CS2100 Computer Organisation

AY 24/25 Sem 2 — github/omgeta

1. Number Systems

Weighted-positional number systems with base-R, has each position weighted in powers of R.

i. N positions give R^N values
ii. M values need $\lceil \log_R M \rceil$ positions

Conversion

Base-R→Decimal: repeated multiply by R^{n-1}

Decimal→Binary (Generalised for Base-R):

i. Whole numbers: repeated division by 2, $LSB \rightarrow MSB$
ii. Fractional: repeated multiplication by 2, $MSB \rightarrow LSB$

Binary→Octal: partition in groups of 3

Octal→Binary: convert each digit to 3 bits

Binary→Hexadecimal: partition in groups of 4

Hexadecimal→Binary: convert each digit to 4 bits

C Programming

ASCII Chars are represented with 7 bits and 1 parity bit

`char` is 1 byte, `int` + `float` are 4 bytes, `long` + `double` are 8 bytes.

Functions only modify `struct` if passed as pointer or in array with: `(*object_p).a = 1;`, `object_p->a = 1;`

Arrays in `struct`s are deep-copied (but not pointers to arrays).

Negative Numbers

Sign-and-Magnitude:

i. MSB is the sign bit (0 is positive)
ii. Range: $-(2^{N-1}-1)$ to $2^{N-1}-1$
iii. Negation: Flip MSB

1s Complement (useful for arithmetic):

i. Negated x, $-x = 2^N - x - 1$
ii. Range: $-(2^{N-1}-1)$ to $2^{N-1}-1$
iii. Negation: Invert bits
iv. Overflow: add carry to result, wrap around
v. $(b-1)s$: $-x = b^N - x - 1$

2s Complement (useful for arithmetic):

i. Negated x, $-x = 2^N - x$
ii. Range: -2^{N-1} to $2^{N-1}-1$
iii. Negation: Invert bits, then add 1
iv. Overflow: truncate
v. bs: $-x = b^N - x$

Excess-M (useful for comparisons):

i. Start at $-M = -2^{N-1}$, for N bits.
ii. Range: -2^{N-1} to $2^{N-1}-1$
iii. Express x in Excess-N: $x + N$

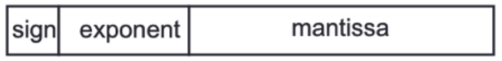
Real Numbers

Fixed-Point (limited range):

i. Reserve bits for whole numbers and for fraction, converting with 1s or 2s complement

IEEE 754 Floating-Point (more complex):

i. Sign 1-bit (0 is positive)
ii. Exponent 8-bits (excess-127)
iii. Mantissa 23-bits (normalised to $1.m \times 2^{exp}$)



single-precision: 1-bit sign / 8-bit exponent / 23-bit mantissa

2. ISA

Instruction Set Architecture (ISA) defines instructions for how software controls the hardware.

von Neumann Architecture:

i. Processor: Perform computations.
ii. Memory: Stores code and data (stored memory).
iii. Bus: Bridge between processor and memory.

Storage Architectures:

i. Stack: Operands are implicitly on top of the stack.
ii. Accumulator: One operand is implicitly in the accumulator.
iii. Memory-Memory: All operands in memory.
iv. Register-Register: All operands in registers.

Endianness (ordering of bytes in a word):

i. Big Endian: MSB in lowest address
ii. Little Endian: LSB in lowest address

Instruction Length:

i. Fixed-Length: easy fetch and decode, simplified pipelining, instruction bits are scarce
ii. Variable-Length: more flexible

Instruction Encoding, under fixed-length instructions involves extending opcode for instruction types with unused bits:

1. Minimise: maximise opcode range of instruction type with least opcode bits
2. Maximise: minimise opcode range of instruction type with least opcode bits

3. MIPS

MIPS Assembly Language:

i. Mainly Register-Register, Fixed-Length
ii. 32 registers, each 32-bits (4-byte) long
iii. 2^{30} words contains 32-bits (4-byte) long
iv. Memory addresses are 32-bits long

Arithmetic Instructions:

i. $C16, C5$ are $16, 5$ bit patterns
ii. $C16_{2S}$ is a sign-extended 2's complement
iii. NOT: `nor` with `$zero`, or `xor` with all 1s
iv. Large constants: `lui` (31:16 bits) + `ori` (15:0 bits)

Operation	Opcode in MIPS	Meaning
Addition	`add $rd, $rs, $rt`	`$rd = $rs + $rt`
	`addi $rt, $rs, C16`$_{2s}$	`$rt = $rs + C16`$_{2s}$
Subtraction	`sub $rd, $rs, $rt`	`$rd = $rs - $rt`
Shift left logical	`sll $rd, $rt, C5`	`$rd = $rt << C5`
Shift right logical	`srl $rd, $rt, C5`	`$rd = $rt >> C5`
AND bitwise	`and $rd, $rs, $rt`	`$rd = $rs & $rt`
	`andi $rt, $rs, C16`	`$rt = $rs & C16`
OR bitwise	`or $rd, $rs, $rt`	`$rd = $rs | $rt`
	`ori $rt, $rs, C16`	`$rt = $rs | C16`
NOR bitwise	`nor $rd, $rs, $rt`	`$rd = $rs ↓ $rt`
XOR bitwise	`xor $rd, $rs, $rt`	`$rd = $rs ^ $rt`
	`xori $rt, $rs, C16`	`$rt = $rs ^ C16`

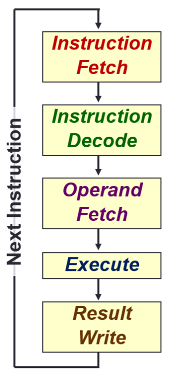
Memory Instructions:

i. `lw $rt, offset($rs)`: load contents to `$rt`
ii. `sw $rt, offset($rs)`: store contents to `$rs`
iii. Use `lb, sb` for loading bytes (like chars)

Control Instructions:

i. `beq/bne $rs, $rt, label/imm`: jump if $=/\neq$.
ii. Branch: adds `imm x 4` to `$PC = $PC + 4` on branch. Range: $\pm 2^{15}$ instructions
iii. `j label/imm`: jump unconditionally.
iv. Pseudo-direct: takes 4 MSBs from `$PC + 4`, and 2 LSB omitted since instructions are word-aligned. Range: 256-byte boundary of `$PC + 4` 4 MSBs
v. `slt $rd, $rs, $rt`: set `$rd` to 1 if `$rs < $rt`

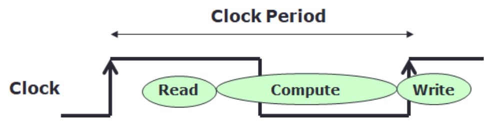
Datapath



1. **Fetch:**
   - Get instruction from memory
   - Address is in **P**rogram **C**ounter (PC) Register
2. **Decode:**
   - Find out the operation required
3. **Operand Fetch:**
   - Get operand(s) needed for operation
4. **Execute:**
   - Perform the required operation
5. **Result Write (Store):**
   - Store the result of the operation

Clock:

i. Single-Cycle: Read + Compute are done within clock period. PC Write on rising clock edge. Time taken depends on longest instruction.



ii. Multicycle: Break up execution into multiple steps (e.g. fetch, decode, alu, memory, writeback), with each step taking one cycle. Time taken depends on slowest step.

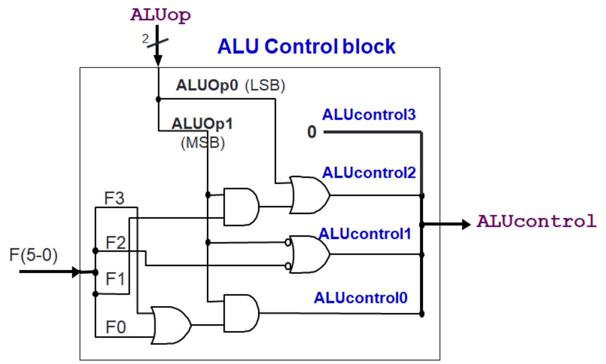
Critical Paths:

i. R-type/Immediate:
Inst Mem→RegFile→MUX (ALUSrc) →ALU→MUX (MemToReg)→RegFile
ii. `lw/sw`:
Inst Mem→RegFile→ALU→DataMem→MUX (MemToReg)→RegFile
iii. `beq`:
Inst Mem→RegFile→MUX (ALUSrc)→ALU→AND→MUX (PCSrc)

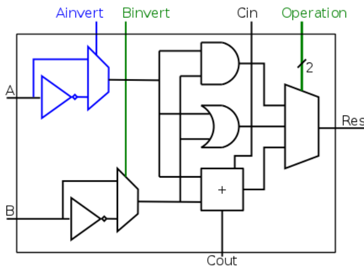
Control

Control Signals:

i. `RegDst` @Decode: Selects destination register.
(0) → `$rt` (1) → `$rd`
ii. `RegWrite` @Decode: Enable register write.
(0) → no write (1) → `WD` written to `WR`
iii. `ALUSrc` @ALU: Second 2nd ALU operand.
(0) → `RD2` (1) → `SignExt(Imm)`
iv. `ALUcontrol` @ALU: Second ALU operation.
2-bit `ALUop` from opcode + funct for 4-bits
v. `MemRead` @Memory: Enable memory read.
(0) → no read (1) → read `Addr.` into `Read Data`
vi. `MemWrite` @Memory: Enable memory write.
(0) → no write (1) → write `Write Data` into `Addr.`
vii. `MemToReg` @Writeback: Select writeback result.
(0) → `ALU result` (1) → Memory `Read data`
viii. `Branch` @Memory: Select next `$PC`.
(0) → `$PC + 4` (1) → `($PC + 4) + (imm x 4)`

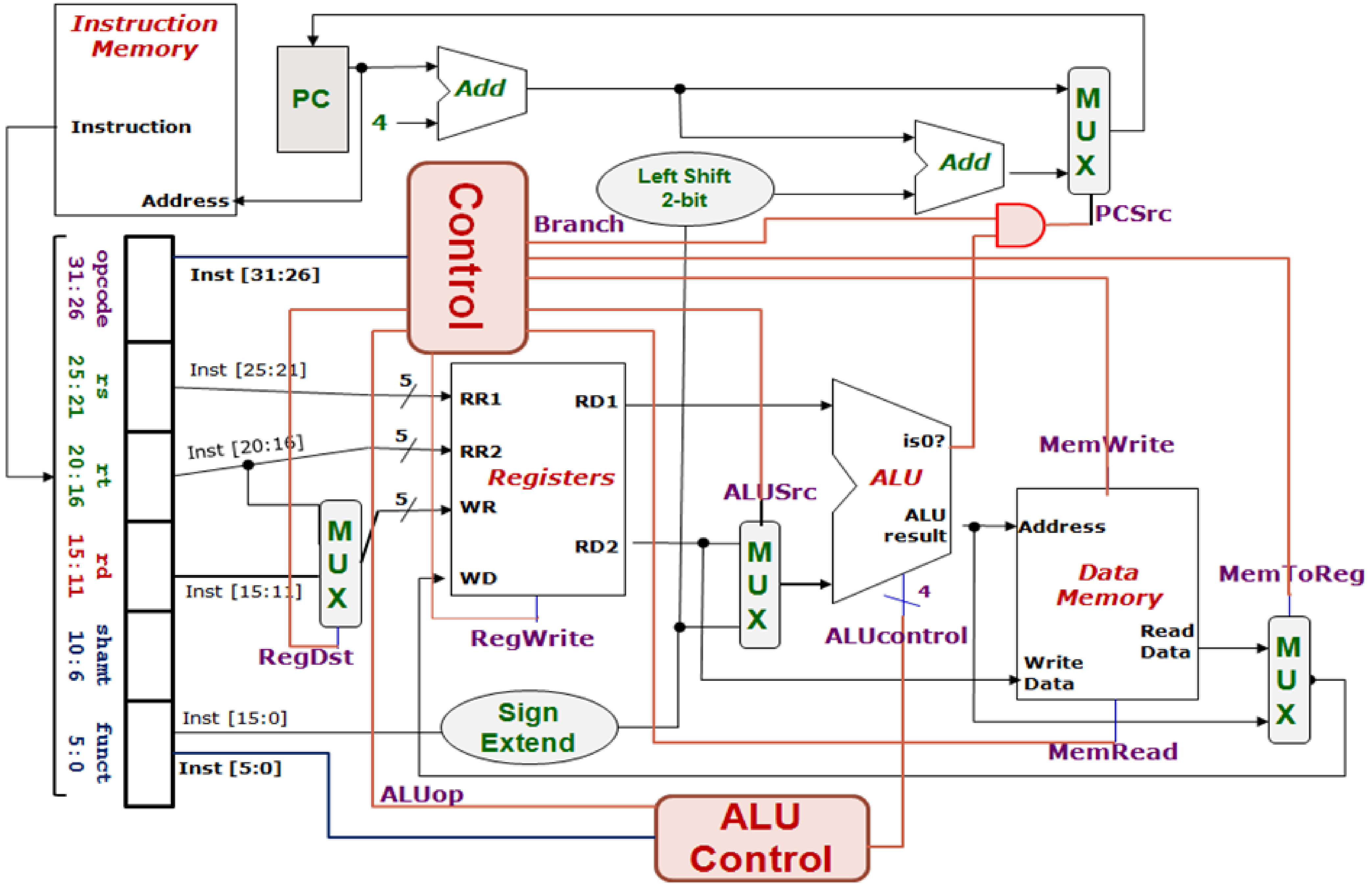


1-bit ALU



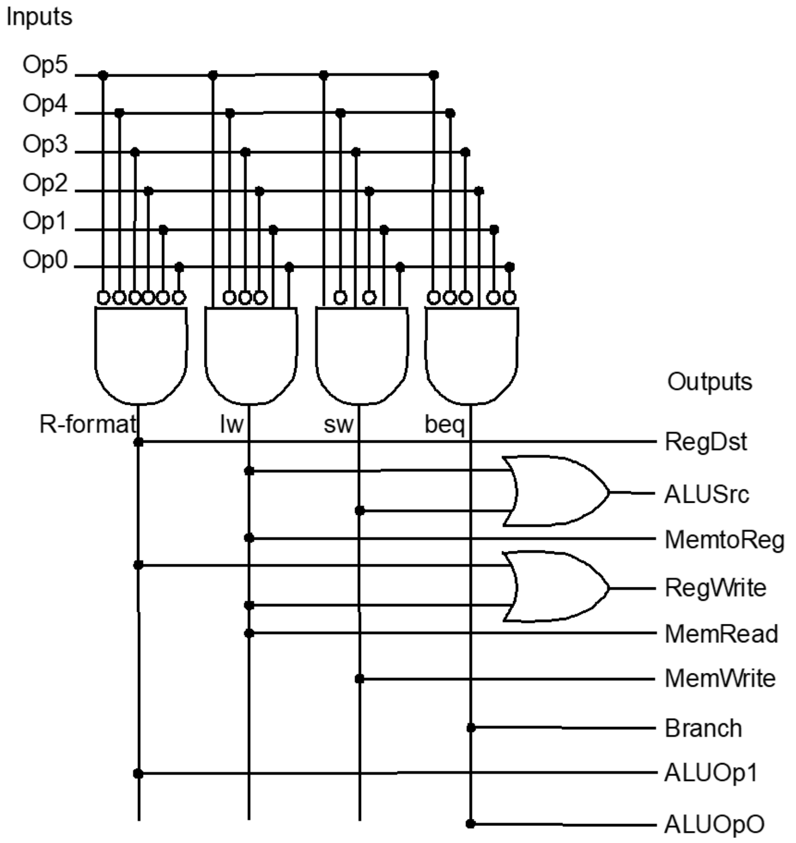
ALUcontrol			Function
Ainvert	Binvert	Operation	
0	0	00	AND
0	0	01	OR
0	0	10	add
0	1	10	subtract
0	1	11	slt
1	1	00	NOR

Reference Sheet



	Opcode					
	Op5	Op4	Op3	Op2	Op1	Op0
R-type	0	0	0	0	0	0
lw	1	0	0	0	1	1
sw	1	0	1	0	1	1
beq	0	0	0	1	0	0

	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



Opcode	ALUop	Instruction Operation	Funct field	ALU action	ALU control
lw	00	load word	XXXXXX	add	0010
sw	00	store word	XXXXXX	add	0010
beq	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	10 0000	add	0010
R-type	10	subtract	10 0010	subtract	0110
R-type	10	AND	10 0100	AND	0000
R-type	10	OR	10 0101	OR	0001
R-type	10	set on less than	10 1010	set on less than	0111

Instruction Type	ALUop
lw / sw	00
beq	01
R-type	10

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

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	

Precedence	Kinds	Operators	Associativity	Examples
1	Postfix Unary	`++`, `--`	Left to right	`x--`, `x++`
2	Prefix Unary	`+`, `-`, `++`, `--`, `&`, `*`, `(type)`, `!`	Right to left	`x = +4;`, `x = -23;`, `--x;`
3	Multiplicative Binary	`*`, `/`, `%`	Left to right	`x * y`, `z % 2;`
4	Additive Binary	`+`, `-`	Left to right	`x + y`, `z - 2;`
5	Relational	`<=`, `>=`, `<`, `>`	Left to right	`x < y`, `z >= 2;`
6	Equality	`==`, `!=`	Left to right	`x != y`, `z == 2;`
7	Conjunction	`&&`	Left to right	`x == y && x == 1`
8	Disjunction	`||`	Left to right	`x == y || x == 1`
9	Assignment	`=`, `+=`, `-=`, `*=`, `/=`, `%=`	Right to left	`x += y`, `z = 2;`