

# Instruction Set Design

The MIPS instruction set will serve as the design example we will study in some detail. The MIPS processor is a good example of RISC processors that have dominated instruction set design since the mid 1980's. The intent of this class is not to do MIPS assembly language programming (covered in chapter 2 and 3), but to understand what instructions and what instruction formats are needed in a modern instruction set. The remainder of the course will investigate how the MIPS assembly language might be implemented in hardware.

## Memory Addressing

Memory address - a number representing the location of information stored in memory.

Byte addressable - the unit of information stored at each address is one byte (8 binary bits)

Most contemporary computers have byte addressable memory so that individual character codes can be addressed (ASCII character codes fit into one byte). As international character sets (UNICODE) become more common, it may not be necessary to have byte addressable memory.

Most information items take more than one byte to represent.

word - basic information element which usually matches the size of data that can be handled by the ALU. Modern processors have instruction codes that all fit into one word. Typical word sizes are 32 or 64 bits (4 or 8 bytes). The MIPS has a word size of 32 bits.

By convention, information items start at some address and extend to higher addresses.

- Big Endian - the starting address corresponds to the most significant (big end) part of the information. MIPS is Big Endian.
- Little Endian - the starting address corresponds to the least significant (little end) part of the information.

**Memory Addressing Modes.** The MIPS addressing modes are summarized on p. 100 and fig. 2.24, p. 101. They are a good example of the historically important addressing modes, that is, they are a good example of the addressing modes used most frequently on older machines. Some of the addressing modes on older machines are special cases of MIPS addressing modes.

Register indirect (sometimes called register deferred) mode is a special case of base-displacement with a displacement of 0).

Other addressing modes on older machines either are used infrequently or take too much time to execute (e. g. memory indirect takes 2 memory accesses => slow!).

**Memory Addressing Impact on Instruction Set Design.** Most modern processors limit the use of memory addressing modes as much as possible because:

1. Accessing memory is slow (the Von-Neuman bottleneck) since memory hardware is optimized for the maximum number of bits in the smallest area which is achieved by sacrificing access speed (take the VLSI course for more details).
2. There is not enough space in an instruction word for many memory operands. A memory operand requires encoding the address of the operand into the instruction.

Modern processors provide a large register file (32 or more registers typically) which is used for source and destination operands for almost all instructions. Only a few instructions, typically load and store, access memory directly with only a single operand in memory. This is often called the “load/store” architecture.

### Operations in Instruction Set

The operations in the MIPS instruction set are summarized on the inside front cover and discussed in more detail in section D.3, p. D-9 to D-16 (on the CD-ROM). Modern processors typically provide at least arithmetic/logical, data transfer, control, floating point and exception operations.

**Necessary Operations.** Detailed design of the hardware components is driven by choice of operations in instruction set. Question: How many instructions are necessary?

Answer: One is enough!

Example: subjump a, b, c

operation:  $a \leftarrow a - b$ , jump to c for the next instruction where a, b, c are memory addresses.

(a stored program computer is assumed so that instructions, in this case addresses, and data share the same memory).

Can show that a program, containing only subjump a, b, c instructions and data words, can be written which is equivalent to any normal instruction.

e.g. addition  
 logical operations  
 conditional branch

Only reasons for more than one instruction

1. programmer convenience  
 (reduces program development time)
2. performance improvement  
 (reduces program execution time)

Note: modern optimizing compilers remove reason #1 because general users program only

in HLL. Only the compiler writer need use the instruction set directly! (Even operating systems are written in HLL, e.g. C. was developed to write UNIX)

**Instruction Frequency.** Instructions should be chosen for ability to improve program execution performance. Performance depends most on instructions most frequently used. Typical instruction frequencies for the MIPS instruction set are given in Fig. 3.26, p.228.

Moral: only a small number of instructions types are executed most of the time!

Instructions can be grouped into the three broad categories in order of importance:

	MIPS int	MIPS FP
arithmetic and logic	43%	51%
data transfer (load/store)	37%	44%
control (branch/jump)	20%	4%

It usually is a shock to hardware designers that the ALU can be used less frequently than other parts of the processor.

In a performance driven design, we will implement the most frequently used instructions as efficiently as possible. The efficiency of less frequently used instructions can be sacrificed if necessary to improve efficiency of more important instructions.

**Conditional Branches.** Instruction frequency studies show that conditional branches are among the most frequently used instructions. There is continuing debate on how best to implement them.

There does seem to be agreement on using relative addressing for branches. This is because branch addresses are usually near the address of the branch instruction. Thus, it takes only a few bits of address to specify relative branch address as

$$(\text{relative branch address}) = (\text{branch address}) - (\text{branch instruction address})$$

### Optimizing Compiler

RISC hardware  $\Leftrightarrow$  optimizing compiler

Compiler “decides” how best to implement HLL construct in computer instruction set. Optimization can involve reordering instructions to make more efficient use of the hardware. It is best to limit the number decisions the compiler must make to get the best machine code.

Compilers generate best code when using simple address modes and have lots of registers to use (at least 16). How many registers should we have? The compiler wants as many as possible, but registers are very expensive since they require more than one data path to get information in and out (memory usually uses a single data path). As technology has pro-

gressed to allow additional registers, it has been found to be more cost effective to add on-chip memory rather than drastically increase the number of registers. These considerations usually limit the total register size to around the 32x32 bit example register file (32 registers each containing 32 bits) used in MIPS instruction set.

### RISC Design Goals

The hardware should be as streamlined as possible. It will be a later goal to keep all of the hardware as busy as possible as much of the time as possible.

The above observations bring about the following RISC instruction set design goals:

1. Few simple instructions that execute in as few clock cycles as possible
2. Most instructions should use register source and destination to avoid reference to (slow) memory (load-store architecture)
3. Provide lots of registers (so compiler may store temporary results without using slow memory)
4. Instructions should all be of the same length (to simplify fetching instructions from memory and decoding them)
5. Simple addressing formats capable of addressing entire memory space (no built in memory segmentation)

### MIPS Instruction Set Design

The following design decisions for the MIPS instruction set are representative of modern RISC design.

**Registers.** 32 registers each with 32 bits are provided as general purpose registers (GPR). The GPR's are designated as R0, ..., R31 where R0 always has a 0 stored in it regardless of what is written into it (the MIPS assembler refers to these as \$0, ..., \$31). The GPR's are used for integer arithmetic. The MIPS assembler enforces software conventions on the use of the GPR's by referring to them with special names shown on the inside front cover.

There are also 32 floating point registers (FPR) each with 32 bits. The FPR's are designated as F0, ..., F31 (the MIPS assembler refers to these as \$f0, ..., \$f31). The FPR's are used for floating point arithmetic.

There are also several other special registers. The most important of these is the program counter (PC) which is used to store the address of the instruction as it is fetched from memory. If we use one of our GPR registers as the PC, then we could use normal displacement addressing modes with the PC as the base register to achieve relative jumps. This would waste one of our registers and also allow programmers to accidentally change their PC contents with arithmetic/logic instructions! Modern instruction sets are wisely designed so that the PC cannot be used directly by the programmer as GPR's and FPR's can.

**Data Types.** For integer operations, the data operands can be bytes (8 bits), half words (16 bits), and words (32 bits). The byte and half word operands are expanded by the CPU into 32 bits. Depending on the operation, the expansion is done by padding with 0's or sign extending the operand.

For floating point operations, the data operands can be a word (32 bits) or a double word (64 bits). The FPR's are used in pairs for double word operands with F0 specifying the pair (F0, F1), F2 specifying (F2, F3), etc.

**Instruction Word Encoding.** Each instruction fits into a single 32 bit word. Each instruction must be read and decoded by the processor hardware before an instruction can be executed. The hardware designer decides how to encode the instructions to simplify instruction decoding as much as possible. In the class design projects, we will make up our own encoding scheme. The real MIPS has the encoding scheme shown on the inside front cover.

The R format is for three register operands. Note that the 11 bits on the right are used as additional opcode bits for this format allowing the instruction set to be larger than what would be possible with just the 6-bit opcode. The rs and rt fields encode the two source registers. The rd field encodes the destination register.

The I format is for two register operands and an immediate operand. In this format, the rt field specifies the destination register. Load and store instructions also use this format, but use the rt field as a base register and the last 16 bits are the displacement in the base displacement memory address format. Branch instructions use this format with two source operands (which are compared) in the rs and rt fields, and the last 16 bits are the offset in PC relative memory address format.

The J format is used by jump type instructions to allow the maximum number of bits in the jump offset.

Note that the registers can be specified with a 5 bit field in the instruction format. This leaves plenty of room for three register fields in the 32 bit instruction word. This gives the maximum flexibility in specifying two source and one destination register for these type operations. Some machines do not allow specification of three different registers in order to simplify the CPU design or because there is not enough space in the instruction for three register specs (consider 16-bit instructions for example).

### MIPS Instruction Types

The MIPS instructions are representative of instructions for a modern RISC. They are summarized on the facing page to the inside front cover. Chapter 2 provides extensive examples of how these instructions are used to implement constructs in a high level language (C is used in the examples).

**Arithmetic and Logical Instructions.** The logical, add and subtract instructions work as described by our earlier implementation of the ALU. We will discuss the multiply and divide instructions later.

Note that the assembly language convention lists the destination register first even though it is not first in the binary instruction format (see the example on p. 53).

Note also that all numbers are decimal numbers unless otherwise specified (C language convention). For example,

```
addi $s1, $s2, 100
```

can also be written as

```
addi $s1, $s2, 0x64
```

using the C language convention for hexadecimal numbers.

None of the arithmetic instructions take operands directly from memory nor do they store results directly in memory. The idea is to force the program to use registers most of the time and use explicit reference to other instructions (load and store) whenever one wishes to transfer data to or from memory.

**Data Transfer.** The load and store instructions use only base-displacement memory addressing.

What about GPR register to register transfers, i.e. why is there no MOV instruction for GPR's? The reason is that we already have a MOV instruction with

```
addi R2, R1, 0 (R1 + 0 => R2)
```

or

```
add R2, R1, R0 (R1 + 0 => R2)
```

which does the same thing since R0 has a hardwired 0 in it when used as an arithmetic operand. Although it may seem inefficient to do a register to register data transfer that goes through the ALU, the above two instructions execute in a single clock cycle already and there is no performance benefit to adding a simpler MOV instruction.

What about loading data directly from the instruction word into a register (load immediate which puts an initial value in a register without using memory operands)? This is possible using the hardwired 0 in R0.

```
add R7, R0, R0 (clear R7, put 0 into R7)
addi R7, R0, 4 (put 4 into R7)
```

**Branching/Jumping.** In the MIPS instruction set, jump instructions always cause the next instruction to be fetched from the address operand in the instruction. This is called an unconditional jump. Branch instructions in MIPS are always conditional. The branch has no effect (next instruction is executed) if the branch condition is false. If the condition is true, the address operand in the branch instruction determines where the next instruction is fetched.

Jump and branch instructions are summarized in the Unconditional Jump and Conditional Branch categories on the facing page to the inside back cover.

The `j` and `jal` instructions use pseudo-absolute memory addressing. In addition, there are register deferred versions of the jump instructions (`jr`, `jalr`) so that a full 32 bit jump can be done.

`BEQZ`, `BNEZ` instructions use relative (to the PC) addressing. Note that the offset is interpreted as a word offset and must be multiplied by 4 to get the byte offset to add to the PC.

The MIPS instruction set designers have chosen the compare and branch technique for the conditional branches. Thus, the register operands are compared to determine whether the branch is taken. This comparison operation is part of the branch instruction, not another instruction.

The “set” type of instructions are necessary to do general comparison operations. Branch instructions only check equality of the two operands.

**No Operation Instructions.** It will be convenient when we start pipelining to have instructions that do not do anything. These instructions are usually called NOP (no ops). There are several ways to implement a NOP. Any instruction that does not change the register contents or memory can be used as a NOP.

```
addi R1, R1, #0 (add 0 to R1)
```

```
add R0, R1, R2 (put result in R0)
```

Note that `R0` has a hardwired 0 in it so that using `R0` as a destination register has no effect (`R0` acts as a “wastebasket”).

**Procedure Call.** The jump and link (`jal`, `jalr`) are used for procedure calls in high level languages. These instructions are just like an unconditional jump (`j`, `jr`), but the incremented value of the PC is stored in `R31`. For example

```
jalr R7
```

causes the following action

$R31 \leftarrow PC + 4$  (store the subroutine return address in R31)  
 $PC \leftarrow \text{Regs}[R7]$  (jump to the beginning of the subroutine)

Note that the procedure call instruction must provide a special mechanism to store the return address since this instruction set does not allow normal data transfers with the PC (the PC is a special register). Observe that the return address is stored in a *register* allowing the programmer to avoid an extra memory operation required to store the return address in memory (at least until there is no other use of R31).

This procedure call instruction is only very minimal. Instruction frequency studies indicate that procedure calls are very common. Other support required for procedure calls that one might consider for hardware implementation:

- Calling procedure must pass data (procedure arguments) to called procedure
- Called procedure must not destroy register contents being used by calling procedure
- Called procedure must obtain memory space to save register contents and storage for local variables and the space must be released when the called procedure is completed
- Called procedure must pass results to calling procedure
- Nested calls must have more space to store return addresses

Software convention is better for all of the above than hardware implementation because

1. additional hardware support for procedure call requires multiple clocks, violating RISC design goals
2. hardware implementation must design for worst case (i.e., call execution would have to save *all* registers in memory) -- software can use optimizing compiler (i.e., compiler inserts additional instructions to save in memory only those registers needed)
3. use of compiler insures everyone uses the same conventions so that there is no need to build conventions into hardware

**I/O Instructions.** The MIPS instruction set does not have any I/O instructions. Was this an unfortunate oversight? What good is a computer without the ability to get information into and out of it?

The MIPS designers are assuming that a technique called memory mapped I/O will be used to allow two way communication with the outside world. The I/O communication interface will be wired up to look like a piece of ordinary memory. Whenever the CPU issues an address corresponding to the I/O interface, the data goes from/to the I/O device instead of from/to memory. The system designer (not the CPU designer) makes the decision about which addresses to map into I/O interfaces instead of memory.

The advantage of memory mapped I/O is that no explicit I/O instructions are necessary. This fits right into our RISC philosophy. The absence of I/O instructions allows a simpli-

fication of the CPU hardware. The disadvantage is that not all of the CPU address space will be available for memory - some of it must be used for the I/O interface. There is such a large address space that we can afford to give up a small part of it to I/O interfaces.