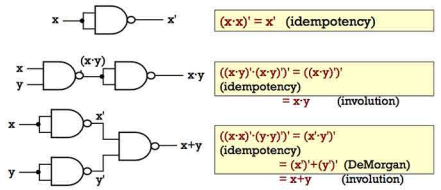
Identity laws	
$A + 0 = 0 + A = A$	$A \cdot 1 = 1 \cdot A = A$
Inverse/complement laws	
$A + A' = 1$	$A \cdot A' = 0$
Commutative laws	
$A + B = B + A$	$A \cdot B = B \cdot A$
Associative laws	
$A + (B + C) = (A + B) + C$	$A \cdot (B \cdot C) = (A \cdot B) \cdot C$
Distributive laws	
$A \cdot (B + C) = (A \cdot B) + (A \cdot C)$	$A + (B \cdot C) = (A + B) \cdot (A + C)$
Idempotency	
$X + X = X$	$X \cdot X = X$
One element / Zero element	
$X + 1 = 1$	$X \cdot 0 = 0$
Involution	
$(X')' = X$	
Absorption	
$X + X \cdot Y = X$	$X \cdot (X + Y) = X$
Absorption (variant)	
$X + X' \cdot Y = X + Y$	$X \cdot (X' + Y) = X \cdot Y$
DeMorgans' (can be generalised to more than 2 variables)	
$(X + Y)' = X' \cdot Y'$	$(X \cdot Y)' = X' + Y'$
Consensus	
$X \cdot Y + X' \cdot Z + Y \cdot Z = X \cdot Y + X' \cdot Z$	$(X+Y) \cdot (X'+Z) \cdot (Y+Z) = (X+Y) \cdot (X'+Z)$

Ravi Suppiah

sum of minterms of F = product of maxterms of F
sum of minterms of F' = negation of product of maxterms of F
product of maxterms of F' = negation of sum of minterms of F

Self-Sufficient NAND (SOP)

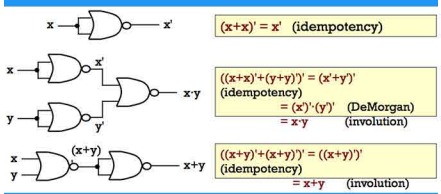


$(x \cdot x)' = x'$ (idempotency)

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

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

Self-Sufficient NOR (POS)

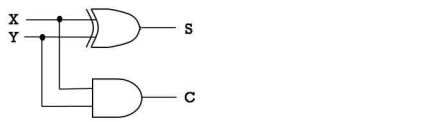


$(x + x)' = x'$ (idempotency)

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

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

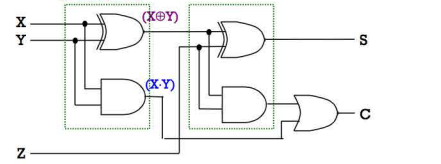
HALF ADDER



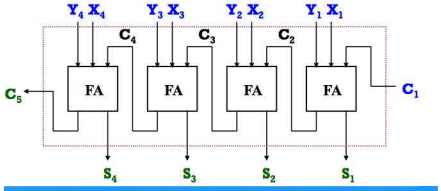
FULL ADDER

$$C = X \cdot Y + X \cdot Z + Y \cdot Z$$
$$= X \cdot Y + (X + Y) \cdot Z$$
$$= X \cdot Y + ((X \oplus Y) + X \cdot Y) \cdot Z$$
$$= X \cdot Y + (X \oplus Y) \cdot Z + X \cdot Y \cdot Z$$
$$= X \cdot Y + (X \oplus Y) \cdot Z$$

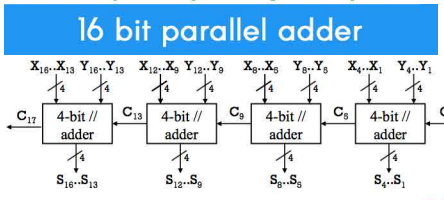
$$S = X' \cdot Y' \cdot Z + X' \cdot Y \cdot Z' + X \cdot Y' \cdot Z' + X \cdot Y \cdot Z$$
$$= X' \cdot (Y' \cdot Z + Y \cdot Z') + X \cdot (Y' \cdot Z' + Y \cdot Z)$$
$$= X' \cdot (Y \oplus Z) + X \cdot (Y \oplus Z)'$$
$$= X \oplus (Y \oplus Z)$$



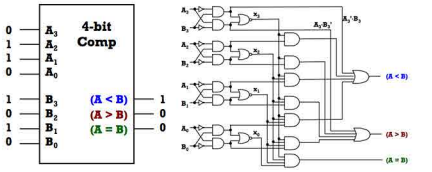
4 bit parallel adder



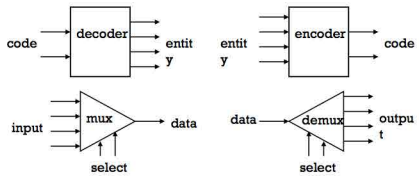
16 bit parallel adder



COMPARATOR



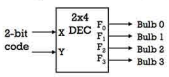
MSI



DECODERS

Convert binary information from n input lines to (a maximum of) 2n output lines

- Example: If codes 00, 01, 10, 11 are used to identify four light bulbs, we may use a 2-bit decoder.



- This is a 2×4 decoder which selects an output line based on the 2-bit code supplied.
- Truth table:

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

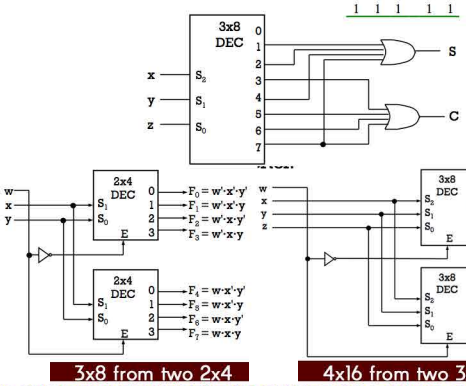
Lecture #18: MSI Components — Ravi Suppiah

- Example: Full adder

$S(x, y, z) = \Sigma\, m(1,2,4,7)$

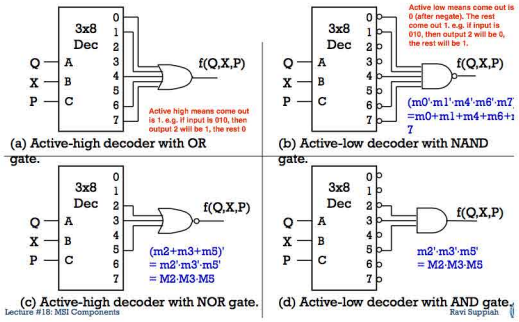
$C(x, y, z) = \Sigma\, m(3,5,6,7)$

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



2. DECODERS: IMPLEMENTING FUNCTIONS REVISIT (2/2)

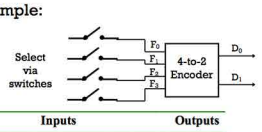
$f(Q,X,P) = \Sigma m(0,1,4,6,7) = \prod M(2,3,5)$



Lecture #18: MSI Components — Ravi Suppiah

ENCODERS

- Encoding is the converse of decoding.
- 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.
- Contains 2^n (or fewer) input lines and n output lines.
- Implemented with OR gates.
- Example:

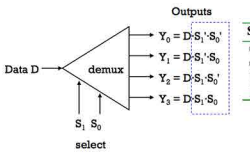


Inputs								Outputs		
D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	x	y	z
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

Inputs				Outputs		
D_0	D_1	D_2	D_3	f	g	V valid
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

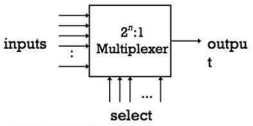
- Given an input line and a set of selection lines, a demultiplexer directs data from the input to one selected output line.
- Example: 1-to-4 demultiplexer.



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

MULTIPLEXER

- A multiplexer is a device that has
  - A number of input lines
  - A number of selection lines
  - One output line
- It steers one of 2^n inputs to a single output line, using n selection lines. Also known as a *data selector*.



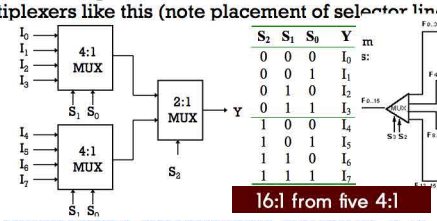
- Truth table for a 4-to-1 multiplexer:

I_0	I_1	I_2	I_3	S_1	S_0	Y
d_0	d_1	d_2	d_3	0	0	d_0
d_0	d_1	d_2	d_3	0	1	d_1
d_0	d_1	d_2	d_3	1	0	d_2
d_0	d_1	d_2	d_3	1	1	d_3

S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

5. CONSTRUCTING LARGER MULTIPLEXERS (1/4)

- Larger multiplexers can be constructed from smaller ones.
- An 8-to-1 multiplexer can be constructed from smaller multiplexers like this (note placement of selector lines):



5. MULTIPLEXERS: IMPLEMENTING FUNCTIONS (2/3)

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

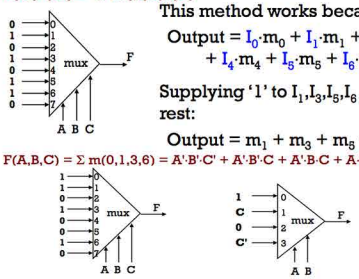
This method works because:

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

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

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

$F(A,B,C) = \Sigma\, m(0,1,3,6) = A' \cdot B' \cdot C' + A' \cdot B \cdot C + A \cdot B \cdot C'$



MISCELLANEOUS

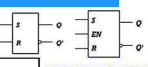
Big-endian:	Little-endian:
Most significant byte stored in lowest address.	Least significant byte stored in lowest address.
Example: IBM 360/370, Motorola 68000, MIPS (Silicon Graphics), SPARC.	Example: Intel 80x86, DEC VAX, DEC Alpha.
Example: **0xDE AD BE EF** Stored as: 0 DE, 1 AD, 2 BE, 3 EF	Example: **0xDE AD BE EF** Stored as: 0 EF, 1 BE, 2 AD, 3 DE

Category	Instruction	Example	Meaning	Comments
	add	`add $s1, $s2, $s3`	$s1 = $s2 + $s3	Three operands; data in registers
Arithmetic	subtract	`sub $s1, $s2, $s3`	$s1 = $s2 - $s3	Three operands; data in registers
	add immediate	`addi $s1, $s2, 100`	$s1 = $s2 + 100	Used to add constants
	load w ord	`lw $s1, 100($s2)`	$s1 = Memory[$s2 + 100]	Word from memory to register
	store w ord	`sw $s1, 100($s2)`	Memory[$s2 + 100] = $s1	Word from register to memory
Data transfer	load byte	`lb $s1, 100($s2)`	$s1 = Memory[$s2 + 100]	Byte from memory to register
	store byte	`sb $s1, 100($s2)`	Memory[$s2 + 100] = $s1	Byte from register to memory
	load upper immediate	`lui $s1, 100`	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
	branch on equal	`beq $s1, $s2, 25`	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1, $s2, 25`	if ($s1 != $s2) go to PC + 4 + 100	Not equal test; PC-relative
onditional anch	set on less than	`slt $s1, $s2, $s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for beq, bne
	set less than immediate	`slti $s1, $s2, 100`	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare less than constant
	jump	`j 2500`	go to 10000	Jump to target address
ncondi- nal jump	jump register	`jr $ra`	go to $ra	For sw itch, procedure return
	jump and link	`jal 2500`	$ra = PC + 4; go to 10000	For procedure call

Circuit -> State eqn -> State table -> State Diagram | -> Charac table | <- Excita table

SR LATCH

Characteristic table for active-high input *S-R* latch: Active High (NOR)

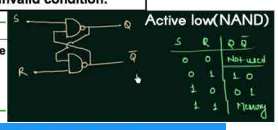


S	R	Q	Q'	
0	0	NC	NC	No change. Latch remained in present state.
1	0	1	0	Latch SET.
0	1	0	1	Latch RESET.
1	1	0	0	Invalid condition.

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

$S \cdot R = 0$

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



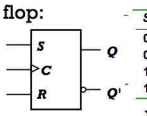
GATED D LATCH

Characteristic table: When *EN*=1, $Q(t+1) = D$

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

SR FLIP FLOP

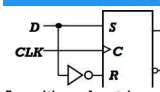
Characteristic table of positive edge-triggered *S-R* flip-flop:



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

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

↑ = clock transition LOW to HIGH

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

Characteristic table:

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	J	K	Q(t+1)
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

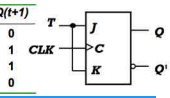
$Q(t+1) = ?$

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

Characteristic table:

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

Q	T	Q(t+1)
0	0	0
0	1	1
1	0	1
1	1	0



STATE EQNS

State equations:

$A^+ = A \cdot x + B \cdot x$

$B^+ = A' \cdot x$

STATE TABLE

m flip-flops and *n* inputs → 2^{m+n} rows.

State equations: $A^+ = A \cdot x + B \cdot x$, $B^+ = A' \cdot x$

Output function: $y = (A + B) \cdot x'$

Present State A	Present State B	Input x	Next State A+	Next State B+	Output y
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	1	1	0
1	0	0	0	0	1
1	0	1	1	0	0
1	1	0	0	0	1
1	1	1	1	0	0

FULL TABLE VS COMPACT

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

EXCITATION TABLE

Excitation tables: given the required transition from present state to next state, determine the flip-flop input(s)

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

Same as jk just that dont allow 1-1

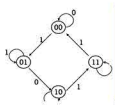
Q	Q+	D
0	0	0
0	1	1
1	0	0
1	1	1

D Flip-flop

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

T Flip-flop

Q = 0 means same, Q = 1 means toggle



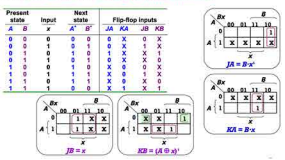
Questions:

How many flip-flops are needed? 4 states = 2bits = 2 flip flops

How many input variable are = 1 input var

6.4 DESIGN: EXAMPLE #1 (4/5)

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



PIPELINE

- **IF**: Instruction Fetch
- **ID**: Instruction Decode and Register Read
- **EX**: Execute an operation or calculate an address
- **MEM**: Access an operand in data memory
- **WB**: Write back the result into a register

At the beginning of a cycle IF/ID register supplies:	At the end of a cycle ID/EX receives:
❖ Register numbers for reading two registers ❖ 16-bit offset to be sign-extended to 32-bit	❖ Data values read from register file ❖ 32-bit immediate value ❖ PC + 4

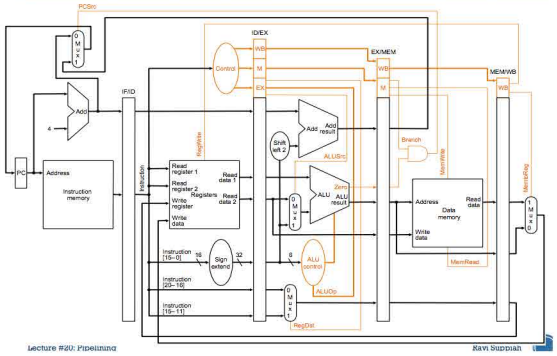
At the beginning of a cycle ID/EX register supplies:	At the end of a cycle EX/MEM receives:
❖ Data values read from register file ❖ 32-bit immediate value ❖ PC + 4	❖ (PC + 4) + (Immediate x 4) ❖ ALU result ❖ isZero? signal ❖ Data Read 2 from register file

At the beginning of a cycle EX/MEM register supplies:	At the end of a cycle MEM/WB receives:
❖ (PC + 4) + (Immediate x 4) ❖ ALU result ❖ isZero? signal ❖ Data Read 2 from register file	❖ ALU result ❖ Memory read data

At the beginning of a cycle MEM/WB register supplies:	At the end of a cycle
❖ ALU result ❖ Memory read data	❖ Result is written back to register file (if applicable) ❖ There is a bug here.....

	RegDst	ALUSrc	MemTo Reg	Reg Write	Mem Read	Mem Write	Branch	ALUop op1	ALUop 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 op1	ALUop op0	Mem Read	Mem Write	Branch	MemTo Reg	Reg Write
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



SINGLE CYCLE

Cycle Time

- Choose the longest total time = **8ns**

To execute **100 instructions:**

- **100 × 8ns = 800ns**

MULTI CYCLE

Cycle Time:

- $CT_{pipeline} = \max(T_k) + T_d$
- $\max(T_k)$ = longest time among the N stages
- T_d = Overhead for pipelining, e.g. pipeline register

Cycles needed for **I** instructions:

- I + N − 1 N = number of stage
- N − 1 is the cycles wasted in filling up the pipeline

Total Time needed for **I** instructions :

- $Time_{pipeline} = Cycle \times CT_{pipeline}$

Cycle Time $= (I + N - 1) \times (\max(T_k) + T_d)$

- assume pipeline register delay of **0.5ns**
- longest stage time + overhead = **2 + 0.5 = 2.5ns**

To execute **100 instructions:**

- **(100 + 5 − 1) × 2.5ns = 260ns**

Move branch decision calculation to earlier pipeline stage

- **Early Branch Resolution**

Guess the outcome before it is produced

- **Branch Prediction**

Do something useful while waiting for the outcome

- **Delayed Branching**

CACHE

SRAM	DRAM
6 transistors per memory cell	1 transistor per memory cell
→ **Low density**	→ **High density**
Fast access latency of 0.5 – 5 ns	**Slow access** latency of 50-70ns
More costly	Less costly
Uses flip-flops	Used in main memory

- **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

HIT

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

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

Offset = **N bits** index = which line in cache block

Index = **M bits** offset = which position in 1 line of cache block

Tag = **32 – (N + M) bits**

WRITE POLICY

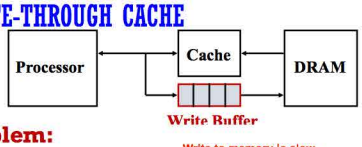
Solution 1: Write-through cache

- Write data both to cache and to main memory

Solution 2: Write-back cache

- Only write to cache
- Write to main memory only when cache block is replaced (evicted)

8. WRITE-THROUGH CACHE



- **Problem:** Write to memory is slow.
  - Write will operate at the speed of main memory!
- **Solution:**
  - Put a write buffer between cache and main memory
    - Processor: writes data to cache + write buffer
    - Memory controller: write contents of the buffer to memory

8. WRITE-BACK CACHE

- **Problem:**
  - Quite wasteful if we write back everv evicted cache blocks
- **Solution:**
  - Add an additional bit (**Dirty bit**) to each cache block
  - Write operation will change dirty bit to 1
    - Only cache block is updated, no write to memory
  - When a cache block is replaced:
    - Only write back to memory if dirty bit is 1

CACHE MISS

Write Miss option 1: **Write allocate**

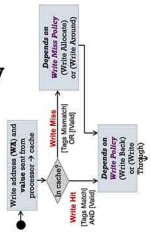
- Load the complete block into cache
- Change only the required word in cache
- Write to main memory depends on write policy

Write Miss option 2: **Write around**

- Do not load the block to cache
- Write directly to **main memory only**

Memory address (in decimal)	Memory address (in hexadecimal)	Hit (H) or Miss (M)?	(For reference)
4	0x4	M	0000 ... 0000 0100
92	0x5C	M	0000 ... 0101 1100

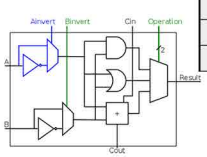
Index	Tag value	Word 0	Word 1
0			
1	0	M[4] M[5] write both	M[6]



CONTROL SIGNAL

ALUcontrol Ainvert	ALUcontrol Binvert	ALUcontrol Operation	Function
0	0	00	AND
0	0	01	OR
0	0	10	add
0	1	10	subtract
0	1	11	slt
1	1	00	NOR

lw/sw= 0010 beq= 0110



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

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

Exponent = 2 + 127 = 129 = 10000001_2 2 bocos raise to exponent 2, 127 becos excess-127 system

mantissa is just the after the dot, copy paste, and add 0 behind until 23 digits

1	10000001	10100000000000000000000
sign	exponent (excess-127)	mantissa