CS2100

Computer Organization

AY2022/23 Semester 1

Notes by Jonathan Tay

Last updated on September 8, 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, and the **shift amount** is limited to **5 bits**.

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`	jumps to the labelled statement
`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>`.

The **program counter** (PC) typically stores the address of the *next* address to be executed, and has to be modified by the branching and jump instructions.

4 Instruction Encoding

Every MIPS instruction is **32 bits** in 3 possible formats:

format	source registers	destination registers	immediate values
R	2	1	0
I	1	1	1
J	0	0	1

4.1 R-format

	`opcode`	`rs`	`rt`	`rd`	`shamt`	`funct`
bits	6	5	5	5	5	6

- `opcode := 0` for all R-format instructions,
- `funct` determines the instruction,
- `shamt := 0`, `rd := arith(rs, rt)` for non-shift (arithmetic) instructions, and,
- `rs := 0`, `rd := shift(rt, shamt)` for shift instructions.

4.2 I-format

	`opcode`	`rs`	`rt`	`immediate`
bits	6	5	5	16

- `opcode` determines the instruction since there is no `funct` field,
- `rt` determines the *destination* register since there is no `rd` field,
- `immediate` is a *signed* integer in 2s complement except for bitwise operations where it is *unsigned*,
- `rt := op(rs, immediate)` in general for all non-branching instructions, and,
- `PC := (PC + 4) + (immediate * 4)` when branching, otherwise `PC += 4`, which is the next instruction.

For branching instructions, `immediate` is the offset from the *next* instruction to the label of the *target* instruction.

PC is incremented in multiples of 4 due to word-alignment, which also means that we can now branch $2^{15} \times 4 = 2^{17}$ bytes away from PC.

4.3 J-format

	`opcode`	`target address`
bits	6	26

The last 2 bits of every instruction are always `00` due to word-alignment, so we leave them out of the target address.

This leaves with an effective range of 28 bits for the target address, and the remaining 4 bits are derived from the most significant (leftmost) bits of `PC + 4`.

The **destination address** is therefore:

	(PC+4)[0:4]	target address	00
bits	4	26	2

This creates a maximum jump range of 256 MB.

4.4 Addressing Modes

Addressing modes are used to calculate the address of an operand.

1. **register addressing** — `add, xor, etc.`:
   operand is a register
2. **immediate addresssing** — `addi, andi, etc.`:
   operand is a constant within the instruction
3. **base/displacement addressing** — `lw, sw`:
   operand is a memory location at the address the sum of a register and a constant in the instruction
4. **PC-relative addressing** — `beq, bne`:
   address is the sum of the PC and a constant in the instruction
5. **psuedo-direct addressing** — `j`:
   part of the instruction concantenated with part of the PC

5 Aside: Instruction Set Architecture

ISAs have two major design philosophies:

1. **CISC** — Complex Instruction Set Computer:
   - single instructions for complex operations
   - smaller program sizes
   - complex implementation, little room for hardware optimization
2. **RISC** — Reduced Instruction Set Computer:
   - smaller and simpler instruction set
   - software combines simpler operations to implement high-level language statements
   - room for compiler optimization

5.1 Data Storage Architecture

There are several common designs:

1. **stack architecture**:
   operands are implcitly popped from the stack
2. **accumulator architecture**:
   one operand is implcitly stored in an accumulator
3. **general-purpose register architecture**:
   operands are stored explicitly in registers, operations are register-memory or register-register
4. **memory-memory architecture**:
   operands are read from memory

5.2 Memory Architecture

Memory transfer takes place across buses:

1. **address bus**: *k*-bits, uni-directional from processor to memory
2. **data bus**: *n*-bits, bi-directional
3. **control lines**: bi-directional, e.g. read/write controls

The address bus feeds addresses from the **memory address register** to the memory.

Data is written to or read from the **memory data register**, depending on the R/W control line.

Endianness is the relative ordering of the *bytes* in a word (*not* the bits in a byte!) in memory:

- **big-endian**: MSB stored in smallest address
- **little-endian**: LSB stored in smallest address

6 Datapath

A collection of components that process data, and perform arithmetic, logical, and memory operations.

The **instruction execution cycle** has 5 stages:

1. **fetch**:
   get instruction from memory, address in PC
2. **decode and operand fetch**:
   determine the operation and get the operands needed
3. **execute (ALU)**:
   perform the computations to get a result
4. **execute (memory access)**:
   read from or write to memory if necessary
5. **write-back**:
   store the result of the operation

6.1 Fetch

There are 3 steps in the fetch stage:

1. Use PC to fetch the instruction from memory.
2. Increment PC by 4 to get the address of the next instruction.
3. Feed the output instruction to the decode stage.

The **instruction memory** element is a **sequential circuit** with an internal state which stores the instructions.

When supplied an instruction address *m*, it outputs the content at address *m*.

The **adder** element takes in the 32-bit PC and the 32-bit constant 4, and outputs the 32-bit sum PC + 4.

The **clock signal** is a *square wave* with *rising* and *falling* edges, and a period controlled by the CPU.

PC is read in the first half of the **clock period** and updated at the *next rising clock edge*.

6.2 Decode

There are 3 steps in the decode stage:

1. Read the `opcode` to determine the instruction type and field lengths.

2. Read data from all necessary registers.
3. Feed the operation and operands to the ALU stage.

The **register file** element is a collection of 32 registers which can be read from or written to.

input	bits	output	bits
read register 1	5	read data 1	32
read register 2	5	read data 2	32
write register	5		
write data	32		

The **RegWrite** control signal determines whether the instruction should read from or write to the registers:

	RegWrite	registers per instruction
read	0	≤ 2
write	1	≤ 1

6.2.1 R-format Decoding

The binary content (Inst) of R-format instructions map to the inputs as follows:

- rs ≡ Inst[25:21] ↦ rr1
- rt ≡ Inst[20:16] ↦ rr2
- rd ≡ Inst[15:11] ↦ wr

6.2.2 I-format Decoding

For I-format instructions, 2 problems arise:

1. rt, not part of imm, needs to be fed to wr
2. imm needs to be fed to the ALU, not rd2

A **multiplexer** chooses the correct Inst slice to feed to wr using the control signal **RegDst**:

	RegDst	input to wr
I-format	0	rt ≡ Inst[20:16]
R-format	1	rd ≡ Inst[15:11]

- rs ≡ Inst[25:21] ↦ rr1
- rt ≡ Inst[20:16] ↦ rr2
- rt ≡ Inst[20:16] ↦ wr

Another multiplexer chooses the correct 32-bit binary data to feed into the ALU:

	ALUSrc	input to ALU opr2
R-format or beq	0	rr2 ≡ *rt ≡ *Inst[20:16]
I-format not beq	1	imm ≡ Inst[15:0], sign extended to 32-bits

6.3 Arithmetic Logic Unit (ALU)

The ALU performs the arithmetic, shifting, logical, memory, and branching operations.

It takes the operation and operands from the decode stage, performs its computation, and feeds the result to the memory stage.

input	bits	output	bits
operand 1	32	is zero?	1
operand 2	32	result	32

The **ALUControl** signal determines the operation to perform:

ALUControl	operation
0000	and
0001	or
0010	add
0110	sub
0111	slt
1100	nor

6.3.1 Branching Instructions

A multiplexer chooses between the next instruction (PC + 4) and the branch target (BT) using the control signal **PCSrc**:

	PCSrc	next instruction
false	0	PC + 4
true	1	BT ≡ PC + 4 + imm * 4

Because both register contents need to be compared, the ALUSrc control value is set to 0 for branching instructions.

BT is computed as follows:

1. left shift by 2 bits the sign-extended imm from the decode stage (multiplying the offset by 4)
2. add it to PC + 4 from the adder in the fetch stage

6.4 Memory

Only lw/lb and sw/sb instructions operate in this stage, the rest remain idle.

The **data memory** element exists in RAM:

input	bits	output	bits
address (addr)	32	read data (rdt)	32
write data (wd)	32		

addr is the effective memory address computed by the ALU, and wd is connected to rd2 from the register file from the decode stage.

The control signals **MemRead** and **MemWrite** control the memory operations:

MemRead	MemWrite	action
0	1	write wd into addr
1	0	read contents of addr into rdt
0	0	do nothing
1	1	undefined, should not occur

6.4.1 Non-memory Instructions

A multiplexer chooses between the result of the ALU and the read data from memory, using the control signal **`MemtoReg`**:

	`MemToReg`	**output**
non-memory instruction	`0`	ALU output
memory instruction	`1`	read data

This output is fed to the write-back stage.

6.5 Write-back

Stores, branches, and jumps remain idle in this stage as there is nothing to be written.

The result of the memory stage is fed into write data (`wd`) of the register file.