C-programming:

Placeholders:

%c - char

%d - int

%f - float/double (printf variant)

%lf - double (scanf)

%p - pointers

%s - string

Boolean values:

>=1 - true

0 - false

Byte and bits conversion:

1 byte = 8 bits

char = 1 byte

int/float = 4 bytes

double = 8 bytes

Address and Pointers:

Let a_ptr be a pointer to b.

-> eg: char *a_ptr = &b

Then *a_ptr = b

- a_ptr++ will increase the address being referenced by number of bytes the data type requires. eg int: +4 to address

Strings:

Strings are terminated by a null terminator. (\0)

Strings are stored as an array of char in C.

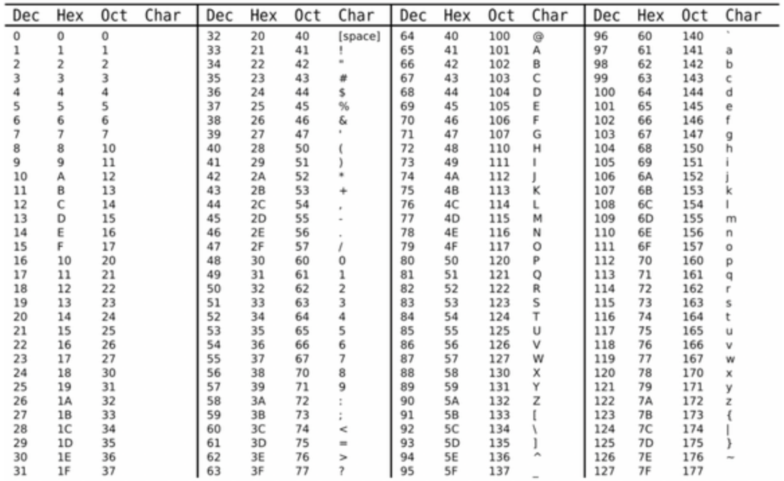
Initialising Strings:

```
char fruit_name[] = "apple"
char fruit_name2[] = {'a','p','p',
'l', 'e', '\0'};
```

Parity:

- Odd parity: bit 7 is set to 1 if there exist an even number of 1's in bits 0-6.
- Even parity: bit 7 is set to 1 if there exist an odd number of 1's in bits 0-6

ASCII table:



Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	0		32	20	40	[space]	64	40	100	@	96	60	140	`
1	1	1		33	21	41	!	65	41	101	A	97	61	141	a
2	2	2		34	22	42	"	66	42	102	B	98	62	142	b
3	3	3		35	23	43	#	67	43	103	C	99	63	143	c
4	4	4		36	24	44	$	68	44	104	D	100	64	144	d
5	5	5		37	25	45	%	69	45	105	E	101	65	145	e
6	6	6		38	26	46	&	70	46	106	F	102	66	146	f
7	7	7		39	27	47	'	71	47	107	G	103	67	147	g
8	8	10		40	28	50	(	72	48	110	H	104	68	150	h
9	9	11		41	29	51	)	73	49	111	I	105	69	151	i
10	A	12		42	2A	52	*	74	4A	112	J	106	6A	152	j
11	B	13		43	2B	53	+	75	4B	113	K	107	6B	153	k
12	C	14		44	2C	54	,	76	4C	114	L	108	6C	154	l
13	D	15		45	2D	55	-	77	4D	115	M	109	6D	155	m
14	E	16		46	2E	56	.	78	4E	116	N	110	6E	156	n
15	F	17		47	2F	57	/	79	4F	117	O	111	6F	157	o
16	10	20		48	30	60	0	80	50	120	P	112	70	160	p
17	11	21		49	31	61	1	81	51	121	Q	113	71	161	q
18	12	22		50	32	62	2	82	52	122	R	114	72	162	r
19	13	23		51	33	63	3	83	53	123	S	115	73	163	s
20	14	24		52	34	64	4	84	54	124	T	116	74	164	t
21	15	25		53	35	65	5	85	55	125	U	117	75	165	u
22	16	26		54	36	66	6	86	56	126	V	118	76	166	v
23	17	27		55	37	67	7	87	57	127	W	119	77	167	w
24	18	30		56	38	70	8	88	58	130	X	120	78	170	x
25	19	31		57	39	71	9	89	59	131	Y	121	79	171	y
26	1A	32		58	3A	72	:	90	5A	132	Z	122	7A	172	z
27	1B	33		59	3B	73	;	91	5B	133	[	123	7B	173	{
28	1C	34		60	3C	74	<	92	5C	134	\	124	7C	174	|
29	1D	35		61	3D	75	=	93	5D	135	]	125	7D	175	}
30	1E	36		62	3E	76	>	94	5E	136	^	126	7E	176	~
31	1F	37		63	3F	77	?	95	5F	137	_	127	7F	177	

Number systems

Negative numbers:

Represented by 2s complement.

Sign bit = MSB

Methodology:

- 1's complement: Invert all bits
- 2's complement:

Shortcut if no decimal point -> Copy all bits from right until hit first 1, then invert everything else.

Otherwise -> Invert everything then +1

Addition of integers:

- Perform binary addition
- Ignore carry out of MSB for 2s complement

OR

- Add 1 to the result if 1s complement addition is performed
- Check overflow. Overflow occur when positive + positive give negative or negative + negative give positive.
- Overflow -> Out of range

Excess Representation(IEEE-754 for floating point representation):

- Sign + Exponent(excess-127) + Mantissa
- 1 bit + 8 bits + 23 bits
- Steps:
  - Convert decimal to binary
  - Normalize such that it becomes in the form 1.xxx
  - Exponent = 127 + number of shifts then represent in 8 bits.
  - Mantissa = copy over what is the xxx and fill to 23 bits.

Decimal-Hexadecimal values:

- 0 - 0
- 1 - 1
- 3 - 3
- 5 - 5
- 7 - 7
- 9 - 9
- 11 - B
- 13 - D
- 15 - F

1 - 1

2 - 2

4 - 4

6 - 6

8 - 8

10 - A

12 - C

14 - E

MIPS programming:

Registers:

- 32 registers in total
- Refer to reference sheet to get register values
- value in $zero or $0 never changes; always 0

Instructions:

- Maximum number of shifts: 32 (2^5)
- AND operation - used to mask unwanted bits
- OR operation - used to force certain bits to 1
- NOR operation - usually used to convert to a NOT operation

(by setting **first operand** to be 0)
eg: nor $t1, $0, $s0 to get -[$s0]

- XOR operation - usually used to convert to a NOT operation

(by setting **second operand** to be all 1s)

- Immediate value can only take up to 16 bits of value
- Load large value immediate: use lui then ori

Memory:

Address is k bits, memory can store up till 2^k locations

- Mnemonic for lw, sw
  - Load to, Store from

Inequalities in MIPS:

Mapping: x -> s0, y -> s1

- x > y condition:
  - slt t1, s1, s0 **(inverted)**
  - beq t1, $zero, label
- x < y condition:
  - slt t1, s0, s1
  - beq t1, $zero, label
- x <= y condition:
  - slt t1, s1, s0 **(inverted)**
  - bne t1, $zero, label
- x >= y condition:
  - slt t1, s0, s1
  - bne t1, $zero, label

Instruction formats:

- R-format:
  - opcode,rs,rt,rd,shamt,funct
    - **opcode is always 0**
- I-format:
  - opcode,rs,rt,immd
    - **immd is always 16 bits max**
- J-format:
  - opcode, **immd(26 bits)**
    - Can only specify up to 2^26 instructions. But we say that it can specify up to 2^28 bytes
    - **Max range: 256MB (2^28 bytes)**
    - First 4 bits from PC+4, last 2 bits ignored since word-aligned.

Instruction Set Architecture:

- Maximum number of instructions given fixed length instruction:
  - Maximise the type that has more bits to play around for opcode

Given n bit instruction and m bit instruction where n < m,

- Set type with lower bits to only have 1 instruction, ie n-bit = 1
- Maximum = (2^n - 1)(2^m-n) + 1

- Minimise number of instructions given fixed length instruction:
  - Fix a prefix for the type that has more bits.

Given number of bits in type A be n and type B be m where n < m.

- Then instructions for type B = 1 * 2^(m-n) and type A = 2^n - 1
- Total = 2^(m-n) + (2^n - 1)

Datapath:

Flow:

- Fetch -> Decode -> ALU -> Memory Access -> Register Write
- **Array access: lw**
- **Array updating: sw**
- **lw has to have RegWrite = 1 && MemRead = 1**
- **sw has to have MemWrite = 1**

ALU:

- isZero values: 0(false) / 1(true)

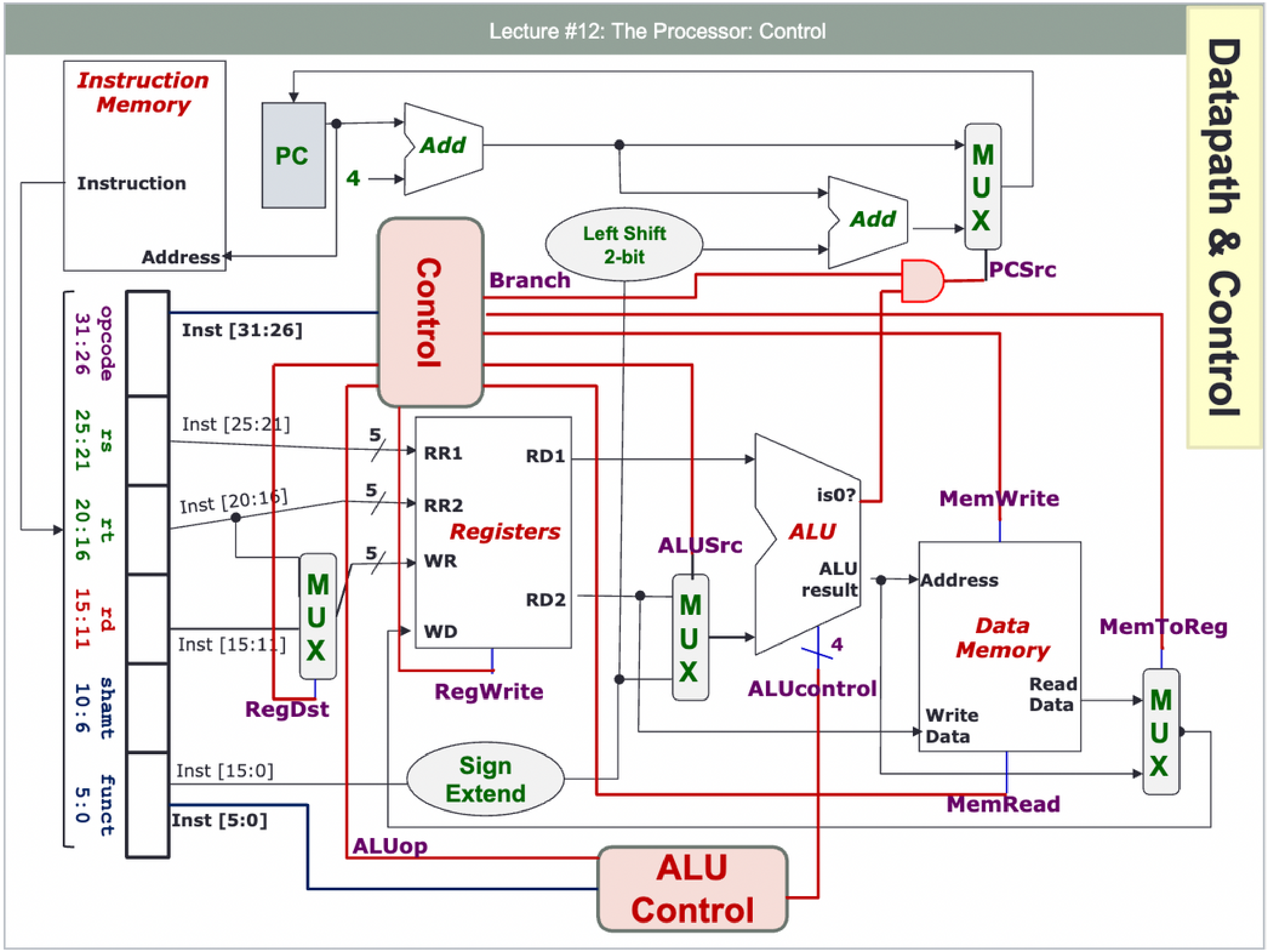
Datapath:

Flow:

- Fetch -> Decode -> ALU -> Memory Access -> Register Write
- **Array access: lw**
- **Array updating: sw**
- **lw has to have RegWrite = 1 && MemRead = 1**
- **sw has to have MemWrite = 1**

ALU:

- isZero values: 0(false) / 1(true)



Lecture #12: The Processor: Control

Datapath & Control

Controlpath:

Control signals:

- MemToReg is reversed(1 on top for Read Data input, 0 for ALU output as input)

Instruction execution:

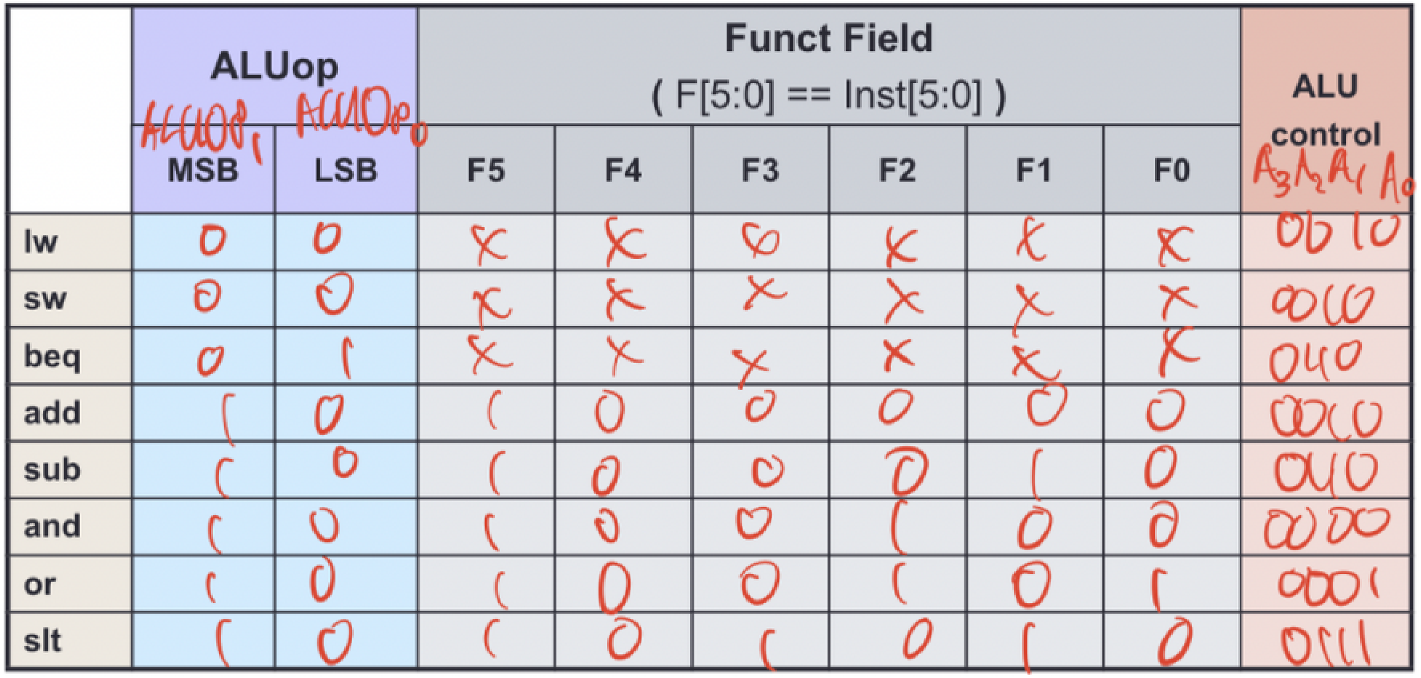
Single cycle implementation:

- Read contents of 1 or more storage elements(register/memory)
- Perform computation
- Write results to 1 or more storage elements
- **WITHIN A CLOCK PERIOD**
- **time taken depends on slowest instruction**
- **Disadvantage:**

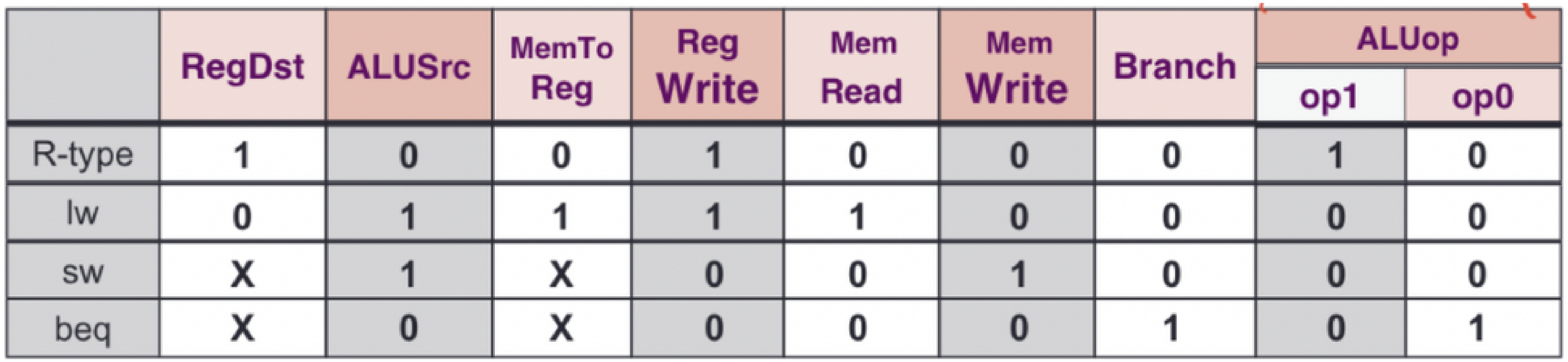
clock cycle must be long enough to accommodate the slowest instruction -> all instructions will take the same time as the slowest instruction

Multicycle implementation:

- Time taken depends on **number of steps now**
- **Disadvantage: might not necessarily be faster -> depends on mix of instructions**



	ALUop		Funct Field (F[5:0] == Inst[5:0])						ALU control
	$ALUop_1$ MSB	$ALUop_0$ LSB	F5	F4	F3	F2	F1	F0	$A_3A_2A_1A_0$
lw	0	0	X	X	X	X	X	X	0010
sw	0	0	X	X	X	X	X	X	0010
beq	0	1	X	X	X	X	X	X	0110
add	1	0	1	0	0	0	0	0	0010
sub	1	0	1	0	0	0	1	0	0110
and	1	0	1	0	0	1	0	0	0000
or	1	0	1	0	0	1	0	1	0001
slt	1	0	1	0	1	0	1	0	0111



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

Boolean Algebra:

- **NAO priority(¬,•,+)**

Duality property:

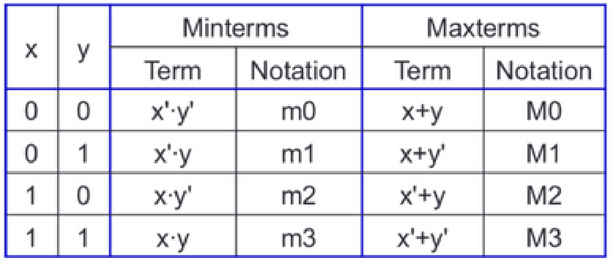
- If AND/OR operators and identity elements 0/1 in a Boolean equation are interchanged, it remains valid. (eg Swapping operands, like AND to OR and OR to AND)
- Duality only concerns with boolean operators and boolean identity elements(no negation involved)

Impt pattern recognitions:

- X'Y + XY' = X⊕Y (XOR)
- X'Y' + XY = X⊙Y (XNOR)

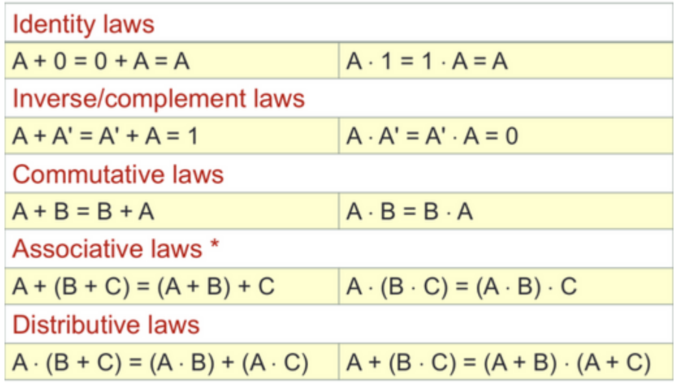
Minterm vs Maxterms:

- Minterm: Non-complementary = 1, Complementary term = 0
- Maxterm: Non-complementary = 0, Complementary term = 0

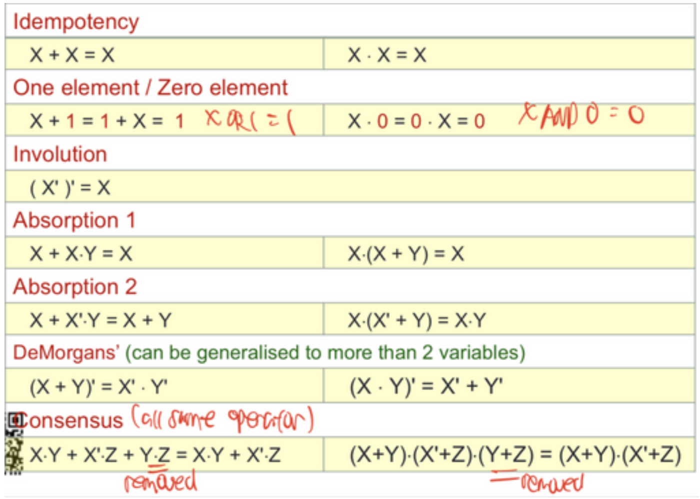


x	y	Minterms		Maxterms	
		Term	Notation	Term	Notation
0	0	x'·y'	m0	x+y	M0
0	1	x'·y	m1	x+y'	M1
1	0	x·y'	m2	x'+y	M2
1	1	x·y	m3	x'+y'	M3

Laws and Theorems:



Identity laws	
A + 0 = 0 + A = A	A · 1 = 1 · A = A
Inverse/complement laws	
A + A' = A' + A = 1	A · A' = A' · A = 0
Commutative laws	
A + B = B + A	A · B = B · A
Associative laws *	
A + (B + C) = (A + B) + C	A · (B · C) = (A · B) · C
Distributive laws	
A · (B + C) = (A · B) + (A · C)	A + (B · C) = (A + B) · (A + C)



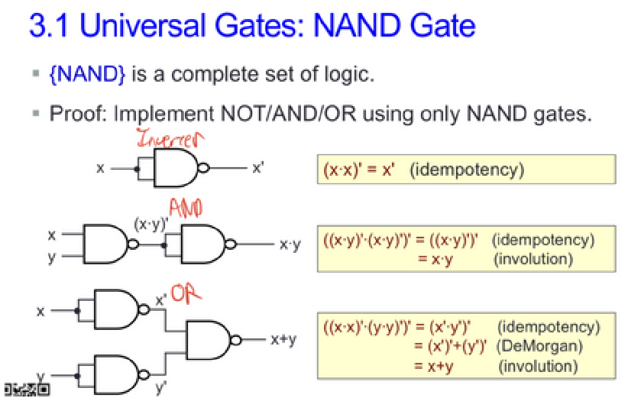
Idempotency	
X + X = X	X · X = X
One element / Zero element	
X + 1 = 1 + X = 1 X OR 1 = 1	X · 0 = 0 · X = 0 X AND 0 = 0
Involution	
(X')' = X	
Absorption 1	
X + X·Y = X	X·(X + Y) = X
Absorption 2	
X + X'·Y = X + Y	X·(X' + Y) = X·Y
DeMorgans' (can be generalised to more than 2 variables)	
(X + Y)' = X' · Y'	(X · Y)' = X' + Y'
Consensus (all same operator)	
X·Y + X'·Z + Y·Z = X·Y + X'·Z removed	(X+Y)·(X'+Z)·(Y+Z) = (X+Y)·(X'+Z) removed

Complement Functions:

- Given boolean function F, the complement of F, F', is obtained by interchanging 1 with 0 in the function's output values.

Logic Circuits & Simplification:

Universal Gates:



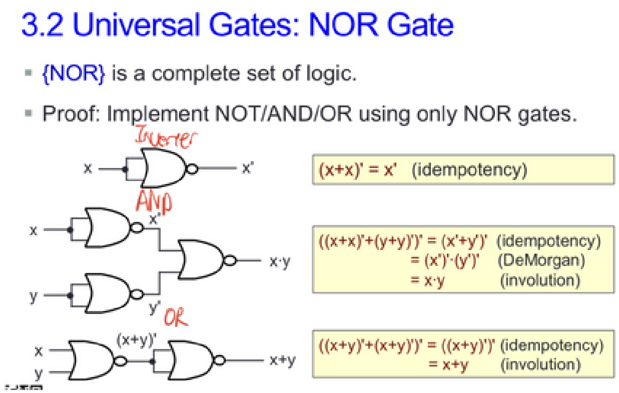
3.1 Universal Gates: NAND Gate

- {NAND} is a complete set of logic.
- Proof: Implement NOT/AND/OR using only NAND gates.

$(x\cdot x)' = x'$ (idempotency)

$((x\cdot y)'\cdot(x\cdot y)')' = ((x\cdot y)')'$ (idempotency)
$= x\cdot y$ (involution)

$((x\cdot x)'\cdot(y\cdot y)')' = (x'\cdot y')'$ (idempotency)
$= (x')'+(y')'$ (DeMorgan)
$= x+y$ (involution)



3.2 Universal Gates: NOR Gate

- {NOR} is a complete set of logic.
- Proof: Implement NOT/AND/OR using only NOR gates.

$(x+x)' = x'$ (idempotency)

$((x+x)'+(y+y)')' = (x'+y')'$ (idempotency)
$= (x')'\cdot(y')'$ (DeMorgan)
$= x\cdot y$ (involution)

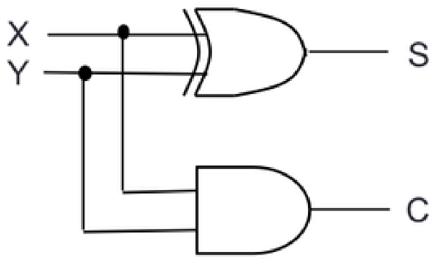
$((x+y)'+(x+y)')' = ((x+y)')'$ (idempotency)
$= x+y$ (involution)

SOP and POS expression:

- An SOP expression can be easily generated by using
  - 2-level AND-OR circuit
  - 2-level NAND circuit
- An POS expression can be easily generated by using
  - 2-level OR-AND circuit
  - 2-level NOR circuit

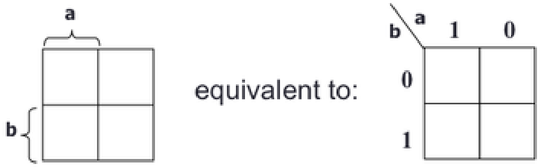
Half Adder:

- Circuit that adds 2 single bits (X,Y) to produce a result of 2 bits (Carry, Sum)
- Canonical form:
  - $C = X \cdot Y$
  - $S = X \oplus Y$



K-Maps:

- Group must be in powers of 2
- Grouping 2^n variables remove n variables from SOP



(Essential) Prime Implicants(EPIs)

- Implicant -> product term that can be used to cover minterms of a function
- Prime Implicant -> product term obtained by combining the maximum possible number of minterms from adjacent squares in K-map.
- Essential Prime Implicant(EPI)

-> Prime implicant that includes at least one minterm that is not covered by any other prime implicant.

K-Maps to find POS expression instead

- Shortcut: group the maxterms(0s) of given function.
- Actual method:

1. Convert K-map of F to K-map of F'(Flip 0s and 1s)
2. get SOP of F'
3. Invert SOP of F' to get POS of F.

Don't care conditions:

- Denoted by d, e.g: F(A,B,C) = Σm(3,5,6)+Σd(0,7)

Combinational Circuits:

General Algorithm:

- Determine problem
- Determine inputs and outputs of circuit
- Draw truth table
- Obtain simplified Bool Algebra expressions
  - Through K-map
- Draw logic diagram based on simplfied Bool Algebra expressions

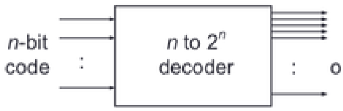
Circuit Delays:

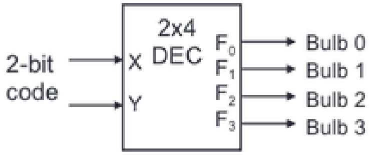
- Given a logic gate with delay *t*, if the inputs are stable at times *t1*, *t2*, ... , *tn*, then the earliest time in which the output will be stable is max(*t1*, *t2*, ... , *tn*) + *t*
- Delay for a combinational circuit: Repeat above for all logic gates.
- **Common circuit delays:**
  - n-bit parallel adder:
    - $Sn = ((n-1) * 2 + 2)t$
    - $Cn+1 = ((n - 1) * 2 + 3)t$
    - max delay = $((n-1) * 2 + 3)t$

MSI Circuits:

Decoders:

- Convert binary information from *n* input lines, to up to 2^n input lines.
  - Select only one output line





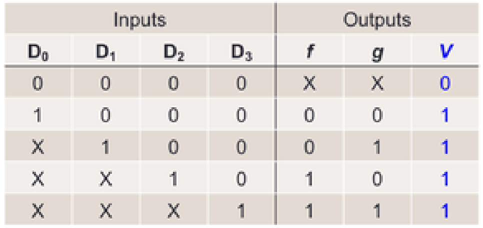
Encoders:

- Reverse decoder: Given a set of input lines, of which exactly one is high and rest are low, provide a code that corresponds to the high input line
- 2^n input lines with n output lines
- implemented w OR gates

Priority Encoder:

- If multiple inputs are equal to 1, **highest priority** takes precedence.
- all inputs 0: invalid input

- Example of a 4-to-2 priority encoder:

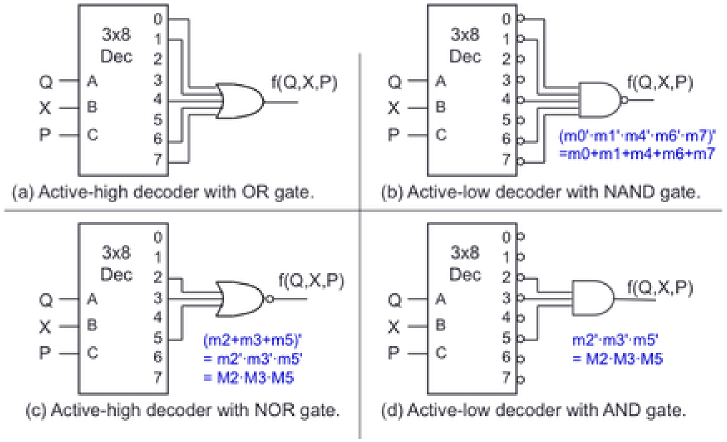


Inputs				Outputs		
D_0	D_1	D_2	D_3	*f*	*g*	*V*
0	0	0	0	X	X	0
1	0	0	0	0	0	1
X	1	0	0	0	1	1
X	X	1	0	1	0	1
X	X	X	1	1	1	1

Implementing Functions with Decoder:

(2/2)

f(Q,X,P) = Σm(0,1,4,6,7) = ∏ M(2,3,5)



3x8 Dec
Q
X
P
A
B
C
0
1
2
3
4
5
6
7
f(Q,X,P)
(m0'·m1'·m4'·m6'·m7)' =m0+m1+m4+m6+m7
(m2+m3+m5)' = m2'·m3'·m5' = M2·M3·M5
m2'·m3'·m5' = M2·M3·M5

(a) Active-high decoder with OR gate.

(b) Active-low decoder with NAND gate.

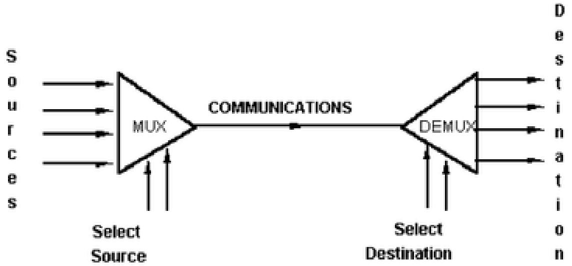
(c) Active-high decoder with NOR gate.

(d) Active-low decoder with AND gate.

BCD Code:

Binary values from 0 to 9 only.

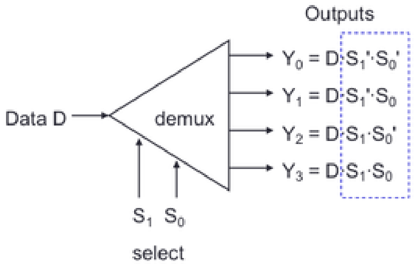
Multiplexers and Demultiplexers:

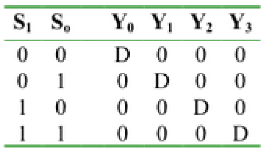


- Helps share a <u>single communication</u> line among a number of devices
- At any time, only <u>one source</u> and <u>one destination</u> can use the communication line.

<u>Demultiplexers:</u>

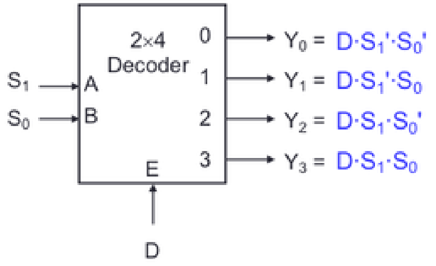
- Directs data from input line to <u>one</u> selected output line
- **Destination selector**


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
select



S_1	S_0	Y_0	Y_1	Y_2	Y_3
0	0	D	0	0	0
0	1	0	D	0	0
1	0	0	0	D	0
1	1	0	0	0	D

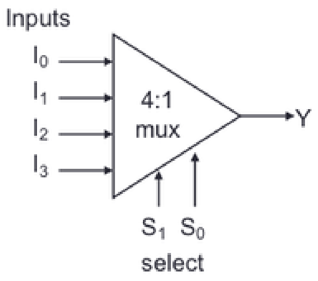
- Identical to a decoder with Enable


2×4 Decoder
S_1
S_0
A
B
E
0
1
2
3
D
$Y_0 = D \cdot S_1' \cdot S_0'$
$Y_1 = D \cdot S_1' \cdot S_0$
$Y_2 = D \cdot S_1 \cdot S_0'$
$Y_3 = D \cdot S_1 \cdot S_0$

<u>Multiplexers:</u>

- Steers one of the <u>2^n inputs</u> to a single output line, using <u>n selection lines</u>
  - input: multiple selection lines and multiple inputs
  - output: one output line
    - sum of the product of <u>data lines</u> and <u>selection lines</u>
      - eg:

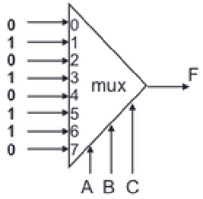
$$Y = I_0 \cdot \underset{0\ 0}{(S_1' \cdot S_0')} + I_1 \cdot \underset{0\ 1}{(S_1' \cdot S_0)} + I_2 \cdot \underset{1\ 0}{(S_1 \cdot S_0')} + I_3 \cdot \underset{1\ 1}{(S_1 \cdot S_0)}$$



- **Source/Data selector**

Implementing Functions with Multiplexer:

- $F(A,B,C) = \Sigma\, m(1,3,5,6)$



This method works because:

$$\text{Output} = I_0 \cdot m_0 + I_1 \cdot m_1 + I_2 \cdot m_2 + I_3 \cdot m_3 + I_4 \cdot m_4 + I_5 \cdot m_5 + I_6 \cdot m_6 + I_7 \cdot m_7$$

Supplying ‘1’ to I_1, I_3, I_5, I_6 , and ‘0’ to the rest:

$$\text{Output} = m_1 + m_3 + m_5 + m_6$$

Sequential Logic (1):

Types:

- Synchronous -> output change at specific time
- Asynchronous -> output change at any time

Classes:

- Bistable: 2 stable states (Latches/Flip Flops)
- Monostable: 1 stable state
- Astable: No stable state

Properties of Latches:

- **Pulse-triggered**
- ON = 1, OFF = 0

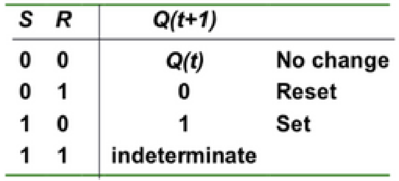
Properties of Flip-Flops:

- **Edge-triggered**
  - Data on inputs is only transferred to flip flop's output only on the triggered edge of clock pulse.
- Positive edge triggered (ON = from 0 to 1, OFF = else)
- Negative edge triggered (ON = from 1 to 0, OFF = else)

Latches

S-R Latch:

- 2 complementary outputs, Q and Q'
  - Q = HIGH (SET)
  - Q = LOW (RESET)



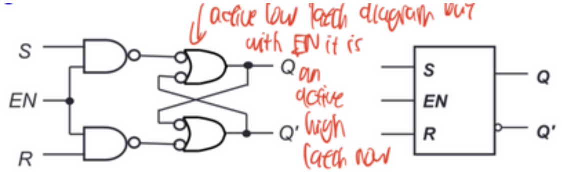
S	R	Q(t+1)	
0	0	Q(t)	No change
0	1	0	Reset
1	0	1	Set
1	1	indeterminate	

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

$S \cdot R = 0$

Gated S-R Latch:

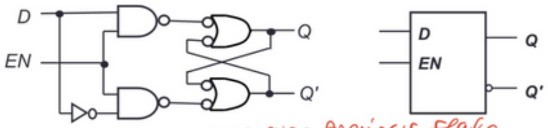
- S-R latch + enable + 2 NAND gates



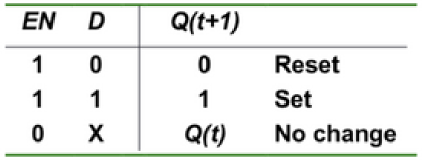
active low latch diagram but with EN it is an active high latch now

Gated D Latch:

- Make input R equal to S'
- D latch eliminates undesirable condition of invalid state in S-R latch.



X when EN = 0, memorise previous state

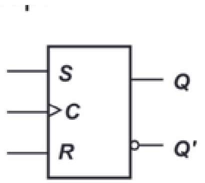


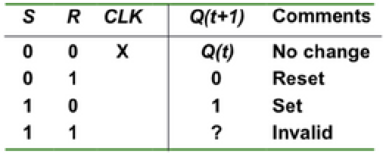
EN	D	Q(t+1)	
1	0	0	Reset
1	1	1	Set
0	X	Q(t)	No change

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

Flip Flops

S-R Flip-Flop:



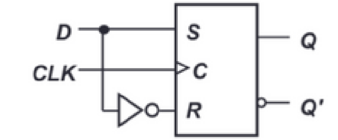


S	R	CLK	Q(t+1)	Comments
0	0	X	Q(t)	No change
0	1		0	Reset
1	0		1	Set
1	1		?	Invalid

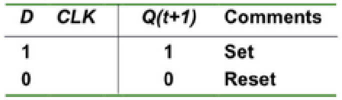
X = irrelevant ("don't care")

↑ = clock transition LOW to HIGH

D Flip-Flop:



A positive edge-triggered D flip-flop formed with an S-R flip-flop.

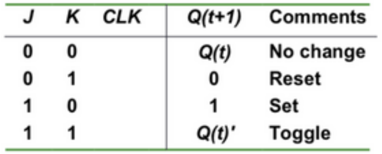


D	CLK	Q(t+1)	Comments
1		1	Set
0		0	Reset

↑ = clock transition LOW to HIGH

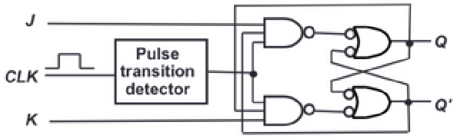
J-K Flip-Flop:

- No invalid state



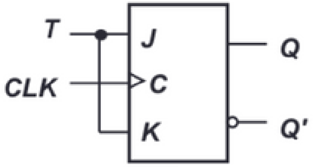
J	K	CLK	Q(t+1)	Comments
0	0		Q(t)	No change
0	1		0	Reset
1	0		1	Set
1	1		Q(t)'	Toggle

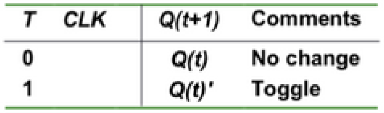
$Q(t+1) = J \cdot Q' + K' \cdot Q$



T Flip-Flop:

- Single input version of J-K flip flop, formed by tying both inputs toget





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

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

↳ XOR

Sequential Logic (2):

Async inputs:

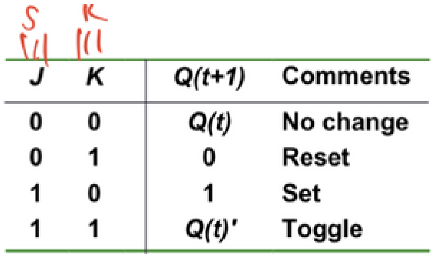
- Inputs independent of clock
- e.g: preset(PRE), clear(CLR), direct set(SD), direct reset(RD)
- When PRE = HIGH, Q is immediately set to **HIGH**
- When CLR = HIGH, Q is immediately set to **LOW**

Self-correcting answer:

- Any unused state is able to transit to a used state after a finite number of cycles.

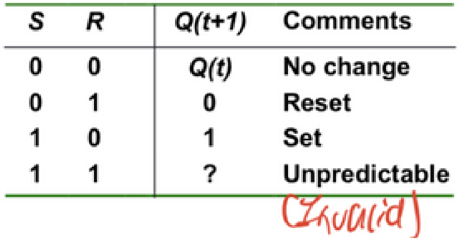
Flip-Flops Tables

Characteristic Tables:



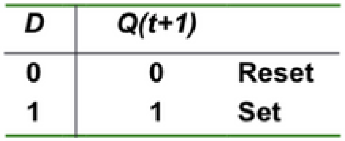
S K

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

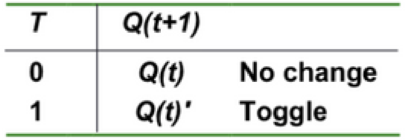


S	*R*	*Q(t+1)*	Comments
0	0	*Q(t)*	No change
0	1	0	Reset
1	0	1	Set
1	1	?	Unpredictable

(Invalid)

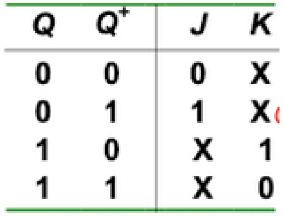


D	*Q(t+1)*	
0	0	Reset
1	1	Set



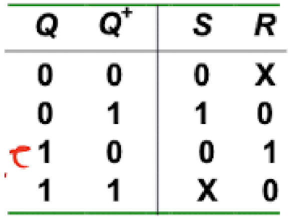
T	*Q(t+1)*	
0	*Q(t)*	No change
1	*Q(t)'*	Toggle

Excitation Tables:



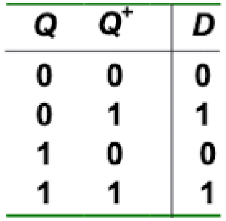
Q	Q^+	*J*	*K*
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

JK Flip-flop



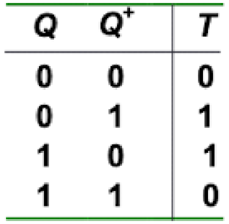
Q	Q^+	*S*	*R*
0	0	0	X
0	1	1	0
1	0	0	1
1	1	X	0

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

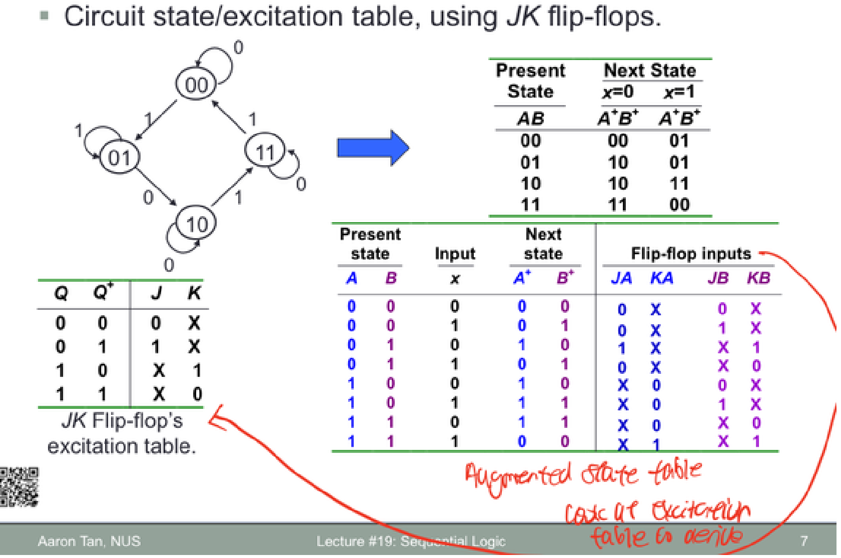
T Flip-flop

Sequential Logic (3):

Sequential Circuit Design:

(From state diagram -> logic circuit)

- Algorithm:
  - Convert state diagram -> state table
    - Fill in flip-flop inputs based on excitation table
  - Get simplified expressions for flip flop state equation **using K-maps**
    - Simplify state equations if possible (ie. A'B + AB' = A⊕B)
  - Draw the circuit
- **Note: Unused states use don't care(d) for input**



- Circuit state/excitation table, using *JK* flip-flops.

Present State	Next State	
	x=0	x=1
AB	A^+B^+	A^+B^+
00	00	01
01	10	01
10	10	11
11	11	00

Q	Q^+	*J*	*K*
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

JK Flip-flop's excitation table.

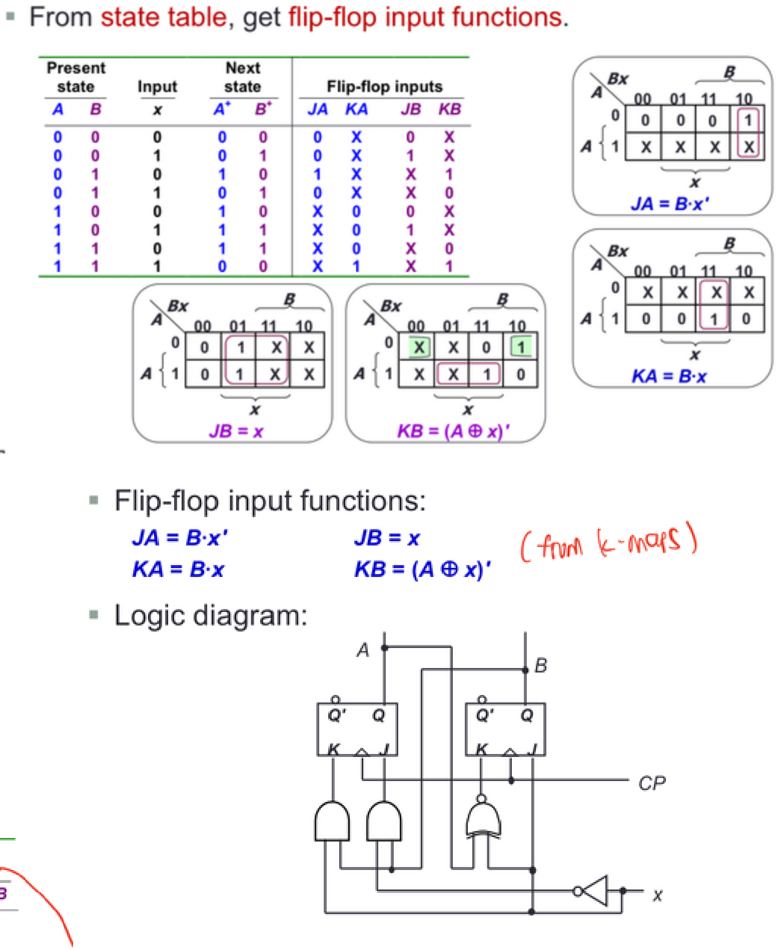
Present state		Input	Next state		Flip-flop inputs			
A	*B*	*x*	A^+	B^+	*JA*	*KA*	*JB*	*KB*
0	0	0	0	0	0	X	0	X
0	0	1	0	1	0	X	1	X
0	1	0	1	0	1	X	X	1
0	1	1	0	1	0	X	X	0
1	0	0	1	0	X	0	0	X
1	0	1	1	1	X	0	1	X
1	1	0	1	1	X	0	X	0
1	1	1	0	0	X	1	X	1

Augmented state table

Look at excitation table to derive

- **Note:** <u>self-correcting</u>: any unused state can transition to a used state after a finite number of cycles

Cont.



- From state table, get flip-flop input functions.

Present state		Input	Next state		Flip-flop inputs			
A	*B*	*x*	A^+	B^+	*JA*	*KA*	*JB*	*KB*
0	0	0	0	0	0	X	0	X
0	0	1	0	1	0	X	1	X
0	1	0	1	0	1	X	X	1
0	1	1	0	1	0	X	X	0
1	0	0	1	0	X	0	0	X
1	0	1	1	1	X	0	1	X
1	1	0	1	1	X	0	X	0
1	1	1	0	0	X	1	X	1

A \ *Bx*	00	01	11	10
0	0	0	0	1
1	X	X	X	X

JA = B·x'

A \ *Bx*	00	01	11	10
0	X	X	X	X
1	0	0	1	0

KA = B·x

A \ *Bx*	00	01	11	10
0	0	1	X	X
1	0	1	X	X

JB = x

A \ *Bx*	00	01	11	10
0	X	X	0	1
1	X	X	1	0

KB = (A ⊕ x)'

- Flip-flop input functions:

  JA = B·x' *JB = x* (from k-maps)

  KA = B·x *KB = (A ⊕ x)'*

- Logic diagram:

Sequential Circuit Analysis:

(From logic circuit-> state diagram)

- Algorithm:
  - Obtain flip flop functions from circuit
  - Fill in state table using characteristic table **(output of function is always based on present state value)**
  - Draw state diagram

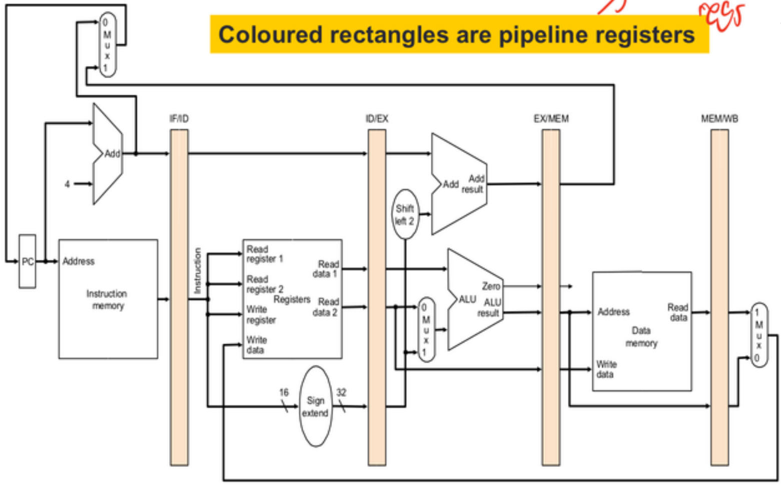
MIPS Pipelining:

- Pipelined implementation:
  - One cycle per pipeline stage
  - Data required for each stage needs to be stored separately

MIPS Pipeline:

- **Stages:**
  - IF - Instruction Fetch
  - ID - Instruction Decode
  - EX - Execute operation
  - MEM - Access operand in memory (lw/sw)
  - WB - Write back result to register (R/I format instructions)
- **Pipeline Registers:**
  - **32-bit** registers
  - IF/ID - between IF and ID
  - ID/EX - between ID and EX
  - EX/MEM - between EX and MEM
  - MEM/WB - between MEM and WB

3. MIPS Pipeline: Datapath (3/3)


Coloured rectangles are pipeline registers

collection of regs

Cont.

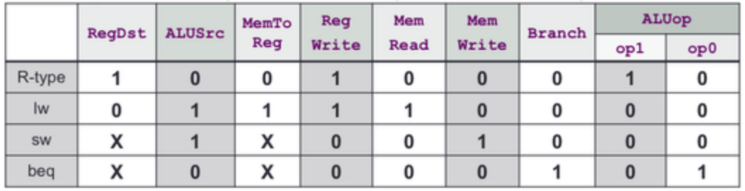
- **IF/ID:**
  - Stores:
    - Instruction read from InstructionMemory[PC]
    - PC + 4 value
- ID/EX:
  - Stores:
    - Data values read from register file
    - 32-bit immediate value
    - PC + 4
    - Write Register number
- EX/MEM:
  - Stores:
    - PC + 4 + (Imm * 4) value
    - ALU result
    - isZero? control signal
    - Data Read 2(RD2) from register file
    - Write Register number
- MEM/WB:
  - Stores:
    - ALU result
    - Memory read data
    - Write Register number

Pipeline Control

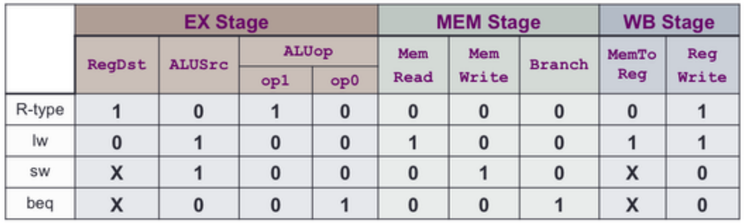
- Same control signals as single-cycle datapath
- Each control signal belongs to a particular pipeline stage

Grouping:

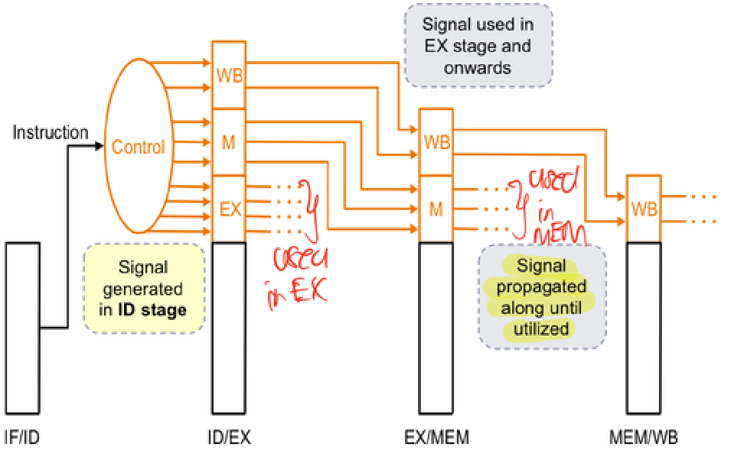
- Group control signals according to pipeline stage



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



	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


Signal used in EX stage and onwards

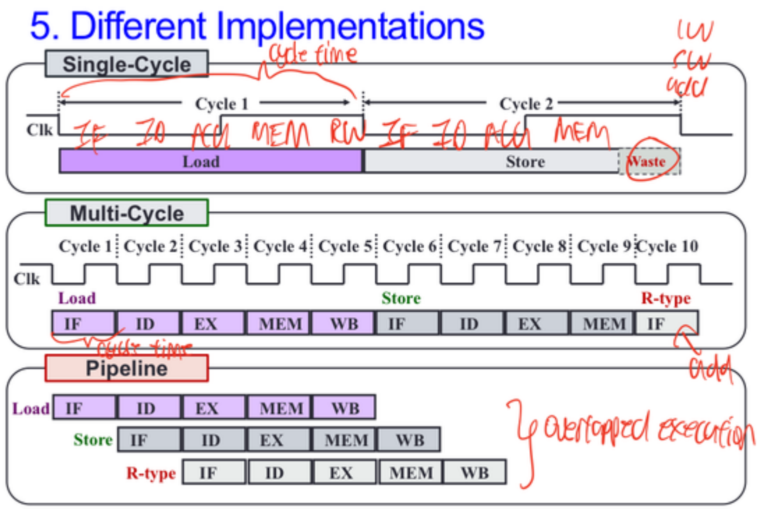
Signal generated in **ID stage**

Signal propagated along until utilized

used in EX

used in MEM

Pipeline Implementation



5. Different Implementations

Performance Comparison

let

- CT = cycle time,
- T_k = time for operation in stage k,
- N = number of stages

single-cycle

- cycle time, $CT_{seq} = \max(\sum_{K=1}^{N} T_k)$
- execution time for I instructions = $I \times CT_{seq}$

multi-cycle

- cycle time, $CT_{multi} = \max(T_k)$
- execution time for I instructions = $I \times \text{Average CPI} \times CT_{multi}$
  - average CPI used since each instruction takes diff number of cycles

pipelining

- cycle time, $CT_{pipeline} = \max(T_k) + T_d$
  - where T_d = overhead for pipelining (e.g. pipeline register)
- cycles needed for I instructions = $(I + N - 1)$
  - $N - 1$ cycles to fill up pipeline
- execution time for I instructions = $(I + N - 1) \times CT_{pipeline}$

Ideal Speedup

- assumptions:
  - every stage takes the same amt of time ⇒ $\sum_{k=1}^{N} T_k = N \times T_1$
  - no pipeline overhead ⇒ $T_d = 0$
  - $I >> N$

$$Speedup_{pipeline} = \frac{Time_{seq}}{Time_{pipeline}} = N$$

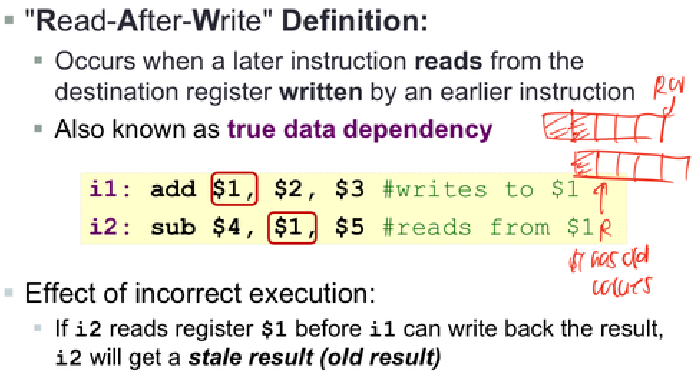
Pipeline Hazards:

Structural Hazards:

- Solve read/write conflict in MEM
- Resolution: Split WB cycle into half
  - First half is to write into register
  - Second half to read from register

Data Hazards:

- **R**ead-after-Write (RAW)



"Read-After-Write" **Definition:**

- Occurs when a later instruction **reads** from the destination register **written** by an earlier instruction
- Also known as **true data dependency**

```
i1: add $1, $2, $3 #writes to $1
i2: sub $4, $1, $5 #reads from $1
```

Effect of incorrect execution:

- If i2 reads register $1 before i1 can write back the result, i2 will get a ***stale result (old result)***

- Write-after-Read (WAR)
- Write-after-Write (WAW)
  - WAR and WAW do not cause any pipeline hazards

Shortcut (No need to draw)

Step 1: Calculate cycles required in an ideal pipeline scenario (Total = I + N – 1), where I = numInstructions, N = numStages in pipeline

Step 2: Add delays according to the following;

Control Dependency:

- **w/o early branch & w/o branch prediction: +3**
- **w early branch & w/o branch prediction: +1**
- **w early branch & w branch prediction:**
  - Correct prediction: +0
  - Wrong prediction: +1
- **w/o early branch & w branch prediction:**
  - Correct prediction: +0
  - Wrong prediction: +3
- **Jump instructions: +1**

Data Dependency:

- **without forwarding: +2**
- **with forwarding: +0**
  - Except the following cases:
    - when current instruction is beq/bne and one of the operand is being written to in the last instruction: +1

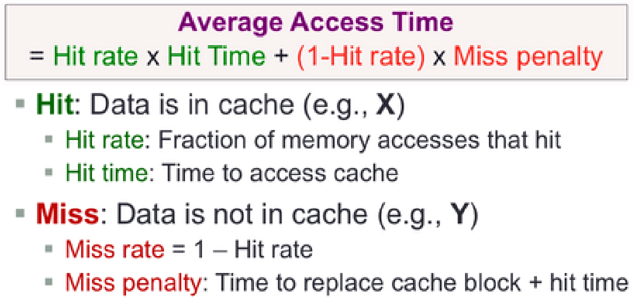
```
add  $s2, $zero, $zero   #inst B
bne  $s2, $zero, done    #inst C
```

    - when previous instruction is lw/sw and need to use the written register in current instruction: +1

```
lw   $s4, 0($t4)     # I11
sub  $s3, $s3, $s4   # I12
```

Cache:

Average Access Time



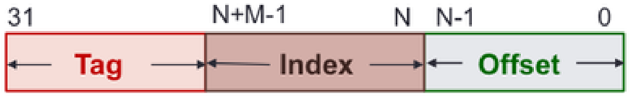
Average Access Time

= Hit rate x Hit Time + (1-Hit rate) x Miss penalty

- **Hit**: Data is in cache (e.g., **X**)
  - Hit rate: Fraction of memory accesses that hit
  - Hit time: Time to access cache
- **Miss**: Data is not in cache (e.g., **Y**)
  - Miss rate = 1 – Hit rate
  - Miss penalty: Time to replace cache block + hit time

- **1 KB = 2^10 bytes**
- **1 MB = 2^20 bytes**
- **1 GB = 2^30 bytes**

Direct map:



- Given block size = 2^N,

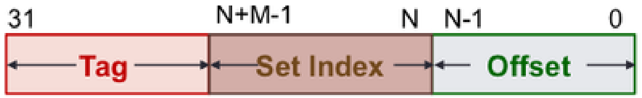
#blocks = Cache size / 2^N

Offset bits = N

Block Index = M (where 2^M = #blocks)

Tag bits = 32 – M – N

K-way set associative:



- Given block size = 2^N,

#blocks = Cache size / 2^N * #sets

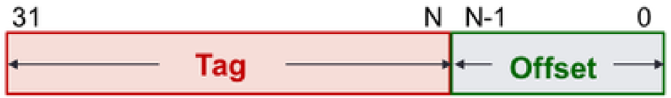
Offset bits = N

Set index = M (where 2^M = #sets)

Tag bits = 32 – M – N

Cont.

Fully Associative:



- Given block size = 2^N,

#blocks = Cache size / 2^N

Offset bits = N

Tag bits = 32 – N