**CS2100 Midterm CheatSheet**

**C Language**

**~** is NOT, ^ is XOR (Bitwise Operator)

! is boolean operator, not bitwise operator

%lf is used when you want to scan double

\t is used in printing to print a tab

*Syntax of Structure*

typedef struct {

members…..

} <structure name>**;**

Use . if you want to access the members of a structure.

We may do assignments with structure, meaning if we assign one structure to another structure variable, it will copy all the members of the structure into the respective members of the structure of the structure variable.

(\*player\_ptr).name /\*legal\*/

\*player\_ptr.name /\*illegal as treated as \*(player\_ptr.name)/

player\_ptr->name /\*legal\*/

**Declaring an array of pointers:** type \*a[]

**Number System**

*Conversion from decimal to base-R*

Whole number: repeated division by R (copy resultant digits down to up)

Fractional number: repeated multiplication by R (copy resultant digits up to down)

*Negation of a number*

**1s**: -x = 2n – x – 1 (n is number of bits of x in binary)

To get negation of x: invert all the bits of x (same for fraction)

**2s**: -x = 2n – x (n is number of bits of x in binary)

To get negation of x: invert all the bits of x and **add 1**

(if fraction, same thing, but plus 1 at the **rightmost digit** after the “.”)

*Addition*

**2s:**Perform addition as usual, **ignore carry out of the MSB**, check for overflow (occur when the resultant sign bit is different from that of the operands in the case where both operands have the same sign)

1s: Same as above, **however** **if there is a carry out of the MSB, add 1 to the rightmost digit (**same in fractional num)

**Hexa: …, 9, A, B, C, D, E, F**

*Subtraction (a – b)*

Convert b to its complement and perform addition.

*Excess*

\*Binary rep. of x in excess-N = Binary rep. of (x + N)

\*Dec rep. of x represented in excess-N: Convert x back to decimal and minus the result by N

*IEE 754 Single-Precision Floating-Point Rep*

1-bit sign, 8-bit exponent represented in **EXCESS 127**, 23-bit mantissa   
**Decimal to IEE:** Convert the number to binary, normalize it, and fill in the parts accordingly

*Sign Extension*

***Unsigned****:* Copy all the bits and put it at the rightmost part, fill the unfilled front part with zeros

***Signed****:* Copy **the magnitude part** and put it to the rightmost part, copy sign bit and put it at the first bit, then fill the in -between gap with zeros

***1s and 2s***: Copy **all the bits** and put it at the rightmost part, fill the unfilled front part with the sign bit of the original num.

**MIPS** (Immediate is a **2s** **complement**)

Given k-bit addresses, you can have 2k addresses.

*Branching*

The immediate field is equal to the number of instructions you skip over, starting from **the end of the branch instruction**. (if jump backwards **then immediate is -ve**, must count the branch instruction as one of the instruction to skip too!)

If branch is not taken: PCnew = PC + 4

If branch is taken: PCnew = (PC + 4) + (immediate x 4)

*Jump*

Given address of instruction, drop the first 4 bits and the last 2 bits, the rest will be your immediate field.

Getting NOT from NOR: nor x, x, $zero (x is the same register)

Getting NOT from XOR: xor x, x, y (y is the register that contain all 1s)

ANDI, ORI, XORI will **zero extend** the 16-bit immediate **to the left**!

LB and LH will **sign extend** the 8 bits and 16 bits that they fetched respectively from the memory and store it in the register

SB and SH will only store the 8 and 16 **least significant bits** in the register mentioned respectively into the memory

LUI shifts the given 16 bit immediate to the left by 16 bits and concatenate it with 16 zeros.

To generate a mask (i.e. keep the bits you want) using ANDI, you use bit 1 to keep the bit that you want and 0 for the bit you don’t want.

OR can be used to force certain bits to become 1.

**ISA** (MSB = most significant Byte)

Big-endian: MSB are stored in lower addresses

Little-endian: LSB are stored in lower addresses

**Data Path**

**Instruction Execution Cycle:** Fetch, Decode, Operand Fetch, Execute, Result Write (Store)

**Multiplexer**: Inputs: n lines, Control: m bits where n = 2m

**Control** (Note the following control signals aren’t complete)

**RegDst**: 0 -> choose rt; 1-> choose rd

**ALUSrc**: 0 -> choose RD2; 1-> choose SignExt(Inst[15:0])

**MemToReg:** 0 -> choose ALU result; 1-> choose Memory Read Data

**PCSrc**: 0 -> PC + 4; 1 -> (PC+4) + SignExt(Inst[15:0]) << 2

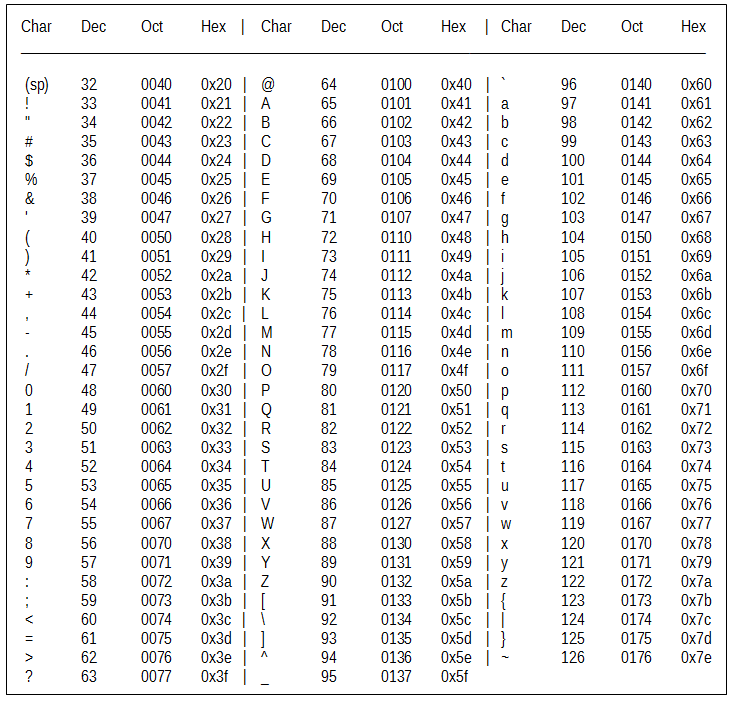
***ALU Control***

0000 AND | 0001 OR | 0010 add | 0110 subtract | 0111 slt

1100 NOR

**ALUOp**

00 lw/sw | 01 beq | 10 R-Type



**Power of 2**

27 = 128, 28 = 256, 29 = 512

Note that each upper-case alphabetical letter in ASCII differs by a value of **32** with its lower-case alphabetical letter.