NUMBER SYSTEM

- Byte: 8 bits Nibble: 4 bits. Word: Multiple of bytes
- N bits can represent up to 2^N values
- To represent M values, $\lceil \log_2 M \rceil$ bits are required
- **Decimal int. to binary**: Repeated div. by 2
- **Fraction to binary**: Repeated mult. by 2

NEGATIVE NUMBERS

- Sign-and-magnitude: 0 +; 1 –
- 1's complement: $-x = 2^n - x - 1$ (flip all bits)
- 2's complement: $-x = 2^n - x$ (flip bits, +1)
- Addition: ignore carry out of MSB for 2's; + 1 to result if there is carry out of MSB for 1's

EXCESS REPRESENTATION

Range of values to be distributed **evenly** between +ve and –ve values.

2 in excess-127 = 2 + 127 = 129 = $(1000\,00001)_2$

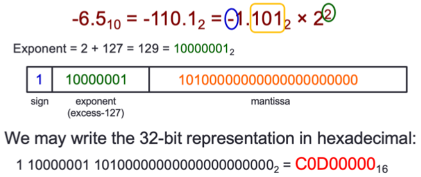
n-bits needed to represent excess 2^{n-1}

IEEE 754 Floating-Point Representation

The base (radix) is assumed to be 2.

Single-precision (32 bits): 1-bit sign, 8-bit exponent with bias 127 (excess-127), 23-bit mantissa

Double-precision (64 bits): 1-bit sign, 11-bit exponent with bias 1023 (excess-1023), and 52-bit mantissa



$-6.5_{10} = -110.1_2 = -1.101_2 \times 2^2$

Exponent = 2 + 127 = 129 = 10000001_2

sign	exponent (excess-127)	mantissa
1	10000001	10100000000000000000000

We may write the 32-bit representation in hexadecimal:

1 10000001 $10100000000000000000000_2$ = $C0D00000_{16}$

NUMBER OF INSTRUCTIONS

A	
Opcode 4 bits	6 bits
B	
Opcode 6 bits	4 bits
C	
Opcode 8 bits	2 bits

- Maximum: $(2^{\text{Highest}}) - (2^{\text{Highest - second highest}}) - (2^{\text{Highest - smallest}}) + 1 + 1 = 2^8 - 2^{8-6} - 2^{8-4} + 1 + 1 = 238$
- Minimum: $(2^{\text{Smallest}} - 1) + (2^{\text{Second highest - smallest}} - 1) + (2^{\text{Highest - second highest}}) = (2^4 - 1) + (2^{6-4} - 1) + (2^{8-6}) = 22$

Radix diminished complement

(r – 1) complement of N = $r^n - r^{-m} - N$

n – number of integer digits of N
m – number of fractional digits of N
where N is represented as a binary number

Logical operation	C operator	MIPS
Shift left	<<	`sll`
Shift right	>>	`srl`
Bitwise AND	&	`and, andi`
Bitwise OR	|	`or, ori`
Bitwise NOT*	~	`nor`
Bitwise XOR	^	`xor, xori`

NOT: `nor $t0, $t0, $zero`
`xor $t0, $t0, $t2` ($t2 contains all 1)

Loading 32-bit const. in register: 0xAAAAF0F0
1) `lui $t0, 0xAAAA` (lower 16 bits are 0)
2) `ori $t0, $t0, 0xF0F0` (set lower bits)

Get bit: AND with positive *mask*
1 for bits to keep, 0 for everything else

Set bits: OR with positive mask
1 for bits to make 1, 0 for everything else

Clear bits: AND with negative mask
0 for bits to make 0, 1 for everything else

Flip bits: XOR with positive mask
1 for bits to flip, 0 for everything else

Positive mask: Use 1 for the bits you want
Negative mask: Use 0 for the bits you want

R-FORMAT

op	rs	rt	rd	shamt	fun
6	5	5	5	5	6
`arith $A, $B, $C`					
0	B	C	A	0	XX
`shift $B, $C, shamt`					
0	0	B	C	shamt	XX

I-FORMAT

op	rs	rt	immediate
6	5	5	16
`addi $A, $B, XX`			
8	B	A	XX
`lw $A, XX($B)`			
35	B	A	XX
`beq $A, $B, LABEL`			
4	A	B	Number of instr starting from next instr

J-FORMAT

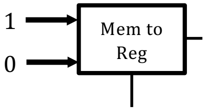
op	address
6bits	26

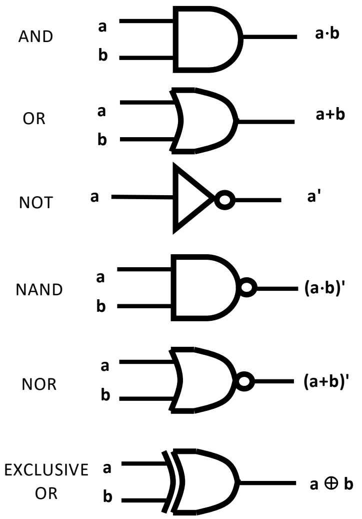
Target address (32 bits)

- Drop the first 4 and last 2 bits: 0000 **address** 00
- Since it uses the first 4 bits of PC as the first 4 of the address, we cannot jump beyond 256 MB boundary
- If instruction is at addr 0x10000000, it cannot jump above itself.

Control signals		
Control Signal	0	1
RegDst	rt	rd/ imm[15:11]
ALUSrc	RD2	SignExt(Imm)
MemToReg	ALU Result	ReadData
PCSrc	PC + 4	(PC+4) + SignExt(Imm)*4
RegWrite	-	Enable writing of reg
MemRead	-	Read from mem
MemWrite	-	Write to mem
Branch	Not branch instr	Is a branch instr

- beq is an exception I-instruction, ALUsrc = 0
- PCSrc = Branch AND isZero
- The control signal coming out from Control: Branch
- For R-Instr → RegDst: 1; ALUSrc: 0; MemToReg: 0





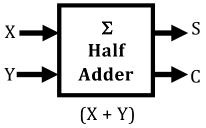
- {AND, OR, NOT}, {NAND}, {NOR} are complete sets of logic

SOP, PI and EPI

- Prime implicant (PI): a product term obtained by combining the maximum possible number of minterms from adjacent squares in the map. (That is, it is the biggest grouping possible.)
- Essential prime implicant (EPI): a prime implicant that includes at least one minterm that is not covered by any other prime implicant.

HALF ADDER

- 2 single bit inputs → 2 bit output

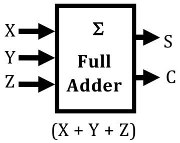


X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

- C = X·Y
- S = = X⊕Y

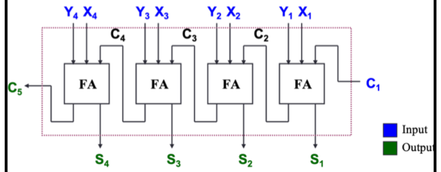
FULL ADDER

X	Y	Z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



- C = X·Y + (X⊕Y)·Z
- S = = X⊕(Y⊕Z)

- Gate-level design (with logic gates)
  - Half adder, full adder
- Block-level design (with functional blocks)
  - 4-bit parallel adder:



- Magnitude comparator (compares 2 unsigned values A and B)

CIRCUIT DELAYS

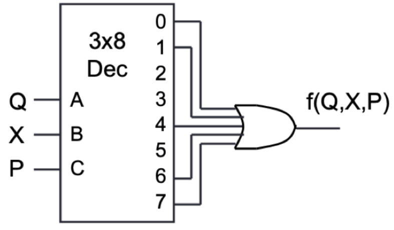
- An n-bit ripple-carry parallel adder will experience the following delay times:
- S_n = ((n – 1)2 + 2) t
- C_n+1 = ((n – 1)2 + 3) t
- Maximum delay = ((n – 1)2 + 3) t

DECODER

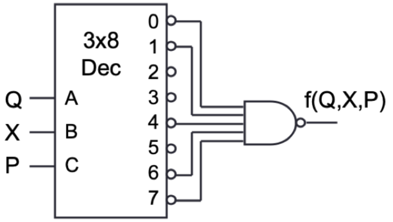
- Convert binary information from n input lines to (a maximum of) 2^n output lines
- 2x4 decoder which selects output line based on 2-bit code supplied:

X	Y	F_0	F_1	F_2	F_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

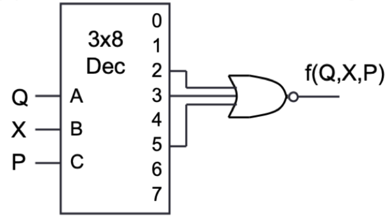
- Normal = active high; negated = active low
- Decoders with enable control signal (one-enable: only activated when E = 1)
- 3x8 decoder: F(x, y, z)
- →F_0 = x'·y' ·z' (0, 0, 0) … F_7 = x·y·z (1, 1, 1)



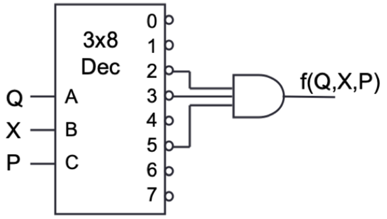
(a) Active-high decoder with OR gate.



(b) Active-low decoder with NAND gate.



(c) Active-high decoder with NOR gate.



(d) Active-low decoder with AND gate.

ENCODER

- Given a set of input lines, of which exactly one is high and the rest are low, the encoder provides a code that corresponds to that high input line
  - For 8x3 encoder:
  - Input D_0 = 1, the rest 0 → output 0 0 0 ...
  - Input D_7 = 1, the rest 0 → output 1 1 1
- Priority encoder:
  - If 2 or more inputs = 1, input with highest priority takes precedence
  - If all inputs are 0, this input combination is invalid

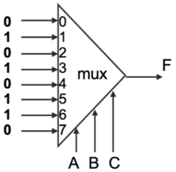
Inputs				Outputs		
D_0	D_1	D_2	D_3	f	g	V
0	0	0	0	X	X	0
1	0	0	0	0	0	1
X	1	0	0	0	1	1
X	X	1	0	1	0	1
X	X	X	1	1	1	1

DEMULTIPLEXER

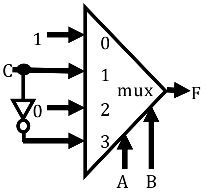
- Given an input line and a set of selection lines, a demultiplexer directs data from the input to one selected output line
- Demultiplexer = decoder with enable

MULTIPLEXER

- Has a number of input lines, selection lines and one output line
- Steers one of the 2^n inputs to a single output line, using n selection lines
- 4-to-1 multiplexer:
  - Y = $\mathbf{I_0 \cdot (S_1' \cdot S_0') + I_1 \cdot (S_1' \cdot S_0) + I_2 \cdot (S_1 \cdot S_0') + I_3 \cdot (S_1 \cdot S_0)}$
  - m_0 = 0, 0; m_3 = 1, 1
  - = $\mathbf{I_0 \cdot m_0 + I_1 \cdot m_1 + I_2 \cdot m_2 + I_3 \cdot m_3}$
- Output of multiplexer is sum of the product of data lines and selection lines
- e.g. F(A,B,C) = ∑ m(1,3,5,6)
- Using smaller multiplexers



A	B	C	F	
0	0	0	1	1
0	0	1	1	
0	1	0	0	C
0	1	1	1	
1	0	0	0	0
1	0	1	0	
1	1	0	1	Ç
1	1	1	0	



SEQUENTIAL CIRCUITS

- Synchronous: outputs change only at specific time
- Asynchronous: outputs change at any time
- Multivibrator: a class of sequential circuits
  - Bistable (2 stable states)
  - Monostable (1 stable state)
  - Astable (no stable state)
- Memory element: a device which can remember value indefinitely, or change value on command from its inputs
- Two types of activation:
  - Pulse (for latches, ON = 1 and OFF = 0),
  - Edge (for flip-flops,
  - positive triggered: ON = from 0 to 1 and OFF = other time,
  - negative triggered: ON = from 1 to 0, and OFF = other time)

S-R LATCH

- Inputs: S and R; Outputs: Q and Q'
- Q=HIGH, latch in SET; Q=LOW, latch in RESET
- Active-high input S-R latch (NOR gate latch)

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

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

$S \cdot R = 0$

GATED D LATCH

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

When EN=1, $Q(t+1) = D$

FLIP-FLOPS

- Change states either at positive (rising) edge or at negative (falling) edge
- Negative → a circle at flip-flop C input ">"
- **S-R Flip Flop**

S	R	CLK	Q(t+1)	Comments
0	0	X	Q(t)	No change
0	1	↑	0	Reset
1	0	↑	1	Set
1	1	↑	?	Invalid

X = irrelevant ("don't care")

↑ = clock transition LOW to HIGH

- **D-Flip Flop**

D	CLK	Q(t+1)	Comments
1	↑	1	Set
0	↑	0	Reset

↑ = clock transition LOW to HIGH

- **J-K Flip Flop**

J	K	CLK	Q(t+1)	Comments
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 Flip Flop**

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

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

ASYNCHRONOUS INPUTS

- Affect the state of the flip-flop independent of the clock; present (PRE), clear (CLR), direct reset (RD)
- PRE = HIGH, Q = HIGH
- CLR = HIGH, Q = LOW
- Normal operation mode with both PRE and CLR are low

STATE DIAGRAMS

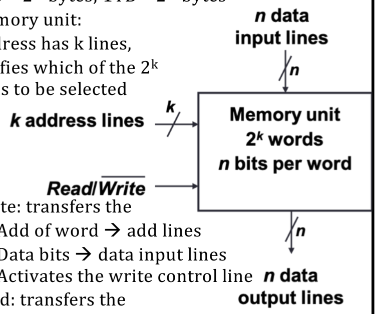
- Each state denoted by a circle
- Each arrow denotes a transition of circuit
- A label, a/b, denotes input/output
- m flip-flops → up to 2^m states

EXCITATION TABLES

- Characteristic tables used in analysis
- Excitation tables used in design
  - Given the required transition from present to next state, determine flip-flop inputs

MEMORY

- 1 byte =8bits; 1KB = 2^{10}bytes; 1MB = 2^{20}bytes
- 1GB = 2^{30}bytes; 1TB = 2^{40}bytes
- Memory unit:
- Address has k lines, specifies which of the 2^k words to be selected

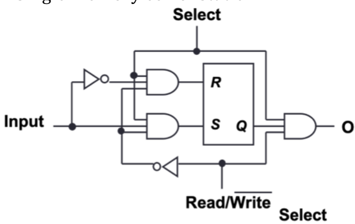


- Write: transfers the
  - Add of word → add lines
  - Data bits → data input lines
  - Activates the write control line
- Read: transfers the
  - Add of word → add lines
  - Activates the read control line

Memory Enable	Read/Write	Memory Operation
0	X	None
1	0	Write to selected word
1	1	Read from selected word

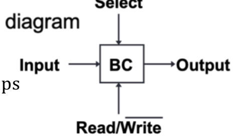
MEMORY CELL

- Two types of RAM:
  - Static RAM use flip-flops as the memory cells
  - Dynamic RAMs use capacitor charges to represent data
- Single memory cell of static RAM:



Logic diagram

- Memory array:
  - Array of RAM chips



Block diagram

PIPELINING

- Doesn't help with latency of single task, helps throughout
- MIPS pipeline stages:
  - IF: instruction fetch
  - ID: instruction decode and reg read
  - EX: execute an operation/ calc. an add
  - MEM: access an operand in data memory
  - WB: write back the result into a register
- Group control signals according to pipeline
  - EX stage: RegDst, ALUSrc, ALUop
  - MEM stage: MemRead, MemWrite, Branch
  - WB stage: MemToReg, RegWrite

Comparison of performance

- Single-cycle processor
  - Cycle time = max($\sum^{N}_{k=1} T_k$)
  - T_k = time for operation in stage k
  - N = number of stages
  - Time for I instr.: I x CT
- Multi-cycle processor
  - Cycle time = max(T_k)
  - Time for I instr.: I x average CPI x CT
- Pipelining processor
  - Cycle time = max(T_k) + T_d
  - T_d = overhead for pipelining (pipeline reg)
  - Cycles needed for I instr. = **ideal pipeline**: I + N – 1
  - Execution time for I instr.: (I + N – 1) x CT

Ideal speedup

- Speedup = $\frac{time\ seq}{time\ pipeline}$ = N
- Assumptions
  - Every stage takes same time
  - No pipeline overhead
  - I >> N

Pipeline hazards

- Structural: simultaneous use of hardware
  - Stall, or use separate memory (data & inst)
- Data: data dependencies between instruction
- No data dependency: WAR, WAW
- Data dependency: RAW
  - Solution: forward the result to later instructions before going in register file
- LOAD instruction
  - Cannot solve with forwarding; data is needed before it is produced
  - Stall the pipeline
- Control: change in program flow
  - An instruction j is control dependent on i if i controls whether or not j executes
  - Early branch resolution (move decision earlier, in ID instead of MEM) → one cycle delay needed; bad if load then branch
  - Branch prediction (correct guess = ideal pipeline) → wrong prediction: delay till branch is known to be taken or not
  - Delayed branching (move non-control dependent instr into X slots) → best if have instr to move

CACHE

- Hit: data in cache; hit rate: fraction of memory access that hit hit time: time to access cache
- Miss: data not in cache; miss rate: 1 – hit rate; miss penalty: time to replace cache block + hit time
- Average access time =
hit rate x hit time + (1 – hit rate) x miss penalty

- Cache block size = 2^N bytes
- Number of cache blocks = 2^M = $\frac{cache\ size}{cache\ block\ size}$
- Offset = N bits
- Set index/ index = M bits = $\frac{no.of\ cache\ blocks}{n-way\ associative}$
- Tag = 32 – (N + M) bits
  - *32 is derived from number of bits needed to represent memory size
- Cache hit when:
  - Valid[index] &&
  Tag[index] == Tag[memory address]

Types of cache misses:

- Compulsory/ cold misses
  - First access to a block
- Conflict/ collision misses
  - Several blocks mapped to same block
- Capacity misses
  - Blocks are discarded from cache

Set associative cache

- N-way set associative cache:
  - Memory block can be placed in a fixed number of locations in the cache
  - Each set contains N cache blocks
- Cache set index = block number % number of cache sets

Fully associative cache

- A memory block can be placed in any location in the cache (no index, only tag and offset)
- No conflict miss, because data can go anywhere