CS2100
Computer Organization

AY2022/23 Semester 1

Notes by Jonathan Tay

Last updated on August 22, 2022

Contents

Part I

Data Representation

1 Number Systems

Any base-R number system is **weighted-positional**.

The **decimal** number system has a **base** (or **radix**) of 10.

Every digit going leftward from the decimal point has a weighted power of 10, starting from 10^0 and increasing.

Every digit going rightward from the decimal point has a weighted power of 10, starting from 10^{-1} and decreasing.

This generalizes to any base-R number system, with R^i in place of 10^i.

1.1 Conversions between Bases

We use the decimal system as an intermediary to convert between bases.

For example, to convert between base-2 (**binary**) to base-7, we first convert to base-10, then convert to base-7.

We can use grouping and/or ungrouping to convert between bases which are powers of 2.

For example, to convert between base-2 and base-16 (**hexadecimal**), we partition the digits of the binary number in groups of 4, starting from the binary point, and partitioning outward.

1.1.1 Base-*R* ⇒ Decimal

The decimal equivalent of a base-R number is the sum of its weighted digits.

1.1.2 Decimal ⇒ Base-*R*

This conversion performs **division-by-*R*** for the whole number portion, and **multiplication-by-*R*** for the fractional portion.

Division-by-R divides the whole number by R until the quotient is 0. The result is produced from the remainders from **right to left**.

Multiplication-by-R multiplies the fractional portion by R until the product is 0, or until the desired number of decimal places. The result is produced from the **carries left to right**.

1.2 Negative Numbers

Signed numbers include all positive and negative values, unlike **unsigned numbers** which only include positive values.

There are 4 representations for signed binary numbers covered in this module.

1.2.1 Sign-and-Magnitude

Negate a number by inverting the sign bit.

This representation uses the leftmost (most-significant bit) as the **sign bit** — 1 for negative, 0 for positive.

This results in **duplicate zeros** (positive and negative), and a range of values of $-\left(2^{n-1}-1\right)$ to $2^{n-1}-1$.

1.2.2 1s Complement

Negate a number by inverting each bit.

An n-bit number x has a negated value $-x = 2^n - x - 1$.

This also results in **duplicate zeros** (positive and negative), and a range of values of $-\left(2^{n-1}-1\right)$ to $2^{n-1}-1$.

The weight of the leftmost bit is $-\left(2^{n-1}-1\right)$.

1.2.3 2s Complement

Negate a number by inverting each bit, then adding 1.

Alternatively, preserve each bit up to and the rightmost 1, then inverting every bit to the left of the rightmost 1.

An n-bit number x has a negated value $-x = 2^n - x$.

This also results in **no duplicate zeros**, and a range of values of -2^{n-1} to $2^{n-1}-1$.

The weight of the leftmost bit is -2^{n-1}.

1.2.4 Excess Representation

Distribute positive and negative values by a simple addition or subtraction.

Excess-k subtracts k from the decimal value of the number, while preserving its base-2 representation.

To convert from a decimal value to its excess-k representation, we add k to the decimal value and convert that to base-2.

Typically, for an n-bit number, $k = 2^n - 1$.

1.3 Real Numbers

Fixed-point representation allocates a fixed number of bits for the whole number and fractional parts.

This results in a limited range of numbers that can be represented.

The IEEE 754 **floating point representation** resolves this by allocating a fixed number of bits to the **sign**, **exponent**, and **mantissa**.

precision	bits	sign	exponent	mantissa	bias
single	32	1	8	23	127
double	64	1	11	52	1023

The **sign bit** is 0 for positive numbers, and 1 for negative numbers.

The **mantissa** is **normalized** with an implicit leading 1 bit, e.g. 110.1_2 is normalized to $1.101_2 \times 2^2$, and only 101 is stored in the mantissa.

Part II

MIPS

An **instruction set architecture** (ISA) is an abstraction on the interface between hardware and low-level software.

MIPS is one such ISA, which runs on a **processor**.

Processors are connected to **memory** (RAM) via a **bus**, across which code and data is transferred.

2 Registers

Memory access is slow. To avoid frequent memory access, temporary values are stored in the processor in **registers**, which are limited in number.

Registers *do not have data types*, unlike program variables. Instructions always assume that the data stored in a register is of the correct type.

MIPS has **32 registers**, which are referred to by *number* or *name*:

name	**#**	**usage**
`$zero`	0	constant value zero
`$at`	1	reserved for the assembler
`$v0` - `$v1`	2 - 3	values for results and expression evaluation
`$a0` - `$a3`	4 - 7	arguments
`$t0` - `$t7`	8 - 15	temporaries
`$s0` - `$s7`	16 - 23	program variables
`$t8` - `$t9`	24 - 25	temporaries
`$k0` - `$k1`	26 - 27	reserved for the OS
`$gp`	28	global pointer
`$sp`	29	stack pointer
`$fp`	30	frame pointer
`$ra`	31	return address

3 MIPS Assembly Language

The **general instruction syntax** is as follows:

`op $s0, $s1, $s2`	
	description
`op`	operation
`$s0`	destination register
`$s1`	source register 1
`$s2`	source register 2

Each instruction executes a *single* command. Each line of assembly code contains *at most* one instruction.

Almost all MIPS operations are *register-to-register*

The # hex synbol is used for comments.

3.1 Arithmetic Instructions

instruction	**effect**
`add $s0, $s1, $s2`	`$s0 = $s1 + $s2`
`sub $s0, $s1, $s2`	`$s0 = $s1 - $s2`
`addi $s0, $s0, <k>`	`$s0 = $s0 + <k>`
`move $s0, $s1`	**pseudo-instruction** for `$s0 = $s1`, equivalent to `add $s0, $s1, $zero`

The constants <k> in **immediate operations** such as `addi` range from -2^{15} to $2^{15}-1$, as the 16-bit 2s complement system is used.

However, if 32-bit constants are required, use the `lui` operation to load the most-significant (leftmost) 16-bits first, followed by an `ori` operation to set the least-significant 16-bits.

There is no `subi` operation as its equivalent to `addi` with a negative constant.

Use the **temporary registers** `$t0` to `$t9` to store intermediate results in complex expressions.

3.2 Logical Instructions

instruction	**effect**
`sll $t2, $s0, <k>`	`$t2 = $s0 « <k>`, equivalent to `t2 *=` $2^{<k>}$
`srl $t2, $s0, <k>`	`$t2 = $s0 » <k>`, equivalent to `t2 /=` $2^{<k>}$
`and $t0, $t1, $t2`	bitwise AND, where `$t2` is the **bit-mask**
`or $t0, $t1, $t2`	bitwise OR, to force certain bits to 1
`nor $t0, $t1, $t2`	bitwise NOR, only 1 if neither bits are 0
`xor $t0, $t1, $t2`	bitwise XOR, only 1 if both bits are different
`andi $t0, $t1, <k>`	bitwise AND with a constant k
`ori $t0, $t1, <k>`	bitwise OR with a constant k
`xori $t0, $t1, <k>`	bitwise XOR with a constant k

In bitshift operations (`sll` and `srl`), the empty positions are filled with zeros.

There is no `not` operation as its equivalent to `nor <dest> <src> $zero`.

There is no `nori` operation as it is rarely used, and not adding it keeps the processor design simple.

3.3 Memory Instructions

Memory can be thought of as a *single-dimensional array* of memory location, with each having an **address**.

Memory addresses allow access to *bytes* of data, or **words** of data, which are usually 2^n bytes — the common unit of transfer between the processor and memory.

Word alignment occurs in memory when words begin at a *byte address* which is a multiple of the word size — 2^n bytes.

In MIPS, each word is 32 bits (4 bytes), and addresses are 32-bits long — such that 2^{30} words are addressable, each of which differing by 4.

instruction	**effect**
`lw   $dst, k($src)`	loads word at `Mem[*src + k]` into register `$dst`
`sw   $src, k($dst)`	stores word in register `$src` into `Mem[*dst + k]`
`lb   $dst, k($src)`	loads byte at `Mem[*src + k]` into register `$dst`
`sb   $src, k($dst)`	stores byte in register `$src` into `Mem[*dst + k]`
`ulw  $dst, k($src)`	psuedo-instruction for loading unaligned words
`usw  $src, k($dst)`	psuedo-instruction for storing unaligned words

Memory operations the only operations which can access data in memory, but there are others which are less frequently used that are not listed here.

Unlike in `lw` and `sw`, the displacement constants `k` for `lb` and `sb` do not need to be multiples of 4.

3.4 Control Flow Instructions

instruction	**effect**
`beq   $r1, $r2, label`	goes to the labelled statement if `*r1 == *r2`
`bne   $r1, $r2, label`	goes to the labelled statement if `*r1 != *r2`
`j     label`	goes to the labelled statement unconditionally
`slt   $dst, $s1, $s2`	`*dst = *s1 < *s2 ? 1 : 0`
`slti  $dst, $src, k`	`*dst = *src < k ? 1 : 0`

Labels are written as `<label>:` to the left of a statement.

A `j` instruction is equivalent to `beq $s0 $s0, <label>`.