Lecture 4 - MIPS instruction set

Gidon Rosalki

2025-04-20

We have so far covered the physics, devices, circuits, and gates and registers of the abstraction layers of modern systems. This week we will discuss Instruction Set Architecture (ISA) (and next week we will discuss microarchitecutre, which sits just below the ISA in the stack). There are a few different ISAs, such as intel, MIPS, ARM, and RISC-V. We will discuss MIPS, since it is one of the simplest ones. RISC-V is the next big one, and is mainly based on MIPS, and even designed by the same people. It is however more complex, but should you need, transitioning there from MIPS is fairly easy.

1 ISA, architecture, and micro-architecture

1.1 Concepts

The **architecture** and **ISA** define the programmer interface for a CPU. This includes the set of instructions, their operations, registers, and so on. Examples of this include MIPS, ARMv8, x86, RISC-V. This is in the end the language in which our code needs to be in order to run on a CPU. When writing code in Python, C, Java, and so on, it is all eventually compiled down to one of these languages. The **Micro-Architecture** (uArch/μArch) defines the implementation of an ISA/Architecture. These may be single cycle (next week), pipeline, out of order, caches, and so on. All of these may have the same ISA, but with a different uArch. Intel and AMD implement the x86 architecture, Arm, Qualcomm, and Apple all use the ARM architecture. Our focus today is the MIPS Architecture, and we will discuss in the next lectures the implementation and uArch.

1.2 CPU model

The CPU is a state machine which is connected to memory. The CPU is the processor, and the state includes registers, which are in turn manipulated by instructions. A program is a set of instructions, which take place on the CPU. Memory is an array, which is indexed by an address, each address points to a byte of data. Programs are indexed in memory by the program counter (PC). The program execution of a CPU is as follows:

1. Read the next instruction from memory, addressed by the PC register
2. Advance the PC: PC = PC + instruction size
3. Preform the instruction - change the CPU state
4. Back to step 1

The **programs** are written by programmers in high level languages, such as C, Java, Python, and so on. The CPU cannot run such programs, so instead they are compiled by the **compiler** from C into assembly (Python and Java are interpreted and run on a virtual machine program). An **assembler** translates the assembly into binary. The **linker** then creates a program by connecting the code file (assembler output), and data into an executable format that can be loaded by the OS. For Python/Java and the like, the files are text files written in the respective languages, which are then loaded by the interpreter (often C/C++), that was compiled into an executable binary, and then the CPU executes the interpreter binary, which reads the text file, and executes it.

```
int foo(int x) {
    int y;
    y = x + 17;
    if (x > 50) {
        y = 0;
    }
    return y;
}
```

This C would be translated into something like

```
foo:
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -20(%rbp)
    movl    -20(%rbp), %eax
    addl    $17, %eax
    movl    %eax, -4(%rbp)
    cmpl    $50, -20(%rbp)
    jle     .L2
    movl    $0, -4(%rbp)
.L2:
    movl    -4(%rbp), %eax
    popq    %rbp
    ret
```

This would then be translated into a binary like

```
ABD67A879BFF898D
8275395DFA56CC78
92AD879D66760976
```

1.3 Assembly

Assembly is the lowest level of programming languages. These are instruction that the processor can execute, ie are written in the CPU ISA. The assembly instructions can operate on registers, and/or memory, and are generally only primitive operations such as addition, subtraction, memory read/write and jumps. In C we might write A = A + B, which translates into the asm `ADD $7, $2, $3` which is in something we recognise better adding together the contents of registers 2 and 3, and storing the result in 7. This is then translated into a direct machine code instruction such as `100110010100`.

2 MIPS

MIPS was developed in Stanford University by Hennessey and Patterson. It implements the RISC (Reduced Instruction Set Computer) architecture, for ease of development.

2.1 CISC vs RISC

CISC is Complex Instruction Set Computer, where the ideas is a high level machine language. Examples include x86, and DSP processors. The goal is to reduce the instruction count of the final program. Each instruction is usually broken into simple operations (micro operations) by the hardware, and it is thus easy to migrate across generations. CISC is usually characterised by a rich set of instructions, which have many instruction formats, addressing modes, and so on, compound instructions which can carry out multiple operations, operations between registers or directly on memory, and mostly variable length instructions.
RISC is Reduced Instruction Set Computer. The concept is to keep the instruction set small and simple, in order to simplify the hardware. The compiler translates compound instructions into simple instructions. Examples include **MIPS**, ARM, and RISC-V. The goal is to move the complexity to the compiler. Since all operations on the CPU are simple, the commands need not be broken down further by the hardware, and instead the complexity of this is moved to the compiler. It is characterised by small instructions sets with few formats and few addressing modes, simple instructions, arithmetic / logic operations may **only** be performed between registers and **not** on memory, memory reverberance explicitly for load/store machine, and mostly fixed length instructions. Lately the borders between CISC nad RISC are blurring, some RISC CPUs also have a rich instruction set (including things like bit manipulation), and most CISC CPUs are actually load-store machines internally. The hardware translates CISC operations into RISC like operations on the fly.

2.2 Registers, memory

There are 32 registers, \$0 to \$31. Each registers holds a 32 bit value, called a word (or 4 bytes). Instructions will operate on registers. There are only 32 since it is simple for implementation (MUXes, decoders), and faster access. 32 registers needs 5 bits in the instruction encoding for each register, so more registers is more expensive in terms of the number of bits, but too few impacts performance, so this is a balance. The disadvantages of 32 registers is that more registers are better for software (compilers). Some of these registers have dedicated purposes for which the hardware is optimised. For example, \$0 stores the constant value 0, \$31 is used for return address by the hardware. Use is defined by a software

	C-Language	CISC	RISC

Example	Copy array contents from a to b:
	`int a[100], b[100];`
	`for (int i = 0; i < 100; i++)`
	`    b[i] = a[i];`
		AX, BX, CX registers, copy CX bytes from memory pointed by AX to BX
		REP MOVSB BX, AX
			$1, $2, $3, $4 are registers, this code copies $3 bytes from memory pointed by $1 to $2:
			`loop: lw   $4, 0($1)`
			`      sw   $4, 0($2)`
			`      addi $3, $3, −4`
			`      addi $1, $1, 4`
			`      addi $2, $2, 4`
			`      bne  $3, $0, loop`

Advantages		Less instructions, hardware can perform optimisations under the hood	Simpler hardware implementation
Disadvantages		Complex hardware to implement the instructions	More instructions (code size, extra memory accesses, change over generations)

Table 1: CISC vs RISC

convention. Names (mnemonics) have been created for various registers to enable this. These define calling convention, such as the stack pointer. Some non general purpose registers:

- SP - stack pointer
- EPC - related to exceptions

Name	Register number	Usage	Preserved on call?
$zero	0	The constant value 0	n.a.
$v0-$v1	2 - 3	Values for results and expression evaluation	No
$a0 - $a3	4 - 7	Arguments	No
$t0 - $t7	8 - 15	Temporaries	No
$s0 - $s7	16 - 23	Saved	Yes
$t8 - $t9	24 - 25	More temporaries	Mo
$gp	28	Global pointer	Yes
$sp	29	Stack pointer	Yes
$fp	30	Frame pointer	Yes
$ra	31	Return address	Yes

Table 2: MIPS registers

2.3 Instructions

MIPS instructions are always 32 bits bits long (or 4 bytes, which is a word). The operation and operands are encoded in these 32 bits, and assembly instructions are the textual representation of the binary encoding. Each instruction executes one simple operation, such as arithmetic / logic, memory operations (such as load or write), and jumps/conditional instructions. These instructions are stored in memory. So each instruction has an address in memory at which it is stored, and each instruction starts at an address that divides by 4 (last 2 bits are 00). Since they are all of length 32 bits, it means that each instruction fits perfectly inside a register (yippee! I wonder why this might be?).

2.3.1 Arithmetic and logic

The operation, and target registers, are all stored in the 32 bits of instruction.

Addition using an Immediate: ADDI

Immediate values are constants that are encoded in the instruction itself (rather than a register). Some instructions also have encodings that use immediate values. Consider for example `addi $s1, $s2, 4 # s1 = s2 + 4`. Sign

Instruction	Operation	Description
add $d, $s, $t	Add: $d = $s + $t	Adds two registers, and stores the result in a register
sub $d, $s, $t	Sub: $d = $s - $t	Subtracts two registers, and stores the result in a register

Table 3: Add/Subtract

Instruction	Operation	Description
addi $t, $s, i	Add Immediate: $t = $s + imm	Adds a register and a sign-extended immediate value, and stores the result in a register

Table 4: Add immediate

extend: Conversion ofa 16bit 2's complement integer to 32 bit 2's complement integer. We need this since the operation operates on 32 bit values, but immediate is encoded into the 32 bit instruction, and hence cannot be 32 bits wide (in MIPS it is usually 16 bits wide). We do not need a subi, we can instead use addi with a negative value: `addi $s1, $s2, −4 # s1 = s2 − 4`

Instruction	Operation	Description
add $d, $s, $t	Add: $d = $s + $t	Adds two registers, and stores the result in a register
sub $d, $s, $t	Sub: $d = $s - $t	Subtracts two registers, and stores the result in a register
addi $t, $s, i	Add Immediate: $t = $s + imm	Adds a register and a sign-extended immediate value, and stores the result in a register

Table 5: Arithmetic instructions

For logic instruction, operations are done bitwise. A logical operation is done between each bit in SRC1, and the corresponding bit in SRC2, the result put into the corresponding bit in DST. All 32 operations occur in parallel.

Instruction	Operation	Description
and $d, $s, $t	Bitwise and: $d = $s & $t	Bitwise ands two registers, and stores the result in a register
andi $t, $s, i	Bitwise and immediate: $t = $s & imm	Bitwise ands a register, and an immediate value, and stores the result in a register
nor $d, $s, $t	Bitwise nor: $d = ($s | $t)	Bitwise nors two registers, and stores the result in a register
or $d, $s, $t	Bitwise nor: $d = ($s | $t)	Bitwise ors two registers, and stores the result in a register
ori $t, $s, i	Bitwise nor immediate: $t = ($s | imm)	Bitwise ors a register, and an immediate value, and stores the result in a register
xor $d, $s, $t	Bitwise nor: $d = ($s $t)	Bitwise exclusive ors two registers, and stores the result in a register
xori $t, $s, i	Bitwise nor immediate: $t = ($s îmm)	Bitwise exclusive ors a register, and an immediate value, and stores the result in a register

Table 6: Logical instructions

There are also shifting instructions. These are operations done on bits in the register. There is no arithmetic or logical operation done on the bits, but there is a meaning to the shift. A "barrel shifter" is a unit that can perform a shift by n steps. With integers, shift right by 1 bit divides a value by 2, and shift left 1 bit multiplies by 2. When doing the same with x, it is instead buy 2^x.

2.3.2 Memory, load and store

CPU registers hold only 31 values ($1 - $31), and accessing them is very fast. When software needs more values, it uses memory instead. The memory is an array, indexed by address. Each address points to a byte of data, and access is slower than registers. The memory contains data, as well as instructions (the program). Note, these addresses must be divisible by 4.

Instruction	Operation	Description
sll $d, $t, a	Shift left logical $d = $t « a	Shifts a register value left by the shift amount listed in the instruction, and places the result in a second register. Zeroes are shifted in.
sllv $d, $t, $s	Shift left logical $d = $t « $s	Shifts a register value left by the shift amount listed in the second register, and places the result in a third register. Zeroes are shifted in.
sra $d, $t, a	Shift right arithmetic $d = $t » a	Shifts a register value right by the shift amount listed in the instruction, and places the result in a second register. The sign bit is shifted in.
srav $d, $t, $s	Shift left arithmetic $d = $t » $s	Shifts a register value left by the shift amount listed in the second register, and places the result in a third register. The sign bit is shifted in.
srl $d, $t, a	Shift right logical $d = $t »> a	Shifts a register value right by the shift amount listed in the instruction, and places the result in a second register. Zeroes are shifted in.
srlv $d, $t, $s	Shift left logical $d = $t »> $s	Shifts a register value left by the shift amount listed in the second register, and places the result in a third register. Zeroes are shifted in.

Table 7: Shifting instructions

Instruction	Operation	Description
lw $t, i($s)	Load word $t = MEM[$s + imm]	A word is loaded into the register from the specified memory address.
sw $t, i($s)	Load word MEM[$s + imm] + $t	The contents of $t are stored at the specified memory address.

Table 8: Memory instructions

There are two main architectures for memory organisation. Von Neumann, and Harvard.

2.3.3 Von-Neumann Architecture

One memory map for both program and data. Reading returns bits, meaning bits depend on operation. Writing mostly data, but can also write a program. Physical partitioning of memory is possible, but everything is mapped to one space. This has the pros of a unified memory view, and simple debug and program loading, but the cons of feth and load/store operations operate on the same resource (less of a problem in the modern day). Examples include x86, MIPS, and ARM

2.3.4 Harvard Architecture

The memory map for code is separated from teh data in this architecture. Load from/store to data memory, and instructions fetch from program memory. Special store to program memory, or a ”mapping function” for handling debug and program loading. This has the pros of no conflicting access, resulting in better performance, but the cons of nonflexible memory allocation, and updating program memory is harder. The is mainly used by DSP processors.

2.3.5 Jumps and conditionals

Instruction	Operation	Description
j label	Jump: Jump to label	Jumps to the calculated address
beq $s, $t, label	Branch Equal: if ($s == $t) jump to label	Conditional branch - if the two registers are equal then jump to label
bne $s, $t, label	Branch Not Equal: if ($s != $t) jump to label	Conditional branch - if the two registers are not equal then jump to label

Table 9: Jump instructions

Label is a location in the program, we use a name that is later converted into an address as an offset or absolute location.

C:

```
if (i == j)
    f = g + h;
else
    f = g - h;
```

ASM:

```
bne $s0, $s1, not_eq # s0 is i, s1 is j
add $v0, $s2, $s4    # f = g + h
j   cont
```

```
not_eq:
    sub $v0, $s2, $s4    # f = g - h
cont:
    ...
```

Instruction	Operation	Description
slt $d, $s, $t	Set less than: $d = ($s < $t) ? 1 : 0	If $s is less than $t, $d is set to one. Otherwise it is set to 0.
slti $t, $s, i	Set less than immediate: $t = ($s < imm) ? 1 : 0	If $s is less than immediate, $t is set to one. Otherwise it is set to 0.

Table 10: Conditional instructions

2.3.6 Calling conventions

We have the following terminology:

- Caller - A function that is making the call
- Callee - The function that was called
- Parameters - Values passed by the caller to the callee
- Result(s) - Value(s) passed from the callee back to the caller
- Return address - The instruction following the coll instruction

The callee has no idea what the caller is doing, and only expects the parameters to be set correctly, and the result to be understood.
The stack is a memory area that a program can use. It is used to hold data locally for a function, and is managed in a way that allows nesting of functions, and tracking back. It is of the style of FILO, First In Last Out, and has the PUSH and POP commands, which push a value to the top, and pop a value off the top, of the stack respectively. All access are to the top of the stack, anything else is invalid.

If *f* is in the middle of execution, and it calls *g*, which registers can *g* use? If it uses all of them, then *f* needs to save everything before it calls *g*. If it uses none of them, then *g* will need to save all its changes, and restore before its return. In the end, *g* can use some.

- $t0 - $t9 are caller saved, and are the responsibility of *f* to restore
- $s0 - $s7 are callee saved, and are the responsibility of *g* to restore
- $a0 - $a3 and $v0 - $v1 are used to pass and return arguments, and are treated as caller saved
- $ra is overwritten by jal, treated as caller saved
- $sp is callee saved

When a register is **caller** saved, then the callee can change the values, and the caller should assume that the callee will change these values, and must save them before making a call if the values will be required later.
When a register is **callee** saved, then the callee should not change such values (or it should make sure it restores them before returning), and the caller can assume the values are not changed on return from the callee.

Values are saved on the stack. The stack grows *downwards*. $sp points to the top of the stack ie the last allocated word, and anything below $sp cannot be used. The push command first allocates, then stores, use can use any data on the stack any time based off an offset from $sp, and pop first loads, and then frees.
To push a value from $ra to stack:

```
addi $sp, $sp, -4
sw   $ra, ($sp)
```

and to pop a value from the stack to $ra

```
lw      $ra, ($sp)
addi    $sp, $sp, +4
```

Let us discuss the stack in more detail. It is a memory structure that is used to store data in a relative way. $SP register is the stack pointer, and it points to the top of the stack. Pushing adds elements to the stack, and popping takes from the stack. When accessing the stack we refer to elements based off the current stack pointer, and a known offset of the element from the sp. Before a function is called, we allocate space for parameters. When it is called, then it allocates space for the values that will need to be saved (think local variables inside the function). The stack pointer always points to the bottom of the stack (since the stack grows **downwards**), and then in use, whatever is pushed is also popped. Access is as described previous $sp + offset, and we can use temporary pointers for specific elements.

The stack is organised as follows. The stack passed parameters are always stored just before the top of stack (TOS). When compiling you calculate how much space is needed, and adjust the SP accordingly. This includes space for local variables, if required, space for saved registers, and space for parameters if calling another function. The stack is returned to how it was before a function return.

In order to call functions, we have some special instructions. jal is used for calling a function, and stores the return

Instruction	Operation	Description
jal label	Jump and link: $ra = nex PC; jump to label	Jumps ot the calculated address, and stores the return address in $31 (which is $ra)
jr $s	Jump register: jump to address in $s	Jump to the address contained in $s

Table 11: Function calling instructions

address into $ra before jumping, and jr $ra is used for returning from functions.

The calling convention defines the software use of registers:

- a0 - a3 are used to pass arguments to functions (or at least the first 4, if there are more than the caller stack is used for what remains).
- v0 and v1 are used as return values
- t0 - t10, a0 - a3 may be changed by the callee
- s0 - s7 must be either untouched or restored by the callee
- $ra - return address
- $sp - stack pointer, whatever is pushed there must also be popped.