C language

Misc

- `int x = -10 % 4; //gives -2`
- $\sim$ is binary NOT, ! is boolean NOT
- When passing an array to a function, it turns into a pointer to the first element of the array.

Format specifiers

Spec.	Variable Type	Function Use
%c	char	printf / scanf
%d	int	printf / scanf
%f	float / double	printf
%f	float	scanf
%lf	double	scanf
%e	float / double	printf (scientific notation)

Examples of format specifiers used in `printf()`:

- `%5d`: to display an integer in a width of 5, right justified
- `%8.3f`: to display a real number (float or double) in a width of 8, with 3 decimal places, right justified

Escape sequences

Seq.	Meaning	Result
\n	New line	Subsequent output will appear on the next line
\t	Horizontal tab	Move to the next tab position on the current line
\"	Double quote	Display a double quote "
%%	Percent	Display a percent character %

Mixed-Type Arithmetic .

```
int m = 10/4;    // m = 2
float p = 10/4; // p = 2.0
int n = 10/4.0; // n = 2
float q = 10/4.0; // q = 2.5
int r = -10/4.0; // r = -2
```

Type Casting .

```
int aa = 6; float ff = 15.8;
float pp = (float) aa / 4; // pp = 1.5
int nn = (int) ff / aa;    // nn = 2
float qq = (float) (aa / 4); // qq = 1.0
```

Order of operations The operators are ranked according to precedence, with the top row having the highest precedence.

Operators in the same row have the same precedence, and they are evaluated in the given associativity direction.

Type	Operator	Assoc
Primary expr	`() [] . -> expr++ expr--`	L2R
Unary	`* & + - ! ~ ++expr --expr sizeof (typecast)`	R2L
Binary	`* / %`	L2R
	`+ -`	
	`<< >>`	
	`< > <= >=`	
	`== !=`	
	`&`	
	`^`	
	`|`	
	`&&`	
	`||`	
Ternary	`?:`	R2L
Assignment	`= += -= *= /= %= <<= >>=`	R2L

Number Systems

Sign extension Used in 2's complement representation. For example, to represent a 4-bit 2's complement using 8 bits,

$$0110_{2s} \rightarrow 00000110_{2s}$$

Sign extension is value-preserving. For example,

$$00000110_{2s} = 0110_{2s}$$
$$11111010_{2s} = 1010_{2s}$$

Rounding in fixed-point 2's complement fixed point representation, total 8 bits, whole number part 6 bits, fractional part 2 bits, want to represent 26.875_{10}. Without restriction,

$$26.875_{10} = 11011.111_2$$

Since we have 6 bits for whole number part, it is 011011. But we only have 2 bits in the fractional part to represent .111. Since the bit immediately after the 2nd bit is a 1, we round up, so we get 1.00.

MIPS

MIPS is big-endian, so it stores bytes in its natural order.

Get certain bits `andi` can be used to obtain specific bits from a variable.

- Place 0 into positions to be ignored, $X \cdot 0 = 0$
- Place 1 into interested positions, $Y \cdot 1 = Y$

```
andi $t0, $t0, mask
```

Force certain bits to 1 `ori` can be used to force certain bits to be 1

- Place 0 into positions to be left as-is, $X + 0 = X$
- Place 1 into interested positions, $Y + 1 = 1$

```
ori $t0, $t0, mask
```

Perform NOT

- `nor $t0, $t0, $zero`
- `xor $t0, $t0, mask`, where mask is all 1s

Load 32-bit constant into register MIPS only allows up to 16 bits in immediate, so we cannot directly load a 32-bit constant into register.

```
lui $t0, upper_16_bits
ori $t0, $t0, lower_16_bits
```

Load value from address .

```
la $t0, addr
lw $s0, 0($t0)
```

Immediate parameter

- `addi` accepts immediate in the range $[-2^{15}, 2^{15} - 1]$
- `sll`, `srl` accepts shift in the range $[0, 2^5 - 1]$
- `andi`, `ori` treat immediate as 16-bit pattern

Boolean Algebra

Gray Code

- Single bit change from one code value to the next
- Standard gray code for n bits can be obtained by listing the gray codes for $n-1$ bits in natural order, followed by the gray codes for $n-1$ bits in reverse order. Prepend 0 to the first half of the list, and prepend 1 to the second half of the list.
- Binary to Standard Gray: `x XOR (x >> 1)`
- Standard Gray to Binary: `x XOR (x >> 1)XOR (x >> 2)... XOR 0`

Karnaugh Maps

AB \ CD	00	01	11	10
00	m0	m1	m3	m2
01	m4	m5	m7	m6
11	m12	m13	m15	m14
10	m8	m9	m11	m10

3-var K-map is just the top half. 5-var K-map is two squares, with the second square offset by 16.

Truth tables

A	B	XOR	NOR	NAND	XNOR
0	0	0	1	1	1
0	1	1	0	1	0
1	0	1	0	1	0
1	1	0	0	0	1

XOR is commutative, associative. A XOR A = 0 and A XOR 0 = A.

Combinatorial/Sequential

S	R	Q(t+1)	Behaviour
0	0	Q(t)	No change
0	1	0	Reset
1	0	1	Set
1	1	indeterminate	Invalid condition

$$Q(t+1) = S + R' \cdot Q$$

D	Q(t+1)	Behaviour
0	0	Reset
1	1	Set
X	Q(t)	No change

$$Q(t+1) = D$$

J	K	Q(t+1)	Behaviour
0	0	Q(t)	No change
0	1	0	Reset
1	0	1	Set
1	1	Q(t)'	Toggle

$$Q(t+1) = J \cdot Q' + K' \cdot Q$$

T	Q(t+1)	Behaviour
0	Q(t)	No change
1	Q(t)'	Toggle

$$Q(t+1) = T \cdot Q' + T' \cdot Q$$

Pipelining

Jump is NOT a control hazard because it does not dictate which instruction to execute, rather it just causes a delay.

Time for each stage

Inst.	IF	ID	EX	MEM	WB	Total
ALU	2	1	2	-	1	6
lw	2	1	2	2	1	8
sw	2	1	2	2	-	7
beq	2	1	2	-	-	5

Reducing stall for branching

Early Branch Resolution

- Move branch decision calculation from MEM to ID stage, (usually) stall 1 cycle instead of 3.
- The early branch calculation might result in a data hazard, so it might stall for 2 cycles.
- In an even worse case, if the instruction before the branch is `lw`, we need to stall for 3 cycles even with early branching, so there is no improvement.

Branch Prediction

- Guess outcome, and if it is right, then there is no pipeline stall. Otherwise flush the wrong instructions from the pipeline.
- The delay for wrong guess depends on whether there is early branch resolution or not.

Delayed Branch

- Find non-control dependent instructions, and place them after the branch. Moving these instructions should not change program behaviour.
- If there are no such instructions, then add no-ops.

Cache

Direct Mapped Cache

⟵ Block number ⟶		
Tag	Index	Offset

Given cache block size = 2^N bytes, and number of cache blocks = 2^M. Then, offset is N bits, index is M bits, and tag is $32 - (N + M)$ bits.

Overhead Valid flag (1 bit) + Tag length, for each block

Set Associative Cache

⟵ Block number ⟶		
Tag	Index	Offset

Given cache block size = 2^N bytes, and number of cache sets = 2^M. Then, offset is N bits, set index is M bits, and tag is $32 - (N + M)$ bits.

Fully Associative Cache

⟵ Block number ⟶	
Tag	Offset

Given cache block size = 2^N bytes, and number of cache blocks = 2^M. Then, offset is N bits, and tag is $32 - N$ bits.

Overhead (Set Assoc. and Fully Assoc.) Valid flag (1 bit) + Tag length, for each block in set

Note that in a fully associative cache, all blocks are searched simultaneously.