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	int	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 %

2 Numbering Systems

2.1 Data Representation

- 1 byte = 8 bits, 1 nibble = 4 bits
- 1 word = Multiple of bytes (usually in powers of 2)
- 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, 1^{st} digit LSB)
- For fractions: repeated multiplication-by-2 (look at "quotient", 1^{st} digit MSB)

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: $-x = 2^n - x - 1$, if given binary: invert all bits
- 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: $-x = 2^n - x$, if given binary: invert all bits then add 1
- 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, $k = 2^{n-1} - 1$ or $k = 2^{n-1}$

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**: $A - B = A + (-B)$, do sign extension before complementing

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.

If doing 1s complement addition on decimals and there is an overflow from MSB, add 1 to LSB of decimal portion

```
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

- Convention: `int *abc;` AND `void f(int *);`
- %*p* used as format modifier for addresses and is printed out in hexadecimal
- When we do `ptr++`, it increases the address by the size of the datatype

3.2 Functions

- Function prototype (just the type of its parameters): e.g. `void g(int, int);`
- Without function prototypes, compiler assumes default return type of `int`

4 Arrays, Strings, Structures

4.1 Arrays

- When initialised with fewer values than elements, the rest are initialised as 0 (for `int`).
- When an array name appears in an expression or passed as a parameter to a function, it refers to the address of the first element (i.e `&a[0]`)
- `int source[5]; int dest[5]; source = dest` is illegal!

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'` and we may need to replace it with `'\0'` if necessary (Lect 5, Slide 21))
- 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, original structure is not modified by function
- 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 == 12
```

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`. Note that `lui` sets the lower 16 bits to 0 automatically
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. (Note that words are usually 2^n bytes)

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`
- Immediate is a 16 bit 2s complement constant
- Displacement address: offset from address in rs
- PC-relative address: no of instructions from next instruction $PC = PC + 4 + (Immediate \times 4)$

6.3.3 J-format (can jump up to 256MB range)

- `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** : Relative ordering of bytes in a multiple-byte word stored in memory
- **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 operation
- **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

9 Useful MIPS commands

• To get a **NOT** operation: nor $t0, $t0 $zero OR xor $t0, $t0 $t2 where $t2 has parity 1 for all its bits

• To get a Branch if less than or equal (i.e BLE $s1, $s2, target) can use slt $t1 $s2 $s1 –> beq $t1, $zero, target OR
sub $t1, $t2, $s1 –> slt $t2, $t1, $zero –> beq $t2, $zero, target

Order of precedence

Operator Type	Operator	Associativity
Primary expression operators	( ) [ ] . -> expr++ expr--	Left to Right
Unary operators	* & + - ! ~ ++expr --expr (typecast) sizeof	Right to Left
Binary operators	* / %	Left to Right
	+ -	
	< > <= >=	
	== !=	
	&&	
	||	
Ternary operator	?:	Right to Left
Assignment operators	= += -= *= /= %=	Right to Left

Decreasing order of precedence

Python
`cond ? expr1 : expr2` → `expr1 if cond else cond2`

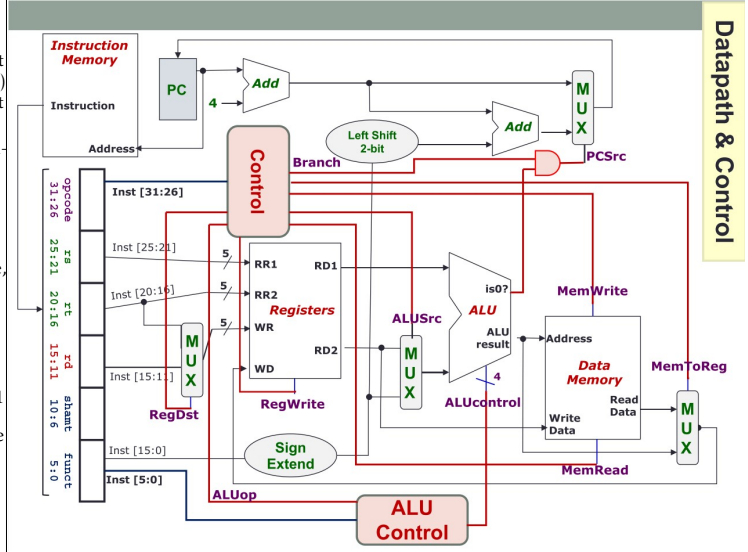
ALU Control

	ALUop		Funct Field (F[5:0] == Inst[5:0])						ALU control
	MSB	LSB	F5	F4	F3	F2	F1	F0	
lw	0	0	X	X	X	X	X	X	0 0 1 0
sw	0	0	X	X	X	X	X	X	0 0 1 0
beq	~~0~~ X	1	X	X	X	X	X	X	0 1 1 0
add	1	~~0~~ X	~~1~~ X	~~0~~ X	0	0	0	0	0 0 1 0
sub	1	~~0~~ X	~~1~~ X	~~0~~ X	0	0	1	0	0 1 1 0
and	1	~~0~~ X	~~1~~ X	~~0~~ X	0	1	0	0	0 0 0 0
or	1	~~0~~ X	~~1~~ X	~~0~~ X	0	1	0	1	0 0 0 1
slt	1	~~0~~ X	~~1~~ X	~~0~~ X	1	0	1	0	0 1 1 1

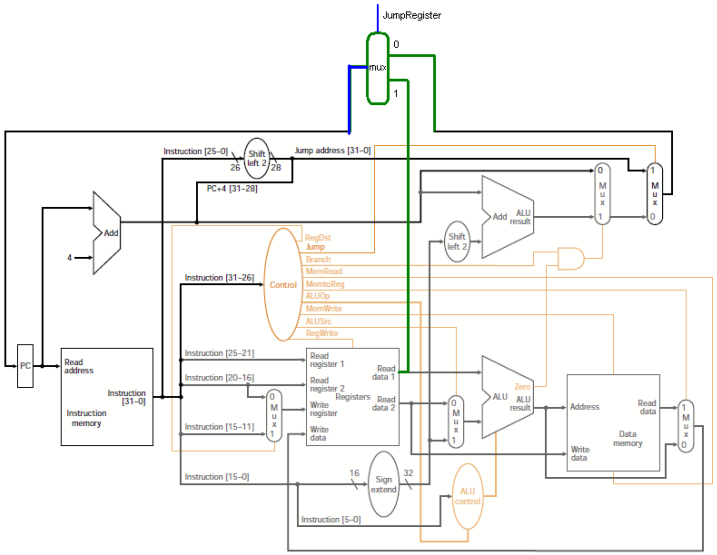
Control Signal Output

	RegDst	ALUSrc	MemTo Reg	Reg Write	Mem Read	Mem Write	Branch	ALUop	
								op1	op0
R-type	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

Complete Data Path



j and jr instruction datapath



ASCII Table

Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	0		32	20	40	[space]	64	40	100	@	96	60	140	`
1	1	1		33	21	41	!	65	41	101	A	97	61	141	a
2	2	2		34	22	42	"	66	42	102	B	98	62	142	b
3	3	3		35	23	43	#	67	43	103	C	99	63	143	c
4	4	4		36	24	44	$	68	44	104	D	100	64	144	d
5	5	5		37	25	45	%	69	45	105	E	101	65	145	e
6	6	6		38	26	46	&	70	46	106	F	102	66	146	f
7	7	7		39	27	47	'	71	47	107	G	103	67	147	g
8	8	10		40	28	50	(	72	48	110	H	104	68	150	h
9	9	11		41	29	51	)	73	49	111	I	105	69	151	i
10	A	12		42	2A	52	*	74	4A	112	J	106	6A	152	j
11	B	13		43	2B	53	+	75	4B	113	K	107	6B	153	k
12	C	14		44	2C	54	,	76	4C	114	L	108	6C	154	l
13	D	15		45	2D	55	-	77	4D	115	M	109	6D	155	m
14	E	16		46	2E	56	.	78	4E	116	N	110	6E	156	n
15	F	17		47	2F	57	/	79	4F	117	O	111	6F	157	o
16	10	20		48	30	60	0	80	50	120	P	112	70	160	p
17	11	21		49	31	61	1	81	51	121	Q	113	71	161	q
18	12	22		50	32	62	2	82	52	122	R	114	72	162	r
19	13	23		51	33	63	3	83	53	123	S	115	73	163	s
20	14	24		52	34	64	4	84	54	124	T	116	74	164	t
21	15	25		53	35	65	5	85	55	125	U	117	75	165	u
22	16	26		54	36	66	6	86	56	126	V	118	76	166	v
23	17	27		55	37	67	7	87	57	127	W	119	77	167	w
24	18	30		56	38	70	8	88	58	130	X	120	78	170	x
25	19	31		57	39	71	9	89	59	131	Y	121	79	171	y
26	1A	32		58	3A	72	:	90	5A	132	Z	122	7A	172	z
27	1B	33		59	3B	73	;	91	5B	133	[	123	7B	173	{
28	1C	34		60	3C	74	<	92	5C	134	\	124	7C	174	|
29	1D	35		61	3D	75	=	93	5D	135	]	125	7D	175	}
30	1E	36		62	3E	76	>	94	5E	136	^	126	7E	176	~
31	1F	37		63	3F	77	?	95	5F	137	_	127	7F	177	

Positive Power of 2

Exp	Val	Exp	Val	Exp	Val	Exp	Val
2^0	1	2^8	256	2^{16}	65,536	2^{24}	16,777,216
2^1	2	2^9	512	2^{17}	131,072	2^{25}	33,554,432
2^2	4	2^{10}	1,024	2^{18}	262,144	2^{26}	67,108,864
2^3	8	2^{11}	2,048	2^{19}	524,288	2^{27}	134,217,728
2^4	16	2^{12}	4,096	2^{20}	1,048,576	2^{28}	268,435,456
2^5	32	2^{13}	8,192	2^{21}	2,097,152	2^{29}	536,870,912
2^6	64	2^{14}	16,384	2^{22}	4,194,304	2^{30}	1,073,741,824
2^7	128	2^{15}	32,768	2^{23}	8,388,608	2^{31}	2,147,483,648

Negative Power of 2

Exp	Val	Exp	Val
2^{-1}	0.5	2^{-9}	0.001953125
2^{-2}	0.25	2^{-10}	0.0009765625
2^{-3}	0.125	2^{-11}	0.00048828125
2^{-4}	0.0625	2^{-12}	0.000244140625
2^{-5}	0.03125	2^{-13}	0.0001220703125
2^{-6}	0.015625	2^{-14}	0.00006103515625
2^{-7}	0.0078125	2^{-15}	0.000030517578125
2^{-8}	0.00390625	2^{-16}	0.0000152587890625