1 C Programming Language

1.1 Data Types

Type	sizeof	range
int	4 bytes	2s complement, thus (-2^{31}) to $(2^{31}-1)$ OR (-2,147,483,648 to 2,147,483,647)
float	4 bytes	1-bit sign, 8-bit exponent (excess-127), 23-bit mantissa
double	8 bytes	1-bit sign, 11-bit exponent (excess-1023), 52-bit mantissa
char	1 byte	ASCII (7 bits + 1 parity bit), A is 100 0001

Note that for mantissa there is an implicit leading bit 1

1.2 Format Specifiers

	Type	fn
%c	char	printf/scanf
%d	char	printf/scanf
%f	float/double	printf

	Type	fn
%f	float	scanf
%lf	double	scanf
%p	pointers	printf

1.3 Escape Sequences

	Meaning
\n	new line
\t	tab

	Meaning
\"	double-quote "
%%	percent %

1.4 Misc

- Short-circuit evaluation

2 Numbering Systems

2.1 Data Representation

- 1 byte = 8 bits
- n bits can represent up to 2^n values. Thus, to present m values, $\lceil \log_2 m \rceil$ is required

2.2 Decimal to Binary Conversion

- For whole numbers: repeated division-by-2 (look at remainder)
- For fractions: repeated multiplication-by-2 (look at "quotient")

2.3 Representation of Signed Binary Numbers

	Negation	Range	Zeroes
Sign-and-Magnitude	invert the sign bit (leading bit)	$-(2^{n-1}-1)$ to $2^{n-1}-1$	$+0_{10}$ and -0_{10}
1s Complement	invert all the bits	$-(2^{n-1}-1)$ to $2^{n-1}-1$	$+0_{10}$ and -0_{10}
2s Complement	invert all the bits, then add 1	-2^{n-1} to $2^{n-1}-1$	$+0_{10}$

For all of the above, the MSB (Most Significant Bit) represents sign.

2.3.1 Sign-and-Magnitude

- Range (8-bit): (1111 1111) to (0111 1111) = -127_{10} to $+127_{10}$
- Zeroes: 0000 0000 = $+0_{10}$ and 1000 0000 = -0_{10}
- e.g. $(0011\ 0100)_{sm} = +011\ 0100_2 = +52_{10}$, $(1001\ 0011)_{sm} = -(001\ 0011)_2 = -(19)_{10}$

2.3.2 1s Complement (Diminished Radix)

- Negation: $-x = 2^n - x - 1$
- Range (8-bit): (1000 0000) to (0111 1111) = -127_{10} to $+127_{10}$
- Zeroes: (0000 0000) = $+0_{10}$ and (1111 1111) = -0_{10}
- e.g. $(0000\ 1110)_{1s} = (0000\ 1110)_2 = (14)_{10}$, $(1111\ 0001)_{1s} = -(0000\ 1110)_2 = -(14)_{10}$

2.3.3 2s Complement (Radix complement)

- Negation: $-x = 2^n - x$
- Range (8-bit): (1000 0000) = -128_{10} to (0111 1111) = $+127_{10}$
- Zero: (0000 0000) = $+0_{10}$
- e.g. $(0000\ 1110)_{2s} = (0000\ 1110)_2 = (14)_{10}$, $(1111\ 0010)_{2s} = -(0000\ 1110)_2 = -(14)_{10}$

2.3.4 Excess-k

- Also known as offset binary. Use 0000 to represent $-k$ (lowest number possible)
- For unsigned, with n-bit number, $k = 2^{n-1} - 1$

2.3.5 Comparison

Value	Sign-and-Magnitude	1s Complement	2s Complement	Excess-8	Value
+7	0111	0111	0111	1111	+7
+6	0110	0110	0110	1110	+6
+5	0101	0101	0101	1101	+5
+4	0100	0100	0100	1100	+4
+3	0011	0011	0011	1011	+3
+2	0010	0010	0010	1010	+2
+1	0001	0001	0001	1001	+1
+0	0000	0000	0000	1000	+0
-0	1000	1111	-	-	-0
-1	1001	1110	1111	0111	-1
-2	1010	1101	1110	0110	-2
-3	1011	1100	1101	0101	-3
-4	1100	1011	1100	0100	-4
-5	1101	1010	1011	0011	-5
-6	1110	1001	1010	0010	-6
-7	1111	1000	1001	0001	-7
-8	-	-	1000	0000	-8

2.4 Operation on binary numbers

Algorithm for **Subtraction**: $A - B = A + (-B)$

Algorithm for **Overflow Check**: if MSB of first and second are the same, then MSB of resulting numbers must be the same too.

2.4.1 2s Complement on Addition

Algorithm: (1) Perform binary addition. (2) Ignore the carry out of the MSB. (3) Check for overflow.

```
Example, 2s Complement 4-bit
+3  0011                -2   1110                -3   1101
+4  0100                -6   1010                -6   1010
--- -----               ---  -----               ---  -----
+7  0111 (No overflow)  -8 (1)1000 (No overflow)  -9  (1)0111 (Overflow!)
```

2.4.2 1s Complement on Addition

Algorithm: (1) Perform binary addition. (2) If there is carry out of the MSB, add 1 to the result. (3) Check for overflow.

```
Example, 1s Complement 4-bit
+3  0011                -2   1101                -3   1100
+4  0100                -5   1010                -6   1001
--- -----               ---  -----               ---  -----
+7  0111 (No overflow)  -7 (1)0111               -9  (1)0101
                              1                        1
                            -----                    -----
                            1000 (No overflow)       0110 (Overflow!)
```

2.5 Floating Point

- Single precision 32 bits: 1-bit sign, 8-bit exponent (excess-127), 23-bit mantissa
- Double precision 64 bits: 1-bit sign, 11-bit exponent (excess-1023), 52-bit mantissa
- e.g. $-6.5_{10} = -110.1_2 = -1.101_2 \times 2^2$, Exponent (excess-127) = $2+127 = 129 = 1000\ 0001_2$

```
Sign Exponent          Mantissa
 1   10000001  10100000000000000000000
```

Hence, $1100\ 0000\ 1101\ 0000\ 0000\ 0000\ 0000\ 0000_2 = C0D00000_{16}$

(as float= −6.5, as int= −1,060,110,336)

3 Pointers and Functions

3.1 Pointers

- New unary operators: * and &
- Convention: `int *abc;` AND `void f(int *);`

3.2 Functions

- Function prototype (just the type of its parameters): e.g. `void g(int, int);`
- Variable scoping: by **functions**

4 Arrays, Strings, Structures

4.1 Arrays

- Array is a homogeneous collection of data, occupying contiguous memory locations.
- When initialised with fewer values than elements, the rest are initialised as 0 (for `int`).
- Equivalence: value: `*(arr+2) == arr[2]` and memory location: `arr + 2 == &arr[2]`

4.2 String

- An array of characters, terminated by a null character `'\0'` (ASCII value: 0)
- Initialising: `char str[4] = "egg";` or `char str[4] = {'e', 'g', 'g', '\0'};`
- Read from stdin: `fgets(str, size, stdin); // reads until (size - 1) or '\n'` and `scanf("%s", str); // reads until whitespace`
  (note that `fgets` also reads in `'\n'`)
- Print to stdout: `puts(str);` which is equivalent to `printf("%s\n", str);`
- String functions:
  - `strlen(s)`: returns the no of chars in s
  - `strcmp(s1, s2)`: compare ASCII values of corresponding characters, returns $\mathbb{Z}^+$ if s1 is lexographically greater, 0 if equal, $\mathbb{Z}^-$ otherwise
  - `strncmp(s1, s2, n)`: compare first n chars of s1 and s2
  - `strcpy(s1, s2)`: copy the string pointed by s2 into array pointed by s1, returns s1. E.g. `char s[4]; strcpy(s, "asdfgh"); // s == {'a', 's', 'd', '\0'};`
  - `strncpt(s1, s2, n)`: copy the first n chars of string pointed by s2 to s1

4.3 Structures

- Structures allow grouping of heterogeneous members of different types.
- Assignment `result2 = result1;` copies the entire structure.
- Passing structure to function: the entire structure is copied.
- Alternatively, to change original structure, one can use pointer. Syntactic sugar: `(*player_ptr).name == player_ptr->name;`
- E.g.:

```
typedef struct {
^^int day, month, year;
} date_t;
typedef struct {
^^int stuNum;
^^date_t birthday;
} student_t;
student_t s1 = {1049858, {31, 12, 2020}}; // s1.birthday.month == 2020
```

5 C for Hardware Programming

5.1 Code Compilation Process

C Program (.c) -> **Preprocessor** -> Preprocessed code (.i) -> **Compiler** -> Assembly code (.asm) -> **Assembler** -> Object code (.o) -> **Linker** -> Executable (.hex)

6 MIPS

6.1 Loading a 32-bit constant into a register

1. Use `lui` to set the upper 16-bit: `lui $t0, 0xAAAA`
2. Use `ori` to set the lower-order bits: `ori $t0, $t0, 0xF0F0`

6.2 Memory Organisation

- Each address contains 1 byte = 8 bit of content.
- Memory addresses are 32-bit long (2^{30} memory words).
- 32 registers, each 4-byte long. Each word is also 4-byte long.

6.3 MIPS Instruction Classification

6.3.1 R-format

- `op $rd, $rs, $rt`
- `sll $rd, $rt, shamt` (rs = 0)

6.3.2 I-format

- `op $rt, $rs, Immediate`
- Displacement address: offset from address in rs
- PC-relative address: no of instructions from next instruction $PC = (PC + immediate) \times 4$

6.3.3 J-format

- `op Immediate`
- pseudo-direct address: remove last 2 bit (since word-aligned, by default the 2 least significant bits are 00) and 4 most significant bits (always the same as instruction address).
- eg xxxx0000111100001111000011110000, immediate is 00001111000011110000111100

7 Instruction Set Architecture

For modern processors: **General-Purpose Register** (GPR) is most common. **RISC** typically uses **Register-Register (Load/Store)** design, e.g. MIPS, ARM. **CISC** use a mixture of Register-Register and Register-Memory, e.g. IA32

7.1 Data Storage

- **Stack architecture** : Operands are implicitly on top of the stack.
- **Accumulator architecture** : One operand is implicitly in the accumulator (a special register)
- **General-purpose register architecture** : only explicit operands
  - **Register-memory architecture** : one operand in memory.
  - **Register-register (or load store) architecture**
- **Memory-memory architecture** : all operands in memory.

7.2 Memory Addressing Modes

- **Endianness** :
  - **Big-endian** : Most significant **byte** stored in lowest address
  - **Little-endian** : Least significant **byte** stored in lowest address ("reverse-order")
- **Addressing modes** : in MIPS, only 3: **Register** `add $t1, $t2, $t3`, **Immediate** `addi $t1, $t2, 98`, **Displacement** `lw $t1, 20($t2)`

7.3 Operations in the instruction set

Amdahl's law: make common cases fast. Optimise frequently used instructions (**Load**: 22%, **Conditional Branch**: 20%, **Compare** 16%, **Store**: 12%)

7.4 Instruction Formats

- **Instruction Length** :
  - **Variable-length instructions** : Require multi-step fetch and decode. Allow for a more flexible (but complex) and compact instruction set.
  - **Fixed-length instructions** : used in most RISC, e.g. MIPS instructions are 4-bytes long. Allow for easy fetch and decode, simplify pipelining and parallelism. Instruction bits are scarce.
  - **Hybrid instructions** : a mix of variable- and fixed-length instructions.
- **Instruction Fields** : **opcode** (unique code to specify the desired operation) and **operands** (zero or more additional information needed for the operation)

7.5 Encoding the Instruction Set

- **Expanding Opcode** scheme:
  - E.g. **Type-A**: **6-bit** opcode, **Type-B**: **11-bits** opcode. Max no of instructions = $1 + (2^6 - 1) \times 2^5 = 2017$
    (1 Type-A instruction, Type-B "steals" $[2^6 - 1]$ opcodes from Type-A to prefix, each prefix having $[2^{11-6} = 2^5]$ opcodes)

8 Datapath

8.1 Instruction Execution Cycle

For MIPS: (1)Fetch (2)Decode & Operand Fetch (3)ALU (4)Memory Access (5)Result Write

- **Fetch** : Get instruction from memory, address is in Program Counter (PC) Register
- **Decode** : Find out the operation required
- **Operand Fetch** : Get operand(s) needed for operation
- **Execute** : Perform the required oepration
- **Result Write (Store)** : Store the result of the operation

8.2 Elements

- **Adder** **Input**: two 32-bit numbers, **Output**: sum of input numbers
- **Register File** **Input**: three 5-bit: Read register 1, Read register 2, Write register; 32-bit Write data, **Output**: two 32-bit Read data 1, Read data 2; **Control**: 1-bit RegWrite (1 = write)
- **Multiplexer** **Input**: n lines of same width, **Control**: m bits where $n = 2^m$, **Output**: Select i^{th} input line if control = i

- **Arithmetic Logic Unit** : **Input**: two 32-bit numbers, **Control**: 4-bit to decide the particular operation, **Output**: 32-bit ALU result, 1-bit isZero?

ALUcontrol	Function	ALUcontrol	Function
0000	AND	0110	subtract
0001	OR	0111	slt
0010	add	1100	NOR

- **Data Memory** **Input**: 32-bit memory address, 32-bit write data; **Control**: 1-bit MemWrite, 1-bit MemRead; **Output**: 32-bit ReadData