Finals Notes

CS2100 Notes

for final, AY24-25 Sem 1

zaidan sani

Quick MIPS Shortcuts

(calculator needs to be able to handle 32-bit values)

Calculating Jump Instructions Encoding

Given an address 0xABCDEFGH (where A…H are hexadecimal digits):

- `jump` : 0x08000000 + BCDEFGH/4 in `HEX` mode

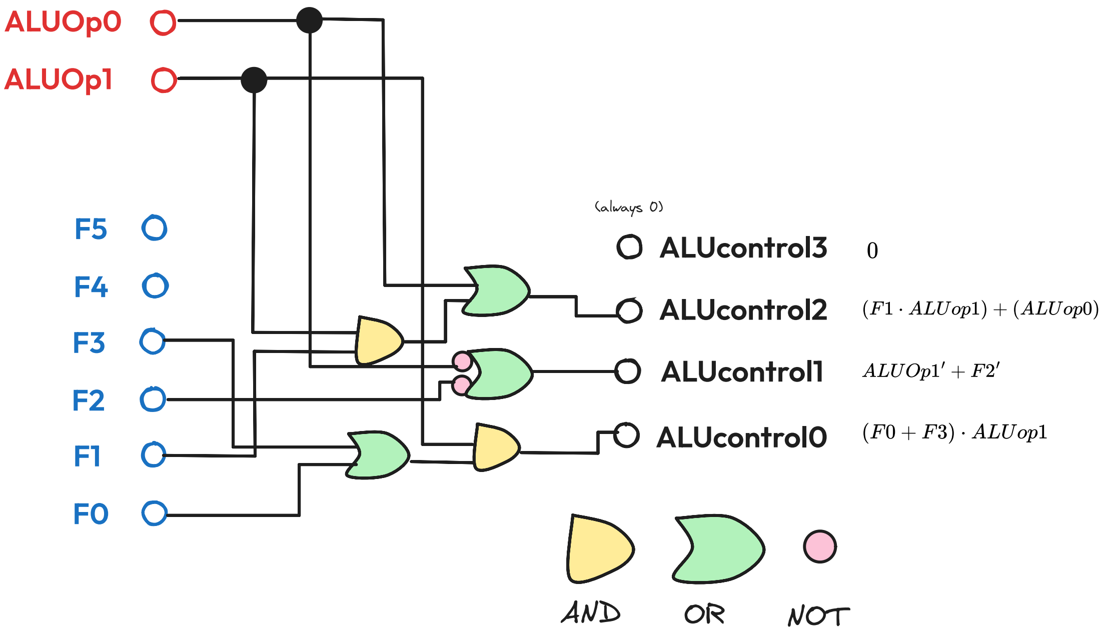
Calculating R Format Encoding

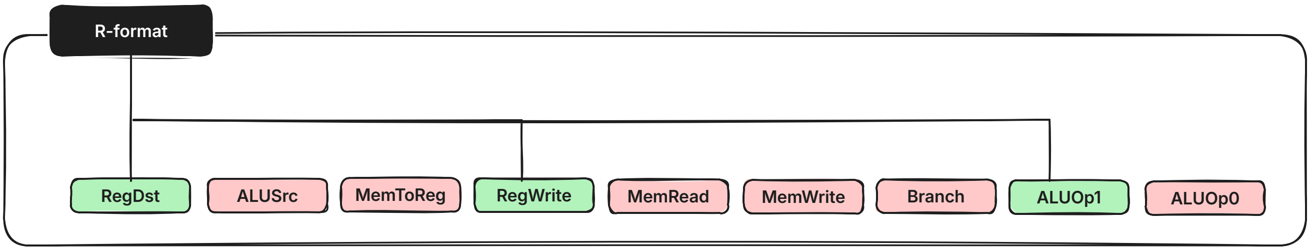
- `SLL/SRL` (`sll $x, $y, z`)
    - get registers `rt $y` , `rd $x`
    - funct = `0` (SLL) or `2` (SRL)
    - $rt \times 2^{16} + rd \times 2^{11} + z \times 2^{6} + funct$
    - convert to `HEX`
- non-shift (`add $x, $y, $z`)
    - get registers `rs $y` , `rt $z` , `rt $x`
    - funct (look up in datasheet)
    - $rs \times 2^{21} + rt \times 2^{16} + rd \times 2^{11} + funct$
    - convert to `HEX`

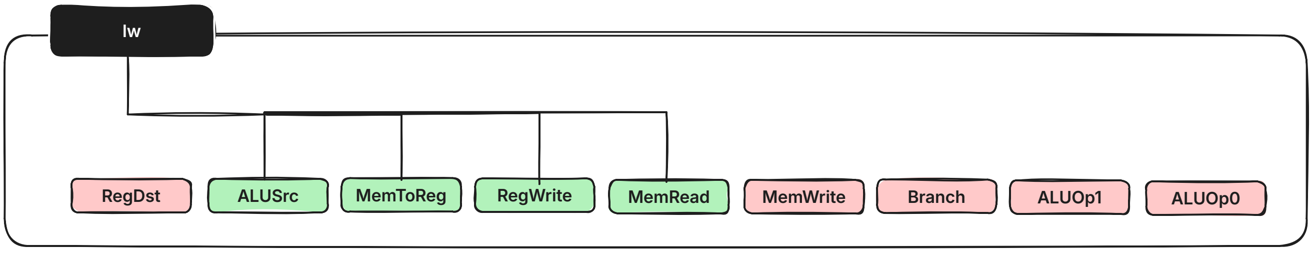
Calculating I-Format Encoding

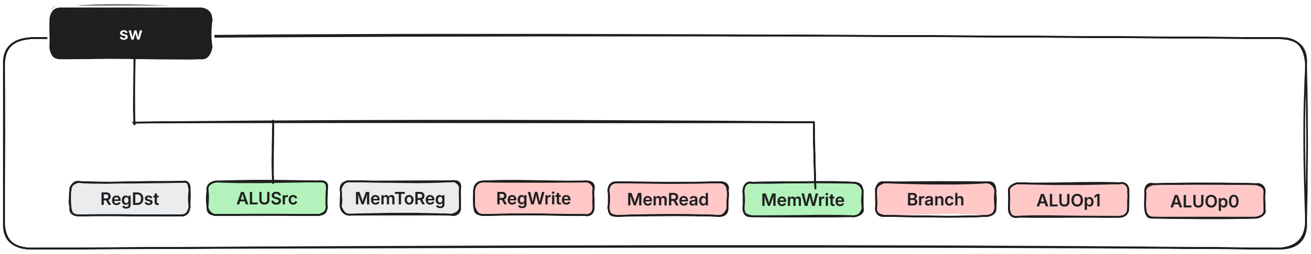
- non-branch (`addi $x, $y, z`)
    - get registers `rs = $y` , `rt = $x`
    - imm = value of z in two's complement
        - enter value in decimal, convert to hexadecimal and remove first 4 digits
    - $opc \times 2^{26} + rs \times 2^{21} + rt \times 2^{16} + imm$
    - convert to `HEX`
- branch (`beq $x, $y, z`)
    - get registers `rs = $x` , `rt = $y`
    - imm = value of z in two's complement
        - enter value in decimal, convert to hexadecimal and remove first 4 digits
    - $opc \times 2^{26} + rs \times 2^{21} + rt \times 2^{16} + imm$
    - convert to `HEX`

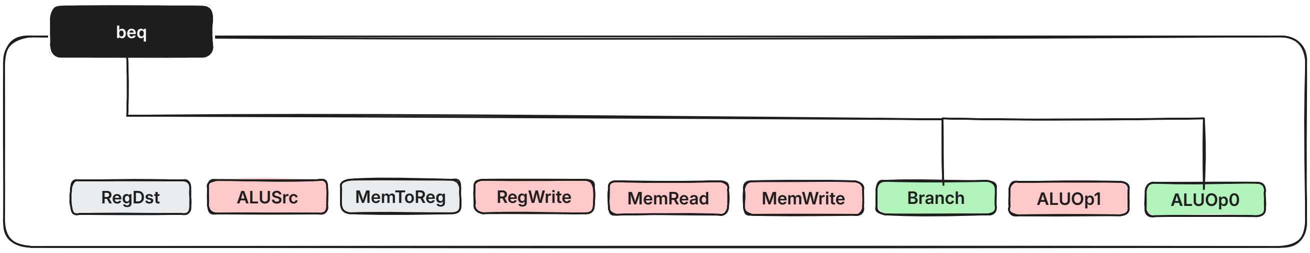
Opcode	ALUop	Instruction Operation	funct	ALU action	ALU control
lw	00	load word	xxxxxx	add	0010
sw	00	store word	xxxxxx	add	0010
beq	01	branch equal	xxxxxx	sub	0110
R-type	10	add	100000	add	0010
R-type	10	sub	100010	sub	0110
R-type	10	and	100100	and	0000
R-type	10	or	100101	or	0001
R-type	10	set on less than	101010	set on less than	0111











Control	Signal	`R-format`	`lw`	`sw`	`beq`
0	`RegDst`	1	0	0	0
1	`ALUSrc`	0	1	1	0
2	`MemToReg`	0	1	0	0
3	`RegWrite`	1	1	0	0
4	`MemRead`	0	1	0	0
5	`MemWrite`	0	0	1	0
6	`Branch`	0	0	0	1
7	`ALUop1`	1	0	0	0
8	`ALUop0`	0	0	0	1

Precedence of Operators

1. Parenthesis
2. `NOT`
3. `AND`
4. `OR`

Laws of Boolean Algebra

Identity laws

$$A + 0 = 0 + A = A$$
$$A \cdot 1 = 1 \cdot A = A$$

Inverse/complement laws

$$A + A' = A' + A = 1$$
$$A \cdot A' = A' \cdot A = 0$$

Commutative laws

$$A + B = B + A$$
$$A \cdot B = B \cdot A$$

Associative laws

$$A + (B + C) = (A + B) + C$$
$$A \cdot (B \cdot C) = (A \cdot B) \cdot C$$

Distributive laws

$$A \cdot (B + C) = (A \cdot B) + (A \cdot C)$$
$$A + (B \cdot C) = (A + B) \cdot (A + C)$$

Duality

If the `AND`/`OR` operations and identity elements `0`/`1` in a Boolean equation are interchanged, the equation remains valid.

$(x + y + z)' = x' \cdot y' \cdot z'$ is valid $\implies$ $(x \cdot y \cdot z)' = x' + y' + z'$ is valid

Idempotency

$$X + X = X$$
$$X \cdot X = X$$

One element/zero element

$$X + 1 = 1 + X = X$$
$$X \cdot 0 = 0 \cdot X = X$$

Involution

$$(X')' = X$$

Absorption

$$X + X \cdot Y = X$$
$$X \cdot (X + Y) = X$$

Absorption

$$X + X' \cdot Y = X + Y$$
$$X \cdot (X' + Y) = X \cdot Y$$

DeMorgan's

$$(X + Y)' = X' \cdot Y'$$
$$(X \cdot Y)' = X' + Y'$$

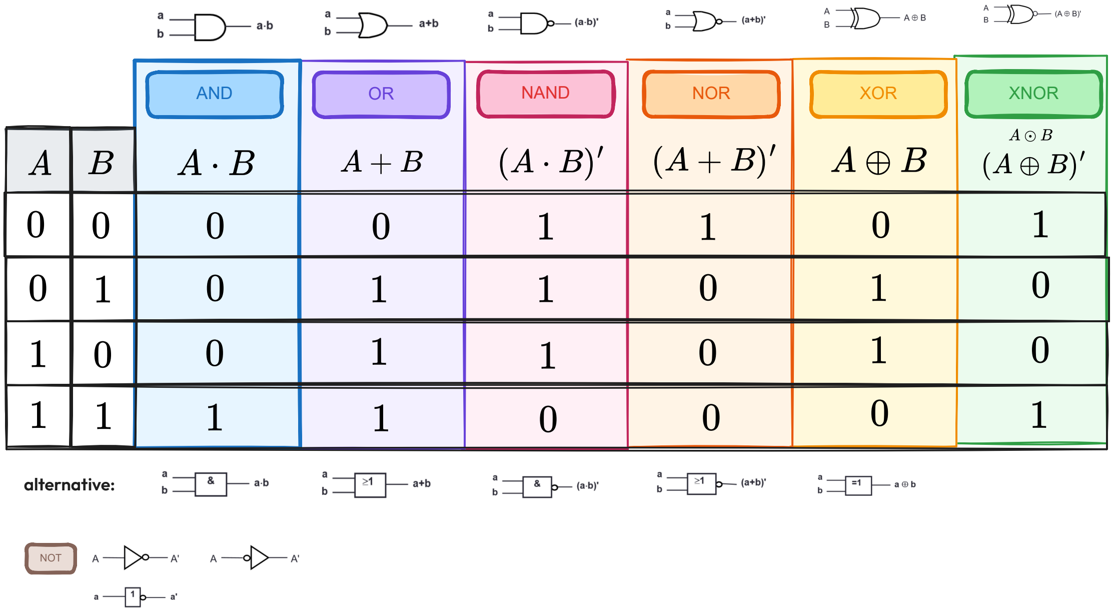
Consensus

$$X \cdot Y + X' \cdot Z + Y \cdot Z = X \cdot Y + X' \cdot Z$$
$$(X \cdot Y) + (X' \cdot Z) + (Y \cdot Z) = (X + Y) + (X' + Z)$$

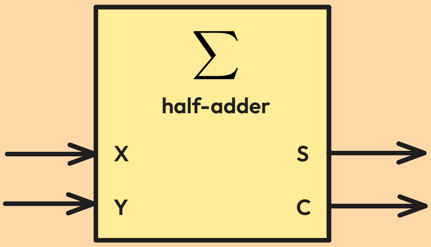
Complement

Obtained by interchanging 1 with 0 in the function's output values

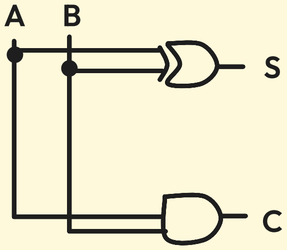
Logic Circuits



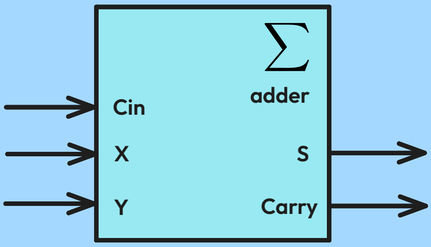
		AND	OR	NAND	NOR	XOR	XNOR
A	B	$A \cdot B$	$A + B$	$(A \cdot B)'$	$(A + B)'$	$A \oplus B$	$A \odot B$ $(A \oplus B)'$
0	0	0	0	1	1	0	1
0	1	0	1	1	0	1	0
1	0	0	1	1	0	1	0
1	1	1	1	0	0	0	1



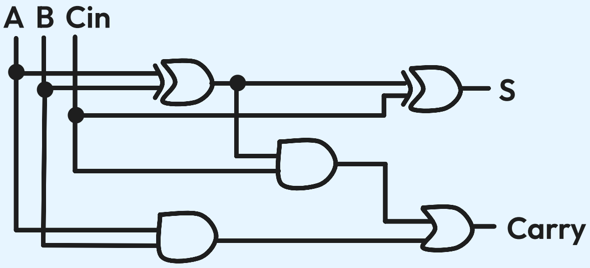
X	Y	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



Adders



X	Y	Cin	S	Carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



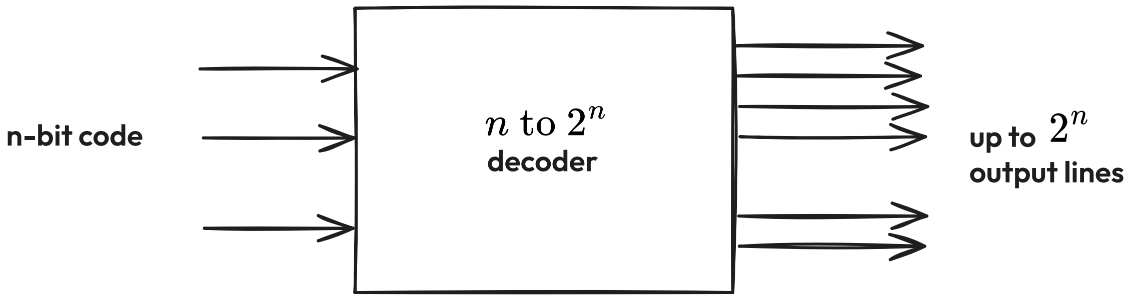
MSI Components

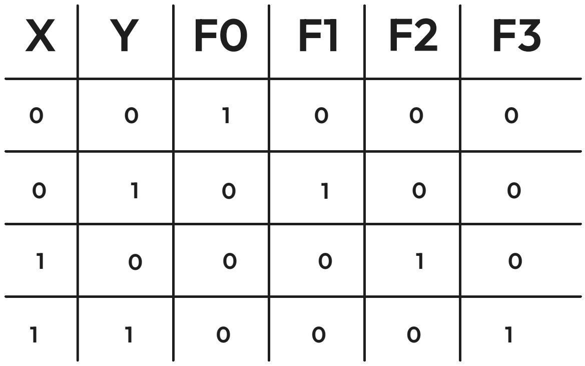
Decoder

Decoder

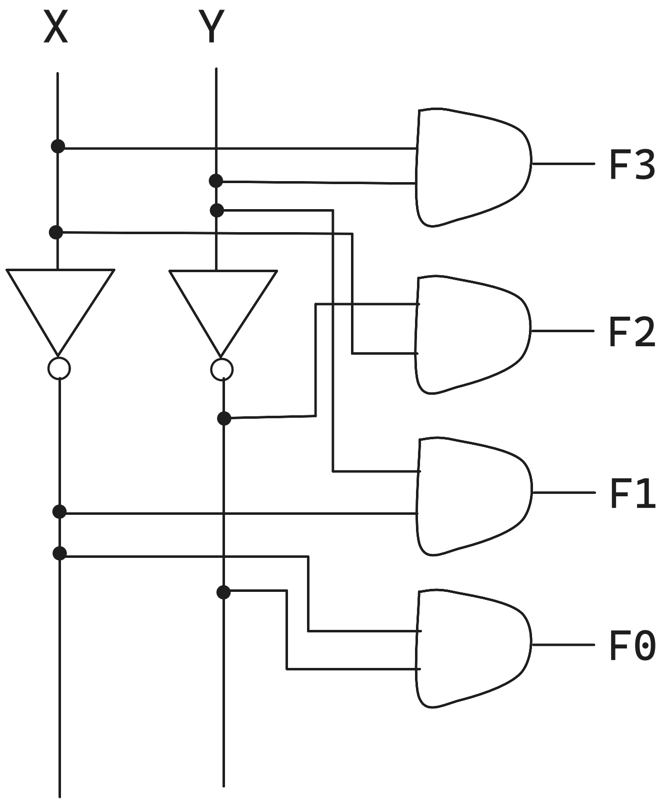
Input: a code (from n input lines)
Output: 2^n output lines.

`n-to-m`, `n:m`, `n x m` $(m \leq 2^n)$





X	Y	F0	F1	F2	F3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

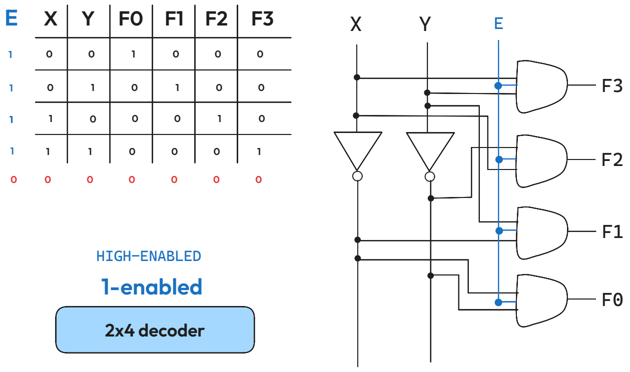


2x4 decoder

Each output of a $n \times m$ decoder is a minterm of a $n-$ variable function.

Enable

Decoders often come with an enable control signal so that device is only activated when $E = 1$.

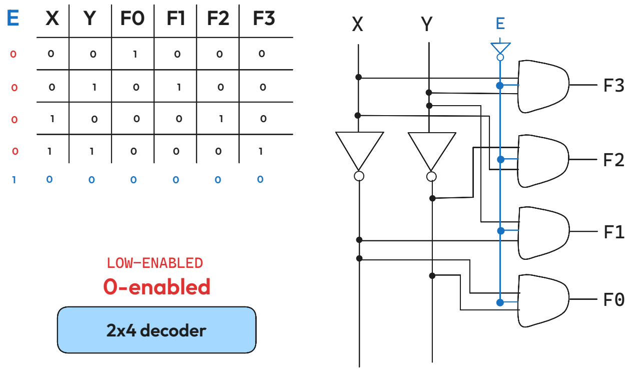


E	X	Y	F0	F1	F2	F3
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1
0	0	0	0	0	0	0

HIGH-ENABLED

1-enabled

2x4 decoder



E	X	Y	F0	F1	F2	F3
0	0	0	1	0	0	0
0	0	1	0	1	0	0
0	1	0	0	0	1	0
0	1	1	0	0	0	1
1	0	0	0	0	0	0

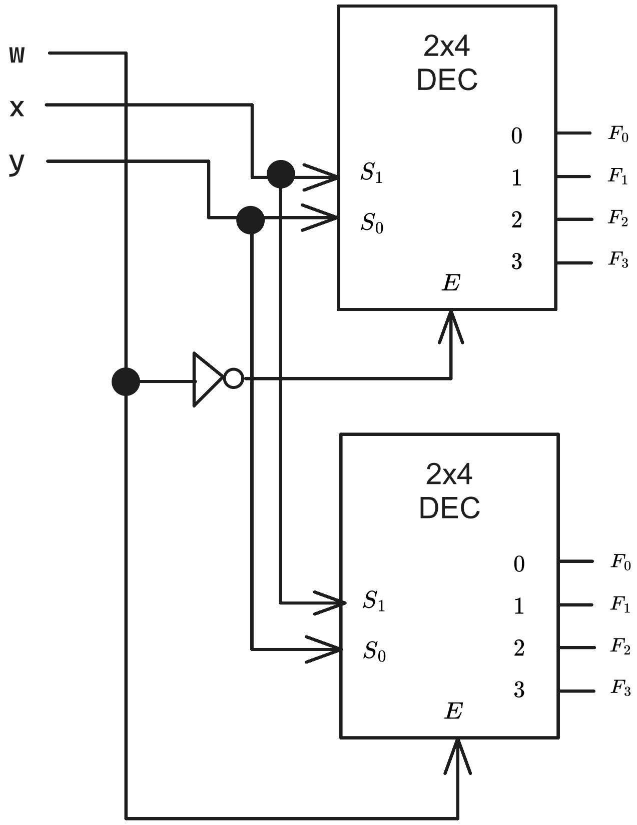
LOW-ENABLED

0-enabled

2x4 decoder

Constructing Larger Decoders

Larger decoders can be constructed from smaller ones.



Encoding

Encoder

Input: 2^n input lines
Output: n bits of code

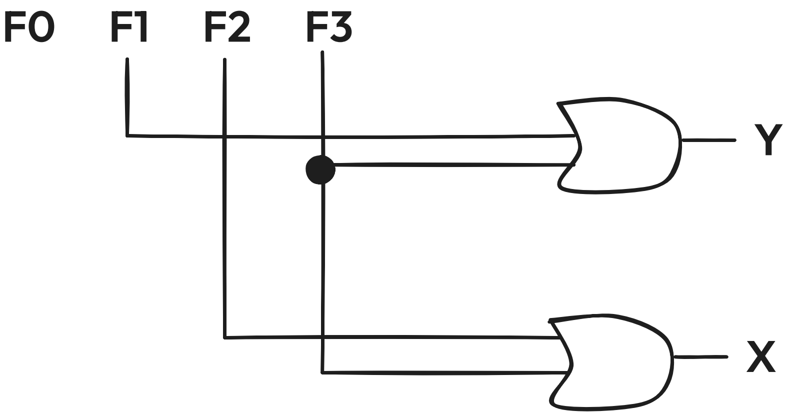
X	Y	F0	F1	F2	F3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

(the rest are don't cares)

With this, we can obtain:

$$X = F2 + F3$$
$$Y = F1 + F3$$

4x2 encoder



At any one time, only one input line of an encoder has a value of `1` (high), the rest are `0` (low).

Priority Encoders

A priority encoder is one with priority

- if two or more inputs, inputs with highest priority takes precedence:

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

Multiplexers

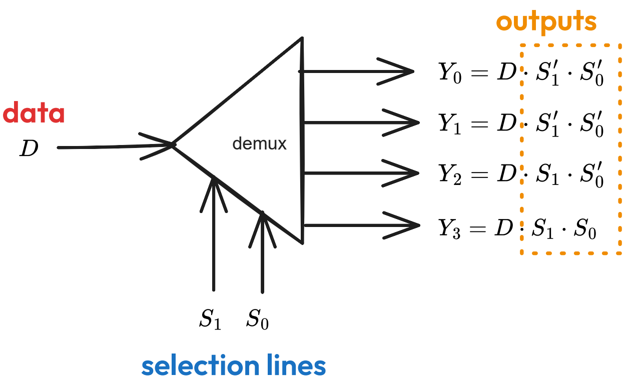
Helps share a single communication line among a number of devices. Only one source and one destination can use the communication line.

Demultiplexer

Demultiplexer

Directs data from input to one selected output line.

Input: Input line, set of selection lines
Output: Data from input to one selected output line


outputs
data
D
demux
$Y_0 = D \cdot S_1' \cdot S_0'$
$Y_1 = D \cdot S_1' \cdot S_0$
$Y_2 = D \cdot S_1 \cdot S_0'$
$Y_3 = D \cdot S_1 \cdot S_0$
S_1
S_0
selection lines

Demultiplexer circuit is identical to a decoder with enable.

Multiplexer

Multiplexer

(Data selector)

Input: A number of input lines, a number of selection lines
Output: Data to one output line - sum of (product of data lines and selection lines)

A 2^n-to-1-line multiplexer - $2^n : 1$ MUX is made from an $n : 2^n$ decoder by adding to it 2^n input lines, one to each `AND` gate.

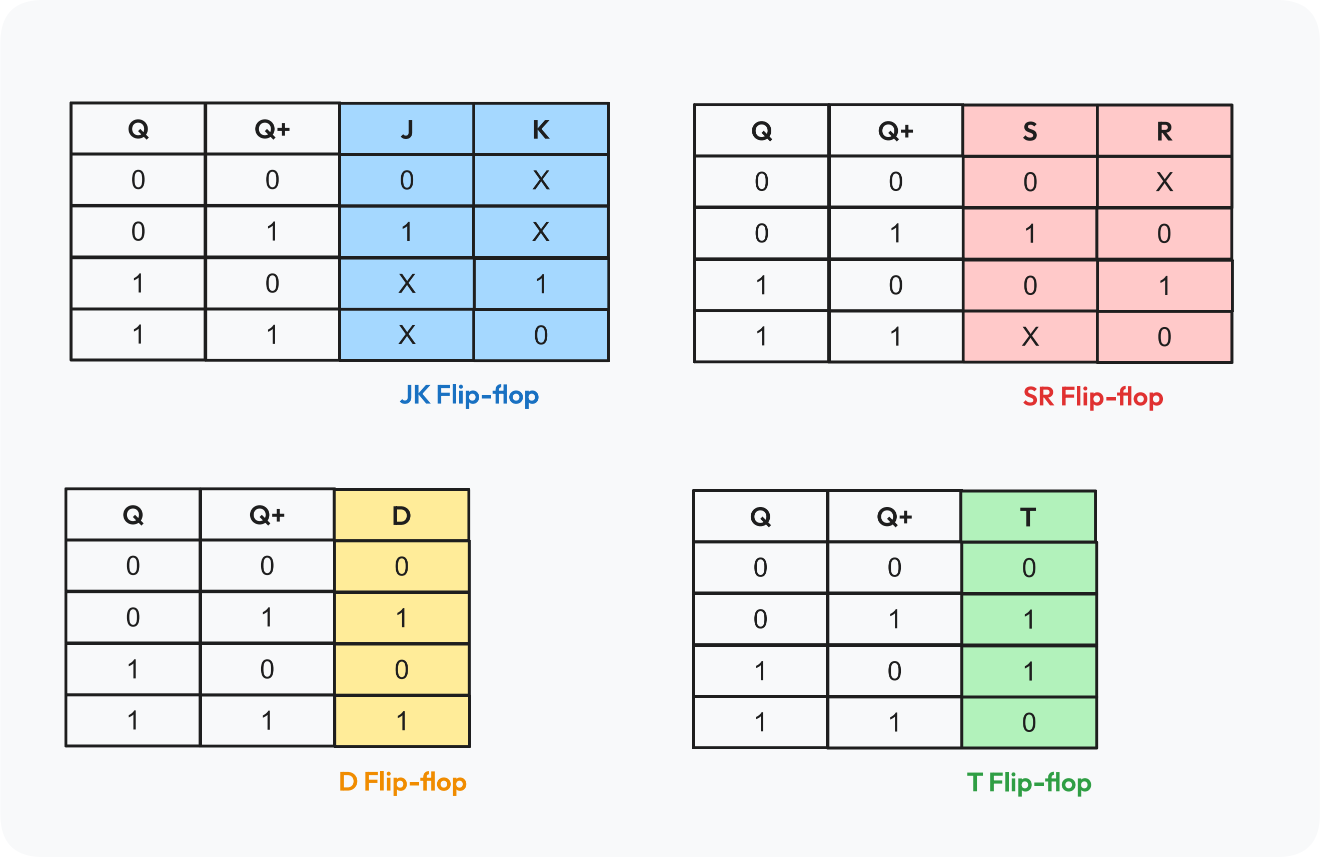
Sequential Logic

S	R	CLK	$Q(t+1)$	
0	0	X	$Q(t)$	No change
0	1	$\uparrow$	0	Reset
1	0	$\uparrow$	1	Set
1	1	$\uparrow$	$?$	Invalid/Unpredictable

D	CLK	$Q(t+1)$	
1	$\uparrow$	1	Set
0	$\uparrow$	0	Reset

J	K	CLK	$Q(t+1)$	
0	0	$\uparrow$	$Q(t)$	No change
0	1	$\uparrow$	0	Reset
1	0	$\uparrow$	1	Set
1	1	$\uparrow$	$Q(t)'$	Toggle

T	CLK	$Q(t+1)$	
0	$\uparrow$	$Q(t)$	No change
1	$\uparrow$	$Q(t)'$	Toggle



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

JK Flip-flop

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

SR Flip-flop

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

D Flip-flop

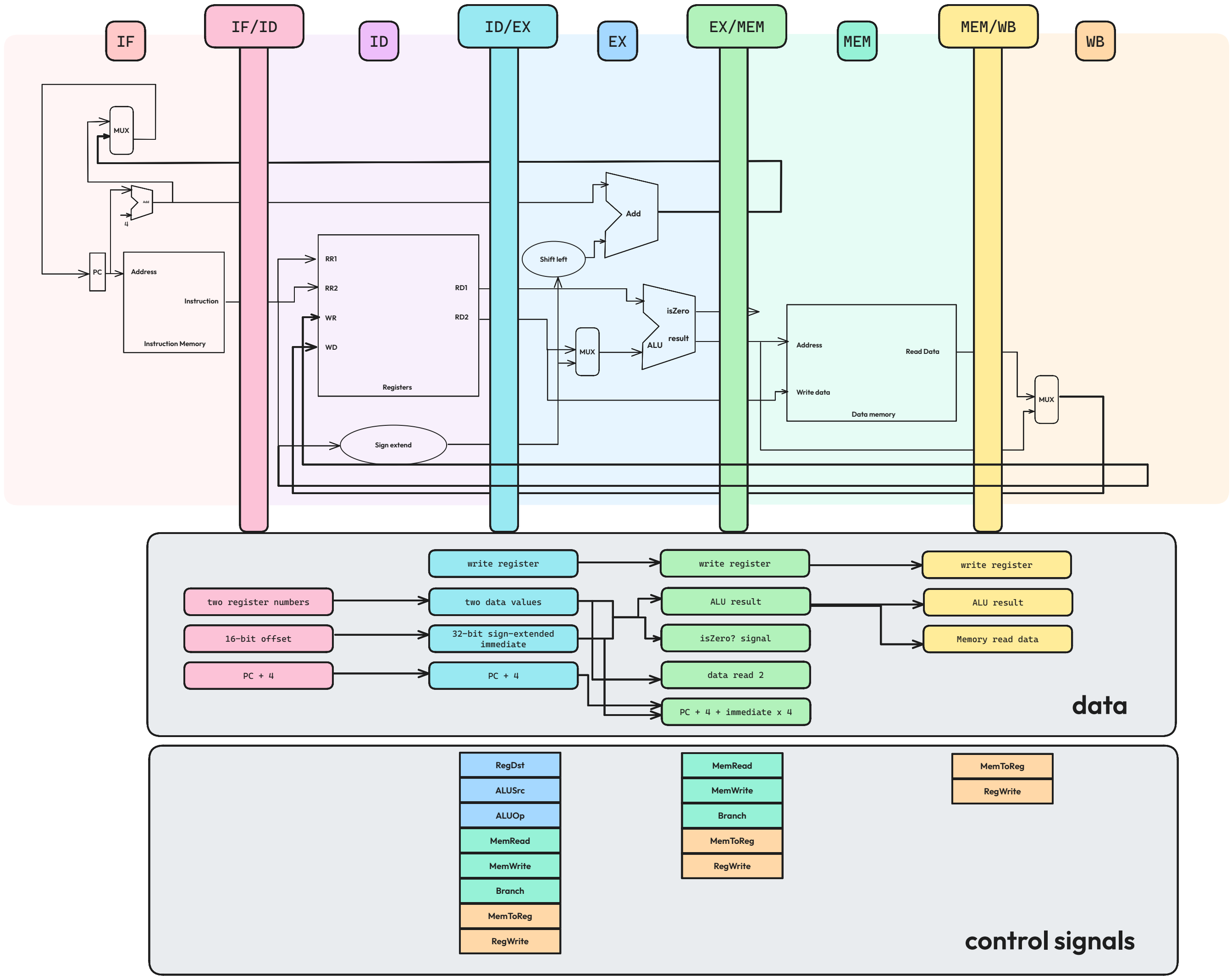
Q	Q+	T
0	0	0
0	1	1
1	0	1
1	1	0

T Flip-flop

Sink state

A state that never moved out of itself to other states

Pipelining



Cycle Delay Table

Jump address calculated at ID (4th stage)

Calculations assume maximally unoptimised instruction order.

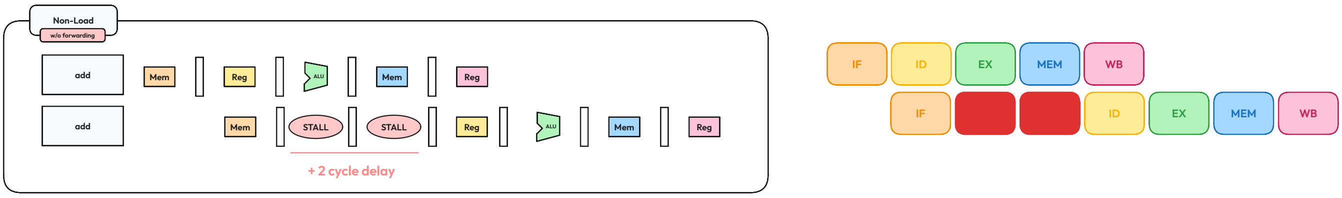
MEM MEM lw

ALU ALU all ALU controls

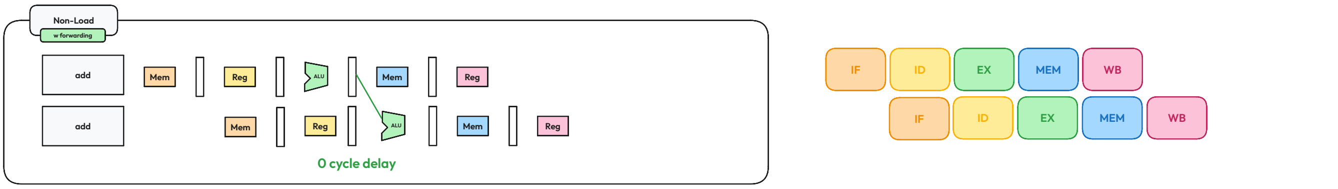
			Data-dependency				Control-related		
			RAW		RAW before branch		Prediction		Jump
			ALU	MEM	ALU	MEM	WRONG	RIGHT	
		No optimisations	+2	+2	+2	+2	+3	+3	1
		Forwarding	0	+1	0	+1	+3	+3	1
		Early branching	+2	+2	+2	+2	+1	+1	1
		Branch prediction	+2	+2	+2	+2	+3	0	1
	Early branching	Forwarding	0	+1	+1	+2	+1	+1	1
	Early branching	Branch prediction	+2	+2	+2	+2	+1	0	1
Forwarding	Early branching	Branch prediction	0	+1	+1	+2	+1	0	1

With forwarding

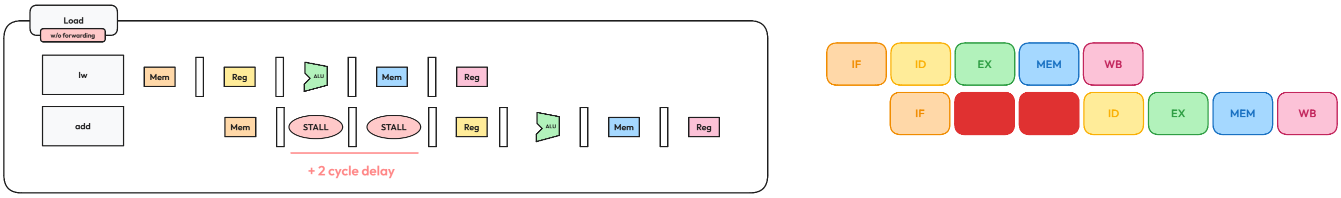
Non-load



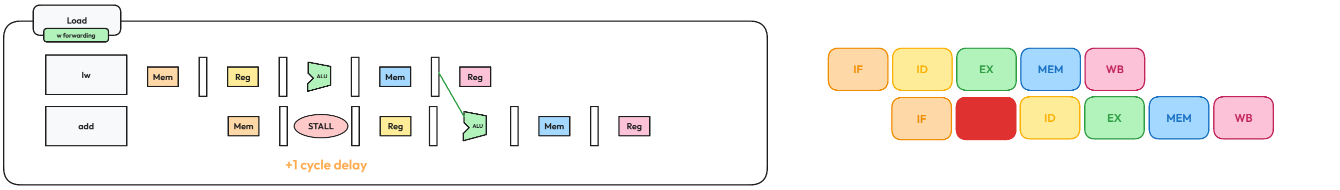
+ 2 cycle delay



0 cycle delay

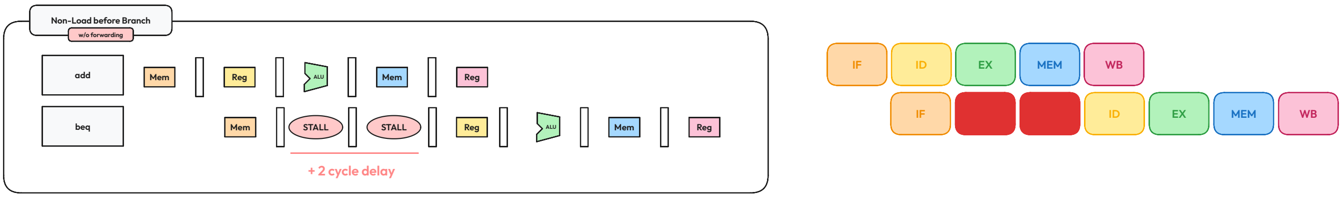


+ 2 cycle delay

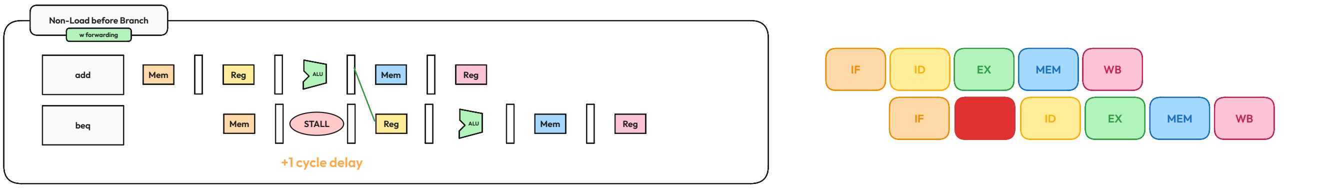


+1 cycle delay

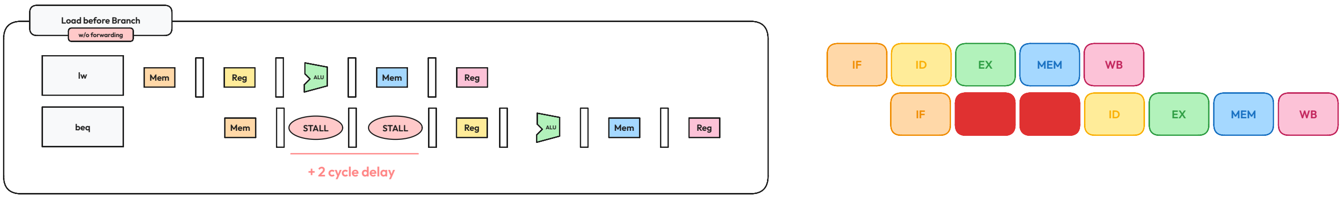
Before branching



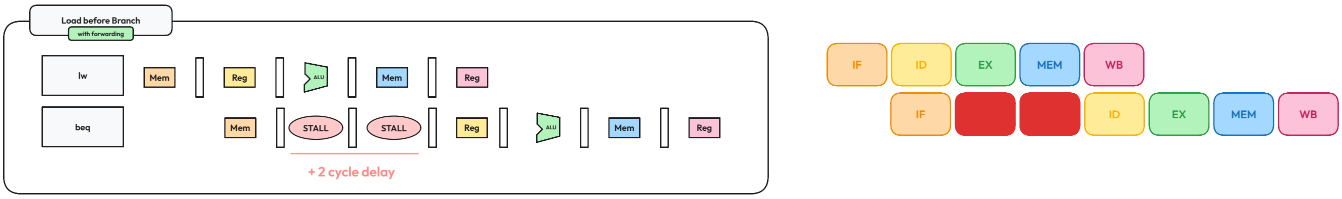
+ 2 cycle delay



+1 cycle delay

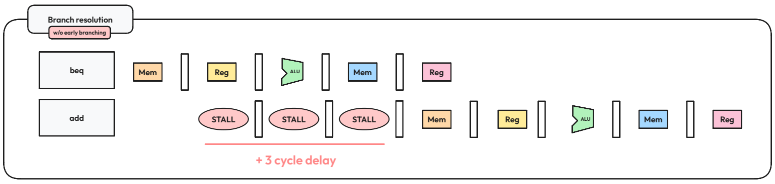


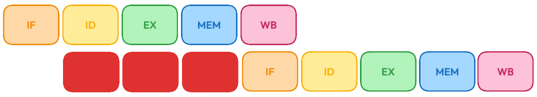
+ 2 cycle delay

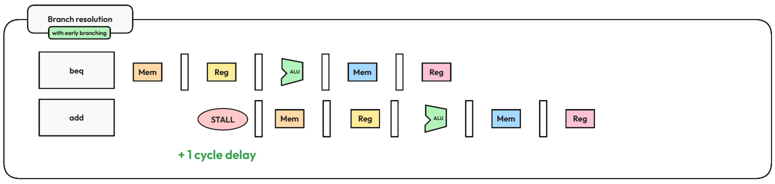


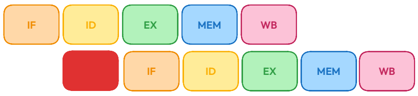
+ 2 cycle delay

Early branching

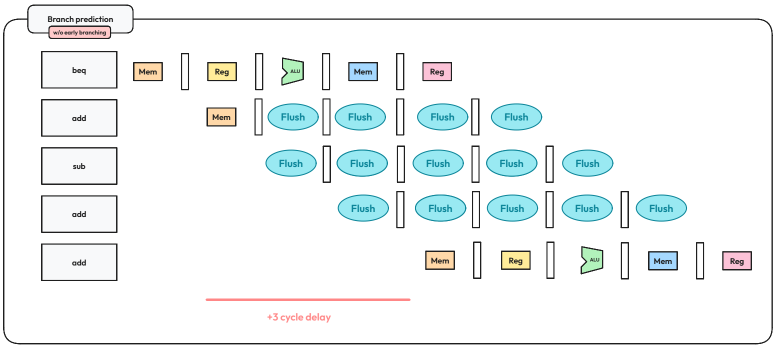


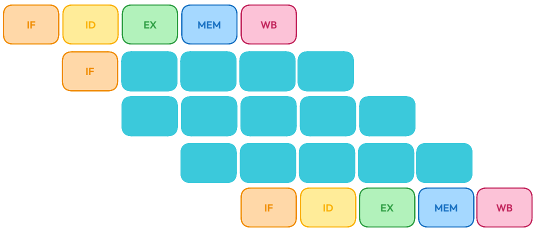


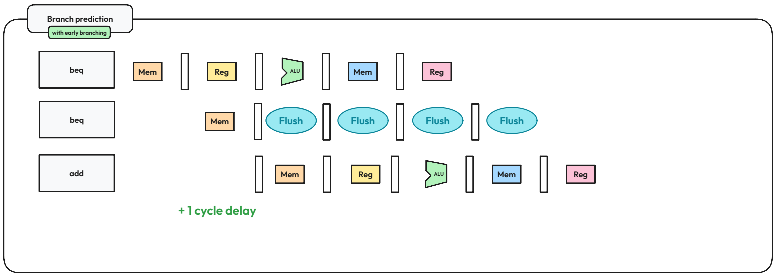


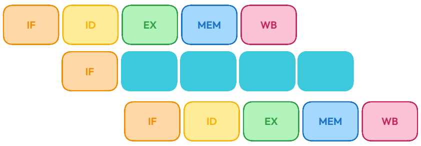


With branch prediction

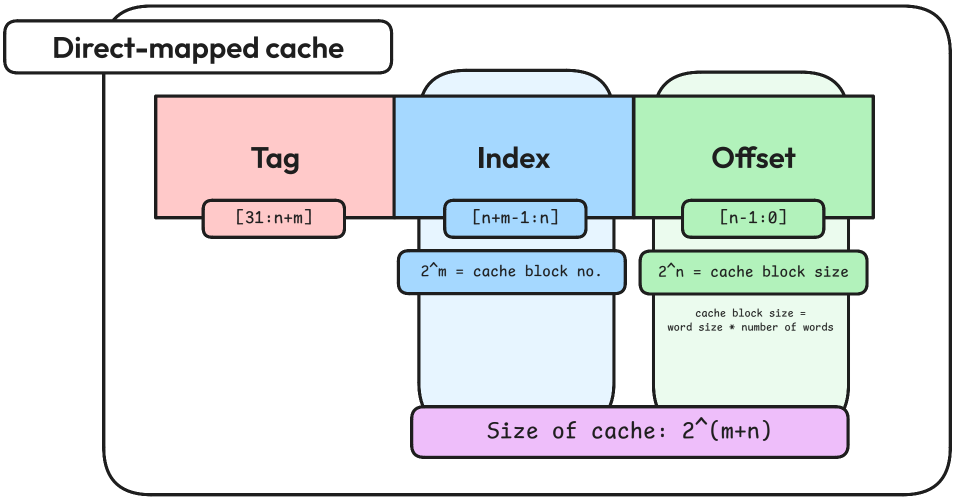








Cache



Direct-mapped cache

Tag

[31:n+m]

Index

[n+m-1:n]

2^m = cache block no.

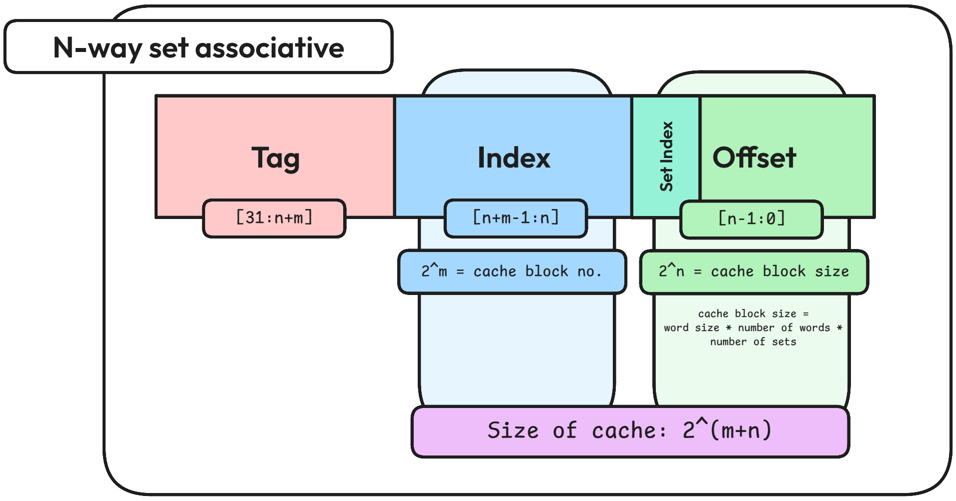
Offset

[n-1:0]

2^n = cache block size

cache block size = word size * number of words

Size of cache: 2^(m+n)



N-way set associative

Tag

[31:n+m]

Index

[n+m-1:n]

2^m = cache block no.

Set Index

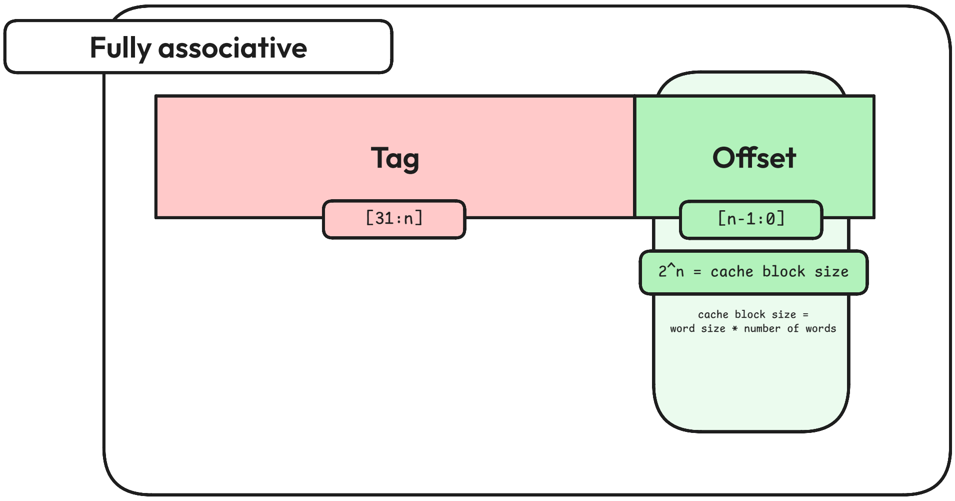
Offset

[n-1:0]

2^n = cache block size

cache block size = word size * number of words * number of sets

Size of cache: 2^(m+n)



Fully associative

Tag

[31:n]

Offset

[n-1:0]

2^n = cache block size

cache block size = word size * number of words

Principle of locality

Program accesses only a small portion of the memory address space within a small time interval

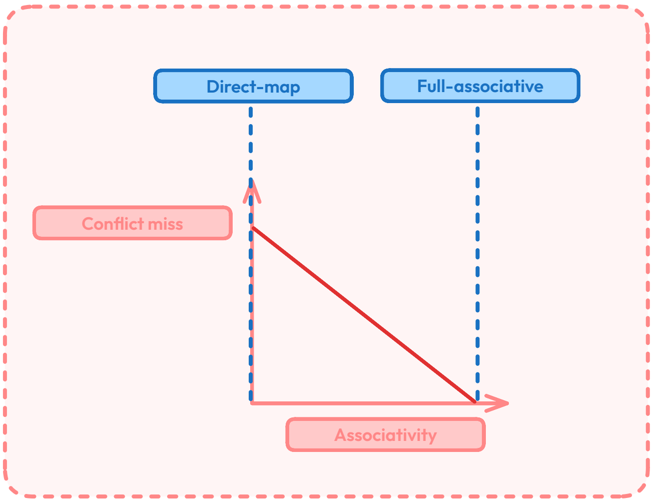
Temporal locality

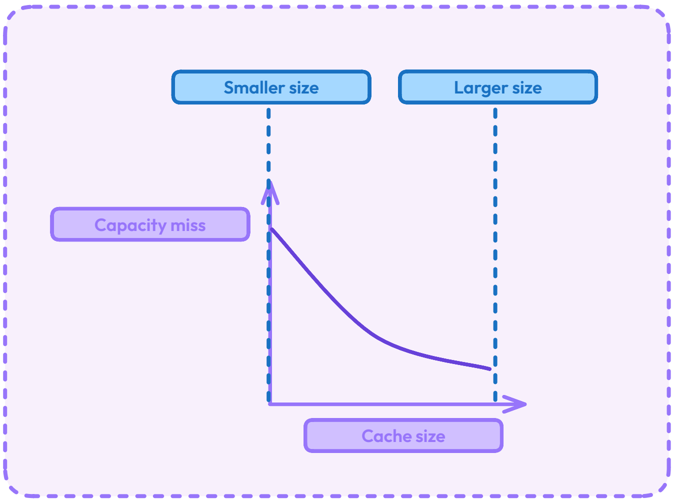
If an item is referenced, it will tend to be referenced again soon

Spatial locality

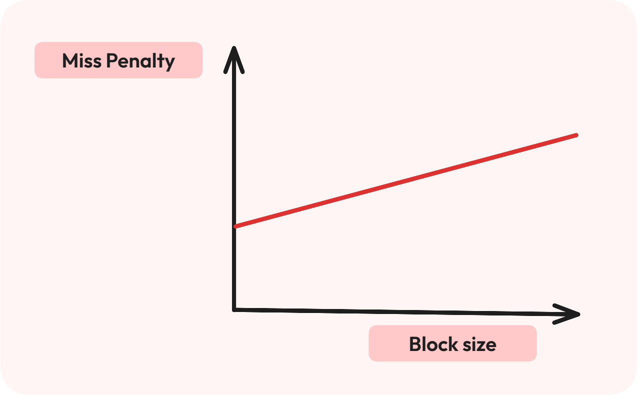
If an item is referenced, nearby items will tend to be referenced soon.

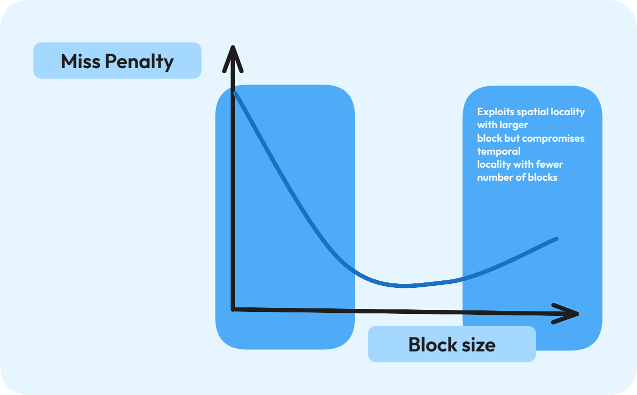
1. Cold/compulsory miss remains the same irrespective of cache size/associativity
2. Conflict miss goes down with increasing associativity on the same cache size
3. Conflict miss is 0 for FA caches
4. For same cache size, capacity miss remains the same irrespective of associativity
5. Capacity miss decreases with increasing cache size



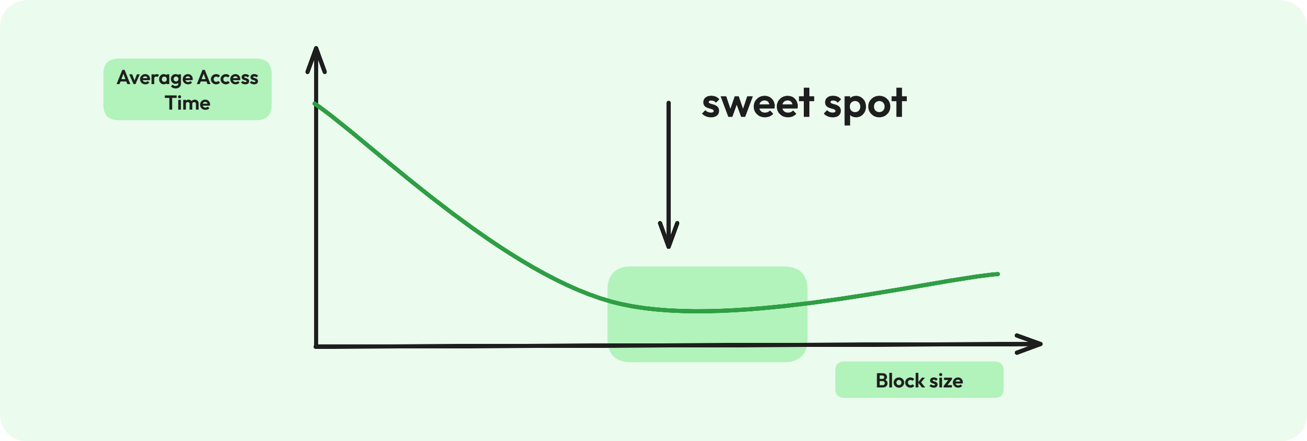


Cache Misses	
Cold/compulsory	same irrespective of cache size/associativity
Conflict miss	for same cache size, decreases with increasing associativity
Capacity miss	decreases with increasing cache size





Exploits spatial locality with larger block but compromises temporal locality with fewer number of blocks



sweet spot

	Direct-Mapped	Set Associative	Fully Associative
Block Placement	Only one block, defined by index	Any one of N blocks defined by index	Any cache block
Block Identification	Tag match with only one block	Tag match for all blocks within set	Tag match for all blocks within cache
Block Replacement	Choose block based on index (no choice)	Choose block based on replacement policy	Choose block based on replacement policy