***COMPUTER ORGANIZATION AND ARCHITECTURE***

***PROJECT***

*#Coding Warriors#*

*AIM:* Develop a simulator on the lines of QtSPIM

Our Simulator supports MIPS (Microprocessor without Interlocked Pipelined Stages) instructions and can read in an assembly file, execute the instructions, and in the end display the contents of the registers, and the memory.

We choose C++ programming language for developing this simulator.

The MIPS instruction-set architecture:

• It is a load-store architecture that uses general-purpose registers.

• It has only two addressing modes, displacement and immediate, but can synthesize other important modes from them.

• It supports 8-, 16-, 32-, and 64-bit integers, and 32- and 64- bit IEEE 754 floating-point numbers.

• It has an orthogonal set of instructions to manipulate these data types.

• It has separate comparison and branching instructions. (This is an example of making the common case fast.)

MIPS has thirty-two 64-bit general-purpose registers, named R0, R1, … , R31. R0 always contains 0 (loading it with another value has no effect). It has 32 floating-point registers, which can hold either single precision (32-bit) or double-precision (64-bit) values.

*MIPS INSTRUCTIONS:*

* Arithmetic Instructions

|  |  |  |
| --- | --- | --- |
| **INSTRUCTION** | **SYNTAX** | **MEANING** |
| ADD (Addition) | ADD R1, R2, R3 | R1 = R2 + R3 |
| ADDI (Add immediate) | ADDI R1, R2, 100 | R1 = R2 + 100 |
| SUB (Subtract) | SUB R1, R2, R3 | R1 = R2 - R3 |
| ADDU (Add unsigned) | ADDU R1, R2, R3 | R1 = R2 + R3 |
| ADDIU (Add immediate unsigned) | ADDIU R1, R2, 100 | R1 = R2 + 100 |

* Load/Store Instructions

|  |  |  |
| --- | --- | --- |
| **INSTRUCTION** | **SYNTAX** | **MEANING** |
| LW (Load Word) | LW R1, 100(R2) | R1←Mem [R2+100] |
| SW (Store Word) | SW R1, 100(R2) | Mem[R2+100] ←R1 |

* Conditional/Branching Instructions

|  |  |  |
| --- | --- | --- |
| **INSTRUCTION** | **SYNTAX** | **MEANING** |
| BEQ (Branch on equal) | BEQ R1, R2, 100 | if (R1==R2) goto PC+4+100 |
| BNE (Branch on not equal) | BNE R1, R2, 100 | if (R1!=R2) goto PC+4+100 |
| SLT (Set on less than) | SLT R1, R2, R3 | if (R2<R3) R1←1; else R1 =0 |

* Unconditional Jump Instructions

|  |  |  |
| --- | --- | --- |
| **INSTRUCTION** | **SYNTAX** | **MEANING** |
| J (Jump) | J 100 | Goto 100 |
| JR (Jump Register) | JR R7 | Goto R7 |
| JAL (Jump and Link) | JAL 100 | R31←PC+4; goto 100 |
| JALR (Jump and Link Register) | JALR R7 | R31←PC+4; PC←R8 |

*Function Implementations:*

These are the function and instructions we implemented in phase 1 of project

* Memory Array function
  + This function is declared globally
  + This collects the argument indexes from parser function and stores the values in arrays.
  + This is not declared dynamically.
* Register function
* The register functions store all the values of register and prints at the end.
* PARSER function:
  + The Parse function enables the user to parse the data in one field in the source file and write the "parts" of the data to a field or fields in the target file.
  + We used parser function to read the file line by line from main function.
  + We described pc counter and pin, the ‘pc counter’ counts the total no. of lines in the given file and ‘pin’ is counts the words in a particular line.
  + If the word is equal to a particular instruction the parser sends the line and parameters to that particular function.
  + The before sending as arguments/parameters to the function the string the converted to int/double.
  + This reads and store indexes value.
* ADD function
  + The argument indexes are loaded from parser function to Add function.
  + The add function performs the substract of last two parameters and the value is stored in first parameter.
  + ADD R1, R2, R3 [R1 = R2 + R3]
* ADDI function
  + The argument indexes are loaded from parser function to Addi function.
  + The addi function performs the addition of last two parameters and the value is stored in first parameter.
  + ADD R1, R2, 100 [R1 = R2 + 100]
* BEQ function
  + The argument indexes are loaded from parser function to Beq function.
  + The beq function checks the equality of first two arguments and for last parameter jump function is called(if the arguments are equal).
  + BEQ R1, R2, 100 [if (R1!=R2) goto PC+4+100]
* SLT function
  + The argument indexes are loaded from parser function to Slt function.
  + The slt function compares the last two arguments and 0 or 1 is stored in first argument.
  + SLT R1, R2, R3 [if (R2<R3) R1←1; else R1 =0]
* JUMP function
  + The address is sent to Jump function by the parser.
  + The jump function goes to the given address argument in the function.
  + J 100 [goto 100]
* LOAD function
  + The argument indexes are loaded from parser function to Load Word function.
  + The last parameter index value is loaded to first parameter index value.
  + LW R1, R2 [R1 ← R2]
* STORE function
  + The argument indexes are loaded from parser function to Load Word function.
  + The last parameter index value is stored in first parameter index value.
  + SW R1, R2 [R1 ← R2]
* SUB function
  + The arguments are loaded from parser function to Sub function.
  + The sub function performs the addition of last two parameters and the value is stored in first parameter.
  + SUB R1, R2, R3 [R1 = R2 - R3]