C Programming

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

Literal Values

Base	Prefix	Example
Hex	0x	0xDEAD, 0x4b
Octal	0	020 is 32, 090 is invalid
Bin	0b	0b1001, 0b100001

Impt Values

Item	Dec	Hex	Binary
'0'	48	30	00110000
'A'	65	41	01000001
'a'	97	61	01100001
$2^{15}-1$	32 767	7FFF	
$2^{16}-1$	65 535	FFFF	
$2^{31}-1$	2 147 483 647	7FFFFFFF	
$2^{32}-1$	4 294 967 295	FFFFFFFF	

Binary Arithmetic

- **1's (Diminished Radix) complement**: To negate, just flip every single bit. E.g. -11 = -01011 = 10100. Formula: Given X with N bits, $-X = 2^N - X - 1$.
- **2's complement**: To negate, flips every single bit, then +1 E.g. -11 = -01011 = 10101. Formula: Given X with N bits, $-X = 2^N - X$.
- **Generalised form $(r-1)$'s complement of $N_{(Base-R)}$**: $r^n - r^{-m} - N$ where n is number of digits and m is number of fractional digits. To get r's complement: +1 to LSB, even for fractions.
- **Bias/Excess Rep:** Excess-X means representation of X is taken to be value 0. Eg: 1001 in Excess-4 is 1.
- **Addition:** Perform binary addition. For 1's complement, add the carry-out of MSB to LSB. For 2's complement, ignore carry-out of MSB. If A and B have the same sign but result has opposite sign, overflow occurred. Additionally for 2's complement, if carry-in to MSB ≠ carry-out of MSB, overflow has occurred.
- **Powers of 2**:

Power	Value	Power	Value
-1	0.5	-6	0.015625
-2	0.25	-7	0.0078125
-3	0.125	-8	0.00390625
-4	0.0625	-9	0.001953125
-5	0.03125	-10	0.0009765625

- **Hexa to Binary**:

H	B	H	B	H	B	H	B
0	0000	4	0100	8	1000	C	1100
1	0001	5	0101	9	1001	D	1101
2	0010	6	0110	A	1010	E	1110
3	0011	7	0111	B	1011	F	1111

- **Sign extension for fixed-point numbers**:
  1's complement: extend sign bit to both left and right
  2's complement: extend sign bit to left and zeroes to right
- **IEEE754 Floating Point**:

Type	Sign	Exponent	Mantissa
Single	1-bit	8-bit (Bias-127)	23-bit
Double	1-bit	11-bit (Bias-1023)	52-bit

Sign=1 if negative. Exponent = Bias-127 $\Longrightarrow$

Excess	Decimal	Excess	Decimal
0000 0000	-127	1000 0000	1
0000 0001	-126	1000 0001	2
0111 1110	-1	1111 1110	127
0111 1111	0	1111 1111	128

Mantissa = Normalised (eg 1.1110101) then take everything after decimal point.

- **IEEE754 Special Values**:

Value	Sign	Exponent	Mantissa
Norm	±	Excess-127	1.xxxx
Zero (0)	±	0 (all 0)	0
Denorm	±	0 (rep. 2^{-126})	0.xxxx
Inf	±	255 (all 1)	0
Qui NaN	NA	255 (all 1)	≠ 0, MSB 1
Sig NaN	NA	255 (all 1)	≠ 0, MSB 0

MIPS

- **Word**: Usually 2^n bytes; The common unit of transfer between processor and memory; usually coincide with reg size, int size and instruction size for most architectures.
- **Word Alignment**: Words are aligned in memory if they begin at a byte address that is a multiple of the number of bytes in a word. Eg: if word is 4 bytes, and address is multiple of 4, then it is word aligned.
- **MIPS**: Load-store register architecture. 32 registers (32bit each), each word is 32bit, memory addresses are 32bit.
- **Byte Addressing**: MIPS uses byte addresses, so consec words differ by 4.
- **R-format**: rs source, rt 2nd source, rd receives results, $shamt$ shift amount (to be 0 for non-shift instructions)
- **I-format**: rs source, rt receives result, imm is 16bit **signed** integer
- **Branching**: If branch is **Not Taken**, $PC = PC + 4$. If branch is **Taken**, $PC = (PC + 4) + (imm * 4)$ imm can be positive or negative
- **J-format**: Add 2 0s to the right of the 26bit immediate value (as all instr are multiple of 4). Preserve the first 4 bits of (PC+4), replace the remaining 28 bits with the shifted immediate value. Max jump range is 256MB. Use jr if jumping across the boundary.

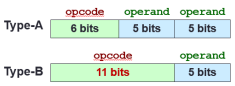
Inst Set Arch (ISA)

- **Complex Instruction Set Computer (CISC)**:
  Example: x86-32 (IA32)
  Single instruction performs complex operation.
  VAX arch had an instr to multiply polynomials
  Smaller program size as memory was premium
  Complex implementation, no room for hardware optimization
- **Reduced Instruction Set Computer (RISC)**:
  Example: MIPS, ARM
  Keeps the instr set small and simple, makes it easier to build/optimize hardware.
  Burden on software to combine simpler operations to imple high-level language statements
- **Storage Architecture**:
  Stack Architecture
  Operands are implicitly on top of the stack.
  Accumulator Architecture
  One operand is implicitly in the accumulator (a special register). All results will be at the accumulator.
  General-purpose register Architecture
  Only explicit operands.
  - Register-memory architecture (one operand in memory)
  - Register-register (load store) architecture.
  Memory-memory Architecture
  All operands are in memory, no fetch and store.

Stack	Accumulator	Register (load-store)	Memory-Memory
Push A	Load A	Load R1,A	Add C, A, B
Push B	Add B	Load R2,B	
Add	Store C	Add R3,R1,R2	
Pop C		Store R3,C	

C = A+B

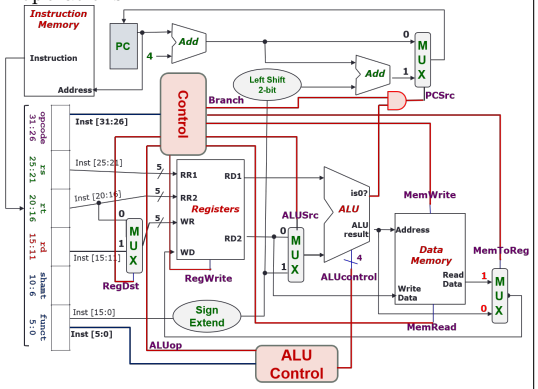
- **Endianness**:
  The relative ordering of the bytes in a multiple-byte word stored in memory.
  Big-endian: Most significant byte stored in the lowest address. (Eg: MIPS)
  Little-endian: Least significant byte stored in the lowest address. (Eg: Intel 80x86)
- **Instruction Encoding**: Instructions can be Fixed Length or Variable Length.
  We can 'extend' opcode for a subset of instructions:



Answer:
$1 + (2^6 - 1) \times 2^5$
$= 1 + 63 \times 32$
$= 2017$

Datapath and Control

- Collection of components that process data. Performs the arithmetic, logical and memory operations.



- **Instruction Execution Cycle**
  - **Fetch**: Get instruction from memory using address from Program Counter (PC).
  - **Decode**: Find out the operation required.
  - **Operand Fetch**: Get the operand(s) needed for operation.
  - **Execute**: Perform the required operation.
  - **Result Write (Store)**: Store the result of the operation.
- **MIPS Instruction Execution**: Merge Decode and Fetch - decode is simple for MIPS. Split Execute into ALU (Calculation) and Memory Access.

	`add $3, $1, $2`	`lw $3, 20($1)`	`beq $1, $2, ofst`
Fetch	Read inst. at [PC]	Read inst. at [PC]	Read inst. at [PC]
Decode & Operand Fetch	○ Read [$1] as *opr1* ○ Read [$2] as *opr2*	○ Read [$1] as *opr1* ○ Use *20* as *opr2*	○ Read [$1] as *opr1* ○ Read [$2] as *opr2*
ALU	*Result* = *opr1* + *opr2*	*MemAddr* = *opr1* + *opr2*	*Taken* = (*opr1* == *opr2*)? *Target* = (PC+4) + ofst×4
Memory Access		Use *MemAddr* to read from memory	
Result Write	*Result* stored in $3	*Memory* data stored in $3	if (*Taken*) PC = *Target*

- **Control Unit**: Combinational logic to generate control signals.

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

- **Arithmetic Logic Unit (ALU)**: Combinational logic to implement arithmetic and logical operations. Inputs 2 32bit numbers, output a 32bit result. 4bit control signal.

Opcode	ALUop	Instruction Operation	Funct field	ALU action	ALU control
lw	00	load word	NA	add	0010
sw	00	store word	NA	add	0010
beq	01	branch equal	NA	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

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

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

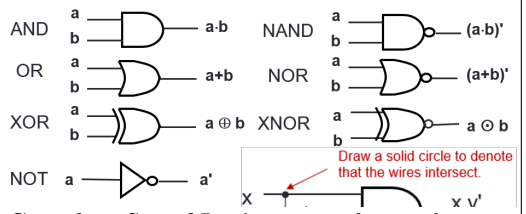
Boolean Algebra

- **Laws**:
  - **Identity**: $A + 0 = A$; $A \cdot 1 = A$
  - **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**: $X + X = X$; $X \cdot X = X$
  - **One Element/Zero Element**: $X + 1 = 1$; $X \cdot 0 = 0$
  - **Involution**: $(X')' = X$
  - **Absorption**: $X + X \cdot Y = X$; $X \cdot (X + Y) = X$; $X + X' \cdot Y = X + Y$; $X \cdot (X' + Y) = X \cdot Y$
  - **DeMorgans'**: $(X + Y)' = X' \cdot Y'$; $(X \cdot Y)' = X' + Y'$; Can be generalized to more than 2 variables.
  - **Concensus**: $(X \cdot Y) + (X' \cdot Z) + (Y \cdot Z) = (X \cdot Y) + (X' \cdot Z)$; $(X + Y) \cdot (X' + Z) \cdot (Y + Z) = (X + Y) \cdot (X' + Z)$
- **Duality**: A boolean equation **remains valid** if all the AND and OR operators are interchanges and all the identity elements 0 and 1 are interchanged. (Eg: $(x + y + z)' = x' \cdot y' \cdot z'$ can be interchanged into $(x \cdot y \cdot z)' = x' + y' + z'$)
- **Minterm** is a product term that represents when a fn should be '1'. $m5 = x \cdot y' \cdot z$ as $5 = 0b101$

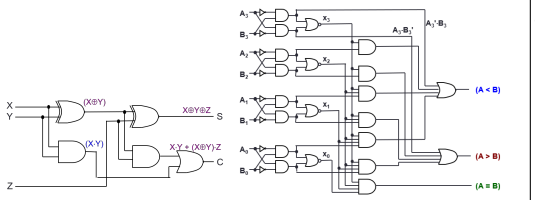
- **Maxterm** is a sum term that represents when a fn should be '0'. $M5 = x' + y + z'$ as $5 = 0b101$
- Each minterm is the *complement* of the corresponding maxterm. $m5' = M5$
- **Gray Code** (reflected binary code): single bit change from one value to next.

Circuit Components

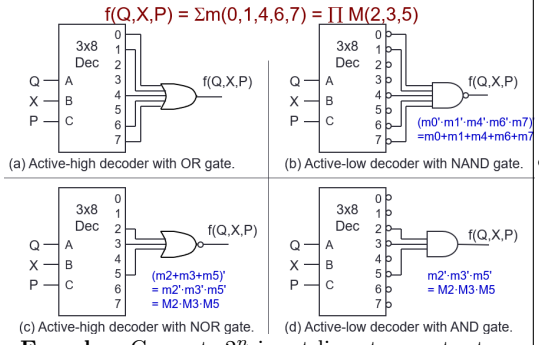
- **Logic Gates**:



- **Complete Set of Logic** are set of gates that can implement any boolean logic. $\{AND, OR, NOT\}$, $\{NAND\}$, $\{NOR\}$ are such complete sets.
- **Full Adder and Magnitude Comparator**

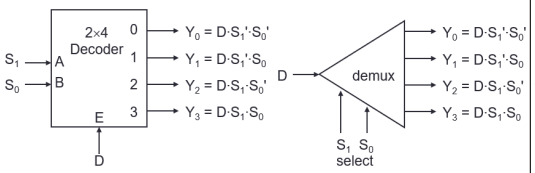


- **Decoder**: Converts binary information from n input lines to up to 2^n output lines.

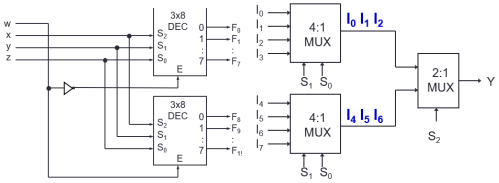


(a) Active-high decoder with OR gate. (b) Active-low decoder with NAND gate.
(c) Active-high decoder with NOR gate. (d) Active-low decoder with AND gate.

- **Encoder**: Converts 2^n input lines to n output lines (opp of decoder). In a priority encoder, the input line with lowest number takes precedence.
- **Demultiplexer**: directs data from 1 input line to one of the 2^n output lines based on the select value. Identical to a decoder with enable.



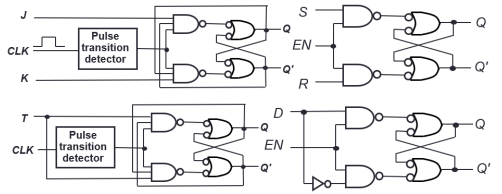
- **Multiplexer**: steers one of the 2^n input to a single output line based on a selector
- Decoders and Multiplexers can be combined to form larger ones.



- **Karnaugh Maps**
  - **Implicant**: Product term that could be used to cover minterms of the function (all 1 or X)
  - **Prime Implicant**: Product term obtained by combining the max poss number of minterms from adj sq in the map (ie: biggest grouping)
  - **Essential PI**: A PI that includes a minterm that is not covered by any other PI. (ie: a PI that you are forced to select)
  - **SOP**: Select PI such that all 1 are covered.
  - **POS**: Use K-map of inverted function (or use the zeros). Find SOP then inv it using DeMorgan's.

Sequential Circuits

- **Flip Flop + Excitation Table**



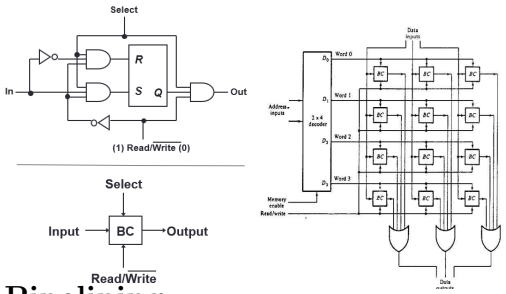
S	R	Q(t+1)	Comments	Q	Q'	S	R
0	0	Q(t)	No change	0	0	0	X
0	1	0	Reset	0	1	1	0
1	0	1	Set	1	0	0	1
1	1	?	Unpredictable	1	1	X	0

D	Q(t+1)		Q	Q'	D
0	0	Reset	0	0	0
1	1	Set	0	1	0
			1	0	1
			1	1	1

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

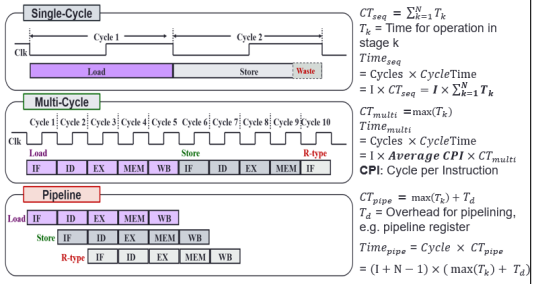
T	Q(t+1)		Q	Q'	T
0	Q(t)	No change	0	0	0
1	Q(t)'	Toggle	0	1	1
			1	0	1
			1	1	0

- **Memory Cell**: **Static** RAM use flip-flops as the memory cells (below). **Dynamic** RAM use capacitor charges to represent data, need to be constantly refreshed.



Pipelining

- **MIPS Pipeline Stages**:
  - IF: Instruction Fetch
  - ID: Instruction Decode and Register Read
  - EX: Execute/calculate (ALU operations)
  - MEM: Read/Write an operand in memory (or decide branch)
  - WB: Write back to register
- **Processor Performance**:

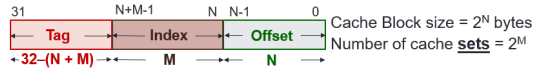


- **Speedup**: $Speedup_{pipe} = \frac{Time_{seq}}{Time_{pipe}}$
- **Ideal Speedup**: Every stage takes the same time; no pipeline overhead; num of instr much larger than num of stages; Tends to N as I gets larger
- **Hazards**:
  - **Structural**: Simultaneous use of a hardware resource
  - **Data**: Data dependencies between instructions
  - **Control**: Change in program flow
- **Read-After-Write (RAW) Hazards**:
  - **Original**: $WB \rightarrow ID$ (delay = 2)
  - **Forwarding (non-load)**: $EX \rightarrow ID$ (delay = 0)
  - **Forwarding (load)**: $MEM \rightarrow ID$ (delay = 1)
- **Control Hazards**: Outcome known at MEM
  - **Original**: $MEM \rightarrow IF$ (delay = 3)
  - **Early Branching**: $ID \rightarrow IF$ (delay = 1) Add hardware to compare registers in stage 2, decide branch outcome at ID stage instead. However, data is req before ID stage, like in IF
  - **Early Branching with RAW**: *add*, *beq*: $EX \rightarrow IF$ w. data fwd (delay = 1+1) *lw*, *beq*: $MEM \rightarrow IF$ w. data fwd (delay = 2+1)
  - **Branch Prediction**: Predict not taken or taken and continue executing instead of stalling. Add hardware to flush if wrong prediction.
  - **Delayed Branching**: Move **non-control dependent** instructions into the X slots following a branch, to be exec **regardless of the branch outcome**. Program correctness must be preserved, add *nop* if no such instr. *Usually*, $X = 1$ with early branching. Compiler must be smart enough to reorder. Usually, can find such instr 50% of the time.

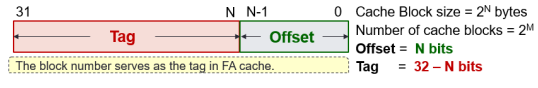
Memory and Cache

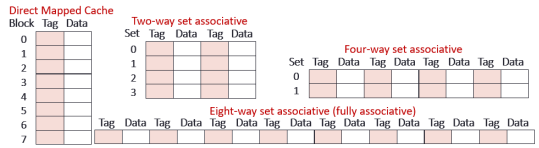
- **Hierarchy**: HDD → DRAM (main mem) → SRAM (cpu cache) → Registers
- **Locality**: Program accesses only a small portion of the memory address space within a small time interval
  - **Temporal**: Same item tends to be re-referenced
  - **Spatial**: Nearby items will tend to be referenced
- **Avg Access Time**: $Rate_{Hit} * Time_{Hit} + (1 - Rate_{Hit}) * Time_{Miss}$
- **Read Misses**:
  - **Compulsory/Cold Start**: First access to a block
  - **Conflict**: Same index gets overwritten (previously got evicted) **Only for direct and set associative**
  - **Capacity**: Cache cannot contain all blocks needed **Only for fully associative**
- **Write Policy**:
  - **Write Through**: Write data both to cache and to main memory
  - **Write-back**: Only write to cache. Write to main memory only when cache block is replaced. Uses dirty bit to flag if the data in cache is updated.
- **Write Miss Handling**:
  - **Write Allocate**: Load to cache, change in cache, actual write depend on write policy
  - **Write Around**: Write direct to main memory
- **Direct Mapped Cache**: Overhead: Valid Tag (1-bit), Tag (? bits) (+ Dirty Flag)

- **N-way Set Associative Cache**: Cache contains a number of sets, each set contains N cache blocks



- Rule of Thumb: A direct-mapped cache of size N has about the same miss rate as a 2-way set associative cache of size $N/2$
- **Fully Associative Cache**: A mem block can be placed in any loc of the cache. Increase temporal locality but however, need to search all cache blocks for mem access. **No conflict miss** since data can go anywhere.





- **Block Size** Larger block size inc spatial locality but also inc miss penalty (inc time to fill block). If block size is too big wrt cache size, miss rate inc too.
- **Block Replacement Policy**:
  - **Least Recently Used**: Remove the block that has not been accessed the longest. (inc. temporal locality)
  - **Write-back**: Only write to cache. Write to main memory only when cache block is replaced. Uses dirty bit to flag if the data in cache is updated.