MIPS

WANG Xiyu

September 16, 2024

Contents

1 Instruction Set Architecture

Instruction Set Architecture includes everything programers need to know to make the machine code work correctly, and allows computer designers to talk about functions independently from the hardware that performs them. This abstraction allows many implementations of varyin cost and performance that can run identical software. Basically an API between software and hardware.

2 Machine Code vs Assembly Language

3 Walkthrough

The major 2 components of a computer are the processor and memory(RAM), which are connected by bus, which is a data connector. The code and data reside in memory, which are transferred into the processor duting execution. The Processor is much faster than memory access, therefore we have load-store model that limits the memory operations and relies on registers for storage during execution.

3.1 Memory Instruction

To move data from memory to registers, and the other way round later.

Variable mapping

mapping of variables to registers.

```
r0 <- load i
```

3.2 Reg-to-Reg Arithmetic, and other calculation instructions

Arithmetic opeartions are be done directly on registers only, therefore very fast.

3.3 Control flow instructions

Instructions that changes the order of sequential execution.

4 General Purpose Register

There are 32 general purpose registers in a mips processor, from number 0 to number 31, each with a name and corresponding functionality.

Reg. Number	**Reg. Name**	**Alias**	**Remarks**
0	$zero	constant zero	Always holds the constant value 0. Cannot be modified.
1	$at	assembler temporary	Reserved for use by the assembler. Not typically used in normal code.
2-3	$v0-$v1	values for results	Used to store function return values and results of expressions.
4-7	$a0-$a3	arguments	Used to pass the first four arguments to functions. Additional arguments are passed via the stack.
8-15	$t0-$t7	temporaries	Temporary registers not preserved across function calls. Free to use within functions.
16-23	$s0-$s7	saved temporaries	Temporary registers, but values must be preserved across function calls (callee-saved).
24-25	$t8-$t9	more temporaries	Additional temporary registers, not preserved across function calls.
26-27	$k0-$k1	kernel reserved	Reserved for the operating system kernel. Do not use in user-level code.
28	$gp	global pointer	Points to the middle of the global/static data segment.
29	$sp	stack pointer	Points to the top of the current stack frame. Grows downward.
30	$fp	frame pointer	Used as a frame pointer for function calls. Can also be $s8 in some conventions.
31	$ra	return address	Stores the return address for function calls. Used by the ‘jal‘ instruction.

Table 1: MIPS General Purpose Registers with Remarks

5 MIPS Assembly Language

5.1 General Instruction Syntax

There are 3 types of MIPS instruction, R type (register), I type (immediate), and J type (jump). All instructions have 32 bits, which fit into 4 bytes blocks in the memory, or the capacity of one register.

Format Type	Field Name	Field Size (bits)	Description
R-Type	Opcode	6	The operation code, always '000000' for R-type instructions.
	rs	5	The first source register.
	rt	5	The second source register.
	rd	5	The destination register where the result is stored.
	shamt	5	The shift amount, used only for shift instructions. Typically zero for other instructions.
	funct	6	Function field that determines the specific operation (e.g., add, sub).
I-Type	Opcode	6	The operation code identifying the instruction type.
	rs	5	The source register.
	rt	5	The destination register.
	Immediate	16	A 16-bit immediate value or address offset.
J-Type	Opcode	6	The operation code identifying the instruction type.
	Address	26	The target address for jump instructions.

Table 2: R, I, and J Type Instruction Formats in MIPS

Notes: The order of registers in MIPS instruction lines are different from the order in the instruction format.
In 32 bits format the register order of R type instructions is rs, rt, rd; i

Instruction	Format	Operation	Description
add	R	$d = s + t$	Adds two registers and stores the result in a register.
addi	I	$t = s + imm$	Adds a register and an immediate value and stores the result in a register.
sub	R	$d = s - t$	Subtracts one register from another and stores the result in a register.
mult	R	$LO = s \times t$	Multiplies two registers and stores the result in special registers LO (low) and HI (high).
div	R	$LO = s/t$	Divides one register by another and stores the quotient in LO and the remainder in HI.
and	R	$d = s \wedge t$	Performs a bitwise AND of two registers and stores the result in a register.
andi	I	$t = s \wedge imm$	Performs a bitwise AND of a register and an immediate value.

Instruction	**Format**	**Operation**	**Description**
or	R	$d = s \vee t$	Performs a bitwise OR of two registers and stores the result in a register.
ori	I	$t = s \vee imm$	Performs a bitwise OR of a register and an immediate value.
xor	R	$d = s \oplus t$	Performs a bitwise XOR of two registers and stores the result in a register.
xori	I	$t = s \oplus imm$	Performs a bitwise XOR of a register and an immediate value.
nor	R	$d = \neg(s \vee t)$	Performs a bitwise NOR (NOT OR) of two registers and stores the result in a register.
sll	R	$d = t << shamt$	Shifts a register value left by the shift amount (shamt) and stores the result.
srl	R	$d = t >> shamt$	Shifts a register value right by the shift amount (shamt) and stores the result.
sra	R	$d = t >> shamt$ (arithmetic)	Shifts a register value right by the shift amount (shamt) with sign extension.
slt	R	$d = (s < t)$	Sets the destination register to 1 if the first source register is less than the second; otherwise, sets it to 0.
slti	I	$t = (s < imm)$	Sets the destination register to 1 if the source register is less than the immediate value; otherwise, sets it to 0.
lw	I	$t = \text{Mem}[s + \text{offset}]$	Loads a word from memory into a register.
sw	I	$\text{Mem}[s + \text{offset}] = t$	Stores a word from a register into memory.
lb	I	$t = \text{Mem}[s + \text{offset}]$	Loads a byte from memory into a register (sign-extended).
lbu	I	$t = \text{Mem}[s + \text{offset}]$	Loads a byte from memory into a register (zero-extended).
sb	I	$\text{Mem}[s + \text{offset}] = t$	Stores a byte from a register into memory.
lh	I	$t = \text{Mem}[s + \text{offset}]$	Loads a halfword from memory into a register (sign-extended).

Instruction	Format	Operation	Description
lhu	I	$t = \text{Mem}[s + \text{offset}]$	Loads a halfword from memory into a register (zero-extended).
sh	I	$\text{Mem}[s + \text{offset}] = t$	Stores a halfword from a register into memory.
beq	I	`if` $s = t$ `then` $PC = PC + 4 + (imm \times 4)$	Branches to a label if two registers are equal.
bne	I	`if` $s \neq t$ `then` $PC = PC + 4 + (imm \times 4)$	Branches to a label if two registers are not equal.
j	J	$PC = (PC\&0xF0000000)\|(\text{address} \times 4)$	Jumps to a specified address.
jr	R	$PC = s$	Jumps to the address contained in a register.
jal	J	$RA = PC + 4; PC = (PC\&0xF0000000)\|(\text{address} \times 4)$	Jumps to a specified address and saves the return address in the link register.
mfhi	R	$d = HI$	Moves the contents of the HI register to a general-purpose register.
mflo	R	$d = LO$	Moves the contents of the LO register to a general-purpose register.
mthi	R	$HI = s$	Moves the contents of a general-purpose register to the HI register.
mtlo	R	$LO = s$	Moves the contents of a general-purpose register to the LO register.
nop	R	No operation	Does nothing; often used for delay slots.

Writing MIPS instructions and MIPS to C translation

General strategy:

1.

8 The Processor - Datapath

8.1 Datapath and Control

Data path consists of components that process data, which performs the arithmetic, logical and memory operations. Control tells the datapath, memory and I/O devices what to do according to program instructions.

8.2 MIPS Processor implementation

Implement a subset of the core MIPS ISA

8.2.1 Arithmatic and Logical operations

- add
- sub
- and
- or
- addi
- slt

Note: andi and ori are not supported with current processor design as we always do sign extension on immediate value.

8.2.2 Data transfer instructions

- lw
- sw

8.2.3 Branches

- beq
- bne

8.3 Instruction Execution Cycle

The execution on ONE instruction consists of 5 stages:

8.3.1 Fetch stage

- Get instruction from memory
- Address is in the Program Counter Register

8.3.2 Decode stage

- Find out the operation required

8.3.3 Operand fetch stage

- Get operand(s) needed for operation

8.3.4 Execute stage

- Perform required operation

8.3.5 Register write stage

- Store the result of operation in a register

Example

Instruction	**Add** (`add $rd, $rs, $rt`)	**Load Word** (`lw $rt, ofst($rs)`)	**Branch on Equal** (`beq $rs, $rt, ofst`)
Fetch	Standard fetch	Standard fetch	Standard fetch
Decode	Operand fetch	Operand fetch	Operand fetch
Operand Fetch	Read [`$rs`] as opr1 Read [`$rt`] as opr2	Read [`$rs`] as opr1 Use ofst as opr2	Read [`$rs`] as opr1 Read [`$rt`] as opr2
Execute	Result = opr1 + opr2	MemAddr = opr1 + opr2	Taken = (opr1 == opr2)? Target = (PC+4) + ofst × 4
Result Write	Result stored in `$rd`	Memory data stored in `$rt`	if (Taken) PC = Target

Table 4: Stages of instruction execution for add, lw, and beq

8.4 Build a MIPss Processor

8.4.1 Fetch stage

8.4.2 Decode stage

8.4.3 ALU stage

8.4.4 Memory stage

8.4.5 Register write stage