstr** return the address corresponding to the current Instruction being executed.
4. **setData / setDataAdd** takes in variable name or address of variable as input along the value to be written to it.
5. **getData / getDataAdd** takes in variable name or address of variable as returns the value corresponding to it's address.

3.4 Control Unit

The control unit is implemented by the function **processInstructions** which takes in vector of inputfile instructions, Register File and MemoryUnit as Input.

The control Unit has several branches associated with it depending on the current instruction. The methods defined for RegisterFile are used to stored the computed values in the registers or reading values from the registers.
The methods defined in the Memory Unit are used to load or save entries from / to the memory.
The program counter is incremented at each iteration or shifted to the targeted branch address after execution of each instruction.

4 Exception Handling

Exception 1) when the number of Instructions executed or the number of data variables stored in the memory exceeds 2^{19}.

Exception 2) If any instruction not in the domain set of executable instructions is provided.

Exception 3) If the syntax for allowed instructions do not match.
Ex. add $t1, $t2 is a bad Input

Exception 4) Incorrect register name in the Input instructions.

5 Output

The Output to the console shows:

1) At each execution, The current instruction being executed along with the states of all 32 Registers are displayed to the console.

2) After all the instructions are executed, the content of all registers, the relevant statistics such as the number of clock cycles and the number of times each instruction was executed are displayed to the console.