MIPS INSTRUCTION SET ARCHITECTURE

2019103601
GUNNALA HEMA

Brief info about MIPS: MIPS stands for Microprocessor without Interlocked Pipeline Stages. Pipelining is a method of executing different stages of multiple instructions in the same clock cycle. In general, a classic pipeline instruction scheduler consists of five pipeline stages: Instruction fetch, Instruction decode, Execute, Memory access, Register write back. This cycle of stages repeats. Multiple pipelines in a processor allow multiple instructions to be executed at the same time. However multiple pipelines can cause:

1. Data hazards that occur when an instruction, scheduled blindly, would attempt to use data before the data is available in the register file.
2. Control hazards that are caused by conditional and unconditional branching, which can disrupt the flow of instructions.
3. Structural hazards, also referred to as resource hazards, which arise when two or more instructions need the same resource, thus causing a portion of the pipeline instructions to execute in series rather than parallel.

MIPS handles such hazards using delay instruction slots in the software, as opposed to interlocking pipelines which introduce delay slots using the hardware. Other resolution methods include feed forwarding and out-of-order execution in the case of data hazards, and branch prediction in the case of control hazards. MIPS is a load/store architecture, meaning that all operations are done by loading data from the main memory to the registers, performing operations on this loaded data in the registers, and then storing the result in the main memory. MIPS is also a reduced instruction set computer (RISC) architecture, with a small yet highly optimized instruction set, large number of registers in its implementations, and a highly regular instruction pipeline, thus ensuring a low clock cycle count per instruction.
Some of the instructions in the MIPS instruction set have been discussed below:

Arithmetic operations: The general syntax is:
Operation destination_operand,source_operand_1,source_operand_2 except the multiply and divide operations.
Add – adds values in source operand registers and places sum value in destination operand register.
add $s1, $s2, $s3 is equivalent to $s1=$s2+$s3

Subtract **- sub $s1, $s2, $s3** is equivalent to $s1=$s2-$s3

Add immediate – adds a constant value with the source operand. **addi $s1, $s2, 20** is equivalent to $s1=$s2+20
Add unsigned – treats the register values as unsigned integers, not 2's complement integers. This operation does not throw an overflow warning if the result exceeds the 31-bit signed numerical capacity of the register.
addu $s1, $s2, $s3 - $s1=$s2+$s3 (unsigned addition)

Subtract unsigned – **subu $s1, $s2, $s3 -** $s1=$s2-$s3

Add immediate unsigned – **addiu $s1, $s2, 100 -** $s1=$s2+100 (unsigned addition)

Multiply (without overflow) - multiplies two 32-bit register operands to get a 64-bit result and stores the least significant 32 bits of the result in a 32-bit register. Does not check for overflow if result exceeds 32 bits. **mul $s1, $s2, $s3** - $s1=$s2 * $s3

Multiply – multiplies two 32-bit register operands to get a 64-bit result, stores the least significant 32 bits in the reserved LO register, and the most significant bits in the reserved HI register. **mult $s4, $s5** – HI: LO = $s4 * $s5

Divide - divides one 32-bit register operand by another, stores the 32-bit quotient in the HI register, and the 32-bit remainder in the LO register. **div $s4, $s5** - HI = $s4/$s5, LO = $s4 % $s5

Logical operations: same syntax as arithmetic operations
And - performs bitwise AND on source operands and stores result in destination operand. and **$s1, $s2, $s3** - $s1 = $s2 & $s3

Or – performs bitwise OR on source operands and stores result in destination operand. or **$s1, $s2, $s3** - $s1 = $s2 | $s3

And immediate – bitwise AND with immediate value. **andi $s1, $s2, 100** - $s1 = $s2 & 100

Or immediate – bitwise OR with immediate value. **ori $s1, $s2, 100** - $s1 = $s2 | 100

Shift left logical – perform bitwise left shift on source operand by n bits, and store result in destination operand. **sll $s1, $s2, 3** - $s1 = $s2<<3

Shift right logical - perform bitwise right shift on source operand by n bits, and store result in destination operand. **srl $s1, $s2, 3** - $s1 = $s2>>3

Data transfer operations: operates on two operands

Load word – copy 4-byte word from memory address specified in source operand register, while accounting for any given address offset, (in bytes) to destination operand register. lw **$s1, 100($s2)** - $s1 = Memory ($s2+100)

Store word – copy 4-byte word from register to memory. **sw $s1, 100($s2)** - Memory($s2+100) = $s1

Load upper immediate – load constant value into upper 16 bits of a register, setting lower 16 bits to zeroes. **lui $s1, 100** - $s1 = 100 * 2 power 16

Load address – A pseudo instruction provided by the assembler for the programmer's convenience. Loads the computed address of the label, and not its contents, to the register specified. la $s1, **label - $s1** = Address of label

Load immediate - A pseudo instruction provided by the assembler for the programmer's convenience. Loads a constant value into register. **li $s1, 100** - $s1 = 100

Move from HI – copies contents of the reserved HI register into specified register. **mfhi $s1 -** $s1 = HI

Move from LO – copies contents of the reserved LO register into specified register**. mflo $s1** - $s1 = LO

Move - A pseudo instruction provided by the assembler for the programmer's convenience. Copies contents of one register to another. **move $s1, $s2** - $s1 = $s2