C Programming:

Data Types:

Data Types	Bytes	Range
int	4	-2^{31} to $2^{31}-1$ (-2,147,483,648 to 2,147,483,647)
float	4	1-bit sign, 8-bit exponent, 23-bit mantissa
double	8	1-bit sign, 11-bit exponent, 52-bit mantissa
char	1	Unsigned: 0 to 255; Signed: -128 to 127

Pointers:

```
int *a_ptr; a_ptr = &a; int *a_ptr = &a;
void swap(int *a, int *b) { int temp = *a; *a = *b; *b = temp; }
```

Arrays:

```
*(arr+2) == arr[2]; arr+2 = &arr[2];
```

Strings:

```
Strings end with "\0";
scanf(%s, str): reads until \s
fgets(str, size, stdin) // reads until \n; puts(str) // ends with \n
strlen(s): returns # of chars;
strcmp(s1, s2) || strncmp(s1, s2, n): compare strings
strcpy(s1, s2) || strncpy(s1, s2, n): copies s2 into s1
```

Structures:

```
typedef struct { int marks; } student_t; student_t s1 = {80};
student s2; s2.marks = 77; // alternative to instantiate structure
void edit(student_t *s) { (*s).marks = 100; };
s2 = s1; // copies entire structure
(*player_ptr).name == player_ptr->name
```

Code Compilation:

C program (.c) -> Pre-processor -> Pre-processed code (.i) -> Compiler -> Assembly code (.asm) -> Assembler -> Object code (.o) -> Linker -> Executable (.hex)

Data and Number Systems:

IEEE-754 Conversion:

-1.101×2^{7}

1	10000110	101 00000000000000000000

Sign bit: 1; Exponent (Excess-127): 7+127=134; Mantissa: 101
Note: Fill with 0s to the back for the Mantissa

Complements:

Sign & Magnitude	Invert Sign Bit	$-2^{n-1}+1$ to $2^{n-1}-1$	+0 & -0 exists
1s Complement	Flip bits	$-2^{n-1}+1$ to $2^{n-1}-1$	+0 & -0 exists
2s Complement	Flip bits; add 1	-2^{n-1} to $2^{n-1}-1$	+0 exists

Overflow:

Checks if MSB_A == MSB_B, MSB_C must be the same for $(A + B = C)_2$
2s Complement: Ignore carry out & check for overflow
1s Complement: Add carry out to LSB_C & check for overflow

MIPS (Microprocessor without Interlocked Pipelined Stages)

32-bit constant into a register

```
lui $t0, 0xAAAA; ori $t0, $t0, 0xF0F0
```

MIPS Instructions:

```
R-format: op $rd, $rs, $rt
sll: op $rd, $rt, shamt
I-format: op $rt, $rs, immd
J-format: j immd
```

Load and Store Word:

```
lw $rt, $rs, immd == (Mem[$rt] <= Mem[$rs + immd])
sw $rt, $rs, immd == (Mem[$rs + immd] <= Mem[$rt})
```

Array Pointers:

```
Word: 4-bytes
Address of A[] => $t0 & i => $t1
sll $t3, $t1, 2 => ($t3 = i * 4);
add $t4, $t0, $t3 => ($t4 = &A[i])
lw $t5, 0($t4) => ($t5 = A[i])
```

J-format pseudo-address:

```
Ignore first 4 bits and last 2 bits of insturction
Retrieve middle 26 bits
```

ISA (Instruction Set Architecture)

Data Storage

Stack	Accumulator	Register	Memory
Push A Push B Add Pop C	Load A Add B Store C	Load R1, A Load R2, B Add R3, R1, R2 Store R3, C	Add C, A, B

Endianness:

```
Big-endian: MSB in lowest address
Little-endian: LSB in lowest address ("reverse-order")
```

Expanding Opcode:

```
Type A: 6-bit opcode & Type B: 11-bit opcode
Min: (000001 to 111111) + (000000 XXXXX)
Max: (000000) + (000001 XXXXX to 111111 XXXXX)

Type A: 3-bit opcode & Type B: 6-bit opcode &
Type C: 10-bit opcode
Min: (001 to 111) + (000001 to 000111) + (000000 XXXX)
Max: (000) + (000001) + (XXXXXX XXXX) – (000 XXX XXXX) – (000 001 XXXX)
```

Instruction Execution Cycle in MIPS:

```
Fetch : Get instruction from memory, address in PC
Decode : Find out the operation required
Operand Fetch : Get operand(s) needed for operation
Execute : Perform the required operation
 Result Write : Store the result of the operation
```

ALU Control:

$A_{INVERSE}$: 0:A, 1:A'
$B_{INVERSE}$: 0:B, 1:B'
OP: 00:AND, 01: OR, 10: ADD

ALU Control	Function
0000	and
0001	or
0010	add
0110	sub
0111	slt
1100	nor

ALU Control Unit:

MSB_3: 0
MSB_2: $ALUOP_0$ || (F_1 && $ALUOP_1$)
MSB_1: $ALUOP_1$ || (!F_2 && $ALUOP_1$)
MSB_0: $ALUOP_1$ && (F_3 || F_0)

Control Signals:

```
RegDst: 0: Inst[20:16], 1: Inst[15:11]
RegWrite:0: No register write, 1: New value will be written
ALUSrc: Operand2 = 0: Register RD2, 1: SignExt(Inst[15:0])
ALUOp: 00 : lw / sw, 01 : beq, 10 : R-type
ALUControl (ALU) : Mentioned earlier
MemRead: 0: No Mem Read, 1: Read from Address
MemWrite: 0: No Mem Write, 1: Read Data 2 -> Mem[Address]
MemToReg (RegWrite) – 1: Mem Read Data, 0: ALU Result
Branch: 0: Not Taken, 1: Taken
PCSrc: (Branch AND is Zero)
Next PC = 0: PC + 4, 1: SignExt(Inst[15:0]) << 2 + (P C + 4)
```

Control Design: Outputs

		R	lw	sw	beq
EX	RegDst	1	0	X	X
EX	ALUSrc	0	1	1	0
EX	$ALUop_1$	1	0	0	0
EX	$ALUop_0$	0	0	0	1
MEM	MemRead	0	1	0	0
MEM	MemWrite	0	0	1	0
MEM	Branch	0	0	0	1
WB	MemToReg	0	1	X	X
WB	Reg Write	1	1	0	0

Single Cycle Implementation

	R	lw	sw	beq
Inst Mem	1	1	1	1
Reg Read	1	1	1	1
ALUSrc	1	1	1	1
ALU	1	1	1	1
Data Mem		1	1	
Reg Write	1		1	

Laws of Boolean Algebra:

Identity	$A + 0 = 0 + A = A$	$A \cdot 1 = 1 \cdot A = A$
Inverse / Complement	$A + A' = 1$	$A \cdot A' = 0$
Commutative	$A + B = B + A$	$A \cdot B = B \cdot A$
Associative	$A + (B + C) = (A + B) + C$	$A \cdot (B \cdot C) = (A \cdot B) \cdot C$
Distributive	$A \cdot (B + C) = (A \cdot B) + (A \cdot C)$	$A + (B \cdot C) = (A + B) \cdot (A + C)$
Idempotency	$X + X = X$	$X \cdot X = X$
1 / 0 element	$X + 1 = 1$	$X \cdot 0 = 0$
Involution	$(X')' = X$	
Absorption 1	$X + X \cdot Y = X$	$X \cdot (X + Y) = X$
Absorption 2	$X + X' \cdot Y = X + Y$	$X \cdot (X' + Y) = X \cdot Y$
De Morgan's	$(X + Y)' = X' \cdot Y'$	$(X \cdot Y)' = X' + Y'$
Consensus	$X \cdot Y + X' \cdot Z + Y \cdot Z = X \cdot Y + X' \cdot Z$	$(X+Y) \cdot (X'+Z) \cdot (Y+Z) = (X+Y) \cdot (X'+Z)$

Minterms and Maxterms

X	Y	Minterms	Maxterms
0	0	x'·y'	x+y
0	1	x'·y	x+y'
1	0	x·y'	x'+y
1	1	x·y	x'+y'

Each minterm is the complement of the maxterm
$F = \sum m(1,4,5,6,7) = \prod M(0,2,3)$

Logic Gates:

```
NOR: (a + b)'
NAND: (a · b)'
XOR: a ⊕ b = (a · b') + (a'· b)
```

Universal Gates:

```
NAND:
x' = (x · x)'
(x · y) = ((x · y)' · (x · y)')'
(x + y) = ((x · x)' · (y · y)')'

NOR:
x' = (x + x)'
(x · y) = ((x + x)' + (y + y)')'
(x + y) = ((x + y)' + (x + y)')'
```

Simplification:

Half-Adder:

$C = (X \cdot Y)$ & $S = X \oplus Y$

Gray Code / Reflected Binary Code

Only a single bit differs from previous to the next value

K maps

```
The larger the group size, the smaller the # of literals
Prime Implicants: Maximum number of minterms
Essential PIs: Includes at least one PI not covered by any other PI
```

0	1	3	2
4	5	7	6
12	13	15	14
8	9	11	10

Combinational Circuits:

```
Full Adder:
C_OUT = X · Y + (X ⊕ Y)·C_IN; S = X ⊕ (Y ⊕ Z)
4-bit parallel adder:
X3:MSB; X0:LSB (Made out of 4 full-adders)

Magnitude Comparator:
A < B: (a'·b'·d') + (a'·c) + (b'·c·d)
A = B: (a·b·c·d) + (a· b'·c·d') + (a'·b·c'·d) + (a'·b'·c'·d')
A > B: (a·c') + (b·c'·d') + (a·b·d')
```

MSI Components

```
Decoders:
2x4 Decoder: F0: X'·Y'; F1: X'·Y; F2: X·Y'; F3: X·Y;
1-enable: if E = 1, F0 to F3 works fine.
Active-high: OR (minterm); NOR (maxterm)
Active-low: NAND (minterm); AND (maxterm)

Encoders:
4-to-2 Encoder: D0: F1 + F3; D1: F2 + F3
Priority Encoder: all before FX =  don't care

Demultiplexers:
1-to-4 demultiplexer: Y0: D·S1'·S0' to Y3: D·S1·S0

Multiplexers:
4-to-1 multiplexer: I0·m0 + I1·m1 + I2·m2 + I3·m3
Implementing functions: 1 for minterm, 0 for maxterm
```

Sequential Logic:

```
Note: 1B = 8 bits; 1KB = 2^10 bits; 1MB = 2^20 bits;
```

Latches and Flip-Flops

S	R	Q(t+1)	Status
0	0	Q(t)	NC
0	1	0	Reset
1	0	1	Set
1	1	0	Invalid

Active-High S-R Latch
Q(t+1) = S + R'·Q
S·R = 0

EN	D	Q(t+1)	Status
1	0	0	Reset
1	1	1	Set
X	X	Q(t)	NC

Active-High D Latch
If EN = 1, Q(t+1) = D

S	R	CLK	Q(t+1)	Status
0	0	X	Q(t)	NC
0	1	UP	0	Reset
1	0	UP	1	Set
1	1	UP	0	Invalid

S-R Flip Flop

D	CLK	Q(t+1)	Status
0	UP	0	Reset
1	UP	1	Set

D Flip Flop

J	K	CLK	Q(t+1)	Status
0	0	UP	Q(t)	NC
0	1	UP	0	Reset
1	0	UP	1	Set
1	1	UP	Q(t)'	Toggle

J-K Flip Flop
Q(t+1) = J·Q' + K'·Q

T	CLK	Q(t+1)	Status
0	UP	Q(t)	NC
1	UP	Q(t)'	Toggle

T Flip-Flop
Q(t+1) = T·Q' + T'·Q

Flip-Flop Excitations Tables

Q	Q+	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

Q	Q+	S	R
0	0	0	X
0	1	1	0
1	0	0	1
1	1	X	0

Q	Q+	D
0	0	0
0	1	1
1	0	0
1	1	1

Q	Q+	T
0	0	0
0	1	1
1	0	1
1	1	0

Async Inputs
Pre = HIGH =>
Q = HIGH
Pre = LOW ->
Q = LOW

State Table and Diagrams:
m flip-flops + n inputs => 2^{m+n} rows

Pipelining:

Execution Stages:
Instruction Fetch: IF
Instruction Decode and Register Read: ID
Execute / Calculate Address: EX
Access operand in Data Memory: MEM
Write Back Result into a register: WB

Pipeline Datapath:
IF/ID: Instruction Read; PC + 4
ID/EX: Get values from Register, Sign Extend, PC + 4, Write Reg Number
EX/ID: (PC + 4) + (Immd * 4); ALU result; is Zero?; Read Data 2 from register; Write Register Number
MEM/WB: ALI result; Memory read data; Write Register Number

Single-Cycle Processor:
$CT_{SEQ} = \sum_{k=1}^{N} T_k$; $Time_{SEQ} = 1 * CT_{SEQ}$

Multi-Cycle Processor:
$CT_{MULTI} = max(T_k)$; $Time_{MULTI} = 1 * Average\ CPI * CT_{MULTI}$

Pipeline Processor:
$CT_{PIPELINE} = max(T_k) + T_d$; $Time_{PIPELINE} = (I+N-1) * CT_{PIPELINE}$
$Speedup = Time_{SEQ} / Time_{PIPELINE}$

Read-After-Write (RAW): Applies for Register (ID & WB)

Data Forwarding:
AND => AND: EX to EX
LOAD => AND: MEM to EX (1 stall)
STORE => AND: MEM to EX (1 stall)

Early Branching:
Make decision in ID stage instead of MEM stage.
R => Branch, 1 stall; Load => Branch, 2 stalls.

Branch Prediction:
Assume all branches are not taken.
Correct guess no stall, wrong guess flush pipeline.

Delayed Branch:
Move non-control dependent instructions after a branch.

Cache:

Memory Access Time:
Miss Rate = (1 – Hit Rate)
Miss Penalty: Time to replace block cache + Hit Time
(Hit Rate * Hit Time) + (Miss Rate * Miss Penalty)

Write Policy:
Write-through:
Write data both to the cache and main memory
Write will operator at speed of main memory
Used a write buffer to solve the issue.

Write-back:
Only write to cache, write to main memory when cache block is evicted.
Wasteful to write back every replaced cache.
Use a dirty bit if cache content is changed, write back only if dirty bit is set to 1.

Cache Misses:
Compulsory misses: First Access to Block
Conflict misses: Collision
Capacity Misses: Blocks are discarded when cache is full

Handling Cache Misses:
Read Miss: Load Data from memory to cache to register.

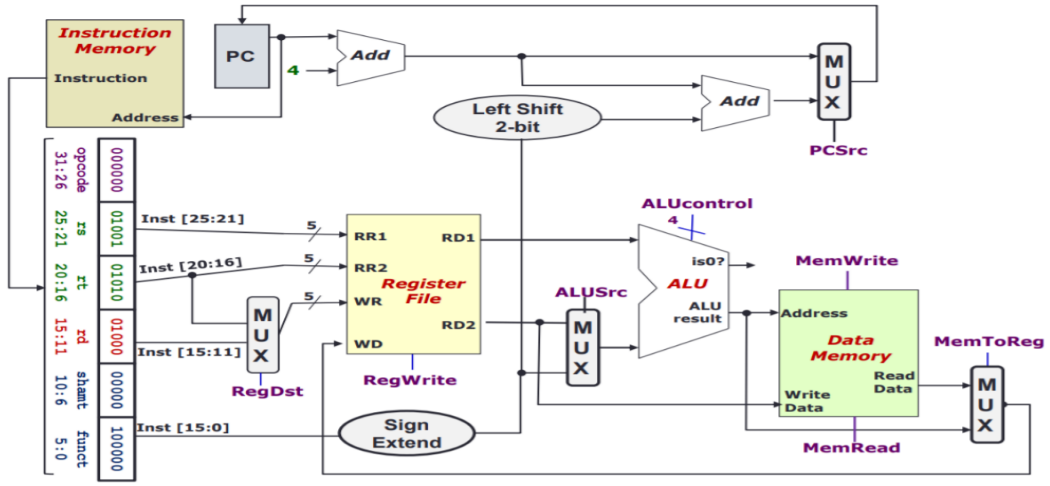
Write Miss:
Write-Allocate: Load complete block to cache, change required word in cache, write to main memory
Write-Around: Write to main memory only.

Direct Mapped Cache:
Cache Block Size: 2^N
# of Cache Blocks = 2^M = (Cache Size / Block Size)
Tag = 32 – (N + M) bits

N-way Set Associative Cache:
# of sets = 2^M = (# of Cache Block / n in N-way)

Fully Associative Cache:
Tag = 32 – N bits
Block Number = tag

Block Replacement Policy:
Least Recently Used => Temporal Locality
First In, First Out
Random Replacement
Least Frequently Used



Binary Table					
2^{-8}	0.00390625	2^0	1	2^8	256
2^{-7}	0.0078125	2^1	2	2^9	512
2^{-6}	0.015625	2^2	4	2^{10}	1024
2^{-5}	0.03125	2^3	8	2^{11}	2048
2^{-4}	0.0625	2^4	16	2^{12}	4096
2^{-3}	0.125	2^5	32	2^{13}	8192
2^{-2}	0.25	2^6	64	2^{14}	16384
2^{-1}	0.5	2^7	128	2^{15}	32768

Dec	Bin	1s	2s	Dec	Bin	1s	2s
-7	1111	1000	1001	+0	0000	0000	0000
-6	1110	1001	1010	+1	0001	0001	0001
-5	1101	1010	1011	+2	0010	0010	0010
-4	1100	1011	1100	+3	0011	0011	0011
-3	1011	1100	1101	+4	0100	0100	0100
-2	1010	1101	1110	+5	0101	0101	0101
-1	1001	1110	1111	+6	0110	0110	0110
-0	1000	1111	-	+7	0111	0111	0111

Hexadecimal Converter

R-format Converter

opcode	rs	rt	rd	shamt	funct

I-format Converter

opcode	rs	rt	immediate

	00	01	11	10		00	01	11	10		00	01	11	10		00	01	11	10
00					00					00					00				
01					01					01					01				
11					11					11					11				
10					10					10					10				